SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD

Korištenje grafičkog procesora i DirectX programskog sučelja u evolucijskim algoritmitma

Ivan Jukić

Zagreb, lipanj, 2010.

SADRŽAJ

1. UVOD

Prema istraživanju [33], znanje ljudske vrste udvostručuje se svakih deset do petnaest godina. Znanje koje ljudi prikupe dolazi iz problema sa kojima se suočavaju i njihovih pokušaja da te probleme riješe. Ljudi na različite načine pristupaju rješavanju problema, no možemo reći da je svima zajednički cilj da se problemi riješe što prije i što učinkovitije, u odnosu na uložena sredstva. Ono što razlikuje probleme je njihova složenost, a složeniji problemi često izlaze iz okvira unutar kojih probleme možemo riješiti samo uporabom našeg intelekta. Kada su problemi dovoljno složeni okrećemo se drugim sredstvima koja nam mogu pomoći u rješavanju istih.

Genetsko programiranje je jedan od načina na koji možemo pristupiti rješavanju složenih problema i uz pomoć računala te probleme efikasno i brzo riješiti. No, je li to baš uvijek tako? Ponekad su problemi toliko složeni da čak i računala zahtijevaju velike količine vremena za njihovo rješavanje. Tada se pristupa kombiniranju različitih načina za rješavanje problema koji nam mogu pomoći da smanjimo vrijeme potrebno za pronalazak rješenja početnog problema. Paralelno programiranje je postupak pomoću kojeg možemo ubrzati izvođenje programa na računalima, a samim time i izvođenje različitih postupaka za rješavanje problema. Grafičko sklopovlje modernih računala pruža jednu visoko paralelnu platformu koja nam može omogućiti izvođenje paralelnih programa. Upravo zbog potrebe za bržim i efikasnijim rješavanjem problema, javila se ideja koja kombinira paralelno i genetsko programiranje, pri čemu se kao platforma za paralelno izvođenje programa koristi grafičko sklopovlje. Nova generacija Microsoftovog programskog sučelja DirectX za izradu grafičkih aplikacija donosi nam programe za izračune opće namjene (eng. *compute shader*) kao novo sredstvo pomoću kojega možemo paralelizirati aplikacije koristeći grafičko sklopovlje. Da li se programi za izračune opće namjene mogu primjeniti na postupke genetskog programiranja i koliki je stupanj paralelizacije moguće ostvariti samo su neka pitanja na koja se pokušava dati odgovore u ovom radu.

U drugom poglavlju ovog rada opisuje se paralelno programiranje i glavna svojstva koja paralelni programi sadrže. Opisuju se zakoni vezani uz paralelno programiranje, tipovi paralelizma, paralelno sklopovlje, te jezici koji podržavaju paralelno programiranje.

Treće poglavlje opisuje programe opće namjene koji se izvode na grafičkom procesoru. Opisuju se tehnike koje se koriste kod pisanja programa za grafičke procesore, daje se usporedba grafičkog i centralnog procesora, te na kojem modelu se temelji obrada podataka na grafičkom procesoru.

Četvrto poglavlje opisuje sučelje DirectX, daje kratki pregled razvoja tog sučelja i opisuje njegovu arhitekturu. Također se opisuje i način na koji se DirectX odnosi prema grafičkom sklopovlju, opisuje se HLSL (*High Level Shading Language*), te programi za izračune opće namjene (*compute shader*) koje izvodi grafičko sklopovlje.

U petom poglavlju dan je pregled genetskog programiranja. Ukratko se opisuju početci gentskog programiranja, zatim načini na koje se prikazuju i inicijaliziraju jedinke u populaciji, te osnovni genetski operatori (selekcija, križanje i mutacija).

U šestom poglavlju opisani su osnovni koncepti programiranja programa za izračune opće namjene (*compute shader*) pomoću sučelja DirectX, opisan je način prikaza podataka na grafičkom sklopovlju, te se opisuju izrađeni programi za izvođenje na grafičkom sklopovlju.

U sedmom poglavlju dani su rezultati testiranja programa izrađenih za grafičko sklopovlje pomoću sučelja DirectX i programa izrađenog za centralni procesor, uspoređuju se strukture programa i objašnjavaju dobiveni rezultati.

2. PARALELNO PROGRAMIRANJE

2.1. UVOD U PARALELNO PROGRAMIRANJE

Paralelno programiranje je tehnika pisanja programa kojom se omogućuje izvođenje različitih dijelova programa istodobno, a zasniva se na činjenici da se veliki problemi često mogu podijeliti u manje probleme koji se tada mogu riješavati istodobno. Paralelno programiranje i paralelizam koristi se već duži niz godina, uglavnom u računarstvu visokih performansi, međutim u zadnje vrijeme zanimanje za paralelno programiranje i paralelne algoritme raste. Razlog tome su fizička ograničenja današnjih integriranih krugova koja onemogućuju daljnje povećanje brzine rada (frekvencije) današnjih centralnih procesora. Paralelne računalne arhitekture su zadnjih nekoliko godina postale dominantna paradigma razvoja novog sklopovlja i to najčešće u obliku višejezgrenih centralnih procesora.

Računala koja podržavaju paralelizam, mogu se grubo podijeliti na razine na kojima podržavaju paralelizam. Višejezgreni i višeprocesorski sustavi sadrže više procesnih elemenata unutar jednog računala, dok grozdovi i sustavi sa stotinama ili tisućama procesnih jedinica (MPP – *Massive parallel processing*) koriste različita računala kako bi radili na istom zadatku [1]. Klasični način programiranja je slijedno, kod kojeg se u jednom vremenskom trenutku izvršava jedna naredba, a nakon što se ona izvršila kreće naredba koja sljedeća po redu. Te naredbe se izvršavaju na centralnom procesoru jednog računala. Pisanje paralelnih programa zahtjevnije je nego pisanje slijednih programa. Paralelni programi koriste više procesnih jedinica istovremeno kako bi riješili neki problem. Svaka ta procesna jedinica izvršava svoj dio programa paralelno sa drugim procesnim jedinica, s obzirom da su ti dijelovi programa međusobno nezavisni.

Od sredine 80-ih godina pa sve do 2004. povećanje frkvencije rada centralnih procesora bio je glavni razlog zbog poboljšanja računalnih performansi. Trajanje programa jednako je broju instrukcija pomnoženo sa prosječnim trajanjem jedne instrukcije, dakle zadržavajući jednak broj instrukcija, a smanjujući prosječno trajanje jedne instrukcije (povećavanjem frekvencije procesora) mogli smo smanjiti ukupno trajanje izvršavanja

programa. Ipak, to je dovodilo i do povećanja potrošnje energije, a samim time i do povećanja disipacije topline na tim procesnim jedinicama. Potrošnju energije možemo opisati jednadžbom (1) [1]:

$$P = C \times V^2 \times F \qquad (1)$$

gdje *P* predstavlja snagu, *C* je kapacitet koji se stvara svakog procesorskog ciklusa, *V* je napon, a *F* je brzina procesora (broj ciklusa u sekundi). Povećanje broja ciklusa povećava i potrebnu energiju procesora.

2.1.1. MOOREOV ZAKON

Jedan od osnivača Intela, Gordon E. Moore, davne 1965. godine dao je jedan od danas najpoznatijih zakona vezanih uz broj tranzistora u integriranim krugovima. Taj zakon danas poznat kao Mooreov zakon kaže da će se svakih 18 do 24 mjeseca broj tranzistora u intregriranim krugovima udvostručavati, uz minimalne troškove komponenti [2]. Mooreov zakon i danas vrijedi, iako se brzine procesnih jedinica više ne povećavaju, broj tranzistora se i dalje udvostručuje, a ti dodatni tranzistori koriste se u novom sklopovlju koje pruža mogućnosti paralelnog izvršavanja programa i obrade podataka (npr. dodatne jezgre procesora).

2.1.2. AMDAHLOV ZAKON

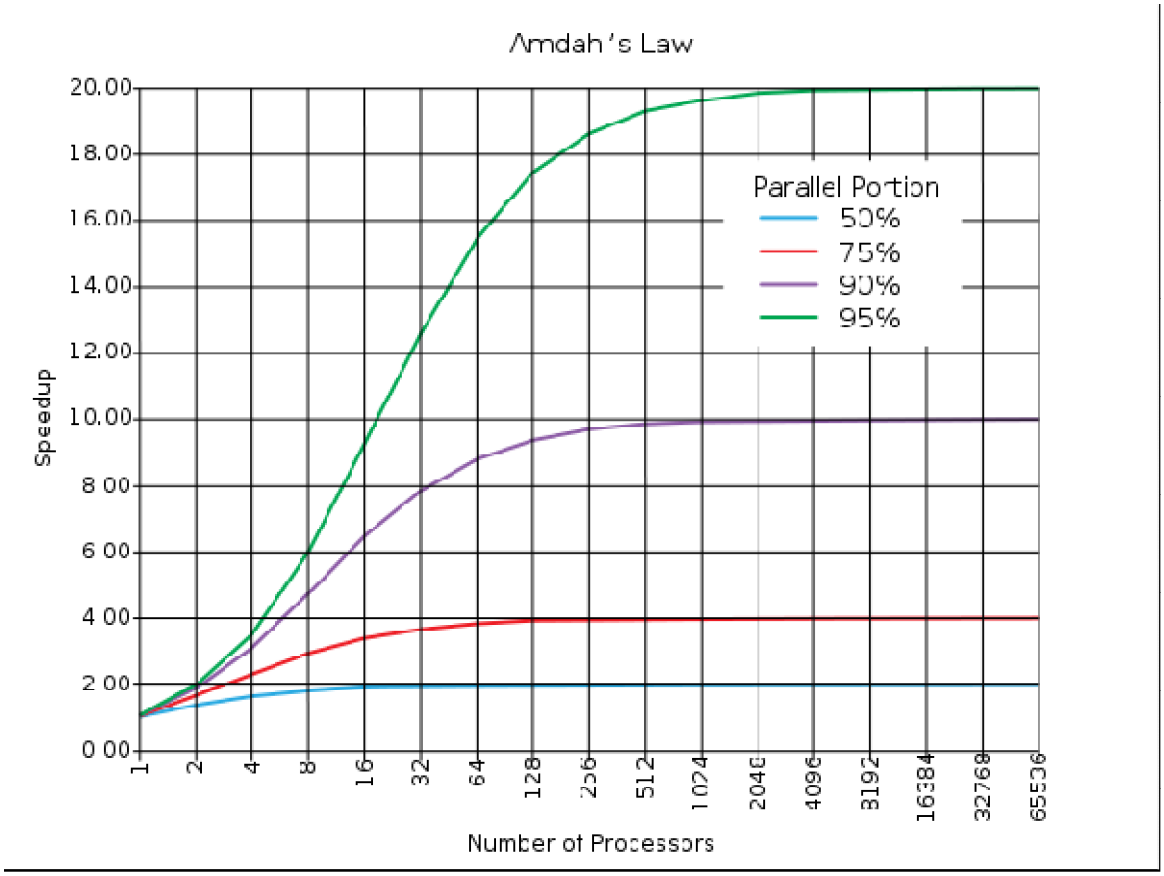
Optimalno, paralelni algoritmi bi dodavanjem novog procesnog sklopovlja dobivali linearno ubrzanje, tj. ukoliko dodamo još jednu procesnu jedinicu u sustav u kojem se nalazi već jedna takva jedinica, vrijeme izvođenja paralelnog algoritma smanjilo bi se za faktor dva. Međutim rijetki paralelni algoritmi postižu linearno ubrzanje. Većina dobija ubrzanje blisko linearnome za male količine novih procesnih elemenata, a dodavanjem većeg broja procesnih elemenata ubrzanje se približava konstantnoj vrijednosti. Potencijalno ubrzanje paralelnog algoritma dano je Amdahlovim zakonom, kojeg je formulirao arhitekt računala Gene Amdahl 60-ih godina 20-og stoljeća. Taj zakon kaže da

će dijelovi programa koji se ne mogu paralelizirati ograničiti ukupno moguće ubrzanje paralelizacijom. Bilo koji veliki matematički ili inženjerski problem tipično se sastoje od dijelova koji se mogu paralelizirati i onih koji ne mogu, a taj odnos dan je formulom (2):

$$SU = \frac{1}{(1-P)} \qquad (2)$$

SU je iznos ubrzanja, a *P* je dio programa koji se može paralelizirati [3].

Uzmimo na primjer, ako program treba 20 sati za svoje izvođenje na jednoprocesorskom sustavu, a određeni dio programa koji se izvodi jedan sat ne može se paralelizirati, tada bez obzira na to koliko dodatnih procesora koristili, program se ne može izvesti ispod vremena od jednog sata. Ostalih 19 sati se može paralelizirati, a to iznosi 95% programa. Korištenjem jednadžbe (2) dobivamo iznos ubrzanja koje možemo postići u ovom slučaju i koje iznosi 20 puta manje vrijeme izvođenja algoritma ukoliko se on paralelizira. Potencijalna ubrzanja za različite postotke paralelnosti programa prikazana su na slici 1.



Slika 1 (preuzeto sa www.wikipedia.org) - Prikazuje ubrzanje programa za različite vrijednosti iznosa paralelnog dijela programa u odnosu na dostupan broj procesnih elemenata

U slučaju paralelizacije, ako je *P* dio programa koji se može paralelizirati, a *(1-P)* je dio koji je slijedni, tada maksimalno ubrzanje koje možemo postići uporabom *N* procesnih jedinica možemo dobiti preko jednadžbe (3):

$$SU = \frac{1}{(1-P) + \frac{P}{N}} \qquad (3)$$

Ukoliko pogledamo limes ove jednadžbe sa *N* koji ide u beskonačno, tada jednadžba (3) teži u jednadžbu (2). Ukoliko opet uzmemo u obzir prethodni primjer i uvrstimo u ovu jednadžbu 0.95 za *P* i proizvoljan *N*, maksimalno ubrzanje i dalje ostaje 20 bez obzira koji *N* koristili. Iz ovog razloga paralelno izvođenje programa korisno je za male brojeve procesora, ili za programe koji imaju visoku vrijednost parametra *P.*

2.1.3. GUSTAFSONOV ZAKON

Gustafsonov zakon (također poznat i kao Gustafson-Barsisov zakon) [4] je zakon iz računarskih znanosti koji kaže da se bilo koji dovoljno veliki problem može efikasno paralelizirati. Gustafsonov zakon blisko je povezan sa Amdahlovim zakonom koji daje limit do kojeg se neki program može paralelizirati. Gustafsonov zakon opisan je jednadžbom (4):

$$S(P) = P - \alpha \cdot (P - 1) \qquad (4)$$

gdje je *P* broj procesnih jedinica, *S* je ubrzanje, a α je dio problema koji se ne može paralelizirati. Gustafsonov zakon ispravlja nedostatak Amdahlovog zakona koji se temelji na problemu fiksnog opterećenja i koji implicira da se slijedni dio programa ne mijenja u odnosu na veličinu sustava, a paralelni dijelovi su jednako raspodijeljeni na *N* procesora. Gustafsonov zakon uklanja problem fiksne veličine i umjesto njega predlaže koncept fiksnog vremena kojim se opisuju dobitci u ubrzanju za veće probleme.
U istraživanjima su se, na temelju Gustafsonovog zakona, mnogi problemi reformulirali tako da bi u jednakom vremenu bilo moguće rješavanje većeg problema, a koncept učinkovitosti je reformuliran kao potreba da se smanji slijedni dio programa čak i ako se time poveća ukupna količina računanja.

2.2. TIPOVI PARALELIZMA

Razumijevanje odnosa podataka i njihovih međuovisnosti vrlo je bitna za paralelno programiranje. Nijedan program ne može se izvesti kraće od vremena potrebnog da se izvede najduži lanac međusobno zavisnih naredbi. Ipak, većina programa se ne sastoji samo od dugih lanaca međusobno zavisnih naredbi, već postoje i mogućnosti da se nezavisni lanci izvedu paralelno.

U paralelnom programiranju, zadatci se često dodijeljuju različitim dretvama na izvođenje, kao što ćemo kasnije vidjeti u primjerima aplikacija pisanim za grafičke procesore. Neke paralelne arhitekture koriste manje i jednostavnije verzije dretvi, a zovu se niti (eng. *Fiber* - vlakno, nit), dok druge arhitekture koriste procese. U paralelnom programiranju često se događa da dretve dijele neke varijable i trenutci u kojima pristupaju tim varijablama mogu biti isprepleteni između dretvi. To može dovesti do pojave koja se naziva „opasnost utrke“ (eng. *race condition, race hazard*), kada je konačna vrijednost varijable izrazito zavisna od redosljeda kojim se toj varijabli pristupa [5]. Kako bi se to izbjeglo, programer mora pružiti mehanizam za međusobno isključivanje (eng. *mutual exclusion*), pri čemu se jednoj dretvi dopušta pristup varijabli, dok se ostalim dretvama onemogućuje pristup toj varijabli. Dretva koja je zaključala varijablu slobodno izvršava operacije nad tom varijablom (čitanje i pisanje), a kada završi otključava je i omogućuje pristup novoj dretvi. Ukoliko dretva treba zaključati više od jedne varijable pomoću skupa operacija koje se ne mogu kombinirati postoji opasnost od potpunog zastoja (eng. *deadlock*). Operacije koje se mogu kombinirati (eng. *atomic operations* [6]) ostatku sustava izgledaju kao jedna operacija sa jednim mogućim završetkom, uspjeh ili neuspjeh. Recimo da jedna dretva treba zaključati iste dvije varijable kao i druga dretva. Ukoliko svaka dretva zaključa po jednu varijablu dolazi do potpunog zastoja, jer im je onemogućen pristup onoj drugoj varijabli i one se ne mogu izvršiti. Korištenjem *atomic* operacija neće se zaključati niti jedna varijabla ukoliko se ne mogu odjednom zaključati sve potrebne varijable. Međusobno isključivanje, iako nužno za ispravan rad programa, može znatno usporiti njegov rad.

Međutim, sve paralelizacije ne rezultiraju smanjenjem vremena potrebnog za izvršavanje

programa. Općenito, kako se zadatak dijeli na više dretvi, one potroše veliki dio vremena međusobno komunicirajući. Konačno, utrošak vremena na dodatnu komunikaciju postaje veći od vremena potrebnog za obavljanje zadatka i daljnja paralelizacija povećava vremenski trošak umjesto da ga smanjuje. Ovo je poznato kao paralelno usporavanje (eng. *parallel slowdown*).

Na temelju količine komunikacije i sinkronizacije podzadataka unutar programa, možemo ih klasificirati u programe koji prikazuju krupno-zrnatu paralelizaciju, sitno-zrnatu paralelizaciju i programe koji su „trivijalno“ paralelni (eng. *embarrassingly parallel*). Teksta programa krupno-zrnate paralelizacije dretve međusobno rijetko komuniciraju tijekom jedne sekunde, teksta programa sitno-zrnate paralelizacije dretve često komuniciraju tijekom jedne sekunde, a teksta programa „trivijalno“ paralelnih programa dretve jako rijetko komuniciraju ili nikada ne komuniciraju, pa se smatra da je takve programe najlakše paralelizirati.

2.2.1. PARALELIZAM NA RAZINI BITOVA (*bit-level parallelism*)

Od početaka proizvodnje računalnih čipova sa visokim stupnjem integracije 1970-ih godina, pa sve do 1986. godine, ubrzanja računalnih komponenti temeljila su se i na povećavanju računalnih riječi, tj. količini informacija kojima procesor može manipulirati po jednom ciklusu. Povećanjem računalne riječi smanjuje se broj instrukcija koje procesor mora izvesti kako bi obavio određenu operaciju nad varijablama koje su duže od procesorske riječi. Na primjer 8-bitni procesor koji zbraja dvije 16-bitne cjelobrojne varijable prvo mora zbrojiti donjih 8 bitova koristeći standardnu instrukciju za zbrajanje, zatim treba zbrojiti gornjih 8 bitova koristeći instrukciju za zbrajanje koja dodaje i prijenos (eng. *carry*) prethodnog zbrajanja. Dakle vidimo da 8-bitnom procesoru trebaju dvije instrukcije za zbrajanje 16-bitnih cjelobrojnih varijabli, dok bi 16-bitnom procesoru trebala samo jedna instrukcija.

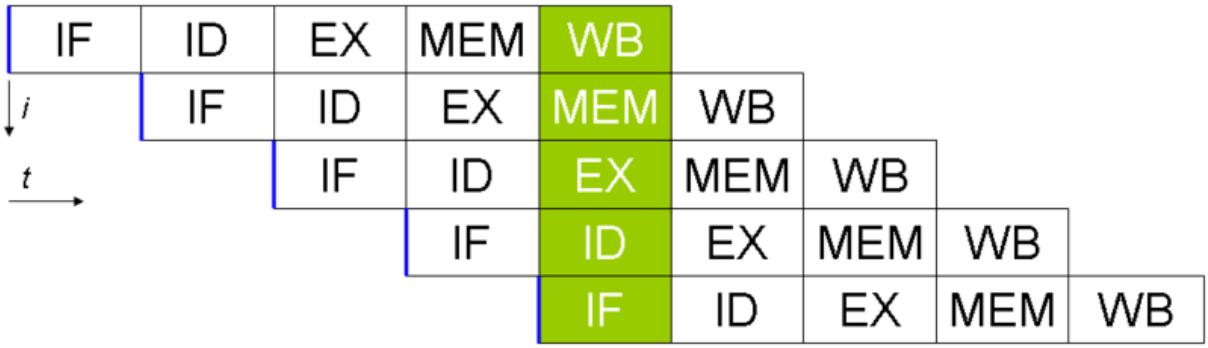
Povijesno gledano ovaj trend je bio naglašen od prvih 4-bitnih procesora do pojave prvih 32-bitnih procesora sredinom 1980-ih godina. 32-bitni procesori bili su standard sljedećih 20ak godina, do otprilike 2003.-2004. kada se sve više pojavljuju 64-bitni procesori.

Otprilike 55% svih procesora koji se prodaju u svijetu su 8-bitni mikrokontroleri, manje od 10% procesora koji se prodaju su 32 ili više bitni [7].

2.2.2. PARALELIZAM NA RAZINI INSTRUKCIJA (*instrucion-level parallelism*)

Računalni program je u biti niz instrukcija koje procesor izvodi. Te instrukcije se mogu poredati proizvoljnim redosljedom i kombinirati u gupe instrukcija koje se zatim mogu izvesti u paraleli ne mijenjajući konačni ishod programa. Ovo je poznato kao paralelizam na razini instrukcija. Napretci u ovom području paralelizma bili su naglašeni od sredine 1980-ih godina pa do sredine 1990-ih godina.

Današnji moderni procesori imaju višerazinske instrukcijske cjevovode. Svako stanje u tom cjevovodu odgovara drugoj akciji koju taj procesor izvodi za određenu instrukciju. Jedan od primjera višerazinskog cjevovoda je i RISC procesor sa pet stanja prikazan na slici 2. dohvaćanje instrukcije (IF - *instruction fetch*), deteksta programairanje instrukcije (ID - *instruction decode*), izvršavanje instrukcije (EX - *execute*), pristup memoriji (MEM - *memory access*) i zapis u registar (WB-*register write back*).



Slika 2 (preuzeto sa www.wikipedia.org) - Pet stanja jednosravnog RISC procesora: dohvaćanje instrukcije (IF - instruction fetch), deteksta programairanje instrukcije (ID - instruction decode), izvršavanje instrukcije (EX - execute), pristup memoriji (MEM - memory access) i zapis u registar (WB-register memory access).

Uz ovaku paralelizaciju na razini instrukcijskog cjevovoda napravljeni su procesori koji mogu obrađivati i više od jedne instrukcije istovremeno, a nazivaju se superskalarni (eng.

superscalar) procesori. Instrukcije se mogu grupirati i izvesti istovremeno ako između njih ne postoji međuovisnosti.

