Univerzitet u Banjoj Luci
Prirodno-matematički fakultet

Diplomski rad

Tema: Primjena paralelnog programiranja na obradu digitalnih fotografija

Student:
Marko Popović

Mentor:
dr Vladimir Filipović

Sadržaj

1 Uvod

Korištenje grafičke procesorske jedinice za obavljanje zadataka koji nisu grafičke prirode se naziva programiranje grafičke procesorske jedinice u opšte svrhe (u daljem tekstu GPGPU programiranje, što je akronim za „general-purpose GPU“). Prije 2010. godine, GPGPU programiranje se smatrano za novinu u svijetu programiranja sa visokim performansama i nije mu se poklanjalo puno pažnje. To se promjenilo u oktobru te godine kada je superračunar Tianhe-1A, konstruisan od strane kineskog Nacionalnog Centra za Superračunare u Tianđinu, postao najbrži superračunar na svijetu. Ono što je specifično za Tianhe-1A je da je za njegovu konstrukciju korištena kombinacija centralnih procesorskih jedinica (CPU – Central Processing Unit) i grafičkih procesorskih jedinica (GPU – Graphical Processing Unit). Tianhe-1A se sastojao od 14 336 Xeon X5670 centralnih procesorskih jedinica, 2 048 FeiTeng 1000 procesora baziranih na SPARC tehnologiji i sa 7 168 Tesla M2050 grafičkih procesorskih jedinica kompanije Nvidia. Ova hardverska kombinacija je omogućila da se dostigne maksimalna brzina od 2,566 PFLOPS-a što je skoro 50% brže od tada najbržeg Cray-ovog superračunara Jaguar, koji je imao brzinu od 1,759 PFLOPS-a.

U periodu nakon ovog događaja drastično je porasla zainteresovanost za GPGPU programiranje, jer je primjećeno da je GPU zbog svoje specifične konstrukcije pogodan za paralelno programiranje. Kod paralelnog programiranja zadaci za obradu podataka se dodjeljuju višestrukim elementima za obradu i ona se vrši istovremeno, što GPU čini idealnim za tu namjenu jer se sastoji od velikog broja malih procesora. Upotreba paralelnog programiranja će u ovom radu biti demonstrirana kroz tehnologiju pod nazivom OpenCL (Open Computing Language). Najveći razlog zašto je odabran baš OpenCL je taj što ova tehnologija ne zavisi o vrsti uređaja kao ni o njegovom proizvođaču. Naime, OpenCL omogućava paralelno programiranje na centralnim procesorskim jedinicama i grafičkim procesorima svih velikih proizvođača hardvera (AMD, Intel, Nvidia, IBM, ...). OpenCL programi se čak mogu pokrenuti i na nekim novijim Andriod uređajima, kao i na konzolama PlayStation 3 i PlayStation 4 kompanije Sony! Pored ovoga, postoji još nekoliko razloga za odabir ove tehnologije i oni će biti navedeni u slijedećem poglavlju.

S obzirom na dvodimenzionalnu prirodu digitalnih fotografija, implementacija algoritma za njihovu obradu je idealna za demonstraciju paralelnog programiranja, stoga su i odabrani za temu ovog rada. Pored toga, tehnologija OpenCL posjeduje specijalne mehanizme za obradu fotografija čime je implementacija tih algoritama olakšana, i njihova brzina izvršavanja ubrzana. Demonstracija ovih tehnika je implementirana u vidu Windows aplikacije ***micrOCLaw***.

Aplikacija ***micrOCLaw*** je napisana u programskom jeziku C++, pri čemu su za grafički dio aplikacije korištene rutine iz standardnog interfejsa za programiranje aplikacija na operativnom sistemu Windows (Windows API) jer je autor već upoznat sa njima. Iz toga razloga je korisnički interfejs minimalistički, s obzirom da učenje korištenja nekog radnog okvira koji omogućava brže i lakše implementiranje grafičkih elemenata aplikacije (QT, ATL, MFC, ...) nije trivijalan zadatak i oduzima puno vremena. Aplikacija ***micrOCLaw*** omogućava učitavanje digitalne fotografije sa diska korištenjem korisničkog interfejsa, njenu obradu i eventualnu kreaciju nove fotografije sa izmjenama koje je korisnik napravio u odnosu na original. Implementirana je podrška za nekoliko osnovnih formata fotografija. Takođe, korisniku je omogućeno da pogleda listu svih uređaja na mašini koji su kompatibilni sa OpenCL tehnologijom i odabere koji od njih želi da koristi.

Druga glava ovog rada će da uopšteno pretstavi tehnologiju OpenCL. Tu će biti opisan njen nastanak, kao prednosti i manama u odnosu na druge tehnologije koje omogućavaju paralelno programiranje. Treća glava će biti posvećena detaljnom opisu OpenCL aplikacija. Pri tome će se posebno obratiti pažnja na one dijelove OpenCL tehnologije koja će se direktno koristiti za implementaciju aplikacije ***micrOCLaw***. U četvrtoj glavi će biti detaljno objašnjena implementacija aplikacije *micrOCLaw*.

2 OpenCL

OpenCL je tehnologija koja omogućava pisanje programa koji se izvršavaju na heterogenim platformama sastavljenim od centralnih procesorskih jedinica (CPU), grafičkih procesorskih jedinica (GPU), digitalnih signalnih procesora (DSP), FPGA čipova, ... Primjena ove tehnologija omogućava paralelnu obradu u odnosu na podatke (eng: data-based paralelism) kao i paralelnu obradu u odnosu na zadatke (eng: task-based paralelism). Ovi pojmovi, kao i način na koji OpenCL omogućava njihovo korištenje, biće detaljnije objašnjeni u poglavlju 2.2. OpenCL je standard otvorenog tipa održavan od strane neprofitnog tehnološkog konzorcijuma zvanog Khronos Group. Ovaj konzorcijum konstantno unapređuje OpenCL izdavanjem novih verzija OpenCL specifikacije. Najnovija verzija OpenCL specifikacije je 2.0 koja je objavljena u novembru 2013. godine.

2.1 Kako je nastao OpenCL

Nakon što su mnogi inžinjeri i akademici došli do zaključka da su sistemi koji kombinuju CPU i GPU pretstavljaju budućnost programiranja sa visokim performansama, u startu su bili suočeni sa značajnom preprekom. Veliki problem je što nije bilo načina da se iskoritsti potencijal ovih sistema jer programi pisani u tradicionalnim programskim jezicima mogu da koriste samo CPU. Čak i programi koji koriste tehnologiju CUDA (Compute Unified Device Architecture) kompanije Nvidia, ne mogu da se koriste na gore pomenuti način jer za izvrsavanje koriste isključivo GPU. Pored toga, CUDA programi mogu da rade samo na grafičkim procesorima koje proizvodi samo Nvidia. Razlog za ove probleme leži u činjenici da ne postoji dominantna hardverska arhitektura u svijetu grafičkih procesora i procesora visokih performansi, kao što je to dugo vremena bila arhitektura x86 u svijetu personalnih računara. Uprkos njihovoj zajedničkoj svrsi, nema puno sličnosti između Maxwell procesora kompanije Nvidia, Volcanic Islands procesora iz AMD-a i Cell Broadband Engine procesora iz IBM-a. Svaki od ovih uređaja ima različit skup instrukcija, i da bi napisao program koji bi se izvršavao na sve tri uređaja, programer bi morao da nauči tri različita jezika.

Rješenje ovoga problema se pojavilo u vidu tehnologije OpenCL (Open Computing Language), čije rutine se mogu izvršavati na uređajima svih velikih proizvođača hardvera. OpenCL je nastao na inicijativu kompanije Apple koja je željela da se stvori zajednički interfejs za sve uređaje, jer Apple ne proizvodi svoje procesore već za svaku slijedeću liniju proizvoda bira ćiji će hardver koristiti. S obzirom na dominantnu poziciju ove kompanije, proizvođači hardvera su odlučili da ispune taj zahtjev.

U avgustu 2008. godine je formirana radna grupa za OpenCL u okviru Khronos Group, konzorcijuma kompanija čiji cilj je napredak grafike i grafičkih medija. Radna grupa je u decembru iste godine objavila prvu OpenCL specifikaciju (verzija 1.0), a slijedeće godine kompanije Nvidia i AMD su objavile alate (SDK) za razvoj OpenCL programa. Specifikacija definiše funkcije i strukture podataka u OpenCL, kao i mogućnosti koje treba da pružaju razvojni alati obezbjeđeni od strane proizvođača hardvera. Pored toga, ona određuje kriterijum koji uređaji moraju da zadovolje da bi se mogli smatrati kompatibilnim sa OpenCL tehnologijom. Pored onih koje zahtjeva OpenCL standard, postoje dodatne mogućnosti za OpenCL aplikacije koje dolaze u vidu ekstenzija. Ekstenzije mogu biti povezane sa softverskim paketom proizvođača (koji se još zove i platforma) ili sa konkretnim uređajem. Trenutno aktualna verzija OpenCL specifikacije je 2.0 ali u ovom radu će se koristiti samo mogućnosti sadržane u prethodnoj OpenCL specifikaciji, verzije 1.2. Razlog za to je što samo najnoviji hardver podrzava OpenCL 2.0.

2.2 Prednosti OpenCL tehnologije

OpenCL standard definiše skup tipova podataka, funkcija i struktura podataka koje proširuju programske jezike C i C++. S vremenom su kreirani portovi za druge programske jezike (C#, Java, Python, ...) ali standard zahtjeva da se obezbijede samo biblioteke za C i C++. Pitanje koje se nameće je šta OpenCL može a da to ne mogu jezici C i C++. Iako je odgovor na to pitanje prilično komplikovan, tri glavne prednosti koje se izdvajaju su:

1. Portabilnost

2. Standardizovana obrada vektorskih struktura podataka

3. Paralelno programiranje

2.2.1 Portabilnost

Filozofija OpenCL-a je slična filozofiji programskog jezika Java, čiji kod se bez izmjena može izvršavati na svakom operativnom sistemu koji podržava Java virtualnu mašinu (JVM). Stoga se za Java kod kaže „Napiši jednom, pokreni svuda“. Razlika sa OpenCL-om je ta što se OpenCL kod može bez izmjena izvršiti na različitim uređajima pa bi se za OpenCL moglo reći da mu je filozofija „Napiši jednom, pokreni na bilo čemu“.

Svaki proizvođač koji proizvodi uređaje kompatibilne sa OpenCL-om takođe obezbijeđuje alatke koje kompiliraju OpenCL kod za izvršavanje na tim uređajima. Ovo znači da se da se OpenCL kod može napisati jednom i kasnije kompilirati za svaki kompatibilan uređaj, nezavisno od toga da li je to grafička precesorska jedinica ili centralna procesorska jedinica sa više jezgara. Ovo je velika prednost u odnosu na standardno programiranje sa visokim performansama, kod kojega se mora naučiti programski jezik specifičan za proizvođača da bi se mogao koristiti uređaj tog proizvođača.

Pored prethodno pomenute prednosti, OpenCL kod se može istovremeno izvršavati na više različitih uređaja, pri čemu ti uređaji ne moraju da imaju istu hardversku arhitekturu i mogu da čak imaju i različite proizvođače. Jedini uslov koji treba da bude zadovoljen je da svi ti uređaji podržavaju OpenCL. Ovo je nemoguće sa klasičnim C/C++ programiranjem, kod kojega se izvršna datoteka može izvršavati samo na jednom uređaju. Ovo konkretno znači da je moguce napisati aplikaciju koja može da se istovremeno izvršava na AMD-ovom procesoru sa više jezgara, grafičkoj kartici od Nvidia i IBM-ovom akceleratoru.

2.2.2 Standardizovana obrada vektorskih podataka

U ovom poglavlju ćemo pod terminom „vektor“ podrazumijevati strukturu podataka koja sadrži više elemenata istog tipa podataka. U toku vektorske operacije, na svakom elementu (komponenti) vektora se ta operacija izvršava u istom ciklusu sata. Izvršavanje operacije na više vrijednosti istovremeno je inače osobina superskalarnih (vektorskih) procesora koji se koriste kada su neophodne visoke performanse.

Većina modernih procesora podržava vektorsku obradu podataka, ali ANSI C/C++ standard uopšte ne definiše ni osnovne vektorske tipove podataka. Ovo je posljedica činjenice da su vektorske instrukcije uglavnom specifične za proizvođača hardvera. Na primjer, Intel-ovi procesori koriste SSE ekstenzije, uređaji koje proizvodi Nvidia zahtjevaju

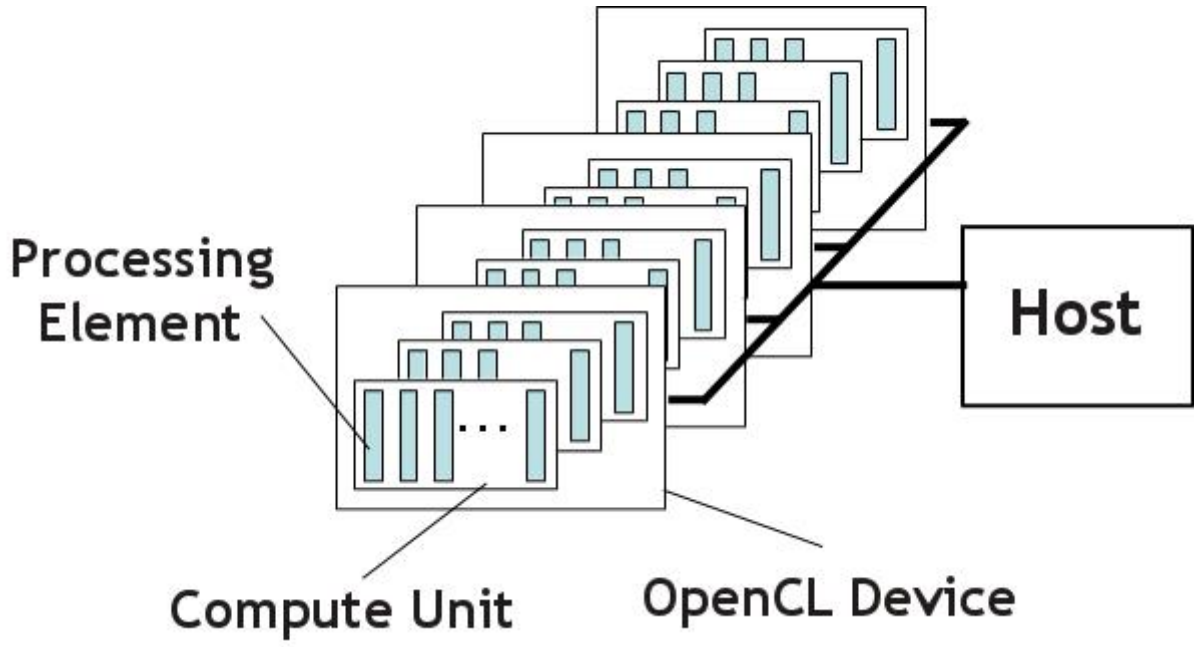
PTX instrukcije, uređaji iz IBM-a za vektorsku obradu koriste isključivo AltiVec instrukcije, a ovi skupovi instrukcija nemaju ništa zajedničko.

Prednost OpenCL-a je što će se pri kompilaciji aplikacije koje vrše vektorsku obradu generisati odgovarajuće vektorske instrukcije za svaki uređaj. Dakle, ako je cilj da se napravi aplikacija sa viskim performansama koja će biti dostupna na više platformi, pisanje te aplikacije koristeći OpenCL će dovesti do velike uštede vremena.

2.2.3 Paralelno programiranje

Kada se koristi paralelno programiranje, tada se zadaci za obradu podataka dodjeljuju višestrukim elementima za obradu i ona se vrši istovremeno. U OpenCL terminologiji, ovi zadaci se nazivaju jezgra (eng: kernel). Jezgro je funkcija napisana na određen način i ona je predviđena da se izvrši na jednom ili više OpenCL kompatibilnih uređaja. Jezgra se šalju na uređaje na kojima će se izvršavati putem kontrolne aplikacije (eng: host application). Kontrolna aplikacija je standardna C/C++ aplikacija koja se izvršava na korisnikovoj centralnoj procesorskoj jedinici.

