Sveučilište u Rijeci – Odjel za informatiku

Preddiplomski jednopredmetni studij informatike

Problem N tijela

Završni rad

Student: Daniel Peruško

Mentor: dr. sc. Vedran Miletić

Kolegij: Paralelno programiranje na heterogenim sustavima

Akademska godina: 2020. / 2021.

Rijeka, 17.02.2021.

Sadržaj

1. Uvod

Problem N tijela je poznati problem u astronomiji. Problem predviđa pomak svakog tijela na kojeg utječe sila gravitacije ostalih tijela u okolini. Za mali broj tijela nije problem izračunati njegove karakteristike. Većim brojem tijela raste i kompleksnost izračuna, time i puno duže vrijeme izvođenja obrade. U današnje vrijeme potreba za bržom obradom podataka je sve veća. Obradu možemo ubrzati na dva načina, optimizacijom obrade ili unaprjeđenjem hardvera. Dok optimizaciju možemo vršiti do jedne granice, hardver možemo uvijek unaprijediti.

2. Osnovne značajke problema

Za simulirati problem n tijela potrebno je puno parametra, od kojih su osnovni:

- Masa
- Pozicija (koordinate)
- Brzina
- Udaljenost između svakog tijela

Sila kojom tijelo ***j*** djeluje na tijelo ***i*** definirano je Newtonovom jednadžbom gravitacije:

$$\mathbf{F}_{ij} = \frac{Gm_i m_j}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|^2} \cdot \frac{\left(\mathbf{q}_j - \mathbf{q}_i\right)}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|} = \frac{Gm_i m_j \left(\mathbf{q}_j - \mathbf{q}_i\right)}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|^3}$$

Gdje je G gravitacijska konstanta, a $||q_j - q_i||$ udaljenost između tijela.

Sumiranjem svih masa tijela vrijedi:

$$m_i \frac{d^2\mathbf{q}_i}{dt^2} = \sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{Gm_i m_j \left(\mathbf{q}_j - \mathbf{q}_i\right)}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|^3} = -\frac{\partial U}{\partial \mathbf{q}_i}$$

Gdje je U potencijalna energija tijela.

$$U = -\sum_{1\leq i<j\leq n} \frac{Gm_i m_j}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|}.$$

Postoje još neke jednadžbe koje su malo manje bitne.

Zbog velike kompleksnosti problema, u našem primjer napraviti ćemo pojednostavljen model te ga simulirati.

3. Paralelna obrada

3.1.Šta je paralelna obrada?

Paralelno procesiranje je način obrade podataka na način da se jedan veći proces podijeli na više manjih procesa. Time se postiže veća efikasnost i brža obrada podataka. Nije svaki proces prikladan za paralelnu obradu. Paralelna obrada podataka će biti najefektivnija kada se radi o obradi velikog broja podataka koje treba obraditi na isti način, kao na primjer operacije nad velikim matricama.

3.2.Kako radi paralelna obrada?

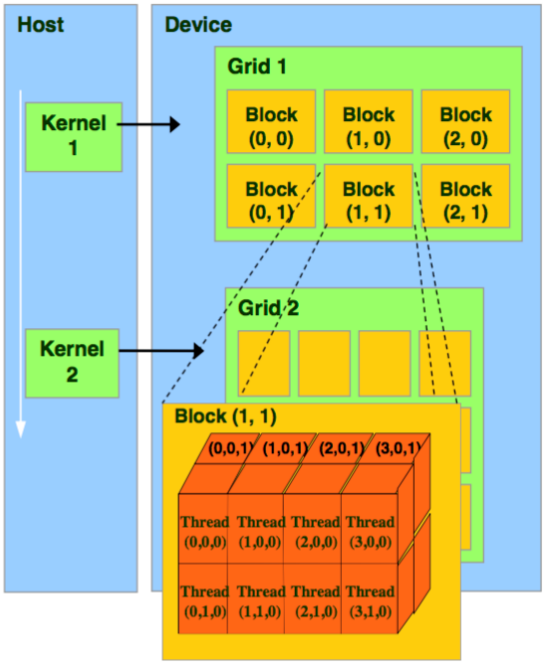
Prije nego šta uopće počinjemo sa obradom podatak, potrebno je definirati neke postavke. Svaka procesna nit ima svoji jedinstven index.

