

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A
INFORMATIKY

VÝUKOVÁ WEBOVÁ APLIKÁCIA NA
PROGRAMOVANIE GPU

Diplomová práca

2017

Bc. Denis Spišák

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A
INFORMATIKY

VÝUKOVÁ WEBOVÁ APLIKÁCIA NA
PROGRAMOVANIE GPU

Diplomová práca

Študijný program: Aplikovaná informatika
Študijný odbor: 2511 aplikovaná informatika
Školiace pracovisko: Katedra aplikovanej informatiky
Školiteľ: Mgr. Pavel Petrovič, PhD.

Bratislava 2017

Bc. Denis Spišák

Čestné vyhlásenie

Čestne prehlasujem, že som predloženú diplomovú prácu vypracoval samostatne s použitím uvedenej literatúry a ďalších informačných zdrojov.

V Bratislave

.....

Pod'akovanie

Moje pod'akovanie patrí školiteľovi Mgr. Pavlovi Petrovičovi, PhD., za jeho pomoc pri vytváraní práce. Rovnako chcem pod'akovať aj svojim rodičom, ktorí mi boli počas štúdia obrovskou oporou.

Abstrakt

Diplomová práca je orientovaná na vývoj webovej aplikácie, zameranej na vyučovanie paralelného programovania. Úvod práce patrí výskumu už existujúcich paralelných jazykov a technológií, no priblíži aj prostredia zamerané na vyučovanie programovania. Výsledky výskumu sú následne aplikované pri návrhu samotnej aplikácie. Dôležitou súčasťou návrhu aplikácie, je návrh jazyka podporujúceho paralelné výpočty. Následne je priblížená realizácia návrhu aplikácie. Súčasťou implementácie návrhu sú aj hotové príklady v navrhnutom jazyku.

Kľúčové slová: webová aplikácia, výuková aplikácia, výpočty na GPU, OpenCL

Abstract

The diploma thesis is oriented on the development of a web application, which is focused to teach parallel programming. The Introduction includes the research of already existing parallel languages and technologies, but also approximate environment focused on teaching programming. The research results are then applied in design of the application. An important part of the application design is design of a language supporting parallel executions. Implementation of the application design is then approximated. Part of the implementation includes examples made in the designed language.

Key words: web application, educational application, GPU executions, OpenCL

Obsah

1	Úvod.....	1
1.1	Cieľ práce.....	2
2	Východiská práce.....	3
2.1	Počítačová architektúra.....	4
2.1.1	Single Instruction Multiple Data.....	4
2.1.2	Multiple Instruction Multiple Data	5
2.2	Modely paralelného programovania	5
2.2.1	OpenMP	6
2.2.2	MPI	7
2.3	Paralelné programovacie jazyky.....	8
2.3.1	C*.....	8
2.3.2	OpenCL	11
2.3.3	Cuda	13
2.3.4	ISPC	14
2.3.5	Porta	15
2.3.6	Connection Machine Lisp	15
2.4	Programovacie jazyky pre výuku programovania	17
2.4.1	Imagine Logo.....	17
2.4.2	Scratch	17
2.5	Ďalšie prostredia.....	18
2.5.1	Node.js	18
3	Návrh.....	20
3.1	Špecifikácia	20
3.1.1	Návrh jazyka	22
3.1.1	Návrh databázových štruktúr.....	27
3.1.2	Návrh prezentačnej vrstvy.....	28

4	Implementácia.....	33
4.1	Jazyk	34
4.2	Úvodná obrazovka.....	35
4.3	Obrazovka Projects	37
4.3.1	Obrazovka My Projects.....	37
4.3.2	Shared projects	42
4.3.3	Obrazovka Results.....	43
4.3	Obrazovka Lectures.....	48
4.3.1	Select Sort	48
4.3.2	Conway's Game of life	50
4.3.3	Najkratšia cesta v grafe.....	51
4.3.4	Hamiltonovská kružnica	52
4.3.5	Ďalšie príklady	54
5	Testovanie aplikácie.....	55
5.1	Priebeh testovania	55
5.2	Výsledky testovania	56
6	Záver	57

1 Úvod

V dnešnom svete moderných technológií, sa čoraz viac využíva možnosť vyučovania elektronickou podobou, pomocou rôznych e-learningových procesov, napríklad v podobe kurzov a podobne. Samozrejme nemusí sa to týkať len vyučovania ekonomických alebo sociálnych vied, ale aj samotnej informatickej oblasti. Takéto e-learningové prostredia zamerané na informatiku, ponúkajú rôzne možnosti, ako sa naučiť pracovať s nejakými technológiami, čo sa týka aj samotného programovania. Existujú prostredia, ktoré vyučujú programovanie pomocou videí ako kurzov, ale je možné nájsť aj také, v ktorých sa očakáva určitá interakcia, takže užívateľ sa sám učí takým spôsobom, že vytvára kód priamo do nejakého systému.

Keďže na školách sa vyučuje v prvom rade sekvenčné programovanie, žiaci si potom ťažšie predstavujú, ako by vyzeral ten istý algoritmus v paralelnej podobe. Existuje viacero foriem vyučovania paralelného programovania, väčšinou sa však pri nich ráta s tým, že študent už má určité programovacie znalosti, a teda nebude mať problém pochopiť daný kód. V mojej diplomovej práci by som chcel uľahčiť vyučovanie takýchto paralelných algoritmov, hlavne pre začiatočníkov.

Postupne sa budem venovať jednotlivým programovacím jazykom, ktoré podporujú paralelné programovanie a na ich základe by som chcel vypracovať návrh jednoduchého jazyka a následne ho implementovať do webovej aplikácie, ktorá bude slúžiť na programovanie paralelných algoritmov.

1.1 Cieľ práce

Cieľom tejto diplomovej práce je vyhľadať a analyzovať informácie, z oblasti paralelného programovania, následne navrhnúť a implementovať viac-užívateľskú webovú aplikáciu, zameranú na vyučovanie takýchto algoritmov. Táto výuková aplikácia pobeží na serveri s výkonnou grafickou kartou. Pôjde o webovú aplikáciu, ktorá sa bude používať na vytváranie programov, napísaných v jednoduchom výukovom programovacom jazyku.

Dôležitou časťou práce je návrh jazyka. V prvej časti návrhu jazyka pôjde o analýzu už existujúcich jazykov, vhodných pre programovanie paralelných algoritmov. Následne, zo získaných informácií odvodím jednoduchý jazyk, zrozumiteľný aj pre začiatočníkov. Jazyk, ktorý navrhmem, bude podporovať efektívne paralelné výpočty, využívajúc výpočtovú silu GPU.

Súčasťou tejto práce je aj navrhnúť prepojenie webovej aplikácie s jazykom programovaným na GPU. Aplikácia s odvodeným jazykom, bude ponúkať už hotové príklady, na ktorých budú znázornené štandardné algoritmy. V prostredí s novým jazykom vytvorím okrem možnosti programovať vlastné úlohy a projekty, aj niekoľko ukážkových príkladov na základných dátových štruktúrach, pre priblíženie paralelného programovania.

V prvej časti mojej práce vyhľadám všetky už existujúce možnosti vývoja aplikácie pre paralelné programovanie. Týka sa to hlavne výberu vhodného programovacieho prostredia. Analyzujem existujúce systémy a programovacie jazyky podporujúce programovanie paralelných algoritmov.

V ďalšej časti sa zameriam na prípravu základného frameworku pre samotnú aplikáciu, s inštaláciou potrebných súčastí. Vyhľadám najvhodnejšie algoritmy pre paralelizáciu. V tomto prípade sa zameriam na algoritmy preberané na predmete Algoritmy a dátové štruktúry. Po prehodnotení a výbere správnych algoritmov, vytvorím ukážkové pseudo kódy, ktoré použijem v tejto diplomovej práci ako východisko pre vytvorenie reálnych paralelných algoritmov. Následne prejdem k samotnej implementácii webovej aplikácie a jej testovaniu. Na záver sa zameriam na výsledky mojej implementácie a priblížim možný vývoj pre budúce verzie.

2 Výhodiská práce

V tejto kapitole budem opisovať systémy, dôležité pre tvorbu tejto webovej aplikácie. Zhrniem jazyky a modely, ktoré sa zameriavajú na paralelné programovanie. Konkrétne pôjde o hotové jazyky alebo podmnožiny iných známych programovacích jazykov, ktoré boli vytvorené pre programovanie paralelných algoritmov alebo programovanie na GPU.

V súčasnosti sa viacero oblastí informatiky zameriava na programovanie pomocou GPU, nakoľko sú výpočty na GPU oveľa rýchlejšie ako na CPU. Týka sa to hlavne strojového učenia, neurónových sietí a podobne. Pre začiatok uvediem rozdiel medzi sekvenčným a paralelným programovaním. Pri riešení nejakého problému, je prirodzené rozdeliť ho na jednotlivé časti, kde každá časť je špecifická nejakým vlastným výpočtom, teda vyrieši sa prvá časť problému a po jej skončení nasleduje ďalší výpočet. Takto sa prechádzajú jednotlivé časti a vyrieši sa celý problém. Tento spôsob riešenia problému sa nazýva *sekvenčný*. Naopak, ak vezmeme, že všetky časti problému sa budú vykonávať súčasne, tak takéto riešenie nazývame *paralelné*. No treba povedať, že paralelný program môže pozostávať aj z niektorých sekvenčných častí.

Výhodou paralelného programu voči sekvenčnému je v prvom rade rýchlosť výpočtu, čo je veľmi výhodné použiť pri väčších alebo komplexnejších programoch.

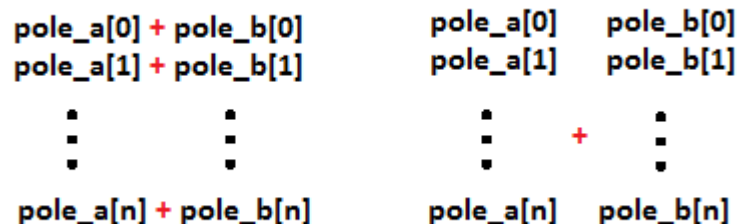
2.1 Počítačová architektúra

Existuje veľa možností ako klasifikovať počítačovú architektúru. V tejto kapitole sa zameriam hlavne na architektúry, podporujúce paralelné výpočty. [1]

2.1.1 Single Instruction Multiple Data

Tento typ počítačovej architektúry je postavený na viacerých jadrách v počítači. Hlavnou črtou tejto architektúry je, že všetky jadrá dokážu vykonávať jednu inštrukciu na rôznych dátach. Výhodou SIMD architektúry je, že umožňuje vykonávať operácie na veľkom množstve dát rýchlejšie ako iné architektúry.

Ak si predstavíme, že máme nejakú sadu výpočtov, napríklad spočítanie prvkov, tak skalárne sa každý jeden výpočet vykonáva osobitne, zatiaľ čo vďaka SIMD môžeme všetky výpočty vykonať naraz (obr.1).



Obr.1: Príklad na rozdiel skalárneho (vľavo) a SIMD (vpravo) výpočtu

2.1.2 Multiple Instruction Multiple Data

Rovnako ako SIMD, je táto architektúra paralelnou architektúrou. Na rozdiel od SIMD umožňuje, pre každý procesor vykonávať rôznu inštrukciu v rôznom čase, niektorá môže napríklad spočítavať, iná odrátavať a ďalšia násobiť. Tieto inštrukcie môžu byť úplne nezávislé.

Príklad pre MIMD možno odvodiť od predošlého príkladu v SIMD, (obr.1) až na to, že tu je možné vykonávať viacero operácií na viacerých prvkoch naraz.

2.2 Modely paralelného programovania

V tejto kapitole špecifikujem základné modely programovania paralelných algoritmov. Tieto modely je možné rozdeliť do troch oblastí, podľa spracovania dát na Shared memory, Message passing a Implicit interaction.

Shared memory, už ako hovorí názov, slúži na vymieňanie dát medzi procesmi pomocou zdieľanej pamäte. Procesy sa tu vykonávajú asynchrónne. S touto formou paralelizmu uvediem v ďalšej časti práce model OpenMP.

Message passing zahŕňa modely, vďaka ktorým si procesy môžu vymieňať dáta medzi sebou pomocou posielania správ. K tejto forme uvediem model MPI.

V *Implicit interaction* má všetko na starosti kompilátor, v tomto prípade nejde o žiadnu výmenu dát medzi procesmi. Tu je možné uviesť ako príklad, programovacie jazyky ako Matlab M-code alebo LabView.

2.2.1 OpenMP

[2] Tento model umožňuje postupnú konverziu sekvenčných programov do paralelných. Všetky detaily, ktoré sú zložitejšie na spracovanie pri implementácii, má na starosti kompilátor. API tejto technológie pozostáva z kompilátora, knižníc a samozrejme premenných. Tento model využíva konštrukciu del'by práce a to tak, že pre každé vlákno stanoví, čo bude robiť. Kompilátor sa vždy pred tým nastaví na začiatok vlákna. Spustenie programu je založené na GSLP štruktúre (Globally Sequential Locally Parallel). V tejto štruktúre ide o to, že existuje nejaký sekvenčný program, ktorý obsahuje nejaké výpočty, ktoré sú časovo najnáročnejšie. Tieto výpočty sa potom vykonávajú plne paralelne. OpenMP je podporou pre C/C++.

Tento model nie je úplne najlepší pre súbežné spustenie procesov, ale predsa len ponúka rýchlejšie spracovanie výsledkov, ak sa zameriame hlavne na cenu vývoja.

Ukážka zdrojového kódu programu OpenMP:

```
#include <iostream>
#include <stdlib.h>
#include <omp.h>
using namespace std;
int main (int argc, char **argv)
{
    int numThr = atoi (argv[1]);
    #pragma omp parallel num_threads (numThr)
    cout << " Hello from thread " << omp_get_thread_num() << endl;
    return 0;
}
```

V tomto príklade je do jednoduchého programu napísaného v jazyku C pridaná knižnica OpenMP s názvom `omp.h`. Následne sa pomocou *`pragma omp parallel`* načíta niekoľko vlákien, v tomto prípade sa počet dostupných vlákien získa pomocou funkcie *`num_threads()`*, ktoré vykonajú nasledujúci príkaz. A funkcia *`omp_get_thread_num()`* vráti ID aktuálneho vlákna.

2.2.2 MPI

[2] Ako som už povedal, tento model je založený na posielaní správ, z čoho vychádza aj samotný názov - The Message-Passing Interface. Považuje sa za akýsi štandard pre distribuovanú pamäť zdieľaného programovania. Model je založený na sade funkcií a keďže používa špecifikáciu, ktorá je jazykovo nezávislá, je možné túto sadu funkcií použiť vo viacerých jazykoch. Prevažne ponúka napojenie pre C/C++ a Fortran. Na pripojenie k jazykom je vytvorená knižnica Boost.MPI. Primárnou a teda základnou úlohou tohto modelu, je rozložiť cieľovú aplikáciu do častí, ktoré môžu byť vykonané ako samostatné procesy.

Ukážka zdrojového kódu MPI programu:

```
#include <mpi.h>
#include <stdlib.h>
int main (int argc, char **argv)
{
    int rank, num, i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &num);
    printf (" Hello from process %i of %i\n", rank, num);
    MPI_Finalize();
    return 0;
}
```

```
}
```

V tomto príklade je MPI inicializované príkazom *MPI_Init()*, kedy sa alokuje potrebné množstvo pamäte. No a *MPI_Finalize()* proces ukončí a uvoľní pamäť. *MPI_COMM_WORLD* vracia informácie o aktuálnom ID procesu.

2.3 Paralelné programovacie jazyky

Najväčšiu skupinu jazykov pre paralelné programovanie tvoria zrejme podmnožiny jazyka C, z ktorého boli vytvorené viaceré jazyky, ktoré v tejto kapitole uvediem.

[2] Existuje viacero vývojových platform pre programovanie na GPU. Uviest' môžem napríklad: Cuda, OpenCL, OpenACC, Thrust, ArrayFire alebo C++ AMP.