Kontrolne aplikacije upravljaju OpenCL uređajima koristeći poseban kontejner zvan kontekst. Da bi stvorila jezgro, kontrolna aplikacija prvo bira jednu od funkcija koje se nalaze u kontejneru zvanom program. Nakon toga, ona stvara asocijaciju između jezgra i podataka koji će biti korišteni kao argumenti, a potom se jezgro proslijeđuje na komandni red. Komandni red je mehanizam putem kojeg kontrolna aplikacija šalje uređajima jezgra na izvršavanje. Uređaj tada uzima jezgro iz reda i izvršava ga koristeći odgovarajuće argumente.



Slika 1.1 **Model OpenCL aplikacije**

OpenCL aplikacija može konfigurisati različite uređaje da vrše različite zadatke obrade podataka, i pri tome se te obrade mogu vršiti na različitim podacima. Drugim riječima, OpenCL omogućava totalni paralelizam po zadacima. Ovo je bitna prednost u odnosu na druge alate za paralelno programiranje, koji podržavaju samo paralelizam u

odnosu na podatke. Većina OpenCL kompatibilnih uređaja se sastoji od više jedinica za računanje, sto znači da unutar svakog uređaja postoji dodatni nivo paralelizma. Figura 1.1 pretstavlja skicu modela OpenCL platforme.

2.2.4 Ekstenzije

Specifikacija OpenCL tehnologije, kreirana od strane konzorcijuma Khronos, precizira zahtjeve koje moraju da ispunjavaju uređaji i razvojni alati da bi se smatrali kompatibilnim sa OpenCL tehnologijom. Pored zahtjevanih, uređaji i razvojni alati mogu da imaju i neke dodatne sposobnosti koje se nazivaju ekstenzije. One mogu biti platformske (vezane za razvojni softverski paket) ili ekstenzije koje su vezane za konkretan OpenCL kompatibilan uređaj. OpenCL program može na početku izvršavanja da ispita postojanje određene ekstenzije i iskoristi je ako je prisutna.

Postoji konvencija za nazive ekstenzija. Ako je određena ekstenzije odobrena od strane konzorcijuma Khronos, tada njen naziv ima prefiks *cl_khr_*, dok se u slučaju ekstenzija koje nisu ispitane i odobrene dodaje prefiks *cl_<naziv_kompanije>_*. Iako ekstenzije nisu u sklopu OpenCL standarda, njihovo postojanje pretstavlja određenu vrstu bonusa jer se njihovim korištenjem mogu dodatno poboljšati performanse OpenCL programa. U narednoj glavi ovog rada će biti više riječi o ekstenzijama, kao i o konkretnim primjerima istih.

2.3 Radni okviri

Da bi se izvršila kompilacija OpenCl aplikacije mora se pristupiti alatima u radnom okviru za OpenCL. Po specifikaciji, radni okvir za OpenCl se sastoji od tri dijela:

1. Platformski sloj – omogućava pristup uređajima i formiranje konteksta.

2. Izvršni sloj – omogućava kontrolnim aplikacijama da kreiraju komande redove za uređaje unutar konteksta i da šalju jezgra na te redove.

3. Kompilator – kompilira programe koji sadrže jezgra za izvršavanje.

Radna grupa za OpenCL ne kreira svoj radni okvir. Radne okvire izdaju proizvođači hardvera u okviru svojih alata za razvoj softvera. Sve alatke za razvoj su besplatne i mogu se preuzeti sa sajtova kompanija koje proizvode hardver kompatibilan sa tehnologijom OpenCL.

3 Struktura OpenCL aplikacija

Cilj ove glave je da detaljno pretstavi faktore koji zajedno šine jednu OpenCL aplikaciju. Prvi dio će da pretstavi kontrolnu aplikaciju, što je dio OpenCL aplikacije koja se izvršava na kontrolnom uređaju, nakon čega će biti riječi o rutinama koje se izvršavaju na OpenCL kompatibilnim uređajima. S obzirom da detaljan opis svih mogućnosti OpenCL aplikacija prevazilazi okvir jednog diplomskog rada, detaljan opis će biti dat samo za one dijelove OpenCL tehnologije koji su direktno korišteni u pisanju aplikacije ***micrOCLaw.***

3.1 Programiranje kontrolne aplikacije

Kao što smo već rekli, kontrolna aplikacija je standardna aplikacija napisana u programskom jeziku C ili u programskom jeziku C++. Da bi se napisala kontrolna aplikacija, potrebno je poznavanje šest osnovnih struktura podataka pomoću kojih kontrolna aplikacija kreira jezgra i memorijske spremnike, kompilira programe za odgovarajuće uređaje u kontekstu, salje jezgra uređajima preko komandih redova, ... Nije lako razumjeti ove strukture pri prvom susretu sa njima ali nakon što se prevaziđe ta prepreka, pisanje kontrolnih aplikacija postaje relativno jednostavno. Iz toga razloga, ove strukture će biti detaljnije opisane pri čemu će se koristiti dijelovi izvornog koda (koji pretstavljaju ove strukture) u programskoj jeziku C++.

Programski jezik C++ je korišten iz više razloga. Kao dio OpenCL standarda je kreiran omotač za razvoj u jeziku C++ koji ne stvara dodatna usporenja programa. Omotač se oslanja na vektorske strukture podataka pa nema potrebe za dinamičkom alokacijom nizova za OpenCL strukture. Pored toga, programski jezik C++ ima jednostavnije pozive funkcija i omogućava automatsko oslobađanje svih resursa zauzetih od strane OpenCL struktura. Konačno, jezik C++ omogućava napredne tehnike rukovanja sa greškama u programu korištenjem C++ izuzetaka. Podrazumjevana podešavanja ne uključuju korištenje izuzetaka ali njihova upotreba se moze lako uključiti dodavanje preprocesorske direktive `__CL_ENABLE_EXCEPTIONS`. Navedene prednosti omogućavaju brže i sažetije pisanje kontrolnih aplikacija korištenjem klasa omotača za sve OpenCL strukture podataka, pa je iz tih razloga i kontrolni dio aplikacije micrOCLaw napisan korištenjem C++ omotača.

3.1.1 Fundamentalne strukture podataka

Prva od ovih struktura koju ćemo da pomenemo je platforma. Platforma pretstavlja konkretnu implementaciju OpenCL standarda od strane jednog proizvođača uređaja. Pomoću nje se u toku izvršavanja programa može ispitati koliko ima uređaja kompatibilnih sa OpenCL tehnologijom da bi se obrada podataka pravilno rasporedila. Pored toga, ona omogućava pristup bitnim informacijama o konkretnoj platformi. Najbitnija od njih verzija OpenCL standarda koju je ta platforma implementirala, da bi program mogao da provjeri kompatiblnost sa uređajem. Program koji koristi mogućnosti precizirane verzijom 1.2 OpenCL standarda se neće moći korektno izvršiti na uređaju koji pripada platformi koja implementira nižu verziju standarda. Pored toga, platforma omogućava pristup informacijama o profilu platforme kao i o OpenCL ekstenzijama koje ta platforma posjeduje. Odgovarajuća klasa omotač za ovu strukturu je `cl::Platform`, i ona posjeduje metode koje omogućavaju prethodno opisane mogućnosti.

Kroz platformu određenog proizvođača kontrolna aplikacija može da pristupi svim OpenCL kompatibilnim uređajima te platforme, što je omogućeno metodom klase `cl::Platform` koja ima slijedeći prototip:

```
cl_int cl::Platform::getDevices(cl_device_type type,
    VECTOR_CLASS<cl::Device> *devices)
```

Pri tome, prvi parametar je konstanta koja određuje tip uređaja kojem želimo da pristupimo i čije vrijednosti i njihovo značenje su pretstavljeni u tabeli 3.1. Drugi parametar pretstavlja pokazivač na vektor koji sadrži elemente tipa `cl::Device`, što je klasa omotač za drugu fundamentalnu OpenCL strukturu koju nazivamo uređaj.

Tip uređaja	Značenje
`CL_DEVICE_TYPE_ALL`	Identifikuje sve uređaje vezane za date platforme
`CL_DEVICE_TYPE_DEFAULT`	Identifikuje podrazumjevane uređaje date platforme
`CL_DEVICE_TYPE_CPU`	Identifikuje kontrolni uređaj (CPU računara)
`CL_DEVICE_TYPE_GPU`	Identifikuje uređaje koji posjeduju grafičku procesorsku jedinicu
`CL_DEVICE_TYPE_ACCELERATOR`	Identifikuje eksterni uređaj koji se koristi za ubrzanje obrade podataka

Tabela 3.1 **Tipovi OpenCL uređaja**

Uređaji u OpenCL aplikaciji dobijaju zadatke koje trebaju da obave nad određenim podacima, i kontrolni uređaj je zadužen da proslijedi te parametre OpenCL uređajima. Preko klase omotača za ovu strukturu se može pristupiti raznim informacijama vezanim za konkretan uređaj, koristeći preopterećenu metodu pod nazivom `getInfo` kojoj je proslijeđena odgovarajuća konstanta sa prefiksom `CL_DEVICE`. Za primjer možemo uzeti `CL_DEVICE_IMAGE_SUPPORT`, čijim se korištenjem ispituje da li uređaj podržava napredne mogućnosti obrade obrade fotografija koristeći posebne strukture podataka koje se smještaju u posebne memorijske spremnike što doprinosi ubrzanju obrade podataka. Postoji više od pedeset različitih informacija vezanih za uređaj koje možemo dobiti na ovaj način, a zbog njihovog velikog broja navešćemo samo nekoliko: tip uređaja, podržana verzija OpenCL standarda, broj jedinica za obradu podataka, maksimalna dimenzionalnost podataka koji uređaj može da obrađuje, veličina memorije koju uređaj posjeduje, OpenCL ekstenzije koje uređaj podržava, itd.

Slijedeća fundamentalna struktura podataka je kontekst. Kontekst identifikuje skup uređaja odabranih da zajedno obrađuju podatke, pri čemu postoji ograničenje da jedan kontekst ne može da sadrži uređaje koji pripadaju različitim platformama. Kontekst omogućava kreiranje komandnog reda preko kojeg kontrolna aplikacija šalje uređajima podatke za obradu. Pored toga, u okviru konteksta se kreiraju memorijski spremnici iz čega i proizilazi prethodno pomenuto ograničenje jer uređaji koji pripadaju različitim platformama ne mogu pristupati istim memorijskim resursima. Klasa omotač za ovu strukturu ima tri veoma korisna konstruktora. Njihovi prototipi su prilično komplikovani ali svi argumenti osim prvog imaju podrazumjevane vrijednosti i često se mogu izostaviti jer se koriste samo u slučaju naprednih metoda kreiranja konteksta. Iz tog razloga ćemo prikazati njihove prototipe samo sa prvim argumentom:

```
Context(const Device &device);
Context(const VECTOR_CLASS<Device> &device);
Context(cl_device_type type);
```

Kao što se da i naslutiti, prvi konstruktor kreira konteks samo za određeni uređaj dok drugi kreira konteks za sve uređaje sadržane u datom vektoru. Treći konstruktor je koristan u slučaju kada se želi izbjeći direktno pristupanje platformama i uređajima jer on interno pristupa prvoj platformi koja sadrži uređaj datog tipa i kreira kontekst koji sadrži sve uređaje tog tipa. Klasa `cl::Context` takođe posjeduje preopterećenu metodu `getInfo` pomoću koje se može pristupiti informacijama vezanim za određenu instancu ove klase.

Slijedeće dvije strukture koje ćemo pretstaviti, program i jezgro, imaju običaj da zbune većinu ljudi koji se prvi put susreću sa OpenCL aplikacijama jer i jedna i druga sadrže izvorni kod koji će se izvršavati na uređajima. Razlika je u tome što jezgro pretstavlja jednu funkciju koja se može izvrsiti na uređaju dok je program kontejner koji sadrži više različitih jezgara. OpenCL aplikacije koriste dva pristupa za kreiranje programa. Prvi je čitanjem izvornog koda iz odvojenog tekstualnog fajla sa ekstenzijom „cl“, dok je drugi čuvanjem izvornog koda kao niza karaktera direktno u kodu kontrolne aplikacije. Iako prvi pristup omogućava jasno razdvajanje koda kontrolne aplikacije i koda koji se izvršava na uređaju, aplikacija micrOCLaw je napisana korištenjem drugog pristupa jer je dosta jednostavniji. Klasa omotač za strukturu program (`cl::Program`) ima konstruktore za kreiranje programa od izvornog koda i od već kompiliranog binarnog koda. Takođe je važno napomenuti da se programi kreiraju u okviru konteksta, pa se stoga referenca na odgovarajući kontekst proslijeđuje konstruktoru klase `cl::Program`.

S obzirom da se jezgra sadržana u programu oslanjaju na funkcije i strukture podataka specifične za OpenCL, svaki program mora da se kompilira koristeći kompilator iz OpenCL radnog okvira. OpenCL standard ne potstavlja puno zahtjeva kompilatorima ali jedna stavka je kritična: svakom kompilatoru se u okviru kontrolne aplikacije mora moći pristupiti putem funkcije `clBuildProgram`. Kada se koristi C++ omotač, tada se umjesto ove funkcije koristi metoda build klase `cl::Program` koja interno koristi funkciju `clBuildProgram` i ima slijedeći prototip:

```
cl_int cl::Program::build(const VECTOR_CLASS<cl::Device> &devices,
    const char * options = NULL,
    (CL_CALLBACK * pfn_notify)(cl_program,  void * user_data) = NULL,
    void * data = NULL);
```

Preko prvog parametra se proslijeđuje vektor koji kao elemente sadrži uređaje iz konteksta (u čijem okviru je kreiran program) za koje želimo da kompiliramo program. Pri tome vrijedi konvencija da se u slučaju prodslijeđivanja praznog vektora vrši kompilacija za sve uređaje koji pripadaju odgovarajućem kontekstu. Pomoću drugog parametra se proslijeđuju opcije za kompilaciju i njihova kompletna lista se može pronaći u tabeli 3.2. Treći i četvrti argument su pokazivač na tzv. povratnu funkciju i pokazivač na podatke koje korisnik može da joj proslijedi. Ako se proslijede ovi argumenti, tada će se pomenuta povratna funkcija asinhrono izvršiti tačno u trenutku kada se završi kompilacija. Ovo može da bude veoma korisno da se signalizira kontrolnoj aplikaciji da je kompilacija završena, s obzirom da proces kompilacije može da potraje u slučajevim kada je pokrenut za veliki broj uređaja. U suprotnom, poziv metode `build` će da blokira nit u kojoj se izvršava kontrolna aplikacija dok se ne završi proces kompilacije.

Parametar	Svrha
`-D NAME`	Definiše makro `NAME` sa vrijednošću 1
`-D NAME=VALUE`	Definiše makro `NAME` sa vrijednošću `VALUE`
`-I DIR`	Identifikuje direktorij `DIR` koji sadrži zaglavlja
`-cl-single-precision-constant`	Tretira konstante sa dvostrukom preciznošću kao konstante sa jednostrukom preciznošću
`-cl-denorms-are-zero`	Tretira sve brojeve manje od najmanjeg reprezentabilnog broja kao 0
`-cl-opt-disable`	Isključuje sve vrste optimizacija
`-cl-strict-aliasing`	Zahtjeva stroga pravila pri korištenju pseudonima
`-cl-mad-enable`	Dozvoljava korištenje atomske operacije MAD koja računa `a * b + c` brže ali sa smanjenom preciznošću
`-cl-no-signed-zeros`	Onemogućuje korištenje pozitinih i negativnih nula definisanih standardom IEEE-754
`-cl-unsafe-math-optimizations`	Optimizuje kalkulacije izbacivanjem provjere grešaka
`-cl-finite-math-only`	Podrazumjeva da su svi rezultati i argumenti konačni. Nijedna operacija neće prihvatiti niti će proizvesti kao rezultat beskonačne ili NaN vrijednosti
`-cl-fast-relaxed-math`	Kombinuje prethodne dvije opcije
`-w`	Isključuje generisanje upozorenja za vrijeme kompilacije
`-Werror`	Zahtjeva od kompilatora da sva upozorenja tretira kao greške

Tabela 3.2 **Opcije za kompilaciju programa**

Kao i prethodno navedene klase omotači, i `cl::Program` ima preopterećenu metodu `getInfo` pomoću koje se mogu dobiti informacije o konkretnoj instanci te klase. Kontekst u okviru kojeg je program stvoren, uređaji za koje je vršena kompilacija i njihov broj su samo neke od njih. Pored toga, ova klasa ima još dvije važne metode. Prva od njih, metoda `getBuildInfo`, pruža informacije o statusu kompilacije kao i što su opcije korištene pri kompilaciji i opis upozorenja i grešaka (ako ih ima). Druga metoda, `createKernels`, se koristi da za pojedinačne funkcije iz programa stvori odgovarajuća jezgra, što je slijedeća OpenCL struktura koju ćemo posmatrati.