*Index = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x*

Blok čini skup procesnih niti. Broj niti u bloku je limitirana ovisno o modelu grafičke kartice, dok broj blokova tehnički nije limitiran. Svaka grafička kartica može pokrenuti neki broj blokova istovremeno. Naprednije grafičke kartice mogu pokretati više blokova istovremeno, šta ju čini bržom. Skup blokova čine grid.

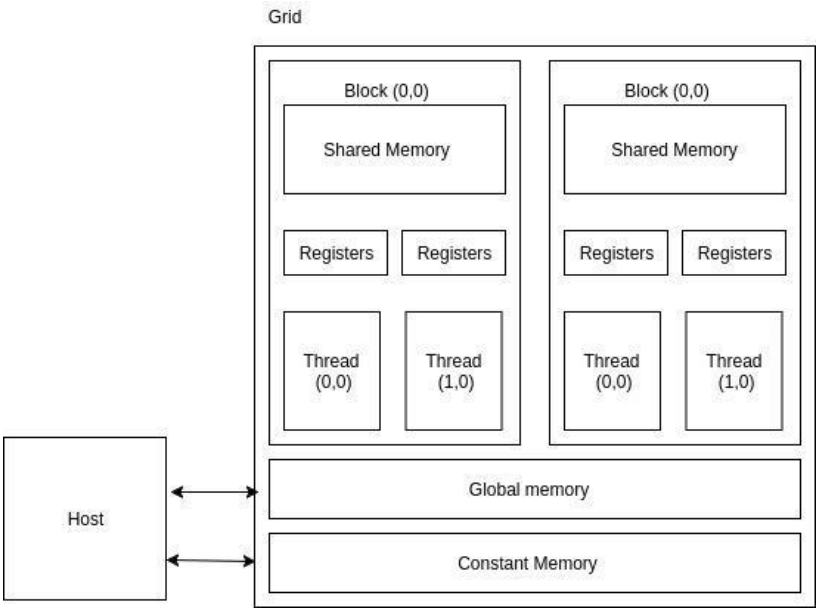
Dim3 je trodimenzionalna struktura ili vektor sa 3 parametra x, y i z. Bitno za napomenuti je da umnožak ne smije prekoračiti veličinu bloka.

dim3 threads(256); // x = 256, y = 1, z = 1
dim3 blocks (100,100); // x = 100, y = 100, z = 1
dim3 blocks (8,8,8); // x = 8, y = 8, z = 8;



Pri obradi podataka koristit će se više vrsta memorija, vrste memorija su:

1. Globalna memorija – Glavna memorija grafičke kartice
2. Konstantna memorija – prijenos podataka od hosta, podaci se tokom obrade ne mijenjaju
3. Registri – privatna memorija svake procesne niti
4. Privatna memorija – koriste ju procesne niti unutar bloka



Obrada počinje tako da se alocira memorije na domaćinu i grafičkoj kartici. Podatke za obradu je potrebno kopirati ga grafičku karticu. Nakon kopiranja podataka, procesne niti počinju paralelno obrađivati podatke. Obrađene podatke vraćamo u radnu memoriju domaćina. Paralelna obrada je ovime završena.

3.3. Prednosti i nedostatci paralelne obrade

Prednosti	Nedostatci
• Rješava veće probleme u manje vremena • Prikladan za simulacije • Ušteda novca na duže vrijeme	• Potreba za poznavanje arhitekture • Nije efikasno za obradu manjih količina podataka • Potreba za kopiranje podataka na grafičku karticu i obrnuto • Potreba za optimizaciju

4. Rješavanje problema

Da bi nam kod radio trebamo pozvati sljedeće biblioteke.

```
#include "hip/hip_runtime.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

Definiramo veličinu bloka.

```
#define BLOCK_SIZE 1024
```

Definiramo strukturu tijela.

```
struct Body{
   float x, y, z, vx, vy, vz;
};
```

Definiramo funkcije min() i max() sa parametrima koje spremaju minimalno i maksimalno vrijeme u varijable minT i maxT.

```
void min(double time, double &minT){
   if(time<minT)
      minT = time;
}

void max(double time, double &maxT){
   if(time>maxT)
      maxT = time;
}
```

Definiramo funkciju randBodies() sa parametrima koja početne vrijednosti svakog tijela nasumično postavi.

