Univerza v Ljubljani
Fakulteta za računalništvo in informatiko ter
Fakulteta za matematiko in fiziko

Blaž Tomažič

Orodje za testiranje implementacij standarda OpenCL

diplomsko delo
na interdisciplinarnem univerzitetnem študiju
računalništva in matematike

Mentor: doc. dr. Matija Marolt

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00051 / 2013
Datum: 5.9.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdajata naslednjo nalogo:

Kandidat: **BLAŽ TOMAŽIČ**

Naslov: **ORODJE ZA TESTIRANJE IMPLEMENTACIJ STANDARDA OPENCL**
A FRAMEWORK FOR TESTING OPENCL IMPLEMENTATIONS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V okviru diplomske naloge preučite standard za porazdeljeno računanje OpenCL in izdelajte orodje za testiranje implementacij standarda OpenCL. Orodje naj omogoča hitro pisanje testov in enostavno razširljivost z novimi tipi testov. Podpira naj tudi združitev z orodjem Piglit za avtomatsko izvajanje skupin testov in vizualizacijo rezultatov.

Mentor:

doc. dr. Matija Marolt

Fakulteta za računalništvo in informatiko - dekan:

prof. dr. Nikolaj Zimic

Fakulteta za matematiko in fiziko - dekan:

prof. dr. Anton Ramšak

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Blaž Tomažič, z vpisno številko **63060249**, sem avtor diplomskega dela z naslovom:

Orodje za testiranje implementacij standarda OpenCL

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matije Marolta,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 12. marca 2014 Podpis avtorja:

Zahvaljujem se vsem, ki so mi ob študiju in pisanju diplomskega dela stali ob strani. Predvsem bi se zahvalil staršem, ki so me podpirali skozi celoten študij; Matiji in Niki, ki sta me sprostila po napornih tednih; Arneli, ki je skrbela zame in za mojo delavnost; ter sošolcem in prijateljem, ki so študirali z mano in ob meni.

Nenazadnje gre zahvala tudi mentorju doc. dr. Matiji Maroltu za pomoč in svetovanje pri izdelavi diplomskega dela ter Tomu Stellardu in skupnosti X.Org za usmerjanje pri implementaciji testnega orodja.

Staršem.

Kazalo

Povzetek

Diplomsko delo obravnava implementacijo in uporabo testnega orodja za testiranje implementacij standarda OpenCL. Standard definira vmesnik API, izvajalno okolje in programski jezik OpenCL C, ki skupaj poenotijo programiranje vseh procesorjev, podprtih s strani implementacije. Implementacija tako obsežnega standarda je zahtevna in nagnjena k napakam, zato je zanjo potrebno primerno testno orodje.

Testno orodje, napisano v sklopu diplomskega dela in sponzorstva skupnosti X.Org, predstavlja odprtokodno rešitev opisanega problema. Razvito orodje omogoča hitro pisanje testov in je enostavno razširljivo. Uvede izvajalca in tipe testov, ki glede na konfiguracijske nastavitve posameznega testa postavijo primerno izvajalno okolje OpenCL in v njem poženejo test. Testi, napisani za orodje, so minimalni in razdeljeni na konfiguracijsko strukturo za postavljanje izvajalnega okolja in testno funkcijo, ali testni program OpenCL C, za testiranje potrebne funkcionalnosti, navedene v standardu. Z dodajanjem novih tipov testov je mogoče orodje poljubno razširiti. Najbolj napreden del orodja omogoča implicitno pisanje testov za programski jezik OpenCL C, kjer se konfiguracijska struktura in test nahajata v izvorni datoteki OpenCL C. Združitev testnega orodja s testnim orodjem Piglit doda avtomatsko izvajanje skupin testov in grafični prikaz rezultatov.

Ključne besede: OpenCL, testno orodje, odprta koda, razvoj, Piglit

Abstract

The thesis discusses implementation and use of a testing tool for testing implementations of the OpenCL standard. The standard defines an API interface, runtime environment and OpenCL C programming language, which together standardize programming of all processors supported by the implementation. Implementation of such a comprehensive standard is difficult and prone to errors, and therefore requires a suitable test tool.

The testing tool written in the context of the thesis and sponsorship of the X.Org community represents an open-source solution of the problem. The developed tool enables quick writing of tests and is easily extendable. It introduces a test runner and types of tests which, depending on the configuration of a test, set a suitable OpenCL runtime environment and execute a test in it. Tests written for the tool are minimal and divided into a configuration structure for setting a runtime environment and a test function, or OpenCL C test program, for testing required functionality specified in the standard. With the addition of new types of tests the tool can be arbitrarily extended. The most advanced part of the tool allows implicit writing of tests for programming language OpenCL C where the configuration structure and the test are located in the OpenCL C source file. Merging of the testing tool with the Piglit testing tool adds automatic execution of groups of tests and graphical presentation of results.

Keywords: OpenCL, testing tool, open-source, development, Piglit

Poglavje 1

Uvod

V zadnjem desetletju smo doživeli porast heterogenih sistemov oziroma sistemov z različnimi vrstami procesorjev. Takšni sistemi vsebujejo splošnonamenske procesorje, kot je centralna procesna enota (CPE), in namenske, kot sta grafična procesna enota (GPE) in enota za procesiranje signalov (DSP). Uporaba posebnih procesorjev ali skupine procesorjev omogoča vzporedno izvajanje programov, vendar izvedba ni enostavna, saj sta za programiranje vsakega potrebna drugačno programsko orodje in okolje.

Rešitev predstavlja standard neprofitne organizacije Khronos, imenovan Open Computing Language (OpenCL), ki ponuja univerzalni vmesnik za programiranje in izvajanje programov na vseh procesorjih na sistemu. Programi so napisani v posebnem jeziku in okolju, kjer se lahko izvajanje delov programa porazdeli po procesorjih in tudi po samih računskih enotah posameznega procesorja. S tem pridobimo na hitrosti izvajanja in izkoristku računske moči celotnega sistema. Vendarle je standard zajeten, zato ga ni enostavno implementirati. Zato potrebujemo testno orodje, da nam pomaga pri razvoju in testiranju pravilnega delovanja takšnih implementacij.

Orodje, napisano v sklopu diplomskega dela, rešuje natanko ta problem, torej iskanje in odkrivanje napak v implementacijah standarda OpenCL. Orodje je napisano tudi v sklopu organizacije X.Org, ki je potrebovala odprtokodno rešitev za omenjeni problem. Z orodjem želimo omogočiti enostavno

in hitro pisanje testov, razširljivost pripadajočega ogrodja in pregleden prikaz rezultatov testiranj.

Na začetku diplomskega dela je predstavljeno ozadje teme, ki je potrebno za razumevanje rešitve problema. Opisan je standard OpenCL, predstavljena je uporabljena programska oprema in definirani so cilji orodja. Jedro dela predstavlja natančen opis implementacije in uporabe orodja. Obdelano je ogrodje za pisanje testov, uporaba orodja Piglit za izvajanje testov in struktura različnih vrst testov. Na koncu analiziramo rezultate testiranj, dobljene z uporabo implementiranega orodja, in ocenimo vpliv orodja na razvoj trenutnih implementacij standarda.

Poglavje 2

Ozadje

Pisanje programov, še posebno zajetnih programov, ni enostavno in velikokrat pride do napak, ki privedejo do napačnega delovanja programa. Te napake so lahko posledica težavnosti implementacije, tipkarske napake, nepozornosti programerja ali česarkoli drugega. Zato so zaradi zmanjšanja možnosti dodajanja novih napak v program in samega odpravljanja napak nastala testna orodja. Z njimi lahko enostavno testiramo pravilno delovanje programa ali posameznih delov programa. Ko je delovanje programa ustrezno definirano oziroma dokumentirano, lahko napišemo testno orodje in teste pred samo implementacijo programa.

Testno orodje in testi, implementirani za diplomsko delo, testirajo skladnost implementacij s standardom OpenCL, zato je za razumevanje programa potrebno dobro poznavanje standarda. V tem poglavju si natančneje pogledamo delovanje standarda OpenCL ter razložimo pojme pomembne za implementacijo in uporabo testnega orodja.

Potreba po testnem orodju je nastala v organizaciji X.Org [3], ki skrbi za razvoj proste in odprtokodne programske opreme za grafične sisteme. Na aktivnih projektih pod okriljem te organizacije dela veliko ljudi, zato je potrebno za sodelovanje slediti sprejetim standardom za delo v prostih in od-

prtokodnih projektih.

2.1 Prosta in odprtokodna programska oprema

Prosta in odprtokodna programska oprema (FOSS) označuje programe, ki so distribuirani skupaj s svojo izvorno kodo in pravico za poganjanje, učenje, spreminjanje in deljenje programa ter izvorne kode. Oba pojma, prosta programska oprema in odprtokodna programska oprema, v večini primerov predstavljata isto stvar, razlikujeta se le v filozofiji. Prosta programska oprema se osredotoča na uporabnikove pravice pri uporabi programa, medtem ko se odprtokodna programska oprema osredotoča na pravice lastnika avtorskih pravic programa, ki je izbral ta razvojni model zaradi poslovne praktičnosti.

V knjigi The Cathedral and the Bazaar [9] avtor Eric S. Raymond predstavi dva različna razvojna tipa programov v skupnosti FOSS. Prvi model se imenuje *katedrala* (cathedral), kjer razvoj poteka za zaprtimi vrati in je izvorna koda dostopna samo po izdaji nove verzije programa. Primer uporabe takšnega modela je mobilna platforma Android. Pri drugemu, ki se imenuj *bazar* (bazaar), pa razvoj poteka javno in je zadnja verzija izvorne kode vedno dostopna. Takšen razvojni model uporabljata Linux in testno orodje Piglit.

V okolju FOSS obstaja več različnih licenc, ki se uporabljajo za licenciranje programov. Glavne med njimi so licenca GNU General Public License (GPL), licenca BSD in licenca MIT. Program, implementiran za to diplomsko nalogo, uporablja licenco MIT, saj je del testnega orodja Piglit, napisanega pod isto licenco.

2.1.1 X.Org

Organizacija X.Org usklajuje, usmerja in sponzorira delo na prosto in odprtokodnih projektih, povezanih z grafičnimi sistemi. Primeri večjih projektov

sta okenska sistema X.Org[1] in Wayland[2] ter grafična knjižnica Mesa[3]. Razvoj vseh programov poteka javno in v skladu s filozofijo FOSS. Pri tem sodelujejo podjetja, kot so Advanced Micro Devices (AMD), Intel in Nvidia.

Podjetje AMD med ostalimi projekti sodeluje tudi pri pisanju odprtokodne verzije gonilnikov za njihove grafične kartice. Poleg ostalih standardov razvijalci delajo tudi na podpori za standard OpenCL. Ker v skupnosti X.Org niso imeli nobenega orodja za testiranje implementacij standarda OpenCL, je tako nastala potreba za razvoj takšnega orodja.

X.Org Endless Vacation of Code (EVoC) je učni program za financiranje in mentorstvo študentov za delo na projektih pod okriljem organizacije X.Org. Glavni namen programa je iskanje in uvajanje študentov za delo na projektih FOSS, da bi v prihodnosti lažje prispevali in sodelovali na izbranem projektu. Razširitev testnega orodja Piglit, s podporo za testiranje implementacij standarda OpenCL, je bila ena od predlaganih rešitev za ta program in je v sklopu tega tudi implementirana.

2.2 Testna orodja

Testna orodja se uporabljajo pri vsakem večjem projektu. Ob pravilni uporabi imamo zagotovljeno kvaliteto kode, lahko se izognemo regresijam v prihodnosti, nadzorujemo skladnost z dokumentacijo, itd. Načinov in ciljev uporabe testnih orodij je mnogo. Vsem pa je skupno, da so napisana z namenom odkrivanja napak v produktu, torej da lahko napake uspešno najdemo in odpravimo.

Obstajata dve vrsti testiranj, **statično** in **dinamično**. Statično testiranje izvaja človek s pregledom kode, medtem ko dinamično testiranje izvaja zunanji program z množico testnih primerov. Tu se bomo osredotočili na

[1] `http://www.x.org/`
[2] `http://wayland.freedesktop.org/`
[3] `http://mesa3d.org/`

dinamično testiranje, ker je naše testno orodje tega tipa.

Glede na način dinamičnega testiranja programa ločimo:

Testiranje črne skrinjice: Program testiramo brez poznavanja notranjega delovanja. Teste pišemo z namenom, da pokrijemo vse možne uporabe programa.

Testiranje bele skrinjice: Program poznamo in imamo vpogled v njegovo notranje delovanje. Teste pišemo z namenom, da izvedemo in testiramo vse izvajalne poti v programu.

Testiranje sive skrinjice: Program poznamo in imamo vpogled v njegovo notranje delovanje. Teste pišemo z namenom, da pokrijemo vse načine uporabe programa, ampak hkrati tudi testiramo vse notranje izvajalne poti.

Ker mora naše testno orodje delovati na vseh implementacijah standarda OpenCL in ker poznamo samo pričakovano delovanje programa, izvajamo z njim testiranje *črne skrinjice*.

Program lahko testiramo na različnih ravneh:

Testiranje enot: Ločeno testiramo posamezne dele programa. Navadno se to izvaja na ravni funkcij.

Testiranje integracije: Testiramo medsebojno delovanje modulov. Tu se testirajo vmesniki in medsebojna komunikacija.

Testiranje sistema: Testiramo delovanje programa kot celote oziroma končnega produka.

Naše testno orodje se uporablja na ravni *testiranja sistema*, ker preverjamo delovanje celotne implementacije glede na standard. Poleg tega se uporablja tudi na ravni *testiranja enot*, saj so testi zelo specifični in testirajo posamezne zelo majhne dele standarda, da s tem lažje lociramo napake pri razvoju

implementacije.

