Prevođenje programa za sjenčanje u strojni kod grafičkog procesora AMD Radeon

Božić, Sanja

Sveučilište u Rijeci – Odjel za informatiku

Jednopredmetni diplomski studij informatike, modul: Poslovna Informatika

Sanja Božić

Prevođenje programa za sjenčanje u strojni kod grafičkog procesora AMD Radeon

Diplomski rad

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

Rijeka, 22. rujna 2020.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 9. ožujka 2020.

Zadatak za diplomski rad

Pristupnik: Sanja Božić

Naziv diplomskog rada: Prevođenje programa za sjenčanje u strojni kod grafičkog procesora AMD Radeon

Naziv diplomskog rada na eng. jeziku: Compilation of shaders into AMD Radeon GPU machine code

Sadržaj zadatka: Programi za sjenčanje neizbježan su dio modernih računalnih igara gdje služe za prikaz svjetla i sjene 3D modela, promjenu zasićenja i tona boje, primjenu efekata kao što su volumetričko osvjetljenje te brojne druge svrhe. Proizvođači grafičkih procesora žele igračima osigurati optimalan ugođaj igranja pa je prevođenje programa za sjenčanje koje se događa dok igrač igra igru jedan od većih izazova za optimizaciju performansi; potrebno je postići i brzo prevođenje koda kako bi isti bio spreman za pokretanje prije nego igrač dođe do dijela gdje se kod pokreće i visoke performanse izvođenja rezultirajućeg strojnog koda. U okviru inicijative GPUOpen, započete u prosincu 2015. godine, AMD je u suradnji sa zajednicom okupljenom oko projekata Mesa i LLVM razvio prevoditelj programa za sjenčanje za vlastite grafičke procesore Radeon. Tijekom 2019. godine Valve je objavio da razvija alternativni prevoditelj programa za sjenčanje nazvan ACO koji LLVM zamjenjuje vlastitim generatorom strojnog koda i ima za cilj skratiti vrijeme prevođenja programa (koje je ovdje kritično) u aktualnim igrama koje koriste Vulkan (Strange Brigade, Rise of The Tomb Raider, Doom (2016) i sl.). Cilj rada je dati pregled strojnih instrukcija grafičkog procesora AMD Radeon arhitektura Graphics Core Next i/ili Radeon DNA te usporediti AMD-ovu i Valveovu implementaciju programa za sjenčanje u terminima dizajna softvera, vremena, točnosti i brzine izvođenja prevedenih programa.

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

Voditeljica za diplomske radove:
izv. prof. dr. sc. Ana Meštrović

Komentor:

Zadatak preuzet: 9. lipnja 2020.

Sažetak

Grafički procesori (GPU) značajan su dio mnogih računalnih sustava koje svakodnevno koristimo, od superračunala na kojima se predviđa vremenska prognoza i ponašanje lijekova do mobilnih uređaja na kojima gledamo YouTube i drobimo bombone u Candy Crushu. GPU-i su prošli dug proces evolucije od uređaja fiksnog cjevovoda do programabilnih procesora opće namjene kakve imamo danas. U tom procesu su se značajno mijenjali i hardver i softver i standardna aplikacijska programska sučelja koji se koriste. U radu opisujemo arhitekturu grafičkih procesora tvrtke AMD te detaljno objašnjavamo aplikacijska programska sučelja OpenGL i Vulkan. Zatim detaljno analiziramo rad upravljačkih programa za OpenGL i Vulkan te mjerimo njihove performanse. Kod upravljačkog programa za Vulkan koristimo dvije neovisne implementacije program-prevoditelja za sjenčanja, od kojih je prva zasnovana na LLVM-u, a druga samostojeća.

Ključne riječi

Vulkan, OpenGL, grafički procesor, GLSL, GPU arhitektura, GCN, RDNA, program za sjenčanje

Sadržaj

1. Uvod....1
2. Arhitekture AMD-ovih grafičkih procesora....2
2.1. Povijesni pregled arhitektura....3
2.1.1. TeraScale....3
2.1.2. GCN....4
2.1.3. RDNA....4
2.2. Način rada arhitektura....5
2.2.1. TeraScale....5
2.2.1.1. SIMD....6
2.2.1.2. VLIW....6
2.2.2. GCN....8
2.2.2.1. ACE....9
2.2.3. GCN četvrte generacije: Polaris....9
2.2.4. GCN pete generacije: Vega....9
2.2.4.1. FP16 i FP32....10
2.2.4.2. Primitivna sjenčanja....10
2.2.4.3. Bolje balansiranje opterećenja geometrije....10
2.2.5. RDNA....11
2.2.5.1. SIMD promjene....11
2.2.5.2. Raspoređivač....12
2.2.5.3. WGP....12
3. Standardi grupe Khronos....13
3.1. Grupa Khronos....13
3.1.1. OpenGL....13
3.2. Programi za sjenčanje....14
3.3. GLSL....14
3.3.1. Primjer vertex koda za sjenčanje....14
3.3.2. Primjer fragment koda za sjenčanje....15
3.3.3. Teksturiranje....15
3.3.4. Svjetlost....15
3.4. Vulkan....16
3.5. Programiranje Vulkan aplikacija....18
3.5.1. Potrebni paketi na operativnom sustavu Arch Linux....18
3.5.1.1. Koristeći grafičku karticu tvrtke NVIDIA....18
3.5.1.2. Koristeći grafičku karticu tvrtke AMD....18
3.5.2. Pisanje koda koristeći Vulkan API....19
3.5.3. GLFW....19
3.5.4. Stvaranje instance....20
3.5.5. Slojevi za validaciju....20
3.5.6. Fizički uređaji i skupine redova....21
3.5.6.1. Fizički uređaji....21
3.5.6.2. Skupine redova....22
3.5.7. Logički uređaji i redovi....22
3.5.7.1. Logički uređaji....22
3.5.7.2. Redovi....22
3.5.8. Površina prozora i swap chain....22
3.5.8.1. Prozor....22
3.5.8.2. Swap chain....23
3.5.9. Prikazi slika....23
3.5.10. Osnove grafičkog cjevovoda....23

3.6. SPIR-V....25
3.7. Standardi za AR/VR....26
4. Stog upravljačkih programa za AMD-ove grafičke procesore na Linuxu i projekt GPUOpen....27
4.1. Ubrzanje prikaza 3D grafike na modernim grafičkim procesorima....27
4.1.1. Međuspremnici okvira....27
4.2. DRM....28
4.2.1. Modul jezgre....29
4.3. Mesa....29
4.3.1. Povijest Mese....29
4.4. LLVM....30
4.4.1. Clang....31
4.4.2. Promjene u GCC-u....31
4.5. Pozadina program-prevoditelja R600....32
4.5.1. Pozadina AMDGPU kao evolucija R600....32
4.6. Video....32
4.7. OpenGL upravljački program RadeonSI....32
4.8. Vulkan upravljački program radv....33
4.9. ACO....33
5. OpenGL i GLSL....34
5.1. Apitrace....34
5.1.1. Glavne značajke Apitrace-a....35
5.2. Prevođenje koda za sjenčanje....37
5.2.1. Relevantne metrike....37
5.3. LLVM IR proces prevođenja bez i sa optimizacijama....38
5.3.1. Način sakupljanja podataka sa Steam-a....38
5.3.2. Primjeri prevođenja bez i sa optimizacije....39
5.3.2.1. Doom....39
5.3.2.2. Dota 2....41
5.3.2.3. Mad Max....42
5.3.2.4. Limbo....44
6. Mjerenje performansi: OpenGL vs Vulkan....47
6.1. Gallium HUD....47
6.2. MangoHud....48
6.3. FPS....50
6.4. Primjena MangoHUD-a za mjerenje FPS-a....50
6.4.1.1. Doom....51
6.4.1.2. Mad Max....53
6.4.1.3. Lara Croft GO....55
6.4.1.4. Serious Sam Fusion 2017....56
6.4.1.5. The Talos Principle....57
7. Borba za Vulkan: LLVM vs ACO....59
7.1. Doom....59
7.2. Dota 2....60
7.3. Dota Underlords....60
7.4. Mad Max....61
7.5. Rise of the Tomb Rider....61
7.6. Serious Sam Fusion 2017....62
7.7. The Talos Principle....62
7.8. vkQuake....63
8. Zaključak....64

1. Uvod

Grafički procesor (engl. *graphics processing unit*, kraće GPU, u doslovnom prijevodu “jedinica za obradu grafike”) je krajem 1990-ih postao često korišten izraz za komponentu koja pokreće grafiku na računalu. Iako su već ranije postojali čipovi tvrtke 3dfx i drugih proizvođača, izraz GPU prvi popularizira proizvođač čipova Nvidia svojim grafičkim procesorom GeForce 256, reklamiranim kao “prvi grafički procesor na svijetu” (engl. *the world’s first GPU*).

Uspjeh grafičkih kartica GeForce je osigurao da se povezane tehnologije poput hardverskog ubrzanja, a nekoliko generacija grafičkih procesora kasnije i programabilnog sjenčanja i obrade toka mogu razvijati dalje. Vremena su se mijenjala te je sada GPU procesor koji je posebno dizajniran za obradu intenzivnih zadataka prikazivanja grafike te za računanje masivno paralelnih, zahtjevnih i dugotrajnih zadataka. Osim Nvidije, danas vrlo značajan proizvođač grafičkih procesora i njihov najveći konkurent je AMD, čijim grafičkim procesorima ćemo se baviti u ovom diplomskom radu [1].

U nastavku ovog diplomskog rada će se kao prvo detaljno objasniti arhitekturu AMD-ovih grafičkih procesora. Naročito ćemo se fokusirati na posljednje dvije arhitekture koje pogotone gotovo sve danas korištene grafičke procesore Radeon, Graphics Core Next i Radeon DNA.

Nakon toga će se opisati aplikacijska programska sučelja OpenGL i Vulkan, standardizirana od grupe Khronos, te će se detaljnije objasniti stvaranje i prikaz grafike od strane upravljačkih programa koji rade na jezgri operacijskog sustava Linux.

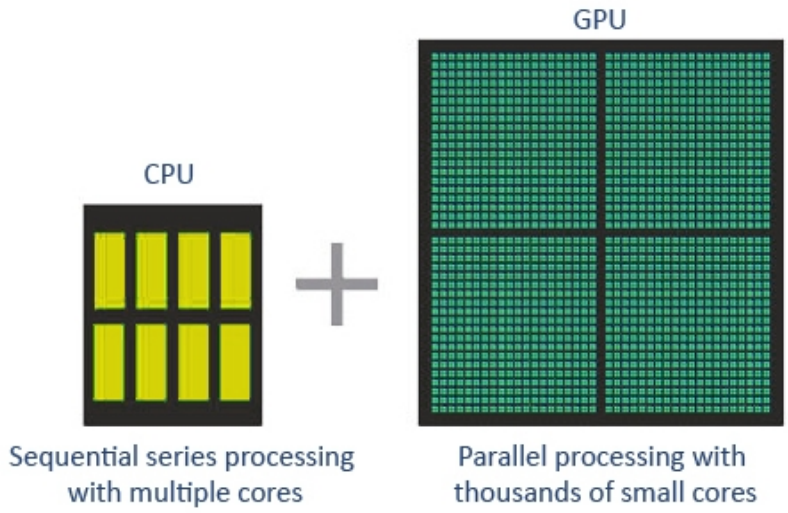
Zatim će se na primjerima sjenčanja prikazati prevođenje programskog jezika GLSL u strojni jezik grafičkog procesora uz korištenje optimizacije i bez njega, objasnit će se potreba za kratkim trajanjem prikaza jedne sličice kako bismo mogli prikazati veći broj sličica u sekundi te će se dati pregled izmjerenih performansi u nekoliko modernih računalnih igara.

Upravljački program za AMD-ove grafičke procesore ima dva program-prevoditelja, noviji ACO kojeg je razvila tvrtka Valve i otvorila kod te stariji AMDGPU koji je razvio AMD u suradnji sa zajednicom slobodnog softvera otvorenog koda unutar projekta LLVM. U radu će biti prezentirana mjerenja performansi oba program-prevoditelja u terminima kvalitete generiranog strojnog koda za grafički procesor i broja sličica u sekundi.

Naposljetku, u zaključku će se dati pregled napravljenih mjerenja i smjernice za budući rad.

2. Arhitekture AMD-ovih grafičkih procesora

Prije detaljnog opisa arhitekture AMD-ove grafičke procesorske jedinice (GPU) potrebno je definirati što je zapravo grafički procesor i kakva je arhitektura grafičkih procesora. GPU ubrzava aplikacije koje se pokreću na centralnoj programskoj jedinici (CPU) tako što prebacuje na sebe neke dijelove koda koji zahtijevaju računanje i koje troše mnogo vremena [2]. Ostatak aplikacije se i dalje izvodi na CPU-u. Aplikacija radi brže jer koristi paralelnu procesorsku snagu GPU-a. Za usporedbu: CPU se sastoji od četiri do osam CPU jezgri, dok se GPU sastoji od stotina manjih jezgri. Paralelna arhitektura rada na toliko puno više jezgri istovremeno je ono što GPU-u daje visoke računske performanse. Na slici 1 vizualno je prikazana razlika između CPU-a i GPU-a [3].



Slika 1: Razlika između CPU i GPU, preuzeto iz: https://www.apps4rent.com/wp-content/uploads/2018/04/cpu-vs-gpu.jpg

Kako bi se bolje razumjelo kako rade AMD GPU-ovi u ovom poglavlju biti će objašnjena njihova arhitektura. Da bi bilo lakše objasniti najnoviju verziju Radeon DNA (kraće RDNA) arhitekture na početku će biti objašnjeni prethodnici RDNA arhitekture, koji su TeraScale i Graphics Core Next (kraće GCN) arhitektura [4].

- TeraScale je prva arhitektura koja je došla na tržište još 2007. godine i prestala se uporabljati 2013. godine (imala je tri uspješne generacije).
- Nakon TeraScale-a je 2012. došla na tržište GCN arhitektura koja je završila 2019 godine (imala je čak 5 generacija).

- RDNA je sad najnovija arhitektura koja je došla na tržište 2019. i još je na svojoj prvoj generaciji.

2.1. Povijesni pregled arhitektura

2.1.1. TeraScale

TeraScale je bila prva ATI-jeva arhitektura GPU-a za koju je temeljni skup instrukcija (engl. *instruction set architecture*, kraće ISA, u suštini je skup uputa koje procesor podržava [5]) i mikroarhitektura javno detaljno predstavljena [6]. Vrijedi spomenuti da je predstavljena u značajnom trenutku povijesti razvoja grafičkih procesora u kojem je tek počeo koncept general purpose GPU, kraće GPGPU.

Što je zapravo GPGPU? Kad obično razmišljamo o GPU-u, mislimo na aplikacije ili programe koji intenzivno koriste grafičko sučelje neke vrste. Na primjer: korištenje grafičke kartice za vrijeme igranja jedan je od najpopularnijih načina za prikazivanje grafičke snage GPU-a. No, GPGPU (*General-purpose computing on graphics processing units*) nadmašuje prijašnje korištenja GPU-a. Danas se svi suvremeni GPU-ovi smatraju GPGPU-ovima, jer se mogu koristiti ne samo za grafiku, već i za pokretanje izračuna i izvršavanje zadataka, baš kao što to mogu CPU-ovi [7]. Oni mogu pokrenuti više operacija u kraćem vremenu od CPU-a. To čini korištenje GPGPU-a idealan izbor za sve trenutne i buduće aplikacije. No, što je danas normala nekoć nije bila, vratimo se natrag u 2007.

Kako se koncept GPGPU tad tek počeo prihvaćati, AMD-ov prvi potez na taj teritorij došao je u obliku podrške za OpenCL (*Open Compute Library*) biblioteku na njihovim TeraScale Gen1 (prva generacija) GPU-ovima. OpenCL je biblioteka otvorenog koda za računanje na heterogenim sustavima, tj. sustavima koji kombiniraju različite vrste procesora kao što su CPU i GPU. Nadalje, AMD-ova Fusion inicijativa nastojala je spojiti CPU-ove i GPU-ove, što je dodatno potisnulo heterogene arhitekture sustava (HSA) i rezultiralo stvaranjem „Ubrzane procesne jedinice“ ili APU-a, koji se i danas koristi.

Iako su temelji AMD-ovog GPGPU-a prvo čvrsto postavljeni unutar arhitekture TeraScale-a, TeraScale-ov nasljednik GCN učvrstio je prednost AMD-e inicijative za GPGPU.

2.1.2. GCN

GCN je dominantna GPU arhitektura AMD-a u i trenutno se nalazi u Polaris i Vega GPU-u s Polaris-ima koji čine četvrtu generaciju i Vega-ma koje čine petu i posljednju iteraciju GCN-a. On predstavlja arhitekturu budućih programibilnih i heterogenih sustava. Osim toga GCN je poznat po svom dizajnu računarske jedinice (CU – *computer unit*) koji se koristi u nizu programa za sjenčanje [8].

GCN je prvi put predstavljen tržištu 2012. godine u tada novoj Radeon HD 7700 seriji GPU-ova. Nakon 8 godina proizvodnje GCN je i više nego zaslužio svoj položaj na tržištu sa svojih 5 generacija.

Trajanje generacija:

1. GCN Gen 1 - od 2012. do 2015. godine
2. GCN Gen 2 - od 2013. do 2015. godine
3. GCN Gen 3 - od 2014. do 2015. godine
4. GCN Gen 4 - od 2016. do 2017. godine
5. GCN Gen 5 - od 2017. do 2019. godine

AMD je uspio svakom novom generacijom GCN-a izvući još bolje i brže performanse no i tome je morao doći kraj. Tako je AMD odlučio predstaviti RDNA arhitekturu koja sadrži značajne promjene u samim temeljima arhitekture detaljno opisane u kasnijim poglavljima [8].

2.1.3. RDNA

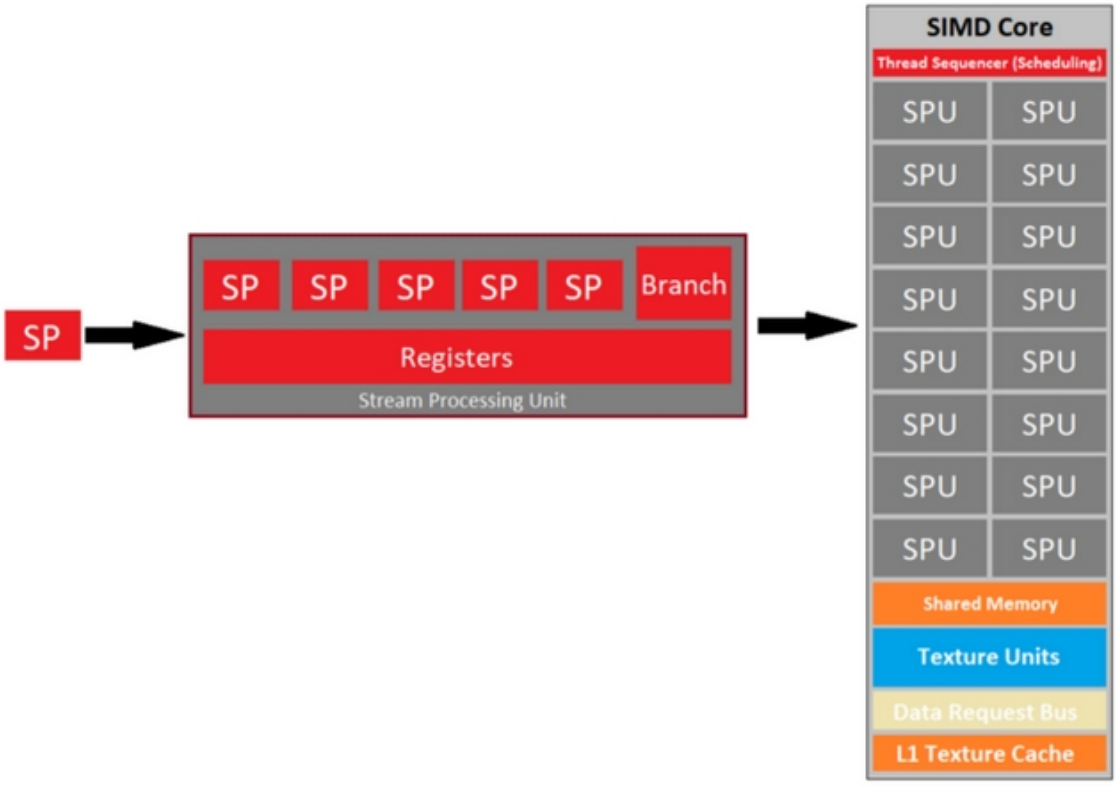
Kako mu je dodan isti broj računskih resursa kao čip temeljen na GCN-u, RDNA uspijeva obaviti više posla dok zahtijeva manje dretvi (engl. *threadova*) u cjevovodu kako bi resursi bili dobro iskorišteni i zauzeti [9]. Opis njene arhitekture i nešto više o samom RDNA opisano je u sljedećem poglavlju.