```
void randBodies(float *data, int n) {
  for (int i = 0; i < n; i++) {
    data[i] = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
  }
}
```

Definiramo funkciju bodyForce() sa parametrima. Ova funkcija će se paralelno izvršavati na svakoj procesnoj niti. Varijabla *i* je jedinstven indeks procesne niti. Slijedi algoritam „*n tijela*“, varijable *rx, ry, rz* su količina energije u toj osi i postavljaju se na 0. Definiramo petlju koja će se ponavljati n puta i služi nam da usporedimo svako tijelo ***j*** sa ***i*** tijelom. Varijable dx, dy, dz su nam udaljenosti između dva tijela u određenim osima. Izračunavamo pravu udaljenost u 3d prostoru, te sumiramo količinu energije. Pri kraju definiramo brzinu tijela u određenim osima.

```
__global__
void bodyForce(Body *p, float dt, int n) {
  int i = hipBlockDim_x * hipBlockIdx_x + hipThreadIdx_x;
  if (i < n) {
    float rx = 0.0f;
    float ry = 0.0f;
    float rz = 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;
      float invDistCube = 1.0f / sqrtf(distSqr);
      float invDist3 = invDistCube * invDistCube * invDistCube;

      rx += dx * invDist3;
      ry += dy * invDist3;
      rz += dz * invDist3;

    }

    p[i].vx += dt*rx;
    p[i].vy += dt*ry;
   p[i].vz += dt*rz;
  }
}
```

U funkciji main() sa argumentima, definiramo varijablu „nBodies“ i postavljamo je na 0. Ako pri pokretanju programa unesemo broj tijela kao argument, varijabla „nBodies“ se postavlja na taj broj, inače program će se ponoviti 3 puta i postaviti varijablu „nBodies“ na 50000, 150000 i 350000.

```
int main(const int argc, const char** argv) {
  int nBodies = 0;
  for(int r = 0; r<3;r++){
    if(r == 0){
      nBodies = 50000;
    }
    else if (r == 1){
      nBodies = 150000;
    }
    else{
      nBodies = 350000;
    }
    double minT = 10000.0, maxT = 0.0;
    if (argc > 1){
      nBodies = atoi(argv[1]);
      r = 3;
    }
```

Definiramo varijablu „dt“ na 0.01 kao vremenski korak i varijablu „iterations“ na 10. Ona nam definira koliko iteracija će program napraviti u jednom segmentu.

```
const float dt = 0.01f;
const int iterations = 10;
```

Definiramo varijablu „allocBytes“ koja nam označava veličinu potrebne alokacije i postavljamo je na umnožak broja tijela (*nBodies*) i veličinu strukture *„Body“* (sizeof(Body)). Slijedi alokacija memorije u radnoj memoriji domaćina (RAM), definiramo pokazivač „buf“ koji pokazuje na alociranu memoriju domaćina. Definiramo pokazivač „p“ strukture „Body“ koji pokazuje na pokazivač „buf“.

```
int allocBytes = nBodies*sizeof(Body);
float *buf = (float*)malloc(allocBytes);
Body *p = (Body*)buf;
```

Zatim definiramo pokazivač „d_buf“, alociramo memoriju na grafičkoj kartici veličine „allocBytes“. Definiramo pokazivač „d_p“ strukture „Body“ koji pokazuje na pokazivač „d_buf“.

```
float *d_buf; //
hipMalloc(&d_buf, allocBytes);
Body *d_p = (Body*)d_buf;
```

Funkcija „randBodies()“ sa parametrima nam postavlja početne vrijednosti tijela na nasumične vrijednosti.

```
randBodies(buf, 6*nBodies);
```

Definiramo varijablu „grid“, čija se vrijednost izračuna po formuli za podjednako opterećenje, varijabla „maxN“ nam izračunava najveći broj procesnih niti ovisno o veličini bloka i grida, te varijabla „timesum“ za izračunavanje sveukupnog vremena izvođenja segmenta i postavljamo ga na 0. Ispisujemo osnovne informacije o postavkama.

