Numeričke simulacije realnog plina na grafičkoj kartici

Balen, Rajko

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO-MATEMATIČKI FAKULTET
FIZIČKI ODSJEK

SMJER: PROFESOR FIZIKE I INFORMATIKE

Rajko Balen

Diplomski rad

Numeričke simulacije realnog plina na grafičkoj kartici

Voditelj diplomskog rada: Ivo Batistić

Ocjena diplomskog rada: _____

Povjerenstvo: 1. _____

2. _____

3. _____

Datum polaganja: _____

Zagreb, 2016.

Sažetak

U ovom diplomskom radu izvodimo numeričke simluacije realnog 2D plina kao pogodnog primjera na kojem se može ispitati postizavanje što boljih numeričkih performanse u višejezgrenim računalima i na grafičkoj kartici. Radi se o simulacijama težine O(n^2) koje traže jaki hardver.

Numerical simulation of real gas on GPU

Abstract

In this diploma thesis we perform numerical simulations of 2D real gas as a suitable example to explore and to accomplish the best numerical performance in multicore computers and on a graphical computing unit (GPU). This simulation is of O(n^2) complexity and requires a powerfull hardware.

Sadržaj

1 Uvod

Zašto baš grafičke kartice? Grafičke kartice su u startu građene da mogu paralelno iscrtavati veliki broj točkica na zaslonu računala. Njihov razvoj i potreba za dodatnim performansama je uglavnom uvjetovala industrija igara. Za razliku od procesora one su specifične po tome što rade na nižoj frekvenciji od otprilike 1 GHz. Snaga procesora a time i njegova toplinska disipacija rastu s trećom potencijom frekvencije[*]. Što znači da je moguće u stroj koji radi na nižoj frekvenciji ugraditiveći broj računskih jedinica, a da se time ne promjeni potrošnja energije i maksimalna temperatura takvog uređaja.

Grafičke kartice isto tako imaju više manje jednoličan modularni dizajn. Njihov čip je sastavljen od više istih ili sličnih elemenata za razliku od tradicionalnih procesora koji imaju mnogo različitih elemenata na svom čipu i generalno su mnogo kompleksniji i skuplji. Ova jednostavnost dizajna osnovnih elemenata grafičke kartice omogućuje da se oni lako dizajniraju i da se na grafički čip upakira 2 reda veličine veći broj aritmetičko logičkih jedinica za istu cijenu kao kod tradicionalnih procesora.

Rezultat toga je 1792 aritmetičko logičke jedinice u Radeonu 280 naspram 12 do 16 aritmetičko logičkih jedinica u četverojezgrenim i7 procesorima. Oba uređaja su približno iste cijene oko 370 Američkih dolara. Zbog ovoga s grafičkom karticom se dobije oko dva reda veličine više Gflops-a po dolaru.

Postoji analogija koja bi objasnila što je grafička kartica. Ako su 4 jezgre tradicionalnog procesora 4 inženjera fizike onda je grafička kartica ekvivalentna stotinama gimnazijalaca koji nemaju znanje inženjera, ali gledajući ih kao grupu u kojoj svaki zasebno rješava relativno jednostavne matematičke probleme onda oni imaju veću brzinu računanja. Problemi se mogu rješavati zasebno ukoliko su oni međusobno nezavisni.

Iako grafičke kartice nisu ništa novo, njihova uporaba u računanju opće namjene je popularizirana s pojavom OpenCL-a od AMD-a/ATI-ja, CUDA-e od Nvidije te DirectCompute-a baziranog na DirectX-u. To su tri softverske platforme koje omogućavaju GPU programiranje u već standardnim jezicima poput C, C++, Java, Python itd. U biti danas skoro svaki jezik ima biblioteku koji omogućava da se kôd iz tog jezika izvršava na grafičkoj

[*] Porast potrošnje električne energije raste linearno s frekvencijom prema formuli $P = fCV^2$, međutim da bi frekvencija procesora rasla linearno potrebno je linearno podizati napon. $V(f) \propto f$ pa je formula za snagu ekvivalentna $P \propto fCf^2 \propto f^3$. U realnom slučaju potencija varira od 2.5-3.5 ovisno o frekvenciji i kvaliteti izrade čipa.

kartici.Postojale su jako kratko kartice koje su računale fiziku[*] u igrama. Ideja je bila da kupite grafičku karticu te još jednu dodatnu karticu za izračun fizike kako bi igra bila fizikalno realističnija. Programerska zajednica zbog nesigurnosti, a time itržište zbog dodatnih troškova, nisu dugo podržavali ovu tehnologiju. Umjesto toga postojeće grafičke kartice su preuređene u strojeve za numeričke račune pa tako i za izračun fizike u igrama. Nvidia i AMD su jako mnogo uložili u razvoj OpenCL-a i CUDA-e i za hardversku prilagodbu svojih grafičkih uređaja za izvođenje numeričkog koda. Garancija isplativosti njihove investicije je prije spomenuta industrija igara koja je podržavala i razvoj GPU hardwera.

Kartice koje podržavaju OpenCL pojavile su se krajem 2009. Ono je time ponudilo tržištu HPC-a (High Performance Computing) ali i običnih korisnika performanse superračunala za male novce. Iz nekog razloga korištenje grafičkog hardvera još uvijek nije standard kod nabave superračunala.

[*] U nekim se igrama radi realističnosti uvelo simuliranje fizike elemenata iz te igre poput padanja kutija. Neko vrijeme je za ovu funkcionalnost bio potreban poseban hardver.

2 Softver

Prije spomenuti OpenCL, CUDA i Direct Compute su platforme ili API-ji koji služe kao baza za pristup GPU hardveru i njihova sintaksa je slična C99 sintaksi.

OpenCL je poprilično težak za početnika ne samo zbog neobične i nepoznate programske sintakse nego i potrebe za poznavanjem rada GPU hardvera. Stoga su razvijene različite biblioteke (eng. *library*) koje više ili manje pojednostavljuju OpenCL kôd i tako daju programeru više ili manje kontrole nad sirovim OpenCL kodom. Neke od biblioteka su: JavaCL i JOCL za Javu, AMP za C (AMP koristi DirectCompute), PyCL i PyCUDA za python.

Mi ćemo naš program pisati u Javi. Aparapi je biblioteka kod koje se kôd za grafičku karticu piše u Javi i onda Aparapi taj kôd prevodi u OpenCL kôd koji se izvodi na grafičkoj kartici. Aparapi je jedan od najjednostavnijih načina da kôd pokrenete na grafičkoj kartici jer vas ne zamara s nekoliko vrsta memorije, kreiranjem konteksta i sličnim nestandardnim procedurama. Poprilično je automatiziran. Dokumentacija za Aparapi uključuje PDF za OpenCL i wiki stranicu koja ima dosta korisnih podataka ali i mnogo zastarjelih naputaka koji ponekad ne rade! Tako da je Aparapi osrednje dokumentiran s nekolicinom primjera na internetu i koji služe više kao odskočna daska za druge OpenCL/CUDA implementacije.

Java sama po sebi nije idealna za numeričko programiranje jer joj nedostaje kontrola nad podacima koja postoji u C-u i alati poput *prefetchera*[*].

Ipak java i Aparapi su za ovaj diplomski odabrani jer su se pokazali kao dobar kompromis između performanse i jednostavnosti programiranja.

[*] Objašnjeno u odjeljku 5.1.5 Prefetch /pretpreuzimanje na stranici 18.

3 Fizikalni model

U početku ne polazimo odmah od egzaktnog modela realnog plina, već krećemo od pojednostavljenog modela na kojem ćemo naučiti kako optimizirati program. Kada to savladamo napraviti ćemo realističniji model za 2D plin.

Jedna od stvari koje karakteriziraju realni plin je međudjelovanje čestica. Naš primitivni model ima sljedeće karakteristike: Čestice se pomiču u dvije dimenzije, ne sudaraju se jedna s drugom i sila koja djeluje na njih opada s $\frac{1}{r}$ što odgovara potencijalu $\ln(r)$. To nije pravi model realnog plina jer sila u realnom plinu ne opada tako. Osim toga čestice u realnom plinu imaju i odbojnu komponentu sile koju ovdje zanemarujemo. Ovaj model jako je sličan popularnom problemu n-tijela (*eng. N-body problem*) koji se standardno izvodi u 3 dimenzije i ima opadanje sile s $\frac{1}{r^2}$ to jest potencijal $\frac{1}{r}$. Problem n-tijela je bio predmet akademskih radova i jedan je od čestih *benchmarka* (hrv. test, mjerilo) kojima se testira hardver s više jezgara zbog toga što se lako paralelizira i zbog toga što jako dobro iskorištava procesni potencijal hardvera.

Zašto smo onda koristili ovaj pojednostavljeni model? Cilj nam je postizanje velike performanse, a lakše ćemo naučiti kako optimizirati program ukoliko je kôd koji se izvodi kratak i jasan. Model u kojem sila ovisi s $\frac{1}{r}$ ima vizualno potpuno drukčiji način gibanja čestica od bilo kojeg modela s većom potencijom na r. Na takvom modelu se lako vizualno uoče nekonzistencije ili greške u računu dok su modeli s većom potencijom na r više-manje vizualno isti.

Napravili smo također i nekoliko 3D modela radi usporedbe tih 3D implementacija u Javi i Aparapiju s implementacijama problema koji su napravljeni u drugim programima. Prelazak s 2D na 3D problem teoretski predviđa otprilike 50% više numeričkih operacija.

3.1 Model

U 2D plinu imamo čestice (točke) koje se nalaze u 2D prostoru određene s *x* i *y* koordinatom. One imaju brzinu u *x* i *y* smjeru. Ako imamo *n* čestica potrebno nam je minimalno *4n* memorijskih lokacija da bismo im opisali položaj i brzinu. Podatke spremamo u 4 polja tipa float[*].

```
float xx[n], yy[n], velxx[n], velyy[n] ;
```
(1)

Float je dovoljno dobra preciznost, a kada bi računali s dvostrukompreciznošću imali bi znatan gubitak performanse, pogotovo na grafičkoj kartici. Preciznost nam u ovoj simulaciji najviše ovisi o *dt* intervalu koji ima konačnu vrijednost. Pretpostavili smo da je masa svake čestice ista tako da nije potrebno zasebno polje za masu već se ona nalazi u konstanti.

Na svaku česticu djeluje sila od svake druge čestice. Znači ako imamo 100 čestica, jedna čestica trpi silu od ostalih 99. Ovo znači da kalkulaciju sile obavljamo *n*(*n-1*) puta. Rezultat ovog računa sile će biti izračun akceleracije na svaku česticu. A iz akceleracija se izračunaju promjene položaja čestica. Ako su čestice stavljene u kutiju one će se odbijati od zida te kutije. Za svaku od n čestica trebamo provjeriti je li se ona odbila od zida i ako jest korigirati joj brzinu i položaj. Ukupni broj kalkulacija je priložen u sljedećoj tablici.

Tabela 1

	Broj kalkulacija	**broj ciklusa**
Račun sile/akceleracije	$n(n-1) * FCalc * dim$	$FCalc \approx 7$
Račun novih brzina	$n * velCalc * dim$	$velCalc \approx 2$
Račun novih položaja	$n * posCalc * dim$	$posCalc \approx 2$
Račun sudara sa zidom	$n * wallColCalc * dim$	$wallColCalc \approx 3$

Iz tablice se vidi da broj operacija ovisi proporcionalno broju dimenzija te za silu/akceleraciju ovisi o n^2 to jest ima O(n^2) težinu, gdje je *n* broj čestica. Za ostale dijelove računa broj kalkulacija ovisi proporcionalno o *n* to jest ima O(*n*) težinu.

Zbog toga se ovaj problem svodi na optimizaciju računa sile/akceleracije.

Dakle imamo model sile u 2D prostoru koja opada s recipročnom vrijednosti udaljenosti.

$$\vec{F} = -m\frac{k}{r}\hat{r} = -m\frac{k\vec{r}}{r^2} \tag{1}$$

[*] float je tip podatka koji sprema decimalne brojeve i ima veličinu 4 bajta, double ima veličinu 8 bajta

gdje je k neka proizvoljna konstanta ovisna o masi i privlačnoj sili među česticama, a r udaljenost među česticama *i* i *j*.

$$\vec{F} = (F_x, F_y) \tag{2}$$

$$\vec{r} = (r_x, r_y) \tag{3}$$

x_i je položaj i-te čestice i u memoriji je spremljen kao *xx*[i] to jest i-ti element u *xx* polju.

$$r_x = (x_i - x_j) \qquad r_y = (y_i - y_j) \tag{4}$$

$$r^2 = (x_i - x_j)^2 + (y_i - y_j)^2 \tag{5}$$

Iz jednadžbi slijedi:

$$F_x = -m\frac{kr_x}{r^2} \qquad F_y = -m\frac{kr_y}{r^2} \tag{6}$$

Ako uzmemo u obzir izraze (4), (5) i (6) krajnji račun za doprinos akceleracije u *x* i *y* smjeru jest:

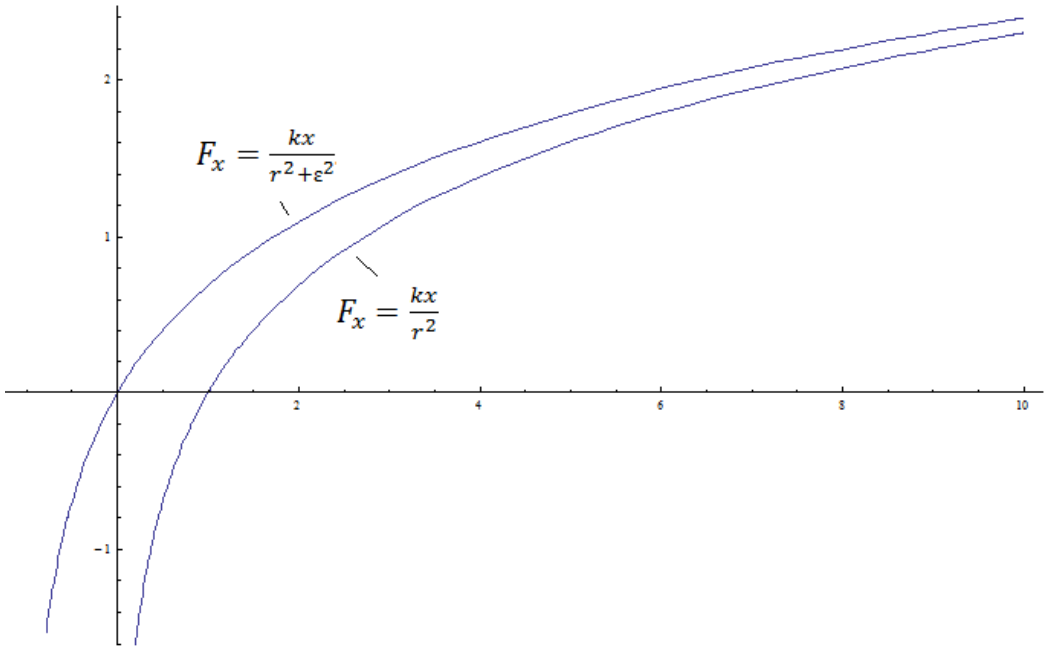
$$\begin{aligned} a_x &= \frac{F_x}{m} = -\frac{k(x_i - x_j)}{(x_i - x_j)^2 + (y_i - y_j)^2} \\ a_y &= \frac{F_y}{m} = -\frac{k(y_i - y_j)}{(x_i - x_j)^2 + (y_i - y_j)^2} \end{aligned} \tag{7}$$

Kada se dvije čestice jako približe imamo situaciju gdje one u relativno malim pomacima naglo mijenjaju akceleraciju jer prolaze kroz veliki gradijent potencijala. Da ne dolazi do nagle promjene akceleracije, a što može uzrokovati nestabilnost u simulaciji, uvodimo malu konstantu ε u nazivnik.

$$a_x = \frac{k(x_i - x_j)}{(x_i - x_j)^2 + (y_i - y_j)^2 + \varepsilon^2} \tag{8}$$

Na slici 1 možemo vidjeti razliku među potencijalima između modela $F_x = \frac{kx}{r^2}$ i $F_x = \frac{kx}{r^2+\varepsilon^2}$.

Slika 1



Ona osim što smanjuje gradijent potencijala oko nule također sprječava dijeljenje s nulom jer je ε uvijek veći od 0. Također sila=0 kada je $i=j$ tj. čestica ne djeluje sama na sebe. Jednadžba (8) je doprinos akceleracije zbog sile između dvije čestice i i j u x-smjeru. Ukupna akceleracija u x smjeru na neku i-tu česticu će biti suma svih doprinosa u x smjeru, tako da račun trebamo ponoviti n puta iterirajući po j. Naš a_x doprinos akceleracije je zapravo a_{xij}.

$$a_{xtot} = \sum_{\substack{j=0 \\ j\neq i}}^{j=n} a_{xij} \tag{9}$$

Isti postupak vrijedi i za y dimenziju.

Primijetimo da smo ovime izračunali akceleraciju samo na jednu i-tu česticu. Da bismo dobili rezultat za sve čestice proces trebamo ponoviti n puta iterirajući po i. Nakon što smo dobili akceleracije u x i y smjeru njih uvrštavamo u jednadžbu pomaka. Najjednostavniji primjer toga je:

$$\begin{aligned} v_x &= v_{x0} + a_{xtot}\Delta t \\ x &= x_0 + v_x\Delta t \end{aligned} \tag{10}$$

Ova jednadžba se izvršava n puta to jest jednom za svaku česticu.

Pogledajmo kôd koji bi sve ovo realizirao na slici 2.

Slika 2

```
public void BasicNbody(){
float deltx,delty,recDistanceSQ;
float acxtot,acytot;
        for(int i=0;i<number;i=i+1){
                acxtot=0;
                acytot=0;
                for(int j=0;j<number;j=j+1){

                        deltx=xx[i]-xx[j];
                        delty=yy[i]-yy[j];
                        recDistanceSQ=1/(deltx*deltx+delty*delty+e2);
                        acxtot=acxtot+deltx*recDistanceSQ*K;
                        acytot=acytot+delty*recDistanceSQ*K;

                }

                velxx[i]=velxx[i]+acxtot*deltat;
                velyy[i]=velyy[i]+acytot*deltat;
                xx[i]=xx[i]+acxtot*deltat;
                yy[i]=yy[i]+acytot*deltat;
        }
```

Račun sile/akceleracije — Ponavlja se n*n puta

Integrator — Ponavlja se n puta

Osnovni koncept izračuna akceleracija i ažuriranja brzina i položaja.

Račun sile akceleracije se ponavlja n^2 puta i sadrži 5 osnovnih linija kôda: udaljenost među česticama u x i y smjeru, račun recipročne udaljenosti koja može biti potencirana više puta ovisno o tome kakav potencijal sile koristimo te ažuriranje doprinosa akceleracije u privremene varijable *acxtot* (što je zapravo a_{xtot} iz jednadžbe (9)) i *acytot.* Ovaj dio kôda se nalazi u *while j* petlji. Izvan nje se nalazi ažuriranje brzine i položaja i taj dio se ponavlja n puta to jest jednom za svaku česticu.

Ovaj način računa ima manju grešku! Računi se za neke čestice izvrše nakon što su nekim česticama ažurirani novi položaji. Pretpostavka našeg računskog modela jest da sve čestice miruju dok se računa sila tako da bi se ažuriranje položaja trebalo vršiti u zasebnoj funkciji nakon kalkulacije akceleracija. Još ćemo kasnije diskutirati eventualne probleme i prednosti ovog načina računanja ali ukoliko je *deltat* dovoljno mali i ukoliko se napravi samo jedan pomak čestica ne bi trebalo biti većih problema za fizikalnu ispravnost simulacije.

Završni program ima oko 7000 linija i jasno je da nećemo ulaziti u analizu svih njih. Ovih 5-10 linija je temelj kôda koji se izvršava i bitno je da ga razumijemo. Izračune koji se ponavljaju, naprimjer *xx*[*i*]-*yy*[*i*], je uputno staviti u varijablu i pozivati je po potrebi jer ovim smanjujemo broj potrebnih kalkulacija i rasterećujemo aritmetičko logičke jedinice. Time zamjenjujemo ALU* (skraćeno od *Arithmetical Logical Unit*) operacije s memorijskom

* ALU operacija je naprimjer operacija zbrajanja: a=b+c

operacijom*. Niti ovo nije uvijek rješenje. Koji put nam je bolje ponovno računati stvari ukoliko je dohvat iz memorije predug. Većina n-body problema koji se mogu pronaći na internetu ima u osnovnom računu i račun mase. Taj problem smo preskočili odredivši da je masa svim česticama ista. Naš problem u unutarnjoj *j* petlji ima 11 kalkulacija ne računajući množenje s konstantom *k* koju ćemo kasnije izbaciti van petlje. Množenja i zbrajanja zahtijevaju otprilike isti broj ciklusa ali dijeljenje na CPU jezgri zahtjeva oko 5 puta više ciklusa nego množenje ili zbrajanje. Tako da je realan broj kalkulacija za svaki prolazak unutarnje petlje veći od 11. Ipak kada se govori o performansi operacija s pomičnim zarezom dijeljenje, množenje i zbrajanje podrazumijevamo kao jednu operaciju.

3.2 Integrator

Δt je vremenski korak kroz koji se izvršava kalkulacija. Manji Δt rezultira preciznijom kalkulacijom. Δt je u analitičkom rješenju ekvivalentan dt . Ako bismo prema jednadžbi (10) računali promjenu brzine i položaja u nekom intervalu od $t_1$do t_2 mi bismo zapravo integrirali jednadžbu (10) po dt od $t_1$do t_2. Kako računamo to isto samo na numerički način s konačno malim Δt možemo reći da numerički integriramo jednadžbu (10). Jednadžba (10) je stoga jedan primjer integratora. Zbog toga je uputno račun brzine i puta gledati kao zasebnu cjelinu. Postoji više različitih integratora od kojih je jednadžba (10) najjednostavniji. Poslije ćemo razmotriti i druge integratore to jest metode integracije.

* Memorijska operacija je naprimjer operacija čitanja iz RAM-a: a=xx[i]

4 Paralelizacija

Da bi se kôd mogao paralelno izvoditi treba ga prilagoditi. N-body problem je relativno lako paralelizirati jer su rezultati akceleracije $\vec{a}_{xtot}$ na svaku česticu međusobno nezavisni jedni od drugih. Zbog toga ih se može razdvojiti u nezavisne dretve (procese,niti eng. *thread*) koje se mogu u isto vrijeme izvršavati na nekoliko jezgara. Primjer problema koji se ne može paralelizirati je izračun n-tog elementa u Fibonaccijevom nizu. Za svaki član tog niza vrijedi $x_n=x_{n-1}+x_{n-2}$. Da biste izračunali 100. element Fibonaccijevog niza morate znati 99. i 98. element. Ukoliko u algoritmu postoji zavisnost rezultata o prijašnjim iteracijama u petlji on se ne može paralelizirati. Sila na neku *i*-tu česticu ovisi o položajima drugih čestica i ovisi o prijašnjem položaju te čestice, ali ona ne ovisi o silama na druge čestice! Iz ovog razloga se ovi računi mogu razdvojiti. Ovo možemo zaključiti i promatrajući kôd sa slike 2. Unutarnja petlja *j* radi *n* iteracija da bi dobila rezultat za neku *i*-tu česticu i onda taj rezultat sprema u neko polje na *i*-tu lokaciju. Sljedeća *i*+1 čestica ne treba rezultat od prethodne *i*-te čestice da bi se izračunala. Promotrimo našu novu funkciju.

for petlju smo zamijenili s *while* petljom. Prednost u sintaksi postat će jasna tek kasnije. Konstantu *k* smo odmah pomnožili u trećem retku *while j* petlje da je ne množimo dva puta u 4. i 5. retku i preimenovali je u *attractiveConstant*. Tehnički varijabla *recDistanceSQ* nije recipročni kvadrat duljine ali da ne uvodimo mnogo različitih naziva za jednu te istu liniju kôda ostavljamo nazive kakvi jesu. Osim što program radi malo brže promjene su zasada kozmetičke.

```
public void BasicNbody(){
        float recDistanceSQ,deltx,delty,acxtot,acytot;
        int j,i=0;
        while(i<number){
                acxtot=0;
                acytot=0;
                j=0;
                while(j<number){
                1//     deltx=xx[i]-xx[j];
                2//     delty=yy[i]-yy[j];
                3//     recDistanceSQ=attractiveConstant /(deltx*deltx+delty*delty+e2);
                4//     acxtot+=deltx*recDistanceSQ;
                5//     acytot+=delty*recDistanceSQ;
                        j=j+1;
                }
                velxx[i]+=acxtot*deltat;
                velyy[i]+=acytot*deltat;
                xx[i]+=velxx[i]*deltat;
                yy[i]+=velyy[i]*deltat;
```

(2)

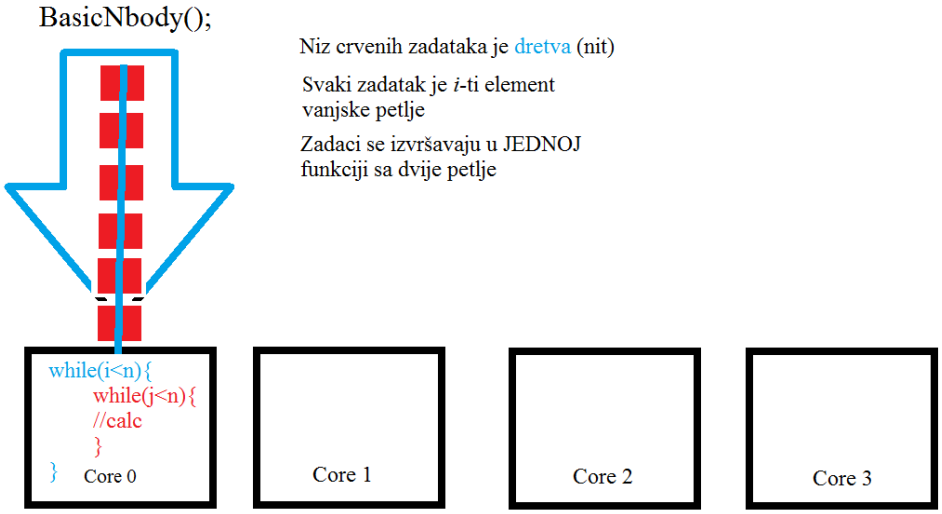
```
        }
    }
```

U glavnom programu ovu funkciju pokrećemo s dvije linije koda.

```
Kuglice svekuglice2= new Kuglice(); // kreiramo objekt s BasicNbody funkcijom
svekuglice2.BasicNbody();           //pokrenemo funkciju
```
(3)

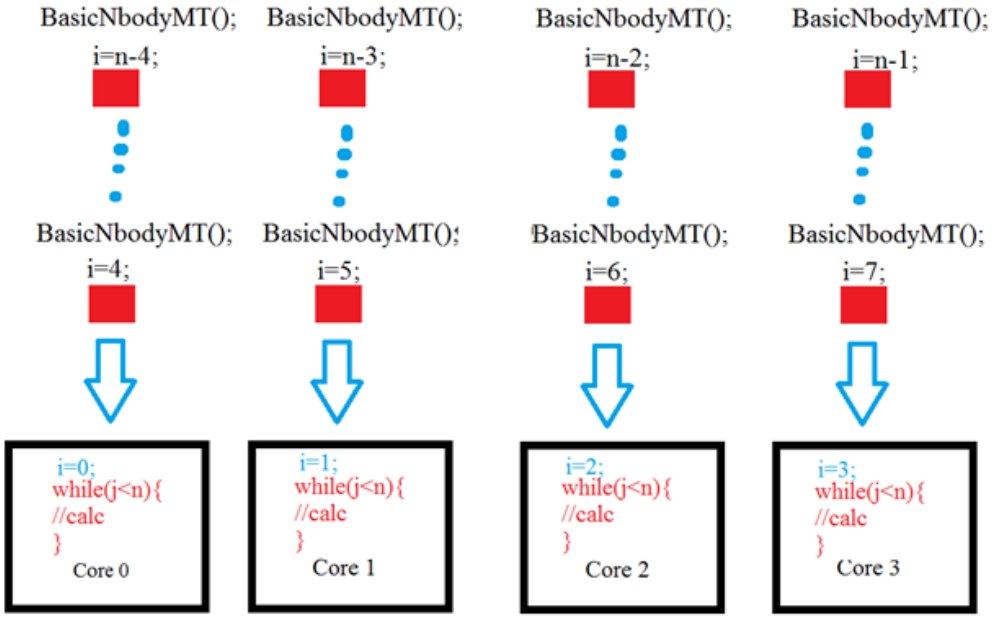
Ovaj kôd se zadano izvršava na jednoj jezgri i izvršava ga jedna *dretva*. Kako bi izgledala njegova paralelizirana verzija? Promotrimo shemu na slici 3 i 4:

Slika 3



Funkcija BasicNbody je pozvana JEDNOM i izvršava dvije ugniježđene petlje na jednoj jezgri od kojih je svaka veličine n.

Slika 4



Umjesto JEDNE funkcije s n crvenih zadataka imamo n INSTANCI funkcije. Svaka instanca ima samo jedan crveni zadatak i razlikuju se po iteratoru "i" koji jednoznačno određuje svaku instancu. Funkcija BasicNbodyMT se poziva n puta čime se stvara "n" INSTANCI i svaki poziv u sebi ima jednu "j" petlju veličine "n" te svaki poziv funkcije ide na drugu jezgru. Ovdje vidimo 4 dretve od kojih svaka ide na svoju jezgru.

Kôd funkcije *BasicNbodyMT* (gdje *MT* znači multithreaded) izgleda ovako:

```
public void BasicNbodyMT(){
        float recDistanceSQ,deltx,delty,acxtot,acytot;
        int j;
        int i=getGlobalId();
                acxtot=0;
                acytot=0;
                j=0;
                while(j<number){
                        deltx=xx[i]-xx[j];
                        delty=yy[i]-yy[j];
                        recDistanceSQ=attractiveConstant /(deltx*deltx+delty*delty+e2);
                        acxtot+=deltx*recDistanceSQ;
                        acytot+=delty*recDistanceSQ;
                        j=j+1;
                }
                velxx[i]+=acxtot*deltat;
                velyy[i]+=acytot*deltat;
                xx[i]+=velxx[i]*deltat;
                yy[i]+=velyy[i]*deltat;
}
```

(4)

Primijetimo da je sve isto osim što nema vanjske *while* petlje po *i* i nema brojača po *i.* To je zato jer se ovdje ništa ne iterira po *i* nego samo po *j.* Vanjska petlja je zamijenjena s

i=getGlobalId; Umjesto da iz glavnog programa jednom pozovemo *BasicNbody* funkciju s dvije ugniježđene petlje od kojih svaka ima *n* elemenata, sada imamo *n* poziva *BasicNbodyMT* funkcije s jednom petljom od *n* elemenata. Jedan poziv ove funkcije jest ono što je bila jedna iteracija po *i* u vanjskoj petlji ili jedan crveni zadatak. To je sada *i*-ta *instanca* nove funkcije. Naredba *getGlobalId();* određuje koja je to iteracija to jest instanca.

U nekoliko linija kôda u glavnom programu određujemo pozivanje ove funkcije i broj instanci.

```
final KugliceMT svekuglice2 = new KugliceMT();
svekuglice2.setExecutionMode(Kernel.EXECUTION_MODE.JTP);
svekuglice2.execute(number);
```
(5)

Prvo kreiramo objekt *svekuglice2* koji ima prethodno diskutiranu funkciju.

Zatim određujemo način izvođenja. Imamo 4 moguća načina:

SEQ : Procesor - sekvencijalno to jest jednodretveno izvođenje.

JTP: Procesor - Java Thread Pool, Java sama stvara dretve i manipulira njima te izvodi kôd u Javi na više jezgara.

CPU: Procesor - Kôd se prevodi u OpenCL kôd i izvodi na procesoru na svim jezgrama. Slično ali drukčije od JTP moda.

GPU: Grafička - Kôd se prevodi u OpenCL kôd i izvodi na grafičkoj kartici.

Na kraju pozivamo *BasicNbodyMT*, i to ne jednom kao kod jednodretvenog primjera, nego s funkcijom *execute* koja kao argument prima *n* poziva funkcije *BasicNbodyMT*. *execute*(*n*) stvara *n* instanci te funkcije.

Važno je napomenuti da *execute* ne pokreće direktno *BasicNbodyMT* nego pokreće funkciju *run()* u višedretvenom objektu pa stvar izgleda ovako u našem objektu:

```
public void run(){
        BasicNbodyMT();
}
```
(6)

Sve drugo je, uz par iznimaka, isto kao i u Javi, konstruktori (graditelji), polja, konstante, davanje argumenata funkcijama i ostale funkcije rade na isti način. Uz učitavanje *com.amd.aparapi.Kernel* biblioteke ove 3 linije kôda su minimum minimuma koji je potreban da bi se program pokrenuo kao višedretvena aplikacija. Da bismo taj isti kôd pokrenuli na grafičkoj kartici jedino trebamo promijeniti način izvođenja u glavnom programu iz *JTP* u *GPU*. Iako možemo kontrolirati broj dretvi na kojima će se program

izvršavati, aparapi automatski sam dodjeljuje prikladan broj dretvi i kôd se prevede u OpenCL i izvršava na grafičkoj kartici.

Da bi program radio na grafičkoj kartici potrebno je instalirati nekoliko programa i ručno namještati njihov *path*[*] pa program neće automatski raditi na računalu na kojem sve to već nije napravljeno.

Za GPU programiranje ne treba poznavati OpenCL sintaksu niti hardver GPU-a, niti 4 vrste memorije, niti pojmove poput *konteksta* ili *kernela* koji su svojstveni OpenCL-u, a možete na GPU izvoditi relativno jednostavne ali računski intenzivne kalkulacije. Kôd koji se izvodi na CPU je isti kôd koji se izvodi na GPU i pisan je u Javi! Ovo nam je važno jer iz ovoga lako možemo uspoređivati kako jedan te isti kôd radi na GPU, na četiri ili na jednoj jezgri procesora.

Manjkavosti su to što Aparapi ne daje previše kontrole nad OpenCL jer je iznimno automatiziran i prilagođen za početnike. Tako kôd u objektu koji se izvodi na grafičkoj kartici ne može stvarati nove objekte. To znači da bilo što tipa *new* uključujući i *System.out.print();* neće raditi dok se izvodi na grafičkoj kartici. Isto tako nije moguće korištenje dvodimenzionalnih polja sa starijim generacijama grafičkih kartica. Ovo nije veliki problem ali kôd treba pisati od početka s ovim na umu ako ga se hoće izvršavati na GPU.

[*] Path je put do koji računalo slijedi kroz hijerarhiju datoteka da bi našlo potrebne biblioteke i programe

5 Optimizacija

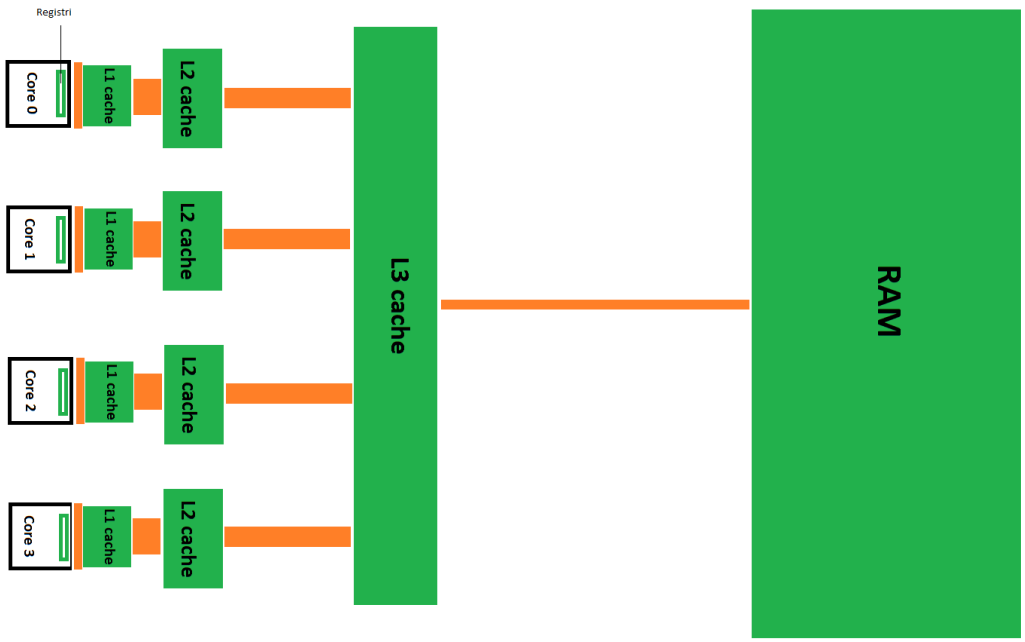
Sada želimo prilagoditi naš program da radi brže te istražiti u kojim uvjetima se to može postići na grafičkoj kartici a u kojim na procesoru. Želimo usporediti performanse i ocijeniti sličnosti i razlike GPU-a i CPU-a. Optimizacija programa može biti hardverska i softverska. S *hardverskom* optimizacijom pokušavamo što bolje iskoristiti postojeći hardver, a sa *softverskom* pokušavamo eliminirati one kalkulacije koje nam nisu potrebne a da je program i dalje ispravno radi. U praksi se programska optimizacija radi prva, a onda se postojeći kôd pokušava prilagoditi postojećem hardveru. Međutim ovdje će zbog procesa učenja taj proces biti obrnut. Da bismo mogli razumjeli hardversku optimizaciju prvo moramo razumjeti osnovne dijelove hardvera s kojim radimo.

5.1 Elementi CPU hardvera i osnovni pojmovi vezani uz njega

U analizu hardvera ćemo ići samo do te razine da možemo razumjeti osnovne hardverske optimizacije koje ćemo koristiti i pojmove vezane uz njih.

Na slici 5 vidimo shemu tradicionalnog procesora.

Slika 5



Shema četverojezgrenog tradicionalnog procesora

Sastoji se od 4 jezgre od kojih svaka ima svoj *cache* (hrv. priručna memorija ili predmemorija) nivoa 1 i nivoa 2 te zajedno dijele predmemoriju nivoa 3. Iznad toga se nalazi RAM. Između svakog nivoa predmemorije i jezgara teku podaci. To je prikazano narančastom linijom. Vrijeme pristupa podacima (latencija) je manje a podatkovna propusnost je veća što su oni u memoriji bližoj CPUu. Međutim, veličina memorije opada što je ona bliže jezgri. Ovo znači da je procesor napravljen tako da podaci najviše i najbrže teku između procesorske jezgre i nižih nivoa predmemorije. Da bi se omogućila veća brzina odziva i veća propusnost memorije ona se mora fizički smanjiti.

Svaka jezgra ima svoje registre koji čine najmanje jedinice memorije u kojima mogu stajati podaci. Vrijeme pristupa registrima je nula ciklusa. Svaka jezgra ima nekoliko logičko aritmetičkih jedinica koje računaju cjelobrojne operacije i rade operacije uspoređivanja te nekoliko jedinica koje računaju operacije s pomičnim zarezom (eng. *floating point unit FPU*). Kada nadalje budemo govorili ALU referirat ćemo se na jedno i drugo. Procesori novijih generacija imaju izmiješane ALU i FPU elemente ali krajnji rezultat jest da je jedna jezgra procesora u jednom ciklusu sposobna obraditi 3-4 cjelobrojne operacije ili operacije s pomičnim zarezom.

5.1.1 Cache /predmemorija

Predmemorija memorija ima 3 nivoa. U donjoj tablici su navedene okvirne performanse svakog nivoa predmemorije. Prvi nivo ima od 64 KB do 128 KB predmemorije od kojih je otprilike pola rezervirano za podatke a pola za instrukcije.