2.2. Način rada arhitektura

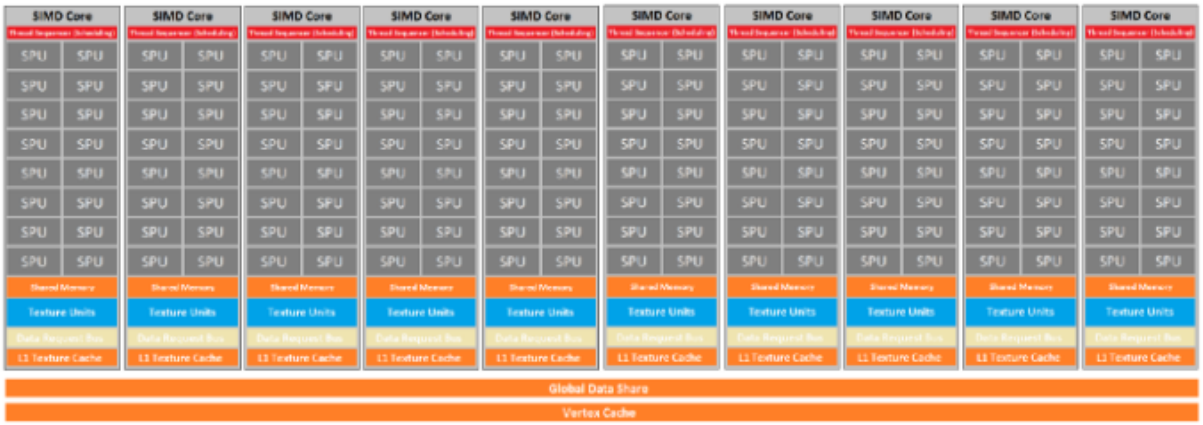
2.2.1. TeraScale

Kao i kod svih AMD-ovih GPU-ova danas, najosnovniji izvedbeni blok je SP (*Stream Processor*). SP je jedinica za aritmetiku i logiku (ALU - *Arithmetic and Logic Unit*), spomenuta je prije pri opisu TeraScale-a. ALU-ovi su, kako im i ime kaže, specijalizirani za izvršavanje matematičkih operacija.

U TeraScale-u nekoliko SP-ova i kontrolna jedinica podružnice, zajedno s registrima za pohranu čine jedinstvenu SPU (*Stream Processing Unit*). Nekoliko SPU-a zajedno s više upravljačkih jedinica i registara za pohranu čine jednu SIMD jezgru. Nekoliko SIMD jezgri i još više upravljačkog hardvera na kraju čine kompletan TeraScale GPU. Ilustracija TeraScale GPU-a prikazana je na slikama 2 i 3.



Slika 2: Prikaz od SP-a do izgleda jedne SIMD jezgre, preuzeto iz: https://miro.medium.com/max/840/1GIINKU8XKq0OgBQctfNPcQ.png



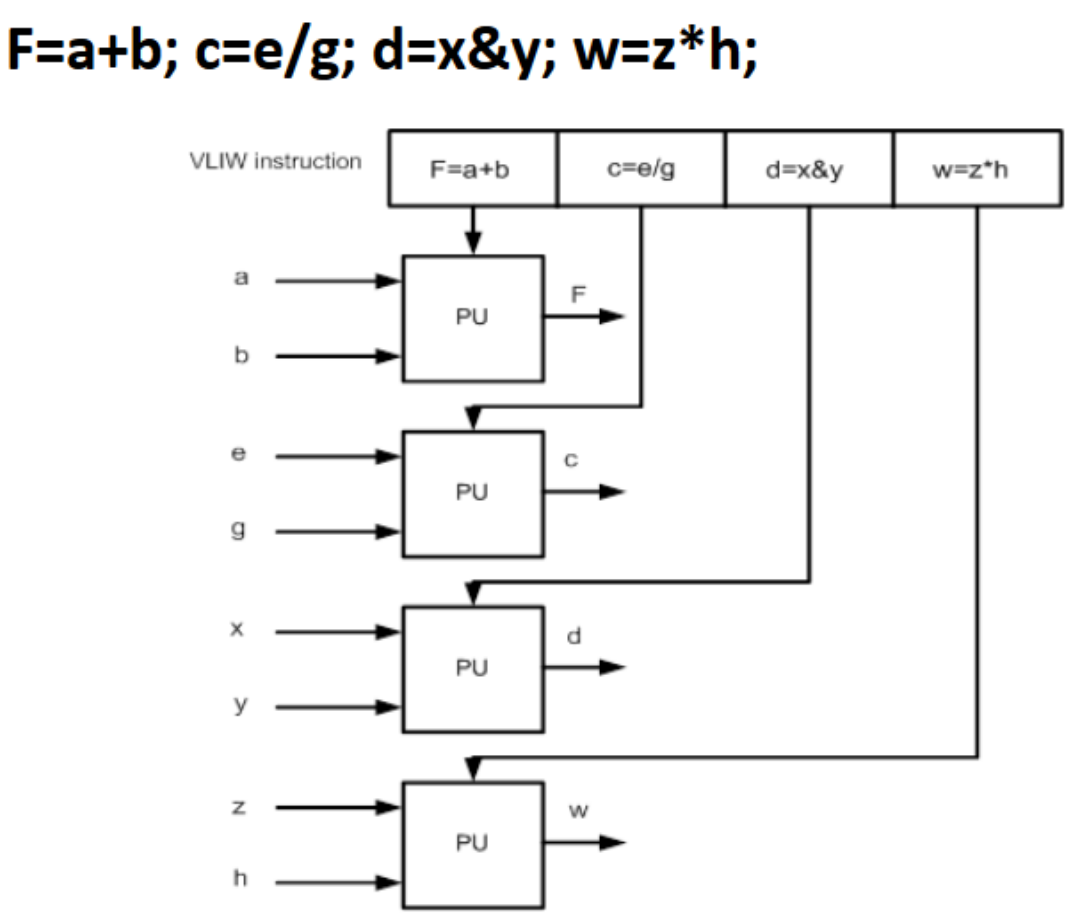
Slika 3: Detaljan izgled jednog TeraScale GPU-a sa 10 SIMD (primjer ovakvog GPU-a je Radeon HD 4870 GPU), preuzeto iz: https://miro.medium.com/max/840/1CCOLOIDUmeZd4ynAVNqnSg.png

2.2.1.1. SIMD

SIMD (*Single Instruction, Multiple Data*) odnose se na hardverske komponente koje istodobno obavljaju istu operaciju na više podatkovnih operanda. Uobičajeno, SIMD jedinica prima kao ulaz dva vektora (svaki ima skup operanda), izvodi istu operaciju na oba skupa operanda (po jedan operand iz svakog vektora) i izbacuje vektor s rezultatima [10].

2.2.1.2. VLIW

Osim toga TeraScale u sebi ima čip nazvan „vrlo duga riječ s uputama” (engl. *Very Long Instruction Word*, kraće VLIW) koji je jedan od ISA tipova (*Instruction Set Architecture*). VLIW opisuje arhitekturu računalne obrade u kojoj program-prevoditelj jezika ili predprocesor razdijeli programsku instrukciju na osnovne operacije koje procesor može paralelno izvoditi [11]. Te se operacije stavljaju u “vrlo dugu riječ s uputama” koje procesor može razdvojiti bez daljnje analize i predati svaku operaciju odgovarajućoj funkcionalnoj jedinici.

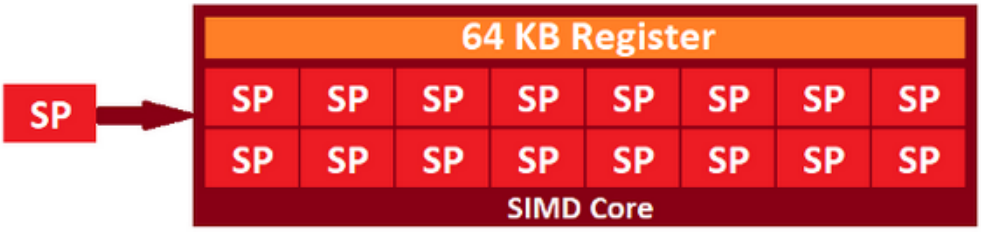


Slika 4: Primjer VLIW-a, preuzeto iz: https://slideplayer.com/slide/6988693/

Na slici 4 prikazan je primjer VLIW-a. Na primjeru je prikazana jedna linija koda koja u sebi ima 4 različite osnovne operacije. VLIW šalje tu uputu procesoru koji onda paralelno rješava sva 4 problema te se tako izračuni dobiju znatno brže nego da bi se izračunavali jedan po jedan.

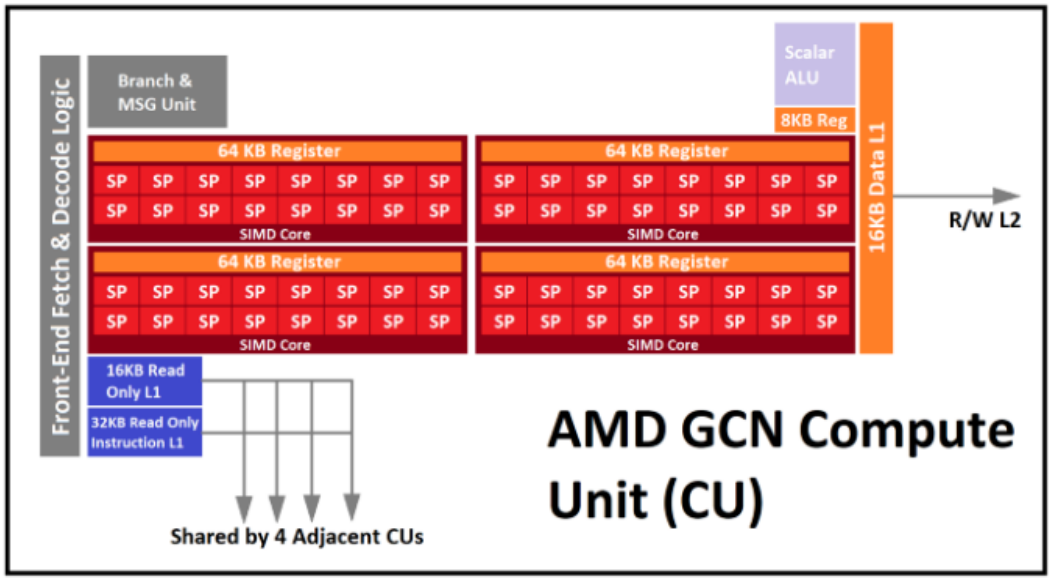
2.2.2. GCN

U slučaju GCN-a, slojevitost u arhitekturi GPU-a postaje malo kompliciranija. Pojedini procesori protoka (SP) još uvijek čine temeljne blokove te se jedan SIMD sastoji od 16 SP-a. Nakon toga 4 zasebne SIMD jedinice za vektorsku obradu čine jedan CU (*Compute Unit*). Tako da sad dijagram jedne SIMD jedinice izgleda kao na slici 5.



Slika 5: GNU - jedna SIMD jedinica, preuzeto iz:
*https://miro.medium.com/max/840/1*EzGHlEZZp8C_6UExpsUB_A.png*

Kao što je napisano iznad, 4 SIMD-a stvaraju jedan CU, a jedan GCN GPU ima nekoliko CU-a. Izgled jednog CU-a prikazan je na slici 6.



Slika 6: Izgled GCN CU-a, preuzeto iz:
*https://miro.medium.com/max/840/1*hIJcIz_PAm_PQH3ceAmurw.png*

Za razliku od TeraScale-a, GCN je čista SIMD arhitektura. Valna fonta (engl. *wavefront*) se više ne sastoji od VLIW dretvi, već od 64 pojedinačne podatkovne točke koje izvršava 16 SP-a u jezgrama SIMD-a. Nadalje, svaka jedinica za računanje sadrži četiri SIMD jezgre i svaka od njih može raditi na zasebnim valnim frontovima, tako da u bilo kojem trenutku CU može obraditi do četiri različita valna fronta [8], [12].

2.2.2.1. ACE

Osim toga, novo u GCN-u je zakazivanje računanja radnog opterećenja (engl. *scheduling of compute workloads*) kroz CU zvan ACE (*Asynchronous Compute Engines*) koji predsjedaju raspodjelom resursa, prebacivanjem konteksta i prioritetima zadataka. Jedan GCN GPU može u sebi imati nekoliko ACE-a [8].

2.2.3. GCN četvrte generacije: Polaris

Polaris donosi podršku za dohvaćanje uputa unaprijed (engl. *instruction pre-fetching*), prediktivni proces u kojem procesori pretpostavljaju upute koje će se sljedeće izvršavati na temelju trenutnog stanja izvršenja i zatim preuzimaju te upute. Ispravno prethodno dohvaćanje dovodi do značajnih poboljšanja performansi jer procesor ne mora čekati da se podaci učitaju iz memorije. No, nepravilnim dohvaćanjem uputa unaprijed dolazi do toga da se upute odbacuju, što smanjuje učinkovitost.

Osim toga u četvrtoj generaciji AMD je nadodao primitivni akcelerator odbacivanja (engl. *Primitive Discard Accelerator*, kraće PDA). Budući da su površine i složeni oblici sastavljeni od mnogo manjih i jednostavnijih mnogokuta (tipično od trokuta), PDA odbacuje vizualno beznačajne trokute (skrivene ili premale) kako bi se povećale performanse. To jest PDA traži ako je geometrija u vidnom polju (FOV - engl. *Field of view*) ili ne, zatim odbacuje one koji nisu potrebni kako bi GPU mogao raditi manje geometrije te tako ubrzava cijeli proces [13].

2.2.4. GCN pete generacije: Vega

Najveća promjena dolazi u petoj generaciji GCN-ova (Vega) koji imaju mogućnost računanja operacija nad brojevi s pomičnom točkom veličine 16 bitova (kraće FP16). Dok se još u trećoj generaciji uveo FP16, same operacije se nisu izvršavale ništa brže jer je svaki pojedinačni SP još uvijek mogao rukovati samo jednom operacijom, bilo FP16 ili FP32. Sa Vegom se to značajno promijenilo. Svaki SP sada može podnijeti dvije FP16 operacije umjesto jedne FP32 operacije,

značajku koju AMD naziva ***"Rapid Packed Math"*** *[14]*.

2.2.4.1. FP16 i FP32

FP32 i FP16 znače engl. *32-bit and 16-bit floating point*. GPU-ovi su se prvobitno fokusirali na FP32 jer su ti proračuni potrebni za 3D igre. Danas puno GPU-a ima izvornu podršku FP16 kako bi se ubrzao proračun neuronskih mreža.
GPU-ovi koji podržavaju FP16 gotovo su dvostruko brži u izračunavanju FP16 nego FP32. *Na primjer*: uzimajući u obzir da su novije kartice koje podržavaju FP16 (poput NVidia RTX 2080 serije) također za oko 20% brže od FP32 u odnosu na prethodnika (GTX 1080), možemo dobiti porast od 140% za treniranje neuronskih mreža FP16 u usporedbi s FP32 na prethodnim karticama [15].

2.2.4.2. Primitivna sjenčanja

Slijede poboljšanja grafičkih motora s uvođenjem primitivnih sjenčanja (engl. *primitive shaders*) koji omogućuju brzo odbacivanje primitivnih objekata tj. vizualno beznačajnih poligona (skrivenih ili vrlo malih trokuta), zajedno s (engl. *Draw Stream Binning Rasterizer*, kraće DSBR). Ovaj stalni naglasak na izbacivanju pomaže u sprječavanju začepljenja cjevovoda pri uklanjanju i oslobađanju GPU-a kad radi bez potrebe, smanjuje pristup memoriji i štedi propusnost memorije uz smanjenje potrošnje energije.

2.2.4.3. Bolje balansiranje opterećenja geometrije

GCN Vega je također donijela poboljšanja kod balansiranja opterećenja u svim motorima (engl. *engine*) geometrije/sjenčanja, pri kojem dolazi do duplo brže geometrijske propusnosti uskih grla (engl. *Bottlenecks*) [14].

2.2.5. RDNA

U najnovijoj AMD arhitekturi dolazimo do tri velike promjene koje su:

1. ažuriranje izračunskih jedinica (CU)
2. dodavanje novog sloja za pred-memoriranje
3. poboljšanja energetske učinkovitosti

2.2.5.1. SIMD promjene

GCN-ove SIMD jezgre su sad duplo veće tj. svaka SIMD jezgra sad ima 32 SP-a u sebi. Izgled SIMD jezgre u RDNA arhitekturi prikazana je na slici 7.

*Slika 7: Promjena SIMD jezgre, preuzeto iz: https://miro.medium.com/max/840/1*7pofYttTr73iNHu4qI5arw.png*

Osim ove velike promijene došlo je do još jedne. U prijašnjoj arhitekturi bilo je 4 SIMD jezgre koje su sada spojene u dvije veće. Uz to svaka SIMD jezgra sada ima svoj raspoređivač (engl. *scheduler*). Kod prijašnjih GCN-ova postojao je samo jedan za cijeli CU.

*Slika 8: RDNA CU, preuzeto iz: https://miro.medium.com/max/840/1*jQyTx-Bj4fx2k3rewkHhMA.png*

Na slici 8 prikazuje se novi izgled prije objašnjenog RDNA CU-a, sivi pravokutnici kod svakog SIMD-a prikazuju 2 raspoređivača koja su novost u RDNA arhitekturi. Najznačajnije razlike GCN-a i RDNA-a:

Tablica 1: Razlike između GCN i RDNA arhitekture

GCN	**RDNA**
64 SP-a u 4 SIMD-a sa 16 jezgri	64 SP-a u 2 SIMD-a sa 32 jezgri
1 skalarni ALU	2 skalarni ALU, 1 po SIMD jezgri
1 raspoređivač po CU-u	1 raspoređivađ po SIMD jezgri, 2 po CU-u
Dijeljena predmemorija kroz 4 CU-a	Dijeljena predmemorija i LDS kroz 2 CU-a

2.2.5.2. Raspoređivač

Raspoređivač, kao što se može pretpostaviti po imenu, određuje kada će se koji kod za sjenčanje izvoditi. Moguće je izvoditi i do 10 kodova za sjenčanje istovremeno. Raspoređivač je zadužen za određivanje rasporeda izvođenja.

Raspoređivanje može povećati pritisak registra (engl. *register pressure*). Zbog toga visok pritisak registra može dovesti do prolijevanja (engl. *spill*) memorije. Kad dolazi do prolijevanja memorije smanjuje se paralelizacija na procesima. To znači da se tada smanjuje istovremeno izvođenje kodova za sjenčanje s maksimalnih 10 na neku nižu vrijednost [16]. Za rješavanje tog problema potrebno je da raspoređivač raspoređuje kod za sjenčanje pod ograničenjima pritiska registra.

2.2.5.3. WGP

Daljnje promjene omogućuju dijeljenje resursa preko CU-ova. RDNA spaja dva CU-a u jedan procesor radne skupine (engl. *Work Group Processor*, kraće WGP) sa skalarnom pred-memorijom podataka i uputa dijeljenom u WGP-u zajedno s lokalnim udjelom podataka (kraće LDS). GCN je s druge strane dijelio svoju skalarnu pred-memoriju na četiri susjedna CU-a uz zadržavanje namjenskog LDS-a po CU-u [9], [17].

3. Standardi grupe Khronos

3.1. Grupa Khronos

Grupa Khronos je nepfrofitni američki industrijski konzorcij kojeg financiraju članovi. Fokusiran je na stvaranje otvorenih standarda aplikacijskih programskih sučelja (engl. *application programming interface*, kraće API) bez licenci za izradu autorskih podataka i ubrzano reproduciranje dinamičnih medija na širokom rasponu platformi i uređaja.

Članovi Khronosa mogu doprinijeti razvoju specifikacija Khronos API-ja, glasati u različitim fazama prije javne primjene i ubrzati isporuku svojih platformi i aplikacija ranim pristupom nacrtima specifikacija i testovima sukladnosti.

Neki od standarda koji se još danas koriste, a stvorila ih je Khronos grupa su OpenCL (API za računanje koji radi na različitim platformama), OpenGL (API za računalnu grafiku koji radi na različitim platformama), OpenVG (API za ubrzanje obrade 2D vektorske grafike), Vulkan (API za računalnu grafiku sa malim utjecajem na performanse), WebGL (JavaScript koji je povezan sa OpenGL ES unutar preglednika na bilo kojoj platformi koja podržava OpenGL ili OpenGL ES grafičke standarde) te mnogi drugi [18].

3.1.1. OpenGL

OpenGL je okruženje (engl. *environment*) za razvoj prijenosnih, interaktivnih 2D i 3D aplikacija. Prvi put je predstavljen 1992. godine i od tada je, uz Microsoftovu vlasničku tehnologiju DirectX, postao najčešće korišteno i podržano sučelje za programiranje 2D i 3D aplikacija. Do sad su ga koristile tisuće aplikacija na različitim računalnim platformama.

OpenGL potiče inovacije i ubrzava razvoj aplikacija tako da u sebe uključuje funkcije za mapiranje tekstura, za specijalne efekte te mnoge druge funkcije za vizualizaciju. Osim toga OpenGL se može koristiti na svim popularnim radnim okruženjima osiguravajući široku primjenu aplikacija.

Svaka usklađena implementacija OpenGL-a ima u sebi različite vrste OpenGL funkcija. Dobro definirani OpenGL standard ima jezične veze (engl. *language bindings*) za C, C++, Fortran, Adu i Javu. Uz to, sve licencirane implementacije OpenGL-a potiču iz jedne specifikacije i moraju proći set testova usklađenosti [19]. Testovi usklađenosti su potrebni jer aplikacije koje koriste OpenGL funkcije moraju biti prenosive na druge platforme zbog maksimalne produktivnosti programera i

kraćeg vremena izlaska na tržište.

3.2. Programi za sjenčanje

Programi za sjenčanje su mali programi koji se izvode na GPU-u. Kao što će kasnije biti opisano grafički cjevovod uzima kao ulaz skup 3D koordinata i pretvara ih u obojene piksele na zaslonu. Kako bi to bilo moguće dijelovi grafičkog cjevovoda se paralelno izvode na GPU-u. Procesorske jezgre pokreću programe za sjenčanje na GPU-u za svaki korak cjevovoda kako bi se na kraju mogao dobiti određeni prikaz na zaslonu [20].

