UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Tomi Erlih

VZPOREDNO RAČUNANJE S POMOČJO GRAFIČNE KARTICE

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Patricio Bulić

Ljubljana, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01639/2010
Datum: 15.03.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **TOMI ERLIH**

Naslov: **VZPOREDNO RAČUNANJE S POMOČJO GRAFIČNE KARTICE**
PARALLEL COMPUTING ON GRAPHICS PROCESSING UNIT

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V diplomskem delu preučite reševanje računsko zahtevnih problemov s pomočjo grafičnih kartic. Osredotočite se na problem simulacije gibanja N teles. Primerjajte hitrost izvajanja algoritma za simulacijo N teles na različnih procesnih platformah: na spošni centralni procesni enoti s SIMD ukazi, na večjedrnem računalniku ter na grafični kartici NVIDIA Tesla. Za razvoj in implementacijo algoritma na različnih arhitekturah uporabite prevajalnike GCC, Intel ICC ter vmesnike OpenMP, OpenCL in CUDA.

Mentor:
doc. dr. Patricio Bulić

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Franc Solina

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Tomi Erlih,

z vpisno številko 63030154,

sem avtor diplomskega dela z naslovom:

Vzporedno računanje s pomočjo grafične kartice

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  doc. dr. Patricia Bulića
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 15.09.2010 Podpis avtorja:

Zahvala

Zahvaljujem se doc. dr. Patriciju Buliću za vso pomoč, vse strokovne nasvete in usmerjanje pri izdelavi diplomske naloge.
Prav tako bi se rad zahvalil svoji družini za vso nudeno podporo in pomoč tekom vseh študijskih let.

Vidki in Bojanu

Kazalo

Seznam uporabljenih kratic in simbolov

GPGPU	(*General-Purpose computing on Graphics Processor Unit*) splošno-namesko računanje s pomočjo procesne enote grafične kartice
SIMD	(*Single Instruction stream, Multiple Data stream*) ena operacija za več podatkov
SSE	(*Streaming SIMD Extention*) pretočna razširitev SIMD
MSVC	prevajalnik Microsoftovega razvojnega okolja
ICC	(*Intel Compiler Collection*) zbirka prevajalnikov podjetja Intel
GNU	(*GNU's Not Unix*) akronim za "GNU ni Unix"
GCC	(*GNU Compiler Collection*) zbirka prevajalnikov projekta GNU
ALE	aritmetično-logična enota
MAD	(*multiply-add*) operacija vektorskega množenja in skalarnega seštevanja vrednosti množenja (vektorski produkt)
API	(*Application Programing Interface*) programski vmesnik
CUDA	(*Compute Unified Device Architecture*) poenotena arhitektura naprav za računanje
SIMT	(*Single-Instruction, Multiple-Thread*) en ukaz za več niti

Povzetek

V diplomskem delu je predstavljeno reševanje računsko zahtevnih problemov z uporabo grafične kartice. Današnje grafične kartice so zelo dovršeni in zmogljivi računski stroji, namenjeni predvsem vzporednemu računanju, kjer se ena operacija izvede nad več operandi. Za preizkušanje njihovih zmogljivosti je razvit testni program. Testni program simulira gibanje zvezd v vesolju, kjer vsaka zvezda zaradi gravitacijske sile privlači vse ostale zvezde. Rešitev problema temelji na simulaciji N-teles. Za izkoriščanje računske moči grafične kartice sta uporabljeni dve različni razvojni knjižnici. Prva knjižnica se uporablja za delo z arhitekturo CUDA, druga pa implementira standard OpenCL. Za realno oceno računske zmogljivosti grafične kartice jo primerjamo z zmogljivostjo vzporednega računanja procesorja ter z zmogljivostjo računanja z naprednim Barnes-Hutovim algoritmom. Cilj diplomske naloge je ugotoviti smotrnost uporabe takih tehnologij ter njihovo zmogljivost v primerjavi z že obstoječimi tehnologijami za vzporedno računanje.

Ključne besede:

grafična kartica, vzporedno računanje, simulacija N-teles, Barnes-Hutov algoritem, CUDA, OpenCL

Abstract

This diploma shows how to solve a compute-intensive problem using a graphics processing unit. Current graphics cards have a highly sophisticated and powerful processing unit, best suited for parallel computing where one operation is performed on multiple data. To test the performance of the graphics card, a test application was developed. It simulates the motion of stars in the universe, where each star pulls on all other stars using only its gravitational pull. N-body simulation is used to solve this astrophysical problem. Two separate development libraries are used to utilize the computing power of the graphics card. One library is used to work with the CUDA architecture while the other library implements the OpenCL standard. For the best estimate of the actual computing power of a graphics card, its performance is compared with the parallel capabilities of a common computer processor and with the performance of this processor when using the advanced algorithm Barnes-Hut. The goal of this diploma is to find out whether or not the use of this technology is feasible and to compare its performance with the parallel computing technologies that are used today.

Key words:

graphics card, parallel computing, N-body simulation, Barnes-Hut algorithm, CUDA, OpenCL

Poglavje 1

Uvod

Odkar človek šteje in računa, obstaja tudi težnja k čim hitrejšemu reševanju zmeraj novih in bolj zapletenih računskih problemov. Človeška sposobnost reševanja takih problemov zgolj z lastnimi možgani, je precej omejena tako količinsko kot tudi hitrostno. Človeški um kaj hitro postane ozko grlo, ko se stopnja zapletenosti računskega problema dvigne nad nekaj deset spremenljivk. A hkrati človeške kreativnosti pri reševanju takih problemov nikoli ne gre podcenjevati. Tako se že zgodaj v človeški zgodovini začeli pojavljati najrazličnejši pripomočki in naprave, s katerimi si je človek omogočil reševanje mnogih prezapletenih računskih problemov.

Če so nekoč ljudje za računanje uporabljali abake, nato papir in v moderni dobi kalkulatorje, se danes za vse znanstvene raziskave uporabljajo zelo zmogljivi računski stroji – računalniki. Njihova dominantna računska moč v primerjavi s človeškim umom je izrazita tako pri sposobnosti pomnjenja ogromnih količin podatkov kot tudi v izjemno hitrem izvajanju vseh računskih operacij, ki se uporabljajo pri reševanju današnjih računskih problemov.

Od izdelave prvih računalnikov pa do sedaj, je zmogljivost računalnikov strmo rasla. Zmogljivost računalnikov se je večala tako z višanjem takta, v katerem se operacije izvajajo, kot tudi z dodajanjem vedno novih komponent, ki so izboljšale način izvajanja operacij. A očitno se počasi približujemo praktičnim, pa tudi fizikalnim mejam izvedljivega, saj zadnji trendi kažejo, da za pohitritev računalnikov vse pogosteje posegamo po tretji metodi – večanjem števila ključnih računskih komponent računalnika. Tukaj v prvi vrsti mislimo na glavno obdelovalno enoto – procesor. Večje število procesorjev je uporabljeno predvsem zato, da se omogoči vzporedno izvajanje določenih delov obdelo-

vanja podatkov, od česar si obetamo tudi pohitritev računanja. Vendar to zaradi precej splošne naravnanosti te procesne enote ni popolna rešitev, saj nam dodatno procesorsko jedro doprinese le nekaj dodatnih računskih enot, medtem pa obseg računskih problemov raste eksponentno. Zato se v želji po še hitrejšem računanju usmerimo k drugi komponenti računalnika, ki že dobro desetletje izvaja vedno bolj obsežne in zapletene računske operacije nad veliko količino podatkov – k procesni enoti grafične kartice.

