Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Paralelné výpočtové spracovanie spektroskopických dát

Bakalárska práca

2018
Adrián Kocúrek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Paralelné výpočtové spracovanie spektroskopických dát

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	Mgr. Pavel Petrovič, PhD.
Konzultant:	Mgr. Peter Čermák, PhD.

Bratislava, 2018
Adrián Kocúrek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Adrián Kocúrek
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Paralelné výpočtové spracovanie spektroskopických dát
Parallel Computational Processing of Spectroscopic Data

Anotácia: Na Katedre experimentálnej fyziky FMFI UK prebieha výskum zaoberajúci sa spracovaním spektier molekúl, za účelom ich presnej detekcie a štúdia ich vnútornej štruktúry. Súčasťou tohto výskumu je aj vývoj software umožňujúceho efektívne a presné spracovanie experimentálnych dát. V súčasnosti existujúca verzia programu funguje v prostredí LabVIEW s niektorými integrovanými parciálnymi výpočtami v C++. Úlohou študentaje zoznámiť sa s výpočtovým modelom používaným vo výskume (ide hlavne o neanalytické funkcie ako napr. Faddeevovej funkcia) a zefektívniť jeho fungovanie paralelizáciou výpočtov s využitím viacjadrových procesorov alebo procesorových jednotiek na grafickej karte (GPU). Práca by mala poskytnúťporovnanie efektivity rôznych modifikácií spôsobu paralelizácie.

Literatúra: J. Cheng, M. Grossman, T. McKercher: Professional CUDA C Programming, Wrox, 2014.
P. Čermák: The Multifit Program for Spectroscopic Data, In-Depth Solutions with Graphical System Design in Eastern Europe, National Instruments, 2012.
J. Tennyson et.al.: Recommended isolated-line profile for representing high-resolution spectroscopic transitions, IUPAC Technical Report, Pure Appl. Chem. 2014; 86(12): 1931–1943, De Gruyter, 2014.

Vedúci: Mgr. Pavel Petrovič, PhD.
Konzultant: Mgr. Peter Čermák, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 05.10.2017

Dátum schválenia: 31.01.2018 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Ďakujem pánovi Mgr. Pavlovi Petrovičovi, PhD. za ochotu odpovedať vždy a rýchlo. Bez ohľadu na to, či som mail poslal o 1-ej ráno, alebo na obed. Ďalej by som chcel poďakovať mojej rodine, ktorá mi poskytla morálnu podporu pri každom kroku písania tejto práce. A nakoniec aj pánovi Mgr. Petrovi Čermákovi, PhD., ktorý nielen poskytol námet tejto práce, ale aj aktívne spolupracoval pri väčšine jej vyhotovenia.

Abstrakt

KOCÚREK, Adrián: Paralelné výpočtové spracovanie spektroskopických dát [Bakalárska práca]. Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra informatiky. Školiteľ: Mgr. Pavel Petrovič, PhD. Bratislava FMFI UK, 2018.

Cieľom tejto práce bolo zoznámiť sa s výpočtovým modelom používaným v optickej spektroskopii atómov a molekúl a jeho zefektívnenie paralelizáciou na grafickej karte (GPU). Práca je členená na 5 kapitol. V prvej kapitole sa venujeme riešeniu paralelných problémov pomocou grafickej karty vrátane stručného opisu práce s jazykom CUDA. Druhá kapitola sa zaoberá zjednodušeným úvodom do problematiky optickej spektroskopie a popisu aktuálneho postupu na generovanie optických spektier použitím programu Peaks Synthesizer, vytvoreného v programovacom jazyku LabVIEW a C++. Tretia kapitola rozoberá použitý algoritmus, možnosti jeho paralelizácie vrátane aktuálne dostupných riešení (knižnica CERNlib_CUDA). V štvrtej kapitole sa sústredíme na popis nášho riešenia a jeho implementáciu. V poslednej, piatej kapitole, rozoberáme presnosť a rýchlosť nášho algoritmu v porovnaní s pôvodnou verziou programu bežiacou na CPU a paralelizovanou verziou, použitím knižnice CERNlib_CUDA. 40 strán.

Kľúčové slová: CUDA, LabVIEW, GPU, spektroskopia

Abstract

KOCÚREK, Adrián: Parallel Computational Processing of Spectroscopic Data [Bachelor thesis]. Comenius University, Bratislava. Faculty of Mathematics, Physics and Informatics; Department of Computer Science. Supervisor: Mgr. Pavel Petrovič, PhD. Bratislava FMFI UK, 2018. 40 pages.

The goal of this work was to get used to the computation model used in optical spectroscopy of atoms and molecules and its optimisation by paralelising on the GPU. The word is divided into 5 chapters. In the first chapter, we focused on the solutions of paralel problems by using a graphics card. We also focused on a basic introduction of the CUDA language. We focused the second chapter on a brief introduction to the field of optical spectroscopy and a description of the current way of generating optical spectra using Peaks Synthesizer, a program created using LabVIEW and C++. The third chapter focused on the algorithm used therein, options for its paralelisation, include publicly available methods (CERNlib_CUDA library). Implementation of the software was the focus of the fourth chapter and in the fifth, we described the accuracy and speed of our algorith in comparison to the original program running on CPU only and a paralelised version which uses the CERNlib_CUDA library.

Keywords: CUDA, LabVIEW, GPU, spectroscopy

Obsah

Úvod **1**

1 Riešenie paralelných problémov pomocou GPU **3**
1.1 Začiatky vývoja . . . 3
1.2 GPU . . . 4
1.2.1 Architektúra GPU . . . 4
1.2.2 Zameranie GPU . . . 4
1.3 Praktická aplikácia GPGPU . . . 5
1.4 API . . . 5
1.4.1 OpenCL . . . 6
1.4.2 Nvidia CUDA . . . 6
1.5 Programovanie s CUDA . . . 6
1.5.1 Vlákna . . . 6
1.5.2 Synchronizácia . . . 7
1.5.3 Pamäť . . . 7

2 Optické spektrá látok a program Peaks Synthesizer **9**
2.1 Optická spektroskopia . . . 9
2.1.1 Spektrum molekuly . . . 10
2.2 Profil spektrálnej čiary . . . 11
2.2.1 Gaussova funkcia . . . 11
2.2.2 Lorentz-Cauchyho rozdelenie . . . 11
2.2.3 Voigtova funkcia . . . 11
2.2.4 Hartmann-Tranova funkcia (HTP) . . . 12
2.2.5 Chybové funkcie . . . 12
2.2.6 Faddejevovej funkcia . . . 12
2.3 Peaks Synthesizer . . . 13
2.4 LabVIEW . . . 13
2.4.1 Vizuálne programovanie . . . 13
2.4.2 Využitie LabVIEW . . . 13
2.5 Spracovanie dát . . . 14

2.5.1 Metódy výpočtu hodnôt 14

3 Návrh implementácie systému 16

3.1 Počiatočný stav 16
3.2 Algoritmus 16
3.3 Návrh riešenia 17
3.4 Príprava zázemia 17
3.5 Výber terminálu 17

4 Implementácia programu 19

4.1 Príprava LabVIEW 19
4.1.1 Verzia pre Linux 19
4.1.2 Problém kompatibility 19
4.1.3 Licenčné problémy 20
4.1.4 Zmeny v postupe 20
4.2 Riešenie problémov 20
4.2.1 Výhody a nevýhody 20
4.2.2 Tvorba nového rozhrania 21
4.2.3 Struct hack 21
4.3 Výmena knižnice Cerf 22
4.3.1 CERNlib_CUDA 22
4.4 Hlavná implementácia 22
4.4.1 Úprava obslužnej knižnice 23
4.4.2 PeaksGPU 23
4.5 linesAdvCUDA 24
4.5.1 PeakAdvCUDA - rozhranie knižnice 24
4.5.2 PeakAdvCUDA - výpočty 25
4.5.3 Manažment zdrojov GPU 25
4.6 Výpočtové jadrá 26
4.6.1 Lorentz 26
4.6.2 Gauss 27
4.6.3 Voigt 27
4.6.4 HTP 28
4.7 FuncAdvCUDA 30
4.7.1 Réžia súbežných výpočtov 30
4.8 Cerf_CUDA 31
4.8.1 Cerf_CUDA - implementácia 32
4.8.2 Cerf_CUDA - testovanie 32

5 Výsledky **34**
5.1 Metodológia 34
5.2 Porovnania presnosti 34
5.2.1 Korektnosť 34
5.2.2 Odchýlka 35
5.3 Záťažové porovnania 36
5.3.1 Prípad I 36
5.3.2 Prípad II 37
5.3.3 Prípad III 38
5.4 Zhrnutie 38

Záver **41**

Appendix A **43**

Zoznam obrázkov

1.1 Všeobecná schéma komponentov CPU a GPU 4

2.1 Príklad spektra so zvýraznenými zložkami 10
2.2 Ukážka prostredia LabVIEW 14

4.1 Priemerná dĺžka výpočtu (ms) náhodných komplexných čísel 32

5.1 Grafy totožnej spektrálnej čiary vygenerované pomocou funkcií Cerf a Cerf_CUDA 35
5.2 Graf odchýlky pre hodnoty z Obr.5.1 36
5.3 Chyba reálnej časti na intervale <0,1> 37
5.4 Chyba imaginárnej časti na intervale <0,1> 38
5.5 Priemerná dĺžka výpočtu (ms) v závislosti od počtu bodov generovanej HTP spektrálnej čiary 39
5.6 Priemerná dĺžka výpočtu (ms) v závislosti od počtu bodov pre 1000 spektrálnych čiar 40
5.7 Priemerná dĺžka výpočtu (ms) v závislosti od počtu generovaných spektrálnych čiar pre konštantných 5 bodov 40

Zoznam tabuliek

4.1 Priemerné časy na CPU 31
4.2 Rozdiel výsledkov na CPU 31

Úvod

Na FMFI UK na Katedre experimentálnej fyziky prebieha výskum, ktorý sa zaoberá spracovaním spektier molekúl kvôli ich detekcii alebo štúdiu ich vnútornej štruktúry. Za týmto účelom bo na katedre vyvinutý softvér, ktorý umožňuje efektívne a presné spracovanie spektroskopických dát získaných z rôznych experimentov Bol vyvinutý v prostredí LabVIEW, ktoré zjednodušilo tvorbu potrebných nástrojov.

Kvôli stále sa zvyšujúcim nárokom výskumu je tento program v neustálom vývoji. V súčasnej dobe sa tento vývoj začal zaoberať paralelizáciou výpočtov. Paralelizácia na úrovni bežných procesorových vlákien sa však ukázala ako nepostačujúca. Z toho dôvodu vznikla myšlienka preskúmať možnosti paralelizácie pomocou GPU.

Práve táto skutočnosť viedla k vzniku tejto práce. Nás osobne táto práca zaujala z troch príčin. Prvou bolo naše odhodlanie zvoliť si bakalársku prácu, ktorá mala priame praktické využitie a vynaložený čas z našej strany by pomohol výskumu v niektorej vednej oblasti. Druhou príčinou bol fakt, že táto práca poskytovala možnosť vyskúšať si ďalší druh programovania 'na železo', čo sa nám počas štúdia na univerzite stalo menšou záľubou. Tou poslednou bol náš vedľajší záujem o oblasti, v ktorých sa grafické karty využívajú ako napríklad 3D modelársky softvér, či rôzne iné graficky orientované programy.

Cieľom tejto práce bolo zoznámiť sa s výpočtovým modelom používaným vo výskume a pokúsiť sa o jeho zefektívnenie paralelizáciou výpočtov s využitím procesorových jednotiek na grafickej karte. Snažili sme sa o porovnanie efektivity odlišných spôsobov paralelizácie a o preskúmanie potenciálneho zrýchlenia používaných algoritmov.

Práca je členená na päť kapitol. V prvej kapitole sme sa venovali riešeniu paralelných problémov pomocou GPU. Popísali sme históriu vývoja, architektúru grafických kariet a dnešné spôsoby práce s grafickými kartami. Osobitnú pozornosť sme venovali stručnému popisu práce s jazykom CUDA.

Druhú kapitolu sme venovali popisu programu Peaks Synthesizer, ktorý je v súčasnosti používaný na Katedre experimentálnej fyziky. Program je určený na generovanie optických spektier. V tejto kapitole sme uviedli prostredie LabVIEW, ktoré program využíva a stručne sme popísali matematické modely použité pri spracovaní získaných dát.

Tretia kapitola sa zaoberá návrhom implementácie systému. Opísali sme v nej technické záležitosti týkajúce sa prípravy na samotnú implementáciu, algoritmus použitý v našom programe a jeho potenciál na paralelizáciu. Venovali sme sa aj problematike využitia dostupných riešení.

Obsahom štvrtej kapitoly bola samotná implementácia, v ktorej sme podrobnejšie rozobrali zvolené postupy, problémy, ktoré nastali a ich riešenia. Medzi iným táto kapitola poskytla náhľad do úvodných ťažkostí práce spôsobených prácou na diaľku, či výskumom naprieč rôznymi platformami.

V piatej a poslednej kapitole sme sa zaoberali porovnaním našich výsledkov s pôvodným programom.

Kapitola 1

Riešenie paralelných problémov pomocou GPU

S príchodom grafických kariet na konci minulého storočia sa zrodila myšlienka ich využitia aj mimo pôvodného účelu. Stalo sa tak po uvedomení si ich výhod oproti tradičným procesorom. V tejto kapitole uvedieme stručný prehľad histórie a základy vývoja systémov použitím GPGPU (General-purpose computing on graphics processing units) technológie.

1.1 Začiatky vývoja

Prvotná myšlienka bola vytvoriť špecializovaný hardvér PPU (Physics processing unit), ktorý by bol schopný spracovávať fyzikálne a iné výpočtové simulácie ako samostatný komponent PC (Personal computer) a odľahčiť tým procesor. Príkladom realizácie boli čipy PhysX od spoločnosti Ageia.[11] Tieto projekty však neboli komerčne úspešné, pretože sa časom našlo veľké prekrytie s architektúrou GPU (Graphics processing unit). V dôsledku toho bola spoločnosť Ageia odkúpená firmou Nvidia. Výskum v tejto oblasti sa tak presunul na grafické karty.

Spočiatku bol tento spôsob použitia grafických kariet doménou pokusných experimentov. Z dôvodu neprítomnosti podpory od vývojárov týchto kariet bolo nutné transformovať problém do geometrickej podoby a implementovať ho na zaužívanom grafickom API (Application programming interface[1]). Výsledok sa spätne transformoval do požadovanej podoby pre potreby simulácie. Bolo to náročné na čas a úsilie.[12]

Postupom času sa začali vyvíjať nadstavby nad tradičnými grafickými API, ktoré tento proces automatizovali. So vzrastajúcim záujmom o túto oblasť sa do vývoja zapojili aj výrobcovia grafických kariet, ktorí pridali možnosti priamych číselných výpočtov

[1]Súbor knižníc umožňujúci komunikáciu medzi jednotlivými programovými celkami v rámci vyvíjaného programu.

