Algoritmi s matricama koristeći CUDA tehnologiju

Lekšan, Dražen

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

ALGORITMI S MATRICAMA KORISTEĆI CUDA TEHNOLOGIJU

Diplomski rad

Dražen Lekšan

Osijek, 2023.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 12.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Dražen Lekšan
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1218R, 08.10.2021.
OIB studenta:	38044829335
Mentor:	izv. prof. dr. sc. Alfonzo Baumgartner
Sumentor:	,
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	izv. prof. dr. sc. Tomislav Keser
Član Povjerenstva 1:	izv. prof. dr. sc. Alfonzo Baumgartner
Član Povjerenstva 2:	doc. dr. sc. Tomislav Galba
Naslov diplomskog rada:	Algoritmi s matricama koristeći CUDA tehnologiju
Znanstvena grana diplomskog rada:	**Procesno računarstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	[Rezervirano: Dražen Lekšan]Koristeći procesore na grafičkoj kartici omogućiti izračun poznatijih algoritama s matricama poput množenja matrica, zbrajanja, računanja inverza matrice. Usporediti točnost i performanse ovih algoritama s onima koji se izvršavaju na CPU. Koristiti nVidia CUDA tehnologiju. Po mogućnosti ispitati na više modela grafičkih kartica.
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	12.09.2023.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTIRADA

Osijek, 06.10.2023.

Ime i prezime studenta:	Dražen Lekšan
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1218R, 08.10.2021.
Turnitin podudaranje [%]:	7

Ovom izjavom izjavljujem da je rad pod nazivom**: Algoritmi s matricama koristeći CUDA tehnologiju**

izrađen pod vodstvom mentora izv. prof. dr. sc. Alfonzo Baumgartner

i sumentora ,

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Algoritmi s matricama se često koriste za utvrđivanje ubrzanja izvođenja algoritma između serijske i paralelne verzije implementiranog algoritma. Paralelni algoritam se može izvoditi višenitno na centralnoj procesorskoj jedinici(CPU) i na grafičkoj procesorskoj jedinici(GPU). Razlika između CPU i GPU je u tome što GPU ima puno veći broj jezgri koje imaju manje performanse, ali su zbog svog velikog broja pogodnije za algoritme koji se mogu paralelizirati. U ovom radu je za izvođenje algoritama na GPU korištena CUDA tehnologija koja je razvijena od strane tvrtke NVIDIA. Programski jezici u kojima su najčešće napisani CUDA programi su C ili C++, a mogu se još koristiti i Fortran, Python te MATLAB. Implementirani su algoritmi zbrajanja, množenja i inverza matrice na serijski i paralelni način. Na navedenim algoritmima je provedeno testiranje brzine i točnosti izvođenja s različitim veličinama matrica.

CUDA tehnologija je opisana u drugom poglavlju. Ukratko je objašnjena CUDA arhitektura, navedeni se neki primjeri primjene CUDA tehnologije i opisan je razvoj tehnologije kroz povijest. U trećem poglavlju su objašnjeni korišteni algoritmi. Prvo su objašnjeni teorijski iz matematičkog gledišta te je potom prikazan način njihove programske implementacije u CUDA tehnologiji. U četvrtom poglavlju je testirana brzina izvođenja i točnost svakog algoritma. Napravljeno je više testnih slučajeva koristeći različite veličine matrice. Korišteno je više modela NVIDIA grafičkih kartica koje imaju različit broj CUDA jezgri i različitu brzinu i količinu memorije. Rezultati mjerenja su također grafički prikazani.

1.1. Zadatak diplomskog rada

Koristeći procesore na grafičkoj kartici omogućiti izračun poznatijih algoritama s matricama poput množenja matrica, zbrajanja, računanja inverza matrice. Usporediti točnost i performanse ovih algoritama s onima koji se izvršavaju na CPU. Koristiti NVIDIA CUDA tehnologiju. Po mogućnosti ispitati na više modela grafičkih kartica.

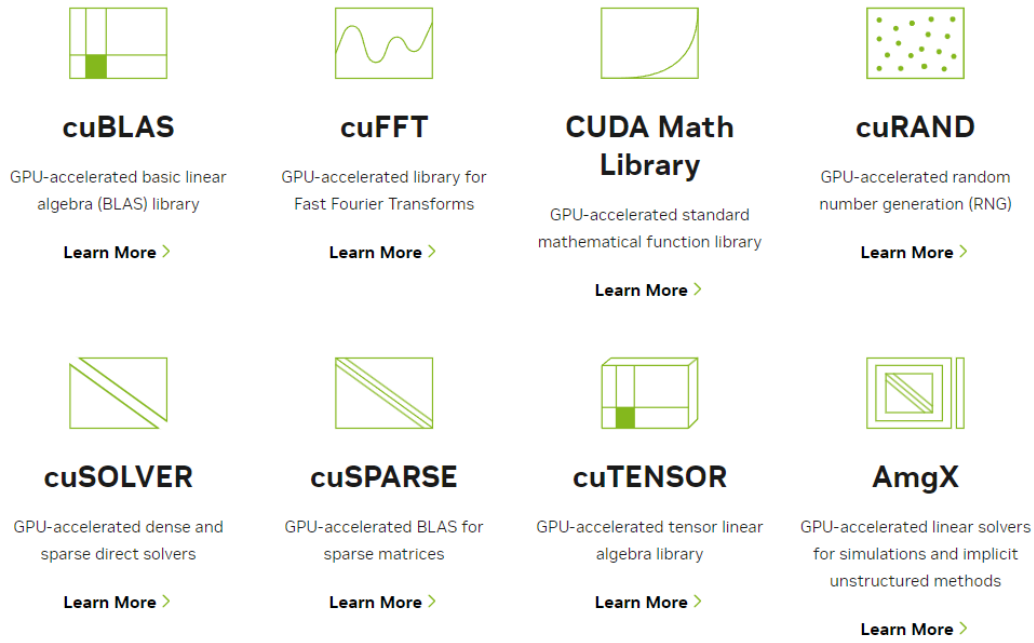
2. CUDA TEHNOLOGIJA

CUDA je paralelna računalna platforma i programski model koji je razvijen od strane tvrtke NVIDIA koji služi za izvođenje računalnih aplikacija na grafičkim procesorskim jedinicama (GPU) [1]. Iskorištavanjem velikog broja jezgri GPU-a omogućava znatno ubrzanje izvođenja računalnih aplikacija. U CUDA aplikaciji sekvencijalni dio programskog koda izvodi se na CPU-u koji je optimiziran za izvođenje na jednoj niti, a računalno zahtjevniji dio se izvodi na stotinama ili tisućama GPU jezgri paralelno [1]. CUDA aplikacije se programiraju u programskim jezicima C, C++, Fortran, Python i MATLAB. Paralelizam se postiže dodavanjem paralelnih regija u programski kod, koje su definirane kroz odgovarajuće ključne riječi. Za izradu CUDA aplikacije je potrebno imati instaliran CUDA *Toolkit.* CUDA *Toolkit* uključuje GPU-ubrzane biblioteke, kompajler, razvojne alate i CUDA *runtime* [1]. U ovom radu je korištena verzija CUDA *Toolkit-a* 11.6.0, dok je najnovija verzija 12.2.0.

NVIDIA CUDA-X je poboljšanje CUDA tehnologije i sadrži biblioteke, alate i tehnologije koje donose daleko bolje performanse u usporedbi s CPU alternativama. Biblioteke nude visoko optimizirane implementacije algoritama koji se stalno ažuriraju. Tipovi biblioteka su:

- Matematičke biblioteke
- Biblioteke za obradu slike i video zapisa
- Biblioteke za duboko učenje
- Biblioteke za paralelne algoritme
- Komunikacijske biblioteke

Na slici 2.1. su prikazane neke od matematičkih biblioteka.



Sl. 2.1. Prikaz CUDA biblioteka

2.1. Početci GPU programiranja

Pojava GPU-ova koji su imali programabilne cjevovode stvorila je mogućnost da se grafički hardver koristi za više stvari od jednostavnog grafičkog prikaza temeljenog na OpenGL-u ili DirectX-u. U početku su standardni grafički API-ji poput OpenGL i DirectX bili jedini način interakcije s GPU-om. Izračune opće namjene je jedino bilo moguće izvršiti kroz grafičke API-je na način da ih se pokuša prikazati GPU-u kao tradicionalne grafičke prikaze.

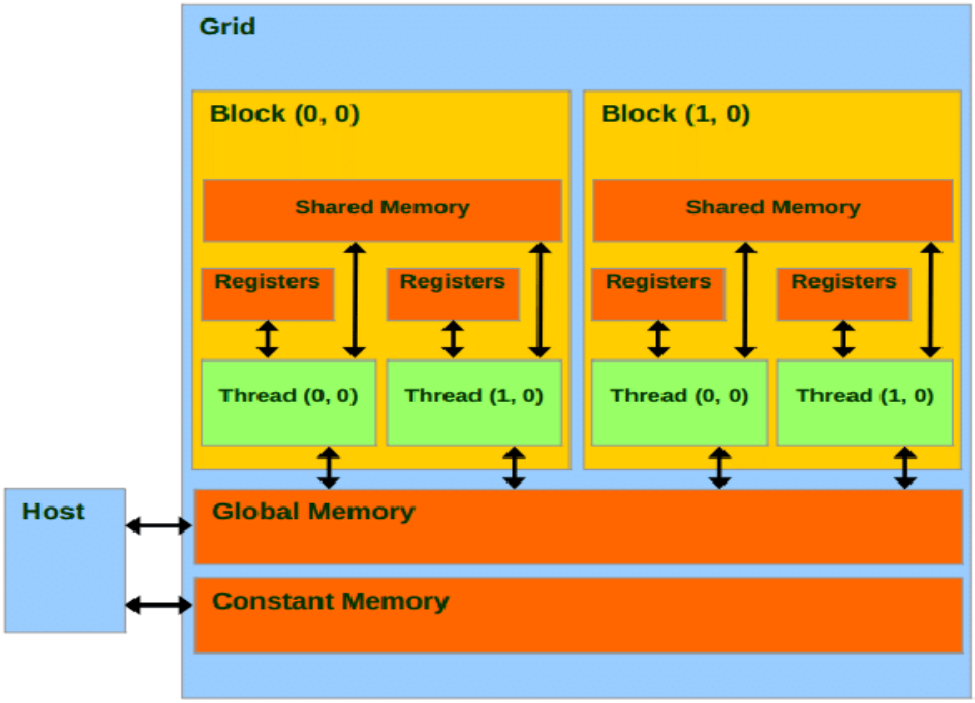
Kod operacija s pomičnim zarezom nije se moglo predvidjeti kako će se GPU ponašati ako je uopće i podržavala operacije s pomičnim zarezom. Također nije postojala nijedna dobra metoda otklanjanja neispravnosti u kodu koji se izvršavao na GPU. Ukoliko je netko, uz veći broj ograničenja, htio koristiti GPU za izračune opće namjene morao je prvo naučiti OpenGL ili DirectX. Uz to, izračuni opće namjene, uz zvanja funkcija OpenGL-a ili DirectX-a, morali su se pisati specijalnim grafičkim programskim jezicima koji su poznati kao *shading* jezici. Zbog potrebe za učenjem OpenGL-a i DirectX-a, korištenjem *shading* jezika i ograničenja kod izračuna opće namjene programiranje GPU-ova nije uspjelo postići širu primjenu.

2.2. CUDA arhitektura

CUDA arhitektura je postala dostupna 2006. godine kada je na tržište izašla NVIDIA GeForce 8800 GTX GPU koja je također prva podržavala DirectX verziju 10. Ova arhitektura uključuje nekoliko novih komponenti dizajniranih isključivo za izvođenje programskih instrukcija na GPU-u i uklanja ograničenja koja su imale prošle generacije grafičkih procesora kod računanja opće namjene.

Za razliku od prethodnih generacija koje su računalne resurse dijelile na *vertex* i *pixel shadere*, CUDA arhitektura uključivala je objedinjeni cjevovod *shadera*, dopuštajući da svaka aritmetičko-logička jedinica (ALU) na čipu bude raspoređena programom koji namjerava izvesti izračune opće namjene [2]. Sve aritmetičko-logičke jedinice su napravljenje u skladu sa zahtjevima IEEE organizacije za izvođenje aritmetičkih operacija s pomičnim zarezom jednostruke preciznosti i instrukcijski set im je prilagođen za izvršavanje općenitih računskih operacija, a ne samo za grafiku [2]. Izvršnim jedinicama na GPU-u dopušteno je proizvoljno čitanje i pisanje u memoriju, kao i pristup predmemoriji kojom upravlja softvera, a koja se još naziva zajednička memorija. Sve ove značajke CUDA arhitekture su dodane u GPU-ove kako bi se stvorila GPU koja bi, osim u grafičkim zadacima, bila efikasna i u izvršavanju općenitih računskih operacija.

Najveći blok memorije koji se koristi pri izvršavanju CUDA programa se zove *Grid* (mreža). Ona je podijeljena na blokove. Blok je podijeljen na više niti. Nit predstavlja osnovnu jedinicu rada koja se može izvršavati paralelno na GPU-u. Blok prikaz CUDA arhitekture prikazan je na slici 2.2.



Sl. 2.2. Blok prikaz CUDA arhitekture

2.3. Primjene CUDA tehnologije

Mnoge industrije su našle primjenu CUDA tehnologije u svojim djelatnostima. Ova tehnologija im je omogućila veliki skok u performansama u odnosu na prošlu korištenu tehnologiju. Aplikacije koje rade na NVIDIA grafičkim procesorima imaju vrhunske performanse po vatu i cjenovno su prihvatljivije od rješenja temeljenih na tradicionalnim procesnim tehnologijama [2].

Mnoge aplikacija su razvijene pomoću CUDA tehnologije i koriste se na ugradbenim sustavima, radnim stanicama, podatkovnim centrima i u oblaku računala. Slika 2.3. prikazuje neke od najpoznatijih aplikacija.

Sl. 2.3. Aplikacije razvijene pomoću CUDA tehnologije

U nastavku su opisani neki primjeri iz industrije na koje je uspješno primijenjena CUDA tehnologija.

2.3.1. Medicinske slike

Mamografija, jedna od trenutno najboljih tehnika za rano otkrivanje raka dojke, ima nekoliko značajnih ograničenja [2]. Potrebne su minimalno dvije snimke koje treba pregledati kvalificirani liječnik kako bi otkrio tumor. Osim toga, ovaj rendgenski postupak sa sobom nosi sve rizike opetovanog zračenja pacijenta [2]. Često su potrebne još i dodatne rendgenske slike kako bi se sa sigurnošću mogla utvrditi postojanost raka dojke. Cijeli postupak je skup i uzrokuje stres kod pacijenta.

Snimanje ultrazvukom je sigurnije od rendgena, pa se zato koristi zajedno s mamografijom kod dijagnosticiranja raka dojke. Konvencionalni ultrazvuk također ima svoje nedostatke i ograničenja. *TechniScan Medical Systems* je razvio trodimenzionalnu ultrazvučnu metodu snimanja kako bi otklonio nedostatke i ograničenja konvencionalnog ultrazvuka. Njegovo rješenje, međutim, nije se moglo primijeniti u praksi zbog ograničenja računalnog hardvera.

Predstavljanje NVIDIA-inog prvog GPU-a temeljenog na CUDA arhitekturi zajedno s njegovim programskim jezikom pružio je platformu na kojoj je *TechniScan* mogao realizirati svoje rješenje. *TechniScan-ov* sustav koristi dva NVIDIA Tesla C1060 procesora kako bi obradio 35 GB podataka koji su generirani snimanjem. U roku od dvadeset minuta je gotova trodimenzionalna slika ultrazvuka.

2.3.2. Računalna dinamika fluida

Dizajn efektivnih rotora i lopatica je godinama bio vrlo zahtjevan zadatak. Izrazito kompleksno gibanje zraka i fluida oko ovih uređaja ne može se modelirati pomoću jednostavnih jednadžbi. Točne simulacije su računski previše zahtjevne da bi se mogle izvoditi na običnim računalima. Za dobivanje točnog i preciznog rezultata su potrebna najveća superračunala u svijetu, koja mogu pružiti računalne resurse potrebne za rješavanje numeričkih modela neophodnih za uspješno dizajniranje rotora i lopatica.

Dolaskom CUDA arhitekture utvrđeno je da se prihvatljive performanse mogu dobiti koristeći osobne radne stanice koje koriste odgovarajući GPU. Korištenjem GPU klastera je moguće dobiti bolje performanse čak i od starijih superračunala koja su se prije koristila za računalne simulacije. CUDA arhitektura i GPU klasteri su zbog svoje dostupnosti, prihvatljive cijene i performansi iz temelja promijenili način istraživanja dinamike fluida.

2.3.3. Znanost o okolišu

U svijetu je sve veća potreba za ekološki prihvatljivom robom zbog klimatskih promjena, rastućih cijena goriva i rastuće razine zagađivača u zraku i vodi. Deterdženti i sredstva za čišćenje su među najpotrebnijim kućanskim potrepštinama ali ujedno mogu biti zagađivači okoliša. Zbog utjecaja na okoliš znanstvenici su počeli istraživati metode kako učiniti deterdžente ekološki prihvatljivim bez da se smanji njihova učinkovitost u čišćenju.

Ključne komponente sredstava za čišćenje poznate su kao površinski aktivne tvari, a još se zovu i surfaktanti. Molekule surfaktanta određuju kapacitet čišćenja i teksturu deterdženta i šampona ali se česti impliciraju kao ekološki najrazornija komponenta proizvoda za čišćenje [2]. Utvrđivanje efikasnosti čišćenja novog surfaktanta zahtijeva opsežna laboratorijska ispitivanja koja su vrlo spora i skupa.

Umjesto laboratorijskih ispitivanja moguće je koristiti računalnu molekularnu simulaciju. Uvođenje računalnih simulacija ubrzava vrijeme ispitivanja i proširuje opseg testiranja. Korištenjem dva NVIDIA Tesla GPU-a postignute su višestruke performanse u odnosu na starije računalne platforme na kojima su se izvršavale simulacije. Povećanjem broja GPU-ova je moguće još više ubrzati vrijeme simulacije. Budući da je NVIDIA-ina CUDA tehnologija drastično smanjila vrijeme izvođenja simulacija, može se i očekivati porast kvalitetnih sredstava za čišćenje koja u manjoj mjeri zagađuju okoliš.

3. ALGORITMI S MATRICAMA

U programskim jezicima je moguće implementirati razne algoritme koji koriste matrice. Algoritam koji obavlja istu operaciju s matricama je moguće implementirati na više različitih načina. Drugačijim načinom implementacije se mogu dobiti različite brzine izvođenja i različito zauzeće memorije. U ovom radu je implementirano zbrajanje, množenje i inverz matrice na dva različita načina. U nastavku poglavlja su detaljnije opisane matrice, navedeni algoritmi i način njihove programske implementacije.

3.1. Matrice

Matrica je pravokutna tablica sačinjena od nekoliko redaka i stupaca ispunjenih njenim elementima [3]. Ti su elementi obično brojevi, najčešće realni, no ponekad i kompleksni [3]. Postoji više tipova matrica koji se koriste u matematici i računarstvu, a neki od najčešćih su:

- Kvadratne matrice
- Nul-matrice
- Identitet matrice
- Dijagonalne matrice
- Jedinične matrice
- Gornjetrokutaste matrice
- Donjetrokutaste matrice

U ovom radu su korištene matrice samo s realnim brojevima. Zbog jednostavnosti izvođenja programskom algoritma korištene su matrice kvadratnog oblika. Kvadratna matrica ima jednak broj redaka i stupaca. Na slici 3.1. prikazan je primjer kvadratne matrice.

$$A = \begin{bmatrix} 3 & 4 & 5 \\ 7 & 8 & 6 \\ 1 & 2 & 0 \end{bmatrix}$$

Sl. 3.1. Primjer kvadratne matrice dimenzije 3x3

Također je za provjeru točnosti rezultata izvođenja algoritma korištena jedinična matrica. Jedinična matrica na glavnoj dijagonali ima sve jedinice, a na ostalim mjestima sve nule. Primjer jedinične matrice prikazan je na slici 3.2.

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Sl. 3.2. Primjer jedinične matrice dimenzije 3x3

3.2. Implementacije algoritama

Implementirani su algoritmi zbrajanja množenja i inverza matrice na paralelni i serijski način. Programski jezik u kojem su implementirani je C++. Paralelni algoritam se izvodi na GPU-u koristeći CUDA-inu biblioteku. Serijski algoritam se izvodi na CPU-u i koristi samo standardne C++ biblioteke.

Za korištenje memorija na GPU-u potrebno ju je prvo alocirati koristeći *cudaMalloc()* funkciju te potom u nju kopirati podatke na kojima se provode operacije koristeći *cudaMemcpy()* funkciju. Kod rada s memorijim *device* predstavlja GPU, a *host* CPU i njegovu pripadajuću memoriju. Slika 3.3. prikazuje alociranje i kopiranje memorija na *device* (GPU).

```
cudaMalloc((void**)&A_d, matrixSize * matrixSize * sizeof(int));
cudaMalloc((void**)&B_d, matrixSize * matrixSize * sizeof(int));
cudaMalloc((void**)&C_d, matrixSize * matrixSize * sizeof(int));


cudaMemcpy(A_d, A, matrixSize * matrixSize * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B, matrixSize * matrixSize * sizeof(int), cudaMemcpyHostToDevice);
```

Sl. 3.3. Primjer alociranja i kopiranja memorije na *device*

Za pozivanje CUDA funkcije je potrebno koristiti ključnu riječ *global* kod deklariranje funkcije. Funkcija se poziva unutar *main()* funkcije tako da joj se zadaju dimenzije mreže i bloka unutar šiljastih zagrada te se argumenti definirani u deklaraciji predaju u uglatim zagradama. Nakon izvršenja CUDA funkcije potrebno je kopirati memoriju nazad na *host.* Slika 3.4. prikazuje poziv CUDA funkcije i kopiranje memorije nazad na *host-a* (CPU).

```
unsigned int gridSize = (matrixSize + BLOCK_SIZE - 1) / BLOCK_SIZE;

dim3 dimGrid(gridSize, gridSize);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

gpu_matrix_mult << <dimGrid, dimBlock >> > (dA, dB, dC, matrixSize);

cudaMemcpy(hC, dC, sizeof(int) * matrixSize * matrixSize, cudaMemcpyDeviceToHost);
```

Sl. 3.4. Primjer poziva CUDA funkcije i kopiranje memorije nazad na *host-a*

Memorija korištena za algoritam može biti jednodimenzionalna, dvodimenzionalna i trodimenzionalna. U ovom radu je korištena samo dvodimenzionalna memorija. U tom slučaju blokovi i niti unutar mreže imaju svoj indeks po retcima i stupcima.

3.2.1. Zbrajanje matrica

Zbrajanje matrica definirano je na način

$$(\boldsymbol{A}+\boldsymbol{B})_{ij} = (\boldsymbol{A})_{ij} + (\boldsymbol{B})_{ij.} \tag{3-1}$$

Matrice *A* i *B* moraju imati isti broj redaka i stupaca da bi im zbroj mogao biti definiran. Rezultat zbrajanja je matrica istog tipa kao *A* i *B*. Element matrice *A*+*B* na mjestu *(i,j),* gdje *i* predstavlja broj retka, a *j* broj stupca, jednak je zbroju elemenata matrica *A* i *B* na istom tom mjestu. Primjer zbrajanja matrica prikazan je na slici 3.5.

$$A+B=\begin{bmatrix}2&2&1\\1&5&0\\0&0&1\end{bmatrix}+\begin{bmatrix}5&7&1\\0&3&0\\1&0&8\end{bmatrix}$$
$$=\begin{bmatrix}2+5&2+7&1+1\\1+0&5+3&0+0\\0+1&0+0&1+8\end{bmatrix}$$
$$=\begin{bmatrix}7&9&2\\1&8&0\\1&0&9\end{bmatrix}$$

Sl. 3.5. Primjer zbrajanja kvadratnih matrica dimenzija 3x3

Kod zbrajanja matrica vrijede iduća svojstva:

- Komutativnost: $A+B = B+A$
- Asocijativnost: $(A + B) + C = A + (B + C)$
- Neutralni element zbrajanja: $A + 0 = 0 + A = A$
- Suprotne matrice: $A + (-A) = 0$

3.2.2. Programska implementacija zbrajanja

Funkcija za zbrajanje matrica kao argumente prima pokazivače na vektore u koje su prethodno kopirane inicijalizirane matrice. Dva pokazivača sadrže matrica za zbrajanje, a treći je namijenjen za rezultat zbrajanja.

Zbrajanje matrica se provodi na način da se preko indeksa bloka nađe trenutna pozicija u retku i stupcu. Potom se dimenzija mreže po horizontalnoj osi pomnoži s trenutnom vrijednosti na vertikalnoj osi te se toj vrijednosti pribroji trenutna vrijednost na horizontalnoj osi. Na taj se način dvodimenzionalna matrica prikaže kao jednodimenzionalni vektor. Slika 3.6. prikazuje CUDA funkciju za zbrajanje matrica.

```
__global__ void matrixAddition_GPU(int* A, int* B, int* C) {

    int x = blockIdx.x;
    int y = blockIdx.y;
    int id = gridDim.x * y + x;
    C[id] = A[id] + B[id];

}
```

Sl. 3.6. CUDA funkcija za zbrajanje matrica

Provjera točnosti vrši se prolaskom kroz matrice koje su rezultat zbrajanja na CPU-u i na GPU-u te se provjerava jesu li elementi na istim indeksima jednaki. Ukoliko elementi nisu jednaki, vraća se *bool* vrijednost *false*, a ukoliko jesu petlja nastavlja s izvođenjem i kada petlja prođe kroz sve indekse matrice, bez da su prethodno pronađeni različiti elementi, vraća se vrijednost *true*. Slika 3.7. prikazuje funkciju za provjeru točnosti.

```
bool isValid()
{
    int row, col;
    for (row = 0; row < matrixSize; row++)
    {
        for (col = 0; col < matrixSize; col++)
        {
            if (C_GPU[row][col] != C_CPU[row][col]) {
                return false;
            }
        }
    }

    return true;
}
```

Sl. 3.7. Funkcija za provjeru točnosti

3.2.3. Množenje matrica

Da bi postojao umnožak dviju matrica, one moraju biti ulančane: broj stupaca prve mora biti jednak broju redaka druge matrice [3]. Rezultat množenja je matrica sa istim brojem redaka kao prva matrica i istim brojem stupaca kao druga matrica.

Zbrajanje matrica u općem obliku prikazano je sljedećom formulom [4]:

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{bmatrix} = \\ = \begin{bmatrix} a_{11} \cdot b_{11} + \cdots + a_{1n} \cdot b_{n1} & \cdots & a_{11} \cdot b_{1n} + \cdots + a_{1n} \cdot b_{nm} \\ \vdots & \ddots & \vdots \\ a_{m1} \cdot b_{11} + \cdots + a_{mn} \cdot b_{n1} & \cdots & a_{m1} \cdot b_{1n} + \cdots + a_{mn} \cdot b_{nm} \end{bmatrix} \tag{3-2}$$

Na slici 3.8. je prikazan primjer množenja matrica.

$$AB = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 0 & 3 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 2 \cdot 5 + 2 \cdot 0 + 1 \cdot 1 & 2 \cdot 0 + 2 \cdot 3 + 1 \cdot 0 \\ 1 \cdot 5 + 0 \cdot 0 + 0 \cdot 1 & 1 \cdot 0 + 0 \cdot 3 + 0 \cdot 0 \\ 0 \cdot 5 + 0 \cdot 0 + 1 \cdot 1 & 0 \cdot 0 + 0 \cdot 3 + 1 \cdot 0 \end{bmatrix}$$

$$= \begin{bmatrix} 11 & 6 \\ 5 & 0 \\ 1 & 0 \end{bmatrix}$$

Sl. 3.8. Primjer množenja matrica

Kod množenja matrica vrijede iduća svojstva:

- Asocijativnost: $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
- Distributivnost prema zbrajanju: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
- Neutralni element za množenje: $A \cdot I = I \cdot A = A$
- Ne vrijedi komutativnost: $A \cdot B \neq B \cdot A$

3.2.4. Programska implementacija množenja

Funkcija za množenje matrica također prima pokazivače u koje su kopirane prethodno inicijalizirane matrice. U treći pokazivač se sprema rezultat, a u prva dva matrice koje se množe.

Trenutni indeks matrice se dobije tako da se indeks i dimenzija bloka pomnože te se tome još pribroji indeks niti. Za retke se gledaju dimenzije i indeksi po vertikalnoj osi, a za stupce po horizontalnoj. Potom se petljom iterira od nule do veličine retka i stupca matrice te se obavlja množenje koje je programski implementirano kao što je opisano u prethodnom potpoglavlju. Na slici 3.9. je prikazana CUDA funkcija za množenje matrica.

```
__global__ void gpu_matrix_mult(int* A, int* B, int* C, int matrixSize)
{

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int sum = 0;

    if (col < matrixSize && row < matrixSize)
    {
        for (int i = 0; i < matrixSize; i++)
        {

            sum += A[row * matrixSize + i] * B[i * matrixSize + col];
        }


        C[row * matrixSize + col] = sum;
    }
}
```

Sl. 3.9. CUDA funkcija za množenje matrica

Provjera točnosti izvođenja algoritma vrši se na isti način kao kod zbrajanja matrica.

3.2.5. Inverz matrice

Kaže se da je matrica *A* regularna ako postoju matrica *B* takva da vrijedi $A \cdot B = B \cdot A = I$. U tom se slučaju matrica *B* zove multiplikativni inverz ili inverzna matrica od *A* i označava s A^{-1} [5]. Za matricu *A* koja nema multiplikativni inverz kažemo da je singularna [5].

U ovom radu je za računanje inverza korištena Gauss-Jordanova metoda eliminacije. Navedena metoda se izvodi na način da se s desne strane početna matrica proširi jediničnom matricom istih dimenzija. Na slici 3.10. je prikazana proširena matrica [6].

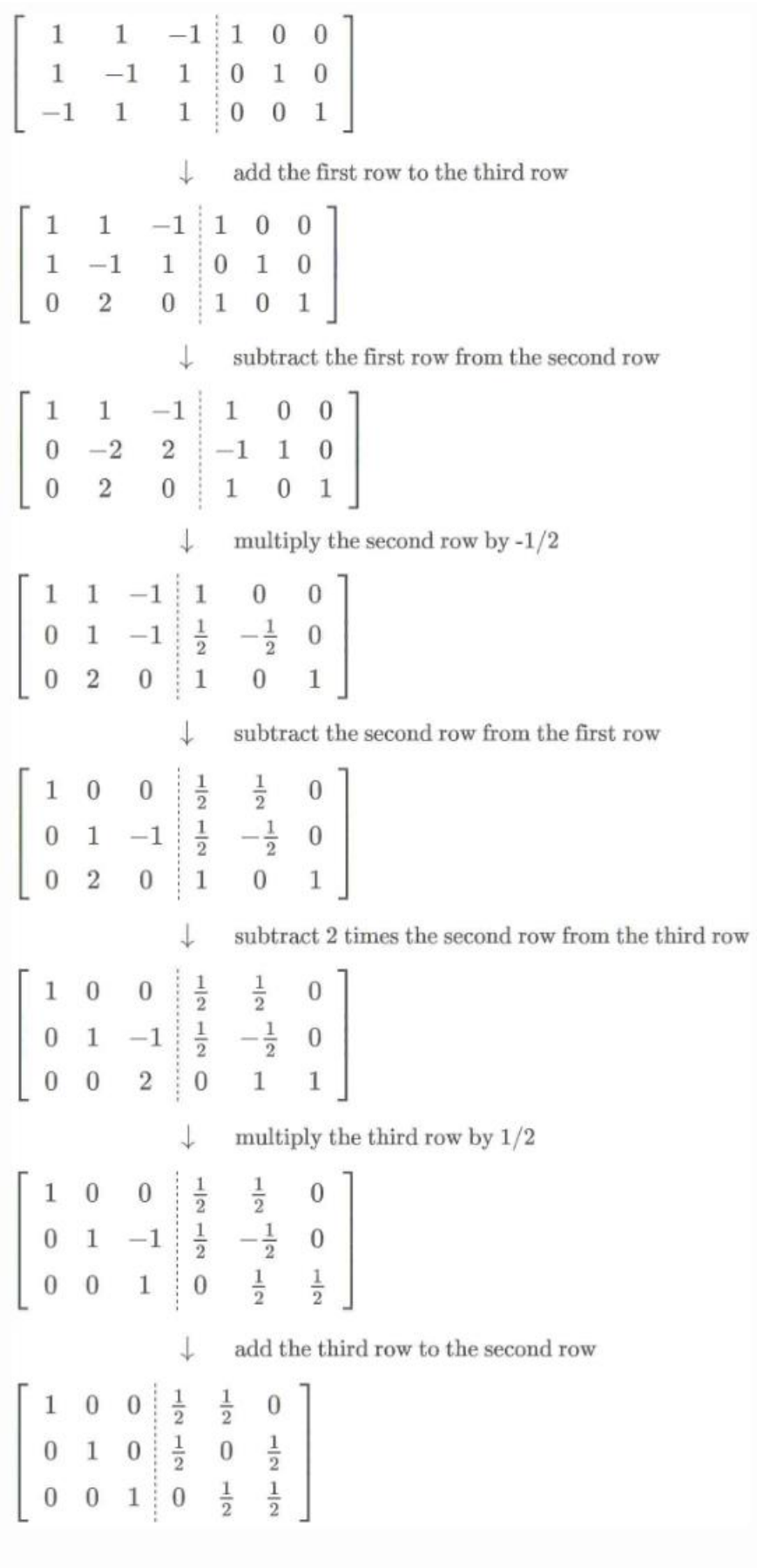
$$\left[\begin{array}{c:c} A & I \end{array}\right] = \left[\begin{array}{ccc:ccc} 1 & 1 & -1 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ -1 & 1 & 1 & 0 & 0 & 1 \end{array}\right]$$

Sl. 3.10. Proširena matrica

Potom se izvršavaju elementarne operacije nad proširenom matricom sve dok se s lijeve strane ne dobije jedinična matrica. Elementarne operacije su:

- Izmjena bilo koja dva retka ili dva stupca
- Množenje retka ili stupca brojem koji nije nula
- Množenje retka ili stupca s brojem koji nije nula i dodavanje rezultata drugom retku ili stupcu

Nakon navedenog postupka će se s desne strane nalaziti inverz početne matrice. Slika 3.11. prikazuje cijeli postupak dobivanja inverza od matrice prikazane na slici 3.10. [6].



Sl. 3.11. Prikaz postupka računanja inverza Gauss-Jordanovom metodom

3.2.6. Programska implementacija inverza

Gauss-Jordanova metoda jedna je od najstarijih metoda za računanje inverza matrice. To je jednostavna i robusna metoda te je posebno prikladna za masivnu paralelizaciju za razliku od mnogih naprednijih metoda [7].

Zbog preglednosti i jednostavnosti koda programska implementacija inverza podijeljena je u više funkcija. Sve funkcije se zovu u *main()* funkciji unutar *for* petlje. Petlja iterira onoliko puta koliko je veličina matrice po jednoj dimenziji. Funkcijama se prosljeđuje trenutni broj iteracije petlje.

Prvo su implementirane dvije funkcije za normaliziranje elemenata na dijagonali i izvan dijagonale. Normaliziranje znači skaliranje elemenata matrice na određenu vrijednost kako bi se olakšalo izvođenje algoritma i pojednostavilo računanje. Normaliziranje se provodi dijeljenjem elemenata početne matrice(*A*) i pripadajuće jedinične matrice(*I*) s elementima na dijagonali početne matrice. Na slici 3.12. je prikazana implementacija funkcije za normaliziranje [7].

```
__global__ void diag_normalize(double* A, double* I, int n, int i) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < n && y < n)
        if (x == y && x == i) {
            I[x * n + y] /= A[i * n + i];
            A[x * n + y] /= A[i * n + i];
        }

}
```

Sl. 3.12. CUDA funkcija za normaliziranje dijagonale

Kod primjene Gauss-Jordanove metode prvo se provjerava u *if* uvjetu je li trenutna pozicija u matrici različita od trenutne pozicije retka s pivot elementom. Zatim se svakim korakom oduzima umnožak pivot elementa retka matrice *I* i odgovarajućeg elementa matrice *A* od trenutnog elementa retka matrice *I*. Drugim *if* uvjetom se utvrđuje je li trenutna pozicija u matrici različita od trenutne pozicije stupca s pivot elementom. U drugom uvjetu se izvršava oduzimanje umnoška pivot elementa retka matrice *A* i odgovarajućeg elementa matrice *A* od trenutnog elementa retka matrice *A*. Navedene operacije smanjuju elemente redaka obje matrice koje

trenutno ne sadrže pivot element. Slika 3.13. prikazuje implementaciju Gauss-Jordanove metode [7].

```
__global__ void gaussjordan(double* A, double* I, int n, int i)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < n && y < n) {
        if (x != i) {
            I[x * n + y] -= I[i * n + y] * A[x * n + i];
            if (y != i) {
                A[x * n + y] -= A[i * n + y] * A[x * n + i];
            }
        }
    }

}
```

Sl. 3.13. CUDA funkcija za izračun Gauss-Jordanove metode

U zadnjoj funkciji se postavljaju elementi početne matrice *A* na vrijednost nula u stupcu koji je jednak proslijeđenom indeksu. Funkcija za postavljanje vrijednosti na nulu je prikazana na slici 3.14. [7].

```
__global__ void set_zero(double* A, double* I, int n, int i) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < n && y < n) {
        if (x != i) {
            if (y == i) {
                A[x * n + y] = 0;
            }
        }
    }
}
```

Sl. 3.14. CUDA funkcija za postavljanje nula

Provjera točnosti izvršavanja algoritma se vrši na način da se početna matrica A pomnoži s dobivenim inverzom i kao rezultat množenja se mora dobiti jedinična matrica. Ukoliko se ne dobije jedinična matrica rezultat nije točan.

4. MJERENJE PERFORMANSI ALGORITAMA

Za svaki implementirani algoritam je provedeno mjerenje vremena s više veličina matrica. Sve korištene matrice su kvadratne. Gdje je bilo moguće promijenjena je i veličina bloka kod poziva funkcije. Mijenjanjem veličine bloka se mijenja broj niti koje se nalaze u njemu. Na taj način se mijenja organizacija memorije i dolazi do drugačijih rezultata.

Mjerenja su provedena na tri različite grafičke kartice s različitim brojem CUDA jezgri. Grafičke kartice su:

1. NVIDIA RTX 3080, 8704 CUDA jezgre
2. NVIDIA RTX 3060 Ti, 4864 CUDA jezgre
3. NVIDIA GTX 1650, 896 CUDA jezgri

Još se u obzir treba uzeti i brzine takta, propusnost memorije, brzinu memorije, količinu memorije i arhitekturu.

CPU koji se koristio za mjerenje vremena kod serijskih verzija algoritama je Ryzen 5 5600X i on se nalazi u konfiguraciji sa RTX 3060 Ti GPU-om. RTX 3080 se nalazi u konfiguraciji sa i7 9700k CPU-om, a GTX 1650 sa Ryzen 5 4600H CPU-om. U različitim konfiguracijama, vrijeme kopiranja memorije s *host-a* na *device* i s *device-a* na *host* može biti različito, što također utječe na brzinu izvođenja algoritma.

4.1. Mjerenje vremena

Brzine izvođenja se mjere pomoću dva mjerača vremena. Za algoritme koji se izvode na CPU-u se koristi *Chrono* mjerač vremena, a za paralelizirane algoritme koji se izvode na GPU-u se koristi CUDIN mjerač vremena.

4.1.1. Chrono mjerač vremena

Chrono je C++ zaglavlje koje pruža zbirku tipova i funkcija za rad s vremenom. Dio je C++ standardne biblioteke predložaka (STL) i uključen je u C++11 i novije verzije [8]. On pruža tri različita tipa sata: *system_clock, steady_clock,* i *high_resolution_clock.* U ovom radu je korišten samo *system_clock* koji predstavlja trenutno vrijeme prema sustavu. Svaki sat se sastoji od početne točke (epohe) i brzine otkucaja. U ovom radu se koristi brzina otkucaja zadana u milisekundama. Slika 4.1. prikazuje primjer korištenja *chrono* mjerača vremena.

```
auto clock_start_CPU = std::chrono::system_clock::now();

cpu_matrix_mult(hA, hB, hCC, matrixSize);


auto clock_now_CPU = std::chrono::system_clock::now();

float CPU_time = float(std::chrono::duration_cast
<std::chrono::milliseconds> (clock_now_CPU - clock_start_CPU).count());


std::cout << "Vrijeme na CPU: " << CPU_time << " ms \n";
```

Sl. 4.1. Primjer korištenja *chrono* mjerača vremena

4.1.2. CUDA mjerač vremena

CUDA nudi relativno laganu alternativu CPU mjeračima vremena putem *CUDA event API-ja*. *CUDA event API* uključuje pozive za stvaranje i uništavanje događaja, snimanje događaja i izračunavanje proteklog vremena u milisekundama između dva snimljena događaja [9]. CUDA događaji su podatkovnog tipa *cudaEvent_t* i stvaraju se pomoću *cudaEventCreate()* funkcije, a uništavaju pomoću *cudaEventDestroy()* funkcije. Funkcija *cudaEventSynchronize()* blokira izvršavanje CPU-a dok se proslijeđeni događaj ne zabilježi. Funkcija *cudaEventElapsedTime()* u prvom argumentu vraća broj milisekundi koji je protekao između početka i kraja mjerenja. Ova vrijednost ima rezoluciju od približno pola mikrosekunde [9]. Na slici 4.2. je prikazan primjer korištenja CUDA mjerača vremena.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);

cudaMemcpy(dA, hA, sizeof(int) * matrixSize * matrixSize, cudaMemcpyHostToDevice);
cudaMemcpy(dB, hB, sizeof(int) * matrixSize * matrixSize, cudaMemcpyHostToDevice);

unsigned int gridSize = (matrixSize + BLOCK_SIZE - 1) / BLOCK_SIZE;

dim3 dimGrid(gridSize, gridSize);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

gpu_matrix_mult << <dimGrid, dimBlock >> > (dA, dB, dC, matrixSize);

cudaMemcpy(hC, dC, sizeof(int) * matrixSize * matrixSize, cudaMemcpyDeviceToHost);

cudaThreadSynchronize();

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&GPU_time, start, stop);

cout << "Vrijeme na GPU:" << GPU_time << "ms" << endl;
```

Sl. 4.2. Primjer korištenja CUDA mjerača vremena

4.2. Performanse zbrajanja

Zbrajanje je najjednostavnija implementirana i provedena operacija. Dobiveni rezultati mjerenja jasno ukazuju na to da zbrajanje nije pogodno za paralelno izvođenje. Vremena izvođenja su sporija na GPU-u nego na CPU-u. U tablici 4.1. su prikazana izmjerena vremena izvođenja algoritma za zbrajanje sa navedenim veličinama matrice.

Tablica 4.1. Vremena za zbrajanje

Veličina matrice	*1000x1000*	*2000x2000*	*3000x3000*	*4000x4000*	*6000x6000*	*8000x8000*	*10000x10000*	*11000x11000*
RTX 3080 vrijeme [ms]	4	14	31	57	126	227	363	417
RTX 3060 Ti vrijeme [ms]	4	14	29	54	120	209	335	363
GTX 1650 vrijeme [ms]	8	26	58	103	232	378	573	660
CPU vrijeme[ms]	1	5	14	25	50	99	137	183

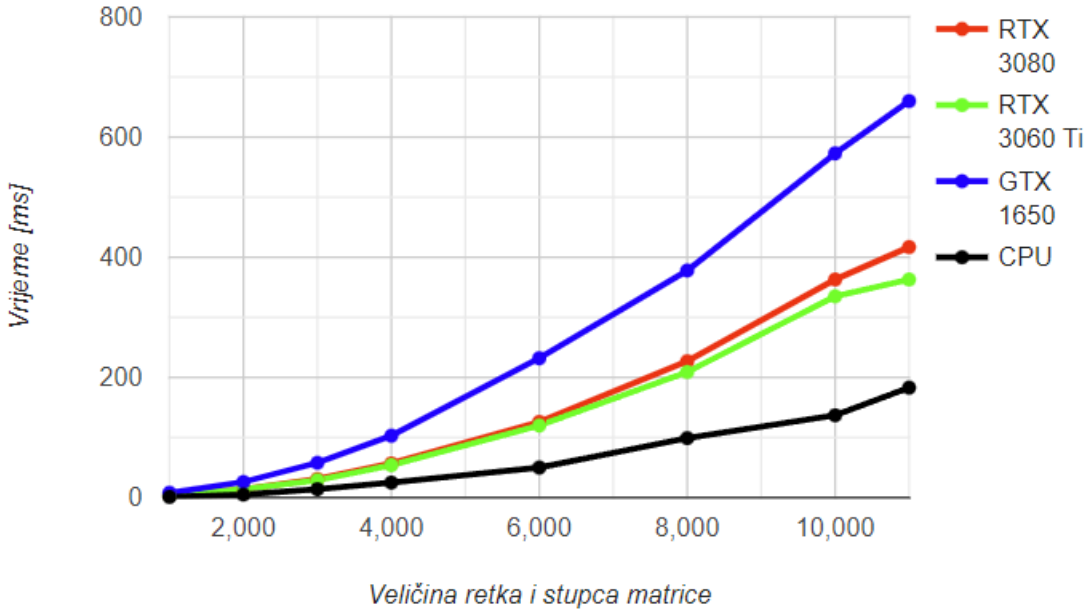
Ako se veličine matrice postave na približne vrijednosti kao u prošlom slučaju, ali tako da budu zapisane pomoću brojeva koji su potencije broja dva, ne postigne se velika razlika u brzinama izvođenja. Tablica 4.2. prikazuje brzine izvođenja kada su veličine matrice zapisane pomoću brojeva koji su potencije broja dva.

Tablica 4.2. Vremena za zbrajanje kod veličina matrice zapisane pomoću potencija broja dva

Veličina matrice	*1024x1024*	*2048x2048*	*3072x3072*	*4096x4096*	*6144x6144*	*8192x8192*	*10240x10240*
RTX 3080 vrijeme [ms]	3	15	33	60	134	233	362
RTX 3060 Ti vrijeme [ms]	4	13	29	53	117	205	310
GTX 1650 vrijeme [ms]	8	27	60	106	238	389	580
CPU vrijeme[ms]	1	5	13	22	48	93	139

U oba slučaja su RTX 3080 i RTX 3060 Ti bili osjetno brži u odnosu na GTX 1650. Unatoč manjem broju jezgri RTX 3060 Ti se pokazao nijansu bržim u odnosu na RTX 3080. Najvjerojatniji razlog tomu je vrijeme kopiranja memorije. U svim mjerenjima algoritmi su dali točan rezultat.

Vremenska složenost zbrajanja matrica teorijski iznosi $O(n^2)$. Pomoću izmjerenih rezultata se ta složenost može grafički prikazati. Na slici 4.3. su grafički prikazani rezultati mjerenja iz prvog slučaja.



Sl. 4.3. Grafički prikaz vremenske složenosti zbrajanja

4.3. Performanse množenja

Kod množenja se postiglo višestruko ubrzanje korištenjem paralelnog algoritma u svim slučajevima. U prvom slučaju je korištena veličina bloka 16x16. U tablici 4.3. su prikazani rezultati mjerenja za prvi slučaj.

Tablica 4.3. Vremena za množenje veličina bloka 16x16

Veličina matrice	*100x100*	*200x200*	*500x500*	*1000x1000*	*1500x1500*	*2000x2000*	*3000x3000*	*4000x4000*
RTX 3080 vrijeme [ms]	0.17	0.2	0.56	2	6	12	37	84
RTX 3060 Ti vrijeme [ms]	0.14	0.16	0.6	4	8	18	58	135
GTX 1650 vrijeme [ms]	0.16	0.3	2	12	41	84	276	660
CPU vrijeme[ms]	1	1	42	542	1874	5430	46596	121705

U drugom slučaju je veličina matrice ostala ista, ali je veličina bloka smanjenja na 4x4. Smanjenjem veličine bloka su se vremena izvođenja povećala na svim GPU-ovima u odnosu na prošli slučaj. Tablica 4.4. prikazuje rezultate mjerenja za drugi slučaj.

Tablica 4.4. Vremena za množenje veličina bloka 4x4

Veličina matrice	*100x100*	*200x200*	*500x500*	*1000x1000*	*1500x1500*	*2000x2000*	*3000x3000*	*4000x4000*
RTX 3080 vrijeme [ms]	0.2	0.22	0.7	4	12	25	84	263
RTX 3060 Ti vrijeme [ms]	0.13	0.18	0.8	7	20	44	193	426
GTX 1650 vrijeme [ms]	0.36	0.4	4	26	93	238	653	1931
CPU vrijeme[ms]	1	1	42	542	1874	5430	46596	121705

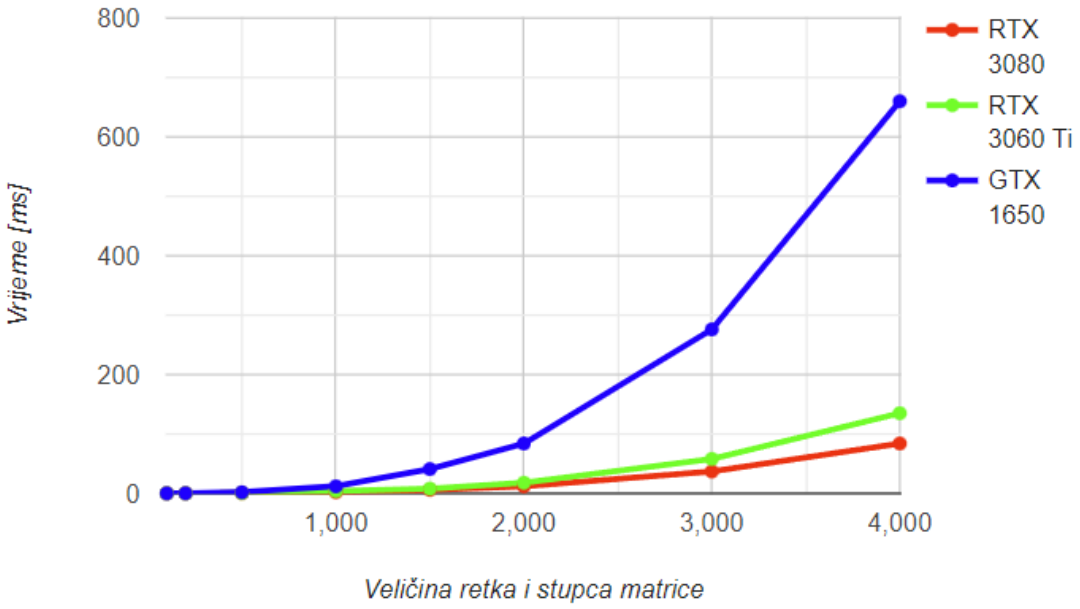
U trećem slučaju je veličina bloka postavljena natrag na 16x16, ali je veličina redaka i stupaca matrice postavljena na brojeve koji su zapisani pomoću potencije broja dva. Zadana veličina matrice nije imala utjecaja na brzine izvođenja paralelnih algoritama, ali je zato osjetno povećala vrijeme izvođenja serijske verzije algoritma. U tablici 4.5. su prikazani rezultati mjerenja za treći slučaj.

Tablica 4.5. Vremena za množenje veličina bloka 16x16 i kod veličine matrice zapisane pomoću potencije broja dva

Veličina matrice	*128x128*	*256x256*	*512x512*	*1024x1024*	*1536x1536*	*2048x2048*	*3072x3072*	*4096x4096*
RTX 3080 vrijeme [ms]	0.2	0.32	0.6	2	7	15	40	90
RTX 3060 Ti vrijeme [ms]	0.14	0.21	0.6	3	9	19	64	151
GTX 1650 vrijeme [ms]	0.2	0.47	2	14	47	105	319	651
CPU vrijeme[ms]	1	1	182	3553	7562	44140	156302	373540

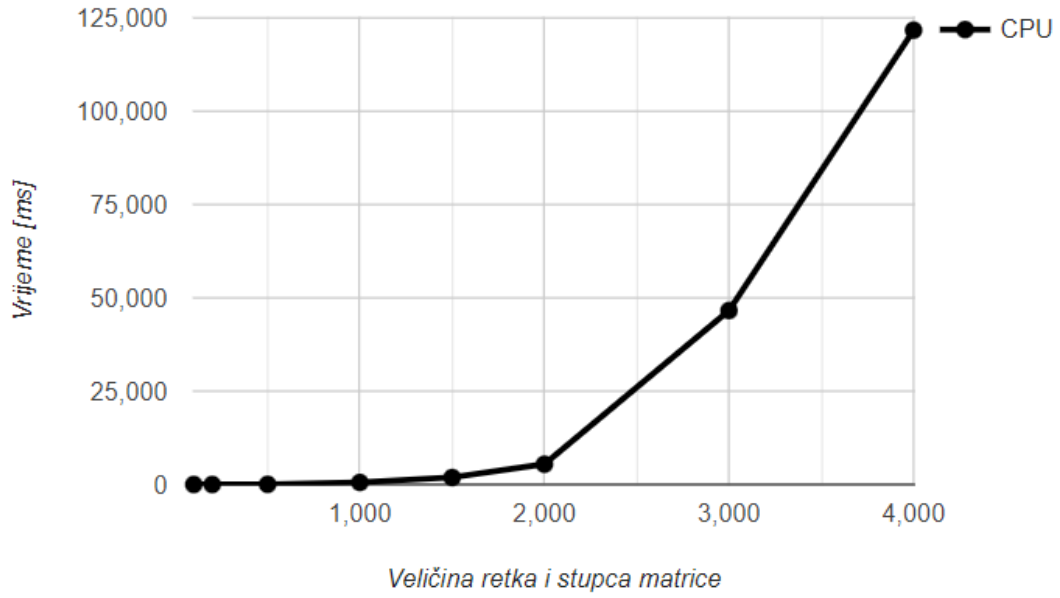
RTX 3080 se pokazala najbržom u svim slučajevima, druga je bila RTX 3060 Ti, a najsporija je bila GTX 1650. Sve tri grafičke kartice su bile brže od CPU-a u svim slučajevima. Kod svakog mjerenja je rezultat bio točan.

Vremenska složenost množenja matrica teorijski iznosi $O(n^3)$. Pomoću izmjerenih vrijednosti je na linijskom grafu prikazano vrijeme izvođenja u ovisnosti o veličini matrice. Na slici 4.4. su prikazana vremena izvođenja za sve GPU-ove s vrijednostima iz prvog slučaja.



Sl. 4.4. Grafički prikaz vremenske složenosti množenja na GPU

Zbog velike vremenske razlike vrijeme izvođenja na CPU-u je prebačeno na posebnu sliku. Slika 4.5. prikazuje vrijeme izvođenja za CPU s vrijednostima iz prvog slučaja.



Sl. 4.5. Grafički prikaz vremenske složenosti množenja na CPU

4.4. Performanse inverza

Algoritam inverza matrice je također postigao višestruko ubrzanje korištenjem paralelne verzije algoritma u svim slučajevima. Vremena izvođenja za prvi slučaj su prikazan u tablici 4.6. Veličina bloka je postavljena na 16x16. Kod veličine matrice 500x500 rezultat paralelne verzije algoritma nije točan kod sva tri GPU-a.

Tablica 4.6. Vremena za inverz veličina bloka 16x16

Veličina matrice	*100x100*	*200x200*	*500x500*	*800x800*	*1000x1000*	*1500x1500*	*2000x2000*	*3000x3000*
RTX 3080 vrijeme [ms]	1.8	4	14	72	157	546	959	4055
RTX 3060 Ti vrijeme [ms]	1.5	5	44	133	257	792	1197	4331
GTX 1650 vrijeme [ms]	3.5	10	84	329	521	2133	4626	17175
CPU vrijeme[ms]	2	16	258	1061	2071	7068	16922	57249

Ako se veličina bloka smanji na 4x4 vremena izvođenja se povećaju. Netočni rezultati su kod veličine 500x500 na svim grafičkim karticama i kod veličine 1500x1500 za RTX 3060 Ti. U tablici 4.7. su prikazana vremena izvođenja za drugi slučaj.

Tablica 4.7. Vremena za inverz veličina bloka 4x4

Veličina matrice	*100x100*	*200x200*	*500x500*	*800x800*	*1000x1000*	*1500x1500*	*2000x2000*	*3000x3000*
RTX 3080 vrijeme [ms]	1.7	4	30	196	364	1211	2692	7900
RTX 3060 Ti vrijeme [ms]	1.5	5	83	325	567	1616	3427	11599
GTX 1650 vrijeme [ms]	3.2	12	178	561	980	3831	10460	50063
CPU vrijeme[ms]	2	16	258	1061	2071	7068	16922	57249

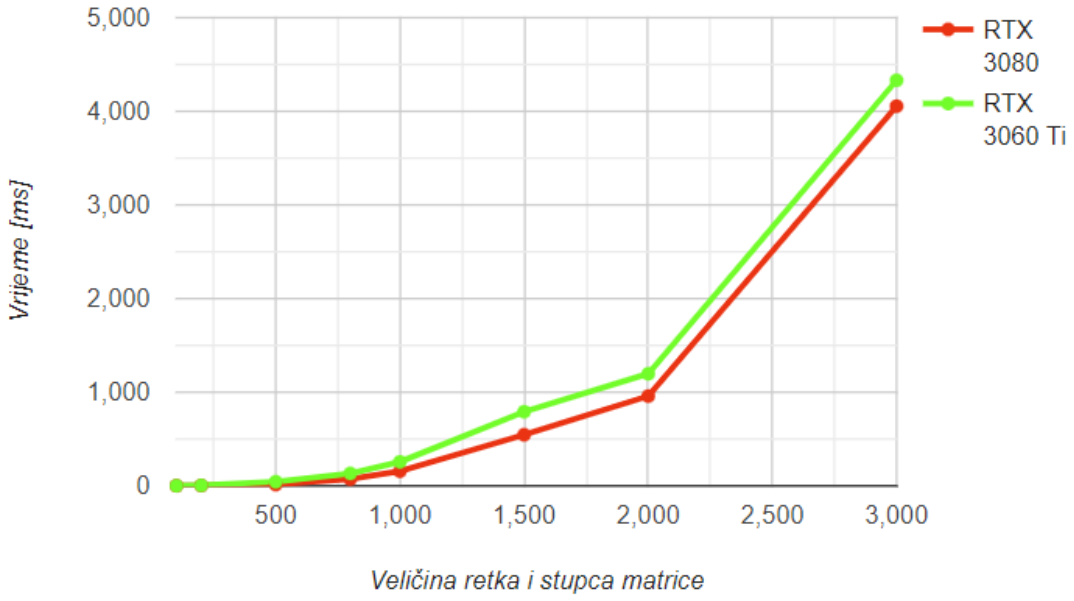
Ukoliko se veličine matrice postave tako da su zapisane pomoću potencije broja dva, a veličina bloka se postavi natrag na 16x16, vrijeme izvođenja se ne mijenja značajno u odnosu na prvi slučaj. U ovom slučaju su svi rezultati točni. U tablici 4.8. su prikazane brzine izvođenja za treći slučaj.

Tablica 4.8. Vremena za inverz veličina bloka 16x16 i kod veličine matrice zapisane pomoću potencije broja dva

Veličina matrice	*128x128*	*256x256*	*512x512*	*768x768*	*1024x1024*	*1536x1536*	*2048x2048*	*3072x3072*
RTX 3080 vrijeme [ms]	2.1	6	18	65	115	475	876	3763
RTX 3060 Ti vrijeme [ms]	3.2	7	27	94	164	717	1031	4674
GTX 1650 vrijeme [ms]	5.4	19	113	301	661	2139	4991	16713
CPU vrijeme[ms]	4	34	277	938	2233	7603	18181	61400

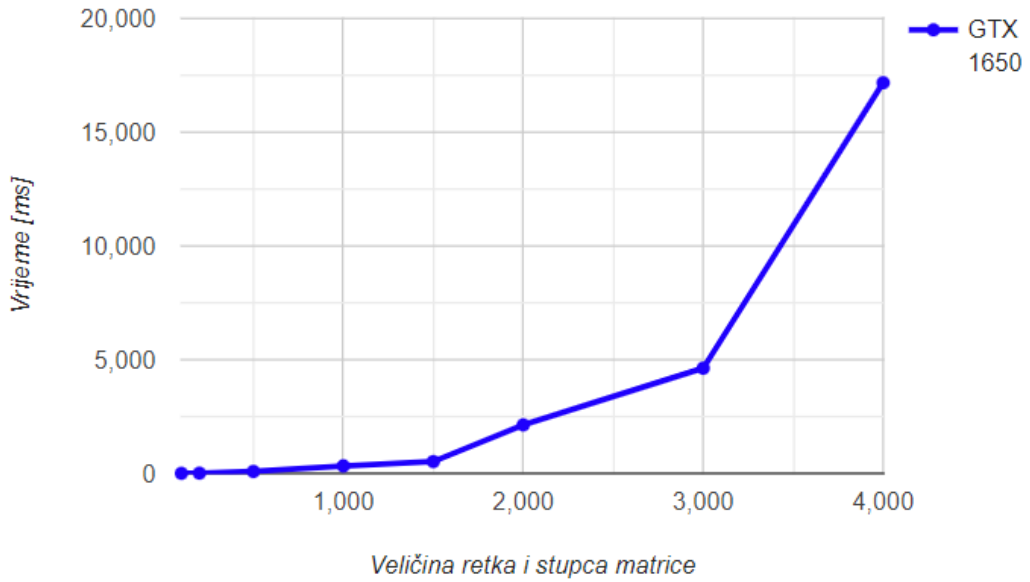
RTX 3080 je bila najbrža u svakom slučaju, slijedi RTX 3060 Ti, dok je najsporija bila GTX 1650. Serijska verzija algoritma je bila uvijek najsporija.

Gauss-Jordanova metoda teorijski ima vremensku složenost $O(n^3)$. Na slici 4.6. su prikazana vremena izvođenja za RTX 3080 i RTX 3060 Ti sa vrijednostima iz prvog slučaja.



Sl. 4.6. Grafički prikaz vremenske složenosti inverza za RTX 3080 i RTX 3060 Ti

Zbog veće razlike u brzini izvođenja vremena za GTX 1650 i CPU su odvojena na posebne slike. Na slici 4.7. je prikazano vrijeme izvođenja za GTX 1650, a na slici 4.8. za CPU.



Sl. 4.7. Grafički prikaz vremenske složenosti inverza za GTX 1650

Sl. 4.8. Grafički prikaz vremenske složenosti inverza za CPU

5. ZAKLJUČAK

CUDA tehnologija omogućuje izvođenje algoritama paralelno na grafičkoj kartici. Svojim dolaskom na tržište omogućila je brže rješavanje kompleksnih algoritama u simulacijama. Prije pojave CUDA tehnologije za pojedine simulacije su bila potrebna superračunala koja nisu bila dostupna većem broju korisnika. Za korištenje CUDA tehnologije potrebno je imati NVIDIA grafičku karticu i instaliran *CUDA toolkit*. U ovom radu CUDA programi su pisani u programskom jeziku C++, iako se mogu koristiti i drugi programski jezici. Dodavanjem određenih ključnih riječi u postojeći C++ program se označava koji dio programa će se izvoditi paralelno na GPU-u.

Za potrebe ovog rada su implementiranu algoritmi zbrajanja množenja i inverza matrice. Implementirani su tako da se mogu izvoditi serijski na CPU-u i paralelno na GPU-u. Paralelnim izvođenjem se u većini slučajeva povećavaju performanse algoritma u odnosu na serijsku verziju koja se izvodi na CPU-u. Neki algoritmi nisu pogodni za paralelno izvođenje, kao što je primjerice zbrajanje matrica. Zbrajanje matrica se sporije izvodi paralelno nego serijski. Algoritmi množenja i inverza matrice su postigli značajno ubrzanje izvodeći se paralelno u svim testiranim slučajevima. Kod paralelnog izvođenja se može mijenjati raspored i organizacija memorije. Mijenjanjem organizacije i rasporeda memorije dolazi se do drugačijih rezultata kod mjerenja vremena izvođenja. Paralelni algoritmi u nekim specifičnim slučajevima mogu dati netočan rezultat kao kod računanja inverza sa određenim veličinama matrice. Rezultati svih mjerenja su grafički prikazani na linijskim grafovima. Prikazano je vrijeme izvođenja u ovisnosti od veličine matrice kako bi se grafički prikazala vremenska složenost koja je teorijski definirana za svaki algoritam.

Moguće je još implementirati obrađene algoritme na drugačiji način koji će možda biti efikasniji i točniji od načina koji je korišten u ovom radu. Matrice koje se koriste za ispitivanje mogu biti različitog oblika od kvadratnog. Pri pozivanju CUDA funkcije može se koristiti različita veličina bloka koja također ne mora biti kvadratnog oblika.

LITERATURA

[1] "CUDA Zone - Library of Resources", https://developer.nvidia.com/cuda-zone , srpanj 2023.

[2] J. Sanders, E. Kandrot, "CUDA by example", str. 4-11, srpanj 2023.

[3] N. Elezović, "Linearna algebra", str. 1-14, kolovoz 2023.

[4] K. Mautner, "Android aplikacija za rad s matricama", kolovoz 2023.

[5] D. Bakić, "Linearna algebra", str. 56-62, kolovoz 2023.

[6] "Examples of Gauss-Jordan elimination", https://semath.info/src/inverse-elimination.html, kolovoz 2023.

[7] G. Sharma, A. Agarwala, B. Bhattacharya, "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA", kolovoz 2023.

[8] "Chrono in C++", https://www.geeksforgeeks.org/chrono-in-c/, kolovoz 2023.

[9] "How to Implement Performance Metrics in CUDA C/C++, https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/, kolovoz 2023.

SAŽETAK

U ovom radu su korištenjem CUDA tehnologije implementirani algoritmi za zbrajanje, množenje i inverz matrice. U početku je objašnjenja CUDA arhitektura i navedeni su neki primjeri primjene CUDA tehnologije u industriji. Zatim su algoritmi objašnjeni teorijski i prikazan je način njihove programske implementacije. Na svim implementiranim algoritmima je provedeno testiranje performansi i točnosti koristeći tri modela grafičkih kartica. Zbog bolje mogućnosti usporedbe korištena je i serijska verzija svih algoritama koja se izvodi na CPU-u. Izmjereni rezultati su grafički prikazani kako bi se dobio bolji uvid u vremensku složenost implementiranih algoritama.

Ključne riječi: algoritam, CUDA, GPU, matrica, performanse

ABSTRACT

Title: Algorithms with matrices using CUDA technology

In this paper, algorithms for addition, multiplication and matrix inverse are implemented using CUDA technology. In the beginning, the CUDA architecture is explained and some examples of the application of CUDA technology in industry are given. Then the algorithms are explained theoretically and the method of their programmatic implementation is shown. All implemented algorithms were tested for performance and accuracy using three models of graphics cards. Due to the better possibility of comparison, the serial version of all algorithms, which is executed on the CPU, was also used. The measured results are graphically presented in order to get a better insight into the time complexity of the implemented algorithms.

Keywords: algorithm, CUDA, GPU, matrix, performance