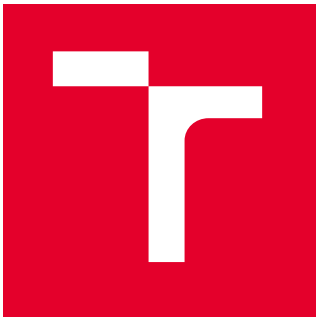


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

AKCELERACE ANALÝZY SENZORICKÝCH DAT NA VESTAVĚNÉM SYSTÉMU S GRAFICKOU KARTOU

ACCELERATED SENSOR DATA ANALYSIS USING AN EMBEDDED SYSTEM WITH A GRAPHICS PROCESSING UNIT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Adam Maczkó**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Adrián Tomašov**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Adam Maczkó ***ID:*** 231252

Ročník: 3 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Akcelerace analýzy senzorických dat na vestavěném systému s grafickou kartou

POKYNY PRO VYPRACOVÁNÍ:

Student se v práci zaměří na možnosti nasazení aplikace na ochranu optických tras/infrastruktur na vestavěný systém Nvidia Jetson Nano. Dodaný projekt je nutné doplnit o podporu graficky akcelerovaných výpočtů a následně vyhodnocovat pomocí dodané neuronové sítě (trénování sítě není součástí zadání). Práce se sestává ze studia dané problematiky, instalace potřebného software, úpravy projektu z numpy do vhodné graficky akcelerovatelné knihovny a vyhodnocení časové úspory. V bakalářské práci se student zaměří na návrh a tvorbu webového grafického rozhraní, které zobrazí spektrogram dat, případně výstup neuronové sítě v reálném čase. Implementované rozhraní bude nasazeno a otestováno na opticke trase se simulovaným provozem.

DOPORUČENÁ LITERATURA:

[1] KETKAR, Nikhil. Introduction to pytorch. In: Deep learning with python. Apress, Berkeley, CA, 2017. p. 195-208.

[2] HALAWA, Hassan, et al. NVIDIA jetson platform characterization. In: European Conference on Parallel Processing. Springer, Cham, 2017. p. 92-105.

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 26.5.2023

Vedoucí práce: Ing. Adrián Tomašov

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Práca sa zaoberá dvoma hlavnými cieľmi, a to akceleráciou analýzy dát a následnou vizualizáciou týchto dát. Účelom je urýchliť dodanú aplikáciu, aby bola vhodná pre ochranu optických infraštruktúr prostredníctvom analýzy zmien stavu polarizácie v reálnom čase a vizualizácie jej výstupov. V práci sú popísané témy dôležité z hľadiska akcelerácie výpočtov na grafickom procesore, najmä princípy paralelizmu, paralelné programovanie, procesy, vlákna a paralelné architektúry. Okrem toho sa v práci popisujú možnosti platformy Jetson Nano. Výstupom práce je aplikácia, ktorá je schopná vykonávať výpočty na grafickom procesore a má webové rozhranie pre vizualizáciu analyzovaných dát. Pre účely akcelerácie na grafickom procesore bola použitá knižnica PyTorch. Vizualizácia dát bola dosiahnutá pomocou frameworku React a knižníc react-spectrogram a ApexCharts.

KĽÚČOVÉ SLOVÁ

akcelerácia spracovania, CUDA, Jetson Nano, paralelné spracovanie, PyTorch, React, spektrogram, vizualizácia, webové rozhranie

ABSTRACT

The thesis deals with two main goals, namely the acceleration of data analysis and the subsequent visualization of this data. The purpose is to speed up the supplied application so that it is suitable for protecting optical infrastructures through real-time analysis of polarization state changes and visualization of its outputs. The thesis describes topics that are important in terms of accelerating computations on the graphics processor, particularly principles of parallelism, parallel programming, processes, threads, and parallel architectures. In addition, the thesis describes the capabilities of the Jetson Nano platform. The output of the thesis is an application that is capable of performing computations on the graphics processor and has a web interface for visualizing the analyzed data. The PyTorch library was used for acceleration on the graphics processor. Visualisation was achieved through the React framework in conjunction with the react-spectrogram and ApexCharts libraries.

KEYWORDS

CUDA, Jetson Nano, parallel processing, processing acceleration, PyTorch, React, spectrogram, visualization, web interface

Vysadené pomocou balíčku `thesis` verzie 4.07; `http://latex.feec.vutbr.cz`

MACZKÓ, Adam. *Akcelerace analýzy senzorických dat na vestavěném systému s grafickou kartou*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023, 52 s. Bakalárska práca. Vedúci práce: Ing. Adrián Tomašov,

Vyhlásenie autora o pôvodnosti diela

Meno a priezvisko autora:	Adam Maczkó
VUT ID autora:	231252
Typ práce:	Bakalárska práca
Akademický rok:	2022/23
Téma závěrečnej práce:	Akcelerace analýzy senzorických dat na vestavěném systému s grafickou kartou

Vyhlasujem, že svoju záverečnú prácu som vypracoval samostatne pod vedením vedúcej/cého záverečnej práce, s využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej záverečnej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto záverečnej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákonníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podpisuje iba v tlačenej verzii.

POĎAKOVANIE

Rád by som poďakoval vedúcemu bakalárskej práce pánovi Ing. Adriánovi Tomašovovi za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Obsah

Zoznam obrázkov

Úvod

Prenos dát prostredníctvom optických sietí je neoddeliteľnou súčasťou modernej sieťovej infraštruktúry. Pri optických sieťach sa naskytuje mnoho otázok a jednou z týchto otázok je aj fyzická ochrana takýchto sietí. Táto bakalárska práca sa zameriava na akcelerované spracovanie dát prostredníctvom grafického procesora na účely analýzy polarizačných zmien. Dáta sú získavané z detektora rýchlych polarizačných zmien, ktorý je určený na ochranu optických infraštruktúr.

Cieľom bakalárskej práce bolo naštudovať problematiku akcelerácie spracovania dát, využiť vhodnú knižnicu pre tieto účely, implementovať triedu schopnú spracovať dáta zo zvukovej karty, vyhodnotiť rozdiel medzi spracovaním na procesore a grafickom procesore a navrhnúť a implementovať grafické užívateľské rozhranie.

Prvým krokom pri dosiahnutí stanovených cieľov bolo naštudovanie špecifikácií a možností vývoja na platforme Jetson Nano. Ďalším dôležitým krokom pri dosiahnutí cieľov bolo naštudovanie konceptu paralelizmu a jeho úrovní, naštudovanie paralelného programovania a jeho modelov a paralelných architektúr, na ktorých sú grafické procesory postavené. Ďalej nasledovalo porovnanie knižníc, ktoré poskytujú grafickú akceleráciu. Po zvolení vhodnej knižnice mohlo prebehnúť implementovanie pôvodnej aplikácie do tejto knižnice. Túto časť sprevádzali mnohé ťažkosti, ale tie sa nakoniec podarilo vyriešiť. Po úspešnej implementácii do vhodnej knižnice bolo potrebné vykonať porovnanie časov spracovania pôvodnej aplikácie a graficky akcelerovanej aplikácie. Následne bolo potrebné navrhnúť a implementovať grafické užívateľské rozhranie. Aplikáciu doplnenú o grafické užívateľské rozrhranie bolo potrebné otestovať na optickej trase so simulovanou prevádzkou a zhodnotiť jej využiteľnosť pre monitorovanie trasy v reálnom čase.

Dokument je členený do piatich kapitol. Prvá kapitola oboznamuje čitateľa s konceptom paralelizmu. Táto kapitola sa taktiež zaoberá paralelným programovaním, jeho modelmi, procesmi a vláknami, ktoré sú neoddeliteľnou súčasťou paralelného spracovania. Druhá kapitola opisuje paralelné architektúry. Sú tu rozoberané najmä organizácie pamäte paralelných počítačov a siete prepojení výpočtových jednotiek. Ďalšou kapitolou je kapitola zaoberajúca sa platformou Jetson Nano. Rozoberajú sa tu technické špecifikácie, možnosti vývoja a nástroj jtop. Štvrtá kapitola sa zaoberá popisom fungovania dodaného systému. V kapitole Akcelerácia analýzy a vizualizácia dát sú popísané jednotlivé knižnice použité pri akcelerácii a ich porovnanie. V tejto kapitole je opísaný spôsob akcelerovania analýzy, porovnanie rýchlosti spracovania medzi implementáciou s pôvodnou knižnicou a graficky akcelerovanou knižnicou. V ďalšej časti tejto kapitoly je opísaný proces vytvárania grafického užívateľského rozhrania aj so všetkými použitými knižnicami. Na záver je tu uvedené ako prebiehalo testovanie aplikácie na optickej trase so simulovanou prevádzkou.

1 Paralelizmus

Paralelizmus je názov charakterizujúci javy, ktoré sa vo výpočtových zariadeniach vykonávajú zároveň. Paralelizmus je možné skúmať na viacerých úrovniach v závislosti na požadovanej úrovni zložitosti [1].

1.1 Úrovne paralelizmu

Paralelizmus sa rozdeľuje na päť úrovní, a to na úroveň *úloh (job), metód (task), procesov (process), premenných/inštrukcií (variable)* a *bitovú (bit)* úroveň [1].

Úroveň úloh – popisuje rôzne na sebe nezávislé úlohy, ktorých súbežné riešenie neovplyvní výsledky ostatných riešených úloh. Príkladom takýchto úloh môžu byť dva počítačové programy bežiace na jednom počítači, z ktorých jeden rieši sčítanie dvoch čísel a ďalší rieši násobenie dvoch čísel. Každý z týchto programov je možné považovať za úlohu a tieto úlohy riešiť zároveň za predpokladu, že existuje dostatok výpočtových prostriedkov.

Úroveň metód – za metódy sa považujú jednotlivé časti úloh, ktoré sú zodpovedné za vykonávanie rôznych funkcionalít. Možným príkladom takýchto metód môže byť meteostanica, ktorej program vyhodnocuje rôzne atribúty prostredia ako napríklad teplotu, vlhkosť a rýchlosť vetra. Program meteostanice predstavuje úlohu, ktorú je možné rozdeliť na niekoľko metód zodpovedných za vyhodnocovanie jednotlivých atribútov prostredia. Tieto metódy alebo ich časti je potom možné vykonávať súbežne aj v prípade keby medzi týmito metódami existovala interakcia.

Úroveň procesov – popisuje delenie jednotlivých metód na ďalšie časti, ktoré sú vykonávané súčasne. Takéto delenie metód môže byť vykonané manuálne používateľom alebo automaticky kompilátorom. Príkladom takéhoto paralelizmu môže byť cyklus na obrázku 1.1. Tento cyklus je možné rozdeliť na dva procesy. Keďže výpočet premennej x ani výpočet premennej y nie sú na sebe závislé, tak je možné vykonať výpočet x v jednom procese a výpočet y v ďalšom.

Úroveň premenných/inštrukcií – jednotlivé procesy sa skladajú z viacerých inštrukcií, ktoré získavajú výstupné hodnoty zo vstupných hodnôt. Paralelizácia pri premenných spočíva v súbežnom vykonávaní inštrukcií procesov, avšak nesmú medzi nimi existovať závislosti. Existencia závislostí medzi inštrukciami odstraňuje možnosť ich súbežného vykonania. Závislosti je možné členiť nasledovne [2].

- **Skutočná závislosť** vzniká v prípade ak inštrukcia I_2 využíva k výpočtom premenné, do ktorých sa ukladajú výsledky inštrukcie I_1. Inými slovami I_2 využíva premenné, ktorých hodnota závisí na inštrukcii I_1.
- **Falošná závislosť** vzniká keď inštrukcia I_1 využíva vo svojich výpočtoch premenné, do ktorých I_2 zapisuje svoje výsledky. Jedná sa o závislosť operandov I_1 na výsledkoch I_2.

- **Závislosť pri výsledkoch** vzniká ak inštrukcia I_1 aj inštrukcia I_2 zapisujú výsledky svojich operácií do tej istej premennej. Čiže jedna inštrukcia prepíše výsledok druhej.

Bitová úroveň - predstavuje najnižšiu úroveň paralelizácie a spočíva v súbežnom vykonávaní bitovej aritmetiky, čo znamená, že operácie nad bitmi najmenšej adresovateľnej jednotky (word) sú vykonávané súbežne. Táto úroveň paralelizácie je implementovaná na úrovni procesora [2].

1: $x \leftarrow 10$
2: $y \leftarrow 0$
3: **for** $i \leftarrow 1$ to 5 **do**
4: $\quad x \leftarrow x - i$
5: $\quad y \leftarrow y + i$
6: **end for**

Obr. 1.1: Príklad paralelizácie cyklu.

1.2 Procesy a vlákna

Proces predstavuje entitu, ktorá je základným elementom vykonávajúcim úlohu v operačnom systéme. Proces je možné si predstaviť ako vykonávaný program, ktorého inštrukcie sú vykonávané sekvenčne. Každý proces má priradenú pamäť, ktorá pozostáva zo *zásobníka (stack), haldy (heap), textovej časti (text)* a *dátovej časti (data)* [3].

- **Zásobník** – obsahuje všetky lokálne premenné.
- **Halda** – využíva sa na dynamickú alokáciu pamäte.
- **Text** – predstavuje skompilovaný kód programu (inštrukcie).
- **Dáta** – obsahuje globálne a statické premenné.

Procesy je možné vytvárať prostredníctvom systémových volaní priamo z iného procesu, potom sa takto vytvorený proces označuje ako „podproces“ (child process) a proces, z ktorého bol vytvorený sa označuje ako „rodičovský“ (parent process). Jedným z možných spôsobov ako vytvoriť nový proces je systémové volanie `fork()`. Pri takto vytvorenom procese je podproces duplikátom rodičovského procesu a zdieľajú virtuálnu pamäť (stránky), ktoré sú označené ako `copy-on-write`. V prípade, že sa niektorý z procesov pokúsi zmeniť zdieľané stránky, tak sa vytvorí kópia týchto stránok a zmeny sa vykonajú iba na týchto skopírovaných stránkach [4, 5]. Proces prechádza počas svojej existencie rôznymi stavmi.