Grafična kartica oz. natančneje, procesna enota grafične kartice, je zasnovana tako, da lahko izjemno hitro izvaja isto računsko operacijo nad več zaporednimi podatki. Zaradi tega si z njo obetamo veliko večjo pohitritev pri računskih postopkih, ki omogočajo izkoriščanje take stopnje vzporednosti računanja. Od tod izvira tudi bistvo tega diplomskega dela. Zanimalo nas je, kako se vse povedano dejansko obnese v praksi – ali si res lahko obetamo večkratne pohitritve računanja z uporabo grafične kartice; kje so meje vzporednega izvajanja trenutne tehnologije; in ali jih tak način računanja tudi dejansko počasi izpodriva.

Poglavje 2

Vzporedno računanje

V naravoslovju se ves čas srečujemo z različnimi poskusi. Tekom vsakega takega poskusa zberemo veliko količino podatkov, ki jo je nato potrebno obdelati. Obdelava podatkov je dolgotrajen proces, saj zahteva veliko obdelovalne moči in učinkovit postopek računanja, t.j. **algoritem**. Prva je določena z računalniškim sistemom, ki nam je na voljo, zato se za pohitritev obdelave podatkov večinoma obračamo k slednjemu. Nadgradnja že obstoječega algoritma ne prinaša večjih pohitritev, vpeljava novega algoritma pa je večinoma dolgotrajna in podvržena veliki verjetnosti vpeljave napak. Zato se zdi vpeljava vzporednega računanja v določene dele že obstoječega algoritma še najbolj smotrna rešitev, od katere si obetamo tudi do n-kratno pohitritev takih delov računanja.

Seveda je n-kratna pohitritev celotnega postopka računanja le idealen primer, ki ga nikoli ne dosežemo. Kaj hitro nas pri tem začno ovirati narava problema, podatkovna odvisnost, kontrolne nevarnosti itd. Tako lahko vzporedno računanje izvajamo le na delu algoritma, preostanek pa se izvaja zaporedno. Čim večji je del algoritma, ki ga lahko izvajamo vzporedno, tem večja je pohitritev. O tem se lahko prepričamo s pomočjo Amdahlovega zakona [1] (enačba 2.1):

$$S = \frac{1}{f + \frac{1-f}{N}} \tag{2.1}$$

Vidimo, da je pohitritev S odvisna tako od deleža postopka f, ki se še naprej izvaja zaporedno, kot od števila vzporednih tokov računanja N. Iz enačbe je takoj razvidno, da je kljub neskončnemu številu vzporednih tokov računanja in le petini zaporednih operacij pohitritev računanja še zmeraj le 5-kratna.

Ob natančnejši analizi izvajanja algoritmov ugotovimo, da izvajamo pri večini teh določene dele pogosteje kot druge. Posledično iščemo pohitritev izvajanja ravno v teh delih algoritma. Ob še podrobnejšem pregledu ugotovimo, da take predele algoritma največkrat predstavljajo zanke. Včasih so te celo ugnezdene znotraj drugih zank. Najbolj notranje ugnezdene zanke so za nas najzanimivejše, saj že vsaka najmanjša pohitritev le-teh doprinese k precejšnem deležu celotne pohitritve izvajanja. Časovne pridobitve izvajanja se namreč seštevajo z vsakim obhodom take zanke ter množijo z vsakim obhodom zunanje zanke. Še več, hkrati zanke pomenijo tudi ponavljanje izvajanja istega zaporedja operacij, kar nam predstavlja odlično iztočnico za izvajanje takih zaporedij vzporedno.

Vzporedno računanje lahko vpeljemo v algoritem na dva načina. Bodisi uporabimo koncept procesov ter niti ali pa z vpeljavo vektorskih ukazov. Pri prvem izvajamo določene dele algoritma vzporedno, pri drugem načinu pa zamenjamo več istih zaporednih ukazov, izvedenih na zaporedno ležečih podatkih, z vektorskimi ukazi. Seveda pa lahko uporabimo tudi kombinacijo obeh.

2.1 Procesorska jedra in niti

Tekom razvoja računalniških procesorjev so določeni koncepti močno pripomogli k večanju zmogljivosti procesorjev. Razdelitev izvajanja ukazov na več stopenj *cevovoda* [1] omogoča, da se naenkrat izvršuje več ukazov. Z uvedbo *superskalarnosti* se v obdelovalni stopnji cevovoda izkoriščajo proste funkcijske enote, zaradi česar se lahko obdela več ukazov hkrati.

Seveda pa nam vse omenjene pohitritve ne bi prav nič koristile, če ne bi ob tem pohitrili tudi dostopa do podatkov. V splošnem velja, da čim bližje so podatki funkcijskim enotam, tem hitrejši je dostop do njih. A hkrati v splošnem tudi velja, da čim blijže je pomnilnik funkcijskim enotam, tem manjša je velikost takega pomnilnika. Velja torej, da sta hitrost dostopa do podatkov in velikost pomnilnika obratno sorazmerna. Zaradi tega ogromnega pomnilnika ni možno vključiti na sam procesor. Procesorji tako že od nekdaj izkoriščajo princip *prostorske* in *časovne lokalnosti* dostopov do podatkov. Ta lastnost nam omogoča, da pred glavni pomnilnik hierarhično namestimo več stopenj manjših, a zelo hitrih pomnilnikov – *predpomnilnikov*, ki vsebujejo podmnožico podatkov glavnega pomnilnika.

Kljub izjemnim pohitritvam, ki so jih prinesle omenjene rešitve in ob doseganju mejhnih hitrosti obdelave podatkov procesorja, se njegove nove pohitritve išče drugje – v večanju števila jeder procesorja. Pri tem z **jedrom procesorja** mislimo na tisto fizično enoto, ki smo jo še prej posplošeno imenovali kar procesor. Jeder v procesorju je sedaj torej več in so med seboj simetrična. To nam omogoča, da se vzporedno izvaja več različnih zaporedij ukazov hkrati, ne da bi za to morali prekinjati njihovo izvajanje v procesorju in preklapljati med njimi.

Razporejanje bremena na različna jedra procesorja je bila prepuščena razvrščevalniku operacijskega sistema. Najenostavnejša rešitev za tako razporejanje, je razporeditev različnih procesov po različnih jedrih. A včasih želimo vsa jedra procesorja izkoristiti za en sam proces. V tem primeru moramo poseči bodisi po vejitvah procesa ali po **nitih**.

