Proširenje mogućnosti OpenCL stoga otvorenog koda za grafičke procesore AMD Radeon

Varelija, Dominik

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:090600

Rights / Prava: Attribution 4.0 International/Imenovanje 4.0 međunarodna

Download date / Datum preuzimanja: **2024-03-29**

Sveučilište u Rijeci
Fakultet informatike
i digitalnih tehnologija

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

zir.nsk.hr

DIGITALNA KNJIŽNICA

Image not found or type unknown

Sveučilište u Rijeci – Odjel za informatiku

Jednopredmetni diplomski studij informatike

Dominik Varelija

Proširenje mogućnosti OpenCL stoga otvorenog koda za grafičke procesore AMD Radeon

Završni rad

Mentor: v. pred. dr. sc. Vedran Miletić

Rijeka, 9. studenog 2020.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 19. lipnja 2020.

Zadatak za diplomski rad

Pristupnik:

Naziv diplomskog rada: Proširenje mogućnosti OpenCL stoga otvorenog koda za grafičke procesore AMD Radeon

Naziv diplomskog rada na eng. jeziku: Extending the capabilities of the open source OpenCL stack for AMD Radeon GPUs

Sadržaj zadatka:

NVIDIA je vodeća tvrtka u domeni vizualizacije i računanja na grafičkim procesorima. U domeni računanja na grafičkim procesorima s NVIDIA-om se natječu AMD i Intel, nudeći podršku za otvoreni standard OpenCL kao alternativu NVIDIA-inoj vlasničkoj tehnologiji CUDA. AMD, pored korištenja otvorenog standarda OpenCL, nudi stog (program-prevoditelj Clang/LLVM, biblioteke, upravljačke programe) otvorenog koda Clover za grafičke procesore Radeon, temeljen na biblioteci Mesa (Gallium). Zadatak rada je usporediti mogućnosti vlasničkog stoga za grafičke procesore tvrtke NVIDIA i stoga otvorenog koda za grafičke procesore AMD Radeon i proširiti mogućnosti stoga da podržava skup alata za molekularne simulacije OpenMM.

Mentor:

v. pred. dr. sc. Vedran Miletić

Komentor:

Voditeljica za diplomske radove:

izv. prof. dr. sc. Ana Meštrović

Zadatak preuzet: 23. lipnja 2020.

Dominik Varelija

(potpis pristupnika)

Sažetak

U ovom radu opisana je tema popravljanja OpenCL implementacije Clover u AMD upravljačkom stogu koristeći Piglit i OpenMM unit testove. Proučava se AMD upravljački stog sa fokusom na Clover i nudi se pregled arihitektura AMD GPU-a. Radi se na popravljanju dijelova implementacije koji ne zadovoljavaju Piglit i OpenMM unit testove koristeći alate GDB i Valgrind. OpenMM je program za simulaciju molekularnih sila i koristi se za predviđanje interakcije proteina i različitih fluida. Clover je unaprijeđen za Piglit i OpenMM unit testove.

Ključne riječi

OpenCL, Mesa, Clover, Piglit, OpenMM

Sadržaj

1. Uvod....1
2. Arhitektura AMD GPU-a....3
   2.1. Povijest grafičkih procesora....3
   2.2. Razlike između CPU-a i GPU-a....5
   2.3. Arhitekture AMD Radeon GPU-a....5
      2.3.1. Arhitektura TeraScale....5
         2.3.1.1. TeraScale 1....6
         2.3.1.2. TeraScale 2....7
         2.3.1.3. TeraScale 3....7
      2.3.2. Arhitektura GCN (Graphics Core Next)....7
         2.3.2.1. Arhitektura GCN 1: Southern Islands....8
         2.3.2.2. Arhitektura GCN 2: Sea Islands....8
         2.3.2.3. Arhitektura GCN 3: Volcanic Islands i Pirate Islands....8
         2.3.2.4. Arhitektura GCN 4: Artic Islands ili Polaris....9
         2.3.2.5. Arhitektura GCN 5: Vega....9
      2.3.3. Arhitektura RDNA: Navi....9
      2.3.4. Arhitektura RDNA2: Big Navi....10
   2.4. Moderan GPU kao skup blokova intelektualnog vlasništva....11
3. Upravljački stog na Linux sustavu....14
   3.1. Mesa....14
      3.1.1. Gallium3D....15
      3.1.2. Clover....15
   3.2. LLVM....16
   3.3. Kernel....16
   3.4. Pregled stoga upravljačkih programa za Radeon GCN/RDNA....17
4. Piglit....20
   4.1. Pokretanje OpenCL testova....22
   4.2. Rezultati testova za Clover implementaciju standarda OpenCL....23
5. OpenMM....25
   5.1. Općenito o programu OpenMM....25
   5.2. OpenMM testovi....26
      5.2.1. Pokretanje testova....26
      5.2.2. Primjer testa....27
6. Izvođenje OpenMM testova na Clover-u....30
7. Zaključak....42

1. Uvod

Moderni grafički procesori započinju priču u 1970-im godinama, kao vrlo specijalizirani paralelni procesori namijenjeni procesiranju i prikazu slika iz memorije, imali su samo jednu zadaću i vrlo mali set operacija koje su mogli izvršavati nad podatcima. Razvojem tehnologije i povećanjem potrebe za izvršavanjem paralelnog procesiranja dolazi do promjene same arhitekture i uloge GPU-a. Grafički procesori dobivaju programibilne shadere koji omogućuju izvođenje funkcija koje su do tada bile rezervirane samo za CPU. Ovim pristupom omogućeno je ubrzano izvršenje paralelnih procesa koji imaju trivijalne funkcije pretvorbe podataka, a u današnje vrijeme najveće ubrzanje izvedbe vidi se upravo u brzini treniranja i izvođenja algoritama za strojno učenje i umjetnu inteligenciju. Suvremeni GPU-ovi danas osim programabilnih shadera imaju mogućnosti kodiranja i enkodiranja videa, te ubrzavanja obrade specijaliziranim jezgrama na GPU-ima.

Grafički procesori se nalaze u većini osobnih računala i svim suvremenim superračunalima. Industrija koja se bavi proizvodnjom GPU-a naglo se razvila zajedno sa povećanjem potražnje na tržištu. GPU-i se koriste za računalne igre, compute, treniranje i korištenje AI algoritama, renderanje videa i simulacije u filmskoj industriji. GPU-ovi danas osim napredaka u rasterizaciji donose nove specijalizirane mogućnosti poput tenzor jezgri [1] koje omogućuju veliki napredak u brzini korištenja AI algoritama. Zahvaljujući Tenzor jezgrama vrijeme potrebno za treniranje AI modela znatno je reducirano, a uz to otvara mogućnost korištenja AI u realnom vremenu za poboljšanje slike u slučaju Deep Learning Super Sampling (kraće DLSS [2]) ili izoliranja glasa od pozadinske buke sa RTXVoice [3]. Mogućnosti brzog treniranja i korištenja AI-a dovela je do korištenja GPU-a u svrhe predviđanja vremena, vožnje autonomnih vozila, predviđanja bankovnih prijevara i mnogih drugih primjena.

GPU-i danas sadrže posebne specijalizirane jezgre, tzv. akceleratore zraka svjetlosti (engl.ray accelerator, kraće RA [4]) za praćenje zrake svjetlosti (engl. ray tracing) u realnom vremenu. RA jezgrama omogućava se vrlo brzo izvođenje računalno jako zahtjevne tehnike za renderanje, a koristi se za produkciju i računalne igre. Velika moć paralelnog procesiranja na GPU-ima može se koristiti za kriptografiju kao i za napad grubom silom (engl. brute force attack) na probijanje lozinke zbog čega je važnije nego ikad koristiti snažne lozinke i dvostruku autentifikaciju. Novije grafičke kartice mogu isprobavati i nekoliko milijuna kombinacija hasheva i lozinki u sekundi što dovodi do potrebe korištenja jačih kriptografskih algoritama i duljine lozinke.

Unatoč brojnim primjenama GPU-a danas, još uvijek se u velikoj većini prodaju i koriste u svrhu

računalnih igara. Kako bi GPU-i bili u mogućnosti prikazati virtualne svjetove potrebne su implementacije OpenGL-a, Vulkana i ostalih programskih jezika za grafiku. Linux nudi mogućnost odabira između vlasničkih upravljačkih programa i upravljačkog programa otvorenog koda Mesa. Mesa je projekt implementacije standarda otvorenog koda za grafičke kartice, trenutno podržava većinu standarda definiranih od strane Khronos grupe, također sadrži mogućnost korištenja sa vlasničkim implementacijama poput ROCm (implementacija standarda OpenCL od strane AMD-a). Za računske operacije postoji više implementacija jezika Open Computing Language (OpenCL) ovisno o hardveru koji se koristi. Za AMD GPU-e se koriste vlasnički upravljački program ROCm ili prednjeg dijela upravljačkog programa otvorenog koda za OpenCL nazvanog Clover. OpenCL je standard otvorenog koda definiran od strane grupe Khronos, a svrha je omogućiti kompilaciju koda za paralelnu obradu na različitim uređajima sa implementiranim standardom.

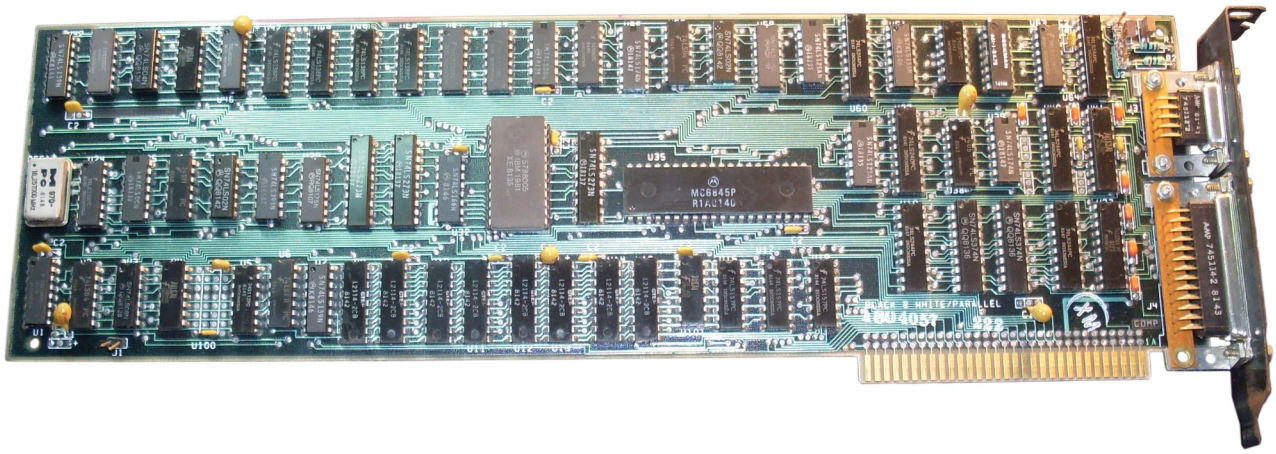
Clover je implementacija standarda OpenCL i dio otvorenog stoga za računanje na grafičkim procesorima, što znači da je cijela implementacija vidljiva sa mogućnosti izmjene, upravo to čini Clover dobrom početnom točkom za učenje i unaprijeđenje upravljačkih programa otvorenog koda na Linux operativnim sistemima. AMD osim Clovera podržava ROCm [5] implementaciju za OpenCL. ROCm je također otvorenog koda ali zbog dodatnih mogućnosti i implementacija uz OpenCL postavlja se kao vrlo kompleksan upravljački program namijenjen za korištenje u HPC-ovima. ROCm sadrži i brojne mogućnosti za korištenje sa algoritmima strojnog učenja poput Docker kontejnera, Python wheel-ova i Kubernets-a kako bi postavljanje HPC-a bili jednostavnije. Superračunalo AMD Frontier koristit će upravljački program ROCm i trebalo bi imati mogućnost izvođenja preko 1.5 exaflopa kalkulacija.

U ovom radu opisan je postupak popravljanja OpenCL implementacije Clover u svrhu unaprijeđenja upravljačkih programa otvorenog koda. Glavni cilj je pronalazak i popravljanje problema u Cloveru koristeći testove za OpenCL implementaciju “Piglit” i programa za molekularnu biologiju OpenMM.

2. Arhitektura AMD GPU-a

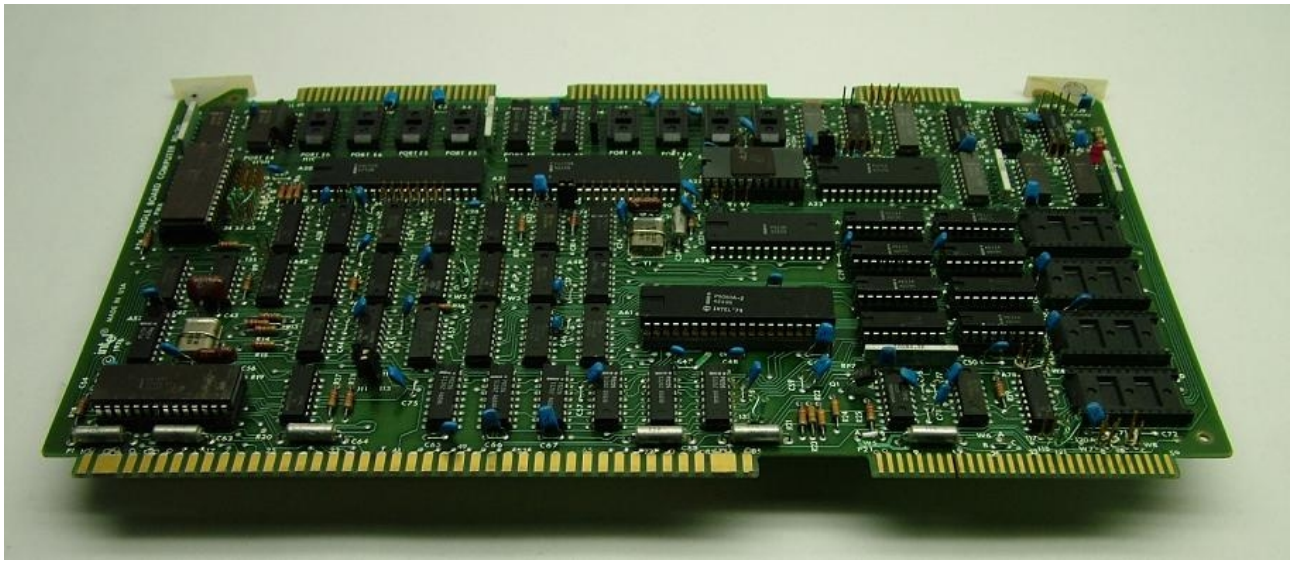
2.1. Povijest grafičkih procesora

Povijest GPU-a započine 1951. godine sa projektom Whirlwind I[6] (WWI) razvijanim od strane MIT (Massachusetts Institute of Technology) u suradnji sa ONR (Office of Naval Research) za potrebe simulacije leta. Ovo je bilo računalo posebne namjene i smatra se prvim 3D grafičkim sistemom. WWI je obrađivalo 20000 operacija po sekundi i sadržavalo registre duljine 16 bitova, a za potrebe točnijih izračuna bilo je moguće izvesti rutine koje bi koristile više registara za jedan zapis. Unatoč ovom računalu GPU-ovi kakve danas poznajemo povlače više paralela sa IBM-ovim adapterom za zaslone Monochrome Display Adapter[7] (MDA) predstavljenim 1981. godine. MDA je prva računalna komponenta napravljena specifično za prikaz videa na zaslon, a imao je mogućnost prikaza 80x25 znakova od kojih je svaki bio 9x14 pixela. MDA je imao memoriju od 4KB što je značilo da kompletan prikaz simbola na zaslonu stane u RAM memoriju na MDA čipu.