Tieto stavy sa môžu líšiť v závislosti na operačnom systéme, avšak všeobecne sa tieto stavy môžu označovať nasledovne [3]:

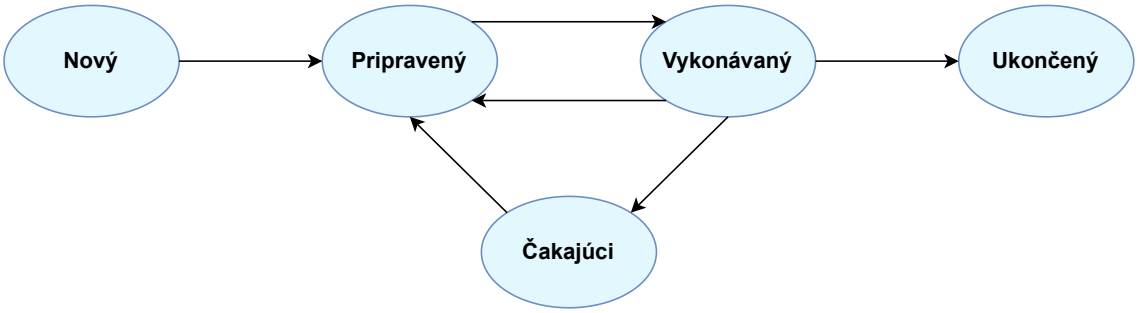
- **Nový** (Start) – počiatočný stav, nastáva hneď pri vytvorení procesu.
- **Pripravený** (Ready) – proces čaká aby mu plánovač operačného systému priradil procesor, aby sa mohol dostať do stavu **vykonávaný**. Do stavu **pripravený** sa môže proces dostať zo stavu **nový** alebo zo stavu **vykonávaný**.
- **Vykonávaný** (Running) – stav, v ktorom plánovač pridelil procesu procesor a proces vykonáva svoje inštrukcie.
- **Čakajúci** (Waiting) – proces v stave **čakajúci**, čaká na nejaké zdroje ako napríklad užívateľský vstup.
- **Ukončený** (Terminated) – do tohto stavu sa proces dostane keď dokončí svoju úlohu alebo je ukončený operačným systémom.

Vlákno (thread) je oddelená cesta vykonávania inštrukcií v rámci procesu. Predstavuje odľahčený proces, ktorého vykonávanie môže operačný systém plánovať zároveň s ostatnými vláknami v rámci procesu. Narozdiel od procesov, ktoré majú každý pridelený vlastný adresný priestor v pamäti, vlákna zdieľajú jeden adresný priestor v pamäti procesu, čo znamená, že nie sú na sebe nezávislé ako procesy. Toto umožňuje vláknam efektívne spolupracovať a komunikovať. Ďalším obsahom vlákien je aj oddelený runtime zásobník (runtime stack), ktorý slúži pre kontrolovanie funkcií, za ktoré je dané vlákno zodpovedné a pre ukladanie lokálnych premenných. Dáta v lokálnych premenných nie sú priamo prístupné z ostatných vlákien. Runtime zásobník je ukončený spolu s ukončením vlákna, z tohto dôvodu nie je odporúčané predávať referencie na lokálne premenné jedného vlákna vláknu druhému [2]. Podobne ako procesy aj vlákna majú rôzne stavy.

Jeden proces môže mať niekoľko vlákien, ktoré bežia paralelne, čím zvyšujú efektivitu programu z hľadiska využívania zdrojov. Vlákna sa delia na dva typy, a to *užívateľské vlákna* (user-level threads) a *vlákna jadra operačného systému* (kernel--level threads) [6].

Užívateľské vlákna – predstavujú vlákna vytvorené užívateľským softvérom prostredníctvom knižníc pre vytváranie, správu a synchronizáciu týchto vlákien. Užívateľské vlákna existujú len v rámci daného procesu, čo znamená, že sa nemôžu odkazovať na užívateľské vlákna v iných procesoch. Tieto vlákna sú rýchlejšie ako vlákna jadra operačného systému [7, 8].

Vlákna jadra – sú realizačnou jednotkou v rámci procesu na úrovni operačného systému. Tieto vlákna sú spravované priamo jadrom operačného systému, čo znamená, že programátor nad nimi nemá žiadnu kontrolu. Na vlákna jadra sa môže odkázať akékoľvek iné vlákno [7, 8].



Obr. 1.2: Obrázok znázorňujúci stavy procesov a prechody medzi nimi [3].

1.2.1 Výmena informácií

Pre kontrolu koordinácie jednotlivých častí paralelného programu je potrebné, aby prebiehala komunikácia medzi jednotlivými časťami. Táto komunikácia je zväčša závislá na organizácii pamäte danej platformy. Vyskytujú sa najmä systémy so zdieľaným adresným priestorom alebo distribuovaným adresným priestorom [2].

Pri platformách s distribuovanou pamäťou má každý procesor vlastnú pamäť a nemá prístup do pamäti ostatných procesorov. Komunikácia medzi procesormi v takomto prostredí prebieha pomocou odovzdávania správ. Odovzdávanie správ medzi procesormi prebieha pomocou páru správ. Odosielajúci procesor vykoná operáciu odoslania, ktorá odošle dátový blok z vlastného adresného priestoru, prijímajúci procesor vykoná operáciu prijatia, ktorá prijme prichádzajúci dátový blok. V prípade, že prijímajúci procesor nevykoná operáciu prijatia nastáva uviaznutie (deadlock), pretože odosielajúci procesor čaká na prijatie [2].

Pri platformách so zdieľaným adresným priestorom existuje jedna globálna pamäť, ktorá je prístupná pre všetky procesory. Jednotlivé procesory alebo v prípade viac jadrových procesorov jadrá, vykonávajú procesy alebo vlákna. Každý proces alebo vlákno môže pristupovať k zdieľaným dátam v zdieľanej pamäti. Zdieľané dáta sú uložené v zdieľaných premenných a pomocou nich sa zaisťuje komunikácia medzi vláknami [2].

Aby bol umožnený bezpečný a koordinovaný prístup viacerých procesov alebo vlákien ku zdieľaným premenným je potrebné zaistiť synchronizáciu (pozri tiež sekciu 1.2.2). Pri súčasnom prístupe viacerých vlákien k zdieľaným premenným môže nastať súbeh. **Súbeh (race condition)** vyjadruje situáciu, pri ktorej sa výsledok paralelného programu mení na základe poradia, v ktorom vlákna vykonávajú svoje časti programu. Časti programu, v ktorých môžu nastať takéto problémy sa nazývajú **kritické sekcie**. Súbehu sa dá predísť tým, že kritickú sekciu môže vykonávať vždy iba jedno vlákno, toto sa nazýva **vzájomné vylúčenie (mutual exclusion)**. V niektorých programovacích modeloch existujú mechanizmy zámkov, ktorých účelom je predchádzať súbehom [2].

Uviaznutie (deadlock) je situácia, pri ktorej každý proces z množiny procesov čaká na uvoľnenie prostriedku, ku ktorému má výlučný prístup iba jeden proces z tejto množiny, a preto ani jeden proces z tejto množiny nemôže pokračovať vo svojej činnosti [9].

1.2.2 Mechanizmy synchronizácie

Paralelné vykonávanie vlákien a procesov nesie riziká v podobe súbehov. Pre predchádzanie týchto rizík je potrebné koordinovať toto vykonávanie. Koordinácia paralelného vykonávania znamená, že neexistujú dva procesy, ktoré by mohli pristúpiť k zdieľaným zdrojom v rovnaký časový okamih. Synchronizácia je dôležitá kvôli predchádzaniu nedterministického správania a aby bolo možné zachovať konzistenciu zdieľaných zdrojov [10]. Koordináciu paralelného vykonávania procesov alebo vlákien je potrebné zabezpečiť napríklad pre zaručenie určitého poradia, v ktorom sa majú vlákna vykonávať alebo pre kontrolu prístupu k zdieľaným premenným [2].

Synchronizácia pomocou zámkov

K zabráneniu súbehov plynúcich zo súbežného prístupu k zdieľaným premenným je možné využiť tzv. „zámky“, ktoré pozostávajú z preddefinovaných premenných, ktoré sa nazývajú premenné vzájomného vylúčenia (mutual exclusion variables). Tieto premenné majú dva možné stavy, a to „zamknutý“ a „odomknutý“.

Vlákno, ktoré vstupuje do svojej kritickej sekcie uzamkne prístup k zdieľanej premennej, aby túto premennú nemohli meniť ostatné vlákna. Vlákno zámok odomkne až po vykonaní svojej kritickej sekcie. Týmto umožní ďalšiemu vláknu vstúpiť do kritickej sekcie. Takýmto spôsobom sa nemôže stať, že dve vlákna vstupujú do svojej kritickej sekcie zároveň, čím sa predchádza súbehom [11].

Semafory

Ďalším mechanizmom pre zabezpečenie vzájomného vylúčenia sú semafory. Semafor predstavuje dátovú štruktúru pozostávajúcu z celočíselného počítadla a dvoch atomických operácií P a V vykonávaných nad počítadlom. Existujú dva typy semaforov: *binárny semafor* (binary semaphore) a *počítací semafor* (counting semaphore). Binárny semafor môže nadobúdať hodnôt 1 alebo 0 [12].

Operácia P čaká kým je hodnota počítadla väčšia ako nula, v prípade, že sa táto podmienka naplní, tak sa hodnota počítadla dekrementuje o 1 a vlákno môže vstúpiť do svojej kritickej sekcie. Po vykonaní svojej kritickej sekcie sa vykoná operácia V, ktorá zvýši hodnotu počítadla o 1 [12].

Monitory

Monitor je ďalším prostriedkom pre vzájomné vylúčenie. Monitor je abstraktná dátová štruktúra, ktorá definuje dátové štruktúry a prístupové metódy k nim. Pristupovať k dátam v týchto dátových štruktúrach je možné iba prostredníctvom prístupových metód monitora. Monitor zabezpečuje, vzájomné vylúčenie vlákien [13]. Inými slovami, k dátam môže pristúpiť najviac jedno vlákno v jeden časový okamih.

1.3 Paralelné programovanie

Paralelné programovanie spočíva v hľadaní takých častí problému, ktoré je možné bezpečne vykonávať v jeden časový okamih. Väčšina výpočtovo náročných problémov obsahuje takéto časti a je výhodné takéto problémy rozdeliť na menšie časti, podproblémy, ktoré sú následne vykonávané súbežne. Takéto delenie výpočtovo náročných problémov v kombinácii s efektívnymi paralelnými algoritmami a vhodnou paralelnou architektúrou hardvéru má za následok zníženie času potrebného na vyriešenie týchto problémov. Cieľom paralelného programovanie je využiť naraz viac procesorov aby sa znížil čas potrebný pre vyriešenie problémov [14].

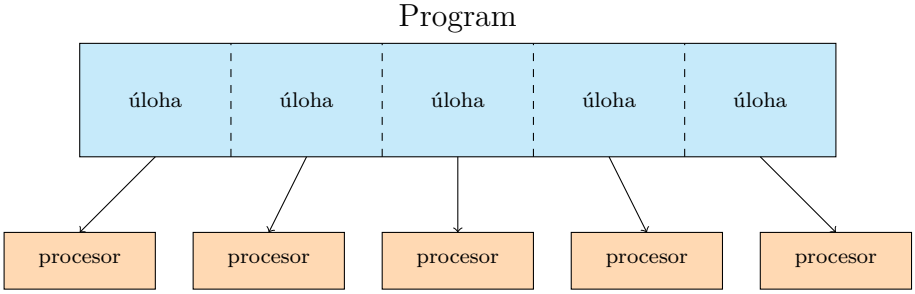
1.3.1 Paralelizácia programov

Paralelizácia je proces, pri ktorom sa program alebo algoritmus vykonávaný sekvenčne prevádza na program alebo algoritmus, ktorého časti sú vykonávané súbežne. Pri paralelizácii je nutné brať do úvahy všetky možné závislosti, ktoré by takémuto vykonávaniu zabránili. Pre správne vykonanú paralelizáciu je nutné zabezpečiť aby výsledky paralelizovaného programu boli konzistentné. Paralelizácia spočíva v nasledujúcich krokoch *dekompozícia, určenie úloh pre jednotlivé procesy a vlákna, priradenie procesov a vlákien jednotlivým jadrám procesorov* [15]. Proces paralelizácie je možné vidieť na obrázku 1.3.

Dekompozícia – cieľom dekompozície je rozložiť program na taký počet úloh, aby nenastal stav, pri ktorom sú niektoré jadrá procesorov nečinné. Zároveň je dôležité klásť dôraz na výpočtový čas jednotlivých úloh, *granularitu*, čiže aby čas potrebný na priradenie úlohy procesu a priradenie procesoru nepresiahol čas vykonávania úlohy.

Určenie úloh pre jednotlivé procesy a vlákna – spočíva v priradení jednotlivých úloh procesom alebo vláknam tak aby každý proces mal približne rovnaký počet výpočtov. Do úvahy treba brať aj prístup do pamäte alebo komunikáciu medzi procesmi.

Priradenie procesov a vlákien jednotlivým jadrám procesorov – jednotlivým fyzickým procesorom alebo jadrám sú priradené procesy a vlákna tak aby boli rovnomerne zaťažené a aby bola komunikácia medzi nimi čo najmenšia.



Obr. 1.3: Znázornenie procesu paralelizácie programu.

Zreťazené spracovanie

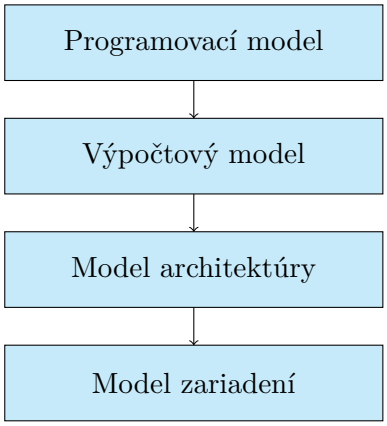
Zreťazené spracovanie je prístup k paralelizácii programov, pri ktorom sú výpočty rozdelené do oddelených ale zreťazených elementov. Úlohy elemetov sú vykonávané v určitom poradí, tak aby bolo možné použiť výstupy predchádzajúceho elementu ako vstupy nasledujúceho elementu. Tieto elementy sú implementované ako procesy alebo ako vlákna a sú vykonávané súbežne [16].

Dôležitým aspektom zreťazeného spracovania je dôraz na to aby boli procesy obsadené čo najväčšie množstvo času, a preto je pri tomto prístupe často využívaný buffering, ktorý umožňuje aby si jednotlivé procesy predpočítali hodnoty, s ktorými sú vykonávané výpočty nasledujúceho procesu. Za týmto účelom sú využívané najmä fronty. Tieto fronty slúžia ako prepájajúci element medzi jednotlivými procesmi, ktoré si takto predávajú dáta [16]. Inak povedané, fronty je možné si predstaviť ako „rúry“, cez ktoré „tečú“ dáta z jedného procesu do ďalšieho.

1.3.2 Modely paralelného programovania

Modely paralelného programovanie (pozri obr. 1.4) popisujú paralelné spracovanie na jednotlivých úrovniach abstrakcie. Existujú štyri základné modely a to model *zariadení (machine), architektúry (architecture), výpočtov (computation)* a *programovací (programming)* model [2].