Niti so krajša zaporedja ukazov, ki jih operacijski sistem zna razvrščati. Po večini jih ustvarimo zato, da bi lahko izvajali več opravil sočasno. V večini izvedb niti velja, da se množica niti izvaja v sklopu enega procesa. Niti se od procesov razlikujejo v tem, da si vse niti (istega procesa) delijo isti naslovni prostor, medtem ko imajo vsi procesi kljub vejitvam iz istega procesa ločen naslovni prostor. Zaradi tega preklop med nitmi zahteva precej manj režije in je posledično tudi veliko hitrejši. Pri enojedrnih procesorjih gre pri izvajanju več niti zgolj za časovno preklapljanje, medtem ko se lahko na večjedrnih sistemih vsaka nit izvaja na poljubnem prostem jedru. Hitrost preklopa, skupen naslovni prostor in dejstvo, da so niti čista programska rešitev, so glavni razlogi zakaj nekateri vmesniki API za vzporedno obdelavo podatkov raje izkoriščajo niti kot procese.

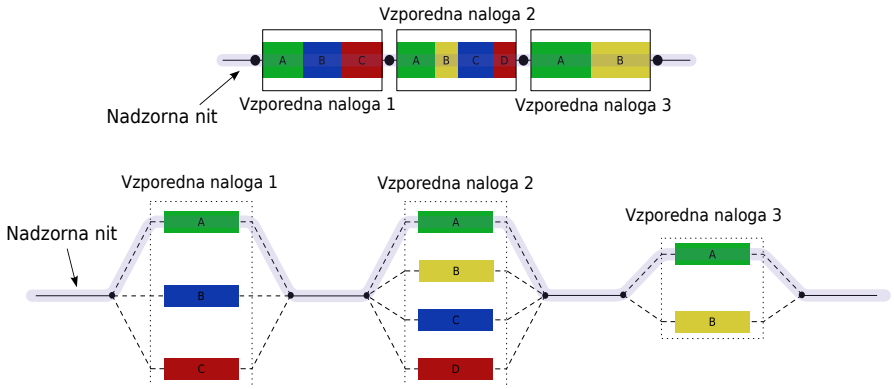
2.1.1 Vmesnik OpenMP

OpenMP[2] je vmesnik API, ki je namenjen izdelavi programov za vzporedno procesiranje na podlagi skupnega pomnilnika, uradno podprt v programskih jezikih *C*, *C++* in *Fortran*.

Prva izvedba specifikacij za vmesnik OpenMP, je bila izdelana oktobra, leta 1997 pod okriljem *OpenMP Architecture Review Board*[3], v katerega je vključena večina glavnih proizvajalcev računalniške strojne in programske opreme. Prva različica specifikacij je bila izdelana le za programski jezik *Fortran*. Nadgrajeni različici 2.0, izdelani leta 2002, pa je bila dodana še podpora za pro-

gramska jezika *C* in *C++*. Leta 2005 so z različico 2.5 poenotili podporo vsem trem programskim jezikom in leta 2008 izdali trenutno zadnjo različico specifikacij – 3.0. Vmesnik OpenMP je danes podprt v vseh večjih in bolj znanih prevajalnikih, se pa podpore različnim različicam specifikacij med seboj razlikujejo. Tako ima Microsoftov prevajalnik *MSVC* še zmeraj podprto le različico 2.0, medtem ko je pri Oraclu, Intlu in drugih že podprta različica 3.0. Pri prenosu izvorne kode na drugo platformo moramo biti na to še posebej pozorni.

Vmesnik OpenMP sledi pri razvoju večnitnosti konceptu vejitvam **nadzorne niti** (slika 2.1). To pomeni, da se v trenutku, ko želimo nek del programa izvajati vzporedno, nadzorna nit razveje na več niti – eno nadzorno in več **delavskih niti**. Največje število niti je določeno s številom logičnih izvajalskih jeder procesorja. Pri večjedrnem procesorju je enako številu fizičnih jeder procesorja, medtem ko je pri procesorjih, ki omogočajo večnitenje, to število odvisno od arhitekture in števila jeder procesorja (tipično dvakratnik števila jeder). Seveda pa lahko to število tekom izvajanja programa poljubno spreminjamo glede na svoje potrebe. Nastavimo ga lahko tudi večje od prej omenjenega največjega, a se moramo pri tem zavedati, da bo število niti v tem primeru še zmeraj enako največjemu številu in nam zato večje številke ne prinesejo nobene dodatne pohitritve.



Slika 2.1: Razporejanje niti vmesnika OpenMP (vir: [2]).

Uporaba večnitnosti v izvorni kodi je precej enostavna. Namesto posebnih funkcij za kreiranje in nadzor niti tukaj dodajamo le direktive prevajalniku, kar še posebej poenostavi uporabo večnitnosti v že obstoječi izvorni kodi. Kot primer lahko navedemo, da smo pri svojem testnem programu morali dodati le dve vrstici (pri čemer je ena od teh samo ukaz za vključitev datoteke glave *omp.h*) in že smo lahko namesto enega jedra procesorja koristili vse štiri. Direktive imajo zmeraj isto začetno sintakso:

#pragma omp *<konstrukt>*

Konstrukti so osnovni elementi vmesnika OpenMP in jih delimo v 5 skupin:

- vzporednost
- delitev dela
- podatkovno okolje
- sinhronizacija
- okolje izvajanja

Zadnje skupine konstruktov, t.j. okolje izvajanja, ne uporabljamo na enak način kot preostalih štirih, saj pri tej skupini ne gre za direktive, ampak za navadne izvajalske funkcije. V programu jih uporabljamo za sprotno nastavljanje vrednosti števila niti (*omp_set_num_threads()*), preverjanje števila nit (*omp_get_num_threads()*), ugotavljanje identifikacijske številke trenutne niti (*omp_get_thread_num()*), itd.

2.2 Vektorski ukazi

Vektorski ukazi so se množično začeli uporabljati v dobi osebnih računalnikov. V procesorje so bili vpeljani za boljšo obdelavo večpredstavnostnih vsebin. Danes ima vektorske ukaze implementirane že vsak moderni procesor.

Ukazi za vektorske operacije se izvajajo po principu SIMD, kjer z enim ukazom izvedemo isto operacijo (npr. seštevanje) nad več zaporednimi podatki, ki si jih predstavljamo kot istoležne komponente vektorja.

Tako lahko štiri ukaze (enačbe 2.2) zamenjamo z enim vektorskim (2.3):

$$\begin{aligned} a[i+0] &= b[i+0] + c[i+0] \\ a[i+1] &= b[i+1] + c[i+1] \\ a[i+2] &= b[i+2] + c[i+2] \\ a[i+3] &= b[i+3] + c[i+3] \end{aligned} \tag{2.2}$$

$$a[i:i+3] = b[i:i+3] + c[i:i+3] \tag{2.3}$$

V spodnji enačbi 2.3 lahko vidimo primer vektorskega ukaza, ki sešteje operande, ki se nahajajo na zaporednih naslovih od i do $i+3$, s samo enim ukazom namesto štirih.