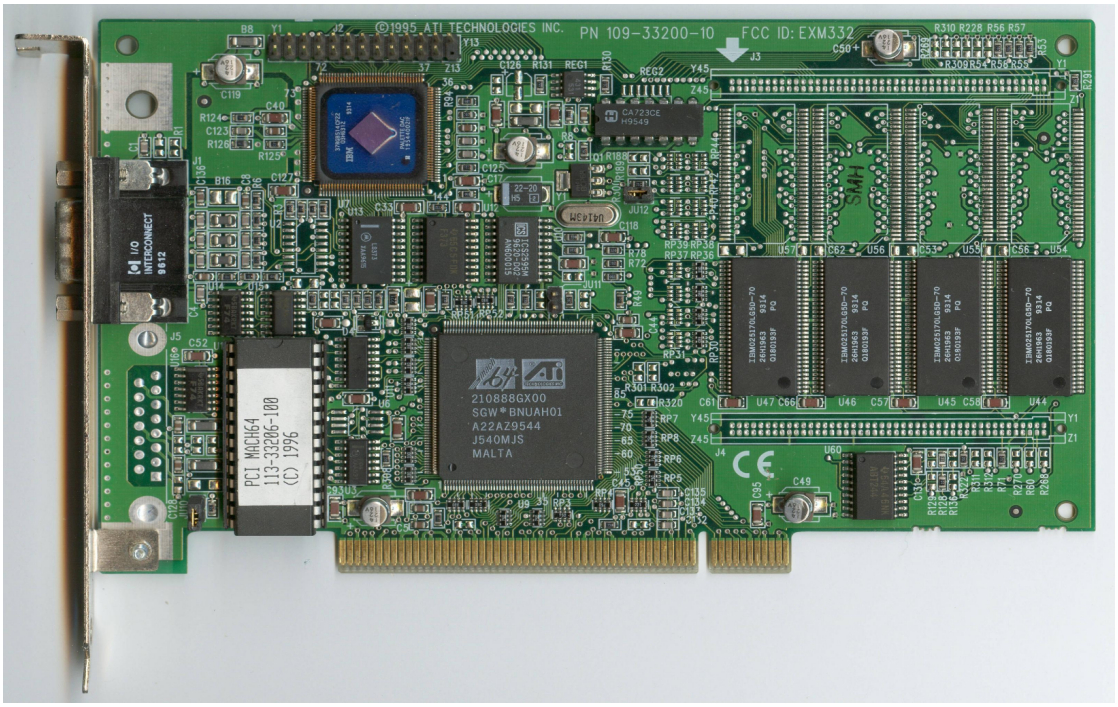
Slika 1: MDA adapter (IBM, 1981. godina)

Intel nedugo nakon IBM-a slijedi sa vlastitim rješenjem za prikaz videa na zaslon sa čipom ISBX 275 iz 1983. godine. ISBX 275[8] je sadržavao 32KB memorije na adapteru i imao mogućnost prikaza rezolucije 512x512 piksela u crno bijelom načinu rada ili 256x256 u načinu rada sa 8 boja.



Slika 2: iSBX adapter za sliku (Intel, 1983. godina)

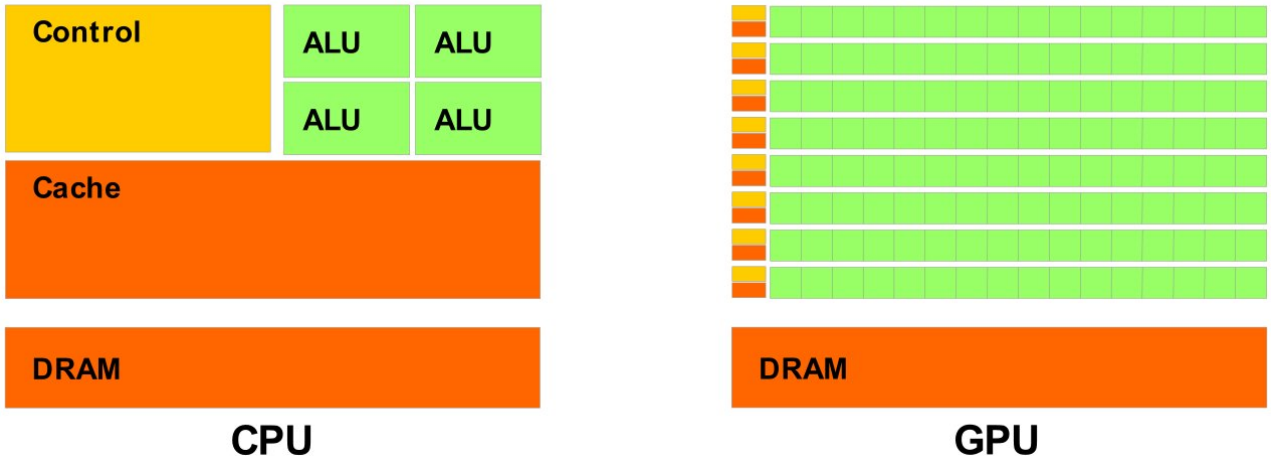
Godine 1985. na tržište dolazi tvrtka ATI(Array Technologies Inc.) koja je proizvodila grafičke kartice za tvrtke poput IBM i Commodore, dvije godine kasnije na tržište izdaju grafičke kartice VGA Wonder. Mach i Rage serije grafičkih kartica dovele su 2D i 3D akceleraciju grafičkih elemenata i dio su 3D revolucije. Grafičke kartice razvijane su primarno zbog prikaza 3D računalnih igara i bržeg prikaza GUI-a. Radeon je zadnja generacija prije nego je AMD kupio ATI. Nakon akvizicije započinje i era GPGPU-a sa novom generacijom zvanom TeraScale koja dovodi OpenCL podršku uz mnoge ostale novosti.



Slika 3: ATI Mach 64

2.2. Razlike između CPU-a i GPU-a

CPU(eng. Central Processing Unit) je elektronički čip koji izvršava zadane instrukcije a glavni zadatak mu je procesiranje aritmetičkih, logičkih, kontrolnih i ulazno/izlaznih podataka. CPU upravlja svim ostalim komponentama računala. GPU je specijalizirani čip napravljen za vrlo paralelnu obradu podataka. Glavne razlike između grafičkih procesora i centralne jedinice dolazi od osnovne funkcije za koju su napravljeni. Osnovni zadatak centralne jedinice je izvođenje različitih instrukcija na podatcima i kontrola ostalih komponenti, ovo se naziva paralelizam zadataka. Za razliku od centralne jedinice, grafički procesori razvijeni su za takozvani paralelizam podataka. Paralelizam podataka je izvođenje malog broja instrukcija, ali na različitim podatcima u vrlo paralelnom okruženju. Razlike u arhitekturi možemo vidjeti na slici 4.



Slika 4: Razlika između arhitekture CPU-a i GPU-a

Zbog osnovnih razlika u arhitekturi svaki pristup ima svoje prednosti i nedostatke. Funkcija CPU-a je da svi procesi mogu biti izvršeni sa niskom latencijom i podržava kompleksnu logiku. GPU koristi jednostavne instrukcije na mnogo podataka koristeći široki paralelizam.

2.3. Arhitekture AMD Radeon GPU-a

2.3.1. Arhitektura TeraScale

Prva GPGPU (General-Purpose Graphics Processing Unit) arhitektura od strane AMD koja je promijenila način na koji se GPU-kartice koriste. Prethodne arhitekture su bile specijalizirane samo za računalnu grafiku, dok se uvođenjem fleksibilnih shadera omogućava izvođenje paraleliziranih izračuna nekoliko puta brže nego na klasičnim CPU-ovima. AMD je uz to izdao implementaciju za

Open Compute Library(kraće, OpenCL) koja olakšava paralelno programiranje na heterogenim sistemima i time ubrzava matematičke i znanstvene kalkulacije.

Osnovna procesorska jedinica sastoji se od Stream procesora koji su slični Arithmetic and Logic Unit (kraće, ALU) procesorima, ali su specijalizirani za aritmetičke operacije. Stream procesori sa dijeljenom memorijom, jedinicama za teksture, jedinicom za prijenos podataka i L1 cache memorijom čine jedan Stream Processing Unit (kraće, SPU), a mnogo SPU-a čini jedan Single Instruction Multiple Data (kraće, SIMD) procesor.



Slika 5: Arhitektura TeraScale

Nova funkcionalnost fleksibilnih shader-a je bazirana na Very Long Instruction Word (kraće, VLIW),a to je način zadavanja hardverskih instrukcija u takvom obliku da se instrukcije izvršavaju paralelno. Ovaj način omogućava bolje iskorištenje hardverskih resursa, ali znatno podiže kompleksnost hardverske arhitekture. Prednosti VLIW instrukcija su korištenje pipelineova, višeprocesorsko izvršavanje neovisnih instrukcija i izvođenje većeg broja neovisnih operacija unutar jedne instrukcije [9]. VLIW instrukcije se grupiraju u wavefront, a on sadrži 64 VLIW threada koji se šalju na svaku SIMD jezgru, ona u naredna 4 ciklusa obrađuje podatke koji su došli u wavefrontu. Sve ove promjene dovele su do bolje iskorištenosti hardvera i napretka u paralelnom procesiranju na GPU-ima.

2.3.1.1. TeraScale 1

TeraScale 1 je arhitektura predstavljena 2007 godine i bila je prva arhitektura koja je koristila programabilne shader procesore. To je predstavljalo veliki napredak nad prošlim arhitekturama u obliku poboljšanih performansi zbog funkcionalnosti shader procesora koji su mogli biti zauzeti sa različitim tipovima zadataka. Najbolji primjer ove arhitekture je Radeon HD2900XT koja je imala 4 SIMD jezgre na 80 nanometarskom proizvodnom procesu.

2.3.1.2. TeraScale 2

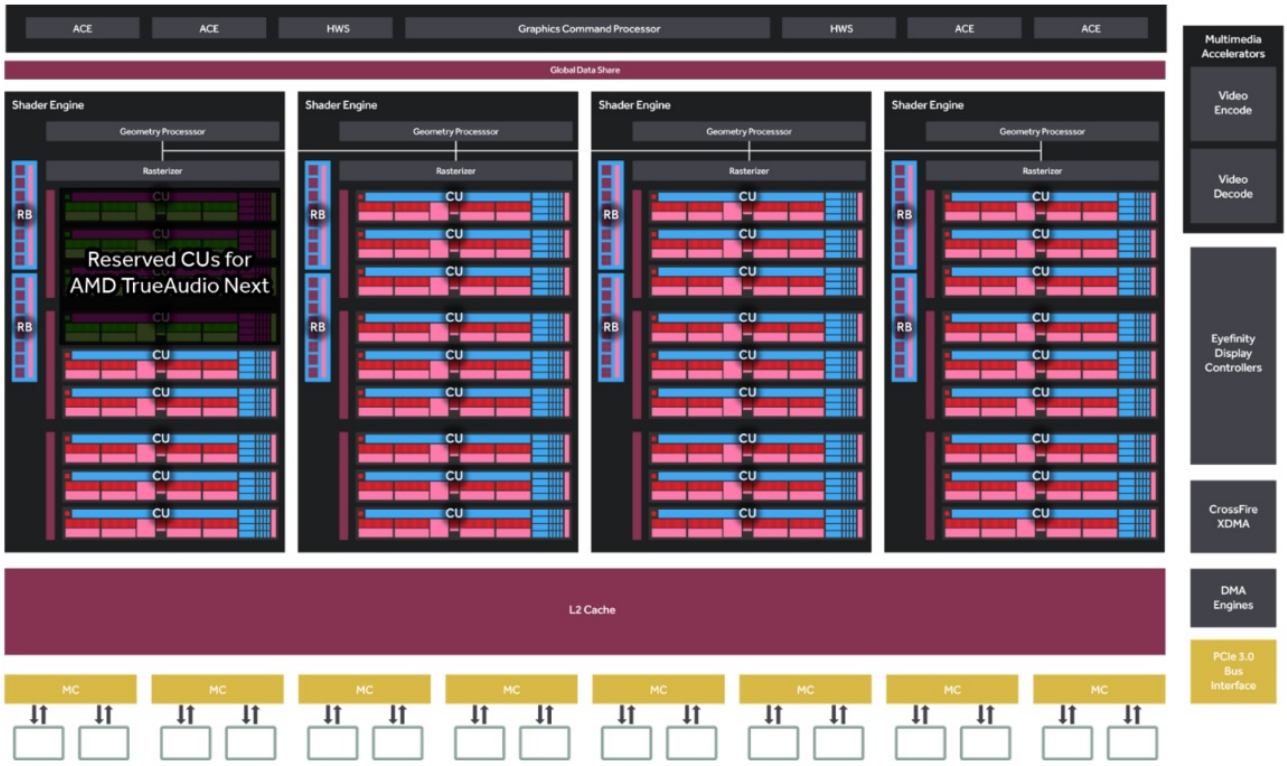
Nakon nekoliko iteracija arhitekture TeraScale 1, AMD je imao sve više problema sa neučinkovitosti planiranja izvršavanja instrukcija. Kako bi nadoknadili nedostatke sa HD 5000 serijom GPU-a razvijaju se optimizacije u obliku dodavanja L2 Cache memorije i povećanja broja SIMD procesora na dvadeset.

2.3.1.3. TeraScale 3

Arhitektura TeraScale 3 došla je uz malo povećanje broja SIMD procesora na 24, ali glavna optimizacija došla je u obliku smanjenja broja SP procesora u SPU sa 5 na 4. Ova promjena dovodi do bolje prosječne iskorištenosti SPU-ova prilikom korištenja wavefrontov-a TeraScale, ovdje dolazi do limita kada novi programi zahtijevaju kompleksne shadere i neučinkovito planiranje instrukcija u VLIW arhitekturi dovode AMD do novog pristupa GCN(Graphics Core Next) arhitekturom.

2.3.2. Arhitektura GCN (Graphics Core Next)

GCN je bila potpuno nova arhitektura, dizajnirana oko drugačijeg seta instrukcija kako bi se izbjegli problemi sa VLIW instrukcijskim setom. GCN je pomaknuo osnovni element jedan sloj više od TeraScale arhitekture, time je SPU postao osnovni blok, a 16 SPU-ova gradi jednu SIMD jezgru. Četiri SIMD sačinjavaju jedan CU (Compute Unit). Nekoliko CU jezgri čini jedan grafički čip prikazano na slici 6, ovaj pristup znači da je GCN potpuna SIMD arhitektura što donosi brojne prednosti poput znatno lakšeg planiranja izvođenja programa i kompilacije. Planiranje izvođenja se odvija dinamički tijekom run-time-a na hardverskoj strani. Svaki CU također ima i Scalar ALU kako matematičke operacije skaliranja vektora podataka ne bi usporile izvođenje ostalih operacija. Dodavanjem Scalar ALU-a u CU broj ciklusa za takve operacije smanjen je sa 44 ciklusa u TeraScale 3 arhitekturi na 1 operaciju u GCN arhitekturi. Ove prednosti dovode do poboljšanja performansi u računski zahtjevnim operacijama kao i u računalnim igrama.



Slika 6: AMD Radeon RX 480, primjer arhitekture GCN

2.3.2.1. Arhitektura GCN 1: Southern Islands

Prva generacija GCN arhitekture potpuno se odmaknula od VLIW instrukcija i zamijenila ih sa SIMD arhitekturom. Nadalje, GCN 1 je prva imala ACE (Asynchronous Compute Engine) napravljene specifično za ubrzanje računskih operacija na GPU-ima. ACE omogućuje nezavisno rezerviranje CU-a kako bi se omogućilo da GPU potpuno različite i neovisne izračune na CU-ima.

2.3.2.2. Arhitektura GCN 2: Sea Islands

Druga generacija povećava broj ACE-a sa 2 na 8, koristi bržu memoriju, povećava broj CU-a, a donosi novi način povezivanja više GPU-a koristeći XDMA Crossfire Bridge, koji kontrolira tok podataka preko PCI Express-a zbog čega hardverski most između GPU-a više nije potreban.

2.3.2.3. Arhitektura GCN 3: Volcanic Islands i Pirate Islands

Donosi dvije nove značajke u obliku rezerviranja resursa za poslove koji su višeg prioriteta (GPU preemtion) i podrške za decimalnu polu preciznost (FP16 or Half-precision floating point). Rezerviranje resursa omogućuje izvođenje više različitih izračuna sa mogućnosti designiranja

glavnog procesa koji prekida ostale u izvršenju kalkulacija kako bi se sam izvršio. Decimalna polu preciznost smanjuje korištenje memorije u slučaju kad za izračune nije potrebna uobičajena preciznost od 32 bita.