Tabela 2

CPU	Propusnost	Latencija (ciklusi)	Latencija (vrijeme)	Veličina
RAM	10-30 GB/s	70-250	23-83 ns	4-16 GB
Cache 3	50-250 GB/s	18-70	6-23 ns	6 - 8 MB
Cache 2	150-500 GB/s	12-15	4-5 ns	256-2048 KB
Cache 1	300-1000 GB/s	3-4	1 ns	64-128 KB
Cache 0/registri	-	0	0 ns	<1KB

Svi dijelovi predmemorije primaju kopije podataka koje se nalaze u RAM-u te dalje prosljeđuju te podatke nižim razinama memorije i samoj jezgri. Predmemorija je bitna jer ona sadrži podatke koji će biti potrebni za naredne kalkulacije. Podaci su u njoj spremljeni u

jednom nizu koji se zove *cache* linija. Ona iznosi 64 Bajta i u nju stane 16 podataka tipa float ili int ili 8 podataka tipa double. Podaci se u predmemoriju ali i iz nje dobavljaju i šalju u ovim cache linijama, minimalno u komadima od 64 Bajta. Ovo je najmanja zrnatnost predmemorije. Kada se briše podatak iz jedne cache linije briše se cijela cache linija. Cache linija se na kraju njezinog puta prebacuje u jezgru procesora u njegove registre kroz jedan ciklus ali ne češće od svaka 3-4 ciklusa. Ovo je značajno jer se ovim u jezgru procesora prebacuje 16 podataka iz iste cache linije. Događa se da je u izvođenju kôda potreban samo jedan podatak od tih 16 i ovih ostalih 15 propada. Ovo je često predmet optimizacije jer je pametnim programiranjem moguća bolja iskoristivost dobavljenih podataka.

5.1.2 Latencija

Latencija je vrijeme pristupa memoriji. To je vrijeme koje je potrebno da memorija postane sposobna za čitanje ili pisanje nakon proteklog pisanja ili čitanja. Jedina memorija koja je svaki ciklus procesora sposobna za čitanje i pisanje su registri. Sva ostala memorija treba nekoliko ciklusa što ovisi o njezinoj veličini i tehnologiji izrade.

Ukoliko podaci koji su potrebni za neku kalkulaciju nisu u predmemoriji nivoa 1 to se zove *cache miss* i ta jezgra mora čekati dok se ti podaci ne dohvate iz predmemorije nivoa 2. Ako su podaci prisutni tamo onda je potrebno vrijeme latencije da se ti podaci dostave u predmemoriju nivoa 1. Ukoliko nisu u predmemoriji nivoa 2 pretražuje se predmemorija nivoa 3, a ako podaci nisu niti tamo, onda se pretražuje RAM. Čekanje je dulje što je podatak dalje u memorijskoj hijerarhiji. Iz ovog razloga je bitno da su predmemorije što veće tako da se poveća vjerojatnost nalaska željenog podatka ili *cache hit*. Međutim, ukoliko su predmemorije fizički veće tada raste i njihova latencija, čime se poništava korist od velike predmemorije.

Ne samo da memorijska hijerarhija ima latenciju između pojedinih elemenata, već postoji latencija između svih elemenata računala. Za nas je posebno značajna latencija između grafičke kartice i procesora. Ona je reda veličine 10 mikrosekundi što je 100 puta sporije od latencije između RAM-a i predmemorije nivoa 3 i 10000 puta sporije od latencije predmemorije nivoa 1 i procesorske jezgre. Isto tako ukoliko koristimo višeprocesorski sustav za naše kalkulacije koji se sastoji od nekoliko povezanih računala postojat će mrežna latenciju između RAM-ova ovih računala. Koji put pod pojmom latencija ulazi i ukupno vrijeme koje je potrebno da se prenesu podaci.

$$T_{uk} = T_{lat} + T_B \tag{71}$$

Gdje je T_{lat} latencija tj. vrijeme odziva između dva memorijska elementa iz tablice (2). a T_B je vrijeme potrebno da se neka veća količina podataka prenese podatkovnom sabirnicom. Ono je jednako $T_B = \frac{D}{B}$, gdje je D veličina podataka a B je propusnost sabirnice. Ono je veće od nule ukoliko se podaci ne mogu prenijeti u jednom ciklusu sabirnice. Ako je podatkovni paket toliko mali da se prenese u jednom ciklusu podatkovne sabirnice, onda je ukupno vrijeme, T_{uk} , potrebno da ti podaci postanu dostupni jednako vremenu latencije T_{lat}. Kod transfera od čak nekoliko megabajta dominira drugi član i onda je vrijeme koje je potrebno da ti podaci postanu dostupni proporcionalno s T_B. Pod vrijeme latencije često se podrazumijeva vrijeme čekanja da podaci postanu operativno korisni pa se miješaju pojmovi T_{lat}, T_Bi T_{uk}. U daljnjem razmatranju pod latenciju ćemo uzimati vrijeme T_{lat} dok će T_B biti vrijeme prijenosa podataka.

5.1.3 Propusnost

Ako bi otvorili pipu kroz koju ide voda, vrijeme potrebno da ta voda počne teći bi bila latencija, a maksimalni tok vode u sekundi bi bila propusnost. Podatkovna propusnost je brzina kojom sabirnica prenosi podatke (eng. *bandwidth*). Brzine podatkovne propusnosti između raznih memorija su navedene u tablici 2. Mjere se u gigabajtima po sekundi. Propusnost isto tako može biti količina operacija koja se može napraviti na hardveru (eng. *throughput*). Ako ne navedemo drukčije, kada se budemo referirali na pojam propusnosti mislimo na podatkovnu propusnost.

5.1.4 Overhead

Overhead je naziv za ono vrijeme u kojem program ne vrši same kalkulacije nego se rade pripreme da bi se kalkulacije napravile. U to spadaju vremena prijenosa podataka ali i održavanja programskih struktura. Program u kojem imamo veliki broj memorijskih transfera s mnogo podataka na kojima napravimo malo kalkulacija ima veliki *overhead*.

5.1.5 Prefetch /pretpreuzimanje

Sam procesor ima pretpreuzimanje (eng. *prefetch*) jedinicu koja se brine da podaci koji će biti potrebni za kalkulaciju budu prisutni na vrijeme u predmemorijama. Ona funkcionira na složenim algoritmima koji, na osnovu prijašnjih kalkulacija, pokušavaju stvoriti obrazac kojim se dohvaćaju podaci u predmemoriju. Jezici poput C-a daju direktnu kontrolu nad

jedinicom za pretpreuzimanje, tako da programer po volji može pretpreuzeti željeni podatak. Bolja jedinica za pretpreuzimanje smanjuje efektivnu latenciju.

5.1.6 Lokalnost

Prostorna lokalnost je ostvarena kada su podaci koje je jezgra tražila blizu u vremenu isto tako blizu u prostoru u memoriji. Vremenska lokalnost je ostvarena kada obrađujemo jedan manji skup podataka učestalo u vremenu bez da ti podaci moraju biti jedan do drugog.

5.1.7 Memorijski pristup

Kada jezgra pristupa podacima u predmemoriji radi čitanja ili pisanja ona mora pretraživati tu memoriju da bi našla valjani podatak.
Postoje 3 glavna načina zapisivanja i čitanja podataka iz memorije, sekvencijalno gdje su podaci jedan iza drugog, sekvencijalno s nekakvim skokom između adresa podataka (eng. *stride*), te nasumično. Nasumično čitanje i zapisivanje je tip pristupa memoriji gdje su potrebni podaci kojima se pristupa raštrkani unutar memorije bez reda. Jezgra mora pretraživati cijelu predmemoriju svaki put da bi našla odgovarajući podatak. Sekvencijalni pristup memoriji se događa kada su podaci potrebni jezgri poslagani jedan iza drugog u memoriji. Sekvencijalni pristup je brži zato jer jezgra ne mora za svaki podatak ponovno pretraživati cijelu memoriju već zna točnu lokaciju svakog sljedećeg potrebnog podatka. Razlike u brzini između sekvencijalnog i i nasumičnog pristupa memoriji ne vrijede za predmemoriju prvog nivoa.

5.1.8 Polje

Sva polja bila ona jedno ili više dimenzionalna procesor vidi kao jednodimenzionalno polje. Dvodimenzionalno polje $a[n][k]$ računalo gleda kao jednodimenzionalno polje gdje prvih k elemenata označava nulti redak, a sljedećih k elemenata označava prvi redak i tako dalje. Takvo polje ima n redaka i k stupaca, računalo ih vidi kao jednodimenzionalno polje $a[n*k]$ veličine. Ovo se zove *row major* (redak glavni) podjela i vrijedi za Javu i C. Koje značenje ima ako je naš raspored *row major*? Ovo znači da ako čitamo dvodimenzionalno polje $a[i][j]$ po j varijabli mi zapravo čitamo podatke u jednodimenzionalnom polju jedan iza drugog jer su $a[i][j]$, $a[i][j+1]$ i $a[i][j+2]$ ... fizički jedan do drugog. Ako čitamo po i varijabli onda čitamo napreskokce jer je između $a[i][j]$, $a[i+1][j]$ i $a[i+2][j]$ k brojeva. Da bi se moglo operirati s tim podacima potrebno ih je dostaviti jezgri. Oni se jezgri dostavljaju u

segmentima od 64 bajta (cache linija) s određenim vremenom latencije. Kada su podaci dostavljeni najpovoljnije ih je sve iskoristiti. Iterirajući po *j* varijabli prolazi se kroz podatke unutar iste *cache* linije. S druge strane, iteracija varijabli *i* traži prijenos većeg broja *cache* linija jer se svaki podatak nalazi u drugom segmentu od 64 bajta. Stoga s brže čitaju i obrađuju podaci po *j* varijabli (indeks stupca u matrici) nego po *i* varijabli (indeks retka u matrici). . Efekt je naročito vidljiv kod računanja s matricama.Čitanjem i pisanjem po *j* se postiže veća učinkovitost cache linije.

Ovdje smo paušalno odredili da je *i*-ta varijabla ona koja određuje redak a *j*-ta koja određuje stupac jer nam je većina programa napravljena tako, a i intuitivno je da *i* stavljamo kao prvu, *j* kao drugu i *k* kao treću varijablu po kojoj iteriramo. U slučaju da imamo dvije ugniježđene petlje *i*-ta je ujedno i vanjska petlja dok je *j*-ta unutarnja petlja.

5.1.9 Koherencija i lažno dijeljenje

Podaci u predmemoriji moraju biti koherentni. To znači da ukoliko se podatak u predmemoriji nivoa 1 prve jezgre promjeni sve se ostale kopije tog podatka u ostalim predmemorijama moraju sinkronizirati s novonastalim podatkom. Brisanjem podatka iz predmemorije se briše cijela *cache* linija. Katkad je nekoj jezgri potreban podatak ali je zbog koherencije on obrisan iz predmemorije kada druga jezgra ažurirajući svoju predmemoriju mijenjala neki drugi podatak koji se nalazio u istoj ili ekvivalentnoj *cache* liniji. Naprimjer, brisala je podatak u svojoj predmemoriji na lokaciji 3, a drugoj jezgri je potreban podatak iz njezine predmemorije iz ekvivalentne cache linije s lokacije 14. Međutim cijela cache linija je obrisana iako nije bilo potrebno mijenjati i podatak na lokaciji 14. Kada se na ovaj način zbog koherencije brišu podaci koje nije potrebno brisati, onda se to naziva lažno dijeljenje (eng. *false sharing*). Učestalo sinkroniziranje podataka može uzrokovati zastoj i usko grlo u performansi izvođenja programa, jer da bi se podaci u svim predmemorijama nivoa 1 sinkronizirali, signal iz njih treba putovati od predmemorije nivoa 1 preko nivoa 2 i 3 pa ponovo kroz predmemoriju nivoa 2 od druge jezgre. Latencija ovog signala je vrlo velika. Intelovi procesori već nekoliko generacija imaju ugrađen *ring bus* (kružnu sabirnicu) koji međusobno povezuje i sinkronizira dijelove predmemorije nivoa 3 i ostale elemente procesora te ubrzava komunikaciju među jezgrama. Iznimno je važno da o ovome vodimo računa pogotovo kada koristimo više procesora u više različitih računala jer je latencija između njih još veća.

5.1.10 Bottleneck ili usko grlo

Ovo je poznat koncept koji kaže da ukoliko jedna komponenta procesora zaostaje značajno za ostalima, onda cijeli sustav radi slabije.

5.1.11 Skaliranje

Kako prelazimo na veći broj jezgara očekujemo da će performansa rasti proporcionalno broju jezgara. U tom slučaju bi imali linearno to jest idealno *skaliranje*. To ne mora biti slučaj. Ukoliko imamo usko grlo u memorijskoj latenciji, propusnosti ili količini memorije, povećanje broja jezgara nam neće dati veću performansu.

5.1.12 Branch predictor ili jedinica za predviđanje grananja

U svakom procesoru postoji jedinica za predviđanje grananja koja, na osnovu prijašnjeg uzorka dobivenih rezultata, predviđa koji će podaci biti potrebni za računanje te njima unaprijed puni kanal.

5.1.13 Pipeline ili kanal

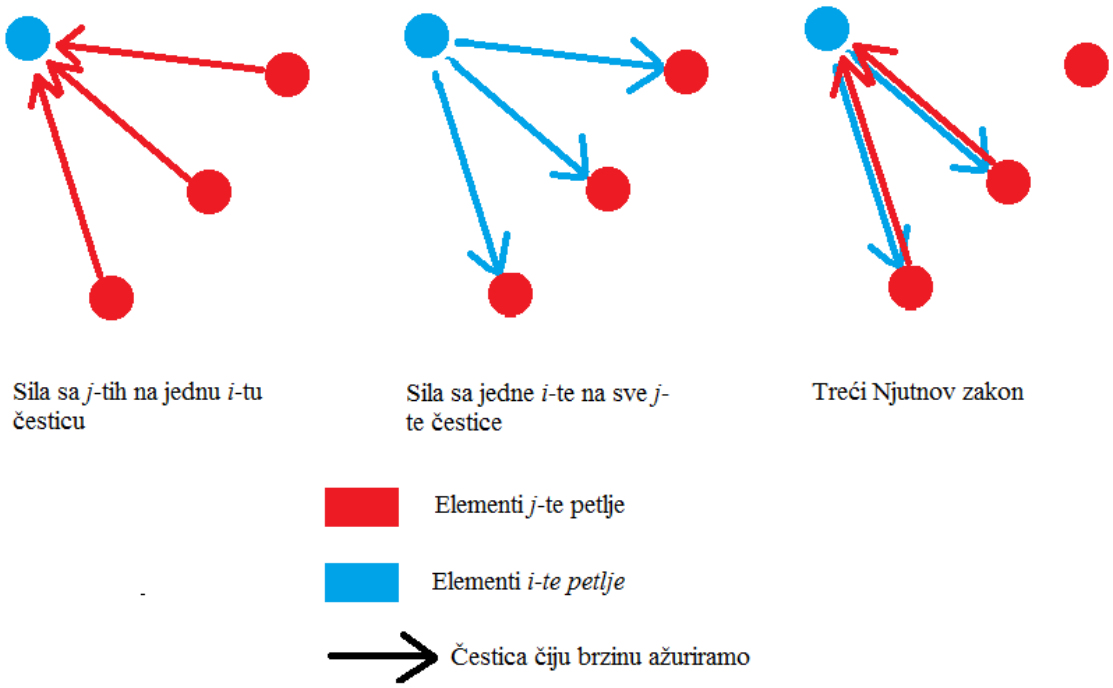
Svaki grafički i tradicionalni procesor ima kanal koji se puni elementima koji su potrebni za izvršenje instrukcije. Zamislimo to kao duljinu proizvodne trake u tvornici. Proizvodi se ne izrađuju na jednom mjestu nego svaki dio proizvodne trake izrađuje jedan dio proizvoda. U našem slučaju proizvodi su instrukcije/kalkulacije. Recimo da je naprimjer potrebno 20 ciklusa da se popuni cijeli kanal s podacima i elementima instrukcije, nakon što se to dogodi instrukcije se izvršavaju jednom svaki ciklus, međutim dok se kanal ne popuni i dok ne iziđe prva potrebno je 20 ciklusa. Kada dođemo do *if* funkcije koja određuje hoće li se izvoditi A ili B dio kôda, a za obadva su potrebni različiti podaci, jedinica za predviđanje grananja u procesoru pokušava predvidjeti na osnovu prijašnjih radnji, koja će od te dvije instrukcije biti izvršena te puni kanal podacima koji su joj potrebni. Recimo da puni podatke za A kôd. Ako rezultat grananja ide u smjeru B onda se odbacuju prije ubačeni podaci za A i dostavljaju za B, kanal se resetira na početak. Tako da je opet potrebno oko 20 ciklusa da se kanal napuni i da izvrši prva instrukcija. Tokom ovog ponovnog punjenja instrukcije se ne izvršavaju pa dolazi do gubitka performanse zbog grananja (eng. *branch penalty*). Iz ovog razloga je bitna lokalnost podataka tako da u slučaju grananja potrebni podaci nisu predaleko.

Mogli bi i dublje zaroniti u analizu hardvera, ali za optimizacije koje smo radili u programu ova razina poznavanja hardvera je sasvim dovoljna, a optimizacije koje iz toga proizlaze spadaju u bazične optimizacije koje bi svaki programer trebao znati.
Prije nego analiziramo GPU hardver pokušajmo razumjeti na tradicionalnom procesoru i na konkretnom primjeru kako funkcioniraju svi ovi elementi hardvera.

6 Hardverske optimizacije

Programski kôd već imamo, to je *BasicNbodyMT*. Ali to nije jedini način na koji možemo implementirati osnovni model n-tijela.

Slika 6



Imamo 3 sličice. Plavo su i-te čestice, crveno su j-te čestice, strelice pokazuju čija se brzina ažurira

Ako pogledamo *BasicNbodyMT* kôd vidimo da mi računamo utjecaje svih *j*-tih čestica na jednu *i*-tu česticu, zbrajamo ih u varijablu *acxtot* i *acytot* te onda tu varijablu pribrajamo *i*-toj brzini u x i y smjeru kada izađemo iz petlje. Ova je situacija prikazana na prvoj sličici na slici 6. No mi možemo i drukčije ažurirati naše brzine. Možemo umjesto da zbrajamo *j*-te utjecaje na *i*-tu česticu zbrajati utjecaj jedne *i*-te čestice direktno u brzine svih *j*-tih čestica. Koja je razlika? Pa nema razlike jer je broj kalkulacija isti (n^2) i iznos sile koju *i*-ta čestica projicira na *j*-tu je isti kao i iznos sile kojom ta ista *j*-ta čestica djeluje na *i*-tu česticu. Samo se ažuriranje vrši drukčijim redom. U jednoj instanci ne djeluju sve čestice na jednu nego jedna čestica djeluje na sve. Ipak kada napravimo kôd koji se ažurira po *j*-tim elementima imamo značajnu razliku u performansi. Nazovimo ovu novu funkciju *BasicNbodyMTJ* gdje "J" označava *j* tip ažuriranja. Radi boljeg razumijevanja preimenujmo prvu višedretvenu inačicu ove funkcije u I-tu verziju: *BasicNbodyMTI*. Promotrimo kôd:

```
public void BasicNbodyMTJ(){
        float recDistanceSQ,distanceSQ,deltx,delty,acxtot,acytot;
        int j;
        int i=getGlobalId();

                float t1x=xx[i];
                float t1y=yy[i];
                j=0;
                while(j<number){
        1//             deltx=t1x-xx[j];
        2//             delty=t1y-yy[j];
        3//             recDistanceSQ=-attCdeltat/(deltx*deltx+delty*delty+e2);
        4//             velxx[j]+=deltx*recDistanceSQ;
        5//             velyy[j]+=delty*recDistanceSQ;
                        j++;
                }
        xx[i]+=velxx[i]*deltat;
        yy[i]+=velyy[i]*deltat;
}
```

(7)

To što se događa izvan *while - j* petlje nije bitno jer je broj kalkulacija tamo zanemariv. Unutar *while j* petlje dvije su stvari drukčije. Umjesto da rezultat kalkuliranja akceleracije (četvrtu i petu liniju koda) dodajemo *n* puta u neku privremenu varijablu i s njome kasnije ažuriramo brzine jedne *i*-te čestice izvan *while j* petlje, mi jednom ažuriramo brzinu svake *j*-te čestice. Naravno, zbog toga što ažuriramo brzinu, a ne akceleraciju stvar moramo pomnožiti odmah s *deltat* i s atraktivnom konstantom. Da skratimo račun ovo smo već napravili i definirali konstantu:

*attCdeltat = attractiveConstant*deltat*

Ispred *attCdeltat* stavimo minus jer je djelovanje sile u suprotnom smjeru. Također smo *xx*[*i*] i *yy*[*i*] spremili u konstante *t1x* i *t1y*.

Performansa koju smo dobili na jednoj jezgri u J-toj verziji ove funkcije je porasla za 40%. Mjerenja radimo na AMD-ovom procesoru Phenom II x4 na 3.2 Ghz. Za jednu sličicu[*] gdje je n=16348 čestica s *BasicNbodyMTI* (kôd 4) funkcijom potrebno nam je 1188 ms na jednoj jezgri. S *BasicNbodyMTJ* (kôd 7) to vrijeme je smanjeno na 840 ms. Kako smo ovo dobili? Ako prvo u našoj *BasicNbodyMTI* funkciji promijenimo 4. i 5. liniju kôda unutar *j* petlje tako da se podaci više ne spremaju u konstante *acxtot* i *acytot* nego se direktno ažuriraju polja brzina *velxx* i *velyy* (znači više nije MTI nego MTJ), vrijeme izvođenja jedne sličice poraste na 1437 ms, međutim ako nakon toga u 1. i 2. liniji *while-j* petlje zamijenimo *xx*[*i*] i *yy*[*i*] s privremenim varijablama *t1x* i *t1y* vrijeme izvođenja jedne sličice nam pada na 840ms.

[*] Jedna sličica (eng. frame) je rezultat svih kalkulacija zajedno sa sudarima, pomacima i kalkulacijama akceleracije izvedenih u *deltat* vremenu.

Za ubrzanjeizvođenja programa su dakle zaslužna dva faktora:

1) Zamjena *xx*[*i*] s *t1x*.

2) Zamjena *acxtot+=deltx*recDistanceSQ ;* s *velxx*[*j*]*+=deltx*recDistanceSQ;*

Rezultat ovoga je *BasicNbodyMTJ* funkcija.

Obije promjene moraju istovremeno biti primijenjene da bismo imali efekt. Analizirajmo prvu.

1) Iz ovoga možemo zaključiti da iako se *xx*[*i*] element polja ne mijenja kroz izvođenje *while-j* petlje kao *xx*[*j*] varijanta koja je svaku *j*-tu iteraciju drugi broj, njegovo čitanje traje duže nego čitanje privremene varijable *t1x* u koju smo spremili taj podatak. Ovo bi impliciralo zaključak da se elementi polja poput *xx*[*i*] čitaju sporije od privremenih varijabli poput *t1x* iako su i jedan i drugi samo jedan te isti broj. Međutim kada izvodimo *I* varijantu *BasicNbody* funkcije i umjesto *xx*[*i*] elementa polja stavimo *t1x* ne opažamo nikakvo ubrzanje.

Zašto se ubrzanje opaža kada u *J* verziji zamijenimo pozivanje polja s privremenom varijablom. U *J* verziji ove funkcije imamo 6 poziva elemenata polja i zamjenom s privremenim varijablama funkciju svodimo na 4 poziva elementa polja dok u *I* verziji te funkcije zamjenom varijabli funkciju svodimo na 2 poziva elementa polja.

Moguće je da imamo ograničeni broj registara za elemente polja i ako ih pozovemo previše odjednom moramo obrisati neke podatke pa ih ponovo pozivati.

Mogli bismo nagađati o pravom mehanizmu ali možemo zaključiti da je uputno elemente polja koji se ne mijenjaju unutar petlje staviti u privremenu varijablu jer to može doprinijeti performansi.

2) Promotrimo kôd:

*acxtot+=deltx*recDistanceSQ ;* (8)

Prethodni kôd je zapravo malo brža verzija ovog:

velxx[*i*]= *velxx*[*i*]+*deltx*recDistanceSQ;* (9)

Taj kôd se nalazi u *I* verziji i sličan je ovom koji se nalazi u *J* verziji:

velxx[*j*]= *velxx*[*j*]+*deltx*recDistanceSQ;* (10)

U *I* verziji funkcije mi zbrajamo rezultat *j*-tog izračuna u *acxtot* ili *velxx*[*i*] i kao argument tog zbrajanja koristimo *acxtot* prethodnog izračuna. Kada zbrajamo *j*+10-ti utjecaj sile zbrajamo ga s *acxtot* elementom izračunatim u *j*+9-oj iteraciji. Dakle da bismo došli do *j*+10-og *acxtot*

elementa moramo izračunati sve prije toga. Primjećujmo da je ovaj mali dio kôda zavisan o prethodnim rezultatima i da se stoga mora izvršavati serijski.

U *J* verziji funkcije mi radimo to isto samo kao argument *j*+10-te iteracije uzimamo *j*+10-tu brzinu koja je izračunata u prethodnoj sličici (vremenskom koraku), pa nam ne treba *j*+9-ti izračun. Ovdje su *j-ti* rezultati računa nezavisni te se mogu paralelizirati ukoliko u procesoru postoje paralelne strukture. Svaki moderni procesor ima 3 ili 4 ALU jedinice koje mogu riješiti isto toliko instrukcija svaki ciklus. To znači da dok se u procesoru na jednom ALU izvršava *j*-ta iteracija, paralelno uz nju se izvršava *j*+1 na drugom ALU. U C-u je moguće koristiti vektorske SSE i AVX instrukcije koje primjenjuju istu instrukciju na više podataka odjednom i automatski iskorištavaju paralelizam unutar procesora.

Ažuriranje podataka po *i* je zapravo serijalizirano, dok ažuriranje po *j* ostavlja mogućnost paralelnog računanja i zapisivanja nekoliko podataka odjednom.

Podaci po j se zapisuju u istu cache liniju pa ažuriranje po j ima veću iskoristivost cache linije i bolje iskorištava paralelizam procesora te zbog toga ima veću performansu.

Zanimljivo je sljedeće opažanje. Ukoliko u *while j* petlju stavimo da iterator *j* raste za 2 (j=j+2) imati ćemo dvostruko manje prolazaka kroz *while j* petlju jer se svaka druga iteracija preskače. Razumljivo da zbog toga program više nije fizikalno ispravan. Međutim, to sada zanemarimo. Ako stavimo da iterator raste za 4 imati ćemo 4 puta manje prolazaka itd. Očekujemo da će se program izvršavati onoliko puta brže koliko preskočimo kalkulacija. Vrijeme izvođenja je prikazano u sljedećoj tablici.

Tabela 3

Vrijeme/ms	840 ms	702 ms	464 ms	349 ms	292 ms	245 ms	211 ms	182 ms
Broj kalkulacija	1	1/2	1/3	1/4	1/5	1/6	1/7	1/8
Puno vrijeme	840 ms	1404 ms	1392 ms	1396 ms	1460 ms	1470 ms	1477 ms	1456 ms

Prvi redak su vremena izvođenja na BasicNbodyMTJ za jednu jezgru i n=16384. Drugi redak je udio kalkulacija koji nije preskočen to jest koji se izvodi. Treći redak je vrijeme koje bi bilo potrebno za izvođenje cijelog računa ali da smo ga radili ovako napreskokce. Naprimjer j,j+2,j+4 u jednom komadu pa onda j+1,j+3,j+5 u drugom komadu. Zbroj obadva komada je puno vrijeme. Naprimjer 8 puta po 1/8 kalkulacija to jest 8 puta 182 ms je 1456 ms.

Vidimo da se ukupan očekivani broj kalkulacija za sva vremena nije mijenjao (oko 1450 ms) osim za prvo mjerenje gdje je to vrijeme 840 ms. Ispada da ukoliko je brojač namješten na 1 to jest j=j+1; imamo kraće vrijeme izvođenja nego što očekujemo. Jedno objašnjenje bi bilo da za svaku *j*-tu kalkulaciju zaista treba 1450 ms ali *j+1* kalkulacija počne prije nego se *j*-ta

završi. Ovo bi potvrđivalo gornju pretpostavku koja kaže da se u paralelnim strukturama procesora započne račun druge kalkulacije prije nego se dovrši prva, i onda rezultati obije ulaze u cache liniju jedan do drugog. Kod MTI verzije nema odstupanja od očekivanih vremena izvođenja kako preskačemo sve više kalkulacija. Zaista ispada da smo za MTJ verziju postigli paralelizaciju unutar same jezgre i zbog toga ide puno brže od MTI verzije.
Još jedno moguće objašnjenje jest da kada zapisujemo rezultate u polje po istoj varijabli po kojoj se iterira petlja, u ovom slučaju *j*, onda može doći do sekvencijalnog zapisivanja koje je brže. Međutim mjerenja nisu pokazala da je brzina sekvencijalnog zapisivanja brža od nasumičnog kada se zapisuje u predmemoriju nivoa 1.

Ova objašnjenja su mogući kandidati koji bi objasnili zašto je *BasicNbodyMTJ* brži od njegove I verzije. Ipak mnogo je nepoznanica i pretpostavki da bismo znali sigurno. Unatoč tome možemo izvući zaključke iz generalnog rada hardvera i opaženog.

Kada čitamo ili zapisujemo u polje iterativno unutar petlje pogodno je to činiti po varijabli po kojoj se iterira ta petlja. To je zbog toga što u računalu postoje cache linije s kojima se brže radi kada su podaci jedan do drugog. Ono je također je pogodno i zbog mogućnosti sekvencijalnog zapisivanja u memoriju, koje je brže od nasumičnog, jer se sljedeća lokacija zapisivanja ne mora pretraživati. A također je pogodno i zbog boljeg iskorištavanja paralelizma unutar jezgre procesora.

Znači generalno:

```
while(j<number){
        rez=b+c[j];
        a[j]=rez;                                                    (11)
        j++;
}
```

je bolje od:

```
while(j<number){
        rez=b+c[i];
        a[i]=rez;                                                    (12)
        j++;
}
```

Čak i za jednodimenzionalno polje.

Na slici 6 imamo i treću sličicu koja pokazuje da je sila *i*-te čestice na *j*-tu ista kao i sila *j*-te na *i*-tu. Sve *j*-te čestice će biti *i*-te kada na njih dođe red. Čestica s indeksom 586 je *i*-ta kada je

i=586 i *j*-ta kada je j=586. Ako su dvije sile jednog para čestica iste onda je dovoljno da ih samo jednom izračunamo. Težina ovog algoritma bi bila skoro dva puta manja, $O(\frac{13}{11}\frac{n^2}{2})$ tako da bi ova optimizacija predstavljala i programsku optimizaciju jer bi smanjivala broj potrebnih kalkulacija. Faktor 13/11 je dodan zato jer imamo 13 operacija s pomičnim zarezom po jednoj iteraciji *while* petlje zbog ažuriranja u dva smjera. Ovu implementaciju nazivamo *BasicNbodyMTIJ.*

Pretpostavimo da imamo 5 čestica s indeksima 0,1,2,3,4. Da bi izračunali međudjelovanje na nultu česticu potrebno je izračunati 4 interakcije. Dobivamo 8 sila. 4 sile djeluju na nultu česticu koja je *i*-ta čestica a 4 djeluju na ostale čestice koje su *j*-te čestice. Ovo implicira da koristimo dijelove I-te i J-te verzije *BasicNbody* funkcije. Kada prijeđemo na česticu s indeksom 1 onda na nju djeluju 4 sile i ona djeluje s 4 sile na ostale čestice ali djelovanje te čestice na nultu smo već izračunali u prethodnoj iteraciji po *i*. Tako da onda računamo djelovanje samo na 3 čestice. Sljedeća na dvije i predzadnja na jednu. Zadnja djeluje sama na sebe to jest na nikog jer ne treba računati djelovanje *i*-te čestice na samu sebe. Iz ovog razloga *j*-ti brojač prije ulaska u *while j* petlju za *i*-tu česticu indeksa 0 ne kreće od 0 nego od 1, za *i*-tu česticu indeksa 3 od 4 i generalno za *i*-tu česticu od indeksa $i+1$ pa je j=i+1. Dolje niže je izvedba ovog algoritma kojeg zovemo *BasicNbodyMTIJ* jer sadržava ažuriranje akceleracija na *i*-te i na *j*-te čestice.

```
public void BasicNbodyMTIJ(){
        float recDistanceSQ,deltx,delty,acxtot,acytot,acx,acy;
        int j;
        int i=getGlobalId();
                float t1x=xx[i];
                float t1y=yy[i];
                acxtot=0;
                acytot=0;
                j=i+1;
                while(j<number){
                1//     deltx=t1x-xx[j];
                2//     delty=t1y-yy[j];                                    (13)
                3//     recDistanceSQ=attCdeltat/(deltx*deltx+delty*delty+e2);
                4//     acx=deltx*recDistanceSQ;
                5//     acy=delty*recDistanceSQ;
                6//     acxtot+=acx;
                7//     acytot+=acy;
                8//     velxx[j]-=acx;
                9//     velyy[j]-=acy;
                        j=j+1;
                }
                velxx[i]+=acxtot;
                velyy[i]+=acytot;
```

```
                xx[i]+=velxx[i]*deltat;
                yy[i]+=velyy[i]*deltat;
    }
```

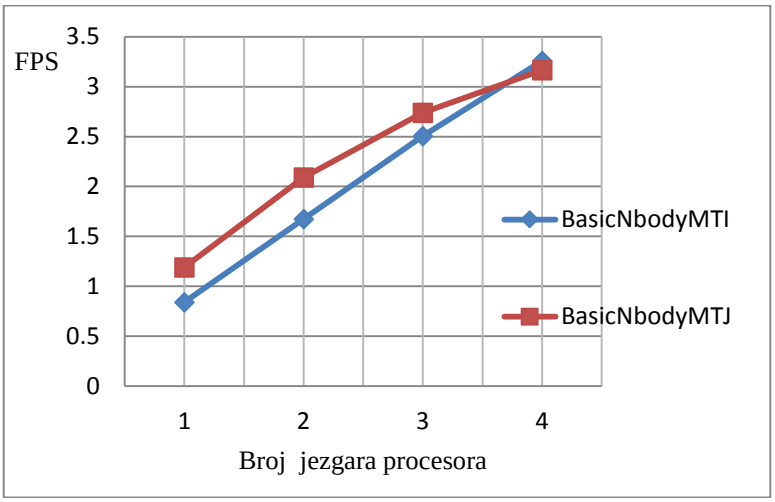
Dakle brojač za *i*-te čestice ne kreće od 0 nego od *i*+1 i to je faktor zbog kojeg imamo dvostruko manje kalkulacija. Vidimo da su 6,7,8, i 9. linija koda ažuriranje brzina po *i* i po *j*. *acxtot-=acx;* je samo malo brža verzija *velxx*[*i*]+=*acx;* Primijetimo također da naša *acx* i *acxtot* varijabla zapravo nisu akceleracije već imaju dimenziju brzine jer smo ih morali pomnožiti s vremenom *deltat*. Isto kao i s atraktivnom konstantnom ostavljamo ove nazive da dalje ne zbunjujemo s izmišljanjem novih naziva varijabli za jednu te istu stvar.

Na jednoj jezgri procesora brzina izvođenja ovog kôda je 789 ms. Ovo je dosada najbrže izvođenje. Zbog skoro dvostrukog smanjenja broja kalkulacija očekivali bismo i bolji rezultat. Analizirajući MTI i MTJ dobiti ćemo potpuniju sliku problema. Na MTIJ model ćemo se još kasnije vratiti. Pogledajmo prvo kako se naše funkcije izvode na više jezgara procesora.

6.1 Primjer lošeg skaliranja i lažnog dijeljenja

Testirajmo brzinu izvođenja *BasicNbodyMTI* i *BasicNbodyMTJ* funkcije na jednoj, dvije, tri i četiri jezgre.

Slika 7



Vidimo kako raste performansa u ovisnosti o porastu broja jezgara, mjera performanse su sličice po sekundi (eng. frames per second - FPS)

Primijetimo da unatoč tome što je *J* verzija inicijalno brža na jednoj jezgri da se ona ne skalira dobro s povećanjem broja jezgara kao *I* verzija.

Zbog čega je ovo? U našem slučaju predmemorija svake jezgre sadrži istu kopiju podataka iz RAM-a. Naprimjer *velxx* ili *velyy* polje. Ovi podaci moraju biti isti u svakoj od predmemorija to jest moraju biti koherentni. Kada jedna jezgra zapisuje u svoju predmemoriju ona mijenja taj podatak i zbog koherencije u ostalim predmemorijama se briše kopija tog podatka i ažurira se s novom (odjeljak 5.1.9 stranica 18). Ovo se zove *sinkronizacija.* Ako promotrimo 4 i 5. liniju kôda iz funkcije *BasicNbodyMTJ* (kôd 7):

```
while(j<number){
                    // računanje akceleracija
4//                 velxx[j]+=deltx*recDistanceSQ;
5//                 velyy[j]+=delty*recDistanceSQ;
                    j++;
          }
```

vidimo da ta funkcija postepeno mijenja sve podatke iz *velxx* i *velyy* polja jer prođe kroz sve *j*-te elemente. Time jedna jezgra briše i ažurira kopije tih podataka u ostalim predmemorijama. Ostale jezgre rade to isto. Kada se ažurira podatak ne samo da se briše jedan podatak, nego se briše i ažurira cijela cache linija. Tako se javlja efekt lažnog dijeljenja[*]. Što više jezgara izvodi ovaj račun to će više jezgara međusobno brisati podatke jedna drugoj. Iz ovog razloga se *J* verzija naše funkcije ne skalira dobro s povećanjem broja jezgara. *BasicNbodyMTI* (kôd (4)) ima puno bolje skaliranje.

```
while(j<number){
                    //računanje akceleracija
4//                 velxx[i]+=deltx*recDistanceSQ;
5//                 velyy[i]+=delty*recDistanceSQ;
                    j++;
          }
```

Ovo je *I* verzija naše funkcije u kojoj smo zamijenili privremenu varijablu *acxtot* s *velxx*[*i*]. Kôd je skoro isti (malo se sporije izvodi ali neznatno) samo što umjesto da zbrajamo kalkulacije u privremenu konstantu *acxtot* i na kraju *while* petlje jednom ažuriramo brzine, ovdje u svakoj iteraciji *while* petlje direktno ažuriramo brzine. Važno je za primijetiti da ovdje u oba dva slučaja mi ažuriramo samo dva podatka *velxx*[*i*] i *velyy*[*i*], bilo direktno bilo preko privremene varijable *acxtot* i *acytot* jer *i* ostaje konstantan tijekom iteriranja while petlje. *i*-ta instanca *BasicNbodyMTI* funkcije zapisuje samo u *velxx*[*i*] i *velyy*[*i*] i briše/ažurira samo te podatke u predmemorijama ostalih jezgara. i+20 instanca zapisuje samo u *velxx*[*i*+20] i *velyy*[*i*+20] i briše/ažurira samo te podatke u ostalim predmemorijama. Svaka *i*-ta instanca *BasicNbodyMTI* funkcije zapisuje/briše/ažurira samo u jednu cache liniju i zbog toga među jezgrama nema interferencije. Nije bitno je li se podatak ažurira učestalo kroz svaku iteraciju *j*

[*] Opisano u odjeljku 5.1.9 Koherencija i lažno dijeljenje

petlje kao kod velxx[i]+= ili samo jednom na kraju *j* petlje kao kod *acxtot* varijante! Bitan je broj podataka/cache linija koji se ažurira kroz jednu iteraciju. Linearno skaliranje nam limitira mala propusnost podataka među jezgrama i visoka latencija signala među jezgrama. Čak i kod problema koji se za manji broj jezgara mogu relativno dobro skalirati ne možemo očekivati da linearno skaliranje postoji i kod većeg broj jezgara ili velikog broja procesora.

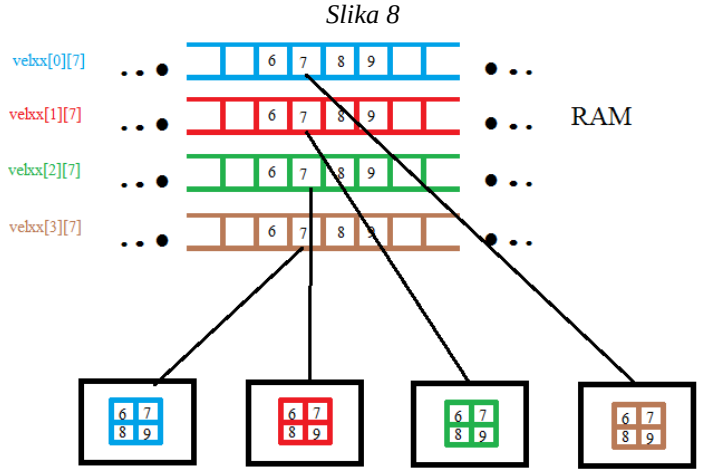
Da bismo izbjegli probleme s lažnim dijeljenjem i općenito probleme s koherencijom podataka, poželjno je da kada pojedina instanca dretve vraća rezultat koji se bilježi u neko globalno polje, da ta instanca ažurira samo svoje elemente tog polja a ne da se skup podataka koje ažurira presijeca sa skupom podataka neke druge dretve. Za naš konkretan primjer poželjno je ažuriranja rezultata računa u svakoj instanci raditi s velxx[i] umjesto velxx[j] ako je velxx neko globalno polje dostupno svim jezgrama.