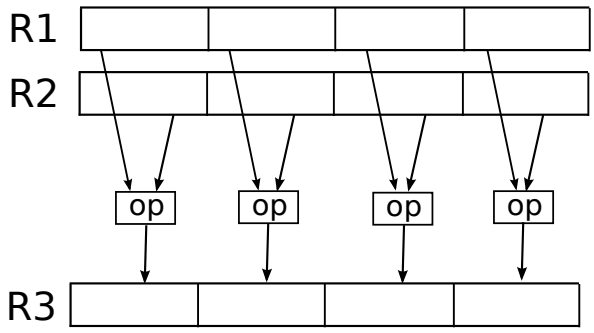
Danes najbolj poznane izvedbe vektorskih ukazov na procesorjih so *SSE (ang. Streaming SIMD Extention)* [4], *AltiVec* [5], *VIS (ang. Visual Instruction Set)* [6] in drugi. Večina teh izvedb si je v osnovi precej podobna. Vse delujejo po principu, kjer spremenljivke najprej prenesemo iz pomnilnika v posebne registre procesorja in jih po končanem izvajanju operacij shranimo nazaj v glavni pomnilnik. Za najhitrejše računanje operaciji nalaganja in shranjevanja uporabimo le, ko je to res nujno – načeloma samo na začetku in koncu algoritma.

2.2.1 Ukazi SSE

Leta 1999 je družba Intel za svoje procesorje z naborom ukazov serije *x86* vpeljala dodaten nabor ukazov za vektorske operacije, t.j. ukaze SSE [4]. Izvajanje novih ukazov je omogočalo osem 128-bitnih registrov. Večina ukazov deluje tako (slika 2.2), da se operacija izvede nad operandi dveh registrov (registra $R1$ in $R2$), rezultat pa se shrani v tretjega (register $R3$).

Prva izvedba ukazov SSE je omogočala predvsem računanje in manipulacijo števil v plavajoči vejici z enojno natančnostjo. Kasnejše izvedbe ukazov SSE so nadgradili še z dopolnjenimi ukazi za delo s celimi števili in z ukazi za delo s števili v plavajoči vejici z dvojno natančnostjo.

Vsak od omenjenih osmih registrov nam danes omogoča hranjenje in vzporedno izvedbo iste operacije za:



Slika 2.2: Izvajanje vektorske operacije

- 16 8-bitnih spremenljivk,
- 8 16-bitnih spremenljivk,
- 4 32-bitne spremenljivke, in
- 2 64-bitni spremenljivki.

A kot je bilo že omenjeno, prva izvedba ukazov SSE podpira zgolj hranjenje in operacije za vzporedno računanje s štirimi 32-bitnimi števili v plavajoči vejici z enojno natančnostjo.

Ukazi SSE prve izvedbe so razdeljeni v 11 skupin:

- Ukazi za aritmetične operacije
- Ukazi za logične operacije
- Ukazi za primerjalne operacije
- Ukazi za pretvorbo spremenljivk
- Ukazi za nalaganje spremenljivk
- Ukazi za postavljanje vrednosti spremenljivk
- Ukazi za shranjevanje spremenljivk
- Ukazi za podporo predpomnjenju

- Ukazi za operacije s celimi števili
- Ukazi za branje in pisanje vrednosti registrov
- Ukazi za druge operacije

Sicer je očitno, da obstaja celotna skupina ukazov za operacije s celimi števili, a nobena od teh operacij ne omogoča splošnega računanja, t.j. seštevanja, množenja in deljenja.

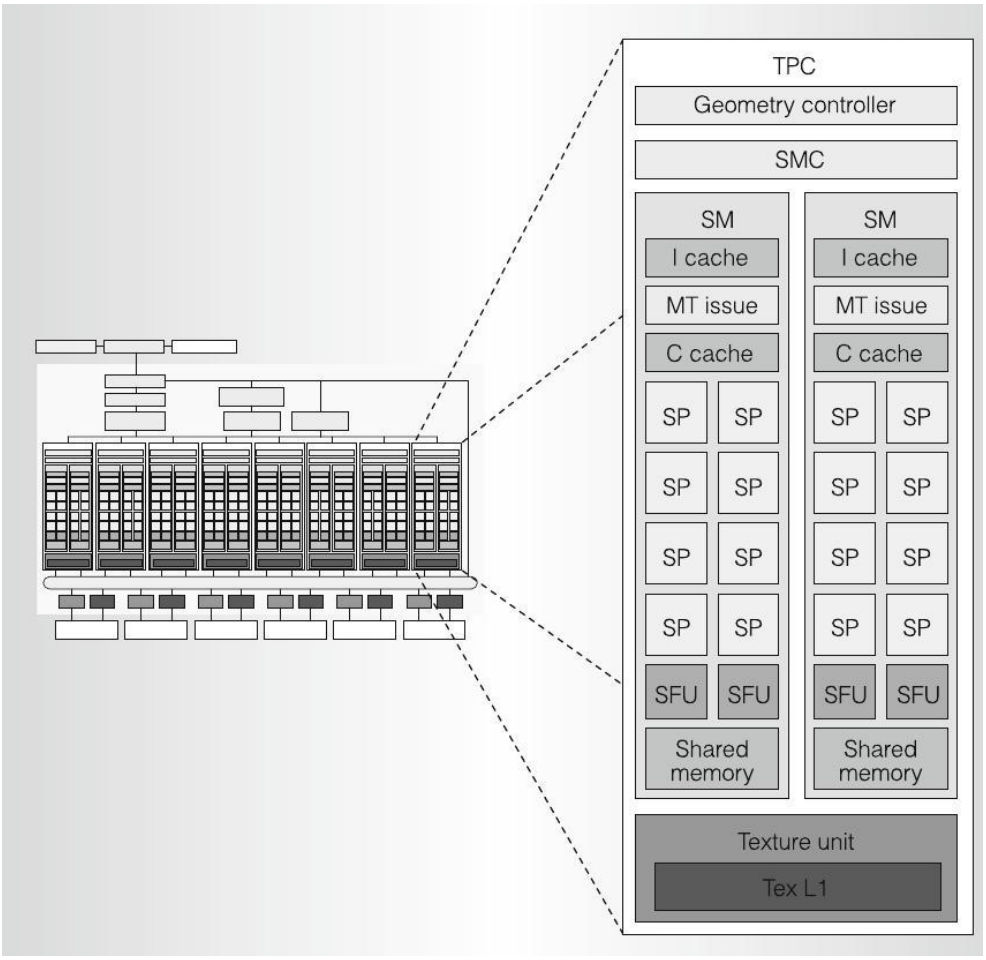
Ukazi SSE so implementirani v vseh večjih prevajalnikih kot *intrisične funkcije* [7]. Ker so imena intrisičnih funkcij v teh prevajalnikih povzeta po izvedbi Intelovega prevajalnika, je izvorno kodo možno prevesti v mnogih razvojnih okoljih, ne da bi jo morali spreminjati. S tem se programerju omogoči enostavno in posledično tudi hitro vpeljavo teh funkcij v že obstoječe pa tudi v nove algoritme.

