UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Rozšírenie jazyka Imagine Logo o programovanie GPU

Bakalárska práca

2016 Michal Rakovský

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Rozšírenie jazyka Imagine Logo o programovanie GPU

Bakalárska práca

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

2016 Michal Rakovský

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Michal Rakovský
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.9. aplikovaná informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Rozšírenie jazyka Imagine Logo o programovanie GPU
Extending Imagine Logo with GPU Programming

Cieľ: Známe detské programovacie prostredie Imagine Logo založené na programovacom jazyku Logo je rozširovateľné o doplnky pomocou viacerých technológií - napríklad komunikáciou s iným softvérom cez sockety. Úlohou je preskúmať programovanie paralelných grafických procesorov moderných grafických kariet (tzv. GPU) a sprístupniť ho pre deti vytvorením knižnice pre Imagine Logo vrátane zaujímavých ukážok a príkladov. Študent navrhne vhodnú sadu príkazov, ktoré umožnia riadenie paralelných procesorov priamo z jazyka Logo.

Literatúra: Petrovič, P. (2007) Program Your NXT Robot with Imagine, Eurologo 2007.

Kľúčové slová: gpu, programming for children, imagine logo

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

Dátum zadania: 27.10.2015

Dátum schválenia: 28.10.2015

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

Pavel Petrovič
.....
vedúci práce

Čestne prehlasujem, že som túto prácu vypracoval samostatne,
s použitím literatúry a zdrojov uvedených v závere práce

_____

Michal Rakovský

Poďakovanie

Chcem sa poďakovať svojmu školiteľovi Mgr. Pavlovi Petrovičovi, PhD. za rady, pomoc, konzultácie a čas, ktorý mi venoval.

Abstrakt

Obrovské množstvá dát hromadiace sa v dátových skladoch, stále sa zvyšujúce nároky na spracovanie multimédií, vedecko-technické výpočty, umelá inteligencia, predikčné, finančné a iné modely dynamických systémov – to všetko v dnešnej dobe vyžaduje obrovský výpočtový výkon. V priebehu najbližších desaťročí sa jeho potreba určite nezníži, naopak, ešte podstatne vzrastie. Historická epocha sekvenčných architektúr a sekvenčného spôsobu programovania končí vo väčšine sfér. Spôsob programovania bude odlišný, do popredia by sa mali dostať implicitne paralelizovatelné programy. Preto by bolo nezodpovedné nepripravovať na tento trend súčasnú mladú generáciu.

Jedným z bežne používaných programovacích jazykov na základných i stredných školách je Imagine Logo, ktoré už je paralelné, avšak len na úrovni ľahkých softvérových vlákien, ktoré sa spracovávajú na jednom jadre.

V tejto práci sme vytvorili rozšírenie jazyka Imagine Logo o možnosť spúšťať logovské programy na paralelnom grafickom koprocesore (GPU), ktorý je dnes súčasťou väčšiny PC pre urýchlenie grafického výkonu. Logovské programy, ktoré spĺňajú stanovené obmedzenia sa pomocou nami vytvoreného API automaticky presunú do démona, ktorý s Imagine komunikuje cez TCP/IP sockety, tam sa skompilujú na kód vykonateľný pomocou technológie OpenCL, spustia a výsledok výpočtu sa automaticky prenesie naspäť do Imagine Loga. Navrhnutú a implementovanú technológiu demonštrujeme na niekoľkých príkladoch – spracovania zoznamov a obrázkov.

Kľúčové slová: gpu, programovanie pre deti, imagine logo

Abstract

Huge amount of data accumulating in data warehouses, increasing demands on multimedia processing, scientific and technical computing, artificial intelligence, predictive, financial models of dynamic systems - all of it demands very big computing power. Certainly, this demands will not decrease during the coming decades and it is very likely that they will increase. Epoch of sequential architectures and sequential programming is ending in most areas. Programming style will be different, to the fore should be brought implicitly parellelizable programs. Therefore, it would be irresponsible, if we would not prepare young generation for this trend.

One of common programming languages used on primary and high schools is Imagine logo, which is parallel, but only at the level of light software threads, which are processed by one core.

In this thesis we have developed extension of Imagine Logo. We have implemented feature of executing programs on parallel graphic coprocessor (GPU), which is part of most PC today. Logo programs, which fulfill given restrictions are, using developed API, transferred to the daemon, which communicates with Imagine through TCP/IP sockets. Then they are compiled using OpenCL, they are run and result of computation is transferred back to Imagine Logo. We demonstrate designed and implemented technology on the examples - lists and image processing.

Key words: GPU, programming for children, Imagine logo, parallel programming

OBSAH

Úvod

Rýchlosť, akou je počítač schopný pracovať nie je nekonečná, ale je obmedzená rýchlosťou svetla. Sekvenčný výpočet môže byť zrýchlený zvýšením rýchlosti procesora, ktorú nie je možné neustále zvyšovať a ľudstvo sa blíži k dosiahnutiu limity výpočtového výkonu jednej výpočtovej jednotky. Riešením tohto problému je paralelné programovanie, teda rozdelenie výpočtu medzi viacero výpočtových jednotiek, ktoré ho dokážu synchronizovane spracovať. Myšlienka paralelného programovania spočíva v tom, že čím viac výpočtových jednotiek je zapojených do výpočtu, tým menší čas je potrebný k dosiahnutiu výsledku za predpokladu dostatočnej rýchlosti výpočtových jednotiek a efektívnemu spracovaniu dát.

1.1 Ciele práce

Cieľom práce je preskúmať možnosti paralelného programovania na grafických procesoroch. Po preskúmaní možností paralelného programovania je cieľom aplikovať tieto poznatky na tvorbu rozšírenia jazyka Imagine Logo, ktoré tento jazyk obohatí o rôzne funkcie, ktoré budú rýchlejšie ako sekvenčné výpočty. Výsledkom bude aplikáciu modifikovať do podoby virtuálneho stroja, ktorý bude jazyk Imagine Logo rozširovať o možnosť tvorby vlastných paralelných funkcií, ako je tomu v bežných programovacích jazykoch s využitím štandartu umožňujúceho paralelné programovanie. Keďže sa na mnohých stredných školách používa Imagine Logo, toto rozšírenie umožní študentom so záujmom o programovanie získať prehľad o efektívnosti paralelného programovania. Keďže sa dá predpokladať, že postupom času už sekvenčné programovanie bude minulosťou vzhľadom na fyzikálne limity výpočtového výkonu, je nutné aby čoraz viac programátorov, a hlavne tých budúcich zvládalo paralelný prístup. Cieľom tejto práce aj vzbudiť záujem mladých programátorov o paralelné programovanie prostredníctvom rozšírenia v deckom programovacom jazyku Imagine Logo, ktoré sa používa na stredných školách.

1.2 Prehľad o obsahu dokumentu

V nasledujúcich kapitolách tohto dokumentu sú popísané niektoré východiská potrebné pri vytváraní vlastnej implementácie programu a návrh softvérovej implementácie. Vo východiskovej kapitole sú popísané zdroje a potrebné technológie k tejto práci. V kapitole návrhu sa budeme venovať popisu návrhu riešenia a v kapitole implementácie si popíšeme všetky súčasti rozšírenia.

2 Východiská

2.1 Prehľad teórie

2.1.1 Imagine Logo

Logo je jednoduchým funkcionálnym programovacím jazykom, ktorý bol navrhnutý pôvodne pre výuku myslenia, no spája sa hlavne s výukou programovania pre deti. Hlavnou súčasťou jazyka Logo je korytnačka (turtle), ktorá umožňuje programovanie korytnačej grafiky. Imagine Logo je verzia jazyka Logo, používaná na aj slovenských aj na českých školách. Je nepriamym nástupcom Comenius Loga, kompletne objektový jazyk riadený udalosťami. Podporuje paralelné programovanie a obsahuje mnoho prvkov typických pre programy pre Windows.

Charakteristika jazyka Imagine Logo

- príkazy sú podobné prirodzenému jazyku
- prevedenie abstraktných pojmov na konkrétne pojmy – akýkoľvek príkaz sa hneď prejaví
- široko použiteľný – Logo je jednoduchý jazyk, ale dostatočne silný aj na zložitejšie úlohy

Niektoré triedy jazyka Logo

- Turtle – korytnačka predstavuje jeden z hlavných objektových tried Imagine Loga. Je to objekt, ktorý má bežné atribúty, procedúry a udalosti. Táto trieda umožňuje programovanie korytnačej grafiky v Imagine Logu
- Net – umožňuje vytvárať projekty, ktoré komunikujú po sieti tak, že niekoľko programov v Imagine Logu bežiacich na rozličných počítačoch prepojených sieťou TCP/IP dokáže navzájom komunikovať. Takto je možné implementovať napríklad rôzne distribuované systémy, hry a aplikácie s architektúrou client/server a podobne. Objekt sa vytvára príkazom: new "Net [… settings …], kde settings sú argumenty potrebné pre spojenie s iným procesom.
  Prehľad Net argumentov:

- name – meno Net objektu, ak sa nezadá Imagine si vygeneruje svoje meno
- nickName – unikátne meno používateľa, zabezpečuje identifikáciu procesu
- port – defaultná hodnota Imagine procesov je 51
- server – hodnotou je meno servera, alebo IP adresa. Tento argument sa vypĺňa iba na klientoch, servery ignorujú toto nastavenie.
- style – hodnotou je „server“ alebo „client“, tu sa nastavuje či má byť proces hostom alebo klientom

V tejto triede vývojari Imaginu naprogramovali metódy, ktoré zabezpečujú odosielanie a prijímanie rôznych typov dát a pripájanie procesov navzájom:

- connect – Net objekt aktivuje spojenie so serverom
- connected? – metóda vracia true / false, zisťuje stav pripojenia na server
- message – metóda vráti prijatý objekt, ktorý môže byť akýkoľvek Imagine typ
- send – metóda má 2 argumenty, prvý je nickName procesu, ktorému odosiela dáta a druhý argument sú dáta, ktoré môžu byť tiež akýkoľvek Imagine objekt

Trieda Net taktiež obsahuje eventy, ktoré sú užitočné pre komunikáciu, ako sú:

- onReceive – po prijatí správy vykoná zoznam príkazov zadaných v tomto evente
- onReceiveObject – pro prijatí objektu vykoná zoznam príkazov zadaných v tomto evente
- onStatusChanged – po zmenení status connected / disconnected sa vykoná zoznam príkazov v tomto evente

2.1.2 Paralelné programovanie

Paralelné programovanie je označenie konceptu, ktorý umožňuje naprogramovať úlohy tak, aby sa počas behu mohlo vykonávať viac činností súčasne. Implementácia paralelného programovania môže byť v podobe knižníc pre tradičné programovacie jazyky, vo forme rozšírenia programovacieho jazyka, alebo vytvorením programovacieho jazyka s podporou paralelizmu. Pri sekvenčnom programovaní sa inštrukcie spracovávajú postupne na jednej výpočtovej jednotke, pričom pri paralelnom programovaní sa jeden výpočtový problém

rozdelí na viacero výpočtových jednotiek, ktoré sú schopné synchronizovaného rozdelenia úlohy a súčasného riešenia daného problému.

V prostredí paralelného programovania je veľmi dôležitá synchronizácia výpočtov. V prípade voľne viazaných výpočtových jednotiek je nutné použiť na synchronizáciu zasielanie správ. V prípade tesne viazaných výpočtových jednotiek systém často podporuje atomické operácie na zdieľanej pamäti. Nevhodné použitie synchronizácie môže viesť k deadlockom, čo sú programátorské chyby.

Paralelné programovanie rieši problémy ako sú napríklad šetrenie nákladov, rýchlosť výpočtu, nemožnosť riešenia problémov na jedinom počítači, pamäťová limitácia a iné.