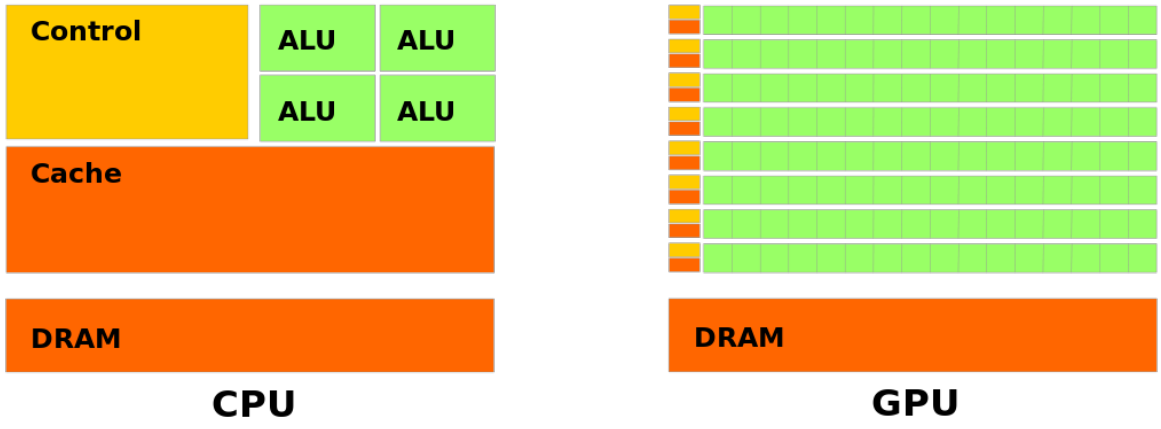
a knižníc, špeciálne určených na tieto účely. Týmto krokom sa dostala táto technológia z čisto experimentálnej sféry do praxe a bežného používateľského softvéru.

1.2 GPU

Štandardné CPU (Central processing unit) je zložené z troch hlavných častí: výpočtových jednotiek, označovaných ako jadrá, kontrolnej jednotky a pamäte. Jadrá slúžia na obsluhu a vykonávanie inštrukcií. Kontrolná jednotka obsluhuje samotné jadrá a internú logiku CPU. Pamäť vo forme registrov a cache poskytuje malý úložný priestor využívaný na vyrovnanie rozdielu v rýchlosti operácie CPU a operačnej pamäte.

1.2.1 Architektúra GPU

Na riešenie problémov pri zobrazovaní potrebujeme vykonať veľké množstvo navzájom nezávislých, a pritom jednoduchých výpočtov. Pri dizajne GPU sa preto snažíme docieliť čo najväčšie množstvo výpočtových jednotiek. Tieto jednotky môžu byť jednoduchšie než tie v CPU. Tým sa docieli ich výrazné zmenšenie (Obr.1.1,[1]). Zjednodušenie výpočtových jednotiek zároveň umožňuje zjednodušiť aj ostatné komponenty zariadenia. Týmto spôsobom vznikne na grafickom čipe voľný priestor, umožňujúci vloženie množstva ďalších výpočtových jednotiek. Docielime tak požadované zvýšenie výkonu pri paralelných výpočtoch.



Obr. 1.1: Všeobecná schéma komponentov CPU a GPU

1.2.2 Zameranie GPU

Hardvérové zmeny oproti tradičným procesorom sa prejavujú aj na spôsobe, akým grafické karty vykonávajú svoju činnosť. Hlavné rozdiely spočívajú v priepustnosti a oneskorení výpočtov.[9] Kým CPU sa zameriavajú na minimalizáciu oneskorenia za účelom

plynulej obsluhy PC ako aj používateľských vstupov, grafické karty si môžu kvôli svojej špecializácii dovoliť oneskorené dodanie požadovaných výsledkov. Vo všeobecnosti sa teda grafické karty považujú za zariadenia s vysokým počiatočným oneskorením, ktoré to vynahradzujú omnoho väčšou priepustnosťou dát (vďaka veľkému počtu súčasných výpočtov), než tomu je u CPU.

Vyššie popísané špecifiká návrhu grafických kariet sú zamerané výhradne na použitie pri grafických úlohách. Spomínaná oneskorená odozva v kontexte počítačovej grafiky nemá žiaden negatívny dopad. Pri iných paralelných výpočtoch však predurčuje isté typy úloh, na ktoré sa grafická karta hodí a na ktoré nie. Kvôli dodatočnej réžii úlohy vyžadujúce okamžitú odozvu, úlohy, ktoré dostávajú vstupy vo vzdialených časových intervaloch a iné môžu byť na grafickej karte pomalšie, než v bežnom programe. Všeobecne platí, že algoritmy implementované na grafických kartách by mali spĺňať sadu podmienok: byť vhodne paralelizovateľné, obsahovať čo najmenšie množstvo vetvenia výpočtu, operovať na veľkom množstve dát a mať čo najnižšiu závislosť medzi jednotlivými výpočtami. Vďaka výkonnosti dnešného hardvéru je však zriedkavé, že by sa začalo pracovať na vytvorení GPGPU riešenia pre algoritmus, ktorý tieto podmienky nespĺňa.

1.3 Praktická aplikácia GPGPU

Dnes technológiu GPGPU používame na riešenie širokej škály problémov. Schopnosti výrazného paralelizmu sa prejavili ako vhodné na riešenie algoritmických problémov, pričom v minulosti by sme na nich potrebovali výkonné a drahé výpočtové centrá. Pri správnej implementácii a vhodnom probléme sa môžeme dočkať výrazného poklesu času, alebo zväčšenia škály simulácie niekoľkonásobne. Asymptoticky algoritmy zostanú rovnako rýchle, ale aj konštantné zrýchlenia niekoľkonásobku rýchlosti pôvodného programu sú v bežnom živote výrazné a môžu ušetriť vynaložený čas aj energiu. V súčasnosti patria medzi najznámejšie využitia softvérové fyzikálne simulácie, distribuované výpočty (napríklad takzvané ťaženie kryptomien), simulácie veľkých štatistických modelov ako predpoveď počasia, obsluha neurónových sietí, či prelamovanie zabezpečených hesiel.

1.4 API

Na samotný vývoj programov pre grafické karty dnes používame mnoho rôznych API. Tieto sa líšia podporovanými jazykmi, funkcionalitou, ale aj podporou hardvéru.

1.4.1 OpenCL

V súčasnosti najpoužívanejším API pre výpočty na grafickej karte je OpenCL (Open Computing Language)[2], vyvíjaný od roku 2009 organizáciou Khronos Group, Inc. Ide o nadstavbu jazykov C/C++, ktorá je podporovaná všetkými dnešnými výrobcami grafických kariet. Táto vlastnosť však pokrýva viac, než len grafické karty a dovoľuje písať jednotný kód pre vedecké výpočty na každom používanom výpočtovom hardvéri. To má za následok pre grafiky menej prívetivý jazyk, ktorý, aj napriek silnej podpore, mierne výkonnostne zaostáva za konkurenciou.

1.4.2 Nvidia CUDA

Druhým najpoužívanejším API je Nvidia CUDA[1, 5] vyvíjané od roku 2007. Toto API bolo prvým komerčným API pre písanie výpočtových GPU programov. Je to nadstavba jazyka C/C++, ktorá je, na rozdiel od OpenCL, plne zameraná len na výpočty na grafickej karte. Priamym prepisom kódu z OpenCL do CUDA sa dá často docieliť výrazného zrýchlenia programu, no existuje tu problém. Na rozdiel od OpenCL, CUDA nepodporuje grafické karty od iných výrobcov. Aby sme s ňou mohli pracovať, je nutné vlastniť grafickú kartu od spoločnosti Nvidia.

Aj napriek tomu sa CUDA môže pýšiť veľmi úspešným portfóliom.[10] Dnes má každá komerčne dostupná Nvidia karta špeciálne CUDA jadrá, určené na využitie práve touto technológiou. Využíva sa v mnohých odvetviach ako napríklad pri samoriadiacich autách, ale aj vcelku bežných fyzikálnych simuláciách v počítačových hrách. Vďačí za to hlavne svojej dostupnosti pre bežných používateľov a lepšiemu výkonu v porovnaní s konkurenciou. Práve preto sme si ju zvolili pre svoj projekt aj my.

1.5 Programovanie s CUDA

V nasledujúcej podkapitole uvedieme pracovné mechanizmy CUDA. Zároveň vysvetlíme základné pojmy, ktoré definuje dokumentácia tohto jazyka.[1]

1.5.1 Vlákna

Stavebnou jednotkou paralelných výpočtov sú vlákna. Nvidia CUDA za účelom jednoduchosti práce s vláknami pridáva do jazyka C/C++ možnosť rozšíriť funkcie na takzvané jadrá (pôvodne kernels). Z funkcie, definíciu ktorej predchádza kľúčové slovo '__global__', sa tak stáva jadro. Takto definovanej funkcii môžeme potom určiť počet vlákien, ktoré ju vykonajú. Jadrá sa prirodzene môžu skladať do jeden, dvoj, až trojdimenzionálnych polí, ktoré sa nazývajú zväzky (pôvodne blocks). Každé vlákno sa v tomto n-rozmernom poli vie identifikovať svojim indexom. Podobne sa dajú skladať a

identifikovať samotné zväzky, tvoriac najvyššiu časť tejto hierarchie vlákien nazývanú sieť (pôvodne grid). Pri volaní jadra si teda určíme počet požadovaných vlákien tak, že po názve funkcie pridáme <<< počet zväzkov, počet jadier na zväzok>>>, kde sú počty vyjadrené buď ako hodnota typu integer, alebo dim3, ktorý reprezentuje daný jeden až troj dimenzionálny vektor. Toto rozšírenie jazyka dovoľuje jednoduché posielanie úloh na spracovanie grafickou kartou, na ktorej sa buduje zvyšok funkčného rozšírenia jazyka v rámci Nvidia CUDA.

1.5.2 Synchronizácia

Ďalším dôležitým faktorom správy vlákien posielaných grafickej karte je ich synchronizácia. Pri štandardných volaniach sa očakáva, že na poradí ani výsledkoch jednotlivých vlákien vo zväzkoch, či zväzkov samotných, nezáleží. Za účelom ideálneho vyťaženia grafickej karty môžu byť vykonané v akomkoľvek poradí. Zo strany CPU sa však všetok kód vykonáva sériovo, čo znamená, že sa vždy implicitne beh programu na CPU zastaví, kým sa nevykoná daná sada jadier. To však plytvá prostriedkami oboch zariadení, a tak Nvidia CUDA pridáva ďalší element, takzvané prúdy (pôvodne stream). Tieto umožňujú definovať postupnosť výpočtov, ktoré sa majú na grafickej karte vykonať bez ohľadu na CPU či stav ostatných výpočtov. Zo strany CPU sa taktiež dá explicitne či implicitne regulovať beh programu na základe stavu výpočtov na grafickej karte.

1.5.3 Pamäť

Práca s pamäťou sa v Nvidia CUDA vyznačuje dualitou medzi hostiteľom (CPU) a zariadením (grafická karta). Na správu pamäte na hostiteľovi sa používajú štandardné konvencie jazyka C. Túto základnú sadu nástrojov môžeme použiť aj na strane zariadenia pomocou pridaných funkcií s prefixom cuda, teda cudaMalloc(), cudaMemcpy() a tak ďalej. Presun dát medzi týmito dvomi úložnými priestormi je možný, ale výrazne pomalý. Tento nedostatok je tiež možné do istej miery obísť ponúkanými nástrojmi. Nvidia CUDA definuje funkcie na alokáciu pamäte hostiteľa s dodatočnými garanciami. Takto alokovaná pamäť má záruku DMA (Direct memory access) prenosu s pamäťou na grafickej karte. Môže byť taktiež zobrazená priamo do adresného priestoru grafickej karty.

Pamäť na zariadení je organizovaná do niekoľkých úrovní. Tieto úrovne stavajú na hierarchii vlákien. Každé vlákno má malé množstvo veľmi ľahko dostupnej pamäte. Nasledujú zväzky, ktoré pre svoju sadu vlákien vyhradzujú o čosi pomalšiu, ale väčšiu zdieľanú pamäť. Na poslednej úrovni sú siete, ktoré používajú globálnu pamäť grafickej karty. Pri programovaní je možné explicitne využiť túto zdieľanú pamäť. Docielime tým

zrýchlenie výpočtu a zjednodušenie synchronizácie prístupu k obsahu pre jednotlivé vlákna. Na toto slúži kľúčové slovo '__shared__' pred inicializovanou premennou.

Kapitola 2

Optické spektrá látok a program Peaks Synthesizer

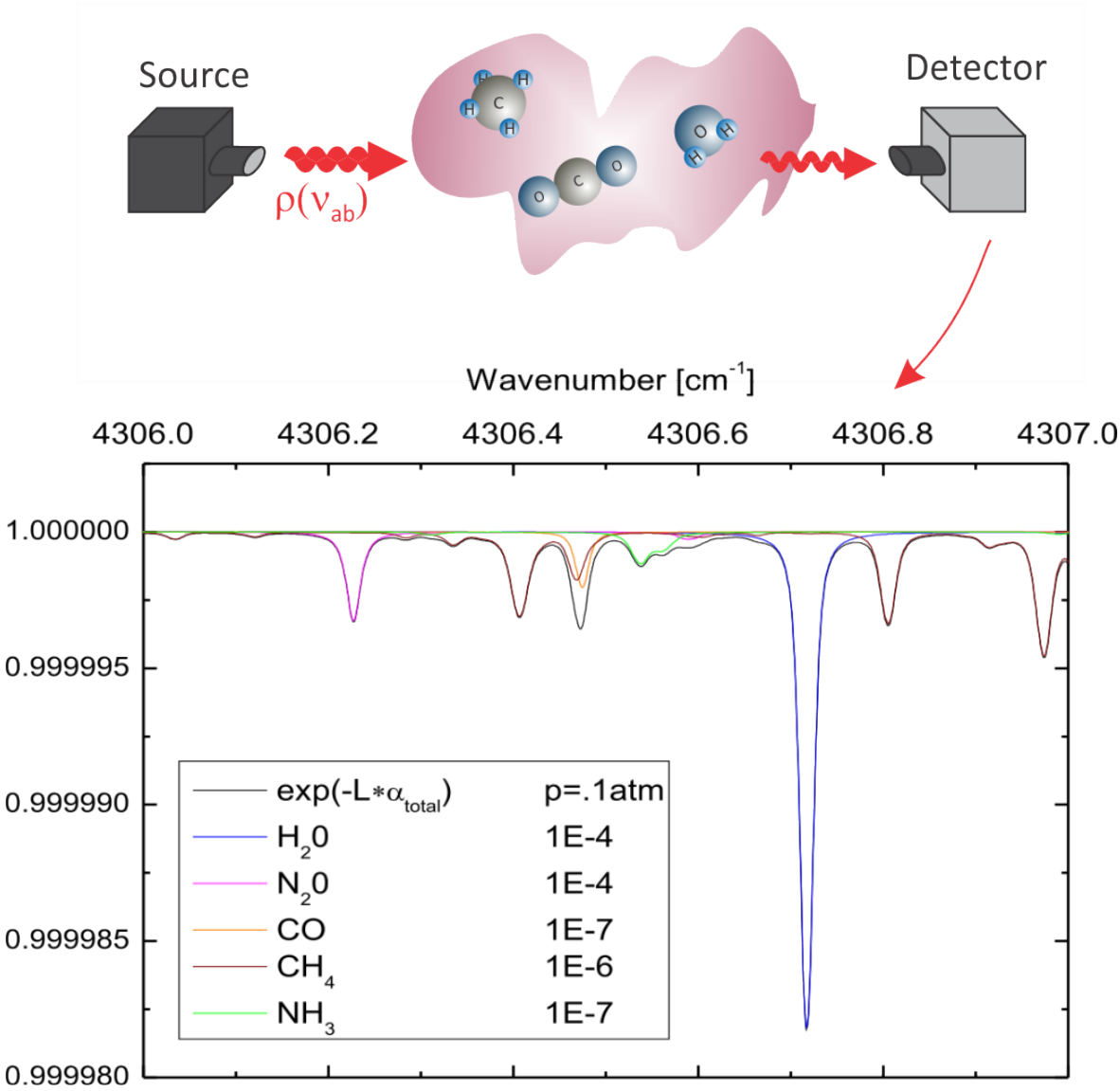
V tejto kapitole stručne popíšeme základné princípy optickej spektrosopie a matematické funkcie potrebné na modelovanie optických spektier. V druhej časti podrobnejšie rozoberieme aktuálne používané programové vybavenie (program Peaks Synthesizer vyvinutý Mgr. Petrom Čermákom, PhD.) ako aj prostredie LabVIEW, ktoré je používané na prácu so spektroskopickými dátami na na Katedre experimentálnej fyziky FMFI UK.