Mogli bismo isto tako ažurirati po *j* lokalna polja koja vide samo njihove instance. Iako ovaj pristup ima smisla on nije dao dobre rezultate.

6.2 Kopiranje podataka za svaku dretvu/jezgru

Ukoliko nekoliko jezgara čita iz jednog te istog globalnog polja neće dolaziti do zastoja, međutim ukoliko sve jezgre zapisuju u to isto polje dolazi do problema s koherencijom kao što smo maloprije pokazali. Jedan od načina da ovo riješimo je da radimo s I varijantom funkcije koja ažurira samo jedan podatak tako da ne dolazi do brisanja cijele predmemorije ostalih jezgara. Međutim ako želimo zadržati J varijantu programa koja se pokazala bržom od I varijante na jednoj jezgri, možemo napraviti njenu optimizaciju za više jezgara.

Slika 8



Svaka jezgra ima globalno polje i iako sve jezgre pristupaju istom elementu nema lažnog dijeljenja jer se svaki element nalazi u svojem polju.

Svakoj jezgri dodijelimo neko njezino globalno polje u koji samo ona zapisuje. Znači umjesto da imamo jedno polje naprimjer *velxx* koji se pozivanjem kopira u svaku od jezgara, napravimo kopije *velxx1*,*velxx2*,*velxx3*, *velxx4* od kojih svako polje poziva jedna jezgra. Svaka od ovih kopija ima sve podatke iz *velxx* polja i jezgre efektivno računaju s *velxx* poljem ali kada *BasicNbodyMTJ* zapisuje podatke u predmemoriju nije potrebno sinkronizirati predmemorije različitih jezgara jer tehnički gledano, svaka jezgra zapisuje u svoje polje. Nakon što smo izračunali sve instance imamo 4 parcijalna rezultata koji su spremljeni u 4 polja i potrebno ih je zbrojiti da bi dobili *velxx*. Problem s ovim načinom rješavanja je što povećavamo korištenje RAM memorije onoliko puta koliko imamo dretvi. Napravili smo novu funkciju *BasicNbodyMTJwide* gdje "wide" označava da su polja koja koriste ovu funkciju šira za broj jezgara ili dretvi koje koriste funkciju.

Umjesto da odredimo 4 fiksna polja definirali smo jedno 2D polje *velxx*[*id*][*j*] gdje je *id* oznaka jezgre/dretve koja poziva to polje. *id* identificiramo posebnom naredbom *getLocalId*(); koja odredi na kojoj se jezgri/dretvi izvodi ova iteracija. Bitne promjene su samo u 4. i 5. liniji kôda unutar *while j* petlje.

```
public void BasicNbodyMTJwide(){
        float recDistanceSQ,distanceSQ,deltx,delty,acxtot,acytot;
        int j;
        int id=getLocalId();
        int i=getGlobalId();

                float t1x=xx[i];
                float t1y=yy[i];
                j=0;
                while(j<number){
        1//              deltx=t1x-xx[j];                                          (14)
        2//              delty=t1y-yy[j];
        3//              recDistanceSQ=-attCdeltat/(deltx*deltx+delty*delty+e2);
        4//              velxxwide[id][j]+=deltx*recDistanceSQ;
        5//              velyywide[id][j]+=delty*recDistanceSQ;
                        j++;
                }
        xx[i]+=velxx[i]*deltat;
        yy[i]+=velyy[i]*deltat;
}
```

Zbog toga što se pomoćna polja moraju zbrojiti tek nakon što smo ih izračunali, funkciju sumiranja parcijalnih podataka radimo tek nakon što završimo *BasicNbodyMTJwide* račun.

```
public void ArrayAdder(){
        for(int k=0;k<TotThNum;k++){          //
            velxx[i]+=velxxwide[k][i];        //sumiranje
            velyy[i]+=velyywide[k][i];        //parcijalnih
            velxxwide[k][i]=0;                //brzina
            velyywide[k][i]=0;                //
        }
}
```
(15)

Također se pomaci položaja rade tek nakon što ažuriramo brzine to jest nakon što zbrojimo parcijalne rezultate pa bi i funkciju pomaka mogli staviti u zasebnu funkciju. Primijetimo da smo mnogo prije definirali Integrator, tj. dio koda koji je odgovoran za integriranje to jest pomicanje čestica ažuriranjem brzina i pomaka. Ovdje se stvorila potreba da ga spremimo u zasebnu funkciju pa ćemo to sada i napraviti.

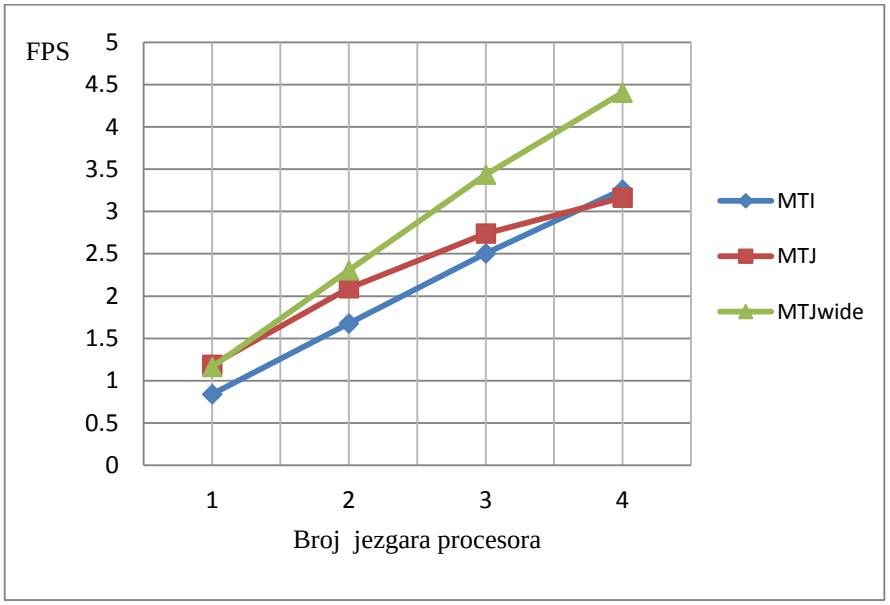
```
public void SemiIntegrator(){
        int i=getGlobalId();
        xx[i]=xx[i]+velxx[i]*deltat;
        yy[i]=yy[i]+velyy[i]*deltat;
}
```
(16)

Ovo nije pravi integrator jer mu nedostaje ažuriranje brzina koje smo napravili odmah pri izračunu akceleracije.

Da bismo izveli novi kôd *execute*(*n*) funkciju pozivamo 3 puta. U prvom pozivu se računa akceleracija preko *BasicNbodyMTJwide*(), u drugom se sumiraju parcijalna polja preko *ArrayAdder*() a u trećem se vrši integracija to jest pomak preko *SemiIntegrator*(); funkcije. Posljednja dva poziva traju iznimno kratko zbog svoje $O(n)$ težine.

Ovo je zapravo *J*-ta verzija ažuriranja akceleracija kod koje svaka jezgra koristi svoj polje. Brzina jedne jezgre je malo sporija : 861 ms za *Jwide* naspram 841 za *J* verziju. Međutim na 4 jezgre *Jwide* ide mnogo brže (227 ms *Jwide* naspram 316 ms *J*). Osim toga rezultat i puno se bolje skalira s brojem jezgara:

Slika 9

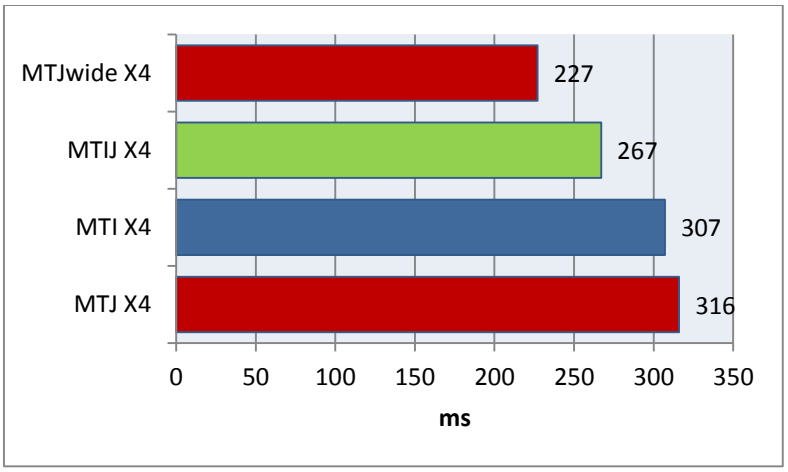


Zeleno je implementacija BasicNbodyMTJwide funkcije s pomoćnim poljima za n=16384

BasicNbodyMTJwide ima prednost ažuriranja po *j* bez problema s koherencijom.

Ovo je dosadašnja performansa svih funkcija izražena u milisekundama.

slika 10



Performansa dosadašnjih izvedbi funkcije BasicNbody na 4 jezgre. Manje je bolje.

6.3 Cijepanje petlji

Kada se pokrene program jedinica za pretpreuzimanje dohvaća potrebne podatke u predmemorije procesora. Ukoliko je podataka previše za predmemorije nižeg nivoa oni se prelijevaju u više nivoe ili čak RAM. Ukoliko su nam potrebni podaci koji nisu blizu jezgre (blizu što se tiče vremena dohvata to jest latencije) jezgra mora čekati na njih. jedinica za pretpreuzimanje pokušava na osnovu prijašnjeg uzorka dohvaćenih podataka prepoznati

obrazac po kojem se podaci dobavljaju u memoriju. Bilo kakvo čekanje na podatke je nešto nepoželjno. Manja je vjerojatnost da ćemo čekati podatke ako je jedinica za pretpreuzimanje pogodila koji podaci su nam potrebni i na vrijeme ih dostavila u predmemoriju nivoa 1. Jedan jednostavan način da pomognemo jedinici za pretpreuzimanje jest da napravimo velika polja podataka i pristupamo im element po element. Jedinica za pretpreuzimanje neće imati previše problema da prepozna da je sljedeći potrebni podatak onaj koji se nalazi iza prethodnog. Ovakvim poljima smo ostvarili prostornu lokalnost. Iako nismo testirali, tip podatka *polje* je načelno brži od *ArrayList* pa smo koristili polja.

Manja je vjerojatnost da ćemo čekati podatke što je predmemorija veća (veća vjerojatnost nalaska željenih podataka) i što je latencija s ostalim elementima memorije manja (kraće čekamo nakon *cache miss*[*] događaja). Ne možemo napraviti ništa po pitanju veličine memorije ili njezine latencije ali možemo se pobrinuti da se ona bolje iskorištava. Nepoželjno je da u memoriju dobavljamo podatke koji neće biti korišteni. Neke od realizacija problema n-tijela su uključivale stvaranje novog tipa podatka koji u sebi ima informaciju o brzinama, položajima i masi. Ukoliko nama u jednom trenutku treba samo informacija o položaju ovi drugi podaci nam samo zauzimaju predmemoriju. Tako se smanjuje efektivna predmemorija i povećava vjerojatnost za *cache miss*. Naši podaci su stavljeni u velika jednodimenzionalna polja od kojih svako predstavlja ili brzinu ili položaj. Ukoliko trebamo obrađivati položaj u *x* smjeru nije vam potrebno dostavljati druge podatke. No ovo je prilično intuitivan način kako realizirati podatkovnu strukturu.

Razmotrimo na trenutak našu *while* petlju.

```
j=0;
while(j<number){
                deltx=t1x-xx[j];
                delty=t1y-yy[j];
                recDistanceSQ=-attCdeltat/(deltx*deltx+delty*delty+e2);
                velxxwide[id][j]+=deltx*recDistanceSQ;
                velyywide[id][j]+=delty*recDistanceSQ;
                j++;
}
```
(17)

Vidimo da ona tokom svog trajanja cijelo vrijeme zahtjeva podatke od 4 polja. Ukoliko bi razdvojili neke elemente ove petlje i stavili ih u drugu *while j* petlju imali bi više mjesta u predmemoriji ili registrima za ona polja koji se trenutno izvršavaju. *xx*[] i *yy*[] ne možemo delegirati u drugu petlju jer su nam uvijek nužni za račun radijusa. Ali možemo naprimjer

[*] Objašnjeno u odjeljku 5.1.2 Latencija

delegirati *velyywide* s tim da moramo ponoviti kalkulacije koje smo već napravili za *velxxwide*! Primjer takvog kôda je sljedeći, funkcija se zove *MTJwideFission*, ovdje smo izostavili *BasicNbody* naziv jer se podrazumijeva da su sve funkcije nadalje tog tipa.

```
public void MTJwideFission (){
        //konstante i privremene varijable t1x i t1y
        j=0;
        while(j<number){
                deltx=t1x-xx[j];
                delty=t1y-yy[j];
                velxxwide[id][j]-=deltx*attCdeltat/(deltx*deltx+delty*delty+e2);
                j++;
        }                                                                    (18)

        j=0;
        while(j<number){
                deltx=t1x-xx[j];
                delty=t1y-yy[j];
                velyywide[id][j]-=delty*attCdeltat/(deltx*deltx+delty*delty+e2);
                j++;
        }
```

Ovo se zove *loop fission* tj. cijepanje petlji. Suprotno je *loop fusion* ili spajanje petlji. Ovisno o situaciji i jedno i drugo nam može biti od koristi. Ovdje smo postigli da se u jednom pozivanju *while j* petlje u predmemoriju dostavljaju samo 3 a ne 4 polja tako da ona polja na kojima se radi imaju veći efektivni cache i veću vjerojatnost za *cache hit*. Drugo objašnjenje je da smanjenjem broja podataka koji se odjednom (unutar jedne while *j* iteracije) pozivaju imamo bolju iskoristivost podataka koji su dostavljeni u registre. Imali smo takav primjer kada smo prelazili s MTI na MTJ varijantu gdje smo u jednom trenutku imali pozive iz 6 polja.

Bilo zbog boljeg iskorištavanja predmemorije ili registara bolje su iskorišteni dostavljeni podaci. Razumno je pretpostaviti da smo ostvarili bolju iskoristivost cache linije.

Ipak, zbog poduplavanja petlji imamo više kalkulacija koje se moraju ponoviti! Imamo 2×9 operacija s pomičnim zarezom. Što znači 18 naprama dosadašnjih 11. Unatoč poduplavanju kalkulacija performansa *MTJwideFission* funkcije je 202 ms, a što je dosada najbrže izvođenje!

Ukoliko imamo previše memorijskih zahtjeva unutar jedne iteracije petlje bilo zbog ograničene predmemorije ili broja registara pogodno nam je napraviti cijepanje petlji i rastaviti petlju na dvije i neke kalkulacije ponoviti.

6.4 Spremanje podataka za ponovno korištenje

Što ako bismo kalkulacije jednom izračunate zapisali u memoriju i pozivali iz memorije umjesto da ih ponovo računamo. Već smo prije imali primjer gdje umjesto da stalno računamo razliku između *xx*[*i*] i *xx*[*j*] položaja taj rezultat spremimo u varijablu *deltx*. Ovdje ćemo spremati cijeli niz podataka u polje. Nova funkcija se zove *MTJwideFissionMem* gdje *Mem* označava da pamtimo jedan dio računa. Primjer takvog koda je sljedeći.

```
public void MTJwideFissionMem(){
        //konstante i privremene varijable t1x i t1y
                j=0;
                while(j<number){
                        deltx=t1x-xx[j];
                        delty=t1y-yy[j];
                        RecDistanceSQ[id][j]=-attCdeltat/(deltx*deltx+delty*delty+e2);
                        j++;
                }

                j=0;
                while(j<number){
                        deltx=t1x-xx[j];
                        velxxwide[id][j]+=RecDistanceSQ[id][j]*deltx;
                        j++;
                }

                j=0;
                while(j<number){
                        delty=t1x-xx[j];
                        velyywide[id][j]+=RecDistanceSQ[id][j]*delty;
                        j++;
                }
}
```

(19)

Dok u *MTJwideFission* funkciji imamo dvije petlje u kojima se u obije računa *recDistanceSQ* ovdje imamo 3 petlje gdje se u prvoj računa *recDistanceSQ* i pamti u *RecDistanceSQ*[] polje iz kojeg se poslije ažuriraju pomoćna polja brzine.

Vrijeme potrebno za jednu sličicu (vremenski korak) je 172 ms. Broj operacija s pomičnim zarezom po jednoj iteraciji prve petlje je 7 a za druge dvije je 3, što ukupno iznosi 13 kalkulacija za jedan *j*-ti element.

Ukoliko se kalkulacije ponavljaju korisno ih je jednom napraviti i spremiti ih u memoriju te ponovno pozvati iz memorije ukoliko je komunikacija s memorijom kraća od samih kalkulacija, u suprotnom je uputno kalkulacije ponoviti.

Sljedeći primjer objedinjuje ove optimizacije u varijanti koju smo već spomenuli: MTIJ varijanta koja ima dvostruko manje prolaza kroj *while j* petlju jer ažurira dvije akceleracije odjednom, te ima 13 operacija s pomičnim zarezom za jednu *j*-tu iteraciju.

Zovemo je *MTIJwideFissionMem*: Ona poput prethodne varijante ima jednu petlju koja računa doprinose novih brzina i sprema ih u pomoćno polje. I poput prethodne verzije ažurira *velxxwide*[*id*][*j*] i *velyywide*[*id*][*j*]. MTIJ trebamo dodati još dvije petlje koje ažuriraju *velxxwide*[*id*][*i*] i *velyywide*[*id*][*i*]. Brojači j ne idu od 0 nego od i+1. Ovo je objašnjeno u *BasicNbodyMTIJ* (13) verziji. U ovoj smo verziji funkcije jednu while petlju rastavili na 5 petlji. Kôd je u svemu isti kao i prethodna verzija samo što ima još dvije petlje koje ažuriraju velxxwide i velyywide po *i*. Ovo je ujedno i programska optimizacija jer smanjuje broj kalkulacija $O(0.6 * n^2)$. Međutim brzina njezinog izvođenja je tek oko 176 ms, dakle približno jednaka prethodnoj verziji koja ima skoro dvostruko više kalkulacija $O(n^2)$. Ona je također oko 15% brža od *MTJwideFission* verzije koja ima bar oko 3 puta više kalkulacija $O(\frac{18}{11} * n^2)$ jer se tamo neke kalkulacije ponavljaju. Zaključak nas navodi na to da nam je komunikacija s memorijom ograničenje. Provjeriti ćemo ovaj zaključak u sljedećem primjeru.

6.5 Spajanje/fuzija petlji

Kao što nije dobro nagurati mnogo kalkulacija u jednu petlju jer se time opterećuju registri, isto tako nije dobro razdvojiti kalkulacije u previše petlji jer se gubi prilika za iskorištavanjem paralelizma unutar jedne jezgre procesora. Metodom pokušaja i pogreške smo napravili sljedeću funkciju *MTIJwideFissionFusionMem* u kojoj smo 5 petlji sveli na dvije.

```
public void MTIJwideFissionFusionMem(){
        //konstante i privremene varijable t1x i t1y
        j=i;
        while(j<number){
                deltx=t1x-xx[j];
                delty=t1y-yy[j];
                recDistanceSQ=attCdeltat/(deltx*deltx+delty*delty+e2);
                RecDistanceSQx[id][j]=recDistanceSQ*deltx;//dva pomoćna polja
                RecDistanceSQy[id][j]=recDistanceSQ*delty; //koja pamte rezultate
                j++;
        }                                                                    (20)

        j=i;
        while(j<number){ //jedna petlja zamijenila 4
                velxxwide[id][j]-=RecDistanceSQx[id][j];
                velyywide[id][j]-=RecDistanceSQy[id][j];
                acxtot+=RecDistanceSQx[id][j]; //velxxwide[i] mijenjamo s acxtot
                acytot+=RecDistanceSQy[id][j]; //
                j++;
                }
```

```
            velxx[i]+=acxtot;
            velyy[i]+=acytot;
    }
```

Ažuriranje *velxxwide*[*id*][*i*]+= smo zamijenili s ažuriranjem konstante *acxtot* i ekvivalentno za y koordinatu. Da smo u jednu petlju stavili da se ažuriraju 4 polja program bi radio sporije, ali kako smo prije zaključili jezgra drukčije radi s poljima i privremenim varijablama tako da smo njihovom kombinacijom vjerojatno oslobodili resurse unutar jezgre procesora.

Brzina izvođenja jednog ciklusa je 108 ms za ovu inačicuprograma! Ovo ubrzanje u izvođenju dolazi od smanjenja broja kalkulacija ali i boljeg korištenja paralelizma jezgre procesora u drugoj petlji te optimizacije memorijske komunikacije. Ova varijanta je 2.6 puta brža od inicijalne MTIJ varijante koja je imala 9 linija koda u while petlji i koja je imala probleme sa skaliranjem zbog ažuriranja po *j*.

Vidimo da procesoru odgovara da u jednoj iteraciji while j petlje nije više niti manje od 4-5 linija kôda i da se za ažuriranje rezultata ne koriste samo polja nego i privremene konstante.

6.6 Odmotavanje petlje

Uzevši u obzir sve ove optimizacije vratimo se još jednom na našu MTI varijantu te ih primijenimo na nju. Pođimo od *MTJwideFissionMem* (kôd (19)) koja ima 1 petlju koja računa i pamti rezultate i 2 petlje koje ažuriraju te rezultate preko *j* iteratora kroz polja. Pretvorimo ove dvije petlje u ažuriranje po *i* preko *acxtot* i *acytot* varijabli. S ovom promjenom dobivamo varijantu *MTIFissionMem* s 13 kalkulacija po *j* iteraciji. Performansa ove varijante je 260 ms. Spojivši drugu i treću petlju koje ažuriraju podatke u jednu dobivamo *MTIFissionFusionMem* varijantu kojoj je vrijeme izvođenja oko 200 ms a broj kalkulacija 13. Sljedeća varijanta koja je bazirana na ovima je *MTIFissionFusionMemUnroll* i isto nasljeđuje 13 operacija s pomičnim zarezom po jednoj iteraciji *j* petlje.

```
public void MTIFissionFusionMemUnroll (){
        //varijable

                j=0; //počinjemo od 0 jer je ovo MTI verzija
                while(j<number){
                        deltx=t1x-xx[j];
                        delty=t1y-yy[j];
                        recDistanceSQ=attCdeltat/(deltx*deltx+delty*delty+e2);
```

(21)

```
                RecDistanceSQx[id][j]=recDistanceSQ*deltx;
                RecDistanceSQy[id][j]=recDistanceSQ*delty;
                j++;
                }
        j=0;
        while(j<number){
                //.....nemamo ažuriranje po j
                acxtot+=RecDistanceSQx[id][j]; //ažuriraje po i
                acytot+=RecDistanceSQy[id][j];
                j++;

                acxtot2+=RecDistanceSQx[id][j]; //odmotano
                acytot2+=RecDistanceSQy[id][j];
                j++;

                }

        velxx[i]+=acxtot+acxtot2;
        velyy[i]+=acytot+acytot2;
}
```

Ovdje je novost je što smo ovdje drugu petlju jednom odmotali (eng. *unroll*). To znači da smo unutar iste petlje kopirali kôd koji se u njoj izvršava i kada petlja završi jednu iteraciju njezin iterator se neće povećati za +1 nego za +2 ili onoliko puta koliko smo odmotali petlju. Tako u principu radi *for petlja.* Kod *while* petlje jednostavno kopiramo komad kôda i iterator zajedno s njim onoliko puta koliko puta želimo odmotati petlju. Prednost koju dobivamo s odmotavanjem petlje je što se smanjuje *overhead* održavanja petlje. Konkretno imamo manji broj testiranja uvjeta *if* (*j*<*number*).

Osim toga odmotavanjem se može bolje iskorititi paralelizam jezgre. Naime ovdje nismo samo kopirali privremenu varijablu *acxtot* nego smo stvorili drugu *acxtot2* tako da se one mogu paralelno izvršavati.

Bez odmotavanja ova verzija se izvodi za 200 ms, što je značajno unapređenje od prve verzije MTI koja je trajala 307ms. S jednim odmotavanjem petlje dobivamo 162ms. To je oko 24% ubrzanja izvođenja samo zbog odmotavanja petlje! Što ako dodatno odmotamo petlju tako da jedna while iteracija ima 4,8 ili 16 puta odmotanu petlju? Ne treba pretjerivati, ali najbolje je testirati koliko je elemenata optimalno jer ako smo previše puta odmotali petlju dolazi do usporenja. CPU ima 3-4 aritmetičko logičke jedinice po jezgri dok ih GPU ima oko 64 ovisno o generaciji i kompaniji koja ih proizvodi. Zbog toga je odmotavanje puno efikasnije kod grafičkih kartica jer ono iskorištava ovaj ugrađeni paralelizam. Važno je napomenuti da ako odmotamo zapisivanje u polje po *j*, naprimjer *velxxwide*[*id*][*j*] onda dobivamo značajno usporenje izvođenja. U razlog ovoga nismo ulazili. Samo su MTI verzije podobne za odmotavanje petlje.

Primijetimo da smo sada već došli do generalnog oblika naše funkcije i ideje kako bi ona trebala izgledati.

Trebala bi imati razdvojene petlje ukoliko imamo previše operacija u jednoj petlji,da se postigne bolja lokalnost podataka u predmemoriji i registrima, ali ne treba imati previše petlji jer u svakoj petlji mora biti dovoljan broj instrukcija da se iskorištava paralelizam procesora.
Ukoliko je moguće treba ažurirati polja i privremene konstante, a ne samo jedno, jer mjerenja upućuju na to da se ove operacije mogu izvršavati paralelno.
Ukoliko imamo previše kalkulacija uputno ih je spremiti privremeno u neko polje i onda ta rješenja pozivati iz memorije umjesto da ih ponovo računamo.
Petlje odmotati uvijek ukoliko se ažuriraju po *i*.
Čitanja polja su uvijek bolja i brža po *j* to jest po unutarnjoj varijabli jer bolje iskorištavaju cache line strukturu.
Ažuriranja po *j* su također poželjna jer se kod njih također iskorištava paralelizam jezgre i struktura cache linije ali za ažuriranja kod više jezgara moramo koristiti dodatna polja da ne dolazi do problema s koherencijom/sinkronizacijom. Takav pristup zahtjeva više memorije i nije pogodan za mnogo jezgara. Ažuriranja po *j* se ne mogu efikasno odmotavati.

6.7 Memorijski blocking

Problem n-tijela se sastoji od računanja dviju petlji od kojih smo vanjsku *i* petlju paralelizirali. Da objasnimo sljedeći problem, vratimo se na *BasicNbody* koji se sastoji od dvije ugniježđene petlje. Ovdje koristimo *for* petlje zbog kompaktnijeg zapisa.

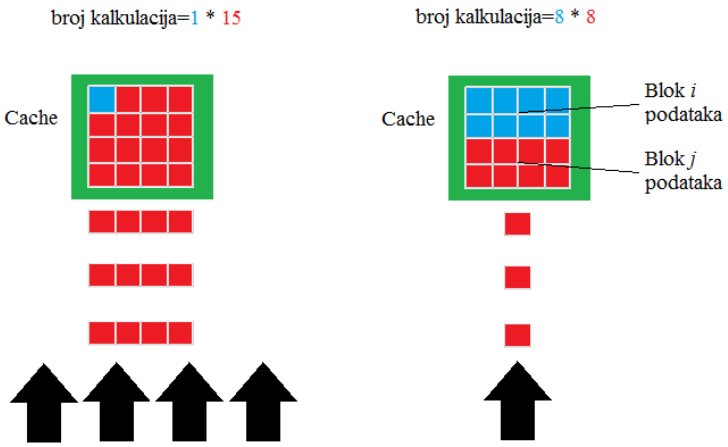
```
for(int i=0;i<number;i++){
                //varijable
                for(int j=0;j<number;j++){
                                deltx=t1x-xx[j];
                                delty=t1y-yy[j];
                                //kalkulacija sile
                }
        }
```
(22)

Za svaku *i*-tu iteraciju moramo proći po cijelom polju po *j* iteratoru. To znači da za svaku *i*-tu česticu moramo dostaviti sve podatke o položaju i provući ih kroz predmemoriju nivoa 1 i jezgru. Ukoliko je broj čestica dovoljno malen da stane u predmemoriju, nemamo problem.

Međutim ako se ti podaci počnu prelijevati u predmemoriju nivoa 2, 3 ili RAM onda stvar nije tako jednostavna. Oni podaci koji su bliže procesoru brže teku jer je propusnost nižih nivoa memorije veća i kada se jednom potroše podaci u tim predmemorijama ostali podaci ne mogu ići brže nego brzinom najsporije memorije u kojoj se nalaze. Zamislimo to kao nekoliko povezanih posuda koje imaju veće otvore kako idemo prema dnu. Voda jako brzo isteče iz najnižih posuda a nakon par sekundi njezina brzina istjecanja je jednaka brzini istjecanja iz najdalje i najuže posude. Stvar ne bi bila problem da se ovaj ciklus ne mora ponoviti za svaku *i*-tu iteraciju to jest *n* puta. Kada se podaci počnu prelijevati u više nivoe memorije oni kroz jezgru teku brzinom najsporije memorije.

Razmotrimo ovo: Svaka *i*-ta čestica dohvati u predmemoriju jedan dio *j*-tih elemenata s kojima računa, nakon što završi račun s njima izbacuje ih iz predmemorije da bi napravila mjesta za sljedeću grupu. Nakon što ih sve prođe sljedeći *i+1* element će dohvaćati te iste podatke i opet ih izbacivati. Zamislimo da umjesto prije nego izbacimo prvu grupu *j*-tih podataka iskoristimo te podatke i napravimo račun s njima za dvije čestice, naprimjer *i* i *i+1*. Sada više ne moramo dozivati sve elemente za *i+1* iteraciju jer smo je već napravili dok su ti podaci bili tamo. Ovo možemo napraviti ne samo za dvije ili tri čestice već za više *i*-tih elemenata. Što više povećamo broj *i*-tih elemenata koje računamo u jednom *bloku* to ćemo imati manje provlačenja cijelog polja podataka kroz memoriju. Recimo da uzmemo *i*-te elemente od 400-500 i *j*-te elemente od 0-100. To je 200 podataka u memoriji. Nakon što izračunamo akceleracije na 100 *i*-tih čestica od 100 *j*-tih čestica uzimamo sljedeći blok *j* podataka od 100-200. Kada izredamo sve *j* elemente idemo na sljedeći blok *i* elemenata od 500-600 i ponavljamo postupak. Broj operacija je isti kao i kod prolaska petlji član po član. Međutim mi ovdje moramo 100 puta manje pozivati cijeli niz podataka po *j* jer za svaki poziv bloka obavimo 100 a ne jednu kalkulaciju akceleracije. U tom vremenu sporiji dijelovi memorije stignu nadomjestiti podatke dok se računa prethodni blok i na ovaj način kompenziramo sporost viših razina memorije. Ovo možemo vidjeti na slici 11.

Slika 11



Lijevo standardna verzija, desno blokovska verzija. Blokovska verzija za jedno punjenje predmemorije napravi 64 kalkulacije dok standardna verzija napravi 15. Ne samo da blokovska verzija zahtjeva 4 puta manju propusnost podataka nego će to zahtjevati 8 puta rijeđe. Zbog ovog je zahtjev za podacima manji.

Generalno takav kôd za jednu jezgru bi izgledao ovako:

```
int block=100;
for(int ii=0;ii<number;ii=ii+block){      // određuje u kojem smo "i" bloku
        for(int jj=0;jj<number;jj=jj+block){ // iterator bloka se povećava za 100
                for(int i=ii;i< ii+block;i++){     //  ii=400     ii+block=500
                        //varijable
                        for(int j=jj; j<jj+block;j++){// jj=0     jj+block=100
                                deltx=t1x-xx[j];
                                delty=t1y-yy[j];
                                //kalkulacija sile
                        }
                }
        }
}
```

(23)

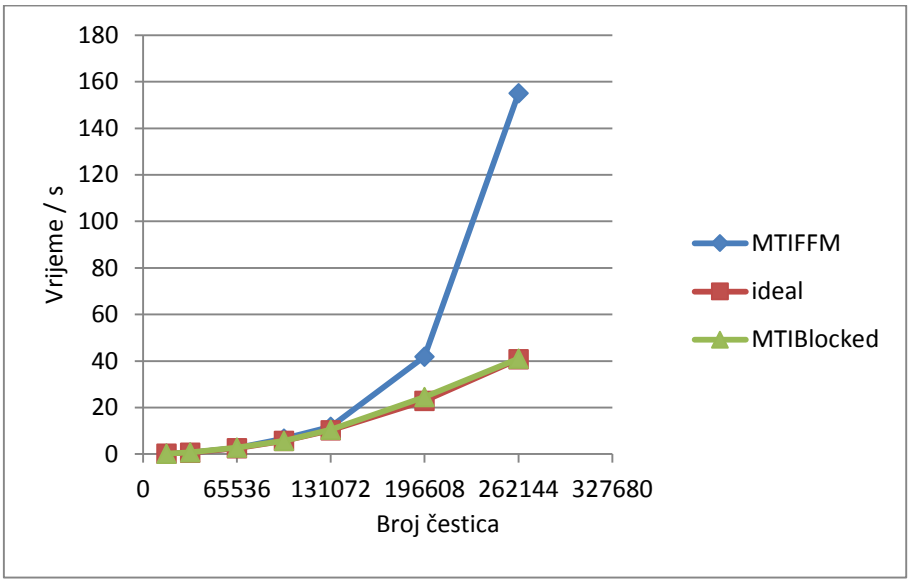
Dodali smo dvije vanjske petlje *ii* i *jj* koje određuju veličinu blokova i njihove granice to jest trenutne blokove koji se računaju. Dvije unutarnje petlje računaju isto kao i prije samo unutar granica blokova koje su definirane vanjskim petljama. Veličina bloka za *i* i *j* petlju ne mora biti ista.

Ovakav način izračuna ne ubrzava račun ukoliko nam se podaci ne prelijevaju u više dijelove memorije to jest ukoliko je propusnost memorije dovoljno velika. Ova metoda optimizacije služi da sprječi gubitak performanse kako povećavamo broj podataka to jest služi da nastavimo dobro skaliranje. Ona se ne može izvesti ako imamo samo jednu petlju već moramo imati barem dvije ugniježđene petlje.

Ukoliko stavimo malu veličinu bloka imati ćemo veliki broj testiranja uvjeta u svim petljama i povećati će nam se *overhead*, tako da ako nismo pažljivi možemo i usporiti račun.

Višedretvena varijanta *blockinga* nije uopće jednostavna. Promotrimo rezultate koje nam metoda *blockinga* daje na procesoru.

Slika 12



MTIFFMU je MTIFissionFusionMemUnroll to jest najbrža izvedba MTI varijante, crveno je teorijski idealno skaliranje, MTIBlocked verzija prethodne varijante napravljena s blokovskom izvedbom. Manje je bolje.

Vrijeme izvođenja treba rasti s kvadratom broja čestica. Idealna varijanta je teoretski procijenjena tako da smo uzeli vrijeme izvođenja *MTIFissionFusionMemUnroll* (T_0) za n=16384 čestice i onda smo pretpostavili ostala vremena izvođenja koristeći $T = T_0 n^2$ ovisnost. Vidimo da *MTIBlocked* dobro slijedi idealnu varijantu dok *MTIFissionFusionMemUnroll* prekida kvadratnu kada taj broj postane prevelik jer dolazi do prelijevanja podataka u više dijelove sporije memorije. Sjetimo se *MTIFissionFusionMemUnroll* ima 4 polja iz kojih čita od kojih se u 2 od ta 4 polja zapisuje svaku *j* iteraciju. *BasicNbodyMTI* varijanta ima samo dva polja iz kojih samo čita i iako radi sporije od bržih MTI varijanti ona neće usporiti svoje izvođenje kako povećavamo broj podataka barem ne u mjerenom intervalu. Zašto nam čitanje nije problem? Usko grlo u čitanju nije RAM jer kada zapisujete podatke u globalno polje koliko god malo da ih imate vi ih na kraju zapisujete u RAM i to ne ide brže od komunikacije s RAM-om, no kada ih čitate onda ih čitate najdalje iz predmemorije u koju stanu. Za 256K čestica imamo 2 MB podataka, procesor ima predmemoriju nivoa 3 od 6MB. Oni su tamo kopirani iz RAM-a samo jednom na početku simulacije a komunikacija s predmemorijom nivoa 3 je 5-10 puta brža od komunikacije RAM-om tako da tu nemamo usko grlo u komunikaciji s memorijom.

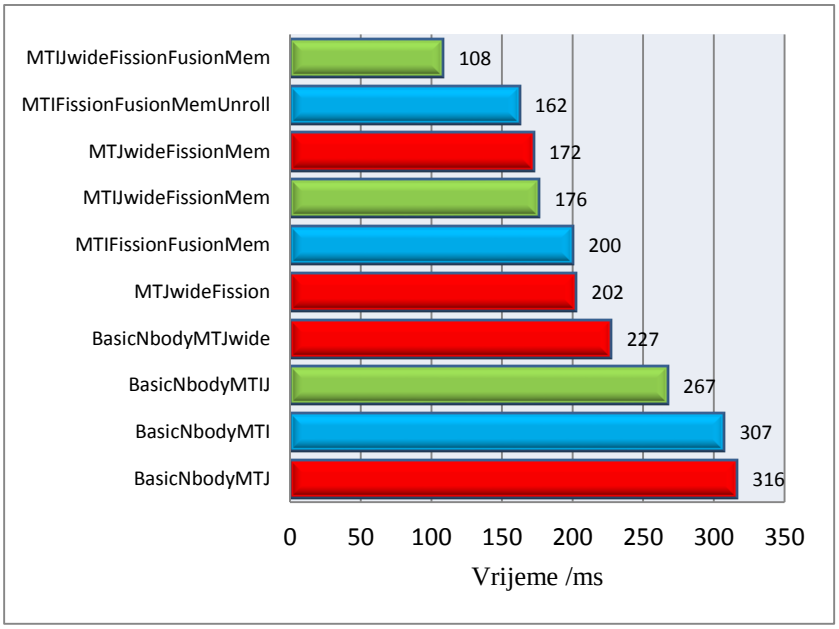
Brzina zapisivanja *MTIFissionFusionMemUnroll* u ovom slučaju ne prelazi 8 GB/s što je blizu teorijske granice našeg procesora od 11 GB/s.

Treba napomenuti da su *MTIBlocked* i *MTIFissionFusionMemUnroll* vremena iz ovog grafa najbolja opažena vremena za te varijante za razliku od svih ostalih mjerenja koja smo radili koja su bila prosječna vremena izvođenja funkcije iz uzorka od 50 mjerenja to jest sličica. Također smo varirali veličinu bloka i broj dretvi koji se izvode na procesoru metodom pokušaja i pogreške da postignemo najbolji mogući rezultat. Ukoliko je broj računskih operacija mnogo veći od broja memorijskih zahtjeva odgađa se potreba za memorijskim *blockingom*. Tip računa koji mnogo dobiva od bilo kakvog *blockinga* je množenje matrica. Taj račun u sebi ima 3 ugniježđene petlje i samo dvije operacije s pomičnim zarezom po svakoj iteraciji.

6.8 Usporedba performansi

Prošli smo više manje sve hardverske optimizacije koje ćemo ili bismo mogli koristiti kada prijeđemo na GPU hardver. Rezimirajmo performanse svih dosadašnjih funkcija na sljedećoj slici.