2.3.2.4. Arhitektura GCN 4: Artic Islands ili Polaris

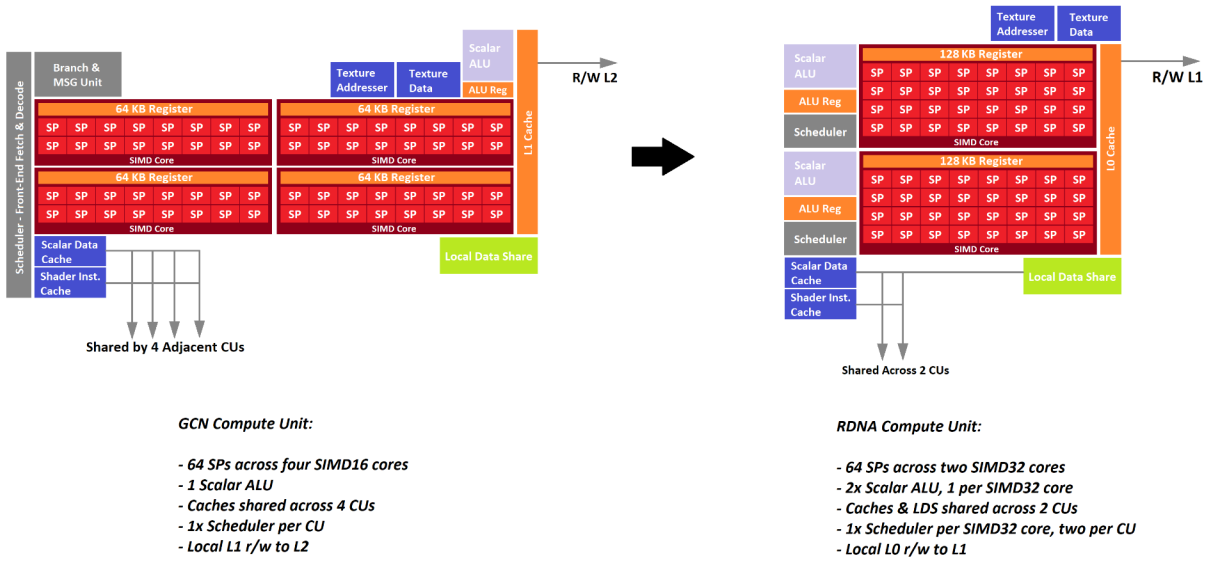
Najveći fokus generacije bio je usmjeren na poboljšanje efikasnosti GPU-a, kako bi postigli bolje rezultate i veći broja kalkulacija koristeći manje snage. Polaris arhitektura donosi brojne prednosti kao što su povećanje buffera instrukcija, te mogućnost dohvaćanja budućih instrukcija. GPU pokušava špekulativno dohvatiti moguće sljedeće instrukcije na osnovu trenutnog stanja podataka. U slučaju dobrog pogotka izvršenje je prihvaćeno i sam proces se ubrzava zbog toga što GPU nije čekao na dohvaćanje instrukcija prije početka izvođenja, no u slučaju promašaja gubi se efikasnost jer se izračun odbacuje.

2.3.2.5. Arhitektura GCN 5: Vega

GCN 3 je uveo decimalnu polu preciznost, ali je tek Vega iskoristila mogućnost do punog potencijala uvođenjem Rapid Packed Math (kraće, RPM), ova značajka omogućuje da svaki Streaming Processor (kraće, SP) može izvršavati dvije FP16 operacije umjesto jedne FP32 operacije. Postupak omogućuje znatno bolje performanse u FP16 izračunima za razliku od prošlih generacija koje su unatoč mogućnosti izvođenja FP16 operacija koristile cijeli SP za izvođenje jedne FP16 operacije, što je značilo da izvođenje FP16 i FP32 operacija traje jednako dugo.

2.3.3. Arhitektura RDNA: Navi

Arhitektura RDNA donosi brojne prednosti u odnosu na GCN arhitekturu. Najveća promjena je dupliciranje osnovnog elementa SP-a unutar SIMD jezgri u odnosu na GCN. RDNA tako ima 32 SP-a po SIMD jezgri i registar od 128 KB. Razlika između GCN i RDNA CU-a prikazano je na slici 7.

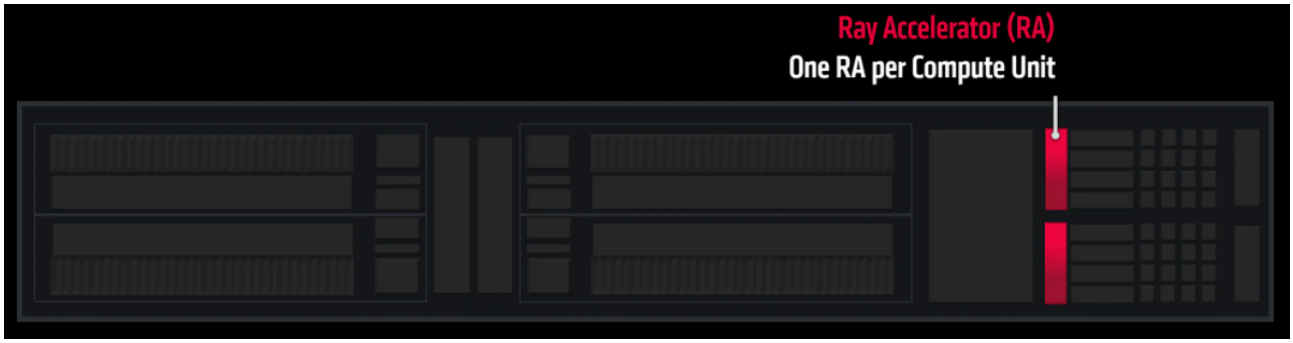


Slika 7: Razlika između arhitektura GCN i RDNA

Osim promjene u broju SP-a po SIMD jezgri mijenja se način dijeljenja memorije, broja schedulera po CU jezgri i veličina wavefronta na 32 elementa po wavefrontu što zajedno dovodi do toga da se wavefront izvršava u jednom ciklusu umjesto u 4 ciklusa. Sve promjene zajedno dovode do istih performansi kao arhitektura GCN 5 Vega, ali uz smanjenje broja SP-a za 33% i smanjenje potrošnje snage za 25% (usporedba AMD Radeon VII i 5700XT grafičkih kartica), čime se postiže bolja efikasnost na istom proizvodnom procesu (7nm).

2.3.4. Arhitektura RDNA2: Big Navi

RDNA 2 je trenutna AMD arhitektura koja će osim za osobna računala i HPC-ove biti korištena u novoj generaciji igraćih konzola. Nad prošlom generacijom donosi značajne promjene u broju CU-a kao i u efikasnosti. Novi GPU-i imat će do 80 CU-a i jedna od novosti je što će svaki CU imati specijalizirani Ray Accelerator (kraće, RA), prikazan na slici 8. Mijenja se struktura memorije unutar GPU-a dodavanjem 128MB cache memorije po prvi put unutar GPU-a kako bi se povećala propusnost memorije. Poboljšanja unutar arhitekture dovode do povećanja efikasnosti od 54% u odnosu na RDNA [10]. RDNA 2 je prva arhitektura koja omogućuje CPU-u direktni pristup radnoj memoriji GPU-a što bi se trebalo u budućnosti znatno odraziti na performanse kada se programi i igre započnu razvijati sa ovom funkcionalnosti.



Slika 8: Ubrzavač svijetlosnih zraka unutar CU-a (eng. Ray Accelerator)

2.4. Moderan GPU kao skup blokova intelektualnog vlasništva

Moderni GPU-i su građeni od više specijaliziranih blokova intelektualnog vlasništva. Blok intelektualnog vlasništva je integrirani čip, a njegov dizajn je vlasništvo tvrtke koja ga dizajnirala. Svaki GPU može imati više istih blokova, te se dizajni blokova mogu dijeliti sa trećom stranom ili biti zaštićeni od strane tvrtke. Blokovi intelektualnog vlasništva na GPU-ima služe analognoj svrsi kao biblioteke u računalnom programiranju. To su odvojene modularne komponente sa jasno definiranom logikom i sučeljem. Blokovi mogu biti od takozvanih soft i hard jezgri. Soft jezgre jedom dizajnirane od strane tvrtke poput AMD-a ili Nvidije predaju se tvrtkama koje se bave proizvodnjom čipova u digitalnom obliku sa jasno definiranom logikom čipa napisanom u sloju apstrakcije Register-Transfer Level (kraće, Register-Transfer Level). Potom se definiranu logiku prevodi u Hardware Definition Language (kraće, HDL) npr. VDHL. Čipovi definirani na ovaj način šalju se jednom od proizvođača čipova te se prilagođavaju prema mogućem proizvodnom procesu. Svaki od proizvođača čipova ima predefinirane hard jezgre koje je moguće ugraditi u čip, a njih nije jednostavno prilagoditi. Izmjena hard jezgri bi značila mijenjanje proizvodnog procesa zbog čega je učestalo korištenje već dobro definiranih i testiranih dizajna zbog brže proizvodnje i manjeg rizika. Navedeni proces zove se IP hardening, a hard IP je blok intelektualnog vlasništva testiran u čipu i aplikaciji u stvarnom svijetu. U nastavku su prikazani blokovi intelektualnog vlasništva za AMD GPU (RX 480).

```
[    0.815247] [drm] add ip block number 0 <soc15_common>
[    0.815248] [drm] add ip block number 1 <gmc_v9_0>
[    0.815249] [drm] add ip block number 2 <vega10_ih>
[    0.815250] [drm] add ip block number 3 <psp>
[    0.815252] [drm] add ip block number 4 <gfx_v9_0>
[    0.815253] [drm] add ip block number 5 <sdma_v4_0>
[    0.815254] [drm] add ip block number 6 <powerplay>
[    0.815255] [drm] add ip block number 7 <dm>
[    0.815256] [drm] add ip block number 8 <uvd_v7_0>
[    0.815258] [drm] add ip block number 9 <vce_v4_0>
```

Slika 9: Izlaz naredbe dmesg za IP blokove na AMD Radeon Rx 480 GPU

Prvi blok vlasništva prikazan na slici 9 je "soc15_common" označava System on a Chip, a to je hardverski čip zadužen za inicijalizaciju, upravljanje GPU-om i ostalim blokovima na GPU-u ovisno o naredbi koja dolazi iz kernela. Upravlja registry-ima poput pokazivača na GPU, pokazivača na registrije unutar GPU-a i pokazivača na veličinu polja registra. Osim toga zadužen je za komunikaciju između upravljačkog stoga i GPU-hardvera, te točno prijavljivanje hardvera koji se nalazi na GPU-u. Primjer inicijalizacije su različite vrijednosti poruka za korištenje PCI-a, ukoliko se radi o APU(Accelerated Processing Unit) poruka je 0 što upravljačkom programu govori da se PCI ne koristi u komunikaciji.

Global Memory Controller (GMC) čip služi za kontrolu L2 cache memorije i raspoređivanje memorijskih lokacija prema CU-ovima na GPU.

Platform Security Processor (PSP) zaslužan je za stvaranje, praćenje i održavanje sigurnog okruženja prilikom izvođenja koda. Sastoji se od ARM procesorske jezgre koja ima pristup glavnoj jezgri i omogućuje zaustavljanje izvođenja sumnjivog koda i događaja na GPU-ima i CPU-ima.

GFX blok glavni je blok GPU-a i unutar njega nalaze se SIMD jezgre koje omogućavaju računalne kalkulacije i renderanje slike.

SDMA blok zadužen je za direktnu komunikaciju sa memorijom GPU-a, a služi za alokaciju, de alokaciju i ostale operacije koje zahtijevaju direktan pristup GPU memoriji. Osim navedenih mogućnosti blok omogućava i asinkroni prijenos podataka između glavne radne memorije i GPU memorije što u kalkulacijama koje zahtijevaju mnogo memorije može dovesti do ubrzanja obrade. Važno je napomenuti kako SDMA ima vlastiti trošak vremena koje je potrebno za usklađivanje memorije, prema tome može dovesti i do sporije obrade u specifičnim situacijama.

Blok PowerPlay zadužen je za kontrolu potrošnje struje unutar GPU-a i povećanja efikasnosti

izvođenja računskih operacija. Čip pokušava smanjiti brzinu radnog takta GPU-a kako bi se smanjila potrošnja u trenucima kad GPU nije u aktivnom stanju ili nije u potpunosti iskorišten bez negativnog utjecaja na performanse.

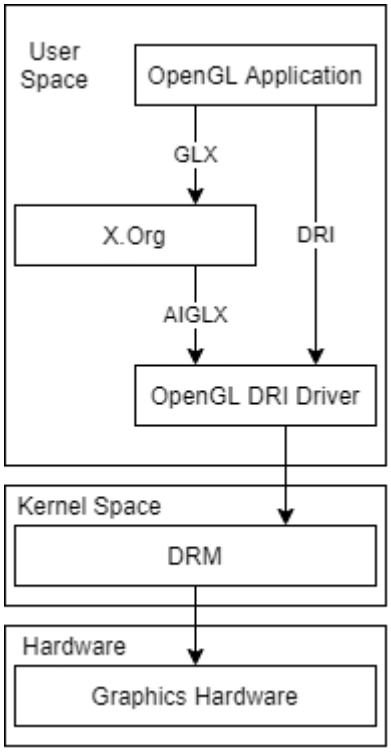
Display Manager (DM) služi za postavljanje inicijalnih vrijednosti zaslona kao i dohvaćanje vrijednosti koje zaslon pruža. Osim inicijalizacije, zaslužan je za kontrolu svih priključenih zaslona i pravilnu komunikaciju između GPU-a i zaslona.

UVD vlasnički blok zadužen je za dekodiranje videa. Suvremeni GPU-i koriste video dekodere kao specijalizirane blokove kako dekodiranje videa ne bi koristilo glavni dio GPU-a koji ostaje slobodan za obradu podataka.

VCE blok kodira video signal, odvojen je od glavnog dijela GPU-a zbog istog razloga kao i UVD blok. Korištenjem ovih blokova povećava se efikasnost i brzina GPU-a i kao i mogućnost simultane obrade videa i ostalih podataka.

3. Upravljački stog na Linux sustavu

Upravljački stog se sastoji od X.Org-a, OpenGL DRI (Direct Rendering Infrastructure) upravljačkog programa, DRM (Direct Rendering Manager) koji radi na kernel nivou i grafičkog hardvera.



Slika 10: Upravljački stog za GPU na Linux sustavu

OpenGL aplikacija za prikaz 3D objekata na zaslonu poziva X.Org koristeći OpenGL Extension to the X Window System (GLX) API te prosljeđuje podatke za prikaz prema OpenGL DRI upravljačkom programu. Ukoliko se aplikacija izvršava na istom računalu i ima kompatibilne upravljačke programe X.Org moguće je zaobići direktnim API pozivima prema OpenGL upravljačkom programu. Primjer upravljačkog programa koji omogućava ovakvo izvođenje je Mesa, ali i upravljački programi zatvorenog koda za grafičke procesore od strane proizvođača. X.Org je projekt implementacije X Window sistema, X pruža osnovne mogućnosti za prikaz grafike i interakciju sa ulaznim uređajima.