Ukratko, programi za sjenčanje su programi koji izmjenjuju ulaze u izlaze, nije im dopušteno međusobno komuniciranje pa im je jedina moguća komunikacija preko njihovih ulaza i izlaza. Programi za sjenčanje se pišu u jezicima za sjenčanje od kojih je najkorišteniji GLSL [21].

3.3. GLSL

OpenGL jezik za sjenčanje (engl. *OpenGL Shading Language*, kraće GLSL) je jezik za pisanje koda za sjenčanje sa sintaksom u stilu C jezika. Programi napisani u njemu imaju glavnu funkciju koja se poziva za svaki objekt. Umjesto da koristi parametre za unos i povratnu vrijednost kao izlaz, GLSL koristi globalne varijable za obradu ulaza i izlaza [22].

Različite vrste koda za sjenčanje uključuju *vertex*, *tessellation*, *geometry*, *fragment* i *compute* kod za sjenčanje koji će biti detaljnije objašnjeni u nastavku. Ispod možemo vidjeti dva jednostavna primjera vertex i fragment koda za sjenčanje.

3.3.1. Primjer vertex koda za sjenčanje

```
uniform mat4 p3d_ModelViewProjectionMatrix;

in vec4 p3d_Vertex;

void main()
{
  gl_Position = p3d_ModelViewProjectionMatrix * p3d_Vertex;
}
```

Uočimo odmah nekoliko razlika u odnosu na programski jezik C. Ključna riječ `uniform` označava da je ova globalna varijabla ista za sve vrhove (engl. *vertexes*). Osim toga ključna riječ `in` označava da je ova globalna varijabla ulazna, a ključna riječ `out` označava da je ova globalna varijabla izlazna. Tipovi podataka `mat4` i `vec4` su ugrađeni tipovi podataka koji implementiraju matricu

veličine 4x4 i vektor veličine 4 elementa te je za njih definirano množenje operatorom * kao množenje matrice i vektora.

3.3.2. Primjer fragment koda za sjenčanje

GLSL fragment kod za sjenčanje koji isječe boju fragmenta u zelenu boju.

```
out vec4 fragColor;

void main() {
  fragColor = vec4(0, 1, 0, 1);
}
```

Važno je za zapamtiti da fragment utječe na najviše jedan piksel na zaslonu, ali na jedan piksel može utjecati mnogo fragmenata [22].

3.3.3. Teksturiranje

Teksture su 2D slike dizajnirane da utječu na izgled 3D modela. Proces teksturanja započinje adresiranjem UV koordinata modela. UV je mehanizam kojim se 2D slika prenosi na 3D model [23].

Teksturiranje uključuje preslikavanje neke boje ili neke druge vrste vektora u fragment pomoću UV koordinata. Koordinate U i V poprimaju vrijednosti od nula do jedan. Svaka točka dobiva UV koordinatu i ona se emitira u *vertex* shaderu [24].

Prilikom izvođenja prikazane teksture, mreža je ravni pravokutnik s istim omjerom kao i zaslon.

UV koordinate se mogu izračunati ukoliko su nam poznate:

1. širina i visina teksture koja je veličine ekrana i koja se preslikava u UV pravokutnik
2. koordinate x i y fragmenta

3.3.4. Svjetlost

Svjetlost se prikazuje izračunavanjem kombinacija ambijentalnog, difuznog, spekularnog i emisijskog svjetla. Pri prikazu svjetla *vertex* kod za sjenčanje treba transformirati i prikazati točku iz prostora pogleda u sjenu ili svjetlosti i tako zasebno za svako svjetlo u sceni dok *fragment* kod za sjenčanje rješava veći dio izračuna svjetlosti.

3.4. Vulkan

Izvorno izveden iz AMD-ovog Mantle API-a te je pod velikim utjecajem Appleovog Metal API-a i Microsoftovog DirectX-a 12. Još nazvan “prijestolonasljednikom” OpenGL-a. Ciljevi Vulkana su [25]:

1. manja kompleksnost upravljačkog-programa i manji utjecaji na performanse u usporedbi s OpenGL-om
2. manje direktnog navođenja korisnika
3. bolje performanse u zadacima koje koriste jednu dretvu (engl. thread) nego OpenGL
4. mogućnost paraleliziranja grafike u više dretvi

Prednosti Vulkana nad OpenGL-om su [26]:

- API nastoji generirati međuspremnike naredbi kroz više dretvi i istovremeno procesiranje njih na cjevovod naredbi. Većina visoko profilnog softvera i većina softvera visokih performansi koji su izgrađeni na OpenGL platformi funkcioniraju na istom principu. Zbog toga programeri ne trebaju održavati programski okvir, a čak i ako trebaju to mogu napraviti bez puno truda.
- Sama aplikacija upravlja dretvama i memorijom bez posredovanja upravljačkog programa. To znači da se ponašanje API-ja može prilagoditi na primjer potrebama igre koja se izvodi.
- U Vulkanu su bolje integrirani alati nego u OpenGL-u zato što se slojevi validacije i dijagnostički slojevi mogu neovisno uključivati.
- Ne postoji velika razlika između mobilne i desktop verzije pa se igre mogu lakše prilagođavati za različite platforme.

Tablica 2: Usporedba OpenGL-a i Vulkan-a [27]

	OpenGL	**Vulkan**
Prvo izdanje	Siječanj 1992.	Veljača 2016.
Trenutna verzija	4.6	1.2.141
Početna motivacija	Pruža standardni način pristupa grafičkom hardveru sa ciljem pisanja 3D softvera	Pristup niskoj razini modernog grafičkog hardvera sa malim utjecajem na performanse (kao poboljšanje u odnosu na OpenGL, koji je izgrađen s hardverom prije 25 godina)
Razina apstrakcije	Vrlo visoka, čineći vrlo jednostavnim brzo razvijanje 3D aplikacije, ali također pružaju moderna proširenja za operacije niske razine performansi	Vrlo niska, što prisiljava programera da uvijek radi sve kako bi omogućio maksimalnu optimizaciju aplikacije
Tehnički najveće razlike	Jedno dretveni stroj stanja sa globalnim stanjem temeljenim na kontekstu, sa provjerama grešaka. Upravljački programi pokušavaju upravljati neispravnim podacima	Nema stanja, slanje naredbi putem upita za asinhronizaciju, precizna kontrola nad hardverom (uključujući upravljanje memorijom)

3.5. Programiranje Vulkan aplikacija

3.5.1. Potrebni paketi na operativnom sustavu Arch Linux

3.5.1.1. Koristeći grafičku karticu tvrtke NVIDIA

- vulkan-icd-loader
- nvidia-utils
- vulkan-headers
- vulkan-tools
- vulkan-validation-layers
- glslang
- glm

Prilikom instalacije se mogu javiti problemi s prepoznavanjem Vulkan upravljačkog programa koji su najčešće uzokovani konfliktima. Problem uzrokovan konfliktima može se dijagnosticirati tako da se provjeri je li na sustavu instaliran paket `vulkan-intel`. Taj se problem najčešće javlja kada postoje dvije grafičke kartice (npr. na laptopu integrirana i namjenska engl. *dedicated* kartica). Više informacija mogu se naći na web stranici Archlinux.org [1].

3.5.1.2. Koristeći grafičku karticu tvrtke AMD

- vulkan-icd-loader
- vulkan-headers
- vulkan-tools
- vulkan-validation-layers
- glslang
- glm
- mesa - uz nju se automatski instalira vulkan-radeon, no ako to nije slučaj potrebna je zasebna instalacija vulkan-radeon

1 Poveznica za više informacija: https://wiki.archlinux.org/index.php/Vulkan

3.5.2. Pisanje koda koristeći Vulkan API

Glavna pravila i informacije za pisanje koda koristeći Vulkan API su:

- Svaki Vulkan objekt koji se napravi treba biti eksplicitno uništen kada ga se više ne koristi. Sam Vulkan to ne radi.
- Vulkanovi objekti su ili kreirani izravno s funkcijama poput `vkCreateXXX`, ili su dodijeljeni kroz drugi objekt s funkcijama poput `vkAllocateXXX`.
- Sve funkcije dijele parametar pAllocator - dodatni parametar koji nam omogućuje da odredimo povratne pozive (engl. *callbacks*) za prilagođeni razdjelnik memorije (engl. *memory allocator*).

Vulkan sam po sebi ne zahtjeva prozor za izvođenje. Ukoliko želimo prikaz prozora potrebno je koristiti neki od pružatelja prozornih usluga poput *Graphics Library Framework* (kraće GLFW). U tom slučaju definiramo i uključujemo slijedeće [28]:

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
```

U slučaju da nam nije potreban prozor (tzv. headless način rada) možemo preskočiti ovaj korak i umjesto toga uključiti `#include <vulkan/vulkan.h>`.

3.5.3. GLFW

GLFW je biblioteka otvorenog koda za razvoj OpenGL, OpenGL ES i Vulkan aplikacija. Pruža jednostavno aplikacijsko programsko sučelje (engl. *application programming interface*, kraće API) za stvaranje prozora, konteksta i površina, čitanje unosa, rukovanje događajima itd.

GLFW izvorno podržava Windows, macOS i Linux te ostale sustave slične Unixu. Na Linuxu su podržani X11 i Wayland, oni su komunikacijski protokoli koji specificiraju komunikaciju između poslužitelja zaslona i njegovih klijenata (engl. display server protocol).

Jedan od boljih vodiča o povezivanju GLFW-a sa Vulkanom i stvaranjem prozora za daljnji rad je web stranica GLFW[2] koja ima jednostavne i kratke članke koji objašnjavaju dijelove koda te kako ih koristiti [29].

GLFW je izvorno dizajniran za stvaranje OpenGL konteksta. Pri pisanju Vulkan koda moramo eksplicitno napisati da ne stvara OpenGL kontekst s naknadnim pozivom `glfwWindowHint (GLFW_CLIENT_API, GLFW_NO_API)`. Tim potezom onemogućujemo tzv. Creation kontekst

2 Poveznica: www.glfw.org

tako da ne možemo koristiti OpenGL ili OpenGL ES s istim prozorom koji koristimo i za Vulkan.

3.5.4. Stvaranje instance

Instanca je veza između naše aplikacije i biblioteke Vulkan. Njeno stvaranje uključuje navođenje nekih detalja o vašoj aplikaciji i upravljačkom programu. Ti detalji prikazani su u funkciji ispod.