Jezgro pretstavlja jednu funkciju koja se može izvršiti direktno na uređaju i ona je pretstavljena klasom omotačem `cl::Kernel`. Da bi se jezgro moglo izvršiti, prethodno je potrebno odrediti koji memorijski spremnici će se koristiti kao argumenti jezgra. Ovo se u praksi postiže korištenjem odgovarajuće metode klase omotača, i njen prototip je slijedeći:

```
cl_int cl::Kernel::setArg(cl_uint index, T value);
```

Prvi argument ove metode pretstavlja redni broj argumenta jezgra, dok drugi pretstavlja memorijski objekat koji će biti korišten kao argument. Tačan tip podatka drugog parametra se određuje za vrijeme kompilacije i on može da uzme jednu od slijedećih vrijednosti:

1. `cl::Memory` - Memorijski objekat koji će biti proslijeđen jezgru preko reference.
2. `cl::Sampler` - Objekat uzornik koji se koristi pri obradi fotografija.
3. `cl::LocalSpaceArg` - Argument koji se smješta u tzv. lokalnu memorijsku lokaciju na uređaju.
4. Primitivni tip podatka koji će biti proslijeđen uređaju po vrijednosti.

S obzirom da bi objašnjavanje ovih tipova pretstavljalo krupnu digresiju, odložiti ćemo to za slijedeće poglavlje u kojem će biti riječi o memorijskim objektima. Kao i prethodno pomenute klase omotači, i klasa `cl::Kernel` ima preopterećenu metodu `getInfo` pomoću koje se može pristupiti informacijama vezanim za jezgro (naziv jezgra, broj argumenta jezgra, program koji sadrži jezgro, ...). Nakon što je jezgro kreirano i njegovi argumenti određeni, ono se može poslati na izvršavanje bilo kojem uređaju koji se nalazi u kontekstu koji sadrži program objekat toga jezgra. Ovo se vrši korištenjem komandnog reda, što je poslednja OpenCL struktura koju ćemo opisati u ovom poglavlju.

Komandni red je struktura pomoću koje kontrolna aplikacija šalje komande uređajima. Komanda je poruka kojom kontrolna aplikacija signaliziraja uređaju da izvrši određenu operaciju. Pored već navedene komande za izvršavanje jezgra, ova struktura može da šalje komande za transfer podataka između uređaja i kontrolne aplikacije, kao i između dva uređaja. Da bi mogli da šaljemo komande uređajima u okviru konteksta, moramo da za svaki od njih kreiramo komandni red. Podrazumjevano ponašanje komandnog reda je da se komande koje on šalje izvršavaju u redoslijedu u kojem su poslate ali ovo se može promjeniti ako za to ima potreba.

Odgovarajuća klasa omotač za komandi red je `cl::CommandQueue`. Uređaj kojem će objekat ove klase slati komande se određuje pri njegovoj kreaciji koja se vrši konstruktorom sa slijedećim prototipom:

```
CommandQueue(const cl::Context &context, const cl::Device &device,
    cl_command_queue_properties = 0, cl_int *err = NULL);
```

Uz pomoć prva dva argumenta se identifikuju kontekst i uređaj iz tog konteksta kojem će se slati komande. Treći argument određuje osobine komandnog reda. U verziji 1.2 OpenCL standarda on pored podrazumjevane vrijednosti može da uzme samo još dvije. To su `CL_QUEUE_PROFILING_ENABLE`, kojom se uključuje profiliranje izvršavanja koda i `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` pomoću koje se uključuje već pomenuta mogućnost izvršavanja komandi u redoslijedu različitom od njihovog izdavanja. Dodatne mogućnosti komandih redova će biti uvedene u verziji 2.0 OpenCL standarda. Četvrti argument se može koristiti za ispitivanje eventualne greške pri kreaciji objekta. Pored već poznate metode `getInfo` putem koje se dobijaju informacije o konkretnom objektu, klasa `cl::CommandQueue` posjeduje veliki broj metoda koje se koriste za transfer podataka i koje su pretstavljene u tabeli 3.3. Njihovi kompletni prototipi su izostavljeni jer su prilično komplikovani (imaju i do deset argumenata) i oslanjaju se na memorijske objekte o kojima će biti riječ tek u slijedećem poglavlju. Pored toga, većina njih se uopšte neće koristiti u aplikaciji micrOCLaw pa ćemo prostor potreban da bi se objasnilo njihovo

Naziv metode	Opis metode
`enqueueReadBuffer`	Šalje komandu za čitanje iz memorijskog spremnika u memoriju kontrolne aplikacije
`enqueueWriteBuffer`	Šalje komandu za pisanje iz memorije kontrolne aplikacije u memorijski spremnik
`enqueueReadBufferRect`	Šalje komandu za čitanje pravougaone oblasti iz memorijskog spremnika u memoriju kontrolne aplikacije
`enqueueWriteBufferRect`	Šalje komandu za pisanje pravougaone oblasti iz memorije kontrolne aplikacije u memorijski spremnik
`enqueueCopyBuffer`	Šalje komandu za kopiranje sadržaja jednog memorijskog spremnika u drugi memorijski spremnik
`enqueueCopyBufferRect`	Šalje komandu za kopiranje pravougaone oblasti iz jednog memorijskog spremnika u drugi memorijski spremnik
`enqueueReadImage`	Šalje komandu za čitanje iz memorijskog objekta tipa `cl::Image` u memoriju kontrolne aplikacije
`enqueueWriteImage`	Šalje komandu za pisanje iz memorije kontrolne aplikacije u memorijski objekat tipa `cl::Image`
`enqueueCopyImage`	Šalje komandu za kopiranje sadržaja između dva memorijska objekta tipa `cl::Image`
`enqueueCopyImageToBuffer`	Šalje komandu za kopiranje sadržaja memorijskog objekta tipa `cl::Image` u memorijski spremnik
`enqueueCopyBufferToImage`	Šalje komandu za kopiranje sadržaja memorijskog spremnika u memorijski objekat tipa `cl::Image`
`enqueueMapBuffer`	Šalje komandu za mapiranje sadržaja oblasti memorijskog spremnika na adresni prostor kontrolne aplikacije i vraća pokazivač na tu oblast memorije
`enqueueMapImage`	Šalje komandu za mapiranje sadržaja oblasti u memorijskom objektu tipa `cl::Image` na adresni prostor kontrolne aplikacije i vraća pokazivač na tu oblast memorije
`enqueueUnmapMemObject`	Šalje komandu za poništavanje mapiranja sadržaja memorijskog objekta

Tabela 3.3 Metode za transfer podataka klase `cl::CommandQueue`

korištenje iskoristiti da bi bolje pojasnili neke druge principe OpenCL programiranja. One metode koje se budu koristile u aplikaciji micrOCLaw će biti objašnjene u nekom od narednih poglavlja. Na kraju ovog poglavlja, napomenimo i metode klase `cl::CommandQueue` koje šalju jezgra na izvršavanje. Prva od njih, pod nazivom `enqueueTask`, šalje jezgro na izvršavanje pri kojem će uređaj korisiti samo jednu jedinicu za računanje, što znači da obrada podataka neće biti efikasna. Iz tog razloga nećemo detaljnije razmatrati ovu metodu. Druga metoda koja omogućava korištenje svih jedinica za računanje na uređaju i pri tome obezbjeđuje paralelizam u odnosu na zadatke, se naziva `enqueueNDRangeKernel`. Razumjevanje ove metode je ključno za razumjevanje kako se u OpenCL programiranju vrši paralelna obrada podataka, stoga ćemo joj posvetiti

čitavo jedno poglavlje koje dolazi nakon poglavlja o memorijskim objektima. Ovim smo završili pregled fundamentalnih struktura u OpenCL programiranju.

3.1.2 Memorijski objekti

Kada želimo da vršimo obradu nekih podataka u standardnoj C/C++ aplikaciji, mi jednostavno podesimo ulazne i izlazne parametre funkcije koja će vršiti tu obradu. Nešto slično treba da uradimo i za OpenCL jezgro, međutim to je ipak komplikovanije jer se jezgro izvršava ne nekom drugom uređaju. U prethodnom poglavlju smo već pomenuli da se ovo vrši korištenjem metode `setArg` klase `cl::Kernel`, koja kao argument uzima memorijski objekat ili primitivni tip podatka (npr. `float`) koji će biti proslijeđen po vrijednosti. U slučaju primitivnog tipa jednostavno ga proslijedimo zajedno sa njegovim indeksom. Kada su u pitanju memorijski objekti, moramo prvo da izvršimo njihovu alokaciju. Memorijski objekti su pretstavljeni klasom omotačem `cl::Memory` iz koje su izvedene klase `cl::Buffer` i `cl::Image`.

Klasa `cl::Buffer` pretstavlja memorijske spremnike opšte namjene i kreiranje objekta te klase se vrši pomoću slijedećeg konstruktora:

```
cl::Buffer::Buffer(const cl::Context &context, cl_mem_flags flags,
    size_t size, void *host_ptr = NULL, cl_int *err = NULL);
```

Kao što se može i pretpostaviti, prvi argument određuje kontekst u čijem će okviru ovaj memorijski spremnik biti kreiran. Treći argument određuje veličinu memorijskog objekta izraženu u bajtovima dok je četvrti argument pokazivač na memorijsku lokaciju unutar kontrolne aplikacije koje će se koristit pri kreaciji memorijskog objekta. Drugi argument je polje bitova koje može da uzme kombinaciju vrijednosti iz tabele 3.4 i njime se kontroliše nivo pristupa uređaja tom objektu i način alociranja memorije za taj spremnik unutar kontrolne aplikacije. Značenje prve tri vrijednosti iz ove tabele je očigledno dok nedostatak razumjevanja ostale tri može da stvori greške u programu koje se teško pronalaze. Naime, pokazivač `host_ptr` može da bude `NULL` u slučaju kreiranja memorijskog objekta koji će biti korišten za primanje rezultata izvršavanja nekog jezgra. Tada se smatra dobrom praksom da se koristi opcija `CL_MEM_WRITE_ONLY` i u tom slučaju se za taj objekat memorija alocira samo na uređaju. Međutim, kada treba da se vrši prenos podataka iz kontrolne aplikacije ka uređaju `host_ptr` mora da pokazuje na memorijsku lokaciju koja sadrži podatke koje treba prenijeti. Tada se u slučaju korištenja opcije `CL_MEM_USE_HOST_PTR` povećava memorijska efikasnost aplikacije jer se ne alocira nova oblast memorije za memorijski objekat već se direktno korist memorija na koju pokazuje `host_ptr`. Sa druge strane, ovo pretstavlja problem u slučaju da želimo da pristupimo toj memorijskoj lokaciji nakon što počne transfer podataka ili ako smo kreirali više memorijskih objekata koristeći istu memorijsku lokaciju (npr. želimo da nekoliko uređaja obrađuje podatke sa te lokacije). Da bi se izbjegli ovi problemi, u zavisnosti od potreba, koristi se jedna od dvije preostale opcije.

Vrijednost	Značenje
`CL_MEM_READ_WRITE`	Uređaju je dozvoljeno čitanje i pisanje u memorijski objekat
`CL_MEM_WRITE_ONLY`	Uređaju je dozvoljeno samo pisanje u memorijski objekat
`CL_MEM_READ_ONLY`	Uređaju je dozvoljeno samo čitanje iz memorijskog objekta

CL_MEM_USE_HOST_PTR	Memorijski objekat će koristiti memorijsku oblast preciranu putem pokazivača iz kontrolne aplikacije (host_ptr)
CL_MEM_ALLOC_HOST_PTR	Memorijski objekat će koristiti memorijsku oblast alociranu u okviru memorije kontrolne aplikacije
CL_MEM_COPY_HOST_PTR	Memorijski objekat će koristiti memorijsku oblast alociranu u okviru memorije kontrolne aplikacije i sadržaj memorije na koju pokazuje argument host_ptr će biti kopiran u tu oblast

Tabela 3.4 Opcije za kreiranje memorijskih objekata

Klasa cl::Image je namjenjena za memorijske objekte koji sadrže podatke grafičkog tipa kao što su vrijednosti piksela u fotografiji ili podaci o teksturama. Iz ove klase su izvedene dvije klase, klasa cl::Image2D koja pretstavlja dvodimenzionalne podatke i klasa cl::Image3D koja pretstavlja trodimenzionalne podatke. Klasa cl::Image3D se koristi u komplikovanim programima koji obrađuju velike količine grafičkih podataka (npr. niz frejmova u video zapisu) pa stoga nećemo objašnjavati kako se koristi. Objekti klase cl::Image2D se kreiraju pomoću slijedećeg konstruktora:

```
cl::Image2D::Image2D(cl::Context &context, cl_mem_flags flags,
    cl::ImageFormat format, size_t width, size_t height,
    size_t row_pitch = 0, void *host_ptr = NULL, cl_int *err = NULL);
```

pri čemu argumenti context, flags, host_ptr i err imaju isto značenje kao i kod konstruktora klase cl::Buffer. Argument width , odnosno height, se koristi za određivanje širine, odnosno visine, fotografije. Šesti argument, row_pitch, označava veličinu jednog reda piksela u fotografiji izraženu u bitovima. Najčešće se koristi njegova podrazumjevana vrijednost čijim se proslijeđivanjem ostavlja konstruktoru da je sam izračuna koristeći širinu fotografije i primitivni tip korišten za pretstavljanje grafičkih podataka. Pomoću argumenta format se precizira priroda grafičkih podataka proslijeđivanjem objekta klase cl::ImageFormat koji se kreira koristeći

```
cl::ImageFormat(cl_channel_order order, cl_channel_type type);
```

Prvi argument ovog konstruktora je polje bitova kojim se precizira način na koji su pretstavljeni grafički podaci. Najčešće se koriste CL_RBGA u slučaju grafičkih podataka u boji (redom su date vrijednosti crvene, plave i zelene boje i nivo transparentnosti) i CL_LUMINANCE kada se obrađuju crno-bijele fotografije. Argument type je takođe polje bitova i sa njim se određuje kojim primitivnim tipom je pretstavljena svaka komponenta grafičkog podatka. Radi uštede prostora, nećemo navoditi sve moguće kombinacije već samo one koje se najčešće koriste:

- CL_HALF_FLOAT (broj u pokretnom zarezu veličine 16 bita)
- CL_FLOAT (broj u pokretnom zarezu veličine 32 bita)
- CL_UNSIGNED_INT8 (cio broj bez predznaka veličine 8 bita)
- CL_UNSIGNED_INT16 (cio broj bez predznaka veličine 16 bita)
- CL_UNORM_INT8 (normalizovan cio broj bez predzanaka veličine 8 bita)
- CL_ UNORM_INT16 (normalizovan cio broj bez predzanaka veličine 16 bita)

Kao što je to slučaj i sa većinom klasa omotača, i klase cl::Buffer i cl::Image imaju

metodu `getInfo` pomoću koje se dobijaju informacije o konkretnom objektu, i ona je u ovom slučaju naslijeđena od klase `cl::Memory`. Pomoću nje se može odrediti veličina i lokacija memorijske oblasti unutar kontrolne aplikacije koja je korištena pri kreaciji objekta provjeriti, kontekstu kojem pripada taj memorijski objekat, tip objekta, ... U slučaju klase `cl::Image` moguće je pristupiti dodatnim informacijama koje su specifične za podatke grafičke prirode korištenjem metode `getImageInfo`, kao što su: širina, visina, objekat klase `cl::ImageFormat` koji precizira kako su pretstavljeni grafički podaci, itd.

