Analiza numeričkih metoda zasnovana na primjeni informacijske tehnologije

Jeličić, Luka

Sveučilište u Rijeci - Odjel za informatiku
Preddiplomski studij informatike

Luka Jeličić

Analiza numeričkih metoda zasnovana na primjeni informacijske tehnologije

Rijeka, srpanj, 2018.

Sveučilište u Rijeci - Odjel za informatiku
Preddiplomski studij informatike

Luka Jeličić

Analiza numeričkih metoda zasnovana na primjeni informacijske tehnologije

MENTOR: dr.sc.Božidar Kovačić, doc.
KOLEGIJ: Numerička matematika, Paralelno programiranje na heterogenim sustavima

Sadržaj

1 SAŽETAK

U ovom radu predstavljen je program CUDA te njegove najbitnije karakteristike i svojstva. Definirane su numeričke metode linearni i kubični splajn. Objašnjeno je kako se mogu numeričke metode programirati za paralelnu obradu koristeći program CUDA. Također, definirana je trapezna formula, generalizirana trapezna formula i Simpsonova formula koje smo popratili grafičkim prikazom i izračunom u programu Python.

2 KLJUČNE RIJEČI

- CUDA,
- Python,
- CPU,
- GPU,
- paralelna obrada,
- podataka,memorije,
- registarska memorija,
- dijeljiva memorija,
- globalna memorija,
- linearni spalajn,
- kubični splajn,
- trapezna formula,
- generalizirana ,
- trapezna formula,
- Simpsonova formula.

3 UVOD

U ovom radu ću se baviti problemom interpolacije funkcije po dijelovima primjenom metoda Simpsonove i trapezne formule. Pritom za lakše razumijevanje tih metoda objasnit ću kako funkcioniraju metode linearnog i kubičnog splina. Pitanje je: što je interpolacija funkcije po dijelovima i po čemu je ona bolja od interpolacije funkcije na čitavoj domeni? Kada se radi interpolacija funkcije na čitavoj domeni, ona ima loša svojstva i vrlo je nestabilna. Nestabilnost i loša svojstva postaju izražajnija što je veći stupanj polinoma. Od petog stupnja nadalje nestabilnost je prevelika da bi se koristila interpolacija na čitavoj domeni. U praksi niti ne koriste polinomi visokog stupnja. Da bi se dobila točnija interpolacija radi se interpolacija po dijelovima na način da se segment podjeli na više podsegmenata te se na tim podsegmentima koristi interpolacija polinomima nižeg stupnja. Tim postupkom se smanjuje ukupna greška interpolacije. Kada se segment podjeli na podsegmente pristupa se interpolaciji na način da se odabere polinom nekog stupnja te se tim polinomom interpolira funkcija na odabranom segmentu. Ovisno u odabranom polinomu razlikujemo više različitih splinova odnosno rezultirajućih funkcija postupka integracije.
Glavna razlika među splinovima je u njihovoj glatkoći. Prilikom biranja splina vodi se briga o glatkoći funkcije, jer što je glađa funkcija to joj je ukupna greška manja i samim time je funkcija primjenjivija. Ovisno o glatkoći funkcije razlikujemo više različitih splinova. Neki od njih su linearni, kvadratni, kubični itd. Svi numerički izračuni se rade u Pythonu. Pitanje je: što je Python?

Python je višenamjenski programski jezik nastao 1991 u Nizozemskoj. Kreator tog jezika je Guido van Rossum, nizozemski student koji je bio nezadovoljan kompleksnošću jezika poput C-a i C++-a. Njega simbolizira dinamično pisanje i automatsko upravljanje memorijom. Podržava više programskih paradigmi uključujući: objektno-orijentiranu, imperativnu, proceduralnu i funkcionalnu. Python također ima vrlo velik broj standardnih biblioteka. Biblioteke se ubacuju u Python na jedan od četiri načina:

1. Ubacivanje punog imena biblioteke.
   Primjer: import numpy, import matplotlib.
2. Ubacivanje sa sinonimima definiranim putem ključne riječi **as**.
   Primjer: import numpy as np, import scipy as sci
3. Ubacivanje podbiblioteka pomoću riječi from.
   Primjer: from statistics import mean
4. Ubacivanje svih funkcija iz biblioteke pomoću riječi from i operatora *.
   Primjer: from numpy **import ***

Najčešće korištena biblioteka za rad sa splinovima je numpy.
Numpy je standardna biblioteka u Pythonu u kojoj ima jako puno funkcija koje se koriste za pripremu podataka koji će se koristiti u numeričko obradi. Numpy

je generalno najkorištenija biblioteka funkcija. Pitanje je: Zbog čega je numpy generalno najkorištenija biblioteka funkcija? Numpy je generalno najkorištenija biblioteka funkcija zbog velike raznovrsnosti funkcija koje mogu sa lakoćom riješiti veliki broj zadataka uključujući: Ubacivanje tekstualnih datoteka u program, Spremanje podataka iz programa u tekstualne datoteke, Generiranje numeričkih polja podataka iz tekstualnih datoteka uz pomoć regularnih izraza, Kreiranje i manipulacija numeričkih polja. Numerička polja se razlikuju od običnih polja po tome što numerička polja podržavaju matematičke izračune dok sa običnim poljima to nije slučaj. Druga važna biblioteka za rad sa splinovima je Matplotlib tj. njena podbiblioteka Pyplot koja se preferira koristiti za crtanje grafova u Pythonu. Postoji i biblioteka Pylab koja se koristi za brzo crtanje grafova jer u sebi sadrži numpy biblioteku. Uz to sadrži funkcije koje služe crtanju grafa poput funkcije plot koja crta graf.

4 CUDA

CUDA je računalna arhitektura namijenjena za paralelnu obradu i programiranje aplikacijski sučelja, kreirana 2007. godine od strane tvrtke *Nvidia Corporation.*
Prilikom rada sa CUDA-om najčešće se koriste jezici C i C++ koji se koriste kao backend[1] aplikacije, dok se čitav program programira u Python-u. Također omogućuje programerima rad sa grafičkom procesnom jedinicom uz pomoć CUDA-e.

Platforma omogućuje pristup virtualnom setu instrukcija te paralelnim elementima za pokretanje na računskim jezgrama[2].
CUDA služi za izgradnju aplikacijskih sučelja nižeg i višeg levela. Aplikacijska sučelja nižeg levela upravljaju događajima unutar računala u specifičnim grupama definiranim od strane korisnika koje nazivamo setovi događaja.Iskusnim programerima omogućuje zrnatu mjerljivost i kontrolu PAPI sučelja [3].

Slika 1: Prikazuje CUDA logo

4.1 Kratka povijest i razvoj CUDA-e

Gordon Moor [4] je objavio 1965. godine u časopisu *Eletronic* predviđanja koja su vrijedila sve do početka 2000. godine prema kojima su rasle periformanse računala, a koja su se zasnivala na procesu minimizacije[5] U jednom trenutku je došlo do problema da se više ne može koristiti minimizacija te se mora pristupiti novom načinu ubrzavanja obrade. Jedna od mogućnosti na koji bi se način mogla ubrzati obrada je proizvodnja čipova s više procesorskih jedinica. Pritom su se razvila dva različita pristupa tom problemu.