2.1 Optická spektroskopia

Interakciou svetla s látkou sa zaoberá oblasť fyziky, nazývaná optická spektroskopia. Optické spektrum je funkcia popisujúca množstvo svetla pohlteného alebo vyžiareného látkou v závislosti od jeho vlnovej dĺžky. Obrázok 2.1 znázorňuje veľmi zjednodušený experiment na meranie absorpcie svetla molekulami plynu. Na ľavej strane máme zdroj svetla, ktorý v danom čase generuje iba jednu vlnovú dĺžku žiarenia. To následne prechádza vzorkou, kde dochádza ku ich vzájomnej interakcii, v tomto prípade absorpcii časti žiarenia látkou. Zvyšné žiarenie je potom zachytené detektorom. Opakovaním tohto experimentu pre rôzne vlnové dĺžky získame absorpčné spektrum vzorky (spodná časť obrázku). V absorpčnej spektroskopii sa častejšie ako vlnová dĺžka používa veličina vlnočet (ν), v jednotkách cm^{-1}. Je to z dôvodu, že táto veličina je priamo úmerná energii žiarenia a v oblasti spektra, ktorá sa vyznačuje veľkou aktivitou molekúl, od 1 po 10 μm, nadobúda „rozumné“ hodnoty od 1000 po $1\,\mathrm{cm}^{-1}$.

V našom prípade (Obr.2.1) ukazuje model absorpčného spektra získaného po prechode žiarenia s vlnočtom od 4306 po 4307 cm^{-1}, po dráhe dlhej 1 m, cez vzorku plynu o celkovom tlaku 0,1 atmosféry obsahujúcu 100 ppm (častíc na milión) H_2O a N_2O, 1 ppm CH_4 a 0,1 ppm CO a NH_3. Vidíme, že príspevky rôznych molekúl majú rôzny

tvar, čo nám umožňuje ich jednoznačnú identifikáciu a následné meranie.



Obr. 2.1: Model absorpčného spektra získaného po prechode žiarenia po dráhe dlhej 1 m, cez vzorku plynu o celkovom tlaku 0,1 atmosféry obsahujúcu 100 ppm (častíc na milión) H_2O a N_2O, 1 ppm CH_4 a 0,1 ppm CO a NH_3.

2.1.1 Spektrum molekuly

Na obrázku 2.1 vidíme, že príspevky od rôznych molekúl majú pomerne komplikovanú štruktúru. Za tú je zodpovedná štruktúra interagujúcej látky. Konkrétne ide o to, že molekula alebo atóm môžu absorbovať alebo vyžiariť fotón o energii (teda s daným vlnočtom), ktorý prislúcha iba rozdielu dvoch existujúcich energetických hladín (stavov) v molekule. Pretože štruktúra atómov a molekúl je diskrétna, výsledkom je čiarový charakter spektra. Z toho vyplýva, že spektrum danej molekuly $S(\nu)$ je rovné súčtu

spektrálnych čiar (Peak), zodpovedajúcich všetkým existujúcim prechodom medzi hladinami η a η' v pre danú oblasť energií:

$$S(\nu) = \sum_{\forall \eta \to \eta'} Peak(\nu, \nu_{\eta \to \eta'}) \ . \tag{2.1}$$

2.2 Profil spektrálnej čiary

Za účelom získania informácie z optických spektier je potrebné ich vedieť namodelovať, teda vypočítať rovnicu 2.1 pre zadaný interval vlnočtov ν_{min}, ν_{max}. Na to potrebujeme poznať funkciu spektrálnej čiary od frekvencie $Peak(\nu, \nu_{\eta \to \eta'})$. Tvar a charakter tejto funkcie závisia od množstva parametrov naviazaných nielen na štruktúru študovanej látky, ale aj to, v akom prostredí sa nachádza. Za účelom tejto práce sa oboznámime len s matematickým modelom najviac používaných funkcií. Fyzikálne pozadie jednotlivých parametrov môže nájsť čitateľ v publikácii Tennyson a kol.[13]

2.2.1 Gaussova funkcia

Gaussova funkcia (2.2) je všeobecne známou a často používanou funkciou. Využíva sa ako model normálnej distribúcie, čím zahŕňa veľké množstvo javov pozorovaných pri výskume.

$$G(x) := \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.2}$$

2.2.2 Lorentz-Cauchyho rozdelenie

Lorentz-Cauchyho rozdelenie (2.3) je ďalšou funkciou reprezentujúcou hustotu pravdepodobnosti. Rovnako ako v prípade Gaussovej funkcie má tvar zvonu. Príkladmi využitia sú problémy ako riešenie diferenciálnej rovnice popisujúcej silnú rezonanciu, či popis rozloženia spektrálnych čiar v spektroskopii. Toto rozdelenie má dva parametre, x_0 označuje stred krivky a γ označuje smerodajnú odchýlku.

$$L(x) := \frac{1}{\pi\gamma \left[1 + \left(\frac{x-x_0}{\gamma}\right)^2\right]} \tag{2.3}$$

2.2.3 Voigtova funkcia

Voigtova funkcia (2.4) je funkcia špecifická pre oblasť spektroskopie. Ide o neanalytickú funkciu, ktorá vznikne konvolúciou Gaussovho a Lorentz-Cauchyho rozdelenia. Jej význam spočíva v modelovaní dvoch javov, z ktorých jeden má normálne rozdelenie a druhý má Lorentz-Cauchyho rozdelenie.

$$V(x) := \int_{-\infty}^{\infty} G(x';\sigma)L(x-x';\gamma)\,dx' \tag{2.4}$$

Keďže sa jedná o neanalytickú funkciu, jej výpočet je značne náročný na výkon a čas. Existuje však rýchlejší spôsob jej výpočtu(2.5) pomocou takzvanej Faddejevovej funkcie(2.7), ktorú popíšeme neskôr. Tento spôsob vyžaduje upraviť vstupné hodnoty do správneho tvaru. Vzniknuté komplexné číslo z využijeme ako parameter Faddejevovej funkcie. Reálna časť výsledku tejto funkcie určí výsledok Voigtovej funkcie. Týmto spôsobom je počítaná Voigtova funkcia aj vo výpočtových knižniciach programu Peaks Sythesizer.

$$V(x) := \frac{\Re[w(z)]}{\sigma\sqrt{2\pi}}, \quad z = \frac{x+i\gamma}{\sigma\sqrt{2}} \tag{2.5}$$

2.2.4 Hartmann-Tranova funkcia (HTP)

Posledná a výpočtovo najnáročnejšia funkcia využívaná v programe je Harmann-Tranova funkcia.[13] Jej vznik podnietila Voigtova funkcia, ktorá sa v istých prípadoch ukázala ako nedostatočne presná. Ide o pomerne zložitú funkciu so siedmimi parametrami. Pri jej výpočte sa tiež využíva Faddejevovej funkcia(2.7).

2.2.5 Chybové funkcie

Funkcia erf(x)(2.6) je v matematike označovaná ako chybová funkcia (Gaussova chybová funkcia). Nie je elementárna a jej grafom je krivka sigmoidného tvaru. V štatistike reprezentuje pravdepodobnosť, že náhodná premenná s normálnou distribúciou so stredom v 0 a rozptylom $\frac{1}{2}$ padne do rozsahu $[-x, x]$.

$$\text{erf(x)} := \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{2.6}$$

Doplnková chybová funkcia je definovaná ako doplnok k chybovej funkcii definovanej vyššie: $erfc(x) = 1 - erf(x)$. Táto funkcia je rozšíriteľná do oboru komplexných čísel $\mathbb{C}$.

2.2.6 Faddejevovej funkcia

Program Peaks Synthesizer využíva takzvanú Faddejevovej funkciu(2.7). Je škálovaným rozšírením doplnkovej chybovej funkcie do oboru komplexných čísel.

$$w(z) := e^{-z^2}\,\text{erfc}(-iz) = e^{-z^2}\left(1 + \frac{2i}{\sqrt{\pi}} \int_0^z e^{t^2} dt\right) \tag{2.7}$$

2.3 Peaks Synthesizer

Používaný program sa skladá z dvoch častí. Prvá časť je vytvorená v programe LabVIEW. Poskytuje jednoduché prostredie na simuláciu kriviek určených nameranými dátami. Dáta pri spektroskopickom výskume prechádzajú transformáciami, ktoré sú pre prostredie programu LabVIEW výpočtovo náročné. Kvôli tomu sa na tento účel používajú pomocné knižnice napísané v jazyku C++.[15] Tieto knižnice sú druhou časťou používaného programu. Upravené dáta sa následne vizuálne zobrazia na zvolenom grafe.

2.4 LabVIEW

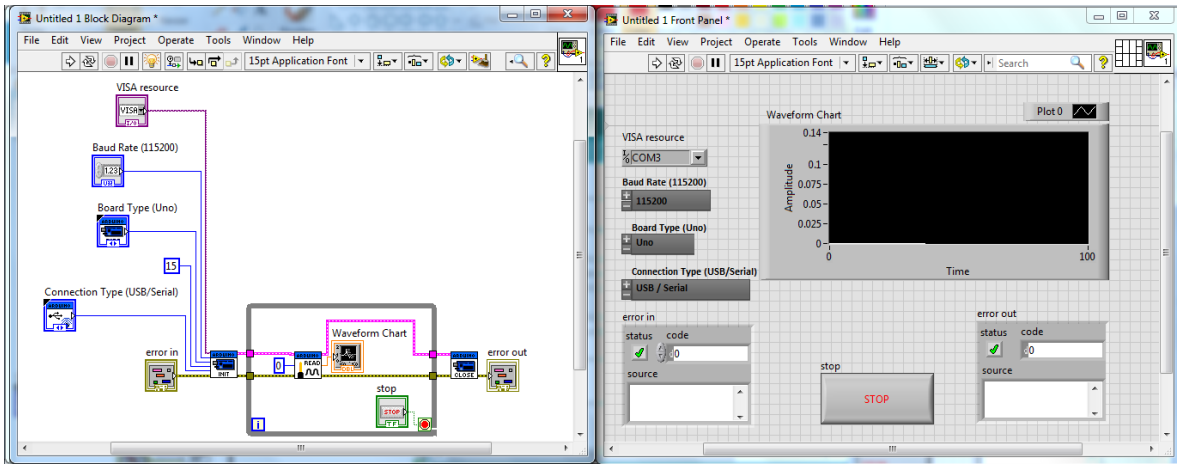
LabVIEW (Laboratory Virtual Instrumentation Engineering Workbench) je programovacie vývojové prostredie na návrh systémov založené na grafickom programovaní (Obr.2.2,[8]). Vytvorila ho spoločnosť National Instruments. V súčasnosti je dostupné na všetkých hlavných platformách používaných v osobných počítačoch.[4] Jadro programu tvorí grafický jazyk G. Vytvoril ho Jeffrey Kodosky, ktorý vyvinul grafické nástroje a implementoval ich do programu, ktorý nazval LabVIEW.

2.4.1 Vizuálne programovanie

Na rozdiel od tradičných vývojových prostredí sa v prostredí LabVIEW programy vytvárajú umiestňovaním a spájaním vizuálnych prvkov. Tento spôsob programovania je rýchly, jednoduchý a efektívny, no hlavne prístupný širšej škále používateľov. Vizuálne progrogramové prvky totiž nevyžadujú znalosť syntaxe a konvencií, ktoré tvoria základ písaných programov. Prostredie LabVIEW obsahuje mnoho vlastných knižníc, ale dokáže integrovať aj knižnice napísané v iných programovacích jazykoch. Používa sa od roku 1986, kedy bol za účelom prístupnejšieho programovania využitý v počítačoch Macintosh od spoločnosti Apple.[7]

2.4.2 Využitie LabVIEW

Toto vývojové prostredie má všestranné využitie. V súčasnosti sa najčastejšie používa na zber dát, kde je možné prepojiť meracie zariadenie s počítačom prostredníctvom DAQ (Data Acquisition) karty. Široké využitie má v ovládaní meracích prístrojov, či priemyselnej automatizácii. Tu sa využíva na riadenie a vizualizáciu výrobných procesov. Je tiež populárne v robotike, kde ho používajú nielen profesionáli, ale aj nadšenci. Ďalším príkladom všestrannosti je NASA projekt Mars Science Lab, kde použili LabVIEW na analýzu, zobrazenie údajov a monitorovanie stavu terénneho vozu na povrchu



Obr. 2.2: Ukážka prostredia LabVIEW

Marsu. Pri práci s LabVIEW možno použiť jednu zo základných šablón, ktoré sú k dispozícií. Ďalšou možnosťou je vytvoriť prázdny projekt, nazývaný Virtual Instrument (VI). Do tohto VI sa vkladajú jednotlivé grafické bloky. Tie spolu komunikujú a navzájom sa ovplyvňujú pomocou virtuálnych vodičov/uzlov. Uzly reprezentujú vstupné premenné grafického bloku. V prostredí možno zapojiť viacero paralelných uzlov, čo umožňuje vývoj viacvláknových programov. Každé VI pozostáva z troch častí: predný panel, konektorový panel a blokový diagram. Akákoľvek časť programu (podprogram, nová funkcia) môže byť naviac abstrahovaná do SubVI a zastúpená na vyššej úrovni novým symbolom (ikonou) so všeobecnými vstupmi a výstupmi.