Využitie paralelného programovania

- Maximalizácia výpočtového výkonu – High Performance Computing (HPC), aplikované pri zložitých modelovaniach alebo simuláciách.
- Bežné výpočty:

Spracovanie rozsiahlych dát – obraz, zvuk

Náročný numerický výpočet – šifrovanie

Klasifikácia paralelného programovania

- Interakcia procesorov

Zaoberá sa komunikáciou medzi dvoma procesormi, buď pomocou zdieľanej pamäte, alebo metódou odosielania správ (message passing)

- Zdieľaná pamäť

V modely so zdieľanou pamäťou paralelné procesy zdieľajú globálny adresný priestor, pričom z neho asynchrónne čítajú a zapisujú. Tento prístup potrebuje bezpečnostné mechanizmy, ako napríklad zamykanie pamäti

- Message passing

V tomto modely si paralelné procesy vymieňajú dáta pomocou posielania správ medzi sebou. Táto komunikácia môže byť synchrónna aj asynchrónna

- Problém dekompozície

Každý paralelný program je zostavený zo súčasne vykonávajúcich sa procesov. Problém dekompozície spočíva v spôsobe definovania procesov

2.1.3 Grafické procesory

Grafický procesor je špecializovaný procesor umiestnený na základnej doske, alebo grafickej karte. Zabezpečuje výpočty potrebné pre vykresľovanie dát z operačnej pamäte na zobrazovacie zariadenie. Grafické procesory sú v špecifických výpočtoch často efektívnejšie ako CPU. V súčasnosti sa moderné grafické procesory používajú aj na iné účely ako výpočty nutné pre vizualizáciu dát, napríklad pre účely paralelných výpočtov, ktoré zvládajú veľmi efektívne, hlavne vďaka veľkému počtu stream procesorov, rýchlej pamäti a frekvencii, vďaka čomu má mnohokrát väčší výkon ako CPU. Paralelný charakter grafických procesorov je známy od čias GeForce2.

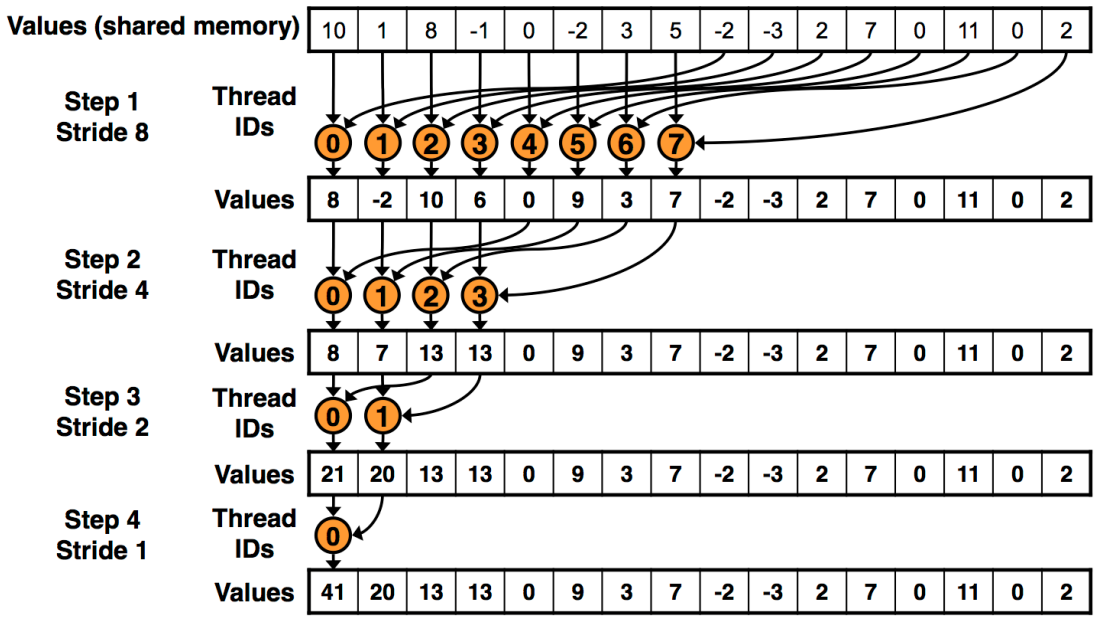
Integrované grafické karty zdieľajú časť systémovej pamäte RAM viac ako dedikované grafické karty. Môžu byť integrované v základnej doske ako súčasť chipsetu, alebo súčasťou procesora (AMD APU, Intel HD Graphics). Integrované grafické čipy sú viazané na systémovú pamäť o ktorú sa musia deliť s procesorom CPU. Prenos dát je oproti dedikovaným grafickým kartám pomalá, dokáže dosiahnuť rýchlosť prenosu dát 29.856 GB/s.

Dedikované grafické karty využívajú vlastnú pamäť, prípadne môžu využiť aj systémovú RAM. V počítači teda robia úlohu samostatného hardvérového riešenia pre účely výpočtov s grafickými dátami. Rýchlosť prenosu dát medzi ich vlastnou pamäťou a procesormi GPU dosahuje až 264 GB/s.

Obrázok 1:Schéma dedikovanej grafickej karty [5]

2.1.4 Paralelné výpočty na GPU

Pre mnohé hardvérové architektúry sa paralelizmus stáva jediným postupom k dosiahnutiu vyššieho výpočtového výkonu. Naproti prekážkam, ako sú napríklad teplotné obmedzenia sa rôzne platformy nasmerovali k nárastu výkonu pomocou zvyšovania počtu výpočtových jednotiek, ktoré umožňujú paralelné spracovanie dát. Dosiahli tým rýchly nárast výkonu a schopnosti aj u najmenších mobilných zariadeniach. Potreba relatívne ľahkého, efektívneho a jednotného zaobchádzania s veľkým množstvom platforiem spôsobila vysoký dopyt po softvérových riešeniach. Súčasné softwarové riešenia, ktoré sú určené pre paralelné výpočty obsahujú často závažné nedostatky. Mnoho riešení je viazaných na konkrétny hardware, napríklad CUDA, alebo sú viazané na software, napríklad DirectCompute. Ďalšie nástroje umožňujú istý stupeň prenositeľnosti na rôzne platformy, avšak nie sú úplne špecializované na daný hardware, kvôli čomu dochádza k pomalšiemu behu aplikácií na nich postavených. Tento stav má za úlohu riešiť priemyslový štandard OpenCL, ktorého prvý návrh uskutočnila firma Apple a dodnes vlastní práva k názvu OpenCL.



Obrázok 2:Paralelný výpočet na dátovej štruktúre opisuje fázy redukcie inštancií kernelu [3]

2.2 Prehľad existujúcich systémov

2.2.1 NXT Logo

NXT Logo je programovací jazyk pre roboty LEGO Mindstorms NXT. Je určený pre deti pre účely výuky programovania s možnosťou grafického výstupu na obrazovku s kombinovaním existujúceho Logo programovacieho jazyka.

Rozšírenie povoľuje používateľom Imagine Loga ovládať NXT robotov z ich vlastných projektov.

2.3 Prehľad technológie

2.3.1 OpenCL

OpenCL je štandard pre paralelné programovanie heterogénnych systémov, vybavených GPU, APU, prípadne DSP podporujúcich paralelné programovanie. Prvá implementácia OpenCL bola navrhnutá vo firme Apple, v súčasnosti sa na vývoji podieľa priemyslové konsorcium Khronos. Za účelom vývoja OpenCL Khronos vytvoril pracovnú skupinu Khronos Compute Working Group, ktorá v krátkom období prepracovala pôvodný návrh OpenCL do konečnej podoby a vydali OpenCL 1.0. Hlavným cieľom OpenCL je dosiahnuť rozšíriteľnosť rovnakých riešení na viaceré platformy, bez potreby písania nového kódu špecifického pre dané zariadenie. Toto riešenie umožňuje síce prenositeľnosť, no výpočtový

výkon nie je pre každé zariadenie rovnaký. rozdielu výkonu medzi zariadeniami sa postupom času optimalizuje vývojármi štandardu OpenCL a jednotlivými výrobcami grafických čipov.

Závislosti platformy OpenCL od technológie

- OpenCL definuje abstraktné hardvérové zariadenie a k nemu ovládacie softwarové rozhranie, pomocou ktorého aplikácie pristupujú ku konkrétnym výpočtovým možnostiam rôznych hardvérových platforiem. Tieto platformy zahŕňajú klasické procesory CPU s podporou inštrukcií SSE3, grafické procesory nVidia GeForce od rady 8xxx, ATI Radeon od rady 4xxx, mobilné čipy, procesory typu Cell, signálové procesory DSP a ďalšie, teda OpenCL štandard nie je závislý od hardvéru.
- Softvérové rozhranie štandardu OpenCL nie je závislé ani na softvérovej platforme, teda nepotrebuje žiadny konkrétny operačný systém. Hlavný výrobcovia grafických čipov v nich zahrnuli implementáciu OpenCL do ovládačov, dostupné pre širokú škálu operačných systémov, ako sú distribúcie Windowsu (od XP), či hlavné distribúcie Linuxu, operačný systém MaxOS, alebo virtuálnych strojoch (Vmware)

Hlavné súčasti OpenCL

- Abstraktné modely – určujú požadované vlastnosti a správanie zariadenia OpenCL
- OpenCL Framework – súčasťou frameworku je definícia OpenCL API
- Špecifikácia programovacieho jazyka (OpenCL C) – programovací jazyk je využívaný v zariadeniach OpenCL

2.3.2 Abstraktné modely OpenCL

Model Platformy:

Definuje heterogénny paralelný stroj ako počítačový systém schopný poskytovať služby štandardu OpenCL. Tento stroj obsahuje hostiteľský systém a jedno, alebo viac zariadení OpenCL, ktorými hostiteľský systém disponuje. Tento model predpokladá, že sa zariadenie skladá z výpočtových jednotiek, ktoré sú delené do procesných elementov

Exekučný model:

Beh systému využívajúceho OpenCl prebieha na dvoch úrovniach heterogénneho paralelného stroja. Klasická aplikačná časť, nazývaná aplikácia, je vykonávaná v rámci hostiteľského systému. Aplikácia zodpovedá predovšetkým za komunikáciu medzi hostiteľským systémom a zariadeniami, taktiež za spustenie a koordináciu výpočtov v týchto

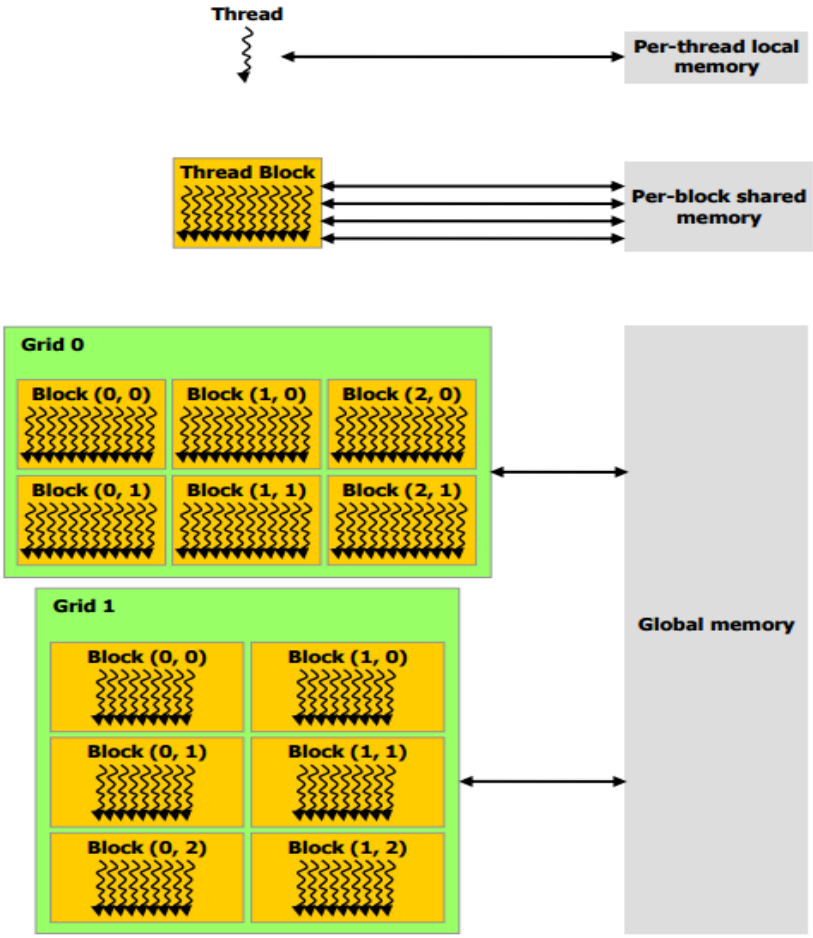
zariadeniach. Jednotlivé zariadenia potom spracovávajú iba tú časť aplikácie, ktorá bola vyjadrená jazykom OpenCL C, nazývaná program. Program má podobu jedného, alebo viacerých výpočtových vláken, ktoré sú spracovávané v rámci procesných elementov daného zariadenia. Výpočtové vlákna sú inštanciami funkčného objektu, ktorý sa nazýva kernel. Pri spustení výpočtu aplikácia špecifikuje jedno až trojrozmerný indexový priestor, celkový počet inštancií kernelu (cl::NDRange) a taktiež veľkosť skupín do ktorých sa vlákna združujú. Na základe týchto informácií OpenCL určí počet skupín a každému vláknu kernelu priradí globálny index a skupinový index. Lokálny, resp. globálny index vlákna identifikuje vlákno kernelu v rámci skupiny, resp. v rámci všetkých vláken. Skupinový index je identifikátor skupiny, do ktorej vlákno kernelu patrí