3.1. Mesa

Mesa je projekt koji započinje kao implementacija OpenGL-a od strane Brian Paula, a danas

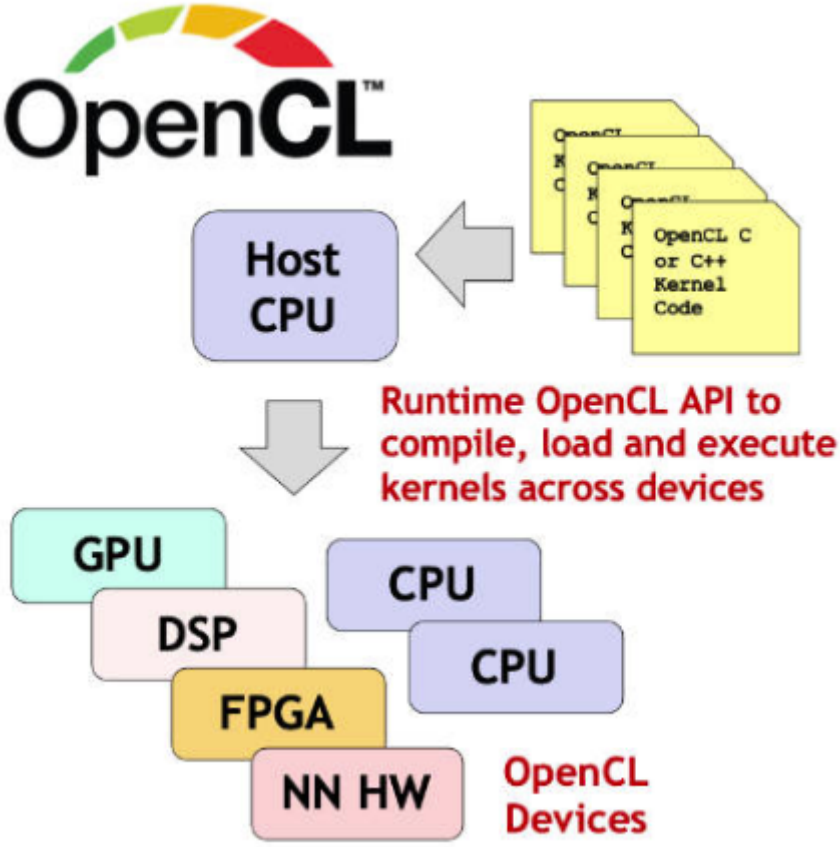
ujedinjuje brojne implementacije upravljačkih programa i Application Programming Interface-a (kraće, API) u jedan projekt otvorenog koda. Mesa je primarno razvijena i korištena na Linux sistemima, ali podržava Windows i Unix sisteme. Trenutno podržava hardverske upravljačke programe za uređaje od strane Intel, AMD, Nvidia, ARM, Qualcom i Broadcom tvrtki. Uz hardverske upravljačke programe podržava i softverske za LLVM, VMWare swr, virgl i swrast. Mesa je usko povezana sa drugim projektima otvorenog koda poput Direct Rendering Infrastructure (kraće, DRI) i X.org kako bi zajedno pružili OpenGL podršku na Linux, FreeBSD i ostalim sustavima. Primarno je fokus bio na OpenGL podršci, ali danas sadrži i implementacije API-a kao što su Vulkan i OpenCL koji omogućuju programiranje aplikacija za izvođenje računskih operacija na GPU-ovima.

3.1.1. Gallium3D

Gallium3D je novi pristup pisanju upravljačkih programa, a dizajniran je kako bi bio portabilan na sva grafička sučelja i sve operacijske sustave. Glavni ciljevi su smanjenje upravljačkih programa, apstrakcija arhitekture i podrška za različite grafičke API-e. Gallium3D pruža tri glavne apstrakcije, *pipe_context* je apstraktna osnovna klasa za postavljanje renderanja, shadera i vertex shader-a. *p_state* definira status izvršavanja, teksture i raspored vrhova 3D objekta. _pipe_screen_ služi za kreiranje tekstura, upite prema hardveru te kreaciju i mapiranje/demapiranje buffera. Koristeći ovu apstrakciju tradicionalne funkcije OpenGL-a su implementirane koristeći shadere, a instrukcije koje nisu direktno podržane na GPU pokušavaju se prevesti u izvedive instrukcije.

3.1.2. Clover

Clover je dio Gallium3D programskog sučelja koji omogućuje OpenCL GPU podršku prema aplikacijama napisanima u OpenCL jeziku. OpenCL je jezik otvorenog koda koji omogućuje paralelno programiranje na različitim uređajima koji podržavaju OpenCL. Ilustraciju korištenja jezika OpenCL sa različitim hardverom moguće je vidjeti u nastavku.



Slika 11: Primjer heterogenog paralelnog procesiranja omogućenog OpenCL-om

Clover implementira Khronos OpenCL [11] standard i trenutno je u verziji OpenCL 1.2 i napreduje prema verziji 3.0.

3.2. LLVM

LLVM je projekt otvorenog koda napravljen kao kolekcija alata, biblioteka i zaglavlja potrebnih za kompilaciju koda u strojni kod. LLVM je pozadina koji sadrži mnogo različitih programa prevoditelja za različite jezike i hardvere. Clang je prednji dio koji služi za kompilaciju C obitelji jezika u oblik programa za izvođenje na hardveru. Clang prevodi jezike u LLVM bit koda LLVM bit kod prevodi u objekte koje hardvere zna koristiti. Clang podržava veći broj jezika poput OpenCL, C++, CUDA, C [12], a osim kompilacije programa podržava i dijagnostiku programa.

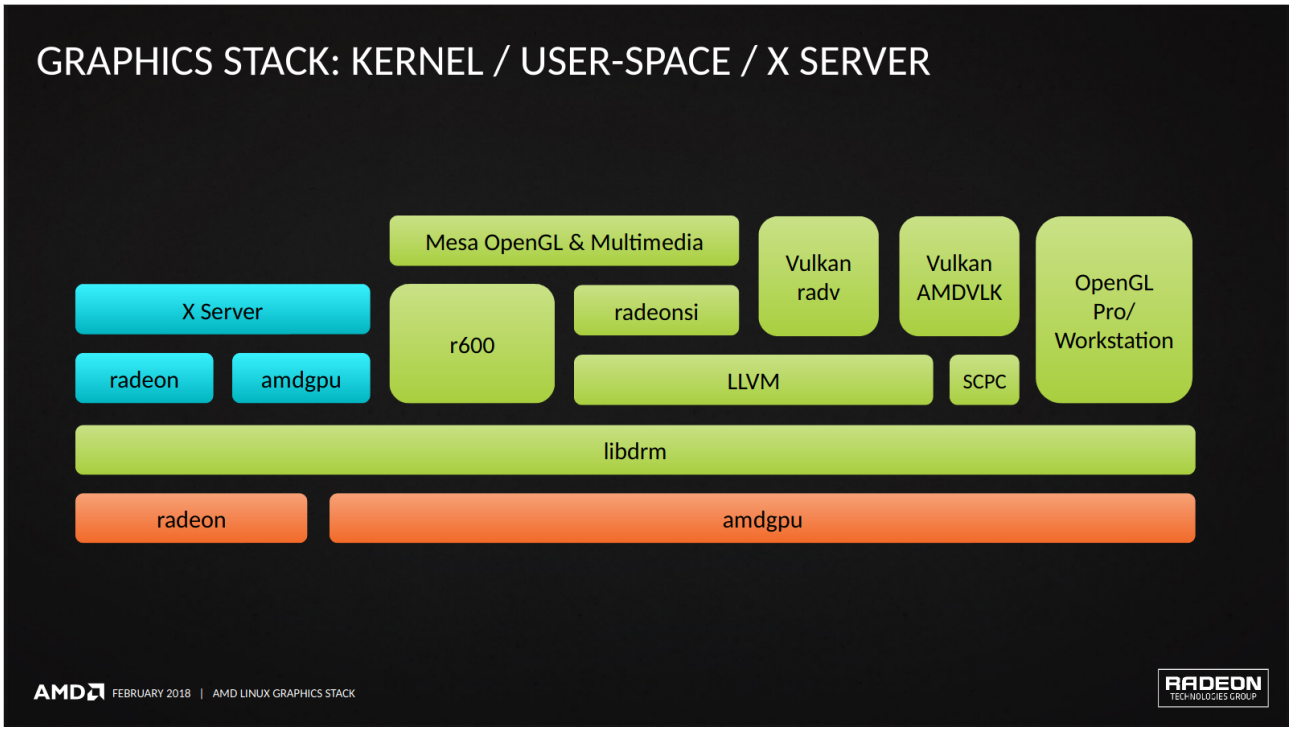
3.3. Kernel

Linux DRM sloj omogućuje interakciju grafičkih upravljačkih programa sa hardverom, uobičajeno

koristeći programabilne pipeline prilagođenje 3D aplikacijama, ali i izvođenju računskih kalkulacija na GPU. GPU driveri koriste DRM funkcije za kontrolu memorije, rad sa prekidima izvođenja i olakšavaju DMA pristup. U prvim verzijama DRM modesetting je bio postavljan iz korisničkog radnog prostora što nije bilo idealno zbog toga što je samo jedna aplikacija mogla izvršavati operacije u danom trenutku. U slučaju da dvije aplikacije pokušavaju prikazivati u isto vrijeme postavljala bi se utrka između modesettinga i prikazivanja. Kernel Mode Setting (KMS) dolazi kao rješenje problema konstantne potrebe za komunikacijom između korisničkog radnog prostora i kernela, te vraća kontrolu kernelu. KMS postavlja inicijalizaciju GPU-a i osnovnu konfiguraciju poput veličine frame buffera i funkcija mode settinga. Kernel space upravljački programi za AMD GPU su radeon (podržava starije GPU-e) i amdgpu (noviji driver koji podržava nove mogućnosti).

3.4. Pregled stoga upravljačkih programa za Radeon GCN/RDNA

Stog upravljačkih programa za AMD GPU-e sastoji se od kernela, DRM-a u, Mesa OpenGL, OpenCL (RadeonSI, i vulkan drivera (radv). Program prevoditelj LLVM u backendu, Clang kao prednji dio programa prevoditelja, OpenCL i libclc za implementaciju osnovnih matematičkih funkcija za OpenCL. Na slici 12 prikazan je pregled čitavog stoga za AMD GPU.



Slika 12: Stog upravljačkih programa za AMD na Linux-u

Dio stoga unutar kernela sastoji se od dva upravljačka programa, radeon i amdgpu. Radeon

upravljački program je nasljedni program namijenjen korištenju sa GPU arhitekturama prije GCN-a. Amdgpu je suvremeni dio stoga koji je trenutno u upotrebi, te podržava generacije GCN i RDNA. Kernel dio programa zaslužan je za inicijalizaciju GPU-a prilikom pokretanja računala, dohvaćanje podataka o zaslonima i postavljanje zaslona. Kernel upravljački programi su najniži dio apstrakcije i kao takvi brinu o komunikaciji sa GPU-om u obliku pisanja u registre, operacija sa memorijom GPU-a i pravima pristupa.

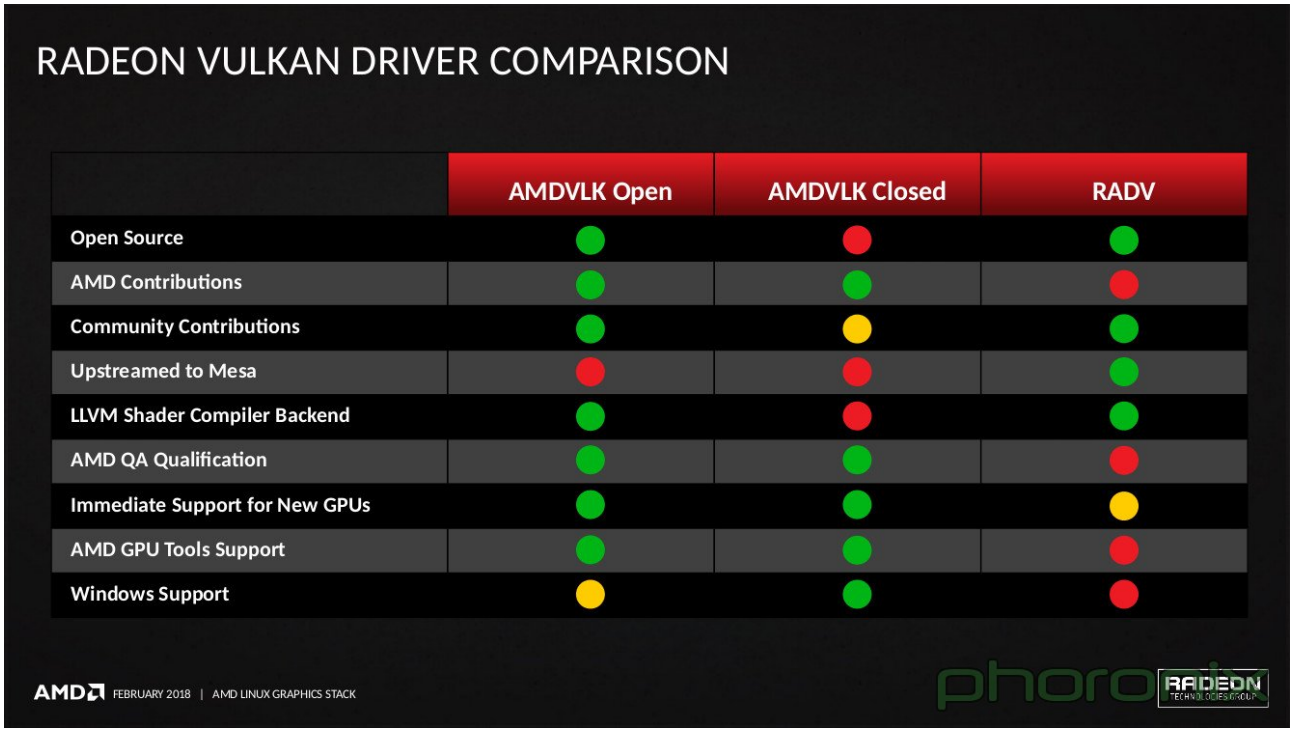
Libdrm je dio upravljačkog stoga u korisničkom prostoru koji komunicira sa kernelom, njegovo sučelje omogućava istovremenu komunikaciju većem broju aplikacija sa GPU-om. Pruženo sučelje je u velikoj mjeri ovisno o GPU-u za koje je napisano i kako bi aplikacija znala koristiti GPU mora biti napisana ili prevedena u kompatibilan oblik za svaki od GPU-a na kojem se treba izvoditi. Zbog velikog broja različitih GPU-a i aplikacija navedeni pristup nije praktično izvediv što dovodi do različitih upravljačkih programa koji apstrahiraju dio komunikacije sa libdrm-om i kernelom. U AMD stogu to su LLVM, RadeonSI(dio Gallium3D programa), radv, r600 i vlasničke implementacije SCPC, OpenGL Pro i AMDVLK.

R600 upravljački program ima mogućnost pokretanja OpenGL aplikacija na starijim AMD GPU-ima koji su izašli prije GCN arhitekture. R600 je robusni upravljački program koji se brine o svemu od početnog poziva iz aplikacije do komunikacije sa libdrm-om.

LLVM je pozadinski program prevoditelj koji pretvara dobiveni kod u binarne datoteke koje libdrm i amdgpu znaju koristiti i izvršiti. Za korištenje sa OpenCL jezikom najbitniji je Clang prednji program prevoditelj koji je dio LLVM-a. Clang pretvara programe pisane u OpenCL-u u računalni kod napravljen za izvođenje na GPU-ima, a zbog toga što je OpenCL standard kojeg prate svi proizvođači i svi upravljački programi, Clang zajedno sa LLVM-om može prevesti kod u verziju prikladnu za pojedini hardver.

RadeonSI je dio Gallium3D upravljačkog programa, napisan za AMD GPU-e te služi kao apstrakcija za komuniciranje aplikacija sa GPU-om. Gallium3D sadrži zajedničke dijelove koda od različitih upravljačkih programa kako bi se smanjila razlika u implementaciji, te kako bi održavanje i razvijanje koda bilo efikasnije.

Radv kao otvorena implementacija vulkan API-a dio je Mesa stoga. AMD ima i vlasničku implementaciju standarda zvanu AMDVLK a razlike između njih su vidljive na slici 13.



Slika 13: Razlike u implementacijama standarda Vulkan

Vulkan API pruža mogućnost pristupa GPU-ima sa velikom efikasnosti i podržava GPU-e različitih platformi od mobitela do računala. Specifikacija je u konstantnom razvoju i prati potrebu novih mogućnosti u grafičkom programiranju i računalnoj obradi podataka.