2.5 Spracovanie dát

Spracovanie dát v programe Peaks Synthesizer prebieha výpočtom rovnice (2.1) zo zadaných parametrov. Slúži na to hlavná knižnica, ktorá poskytuje zvyšku aplikácie rozhranie na rôzne, navzájom nezávislé výpočty.

2.5.1 Metódy výpočtu hodnôt

Súčasťou tejto knižnice sú implementácie výpočtu rôznych funkcií, spravidla v dvoch formách. Prvá z nich je určená na výpočet hodnôt, potrebných na zobrazenie grafu danej funkcie, pomocou amplitúdovej metódy. Druhá forma sa zaoberá výpočtom potrebných hodnôt pomocou veľkosti požadovanej plochy fukcie. Knižnica obsluhuje výber medzi týmito funkciami a spôsobmi výpočtu pomocou binárnej vlajky a identifikačného čísla požadovanej funkcie. Implementované sú funkcie na tvorbu grafu Gaussovej krivky, Lorentz-Cauchyho krivky, Voigtovej krivky a Hartmann-Tranovej krivky. Tieto krivky ponúkajú rozlične silné matematické modely, ktorými sa v spektrometrii aproximujú jednotlivé zložky skúmaného spektra.

Výpočet neanalitických funkcií (Voigt a HTP) v programe Peaks Synthesizer rieši knižnica Cerf napísaná výskumným tímom z Európskej organizácie pre jadrový výskum CERN a Národného inštitútu pre subatómové častice Nikhef z Holandska.[6] Táto knižnica bola vytvorená za účelom poskytnutia rýchlejšieho výpočtu Faddejevovej funkcie, než ponúkali vtedajšie výskumné programy. Výpočet je realizovaný pomocou aproximácie Fourierovým radom, ktorý reprezentuje odhad Faddejevovej funkcie v požadovanom bode. Presnosť výsledku je určená na úroveň **double** pre štandard C++, teda $2 \cdot 10^{-16}$. Tento výpočet je paralelizovaný na úrovni SIMD (Single Instruction Multiple Data) inštrukcií CPU.

Kapitola 3

Návrh implementácie systému

V tejto kapitole uvedieme stav na začiatku práce. Zameriame sa na podnety vedúce k požiadavke Mgr. Petra Čermáka, PhD. o pomoc študentov informatických katedier FMFI. Následne popíšeme základný plán, ktorým sme sa počas implementácie snažili riadiť.

3.1 Počiatočný stav

Na Katedre experimentálnej fyziky bola používaná funkčná aplikácia vyvíjaná v spolupráci s katedrou AIN FMFI, ktorá však z používateľského hľadiska nedosahovala požadované výsledky v rýchlosti odozvy. Predošlé pokusy o jej zrýchlenie neviedli k požadovanému zlepšeniu. Kúpa výkonnejších počítačov by bola pre katedru finančne náročná a pritom by dostatočné zlepšenie pravdepodobne neposkytla. Optimalizácia aplikácie sa tiež ukázala ako nepostačujúca. Nakoniec vznikol návrh využiť pri implementácii výpočtov schopnosti grafickej karty. Kúpa komerčnej grafickej karty bola finančne únosná a zároveň sa ukázala ako vhodná voľba na výraznejšie zrýchlenie grafovacích častí aplikácie. Vyžadovala si však novú implementáciu niektorých častí používanej aplikácie. Táto požiadavka sa stala cieľom tejto práce.

3.2 Algoritmus

Program, s ktorým sme začali pracovať fungoval na základnom algoritme vnorených cyklov. Vonkajší cyklus pozostával z načítania potrebných hodnôt, vypočítania sady konštánt a zavolania funkcie, ktorá pripočítala ku grafu novú sadu funkčných hodnôt. Táto funkcia vnútorne rozhodla o funkcii, ktorá sa bude počítať a potom tak vo vlastnom cykle bod po bode urobila. Tieto výpočty pracovali na spoločnej pamäti, ktorá reprezentovala funkčné hodnoty grafu, ale v rámci samotnej funkcie boly úplne nezávislé. Dôležitou súčasťou týchto výpočtov bola pridružená knižnica 'Cerf', ktorá

predstavovala zložitý výpočet komplexného čísla pomocou aproximácie. Veľké množstvo navzájom nezávislých dát počítaných rovnakým spôsobom, ako v tomto algoritme, je ideálne na paralelný výpočet.

3.3 Návrh riešenia

Pri vývoji sme sa najprv rozhodli zamerať na vytvorenie novej implementácie knižnice Cerf.[6] Jej výpočty tvorili jednu z hlavných častí implementovaného programu a bez jej sprevoznenia by sme nemohli na GPU počítať žiadne zo zložitejších funkcií požadovaných od nášho programu. Bola relatívne stručná, s dobrou dokumentáciou a mala už implementovanú istú úroveň paralelizmu. Pri výskume tejto funkcie sme zistili, že existuje knižnica, ktorá tento problém už rieši. Táto knižnica, nazývaná 'CERNlib_CUDA', avšak implementuje starší algoritmus, preto sme sa rozhodli použiť ju len dočasne.

Pre nasledujúci krok sme navrhli dve možnosti. Prvou bolo využiť nezávislosti jednotlivých funkčných hodnôt a paralelizovať výpočet na tejto úrovni. Druhá možnosť sa zameriavala na celý výpočet, teda aj vonkajší cyklus. Táto možnosť by vyžadovala dodatočnú pamäť a réžiu, no dovolila by nám zrýchliť aj problémy, v ktorých sú výpočty vnútorného cyklu príliš krátke na to, aby boli pre GPU vhodné. Nakoniec sme sa rozhodli použiť oba postupy a porovnať ich podľa ich správania pri bežných výpočtoch.

3.4 Príprava zázemia

Naša práca sa zaoberala úpravou už existujúceho a fungujúceho softvéru. Vzhľadom na to sme sa na začiatku rozhodli s ním zoznámiť a detailne ho preskúmať. Pri tomto prieskume sme diskutovali o jednotlivých zložkách softvéru a určovali sme, ktoré z nich majú najväčší vplyv na rýchlosť aplikácie. Zoradili sme si ich podľa šance na úspešnú paralelizáciu. Po analýze sme sa rozhodli upraviť podporné výpočtové knižnice, ponechajúc prostredie implementované v LabVIEW v pôvodnom stave. Pán Mgr. Peter Čermák, PhD. nám poskytol zdrojový kód týchto knižníc a tiež inštaláciu LabVIEW schopnú spustiť obmedzenú verziu aplikácie.

3.5 Výber terminálu

Problém, ktorý sa vyskytol, bola kompatibilita LabVIEW projektov medzi rôznymi platformami. LabVIEW je prístupný na všetkých hlavných platformách. Tieto verzie programu sa však licencujú každá zvlášť a nie sú schopné vytvoriť spustiteľné súbory naprieč platoformami. Toto sa prejavilo ako problém, keďže sme sa rozhodli vytvárať

novú verziu knižníc na počítači s operačným systémom postavenom na Linuxe. Lenže pôvodná verzia bola vytvorená na LabVIEW určenom výhradne pre operačný systém Windows. My sme museli vyvíjať implementáciu v Linuxovom prostredí kvôli požiadavke na implementáciu pomocou grafického API Nvidia CUDA. To vyžaduje grafickú kartu od spoločnosti Nvidia, ku ktorej sme nemali priamy prístup. Bolo preto potrebné nájsť počítač, ktorý má potrebnú grafickú kartu a zároveň je nepretržite dostupný pre naše potreby. Vhodným sa ukázal jeden z počítačov, ktoré sú na našej fakulte dostupné pre výpočtové potreby študentských projektov. Po zriadení konta a SSH (Secure Shell) spojenia s týmto počítačom sme mohli začať implementáciu programu.

Kapitola 4

Implementácia programu

Predposledná kapitola sa venuje celému procesu implementácie programu. Od prvotných pokusov, cez riešenia vzniknutých problémov a úpravy kódu, až po popis výsledného softvéru.

4.1 Príprava LabVIEW

Na základe navrhnutého plánu sme začali pracovať na úprave softvéru. Prvou úohou bolo úspešne spustiť používanú verziu programu na našom počítači a na fakultnom počítači, ku ktorému sme zriadili diaľkový prístup.

4.1.1 Verzia pre Linux

Hneď na začiatku však nastalo niekoľko problémov. Na našom osobnom počítači bol po nainštalovaní prostredia LabVIEW Run Time Environment na platforme Windows program Peaks Synthesizer plne funkčný.

Nájsť vhodnú verziu pre operačný systém postavený na Ubuntu Linuxe sa prejavilo ako problém. Oficiálna stránka LabVIEW neponúkala staršie verzie LabVIEW Run Time Environemnt pre Linux. Link na stiahnutie sme nakoniec našli vďaka užívateľom s rovankým problémom, ktorí poskytli URL na časť oficiálnych stránok, ktoré nie sú bežne dostupné.

4.1.2 Problém kompatibility

Ďalší problém bol v tom, že distribúcie operačného systému Debian nie sú oficálne podporované. Navyše, pre verziu LabVIEW 2013 použitú v programe Peaks Synthesizer nebola ešte vyvinutá podpora pre architektúru AMD64. Problém kompatibility sme vyriešili použitím programu Alien z oficiálneho repozitára operačného systému Ubuntu.

Tento program poskytuje konverziu medzi rôznymi formátmi balíčkov používaných v operačných systémoch, ktoré nie sú postavené na Debiane.

Problém podpory AMD64 a teda aj 64-bitového operačného systému sme však neboli schopní vyriešiť. Nakoniec sme sa rozhodli nainštalovať novšiu verziu LabVIEW. Spätná kompatibilita však nie je oficiálne podporovaná a nebola zaručená.

4.1.3 Licenčné problémy

Pôvodný program bol vytvorený a skompilovaný vo vývojovom prostredí pre platformu Windows. Vzhľadom k tomu nebol na používanom fakultnom počítači spustiteľný. Boli sme teda nútení nájsť spôsob, ako získať funkčný binárny súbor aj pre platformu Linux.

Tu sa ukázal problém s licenciou. Licencia programu LabVIEW sa viaže na platformu, na ktorej je program vyvíjaný. To však znamená, že jednotlivé verzie LabVIEW neumožňujú vytvárať programy spustiteľné na iných platformách.

4.1.4 Zmeny v postupe

Toto zistenie nás donútilo zmeniť postup pri vývoji. Stále sme však plánovali skúšať zmeny voči implementácii na Windowse. Problém nastal pri opätovnej kompilácii zmenených knižníc. Tieto knižnice si kvôli integrácii do programu vytvoreného v prostredí LabVIEW linkujú isté súbory, ktoré naša inštalácia LabVIEW Run Time Environment neobsahovala.

Ukázalo sa, že voľne dostupná verzia prostredia neodhaľuje potrebné súbory za účelom zabránenia úpravy knižníc využitých v programe. Vývojové prostredie je síce pre študijné účely voľne dostupné, ale len v najnovšej verzii. Keďže Peaks Synthesizer využíva LabVIEW licencovaný v roku 2013, vytvorené knižnice budú musieť byť pred nasadením znova skompilované voči tejto staršej verzii.

4.2 Riešenie problémov

Po úvodnom neúspechu s LabVIEW sme vytvorili nový plán, ako postupovať pri vývoji. Cieľom bolo čiastočne nahradiť funkciu, ktorú plnila užívateľská časť programu Peaks Synthesizer, novým programom. Ten mal slúžiť na volanie požadovaných funkcií, ktoré sme vyvinuli.

4.2.1 Výhody a nevýhody

Toto rozhodnutie malo niekoľko kritických výhod. Prvou z nich bol fakt, že odpadla nutnosť udržiavať platformovo nekompatibilné časti kódu. Ďalšou bolo zjednodušenie a

urýchlenie testovania nových zmien v programe. Poslednou výhodou bolo zjednodušenie korektného porovnania výkonu knižnice pred našimi zmenami a po nich.

Hlavnou nevýhodou tohto riešenia bola nutnosť znova implementovať vstupy a voľbu funkcií, ktorú za nás pôvodne riešil už vytvorený a odladený program. Stratili sme tiež možnosť priameho porovnávania výsledkov pomocou vykreslených grafov. Tieto nevýhody však boli prevážené potenciálom pre ušetrenie veľkého množstva času.

4.2.2 Tvorba nového rozhrania

Pri tvorbe nového rozhrania na volanie sme sa zamerali na tvorbu programu, ktorý by volal internú knižnicu 'code_Peaks_2017.cpp'. Táto knižnica v bežnej verzii programu Peaks Synthsizer slúžila hlavne ako rozhranie medzi LabVIEW a ostatnými používanými knižnicami. Bola napísaná tak, aby sa pomocou Visual C++ skompilovala do bežnej Windowsovej dynamicky linkovateľnej knižnice. Jej preklad do Linuxovo spustiteľnej podoby bol priamočiary. Po nahradení špeciálnej Visual C++ syntaxe za jej gcc alternatívy a odkomentovaní Windowsových hlavičkových súborov sa skompilovala bez väčších ťažkostí.

Na to, aby sme túto knižnicu mohli využiť, sme museli preskúmať spôsob, akým jej prostredie LabVIEW posielalo dáta. To sa ukázalo ako veľmi zaujímavé. Vstupné parametre a premenné sa totiž do samotných funkcií dostávali pomocou polí štruktúr, využívajúcich takzvaný 'struct hack'. Za účelom dodržania kompatibility s pôvodným programom sme teda implementovali do nášho rozhrania rovnaký spôsob posielania vstupu.

4.2.3 Struct hack

Išlo o starú metódu tvorby polí predom neurčenej dĺžky vnútri štruktúr za účelom zjednodušenia práce s pamäťou. Princíp tejto metódy spočíval v tom, že daná štruktúra obsahovala počet prvkov v obsiahnutom poli a na jej konci bolo pole najmenšej možnej dĺžky. V našom prípade sa v týchto štruktúrach spravovaných aplikáciou nachádzalo toto: 'double elt[1];' Následne sa pri alokácii danej štruktúry alokuje množstvo pamäte potrebné na uloženie požadovanej dĺžky poľa. Táto pamäť sa nachádza za koncom štruktúry, a teda sa do nej dá prirodzeným spôsobom indexovať pomocou minimálneho poľa (ukazovateľa) na konci štruktúry. Výhodou je, že takáto inicializácia pokrýva pole aj štruktúru zároveň, nie je preto nutné ich dealokovať osobitne. Navyše sa nemôže stať, že by sa omylom odalokovala štruktúra skôr, čím by sa stratil ukazovateľ na pole a vznikol by únik pamäte.

```
typedef struct {
    long dimSize;
```

```
    double elt[1];
} TD2;
typedef TD2 **TD2Hdl;

X = (TD2Hdl)malloc(sizeof(TD2*));
//allocate a structure and some memory on top for the array
X[0] = (TD2*)malloc(sizeof(TD2) + 9 * sizeof(double));
X[0]->dimSize = 10;
for (i = 0; i < 10; ++i) {
    X[0]->elt[i] = i;
}
```

4.3 Výmena knižnice Cerf

V ďalšom kroku sme sa rozhodli v paralelnej implementácii výpočtových knižníc využiť knižnicu CERNlib_CUDA. Táto knižnica dočasne nahradila funkcionalitu knižnice Cerf pri paralelných výpočtoch. Rozhodli sme sa tak po dôkladnejšom prieskume možných spôsobov paralelizácie. Z toho vyplinulo, že hlavné zrýchlenie algoritmu bude závisieť od paralelizácie zvyšku výpočtu a voľba pomalšieho algoritmu ho výrazne neovplyvní.