2.2.3. PODATKOVNI PARALELIZAM (*data parallelism*)

Podatkovni paralelizam [8], koji je prisutan u programskim petljama, je paralelizam fokusiran na raspodjelu podataka različitim procesnim čvorovima, tako da se ti podatci mogu paralelno obraditi. Paralelizacija programskih petlji često dovodi do vrlo sličnih slijedova funkcija koje se izvode nad elementima neke velike podatkovne strukture. Mnoge znanstvene i inženjerske aplikacije temelje se na podatkovnoj paralelizaciji. Međutim, ukoliko poziv funkcija unutar neke petlje ovisi o nekom prethodnom izvršavanju te petlje, tada paralelizacija neće biti moguća.

U multiprocesorskim sustavima podatkovni paralelizam se postiže tako da svaka procesna jedinica izvršava isti skup operacija nad različitim skupovima distriburianih podataka . U nekim situacijama jedna dretva kontrolira operacije nad svim skupovima podataka, dok u drugim različite dretve kontroliraju operacije, ali sve izvršavaju isti dio programa.

2.2.4. PARALELIZAM ZADATAKA (*task parallelism*)

Paralelizam zadataka [9] je oblik paralelizacije računalnog programa na više procesnih elemenata u okruženjima za paralelno izvršavanje programa. Paralelizam zadataka se fokusira na raspodjeljivanje izvršnih procesa (dretvi) na različite procesne elemente. Paralelizacija zadataka se može smatrati kao kontrast paralelizaciji podataka. U višeprocesorskim sustavima paralelizacija na razini zadataka se postiže tako da se različite dretve ili procesi izvršavaju nad istim ili različitim podatcima. Te dretve mogu sadržavati isti ili različiti tekst programa. One ujedno i komuniciraju međusobno, najčešće kako bi predale podatke jedna drugoj kao dio sljedećeg stupnja u obradi podataka.

2.3. PARALELNO SKLOPOVLJE

2.3.1. MEMORIJA I KOMUNIKACIJA

Glavna memorija u paralelnim sustavima može biti dijeljena memorija koju dijele svi procesi ili dretve unutar jednog adresnog prostora, ili raspodijeljena memorija unutar koje svaki procesni element ima svoj adresni prostor. Raspodijeljena memorija se najčešće odnosi na činjenicu da je memorija logički raspodijeljena, no može biti i fizički raspodijeljena. Ova dva pristupa mogu se i kombinirati, tako da nije neobično da procesni element ima svoju lokalnu memoriju i pristup dijeljenoj memoriji. Pristupi lokalnoj memoriji procesora su tipično brži nego pristupi dijeljenoj memoriji.

Računalne arhitekture u kojima se svakom elementu u glavnoj memoriji može pristupiti sa jednakom latencijom (*Column Address Strobe* [CAS] latencija - vrijeme od trenutka kada memorijski kontroler zatraži od memorijskog modula dohvat podataka u određenom memorijskom stupcu do trenutka kada su ti podatci dostupni na priključcima tog memorijskog modula) i širinom memorijskog pojasa poznate su kao sustavi sa jednolikim pristupom memoriji (UMA - *Uniform Memory Access*). Ovakvi sustavi sadrže dijeljenu memoriju i ta memorija nije fizički podijeljena već samo logički. Sustavi koji imaju različitu latenciju i širinu memorijskog pojasa nazivaju se sustavi sa nejednolikim pristupom memoriji (NUMA - *Non-Uniform Memory Access*). Sustavi sa distribuiranom memorijom imaju nejednolik pristup memoriji.

Svaki računalni sustav koristi male količine brze memorije poznate kao priručna ili *cache* memorija, a služi za pohranjivanje vrijednosti koje se često koriste, čime se izbjegava prečesto pristupanje glavnoj memoriji. Paralelni sustavi imaju problema sa priručnom memorijom budući da se ista vrijednost sprema na više mjesta gdje se može lokalno mijenjati i u konačnici rezultirati neispravnim radom programa. Zbog toga paralelni sustavi trebaju podsustav koji će održavati koherenciju podataka između različitih priručnih memorija. Takvi sustavi (eng. *cache coherency systems*) prate promjene vrijednosti koje se nalaze u priručnim memorijama različitih procesnih elemenata, te ih konstantno osvježavaju osiguravajući na taj način ispravan rad programa.

2.3.2. RAZREDI PARALELNIH SUSTAVA

Paralelni sustavi mogu se ugrubo podijeliti u razredi na temelju razine na kojoj sklopovlje podržava paralelizaciju. Ova klasifikacija analogna je udaljenosti između osnovnih procesnih elmenata.

a) VIŠEJEZGRENI PROCESORI

Višejezgreni procesor je procesor koji na jednoj pločici uključuje više izvršnih jedinica, u ovom slučaju procesorskih jezgri. Ovi procesori imaju dodirnih točaka sa superskalarnim procesorima, i jedni i drugi mogu u jednom procesorskom ciklusu obrađivati više od jedne instrukcije iz skupa instrukcija, međutim superskalarni procesori te instrukcije dohvaćaju iz istog skupa (jedna dretva), dok višejezgreni procesor može dohvatiti instrukcije iz različitih skupova (različite dretve). Moguća je kombinacija u kojoj je svaka jezgra zasebno superskalarni procesor i može dohvatiti više od jedne instrukcije iz istog skupa instrukcija.

Jedan od ranijih pokušaja izrade pseudo-višejezgrenih procesora je bila izrada procesora sa podrškom za istovremenu višedretvenost (eng. *simultaneous multithreading*). U ovom slučaju, procesor obrađuje dretvu do trenutka kada uđe u prazni hod (eng. *idle*), npr. zbog dohvaćanja instrukcije ili podataka iz memorije, tada se aktivira druga dretva i procesor kreće sa njenim obrađivanjem. Današnji moderni višejezgreni procesori također nasljeđuju ovaj način rada, povećavajući time svoje mogućnosti (npr. Intel i7 procesori).

b) SIMETRIČNO MULTIPROCESIRANJE

Simetrično multiprocesiranje (eng. *symmetric multiprocessing*, SMP) je multiprocesorski računalni sustav u kojem se nalazi veći broj identičnih procesora koji dijele memoriju i povezani su preko sabirnice [10]. Većina današnjih višeprocesorskih sustava koriste SMP arhitekturu, a u slučaju višejezgrenih procesora SMP arhitektura se primjenjuje na jezgre, tretirajući svaku jezgru kao zasebni procesor. SMP sustavi omogućuju bilo kojem procesoru da radi na bilo kojem zadatku bez obzira na to u kojem se dijelu memorije nalaze podatci, uz uvijet da se isti zadatak ne izvršava na dva ili više procesora u isto vrijeme. Uz pravilu podršku od strane operacijskog sustava,

SMP sustavi mogu jednostavno prebacivati zadatke između različitih procesora i tako efikasno balansirati sa opterećenjem pojedinih procesora.

Sabirnica kojom su procesori procesori povezani predstavlja usko grlo budući da sa povećanjem broja procesora može doći do stanja na sabirnici u kojem više procesora istovremeno želi postaviti određenu vrijednost na sabirnicu, što može rezultirati greškama u radu ili čak fizičkim oštećenjima sabrinice. Arbitracijski protokoli koji dodjeljuju sabirnicu paze da do navedenog stanje ne dođe.Međutim u slučaju logičkih grešaka, proizvodnih defekata ili rada sabirnice na većim frekvencijama od predviđenog može doći do grešaka i protokoli mogu prestati raditi ispravno. Zbog toga sustavi sa SMP arhitekturom općenito ne sadrže više od 32 procesora. Isto tako, zbog male veličine procesora i smanjenja potrebe za velikom memorijskom propusnosti povećanjem priručne memorije takvi multiprocesorski sustavi mogu biti vrlo efikasni, uz uvijet da je propusnost na sabirnici dovoljno velika.

c) RASPODIJELJENI SUSTAVI

Raspodijeljeni sustav je sustav u kojem se nalazi veći broj autonomnih računala (procesnih jedinica) koja komuniciraju putem računalne mreže. Računalni program koji se izvršava u ovakvom okruženju naziva se raspodijeljeni program. Glavna prednost ovakvih sustava je velika skalabilnos. Klasteri računala su jedan od primjera distribuiranog sustava. Klaster je skup samostalnih računala koja su usko povezana i komuniciraju putem računalne mreže. Za računala koja se nalaze u klasteru uobičajeno je da budu simetrična (da se sastoje od istih procesnih elemenata), jer je tada uravnotežavanje opterećenja najjednostavnije za izvesti. Grozdovi računala su također primjer distribuiranog sustava i ovakvi sustavi pokazuju najveći stupanj distribuiranosti. Računala u grozdovima koriste Internet za međusobnu komunikaciju. Zbog male propusnosti i visokih latencija na internetu, grozdovi računala se uglavnom bave problemima koji su „trivijalno“ paralelni.

d) SPECIJALIZIRANO SKLOPOVLJE

U paralelnom računarstvu postoji i sklopovlje specijalizirane namjene koje možemo smatrati posebnom klasom. Iako takvo sklopovlje nije nužno povezano samo sa jednom domenom rada, uglavnom se koristi za manji broj problema paralelizacije.

2.4. PROGRAMSKI JEZICI ZA PARALELNO PROGRAMIRANJE

Za paralelne programe razvijeni su jezici za paralelno programiranje, različite knjižnice funkcija, programska sučelja i različiti programski modeli. Njih možemo podijeliti u različite razrede ovisno o tome na kojim pretpostavkama se temelje, tj. da li sustav ima dijeljenu ili distribuiranu memoriju ili kombinaciju. Programi pisani jezicima za sustave sa dijeljenom memorijom rade tako da međusobno komuniciraju i manipuliraju varijablama u dijeljenoj memoriji. Programi pisani jezicima za sustave sa distribuiranom memorijom međusobno razmjenjuju poruke (eng. *message passing*). Standard *POSIX Threads* i *OpenMP* su dva programska sučelja najčešće korištena za sustave sa dijeljenom memorijom, dok je MPI (*Message Passing Interface*) najčešće korišteno programsko sučelje za razmjenu poruka u raspodijeljenim sustavima. Za paralelne programe karakterističan je koncept budućeg vremena, kada jedan dio programa mora drugom dijelu programa dostaviti potrebne podatke u nekom budućem vremenu.

Automatsku paralelizaciju slijednog programa prilikom prevođenja (eng. *compiling*) možemo smatrati konačnim ciljem u području paralelnog programiranja. Iako znanstvenici već desetljećima rade na paralelizaciji prilikom prevođenja uspjeh je bio prilično ograničen.

3. PROGRAMI OPĆE NAMJENE ZA GRAFIČKI PROCESOR

3.1. UVOD U PISANJE PROGRAMA OPĆE NAMJENE ZA GPU

Pisanje programa opće namjene za grafički procesor (eng. *General-Purpose computing on Graphics Processing Unit*) kao izraz označava jedan od novijih smjerova u razvoju aplikacija. To je posebna tehnika pisanja programa opće namjene (programa tradicionalno pisanih za centralni procesor) koji se izvode na grafičkom procesoru [11]. Iako u ranijim razdobljima razvoja grafičkog sklopovlja to nije bilo moguće, nakon dodavanja dodatnih programibilnih stanja u grafički cjevovod i povećanjem preciznosti aritmetike samog grafičkog procesora javila se ideja o iskorištavanju grafičkog procesora za proračune opće namjene. Prvi pokušaji u ovom području pojavili su se 2005.-2006. godine izlaskom grafičkog sklopovlja sa podrškom za *Shader Model 3.0* [12] (Shader model označava stupanj u razvoju grafičkog sklopovlja i mogućnostima koje pruža) i iako je potencijal uvijek bio velik, ovo je bilo područje od interesa za vrlo mali broj istraživača koji su ovu tehniku koristili za vrlo specifične primjene. Jedan od problema je bio i taj što su se morala koristiti programska sučelja za računalnu grafiku, poput Microsoftovog DirectX-a ili OpenGL-a. Ovi API su bili dizajnirani za grafičke proračune, a ne za proračune opće namjene, pa kako bi u potpunosti iskoristili mogućnosti koje je grafičko sklopovlje pružalo bilo je potrebno koristiti sučelja koja su bila na nižoj razini od navedenih (*low-level* pristup sklopovlju). Ovo je rezultiralo u API-jima koji su bili specifični za proizvođače grafičkog sklopovlja, poput ATI CTM (*Close-to-the-Metal*, doživio samo beta izdanje) i ATI Stream SDK ili Nvidia Cuda.

3.2. GPU NASUPROT CPU

Performanse grafičkih procesora rastu zapanjujućom brzinom, zapravo u posljednjem desetljeću stopa rasta performansi grafičkog procesora bila je veća od one definirane sa Mooreovim zakonom, koji zapravo definira stopu rasta broja tranzistora u centralnom procesoru. Kako bi shvatili taj veliki potencijal i zašto se grafički procesor može vrlo jako

približiti svom teoretskom maksimumu, pogledajmo u čemu se on razlikuje od standardnog centralnog procesora, koji vrlo rijetko postiže performanse bliske teoretskim. Razlika je u načinu kojim se pristupa operacijama sa velikom latencijom, osobito memorijski pristup. Kada centralni procesor (CPU) treba vrijednost iz memorije on uđe u prazan hod i čeka dok ne primi zatraženu varijablu. Kako bi izbjegli prazan hod tijekom kojeg CPU čeka na varijablu iz memorije, centralni procesori imaju velike količine priručne memorije, zapravo sadrže više memorije nego logike za računanje. Grafički procesori (GPU) sa druge strane imaju jako malo priručne memorije i puno logičkih elemenata za računanje. Priručna memorija na GPU se koristi za poboljšanje performansi lokaliziranih pristupa memoriji, međutim veći dio latencije koji iz toga proizlazi se skriva kroz višedretvenost. Općenito, kada dretva koju GPU izvodi dođe do trenutka kada treba podatak iz memorije GPU neće čekati da taj podatak pristigne, već će početi obrađivati drugu dretvu i kada ta dretva dođe do trenutka kada treba podatak iz memorije GPU će se prebaciti na sljedeću dretvu itd. U jednom trenutku biti će aktivan maksimalan broj dretvi koje GPU može obrađivati, te će se GPU vratiti na onu početnu dretvu. U tom trenutku vrlo je vjerojatno da je podatak dohvaćen i spremljen u odredišni registar. Zbog ovog pristupa GPU treba veliki skup registara koji sadrže vrijednosti svih aktivnih dretvi. Programi koji se izvode na grafičkom procesoru nazivaju se *shaderi*, i što je veći broj registara koje jedan takav program koristi, to je manji ukupni broj aktivnih dretvi koje GPU odjednom može podržati i smjestiti njihove podatke u taj skup registara. Stoga, broj registara na grafičkom sklopovlju može imati znatan utjecaj na performanse budući da manji ukupni broj aktivnih dretvi znači manju vjerojatnost da je ona početna dretva koje je čekala na vrijednost iz memorije dobila tu vrijednost u trenutku kada se vratimo natrag na njeno izvršavanje.

Ovaj način rada grafičkog procesora prenesen je i na centralni procesor, najpoznatija je *Intel HyperThreading* tehnologija koja također skriva latencije tako da se procesor prebacuje na obrađivanje druge dretve umjesto da čeka na podatak iz memorije. Zbog toga se jedna jezgra čini kao dvije logičke jezgre jer može obrađivati dvije dretve i prebacivati izvođenje između njih kada jedna dođe do trenutka gdje zahtjeva vrijednost iz memorije. Veliki skup registara grafičkog procesora također ima svoj ekvivalent u obliku SPE (*Synergistic Processing Elements*) u Cell procesoru. Cell procesor nema automatsko

manipuliranje dretvama kao grafički procesori, no takvo se ponašanje može emulirati.

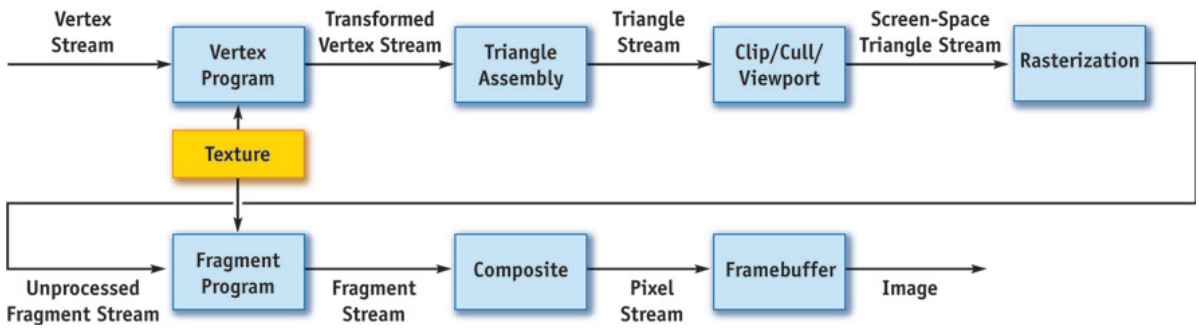
Dretve na grafičkom procesoru su instance koje izvršavaju programe na grafičkom procesoru, npr. piksel u slučaju programa za obradu piksela ili vrh u slučaju programa za obradu vrhova. Iako su grafički procesori oduvijek na ovaj način manipulirali dretvama to je bilo implicitno i skriveno od samog programera, budući da su dretve ionako uvijek bile nezavisne međusobno.

3.3. OBRADA TOKA PODATAKA

Model obrade toka podataka [13][14] danas je temelj za izradu programa namijenjenih za grafički procesor. Glavno svojstvo modela obrade toka podataka je da se program strukturira na način da se postigne velika učinkovitost u smislu izračunavanja i komunikacije sa memorijom. U modelu obrade toka podataka svi podatci prikazani su kao tok (eng. *stream* - tok), koji definiramo kao uređeni skup podataka istog tipa. Ti podatci mogu biti jednostavni kao npr. tok cjelobrojnih varijabli ili varijabli sa pomičnim zarezom, ali mogu biti i složeni kao što je tok točaka, trokuta ili transformacijskih matrica. Iako ti tokovi mogu biti različitih duljina, operacije koje se izvode nad podatcima u toku su najučinkovitije ako je taj tok dugačak (100 ili više elemenata u toku). Operacije nad podatcima u tokovima izvode programske jezgre (*kernels*, možemo ih shvatiti kao funkcije), ukoliko su podatci u tokovima vrhovi i fragmenti, tada su programi za obradu vrhova i piksela jezgre koje će te podatke obraditi. Glavna karakteristika jezgri je da se operacije izvode nad svim elementima toka, a ne nad individualnim elementima. Uobičajeni način na koji se koriste jezgre je evaluacija neke funkcije nad svim elementima ulaznog toka (npr. transformacijska jezgra može prebaciti sve elemente toka u drugi koordinatni sustav).

Izlazni tokovi iz jezgri su samo funkcije ulaznih tokova, i unutar jezgri izračuni nad jednim elementom toka nikada ne ovise o nekom drugom elementu. Ova ograničenja imaju dvije glavne prednosti. Prvo, podatci potrebni za izvođenje jezgrene funkcije su potpuno poznati kada je jezgra napisana (eng. *compiled* - prevedena). Drugo, potrebna neovisnost

teksta programa izračuna nad različitim elementima toka unutar jedne jezgre omogućuje raspoređivanje izračunavanja koje izgleda kao serijsko na podatkovno paralelnom sklopovlju. Aplikacije se konstruiraju tako da se više jezgri ulančava jedna iza druge. Npr. implementacija grafičkog cjevovoda u modelu obrade toka podataka uključuje jezgru za obradu vrhova, jezgra za sastavljanje trokuta, odrezujuću jezgru (eng. *clipping kernel*) itd., te na kraju povezivanje izlaza jedne jezgre u ulaz sljedeće jezgre kao što je prikazano na slici 3. Ovaj model čini komunikaciju između jezgri eksplicitnom, iskorištavajući tako svojstvenu lokalnost podataka između jezgri u grafičkom cjevovodu.



Slika 3 (preuzeto sa [14])- Preslikavanje grafičkog cjevovoda u model obrade toka podataka.

Model obrade toka podataka omogućuje učinkovito izračunavanje na više načina, a što je najvažnije, razotkriva paralelizam u aplikacijama. Budući da jezgre izvode operacije nad cijelim tokovima, elementi toka mogu se obraditi u paraleli pomoću podatkovno paralelnog sklopovlja. Dugi tokovi sa puno elemenata omogućuju da taj podatkovni paralelizam bude vrlo učinkovit. Unutar obrade jednog elementa možemo iskoristiti paralelizam na razini instrukcije, a zato jer se aplikacije grade iz više jezgri, te jezgre mogu biti dublje povezane u cjevovodu i izvedene u paraleli koristeći paralelnost na razini zadataka.

Efikasna komunikacija je također jedan od glavnih ciljeva teksta programa modela obrade toka podataka. *Off-chip* komunikacija (kada komuniciraju različiti tipovi sklopovlja, globalna komunikacija) je učinkovitija ako se cijeli tokovi prenose u ili iz memorije, za razliku od prijenosa individualnih elemenata, jer se fiksni vremenski trošak komunikacije amortizira preko cijelog toka umjesto samo jednog elementa. Dakle, budući da se radi sa

tokovima podataka, u najgorem slučaju iz memorije čitamo tok podataka koji se sastoji od određenog broja elemenata, a ne podatak po podatak za što nam treba više vremena.

Zatim, strukturiranje aplikacija kao skupa ulančanih jezgri omogućuje da se podatci između susjednih jezgri zadrže u lokalnoj memoriji sklopa, a ne da se prenose u memoriju i iz nje. Efikasne jezgre pokušavaju zadržati svoje ulazne podatke i one izračunate lokano unutar jezgrine izvršne jedinice, zato se reference podataka ne šalju sa sklopa (*off-chip*) u memoriju ili preko sklopa (*accross a chip*) u priručnu memoriju što je tipično za centralni procesor. I na kraju, duboko ulančavanje izvršavanja omogućuje sklopovskim implementacijama da i dalje rade koristan posao dok čekaju da se podatci dohvate iz globalne memorije. Ovaj visoki stupanj tolerancije memorijske latencije omogućuje da se sklopovske implementacije optimiziraju za visoku memorijsku propusnost umjesto potrebe za smanjenjem latencije.

3.4. PROGRAMSKI KONCEPTI

Postoji više različitih računalnih sredstava dostupnih grafičkom procesoru [11][13][14]. Procesori vrhova izvode programe za obradu vrhova (*vertex shadere*), koji su zapravo funkcije koje se koriste kako bi se dodali posebni efekti objektima u 3D prostoru tako da izvode matematičke operacije nad podatcima koji su karakteristični za vrhove objekta. Svaki se vrh može definirati pomoću puno različitih varijabli. Npr. vrh je u prostoru uvijek definiran sa x, y i z koordinatom, ali vrhovi mogu također biti definirani i po boji, teksturama i karakteristikama osvjetljenja. Pri tome se tipovi podataka ne mijenjaju, jedino se mijenjaju njihove vrijednosti tako da vrhovi samo mijenjaju boju, teksturu ili položaj u prostoru [15][16].

Procesor piksela izvodi program za obradu fragmenata (eng. *fragment shader*, ovaj naziv koristi OpenGL) ili piksela (eng. *pixel shader*). Program za obradu piksela izračunava boju i ostale atribute svakog piksela, a mogu biti takvi da uvijek daju istu boju piksela, obrađuju sjene, postavljaju spekularno osvjetljenje, omogućuju zamagljenost (eng. *blur*) i mnoge druge efekte. On sam po sebi ne može dati kompleksne efekte ako ne zna geometriju

scene ili svojstva susjednih piksela, zato jer radi sa pojedinačnim pikselima, ali mogu promijeniti dubinu piksela ili kao izlaz dati više od jedne boje ako postoji više objekata koji se iscrtavaju na zaslonu.