2.3 Splošno-namensko računanje s procesno enoto grafične kartice

Splošno-namensko računanje s procesno enoto grafične kartice, ki ga na kratko označujemo s kratico *GPGPU (ang. General-Purpose computing on Graphics Processor Unit)*, je tehnika, pri kateri uporabimo procesor grafične kartice za izračun funkcij, ki so običajno prepuščene računalniškemu procesorju. Uporabo grafičnih kartic kot splošno-namenskih računskih strojev je omogočilo dodajanje programirljivih stopenj v cevovod za izrisovanje grafične kartice. Programirljive stopnje cevovoda omogočajo programerju, da procesor grafične kartice izkorišča za poljubne namene računanja.

Čeprav gre tudi pri cevovodu grafične kartice za podoben koncept cevovoda kot pri procesorju, so izvedbe posameznih stopenj obeh cevovodov zaradi narave računskih nalog, ki jih opravljata, dokaj različne. Če ima procesorski cevovod v izvajalski stopnji le nekaj (manj kot deset) funkcijskih enot, namenjenih računanju, se pri cevovodu procesorja grafične kartice uporablja veliko enakih računskih funkcijskih enot (slika 2.3) [8]. Takšno računsko enoto imenujemo pretočni procesor (ang. *streaming processor - SP*) in vsebuje eno enoto za množenje-seštevanje *MAD (ang. multiply-add)* in eno aritmetično-logično enoto. Več pretočnih procesorjev je (skupaj s še drugimi funkcijskimi enotami

in pomnilniki) združenih v en pretočni multiprocesor (ang. *streaming multiprocessor - SM*). Več teh je nato združenih v eno teksturno-procesorsko gručo (ang. *texture/processor cluster - TPC*) in več teh med seboj povezanih enot nato predstavlja obdelovalno enoto procesorja grafične kartice.



Slika 2.3: Razporeditev računskih enot v grafičnem cevovodu (vir: [8]).

Arhitektura je zasnovana tako, da jo lahko kar najbolje izkoriščamo za **pretočno procesiranje**. Pri pretočnem procesiranju gre za programersko paradigmo, pri kateri več enakih funkcijskih enot uporabimo za vzporedno procesiranje. Tak način izrabe funkcijskih enot obeta precejšno pohitritev računanja, a le v podatkovno zasnovanem modelu računanja. V mislih imamo predvsem digitalno procesiranje signalov, obdelovanje slik ali videa, oziroma

vse naloge, kjer imamo opravka z veliko podatki, nad katerimi izvajamo iste operacije. Po drugi strani pa algoritme sprehajanja po drevesnih podatkovnih strukturah ali kakšnih drugih algoritmih, kjer imamo veliko naključnih dostopov do pomnilnika, tak način procesiranja povsem zgrešen. Zato od algoritmov za pretočno procesiranje zahtevamo naslednje tri lastnosti:

- **računsko intenzivnost**, ki zahteva, da je aritmetično-logičnih operacij vsaj petdesetkrat več kot dostopov do glavnega pomnilnika ali celo vhodno-izhodnih operacij;
- **podatkovno vzporednost**, ki izhaja iz vzporedno-nitnega načina procesiranja, saj je potrebno, da se vsi podatki, nad katerimi izvedemo operacijo, nahajajo na zaporednih naslovih pomnilnika;
- **podatkovno lokalnost**, ki je posebna oblika časovne lokalnosti, kjer predpostavljamo, da bomo nek podatek samo enkrat spremenili, ga še enkrat ali dvakrat prebrali in nato tega podatka ne bomo več potrebovali

Ob vsem povedanem, pa ne smemo pozabit še na eno pomembno stvar arhitekture grafične kartice, t.j. zgradbo pomnilnika. Pomnilniško arhitekturo grafične kartice sestavlja več različnih pomnilnikov:

- glavni pomnilnik,
- lokalni pomnilnik,
- pomnilnik za teksture, in
- pomnilnik za konstantna števila.

Če smo pri procesorju imeli pomnilniško hierarhijo, kjer je vsaka stopnja v hierarhiji predstavljala podmnožico sebi nadrejene stopnje, je tukaj pomnilniška arhitektura grafične kartice bolj zapletena. Glavni pomnilnik ima dejansko dva pod-nivoja predpomnilnika, zato da zmanjša zakasnitve pri prenosu podatkov do računskih enot. Istega pa ne moremo trditi za preostale naštete pomnilnike. Vsak od njih je v bistvu poseben pomnilnik in ni vključen v nobeno pomnilniško hierarhijo, saj je vezan direktno na računske enote. Vsi so namreč izredno hitri, zato ne potrebujejo vmesnih predpomnilnikov. Z računskega stališča se to morda zdi rahlonesmiselno, a ne smemo pozabiti, da je grafična kartica v prvi vrsti namenjena delu z grafiko, pri katerem so ti posebni pomnilniki še kako koristni.

Če je iz imen zadnjih dveh pomnilnikov nekako razvidno, da imata posebne funkcije, pa tega ne moremo trditi za lokalni pomnilnik. Samo ime bi nam dalo misliti, da gre za predpomnilnik pod glavnim pomnilnikom, a se o nasprotnem lahko prepričamo v logični shemi pomnilniške arhitekture. Lokalni pomnilnik je resda blizu računskih enot in si ga določene skupine le-teh celo delijo (podobno kot predpomnilnik stopnje 3 pri današnjih procesorjih), vendar pa vsebuje podmnožico podatkov glavnega pomnilnika le ob eksplicitnem kopiranju le-teh iz glavnega pomnilnika. Ta zahteva pa izhaja iz že prej omenjene podatkovne lokalnosti. Ker večine procesiranih podatkov ne uporabljamo večkrat, bi bilo neprestano kopiranje podatkov med glavnim in lokalnim pomnilnikom nespametno. Zato se v lokalni pomnilnik prenašajo samo tisti podatki, za katere vemo, da jih bomo potrebovali večkrat. Seveda pa to ne pomeni, da se lokalnega pomnilnika ne uporablja pogosto. Prav nasprotno, njegovo uporabo se zmeraj priporoča, saj je od glavnega pomnilnika precej hitrejši. Enako velja tudi za preostala posebna pomnilnika, le da je pri njiju podatke potrebno prenesti pred začetkom izvajanja računanja, ker se v času računanja iz njiju lahko le bere. Tako je njuna uporaba zaželena, a zaradi narave njunega delovanja od programerja zahteva nekaj več spretnosti.