Slika 13



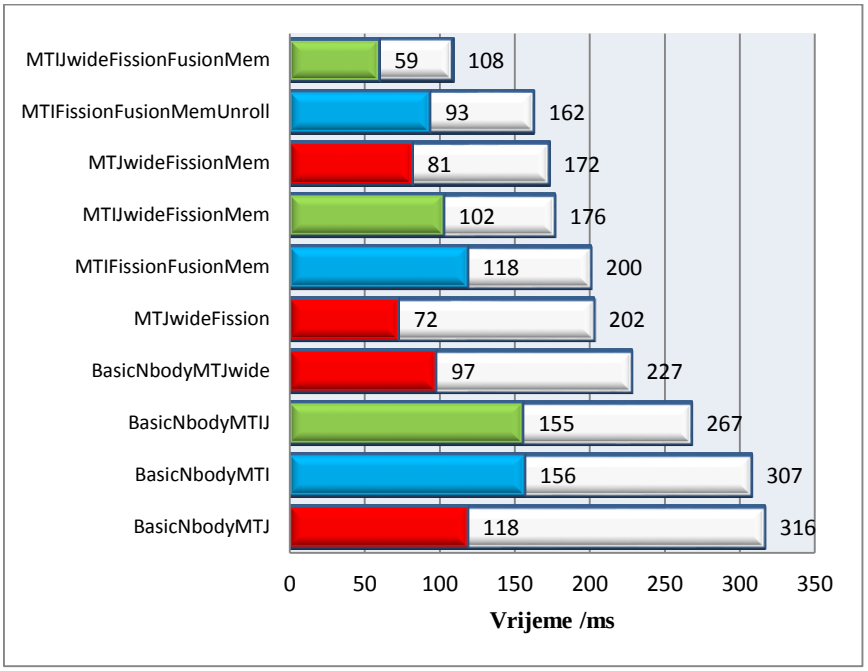
PhenomII x4 3.2 Ghz n=16384 čestica, manje je bolje

Sve simulacije se izvode za n=16384 na 4 jezgre. Iako se *BasicNbodyMTJ* (kôd (7),840 ms) izvodi brže od *BasicNbodyMTI* (kôd (4),1188 ms) na jednoj jezgri, zbog lošeg skaliranja na 4 jezgre osnovna MTI verzija je malo brža (307 naprama 316 ms). *BasicNbodyMTIJ* (kôd (13)) je brži od prethodnih varijanti uglavnom zbog manje kalkulacija (267 ms) ali generalno podbacuje s obzirom na smanjen broj kalkulacija. *BasicNbodyMTJwide* (kôd (14), 227 ms)

ima pomoćna polja s kojima se skoro pa linearno skalira i ovdje iskorištava svoju prednost nad MTI verzijom, a to je da iskorištava paralelizam jezgre ažuriranjem rješenja po *j*. Razdvajajući while petlju na 2. petlje kod *MTJwideFission* verzije (kôd (18), 202 ms) u svakoj pojedinoj petlji imamo manje opterećenja na registre i predmemoriju pa se postiže bolja lokalnost, ali zato imamo veći broj kalkulacija (18) za jedan *j*-ti element jer neke ponavljamo. U *MTJwideFissionMem* (kôd (19), 172 ms) verziji razdvajamo račun na 3 petlje od kojih jedna računa akceleracije i pamti izračune umjesto da ih ponavlja tako da se opet vraćamo na 13 kalkulacija po *j* tom elementu te se računske operacije zamjenjuju memorijskim dohvatom. Istu stvar radimo i s *MTIJwideFissionMem* gdje koristimo razdvajanje petlji i pamćenje rezultata ali zbog ažuriranja dviju sila odjednom imamo 5 petlji a ne 3. Iako ova varijanta ima 2 puta manje kalkulacija nego prethodna brzina je otprilike ista (176 ms). Napravili smo i MTI verziju koja nije u grafu i koja poput *MTJwideFissionMem* ima 3 petlje od kojih jedna računa i pamti rješenja, a druge dvije ažuriraju rješenja. Takva MTI verzija ima oko 260 ms i zaostaje značajno za svojom *J* verzijom kao i sve *I* verzije do sada. *MTIFissionFusionMem* (200 ms) je njoj slična samo što spaja dvije petlje koje ažuriraju podatke u jednu i time postiže bolji paralelizam jezgre. Ako takvoj funkciji dodamo i odmotavanje petlje u petlji koja se ažurira dodatno iskorištavamo paralelizam te dobivamo *MTIFissionFusionMemUnroll* (kôd (21)) verziju koja je prva *I* verzija koja je brža od slične *J* verzije (162 naspram 172 ms). Odmotavanje se nije pokazalo pogodnim za *J* verzije isto kao i spajanje petlje. Na kraju dobivamo *MTIJwideFissionFusionMem* ((kôd (20)) koji prijašnjih 5 petlji spaja u dvije miješajući ažuriranja podataka u privremene varijable i polja te na taj način iskorištava paralelizam u jezgri. Smanjena memorijska komunikacija kao posljedica toga omogućuje kraće izvođenje MTIJ varijante koja ionako ima dvostruko manje kalkulacija nego ostale. Za razliku od prijašnjih MTIJ varijanti koje su u usporedbi sa svojim MTI ili MTJ varijantama podbacile u očekivanjima ova varijanta ostvaruje svoj potencijal i jedna sličica se izvodi za 108 ms. Krenuli smo od naivnog *BasicNbody* modela za koji nam je trebalo 1188 ms na jednoj jezgri, preko najjednostavnije paralelizacije za više jezgara *BasicNbodyMTI* koji se izvodi za 307 ms došli smo do MTIJ verzije koja ide 3 puta brže od toga! Iako smo istražujući došli do generalnih recepata kako optimizirati kôd i dalje moramo metodom pokušaja i pogreške testirati jesmo li ostvarili ubrzanje.

Kao referentni procesor koristili smo Phenom II x4 na 3.2 Ghz. Međutim testiranja smo također napravili i na drugim procesorima. Dolje je slika performansi na i5 3570 procesoru fiksiranom na 3.4 Ghz.

slika 14

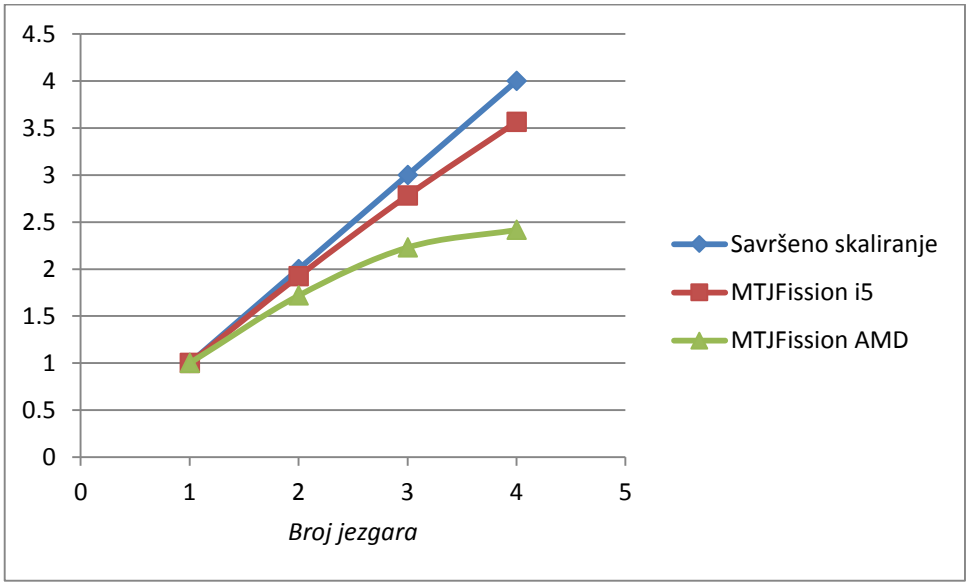


Performansa u milisekundama pojedinih varijanti na i5 3570 četverojezgrenom procesoru fiksiranom na 3.4 Ghz. Broj čestica =16384. Manje je bolje.

Promotrivši rezultate odmah možemo primijetiti da je i5 generacije *Ivy Bridge* 2 do 3 puta brži od Phenoma II. Ovo ubrzanje ne dolazi samo od naprednije ALU jedinice nego i od znatno bolje memorije koja ima oko 2-3 puta veću propusnost i oko 2 puta manju latenciju između predmemorije nivoa 2 i 3. No možemo primijetiti još nešto. Crvene to jest MTJ varijante su značajno brže na i5 u usporedbi s MTI i MTIJ varijantama na istom procesoru. Na jednoj jezgri i5 procesora se osnovna MTI varijanta izvodi za 624 ms, MTJ varijanta se izvodi za 376 ms, što je 66% brže od MTI varijante. Sjetimo se ovaj postotak je na *AMDovom* stroju bio 41% (841/1188). Što bi značilo da i5 bolje iskorištava paralelizam ili svoju cache liniju kada koristi MTJ varijantu, što je i očekivano zbog naprednijeg ALU-a. "wide" varijanta i "fission" varijanta dodatno povećavaju jaz između *AMDovog* i *Intelovog* procesora.

Promotrimo sljedeću sliku:

slika 15

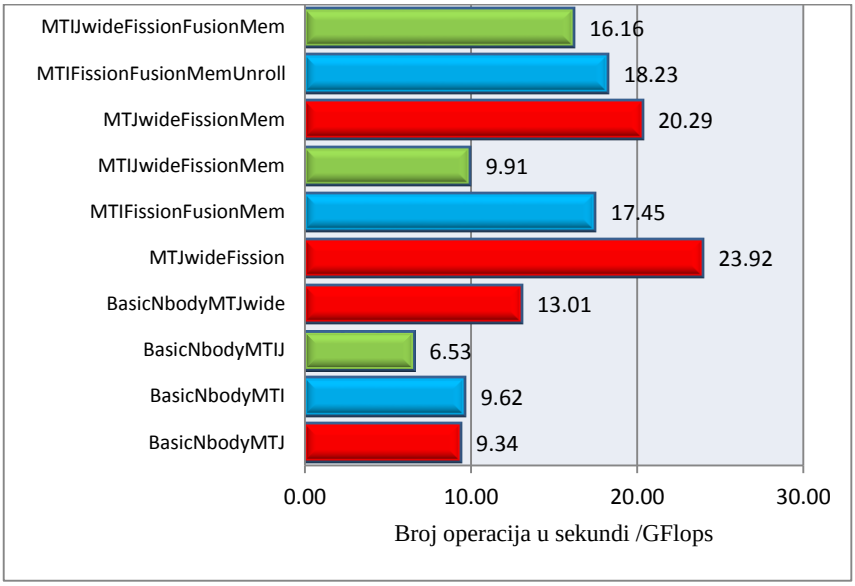


Skaliranje MTJFission varijante na Intelovom i AMDovom stroju

MTJFission varijantu nismo diskutirali ali kako joj samo ime kaže to je MTJ varijanta s razdvojenom while petljom u dvije petlje. Ima 18 kalkulacija po jednoj iteraciji *while j* petlje. Brzine se ažuriraju jedna po jedna, prvo *velxx* pa u drugoj petlji *velyy* i to direktno a ne preko pomoćnih "wide" polja. Ova varijanta trpi od lažnog dijeljenja i problema s koherencijom ali Intelov stroj to ipak podnosi mnogo bolje. Zašto? Intelovi procesori unatrag nekoliko generacija imaju ugrađen *ring bus* koji ubrzava komunikaciju između dijelova predmemorije nivoa 3 i ostalih elemenata efektivno povećavajući propusnost i smanjujući latenciju među jezgrama. Iz ovog razloga se nakon lažnog dijeljenja i brisanja podataka iz predmemorije jedne jezgre puno brže dostave novi podaci pa se skaliranje može dobro nastaviti. Na *AMDovom* stroju se ova varijanta na 4 jezgre izvodi za otprilike 310 ms dok se na Intelove 4 jezgre izvodi za 78 ms. Skoro 4 puta brže! *Ring bus* omogućava da se nastavlja skaliranje, brža memorija omogućava da se ALU dovoljno brzo hrani potrebnim podacima dok snažni ALU bolje iskorištava paralelizam i cache liniju.

Promotrimo broj operacija s pomičnim zarezom po sekundi. Na slici 16 je njihova tablica za Phenom II procesor.

Slika 16



Phenom II x4 3.2. Performansa u giga operacijama s pomičnim zarezom u sekundi, više je bolje.

Ako gledamo po bojama od dna prema vrhu vidimo kako s rastućom performansom raste i iskoristivost računskog potencijala procesora. Primjećujemo da ekvivalentne MTI (plave) i MTJ (crvene) varijante imaju otprilike isti broj kalkulacija za razliku od MTIJ (zeleno) varijanti koje imaju dvostruko manje operacija s pomičnim zarezom, uz jednu iznimku *MTJwideFission* koja ponavlja neke svoje kalkulacije pa stoga ima i mnogo više operacija. Zabilježeni potencijal operacija s pomičnim zarezom Phenom II procesora je oko 38 gigaflopsa i dobiven je u *linpack* (intel burn test) *benchmarku*. Vidimo da je računski potencijal procesora mnogo veći od postignutih performansi pogotovo kada vidimo *MTJwideFission* funkciju koja nije niti najbrža ali najviše iskorištava računski potencijal. Međutim ako uzmemo u obzir da operacija dijeljenja traje oko 5 puta duže nego operacija množenja i zbrajanja brojevi na slici 16 su minimalni broj računskih operacija koje procesor izvršava. Maksimalni broj operacija možemo procijeniti tako da operaciju dijeljenja računamo kao 5 operacija s pomičnim zarezom pa tako broj operacija više nije 11,13 i 18 za MTI, MTIJ i *MTJwideFission* nego je 15,17 i 26. Posljednja verzija ima 26 kalkulacija jer ima dvije operacije dijeljenja. Uzevši to u obzir dobijemo da je računska performansa *MTJwideFission* varijante $\frac{26}{18} * 23.92\ Gflop = 34.55$ Gflops. Što je približno našem očekivanju od 38 Gigaflopsa za Phenom II procesor. Ipak za ostale varijante smo daleko od tog potencijala. Za i5 očekujemo oko 90 Gflops-a koliko daje *linpack* test. Najbrža varijanta je *MTIJwideFission* koja izračuna jednu sličicu za 72 ms. Ona ima od 18-26 operacija s pomičnim zarezom ovisno kako računamo operaciju dijeljenja te performansa iznosi od 67.1 do 96.93 Gflopsa. Ovo su

zaista impresivne brojke. U većini realnih aplikacija ne priđe se niti blizu ovim brojkama. Također u većini realnih aplikacija intelov procesor nije niti 50% jači od *AMDovog.* Iz očitih razloga mjere u gigaflopsima su malo nezgodne pogotovo zato jer složenije operacije poput korjenovanja ili dijeljenja ne traju isto kao množenje ili zbrajanje, također ne traju isto na različitim procesorima ili čak na grafičkoj kartici. Uzevši ovo u obzir koristimo *donju* procjenu operacija s pomičnim zarezom gdje se dijeljenje i korjenovanje računaju kao jedna operacija.

Da bismo bolje razumjeli u čemu je zapravo problem kod iskorištavanja ALU-a morali bismo zaroniti u assemblerski kôd procesora i arhitekturu same jezgre. U ovom diplomskom to nismo radili.

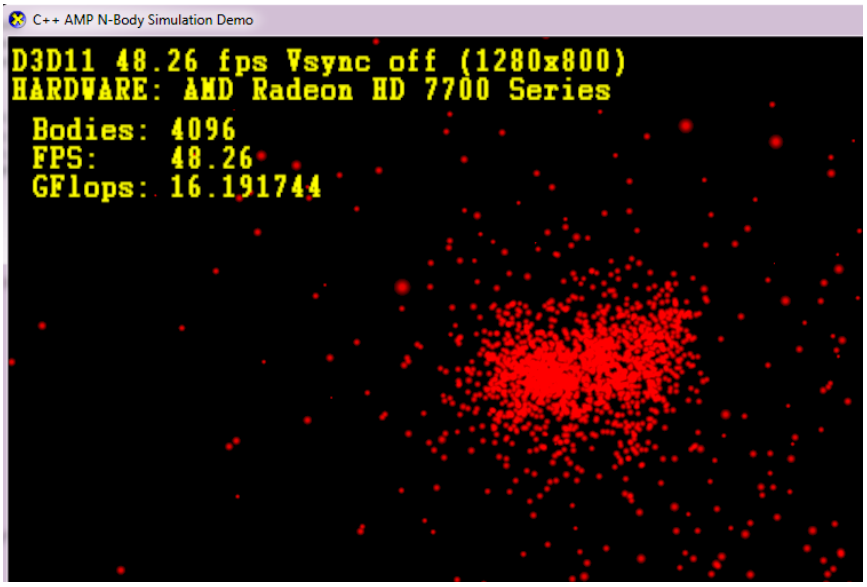
Međutim mogli smo opaziti trendove, a to je da za postizanje maksimalne performanse procesora nije bitno samo minimalizirati računske operacije već napraviti balans između broja računskih i memorijskih operacija, lokalnosti u memoriji, iskorištavanja cache linije, korištenja registara i korištenja paralelnih struktura u samoj jezgri procesora.

MTJ varijante pokazuju prednost u performansi zbog boljeg iskorištenja paralelizacije procesora ta se osobina može nadomjestiti odmotavanjem petlje kod MTI varijanti. Također se ažuriranjem preko MTJ varijante dolazi do problema sa sinkronizacijom i koherencijom (lažno dijeljenje, *race condition*[*]). Ovi se problemi mogu prijeći uvođenjem pomoćnih *wide* polja i dalje ostaju problemi s učestalim zapisivanjem i korištenjem memorijske propusnosti koji može dovesti do problema sa skaliranjem ili zauzimanjem mnogo memorijskog prostora u RAM-u. Neki od tih problema mogu se prijeći memorijskim *blockingom*. Ažuriranje po *j* je samo po sebi kompleksnije jer trebamo voditi računa o više stvari poput koja dretva što ažurira. Iz ovog razloga je jednostavnije je nastaviti s MTI verzijama, a iskorištavanje paralelizacije nadoknaditi odmotavanjem petlji.

Analizirajmo nakratko kako ovisi performansa u usporedbi s drugim programskim jezicima. Imamo primjer jednog programa problema n-tijela u 3 dimenzije koji je pisan u C++ i da bi se izvodio na grafičkoj kartici koristi AMP biblioteku (*Accelerated Massive Parallelism*). AMP je baziran na DirectX-u.

[*] Objašnjeno na stranici 56

slika 17



C++ 3D implementacija za procesor, koristi SSE instrukcije

Zasada još ne razmatramo GPU tako da se ova C++ 3D implementacija izvodi na procesoru i ne koristi AMP biblioteku ali zato koristi SSE instrukcije koje iskorištavaju paralelizam procesora. 3D problem računa izraz za silu prema $\frac{\Delta x}{r^3}$ i odgovara potencijalu ovisnosti $\frac{1}{r}$, za razliku od naše implementacije koja ima $\frac{\Delta x}{r^2}$ izraz za silu i odgovara potencijalu od $\ln(r)$. Da bismo dobili izraz r^2 trebamo samo pomnožiti razlike u udaljenostima:

$$r^2 = \Delta x\Delta x + \Delta y\Delta y + \Delta z\Delta z \tag{12}$$

Dok račun za r^3 treba i korijen:

$$r^3 = \sqrt{(\Delta x\Delta x + \Delta y\Delta y + \Delta z\Delta z)^3} \tag{13}$$

Iz ovog razloga r^3 ovisnost ali i svi računi sile i potencijala koji koriste ovisnost u kojem je potencija na radijusu neparan, zahtijevaju korjenovanje a time i više kalkulacija od parnih varijanti sile i potencijala. Zbog korjenovanja i dodatne dimenzije položaja i brzine u *z* smjeru broj kalkulacija za 3D implementaciju je 20 računskih operacija za jednu iteraciju *while j* petlje kod obične MTI varijante gdje se korjenovanje i dijeljenje računaju pod jednu operaciju svaka zasebno.

3D MTI varijanta u Javi za 4096 čestica radi na 21 FPS-u i 7.13 Gflops (manja mjera računskih operacija).

Ekvivalent MTI varijante u C++ pojačan sa SSE instrukcijama radi na 48 FPS-a i 16.3 Gflops.

C++ radi više od 2 puta brže nego Java, ne znamo točno koliko se performanse dobiva od SSE instrukcija ali ovo je očekivano jer je C ipak brži od Jave. Iznenađenje je što neke od ostalih 3D MTJ ili MTI varijanti nisu pokazale značajno ubrzanje ili su čak radile sporije. Vjerojatno je problem računski intenzivna operacija korjenovanja. U dublju analizu ovoga nismo ulazili ali iz ovoga možemo zaključiti:

Metode optimizacije nisu garancija ubrzanja već recepti koji mogu ali ne moraju ubrzati aplikaciju.

7 GPU

7.1 GPU hardver i generalni princip rada

Grafička kartica ima oko 3-4 puta manju frekvenciju od tradicionalnih procesora. Najmanja latencija njezine predmemorije nije ispod 100 ciklusa a maksimalna ide i do 400. Ima više registara nego procesor ali ima manje predmemorije. Njezine aritmetičko logičke jedinice su sposobne za manje instrukcija nego one od procesora. GPU može obavljati operacije 64 bitne preciznosti (dvostruka preciznost) ali time gubi performansu ugrubo oko 10 puta za razliku od procesora koji gubi samo dvostruko performanse. Ovi brojevi variraju od generacije do generacije i također variraju ovisno o kompaniji koja je proizvela hardver, te o tome da li je hardver profesionalan* ali su dosljedni unutar reda veličine. CPU može obavljati sve više i više složenijih instrukcija poput starijih FMA3 ili FMA4 ili novih 256 bitnih AVX instrukcija. Bitna razlika u hardverskom dizajnu grafičke kartice naspram procesora je što ona ima *lokalnu memoriju*. Ovo ćemo još poslije objasniti. U sljedećoj tablici vidimo potpune performanse hardvera.

* Profesionalni grafički hardver ne gubi 10 puta performansu nego samo 2 puta te ima mnogo manju latenciju nego igraći hardver.

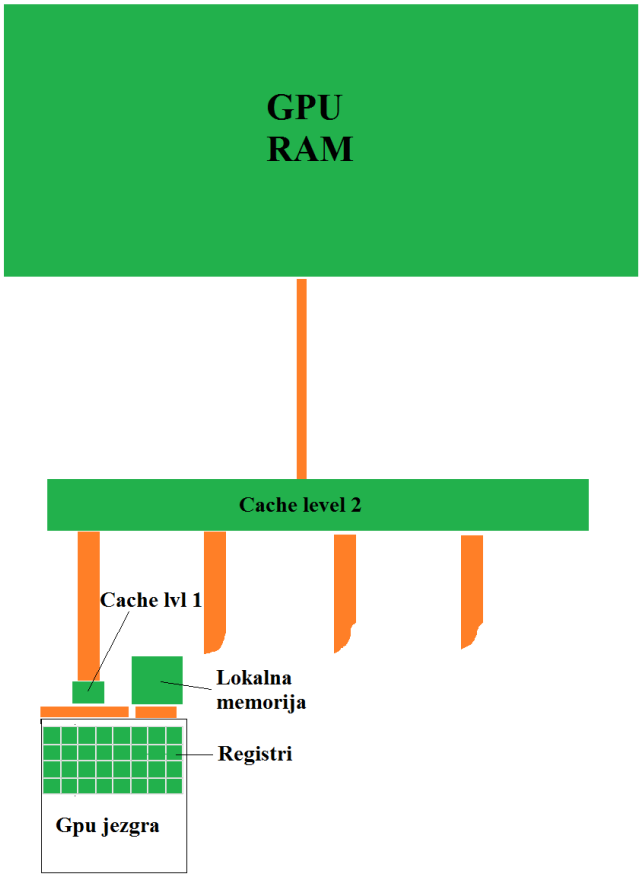
Tabela 4

	PhenomII x4	i5 3570	Radeon 5670	Radeon 6770	Radeon 7790
Frekvencija	3.2 Ghz	3.4-3.8 Ghz	0.75 Ghz	0.85 Ghz	1.05 Ghz
Broj jezgara	4	4	5	10	14
ALU	12	16	400	640	896
Očekivana performansa	38 Gflops	90 Gflops	310 Gflops	650 Gflops	940 Gflops
Propusnost podataka	11 GB/s	27 GB/s	44 GB/s	67 GB/s	86 GB/s
Broj registara po jezgri	<1000	<1000	16384	16384	65536*
Lokalna mem. po jezgri	Nema	Nema	32 KB	32 KB	64 KB
Cache lvl 1 po jezgri	64 KB data 64 KB instr	32 KB data 32 KB instr	8 KB	8 KB	16 KB
Cache lvl 2 total	2 MB	1 MB	128 KB	256 KB	512 KB
Cache lvl 3 total	6 MB	6 MB	Nema	Nema	Nema
Latencija min**	0.9 ns	1 ns	410 ns	-	210 ns
Latencija max***	14.4 ns	-	630 ns	-	410 ns

**registri kod Radeona 7790 su zrnatiji ali je ukupan kapacitet u usporedbi s prijašnjim generacijama isti*
***Minimalne latencije su izmjerene za sekvencijalno čitanje i pisanje.*
****Maksimalne latencije su izmjerene za "in page sequential read and write".*

Ugrubo, tradicionalni procesori su građeni da jednu dretvu izvršavaju jako brzo i da nema puno čekanja dok je grafička kartica napravljena da računa veliki broj podataka i da ima puno veću računsku propusnost (eng. *throughput*). Iznimno velika latencija kod grafičke kartice je posljedica njezine prvotne namjene. U renderiranju grafike velika latencija nije značajno ograničenje, ali u programiranju opće namjene ona je nešto nepoželjno. Unatoč tome postoje načini da se ona zaobiđe. Memorijska hijerarhija grafičke kartice je malo drukčija od procesorske. Promotrimo sljedeću sliku:

slika 18



Kao i kod sheme procesora udaljenosti među memorijama otprilike predstavljaju latenciju a debljine podatkovnu propusnost. Vidimo da GPU nema predmemoriju nivoa 3 već njegovu funkciju obavlja predmemorija nivoa 2. Vidimo da se količina memorije povećava kako prilazimo jezgri, suprotno od procesora. Predmemorija nivoa 2 ima 512 KB, predmemorija nivoa 1 ima 25 puta manje memorije (16 KB) ali zato registri svake jezgre imaju 256KB memorije (pola predmemorije nivoa 2). Ovo se zove *privatna memorija*. Kod Radeona serije 7000 u registre se može staviti preko 65000 podataka veličine 4 bajta to jest 65000 podataka tipa *float* ili *int*. Registri pojedine jezgre imaju 4 puta više memorije nego predmemorija nivoa 1 u procesoru! Razlog ovoga ćemo objasniti za trenutak.

Osim toga svaka jezgra ima *lokalnu memoriju*. Lokalna memorija ima 64KB veličine međutim ona se može dijeliti među dretvama grafičke jezgre.

Za razliku od procesorske jezgre na kojoj se u jednom trenutku izvršava samo jedna dretva, u grafičkoj jezgri se u jednom trenutku može izvršavati nekoliko dretvi! Ovo je razlog zašto ona ima tako mnogo registara - jer u jednom trenutku mora imati dovoljno memorije za 64 dretve. Ovim se lako iskorištavaju 64 aritmetičko logičke jedinice koje ima jedna grafička jezgra Radeona generacije od 6000 pa nadalje. Podaci pojedinih dretvi zauzimaju prostor u

registrima (privatnoj memoriji) i takve dretve međusobno ne vide podatke jedne drugima. Međutim podaci koji su prisutni u lokalnoj memoriji i predmemoriji su vidljivi svim dretvama jedne jezgre koje se u danom trenutku izvršavaju. Lokalna memorija je po ovome slična predmemorji. Elementi ovih dretvi se izvršavaju zajedno (eng. *lock step*). Optimalno je da koriste iste instrukcije i rade na istoj grupi podataka. Ovo se zove valna fronta i ona se izvršava brzinom najsporijeg elementa. Nekoliko valnih fronti čini radnu grupu (eng. *workgroup*). Podaci koji se dijele u lokalnoj memoriji ali isto tako u predmemoriji vide samo elementi jedne radne grupe. Kada pozivamo podatke u lokalnu memoriju oni se samo jednom pozivaju iz RAM-a grafičke kartice i dostavljaju u lokalnu memoriju te su dostupni samo jednoj radnoj grupi. A kada svaki sljedeći put neka *i*-ta dretva zatraži *i*-ti ili *j*-ti element iz te grupe oni se dostavljaju iz lokalne memorije a ne iz RAM-a. Ovo je puno bolje jer lokalna memorija ima puno veću propusnost prema registrima i puno manju latenciju od RAM-a grafičke kartice. Iz ovoga možemo zaključiti da lokalna memorija služi kao međuspremnik. Podaci se u lokalnu memoriju moraju dostavljati u blokovima/radnim grupama po uzoru na blokovsko dohvaćanje podataka iz memorije. U OpenCL-u naredbe za prebacivanje podataka iz RAM-a u lokalnu memoriju trebaju se dati eksplicitno jer obični memorijski zahtjevi za podacima iz RAM-a podatke zovu direktno u privatnu memoriju to jest registre.

Nije sasvim jasno je li Aparapi automatski koristi lokalnu memoriju ali ćemo pretpostaviti da ne koristi. Postoje naredbe u *Aparapiju* koje bi trebale dozvoljavati eksplicitno korištenje lokalne memorije ali ne rade ili njihovo korištenje nije dovoljno dobro dokumentirano tako da lokalnu memoriju nažalost nismo koristili. Ipak zbog njezinih posebnosti tu vrstu memorije vrijedi spomenuti jer je GPU po svemu osim po lokalnoj memoriji sličan procesoru.

Postoji još i *constant memory* koji nije velik i u njegovu funkcionalnost i primjenu nismo ulazili.

Da rezimiramo. U usporedbi s procesorom koji ima manju i bržu memoriju kako se približava jezgri grafička kartica ima više brze memorije u samim jezgrama nego u ostatku cache hijerarhije. Ima ogromne latencije ali i puno veću propusnost od procesora. Napravljena je tako da rjeđe prenosi velike količine podataka na kojima se onda izvodi mnogo kalkulacija. Ovo opravdava visoku latenciju. Osim toga grafička kartica ima lokalnu/dijeljenu memoriju tik do registara koja služi kao međuspremnik za registre. Na jednoj jezgri se mogu izvršavati 64 dretve, ovo je pogodno jer one mogu dijeliti podatke koji se nalaze u predmemoriji ili lokalnoj memoriji pa time rasterećuju sabirnicu.

Naša grafička kartica (Radeon 7790) ima 14 jezgara u usporedbi s procesorom koji ima 4. Ima ih koje imaju i preko 40 jezgara. Svaka grafička jezgra ima 64 aritmetičko logičke jedinice za

razliku od procesora koji ih ima 3 do 4. Usporedba GPU-a i procesora je nešto kao usporedba sporog trajekta s mnogo putnika koji pristaje na veliko pristanište u kojem ima mnogo mjesta za parkiranje (registara) i brzog glisera s malo putnika koji pristaje na malu rivu s ograničenim mjestom za prihvat putnika. Trajekt ima mnogo veću računsku i podatkovnu propusnost od glisera ali mu treba mnogo vremena da preveze mali broj putnika. Procesor je u tome efikasniji.

Važno je napomenuti da kod grafičke kartice ne postoji sinkronizacija podataka izvan pojedine grafičke jezgre osim ako je na neki način nismo isprogramirali. Podaci se automatski dijele i sinkroniziraju samo unutar jedne grafičke jezgre. To znači da ukoliko imamo MTJ varijantu programa koja zapisuje u svakoj *i*-toj instanci u cijelo polje, jedna jezgra GPU-a to čini bez svijesti da to čine i ostale jezgre. Ako naprimjer jedna jezgra zbraja broj 3 s naprimjer brojem 12 koji se već nalazi na nekoj memorijskoj lokaciji, a druga zbraja broj 7 na to isto mjesto, umjesto da se ti podaci zbroje u 12+3+7=22 oni će se prepisati jedan preko drugog i s 12 će se zbrojiti ili 3 ili 7 u 15 ili 19 ili nešto nedefinirano. Jezgre se "utrkuju" da zapišu podatak i ovo se zove *race condition*. Ovakvo izvođenje problema n-tijela je i vizualno drukčije tj. nije fizikalno ispravno. Ukoliko se zapisivanje događa između dretvi koje se izvršavaju na jednoj grafičkoj jezgri onda će se stvar sinkronizirati, ali ako se zapisivanje događa s dretvama nekoliko jezgara onda među njima imamo *race condition* i dobiveni rezultat nije fizikalno ispravan. Stvar se dodatno komplicira time da se svaka dretva nakon nekog broja elemenata (naprimjer 40) prebacuje na drugu grafičku jezgru da bi se sve jezgre opteretile. Ovo Aparapi radi automatski. Ako imamo jednu dretvu koja se izvodi na grafičkoj kartici, komponente te dretve će se prebacivati na sve jezgre te će međusobno interferirati. Što znači da će jedna dretva MTJ varijante interferirati sama sa sobom jer se izvodi na više jezgara, što nije slučaj kod izvođenja na procesoru gdje se ta dretva izvodi samo na jednoj jezgri. Iz ovog razloga ali i iz prethodno nabrojanih problema s MTJ varijantom dalje nastavljano s MTI varijantom programa.

Na početku smo rekli da se jedan te isti kôd može izvršavati na grafičkoj kartici kao i na procesoru. Koja su nam očekivanja?

Ako pogledate u specifikacije Radeona 7790 vidjeti će te da je njegova teorijska performansa oko 1.84 TFlopsa. Teorijske performanse grafičkih kartica kao i procesora zapravo nikada nisu ostvarene. Kod Radeon generacije kartica ali i drugih, realne performanse nikada ne prelaze 50-60% navedenog teorijskog potencijala.

Aritmetičko logičke jedinice su sposobne napraviti operaciju FMA (*fused multiply add)* u jednom ciklusu.

$$A = A * (B + C) \tag{14}$$

Što znači da u jednom ciklusu mogu obaviti dvije kalkulacije. Ove dvije kalkulacije ulaze u ovaj teoretski plafon. Međutim ovo bi se moglo iskoristiti samo ukoliko su sve kalkulacije koje se vrše isključivo operacije množenja i zbrajanja ali čak i kod matričnog množenja čiji se račun isključivo svodi na formulu (14), postignuta performansa ne prelazi 50% teorijskog potencijala. Problem proizlazi iz nemogućnosti da se ALU jedinice dovoljno brzo hrane podacima. Zato su nam očekivane performanse tek pola tog potencijala.

7.2 GPU bez optimizacija

Daljnja mjerenja performansi ćemo izvoditi za 65536 ili 64K čestica jer grafička kartica ima bolju performansu na većem broju elemenata. Ovo ćemo isto kasnije diskutirati. Mjerenja ćemo normirati prema izvođenju *BasicNbodyMTI* varijante od 65536 čestica na Phenom II procesoru koje iznosi 4830 ms ili 9.78 Gflops.

Na Radeonu 7790 se jednu sličicu za 64K čestica izvodi za 121 ms dok je na Radeonu 5670 to vrijeme jednako 1542 ms. Što znači da je Radeon 7790 inicijalno 40 puta brži od procesora a Radeon 5670 samo 3 puta! Ako uzmemo konzervativnu procjenu računskih operacija Radeon 7790 ima 384 Gflopsa od očekivanih 940, dok Radeon 5670 ima oko 30 Gflopsa od očekivanih 310. Noviji Radeon postiže malo više od jedne trećine očekivane performanse dok stariji postiže tek jednu desetinu očekivane performanse. Ako uzmemo u obzir da je cjenovno Radeon 7790 dva puta skuplji od Phenom procesora, a Radeon 5670 dvostruko jeftiniji, brojke su impresivne makar su one još jako daleko od realnih performansi.

Na performansu Radeona 5670 utječe i veličina radne grupe koju smo ostavili na 64. To znači da se pokreću 64 dretve koje se raspodjeljuju po pojedinim grafičkim jezgrama. Ovo je minimalna optimalna veličina radne grupe, ostale optimalne veličine su višekratnici broja 64. Veličinu radne grupe i njezin utjecaj na performansu nećemo previše diskutirati jer prema mjerenjima ispada da se korist od manipuliranja veličinom radne grupe može nadomjestiti i premašiti jednostavnim odmotavanjem petlje. Najsigurnije je ovaj broj jednostavno ostaviti na 64.

7.3 rsqrt funkcija

Grafička kartica vjerojatno zbog potrebe računanja prostornih koordinata za svrhu renderiranja u 3D prostoru jako brzo izvodi operaciju korjenovanja (sqrt) i recipročnog korjenovanja (rsqrt). Brzina ovih operacija je usporedive brzine s brzinom množenja i dijeljenja. Tako da umjesto operacije dijeljenja s kvadratom radijusa:

```
distanceSQ=deltx*deltx+delty*delty+e2;
recDistanceSQ =1/distanceSQ;
```
(8)

Koristimo operaciju recipročnog korjenovanja:

```
distanceSQ=deltx*deltx+delty*delty+e2;
recDistanceSQ=rsqrt(distanceSQ*distanceSQ);
```
(9)

Ovo je zapravo računski isto kao i dijeljenje s distanceSQ samo što se na grafičkoj kartici odvija brže nego samo dijeljenje.

Naša MTI varijanta tada postaje:

```
while(j<number){
        deltx=t1x-xx[j];
        delty=t1y-yy[j];
        distanceSQ=deltx*deltx+delty*delty+e2;
        recDistanceSQ=1/distanceSQ;                  //za procesor
        recDistanceSQ=rsqrt(distanceSQ*distanceSQ); //za grafičku,međusobno isključivo
        acxtot+=deltx*recDistanceSQ;
        acytot+=delty*recDistanceSQ;
        j=j+1;
        }
```
(10)

Nije se dogodilo ništa spektakularno, u standardnoj MTI varijanti imamo 5 linija kôda a ovdje 6 jer *recDistanceSQ* računamo u dvije linije kôda a ne jednoj. Za kompajler nema razlike. Primijetimo da je *attractiveConstant* sada izvan *while j* petlje. MTI varijanta koja se izvodi na grafičkoj kartici preimenovana je u *GPUMTI* i ima 12 računskih kalkulacija dok je njena procesorska verzija preimenovana u *CPUMTI* i ima i dalje 11 kalkulacija.

Performansa ovog kôda je 99.4 ms i 520/(max 940) Gflops na Radeonu 7790 što je ubrzanje od oko 21%. Na Radeonu 5670 ona iznosi 1473 ms i 31.5/(max 310) Gflops što je ubrzanje od oko 5%. Kôd za grafičku karticu se može izvoditi na procesoru ali izvođenje traje otprilike dvostruko dulje nego standardni CPUMTI kôd zbog toga što se korjenovanje ne izvodi u jednom ciklusu.

7.4 GPU odmotavanje petlje

Odmotavanje petlje smo već objasnili. Jednostavnim kopiranjem kôda unutar *while* petlje bolje iskorištavamo paralelizam unutar jezgre i smanjujemo *overhead* to jest broj testiranja uvjeta izlaska iz petlje. Međutim različitom hardveru pašu različiti broj puta odmotavanja petlje. Naprimjer, na Phenom II nema koristi od odmotavanja petlje više od dva puta. Radeon 5670 radi optimalno kada je petlja odmotana 8 puta ali ne i više, dok Radeon 7790 nema koristi od odmotavanja petlje preko 16 puta. Naznačiti ćemo s U16 u nazivu varijante programa da je neka petlja odmotana 16 puta. Npr: *GPUMTI_U16*.

Tako odmotane varijante imaju:

Radeon 7790	*GPUMTI_U16*	*67 ms*	*769/(max 940) Gflops*	*48% ubrzanja*
Radeon 5670	*GPUMTI_U8*	*601 ms*	*86/(max 310) Gflops*	*245% ubrzanja*
Phenom II	*CPUMTI_U2*	*4504 ms*	*10.49/(max 38)Gflops*	*7% ubrzanja*

7.5 Polu blocking

Odmotavanje se može napraviti na više načina. Odmotavanjem *while* petlje mi povećavamo broj *j*-tih čestica koje djeluju na *i*-tu česticu u jednom prolazu *while* petljom. Isto tako možemo povećati broj *i*-tih čestica na koje se djeluje prolazom kroz petlju. Ovo je slično memorijskom *blockingu* gdje u jednom prolazu kroz *j* elemenata obradimo više od jednog *i*-to elementa, to jest obradimo blok *i*-tih elemenata.