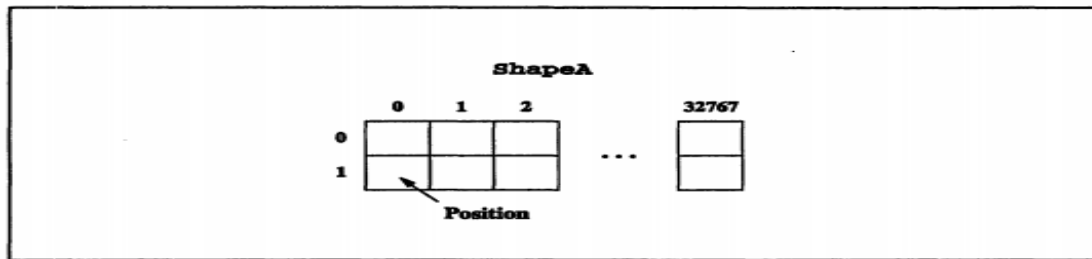
[2] *OpenAAC* je otvorenou špecifikáciou pre aplikácie, ktoré umožňujú používať kompilátory na automatické mapovanie výpočtov na GPU. Jednou z knižníc C++, ktorá zrýchľuje vývoj GPU softvéru využitím sady *container* tried a sady algoritmov pre automatické mapovanie výpočtov na GPU vláknoch je platforma *Thrust*. Spolieha sa na backend Cudy, ale od verzie 1.6 už umožňuje použiť viacero zariadení pre back-end, vrátane CPU. A *ArrayFire* je knižnicou funkcií na GPU. Ostatným platformám sa budem venovať v nasledujúcich podkapitolách.

2.3.1 C*

[3] Prvý jazyk, o ktorom budem hovoriť je C*. Tento jazyk je objektovo orientovaný a je paralelnou nadstavbou ANSI C so synchronnou sémantikou. Táto nadstavba ponúka okrem základov jazyka C aj metódy, na definovanie veľkosti paralelných dát a na vytváranie paralelných premenných. Premenné sa tu definujú tak, že sa najprv určí o akú dátovú štruktúru ide, konkrétne sa využíva výraz *shape*, ktorým sa dátová štruktúra definuje, napríklad:

```
shape [2] [32768] ShapeA;
```


Takýto výraz hovorí, že idem vytvárať dátovú štruktúru s názvom *ShapeA*, ktorá bude dvojrozmerná a teda bude mať 65536 pozícií (obr.2).

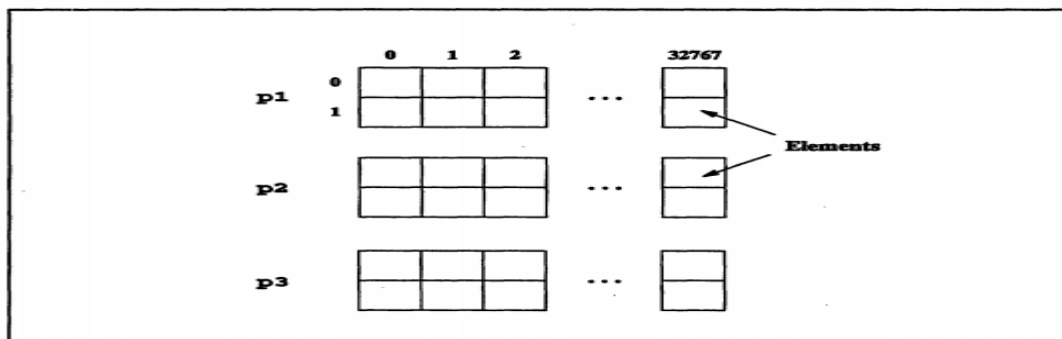


Obr.2: Zobrazenie vytvorenej dátovej štruktúry *ShapeA*

Následne, keď je takto vytvorená dátová štruktúra, môžem vytvoriť paralelnú premennú takejto štruktúry.

```
int:ShapeA p1, p2, p3;
```

V tomto príklade sú definované tri premenné p1, p2, p3, ktoré sú typu *int* a dátovej štruktúry *ShapeA*. To znamená, že každá premenná dedí vlastnosti dátovej štruktúry *ShapeA* (obr.3).



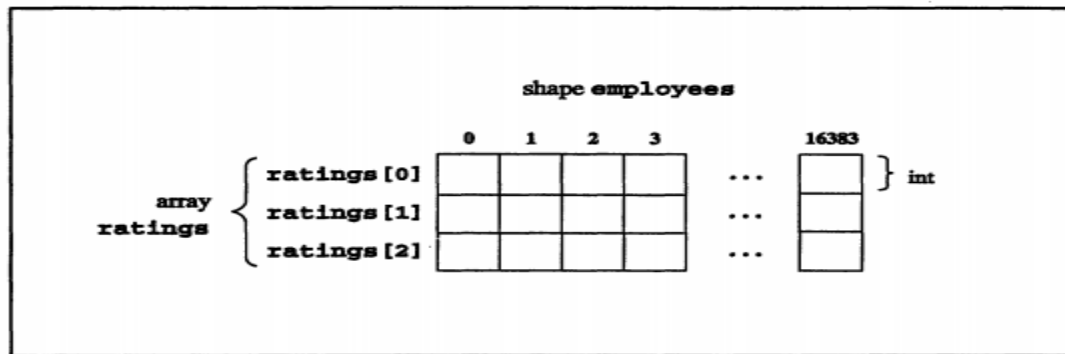
Obr.3: Zobrazenie vytvorených premenných dátovej štruktúry *ShapeA*

Skalárne premenné, definované v tomto jazyku, pozostávajú len z jedného prvku – jedno číslo alebo jeden znak. Paralelná premenná pozostáva z viacerých prvkov. Príklad definovania poľa:

```
shape [16384] employees;
```

```
int:employees ratings[3];
```

Prvý riadok definuje dátovú štruktúru s názvom *employees*, s 16384 prvkami. Druhý riadok hovorí, že budem mať 3 polia, premenné, patriace do tejto štruktúry (obr.4).



Obr.4: Zobrazenie príkladu vytvorenia poľa

Taktiež tento jazyk pridáva nové operátory, výrazy a rôzne metódy, napríklad pre komunikáciu medzi paralelnými premennými. C* pridáva funkciu *current*, ktorá umožňuje špecifikovať, že premenná je aktuálnej dátovej štruktúry, teda *shape*. Komunikácia s premennými C* umožňuje dva spôsoby:

- hodnota niektorého elementu, paralelnej premennej, môže byť priradená inému elementu, aj keď premenná nie je rovnakej dátovej štruktúry. Na to sa použije *parallel left indexing*. Parallel left indexing preskupuje elementy paralelnej premennej na základe hodnôt uložených v elemente indexu aktuálnej dátovej štruktúry.
- druhá možnosť je, že premenné, ktoré patria do rovnakej dátovej štruktúry, medzi sebou komunikujú v pravidelných intervaloch pomocou súradníc. Tento spôsob komunikácie sa nazýva *grid communication* a je rýchlejší ako prvý spomenutý, ktorý sa nazýva *general communication*. V druhej komunikácii sa využíva funkcia *pcoord* v kombinácii s *parallel left indexing*. Funkcia *pcoord* sa používa na vytvorenie paralelnej premennej pre aktuálnu dátovú štruktúru tak, že každý prvok v tejto

premennej je inicializovaný pozdĺž súradníc osi, ktorá je špecifikovaná ako argument vo funkcii `pcoord`.

2.3.2 OpenCL

[4] Open Computing Language. OpenCL je framework, vyvinutý pre paralelné programovanie rôznych procesov. Základom takéhoto programovania je rozdeľovanie úloh medzi procesormi GPU. Konkrétne ide o to, že každá GPU má niekoľko procesorov, ktorým je možné úlohy zadávať jednotlivo, napríklad ak triedim pole, tak každému procesoru viem dovoliť triediť niektorú časť a tým sa získava čas výpočtu. V OpenCL viem dokonca sám zadať koľko procesorov potrebujem, teda ak mi to dovoľí kapacita grafickej karty a zároveň každý procesor má svojich worker-ov, teda pracovníkov, ktorý si potom jednotlivé úlohy medzi sebou rozdeľia. V ďalších častiach práce budem týchto pracovníkov označovať výrazom *worker*.

Rozdeľovanie procesov je možné ako na globálnej pamäti, tak aj na lokálnej v rámci nejakého procesora. Lokálna pamäť je rozdelená na takzvané *memorybanks*, kde každá banka má vymedzené množstvo dát ku ktorým môže pristupovať, takže môže dôjsť ku konfliktu pri spracovaní, keď niektorý procesor zasahuje do dát inej banky, než mu je vyhradené. Výnimka nastáva vtedy, keď je umožnený *broadcast*, v tom prípade môžu všetky vlákna zasahovať do jednej adresy, ale výsledok sa prečíta len raz pre všetky vlákna. Preto je používanie lokálnej pamäti v OpenCL riskantné.

Ďalšou dôležitou funkcionalitou je synchronizačný mechanizmus, ten v OpenCL funguje v podobe *atomics-ov*. Atomics slúžia nato, aby nedochádzalo ku konfliktom medzi jednotlivými procesmi. Takýto konflikt môže nastať v prípade, že niektorú globálnu premennú chcú zmeniť viaceré procesory na rovnakom mieste. Atomics slúžia k tomu, aby nedovolili procesoru meniť premennú v prípade, že ju v rovnakom čase mení iný procesor. Medzi základné funkcie atomics patria: *atomic_inc*, *atomic_dec*, *atomic_xchg*, *atomic_add*, tieto funkcie slúžia, či už na zvyšovanie premennej, znižovanie, sčítavanie, odčítavanie a podobne. V každej verzii OpenCL vznikajú nové spôsoby implementácie atomics.

Tento framework výrazne zvyšuje rýchlosť a odozvu aplikácii. Jeho súčasťou je jazyk, v ktorom je možné písať programy, nazývané *kernely*. Tieto kernely sú vlastne funkcie, ktoré sa vykonávajú na OpenCL zariadeniach. Aplikácia naprogramovaná v tomto frameworku, je založená na samotnom kernely a jeho akomsi riadiacom pozadí, v ktorom sa vykonávajú všetky potrebné základné riadiace príkazy. Ako príklad, možno uviesť niektoré zo základných funkcionalít, ako spracovanie dát na vstupe a ich presun medzi CPU a GPU, respektíve kopírovanie.

Základom takéhoto programu, je získanie dát o zariadení, na ktorom OpenCL beží. Tieto dáta sa jednoducho získajú príkazmi ako: *clContextPlatform* a *cl.GetPlatformIDs*, týmito príkazmi získame základné údaje o zariadení, prípadne aké GPU a CPU zariadenia sú k dispozícii. Následne sa spracujú dáta, z ktorými budeme v danom kernely pracovať. V rámci CPU sú premenné definované ako základné dátové štruktúry. Následne sa tieto CPU štruktúry prekopírujú do GPU, nato v OpenCL slúži nasledovný príkaz:

```
clCreateBuffer (cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret)
```

Táto funkcia dostáva ako parameter *context*, čo je vlastne samotné prostredie, ktoré vznikne pomocou funkcie *clCreateContext*, flags hovoria o tom, či chcem, aby daný buffer na GPU fungoval ako READ_ONLY, WRITE_ONLY alebo READ_WRITE, teda či bude buffer len na čítanie, zapisovanie alebo obe možnosti, nasleduje parameter, ktorý špecifikuje veľkosť *buffra* (size) a samotná CPU premenná, ktorej obsah chcem nakopírovať. Súčasťou funkcie je aj zoznam chýb, ktoré funkcia vráti v prípade, že k nejakej chybe počas výpočtu dôjde, ak chyba nevznikne *errcode_ret* má hodnotu NULL. Ako je vidieť, takýto príkaz by bol pre začiatočníka príliš komplikovaný, v takýchto prípadoch je potrebné minimalizovať nároky na študenta.

Okrem príkazu *clCreateBuffer*, existuje aj príkaz *clEnqueueWriteBuffer*, ktorým sa dáta prepíšu na GPU v tom prípade, že už mám buffer vytvorený a chcem ho prepísať. Takto zadané buffre pre GPU je potom potrebné pre daný kernel poslať ako vstupné parametre, k tomu slúži funkcia *clSetKernelArg*. Po príprave dát sa samotný kernel načíta *clCreateProgramWithSource* a spustí príkazom *clEnqueueNDRangeKernel*. Po vykonaní tohto

príkazu už ostáva iba výsledné dáta premiestniť späť z GPU na CPU, pre jednoduchší prístup k dátam pomocou príkazu *clEnqueueReadBuffer*. Tento proces je možné opakovať s opätovným spustením kernelu, prípade predefinovaním vstupných parametrov pre kernel.

V prípade, že niektorý proces trvá dlhšie a je nevyhnutné počkať kým sa dokončí, každej funkcii viem nastaviť ako parameter aj *event*-udalosť, ako premennú, ktorá zmení svoju hodnotu na *true*, v momente, keď sa daný proces dokončí. Zoznam eventov sa zapisuje do *WaitListu*, do ktorého vie pristupovať každá funkcia. Funkcia, ktorá tak nasleduje bude čakať kým sa daný proces vo waitListe nedokončí a až potom sa vykoná. Zároveň je možné určiť, ktorá funkcia má čakať, na ktorú, takým spôsobom, že si viem vytvoriť viacero waitlistov a každému definujem nejaký event, ktorý potom jednotlivé funkcie spracujú.

OpenCL má ešte viacero ďalších funkcií, v tejto kapitole som vymenoval všetky základné pre tvorbu jednoduchého paralelného programu. Jazyk kernelu je podmnožinou programovacieho jazyka C++, takže obsahuje jeho klasifikáciu tried, výrazov a funkcií. Aktuálne existuje jeho verzia OpenCL 2.2. OpenCL Podporuje grafické karty Nvidia aj AMD, no primárne je určený pre AMD.

2.3.3 Cuda

[5] Compute Unified Device Architecture. Ako ďalší systém možno uviesť práve platformu Cuda. Cuda je platforma pre paralelné výpočty a aplikačné rozhrania. Umožňuje programátorom používať GPU na všeobecné účely. Platforma Cuda je softvérovou vrstvou, ktorá poskytuje priamy prístup k virtuálnej inštrukčnej sade GPU. Príklady niektorých základných príkazov:

```
cudaMalloc( void**devPtr, size_t size );
```

Funkcia *cudaMalloc()*, prideliť veľkosť pamäte na zariadení. *cudaMemcpy()* zase kopíruje dáta z jednej pamäte do druhej, definovanej ako parametre funkcie nasledovne:

```
cudaMemcpy( void * dst, const void * src, size_t count, enum kind);
```

Táto platforma je vytvorená pre grafické karty Nvidia. Uvedená v tejto diplomovej práci je ako ukážka iných platforiem pre účely paralelného programovania. V aplikácii sa ale nepoužije.

2.3.4 ISPC

[7] ISPC možno považovať za prekladač rôznych variantov programovacieho jazyka C. Programátor v tomto modeli napíše program, ktorý vyzerá, že je sekvenčný, aj keď v skutočnosti, vďaka tomuto modelu, je program vykonávaný paralelne.

ISPC prekladá SPMD programovací jazyk založený na jazyku C, pomocou jednotiek CPU a Intel architektúry, čo často spôsobuje 3-násobne alebo aj väčšie zrýchlenie na CPU, bez nejakých obtiažností. Rovnako podporuje aj paralelizácie naprieč viacerými jadrami, aby bolo možné písať programy, ktoré zlepšujú výkon.

Ukážka kódu v ISPC:

```
int main() {  
    unsigned int width = 768, height= 512;  
    float x0 = -2., x1 = 1.;  
    float y0 = -1., y1 = 1.;  
    int maxIterations = 256;  
    int *buf = new int[width*height];  
  
    mandelbrot_ispc(x0, y0, x1, y1, width, height, maxIterations,  
    buf);  
}
```

Základom v tejto ukážke je funkcia *mandelbrot_ispc()*, ktorá má na starosti kompiláciu programu, všetky parametre sú potom pri preklade klasifikované ako *uniform* a buf je potom výstupom, ktorý funkcia po spracovaní vráti ako výsledok.