U prethodnom poglavlju smo pomenuli da se pored objekata klase `cl::Memory` i primitivnih tipova, jezgru kao argumenti mogu proslijediti i objekti klasa `cl::LocalSpaceArg` i `cl::Sampler`. Objekat klase `cl::LocalSpaceArg` je jednostavan objekat koji samo sadrži veličinu memorije koju želimo da se rezerviše u lokalnoj memoriji uređaja za vrijeme izvršavanja jezgra. Ovo će postati jasnije nakon što u slijedećoj glavi bude riječi o memorijskom modelu OpenCL uređaja. Objekti klase `cl::Sampler` se prosljeđuju jezgru samo u slučaju da se jezgro koristi za obradu podataka grafičke prirode. Oni tada određuju kako se pristupa podacima unutar objekata klase `cl::Image`. O ovom će biti više riječi u glavi koja objašnjava programiranje OpenCL jezgara.

3.1.3 Paralelno izvršavanje jezgara

Kada je u pitanju paralelna obrada podataka, veoma je važno da se podaci koji se obrađuju pravilno rasporede u skladu sa tipom i količinom resursa za njihovu obradu. Pored već pomenute mogućnosti da u OpenCL aplikaciji podijelimo podatke za obradu na dijelove i svaki dio posaljemo drugom uređaju na obradu, postoji i dodatna mogućnost raspodijele podataka unutar samog uređaja što omogućava dodatni nivo paralelizma. Razlog za postojanje ove mogućnosti je činjenica da skoro svi OpenCL kompatibilni uređaji posjeduju više jedinica za obradu podataka.

Pomenuta podjela se vrši direktno pri izdavanju naredbe za izvršavanje jezgra korištenjem metode `enqueueNDRangeKernel` klase `cl::CommandQueue`, koju smo već pominjali i čiji je prototip slijedeći:

```
cl_int cl::CommandQueue::enqueueNDRangeKernel(
    const Kernel &kernel,
    const NDRange &offset, const NDRange &global_size,
    const NDRange &local_size,
    const VECTOR_CLASS<cl::Event> *events = NULL,
    Event *event = NULL);
```

Kao što se može i pretpostaviti, prvi argument metode je jezgro koje će se izvršiti na uređaju. Slijedeća tri argumenta su data tipom `cl::NDRange` koji je jednostavan kontejner za vrijednosti tipa `size_t` i njihovo razumjevanje je ključno za ispravno korištenje ove metode. Da bismo mogli razumjeti značenje ovih argumenata moramo se prvo upoznati sa pojmom radnog predmeta, pa ćemo stoga napraviti malu digresiju.

Kada se obrađuju velike količine podataka, česta je praksa da se vrši iteracija kroz te podatke. Na primjer, u klasičnoj C++ aplikaciji bi se primjena nekog algoritma za obradu fotografije vršila iteracijom po svim pikselima te fotografije. Izvorni kod bi u tom slučaju izgledao otprilike ovako:

```
for(int i = 0; i < width; i++)
{
    for(int j = 0; j < height; j++)
    {
        pixel[i][j] = deblur(pixel[i][j]);
    }
}
```

Pri izvršavanju svake iteracije potrebno je izvršiti sabiranje da bi se uvećao brojač i poređenje sa uslovom, što čini ove petlje neefikasnim kada je u pitanju programiranje sa visokim performansama. Razloga za to je što su operacije poređenja spore i na najjačim procesorima, a pogotovu na grafičkim procesorima koji su zbog svoje namjene konstruisani da budu dobri za izvršavanje istih operacija u kontinuitetu. Međutim, kada grafički procesor treba da ispita neki uslov grananja, ponekad mu je potrebno i nekoliko stotina ciklusa prije nego se vrati u punu brzinu obrade podataka. Ovo je jedan od razloga zašto se ovi procesori zovu mrvilice brojeva (eng. number cruncher). Da bi se izbjeglo ovo usporenje, u OpenCL programiranju će se u okviru jezgra izvršiti samo operacije koje bi se u klasičnoj aplikaciji izvršavale u jednoj iteraciji petlje. Ovo individualno izvršavanje jezgra se u OpenCL programiranju naziva radni predmet. Treba praviti razliku između jezgra i radnog predmeta koji je sa njim određen. Jezgro je skup naredbi koje se izvršavaju nad podacima, dok je radni predmet jedna implementacija tog jezgra na konkretnom skupu podataka. U konteksu prethodnog primjera, jezgro bi se sastojalo od naredbe

```
pixel[i][j] = deblur(pixel[i][j]);
```

dok bi se jedan radni predmet određen ovim jezgrom odnosio na neki konkretan piksel, kao šte je

```
pixel[3][4] = deblur(pixel[3][4]);
```

Niz `{ i, j }` se naziva globalnim identifikatorom radnog predmeta i on je jedinstven za svaki radni predmet. Ovaj identifikator ima elemenata kolika je i dimenzionalnost podataka koje radni predmet treba da obradi i pomoću njega radni predmet može da tim podacima pristupi. U praksi se dimenzionalnost podataka odabira proslijeđivanjem objekata `cl::NDRange` sa onoliko vrijednosti tipa `size_t` koliko i podaci imaju dimenzija. Minimalan broj dimenzija je `1`, dok se maksimalan broj moze ispitati pozivanjem metode `getInfo` iz objekta klase `cl::Device`. Kada je u pitanju rad sa memorijskim objektima koji pretstavljaju grafičke podatke, broj dimenzija se postavlja na `2` ili `3`.

Nakon što smo objasnili šta su radni predmeti, možemo se vratiti metodi `enqueueNDRangeKernel`. Argument `global_size` određuje koliko radnih predmeta treba da se obradi, za svaku dimenziju. Na primjer, u slučaju obrade cijele fotografije veličine `800 x 600`, proslijedićemo 800 radnih predmeta za prvu dimenziju i 600 za drugu. Kombinovanjem argumenata `offset` i `global_size`, dio podataka se može isključiti iz obrade. Na primjer, ako bi htjeli modifikovati prethodno pomenutu fotografiju bez njenog ruba širine 100 piksela, proslijedićemo `offset` dat sa `{ 100, 100 }` i `global_size` sa vrijednostima `{ 600, 400 }`. Pomoću argumenta `local_size` se određuje veličina radne grupe koja će biti korištena pri izvršavanju jezgra. Radna grupa je skup radnih predmeta koji imaju pristup istim resursima za obradu podataka. Dodatna kompleksnost koja nastaje zbog postojanja radnih grupa je opravdana sa slijedeće dvije važne prednosti:

1. Radni elementi unutar jedne radne grupe imaju pristup istom bloku memorije sa

visokim performansama, zvanoj lokalna memorija.

2. Izvršavanje radnih elemenata unutar jedne radne grupe se može sinhronizovati korištenjem mehanizama zvanih barijere i ograde.

Lokalna memorija će biti objašnjena u slijedećoj glavi, kada bude riječi o memorijskom modelu OpenCL uređaja, dok barijere i ograde nećemo više pominjati jer se radi o veoma naprednim i komplikovanim mehanizmima. U praksi se preko argumeta `local_size` određuje koliko je, za svaku dimenziju, potrebno smjestiti radnih predmeta u radnu grupu. Ako želimo da izvršno okruženje za OpenCL automatski odredi veličinu radne grupe, tada ćemo za ovaj argument proslijediti specijalan objekat `cl::NullRange`. Resurs za obradu podataka na OpenCL uređaju koji je sposoban da izvrši čitavu radnu grupu se naziva jedinica za obradu, i pri tome vrijedi da jedinica za obradu ne moze da izvršava više radnih grupa istovremeno. Broj jedinica za obradu koje uređaj posjeduje se može odrediti unutar kontrolne aplikacije korištenjem metode `getInfo` klase `cl::Device`. Poslednja dva argumenta se koriste za obradu događaja unutar kontrolne aplikacije. Argument `events` precizira koji događaji treba da se dese unutar aplikacije prije nego se započne izvršavanje ovog jezgra, dok poslednji argument pretstavlja događaj koji će biti proslijeđen na obradu nakon što se završi izvršavanje jezgra.

S obzirom da je obrada događaja napredna tema, a i u cilju uštede prostora, nećemo ulaziti u detalje o konkretnoj implementaciji obrade događaja unutar OpenCL aplikacije. Detaljnim opisom metode `enqueueNDRangeKernel` smo došli do kraja pregleda mogućnost kontrolnih aplikacija. Neke napredne teme smo izostavili s obzirom da one izlaze izvan okvira teme ovoga rada i nisu korištene pri pisanju aplikacije micrOCLaw. U slijedećoj glavi ćemo se fokusirati na programiranje jezgara.

3.2 Programiranje jezgara

Nakon što smo izvršili detaljan uvid u programiranje kontrolne aplikacije spremni smo da napravimo skok na programiranje jezgara. Cilj ove glave je da pojasnimo fundamentalne principe programiranja jezgara, pri čemu ćemo veću pažnju podariti onima koje smo koristili u pisanju aplikacije micrOCLaw. Prvo ćemo objasniti pravila kojih se moramo pridržavati pri pisanju jezgra. Nakon ovoga ćemo detaljnije opisati različite aspekte programiranja jezgra, od kojih će prvi biti tipovi podataka. Posebnu pažnju ćemo obratiti na vektorske tipove podataka jer se njhovim korištenjem postiže paralelizam u odnosu na podatke. Poslije tipova podataka dolazi opis memorijskog modela OpenCL uređaja, čime će se konačno razjasniti neki pojmovi pomenuti u prethodnoj glavi. Pored prethodno pomenutih pojmova, opis memorijskog modela će takođe pojasniti kako se ispravnim korištenjem određenih memorijskih regiona može dodatno ubrzati izvršavanje OpenCL jezgra. Konačno, izvršićemo pregled operatora i ugrađenih funkcija. Ovaj pregled će naravno biti pojednostavljen, s obzirom da detaljan pregled svih operatora i funkcija izlazi izvan okvira ovoga rada.

3.2.1 Struktura jezgra

Kao što smo već rekli, kontrolna aplikacija je standardna aplikacija koja se izvršava na centralnoj procesorskoj jedinici i njen izvorni kod je napisan u programskom jeziku C ili u programskom jeziku C++. Kada je u pitanju kod jezgra koje se izvršava na nekom OpenCL uređaju, njegova sintaksa se ne razlikuje puno od izvorno koda funkcije u programskom jeziku C. To se može vidjeti na slijedećem jednostavnom primjeru:

```
__kernel void simple_kernel(__global int4 *quad)
{
    *quad = (int4)(1, 2, 3, 4);
}
```

Ipak, iako je očigledno da je sintaksa inspirisana programskim jezikom C, postoji veliki broj razlika između jezgra i programa napisanog u jeziku C od kojih su slijedeće tri osnovne:

1. Svaka deklaracija jezgra mora da počne sa ključnom riječi `__kernel`
2. Povratni tip svakog jezgra mora biti `void`
3. Jezgro mora da ima bar jedan argument

Ključna riječ `__kernel` označava namjeru da se kod koji slijedi izvršava na uređaju, s obzirom da se izvorni kod jezgra ne mora nalaziti u datotekama tipa `*.cl` već se može nalaziti i u `*.c` i `*.cpp` datotekama. Pravila 2. i 3. su poslijedica činjenice da jezgro pristupa podacima samo preko svojih argumenata. Jezgra uglavnom primaju argumente po referenci jer se najčešće koriste memorijski objekti, mada je moguće i proslijeđivanje argumenata po vrijednosti, kao što smo već pominjali u prethodnoj glavi. Kod proslijeđivanja po referenci, argumenti su dati preko pokazivača i pri tome vrijedi da svaki od njih ima odgovarajući tzv. kvalifikator adresnog prostora. Ovaj kvalifikator određuje u koji adresni prostor uređaja će da se smjesti argument. Više riječi o kvalifikatorima će biti u slijedećem poglavlju a zasad možemo reći da može da uzme slijedeće vrijednosti: `__global`, `__constant`, `__local` i `__private`. Kada je u pitanju tijelo jezgra, slijedećih nekoliko poglavlja zajedno će nam dati pretstavu šta sve može da se nađe u njemu.

3.2.2 Tipovi podataka u jezgru

Kao što je to uvijek slučaj, prvo ćemo pretstaviti skalarne tipove podataka a nakon toga ćemo preći na vektorske. Pod skalarnim tipovim podataka podrazumjevamo tip podatka kod kojeg svaka reprezentacija podatka sadrži samo jednu vrijednost. Svi skalarni tipovi podataka u OpenCL jezgru su pretstavljeni slijedećom tabelom

Skalarni tip podatka	Svrha
`bool`	Bulovska logička vrijednost: true (1) ili false (0)
`char`	Cio broj dužine 8 bita sa predznakom
`unsigned char / uchar`	Cio broj dužine 8 bita bez predznaka
`short`	Cio broj dužine 16 bita sa predznakom
`unsigned short / ushort`	Cio broj dužine 16 bita bez predznaka
`int`	Cio broj dužine 32 bita sa predznakom
`unsigned int / uint`	Cio broj dužine 32 bita bez predznaka
`long`	Cio broj dužine 64 bita sa predznakom
`unsigned long / ulong`	Cio broj dužine 64 bita bez predznaka
`half`	Broj u pokretnom zarezu dužine 16 bita, u skladu sa IEEE-754-2008 standardom
`float`	Broj u pokretnom zarezu dužine 32 bita, u skladu sa IEEE-754 standardom
`intptr_t`	Cjelobrojna vrijednost sa predznakom u koju se može konvertovati iz pokazivača na void tip
`uintptr_t`	Cjelobrojna vrijednost bez predznaka u koju se može konvertovati iz pokazivača na void tip
`ptrdiff_t`	Cjelobrojna vrijednost sa predznakom dobijena kao rezultat oduzimanja pokazivača
`size_t`	Cjelobrojna vrijednost bez predznaka dobijena pozivom `sizeof` operatora
`void`	Podatak koji nema tip

Tabela 3.5 Skalarni tipovi podataka

i nema potrebe da se detaljnije objašnjavaju s obzirom na veliku sličnost sa tipovima iz jezika C i C++. Ipak, razjasnićemo samo razlog izostanka tipa `double`, koji pretstavlja broj u pokretnom zarezu dužine 64 bita, iz tabele 3.5. Naime, s obzirom da većina grafičkih procesora koristi brojeve u pokretnom zarezu dužine 32 bita, OpenCL standard zahtjeva da uređaj podrzava samo tip `float`. Tip `half` takođe ne mora biti podržan od strane uređaja, ali se ipak nalazi u gornjoj tabeli jer ga veliki broj uređaja podržava bez obzira što to nije zahtjevano od strane OpenCL standarda. Iz toga razloga, ako želimo da izvršimo jezgro koje koristi tip `half`, odnosno tip `double`, tada prethodno moramo ispitati da li uređaj podržava ekstenziju `cl_khr_fp16`, odnosno ekstenziju `cl_khr_fp64`. Još jedna stvar vezana za brojeve u pokretnom zarezu koja je bitna je da OpenCL ne zahtjeva od uređaja strogu kompatibilnost sa IEEE-754 standardom, što je veoma bitno ako se piše

aplikacija koja se puno oslanja na operacije sa brojevima u pokretnom zarezu. Na primjer, OpenCL ne zahtjeva da uređaj podržava zaokruživanje prema nuli. Iz toga razloga, ako želimo da koristimo ovo zaokruživanje, moramo da ispitamo da li je ono podržano za svaki tip posebno (`half`, `float` i `double`) korištenjem funkcije `cl::Device::getInfo`.