Model zariadení sa nachádza na najnižšej úrovni abstrakcie a popisuje hardvér a operačný systém. Na tejto úrovni sa nachádzajú jazyky symbolických adries. Na ďalšej úrovni abstrakcie sa nachádza **model architektúry**, ktorý popisuje sieť prepojení paralelných platforiem, či je použitá synchrónna alebo asynchrónna metóda spracovania a organizáciu pamätí. O úroveň abstrakcie vyššie sa nachádza **výpočtový model**, jedná sa o funkciu nákladov, ktorá odráža čas potrebný na vykonanie algoritmu na hardvéri určenom modelom architektúry. Na najvyššej úrovni abstrakcie sa nachádza **programovací model**. Tento model popisuje systém paralelného spracovania na základe programovacieho jazyka. Špecifikuje ako programátor dokáže implementovať algoritmus [2].

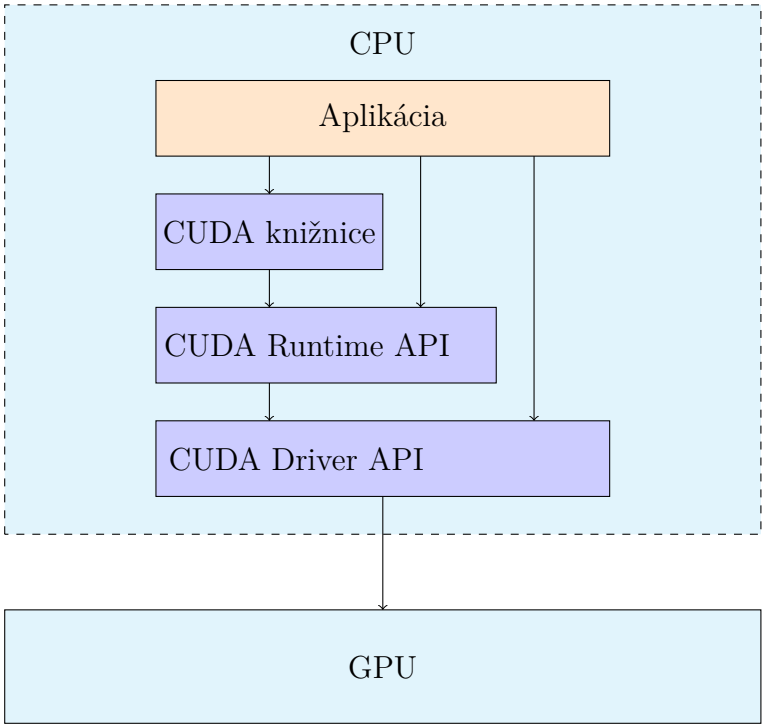


Obr. 1.4: Jednotlivé modely paralelného programovania zoradené podľa úrovne abstrakcie od najvyššej po najnižšiu.

1.3.3 Programovací model CUDA

CUDA (Compute Unified Device Architecture) (pozri obr. 1.5) je heterogénny programovací model vyvinutý spoločnosťou NVIDIA. Heterogénny znamená, že využíva ako procesor (CPU), tak aj grafický procesor (GPU). Tento programovací model rozdeľuje výpočtové jednotky na *hostiteľa (host)* a *zariadenie (device)*. Za hostiteľa sa považuje CPU, za zariadenie sa považuje GPU. Na hostiteľovi je vykonávaný program, ktorý môže pristupovať do pamäte hostiteľa ako aj zariadenia. Tento program dokáže využívať funkcie zariadenia, ktoré sa v CUDA terminológii označujú ako tzv. „kernely“. Tieto kernely pristupujú k zdrojom zariadenia prostredníctvom CUDA rozhraní [17]. Každé volanie kernelov má za následok vytvorenie veľkého počtu CUDA vlákien [2].

Programovací model CUDA poskytuje rôzne CUDA knižnice a dve softvérové rozhrania pre prístup k zdrojom zariadenia a to CUDA Runtime API a CUDA Driver API. Jedná sa o navzájom vylučujúce sa rozhrania, nie je ich možné kombinovať v rámci programu. Tieto rozhrania ponúkajú takmer to isté s tým rozdieľom, že v prípade CUDA Runtime API sa jedná o rozhranie vyššej vrstvy abstrakcie, čiže je užívateľsky prívetivejšie ako CUDA Driver API. CUDA Driver API je rozhraním na nižšej vrstve abstrakcie v porovnaní s CUDA Runtime API, avšak ponúka väčšiu kontrolu nad vykonávanými operáciami. Je nutné spomenúť, že všetky volania CUDA Runtime API sú preložené do inštrukcií CUDA Driver API, ktoré sú následne vykonávané na zariadení [18, 19].



Obr. 1.5: Štruktúra programovacieho modelu CUDA [18].

2 Paralelné architektúry

V priebehu rokov bolo vyvinutých veľa rôznych paralelných architektúr a ich následných implementácii v paralelných počítačoch. Paralelný počítač je možné charakterizovať ako súbor výpočtových elementov komunikujúcich medzi sebou a spolupracujúcich na riešení problémov [2].

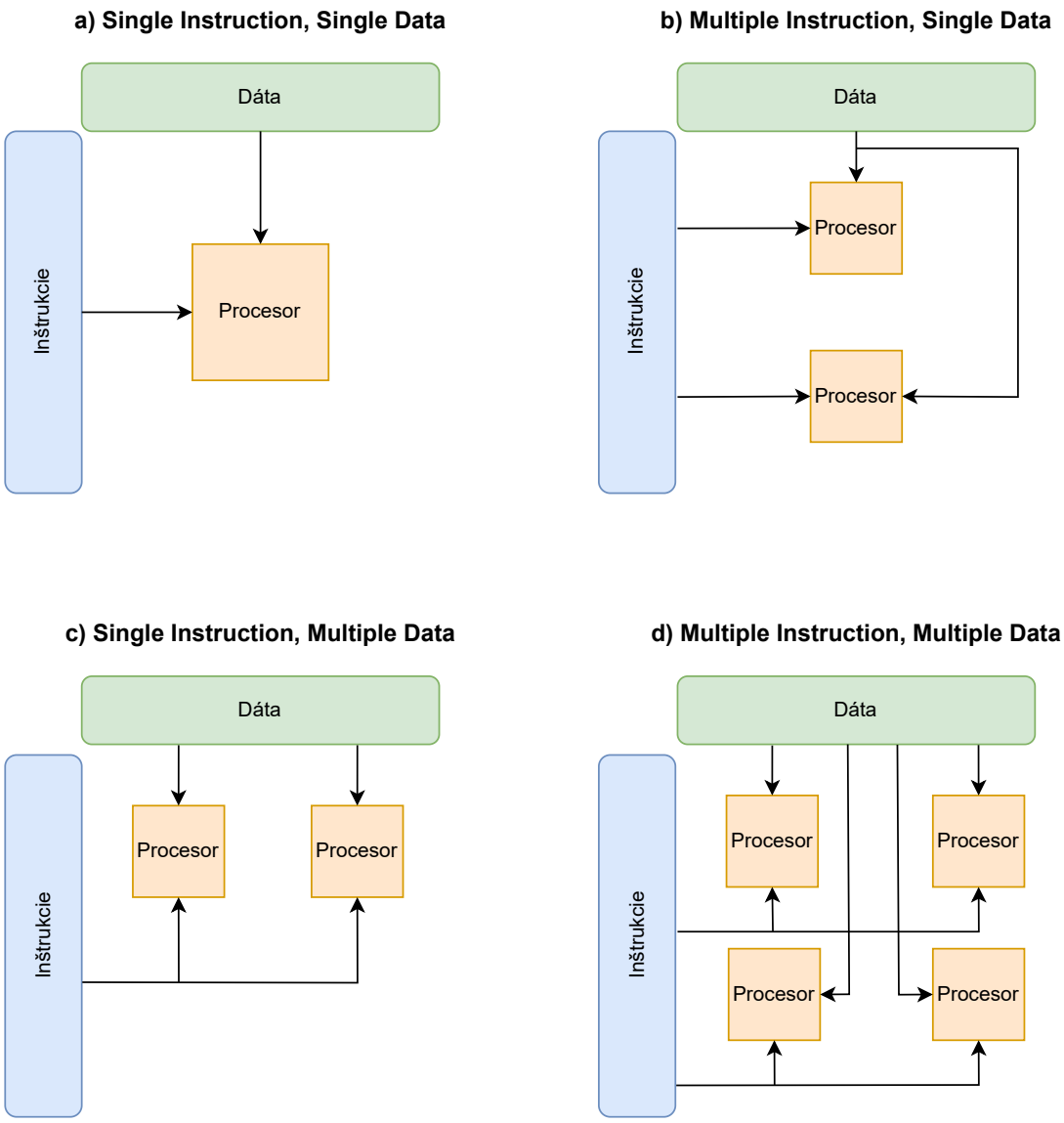
Paralelné architektúry je možné klasifikovať na základe tzv. Flynnovej klasifikácie (pozri obr. 2.1). Podľa tejto klasifikácie existujú štyri kategórie [2].

Single Instruction, Single Data (SISD) – pri tejto architektúre je k dispozícii jediný výpočtový element, ktorý v každom kroku načíta inštrukciu a prislúchajúce dáta a následne vykoná inštrukciu. Počítače postavené na tejto architektúre sú sekvenčné počítače podľa von Neumannovho modelu [20].

Multiple Instruction, Single Data (MISD) – táto architektúra sprístupňuje viacero výpočtových elementov s vlastnou programovou pamäťou, avšak existuje iba jedna globálna dátová pamäť. V každom kroku dostane každý element rovnaké dáta z globálnej dátovej pamäte a načíta inštrukciu zo svojej programovej pamäte, ktorá môže byť odlišná ako inštrukcie ostatných elementov. Následne túto inštrukciu vykoná súbežne s ostatnými elementami [20].

Single Instruction, Multiple Data (SIMD) – pri tejto architektúre je k dispozícii viacero výpočtových elementov, ktoré majú prístup k súkromnej dátovej pamäti, avšak existuje iba jedna programová pamäť. Z tejto programovej pamäte načítava inštrukcie špeciálny procesor. Pri každom kroku tento procesor predáva rovnakú inštrukciu jednotlivým elementom, ktoré si načítajú dáta zo svojej dátovej pamäte. Čiže tá istá inštrukcia je vykonávaná synchrónne a v jeden okamih na rôznych výpočtových elementoch nad rôznymi dátami [20].

Multiple Instruction, Multiple Data (MIMD) – architektúra poskytuje viacero výpočtových elementov, z ktorých každý má k dispozícii privátnu programovú aj dátovú pamäť. V každom kroku si každý výpočtový element načíta vlastnú inštrukciu a vlastný dátový element nad ktorým bude inštrukcia vykonávaná. Výpočtové elementy pracujú asynchrónne vzhľadom na ostatné výpočtové elementy [20].



Obr. 2.1: Klasifikácia paralelných architektúr podľa M. J. Flynna. Jednotlivé časti predstavujú paralelné architektúry popísané vyššie v texte. Vytvorené podľa [20].

2.1 Organizácia pamäte

Paralelné počítače sú vo veľkej väčšine prípadov založené na paralelnej architektúre typu MIMD. Tieto počítače je možné ďalej rozdeliť podľa organizácie pamäte a to z hľadiska fyzického usporiadania pamäte a z hľadiska pohľadu programátora na pamäť. Podľa fyzickej organizácie pamäte existujú paralelné počítače s fyzicky *distribuovanou* alebo *zdieľanou* pamäťou [2].

Z hľadiska pohľadu programátora existujú paralelné počítače s distribuovaným adresným priestorom alebo zdieľaným adresným priestorom, ktoré nemusia nutne korešpondovať s fyzickým usporiadaním pamäte. Je možné, že počítač s fyzicky distribuovanou pamäťou má z hľadiska programátora zdieľaný adresný priestor alebo naopak počítač s fyzicky zdieľanou pamäťou má distribuovaný adresný priestor [2].

2.1.1 Počítače s fyzicky distribuovanou pamäťou

Počítače s fyzicky distribuovanou pamäťou pozostávajú z elementov nezávislých od seba, tieto elementy sa môžu tiež nazývať uzly. Každý uzol je samostatná jednotka a má vlastný procesor, programovú pamäť a dátovú pamäť, do ktorej ostatné uzly nemajú prístup. V prípade, že uzol potrebuje dáta od iného uzlu, vyžiada si ich pomocou predávania správ. Komunikácia medzi uzlami je zaručená prostredníctvom siete prepojení medzi uzlami. Topológia siete prepojení je veľmi dôležitá pri takýchto systémoch, keďže je to obmedzujúci faktor pri komunikácii medzi uzlami [21].

Takýto systém je možné si predstaviť ako lokálnu sieť, kde jednotlivé uzly predstavujú stolné počítače, ktoré medzi sebou komunikujú a všetky tieto počítače využívajú ten istý operačný systém [2].

2.1.2 Počítače s fyzicky zdieľanou pamäťou

Počítače s fyzicky zdieľanou pamäťou pozostávajú z niekoľkých procesorov, globálnej pamäte a sieti prepojení, ktorá zabezpečuje komunikáciu procesorov s pamäťou. Komunikácia medzi procesormi v takomto systéme je zaručená prostredníctvom zdieľaných premenných v zdieľanej pamäti. Výhodou takejto komunikácie je fakt, že komunikácia prostredníctvom zdieľaných premenných je jednoduchá a dáta nemusia byť duplikované, čo sa môže stať v prípade systémov s distribuovanou pamäťou. Pri systémoch so zdieľanou pamäťou môže nastať pri súčasnom prístupe ku zdieľaným premenným súbeh [21].

Realizácia týchto systémov je náročnejšia v porovnaní so systémami s distribuovanou pamäťou, pretože sieť prepojení musí poskytovať všetkým procesorom dostatočne rýchlu komunikáciu s pamäťou [2].

2.2 Siete prepojení

Jednu z najkritickejších častí paralelných systémov tvorí komunikácia medzi výpočtovými elementami. Pri paralelných systémoch je prepojenie medzi jednotlivými elementami zabezpečené prostredníctvom siete prepojení. Ideálnym prípadom je keď existuje cesta z každého elementu do každého elementu. Sieť prepojení musí zabezpečiť aby bola komunikácia čo najrýchlejšia a zároveň spoľahlivá, a to aj v prípade, že komunikuje viac elementov naraz. Nedostatočná rýchlosť alebo spoľahlivosť siete môže mať za následok významný pokles výkonu paralelného systému.

Siete prepojení je možné rozdeliť na *statické* a *dynamické*, podľa spôsobu ako sú jednotlivé elementy prepojené [2].

2.2.1 Statické siete prepojení

V prípade statických je sieť prepojení daná topológiou, z čoho vyplýva, že pri týchto sieťach sú prepojenia medzi výpočtovými elementami pevne dané a nie je ich možné meniť. Takýmto sieťam sa hovorí aj ako siete typu bod–bod.

