|  |  |  |
| --- | --- | --- |
|  | UNIVERZITET U NOVOM SADU  **FAKULTET TEHNIČKIH NAUKA** |  |

**Analiza paralelizacije algoritama za iscrtavanje fraktala Mandelbrot i Julia tipa i uticaj preciznosti izračunavanja na performanse iscrtavanja**

Autor: Uroš Jakovljević Profesor: dr Veljko Petrović

Sadržaj

[**1.** **UVOD** 3](#_Toc33291134)

[**2.** **ŽULIJIN I MANDELBROTOV SKUP** 4](#_Toc33291135)

[2.1. Žulijin skup 4](#_Toc33291136)

[2.2. Mandelbrotov skup 4](#_Toc33291137)

[**3.** **ALGORITMI ZA ISCRTAVANJE ŽULIJINOG I MANDELBROTOVOG SKUPA** 6](#_Toc33291138)

[3.1. *Escape time* algoritam za iscrtavanje 6](#_Toc33291139)

[3.2. Kontinualni algoritmi za iscrtavanje 8](#_Toc33291140)

[**4.** **Poboljšanje performansi iscrtavanja i paralelizacija algoritama za iscrtavanje Žulijinog i Mandelbrotovog skupa** 9](#_Toc33291141)

[4.1. Sekvencijalno iscrtavanje i optimizacija sekvencijalnog iscrtavanja 9](#_Toc33291142)

[4.2. Paralelizacija algoritama za iscrtavanje fraktala 12](#_Toc33291143)

[4.3. Paralelizacija algoritama za iscrtavanje fraktala na grafičkim karticama 13](#_Toc33291144)

[4.4. Uticaj preciznosi izračunavanja na paralelizaciju i performanse iscrtavanja fraktala 15](#_Toc33291145)

[**5.** **Zaključak** 20](#_Toc33291146)

[**LITERATURA** 21](#_Toc33291147)

# **UVOD**

Fraktal je geometrijski lik koji se može razložiti na manje delove tako da je svaki od njih, makar približno, umanjena kopija celine, odnosno da delovi fraktala podsećaju na ceo fraktal (samosličnost). Termin je izveo Benoa Mandelbrot 1975. godine. Pošto se čine sličnim na svim nivoima uvećanja, fraktali se često smatraju beskonačno kompleksnim u neformalnom smislu reči [1].

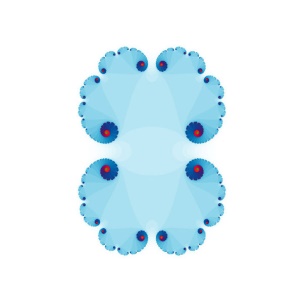
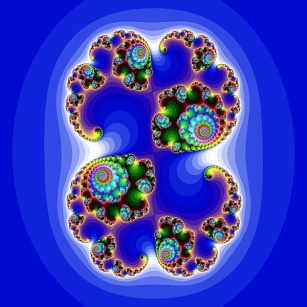
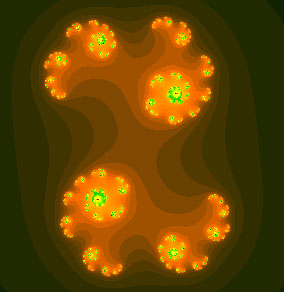
Postoje prirodni oblici koji aproksimiraju fraktale do izvesne granice i to su na primer oblaci, munje, snežne pahuljice. Međutim, najčuveniji primer fraktala jesu Žulijin i Mandelbrotov skup (*eng. Julia and Mandelbrot set*). Oni spadaju u algebarske fraktale, što znači da se za njihovu konstrukciju koriste iterativne nelinearne funkcije koje se zadaju jednostavnim algebarskim formulama [1].

U ovom radu će biti opisani algoritmi za konstruisanje, odnosno iscrtavanje ovih fraktala na računaru. Takođe će biti razmatrane mogućnosti paralelizacije ovih algoritama, radi poboljšanja performansi izračunavanja i iscrtavanja fraktala. Poseban akcenat će biti na paralelizaciji algoritama i iscrtavanju na grafičkim karticama. Takođe će biti analiziran uticaj preciznosti izračunavanja na iscrtavanje i performanse iscrtavanja fraktala.

# **ŽULIJIN I MANDELBROTOV SKUP**

## Žulijin skup

Ako se za svaku tačku kompleksne ravni **z0 = x + *i*y** definiše niz **zn+1 = *f*(zn)**, gde ***f*(zn)** može biti bilo koja funkcija, možemo definisati dva skupa: skup tačaka **z0** za koje taj niz konvergira i skup tačaka **z0** za koje taj niz divergira, odnosno teži u beskonačnost. Žulijin skup (u širem smislu) je granica tih skupova. Žulijin skup (u užem smislu) dobijamo ako za funkciju ***f*(zn)** izaberemo ***f*(zn) = zn2 + c**. Obično se ovaj skup kao i svi algebarski fraktali prikazuje tako da su tačke koje konvergiraju crne, a one koje divergiraju u raznim nijansama iste ili različitih boja. Nijansa boje zavisi od brzine kojom niz raste – što se više odmičemo od Žulijinog skupa, niz brže raste. Menjanjem konstante **c** u Žulijinom skupu u užem smislu, dobijamo najrazličitije skupove (slika 2.1) [2].

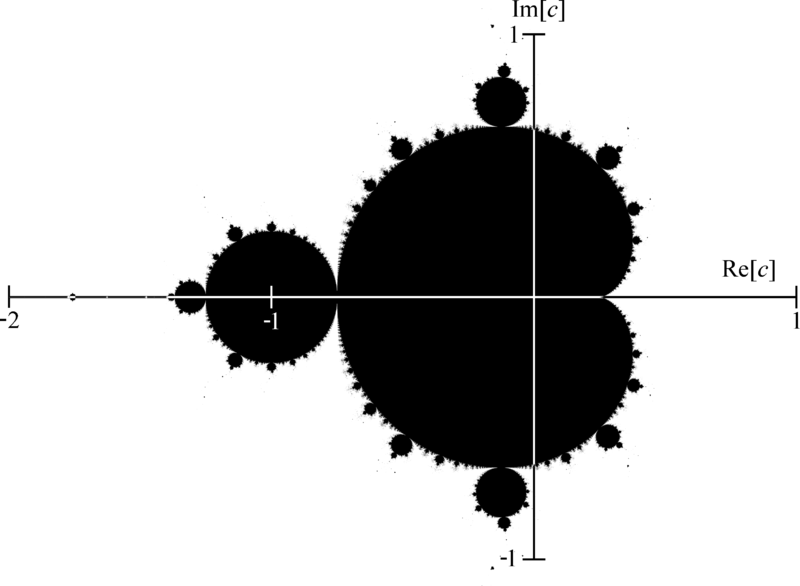
  

*Slika 2.1 Žulijin skup za vrednosti c=0.285+0i, c=0.285+0.01i i c=0.45 + 0.1428i* [3]