```
int grid = (nBodies + BLOCK_SIZE - 1) / BLOCK_SIZE;
int maxN = BLOCK_SIZE*grid;
double timesum = 0.0;
printf("Number of body's: %d\nBlock size: %d\nGrid size: %d\nMaximum n in with this setup (Block * Grid): %d\nThreds not used: %d\nUsed threds: %d %%\nNumber of iterations: %d\n\n",nBodies, BLOCK_SIZE, grid, maxN, maxN-nBodies, nBodies*100/maxN, iterations);
```

Ispisujemo osnovne informacije o postavkama. Petlju ponavljamo ovisno o vrijednosti „iterations“, u ovom slučaju 10 puta. Za svaku iteraciju definiramo varijablu „start“ koja nam sprema trenutno vrijeme. Pozivamo funkciju „hipMemcpy()“ sa parametrima. Funkcija kopira podatke sa domaćina na grafičku karticu, kako je parametrom definirano. Nakon toga počinje obrada podatak na grafičkoj kartici paralelno. Obradu započinjemo pozivanjem funkcije hipLaunchKernelGGL() sa parametrima. Nakon šta se obrada podataka izvrši, potrebno je vratiti obrađene podatke iz memorije grafičke kartice na memoriju domaćina. Funkcija „hipMemcpy()“ sa odgovarajućim parametrima nam to omogućuje. Ažuriramo poziciju svakog tijela pomoću prije izračunatim podacima. Definiramo varijablu “timediff” koja nam sprema vrijeme obrade podataka. Pozivamo funkcije min() i max() sa parametrima. Varijabli “timesum” nadodajemo vrijeme obrade podataka. Ispisujemo podatke o broju iteracije I proteklom vremenu.

```
for (int i = 1; i <= iterations; i++) {
        int start = clock();
        hipMemcpy(d_buf, buf, allocBytes, hipMemcpyHostToDevice);
        hipLaunchKernelGGL(bodyForce, dim3(grid), dim3(BLOCK_SIZE), 0, 0, d_p, dt, nBodies);
        hipMemcpy(buf, d_buf, allocBytes, hipMemcpyDeviceToHost);

        for (int j = 0 ; j < nBodies; j++) {
        p[j].x += p[j].vx*dt;
        p[j].y += p[j].vy*dt;
        p[j].z += p[j].vz*dt;
        }
        const double timediff = (clock()-start) / 1000000.0;
        min(timediff, minT);
        max(timediff, maxT);
        if (i > 1) {
                timesum += timediff;
        }
        printf("Iteration %d: %.5f seconds\n", i, timediff);
}
```

Definiramo varijablu “avgT” koja će izračunati prosječno vrijeme iteracije. Zatim ispišemo informacije o obrađenim podatcima.

```
double avgT = timesum / (double)(iterations-1);

printf("\nMinimum time: %0.5f seconds, Maximum time: %0.5f seconds, Average time:
%0.5f\n",minT, maxT, avgT);
printf("%d Bodies: average %0.0f Operations / second\n\n\n", nBodies, round(nBodies / timesum));
```

Pri kraju potrebno je osloboditi radnu memoriju domaćina i grafičke kartice pomoću funkcija free() i hipFree().