Tijekom procesa iscrtavanja slike na zaslonu (eng. *rendering*), u koraku rasterizacije uzimamo geometrijske primitive (točke, linije, trokuti...) opisane koordinatama vrhova i pripadajućim informacijama o boji i teksturama, te ih konvertiramo u skup fragmenata koji zatim prolaze kroz skup operacija. Prema tome, fragmenti su skupovi podataka koji su potrebni kako bi iscrtali piksele u spremnik okvira (eng. *framebuffer*), a pomoću programa za obradu vrhova obrađujemo te podatke.

Dakle, kod pisanja programa opće namjene za grafički procesor, programe za obradu vrhova i piksela možemo iskoristiti za obavljanje proračuna, no sa pojavom programa za izračune opće namjene (*compute shader*) ta mogućnost će se vjerojatno izbjegavati. Ono što je zajedničko programima za obradu vrhova i piksela je primanje ulaznih tokova podataka, koji kod programa pisanih za grafički procesor najčešće dolaze u obliku 2D polja. Mnogi izračuni prirodno se mogu opisati pomoću 2D polja podataka, kao npr. množenje matrica, obrada slika, fizikalno temeljene simulacije i mnogi drugi. Najčešći način na koji se izvode operacije nad tim poljem je taj da se postavi međuspremnik sa dimenzijama izlazne jedinice (zaslona) u pikselima, te se definira program za obradu piksela. Tada se pokrene program za obradu piksela koji obrađuje podatke u tom međuspremniku, a dobiveni pikseli se pohranjuju u stražnji međuspremnik. Ako algoritam ima više koraka, tada se mogu koristiti operacije *render-to-texture* (RTT) ili *copy-to-texture* (CTT) , pri čemu se podatci spremaju u teksture koje se zatim mogu pročitati kao ulaz za sljedeći korak [11][14]. Budući da se u tom slučaju teksture koriste kao memorija, koordinate tekstura se koriste kao memorijske adrese. Zbog toga neke operacije GPU može obaviti automatski. Npr. generira se tekstura veličine zaslona na temelju podataka, definiraju se njene koordinate i pohrani se u međuspremnik. Tada GPU računa odgovarajuće adrese za svaki generirani fragment podataka i ti podatci su tada dostupni programeru bez potrebe za dodatnim prijenosom podataka.

Programi za obradu vrhova i piksela su ono što se u modelu obrade toka podataka naziva

jezgrama, dakle funkcije koje obavljaju određene operacije nad tokovima podataka. U principu o njima možemo razmišljati kao o petljama, ali samo sa tijelom petlje. Dakle ako bi programer radio sa dvodimenzionalnim poljem na centralnom procesoru, tekst programa koji bi procesirao podatke u tom polju izgledao bi otprilike kako je prikazano na slici 4.

```
/* PSEUDOTEKSTA PROGRAMA */
napravi polje A[ 10 000 ][ 10 000 ]

za i od 1 do 10 000 {
     za i od 1 do 10 000 {
          procesiraj( A[i][j] );
     }
}
```

Slika 4 - Pseudoteksta programa funkcije za obradu elemenata polja pisana za centralni procesor.

Ako bi taj isti tekst programa htjeli napisati za grafički procesor, jedino bi nam trebalo tijelo petlje, tj. trebalo bi nam: `procesiraj( A[i][j])`.

3.5. GPU TEHNIKE

GPU tehnike su zapravo posebne operacije koje određuju na koji će se način, prethodno definirana funkcija primjeniti na tok podataka. Ovdje je dan pregled tih operacija i opisan način na koji pojedina od njih primjenjuje funkcije na tokove podataka [11].

3.5.1. MAP

Map (eng. map – preslikavanje) operacija jednostavno primjenjuje danu funkciju (jezgru) na sve elemente u toku podataka. Jednostavan primjer je množenje svakog elementa toka

sa konstantom. Ovu operaciju je jednostavno impelemntirati na GPU. Generira se fragment za svaki piksel na ekranu i primjenuje se program za obradu vrhova za svaki od tih fragmenata, a rezultirajući tok iste veličine sprema se u izlazni međuspremnik.

3.5.2. REDUCE

Neki izračuni zahtjevaju da se iz većeg toka izračuna manji tok (moguće i samo jedan element). Ovo se naziva redukcija ili smanjenje toka (eng *reduce* – smanjiti, umanjiti). Generalno, redukcija se može postići u više koraka, rezultati iz prijašnjih koraka koriste se u sljedećim koracima, raspon elemenata nad kojima se izvodi operacija se smanjuje i tako sve dok ne ostane traženi broj elemenata.

3.5.3. STREAM FILTERING

Filtriranje toka je u principu nejednolika redukcija. Filtriranje uključuje brisanje određenih elemenata iz toka na temelju određenih kriterija.

3.5.4. SCATTER

Scatter (eng. *scatter* – raspršiti, raspršenje) je operacija koja je prirodno definirana za procesor vrhova. Procesor vrhova sposoban je prilagoditi položaj vrha, što omogućuje programeru da kontrolira gdje su informacije pohranjene na 2D mreži, koja je najprirodniji način za prikaz toka podataka. Druga proširenja funkcionalnosti su također moguća, kao što je kontrola veličine područja na koje vrh utječe. Procesor fragmenata ne može obavljati operaciju raspršenja jer je lokacija svakog fragmenta na mreži eksplicitno određena u trenutku stvaranja fragmenta i programer je ne može promijeniti.

3.5.5. GATHER

Gather (eng. *gather* – prikupljati) operacija je jedino moguća u programu za obradu piksela, jer on može čitati teksutre tako da proizvoljno pristupa ćelijama u 2D mreži (*random access*), i na taj način može prikupiti informacije iz bilo koje ćelije ili više njih, ovisno o potrebama. Program za obradu vrhova ne može obavljati operaciju prikupljanja jer ne može pristupiti podatcima o drugim vrhovima u toku, gledajući iz perspektive proizvoljnog vrha. Novije grafičke kartice omogućuju čitanje podataka o teksturama iz programa za obradu vrhova, ali to nije prava operacija prikupljanja podataka budući da se ne odvija nad samim tokom vrhova.

3.5.6. SORT

Sort (eng. *sort* – sortirati, razvrstati) operacija transformira neporedani skup elemenata u poredani skup. Najčešća implementacija sort operacije na GPU je pomoću sortirajućih mreža.

Sortirajuće mreže su apstraktni matematički model mrežnih linija i uspoređivačkih modula koji se koriste kako bi se sortirao skup brojeva. Svaki uspoređivač povezuje dvije žice i sortira ih tako da na jednu žicu stavi manju vrijednost, a na drugu veću.

3.5.7. SEARCH

Search (eng. *search* – tražiti) operacija omogućuje programeru da pronađe određeni element u toku podataka ili da pronađe susjede specificiranog elementa. GPU se ne koristi da se ubrza traženje za individualnim elementom, već se koristi da se pokrene više traženja paralelno.

4. SUČELJE DIRECTX

4.1. RAZVOJ DIRECTX API-a

Microsoftov DirectX je skup aplikacijskih programskih sučelja (API – *Application Programming Interfaces*) za obavljanje zadataka povezanih sa multimedijom, a posebice za programiranje računalnih igara i obradu videa na Microsoftovim platformama [17]. Orginalno, početni dio imena svake komponente u programskom sučelju počinjao je sa riječi *Direct*, poput Direct3D, DirectDraw, DirectInput, DirectSound, itd. Ime DirectX je nastalo kao skraćenica za sve uključene komponente gdje *X* zamjenjuje uobičajeno ime komponente, te je uskoro postalo glavni naziv za taj skup programskih sučelja. Kada je Microsoft kasnije radio na razvoju svoje konzole za računalnu zabavu, iskoristio je *X* kao bazu za njeno ime (*Xbox*) i na taj način označio da se bazira na DirectX tehnologiji.

Direct3D je glavno sučelje DirectX-a za izradu trodimenzionalnih grafičkih aplikacija i široko se koristi u području računalne zabave za izradu računalnih igara. Direct3D također koriste i druge aplikacije za vizualizaciju i grafičke zadatke poput CAD/CAM sustava. Budući da je Direct3D najpoznatija komponenta DirectX-a, ta dva naziva često se koriste kao sinonimi.

Povijest DirectX-a počinje kasne 1994-te godine, nekoliko mjeseci prije izlaska Windows 95 operacijskog sustava. Koje aplikacije će taj novi sustav moći pokrenuti bio je glavni faktor pri određivanju iznosa koji bi kupci bili spremni platiti za njega. Tri Microsoftova zaposlenika, Craig Eisler, Alex St. John i Eric Engstrom, uočili su da programeri vide prethodni Microsoftov operacijski sustav, MS-DOS, kao bolji za izradu igara što je za sobom povlačilo činjenicu da će manje igara biti rađeno za Windows 95 i da neće postići planirani uspjeh. DOS je dopuštao direktan pristup video kartici, ulaznim jedinicama, uređajima za reprodukciju zvuka i svim ostalim dijelovima sustava, dok Windows 95 sa svojim modelom zaštićene memorije nije dopuštao pristup većini od navedenih uređaja. Microsoft je trebao programerima pružiti ono što su oni zahtjevali, a kako su samo mjeseci ostali do izlaska novog operacijskog sustava trebalo je brzo doći do rješenja.

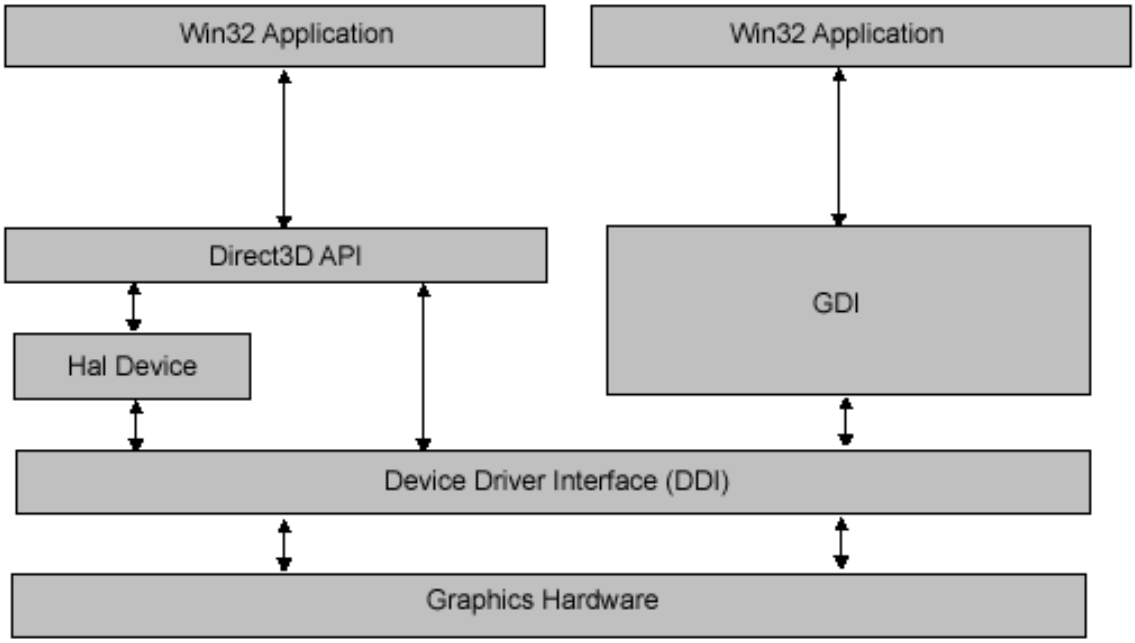
Eisler, St. John i Engstrom su radili zajedno na rješavanju tog problema, a konačno rješenje nazvali su DirectX. Prva verzija DirectX 1.0 izašla je u rujnu 1995. godine, mjesec dana nakon izlaska Windows 95 operacijskog sustava i jedina je verzija koja se nije isporučivala sa operacijskim sustavima iz Windows serije.

Sve verzije DirectX-a od verzije 2.0 nadalje isporučivale su se sa Windows operacijskim sustavom i sve novije verzije bile su kompatibilne sa starijim Windows sustavima. No to se promijenilo pojavom DirectX 10 koji je bio isključivo dostupan samo na Windows Vista OS. Razlog je bio u tome što je novi DirectX ovisio o novom pogonskom programu (eng. *driver*) koji je došao sa novim Windows Vista operacijskim sustavom, sa ciljem poboljšanja grafičkih performansi i bolje funkcionalnosti. Novi WDDM (*Windows Display Driver Model*) pruža funkcionalnost koja je potrebna za grafički prikaz aplikacija i pozadine zaslona, a napravljen je povrh Direct3D komponente DirectX-a što omogućuje različite trodimenzionalne efekte u grafičkom sučelju. WDDM ujedno podržava i nova sučelja u DirectX API-ju koja su nužna za programere prilikom izrade grafičkih aplikacija [18]. DirectX 10 je sa sobom donio mnoge promjene, stoga kako bi se održala kompatibilnost sa starijim sustavima postoji nekoliko različitih Direct3D komponenti u samom DirectX sučelju, a ovisno o sustavu na kojem se DirectX 10 nalazi koristi se odgovarajuća verzija Direct3D komponente.

DirectX 11 verzija je koja dolazi sa Windows 7 operacijskim sustavom i donosi nove mogućnosti poput podrške za teselaciju (dinamičko povećanje broja vidljivih poligona), poboljšanje potpore za višedretvenost koja omogućuje programerima bolje iskorištavanje dostupnih procesnih elemenata na višejezgrenim procesorima i ono što je vjerojatno najbitnije, podršku za pisanje programa opće namjene za grafički procesor (GPGPU) u obliku *DirectCompute* komponente.

4.2. ARHITEKTURA DIRECTX-a

Kada se govori o arhitekturi DirectX-a, misli se na položaj njegove najvažnije komponente Direct3D, u odnosu na sustav u kojem se nalazi. Glavna zadaća Direct3D komponente je učiniti apstraktnom svu komunikaciju između DirectX aplikacije i pogonskih programa grafičkog sklopovlja. Direct3D komponenta predstavljena je kao apstraktni sloj (eng. *layer*) u istoj razini kao i Windows GDI (*Graphics Device Interface*) što je vidljivo sa slike 5.



Slika 5 (preuzeto sa msdn.microsoft.com) - Položaj Direct3D komponente DirectX API-ja u odnosu na

Windows GDI je također programsko sučelje odgovorno za prikaz grafičkih objekata i njihov prijenos do izlaznih uređaja. Iako GDI nije direktno odgovoran za iscrtavanje prozora i menija na zaslonu, koristi se za crtanje linija, krivulja, iscrtavanje tekstualnih fontova i kontrolu paleta boja. Razlika izmeđi GDI i DirectX-a je u direktnoj povezanosti DirectX-a i upravljačkih programa grafičkog sklopovlja, te nemogućnosti rasterizacije 3D scena pomoću GDI API-ja.

Direct3D je grafički API koji radi u neposrednom načinu rada (*immediate mode*). On pruža sučelje niske razine za svaku funkciju koju 3D grafčko sklopovlje podržava (transformacije, odrezivanja, materijali, teksture, itd.). Neposredni način rada pruža nam tri glavne

apstrakcije: uređaji (*devices*), sredstva (*resources*) i izmjenivi lanci podataka (*swap chain*). Uređaj možemo podijeliti u četiri različita tipa:

a) HAL (*Hardware Abstraction Layer*) je uređaj koji u potpunosti podržava sklopovlje na kojem se 3D aplikacija izvodi i brzina izvršavanja teksta programaa jednaka je brzini sklopovlja.

b) Referentni uređaj omogućava simuliranje novih grafičkih mogućnosti koje sklopovlje još ne podržava. Npr. razvoj aplikacije sa novom verzijom API-ja, prije nego je sklopovlje koje podržava tu novu verziju postalo dostupno programerima. Budući da je simuliranje programski izvedeno, brzina izvođenja aplikacije osjetno je manja nego je to u slučaju HAL uređaja.

c) NULL referentni uređaj vraća samo crni ekran kada razvojni alati na nekom sustavu nisu dostupni (točnije DirectX SDK nije instaliran), a zatražen je referentni uređaj.

d) Programski ostvareni uređaj (*Pluggable Software Device*) koristi se teksta programa programske rasterizacije.

Svaki tip uređaja sadrži barem jedan lanac za izmjenu podataka. Lanac za izmjenu podataka se sastoji od jednog ili više stražnjih međuspremnika (eng. *back buffer*). Stražnji međuspremnik sadrži podatke o pojedinim pikselima, poput boje, dubine prozirnosti ili teksturi. On ujedno služi i kao spremnik u koji će se iscrtati slika prije prikaza na zaslon. Najćešća tehnika korištenja međuspremnika za iscrtavanje slike na zaslon uključuje dva takva međuspremnika. Dok se u jednome iscrtava slika za prikaz (stražnji međuspremnik, eng. *back buffer*), slika iz drugog međuspremnika prikazuje se na zaslonu (prednji međuspremnik, eng. *front buffer*). Kada je slika u stražnjem međuspremniku spremna za prikaz međuspremnici mijenaju mjesta, tako stražnji postaje prednji, a prednji postaje stražnji međuspremnik i u njega se tada iscrtava nova slika. Ovaj način rada može se primjeniti na više od dva međuspremnika za iscrtavanje, dakle dobivamo jednu lančano povezanu strukturu izmjenjujućih međuspremnika sa podatcima.

Treća glavna apstrakcija je skup sredstava koje uređaj sadrži. Sredstva su specifični

podatci koji se koriste prilikom procesa iscrtavanja slike na zaslonu, a svako sredstvo sastoji se od četiri atributa:

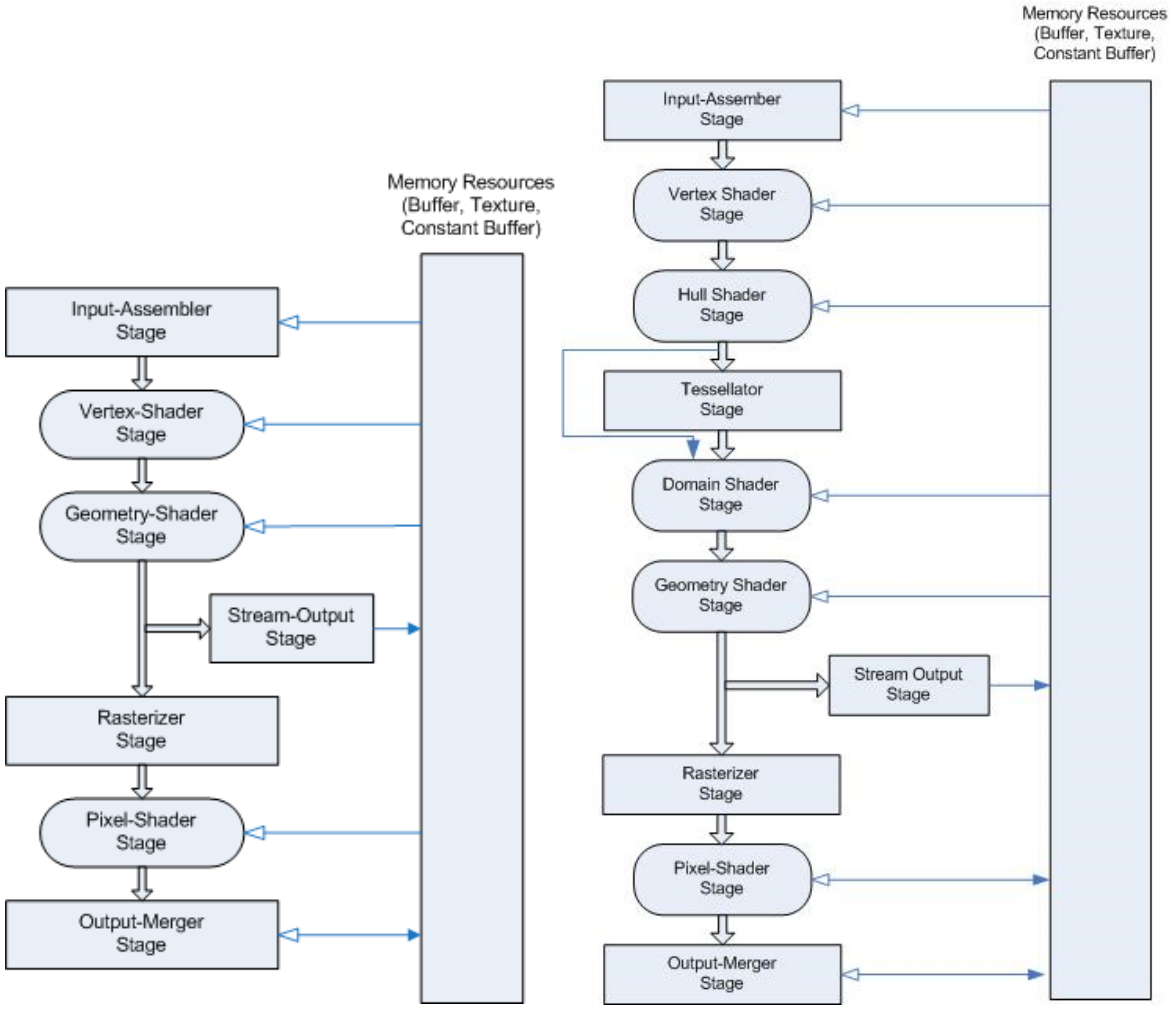
a) Tip (eng. *type*), opisuje o kakvom se sredstvu radi (spremnik vrhova, spremnik indeksa vrhova, površina, volumen, tekstura, kubna tekstura ili volumna tekstura).

b) Memorijsko korištenje sredstva (eng. *memory pool*), opisuje kako se sredstvom upravlja prilikom pokretanja aplikacije i gdje se sprema. Standardno memorijsko korištenje (eng. *default pool*) označava da se sredstvo sprema samo u memoriju grafičkog sklopovlja. Upravljano memorijsko korištenje (eng. *managed pool*) označava da će se sredstvo pohraniti u glavnu memoriju sustava i poslati na grafičko sklopovlje samo kada bude potrebno. Korištenje u memoriji sustava (eng. *system memory pool*) označava da će se sredstvo nalaziti samo u glavnoj memoriji sustava. Radno korištenje (eng. *scratch pool*) također označava da se sredstvo pohranjuje u glavnu memoriju sustava, ali u ovom slučaju sredstvo nije vezano ograničenjem sklopovlja.

c) Format (eng. *format*), opisuje kako će se sredstvo pohraniti u memoriji, uglavnom su to podatci o pikselima, npr. format *D3DFMT_R8G8B8* označava da se radi o pikselima koji sadrže 24-bitnu boju, a svaka komponenta boje ima 8-bita (R-crvena, G-zelena, B-plava).

a) Korištenje (eng. *usage*), opisuje pomoću skupa zastavica način na koji će aplikacija koristiti sredstvo. Zastavice uglavnom služe da se sredstva označe kao statična ili dinamička. Statička sredstva neće mijenjati svoju vrijednost nakon pokretanja aplikacije, dok dinamička sredstva kontinuirano mijenjaju svoju vrijednost.

DirectX API također definira način na koji se skup podataka, vrhova, tekstura i spremnika pretvara u sliku na zaslonu. Ovaj proces je opisan kao grafički cjevovod (eng. *rendering pipeline*) sa nekoliko različitih stanja. Izlgedi cjevovoda kakav podržavaju DirectX 10 i DirectX 11 prikazani su na slici 6. Grafički cjevovod za DirectX 11, za razliku od DirectX 10, ima još tri dodatna stanja između *vertex shader* i *geometry shader* stanja, a to su *hull shader, tessellator* i *domain shader.*



Slika 6 (preuzeto sa msdn.microsoft.com) - Grafički cjevovod kako ga definira DirectX10 lijevo i grafički cjevovod kako ga definira DirectX11 desno.