## Mandelbrotov skup

Mandelbrotov skup je određen rekurentnom funkcijom **zn+1 = zn2 + c** gde je **c** kompleksan broj takav da je Žulijin skup povezan. Žulijin skup je povezan ako je skup koji okružuje kompaktan. Skup je kompaktan ako je ograničen i zatvoren. Tačnije, Mandelbrotov skup je skup svih kompleksnih brojeva **c** takvih da je, za početni uslov **z0 = 0**, moduo kompleksnog broja **zn+1 = zn2 + c** ograničen. Sve tačke Mandelbrotovog skupa **M** leže unutar kruga poluprečnika 2 (slika 2.2) [2].

Razni samoslični fraktali, kojima pripada i Mandelbrot, najjednostavnije se konstruišu uz pomoć *Escape time* algoritma o kome će biti reči u narednim poglavljima.



*Slika 2.2 Mandelbrotov skup* [3]

# **ALGORITMI ZA ISCRTAVANJE ŽULIJINOG I MANDELBROTOVOG SKUPA**

Kako su skupovi koji su do sada opisani sačinjeni od beskonačno tačaka, a čovek i računari mogu izvršiti samo konačno mnogo operacija, potrebno je izabrati konačno mnogo tačaka kompleksne ravni da bi se ti skupovi nacrtali. Osnovna ideja svih algoritama za crtanje Žulijinog i Mandelbrotovog skupa je ta da se odabirom što većeg broja tačaka greška prilikom crtanja navedenih skupova smanjuje. Za crtanje Žulijinog i Mandelbrotovog skupa, na slici dimenzija **w ⋅ h**, potrebno je prvo izabrati pravougaonu oblast u kompleksnoj ravni čije se stranice odnose u razmeri **w : h**, gde su **w** i **h** prirodni brojevi. Zatim je potrebno tu oblast izdeliti na **w ⋅ h** manjih podudarnih kvadrata (piksela), i iz svakog kvadrata je potrebno izabrati po jednu tačku (na primer gornje levo teme kvadrata, ili sam centar kvadrata). Za fiksirane dimenzije velikog pravougaonika, uvećavanjem rezolucije slike, površina kvadrata će se smanjivati, a samim tim će se i „vernost“ slike uvećavati [2].

## *Escape time* algoritam za iscrtavanje

Ovo je jedan od najranijih algoritama korišćenih za iscrtavanje fraktala i dan danas je za pojedine slučajeve jedini. Dosta je popularan zbog svoje jednostavnosti.

Ovaj algoritam se zasniva na teoremi koja kaže da tačka ***z*** pripada Žulijinom skupu ako i samo ako njena orbita ostaje ograničena prilikom iteracija funkcije ***f***. Po ovoj teoremi, za svaku polinomsku funkciju ***f*** može se izračunati broj **R** takav da ***z* → ∞** ako i samo ako postoji prirodno **k** takvo da je ∣ ***f* k (*z*)** ∣ **> R**. Ovakvo (i svako veće) **R** nazivamo **poluprečnik izlaska**, a kružnicu centriranu u 0 sa poluprečnikom **R** nazivamo **kružnica izlaska** [2].

*Escape time* algoritam za iscrtavanje Žulijinog skupa kaže sledeće: Neka je **N** fiksiran prirodan broj koji označava maksimalan broj iteracija. Neka je **Z** proizvoljan piksel na slici, a ***z*** kompleksan broj određen pikselom **Z**. Ako je ***z*** > **R** onda sledi da tačka ***z*** ne pripada ispunjenom Žulijinom skupu. Ako je ***z* ≤ R** tada proveravamo da li je ***f* (*z*) > R**. Ako jeste, tada ponovo sledi da ***z*** ne pripada ispunjenom Žulijinom skupu. U suprotnom, proveravamo da li je ***f*****2 (*z*) > R**. Ovaj proces ponavljamo dok ne stignemo do **N**-te iteracije, to jest maksimalnog broja iteracija. Ako je ***f*****n (*z*) < R**, tada smatramo da tačka ***z*** pripada Žulijinom skupu i piksel bojimo određenom, unapred zadatom bojom (najčešće crnom). Piksele koji određuju tačke van ispunjenog Žulijinog skupa bojimo nekom drugom bojom. Opisani postupak se ponavlja za svaki piksel na slici [2].

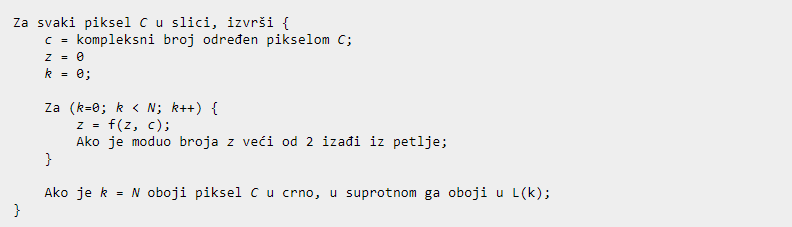
*Escape time* algoritam za iscrtavanje Mandelbrotovog skupa je gotovo identičan. Ideja se sastoji u tome da se za svaki piksel **C** posmatra niz ***f****c* (0), ***f****c***2** (0), ***f****c***3** (0),…, ***f****c***N** (0), gde je ***c*** kompleksan broj određen pikselom **C**, a **N** maksimalan broj iteracija. Ako je navedeni niz ostao ograničen kružnicom poluprečnika 2, tada smatramo da tačka ***c***  pripada Mandelbrotovom skupu. U suprotnom smo sigurni da ne pripada [2].

Što se tiče maksimalnog broj iteracija, njegovim povećavanjem se smanjuju greške pri iscrtavanju, ali se vreme izvršavanja algoritma povećava. Takođe, povećanjem iteracija dolazi i do pojave šuma, koji se može otkloniti iscrtavanjem slike u većoj rezoluciji a zatim njenim smanjivanjem na željenu. U praksi se najčešće eksperimentalno dolazi do maksimalnog broja iteracija koji daje zadovoljavajuće rezultate.