4. Piglit

Piglit je kolekcija testova za OpenGL i OpenCL implementacije. Cilj je ubrzati testiranje i pronalazak problema u upravljačkim programima. Većina Piglit testova preuzeta je od Glean i Mesa projekata kako bi se stvorila zajednička biblioteka za testiranje. Danas je Piglit postao dio Mesa projekta i dalje je u aktivnom razvoju. OpenCL programski okvir unutar Piglita sadržava tri vrste testova:

- API testovi – testovi namijenjeni testiranju API poziva i funkcija
- Testovi programa – Testovi kompilacije OpenCL programa i izvršavanja u kernelu
- Ostali testovi – Ostali testovi koji ne pripadaju već spomenutim kategorijama

U nastavku je prikazan primjer API testa za kreaciju 2D slike.

```
#define CL_USE_DEPRECATED_OPENCL_1_1_APIS
#include "piglit-framework-cl-api.h"

PIGLIT_CL_API_TEST_CONFIG_BEGIN
        config.name = "clCreateImage";
        config.version_min = 10;
        config.run_per_platform = true;
        config.create_context = true;

PIGLIT_CL_API_TEST_CONFIG_END
static void
no_image_check_invalid(
        cl_int errcode_ret,
        enum piglit_result *result,
        const char *name)
{
        if (!piglit_cl_check_error(errcode_ret, CL_INVALID_OPERATION)) {
                fprintf(stderr, "%s: CL_INVALID_OPERATION expected when no "
                                "devices support images.\n", name);
                piglit_merge_result(result, PIGLIT_FAIL);
        }
}

static enum piglit_result
```

```
no_image_tests(const struct piglit_cl_api_test_env* env)
{
        enum piglit_result result = PIGLIT_PASS;
        cl_context cl_ctx = env->context->cl_ctx;
        cl_mem_flags flags = CL_MEM_READ_ONLY;
        cl_image_format image_format;
        size_t image_width = 1;
        size_t image_height = 1;
        size_t image_depth = 2;
        size_t image_row_pitch = 0;
        size_t image_slice_pitch = 0;
        void *host_ptr = NULL;
        cl_int errcode_ret;
        image_format.image_channel_order = CL_RGBA;
        image_format.image_channel_data_type = CL_FLOAT;

        clCreateImage2D(cl_ctx, flags, &image_format, image_width,
                        image_height, image_row_pitch, host_ptr,
                        &errcode_ret);

        no_image_check_invalid(errcode_ret, &result, "clCreateImage2D");

        clCreateImage3D(cl_ctx, flags, &image_format, image_width,
                        image_height, image_depth, image_row_pitch,
                        image_slice_pitch, host_ptr, &errcode_ret);

        no_image_check_invalid(errcode_ret, &result, "clCreateImage3D");
        return result;
}
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)
{
        if (!piglit_cl_get_context_image_support(env->context)) {
                return no_image_tests(env);
        } else {
                return PIGLIT_PASS;
        }
}
```

Test inicijalizira početne podatke i zatim poziva OpenCL funkcije clCreateImage2D i clCreateImage3D sa svim parametrima objekta. U slučaju da funkcije nisu podržane na danoj implementaciji OpenCL-a testova, dobiva se obavijest o tome da dana funkcija nije podržana. U prikazanom kodu je specifikacija za funkciju clCreateImage2D kako je definirana prema OpenCL standardu:

```
cl_mem clCreateImage2D(
    cl_context context,
    cl_mem_flags flags,
    const cl_image_format* image_format,
    size_t image_width,
    size_t image_height,
    size_t image_row_pitch,
    void* host_ptr,
    cl_int* errcode_ret);
```

clCreateImage2D objekt prema OpenCL specifikaciji[13]

4.1. Pokretanje OpenCL testova

Pokretanje OpenCL testova sastoji se od podizanja Piglita, za njegovo podizanje potrebno ga je preuzeti sa GitLab-a [14], te pokrenuti sljedeće naredbe iz Piglit direktorija. Također je potrebno preuzeti sve pakete potrebne za podizanje Piglita.

```
sudo apt-get install cmake g++ mesa-common-dev libgl1-mesa-dev \
     python-numpy python-mako freeglut3-dev x11proto-gl-dev \
     libxrender-dev libwaffle-dev
cmake .
make
```

Nakon toga Piglit izvršna datoteka trebala bi biti u direktoriju i kako bi provjerili da li je Piglit podizanje dobro prošlo pokreće se sljedeća naredbu:

```
./piglit run sanity results/sanity
```

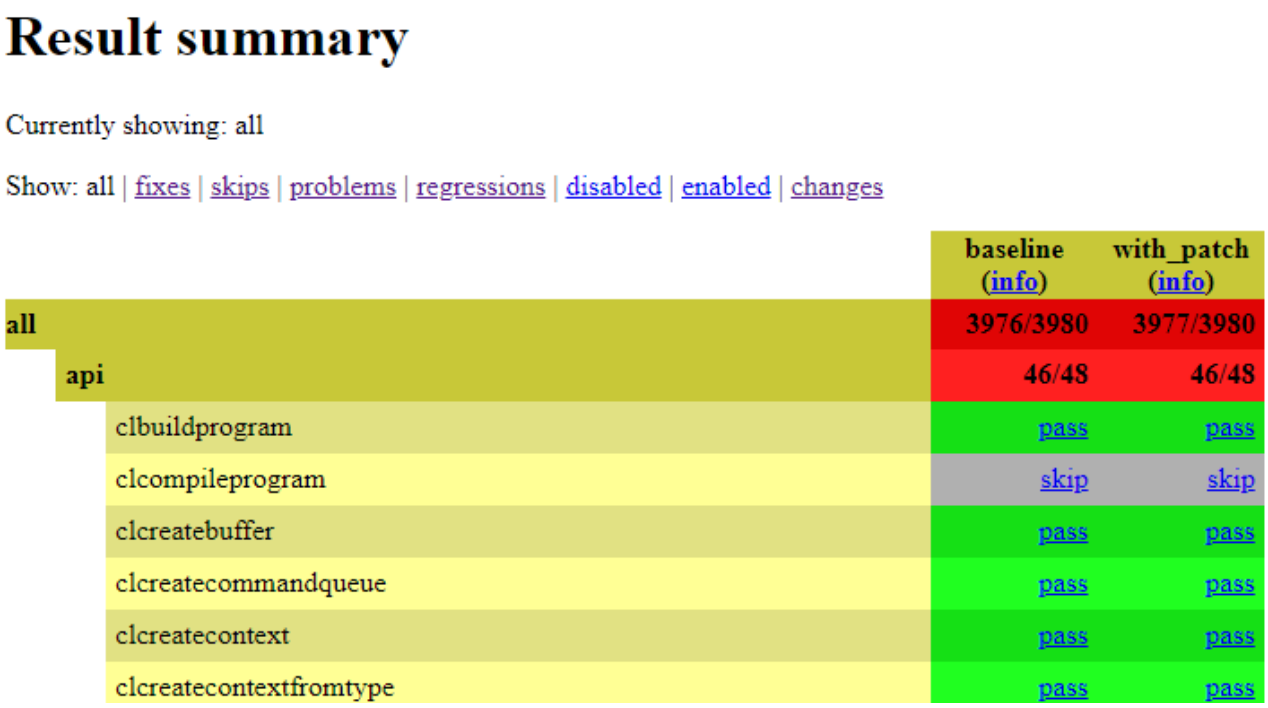
Pokretanjem Piglita na ovaj način validira se instalacija i rezultati pokretanja zapisuju se u results/sanity direktorij. Rezultati su u ovom trenutku zapisani kao zapakirana json datoteka. Kako bi se rezultati lakše interpretirali potrebno ih je zapisati u oblik čitljiv ljudima. Piglit sadrži alat za pretvorbu oblika zapisa i koristeći njega moguće je dobiti izvješće o testovima u obliku HTML stranice.

```
./piglit summary html summary/sanity results/sanity
```

Na sličan način pokreću se testovi za OpenCL implementaciju instaliranih upravljačkih programa. Za dobivanje izvješća se koriste sljedeće naredbe.

```
# Pokretanje OpenCL testova
./piglit run cl results/cl

# Zapisivanje rezultata u oblik prigodan za ljude
./piglit summary html summary/cl results/cl
```



Result summary

Currently showing: all

Show: all | fixes | skips | problems | regressions | disabled | enabled | changes

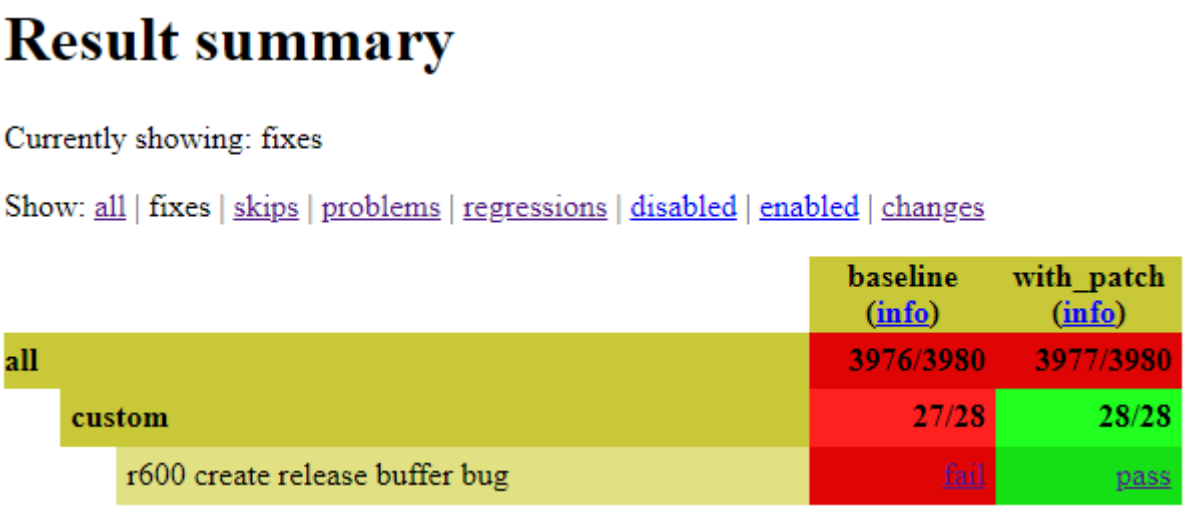
	baseline (info)	with_patch (info)
all	3976/3980	3977/3980
api	46/48	46/48
clbuildprogram	pass	pass
clcompileprogram	skip	skip
clcreatebuffer	pass	pass
clcreatecommandqueue	pass	pass
clcreatecontext	pass	pass
clcreatecontextfromtype	pass	pass

Slika 14: Piglit izvješće nakon pokretanja testova

Na slici 14 su prikazani primjeri API testova za OpenCL implementaciju, mogući ishodi testova su prolaz, pad i preskakanje testa. Klikom na rezultat testa prikazuje se detaljni izvještaj za pojedini test. Na ovim primjerima prikazano je kako Piglit olakšava testiranje OpenCL implementacije i praćenje regresija u implementaciji.

4.2. Rezultati testova za Clover implementaciju standarda OpenCL

U nastavku je moguće vidjeti usporedbu rezultata pokretanja Piglit testova prije i nakon izmjena u kodu Clover-a, opisanih u šestom poglavlju ovog rada.



Result summary

Currently showing: fixes

Show: all | fixes | skips | problems | regressions | disabled | enabled | changes

	baseline (info)	with_patch (info)
all	3976/3980	3977/3980
custom	27/28	28/28
r600 create release buffer bug	fail	pass

Slika 15: Napredak u implelmentaciji potvrđen Piglit testovima

Na slici 15 prikazano je da apliciranjem promjena u Cloveru jedan od ostalih testova prolazi koji nije prolazio na početnoj implementaciji i nije došlo do regresije u ostalim testovima.

5. OpenMM

5.1. Općenito o programu OpenMM

Alat za molekularne simulacije OpenMM je jedan od brojnih aplikacija ubrzanih korištenjem GPU-a. OpenMM aplikacijski sloj sadrži kolekciju biblioteka, povezivanjem istih mogu se stvarati simulacije koristeći programski jezik Python. U nastavku je prikazan jednostavni primjer simulacije polja sila AMBER ff14SB i TIP3P-FB model vode sa 10000 iteracija koristeći Langevinov integrator.

```
from simtk.openmm.app import *
from simtk.openmm import *
from simtk.unit import *
from sys import stdout

pdb = PDBFile('input.pdb')
forcefield = ForceField('amber14-all.xml', 'amber14/tip3pfb.xml')
system = forcefield.createSystem(pdb.topology, nonbondedMethod=PME,
                                 nonbondedCutoff=1*nanometer, constraints=HBonds)
integrator = LangevinIntegrator(300*kelvin, 1/picosecond, 0.002*picoseconds)
simulation = Simulation(pdb.topology, system, integrator)
simulation.context.setPositions(pdb.positions)
simulation.minimizeEnergy()
simulation.reporters.append(PDBReporter('output.pdb', 1000))
simulation.reporters.append(StateDataReporter(stdout, 1000, step=True,
                            potentialEnergy=True, temperature=True))
simulation.step(10000)
```

Primjer korištenja OpenMM biblioteke sa Python programskim jezikom[15]

Simulacije je moguće pokrenuti na mnogim platformama, ali preporučeno je korištenje GPU-a putem jezika CUDA ili OpenCL zbog znatno boljih performansi. Usporedbu nekoliko GPU-a i CPU-a moguće je vidjeti u tablici 1.

Tablica 1: Usporedba brzine Nvidia GPU-a, AMD GPU-a i CPU-a prilikom izvođenja alata OpenMM

	CUDA Titan V	OpenCL Radeon RX 5700XT	CPU (i7 - 7740X)
Implicit, 2fs	1004	403	9.4
Implicit, 5fs	1437	521	23.2
Explicit- RF, 2fs	697	295	20.2

Iz tablice 1 vidi se da je izvođenje simulacija na GPU-ovima u svim situacijama brže od 100 do

1000 puta u odnosu na CPU. Ovakve razlike u performansama između CPU-a i GPU-a vidimo upravo zbog mogućnosti paralelizacije problema u molekularnim simulacijama i izvršavanje mnogo jednostavnih operacija na velikom skupu podataka (159 proteina sa 2489 atoma).

5.2. OpenMM testovi

OpenMM sadrži kolekciju testova sa svrhom validacije rezultata dobivenih pokretanjem funkcija koje pruža. OpenMM mora raditi očekivano na različitim platformama i sistemima što znači da sa hardverske strane mora podržavati CPU-e i GPU-e sa različitim softverskim implementacijama poput OpenCL-a, CUDA-e, a uz to mora podržavati i različite operacijske sustave. Kako bi se to ostvarilo OpenMM sadrži tri vrste testova. Unit testovi su mali testovi dizajnirani za testiranje specifičnih funkcija ili dijelova koda u potpunoj izolaciji od ostalog koda i sistema. Sistemski testovi za razliku od unit testova validiraju biblioteke u cijelosti i simuliraju realistične modele. Treća vrsta testova direktna je usporedba OpenMM-a sa drugim programima za molekularne simulacije poput GROMACS-a [16] i Tinker-a [17], kako bi se utvrdila točnost i očekivani rezultati. Za testiranje OpenCL implementacije od strane Clovera relevantni su unit testovi zbog relativno brzog i lakog iteriranja nad testovima.

5.2.1. Pokretanje testova

Testiranje OpenCL implementacije na OpenMM aplikaciji sastoji se od kompilacije OpenMM-a i pokretanja unit testova. Kako bi kompilirali OpenMM prvo je potrebno preuzeti kod sa GitHub-a i sve pakete koju su potrebni za podizanje OpenMM-a. Nakon toga izvorni kod je zatim potrebno kompilirati prema opisanom postupku:

```
cd openmm
mkdir builddir/ & cd builddir/
ccmake -i ../
make
```

Nakon uspješnog kompiliranja programa unit testovi bi trebali biti dostupni u *builddir/* direktoriju. Testovi se pokreću naredbom *make test*. Nakon izvršavanja testova dobivaju se rezultati za svaki pojedini test u obliku prolaza ili pada prilikom testiranja. Za više detalja izvršavanja nekog od testova može se koristiti linux paket gdb (GNU Project Debugger). Za dobivanje informativnih poruka pomoću gdb paketa potrebno je prilikom kompilacije postaviti zastavicu “Debug” za *CMAKE_BUILD_TYPE*, kako bi testovi imali traceback čitljiv ljudima.