Obstaja tudi veliko testnih tipov, ki opisujejo testne programe, vendar jih tu zaradi preobsežnosti ne bomo našteli. Pomembno je vedeti le, da naše testno orodje spada med naslednja orodja:

Skladnostna: Testira se skladnost programa z njegovo dokumentacijo oziroma v našem primeru s standardom.

Razvojna: Teste se uporablja in piše med razvojem programa.

Regresijska: Skrbimo, da večje spremembe v programu ne privedejo do napak v delih programa, kjer je že deloval pravilno.

2.3 Standard OpenCL

Na trgu se je pred desetletjem začelo v računalniški industriji večati število naprav, ki so sposobne vzporedno izvajati različne računske naloge. Ker so te naprave namensko in fizično različne med seboj in se programski jeziki in orodja za delo z njimi razlikujejo, je nastala potreba po enotnem orodju za pisanje programov, ki se lahko brez sprememb izvajajo na vseh teh heterogenih sistemih. Rešitev tega problema predstavlja standard OpenCL.

Standard OpenCL [5] je odprt brezplačen standard za večnamensko vzporedno programiranje heterogenih sistemov, ki so sestavljeni iz procesorjev tipov CPE, GPE in ostalih. Razvija ga organizacija Khronos, ki med drugim skrbi za standarde, kot sta OpenGL in OpenVG. Prva verzija standarda OpenCL je bila izdana leta 2008 [4]. Standard je napisan z namenom, da industriji ponudi standardiziran način komunikacije s strojno opremo za izvajanje vzporedno pisanih programov. Proizvajalci strojne opreme morajo za skladnost s standardom implementirati izvajalno okolje in njegove vhodne točke.

Na drugi strani lahko uporabniki standarda izvajajo programe, napisane za okolje OpenCL, na vsaki strojni opremi, ki ima implementirano takšno izvajalno okolje. Primeri uporabe segajo od osebne, kot sta obdelava slik (GIMP[4]) in računanje bitcoinov (bfgminer[5]), do poslovne, kot je računanje vremenske napovedi [11]. Implementacije standarda obstajajo za vse pospeševalne naprave večjih podjetij (AMD, Intel, Nvidia). Konkurenčni standardi s podobno funkcionalnostjo so CUDA [6], OpenACC in OpenMP.

Standard OpenCL definira *programski vmesnik API (API)*, *izvajalno okolje* in *programski jezik OpenCL C*. Vmesnik API skupaj z izvajalnim okoljem uporabniku omogoča izbiro platforme OpenCL, upravljanje s podprtimi računskimi napravami ter prevajanje in izvajanje programov na njih. Programi za računske naprave so napisani v jeziku OpenCL C, okleščeni različici programskega jezika C z razširitvami za pisanje vzporednih programov.

Sledi bolj podrobna razlaga delovanja standarda OpenCL in programskega jezika OpenCL C. Prevedeni pojmi, ki bodo tukaj uporabljeni, so pomembni za razumevanje testnega orodja, zato so poleg njih v oklepajih podani izvirni angleški izrazi, ker se kot takšni uporabljajo v izvorni kodi diplomskega dela.

2.3.1 Delovanje

V tem poglavju so predstavljeni konceptualni modeli standarda OpenCL [7], predstavijo nam, kako so porazdeljene naprave in pomnilnik, kako se programi izvajajo ter na kakšne načine lahko programiramo probleme za vzporedno izvajanje.

[4] `http://www.gimp.org/`
[5] `http://bfgminer.org/`

Platforma

Standard OpenCL omogoča vzporedno namestitev več različnih implementacij na istem sistemu. Vsaka implementacija lahko uporabniku predstavi eno ali več **platform** na katerih izvajamo programe OpenCL. Platforma (slika 2.1) je sestavljena iz naslednjih logičnih delov (omenjeni so tudi tipi pomnilnika, ki so razloženi v nadaljevanju):

Gostitelj (host): Naprava, ki upravlja in poganja izvajalno okolje OpenCL. Skrbi za komunikacijo med ostalimi deli aplikacije in platformo OpenCL. Gostitelj je vedno samo *en*, navadno je to CPE sistema, in lahko spada tudi med *računske naprave*.

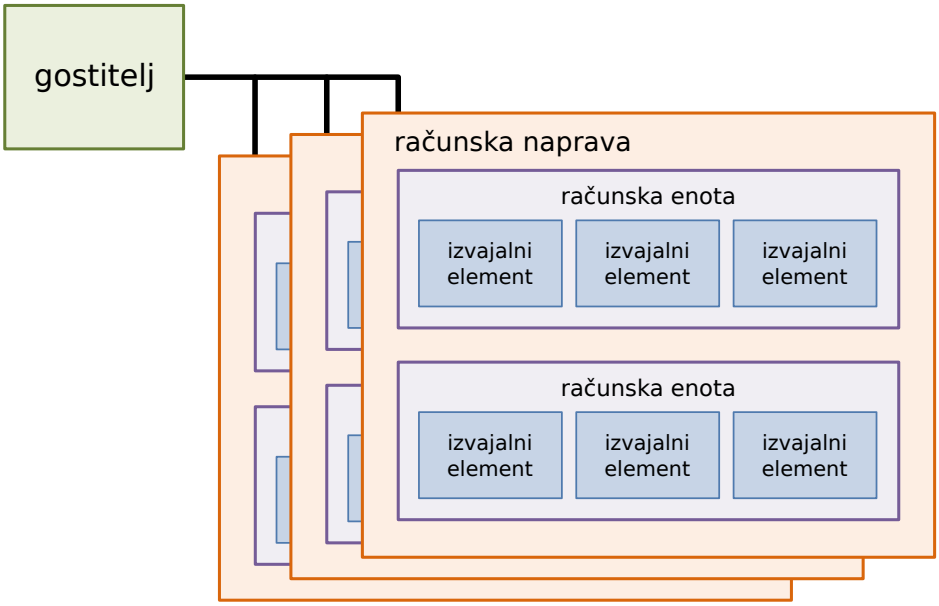
Računske naprave (OpenCL devices ali compute devices): Gostitelj je povezan na eno ali več računskih naprav. Na njih se izvajajo prevedeni programi OpenCL C in vsaka naprava ima svoj *globalni pomnilnik*. Navadno so te procesorji CPE, GPE in ostali, ki jih platforma podpira.

Računske enote (compute units): Računske naprave so razdeljene na več računskih enot. To so deli naprav, ki lahko samostojno izvajajo programe in si z ostalimi računskimi enotami na napravi delijo *lokalni pomnilnik*. Navadno so to jedra na procesorjih CPE in tokovni procesorji na procesorjih GPE.

Izvajalni elementi (processing elements): Računske enote so nadalje razdeljene na izvajalne elemente. To so deli računskih enot, ki imajo svoj ločen *privatni pomnilnik*. Navadno so to tokovne enote v tokovnih procesorjih (na procesorjih GPE), medtem ko je na procesorjih CPE vsaka računska enota tudi sama sebi izvajalni element.

Pomnilniški model

Pomnilniška hierarhija je predstavljena na sliki 2.2. Kot vidimo ima gostitelj svoj **pomnilnik gostitelja** in računska naprava ima svoj ločen pomnilnik, ki



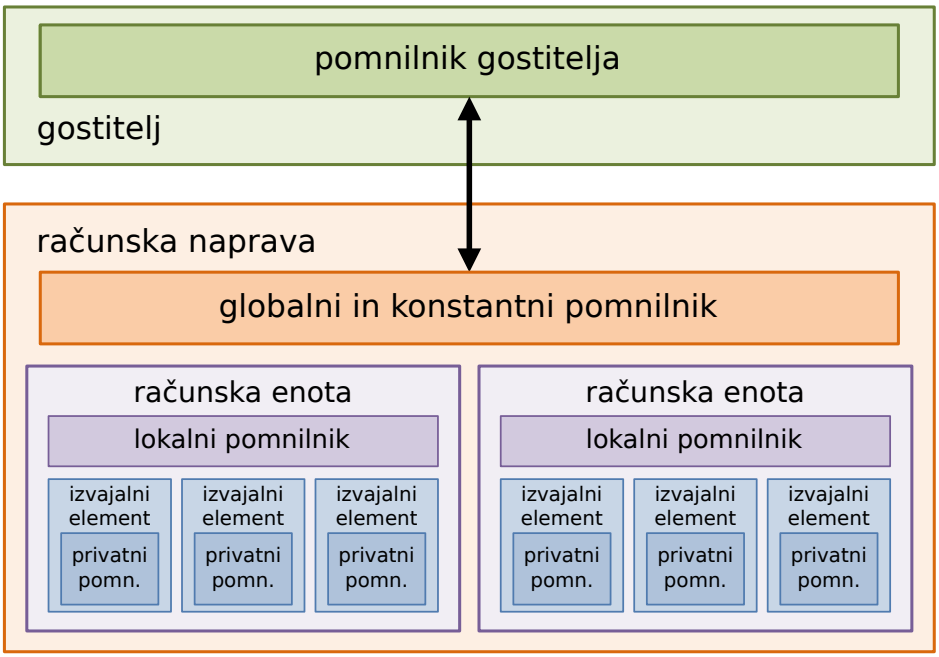
Slika 2.1: Model platforme OpenCL

je dodatno razdeljen. Prenos podatkov med gostiteljem in računsko napravo je mogoč le z klici vmesnika API in to le med pomnilnikom gostitelja in globalnim pomnilnikom računske naprave.

Izvajanje programa na računskih napravah poteka na izvajalnih elementih, zato bomo pogledali, kako izvajalni elementi dostopajo do pomnilnika. Vsak izvajalni element ima svoj **privatni pomnilnik**, ki ni dostopen ostalim elementom. **Lokalni pomnilnik** si delijo vsi izvajalni elementi v računski enoti in **globalni pomnilnik** vsi na računski napravi. Prenos podatkov med tipi pomnilnikov na računski napravi je mogoč le znotraj programa OpenCL C.

Izvajalni model

Sedaj ko poznamo porazdelitev naprav in pomnilnika, si lahko pogledamo, kako na njih izvajamo dejanske programe. Ko si izberemo eno izmed ponujenih *platform* in na njej izberemo *računske naprave*, na katerih želimo izvajati programe, ustvarimo z njimi **kontekst** (**context**) OpenCL. V tem kontekstu lahko ustvarjamo objekte OpenCL, katere si delijo vse naprave v kontekstu.



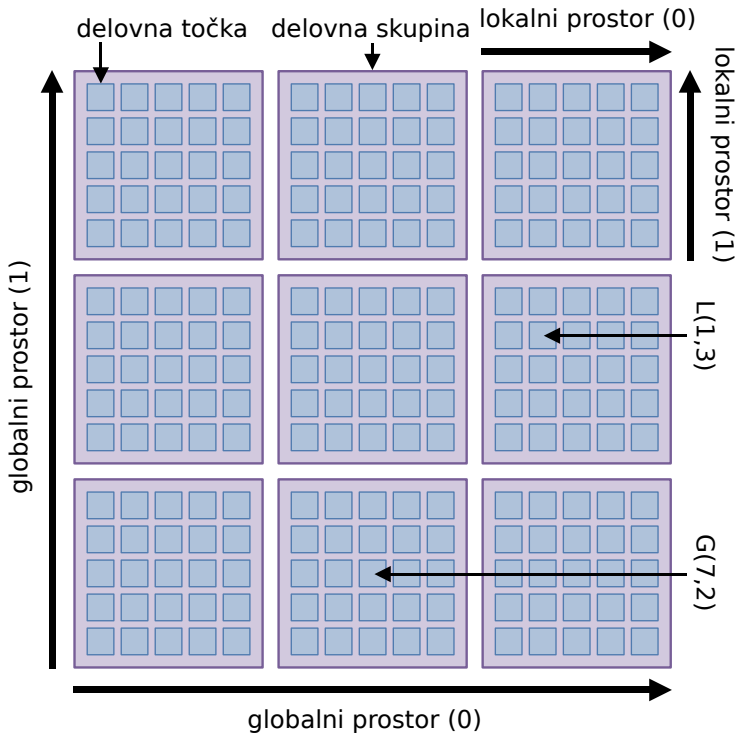
Slika 2.2: Pomnilniški model OpenCL

Eden takšnih objektov je **program** (**program**), kar je v standardu OpenCL nekakšen sinonim za *dinamično knjižnico*. Predstavlja skupek izvorne kode, preveden za vse naprave v kontekstu in vsebuje vhodne funkcije, imenovane **jedra** (**kernels**). Jedra predstavljajo funkcije, označene s ključno besedo **`kernel`**, katere lahko gostitelj požene na računski napravi. Torej če želimo nekaj izvajati na računski napravi, moramo s klici vmesnika API določiti, katero jedro želimo pognati in kakšne argumente mu podamo.

Argumenti so lahko enostavnega tipa, kot sta **`int`** in **`float`**, ali pa poseben tip, imenovan **medpomnilnik** (**buffer**), ki predstavlja polje z določeno velikostjo. Obstajajo še drugi tipi, ki jih lahko podamo jedru, ampak za razumevanje standarda niso pomembni in zato tukaj niso omenjeni.

Da lahko jedra s svojimi parametri izvajamo, moramo na računski napravi ustvariti **ukazno vrsto** (**command queue**). Vsaka ukazna vrsta pripada natanko *eni računski napravi* in preko nje določimo, kaj in s kakšnimi para-

metri želimo poganjati na napravi. Za celovitost omenimo še, da lahko med posameznimi ukazi v isti ali različnih vrstah določimo odvisnosti, s čimer omejimo vrstni red izvajanja.



Slika 2.3: Primer razdelitve dvodimenzionalnega prostora velikosti $15 * 15$ v okolju OpenCL

Ko dodamo jedro v ukazno vrsto, moramo določiti, v kakšnem prostoru se bo jedro vzporedno izvajalo. Tu določimo velikost celoštevilskega **n-dimenzionalnega** (**ND-range**) prostora, ki predstavlja razbitje problema na vzporedno izvedljive dele. Ta prostor imenujemo **globalni prostor** in ga bomo označili z G. Za primer vzemimo prostor $G(15, 15)$, velikosti $15 * 15$ (slika 2.3). Vsaka točka v tem prostoru predstavlja eno **izvajalno točko** (**work-item**), ki se izvaja na natanko *enem izvajalnem elementu* in ima svoj *privatni pomnilnik*. Tako se točka $(0, 0)$ izvaja na enem elementu, točka $(0, 1)$ na drugem in tako naprej do točke $(14, 14)$.