Kada su pitanju vektorski tipovi podataka, može da dođe do zabune zbog njihove sličnosti sa nizovima jer se u oba slučaja radi o posjedovanju više elemenata istog tipa. Ipak, razlika je velika jer vektor nekog tipa ima predefinisanu veličinu, i pri tome vrijedi da se pri izvršavanju neke operacije na vektoru ona vrši na svim elementima istovremeno. Kao i kod skalarnih tipova, pretstavićemo sve moguće vektorske tipove tabelom.

Vektorski tip podatka	Svrha
`char`*n*	Vektor koji sadrži *n* cijelih brojeva sa predznakom dužine 8 bita
`uchar`*n*	Vektor koji sadrži *n* cijelih brojeva bez predznaka dužine 8 bita
`short`*n*	Vektor koji sadrži *n* cijelih brojeva sa predznakom dužine 16 bita
`ushort`*n*	Vektor koji sadrži *n* cijelih brojeva bez predznaka dužine 16 bita
`int`*n*	Vektor koji sadrži *n* cijelih brojeva sa predznakom dužine 32 bita
`uint`*n*	Vektor koji sadrži *n* cijelih brojeva bez predznaka dužine 32 bita
`long`*n*	Vektor koji sadrži *n* cijelih brojeva sa predznakom dužine 64 bita
`ulong`*n*	Vektor koji sadrži *n* cijelih brojeva bez predznaka dužine 64 bita
`float`*n*	Vektor koji sadrži *n* brojeva u pokretnom zarezu dužine 32 bita

Tabela 3.5 Vektorski tipovi podataka

Pored tipova navedenih u tabeli OpenCL podržava i tipove `half`*n* i `double`*n*, pod uslovom da su odgovarajući skalarni tipovi podržani. Što se tiče podržanih vrijednosti za *n*, moguće su 2, 3, 4, 8 I 16. Ipak, ne podržava svaki uređaj ove vrijednosti za svaki tip podatka. Na primjer, većina grafičkih procesora ne može direktno da izvršava operacije sa tipom `float16`, što je vektor veličine 512 bita. Veličine vektora za svaki tip sa kojima uređaji mogu da operišu direktno se nazivaju preferirane širine vektora. Ove vrijednosti možemo saznati korištenjem funkcije `cl::Device::getInfo` kojoj se proslijeđuje parametar `CL_DEVICE_PREFERRED_VECTOR_WIDTH_`*`TYPE,`* pri čemu *`TYPE`* može da uzme vrijednosti `CHAR`, `SHORT`, `INT`, `LONG` i `FLOAT`. S obzirom da se dužina vektora ne može odabrati za vrijeme izvršavanja jezgra, ako želimo da program koji koristi vektore bude portabilan onda pri pisanju jezgra moramo koristiti jedan od slijedeća dva trika. U slučaju da algoritam ne zavisi od dužine vektora, proslijedićemo konstatnu pri kompilaciji programa za taj određeni uređaj koja pokazuje kolika je preferirana dužina vektora, dok ćemo u kodu jezgra imati preprocesorske direktive koje određuju koji dio će se izvršiti u zavisnosti od te dužine. Ako

je dužina vektora ključna za algoritam, onda će se morati pisati funkcije koje koriste različite dužine vektora i za vrijeme izvršavanja kontrolna aplikacija će odlučiti od koje funkcije da stvori jezgro koje će se koristiti.

Kada je u pitanju incijalizacija vektora i pristup njegovim elementima, sintaksa je prilično intuitivna. Inicijalizacija se vrši grupisanjem komponenti vektora u zagrade i kastingom u odgovarajući tip vektora, pri čemu se mogu koristiti skalari, vektori manje dimenzije ili kombinacija skalara i vektora manje dimenzije. Ovo je najlakše razumjeti gledajući primjer:

```
float scal = 4.0;
float2 vec1 = (float2)(2.0, 3.0);
float4 vec2 = (float4)(vec1, vec1);
float4 vec3 = (float4)(1.0, vec1, scal);
```

Sto se tiše sintakse za pristup komponentama vektora, OpenCL omogućuje indeksiranje pomoću brojeva i pomoću slova. Kod indeksiranja pomoću brojeva, na naziv vektora se dodaje `.s` nakon čega slijede indeksi komponenti kojim se želi pristupiti. S obzirom da vektor može imati maksimalno 16 komponenti, koriste se heksadecimalni brojevi od 0 do F da bi za pristup svakoj komponenti bila potreban samo jedan broj. Što se tiče indeksiranja pomoću slova, ono je specijalan slučaj jer se koristi za vektore dužine manje ili jednake četiri i prvenstveno se koristi sa vektorima koji sadrže grafičke podatke (tri vrijednosti za boju i jedna za transparentnost). Slova korištena za indeksiranje su `x`, `y`, `z` i `w`. Pogledaćemo sada nekoliko primjera da vidimo kako se ovo koristi u praksi:

```
char8 str = (char8)('O','p','e','n','C','L','a','w');
char c = str.s4;    // C
char4 vc = str.s0246;    // OeCa
char2 vc2 = vc.yw;        // ea
char4 vc3 = (char4)(str.s1, vc.yz, str.s5);    // peCL
```

Prije kraja priče o skalarima i vektorima napomenimo još jednu bitnu stvar. U slučaju da se podacima pristupa pomoću memorijskih operacija sa pokazivačima veoma je bitno kako su ti podaci upisani u memoriji uređaja na kome će se jezgro izvršavati. Iz toga razloga treba napisati dvije verzije algoritma, pri čemu imamo dvije opcije. Prva je da se te verzije napišu u dva različita jezgra pa se za vrijeme izvršavanja iz kontrolne aplikacije direktno ispita da li je uređaj „little-endian“ ili „big-endian“ i pozove odgovarajuće jezgro. Druga je da se piše jedno jezgro čiji je kod razdvojen preprocesorskom direktivom `__ENDIAN_LITTLE__` koja je definisana samo za uređaje koji su „little-endian“ pa će se pri kreaciji jezgra generisati odgovarajući kod.

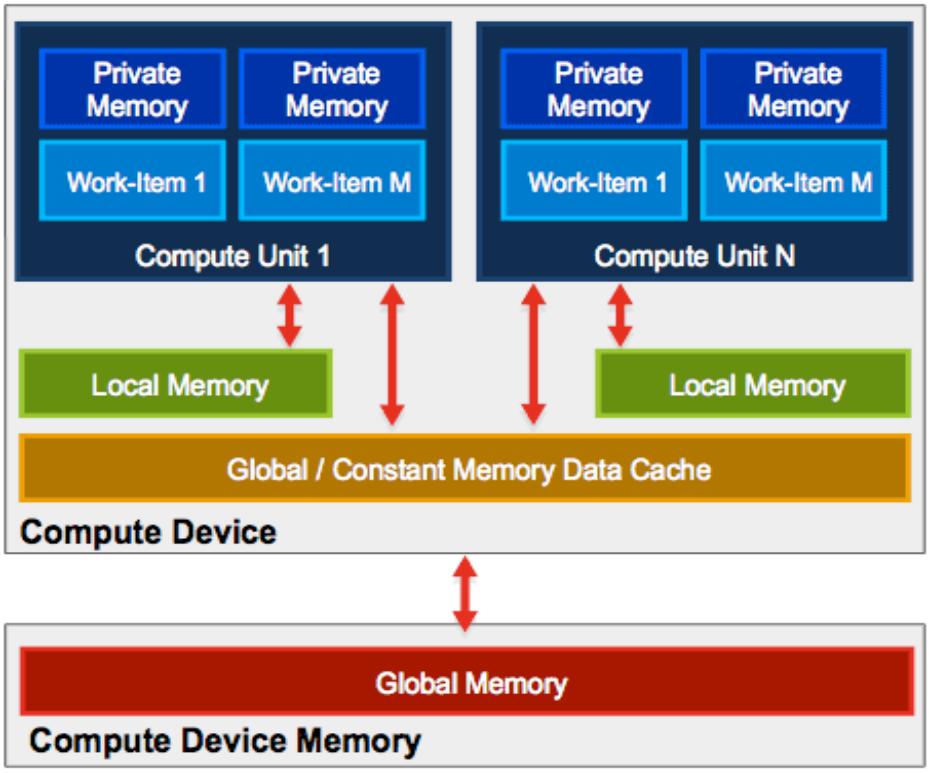
3.2.3 Memorijski model OpenCL uređaja

Cilj ovog poglavlja je da se razjasne već ranije pominjane različite vrste momorijskih regiona u OpenCL uređajima. Pored toga, želimo da pretstavimo odnose između memorijskih regiona i sa radnim predmetima i radnim grupama. Ovi odnosi nisu jednostavni ali njihovo razumjevanje je apsolutno neophodno da bi se razumjelo kako se jezgro izvršava na uređaju, što dovodi do pisanja efikasnijih algoritama.

OpenCL uređaj ima četiri adresna prostora:

* Globalna memorija - Koristi se za skladištenje podataka dostupnih čitavom uređaju.
* Konstantna memorija - Isto kao i globalna memorija, s tim da dozvoljava samo čitanje.
* Lokalna memorija - Koristi se za skladištenje podataka dostupnih radnim grupama i radnim predmatima.
* Privatna memorija – Koristi se za skladištenje podataka dostupnih samo određenom radnom predmetu.

Ovi adresni prostori su šematski pretstavljeni na slijedećoj slici



Slika 3.1 **Memorijski model OpenCL uređaja**

Većina uređaja drži globalnu i konstantu memoriju u jednom velikom memorijskom regionu pa se one ponekad i pretstavljaju kao jedan region. Pored samog uređaja, ovom regionu ima pristup i kontrolna aplikacija koja na njega upisuje i sa njega čita sadržaj memorijskih spremnika. Ovaj region je drastično veći u odnosu na druge, ali je zato pristup njemu od strane radnih predmeta i drastično sporiji. Lokalna memorija je puno manja ali joj radni predmeti pristupaju približno stotinu puta brže, pa je dobra za smještanje međurezultata operacija. Ovaj memorijski region se nalazi na jedinici za obradu i dostupan je radnoj grupi čiji se predmeti trenutno izvršavaju. Iz tog razloga, podacima koje jedan radni predmet smjesti u lokalnu memoriju mogu pristupiti samo radni predmeti unutar iste radne grupe. Pored lokalne, radni predmet ima pristup regionu veoma brze privatne memorije. Ova memorija je prilično mala pa se mora pažljivo koristiti.

U kodu se za deklaraciju adresnog prostora koriste kvalifikatori `__global`, `__constant`, `__local` i `__private`, koje smo već pomenuli u prethodnoj glavi. Postoji više

pravila kojih se mora pridržavati u radu sa kvalifikatorima, inače se kod neće kompilirati. Unutar tijela jezgra, kvalifikator `__global` se može koristiti samo za pokazivače. Što se tiče varijabli sa kvalifikatorom `__local`, treba imati u vidu da se one dodjeljuju jednom za svaku radnu grupu koja se izvršava, kao i da se poništavaju nakon što se završi izvršavanje radne grupe. Pored toga, ako se varijabla sa ovim kvalifikatorom kreira unutar jezgra (tj. nije mu proslijeđena kao argument), onda se mora prvo deklarisati pa onda inicijalizovati. Još jedno pravilo je da se za sve varijable, deklarisane unutar jezgra bez kvalifikatora, uzima podrazumjevani kvalifikator `__private`. Konačno, ne smije se vršiti kasting između pokazivača koji referenciraju memoriju u različitim adresnim prostorima.

Ako jezgro kao argumente ima memorijske objekte, pri slanju jezgra na izvršavanje njihov sadržaj će biti skladišten unutar globalne ili lokalne memorije uređaja. Iz toga razloga, odgovarajući parametri u listi parametara jezgra imaju kvalifikator `__global` ili `__constant`. Pri tome, ti parametri moraju biti pokazivači jer jezgra pristupaju memorijskim objektima po referenci. Ovo je posljedica činjenice da kontrolna aplikacija ne može direktno da pristupa memoriji uređaja. Ipak, ako želimo da neki argument bude smješten u lokalnu memoriju (što je relativno čest slučaj zbog veće brzine pristupa), možemo da to uradimo indirektno proslijeđivanjem objekta `cl::LocalSpaceArg` metodi `cl::Kernel::setArg`. U tom slučaju će uređaj samo da dodjeli memoriju za argument, a njegova inicijalizacija mora da bude uređena od strane radnih predmeta kada se budu izvršavali. Ovaj objekat služi za određivanje koliko memorije je potrebno da se dodjeli za argument i to se unutar kontrolne aplikacije vrši pomoću funkcije `__local` , što možemo vidjeti u slijedećem fragmentu izvornog koda

```
cl::LocalSpaceArg arg = cl::__local(sizeof(long));
kernel.setArg(0, arg);
```

U kodu jezgra, odgovarajući argument mora da bude pokazivač sa kvalifikatorom `__local`:

```
__kernel void deblur(__local long *arg) {
    ...
}
```

Ako kao drugi parametar funkcije `cl::Kernel::setArg` proslijedimo primitivni tip podatka umjesto memorijskog nekog objekta, oni će biti proslijeđeni po vrijednosti. Ova metoda omogućava da se iz kontrolne aplikacije inicijalizuje argument jezgra koji će biti smješteni u privatnoj memoriji svakog radnog predmeta koji se bude izvršavao. Odgovarajući argument jezgra ne smije biti pokazivač već primitivni tip i ne smije da ima kvalifikator adresnog prostora. Pri tome, činjenica da argument mora da bude primitivni tip ne znači da mora da bude skalar. Slijedeći fragment koda postavlja argument koji će da bude smješten u privatnu memoriju:

```
float nums[8] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 };
kernel.setArg(0, nums);
```

Odgovarajući argument će se tada u jezgru interpretirati kako vektor:

```
__kernel void blurring(float8 arg) {
    ...
}
```

Ovim smo završili pregled memorijskih regiona OpenCL uređaja i njihovog odnosa sa argumentima jezgara.

3.2.4 Operatori i funkcije

Nakon što smo u prethodnom poglavlju dobili uvid u skalarne i vektorske tipove, želimo da damo pregled operatora i funkcija koji se mogu primjeniti na te tipove. Pri tome ćemo naravno posmatrati operatore i funkcije koji su „ugrađeni“ u jezik (ne zahtjevaju dodatna zaglavlja ni biblioteke). S obzirom na veliki broj ugrađenih funkcija, fokusiraćemo se na one koje su specifične za OpenCL, dok ćemo u slučaju one opšte (npr. matematičkih) samo naglasiti da postoje.

Ono što je veoma praktično kada se programira OpenCL jezgro je što se koriste isti operatori kao i u programskom jeziku C, i pri tome se oni mogu koristiti sa vektorskim i skalarnim tipovima. Pri tome se prenose sva pravila sa korištenjem operatora, tako da ćemo objasniti samo neke specifične situacije. U slučaju primjene aritmetičkog operatora na vektor i skalar, taj operator se primjenjuje na skalar i svaku komponentu vektora pojedinačno i rezultat je vektor odgovarajuće dužine. U slučaju relacionih operatora, situacija je slična. Operator se opet primjenjuje na slalar i svaku komponentu vektora pojedinačno, pri čemu odgovarajuća komponenta u rezultatu pretstavlja rezultat koji pokazuje da li je poređenje vratilo rezultat tačno ili netačno. Da bi se navedeni izrazi sa relacionim operatorima mogli koristiti u uslovnim naredbama koriste se ugrađene funkcije `all` i `any`, koje ispituju da li je uslov ispunjen za sve, odnosno bar jednu komponentu vektora.