1. **Multicore arhitektura**
2. **Manycore arhitektura**

Multicore se sastoji od više jezgri na istom čipu koji su lokalno povezani.Osnovni cilj je brže izvođenje programa. To se postiže kompleksnom logikom koja omogućava

[1]Backend aplikacije su aplikacije koje sadrže logiku čitavog programa u sebi. Korisnik na njih nema utjecaj.

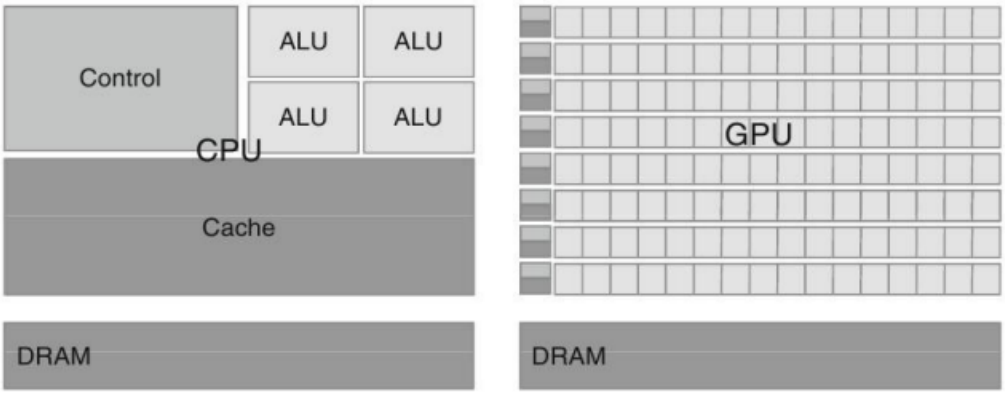
[2]Računske jezgre se koriste za ubrzanje izvođenja programa koji je izdvojen iz osnovnog programa, ali je korišten od strane tog programa.

[3]PAPI sučelje je aplikacijsko sučelje sa poboljšanim performansama koje su omogućene u mnogim današnjim modernim mikroprocesorima.

[4]Gordon Moor, rođen 3.siječnja,1929. američki inženjer i jedan od osničava tvrtke Intel.

[5] Minimizacija je sabijanje većeg broja tranzistora po jedinici površine čipa.

da se instrukcije na jednoj niti izvršavaju paralelno, iako je njihov redoslijed zadan drugačije, čuvajući sekvencialnu prirodu programa. Međutim, ako se doda više jezgri tada lako dolazi do zagušenja niti preko kojih se obavlja komunikacija između procesa. Ako promatramo glavnu ideju **manycore-**a , ona podrazumijeva da je broj jezgri dovoljno velik da se klasične multiprocesorske tehnike više ne mogu koristiti. Kod manycore-a arhitektura je organizirana u grupe po 8 jezgri te one dijele zajedničku upravljačku jedinicu te cache u kojemu su instrukcije koje se koriste za procesiranje. Glavna ideja ove arhitekture je povećana propusnost, tj. da se odjednom može izvršiti više instrukcija.



Slika 2: Prikazuje Multicore i manycore arhitektura

Na prvi pogled grafički procesori djeluju kao idealno rješenje za buduća ubrzanja aplikacija, ali na tom putu postojale su brojne prepreke.

Prva prepreka bila je što su GPU uređaji namjenjeni za numeričke izračune i kao takvi ne mogu biti konkurencija CPU-u po pitanju izvođenja sekvencionalnih programa. Da bi s eovakakv problem riješio potrebno je da rastavimo izvorni program na dva dijela tako da se jedan dio završava na GPU-u, dok se drugi dio izvršava na CPU-u.

Problem je predstavljala i činjenica da su se sva izračunavanja izvršavala sa jednostrukom preciznošću te se GPU uređaji nisu mogli nositi sa brojevima koji su zahtjevali dvostruku preciznost. Noviji uređaji ne samo da podržavaju računje sa dvostrukom preciznošću, nego su po brzini prestigli CPU-ove. Drugi problem je taj što je pisanjem programa zahtjevalo uporabu funkcija grafički aplikativnih programskih sučelja.

NVIDIA je prepoznala probleme i već je 2007. godine ponudila rješenje u okviru CUDA programskog modela. Ti programski modeli nisu ništa drugo nego proširenja viših programskih jezika. Proširenja nam omogućavaju da dijelove koda pišemo u nekom višem programskom jeziku i prevodimo ga prevoditeljem za taj jezik, a drugi dio koda koji će se izvršavati na GPU-u prevodimo pomoću NVIDIA-inog prevoditelja. CUDA-ini programi više ne koriste grafičko sučelje, nego imaju sučelje opće namjene namjenjeno za paralelno programiranje.

Još jedna bitna osobina CUDA-e njena skalabilnost [6]. GPU može imati 256,512,1024 i tako dalje, izvršnih sučelja, ali tada se postavlja pitanje je li se CUDA program pisan za neku konfiguraciju može pokrenuti i na ostalima. Na to pitanje možemo odgovoriti potvrdnim odgovorom, budući da u njegovoj osnovi leže tri ključne apstrakcije modela, a one su sljedeće:

1. hijerarhijska organizacija niti,
2. dijeljiva memorija,
3. sinkronizacija uz pomoć barijere.

[6] Skalabilnost opisuje ovisnost potrebnih resursa u odnosu na količinu podataka.

4.2 Indeksiranje u CUDA-i

Obrada na CUDA grafičkim karticama izvodi se primjenom niti(threads) organiziranim u blokove i gridove.
Varijable ***threadIdx.x, threadIdx.y i threadIdx.z*** indeksiraju niti unutar pojedinog bloka. Niti unutar 1D bloka indeksiramo sa ***threadIdx.x***, unuutar 2D bloka sa ***threadIdx.x*** i ***threadIdx.y,*** a unutar 3D bloka sa ***threadIdx.x,threadIdx.y, i threadIdx.z*** varijablama.

Varijable ***blockIdx.x*** i ***blockIdx.y*** indeksiraju blokove unutar pojednog grida. Blokove unutar 1D grida indeksiramo sa ***blockIdx.x*** varijablama, a blokove unutar 2D grida sa ***blockIdx.x*** i ***blockIdx.y*** varijablama.

Osim navedenih,za indeksiranje se koriste **blockDim.x,y,z** i ***gridDim.x,y,z*** varijable, koje daju broj niti u bloki i gridu.

Primjer 1. Neka je definirano polje sa 10 elemenata, s time da koristimo 2 bloka, svaki sa po 5 niti. Za indeksiranje možemo koristiti blockDim.x varijablu (u ovom slučaju blockDim.x=5). Tada određeni element polja referenciramo indeksom (nazivamo ga idx)kako slijedi

$$intidx = vlockDim.x * blockIdx + threadIdx.x$$

idx poprima vrijednosti 0,1,2,3,4 za prvi blok, jer je blockIdx.x=0. Za drugi blok idx poprima vrijednosti 5,6,7,8,9, jer je blockIdex.x=1 i blockDim.x=5

4.3 Memorije

Svi ulazni podaci moraju se čitati iz domaćina, zatim se prebacuju na memoriju grafičkih uređaja te se tamo obrađuju. Potom se rezultati vraćaju na domaćina. Memorija na GPU-u se naziva **globalna memorija**. Ako se tijekom izračunavanja koristi samo globalna memorija ona može predstavljati "usko grlo". CGMA [7] predstavlja odnos između broja pristupa globalnoj memoriji i broja računskih operacija sa brojevima s pomičnim zarezom.
Pored globalne imamo još nekoliko vrsta memorije, a to su:

- registri
- lokalna memorija,
- dijeljena memorija,
- memorija konstanti,
- memorija tekstura.