Sljedeća optimizacija će raditi tako da za svaku *i*-tu instancu obradimo više *i*-tih elementa, točnije 4. Dakle umjesto da obrađujemo sve *j*-te elemente za jednu *i*-tu česticu mi ćemo obrađivati sve *j*-te elemente za 4 *i*-te čestice u jednoj *i*-toj instanci što nam čini manji *i*-ti blok podataka. Ovo nazivamo polu *blocking* jer *j*-te čestice nisu stavljene u blokove kao *i*-te, te se za svaki *i*-ti blok poziva cijelo polje podataka položaja od početka do kraja preko *j* varijable.

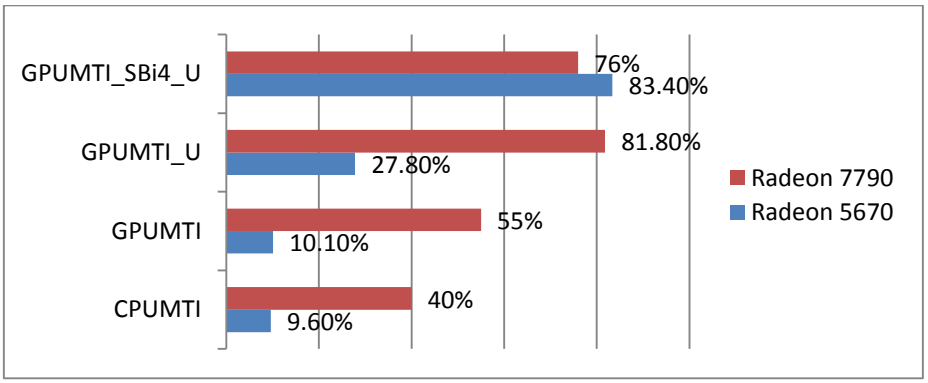
Ovu funkciju nazivamo GPUMTI_SBi4_U8. SBi4 o(eng. *semi* blocking) označava da se u jednoj *i*-toj iteraciji/instanci računaju 4 *i*-ta elementa a U8 označava da su i ti računi odmotani 8 puta. Sve skupa unutar *while* petlje imamo 8*4=32 računa sile/akceleracije.

Radeon 7790	*GPUMTI_SBi4_U8*	*72 ms*	*715/(max 940) Gflops*	*-7% ubrzanja*
Radeon 5670	*GPUMTI_SBi4_U8*	*198 ms*	*260/(max 310) Gflops*	*303% ubrzanja*

Radeon 7790 je malo sporiji ali vidimo da je Radeon 5670 imao ubrzanje od 3 puta i ovim dostigao svoj računski potencijal.

Pogledajmo na sljedećoj slici koliko dobro koja optimizacija iskorištava koji GPU:

Slika 19



Efikasnost izražena u postocima od očekivane performanse (50% teorijske) za 64K čestica

Radeon generacije 5000 je dobio najviše performanse od odmotavanja i od polu *blockinga.* Isti će efekt vrijediti i za Radeon generaciju 6000. Nije sasvim jasno zbog čega dolazi do ubrzanja kod starijih Radeona ali jasno je da ima veze s propusnošću. Ovo ćemo diskutirati još kasnije kada se dotaknemo efektivne podatkovne propusnosti našeg problema.

Do generacije 7000 Radeoni su bili građeni od tako da jedna grafička jezgra ima 16 većih jedinica (16 procesora unutar grafičkog procesora) a svaka jedinica 4 ili 5 ALU-a. Od generacije 7000 ova se raspodjela promjenila u 4 veće jedinice od kojih svaka ima 16 ALU-a. Osim toga organizacija registara generacije 7000 je takva da im je zrnatost manja to jest da ih efektivno imaju više. Rezultat toga jest da su generacije od 7000 pa nadalje fleksibilnije i u startu iskorištavaju veći računski potencijal bez većih optimizacija. Jasno je da zbog toga bilo kakve optimizacije puno više pomažu starijim generacijama. Važno je napomenuti da optimizacije ne pomažu da se prijeđe 50% teoretske performanse već se s njima samo omogućava da se ta performansa ostvari.

Pokušali smo primijeniti na grafičku karticu blokovsku verziju MTI varijante koja je radila na procesoru ali nije dala dobre rezultate jer grafička kartica radi drukčije. Jednim dijelom možda i zato što Aparapi automatski rastavlja podatke u radne grupe i već time čini nekakav *blocking*. I zaista mijenjanjem veličine radne grupe mijenjaju se performanse izvođenja programa. Međutim ako iskoristimo odmotavanje petlje i polu *blocking* radi povećavanja performansi tada nam mijenjanje veličine radne grupe ništa ne pomaže.

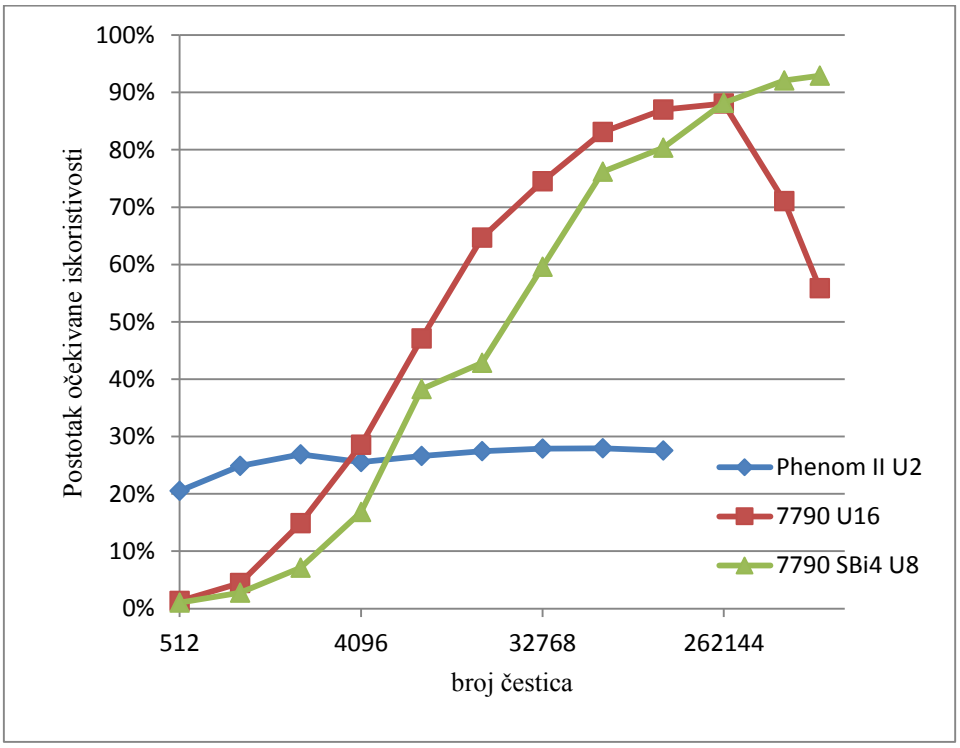
Imali smo još nekoliko vrsta implementacija koje se svode na kombiniranje postojećih varijanti programa ali ili nisu pokazale bolju performansu ili su bile sporije. Zapravo smo samo s dvije osnovne optimizacije i *rsqrt* funkcijom postigli najbolje rezultate.

Grafička kartica zapravo radi jako jednostavno. Dok kod procesora zbog ograničenog broja registara morate naći dobar balans između memorijske komunikacije, količine podataka koja se odjednom nalazi u registrima i kalkulacija na njima, na grafičkoj kartici je dovoljno samo postići memorijski *blocking* da zaobiđete eventualno usko grlo koje može predstavljati podatkovna propusnost i jednostavnim odmotavanjem petlje iskoristite masivni paralelizam grafičke jezgre te postignete pristojnu performansu. Na procesoru smo koristili odmotavanje petlje, razdvajanje petlje, spajanje petlje, memorijski *blocking* i memoriranje parcijalnih rješenja da bismo izvukli maksimum iz procesora.

GPU ima veliku propusnost, puno registara i mnogo ALU-a tako da ne moramo raditi puno finog podešavanja da bismo imali pristojnu efikasnost GPU-a, barem što se tiče problema n-tijela.

Analizirajmo kako performansa grafičke kartice i procesora ovisi o broju čestica koje se na njemu izvode.

slika 20



Broj čestica u ovisnosti o postotku ***očekivane*** *iskoristivosti (pola teorijskog potencijala grafičke kartice i procesora.*

Usredotočimo se na početku na pad i rast iskoristivosti prikazan na gornjoj slici.

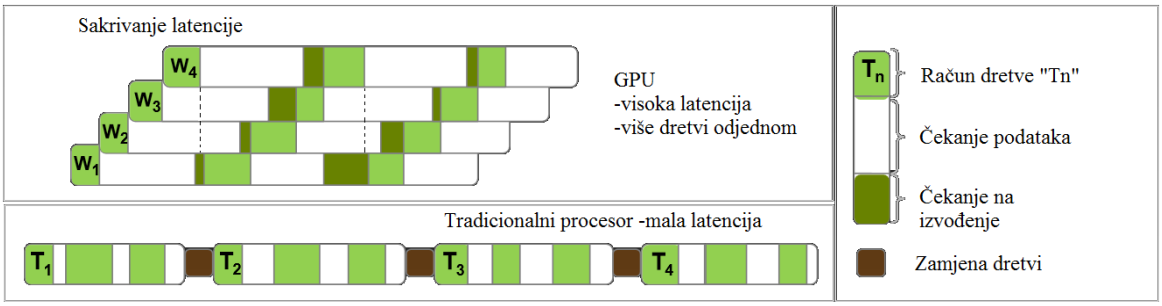
Primijetimo prvo da za brojčestica preko 256K performansa grafičke kartice pada kada koristimo *GPUMTI_U16* međutim s *GPUMTI_SBi4_U8* ona nastavlja rasti ili barem održava visoku učinkovitost. Kada se opaža ovakvo ponašanje onda je to zbog problema sa skaliranjem. Pad performanse se događa zbog ograničenja u podatkovnoj propusnosti koja je potrebna za tako veliki broj čestica i SBi4 varijanta koja jest neka vrsta memorijskog *blockinga* uklanja taj problem.

Vidimo da procesor i za mali broj čestica ima iskoristivost podjednaku maksimalnoj postignutoj. Kod grafičke kartice ona značajno varira od broja čestica. Promotrimo prvo crvenu *GPUMTI_U16* varijantu. Iskoristivost raste s brojem čestica to jest *i*-tih instanci. No taj je rast brži za broj čestica manji od 4000. Sjetimo se da ova grafička kartica ima 896 ALU-a i ako je broj instanci manji od tog broja jasno je da neće sve aritmetičko logičke jedinice biti aktivne. Čak i da imamo točno 896 instanci one ne dolaze u kontakt s ALU-ima odjednom nego prolaze kroz kanal, te se svi ALU-i zasićuju na otprilike 4000-16000 instanci. Kako broj čestica polagano raste povećava se i broj ALU-a koji se iskorištavaju pa je rast performanse brži u tom području.

Na brzinu izvođenja također utječe *overhead* pozivanja funkcije *execute.* Kada Aparapi poziva funkciju execute on automatski kopira podatke iz RAM-a procesora u RAM od grafičke kartice preko sabirnice. Kada je broj podataka mali i račun na njima relativno kratak većina vremena se potroši na latenciju između memorije procesora i grafičke kartice. Povećavanjem broja čestica povećava se omjer korisnog računa i *overheada* u korist računa čime se povećava efikasnost i to se vidi na grafu.

Mogući čimbenik je također i latencija unutar grafičke kartice koja je kod nje velika te dolazi do izražaja ukoliko se podaci ne dobavljaju u grafičke jezgre uređeno i predvidljivo. Na slici 21 vidimo kako grafički procesor paralelizacijom dretvi pokušava sakriti latenciju*.

slika 21

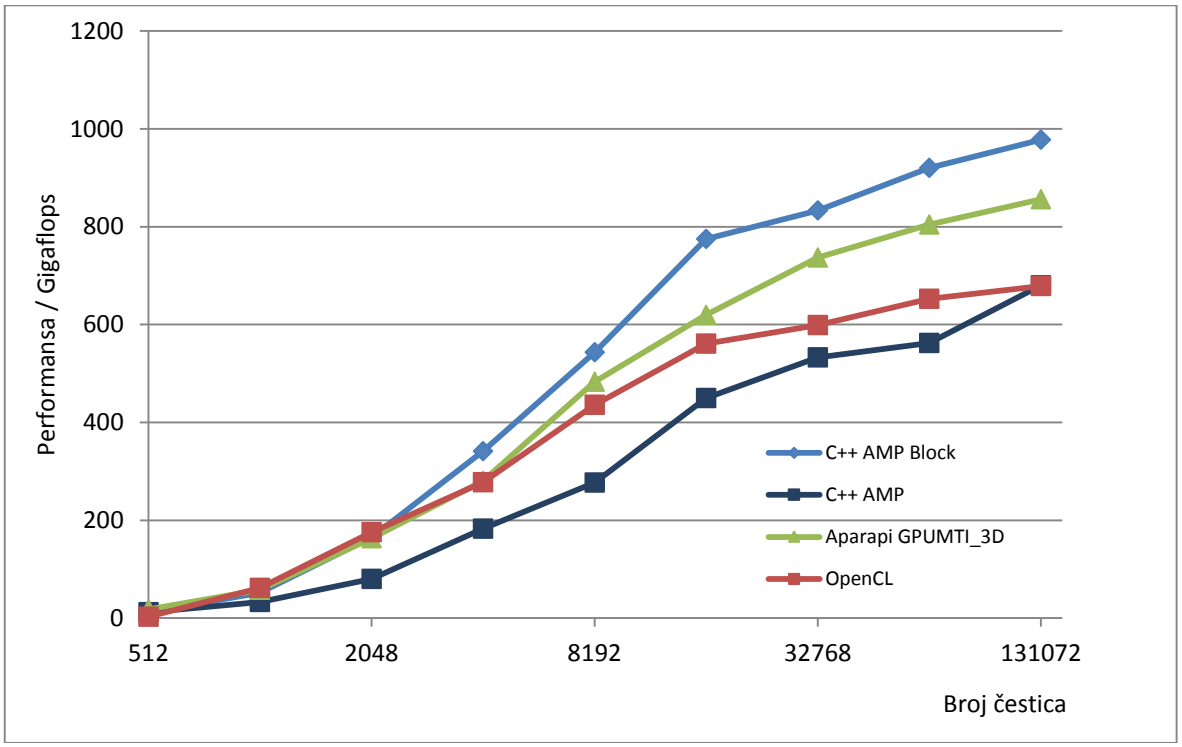


Slika prikazuje izvođenje dretvi na jednoj jezgri procesora (dolje) i više procesnih jedinica unutar grafičke kartice.
**dijelom preuzeto s https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf*

* Ako procesnu jedinicu opteretimo kalkulacijama te je ona zaposlena računanjem i nema praznog hoda dok ne dođu podaci potrebni za slijedeći račun možemo reći da smo sakrili latenciju sustava.

S nekoliko različitih implementacija smo pokušali riješiti ovaj problem međutim rješenja su ograničena te nisu dala ploda. Možemo provjeriti je li je ovaj problem svojstven samo Aparapiju? U C++ AMP-u imamo napisanu 3D verziju problema n-tijela. Ona ima 19 kalkulacija. Od optimizacija sadrži korištenje *lokalne* memorije to jest memorijski *blocking* i odmotavanje petlje. Napravili smo i mi svoju 3D verziju te je odmotali 4 puta (toliko puta se pokazalo optimalno) ali bez memorijskog *blockinga*. Osim toga imamo C++ AMP i OpenCL jednostavnu verziju bez memorijskog *blockinga*.

slika 22



3D problem n-tijela implementacije za: Java-Aparapi, OpenCL, C++ AMP s i bez memorijskog blockinga. Izvođenje se izvršava na Radeonu 7790.

Primjećujemo da Aparapi ne zaostaje mnogo za C++ implementacijom, što je impresivno jer je Java implementacija zaostajala više od dva puta za C-om. Primjećujemo isto tako da je C++AMP probio naše očekivanje od 940 Gflopsa. Stvar je u tome što neke operacije u našem problemu jesu operacije simultanog množenja i zbrajanja te ih računske jedinice obave u jednom ciklusu preko FMA* instrukcije. Isto tako iznenađuje da je Aparapi bolji od OpenCL-a, vjerojatno zato jer OpenCL verzija nema nikakav memorijski *blocking* kao ni jednostavna C++ AMP verzija. Što implicira da Aparapi koristi nekakve optimizacije koje više puta koriste jednom dostavljene podatke.

* Objašnjeno u poglavlju 7.1 jednadžba (14) stranica 57.

Razmotrimo koliki su memorijski zahtjevi. Mogli bismo crtati graf ovisnosti podatkovne propusnosti o broju čestica ali bi izgledao identično. Rezultate u gigaflopsima trebali bi pomnožiti s 0.63 . Naime za svaki prolazak kroz *while j* petlju u 3D modelu imamo 19 operacija s pomičnim zarezom i imamo 3 čitanja iz memorije, svako od 4 Bajta. Čime dobivamo da je omjer Byte/Flop = 3*4/19= 0.63 . Za 2D problem ovaj koeficijent je 0.66 . Ovime u sljedećoj tablici dobivamo efektivnu propusnost svake varijante za 131072 čestica.

Tabela 5

	C++AMP Block	C++AMP	Aparapi	OpenCL
Propusnost	617 GB/s	429 GB/s	540 GB/s	429 GB/s

Brzina prijenosa podataka za svaku od platformi kod 3D problema n-tijela za 131072 čestica.

Nemamo detaljnije tehničke podatke za Radeon 7790 ali imamo za Radeon 7770 i Radeon 7850 koji redom imaju 10 i 16 grafičkih jezgara. Propusnost predmemorije nivoa 2 je za Radeone redom 256 i 440 GB/s. On je proporcionalan s brojem grafičkih jezgara. To znači da ćemo za Radeon 7790 očekivati najmanje oko 360 GB/s. Znamo da je Radeon 7790 malo napredniji u arhitekturi jer se bazira na GCN 1.1, za razliku od prethodna dva koja se baziraju na GCN 1.0. Također znamo da mu je teorijska propusnost RAM memorije 96 GB/s naspram 72 GB/s od Radeona 7770. To je 33% više. Stoga možemo pretpostaviti da je i brzina predmemorije veća. Ovo je gruba procjena ali možemo pretpostaviti da njegova propusnost nije veća od Radeona 7850 s više grafičkih jezgara i da je limit našeg Radeona negdje na 430 GB/s kako su i pokazale varijante AMP-a i OpenCL-a bez bilo kakvog memorijskog *blockinga*. Znamo da C++AMP Block verzija koristi lokalnu memoriju, pa joj je efektivna propusnost veća od 430 GB/s. Ima li onda i Aparapi ima posebne optimizacije ako postiže propusnost od 530 GB/s ?

Za 131072 čestice 3D modela trebamo 3 polja položaja a za svaki podatak u njima trebamo 4 bajta. To znači da nam je potrebno 131072*3*4≈1.5 MB podataka za račun jedne sličice. Predmemorija nivoa 2 Radeona 7790 je tek jednu trećinu od toga. To znači da se podaci prelijevaju u RAM i da se za svaku instancu ti podaci nužno moraju pozivati iz RAM-a grafičke kartice i da trebaju biti ograničeni brzinom RAM-a koji je 96 GB/s. To bi davalo otprilike 5 puta manju performansu nego očekivanu. I ovaj efekt ograničenosti brzinom RAM-a smo opazili kod Radeona 5670. Međutim znamo da su podaci koji su potrebni jednoj dretvi otprilike u isto vrijeme potrebni i drugim dretvama. Naprimjer podaci položaja iz sredine *xx* i *yy* polja će biti potrebni otprilike u isto vrijeme svim dretvama koje se izvršavaju u danom trenutku. Tako da nije potrebno da se za svaku dretvu cijelo polje *xx* i *yy* podataka kopira direktno iz RAM-a grafičke kartice, nego se jedan veći segment tih polja kopira u

predmemoriju nivoa 2 i onda se iz njega one šalju u predmemoriju nivoa 1, te se tamo još dodatno dijele među dretvama jedne jezgre. Aparapi navodno ima posebne algoritme za optimizaciju podataka u predmemoriji.

Kod Radeona 5670 smo za verzije programa koje nemaju *blocking* opazili ograničenje na nekih 87 Gflopsa. To odgovara propusnosti od 57.42 GB/s što je tek malo ispod njegove deklarirane brzine RAM-a od 64 GB/s i to za količinu podataka koja stane u predmemoriju nivoa 2. Za predmemoriju nivoa 2 znamo da je dovoljno brza da nadvlada ovo usko grlo u podatkovnoj propusnosti.

Ovi rezultati su u najmanju ruku zbunjujući. Mjerena propusnost na Radeonu 5670 odgovara teorijskoj, ali se ono se događa za mnogo manji broj čestica - puno prije nego bismo to očekivali. Možda je manja zrnatost registara ili starija arhitektura odnosno organizacija ALU-a odgovorna za ovakvo ponašanje. Ovo primijetili i za Radeon 6770 koji ima sličnu arhitekturu kao i Radeon 5670 i koji je dvostruko je jači od njega. Teško je izvući zaključke osim kvalitativnih. A to su:

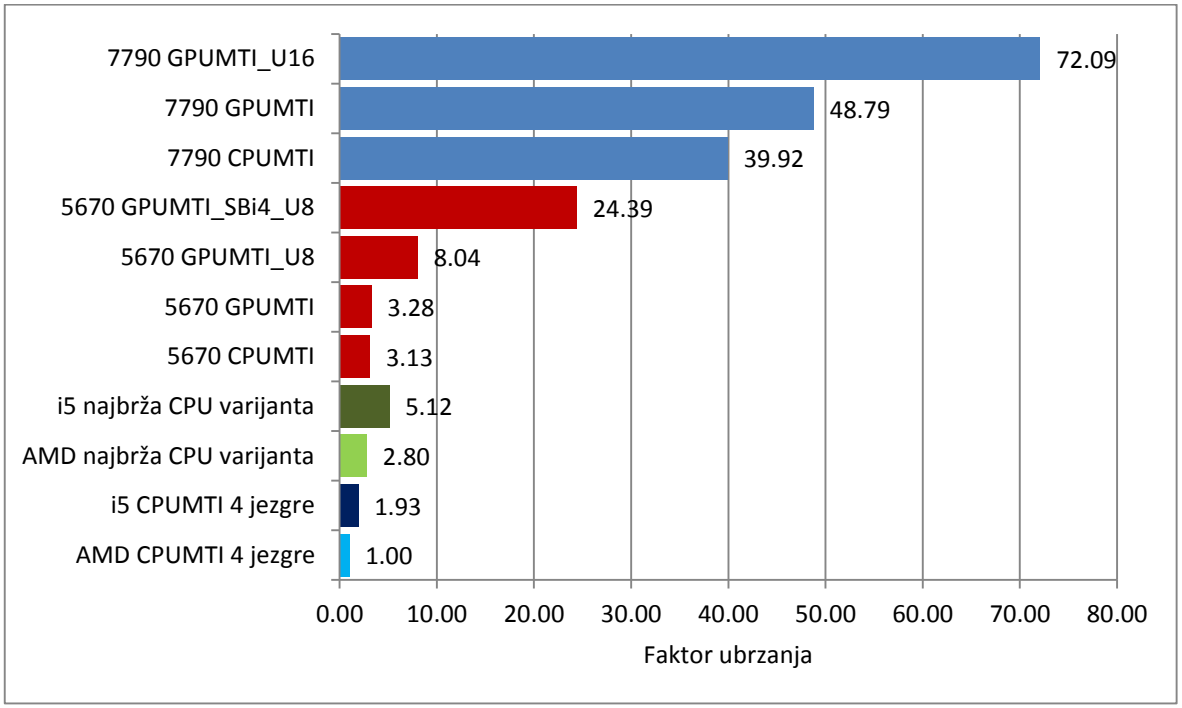
Radeon 7000 generacija je značajno naprednija u memorijskoj komunikaciji naspram generacije 5000 i 6000 koje pokazuju slične osobine. Primitivna verzija Aparapi varijante pokazuje veću propusnost podataka nego primitivne verzije u C++AMP-u i OpenCL-u. Ovo je vjerojatno zbog optimizacija u korištenju predmemorije koje su ugrađene u Aparapi.

Osim toga bismo mogli zaključiti:

Ukoliko želimo efikasno koristiti grafičku karticu, osim nekih osnovnih metoda optimizacije (*blockinga* i odmotavanja petlji) poželjno je da nam problem ima veliki omjer kalkulacija po prenesenom podatku. U našem standardnom modelu to je bilo 11-12 kalkulacija dok je u 3D modelu taj broj 19 kalkulacija. Povećavanjem broja kalkulacija po podatku sakrivamo latenciju sustava, tj. opterećujemo sustav kalkulacijama dok se ne dobave novi podaci. Također je poželjno da se ti podaci ponovno koriste to jest da je težina problema $O(n^2)$ kao kod problema n-tijela ili $O(n^3)$ kao kod množenja matrica. Ukoliko nam je prethodni uvjet zadovoljen poželjno nam je da na grafičku karticu šaljemo veće količine podataka za obradu jer se tada zasićuju stotine ALU-a i umanjuje *overhead* pozivanja *execute*() funkcije, dakle efikasnost grafičke kartice je veća.

Ovime smo više manje završili sve hardverske optimizacije. Sumirajmo ih na sljedećem grafu.

slika 23



Performanse su normalizirane na performansu BasicNbodyMTI ili CPUMTI varijante na 4 jezgre Phenom II procesora na 3.2 GHz

Najbolja optimizacija koja radi na Radeon 7790 kartici ima performansu ekvivalentnu najboljoj procesorskoj optimizaciji koja radi na 80 jezgara Ivy bridge generacije (ista generacija kao i i5 3570) ali na 2.4 Ghz. To je ekvivalentno osam E5-4650-v2 procesora od kojih svaki ima 10 jezgara i radi na 2.4 Ghz. U vrijeme kad je izišao na tržište u prvom kvartalu 2014 godine cijena mu je bila 3616.00 $. Radeon 7790 je 5-6 mjeseci prije toga koštao 110 dolara. Ukupna cijena tih procesora performanse ekvivalentne našoj grafičkoj kartici je 28928.00 $ i ona je 263 puta skuplja nego ta ista grafička kartica. Ako uzmemo najlošije varijante programa ovaj odnos je 386! Ove brojke rastu oko 40% u korist grafičke kartice u 3D slučaju ili u slučaju neparnih potencijala kada korjenovanje dolazi do izražaja. Treba uzeti u obzir da je cijena jedne jezgre procesora nekoliko puta skuplja ako slažemo superračunalo. Zbog toga su razlike u cjenovnoj efikasnosti 2 reda veličine. Ali ako uzmemo samo performanse naših i5 i Phenom II procesora s obzirom na trenutnu cijenu u Hrvatskoj dobivamo da je Radeon 7790 cjenovno efikasniji 24-35 puta od i5 3570 procesora te 12-20 puta od Phenom II x4 3.2 Ghz procesora ovisno koristimo li osnovne ili najbolje verzije programa. Odnos cijena varira u Hrvatskoj. Također treba uzeti da je cijena cijelog sustava (napajanje, kućište, matična,RAM itd..) smanjuje ovaj odnos. Napomenimo da u jednu matičnu ploču može stati od 1 do 14 GPU-a. Ove brojke nisu reprezentativne za druge računske probleme poput naprimjer brzog Fourierovog transformata ali sasvim je razumno

očekivati barem red veličine veću efikasnost po cijeni uređaja kod grafičkih kartica nego tradicionalnih procesora.

7.6 Multi GPU

Kako smo u posjedu 2 grafičke kartice i kako u našu matičnu ploču stanu dvije grafičke kartice možemo pokušati pokrenuti program na obije kartice. Generalni pristup je ovaj: Podjelimo problem tako da svaka kartica obrađuje broj čestica proporcionalan njezinoj performansi. Nakon što svaka izračuna svoj dio posla grafičke kartice moraju razmijeniti izračunate podatke da bi svaka imala potpuno rješenje a ne samo svoj dio. Ovaj dio kopiranja Aparapi radi automatski. I iako smo napravili dva objekta: *svekuglice2 i svekluglice3* svaki sa zasebnim skupom čestica s kojima radi Aparapi ih tretira kao jedan objekt i međusobno kopira podatke iz jednog u drugog. Ovo nam sada paše ali u nekom drugom problemu ovo je nepoželjno ponašanje. Također trebamo odrediti koji dio kopije čestica se renderira, onaj koji stoji ili onaj koji se pomaknuo. Aparapi nekako sam "shvati" što se treba prikazati u prozoru bez da mu mi ovo direktno kažemo. Ovdje nam nije sasvim jasno što se događa, jer postoji automatizam kojeg nismo svjesni, ali po vizualnom rezultatu koji dobivamo znamo da je račun točan ili dovoljno dobar jer je identičan rezultatu s jednom grafičkom karticom.

Grafičke kartice su prilično asimetrične, po performansi su različite za faktor 3 do 4 a između njih je također 2 generacije razlike u tehnologiji. Zbog ovog smo posao podijelili tako da će Radeon 5670 izvoditi 9/32 kalkulacija s *GPUMTI_SBi4_U8* varijantom programa a Radeon 7790 23/32 kalkulacija s *GPUMTI_U16* varijantom programa. Svaka grafička kartica radi s verzijom koja joj najbolje paše, a omjer 9/32 smo pronašli testiranjem i on je proporcionalan omjeru performanse grafičkih kartica. Program se izvodi za 128K čestica. Radeon 7790 na ovom omjeru i broju čestica program izvede za 192 ms dok Radeon 5670 to napravi za oko 200 ms. Rezultat toga jest da ćemo imati jedan dio čestica koji će brzati ispred drugog te računi neće biti sinkronizirani. Da bismo ovo sinkronizirali napravili smo zasebnu dretvu koja pokreće izvođenje na jednoj od kartica. Ona isto tako inkrementira brojač u nekom objektu kojeg vide obije grafičke kartice svaka u svojoj dretvi i kojeg smo stvorili samo za tu svrhu. Kada glavna dretva detektira da je brojač povećan ona nastavlja izvođenje programa i printa sličicu. Ukoliko brojač nije povećan, to znači da se račun na sporijoj kartici još nije završio te ga glavna dretva čeka. Na ovaj način sinkroniziramo obije dretve tako da ne brzaju jedna ispred druge. Za 128K čestica Radeonu 7790 je potrebno 252ms u standardnoj izvedbi. U dualnom modu s Radeonom 5670 im je potrebno 200ms.

Ovo je ubrzanje od 26%. Očekivano teorijsko ubrzanje računanja za taj broj čestica prema podacima za svaku karticu zasebno je 30%. Možda smo mogli postići bolju performansu s malo boljim omjerom ali i ovo je unutar očekivanja. Valja napomenuti da smo s Radeonom 6770 za koji smo isto vršili mjerenja u kombinaciji s Radeonom 7790 postigli oko 60% ubrzanja jer je njihov omjer snage oko 3 naprama 5.

Problemi koji se javljaju s korištenjem više komada asimetričnog hardvera su ti da često ne dobivamo ubrzanje nego dobijemo neku performansu između performanse jačeg i slabijeg hardvera umjesto zbroja tih performansi. C++ AMP implementacija ima isto tako opciju za izvođenje na više grafičkih kartica ali performansa je tek oko 300 Gflops, što je puno manje od performanse Radeon 7790 čiji je rezultat u C++AMP-u bio blizu teraflopsa. Dualni postav grafičkih kartica dalje nećemo koristiti jer komplicira stvar prilikom izračuna. Ali ovim smo samo pokazali da je to moguće.

Ovdje je važno naglasiti ovu komunikaciju između grafičkih kartica. Nakon što svaka od njih napravi račun one moraju izmijeniti podatke i to čine tako da ih prvo šalju na procesor a tek onda jedna drugoj. Ovo može biti usko grlo ukoliko je količina podataka prevelika i ukoliko više GPU-a međusobno komunicira. No u našem slučaju veličina naših polja koja se šalju ne prelazi 2 MB i ovo se prenese PCIexpress sabirnicom za manje od jedne milisekunde. Kod većih količina podataka imali bismo problem s latencijom i propusnošću sabirnice, veličinom RAM-a grafičke kartice te bi dolazilo do problema sa skaliranjem. Imamo sreće što je problem n-tijela računski a ne memorijski intenzivan.

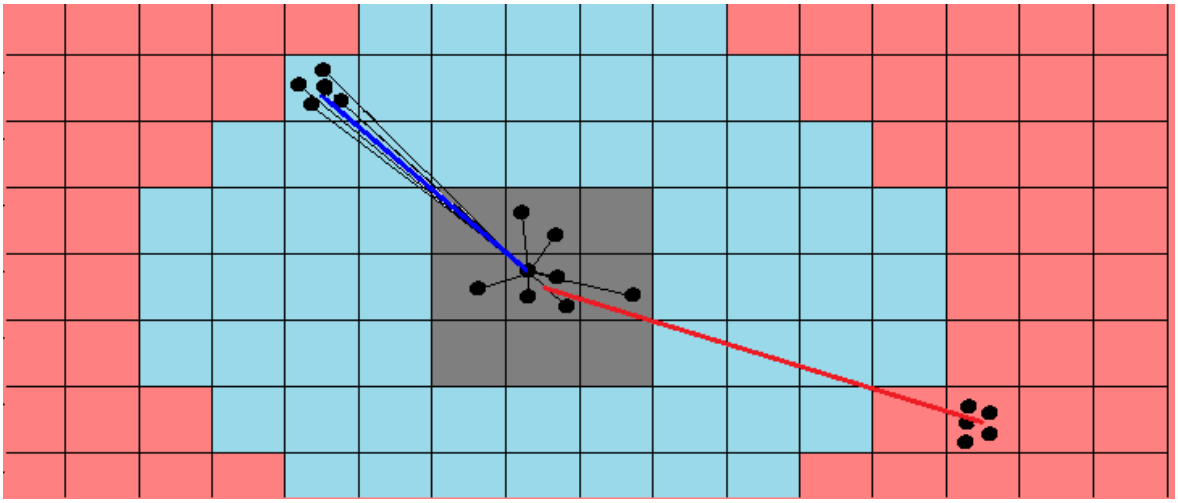
8 Programske optimizacije

Doveli smo hardver do maksimuma i istražili uvjete u kojima se to postiže. Također smo se upoznali s generalnim radom hardvera. Ako želimo još bolje performanse moramo implementirati programske tj. softverske optimizacije.

8.1 Sustav ćelija

Zamislimo da smo 2D prostor u kojem se nalaze čestice podijelili u *ćelije*. Naše standardno djelovanje je djelovanje svake čestice na svaku drugu česticu i zahtjeva n^2 kalkulacija. Međutim je li je potrebno svaki put računati izraz $\frac{1}{r^2}$ ukoliko se čestice za koje se računa izraz nalaze jedna blizu druge. Promotrimo sljedeću sliku:

slika 24



Crno je djelovanje čestice na česticu, tamnoplavo je djelovanje čestice na ćeliju, crveno je djelovanje ćelije na ćeliju.

U tom slučaju dovoljno je izračunati samo jedan radijus: od *i*-te čestice za koju radimo račun, do centra mase *j*-tih čestica koje se nalaze u ćeliji. Na slici 24 je to naznačeno kao plavo. Isto tako ukoliko se grupe čestica nalaze jako daleko ili ako su ćelije u kojima se čestice nalaze male možemo računati radijus između pojedinih ćelija to jest centara masa u njima (naznačeno crveno).

Definirajmo *prvo područje* u kojem se računa čestica - čestica međudjelovanje (sivo) i nazovimo je PP (*particle-particle*) međudjelovanje. Za svaku *i*-tu česticu to će biti područje od barem jedne ćelije unutar koje se ta čestica nalazi ili nekoliko ćelija oko te ćelije. Radi

preciznosti pogodno je uzeti oko 9 ćelija za prvo područje. Broj ćelija koje se uzimaju biti će određen *prvim radijusom* unutar kojeg padaju centri tih ćelija.
Drugo područje će biti područje u kojem djeluje ćelija na česticu (svjetlo plavo) i nazovimo je TP međudjelovanje (*tile-particle*). Ovo je područje koje se nalazi unutar *drugog radijusa.*
Treće područje će biti područje izvan *drugog radijusa* (rozo) i u njemu će se vršiti TT međudjelovanje (*tile-tile*).

8.1.1 GPUTiled1

Napravili smo nekoliko implementacija ove programske optimizacije. Prvih nekoliko su računale samo s 2 područja (sivim i plavim). U takvoj varijanti svaka od *n* čestica je trebala imati:

-PP račun
-TP račun
-pomak/integrator
-sudaranje sa zidom

Ovaj problem zahtjeva i sudaranje s zidom da bi čestice ostale u zadanom virtualnom prostoru. Kasnije ćemo diskutirati neka od rješenja sudaranja sa zidom.
PP i TP se treba računati *za svaku česticu*. Kôd koji je unutar *while j* petlje MTI funkcije je zapravo bilo PP (*particle - particle*) međudjelovanje i prolazilo je *n* puta kroz *while j* petlju za svaku *i*-tu česticu. Sada je zamjenjuju PP i TP međudjelovanja gdje PP djeluje samo na čestice unutar nekoliko ćelija sive zone i zato je ovaj račun mnogo kraći. Nazovimo novu funkciju *GPUTiled1*. Prije nego pogledamo kôd uzmimo u obzir da smo čestice poredali u memoriji tako da one čestice koje se nalaze u istoj ćeliji jesu isto tako jedna do druge u istom memorijskom prostoru. Isto tako uzmimo u obzir da umjesto centara masa u pojedinoj ćeliji koristimo centre ćelija! Za dovoljno male ćelije i jednoliku gustoću plina ovo nam je dovoljno dobra aproksimacija zbog koje ćemo imati prednost u brzini računa. Također uzmimo u obzir da je kôd na sljedećoj stranici malo pojednostavljen da bi bio razumljiviji.
searchTile() funkcija traži ćeliju unutar koje se nalazi *i*-ta čestica i označava je s *ki*. Nakon toga pokrećemo *while k* petlju koja prolazi kroz cijeli prostor ćelija i testira koje se ćelije nalaze unutar prve zone, tako da računamo udaljenost centra *k*-te ćelije do centra *ki*-te ćelije te gledamo je li je ona manja od *firstRadius* konstante koja nam određuje granicu prve zone. Kada smo ustvrdili da se *k*-ta ćelija nalazi u prvoj/sivoj zoni izračunavamo PP (*particle-particle*) međudjelovanje za *i*-tu česticu, ali ne prolazimo kroz cijelu *while j* petlju nego idemo

samo od *min* do *max* elemenata, to jest samo za elemente koji se nalaze u *ki*-toj ćeliji. PP kao argument stoga uzima *PP*(*min,max,i*).

```
public void GPUTiled1(){
        //konstante
        i=getGlobalId();
        ki=searchTile(i);               //ki= broj ćelije u kojoj se nalazi i-ta čestica
        min=0;                          // najmanji j-ti element od kojeg polazimo
        k=0;
        while(k<numberOfTiles){                 //iteriramo kroz k ćelija
                //---------------------------------
                max=FindMaximumJ(ki);           //0 flop/  najveći j-ti element
                rad=CalculateRadius(ki,k);      //5 flop/  udaljenost k-te i ki-te ćelije
                if(rad<firstRadius){
                        PP(min, max, i);        //PP, moguće odmotavanje
                }else{
                        TP(min,max,i, k );      //TP nema petlju pa se nemože odmotati
                }
                min=max;
                k++;
                //---------------------------------   moguće odmotavanje ovog dijela
        }
}
```

(11)

Ako se *k*-ta ćelija ne nalazi u sivoj već plavoj zoni računamo TP (*tile-particle*) međudjelovanje. Pri tome se računa udaljenosti između položaja *i*-te čestice (*xx*[*i*],*yy*[*i*]) i položaja centra *k*-te ćelije (*tileCenterX*[*k*],*tileCenterY*[*k*]). Na osnovu ove udaljenosti izračunamo $\frac{1}{r^2}$ i to množimo s brojem čestica koje se nalaze u toj ćeliji.