Kontexty a príkazové fronty:

V exekučnom modely si aplikácie vytvárajú kontexty, v ktorých sú zahrnuté informácie o zariadeniach (typ zariadenia, typ platformy), množinách kernelu (cl::NDRange) a programoch OpenCL v ktorom sú kernely uložené, pamäťové objekty, ktoré kernely budú spracovávať (cl::Buffer), prípadne ďalšie údaje potrebné pre výpočet. Na ovládanie zariadení, ich synchronizácie, presun dát a spustení kernelu slúžia príkazové rady (cl::CommandQueue). Tieto príkazové rady spracovávajú príkazy v poradí, v akom idú za sebou, alebo nezávisle od poradia. Každý príkaz pri zaradenú do príkazového radu generuje udalosť. Väčšinu príkazov je možné vykonávať asynchrónne a vďaka tomu je možné udalosti využívať na sledovanie stavu príkazov a ich vzájomnú synchronizáciu.

Pamäťový model:

Pamäťový model definuje pamäťovú hierarchiu zariadenia, spôsoby akými sú dáta v zariadení ukladané, spôsoby komunikácie medzi zariadeniami a hostiteľským systémom. Špecifikuje typy pamäti, druhy prístupu k pamäti a ich alokáciu v pamäťových moduloch



Obrázok 3:Schéma prístupu výpočtových jednotiek do typov pamäte grafickej karty [3]

Typy pamäte modelu:

- Globálna pamäť – oblasť pamäti viditeľná všetkým vláknam kernelu. Môže byť cachovaná v OpenCL zariadení pre zvýšenie výkonu a efektivite, alebo môže byť priamo v DRAM. Je plne prístupná hostiteľskému systému.
- Konštantná pamäť – oblasť globálnej pamäti, do ktorej vlákna kernelu nemajú právo zápisu
- Lokálna pamäť – oblasť pamäte viditeľná len vláknam kernelu v skupine (Grid)

- Privátna pamäť – oblasť pamäte viditeľná len v rámci jedného vlákna kernelu (vlastná pamäť jedného vlákna)

Spôsoby alokácie pamäti v jednotlivých pamäťových oblastiach:

	Globálna	***Konštantná***	***Lokálna***	***Privátna***
Aplikácia	dynamická	dynamická	dynamická	žiadna
Kernel	žiadna	statická	statická	statická

Druhy prístupu k pamäťovým oblastiam:

	Globálna	***Konštantná***	***Lokálna***	***Privátna***
Aplikácia	čítanie/zápis	čítanie/zápis	žiadny	žiadny
Kernel	čítanie/zápis	čítanie	čítanie/zápis	čítanie/zápis

Programovací model:

Podľa exekučného modelu OpenCL podporuje úlohovo paralelné a dátovo paralelné modely. Štandard OpenCL sa sústreďuje predovšetkým na dátovo paralelný programovací model. Tento programovací model definuje výpočet ako súbeh inštancií kernelu spracovávajúcich dátové zložky vstupnej dátovej štruktúry. V rámci indexového priestoru exekučného modelu sú typy inštancie a spôsoby ich mapovania na dátové zložky definované jednoznačne. V najjednoduchšom prípade pripadá jedna inštancia kernelu na jednu dátovú zložku. Úlohovo paralelný programovací model umožňuje súbežne spúšťať niekoľko inštancií rôznych kernelov. V tomto modeli nie je možné vyžadovať tesný súbeh niekoľko inštancií rovnakého kernelu, čo je hlavným rozdielom oproti dátovo paralelnému programovaciemu modelu, kde sa v rovnakom čase spúšťa veľký počet inštancií jedného kernelu, ktoré spolu dokážu komunikovať.

2.3.3 OpenCL Framework

- Poskytuje aplikáciám možnosť využívať hostiteľský systém a jeho zariadenia v súlade s modelmi OpenCL ako heterogénny paralelný stroj.
- Súčasti OpenCL frameworku:
  - Programovacie aplikačné rozhranie OpenCL API, umožňujúce prácu so systémom OpenCL

- Kompilátor jazyka OpenCL, ktorý prekladá programy písané v tomto jazyku do konkrétneho strojového kódu danej hardvérovej platformy

Jazyk OpenCL C

- OpenCL C je založený na norme C99 jazyka C. Poskytuje vstavanú work – group bariérovú funkcionalitu. Táto funkcionalita môže byť použitá pri spúšťaní kernelu v zariadení pre synchronizáciu work - group, ktoré vykonávajú kernel. Všetky pracovné jednotky work – groupy musia vykonať konštrukciu bariéry predtým ako nejakej z nich bude povolené pokračovať vo vykonávaní kernelu za bariérou.
- Rozšírenia OpenCL C:
  - Vektorové dátové typy
  - Kvalifikátory adresného priestoru
  - Kvalifikátory prístupových práv
  - Kernelové funkcie
  - Presné definície správania dátových typov čísel s pohyblivou desatinnou čiarkou podľa štandardu IEEE 754, ktoré platia pre všetky funkcie a operátory nad týmito typmi
- Obmedzenia OpenCL C:
  - Smerníky na funkcie, pole dynamickej dĺžky a bitové polia sú zakázané
  - Väčšina hlavičkových súborov štandardnej knižnice jazyka C je nedostupná
  - Rekurzívne funkcie nie sú povolené
  - Kernelové funkcie nesmú deklarovať argumenty typu smerník na funkciu, ani nič vracať
  - Zápisy na pole číselných typom menších ako 32 bitov sú zakázané

2.3.4 Triedy OpenCL

Buffer:

Buffer je pamäťový objekt ktorý uchováva lineárny zoznam bytov. Buffer objekty sú zvyčajne dostupné cez smerník v kernely vykonávanom v zariadení. K týmto objektom sa môže pristupovať cez OpenCL API na strane hosta.

Buffer Objekt zahŕňa tieto informácie:

- veľkosť v bytoch
- vlastnosti, ktoré opisujú použitie informácie a v akej oblasti sú dáta alokované
- buffer dáta

Command:

OpenCL operácie, ktoré sú predložené príkazovému radu na vykonanie. OpenCL commandy posúvajú kernely na vykonanie výpočtovému zariadeniu, pracujú s pamäťovými objektami a podobne.

Command-queue:

Príkazový rad OpenCL, uchováva v sebe príkazy (cl::Command), ktoré budú vykonané na určenom zariadení. Príkazový rad je vytvorený na špecifickom zariadení v contexte (cl::Context). Príkazy v rade sú usporiadané v poradí, ale môžu byť vykonávané aj mimo tohto poradia, v závislosti od toho, či je nastavené vykonávania v poradí (In-order Execution), alebo vykonávanie mimo poradia (Our-of-order Execution)

Context:

Je prostredie, v ktorom sa spúšťajú kernely a je dómenou, kde je definovaná synchronizácia a správa pamäte. Context zahŕňa sadu zariadení, pamäťovú dostupnosť k týmto zariadeniam, zodpovedajúce pamäťové vlastnosti a jeden, alebo viacero príkazových radov, použitých na záznam vykonávaných kernelov alebo operácií na pamäťových objektoch.

Device:

Trieda obsahuje informácie o zariadení. Táto trieda typicky prislúcha GPU, viac jadrovému CPU, a iným procesorov ako sú DSPs a Cell/B.E procesory.

Image:

Je pamäťový objekt, ktorý ukladá dvoj- , alebo troj – rozmerné štruktúrované polia. Obrazové dáta sú dostupné pre čítanie aj zápis. Čítacie funkcie požívajú sampler (cl::Sampler)

Image objekt zahŕňa tieto informácie:

- dimenziu obrazu

- popis každého elementu v obraze
- vlastnosti, ktoré opisujú použitie informácie a v akej oblasti sú dáta alokované
- dáta obrazu

Platform:

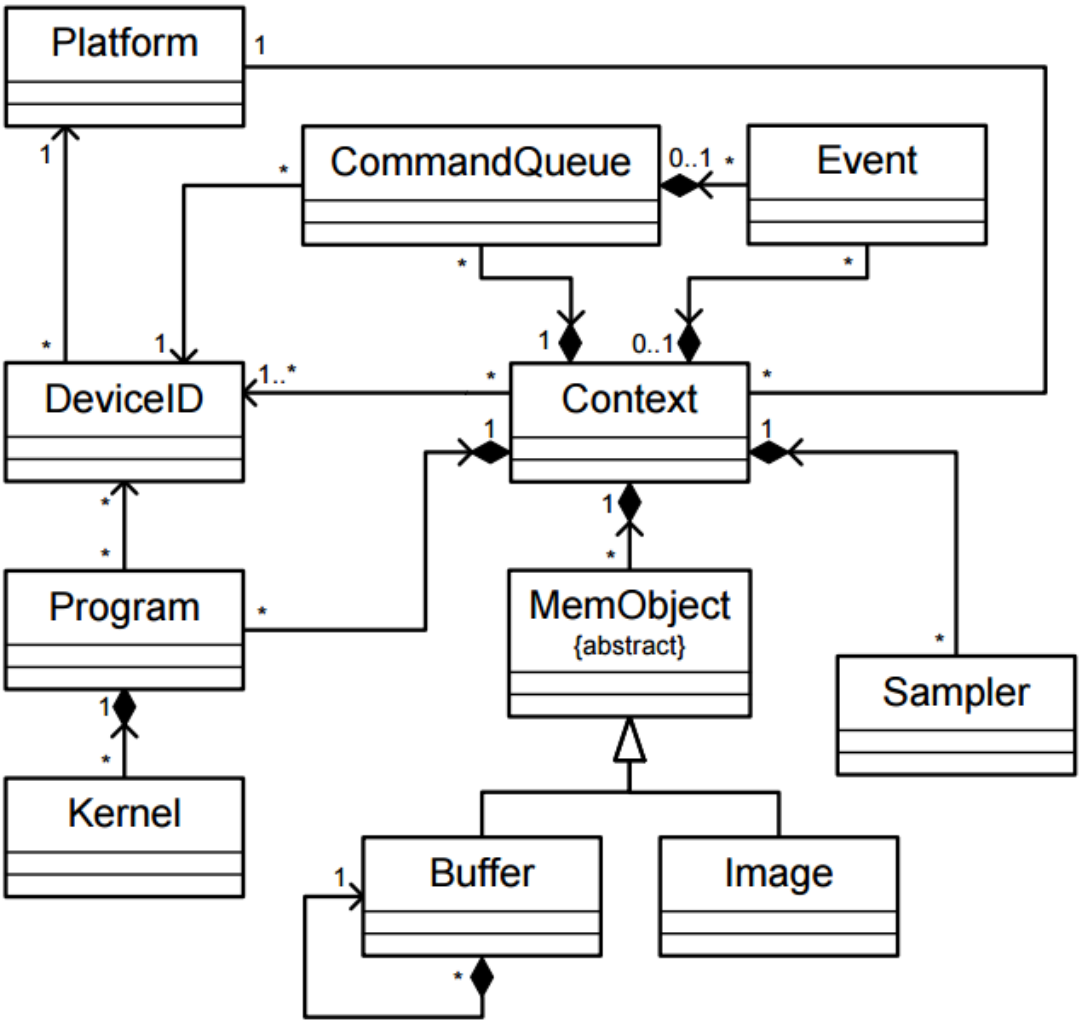
Je hostiteľský súbor zariadení, ktoré v rámci OpenCL frameworku umožňujú aplikácii zdieľanie zdrojov a spúšťanie kernelov na zariadeniach v danej platforme.