Registri su najbrži oblik memorije koji je i do 150 puta brži od globalne memorije. Dakle, svaka nit ima posebnu registarsku memoriju kojoj samo ona može pristupiti. Sve automatske generirane varijable prilikom deklaracije se smještaju u registre. Kada se deklarira funkcija privatna kopija varijable generira se za svaku nit koja se izvršava uz pomoć funkcije. Pristupanje je veoma brzo, ali se mora paziti da se ne prijeđu kapaciteti registra, jer tada prevoditelj može premjestiti neke varijable u lokalnu memoriju što dovodi do pada performansi.

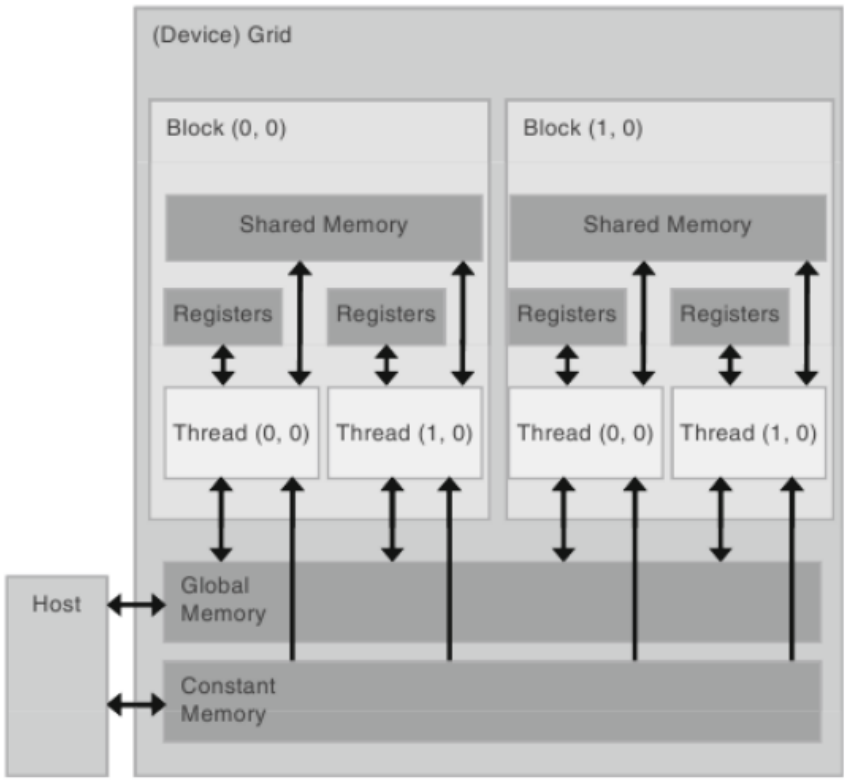
Automatski nizovi se čuvaju u lokalnoj memoriji koja je podijeljena na dijelove koji pripadaju pojedinačnim nitima, a koja je dio globalne memorije. Lokalne niti jedine mogu pristupiti dijelovima lokalne memorije na način da se za svaku nit kreira lokalna kopija niza. Kada se završi izvođenje koda sve automatske promjene niti prestaju postojati. Varijable se mogu eksplicitno deklarirati kao lokalne uporabom ključne riječi ***local*** ispred deklaracije varijabli.

Dijeljiva memorija je vidljiva za sve niti jednog bloka i moguće joj je pristupiti za vrijeme izvršavanja programa i obično se koristi za čuvanje podataka koji se često koriste u izvršnoj fazi. Da bi se deklarirala dijeljiva memorija koristi se ključna riječ ***shared***. Ta je memorija efikasno sredstvo koje omogućava da postoji suradnja između niti koje se nalaze unutar istog bloka.

Memorija konstanti je dio globalne memorije koja je uz pomoć cache-a mnogo efikasnija od ostatka globalne memorije. Prostor u memoriji namjenjen za varijable se zauzima na području memorije konstanti sa ključnom riječi ***constant***. Deklaracja mora biti izvan tijela funkcije.

[7] Compute to Global Memory Acces

Spomenimo još poslijednju vrstu memorije koja se naziva **memorija tekstura.** Memorija tekstura predstavlja dio globalne memorije koja osim što omogućuje čitanje, koristi se i za mapiranje određenog skupa podataka.



Slika 3: Prikazuje CUDA memorije

Prilikom kreiranja CUDA programa vrlo je važno da poznajemo raspoložive tipove memorije. Možemo uočiti da je globalna memorija znatno sporija gledano s aspekta pristupa, stoga bi ju trebalo izbjegavati. Navedimo tipičan postupak u CUDA programima kako treba izgledati.

- raspodijeliti problem na potprobleme,
- podijeliti ulazne podatke na skupove podataka koji se mogu spremiti u registre i dijeljivu memoriju,
- učitati jedan skup podataka iz globalne memorije u refistre ili dijeljenu memoriju,
- Obraditi skup podataka pomoću niti
- spremiti kopiju rezultata u globalnu memoriju.

4.4 PREDNOSTI CUDA-E

CUDA ima nekoliko prednosti nad standardnim pristupom računanju uz pomoć računala u odnosu na GPU [8], a to su:

- raštrkano čitanje - informacije se pribavljaju sa raznovrsnih memorijskih adresa na jednom mjesto,
- jedinstvena virtualna memorija,
- jedinstvena memorija,
- dijeljiva memorija,
- brže skidanje podataka u odnosu na GPU,
- potpuna podrška za bitne i matematičke operatore.

4.5 OGRANIČENJA CUDA-E

CUDA ima i nekoliko ograničenja,a to su:

- čitav kod u CUDA-i se treba napraviti prema sintaksnim pravilima kao u C++ programu, jer je CUDA-in compiler dizajniran tako da prepoznaje C++ sintaksu,
- kopiranje između memorije domaćina i uređaja može uzrokovati smanjene prerformanse rada zbog veličine internetskog pojasa,
- informacije za pokretanje i stilovi izuzetaka u C++ su jedino prihvaćeni na domaćinu, ali ne i na uređaju,
- ispravan C++ kod u nekim slučajevima može onemogućiti pravilno prevođenje koda, razlog tome je što prevoditelj pristupa optimaziciji zbog ograničenja na GPU uređaju.