PP je moguće odmotati više puta jer je u njemu *while j* petlja od *min* i do *max*. Međutim zbog toga što razlika među njima nije uvijek ista dobivamo veći broj kalkulacija nego što bi trebali. Naprimjer, recimo da je broj čestica u ćeliji 86 i recimo da smo petlju odmotali 4 puta. Petlja će se izvršiti za 88 elemenata to jest za 22 prolaska kroz petlju. Ovime ćemo imati dvije dodatne kalkulacije koje inače ne bi trebale ulaziti u račun. Kako je to malo s obzirom na broj čestica u ćeliji ovo ne predstavlja veliki problem. Na CPU ovakvo odmotavanje PP petlje neće raditi jer program javlja grešku.

TP ne možemo odmotati jer on ima samo jednu kalkulaciju (*k*-ta ćelija - čestica) za dani *k*. Međutim možemo odmotati *while k* petlju u kojoj se nalaze PP i TP, te to možemo napraviti svaki put jer ona ima točno *k* elemenata. Trebamo paziti da je broj odmotavanja višekratnik broja ćelija. Trebamo paziti s odmotavanjem *while k* petlje s PP i TP izrazima, jer koji put ne dobivamo ubrzanje izvođenja nego to dodatno usporava izvođenje programa.

Procjenimo uštedu u broju kalkulacija. Neka k bude broj ukupnih ćelija a k_1 broj ćelija u prvoj/sivoj zoni. k_2 neka bude broj ćelija u plavoj zoni i on će onda biti $k_2 = (k - k_1)$. Račun zahtjeva k prolazaka kroz petlju u kojoj imamo k CalculateRadius(ki,k) računanja udaljenosti među ćelijama. Svako računanje zahtjeva 5 računskih operacija. Također zahtjeva k_1 puta PP račun te k_2 puta TP račun. PP račun ima 12 kalkulacija a TP 15. Pretpostavimo da je gustoća čestica jednolika posvuda. Tada je prosječan broj čestica unutar jedne ćelije:

$$n_k = \frac{n}{k} \tag{15}$$

Performansu programskih optimizacija ćemo uspoređivati s najboljim vremenima dobivenim na grafičkoj kartici, to jest s *GPUMTI_U16* varijantom do 128K čestica i *GPUMTI_SBi4_U8* varijantom za više od 128K čestica. Druga varijanta omogućava nastavak kvadratnog skaliranja s povećanjem broja čestica. Ove obje varijante su bez ikakvih programskih optimizacija te koriste samo snagu hardvera da bi postigle rezultat. Na njih ćemo se referirati kao *brute force* metodu.

Broj kalkulacija *brute force* metode potreban za jednu sličicu (jedan vremenski korak) je:

$$N_{brute} = 12n^2 \tag{16}$$

Za *GPUTiled1* metodu on će iznositi:

$$N_{Tiled1} = n(5k + 12k_1 n_k + 15k_2) \tag{17}$$

Smanjenje broja kalkulacija *U* za ovaj slučaj će iznositi:

$$\mathrm{U} = \frac{N_{brute}}{N_{Tiled1}} = \frac{12n}{(5k + 12k_1 n_k + 15k_2)} \tag{18}$$

Za *n*=262144 čestica, *k*=2500 i k_1=9 ćelija smanjili smo broj kalkulacija za *U*=51 puta. Izmjereno ubrzanje je 19 puta, ali uzevši u obzir vrijeme sortiranja na procesoru ukupno ubrzanje je 17 puta.

8.1.2. CPU sortiranje

U ovoj vrsti računa važno je urediti čestice po ćelijama. Da bismo mi mogli u PP funkciji za danu ćeliju računati od *min* do *max* moramo te podatke posložiti tako da u RAM-u stoje u polju položaja u jednom komadu memorije između *min* i *max* indexa. Ukoliko ovo ne napravimo svaki put kada se poziva PP funkcija moramo proći kroz cijelo polje položaja testirajući je li se pojedina čestica nalazi u danoj ćeliji i tako našli podatke s kojima ćemo računati. Očekujemo da programske optimizacije *GPUTiled* funkcije rade barem red veličine brže od*brute force* metode i memorijski zahtjev bi bio red veličine veći ako ćemo za svaki PP poziv provlačiti cijelo polje položaja kroz jednu grafičku jezgru. Iako bi se to moglo riješiti s memorijskim *blockingom* i dalje nam ostaje problem testiranja svake čestice. Računski je manje intenzivno samo sortirati česticee u RAM-u onako kako stoje u virtualnom prostoru ćelija i onda za danu ćeliju uzimati samo elemente koji se nalaze u njoj. Ne samo da se time postiže lokalnost podataka nego se ne treba izvršavati veliki broj *if* testiranja.

Samo sortiranje je manje računski intenzivno nego ostatak računa pa se može izvesti na procesoru. Radimo ga na procesoru iz dva razloga. Prvi je teškoća pisanja kôda za sortiranje na grafičkoj kartici zbog načina na koji Aparapi raspodjeljuje dretve po grafičkim jezgrama. Drugi je da istražimo kako grafička kartica i procesor surađuju kada zajedno rade na istom problemu rješavajući svaki svoj segment.

Dio programa koji se izvršava na procesoru nećemo analizirati u detalje jer je poprilično kompleksan ali ćemo okvirno proći šta on radi.

Nakon što grafička kartica završi svoj dio posla Aparapi automatski vraća 4 polja brzina i položaja u objekt *sorter* u RAM od procesora. Nakon inicijalizacije pokreće se funkcija koja broji koliko čestica ima u pojedinoj ćeliji i uz to svakoj čestici pridružuje "adresu" ćelije u kojoj se čestica nalazi. Ti se elementi trebaju prebaciti u novo polje poredani po tim novim "adresama" i poslati natrag na grafičku karticu. Ovo možemo riješiti na nekoliko načina.

Jedan je da izračunamo koliko elemenata imamo u svakoj ćeliji i onda te podatke koristimo kao indekse na kojima počinju i završavaju granice pojedinih ćelija u polju položaja i brzina u koje podatke ponovo sortiramo. Potrebna su nam pomoćna polja u koja ćemo privremeno sortirati i prebaciti podatke prije nego ih sortirane vratimo u polja položaja i brzina.

Glavna ideja je da svako pomoćno polje predstavlja jednu ćeliju i ukoliko se čestica nalazi u virtualnom prostoru te ćelije kopiramo je u pomoćno polje. Prva verzija s pomoćnim poljima je stvarala ogromna pomoćna polja jer se teoretski sve čestice mogu nalaziti samo u nekoliko ćelija ako je plin koncentriran u sredini virtualnog prostora, te je tako koristila jako puno

RAM memorije. Ovo nije bilo prikladno rješenje. Stvar je u tome da mi moramo alocirati pomoćna polja prije nego počnemo pobrojavati broj čestica u svakoj ćeliji, a ne znamo kolika trebaju biti ta pomoćna polja bez da smo izbrojali čestice u njima. Problem je kada alociramo tip podatka polje da je njegova veličina poslije toga nepromjenjiva. Idealno bi bilo dinamički alocirati veličinu pomoćnih polja. Međutim ako mu želimo proširiti veličinu za jedan element moramo kopirati i ponovno kreirati to polje u memoriji s jednim elementom više. A staru verziju brisati. Ovo zahtjeva mnogo operacija. Problem smo riješili tako da smo koristili poludinamičko alociranje memorije. Zašto poludinamičko? U Javi je moguće u pokretanju programa stvoriti 2D polje s *k* redaka od kojih je svaki redak prazan i onda u toku izvođenja programa alocirati veličinu retka kao zasebnog polja. *Svaki redak odgovara jednoj ćeliji*. Ovisno o tome koliko smo čestica pobrojali u svakoj ćeliji toliko mjesta to jest lokacija stvorimo u pojedinom retku 2D polja. Znači nakon što smo definirali 2D polje, pobrojili čestice u svakoj ćeliji i stvorili odgovarajući broj elemenata za svaku ćeliju u recima tog polja dobijemo polje koji nema isti broj stupaca, ovo se zove *jagged array* (hrv. nazubljeno polje). Na ovaj način dobivamo točan broj mjesta u svakoj ćeliji za potrebne čestice i ne bacamo memorijski prostor.

Sljedeća funkcija sortira čestice u pomoćna polja tako da ih stavi u redak koji predstavlja njihovu ćeliju. Adrese njihovih ćelija su određene u funkciji koja ih je pobrojavala.

Nakon što su one sortirane po ćelijama, u sljedećoj funkciji se lako prepisuju u 1D polja koja se šalju natrag na GPU. Zbog toga što se svako 2D polje može prikazati i zapisati kao 1D polje čestice je moguće odmah sortirati u polje koji se šalje na grafičku karticu bez potrebe za pomoćnim poljima. U trenutku kada smo rješavali ovaj problem rješenje s 2D poljem se činilo preglednije zbog načina adresiranja.

Druga verzija sortiranja s poludinamičkim alociranjem memorije je imala još jednu optimizaciju. Naime kada tražimo u kojoj se ćeliji nalazi čestica pretraživali smo cijeli virtualni prostor ćelija i gledali je li koordinata čestice unutar zadanih prostornih granica neke ćelije. Ako imamo 2500 ćelija moramo napraviti 2500 testiranja za svaku česticu. Ovo je nepotrebno dugo. Kao poboljšanje ovog algoritma smo koristili činjenicu da nam sama koordinata čestice odaje njezin položaj. Recimo da je veličina ćelije 20 točkica i da je položaj čestice u x smjeru 856. Podijelimo 856 s 20 te dobijemo 42.8 . Ovaj broj zaokružimo na manje te dobijemo broj 42 što znači da se čestica nalazi u 42. ćeliji. Cijeli program s poludinamičkim alociranjem čestica je radio 200 puta brže na jednoj jezgri nego prva verzija sortiranja.

Sortiranje smo paralelizirali za 2 i za 4 jezgre. Ubrzanje od paralelizacije na 2 jezgre je 1.9 puta a ubrzanje od paralelizacije na 4 jezgre je 2.2 puta. Sveukupni program sortiranja sadrži 6 do 7 funkcija. Iako je svaka od tih funkcija paralelizirana, samo ona koja dijeli položaj čestice s veličinom ćelije ima koristi od paralelizacije jer samo ona zapravo nešto računa. Većina ostalih operacija su memorijske operacije kopiranja. Na grafičku karticu se šalju sortirani podaci polja s *min* i *max* podacima koji određuju granice ćelija u *xx*, *yy*, *velxx* i *velyy* poljima .

Iako se čini da je grafička kartica bolja solucija za sortiranje zbog veće podatkovne propusnosti i zbog načina na koji Aparapi raspodjeljuje dretve po jezgrama grafičke kartice, paralelizacija sortiranja na grafičkoj kartici unutar Aparapija je iznimno teška i komplicirana.

Java/Aparapi omogućuje da procesor zapravo jako lako komunicira s grafičkom karticom. Relativno smo lako napisali kôd koji se zapravo izvodi u Javi (JTP modu) na procesoru i svoja rješenja šalje na grafički procesor koji prevodi svoj Java kôd u OpenCL. Vidjeli smo da procesor ne samo da može biti jako koristan grafičkoj kartici nego je za kôd koji se ne može paralelizirati ili se teško paralelizira procesor jedina solucija.

Snaga jedne jezgre procesora je još uvijek i uvijek će biti bitna bez obzira na paralelni potencijal grafičkih kartica, ali grafičke kartice zasigurno imaju budućnost u programiranju opće namjene zbog 2 reda veličine većeg računskog potencijala.

Objedinjenje prednosti i jednog i drugog tipa hardvera su APU-i (Accelerated Processing Unit) kako ih zove AMD ili SoC (Systen on a Chip) kako ih zove Intel koji objedinjuju i tradicionalni procesor i grafički čip na jednom te istom komadu silicija. Oni čine 95% današnje prodaje procesora.

8.1.3 GPUTiled2

Sada kada malo bolje razumijemo osnovu naše programske optimizacije možemo prijeći na njezinu bržu varijantu. Brža varijanta uključuje kalkulaciju u trećem rozom prostoru. Plavi i rozi prostor razdvaja kružnica drugog radijusa. To znači da u plavom prostoru sada imamo manje ćelija nego prije, te moramo odrediti koje od svih ćelija ulaze u plavi prostor. Razmotrimo *GPUTiled2n* varijantu gdje "n" označava broj instanci u kojima je ta funkcija stvorena:

```
public void GPUTiled2n(){
        //konstante
        i=getGlobalId();
        ki=searchTile(i);
        min=0;
        k=0;
        while(k<numberOfTiles){
                max=tilebreaksacumulated[k+1];                  //FindMaximumJ
                deltx=t2x-tilecenterx2[k];                      //CalculateRadius
                delty=t2y-tilecentery2[k];                      //
                currentCellRadSQ=deltx*deltx+delty*delty;       //
        //novo  if(rad<secondRadius){                   //testiranje za plave elemente
                        if(rad<firstRadius){
                                PP(min, max, i, t1x, t1y);      //račun u sivom području
                        }else{
                                TP(min,max,i, t1x,t1y,k );      //račun u plavom području
                        }
                        min=max;
                        k++;
                }
                                        //ovdje bi trebao biti račun u rozom području
        }
}
```
(28)

Ovdje smo zamijenili imena funkcija FindMaximumJ i CalculateRadius i zapisali ih onako kako one stvarno rade i upotpunili smo prave argumente funkcija PP i TP.

GPUTiled2 varijanta ima samo jednu novu liniju kôda:

```
if(rad<secondRadius)
```

Ona testira je li se dana ćelija nalazi unutar drugog radijusa, te izostavlja račun za ćelije koje se ne nalaze u tom području. U pravom kôdu u *if* funkciji testiraju se kvadrati udaljenosti što se svede na isto ali nije potrebno dodatno korjenovanje. Ovdje je samo radi čitljivosti kôd pojedostavljen. Za razliku od *GPUTiled1* verzije, *GPUTiled2* ima mnogo manje plavo područje pa nije potrebno da za svaku *k*-tu ćeliju računa PP ili TP. Ali ipak za svaku *k*-tu ćeliju ona izračuna radijus između nje i *ki*-te ćelije te testira je li ta ćelija spada u plavo područje. Ovo zahtjeva barem 5 računskih operacija i za veliki broj ćelija ovo ima značajan efekt.

Intuitivno nam je da odmah u istoj funkciji primijenimo kalkulaciju TT (ćelija-ćelija) na ćelije koje ne zadovoljavaju prethodni uvjet, to jest koje se nalaze u trećem rozom području. Postoji jedan problem s ovim. *GPUTiled2n* funkcija se poziva *n* puta to jest formira *n* instanci. Za svaku česticu izvodi PP i TP račun. TT račun zahtjeva međudjelovanje svake *k* ćelije s malo manje od *k* ćelija (ne s ćelijama u plavom i sivom području). Zbog toga TT račun zahtjeva *k* a ne *n* instanci. Zbog toga trebamo u glavnom programu pokrenuti još jednu *execute* funkciju

koja će stvoriti *k* instanci nove funkcije od kojih će svaka predstavljati jednu *k*-tu ćeliju i računati TT međudjelovanje. Nova funkcija se zove *GPUTiled2k*() i ima k instanci.

```
public void GPUTiled2k(){

        //konstante
        ki= getGlobalId();//ki-ta instanca
        float t1x=tilecenterx2[ki];
        float t1y=tilecentery2[ki];
        k=0;
        min=0;                                          //prvi  j-ti element u ćeliji
        while(k<tileNum){
                //------------------------------
                max=tilebreaksacumulated[k+1];          //posljednji  j-ti element u ćeliji
                //<<<<<<<<<<<<<<<<<<<<<                 TT međudjelovanje
        //      //6 flops
        //      deltax=t1x-tilecenterx2[k];             //
        //      delty=t1y-tilecentery2[k];              // CalculateRadius(ki,k);
        //      distanceSQ=deltx*deltx+delty*delty;     //
        //      mass=max-min;
    TT  //      if(distanceSQ>=limit2CellRadSQ){
        //      //8 flops
        //              recDistanceSQ=rsqrt(distanceSQ*distanceSQ);
        //              acxtot+=recDistanceSQ*deltx*mass;
        //              acytot+=recDistanceSQ*delty*mass;
        //      }
                //<<<<<<<<<<<<<<<<<<<<<                 TT međudjelovanje
                min=max;
                k++;
                //------------------------------        unrolano 5 puta
        }

        deltaVelxxTmp [ki]=acxtot*attCdeltat;
        deltaVelyyTmp[ki]=acytot*attCdeltat;

}
```

(29)

Ova funkcija ima samo TT međudjelovanje. Sastoji se od dva dijela. Jednog koji računa udaljenost među ćelijama i određuje je li se *k*-ta ćelija nalazi unutar drugog radijusa, te drugi dio koji koristi dio tog računa za izračunavanje akceleracija između *ki*-te i *k*-te ćelije to jest grupe čestica. Ti rezultati se spremaju u privremeni *deltaVelxxTmp* i *detaVelyyTmp* polje. Iz njih se ažuriraju *velxx* i *velyy* brzine. *deltaVelxxTmp*[ki] sadrži razliku u brzinama koja će se dodati svakom *velxx*[i] elementu koji se nalazi u *ki*-toj ćeliji. Ona je jednaka utjecaju svih čestica u trećem/rozom području na čestice u *ki*-toj ćeliji.

velxx brzine možemo ažurirati s *deltaVelxxTmp* na dva načina.

Prvi je da stvorimo *n* instanci funkcije od kojih svaka instanca za svoju *i*-tu česticu to jest za svaki *velxx*[i] element dodaje vrijednost koja se nalazi u *deltaVelxxTmp*[ki].

```
public void GPUNadder{//n instanci
        int ki;
        int i=getGlobalId();
        float t1x=xx[i];
        float t1y=yy[i];
                ki=searchTile2(i, t1x,t1y);        //trazi u kojoj se celiji nalazi i-ta cestica
                velxx[i]+= deltaVelxxTmp [ki];
                velyy[i]+= deltaVelyyTmp [ki];
}
```
(30)

Drugi način je da stvorimo *k* instanci funkcije u kojoj za svaku *k*-tu ćeliju dodamo vrijednost *deltaVelxxTmp*[k] u niz *velxx*[i] elemenata od *min* do *max.*

```
public void GPUKadder(){//k instanci
        int ki;
        int min=tilebreaksacumulated[ki];
        int max=tilebreaksacumulated[ki+1];
        for(int i=min;i<max;i++){
                velxx[i]+= deltaVelxxTmp [ki];
                velyy[i]+= deltaVelyyTmp [ki];
        }
}
```
(31)

Kako u glavnom programu već imamo dva poziva *execute* funkcije, jedan s *k* a drugi s *n* instanci, nije potrebno imati dodatne pozive *execute* funkcije, nego ovisno o tome koje od ova dva ažuriranja brzina koristimo, uvrstiti je u popis funkcija koje se pozivaju u *run()* funkciji. Naprimjer:

```
public void run(){
        if(kfrommain==1){ //k instanci
                GPUTiled2k();
         }
        if(kfrommain==2){ //n instanci
                GPUNadder();
                GPUTiled2n();
                SemiIntegrator();
                WallCollision();
        }
}
```
(32)

Ovo gore je kompletna *GPUTiled2* varijanta koja se sastoji od *GPUTiled2k* i *GPUTiled2n* funkcije ali i ostalih funkcija. *kfrommain* se šalje iz glavnog programa te određuje koja *execute* funkcija se izvršava.

Raspravimo na trenutak uvedene promjene. Dakle *GPUTiled1* varijanta ima $n \times n_k$ međudjelovanja u sivom području, $n \times (k - k_1)$ u plavom, što je približno $n \times k$ jer je k_1 jako mali, te ima $k \times k$ računanja udaljenosti među ćelijama za testiranje područja u kojem se nalazi *k*-ta ćelija. *GPUTiled2* ima $n \times n_k$ međudjelovanja u sivom području i $n \times (k - k_1 - k_3)$ gdje je k_3 broj ćelija u rozom području i on je jako velik. Zbog toga je broj kalkulacija u plavom području bitno smanjen i zamjenjen je efikasnijim $k \times k_p$ međudjelovanjima u rozom području. Broj kalkulacija za *GPUTiled2k* je:

$$N_{Tiled2k} = k(6k + 8k_3) + 8n \tag{19}$$

Gdje $6k$ dolazi od 6 operacija potrebnih za računanje radijusa i izračun mase čestica u ćeliji, $8k_3$ dolazi od 8 operacija koje se naprave samo za ćelije u rozom području. $8n$ dolazi od ažuriranja brzine u dva smjera s *Nadder*() funkcijom.
Uzevši u obzir da je broj plavih ćelija sada $k_2 = (k - k_1 - k_3)$ i da je on znatno manji nego prije, ukupni broj kalkulacija za *GPUTiled2* funkciju će biti:

$$N_{Tiled2} = n(5k + 12k_1 n_k + 15k_2 + 8) + k(6k + 8k_3) \tag{20}$$

Teorijsko ubrzanje naspram *brute force* varijante za *n*=262144 čestica, *k*=2500 i k_1=9 ćelija (r=2 ćelije), k_2=295 ćelija (r=10 ćelija) je 110 puta. Opaženo ubrzanje na grafičkoj kartici je 33 puta, odnosno uzevši u obzir i vrijeme sortiranja na procesoru ono je jednako 26 puta. Ugrubo je dvostruko brža od *GPUTiled1* metode za ovaj broj čestica. Još ćemo detaljnije malo kasnije razmotiriti ova programska ubrzanja. Iz jednadžbe (20) možemo vidjeti da je omjer prvog i drugog člana oko 100, to jest dva reda veličine. Drugi član u teorijskom razmatranju praktički možemo zanemariti iako vrijeme pozivanja i izvršavanja *GPUTiled2k* funkcije nije zanemarivo pogotovo zbog toga što kod manjeg broja instanci (2500) imamo manju efikasnost grafičkog računa. Teorijska razmatranja iako jesu zanimljiva imaju malo veze sa stvarnim mjerenjima jer ne uračunavaju:

1 veliki overhead pokretanja dvije execute funkcije

2 malu efikasnost grafičke kartice za slučaj malog broja k iteracija

3 malog broja j-tih elemenata za svaku instancu, bilo oni čestice u prvom području ili ćelije u drugom i trećem.

4 veliku latenciju dostavljanja podataka

5 korištenja if funkcija

6 vrijeme testiranja if uvjeta

7 neujednačenosti instrukcija koje se izvode na grafičkoj jezgri.

Ovo su mogući razlozi razilaženja teorijskog i mjerenog ubrzanja izračuna. Prije nego pokušamo objasniti tu razliku pogledajmo treću i posljednju programsku optimizaciju.

8.1.4 GPUTiled3

Treća verzija mijenja *GPUTiled2* verziju tako da umjesto da imamo jednu *k* petlju s (2500) prolaza, stvaramo dvije petlje. Jedna je *while* k_1 petlja za sivo/prvo područje a druga je *while* k_2 za drugo/plavo područje. Obije zajedno imaju oko 300-tinjak prolaza: ovaj broj naravno ovisi o veličini plavog k_2 i sivog k_1 područja. Međutim kako program zna koja ćelija je u plavom koja u sivom a koja u rozom području? U prijašnjoj verziji smo za svaku česticu računali je li se ona nalazi unutar prvog ili drugog radijusa. Sada radimo drukčije. Prije nego pokrenemo simulaciju napravimo *predradnje* u posebnom objektu (precCalc) kojima izračunamo sve udaljenosti među centrima ćelija za svaku ćeliju. Ovo možemo napraviti jer radimo s centrima ćelija koji su uvijek isti za svaku iteraciju a ne centrima masa koji su različiti. Također smo u tim predradnjama odredili za svaku *k* ćeliju koje su joj ćelije u plavom a koje u sivom području i njihove indekse spremili u pomoćna polja *sort00* i *sort11*. Ovim smo točno odredili kojih 295 plavih i kojih 9 sivih ćelija ulazi u *while* k_2 i *while* k_1 petlju. Time se izbjegava 2500 računanja udaljenosti među ćelijama za *n* čestica. Ažuriranja brzina se ne rade unutar PP i TP funkcija nego izvan njih a u funkcijama se rezultati spremaju u privremene *acxtot* i *acytot* varijable.

Nova funkcija se zove *GPUTiled3n*:

```
public void GPUTiled3n(){

//konstante
i=getGlobalId();
ki=searchTile2(i, t1x,t1y);
        kj=0;
        k=sort00[ki][kj]; // sort00[ki][kj] = index ćelija u sivom području                    (33)
        while(k!=tileNum){ // uvijet za izlazak, realno se iziđe prije 9 ciklusa za ćelije uz
rub
                min=tilebreaksacumulated[k];
                max=tilebreaksacumulated[k+1];
                PP(min, max, i,t1x,t1y);          // min, max služi za granice while j petlje
                kj++;
```

```
            k=sort00[ki][kj];
        }

        kj=0;
        k=sort11[ki][kj];          // sort11[ki][kj] = index ćelija u plavom području
        while(k!=tileNum){
            min=tilebreaksacumulated[k];
            max=tilebreaksacumulated[k+1];
            TP(min, max, i,t1x,t1y,k);        // min, max služi za masu k-te ćelije
            kj++;
            k=sort11[ki][kj];
        }
        velxx[i]+=acxtot*attCdeltat;
        velyy[i]+=acytot*attCdeltat;
}
```

Ako ovaj način ažuriranja brzina primijenimo na *GPUTiled1* verziju i nazovemo je *GPUTiled11* dobivamo oko 67% ubrzanja na grafičkoj kartici za 256K čestica te joj je brzina usporediva s *GPUTiled2*. Sve performanse ćemo još detaljno razraditi. Recimo da je broj plavih ćelija 295 (r=10 ćelija) i recimo da odredimo da se druga petlja ponavlja točno 295 puta tada će se za ćelije koje su uz rub prostora ćelija i koje nemaju 295 ćelija u svojem plavom području napraviti više kalkulacija nego je potrebno. Iz ovog razloga prekidamo *while* petlje prije i to u trenutku kada *k* bude jednak ćeliji s indeksom 2500. To nije 2500-ta ćelija nego 2501. Nju smo stavili kao pomoćnu ćeliju koja se nalazi izvan prostora ćelija i u kojoj se ne nalaze čestice ali služi kao kontrolna ćelija za izlazak iz petlji. Također možemo primijetiti da zbog toga što nemamo fiksan broj do kojeg se petlje izvode ne možemo ih odmotati.

Zbog toga što smo napravili predkalkulacije s centrima ćelija, te rezultate možemo iskoristiti u TT računu. Jednu varijantu već imamo - *GPUTiled2k*. Prisjetimo se, račun s ćelijama računa utjecaj među ćelijama i bitno je da testira koje se ćelije nalaze u rozom prostoru. *GPUTiled2k* funkcija ima 6 kalkulacija koje izračunaju radijus, jednu operaciju usporedbe i zatim 8 kalkulacija sile ukoliko je ćelija u rozom prostoru. Ovo su još neke realizacije ove funkcije:

GPUTiled2k:

```
max=tilebreaksacumulated[k+1];
deltx=t1x-tilecenterx2[k];                    //6 flop
delty=t1y-tilecentery2[k];                    //
distanceSQ=deltx*deltx+delty*delty;           //
mass=max-min;                                 //
if(distanceSQ>=limit2CellRadSQ){                                    (34)
    recDistanceSQ=rsqrt(distanceSQ*distanceSQ); // 8 flop
    acxtot+=recDistanceSQ*deltx*mass;         //
    acytot+=recDistanceSQ*delty*mass;         //
}
```

GPUTiled2kHa1: (Ha1- 1 *help array*)

```
max=tilebreaksacumulated[k+1];                    //
deltxx=t1x-tilecenterx2[k];                       //2 flop
deltyy=t1y-tilecentery2[k];                       //
if(sort22[ki][k]>=limit2CellRadSQ){               //sort22[ki][k]  je radijus
        mass=max-min;                                          //9 flop
        recDistanceSQ=rsqrt(sort22[ki][k]*sort22[ki][k]);      //
        acxtot+=recDistanceSQ*deltxx*mass;                     //
        acytot+=recDistanceSQ*deltyy*mass;                     //
}
```

(35)

GPUTiled2kHa3: (Ha3- 3 *help array*)

```
max=tilebreaksacumulated[k+1];
        if(sort22[ki][k]>=limit2CellRadSQ){
                mass=max-min;                  // 5 flop
                acxtot+=sort22x[ki][k]*mass;   //sort22x[ki][k]  je recDistanceSQ*deltxx
                acytot+=sort22y[ki][k]*mass;   //
        }
        min=max;
        k++;
```

(36)

Analizirajmo koliko brzo radi GPUTiled2k funkcija na grafičkoj kartici ali i na procesoru. Performansa ove funkcije će ovisiti samo o broju ćelija tako da fiksiramo broj ćelija na 2500. Za izvođenje na procesoru ćemo umjesto rsqrt(distanceSQ*distanceSQ) koristiti 1/distanceSQ. Takve verzije ćemo nazvati *CPUTiled2k* funkcijama.

Tabela 6

	Flop	CPU	GPU
Tiledk2	14	13.7 ms	6.7 ms
Tiledk2Unrolled	14	13.7 ms	1.4 ms
Tiledk2Ha1	11	8.78 ms	16.75 ms
Tiledk2Ha1Unrolled	11	8.9 ms	9 ms
Tiledk2Ha3	5	11.11 ms	51 ms

Račun u trećem području za 2500 ćelija na grafičkoj kartici i procesoru. Vrijeme je prikazano u milisekundama, manje je bolje

Primijetimo prvo da su performanse procesora i grafičke kartice usporedive iako ona ima 2 reda veličine više računskog potencijala. To je zbog toga što ovaj problem nema mnogo kalkulacija. Broj operacija u TT funkciji oko 100 puta manji nego zbroj kalkulacija PP i TP funkcija. Ovdje se vidi prednost procesora, a to je da manje pakete podataka obradi brzo ili u najmanju ruku jednako brzo kao grafička kartica.

Tiledk2 ima najviše kalkulacija i ima veliku korist od odmotavanja na grafičkoj kartici (1.4ms). Ne koristi *sort* pred kalkulirana polja iako koristi puno manja pomoćna *tilecenter* polja.
Tiledk2Ha1 ima manje kalkulacija (11/14) od kojih smo neke zamijenili operacijama s memorijom pozivajući ih iz *sort22* polja. Ovo čini se pomaže procesoru ali usporava račun na grafičkoj kartici (16.75 ms)!
Tiledk2Ha3 ima najmanje kalkulacija (5/14) ali ima najviše komunikacije s memorijom preko *sort22, sort22x i sort22y* polja u kojima su podaci za *distanceSQ* odnosno *distanceSQ*deltx* odnosno *distanceSQ*delty*.

Procesor gubi malo performanse ali ne previše jer je balansiran da podjednako brzo obavlja memorijsku komunikaciju kao i računske operacije. Grafička kartica kod ove varijante značajno gubi performansu (51ms) . Iako grafička kartica ima jako veliku propusnost (5-10 puta više nego procesor), ima još veću procesnu moć (50-100 puta), ali ovolika procesna moć ne dolazi do izražaja jer radimo s premalim brojem elemenata (zasićenje je tek na 64K instanci). Isto tako ima i jako veliku latenciju. Kad je broj elemenata mali omjer latencije grafičke kartice i procesora je 200.

Možemo zaključiti da kada imamo mali broj elemenata (instanci) u obradi, latencija prijenosa podataka postaje najveće usko grlo. Kod procesora se pokazalo korisnim koristiti već izračunata rješenja spremljena u RAM umjesto da ih ponovo računamo, dok je kod grafičke kartice zbog velike latencije bilo bolje ponovo napraviti kalkulacije.

Ovo opažanje nismo testirali za veći broj elemenata zbog memorijskih zahtjeva pomoćnih ćelija ali takvo nešto je i navedeno kao preporuka u generalnom korištenju GPU-a.
Najbolja varijanta za TT račun se pokazala *GPUTiled2k* tako da ju opet koristimo i kod *GPUTiled3n funkcije*. Možemo je odmotati jer *while k* petlja u TT računu ima fiksan broj iteracija.
Procijenimo broj kalkulacija. Broj kalkulacija je skoro isti kao i kod *GPUTiled2* varijante u jednadžbi (20) samo što nemamo $n * 5k$ računanja udaljenosti među ćelijama. Tako da je broj kalkulacija:

$$N_{Tiled3} = n(12k_1 n_k + 15k_2 + 8) + k(6k + 8k_3) \quad (21)$$

Ili:

$$N_{Tiled3} = 12nk_1 n_k + 15nk_2 + 8n + 6k^2 + 8kk_3 \quad (22)$$

Teorijsko ubrzanje naspram *brute force* metode za *n*=262144 čestica, *k*=2500, k_1=9 ćelija (r=2 ćelije), k_2=295 ćelija (r=10 ćelija) je 197 puta. Opaženo ubrzanje na grafičkoj kartici je

80 puta ali uzevši u obzir i vrijeme sortiranja na procesoru ono iznosi oko 47 puta što je oko dvostruko više nego *GPUTiled2* i 4 puta više nego *GPUtiled1* verzija za 256K čestica.
Za 256K čestica i *GPUTiled1*, *GPUTiled2*, *GPUTiled3* verzije smo redom imali 17, 26 i 47 puta ubrzanje naspram *brute force* verzije.

8.2 Kratki rezime i usporedba performansi

Razmotrimo koje od prije diskutiranih hardverskih optimizacija možemo primjeniti?

Odmotavanje petlje:

Ako napravimo odmotavanje petlje u PP petlji 4 puta onda će se za brojeve čestica u ćelijama koji nisu višekratnici broja 4 dogoditi bar 3 kalkulacije viška. Npr za 105 čestica u ćeliji će se napraviti 27 prolazaka kroz petlju od kojih će zadnji prolazak računati s 106-om 107-om i 108-om česticom iako se one nalaze u drugoj ćeliji. Kada je broj čestica u ćeliji poprilično velik ovo je prihvatljiva optimizacija. Stoga odmotamo PP petlje u svakoj od 3 varijante.
TP dio računa ne možemo odmotati jer to zapravo nije petlja nego samo jedan element ali i on se nalazi unutar *while k* petlje pa možemo skupa odmotati PP i TP segment. Znači naprimjer odmotamo PP i TP kao jedan komad 4 puta, a već smo prije odmotali *while j* petlju unutar PP 4 puta. Ovo možemo napraviti kod prve i druge varijante, a ne možemo kod treće jer ona nema konstantan broj *while k* elemenata. Kod nje bi trik iz primjera s PP odmotavanjem petlje napravio puno veću numeričku grešku.
Tek toliko da spomenemo, između ove 3 varijante smo imali još barem 10 različitih implementacija od kojih smo neke upotrijebili, a neke odbacili, jedna od njih je bila implementacija gdje je *GPUTiled3* verzija imala fiksan broj iteracija za petlju koja iterira kroz sivo i kroz plavo područje. *Bug* je bio prilično suptilan, kada se račun izvodio na rubnim ćelijama koje nemaju 9 ćelija u sivom području nego recimo 6, program je iterirao preko tih 6 ćelija i onda je prešao preko ostale 3 koje su imale indeks 0 jer se polja koja reprezentiraju brojeve ćelija inicijaliziraju s nulama. Ispada da su rubne ćelije imale naprimjer 6 međudjelovanja s okolnim ćelijama i onda su imale 3 međudjelovanja s nultom to jest prvom ćelijom u virtualnom prostoru ćelija. Ukoliko je u nultoj ćeliji bila veća količina čestica javljala se jaka sila prema centru te ćelije na sve ostale čestice. Ukoliko nije bilo čestica unutar prve ćelije problem nije bio opaziv. *Bug* je bilo prilično teško za pronaći i riješiti. Možda su moguća i drukčija rješenja ali rješenje gdje se izlazi iz *while* petlje za rubne ćelije se pokazao najjednostavnijim i najbržim. Ipak zbog toga odmotavanje petlje nije moguće.
Kod druge i treće varijante programa imamo *GPUTiled2k* funkciju koja rješava račun u trećem dijelu prostora i ona se može odmotati. Ipak trebamo paziti da je to višekratnik od

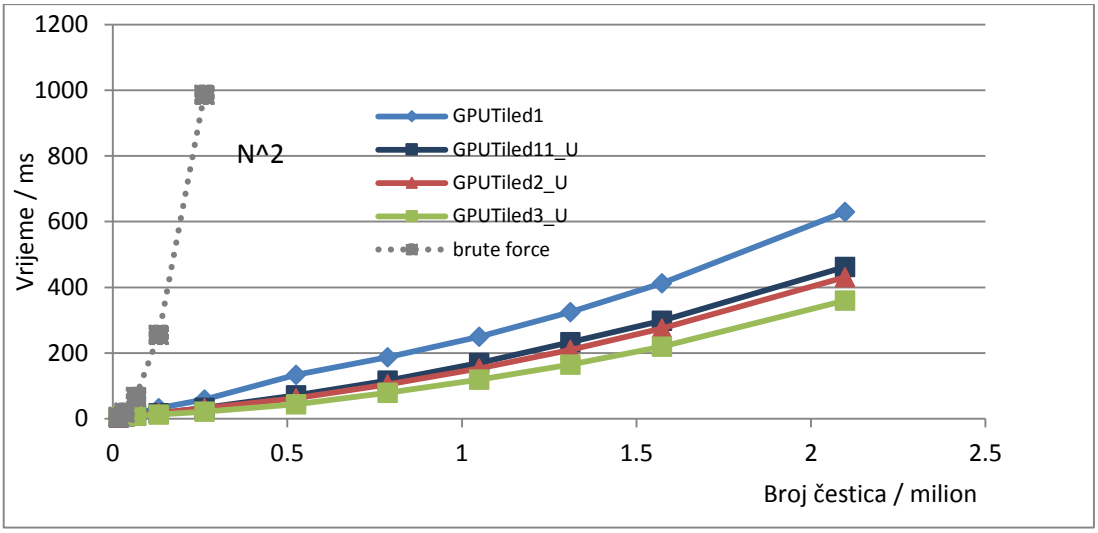
broja ćelija (2500). Sve 3 varijante u kojima postoji barem nekakvo odmotavanje dobivaju nastavak _U u svom imenu.

Memorijski blocking:

Zbog toga što radimo na Radeonu 7790 koji je pokazao da ima skoro pa maksimalnu performansu i bez memorijskog *blockinga* on onda nije potreban. Još jedan razlog zašto smo se odlučili za Radeon 7790 je taj što kod Radeona generacije 5000 i 6000 nismo mogli adresirati 2D polja u Aparapiju.

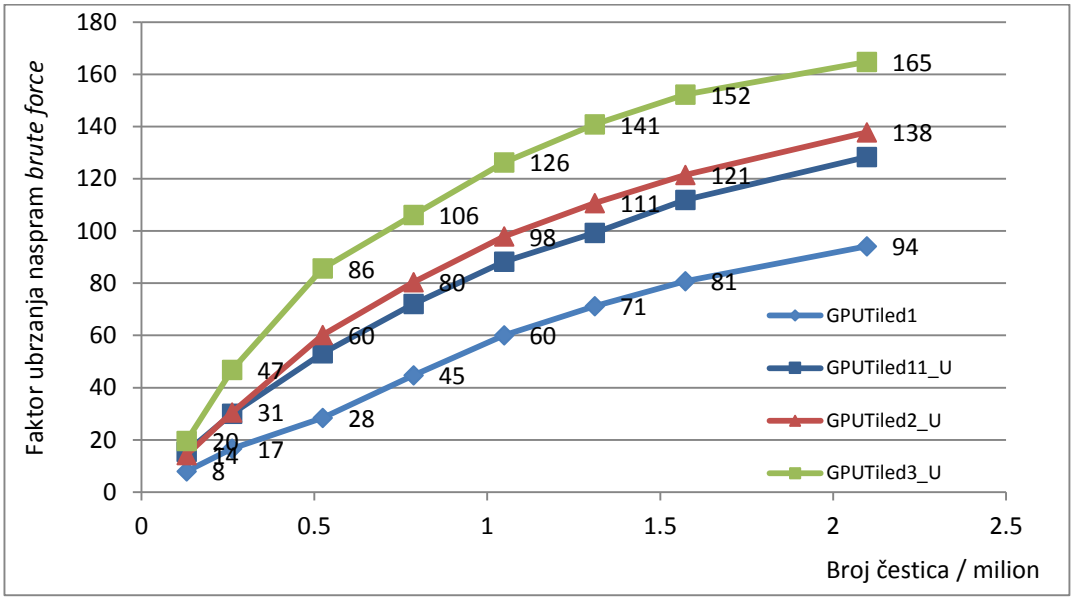
Pogledajmo sada kako ovise performanse naših metoda u ovisnosti o broju čestica.