4.3. SUČELJE PREMA GRAFIČKOM SKLOPOVLJU

Da bi neke podatke uopće mogli poslati na grafičko sklopovlje i pokrenuti proces iscrtavanja slike na zaslonu, potrebno je inicijalizirati sučelje prema grafičkom sklopovlju. U DirectX 11 API-ju, funkcionalnost za stvaranje sredstava (međuspremnici, teksture, UAV i SRV pogledi, itd.) i za iscrtavanje slike na zaslonu podijeljena je na komponentu uređaja (eng. *device*) i jednu ili više komponenti konteksta uređaja (eng. *device context*). Komponenta uređaja nam služi za stvaranje sredstava, dok nam komponenta konteksta uređaja služi za postavljanje različitih stanja grafičkog cjevovoda (npr. postavlja programe za obradu vrhova i piksela), te za generiranje naredbi za iscrtavanje slike na zaslonu pri čemu koristi sredstva koja je inicijalizirala komponenta uređaja.

Npr. pomoću komponente uređaja inicijalizirali smo međuspremnik koji sadži popis svih vrhova nekog objekta u prostoru i koje želimo iscrtati na ekranu, tada pomoću komponente konteksta uređaja postavimo program za obradu vrhova koji će napraviti transformaciju tih vrhova u prostor promatrača (točka iz koje promatramo objekt u sceni) i postavimo te vrhove u početno stanje grafičkog cjevovoda.

U jednom trenutku možemo imati i više od jedne komponente konteksta uređaja inicijalizirane, te na taj način možemo bolje iskoristiti sustave koji sadrže više procesnih elemenata i pružaju mogućnost višedretvenosti. Kada je u sustavu inicijalizirana samo jedna komponenta konteksta uređaja, tada ona radi u neposrednom (eng. *immediate*) načinu rada što znači da podatke šalje direktno upravljačkim programima grafičkog sklopovlja, tj. u ovom načinu rada komponenta konteksta uređaja može neposredno pokrenuti proces iscrtavanja slike na zaslonu. U jednom trenutku može postojati samo jedna komponenta konteksta uređaja koji radi u neposrednom načinu rada i koja može dohvaćati podatke sa grafičkog sklopovlja.
Kada sustav ima više procesnih elemenata, ili podržava višedretvenost, možemo inicijalizirati više komponenti konteksta uređaja koje onda rade u odgođenom (eng. *deferred*) načinu rada. U ovom načinu rada postoji jedna glavna dretva i njena pripadajuća komponenta konteksta uređaja koja radi u neposrednom načinu rada i koja može pokrenuti proces iscrtavanja slike na zaslonu, dok sve ostale komponente konteksta

uređaja rade u odgođenom načinu rada, te svoje naredbe pohranjuju u listu naredbi, koje onda kasnije komponenta konteksta uređaja u glavnoj dretvi može prosljediti grafičkom sklopovlju.

4.4. HLSL

High level shading language ili *high level shader language* je jezik za programiranje shader programa koji je razvio Microsoft za uporabu sa svojim Direct3D API-jem, zadržavajući sintaksu sličnu sintaksi jezika C, čime se olakšava učenje jezika iskusnim programerima. Analogan je GLSL jeziku koji se koristi u OpenGL aplikacijama i isti je kao Nvidia-in Cg jezik budući da su razvijani usporedno [21].

Početci HLSL-a počinju sa DirectX 8 kao prvi korak prema programibilnom grafičkom cjevovodu. DirectX 8 grafički cjevovod programirao se kombinacijom asemblerskih instrukcija, HLSL instrukcija i fiksnih funkcija. Dolaskom DirectX 10 API-ja, grafički cjevovod se u potpunosti programira samo pomoću HLSL jezika [22].

4.4.1. HLSL VARIJABLE

HLSL varijable vrlo su slične varijablama definiranim u C programskom jeziku. Slično C-u, varijable imaju ograničenja teksta programa imenovanja, svojstvo dosega koje ovisi o tome gdje su u programu definirane i mogu sadržavati dodatne korisnički definirane metapodatke. Poput C jezika, postoji nekoliko standardnih tipova podataka, ali za razliku od C jezika postoje i dodatni tipovi podataka koje HLSL definira kako bi se maksimizirale performanse operacija nad 4-komponentim vektorima koji koriste matrične matematičke operacije nad 3D podatcima [23].

Skalar je tip podatka koji je jednak skalarnim tipovima podataka iz C jezika. U HLSL jeziku skalar može biti *bool*, *int* (32-bitni cjelobrojni tip sa predznakom), *uint* (32-bitni cjelobrojni tip bez predznaka), *half* (16-bitna vrijednost sa pomičnim zarezom), *float* (32-bitna vrijednost sa pomičnim zarezom) i *double* (64-bitna vrijednost dvostruke preciznosti).

Vektori u HLSL jeziku su varijable koje mogu sadržavati više komponenti, tj. skalara. Ukoliko želimo varijablu koja sadrži samo jednu komponentu napisati ćemo samo tip varijable i njeno ime kao u klasičnom C jeziku, ali ukoliko želimo varijablu sa većim brojem komponenti tada nakon tipa varijable moramo navesti broj komponenti koji želimo (maksimalno četiri), npr. dva načina definiranja vektora sa četiri komponente i inicijalizacija:način 1: `int4 ime_varijable = { 1, 2, 3, 4 };`
način 2: `vector<int, 4> ime_varijable = { 1, 2, 3, 4 };`

Postoje dva skupa za imenovanje pojedinih komponenti, preko kojih možemo pojedinim komponentama pristupati, pozicijski skup (x, y, z, w) i skup prema boji (r, g, b, a). Ukoliko napišemo `ime_varijable.x` ili `ime_varijable.r` prema gornjem primjeru, u oba slučaja dobivamo vrijednost 1. Pomoću ovih skupova za imenovanje komponenti možemo pristupati istovremeno i više od jedne komponente, ali skupovi se ne smiju miješati.

Matrični tip podatka je podatkovna struktura koja sadrži redove i stupce podataka. Podatci mogu biti bilo kojeg skalarnog tipa, ali svaki element matrice mora biti istog tipa kao i ostali elementi. Broj redova i stupaca matrice specificira se kao string koji se doda na osnovni skalarni tip podatka i sadrži broj redova puta broj stupaca, npr. `intNxM matrica` ili `matrix<float,N,M> matrica` gdje su N i M proizvoljni cijeli brojevi između jedan i četiri. Elementima matrice također možemo pristupati preko dva odvojena skupa za imenovanje komponenti, a razlikuju se po tome kako želimo označiti prvi element matrice, je li prvi element matrice indeksiran sa 0,0 ili 1,1. Pristupanje svim elementima dijagonale matrice veličine 4×4 u prvom slučaju napisali bi na sljedeći način: `matrica._m00_m11_m22_m33`, dok bi u drugom slučaju to ovako napisali: `matrica._11_22_33_44`.

4.4.2. HLSL UVJETNA GRANANJA I PETLJE

Većina grafičkog sklopovlja dizajnirana je tako da izvršava teksta programa u shader programu naredbu po naredbu. Kontrola toka određuje koji će se blok HLSL naredbi

izvesti sljedeći. Koristeći kontrolu toka shader može koristiti petlje ili grananja za izvođenje instrukcija koje nisu sljedeće po redu. HLSL podržava `break`, `continue`, `do`, `for`, `while`, `switch`, `if` i `discard`. Prvih sedam naredbi dobro su poznate iz C jezika, jedina 'nova' naredba je `discard` koja u programima za obradu piksela označava da se rezultat aktualnog piksela neće prikazati na zaslon.

Za razliku od C jezika, naredbe za kontrolu toka imaju dodatne atribute kao signale prevodiocu kojima se označava na koji način da se naredbe izvedu. Ti atributi se navode u uglate zagrade ispred naredbe za kontrolu toka. Za naredbe `do`, `while` i `for` možemo tako navesti dva atributa `[loop]` i `[unroll(x)]`. Ako navedemo atribut `loop` prevodioc generira teksta programa koji koristi kontrolu toka za izvođenje svake iteracije petlje, dok `unroll` atribut 'odmotava' petlju do kraja, te se opcionalno može navesti željeni broj iteracija petlje. Važno je za navesti da se sve petlje u shaderu moraju izvesti u manje od 1024 iteracije, inače se prilikom prevođenja generira pogreška [24].Za `if` naredbu postoje druga dva atributa, `[branch]` i `[flatten]`. Ako se navede `branch` atribut, tada se evaluira samo jedna strana `if` naredbe ovisno o danom uvjetu. Sa `flatten` atributom evaluiraju se obje strane `if` naredbe i odabere se na temelju dva rezultata koja grana će se izvršiti [25].`Switch` naredba koristi četiri atributa, dva prethodno navedena teksta programa `if` naredbe i još dva nova atributa, `[forcecase]` i `[call]`. `Forcecase` forsira sklopovsku izvedbu `switch` naredbe, dok se sa `call` atributom svi individualni slučajevi premještaju u podrutine (eng. *subroutine*), pa `switch` naredbu čine pozivi podrutina.

4.4.3. HLSL FUNKCIJE I SEMANTIKA

HLSL funkcije, poput funkcija u drugim programskim jezicima, enkapsuliraju niz naredbi. Kod programa pisanih za grafički procesor praktična uporaba funkcija je napraviti funkcije za različite efekte, otkloniti moguće greške i zatim ih koristiti u različitim programima. HLSL ima i puno ugrađenih intrinzičnih funkcija. Sve intrinzične funkcije su ispitane i optimizirane pa je dobra praksa koristiti te gotove funkcije umjesto da se pišu nove.

Sama sintaksa vrlo je slična funkcijama u C jeziku, definiramo tip povratne vrijednosti, ime funkcije, listu argumenta, ali dodatno možemo specificirati semantiku funkcije, tj. opcionalni string koji navodi na koji način će se iskoristiti povratni podatci. Npr. ukoliko definiramo fukciju sa semantikom `SV_Target[n]`, tada kažemo funkciji da će se povratni podatci pohraniti u jedan od *n* stražnjih međuspremnika ($0 \leq n \leq 7$).
Semantika nije isključivo vezana za funkcije, već i za varijable. Uglavnom semantika se dodjeljuje ulaznim i izlaznim varijablama programa za grafički procesor i potrebna je za sve varijable koje se prenose između različitih stanja grafičkog cjevovoda. Općenito, svi podatci koji se prenose između različitih stanja grafičkog cjevovoda su potpuno generički i nemaju jedinstvenu interpretaciju od strane sustava zbog čega je potrebno definirati što predstavljaju, a dozvoljeno je koristiti i proizvoljnu semantiku koja nema nikakvo specijalno značenje [26].

Od inačice DirectX 10 uvedene su i specijalne semantike koje se nazivaju *System-Value* semantike. Sve takve specijalne semantike počinju sa SV_ prefiksom, uobičajena je SV_POSITION semantika koja se interpretira u stanju rasterizacije, no SV semantike su valjane i u drugim stanjima grafičkog cjevovoda. Npr. SV_POSITION se može specificirati i kao semantika uzlaznih podataka i kao semantika izlaznih podataka programa za obradu vrhova. Programi za obradu piksela na primjer mogu pisati samo u vrijednosti označene sa SV_DEPTH i SV_TARGET semantikama. Pogledajmo konkretniji primjer na slici 7.

```
struct VS_OUTPUT
{
    float4 Pos       : SV_POSITION;
    float4 Color     : COLOR0;
};


float4 BasicPS( VS_OUTPUT input ) : SV_Target
{

   return input.Color;
}
```

Slika 7 - Jednostavni pixel shader.

Funkcija na slici predstavlja jednostavni program za obradu piksela. Ovaj program prima izlaz programa za obradu vrhova u obliku strukture koja sadrži dvije varijable. Prva varijabla u strukturi je 4-komponenti vektor `Pos` označen sa SV_POSITION semantikom, koja označava da se radi kordinatama vrha u prostoru. Druga varijabla je također 4-komponentni vektor označen semantikom COLOR0 i sadrži informaciju o boji u tom vrhu. Funkcija `BasicPS` prima te dvije informacije i jednostavno vraća primljenu boju. Vidimo da je `BasicPS` funkcija označena semantikom SV_Target koja indicira da će se ta boja spremiti u međuspremnik iz kojeg se iscrtava konačna slika i da je povratna vrijednost također 4-komponentni vektor.

4.4.4. OGRANIČENJA HLSL JEZIKA

Iako HLSL predstavlja moćnu platformu za programiranje različitih stanja grafičkog cjevovoda, također sadrži neka ograničenja. HLSL pruža iste konstrukte za kontrolu toka poput viših programskih jezika, no kontrola toka na grafičkom procesoru općenito se razlikuje od one na centralnom procesoru. Centralni procesori generalno imaju duge instrukcijske cjevovode, pa je prilično važno dobro predviđanje koja će se grana u kontroli toka izvršiti. Ako je predviđanje točno kontrola toka praktički neće imati posljedica, no ako predviđanje nije točno centralni procesor može pasti u prazan hod jer će se instrukcijski cjevovod isprazniti i on mora čekati da se napuni instrukcijama sa prave adrese u memoriji.

Karakteristike kontrole toka i grananja na grafičkim procesorima su drugačije. Dva najčešća kontrolna mehanizma u paralelnim arhitekturama su jedna instrukcija na više podataka (SIMD) i više instrukcija na više podataka (MIMD). Svi procesni elementi u SIMD paralelnim arhitekturama izvršavaju istu instrukciju u isto vrijeme, dok u MIMD arhitekturama različiti procesni elementi izvršavaju različite instrukcije. Trenutno se koriste tri metode za predviđanje grananja na grafičkim procesorima: MIMD grananje, SIMD grananje i uvjetni teksta programaovi (eng. *condition codes*) [27].

MIMD grananje je idealan slučaj u kojem različiti grafički procesni elementi mogu različito evaluirati grananja i izvršiti teksta programa bez posljedica, što je vrlo slično centralnim

procesorima.

SIMD grananje omogućava grananja i petlje unutar programa, ali svi procesni elementi moraju izvršiti jednaku instrukciju, zbog toga različiti ishodi grananja mogu rezultirati smanjenjem performansi shadera. Npr. uzmimo program za obradu piksela, koji na temelju slučajnih ulaznih vrijednosti, vraća konačnu boju piksela. Taj će program za obradu piksela u svojim različitim instancama na grafičkom procesoru imati različite ishode grananja, što će uzrokovati da procesni elementi grafičkog sklopovlja koji ne izvršavaju trenutnu moguću granu, čekaju ostale procesne elemente da završe sa obrađivanjem trenutne grane. Ovo će rezultirati time da će mnogi procesni elementi na grafičkom sklopu koji izvode programe za obradu piksela provesti puno vremena u praznom hodu i izgubiti puno procesnog vremena. SIMD grananja su korisna kada su uvjeti grananja prostorno koherentni, nekoherentna grananja mogu biti vremenski vrlo skupa.

Uvjetni kodovi se koriste kod starijih arhitektura kako bi se emuliralo pravo (eng. *true*) grananje. If-then naredbe evaluiraju i onu granu za koju je zadovoljen uvjet i onu za koju uvjet nije zadovoljen, a nakon evaluacije postavljaju se uvjetni teksta programaovi. Instrukcije u svakoj grani moraju provjeriti vrijednosti uvjetnih kodova prije zapisa rezultata u registar. Na taj način samo grane koje su se izvršile zapisuju svoje rezultate u registre. Dakle, vremenski troškovi jednaki su vremenu potrebnom da se evaluiraju sve grane grananja i vremenu potrebnom za evaluaciju uvjeta grananja.

Iako ovo nije direktno nedostatak HLSL programskog jezika, već sklopovlja na kojem se shader programi izvode, kontrola toka može imati velike vremenske troškove i utjecaj na performanse shader programa pisanih u HLSL jeziku.

4.5. PROGRAMI ZA IZRAČUNE OPĆE NAMJENE (COMPUTE SHADER)

HLSL programi ne prevode se zajedno sa aplikacijom, već se prevode unutar aplikacije nakon njenog pokretanja. Za programe za obradu vrhova i piksela karakteristično je da se

nakon prevođenja povezuju sa određenim stanjima grafičkog cjevovoda (*vertex-shader stage* i *pixel-shader stage*), te obrađuju podatke koji prolaze kroz grafički cjevovod.

Programi za izračune opće namjene (*compute shader*) predstavljaju novo programibilno stanje koje proširuje mogućnosti Direct3D sučelja izvan dosega samo grafičkih proračuna, prvi puta se pojavljuje u DirectX 11 API-ju i dio je *Shader Model 5.0* specifikacije. Poput ostalih programibilnih stanja (programi za obradu vrhova i geometrije na primjer) programi za izračune opće namjene namijenjeni su implementaciji u HLSL jeziku, no tu svaka sličnost sa drugim programibilnim stanjima prestaje. Programi za izračune opće namjene pružaju mogućnosti za obavljanje proračuna opće namjene iskorištavanjem velikog broja paralelnih procesnih elemenata na grafičkom sklopovlju i kao stanje nisu striktno vezani za određeni stupanj u grafičkom cjevovodu, možemo ih smjestiti između bilo koja dva stanja u grafičkom cjevovodu (npr. možemo pokrenuti program za izračune opće namjene prije ili poslije programa za obradu vrhova, ili poslije programa za obradu piksela, tj. može doći na izvršavanje u bilo kojem stupnju grafičkog cjevovoda). Programi za izračune opće namjene pružaju opcije za dijeljenju memoriju i sinkronizaciju između različitih dretvi kako bi se omogućile različite metode za učinkovitije programiranje.

Ranije smo spomenuli da su dretve na grafičkom procesoru instance koje izvršavaju programe i da je svaka dretva nezavisna od drugih dretvi. Kod programa za izračune opće namjene višedretvenost je malo eksplicitnija. Kod programa za izračune opće namjene razlikujemo pojedinačne dretve i grupe dretvi. Unutar samog programa za izračune opće namjene definiramo koliko dretvi želimo unutar grupe, a prilikom pozivanja tog programa definiramo željeni broj grupa. Dretve unutar jedne grupe mogu sadržavati dijeljene varijable. Ukoliko želimo da neka varijabla bude dijeljena između dretvi u grupi definiramo je kao *groupshared* (iako su HLSL varijable ranije bile spomenute, ovo svojstvo nismo objašnjavali jer je dostupno samo za programe za izračune opće namjene). Glavni razlog smještanju dretvi unutar grupe je zbog sinkronizacije, jer je moguće da će dretve unutar grupe u određenim trenutcima zavisiti jedne od drugih na temelju dijeljenih podataka, iako to nije nužno. Kada bi promatrali organizaciju dretvi u programima za izračune opće namjene, onda bi grupe mogli promatrati kao dijelove prostora, dok bi pojedinačne dretve bile grupe točaka u tim dijelovima prostora, zbog čega možemo reći

da postoji X × Y × Z dretvi unutar svake grupe, a postoji U × V × W grupa dretvi. Što se tiče ograničenja, unutar jedne grupe može biti maksimalno 1024 dretve (X × Y × Z <= 1024), pri čemu Z dimenzija može maksimalno biti 64 (Z <= 64). Broj grupa dretvi može biti daleko veći, a ukupni broj grupa po svakoj komponenti mora biti manji od 65535 [28]. Dretve unutar iste grupe izvode se istovremeno, dok se dretve u različitim grupama mogu i ne moraju izvoditi istovremeno. Postoje četiri različite semantike koje se mogu pridjeliti ulaznim parametrima glavne funkcije programa za izračune opće namjene. Pomoću tih semantika prevodioc zna koje vrijednosti se predaju u program za izračune opće namjene. Najčešće korištene semantike su *SV_DispatchThreadID* koja označava da je u varijablu pohranjen globalni prostorni otklon (eng. *offset*) dretve (možemo ga smatrati indeksom dretve ukoliko bi kombinirali se sve grupe dretvi) i *SV_GroupThreadID* koja označava da varijabla sadrži prostorni otklon dretve unutar pripadajuće grupe dretvi. Varijabla koja sadrži globalni prostorni otklon dretve (varijabla sa semantikom *SV_DispatchThreadID*) vrlo nam je važna zbog pohrane konačnih rezultata programa za izračune opće namjene. Ostale dvije semantike su *SV_GroupID* koja definira prostorni indeks grupe dretvi i *SV_GroupIndex* koja predstavlja jednodimenzionalni indeks dretve unutar pripadajuće grupe dretvi (možemo smatrati kao redni broj dretve unutar grupe).

Program za izračune opće namjene zapravo nema izlaznih podataka. Rezultati proračuna spremaju se u memoriju na grafičkom sklopovlju i dohvaćaju u međuspremnike u glavnoj memoriji. Da bi dohvatili podatke prvo se u aplikaciji inicijalizira sredstvo koje predstavlja međuspremnik i potom se to sredstvo definira kao *unordered access view* (UAV) tip pogleda (eng. *view*). To je međuspremnik u koji se može pisati i iz kojeg se može čitati u odnosu na program na grafičkom sklopovlju, što mu omogućuje da zapiše rezultate svojih izračuna u taj međuspremnik. Veličina ovog spremnika mora biti jednaka broju pokrenutih dretvi programa za izračune opće namjene, jer svaka dretva preko svog globalnog prostornog otklona definira lokaciju u međuspremniku na koju će se zapisati rezultati. Ovaj tip pogleda za međuspremnike dostupan je i na programima za obradu vrhova, ali samo na sklopovlju koje podržava DirectX 11 [29].

Da bi podatke poslali nekom programu koji se izvodi na grafičkom sklopovlju također koristimo međuspremnike kao sredstvo, ali taj međuspremnik definiramo kao *shader*

resource view (SRV) pogled i u tom slučaju shader programi mogu samo čitati podatke iz tih međuspremnika. Kao primjer praktične primjene, pretpostavimo da procesor za izračune opće namjene izvršava neku simulaciju koja nam je bitna za konačne rezultate u procesu iscrtavanja slike na ekranu. Tada bi rezultate proračuna spremili u međuspremnik definiran kao UAV tip pogleda, a iz programa za obradu piksela bi pomoću SRV tipa pogleda pročitali podatke i napravili izračune prema njima. Međuspremnici se mogu definirati sa oba tipa pogleda čime se izbjegava potreba za kopiranjem podataka ukoliko se podatci programa za izračune opće namjene koriste u drugim stanjima grafičkog cjevovoda.

Pomoću SRV i UAV pogleda možemo poslati različitim programima različite podatke i pohraniti rezultate njihovog izvođenja, no što slučaju kada svim programima moramo poslati iste podatke. Kopiranje istih podataka u memoriji za svaki program na grafičkom procesoru koji smo pokrenuli čini se nepraktično i programski i vremenski. Za te slučajeve koristimo konstantne međuspremnike na shaderima. Umjesto kopiranja istih podataka u glavnoj memoriji, isti podatak iz memorije mapira se u konstantne međuspremnike svih pokrenutih programa istog tipa (npr. programi za obradu vrhova ili piksela). Praktični primjer bila bi transformacija vrhova u programu za obradu vrhova u koordinatni sustav kamere u sceni. Budući da se koristi ista transformacijska matrica za sve vrhove, umjesto da istu matricu kopiramo *n* puta u memoriji za svaki od *n* vrhova i onda pomoću SRV pogleda omogućimo svim programima pristup podatcima, tu matricu mapiramo u konstantne međuspremnike svakog programa za obradu vrhova.

Programi za izračune opće namjene svoj puni potencijal mogu ispuniti na sklopovlju koje podržava DirectX 11 grafički API, no moguće ih je izvoditi i na sklopovlju koje podržava DirectX 10. U tom slučaju procesori za izračune opće namjene pružaju manje mogućnosti od onih prethodno navedenih, a razlike možemo vidjeti u tablici 1.