Počnimo sada sa pregledom ugrađenih funkcija. One se mogu grupisati u slijedećih sedam kategorija:

* Funkcije vezane za radne grupe i radne predmete - Koriste se za određivanje dimenzionalnosti podataka, određivanje globalnih i lokalnih identifikatora radnih predmeta, ...
* Funkcije za transfer podataka – Koriste se za učitavanje i skladištenje podataka između različitih memorijskih regiona
* Funkcije za obradu brojeva u pokretnom zarezu - Koriste se za razne operacije nad vektorima i skalarima koji sadrže brojeve pretstavljene u pokretnom zarezu (zaokruživanje, eksponencijalne i logaritamske operacije, …)
* Funkcije za obradu cijelih brojeva - Koriste se za poređenje i aritmetičke operacije nad vektorim čije su komponente cijeli brijevi
* Funkcije za odabir i razmještanje - Koriste se za kreiranje novih vektora odabirom ili ramještanjem komponenti postojećih vektora
* Funkcije za poređenje vektora - Koriste se za ispitivanje da li komponente vektora zadovoljavaju određene uslove
* Geometrijske funkcije - Ove funkcije se koriste za sklarni i vektorski proizvod, normalizaciju vektora, …

Kao što smo već objasnili u prethodnoj glavi, kontrolna aplikacija korištenjem metode `enqueueNDRangeKernel` određuje dimenzionalnost podataka unutar jezgra, broj radnih predmeta po svakoj dimenziji, itd. Da bi radni predmet moga da pravilno odradi ono za šta je namjenjen, treba da ima mogućnost da odredi dio podataka koje baš on treba da obradi. On to postiže korištenjem specijalnih ugrađenih funkcija pretstavljenih u slijedećoj tabeli:

Funkcija	Svrha
`uint get_work_dim()`	Vraća broj dimenzija u indeksnom prostoru jezgra
`size_t get_global_size(uint dim)`	Vraća broj radnih predmeta za odabranu dimenziju
`size_t get_global_id(uint dim)`	Vraća element globalnog identifikatora za radni predmet, koji odgovara odabranoj dimenziji
`size_t get_global_offset(uint dim)`	Vraća početno odstupanje koje je korišteno pri određivanju globalnog identifikatora

Tabela 3.6 Funkcije specifične za radne predmete

Pored navedenih funkcija koje pružaju globalne informacije, postoje i odgovarajuće funkcije koje pružaju lokalne informacije vezane za radnu grupu unutar koje se radni predmet izvršava. Te funkcije su pretsavljene u slijedećoj tabeli:

Funkcija	Svrha
`uint get_num_groups(uint dim)`	Vraća broj radnih grupa za datu dimenziju
`size_t get_group_id(uint dim)`	Vraća identifikator grupe kojoj radni predmet pripada, koji odgovara odabranoj dimenziji
`size_t get_local_id(uint dim)`	Vraća identifikator radnog predmeta unutar njegove radne grupe, koji odgovara odabranoj dimenziji
`size_t get_global_offset(uint dim)`	Vraća broj radnih predmeta unutar radne grupe za odabranu dimenziju

Tabela 3.7 Funkcije specifične za radne grupe

U prethodnom poglavlju smo imali priliku da vidimo razlike između adresnih prostora u OpenCL jezgru. Uzimajući te razlike u obzir, jasno je da je važno imati mogućnost prebacivanja podataka iz jednog memorijskog regiona u drugi. Ako želimo da prebacujemo podatke koji su istog tipa (skalari istog tipa ili vektori istog tipa i dužine) onda će operator dodjele biti dovoljan za obavljanje ovoga zadatka. Sa druge strane, ako želimo da prebacujemo podatke između vektora i nizova primitivnih tipova, moramo koristi posebne ugrađene funkcije. Na primjer, često kao ulazni parametar jezgra dobijamo neki niz koji sadrži podatke za obradu. U slučaju da želimo da iskoristimo mogućnosti vektorske obrade podataka koju nam pruža OpenCL, moramo da te podatke unutar jezgra učitamo u vektor. Ovo se postiže funkcijom čiji je prototip slijedeći

```
vector vload<i>n</i>(size_t offset,
    const __(global|constant|local|private) scalar *mem);
```

i pri čemu *n* označava broj komponenti u vektoru a izraz `__(global|constant|local|private)` znači da su dozvoljena sva četiri kvalifikatora adresnog prostora. Kada je u pitanju transfer podataka iz vektora u niz, imamo odgovarajuću funkciju sličnog prototipa

```
void vstore<i>n</i>(vector vec, size_t offset,
    __(global|local|private) scalar *mem);
```

Kao i u prethodnom slučaju, *n* označava broj komponenti u vektoru. Iz očiglednih razloga, ne može se koristiti kvalifikator `__constant` sa ovom funkcijom. U oba slučaja, parametar označava odstupanje od prvog elementa u nizu a skalar i vektor koji učestvuju u operaciji transfera memorije moraju imati isti tip podatka (npr. `char`, `uint`, `float`, …).

Prethodno opisane ugrađene funkcije su ujedno i najčešće korištene u OpenCL aplikacijama. Pored njih postoji još veliki broj funkcija u preostalih pet kategorija, od kojih se većina koristi za matematičke operacije. Radna grupa za OpenCL je odlučila da svi nazivi ovih funkcija budu isti kao oni definisani u zaglavlju `math.h` koje se koristi u programskom jeziku C, tako da su one poznate i nema potrebe da ih objašnjavamo. Pored njih, radi uštede prostora, takođe ćemo izostaviti diskusiju funkcija za poređenje vektora i funkcija za odabir i razmještanje. Razlog za izostavljanje funkcija za poređenje vektora je njihov veliki broj i činjenica da je teško odabrati one koje su važnije od ostalih pa da eventualno njih opišemo. Što se tiče funkcija za odabir i razmještanje, one se generalno ne koriste često i uopšte se ne koriste u pisanju aplikacije koju ćemo pretstaviti u slijedećoj glavi.

3.2.5 Obrada grafičkih podataka

U prethodnoj glavi smo već pominjali memorijske objekte koji pretstavljaju grafičke podatke i njihovo proslijeđivanje jezgrima na obradu. U ovoj glavi ćemo obratiti pažnju na to kako se ta obrada vrši unutar jezgra. Prije toga, s obzirom da se za obradu grafičkih podataka mogu koristit i opšti memorijski spremnici, navešćemo prednosti korištenja memorijskih objekata namjenjenih grafičkim podacima:

* Ako se za obradu koristi grafička procesorska jedinica, grafički podaci su smješteni unutar posebnog dijela globalne memorije koji se zove teksturna memorija. Ova memorija je keširana radi bržeg pristupa, što ubrzava proces obrade podataka.
* Funkcije koje se koriste za čitanje i upis grafičkih podataka se mogu koristi bez obzira na način formatiranja podataka koji pretstavljaju piksele, dokle god je način formatiranja jedan on mnogih koje OpenCL podržava.
* OpenCL posjeduje posebne strukture podataka zvane uzornici, pomoću kojih se lako konfiguriše metoda čitanja grafičkih podataka.
* U OpenCL jezgrima se mogu koristiti ugrađene funkcije za pristupanje podacima specifičnim za grafičke podatke (dimenzije, bitna dubina, format pretstavljanja piksela), što olakšava programiranje algoritama za obradu grafičkih podataka.

Kada su u pitanju strukture podataka, najbitnije su `image2d_t` i `image3d_t` (koje pretstavljaju dvodimenzionalne i trodimenizionalne grafičke podatke) i `sampler_t` koja pretstavlja gore pomenute uzornike.

Objekti `image2d_t` i `image3d_t` služe kao mehanizam za čuvanje grafičkih podataka, koje aplikacije koriste za transfer između kontrolne aplikacije i uređaja, kao i obratno. U zavisnosti od dimenzija memorijskog objekta unutar kontrolne aplikacije koji se uređaju prosljeđuje, koristiće se jedna od gore pomenutih struktura kao argument jezgra. S obzirom da većina uređaja smješta ove strukture u poseban memorijski region, ograničenje je da jezgro može pisati ili čitati ove strukture ali ne i oboje. Stoga, argumenti jezgra ovoga tipa koji pretstavljaju ulazne parametre imaju kvalifikator `read_only`, dok oni koji pretstavljaju izlazne imaju kvalifikator `write_only`. Pri tome, ti argumenti ne smiju biti

pokazivači jer ove strukture nisu namjenjene za direktan pristup korištenjem memorijskih operacija. Prototip jezgra sa dva argumenta, od kojih je prvi trodimenzionalan i pretstavlja ulazne podatke a drugi dvodimenzionalan i pretstavlja izlazne podatke, izgleda ovako:

```
__kernel void merge(read_only image3d_t mpg, write_only image2d_t jpg)
```

Što se tiče uzornika, oni se mogu definisati unutar kontrolne aplikacije pa proslijediti jezgru. Mi nećemo obraćati pažnju na tu metodu jer postoji puno elegantnija i jednostavnija varijanta definisanja uzornika direktno unutar jezgra. Da bi se to postiglo, dovoljna je jedna naredba unutar jezgra kojom se u potpunosti definiše uzornik, i čija sintaksa izgleda ovako:

```
const sampler_t sampler_name = sampler_properties;
```

Izraz `sampler_properties` se kreira kombinovanjem predefinisanih konstanti (razdvojenih znakom |) koje određuju osobine uzornika, a samim tim i način na koji se vrši čitanje podataka iz struktura koje pretstavljaju grafičke podatke. Radi uštede prostora nećemo objašnjavati sve ove konstante, već ćemo dati primjer kombinacije ovih konstanti koji se najčeće koristi:

```
const sampler_t smpl = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP;
```

Na ovaj način je kreiran uzornik koji čita podatke korištenjem standardnih cjelobrojnih koordinata pri čemu se u slučaju korištenja koordinata izvan dozvoljenih dimenzija vraća fiksna boja (podrazumjevana vrijednost je crna). Ovo je još jedna prednost u odnosu na korištenje običnih memorijskih spremnika za obradu grafičkih podataka, jer bi u tom slučaju program prestao da se izvršava.

Nakon što smo se upoznali sa grafičkim objektima i uzornicima, možemo svoju pažnju da usmjerimo na ugrađene funkcije koje se unutar jezgra mogu koristiti za obradu grafičkih podataka. OpenCL posjeduje preko dvadeset ovakvih funkcija i one se mogu podjeliti u tri kategorije: funkcije za čitanje, funkcije za pisanje i funkcije za pristupanje informacijama specifičnim za grafički objekat. Sve funkcije za čitanje su međusobno veoma slične. Svaka od njih kao ulazne parametre uzima grafički objekat, uzornik i vektor koji sadrži koordinate piksela kojem želimo pristupiti. Takođe, svaka vraća vektor koji sadrži podatke piksela, s tim da je jedina razlika u tipu podatka kojim su oni pretstavljeni. Ako se radi o `image2d_t` objektu, koordinate moraju biti date u obliku `int2` ili `float2`. U slučaju objekta tipa `image3d_t`, koordinate se daju sa `int4` ili `float4`. U slijedećem primjeru vidimo pristupanje pikselu čije su koordinate date sa (10, 15) i čiji su podaci pretstavljeni tipom `uint4`:

```
uint4 pixel = read_imageui(jpg, sampler, (int2)(10, 15));
```

Poput funkcija za čitanje, i funkcije za pisanje su međusobno veoma slične. One ne zahtjevaju korištenje uzornika a glavna razlika im je u tipu podatka korištenom za pretstavljanje podataka piksela. Skoro svi OpenCL kompatibilni uređaji podržavaju pisanje u objekte tipa `image3d_t` ali ovo ipak nije podrazumjevana sposobnost. Iz toga razloga, ako želimo da je koristimo, moramo ispitati postojanje `cl_khr_3d_image_writes` ekstenzije i dodati slijedeću liniju koda našem jezgru:

```
#pragma OPENCL EXTENSION  cl_khr_3d_image_writes: enable
```

Da bi ove funkcije mogle ispravno koristiti, mora se znati format kojim je pretstavljen piksel. Slijedeći primjer demonstrira upisivanje piksela u standardnom `CL_RGB` formatu:

```
write_imageui(tiff, coords, (uint4)(255, 128, 255, 0));
```

Funkcije za pristup informacijama se koriste za određivanje dimenzija grafičkog objekta i načina na koji se njegovi pikseli pretstavljeni. Ove funkcije su intuitivne i jednostavne za korištenje pa ih iz tog razloga nećemo detaljno objašnjavati. Pregledom ugrađenih mehanizama za obradu grafičkih podataka u OpenCL jezgrima smo završili i pregled OpenCL programiranja kojem je čitava ova glava bila posvećena. U slijedećoj glavi se bavimo opisom i detaljima implementacije aplikacije micrOCLaw.

4 Dizajn i implementacija aplikacije „micrOCLaw“

U ovoj glavi ćemo pretstaviti aplikaciju micrOCLaw koja demonstrira obradu digitalnih fotografija korištenjem metoda paralelnog programiranja. Pri tome će pored tehničkih detalja, vezanih za konkretnu implemetaciju, biti riječi i o procesu dizajna ove aplikacije. Tehnologija koja nam omogućuje paralelno programiranje je OpenCL, koji je opisan u prethodne dvije glave. Razlozi za odabir baš ove tehnologije će također biti pretstavljeni u ovoj glavi.

4.1 Dizajn

U procesu dizajna je prvo potrebno precizno identifikovati funkcionalnost aplikacije, kao i korisničku grupu kojoj je ta aplikacija namjenjena. Razlog za to je da ta dva faktora imaju veliki uticaj na ostale detalje dizajna, kao što je na primjer korisnički interfejs. Nakon toga, s obzirom da će se za implementaciju većeg dijela aplikacije (kontrolna aplikacija) koristiti programski jezik C++ koji posjeduje mehanizme za objektno-orijentisano programiranje, osnovne funkcionalne cjeline se preslikavaju u odgovarajuće klase. Pri tome se sve vrijeme treba nastojati da se dizajn što pažljivije osmisli, da ne bi bilo neugodnih iznenađenja za vrijeme implementacije koji bi mogli nepotrebno produžiti tu fazu razvoja aplikacije.

S obzirom da treba da demonstrira primjenu paralelnog programiranja korištenjem uređaja kao što grafička procesorska jedinica, klijent verzija operativnog sistema Windows se nametnula kao logičan izbor za ciljanu platformu aplikacije micrOCLaw. Ova aplikacija posjeduje slijedeću funkcionalnost:

* Mogućnost učitavanja digitalne fotografije sa diska, kao i prikazivanje iste u klijentskom prostoru aplikacije.
* Mogućnost pregleda OpenCL kompatibilnih uređaja na sistemu, kao i odabira uređaja koji će se koristiti za obradu fotografije.
* Obrada fotografije primjenom jednog od podržanih algoritama.
* Snimanje modifikovane fotografije na disk.

S obzirom na ovu ograničenu funkcionalnost, ova aplikacija je namjenjena isključivo studentima i ostalim osobama zainteresovanim za OpenCL programiranje, koji pregledom njenog izvornog koda mogu da steknu uvid u određene aspekte OpenCL programiranja. Postoji nekoliko razloga za ograničavanje funkcionalnosti aplikacije micrOCLaw na one koji su navedeni iznad. Prvi od njih je činjenica da je razvoj aplikacije sa grafičkim interfejsom u programskom jeziku C++ komplikovan i dosta sporiji u odnosu na „jezike višeg nivoa“. Pored toga, programiranje OpenCL mehanizama za pristupanje kompatibilnim uređajima i njihovo korištenje dugo traje pa bi se uvođenjem dodatne funkcionalnosti vrijeme razvoja aplikacije drastično povećalo.

Aplikacija micrOCLaw je napisana korištenjem arhitekturalnog šablona baziranog na komponentama, i one su slijedeće:

* Korisnički interfejs - U ovu komponentu spadaju moduli koji omogućavaju odabir fotografije sa diska koju treba učitati, prikazivanje podržanih uređaja koji su prisutni
  na sistemu, prikazivanje učitane aplikacije na ekranu, ...
* Modul za rukovanje sa fotografijama - Koristi interfejse za aplikacija za Windows

platformu (Windows API) da bi omogućio fotografije sa diska u objekat koji je pretstavlja, kao i njeno snimanje na nakon modifikacije. Pored toga, on posjeduje mehanizam za čitanje i piksela u objektu koji pretstavlja fotografiju.

* OpenCL modul - Ovo je najkomplikovanija komponenta aplikacije. Ona posjeduje mehanizme za pružanje informacija o kompatibilnim uređajima. Pored toga, zadužena je za transfer piksela fotografije između kontrolne aplikacije i uređaja, kao i za pripremanje odgovarajućeg jezgra za izvršavanje i proslijeđivanje istog uređaju, u zavisnosti od algoritma koji je odabran pomoću korisničkog interfejsa.
* Jezgra – Skup svih jezgara koja obuhvataju implementaciju algoritama za obradu fotografija, koja se izvršavaju na uređaju.