Slika 25



Performansa brute force funkcije, obične GPUTiled1, GPUTiled11, GPUTiled2_U i GPUTiled3_U. Ukupni broj ćelija k=2500. Sivo područje k1=9 ćelija. Plavo područje k2=295 ćelija za varijante s rozim područjem a za prvu varijantu plavo područje ima 2491 ćeliju. Performansa pokazuje vrijeme potrebno za izvođenje jedne sličice za dani broj čestica kod svake metode.

Slika 26



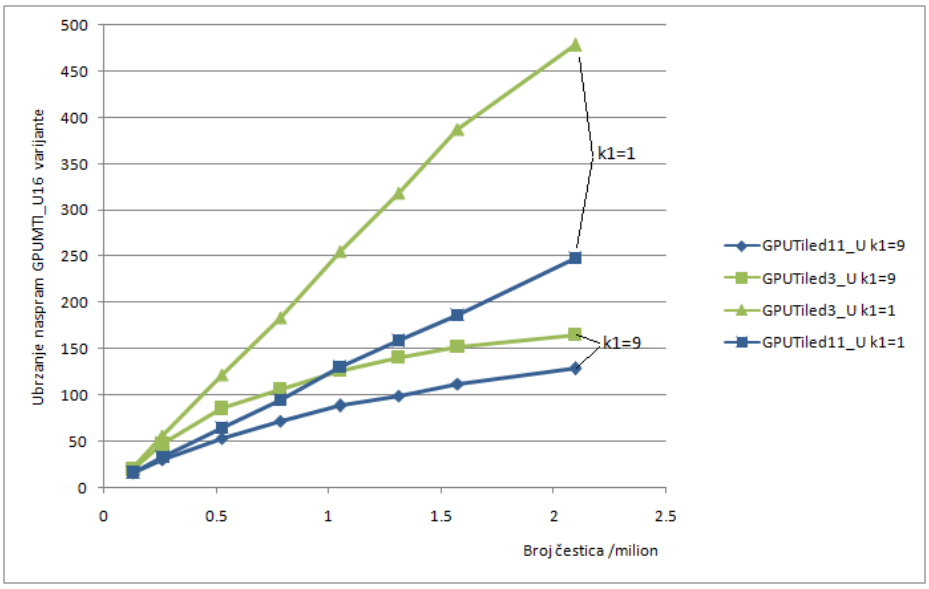
Obična GPUTiled1 metoda i 3 najbrže verzije od svake varijante. GPUTiled11 ima ažuriranje brzina izvan PP i TP funkcije. Performansa pokazuje koliko puta je pojedina varijanta brža od GPUTMI_U16 varijante (najbrže varijante "sirovom snagom") za dani broj čestica.

Na slici 25 vidimo da su *GPUTiled* varijante mnogo brže od *brute force* varijante kojoj vrijeme izvođenja raste s kvadratom broja čestica. U usporedbi s njom *Tiled* varijante imaju skoro pa linearni rast koji ipak ima slabu kvadratnu ovisnost. Ona se pojavljuje jer u jednadžbi (22) imamo $\mathbf{12k_1nn_k}$ faktor što se svede na:

$$\frac{12k_1n^2}{k} \tag{23}$$

Teško je pitati za koliko je naš program brži od *brute force metode.* Kako vidimo sa slike 26 njegovo ubrzanje se mijenja kako povećavamo broj čestica. Za veliki broj čestica ili veliki broj ćelija u sivom području dominira izraz (23) koji daje ovisnost $O(\frac{n^2}{k})$. Zbog čega i dalje imamo kvadratnu ovisnost koja je jako reducirana brojem ćelija. Na slici 26 je prikazana ovisnost performanse *Tiled* metoda relativno prema *brute force* varijanti. Primjećujemo da je ona iznimno velika za velike brojeve ali da joj se rast performanse smanjuje s brojem čestica. Ovo je zbog dominirajućeg kvadratnog člana kod velikog broja čestica. Ukoliko želimo nastaviti skaliranje performanse naših *Tiled* metoda za rastući broj čestica moramo ili smanjiti broj ćelija u sivom području ili smanjiti same ćelije.

Slika 27



Performansa izražena u faktoru ubrzanja relativno brute force metodi. GPUTiled11_U i GPUTiled3_U varijante za broj ćelija sivog područja k1=9 koja se isto tako može bolje vidjeti na slici 26 ovdje je stavljena usporedno s performansama tih istih varijanti ali s brojem ćelija u sivom području jednakim k1=1.

U ovom diplomskom nismo matematičkim računom diskutirali ovisnost preciznosti o broju i veličini ćelija programske optimizacije. I iako je jedna ćelija u sivom području premali broj za dobru preciznost ovdje to razmatramo samo u svrhu utjecaja na performansu.

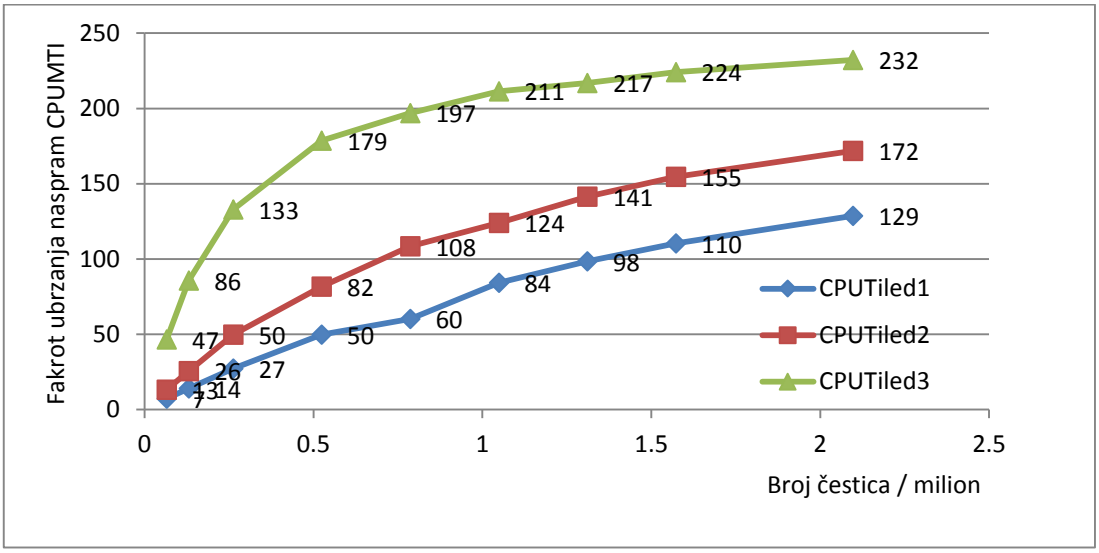
Vidimo sa slike 27 kako se skaliranje *GPUTiled11* i *GPUTiled3* metode nastavlja linearno ukoliko broj ćelija u sivom području smanjimo s 9 na 1. U ovom slučaju nam dominira drugi član u jednadžbi (22) koji je jednak $15nk_2$ te ovisi o broju ćelija u plavom području i linearno o broju čestica. Vrijeme izvođenja Tiled metoda na grafičkoj kartici se bitno smanjilo i sada utjecaj vremena sortiranja na procesoru postaje značajan. On zavisi također linearno s brojem čestica i za 2 milijuna čestica vrijeme sortiranja je oko 60 ms na Phenom II procesoru, otprilike isto kao i vrijeme izvođenja *GPUTiled3_U* metode na grafičkoj kartici za jednu ćeliju u sivom području i zajedno im treba oko 120 ms da renderiraju jednu sličicu.

S programskim optimizacijama postigli smo ubrzanje od impresivnih 450 puta za preko 2 milijuna čestica. Vjerojatno bismo mogli i više. Upitno je koliko ima smisla renderirati 2 milijuna čestica s dvojbenom preciznošću od jedne sive ćelije na grafičkoj kartici koja u isto vrijeme i iscrtava 2 milijuna piksela. Dakle i s jednom ćelijom u sivom području se dobivaju relativno dobri rezultati što su ćelije manje. Ipak ćelije ne treba smanjivati previše, jerdolazimo do granice kada grafička kartica ništa brže ne dohvaća podatke a povećava overhead pozivanja svih tih podataka. U analizu ovoga nismo dublje ulazili. Uostalom smanjivanjem veličine ćelija nam se povećava seračun u plavom području. Dakle osim

hardverske granice imamo i programsku granicu za veličinu ćelije. U našem slučaju ono je oko 20 piksela. Četvrti i peti član jednadžbe (23) imaju kvadratnu ovisnost isključivo o broju ćelija. Zbog toga što se ne isplati previše usitnjavati ćelije, čestica će uvijek biti barem jedan ili dva reda veličine više nego ćelija te će treći član s nk ovisnošću uvijek biti puno veći od k^2 ovisnosti četvrtog i petog člana. Račun u rozom području je za veliki broj kombinacija veličina ćelija i broja čestica mnogo manji s obzirom na plavo i sivo područje. Tada se naš problem n-tijela svodi samo na plavo i sivo područje.

Naše programske optimizacije se izvode jednim dijelom na grafičkoj kartici a jednim dijelom na procesoru. Pogledajmo kako se programske optimizacije ponašaju kada se izvode isključivo na procesoru.

Slika 28

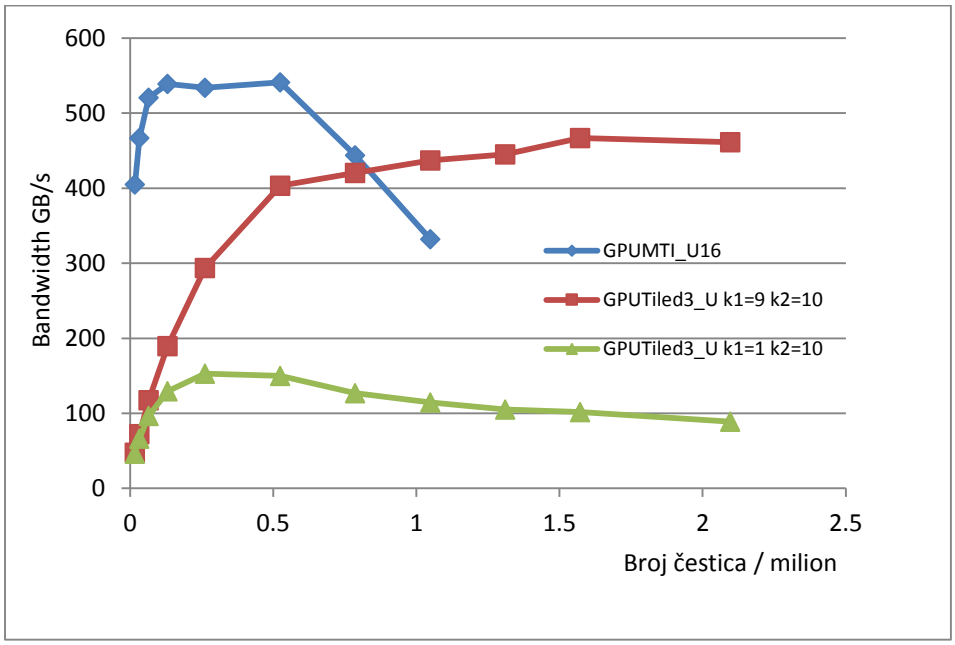


Performansa triju Tiled metoda optimiziranih za procesor izraženih preko usporedbe s CPUMTI to jest BasicNbodyMTI metodom izvedenoj na procesoru. Nemaju odmotavanja niti u jednom segmentu CPUTiled varijanti.

Naše 3 metode su optimizirane za procesor, nazvane su prema tome *CPUTiled*. Nisu odmotane jer se pokazalo da im to ne koristi ili sprječava izvođenje programa. Dijeljenje se izvodi s operacijom dijeljenja a ne korištenjem *rsqrt* funkcije. Dakle nema nikakvih značajnih hardverskih optimizacija. Performansa je izražena kao usporedba *CPUTiled* varijanti na procesoru s izvođenjem *CPUMTI* varijante na procesoru. Vidimo da performansa *Tiled* programskih optimizacija nije ni blizu onima na grafičkoj kartici. Ako usporedimo sliku 28 sa slikom 26 na kojoj je performansa grafičkih *GPUTiled* varijanti vidimo nekoliko stvari.
CPUTiled3 je postigao višu maksimalnu performansu s obzirom na *brute force* metodu na procesoru nego što je *GPUTiled3_U* ostvario naspram *brute force* metode na grafičkoj kartici.

Ovo smo mogli očekivati, to je zbog male latencije i malog broja jezgara kod procesora. Napomenimo da bismo procesor još mogli hardverski optimizirati, ali bi cijeli kôd bio kompleksniji. Zabilježimo da je *CPUTiled3* varijanta je na procesoru za 2 milijuna čestica 3 puta brža (20.6 sekundi) nego *brute force* varijanta (59.3 sekundi) na grafičkoj kartici.

Slika 29

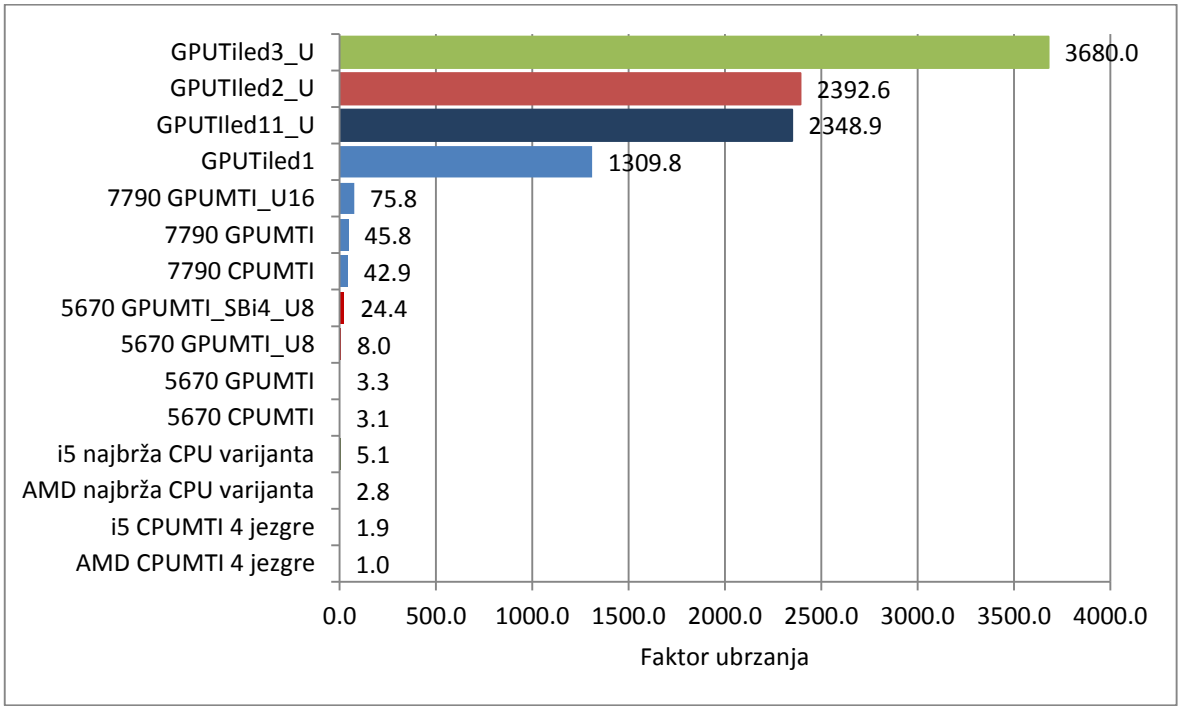


Memorijska propusnost za GPUMTI_U16 varijantu i za GPUTiled3 varijante sa sivim područjem od jedne i od devet ćelija.

Iz slike 29 se vidi da je memorijski zahtjev *GPUTiled3* verzije manji od *brute force* metode. Za ovo je primarno zaslužno sortiranje kojim se postiže lokalnost podataka i omogućuje kopiranje samo potrebnih podataka.

Na sljedećoj slici imamo rezime bitnijih ubrzanja dobivenih hardverskim ili programskim optimizacijama.

Slika 30



Ubrzanja normirana prema BasicNbodyMTI verziji na 4 jezgre Phenom II procesora na 3.2 Ghz. Mjerenja izvedena za 262144 čestica, k1=2, k2=10

9 Realni plin

Sada kada smo analizirali sve optimizacije možemo prijeći na realizaciju realnog plina. Za razliku od dosadašnje analize u realnom plinu ćemo isto tako imati i odbijanje među česticama. Također ćemo razmatrati i odbijanje s zidom kao i preciznije metode integracije.

9.1 Odbijanje čestica

Odbijanje možemo riješiti na nekoliko načina. Prvi je odbijanje tvrdih kuglica u kojem se uzimaju kutevi pod kojim se čestice sudaraju, također se uzima u obzir da sudari nisu frontalni. Potrebno je napraviti nekoliko translacija koordinatnog sustava. Kôd kojim smo ovo pokušali riješiti je imao oko 100-tinjak linija ali nije radio ono to smo namijenili da radi. Zbog kompeksnosti smo odustali od ovog pristupa i prešli na jednostavniji način, a to je da uvedemo odbojni potencijal koji ima veću vrijednost potencije na recipročnoj udaljenosti. Sam *Lennard-Jonesov* potencijal se iskazuje s:

$$V = -\frac{a}{r^6} + \frac{b}{r^{12}} \tag{24}$$

Gdje su a i b neke konstante plina. U praksi bilo kakvom kombinacijom odbojnog i privlačnog potencijala gdje nam je potencija na recipročnoj vrijednosti udaljenosti odbojnog potencijala veća nego na njegovoj privlačnoj komponenti, možemo postići efekt potencijalne jame.

Dakle isto kao što smo imali privlačnu komponentu koja je računala *acxtot* i *acytot* sada još dodamo i dvije linije kôda koje računaju i odbojnu komponentu. Ovo je PP račun s primjerom prostorno protežnog odbojnog potencijala:

```
while(j<number){
        deltx=t1x-xx[j];
        delty=t1y- yy[j];
        distanceSQ=deltx*deltx+delty*delty+e2;
        recDistanceSQ=rsqrt(distanceSQ*distanceSQ);
        acxtot+= attractiveConstant*recDistanceSQ*deltx;                    //privlačna sila x
        acytot+= attractiveConstant*recDistanceSQ*delty;                    //privlačna sila y
        acxtot+= repulsiveConstant*recDistanceSQ*recDistanceSQ*deltx; //odbojna sila x
        acytot+= repulsiveConstant*recDistanceSQ*recDistanceSQ*delty;//odbojna sila y
}
```
(37)

Ovo je primjer privlačne sile koja opada s $\frac{1}{r}$ to jest potencijal je $\ln(r)$ a odbojna sila opada s $\frac{1}{r^3}$ to jest potencijalom $\frac{1}{r^2}$.

U slučaju *Lennard-Jonesovog* potencijala pojavljuju se numeričke teškoće zbog strmine potencijala na malim udaljenostima. Odlučili smo koristiti manje strme potencijale jer i oni mogu dobro opisati problem koji razmatramo.Ustvrdili smo da stabilnost simulacije značajno ovisi o vrsti potencijala, preciznosti to jest *deltat* vrijednosti, gustoći čestica, prosječnoj brzini i dimenziji čestica tako da ćemo se zasada fokusirati na one parametre uključujući i odabir potencijala koji nam daju stabilno rješenje, to jest rješenje u kojem su energija i impuls očuvani barem kroz dulji period.

Iz prijašnjeg kôda možemo zapaziti da smo operaciju sudaranja zamijenili dodatnim operacijama koje imaju $O(n^2)$ težinu. Ovo nam se može činiti previše. Ako u prostoru imamo veliki broj čestica za koje znamo da su značajno udaljene nije potrebno da se za njih računa odbojni utjecaj to jest efekt sudara. Programska optimizacija s ćelijama nam ovdje može značajno pomoći jer ćemo zbog nje imati odbojni račun samo za čestice koje se nalaze unutar jedne ćelije za koje znamo da su relativno blizu. Za sve druge znamo da ne dolaze u obzir za

račun sudara. Ali razmotrimo koliko nam zapravo treba kalkulacija za račun sudara ukoliko ne bismo primjenjivali dalekodosežni odbojni potencijal.

9.2 Razmatranje sudara

Dvije čestice se nalaze u sudaru ako im se presjeku njihovi radijusi to jest ako je ukupna udaljenost među njima manja od 2 njihova radijusa. Tada možemo pokrenuti proceduru sudara. Jedna je *hard body collision* u kojoj se egzaktno računa sudar dvaju nestlačivih kuglica, a druga *soft body collision* u kojoj čestice mogu prodirati jedna u drugu. Pri tome se javlja jaka odbojna sila proporcionalna iznosu prodiranja jedne čestice u drugu ili proporcionalna nekom drugom potencijalu. Odlučili smo zbog računske jednostavnosti koristiti *soft body collision*. Komad prijašnjeg kôda je primjer *soft body collisiona.* To je zapravo nešto poput kulonskog odbijanja točkastih čestica.

Soft body collision možemo riješiti na dva načina. Jedan je preko potencijala koji se proteže kroz cijeli prostor kao u prethodnom primjeru. Drugi je preko potencijala koji se aktivira samo kada se čestice dovoljno približe. Postoje dva načina testiranja tog kritičnog položaja. Jedan je *a priori* a drugi *a posteriori*. *A posteriori* se pokreće nakon što su čestice već prodrle jedna u drugu, a *a priori* računa hoće li u sljedećoj iteraciji doći do prodiranja to jest sudara. Odabrali smo *a posteriori* princip koji je računski jednostavniji i intuitivniji. Ako uzmemo prethodni dio kôda to izgleda ovako:

```
while(j<number){
        deltx=t1x-xx[j];
        delty=t1y- yy[j];
        distanceSQ=deltx*deltx+delty*delty+e2;
        recDistanceSQ=rsqrt(distanceSQ*distanceSQ);
        acxtot+= attractiveConstant*recDistanceSQ*deltx;
        acytot+= attractiveConstant*recDistanceSQ*delty;                    (38)

        if(distanceSQ<criticalRadSQ){                    //testiranje sudara
                acxtot+= repulsiveConstant*recDistanceSQ*recDistanceSQ*deltx;
                acytot+= repulsiveConstant*recDistanceSQ*recDistanceSQ*delty;
        }
}
```

Znači kada se dogodi da su dvije čestice bliže od neke zadane udaljenosti, to ne mora biti baš dvije udaljenosti njihovog radijusa, onda se pokreće odbojna komponenta sile.

Na prvi pogled ovaj račun zahtjeva manje kalkulacija jer se procedura odbijanja izvodi samo za čestice koje se nalaze unutar zadanog radijusa. Program ćemo prvo testirati za implementaciju bez ćelija. Tako da ako imamo naprimjer 1000 čestica u cijelom prostoru za

svaku od njih moramo testirati sudar 999 puta. Ovo nam se opet svede na n^2 problem. Međutim ako je operacija usporedbe znatno kraća nego računanje same sile odbijanja, trebali bismo opaziti značajnu uštedu vremena jer će usporedba poništiti potrebu za većinu računa odbijanja. Od ukupnih 22 kalkulacije za grafičku realizaciju i 21 za procesor račun odbijanja ima 8 kalkulacija s pomičnim zarezom.

Ako koristimo testiranje sudara umjesto da djelujemo potencijalom na cijeli prostor vrijeme izvođenja je tek malo kraće, oko 15% na procesoru i 5% na grafičkoj kartici. Očekivali bismo veću uštedu jer većina parova čestica nije u sudaru i teoretsko ubrzanje varijante s testiranjem sudara nad varijantom s odbojnim prostorno protežnim potencijalom je od 57 do 61%. Kako ovo objasniti?

Operacije grananja jako usporavaju procesore s velikim kanalom[*] i *benchmarci* poput simulacija šaha čije je poteze teško predvidjeti su dobar test procesorske jedinice za predviđanje grananja. Stariji modeli grafičkih kartica su grananje rješavale tako da izvrše i A i B kôd te odaberu onaj koji nam je potreban nakon testiranja *if* funkcijom. Za ovo je potrebno vrijeme koje je jednako zbroju izvršavanja A i B kôda. Nismo u detalje ovo analizirali ali u našem kôdu ovakav efekt ili nije primjećen ili je zanemariv. Kod funkcija kod kojih smo imali *if-else* uvjet trajanje izvođenja nije bilo dvostruko duže od vremena izvođenja samo jednog uvjeta. Uostalom Radeon 7790 ima ugrađenu jedinicu za predviđanje grananja.

I kod procesora i kod grafičke kartice račun sudaranja je $O(k * n^2)$ težine, samo je kod nje k oko 0.95 a kod procesora je oko 0.85 te tako nemamo bitnog ubrzanja nad modelom odbojnog prostorno protežnog potencijala.

Pokazalo se radi numeričkih razloga da je ipak bolje koristiti testiranje odbijanja jer je tada simulacija stabilna.

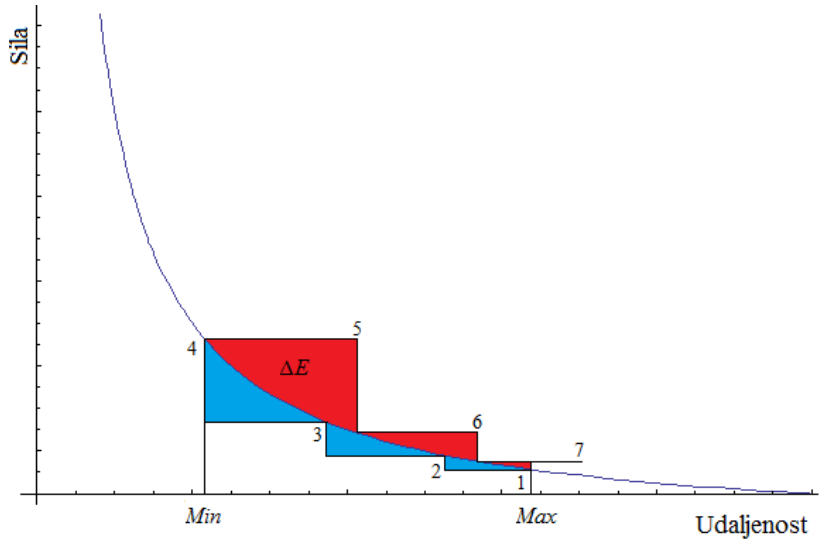
Zbog odabira *soft body collision*, čestice će imati različitu dubinu prodiranja jedna u drugu, i to ovisi o njihovoj brzini. Zbog toga ćemo za različite temperature plina imati različito prodiranje u plinu. Da bismo postigli bolji vizualni efekt potrebno je namjestiti konstante tako da čestice ne prodiru previše jedna u drugu ili radijus na kojem se aktivira odbojni potencijal postaviti dalje od realnog radijusa čestice, na naprimjer 1.5 ili 2 radijusa čestice.

Ukoliko je prodiranje preveliko, to jest ako je odbojni potencijal preslab s obzirom na brzinu nadolazeće čestice, ona prođe kroz prvu česticu tako da se samo lagano otkloni umjesto da se odbije. Ovdje nam je korisna verzija s testiranjem sudara, jer možemo proizvoljno prilagoditi odbojni potencijal lokalno u okolini čestice bez utjecaja na ostale čestice u prostoru.

[*] Objašnjeno u odjeljku 5.13 Pipeline ili kanal na stranici 21

9.3 Očuvanje energije

Slika 31



Funkcija odbojne sile i hod kroz potencijal u konačnim deltat koracima

Promotrimo sliku 31. Radi lakše analize uzmimo funkciju sile a ne funkciju potencijalne energije. U idealnom slučaju čestice putuju kroz prostor u nekom intervalu vremena s beskonačno mnogo *dt* intervala u infinitezimalnim *dr* koracima. U numeričkoj simulaciji one putuju u nekom intervalu u konačnim *deltat* koracima većima od nule i pomiču se za neki *deltax* veći od nule. Razmotrimo situaciju u kojoj čestica putuje od točke *Max* na slici 31, staje u točki *Min* i vraća se u točku iz koje je krenula. Putujući od jedne točke do druge ona trpi silu te se na tom putu vrši rad na račun kinetičke energije te čestice koji je jednak površini ispod funkcije. Kad se ona vraća od *Min* do *Max* sila ju ubrzava to jest obavlja se rad na račun potencijalne energije polja sile. U fizikalnom slučaju čestica točno slijedi funkciju sile, te je rad pri dolasku i odlasku isti ali s različitim predznakom. Tako da je ukupan rad jednak nuli. Drukčije rečeno rad po zatvorenoj krivulji je nula. Međutim mi nemamo egzaktnu fizikalnu situaciju nego imamo numeričku simulaciju u kojoj imamo konačne korake kroz koje se djelovanje sile ne mijenja. Rezultat toga je da imamo *stepeničasto* hodanje duž funkcije sile. Dok se čestica približava ona gubi kinetičku energiju, u numeričkom slučaju ona izgubi manje energije jer ne gubi energiju iz plavog područja. Dok se ona udaljava ona dobiva kinetičku energiju (naznačeno crveno) i u numeričkom slučaju dobiva više energije nego što je dobiva u fizikalnom. Dok se ona ponovno nađe u istoj točki ona je dobila energiju to jest rad po zatvorenoj krivulji nije bio nula. Kroz dulji period cijeli sustav čestica (plin) dobiva ili gubi energiju.

Očuvanje energije u fizikalnim simulacijama s promjenjivim potencijalom je sasvim normalan i očekivan problem. Postoji nekoliko načina na koji se ovaj problem rješava.

Najjednostavnije je smanjiti *deltat.* Međutim može se uvesti i vrsta gušenja u sudarima. Jedan od osnovnih problema već jesmo riješili uvođenjem e2 konstante u račun udaljenosti među česticama koja efektivno izglađuje potencijal u neposrednoj blizini čestice. U situaciji gdje su čestice jako udaljene očuvanje energije poboljšamo uvođenjem manjeg *deltat.* Kod sudaranja u gustom plinu čestice provode jako puno vremena u području *strmog* potencijala ili velikog gradijenta potencijala pa je je narušenje očuvanja energije u takvim slučajevima još brže.
Postoje također razne numeričke metode koje bolje ili lošije čuvaju energiju sustava. Nismo previše vremena potrošili na njihovo proučavanje već smo samo uzeli metodu koja nam za naš slučaj garantira očuvanje energije. U sljedećem odlomku ćemo je kratko opisati.

9.4 Leapfrog metoda

Metode o kojima govorimo su zapravo metode integriranja o kojem je bilo nešto govora na početku. Naša osnovna metoda je eulerova. Izračunamo sile na osnovu trenutnog položaja, iz toga dobijemo akceleracije i brzine i ažuriramo sljedeći položaj.

$$v_{i+1} = v_i + a_i \Delta t \tag{27}$$

$$x_{i+1} = x_i + v_{i+1} \Delta t \tag{28}$$

Prethodne dvije jednadžbe predstavljaju integrator. Prepišimo kôd tako da sada ne ažuriramo odmah *velxx* i *velyy* brzine nego izračunatu akceleraciju ažuriramo u *accxx*[] i *accyy*[] polja a ažuriranje brzine prebacimo u integrator.
Takav integrator je potpuni integrator:

```
public void FullIntegrator(){
        int i=getGlobalId();
        velxx[i]=velxx[i]+accxx[i]*deltat;
        velyy[i]=velyy[i]+accyy[i]*deltat;

        xx[i]=xx[i]+velxx[i]*deltat;
        yy[i]=yy[i]+velyy[i]*deltat;
}
```
(39)

Kao što smo vidjeli, dok se čestice gibaju kroz promjenjivi potencijal poput harmoničkog, eulerova metoda ne čuva energiju. Promotrimo računski što se događa. Prepišimo jednadžbu (27) tako da vrijeme teče unatrag. Poslužimo se slikom 31 i obratimo pozornost na točku *3* i *4*. Prema jednadžbi (27) brzina u točki *4* je jednaka brzini u točki *3* plus $a_3 \Delta t$. Ako krenemo

unatrag iz točke *4* brzina u točki *3* neće biti brzina u točki *4* minus $a_3\Delta t$ nego minus $a_4\Delta t$. Brzina a time i energija nam nije ista. Jednadžba brzine kada vrijeme ide unatrag je:

$$v_i = v_{i+1} - a_{i+1}\Delta t \tag{29}$$

Ako ovo uvrstimo u jednadžbu (27) dobijemo:

$$a_{i+1}\Delta t = a_i\Delta t \tag{30}$$

Što NIJE točno. Važno je za primijetiti da akceleracija u točki 4 nije ista već ovisi o smjeru gibanja. Jasno je da i položaj kod kretanja unatrag neće biti isti što je i vidljivo iz slike 31.
Postoji metoda integracije koja se naziva *leapforg* metoda. U diferencijalnim jednadžbama drugog reda poput onih kod kojih akceleracija ovisi samo o položaju *leapfrog* metoda integracije čuva energiju sustava. Ovo je moguće zato jer je ona vremenski reverzibilna.
Ona funkcionira na principu toga da definiramo iznose brzina u poluperiodu. Položaj i akceleracije su definirani u cjelobrojno.

$$v_{i+1/2} = v(t + \frac{\Delta t}{2}) \tag{31}$$

Definirajmo nove položaje i brzine.

$$x_{i+1} = x_i + v_{i+1/2}\Delta t \tag{32}$$

$$v_{i+3/2} = v_{i+1/2} + a(x_{i+1})\Delta t \tag{33}$$

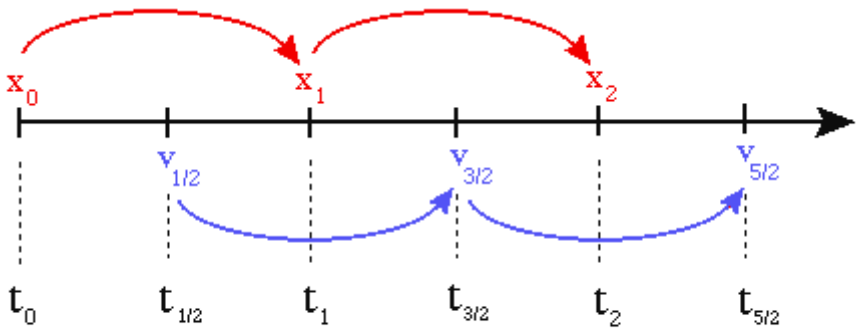
Računamo *i*-tu iteraciju. Iako su naznačeni poluperiodi oni ne postoje kao zasebna iteracija računa. *Novi položaj se dobiva od brzine koja se nalazi pola perioda ispred trenutnog perioda. Nova brzina se dobiva od akceleracije koja se nalazi pola perioda ispred trenutne brzine.*
Pogledajmo što se događa kada vrijeme ide unatrag.
Nalazimo se u i+1 trenutku i idemo unatrag. Početni položaj nam je x_{i+1} i idemo prema x_i, a početna brzina nam je $v_{i+3/2}$. Znači ne brojimo i, i+1/2, i+1, i+3/2 itd. nego brojimo i+1, i+1/2, i, i-1/2 itd. Kao argument uzimamo brzinu koju će čestica imati pola perioda *ispred trenutnog perioda* i s obzirom da vrijeme ide u natrag to je $v_{i+1/2}$ iz jednadžbe (35). Da bismo izračunali tu brzinu moramo se poslužiti akceleracijom koja se nalazi pola perioda ispred trenutne brzine $v_{i+3/2}$. Ako uzmemo u obzir tok vremena dobivamo jednadžbu (34).

$$v_{i+1/2} = v_{i+3/2} + a(x_{i+1})(-\Delta t) \tag{34}$$

$$x_i = x_{i+1} + v_{i+1/2}(-\Delta t) \tag{35}$$

Kada pomnije sagledamo jednadžbe (34) i (35) vidimo da su one zapravo jednadžbe (32) i (33) napisane u drukčijem obliku. Ako pogledamo sljedeću sliku stvari će nam biti jasnije.

Slika 32



Leapfrog metoda

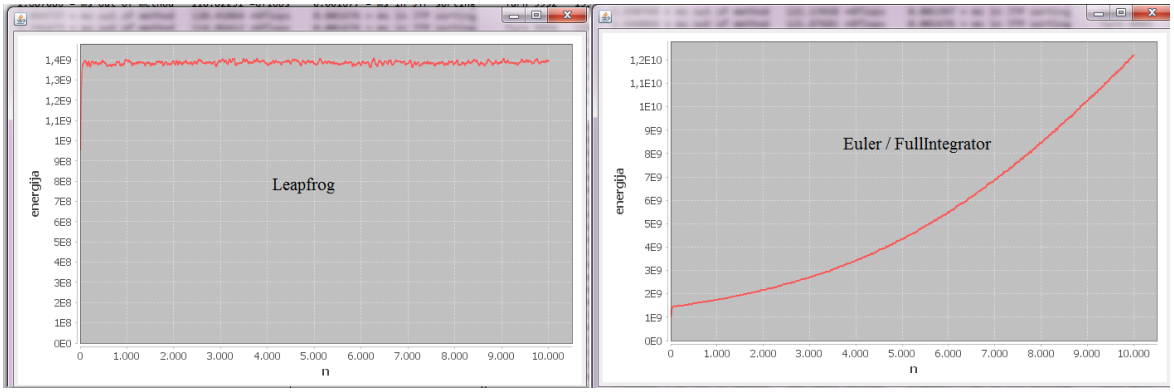
Ako samo okrenemo smjer strelica vidimo da bez obzira na smjer vremena mi u istoj točki računamo istu akceleraciju dok u eulerovoj metodi akceleracija za istu točku ovisi o smjeru vremena/gibanja kao što smo mogli vidjeti na slici 31.

Ako je metoda vremenski reverzibilna onda se gibajući u dva smjera istog pravca giba uvijek po istim točkama i ima uvijek iste iznose akceleracija a time i brzina.

Ovo je iznimno važna stvar. Jer omogućava da simulacija čuva energiju za harmoničke oscilacije poput naše!

Bez obzira na teoretsku podlogu bilo nam je poprilično iznenađenje kada je ova metoda zaista i proradila u našoj simulaciji.

Slika 33



Količina kinetičke energije sustava kroz 10000 iteracija

Programska realizacija *leapfrog* metode je realizirana u dva dijela:

Prvom koji se računa prije računa sile:

```
public void LeapfrogIntegrator0(){

        int i=getGlobalId();
        velxx[i]=velxx[i]+accxx[i]*deltat*0.5f;
        velyy[i]=velyy[i]+accyy[i]*deltat*0.5f;

        xx[i]=xx[i]+velxx[i]*deltat;
        yy[i]=yy[i]+velyy[i]*deltat;

}
```

(40)

I drugom koji se računa nakon računa sile to jest nove akceleracije:

```
public void LeapfrogIntegrator1(){

        int i=getGlobalId();
        velxx[i]=velxx[i]+accxx[i]*deltat*0.5f;
        velyy[i]=velyy[i]+accyy[i]*deltat*0.5f;

}
```

(41)

Za razliku od *FullIntegrator* metode ovdje se dva puta ažurira brzina s pola iznosa akceleracije s tim da se drugi put ažurira novo izračunata akceleracija.

9.5 Odbijanje sa zidom

Želimo da taj naš plin bude u kutiji. Tako da moramo riješiti odbijanje sa zidom. Postoji nekoliko načina na koji smo to napravili tako da ćemo ih kratko diskutirati.

9.5.1 WallCollisionDeltaV

Najjednostavnije rješenje odbijanja sa zidom je takvo da za svaku česticu testira je li se našla izvan prostora kutije te joj promijeni smjer brzine u x ili y smjeru ovisno o zidu u kojeg je čestica lupila. Ovo rješenje u nekom danom trenutku ima čestice izvan zidova i iako radi prihvatljivo za normalne simulacije nije pogodno za sustav ćelija kojeg smo razvili jer ako se čestica nađe izvan virtualnog prostora ćelija ona ili prestaje postojati ili dolazi do greške u programu.