```
void createInstance() {
    VkApplicationInfo appInfo{};
    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.pApplicationName = "Hello Triangle";
    appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.pEngineName = "No Engine";
    appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.apiVersion = VK_API_VERSION_1_0;
}
```

Kao prvo možemo primijetiti kako smo naveli za sType `VK_STRUCTURE_TYPE_APPLICATION_INFO`. On nam prikazuje koji je tip appInfo, no pošto smo ispred te linije napisali `VkApplicationInfo appInfo{};` automatski se dodjeljuje naziv tipa `APPLICATION_INFO`. Ipak da bi bilo jasnije može se i dodatno napisati kako ne bi došlo do zabune (kao što je napisano u kodu iznad). Nakon toga napisano je ime aplikacije i verzija aplikacije na kojoj se radi (sam korisnik odabire broj verzije, te izmjenjuje ju ovisno o tome koliko je toga ažurirao u aplikaciji). Verzija aplikacije se može zapisati i samo brojem ali koristeći `VK_MAKE_VERSION(1, 0, 0);` kreira se verzija aplikacijskog programskog sučelja čiji prvi broj prikazuje znatno veliku promjenu u aplikaciji, srednji broj se mijenja pri ažuriranju, a najzadnji broj se inače mijenja kad su se nadodale neke zakrpe u aplikaciji [30].

Pri kreiranju instance može se pojaviti greška `VK_ERROR_EXTENSION_NOT_PRESENT`. Greška upozorava programera kako je došlo do greške sa dodacima (engl. extension) te bi tada bilo dobro pregledati sve dodatke i nadodati one koje nedostaju. Osim toga, važno je uništiti instancu sa `vkDestroyInstance(instance, nullptr);` na kraju koda, jer kao što je prije navedeno instance se ne uništavaju same [31].

3.5.5. Slojevi za validaciju

U zadanim postavkama API-ja provjeravanje grešaka je vrlo ograničeno. Pri pokretanju aplikacije koja ima grešku, aplikacija će samo prestati raditi bez ikakvog objašnjenja ili ispisivanja gdje se točno greška nalazi i zbog kojeg razloga. Zbog tog razloga potrebni su nam slojevi za validaciju. Slojevi za validaciju (engl. *validation layers*) su neobavezne komponente koje povezuju sa Vulkan

pozivima funkcija i pružaju dodatne operacije [32].

Uobičajene operacije u slojevima za validaciju su:

1. Provjera vrijednosti parametara u odnosu na specifikacije za otkrivanje zlouporabe
2. Praćenje stvaranja i uništavanja objekata radi pronalaženja curenja resursa (engl. *resource leak*)
3. Provjera sigurnosti dretve (engl. *thread safety*) praćenjem dretvi iz kojih potječu pozivi
4. Bilježenje svakog poziva i njegovih parametara na standardni izlaz (engl. *standard output*)
5. Praćenje Vulkan poziva za profiliranje i ponovno izvođenje

Osim toga iako Vulkan ne sadrži ugrađene slojeve za provjeru valjanosti LunarG Vulkan SDK (engl. *Software development kit*) pruža skup slojeva koji provjeravaju uobičajene pogreške. Oni su ujedno i potpuno otvoreni kod.

U Vulkanu su postojala dva različita tipa validacijskih slojeva:

1. specifično za određenu instancu
2. specifično za određen uređaj

Slojevi specifično za određenu instancu (engl. *instance layers*) provjeravali bi samo pozive koji se odnose na globalne Vulkan objekte, a slojevi specifično za određen uređaj provjeravali bi samo pozive koji se odnose na određeni GPU. Slojevi specifični za uređaj su zastarjeli, što znači da se slojevi za određenu instancu primjenjuju na sve Vulkan pozive.

3.5.6. Fizički uređaji i skupine redova

3.5.6.1. Fizički uređaji

Nakon izrade instance sljedeći je korak traženje grafičke kartice u sustavu i odabir iste, ta kartica treba podržavati značajke koje su potrebne.

Popis grafičkih kartica koje imaju podršku za Vulkan:

- NVidiju se nalazi na stranici developer.nvidia.com[3]
- AMD se nalazi na stranici amd.com[4]

Pri pisanju koda nakon odabira grafičke kartice ta se odabrana grafička kartica onda sprema u `VkPhysicalDevice` koja je dodana kao novi član klase [33]. Osnovna svojstva uređaja poput

3 Poveznica na NVidia podršku za Vulkan: https://developer.nvidia.com/vulkan-driver
4 Poveznica na AMD podršku za Vulkan: https://www.amd.com/en/technologies/vulkan

imena, vrste i podržane Vulkanove verzije mogu se zatražiti pomoću `vkGetPhysicalDeviceProperties`.

3.5.6.2. Skupine redova

Sve, od crtanja do prijenosa tekstura, zahtijeva da se naredbe pošalju u redovima. Postoje različite vrste čekanja koje potječu iz različitih skupina redova i svaka skupina redova omogućuje samo podskup naredbi.

Moraju se provjeriti koje skupine redova podržavaju uređaj i koja od tih podržava naredbe koje želimo koristiti. U tu svrhu u kod se dodaje nova funkcija (`findQueueFamilies`) koja traži sve vrste čekanja koje nam potrebne [33].

3.5.7. Logički uređaji i redovi

3.5.7.1. Logički uređaji

Nakon odabira fizičkog uređaja koji će se koristiti, mora se postaviti logički uređaj koji će biti sučelje za fizički uređaj.

3.5.7.2. Redovi

Trenutno dostupni upravljački programi omogućuju stvaranje malog broja reda, ali obično nam treba samo jedan. To je zato što se mogu stvoriti komande među spremnicima na više dretvi (engl. *thread*) i zatim ih istovremeno poslati na glavnu dretvu s maksimalnom optimizacijom aplikacije.

Osim toga redovi se automatski stvaraju zajedno s logičkim uređajem te se redovi uređaja implicitno čiste kada je uređaj uništen.
Funkciju `vkGetDeviceQueue` možemo koristiti za dohvaćanje reda čekanja za svaku skupinu redova [34].

3.5.8. Površina prozora i swap chain

3.5.8.1. Prozor

Da bi se uspostavila veza između Vulkana i prozorskog sustava za prezentiranje rezultata na zaslonu, moramo koristiti integraciju prozora sustava (engl. *Window System Integration* - WSI) proširenje. Podloga u programu bit će potpomognuta prozorom koji se već otvorio sa GLFW-om.

Površina prozora je potpuno neobavezna komponenta u Vulkanu. Vulkan nam omogućuje to bez ikakvih dodataka u kodu poput stvaranja nevidljivog prozora (koji je potreban za OpenGL).

No ako se želi napraviti prozor važno je znati da je potrebno površinu prozora stvoriti odmah nakon stvaranja instance, jer može utjecati na fizički odabir uređaja. Iako Vulkanova implementacija može podržavati integraciju prozora sustava (engl. *window system integration*), to ne znači da ga podržava svaki uređaj u sustavu. Zbog toga se u kodu mora dodati `isDeviceSuitable` funkcija koja provjerava da li uređaj u sustavu podržava integraciju prozora sustava. Tako se osigurava da uređaj može prikazati slike na površini koja se stvara u kodu [35].

Osim toga moguće je i da se skupovi redova koji podržavaju naredbe za crtanje i oni koji podržavaju prezentaciju ne preklapaju. Stoga moramo uzeti u obzir da bi mogao postojati poseban red prezentacije.

3.5.8.2. Swap chain

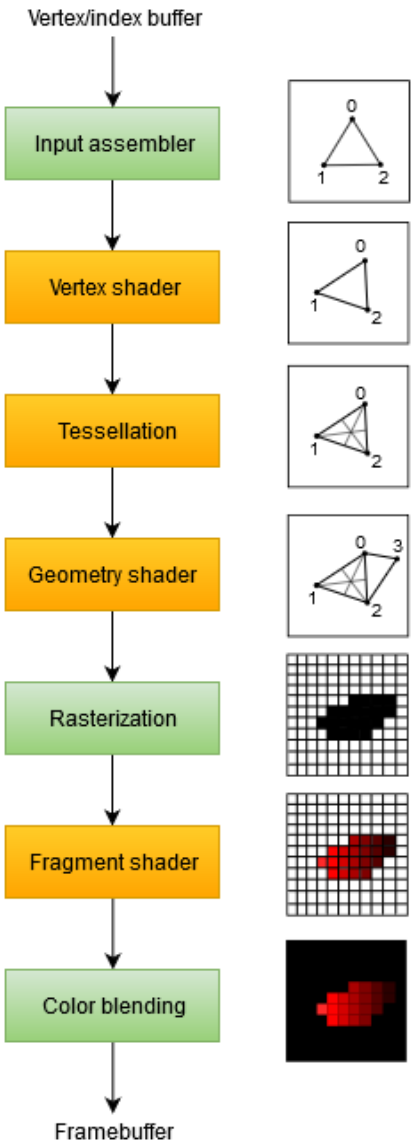
Swap chain je niz međuspremnika ili mjesta u računalnoj memoriji koji se koristi za generiranje slike prije nego što se prikaže na monitoru ili drugom uređaju [36]. U njemu uvijek postoje najmanje dva međuspremnika jer se jedan (aktivan međuspremnik) prepoznaje kao onaj koji je trenutno prikazan na zaslonu. Dok se prikazuje jedan međuspremnik, sljedeći se međuspremnik može pripremiti za prikaz, a zatim prema potrebi zamijeniti sa onim koji se prikazuje [37].

3.5.9. Prikazi slika

“Prikazi” (engl. *view*) su dodatne informacije u resursima koji opisuju kako se taj resurs koristi. Područje memorije koja pripada slici može se organizirati na različite načine, ovisno o uporabi slike. Također je korisno opisati format slike (npr. `VK_FORMAT_R8G8B8A8_UNORM`), redoslijed komponenata i informacije o slojevima. Sve ove informacije su metapodaci slike sadržani u `VkImageView [37].`

3.5.10. Osnove grafičkog cjevovoda

Grafički cjevovod (engl. *pipeline*) uzima kao ulaz skup 3D koordinata i pretvara ih u obojene piksele na zaslonu. Kako bi se grafički cjevovod bolje objasnio slika 9 prikazuje pregled grafičkog cjevovoda za trokut [38]:



Slika 9: Pregled grafičkog cjevovoda za trokut, preuzeto iz: https://vulkan-tutorial.com/images/vulkan_simplified_pipeline.svg

- alat za prikupljanje ulaza (engl. *input assembler*) prikuplja određene neobrađene vertex podatke iz međuspremnika te može također koristiti kao indeksni međuspremnik za ponavljanje određenih elemenata bez potrebe za dupliciranjem samih podataka verteksa
- vertex program za sjenčanje (engl. *vertex shader*) se pokreće za svaki vrh i primjenjuje transformacije kako bi se položaji vrhova trokuta pretvorili iz prostora modela u prostor zaslona

- *tessellation shaders* omogućuju da se podijeli geometrija na temelju određenih pravila kako bi se povećala kvaliteta mreže (ovo se često koristi da površine poput zidova i stubišta izgledaju manje ravne kada su u blizini)
- geometrijski program za sjenčanje (engl. *geometry shader*) se pokreće na svakom primitivnom obliku (trokutu, liniji, točki) i može odbaciti ili proizvesti više primitivnih oblika nego što je bilo. To je slično kao tessellation shader, ali mnogo fleksibilniji.
- faza rasterizacije razlučuje primitivne oblike u fragmente
- fragment program za sjenčanje (engl. *fragment shader*) se poziva za svaki fragment koji "preživi" i određuje u koji su međuspremnik fragmenti napisani i s kojim vrijednostima boja i dubine
- Faza miješanja boja (engl. *color blending*) primjenjuje operacije miješanja različitih fragmenata koji preslikavaju na isti piksel u međuspremnika okvira (fragmenti se mogu jednostavno prebrisati, nadopuniti ili miješati na temelju transparentnosti)

Faze sa zelenom bojom na slici poznate su kao faze sa fiksnom funkcijom. Ove faze omogućuju podešavanje njihovih operacija pomoću parametara, ali način njihovog rada je unaprijed definiran.

Faze s narančastom bojom na slici s druge strane mogu se programirati, što znači da se može prenijeti vlastiti kod na grafičku karticu da bi se primijenile točne operacije koje su potrebne.

3.6. SPIR-V

Kod za sjenčanje u Vulkanu mora biti naveden u formatu bajt kodova (engl. bytecode), za razliku od sintaksi koja je razumljiva ljudima kao što su GLSL i HLSL. Taj se format bajt kodova naziva SPIR-V i dizajniran je za upotrebu s Vulkanom i OpenCL (oba Khronos API-ja).

Standard Portable Intermediate Representation (SPIR) je prvotno razvijen za upotrebu od strane OpenCL-a, a verzije SPIR-a 1.2 i 2.0 zasnovane su na LLVM-u. SPIR se nakon toga razvio u jezik koji je neovisan o API-aju. Njega je grupa Khronos u potpunosti definirala s izvornom podrškom za sjenčanje (engl. *native support for shaders*) koje koriste API-ji poput Vulkana - zvani SPIR-V [39].

Prednost upotrebe bajt kod formata je u tome što su program-prevoditelji napisani od strane dobavljača GPU-a pa se kod za sjenčanje lakše pretvori u izvorni kod.

Prošlost je pokazala da su s lako čitljivom sintaksom poput GLSL-a, neki dobavljači GPU-a bili prilično fleksibilni u svojoj interpretaciji standarda. Zbog toga je potrebno jako paziti pri pisanju koda za sjenčanje. Ako se piše kod na GPU-u koji ima jako fleksibilnu interpretaciju standarda

može doći do toga da će taj isti kod imati greške ako se izvodi na GPU-u od drugog dobavljača zbog sintaktičkih grešaka ili pogreška prevoditelja. To se može izbjeći ako se koristi jednostavni format bajt kodova kao što je SPIR-V.

Osim toga dobro je znati da je Khronos grupa objavila svoj vlastiti prevoditelj neovisan o dobavljaču koji prevodi GLSL u SPIR-V. Ovaj prevodilac dizajniran je da provjerava je li kod za sjenčanje u potpunosti usklađen sa standardima te proizvodi jednu binarnu SPIR-V datoteku koja se može isporučiti zajedno sa programom [40].

3.7. Standardi za AR/VR

Osim OpenGL-a i OpenCL-a postoji OpenXR koji se koristi pri radu sa virtualnom stvarnošću (VR), proširenom stvarnošću (AR) i mješovitom stvarnošću (MR).

OpenXR je API za XR aplikacije. XR se odnosi na kontinuum stvarnog i virtualnog kombiniranog okruženja koje generiraju računala čovjek-stroj interakcijom i uključuje tehnologije povezane s VR, AR i MR [41].

Aplikacijskom programeru OpenXR je skup funkcija koje se međusobno povezuju tijekom izvođenja radi izvođenja uobičajenih operacija, poput pristupa kontroloru / perifernom stanju, dobivanja trenutnih i/ili predviđenih pozicija praćenja te radi slanja prikazanih okvira [41].

4. Stog upravljačkih programa za AMD-ove grafičke procesore na Linuxu i projekt GPUOpen

U ovom poglavlju biti će kratki opis generiranja grafike na Linux sustavu. Također će ukratko biti opisani: DRM, Mesa, pozadina program-prevoditelja AMDGPU kao evolucija pozadine R600 u LLVM-u, OpenGL upravljački program RadeonSI, Vulkan upravljački program radv i program-prevoditelj za sjenčanje ACO.

4.1. Ubrzanje prikaza 3D grafike na modernim grafičkim procesorima

Moderne grafičke kartice u osnovi su koprocesori tj. izvršavaju rasterećenje rada na osnovnom procesoru. Općeniti postupak prikazivanja grafike je:

- Dodjeljuje se međuspremnik u koji ćete upisati izlaz
- Dodjeljuje se jedan ili više međuspremnika i napuni ih se ulaznim parametrima, npr. n-torka od (x, y, z, boja)
- Dodjeljuje se jedan ili više međuspremnika i napuni ih se CU uputama za izvršenje računarskih jedinica GPU-a
- Napuni se međuspremnik toka naredbi s naredbama tipa *Card Settings* (kraće CS) koje sadrže upute za konfiguriranje GPU-a
- Nakon svega se prosljeđuju međuspremnici u GPU na daljnju obradu

4.1.1. Međuspremnici okvira

Sve grafičke kartice generiraju međuspremnik okvira tj. blok memorije s 3 bajta po pikselu, kod kojeg je svaki piksel RGB boja za prikaz na zaslonu. Jedan međuspremnik okvira poznat grafičkoj kartici je “međuspremnik za skeniranje” tj. međuspremnik koji će se prikazati [42].

4.2. DRM

Direct Rendering Manager (kraće DRM) je upravljački program grafičkog uređaja unutar jezgre (engl. *kernel-level*) koji se učitava u jezgri Linuxa [43].

DRM podržava infrastrukturu izravnog prikazivanja (DRI) na tri načina:

1. DRM omogućuje sinkronizirani pristup grafičkom hardveru
   Sustav izravnog prikazivanja (engl. *direct rendering system*) ima više entiteta (ti entiteti su: poslužitelji, više klijenata izravnog prikazivanja i jezgra). Oni se natječu za izravan pristup grafičkom hardveru. No hardver koji je trenutno dostupan može biti zaključan u slučaju kada na njega želi pristupiti više entiteta. DRM omogućuje pojedinačno zaključavanje hardvera po uređaju za sinkronizaciju pristupa hardveru. Takvo hardversko zaključavanje može biti potrebno kada X poslužitelj izvodi 2D prikazivanje, a jezgra šalje izravni pristup memoriji (engl. *Direct Memory Access*, kraće DMA) međuspremnicima (engl. *buffers*).
2. DRM provodi DRI sigurnosnu politiku za pristup grafičkom hardveru
   X poslužitelj, pokrenut s pravima korijenskog korisnika (engl. *root*), obično dobiva pristup međuspremniku okvira i memorijsko preslikanom ulazu/izlazu (engl. *Memory-mapped I/O*, kraće MMIO) regijama na grafičkom hardveru mapiranjem tih regija pomoću /dev /mem [44]. Klijenti izravnog prikazivanja, međutim, ne rade kao *root*, ali još uvijek zahtijevaju slična mapiranja. Kao i /dev /mem, sučelje DRM uređaja omogućuje klijentima stvaranje ovih preslikavanja, ali uz dodatna ograničenja.
3. DRM pruža generički DMA mehanizam
   Većina modernog računalnog grafičkog hardvera omogućuje DMA pristup naredbi “prvi unutra prvi van” (engl. *First In First Out*, kraće FIFO) [45]. DMA pristup često je optimiziran tako da pruža znatno bolju propusnost od MMIO pristupa. Za ove kartice DRM pruža generički DMA mehanizam sa zasebnim značajkama.
4. DRM je proširiv
   DMA mehanizam je proširiv upotrebom modula jezgre specifičnog za uređaj (engl. *device-specific kernel module*) koji može spojiti neke ili sve generičke funkcije.

4.2.1. Modul jezgre

Moduli su dijelovi koda koji se na zahtjev mogu učitati u jezgru. Proširuju funkcionalnost jezgre bez potrebe za ponovnim podizanjem sustava. Na primjer, jedna vrsta modula je upravljački program uređaja koji jezgri omogućuje pristup hardveru povezanom sa sustavom. Bez modula, morao bi se koristiti monolitna jezgra i dodavati nove funkcije izravno u sliku jezgre [46].

Ako se gleda samo AMD-ova podrška za grafičke procesore, može se primijetiti kako su se paralelno razvijali upravljački programi otvorenog i zatvorenog koda. Od 2015. nadalje događa se zaokret i AMD stvara ujedinjeni upravljački program otvorenog koda u jezgri Linuxa. On je stvoren nadogradnjom postojećeg upravljačkog programa za grafičke procesore AMD Radeon u jezgri Linuxa tako da se na njega može vezati novi upravljački program AMDGPU koji radi u korisničkom prostoru i koji je kombinacija komponenata otvorenog i zatvorenog koda [47].

4.3. Mesa

4.3.1. Povijest Mese

Projekt Mesa započet je sad već davne 1993. godine od strane jednog čovjeka, Briana Paula koji je zbog svojih zasluga nazvan “ocem Mese” (engl. *the father of Mesa*). Brian Paul ističe da je počeo programirati Mesu još 1992. godine te se nada da će kad-tad pronaći dokaz za to na starim Amiginim disketama (engl. *Amiga floppies*) s oznakama vremena (engl. *timestamps*) kako bi promijenio službeni datum početka izrade Mese [48].

Na početku razvoja radio je na Amigi kod kuće i na Unix sustavima na poslu, koristeći diskete za prijenos koda od kuće do posla. Razvio je sučelje upravljačkih programa (engl. *device driver interface*) za uređaj kako bi se prilagodila dva različita crtaća sučelja (engl. *drawing interfaces*) (Amiga i Xlib) [49].

S tako odlučnim stavom i puno truda imao je poprilično cjelovitu OpenGL implementaciju u studenom 1994. godine. Bilo je potrebno još rada na tome, ali je već počeo koristiti svoju implementaciju na poslu te je došao do zaključka kako bi njegova implementacija mogla pomoći mnogim ljudima te je odlučio imati otvoreni izvorni kod. Prije toga morao je razgovarati sa Silicon Graphics, Inc. (kraće SGI) koji su počeli razvijati OpenGL još 1991. godine te su imala sva pridržana prava. Nakon nekoliko razgovora sa SGI došli su do zaključka kako Paul može otvoriti svoj kod pod uvjetom da ne koristi ime OpenGL. Tako se “rodila” Mesa [50].

U ranim je danima projekt naišao na brojne probleme. Jedan od glavnijih je bio taj da je bilo teško podržati više različitih sustava (IRIX, SunOS, HPUX, …). Taj problem se i dan danas može javiti. Osim toga prikazi slike su bili problematični. 24-bitni su zasloni tada bili skupi, tako da je većina zaslona bila 8 ili 16-bitna. To je značilo da se Mesa trebala baviti sa više mapa boja (engl. *color maps*) i usitnjavanjem (engl. *dithering*, proces primjene šuma za nasumično razbacivanje grešaka kvantizacije).

Osnivač Paul je rekao da nikada nije očekivao da će Mesa biti tako uspješna, niti da će trajati toliko dugo. Pretpostavio je da će dobavljači stavljati vlastite biblioteke OpenGL-a te da će tada Mesa biti nepotrebna. No, ljudima i dalje treba softver za prikaz, koristeći *llvmpipe* i druge upravljačke programe [49].

Tijekom godina bilo je mnogo doprinosa Mesi. Projekt je narastao u veličini i složenosti iznad očekivanja. Mesa je u svim distribucijama Linuxa i svakodnevno je koristi na milijune korisnika.

4.4. LLVM

LLVM je započeo kao istraživački projekt na Sveučilištu u Illinoisu pod vodstvom Vikrama Advea i Chrisa Lattnera 2000. godine. Cilj projekta bio je pružanje modernog, oblika statičkog jednostrukog dodjeljivanja vrijednosti varijabli (engl. *Static single assignment form*, kraće SSA) strategije prevođenja koja može podržati statičko i dinamičko prevođenje proizvoljnih programskih jezika. Objavljen je pod licencom Sveučilišta Illinois/NCSA s otvorenim kodom i dozvolom za slobodan softver [51].

Godine 2005. tvrtka Apple angažirala je Chrisa Lattnera i formirala tim za rad na LLVM-u za buduće različite namjene u Appleovim razvojnim alatima [52]. Sada je LLVM sastavni dio Appleovih najnovijih razvojnih alata za macOS i iOS. Osim Applea, od 2013. godine Sony koristi LLVM-ov primarni prednji program-prevoditelj (engl. *compiler frontend*) Clang u kompletu za razvoj softvera (engl. *software development kit*, kraće SDK) za konzolu PlayStation 4 [53].

Osim toga LLVM je prerastao u širok projekt koji se sastoji od niza podprojekata. Mnogi se od tih podprojekata koriste u proizvodnji različitih komercijalnih projekata i projekata otvorenog koda (jedan od tih podprojekata koji se prije naveden, a koristi ga Sony, je Clang). Osim toga ti podprojekti se često koriste u akademskim istraživanjima [54].

Najznačajniji podprojekti LLVM-a su:

1. Clang
2. LLDB
3. libc++
4. compiler-rt
5. MLIR
6. OpenMP
7. Polly
8. libclc
9. KLEE
10. LLD

4.4.1. Clang

Clang (*C language family frontend for LLVM*) je prednji program-prevoditelj za programske jezike C, C ++ i Objective-C. Kao pozadinu (engl. *backend*) koristi virtualni stroj niske razine (LLVM). Ima za cilj dostaviti brže prevođenje, korisne poruke o pogreškama, upozorenjima te pruža platformu za izgradnju alata. Koristi se zbog bržih performansi od GCC-a (*GNU Compiler Collection-a*) [55], [56].

4.4.2. Promjene u GCC-u

Suočavajući se s konkurentskim pritiskom LLVM-a i Clanga, zajednica okupljena oko GNU Compiler Collectiona (kraće GCC) aktivno je izvršila mnoge prilagodbe, kao što su: ubrzanje prevođenja, poboljšanje prikaza grešaka pri prevođenju te mnoge druge promijene [57]. Možemo očekivati da će konkurencija između dviju tehnologija prevođenja i dalje pružati programerima bolje program-prevoditelje.

4.5. Pozadina program-prevoditelja R600

Pozadina program-prevoditelja R600 je komponenta AMD-ovih Open Source GPU upravljačkih programa koja se koristi za prevođenje GLSL i OpenCL programa. Komponenta podržava sve AMD-ove generacije GPU-a. Osim toga poznata je i po tome što je nastala iz suradnje između AMD-a i zajednice otvorenog koda (engl. *Open-Source community*). Ime R600 je dobila zbog toga što AMD inače svoje komponente otvorenog koda naziva prema prvoj generaciji koju podržavaju [58].

4.5.1. Pozadina AMDGPU kao evolucija R600

Pozadina AMDGPU omogućuje generiranje skupa instrukcija arhitekture (engl. *instruction set architecture*, kraće ISA) koda za AMD GPU-e počevši još od prve R600 pozadine sve do današnjeg GCN-a [59], [60].

Sama AMDGPU pozadina ima ciljne značajke koje kontroliraju način generiranja koda. Svi procesori ne podržavaju sve ciljane značajke nego samo neke specifične, a one koje podržava koji AMD GPU mogu se pogledati na LLVM-ovim stranicama s dokumentacijom[5]. *Runtime* mora osigurati da se značajke koje su podržane i koje se koriste za izvođenje koda podudaraju sa značajkama omogućenim prilikom generiranja koda. Ako dođe do neusklađenosti značajki mogu se dogoditi pogrešno izvršavanje koda ili smanjenje performansi [61].

4.6. Video

Potrebno je spomenuti kako GPU za prikaz i kodiranje videa koristi drugi dio hardvera nego za prikaz 3D grafike i računanje na grafičkim procesorima te njime upravlja sasvim odvojeni dio upravljačkog programa. U ovom diplomskom radu se njima nećemo baviti te neće biti detaljno opisani budući da je fokus ovog diplomskog rada na prikazu 3D grafike i povezanim izračunima na GPU-u te promjenama u upravljačkom programu s ciljem poboljšanja performansi.

4.7. OpenGL upravljački program RadeonSI

RadeonSI su službeno razvili programeri AMD otvorenog koda uz podršku zajednice i mogu se pronaći u novijim verzijama Mese [62]. 2019. godine u Mesa verziji 20.0. dodana je podrška za

5 Poveznica na stranicu s LLVM dokumentacijom: https://llvm.org/docs/AMDGPUUsage.html#amdgpu-processor-table

RadeonSI koja omogućuje da Linux upravljački program koristi OpenGL 4.6 [63]. Podrška za RadeonSI zahtijeva upotrebu NIR-a zbog dijeljenja koda s RADV-om (zbog podrške za unošenje SPIR-V i ponovne upotrebe postojećih putova koda). NIR je novo unutarnje predstavljanje (engl. *internal representation*, kraće IR) za Mesa program-prevoditelj za sjenčanje. Primarna svrha NIR-a je da bude učinkovitiji u optimizaciji i generiranju boljeg koda za back-endove [64].

4.8. Vulkan upravljački program radv

Upravljački program radv integriran je u Mesu kao početna podrška za Radeon Vulkan. Ovaj upravljački program izrađen je u suradnji sa AMD zajednicom otvorenog koda. Prvi put je bio integriran sa Mesa verzijom 12.1/13.0 [65]. Osoba koja je objavila vijest o ovom upravljačkom programu i koji je sudjelovao u njegovoj izradi je Dave Airlie [66]. Osim što je radio na radv-u on je sudjelovao u projektu X.org, Mesa projektu te je pisao različite upravljačke programe korisničkog prostora za razne GPU-ove tijekom godina.

4.9. ACO

Valve je 2019. godine objavio da samostalno razvija vlastiti program-prevoditelj za sjenčanje s otvorenim kodom (nazvan “ACO”) za Linux koji će djelovati kao alternativa postojećem LLVM program-prevoditelju [67]. Upravo se ACO može pronaći u GCN i RDNA GPU-ovima. Ovaj novi program-prevoditelj koda za sjenčanje rezultirao je povećanjem brzine kadra od 44% na određenim AMD-ovim grafičkim karticama, a u jednoj određenoj igri koja koristi DirectX čak je i bolji od Windowsa iako igra na Linuxu nije nativno pokretljiva [16]. U zadnjem poglavlju ovog rada prikazuju se primjeri igara sa ACO-om te rezultati koji su dobiveni.

5. OpenGL i GLSL

5.1. Apitrace

Apitrace je skup alata za ispravljanje pogrešaka OpenGL aplikacija i upravljačkih programa. Uključuje alat za generiranje traga svih OpenGL poziva koje aplikacija izvodi i alat za ponovnu reprodukciju tih tragova i inspekciju prikaza i stanja OpenGL-a tijekom izvršavanja programa. To ga čini korisnim za prepoznavanje izvora grafičkih grešaka u OpenGL aplikacijama [68].

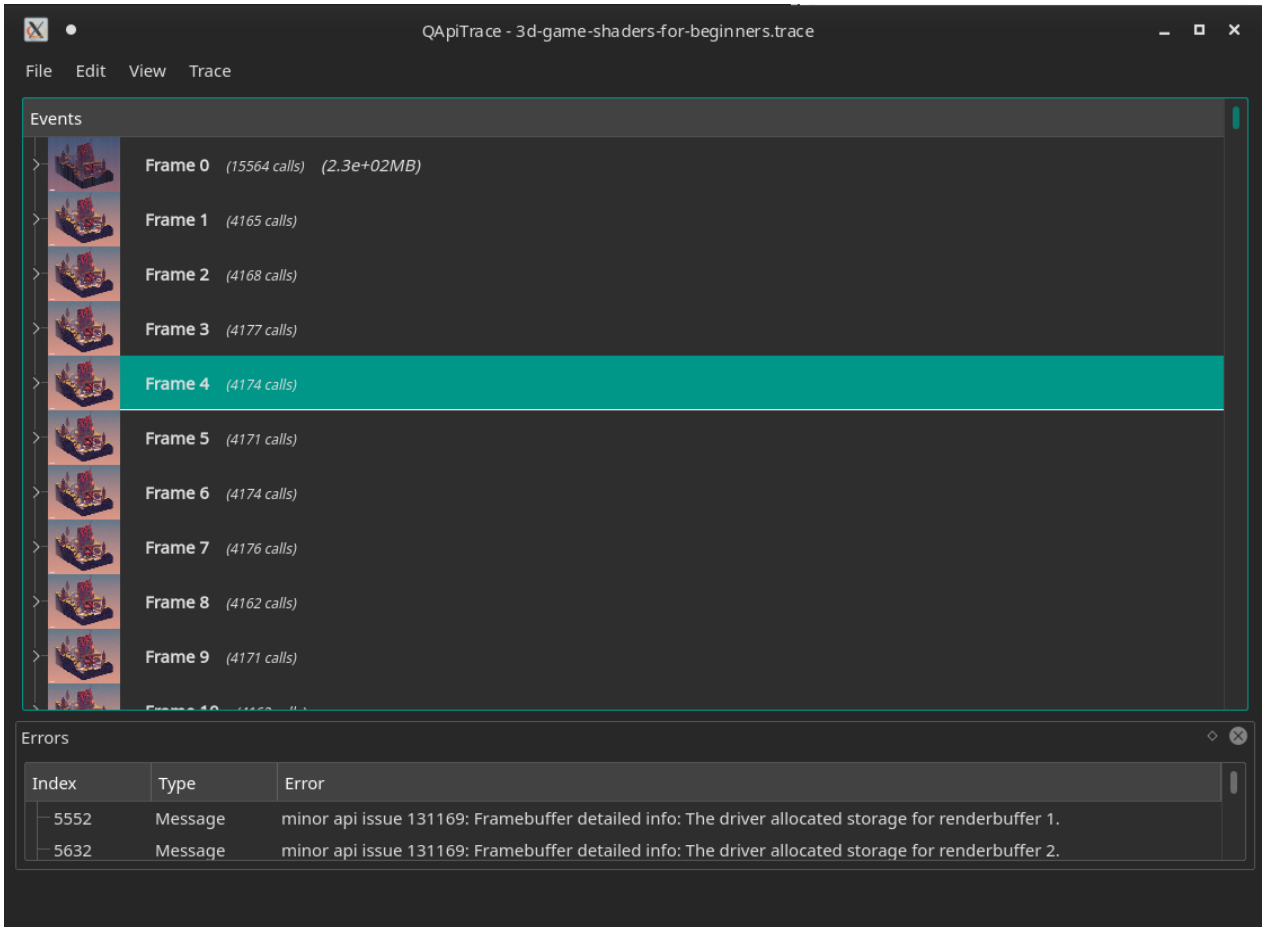
Za pokretanja Apitrace-a na nekoj aplikaciji koja je instalirana ili izrađena na računalu potrebno je otvoriti terminal te upisati naredbu