Program:

OpenCL program obsahuje sadu kernelov. Program môže tiež zahŕňať jeden alebo viac procesných elementov.

Program objekt zahŕňa v sebe tieto informácie:

- referenciu na pridružený cl::Context
- programové zdroje, alebo binárne dáta
- posledný úspešný vykonaný program, zoznam zariadení pre ktoré bolo vykonávanie programu zostavené, build nastavenia ktoré boli použité a build log
- Počet kernelových objektov, ktoré sú práve pripnuté



Obrázok 4:OpenCL UML class diagram [4]

2.3.5 Ukážka tvorby OpenCL aplikácie a kernelových OpenCL C funkcii

Kód v C++ na strane hosta:

```
void paraSum() {
     int len = 100000000;
     int result;
     std::vector<int> A(len,1);

     std::vector<cl::Platform> all_platforms;
     cl::Platform::get(&all_platforms);
     if (all_platforms.size() == 0) {
            std::cout << " No platforms found. Check OpenCL installation!\n";
            exit(1);
     }
     cl::Platform default_platform = all_platforms[1];
     std::cout << "Using platform: "
        << default_platform.getInfo<CL_PLATFORM_NAME>() << "\n";

     std::vector<cl::Device> all_devices;
```

```
        default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
        std::cout << "Found " << all_devices.size() << " different devices."
           <<      std::endl;
        for (int i = 0; i < all_devices.size(); i++) {
               std::cout << i << ": " << all_devices[i].getInfo<CL_DEVICE_NAME>()
                 << std::endl;
        }
        if (all_devices.size() == 0) {
               std::cout << " No devices found. Check OpenCL installation!\n";
               exit(1);
        }
        cl::Device default_device = all_devices[0];
        std::cout << "Using device: " << default_device.getInfo<CL_DEVICE_NAME>()
          << "\n";

        cl::Context context({ default_device });
        cl::Program::Sources sources;

        std::ifstream t("kernels.cl");
        std::string kernel_code((std::istreambuf_iterator<char>(t)),
        std::istreambuf_iterator<char>());

        sources.push_back({ kernel_code.c_str(),kernel_code.length() });

        cl::Program program(context, sources);
        if (program.build({ default_device }) != CL_SUCCESS) {
               std::cout << " Error building: " <<
        program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(default_device) << "\n";
               exit(1);
               std::cin.get();
        }

        cl::Buffer buffer_A(context, CL_MEM_READ_WRITE, sizeof(int) * len);
        cl::Buffer buffer_B(context, CL_MEM_READ_WRITE, sizeof(int) * len);
        cl::Buffer buffer_R(context, CL_MEM_READ_WRITE, sizeof(int));
        cl::Buffer buffer_L(context, CL_MEM_READ_WRITE, sizeof(int));

        cl::CommandQueue queue(context, default_device);

        queue.enqueueWriteBuffer(buffer_A, CL_TRUE, 0, sizeof(int) * len, A.data());
        queue.enqueueWriteBuffer(buffer_L, CL_TRUE, 0, sizeof(int), &len);

        cl::Kernel kernel_add = cl::Kernel(program, "par_sum");
        kernel_add.setArg(0, buffer_A);
        kernel_add.setArg(1, buffer_B);
        kernel_add.setArg(2, buffer_L);
        kernel_add.setArg(3, buffer_R);

        queue.enqueueNDRangeKernel(kernel_add, cl::NullRange,
          cl::NDRange(500), cl::NullRange);
        queue.enqueueReadBuffer(buffer_R, CL_TRUE, 0, sizeof(int), &result);
}
```

- Musí byť známa platforma a zariadenie:

  Zistenie platformy používanej platformy a zariadenia je dôležité, pretože OpenCL framework prekladá OpenCL C jazyk do inštrukcií konkrétnej platformy.

  Príkazom `cl::Platform::get(&all_platforms)` sa uložia dostupné platformy

podporujúce OpenCL do vektora `std::vector<cl::Platform> all_platforms` V indexe 0 sa nachádza defaultná platforma nastavená systémom, preto jej získanie je jednoduché `cl::Platform default_platform = all_platforms[0]`. Následne je potrebné, aby framework OpenCL poznal zariadenie, na ktorom bude prebiehať výpočet.

Príkazom `default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices)` sa načítajú všetky dostupné zariadenia, ktoré podporujú OpenCL do vektora `std::vector<cl::Device> all_devices`. Rovnako ako pri platforme, aj defaultné nastavené zariadenie sa nachádza v indexe 0 a je ho možné získať `cl::Device default_device = all_devices[0]`

- Ak už je známe zariadenie, je potrebné vytvoriť context. OpenCl context je vytvorený jedným, alebo viacerými zariadeniami. Je používaný za behu OpenCL a slúži na spravovanie objektov, ako sú príkazové rady, pamäť, program a kernelové objekty pre vykonávanie kernelov na jednom, alebo viacerých zariadeniach špecifikovaných v contexte.

  `cl::Context context({ default_device })` deklaruje premennú context, ako odkaz na zariadenia a platformy
- Následne treba vytvoriť program pomocou `cl::Program program(context, sources)`, ktorý obsahuje informácie o contexte a zdrojoch `cl::Program::Sources sources` , ktoré ukladajú zdrojové súbory písané v jazyku OpenCL C v podobe stringu. Trieda cl::Program::Sources má podobu vectora, do ktorého je možné uložiť viacero funkcií, alebo celých zdrojových súborov písaných v jazyku OpenCL C `sources.push_back({ kernel_code.c_str(),kernel_code.length() })`, kde `kernel_code` je string obsahujúci zdrojový kód jazyka OpenCL C. Tie treba skompilovať metódou `program.build({ default_device })`, ktorá v prípade úspešného zostavenia programu v exekučnom modely vráti hodnotu `CL_SUCCESS`
- Alokovanie pamäte pre objekty, s ktorými bude OpenCL pracovať. OpenCL API obsahuje triedu cl::Buffer, ktorá v kontexte vyhradí pamäť pre všetky potrebné vstupné a výstupné hodnoty `cl::Buffer buffer(context, CL_MEM_READ_WRITE, sizeof(type) * len)`, Prvý atribút je cieľ bufferu, teda context. Druhým je určenie typu prístupu do tejto pamäte, tretím je veľkosť potrebná na alokovanie. Konštanta „len“ v tomto prípade určuje, že ide o štruktúru poľa. Triedu `cl::Buffer` je možné používať aj pre vytvorenie pomocných štruktúr pre ukladanie priebežných výpočtov

inštancií kernelov v podobe prístupu ku globálnej pamäti

- Vytvorenie príkazového radu `cl::CommandQueue`, to ktorého sa ukladajú príkazy pre zariadenie `cl::CommandQueue queue(context, default_device)`. Po vytvorení radu je nutné zapísať vstupné objekty do radu `queue.enqueueWriteBuffer(buffer, CL_TRUE, 0, sizeof(int), obj)`, kde „obj“ je objekt ktorý sa posiela programu v podobe bufferu.
- Keďže kernelové funkcie nevracajú žiadne hodnoty, je potrebné alokovať v programe aj miesto, kam sa má uložiť výsledok funkcie, vytvorením `cl::Buffer` pre výsledok `cl::Buffer result(context, CL_MEM_READ_WRITE, sizeof(type))`
- Posledným krokom je priradenie bufferov k argumentom funkcíi, vytvorenie a spustenie kernelu v OpenCL `cl::Kernel kernel = cl::Kernel(program, func_name)`, v tomto príkaze sa nastaví výpočet na funkciu s názvom, ktorý je uložený v premennej func_name v podobe stringu. Ak by funkcia v jazyku OpenCL C vyzerala takto: `void kernel func(global const int* A, global int* R) {telo funkcie …}`, spojenie argumentov s buffermi by vyzeralo nasledovne: `kernel.setArg(0, buffer); kernel.setArg(1, result);` Teda argumenty sa indexujú od 0 … n, v poradí, akom sú zapísané v hlavičke funkcie. Následne sa celý kernel vloží do príkazového radu aj s potrebnými parametrami `queue.enqueueNDRangeKernel(kernel,cl::NullRange,cl::NDRange(10),cl::NullRange)`,kde `cl::NullRange` určuje nula - dimenziónalne rozpätie, `cl::NDRange(10)` konštruuje jedno - dimenzionálne rozpätie, vstupný parameter určuje, koľko inštancií sa má výpočtu zúčastniť.
- Spustenie výpočtu prebehne po zavolaní príkazu `queue.finish()`, ktorý zabezpečí zbehnutie príkazového radu.
- Ak celý program zbehol úspešne, v bufferi „result“ sa nachádza výsledná hodnota výpočtu, ktorú je možné pomocou metódy príkazového radu `queue.enqueueReadBuffer(result, CL_TRUE, 0, sizeof(type), &R);` uložiť do premennej `type R` mimo frameworku OpenCL, a ďalej s ňou pracovať.

Kód v OpenCL C (kernel):

```
__kernel void par_sum(__global int* buffer, __global int *scratch, __global int *len,
__global int *result) {
	int global_index;
	int accumulator;
	size_t global_size;
	int local_len;

	barrier(CLK_GLOBAL_MEM_FENCE);
	global_index = get_global_id(0);
	accumulator = 0;
	global_size = get_global_size(0);
	local_len = *len;

	while (global_index < *len) {
		int element = buffer[global_index];
		accumulator += element;
		global_index += global_size;
	}

	global_index = get_global_id(0);
	scratch[global_index] = accumulator;
	barrier(CLK_GLOBAL_MEM_FENCE);
	for (int last_size = global_size; last_size > 1;
	  last_size = (last_size + 1)/2) {
		int half_block = (last_size + 1) / 2;
		if (global_index < last_size/2) {
			int other = scratch[global_index + half_block];
			scratch[global_index] += other;
		}
		barrier(CLK_GLOBAL_MEM_FENCE);
	}

	if (global_index == 0) {
		*result = scratch[0];
	}
}
```

Na strane hosta sa zabezpečuje aj nahrávanie kernel kódu do zariadenia, priradenie buffrov k argumentom kernelovej funkcie, nastavenie indexovania, určenie počtu výpočtových jednotiek, ktoré budú výpočet spracovávať.

Kód na strane kernelu je kompilovaný v pre dané zariadenie a vykonávajú ho každá výpočtová jednotka. Každá táto jednotka ktorá sa zúčastňuje na výpočte tento kernelový kód vykonáva.

3 Návrh

V tejto kapitole si popíšeme celkový návrh aplikácie, návrh komunikácie procesov Imagine Logo a OpenCL daemon. Predstavíme diagram aplikácie spolu s popisom jeho komponentov, návrh odosielania, prijímania a spracovania dát na strane Imagine Logo a taktiež na strane démona.