```
      free(buf);
      hipFree(d_buf);
   }
}
```

5. Analiza rješenja

Kompajliramo datoteku nbody.cpp pomoću "*HIP: AMD ROCm hipcc kompajlera. U našem slučaju:*

/opt/rocm/bin/hipcc -I/opt/rocm/include -g /home/pphs23/nBody/hip/nbody.cpp -o /home/pphs23/nBody/hip/nbody -lfmt -I/usr/include/eigen3 -L/opt/rocm/lib -lrocrand -lrocblas -lrocsolver -lrocsparse -lrocalution -lrocfft

Program pokrećemo sa naredbom *nbody* sa parametrom *150000* u terminalu.

>>pphs23@ares:~$ ./nBody/hip/nbody 150000

Program nam ispiše sljedeći kod.

Number of body's: 150000
Block size: 1024
Grid size: 147
*Maximum n in with this setup (Block * Grid): 150528*
Threds not used: 528
Used threds: 99 %
Number of iterations: 10

Iteration 1: 0.42397 seconds
Iteration 2: 0.19711 seconds
Iteration 3: 0.19699 seconds
Iteration 4: 0.19687 seconds
Iteration 5: 0.19768 seconds
Iteration 6: 0.19668 seconds
Iteration 7: 0.19704 seconds
Iteration 8: 0.19665 seconds
Iteration 9: 0.19690 seconds
Iteration 10: 0.19714 seconds

Minimum time: 0.19665 seconds, Maximum time: 0.42397 seconds, Average time: 0.19701
150000 Bodies: average 84599 Operations / second

Prvi odlomak nam ispisuje broj tijela simulacije, veličinu bloka, veličinu grida, najveći broj procesnih niti, broj neiskorištenih procesnih niti, postotak korištenih niti te broj iteracija. Zatim se ispisuje broj iteracije i vrijeme izvođenja te iteracije. Zadnji odlomak nam ispisuje vrijeme najkraće iteracije , vrijeme najduže iteracije i vrijeme prosječne iteracije. Pri kraju nam se ispisuje prosječan broj operacija po sekundi ovisno o broju tijela.

6. Usporedba vremena obrade sa drukčijim konfiguracija

ROCm i CUDA imaju veoma sličan način rada. Vremenski ćemo usporediti rješavanje problema na oba poslužitelja. Prevođenje koda poslužitelja sa ROCm na CUDA-u je poprilično jednostavno. Provesti ćemo mjerenje vremena izvođenja 10 iteracija na svakom poslužitelju.

Specifikacije ROCm poslužitelja:

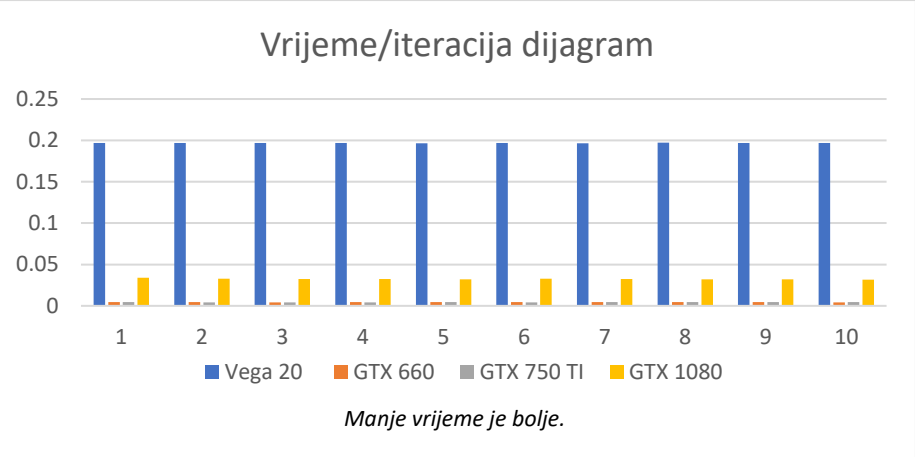
Vega 20 [Radeon VII]

Specifikacije CUDA-a poslužitelja:

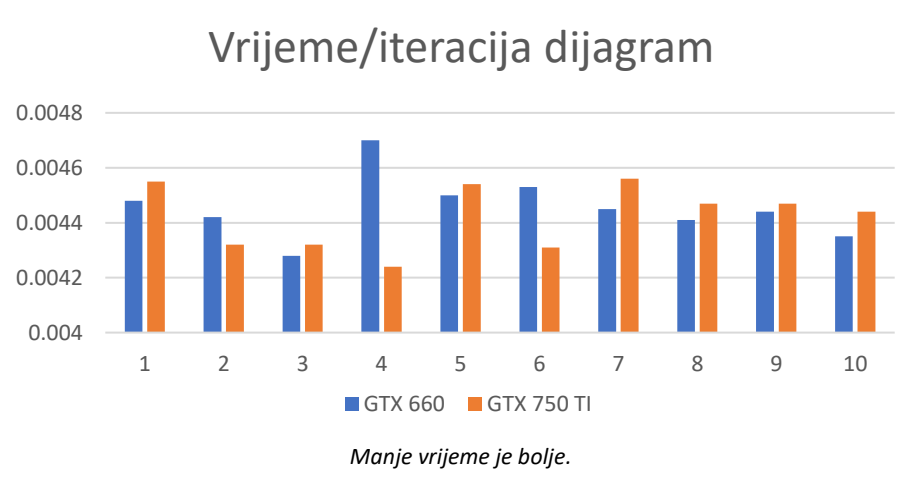
GeForce GTX 660

GeForce GTX 750 Ti

GeForce GTX 1080



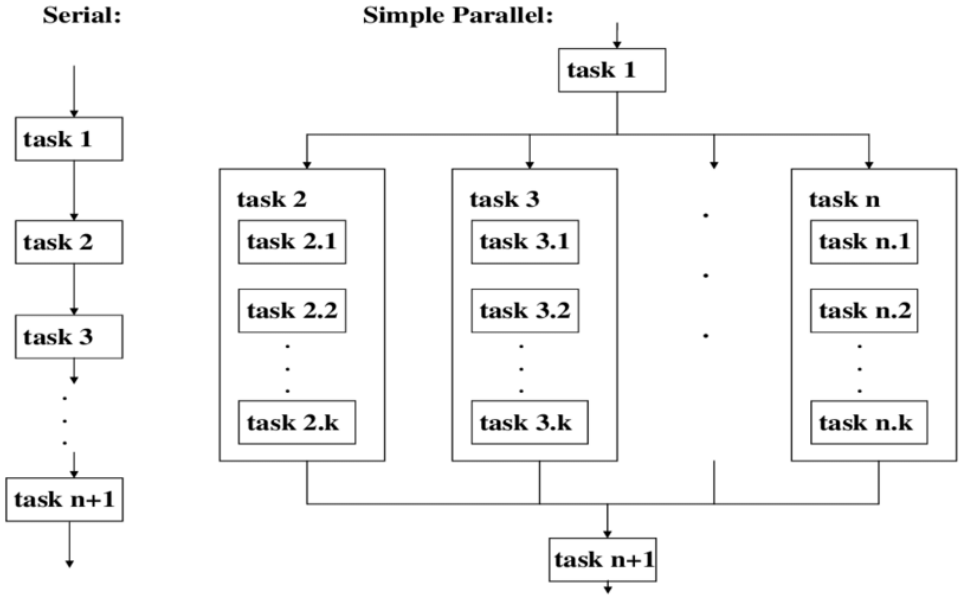