4.3.1 CERNlib_CUDA

Zvolená náhrada bola súčasťou balíčka funkcií na riešenie Faddejevovej funkcie pre potreby PyCOMPLETE (**Py**thon **Co**llective **M**acro-**P**article Simulation **L**ibrary with **E**xtensible **T**racking **E**lements). Išlo o balíček knižníc zameraných na výskum určených pre odbornú verejnosť. Integrácia s existujúcim kódom bola priamočiara, keďže staršie verzie Peaks Synthesizer využívali na výpočet Faddejevovej funkcie knižničnú funkciu z tohto balíčka.

4.4 Hlavná implementácia

Spektroskopický výskum sa zameriava na sadu problémov, ktoré vyžadujú odlišný prístup k ich vyriešeniu. Tieto prístupy sa líšia v druhoch dát, s ktorými je nutné pracovať a vplývajú teda aj na profil paralelizácie, pre ktorú sú vhodné.

Príkladom týchto odlišností je skúmanie vzdialených vesmírnych telies. Pri tomto výskume je rozlíšenie spektra výrazne obmedzené (1 na 10 nm), čo sa pri výpočte odrazí na výrazne menšom (< 10) množstve bodov, ktoré modelujúce funkcie obsahujú. Na

druhej strane, vzhľadom na komplexnosť modelovaného prostredia je potrebné, aby použitý model obsahoval veľa spektrálnych čiar (1000 až 10000 na 10 nm), čo sa odrazí aj na správaní použitého algoritmu.

Naša práca implementuje a porovnáva efektivitu paralelizácie pri dvoch extrémoch. Prvá implementácia 'PeakAdvCUDA' sa zaoberá bežným problémom modelovania priemerného množstva zložitých neanalytických funkcií, ktorý sa rieši aj v priestoroch našej fakulty. Druhá implementácia 'FuncAdvCUDA' sa opiera o problém opačnej povahy, kde sa modeluje veľké možstvo funkcií, ktoré sú osobitne výrazne jednoduchšie na výpočet.

4.4.1 Úprava obslužnej knižnice

Obsluhu našej funkcie sme zakomponovali do kódu knižnice 'code_Peaks_2017.cpp'. Táto knižnica pôvodne poskytovala rozhranie na volanie monolitických funkcií 'PeakAdv2017Single' a 'PeakAdv2017Vector', ktoré boli používané na výpočet všetkých momentálne dostupných funkcií. Nachádzal sa v nej návrh lepšieho riešenia. Tým bola nedokončená implementácia paralelizácie na úrovni procesorových vlákien.

Práve na nej sme postavili svoju vlastnú implementáciu pre spracovanie vstupov a volania funkcií, tentokrát na grafickej karte.

4.4.2 PeaksGPU

Teraz popíšeme funkciu 'PeaksGPU'. Jej účelom je poskytnúť LabVIEW funkciu, do ktorej je potrebné vložiť štyri parametre. Dva z nich sú polia hodnôt na jednotlivých osiach planárneho grafu reprezentujúce vstupné hodnoty a ich funkčné hodnoty. Ďalším parametrom je pole parametrov určujúcich jednotlivé konštanty pri výpočte (stred grafu, smerodajná odchýlka, veľkosť amplitúdy, a tak ďalej), ako aj samotnú počítanú funkciu. Posledným parametrom je hodnota, ktorá určuje, či je zvolený výpočet riešený pomocou amplitúdy, alebo obsahu krivky v stredovom bode.

```
int32_t PeaksGPU
(
        int32_t Options,
        TD2Hdl *X,
        TD2Hdl *Y,
        TD3Hdl *PeaksData)

{
        ...
        //Initialisation and error checking code
```

```
        ...

        PointerToXs    = (**X)->elt;
        PointerToYs    = (**Y)->elt;
        PointerToPeak = (**PeaksData)->elt;

    for (size_t i = 0; i < PeaksNumber; i++)
    {
        PeakAdvCUDA(PointerToXs, PointerToYs, PointsNumber,
                    PointerToPeak + i*ParNumber, am);
    }

        return 0;
}
```

Všetky polia vstupujúce do tejto funkcie sú zabalené v štruktúrach spomenutých pri opise 'struct hacku', a teda okrem samotného poľa obsahujú aj parametre navyše. Hlavným parametrom je dĺžka obsiahnutého poľa typu long. Okrem nej však pole parametrov obsahuje navyše počet vykresľovaných kriviek. Tie využívajú rovnaké vstupy, no majú vyhradené vlastné parametre.

4.5 linesAdvCUDA

Funkcia PeaksGPU po spracovaní vstupných parametrov a kontrole chýb začne sekvenčne vykresľovať jednotlivé krivky volaním výpočtovej funkcie z knižnice linesAdvCUDA s postupne meniacimi sa parametrami. Volanou funkciou je 'PeakAdvCUDA'. V prípade malého počtu (<1000) bodov vypočítaného na jednu krivku a výraznejšie väčšieho množstva počítaných kriviek je možné zvoliť funkciu 'FuncAdvCUDA', ktorá daný jednotlivé vykreslenia vykonáva naraz. Táto funkcia má však zvýšené množstvo potrebnej réžie a tak nie je vhodná pre všeobecný prípad výpočtu kriviek.

4.5.1 PeakAdvCUDA - rozhranie knižnice

PeakAdvCUDA v našej implementácii vznikla ako časť monolitov 'PeaksAdv17Single' a 'PeaksAdv17Vector', ktorá nebola priamo spätá s výpočtom funkcií. Okrem toho, že značne sprehľadnila štruktúru celého kódu, poslúžila nám ako jediná funkcia definovaná v hlavičkovom súbore, teda jediná funkcia volateľná zvonka.

Keďže išlo o knižnicu napísanú v jazykovom rozšírení CUDA, prístup k zavolaniu ostatných funkcií bol silne dizajnovo obmedzený. Premiestniť tieto funkcie do iných súborov bolo možné, ale znamenalo by to nutnosť ich kompilovať cez nvcc (kompilátor

vytvorený spoločnosťou Nvidia na kompiláciu CUDA kódu) a nebolo zaručené, ako by sa s týmito súbormi vysporiadal program ako LabVIEW 2013.

4.5.2 PeakAdvCUDA - výpočty

Hlavným cieľom tejto funkcie je sprostredkovanie výpočtov a základná obsluha potrieb grafickej karty pri volaní jadier (funkcií spúšťaných na grafickej karte).

Sprostredkovanie výberu správneho jadra sa rieši cez hlavný switch. Ten má dve vetvy. Jedna je určená pre výpočty funkcií asymptotou, druhá výpočty pomocou obsahu. Toto rozdelenie sa v budúcnosti môže rozšíriť o normalizáciu druhej premennej. Toto potenciálne rozšírenie sa tu vyskytuje preto, že užívateľ môže kedykoľvek zmeniť výpočtový model na opačný a ručne nastavená hodnota by mohla rozhodiť výsledný graf.

```
 switch (shape)
{
   case 0:
   LorentzA<<<groupSize, blockSize>>>
   (gpu_xs, gpu_ys, *(Par + 1), *(Par + 2), *(Par + 5), *(Par + 6));
   break;

   case 1:
   GaussA<<<groupSize,blockSize>>>
   (gpu_xs, gpu_ys, *(Par + 1), *(Par + 2), *(Par + 4), *(Par + 6));
   break;

   case 2:
   VoigtA<<<groupSize,blockSize>>>
   (gpu_xs, gpu_ys, *(Par + 1), *(Par + 2), *(Par + 4), *(Par + 5),
                                                        *(Par + 6));
   break;

   ...
```

4.5.3 Manažment zdrojov GPU

Predtým, než však môže funkcia PeakAdvCUDA zavolať jeden z vyššie spomenutých jadier, musí najprv vhodne pripraviť pamäť a definovať parametre určujúce počet vlákien riešiacich toto jadro paralelne.

Počet vytvorených vlákien riadia konštanty v podobe počtu vlákien/jadier na jeden zväzok a počtu zväzkov. Počet vlákien/jadier je voliteľný, no štandardne sa definuje ako mocnina 2 nad hranicou 32. Dôvodom tohto ohraničenia je, že grafická karta môže poslať na výpočet naraz najmenej 32 vlákien. V našom prípade sme konštantu zvolili na 256, no neskoršie testy by ju mohli zmeniť.

Počet zväzkov začína na jednotke (až na prípad, že je počet vstupných premenných presným násobkom počtu vlákien na zväzok) a celočíselným delením sa k nim pridá potrebný počet, aby pokryli všetky vstupné hodnoty. Týmto spôsobom môžeme poslať na výpočet všetky hodnoty v rámci jedného volania, presah výpočtu na nepotrebné hodnoty je pre nás zanedbateľne malý.

V prípade pamäte sa vytvoria v grafickej pamäti nové polia 'gpu_xs' a 'gpu_ys', do ktorých sa volaním cudaMalloc() vložia hodnoty vstupných polí 'xs' a 'ys'. Po vykonaní výpočtov sa obsah 'gpu_ys' ako jediný výstup celej funkcie zapíše do poľa 'ys' a pamäť sa uvoľní. Množstvo alokovanej pamäte je určené počtom vlákien, teda môžu presahovať hodnoty definované v 'xs' a 'ys'. Tieto hodnoty ignorujeme a pri prenose dát späť z grafickej pamäte do systémovej sa orientujeme veľkosťou vstupných hodnôt.

4.6 Výpočtové jadrá

V knižnici 'linesAdvCUDA' sme implementovali celkovo osem výpočtových jadier. Týmito jadrami sú dvojaké spôsoby výpočtu Lorentz-Cauchyho funkcie, Gaussovej funkcie a výpočet Voigtovej funkcie pomocou dvoch odlišných prístupov. Jadrá sú odlíHšené podľa mena, kde 'JadroA' znamená implementáciu pomocu amplitúdy a 'JadroS' pomocou plochy.

Všetky jadrá sú navrhnuté tak, aby boli spustené v jednom behu (pôvodne Stream) kvôli zaručeniu maximálneho paralelizmu a ostránenia nutnosti spomaľovať výpočet synchronizáciou.

4.6.1 Lorentz

```
__global__ void LorentzA(const double *xi, double *yi, const double x0,
                         const double a, const double w0, const double
{
    int myIndex = blockIdx.x * blockDim.x + threadIdx.x;

    yi[myIndex] = a * (w0 * w0) / ((w0 * w0) +
                  (xi[myIndex] - (x0 + d0))*(xi[myIndex] - (x0 + d0)));
}
```

Lorentz-Cauchyho funkcia má najjednoduchšie z jadier. Keďže v jednom behu počítame všetky možné indexy, každé jadro si najprv zistí svoju pozíciu vo vláknovej hierarchii:

```
int myIndex = blockIdx.x * blockDim.x + threadIdx.x;
```

Tento výsledok následne určí, ktorý index vo vstupnom a výstupnom reťazci danému jadru prináleží. Rovnakým spôsobom sa určuje zvolený index vo všetkých jadrách.

4.6.2 Gauss

```
__global__ void GaussS(const double *xi, double *yi,... )
{
    int myIndex = blockIdx.x * blockDim.x + threadIdx.x;

    yi[myIndex] = s * (M_SqrtLn2oPi / wD) *
                  exp(-(xi[myIndex] - (x0 + d0))
                  *(xi[myIndex] - (x0 + d0))
                  / (2 * M_1o2Ln2 * wD * wD));
}
```

Jadro na výpočet Gaussovej funkcie pri svojich výpočtoch používa predpočítané konštanty, ktoré sú zachované v hlavičkovom súbore. Rovnako ako v prípade Lorentz-Cauchyho funkcie, aj tu je výsledok dosiahnutý bez alokácie dodatočnej pamäte.

4.6.3 Voigt

```
__global__ void VoigtA(const double *xi, double *yi,... )
{
    int myIndex = blockIdx.x * blockDim.x + threadIdx.x;

    double rx = (M_SqrtLn2 * (xi[myIndex] - (x0 + d0)) / wD);
    double ry = ((w0 / wD) * M_SqrtLn2);
    double vr, vi, vr0;
    wofz(0.0, ry, &vr0, &vi);
    wofz(rx, ry, &vr, &vi);
    yi[myIndex] = a * vr / vr0;
}
```

Nasleduje implementácia Voigtovej funkcie. Toto jadro pracuje s komplexnými číslami. Pri jeho implementácii sme sa riadili vstupnými hodnotami funkcie, ktorú sme prevzali z balíčka PyCOMPLETE. Komplexné čísla sú reprezentované jednoduchou dvojicou premenných typu double. Týmto spôsobom môžeme ušetriť trocha pamäte pri opaku-

júcich sa, či nepodstatných imaginárnych častiach.