ISPC možno vytvoriť ako malú sadu rozšírenia jazyka C. Umožňuje využiť výpočtovú silu vektorových jednotiek SIMD.

2.3.5 Porta

[8] Optimálne prenosný jazyk musí mať pre každý program špecifikovanú cieľovú architektúru, tiež musí poskytovať všetky základné vlastnosti tejto architektúry a zakázať funkcie jazyka, ktoré daná architektúra nepodporuje.

Cieľová architektúra môže byť definovaná ako celok alebo môže byť špecifikovaná po jednotlivých úrovniach. Napríklad, ak je viacero súborov so zdrojovým kódom, tak architektúra je špecifikovaná vo vnútri súboru. Ďalšia môže byť špecifikovaná vo vnútri nejakého algoritmu.

Základom návrhu Porta-SIMD bolo ukázať, že prenášateľnosť je dosiahnuteľný cieľ SIMD jazykov. Porta-SIMD je implementovaná ako jazyk C++, vďaka čomu nebola potrebné žiadna nová implementácia syntaxe a sémantiky pre sekvenčné časti SIMD programu. Špeciálne funkcie a výrazy, ktoré boli pridané sú špecifikované cez prefixy *label_* a *direction_*, ktoré slúžia na komunikáciu a hlavne *simd_*, ktorým sú definované premenné ale aj funkcie:

```
simd_float f;  
  
simd_2_bool inside(m);  
  
simd_2_bool rectangle(simd_2_mach *m,int x1,int y1,int x2,int y2);
```

2.3.6 Connection Machine Lisp

[9] Connection Machine Lisp (ďalej len CM Lisp) je určený pre výpočtové aplikácie, ktoré môžu byť použité pre paralelné riešenie algoritmu. CM Lisp vychádza zo štandardu Lisp a pridáva sa len nový dátový typ *xapping* a niektoré ďalšie výrazy pre paralelné programovanie.

Všetok paralelizmus je definovaný okolo dátového typu xapping. Tento dátový typ je spojenie poľa a hashovacej tabuľky. Bližšie je možné povedať, že tento typ tvorí neusporiadaná množina usporiadaných dvojíc. Prvý prvok z dvojice je index a druhý je hodnota. Oba tieto prvky môžu tvoriť aj Lisp objekt. Tento typ nemôže obsahovať dva páry s rovnakými indexami.

Príklad:

{sky->blue apple->red grass->green}

Ďalšou dôležitou vlastnosťou CM Lisp je operátor α , ktorý sa využíva vtedy, ak sa chceme niečo implikovať pre všetky prvky nejakej funkcie f , napríklad:

$$\alpha f: (x_1, x_2, \dots, x_n) \equiv (f: x_1, f: x_2, \dots, f: x_n)$$

konkrétne, pre výpočet druhej odmocniny:

(α sqrt '[1 2 3 4]) -> [1 1.4142135 1.7320608 2]

Ďalším znakom je β , ktorý slúži na zozbieranie paralelných dát do jedného výsledku a tiež na vyjadrenie permutácii. Tento výraz je definovaný s binárnou funkciou a dátovým typom xapping x a výsledok je skombinovaná hodnota prvkov x funkciou f. Napríklad ak máme (β^+ array), dostávame ako výsledok súčet prvkov z x.

2.4 Programovacie jazyky pre výuku programovania

2.4.1 Imagine Logo

Imagine Logo je programovacie prostredie na vytváranie malých aplikácií pomocou korytnačej grafiky. Tento systém je edukačným systémom, pretože je možné v ňom vyučovať základy programovania. Jazyk je vytvorený tak, aby bol ľahko pochopiteľný a je preložený aj do slovenčiny.

Korytnačka je riadená základnými príkazmi ako posun vpred, vzad a otočenie: *fd x*, *rt x*, *lt x*, *bk x*. Korytnačke je tiež možné meniť tvar alebo farbu pomocou príkazu *shape*. Nové premenné sa v tomto jazyku definujú jednoducho, cez *make*“. Nie je potrebné bližšie určovať presný typ premennej podobne ako v JavaScript-e.

Z tohto systému sa rovnako dá vychádzať aj pri vytváraní mojej webovej aplikácie. Niektoré prvky, ktoré Imagine Logo ponúka, môžem použiť pri vytváraní, či už prostredia aplikácie, tak aj samotného jazyka.

[10] V rámci tohto prostredia bol vytvorený doplnok pre vykonávanie výpočtov na GPU, vďaka ktorému je možné v Imagine Logu okrem sekvenčných vykonávať aj paralelné výpočty. Tento doplnok bol vytvorený ako súčasť bakalárskej práce Michala Rakovského.

2.4.2 Scratch

[12] Článok, na ktorý sa v tejto podkapitole odkazujem, hovorí o tom, že viacero základných škôl, alebo aj stredných škôl venuje veľmi málo priestoru rozvíjaniu algoritmickeho myslenia u detí vo vyučovaní informatiky. Existuje ale výukové prostredie, Scratch, ktorého tvorcovia sa zamerali práve na to, aby u detí podporili rozvoj takéhoto myslenia vo vyučovaní, pomocou vytvárania hier a rôznych animácií.

Celé prostredie je postavené na interakcii so žiakom. Scratch pôsobí podobne ako napríklad kresliaci editor, má paletu nástrojov, z ktorých si je možné vybrať a tvoriť tak

príbeh, program. Súčasťou tohto prostredia je aj editor na prácu so zvukom, ktorý žiakov určite zaujme.

Článok, ktorý som spomenul na začiatku, sa zameriaval na výskum, ktorý bol spojený práve s týmto jazykom. Deti si otestovali priebeh celej tvorby programu od návrhu až po implementáciu. Šlo konkrétne o žiakov vo veku 11 rokov. Hlavnou myšlienkou tohto výskumu, bolo deti naučiť ako program pracuje tým, že si najprv nakreslili príbeh, ktorý potom vedeli jednoducho pomocou tohto prostredia vytvoriť.

2.5 Ďalšie prostredia

V tejto podkapitole ešte popíšem niektoré ďalšie prostredia, ktoré sú dôležité pre návrh a vývoj v tejto diplomovej práci.

2.5.1 Node.js

Node.js je open-sourcová platforma JavaScript-u, na vývoj rôznych aplikácií. Node.js má svoje moduly naprogramované v JavaScript-e a aj nové moduly je možné vyvíjať v tomto jazyku. Podporujúce moduly sú potom jednoducho doinštalovateľné príkazom npm v konzole. Samotný node.js ale nie je framework-om JavaScriptu, aj keď tak môže na prvý pohľad pôsobiť. Vďaka tejto platforme sa spoločne s modulom socket.io veľmi dobre komunikuje medzi serverom a klientom. Preto sa táto platforma dá vo webovej aplikácii veľmi dobre použiť ako backend-ová časť aplikácie.

V rámci tejto práce využijem tiež rôzne moduly node.js:

- *fs* – slúži na prácu so súbormi, čítanie zo súborov, respektíve samotné prístupovanie k súborom, či už textovým alebo obrázkovým;
- *express* – je framework-om pre node.js, má viacero dôležitých funkcionalít, možno spomenúť komunikáciu v rámci servera, pre prístup k jednotlivým stránkam;
- *http* – modul http podporuje prácu s rovnomenným protokolom;

- *mkdirp* – slúži na vytváranie a prístupovanie k adresárom;
- *child_process* – tento modul je možné využiť v prípade, že chceme v rámci jedného spusteného procesu spustiť ďalší prípadne viacero ďalších procesov;
- *url* – modul je možné použiť pre lepší prístup k dátam z url;
- *socket.io* – ako som už spomenul, sa hlavne využíva na komunikáciu medzi serverom a klientom;
- *mysql* – vďaka tomuto modulu je možné pracovať s databázou pomocou MySQL;
- *js-sha256* – tento modul som v rámci mojej práce použil na šifrovanie hesiel;
- *vis* – je modul, ktorý slúži na vytváranie a vykresľovanie grafov.

3 Návrh

Kapitola Návrh, obsahuje bližšiu špecifikáciu práce. Zahŕňa popis funkčnosti aplikácie, použité technológie a jej možné využitie.

3.1 Špecifikácia

Práca je zameraná na vytvorenie webovej aplikácie, ktorá bude slúžiť ako edukačná aplikácia, na pomoc pri vyučovaní paralelného programovania na GPU. Táto aplikácia bude použitá prevažne na vyučovanie programovania na stredných školách, no je možné jej využitie aj v nižších ročníkoch vysokých škôl. Aplikácia sa zameria na štandardné dátové štruktúry, ako polia, zoznamy, množiny či grafy, no bude ponúkať aj možnosť výberu obrázka ako vstupnej štruktúry.

Keďže má byť aplikácia viac-užívateľská, bude po otvorení ponúkať možnosť registrácie, resp. prihlásenia, pre študenta alebo učiteľa. Registrácia bude požadovať len základné údaje ako email, meno a heslo. Prihlásený užívateľ bude mať k dispozícii jednoduché menu, ktoré bude pozostávať zo sekcie vlastných projektov, verejných projektov ostatných užívateľov, ukážkových hotových príkladov a sekcie pomocníka, kde budú bližšie popísané základy navrhnutého jazyka a funkčnosti aplikácie.

Zoznamy projektov, ktoré bude užívateľ vidieť na svojom okne po prihlásení budú rozdelené na dve sekcie, zoznam vlastných projektov, ktoré bude mať možnosť editovať, mazať alebo zverejňovať a zoznam verejných projektov, teda projektov, ktoré zverejnili ostatní užívatelia, v rámci tohto zoznamu sa užívateľ dostane ku zdrojovým kódom projektov a tiež si ich bude môcť spustiť. Zároveň bude mať možnosť takto zdieľaný projekt skopírovať do zoznamu svojich projektov, kde si ho už bude môcť sám editovať, bez toho aby menil pôvodný.

Sekcia vlastných projektov bude rozdelená na tri sekcie – podstránky.

Prvá časť bude pozostávať z jedného okna, do ktorého užívateľ, pri vytváraní projektu zadá jeho názov a kratší popis v prípade, že projekt bude chcieť zverejniť, pri opakovanom otvorení tejto časti, bude môcť užívateľ tieto dva základné položky meniť.

Druhá časť bude založená na definovaní dátových štruktúr, respektíve premenných, ktoré bude projekt vyžadovať. Užívateľ si tu bude môcť navoliť aké dátové štruktúry bude daný projekt používať, s tým, že osobitne definuje premenné pre CPU a pre GPU. Táto časť bude pozostávať z jednoduchého editora.

A v tretej časti už pôjde o samotné programovanie. Tu budú pripravené editory pre programovanie projektu, ktoré budú mať možnosť ukladania a zároveň spustenia.

Ďalšia sekcia bude venovaná hotovým ukázkovým príkladom, na ktorých bude možné vidieť, ako taký projekt funguje, táto časť dostane názov *Lectures*. K dispozícii bude aj kód, ktorý priblíži navrhnutý jazyk, v ktorom sa budú tieto paralelné algoritmy programovať. Malo by ísť o príklady, ktoré budú jednoducho prezentovať funkčnosť, napríklad v podobe triediacich algoritmov, algoritmov na grafoch alebo zložitejších príkladov v podobe evolučných algoritmov.

Posledná časť je venovaná pomocníkovi, v ktorom bude definovaná funkcionálna vytvárania projektov, no hlavne tu budú definované základy navrhnutého jazyka.

Aplikácia bude naprogramovaná v JavaScripte s pomocou NodeJS, ktorý bol definovaný v predošlej kapitole, využívajúc viacero knižníc na podporu, či už komunikácie na serveri alebo spracovania údajov. Návrh jazyka, ktorý bude slúžiť na vyučovanie paralelných algoritmov v aplikácii, bude špecifikovaný v nasledujúcej podkapitole.

3.1.1 Návrh jazyka

Jednou z najdôležitejších úloh tejto diplomovej práce je návrh programovacieho jazyka pomocou, ktorého sa budú paralelné projekty programovať na GPU. Jazyk, ktorý navrhнем má byť dostatočne zrozumiteľný a jednoduchý aj pre začiatočníkov. Mal by vyzerat' podobne, ako niektorý z iných jednoducho zadefinovaných jazykov, ako napríklad Python.

Samotná aplikácia bude postavená na platforme node.js s tým, že na prepojenie z GPU sa využije framework OpenCL, o ktorom som už písal v prehľadovej kapitole. Jazyk bude teda vychádzať práve zo spojenia node.js a OpenCL. Toto spojenie ponúka samotný node.js vo svojom module s názvom node-opengl. Tento modul je prekladom OpenCL do JavaScriptu na platforme node.js. Dá sa povedať, že je novšou verziou jazyka WebGL. Keďže tento modul má dosť komplikovanú syntax, ktorá bude pre začiatočníkov dosť ťažko pochopiteľná a nie je dosť vhodná pre účely výukovej aplikácie bude potrebná modifikácia tohto jazyka.

JavaScript sa dá považovať za celkom jednoduchý a zrozumiteľný jazyk, syntax či už pre vytváranie premenných alebo cyklov, nie je nijak komplikovaná. Preto považujem modifikáciu JavaScriptu za ideálne riešenie jazyka, ktorý sa bude v tejto aplikácii využívať, nakoľko v paralelnom programovaní ide o to, aby sa daný proces uskutočnil rýchlejšie, teda ušetril sa čas a pamäť, preklad ďalšieho jazyka do JavaScriptu a OpenCL, by celú aplikáciu iba spomalil.

Modifikácia JavaScriptu bude spočívať v tom, že ju doplním o paralelné príkazy, teda jazyk, ktorý vytvorím bude celý zadefinovaný v module, ktorý sa potom jednoducho importuje do JavaScriptu tak, že sa vždy pri spustení aplikácie načíta. Príkazy zadefinujem v anglickom jazyku, pretože vytvárať príkazy, ktoré sú typické pri programovaní, by mohlo byť v slovenčine zbytočne zmätočné. Jazyk bude obsahovať okrem samotných paralelných OpenCL príkazov, aj ďalšie príkazy pre zefektívnenie priebehu programovania a zjednodušenie práce študenta natoľko, aby daný problém algoritmu pochopil a zároveň sa naučil myslieť paralelne.

Jazyk bude mať základné premenné definované rovnako ako v JavaScripte, čo sa týka komplexnejších dátových štruktúr, tie bude mať modul definované samostatne. Konkrétne pôjde o jednorozmerné a dvojrozmerné polia, množiny, zoznamy alebo grafy. Aby sa žiak nemusel trápiť s tým, ako vytvoriť napríklad maticu, budem potrebovať zdefinovať metódy na prípravu dát. Takýmto spôsobom by mal jazyk zjednodušiť prácu študentovi. Jednotlivé dátové štruktúry, ktoré bude aplikácia podporovať, bude študent definovať rovnako ako v JavaScript-e **var názovPremennej;**, s tým že typ dátovej štruktúry študent definuje pri vytváraní projektu v editore dátových štruktúr a jazyk sa už na pozadí postará o to, aby premennú prekonvertoval do danej štruktúry.

Jazyk pre časť, ktorá pobeží na GPU, je v node-opengl naprogramovaný v OpenCL. V tejto časti budem vychádzať zo základov tohto jazyka. Aj tento jazyk má základné cykly podobné tým v JavaScripte. Premenné je už ale potrebné zdefinovať s typom, teda ak ide o integer zadá sa `int` a podobne. Kernel tiež potrebuje získať informácie o tom, na akom procesore má výpočet prebiehať, prípadne koľko takých procesorov má k dispozícii. Ak niektorý procesor má čakať na druhého OpenCL používa aj príkaz *barrier*. Keďže samotný modul node-opengl využíva pri čítaní kernelov ich preklad, nebude problém v mojom module jednotlivé príkazy, ktoré OpenCL používa modifikovať, zjednodušiť pre pochopenie a pri preklade sa preložia súčasne s celým programom. V tomto prípade nedôjde k žiadnemu spomaleniu programu, keďže sa kód z openC aj tak prekladá.

Rozdeľovanie úloh na GPU záleží, ako som už povedal, na počte procesorov. Procesor na GPU tvorí WorkGroup, teda skupinu pracovníkov. Túto skupinu pracovníkov tvoria jednotlivé vlákna procesora - workers.