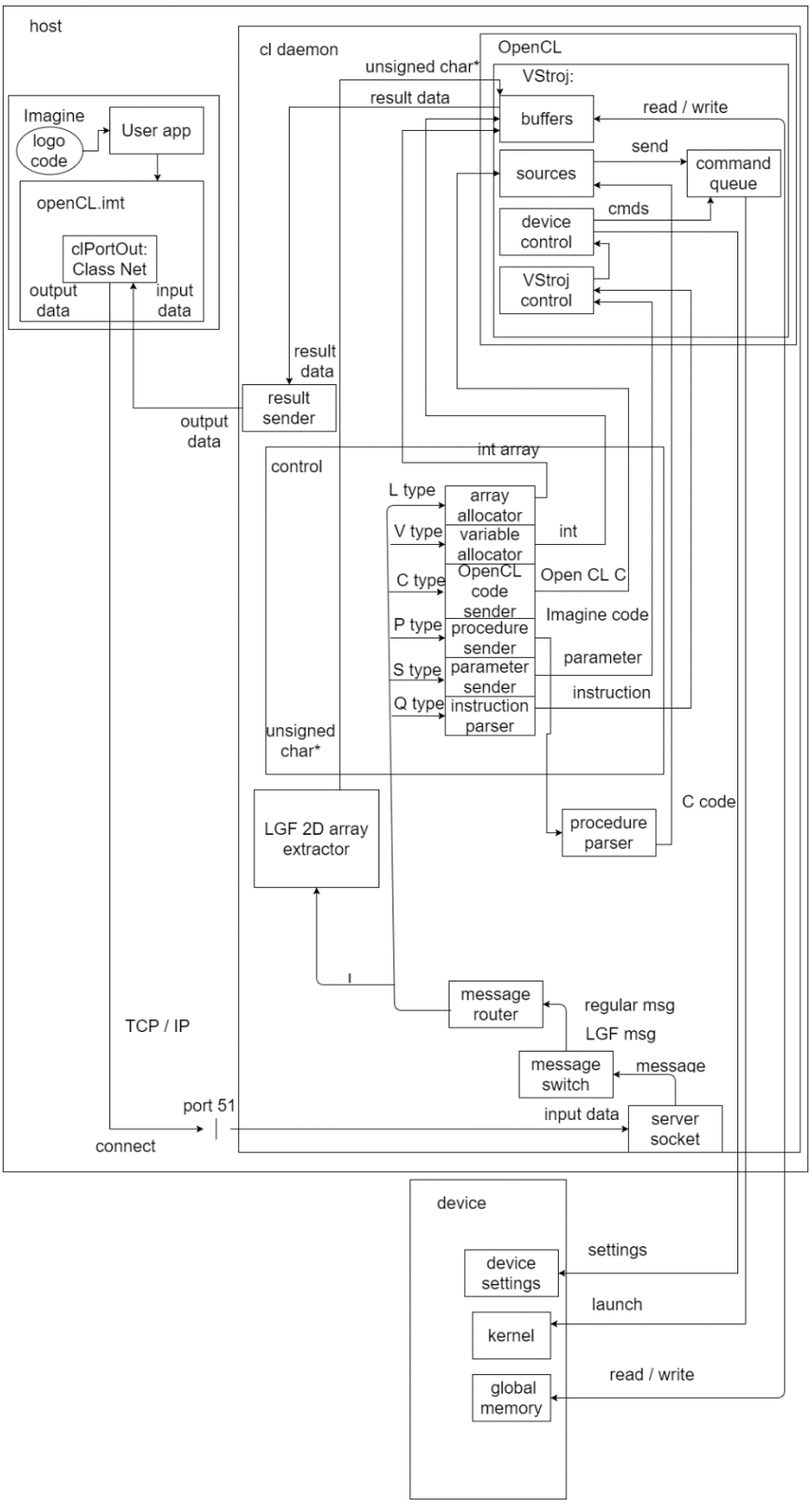
Ďalej sa budeme venovať návrhu samotného rozšírenia, rozšírenie významovo rozdelíme na dva prístupy k paralelným výpočtom. Prvý prístup bude predstavovať OpenCL API v prostredí jazyka Imagine logo, pomocou ktorého bude možné programovať procedúry v Imagine s využitím množiny OpenCL API, podobne ako v programovacích jazykoch C, C++.

V tomto prvom prístupe bude zasielať požiadavky démonovi, aby alokoval potrebnú pamäť pre dáta, vytváral workgroupy, či spúšťal kernelové funkcie. API nám umožní prostredníctvom Imagine Logo zaslať démonovi kernelový kód naformátovaný v regulárnej správe. Tento kód môžeme prostredníctvom Imaginu získať z textového súboru, alebo ho jednoducho poskladať ako string do premennej a tú odoslať.

Druhý prístup zavedieme prostredníctvom OpenCL API v Imagine. Od prvého sa takmer nelíši, jediný rozdiel, kvôli ktorému sme rozšírenie logicky rozdelili na dva prístupy je v tom, že druhým prístupom nebudeme posielať kernelové funkcie, ale budeme pomocou inštrukcii informovať démona o tom, aký z predprogramovaných kernelov má použiť. V tomto prístupe taktiež pre vybrané kernely zavedieme možnosť napísania procedúry v Imagine, ktorá sa preloží do jazyka C a bude sa volať pre každý prvok dátovej štruktúry ktorou prejde každá výpočtová jednotka kernelu. Tento spôsob nám umožní riešiť špecifický paralelný problém, napríklad mapovanie dátovej štruktúry všeobecne, pretože bude možné v Imagine vytvoriť vlastnú procedúru ktorá sa aplikuje na každý prvok napríklad pri spomenutom mapovaní dátovej štruktúry.

V tento kapitole si tiež povieme o návrhu prekladu procedúr z Imagine do jazyka C a o obmedzeniach, ktoré OpenCL paralelné riešenia prinášajú a musíme s nimi počítať pri tvorbe aplikácie.

3.1 Diagram aplikácie



Obrázok 5: Diagram aplikácie

3.2 Popis komponentov diagramu

Host komponenty:

Imagine:

- V Imagine bude naprogramovaný projekt OpenCL.imt, ktorý bude obsahovať procedúry pre dátovú komunikáciu s démonom. Tieto procedúry sú navrhnuté tak, aby pokrývali oba prístupy rozšírenia paralelného výpočtu na GPU, teda prístup s automatickým prekladom a prístup s použitím API v Imagine s využitím posielania kernelových kódov. Používateľ Imaginu si naprogramuje procedúru, ktorú pomocou procedúr v tomto projekte odošle do grafickej karty, kde sa spustí paralelný výpočet.
- Projekt OpenCL.imt obsahuje objekt triedy Net, ktorým sa Imagine spojí s démonom pomocou socket komunikácie na porte 51. Prostredníctvom tejto triedy bude umožnená dátová komunikácia medzi týmito dvoma procesmi

CL daemon

- Server socket bude predstavovať komunikačný proces s Imaginom na strane démona. V tomto komponente implementujeme spracovanie packetov z Imaginu a taktiež odosielanie packetov do Imaginu. Podľa typu packetu prijatého z Imaginu (z protokolu o komunikácii Imagine procesov, implementovanom v Ctrllab4Imagine) sa tieto packety ďalej spracujú.
- Dátové pakety, obsahujúce regulárne správy alebo LGF súbory budú odoslané do message routeru, kde sa spracujú podľa toho o aký typ regulárnej správy ide. Message router posiela ďalej packety podľa typu buď do LGF 2D array extractora ak ide o LGF súbor, alebo do controllera ak ide o regulárnu správu.
- Ak do routeru príde správa obsahujúca LGF súbor, tá sa odošle do LGF 2D array extractora, kde sa vyextraktuje 2D pole. Tento súbor obsahuje množstvo Imagine hlavičiek a sekcií, ktoré je potrebné spracovať korektne. Vývojári Imaginu nám dodali protokol, v ktorom je popísaná štruktúra LGF súboru. Obrázok je v ňom uložený vo formáte Windows Bitmap a zakomprimovaný knižnicou podobnou zlib. Pomocou knižnice zlib tieto komprimované dáta dekomprimujeme a následne uložíme do bufferu.
- Ak router zistí že ide o regulárnu správu, zistí jej hlavičku, ktorá sa naformátuje ešte v Imagine pred odoslaním dát. Podľa hlavičky sa správy ďalej odošlú do príslušných komponentov, v ktorých sa rozložia a spracujú.

- Array allocator zabezpečuje prečítanie správy obsahujúcej pole vo formáte Imagine list, následné zistenie počtu prvkov, vytvorenie int poľa a alokovanie poľa v bufferi. Keďže OpenCL buffer má konštruktor, ktorý vstupy kopíruje, alokovanú pamäť je možné po uložení do Bufferu uvoľniť
- Variable allocator prečíta správu obsahujúcu premennú, získa jej hodnotu, pretypuje na C int a odošle do bufferu
- OpenCL code sender prečíta správu, v ktorej sa nachádza kernel kód v jazyku OpenCL C poslaný z Imaginu v podobe stringu. Tento kód potom odošle do vektora zdrojových súborov sources.
- Procedure sender dostane správu ktorá obsahuje hlavičku a telo Imagine procedúry. V procedure parser sa táto procedúra preloží do jazyka C. Táto procedúra sa ďalej odošle do komponentu sources, v ktorom sa uloží do zdrojových súborov. Pamäť ktorá sa alokuje pri preklade procedúry nie je možné uvoľniť ihneď po uložení do buffera. Toto uvoľnenie sa bude vykonávať hromadne pri uvoľňovaní celkovej pamäte démona.
- Komponent Parameter sender zo správy získa parameter a hodnotu ktorá sa doňho uloží. Tento komponent nastavuje parametre výpočtu, napríklad ID result bufferu, NDRange workgroupy atď.
- Instruction parser získa zo správy inštrukciu, ktorú potom odošle do VStroj control, kde sa vykoná. Týmito inštrukciami je možné riadiť chod programu, buildovať zdrojové súbory, spúšťať kernely, dealokovať pamäť atď.
- VStroj komponent bude predstavovať triedu, ktorá bude nadstavbou OpenCL frameworku. Táto trieda v sebe bude udržiavať dáta ako sú zdrojové súbory kernelových funkcií, buffre, objekty na zariadenia, platformy, príkazové rady, kontext a iné. Taktiež v nej naprogramujeme množstvo funkcií ktoré budú nad týmito dátami pracovať
- Sources je komponent VStroj, v ktorom sa ukladajú kernelové funkcie a všetky funkcie volané z kernelových funkcií.
- Device kontrol je komponent, ktorý odosiela inštrukcie do zariadenia, rôzne nastavenia a pod.
- Command queue reprezentuje prepojenie so zariadením, pomocou ktorého je možné posielať zariadeniu príkazy. Command queue spúšťa v zariadení kernelový kód.

Device komponenty:

- Device settings komponent obsahuje nastavenia zariadenia, ako počet jednotiek, ktoré sa majú na danom kernely vykonať, informácie o správe pamäti v zariadení a podobne
- Kernel komponent predstavuje jednu výpočtovú jednotku, ktorá sa podieľa na výpočte. Táto výpočtová jednotka plní príkazy dané v kernelovom kóde, pristupuje do pamäti, číta z nej a zapisuje do nej
- Global memory v zariadení predstavuje globálnu pamäť na ktorú ukazujú buffre z hosta. Táto pamäť je prístupná pre výpočtové jednotky, ktoré sú v danom kernely aktívne.

3.3 Návrh komunikácie medzi Imagine Logom a démonom

Imagine Logo komunikuje s paralelne spustenými procesmi pomocou triedy Net. Systém má prijímať správy z Loga a následne ich vhodne spracovať a odoslať výsledky riešenia naspäť na stranu Imagine Loga. Na implementovanie tejto komunikácie použijeme zdrojové súbory aplikácie Ctrllab4Imagine, v ktorých je už komunikácia s Imaginom implementovaná. Prenos dát budeme realizovať ako prenos regulárnych správ a nebudeme využívať ďalšie typy z protokolu komunikácie. Démon má rozoznať o aký typ informácie z Loga ide, preto na strane Imaginu tieto informácie vhodne naformátujeme a taktiež pri odosielaní dát z aplikácie do Imaginu naformátujeme paket tak, aby Imagine rozpoznal dáta a dokázal ich spracovať.