U prvoj generaciji CUDA-e problem su bili su brojevi u znanstvenom zapisu [9] koji su bili zapisani kao nule te je preciznost dijeljenja i korijenovanja bila znatno manja nego što je kasnije propisano u **IEEE 754** za brojeve s pomičnom točkom.
CUDA se također koristi i u bioinformatici, kriptografiji i ostalim područjima znanosti, ali i u računalnim igricama za kreiranje različitih fizikalnih efekata kao što su vatra i voda. CUDA radi sa svim Nvidia GPU-ovima od G8x serije pa na dalje, uključujući GeForece, Quadro i Tesla linije procesora.

[8] GPU je grafička procesna jedinica koju računalo koristi za matematičke izračune, obradu slika i slično.

[9] Primjer za znanstveni zapis: $1.23 \cdot 10^{-2}$ bio bi zapis broja 0.0123

Uz podršku za CCC, C++, Fortran CUDA podržava i ostale programske jezike kao što su na primjer Haskell, Ruby, Java, R, Perl, Lua i slični.

CUDA ima veliku važnost zbog toga jer se koristi prilikom izrade različitih programa za duboko učenje te za revoluciju na području umjetne inteligencije.

4.6 Pokretanje obrade

Za razliku od zajedničke memorije na CUDA-i koja donosi jednu memorijsku adresu za sve GPU i CPU procesore na računalu, za računanje na GPU potrebno je alocirati memoriju koja će biti dostupna od strane GPU-a.

Za alokaciju memorije koristi se funkcija **cudaMallocManaged()** koja vraća pokazivač koji može pristupiti domaćinu ili uređaju. Za oslobađanje podataka poziva se funkcija **cudaFree()**. Slijedi primjer koda u CUDA-i za alokaciju memorije.

```
// Allocate Unified Memory -- accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

...

// Free memory
cudaFree(x);
cudaFree(y);
```

Slika 4: Prikazuje Alokaciju memorije

Prethodna slika prikazuje kako se može alocirati memorija uz pomoć CUDA-e. Kod je napisan u C++ programu. Za zbroj dva broja koriste se zagrade u obliku trostrukih strelica ispred kojih se mora dodati funkcija za zbroj.

Da bi se kod mogao pokretati potrebno je u Python-u koristiti modul PyCUDA. U modulu PyCUDA postojeposebne funkcije koje se koriste prilikom paralelnog programiranja. Da bi se moglo paralelno programirati potrebno je dodati **SourceModul** iz biblioteke **pycuda.compiler** . On omogućava čitanje C++ koda iz datoteke koja ima ekstenziju **.cu**. Trebamo putem SourceModula pročitati danu CUDA datoteku. Nakon toga se pozove funkcija koja je definirana u **.cu** modulu i u Pythonu se inicijaliziraju potrebni podaci za obradu. Nakon toga definira se rešetka preko ključnih riječi *block* i *grid* koja prihvaća podatke i izračunava ih posebno na bloku, a posebno na gridu.Važan princip koji CUDA koristi je princip **"zrna"**. Datoteka s ekstenzijom **.cu** se naziva **zrno**. Zrna su funkcije koje se definiraju u

C++ programu i koje su potom pozvane od strane Pythona kako bi se izvele nad podacima koji su im dani.

Primjer 2.[Zbrajanje dva vektora uz pomoć modula CUDA-e] Da bismo mogli zbrojiti dva vektora potrebno je prvo uključiti potrebne module u program. Potom je potrebno pročitati potrebnu CUDA datoteku na način prikazan na sljedećoj slici.

```
mod = SourceModule(open("zbroj.cu").read())

zbroj_vektora = mod.get_function("zbroj_vektora")
```

Slika 5: Prikazuje zbroj vektora pomoću CUDA-e

Ovaj kod iz CUDA datoteke pročita potrebnu funkciju za zbrajanje vektora te potom Python kod primjenjuje tu funkciju na dva definirana vektora $\vec{a}$ i $\vec{b}$ koja su definirana na sljedeći način:

```
a = np.ones(400, dtype=np.float32)
b = np.ones(400, dtype=np.float32)

result_gpu = np.zeros_like(a)
```

Slika 6: Prikazuje definiranje vektora u Pythonu

Način na koji se primjenjuje funkcija zbroja vektora prikazano je na sljedećoj slici.

```
zbroj_vektora(drv.Out(result_gpu), drv.In(a), drv.In(b), block=(400,1,1), grid=(1,1))
```

Slika 7: Prikazuje pozivanje funkcije zbroja vektora za dane parametre

Funkcije **drv.In(), drv.Out()** i **drv.InOut()** koriste se da bi se zrna pretvorila u numpy polja s kojima Python može raditi.

- Funkcija **drv.In()** služi za inicijalizaciju polja koje treba biti kopirano prije uporabe zrna.
- Funkcija **drv.Out()** služi za inicijalizaciju polja koje treba biti kopirano nakon uporabe zrna.
- Funkcija **drv.InOut()** služi za inicijalizaciju polja koje treba biti kopirano i prije i nakon uporabe zrna.

5 Numeričke metode

5.1 Aproksimacija

Postavlja se pitanje što je aproksimacija i koje se činjenice uz nju vežu.

Neka su nam poznate određene informacije i funkciji g koja je definirana na skupu X. Ako je nama ta funkcija g nepoznata tada ju želimo odrediti pomoću neke druge funkcije koju smo sami zadali. Nepoznatu funkciju g određujemo pomoću vlastite funkcije na način da gledamo koliko je naša funkcija bliska nepoznatoj funkciji. Možemo imati dva slučaja na koji je način zadana naša nepoznata funkcija. Problemi na koje se dolazi prilikom aproksimacije mogu biti sljedeći:

1. Znamo funkciju analitički, ali je njezina forma prekomplicirana da bismo ju upotrebljavali. U tom slučaju pristupamo biranju vlastite funkcije uz neki kriterij te pomoću nje aproksimiramo zadanu funkciju. Tada možemo birati određene informacije koje ćemo koristiti prilikom aproksimacije funkcije.

2. Postoji slučaj u kojemu ne znamo funkciju f analitički, ali znamo određene informacije o njoj.Koristeći te informacije pokušavamo odrediti vlastitu funkciju koja će uz zadovoljavajuću grešku funkcije [10].

U praksi se veoma rijetko događa prvi slučaj. Najčešće su nam poznate diskretne vrijednosti određene funkcije i to se dosta često javlja kod mjerenja nekih veličina. Zbog loših mjernih uređaja podaci koje dobivamo su lošije kvalitete te se stoga događaju velika odstupanja od stvarnih veličina. Koristeći posebne tehnike moguće je smanjiti postojeću grešku. U početku se koriste tehnike koje pokrivaju čitavu domenu problema. Međutim te tehnike stvaraju velike greške prilikom korištenja. Zbog toga se domena dijeli na podsemente te se koriste tehnike koje omogućuju aproksimaciju podsegmenata domene.

5.1.1 Interpolacija

Interpolacija je zahtjev da se vrijednsot ili neka derivacija funkcija f i g podudara na konačnom skupu točaka interpolacije.