```
apitrace trace --api gl ./ime-aplikacije
```

Pri tome moramo obratiti pažnju na gl atribut koji predstavlja OpenGL i koristi se za OpenGL aplikacije. Nakon pokretanja Apitrace-a i izlaska iz aplikacije koju smo promatrali možemo primijetiti kako je nastalo novi dokument u mapi gdje se nalazi aplikacija. Taj dokument ima ekstenziju .trace i u njemu su ispisani rezultati Apitrace-a. Kako bi se otvorilo Apitrace grafičko korisničko sučelje (engl. *graphical user interface* - GUI) potrebno je u terminalu napisati i pokrenuti naredbu

```
qapitrace ime-aplikacije.trace
```

Izgled grafičkog korisničkog sučelja Apitrace-a nazvanog QApiTrace može se vidjeti na slici 10.



Slika 10: Izgled grafičkog sučelja Apitrace-a

U slučaju da grafičko korisničko sučelje nije potrebno ili nije inicijalizirano moguće je i samo ispisati informacije prikupljene Apitrace-om preko terminala pokretanjem naredbe

```
apitrace dump ime-aplikacije.trace
```

5.1.1. Glavne značajke Apitrace-a

GPU i CPU profiliranje s alatom Apitrace može se izvršiti korištenjem slijedećih parametara:

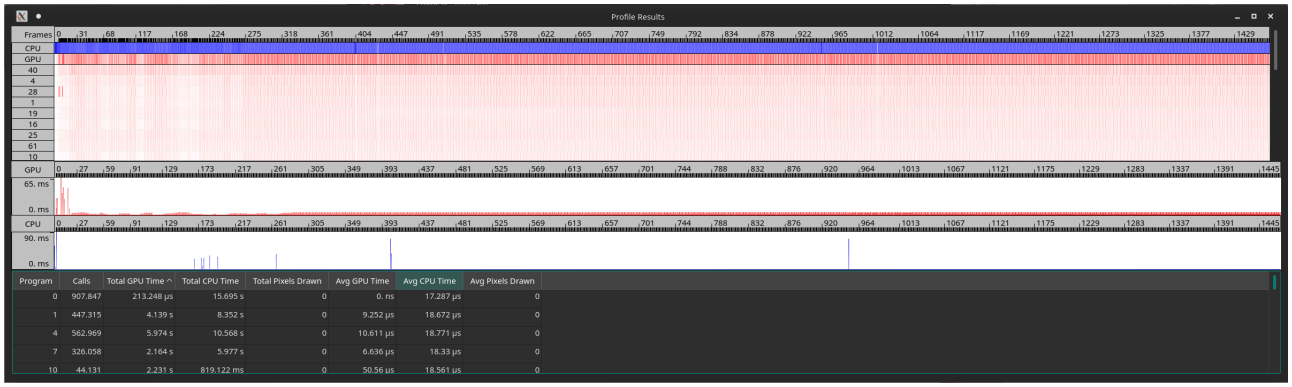
- `--pgpu` bilježi GPU vremenske okvire (engl. time frames) i zapisuje pozive
- `--pcpu` bilježi CPU vremenske okvire (engl. time frames) i zapisuje pozive
- `--ppd` snima nacrtane piksele za svaki poziv

Rezultati koji proizlaze iz ovih značajki se mogu onda pročitati ručno ili ih je moguće analizirati pomoću Python skripte koja dolazi uz Apitrace. Skripte za Apitrace se mogu naći u mapi ‘scripts’ na mjestu gdje je Apitrace instaliran. Za glavne značajke koje su prije navedene potrebna je skripta profileshader.py koja će pročitati rezultate te ih pretvoriti u formatiranu tablicu. Tablica je formatirana tako da prikazuje rezultate za svaki kod za sjenčanje zasebno. Na primjer, naredba za

analizu svih gornjih značajki putem skripte je

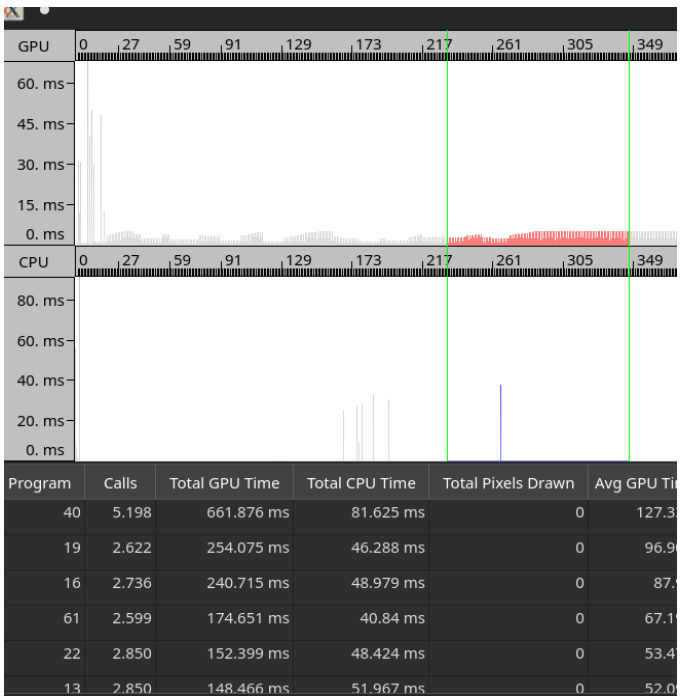
```
apitrace replay --pgpu --pcpu --ppd foo.trace | ./scripts/profileshader.py
```

Umjesto da se koristi gore navedena naredba može se koristiti i GUI koji nudi istu opciju korištenja te skripte jednostavnim klikom ctrl + P. Tim se klikom prvo prikazuje video korištenja aplikacije dok je Apitrace radio u pozadini te nakon nje se otvara novi prozor sa rezultatima. Prozor sa rezultatima za GPU i CPU vremenske okvire i pozive prikazana je na slici 11 (rezultati ne prikazuju nacrtane piksele za svaki poziv)[69].



Slika 11: Apitrace - prozor s rezultatima za GPU i CPU vremenske okvire i pozive

Osim toga GUI omogućuje prikaz rezultata samo za označene kadrove. Ovakav detaljniji prikaz može se vidjeti na slici 12.



Slika 12: Apitrace - prikaz samo označenih kadrova

5.2. Prevođenje koda za sjenčanje

Prevođenje koda za sjenčanje se koristi za postupak kojim se skripte OpenGL jezika za sjenčanje učitavaju u OpenGL da bi se koristile za sjenčanje. OpenGL ima tri načina za prevođenje koda za sjenčanje u OpenGL objekte.
Programski objekt može sadržavati izvršni kôd za sve stupnjeve sjenčanja, tako da je sve što je potrebno za generiranje sjenčanja zapravo vezanje jednog programskog objekta. Izgradnja programa koji sadrži više stupnjeva sjenčanja zahtijeva dvostupanjski postupak prevođenja.

Prvi korak koji je potreban je izrada objekta za sjenčanje koji će se koristiti i prevoditi. Na početku je taj objekt prazan i može se u njega kao prvo upisati tip koda za sjenčanje za koji je namijenjen objekt. Drugi korak je upisivanje GLSL koda koji će se izvoditi u objekt. Nakon što objekt za sjenčanje ima svoj pripadajući GLSL kod može se prevoditi i kao rezultat dobije se prevođeni kod za sjenčanje [70].

5.2.1. Relevantne metrike

Neke od relevantnih metrika koje su dobre za usporedbu performansi aplikacija su [71]:

- SGPR (*Scalar General-Purpose Register*)
  Na modernim AMD GPU-ovima ima do 64 CU-a koji ukupno imaju 3,200 registra tj. 800 registra po SIMD (veličina datoteke je 12.5KB). Jedan valni front (engl. *wavefront*) može alocirati do 112 skalarnih registara u serijama od 16. Posljednjih 6 skalarnih registara koristi se za posebne svrhe (kao što je VCC) i ne mogu se koristiti kao skalarni registri opće namjene za kod korisnika.
- VGPR (*Vector General-Purpose Register*)
  Veličina datoteke vektorskog registra je 64 KB po SIMD-u tj. 16 384 registra (64 * 256). Jedan valni front može dodijeliti do 256 vektorskih registara po dretvi te su oni alocirani u skupini od 4.
- Spill SGPR
  U slučaju kada se popune svi SGPR-ovi i kad ih je potrebno više dolazi do “prolijevanja” (engl. *spill*) podataka koji tada ne mogu samo koristiti memoriju koja je dodijeljena SGPR-u nego dolazi do usporenja zbog korišenja druge memorije.
- Spill VGPR
  U slučaju kada se popune svi VGPR-ovi i kad ih je potrebno više dolazi do “prolijevanja” (engl. *spill*) podataka koji tada ne mogu samo koristiti memoriju koja je dodijeljena VGPR-u

nego dolazi do usporenja zbog korišenja druge memorije.

- Veličina koda

  Veličina (prikazana u bajtovima) će utjecati na kraće učitavanje i potencijalno izvođenje koda.

5.3. LLVM IR proces prevođenja bez i sa optimizacijama

LLVM IR proces prevođenja koda za sjenčanje se inače automatski izvodi sa ili bez optimizacije (na primjer, pri pokretanju neke igre ili ulaska na novu razinu u igri kod koje su potrebni novo prevedeni kodovi za sjenčanje). Iako se to inače izvodi automatski i korisnik to ne primjećuje, moguće je prevesti kod za sjenčanje van upravljačkih programa pomoću LLVM statičnih program-prevoditelja (engl. *LLVM static compiler*, kraće LLC) [72].

Za potrebe ovog diplomskog rada korišten je LLC za komiliranje igara Doom, Dota2, Limbo, Mad Max i Serious Sam BFE 3 sa i bez optimizacije kako bi se detaljnije mogla vidjeti razlika koju donosi optimizacija te kako bi se prijašnje opisane relevantne metrike mogle vidjeti na primjeru. Ukupan broj kodova za sjenčanje koji je korišten za ovaj primjer je 93. Kod svakog koda za sjenčanje 4 puta se pokrenulo prevođenje pomoću LLC-a. Prvi put kako bi se kod za sjenčanje prevodio bez optimizacije (-O0), drugi put sa optimizacijom (-O1), treći put sa jačom optimizacijom (-O2) te četvrti put sa još jačom optimizacijom (-O3). Nažalost u niti jednom od 93 koda za sjenčanje nije se vidjela razlika između -O1, -O2 i -O3 pa će se u daljenjem radu diskutirati samo o razlici između optimiziranog prevođenja (-O1) i ne optimiziranog prevođenja (-O0).

5.3.1. Način sakupljanja podataka sa Steam-a

1. Na Steamu je za odgovarajući igru potrebno postaviti u opcijama pokretanja (`Set launch options`) da naredba bude oblika `AMD_DEBUG=vs,ps,gs,tcs,tes,cs MESA_SHADER_DUMP_PATH=/home/user/igra-shaders MESA_SHADER_CAPTURE_PATH=/home/user/Desktop/igra-shaders %command%`.
2. Pokretanje igre preko terminala te spremanje standardnog izlaza i standardnog izlaza za greške (Mesa ispisuje sjenčanja na standardni izlaz za greške) u tekstualne datoteke, npr. `steam steam://run/570 > log.txt 2> err.txt`.
3. Odvajanje sjenčanja iz tekstualne datoteke `err.txt` u zasebne tekstualne datoteke.
4. Prevođenje odvojenih sjenčanja u strojni kod grafičkog procesora, npr. `llc -O0 -`

`mcpu=gfx1010 shader.ll -o shader.s`. Ovdje je `-O0` nivo optimizacije pa se može koristiti `-O1`, `-O2` ili `-O3`, ovisno o potrebi. Parametrom `-mcpu` navodimo procesor koji koristimo, u ovom slučaju `gfx1010`, odnosno Radeon RX 5700 XT i srodni GPU-i.

5.3.2. Primjeri prevođenja bez i sa optimizacije

Datoteke sa dodatnim informacijama o svakom kodu za sjenčanje (svih 93) se nalaze na Github stranici od korisnice Sanje Božić[6]. Ovdje su kao primjer navedeni samo neki najzanimljiviji kodovi za sjenčanje tj. njihove informacije.

5.3.2.1. Doom

Prevest ćemo kod za sjenčanje broj 17.

-O0

; Kernel info:
; codeLenInByte = 2684
; NumSgprs: **50**
; NumVgprs: **48**
; ScratchSize: 60
; MemoryBound: 0
; FloatMode: 192
; IeeeMode: 0
; LDSByteSize: 0 bytes/workgroup (compile time only)
; SGPRBlocks: 6
; VGPRBlocks: 5
; NumSGPRsForWavesPerEU: 50
; NumVGPRsForWavesPerEU: 48
; Occupancy: 20
; WaveLimiterHint : 0

-O1

; Kernel info:
; codeLenInByte = 1060
; NumSgprs: **38**

6 Poveznica na Github stranicu Sanje Božić: https://github.com/SanjaBozic/diplomski-rad/tree/master/OpenGL

; NumVgprs: **52**

; ScratchSize: 0

; MemoryBound: 0

; FloatMode: 192

; IeeeMode: 0

; LDSByteSize: 0 bytes/workgroup (compile time only)

; SGPRBlocks: 4

; VGPRBlocks: 6

; NumSGPRsForWavesPerEU: 38

; NumVGPRsForWavesPerEU: 52

; Occupancy: 18

; WaveLimiterHint : 0

Prvi po redu kao primjer je kod za sjenčanje kod kojeg je broj skalarnih registara bez optimizacije 50, a sa optimizacijom 35, no broj vektorskih registara bez optimizacije je 48, a sa njom 52. Osim toga broj veličine koda u bajtovima se od 2684 smanjio na 1060 što je upola manje.

Prevest ćemo i kod za sjenčanje broj 5.

-O0

; Kernel info:

; codeLenInByte = 632

; NumSgprs: **26**

; NumVgprs: **8**

; ScratchSize: 16

; MemoryBound: 0

; FloatMode: 192

; IeeeMode: 0

; LDSByteSize: 0 bytes/workgroup (compile time only)

; SGPRBlocks: 3

; VGPRBlocks: 1

; NumSGPRsForWavesPerEU: 26

; NumVGPRsForWavesPerEU: 11

; Occupancy: 20

; WaveLimiterHint : 0

-O1

; Kernel info:
; codeLenInByte = 224
; NumSgprs: **33**
; NumVgprs: **6**
; ScratchSize: 0
; MemoryBound: 0
; FloatMode: 192
; IeeeMode: 0
; LDSByteSize: 0 bytes/workgroup (compile time only)
; SGPRBlocks: 4
; VGPRBlocks: 1
; NumSGPRsForWavesPerEU: 33
; NumVGPRsForWavesPerEU: 11
; Occupancy: 20
; WaveLimiterHint : 0

Drugi primjer na igri Doom prikazuje drugačiju situaciju. U ovom slučaju se sa 26 povećao broj skalarnih registra na 33, a veličina koda se od 632 smanjila na čak 224 što je skoro trostruko manja veličina koda.

5.3.2.2. Dota 2

Prevest ćemo kod za sjenčanje broj 12.

-O0

; Kernel info:
; codeLenInByte = 36
; NumSgprs: **5**
; NumVgprs: **4**
; ScratchSize: 0
; MemoryBound: 0
; FloatMode: 192
; IeeeMode: 0

; LDSByteSize: 0 bytes/workgroup (compile time only)

; SGPRBlocks: 0

; VGPRBlocks: 1

; NumSGPRsForWavesPerEU: 5

; NumVGPRsForWavesPerEU: 15

; Occupancy: 20

; WaveLimiterHint : 0

-O1

; Kernel info:

; codeLenInByte = 28

; NumSgprs: **5**

; NumVgprs: **4**

; ScratchSize: 0

; MemoryBound: 0

; FloatMode: 192

; IeeeMode: 0

; LDSByteSize: 0 bytes/workgroup (compile time only)

; SGPRBlocks: 0

; VGPRBlocks: 1

; NumSGPRsForWavesPerEU: 5

; NumVGPRsForWavesPerEU: 15

; Occupancy: 20

Kod Dote 2 kao primjer je prikazan kod za sjenčanje koji se nije mijenjao ni sa optimizacijom. I nakon optimizacije ostao je isti broj skalarnih i vektorskih registara, samo se malo smanjila veličina koda (sa 36 na 27).

5.3.2.3. Mad Max

Prevest ćemo kod za sjenčanje broj 14.

-O0

; Kernel info:

; codeLenInByte = **1124**

; NumSgprs: **42**

; NumVgprs: **13**

; ScratchSize: 24

; MemoryBound: 0

; FloatMode: 192

; IeeeMode: 0

; LDSByteSize: 0 bytes/workgroup (compile time only)

; SGPRBlocks: 5

; VGPRBlocks: 1

; NumSGPRsForWavesPerEU: 42

; NumVGPRsForWavesPerEU: 13

; Occupancy: 20

; WaveLimiterHint : 0

-O1

; Kernel info:

; codeLenInByte = **356**

; NumSgprs: **34**

; NumVgprs: **12**

; ScratchSize: 0

; MemoryBound: 0

; FloatMode: 192

; IeeeMode: 0

; LDSByteSize: 0 bytes/workgroup (compile time only)

; SGPRBlocks: 4

; VGPRBlocks: 1

; NumSGPRsForWavesPerEU: 34

; NumVGPRsForWavesPerEU: 12

; Occupancy: 20

Kod za sjenčanje iz igre Mad Max primjer je optimizacije u kojoj se smanjuje dužina koda, kao i broj vektorskih i skalarnih registara. Inicijalni broj od 42 skalarna registra i 13 vektorskih registra se nakon optimizacije smanjio na 34 skalarna i 12 vektorskih registra. Osim toga smanjila mu se veličina koda od 1124 na 356 bajta.