Statické siete je možné matematicky reprezentovať pomocou neorientovaného grafu $G = (V, E)$, kde množina vrcholov V grafu G predstavuje jednotlivé výpočtové elementy siete a množina hrán E predstavuje jednotlivé fyzické prepojenia medzi elementami. Príklady statickej topológie siete prepojení môžu byť úplný lineárny graf, presteň (ring), úplný graf, n-dimenzionálna kocka a binárny strom [2, 22].

Pri topológii typu **lineárny graf** je jednotlivým výpočtovým elementom priradené poradové číslo, na základe ktorého sú zoradené vzostupne. Potom má každý element dvoch susedov, predchodcu a nasledovníka, okrem prvého a posledného elementu, ktoré majú buď len predchodcu alebo nasledovníka. Pri každom elemente existuje priame prepojenie len s jeho susedmi. Nevýhodou takejto siete je, že v prípade komunikácie prvého a posledného elementu musí správa putovať cez všetky medziľahlé elementy, čo má za následok relatívne vysokú odozvu pri komunikácii. Prepojením prvého a posledného elementu lineárneho grafu vzniká topológia typu **prsteň** a prepojením každého elementu s každým vzniká topológia typu **úplný graf** [22].

Topológia typu **n-dimenzionálna kocka** vyžaduje aby počet výpočtových elemetov bol n-tou mocninou čísla 2, kde n predstavuje dimenziu a platí $n \geq 0$, potom každý element má presne n susedov. Elementy sú označené číslom $0, \ldots, 2^n - 1$ zapísaných v binárnej reprezentácii. Susedné elementy spĺňajú podmienku, že binárna reprezentácia ich označenia sa líši presne v jednej jednotke. Inými slovami Hammingova váha ich označení je rovná jednej [22].

Ďalším spôsobom ako prepojiť procesory je **binárny strom**, ktorý je najčastejšie implementovaný ako **úplný binárny strom**[1], ktorý má $N = 2^k - 1$ vrcholov (procesorov) a k úrovní. V tejto topológii existujú tri druhy procesorov na základe ich umiestnenia. Koreňový procesor je taký, ktorý nemá predchodcu a má maximálne dvoch nasledovníkov. Vnútorný procesor je taký, ktorý má dvoch nasledovníkov a jedného predchodcu. Listový procesor je taký, ktorý má len predchodcu. Procesory sú zväčša očíslované začínajúc od koreňového procesora, ktorý je označený jednotkou. Nasledovníci procesoru x sa označujú ako $2x$ pre ľavého nasledovníka a $2x + 1$ pre pravého nasledovníka [2].

2.2.2 Dynamické siete prepojení

Pri dynamických sieťach jednotlivé výpočtové elementy nie sú priamo prepojené jeden s druhým. Namiesto priameho prepojenia medzi jednotlivými elementami sú v takýchto sieťach prepínače. Prepínače sú medzi sebou prepojené fyzickými linkami a zaisťujú komunikáciu medzi jednotlivými výpočtovými elementami systému, na základe čoho je možné povedať, že topológia je tvorená prepínačmi. Dynamické siete prepojení je možné kategorizovať na základe topologickej štruktúry na *zbernicové siete (bus networks), viacúrovňové siete (multistage networks)* a *priečkové siete (crossbar networks)* [2].

Zbernicové siete sú založené na zberniciach, ktoré sa skladajú z veľkého počtu vodičov, cez ktoré sa prenášajú dáta veľkou rýchlosťou. V každý okamih môže prebiehať iba jeden prenos. V prípade, že sa pokúša naraz prenášať dáta viacero procesorov je koordinácia medzi nimi zaistená prostredníctvom tzv. zbernicového rozhodcu (bus arbiter), ktorý určuje, ktorý procesor je na rade prenášať. Zbernicové siete sú vhodné pre menšie siete z dôvodu, že pravdepodobnosť súčasných prenosov rastie s počtom procesorov v systéme [23].

Viacúrovňové siete pozostávajú z viacerých úrovní prepínačov, ktoré sú prepojené so susediacimi úrovňami. Cieľom siete je prepojiť vstupy a výstupy tak aby dĺžka trasy medzi týmito bodmi bola čo najkratšia. Tým pádom sa medzi nimi zaistí čo najrýchlejšia komunikácia. Vstupmi takéhoto systému sú pri paralelných systémoch procesory a výstupy sú vzhľadom na organizáciu pamäte buď procesory pri distribuovanej pamäti alebo pamäťové moduly pri zdieľanej pamäti [23].

[1] Úplný binárny strom je špeciálnym prípadom binárneho stromu, ktorý má úplne zaplnené všetky úrovne až na poslednú, ktorá je čo najviac zaplnená zľava [24].

Priečkové siete pozostávajú z n vstupov a m výstupov. Celá sieť potom pozostáva z $n \cdot m$ prepínačov. Na základe organizácie pamäte môžu byť vstupy procesory a výstupy procesory v prípade distribuovanej pamäte alebo pamäťové jednotky pri zdieľanej pamäti. Každý prenos dát medzi vstupmi a výstupmi vyžaduje aby prepínače vytvorili medzi týmito bodmi cestu. Priečkové siete sú vhodné pre menšie siete z dôvodu vysokých hardvérových nákladov [23].

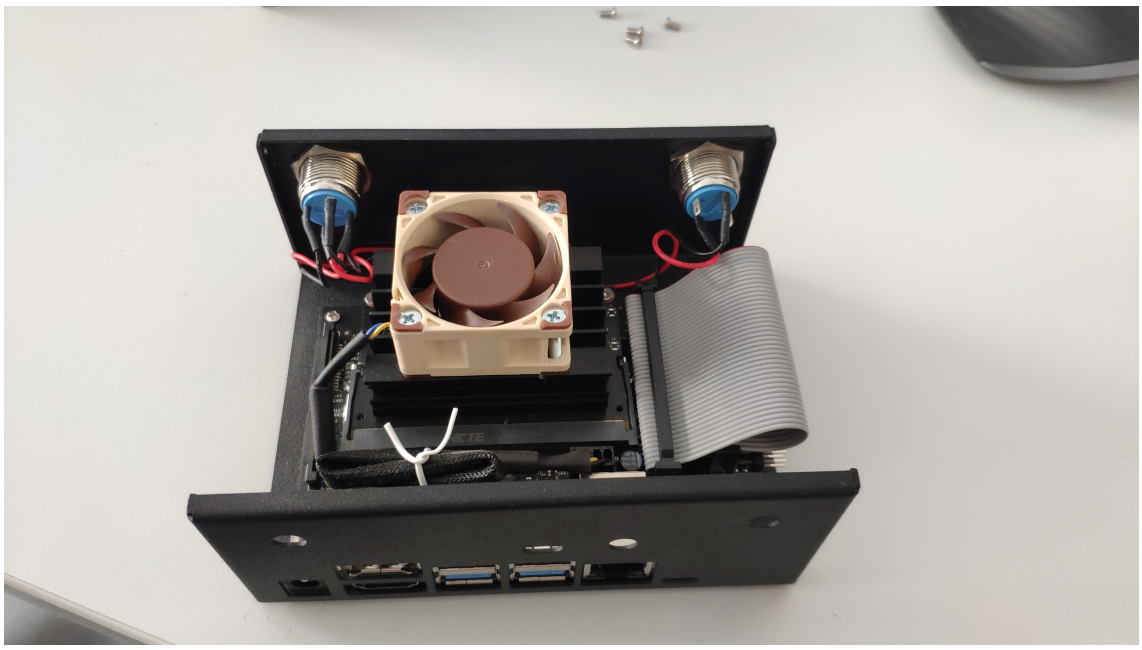
3 Jetson Nano

Jetson Nano (pozri obr. 3.1), je výkonný a kompaktný počítač, ktorý je vhodný pre vstavané aplikácie vďaka svojim veľmi pôsobivým technickým parametrom (pozri tab. 3.1). Keďže sa jedná o počítač, tak je naň možné nainštalovať operačný systém. Výrobca ponúka na svojich stránkach operačný systém, ktorý je založený na linuxovej distribúcii Debian/Ubuntu. Vďaka operačnému systému podporuje platforma Jetson Nano vývoj v rôznych programovacích jazykoch ako napríklad C/C++ alebo Python [25].

Jetson Nano je považovaný za heterogénny systém, pretože je oboma výpočtovými jednotkami, procesorom a grafickým procesorom, využívaná zdieľaná zbernica [26].

Tento počítač zvláda všetky úlohy bežného stolového počítača, avšak primárnym účelom tohto zariadenia je jeho využitie pri IoT riešeniach, čo podporujú aj ponúkané rozhrania. K týmto rozhraniam môžeme pripojiť rôzne externé zariadenia ako senzory, kamery, aktuátory, ktoré umožňujú interakciu s vonkajším svetom [25].

Vďaka svojmu relatívne výkonnému grafickému procesoru je možné na platforme využiť rôzne knižnice pre paralelné spracovanie ako napríklad PyTorch, TensorFlow a podobne. Tento fakt umožňuje platformu využiť aj pre účely umelej inteligencie [25].



Obr. 3.1: Vstavaný mikroprocesor s grafickým procesorom, NVIDIA Jetson Nano.

Jetson Linux je operačný systém, ktorý v sebe zahŕňa Linux Kernel, bootloader UEFI, ovládače od spoločnosti NVIDIA, súborový systém založený na Linuxovej distribúcii Ubuntu a množstvo ďalších vymožeností [27].

Tab. 3.1: Technické parametre systému NVIDIA Jetson Nano [28].

Komponent	Bližšia špecifikácia
CPU	ARM Cortex-A57 1,5 GHz, 4 jadrá
GPU	architektúra NVIDIA Maxwell, 128 CUDA jadier
RAM	4 GB 64-bit LPDDR4, 1600 MHz 25,6 GB/s
Úložisko	16 GB eMMC 5.1

3.1 Výhody mikroprocesorov oproti mikrokontrolérom

Jednou z mnohých výhod mikroprocesorov je fakt, že na nich môže bežať operačný systém, čo im umožňuje vykonávať komplexnejšie operácie ako v prípade mikrokontrolérov. Mikrokontroléry vo väčšine prípadov dokážu vykonávať iba jediný program, ktorý je nahratý do ich programovej pamäte. Mikroporocesory s operačným systémom dokážu vykonať množstvo programov rôzneho zamerania. Väčšina mikroprocesorov poskytuje v základnej konfigurácii viac možností na komunikáciu ako mikrokontroléry, ako príklad je možné uviesť WiFi, Bluetooth a rôzne iné. Výpočtový výkon mikroprocesorov je značne vyšší ako v prípade mikrokontrolérov, pretože mikroprocesory majú väčšinou viac než jedno jadro a môžu poskytovať aj ďalšie výpočtové jednotky ako napríklad grafický procesor [29].

3.2 Možnosti vývoja

Vývoj na platforme Jetson Nano je podporovaný riešením NVIDIA JetPack SDK, ktoré sprístupňuje operačný systém Jetson Linux, rôzne vývojové nástroje, CUDA-X knižnice pre akceleráciu a rôzne iné technologické riešenia spoločnosti NVIDIA. Tento vývojový balíček ďalej zahŕňa aj ukážky, dokumentáciu a taktiež podporuje aj vývojové nástroje pre vývoj na vyššej vrstve abstrakcie ako napríklad DeepStream pre streamovanie videa, Isaac pre vývoj riešení v robotike a Riva [30].

3.2.1 Nástroj jtop

Jtop[1] predstavuje jednoduchý nástroj na monitorovanie a kontrolu zariadenia Jetson Nano a iných zariadení z rodiny Jetson [31]. Poskytuje rôzne informácie o systéme ako sú teplota jednotlivých komponentov, frekvencie jadier procesora, frekvencie pamätí, frekvencie grafického procesora, využitie procesora, grafického procesora a pamätí.

[1]Repozitár nástroja jtop `https://github.com/rbonghi/jetson_stats`

Tento nástroj poskytuje taktiež rôzne informácie o verziách nainštalovaných balíčkov, dá sa tu napríklad zistiť nainštalovaná verzia CUDA, JetPack, verzia operačného systému a mnoho ďalších. Jtop poskytuje funkcionalitu jtop clocks, ktorá umožňuje nastaviť výpočtové jednotky na ich maximálny výkon. Balíček umožňuje ovládať rýchlosť ventilátora určeného pre chladenie systému.

3.2.2 Problémy pri vývoji

Pri vývoji aplikácie na platforme Jetson Nano nastali problémy, ktoré vznikli najmä pri inštalácii balíčkov knižníc potrebných pre vývoj aplikácie. Prvým takýmto problémom bola inštalácia knižnice PyTorch. Tu sa vyskytol problém pri ktorom nainštalovaná verzia PyTorch nepodporovala grafický procesor zariadenia. Pri bližšom skúmaní problému bolo zistené, že je nutné nainštalovať knižnicu PyTorch skompilovanú pre Python 3.6. Všetky verzie tejto knižnice sú dostupné zo stránok výrobcu. Ďalším problémom sa ukázala byť knižnica Numba, ktorá využíva ďalšie závislosti. Problém predstavovala najmä závislosť `llvmlite`, ktorá vyžaduje Python vo verzii aspoň 3.7, avšak Jetson Nano podporuje Python vo verzii 3.6.

4 Analyzátor polarizačných zmien

Projekt FiberGuard analyzuje dáta za účelom zaistenia bezpečnosti optických trás na fyzickej vrstve [32]. Útoky na optické trasy na fyzickej vrstve môžu zahŕňať rušenie, odpočúvanie alebo útoky na infraštruktúru ako takú [33]. Všetky tieto útoky majú jeden spoločný znak a to, že pre ich realizáciu je potrebný fyzický prístup k optickej trase a následná manipulácia so samotným optickým vláknom. Táto manipulácia s optickým vláknom spôsobuje vibrácie, ktoré spôsobujú zmenu v polarizácii. Zmenu polarizácie je možné zachytiť napríklad prostredníctvom analýzy zmien stavu polarizácie, ktorý je využitý aj v projekte FiberGuard [34].

Analyzátor implementovaný v projekte FiberGuard pozostáva z dvoch častí. Prvou časťou je detektor rýchlych polarizačných zmien a druhou je zreťazené spracovanie výstupu detektora. Jednotlivé časti systému sú znázornené na obrázku 4.2. Na základe dát získaných prostredníctvom detektora a následnou analýzou je možné zistiť anomálie spôsobené vibráciami. Tieto anomálie je potom možné vyhodnotiť prostredníctvom modelu umelej inteligencie [32]. Avšak podmienkou pre užitočnosť takejto analýzy je, že analýza a aj následné vyhodnotenie musí prebiehať v reálnom čase. Vzhľadom na vysoké prenosové rýchlosti optických infraštruktúr, kde nie sú nezvyčajné prenosové rýchlosti na úrovni niekoľko desiatok Gbps, je analýza v reálnom čase výpočtovo náročná. Tento fakt plynie z potreby analýzy veľkého množstva dát, kde je pre dosiahnutie primeranej rýchlosti analýzy potrebné použiť výkonné výpočtové zariadenie alebo adaptovať priebeh analýzy k akcelerácii prostredníctvom využitia paralelného spracovania [33].