Tablica 1 - Usporedba shader model 4.0 i shader model 5.0, te posljedice za mogućnosti procesor za izračune opće namjene (compute shader)a.

Mogućnosti	Compute Shader Model 4.0	Compute Shader Model 5.0	Posljedice
Organizacija grupa dretvi	2D	3D	Za SM 4.0 *W* i *Z* dimenzije su 1, dakle U× V × 1 za grupe i X × Y × 1 za dretve
Broj dretvi po grupi	768	1024	SM 5.0 može iskoristiti 33% veći broj dretvi po grupi za bolje performanse
Veličina dijeljene memorije unutar grupe dretvi	16 kB	32 kB	Dvostruko više memorije za pohranjihvanje dijeljenih varijabli i komunikaciju između dretvi
Pristup dijeljenoj memoriji	256 B *write-only* regija	Potpuni 32 kB *read/write* pristup	Učinkovitije čitanje i pisanje u zajedničku memoriju
Nedjeljive, kombinirane (*atomic*) operacije	Nepodržano	Podržano	Omogućava svakoj dretvi da izvršava operacije nad zaštićenim dijelovima memorije – pojednostavljuje implementaciju CPU algoritama na GPU
Dvostruka preciznost	Nepodržano	Podržano	Omogućava 64-bitnu dvostruku preciznost u proračunima
Broj dostupnih UAV pogleda za programe za izračune opće namjene	1	8	SM 5.0 omogućava svakom programu za izračune opće namjene pristup više međuspremnika odjednom
Broj dostupnih UAV pogleda za programe za obradu vrhova	Nepodržano	8	SM 5.0 omogućava programima za obradu piksela pristup podatcima programa za izračune opće namjene što omogućuje interoperabilost grafičkog cjevovoda

5. GENETSKO PROGRAMIRANJE

5.1. UVOD U GENETSKO PROGRAMIRANJE

Glavni cilj umjetne inteligncije, strojnog učenja, te širokog područja koje sadži elemente „strojne inteligencije“, su računala koja bi automatski rješavala probleme postavljene pred njih. Jedan od pionira u području strojnog učenja Arthur Samuel, u svom radu iz 1983. naslovljenom „*AI: Where it has been and where it is going*“, izjavio je da je glavni cilj strojnog učenja i umjetne inteligencije [30]: „naučiti strojeve da prikažu ponašanje koje bi, ukoliko prikazano od strane ljudi, podrazumijevalo da je uključena uporaba inteligencije.“

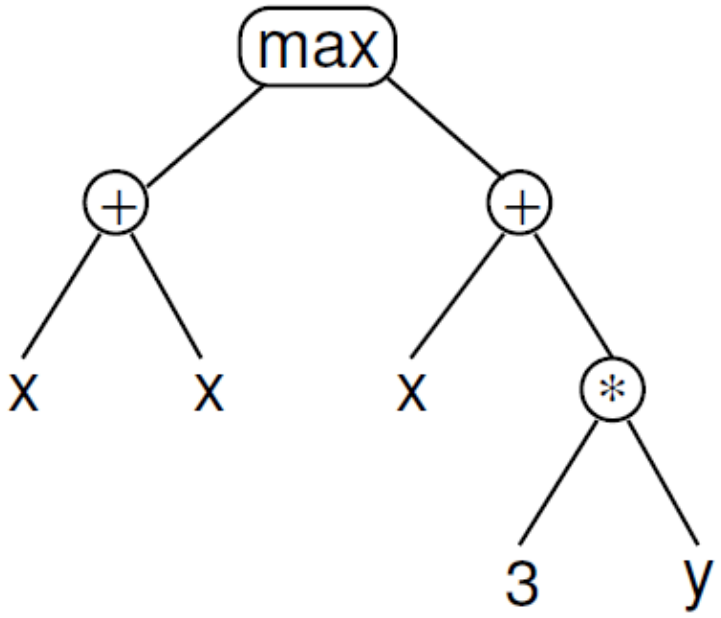
Genetsko programiranje (eng. *genetic* programming, GP) je tehnika evolucijskog računanja (eng. *evolutionary computation*, EC) koja je namijenjena rješavanju problema bez potrebe korisnika za poznavanjem ili specificiranjem moguće strukture rješenja. Ukoliko pogledamo genetsko programiranje na najvišoj razini apstrakcije, tada ga možemo smatrati semantičkom, domenski nezavisnom metodom za stvaranje programa koji računalima omogućuju automatsko rješavanje problema i koja na visokoj razini određuje kako ćemo do rješenja doći.

Početkom genetskog programiranja možemo smatrati 1954. godinu kada su se prvi puta počeli upotrebljavati evolucijski algoritmi u svrhu evolucijskih simulacija. U 1960-im i 1970-im godinama 20-og stoljeća, evolucijski algoritmi postali su široko priznati kao metode optimizacije.Prve oblike modernog genetskog programiranja baziranog na stablastim strukturama podataka dao je Nichael L. Cramer 1985-te godine. Njegov rad proširio je i nadopunio John R. Koza, glavni zagovornik GP-a, koji je među prvima genetsko programiranje upotrijebio za rješavanje raznih problema složenih aplikacija i pretraživanja. 1990-ih godina, genetsko programiranje koristilo se za rješavanje relativno jednostavnih problema budući da je to računski vrlo zahtjevna metoda. Danas, genetsko programiranje dobiva sve veći zamah zahvaljujući napretku u tom području i eksponencijalnom rastu brzine procesiranja, a ujedno i jako dobrim rezultatima u

područjima poput kvantnog računarstva, sortiranjima, pretraživanjima, industriji računalne zabave (igre), itd [31].

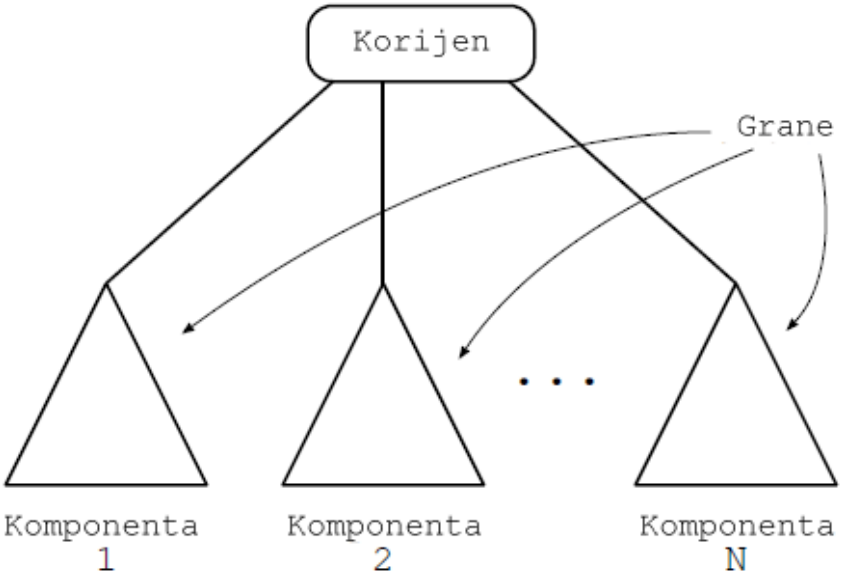
5.2. PRIKAZ JEDINKI

U genetskom programiranju programi se uobičajeno izražavaju kao sintaksna stabla umjesto da se prikazuju kao linije teksta programaa. Varijable i konstante programa nalaze se u listovima tog stabla i nazivaju se *terminali*, dok su aritmetički operatori unutarnji čvorovi koji se nazivaju *funkcije*. Skupovi dozvoljenih terminala i funkcija u GP zajedno formiraju skup primitiva GP sustava. Na slici 8 možemo vidjeti reprezentaciju jednog takvog programa, `max(x+x,x+3*y)`, u stablastoj strukturi.



Slika 8 (preuzeto iz [30])- Primjer stablaste strukture programa `max(x+x,x+3*y)` u genetskom programiranju (jedinka populacije).

U naprednijim oblicima genetskog programiranja, programi mogu sadržavati više komponenti. U tom slučaj prikaz koja se koristi u programu je skup stabala (jedno stablo za svaku komponentu) koja su grupirana pod jednim zajedničkim čvorom kako je prikazano na slici 9. Ova podstabla nazivamo granama. Broj i tip grana jednog programa, zajedno sa određenim svojstivma tih grana, tvore arhitekturu programa.



Slika 9 (preuzeto iz [30]) - Prikaz programa sa više komponenti.

U literaturi o genetskom programiranju uobičajeno je izraze programa predstaviti u prefiks notaciji. Npr. uzmimo program sa slike 8, izraz programa `je max(x+x, x+3*y)`, u prefiks notaciji taj izraz postaje `(max (+xx)( +x(*3y))`. Prefiks notacija omogućuje da se lakše uoči odnos između izraza i njegovih odgovarajućih podstabla.

Način na koji ćemo implementirati genetski program očito će velikim dijelom ovisiti o programskom jeziku i programskim knjižnicama koje koristimo. Jezici koji pružaju automatski *garbage collection* (brisanje dereferenciranih podataka iz memorije računala) i dinamičke liste podataka kao osnovne tipove podataka, omogućuju jednostavniju implementaciju stablastih struktura podataka i potrebnih genetskih operatora. Većina jezika koji se tradicionalno koriste u istraživanju umjetne inteligencije (npr. Lisp, Prolog), mnogi noviji jezici (npr. Ruby, Phyton), te mnogi znanstveni programski alati (npr. MATLAB, Mathematica) pružaju te mogućnosti.

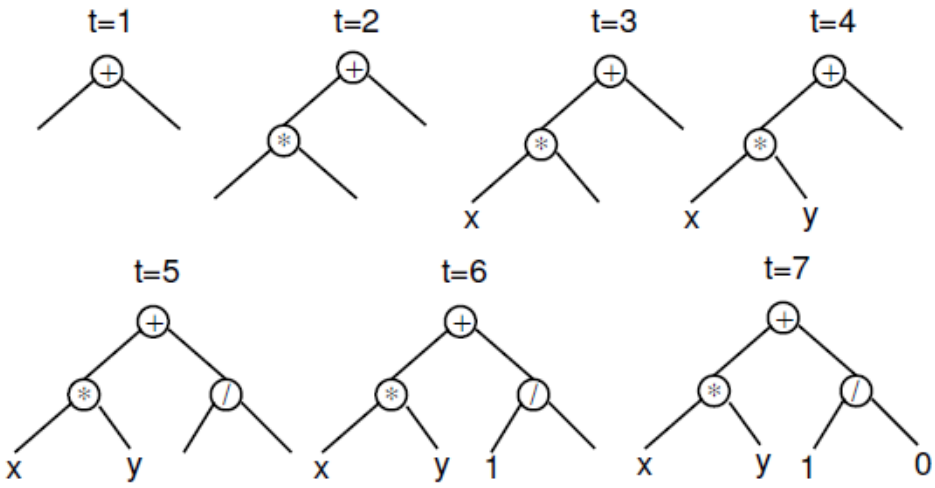
U okruženjima visokih performansi, stablasti prikaz programa može biti prilično neefikasan budući da se zahtjeva pohrana i upravljanje brojnim pokazivačima. U nekim slučajevima, može biti poželjna uporaba primitiva koje primaju proizvoljan broj argumenata, no danas je prilično uobičajeno da se u GP aplikacijama koriste funkcije sa fiksnim brojem argumenata. U tom slučaju zagrade teksta programa prefiksne

reprezentacije izraza su nepotrebne i stablaste strukture mogu se napisati jednostavnim linearnim izrazom. U slučaju programa sa slike 8 prefiksna notacija sa zagradama `(max (+xx)( +x(*3y))` prelazi u sljedeći oblik: `max + x x + x * 3 y`. Odabir linearne ili stablaste reprezentacije programa u genetskim programima ovisi o jednostavnosti, učinkovitosti, genetskim operatorima koji se koriste i ostalim podatcima o kojima program ovisi. Prikaz programa stablastom strukturom najviše se koristi u genetskom programiranju i mnoge kvalitetne, slobodno dostupne implementacije ih koriste.

5.3. INICIJALIZACIJA POPULACIJE

Kao i u drugim evolucijskim algoritmima, jedinke početne populacije generiraju se slučajnim odabirom. Postoji više različitih pristupa za generiranje te početne populacije, a spomenuti ćemo dvije jednostavnije metode i njihovu široko korištenu kombinaciju.

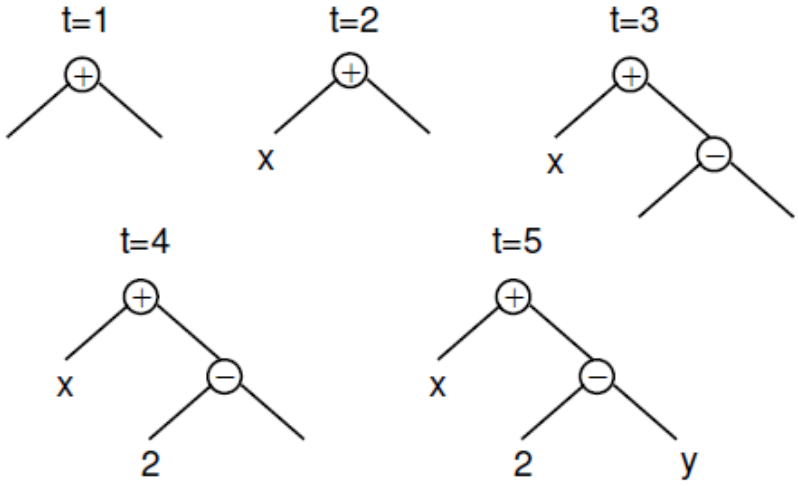
Prva metoda za generiranje početne populacije naziva se *full* metoda. Teksta programa *full* metode svako stablo u populaciji gradi se tako da se iz skupa funkcija odabiru elementi i postavljaju kao vrijednosti čvorova. Kada se dosegne maksimalna dubina stabla mogu se odabrati samo elementi iz skupa terminala koji čine vrijednosti u listovima. Dubina nekog čvora određena je brojem bridova koji se moraju proći od korijenskog čvora do našeg ciljnog čvora. *Full* metoda dobila je ime po tome što su sva stabla generirana ovom metodom potpuna. Na slici 10 možemo vidjeti kako se stablo generira pomoću full metode kroz različite vremenske trenutke.



Slika 10 (preuzeto iz [30]) - Generiranje stabla *full* metodom.

Iako *full* metoda generira stabla koja imaju listove na istoj dubini, to ne znači nužno da će sva stabla početne populacije imati jednak broj čvorova (veličina stabla) ili da će biti istog oblika. Ovo se jedino događa kada sve funkcije u skupu primitiva primaju jednak broj argumenata, no čak i kada se u skupu primitiva nalaze funkcije koje primaju različit broj argumenata, veličine različitih stabala i njihovi oblici biti će ograničeni.

Grow, kao druga jednostavna metoda, generira stabla koja sadrže više varijacije što se tiče veličine i oblika pojedinih stabala u odnosu na *full* metodu. Vrijednosti čvorova odabiru se iz cijelog skupa primitiva dok se ne dosegne maksimalna dubina stabla. Kada se dosegne maksimalna dubina mogu se odabrati samo elementi iz skupa terminala (ekvivalentno *full* metodi). Na slici 11 možemo vidjeti konstrukciju stabla pomoću *grow* metode do maksimalne dubine 2. Na slici vidimo da je prvi odabrani argument korijenskog čvora (+) iz skupa terminala (x), njegovim dodavanjem u lijevu granu završava se daljnje proširivanje te grane i prije nego je dosegnuta maksimalna dubina. Drugi argument korijenskog čvora je funkcija (-) i dodana je u desnu granu, no argumenti te nove funkcije mogu biti samo terminali zato jer bi inače premašili zadanu maksimalnu dubinu za stablo.



Slika 11 (preuzeto iz [30]) - Generiranje stabla *grow* metodom.

Budući da zapravo niti *grow* niti *full* metoda same po sebi ne pružaju veliku raznolikost veličina i oblika stabala u populaciji, za inicijalizaciju početne populacije možemo koristiti kombinaciju te dvije metode koja se zove *ramped half-and-half.* Polovica početne populacije konstruira se *full* metodom, a druga polovica populacije *grow* metodom. To se postiže korištenjem različitih maksimalnih dubina stabala, čime se osigurava da populacija sadrži jedinke raznolikih veličina i oblika. Iako su ove metode jednostavne za implementaciju i korištenje često je teško kontrolirati statističku distribuciju bitnih svojstava, poput veličina i oblika generiranih stabala. Veličine i oblici stabala generiranih *grow* metodom ovise o veličinama skupa funkcija i skupa terminala. Ako je skup terminala puno veći od skupa funkcija, *grow* metoda generirati će uglavnom stabla male dubine, bez obzira na maksimalnu dubinu. Ako je skup funkcija puno veći od skupa terminala, *grow* metoda ponašati će se slično *full* metodi. Početna populacija ne mora biti u potpunosti slučajno generirana, ukoliko su poznata željena svojstva konačnog rješenja, stabla sa tim svojstvima mogu se definirati i dodati u početnu populaciju.

5.4. SELEKCIJA

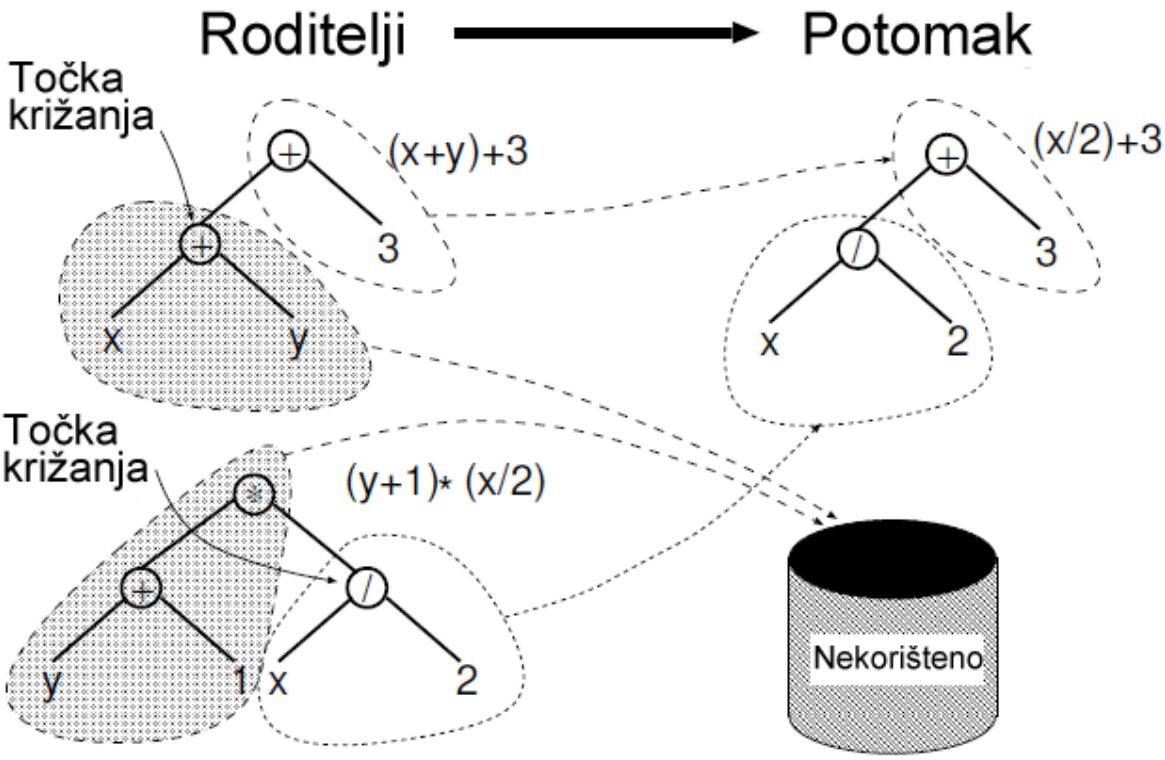
Kao i u svim drugim evolucijskim algoritmima, genetski operatori u genetskom programiranju primjenjuju se na pojedine jedinke koje su probabilistički odabrane na

temelju njihove dobrote. To znači da će bolje jedinke vjerojatnije imati više djece programa od lošijih jedinki. Najčešće korištena metoda odabira jedinki je turnirska selekcija. Sljedeća najčešće korištena metoda odabire jedinke na temelju iznosa njihove dobrote, no općenito se može koristiti bilo koja standardna metoda za selekciju jedinki.

U turnirskoj selekciji odabire se nasumice proizvoljan broj jedinki iz populacije. One se međusobno uspoređuju i ona sa najboljom dobrotom se odabire kao roditelj, no za križanje su nam potrebna dva roditelja, pa se turnirska selekcija obavlja još jednom, no ovaj puta bez prethodno odabranih jedniki. Treba pimjetiti da se kod turnirske selekcije gleda samo je li jedna jedinka bolja od druge, a ne se za koliko bolja. Na ovaj način se pritisak selekcije na populaciju ostavlja konstantnim i jedna jedinka koja ima najbolju dobrotu ne može preplaviti sljdeću generaciju svojom djecom. Kada to ne bi bilo tako, došlo bi do ubrzanog gubitka raznolikosti populacije sa potencijalno kobnim posljedicama za izvođenje GP programa. Iako preferira najbolje jedinke, turnirska selekcija omogućava da i prosječno dobre jedinke imaju šansu biti roditelj i imati djecu.

5.5. KRIŽANJE I MUTACIJA

Genetsko programiranje se značajno razlikuje od ostalih evolucijskih algoritama kada se radi o implementaciji operatora križanja i mutacije. Najčešće korišteni tip križanja je križanje podstabala (eng. *subtree crossover*). Kod ovog tipa križanja u svakom roditelju se nezavisno i nasumce odabere jedna točka križanja, tj. čvor. Potomak se potom stvara tako da se kopira prvi roditelj, a podstablo sa korijenskim čvorom u točki križanja kopije prvog roditelja mijenja se sa podstablom koje ima korijenski čvor u točki križanja drugog roditelja (ilustrirano na slici 12). Kopije se koriste da se ne bi promijenile vrijednosti čvorova orginalnog roditelja. Na ovaj način jedinka koja je odabrana više puta za roditelja može sudjelovati u stvaranju više potomaka. Također, moguće je definirati križanje koje daje dva potomka, no to se uobičajeno ne koristi.



Slika 13 (preuzeto iz [30]) - primjer križanja u genetskom programiranju.

Točke križanja često se ne biraju sa uniformnom vjerojatnošću. Tipični skup primitiva dovodi do stabala koja imaju prosječni faktor grananja najmanje jednak dva, stoga će većina čvorova biti listovi. Uniforma selekcija točki za križanje ima za posljedicu to da se u procesu križanja često izmjenjuju samo male količine genetskog materijala (mala podstabla). Zapravo moguće je da se mnoga križanja svedu samo na zamjenu listova. Kako bi se to izbjeglo koristi se pristup u kojem se za križanje u 90% slučajeva odabire točka koja sadži funkciju, a samo u 10% slučajeva list.

Najčešće korišteni oblik mutacije u genetskim programima je mutacija podstabla (eng. *subtree mutation*). Slučajno se odabere točka mutacije u stablu i cijelo podstablo sa korijenom u toj točki se mijenja sa slučajno generiranim podstablom. Ovaj način križanja se često implementira kao križanje između nove slučajno generirane jedinke i jedinke koju mutiramo.

Drugi često korišteni oblik mutacije je mutacija točke (eng. *point mutation*). To je

ekvivalent promjeni vrijednosti bita u genetskim algoritmima. Teksta programa mutacije točke slučajno odaberemo čvor u stablu i primitiva koja je tamo pohranjena mijenja se sa drugom slučajno odabranom primitivom iz skupa primitiva koja sadrži jednaki broj argumenata (broj podstabala čvora). Ukoliko ne postoji druga primitiva u skupu primitiva koja podržava jednaki broj argumenata, čvor se ne mutira, no tada se može odabrati drugi čvor za mutiranje.