U opštem slučaju, pogotovo kada se radi o komercijalnim aplikacijama, korisnički interfejs je nešto čemu treba posvetiti dosta pažnje jer je to ono što korisnik prvo zapazi. On treba da bude intuitivan i jednostavan za korištenje a da pritom nudi zadovoljavajuću funkcionalnost. Pored toga, pri dizajnu korisničkog interfejsa treba uvijek da se nastoji da bude originalan i drugačiji od interfejsa aplikacija slične namjene. S obzirom da je aplikacija micrOCLaw demonstrativne prirode, dizajnu njenog korisničkog interfejsa nije posvećeno puno pažnje. Dodatni razloz za to su i već naveda komplikovanost implementacije grafičkog interfejsa u nativnim aplikacijama napisanim u programskom jeziku C++. Dizajn korisničkog interfejsa aplikacije micrOCLaw je minimalistički i prilično jednostavan, s tim da je posebna pažnja uložena da bi njegov odziv bio zadovoljavajući. Mogućnostima aplikacije micrOCLaw se pristupa isključivo korištenjem elemenata menija. Ovo se čini kao logična odluka pri dizajnu korisničkog interfejsa jer je ovo aplikacija koja prikazuje i obrađuje fotografije, pa je stoga klijentski prostor aplikacije isključivo rezervisan za prikaz istih.

Od svih komponenti, OpenCL modul je najkomplikovaniji zbog posjedovanja mehanizama za interakciju sa hardverom na mašini. On je najosjetljiviji dio čitave aplikacije jer eventualne greške u njenom kodu mogu dovesti do „pucanja“ aplikacije za vijeme izvršavanja. S obzirom da koristimo objektno-orijetisan programski jezik za razvoj aplikacije, logična je odluka da se ova komponenta bude pretstavljena jednom klasom. Pored toga, prirodno je da imamo samo jedan objekat ove klase jer bi istovremeno pristupanje hardveru od strane više objekata drastično zakomplikovalo implementaciju i moglo da stvori velike probleme. Iz tog razloga, odluka da se ova klasa implementira korištenjem šablona dizajna zvanog „singleton“ (koji omogućava postojanje samo jedne instance klase), je opravdana. Primjetimo da se ova odluka u fazi dizajna mogla primjeniti i na modul za rukovanje sa fotografijama. Međutim, mora se uzeti u obzir da imamo tzv. „1-1“ odnos između fotografije i modula koji omogućava rukovanje sa njom. Stoga bi, u slučaju eventualnog proširenja aplikacije sa podrškom za istovremenu obradu više od jedne fotografije istovremeno, bila potrebna mogućnost kreiranja više od jedne instance klase koja pretstavlja ovaj modul.

4.2 Implementacija

Konačno smo došli do najbitnijeg dijela u procesu izrade aplikacije. U ovoj glavi ćemo prodiskutovati implementaciju svih komponenti aplikacije micrOCLaw navedenih u prethodnom poglavlju. Pri tome ćemo pomenuti i neke dodatne tehnologije i metode programiranja koje su (pored tehnologije OpenCL) korištene u implementaciji ove aplikacije. Kao što smo već rekli, kontrolni dio aplikacije je napisan u programskom jeziku C++. Pri tome, kao integrisano razvojno okruženje je korišten Visual Studio. Ovo okruženje je odabrano jer ga autor već poznaje od ranije, kao i zbog velikog broja korisnih alata koje ono posjeduje. Sama struktura datoteka implementacije je standardna za aplikaciju napisanu korištenjem gore pomenutog okruženja. U baznom direktoriju se nalaze datoteke vezane za tzv. „rješenje“ (eng. solution) i direktoriji za svaki projekat. U našem slučaju imamo samo jedan projekat (koji pretstavlja aplikaciju micrOCLaw) i on sadrži „projektne datoteke“, datoteke koje definišu resurse aplikacije i datoteke sa izvornim kodom. Datoteke sa izvornim kodom su grupisane u direktorijume koji pretstavljaju komponente dizajna, što nije neophodno i urađeno je samo zbog preglednosti.

S obzirom da aplikacija micrOCLaw zahtjeva hardver koji je kompatibilan sa OpenCL tehnologijom, važno je navesti uređaje koji su korišteni pri razvoju i testiranju ove aplikacije. Oba uređaja su proizvodi kompanije Advanced Micro Devices (AMD). Centralna procesorska jedinica nosi naziv AMD FX 8320, baziran je na „Piledriver“ arhitekturi i posjeduje osam jezgara od kojih svako radi na fabričkoj brzini od 3,5 GHz. Ovo znači da ovaj procesor ima osam jednica za obradu koje OpenCL aplikacija može da koristi istovremeno za obradu. Grafička procesorska jedinica koja je korištena je Asus-ova pojačana verzija referentog dizajna modela HD 7990 kompanije AMD. Opremljena je sa 16 jedinica za obradu, 2 GB GDDR5 memorije i brzinom jezgra od 1075 MHz.

Kada smo objašnjavali OpenCL tehnologiju rekli smo da svaki proizvođač uređaja posjeduje svoju implementaciju OpenCL-a, pri čemu ta implementacija mora da bude u skladu sa standardom. S obzirom da je pri razvoju aplikacije micrOCLaw koištem hardver kompanije AMD, takođe su korišteni i alati za razvoj OpenCL aplikacija koji su proizvod ove kompanije. Ovaj paket alata za razvoj se naziva APP SDK (Accelerated Parallel Processing Software Development Kit) i može se besplatno preuzeti sa lokacije

http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/

U ovom paketu se nalaze zaglavlja sa definicijama struktura i funkcija, kao i dinamička i statička biblioteka unutar kojih se nalazi OpenCL implementacija. Nakon preuzimanja i instalacije ovoga paketa potrebno je izvršiti podešavanje projekta u Visual Studio-u. Treba dodati putanje za direktorije OpenCL zaglavlja, kao i direktorije OpenCL biblioteka. Pri tome treba izvršiti povezivanje sa bibliotekom koja odgovara željenoj arhitekturi, a u slučaju 64-bitne aplikacije treba napraviti i konfiguraciju koja cilja ovu arhitekturu. Ovaj proces je relativno jednostavan i na internetu se lako mogu naći članci koji detaljno opisuju ovaj proces, jedan od njih je naveden u spisku korištene literature. Pored ovih neophodnih podešavanja razvojnog okruženja, važno je napomenuti da je veoma poželjno instalirati najnovije drajvere za GPU jer programi na starim drajverima imaju običaj da pucaju bez nekog jasnog razloga.

4.2.1 Korisnički interfejs

Kao što smo već rekli, mogućnostima aplikacije micrOCLaw se pristupa korištenjem elemenata menija. S obzirom da se radi o nativnoj Win32 aplikaciji, u čijem kreiranju nije korišten nikakav radni okvir višeg nivoa, meni je definisan kao resurs aplikacije. Ovaj resurs se pri izvršavanju koristi za kreiranje menija sa odgovarajućim elementima. U našem konkretnom slučaju definicija menija unutar resursa, koja je pojednostavljena zbog uštede prostora, izgleda ovako:

```
IDC_MICROCLAW MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Open",                   IDM_FILE_OPEN
        MENUITEM "&Save",                   IDM_FILE_SAVE
        MENUITEM "E&xit",                   IDM_EXIT
    END
    POPUP "&Image Processing"
    BEGIN
        MENUITEM "&Device Selection",       IDM_DEVICE_SELECTION
        POPUP "&Algorithm"
        BEGIN
            MENUITEM "Arithmetic Mean Filter",      IDM_ARITH_MEAN_FILTER
            MENUITEM "Geometric Mean Filter",       IDM_GEO_MEAN_FILTER
            MENUITEM "Harmonic Mean Filter",        IDM_HARM_MEAN_FILTER
            .                                       .
            .                                       .
            .                                       .
        END
        MENUITEM "A&pply Algorithm",        IDM_APPLY_ALGORITHM
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About ...",              IDM_ABOUT
    END
END
```

Konstante sa prefiksom `IDM_` su definisane u posebnom zaglavlju i koriste se u izvornom kodu kontrolne aplikacije za identifikaciju odgovarajućih elemenata menija. Tako na primjer, slijedeći fragment izvornog koda pretstavlja izvršavanje rutine koja otvara tzv. „File Open“ dijalog pomoću kojega se odabira fotografija sa diska koju želimo da učitamo radi obrade:

```
case IDM_FILE_OPEN:
    g_ImageManager.OpenImage();
    InvalidateRect(hWnd, NULL, TRUE);
    UpdateWindow(hWnd);
    break;
```

Kada je u pitanju učitavanje fotografije sa diska, ova operacija je implemetirana u dvije faze. Prva faza je gore pomenuto prikazivanje dijaloga za odabir i čije objašnjenje dajemo

u vom poglavlju. Ovo je realizovano korištenjem odgovarajućih COM (Component Object Model) klasa i interfejsa. Prvo je izvršena implementacija interfejsa `IFileDialogEvents` u vidu klase `CDialogEventHandler`, kao što možemo vidjeti u slijedećem fragmentu koda u kojem su izostavljene implementacije metoda s obzirom da zauzimaju veliku količinu prostora:

```
class CDialogEventHandler : public IFileDialogEvents
{
public:
    CDialogEventHandler() : _cRef(1) { };

    // Metode interfejsa IUnknown
    IFACEMETHODIMP QueryInterface(REFIID riid, void** ppv);
    IFACEMETHODIMP_(ULONG) AddRef();
    IFACEMETHODIMP_(ULONG) Release();

    // Metode interfejsa IFileDialogEvents
    IFACEMETHODIMP OnFileOk(IFileDialog *);
    IFACEMETHODIMP OnFolderChange(IFileDialog *);
    IFACEMETHODIMP OnFolderChanging(IFileDialog *, IShellItem *);
    IFACEMETHODIMP OnHelp(IFileDialog *);
    IFACEMETHODIMP OnSelectionChange(IFileDialog *);
    IFACEMETHODIMP OnShareViolation(IFileDialog *, IShellItem *,
      FDE_SHAREVIOLATION_RESPONSE *);
    IFACEMETHODIMP OnTypeChange(IFileDialog *pfd);
    IFACEMETHODIMP OnOverwrite(IFileDialog *, IShellItem *,
      FDE_OVERWRITE_RESPONSE *);

private:
    ~CDialogEventHandler() { };
    long _cRef;
};
```

Ova klasa se koristi za rukovanje sa odgovarajućim događajima karakterističnim za dijaloge datoteka, kao što su promjena direktorija i promjena tipa datoteke. Objekat ove klase se dinamički vezuje za objekat koji pretstavlja dijalog, i čiji je identifikator `CLSID_FileOpenDialog`, što se vidi u slijedećem fragmentu:

```
// Kreiraj instancu FileOpenDialog klase
IFileDialog *pfd = NULL;
HRESULT hr = CoCreateInstance(CLSID_FileOpenDialog, NULL,
    CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pfd));
if(SUCCEEDED(hr))
{
    // Kreiraj objekat za obradu događaja
    IFileDialogEvents *pfde = NULL;
    hr = CDialogEventHandler_CreateInstance(IID_PPV_ARGS(&pfde));
    if(SUCCEEDED(hr))
    {
        // Izvrši vezivanje dijaloga i objekta za obradu događaja
        DWORD dwCookie;
```

```
        hr = pfd->Advise(pfde, &dwCookie);
        if(SUCCEEDED(hr))
        {
            // Kod kojim se određuje putanja do izabrane datoteke
            ...

            // Poništi vezivanje dijaloga i objekta za obradu događaja
            pfd->Unadvise(dwCookie);
        }
        pfde->Release();
    }
    pfd->Release();
}
```

Polazeći od ovog objekta, korištenjem niza COM metoda dolazi se do putanje odabrane datoteke. Detaljna implementacija ovog procesa se može vidjeti u izvornom kodu aplikacije. Odabrana datoteka se učitava u objekat klase `Bitmap`, o čemu će puno više biti riječi u poglavlju koje objašnjava implementaciju komponente za rukovanje sa fotografijama.

Obratimo sada pažnju na ostale elemente korisničkog interfejsa. Već smo pomenuli da se može vršiti pregled OpenCL kompatibilnih uređaja na sistemu, kao i odabir onog koji će se koristiti u slučaju da ih ima više od jednog. Ovaj odabir se vrši pomoću modalnog dijaloga koji prikazuje podatke o uređajima i omogućava odabir jednog od njih pomoću odgovarajućeg dugmeta. Osnova ovog dijaloga je definisana kao resurs, dok se dodatni elementi grafičkog interfejsa dinamički iscrtavaju. Ova funkcionalnost je implementirana u vidu klase `CDeviceSelectionDialog`, koja pristupa podacima o uređajima korištenjem metode `COpenCLManager::GetDeviceInfo` i nakon toga prikazuje te podatke unutar klijentskog prostora dijaloga koristeći standardne Win32 funkcije. U slučaju odabira nekog drugog uređaja, poziva se metoda `COpenCLManager::ChangeDevice` koja implementira datu logiku. Detalji implementacije klase `COpenCLManager` će biti dati u poglavlju koje se bavi OpenCL komponentom aplikacije. Pored dijaloga za odabir, imamo i jednostavan „About“ dijalog koji je postao standard u izradi Windows aplikacija u koji pruža osnovne informacije o aplikaciji. Ovaj dijalog je u potpunosti definisan kao resurs i nema dinamičkih elemenata, već se samo učita iz resursa i prikaže.

Kao što smo već pomenuli, fotografija se učitava u objekat klase `Bitmap`, koja pripada interfejsu za programiranje aplikacija zvanom GDI+. Korištenjem ove klase ovo učitavanje je krajnje jednostavno, konstruktoru ove klase proslijedimo putanju do datoteke a za ostatak će se pobrinuti GDI+. Nakon učitavanja fotografije, potrebno je da se odabrana fotografija prikaže u klijentskom prostoru aplikacije, što opet postižemo koristeći klasu iz gore pomenutog interfejsa:

```
bool CImageManager::DrawImage(HDC hDC)
{
    Status status = Ok;

    if(m_Image)
    {
        Graphics graphics(hDC);
        status = graphics.DrawImage(m_Image, 0, 0);
    }
```

```
    return Ok == status;
}
```

Kao što vidimo iz ovog fragmenta koda, i iscrtavanje fotografije je prilično jednostavno, zahvaljujući klasi `Graphics`. Dakle, dovoljno je pozvati ovu metodu kada se obrađuje poruka `WM_PAINT`. Ovim smo završili pregled implementacije grafičkog interfejsa.

4.2.2 Modul za rukovanje sa fotografijama

Kao što se može zaključiti iz prethodnog poglavlja, implementacija ove komponente se uveliko oslanja na interfejs za programiranje aplikacija pod nazivom GDI+. Ovaj interfejs omogućava aplikacijama olakšano korištenje grafičkih elemenata korisničkog interfejsa i posebno formatiranog teksta. Razlog za to je što GDI+ služi kao posrednik između Windows aplikacije i grafičkog hardvera, kao i u tome što je implementiran u objektno-orijentisanom maniru korištenjem klasa.