4.1 Aplikácia FiberGuard

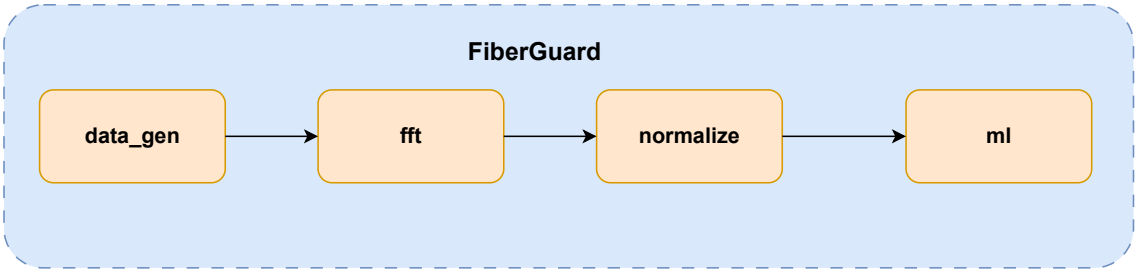
Softvérová časť projektu FiberGuard sa zaoberá analýzou a následným vyhodnotením výstupu detektora rýchlych polarizačných zmien. Analyzované dáta sú spracované využitím zreťazeného spracovania (pozri sekciu 1.3.1), kde prechádzajú jednotlivými modulmi vykonávajúcimi operácie nad týmito dátami. Jedná sa o moduly `fft`, `normalize`, `ml` a načítavané sú prostredníctvom modulu `data_gen`. Zreťazené spracovanie implementované v aplikácii FiberGuard je možné vidieť na obrázku 4.1.

Modul `fft`, ako už názov napovedá zodpovedá za prevedenie signálu z detektora do frekvenčného spektra prostredníctvom Diskrétnej Fourierovej Transformácie (DFT) implementovanej algoritmom Fast Fourier Transform (FFT). Následne je v tomto module vypočítaná absolútna hodnota a aplikuje sa funkcia logaritmovania za účelom získania spektra veľkostí signálu. Výstup modulu `fft` je potom normalizovaný za účelom zníženia nežiadúceho šumu spôsobeného vibráciami optického vlákna [32].

Za normalizáciu zodpovedá modul `normalize` a spočíva v odpočítaní strednej hodnoty od každého vektora. Implementované sú aj iné metódy normalizácie, avšak podľa [32] nedosahujú tak dobrých výsledkov ako spomenuté odčítanie strednej hodnoty.

Ďalším krokom analýzy je vyhodnotenie dát a zhodnotenie, či sa jedná o anomáliu, ktorú je možné považovať za bezpečnostné riziko. Tohto sa dosiahne prostredníctvom modulu `ml`, kde sa vykoná vyhodnotenie prostredníctvom modelu umelej inteligencie [32].

Výsledok analýzy je potom prezentovaný užívateľovi prostredníctvom spektrogramu. Na spektrogame sú jednotlivé anomálie spôsobené vibráciami vzniknutými pri fyzickej manipulácii optickým vláknom viditeľné.



Obr. 4.1: Štruktúra zreťazeného spracovania aplikácie FiberGuard.

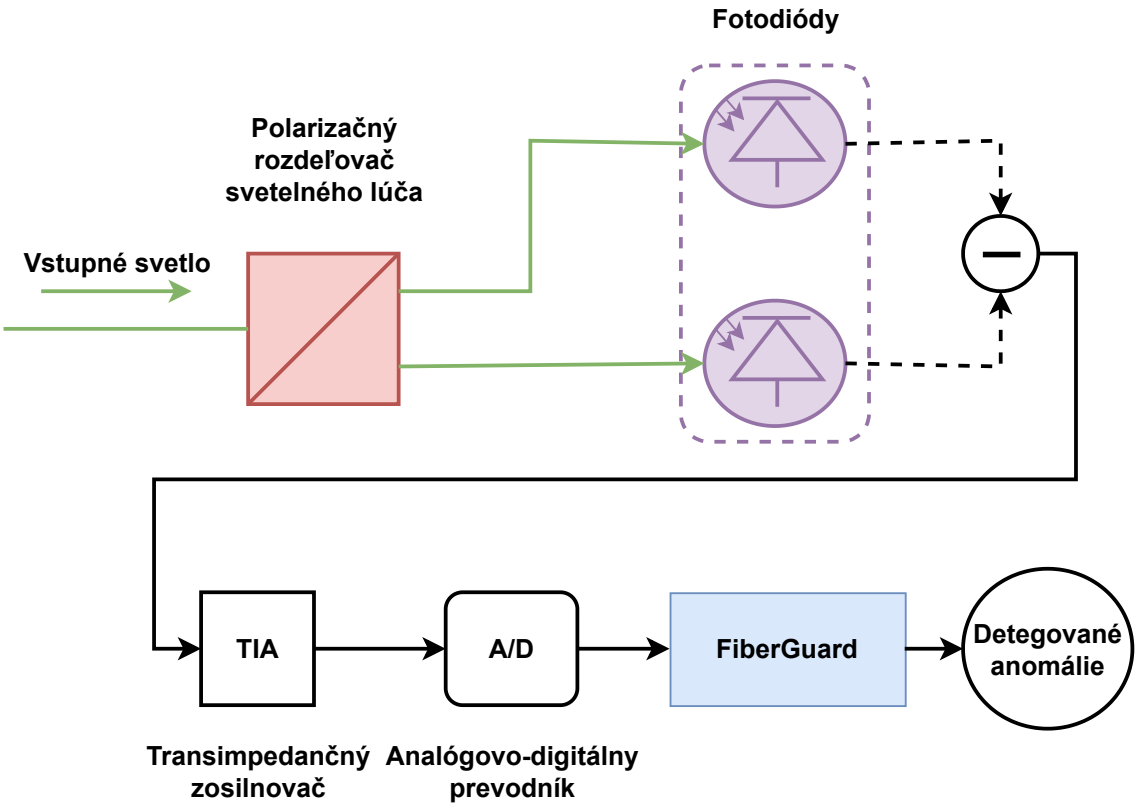
4.2 Detektor rýchlych polarizačných zmien

Senzory založené na optických vláknach využívajú zmien v indexe lomu spôsobených vonkajšími vplyvmi. Táto zmena je definovaná propagačnou konštantou β v dvoch ortogonálnych osiach. Platí, že pre izotropné materiály je index lomu rovnaký v každom smere polarizácie. Čiže, keď je optické vlákno symetrické vo všetkých smeroch, nedochádza v takomto vlákne k polarizácii dvojlomom, $\beta_x = \beta_y$, toto je ideálny prípad. V skutočnom optickom vlákne toto neplatí, index lomu nie je rovnaký vo všetkých smeroch polarizácie, propagačné konštanty jednotlivých osí sú rôzne $\beta_x \neq \beta_y$ [35].

Existujú dva druhy polarizácie dvojlomom: **vnútorná** a **vonkajšia**. **Vnútorná** je spôsobená samotnými vlastnosťami optického vlákna. **Vonkajšia** polarizácia dvojlomom je spôsobená externými vplyvmi na optické vlákno, ktoré môžu byť namáhanie, zmena teploty alebo stláčanie. Polarizácia dvojlomom spôsobuje zmenu polarizácie, ktorú je možné zachytiť [36].

Detektor použitý v projekte je určený na zachytávanie zmien polarizácie. Pozostáva z polarizačného rozdeľovača svetelného lúča, ktorý rozdeľuje svetlo na základe ortogonálnych rovín polarizácie.

Tieto oddelené lúče sú potom prevedené na elektrický signál pomocou dvoch fotodiód. Výstupy jednotlivých fotodiód sú potom od seba odčítané za účelom vytvorenia vyváženého detektora. Výstupný prúd je po odčítaní prevedený na napätie pomocou transimpedančného zosilňovača. Napätie je potom prevedené na digitálny signál prostredníctvom analógovo-digitálneho prevodníka. Tento signál je ďalej analyzovaný v aplikácii [32]. Detektor aj s jeho jednotlivými časťami je možné vidieť na obrázku 4.2.



Obr. 4.2: Znázornenie štruktúry celého systému. Horná časť zodpovedá popisu detektoru rýchlych polarizačných zmien. Časť FiberGuard je softvérová a je znázornená na obrázku 4.1. Vytvorené podľa [32].

5 Akcelerácia analýzy a vizualizácia dát

V tejto kapitole sú popísané výsledky práce. Dôraz je kladený na celý proces implementácie riešenia, ako aj na použité knižnice a frameworky.

5.1 Prostriedky využité k akcelerácii analýzy dát

Pri pôvodnej aplikácii bol použitý pre analýzu dát programovací jazyk Python a jeho mnohé knižnice v kombinácii s platformou Jetson Nano (pozri kapitolu 3) s operačným systémom založenom na distrubúcii Ubuntu vo verzii 18.04. Platforma vyhovuje svojimi technickými parametrami účelu aplikácie.

Aplikácia je zameraná na spracovanie veľkého množstva senzorických dát, čo je výpočtovo náročné, a preto je potrebné tieto výpočty akcelerovať. Platforma Jetson Nano má k dispozícii výkonný grafický procesor, ktorý je na tieto účely vhodný. V kombinácii s programovacím jazykom Python a jeho knižnicou PyTorch s podporou programovacieho modelu CUDA (pozri sekciu 1.3.3), je možné relatívne jednoduchým spôsobom využiť k výpočtom grafický procesor zariadenia. Knižnica PyTorch bola zvolená pre tieto účely na základe porovnania s knižnicou TensorFlow.

5.1.1 Programovací jazyk Python

Pre analýzu dát bol v pôvodnom projekte zvolený programovací jazyk Python. Jedná sa o objektovo orientovaný a interpretovaný programovací jazyk. Tento jazyk využíva dynamické dátové typy, čiže nie je striktne typový ako napríklad jazyk C, v čom spočíva aj jeho užívateľská prívetivosť. Python je prenositeľný, čo znamená, že pracuje na rôznych operačných systémoch ako Windows, rôzne Linuxové distribúcie alebo MacOS, avšak program napísaný v tomto jazyku nemusí byť nutne spustiteľný na všetkých spomínaných operačných systémoch, aj keď je tomu tak vo veľkej väčšine prípadov [37].

Python poskytuje širokú škálu knižníc, ktoré poskytujú spôsoby ako implementovať aplikácie rôznych typov a zameraní. V projekte boli knižnice slúžiace pre získavanie dát zo sieťovej karty, zvukovej karty, knižnice, ktoré podporujú prácu s maticami a tenzormi a knižnica podporujúca multiprocessing.

Knižnica NumPy

Názov NumPy je skratkou anglického slovného spojenia „Numerical Python“ a je knižnicou jazyka Python, ktorá podporuje prácu s vektormi, maticami a tenzormi, ktoré sú implementované ako „ndarray“ – n-rozmerné pole a majú fixnú veľkosť narozdiel od zoznamu implementovaného v jazyku Python.

Knižnica podporuje rôzne matematické operácie nad týmito objektami. Zdrojový kód knižnice je optimalizovaný pre matematické výpočty, čo jej umožňuje vykonávať tieto výpočty veľmi rýchlo, avšak v základnej konfigurácii nepodporuje akcelerovanie výpočtov na grafickom procesore, čo nie je výhodné pri veľkom množstve spracovávaných dát [38, 39].

Knižnica PyTorch

Knižnica PyTorch je knižnica jazyka Python a je alternatívou k už spomínanej knižnici NumPy. Tiež podporuje operácie nad viac rozmernými poľami, ktoré sa v PyTorch terminológii nazývajú „tensor“. Knižnica ďalej implementuje algoritmy strojového a hĺbkového učenia. Oproti knižnici NumPy podporuje PyTorch aj programovací model CUDA, z čoho vyplýva aj podpora akcelerácie výpočtov na grafickom procesore, čo umožňuje výrazne zrýchliť vykonávanie výpočtov oproti knižnici NumPy. Táto podpora je plne zautomatizovaná a programátor sa nemusí zaoberať vytváraním „kernelov“. Čím sa akcelerovanie výpočtov stáva užívateľsky prívetivejšou činnosťou [40]. Knižnica už v základnom nastavení využíva paralelné spracovanie [41]. Táto knižnica bola zvolená pre akceleráciu analýzy nad knižnicou TensorFlow z dôvodu, že je užívateľsky prívetivejšia a ponúka vyšší výkon.

Knižnica TensorFlow

TensorFlow je knižnica využívaná primárne pre účely umelej inteligencie vyvinutá spoločnosťou Google. Zdrojový kód knižnice je napísaný v jazyku C++, avšak nie je obmedzená iba na jazyk C++, je jazykovo nezávislá. Knižnica je ďalšou alternatívou k už spomínaným knižniciam NumPy a PyTorch. Podobne ako PyTorch podporuje aj TensorFlow programovací model CUDA. Zatiaľčo PyTorch ponúka zautomatizovanú podporu CUDA, tak pri TensorFlow tomu tak nie je. K dosiahnutiu podobných výsledkov ako pri PyTorch je potrebné manuálne optimalizovať jednotlivé časti programu [42, 43].

5.2 Akcelerácia analýzy dát

Štruktúru aplikácie tvoria viaceré moduly, ktoré sú zodpovedné za získavanie dát a jednotlivé operácie nad získanými dátami. Tieto moduly sú paralelizované pomocou zreťazeného spracovania (pozri aj sekciu 1.3.1) prostredníctvom modulu `pipeline`. Moduly zodpovedné za jednotlivé operácie nad dátami, ktoré bolo treba akcelerovať predstavujú: `data_gen`, `fft` a `normalize`.

Pôvodným zámerom bolo zachovať pôvodnú štruktúru zreťazeného spracovania, avšak toto nebolo možné z dôvodov popísaných ďalej v texte (pozri sekciu 5.2.2).

Modul `data_gen` bol prispôsobený akcelerácii na grafickom procesore prostredníctvom knižnice PyTorch, takým spôsobom aby sa zjednodušila práca s dátami v ostatných moduloch. Toto zjednodušenie spočíva vo formátovaní dát. Dáta sú po načítaní automaticky formátované do torch tenzorov určitej dĺžky (dĺžku určuje užívateľ pri spustení aplikácie).

Ďalším akcelerovaným modulom je modul `fft`. Akcelerácia tohto modulu bola do značnej miery zjednodušená knižnicou PyTorch, ktorá ponúka funkcie pre Diskrétnu Fourierovu transformáciu vo forme, ktorá je vhodná pre použitie na grafickom procesore.