Napomena 5.1.1. *Problem interplacije može se smatrati specijalnim, ali posebno važnim slučajem aproksimacije po normi na diskretnom skupu točaka interpolacije.*

Oblik interpolacijemože biti zadan na dva načina. Prvi način je da su zadane samo funkcijske vrijednosti, dok je drugi način takav da su uz funkcijske vrijenosti zadane i derivacije.
Prvi način se još naziva **Lagrangeova interpolacija**, dok se drugi način naziva **Hermitova interpolacija.**

Teorem 5.1.1. *Neka je $n \in \mathbb{N}_0$. Za zadane točke (x_i, y_i), $i = 0, \dots n$, gdje je $x_0 < \cdots < x_n$ postoji jedinstveni interpolacijski polinom $p \in P_n$, stupnja najviše n tako da vrijedi $p(x_i) = y_i$, $i = 0 \dots n$.*

[10] Greška funkcije je omjer između funkcije koja nam je zadana i funkcije koju aproksimiramo

5.2 Linearni splajn

U praksi se rijetko koriste polinomi vrlo visokog stupnja.Umjesto jednog polinoma visokog stupnja često se koristi više polinoma nižeg stupnja kojima aproksimiramo funkciju. Kako bismo odredili funkciju ϕ potrebno je odrediti $(m+1)n$ koeficijenata. Za funkciju ϕ postavit ćemo dodatne uvjete iz kojih će se odrediti koeficijenti.

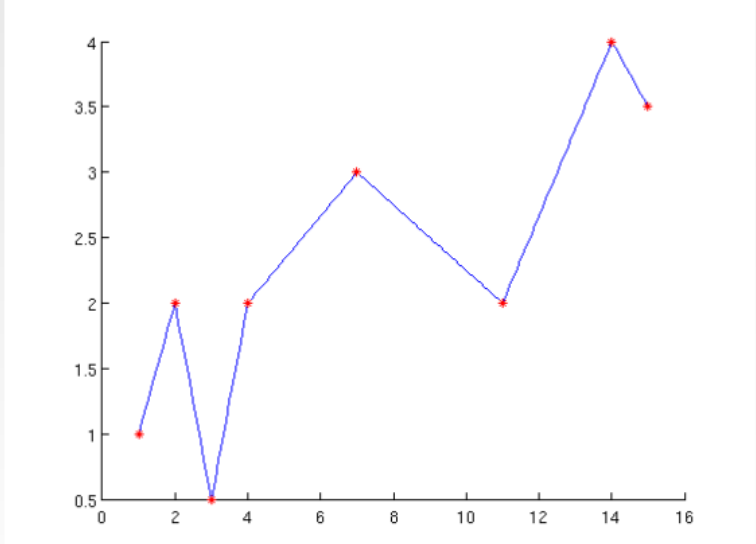
Iz uvjeta interpolacije

$$\phi(x_i) = y_i, \qquad i = 0, \ldots, n$$

dobijemo po dva uvjeta za svaku točku

$$\begin{aligned} p_0(x_0) &= y_0, \\ p_{i-1}(x_i) &= y_i, \qquad i = 1, \ldots, n-1 \\ p_{n-1}(x_n) &= y_n \end{aligned}$$

odnosno $2n$ uvjeta kojima se osigurava neprekidnost funkcije $phi(x)$. Ako odaberemo da su polinomi $p_i(x)$ inP_1 (polinom prvog stupnja) tada imamo dovoljno uvjeta iz kojih možemo odrediti sve koeficijente linearnog interpolacijskog splajna.



Slika 8: Prikazuje grafički prikaz linearnog splajna

Svakom intervalu $[x_i, x_{i+1}]$ pridružimo jedinstveni polinom $p_i(x)$ kojeg jednostavno zapisujemo u obliku Lagrangeova polinom

$$p_i(x) = \frac{x - x_{i+1}}{x_{i+1} - x_i} y_i + \frac{x - x_i}{x_{i+1} - x_i} y_{i+1}$$

Polinom možemo zapisati i u obliku Newtonovog interpolacijskog polinoma

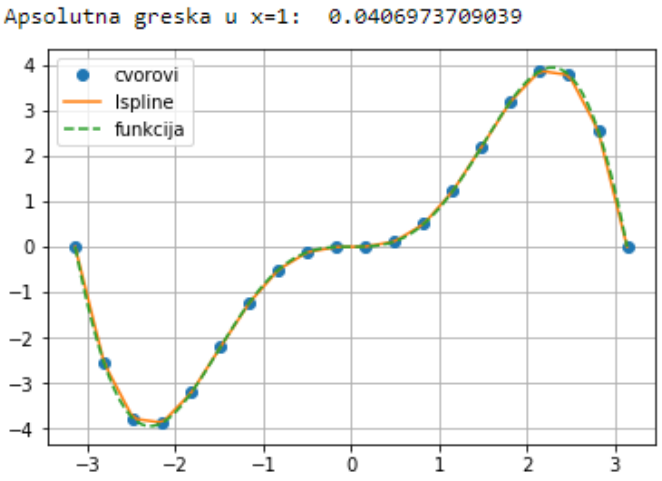
$$p_i(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i).$$

Launcher | SplineZadatak2 | SplineZadatak1

```
#Linearnim splineom aproksimirajte funkciju x**2*sinx na intervalu [-pi,pi].
#Koristite ekvidistantnu mrezu od 20 numerickih cvorova.
#Izracunajte apsolutnu gresku aproksimacije u cvoru x = 1.

from numpy import *
from scipy import interpolate
import pylab as pl

def f(x):
    return (x**2)*sin(x)

xcvorovi=linspace(-pi,pi,20)
ycvorovi=f(xcvorovi)

lspline = interpolate.interp1d(xcvorovi, ycvorovi)

print 'Apsolutna greska u x=1: ',abs(f(1)-lspline(1))

#Grafovi
xtocke=linspace(min(xcvorovi),max(xcvorovi),100)
yspline = lspline(xtocke)
yfunkcija = f(xtocke)

pl.figure()
pl.plot(xcvorovi,ycvorovi,'o',xtocke,yspline,'-',xtocke,yfunkcija,'--')
pl.legend(['cvorovi', 'lspline', 'funkcija'])
pl.grid()
pl.show()
```

Slika 9: Prikazuje python kod linearnog splajna

Sljedeća slika prikazuje ispis navedenog prevedenog koda



Slika 10: Prikazuje python ispis linearnog splajna

5.3 Kubični splajn

Ako želimo dobiti veću glatkoću i brzu konvergenciju od linearnog splajna možemo koristiti polinome višeg stupnja. Na svakom intervalu koristit će se polinomi

$$\phi|_{[x_i,x_{i+1}]} = p_i(x) \qquad i = 0, \ldots, n,$$

pri čemu je $p_i(x) \in P_3$.
Kubični polinom $p_i \in P_3$ ima četiri koeficijenta i možemo ga zapisati u obliku

$$p_i(x) = a_0 + a_1(x - x_i) + a_2(x - x_i)^2 + a_3(x - x_i)^3.$$

Moramo zadati dovoljno uvjeta da se odredi $4n$ koeficijenata kubičnog splajna.