V prehľadovej kapitole som uviedol niekoľko základných príkazov pre prácu s OpenCL, tieto príkazy sú takmer rovnaké aj v module node-opengl s minimálnymi rozdielmi. V týchto príkladoch bolo jasne vidieť, že zápis funkcií by mohol, nie len začiatočníkom, narobiť problémy, preto vyviním nasledovné funkcie, ktoré by mali zápis a pochopenie študentovi dostatočne uľahčiť. Okrem toho, že funkcie vyviním, musím jazyk pripraviť aj do takej podoby, že študent bude písať iba tie príkazy, ktoré sú nevyhnutné pre funkčnosť programu a zároveň budú príkazy dostatočne vyjadrovať paralelizmus.

Definovanie dátových štruktúr:

randomFill / zeroFill – tieto dve metódy budú slúžiť na vytváranie jednorozmerných polí, ako hovoria samostatné názvy, tak funkcie pole vytvoria a následne ho naplnia dátami, buď náhodnými – randomFill, alebo bude celé pole pozostávať z núl – zeroFill. Ako parameter tieto funkcie dostanú dĺžku poľa.

randomBinFill – funkcia, ktorá naplní jednorozmerné pole binárnymi hodnotami, čiže náhodne rozhodí v poli hodnoty 1 a 0, rovnako je parametrom funkcie veľkosť poľa.

createRandomMatrix / createZeroMatrix – táto dvojica funkcií bude fungovať takmer rovnako ako randomFill a zeroFill, až na to, že pôjde o vytváranie dvojrozmerných polí, respektíve matíc. S takouto dátovou štruktúrou sa potom pracuje ako s maticou. Parametre funkcií tvoria rozmery, teda počet stĺpcov a riadkov.

randomGraph / zeroGraph – keďže aplikácia ponúka aj programovanie grafových úloh, pridám aj funkcie na vytváranie grafov. Tieto funkcie budú grafy vytvárať nasledovným spôsobom, zeroGraph, vytvorí graf s vrcholmi ale bez hrán, jej jediným parametrom je počet vrcholov; randomGraph zasa vytvára náhodný graf spolu s hranami, ako parametre táto funkcia potrebuje počet vrcholov a počet hrán, tie sa potom prevedú do maticového systému ako dvojrozmerné pole.

Riadiace procesy:

readCore – táto funkcia, bude mať na starosti načítanie kódu, ktorý pobeží na GPU, z programu, ktorý sa v OpenCL nazýva kernel. Metóda načíta do svojej pamäte tento kód pre GPU a následne sa na pozadí kód preloží do úplnej podoby OpenCL kernelu. Algoritmus, ktorý bude bežať na GPU, upravím pre študenta tak, aby bol jednoduchší na pochopenie, keďže kernel v OpenCL obsahuje viacero znakov, ktoré môžu byť pre študenta, ktorý neprogramoval v niektorom C jazyku, dosť nepochopiteľné. Metódu readCore študent sám nemusí volať, pretože sa sama inicializuje pri spustení programu, nakoľko sa vždy vykonáva to isté a študent tým nepotrebuje byť zaťažovaný.

buildProgram – táto funkcia sa vykoná až potom, čo prebehne predošlá funkcia readCore. Po preložení a načítaní kódu v readCore, funkcia pripraví program, ktorý pobeží na GPU, pre build a zároveň v ňom skontroluje syntaktické chyby pred spustením programu.

createBuffer – keďže jednou zo základných úloh samotného programu je príprava dát pre GPU, aj tento akt potrebuje svoju metódu. V tejto funkcii ide o prekopírovanie dát z CPU na GPU, takže úlohou tejto funkcie je vytvorenie buffrov pre GPU, ktorých obsah je zložený z obsahu CPU parametrov. Užívateľ ju ale opäť nezadáva o to sa stará nasledujúca funkcia.

copy – ak študent zavolá túto metódu, tak na pozadí sa vykoná predošlá funkcia createBuffer, iba v takom prípade, že buffer pre GPU ešte nebol vytvorený. Študent pošle vo volaní funkcie parameter, ktorým je CPU štruktúra, ktorá sa má prekopírovať na GPU. Následne sa vykoná funkcia createBuffer, ktorá mu pošle späť vytvorený buffer, ak už nebol vytvorený predtým. To sa môže stať v prípade, že chcem daný program spustiť viackrát a hodnotu v buffri mu budem chcieť iba modifikovať. V prípade, že buffer už existuje tak funkcia copy iba vloží/skopíruje nový obsah do buffra z CPU premennej.

read – opakom funkcie copy je práve táto metóda. Jej úlohou je výsledné dáta, ktoré sa spracujú na GPU, prečítať do CPU premennej naspäť, aby s ňou mohol potom študent ďalej pracovať na CPU, ak ju chce vypísať a podobne. Táto funkcia sa volá až po skončení behu programu na GPU. Ako parametrom sú potom GPU buffer, z ktorého sa bude čítať a CPU premenná, do ktorej sa má výsledok zapísať.

parameter – keďže funkcia, ktorá pobeží na GPU potrebuje vstupné parametre, konkrétne ide o GPU buffre, s ktorými potom pracuje, je potrebné ich pre funkciu na GPU inicializovať. K tomu sa použije funkcia parameter. Parametrami tejto metódy sú potom číslo, teda poradie v akom sa majú do funkcie core zapísať, čísluje sa od 0. Druhým parametrom tejto metódy je samotný buffer – premenná GPU.

run – touto funkciou užívateľ spustí celý program. Samotné paralelné programovanie pozostáva v rozdeľovaní úloh medzi procesormi. V tomto prípade treba definovať, koľko procesorov potrebujem k danému programu. Ako parametre tak funkcia dostane počet procesorov a počet jednotlivých worker-ov v rámci procesora, ktorí si prácu medzi sebou rozdelia na GPU.

Príkazy pre programovanie na GPU:

getWorkerNumber – príkaz vráti id worker-a patriaceho aktuálnemu procesoru.

getWorkersCount – vráti počet všetkých worker-ov v aktuálnom procesore.

barrier – bude slúžiť na zastavenie procesorov, v prípade, že niektorý z ostatných ešte nedobehol a je potrebné, aby sa všetky procesory počkali.

Spracovanie výsledkov na CPU:

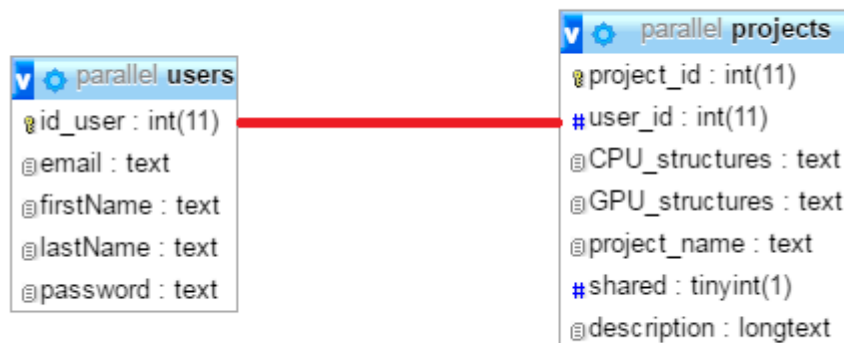
printResult – funkcia vypíše výsledok spracovania programu, po získaní dát z funkcie read. Parametrom bude premenná z CPU.

draw – funkcia vykreslí výsledok programu do predpripraveného Canvasu. Kresliť bude v podobe polí, jednorozmerných alebo dvojrozmerných a grafov. Typ vykreslenia študent zadá ako druhý parameter v podobe: 1D, 2D alebo matrix a graph. Prvým parametrom bude premenná, CPU, ktorú bude chcieť študent vykresliť. V rámci vykresľovania 2D poľa respektíve matice, funkcia draw akceptuje ešte parametre ako počet riadkov a stĺpcov. Súčasťou grafových úloh môže byť aj vykresľovanie ciest medzi vrcholmi.

Modul sa sám automaticky vždy pred spustením programu načíta, čiže nebude potrebné, aby študent vpisoval príkaz na prístup k modulu. Modul dostane názov *parallel*, takže jediné, čo bude musieť študent robiť, aby sa dostal k jeho metódam, je pred každú metódu napísať príkaz **parallel.nazovMetody**. Je to rovnaký spôsob, ako funguje pre všetky ostatné moduly JavaScript-u.

3.1.1 Návrh databázových štruktúr

Databáza tejto aplikácie pobeží na MySQL. NodeJS s MySQL spolupracuje jednoducho vďaka modulu *mysql*, ktorý sa na server doinštaluje. Pre potreby tejto aplikácie budem mať dve tabuľky, jednu na správu užívateľov a druhú pre projekty.



Obr.5: Návrh databázovej štruktúry

Tabuľka **users** uchováva pre každého užívateľa jeho osobné id, email, meno a priezvisko ako firstName a lastName spolu s heslom. Všetky tieto údaje užívateľ zadáva už pri prvotnej registrácii.

Druhá tabuľka **projects** bude uchovávať pre každý projekt jeho id ako project_id, zároveň si tabuľka bude uchovávať aj id užívateľa, ktorý projekt vytvoril a práve týmto id užívateľa budú tieto dve tabuľky prepojené. Tabuľka tiež uchováva štruktúry, ktoré si autor projektu vyberá v editore štruktúr pri vytváraní projektu. Súčasťou sú aj položky project_name a description. Tabuľka bude uchovávať informáciu o tom, či je projekt možné zdieľať s ostatnými užívateľmi ako shared.

3.1.2 Návrh prezentačnej vrstvy

Projects

V tejto podkapitole priblížim návrh už spomínanej sekcie pre vytváranie projektov. Ako už bolo povedané prvá časť bude založená na vytvorení názvu projektu, jednoduchým oknom, do ktorého bude môcť autor vpísať názov s tým, že ho bude môcť kedykoľvek zmeniť spolu s popisom projektu.

Zaujímavejšia je ale nasledujúca časť, v ktorej si autor zdefínuje dátové štruktúry, ktoré bude vo svojom projekte využívať. Toto okno bude založené na jednoduchom editore rozdelenom na CPU a GPU časť, do ktorých si bude autor doťahovať, jednotlivé štruktúry (Obr.5).



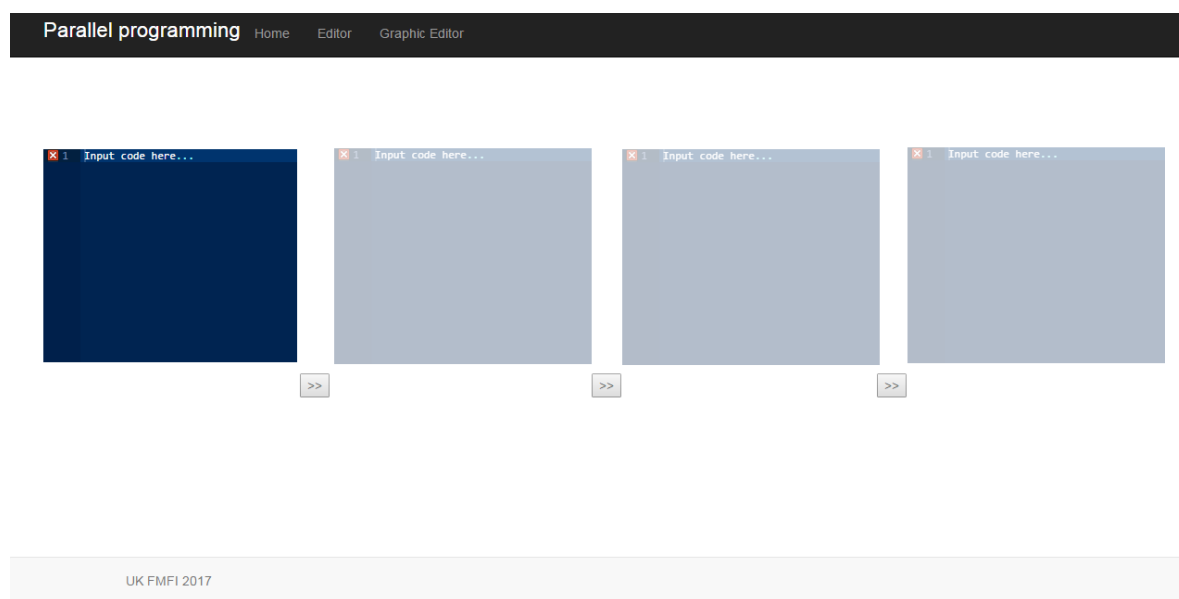
Obr.6: Návrh štruktúry formulára

V pravej časti okna bude zásobník štruktúr, ktorý bude postavený na systéme *drag & drop*, takže užívateľ bude môcť tieto dátové štruktúry doťahovať do boxov pre CPU a GPU panel, ako je možné vidieť na predošlom obrázku. Výber jednotlivých štruktúr má neobmedzené množstvo, to znamená, že sa po odobratí vytvorí nová kópia štruktúry. Zároveň, box so štruktúrou, ktorý už je vybraný, či už pre CPU alebo GPU panel, neostáva statický ale je možné ho naďalej medzi jednotlivými boxami presúvať, respektíve ho vrátiť do pôvodného

zásobníka. Po vybratí štruktúry bude môcť užívateľ štruktúre už ako premennej definovať meno. Ako je možné vidieť na predošlom obrázku, užívateľ môže zo zásobníka vybrať aj premennú typu obrázok, po vybratí takejto premennej sa užívateľovi zobrazí možnosť uploadovať obrázok a zároveň ho aj pomenovať. Takto vybraný obrázok sa následne uloží do adresára projektu na serveri. Spracovanie obrázka podlieha konverzii do .b64, v takejto podobe potom ostáva uložený na serveri a pred jeho zobrazením sa prekonvertuje späť.

Takto zadefinované štruktúry a premenné sa následne prenesú do ďalšej časti, ktorá už bude zahŕňať samotné programovanie. Všetky z týchto troch sekcií budú mať vlastnú navigáciu. Ukladanie jednotlivých nastavení projektu bude prebiehať vždy po zatvorení danej sekcie.

Ďalšia časť patrí samotnému programovaniu, v nej bude mať žiak k dispozícii editor, do ktorého bude vpisovať svoj kód v modifikovanom jazyku. Editor bude ponúkať highlighting a návrhy na úpravu chýb. Samotné programovanie bude rozdelené do štyroch fáz, v nich bude užívateľ programovať štyri základné funkcie. Každá funkcia bude mať svoj vlastný editor. Na podobnom princípe funguje napríklad Imagine Logo, kde má užívateľ možnosť pristupovať k zoznamu všetkých funkcií a po otvorení niektorej funkcie zo zoznamu, sa mu zobrazí kód a ten môže upravovať, prípadne dopĺňať nové funkcie.



Obr.7: Návrh programovacích okien

Užívateľ teda bude programovať 4 základné funkcie, ktorých editory budú vždy vopred nachystané. Tieto štyri funkcie budú ponúkať nasledujúce možnosti:

funkcia start – prvá základná funkcia, bude obsahovať kód, ktorý pobeží na CPU, pretože pôjde o základné načítanie dát pre CPU a ich komunikáciu s GPU. Táto funkcia bude slúžiť ako základný riadiaci operátor celého programu. V rámci funkcie pôjde o načítanie funkcie core a prípravu parametrov pre túto funkciu, následne sa program odtiaľto spustí a tiež sa v nej bude získavať výsledok.

funkciu core - bude ju tvoriť kód, ktorý pobeží na GPU, v OpenCL je táto funkcia nazývaná kernel. Ide o funkciu, v ktorej bude zapísaný algoritmus, ktorý už pobeží na GPU.

funkcia data_preparation - táto funkcia slúži na prípravu samotných údajov, takže vytvorím funkciu s rovnomenným názvom. V tejto funkcii bude dochádzať k definovaniu vstupných údajov pre CPU, pripraveniu ich buffrov a následným prekopírovaním dát z premenných z CPU do premenných GPU, a to ešte pred spustením kernelu, v našom prípade funkcie core.

funkcia take_data - posledná zo základných funkcií, bude opak funkcie data_preparation, pretože pôjde o funkciu, ktorá skopíruje údaje z premenných GPU do premenných CPU po skončení funkcie core, teda kernelu. V rámci tejto funkcie sa najprv zdefinujú premenné pre GPU a následne dôjde k prekopírovaniu dát na CPU.