U navedenim *escape time* algoritmima, najviše vremena se utroši na tačke koje pripadaju ispunjenom Žulijinom, odnosno Mandelbrotovom skupu, zbog toga što ove tačke prolaze kroz maksimalan broj iteracija. Postoje načini da se navedeni algoritmi optimizuju [2]:

* Provera periodičnosti orbite tačke *z*. To se može uraditi tako što se prilikom svake iteracije vrednost *f**k* (*z*) upoređuje sa vrednostima *f* *k-1* (*z*), *f* *k-2* (*z*),…, *f* *k-l* (*z*). Na ovaj način će se detektovati preperiodične tačke sa periodom ne većim od *l* + 1. Jasno je da se ovakvim pristupom troši dodatna memorija i dodatne instrukcije poređenja.
* Način optimizacije *escape time* algoritma koji je karakterističan za Mandelbrotov skup. Znamo da glavna kardioida čini većinu površine Mandelbrotovog skupa, a znamo i parametrizaciju glavne kardioide. Prema tome, pre iteracije polinoma ***f***c možemo veoma „jeftino“ proveriti da li tačka *c* pripada glavnoj kardioidi. Ako je to slučaj, tada znamo da tačka *c* pripada i Mandelbrotovom skupu. Ova optimizacija ima smisla samo ako veliki deo slike pripada kardioidi.

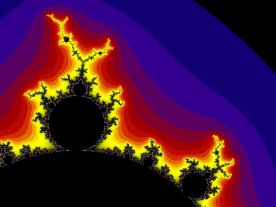
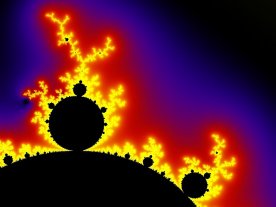
Najjednostavnija modifikacija bojenja *escape time* algoritmom je da piksele van kružnice izlaska bojimo različitim bojama na osnovu toga koliko iteracija im je bilo potrebno da izađu van kružnice. Na slici 3.1 se može videti pseudokod za iscrtavanje Mandelbrotvog skupa sa bojenjem piksela u zavisnosti od broja iteracija koji je potreban da bi se izračunalo da li tačka pripada ili ne pripada Mandelbrotovom skupu. Funkcija L(*k*) je funkcija boje, koja za zadati broj iteracija vraća boju.



*Slika 3.1 Pseudokod escape time algoritma za iscrtavanje Mandelbrotovog skupa* [2]

## Kontinualni algoritmi za iscrtavanje

U prethodnom odeljku je bilo reči o *escape time* algoritmu koji spada u algoritme diskretnog iscrtavanja, odnosno koristi diskretne vrednosti (broj iteracija – uvek prirodan broj) za iscrtavanje slike. Ovo proizvodi „trakast“ efekat, odnosno, jasno su vidljive granice između boja kojima pripadaju pikseli u zavisnosti od iteracija. Ovo u nekim slučajevima može biti korisno, međutim, u interesu je bilo i da se ovaj efekat ublaži, odnosno prikrije (slika 3.2). Pošto *escape time* algoritam u suštini meri udaljenost proizvoljne tačke od neke granice **R**, cilj je bio da se umesto toga razviju kontinualne funkcije za merenje ove udaljenosti. Nijedna od njih ne daje u potpunosti preciznu Euklidsku udaljenost, ali generalno daju zadovoljavajuće kontinualne vrednosti [4].

*Slika 3.2 Levo se nalazi fraktal iscrtan uz pomoć diskretnog (escape time) algoritma, dok se desno nalazi fraktal iscrtan uz pomoć kontinualnog algoritma* [4]

# **Poboljšanje performansi iscrtavanja i paralelizacija algoritama za iscrtavanje Žulijinog i Mandelbrotovog skupa**

Ono što se može primetiti iz prethodnog poglavlja jeste to da ukoliko želimo da iscrtamo fraktal iz Mandelbrotovog ili Žulijinog skupa, tako da dobijemo sliku visokog kvaliteta, potrebno je da iscrtamo sliku i do nekoliko puta veće rezolucije nego što nam je potrebna, pri čemu broj iteracija za proveru da li neki kompleksan broj (odnosno piksel slike) pripada skupu potencijalno nije mali i meri se u hiljadama iteracija. Ovo može imati veliki uticaj na performanse, odnosno na količinu vremena koje je potrebno da bi se ovakav fraktal iscrtao. U ovom poglavlju će biti reči o tome kako bi mogle da se poboljšaju performanse iscrtavanja ovakvih fraktala. Sve tehnike koje se budu pominjale u ovom poglavlju se odnose na primer iscrtavanja *escape time* algoritmom i njegovo ubrzavanje, pošto je on korišćen u realizaciji projekta i najjednostavniji je za objašnjavanje.

## Sekvencijalno iscrtavanje i optimizacija sekvencijalnog iscrtavanja

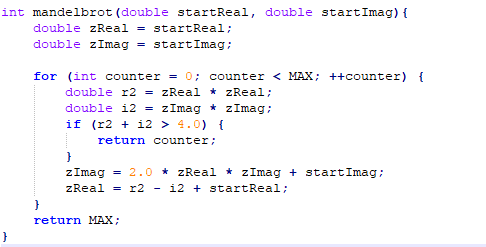
Ono što se može primetiti kod *escape time* algoritma, jeste to da mu je za sliku visoke rezolucije, sa velikim brojem iteracija za proveru, potreban ogroman broj ukupnih iteracija za iscrtavanje. Slike visoke rezolucije mogu imati milione ili desetine miliona piksela. Za dosta tih piksela su često potrebne hiljade iteracija da bi se utvrdilo da li pripadaju ili ne pripadaju Mandelbrotovom skupu, što dovodi do velikog rasta ukupnog broja iteracija koji je potreban da se čitav fraktal iscrta.

Najjednostavnije optimizacije koje mogu da se izvedu su aritmetičkog tipa, odnosno uprošćavanje matematičkih operacija tako da se lakše izvršavaju na računaru. U to spada deljenje kompleksnog broja na dva realna pa onda vršenje operacija nad tim brojevima, kvadriranje jednačina da se ne bi vršila računanja sa kvadratnim korenom, itd.