5.2.2. Primjer testa

Unit testovi se mogu pokrenuti u grupi ili pojedinačno. Prilikom testiranja implementacije pokretani su testovi prvo u grupi kako bi se dobilo početno stanje implementacije prije izmjena na Cloveru. Za detaljnije razumijevanje događaja unutar testova potrebno je proučiti testove. Primjer izvornog koda testa koji se kasnije pokreće moguće je vidjeti u nastavku.

```
void testCMAPTorsions() {
    const int mapSize = 36;

    // Create two systems: one with a pair of periodic torsions, and one with a
CMAP torsion
    // that approximates the same force.

    System system1;
    for (int i = 0; i < 5; i++)
        system1.addParticle(1.0);
    PeriodicTorsionForce* periodic = new PeriodicTorsionForce();
    periodic->addTorsion(0, 1, 2, 3, 2, M_PI/4, 1.5);
    periodic->addTorsion(1, 2, 3, 4, 3, M_PI/3, 2.0);
    system1.addForce(periodic);
    ASSERT(!periodic->usesPeriodicBoundaryConditions());
    ASSERT(!system1.usesPeriodicBoundaryConditions());
    System system2;
    for (int i = 0; i < 5; i++)
        system2.addParticle(1.0);
    CMAPTorsionForce* cmap = new CMAPTorsionForce();
    vector<double> mapEnergy(mapSize*mapSize);
    for (int i = 0; i < mapSize; i++) {
        double angle1 = i*2*M_PI/mapSize;
        double energy1 = 1.5*(1+cos(2*angle1-M_PI/4));
        for (int j = 0; j < mapSize; j++) {
            double angle2 = j*2*M_PI/mapSize;
            double energy2 = 2.0*(1+cos(3*angle2-M_PI/3));
            mapEnergy[i+j*mapSize] = energy1+energy2;
        }
    }
    cmap->addMap(mapSize, mapEnergy);
    cmap->addTorsion(0, 0, 1, 2, 3, 1, 2, 3, 4);
    system2.addForce(cmap);
    ASSERT(!cmap->usesPeriodicBoundaryConditions());
    ASSERT(!system2.usesPeriodicBoundaryConditions());

    // Set the atoms in various positions, and verify that both systems give
equal forces and energy.

    OpenMM_SFMT::SFMT sfmt;
    init_gen_rand(0, sfmt);
    vector<Vec3> positions(5);
    VerletIntegrator integrator1(0.01);
    VerletIntegrator integrator2(0.01);
    Context c1(system1, integrator1, platform);
    Context c2(system2, integrator2, platform);
    for (int i = 0; i < 50; i++) {
        for (int j = 0; j < (int) positions.size(); j++)
            positions[j] = Vec3(5.0*genrand_real2(sfmt),
5.0*genrand_real2(sfmt), 5.0*genrand_real2(sfmt));
        c1.setPositions(positions);
```

```
        c2.setPositions(positions);
        State s1 = c1.getState(State::Forces | State::Energy);
        State s2 = c2.getState(State::Forces | State::Energy);
        for (int i = 0; i < system1.getNumParticles(); i++)
            ASSERT_EQUAL_VEC(s1.getForces()[i], s2.getForces()[i], 0.05);
        ASSERT_EQUAL_TOL(s1.getPotentialEnergy(), s2.getPotentialEnergy(), 1e-
3);
    }
}

/*
 * Ostali unit testovi izuzeti su zbog duljine koda
 *
 * */
int main(int argc, char* argv[]) {
    try {
        initializeTests(argc, argv);
        testCMAPTorsions();
        testChangingParameters();
        testPeriodic();
        runPlatformTests();
    }
    catch(const exception& e) {
        cout << "exception: " << e.what() << endl;
        return 1;
    }
    cout << "Done" << endl;
    return 0;
}
```

Primjer OpenMM Testa [18]

Test iz primjera prikazuje jedan od četiri unit testa iz izvornog koda zbog njegove duljine. Test prvo inicijalizira potrebne resurse, a zatim se sekvencijalno pokreću unit testovi. Svaki od njih validira dio OpenCL implementacije kako bi se osiguralo da OpenMM radi očekivano. Svi unit testovi prvo inicijaliziraju vrijednosti i pripremaju podatke za izračun. Specifični način na koji se test izvršava ovisi o implementaciji koja se koristi prilikom kompilacije testova mogu biti CPU, OpenCL ili CUDA. Prilikom kompilacije kod se prevodi u jezik definiran prilikom konfiguracije projekta. Jednom kompilirani kod može se pokretati koristeći izvršnu datoteku testa.

U nastavku je prikazan primjer pokretanja jednog od testova koji radi očekivano na trenutnoj implementaciji. Test je pokrenut unutar gdb-a za dobivanje više informacija.

```
run
Starting program: /home/dominik/Documents/openmm/build_openmm/TestOpenCLCMAPTorsionFo
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff7259700 (LWP 4975)]
...
[New Thread 0x7fffa37fe700 (LWP 4997)]
[Thread 0x7fffa37fe700 (LWP 4997) exited]
...
[Thread 0x7fffc77fe700 (LWP 4990) exited]
...
Done
[Thread 0x7ffff5a56700 (LWP 5245) exited]
...
[Thread 0x7fffed6ae700 (LWP 4980) exited]
```

Slika 16: OpenMM test pokrenut unutar programskog okvira GDB

Na slici 16 se vidi očekivani redoslijed izvođenja programa. Test na početku stvara threadove i alocira memoriju i resurse na GPU-u. Nakon što svaki od threadova završi izvođenje rezultati se vraćaju i thread se oslobađa za novo korištenje. Ovaj postupak se ponavlja sve dok svi podatci nisu obrađeni i svi testovi riješeni. Nakon vraćanja konačnog rezultata svi threadovi završavaju sa radom, memorija i rezervirani resursi na GPU-u se oslobađaju.

6. Izvođenje OpenMM testova na Clover-u

Postupak popravljanja testova započinje pronalaskom svih unit testova koji ne prolaze ili se ponašaju neočekivano. Pokretanjem testova na način prikazan u petom poglavlju dolazi se do sljedećih testova koji ne prolaze:

85% tests passed, 39 tests failed out of 266

Total Test time (real) = 5976.05 sec

The following tests FAILED:

44 - TestOpenCLAndersenThermostatSingle (SEGFAULT)
45 - TestOpenCLAndersenThermostatMixed (SEGFAULT)
46 - TestOpenCLAndersenThermostatDouble (SEGFAULT)
47 - TestOpenCLBrownianIntegratorSingle (SEGFAULT)
48 - TestOpenCLBrownianIntegratorMixed (SEGFAULT)
49 - TestOpenCLBrownianIntegratorDouble (SEGFAULT)
58 - TestOpenCLCheckpointsDouble (Failed)
59 - TestOpenCLCompoundIntegratorSingle (SEGFAULT)
60 - TestOpenCLCompoundIntegratorMixed (SEGFAULT)
61 - TestOpenCLCompoundIntegratorDouble (SEGFAULT)
75 - TestOpenCLCustomCompoundBondForceMixed (SEGFAULT)
86 - TestOpenCLCustomIntegratorSingle (SEGFAULT)
87 - TestOpenCLCustomIntegratorMixed (SEGFAULT)
88 - TestOpenCLCustomIntegratorDouble (SEGFAULT)
92 - TestOpenCLCustomNonbondedForceSingle (SEGFAULT)
93 - TestOpenCLCustomNonbondedForceMixed (SEGFAULT)
94 - TestOpenCLCustomNonbondedForceDouble (SEGFAULT)
122 - TestOpenCLLangevinIntegratorSingle (SEGFAULT)
123 - TestOpenCLLangevinIntegratorMixed (SEGFAULT)
124 - TestOpenCLLangevinIntegratorDouble (SEGFAULT)
131 - TestOpenCLMonteCarloAnisotropicBarostatSingle (SEGFAULT)
132 - TestOpenCLMonteCarloAnisotropicBarostatMixed (SEGFAULT)
133 - TestOpenCLMonteCarloAnisotropicBarostatDouble (SEGFAULT)

134 - TestOpenCLMonteCarloBarostatSingle (SEGFAULT)
135 - TestOpenCLMonteCarloBarostatMixed (SEGFAULT)
136 - TestOpenCLMonteCarloBarostatDouble (SEGFAULT)
143 - TestOpenCLNoseHooverIntegratorSingle (SEGFAULT)
144 - TestOpenCLNoseHooverIntegratorMixed (SEGFAULT)
145 - TestOpenCLNoseHooverIntegratorDouble (SEGFAULT)
164 - TestOpenCLVariableLangevinIntegratorSingle (Failed)
165 - TestOpenCLVariableLangevinIntegratorMixed (Failed)
166 - TestOpenCLVariableLangevinIntegratorDouble (Failed)
167 - TestOpenCLVariableVerletIntegratorSingle (Failed)
168 - TestOpenCLVariableVerletIntegratorMixed (Failed)
169 - TestOpenCLVariableVerletIntegratorDouble (Failed)
170 - TestOpenCLVerletIntegratorSingle (Failed)
171 - TestOpenCLVerletIntegratorMixed (Failed)
172 - TestOpenCLVerletIntegratorDouble (Failed)
174 - TestOpenCLVirtualSitesMixed (SEGFAULT)

Lista testova koji ne prolaze na Clover-u

Iz primjera je vidljivo da unit testovi ne prolaze sa različitim greškama. Testovi sa statusom "Failed" upućuju na problem u samom izvršavanju koda ili problem prilikom validacije rezultata sa očekivanim vrijednostima. "Exception: SegFault" upućuje na to da je u nekom trenutku izvršavanja testa, od incijalizacije podataka do vraćanja rezultata, došlo do problema sa alokacijom memorije ili prijenosom podataka između OpenMM-a i grafičkog stoga, a postoji i mogućnost da unutar Clovera dolazi do pogrešne komunikacije između softvera i hardvera. Kao početnu točku istraživanja problema uzima se traceback testova koji ne prolaze i proučavaju se pozivi unutar Clovera. Traženje problema unutar Clovera postavlja se kao početni korak zbog toga što isti testovi prolaze na OpenCL implementaciji koja dolazi uz vlasnički upravljački program ROCm. Ova kombinacija slučaja navodi na to da je problem u implementaciji OpenCL-a od strane Clovera, a ne u testovima koje nam pruža OpenMM. Za traceback problema koristi se gdb i valgrind [19]. Valgrind je programski okvir koji ima mogućnost dinamičke analize programa i automatskog pronalaženja problema sa memorijom i threadingom unutar programa. Nakon utvrđene liste problema testova kreće se od jednog od njih i proučava se koji od događaja radi nepredviđeno. U nastavku se vidi primjer korištenja gdb-a na testu koji ne prolazi. Prvo se pokreće test u izolaciji od ostalih testova i unutar gdb sučelja. GDB također ima mogućnost postavljanja break pointa unutar programa koji se

izvršava i u nastavku se koristi upravo ta mogućnost.

```
dominik@dominik:~/Documents/openmm/build_openmm$ gdb
./TestOpenCLAndersenThermostat
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./TestOpenCLAndersenThermostat...
(No debugging symbols found in ./TestOpenCLAndersenThermostat)
(gdb) run
Starting program:
/home/dominik/Documents/openmm/build_openmm/TestOpenCLAndersenThermostat
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff7259700 (LWP 4504)]
[New Thread 0x7ffff6a58700 (LWP 4505)]
[New Thread 0x7ffff6257700 (LWP 4506)]
[New Thread 0x7ffff5a56700 (LWP 4507)]
[New Thread 0x7ffff5255700 (LWP 4508)]
[New Thread 0x7fffed6ae700 (LWP 4509)]
[New Thread 0x7fffecd58700 (LWP 4510)]
[New Thread 0x7fffe7fff700 (LWP 4511)]
[New Thread 0x7fffe77fe700 (LWP 4512)]
[New Thread 0x7fffe6ffd700 (LWP 4513)]
[New Thread 0x7fffe67fc700 (LWP 4514)]
[New Thread 0x7fffe5ffb700 (LWP 4515)]
[New Thread 0x7fffe57fa700 (LWP 4516)]
[New Thread 0x7fffe4ff9700 (LWP 4517)]
[New Thread 0x7fffc3fff700 (LWP 4518)]
[New Thread 0x7fffc37fe700 (LWP 4519)]
[New Thread 0x7fffc2ffd700 (LWP 4520)]
[New Thread 0x7fffc27fc700 (LWP 4521)]
[New Thread 0x7fffc1ffb700 (LWP 4522)]
[New Thread 0x7fffc17fa700 (LWP 4523)]
[New Thread 0x7fffc0ff9700 (LWP 4524)]
[New Thread 0x7fffa3fff700 (LWP 4525)]
[New Thread 0x7fffa37fe700 (LWP 4526)]
[Thread 0x7fffa37fe700 (LWP 4526) exited]
[Thread 0x7fffa3fff700 (LWP 4525) exited]
[Thread 0x7fffc0ff9700 (LWP 4524) exited]
[Thread 0x7fffc17fa700 (LWP 4523) exited]
[Thread 0x7fffc1ffb700 (LWP 4522) exited]
[Thread 0x7fffc27fc700 (LWP 4521) exited]
[Thread 0x7fffc37fe700 (LWP 4519) exited]
[Thread 0x7fffc2ffd700 (LWP 4520) exited]
```

GDB ispis za test TestOpenCLAndersenThermostat ne otkriva mnogo o razlogu zbog čega test ne

prolazi, ali prema toku izvršavanja programa vidi se da test traje predugo i da se ne zatvaraju svi procesi pokrenuti na GPU-u kao na testovima koji prolaze očekivano. Ovo upućuje na to da se dio događaja ne propagira očekivano. Kao sljedeća opcija postavlja se Valgrind kako bi se utvrdilo postoji li curenje memorije prilikom pokretanja testova. Valgrind se pokreće na sljedeći način.

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
         --verbose  --log-file=LOG_FILE.txt ./OPENMMTEST
```

Pokretanjem navedene naredbe Valgrind pokreće OpenMM test unutar vlastitog programskog okvira i prati sve alokacije i de alokacije memorije kao i sve pozive funkcija prilikom izvršavanja testova. Izlaz Valgrind-a vidljiv je u nastavku.

```
==6015== LEAK SUMMARY:
==6015== definitely lost: 0 bytes in 0 blocks
==6015== indirectly lost: 0 bytes in 0 blocks
==6015== possibly lost: 225,986 bytes in 712 blocks
==6015== still reachable: 277,430,766 bytes in 7,512,316 blocks
==6015== of which reachable via heuristic:
==6015== multipleinheritance: 2,808 bytes in 13 blocks
==6015== suppressed: 0 bytes in 0 blocks
==6015==
==6015== ERROR SUMMARY: 26043 errors from 537 contexts (suppressed: 2 from 2)
==6015==
==6015== 25507 errors in context 1 of 537:
==6015== Conditional jump or move depends on uninitialised value(s)
==6015== at 0xB1DC343: ??? (in /usr/lib/x86_64-linux-gnu/libLLVM-10.so.1)
==6015== by 0xB11C5E7: llvm::MachineFunctionPass::runOnFunction(llvm::Function&) (in
/usr/lib/x86_64-linux-gnu/libLLVM-10.so.1)
==6015== by 0xAF86D75: llvm::FPPassManager::runOnFunction(llvm::Function&) (in
/usr/lib/x86_64-linux-gnu/libLLVM-10.so.1)
==6015== by 0xBBDDFE9: ??? (in /usr/lib/x86_64-linux-gnu/libLLVM-10.so.1)
==6015== by 0xAF8749F: llvm::legacy::PassManagerImpl::run(llvm::Module&) (in
/usr/lib/x86_64-linux-gnu/libLLVM-10.so.1)
==6015== by 0x8D78D9D: (anonymous namespace)::emit_code(llvm::Module&,
clover::llvm::target const&, llvm::CodeGenFileType, std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >&) (native.cpp:136)
==6015== by 0x8D790C5: clover::llvm::build_module_native(llvm::Module&,
clover::llvm::target const&, clang::CompilerInstance const&,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >&)
(native.cpp:148)
==6015== by 0x8D6078D: clover::llvm::link_program(std::vector<clover::module,
std::allocator<clover::module> > const&, clover::device const&,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >&)
(invocation.cpp:412)
==6015== by 0x8CE286E: clover::compiler::link_program(std::vector<clover::module,
std::allocator<clover::module> > const&, clover::device const&,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >&)
(compiler.hpp:59)
==6015== by 0x8CE321C: clover::program::link(clover::ref_vector<clover::device> const&,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&,
clover::ref_vector<clover::program> const&) (program.cpp:77)
==6015== by 0x8C76EA3: clBuildProgram (program.cpp:188)
==6015== by 0x4A663DA: cl::Program::build(std::vector<cl::Device,
std::allocator<cl::Device> > const&, char const*, void (*)(_cl_program*, void*), void*)
```

```
const (cl.hpp:5278)
==6015== Uninitialised value was created by a stack allocation
==6015== at 0xB1DAD3A: ??? (in /usr/lib/x86_64-linux-gnu/libLLVM-10.so.1)
```

Valgrind ispis za test TestOpenCLAndersenThermostat

U navedenom ispisu može se vidjeti nekoliko potencijalnih problema koji su nastali prilikom izvođenja programa. TestOpenCLAndersenThermostat test prilikom pokretanja unit testa testTemperature() poziva OpenCL funkcije poput clEnqueueNDRangeKernel i prilikom tog poziva Valgrind primjećuje dijelove memorije koje ne može dohvatiti nakon izvršavanja poziva. Valgrindov ispis potvrđuje početnu pretpostavku o curenju memorije i prikazuje malo bolju sliku o problematičnom dijelu testa. Primjećuje se da neki od poziva ne daju očekivane rezultate, a sljedeći korak je proučavanje samog koda OpenMM testa, kako bi se utvrdilo da OpenMM testovi koriste funkcije prema standardu definiranom od strane Khronos grupe. Pregledavanjem koda TestOpenCLAndersenThermostat testa vidljivog u nastavku moguće je utvrditi da svi pozivi unutar testa odgovaraju definiranom OpenCL standardu. Za razliku od nekih drugih implementacija OpenMM testovi nakon inicijalizacije resursa ne prate resurse i događaje nego prosljeđuju *null* vrijednost i prepuštaju upravljačkom stogu brigu oko praćenja rezerviranih resursa i događaja.