Akcelerácia modulu `normalize`, bola vykonaná na metóde `mean`. Tu bola vytvorená ekvivalentná metóda k metóde `mean` s názvom `gpu_mean`. Nová metóda sa líši od starej v tom, že sú jednotlivé operácie vykonávané na grafickom procesore.

Porovnanie rýchlostí spracovania prostredníctvom jednotlivých knižníc a výpočtových jednotiek je popísané v sekcii 5.2.3.

5.2.1 Trieda pre generovanie dát zo zvukovej karty

Modul `data_gen` bolo potrebné rozšíriť o možnosť generovania dát zo zvukovej karty zachytených vstupným zariadením. Pre tento účel bola použitá knižnica `Soundcard`, ktorá umožňuje využiť zvukovú kartu zariadenia pre zachytávanie signálu a týmto spôsobom generovať dáta. Vstupné zariadenie nemusí byť nutne mikrofónom, ale v ďalšom texte bude toto zariadenie označené slovom „mikrofón“, pretože knižnica nerozlišuje iné vstupné zariadenie, respektíve každé vstupné zariadenie je objekt triedy `Microphone` [44].

Trieda spočíva v špecifikovaní mikrofónu, prostredníctvom ktorého sa bude zachytávať a následným nahrávaním signálu. Nutnou podmienkou je, že je tento mikrofón pripojený ku zvukovej karte zariadenia, na ktorom program beží. Špecifikovaním tohto mikrofónu sa vytvorí objekt triedy `Microphone`, ktorý má metódu `record()`, ktorá umožňuje zachytiť špecifikovanú dĺžku signálu. Spomenutý spôsob zachytávania je len jednorázový, avšak tento objekt ponúka aj zachytávanie pomocou objektu triedy `Recorder`, ktorý zachytáva signál počas celej svojej existencie. Dĺžku zachyteného signálu je možné definovať atribútom `numframes` a v tomto prípade predstavuje dĺžku okna, ktorú zadáva užívateľ pri spustení programu. Zachytený signál je potom interpretovaný ako matica, ktorej rozmery sú dané nasledovne:

$$počet\ kanálov \times dĺžka\ okna.$$

V prípade tejto implemetácie je ponechaný iba jeden kanál, pretože mikrofón pripojený ku zvukovej karte má vždy iba jeden kanál, alebo má na rôznych kanáloch rôzne výstupy, ktoré nie sú potrebné pre účely analýzy.

Knižnica umožňuje nastaviť aj vzorkovaciu frekvenciu, ktorá je v tomto prípade nastavená na 8 kHz. Vygenerované dáta sú určené pre spracovanie ďalšími modulmi.

Soundcard

Soundcard je knižnica podporujúca prácu so zvukovou kartou prostredníctvom natívnych knižníc operačných systémov pracujúcich so zvukom. Táto knižnica je prenositeľná medzi operačnými systémami. Umožňuje nahrávanie a prehrávanie zvuku. Prostredníctvom tejto knižnice je možné pracovať s jednotlivými kanálmi mikrofónu, kde je možné špecifikovať vzorkovaciu frekvenciu, dĺžku okna pre zachytávanie dát a sprístupňuje mnoho ďalších funkcionalít programátorovi [44].

5.2.2 Migrácia aplikácie do knižnice PyTorch

Pôvodný projekt bol postavený na knižnici NumPy (pozri sekciu 5.1.1), ktorá je výhodná pre prácu s maticami, avšak nemá podporu programovacieho modelu CUDA. Preto bolo potrebné zvoliť vhodnú knižnicu, ktorá podporuje tento programovací model. Pre tento účel bola zvolená knižnica PyTorch (pozri sekciu 5.1.1). Výstupom migrácie je aplikácia, ktorá k svojim výpočtom využíva grafický procesor. Takáto migrácia je výhodná z dôvodu, že urýchľuje spracovanie dát. Táto migrácia sa nezaobišla bez problémov.

Technické ťažkosti pri migrácii aplikácie

Pri migrácii jednotlivých častí aplikácie do knižnice PyTorch nastali problémy, ktoré neboli jednoducho riešiteľné. Najväčším problémom sa ukázala byť nekompatibilita programovacieho modelu CUDA a procesov vytváraných pomocou metódy `fork`. Táto metóda je predvolená v operačných systémoch založených na jadre GNU/Linux. Podľa dokumentácie knižnice je možné pracovať s tenzormi na grafickom procesore prostredníctvom CUDA runtime iba pri procesoch spustených metódou `spawn` alebo `forkserver` [45]. Použitím metódy `spawn` v module `pipeline` však vznikajú nové problémy. Jedným z týchto problémov je komunikácia medzi takto vytvorenými procesmi. Pri tejto metóde dochádza k spusteniu nového Python intepretéra a celý program sa v tomto procese začne vykonávať znova. Tento problém je možné riešiť chránením vstupného bodu aplikácie podmienkou `if __name__ == "__main__"`, ktorá zamedzí vykonávaniu funkcie `main()` vo vytvorených procesoch. Problémom však zostáva inicializácia modulu, kde sa procesy vytvárajú, čo zamedzuje prenosu dát medzi procesmi, pretože sa vo všetkých procesoch znova vytvoria všetky fronty.

Po ďalšom skúmaní možností platformy Jetson Nano bolo zistené, že operačný systém tejto platformy neumožňuje nainštalovať najnovšiu verziu PyTorch. Táto staršia verzia ešte neumožňuje zdieľať CUDA tenzory medzi procesmi. Jediným spôsobom ako dostať dáta z jedného procesu do druhého je kopírovať dáta zo zariadenia (device) na hostiteľa (host). Tieto dáta potom predať ďalšiemu procesu pre ďalšie spracovanie. Toto nie je úplne výhodné, lebo kopírovanie dát zo zariadenia na hostiteľa alebo opačne je náročná operácia.

Tento problém nie je jednoducho riešiteľný, avšak po podrobnejšom skúmaní knižnice bolo zistené, že už v základom nastavení knižnica vykonáva výpočty paralelne [46], čiže v prípade tejto knižnice sa dá zaobísť aj bez zreťazeného spracovania.

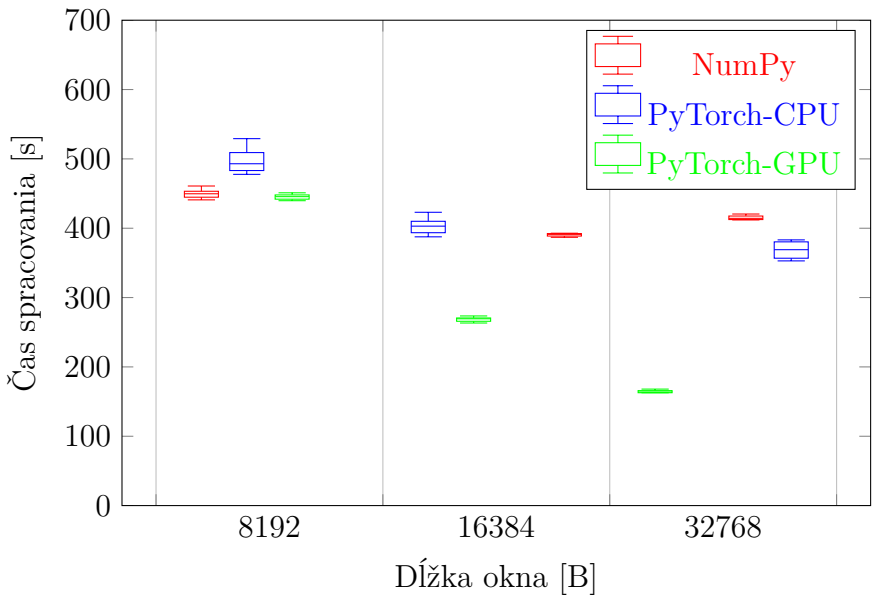
5.2.3 Porovnanie rýchlosti spracovania prostredníctvom knižníc NumPy a PyTorch

Porovnanie časov spracovania bolo simulované na načítaní dát zo súboru prostredníctvom triedy `ReadFile`. Pre spracovanie bol zvolený súbor vhodnej veľkosti, pri ktorej sa prejavili rozdiely medzi časmi potrebnými na spracovanie. Veľkosť súboru predstavovala 7 688 967 312 bajtov (7,68 GB). Súbor bol načítaný a následne spracovaný po oknách o určitej veľkosti. Veľkosti okna predstavovali 8 192, 16 384 a 32 768 bajtov. Veľkosti okien boli zvolené tak, aby dostatočne reflektovali rozdiely medzi rýchlosťami spracovania na jednotlivých výpočtových zariadeniach. Pre každú veľkosť okna bolo vykonaných päť meraní v rámci každej knižnice a výpočtovej jednotky. Jednotlivé časové vstupy tabuľky 5.2.3 predstavujú rozdiel času tesne po skončení výpočtu a času tesne pred započatím výpočtu. Aby sa dosiahlo čo najrýchlejšieho možného spracovania bol na zariadení prostredníctvom nástroja `jtop` povolený režim, pri ktorom bežia všetky výpočtové jednotky a pamäte na maximálny výkon.

Podľa tabuľky 5.2.3 je možné vidieť, že existuje závislosť času spracovania na grafickom procesore a dĺžkou okna. Dĺžka okna je v tomto prípade nepriamo úmerná času spracovania. Toto je spôsobené faktom, že jednotlivé okná sú načítané prostredníctvom hostiteľa a následne kopírované do pamäte zariadenia (pozri tiež sekciu 1.3.3), kde sa vykonávajú operácie nad dátami. Táto operácia je pomerne časovo náročná, a preto je pri menších dĺžkach okna spracovanie na grafickej karte porovnateľne rýchle ako na procesore. Pri väčších veľkostiach okna je pomerne zreteľne viditeľné, že grafický procesor je výhodnejší a prekonáva procesor v časoch spracovania. Tabuľka 5.2.3 tiež ukazuje, že medzi knižnicami NumPy a PyTorch nie je veľký rozdiel pri spracovaní na procesore a je vidieť, že dĺžka okna nijak zásadne neovplyvňuje čas spracovania na procesore. Respektíve, jednotlivé časy sú pri rôznych dĺžkach okna menej odlišné ako v prípade grafického procesora.

Tab. 5.1: Tabuľka zachytáva čas potrebný pre spracovanie pri rôznych knižniciach a výpočtových jednotkách.

Dĺžka okna [B]	**CPU**		**GPU**
	NumPy	**PyTorch**	**PyTorch**
	Čas spracovania [s]		**Čas spracovania [s]**
8 192	460,9	520,9	447,0
	448,6	529,2	439,7
	455,8	477,7	444,1
	440,9	497,3	449,1
	450,8	488,7	451,1
16 384	390,5	387,6	270,7
	404,4	399,5	263,3
	391,8	406,5	268,3
	386,9	413,5	270,5
	392,8	423,0	273,5
32 768	419,7	352,9	162,7
	420,5	383,2	166,8
	415,5	360,6	164,8
	412,0	377,5	168,0
	413,0	418,0	162,8



Obr. 5.1: Graf závislosti rýchlosti spracovania na dĺžke okna.

5.3 Vizualizácia dát

Analyzované dáta je potrebné nejakým spôsobom prezentovať používateľovi. Dáta je možné prezentovať rôznymi spôsobmi, avšak vstupné dáta sú v podstate elektromagnetickým signálom, a preto ich je vhodné prezentovať formou grafu, napríklad pomocou spektrogramu (pozri podkapitolu 5.3.1). Vrámci pôvodného projektu bol spektrogram vykresľovaný tým istým programom, ktorý dáta analyzoval. Toto nebolo výhodné z dôvodu, že pre pozorovanie priebehu bol potrebný prístup k zariadeniu. Priebeh analyzovaného signálu bolo potrebné sledovať vzdialene. Možným riešením tohto problému je vykresľovať dáta externe, mimo Jetson Nano. Napríklad prostredníctvom jednoduchej webovej aplikácie zobrazujúcej prichádzajúce analyzované dáta zo zariadenia.

5.3.1 Spektrogram

Spektrogram sprostredkováva používateľovi zmenu frekvencie signálu v čase, v človeku zrozumiteľnom formáte. Najčastejšie sa jedná o dvoj-dimenzionálny graf, kde horizontálna os reprezentuje frekvenčný rozsah, vertikálna os predstavuje čas a farba predstavuje amplitúdu. Takýto spektrogram sa taktiež nazýva ako „vodopádový“ (waterfall), pretože nové vstupy do spektrogramu sa objavujú na vrchu. Spektrogram môže fungovať na základe rozkúskovania vstupného signálu na okná určitej dĺžky, nad ktorými sa vykonávajú určité operácie ako Diskrétna Fourierova transformácia, ktorá je najčastejšie implementovaná v podobe algoritmu FFT (Fast Fourier Transform). Medzi týmito oknami je potom častokrát prekryv, ktorý slúži na to, aby bolo vykresľovanie prechodov medzi oknami uhladenejšie, čiže aby sa v čo najväčšej miere predišlo skokovým prechodom. Najčastejšie sa volí 25–50% prekryv medzi oknami [47, 48]. Tento spôsob je vhodný pre účely aplikácie, keďže sú vstupné dáta rozkúskované na okná. Príklad spektrogramu je možné vidieť na obrázku 5.3.

5.3.2 Webové rozhranie aplikácie

Pri vytváraní webového rozhrania aplikácie nastávajú dve základné otázky, a to ako preniesť dáta na stranu klienta a ako ich klient vykreslí.

Pre prenos dát je potrebné zvoliť vhodné riešenie, ktoré umožní pozorovanie výstupov aplikácie v reálnom čase. Vhodným riešením tohoto problému je využitie protokolu **WebSocket** (pozri sekciu 5.3.3).

Pre implementáciu rozhrania bol zvolený framework React[1] (pozri sekciu 5.3.2), ktorý je vhodný pre webové aplikácie s dynamickými prvkami. Rozhranie aplikácie pozostáva z dvoch komponentov.

[1]Stránka frameworku dostupná z `https://react.dev/`

Jeden z nich zobrazuje spektrogram a druhý sa používa v prípade, že bolo pri spustení aplikácie zvolené vyhodnocovanie dát modelom umelej inteligencie. Webové rozhranie je možné vidieť na obrázku 5.3, kde spektrogram a vrchný graf predstavujú komponent pre vykresľovanie analyzovaných dát a spodný graf vykresľuje výstup modelu umelej inteligencie.

Komponent vykresľujúci spektrogram využíva knižnicu `react-spectrogram`[2]. Tá umožňuje vykresľovanie vodopádového spektrogramu doplneného o graf znázorňujúci amplitúdu signálu. Horizontálna os spektrogramu predstavuje frekvenciu, vertikálna os predstavuje poradové číslo vzorku.