Izvajalne točke se združujejo v **delovne skupine** (**work-groups**), ki raz-

delijo delo po računskih enotah. Delovne skupine globalni prostor razbijejo na manjše **lokalne prostore**, ki jih bomo označili z L. Torej če v našem primeru za lokalni prostor vzamemo $L(5, 5)$, potem imamo devet enako velikih delovnih skupin velikosti $5 * 5$. Vsaka delovna skupina se izvaja na natanko *eni računski enoti* in ima svoj *lokalni pomnilnik*. Pomembno je vedeti, da standard OpenCL definira vzporedno izvajanje jeder le na ravni izvajalnih točk v isti delovni skupini. Sinhronizacija izvajanja je tako možna le med jedri iste delovne skupine. Delovne skupine se kljub temu lahko izvajajo vzporedno, ni pa nujno in je odvisno od implementacije.

Nenazadnje si vse izvajalne točke iz vseh izvajalnih skupin delijo isti *globalni pomnilnik*.

Pri predstavitvi jezika OpenCL C v podpoglavju 2.3.2 bo bolj nazorno prikazano, kako se tukaj predstavljeni prostor uporabi za vzporedno izvajanje programa.

Programski model

OpenCL podpira **podatkovno** in **opravilno** gnano vzporedno izvajanje ali mešanico obojega.

Podatkovno vzporedno izvajanje nam omogoča izvajanje *ene naloge* na *več izvajalnih elementih*. Enostaven primer je seštevanje istoležečih elementov v dveh tabelah. Ker je seštevanje dveh elementov neodvisno od drugih seštevanj, se lahko posamezna seštevanja ali skupine seštevanj porazdelijo po izvajalnih elementih. To nam omogoča razbitje problema na izvajalne točke v nekem prostoru.

Na drugi strani nam opravilno vzporedno izvajanje omogoča izvajanje *različnih nalog* na *svojih računskih enotah*. Ukazne vrste se izvajajo neodvisno ena od druge in vsaka na svoji računski napravi. Če implementacija podpira dodatno funkcionalnost, se lahko posamezni ukazi znotraj vrste izvajajo tudi v poljubnem vrstnem redu.

Očitno je, da lahko z uporabo standarda OpenCL ta dva modela poljubno mešamo.

2.3.2 Programski jezik OpenCL C

Programski jezik OpenCL C je okleščena različica programskega jezika C verzije ISO C99 z razširitvami za pisanje vzporednih programov [8]. Zaradi različnih tipov procesorjev so nekateri deli morali biti izpuščeni iz standarda, medtem ko so zaradi večjega in lažjega izkoristka paralelnosti bili dodani novi deli.

Poglejmo najprej izvzete dele. Ker vsi procesorji ne podpirajo istih funkcionalnosti kot procesorji CPE, za katere je tudi bil jezik C napisan, je moral biti jezik OpenCL C okleščen. Glavni izvzeti deli so:

- rekurzivne funkcije,
- kazalci na funkcije,
- bitna polja.

Poleg tega je omejena uporaba standardnih knjižnic. Večino knjižnic, ki spadajo vanjo, ni enostavno ali sploh mogoče implementirati na vseh tipih procesorjev, zato so morale biti izvzete.

Med glavno dodano funkcionalnost spadajo:

- vektorski tipi (**`char2`**, **`int8`**, **`float4`** ...),
- različne vrste pomnilnikov (**`global`**, **`constant`** in **`local`**),
- množica vgrajenih funkcij za pogosto uporabljeno funkcionalnost v aplikacijah OpenCL (matematične funkcije, funkcije za upravljanje s slikami, vektorske operacije ...),
- atomske funkcije.

Obvezna je podpora standarda IEEE 754 za računanje s plavajočo vejico, vendar z izvzetimi določenimi deli, ki so zahtevni za strojno implementacijo.

Poleg tega imajo vse vgrajene matematične funkcije določeno dovoljeno relativno napako izračunanih vrednosti.

Poglejmo si še primer pisanja programa OpenCL C glede na navaden program C [10]. Vzemimo dve tabeli dolžine n, na katerih želimo sešteti istoležeče elemente in tako dobiti novo tabelo. Enostavna implementacija takšne funkcije v jeziku C izgleda tako:

Izvorna koda 2.1: Zaporedno seštevanje v jeziku C

```
void
regular_add (int n,
             const float *a,
             const float *b,
             float *r)
{
    int i;
    for (i=0; i<n; i++)
        r[i] = a[i] + b[i];
}
```

Razvidno je, da se seštevanje izvaja zaporedno za vsak indeks posebej.

Ker so posamezna seštevanja med seboj neodvisna, lahko v okolju OpenCL to izkoristimo pri vzporednem izvajanju izvajalnih točk. Za globalni računski prostor vzamemo kar enodimenzionalni prostor velikosti n, ker potrebujemo natanko n seštevanj. Našo funkcijo oziroma jedro zato napišemo samo za seštevanje dveh istoležečih elementov:

Izvorna koda 2.2: Vzporedno seštevanje v jeziku OpenCL C

```
kernel void
parallel_add (global const float *a,
              global const float *b
              global float *r)
{
    int id = get_global_id(0);
    r[id] = a[id] + b[id];
}
```

Ta funkcija se bo izvedla natanko n-krat in vsaka njena instanca bo s klicem funkcije `get_global_id(int dimension)` dobila ustrezen indeks z globalnega prostora. To bodo indeksi od 0 do $n - 1$. Tako bo n instanc te funkcije izvedlo n seštevanj različnih istoležečih elementov v tabelah. Ko se vse funkcije uspešno izvedejo, dobimo isti rezultat kot pri implementaciji v jeziku C, le da smo tukaj izkoristili vzporedno izvajanje posameznih seštevanj in tako pospešili izvajanje.

Poglavje 3

Načrt

Ideja za implementacijo orodja za testiranje implementacij standarda OpenCL je nastala v skupnosti X.Org, natančneje v skupnosti Mesa, ki razvija gonilnike FOSS za grafične kartice. Tu je večina gonilnikov napisana z uporabo ogrodja Gallium3D.

Ogrodje Gallium3D skuša deliti čim več skupne kode med gonilniki za različne grafične kartice, z namenom da omogoči hitrejši razvoj in enostavnejše vzdrževanje. Grobo gledano je sestavljen iz treh ločenih slojev. Najvišji sloj predstavljajo *grafični vmesniki* (OpenGL, OpenCL, OpenVG ...), na sredini stojijo *strojni gonilniki* za grafične kartice (Radeon, Nouveau ...) in na dnu ležijo *sistemski vmesniki* za različne operacijske sisteme (Linux, FreeBSD, Windows ...). Med sloji poteka komunikacija preko natančno definiranih vmesnikov in tako lahko na primer s pisanjem samo strojnega gonilnika za okolje Gallium3D, dobimo grafični gonilnik, ki podpira standard OpenGL brez potrebnega dodatnega dela, ker je ta grafični vmesnik že implementiran.

Grafični vmesnik za standard OpenCL v okolju Gallium3D se imenuje clover. Vmesnik je v času pisanja diplomske še vedno v razvoju. Cilj našega testnega programa je, da s testiranjem ujame čim več napak v razvoju, olajša implementacijo nove funkcionalnosti s testno vodenim razvojem in skrbi za odkritje regresijskih napak. Testni program ne sme biti omejen na gonilnike

Mesa in mora podpirati testiranje vseh implementacij standarda OpenCL. Vmesnik clover je le glavno gonilo za pisanje takšnega programa.

3.1 Izbrane tehnologije

Naše testno orodje mora testirati skladnost s standardom OpenCL. Takšnega orodja z javnim razvojem in večjo razvojno skupnostjo žal ni. Ena izmed izbir bi lahko bil uraden test skladnosti v lasti organizacije Khronos[1], vendar razvoj in izvorna koda nista javna. Testno orodje je bilo zato potrebno napisati na novo oziroma razširiti neko obstoječe orodje.

Piglit

V skupnosti Mesa se uporablja orodje za testiranje skladnosti s standardom OpenGL, imenovano Piglit. Vsi razvijalci gonilnikov Mesa že znajo uporabljati in pisati teste za orodje Piglit, zato je bila logična izbira, da se to orodje razširi s podporo za testiranje skladnosti s standardom OpenCL.

Orodje Piglit testira skladnost grafičnih gonilnikov s standardom OpenGL. Implementirano je večinoma v programskih jezikih Python in C. Testi in testna ogrodja za orodje Piglit so lahko napisani v *poljubnem jeziku*, čeprav večina uporablja jezik C.

Večina testov je prevedenih v svoje izvršljive datoteke, tako da se lahko vsak posamezen test izvaja neodvisno od drugih. Tu nam orodje Piglit s pomočjo jezika Python ponuja ustvarjanje skupin testov, ki se jih izvaja avtomatsko in po želji tudi vzporedno. V skupine lahko dodajamo posamezne izvršljive datoteke skupaj z argumenti za izvajanje.

Orodje Piglit se uporablja tako, da s priloženo skripto poženemo neko vnaprej definirano skupino testov. Ko se vsi testi v skupini izvedejo, dobimo datoteko z rezultati vseh testov. Ti rezultati so lahko pretvorjeni v dokument HyperText Markup Language (HTML), ki nam omogoča grafični

[1] `http://www.khronos.org/opencl/adopters/`

pregled vseh uspešno in neuspešno opravljenih testov.

Z uporabo orodja Piglit smo pridobili na *hitrosti* razvoja testnega orodja, saj smo uporabili veliko že implementirane funkcionalnosti. Med njo spadajo *vzporedno izvajanje testov*, *grupiranje testov* in *grafični prikaz* rezultatov. Dodatna prednost je, da je z razširitvijo orodja Piglit razvijalcem potrebno poznati le *eno testno orodje* za testiranje skladnosti s standardoma OpenGL in OpenCL.

Programski jezik C

Izbira večine tehnologij za razvoj testnega orodja je bila pogojena z enostavnostjo uporabe in možnostjo nadgradnje s strani razvijalcev grafičnih gonilnikov Mesa. Čeprav je testno orodje Piglit pisano tako, da podpira izvajanje testov, napisanih v različnih programskih jezikih, je bilo naše testno orodje implementirano v programskem jeziku C. Jezik C je bil izbran, ker je vmesnik API standarda OpenCL definiran za jezik C, ker uporabniki orodja Piglit programirajo večinoma v njem in ker v orodje Piglit nismo želeli dodati odvisnosti za nov jezik.

Programski jezik C je po nekaterih merilih najbolj uporabljen programski jezik na svetu [2]. Uporablja se za programiranje večine programov, ki potrebujejo natančen nadzor nad izvajanjem programa na strojni opremi in ne potrebujejo nobenega posebnega izvajalnega okolja oziroma implementirajo svojega. V to skupino programov spadajo operacijski sistemi, gonilniki in razni programi s potrebo po hitrem izvajanju.

Z uporabo jezika C smo omogočili *direktne klice* vmesnika API implementacij standarda OpenCL in ohranili *enotnost* razvojnega jezika za orodje Piglit.

Git

Orodje Piglit se razvija javno s pomočjo orodja za nadzor izvorne kode (SCM), imenovanim Git. Ker je naš projekt moral biti integriran v orodje Piglit in ker je orodje Git eden izmed najbolj uporabljenih orodij za nadzor izvorne kode [1], smo za razvoj in nadzor razvoja projekta izbrali orodje Git.

Orodje Git se uporablja za nadzor izvorne kode veliko projektov, največ uporabe ima v skupnosti FOSS in večina projektov v skupnosti X.Org se razvija ravno z uporabo tega orodja[2]. Zaradi enostavnega ustvarjanja vej razvoja, lokalno prisotne celotne zgodovine in porazdeljenega nadzora, je primeren za uporabo na projektih z veliko razvijalci.

Z uporabo orodja Git smo sledili sprejetim standardom za razvoj v skupnosti FOSS in *združili* izvorno kodo implementiranega orodja v glavni repozitorij orodja Piglit. Razbitje projekta na logične dele in pripadajoča zgodovina sprememb so omogočili enostaven *pregled kode* pred združevanjem.

3.2 Zasnova programa

Glavni cilj našega programa je, da razvijalcem gonilnikov ponudi orodje za poganjanje in pisanje testov za standard OpenCL. Ker so naši ciljni uporabniki razvijalci, katerim ni glavna skrb pisanje testov, smo si za testno orodje zadali naslednje cilje:

- enostavno izvajanje testov in pregled rezultatov,
- enostavno in hitro pisanje testov,
- razširljivost orodja.

Za *enostavno izvajanje in pregled rezultatov* smo uporabili kar orodje Piglit. Teste je zato potrebno prevesti v samostojne izvršljive datoteke ali

[2]Imenik repozitorijev: `http://cgit.freedesktop.org/`

datoteke, katerih izvajanje nadzirajo vhodni argumenti. Tako lahko orodje Piglit uporabi posamezne teste za skupinsko izvajanje in grafično prikaže dobljene rezultate. S tem smo pridobili tudi na tem, da za izvajanje posameznega testa ne potrebujemo nekega zunanjega orodja, ampak lahko samo poženemo samostojno izvršljivo datoteko.