Neke od optimizacija koje mogu u zavisnosti od potrebe da dosta smanje broj potrebnih iteracija za iscrtavanje fraktala su provera periodičnosti orbite tačke *z* i provera da li posmatrana tačka pripada glavnoj kardioidi (ova provera se odnosi na Mandelbrotov skup). Ove provere su detaljnije opisane u odeljku 3.1. Takođe, Mandelbrotov set je simetričan u odnosu na realnu osu, tako da ukoliko se zumira direktno prema realnoj osi, može da se vrši duplo manje proračuna [4].

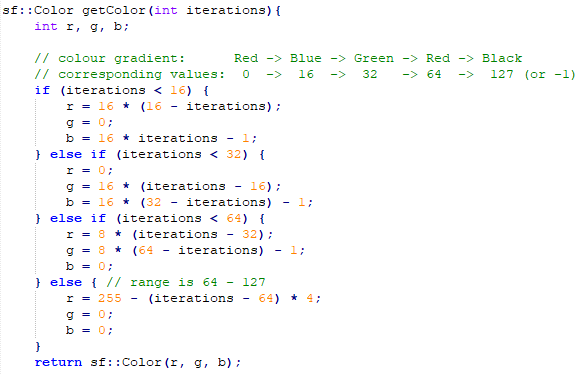
Jedna od tehnika koja takođe može značajno da ubrza iscrtavanje Mandelbrotovog seta je i *boundary tracing* [5]. Ovaj algoritam se zasniva na tome da ne postoji površina Mandelbrotovog skupa „obojena“ jednom bojom, unutar površine „obojene“ drugom bojom. To je posledica toga što je Mandelbrotov skup prosto povezan. To znači da ukoliko mogu da se pronađu granice površine koja je obojena jednom bojom, sve tačke unutar te površine će takođe biti obojene tom bojom. U zavisnosti od slučaja, ovo može da smanji broj piksela za koje je potrebno vršiti izračunavanja i preko 80%. Međutim, postoje i slučajevi u kojima ova tehnika može da vrati pogrešne rezultate, pa je ne treba uvek koristiti [5].

Na slici 4.1 se može videti funkcija koja vrši izračunavanje da li je tačka unutar ili izvan Mandelbrotovog seta i ona kao parametre prima realni i imaginarni deo kompleksnog broja, a vraća broj iteracija koji bio potreban da tačka izađe iz zadatog opsega, odnosno maksimalan broj iteracija ukoliko ne izađe. Kompleksan broj je razložen na dva realna i moduo se računa preko kvadrata umesto kvadratnog korena.



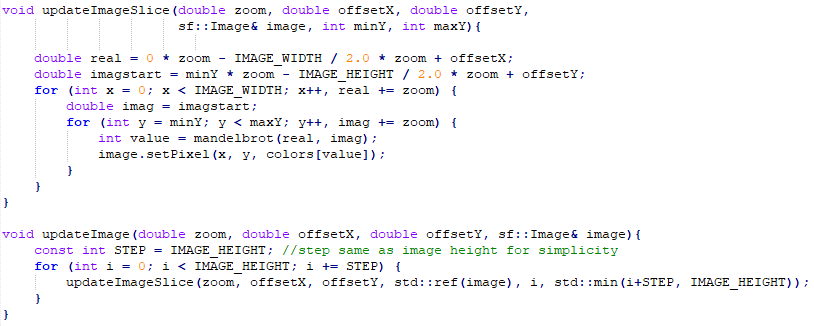
*Slika 4.1 Funkcija mandelbrot za izračunavanje broja iteracija*

Na osnovu izračunatog broja iteracija, primer funkcije sa slike 4.2 vraća određenu boju za taj broj iteracija (ovo je jednostavna funkcija bojenja kada je maksimalan broj iteracija fiksan i jednak 128, kasnije je implementirana složenija funkcija koja vrši lepše bojenje za proizvoljan maksimalan broj iteracija).



*Slika 4.2 Funkcija koja vraća boju u zavisnosti od broja iteracija*

Na slici 4.3 se mogu videti dve funkcije koje se koriste za iscrtavanje slike. Ideja je bila da se slika podeli po visini na broj niti koje ima procesor pa da te niti konkurentno obrađuju sliku, što bi dovelo do ubrzanja, pa zbog toga i dve zasebne funkcije. Pošto je u ovom odeljku reč o sekvencijalnom kodu, deo oko podele na niti je izuzet iz koda i o njemu će biti reči kasnije. Niz *colors* koji se može videti u prvoj funkciji pri dodeli boje pikselu slike, je u stvari globalni niz koji je pre toga, pri inicijalizaciji popunjen bojama na osnovu broja iteracija pomoću funkcije sa slike 4.2 (*colors[2]* daje boju dobijenu iz funkcije *getColors(2)*).



*Slika 4.3 Funkcije za iscrtavanje, odnosno osvežavanje slike*

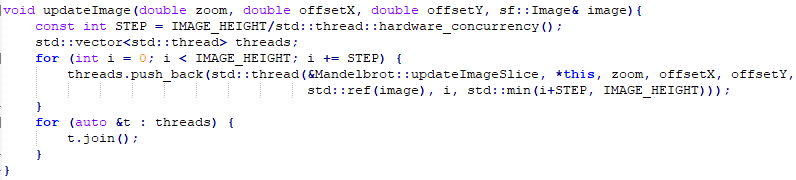
Kao što se može videti na prethodnim slikama, u samom kodu nisu implementirane neke od prethodno pomenutih optimizacija za iscrtavanje, iz razloga što je tema ovog rada poboljšanje performansi pomoću paralelizacije iscrtavanja. Paralelizacija dovodi do najvećeg poboljšanja u performansama iscrtavanja, pa jednom kada se ona uspešno izvrši, ukoliko je dodatno potrebno smanjiti broj iteracija, mogu se iskoristiti neke od prethodno pomenutih optimizacija.

## Paralelizacija algoritama za iscrtavanje fraktala

Kao što se moglo videti u prethodnom odeljku, postoji dosta načina da se poboljšaju performanse iscrtavanja fraktala. Međutim, to često nije dovoljno dobro iz prostog razloga sto postoji veliki broj piksela za koje je potrebno u dosta iteracija vršiti izračunavanja i pored svih tih optimizacija. Zbog toga je paralelizacija izvršavanja najbolji način da se poboljšaju performanse ovih algoritama.