Za razliku od implemetacije korisničkog interfejsa, čija se implementacija nalazi u resursima i metodama nekoliko različitih klasa, kompletna implementacija ove komponente je obuhvaćena klasom `CImageManager`. Ova klasa posjeduje metode za učitavanje fotografija sa diska u `Bitmap` objekat i njeno eventualno snimanje na disk nakon obrade. Pored toga, posjeduje informacije kao što su dimenzije fotografije i format piksela u memoriji. Kompletna definicija klase je data sa

```
class CImageManager
{
public:
    CImageManager(void);
    ~CImageManager(void);

    void OpenImage();
    bool DrawImage(HDC hDC);
    Gdiplus::Bitmap * GetImage();
    bool ApplySelectedAlgorithm();
    void SaveImage();

private:
    std::string m_PathToFile;
    Gdiplus::Bitmap *m_Image;
    Gdiplus::Bitmap *m_OriginalImage;

    bool m_ImageLoaded;
    bool m_ImageModified;

    unsigned int m_Width;
    unsigned int m_Height;
    unsigned int m_ColorType;
    unsigned int m_BitDepth;

    static const std::vector<std::string> m_SupportedFormats;
```

```
    void LoadImageData();
    int GetEncoderClsid(const WCHAR *format, CLSID *pClsid);
};
```

Već smo diskutovali o implementaciji metoda `OpenImage` i `DrawImage` pa ćemo stoga posvetiti pažnju ostalim metodama. Metoda `GetImage` je trivijalna, kao i metoda `ApplySelectedAlgorithm` koja jednostavno poziva odgovarajuću funkcionalnost iz OpenCL modula. Metoda `SaveImage`, koja snima fotografiju na disk, je realizovana korištenjem COM klasa i interfejsa i pri tome se oslanja na privatnu metodu `GetEncoderClsid` pomoću koje kreira odgovarajuću COM klasu koja pretstavlja enkoder za format fotografije. Ovaj enkoder je ugrađen u operativni sistem i mi ga koristimo da bi snimili fotografiju na disk u željenom formatu. Kao što se može i primjetiti, komponenta za rukovanje sa fotografijama je najednostavnija od svih komponenti aplikacije. Iz toga razloga ćemo preći na diskusiju implementacije OpenCL modula, koja je dosta komplikovanija i pri tome je mnogo više vezana za temu ovoga rada.

4.2.3 OpenCL modul

Poput prethodno opisane komponente, funkcionalnost OpenCL modula je takođe obuhvaćena jednom klasom koja se zove `COpenCLManager`. Kada smo diskutovali dizajn, već smo pomenuli da je korišten šablon dizajna „singleton“ što za posljedicu ima postojanje samo jedne instance ove klase. Definicija ove klase je slijedeća

```
class COpenCLManager
{
public:
    static COpenCLManager& GetInstance();

    bool EnqueueKernel(Gdiplus::Bitmap *pImage);
    void ChangeDevice(unsigned int uiPlatformIndex,
      unsigned int uiDeviceIndex);
    void ChangeKernel(int iAlgorithmMenuId);
    VECTOR_CLASS<std::string> GetDeviceInfo();

private:
    VECTOR_CLASS<cl::Platform> m_vPlatforms;
    VECTOR_CLASS<cl::Device> *m_pAllDevices;
    VECTOR_CLASS<cl::Kernel> m_Kernels;

    cl::Context m_Context;
    cl::Device m_CurrentDevice;
    cl::Program::Sources m_ProgramSource;
    cl::Program m_Program;
    cl::CommandQueue m_CommandQueue;

    unsigned int m_uiCurrentPlatform;
    unsigned int m_uiCurrentKernel;

    void InitializePlatforms();
```

```
    void BuildProgramAndKernels();

    COpenCLManager(void);
    COpenCLManager(const COpenCLManager &);
    COpenCLManager(const COpenCLManager &&);
    COpenCLManager& operator=(const COpenCLManager &);
    COpenCLManager& operator=(const COpenCLManager &&);

    ~COpenCLManager();
};
```

Kao što vidimo, svi mogući načini konstrukcije objekta klase su nedostupni i jedini način prustupa objektu je putem statične metode `GetInstance` koja vraća referencu na jedinu instancu ove klase. Ta instanca se kreira kada se ova metoda prvi put pozove za vrijeme izvršavanja programa. Pri kreiranju instance se izvršava inicijalizacija fundamentalnih OpenCL struktura, što je traje dugo zbog ispitivanja hardvera na sistemu i kompilacije jezgara za specifičan uređaj. Očigledno je da želimo da se ova inicijalizacija obavi što prije pa iz tog razloga na samom startu aplikacije zovemo jednom metodu `GetInstance` da bi se taj proces tad obavio. Pri inicijalizaciji se prvo obavlja ispituje postojanje kompatibilnih uređaja na sistemu. U slučaju da ne postoji nijedan kompatibilan uređaj, korisnik se obavještava o tome i aplikacija se sama zatvara jer ne može da ispravno funkcioniše. U suprotnom, kreiraju se strukture za svaku dostupnu platformu i sve njihove uređaje pri čemu se za podrazumjevani odabrani uređaj uzima prvi GPU uređaj iz prve platforme. Potom se vrši kreiranje odgovarajućeg konteksta i komandnog reda za odabrani uređaj nakon čega se prelazi u drugu fazu inicijalizacije unutar koje se kompilira kod jezgara za odabrani uređaj. Metode `InitializeOpenCLStructures` i `BuildProgramAndKernels` obuhvataju proces inicijalizacije i pregledom njihovog izvornog koda se može dobiti detaljan uvid u detalje implementacije koja se u potpunosti oslanja na osnove programiranja kontrolne aplikacije koje su opisane u prethodnoj glavi.

Pređimo sada na javne metode. Metoda `ChangeDevice` omogućava promjenu uređaja koji se koristi za obradu podataka, u slučaju da se ova akcija odabere putem korisničkog menija. Ona kao parametre prima identifikatore uređaja i platforme kojoj taj uređaj pripada i koristi ih da promjeni postojeće. Kao što možemo vidjeti na slijedećem fragmentu izvornog koda, pri promjeni uređaja mora se izvršiti i kreiranje novog konteksta i komandnog reda za novi uređaj.

```
m_uiCurrentPlatform = uiPlatformIndex;
m_CurrentDevice = m_pAllDevices[m_uiCurrentPlatform][uiDeviceIndex];

cl_int error = CL_SUCCESS;
m_Context = Context(m_CurrentDevice);
m_CommandQueue = CommandQueue(m_Context, m_CurrentDevice, 0, &error);
```

Slijedeća javna metoda, zvana `ChangeKernel`, je još jednostavnija. Ona se poziva kada korisnik promjeni algoritam za obradu fotografije putem menija, pri čemu tada jednostavno podesi indeks odabranog jezgra na indeks jezgra koji implementira dati algoritam. Došli smo do poslednje javne metode klase `COpenCLManager`, pod nazivom `EnqueueKernel`, koja je najbitnija pa je iz tog razloga njena implementacija i najkomplikovanija. Ova kompleksnost leži u činjenici da se implementacija ove metode oslanja na metodu klase , kao i veći dio OpenCL programiranja koje smo opisali u prethodnoj glavi. Posmatrajmo definiciju ove metode

```
bool COpenCLManager::EnqueueKernel(Gdiplus::Bitmap *pImage)
{
    Gdiplus::BitmapData bitmapData;
    int width = pImage->GetWidth();
    int height = pImage->GetHeight();
    Gdiplus::Rect rect(0, 0, width - 1, height - 1);
    Gdiplus::Status status = pImage->LockBits(&rect,
        Gdiplus::ImageLockModeWrite, PixelFormat32bppARGB, &bitmapData);

    if(Gdiplus::Ok == status)
    {
        unsigned char *pixels = (unsigned char *)bitmapData.Scan0;

        int error = 0;
        try
        {
            ImageFormat format(CL_RGBA, CL_UNSIGNED_INT8);
            Image2D inputImage(m_Context,
                CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, format,
                width, height, 0, reinterpret_cast<void *>(pixels),
                &error);
            Image2D outputImage(m_Context, CL_MEM_WRITE_ONLY, format,
                width, height, 0, NULL, &error);
            error = m_Kernels[0].setArg(0, inputImage);
            error = m_Kernels[0].setArg(1, outputImage);

            NDRange global(height, width);
            cl::size_t<3> origin, region;
            origin[0] = origin[1] = origin[2] = 0;
            region[0] = width;
            region[1] = height;
            region[2] = 1;
            error = m_CommandQueue.enqueueNDRangeKernel(m_Kernels[0],
                NullRange, global, NullRange, NULL, NULL);
            error = m_CommandQueue.enqueueReadImage(outputImage, CL_TRUE,
                origin, region, 0, 0, reinterpret_cast<void *>(pixels));
            error = m_CommandQueue.finish();
        }
        catch(cl::Error clError)
        {
            std::string messageText("Error while trying to execute" \
                "kernel!\nDescription: ");
            messageText += std::string(clError.what()) + "\n";
            MessageBox(NULL, messageText.c_str(), "Error Box",
                MB_ICONERROR | MB_OK);
        }
        status = pImage->UnlockBits(&bitmapData);
    }

    return Gdiplus::Ok == status;
```

```
}
```

Prva stvar koju možemo primjetiti je poziv metode `LockBits`, kojom dobijamo pristup podacima piksela iz objekta koji pretstavlja fotografiju. Nakon toga kreiraju se dva memorijska spremnika za dvodimenzionalne grafičke podatke, jedan za ulazni argument jezgra (koji se kreira korištenjem podataka o pikselima koje smo dobili u prethodnom koraku) i jedan za izlazni. Nakon podešavanja ovih objekata kao parametara jezgra, podešava se broj globalnih radnih predmeta koji će se izvršiti. Kao što je to čest slučaj pri implementaciji manje komplikovanih algoritama za obradu fotografija, za broj radnih predmeta po prve dvije dimenzije uzimamo širinu i visinu fotografije a određivanje veličine radne grupe ostavljamo da podesi izvršno okruženje za OpenCL. Nakon ovih podešavanja, u komandni red postavljamo naredbe za izvršavanje jezgra i čitanje drugog parametra jezgra koji pretstavlja rezultat. Konačno, zovemo još jednu GDI+ funkciju zvanu `UnlockBits` koja ažurira promjene u podacima piksela objekta klase `Bitmap`.

4.2.4 Jezgra

Prodiskutujmo na kraju i implementaciju jezgara. Već smo ranije pominjali da se kod jezgara može držati u odvojenim datotekama i učitati u kontrolnu aplikaciju iz istih prije kompilacije OpenCL programa za konkretan uređaj. Da bi pojednostavili implementaciju aplikacije micrOCLaw, izvorni kod jezgara je smješten u objektima C++ klase `std::string`. S obzirom da sva naša jezgra pretstavljaju implementacije algoritama za obradu fotografija, njihov izvorni kod može biti pretstavljen slijedećim šablonom

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
    CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_LINEAR;

__kernel void algorithm(read_only image2d_t sourceImage,
    write_only image2d_t outputImage)
{
    int2 coord = (int2)(get_global_id(1), get_global_id(0));
    uint4 pixel = read_imageui(sourceImage, sampler, coord_up_left);
    ...

    uint red_channel = ...
    uint green_channel = ...
    uint blue_channel = ...

    pixel.xyzw = (uint4)(red_channel, green_channel, blue_channel,
        pixel.w);
    write_imageui(outputImage, coord, pixel);
}
```

Primjetimo da korištenjem ugrađene funkcije `get_global_id` lako pronalazimo koordinate piksela čiju modifikovanu vrijednost trebamo da izračunamo u odnosu na dati algoritam i upišemo u izlazni `image2d_t` objekat. Ova jednostavnost je razlog zašto je uglavnom u slučaju obrade fotografija, ili frejmova u video datoteci, broj radnih predmeta po prve dvije dimenzije jednak dimenzijama te fotografije, odnosno frejma.

Tri tačke nakon prve dvije linije koda u gornjem šablonu označavaju da se tu

pristupa dodatnim vrijednostima potrebnim da bi se izvršila izračunavanja predviđena algoritmom. U većini slučajeva, ove dodatne vrijednosti su podaci okolnih piksela da bi se izračunale vrijednosti poput geometrijske sredine i sl. Nakon toga, dolazi računanje vrijednosti crvene, zelene i plave boje u rezultujućem pikselu primjenjujući formulu algoritma. Konačno, korištenjem ugrađene funkcije `write_imageui`, vrši se upisivanje vrijednosti rezultujućeg piksela u izlazni argument jezgra. Pregledom izvornog koda jezgara odmah se primjećuje da za pristupanje okolnim pikselima nisu korištene petlje, iako bi one bile pogodne za to, čime bi kod bio sažetiji i lakši za razumjevanje. Razlog za to je ranije pominjana osobina grafičkih procesora da daju lošije performanse kada moraju da vrše ispitivanje uslova i grananje. Diskusijom implementacije jezgara smo došli do kraja ove glave.

5 Zaključak

Programiranje sa visokim performansama se koristi u raznim oblastima u kojima se zahtjeva obrada enormnih količina podataka, kao što su: kvantna mehanika, vremenska prognoza, istraživanje klimatskih promjena, molekularno modelovanje (vršenje proračuna o strukturi hemijskih jedinjenja, bioloških markomolekula, polimera i kristala), simulacije u fizici (simuliranje ranih faza svemira, simuliranje aerodinamičkih osobina aviona i svemirskih letjelica, simulacije eksplozija nuklearnih bombi i nuklearne fisije), itd. U poslednjih nekoliko godina sve češće koriste grafičke procesorske jedinice za ovu vrstu programiranja. Iz tog razloga je viđena pojava nekoliko različitih radnih okvira koji omogućavaju GPGPU programiranje, kao što su C++ AMP, DirectCompute, CUDA i OpenACC. Mi smo smo u ovom radu pretstavili jednu alternativu prethodno navedenim radnim okvirima koja se ističe sa nekoliko prednosti, pri čemu su najvažnije od od njih mogućnost obrade na uređajima koji nisu GPU, kao i nezavisnost od arhitekture i proizvođača uređaja koji se koristi za obradu.

Konkretna demonstracija je implemetirana u vidu aplikacije micrOCLaw koja omogućava obradu digitalnih fotografija. Razlog za to je činjenica da je obrada podataka grafičkog tipa postala veoma čest zadatak u zadnjih nekoliko godina. S obzirom na brzinu današnjih centralnih procesorskih jedinica, implementacija algoritama za obradu fotografija na njima je dovoljno brza da se prebacivanje na implementaciju koja koristi paralelnu obradu na GPU ne isplati s obzirom da se razlika u brzini ne osjeti u radu aplikacije. Međutim, ova razlika se osjeti kada su u pitanju video datoteke koje su sastavljene od velikog broja frejmova koji su slični fotografijama. Na primjer, aplikacija ImTOO Video Converter, pri konverziji video datoteke iz jednog formata u drugi, dostiže petostruko ubrzanje ako na mašini na kojoj je pokrenuta ima dostupan grafički procesor kompatibilan sa CUDA ili AMD APP tehnologijom. Ovo znači da bi implementacija konverzije video datoteka bila puno bolja demonstracija OpenCL tehnologije, međutim ovo je veoma komplikovanije za realizovati i drastično izlazi iz okvira jednog diplomskog rada.

Sama aplikacija je napisana tako da bi bilo pogodno dodati proširenja postojeće funkcionalnosti. Neka od njih su dodavanje tulbara koji sadrži neke od opcija iz menija, da bi korištenje aplikacije bilo lakše. Takođe bi se mogle mapirati određene kombinacije tastera za određenu funkcionalnost, kao i implemetacija skrolovanja koja je potrebna za fotografije velikih dimenzija.

Korištena literatura

[1] Matthew Scarpino, OpenCL in Action (2012), Addison-Wesley Professional
[2] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki, The OpenCL Programming Book (2010), Fixstars Corporation
[3] theForger's Win32 API Programming Tutorial, http://www.winprog.org/tutorial/
[4] MSDN, Common Item Dialog, http://msdn.microsoft.com/en-us/library/windows/desktop/bb776913(v=vs.85).aspx
[5] MSDN, Retrieving the Class Identifier for an Encoder, http://msdn.microsoft.com/en-us/library/windows/desktop/ms533843(v=vs.85).aspx
[6] Chris McCabe, Setting up OpenCL in Visual Studio, http://kode-stuff.blogspot.com/2012/11/setting-up-opencl-in-visual-studio_1.html
[7] Xue-Jun Yang, Xiang-Ke Liao, Kai Lu, Qing-Feng Hu, Jun-Qiang Song, Jin-Shu Su, The TianHe-1A Supercomputer: Its Hardware and Software (maj, 2011), Journal Of Computer Science And Technology
[8] Experience 5X faster speed with NVIDIA CUDA & AMD APP, http://www.imtoo.com/cuda.html