Da je *pisanje testov enostavno in hitro*, smo poskrbeli tako, da je za ustvarjanje novih testov potrebno čim manj pisanja nove kode. Ena izmed uporabljenih rešitev je, da pogosto uporabljene skupke kode ponudimo kot pomožne funkcije in strukture. Primeri so klici funkcij API z zmanjšanim številom argumentov, grupirana zaporedja klicev funkcij in grupirane vrednosti argumentov funkcij API. Druga rešitev je, da za teste, ki se izvajajo v podobnem okolju, uporabimo skupno izvorno kodo, katera nam to okolje avtomatsko pripravi. Primer je testiranje jeder, kjer za vsako testno jedro potrebujemo isto okolje, ki nam požene jedro na izbrani računski napravi. Z uporabo obeh rešitev je za test potrebno napisati le del izvorne kode, kjer se test dejansko izvede. Tako smo naše teste razdelili na:

- minimalni testni del,
- izvajalno okolje (skupno vsem testom),
- izvajalno okolje (skupno le neki skupini testov),
- pomožne funkcije in strukture.

Vsakemu piscu testov je tako v veliki večini primerov potrebno napisati le minimalni testni del.

Razširljivost programa smo omogočili ravno z različnimi izvajalnimi okolji oziroma, kot bomo to kasneje definirali, tipi testov. Okolja lahko nadziramo s konfiguracijskimi možnostmi, s katerimi lahko okolje delno prilagodimo vsakemu testu in tako z njim pokrijemo večjo množico testov. Če nam to ni dovolj, potem lahko enostavno napišem nov tip testov, ki nam ustvari neko

novo okolje. Z različnimi tipi testov se naše orodje lahko poljubno širi in poenostavi pisanje zapletenih testov.

Ker želimo, da je naše orodje kar se da natančno pri preverjanju skladnosti s standardom OpenCL, podpiramo izbiro testiranja na vseh ali samo izbranih:

- verzijah standarda OpenCL,
- platformah in računskih napravah.

Testiranje mora biti ločeno po posameznih verzijah standarda OpenCL, ker navadno gonilniki postopoma po verzijah pridobivajo novo funkcionalnost in potrebujejo testiranje le določene verzije. Testirano verzijo standarda lahko določimo pri poganjanju testa, medtem ko je piscem testov omogočeno, da s pisanjem enega testa testirajo različne verzije standarda. Izbira platforme in računskih naprav, na katerih se testi izvajajo, nam dovoli omejevanje izvajanja testov na samo določene naprave. To lahko izvedemo na ravni specifičnega testa ali skupine testov.

Poglavje 4

Orodje

Testno orodje, napisano v sklopu diplomskega dela, je razširitev obstoječega orodja Piglit s podporo testiranja implementacij standarda OpenCL. Glavni sestavni del podpore novega standarda je ogrodje za pisanje testov. Implementacijsko gledano ogrodje uporablja zelo majhen del orodja Piglit in je skoraj v celoti neodvisno napisano. Da lahko orodje Piglit grafično prikaže rezultate testov in ločeno požene vsak test, smo morali ogrodje temu ustrezno prirediti.

V tem poglavju bomo razložili, kako je implementirano naše orodje, kateri so njegovi sestavni deli in kako so povezani skupaj. Tu bomo dobili osnoven pregled nad delovanjem ogrodja in kako ga lahko uporabimo za pisanje testov.

Povzeli bomo tudi funkcionalnost orodja Piglit. Pregledali bomo, katere dele orodja smo uporabili, kako smo mu priredili ogrodje ter kako ga uporabljamo za poganjanje testov in pregled rezultatov.

4.1 Ogrodje za pisanje testov

V poglavju 3 smo si med cilji našega ogrodja zadali, da je ogrodje *enostavno razširljivo* ter da omogoča *enostavno in hitro pisanje testov.* Testi v tem ogrodju naj bi potrebovali *minimalno kodiranja* za pisanje testa in se izvajali

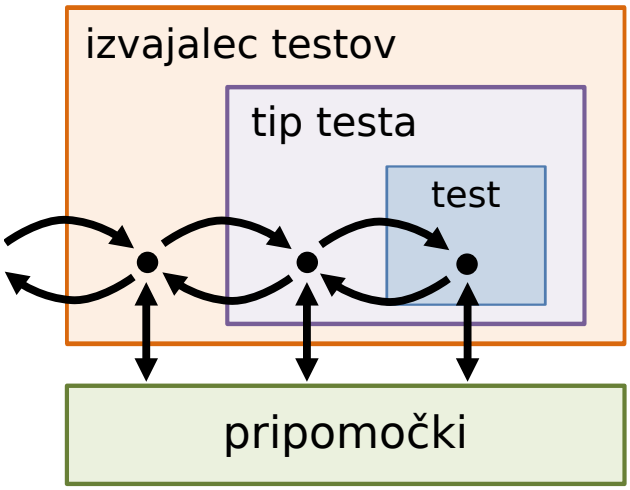
v natančno določenem *izvajalnem okolju* skupaj z uporabo *pomožnih funkcij in struktur*. Poleg tega naj bi bila za vsak test možna izbira testirane *verzije standarda OpenCL* ter *platform in računskih naprav*.

Ogrodje smo za doseg teh ciljev razdelili na tri logične dele:

Izvajalec testov: Skrbi za izvajanje testa za določeno verzijo standarda OpenCL ter izvajanje na vseh oziroma samo izbranih platformah in računskih napravah. S poganjanjem testa pod različnimi pogoji omogoča krajše pisanje testov in določa del izvajalnega okolja, skupnega vsem testom.

Tipi testov: Združujejo in izvajajo pogosto uporabljeno skupno kodo podobnih testov in jo ponudijo kot izvajalno okolje tem testom. S tem omogočimo krajše pisanje testov ter enostavno in natančno določimo izvajalno okolje vsaki skupini testov.

Pripomočki: Ponujajo funkcije in strukture s katerimi lažje in hitrejše pišemo vse ostale dele ogrodja kot tudi same teste. Največkrat so to skupki funkcij, združeni v enostavnejše funkcije.



Slika 4.1: Potek izvajanja testa v ogrodju za pisanje testov

Na sliki 4.1 lahko vidimo, kako so med seboj odvisni deli ogrodja in kako poteka izvajanje posameznega testa. Poglejmo si bolj podrobno, kako deluje, kako je implementiran in kako se uporablja vsak izmed teh delov.

4.1.1 Izvajalec testov

Skupni del vsem testom našega ogrodja je skupek kode, imenovan **izvajalec testov** (izvorni datoteki `tests/util/piglit-framework-cl.c` in `tests/util/piglit-framework-cl.h`). Z njim uvedemo programske strukture za pisanje testnih tipov in preko njih tudi samih testov. Skrbi, da se testi izvedejo na zadnji podprti verziji standarda OpenCL ter na vseh ali samo izbranih platformah in računskih napravah.

Izvajalec testov implementira glavno vhodno funkcijo ogrodja, imenovano `piglit_cl_framework_run(...)`, ki skrbi za ustrezno poganjanje testov. Funkcija glede na konfiguracijo testa, ki je razložena v poglavju 5, požene test samo enkrat, za vse različne platforme ali za vse različne platforme in naprave.

V drugem in tretjem primeru izvajalec testov najprej poišče vse implementacije standarda OpenCL oziroma platforme prisotne na sistemu in za vsako preveri najvišjo podprto verzijo standarda. Nato vsaki računski napravi na vsaki platformi (ali samo na vsaki platformi) izvede test za ustrezno verzijo standarda. Izbiranje točno določenih verzij, platform in računskih naprav, na katerih se izvaja test, je mogoče določiti ob poganjanju testa (atributi okolja, definirani v razdelku 4.2.3) ali ob pisanju testa (atributi konfiguracije, definirani v razdelku 5.1).

Vsakič, ko se test izvede, je njegov rezultat združen z rezultati prejšnjih izvedb istega testa in na koncu je na standardni izhod izpisan skupni rezultat vseh testov. Natančen format izpisa rezultata je definiran v razdelku 4.2.1.

Določene funkcije in strukture, definirane v izvajalcu testov, morajo uporabiti vsi testi, napisani za naše ogrodje. Če smo bolj natančni, jih uporabijo

vsi tipi testov in posledično preko njih tudi sami testi. Tako z uporabo spodaj predelanih struktur omogočimo, da se vsak test prevede v svojo izvršljivo datoteko. To potrebujemo za ločitev testov za potrebe orodja Piglit in za samostojno poganjanje posameznih testov.

Vsak tip testov nam postavi neko izvajalno okolje, zato mora *definirati konfiguracijsko strukturo*, da lahko testi nadzirajo postavljanje sebi primernega okolja. Izvajalec testov za ta namen izvozi dva makro izraza `PIGLIT_CL_DEFINE_TEST_CONFIG_*` (izvorna koda 4.1).

Izvorna koda 4.1: Makro izraza, ki se uporabita za definicijo konfiguracijske strukture

```
#define PIGLIT_CL_DEFINE_TEST_CONFIG_BEGIN(test_config_struct_t)\
    test_config_struct_t {\
        PIGLIT_CL_TEST_CONFIG_HEADER
//#enddefine PIGLIT_CL_DEFINE_TEST_CONFIG_BEGIN

        /* Here goes the test configuration definition */

#define PIGLIT_CL_DEFINE_TEST_CONFIG_END\
    };
//#enddefine PIGLIT_CL_DEFINE_TEST_CONFIG_END
```

Ker izvajalec testov že določi del te konfiguracije z izbiranjem verzije standarda, platform in računskih naprav, mora v konfiguracijo vriniti svoje konfiguracijske možnosti. To stori z uporabo makro izraza `PIGLIT_CL_TEST_CONFIG_HEADER` znotraj definicije konfiguracijske strukture (`test_config_struct_t`). Natančne konfiguracijske možnosti vseh testov in posameznih tipov testov so opisane v poglavju 5.

Druga pomembna struktura za pisanje testov, katero definira izvajalec testov, sta makro izraza `PIGLIT_CL_TEST_CONFIG_*` (izvorna koda 4.2). Z njima mora tip testov določiti nova makro izraza (navadno `PIGLIT_CL_SOMETYPE_TEST_CONFIG_*`), katera potem uporabi sam test za *nastavitev konfiguracijske strukture*. Tu mora tip testov kot argumente podati tip konfiguracijske struk-

ture (`test_config_struct_t`), funkcijo (`get_empty_test_config_f`), ki vrne vrednosti prazne konfiguracijske strukture, in testno funkcijo (`test_run_f`), ki izvede test.

Izvorna koda 4.2: Makro izraza, ki se uporabita za nastavitev konfiguracijske strukture in izvedbo testa

```
#define PIGLIT_CL_TEST_CONFIG_BEGIN(test_config_struct_t,\
                                    get_empty_test_config_f,\
                                    test_run_f)\
    test_config_struct_t config;\
    \
    void* piglit_cl_get_test_config(\
        const int argc,\
        const char** argv,\
        const struct piglit_cl_test_config_header* ↩
    config_header_ptr)\
    {\
        piglit_cl_get_empty_test_config_t* _get_empty_config = ↩
    get_empty_test_config_f; /*for compile time check*/\

        memcpy(&config, _get_empty_config(), sizeof(↩
    test_config_struct_t));\
        memcpy(&config, config_header_ptr, sizeof(struct ↩
    piglit_cl_test_config_header));\
        \
        config._test_run = test_run_f;\
        config._filename = __FILE__;
//#enddefine PIGLIT_CL_TEST_CONFIG_BEGIN

        /* Here goes the configuration of the test type */

#define PIGLIT_CL_TEST_CONFIG_END\
        return &config;\
    }\
    \
    int main(int argc, char** argv)\
    {\
        return piglit_cl_framework_run(argc, argv);\
    }
//#enddefine PIGLIT_CL_TEST_CONFIG_END
```

Oba makro izraza skupaj poskrbita za definicijo funkcije `piglit_cl_get_test_config(...)`, s katero izvajalec testov dobi od samega testa ustrezno nastavljeno konfiguracijsko strukturo. Test med tema makro izrazoma nastavi to strukturo z uporabo spremenljivke `config`. Na podlagi te konfiguracije izvajalec testa in testni tip postavita ustrezno okolje za test. Izvajalec testa pri tem požene test na ustrezni verziji standarda ter na izbranih platformah in računskih napravah.

Makro izraza poleg tega definirata funkcijo `main()`. Tako je vsak test lahko preveden v svojo izvršljivo datoteko, ker ima implementirano to funkcijo. Posledično je zaradi tega izvajalec testov preveden v deljeno knjižnico, ki jo izvršljive datoteke testov uporabijo ob samem izvajanju. Ker je vsak test neodvisna izvršljiva datoteka, ga lahko poganjamo posamično ali pa dodamo v skupino testov v orodju Piglit.

Pri zadnjem makro izrazu lahko vidimo, da funkcija `main()` samo kliče funkcijo izvajalca testov `piglit_cl_framework_run(...)`. Tako je nadziranje izvajanje testa v popolnosti prepuščeno izvajalcu testov in se testom ni potrebno ubadati s tem.

4.1.2 Tipi testov

Drugi pomembni del ogrodja so **tipi testov** (izvorne datoteke `tests/util/piglit-framework-cl-*.c` in `tests/util/piglit-framework-cl-*.h`). Odvisni so od izvajalca testov, ker jih je mogoče implementirati le z njim. Glavna naloga tipov testov je, da skupinam testov ponudijo konfiguracijske možnosti, s pomočjo katerih za teste ustvarijo skupini prirejeno specifično izvajalno okolje. S tem poenostavimo pisanje večje skupine podobnih testov, saj izluščimo skupno kodo in jo dodamo v tip testov. Poleg tega lahko s tipi testov poljubno nadgradimo funkcionalnost izvajalca testov.

V nadaljevanju predstavljata izraza `testtypename` in `TESTTYPENAME` ime tipa testov. Vsak tip testov ima v svoji implementaciji ta dva izraza zamenjana z imenom, ki ustrezno predstavlja implementirani tip.

Tipi testov navadno definirajo funkcijo `piglit_cl_testtypename_test_run(...)`, ki jo kliče izvajalec testov za izvedbo testa. S konfiguracijsko strukturo, dobljeno preko argumenta, tip testa ustvari želeno okolje za test. Ko ustvari to okolje, kliče testno funkcijo ter po njeni izvedbi sprosti ustvarjeno okolje in vrne rezultat testa.