Ob tem se moramo tudi zavedati, da podjetja, ki proizvajajo grafične kartice, na različne načine izboljšujejo njihovo zmogljivost, kar pa vodi do tega, da se dejanske arhitekture grafičnih jeder razlikujejo [11]. Če ob tem upoštevamo še dejstvo, da so vse najnovejše rešitve poslovna skrivnost vsakega od podjetij, pridemo do tega, da je dejanska arhitektura precej prikrita. Programerju so tako pri razvoju programa na voljo le logične sheme delovanja jedra. Po eni strani nas to dejstvo pri optimizaciji naših algoritmov precej omejuje, a hkrati prav tak način omogoča poenoteno pisanje kode, ki je prenosljiva med vsemi različnimi arhitekturami, mikrooptimizacije pa so tako prepuščene prevajalnikom in razvojnim orodjem. se povedano pa nosi za seboj še eno posledico. Zaradi različnih arhitektur grafičnih kartic se je pojavilo več različnih orodij in knjižnic, ki bi naj optimalno izkoristila zmogljivosti teh različnih grafičnih kartic. Tako poznamo razvojna orodja *CUDA Toolkit* [12], ki jih je izdelalo podjetje NVIDIA ter razvojna orodja *ATI Stream SDK* [13] podjetja *AMD*. Oboja so namenjena grafičnim karticam posameznega podjetja, a ni posebej vezana na noben operacijski sistem. Prav nasprotno pa lahko trdimo za vmesnik *DirectCompute* [14], ki ga je za svoj operacijski sistem Windows izdelalo podjetje *Microsoft*. Zanimivost zadnjega je, da ni posebej vezan na nobeno arhitekturo grafičnih kartic. Enako pa lahko trdimo za ogrodje *OpenCL* [17], ki si ga bomo podrobneje ogledali v poglavju 2.3.2.

2.3.1 CUDA

CUDA (Compute Unified Device Architecture) je arhitektura namenjena vzporednemu računanju, ki so jo razvili pri podjetju *NVIDIA*. S pomočjo razvojnih orodji programerju omogoča poenoten dostop do računskega jedra grafičnih kartic izdelanih pri istem podjetju. Čeprav so uradna orodja namenjena pisanju programov v programskem jeziku *C*, pa so nam dostopne tudi neuradne ovojnice za druge programske jezike, kot so *Python*, *Java*, *MATLAB* itd.

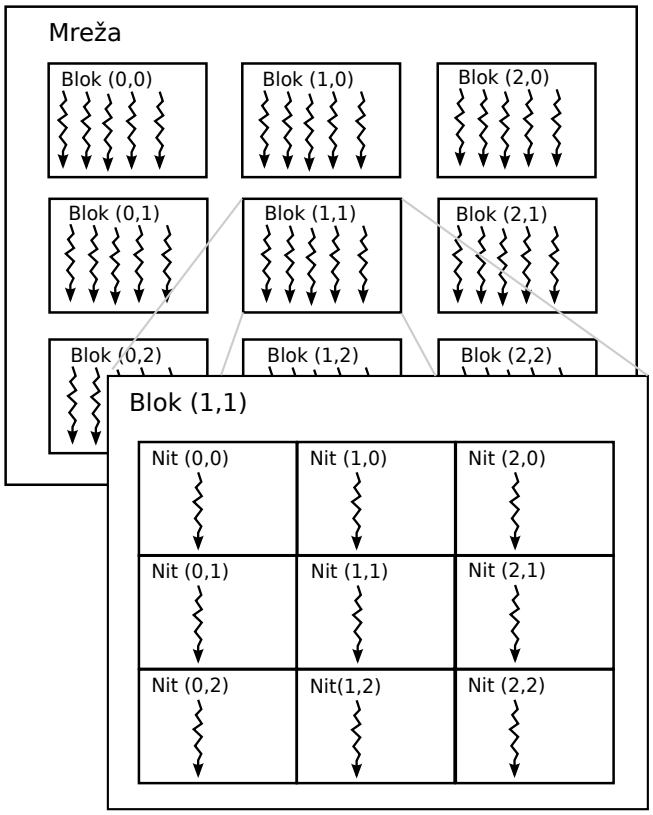
Arhitektura CUDA v svojem jedru vsebuje tri preproste abstrakcije, ki so izpostavljene programerju:

- hierarhija skupin niti,
- deljenje pomnilnikov, in
- pregradna sinhronizacija.

Te abstrakcije omogočajo drobnozrnato podatkovno vzporednost in vzporednost niti, ugnezdeno znotraj grobozrnate podatkovne vzporednosti in vzporednosti računskih opravil. Na tak način se programerju omogoči enostavno deljenje enega zapletenega problema na več manjših in enostavnejših, ki jih tako lahko rešujemo s pomočjo več skupin niti.

Način izvajanja skupin niti so pri podjetju NVIDIA poimenovali *SIMT (ang. Single-instruction, Multiple-Thread)*. Izvajanje niti omogoča pretočni multiprocesor, ki ima večnitnost implementirano na nivoju strojne opreme. Najhitrejše izvajanje dobimo, če izvajamo iste ukaze za večjo skupino niti. Najmanjšo skupino niti, ki jo pretočni multiprocesor zna razvrstiti se imenuje *snop (ang. warp)*. V današnji arhitekturi grafičnih kartic je tak snop sestavljen iz 32 niti. Seveda pa vsaka nit lahko izvaja svoje ukaze. Možna je celo uporaba pogojnih skokov. A je v takem primeru hitrost izvajanja precej počasnejša.

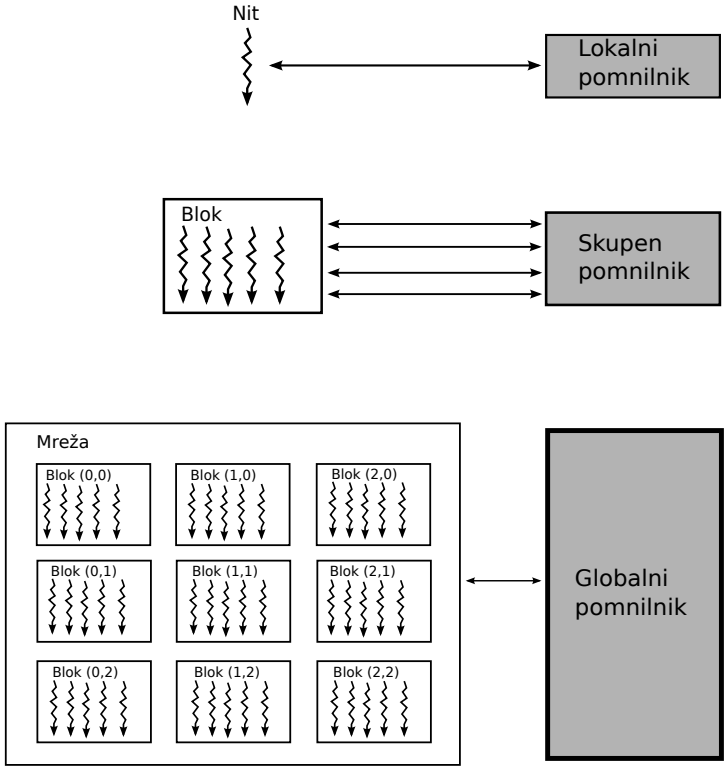
Arhitektura CUDA nam grafično kartico predstavi kot polje z veliko **izvajalskimi nitmi** [16], ki so razdeljene v **bloke** (slika 2.4). Število niti znotraj enega bloka je konstantno in je določeno glede na količino razpoložljivega skupnega pomnilnika. Urejenost niti znotraj bloka si določi programer glede na naravo problema, ki ga rešuje, in je lahko eno, dvo ali tridimenzionalna. Določeno število blokov je naprej urejenih v eno ali dvodimenzionalno **mrežo**. Obseg problema in zapletenost računskega postopka določata tako obe urejenosti kot tudi število uporabljanih niti znotraj blokov ter število uporabljenih blokov.