Ono što je neophodno uraditi pre pokušaja paralelizacije bilo kog algoritma, jeste analiza algoritma i provera da li je uopšte moguće paralelizovati algoritam i u kojoj meri. Ono što je karakteristično za *escape time* algoritam, a i većinu algoritama za iscrtavanje fraktala, jeste to da se za svaki piksel vrše posebno izračunavanja i uopšte ne zavise jedna od drugih. To teoretski znači da bi izračunavanja za sve piksele mogli da obavimo skroz paralelno, u vremenu koje je potrebno da se izvrše računanja za jedan piksel. To bi naravno podrazumevalo da imamo arhitekturu odnosno računar sa dovoljnim brojem procesorskih jedinica da tako nešto izvede. To često nije slučaj, ali paralelizacija i u manjoj meri, sa dostupnim resursima, često može dosta da pomogne.

Ono što prvo padne na pamet kada se kaže paralelizacija izvršavanja, jeste paralelizacija izvršavanja na procesoru. Današnji procesori su često višejezgarni i poseduju niti koje mogu u paraleli da izvršavaju zadatke. Moderni programski jezici poseduju funkcionalnosti koje im omogućavaju da relativno jednostavno podele posao na vise niti i tako ga obavljaju paralelno ili konkurentno. Jednostavna modifikacija koda za iscrtavanje slike iz prethodnog odeljka (slika 4.4) daje čak do nekoliko puta bolje performanse pri iscrtavanju slike. Funkcija na slici 4.4 izračunavanja vrši konkurentno a ne paralelno!



*Slika 4.4 Konkurentno iscrtavanje slike*

Kao što se može videti na slici 4.4, slika fraktala se izdeli na onoliko delova koliko ima procesorskih niti i svaki deo se dodeli jednoj niti koja onda vrši izračunavanja samo za svoj deo. Ovakav pristup je relativno jednostavan za implementaciju i donosi dobra poboljšanja u performansama, međutim za slike visoke rezolucije sa velikim brojem iteracija, ovo nije dovoljno. Čak i kada bi se izvršila potpuna paralelizacija umesto konkurentnosti, ni to ne bi dovelo do željenih rezultata, jer i najbolji procesori poseduju par desetina jezgara, odnosno niti.

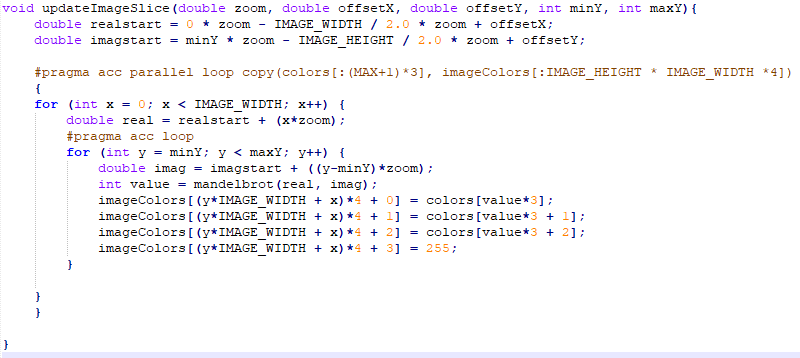
## Paralelizacija algoritama za iscrtavanje fraktala na grafičkim karticama

Za razliku od procesora, grafičke kartice poseduju stotine, pa i hiljade procesorskih jezgara koja su u mogućnosti da vrše paralelna izračunavanja nezavisno jedna od drugih. Dok je procesor računara napravljen da bude gotovo beskonačno podesiv i prilagodi se bilo kakvom zadatku koji se od njega traži, grafičke kartice su komponente koje omogućavaju računaru da neke stvari obavlja brže kroz hardver optimizovan za specifičan scenario korišćenja. Procesor je napravljen da se fokusira na jedan zadatak unutar nekog posla, ali taj zadatak može da bude gotovo bilo šta. To kao posledicu ima to da su određene primene procesora brže od drugih, ali nije maksimalno prilagođen nijednom poslu [6].

Posledica onoga za šta su grafičke kartice napravljene (3D grafika) je ta da veoma dobro vrše kompleksne matematičke operacije nad velikim nizovima podataka (pogotovo operacije pokretnog zareza). Zbog ovoga, a i zbog činjenice da poseduju hiljade jezgara koja mogu paralelno da obavljaju zadatke, grafičke kartice su idealno rešenje za paralelizaciju algoritama za iscrtavanje Mandelbrotovih i Žulijinih fraktala [6].

Grafička kartica je akceleratorski uređaj sa posebnom namenom koji ima posebnu hijerarhiju memorije i nečim ograničen transfer. Da bi bila korišćena za izračunavanja kao što su ova potrebna za iscrtavanje fraktala, potrebna je posebna specifikacija koja omogućava pisanje optimizovanog koda koji uzima u obzir sve prethodno navedene karakteristike akceleratorskog uređaja. U ovom projektu je za te potrebe korišćen *OpenACC* [7]koji spada u otvorenu specifikaciju koja je namenjena da podrži širok spektar akceleratorskih uređaja. *OpenACC* omogućava pisanje optimizovanog koda koji uzima u obzir posebnu hijerarhiju memorije na akceleratorima i transfer podataka umesto programera. To znači da je za programera promenljiva samo objekat koji može biti u memoriji sistema i akceleratorskog uređaja. *OpenACC* se oslanja na pragme za definisanje paralelizacije i podrška za njega mora biti ubačena direktno u kompajler [6].

Često, sekvencijalni kod nije pogodno napisan za paralelizaciju na grafičkoj kartici, to jest, nije napisan tako da je moguće samo dodati *OpenACC* direktive i da odjednom sve počne da se obavlja paralelno. Da bi sekvencijalna funkcija za iscrtavanje fraktala koju smo videli u odeljku 4.1 (slika 4.3) bila pogodna za paralelizaciju, bilo je potrebno malo izmeniti strukturu same funkcije, ali i strukture podataka koje ta funkcija koristi, a zatim dodati odgovarajuće direktive koje bi omogućile prenos podataka i računanje na grafičkoj kartici. Na slici 4.5 se može videti izmenjena verzija te funkcije.