Testna funkcija, s katero mora biti implementiran test, se navadno imenuje `piglit_cl_test`. Ta testna funkcija prejme *vhodne argumente izvršljive datoteke*, *konfiguracijsko strukturo* in *strukturo okolja*. Vsak tip testov mora definirati svoj tip testne funkcije, ker testi dobijo pri vsakem tipu testov drugačno okolje.

Kot smo videli pri opisu izvajalca testov, moramo uporabiti štiri makro izraze, da implementiramo tip testov. Najprej poglejmo primer uporabe makro izrazov `PIGLIT_CL_DEFINE_TEST_CONFIG_*` (izvorna koda 4.3), s katerima *definiramo konfiguracijske možnosti* za naš tip testa.

Izvorna koda 4.3: Definiranje konfiguracijske strukture tipa testov

```
PIGLIT_CL_DEFINE_TEST_CONFIG_BEGIN(struct ↩
    piglit_cl_testtypename_test_config)

    bool create_context; /* Create OpenCL context */

PIGLIT_CL_DEFINE_TEST_CONFIG_END
```

Prvi makro izraz dobi za argument tip strukture (`struct piglit_cl_testtypename_test_config`), ki bo hranila konfiguracijske možnosti. Nato pod njim naštejemo konfiguracijske možnosti, ki bodo prisotne poleg že določenih možnosti izvajalca testov. Na koncu le še zaključimo strukturo z drugim makro izrazom. V našem primeru bo vsak test, ustvarjen s pomočjo tega tipa testov, lahko uporabil konfiguracijsko možnost `create_context`, ki določa, če test potrebuje kontekst OpenCL.

Naslednje, kar uporabimo za definiranje tipa testov, sta makro izraza

`PIGLIT_CL_TEST_CONFIG_*` (izvorna koda 4.4). Z njuno uporabo definiramo dva nova makro izraza (v našem primeru sta to `PIGLIT_CL_TESTTYPENAME_TEST_CONFIG_*`), ki ju morajo uporabiti testi, napisani za ta tip testov. Z uporabo teh makro izrazov testi določijo, katerega tipa so in poleg tega med njima naštejejo konfiguracijske nastavitve. Natančno kako to poteka, bo razdelano v poglavju 5.

Izvorna koda 4.4: Definicija strukture okolja, tipa testne funkcije in makro izrazov, ki se uporabita pri pisanju testa

```
struct piglit_cl_testtypename_test_env {
    cl_context context; /* Opencl context */
};

typedef enum piglit_result
    piglit_cl_testtypename_test_t(const int argc,
        const char** argv,
        piglit_cl_testtypename_test_config_t* config,
        piglit_cl_testtypename_test_env_t* env);

#define PIGLIT_CL_TESTTYPENAME_TEST_CONFIG_BEGIN\
    piglit_cl_testtypename_test_t piglit_cl_test;\
    \
    PIGLIT_CL_TEST_CONFIG_BEGIN(struct ↩
   piglit_cl_testtypename_test_config,\
        piglit_cl_get_empty_testtypename_test_config,\
        piglit_cl_testtypename_test_run)
//#enddefine PIGLIT_CL_TESTTYPENAME_TEST_CONFIG_BEGIN

    /* Here goes the configuration of the test */

#define PIGLIT_CL_TESTTYPENAME_TEST_CONFIG_END\
    config._testtypename_test = piglit_cl_test;\
    \
    PIGLIT_CL_TEST_CONFIG_END
//#enddefine PIGLIT_CL_TESTTYPENAME_TEST_CONFIG_END
```

Preden definiramo nova makro izraza, moramo definirati *strukturo okolja* (`struct piglit_cl_testtypename_test_env`), ki vsebuje spremenljivke okolja za uporabo v testu. V našem primeru imamo tu podan le kontekst

OpenCL (`cl_context context`). Nato definiramo tip *testne funkcije* (`piglit_cl_testtypename_test_t`), ki jo kliče tip testov, ko želi izvesti test. Funkcijo implementira sam test in kot lahko vidimo prejme za vhodne argumente naslednje vrednosti: vhodne argumente izvršljive datoteke, konfiguracijsko strukturo in strukturo okolja, katero zapolni tip testov.

Sedaj ko smo definirali potrebne strukture, lahko uporabimo prej omenjena makro izraza za definicijo novih, ki ju bodo morali uporabiti testi napisani za ta tip testov. Pri definiciji prvega makro izraza definiramo ime testne funkcije (`piglit_cl_test`), ki jo uporabi test, za pisanje dejanskega testa. Nato uporabimo `PIGLIT_CL_TEST_CONFIG_BEGIN`, da določimo tip konfiguracijske strukture (`struct piglit_cl_testtypename_test_config`), funkcijo (`piglit_cl_get_empty_testtypename_test_config`), ki vrne prazno konfiguracijsko strukturo, in funkcijo (`piglit_cl_testtypename_test_run`), ki jo požene izvajalec testov za izvedbo testa. Obe funkciji morata biti v tipu testov implementirani in zadnja mora vsebovati kodo za postavljanje primernega testnega okolja in klic testne funkcije testa.

Kot smo že povedali, funkcijo, ki jo požene izvajalec testa za izvedbo testa, implementira testni tip in ne test. Testni tip jo uporabi za vzpostavitev izvajalnega okolja in podatkovne strukture okolja. Šele ko to stori, kliče pravo testno funkcijo, ki jo implementira sam test (`piglit_cl_test`). Testna funkcija testa je vnaprej nastavljena v konfiguracijski strukturi pri definiciji drugega makro izraza, da jo lahko tip testa kasneje kliče direktno iz svoje implementacije.

4.1.3 Pripomočki

Tretji del ogrodja so **pripomočki** (izvorne datoteke `tests/util/piglit-util-cl*.c` in `tests/util/piglit-util-cl*.h`). Sem spadajo razne pomožne funkcije in strukture. Z njimi želimo poenostaviti in skrajšati pisanje tako ogrodja kot tudi samih testov.

Ker je implementiranih pomožnih funkcij in struktur veliko, smo jih za

opis razdelili v skupine. Tako za ogrodje in teste uporabljamo naslednje programske konstrukte:

Prikladnostne funkcije: Funkcije vmesnika API standarda OpenCL z zmanjšanim številom argumentov. Veliko agumentov v funkcijah, definirah v standardu, se redko uporablja. Prikladnostne funkcije sprejmejo zmanjšano število argumentov in za manjkajoče vstavijo standardne vrednosti ter kličejo ustrezno funkcijo vmesnika API.

Združevalne funkcije: Skupki klicev funkcij vmesnika API standarda OpenCL združeni v eno funkcijo. Veliko funkcij standarda se uporablja v določenem zaporedju ali pa je potrebno veliko programske kode za ustvarjanje določenega argumenta funkcije. Združevalne funkcije združijo takšne skupke kode v posamezne funkcije s čim manj vhodnimi argumenti.

Informativne funkcije: Vračajo informacije o raznih delih platforme OpenCL. Uporabljajo se za preverjanje stanja in zmožnosti tako naprav kot objektov, ki jih definira standard.

Preverjevalne funkcije: Preverjajo stanje izvajalnega okolja OpenCL. Uporabljajo se za preverjanje verzije in razširitev ter povpraševanje po napakah pri izvajanju implementacije.

Primerjalne funkcije: Primerjajo vrednosti spremenljivk z določenim dovoljenim odstopanjem. Uporabljajo se za preverjanje rezultatov pri računanju na računskih napravah.

Pomožne strukture: Združujejo več spremenljivk stanja izvajalnega okolja OpenCL. Z njimi omogočimo lažje in krajše podajanje argumentov raznim funkcijam.

Konstante in simboli: Konstante in simboli, določeni v standardu, so združeni v skupine za enostavnejše iteriranje po njih. Združeni so po namenih uporabe in po verzijah standarda.

Vsi pomožni konstrukti so namenjeni za uporabo v jeziku C. Trenutno nimamo nobenih podobnih konstruktov za uporabo pri pisanju testov v jeziku OpenCL C.

4.2 Orodje Piglit

Testno orodje Piglit je nastalo kot skupek skript za poganjanje testov in testov za testiranje skladnosti s standardom OpenGL. Del tega orodja, ki ga želimo izkoristiti, omogoča grafični prikaz in poganjanje skupine testov oziroma profilov. Naše ogrodje smo zato že na začetku prilagodili in napisali za orodje Piglit.

Po začetni implementaciji ogrodja je sledila njegova vključitev v orodje Piglit, kjer se ogrodje še vedno razvija in pridobiva nove teste. Tako imamo eno orodje za testiranje dveh različnih standardov. Omenimo še, da je s tem postalo možno pisanje testov, ki testirajo interoperabilnost med poljubnima implementacijama obeh standardov.

4.2.1 Testi

Orodje Piglit podpira različne načine poganjanja testov. Pri našem ogrodju smo se odločili za uporabo *navadnih izvajalnih testov*, kjer Piglit požene izvršljivo datoteko, ki izvede test in na *standardni izhod* izpiše rezultat testa. Da je vsak test svoja izvršljiva datoteka, smo v ogrodju poskrbeli z izvajalcem testov.

Piglit pridobi uspešnost izvedbe testa, tako da bere ves standardni izhod testnega programa. Od tu izlušči vse vrstice, ki vsebujejo ključno besedo `PIGLIT:` in preostanke teh vrstic razčleni po formatu JSON. Rezultat celotnega testa tako pridobi iz vrstice, ki vsebuje ključ `result`:

```
PIGLIT: { 'result':  '(pass|fail|skip|warn)' }
```

Od tu ugotovi, ali je bil test uspešno prestan (`pass`), ali je spodletel (`fail`), ali je prišlo do nekritične napake (`warn`), ali sploh ni bil izveden (`skip`).

Piglit podpira tudi delne rezultate testov oziroma **podteste**. Rezultati podtestov so predstavljeni v podobnem formatu, le da se tu uporablja ključna beseda `subtest`. Tu je rezultat podtesta naveden pod poljubnim ključem, ki definira ime podtesta:

```
PIGLIT:subtest { 'ime_podtesta':  '(pass|fail|skip|warn)' }
```

Takšen izpis programa je edini potreben pogoj za navadne izvajalne teste orodja Piglit. Testni programi so zaradi tega lahko implementirani v poljubnem jeziku. Naše ogrodje je napisano v jeziku C kot tudi ogrodje za teste OpenGL, zato smo iz ogrodja OpenGL izluščili majhen del kode, ki je skupen obema. V njem so funkcije za izpis rezultatov testa in nekaj ostalih pomožnih funkcij. To je tudi edini del našega ogrodja, ki je združen z ostalimi deli orodja Piglit.

4.2.2 Testni profili

Ker orodje Piglit ni namenjeno za poganjanje le enega testa, saj to lahko enostavno storimo sami, je potrebno teste združiti v skupine oziroma profile. **Profili** predstavljajo najmanjšo enoto, s katero lahko izvajamo teste v orodju Piglit.

Za definiranje profila ustvarimo novo datoteko z izvorno kodo Python (primer: izvorna koda 4.5). V njej iz orodja Piglit najprej uvozimo potrebne objekte: `TestProfile`, `Group` in `PlainExecTest`. Prva dva objekta sta nujna za definiranje novih profilov, medtem ko je `PlainExecTest` potreben za definiranje testov na podlagi izvršljivih datotek, s Piglitu prirejenim izpisom. Ker z našim ogrodjem ustvarjamo samo takšne teste, je dovolj, da poleg nujno potrebnih elementov uvozimo le ta objekt.

Nato moramo ustvariti spremenljivko tipa `TestProfile` z imenom `profile`. Ta se obnaša kot slovar in predstavlja vrh drevesa, v katerem se bodo nahajale vse skupine testov in testi novega *profila*. Teste združujemo v *skupine* z uporabo objekta `Group`, ki ga dodamo samemu profilu ali kakšni drugi skupini. Skupine se tudi obnašajo kot slovar in lahko vsebujejo tako druge skupine kot posamezne teste. *Teste* definiramo z objektom `PlainExecTest`, kateremu podamo izvršljivo datoteko in potrebne argumente.

Glede na to, kako poimenujemo posamezen ključ pri profilu in skupinah, tako bo potem ime objektu, na katerega kaže ta ključ.

Izvorna koda 4.5: Primer testnega profila

```
# Imports
from framework.core import Group, TestProfile
from framework.exectest import PlainExecTest

# Profile
profile = TestProfile()

group1 = Group()
group2 = Group()
subgroup1 = Group()
profile.tests['Group␣1␣name'] = group1
profile.tests['Group␣2␣name'] = group2
profile.tests['Group␣2␣name']['Subgroup␣1␣name'] = subgroup1

# Tests
group1['Test␣1␣name'] = PlainExecTest(['test1_executable', '↩
    first_arg', 'second_arg'])
group1['Test␣2␣name'] = PlainExecTest(['test2_executable'])
group2['Test␣3␣name'] = PlainExecTest(['test3_executable'])
group2['Test␣4␣name'] = PlainExecTest(['test4_executable'])
subgroup1['Test␣5␣name'] = PlainExecTest(['test5_executable'])
```

Profil za vse teste za standard OpenCL se nahaja v datoteki `test/cl.py`. V njem so testi razdeljeni v skupine po tipih testov in v nekaj ostalih podskupin.

4.2.3 Poganjanje profilov in prikaz rezultatov

Za poganjanje profila podamo profil izvršljivi datoteki `piglit-run.py`. Ta požene vse teste v profilu v ločenih procesih ter v izbrano mapo shrani rezultate in potek testiranja.

```
./piglit-run.py profile.py results/
```

V ciljno mapo se tako v različne datoteke v formatu JSON shranijo podatki o profilu, okolju, izvajanju testov in njihovih rezultatih.