Ukoliko se koristi mutiranje podstabla, samo će jedno podstablo biti zahvaćeno. Mutacija točke sa druge strane tipično se primjenjuje tako da se gleda svaki čvor zasebno i mutira se sa određenom vjerojatnošću.
Odabir navedenih operatora u stvaranju novih jedinki u populaciji potpuno je vjerojatnosni. Operatori u GP su normalno međusobno isključivi, za razliku od evolucijskih algoritama gdje se potomci ponekad dobivaju kompozicijom operatora. Vjerojatnost primjene određenog operatora naziva se stopa primjene operatora (eng. *operator rates*). Uobičajeno je da se križanje primjenjuje sa najvećom stopom primjene, 90% ili više. Mutacija ima puno manju stopu primjene koja se kreće oko vrijednosti od približno 1%.

Kada kombinirana vrijednost stope križanja i stope mutacije iznosi *p*, a taj iznos je manji od 100%, moguće je koristiti i operator reprodukcije (eng. *reproduction*) čija stopa primjene iznosi 1 – *p.* Operator reprodukcije jednostavno odabere jedinku iz populacije na temelju njene dobrote i u novu generaciju umetne kopiju te jedinke.

6. OSTVARENJE GENETSKOG PROGRAMIRANJA POMOĆU DIRECTX-a I PROGRAMA ZA IZRAČUNE OPĆE NAMJENE

6.1. OSNOVE PROGRAMIRANJA PROGRAMA ZA IZRAČUNE OPĆE NAMJENE

Razmotrimo jedan jednostavan problem. Recimo da postoji 1000 parova varijabli čije su vrijednosti poznate, koje treba zbrojiti i to je potrebno napraviti paralelno na GPU pomoću programa za izračune opće namjene. Prvi korak je priprema podataka za grafičko sklopovlje, a najbolji način za to je definirati strukturu podataka koja će sadržavati obje vrijednosti koje treba zbrojiti, te inicijalizirati polje tih struktura koje će sadržavati sve parove za zbrajanje. Kako bi bilo moguće podatke proslijedili grafičkom procesoru potrebno je prvo inicijalizirati sučelje prema grafičkom sklopovlju. Time se dobivaju dvije komponente (spomenute u poglavlju o DirectX-u), komponentu uređaja i komponentu konteksta uređaja, koje se koriste za inicijaliziranje sredstava, postavljanje programa za izračune opće namjene u grafički cjevovod, slanje podataka i dohvaćanje podataka sa grafičkog sklopovlja, te za pokretanje programa za izračune opće namjene.

Nakon što su inicijalizirane komponene uređaja i konteksta uređaja, potrebno je inicijalizirati program za izračune opće namjene. U ovom primjeru, program za izračune opće namjene prevodi se u strojni tekst programa unutar programa (*run-time compile*), a nakon prevođenja dobiva se sučelje prema programu za izračune opće namjene i njegov međukod (eng. *byte-code*). Ukoliko se taj isti program za izračune opće namjene želi koristiti u drugim programima, moguće je njegov međukod pohraniti u datoteku, te ga u drugom programu samo pročitati iz datoteke i inicijalizirati novo sučelje bez potrebe za ponovnim prevođenjem.

Za ovaj primjer potrebna su dva sredstva i oba sredstva su međuspremnici. U DirectX API-ju međuspremnici se mogu inicijalizirati bez podataka ili im se može predati pokazivač na podatke u memoriji. Kada se međuspremnik inicijalizira bez podataka samo se zauzme specificirana količina memorije u koju kasnije možemo pohraniti podatke, a kada se

inicijalizacija vrši sa postavljenim početnim podatcima, tada se oni kopiraju u memoriju koju smo zauzeli inicijalizacijom tog međuspremnika.

Prvi od dva međuspremnika inicijalizira se tako da se funkciji koja stvara međuspremnik predaje pokazivač na polje parova vrijednosti koje treba zbrojiti, dok se drugi međuspremnik inicijalizira prazan, s tim da oba međuspremnika zauzimaju istu količinu memorije kao i podatci koje je potrebno zbrojiti. Nakon inicijalizacije, prvi međuspremnik označi se kao SRV (*Shader Resource View*) pogled, dok se drugi označi kao UAV (*Unordered Access View*) pogled. Na temelju tipa pogleda, program za izračune opće namjene zna koji međuspremnici su samo za čitanje, a koji su za čitanje i pisanje. U ovom slučaju prvi međuspremnik označen je samo za čitanje, dok drugi međuspremnik može poslužiti za operacije čitanja i pisanja.

Nakon što su svi potrebni podatci spremljeni u međuspremnike može se pokrenuti izvođenje programa za izračune opće namjene. Prvo se taj program postavlja u grafički cjevovod, vežu se SRV i UAV međuspremnici za njega i zatim se pokreće njegovo izvođenje pomoću komponente konteksta uređaja, tj. sa njenom funkcijom *Dispatch*. Funkcija *Dispatch* kao argumente prima broj grupa u *x*, *y* i *z* smjeru, tj. ukupan broj grupa dretvi koje će biti inicijalizirane (x·y·z), dok je broj dretvi po grupi zapisan u prgramu za izračune opće namjene (u našem slučaju je to jedna dretva po svakoj komponenti). Program za izračune opće namjene nakon pokretanja jednostavno pročita sadržaj SRV međuspremnika, zbroji dobivene podatke i pohrani rezultat u UAV međuspremnik. Budući da su parovi varijabli spremljeni u strukturu koja sadrži dva člana, odgovarajuću strukturu potrebno je definirati i u programu za izračune opće namjene. Kada je program za izračune opće namjene završio sa radom, podatci se mogu dohvatiti. Podatci se dohvaćaju preko komponente konteksta uređaja i njene funkcije *Map*, koja pruža pristup sredstvu u koje je program pohranio podatke. Kako je rezultat zbrajanja cjelobrojna varijabla i UAV međuspremnik sadržavat će samo cjelobrojne rezultate, a *Map* funkcija vratiti će pokazivač na te rezultate nakon što se oni prenesu u radnu memoriju. Pseudokod programa opisanog primjera i pripadni program za izračune opće namjene prikazani su na slici 13.

```
/****************** compute shader pseudokod ******************/

struktura parovi{ int a, int b }

ČitajMeđuspremnik<parovi> ulaz;
ČitajPišiMeđuspremnik<int> izlaz;

[Broj_dretvi(1, 1, 1)]
void CSMain( ID_dretve ) {
    izlaz[ID_dretve] = ulaz[ID_dretve].a + ulaz[ID_dretve].b;
}

/******************** program pseudokod **********************/

struktura parovi { int a, int b }

int main()
{
    // Napunimo podatcima ...
    polje parovi[1000];
    napuni_brojevima( *parovi );

    // Inicijaliziramo sučelje prema sklopovlju ...
    (Device, Context) = inicijaliziraj_sučelje_prema_GPU();

    // Inicijaliziramo compute shader ...
    ComputeShader = Prevedi_i_inicijaliziraj( ime_datoteke.hlsl );

    // Inicijalizacija sredstava ...
    SRVspremnik = Device->napravi_međuspremnik( *parovi, 1000 );
    Postavi_kao_SRV( SRVspremnik );

    UAVspremnik = Devce->napravi_međuspremnik( NULL, 1000 );
    Postavi_kao_UAV( UAVspremnik );

    // Pokrenemo compute shader ...
    Context->PostaviCS ( ComputeShader );
    Context->PostaviSredstvaCS( SRVspremnik, UAVspremnik );
    Context->Dispatch( 1000, 1, 1 );

    // Dohvatimo podatke ...
    int *rezultati = (int *)Context->Map( UAVspremnik );

    // Kraj ...
    Izlaz();
}
```

Slika 14 - Pseudoteksta programa opisanog primjera i odgovarajući procesor za izračune opće namjene (compute

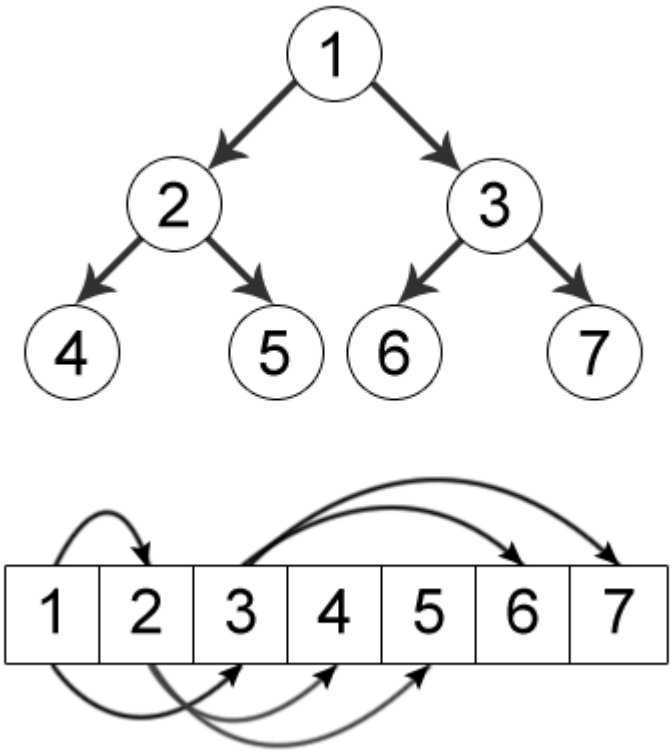
6.2. PRIKAZ PODATAKA NA GPU

Glavni problem kod pisanja programa za grafičko sklopovlje je prikaz podataka u programima pisanim za grafički procesor. Budući da HLSL jezik ne pruža mogućnosti dinamičke alokacije memorije, veličine polja i broj podataka koje će programi primiti moraju biti unaprijed poznate veličine. Isti takav prikaz podataka mora biti i u dijelu programa koji se odvija na centralnom procesoru. Budući da su jedinke populacije u genetskom programiranju prikazane u obliku strukture stabla, klasični programski model u kojem se koriste pokazivači na sljedeće čvorove u stablu ne može se prenijeti na grafičko sklopovlje. Stoga se u programu koriste strukture koje će imati konstantnu veličinu zadanu prilikom prevođenja programa.

Određivanje konstantne veličine strukture prilikom prevođenja osigurati će da se svi podatci strukture u memoriji nalaze slijedno poredani jedni iza drugih. Kada se inicijalizira sredsvo sa nekim podatcima, npr. međuspremnik, tada se u inicijalizacijsku funkciju predaje adresa prvog elementa strukture u memoriji i količina memorije koju ta struktura zauzima. Tako će se svi elementi strukture, počevši od prvog elementa na temelju dane adrese do zadnjeg elementa određenog prema količini memorije koju ta struktura zauzima, prenijeti u sredstvo. Dakle umjesto da se koristi klasičan prikaz stablastih struktura podataka u memoriji pomoću pokazivača, koristi se indeksirano polje struktura. Svaki element polja je jedan čvor u stablu prikazan odgovarajućom strukturom, a veličina polja ovisi o maksimalnoj dubini stabla. Veličina polja mora se odrediti prilikom prevođenja programa i mora biti konstantna. Npr. ako je stablo binarno, maksimalna dubina stabla je *n*, a maksimalni broj čvorova u stablu određen je izrazom $2^{n+1}-1$ (npr. za *n* = 2, maksimalni broj čvorova je 7), a budući da maksimalnu dubinu znamo prilikom prevođenja, pomoću nje možemo odrediti i maksimalni broj čvorova u polju. Iako ovaj primjer vrijedi samo za binarna stabla, maksimalni broj čvorova do određene dubine moguće je odrediti za bilo koju vrstu stabala, pa pristup prikazu stablastih struktura podataka preko indeksiranih polja možemo i dalje koristiti.

Razmotrimo klasično binarno stablo - jedan čvor tog stabla je struktura koja sadrži varijable za spremanje određenih podataka i tipično dva pokazivača na memorijske

lokacije čvorova djece. Ukoliko se pokuša zadržati ta forma, jedna struktura u indeksiranom polju sadržavat će varijable za podatke i dvije varijable koje će sadržavati indekse elemenata u polju koji su čvorovi djeca tog elementa. Na ovaj način postiže se isto međusobno referenciranje elemenata u polju kao da se radi o klasičnoj stablastoj strukturi podataka, memorijske lokacije zamijenili su indeksi zbog čega ovaj prikaz stabla pomoću polja struktura možemo nazvati indeksiranim prikazom stabla. Slika 14 prikazuje klasično stablo sa sedam čvorova i jedan mogući odnos čvorova indeksiranog stabla.



Slika 15 - Klasični prikaz strukture stabla i prikaz mogućeg odnosa u indeksiranom stablu.

Ovaj način prikaza stabla pogodan je i za programe pisane za grafički procesor, koristeći naredbe za kontrolu toka i petlje možemo manipulirati elementima tog stabla. Dvije vrijednosti koje su bitne za svako stablo su maksimalni broj čvorova i maksimalna dubina stabla. Budući da u programima za operacije nad indeksiranim stablom najčešće koristimo petlje, potrebno je znati do kojeg elementa maksimalno možemo ići. Iako je maksimalni broj čvorova moguće izračunati iz maksimalne dubine stabla, možemo izbjeći konstantno računanje broja elemenata ako se ta vrijednost spremi u varijablu i poveže sa tim stablom. Uzimajući sve navedeno u obzir dolazimo do konačne strukture jednog stabla koja sadrži polje čvorova stabla, maksimalnu dubinu stabla i maksimalni broj elemenata, a

čvorovi u polju sadrže određene podatke i indekse čvorova djece u tom polju. Na slici 15 prikazana je takva struktura podataka u programskom jeziku C.

```
/* Struktura jednog čvora stabla */

struct Čvor
{
  int podatak;
  int čvor_lijevo;
  int čvor_desno;
}
```

```
/* Konačna struktura koja sadrži sve podatke o stablu */

struct Stablo
{
  int max_dubina;
  int max_broj_čvorova;
  čvor čvorovi_stabla[max_broj_čvorova];
}
```

Slika 16 - Struktura podataka indeksiranog stabla u C jeziku.

6.3. PROGRAMSKO RJEŠENJE GENETSKOG PROGRAMA ZA GPU

Genetsko programiranje samo po sebi za posljedicu ima programska ostvarenja koja su računski vrlo zahtjevna. Zbog velikog broja podataka i faktora vjerojatnosti koji je karakterističan za genetsko programiranje teško je odrediti vrijeme izvođenja takvog programa, ali cilj je pomoću visoko paralelne platforme smanjiti prosječna vremena potrebna za izvođenje takvih programa.

Općenito, svaki genetski program sastoji se od tri bitna dijela: selekcija, križanje i određivanje dobrote jedinki u populaciji. Klasične genetske programe možemo prikazati pomoću ova tri dijela koji se ponavljaju u petlji dok uvjet za zaustavljanje nije postignut ili se dogodila neka greška. Uobičajeni genetski programi imaju dva uvjeta za zaustavljanje, prvi je da je pronađeno dovoljno dobro rješenje problema koji rješavamo ili je izvršen zadani broj iteracija programa i koristi se najbolje rješenje do tada pronađeno.

Algoritam koji će se izvršavati na grafičkom sklopovlju sadržavati će iste dijelove kao i bilo koji drugi klasični genetski program, pitanje je jedino koje se dijelove programa isplati paralelizirati, a koje ne. Pri tome moramo uzeti u obzir da prijenos podataka na grafičko

sklopovlje zahtjeva određeni dio vremena, a isto tako i dohvat podataka. Također, shader programi koji se izvode na GPU moraju biti što jednostavniji sa što manje naredbi za kontrolu toka koje mogu dosta negativno djelovati na performanse programa. No budući da se ovdje radi o jednoj kompleksnoj strukturi podataka, naredbe za kontrolu toka u programima za grafički procesor su neizbježne, stoga se potrebno orijentirati na efikasno upravljanje tim podatcima i iskoristiti svojstva koja pruža DirectX API i grafičko sklopovlje.

6.3.1. DEFINICIJA PROBLEMA I INICIJALIZACIJA PROGRAMA

Iako postoji široka lepeza složenih i manje složenih problema koji se mogu rješavati, razmotrimo jedan jednostavniji problem iz domene simboličke regresije. Recimo da su poznate točke u dvodimenzionalnome koordinatnom sustavu koje predstavljaju neke podatke i da je potrebno odrediti funkciju koja bi te podatke opisivala. Ovaj problem može se rješiti tako da se definira skup terminala i funkcija, te pomoću genetskog programa i zadanih parova točaka (x, y) nađe funkcija koja bi najbolje opisivala te podatke. U programu se traži kvadratna funkcija $y=x^2+x+1$, na temelju vrijednosti te funkcije za točke od [-5, 5], te skupa funkcija sa dva argumenta {+, -, ÷, ×} i skupa terminala {1, 2, 3, 4, 5, 6, 7, 8, 9, x}.

Radi zadržavanja jednostavnosti, sve jedinke populacije generiraju se *full* metodom za inicijalizaciju strukture podataka, što znači da su sva stabla u populaciji potpuna, a podatci u čvorovima stabla slučajnim odabirom se izvlače ili iz skupa funkcija (unutarnji čvorovi stabla) ili iz skupa terminala (vanjski čvorovi stabla).

6.3.2. 3-TURNIRSKA SELEKCIJA I KRIŽANJE

Način na koji se odabiru jedinke za križanje u programu je 3-turnirska selekcija. Kod 3-turnirske selekcije iz populacije se slučajnim odabirom naprave grupe po tri jedinke i zatim se iz svake grupe odaberu dvije jedinke sa boljim dobrotama. Budući da ovaj

postupak nije računski zahtjevan i podatkovna ovisnost je samo unutar grupe, turnirska selekcija predstavlja dobar odabir za paralelizaciju na grafičkom sklopovlju.

U dijelu programa koji se odvija na centralnom procesoru odaberu se jedinke za turnirsku selekciju zajedno njihovim dobrotama i ti podatci pošalju se grafičkom sklopovlju. Prilikom pokretanja programa za izračune opće namjene (*compute shader*) koji obavlja 3-turnirsku selekcij, ukupan broj grupa dretvi koji definiramo biti će tri puta manji od ukupnog broja jedinki u selekciji. To znači da će svaka grupa dretvi obrađivati jednu grupu od tri jedinke odabranih za 3-turnirsku selekciju. Unutar svake grupe dretvi pokrenute su tri dretve, a svaka dretva obrađuje podatke vezane uz jednu jedinku iz pripadne grupe jedinki za 3-turnirsku selekciju. Na taj način svaka grupa od tri jedinke za 3-turnirsku selekciju dobiva jednu grupu od tri dretve koja će napraviti selekciju.

U programu za izračune opće namjene za svaku grupu dretvi definirano je dijeljeno polje od tri elementa u koje dretve mogu zapisivati i čitati podatke. Nakon primitka podataka svaka dretva na odgovarajući indeks u tom polju zapisuje dobrotu jedinke koju je dobila. Nakon sinkronizacije dretvi (moramo pričekati da sve dretve završe sa pisanjem podataka prije nego se podatci počnu čitati) svaka dretva uspoređuje dobrotu svoje jedinke sa dobrotama jedinki drugih dretvi i gleda je li dobrota njene jedinke lošija od dobrota druge dvije jedinke. Ukoliko to nije slučaj, tada će jedinka koju je ta dretva primila sudjelovati u procesu križanja, a ukoliko jest, tada će ta jedinka biti zamijenjena novom jedinkom koja će biti rezultat križanja dvije bolje jedinke.

Nakon što su odabrane jedinke za križanje mogu se odabrati dva načina na koji će se dalje obrađivati podatci u programu. Kod prvog načina, informacije o tome koje su jedinke odabrane za križanje vraćaju se u dio programa koji se odvija na centralnom procesoru. U tom slučaju podatci se moraju prvo dohvatiti sa grafičkog sklopovlja, obraditi na centralnom procesoru (npr. izvlačenje jedinki za križanje iz populacije na temelju informacija od programa za izračune opće namjene) i zatim izvršiti križanje na centralnom procesoru ili podatke za križanje poslati drugom programu za izračune opće namjene na obradu, no to sve za sobom povlači velike vremenske troškove.

Drugi i puno efikasniji način je križanje izvršiti sa programom za izračune opće namjene odmah nakon 3-turnirske selekcije. Ovako se izbjegava slanje podataka sa grafičkog sklopovlja u dio aplikacije koji se izvršava na centralnom procesoru i dodatnu obradu tih podataka. Za križanje pomoću programa za izračune opće namjene definira se novo dijeljeno polje podataka u koje će dretve sa boljom dobrotom zapisivati podatke svojih jedinki koji su bitni za proces križanja, tj. samo čvorove koji će sačinjavati novu jedinku. Kada obje dretve završe proces pisanja, dretva koja je imala jedinku sa najlošijom dobrotom čita podatke koje su ostale dvije dretve zapisale i pohranjuje ih u svoj izlazni međuspremnik. Ostale dvije dretve ne pohranjuju ništa u izlazni međuspremnik budući da njihove jedinke nisu mijenjane.

Iako je i sam proces križanja relativno kompleksan za GPU s obzirom da postoji puno naredbi za kontrolu toka, ušteda koju dobijamo na vremenu ako kombiniramo selekciju i križanje na jednom mjestu veća je nego da smo ta dva procesa razdvojili.

6.3.3. ODREĐIVANJE DOBROTE

Budući da svaka jedinka u populaciji predstavlja neku funkciju, da bi se odredila dobrota te jedinke potrebno ju je evaluirati i usporediti rezultate sa rezultatima tražene funkcije. Svaka jedinka evaluira se za svaku točku u intervalu [-5, 5], a dobiveni rezultat za svaku točku se oduzima od vrijednosti tražene funkcije za odgovarajuću točku. Od svake razlike uzima se apsolutna vrijednost, a na kraju se apsolutne vrijednosti svih razlika zbroje i daju konačnu dobrotu jedinke. Na ovaj način zapravo ne dobivamo dobrotu jedinke, već njenu kaznu. To znači da jedinke sa manjom kaznom bolje aproksimiraju ciljnu funkciju na danom intervalu od jedinki sa većom kaznom, tj. rezultati jedinke za točke u danom intervalu vrlo su bliske rezultatima tražene funkcije ako je kazna mala.

Iako se evaluacija dobrote ili kazne pojedine jedinke ne čini tako zahtjevnom, zbog određenih ograničenja procesora za izračune opće namjene (velika složenost kontrole toka koja smanjuje performanse) paralelizacija ovog postupka dovela je do ukupne degradacije performansi izrađenog programa, što će kasnije biti i prikazano. Zbog toga se

ovaj postupak implementira za izvođenje na centralnom procesoru i time uvelike smanjuje vremenske troškove programa.

6.3.4. LINEARNI KONGRUENTNI GENERATOR SLUČAJNIH BROJEVA

Jedan od velikih nedostataka HLSL jezika je nedostatak funkcije za generiranje slučajnih brojeva. Kada bi takva funkcija postojala, tada bi u programima koji uvelike ovise o generiranju slučajnih brojeva mogli jednim pozivom funkcije paralelno generirati veliku količinu pseudoslučajnih brojeva. No iako takva funkcija nije implementirana u HLSL jeziku, moguće je na temelju algoritma za generiranje pseudoslučajnih brojeva implementirati jedan takav generator pomoću kojeg bi onda vrlo brzo mogli generirati veliku količinu pseudoslučajnih brojeva za korištenje u programu.