Avšak knižnica neponúka možnosť priradiť jednotlivým osiam jednotky, čo je problémom nakoľko bez tejto možnosti strácajú jednotlivé grafy na svojej užitočnosti. Možným riešením je získať referenciu na dané elementy, v ktorých sú jednotlivé grafy umiestnené v rámci `HTML` kódu. Tieto referencie je následne možné využiť k priradeniu `CSS` tried, v ktorých sú definované popisy osí a ich umiestnenie v rámci daných elementov.

Komponent vykresľujúci výstup z modelu umelej inteligencie je riešený pomocou grafu z knižnice `ApexCharts`[3]. Výstupom modelu umelej inteligencie je hodnota 1 alebo -1, kde 1 predstavuje anomáliu a -1 predstavuje bežný stav. Vykresľovanie grafu je na základe bodu, ktorého x-ovú súradnicu predstavuje číslo vzorku a y-ovú súradnicu predstavuje výstup z modelu. Tieto body sú následné spojené čiarou. Prijaté hodnoty sa udržiavajú vo fronte s dĺžkou rovnakou ako je počet riadkov spektrogramu, aby bolo možné jednoduchšie priradiť prípadnú anomáliu k danému vzorku.

React

React je JavaScript-ový framework vyvinutý spoločnosťou Facebook, určený pre vývoj a implementáciu interaktívnych užívateľských rozhraní. React je založený na koncepte znovupoužiteľných komponentov. Tieto komponenty predstavujú jednotlivé časti užívateľského rozhrania a je ich možné modulárne skladať. Výhodou tohto frameworku je jednoduché a rýchle vytváranie dynamických webových aplikácií práve vďaka komponentom, ktoré umožňujú rozdeliť komplexné užívateľské rozhranie na jednotlivé menej komplexné časti [49, 50].

[2]Odkaz na repozitár knižnice: `https://github.com/SvajkaJ/react-spectrogram`
[3]Odkaz na stránku knižnice: `https://apexcharts.com/`

Framework React využíva virtuálny DOM (Domain Object Model)[4], čo prispieva k rýchlosti webových aplikácii v porovnaní s inými frameworkami ako napríklad Angular [51].

Dôležitou časťou tohto frameworku sú „States“. States sú objekty, ktoré predstavujú stav daného komponentu. V prípade zmeny stavu komponentu sa tento komponent vykreslí znovu v aktualizovanej podobe [52].

5.3.3 Prenos dát

Pre prenos dát medzi serverom a klientom bol zvolený protokol WebSocket (pozri sekciu 5.3.3). Webový server je implementovaný v jazyku Python za pomoci webového frameworku Flask a spustí sa iba v prípade, že si užívateľ pri spustení aplikácie zvolí, že chce dáta vykresľovať cez web. Užívateľ tohto dosiahne pomocou prepínača `-web`.

Webový server pozostáva z jednej cesty a metódy pre odosielanie dát. Na odosielanie dát bola využitá knižnica SocketIO, ktorá sprístupňuje obojsmernú komunikáciu s nízkou odozvou založenú na udalostiach [53]. V aplikácii beží webový server v oddelenom vlákne, aby sa predišlo blokovaniu analýzy dát. Webový server pri spustení spustí ďalšie vlákno, ktoré má za úlohu odosielať dáta každému pripojenému klientovi. Tomuto vláknu sú predávané analyzované dáta z hlavného vlákna prostredníctvom fronty. Fronta bola zvolená pre tento účel z dôvodu, že je bezpečná z pohľadu zdieľania dát medzi vláknami.

Klient sa pripojí na webový server pomocou prehliadača, kde zadá adresu servera a server mu vráti stránku, ktorá automaticky vytvorí WebSocket-ové spojenie so serverom, cez ktoré sa uskutočňuje prenos dát.

WebSocket

WebSocket je protokol, ktorý poskytuje obojsmernú komunikáciu medzi klientom a serverom. Narozdiel od protokolu HTTP nefunguje na princípe žiadosť–odpoveď ale umožňuje komunikujúcim stranám vysielať kedykoľvek počas nadviazaného spojenia. Protokol WebSocket predstavuje stavový protokol, čo znamená, že je spojenie udržované až kým ho jedna zo zúčastnených strán neukončí. Server dokáže sledovať všetky pripojené stanice a komunikovať s každou z týchto staníc individuálne [54].

Pre nadviazanie spojenia prostredníctvom protokolu WebSocket zobrazeného na obrázku 5.2, je potrebné najprv vytvoriť spojenie protokolu HTTP. Po tom čo je nadviazané HTTP spojenie, prebehne medzi klientom a serverom „WebSocket handshake“. WebSocket handshake pozostáva zo žiadosti klienta a odpovedi servera.

[4]DOM reprezentuje užívateľské rozhranie webovej aplikácie ako stromovú štruktúru, ktorej uzly predstavujú komponenty a ich charakteristiky [51]

Žiadosť klienta pozostáva z bežnej HTTP žiadosti doplnenej o hlavičky identifikujúce povýšenie spojenia na WebSocket spojenie. Server odpovedá špeciálnou odpoveďou, ktorá oznamuje klientovi, že ďalšia komunikácia bude prebiehať prostredníctvom protokolu WebSocket. Po nadviazaní WebSocket spojenia komunikácia už nezodpovedá HTTP protokolu [55, 56].

Tento protokol je vhodný pre webové aplikácie, ktoré zobrazujú dáta v reálnom čase. Príkladom takýchto aplikácií môžu byť aplikácie zobrazujúce grafy v reálnom čase [57].

Obr. 5.2: Nadviazanie spojenia medzi klientom a serverom prostredníctvom protokolu WebSocket [56].

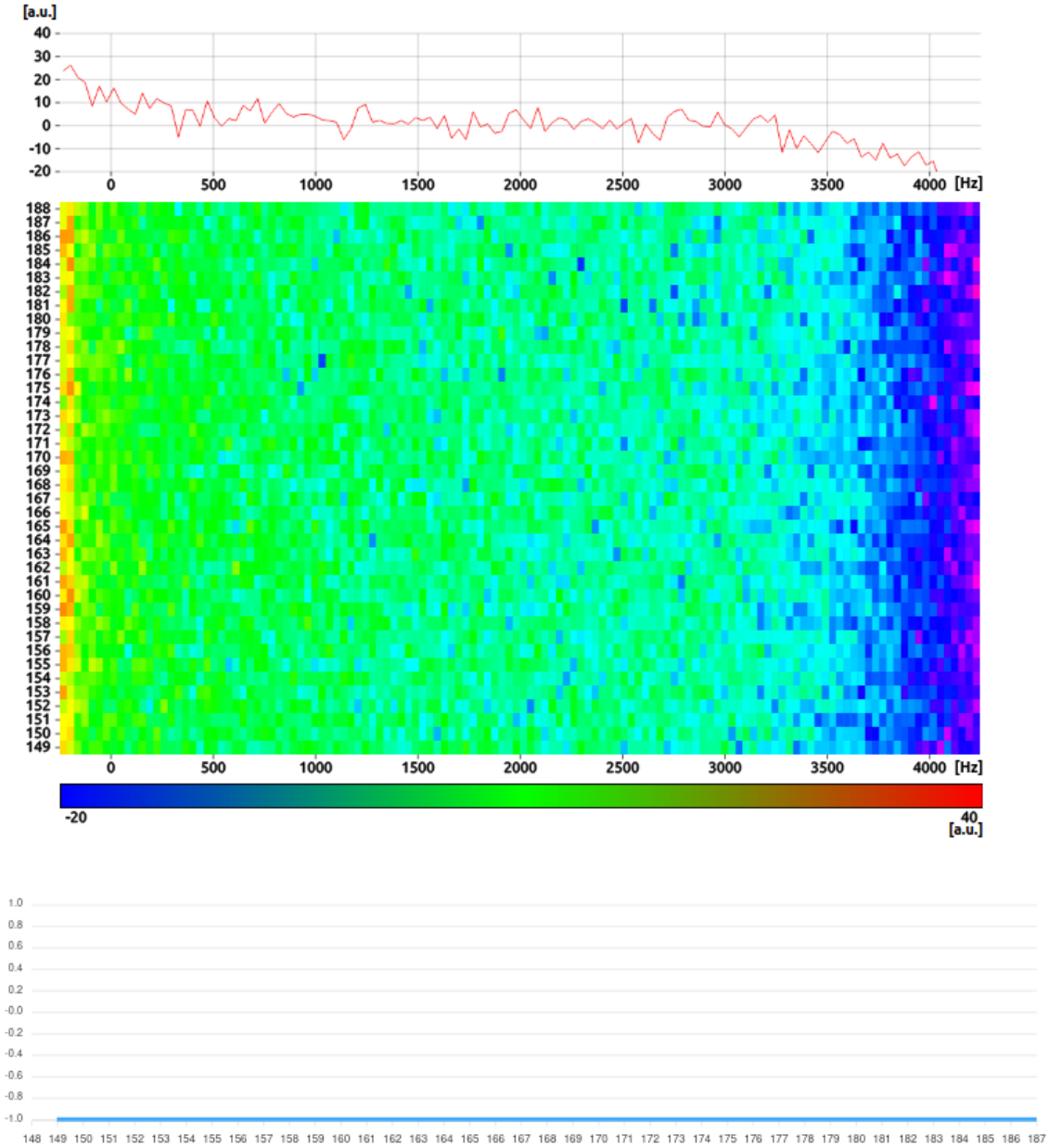
Flask

Flask je webový framework, čo znamená, že poskytuje radu nástrojov a knižníc pre vytváranie webových aplikácií. Konkrétne sa jedná o micro-framework. Micro-framework predstavuje framework, ktorý má malé množstvo závislostí a externých knižníc. Výhoda plynúca z faktu, že má tento framework málo závislostí je, že je odľahčený. Nevýhodou je, že je komplikovanejší z hľadiska užívateľskej prívetivosti, v zmysle implementácie webových serverov.

Flask umožňuje vytvárať cesty (routes) prostredníctvom knižnice Werkzeug. Pomocou tejto knižnice je možné jednoducho definovať jednotlivé cesty za pomoci dekorátorov, kde sa definuje názov cesty a priradí danej webovej aplikácii [58]. Potom cesta definovaná vo frameworku Flask vyzerá ako na výpise 5.1. Tento framework bol vybraný na základe predošlých skúseností.

Výpis 5.1: Príklad definovania cesty vo frameworku Flask.

```
1 @app.route("/")
2 def index():
3     return render_template("index.html")
```



Obr. 5.3: Webové rozhranie aplikácie.

5.4 Testovanie aplikácie na optickej trase so simulovanou prevádzkou

Po úspešnej implementácii jednotlivých častí programu bolo potrebné otestovať funkčnosť aplikácie v testovacom prostredí. Testovacie prostredie predstavovalo učebňu, kde bola testovacia optická trasa s prístupnou časťou. Testovanie aplikácie sa uskutočnilo v troch krokoch.

Prvým krokom bola príprava pracoviska, čo zahŕňalo nasadenie programu na platformu Jetson Nano a pripojenie detektoru rýchlych polarizačných zmien k zvukovej karte platformy.

Ďalším krokom testovania bola simulácia anomálií, ktorá bola vykonaná mechanickou manipuláciou prístupnej časti optickej trasy. Táto manipulácia mala za následok vznik vibrácií na optickom vlákne, ktoré vyvolali zmenu polarizácie.

Nasledoval posledný krok, pri ktorom bolo pozorované grafické užívateľské rozhranie aplikácie za účelom overenia správnosti výstupu. Grafické užívateľské rozhranie aplikácie je možné vidieť na obrázku 5.3. Pri tomto kroku, bolo pozorované, že jednotlivé časti aplikácie fungujú správne a dokážu zachytiť aj veľmi jemné vibrácie v okolí optickej trasy. Veľmi jemnou vibráciou je myslené aj zatvorenie dverí na miestnosti s prístupnou časťou optickej trasy.

Na základe pozorovaných výsledkov je možné skonštatovať, že so systémom je možné pozorovať analýzu polarizačných zmien v reálnom čase. Tým pádom aj pozorovať jednotlivé anomálie.

Záver

Hlavným cieľom tejto bakalárskej práce bolo naštudovanie problematiky, migrácia pôvodnej aplikácie do graficky akcelerovanej knižnice, vytvorenie triedy pre spracovanie dát zo zvukovej karty a návrh a vytvorenie grafického užívateľského rozhrania.

Štúdium problematiky bolo rozdelené do štyroch kapitol. Prvá kapitola sa zameriava na koncept paralelizmu, ktorý je pre akceleráciu kľúčový. V tejto kapitole bolo popisované najmä paralelné programovanie, jeho modely a procesy. Ďalšia kapitola popisuje paralelné architektúry, najmä z hľadiska organizácie pamäte a sietí prepojení. V poradí tretej kapitole sa popisujú špecifikácie platformy Jetson Nano a možnosti vývoja na nej. V štvrtej kapitole boli popísané jednotlivé súčasti dodaného systému.

Praktická časť práce bola popísaná v piatej kapitole. Kapitola popisuje použité knižnice, triedu pre spracovanie dát zo zvukovej karty a porovnanie rýchlostí spracovania medzi pôvodnou aplikáciou a aplikáciou po migrácii do knižnice PyTorch. Táto časť bola sprevádzaná mnohými ťažkosťami, ktoré sa ale napokon podarilo vyriešiť. Problémy predstavovala najmä migrácia modulu, kde bolo využité zreťazené spracovanie (pozri tiež sekciu 5.2.2). Táto kapitola zahŕňa aj popis spôsobu vizualizácie a prenosu analyzovaných dát.

Výstupom práce je aplikácia vhodná pre analýzu zmien stavu polarizácie v reálnom čase a vizualizáciu jej výstupov. Na základe tabuľky 5.2.3 a grafu (obr. 5.1) je možné vidieť, že oproti pôvodnej aplikácii bolo dosiahnuté urýchlenie výpočtov. Je možné skonštatovať, že takáto akcelerácia má skutočne zmysel.

Vizualizácia dát bola dosiahnutá vytvorením webového rozhrania, ktoré je možné vidieť na obrázku 5.3. Po implementácii jednotlivých častí bolo vykonané testovanie na optickej trase so simulovanou prevádzkou, ktoré ukázalo, že systém je vhodný pre monitorovanie optickej trasy v reálnom čase.

Literatúra

[1] MOLDOVAN, Dan. *Parallel Processing from Applications to Systems.* San Mateo, CA: Morgan Kaufmann Publishers, 1993. ISBN 1-55860-254-2.

[2] RAUBER, Thomas a Gudula RÜNGER. *Parallel Programming: for Multicore and Cluster Systems.* 2nd ed. 2013. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37801-0.