Ker se naši testi poženejo na zadnji podprti verziji standarda OpenCL in na vseh platformah ter računskih napravah na sistemu, smo v izvajalcu testov izpostavili možnost nadziranja teh parametrov. Tako smo vanj dodali branje spremenljivk izvajalnega okolja, ki vplivajo na izvajanje testa:

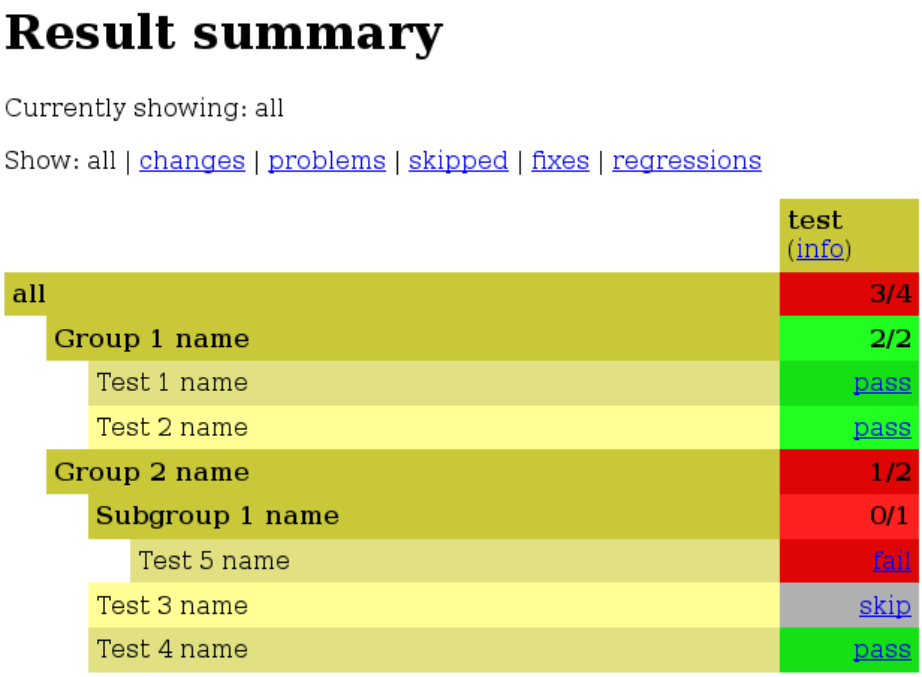
- **`PIGLIT_CL_VERSION`**: verzija standarda OpenCL
- **`PIGLIT_CL_PLATFORM`**: ime platforme OpenCL
- **`PIGLIT_CL_DEVICE`**: ime računske naprave OpenCL

Ob ustrezni nastavitvi spremenljivk se bodo testi pognali le za določeno verzijo standarda OpenCL in samo na izbrani platformi ter računski napravi.

Ko se izvedejo vsi testi, lahko iz dobljenih rezultatov generiramo grafični prikaz rezultatov. To storimo z uporabo izvršljive datoteke `piglit-summary-html.py`, kateri podamo ciljno mapo za generiranje HTML dokumentov in mapo z rezultati izvedenih testov.

```
./piglit-summary-html.py html/ results/
```

Če v ciljni mapi z mrežnim brskalnikom odpremo datoteko `index.html`, lahko vidimo povzetek testiranja in pregledamo rezultate posameznih testov. Primer prikaza povzetka testiranja za profil, določen z izvorno kodo 4.5, nam kaže slika 4.2.



Result summary

Currently showing: all

Show: all | changes | problems | skipped | fixes | regressions

	test (info)
all	3/4
Group 1 name	2/2
Test 1 name	pass
Test 2 name	pass
Group 2 name	1/2
Subgroup 1 name	0/1
Test 5 name	fail
Test 3 name	skip
Test 4 name	pass

Slika 4.2: Prikaz rezultatov orodja Piglit

Kot vidimo na sliki, je iz nastale internetne strani mogoče enostavno razbrati število uspešno in neuspešno izvedenih testov. Na vsakega izmed testov je mogoče klikniti ter podrobno pregledati izpis testa in napake, ki so se pojavile pri njegovi izvedbi.

Orodje Piglit nam poleg tukaj predstavljene funkcionalnosti ponuja še veliko več, vendar bi za opis celotnega orodja potrebovali veliko več prostora.

Poglavje 5

Testi

Testi predstavljajo največji del našega orodja. Medtem ko se ogrodje širi in spreminja počasi, se v testno orodje z veliko večjo hitrostjo dodajajo novi testi. Z njimi poskušamo pokriti celoten standard OpenCL in ga hkrati razbiti na čim manjše dele, da lahko vsako potrebno funkcionalnost posebej testiramo.

V tem poglavju bomo razdelali, kako so testi sestavljeni in katere tipe testov lahko uporabimo za postavitev testnega okolja. Za vsak tip testov bomo pogledali, čemu je namenjen in kako ga lahko konfiguriramo.

Pogledali bomo tudi dva realna testa in z njima razložili potek izvajanja testa.

5.1 Sestava testa

Vsak test je napisan v *svoji izvorni datoteki*. Tako so vsi testi ločeni in se lahko prevedejo v ločene izvršljive datoteke. Vsebina izvorne datoteke je razdeljena na **konfiguracijski** in **testni** del. S konfiguracijskim delom določimo, v kateri tip testov spada naš test in v kakšnem okolju se bo izvajal. S testnim delom testiramo želeno funkcionalnost implementacije standarda OpenCL in poročamo o uspešnosti izvedbe testa.

Poglejmo si primer praznega testa tipa custom (izvorna koda 5.1).

Izvorna koda 5.1: Prazen test tipa custom

```
#include "piglit-framework-cl-custom.h"

PIGLIT_CL_CUSTOM_TEST_CONFIG_BEGIN

    config.name = "Do␣nothing␣test";
    config.run_per_device = true;

PIGLIT_CL_CUSTOM_TEST_CONFIG_END


enum piglit_result
piglit_cl_test(const int argc,
               const char** argv,
               const struct piglit_cl_custom_test_config* config,
               const struct piglit_cl_custom_test_env* env)
{
    enum piglit_result result = PIGLIT_PASS;

    /* Test code */

    return result;
}
```

Konfiguracijski del se nahaja med makro izrazoma `PIGLIT_CL_CUSTOM_TEST_CONFIG_BEGIN` in `PIGLIT_CL_CUSTOM_TEST_CONFIG_END`. Ključna beseda `CUSTOM`, ki se nahaja v obeh izrazih, določa, da gre za tip testa z imenom custom. Med obema izrazoma lahko spreminjamo **konfiguracijsko podatkovno strukturo** `config`, kateri določimo razne konfiguracijske nastavitve. Te nastavitve vplivajo na izvajanje testa in vzpostavljanje okolja za izvedbo testa. Atributi konfiguracijske strukture so prikazani v tabeli 5.1. V našem primeru vidimo, da smo v konfiguraciji poimenovali test in želimo, da se izvede za vsako računsko napravo na vsaki platformi.

Testni del predstavlja **testna funkcija** `piglit_cl_test(...)`. V njej na-

pišemo test in vrnemo ustrezno vrednost glede na uspešnost testa. Glede na nastavljeno konfiguracijo se lahko ta funkcija izvede večkrat ali je izvedena samo na določenih platformah in računskih napravah. Ker smo v našem primeru nastavili atribut `config.run_per_device` na `true`, se bo funkcija izvedla za vsako računsko napravo na vsaki platformi. Testna funkcija za argumente prejme vhodne argumente izvršljive datoteke (`argc` in `argv`), konfiguracijsko podatkovno strukturo (`config`) in podatkovno strukturo okolja (`env`). **Podatkovna struktura okolja** vsebuje vrednosti, ki predstavljajo izvajalno okolje in jih za test ustvari tip testa. Atributi strukture okolja so prikazani v tabeli 5.2.

5.2 Tipi testov

Tipi testov nam nudijo možnost za večje deljenje kode in krajše pisanje testov. Z njimi ima pisec testov več *konfiguracijskih možnosti*, s katerimi ustvari bolj *specifično okolje* za teste, ki so si med seboj podobni.

Kot smo videli v poglavju 4, je nove tipe testov enostavno dodati, vendar imamo za trenutne potrebe implementirane tri različne tipe, predstavljene v nadaljevanju.

5.2.1 Testi custom

Testi custom so najbolj enostaven tip testov. Sem spadajo testi, ki ne ustrezajo nobeni drugi skupini, ker so preveč specifični za izvedbo v okoljih ostalih skupin.

V tej skupini postavimo *minimalno* okolje za izvedbo testov in ostalo prepustimo piscu testa. Testi tako nimajo dodanih nobenih novih atributov konfiguracije in okolja.

Primer testa za to okolje bi bil test, ki reproducira določen hrošč v neki implementaciji standarda OpenCL, kjer za reprodukcijo potrebujemo točno

Tabela 5.1: Atributi konfiguracijske podatkovne strukture za vse tipe testov

Atribut	**Opis**
`char* name`	Ime testa.
`bool run_per_platform`	Test se požene za vsako platformo.
`bool run_per_device`	Test se požene za vsako računsko napravo na vsaki platformi.
`char* platform_regex` `char* device_regex`	Test se izvede le na platformah in računskih napravah, ki ustrezajo regularnim izrazom.
`char* require_platform_extension` `char* require_device_extension`	Test se izvede le na platformah in računskih napravah, ki podpirajo naštete razširitve.
Atribut (API)	**Opis**
`int version_min` `int version_max`	Test se požene le, če je podprta verzija standarda OpenCL med izbranima vrednostima.
`bool create_context`	Ustvari kontekst OpenCL.
`char* program_source`	Izvorna koda, ki se prevede v program OpenCL za ustvarjeni kontekst.
`char* build_options`	Nastavitve za prevajanje izvorne kode programa OpenCL.
Atribut (program)	**Opis**
`int clc_version_min` `int clc_version_max`	Test se požene le, če je podprta verzija jezika OpenCL C med izbranima vrednostma

`char* program_source`
`char* program_source_file`
`unsigned char* program_binary`
`char* program_binary_file`
	Izvorna koda ali prevedeni program, ki se uporabi za program OpenCL.

`char* build_options`	Nastavitve za prevajanje izvorne kode programa OpenCL.
`bool expect_build_fail`	Pričakovano je, da prevajanje programa OpenCL ne uspe.
`char* kernel_name`	Ime jedra, za katerega se ustvari jedro OpenCL.

Tabela 5.2: Atributi podatkovne strukture okolja za vse tipe testov

Atribut	**Opis**
`int version`	Verzija standarda OpenCL.
`cl_platform_id platform_id`	Platforma OpenCL.
`cl_device_id device_id`	Računska naprava OpenCL.
Atribut (API)	**Opis**
`piglit_cl_context context`	Struktura, ki vsebuje kontekst OpenCL in nekaj drugih dodatnih spremenljivk.
`cl_program program`	Program OpenCL.
Atribut (program)	**Opis**
`int clc_version`	Verzija jezika OpenCL C.
`piglit_cl_context context`	Struktura, ki vsebuje kontekst OpenCL in nekaj drugih dodatnih spremenljivk.
`cl_program program`	Program OpenCL.
`cl_kernel kernel`	Jedro OpenCL.

določeno okolje in zaporedje ukazov. Trenutno so v tej skupini implementirani štiri testi.

5.2.2 Testi API

Testi API so namenjeni za testiranje programskega vmesnika API, ki ga mora implementirati vsaka implementacija standarda OpenCL. V tej skupini vsak test testira neko določeno vhodno funkcijo ali skupino simbolov. Za vsako testirano funkcijo mora biti testirano delovanje vsakega možnega načina uporabe tako pravilnega kot nepravilnega.

Glede na nastavljene atribute konfiguracijske strukture se za vsak klic testne funkcije lahko ustvarita ustrezni *kontekst* in prevedeni *program*. Dodatni atributi konfiguracije in okolja so podani v tabelah 5.1 in 5.2.

Primer testa za to okolje bi bil test, ki kliče vhodno funkcijo `cl_int clGetPlatformIDs(cl_uint, cl_platform_id*, cl_uint*)` z vsemi možnimi načini uporabe in preveri, ali so vrnjene vrednosti pravilne. Trenutno je v tej skupini implementiranih 31 testov.

5.2.3 Testi program

Testi program so namenjeni za testiranje prevajanja programov OpenCL in izvajanja jeder. Vsak test testira določeno lastnost oziroma funkcionalnost jezika OpenCL C, pravilnost izvajanja jedra ali pravilnost prevajanja programa.

Glede na nastavljene atribute konfiguracijske strukture se za vsak klic testne funkcije lahko ustvarijo ustrezni *kontekst*, prevedeni *program* in *jedro* iz prevedenega programa. Dodatni atributi konfiguracije in okolja so podani v tabelah 5.1 in 5.2.

Primer testa za to okolje bi bil test, ki preveri, ali izvajanje jedra deluje pravilno, če za velikost delovne skupine uporabimo največjo možno vrednost, ki jo oglašuje testirana računska naprava. Trenutno so v tej skupini implementirani tri testi. Od tega eden izmed njih deluje kot nov tip oziroma podtip testov, saj omogoča testiranje poljubnih izvornih datotek OpenCL C s posebno definicijo konfiguracije testa. Podrobneje je opisan v naslednjem podpoglavju.

Testi OpenCL C program

Poseben podtip *testov program* so **testi OpenCL C program**. To so testi, ki so v celoti napisani v izvorni datoteki OpenCL C ter testirajo prevajanje programov OpenCL C in izvajanje jeder. Izvedejo se tako, da so podani kot vhodni argument izvršljivi datoteki `bin/cl-program-tester`. Piscu testov tako ni potrebno pisati nobene kode C, ki bi prevajala in klicala testirani program OpenCL. Za primer poglejmo izvorno kodo praznega testa tipa OpenCL C program (Izvorna koda 5.2).