*Slika 4.5 Funkcija za iscrtavanje fraktala napisana tako da može da se izvrši na grafičkoj kartici*

Sekvencijalna verzija funkcije (slika 4.3) je kao parametar primala objekat tipa *sf::Image* [8]koji je objekat klase koja se nalazi u *SFML* [8]biblioteci korišćenoj za iscrtavanje slike. Zatim je na tom objektu direktno vršila bojenje piksela u odgovarajuću boju. Ovo nije pogodno za izvršavanje na grafičkoj kartici, jer bi čitav taj objekat koji predstavlja sliku sa svim svojim metodama morao da se prenese na grafičku karticu radi jednostavnog obavljanja radnje bojenja piksela, što nije ni praktično, a ni jednostavno za izvesti. Zbog toga je čitav taj objekat slike izbačen kao parametar i umesto njega se koristi niz tipa *uint8\_t* koji sadrži vrednosti crvene, plave, zelene i alfa komponente boje za svaki od piksela i to poređanih po redu u kome se pojavljuju na slici. Taj niz se pomoću *copy* [6]direktive prebaci na grafičku karticu, popuni i onda vrati na domaćina, gde se na osnovu njega konstruiše objekat klase *sf::Image* koji se nakon toga iscrta. Pored ovog niza, na grafičku se prebacuje i niz koji je prethodno inicijalizovan vrednostima crvene, zelene i plave komponente boje za svaku od iteracija od nulte do maksimalnog broja iteracija.

Pored ove promene struktura podataka koje se prenose na grafičku, bilo je neophodno i preurediti samu funkciju tako da se izgube međusobne zavisnosti promenljivih u trenutnoj iteraciji od vrednosti iz prethodne iteracije. Konkretno, to je bilo potrebno učiniti za promenljive *real* i *imag*.

Nakon obavljanja ovih promena je postala moguća paralelizacija ovog koda na grafičkoj kartici. Dodavanjem *OpenACC* direktiva za paralelizaciju petlji, spoljašnja petlja je prebačena da se izvršava u *gang* [6]režimu na akceleratoru, dok se unutrašnja izvršava u *vector* [6] režimu. Time smo postigli paralelno izvršavanje provera da li pikseli, odnosno kompleksni brojevi pripadaju Mandelbrotovom setu. Što se tiče unutrašnje petlje, možemo videti da ona poziva funkciju za izračunavanje broja iteracija za zadati piksel i u njoj takođe postoji petlja (slika 4.1). Međutim, ta petlja ima zavisnost između iteracija, odnosno izračunavanje u trenutnoj iteraciji koristi vrednosti dobijene u prethodnoj iteraciji. Zbog toga nije moguća paralelizacija ove petlje, pa se ona izvršava sekvencijalno. Takođe, u samoj petlji postoji i grananje.

Ono što je dobijeno kao rezultat ove paralelizacije je 8 do 9 puta brže iscrtavanje fraktala nego pomoću sekvencijalne verzije. Za sliku Mandelbrotovog skupa rezolucije 1920x1080 i 60.000 iteracija za proveru da li pripada Mandelbrotovom skupu, paralelnoj verziji koja se izvršava na grafičkoj kartici je bilo potrebno oko 9 sekundi da izračuna boje za sve piksele, dok je sekvencijalnoj verziji trebalo nešto manje od 80 sekundi, što je skroz zadovoljavajuće ubrzanje. Grafička kartica korišćena za iscrtavanje je *NVIDIA GeForce GTX 850M* [9]*.*

## Uticaj preciznosti izračunavanja na paralelizaciju i performanse iscrtavanja fraktala

Što se tiče preciznosti izračunavanja, ovde će se govoriti o preciznosti izračunavanja operacija nad realnim brojevima koji su najčešće u računarima predstavljeni kao brojevi sa pokretnim zarezom. Kada se govori o preciznosti ovakvih brojeva, najčešće se govori o *single precision* [10]i *double precision* [11] brojevima sa pokretnim zarezom po *IEEE 754* [12]standardu. *Single precision* brojevi se sastoje od 32 bita (1 bit za znak, 8 bitova za eksponent i 23 bita za vrednost broja) i imaju preciznost od 7 decimalnih cifara. *Double precision* brojevi se sastoje od 64 bita (1 bit za znak, 11 bitova za eksponent i 52 bita za vrednost) i imaju preciznost od 15 decimalnih cifara. Ovo se u *C* programskom jeziku mapira na *float* i *double* tipove podataka.

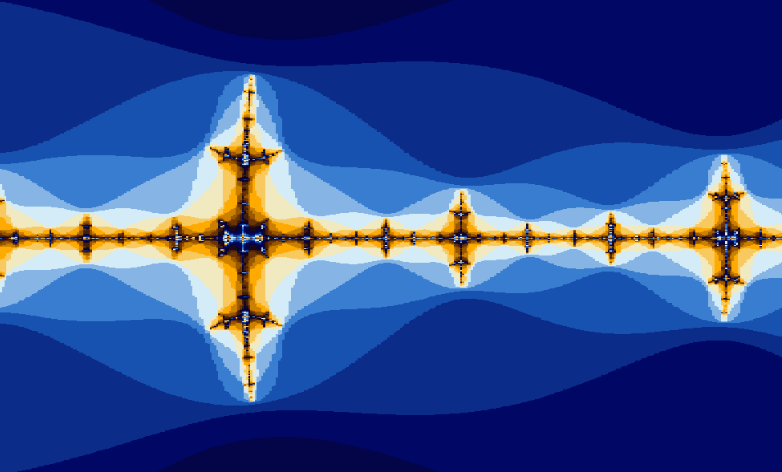
Ono što se javlja kao problem pri aritmetičkim operacijama sa ovim brojevima jeste problem zaokruživanja. Većina operacija sa realnim brojevima će proizvesti realne brojeve koji ne mogu biti najtačnije predstavljeni korišćenjem određenog broja bitova (jer imaju beskonačno mnogo decimala iza decimalnog zareza), pa se oni zaokružuju na vrednosti koje mogu biti predstavljene zadatim brojem bitova. Ovo dovodi do pojave apsolutne i relativne greške pri zaokruživanju, i ona može drastično da poraste ukoliko se vrši dosta aritmetičkih operacija nad takvim brojevima, pogotovo ako su oni ekstremno mali ili veliki ili je razlika između njih velika. To dovodi do smanjene preciznosti, odnosno netačnih rezultata pri izvršavanju ovih operacija. Da bi se izbegle razlike u rezultatima izračunavanja koje bi nastale kao posledica različitih načina zaokruživanja vrednosti na različitim arhitekturama, uveden je *IEEE 754* standard koji tačno definiše kako su brojevi sa pokretnim zarezom predstavljeni u memoriji i kako se obavlja zaokruživanje tih brojeva pri svakoj od aritmetičkih operacija. To dovodi do toga da obavljanje istog računa sa istim brojevima na različitim arhitekturama najčešće proizvodi iste rezultate i pored postojanja grešaka nastalih kao posledica zaokruživanja [13].