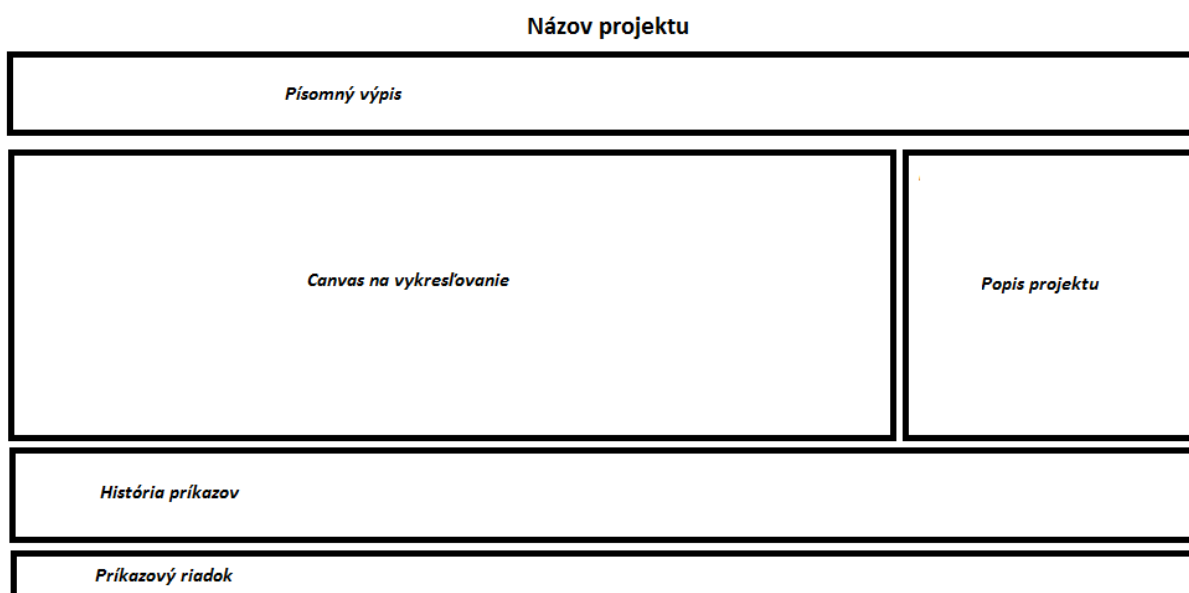
Zároveň bude k dispozícii ešte jedno okno s editorom, v prípade, žeby si chcel užívateľ doprogramovať nejaké vlastné pomocné funkcie.

Ukladanie týchto zdrojových súborov bude zaobstarané tlačidlom *Save*. Všetky projekty budú mať na serveri svoj vlastný adresár, do ktorého sa budú tieto súbory ukladať spolu s prípadnými obrázkami. Spustenie projektu prebehne stlačením tlačidla *Run*, ktorým sa otvorí v prehliadači nové okno s výsledkovou obrazovkou.

Results

Obrazovka Results bude slúžiť na vyhodnotenie výsledkov programu, bude zobrazovať názov a popis projektu. Základom bude príkazový riadok, do ktorého bude môcť užívateľ zadávať príkazy, ktoré bude mať jeho program spracovať. Na to budú v jazyku vytvorené príkazy ako printResult alebo draw. Po zadaní takýchto príkazov sa výsledok zobrazí na obrazovke v dvoch podobách, ako písomný výpis, čiže obrazovka bude mať box s výpisovými hodnotami a druhou možnosťou je vykreslenie výsledku, to bude umožnené pomocou HTML Canvas-u. Takýto Canvas bude vedieť kresliť grafy a zobrazovať polia v podobe políčok s hodnotami.

Obrazovka bude zobrazovať aj históriu posledných zadaných príkazov v príkazovom riadku.



Obr.8: Návrh obrazovky Results

Po otvorení tejto obrazovky sa projekt ešte nespustí, ten sa spustí až príkazom run, po spustení doňho ešte môže študent zasahovať tým, že si bude môcť nechať vypísať rôzne hodnoty z projektu, prípadne si ich nechá vykresliť. Zasahovať do aktuálneho výpočtu bude môcť dovtedy, kým znova nezadá príkaz run, vtedy sa začne nový výpočet algoritmu.

Zoznamy projektov budú v dvoch podobách, ako vlastné projekty a zdieľané projekty. Obe budú mať vlastné tabuľky, rozdiel bude v tom, že tabuľka vlastných projektov bude umožňovať editáciu, spustenie, mazanie projektov a tiež bude umožňovať projekt zdieľať. Tabuľka zdieľaných projektov bude umožňovať iným užívateľom ako autorovi daného projektu, zobrazenie zdrojových súborov a spustenie takéhoto projektu v obrazovke Results, prípadne si bude môcť iný užívateľ takto vyzdieľaný projekt skopírovať do svojho adresára, z ktorého už bude môcť takúto kópiu aj editovať.

Lectures

Táto sekcia aplikácie bude slúžiť študentom na ukážku už hotových projektov spolu s vysvetlením ako sa takýto projekt vyrába a ako funguje. Obsahovať bude 4 ukážkové príklady. Konkrétne by malo ísť o príklady s gradovanou úrovňou s tým, že pôjde o príklady na rôznych dátových štruktúrach.

Jednoduchšie príklady, tak budú zobrazovať prácu s poliami, prípadne maticami. Konkrétne pôjde o príklady s ukážkami niektorého z triediacich algoritmov, prípadne čo sa týka matic, zobrazenie algoritmu pre násobenie matic. Ďalšie príklady by sa mali zamerať na prácu s grafmi, ako napríklad algoritmus hľadania najkratšej cesty alebo algoritmus hľadania Hamiltonovskej kužnice.

Takýto hotový projekt bude obsahovať popis spolu s obrázkami dátových štruktúr. Keďže vo vlastných projektoch si študent vyberá aké dátové štruktúry využije v editore štruktúr, tak dostane na ukážku už naplnený editor so štruktúrami pre tento príklad. Obrázky budú tiež vysvetľovať priebeh algoritmu. Následne bude mať študent k dispozícii aj zdrojové súbory spolu s komentármi.

Študent si bude môcť takýto projekt spustiť, tým že sa dostane do obrazovky Results, do ktorej už bude môcť sám zasahovať tým, že bude môcť vykonávať všetky príkazy, ktoré obrazovka Results dovoľuje aj pri vlastných projektoch.

Jednotlivé príklady, ktoré časť Lectures obsahuje si môže študent skopírovať do svojho adresára podobne ako to je s projektmi, ktoré sú v tabuľke zdieľaných projektov.

4 Implementácia

Celá aplikácia je vytvorená v jazyku JavaScript a postavená na platforme node.js. Vďaka tejto platforme je možné jednoducho pracovať na backend-ovej časti aplikácie.

Aplikácia sa inicializuje tak, že v node.js pomocou modulu *http* zavolám server, na ktorom chcem aplikáciu spustiť, pomocou príkazu `createServer`. Následne keď už mám špecifikovaný server, na ktorom aplikácia beží, nastavím prepojenie s databázou pomocou modulu *mysql*. Vďaka tomuto modulu, môžem uchovávať informácie o užívateľoch, no hlavne o projektoch. S databázou spolupracujem pri vytváraní, editovaní, mazaní projektov.

Základom celej aplikácie je komunikácia pomocou socket.io, na tejto komunikácii je postavená celá aplikácia. Socket.io funguje tak, že na strane servera vyvolá nejakú udalosť a čaká, kedy na ňu zareaguje klient. Ako náhle dôjde ku spojeniu, klient a server si môžu dáta začať posielat'. Všetko zaleží na príkazoch **emit** a **on**. *Emit* slúži na posielanie a *on* slúži na prijímanie informácií.

Vďaka ďalšiemu modulu *mkdir*, vytváram adresáre a zdrojové súbory jednotlivým projektom. Jednotlivé projekty potom rovnako fungujú na socket.io, takým spôsobom, že po spustení projektu ešte dovolím užívateľovi zasahovať do obsahu programu. Takže ak mám nejaký program, ktorý som ešte nespustil, viem mu cez socket.io povedať, aby projekt spustil a zapamätal si obsah všetkých premenných v programe. Tieto informácie ostanú na strane servera a ak teraz na strane klienta požiadam o výpis niektorej premennej, pošlem cez socket.io informáciu, že chcem obsah danej premennej a server mi ho pošle cez socket.io. A takto to môžem niekoľkokrát opakovať.

Všetku prácu servera mám naimplementovanú v súbore `MainController.js`, ktorý slúži ako riadiaci proces celej aplikácie. Následne v každom html súbore, ktorý aplikácia obsahuje mám pripravenú komunikáciu so serverom na klientskej úrovni. A samotný `MainController.js` vie zasiahnuť do každého z nich.

4.1 Jazyk

Programovací jazyk, ktorý aplikácia využíva na vyučovanie paralelného programovania, vychádza z jazyka JavaScript. Konkrétne som jazyk vyvinul v samostatnom module JavaScript-u. Vychádzal som pri tom zo základov modulu node-opengl.

Modul na pozadí programu využíva JavaScript-ovú syntax aj so všetkými jeho základnými funkciami, spolu s jazykom OpenCL, ktorý má na starosti výpočty na GPU a teda samotné paralelné programovanie. Modul je zložený z funkcií, ktoré sú na pozadí programu, ktorý študent píše, preložené do jazyka OpenCL, ale vykonávajú aj niektoré dôležité úkony, ktorým nechcem študenta zaťažovať, ako napríklad vytváranie dátových štruktúr, čo je vlastne hlavnou úlohou tohto jazyka, zjednodušiť študentovi prácu.

Modul som príznačne nazval **parallel**. Pri vytváraní programu, ho študent nemusí nijako inicializovať, ten sa automaticky sám načíta pri spustení programu. Funkcie, ktoré tento modul ponúka, sa potom jednoducho zavolajú príkazom *parallel.nazovFunkcie()*.

Modul si pri spustení sám získa údaje o zariadení, v tomto prípade GPU a informácie si ponechá uložené, keďže s nimi následne bude potrebovať pracovať. Jazyku som zadefinoval všetky funkcie, ktoré sú zadefinované v návrhu jazyka v predošlej kapitole. Pridal som však niektoré ďalšie príkazy, ktoré sa mi zdali potrebné počas implementácie. Napríklad ide o vyčistenie Canvasu pri vykresľovaní, na to som vytvoril jednoduchý príkaz *parallel.clear()* a rovnako aj *parallel.clearAll()*, ktorý vyčistí Canvas aj výpisový box na obrazovke Results.

Takýto typ riešenia jazyka som si vybral kvôli tomu, že aplikácia je založená na node.js, teda JavaScript-e, preto by nebolo dobré, aby jazyk sa ešte prekladal do JavaScript-u, keďže ide o paralelné programovanie, ktorého hlavnou devízou je rýchlosť a šetrenie behu programu a preklad by program iba zbytočne predĺžil. A zároveň má byť kód, ktorý bude študent programovať jednoduchý na pochopenie.

Modul je vyrobený tak, že v prípade potreby je jednoduché doňho zasiahnuť a pridať ďalšie metódy alebo modifikovať existujúce. Modul s jazykom má svoje samostatné miesto na serveri a pri každom spustení programu vie doňho akýkoľvek projekt vstúpiť.

4.2 Úvodná obrazovka

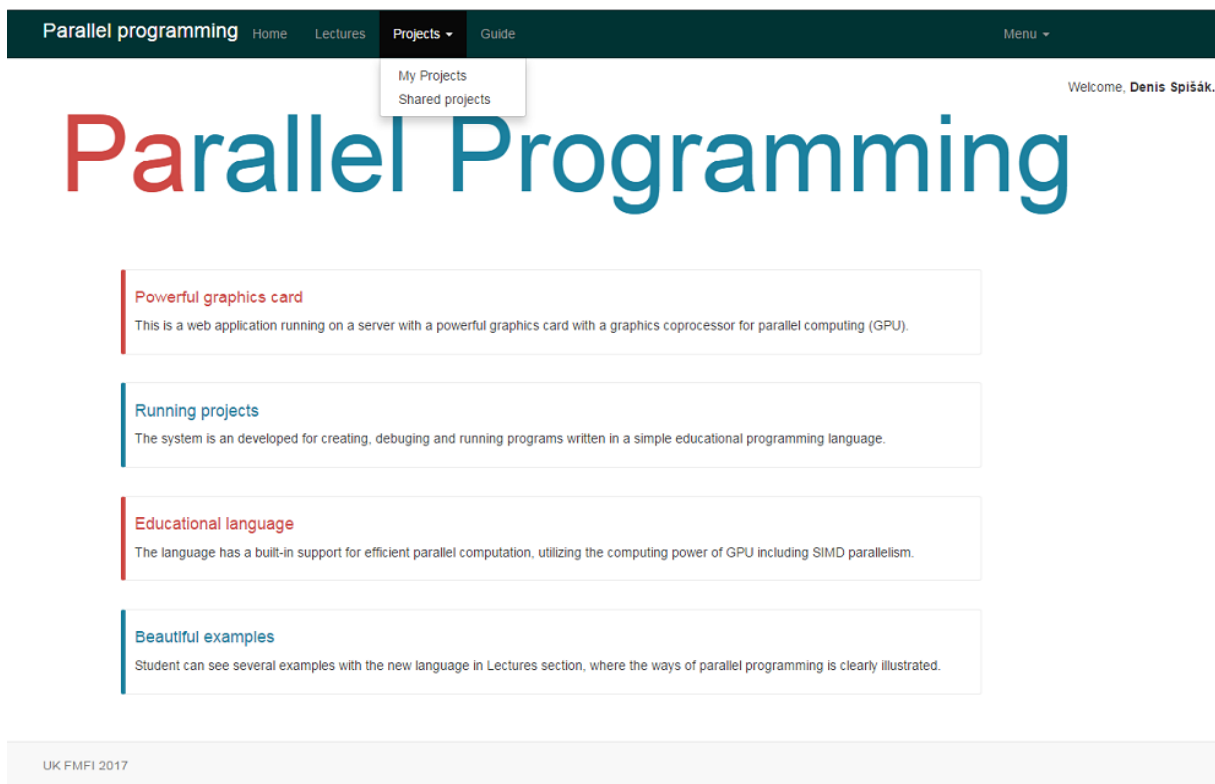
Úvodná obrazovka ponúka úplne základnú funkcionality, vyvinul som ju tak, aby užívateľ hneď videl aké má možnosti. Obsah funkcionality stránky je v angličtine.

Neprihlásený užívateľ má možnosť prihlásenia alebo registrácie. Formulár pre prihlásenie je podľa návrhu úplne jednoduchý, vyžaduje len email, pomocou, ktorého sa bude užívateľ prihlasovať, meno a priezvisko, v prípade kontroly a heslo.

The image shows a registration form titled "Sign Up" in a light blue header. In the top right corner of the header is a blue button labeled "Login". The form contains four input fields with labels to their left: "Email", "First Name", "Last Name", and "Password". The placeholder text for the first three fields is "Enter email", "Enter first name", and "Enter last name" respectively. The placeholder for the password field is "Password". Below the password field is a button labeled "Sign up".

Obr.9: Registračný formulár

Po registrácii sa už môže užívateľ priamo prihlásiť. Medzi registráciou a prihlásením sa užívateľ jednoducho preklikne, ako je možné vidieť na predošlom obrázku (obr.9), v pravom hornom rohu. Po prihlásení sa užívateľ dostáva na úvodnú obrazovku, kde má v navigácii možnosť dostať sa či už k prednáškovým príkladom v časti Lectures alebo môže rovno prejsť k projektom, vlastným alebo zdieľaným. Poslednou možnosťou je pomocník Guide, v ktorom užívateľ nájde rady, ako funguje samotná aplikácia alebo priamo programovanie v modifikovanom jazyku, ktorý aplikácia využíva. Položka menu obsahuje rovnaké podsekcie ako základná navigácia. Úvodnú navigáciu je možné vidieť na nasledujúcom obrázku (obr.10).