Z aplikácie budeme do Imaginu posielať 2 typy dát ako výsledok výpočtu:

- číslo – uložíme do premennej
- list – potrebné použiť Imagine metódu parse pred uložením do premennej

Z Imaginu budeme do aplikácie posielať 2 typy dát:

- naformátovaný reťazec znakov – obsahuje premenné, polia, procedúry, inštrukcie
- LGF – Imagine súbor LGF, ktorý obsahuje Windows bitmapu

3.4 Spracovanie dát v aplikácii

Aplikácia po prijatí správy z Imaginu zistí o aký typ správy ide a podľa toho ju spracuje. Základné typy správ:

- potvrdzovacie – slúžia na vytvorenie funkčného spojenia medzi Imaginom a aplikáciou, sú z protokolu o komunikácii paralelne bežiacich Imagine procesov
- regulárne – tieto správy použijeme na prenos premenných, polí a inštrukcií, ktoré v aplikácii spracujeme
- LGF – správa ktorá obsahuje obrázok vo formáte LGF

3.4.1 Regulárne správy

Správy sa formátujú v tvare TYP#[telo správy]

Rozpis jednotlivých dátových správ:

- V#[value] – správa obsahujúca premennú
- L#[list_data] – správa obsahujúca list
- P#[[args] [body]] – správa obsahujúca procedúru napísanú v jazyku Imagine Logo
- Q#[#instruction] – správa obsahujúca inštrukciu, ktorú démon vykoná.
  - #RUN – spustenie kernelu
  - #BUILD – zostavenie programu
  - #GETVRESULT – získanie návratovej hodnoty premennej zo zariadenia
  - #GETLRESULT – získanie návratovej hodnoty poľa zo zariadenia
  - #CLEAR – uvoľnenie alokovanej pamäte buffrov a príkazového radu
- C#[kernel code] – správa obsahuje kernel kód napísaný v jazyku OpenCL C v podobe stringu
- S#[[name][value]] – nastaví parameter démona na danú hodnotu
  - KERNEL_NAME – ako hodnotu očakáva názov kernelu, ktorý bude zodpovedný za výpočet
  - ND_RANGE – ako hodnotu očakáva hodnotu, alebo 1-3 prvkové pole hodnôt pre vytvorenie workgroupy typu GLOBAL
  - STORED_KERNEL – ako hodnotu očakáva názov kernelovej funkcie, ktorá je naprogramovaná v aplikácii ako všeobecné riešenie paralelného indexovania dátových štruktúr

- RESULT_BUFFER_ID – ako hodnotu očakáva id bufferu alokovaného v aplikácii. Hodnotu v tomto bufferi potom vráti ako výsledok

3.5 Preklad jazyka

Myšlienka prekladu jazyka z Imagine Loga do paralelného kódu spočíva v nájdení podmnožiny príkazov Imagine Loga, ktoré sa dajú spustiť v Imagine Logo sekvenčne, ale na zariadení sa preložia do kernelového kódu a výpočet prebehne paralelne. Ideálne je použitie takých príkazov, aby používateľ nepotreboval znalosť jazyka OpenCL C, teda vôbec nemusel mať podrobnú znalosť kernelových funkcií. Vhodnými príkladmi takýchto výpočtov sú také, ktoré prechádzajú každým prvkom dátovej štruktúry a na každom z nich aplikujú výpočet. Iterovanie dátových štruktúr v jazyku Imagine Logo pomocou indexovania je neefektívne, pretože pri zmene hodnoty na danom indexe sa nakopíruje znovu celá dátová štruktúra so zmenenou hodnou na danom indexe. Imagine Logo dokáže pristupovať k dátovým štruktúram pomocou metód first, last, butfirst efektívne, no v tomto prípade by bolo potrebné vyrobiť novú dátovú štruktúru. Keďže metóda first pristupuje k prvému prvku dátovej štruktúry, preklad by bol komplikovaný, pretože v paralelizme sa pristupuje k prvkom štruktúr cez indexy a každá výpočtová jednotka má v kernelovej funkcii naprogramovaný prístup k svojej množine indexov. Preto sme sa rozhodli prekladať také procedúry z Imagine Loga, ktoré sa volajú z kernelových funkcií, pričom kernelové funkcie budú predprogramované na strane démona a každý kernelový kód bude prechádzať dátovou štruktúrou a posielať na spracovanie do preloženej procedúry také argumenty, aké si daný problém pre riešenie vyžaduje.

3.5.1 Preklad procedúr z Loga

Rozhodli sme sa prekladať procedúry, ktoré sa budú volať z kernelových funkcií. Tieto procedúry sa preložia z jazyka Imagine Logo do jazyka C. Sekvenčný prístup týchto procedúr ostane zachovaný, pretože sú to funkcie, ktoré používa každá výpočtová jednotka. Paralelizmus na dátovej štruktúre budeme teda implementovať v kernelových funkciách, ktoré budú zahŕňať všeobecné riešenia pre širokú škálu výpočtov. Kernelové funkcie budú naprogramované na strane démona a z Imaginu sa budú zasielať požiadavky na nastavenie výpočtu pre tento kernel. Ak kernel volá aj nejakú procedúru v jazyku C, z Imaginu ju môžeme odoslať, démon ju preloží do jazyka C a táto procedúra sa bude volať vo výpočte pre každý prvok dátovej štruktúry.

Popis prekladu imagine procedúry:

- Aplikácia preloží typ a názov procedúry na inline int func(){}
- OpenCL je nadstavba jazyka C podľa štadartu C99, preto musíme túto metódu vložiť do zdrojového súboru nad kernelovú funkciu, ktorá bude obsahovať volanie funkcie func s potrebnými argumentami

Popis prekladu hlavičky procedúry:

- V procedúrach Imaginu sú argumenty procedúr vo formáte :arg1 :arg2 .. :argN, keďže obmedzíme aplikáciu na výpočty integer typov, tieto argumenty preložíme ako int arg1, int arg2 .. int argN

Popis prekladu príkazov v tele procedúry:

- Príkaz let :name :value – v Imagine tento príkaz vytvára lokálne premenné. Budeme prekladať podľa toho, či chceme definovať premennú, alebo pole. Preto najprv zistíme, či :value nemá formát [ ]. Ak má tento format, potom ide o list a teda ho preložíme do jazyka C ako pole int name [počet prvkov listu] = {prvok1, prvok2, .., prvokN}; Ak :value nie je list, potom sa príkaz let preloží ako premenná int name = value;
- Príkaz make :name :value – tento príkaz v Imagine vytvára globálne premenné, ale pokiaľ existuje nejaká lokálna premenná vytvorená príkazom let, tak do nej priradí hodnotu. V našich funkciách nebudeme používať globálne premenné, preto sa príkaz make bude prekladať ako name = value; a teda bude priraďovať hodnotu do už vytvorenej premennej.
- Príkaz item :id :arr – tento príkaz vracia hodnotu prvku poľa na danom indexe. V Imagine sú indexy od 1 – N, preto pri preklade musíme tento index znížiť o 1. Výsledne príkaz preložíme ako arr[id], za týmto príkazom nenasleduje bodkočiarka, pretože je súčasťou príkazov, ktoré modifikujú premenné a bodkočiarka musí byť až na konci týchto príkazov.
- Príkaz op :value – v týchto funckiách budeme potrebovať návratovú hodnotu, preto sme do prekladu zahrnuli aj príkaz op, ktorý preložíme na return value;

- if, ifelse – tieto príkazy umožňujú v Imagine písať podmienky. Aplikácia ich bude prekladať s využitím zásobníka a zvládne aj preklad vnorených if-ov. Príkazy v týchto vetvách preloží podľa vyššie spomenutých bodov.

```
to testIFELSE_VNORENIA :a :b :c
let "v1 10
let "v2 20

if :a <= :b [
  make "v1 15
]

if :b = :c [
  make "v2 25
]

ifelse :b > a [
  if :b < 10 [
  make "v1 10
  ]
]
[
  if :b >= 10 [
  make "v1 10
  ]
  make "v2 10
]
op :v1
end
```

```
inline int func (int a ,int b ,int c) {
  int v1 = 10 ;
  int v2 = 20 ;
  if( a <= b ) {
    v1 = 15 ;
  }
  if( b == c ){
    v2 = 25 ;
  }
  if( b > a ){
    if( b < 10 ){
      v1 = 10 ;
    }
  }
  else{
    if( b >= 10 ){
      v1 = 10 ;
    }
  v2 = 10 ;
  }
  return v1 ;
}
```

Obrázok 6: Ukážka prekladu procedúry z Imagine Loga do jazyka C

3.6 Obmedzenia aplikácie

Aplikáciu obmedzíme na prácu s celočíselnými typmi C, teda int, char a unsigned char, ktoré na pokrytie potrebnej funkcionality postačujú.

Nastavenie platformy a zariadenia zavedieme v konzolovom menu aplikácie, ktoré sa zobrazí po spustení.

Nebudeme zavádzať návratové hodnoty paralelných procudúr, tieto hodnoty sa uložia do zásobníka, ktorý bude naprogramovaný v projekte OpenCL.imt a po prijatí správy Imagine upozorní na to, že démon poslal výsledok. Toto obmedzenie ušetrí čas pre riešenie veľmi komplikovaných riešení, pretože takto môžu Imagine a démon fungovať asynchrónne.

V preložených funkciách taktiež nebudeme zavádzať smerníky na funkcie a dynamické alokovanie pamäte, pretože to OpenCL nepodporuje.

Keďže OpenCL nepodporuje rekurzívne volania funkcií, nebudeme prekladať volania rekurzívnych funkcií. Preto procedúry ktoré budú volané z kernelových funkcií nesmú byť rekurzívne. Toto obmedzenie nemá vplyv na funkcionalitu, pretože každá rekurzívna funkcia sa dá napísať v podobe cyklov.

```
to stromcek :d :n
if :n > 0
[
  fd :d
  lt 45
  stromcek :d / 2 :n - 1
  rt 90
  stromcek :d / 2 :n - 1
  lt 45
  bk :d
]
end
```

```
to strom_iter :d :n

 let "vnorenia [4]
 let "stav 0
 let "an :n + 1
 let "ad :d * 2

 while [not empty? :vnorenia]
 [
  ifelse 0 = :stav
  [
   ifelse :an = 1
    [make "stav 4]
    [make "stav 1]
  ]
  [ifelse 1 = :stav
   [
    make "ad :ad / 2
    make "an :an - 1
    fd :ad
    lt 45
    inc "stav
    make "vnorenia fput :stav :vnorenia
    make "stav 0
   ]
   [ifelse 2 = :stav
    [
     rt 90
     inc "stav
     make "vnorenia fput :stav :vnorenia
     make "stav 0
    ]
    [ifelse 3 = :stav
     [
      lt 45
      bk :ad
      make "ad :ad * 2
      make "an :an + 1
      make "stav 4
     ]
     [
      make "stav first :vnorenia
      make "vnorenia bf :vnorenia
     ]
    ]
   ]
  ]
 ]

end
```

Obrázok 7Príklad prepisu rekurzívnej procedúry pomocou while cyklov

4 Implementácia

4.1 Implementácia démona

Démon je naprogramovaný v jazyku C++ v prostredí Visual Studio 2015. Skladá sa z viacerých komponentov, ktoré spolu úzko spolupracujú. V aplikácii prebieha spracovanie inštrukcií a dátových štruktúr z Imagine Loga. Používateľom umožňuje dva druhy prístupu k paralelnému spracovaniu dát.

Prvý prístup pokrýva podmnožinu OpenCL v podobe API priamo z Loga pomocou predprogramovaných funkcií v Imagine, čo zahŕňa alokovanie listov, premenných, posielanie kernelového kódu v podobe stringu a nastavenie workgroupy a globálneho indexovania. Tento prístup by mohli využiť skúsení programátori v Imagine na efektívne spracovanie dát. Je však potrebná znalosť kernelových funkcií, keďže ich treba posielať v ich OpenCL C tvare a taktiež je nutné mať jednoduchý prehľad o funkcionalite OpenCL frameworku. Metódy triedy OpenCL sú v jazyku Imagine Logo naprogramované v podobe procedúr a zjednodušené na prístup ku globálnej pamäte. Od týchto procedúr démon príma regulárne správy, ktoré spravuje a podľa parametrov obsiahnutých v správach démon nastavuje Open CL pomocou triedy VStroj.

Druhý prístup umožňuje posielanie dát na výpočet na stranu démona bez potrebnej znalosti OpenCL a OpenCL C. Démon tieto procedúry prekladá na kernelové funkcie automaticky. K tomuto prístupu sme v jazyku Imagine Logo naprogramovali procedúry ktoré správne odošlú potrebné dáta do aplikácie a taktiež odošlú procedúru, ktorá sa v démonovi prekladá. Automatický preklad pokrýva procedúry, ktoré spracovávajú všetky prvky dátovej štruktúry. Pre potreby tejto implementácie sú v démonovi zakompilované predprogramované kernelové kódy, ktoré posielajú dané parametre do preloženej funckie. Ak si používateľ zvolí použitie jedného z uložených kernelových kódov, musí dodržať pri programovaní poradie a počet vstupných argumentov procedúry, ktorú prepája s týmto kernelom.