5.3.2.4. Limbo

Prevest ćemo kod za sjenčanje broj 4.

-O0

; Kernel info:
; codeLenInByte = 72
; NumSgprs: 8
; NumVgprs: 15
; ScratchSize: 0
; MemoryBound: 0
; FloatMode: 192
; IeeeMode: 0
; LDSByteSize: 0 bytes/workgroup (compile time only)
; SGPRBlocks: 0
; VGPRBlocks: 3
; NumSGPRsForWavesPerEU: 8
; NumVGPRsForWavesPerEU: 26
; Occupancy: 20
; WaveLimiterHint : 0

-O1

; Kernel info:
; codeLenInByte = 20
; NumSgprs: 8
; NumVgprs: 15
; ScratchSize: 0
; MemoryBound: 0
; FloatMode: 192
; IeeeMode: 0
; LDSByteSize: 0 bytes/workgroup (compile time only)
; SGPRBlocks: 0
; VGPRBlocks: 3
; NumSGPRsForWavesPerEU: 8
; NumVGPRsForWavesPerEU: 26

; Occupancy: 20

; WaveLimiterHint : 0

Promatranjem igre Limbo primjećujemo kako su kod nje (kao i kod Dote 2) postojali kodovi za sjenčanje koji se nisu pretežito mijenjali pri optimizaciji tj. broj skalarnih i vektorskih registara je isti, samo se veličina koda mijenjala na manje.

Prevest ćemo i kod za sjenčanje broj 1.

-O0

; Kernel info:

; codeLenInByte = **104**

; NumSgprs: **22**

; NumVgprs: **10**

; ScratchSize: 0

; MemoryBound: 0

; FloatMode: 192

; IeeeMode: 1

; LDSByteSize: 0 bytes/workgroup (compile time only)

; SGPRBlocks: 2

; VGPRBlocks: 1

; NumSGPRsForWavesPerEU: **22**

; NumVGPRsForWavesPerEU: **10**

; Occupancy: 20

; WaveLimiterHint : 0

-O1

; Kernel info:

; codeLenInByte = **56**

; NumSgprs: **11**

; NumVgprs: **5**

; ScratchSize: 0

; MemoryBound: 0

; FloatMode: 192

; IeeeMode: 1

; LDSByteSize: 0 bytes/workgroup (compile time only)

; SGPRBlocks: 1

; VGPRBlocks: 0

; NumSGPRsForWavesPerEU: **11**

; NumVGPRsForWavesPerEU: **5**

; Occupancy: 20

; WaveLimiterHint : 0

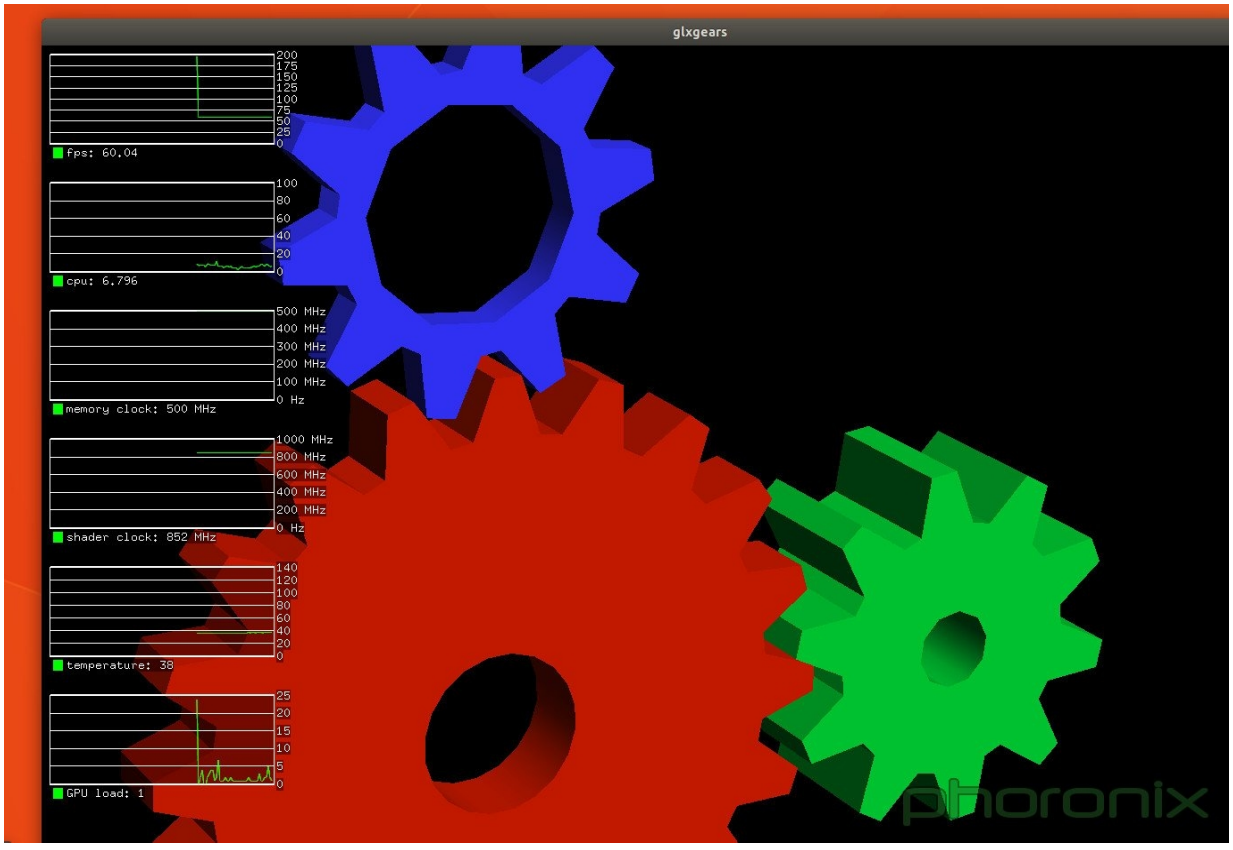
Kao zadnji primjer igre Limbo izabran je kod za sjenčanje koji nakon optimizacije duplo bolji nego prije. Broj skalarnih registra je sa 22 pao na 11, broj vektorskih registra je sa 10 pao na 5 te se veličina koda u bajtovima sa 104 smanjila na 56.

6. Mjerenje performansi: OpenGL vs Vulkan

6.1. Gallium HUD

Gallium HUD (*Heads-Up Display*) omogućuje prikaz velikog broja sustavskih podataka o GPU-u i CPU-u u sloju koji se nalazi iznad OpenGL programa. Može se koristiti sa svim Gallium3D upravljačkim programima poput RadeonSI, Nouveau NVC0 i LLVMpipe. HUD se aktivira putem varijable okoline GALLIUM_HUD, putem koje se može i odabrati koji će podaci biti prikazani [73].

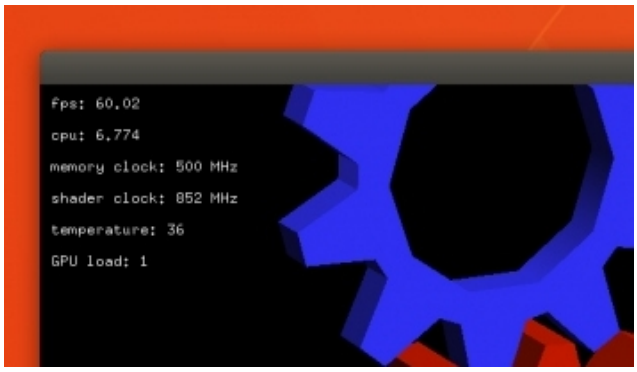
Iako Gallium3D HUD nudi puno korisnih informacija, svaki senzor također prikazuje linijski grafikon rezultata, to može uzrokovati nered čak i ako je omogućeno samo nekoliko opcija. To se može najbolje vidjeti na slici 13.



Slika 13: Gallium HUD prikaz sa grafovima, preuzeto iz: https://www.phoronix.com/image-viewer.php?id=2018&image=gallium_hud_simple1_lrg

Zbog tog razloga Gallium HUD ima mogućnost jednostavnog prikaza koji sakriva grafikone te

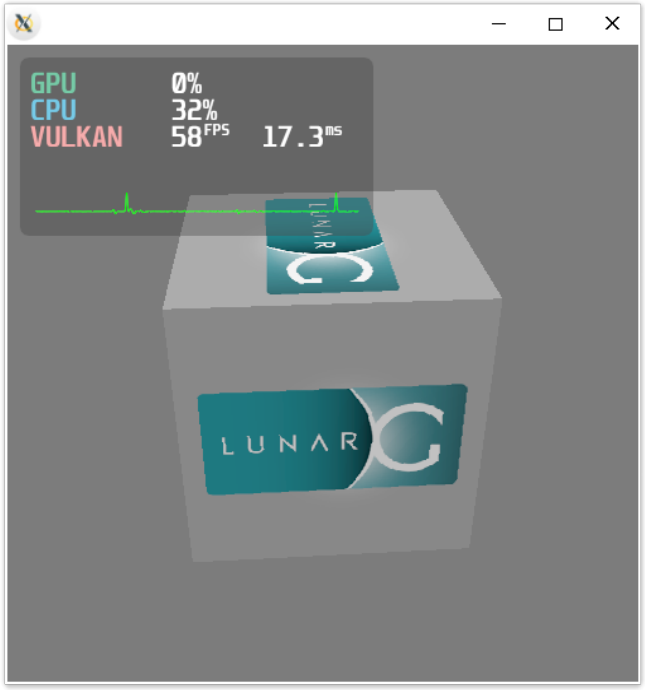
prikazuje samo osnovne podatke koji su potrebni na lijevoj strani aplikacije koja je pokrenuta. Slika ovakvog prikaza prikazana je na slici 14.



Slika 14: Jednostavan prikaz Gallium HUD-a, preuzeto iz: https://www.phoronix.com/image-viewer.php?id=2018&image=gallium_hud_simple2_lrg

6.2. MangoHud

MangoHud je alternativa Gallium HUD-u koja je jednostavnija za korištenje te ima ljepši i kompaktniji prikaz informacija na ekranu [74]. Između ostalog, to znači da zauzima manje prostora nego Gallium HUD i koristi modernije zaglađene oblike pisma. Osim toga, kompatibilan je sa Vulkan i OpenGL igrama. Osnovni prikaz bez ikakvih dodatnih opcija izgleda kao na slici 15.



Slika 15: Izgled MangoHUD-a, preuzeto iz: https://plus.diolinux.com.br/uploads/default/original/2X/b/b4cacadb3ea61fc26d604d5552683302d4a2e589.png

Za pokretanje MangoHuda sa igrama koje se nalaze na Steam-u potrebno je u postavkama za igru otvoriti dodatne opcije za pokretanje (engl. *set launch options*) te u njima nadodati `MANGOHUD = 1` za Vulkan igre ili `mangohud` za OpenGL igre. Od svih testiran igara iznimka tom pravilu je Lara Croft GO gdje je potrebno napisati `MANGOHUD_DLSYM=1 mangohud` jer samo `mangohud` nije dovoljan.

Osim toga, velika razlika između Gallium HUD-a i MangoHuda je u tome da MangoHud ima svoj GUI kod kojeg se mogu namjestiti dodatne opcije kao što su mjerenje temperature GPU-a i CPU-a, informacije o grafičkoj kartici, informacije o potrošnji memorije s nasumičnim pristupom (engl. *random-access memory*, kraće RAM) te mnoge druge opcije koje bi se inače trebale sve opisati u jednoj liniji pri pokretanju igre [75].

Uz to postoji i opcija spremanja podataka o izmjerenom broju sličica po sekundi (engl. *frames per second*, kraće FPS) u datoteku koja se poslije može učitati u web stranicu FlightlessMango. Sa učitanim podacima stranica automatski generira ilustrirani prikaz rezultata koji se može koristiti pri uspoređivanju performansi igara [76].

6.3. FPS

FPS je mjera brzine kojom se prikazuju sličice na ekranu. Na primjer, videozapis reproducira sa 24 sličice u sekundi, to znači da svaka sekunda videozapisa prikazuje 24 različite slike. Ta brzina kojom se prikazuju sličice vara ljudski mozak tako da se umjesto odvojenih statičnih slika vidi kontinuirani pokret. Značajne vrijednosti FPS-a su redom [77]:

- 24 FPS-a – minimalna brzina sličica u sekundi kako bi se vidjelo realistično kretanje, no ako se na sceni događa previše stvari u isto vrijeme prikaz će biti mutan
- 30 FPS-a - vidjet ćete više detalja tijekom scena s jakim pokretima, međutim, kretanja će početi izgledati pomalo neprirodno
- 60+ FPS-a - donosi fluidnost prikaza, sve više od 30 sličica u sekundi obično je rezervirano za snimanje užurbanih scena s puno pokreta, poput video igara, atletike ili bilo čega za što je potreban usporen prikaz. Igre se reproduciraju ovom brzinom jer se na zaslonu odjednom događa puno toga, a više sličica znači da se vidi više detalja u sceni

Potrebno je naglasiti da prikaz jedne sličice treba trajati maksimalno 16.6 ms kako bi sama igra radila na 60 FPS-a. To je posebno velik problem kad postoji potreba za prevođenjem koda za sjenčanje pa se potreba za pokretanjem koda za sjenčanje treba predvidjeti i prevođenje pokrenuti dovoljno rano da prevedeni kod bude spreman za korištenje u trenutku kad se treba izvesti. Naravno, kad su kodovi za sjenčanje već prevedeni nije potrebno nikakvo čekanje te je igra još brža [78].

Testiranje performansi igara uglavnom se svodi na mjerenje i usporedbu FPS-a. U ovom radu je na ovaj način testirano 5 igara što je opisano u slijedećem poglavlju.

6.4. Primjena MangoHUD-a za mjerenje FPS-a

Kao primjere primjene MangoHUD-a koristile su se igre Doom, Mad Max, Lara Croft GO, Serious Sam Fusion, The Talos Principle. Sve igre su se pokretale preko platforme Steam. Sve datoteke korištene u ovom dijelu diplomskog rada mogu se pogledati na github stranici Sanje Božić[7].

Osim kao primjere MangoHUD-a ove igre su izabrane i za usporedbu performansi između Vulkana i OpenGL-a. Prije nego što pređemo na usporedbu važno je nadodati par informacija o samom računalu. GPU koji je u računalu je Radeon RX 5700 XT dok je CPU AMD Ryzen 7 3700X sa 16 GB RAM-a. Operativni sustav na kojemu su izvedena mjerenja je Arch Linux s verzijom jezgre

7 Poveznica: https://github.com/SanjaBozic/diplomski-rad/tree/master/FPS

Linuxa 5.8.1.arch1-1, a GPU upravljački program koji je korišten je dio razvojne verzije Mese 20.2.

6.4.1.1. Doom

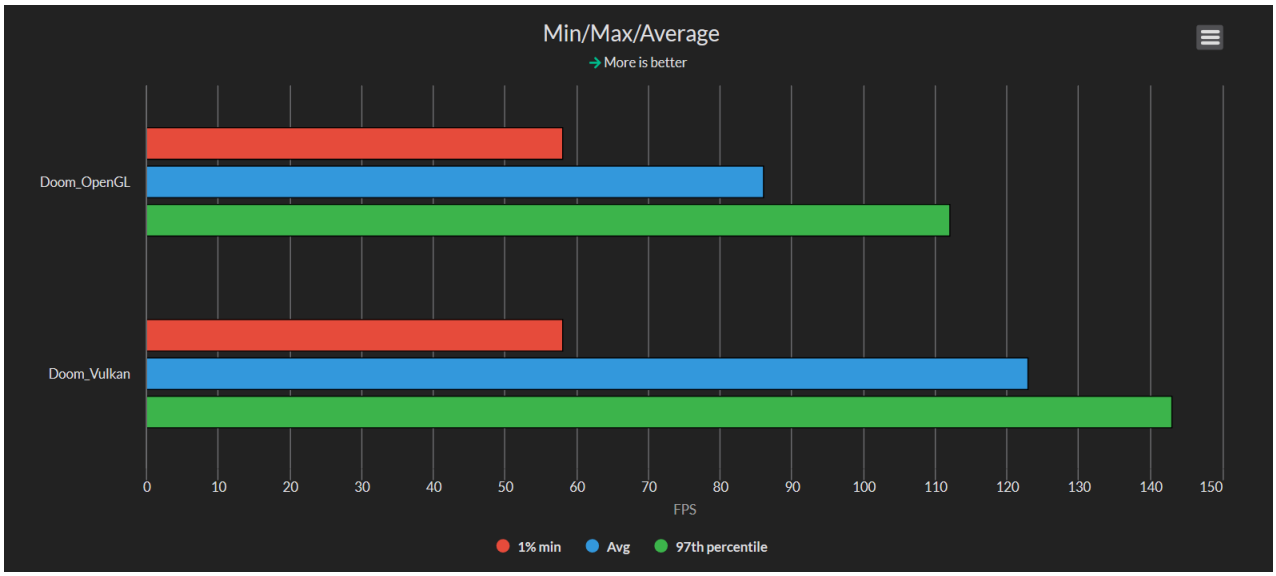
U nastavku je dana usporedba FPS-a kod korištenja OpenGL-a i Vulkana. Možemo primjetiti na grafikonu kako zelena linija prikazuje Doom Vulkan, a crvena Doom OpenGL. Po ovoj prvoj slici vidimo kako je FPS na Vulkanu mnogo bolji i stabilniji dok na OpenGL-u i mnogo više varira i ukupno je znatno manji. U OpenGL-u vrhunac FPS-a je na 130 FPS-a dok je na Vulkanu stabilnih 144 FPS-a. Odmah po prvom grafikonu možemo zaključiti kako je po FPS-u Vulkan imao znantno bolje performanse nego OpenGL. Pad FPS-a na 60 prikazan na grafikonu predstavlja pred-snimljenu scenu (engl. *cutscene*) na kojoj je maksimalni FPS zaključan na 60.

Slika 16: Doom - grafički prikaz FPS-a, preuzeto iz: https://flightlessmango.com/games/6406/logs/790

Na slici 17 prikazuje se vrijeme sličica koji bi trebalo biti što manje za bolje performanse. Možemo primjetiti kako je i u ovom slučaju Vulkan znatno bolji od svog konkurenta OpenGL-a. Naime Vulkan stabilno zadržava vrijeme sličica od 7 milisekundi dok je OpenGL imaju samo jedan trenutak na kojem je imao vrijeme sličica od 7,5 milisekundi. OpenGL se više zadržavao na oko 11 milisekundi po sličici.

Slika 17: Doom - grafički prikaz vremena sličica, preuzeto iz: https://flightlessmango.com/games/6406/logs/790

Na slici 18 prikazuje se stupčastim grafom FPS vrijeme. Zelenom bojom je prikazan maksimalni FPS, plavom bojom prosječan FPS, a crvenom minimalan FPS. Možemo vidjeti kako je prosječan FPS sa Vulkanom oko 122 FPS-a, a sa OpenGL-om oko 85 FPS-a. Osim toga vidimo značajnu razliku u maksimalnoj vrijednosti FPS-a gdje je Vulkan imao 144 FPS-a, a OpenGL 112 FPS-a. Ipak možemo primijetiti da su oba dva konkurenta imala jednaku minimalnu vrijednosti FPS-a od oko 58 FPS-a zbog predsnimljene scene koja se odvijala.



Slika 18: Doom - prikaz minimalnog, maksimalnog i prosječnog FPS-a, preuzeto iz: https://flightlessmango.com/games/6406/logs/790

Slika 19 prikazuje sumiranu usporedbu između OpenGL-a i Vulkan-a za igru Doom. Kao što je

prikazano i na prijašnjim grafikonima Vulkan je bio znatno bolji sa prosječno 43% boljim performansama nego OpenGL.

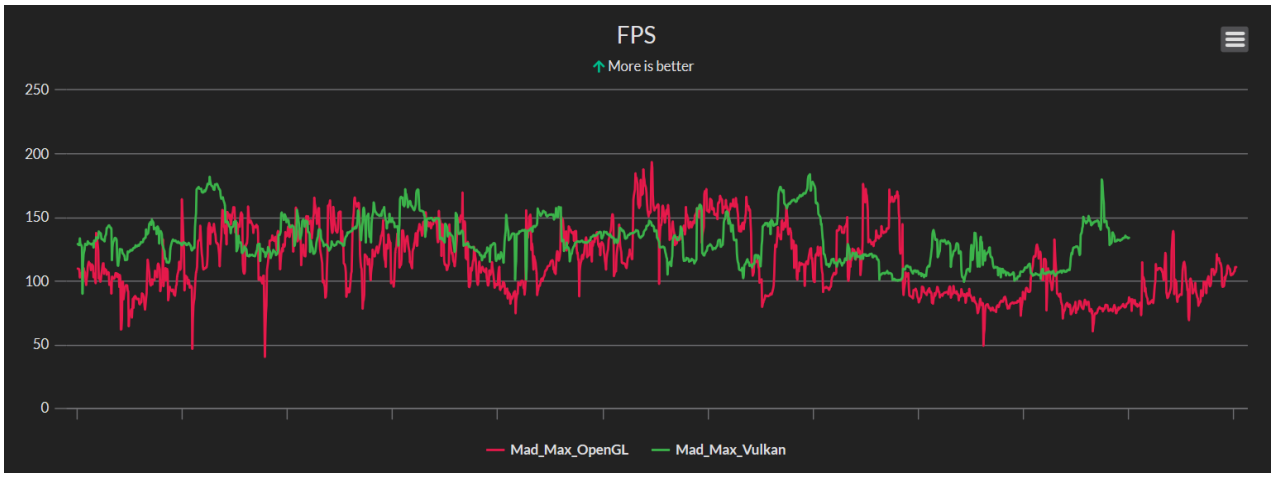
Color	Name	1% Min	Avg	97th percentile	Avg % vs Doom_OpenGL
	Doom_OpenGL	58	86	112.0	100.0 %
	Doom_Vulkan	58	123	143.0	143.02 %

Slika 19: Doom - ukupan prikaz rezultata, preuzeto iz: https://flightlessmango.com/games/6406/logs/790

Sve informacije o ovoj usporedbi se mogu vidjeti i na flightlessmango stranici gdje je i napravljena usporedba[8].

6.4.1.2. Mad Max

Sljedeća igra kod koje je uspoređen Vulkan i OpenGL je Mad Max. Na slici 20 prikazana je razlika u FPS-u koja nije toliko različita kao i u prijašnjem primjeri (igri Doom). Ovaj put Vulkan i OpenGL variraju slično u FPS-u, ali ako bolje pogledamo primijeti se kako je zelena linija za Vulkan ipak malo više nego crvena (OpenGL).

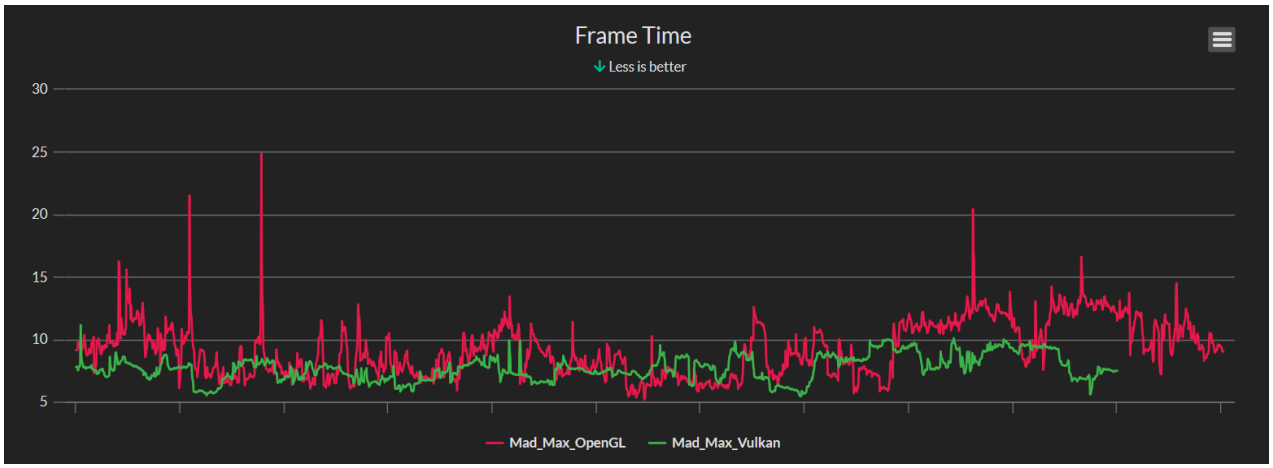


Slika 20: Mad Max - grafički prikaz FPS-a, preuzeto iz: https://flightlessmango.com/games/12978/logs/791

Pri slici 21 rezultati su isto poprilično različiti nego na igri Doom. OpenGL i Vulkan su jako blizu sa svojim vremenima sličica ali možemo primijetiti kako Vulkan manje varira u svojim vremenima

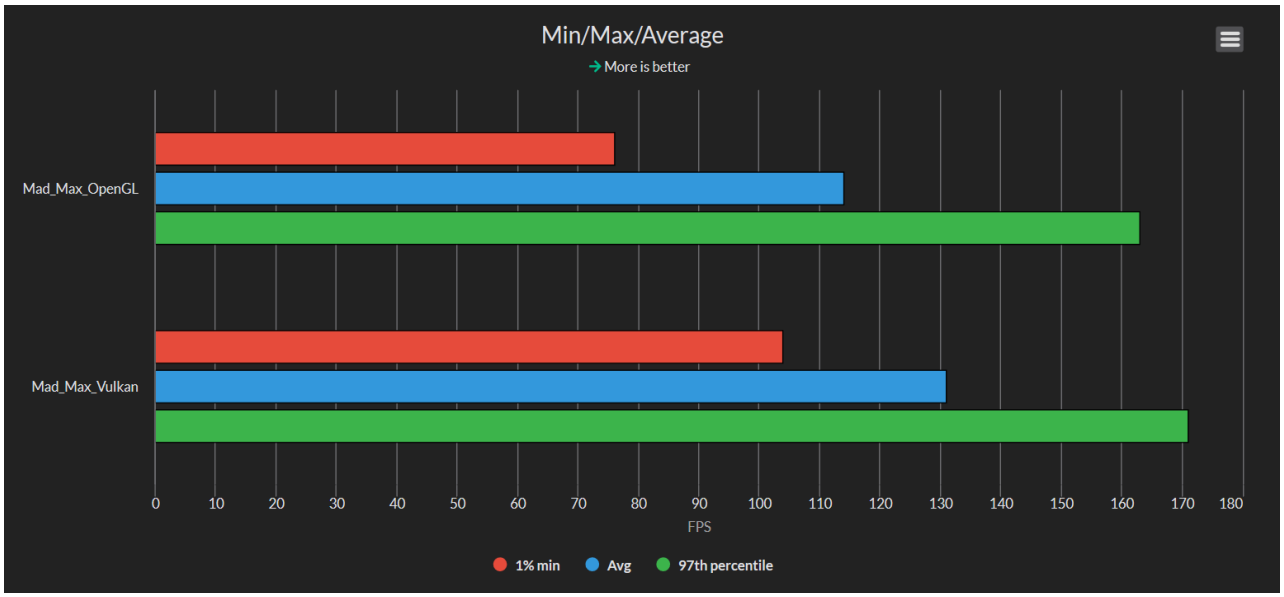
8 Poveznica do Doom vizualizacija: https://flightlessmango.com/games/6406/logs/790

nego OpenGL koji je u jednom trenutku bio porastao i na 25 milisekundi dok je Vulkan ostao pri maksimalno 12 milisekundi.



Slika 21: Mad Max - grafički prikaz vremena sličica, preuzeto iz: https://flightlessmango.com/games/12978/logs/791

Kod minimalnog, maksimalnog i prosječnog FPS-a se već vidi razlika. Za razliku od igre Doom, OpenGL i Vulkan u ovom slučaju nemaju jednaki minimalni FPS. Kod Vulkana on je oko 104 FPS-a dok je kod OpenGL-a oko 75 FPS-a. Osim toga prosječan FPS za Mad Max igru na Vulkanu je 131 FPS-a dok je na OpenGL-u 115 FPS-a. Maksimalni FPS je poprilično veći nego što je bio kod Doom-a gdje je za Vulkan 171 FPS, a za OpenGL 162 FPS-a.



Slika 22: Mad Max - prikaz minimalnog, maksimalnog i prosječnog FPS-a, preuzeto iz: https://flightlessmango.com/games/12978/logs/791