```
void OpenCLContext::executeKernel(cl::Kernel& kernel, int workUnits, int
blockSize) {
    if (blockSize == -1)
        blockSize = ThreadBlockSize;
    int size = std::min((workUnits+blockSize-1)/blockSize,
numThreadBlocks)*blockSize;
    try {
        currentQueue.enqueueNDRangeKernel(kernel, cl::NullRange,
cl::NDRange(size), cl::NDRange(blockSize));
    }
    catch (cl::Error err) {
        stringstream str;
        str<<"Error invoking kernel
"<<kernel.getInfo<CL_KERNEL_FUNCTION_NAME>()<<": "<<err.what()<<"
("<<err.err()<<")";
        throw OpenMMException(str.str());
    }
}
```

API poziv prema OpenCL-u od strane OpenMM testa

Definirani OpenCL standard dopušta rezerviranje resursa na način definiran u OpenMM testu i upozorava na slučajeve *null* vrijednosti za nedefinirana polja. U nastavku je moguće vidjeti definiciju poziva prema standardu OpenCL.

```
clEnqueueNDRangeKernel(cl_command_queue command_queue,
                       cl_kernel kernel,
                       cl_uint  work_dim,
                       const size_t *global_work_offset,
                       const size_t *global_work_size,
                       const size_t *local_work_size,
```

```
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
```

Definicija API poziva u standardu OpenCL

Usporedbom poziva iz OpenMM-a i pregledavanjem dokumentacije OpenCL-a moguće je uočiti da se lista događaja na koje program čeka i pokazivač na događaj unutar OpenMM-a postavlja na *null* vrijednosti što znači dvije stvari. Prva je da novi događaji generirani ne čekaju na izvršavanje prošlih događaja, a druga je da se ne prati koji se događaj trenutno izvršava što znači da ga nije moguće pronaći od strane aplikacije jednom kad se inicijalizira na GPU-u. Ovakav način implementacije potpuno je korektan od strane OpenMM-a. Saznanjima o implementaciji OpenCL poziva u OpenMM-u dobiva se bolji pogled u potencijalni problem i sljedeći korak je praćenje specifičnih poziva unutar Clovera što je moguće sa break point naredbom kao što je vidljivo u nastavku.

```
#0 clover::event::~event (this=0x0, __in_chrg=<optimized out>) at
../src/gallium/state_trackers/clover/core/event.cpp:36
#1 0x00007ffff326a1c4 in clover::hard_event::~hard_event (this=0x555555c7a2f0,
__in_chrg=<optimized out>) at ../src/gallium/state_trackers/clover/core/event.cpp:138
#2 0x00007ffff326a1e4 in clover::hard_event::~hard_event (this=0x555555c7a2f0,
__in_chrg=<optimized out>) at ../src/gallium/state_trackers/clover/core/event.cpp:141
#3 0x00007ffff3203174 in clover::intrusive_ref<clover::hard_event>::~intrusive_ref
(this=0x7fffffffd5a0, __in_chrg=<optimized out>) at
../src/gallium/state_trackers/clover/util/pointer.hpp:202
#4 0x00007ffff320c438 in clEnqueueNDRangeKernel (d_q=0x555555639348,
d_kern=0x555555734b08, dims=1, d_grid_offset=0x0, d_grid_size=0x7fffffffd700,
d_block_size=0x7fffffffd720, num_deps=0, d_deps=0x0, ... Thread 1 "TestOpenCLAnder" hit
Breakpoint 1, clover::event::~event (this=0x0, __in_chrg=<optimized out>) at
../src/gallium/state_trackers/clover/core/event.cpp:36 36 event::~event() {
(gdb) bt
#0 clover::event::~event (this=0x0, __in_chrg=<optimized out>) at
../src/gallium/state_trackers/clover/core/event.cpp:36
#1 0x00007ffff326a1c4 in clover::hard_event::~hard_event (this=0x555555c7a2f0,
__in_chrg=<optimized out>) at ../src/gallium/state_trackers/clover/core/event.cpp:138
#2 0x00007ffff326a1e4 in clover::hard_event::~hard_event (this=0x555555c7a2f0,
__in_chrg=<optimized out>) at ../src/gallium/state_trackers/clover/core/event.cpp:141
#3 0x00007ffff3203174 in clover::intrusive_ref<clover::hard_event>::~intrusive_ref
(this=0x7fffffffd5f0, __in_chrg=<optimized out>) at
../src/gallium/state_trackers/clover/util/pointer.hpp:202
#4 0x00007ffff324c7de in clEnqueueReadBuffer (d_q=0x555555639348, d_mem=0x555555626290,
blocking=1, offset=0, size=512, ptr=0x7fffedc16000, num_deps=0, d_deps=0x0, rd_ev=0x0) at
../src/gallium/state_trackers/clover/api/transfer.cpp:299
#5 0x00007ffff7dcc2c0 in OpenMM::OpenCLArray::download(void*, bool) const () from
/home/dominik/Documents/openmm/build_openmm/libOpenMMOpenCL.so
#6 0x00007ffff7df547a in
OpenMM::OpenCLUpdateStateDataKernel::setVelocities(OpenMM::ContextImpl&,
std::vector<OpenMM::Vec3, std::allocator<OpenMM::Vec3> > const&) () from
/home/dominik/Documents/openmm/build_openmm/libOpenMMOpenCL.so
#7 0x00007ffff7af66e5 in OpenMM::ContextImpl::setVelocities(std::vector<OpenMM::Vec3,
std::allocator<OpenMM::Vec3> > const&) () from
/home/dominik/Documents/openmm/build_openmm/libOpenMM.so.7.5
#8 0x00007ffff7af44bc in OpenMM::Context::setVelocities(std::vector<OpenMM::Vec3,
std::allocator<OpenMM::Vec3> > const&) () from
```

```
/home/dominik/Documents/openmm/build_openmm/libOpenMM.so.7.5
#9 0x00007ffff7af4542 in OpenMM::Context::setVelocitiesToTemperature(double, int) () from
/home/dominik/Documents/openmm/build_openmm/libOpenMM.so.7.5
#10 0x0000555555558321 in testTemperature() () #11 0x0000555555557a9f in main ()
(gdb) c
```

Izlaz GDB-a sa točkom prijeloma na Clover hard događaju.

Postavljanje break pointa na event::~event() GDB zaustavlja izvođenje programa na svakom od poziva koji odgovaraju zadanom izrazu. Postupak omogućuje praćenje odvijanja programa poziv po poziv. Backtrace prikazuje rezerviranje novih resursa, prebacivanje podataka na GPU i vraćanje dijela rezultata sa GPU-a, ali niti u jednom trenutku se ne prati koliko je događaja trenutno na GPU i u kojem se statusu nalaze, što dovodi do curenja memorije i testova koji ne prolaze ili ne rade očekivano. Nakon što je dokazano da OpenMM prati OpenCL standard, ali unatoč tome testovi ne prolaze, prelazi se na upravljački stog programa i kreće se od Clovera. Clover se sastoji od API i Core dijela implementacije. API dio implementacije služi za komunikaciju sa ostalim dijelovima stoga i programi koji koriste OpenCL svoje pozive šalju prema tom sučelju. Core dio programa obavlja funkcije komunikacije sa hardverom poput rezervacije memorije, prijenosa podataka i upravljanja greškama.

Clover razlikuje hard i soft događaje. Hard događaji u Cloveru vezani su uz komunikaciju sa hardverom, a soft događaji su komunikacija sa ostalim programima, priprema podataka za obradu na hardveru i vraćanje obrađenih podataka nazad prema programu u obliku koji je program zatražio. Pronalaskom OpenCL poziva unutar Valgrind-a i GDB-a koji stvaraju hard evente, ali ne čekaju na njihovo izvođenje, niti prate stanje izvođenja programa početna su točka traženja problema u Cloveru. Proučavanje OpenCL implementacije od strane Clovera započinje u API dijelu, kao prvom dijelu koji se poziva od strane vanjskog programa. API dio Clovera sadrži sučelje za stvaranje konteksta, inicijalizaciju uređaja, pozivanje događaja, kontrolu memorije na događaju, provjere platforme, stvaranje reda čekanja i transformacije poruka između uređaja i vanjskog programa. Core dio programa sadrži implementaciju navedenih sučelja na hardverskoj strani. U nastavku je vidljiv dio koda iz API dijela Clovera koji se pojavljuje u ispisu GDB-a i Valgrind-a.

```
CLOVER_API cl_int
clEnqueueNDRangeKernel(cl_command_queue d_q, cl_kernel d_kern,
                       cl_uint dims, const size_t *d_grid_offset,
                       const size_t *d_grid_size, const size_t *d_block_size,
                       cl_uint num_deps, const cl_event *d_deps,
                       cl_event *rd_ev) try {
   auto &q = obj(d_q);
   auto &kern = obj(d_kern);
   auto deps = objs<wait_list_tag>(d_deps, num_deps);
   auto grid_size = validate_grid_size(q, dims, d_grid_size);
   auto grid_offset = validate_grid_offset(q, dims, d_grid_offset);
   auto block_size = validate_block_size(q, kern, dims,
```

```
                                        d_grid_size, d_block_size);

   validate_common(q, kern, deps);

   auto hev = create<hard_event>(
      q, CL_COMMAND_NDRANGE_KERNEL, deps,
      [=, &kern, &q](event &) {
         kern.launch(q, grid_offset, grid_size, block_size);
      });

   ret_object(rd_ev, hev);
   return CL_SUCCESS;

} catch (error &e) {
   return e.get();
}
```

Dio Clover api/kernel.cpp koda

Na navedenom kodu prikazan je postupak inicijalizacije core poziva prema kernelu sa vrijednostima poput pokazivača na red, pokazivača na kernel, liste događaja na koju poziv čeka i rezerviranja dostupnih hardverskih resursa. Poziv bi zatim trebao vratiti pokazivač na inicijalizirani događaj i pokazivač na hard događaj. Ukoliko prilikom inicijalizacije dođe do pogreške kod vraća kodirani broj pogreške koja se dogodila. Prilikom izvođenja testova za OpenMM moguće je primijetiti da GDB i Valgrind ne podižu kod pogreške što navodi na to da Clover ne zna da je došlo do pogreške u izvođenju programa.

Dosad navedeni slučajevi dovode do prvog rješenja problema u izvođenju testova. OpenMM testovi izvode se bez problema na ROCm implementaciji programa, LLVM i Clang dokazano dobro kompiliraju programe što je vidljivo u pokretanju testova za OpenMM na ROCm implementaciji. Clover ovdje ostaje kao problematični dio stoga, a ispisi GDB-a i Valgrinda upućuju na curenje memorije prilikom izvođenja testova. Podiže se mogućnost pogreške u kodu prilikom implementacije brisanja ili vraćanja događaja sa GPU-a, te nastanka curenja memorije zbog loših pokazivača. Provjeravanje prve teorije započinje pregledavanjem već navedenog Clover koda. Navedeni kod trebao bi vratiti pokazivač na događaj i inicijalizirani hard događaj. U nastavku se vidi core klasa događaja unutar Clovera

```
class hard_event : public event {
   public:
      hard_event(command_queue &q, cl_command_type command,
                 const ref_vector<event> &deps,
                 action action = [](event &){});
      ~hard_event();

      virtual cl_int status() const;
      virtual command_queue *queue() const;
      virtual cl_command_type command() const;
      virtual void wait() const;
```

```
      const lazy<cl_ulong> &time_queued() const;
      const lazy<cl_ulong> &time_submit() const;
      const lazy<cl_ulong> &time_start() const;
      const lazy<cl_ulong> &time_end() const;

      friend class command_queue;

      virtual struct pipe_fence_handle *fence() const {
         return _fence;
      }

   private:
      virtual void fence(pipe_fence_handle *fence);
      action profile(command_queue &q, const action &action) const;

      const intrusive_ref<command_queue> _queue;
      cl_command_type _command;
      pipe_fence_handle *_fence;
      lazy<cl_ulong> _time_queued, _time_submit, _time_start, _time_end;
   };
```

Dio Clover core/event.hpp koda

Kod korektno definira sve pokazivače, metode, konstruktor i destruktor. Implementacija konstruktora odgovara pozivu iz API dijela koda, a zahvaljujući komentarima u kodu, definicija hard događaja postaje jasna. Hard događaj je događaj koji se izvršava od strane command queue-a, a smatra se izvršenim u trenutku kad hardver završi sa kalkulacijom događaja. Command queue je u ovom trenutku nepoznanica, te se istraživanje nastavlja tamo.

```
#include "api/util.hpp"
#include "core/queue.hpp"

using namespace clover;

CLOVER_API cl_command_queue
clCreateCommandQueue(cl_context d_ctx, cl_device_id d_dev,
                     cl_command_queue_properties props,
                     cl_int *r_errcode) try {
   auto &ctx = obj(d_ctx);
   auto &dev = obj(d_dev);

   if (!count(dev, ctx.devices()))
      throw error(CL_INVALID_DEVICE);

   if (props & ~(CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE |
                 CL_QUEUE_PROFILING_ENABLE))
      throw error(CL_INVALID_VALUE);

   ret_error(r_errcode, CL_SUCCESS);
   return new command_queue(ctx, dev, props);

} catch (error &e) {
   ret_error(r_errcode, e);
   return NULL;
}
CLOVER_API cl_int
clRetainCommandQueue(cl_command_queue d_q) try {
   obj(d_q).retain();
```

```
   return CL_SUCCESS;

} catch (error &e) {
   return e.get();
}

CLOVER_API cl_int
clReleaseCommandQueue(cl_command_queue d_q) try {
   auto &q = obj(d_q);

   q.flush();

   if (q.release())
      delete pobj(d_q);

   return CL_SUCCESS;

} catch (error &e) {
   return e.get();
}
// ...
CLOVER_API cl_int
clFlush(cl_command_queue d_q) try {
   obj(d_q).flush();
   return CL_SUCCESS;

} catch (error &e) {
   return e.get();
}
```

Dio Clover api/queue.api koda

Prikazani dio koda omogućava pogled u stvaranje, zadržavanje, oslobađanje i ispiranje događaja iz reda za izvršavanje. GDB i Valgrind izlazi upućuju na to da se prilikom izvođenja OpenMM testova pozivaju samo funkcije za stvaranje reda događaja i ne postoji niti jedan zapis o pozivu brisanja reda ili ispiranja reda događaja. Kao što je utvrđeno ranije, problem izgleda kao curenje memorije, a stvaranje događaja bez brisanja ili ispiranja reda događaja izgleda kao mogući problem u implementaciji. Kako bi se testirala teorija vezana za zadržavanje događaja nakon izvršavanja moguće je koristiti implementaciju metode *clFlush* direktnim pozivom iz funkcija unutar event.cpp i kernel.cpp API dijela Clovera. U nastavku je prikazano dodavanje ispiranja reda događaja nakon svakog događaja.