Izvorna koda 5.2: Prezen test tipa OpenCL C program

```
/*!
[config]
name: Do nothing tests

[test]
```

```
name: Do nothing test 1
kernel_name: test_func

# A comment
!*/

kernel void test_func() {
    /* Execution test code */
}
```

Kot lahko vidimo, so tudi tukaj testi ločeni na *konfiguracijski* in *testni* del kot pri vseh drugih tipih testov. Ker je test definiran samo z izvorno kodo OpenCL C, je konfiguracijski del testa podan kot komentar. Označen je s posebnim večvrstičnim komentarjem, kjer dodamo klicaje na začetku in koncu komentarja (`/*! ... !*/`). Pod tem komentarjem se nahaja testni del, kjer so podana jedra, ki se poženejo pod pogoji, navedenimi v konfiguracijskem delu.

Ker želimo s to skupino testov pokriti večino testov programov OpenCL in ker tu ne uporabljamo nobenega programskega jezika za prilagajanje okolja, smo dodali več konfiguracijskih možnosti kot pri ostalih tipih testov, da lahko natančneje nadziramo izvajanje.

Kot smo že povedali, se konfiguracijski del nahaja v posebno označenem večvrstičnem komentarju. Format konfiguracije v komentarju je podoben tekstovni datoteki `INI`. Razdelki so določeni z besedami v oglatih oklepajih (`[]`), lastnosti in vrednosti so ločene z dvopičjem (`:`) in komentarji se začnejo s številskim znakom (`#`).

Obvezni razdelek v konfiguraciji je razdelek `config`. V njem lahko določimo iste možnosti kot pri navadnih, v jeziku C napisanih, testih tipa program (tabela 5.1). Poleg tega lahko definiramo dodatne možnosti, s katerimi bolj natančno nadziramo izvajanje jeder. Te možnosti so prikazane v tabeli 5.3.

Drugi podprt razdelek je razdelek `test`. Teh razdelkov je lahko poljubno, vendar mora biti definiran *vsaj eden*. Vsak razdelek `test` predstavlja *podtest*, ki testira izvajanje nekega jedra. Podprte so enake možnosti kot pri razdelku

`config` iz tabele 5.3. Če katera izmed njih ni uporabljena, se uporabi vrednost, nastavljena v razdelku `config`. Ostale dodatne nastavitve lastne samo razdelku `test` so vhodni in izhodni argumenti jedra (tabela 5.3).

Vhodni argumenti definirajo tipe, ki jih jedro sprejme, in vrednosti, s katerimi se *inicializirajo*. **Izhodni argumenti** prav tako definirajo tipe in vrednosti, vendar tu namesto vrednosti za inicializacijo naštejemo *pričakovane* vrednosti argumentov po končani izvedbi jedra. Vhodni argumenti so lahko tudi izhodni argumenti, ampak morata biti tipa pri obeh definicijah ista. Format zapisa argumentov je naslednji:

```
(arg_in|arg_out): indeks tip vrednost [tolerance odstopanje]
```

***indeks*:** Indeks argumenta jedra (nenegativno celo število).

***tip*:** Tip argumenta. Lahko je eden izmed skalarnih (**`int`**, **`float`** ...) ali vektorskih tipov (**`int4`**, **`float4`** ...), ali medpomnilnik enega izmed tipov (`buffer` **`int`**`[16]`, `buffer` **`float4`**`[32]` ...)

***vrednost*:** Vhodna vrednost ali pričakovana izhodna vrednost. Za vektorske tipe in medpomnilnike so posamezne vrednosti ločene s presledki.

***odstopanje*:** Dovoljeno odstopanje vrednosti izhodnega argumenta (pozitivno število).

Podtesti so izvedeni eden za drugim in celoten test je uspešen samo ob uspešni izvedbi *vseh podtestov*. Posamezen podtest je uspešen, če so vsi izhodni argumenti enaki pričakovanim ali je program samo uspešno preveden, če smo samo to testirali.

Primer testa tipa OpenCL C program bi bil test, ki preveri, če jedro pravilno prekopira vrednosti iz enega argumenta jedra v drugega. Trenutno je v tej skupini implementiranih 96 testov in 227 testov je avtomatsko generiranih s skriptami. Vsak test vsebuje večje število podtestov, ki se prikažejo

Tabela 5.3: Dodatni atributi konfiguracijske podatkovne strukture za teste tipa OpenCL C program

Atribut (`config` in `test`)	**Opis**
`char* kernel_name`	Ime jedra za izvajanje testa.
`bool expect_test_fail`	Pričakovano je, da podtest ne uspe.
`uint dimensions`	Število dimenzij (ND-range) za globalni prostor. Veljavne vrednosti: $[1, 3]$.
`uint[3] global_size`	Velikost vseh dimenzij globalnega prostora. Veljavne vrednosti posamezne dimenzije: $[0, \infty]$.
`uint[3] local_size`	Velikost vseh dimenzij lokalnega prostora. Veljavne vrednosti posamezne dimenzije: $[0, \infty]$.
Atribut (`test`)	**Opis**
`char* name`	Ime podtesta.
`char* arg_in` `char* arg_out`	Vhodni in izhodni argumenti jedra.

pri rezultatih orodja kot posamezni testi. Takšen tip testov naj bi se največ uporabljal in zato tukaj pričakujemo največje število testov.

5.3 Primeri testov

Za lažjo predstavo si poglejmo dva realna primera testov, enega v jeziku C in drugega v jeziku OpenCL C. Za jezik C bomo pogledali test iz skupine *testov API*, medtem ko bo za jezik OpenCL C test iz skupine *testov OpenCL C program*.

Analizirajmo najprej *test API*, ki testira funkcijo `clEnqueueCopyBuffer(...)` (izvorna koda 5.3). Na začetku izvorne kode imamo vključene potrebne datoteke C, nato takoj za tem sledi konfiguracijski del. Glede na izbrane nastavitve vidimo, da se bo test izvedel le, če je verzija standarda testirane platforme večja ali enaka 1.0. Poleg tega bo test pognan za vsako platformo

na sistemu in za vsak klic testne funkcije se bo ustvaril posebni kontekst.

Testna funkcija v prejetem kontekstu (`env->context`), ki je del strukture okolja, prejete kot argument, najprej ustvari dva medpomnilnika z različno vsebino. Nato kliče funkcijo `clEnqueueCopyBuffer(...)`, ki bi po standardu morala prekopirati vsebino enega medpomnilnika v drugega. Vsebina ciljnega pomnilnika je nato prebrana in preverjena, ali je enaka začetni vsebini prvega pomnilnika. Če na nobenem koraku ne pride do napake, testna funkcija vrne vrednost za uspešen rezultat (`PIGLIT_PASS`).

Pomembno je vedeti, da za vsak test predvidevamo, da ostale funkcije vmesnika API standarda OpenCL, ki jih ne testiramo, delujejo pravilno. Kljub temu je vseeno dobra praksa, da ob izvajanju preverimo, ali ni prišlo do izvajalne napake pri vsaki takšni klicani funkciji.

Izvorna koda 5.3: Test tipa API za vhodno funkcijo `clEnqueueCopyBuffer(...)`

```
#include "piglit-framework-cl-api.h"
#include "piglit-util-cl.h"

PIGLIT_CL_API_TEST_CONFIG_BEGIN

    config.name = "clEnqueueCopyBuffer";
    config.version_min = 10;

    config.run_per_platform = true;
    config.create_context = true;

PIGLIT_CL_API_TEST_CONFIG_END

enum piglit_result
piglit_cl_test(const int argc,
           const char **argv,
           const struct piglit_cl_api_test_config* config,
           const struct piglit_cl_api_test_env* env)
{
    int host_src_buffer[4] = {1, 2, 3, 4};
    int host_dst_buffer[4] = {0, 0, 0, 0};
    cl_mem device_src_buffer, device_dst_buffer;
```

```
cl_command_queue queue = env->context->command_queues[0];
cl_int err;
int i;

/* Initialize buffers */
device_src_buffer = piglit_cl_create_buffer(
    env->context, CL_MEM_READ_WRITE, sizeof(host_src_buffer))
;
device_dst_buffer = piglit_cl_create_buffer(
    env->context, CL_MEM_READ_WRITE, sizeof(host_dst_buffer))
;
if (!piglit_cl_write_whole_buffer(queue,
                device_src_buffer, host_src_buffer) ||
    !piglit_cl_write_whole_buffer(queue,
                device_dst_buffer, host_dst_buffer)) {
    return PIGLIT_FAIL;
}

/* Copy buffers */
err = clEnqueueCopyBuffer(queue, device_src_buffer, device_
dst_buffer,
            0, 0, sizeof(host_src_buffer), 0, NULL, NULL);
if (!piglit_cl_check_error(err, CL_SUCCESS)) {
    return PIGLIT_FAIL;
}

/* Read copied buffer */
if (!piglit_cl_read_whole_buffer(queue, device_dst_buffer,
                        host_dst_buffer)) {
    return PIGLIT_FAIL;
}

/* Check copied buffer data */
for (i = 0; i < sizeof(host_src_buffer) / sizeof(host_src_
buffer[0]); i++) {
    if (!piglit_cl_probe_integer(host_dst_buffer[i],
                    host_src_buffer[i], 0)) {
        fprintf(stderr, "Error␣at␣%d\n", i);
        return PIGLIT_FAIL;
    }
}
```

```
    return PIGLIT_PASS;
}
```

Za *test OpenCL C program* bomo analizirali test funkcije vgrajene v jezik OpenCL C, imenovane `get_work_dim()` (izvorna koda 5.4). Na začetku izvorne kode se nahaja konfiguracijski del, kjer sta v razdelku `config` določena minimalna verzija jezika OpenCL C, ki definira funkcijo `get_work_dim()`, in ime jedra skupnega za vse podteste.

Nato sledijo trije razdelki `test` oziroma podtesti, ki poženejo isto jedro z različnimi dimenzijami in globalnim prostorom. Za izhodni argument je povsod izbran medpomnilnik s pričakovano vsebino števila uporabljenih (delovnih) dimenzij. Tako vsak podtest preveri, če jedro pravilno vrne število dimenzij, s katerimi je bilo pognano.

Na dnu izvorne kode imamo v testnem delu definirano eno jedro, katerega ime odgovarja imenu definiranem v razdelku `config` in ga zato poženejo vsi podtesti. To jedro ima enostavno nalogo in samo zapiše v prejeti medpomnilnik vrednost, ki jo vrne funkcija `get_work_dim()`.

Če se vsi podtesti pravilno izvedejo in dobijo po izvedbi jedra v medpomnilniku pričakovano vrednost, je test uspešen.

Izvorna koda 5.4: Test tipa OpenCL C program za uporabo dimenzij globalnega prostora

```
/*!
[config]
name: get_work_dimensions
clc_version_min: 10
kernel_name: builtin_work_dim

[test]
name: get_work_dim (1)
dimensions: 1
global_size: 1 0 0
arg_out: 0 buffer int[1] 1

[test]
```

```
name: get_work_dim (2)
dimensions: 2
global_size: 1 1 0
arg_out: 0 buffer int[1] 2

[test]
name: get_work_dim (3)
dimensions: 3
global_size: 1 1 1
arg_out: 0 buffer int[1] 3
!*/

kernel void builtin_work_dim( __global int *ret ) {
  *ret = get_work_dim();
}
```

Poglavje 6

Testiranje

Orodje Piglit se neprestano razvija in spreminja. Njegovi razvijalci ne izdajajo nobenih oštevilčenih izdaj in priporočajo, da se za testiranje uporablja zadnja razvojna verzija programa, ki je dostopna na javnem repozitoriju. Tako je za uporabo potrebno prenesti celoten projekt na testni sistem, ga tam prevesti in potem pognati. Ker je večina uporabnikov testnega orodja tudi razvijalcev programske opreme, naj to ne bi predstavljalo nobene omejitve glede uporabe, ampak le spodbujalo uporabo zadnjih dodanih testov.

V tem poglavju bomo analizirali rezultate testov *dveh implementacij* standarda OpenCL na *dveh različnih računskih napravah*. Opisali bomo okolje, v katerem so se testi izvajali, do kakšnih rezultatov smo prišli in kaj nam ti rezultati predstavljajo.

Poleg tega bomo pogledali, kako se je naše orodje razvijalo skozi čas in koliko novih testov je pridobilo. Zato bomo v analizo vključili *dve testiranji*, eno izvedeno kmalu po implementiranem orodju in drugo v času pisanja diplomske naloge, leto in pol kasneje. Večina mojega dela na orodju je bila izvedena pred prvim testiranjem. Po njem je sledila vključitev v orodje Piglit, kjer so začeli sodelovati tudi ostali razvijalci. Od takrat so oni orodje razširili, odpravili nekaj napak in dodali veliko množico ostalih testov.

6.1 Okolje

Imenujmo naši dve testiranji kar **testiranje 1** in **testiranje 2**. Vsako testiranje je bilo izvedeno z različnimi verzijami orodja Piglit in implementacijami standarda OpenCL. Testiranje 1 je bilo izvedeno z verzijami dostopnimi na datum *21.9.2012*, medtem ko je bilo testiranje 2 izvedeno z verzijami dostopnimi na datum *10.2.2014*.

Prvi datum je bil izbran na podlagi tega, kdaj je bilo naše orodje vključeno v skupni razvoj z orodjem Piglit, medtem ko je drugi nek poljuben datum iz časa pisanja diplomske naloge.

Za testiranji smo izbrali standard *OpenCL 1.1*. Obe testiranji sta bili izvedeni na istih računskih napravah in programski opremi. Za računski napravi smo izbrali dve namensko različni napravi, prisotni na istem sistemu, CPE in GPE. Dejanski testirani napravi sta:

- *Intel Core i5 460M*,
- *ATI Mobility Radeon HD 5470*.

Testiranje je potekalo na operacijskem sistemu GNU/Linux in vsaka izmed naprav je uporabljala svojo implementacijo standarda OpenCL. Imeni implementacij in uporabljene verzije so podane v tabeli 6.1.