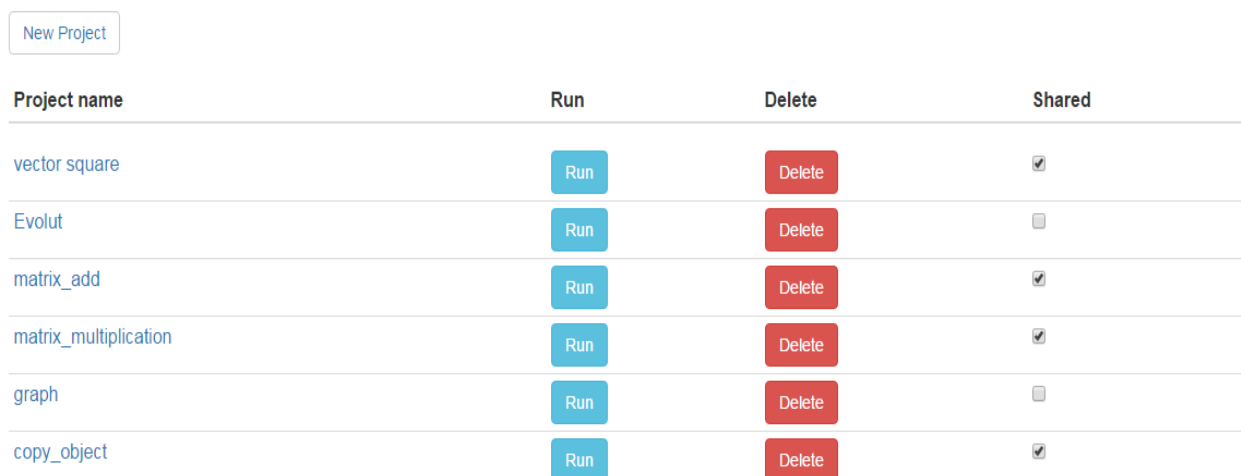
Obr.10: Úvodná obrazovka so základnou navigáciou

4.3 Obrazovka Projects

Pre lepšie vysvetlenie celkovej funkčnosti teraz začnem s obrazovkou Projects. Ako som už povedal, časť projektov ponúka zoznamy projektov v dvoch podobách, projekty patriace užívateľovi, v aplikácii je to časť *My projects* a projekty zdieľané užívateľmi pre ostatných, v aplikácii *Shared projects*.

4.3.1 Obrazovka My Projects

V zozname vlastných projektov, teda My projects, si môže užívateľ okrem vytvárania nových a editácie starších projektov, tieto staršie aj mazať, priamo v zozname, pomocou tlačidla *Delete* a zároveň, má možnosť označiť projekt za zdieľaný a tým sa projekt začne zobrazovať aj ostatným užívateľom v zozname Shared projects. Projekt je možné spustiť aj priamo zo zoznamu projektov tlačidlom Run. Tabuľka so zoznamom projektov je zobrazená na nasledujúcom obrázku (obr.11).



The screenshot shows a web interface for 'My Projects'. At the top left is a 'New Project' button. Below it is a table with four columns: 'Project name', 'Run', 'Delete', and 'Shared'. The table lists six projects: 'vector square', 'Evolut', 'matrix_add', 'matrix_multiplication', 'graph', and 'copy_object'. Each project row has a blue 'Run' button, a red 'Delete' button, and a checkbox in the 'Shared' column. The 'Shared' checkboxes for 'vector square', 'matrix_add', 'matrix_multiplication', and 'copy_object' are checked, while for 'Evolut' and 'graph' they are not.

Project name	Run	Delete	Shared
vector square	Run	Delete	<input checked="" type="checkbox"/>
Evolut	Run	Delete	<input type="checkbox"/>
matrix_add	Run	Delete	<input checked="" type="checkbox"/>
matrix_multiplication	Run	Delete	<input checked="" type="checkbox"/>
graph	Run	Delete	<input type="checkbox"/>
copy_object	Run	Delete	<input checked="" type="checkbox"/>

Obr.11: Zoznam projektov užívateľa

Vytváranie nového projektu, po stlačení tlačidla New Project, privedie užívateľa k prvému oknu, kde zadáva názov projektu a tiež aj bližší popis programu, k tomuto oknu sa vie užívateľ kedykoľvek vrátiť. Potvrdením, tlačidlom *Next*, pokračuje do ďalšej časti prípravy projektu. V tejto fáze sa projekt s názvom a popisom zapíše do databázy a vyhradí si na serveri svoj vlastný adresár, do ktorého bude ukladať všetky zdrojové súbory a obrázky. Jednotlivé adresáre projektov dostávajú názov podľa id projektu.

V nasledujúcom okne sa už dostáva k editoru na vytváranie základných premenných pre svoj projekt, tento editor som nazval editor štruktúr, takto sa budem naňho odvolávať aj v nasledujúcich častiach práce. Ako bolo uvedené v návrhu, študent si vyberá spomedzi nasledujúcich dátových štruktúr: jednorozmerné a dvojrozmerné pole, graf, množina, zoznam a obrázok. Výber dátových štruktúr spočíva v jednoduchom ťahaní boxov s názvami štruktúr, do stĺpca CPU alebo GPU (na obr.12 ide o CPU panel a GPU panel), podľa toho, kde bude premenná využívaná. Následne po jej zaradení, jej môže užívateľ zadať názov do zobrazeného okna priamo v boxe. Ak pôjde o obrázok, ako štruktúru, užívateľ zadá najprv meno a následne naimportuje obrázok do adresára s projektom.

Jednotlivé boxy so štruktúrami v niektorom z panelov CPU alebo GPU, môže študent kedykoľvek opäť zobrať a presunúť do iného panelu, ak chce niektorí z boxov v paneli vyhodíť stačí ho potiahnuť do zásobníka štruktúr (na obr.12 ide o panel s ružovým pozadím).

Select structures



Obr.12: Editor na výber dátových štruktúr

Po výbere štruktúr a priradení ich mien sa dátové štruktúry uložia spolu s názvami do databázy projektu, následne pri editácii, sa do tohto editora dotiahnu všetky vybrané dátové štruktúry spolu s priradeným menom. Takto zadané štruktúry sa potom sami načítajú do zdrojových súborov projektu.

Ďalšia časť je už zameraná na samotné programovanie. Programovanie projektu je rozdelené do 4 častí: príprava dát pre GPU, príprava dát pre CPU, program bežiaci na CPU a program bežiaci na GPU.

Function prepare

Táto funkcia je založená na príprave a kopírovaní dát z CPU na GPU. Pri vytváraní premenných v editore z predošlej časti sa premenné automaticky vložia do zdrojového kódu tejto funkcie, takže ich užívateľ už nemusí znova definovať, definuje premenné už len

v prípade, že chce vytvoriť nejakú pomocnú premennú ako *string* a podobne. V rámci tejto funkcie si užívateľ môže naplniť premenné, patriace CPU, dátami, s ktorými bude potom program pracovať. K tomu sú vytvorené funkcie ako *parallel.randomFill*, *parallel.zeroFill* a ďalšie, ktoré slúžia k naplneniu štruktúr, všetky sú definované v návrhu jazyka. Následne prekopíruje dáta z premenných na CPU do premenných na GPU, nato slúži príkaz *parallel.copy*.

Function take

Funkcia *take*, je opakom k funkcii *prepare*. Tu užívateľ kopíruje, respektíve číta dáta z GPU pre CPU. Aj v tomto prípade sa premenné pre GPU, priamo zapisujú do zdrojového kódu pri ich vytváraní v editore s dátovými štruktúrami. Ich obsah sa naplňuje v predošlej funkcii *prepare*, prekopírovaním dát z CPU na GPU. Príkazy tejto funkcie teda prebehnú až po skončení programu na GPU, keďže dáta sa zmenia až po jeho vykonaní a základnou úlohou tejto funkcie je výsledné dáta prečítať z GPU premennej na CPU premennú, nato študentovi poslúži príkaz *parallel.read*. Môže ísť len o jednu premennú, ale aj o všetky premenné, záleží na programe a čo je úlohou takého programu.

Function core

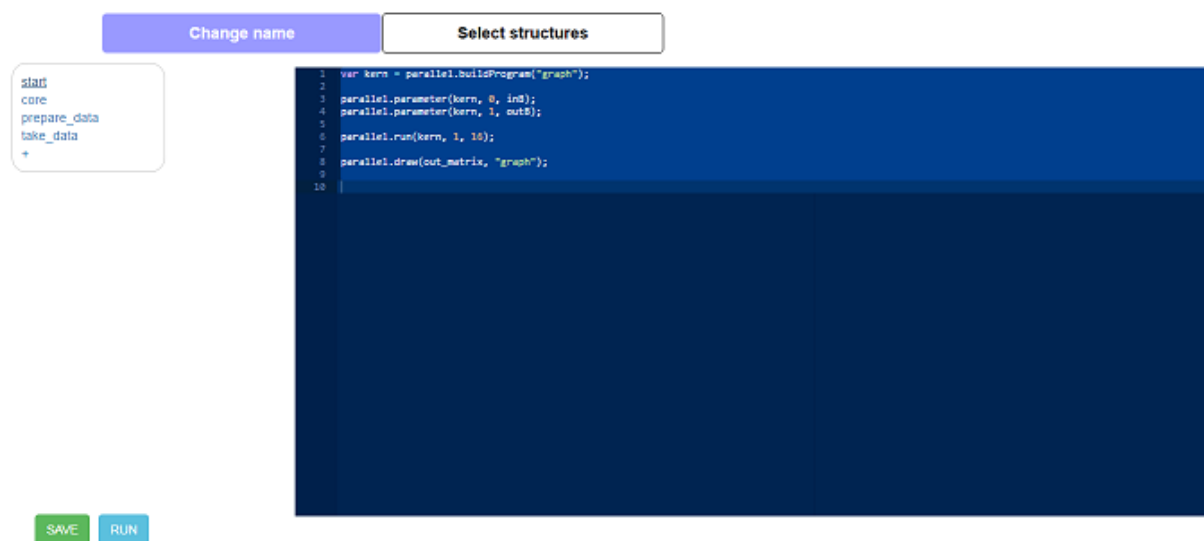
Táto funkcia beží na GPU zariadení, v OpenCL sa táto funkcia nazýva kernel. Táto funkcia môže obsahovať ďalšie funkcie, ale jedna z nich musí byť hlavná, ktorá sa potom bude volať pri spustení programu vo funkcii *start*. Hlavná funkcia by mala byť typu *void*. Tá sa potom na pozadí preloží do podoby kernelu OpenCL, aby s ním vedeli OpenCL funkcie pracovať. Základom tejto funkcie je získanie dát o procesore. K tomu slúži funkcia *getWorkerNumber*. Editor tohto programu je nastavený pre programovanie C jazyka, nakoľko jeho syntax je takmer totožná. Aj v tomto prípade som sa jazyk snažil čo najviac priblížiť jazyku OpenCL a veľmi jeho základy nemeniť, aby som program zbytočne nespomaľoval.

Function start

Start je hlavná funkcia, je možné ju označiť aj ako riadiaca funkcia, je mozgom celého programu. Po vytvorení a príprave dát, sa zavolá funkcia z *core*, ktorá bude spúšťačom GPU časti programu. Zároveň po inicializácii funkcie sa zadefinujú parametre tejto metódy

príkazom *parallel.parameter* a program sa môže spustiť. Po jeho skončení je ešte možné výsledky vrátiť jednoduchým JavaScript-ovým príkazom *return*. Ten výsledky spracuje a pošle cez socket.io do obrazovky Results.

Zoznam funkcií je vytvorený ako menu v ľavej časti obrazovky. Táto obrazovka obsahuje okrem tlačidla pre spustenie programu *Run* aj tlačidlo na ukladanie projektu *Save*.



Obr.13: Obrazovka programovacej časti

Ako je možné vidieť na predošlom obrázku, medzi jednotlivými obrazovkami, či už na zadanie názvu projektu, výberu premenných alebo výrobu funkcií, sa je možné preklíkavať pomocou menu v hornej časti obrazovky.

Následne po spustení projektu, tlačidlom Run sa užívateľovi zobrazí okno s výsledkami. Obrazovka s výsledkami sa otvára vždy do nového okna prehliadača. Kvôli tomu, aby študent mohol súčasne vidieť do zdrojového kódu a aj výpisovej obrazovky.

Editácia projektu sa veľmi od vytvárania nového projektu nerozlišuje. Editácia sa otvára kliknutím na názov projektu v zozname My Projects. Po otvorení sa užívateľ hneď

dostáva do sekcie editora štruktúr, kde sú už v boxoch pre CPU a GPU priamo z databázy dotiahnuté uložené štruktúry aj s názvami.

V prípade, že chce užívateľ zmeniť meno projektu alebo jeho popis, v hornom menu je možnosť prekliknúť sa do danej obrazovky a meno zmeniť. Obrazovka s funkciami už zobrazuje uložené zdrojové súbory. Táto navigácia slúži aj ako ukladacie tlačidlo, keďže po každom kliknutí dochádza k ukladaniu aktuálneho obsahu.

4.3.2 Shared projects

V zozname zdieľaných projektov užívateľ vidí okrem projektov ostatných užívateľov aj svoje projekty. Rozdiel medzi jeho projektami a ostatnými v tomto zozname, spočíva v tom, že ak klikne na svoj projekt tak ho môže editovať a dostáva sa do rovnakej sekcie, ako keď klikne na editáciu projektu v zozname svojich projektov.

Na rozdiel od projektov ostatných, ktoré si po otvorení môže len prezerať, s tým, že do nich nemá právo vpisovať alebo upravovať. Zároveň môže projekt priamo spustiť. Tým sa dostane do obrazovky Results, rovnako ako pri svojich projektoch. To čo môže robiť na obrazovke Results by mu mal povedať sám autor projektu v popise projektu, ktorý sa zobrazí na tejto obrazovke.

Zoznam týchto zdieľaných projektov umožňuje okrem prezerania kódu, si daný projekt skopírovať do zoznamu svojich projektov, s tým, že sa v jeho zozname vytvorí kópia pôvodného projektu, takže užívateľ môže následne tento projekt vo svojom zozname editovať, ale mení sa len jeho kópia a pôvodný projekt originálneho užívateľa sa nemení. Ak užívateľ svoj projekt zdieľa, pridá ho do zoznamu zdieľaných projektov, súhlasí tým s takouto možnosťou, že daný projekt si skopíruje iný užívateľ.

Project name	Run	Copy
project id:4	Run	Copy
novy projekt pre id 3	Run	Copy
MinSort	Run	
vector addition	Run	
vector square	Run	
matrix_add	Run	
matrix multiplication	Run	
copy_object	Run	

Obr.14: Zoznam zdieľaných projektov

4.3.3 Obrázovka Results

Táto obrazovka sa študentovi otvorí v momente, keď spustí program tlačidlom Run, či už v časti prednášok, zdieľaných projektoch alebo vlastných projektoch. Táto obrazovka ponúka študentovi akúsi formu interakcie, aby študent iba nezaklikol RUN a stránka mu vráti hotový výsledok programu a on si ho len pozrie a môže sa vrátiť späť ku kódu.

Obrázovka je rozdelená do viacerých blokov. Nadpis stránky tvorí názov projektu. Pod názvom je blok, ktorý slúži na výpis výsledkov. Zároveň pravá časť je vyhradená pre popis projektu, tento popis môže autor projektu meniť v časti *Change project name*. Tento popis slúži v prípade, že študent projekt zverejní a chce priblížiť inému užívateľovi jeho funkcionality, respektíve, čo všetko je v tejto časti možné vykonať, samozrejme sa tento popis využije aj v časti prednáškových príkladoch.