Što se tiče grafičkih kartica, sve moderne grafičke kartice imaju hardversku podršku i za *single precision* i za *double precision* račune. Međutim, kartice se razlikuju od modela do modela po tome koliko dobro podržavaju jedan ili drugi račun. Većina kartica namenjenih običnim korisnicima (na primer *NVIDIA GeForce GTX* serija kartica) ima dosta bolji *throughput* za *single precision* brojeve, nego za *double precision* (*NVIDIA GeForce GTX 1060* ima *double precision throughput* koji je 1/32 od *single precision throughput*-a). To je posledica toga da su one namenjene većinom za renderovanje složene grafike i slike gde nije potrebna preciznost koju omogućava *double precision*. Sa druge strane, grafičke kartice namenjene za računanje koje se koristi u naučne svrhe (kao što je na primer *NVIDIA Tesla* serija [14]) imaju dosta bolji *throughput* za *double precision* brojeve, jer je za njihove primene preciznost u proračunima dosta bitnija.

Računari mogu da vrše aritmetičke operacije nad brojevima i takve da preciznost koja se dobija bude dosta veća od *double precision*, odnosno 64 bita. Teoretski, ta preciznost može da bude onolika koliko dozvoljava dostupna memorija na računaru. Računanje sa ovako velikom preciznošću se naziva i aritmetika proizvoljne preciznosti odnosno *arbitrary-precision arithmetic* [15]. Moderni programski jezici često imaju ugrađenu podršku za ovakve operacije ili postoje biblioteke koje omogućavaju takav račun. Oni vrednosti proizvoljne preciznosti ne čuvaju pomoću fiksnog broja bitova koji odgovara procesorskim registrima, već umesto toga tipično koriste nizove varijabilne dužine u koje skladište cifre. Ono što je karakteristično za aritmetiku proizvoljne preciznosti, jeste to da omogućava veću preciznost pri računanju, ali to za posledicu ima drastičan pad performansi samog računa. To je zbog toga što račun nije hardverski podržan u samim procesorskim jedinicama kao što je to inače slučaj, već on mora da se vrši softverski. Zbog toga aritmetika proizvoljne preciznosti svoju primenu nalazi u oblastima u kojima brzina izvršavanja nije kritičan faktor ili u oblastima gde su neophodni precizni rezultati izračunavanja sa jako velikim ili jako malim brojevima [16].

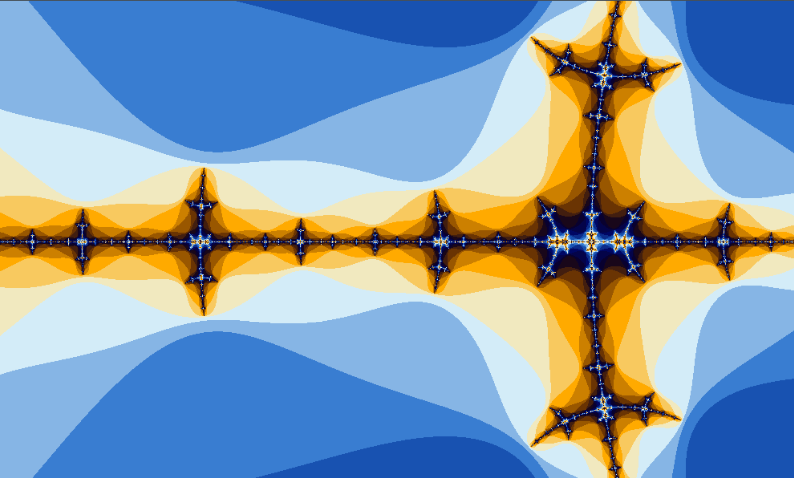
Aritmetika proizvoljne preciznosti može biti veoma korisna u operacijama sa brojevima sa pokretnim zarezom, zbog toga što povećanje preciznosti dovodi do smanjenja greške pri zaokruživanju brojeva i samim tim se dobijaju dosta bolji rezultati. Operacije sa tako povećanom preciznošću će naravno dovesti do značajnog pada u performansama u odnosu na operacije nad brojevima sa običnom preciznošću.

Čitava ova priča oko preciznosti računanja brojeva sa pokretnim zarezom je veoma bitna za iscrtavanje fraktala kao što su Mandelbrotov i Žulijin skup. Da bi se dobio efekat „zumiranja“ u skup koji se iscrtava, potrebno je vršiti računanja za sve manje i manje brojeve, sa sve većom preciznošću, što se više zumira u skup. Što se više zumira u skup, to je sve bitnije da vrednosti cifara dalje od decimalnog zareza budu što tačnije, da bi se skup lepo iscrtao. Takođe, onda kada brojevi postanu premali, odnosno, dostigne se granica do koje *single* i *double precision* brojevi mogu da predstave male realne brojeve, tada više nije moguće „zumirati“ u skup i potpuno se gubi slika koju je potrebno dobiti (dobija se efekat zamutnjivanja slike, a nakon još nekoliko iteracija zumiranja dolazi i do potpunog gubitka obrisa na slici). Taj efekat se može primetiti na slici 4.6 pri vrednosti promenljive *zoom* od 8\*10-17 pri čemu je račun vršen u duploj preciznosti (*double precision*).

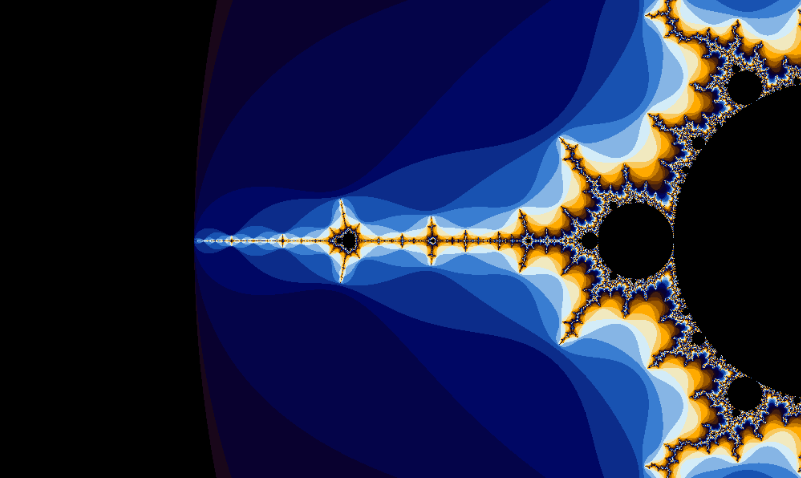


*Slika 4.6 Izgled Mandelbrotovog skupa za vrednost promenljive zoom od 8\*10-17*

Pri manjim vrednostima promenljive *zoom*, slika nije zamućena (slika 4.7 i slika 4.8).