4.6.4 HTP

Poslednou implementovanou funkciou je funkcia HTP. Výpočet tejto funkcie môže prebiehať jedným zo štyroch spôsobov podľa oblasti krivky, v ktorej sa hodnoty nachádzajú a podľa toho, aké vstupné hodnoty sú dostupné. Keďže vetvenie výpočtov značne na grafickej karte spomaľuje beh programu, našim prvoradým cieľom bolo mu čo najviac zamedziť. To sme docielili rozdelením výpočtu na dve jadrá. To prvé je jadro HTP_simple, ktoré pokrýva jednu celú vetvu programu, ktorá nastala pri menšom počte zadaných parametrov. V prípade, že parametre spĺňajú isté vlastnosti je dokonca možné miesto HTP algoritmu spustiť výpočet pomovou Voigtovej funkcie.

```
__global__ void HTP_simple(const double *xi, double *yi,... )
{
  ...
  gpuCplx wZm = Zm * iCplx;
  faddeeva(wZm.real(), wZm.imag(), &real, &img);
  wZm.real(real), wZm.imag(img);

  gpuCplx Aterm = M_SqrtPI * x0c * wZm;
  if (abs(sqrt(Zm)) < (double)4000.0)
  {
    Bterm = M_SqrtPI * x0c * vMP * vMP *  (((OneCplx - (Zm*Zm)) * wZm)
                                          + Zm * M_1oSqrtPI);
  }
  else
  {
    Bterm = M_SqrtPI * x0c * vMP * vMP *  (wZm + ((double)0.5 / Zm)
          - ((double)0.75 *  M_1oSqrtPI * (Zm * Zm *Zm)));
  }
  Bterm = Aterm * (nuVC - eta*(C0 - 1.5*C2)) + Bterm *
                  (eta*C2 / (vMP*vMP));

  wZm = (OneCplx + iCplx * Ylm) * Aterm * (OneCplx/(OneCplx - Bterm));
  yi[myIndex] += s * M_1_PI * wZm.real() - (aLim * xi[myIndex] + bLim);
}
```

Druhé jadro, nazvané HTP pokrýva zvyšné tri vetvy s tým, že sa zameriava na vetvu, ktorá tvorí prevažnú časť výpočtu. Zvyšné dve vetvy sme premietli do rovnakého kódu s tým, že v kritických úsekoch modifikujú odlišujúcu premennú, po čom sa výpočty znova synchronizujú a môže byť vykonávané naraz. Zároveň sme sa za

účelom šetrenia registrov snažili udržať počet používaných premenných na minimum. Obe jadrá počítajú s hodnotami reprezentujúcimi komplexné čísla. Na zabezpečenie aritmetiky v rámci tohto číselného oboru sme zvolili dátový typ complex poskytnutý knižnicou Thrust, ktorá je súčasťou moderných verzií vývojárskeho balíčka pre jazyk CUDA. V našom kóde sme tento typ 'complex<double>' premenovali na gpuCplx.

```
__global__ void HTP(const double *xi, double *yi,... )
{
  ...
  gpuCplx ix = iCplx * (x0 - xi[myIndex]);
  gpuCplx X = (ix + C0Tilde) * invC2Tilde;

  gpuCplx Zm, Zp;

  Zm = sqrt(X + Y) - Yroot;
  Zp = Zm + (2.0 * Yroot);

  if (abs(Y) < (double)1.0e-15*abs(X)) {
    Zm = sqrt(X);
    Zp = sqrt(X + Y);
  }
  if (abs(X) < (double)3.0e-8*abs(Y)) {
    Zm = (ix + C0Tilde) * x0c;
    Zp = sqrt(X + Y) + Yroot;
  }

  double real, img;

  gpuCplx wZp = iCplx * Zm;
  faddeeva(wZp.real(), wZp.imag(), &real, &img);
  gpuCplx wZm(real, img);

  ...
}
```

Výber medzi týmito jadrami, ako aj predpočítanie všetkých spoločných parametrov sa vykonáva v hlavnej časti rozhrannej funkcie PeakAdvCUDA.

```
case 7: {
  ...constants
```

```
if (C2Tilde != ZEROCplx) {
  const gpuCplx invC2Tilde(OneCplx / C2Tilde);
  const gpuCplx Yroot(x0 * vMP * invC2Tilde * P_1o2C);
  const gpuCplx Y(Yroot * Yroot);

  HTP<<<groupSize, blockSize>>>
    (gpu_xs, gpu_ys, x0, d0, s, nuVC,
    eta, Ylm, x0c, C0Tilde, C0, C2, iCplx, OneCplx,
    invC2Tilde, Yroot, Y, mOneCplx, vMP, xLimit, aLim, bLim);
}
else {
  if ((nuVC == 0) && (eta == 0) && (w2 == 0) && (d2 == 0))
    VoigtHS<<<groupSize, blockSize>>>(gpu_xs, gpu_ys, x0,
                        s, wD, w0, d0, xLimit, aLim, bLim);
  else
    HTP_simple<<<groupSize, blockSize>>>(gpu_xs, gpu_ys, x0,
      d0, s, nuVC, eta, Ylm, x0c, C0Tilde, C0, C2,
      iCplx, OneCplx, vMP, xLimit, aLim, bLim);
};
break;
```

4.7 FuncAdvCUDA

Pri tvorbe programu, ktorý by sa zameriaval na výpočet veľkého množstva funkcií s odlišnými parametrami sme sa najprv pozreli na to, ako sa správala naivná implementácia. Táto paralelne nastavila všetky potrebné parametre z kópie vstupných polí (poskytnutých ako referencie z bežiacej inštancie LabView) a potom volala už implemenované funkcie na paralelný výpočet potrebných funkcií. Naivná implementácia však rýchlostne zaostávala za celým výpočtom na CPU aj bez toho, aby začala akýkoľvek výpočet. Keďže sa ukázalo, že posielať túto časť programu na GPU neposkytuje žiaden časový benefit, naším novým cieľom sa stalo tieto parametre čo najefektívnejšie poslať algoritmu na výpočet.

4.7.1 Réžia súbežných výpočtov

Na tento účel sme použili polia parametrov v pripnutej pamäti[1], čo poskytuje rýchlejší prenos medzi pamäťami CPU a GPU. Keďže sú všetky parametre vložené do globálnej

[1]Pamäť, ktorá bola alokovaná bez možnosti jej odswapovania do virtuálnej pamäte.

pamäte grafickej karty, vytvorili sme nové výpočtové jadrá, ktoré si potrebné parametre načítajú počas behu programu. Ďalšou zmenou v správaní výpočtových jadier bolo ukladanie ich výsledkov. Tie sa ukladajú do pamäte usporiadané podľa ich vstupnej hodnoty. Výsledkom je pole obsahujúce výsledky všetkých modelovaných funkcií pre X_1, potom X_2, atď. Toto usporiadanie sme zvolili preto, aby sme mohli nakoniec tieto výsledky sčítať dokopy v čo najmenšom možnom čase.

4.8 Cerf_CUDA

Po zavedení a otestovaní vytvorenej knižnice sme sa vrátili k paralelizácii novšieho algoritmu na výpočet Faddeevovej funkcie. Tento algoritmus je používaný sériou CERN knižníc pod názvom ROOT, v pôvodnom programe sa nachádzal pod názvom Cerf. Keďže algoritmus cituje vyššiu presnosť (priemerná chyba 10^{-16}, v najhoršom prípade 10^{-13}, CERNlib_CUDA poskytuje v priemere presnosť 10^{-14} a pre čísla menšie ako 8 sa presnosť zníži až na 10^{-9}) a rýchlosť (3-násobné zrýchlenie pri vstupoch s číslami menšími ako 8) ako nami použitý algoritmus, rozhodli sme sa najprv otestovať tieto parametre v rámci čistých CPU implementácií. Tento test použil knižničné implementácie oboch algoritmov a na rovnakom pseudonáhodnom vstupe spustil milión iterácií výpočtu. Raz v rozsahu vstupov < 1 a raz bez obmedzení.

CERNlib <1	CERNlib	ROOT <1	ROOT
416.6 ms	135.6 ms	173.5 ms	210.1 ms

Tabuľka 4.1: Priemerné časy na CPU

Chyby	$> 1 \times 10^{-13}$	$> 1 \times 10^{-12}$	$> 1 \times 10^{-11}$	$> 1 \times 10^{-10}$
0 - 1	všetky	759490	233841	0
> 1	0	0	0	0

Tabuľka 4.2: Rozdiel výsledkov na CPU

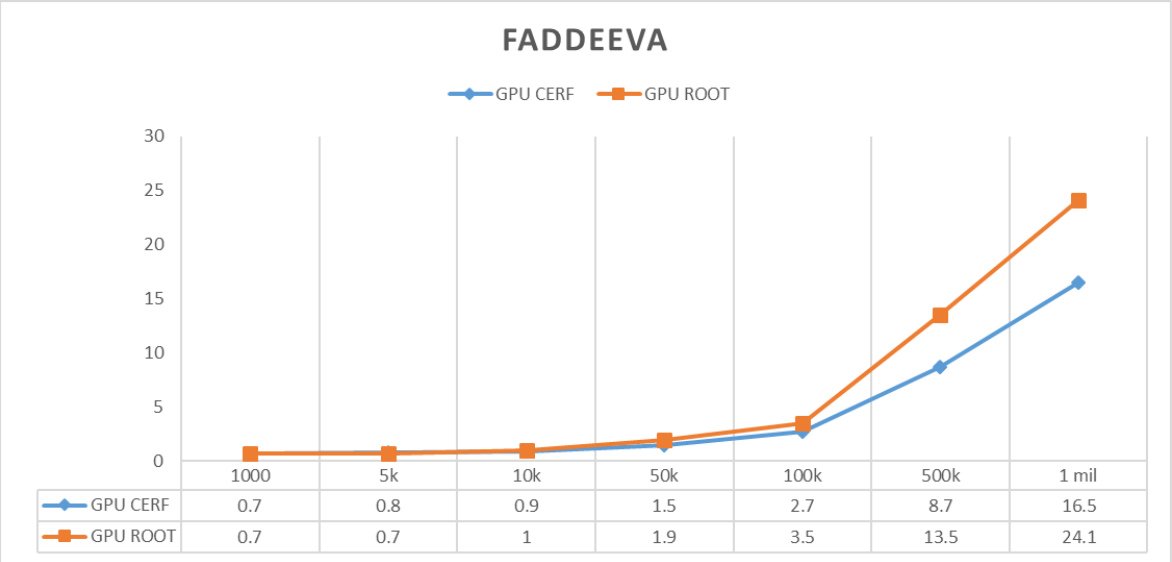
Ako vidno v tabuľkách (Tab.4.1; 4.2) CERNlib je celkovo rýchlejší. Je však menej presný a nekonzistentný. Keď sa dostane k číslam pod hodnotou 1, výrazne sa spomalí a práve tu prichádza k najväčším chybám výpočtu. ROOT je rovnako rýchly na všetkých vstupoch. Našim cieľom bolo dosiahnuť aspoň zhodné rozdiely rýchlosti pri implementácii paralelnej verzie algoritmu v porovnaní s programom prevzatým z CERNlib_CUDA knižnice.

4.8.1 Cerf_CUDA - implementácia

Algoritmus sa v knižnici ROOT nachádza v dvoch variantoch, a to 'faddeeva' a 'faddeeva_fast'. Rozdielom je výsledná presnosť, korá sa pri 'faddeeva_fast' pohybuje len na úrovni 10^{-9} a tá sa zhoduje s horším prípadom nášho pôvodného riešenia. Hlavnou motiváciou na toto rozšírenie je zlepšenie presnosti výpočtov, preto sme sa rozhodli implementovať iba funkciu 'faddeeva', ktorá v tomto obore presahuje pôvodné riešenie. Samotná paralelizácia kódu bola priamočiara. Funkcia bola napísaná tak, aby výrazne zlepšovala svoje výsledky pomocou predpočítaných dát a za použitia aproximácií, ktoré zvládne SIMD jednotka CPU. Predpočítané dáta boli zväčša statické. Výnimkou bol výber násobku čísla π určeného počas behu programu, ktorý sme nahradili priamym výpočtom. Ostatné konštanty sme označili ako 'constexpr' a do kódu sme pridali direktívy na rozbalenie cyklov, vďaka čomu ich kompilátor vpísal priamo do volaných inštrukcií, šetriac potrebné prístupy do pamäte. Konštanty predpočítaných exponentov taylorovho radu sme vložili do konštantnej pamäte, ktorá bola na tento účel najvhodnejšia.

4.8.2 Cerf_CUDA - testovanie

Výslednú implementáciu sme porovnali s funkciou wofz, ktorú sme použili pôvodne. Testy prebiehali na pracovnom PC pána Mgr. Petra Čermáka, PhD., ktorý nám poskytol staršiu grafickú kartu radu GeForce GTX750 Ti.



	1000	5k	10k	50k	100k	500k	1 mil
GPU CERF	0.7	0.8	0.9	1.5	2.7	8.7	16.5
GPU ROOT	0.7	0.7	1	1.9	3.5	13.5	24.1

Obr. 4.1: Priemerná dĺžka výpočtu (ms) náhodných komplexných čísel

Keďže výsledky (Obr.4.1) poukazovali na našu implementáciu ROOT faddeeva algoritmu ako lepšiu alternatívu k pôvodnému riešeniu (dosahujúcu presnejšie výledky pri zanedbateľnom spomalení), vo výpočtovej knižnici sme ňou nahradili pôvodnú funkciu

z CERNlib_CUDA. Profilér nám taktiež ukázal, že pri oboch algoritmoch je hlavným obmedzením ($> 80\%$ celkového času) aritmetika s dvojitou presnoťou. Tento výsledok sme očakáavali, keďže grafické karty série GeForce sú v tomto obore značne obmedzené. Naša karta zvládne iba 4 výpočty na jeden tik hodín mikroprocesora. Pri aritmetike s bežnou presnosťou je schopná až 128 výpočtov za rovnaký čas. Keďže je táto presnosť požadovaná (a teda zostane najpomalšou časťou programu), ďalšia optimalizácia by viedla iba k zanedbateľnému zrýchleniu.

Kapitola 5

Výsledky

V tejto kapitole sa zameriame na finálne porovnania našej knižnice po zavedení do prevádzky.

5.1 Metodológia

Vytvorená knižnica bola zakomponovaná do predošlej výpočtovej knižnice a bola volaná zjednoteným systémom, ktorý poskytoval prepínanie medzi funkciami pomocou vstupnej vlajky. Výsledné časy sme získali pomocou interného profilovacieho nástroja LabVIEW zvaného 'Performance and Memory Profiler'. Tento nástroj meria čas výpočtu a spotrebovanú pamäť jednotlivých LabVIEW VI. Pri profilovaní sme pre dosiahnutie maximálnej presnosti oddelili vykresľovanie kriviek a správu užívateľského rozhrania od LabVIEW VI, ktorého jedinou funkciou bolo zavolať pomocnú knižnicu na výpočet. Pri porovnávaní presností sme výsledné hodnoty získali uložením vypočítaných hodnôt grafu vytvorených našou LabVIEW aplikáciou.

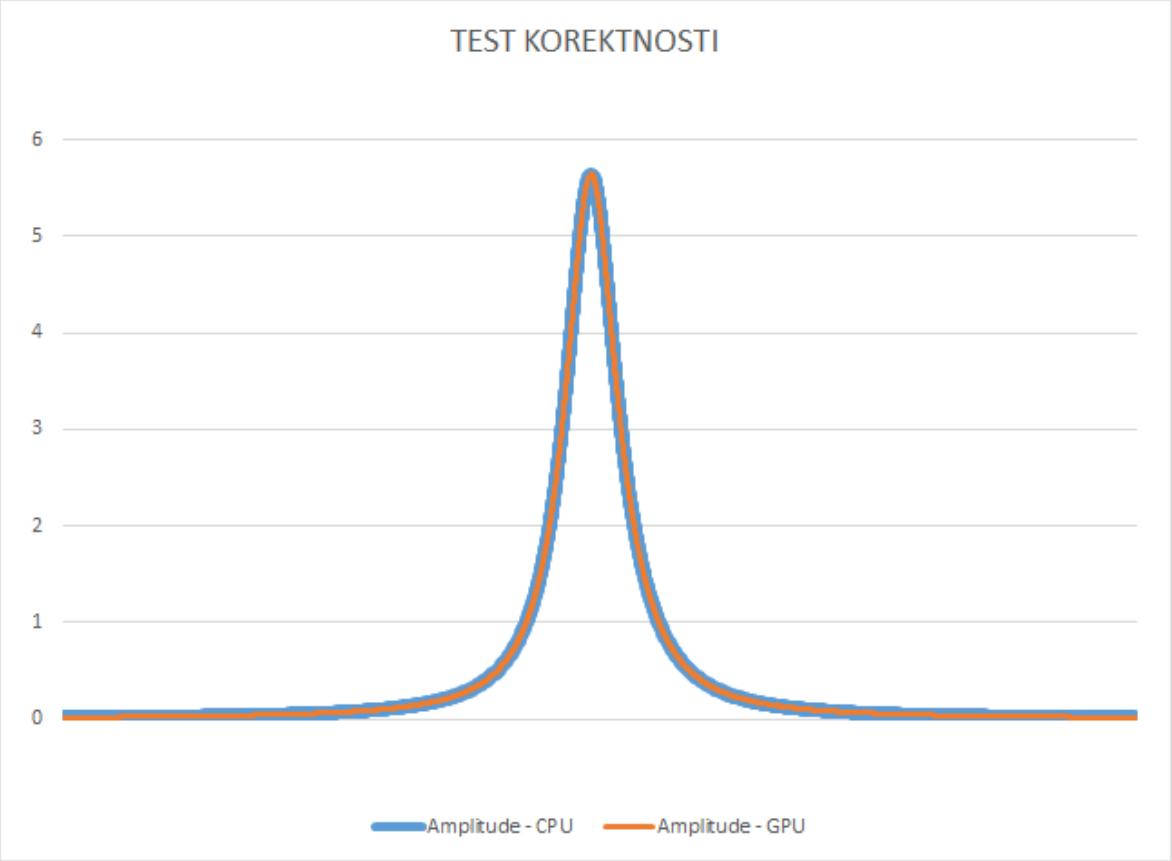
Použitý počítač mal nasledovné parametre: Intel Core i7 6700 Skylake 3.4 GHz, 4.0 GHz Turbo Boost, RAM 16 GB DDR3, NVIDIA GeForce GTX 750 Ti, Windows 10 64-bit. Všetky testy používali program LabVIEW 14 64-bit.