Obrázovka ešte obsahuje Canvas, v ktorom je možné výsledky graficky spracovať. Na spodku strany je príkazový riadok, doňho môže študent zapisovať príkazy, ktoré chce, aby sa ešte spracovali. Tento riadok spracúva aj základné operácie ako sčítanie, násobenie a podobne, ale prioritne je pripravený pre prácu so samotným programom. Na toto vyhodnocovanie má pripravenú funkciu, ktorá obsah hodí do reťazca a snaží sa ho spracovať. Táto funkcia dokáže

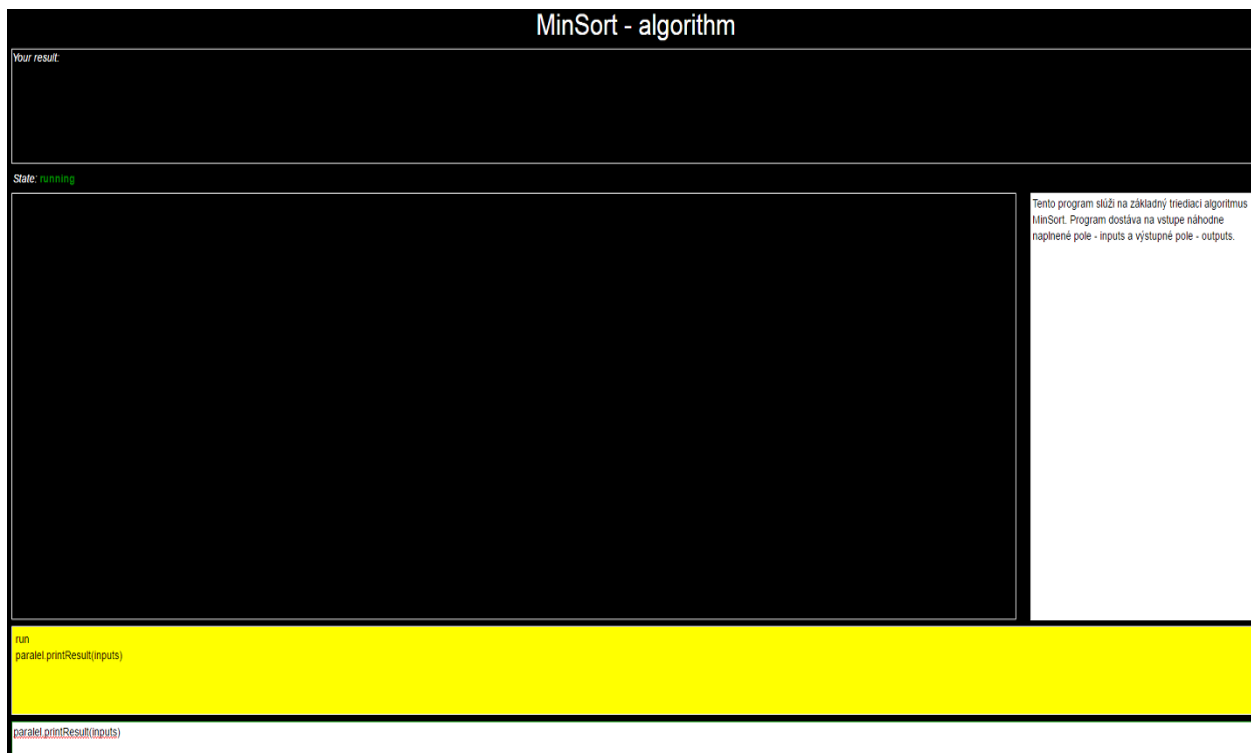
spracovať základné operácie ako sčítanie, odčítanie, násobenie a ďalšie matematické operácie, rovnako vie spracovať aj jednoduchý výpis hodnôt. Hodnotu, ktorú spracuje potom vypíše do bloku s výpisom hodnôt vo vrchnej časti obrazovky. Spustenie príkazu sa potvrdí tlačidlom *Enter*.

Keďže má obrazovka statickú veľkosť, responsabilnú pre rôzne obrazovky, výpis hodnôt je možné v tomto bloku skrolovať. Rovnako ako popis projektu, v prípade, žeby bol popis príliš dlhý.

Popis, ktorý študent pre daný projekt vytvorí, môže zapisovať v html podobe, aby ho potom vedelo okno s popisom lepšie spracovať, v prípade, žeby chcel autor popis farebne rozlíšiť a podobne.

Obrazovka tiež obsahuje stavový riadok nad Canvasom, ktorý popisuje čo sa práve s programom deje. Konkrétne ide o stavy: *ready*, *running* a *drawing*. Na obrazovke sú vypísané zeleným písmom, pre lepšie rozlíšenie.

Príkazy, ktoré študent zadáva sa uchovávajú v histórii, v prípade, žeby ich chcel zopakovať, konkrétne ide o posledných päť príkazov, aby táto časť nezaberala zbytočne veľa miesta. Na obr.15, patrí histórii príkazov blok so žltým pozadím, pod ním sa nachádza spomínaný príkazový riadok. V príkazovom riadku fungujú aj príkazy \uparrow a \downarrow pre pohyb v histórii príkazov, aby ich študent nemusel stále zapisovať, ak sa chce vrátiť k niektorému už použitému.



Obr.15: Ukážka obrazovky Results

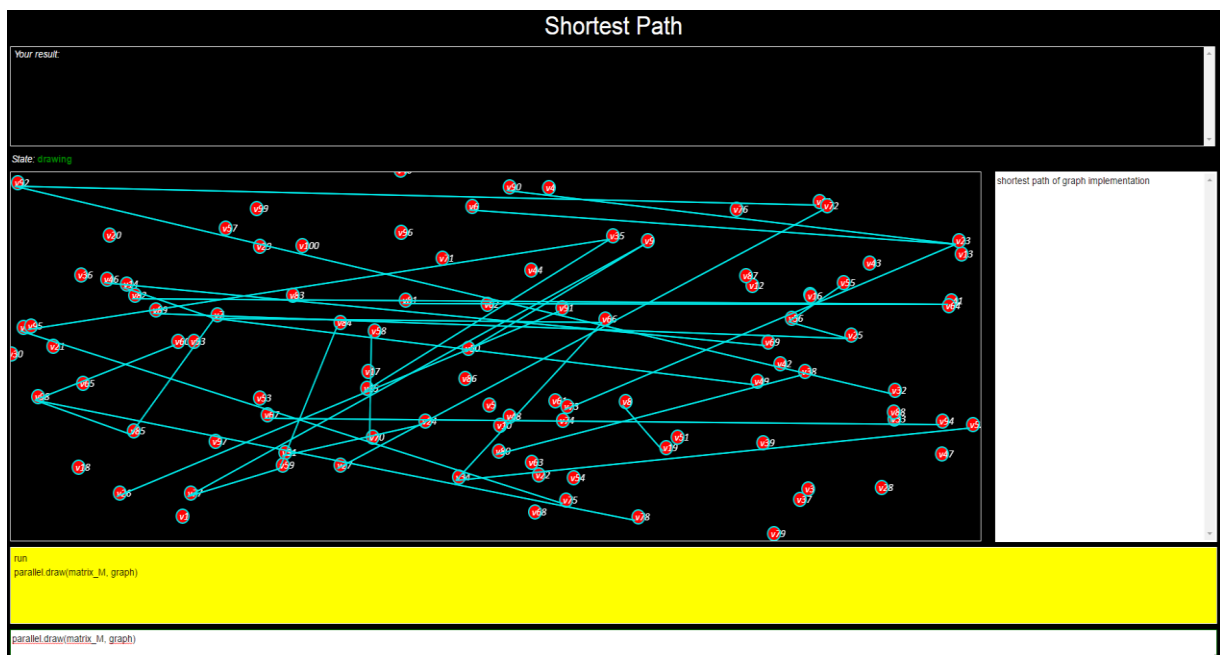
Príkazový riadok spracuje základné operácie a tiež príkazy, ktoré som mu samostatne vyvinul. Tento príkazový riadok slúži nato, aby doňho študent zadával také príkazy, ktoré nedefinoval v samotnom kóde, práve pre prípad lepšej kontroly, respektíve pri prednáškach je možné využiť všetky príkazy. Ide o príkazy, ktoré som špecifikoval v návrhu:

- **parallel.run** – týmto príkazom sa spustí program na pozadí, stavový riadok zmení svoj stav z *ready* na *running*.
- **parallel.printResult** – tento príkaz vypíše do bloku s výsledkami svoju odpoveď. Ako parameter sa zadáva premenná, ktorú chcem nechať vypísať. Každý výpis sa robí v novom riadku.
- **parallel.draw** – tento príkaz slúži na vykresľovanie dát v Canvase. Vykresľovať dokáže polia a grafy. Polia vykresľuje v podobe škatuliek. Polia môžu byť jednorozmerné alebo dvojrozmerné respektíve matice. Ako parameter dostáva premennú, ktorá obsahuje výsledok potrebný na vykreslenie a typ vykreslenia.

Teda buď jednorozmerné pole - 1D, dvojrozmerné pole, matica – 2D / matrix, binárne pole – bin alebo graf – graph. Graf vykresľuje priamo s vrcholmi a hranami. Binárne pole som vyvinul pre jeden z hotových príkladov, ktoré slúžia na ukážku. Tento príkaz vykresľuje škatuľky celé biele alebo celé čierne, podľa binárnej hodnoty, k vysvetlenie príkladu sa dostanem neskôr.

- **parallel.showPath** – tento príkaz slúži na vykresľovanie ciest medzi vrcholmi v grafe
- **parallel.clear** – vyčistí plochu canvasu.
- **parallel.clearAll** – vyčistí okrem canvasu aj blok s výpismi.
- Ostatné príkazy vždy prevedie do stringu a snaží sa ich vyhodnotiť a vypísať do bloku pre výsledky, tak som špecifikoval vyššie.

Okrem spomenutých blokov v časti prednášok obrazovka results vykonáva časovú kontrolu, teda program má spustený časovač, ktorý po skončení vypíše čas zbehnutia programu na plochu. Tieto časové hodnoty, je potom možné porovnávať so sekvenčným typom programu.



Obr.16: Ukážka vykreslenia grafu

Obr.17: Ukážka vykreslenia najkratšej cesty v grafe

4.3 Obrazovka Lectures

Táto časť obsahuje 4 už hotové projekty, ktoré budú užívateľovi slúžiť ako ukázkový príklad programovania v modifikovanom jazyku a samotného paralelného programovania. Príklady majú vzostupnú zložitosť. Ide o príklady na ukážku programovania polí, respektíve matic a grafov.

Obrazovka takéhoto príkladu má trochu rozšírenejší obsah ako tá ktorú študent uvidí pri vytváraní vlastného príkladu. Ide o to, že v prednáškových príkladoch sú jednotlivé príklady konkrétne vysvetlené. Vysvetlenie sa nachádza v podobe textu, popisu funkčnosti a vlastností programu a tiež aj v podobe obrázkov. Vo vlastných projektoch si študent sám navolí štruktúry v editore štruktúr, v prednáškovom príklade dostáva tieto štruktúry už vybrané ale vidí ich v takomto editore, len s tým, že doňho už nemôže zasahovať, slúži vlastne iba na ukážku.

Obrazovka tiež obsahuje obrázkový popis funkčnosti príkladu, pre lepšie pochopenie algoritmu. Zdrojové súbory si študent otvorí jednoduchým rozkliknutím tlačidla Show Sources. Tie už študent vidí rovnako ako pri vlastnom projekte, až nato, že kódy jednotlivých príkladov nemôže meniť.

Spustiť takýto program dokáže jednoducho klikom na tlačidlo Run. Následne sa študentovi otvorí obrazovka Results.

4.3.1 Select Sort

Prvý z príkladov, ktoré som vytvoril, slúži na ukážku triediaceho algoritmu, konkrétne som si vybral SelectSort. V tomto príklade som si vytvoril dve jednorozmerné polia pre CPU a dve pre GPU. Ide o vstupné a výstupné pole na CPU, ktoré sa prekopírujú do polí na GPU.

V sekvenčnej podobe algoritmu, by výpočet vykonával jeden procesor s jedným vláknom. Procesor by si vždy vybral jeden prvok poľa a porovnával by ho s ostatnými

prvkami poľa a nakoniec by ho umiestnil na svoju pozíciu. Takto by to opakoval so všetkými prvkami, kým by neutriedil celé pole.

Tento algoritmus som dal do paralelnej podoby nasledovne. Pred spustením programu som v riadiacej časti, v mojej aplikácii ide o funkciu `start`, povedal programu, že bude bežať na určitom počte procesorov takej dĺžky, akej dĺžky je pole, pretože každý jeden procesor bude triediť jeden prvok poľa a všetci zbehnú v rovnakom čase, nikto nie je závislý na inom procesore. Na začiatku algoritmu, ktorý už beží na GPU, zistím id worker-a procesoru (`getWorkerNumber`). Toto id použijem ako poradie prvku v poli, ktorý bude použitý ako vybraný prvok v triedení, teda pre vstupné pole *input*, vyberám prvok *input[id]*, toto bude prvok, s ktorým bude daný worker porovnávať ostatné prvky a hľadať preňho správnu pozíciu. Teraz prechádzam pre daný worker - *input[id]*, všetky prvky poľa a porovnávam ich hodnoty nasledovne:

```
for (int item=0; item<size; item++){
    if ((input[item] < input[id]) || (input[item] == input[id] && item < id)){
        pos += 1;
    }
}
```

Prvý riadok tohto algoritmu patrí cyklu na prechádzaní všetkých prvkov poľa. Prvok *size*, je v tomto prípade počet všetkých prvkov poľa. Aktuálny prvok cyklu je *input[item]*, s ktorým má byť *input[id]* porovnaný. Nasledujúci riadok už porovnáva, ktorý z prvkov je menší, v prípade, že *input[id]* je väčší ako porovnávaný prvok zvýším jeho novú pozíciu, *pos*, o jedna. Môže však dôjsť k prípadu, že mám v poli viacero prvkov s rovnakou hodnotou, *input[item] == input[id]*, vtedy zavediem pravidlo, že ak sú prvky rovnaké, tak na nižšiu pozíciu v usporiadanom poli sa dostane ten prvok, ktorý bol v pôvodnom vstupnom poli skôr. Nakoniec priradím do výstupného poľa *output*, na pozíciu *pos* z cykla, hodnotu *input[id]*.

4.3.2 Conway's Game of life

Ako ďalší príklad som si vybral Conwayovu Game of life, kde ide o celúrný automat, ktorý je postavený na mriežke buniek, v ktorej každá bunka má priradený jeden zo stavov živý/mŕtvy. Každá bunka sa riadi nasledovnými pravidlami:

- ak je bunka živá a má menej ako dvoch žijúcich susedov zomiera;
- ak je bunka živá a má dvoch alebo troch žijúcich susedov zostáva živá do ďalšej generácie;
- ak je bunka živá a má viac ako troch žijúcich susedov zomiera;
- ak je bunka mŕtva a má troch živých susedov, tak ožije.

Svoj príklad som postavil na takom princípe, že každá mriežku tvorí matica prvkov, ktoré majú hodnotu 1 alebo 0. A ich život (hodnota 1 – živá, hodnota 0 - mŕtva), už závisí na predošlom algoritme. Paralelizmus som potom vyriešil v takej podobe, že každému prvku matice, každej bunke mriežky, som priradil jeden procesor a každú generáciu tvorí jedna sada procesorov. Pred spustením algoritmu, programu poviem, koľko generácii chcem spustiť, každá generácia sa potom vykreslí v canvase, takým spôsobom, že vykreslím mriežku s bunkami, tak, že ak je bunka živá = 1, tak je bunka vyfarbená na bielo, ak je mŕtva = 0, tak je čierna. A pri každej generácii sa mriežka prekreslí.

Algoritmus tvorí jedno vstupné a jedno výstupné pole na CPU, rovnako tak aj na GPU. Všetky polia sú dvojrozmerné. Vstupné pole sa v každej generácii prepíše, podľa daných pravidiel. Výstupné pole som vytvoril preto, aby som nespomaľoval algoritmus vykresľovaním canvasu, tak po každom cykle zapíšem aktuálnu generáciu do výstupného poľa. Výstupné pole je potom postavené z toľkých prvkov, koľko generácii algoritmus má vynásobené počtom prvkov v každej generácii.

Po skončení algoritmu prebehne vykresľovanie generácii. Každá generácia má stanovený určitý čas a potom sa prekreslí ďalšou generáciou.