Slika 2.4: Urejenost izvajanja v arhitekturi CUDA.

Urejenost , oz. število dimenzij uporabljenih za razporejanje blokov v mrežo in niti v bloke, določa tudi način indeksiranja blokov in niti. Na sliki 2.4 se uporablja dvodimenzionalno razporejanje blokov in tudi niti, zato imajo oboji indekse določene kot par (x, y).

Pomemben koncept arhitekture CUDA pa je tudi deljenje pomnilnika na različnih nivojih arhitekture (slika 2.5). V prvi vrsti ima vsaka nit na voljo določeno omejeno količino pomnilnika (in registrov), imenovanega **lokalni pomnilnik**, ki je namenjen zgolj njej sami. Pomnilnik, katerega podatke vidijo vse niti istega bloka, se imenuje **skupen pomnilnik**. Na najvišjem nivoju pa obstaja **globalni pomnilnik**, do katerega lahko dostopajo prav vse niti arhitekture. Vendar pa v nasprotju z logiko pomnilniške arhitekture pri procesorju, tukaj različni nivoji pomnilnikov ne vsebujejo podmnožice podatkov pomnilnikov



Slika 2.5: Deljenje pomnilnika arhitekture CUDA.

na višjih nivojih. Prenos podatkov na nižje nivoje se izvede le na zahtevo programerja. Seveda pri tem ne gre za pomanjkljivo delo inženirjev, pač pa tak koncept izvira iz prej omenjene podatkovne odvisnosti. Problemi, ki jih rešujemo, sproti pogosto spreminjajo večino vrednosti, ki jih računamo, kar pomeni, da se istih podatkov ne potrebuje pogosto – po tem ko smo podatek spremenili ga preberemo kvečjemu dvakrat, nato je že spremenjen. Kljub temu pa je prenos podatkov na nižje nivoje zmeraj zaželen, saj jih s tem prenašamo v hitrejše pomnilnike in s tem povišamo končno zmogljivost programa. Dalje imamo na voljo še dva dodatna pomnilnika – **pomnilnik za teksture** in **pomnilnik za konstante**. Slednja sta tako kot skupen pomnilnik veliko hitrejša od globalnega, zaradi česar se njuno uporabo priporoča. Vendar pa imata dodatno pomanjkljivost. Vanju lahko pišemo le pred začetkom izvajanja naše kode, med samim računanjem pa iz njiju lahko le beremo. Vsi omenjeni

pomnilniki so pomnilniki na **napravi** (grafični kartici), na voljo pa imamo še pomnilnik **gostitelja**, ki dejansko predstavlja glavni pomnilnik računalnika.

Seveda pa moramo biti zaradi vseh teh skupnih pomnilnikov, po koncu izvajanja zelo pazljivi, ko združujemo pridobljene podatke. Zaradi tega se mora programer mora zavedati **pregradne sinhronizacije** niti. Povsod v programu, kjer želimo, da se predhodno asinhrono izvajanje niti zaključi (npr. zato da vemo, da smo prenesli vse trenutne podatke iz globalnega v skupen pomnilnik), vstavimo pregrade - funkcije za sinhronizacijo niti, pri katerih se izvajanje niti ustavi za toliko časa, da to funkcijo dosežejo vse niti. Od te točke v kodi naprej imamo zagotovilo, da so vse niti izvedle vse, kar je bilo zadano.

Model za programiranje za arhitekturo CUDA predvideva, da se niti izvajajo na fizično ločeni napravi, ki deluje kot koprocesor gostitelja, ki izvaja neračunski del programa. V našem primeru to pomeni, da grafična kartica izvaja računska jedra, medtem ko se preostanek programa izvaja na procesorju. Ob tem se tudi predpostavlja, da tako naprava kot tudi gostitelj uporabljata vsak svoj pomnilniški prostor v glavnem pomnilniku (*pomnilnik gostitelja*) in v pomnilniku grafične kartice (*pomnilnik naprave*). Zaradi tega tisti del programa, ki se izvaja na gostitelju, upravlja z naslovnim prostorom na napravi le preko izvajalnika CUDA (*CUDA runtime*). V sklop tega upravljanja spadajo dodeljevanje pomnilnika, sproščanje pomnilnika ter obojestranski prenos podatkov med pomnilnikom naprave in pomnilnikom gostitelja.

Programer ima dostop do arhitekture CUDA s pomočjo knjižnic in datotek v glavi razvojnega okolja na dveh nivojih. Prvi nivo je osnovni in omogoča delo preko vmesnika API za napravo. Višji nivo pa kliče funkcije osnovnega nivoja, zaradi česar je pisanje programov za arhitekturo CUDA precej poenostavljeno. Nič nam pa ne prepričuje izmenične uporabe obeh nivojev znotraj istega programa. Vsaka taka izvorna koda se nato s pomočjo posebnega prevajalnika, prevede v vmesno kodo, ki je prenosljiva med različnimi arhitekturami grafičnih kartic. Komaj ko zaženemo program, gonilnik grafične kartice prevede vmesno kodo v čisto binarno obliko, ki je razumljiva naši specifični grafični kartici. To prevajanje se izvaja po principu *ravno-v-pravem-trenutku (ang. just-in-time).*

2.3.2 OpenCL

OpenCL je standard [17], ki gre še korak dlje od arhitekture CUDA, saj z njim nismo omejeni le na grafične kartice. Omogoča namreč programiranje heterogenih zbirk, sestavljenih iz procesorjev, grafičnih kartic in drugih računskih naprav v sklopu iste platforme. Tako omogoča izdelavo zmogljive in prenosljive izvorne kode.

Prva različica standarda je bila izdelana konec leta 2009 v sklopu združenja *Khronos*, katerega člani so skoraj vsa večja podjetja računalniške industrije [18]. Za razliko od razvojnega okolja za arhitekturo CUDA pa standard definira le vmesnik z datotekami glave ter opise vsebujočih funkcij. Vpeljava knjižnic, prevajalnika, izvajalnika (ang. *runtime*) in ostalih delov ogrodja pa je prepuščena posameznim skupinam, ki želijo omenjeni standard podpreti – podobno, kot to poznamo že pri grafičnem vmesniku *OpenGL* [19], ki je prav tako izdelek skupine *Khronos*. Prenosljivost izvorne kode je tako zagotovljena med vsemi realizacijami standarda OpenCL, hkrati pa je vsakemu posamezniku omogočeno, da izdela knjižnice, ki kar najboljše izkoriščajo njegove komponente. V primeru, ko izvajalnik ugotovi, da se bo računanje izvajalo na procesorjih, izvorno kodo prevede tako, da le-ta uporablja ukaze SSE, kjer je to mogoče in smiselno.