Kao što je moglo biti i očekivano na slici 23 možemo primijetiti kako je razlika između OpenGL-a i Vulkana za Mad Max igru manja nego što je bila razlika za Doom, no ipak je Vulkan pobijedio sa 14,91% boljim performansama.

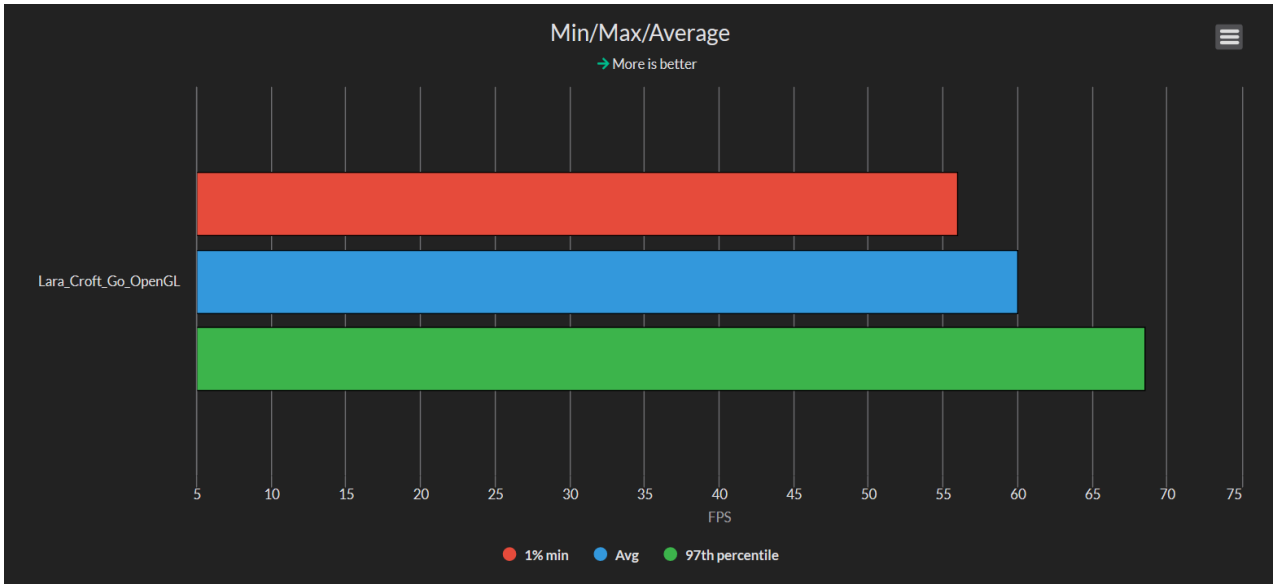
Color	Name	1% Min	Avg	97th percentile	Avg % vs Mad_Max_OpenGL
	Mad_Max_OpenGL	76	114	163.0	100.0 %
	Mad_Max_Vulkan	104	131	171.0	114.91 %

Slika 23: Mad Max - ukupan prikaz rezultata, preuzeto iz: https://flightlessmango.com/games/12978/logs/791

Sve informacije o ovoj usporedbi se mogu vidjeti i na flightlessmango stranici gdje je i napravljena usporedba[9].

6.4.1.3. Lara Croft GO

Osim usporedbe OpenGL-a i Vulkana ovdje je jedan primjer samog testiranja OpenGL-a na igri Lara Croft GO pošto ova igra nema mogućnost pokretanja sa Vulkanom. Na stupičastom grafu prikazan je minimalan FPS pri igranju igre koji je bio 56 FPS-a, prosječan FPS od 60 FPS-a i maksimalan FPS od 68 FPS-a.



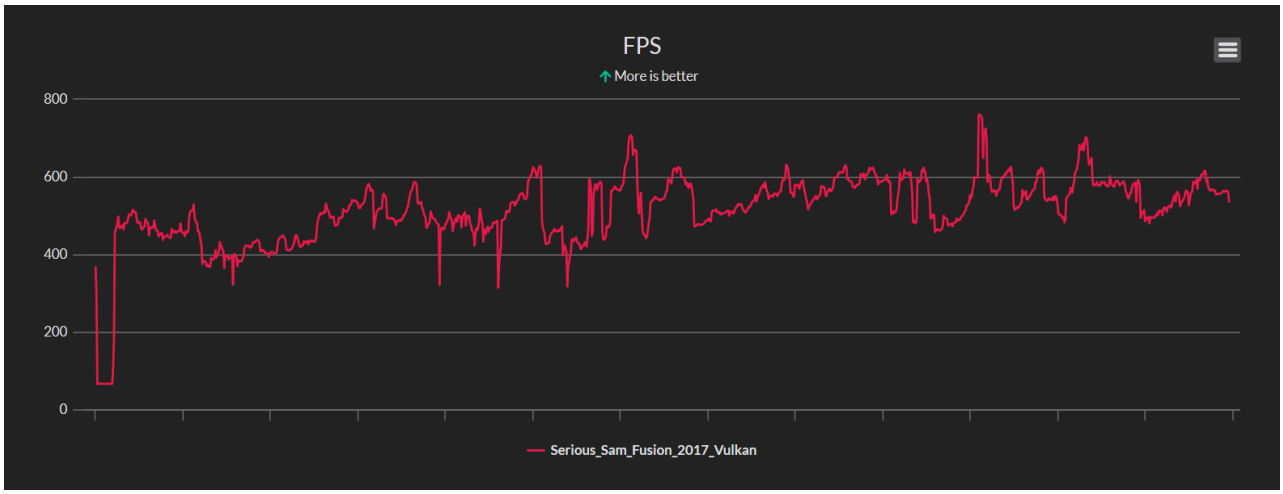
Slika 24: Lara Croft GO - prikaz minimalnog, maksimalnog i prosječnog FPS-a, preuzeto iz: https://flightlessmango.com/games/12135/logs/792

9 Poveznica do Mad Max vizualizacija: https://flightlessmango.com/games/12978/logs/791

Sve informacije o ovoj usporedbi se mogu vidjeti i na flightlessmango stranici gdje je i napravljena vizualizacija[10].

6.4.1.4. Serious Sam Fusion 2017

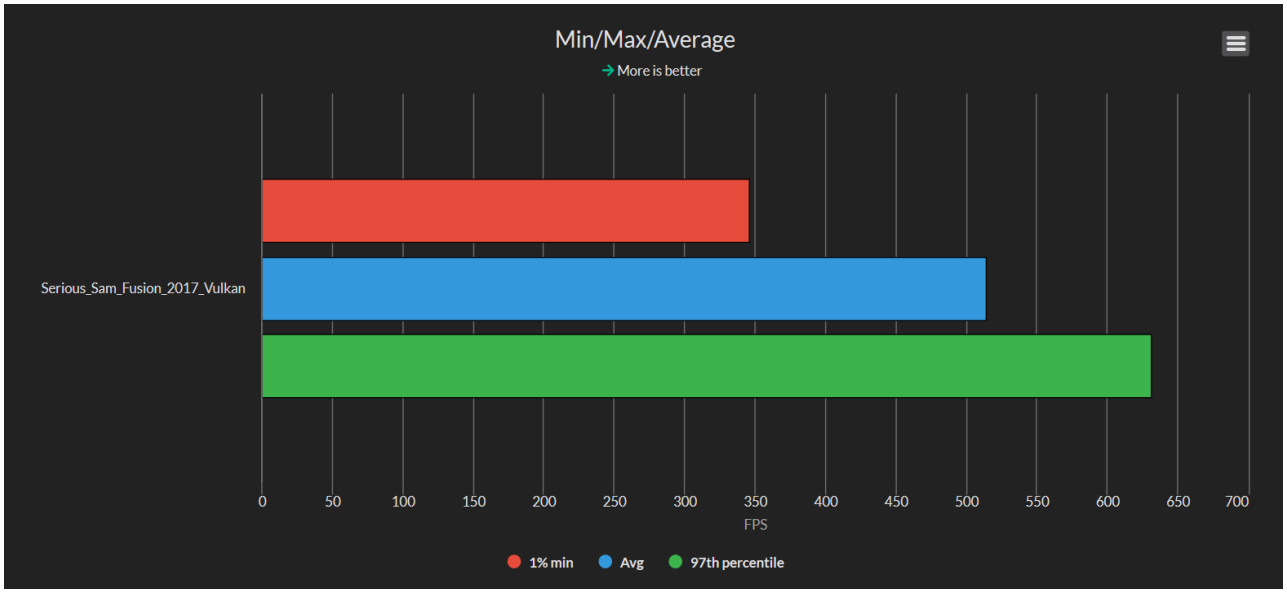
Kao primjer Vulkan igre ovdje je Serious Sam Fusion koji nema mogućnost pokretanja preko OpenGL-a. Osim toga jedna je od dvije igre koja je napravljena u Hrvatskoj od poznate zagrebačke tvrtke Croteam. Kao što se može vidjeti na prvom grafikonu FPS u igri varira ali je jako visok sa malim padom na početku igranja igre.



Slika 25: Serious Sam Fusion 2017 - grafički prikaz FPS-a, preuzeto iz: https://flightlessmango.com/games/18971/logs/793

Sa pregledom druge slike koja prikazuje stupičasti graf primjećuje se kako je minimalni FPS za ovu igru oko 345 FPS-a, prosječan FPS oko 510 FPS-a, a maksimalni FPS oko 625 FPS-a što je više nego odlično.

10 Poveznica do Lara Croft GO vizualizacija: https://flightlessmango.com/games/12135/logs/792



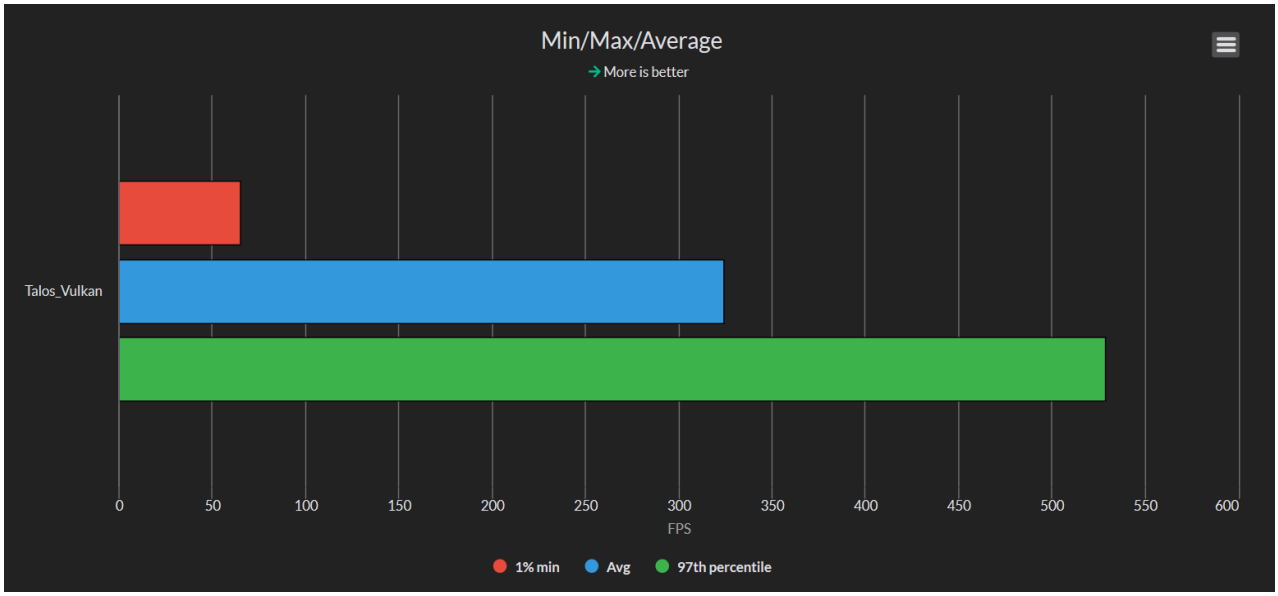
Slika 26: Serious Sam Fusion 2017 - prikaz minimalnog, maksimalnog i prosječnog FPS-a, preuzeto iz: https://flightlessmango.com/games/18971/logs/793

Sve informacije o ovoj usporedbi se mogu vidjeti i na flightlessmango stranici gdje je i napravljena vizualizacija[11].

6.4.1.5. The Talos Principle

Zadnja igra u ovom skupu primjera je The Talos Principle (druga hrvatska igra na ovom popisu) koja je ovdje interesantna zbog svoje velike razlike u minimalnom i maksimalnom FPS-u osim toga prva je igra koja je imala podršku za Vulkan na svijetu [79]. Naime, minimalni FPS pri igranju ove igre je 60, a maksimalni je 525 FPS-a. Uz to prosječan FPS u igri je oko 325 FPS-a.

11 Poveznica do Serious Sam Fusion 2017 vizualizacija: https://flightlessmango.com/games/18971/logs/793



Slika 27: The Talos Principle - prikaz minimalnog, maksimalnog i prosječnog FPS-a, preuzeto iz: https://flightlessmango.com/games/23147/logs/794

Sve informacije o ovoj usporedbi se mogu vidjeti i na flightlessmango stranici gdje je i napravljena vizualizacija[12].

12 Poveznica do The Talos Principle vizualizacija: https://flightlessmango.com/games/23147/logs/794

7. Borba za Vulkan: LLVM vs ACO

U ovom poglavlju radimo usporedbu performansi LLVM-a i ACO-a na igrama: Doom, Dota 2, Dota Underlords, Mad Max, Rise of the Tomb Rider, Serious Sam Fusion 2017, The Talos Principle i vkQuake. U nastavku su u tablicama navedene promjene kod korištenja ACO-a u odnosu na korištenje LLVM-a.

Red “svi pogođeni” u tablici označava programe za sjenčanje koji su pokrenuti u igri (ne samo prevedeni).

7.1. Doom

Tablica 3: Doom - usporedba LLVM i ACO

Detalji	Broj programa za sjenčanje	SGPR	VGPR	Spill SGPR	Spill VGPR	Veličina koda	Max Waves
	718	-14,40%	-27,40%	-100,00%		-6,37%	2,53%
Svi pogođeni	718	-14,40%	-27,40%	-100,00%		-6,37%	2,53%
Ukupno	718	-14,40%	-27,40%	-100,00%		-6,37%	2,53%

Kao što je na tablici prikazano, veličina koda se u usporedbi sa LLVM-om u ACO-u smanjila za 6,37%, zauzeće SGPR-a je smanjen za 14,40%, a zauzeće VGPR-a za 27,40%. Osim toga na novom ACO-u više nema prolijevanja SGPR-a kojeg je na LLVM-u bilo.

7.2. Dota 2

Tablica 4: Dota 2 - usporedba LLVM i ACO

Detalji	Broj programa za sjenčanje	SGPR	VGPR	Spill SGPR	Spill VGPR	Veličina koda	Max Waves
	185	-9,94%	-11,06%			-2,32%	-3,28%
Svi pogođeni	185	-9,94%	-11,06%			-2,32%	-3,28%
Ukupno	185	-9,94%	-11,06%			-2,32%	-3,28%

Na tablici vidimo kako se veličina koda u usporedbi sa LLVM u ACO smanjila za 2,32%, zauzeće SGPR-a je smanjeno za 9,94%, a zauzeće VGPR-a za 11,06%.

7.3. Dota Underlords

Tablica 5: Dota Underlords - usporedba LLVM i ACO

Detalji	Broj programa za sjenčanje	SGPR	VGPR	Spill SGPR	Spill VGPR	Veličina koda	Max Waves
	485	-6,74%	-5,09%			-3,31%	-4,34%
Svi pogođeni	485	-6,74%	-5,09%			-3,31%	-4,34%
Ukupno	485	-6,74%	-5,09%			-3,31%	-4,34%

Za Dota Underlords tablica prikazuje kako se veličina koda u usporedbi sa LLVM u ACO smanjila za 3,31%, zauzeće SGPR-a je smanjeno za 6,74%, a zauzeće VGPR-a za 5,09%.

7.4. Mad Max

Tablica 6: Mad Max - usporedba LLVM i ACO

Detalji	Broj programa za sjenčanje	SGPR	VGPR	Spill SGPR	Spill VGPR	Veličina koda	Max Waves
	597	-11,09%	-15,47%	-100,00%		-8,73%	-0,85%
Svi pogođeni	597	-11,09%	-15,47%	-100,00%		-8,73%	-0,85%
Ukupno	597	-11,09%	-15,47%	-100,00%		-8,73%	-0,85%

Pri pregledu Mad Max igre, tablica prikazuje kako se veličina koda u usporedbi sa LLVM u ACO smanjila za 8,73%, zauzeće SGPR-a je smanjeno za 11,09%, a zauzeće VGPR-a za 15,47%. Osim toga na novom ACO-u više nema prolijevanja SGPR-a kojeg je na LLVM-u bilo.

7.5. Rise of the Tomb Rider

Tablica 7: Rise of the Tomb Rider - usporedba LLVM i ACO

Detalji	Broj programa za sjenčanje	SGPR	VGPR	Spill SGPR	Spill VGPR	Veličina koda	Max Waves
	16797	-16,60%	-1,77%	-100,00%		-17,11%	-8,68%
Svi pogođeni	16797	-16,60%	-1,77%	-100,00%		-17,11%	-8,68%
Ukupno	16797	-16,60%	-1,77%	-100,00%		-17,11%	-8,68%

Na tablici vidimo kako se veličina koda u usporedbi sa LLVM u ACO smanjila za 17,11% što je najviše od svih igara koje su bile testirane, zauzeće SGPR-a je smanjeno za 16,60%, a zauzeće VGPR-a za 1,77%. Osim toga na novom ACO više nema prolijevanja SGPR-a kojeg je na LLVM-u bilo.

7.6. Serious Sam Fusion 2017

Tablica 8: Serious Sam Fusion 2017 - usporedba LLVM i ACO

Detalji	Broj programa za sjenčanje	SGPR	VGPR	Spill SGPR	Spill VGPR	Veličina koda	Max Waves
	390	-17,62%	-16,70%	-100,00%		-7,33%	3,60%
Svi pogođeni	388	-17,67%	-16,79%	-100,00%		-7,35%	3,62%
Ukupno	390	-17,62%	-16,70%	-100,00%		-7,33%	3,60%

Za Serious Sam 2017 tablica prikazuje kako se veličina koda u usporedbi sa LLVM u ACO smanjila za 7,33%, zauzeće SGPR-a je smanjeno za 17,62%, a zauzeće VGPR-a za 16,70%. Osim toga na novom ACO-u više nema prolijevanja SGPR-a kojeg je na LLVM-u bilo.

7.7. The Talos Principle

Tablica 9: The Talos Principle - usporedba LLVM i ACO

Detalji	Broj programa za sjenčanje	SGPR	VGPR	Spill SGPR	Spill VGPR	Veličina koda	Max Waves
	453	-19,81%	-15,01%	-100,00%		-6,33%	2,58%
Svi pogođeni	451	-19,86%	-15,07%	-100,00%		-6,34%	2,59%
Ukupno	453	-19,84%	-15,01%	-100,00%		-6,33%	2,58%

Pri pregledu The Talos Principle igre tablice prikazuje kako se veličina koda u usporedbi sa LLVM u ACO smanjila za 6,33%, zauzeće SGPR-a je smanjeno za 19,84% što je najviše smanjenje zauzeća SGPR-a od svih igara, a zauzeće VGPR-a za 15,01%. Osim toga na novom ACO više nema prolijevanja SGPR-a kojeg je na LLVM-u bilo.

7.8. vkQuake

Tablica 10: vkQuake - usporedba LLVM i ACO

Detalji	Broj programa za sjenčanje	SGPR	VGPR	Spill SGPR	Spill VGPR	Veličina koda	Max Waves
	60	-2,93%	-44,87%			-3,69%	
Svi pogođeni	60	-2,93%	-44,87%			-3,69%	
Ukupno	60	-2,93%	-44,87%			-3,69%	

Na tablici vidimo kako se veličina koda u usporedbi sa LLVM u ACO smanjila za 3,69%, zauzeće SGPR-a je smanjeno za 2,93%, a zauzeće VGPR-a za 44,87% što je znatno velika razlika i najveće smanjenje zauzeća VGPR-a u usporedbi sa drugim igrama.

Ukupno vidimo da je u svim primjerima prosječna vrijednost smanjenja veličine koda pri korištenju ACO-a 6,90%, prosječno zauzeće SGPR-a 12,42% manje te zauzeće VGPR-a 17,17% manje. Možemo zaključiti kako je ACO bolji nego LLVM.

8. Zaključak

U prijašnjim poglavljima mogli smo vidjeti kako se razvijala arhitektura AMD-ovih grafičkih procesora te naslutiti kako će razvoj i dalje nastaviti. AMD je već objavio kako će u budućnosti biti predstavljena nova arhitektura RDNA 2 koja će biti iskorištena u grafičkim karticama kodnog imena “Big Navi”. Najnovija značajka koju će donijeti nova RDNA 2 je praćenje zrake (engl. *ray tracing*), koje se već dugo koristi za prikazivanje u statičkim slikama i pruža realistično osvjetljenje simulirajući fizičko ponašanje svjetlosti [9]. Praćenje zraka izračunava boju piksela prateći put kojim bi svjetlost prošla kad bi putovala od oka gledatelja kroz virtualnu 3D scenu. Tako se u budućnosti možemo nadati još boljoj kvaliteti grafike, a možda i boljim performansama na AMD-ovim GPU-ima nego što je sada [80].

Nakon pregleda AMD-ovih arhitektura GPU-a prikazani su neki od rezultata razlike prevođenja sjenčanja iz različitih igara sa optimizacijom i bez nje, gdje se može vidjeti čak i dvostruko manje zauzeće registara s optimizacijom nego bez nje. Na primjer, kod prevođenja jednog sjenčanja u igri Limbo s optimizacijama veličina koda je smanjena od 104 na 56 bajta i pritom je broj skalarnih registara je pao sa 22 na 11 i broj vektorskih sa 10 na 5.

Slijedio je prikaz razlika između pokretanja iste igre sa Vulkan-om i OpenGL-om. U toj usporedbi je Vulkan u igri Doom imao čak 44% više FPS-a nego OpenGL, a igra Mad Max s Vulkanom imala 14% više FPS-a od OpenGL-a. Osim toga, spomenule su se dvije hrvatske igre koje podržavaju vulkan, a od njih je upravo The Talos Principle bila prva igra uopće na svijetu koja je podržala Vulkan.

Kao zadnju usporedbu uspoređivao se Vulkan-ov LLVM i ACO gdje je u svih 8 igara Vulkan ACO imao manje zauzeće registara i posljedično bolje performanse u terminima FPS-a.

Za budući rad bilo bi zanimljivo izmjeriti promjenu performansi kroz verzije Mese, LLVM-a i ACO-a te istražiti koje su još moguće tehnike poboljšanja performansi u LLVM-u i ACO-u.

Literatura

[1] A. Shepherd, „What is a GPU?“, velj. 27, 2020. https://www.itpro.co.uk/hardware/30399/what-is-a-gpu.

[2] M. D., „Što je CPU odnosno procesor?“, sij. 25, 2018. https://geek.hr/pojmovnik/sto-je-cpu/.

[3] „What is GPU computing?“, 2020. https://www.boston.co.uk/info/nvidia-kepler/what-is-gpu-computing.aspx.

[4] M. J. Allen Sexton, „The 30 Year History of AMD Graphics, In Pictures“, kol. 19, 2017. https://www.tomshardware.com/picturestory/735-history-of-amd-graphics.html.

[5] „What is the difference between ISA and microarchitecture?“, 2020. https://iliketoknow.wordpress.com/what-is-the-difference-between-isa-and-microarchitecture/.

[6] ATI Technologies Inc., „ATI Technologies i“. svi. 10, 2006, [Na internetu]. Dostupno na: https://web.archive.org/web/20061005160511/http://www.ati.com/companyinfo/about/CorporateBackground.pdf.

[7] Study.com, „GPGPU: Definition, Differences & Example“, tra. 14, 2018. https://study.com/academy/lesson/gpgpu-definition-differences-example.html.

[8] Advanced Micro Devices, Inc., „AMD Graphics Cores Next (GCN) Architecture“. lip. 2012, [Na internetu]. Dostupno na: https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf.