9.5.2 WallCollisionDeltaVDeltaR

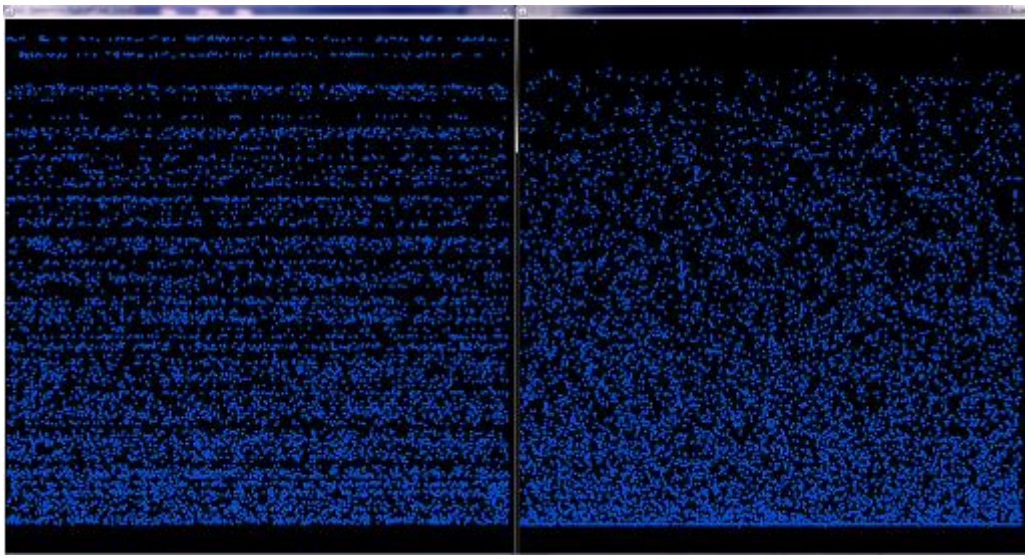
Iz ovog razloga smo dodali još jednu komponentu našem sudaru. Kada se čestica nađe izvan prostora kutije osim što promjeni smjer brzine ona promjeni i položaj tako da ako je izišla izvan zida za 3 piksela funkcija joj prebaci položaj 3 piksela unutar zida. Međutim ova varijanta odbijanja ne čuva energiju, čak ni s leapfrog metodom. Zamislimo česticu koja se nalazi 3 piksela udaljenosti od zida prije nego udari u zid. Neka njezina brzina bude 10 piksela u jednoj iteraciji. U sljedećoj iteraciji ona će se naći 7 piksela iz zida i naša funkcija će je vratiti 7 piksela s unutarnje strane zida. 3 + 7 =10. Stvar se čini u redu, gdje je onda problem s energijom? Ovaj primitivni račun je dobar kada u prostoru nemamo polja sile. Recimo da imamo vanjsku silu koja djeluje na česticu i *ubrzava* je prema zidu. Fizikalno gledano kada se ta čestica odbije ta sila će je usporavati. U našem rješenju mi izračunamo put u nekom *deltat* koji bi čestica prešla kao da smo je cijelim putem *ubrzavali* i onda jedan dio tog puta samo okrenemo na drugu stranu. Ispada da čestica ubrzava onaj dio puta (7 piksela) koji bi se trebala usporavati. Možemo napraviti složeniji račun sudara sa zidom koji će onda uzimati u obzir i ubrzanje. Ako uzmemo da čestice međusobno ne međudjeluju i da na njih djeluje neka *konstantna* vanjska sila onda je ubrzanje kao argument konstantno. Međutim čestice djeluju jedna na drugu i sila tih čestica na onu koja se sudara sa zidom nije uvijek ista nego je funkcija položaja svih čestica i za svaku česticu bi je trebalo ponovo računati. Takvo polje nije prostorno homogeno. Nije problem izračunati djelovanje sile svih čestica u jednoj točki, ali predstavlja problem ako moramo izračunati djelovanje svih čestica računati na nekom pravcu po kojem se čestica giba dok se odbija od zida. U ovaj problem nismo previše

zalazili ali značajno je prepoznati da je akceleracija kao argument funkcije sudara sa zidom u našem slučaju varijabla prostorne raspodjele čestica, barem onih u neposrednoj okolini ili ćeliji. Iako bismo ovom problemu mogli posvetiti više vremena našli smo rješenja koja su zadovoljavajuća a manje računski intenzivna.

9.5.3 WallCollisionAprioriDeltaV

Ukoliko čestice samo prebacimo s jedne strane zida na drugu mi ih transportiramo u prostoru bez da vodimo računa o silama koje u tom prostoru djeluju pa tako ne vodimo ni računa o radu koji bi se trebao izvršiti u tom transportu. Takva solucija ne čuva energiju. Iz ovog razloga napuštamo mijenjanje prostorne koordinate u sudaru sa zidom i ostajemo samo pri mijenjanju smjera brzine. Međutim onda moramo našu testiranje sudara izvesti prije nego čestica prijeđe na drugu stranu zida u prostor koji je izvan definiranog prostora ćelija. Zbog toga radimo *a priori* kalkulaciju koja računa hoće li se čestica u sljedećoj iteraciji nalaziti s druge strane zida i ako je to točno samo joj promijenimo brzinu bez pomicanja kroz prostor prije nego ona probije zid. Takva čestica se smatra odbijenom.
Niti ovakvo rješenje nije skroz fizikalno ispravno jer se dobivaju čudni uzorci čestica koje se grupiraju zajedno umjesto da su jednoliko raspoređene u prostoru.

Slika 34



Snimak nakon 10000 iteracija kod djelovanja sile prema dolje. Nema nikakvih drugih odbojnih ili privlačnih sila. Lijevo je slika odbijanja od zida bez korekcije položaja s WallCollisionAprioriDv metodom, desno je kompleksnija korekcija koja uzima u obzir silu prema dolje.

Lijevo je naša *WallCollisionAprioriDeltaV* metoda sudaranja sa zidom. Desna slika pokazuje rješenje puno složenijeg algoritma koji uzima kao argument djelovanje vanjskog polja prema dolje te na osnovu njega korigira položaj odbijene čestice. Ovdje smo to uspjeli riješiti jer nam je akceleracija tog polja *konstantna*. Da bismo koristili ovaj model kod realnog plina gdje

polje sile koje izaziva akceleraciju čestice ovisi o promjenjivom položaju svih čestica morali bismo imati funkciju koja izračunava to polje za svaku česticu uzimajući kao argument položaje svih drugih čestica. Ta akceleracija bi bila *funkcija* položaja drugih čestica a ne konstanta.

Iako dobivamo čudne uzorke kod sudaranja sa zidom bez korekcije položaja, očuvanje energije nam je ipak bitnije. U plinu u kojem dolazi do stalnog sudaranja među česticama ovaj efekt se poništi, barem vizualno. Ovo je kôd *WallCollisionAprioriDeltaV* funkcije:

```
public void WallCollisionAprioriDeltaV() {

        int i=getGlobalId();
        float xxtest=    xx[i]+velxx[i]*deltat+accxx[i]*deltat*deltat;
        float yytest=    yy[i]+velyy[i]*deltat+accyy[i]*deltat*deltat;
                if(xxtest>rightwall){
                        velxx[i]=-velxx[i];
                }
                if(xxtest<0){
                        velxx[i]=-velxx[i];
                }
                if(yytest>bottom){
                        velyy[i]=-velyy[i];
                }
                if(yytest<0){
                        velyy[i]=-velyy[i];
                }
        }
```

(12)

9.5.4 Ostale metode sudaranja

Vrijedi spomenuti neke druge metode sudaranja sa zidom koja nismo previše analizirali. Jedna od njih je bazirana na *soft body* ideji. Čestica prodire u zid i kada prijeđe granicu zida javlja se sila u suprotnom smjeru. Ova metoda nije pokazala bolje osobine nego bilo koja druga ali smo je diskvalificirali zbog toga što nema čvrstu granicu koja nam treba za model ćelija.

Osim nje često se kao rješenje sudaranja sa zidom koje nije sudaranje nego prolaženje kroz zid. Čestica se nađe pred zidom i ako u sljedećoj iteraciji prodre u zid umjesto da se nađe s druge strane zida nađe se na nasuprotnom zidu s unutarnje strane kutije imajući brzinu i položaj koju je imala kada je prodrla u zid. Problem koji ima ova metoda je isti onaj koji imaju metode s korekcijom položaja a to je očuvanje energije zbog "teleportiranja" kroz prostor u kojem djeluje sila.

Vrijedi napomenuti da očuvanje energije nije 100% očuvano. Kada smo uveli potencijal koji se pali i gasi ako čestice dođu preblizu narušili smo očuvanje. Nismo studirali zbog čega nastaje problem ali je vrlo vjerojatno da čestice ne izlaze na istom mjestu u tom potencijalu u kojem ulaze pa rad po zatvorenoj krivulji opet nije očuvan. Ipak efekt ovoga nije previše zamjetan.

9.6 Burst limiter

Kako energija ipak nije očuvana moramo razmišljati o alternativnim načinima njezinog očuvanja. Leapfrog metoda značajno pomaže ona sama nije dovoljna. Skoro sve simulacije su pokazale da ako energija nije očuvana ona uvijek lagano raste. Njezin rast je brži što je manja preciznost ili *deltat,* Također opažaju se veliki skokovi energije kod manjih preciznosti. Ako su brzine čestica prevelike to jest preciznosti premale, može se desiti da u jednom intervalu čestica naglo dobije veću brzinu iz numeričkih razloga. Kada postoji previše ovakvih čestica događa se neka vrsta eksplozije koja je rezultat numeričke greške. Simulacija može biti stabilna s malim ali jednolikim porastom energije oko 20000 iteracija i onda u jednom trenutku energija može naglo početi rasti. Jedan način limitiranja ovoga jest da implementiramo algoritam koji traži ovakve čestice i smanji im brzinu.

Prvi algoritam se bazira na detektiranju čestica koje se nalaze u vrhu maxwellove raspodjele to jest čestica koje su naprimjer 15 puta brže od prosječne energije čestica. Za simulacije gdje je *deltat* toliko malen da stvara numeričke greške i eksploziju energije ova optimizacija značajno pomaže jer su gotovo sve čestice koje su 15 puta više energije od prosjeka numeričke greške. Međutim jasno je da postoji jedan manji postotak čestica koje statistički trebaju imati energiju iznad te granice. Algoritam detektira sve čestice koje se nalaze izvan te granice i stavlja ih naprimjer na energiju koja je 15 puta veća od energije prosječne čestice. Kroz neki dulji period ovaj algoritam guši energiju sustava jer čestice koje statistički trebaju imati visoku brzinu neprirodno usporava. Ukoliko se izjednače utjecaji rasta i gušenja energije možemo imati koliko toliko stabilnu razinu energije, barem u nekom većem intervalu vremena.

Drugi algoritam se bazira na tome da detektira koja čestica je ubrzala naprimjer 100 puta od svoje prijašnje brzine i onda je vrati na tu istu prijašnju brzinu. Problem s ovim algoritmom je što kada čestice uđu u kratkodosežni potencijal koji se pali i gasi one počnu zbog njega prirodno dobivati veliku brzinu pa je teže odrediti šta je numerička greška a što posljedica potencijala.

Burst limiter algoritme nismo previše analizirali i iako bi takvi ili slični algoritmi mogli značajno pridonijeti očuvanju energije u numeričkim simulacijama. Ova dva koja smo mi napravili nisu baš pouzdana jer su nepredvidivi. Za različite razine energije oni mogu smanjivati ili povećavati unutarnju energiju sustava.

9.7 Kondenzacija

Glavni cilj ovog diplomskog rada je bilo postizavanje što boljih numeričkih performansi u simulacijama realnog plina. Međutim sa simulacijama smo izračunali i dobili zanimljive fizikalne rezultate. Posebice je bilo zanimljivo razmotriti problem kondenzacije realnog plina.

Razmotrimo prvo fiziku kondenzacije:

Ako imamo dvije čestice koje miruju na nekoj udaljenosti i među njima djeluje sila, one će se privlačiti, sudariti, odbiti i vratiti na isto mjesto s kojeg su krenule te ponavljati proces ciklički. Da bi se plin takvih čestica mogao smatrati kondenziranim čestice trebaju ostati zarobljene u potencijalu sila čestica s kojim međudjeluju. Da bi one ostale zarobljene, a ne da se vrate natrag na poziciju s koje su krenule, one trebaju izgubiti kinetičku energiju koju imaju neposredno nakon sudara. S manjkom kinetičke energije neće se moći vratiti na prvotno mjesto te će tako ostati zarobljene u potencijalu druge čestice u njezinoj neposredno blizini.

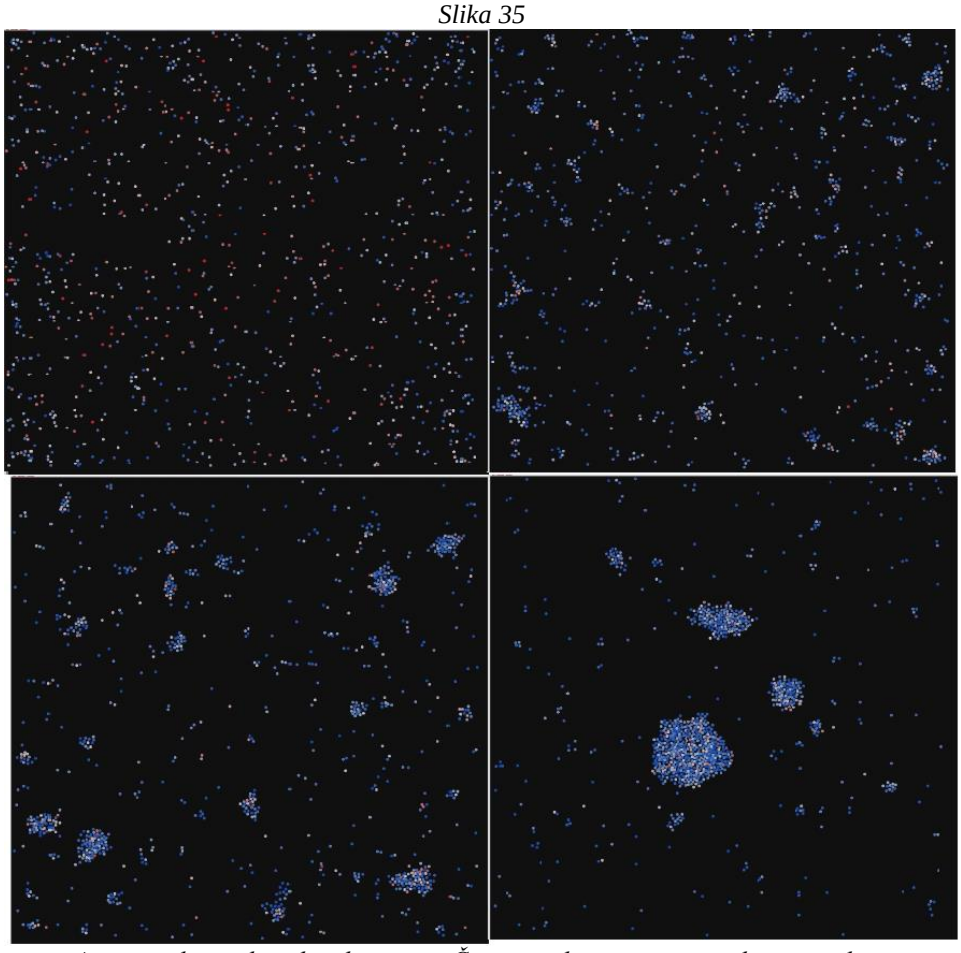
Da bi se mogao formirati potencijal koji drži drugu česticu u neposrednoj blizini prve moramo imati jednu privlačnu i jednu odbojnu komponentu potencijala. *Lennard-Jonesov* potencijal iz jednadžbe (24) ima djelovanje u neposrednoj okolini čestice i trne u nulu za neku veću udaljenost čestice. Međutim to je prestrmi potencijal za numeričku simulaciju pa koristimo potencijale manje strmosti koji se pale i gase na nekom radijusu. Da bismo ipak zadržali dalekosežno djelovanje potencijala u cijelom prostoru koristimo i treći potencijal.

```
while(j<max){

        deltx=nCalc1GPU(t1x, xx[j]);
        delty=nCalc1GPU(t1y, yy[j]);
        distanceSQ=deltx*deltx+delty*delty+e2;
        recDistanceSQ=rsqrt(distanceSQ*distanceSQ);
        acxtot+=recDistanceSQ*deltx*attractiveConstant;//ln(r)
        acytot+=recDistanceSQ*delty*attractiveConstant;                    (13)

        if(distanceSQ<criticalRadSQ){
                acxtot+=recDistanceSQ*recDistanceSQ*deltx*repulsiveConstant; // 1/r^2
                acytot+=recDistanceSQ*recDistanceSQ*delty*repulsiveConstant;
                acxtot+= recDistanceSQ*deltx*attractiveConstant2; //ln(r)
                acxtot+= recDistanceSQ*deltx*attractiveConstant2;
        }
```

```
        j=j+1;
    }
```

To izgleda otprilike ovako. Imamo standardni prostorno protežni potencijal koji djeluje na sve čestice u svakom trenutku. Imamo druga dva potencijala od kojih je repulzivni $\frac{1}{r^2}$ a privlačni *ln(r)* ovisnosti. Drugi privlačni potencijal ima puno veću *attractiveConstant2* konstantu. Samo su nam ova 3 potencijala potrebna da imamo efekt kondenzacije. Konstante je potrebno namjestiti da stvaraju potencijalnu jamu. Potrebno je samo uvesti gušenje brzine u sudaru sa zidom da simuliramo gubitak energije koji je potreban da bi se čestice plina kondenzirale u kapljice.

Slika 35

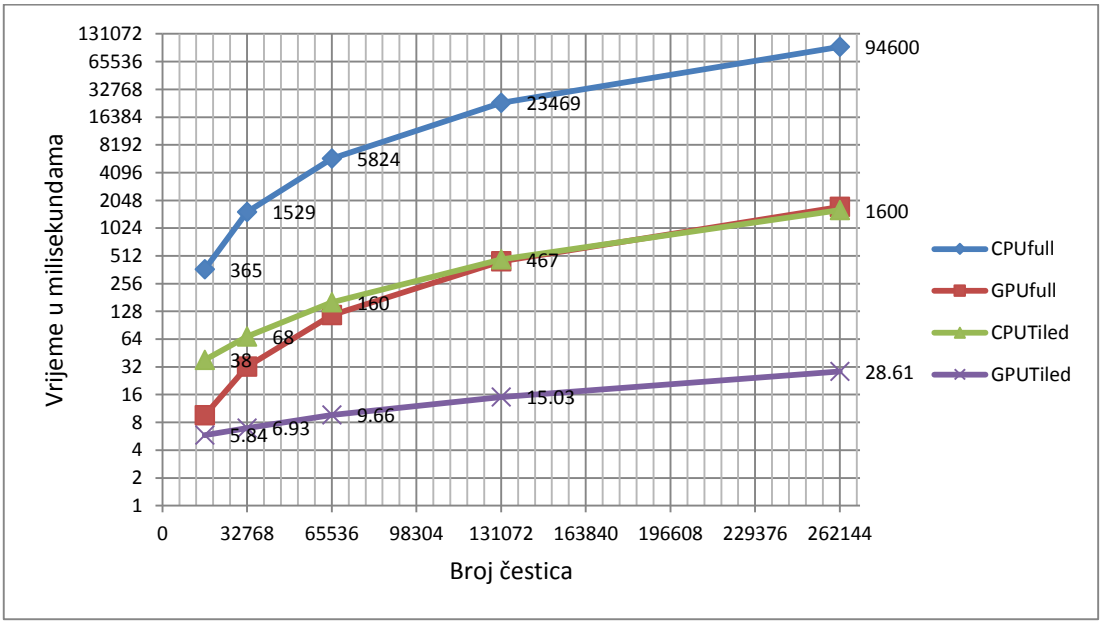


4 stanja plina tokom kondenzacije. Čestice gube energiju u sudaru sa zidom.

10 Fuzija

Sada kada imamo hardverske i softverske optimizacije te fizikalni model možemo sve skupa implementirati. Verzija bez ćelija na procesoru i grafičkoj kartici izvodi kôd (45) za cijeli virtualni prostor. Nazovimo te verzije *CPUFull* i *GPUFull*. Međutim kôd koji se izvodi s programskom optimizacijom ćelija nema u plavom i rozom području kalkulacije kratkodosežnih potencijala to jest sudara jer znamo da čestice koje nisu blizu (u istoj ćeliji) nisu niti u mogućnosti da se sudare. Tako da u plavom i rozom području imamo samo prostorno protežni privlačni potencijal, a tek u sivom području oko pojedine čestice djeluju sva 3 potencijala. Nazovimo ih *GPUTiled* i *CPUTiled* verzije. Na donjoj slici možemo vidjeti njihovu performansu.

Slika 36

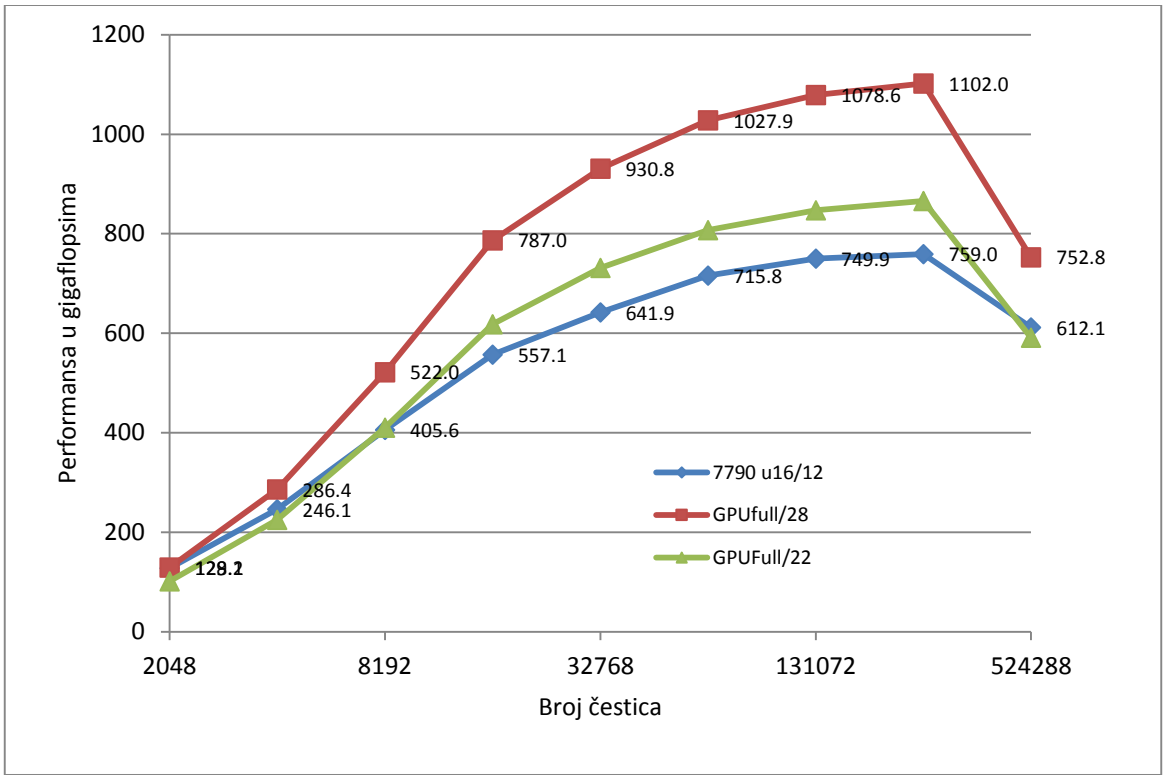


Vrijeme izvođenja je u milisekundama. Manje je bolje. Prostor je 5000x5000 točkica a broj ćelija je 2500. Sivo područje ima 9 ćelija a plavo oko 300.

Vidimo da za 254K čestica vrijeme izvođenja *GPUTiled* metode iznosi oko 30 ms što je oko 33 sličice u sekundi te se ta simulacija izvodi u realnom vremenu. Oko 56 puta je brža od procesorske verzije s ćelijama. Iako je ukupno vrijeme računanja jedne sličice na procesoru i grafičkoj kartici oko 30 ms, iscrtavanje sličica nije baš glatko za oko što znači da je realno renderiranje manje od 24 sličice u sekundi. Vjerojatno dolazi do uskog grla na podatkovnoj sabirnici koja prebacuje veću količinu podataka nekoliko puta tijekom jedne sličice.

Možemo primijetiti još jednu stvar. Usporedimo kako varira zasićenje procesne moći na grafičkoj ovisno o fizikalnom modelu koji se izvodi:

Slika 37



Performansa triju modela izražena u gigaflopsima u ovisnosti o broju čestica. Varijanta 7790 u16/12 ima 12 kalkulacija s pomičnim zarezom, ona je standardni problem n-tijela. GPUFull/22 ima 22 kalkulacije i ima samo kratkodosežno privlačenje i odbijanje. GPUFull/28 ima 28 kalkulacija i računa sva 3 potencijala.

Vidimoda se grafička kartica brže zasićuje ukoliko imamo više kalkulacija unutar *while j* petlje. Ovo možemo objasniti tako da grafička kartica traži podatke iz memorije i dok ih ne dostavi izvršava kalkulacije. Ukoliko je broj kalkulacija dovoljan da pokrije vrijeme koje je potrebno da se dostave podaci možemo reći da smo "sakrili latenciju" to jest vrijeme potrebno da se dostave podaci iz memorije.

Primijetimo također da u jednom trenutku imamo performansu od 1100 Gflopsa koja iznosi 58.5% teorijske performanse. Očekivana performansa je 50% teorijske. Ostvarili smo ovu performansu jer su ipak neke od instrukcija FMA instrukcije koje računaju množenje i zbrajanje u jednom ciklusu.

Kod Nvidijinih kartica bismo očekivali strmije zasićenje procesnog potencijala jer u pravilu imaju manje latencije od Radeona.

11 Metodički dio

U metodičkom dijelu nećemo analizirati cijeli diplomski nego ćemo samo pokušati metodički objasniti način na koji radi opisani hardver i neke od optimizacija.

Razlika između grafičke kartice i procesora:

Zamislimo da na nekakvom otoku imamo tvornicu koja nešto proizvodi. Ta tvornica treba sirovine. Proizvodnja iz sirovine u gotov proizvod neka budu operacije s pomičnim zarezom. Prijenos tih proizvoda natrag na kopno neka budu operacije zapisivanja u memoriju. Procesor bi bio uređenje gdje s brzim gliserom transportiramo potrebne sirovine na otok na kojem nema puno mjesta za skladištenje (registri). Nekoliko proizvodnih traka (procesorskih jezgara) koji rade na otoku brzo proizvode gotove proizvode. U takvom uređenju potrebno je izbalansirati prijenos sirovina (memorijske operacije) i brzinu proizvodnje (računske kalkulacije) da bi postojeće proizvodne trake uvijek imale nešto za raditi.

Grafička kartica bi bila takvo uređenje gdje na otok vozi trajekt. Na otoku ima puno više proizvodnih traka i ima ogromni prostor za skladištenje. Trajekt putuje jako dugo (ima veliku latenciju) ali sposoban je prevesti mnogo više sirovina nego gliser u nekom vremenskom roku (ima veću podatkovnu propusnost). Stvar je u tome da kod grafičke kartice trajekt prenese 10 puta više stvari ali na samom otoku grafičke kartice postoji 100 puta više proizvodnih traka nego kod procesorskog otoka. Tako da je kod grafičke kartice bolje proizvoditi proizvode koji zahtijevaju dulje vrijeme proizvodnje (više kalkulacija) po jednoj prenesenoj sirovini.

Da bi proizvodne trake stalno izbacivale proizvode potrebno je da budu popunjene i da sirovine (instrukcije i podaci) spremno čekaju na svoj red. Koji put dolazi do grananja u kôdu te se ne zna koji će podaci to jest sirovine biti potrebni. U tom slučaju procesor pogađa koje se sirovine moraju staviti na trake i ako pogriješi te sirovine se odbacuju te se stavljaju nove. Mora proći neko vrijeme dok se nove sirovine dostave na otok i stave na traku te dok ona opet počne izbacivati jedan proizvod svaki ciklus. Kod grafičke kartice bi zbog ovog mehanizma čekanja to trajalo predugo. Tako da kada na grafičkoj kartici dođe do grananja izvršavaju se obije instrukcije to jest naprave se dva proizvoda i onda se uzima onaj koji nam treba. Ovo teoretski produljuje vrijeme proizvodnje (izvođenja) ali je efikasnije nego alternativa. Zbog velikog vremena dostave sirovina na otok kod grafičke kartice je bolje urediti dostavljanje podataka na predvidljiv način tako da proizvodne jedinice unaprijed znaju koje će im sirovine biti potrebne a ne da zbog loše organizacije moraju dugo čekati spori trajekt. Grafička kartica srećom ima veliki skladišni prostor zbog kojeg su neki programi na njoj još lakši za

isprogramirati. Naime kod procesora je jako bitno balansirati koja vrsta sirovina se donosi na otok zbog malog skladišnog prostora, jer ako nestane sirovine koja je potrebna za proizvodni proces on staje. Grafička kartica ima veliki skladišni prostor tako da ne moramo voditi previše računa o pravom omjeru sirovina jer će na tom otoku biti mjesta za sve.

Grafička kartica također ima dijeljenu memoriju. Zamislimo to kao jedan dio otoka na koji se dostavljaju sirovine koje sve proizvodne trake mogu koristiti. Ovdje analogija prestaje biti dobra jer su sirovine potrošne i ako ih jedna proizvodna traka iskoristi druge trake ne mogu, dok se podaci mogu samo kopirati da bi ih drugi ALU-i koristili.

Uz to njezine proizvodne trake u grupama proizvode jedan te isti proizvod pa tako trebaju i istu vrstu sirovina. Tako da ako poneka proizvodna traka nema potrebne sirovine u svojoj okolini one se vrlo vjerojatno već nalaze negdje na otoku (dijelu memorije koji je dijeljen među trakama). Ovdje bismo ponovno naglasili važnost uređenog dopremanja podataka to jest programiranja tako da kada su jednoj proizvodnoj traci potrebne neke sirovine (podaci, instrukcije) da su one otprilike u isto vrijeme potrebne i drugim proizvodnim trakama iste grupe. Kada se one jednom dostave na otok dostave one su dostupne svim proizvodnim trakama. Nije potrebno da svaka traka zasebno traži trajekt svoj set sirovina (podataka) nego nekoliko proizvodnih traka naruči istu vrstu sirovine te trajekt u jednom putovanju dostavi sve to za sve proizvodne trake. Ovime se ostvaruje jedan od osnovnih principa optimizacije a to je ponovno korištenje podataka. Na ovaj način trajekt ne mora mnogo puta ići na kopno.

Grafička kartica je osjetljivija na kašnjenje u dostavljanju sirovina. Procesor je tu mnogo fleksibilniji jer ukoliko dođe do zastoja u proizvodnji gliser jako brzo dostavi potrebne sirovine. Procesor je doduše osjetljiviji na loše izbalansiranu dostavu sirovina zbog malog skladišnog prostora na otoku te takva dostava mora biti jako dobro planirana to jest isprogramirana. Kod grafičke kartice je zbog većih spremnika dovoljno samo dostaviti sirovine da bi se proces nastavio izvršavati. To znači da je u nekim situacijama kod procesora potrebno puno više finog podešavanja da bi se našla prava ravnoteža između broja ambalaže i sirovina koje se šalju na otok.

Superračunalo je računalo s mnogo jezgara i u našoj analogiji to bi bio veliki skup otoka na koje vozi gliser. Ovakav skup otoka je sličan grafičkom uređenju jer ima mnogo proizvodnih traka ali je u globalu komunikacija s kopnom puno brža nego kod grafičkog uređenja.

12 Zaključak

Reflektirajući se na ovaj diplomski rad mogli bismo zaključiti nekoliko stvari.

Programirati na grafičkoj kartici nije teško. Iako OpenCL može biti previše za početnika, biblioteke poput Aparapija umanjuju zahtjev za znanjem OpenCL-a ili CUDA-e i kôd pojednostavljuju maksimalno. Jedino što je potrebno znati da bismo efikasno mogli iskorištavati GPU hardver jesu neke od hardverskih optimizacija koje vrijede za standardne procesore ali i za superračunala. Na sreću ti koncepti nisu ništa novo i postojali su i prije pojave grafičkih kartica.

Grafičke kartice nisu zamjena za procesor, one su zamjena za superračunalo. Ukoliko se bavimo problemima koji se mogu dobro skalirati na superračunalima tada s grafičkom karticom dobivamo performansu manjeg superračunala kojem inače ne bismo imali pristup.

Jedna od glavnih razlika između grafičke kartice s mnogo jezgara i superračunala s mnogo jezgara jest to što jezgre superračunala imaju puno manju latenciju prema RAM-u iako im mrežna latencija (vrijeme odziva između čvorova) superračunala može biti i veće nego latencija između jezgara grafičke kartice. Latencija jest jedna od kritičnih elemenata za iskorištavanje maksimalnog računskog potencijala. Zbog toga moderne profesionalne grafičke kartice imaju 2 do 3 puta manju latenciju nego njihove igraće* varijante iako imaju isti broj ALUa. Uz to imaju i 4 puta veći RAM te su 10 puta skuplje.

Problem n-tijela s kojim smo se bavili ima koristi od manje latencije ali ona nije krucijalna da bismo iskoristili maksimum računske moći grafičke kartice. Zbog toga se ovaj problem može izvršavati efikasno i na jeftinijem igraćem hardveru.

Problem n-tijela je općenito iznimno zahvalan bilo kakvom grafičkom hardveru. Da bi se bilo koji problem efikasno izvršavao na grafičkoj kartici moramo imati zadovoljeno nekoliko uvjeta.

Prvi je programski model u kojem dretve ne interferiraju to jest ne zapisuju rezultate u skupove podataka koji se sijeku to jest svaka dretva zapisuje u svoj dio polja. Ovo je zadovoljeno izborom MTI varijante programa.

Drugi uvjet je da su čitanja i zapisivanja u polja serijska i bez skokova. Ovo je automatski zadovoljeno samom naravi problema koji čita serijski iz polja položaja i rezultat zapisuje jednom u polje akceleracije.

* Igraće grafičke kartice su ove na kojima smo izvodili testiranje.

Treći uvjet jest da dijeljenjem podataka nadomjestimo eventualno usko grlo u podatkovnoj propusnosti grafičke kartice. Ovo je ostvareno izvođenjem više dretvi na jednoj grafičkoj jezgri koje onda dijele podatke iz dijeljene memorije (predmemorija level 1 i 2 ili lokalna memorija) ali samo ukoliko dretve traže iste podatke u vrijeme kada se oni i nalaze u dijeljenim memorijama. Također optimizacije unutar samog Aparapija i Radeoni novije generacije automatski optimiziraju pristup dijeljenoj memoriji. Ukoliko se podaci dijele oni se ne moraju ponovo kopirati iz RAM-a te se ovime ostvaruje dovoljna podatkovna propusnost.

Četvrti uvjet jest da nakon što smo dostavili potrebne podatke što više opteretimo računski potencijal grafičke kartice da iskoristimo njezin paralelizam i sakrijemo latenciju. Ovo smo činili odmotavanjem petlji ili povećanjem broja kalkulacija. Ovaj uvjet je isto tako zadovoljen modelom koji zahtjeva 28 kalkulacija kod međudjelovanja realnog plina.

Važno je napomenuti da se zbog ova 4 razloga problem n-tijela ali i svi njemu slični problemi poput vremenske prognoze koja se zasniva na mjerenjima na *n* postaja i utjecajima tih mjerenja na okolnih *n* položaja mogu jako efikasno izvoditi na mnogo jeftinijim igraćim grafičkim karticama.

Problem n-tijela je bilo mnogo lakše optimizirati za grafičku karticu nego za procesor te smo uz to postigli 2 reda veću performansu nego na procesoru. Unatoč tome procesor nikada neće biti zamijenjen kao tehnologija jer će uvijek postojati kôd koji se ne može paralelizirati ili ga je iznimno teško paralelizirati pa će brzina jedne jezgre procesora i dalje ostati bitna. Kod nas se procesor pokazao ne samo korisnim nego i neophodnim za sortiranje kod modela s ćelijama. Ovime smo pokazali da grafička kartica i procesor mogu nadopuniti međusobne nedostatke. Ovo su prepoznale i vodeće kompanije koje razvijaju procesore. One već posljednjih nekoliko godina integriraju na čip od procesora i grafički čip ne samo radi uštede energije nego i radi izvođenja generalnih kalkulacija s tim grafičkim čipom. Svi laptopi, mobiteli i velika većina stolnih računala danas na istom komadu silicija ima tradicionalni i grafički procesor. Kao što se dogodila fuzija s RISC i CISC[*] modelima procesora tako se već dogodila fuzija tradicionalnog procesora i grafičke kartice. Ovaj računski potencijal se tek počinje iskorištavati.

Java, Aparapi i problem n-tijela zbog lakoće programiranja i jednostavnosti optimizacije čine jednu dobru platformu na kojoj se mogu lako i jednostavno učiti osnove paralelnog

[*] RISC = Reduced Instruction Set Computing, CISC = Complex Instruction Set Computing

programiranja te opaziti razlika između programiranja na procesoru i na grafičkom hardveru. Ova platforma je dobra odskočna daska za početnike u ovom polju.

13 Popis literature

[1] A simple benchmark of various math operations
http://latkin.org/blog/2014/11/09/a-simple-benchmark-of-various-math-operations/
10.02.2016

[2] Gallery of processor Cache Effects
http://igoro.com/archive/gallery-of-processor-cache-effects/
10.02.2016

[3] Rogue Wave ThreadSpotter , September 2011
http://ftp.emc.ncep.noaa.gov/exper/nova/temp/Tide/ThreadSpotter_Tutorial.pdf
10.02.2016

[4] Optimizing Made Easy: ThreadSpotter, 2011
http://www.vi-hps.org/upload/projects/hopsa/hopsa-nov12-threadspotter.pdf
10.02.2016

[5] 8 Steps to Optimizing Cache Memory Access and Application Performance, 2012
http://www.roguewave.com/getattachment/4b39d128-cbb0-42a0-bad9-726ecca72b8b/8-Steps-Optimizing-Cache-Memory-Access-Performance?sitename=RogueWave
10.02.2016

[6] Optimizing Memory Performance of an N-Body Simulation Using ThreadSpotter, 12.05.2011
https://www.youtube.com/watch?v=yHDDop36INM
10.02.2016

[7] Chapter 31. Fast N-Body Simulation with CUDA
http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html
10.02.2016

[8] AMD Accelerated Parallel Processing User Guide ™OpenCL December 2014
http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide.pdf
10.02.2016

[9] Aparapi Documentation , 27.09.2015
https://github.com/aparapi/aparapi/blob/master/doc/README.md
10.02.2016

[10] Intel® Xeon® Processor E5-4650 v2 (25M Cache, 2.40 GHz)
http://ark.intel.com/products/75289/Intel-Xeon-Processor-E5-4650-v2-25M-Cache-2_40-GHz
10.02.2016

[11] Radeon 7790 price, by W1zzard, 08.10.2013
https://www.techpowerup.com/reviews/AMD/R7_260X/
10.02.2016

[12] Cpu latency, by Ian Cutress 0*5.08.2015*
http://www.anandtech.com/show/9483/intel-skylake-review-6700k-6600k-ddr4-ddr3-ipc-6th-generation/9
10.02.2016

[13] Multi GPU Aparapi
https://github.com/aparapi/aparapi/issues/7
10.02.2016

[14] The Leapfrog Integrator
http://einstein.drexel.edu/courses/Comp_Phys/Integrators/leapfrog/
10.02.2016

[15] The Art of Computational Science 25.01.2004, 4.1 Two ways to Write the Leapfrog
http://www.artcompsci.org/vol_1/v1_web/node34.html
10.02.2016

[16] Van der Waals interaction (also known as London dispersion energies)
http://www.bio.brandeis.edu/classes/biochem104/VanderWaals.pdf
10.02.2016

[17] Lennard-Jones potential
https://en.wikipedia.org/wiki/Lennard-Jones_potential
10.02.2016