Implementacija standarda OpenCL za procesor Intel je zaprtokodna in smo zato za obe testiranje uporabili zadnjo prevedeno verzijo dostopno na izbrana datuma. Za grafično kartico Radeon smo na izbrana datuma prevedli zadnjo razvojno kodo odprtokodnih gonilnikov grafične kartice kot tudi vmesnika clover. Pri obeh napravah in implementacijah se je za testiranje uporabljala na izbrani datum prevedena zadnja razvojna verzija orodja Piglit.

Tabela 6.1: Testirane naprave in platforme

Testiranje	**Naprava**	**Implementacija**	**Verzija**
1 (21.9.2012)	Core i5 460M	Intel SDK for OpenCL Applications	2012 R1
	Radeon HD 5470	Mesa radeon + Gallium3D clover	git (razvoj)
2 (10.2.2014)	Core i5 460M	Intel SDK for OpenCL Applications	2013 R3
	Radeon HD 5470	Mesa radeon + Gallium3D clover	git (razvoj)

6.2 Rezultati

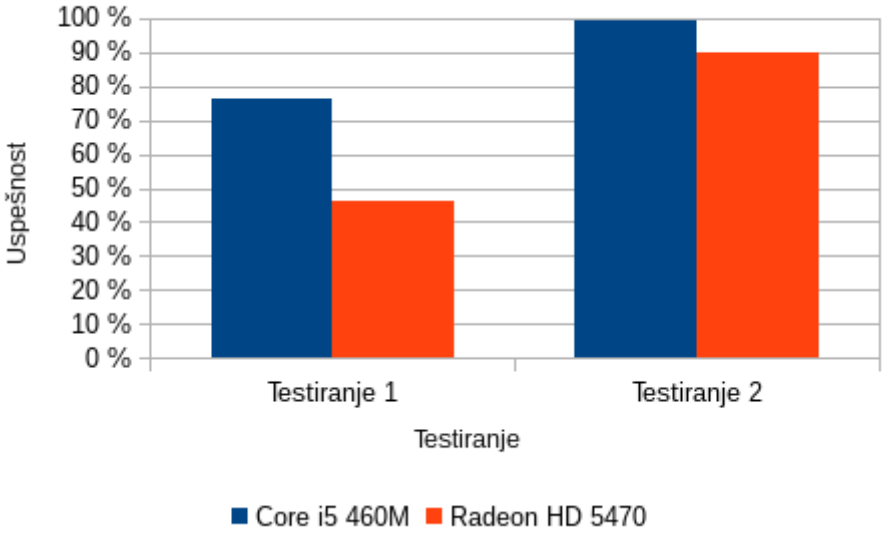
Po pognanih testih iz obeh testiranj na obeh napravah smo prišli do rezultatov, prikazanih v tabeli 6.2. V zadnjem stolpcu imamo podano število uspešno opravljenih testov računske naprave glede na vse razpoložljive teste pri izbranem testiranju. Razvidno je, da ima testiranje 1 na razpolago samo 70 testov, medtem ko ima testiranje 2 nad 1500 testov. Za lažjo predstavo rezultatov je podan graf uspešnosti izvedenih testiranj (slika 6.1).

Ob hitrem pregledu rezultatov lahko razberemo, da se je implementacija podjetja Intel obnesla bolje pri obeh testiranjih. To ne pomeni nujno, da je boljša ali hitrejša za uporabo, ampak le, da je bolj skladna s standardom OpenCL.

Za informacijo podajmo še, da se je testiranje 1 izvajalo na eni računski napravi v povprečju okoli 100 sekund, medtem ko se je testiranje 2 izvajalo okoli 600 sekund.

Tabela 6.2: Rezultati testiranj

Testiranje	**Naprava**	**Rezultat (uspešni/vsi)**
1 (21.9.2012)	Core i5 460M	55/70 (76%)
	Radeon HD 5470	32/70 (46%)
2 (10.2.2014)	Core i5 460M	1519/1532 (99%)
	Radeon HD 5470	1376/1532 (90%)



Slika 6.1: Graf rezultatov testiranj

6.3 Analiza rezultatov

Poglejmo najprej uspešnost testov pri posameznih testiranjih. Pri testiranju 1 lahko vidimo, da sta kljub majhnemu številu testov imeli obe implementaciji *probleme pri četrtini ali več testov*. Podrobni pregled posameznih rezultatov je pokazal, da se je večina napak pojavila pri robnih primerih klicev različnih funkcij vmesnika API. Implementacije so ob pravilni rabi funkcij vmesnika API delovale pravilno, ampak niso javljale napak ob nepravilni uporabi, ko so na primer prejele neveljavne argumente. Pri implementaciji podjetja AMD je bilo to bolj opazno, ker je bila implementacija še v zgo-

dnjem razvoju in se razvijalci še niso posvečali tem robnim primerom.

Pri testiranju 2 vidimo, da sta se obe implementaciji znatno izboljšali in približali polnemu pokritju uspešno izvedenih testov. Intelova implementacija je bila uspešno pri *99% testov* in ni pravilno delovala le pri nekaj testih tipa API in testih jeder, kjer je rezultat izstopal iz potrebnega območja natančnosti. Implementacija podjetja AMD se je obnesla nekoliko slabše in je dobila pozitiven rezultat pri *90% testov*. Napake so bile podobne kot pri implementaciji podjetja Intel, le da je bilo tu več napak pri izvajanju in prevajanju jeder.

Ker je nekaj napak v implementaciji podjetja AMD privedlo do zrušitve programa, se nekaj podtestov v testiranju sploh ni izvedlo. Kljub temu nismo mogli šteti neopravljene teste med uspešne in smo jih zato dodali v neuspešne. To pomeni, da je dejanski rezultat implementacija podjetja AMD lahko, in verjetno tudi je, boljši od tukaj navedenega.

Napredek implementacij v pokritosti testov v letu in pol med obema testiranjema je za podjetje Intel *23%* in za podjetje AMD *44%*. Lahko vidimo, da obe implementaciji skrbno sledita standardu in sta med izbranima časoma testiranj odpravili veliko napak. Žal ne moremo direktno izmeriti, koliko izboljšav je posledica uporabe našega orodja ali samo nadaljnjega razvoja implementacije. Lahko pa sklepamo, ker razvijalci obeh podjetij dodajajo teste za naše orodje in ker se pokritost uspešno izvedenih testov močno veča, da se implementirano orodje uspešno uporablja in dejansko pripomore pri odpravljanju napak.

Če pogledamo število pognanih testov pri vsakem testiranju, lahko opazimo, da se je v *letu in pol* število testov *povečalo za skoraj 1500*. To pomeni, da orodje Piglit sedaj pokriva znatno več standarda OpenCL kot ga je pri združitvi z našim orodjem. Koliko standarda je pokritega, žal ne moremo enostavno izvedeti, saj bi za to potrebovali zelo natančen pregled vseh testov

in podrobno poznavanje celotnega standarda. Lahko pa vidimo, da kljub zrelosti obeh implementacij naši testi še vedno odkrivajo napake.

Poglavje 7

Sklepne ugotovitve

V diplomskem delu smo razvili orodje za testiranje pravilnosti delovanja implementacij standarda OpenCL. Omogoča enostavno izvedbo testov, grafični prikaz rezultatov in hitro pisanje novih testov. Čeprav z implementiranimi testi nimamo še pokritega celotnega standarda, smo z orodjem postavili stabilno osnovo, ki je razširljiva in se uspešno uporablja.

Pisanje testov za implementirano orodje je dokaj enostavno in intuitivno. Teste smo ločili na konfiguracijski in testni del, kar pripomore k logični in vizualni razdelitvi postavljanja testnega okolja in izvajanja testa. Konfiguracijski del je sestavljen iz priredbe atributov podatkovnih struktur, ki manipulirajo ogrodje našega orodja za postavljanje ustreznega izvajalnega okolja. Testni del je na drugi strani sestavljen iz kratke testne funkcije oziroma programa OpenCL C, ki vsebuje samo potrebno kodo za izvedbo testa. S takšno razdelitvijo oblike testov je pisanje testov tudi hitro in osredotočeno.

Ker ne moremo vnaprej poznati vse možne potrebe po izvajalnem okolju za različne teste, smo poskrbeli da je ogrodje razširljivo. To smo izvedli z uvedbo testnih tipov, ki testom podajo nove konfiguracijske možnosti in glede na njih postavijo primerno izvajalno okolje. Trenutno imamo implementirane štiri takšne tipe testov in pričakujemo, da se bo njihovo število večalo skupaj z večanjem uporabe orodja.

Vključitev našega orodja v orodje Piglit je omogočila grafičen prikaz rezultatov ter povečala bazo razvijalcev in uporabnikov orodja. Implementacija orodja je bil sponzoriran projekt in želja skupnosti X.Org, zato vključitev v njihovo orodje Piglit ne pomeni nujno uspešno izvedbo projekta. Ampak glede na uporabo našega orodja v času po vključitvi - med njo štejejo povečanje števila testov iz 70 na okoli 1500, uporaba in omemba orodja pri raznih projektih[1] [2] [3] ter konstanten aktiven razvoj - lahko sklepamo, da je bilo orodje pozitivno sprejeto, se uspešno uporablja in je doseglo zadane cilje.

Izmed vseh lastnosti orodja bi kot njegovo glavno prednost izpostavili pisanje testnih programov v jeziku OpenCL C brez uporabe vmesnika API za prevajanje in izvedbo programa. Tako imenovani testi OpenCL C program podpirajo statično nastavljanje vhodnih argumentov jeder programa in tudi primerjanje izhodnih argumentov s pričakovanimi vrednostmi glede na določeno odstopanje. S tem pokrijemo ogromno količino možnih testov za prevajanje in izvajanje programov OpenCL C, kar je navadno tudi največji in implementacijsko gledano najzahtevnejši del implementacij standarda OpenCL.

Na drugi strani je slabost našega orodja dejstvo, da trenutno nimamo nobenega merila, ki bi nam povedalo, kolikšen del standarda pokriva. Za to bi potrebovali natančen pregled celotnega standarda in razdelitev posameznih delov na testirane in netestirane. Takšen podvig bi vzel veliko časa, ampak bi s tem pridobili natančne smernice za pisanje novih testov. Poleg tega potrebujemo tudi izboljšavo načina pisanja testov tipa OpenCL C program, kjer imamo težavo s ponavljajočim pisanjem zelo podobnih testov. Testi se prepogosto ponavljajo, ker moramo velikokrat napisati isti test za vse možne tipe spremenljivk, podprte v jeziku OpenCL C.

[1] pocl: `https://raw.github.com/pocl/pocl/master/CHANGES`

[2] linaro GPGPU: `http://lists.linaro.org/pipermail/linaro-gpgpu/2013-April/000000.html`

[3] Beignet: `http://lists.freedesktop.org/archives/beignet/2014-February/002495.html`

Logično je, da želimo v prihodnosti odpraviti vse slabosti našega orodja. Za pregled pokritosti standarda OpenCL trenutno nimamo nobenih načrtov, vendar za problem ponavljajočih se testov tipa OpenCL C program, imamo že idejno rešitev. Testom želimo v konfiguracijski del dodati atribut s seznamom, katerega vrednosti bi nadomestile izbrano ključno besedo, uporabljeno v izvorni kodi testa. Tako bi lahko na primer nadomestili besedo `TYPE_GENERATOR` z besedami **`short4`**, **`int4`** in **`long4`** ter za vsako substitucijo pognali test. Nadobudnemu bralcu prepuščamo implementacijo za vajo in v primeru uspeha vsekakor tudi vključitev izvorne kode v orodje Piglit.

Seveda je še zmeraj glavni in osnovni cilj našega orodja, da pridobi zadostno količino testov za pokritje vseh verzij standarda OpenCL. Na tem področju se trudimo z večanjem uporabnikov in razvijalcev orodja ter aktivnim razvojem in sodelovanjem z drugimi projekti.

Seznam kratic

AMD Advanced Micro Devices. 5, 8, 56, 57

API Application Programming Interface. 8, 10, 11, 19, 21, 32, 43, 48, 60

BSD Berkeley Software Distribution. 4

CPE centralna procesna enota. 1, 7, 9, 14, 54

DSP enota za procesiranje signalov. 1

EVoC Endless Vacation of Code. 5

FOSS Free and Open Source Software. 4, 5, 17, 20

GPE grafična procesna enota. 1, 7, 9, 54

GPL GNU General Public License. 4

HTML HyperText Markup Language. 18, 36

IEEE Institute of Electrical and Electronics Engineers. 14

ISO International Organization for Standardization. 14

MIT Massachusetts Institute of Technology. 4

OpenCL Open Computing Language. 1–3, 5–15, 17–20, 22–25, 29, 31, 32, 35, 36, 39, 41–45, 48, 53–55, 57, 59–61, 65

SCM Source Code Management. 20

Slike

Tabele

Seznam izvorne kode

Literatura

[1] Ohloh repository statistics. Dostopno na: `https://www.ohloh.net/repositories/compare`. Prebrano: 20.2.2014. 20

[2] Programming language popularity. Dostopno na: `http://langpop.com/`. Prebrano: 20.2.2014. 19

[3] X.org fundation. Dostopno na: `http://www.x.org/wiki/XorgFoundation/`. Prebrano: 25.11.2013. 3

[4] Khronos OpenCL Working Group et al. The opencl specification. *A. Munshi, Ed*, 2008. 7

[5] OpenCL Working Group et al. *The OpenCL specification, version 1.2, revision 16*, 2011. 7

[6] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. 2010. 8

[7] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*, chapter 1. Addison-Wesley Professional, 2011. 8

[8] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*, chapter 4. Addison-Wesley Professional, 2011. 14

[9] Eric S Raymond. *The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary*. O'Reilly Media, 2001. 4

[10] Neil Trevett. Opencl overview. Dostopno na: https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf. Prebrano: 15.1.2014. 15

[11] Fahad Zafar, Dibyajyoti Ghosh, Lawrence Sebald, and Shujia Zhou. Accelerating a climate physics model with opencl. 8