[9] Advanced Micro Devices, Inc., „Leadership high-performance computing“, 2020, [Na internetu]. Dostupno na: https://ir.amd.com/static-files/fd06c15e-0241-424d-9fd9-5a469d96012d.

[10] J. M. P. Cardoso, „Embedded Computing for High Performance“, 2017, [Na internetu]. Dostupno na: https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data.

[11] M. Rouse, „VLIW“, ožu. 2011. https://whatis.techtarget.com/definition/VLIW-very-long-instruction-word.

[12] Advanced Micro Devices, Inc., *Introduction to AMD GPU Hardware*. 2020.

[13] Advanced Micro Devices, Inc., „Radeon: Dissecting the Polaris Architecture“. 2016, [Na internetu]. Dostupno na: https://www.amd.com/system/files/documents/polaris-whitepaper.pdf.

[14] Advanced Micro Devices, Inc., „‚Vega‘ Instruction Set Architecture“. srp. 28, 2017, [Na internetu]. Dostupno na: https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf.

[15] M. Wilbur, „Choose FP16, FP32 or int8 for Deep Learning Models“, tra. 04, 2020. https://software.intel.com/content/www/us/en/develop/articles/should-i-choose-fp16-or-fp32-for-my-deep-learning-model.html.

[16] D. Schurmann, „ACO, a new compiler backend for GCN GPUs“, velj. 10, 2019, [Na internetu]. Dostupno na: https://xdc2019.x.org/event/5/contributions/334/attachments/431/683/ACO_XDC2019.pdf.

[17] Advanced Micro Devices, Inc., „RDNA Architecture“, lip. 10, 2019, [Na internetu]. Dostupno na: https://gpureport.cz/info/Graphics_Architecture_06102019.pdf.

[18] K. Pulli, „New APIs for Mobile Graphics“, velj. 2013, [Na internetu]. Dostupno na: https://web.archive.org/web/20130502171637/http://research.nokia.com/files/pulli_spie06.pdf.

[19] „OpenGL Overview“, 2020. https://www.opengl.org/about/.

[20] J. de Vries, „Shaders“, lip. 17, 2020. https://learnopengl.com/Getting-started/Shaders.
[21] J. de Vries, „Hello Triangle“, lip. 17, 2020. https://learnopengl.com/Getting-started/Hello-Triangle.
[22] D. Lettier, „3D Game Shaders For Beginners“, 2019. https://lettier.github.io/3d-game-shaders-for-beginners/glsl.html.
[23] K. Trammell i J. Lampel, „Texturing“, 2015. https://cgcookie.com/course/introduction-to-texturing.
[24] D. Lettier, „Texturing“, 2019. https://lettier.github.io/3d-game-shaders-for-beginners/texturing.html.
[25] M. Bailey, „Vulkan and its GLSL Shaders“, tra. 03, 2020, [Na internetu]. Dostupno na: http://web.engr.oregonstate.edu/~mjb/cs557/Handouts/VulkanGlslShaders.6pp.pdf.
[26] Gizbot Bureau, „Difference between OpenGL and Vulkan“, ruj. 09, 2018. https://www.gizbot.com/computer/features/difference-between-opengl-vulkan-053725.html.
[27] M. Alka, „What is the difference between OpenGL and Vulkan?“, pros. 14, 2017. https://hashnode.com/post/what-is-the-difference-between-opengl-and-vulkan-cjb637szx01hrogwtw19x9voj.
[28] A. Overvoorde, „Base code“, 2016. https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Base_code.
[29] „GLFW“, *Vulkan guide*, sij. 09, 2020. https://www.glfw.org/docs/latest/vulkan_guide.html.
[30] Khronos Group Inc., „vkCreateInstance(3) Manual Page“, kol. 26, 2020. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/vkCreateInstance.html.
[31] A. Overvoorde, *Vulkan-tutorial*. 2020.
[32] A. Overvoorde, „Validation layers“, 2016. https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Validation_layers#page_What-are-validation-layers.
[33] A. Overvoorde, „Physical devices and queue families“, 2016. https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Physical_devices_and_queue_families.
[34] A. Overvoorde, „Setup Logical device and queues“, 2016. https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Logical_device_and_queues.
[35] A. Overvoorde, „Window surface“, 2016. https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Window_surface.
[36] E. P., „What Is a Swap Chain?“, kol. 24, 2020. https://www.wisegeek.com/what-is-a-swap-chain.htm.
[37] LunarG, Inc., „Create a Swapchain“, 2016. https://vulkan.lunarg.com/doc/view/1.2.135.0/linux/tutorial/html/05-init_swapchain.html.
[38] A. Overvoorde, „Pipeline introduction“, *Indtroduction*, 2016. https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction.
[39] LunarG, Inc., „SPIR-V“, 2015. https://www.khronos.org/registry/spir-v/papers/WhitePaper.html.
[40] A. Overvoorde, „Shader modules“, 2016. https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Shader_modules.
[41] Khronos Group Inc., „The OpenXR Specification“, kol. 14, 2020. https://www.khronos.org/registry/OpenXR/specs/1.0/html/xrspec.html.

[42] S. Kitching, *The Mine of Information*. 2015.
[43] R. E. Faith, „The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure“, stu. 05, 1999. http://dri.sourceforge.net/doc/drm_low_level.html.
[44] S. Singh, „Memory mapped I/O and Isolated I/O“, stu. 21, 2019. https://www.geeksforgeeks.org/memory-mapped-i-o-and-isolated-i-o/.
[45] W. Kenton, „First In, First Out (FIFO)“, sij. 29, 2020. https://www.investopedia.com/terms/f/fifo.asp.
[46] O. Pomerantz, „The Linux Kernel Module Programming Guide“, pros. 13, 2005. https://linux.die.net/lkmpg/x44.html.
[47] Advanced Micro Devices, Inc., „AMD Linux driver stack“, lis. 2014, [Na internetu]. Dostupno na: https://www.x.org/wiki/Events/XDC2014/XDC2014DeucherAMD/amdgpu_xdc_2014_v3.pdf.
[48] „Mesa Version History“, 2005. https://docs.mesa3d.org/versions.html.
[49] J. Edge, „The History of Mesa“, velj. 10, 2013. https://lwn.net/Articles/569083/.
[50] CodingUnit.com, „The History of OpenGL“, 2020. https://www.codingunit.com/the-history-of-opengl.
[51] C. Arthur Latter, „LLVM:AN INFRASTRUCTURE FOR MULTI-STAGE OPTIMIZATION“. 2002, [Na internetu]. Dostupno na: https://llvm.org/pubs/2002-12-LattnerMSThesis.pdf.
[52] A. Treat, „Qt4-preview-feedback Archive, February 2005 mkspecs and patches for LLVM compile of Qt4“, velj. 19, 2005. https://web.archive.org/web/20111004073001/http://lists.trolltech.com/qt4-preview-feedback/2005-02/msg00691.html.
[53] P. T. Robinson, „Developer Toolchain for PS4“, srp. 11, 2013, [Na internetu]. Dostupno na: http://llvm.org/devmtg/2013-11/slides/Robinson-PS4Toolchain.pdf.
[54] „The LLVM Compiler Infrastructure“, lip. 08, 2020. http://llvm.org/.
[55] S. Emms, „Clang“, 2020. https://www.linuxlinks.com/clang/.
[56] M. Larabel, „LLVM/Clang 3.2 Compiler Competing With GCC“, pros. 27, 2012. https://www.phoronix.com/scan.php?page=article&item=llvm_clang32_final.
[57] M. Lopez, „Comparison of Diagnostics between GCC and Clang“, kol. 19, 2017. https://gcc.gnu.org/wiki/ClangDiagnosticsComparison.
[58] T. Stellard, „A Detailed Look at the R600 Backend“, srp. 11, 2013, [Na internetu]. Dostupno na: https://llvm.org/devmtg/2013-11/slides/Stellard-R600.pdf.
[59] „User Guide for AMDGPU Backend“, lip. 09, 2020. https://llvm.org/docs/AMDGPUUsage.html#amdgpu-processor-table.
[60] P. Durand, „Instruction Set Architecture (ISA)“, kol. 29, 2006. http://www.cs.kent.edu/~durand/CS0/Notes/Chapter05/isa.html.
[61] „GCN Native ISA LLVM Code Generator“, 2020. https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Native-ISA.html.
[62] Phoronix Media, „RadeonSI“, *RadeonSI*, kol. 25, 2020. https://www.phoronix.com/scan.php?page=search&q=RadeonSI.
[63] M. Larabel, „AMD’s RadeonSI Driver Finally Enables OpenGL 4.6 But You Need To First Enable NIR“, stu. 27, 2019. https://www.phoronix.com/scan.php?page=news_item&px=RadeonSI-GL-4.6-NIR-Lands.
[64] J. Ekstrand, „NIR: A new compiler IR for Mesa“, 2013.

http://www.jlekstrand.net/jason/projects/mesa/nir-notes/.
[65] Phoronix Media, „RADV Radeon Vulkan Driver Submitted For Review To Be Included In Mesa“, ožu. 10, 2016. https://web.archive.org/web/20161104080446/https://phoronix-media.com/scan.php?page=news_item&px=RADV-Mesa-Submission-ML.
[66] D. Airlie, „Speaker - David Airlie“, sij. 13, 2020. https://linux.conf.au/speaker/profile/70/.
[67] J. Evangelho, „Valve’s Latest Linux Gaming Work Is Boosting AMD Vulkan Framerates By Up To 44 Percent“, stu. 07, 2019. https://www.forbes.com/sites/jasonevangelho/2019/07/11/valves-latest-linux-gaming-work-is-boosting-amd-vulkan-performance-by-up-to-44-percent/#570abf414e96.
[68] Debian, „Package: apitrace“, *Debian*, kol. 03, 2020. https://packages.debian.org/sid/apitrace.
[69] J. Fonseca, „Apitrace“, *Apitrace*, srp. 30, 2020. https://apitrace.github.io/.
[70] Alfonse, „Shader Compilation“, *Shader Compilation*, ruj. 16, 2017. https://www.khronos.org/opengl/wiki/Shader_Compilation.
[71] N. Malaya i N. Wolfe, „AMD GPU Hardware Basics“, lis. 2019. https://www.olcf.ornl.gov/wp-content/uploads/2019/10/ORNL_Application_Readiness_Workshop-AMD_GPU_Basics.pdf.
[72] „LLC“, *llc - LLVM static compiler*, lip. 09, 2020. https://llvm.org/docs/CommandGuide/llc.html.
[73] M. Larabel, „Mesa’s Gallium HUD Gets A Simple Option“, *Phoronix*, tra. 14, 2018. https://www.phoronix.com/scan.php?page=news_item&px=Gallium-HUD-Simple.
[74] L. Dawe, „MangoHud, a new open source Vulkan overlay layer for gaming on Linux“, *GamingOnLinux*, tra. 02, 2020. https://www.gamingonlinux.com/2020/02/mangohud-a-new-open-source-vulkan-overlay-layer-for-gaming-on-linux.
[75] Flightless Mango, „MangoHud“, *flightlessmango/mangohud*, lip. 09, 2020. https://github.com/flightlessmango/MangoHud.
[76] Flightless Mango, „Flightless Mango“, *Flightless Mango*, 2020. https://flightlessmango.com/.
[77] D. Brunner, „Frame Rate: A Beginner’s Guide“, *TechSmith*, ožu. 2017. https://www.techsmith.com/blog/frame-rate-beginners-guide/.
[78] R. Johnson, „Where minimum-FPS figures mislead, frame-time analysis shines“, *The Tech Report*, srp. 03, 2017. https://techreport.com/review/31546/where-minimum-fps-figures-mislead-frame-time-analysis-shines/.
[79] S. Stahie, „The Talos Principle Is the First Game Ready for Vulkan“, *Softpedia news*, velj. 17, 2016. https://news.softpedia.com/news/the-talos-principle-is-the-first-game-ready-for-vulkan-500531.shtml.
[80] NVidia Developer, „Get Started With Real-Time Ray Tracing“, 2020. https://developer.nvidia.com/rtx/raytracing.

Popis slika

Razlika između CPU i GPU....2
Prikaz od SP-a do izgleda jedne SIMD jezgre....5
Detaljan izgled jednog TeraScale GPU-a sa 10 SIMD (primjer ovakvog GPU-a je Radeon HD 4870 GPU)....6
Primjer VLIW-a....7
GNU - jedna SIMD jedinica....8
Izgled GCN CU-a....8
Promjena SIMD jezgre....11
RDNA CU....11
Pregled grafipčkog cjevovoda za trokut....24
Izgled grafičkog sučelja Apitrace-a....35
Apitrace - prozor s rezultatima za GPU i CPU vremenske okvire i pozive....36
Apitrace - prikaz samo označenih kadrova....36
Gallium HUD prikaz sa grafovima....47
Jednostavan prikaz Gallium HUD-a....48
Izgled MangoHUD-a....49
Doom - grafički prikaz FPS-a....51
Doom - grafički prikaz vremena sličica....52
Doom - prikaz minimalnog, maksimalnog i prosječnog FPS-a....52
Doom - ukupan prikaz rezultata....53
Mad Max - grafički prikaz FPS-a....53
Mad Max - grafički prikaz vremena sličica....54
Mad Max - prikaz minimalnog, maksimalnog i prosječnog FPS-a....54
Mad Max - ukupan prikaz rezultata....55
Lara Croft GO - prikaz minimalnog, maksimalnog i prosječnog FPS-a....55
Serious Sam Fusion 2017 - grafički prikaz FPS-a....56
Serious Sam Fusion 2017 - prikaz minimalnog, maksimalnog i prosječnog FPS-a....57
The Talos Principle - prikaz minimalnog, maksimalnog i prosječnog FPS-a....58

Popis tablica

Razlike između GCN i RDNA arhitekture....12
Usporedba OpenGL-a i Vulkan-a [27]....17
Doom - usporedba LLVM i ACO....59
Dota 2 - usporedba LLVM i ACO....60
Dota Underlords - usporedba LLVM i ACO....60
Mad Max - usporedba LLVM i ACO....61
Rise of the Tomb Rider - usporedba LLVM i ACO....61
Serious Sam Fusion 2017 - usporedba LLVM i ACO....62
The Talos Principle - usporedba LLVM i ACO....62
vkQuake - usporedba LLVM i ACO....63