[3] *Operating System – Processes*: Process [online]. Madhapur: Tutorialspoint, c2023 [cit. 2023-04-14]. Dostupné z: `https://tinyurl.com/4xjzju6c`

[4] *Fork() in C* [online]. Noida: GeeksforGeeks, 2023 [cit. 2023-04-14]. Dostupné z: `https://www.geeksforgeeks.org/fork-system-call/`

[5] *Copy on Write* [online]. 2020: GeeksforGeeks, Noida [cit. 2023-04-14]. Dostupné z: `https://www.geeksforgeeks.org/copy-on-write/`

[6] *Thread in Operating System* [online]. GeeksforGeeks, 2023 [cit. 2023-04-15]. Dostupné z: `https://www.geeksforgeeks.org/thread-in-operating-system/`

[7] *Difference between User Level thread and Kernel Level thread* [online]. Noida: GeeksforGeeks, 2023 [cit. 2023-04-15]. Dostupné z: `https://tinyurl.com/4pnnpt2c`

[8] *Kernel Threads and User Threads* [online]. IBM, 2023 [cit. 2023-04-15]. Dostupné z: `https://tinyurl.com/52s8afxm`

[9] *Introduction to Deadlock* [online]. Noida: JavaTpoint, c2011–2021 [cit. 2022-12-09]. Dostupné z: `https://tinyurl.com/2p97yzw8`

[10] SUGANDHI, Abhresh. *What is Process Synchronization in Operating System (OS)?* [online]. KnowledgeHut, 2023 [cit. 2023-05-06]. Dostupné z: `https://tinyurl.com/2naw3pdv`

[11] *Mutex lock for Linux Thread Synchronization* [online]. Noida: GeeksForGeeks, 2019 [cit. 2023-05-07]. Dostupné z: `https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/`

[12] STARK, Eugene W. Semaphore primitives and starvation-free mutual exclusion. *Journal of the ACM* [online]. 1982, **29**(4), 1049–1072 [cit. 2023-05-07]. ISSN 0004-5411. Dostupné z: `https://doi.org/10.1145/322344.322352`

[13] HOARE, C. A. R. Monitors. *Communications of the ACM* [online]. 1974, **17**(10), 549–557 [cit. 2023-05-07]. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/355620.361161

[14] MATTSON, Timothy, Beverly SANDERS a Berna MASSINGILL. *Patterns for Parallel Programming.* Westford (Massechusetts): Addison-Wesley, 2005. ISBN 0-321-22811-1.

[15] PACHECO, Peter S. *An introduction to parallel programming.* Burlington, MA: Morgan Kaufmann Publishers, c2011. ISBN 978-0-12-374260-5.

[16] PADUA, David. Pipelining. In: PADUA, David. *Encyclopedia of Parallel Computing.* New York: Springer, 2011, s. 1562–1563. ISBN 978-0-387-09765-7.

[17] HARRIS, Mark. *An Easy Introduction to CUDA C and C++* [online]. NVIDIA Corporation, 2012 [cit. 2022-11-05]. Dostupné z: https://tinyurl.com/bddmtcrh

[18] *Nvidia's CUDA: The End of the CPU?: The CUDA APIs* [online]. New York: Fedy Abi-Chahla, 2008 [cit. 2022-11-05]. Dostupné z: https://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954-6.html

[19] *CUDA Runtime API* [online]. NVIDIA Corporation, October 3, 2022 [cit. 2022-11-05]. Dostupné z: https://tinyurl.com/2ujwn6t4

[20] FLYNN, Michael J. a Kevin W. RUDD. Parallel architectures. *ACM Computing Surveys* [online]. 1996, **28**(1), 67–70 [cit. 2023-05-07]. ISSN 0360-0300. Dostupné z: https://doi.org/10.1145/234313.234345

[21] ALNUWEITI, H.M. Parallel architectures and algorithms for image component labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence* [online]. 1992, **14**(10), 1014–1034 [cit. 2023-05-07]. ISSN 01628828. Dostupné z: https://doi.org/10.1109/34.159904

[22] ROOSTA, Sayed. *Parallel Processing and Parallel Algorithms: Theory and Computation.* New York: Springer, 2000. ISBN 978-1-4612-7048-5.

[23] DUNCAN, R. A survey of parallel computer architectures. *Computer* [online]. 1990, **23**(2), 5–16 [cit. 2023-05-07]. ISSN 0018-9162. Dostupné z: https://doi.org/10.1109/2.44900

[24] *Complete Binary Tree: What is a Complete Binary Tree?* [online]. Noida: GeeksForGeeks, 2022 [cit. 2022-11-25]. Dostupné z: https://www.geeksforgeeks.org/complete-binary-tree/

[25] KURNIAWAN, Agus. In: *IoT Projects with NVIDIA Jetson Nano: AI-Enabled Internet of Things Projects for Beginners*. California: Apress Berkeley, 2021, 1–6. ISBN 978-1-4842-6452-2.

[26] HALAWA, Hassan. NVIDIA jetson platform characterization. In: *European Conference on Parallel Processing*. Springer, Cham, 2017, s. 92–105.

[27] *Jetson Linux* [online]. NVIDIA Corporation, c2022 [cit. 2022-10-27]. Dostupné z: `https://developer.nvidia.com/embedded/jetson-linux-r351`

[28] *Jetson Nano* [online]. NVIDIA Corporation, c2022 [cit. 2022-10-22]. Dostupné z: `https://developer.nvidia.com/embedded/jetson-nano`

[29] THORNTON, Scott. *Microcontrollers vs. Microprocessors: What's the difference?* [online]. Microcontrollertips, 2017 [cit. 2022-12-08]. Dostupné z: `https://tinyurl.com/4vnnz8uy`

[30] *JetPack SDK* [online]. NVIDIA Corporation, c2022 [cit. 2022-10-27]. Dostupné z: `https://developer.nvidia.com/embedded/jetpack`

[31] BONGHI, Raffaello. *Jtop package* [online]. Bonghi, c2020-2021 [cit. 2022-12-03]. Dostupné z: `https://rnext.it/jetson_stats/jtop.html`

[32] TOMASOV, Adrian, Petr DEJDAR, Tomas HORVATH, Petr MUNSTER, Glen A. SANDERS, Robert A. LIEBERMAN a Ingrid U. SCHEEL. Physical fiber security by the state of polarization change detection. In: *Fiber Optic Sensors and Applications XVIII* [online]. SPIE, 2022, 2022-5-27, 10–15 [cit. 2023-05-10]. ISBN 9781510650862. Dostupné z: `https://doi.org/10.1117/12.2618490`

[33] FOK, Mable P., Zhexing WANG, Yanhua DENG a Paul R. PRUCNAL. Optical Layer Security in Fiber-Optic Networks. *IEEE Transactions on Information Forensics and Security* [online]. 2011, **6**(3), 725–736 [cit. 2023-05-10]. ISSN 1556-6013. Dostupné z: `https://doi.org/10.1109/TIFS.2011.2141990`

[34] RUZICKA, Michal, Lukas JABLONCIK, Petr DEJDAR, Adrian TOMASOV, Vladimir SPURNY a Petr MUNSTER. Classification of Events Violating the Safety of Physical Layers in Fiber-Optic Network Infrastructures. *Sensors* [online]. 2022, **22**(23) [cit. 2023-05-10]. ISSN 1424-8220. Dostupné z: `https://doi.org/10.3390/s22239515`

[35] BARCIK, Peter a Petr MUNSTER. Measurement of slow and fast polarization transients on a fiber-optic testbed. *Optics Express* [online]. 2020, **28**(10)

[cit. 2023-05-12]. ISSN 1094-4087. Dostupné z: https://doi.org/10.1364/oe.390649

[36] TOMASOV, Adrian, Petr DEJDAR, Petr MUNSTER, Tomas HORVATH, Peter BARCIK a Francesco DA ROS. *Enhancing fiber security using a simple state of polarization analyzer and machine learning.*

[37] *General Python FAQ: What is Python?* [online]. Python Software Foundation, c2001–2022 [cit. 2022-11-18]. Dostupné z: https://docs.python.org/3/faq/general.html#what-is-python

[38] *What is NumPy?* [online]. NumPy Developers, c2008-2022 [cit. 2022-11-18]. Dostupné z: https://numpy.org/doc/stable/user/whatisnumpy.html

[39] *What is NumPy* [online]. pythontutorial.net, c2022 [cit. 2022-11-18]. Dostupné z: https://www.pythontutorial.net/python-numpy/what-is-numpy/

[40] KETKAR, Nikhil a Jojo MOOLAYIL. Introduction to PyTorch. In: *Deep Learning with Python: Learn Best Practices of Deep Learning Models with PyTorch*. 2nd ed. Apress, 2021, s. 27–93. ISBN 978-1-4842-5364-9. Dostupné z: doi:https://doi.org/10.1007/978-1-4842-5364-9

[41] BOESCH, Gaudenz. *Pytorch vs Tensorflow: A Head-to-Head Comparison* [online]. visio.ai, c2022 [cit. 2022-12-03]. Dostupné z: https://viso.ai/deep-learning/pytorch-vs-tensorflow/

[42] *An Introduction to TensorFlow* [online]. Niklas Lang, 2022 [cit. 2022-11-19]. Dostupné z: https://tinyurl.com/34j2mzey

[43] *Keras vs Tensorflow vs Pytorch: Key Differences Among the Deep Learning Framework* [online]. John Terra, 2022 [cit. 2022-11-19]. Dostupné z: https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article

[44] BECHTOLD, Bastian. *SoundCard* [online]. Bechtold, c2022 [cit. 2022-12-03]. Dostupné z: https://pypi.org/project/SoundCard/

[45] *Multiprocessing best practices: CUDA in multiprocessing* [online]. PyTorch Contributors, c2022 [cit. 2022-11-25]. Dostupné z: https://pytorch.org/docs/stable/notes/multiprocessing.html#cuda-in-multiprocessing

[46] *CPU threading and TorchScript inference: Build options* [online]. PyTorch Contributors, c2023 [cit. 2023-05-13]. Dostupné z: https://tinyurl.com/ybpkbcpt

[47] WOLKE, Alan. *Spectrogram Types: The Many Faces of the Spectrogram* [online]. Beaverton (California): Tektronix, 2022 [cit. 2023-04-12]. Dostupné z: https://tinyurl.com/28b8569v

[48] SMITH, Julius. *Spectrograms* [online]. Stanford University, c2022 [cit. 2023-04-14]. Dostupné z: https://ccrma.stanford.edu/~jos/mdft/Spectrograms.html

[49] HERBERT, David. *What is React.js? (Uses, Examples, & More): What is React.js?* [online]. Cambridge (Massachuttes): HubSpot, 2022 [cit. 2023-04-23]. Dostupné z: https://blog.hubspot.com/website/react-js

[50] *Angular vs. React: Which is a Better Choice for Dynamic Web App Development?* [online]. Dallas: Covetus, 2021 [cit. 2023-04-23]. Dostupné z: https://tinyurl.com/eu4zff2d

[51] KS, Adwaith. *React.js Basics – The DOM, Components, and Declarative Views Explained: Component-Based Approach* [online]. freeCodeCamp, 2022 [cit. 2023-04-23]. Dostupné z: https://tinyurl.com/ym3v272k

[52] SUFIYAN, Taha. *ReactJS State: SetState, Props and State Explained: What Is 'State' in ReactJS?* [online]. San Francisco: Simplilearn, 2023 [cit. 2023-04-23]. Dostupné z: https://tinyurl.com/y6ax5d9k

[53] *Introduction: What Socket.IO is* [online]. Socket.IO, c2023 [cit. 2023-04-22]. Dostupné z: https://socket.io/docs/v4/

[54] ASATI, Arpit. *What is web socket and how it is different from the HTTP?* [online]. Noida: GeeksForGeeks, 2023 [cit. 2023-04-21]. Dostupné z: https://tinyurl.com/ms9sx4sn

[55] *Writing WebSocket servers: The WebSocket handshake* [online]. Mozzila Corporation, c1998–2023 [cit. 2023-04-29]. Dostupné z: https://tinyurl.com/y8v66n6s

[56] *WebSocket protocol* [online]. San Francisco: Wallarm, c2023 [cit. 2023-04-29]. Dostupné z: https://tinyurl.com/bdvpm82y

[57] *WebSocket* [online]. TechTarget, c1999–2023 [cit. 2023-04-21]. Dostupné z: https://www.techtarget.com/whatis/definition/WebSocket

[58] DAS, Kushal. *Introduction to Flask: What is flask?* [online]. Das, c2008-2022 [cit. 2023-04-22]. Dostupné z: https://tinyurl.com/2736fvvp

Zoznam symbolov a skratiek

API	aplikačné programové rozhranie – Application Programming Interface
ARM	Advanced RISC Machine
CPU	procesor – Central Processing Unit
CSS	Cascade Style Sheets
CUDA	Compute Unified Device Architecture
DFT	Diskrétna Fourierova Transformácia
DOM	Domain Object Model
eMMC	embedded MultiMediaCard
FFT	Rýchla Fourierova transformácia – Fast Fourier Transform
GNU	GNU's Not Unix
GPU	grafický procesor – Graphics Processing Unit
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	internet vecí – Internet of Things
LPDDR	Low Power Double Data Rate
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
RAM	pamäť s náhodným prístupom – Random Access Memory
SDK	Software Developement Kit
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
UDP	User Datagram Protocol
UEFI	Unified Extensible Firmware Interface
β	propagačná konštanta

Zoznam príloh

A Obsah elektronickej prílohy

Elektronická príloha obsahuje všetky zdrojové súbory aplikácie. Toto zahŕňa zdrojové súbory pre analýzu dát, zdrojové súbory pre webové rozhranie a webový server.

V zložkách `static/` a `templates/` sú súbory vygenerované buildom React-ového projektu.

Po nainštalovaní všetkých závislostí je aplikáciu možné spustiť napríklad príkazom `python3 fiber_guard.py -mc USB -e torch -n mean -web -l 1024`.

- `/` — koreňový adresár priloženého archívu
  - `fiber_guard/` — zdrojové súbory pre analýzu dát a web server
    - `static/` — adresár so statickými súbormi webového rozhrania
    - `templates/` — adresár HTML súbormi
    - `__init__.py`
    - `__main__.py` — vstupná metóda aplikácie
    - `data_gen.py` — načítanie dát
    - `fft.py` — metódy pre FFT
    - `globals.py` — zdieľané premenné medzi vláknami
    - `chart.py` — vizualizácia dát
    - `ml.py` — načítanie modelu umelej inteligencie
    - `normalize.py` — normalizačné metódy
    - `pipeline.py` — vykonávanie operácií nad dátami
    - `websocket_server.py` — webový server
  - `web_client/` — zdrojové súbory webového rozhrania
  - `fiber_guard.py` — súbor pre spustenie aplikácie
  - `Pipfile` — súbor pre správu balíčkov vo virtuálnom prostredí
  - `setup.py` — nainštalovanie potrebných balíčkov