4.1.1 Spracovanie dát

Pre potrebu dynamického spracovania dát sme v aplikácii zaviedli vector buffrov, do ktorého sa ukladajú prijaté premenné a vector zdrojových súborov, do ktorého sa ukladajú prijaté kernelové kódy, alebo preložené procedúry.

Spracovanie dát na strane démona prebieha tak, že aplikácia zistí akým prefixom správa začína. Podľa protokolu v kapitole návrhu 3.4.1 démon zistí prefix a vnútro správy ďalej posiela príslušných funkciám.

Zdrojový kód smerovania regulárnych správ podľa prefixu:

```
void messageSwitch(char *msg) {
	char * prefix = getPrefix(msg);
	char * body = getBody(msg);

	switch (*prefix) {
	case 'V': allocVariable(body); break;
	case 'L': allocList(body); break;
	case 'P': addProcedure(body); break;
	case 'Q': makeInstruction(body); break;
	case 'C': addKernelCode(body); break;
	case 'S': setParameter(body); break;
	default: break;
	}
	delete[] prefix;
	delete[] body;
}
```

Aj správa obsahuje iný prefix, ignoruje sa a packet sa vymaže bez spracovania.

Popis spracovania jednotlivých typov regulárnych správ:

- V: tento prefix určuje, že správa z Imaginu obsahuje premennú. Telo tejto správy sa ďalej odošle do funkcie allocVariable, ktorá túto premennú uloží do vectora OpenCL Buffrov s parametrami, ktoré definujú veľkosť v pamäti potrebnú na alokovanie premennej a určujú typ prístupu do tejto pamäte. Funkcia allocVariable uloží taktiež veľkosť premennej (počet prvkov) do vectora, v ktorom si démon udržiava informácie o veľkostiach dát, ktoré prijal.
- L: správy s týmto prefixom obsahujú prvky listu z Imaginu. Telo tejto správy, v ktorom sa nachádzajú tieto prvky listu sa odosielajú do funkcie allocList, ktorá túto správu prečíta a spočíta počet prvkov. Potom alokuje dostatočne veľké pole typu int, do ktorého tieto prvky uloží. Keď sú prvky v int poli, funkcia ich uloží do bufferu s parametrami podobne ako pri type V, len namiesto dĺžky 1 do vectora, v ktorom si démon udržiava informácie o veľkostiach dát uloží hodnotu zodpovedajúcu počtu prvkov tohto poľa.
- P: správa s prefixom P obsahuje procedúru v jazyku Imagine Logo. Telo správy sa odosiela do funkcie addProcedure. Táto funkcia odosiela Imagine procedúru do parseru, ktorý ju preloží podľa definície prekladu popísanú v kapitole návrh

3.5.1. Z parseru sa do funkcie addProcedure vráti string obsahujúci už túto procedúru preloženú do funkcie jazyka C. Táto funkcia sa nakoniec pridá do vectora zdrojových kódov.

- Q: tento prefix určuje že ide o regulárnu správu, v ktorej tele sa nachádza inštrukcia, ktorú démon vykoná. Medzi tieto inštrukcie patrí spustenie kernelu, zmazanie zdrojových kódov
- C: správy tohto typu obsahujú v tele kernelový kód v jazyku OpenCL C, ktorý Imagine poslal v podobe stringu. Tento kód démon pridá do vectora zdrojových kódov vo funkcii addKernelCode
- S: prefix, ktorý určuje že správa obsahuje parameter. Telo tejto správy obsahuje dve sekcie. V prvej sekcii sa nachádza informácia o aký typ parametru ide a v druhej sekcii sú dáta, ktoré sa tomuto parametru v aplikácii nastavia. O túto činnosť sa stará funkcia setParameter, v ktorej sa parametre rozlišujú a ďalej posielajú príslušným funkciám, tak ako je zobrazené v ukážke zdrojového kódu:

```
char * line = getBodyString(body);
char * type = getBodyString(line, 0);
char * value = getBodyString(line, 1);
if (strcmp(type, "KERNEL_NAME") == 0) setKernel(value);
else if (strcmp(type, "ND_RANGE") == 0) setNDRange(value);
else if (strcmp(type, "STORED_KERNEL") == 0) setStoredKernel(value);
else if (strcmp(type, "RESULT_BUFFER_ID") == 0) setResultID(value);
```

4.1.2 Popis kódov kernelových funkcii obsiahnutých v démonovi

- p1D1F – parameter, podľa ktorého démon rozpozná, že používateľ Imaginu chce použiť kernelový kód, ktorý prechádza všetky prvky listu a na každý z nich chce aplikovať procedúru. Tento kód dostáva dva argumenty (__global int * A, __global int * len). Používateľ nenastavuje počet prvkov listu, tento argument sa nastaví automaticky. Pred spúšťaním tohto kernelu stačí odoslať list a procedúru s jedným argumentom, ktorej bude tento kernel odosielať daný prvok, na ktorý sa aplikujú príkazy v procedúre. Tento kernelový kód predstavuje mapovanie jednorozmernej dátovej štruktúry.
- p1DavgNum – parameter, podľa ktorého démon rozpozná že používateľ Imaginu chce použíť kernelový kód, ktorý priemeruje prvky poľa. Tento kód nemá volanie na funkciu, dostáva tri argumenty. Smerník na pole v globálnej pamäti, dĺžku poľa a po-

čet prvkov idúcich za sebou, z ktorých je potrebné urobiť priemer. Kód je naprogramovaný tak, že sa pole rozdelí na sekcie dĺžky rovnej počtu prvkov ktoré sa budú priemerovať, ďalej tieto sekcie rovnomerne pridelí každej výpočtovej jednotke, ktorá ich prechádza a priemeruje. Posledná jednotka vždy spracuje zvyšok poľa po jeho koniec. Tento typ kernelu je určený pre výpočty typu RGB2Grayscale, teda trojice pixlov idúcich po sebe, ktoré sú reprezentované v podobe jedno-rozmerného poľa. Kernel pole neredukuje, do idexov poľa, z ktorých robí priemer vkladá priemer ktorý získa.

- p1DKParSum – tento parameter určí démonovi že prišla požiadavka pre použitie kernelu, ktorý dostáva 4 argumenty: smerník na vstupné pole v globálnej pamäti, smerník na výpočtové pole scratch, ktorého dĺžka je rovná polovici dĺžky vstupného poľa, dĺžku vstupného poľa a smerník na výsledok typu int. Tento kernel funguje na princípe, ktorý je znázornený na obrázku 2 a paralelne sčítava prvky poľa.

4.2 Implementácia na strane Loga

V jazyku Imagine Logo sme naprogramovali v projekte OpenCL.imt procedúry pre oba prístupy. Každá procedúra ktorá odosiela údaje aplikácii naformátuje svoj výstup, ktorý odošle prostredníctvom procedúry cl_sendMessage, v ktorej sa údaje posielajú cez zbernicu na port 51 pomocou Imagine triedy Net. Objekt tejto triedy je v projekte OpenCl.imt nazvaný ako ClPortOut. Tieto správy sa posielajú metódou ClPortOut 'send. Prijaté správy do Imaginu sú tiež typu regulárnych správ a tieto v Imagine parsujeme metódou parse, ktorá je naprogramovaná vývojármi Imaginu. Táto metóda vyrobí zo vstupného stringu list. Ak je list veľkosti 1, výsledok sa uloží do premennej a vloží do zásobníka stack. Ak je list väčší, výsledok mal byť vo forme listu a tento list vložíme do zásobníka.

Imagine formátuje odosielané správy podľa návrhu odosielania regulárných správ v kapitole Návrh 3.4.1.

Problém pri implementácii v jazyku Imagine Logo je, že dátová štruktúra Imaginu – list, nie je určená pre veľké dátové súbory. Počet prvkov ktoré zvláda uložiť je do 100 000, čo nepostačuje pre obrázky vyššieho rozlíšenia. Pôvodne sme chceli implementovať aj možnosť vyčítať z plochy hodnoty pixlov do listu, ktorý by sme odoslali na stranu démona a tam spracovávali ako jedno-rozmerné pole reprezentujúce obrázok. Toto pole sme chceli potom spracované vrátiť naspäť na stranu Imagine Loga, aby sme ho mohli prípadne vykresliť, no limity štruktúr list nám nedovoľujú toto rozšírenie aplikovať. Preto sme

implementovali rozšírenie, ktoré dokáže spracovávať iba LGF súbory. Toto rozšírenie pre LGF súbory nechávame však otvorené vzhľadom na nemožnosť odosielania veľkých dátových štruktúr v podobe listov.

4.2.1 Realizácia rozšírenia na strane Imagine Logo

Pri vytváraní tohto rozšírenia sme naprogramovali v Imagine Logo procedúry, ktoré pokrývajú časť OpenCL frameworku v podobe OpenCL API v jazyku Imagine Logo. Pomocou tohto API v Imagine používateľ získava možnosť priamo posielať dáta do výpočtového zariadenia, nastavovať veľkosť worgroupy, odosielať kernelové kódy v podobe stringu do zariadenia, spúšťať ich, či získavať výsledky výpočtov. Pomocou tohto rozšírenia je možné naprogramovať procedúru, ktorá odošle potrebné dáta na stranu démona, spustí výpočet a odošle požiadavku o zaslanie správy obsahujúcej výsledok výpočtu.

Popis jednodlivých procedúr rozšírenia:

- cl_connect – procedúra vytvorí objekt triedy Net s potrebnými parametrami ak ešte objekt neexistuje. Tomuto objektu taktiež nastaví potrebné eventy onReceive, ktoré využívame pre upozornenie používateľa Imaginu o tom, že do Imaginu prišla správa s výsledkom a taktiež výsledok ukladáme do globálnej premennej stack za pomoci procedúry cl_receive.
- cl_getLResult – odošle požiadavku démonovi, ktorý následne odošle regulárnu správu do Imaginu, ktorá obsahuje naformátovane pole do podoby, z ktorej ho môže Imagine procedúra parse spracovať na list
- cl_getVResult – odošle na stranu démona požiadavku na získanie premennej, ktorá je výsledkom paralelného výpočtu
- cl_receive – táto procedúra sa volá z eventu Imagine Loga onReceive po prijatí správy. Upozorňuje používateľa o tom, že Imagine získalo dáta a potom tieto dáta ukladá do globálnej premennej typu list – stack
- cl_sendMessage – procedúra ktorá je spoločná pre všetky procedúry odosielajúce dáta v podobe regulárnych správ. Jej parametrom je :msg string, ktorý táto procedúra odošle objektom triedy Net clPortOut volaním jeho metódy send.
- cl_sendIntValue – procedúra ktorá dostane ako parameter číselnú hodnotu. Túto hodnotu odošle ako regulárnu správu prostredníctvom procedúry cl_sendMessage.

V tejto procedúre prebieha taktiež formátovanie správy aby démon rozpoznal že sa mu posiela premenná.

- cl_sendListData – táto procedúra dostáva ako parameter Imagine list, ktorý pomocou Imagine metódy „word“ procedúra prerobí na string, ktorý naformátuje na regulárnu správu s prefixom, pre správy obsahujúce Imagine list a tento string odošle procedúre cl_sendMesage
- cl_sendInsruction – táto procedúra očakáva vstupný argument string, ktorý odošle do démono prostredníctvom cl_sendMessage. Procedúra predtým naformátuje regulárnu správu, aby démon rozoznal, že ide o správu obsahujúcu inštrukciu.
- cl_sendKernelCode – procedúra odosiela na stranu démona kernelový kód v jazyku OpenCL C. Tento kód je možné napísať v Imagine ako string, alebo nahrať zo súboru do premennej a potom túto premennú pomocou cl_sendKernelCode odoslať. Táto procedúra formátuje regulárnu správu tak, aby démon rozoznal že mu prišiel tento kernelový kód. Metóda posiela regulárne správy za pomoci cl_sendMessage.
- cl_setParameter – procedúra obsahuje dva argumenty. Prvý je typ parametru a druhý je hodnota parametru. Tieto dva argumenty procedúra cl_setParameter naformátuje do dvoch sekcii, pridá prefix, ktorý určuje démonovi typ správy obsahujúcej nastavenia daného parametra a naformátovanú regulárnu správu odošle pomocou cl_sendMessage na stranu démona.
- cl_sendProcedure – procedúra očakáva ako vstup meno procedúry ktorý je naprogramovaná v Imagine Logu. Pomocou metódy „word“ sa táto procedúra prerobí na string. Do výslednej správy sa pridá prefix určujúci typ správy obsahujúcej procedúru. Správa sa odosiela démonovi prostredníctvom cl_sendMessage.
- cl_sendObject – procedúra odosiela neformátované dáta priamo prostredníctvom clPortOut’send. Táto procedúra dokáže odoslať akýkoľvek Imagine objekt. Využívame ju najmä na odosielanie LGF súborov, pretože LGF súbor nie je možné v Imagine skonvertovať na list, ani na string.