Uvjeti:
Prvi uvjet:

$$\begin{aligned} p_0(x_0) &= y_0, \\ p_{i-1}(x_i) &= y_i, \qquad i = 1, \ldots, n-1 \\ p_{n-1}(x_n) &= y_n, \end{aligned}$$

Drugi uvjet

$$p_i^{'}(x_{i+1}) = p_{i+1}^{'}(x_{i+1}), \qquad i = 0, \ldots, n-2$$

Treći uvjet

$$p_i^{''}(x_{i+1}) = p_{i+1}^{''}(x_{i+1}), \qquad i = 0, \ldots, n-2$$

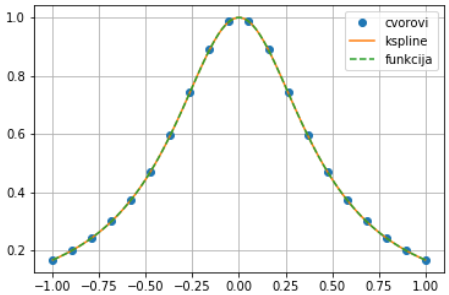
Prvi uvjet osigurava neprekidnost interpolacije i daje $2n$ uvjeta.
Drugi uvjet osigurava neprekidnost prve derivacije na unutarnim točkama interpolacije i daje $n - 1$ uvjeta.
Treći uvjet osigurava neprekidnost druge derivacije na unutarnjim točkama interpolacije i daje dodatnih $n - 1$ uvjeta.

Untitled1.ipynb | SplineZadatak2 | SplineZadatak1

```
#Kubicnim splineom aproksimirajte funkciju 1/(1+5*x**2) na intervalu [-1,1]
#upotrebom 20 ekvidistantnih cvorova. Izracunajte relativnu gresku u cvoru 0.95.

from numpy import *
from scipy import interpolate
import pylab as pl

def f(x):
    return 1/(1+5*x**2)

xcvorovi=linspace(-1,1,20)
ycvorovi=f(xcvorovi)

kspline = interpolate.interp1d(xcvorovi, ycvorovi, kind='cubic')

print 'Relativna greska u x=0.95: ',abs(f(0.95)-kspline(0.95))/abs(f(0.95))

#Grafovi
xtocke=linspace(min(xcvorovi),max(xcvorovi),100)
yspline = kspline(xtocke)
yfunkcija = f(xtocke)

pl.figure()
pl.plot(xcvorovi,ycvorovi,'o',xtocke,yspline,'-',xtocke,yfunkcija,'--')
pl.legend(['cvorovi', 'kspline', 'funkcija'])
pl.grid()
pl.show()
```

Slika 11: Prikazuje python kod kubičnog splajna

```
Relativna greska u x=0.95:  0.000143533561141
```



Slika 12: Prikazuje python ispis kubičnog splajna

5.4 Newton-Cotesove formule

Sada ćemo obraditi Newton -Cotesove formule, odnosno dotaknut ćemo se trapezne i Simpsonove metode.

Treba izračunati integral $\int_a^b f(x)dx$ ako je zadano sljedeće:

$h = \frac{b-a}{n}$, a točke su $x_0 = a, x_1 = a + h, \ldots, x_i = a + ih, \ldots x_n = b$, odnosno segment na kojemu integriramo podijeljen je na ekvidistantnim točkama. Odredimo

$$f(x_i) = y_i \qquad i = 0, \ldots, n.$$

Primjenimo istu formulu $\int_a^b f(x)dx \approx \sum_{i=0}^{n} A_i y_i$ pri čemu je

$$A_i = \int_a^b \frac{\Pi_{n+1}(x)}{(x - x_i)\Pi'_{n+1}(x_i)}dx,$$

gdje je $\Pi_{n+1}(x) = (x - x_0)(x - x_1)\dots(x - x_n)$. Uvodimo oznaku $q = \frac{x-x_0}{h}$.
Tada $\Pi_{n+1}(x)$ postaje:

$$\Pi_{n+1}(x) = qh(q-1)h(q-2)h\cdots(q-n)h$$

$$\Pi_{n+1}(x) = h^{n+1}q(q-1)\cdots(q-n),$$

$x - x_i = x - x_0 - ih = (q - i)h$.

Sada imamo:

$$\Pi'_{n+1}(x) = (x_i - x_0)(x_i - x_1)\cdots(x_i - x_{i-1})(x_i - x_{i+1})\cdots(x_i - x_n),$$

odnosno,

$$\Pi'_{n+1}(x) = i! \cdot h^i \cdot (-1)^{n-i} \cdot h^{n-i} \cdot (n-i)!$$

i $dx = hdq$. Uočimo da sređivanjem izraza dolazimo do konačne formule za određivanje koeficijenta $A_i, \qquad i = 0, \dots, n$:

$$A_i = \frac{(-1)^{n-i}h}{i!(n-i)!}\int_a^b \frac{q(q-1)\cdots q(n)}{q-i}dq.$$

Kako je $h = \frac{b-a}{n}$ tada iz formule za određivanje koeficijenta $A_i, \qquad i = 0, \dots, n$ dolazimo do zaključka da se A_i može pisati kao $A_i = (b-a)H_i$, gdje je

$$H_i = \frac{(-1)^{n-i}}{ni!(n-i)!}\int_0^n \frac{q(q-1)\cdots q(n)}{q-i}dq.$$

Koristeći prethodno navedene formule dobivamo da je

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} H_i y_i.$$

Ako u prethodnoj formuli uzmemo da je $n = 1$ tada dobivamo:

$$\int_{x_0}^{x_1} f(x)dx \approx h[H_0 y_0 + H_1 y_1],$$

pri čemu je

$$H_0 = \frac{(-1)^1}{1 \cdot 0! \cdot 1!}\int_0^1 \frac{q(q-1)}{q}dq,$$

$H_0 = -\int_0^1 (q-1)dq$, a time smo dobili $H_0 = \frac{1}{2}$. Nadalje, $H_1 = \frac{-(1)^0}{1\cdot 1!\cdot 0!}\int_0^1 \frac{q(q-1)}{q-1}dq = \frac{1}{2}$.

Time dobivamo da je $\int x_0 x_1 f(x)dx$ moguće izračunati kao:

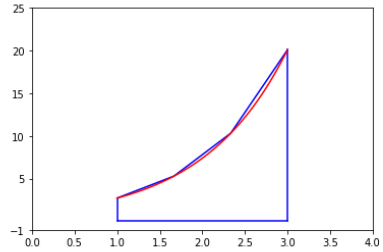
$$\int_{x_0}^{x_1} f(x)dx \approx \frac{h}{2}[y_0 + y_1].$$

Navedena formula naziva se **TRAPEZNA FORMULA.**

```
from matplotlib import *
from pylab import *
def trapezna(a,b,uy):
    tocke=uy.size
    suma=uy[0]+uy[-1]
    for i in np.arange(tocke-2):
           suma+=2*uy[i+1]
    plot(np.linspace(a,b,tocke),uy,'b')
    plot([a,a],[0,uy[0]],'b')
    plot([b,b],[0,uy[-1]],'b')
    plot([a,b],[0,0],'b')
    return (b-a)/(tocke-1)*suma/2
print('Aproksimacija površine:',trapezna(1,3,np.exp(np.linspace(1,3,4))))
print('Točna površina:',np.exp(3)-np.exp(1))
xlim(0,4)
ylim(-1,25)
yticks([-1,5,10,15,20,25])
plot(np.linspace(1,3),np.exp(np.linspace(1,3)),'r')
show()
```

```
Aproksimacija površine: 18.0057719517
Točna površina: 17.3672550947
```

Slika 13: Prikazuje python kod trapezne formule



Slika 14: Prikazuje python ispis za grafički prikaz trapezne formule

Sada želimo primjeniti trapeznu formulu na funkciju za koju smo uzeli više točaka $x_0, \ldots, x_n$. Tražimo $\int_{x_0}^{x_n} f(x)dx$.