```
CLOVER_API cl_int
clEnqueueMarkerWithWaitList(cl_command_queue d_q, cl_uint num_deps,
                            const cl_event *d_deps, cl_event *rd_ev) try {
   auto &q = obj(d_q);
   auto deps = objs<wait_list_tag>(d_deps, num_deps);

   for (auto &ev : deps) {
      if (ev.context() != q.context())
         throw error(CL_INVALID_CONTEXT);
   }

   // Create a hard event that depends on the events in the wait list:
```

```
   // previous commands in the same queue are implicitly serialized
   // with respect to it -- hard events always are.
   auto hev = create<hard_event>(q, CL_COMMAND_MARKER, deps);

   ret_object(rd_ev, hev);
   q->pipe.flush();
   return CL_SUCCESS;

} catch (error &e) {
   return e.get();
}
```

Dodavanje ispiranja reda događaja

Dodavanje ispiranja reda događaja nakon svakog hard događaja dovodi do dva nova zaključka. Na suvremenom GPU-u (RX 480) u ovom slučaju test prolazi, ali na starijem GPU-u (R9 390) testovi i dalje ne prolaze, ali sa različitom greškom. Početna greška prilikom izvršavanja testa bila je SegFault, a trenutnom implementacijom dolazi do Timeout greške. Drugi zaključak je da su se performanse znatno smanjile zbog brisanja događaja i potrebe ponovne inicijalizacije za svaki od hard događaja. Unatoč problemima, promjena daje dovoljno informacija za daljnje traženje problema. Kako bi se performanse vratile na stare vrijednosti potrebno je brisati događaje na prikladniji način ili implementirati praćenje stanja događaja. Implementacija praćenja događaja u teoriji zvuči kao bolja opcija, ali pogledom na dio koda iz OpenMM testa navedenog prije, uočava se problem. Test stvara događaje na način da OpenCL implementaciji umjesto pokazivača na događaj i status događaja šalje *null* vrijednosti. Ovaj postupak je potpuno u skladu sa OpenCL standardom prema Khronos grupi, što znači da je Clover zadužen za pravilnu izvedbu unatoč tome. Preostaje mogućnost čišćenja događaja na optimiziraniji način.

Dosadašnjim pristupom pronađen je problem u implementaciji i dokazano je da se čišćenjem događaja oslobađa dovoljno resursa na GPU-u kako se ne bi događala SegFault greška. Daljnjim promatranjem koda dolazi se do potencijalnog rješenja, čekanje na hard događaje prije stvaranja novih. Pristup čekanja na događaje daleko je od idealnog, ali i dalje pruža rezultate uz manji utjecaj na performanse od čišćenja reda događaja navedenog u prijašnjem rješenju. Implementaciju ove solucije moguće je vidjeti u kodu navedenom u nastavku.

```
CLOVER_API cl_int
clEnqueueNDRangeKernel(cl_command_queue d_q, cl_kernel d_kern,
                       cl_uint dims, const size_t *d_grid_offset,
                       const size_t *d_grid_size, const size_t *d_block_size,
                       cl_uint num_deps, const cl_event *d_deps,
                       cl_event *rd_ev) try {
   auto &q = obj(d_q);
   auto &kern = obj(d_kern);
   auto deps = objs<wait_list_tag>(d_deps, num_deps);
   auto grid_size = validate_grid_size(q, dims, d_grid_size);
   auto grid_offset = validate_grid_offset(q, dims, d_grid_offset);
```

```
   auto block_size = validate_block_size(q, kern, dims,
                                         d_grid_size, d_block_size);

   validate_common(q, kern, deps);

   auto hev = create<hard_event>(
      q, CL_COMMAND_NDRANGE_KERNEL, deps,
      [=, &kern, &q](event &) {
         kern.launch(q, grid_offset, grid_size, block_size);
      });

   ret_object(rd_ev, hev);
   if (rd_ev == nullptr)
      hev().wait();
   return CL_SUCCESS;

} catch (error &e) {
   return e.get();
}
```

Verzija Clover api/event.cpp-a koja prolazi TestOpenCLAndersenTest

Promjene vidljive na linijama 24 i 25 navedenog koda rade sljedeće. Na liniji 24 postavljena je provjera u slučaju da Clover dobije poruku sa *null* pokazivačem prema događaju, a ako je to slučaj u linij 25 postavlja se čekanje na izvođenje tog događaja prije nastavka izvođenja testa. Ovaj postupak potrebno je postaviti na sve API pozive gdje postoji mogućnost prosljeđivanja *null* pokazivača, a funkcije se nalaze u api/event.cpp, api/kernel.cpp i api/transfer.cpp datotekama.

Izvršavanjem testova nakon apliciranja promjena u Cloveru dobiveni su sljedeći rezultati:

```
98% tests passed, 6 tests failed out of 266

Total Test time (real) = 8651.97 sec

The following tests FAILED:
 47 - TestOpenCLBrownianIntegratorSingle (Failed)
 133 - TestOpenCLMonteCarloAnisotropicBarostatDouble (Failed)
 165 - TestOpenCLVariableLangevinIntegratorMixed (Failed)
 170 - TestOpenCLVerletIntegratorSingle (Failed)
 171 - TestOpenCLVerletIntegratorMixed (Failed)
 172 - TestOpenCLVerletIntegratorDouble (Failed)
```

Rezultati OpenMM unit testova nakon izmjena u Cloveru

Navedenim promjenama u Cloveru smanjuje se mogućnost paralelizacije obrade podataka zbog toga što se obrada u dijelovima gdje se postavljaju signali za čekanje izvršava sekvencijalno. Ovaj pristup dovodi do duljeg izvršavanja testova i smanjenja performansi, ali prolazi gotovo sve OpenMM unit testove i ne primjećuje se regresija u Piglit testovima što je moguće vidjeti u četvrtom poglavlju rada. Trenutna implementacija nije idealna ali je dobar početak za popravljanje Clovera, a sljedeći koraci bili bi implementacija rješenja koje će blokirati događaje samo u slučaju da GPU ne može stvoriti nove ili praćenje događaja od strane Clovera.

7. Zaključak

Proizvodnja GPU-a ostvaruje veliku profit za tvrtke unutar industrije. Dvije najveće tvrtke Nvidia [20] i AMD [21] su 2019. godine imale preko šest milijardi Američkih dolara od prodaje GPU-a. Većina prodaje bila je upravo za kućnu upotrebu, primarno za računalne igre. Osim proizvodnje i prodaje za GPU-a za računalne igre, porast prodaje bilježi se i u upotrebi za podatkovne centre. AMD trenutno u izradi ima dva superračunala u SAD-u, Frontier [22] za U.S. Department of Energy koji bi trebao biti gotov u 2021. sa maksimalnom mogućnosti kalkulacije od preko 1.5 exaflop-a, a koristit će se za AI, Analizu podataka i velike simulacije. Frontier će koristiti AMD GPU-e i AMD CPU-e u omjeru 4:1 čime će osigurati mjesto kao najsnažnije superračunalo na svijetu. Projekt se procjenjuje na 600 milijuna Američkih dolara. El Capitan [23] planira se upogoniti dvije godine kasnije poslije Frontiera isto kao projekt za U.S. Department of Energy, a trebao bi pružiti preko 2 exaflop-a mogućnosti kalkulacija sa duplom preciznošću (64 Bita). El Capitan će se bazirati na sljedećoj generaciji AMD-ovih CPU-a i GPU-a.

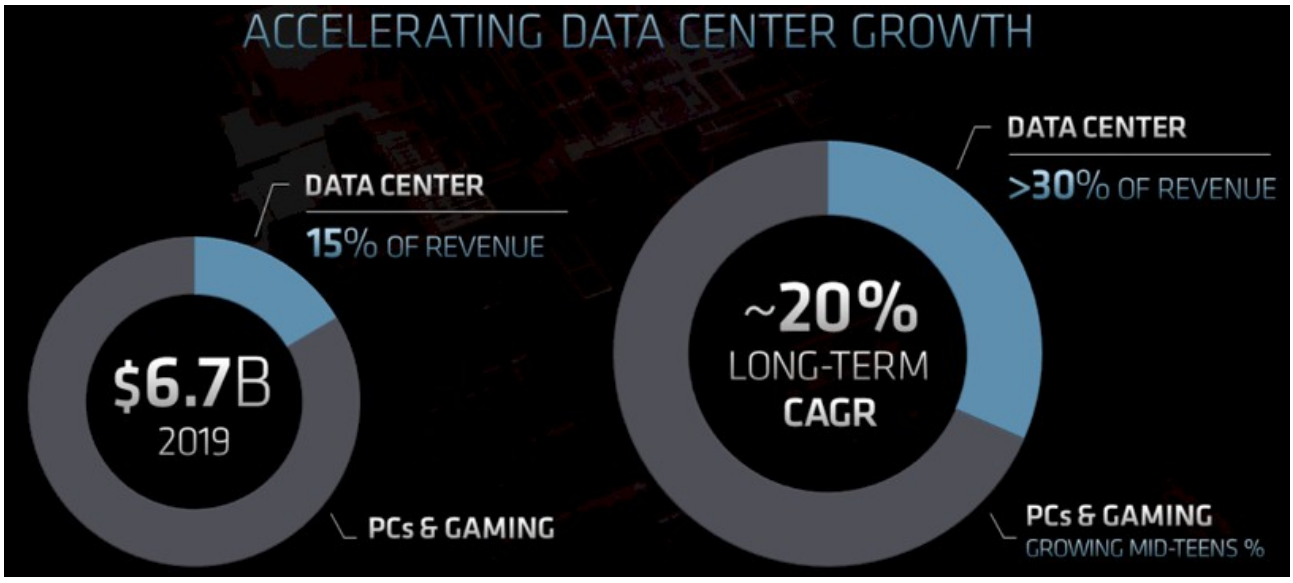


Figure 1: AMD zarada od prodaje GPU-a prema upotrebi i dugoročni plan

Nvidija također ima vlastita rješenja za GPU-e u superračunalima u obliku HGX solucija za ubrzanje obrade podataka bazirano na Nvidija A100 GPU-jezgrama. Korištenjem HGX-a unutar podatkovnog centra moguće je znatno ubrzanje treniranja AI modela, analize podataka i kalkulacija, za razliku od AMD superračunala Nvidija nema cjelovito rješenje za vlastito superračunalo.

Clover kao implementacija OpenCL-a se ne koristi na AMD GPU-ima u produkciji zbog svojih nedostataka i nesavršenosti implementacije. HPC računala umjesto Clovera koriste ROCm koji je

platforma napravljena od strane AMD-a i otvorenog je koda. Za razliku od Clovera, ROCm osim OpenCL-a implementira i modele za OpenMP, HIP i Python. Iako je ROCm isto upravljački program otvorenog koda, ostali modeli i implementacije čine ga znatno kompleksnijim od Clovera zbog čega je Clover dobar odabir za istraživanje i učenje stoga upravljačkih programa za izvođenje kalkulacija na GPU-ima.

Literatura

[1] “Tenzor jezgre, Nvidia.” Nvidia, [Online]. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/.

[2] “Deep Learning Super Sampling.” Nvidia, [Online]. Available: https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/.

[3] “RTXVoice.” Nvidia, [Online]. Available: https://www.nvidia.com/en-us/geforce/guides/nvidia-rtx-voice-setup-guide/.

[4] “Ray Accelerator.” AMD, [Online]. Available: https://www.amd.com/en/technologies/rdna-2.

[5] “AMD ROCm Github.” AMD, [Online]. Available: https://github.com/RadeonOpenCompute/ROCm.

[6] R. R. Everet, “The Whirlwind I Computer.” MIT, [Online]. Available: https://www.computer.org/csdl/pds/api/csdl/proceedings/download-article/12OmNBvkdmJ/pdf.

[7] J. Elliott, “Monochrome Display Adapter: Notes.” Aug. 06, 2020, [Online]. Available: http://www.seasip.info/VintagePC/mda.html.

[8] “iSBX 275 Video Graphics Controller Multimodule Board Reference Manual.” Intel Corporation, 1982, [Online]. Available: http://www.bitsavers.org/pdf/intel/iSBX/144829-001_iSBX_275_Video_Graphics_Multimodule_Sep82.pdf.

[9] “Very-Long Instruction Word (VLIW) Computer Architecture.” Philips Semiconductors, [Online]. Available: https://web.archive.org/web/20110929113559/http://www.nxp.com/acrobat_download2/other/vliw-wp.pdf.

[10] “RDNA 2 AMD GPU Arhitecture.” AMD, [Online]. Available: https://www.amd.com/en/technologies/rdna-2.

[11] “Khronos OpenCL Standard.” Khronos Group, [Online]. Available: https://www.khronos.org/opencl/.

[12] “Clang: a C language family frontend for LLVM.” LLVM org, [Online]. Available: https://clang.llvm.org/index.html.

[13] “clCreateImage2D Manual page.” Khronos Group, [Online]. Available: https://www.khronos.org/registry/OpenCL/sdk/2.2/docs/man/html/clCreateImage2D.html.

[14] “Piglit GitLab.” [Online]. Available: https://gitlab.freedesktop.org/mesa/piglit.

[15] “OpenMM User Guide.” OpenMM, [Online]. Available: http://docs.openmm.org/latest/userguide/application.html.

[16] “GROMACS Documentation.” GROMACS, [Online]. Available: http://www.gromacs.org/.

[17] “Tinker - Software Tools for Molecular Design.” [Online]. Available: https://dasher.wustl.edu/tinker/.

[18] “OpenMM GitHub, Source Code.” OpenMM, [Online]. Available: https://github.com/openmm/openmm/blob/master/tests/TestCMAPTorsionForce.h.

[19] “Valgrind User Manual.” Valgrind, [Online]. Available: https://valgrind.org/docs/manual/manual.html.

[20] “Nvidia Financial Information.” Nvidia, [Online]. Available: https://s22.q4cdn.com/364334381/files/doc_financials/2021/Q2/65b8fd67-6306-47d4-8a29-9a6be3860297.pdf.

[21] K. Fogary and T. Tariq, “Chasing profits and Intel, AMD sets sights on data centers.” S&P Global Market Intelligence, [Online]. Available: https://www.spglobal.com/marketintelligence/en/news-insights/latest-news-headlines/chasing-profits-and-intel-amd-sets-sights-on-data-centers-57996772.

[22] M. McCorkle, “U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL.” US Department of Energy, [Online]. Available: https://www.ornl.gov/news/us-department-energy-and-cray-deliver-record-setting-frontier-

supercomputer-ornl.
[23] “HPE and AMD power complex scientific discovery in world’s fastest supercomputer for U.S. Department of Energy’s (DOE) National Nuclear Security Administration (NNSA).” HP Enterprise, [Online]. Available: https://www.hpe.com/us/en/newsroom/press-release/2020/03/hpe-and-amd-power-complex-scientific-discovery-in-worlds-fastest-supercomputer-for-us-department-of-energys-doe-national-nuclear-security-administration-nnsa.html.

Popis Slika

Slika 1: MDA adapter (IBM, 1981. godina).....3
Slika 2: iSBX adapter za sliku (Intel, 1983. godina).....4
Slika 3: ATI Mach 64.....4
Slika 4: Razlika između arhitekture CPU-a i GPU-a.....5
Slika 5: Arhitektura TeraScale.....6
Slika 6: AMD Radeon RX 480, primjer arhitekture GCN.....8
Slika 7: Razlika između arhitektura GCN i RDNA.....10
Slika 8: Ubrzavač svijetlosnih zraka unutar CU-a (eng. Ray Accelerator).....11
Slika 9: Izlaz naredbe dmesg za IP blokove na AMD Radeon Rx 480 GPU.....12
Slika 10: Upravljački stog za GPU na Linux sustavu.....14
Slika 11: Primjer heterogenog paralelnog procesiranja omogućenog OpenCL-om.....16
Slika 12: Stog upravljačkih programa za AMD na Linux-u.....17
Slika 13: Razlike u implementacijama standarda Vulkan.....19
Slika 14: Piglit izvješće nakon pokretanja testova.....23
Slika 15: Napredak u implelmentaciji potvrđen Piglit testovima.....24
Slika 16: OpenMM test pokrenut unutar programskog okvira GDB.....29

Popis Tablica

Tablica 1: Usporedba brzine Nvidia GPU-a, AMD GPU-a i CPU-a prilikom izvođenja alata OpenMM.....25