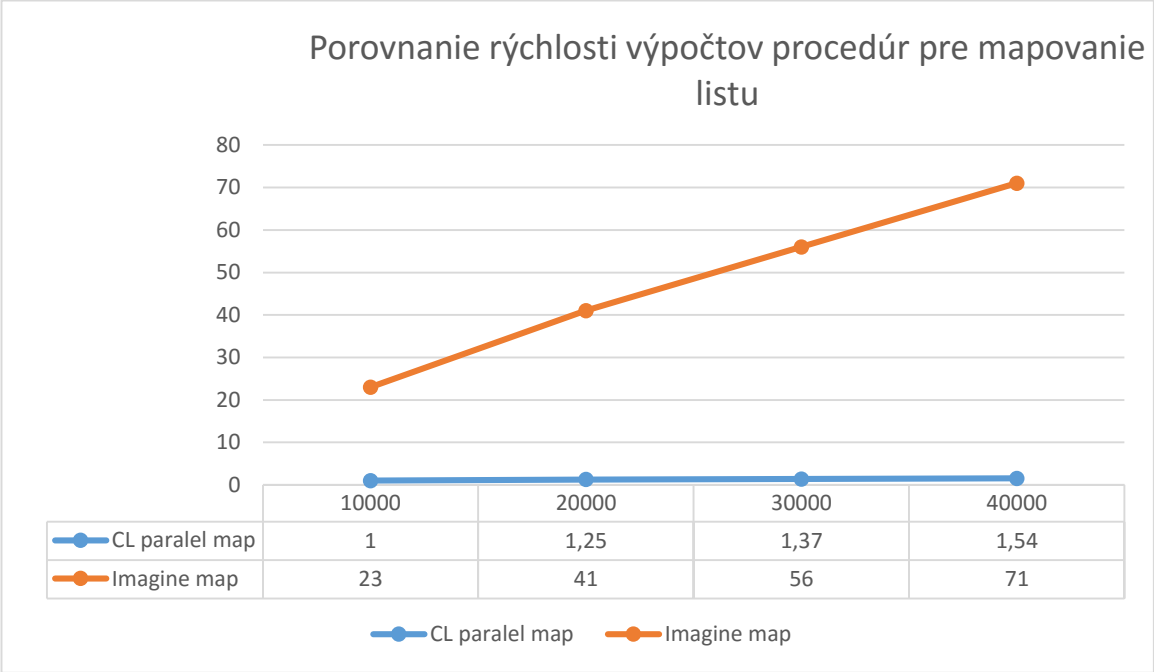
Ako príklad tvorby Imagine procedúry, pomocou ktorej môžeme riadiť paralelný prístup démona na zariadení uvedieme cl_map, ktorú sme naprogramovali v Imagine Logu pomocou vyššie uvedeného API ako všeobecné riešenie paralelného mapovania jedno-rozmerných dátových štruktúr:

```
to cl_map :arr :proc
 cl_sendInstruction "CLEAR

 cl_sendListData :arr
 cl_sendProcedure :proc

 cl_setParameter "STORED_KERNEL "p1D1F
 cl_sendInstruction "BUILD
 cl_setParameter "KERNEL_NAME "p1D1F
 cl_setParameter "ND_RANGE [100]
 cl_sendInstruction "RUN
 cl_setParameter "RESULT_BUFFER_ID 0
 cl_getLResult
end
```

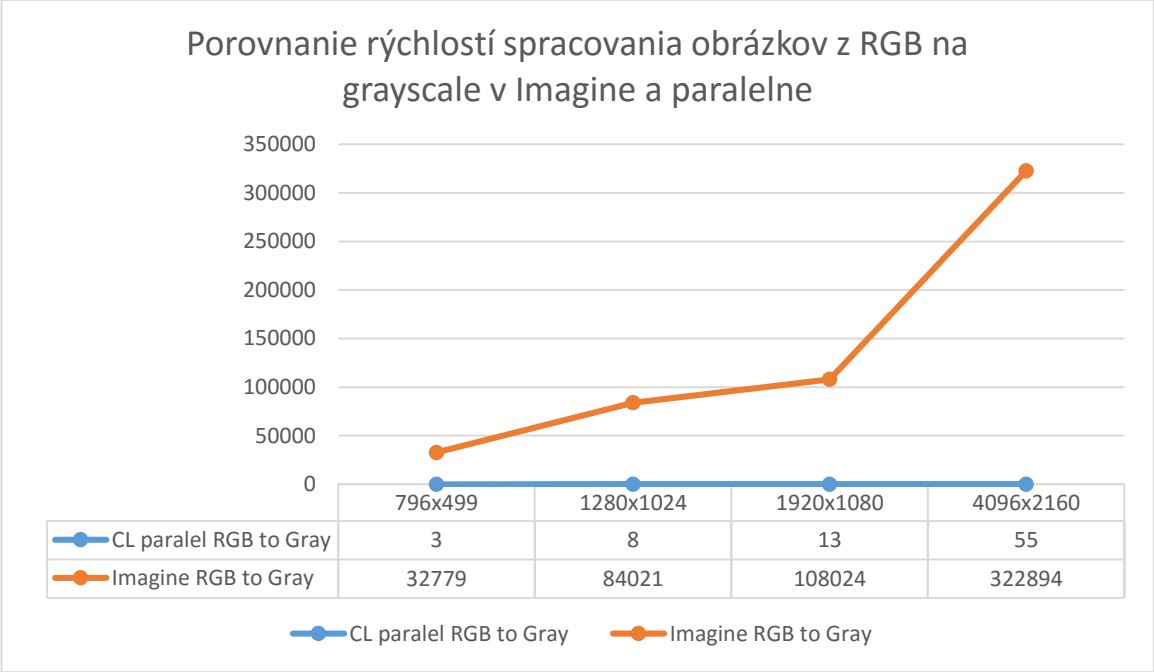
Obrázok 8: Príklad procedúry Imagine Logo komuniujúcej s démonom

Táto procedúra očakáva dva vstupné argumenty a to vstupné pole, na ktorom bude výpočet paralelne prebiehať a druhým vstupným argumentom je názov procedúry, ktorá sa preloží do jazyka C a bude volaná s príslušného kernelu.

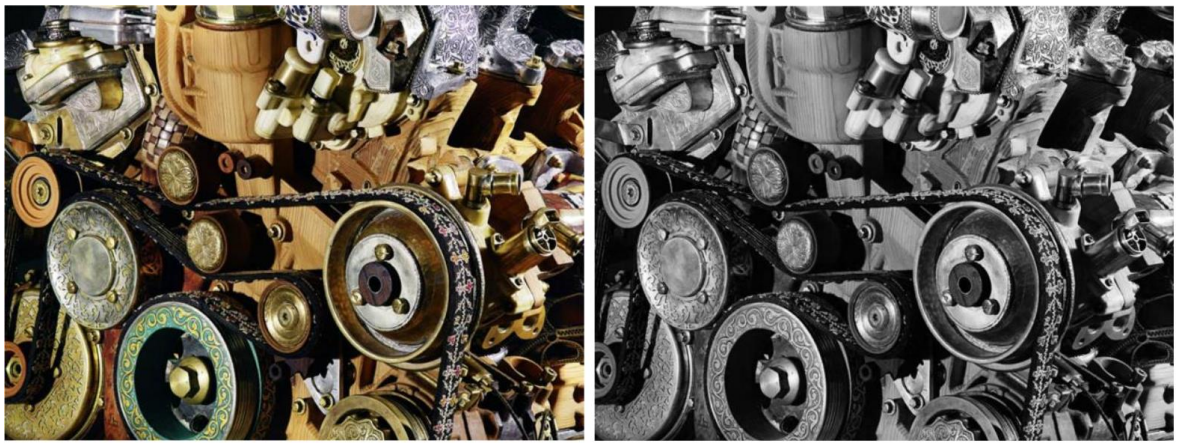
5 Vyhodnotenie efektívnosti rozšírenia



Analýzu sme robili od začiatku výpočtu po koniec, teda v Imagine pri volaní procedúry map a na strane démona pri vykonávaní kernelovej funkcie. Nebrali sme do úvahy vytváranie ani posielanie dátovej štruktúry. Graf znázorňuje porovnanie sekvenčného a paralelého výpočtu. V praxi však rozšírenie nie je natoľko rýchlejšie od Imaginu, pretože posielanie listu cez socket konečný výsledok spomaľuje



Analýza tohto výpočtu nie je úplne smerodajná, pretože v Imagine nedokážme vytvoriť štruktúru, ktorú by sme naplnili takým množstvom dát. Preto v Imagine vykresľujeme jednotlivé pixely obrázkov do plochy, kde si udržiavame informácie o hodnotách v pixloch. Taktiež využívame korytnačku, ktorej pohyb môže výpočet spomaliť.



Obrázok 9: Obrázok pred a po výpočte RGB na grayscale

6 Záver

Motiváciou tejto práce bolo preštudovať možností paralelného programovania a ich implementáciu v Imagine Logu. Po preštudovaní týchto možností bolo rozhodnuté o implementovaní rozšírenia Imagine Logo, ktoré dokáže paralelne spracovávať dáta a čo najviac toto rozšírenie sprístupniť pre riešenia širokej množiny výpočtov na dátových štruktúrach.

Aplikáciu sa nám podarilo navrhnúť a vytvoriť tak aby spĺňala požiadavky a bola ľahko rozšíriteľná v ďalších verziách. Naprogramovali sme OpenCL API v Imagine Logo, pomocou ktorého je možné programovať procedúry, ktoré dokážu komunikovať so zariadením. Toto rozšírenie sme doladili pre jedno-rozmerné výpočty na dátových štruktúrach Imaginu – listoch. Aplikácia obsahuje zakompilované kernelové kódy, ktoré je možné používať z Imaginu prostredníctvom zasielania požiadaviek s parametrom, ktorý určuje typ kernelového kódu, ktorý používateľ požaduje. Používateľom Imaginu sme umožnili aj programovanie vlastných procedúr v Imagine Logo, ktoré aplikácia preloží do jazyka C a sú volané z kernelového kódu prechádzajúceho dátovú štruktúru pre každý prvok tejto štruktúry, čo je paralelné mapovanie listu.

Rozšírenie taktiež ponúka možnosť naprogramovanie vlastného kernelového kódu, ktorý je možné pomocou tejto množiny OpenCL API v Imagine odoslať do aplikácie spolu s dátami, na ktorých tento kernelový kód aplikuje výpočet.

Návrhy do budúcnosti, ktoré by bolo zaujímavé použiť v nových verziách aplikácie:

- Preštudovať možnosti OOP v Imagine Logo a naprogramovať dátovú štruktúru, ktorá by bola vhodná pre spracovanie rozsiahlejších dát a umožnila jednoduché spracovanie obrazu
- Naprogramovať odosielanie LGF súborov z aplikácie do Imagine Loga
- Preštudovať možnosti paralelného spracovania 2-3 dimenzionálnych dátových štruktúr

7 Použitá literatúra a prílohy

7.1 Literatúra

[1] Petrovič P. (2007) Program Your NXT Robots with Imagine, Eurologo 2007 [Prístup 4.1.2016].

[2] Imagine Logo Help (nápoveda prostredia) [Prístup 4.1.2016].

[3] Nvidia Cuda tutorial, url: http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf [Prístup 5.1.2016]

[4] Khronos OpenCL documentation, url: https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf [Prístup 9.1.2016]

[5] AMD Developer Central, url: http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/programming-in-opencl/image-convolution-using-opencl/ [Prístup 15.4.2016]

7.2 Prílohy

CD obsahujuce:

- Elektronickú verziu
- Zdrojovy kód aplikácie