4.3.3 Najkratšia cesta v grafe

Tretím príkladom je hľadanie najkratšej cesty v grafe. Existuje viacero algoritmov na riešenie takéhoto problému v sekvenčnej podobe. Rozhodol som sa vybrať si Floyd-Warshall algoritmus. Tento algoritmus ráta s tým, že mám nejaký orientovaný graf s hranami, ktoré majú rôznu váhu. Algoritmus tak prechádza každú dvojicu a hľadá medzi nimi najkratšiu cestu tak, že vždy pridá do cesty nejaký bod z grafu, mimo dvojice a kontroluje ich vzdialenosť, ak nájde kratšiu trasu zapíše ju, takto to skontroluje pre všetky body grafu s každou dvojicou.

V mojom príklade vytváram náhodný graf s určitým počtom vrcholov a hrán. Na CPU mám maticu (dvojmerné pole), ktorá obsahuje zoznam najkratších existujúcich ciest, túto maticu môžem nazvať ako matica susedností a druhú maticu, ktorá slúži na zápis navrhnutých ciest, teda v prípade, že medzi dvoma bodmi neexistuje hrana, tak algoritmus po nájdení najkratšej cesty, do tejto matice zapíše bod, cez ktorý sa musí prejsť. Pre obe matice som vytvoril ich GPU kópiu.

Tento algoritmus som zparallelizoval tak, že používam pre každý procesor dve vlákna, ktoré reprezentujú dva vrcholy, pre ktoré hľadám najkratšiu cestu. Teraz každej dvojici vrcholov priradím ďalší cez ktorý majú prejsť.

```
for (int k = 0; k < count; k++){
    if (M[i+j*count] > M[i+k*count] + M[k+j*count]){
        R[i+j*count] = k+1;
    }
    barrier();
}
```

Cyklus prechádza všetky vrcholy grafu k , kde $count$ je počet všetkých vrcholov. Aby som mohol ku poľu pristupovať ako ku matici musím použiť nasledovný zápis $M[i+j*count]$, kde M je matica susedností, i reprezentuje stĺpec matice, $j*count$ patrí riadku. A teraz kontrolujem či medzi bodmi i a j existuje kratšia cesta cez bod k , $M[i+j*count] > M[i+k*count] + M[k+j*count]$, ak takáto cesta existuje, tak ju zapíšem do matice navrhovaných ciest R . V riadku $R[i+j*count] = k+1$, vrchol k musím navýšiť o jednu pretože vrcholy mám očíslované od 0 v cykle k . Pre každú dvojicu vlákien ešte môžem použiť príkaz *barrier*, ktorý sa spustí v prípade, že niektorá cesta sa zacyklí príliš dlho, čo umožní procesorom na seba počkať.

V prípade, že sa cesta nenájde okno s výpismi na obrazovke Results zobrazí správu: Hľadaná cesta neexistuje. Ak existuje, tak ju vykreslí na červeno.

4.3.4 Hamiltonovská kružnica

Štvrtým príkladom je algoritmus na hľadanie hamiltonovskej kružnice v grafe. Môj algoritmus bude zisťovať, či sa vôbec nejaká kružnica v grafe nachádza, ak áno tak ju zapíše a vráti ako výsledok.

Môj program dostáva graf s určitým počtom vrcholov a hrán, takýto graf reprezentujem ako dvojrozmerné pole. A výsledok zapíšem ako jednorozmerné pole, do ktorého budem zapisovať cestu kružnice. Pre potreby paralelizmu som musel pridať ešte ďalšie premenné.

Tento algoritmus budem paralelne riešiť s viacerými worker-mi procesora. Každý worker si uchováva svoju cestu grafom spolu s jej dĺžkou. Cestu som nazval *path* a dĺžku *pathLen*, pre každý worker uchovávam aj premennú *hasWork*, v ktorej si bude daný worker pamätať, či práce pracuje na nejakom výpočte.

Pre prvý worker priradzujem do jeho cesty prvý vrchol, zvyšujem dĺžku cesty o 1 a zaznačím mu do pamäte *hasWork* na true. Pre každý worker kontrolujem while cyklom, či už náhodou nenašiel kružnicu a či náhodou ešte nejaký worker nehľadá cestu, túto informáciu si ukladám do premennej *numberOfJobs*, kde pripočítavam počet pracujúcich workerov a premennú *workersWaiting*, kde pripočítavam počet workerov čakajúcich. Následne pre každý worker kontrolujem či pracuje alebo nie. Ak má premennú *hasWork* nastavenú na true, tak jeho úlohou je skontrolovať, či už náhodou svoju cestu nedokončil, v tom prípade zmení svoj stav *hasWork* na false a teraz som potreboval, aby každý worker menil premennú *workersWaiting*. Tu som sa dostal ku problému. Keďže táto premenná je globálna, to znamená, že každý worker môže jej hodnotu meniť a v nejakom momente sa môže stať, že ju bude chcieť zmeniť naraz nejaká dvojica worker-ov. V tomto prípade treba vytvoriť pre semafor. OpenCL má pre takýto synchronizačný mechanizmus vytvorenú skupinu príkazov *atomics*. Ako som už opisoval OpenCL má tieto *atomics* v každej verzii v inej podobe, vo

verzii OpenCL 1.1, ktorú vo svojej práci používam, je možné atomics použiť, lenže keď som ich práve v tomto príklade mal, nefungovali správne, pravdepodobne boli v 1.1 verzii vyvinuté len na ukážku a neboli dotiahnuté do dokonalosti. Atomics som využil pre navyšovanie počtu v premennej *workersWaiting*, grafická karta ale takýto príkaz nedokázala spracovať.

Rozhodol som sa preto pohľadať v histórii a použiť iný systém pomocou zablokovania a odblokovania prístupu, pomocou funkcií, ktoré bránia prístupu worker-a do premennej v prípade, že s ňou iný worker pracuje. V takom prípade iný worker čaká, kým ho funkcia pustí.

V prípade, že kružnica ešte nebola dokončená, hľadá ďalšie vrcholy a kontroluje, či daný vrchol už v kružnici je alebo nie. Každý, ktorý worker nájde si zapíše do cesty navýši dĺžku a prebehne kontrola, ak v kružnici ešte nebol pokračuje ak bol skráti dĺžku a hľadá ďalší vrchol.

Ďalšou možnosťou je, že z posledného vrcholu vedie viacero hrán. V takom prípade som pridal premennú *front*, ktorú tvorí dvojrozmerné pole. Worker doňho vloží cestu, v prípade, že nájde viacero riešení. Jednu možnosť si nechá ostatné zapíše do frontu. K frontu je vytvorená pomocná premenná *emptySlot*, ktorá pre každý worker uchováva informáciu, či vo fronte zanechal nejakú prácu prípadne si niečo z frontu vzal. Tento front potom slúži nato, že ak daný worker nemá prácu, zistí či je vo fronte niečo, na čom by mohol robiť. Ak niečo nájde vezme si to z frontu zaznačí pomocnej premennej *emptySlot*, že si prácu vzal a začne na nej pracovať. Práve týmto spôsobom som vyriešil paralelizmus. Vždy ak nejaký worker nemá čo robiť nazrie do frontu a vezme si odtiaľ úlohu, na ktorej začne pracovať.

Týmto spôsobom zbehne celý while cyklus, kým nenájde jedno riešenie, ktoré potom pošle na CPU a to sa potom vykreslí v canvase na obrazovke Results.

4.3.5 Ďalšie príklady

Vytvoril som aj niekoľko ďalších jednoduchších príkladov, ktoré môžu ostatným užívateľom pomôcť pri pochopení jazyka a samotného paralelizmu. Sú to jednoduché príklady, na ktorých je paralelizmus dostatočne vysvetlený. Ide hlavne o príklady na poliach.

Jedným z príkladov je násobenie alebo sčítavanie polí. Tento príklad je postavený na jednoduchom rozdelení si prvkov poľa medzi workerov a každý sčíta dvojicu prvkov. Ďalšie príklady slúžia na násobenie a sčítavanie matic, tu už príklady rátajú s viacerými procesormi a ich worker-mi.

Niektoré príklady som vytvoril tak, aby slúžili ako cvičenia v prípade testovania alebo vyučovania, sú pripravené ale treba ich doplniť o chýbajúci kód, alebo nájsť v nich chyby.

Tieto príklady sú postavené tak, že ich môžu riešiť aj úplní začiatočníci.

5 Testovanie aplikácie

V tejto kapitole zhrniem výsledky, ktoré som získal počas testovania aplikácie. Testovanie aplikácie prebehlo na Spojenej škole sv. Františka z Assisi v Bratislave. Tohto testovania sa zúčastnili žiaci tretieho ročníka gymnázia.

5.1 Priebeh testovania

Na začiatku testovania som žiakom vysvetlil, načo sa moja práca zameriava. Následne som hotovú aplikáciu žiakom predstavil a nechal im ju vyskúšať. So žiakmi som najprv prešiel časť Lectures, na ktorej som im celý princíp aplikácie a vytvárania programov vysvetlil. Následne som sa zameral na vysvetlenie môjho programovacieho jazyka. Žiakom som jednotlivé príklady zo sekcie Lectures vysvetlil, aby si ich mohli následne sami otestovať v rámci obrazovky Results. Na tejto obrazovke som si u žiakov vyskúšal ich interakciu s aplikáciou.

Pred testovaním aplikácie som pre žiakov okrem hotových príkladov v sekcii Lectures pripravil aj príklady, ktoré som modifikoval do takej podoby, aby ich mohli sami dokončiť, teda som z fungujúcich príkladov odobral niektoré príkazy, ktoré som následne chcel, aby žiaci pridali, tak aby program fungoval.

Obr.18: Testovanie aplikácie

5.2 Výsledky testovania

Na konci testovania som žiakom nechal vyplniť dotazník ako formu spätnej väzby. V tomto dotazníku som sa žiakov pýtal nato, či bola pre nich aplikácia dostatočne prehľadná. Všetci žiaci sa zhodli, že aplikácia pôsobí jasne a vedia sa v nej po chvíli veľmi ľahko orientovať.

Ďalej som sa žiakov pýtal, či sa im zdá aplikácia dostatočne interaktívna. Na túto otázku som dostal dva druhy odpovede, jedným sa interakcia v rámci obrazovky Results páčila, iným sa síce páčila, ale vedeli by si predstaviť aj ďalšie možnosti interakcie. Ako príklad uviedli možnosť klikania na vrcholy v rámci hľadania najkratšej cesty v grafe.

Otázky som zameral aj na môj programovací jazyk, konkrétne som sa pýtal žiakov nato, či bol pre nich jazyk dostatočne zrozumiteľný. Príkazy, ktoré jazyk ponúka sa zdali žiakom dostatočné, nakoľko uviedli, že im okamžite porozumeli.

Na záver som sa pýtal, či by si žiaci vedeli predstaviť použiť takýto jazyk spolu s aplikáciou aj vo vyučovaní. Žiaci zhodnotili, že by si jazyk vedeli predstaviť vo vyučovacom procese, ale záležalo by od oblasti programovania. V rámci strojového učenia, v ktorom sa výpočty na GPU bežne využívajú, si žiaci prácu s jazykom vedeli predstaviť.

V rámci testovania došlo k pár chybám, ktoré som hneď vedel napraviť alebo to boli minimálne chyby, ktoré sa týkali dizajnu. Vďaka týmto výsledkom testovania som zhodnotil, že aplikácia pôsobí na žiakov dostatočne zrozumiteľne a tak sa v rámci prostredia vedeli hneď zorientovať.

6 Záver

Cieľom tejto diplomovej práce bolo v prvom rade analyzovať existujúce technológie a jazyky, ktoré je možné využiť pre programovanie na GPU a tiež prostredia zamerané na vyučovanie programovania. Následne bolo úlohou navrhnúť a implementovať vlastné prostredie pre vyučovanie programovania paralelných algoritmov a zároveň navrhnúť a implementovať aj jazyk pre programovanie týchto algoritmov.

V práci som postupne analyzoval možnosti paralelného programovania, či už vo vyučovaní, ale aj mimo neho. Zároveň som analyzoval aj základné všeobecne známe algoritmy, ktoré je možné zparallelizovať. Získané poznatky som následne aplikoval do návrhu svojej webovej aplikácie. Aplikáciu som navrhol tak, aby pre užívateľa pôsobila interaktívne, no hlavne, aby bola dostatočným materiálom pri vyučovaní. Hlavnou úlohou návrhu bolo vymyslieť jazyk pre programovanie na GPU. V rámci jazyka som sa inšpiroval najmä analýzou existujúcich paralelných jazykov, no využil som aj poznatky z prostredí ako je Imagine Logo a Scratch. Môj programovací jazyk som vytvoril tak, aby bol dostatočne pochopiteľný aj pre začiatočníkov. Jazyk, ktorý som vytvoril dokáže vykonávať základné programovacie úkony, má zadefinované základné dátové štruktúry ako sú polia, grafy, množiny, no hlavne dokáže vykonávať výpočty na GPU. V rámci implementácie som okrem vytvorenia jazyka, vyrobil aj niekoľko konkrétnych príkladov, niektorých známych algoritmov. Konkrétne ide o implementáciu triediaceho algoritmu Select Sort, grafové úlohy ako algoritmus hľadania najkratšej cesty v grafe či hľadanie hamiltonovskej kružnice. A tiež som zparallelizoval aj Conwayovu Game of life. Vytvoril som aj ďalšie jednoduchšie príklady pre začiatočníkov. Všetky spomenuté algoritmy som vyrobil vo svojom paralelnom jazyku.

Nakoľko sa webové aplikácie neustále obnovujú a rovnako chcem, aby obsah aplikácie bol aktuálny je určite možnosť ako aplikáciu ešte vylepšiť. Pri programovaní rôznych ďalších programov môže dôjsť k tomu, že bude potrebné pridať nejakú ďalšiu funkcionálnosť pre môj jazyk. Tiež je možné do budúceho vývoja pridať ďalšie možnosti interakcie na obrazovke Results a ďalšie nápady, ktoré používaním aplikácie môžu vzniknúť. Práve preto som celú aplikáciu zverejnil na GitHubu [12], takže do nej môže ktokoľvek zasiahnuť.

Literatúra

- [1] Cheng, J. *et al.*, *Professional Cuda C Programming*, Indianapolis, USA, John Wiley & Sons, Inc., 2014
- [2] Barlas, G., *Multicore and GPU Programming An Integrated Approach*, Elsevier Inc., 2015
- [3] *C* Programming Guide*, Thinking Machines Corporation, Cambridge, Massachusetts, 1993
- [4] Kaeli, D. *et al.*, *Heterogeneous Computing with OpenCL 2.0*, Eastbourne, UK, Morgan Kaufmann Publishers, 2015
- [5] Sanders, J., Kandrot, E., *Cuda by example*, Boston, USA, Nvidia Corporation, 2011
- [6] Wilt, N., *The Cuda handbook*, Indiana, USA, Pearson Education, Inc, 2013
- [7] *Intel SPMD Program Compiler*, Intel Corporation, Kalifornia, USA [Online] [Dátum: 30.11.2016] <http://ispc.github.io/>
- [8] Tuck, R., *Porta-SIMD: An Optimally Portable SIMD Programming Language*, 1990
- [9] Steele, G., Hillis, D., *ConnectionMachine Lisp: Fine-Gained Parallel Symbolic Processing*, Thinking Machines Corporation, Cambridge, Massachusetts, 1986
- [10] Rakovský, M., *Rozšírenie jazyka Imagine Logo o programovanie GPU* (2016), bakalárska práca, FMFI UK

[11] Černochová, M., Šandová, H., *První ohlédnutí za výukou základů programování ve Scratch na ZŠ aneb Čím nás žáci překvapili i zaskočili, co musíme příště dělat jinak*, [Online][Datum:2015]

http://didinfo.umb.sk/public/filestore/documents/richtext/181/didinfo_2015.pdf

[12] Vyukova aplikacia na programovanie GPU [Online][Datum:2.5.2017]

<https://github.com/deno287/Vyukova-aplikacia-na-programovanie-GPU>

Prílohy

- CD so zdrojovými súbormi