5.2 Porovnania presnosti

Pri porovnávaní presnosti sme chceli overiť korektnosť našej implementácie Faddejevovej funkcie a porovnať mieru chyby pôvodnej prevzatej funkcie s tou našou.

5.2.1 Korektnosť

Na overenie korektnosti sme vygenerovali množstvo hodnôt použijúc funkcie Cerf a Cerf_CUDA, ktoré sme vzájomne porovnávali. Ako príklad uvedieme jednu z HTP



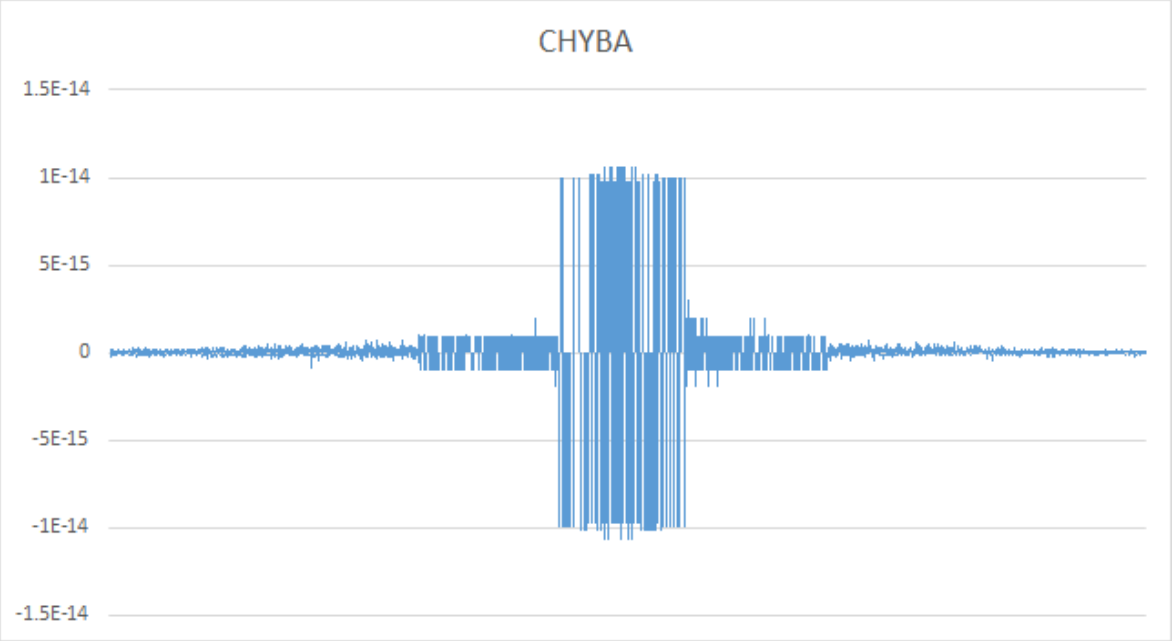
Obr. 5.1: Grafy totožnej spektrálnej čiary vygenerované pomocou funkcií Cerf a Cerf_CUDA

funkcií vygenerovaných oboma spôsobmi (Obr.5.1)[1]. Vygenerované spektrálne čiary sú zväčša totožné, no v oblasti okolo centra grafu vznikajú chyby ±1 na poslednom desatinnom mieste výsledku. Chyby tohto charakteru predstavujú prirodzenú chybu, ktorá vzniká z dôvodu zmeny poradia výpočtov. Tie pri GPU programovaní určuje kompilátor.

5.2.2 Odchýlka

Ako posledný test našej implementácie Faddejevovej funkcie sme zvolili jej porovnanie s knižničnou implementáciou staršej funkcie na jej výpočet. Keďže tá obyčajne vykazuje chyby pri výpočte hodnôt menších ako 1 (Tab.4.2), zopakovali sme tento pokus na našich GPU implementáciách. Výsledné rozdielové funkcie (Obr.5.3;5.4) sa zhodujú s našimi očakávaniami ako aj s predošlými porovnaniami týchto dvoch algoritmov. Chyby sa v zásade pohybujú v okolí 10^{-12}-10^{-10} a klesajú, keď sa blížime k hodnote 1.

[1]Priložené DVD obsahuje súbor s vygenerovanými funkčnými hodnotami.



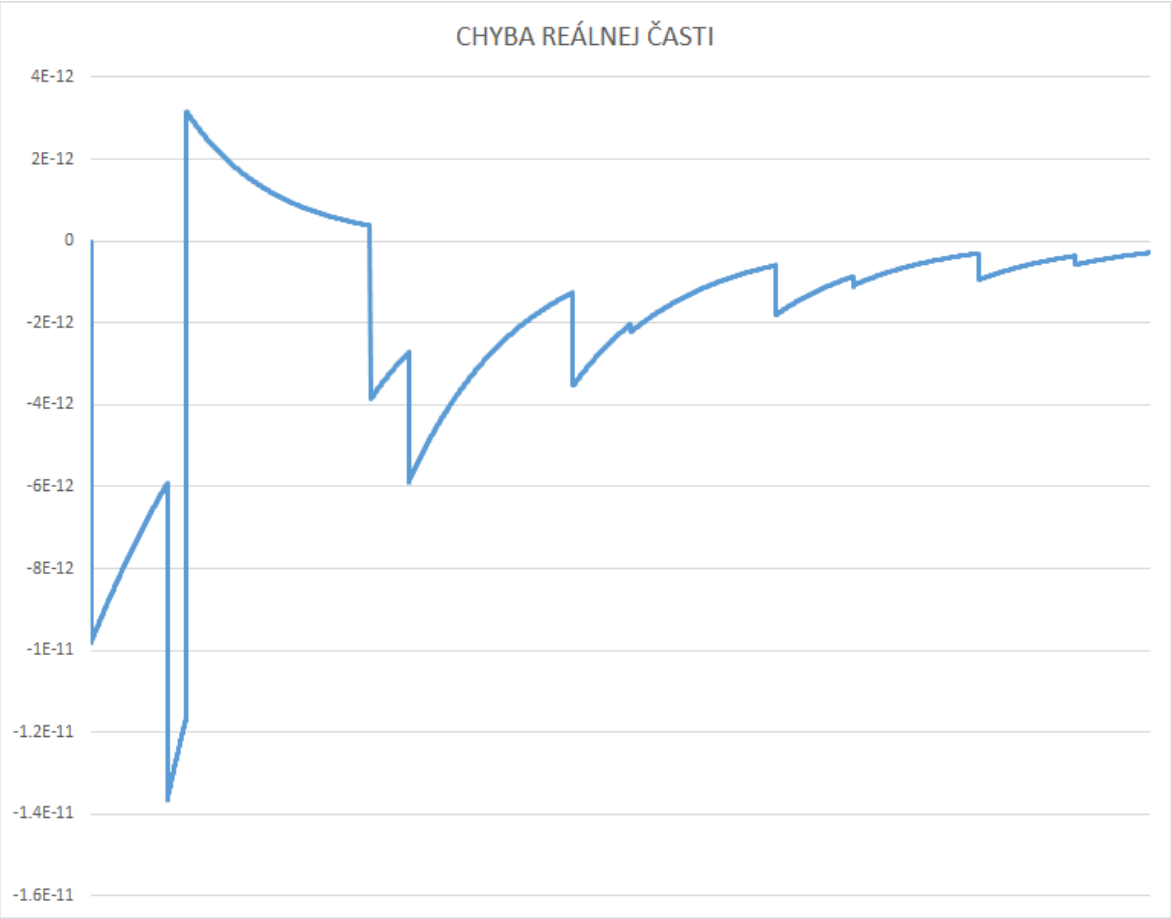
Obr. 5.2: Graf odchýlky pre hodnoty z Obr.5.1

5.3 Záťažové porovnania

Pri záťažových testoch sme sa pozreli na sériu exemplárnych prípadov reprezentujúcich najbežnejšie použitia knižnice. Tieto prípady boli zvolené tak, aby pokryli bežné používanie knižnice pri výpočte neanalytických funkcií ako HTP na CPU a GPU. Výpočty GPU prebiehali dvoma spôsobmi: GPU Peak, paralelizovaného na úrovni výpočtu jednotlivých kriviek a GPU Func, pri ktorom je paralelizovaný celý výpočet výsledného spektra. Prvý test sa zameral na výpočet samostatnej zložitej funkcie. Druhý test sa zameral na rovnaký prípad, ale pri väčšom množstve zavolaných funkcií, pozerajúc sa na škálovateľnosť pri viacnásobnom volaní funkcií. Posledný sa zameral na výpočet variabilného množstva funkcií pre veľmi malé množstvo bodov. Jeho účelom bolo určiť vzťah medzi opakovaním volania a výsledným časom.

5.3.1 Prípad I

V prvom teste (Obr.5.5) sme dosiahli očakávateľné výsledky. Oba spôsoby paralelizácie sa v prípade jednej krivky správajú zväčša rovnako. Rozdiel medzi ich časmi je zapríčinený dodatočnou réžiou a pamäťovými presunmi, ktoré v tomto prípade nemajú význam. Aj napriek tomu obe implementácie pri záťaži poskytujú v priemere 10-násobné zrýchlenie výpočtu oproti pôvodnému algoritmu.

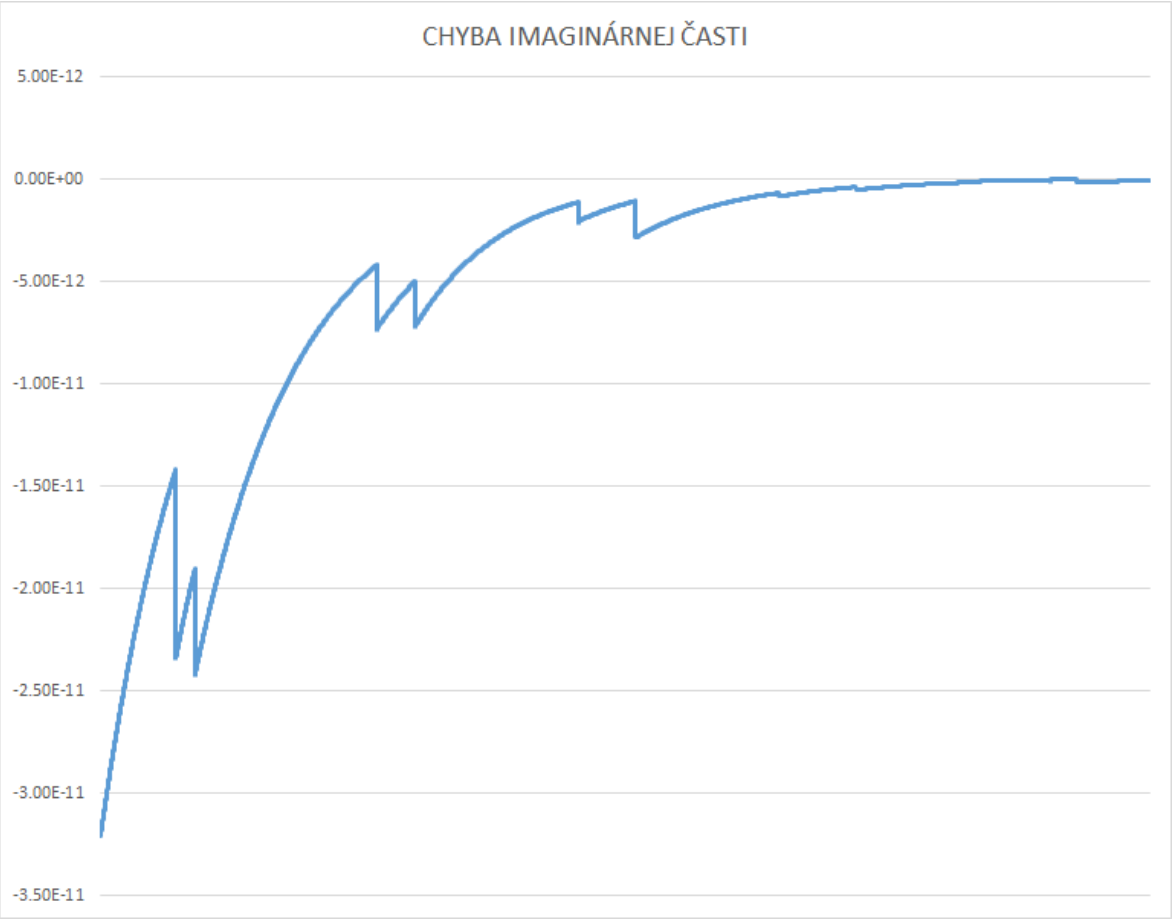


Obr. 5.3: Chyba reálnej časti na intervale <0,1>

5.3.2 Prípad II

Druhý test (Obr.5.6) odhalil niekoľko faktov. Pri rovnakom celkovom množstve vypočítaných bodov sa zachováva rýchlosť výpočtu na strane CPU. To dosahuje v priemere skoro rovnaké výsledky, ako v teste prvom. Zároveň sa potvrdzuje, že algoritmus 'FuncAdvCUDA' (v grafe označený ako 'GPU FUNC') správne škáluje aj s pribúdajúcim počtom funkcií. Tento algoritmus tiež dosahuje porovnateľné časy s prvým testom.