Počinjemo sa

$$\int_{x_0}^{x_1} f(x)dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)dx,$$

primjenimo trapeznu formulu i sada imamo:

$$\int_{x_0}^{x_n} f(x)dx \approx \frac{h}{2}[y_0 + y_1] + \cdots + \frac{h}{2}[y_{n-1} + y_n],$$

$$\int_{x_0}^{x_n} f(x)dx = \frac{h}{2}[y_0 + y_n + 2 \cdot (y_1 + y_2 + \cdots + y_{n-1})].$$

Navedena formula naziva se **GENERALIZIRANA TRAPEZNA FORMULA.**

Primjenimo Newton-Cotesove formule $A_i = (b-a)H_i$, gdje je

$$H_i = \frac{(-1)^{n-i}}{ni!(n-i)!}\int_0^n \frac{q(q-1)\cdots q(n)}{q-i}dq,$$

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} H_i y_i,$$

uzimajući da je $n = 2$. Tada je

$$\int_{x_0}^{x_2} f(x)dx \approx 2h[H_0y_0 + H_1y_1 + H_2y_2].$$

Ako promatramo za $n = 2$ imamo:

$$H_0 = \frac{(-1)^2}{2\cdot 0!\cdot 2!}\int_0^2 \frac{q(q-1)(q-2)}{q}dq = \frac{1}{6},$$

$$H_1 = \frac{(-1)^1}{2\cdot 1!\cdot 1!}\int_0^2 \frac{q(q-1)(q-2)}{(q-1)}dq = \frac{2}{3},$$

$$H_2 = \frac{(-1)^0}{2\cdot 2!\cdot 0!}\int_0^2 \frac{q(q-1)(q-2)}{(q-2)}dq = \frac{1}{6}.$$

Time formula

$$\int_{x_0}^{x_2} f(x)dx \approx 2h[H_0y_0 + H_1y_1 + H_2y_2]$$

postaje

$$\int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}[y_0 + 4y_1 + y_2]$$

i ta se formula naziva **SIMPSONOVA FORMULA.**

Geometrijsko tumačenje. Geometrijski desna strana u formuli

$$\int_{x_0}^{x_2} f(x)dx \approx \frac{h}{3}[y_0 + 4y_1 + y_2]$$

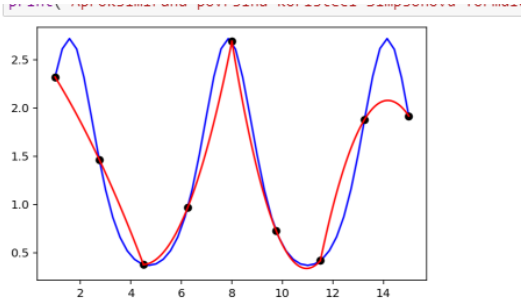
predstavlja površinu lika koji je ograničen parabolom koja prolazi kroz točke

$$(x_0, y_0), (x_1, y_1), (x_2, y_2).$$

```
def simpsonova(a,b,uy):
    tocke=uy.size
    suma=0
    for i in np.arange(tocke):
        if i==0 or i==tocke-1:
            suma+=uy[i]
        elif i%2==0:
            suma+=2*uy[i]
        else:
                suma+=4*uy[i]
        return (b-a)/(tocke-1)*suma/3
print('Aproksimacija površine-trapeznom:',trapezna(1,3,np.exp(np.linspace(1,3,5))))
print('Aproksimacija površine-Simpsonovom:',simpsonova(1,3,np.exp(np.linspace(1,3,5))))
print('Točna površina:',np.exp(3)-np.exp(1))
```

```
Aproksimacija površine-Simpsonovom: 0.45304697141
Točna površina: 17.3672550947
```

Slika 15: Prikazuje python kod Simpsonove formule



Slika 16: Prikazuje python ispis za grafički prikaz Simpsonove formule

```
%%time
from matplotlib import *
from pylab import *
def trapezna(a,b,uy):
    tocke=uy.size
    suma=uy[0]+uy[-1]
    for i in np.arange(tocke-2):
        suma+=2*uy[i+1]
    plot(np.linspace(a,b,tocke),uy,'b')
    plot([a,a],[0,uy[0]],'b')
    plot([b,b],[0,uy[-1]],'b')
    plot([a,b],[0,0],'b')
    return (b-a)/(tocke-1)*suma/2
print(f'Aproksimacija površine: {trapezna(1,3,np.exp(np.linspace(1,3,4)))}')
print(f'Točna površina: {np.exp(3)-np.exp(1)}')
xlim(0,4)
ylim(-1,25)
yticks([-1,5,10,15,20,25])
plot(np.linspace(1,3),np.exp(np.linspace(1,3)),'r');show()
```

Slika 17: Prikazuje python kod sa izračunom vremena za trapeznu formulu

6 Rezultati izvođenja metoda na CPU i CUDA

U ovom djelu ću prikazati brzinu izvođenja koda za trapeznu i Simpsonovu formulu na CPU-u te na grafičkoj kartici. Kod izbora koja će se numerička metoda koristiti nije bitna samo njena složenost nego i njena brzina izvođenja na računalu. Brzina izvođenja uvelike ovisi o načinu programiranja koda bilo da kod izvodite sekvencijonalno bilo da se izvodi paralelno. U nastavku ću prikazati kolika je brzina izvođenja koda na CPU-u za trapeznu formulu. Sljedići kod prikazuje kod moderniziran na način da uz kod za trapeznu formulu računa i vrijeme izvođenja koda: Kao što se vidi iz samog koda koristio sam naredbu *time* za izračum vremena na CPU-u. Slijedi ispis vremena koje je potrebno za izvršavanje koda.

```
CPU times: user 176 ms, sys: 256 ms, total: 432 ms
Wall time: 125 ms
```

Slika 18: Prikazuje CPU vrijeme za trapeznu formulu

Da bi se smanjilo vrijeme izvođenja koda često se kod razdvaja na dva djela na način da se u Python stavljaju pozivi CUDA zrna, dok u CUDA-u stavlja osnova programa. Slijedeći kod prikazuje realizaciju trapezne formule uz pomoć CUDA-e. Obzirom da je koda dugačak odlučio sam ga razbiti u više slika. Na prvoj slici se prikazuje koja funkcija se računa te inicijalizacija kernela.

```
#include<iostreme>
#include<ctime>
using namespace std;

#include<cuda.h>
#include<math_constants.h>
#include<cuda_runtime.h>


__device__ float myfunc(float a){
return (b-a)/( f(b)+f(a)/3 )
}
__global__ __device__ void Kernel( float *a, float c, float dx, int n){
        int idx=blockIdx.x*blockDim.x+threadIdx.x;
        if(idx<n){
                a[idx]=myfunc(x)+myfunc(x+dx);
        }
}
```