Manje vrijeme je bolje.



Manje vrijeme je bolje.

6.1. Usporedba serijske i paralelne obrade

Serijska obrada je način obrade kod koje se sljedeća operacija izvodi pri završetku prijašnje. Provođenje algoritma na serijski način nam omogućuje usporedbu vremenskog izvođenja kod i efikasnost paralelne obrade.



Korišteni procesori sa prosječnim vremenom izvođenja:

- Intel i7 6700k (4 jezgri, 8 procesnih niti) - 103.281s
- AMD Ryzen 7 2700X (8 jezgri, 16 procesnih niti) - 29.1365s
- AMD FX(tm)-6100 (6 jezgri, 6 procesnih niti)- 68.409s

Vrijeme izvođenja programa ovisi nekoliko parametra, od kojih su najosnovniji broj jezgri, broj procesnih niti po jezgri te brzina jezgre. Ako usporedimo dobivena rješenja najmanje vrijeme izvođenja ima AMD Ryzen 7 2700X procesor, koji ima najveći broj jezgra i procesnih niti, šta je logično.

Usporedimo li vrijeme izvođenja serijske i paralelne obrade, paralelna obrada je puno efektivnija, u našem slučaju od 147 puta do 524 puta brže.

7. Zaključak

U današnje vrijeme najnoviji i najefikasniji način za obradu podataka je paralelna obrada. Paralelna obrada nam omogućuje rješavanje kompleksnih problema za pretraživanje, sortiranje, razne algoritme itd. Bitno je spomenuti da paralelna obrada nije rješenje za svaki problem, ali ukoliko se ona može implementirati, isplati se.

U ovom radu opisali smo problem n tijela, objasnili šta je paralelna obrada i kako ona radi, napisali smo program koji rješava problem n tijela pomoću paralelne obrade, analizirali rješenja, te usporedili rješenja raznih konfiguracija.

8. Literatura

Internet:

1. https://lab.miletic.net/hr/nastava/kolegiji/PPHS/#kolegij-pphs
2. https://docs.nvidia.com/cuda/samples/5_Simulations/nbody/doc/nbody_gems3_ch31.pdf
3. https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-n-body-simulation
4. https://www.olcf.ornl.gov/tutorials/cuda-vector-addition/
5. https://www.youtube.com/watch?v=cuCWbztXk4Y
6. https://www.britannica.com/science/science
7. https://aip.scitation.org/doi/pdf/10.1063/1.4822898
8. https://www.geekboots.com/story/parallel-computing-and-its-advantage-and-disadvantage
9. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html