Zaujímavý je ale prípad pôvodného algoritmu. Ten zaostáva za svojimi výsledkami v prvom teste vcelku výrazným spôsobom. Pri zvýšení množstva počítaných hodnôt začne výpočet trvať v priemere 103 ms a toto číslo sa vo výsledkoch zachová až po hranicu jedného milióna bodov. Tento výsledok je očakávaný, keďže aj prvý test poukazuje na nízku efektivitu GPU pri výpočte malého množstva dát. Skutočným 'úzkym hrdlom' je v tomto prípade postupné volanie GPU. To sa zopakuje 1000-krát bez toho, aby grafická karta vykonala postačujúce množstvo výpočtov na zakrytie jej odozvy. V prípade 'FuncAdvCUDA' sa tento problém obchádza tak, že je volanie GPU vždy iba jedno.



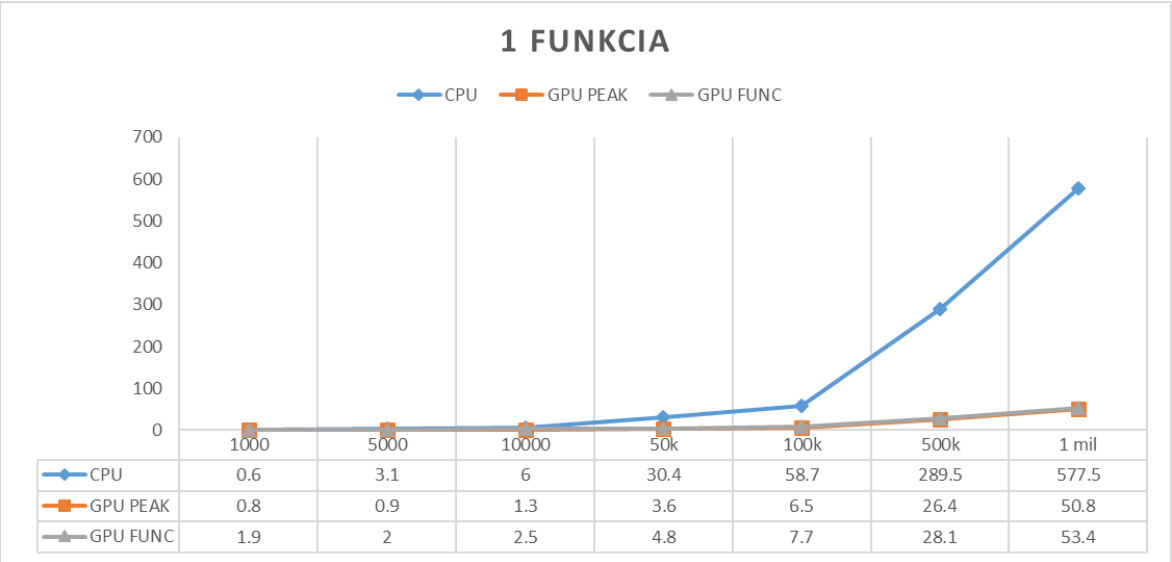
Obr. 5.4: Chyba imaginárnej časti na intervale <0,1>

5.3.3 Prípad III

Posledný záťažový test (Obr.5.7) sa zameral na vplyv množstva zavolaných funkcií na celkový čas výpočtu, odhliadnuc od množstva vypočítaných bodov. V tomto teste sa potvrdila teória o negatívnom vplyve opakovaných volaní výpočtových jadier bez primeraného množstva spracovaných dát. Čas výpočtu sa v pôvodnej implementácii zvyšuje rovnako rýchlo, ako je celkový počet modelovaných funkcií. Na jedno volanie jadra pripadá približne 1/10 ms. Toto spôsobuje, že aj jednoduchý výpočet 50 tisíc trvá až 1 sekundu. V prípade novej funkcie sa výpočty postupne dostávajú na dvojnásobné zrýchlenie. Toto zrýchlenie je však pre praktické účely nevýrazné a hlavné zrýchlenie modelovania spektra je striktne naviazané na množstvo počítaných bodov, nie funkcií.

5.4 Zhrnutie

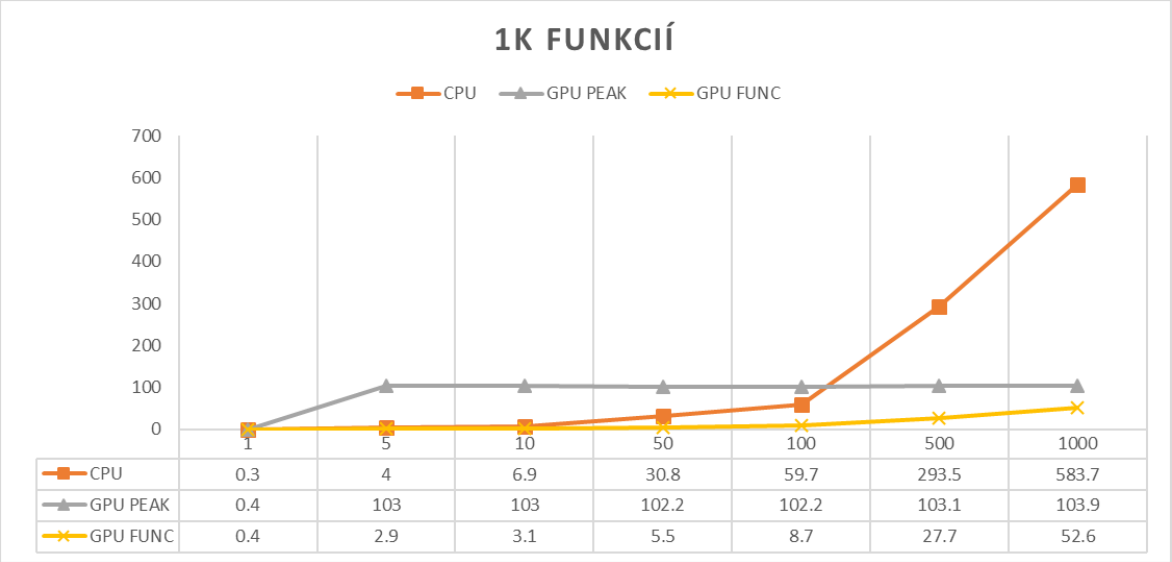
Po ukončení testov sme dospeli k viacerým záverom. Prvým z nich je fakt, že aj keď je naša implementácia Faddejevovej funkcie korektná, spôsob jej paralelného výpočtu zapríčiňuje v extrémnych prípadoch mierne zníženie jej presnosti na úroveň 10^{-14}.



	1000	5000	10000	50k	100k	500k	1 mil
CPU	0.6	3.1	6	30.4	58.7	289.5	577.5
GPU PEAK	0.8	0.9	1.3	3.6	6.5	26.4	50.8
GPU FUNC	1.9	2	2.5	4.8	7.7	28.1	53.4

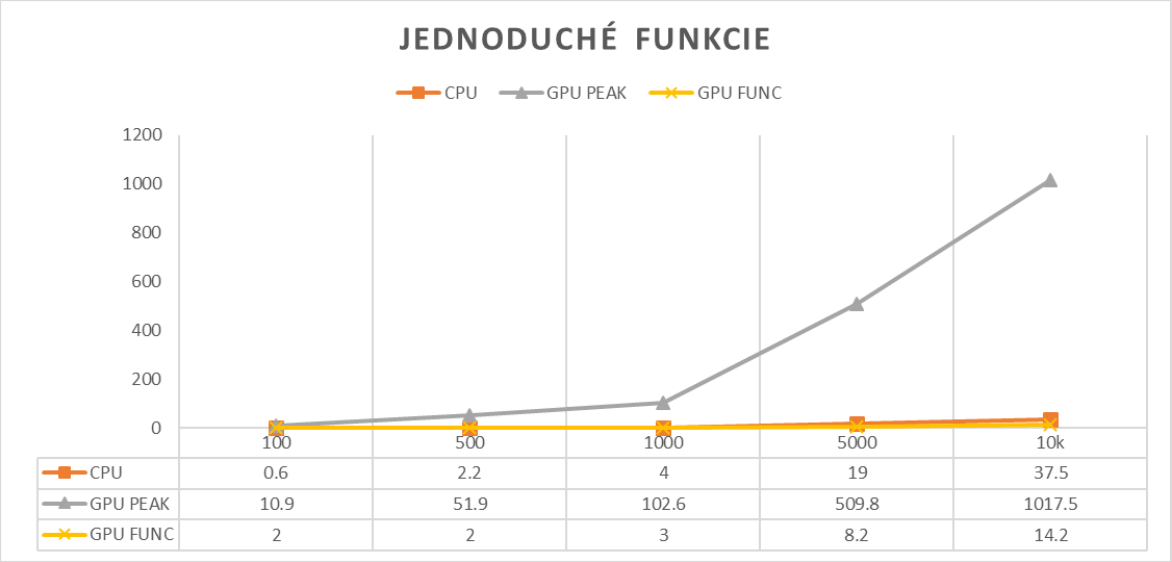
Obr. 5.5: Priemerná dĺžka výpočtu (ms) v závislosti od počtu bodov generovanej HTP spektrálnej čiary

Aj napriek tomu je však presnejšia, než implementácia používajúca starší algoritmus, 'CERNlib_CUDA'. Taktiež sme testovaním našej knižnice zistili, že hlavný potenciál na zrýchlenie výpočtu tkvie v množstve bodov, ktoré jednotlivé spektrálne čiary majú.



	1	5	10	50	100	500	1000
CPU	0.3	4	6.9	30.8	59.7	293.5	583.7
GPU PEAK	0.4	103	103	102.2	102.2	103.1	103.9
GPU FUNC	0.4	2.9	3.1	5.5	8.7	27.7	52.6

Obr. 5.6: Priemerná dĺžka výpočtu (ms) v závislosti od počtu bodov pre 1000 spektrálnych čiar



	100	500	1000	5000	10k
CPU	0.6	2.2	4	19	37.5
GPU PEAK	10.9	51.9	102.6	509.8	1017.5
GPU FUNC	2	2	3	8.2	14.2

Obr. 5.7: Priemerná dĺžka výpočtu (ms) v závislosti od počtu generovaných spektrálnych čiar pre konštantných 5 bodov

Záver

Našim cieľom bolo vytvoriť knižnicu, ktorá by bola schopná paralelných výpočtov grafov v programe Peaks Synthesizer vytvorenej cez aplikáciu LabVIEW. Tieto výpočty mali byť realizované na grafickej karte a riešenie malo využívať jazyk CUDA. Zároveň sme sa mali pokúsiť o porovnanie rôznych spôsobov paralelizácie tohto problému.

Výsledkom našej práce je knižnica schopná pomocou GPGPU vypočítať nasledujúce funkcie: Gauss, Lorentz-Cauchy, Voigt a Hartmann-Trann, tiež známu ako HTP. Táto knižnica je ľahko rozšíriteľná o nové funkcie, ktoré by výskum požadoval. Zároveň sme, zachovajúc pôvodnú kódovú konvenciu a štruktúru knižníc, rozšírili knižnicu stojacu za rozhraním dostupným pre LabVIEW o novú funkciu, ktorá tieto možnosti využíva.

V rámci práce na knižnici sme porovnali dva hlavné prístupy k paralelizácii zadaného problému. Tieto prístupy sa líšia v úrovni, na ktorej používajú GPU. Funkcia 'PeakAdvCUDA' paralelizovala výpočty funkcií a bola cyklicky volaná z CPU. Funkcia 'FuncAdvCUDA' paralelizovala celý výpočet za cenu dodatočnej réžie pamäte. Vytvorili sme aj paralelnú verziu ROOT algoritmu na výpočet Faddejevovej funkcie, 'Cerf_CUDA', ktorý sa dnes v spektrometrii využíva. Tento algoritmus sme porovnali so starším knižničným algoritmom 'CERNlib_CUDA', ktorý je dostupný k voľnému stiahnutiu.

Pre potreby rýchleho testovania sme tiež vytvorili nový program, ktorý je schopný emulovať vstupy prichádzajúce z LabVIEW. Využitie tohto programu by sa mohlo osvedčiť pri testovaní správania programu pri zmenenej definícii konštánt pri volaní jadier, či všeobecnom porovnávaní výkonu jednotlivých výpočtov vo funkciách.

Literatúra

[1] NVIDIA Corporation. CUDA toolkit documentation, 2017. [Citované 2017-12-4] Dostupné z `http://docs.nvidia.com/cuda/`.

[2] Khronos Group. OpenCL overview, 2018. [Citované 2018-2-7] Dostupné z `https://www.khronos.org/opencl/`.

[3] Mark Harris. An even easier introduction to CUDA, 2017. [Citované 2017-12-4] Dostupné z `https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/`.

[4] National Instruments. LabVIEW, 2018. [Citované 2018-5-11] Dostupné z `http://czech.ni.com/labview`.

[5] Ty McKercher John Cheng, Max Grossman. *Professional CUDA C Programming*. Wrox, 2014.

[6] Till Moritz Karbach, Gerhard Raven, and Manuel Schiller. Decay time integrals in neutral meson mixing and their efficient evaluation. *arXiv preprint arXiv:1407.0748*, 2014.

[7] Róbert Lenický. LabVIEW, 2011. [Citované 2018-5-11] Dostupné z `https://itnavody.sk/blogy/software/labview`.

[8] leocorte. Calling Arduino from LabVIEW: Set up and simple analog read, 2014. [Citované 2018-5-11] Dostupné z `https://thebrickinthesky.wordpress.com/2014/12/22/calling-arduino-from-labview-set-up-and-simple-analog-read/`.

[9] David P. Luebke and Greg Humphreys. How GPUs work. *Computer*, 40, 2007.

[10] Z. Obermaier. Nvidia CUDA - několik faktů a zajímavostí, 2012. [Citované 2018-4-17] Dostupné z `https://pctuning.tyden.cz/index.php?option=com_content&view=article&id=25591&catid=1&Itemid=57`.

[11] Ryan Shrout. AGEIA PhysX physics processing unit preview, 2005. [Citované 2018-2-7] Dostupné z `https://web.archive.org/web/20070703170724/http://www.pcper.com/article.php?aid=140`.

[12] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. *ACM SIGARCH Computer Architecture News*, 34(5):325–335, 2006.

[13] Jonathan Tennyson, Peter F Bernath, Alain Campargue, Attila G Császár, Ludovic Daumont, Robert R Gamache, Joseph T Hodges, Daniel Lisak, Olga V Naumenko, Laurence S Rothman, et al. Recommended isolated-line profile for representing high-resolution spectroscopic transitions (iupac technical report). *Pure and Applied Chemistry*, 86(12):1931–1943, 2014.

[14] H Tran, NH Ngo, and J-M Hartmann. Efficient computation of some speed-dependent isolated line profiles. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 129:199–203, 2013.

[15] Peter Čermák. The multifit program for spectroscopic data. *In Depth Solutions with Graphical System Design in Eastern Europe*, pages 105–107, 2012. [Citované 2018-5-13] Dostupné z `http://download.ni.com/pub/branches/ee/2012/nidays/nidays_2012_case_study_booklet.pdf`.

Príloha A

K práci je priložené DVD so zdrojovým kódom vytvorenej knižnice, vstupného rozhrania a upravenou časťou kódu použitou na prepojenie s LabVIEW. Taktiež sa na ňom nachádza príklad funkcie vygenerovanej raz na CPU a raz GPU.