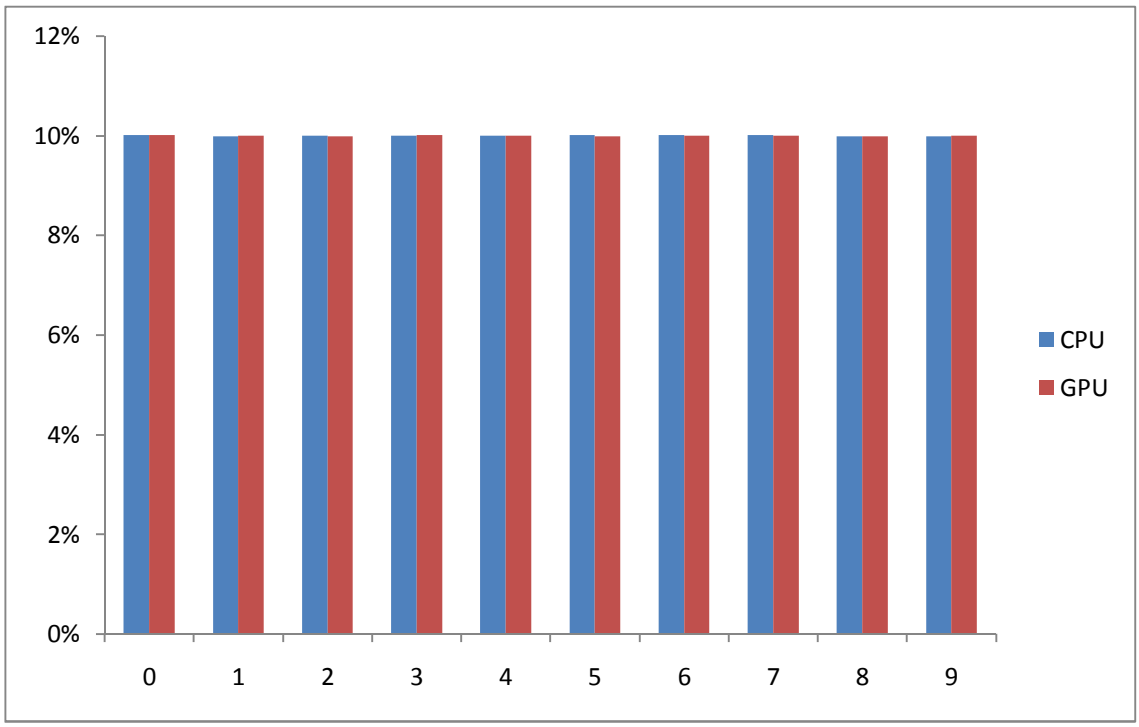
Jedan od takvih postupaka za generiranje slučajnih brojeva je i linearni kongruentni generator (eng. *Linear Congruential Generator*, LCG) pseudoslučajnih brojeva koji je implementiran u *rand* funkcije mnogih prevodioca [32]. Generator možemo definirati pomoću ponavljajuće relacije (5):

$$X_{n+1} = (a \cdot X_n + c) \bmod(m) \qquad (5)$$

gdje je X_n slijed pseudoslučajnih vrijednosti, m ($0 < m$) je modul, a ($0 < a < m$) je množitelj, c ($0 \le c < m$) je korak povećanja, a X_0 ($0 \le X_0$) je početna vrijednost. Iako LCG daje vrlo dobre pseudoslučajne brojeve, vrlo je osjetljiv na odabir koeficijanata *c*, *m* i *a*, što može dovesti do neučinkovitih implementacija. LCG generator je vrlo brz i zahtjeva minimalne količine memorije, ali ne bi se trebao koristiti u slučajevima kada se zahtjeva pseudoslučajni generator visoke kvalitete.

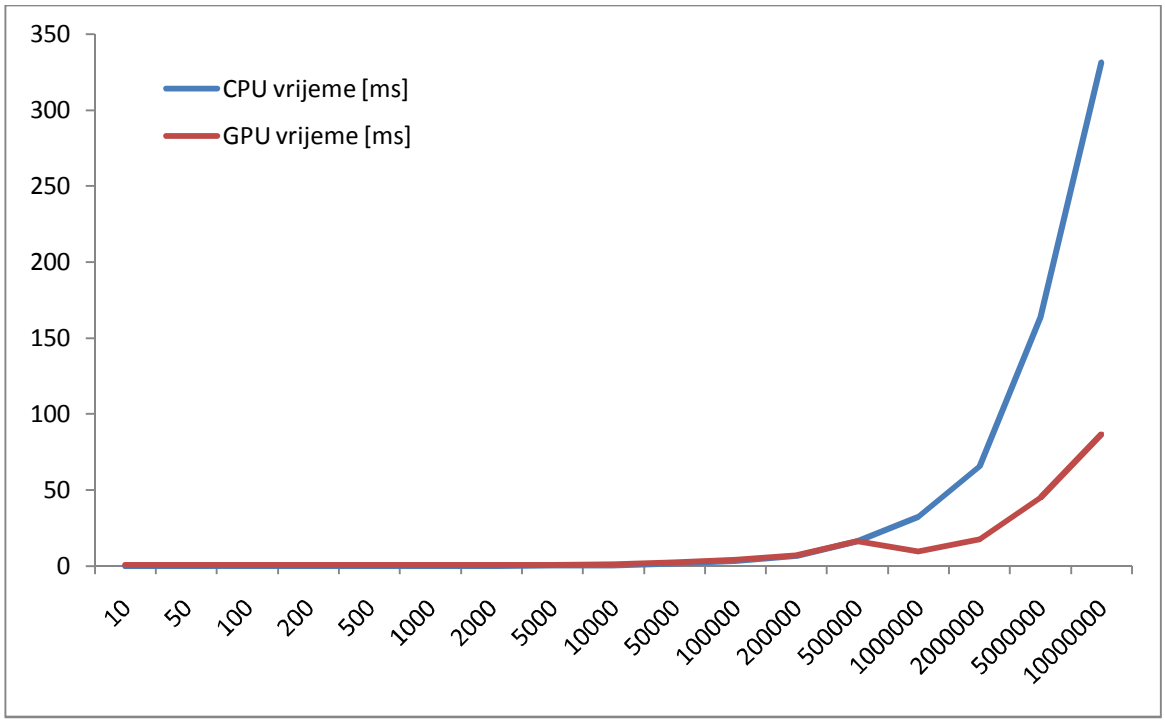
U ovom radu odabrani su sljedeći koeficijenti: *a* = 214013, *c* = 2531011 i *m* = 2^{32}. Iste ove koeficijente koristi i *rand* funkcija implementirana u Microsoftov C/C++ prevodioc. Kako bi se uvjerili da program za izračune opće namjene radi ispravno, napravimo usporedbu sa *rand* funkcijom. U ispitivanju je izvedeno stotinu iteracija generiranja pseudoslučajnih brojeva, u svakoj iteraciji se generiralo 10^5 slučajnih brojeva u intervalu (0, 9) i bilježena

je frekvencija ponavljanja svakog broja. Ukoliko generator radi ispravno, svaki će broj imati jednaku frekvenciju ponavljanja, a sa slike 16 možemo upravo to vidjeti.



Slika 17 - Usporedba frekvencije generiranja pseudoslučajnih brojeva u intervalu (0, 9) na centralnom procesoru (CPU) i grafičkom procesoru (GPU).

Što se tiče brzine, zbog potrebe prijenosa podataka na grafičko i njihovog dohvata, paralelni način generiranja pseudoslučajnih brojeva nije isplativ za male količine brojeva, no ukoliko taj broj pređe otprilike 500 000 potrebnih pseudoslučajnih brojeva, uštede u vremenu mogu biti znatne (usporedba prikazana na slici 17).



Slika 18 - Brzina generiranja različitih količina pseudoslučajnih brojeva.

6.3.5. CJELOVITI POGLED NA GENETSKI PROGRAM

Od tri glavna dijela svakog genetskog programa, selekcije, križanja i određivanje dobrote jedinki, uspješno je paralelizirana slekcija i križanje, dok je određivanje dobrote implementirano na centralnom procesoru.

Da bi upotpunili genetske operatore potrebno je spomenuti i mutaciju. Mutacija je općenito proces koji se vrlo rijetko događa ($\approx$1% slučajeva), računski nije zahtjevan i vrlo se efikasno može implementirati na centralnom procesoru. Iako se može koristiti ranije opisan način za generiranje pseudoslučajnih brojeva na grafičkom sklopovlju, troškovi prijenosa podataka na grafičko sklopovlje i potrebna količina pseudoslučajnih brojeva koja bi se trebala generirati kod paralelizacije ovog proces poništili bi eventualne dobitke u performansama.

Kod pokretanja programa prvo se obavlja inicijalizacija populacije jedinki, zatim se inicijalizira sučelje prema grafičkom sklopovlju, te se na kraju obavlja inicijalizacija

programa za izračune opće namjene i potrebnih sredstava. Od sredstava se inicijaliziraju SRV i UAV međuspremnici. Budući da se UAV međuspremnici koriste samo za pohranjivanje rezultata programa za izračune opće namjene, dovoljno ih je samo jednom inicijalizirati, dok je SRV međuspremnike potrebno inicijalizirati svaki put prilikom slanja novih podataka na grafičko sklopovlje.

Nakon svih potrebnih inicijalizacija može se krenuti sa računanjem i prvi korak u tome je određivanje dobrote svih jedinki u populaciji. Ovo se radi samo jednom, jer kasnije je dovoljno izračunati samo dobrote novih jedinki u populaciji, a ne svih. Sljedeći korak je 3-turnirska selekcija i križanje, stoga je potrebno odabrati i poslati podatke na grafičko sklopovlje. No umjesto da centralni procesor slučajnim odabirom izvlači jedinke iz populacije i sprema ih za selekciju i križanje, u ovom koraku jednostavno se uzme prvih *n* članova populacije i pošalje se grafičkom sklopovlju. To se izvede tako da se prilikom inicijalizacije SRV međuspremnika, funkciji za inicijalizaciju preda pokazivač na prvi element populacije i broj elemenata koji je potreban za selekciju i križanje. Tada će se u taj međuspremnik spremiti samo prvih *n* elemenata populacije koji će biti dostupni programu za izračune opće namjene. Nakon što program za izračune opće namjene izvrši selekciju i križanje, dobvaju se nove jedinke koje su nastale u tom procesu. Budući da se nove jedinke nakon dohvaćanja podataka sa GPU nalaze na istim indeksima kao i jedinke koje su izgubile 3-turnirsku selekciju, vrlo se lako zamjene sa starim jedinkama u populaciji i pri tome im se izračuna dobrota, tj. kazna.

Nakon što je obavljena selekcija, križanje i određivanje dobrote, kada bi se krenulo u novu iteraciju opet bi se uzelo onih *n* istih jedinki koje su korištene u prijašnjoj iteraciji. To bi rezultiralo time da bi grupe za selekciju i križanje bile slične onima koje su ranije korištene. Razlika bi bila u novim jedinkama, ali jedna grupa sadržavala bi iste dvije jedinke koje su bile korištene u prijašnoj iteraciji. Kako bi se to izbjeglo iskoristi se program za izračune opće namjene koji generira pseudoslučajne brojeve. Pomoću njega dobija se *m* slučajnih brojeva, a taj *m* je jednak polovici veličine populacije i generirani brojevi su u rasponu od nula do veličine populacije. Ti pseudoslučajni brojevi koriste se za miješanje jedinki u populaciji, tj. da za zamjenu indeksa različitih jedinki. Na taj način

osigurava se odabir različitih grupa jedinki u svakoj novoj iteraciji programa. Pseudokod programa prikazan je na slici 18.

```
GPU_Program()
{
   // Inicijaliziramo populaciju, N je ukupni broj jedinki ...
   Stablo populacija[ N ];
   Inicijaliziraj_populaciju( *populacija );

   // Inicijaliziramo sučelje prema sklopovlju ...
   (Device, Context) = Inicijaliziraj_sučelje_prema_GPU();

   // Napravimo procesor za izračune opće namjene (compute shader)
...
   ComputeShader = Prevedi_i_inicijaliziraj( ime_datoteke.hlsl );

   // Inicijalizacija sredstava, M je broj jedinki za
   // 3-turnisku selekciju ...
   SRVspremnik = Napravi_SRV_međuspremnik( *populacija, M );

   // Spremnik za dohvaćanje rezultata procesor za izračune opće
namjene (compute shader)a ...
   UAVspremnik = Napravi_UAV_međuspremnik( NULL, M );

   // Početni izračun dobrote ...
   Izračunaj_dobrotu_cijele_populacije();

   ponavljaj_dok( uvijet_za_kraj_ponavljanja != istina )
   {
      // Jedinke za selekciju i križanje, uzimamo M jedinki ...
      SRVspremnik = Napravi_SRV_međuspremnik( *populacija, M );
      // Pokrenemo procesor za izračune opće namjene (compute
shader) ...
      PokreniComputeShader( SRVspremnik, UAVspremnik, M/3, 1, 1);

      // Dohvtimo podatke i zapisemo nove jedinke u populaciju...
      rezultati = Dohvati_rezultate( UAVspremnik );
      Smjesti_nove_jedinke_u_populaciju( rezultati, populacija );
      Izračunaj_dobrotu_samo_novih_jedinki( populacija );

      // Generiramo random brojeve, N/2 random brojeva ...
      Generiraj_slučajni_broj_i_pošalji_na_GPU();
      PokreniComputeShader( NULL, UAVspremnik, N/2, 1, 1 );

      // Dohvatimo generirane pseudoslučajne brojeve i na temelju
```

```
        // njih zamijenimo indeske jedinki u populaciji ...
        rezultati = Dohvati_rezultate( UAVspremnik );
        Izmiješaj_populaciju( rezultati, populacija );

    }

    // Kraj ...
    Izlaz();
}
```

Slika 19 - Pseudokd GPU implementacije GP programa (veći stupanj apstrakcije od primjera na slici 13).

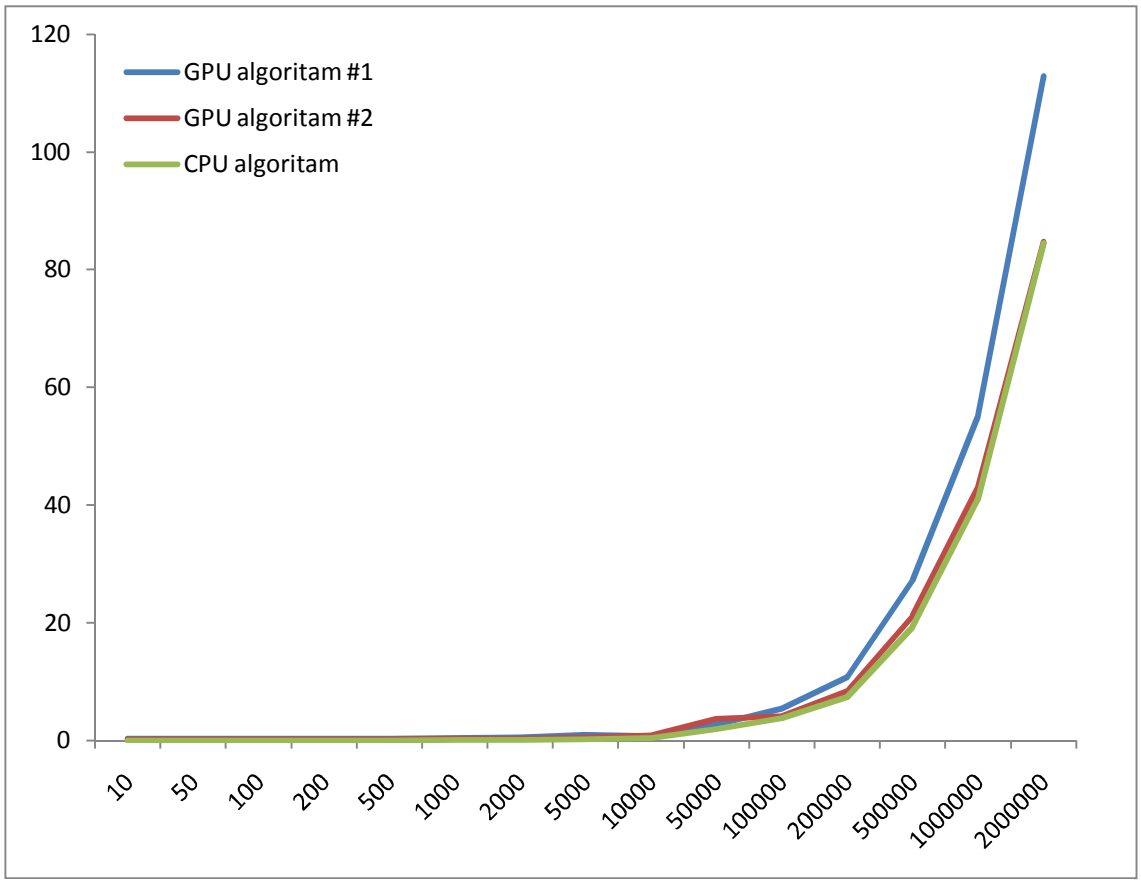
7. USPOREDBA GENETSKIH PROGRAMA NA CPU I GPU

Kako bi dobili informaciju o iskoristivosti GPU implementacije genetskog programa, potrebno je taj program usporediti sa klasičnom CPU implementacijom. Pitanje na koje ovdje moramo dobiti odgovor je da li se paralelizacijom genetskog programa pomoću GPU platforme može dobiti manje prosječno trajanje izvođenja u odnosu na klasično ostvarenje gentskog programa na centralnom procesoru.

Kako bi dobili odgovor na to pitanje provedeno je ispitivanje na tri genetska programa. Prvi program je klasična CPU implementacija, dok su druga dva programa GPU implementacije. Prva GPU implementacija pomoću procesor za izračune opće namjene (compute shader)a obavlja selekciju, križanje i izračun dobrote nove jedinke, dok druga GPU implementacija izračun dobrote nove jedinke obavlja na centralnom procesoru, a selekciju i križanje pomoću procesor za izračune opće namjene (compute shader)a. Sustav na kojem smo napravili ispitivanje sadržavao je AMD Phenom II x2 centralni procesor sa dvije jezgre koje rade na frekvenciji od 3.1 GHz i grafičku karticu ATI HD5870 sa 1600 stream procesora i frekvencijom rada od 850MHz.

7.1. BRZINE RADA PROGRAMA

Budući da nas zanima kako se međusobno vremenski odnose različiti programi, uvijet za zaustavljanje svakoga programa bio je određeni broj iteracija. Broj iteracija je konstantan i u ovom slučaju iznosi 100 iteracija programa. Ono što se u programima mijenja je ukupni broj jedinki koji se nalazi u populaciji, a sa povećanjem jedinki povećava se i ukupni utrošak vremena potrebnog za njihovu obradu i upravo je to vrijeme koje se paralelizacijom pokušava smanjiti. Tijekom ispitivanja broj jedinki u populaciji mijenjao se od 10 do $2 \cdot 10^6$, a rezultati su vidljivi na slici 19 gdje *x* os predstavlja broj jedinki, a *y* os je vrijeme u sekundama.



Slika 20 - Usporedni prizak vremena izvođenja dvije GPU implementacije programa i jedne CPU implementacije.

Sa slike je vidljivo da su za male veličine populacije sva tri programa prilično brza, međutim CPU implementacija je zapravo puno brža nego ostala dva algoritma što se može vidjeti u tablici 2. Kako se broj jedinki populacije povećava tako se povećava i vrijeme izvođenja programa. U ovo slučaju najbrže raste vrijeme izvođenja prve implementacije GPU programa koja selekciju, križanje i izračun dobrote obavlja pomoću programa za izračune opće namjene. Razlog tome je prevelika složenost programa za proračune opće namjene u smislu kontrole toka programa, koja degradira ukupne performanse zbog čega je ovaj program najsporiji od ispitanih.

CPU implementacija je u prednosti nad ostala dva programa sve do veličine populacije između 10^6 jedinki i $2 \cdot 10^6$, kada druga implementacija programa na GPU po performansama postaje približno jednaka sa CPU implementacijom.

Ukoliko se usporede prva i druga GPU implementacija programa može se primjetiti da smanjenje složenosti programa za izračune opće namjene i pametna raspodjela posla između centralnog procesora i grafičkog sklopovlja može znatno ubrzati izvođenje programa koji su djelomično implementirani na GPU. Razlika između ova dva programa je u implementaciji procesa određivanja dobrote koji je kod prvog GPU programa implementiran na grafičkom procesoru, a kod drugog GPU programa na centralnom procesoru. Ovdje također treba uzeti u obzir i svojstva sustava na kojem se proračuni izvode.

Ukoliko se usporede brzine rada grafičkog procesora (850 MHz) i centralnog procesora (3100 MHz) u sustavu na kojem se ispitivanje provelo, vidi se da je grafički procesor sporiji od centralnog procesora približno za faktor 3.65, no uz pomoć svoje paralelne arhitekture daje performanse koje se mogu usporediti sa puno bržim centralnim procesorom.

Tablica 2 - Vremena izvođenja programa za različite veličine populacije.

Populacija	GPU algoritam #1 [s]	GPU algoritam #2 [s]	CPU algoritam [s]
10	0,303163	0,200273	0,000541
50	0,314173	0,195091	0,001249
100	0,310728	0,199981	0,004081
200	0,297858	0,234805	0,007864
500	0,323414	0,209128	0,021657
1000	0,414531	0,251422	0,037542
2000	0,525444	0,319175	0,064448
5000	0,958863	0,531981	0,205101
10000	0,725489	0,890736	0,381621
50000	2,732907	3,679662	1,884712
100000	5,401108	4,149066	3,778232
200000	10,760893	8,319077	7,378326
500000	27,130475	21,12277	19,283647
1000000	55,033449	42,803501	40,845967
2000000	112,92807	84,712799	84,52925

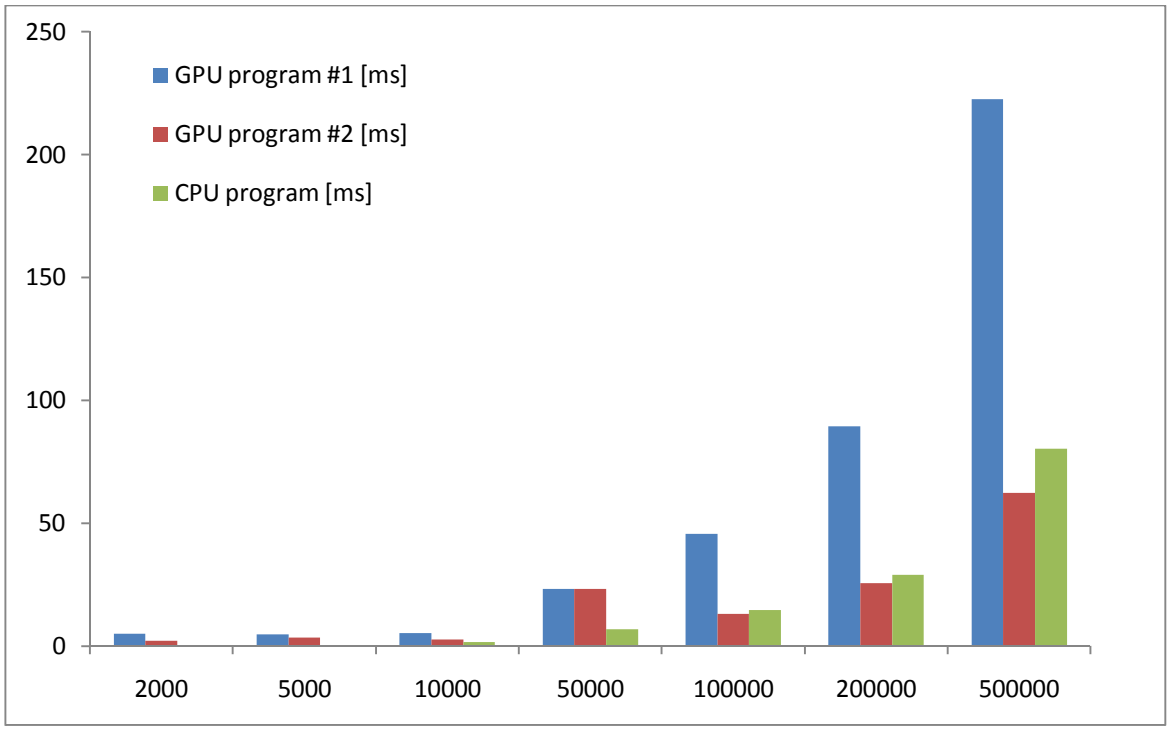
7.2. USPOREDBA STRUKTURE PROGRAMA

Ukoliko se pogleda struktura samih programa, može se primjetiti da dijele puno zajedničih elemenata, što proizlazi iz činjenice da se svi programi temelje na osnovama genetskog programiranja. Prvi dio strukture svih programa je selekcija i križanje jedinki. CPU implementacija selekciju i križanje ima implementiranu u nekoliko stupnjeva. Prvo se odaberu jedinke populacije koje će sudjelovati u turniru. Budući da su za to potrebni samo indeksi jedinki i njihove dobrote, nema potrebe odabrane jedinke kopirati na drugu memorijsku lokaciju jer bi to vremenski bilo zahtjevno. Na temelju indeksa i dobrote jedinki u 3-turnirskoj selekciji odabiru se jedinke za križanje. Potom se odabrane jedinke kopiraju i međusobno križaju, a ovako se može manipulirati jedinkama bez promjene orginala. Kada je križanje završeno, nove jedinke se stavljaju u populaciju na indekse jedinki koje su izgubile turnir i pri tome im se računa dobrota. GPU implementacija taj postupak implementira na grafičkom sklopovlju, pa iz dijela programa koji se odvija na centralnom procesoru ovaj postupak se odvija tako da se prvo pošalju podatci na grafičko sklopovlje, a kada program za izračune opće namjene završi proračune podatci se dohvate i nove jedinke se kopiraju u populaciju.