*Slika 4.7 Izgled Mandelbrotovog skupa za vrednost promenljive zoom od 8\*10-13*



*Slika 4.8 Izgled Mandelbrotovog skupa za vrednost promenljive zoom od 12.5\*10-3*

Da bi se rešio ovaj problem prilikom „zumiranja“ u skup, može da se koristi aritmetika proizvoljne preciznosti. Međutim, pošto je ova aritmetika dosta zahtevna, i dosta se negativno odražava na performanse iscrtavanja, pošto moderne grafičke kartice i procesori nemaju hardversku podršku za nju, potrebno je naći neku sredinu. To znači, vršiti iscrtavanja fraktala uz pomoć klasične, *double precision* aritmetike dok god je to moguće, odnosno pogodno, a zatim, pri dubljem „zumiranju“ u skup, preći na aritmetiku proizvoljne preciznosti. Ovo će kao posledicu imati da će pri većem zumiranju biti potrebno da se slike duže iscrtavanju, ali neće dolaziti do zamućivanja i gubljenja kvaliteta slike.

Iscrtavanje ovih fraktala pomoću aritmetike proizvoljne preciznosti nije implementirano u ovom projektu. Programski jezik *C++* ima dobro razvijene biblioteke za ovakva izračunavanja, međutim one nisu podržane od strane *OpenACC* specifikacije, pa tipovi podataka koji se u njima koriste za reprezentaciju brojeva veće preciznosti ne mogu da budu prebačeni za računanje na grafičkoj kartici. Bilo je pokušaja u projektu da se ovo reši, međutim svi su bili bezuspešni. Ono što bi moglo da se uradi, jeste da se ručno implementiraju funkcije koje će vršiti aritmetiku proizvoljne preciznosti, odnosno neke preciznosti veće od duple preciznosti, pa se one označe kao *OpenACC* rutine i na taj način pozivaju. U tom pravcu bi se dalje razvijao ovaj projekat.

# **Zaključak**

U prvom poglavlju ovog rada je bilo reči o fraktalima uopšteno, a kasnije i o fraktalima koji pripadaju Mandelbrotov i Žulijinom skupu. U drugom poglavlju je objašnjeno šta su i na koji način se mogu jednostavno izračunati ovi fraktali. Treće poglavlje se bavi nekim od popularnih algoritama koji se koriste za iscrtavanje ovih fraktala i njihovim prednostima i manama pri iscrtavanju. U četvrtom poglavlju se govori o sekvencijalnim i paralelnim implementacijama ovih algoritama, kao i o tome kako se paralelni račun može prebaciti i obavljati na grafičkoj kartici. Na kraju je bilo reči i o preciznosti izračunavanja i tome kako ona utiče na kvalitet iscrtanog fraktala ali i na performanse crtanja.

Ono što se može izvesti kao zaključak nakon svega, je to da paralelizacija računa i prebacivanje računanja sa procesora na grafičku karticu, može najviše da doprinese poboljšanju performansi iscrtavanja. Takođe, postoje i neke aritmetičke i matematičke metode pomoću kojih možemo da smanjimo broj iteracija za iscrtavanje slike i time poboljšamo performanse. Samo poznavanje osobina fraktala i njihovog ponašanja takođe može da utiče na to da se napiše kod koji će raditi brže. Na kraju se moglo videti i da je ponekad potrebno koristiti aritmetiku proizvoljne preciznosti radi dobijanja tačnijih rezultata, ali da to ima velike posledice na performanse izvršavanja.

U projektu koji je realizovan postoji dosta prostora za napredak i još veće poboljšanje performansi iscrtavanja. Mogle bi se implementirati razne matematičke i aritmetičke metode koji bi smanjile ukupan broj iteracija koji je potreban da se dođe do krajnje slike fraktala. Takođe, razvojem znanja o paralelizaciji i samih tehnologija za paralelizaciju na grafičkim karticama bi moglo još više da se paralelizuje ovo rešenje i da se time dobiju bolje performanse. Takođe bi se mogla implementirati aritmetika proizvoljne preciznosti radi dobijanja boljih slika fraktala prilikom „dubokih zumiranja“ u fraktal.

# **LITERATURA**

[1] Wikipedia, Fractal, <https://en.wikipedia.org/wiki/Fractal>

[2] Matematički fakultet Beograd, Ubavić N, Žulijev i Mandelbrotov skup, <http://alas.matf.bg.ac.rs/~mm16310/text/zulijev_i_mandelbrotov_skup.html>

[3] Wikipedia, Mandelbrot set, <https://en.wikipedia.org/wiki/Mandelbrot_set>

[4] Mathematical Institute of the Serbian Academy of Sciences and Arts, Distance Estimators, <https://www.mi.sanu.ac.rs/vismath/javier/b3.htm>

[5] Wikipedia, Boundary tracing, <https://en.wikipedia.org/wiki/Boundary_tracing>

[6] Petrović V, Materijali sa predavanja, *Tehnologije paralelnog programiranja,* Fakultet tehničkih nauka Novi Sad*,* 2020

[7] OpenAcc, <https://www.openacc.org/>

[8] SFML, <https://www.sfml-dev.org/documentation/2.5.1/>

[9] Nvidia, GeForce, <https://www.geforce.com/hardware/notebook-gpus/geforce-gtx-850m>

[10] Wikipedia, Single Precision Floating point numbers, <https://en.wikipedia.org/wiki/Single-precision_floating-point_format>

[11] Wikipedia, Double Precision Floating point numbers, <https://en.wikipedia.org/wiki/Double-precision_floating-point_format>

[12] GeeksForGeeks, IEEE 754, <https://en.wikipedia.org/wiki/Double-precision_floating-point_format>

[13] Validlab, What Every Computer Scientist Should Know About Floating-Point Arithmetic, <http://www.validlab.com/goldberg/paper.pdf>

[14] Nvidia, Tesla, <https://www.nvidia.com/en-us/data-center/tesla/>

[15] Wolfram, Arbitrary Precision, <http://mathworld.wolfram.com/ArbitraryPrecision.html>

[16] Wikipedia, Arbitrary precision arithmetic, <https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic>