UNIVERZITET U BEOGRADU - ELEKTROTEHNIČKI FAKULTET

MULTIPROCESORKI SISTEMI (13S114MUPS, 13E114MUPS)

DOMAĆI ZADATAK 4 – CUDA

Izveštaj o urađenom domaćem zadatku

Predmetni saradnici:

dipl. ing. Matija Dodović

Studenti:

Ljubica Majstorović 2020/0253

Pavle Šarenac 2020/0359

Beograd, januar 2023.

SADRŽAJ

1. Problem 1 – Izračunavanje aritmetičkih brojeva

1.1. Tekst problema

Paralelizovati program koji vrši izračunavanje aritmetičkih brojeva. Pozitivan ceo broj je aritmetički ako je prosek njegovih pozitivnih delilaca takođe ceo broj. Program se nalazi u datoteci aritmetic.c u arhivi koja je priložena uz ovaj dokument. Program testirati sa parametrima koji su dati u run skripti.

Prilikom zadavanja izvršne konfiguracije jezgra, koristiti 1D rešetku (grid).

1.2. Delovi koje treba paralelizovati

1.2.1. Diskusija

Segment koda nad kojim je izvršena paralelizacija u ovom problemu je sledeći:

```
for (n = 1; arithmetic_count <= num; ++n)
{
    unsigned int divisor_count;
    unsigned int divisor_sum;
    divisor_count_and_sum(n, &divisor_count, &divisor_sum);
    if (divisor_sum % divisor_count != 0)
        continue;
    ++arithmetic_count;
    if (divisor_count > 2)
        ++composite_count;
}
```

Ova petlja je vrlo pogodna za paralelizaciju zato što ne postoji nikakva zavisnost po podacima između njenih iteracija, pa one mogu da se izvršavaju potpuno nezavisno.

Jedina nezgodna stvar je što broj iteracija ovako napisane petlje nije unapred poznat, pa je potrebno malo preurediti ovaj kod kako bi bila moguća takva paralelizacija.

Još jedan deo koda koji troši značajan deo procesorskog vremena:

```
for (unsigned int p = 3; p * p <= n; p += 2)
{
    unsigned int count = 1, sum = 1;
    for (power = p; n % p == 0; power *= p, n /= p)
    {
        ++count;
        sum += power;
    }
    divisor_count *= count;
    divisor_sum *= sum;
}
```

Zaključili smo da nije dobro pokušati paralelizaciju ove petlje jer postoji zavisnost po podacima između iteracija spoljašnje petlje zato što se „n“ koje je u uslovu spoljašnje petlje menja u unutrašnjoj petlji, pa zato ovako napisan algoritam nije pogodan za paralelizaciju. Da bi paralelizacija potencijalno bila moguća, bilo bi potrebno kompletno restrukturiranje ovog dela algoritma.

1.2.2. Način paralelizacije

Zaključili smo da nije moguće našu glavnu for petlju paralelizovati zato što forma petlje nije odgovarajuća pošto nije poznat broj iteracija unapred.

Međutim, smislili smo način da ipak upotrebimo napišemo petlju kod koje znamo broj iteracija unapred. Rešenje leži u činjenici da ako želimo da nađemo npr. prvih 10 000 aritmetičkih brojeva, mi zasigurno moramo da imamo najmanje 10 000 iteracija petlje. Ne znamo unapred za koliko će broj iteracija petlje biti veći od 10 000, ali znamo da ih mora biti najmanje 10 000. Ovo možemo iskoristiti onda tako što ćemu napisati for petlju koja će imati 10 000 iteracija, i u njoj je onda poznat broj iteracija unapred pa se može paralelizovati. Uslov prvobitne for petlje ćemo izmestiti u novu spoljašnju while petlju. Recimo da se u 10 000 iteracija for petlje našlo 8 000 aritmetičkih brojeva. To znači da for petlja u narednoj iteraciji while petlje mora da se izvrši još najmanje 2 000 puta jer je toliko aritmetičkih brojeva preostalo da se pronađe. Ovaj postupak ponavljamo sve dok se ne pronađe željeni broj aritmetičkih brojeva. Na ovaj način smo u svakoj iteraciji spoljašnje while petlje menjali broj iteracija for petlje tako da se uvek znao unapred taj broj, što nam je omogućilo paralelizaciju.

1.3. Rezultati

U okviru ove sekcije su izloženi rezultati paralelizacije problema 1.

1.3.1. Logovi izvršavanja

```
Sequential implementation execution time: 0.002882s
Parallel implementation execution time: 0.000243s
Test PASSED
```

Izvršavanje komande ./arithmetic 10000

```
Sequential implementation execution time: 0.008618s
Parallel implementation execution time: 0.000405s
Test PASSED
```

Izvršavanje komande ./arithmetic 100000

```
Sequential implementation execution time: 0.177049s
Parallel implementation execution time: 0.003283s
Test PASSED
```

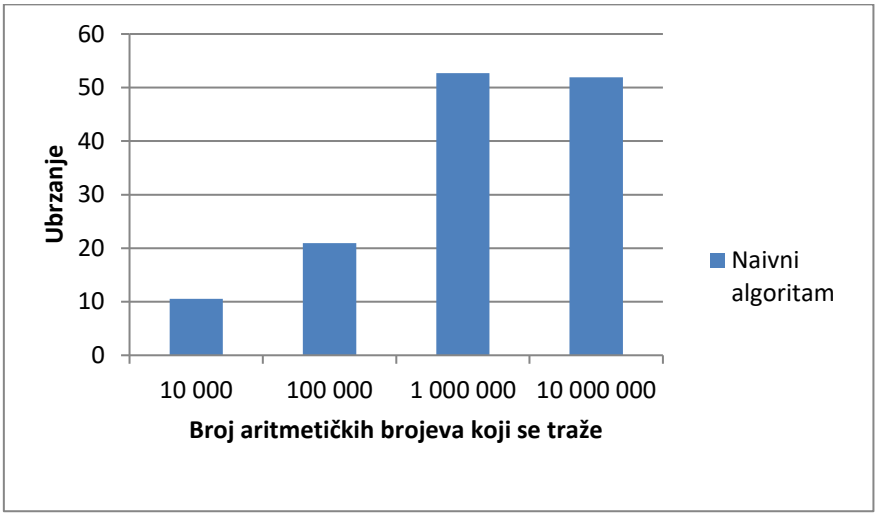
Izvršavanje komande ./arithmetic 1000000

```
Sequential implementation execution time: 4.133159s
Parallel implementation execution time: 0.080925s
Test PASSED
```

Izvršavanje komande ./arithmetic 10000000

1.3.2. Grafici ubrzanja

U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja aritmetičkih brojeva koji se traže

1.3.3. Diskusija dobijenih rezultata

Kako bi grafici i naši rezultati bili precizniji, zabeležili smo rezultate kako sekvencijalnog, tako i paralelnog izvršavanja programa za sve parametre i za svaki broj niti 5 puta i uzimali prosečna vremena.

Postignuta ubrzanja su odlična, vidimo da idu i do preko 50 puta za najveće ulazne parametre.

2. PROBLEM 2 – GENERISANJE ELEMENATA HALTON QUASI MONTE CARLO SEKVENCE

2.1. Tekst problema

Paralelizovati program koji vrši generisanje elemenata Halton Quasi Monte Carlo (QMC) sekvence. Program se nalazi u datoteci halton.c u arhivi koja je priložena uz ovaj dokument. Program testirati sa parametrima koji su dati u run skripti.

2.2. Delovi koje treba paralelizovati

2.2.1. Diskusija

Ova petlja bi mogla da se paralelizuje:

```
for (m = 1; m <= iter; m++)
{
    r = halton_sequence_sequential(0, 10, m);
    free(r);
}
```

Bili smo svesni da je bilo potrebno odlučiti da li je bolje paralelizovati ovu petlju ili paralelizovati petlje unutar funkcije halton_sequence_sequential. Očigledno je da sa svakom iteracijom raste i količina posla koju je potrebno obaviti, pa samim tim nije efikasno raspodeliti posao između niti tako da svaka dobije približno podjednak broj iteracija. Tako smo zaključili (a i uverili se isprobavanjem) da je bolje opredeliti se za paralelizaciju petlji unutar funkcije halton_sequence_sequential.

Sledeći deo koda je pogodan za paralelizaciju:

```
for (j = 0; j < m; j++)
{
    prime_inv[j] = 1.0 / (double)(prime(j + 1));
}
```

I narednu unutrašnju petlju treba paralelizovati:

```
for (j = 0; j < n; j++)
{
    for (i = 0; i < m; i++)
    {
        r[i + j * m] = 0.0;
    }
}
```

Primećeno je i da se isplati paralelizovati i sledeću petlju:

```
for (j = 0; j < m; j++)
{
    d = (t[j] % prime(j + 1));
    r[j + k * m] = r[j + k * m] + (double)(d)*prime_inv[j];
    prime_inv[j] = prime_inv[j] / (double)(prime(j + 1));
    t[j] = (t[j] / prime(j + 1));
}
```

Sve ove petlje su pogodne za paralelizaciju jer su im iteracije međusobno nezavisne.

Bila je razmatrana i paralelizacija naredne spoljašnje petlje:

```
for (k = 0; k < n; k++)
{
    for (j = 0; j < m; j++)
    {
        t[j] = i;
    }
    for (j = 0; j < m; j++)
    {
        prime_inv[j] = 1.0 / (double)(prime(j + 1));
    }

    while (0 < i4vec_sum(m, t))
    {
        for (j = 0; j < m; j++)
        {
            d = (t[j] % prime(j + 1));
            r[j + k * m] = r[j + k * m] + (double)(d)*prime_inv[j];
            prime_inv[j] = prime_inv[j] / (double)(prime(j + 1));
            t[j] = (t[j] / prime(j + 1));
        }
    }
    i = i + i3;
}
```

Međutim, na taj način se ne bi dobili isti rezultati kao kod sekvencijalne implementacije problema pošto postoji zavisnost po podacima između iteracija ove petlje – u svakoj iteraciji se promenljiva „i“ menja, a takođe se i koristi pri izračunavanjima.

2.2.2. Način paralelizacije

Paralelizacija je odrađena tako što se u svakom pozivu kernela obradi jedna kolona matrice, pri čemu svaka nit iz bloka niti obradi po jedan element kolone matrice.

2.3. Rezultati

2.3.1. Logovi izvršavanja

```
Sequential implementation execution time: 0.004241s
Parallel implementation execution time: 0.095614s
Test PASSED
```

Izvršavanje komande ./halton 10

```
Sequential implementation execution time: 0.028559s
Parallel implementation execution time: 0.085963s
Test PASSED
```

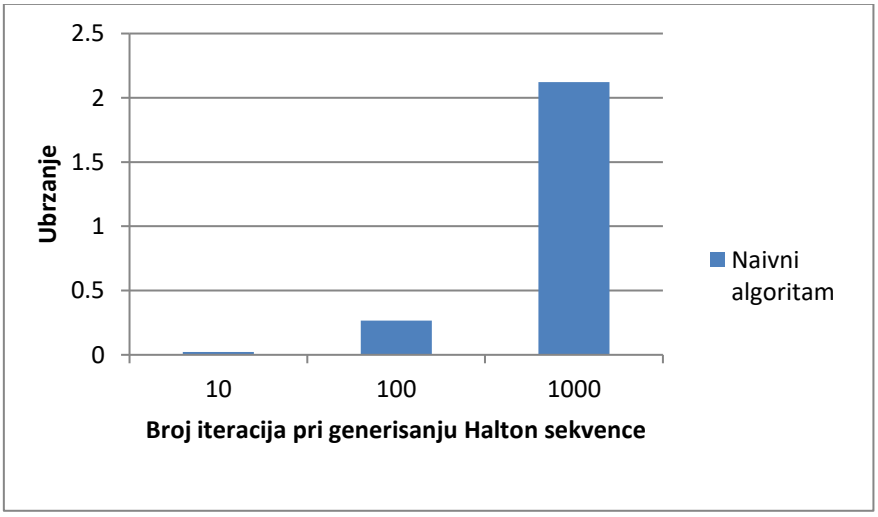
Izvršavanje komande ./halton 100

```
Sequential implementation execution time: 2.268099s
Parallel implementation execution time: 1.052453s
Test PASSED
```

Izvršavanje komande ./halton 1000

2.3.2. Grafici ubrzanja

U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja iteracija pri generisanju Halton sekvence

2.3.3. Diskusija dobijenih rezultata

Vidimo da je postignuto duplo ubrzanje za parametar od 1000, dok za ostale imamo usporenja jer su premali parametri za obradu na GPU. Svakako su svi parametri dosta mali za obradu na GPU.

3. Problem 3 – Simulacija kretanja n tela (n-body problem)

3.1. Tekst problema

Paralelizovati program koji se bavi problemom n tela (n-body problem). Sva tela imaju jediničnu masu, trokomponentni vektor položaja (x, y, z) i trokomponentni vektor brzine (vx, vy, vz). Simulaciju n tela se odvija u iteracijama, pri čemu se u svakoj iteraciji izračunava sila kojom sva tela deluju na sva ostala, a zatim se brzine i koordinate tela ažuriraju prema II Njutnovom zakonu. Brzine i položaji su slučajno generisani na početku simulacije. Zbog same prirode numeričke simulacije uveden je parametar SOFTENING, koji predstavlja korektivni faktor prilikom izračunavanja rastojanja između čestica (kako je gravitaciona sila obrnuto proporcionalna rastojanju između čestica, za nulta rastojanja i rastojanja bliska nuli, izračunata gravitaciona sila postaje izuzetno velika – teži beskonačnosti).

Program se nalazi u datoteci direktorijumu nbodymini u arhivi koja je priložena uz ovaj dokument. Program koji treba paralelizovati nalazi se u datoteci nbody.c. Pored samog izračunavanja, program čuva rezultate svake iteracije u zasebnim datotekama (za svako telo se čuvaju pozicije i brzine), dok kod show_nbody.py kreira gif same simulacije.

Skripta run pokreće simulaciju za različite parametre, i nakon toga, za određene simulacije poziva python kod koji kreira gifove.

3.2. Delovi koje treba paralelizovati

3.2.1. Diskusija

Vrlo je pogodno paralelizovati ovde spoljašnju petlju pošto su iteracije međusobno nezavisne.

```
void bodyForce(Body *p, float dt, int n)
{
    for (int i = 0; i < n; i++)
    {
        float Fx = 0.0f;
        float Fy = 0.0f;
        float Fz = 0.0f;

        for (int j = 0; j < n; j++)
        {
            float dx = p[j].x - p[i].x;
            float dy = p[j].y - p[i].y;
            float dz = p[j].z - p[i].z;
            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3;
            Fy += dy * invDist3;
            Fz += dz * invDist3;
        }

        p[i].vx += dt * Fx;
        p[i].vy += dt * Fy;
        p[i].vz += dt * Fz;
    }
}
```

Ovo je takođe jedina paralelizacija petlje koja nam je donela ubrzanje programa za veći broj nebeskih tela.

Pomislili smo i da paralelizujemo ovde spoljašnju petlju.

```
for (int iter = 0; iter < nIters; iter++)
{
    bodyForce(p, dt, nBodies);

    saveToCSV(p, nBodies, iter, folder);

    for (int i = 0; i < nBodies; i++)
    {
        p[i].x += p[i].vx * dt;
        p[i].y += p[i].vy * dt;
        p[i].z += p[i].vz * dt;
    }
}
```

Međutim shvatili smo da tako ne bismo dobili korektne rezultate zato što je važno da se uvek prvo pozove funkcija bodyForce kako bi se ažurirale koordinate brzine svakog nebeskog tela, pa onda nakon toga da se ažuriraju koordinate položaja svih nebeskih tela. Jasno je da, kada bismo paralelizovali ovu spoljašnju petlju, ne bi bio garantovan ovakav sled izvršavanja jer bi bilo omogućeno da se dogodi da više niti istovremeno pozove funkciju bodyForce i pre nego što su ikakva ažuriranja koordinata položaja urađena od strane bilo koje niti, pa se tada uopšte ne bi dobilo isto pomeranje nebeskih tela kao u sekvencijalnoj implementaciji problema, već bismo imali nedeterminističko kretanje nebeskih tela. Ovo naravno ne želimo jer je neophodno da paralelna implementacija problema ima iste rezultate kao i sekvencijalna kako bi paralelizacija uopšte imala smisla.

3.2.2. Način paralelizacije

Paralelizacija je odrađena tako što je svaka nit zadužena za po jedno nebesko telo i preračunava nove koordinate brzine u svakoj iteraciji najspoljnije for petlje.

3.3. Rezultati

3.3.1. Logovi izvršavanja

```
Sequential implementation execution time: 0.007179s
Parallel implementation execution time: 0.092211s
Test PASSED
```

Izvršavanje komande ./nbody 30 100 simulation_1

```
Sequential implementation execution time: 0.036442s
Parallel implementation execution time: 0.136696s
Test PASSED
```

Izvršavanje komande ./nbody 30 1000 simulation_2

```
Sequential implementation execution time: 2.667426s
Parallel implementation execution time: 0.344520s
Test PASSED
```

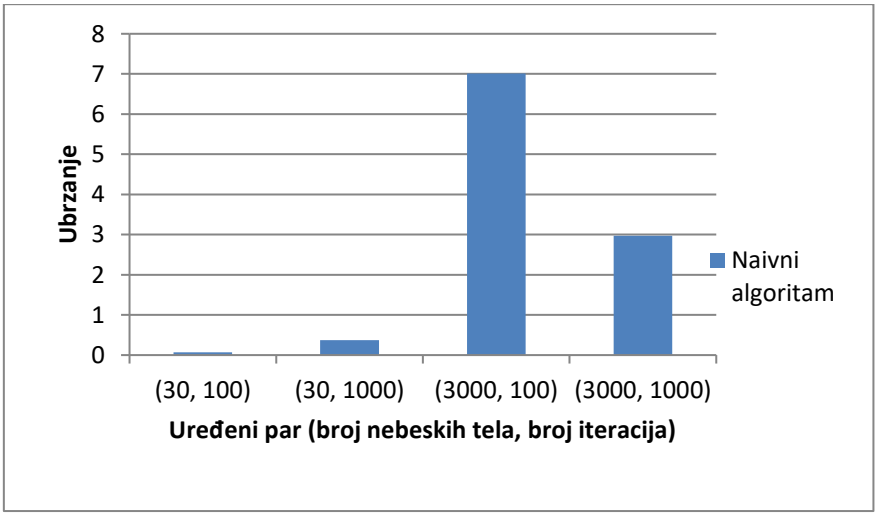
Izvršavanje komande ./nbody 3000 100 simulation_3

```
Sequential implementation execution time: 26.782410s
Parallel implementation execution time: 8.045561s
Test PASSED
```

Izvršavanje komande ./nbody 3000 1000 simulation_4

3.3.2. Grafici ubrzanja

U okviru ove sekcije su dati grafici ubrzanja u odnosu na sekvencijalnu implementaciju.



Slika 1. Grafik zavisnosti ubrzanja naivnog algoritma od broja nebeskih tela i broja iteracija

3.3.3. Diskusija dobijenih rezultata

Očekivano, za veći broj nebeskih tela (3000) imamo dobra ubrzanja, dok za manji broj (30) imamo usporenje jer je premali parametar da bi se isplatila paralelizacija na GPU.