Na slici 20 vidi se grafički prikaz vremena potrebnih za obavljanje slekcije, turnira i križanja, za sva tri programa i veličine populacije između 2000 jedinki i $5 \cdot 10^5$ jedinki. U tablici 3 nalazi se brojčani prikaz tih podataka. Važno je još napomenuti da kod prvog GPU programa u navedena vremena ulazi i vrijeme potrebno za računanje dobrote novih jedinki, budući da je to implementirano zajedno sa križanjem i rekombinacijom u programu za izračune opće namjene.

Tablica 3 - Vremena selekcije jedinki, turnira i križanja za sva tri programa. Za GPU program #1 u ova vremena uračunato je i vrijeme izračuna dorbote novih jedinki.

Populacija	GPU program #1 [ms]	GPU program #2 [ms]	CPU program [ms]
2000	5,00151	2,15155	0,223178
5000	4,849177	3,655197	0,536315
10000	5,368795	2,787624	1,597652
50000	23,398714	23,386718	7,0206229
100000	45,737865	13,311716	14,675199
200000	89,618103	25,621694	29,191735
500000	222,606975	62,541423	80,328701



Slika 21 - Vremena izvođenja turnirske selekcije i križanja za sva tri testna programa za različite veličine populacije. GPU program #1 uključuje i vrijeme potrebno za određivanje dobrote novih jedinki.

Sa slike 20 jasno se može vidjeti zašto prva implementacija GPU programa ima velika vremena izvođenja u odnosu na druga dva programa, a isto tako može se vidjeti zašto druga implementacija GPU programa u jednom trenutku dostigne performanse jednake preformansama CPU programa.

Zbog velike složenosti programa za izračune opće namjene koji obavlja turnirsku selekciju, križanje i izračun dobrote, prva implementacija GPU programa u najgorem slučaju ima za nekoliko puta veće vrijeme izvođenja programa za izračune opće namjene u odnosu na drugu implementaciju koja je implementirala samo turnirsku selekciju i križanje. Iz ovoga možemo zaključiti da je upravo program za izračune opće namjene usko grlo zbog kojega su ukupna vremena izvođenja prvog GPU programa lošija u odnosu na druga dva programa.

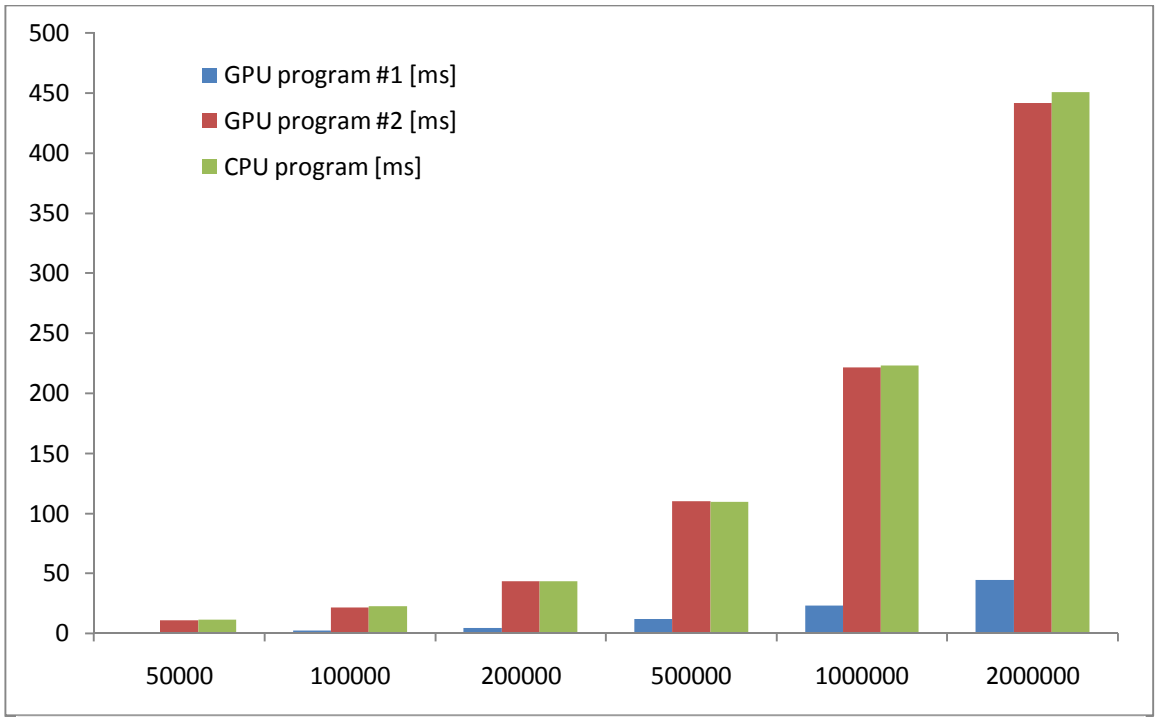
Druga implementacija GPU programa pokazuje lagani rast vremena koje je potrebno da se podatci pošalju na grafičko sklopovlje, da ih program za izračune opće obradi, te da se podatci dohvate natrag u glavni dio programa, no taj rast je puno sporiji od ekvivalentog skupa operacija u CPU programu. Kako je taj dio programa druge GPU imlementacije sve brži u odnosu na CPU tako će se i razlika između vremena izvođenja ta dva programa smanjivati, a to je vidljivo na slici 19. No zbog velikog broja registara koji su potrebni

svakoj dretvi za pohranu podataka svake jedinke, ukupni broj dretvi koji je moguće pokrenuti na grafičkom procesoru je ograničen. Stoga ispitivanje za populacije veće od pet milijuna jedinki nije bilo moguće izvršiti pa se ne može sa sigurnošću tvrditi da bi druga GPU implementacija u jednom trenutku prestigla CPU implementaciju. Ovaj problem mogao bi se rješiti dodavanjem dodatnog grafičkog sklopovlja.

Ono što se još može usporediti između programa je vrijeme potrebno za kopiranje podataka natrag u populaciju i određivanje dobrote novih jedinki. Budući da se kod prvog GPU programa određivanje dobrote novih jedinki implementira u prgramu za izračune opće namjene, vremena prikazana na slici 21 i u tablici 4 za taj program, odnose se samo na vrijeme potrebno za kopiranje novih jedinki u populaciju. Za druga dva programa vremena uključuju kopiranje jedinki i određivanje dobrote tih jedinki.

Tablica 4 - Ukupno vrijeme za dodavanje novih jedinki u populaciju i izračun njihove dobrote za GPU program #2 i CPU program. ZA GPU program #2 vremena se samo odnose na dodavanje novih jedinki u populaciju.

Populacija	GPU program #1 [ms]	GPU program #2 [ms]	CPU program [ms]
50000	0,940946	10,767026	11,485937
100000	2,262973	21,752254	22,512584
200000	4,363867	43,349873	43,416146
500000	11,872314	110,062948	109,700492
1000000	23,4207	221,330136	223,341855
2000000	44,508453	441,5503707	450,655788



Slika 22 – Grafički prikaz vremena potrebnog za kopiranje novih jedinki u populaciju i izračun njihove dobrote za GPU program #2 i CPU program, te vremena kopiranja novih jedinki u populaciju za GPU #1 program (grafički prikaz podataka iz tabele 4).

Treba primijetiti da se na slici 20 i na slici 21 razlikuje raspon veličina populacija i ukoliko se žele usporediti grafovi na tim slikama, mogu se uspoređivati samo stupci koji se odnose na iste veličine populacije. Na slici 21 je vidljivo da su vremena, potrebna za kopiranje novih jedinki u populaciju i određivanje njihove dobrote, kod drugog GPU programa i CPU programa gotovo identična, što je i za očekivati budući da se radi o jednakoj implementaciji. No isto tako primjećuje se da je prvi GPU program višestruko brži, budući da ne obavlja izračun dobrote, već samo kopira jedinke u populaciju. To dodatno naglašava koliko je proces određivanja dobrote zahtjevan i za grafički procesor i za centralni procesor. Ipak ta razlika nije toliko velika da bi poništila zaostatak prvog GPU programa koji nastaje kod pokretanja njegovog programa za izračune opće namjene, što je najbolje vidljivo ako se usporede retci u tablicama 3 i 4 za veličinu populacije od 500 000 jedinki. Ukoliko napravimo tu usporedbu dobivamo da prva GPU implementacija kasni za drugom GPU implementacijom otprilike 70 milisekundi po iteraciji, dok za CPU implementacijom kasni otprilike 55 milisekundi po iteraciji.

Ono što koncepcijski razlikuje ova dva GPU programa od CPU programa odnosi se na

završni korak u kojem se u populaciji slučajnim odabirom mijenaju mjesta jedinki. Ovaj korak u CPU implementaciji ne postoji, no u GPU implementacijama je nužan zato jer simulira slučajni odabir jedinki za 3-turnirsku selekciju. Posljedica ovog koraka je što poništava eventualnu prednost koju smo postigli prethodnom paralelizacijom.

8. ZAKLJUČAK

Procesor za izračune opće namjene i grafičko sklopovlje pružaju platformu koja je sposobna za veliki stupanj paralelizacije, no konačni rezultati paralelizacije programa uvelike ovise i o strukturi podataka u tom programu i njegovoj koncepciji. Glavna stvar na koju treba paziti prilikom paralelizacije programa pomoću sučelja DirectX je da programi koji se pišu za grafički procesor budu što jednostavniji, čime se izbjegavaju moguće pogreške i omogućuje njihovo brže izvođenje. U ispitivanjima je također vidljivo kako se velikim smanjenjem složenosti programa za izračune opće namjene mogu postići značajna ubrzanja, te da je vrlo važno dobro raspodijeliti proračune između grafičkog sklopovlja i centralnog procesora. Objektivno gledajući, pomoću procesor za izračune opće namjene možemo napraviti puno proračuna, no treba uzeti u obzir da prečesto prebacivanje podataka između grafičkog sklopovlja i radne memorije može uzeti puno vremena i uzrokovati pad performasi programa.

Također, iz ispitivanja je vidljivo da je struktura podataka u genetskim programima prilično kompleksna za potpunu obradu pomoću programa za izračune opće namjene. Iako se podatci u tim strukturama obrađuju paralelno na grafičkom sklopovlju, prvi dobitci u performansama mogu se primjetiti tek za vrlo velike populacije. Ukoliko pogledamo vremena za selekciju i križanje u tablici 3, vidjet ćemo da se prvi dobitci na performansama dobivaju kada se u populaciji nalazi oko 10^5 jedinki, što je često puno više nego je potrebno.

Odgovor na pitanje o iskoristivosti programskog sučelja DirectX i programa za izračune opće namjene u genetskom programiranju je dvojak. Budući da prava snaga grafičkog sklopovlja leži u količini podataka koja se odjednom može obraditi, procesor za izračune opće namjene možemo koristiti za rješavanje GP problema koji generiraju jako velike skupove mogućih rješenja, pod uvijetom da se strukture podataka optimiraju za izvođenje na grafičkom sklopovlju. Za manje probleme, paralelizacija na grafičkom sklopovlju vjerojatno neće dovesti do poboljšanja performansi, kako smo vidjeli u napravljenim ispitivanjima.

Iako programi za izračune opće namjene možda nisu dali željene rezultate u ispitivanjima, moramo uzeti u obzir da je ovo nova tehnologija koja će se nastaviti razvijati i u bližoj budućnosti donijeti značajne napretke u postupcima pisanja programa opće namjene za grafički procesor. HLSL jezik, koji nam služi za programiranje programa za izračune opće namjene (*compute shader*), također se konstantno razvija. Budući da je ovo tek početak ove tehnologije, možemo očekivati da će HLSL jezik u budućim verzijama donijeti nove mogućnosti koje će omogućiti fleksibilnije pisanje programa za izračune opće namjene.

Programi za proračune opće namjene (*compute shader*) predstavljaju prvi korak prema jednom sučelju visoke razine za razvijanje programa opće namjene za grafičke procesore. Budući da već sada postoje mnoge uspješne implementacije, posebice fizikalnih simulacija, možemo očekivati da će sa razvojem DirectX tehnologije i programa za proračune opće namjene samo dobivati na performansama i širiti područje primjene, od kojih će jedno nesumnjivo biti i genetsko programiranje.

9. LITERATURA

[1] Parallel computing, 13.05.2010., *Parallel computing*, http://en.wikipedia.org/wiki/Parallel_computing, 20.05.2010.

[2] Mooreov zakon, 19.05.2010., *Moore's Law*, http://en.wikipedia.org/wiki/Moore%27s_Law, 23.05.2010.

[3] Amdahlov zakon, 14.05.2010., *Amdahl's law*, http://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&action=history, 23.05.2010.

[4] Gustafsonov zakon, 23.03.2010., *Gustafson's law*, http://en.wikipedia.org/wiki/Gustafson%27s_law, 23.05.2010.

[5] Opasnost utrke, 12.05.2010., *Race condition*, http://en.wikipedia.org/wiki/Race_condition, 25.05.2010.

[6] Atomic operation, 11.02.2010., *Atomic operation*, http://en.wikipedia.org/wiki/Atomic_operation, 25.05.2010.

[7] Bit-level parallelism, 28.11.2009., *Bit-level parallelism*, http://en.wikipedia.org/wiki/Bit-level_parallelism, 25.05.2010.

[8] Data parallelism, 13.01.2010., *Data parallelism*, http://en.wikipedia.org/wiki/Data_parallelism, 26.05.2010.

[9] Task parallelism, 27.05.2010., *Task parallelism*, http://en.wikipedia.org/wiki/Task_parallelism, 27.05.2010.

[10] Symmetric multiprocessing, 22.05.2009., *Symmetric multiprocessing*, http://en.wikipedia.org/wiki/Symmetric_multiprocessing, 28.05.2010.

[11] GPGPU Programming Concepts, *GPGPU: Encyclopedia II - GPGPU - GPGPU Programming Concepts,* http://www.experiencefestival.com/a/GPGPU_-_GPGPU_Programming_Concepts/id/5071506, 22.05.2010.

[12] Some thoughts on the procesor za izračune opće namjene (compute shader) and the future GPU architecture, 04.09.2009., *Some thoughts on the procesor za izračune opće namjene (compute shader) and the future GPU architecture*, http://www.opengpu.org/bbs/archiver/?tid-835.html, 05.04.2010.

[13] GPGPU, 10.05.2010., *General-purpose computing on graphics processing units*, http://en.wikipedia.org/wiki/GPGPU, 15.05.2010.

[14] Matt Pharr, GPU Gems 2, *GPU Gems 2 – Programming techniques for high-performance graphics*, http://developer.nvidia.com/object/gpu_gems_2_home.html, 20.04.2010.

[15] Vertex shader, 15.02.2010., *Vertex shader*, http://en.wikipedia.org/wiki/Vertex_shader, 16.05.2010.

[16] Vertex Shaders, *Vertex Shaders*, http://www.nvidia.com/object/feature_vertexshader.html, 16.05.2010.

[17] DirectX, 20.05.2010., *DirectX*, http://en.wikipedia.org/wiki/DirectX, 25.05.2010.

[18] Windows Display Driver Model, 12.05.2010., *WDDM:Windows Display Driver Model*, http://en.wikipedia.org/wiki/Windows_Display_Driver_Model, 26.05.2010.

[19] Microsoft Direct3D, 18.05.2010., *Microsoft Direct3D*, http://en.wikipedia.org/wiki/Direct3D#Architecture, 27.05.2010.

[20] Graphics Device Interface, 19.05.2010., *Graphics Device Interface*, http://en.wikipedia.org/wiki/Windows_GDI, 27.05.2010.

[21] High Level Shader Language, 07.05.2010., *High Level Shader Language*, http://en.wikipedia.org/wiki/High_Level_Shader_Language, 28.05.2010.

[22] HLSL, 05.04.2010., *HLSL*, http://msdn.microsoft.com/en-us/library/bb509561%28VS.85%29.aspx, 28.05.2010.

[23] Variables (DirectX HSLS), 05.04.2010., *Variables (DirectX HLSL)*, http://msdn.microsoft.com/en-us/library/bb509705%28v=VS.85%29.aspx, 29.05.2010.

[24] For statement (DirectX HLSL), 05.04.2010., *For statement (DirectX HLSL), http://msdn.microsoft.com/en-us/library/bb509610%28v=VS.85%29.aspx, 30.05.2010.*

[25] If statement (DirectX HLSL), 05.04.2010., *If statement (DirectX HLSL)*, http://msdn.microsoft.com/en-us/library/bb509610%28v=VS.85%29.aspx, 30.05.2010.

[26] Semantic (DirectX HLSL), 05.04.2010., *Semantic (DirectX HLSL)*, http://msdn.microsoft.com/en-us/library/bb509647%28v=VS.85%29.aspx, 31.05.2010.

[27] GPU Flow-Control Idioms, 04.2005., *Chapter 34: GPU Flow-Control Idioms*, http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter34.html, 01.06.2010.

[28] ID3D11DeviceContext::Dispatch Method, 05.04.2010., *ID3D11DeviceContext::Dispatch Method*, http://msdn.microsoft.com/en-us/library/ff476405(VS.85).aspx, 01.06.2010.

[29] DirectX 11 Procesor za izračune opće namjene (compute shader) tutorial, 26.11.2009., *DirectX 11 Procesor za izračune opće namjene (compute shader) tutorial*, http://www.gamedev.net/community/forums/topic.asp?topic_id=516043, 01.06.2010.

[30] Poli R., Langdon W.B., McPhee N.F., A Field Guide To Genetic Programming, 03.2008., *A Field Guide To Genetic Programming*, http://www.lulu.com/items/volume_63/2167000/2167025/2/print/book.pdf, 02.04.2010.

[31] Genetic Programming, 21.05.2010., *Gentic Programming*, http://en.wikipedia.org/wiki/Genetic_programming, 02.04.2010.

[32] Linear Congurential Generator, 23.04.2010., *Linear Congruential Generator*, http://en.wikipedia.org/wiki/Linear_congruential_generator, 15.05.2010.

[33] Thomas Fuller, Increase in Knowledge Workers, 28.07.2007., *Does Human Knowledge Double Every 5 Years? - Increase in Knowledge Workers*, http://newsfan.typepad.co.uk/does_human_knowledge_doub/increase_in_knowledge_workers/, 08.06.2010.

10. SAŽETAK

Računala u današnje doba služe za rješavanje različitih problema iz različitih domena primjene, a mnogi od tih problema mogu se riješiti pomoću evolucijskih algoritama. Iako evolucijski algoritmi pružaju sredstvo za rješavanje problema, često su ti problemi veoma složeni, pa se njihovo rješavanje pokušava ubrzati paralelizacijom. Paralelizacija programa pomoću grafičkog sklopovlja relativno je nova metoda koja se pokušava primjeniti na evolucijske algoritme.

Paralelno programiranje je postupak u kojem se jedan skup podataka ili jedan skup naredbi obrađuje pomoću više procesnih elemenata (npr. različite jezgre kod višejezgrenih procesora). Općenito programi se mogu podijeliti na manje dijelove koji međusobno nisu zavisni i mogu se uspješno paralelizirati. Međutim, različiti programi nemaju isti stupanj moguće paralelizacije. Dobitke u performansama koji se postižu paralelizacijom nekog programa opisuje Amdahlov zakon, koji kaže da ukupno ubrzanje ovisi o dijelu programa koji se ne može paralelizirati.

Pisanje programa opće namjene za grafičko sklopovlje (GPGPU) relativno je nova metoda programiranja koja pokušava iskoristiti visoku razinu paralelnosti grafičkog sklopovlja. Za pisanje takvih programa potrebno je sučelje koje će omogućiti pristup grafičkom sklopovlju. Microsoftov DirectX API je upravo takvo sučelje koje nam omogućava pristup grafičkom sklopovlju i u svojoj zadnjoj verziji (DirectX 11) donosi podršku za pisanje programa opće namjene za grafički procesor u obliku programa za izračune opće namjene *(compute shader).* Program za izračune opće namjene je mali program koji se izvodi na grafičkom procesoru i pruža mogućnost obrade različitih tipova podataka i obavljanja proračuna različitih namjena.

Genetsko programiranje spada u područje evolucijskih algoritama. Cilj genetskog programiranja je postupcima selekcije, križanja, određivanja dobrote i mutacije evoluirati početnu populaciju potencijalnih rješenja nekog problema u novu populaciju koja bi mogla sadržavati konačno rješenje. Ovi postupci često znaju biti vremenski

zahtjevni, zbog toga ih pokušavamo paralelizirati za izvođenje na grafičkom procesoru, pri čemu koristimo DirectX i programe za izračune opće namjene kao programsko sučelje prema grafičkom procesoru. Budući da programi za izračune opće namjene moraju biti što jednostavniji da se osigura njihovo optimalno izvođenje, a podatkovne strukture u genetskom programiranju mogu biti prilično kompleksne, vrlo je važno da se odredi koje se dijelove genetskog programa isplati paralelizirati a koje ne.

Konačni rezultati pokazali su da je bila najefikasnija paralelizacija selekcije i križanja jedinki populacije. Kada se doda izračun dobrote novih jedinki u program za izračune opće namjene, on postaje veoma složen što dovodi do degradiranja performansi programa. Slijedna implementacija programa, napisana isključivo za centralni procesor, ima približno jednake performanse kao i paralelni program koji je implementirao selekciju i križanje na grafičkom procesoru. Iako je paralelni dio programa, za veće populacije jedinki, bio brži od ekvivalentnog dijela u slijednom programu, gubitak vremena u drugim dijelovima paralelnog programa zbog različitog koncepta izjednačio je njihove performanse.

Programi za izračune opće namjene nova su tehnologija koja će još puno razvijati u smislu mogućnosti i u smislu performansi, te koja predstavlja veliki korak prema jednom sučelju visoke razine za programiranje programa opće namjene za grafičke procesore.