Slika 19: Prikazuje prvi dio CUDA koda za trapeznu formulu

Na drugoj slici je glavni dio koda koji računa trapeznu formulu na domaćinu definirajući broj i veličinu blokova. Također kod definira postupak oporavka od

greški koje se mogu dogoditi prilikon računanja trapezne formule. Na kraju se izvodi zbrajanje te oslobađanje memorije potrebne za izvođenje koda.

```
__host__ float Integrated(float a, float d, int n){
        float dx=(d-c)/n;

        cudaError_t errorcode=cudaSuccess;

        int size=n*sizeof(float);
        float* ah=(float *)malloc(size);

        float* ad;
        if(( errorcode =codaMalloc((void **)&ad,size))!=cudaSuccess){
                cout<<"cudaMalloc(): "<< cudaGetErrorString(errorcode) <<endl;
                exit(1);
        }
        int block_size=256;
        int n_block=n / block_size +(n % block_size==0 ? 0:1);
        Kernel <<<n_block, block_size >>> (ad,c,dx,n)

        if((errorcode= cudaMemcpy(ah,ad,sizeof(float)*n, cudaMemcpyDeviceToHost))!=cudaSucces){
        cout<<"cudaMemcpy()" << cudaGetErrorString(errorcode)<<endl;
        exit(1);
        }


        float sum=0.0;
        for(int i=0;i<n;i++)
              sum+=ah[i];
        sum*=dx/2.0;
        free(ah);
        cudaFree(ad);
        return sum;
}
```

Slika 20: Prikazuje drugi dio CUDA koda za trapeznu formulu

Na trećoj slici se prvo definira funkcija $diffclk$ koja računa vrijeme potrebno da se izvede CUDA kod za trapeznu formulu koja koristi razlomak $\frac{CLOCKS-PER-SEC}{1000}$ da bi mjerna jedinica bila milisekunde budući da je funkcija $CLOCKS-PER-SEC$ definirana u sekundama. U glavnom programu se prvo defirnira početak mjerenja vremena potom se poziva funkcija za trapeznu formulu te nakon njenog izvršavanja se zaustavlja vrijeme. Na kraju se pomoću funkcije $diffclk$ izračunava ukupno vrijeme trajanja programa pomoću parametara *start* koje definira vrijeme na početku izvođenja te *end* koje definira vrijeme na kraju izvođenja koda.

U nastavku je prikazan ispis vremena kod CUDA-e.

```
__host__ double diffclk(clock_t ck1, clock_t ck2){
        double difftick=ck1-ck2;
        double diffms=difftick/(CLOCKS_PER_SEC/1000);
        return diffms;
}

int main()[
        clock_t start=clock();
        float ans=Integrated(0.0,1.0,1000)
        clock_t end=clock();


        cout<<"Odgovor je"<<ans<<endl;
        cout<<"Vrijeme koje je potrebno za izračun"<<diffclk(end,start)<<"mikro sekunde"<<endl;

        return 0;
}
```

Slika 21: Prikazuje treći dio CUDA koda za trapeznu formulu

Slika 22: Prikazuje vrijeme Izvođenja na CUDA-i

6.1 Usporedba izvođenja na CUDA-i i CPU-u

Prilikom izvođenja CUDA programa primjetio sam značajnije ubrzanje u odnosu na kod izvođen na CPU-u. Razlog takvog rezultata je to što se python kod znatno sporije izvodi u odnosu na C i C++ kod. Način na koji se izvodi python izvodi je znatno drugačiji od C i C++ jer se python prvo šalje interpreteru koji ga interpretira i pretvara u kod koji se kompajlira. To zna oduzeti dosta vremena te se često python kod kombinira sa C i C++ kodom radi poboljšanja performansi. Način na koji se izvodi C i C++ kod bitno je drugačiji od pythona jer se kod nakon pokretanja odmah šalje kompajleru. To znatno utječe na perfomanse koda.

7 Zaključak

U ovom radu bavio sam se primjenom različitih numeričkih metoda i njihovom implementacijom u Pythonu. Tijekom rada obrađeni su linearni i kubični splajn te Newton Cotesove formule koje se odnose na trapeznu formulu, generaliziranu trapeznu formulu i Simpsonovu formulu.

U današnjoj znanosti i suvremenoj tehnologiji postoji široka primjena numeričkih metoda. U prošlosti su se koristile kod izgradnje brodova, dok se danas sve više povezuju sa informatičkom tehnologijom. Velika većina numeričkih metoda koristi se u strojnom učenju te u drugim prirodnim znanostima kao što su fizika, kemije i biologije. Da bi se mogao najbolje iskoristiti kapacitet koji nam omogućavaju numeričke metode potrebno je poznavati problem i mogućnosti numeričkih metoda prilikom njihova korištenja. Praktična realizacija pojedinih numeričkih metoda izvršava se uz pomoć Pythona. Koju ćemo metodu odlučiti za korštenje ovisi o njenoj matematičkoj složenosti kao i o brzini izvođenja na računalu.

Tijekom ovog rada sam obrađivao rad na CUDI. Danas je CUDA kao računalna arhitektura važan dio svakodnevne obrade podataka na superračunalima jer ona čini njihov temelj. Zahvaljujući CUDA-i programiranje u C++ dobiva na važnosti, budući da je u poslijednje vrijeme program C++ na neki način potisnut ako uspoređujemo koliko ljudi koristi programiranje u ostalim programima, kao što su Java i Python.

Veoma je važno u današnje vrijeme da različite aplikacije imaju brzo vrijeme izvođenja. Tu CUDA ima nezamijenjivu ulogu zahvaljujući mogućnosti paralelizma koji se sa njom može napraviti. Takav način rada je najiskorišteniji način. Da bi se moglo dobro programirati na CUDA-i potrebno je pažljivo pristupiti problemima koji zahtijevaju paralelnu obradu da bi se uspjele dobiti najbolje performanse koda.

Literatura

[1] Blanco Silva, F., *Learning SciPy for Numerical and Scientific Computung*, 2013.

[2] Bressert, E., *SciPy and NumPy Overview for Developers*, 2012.

[3] Idris, I., *NumPy Cookbook*, 2015.

[4] Johansson, R., *Numerical Python* 2015.

[5] Kiusalaas, J., *Numerical methods in Engineering with Python*, 2005.

[6] dd) Linge, S., Langtangen H.P., *Programming for Computation(Python)*, 2015.

[7] Prodanović, S. *Programiranje grafičkih uređaja: platforma CUDA*, Matematički fakultet, Sveučilište u Beogradu, 2012.

[8] Sergio J. Rojas G, Erik A. Christensen, Francisco J. Blanco Silva, *Learning SciPy for Numerical and Scientific Computing*,2015.

[9] Nagar, S., *Open source solutions for Numerical Computation*, 2018.

[10] Van der Plas, J., *Python Data Science Handbook*, 2016.

[11] Laboratoj za računalne mreže, paralelizaciju i simulaciju, URL:`https://lab.miletic.net/hr/nastava/materijali/python-modul-pycuda-zbrajanje-vektora/`
(pristup: 22.6.2018.)

Popis slika