Realizacija standarda OpenCL je razdeljena v štiri modele:

- model platforme
- model izvajanja
- model pomnilnika
- model programiranja

Model platforme sestavlja **gostitelj**, ki je povezan z eno ali več **naprav** OpenCL. Naprava OpenCL se naprej deli v **računske enote**, te pa dalje v **procesne elemente**. Računanje na napravi se izvaja znotraj procesnih elementov. Program OpenCL se izvaja na gostitelju, ki pošilja **ukaze** procesnim elementom v sklopu naprave OpenCL. Model platforme je tako zadolžen za odkrivanje razpoložljivih naprav OpenCL ter ugotavljanje njihovih sposobnosti, ki se lahko izkoriščajo pri prevajanju izvorne kode.

Izvajanje programa OpenCL se dogaja na dveh mestih: **jedra** se izvajajo na

enem ali več napravah OpenCL, medtem ko se **gostiteljski progam** izvaja na gostitelju. Gostiteljski program definira kontekst za jedra ter skrbi za njihovo izvajanje. Kontekst vsebuje sledeče štiri vire:

- **naprave**: zbirka naprav OpenCL, ki jih lahko uporabi gostitelj (npr. grafična kartica, procesor, druga računska naprava).
- **jedra**: funkcija OpenCL, ki se izvaja na napravah OpenCL.
- **programski objekti**: izvorna ter izvajalna koda programa, ki vsebuje jedra.
- **pomnilniški objekti**: množica pomnilniških objektov, ki so vidni tako gostitelju, kot tudi napravam OpenCL. Instance jeder lahko za delo uporabljajo le te objekte za pomnjenje podatkov.

Slika 2.6: Urejenost izvajanja v OpenCL (vir: [17]).

Ko gostitelj pošlje zahtevo za izvajanje jedra, se zanj definira n-dimenzionalen *indeksni prostor* (ang. *NDRange*), kjer je n enak 1, 2 ali 3 (slika 2.6). Instanca jedra se izvede za vsako točko znotraj tega indeksnega prostora. Imenuje se **delovni predmet** (ang. *work-item*) in je prepoznavna po lokalni identifikacijski številki, ki izhaja iz koordinat te točke znotraj indeksnega prostora. Čeprav vsak delovni predmet izvaja enako kodo, pa je lahko izbrana pot izvajanja za vsak delovni predmet različna. Delovni predmeti so dalje združeni v **delovne**

skupine (ang. *work-group*), ki so prepoznavne po identifikacijski številki, ki je enakih dimenzij kot indeksni prostor delovnih predmetov.

Gostitelj ustvari podatkovno strukturo, imenovano **ukazna vrsta**, preko katere nato koordinira izvajanje jeder na napravah znotraj enega konteksta, tako da vanjo dodaja zahteve za izvajanje jeder. Ukazna vrsta skrbi za razvrščanje ukazov za izvajanje na napravi. Ti ukazi se izvajajo asinhrono med gostiteljem in napravo. Relativno en na drugega pa se izvajajo bodisi v **zaporednem** ali **nezaporednem izvajanju**. Pri slednjem se med seboj ne čakajo, zato mora programer ob koncu izvajanja eksplicitno zahtevati sinhronizacijo izvajanja. Zaradi heterogene narave zbirk naprav, nam OpenCL dovoljuje ustvarjanje več ukaznih vrst znotraj enega konteksta, pri čemer pa ne zagotavlja nobenega eksplicitnega mehanizma za sinhronizacijo med njimi.

Delovni predmeti, ki izvajajo jedro, imajo dostop do štirih različnih pomnilniških lokacij:

- **globalnega pomnilnika** – omogoča branje in pisanje vsem delovnim predmetom v vseh delovnih skupinah;
- **pomnilnika konstant** – predel globalnega pomnilnika, ki ostaja tekom izvajanja jedra nespremenjen, pri čemer gostitelj dodeli in inicializira objekte naslovnega prostora v pomnilniku konstant;
- **lokalnega pomnilnika** – pomnilnik, ki je namenjen delovnim predmetom znotraj svoje delovne skupine;
- **privatnega pomnilnika** – pomnilnik, ki je lasten le delovnemu predmetu.

Modela pomnilnika gostitelja in naprave sta medsebojno zmeraj neodvisna, zaradi česar moramo za prenos podatkov med njima uporabiti bodisi kopiranje ali preslikovanje. Oba načina omogočata izbiro med zapornim in nezapornim prenosom podatkov.

Izvajalni model podpira tako **podatkovno-vzporedni** kot tudi **opravilno-vzporedni** model programiranja (pa tudi hibridne izvedbe obeh). Privzeti model je prvi – t.j. podatkovno-vzporedni.

Podatkovno-vzporedni model programiranja predpostavlja računanje, kjer zaporedje ukazov izvedemo na več elementih objekta pomnilnika. S pomočjo

indeksnega prostora je točno določeno, kateri podatek se dodeli kateremu delovnemu predmetu.

Opravilno-vzporedni model programiranja definira model, v katerem je ena sama instanca jedra izvedena neodvisno od kakršnega koli indeksnega prostora. Gre za logični ekvivalent izvajanja enega delovnega predmeta znotraj ene delovne skupine, kjer se vzporednost izvaja tako, da bodisi uporabimo vektorske podatkovne tipe, ki so na voljo na napravi ali dodamo več različnih opravil v ukazno vrsto.

Kljub uporabi različnih izrazov pa je dokaj očitno, da se standard OpenCL precej opira na principe arhitekture CUDA. Nenazadnje je podjetje NVIDIA del združenja Khronos. Zelo hitro lahko potegnemo kar nekaj vzporednic med izrazi v OpenCL in CUDA, ki pa nosijo enak pomen (tabela 2.1).

OpenCL	CUDA
delovni predmet	nit
delovna skupina	blok
indeksni prostor	mreža
globalni pomnilnik	globalni pomnilnik
lokalni pomnilnik	skupen pomnilnik
privatni pomnilnik	lokalni pomnilnik

Tabela 2.1: Primerjava izrazov istega pomena v OpenCL in CUDA.

In vendar se obe rešitvi v določenih podrobnostih razlikujeta. Kot smo že omenili, OpenCL podpira različne tipe naprav, medtem ko smo pri arhitekturi CUDA omejeni le na grafične kartice. Pomembna razlika se pojavi tudi pri prevajanju, kjer se pri prvem izvorna koda prevede ob zagonu progama, medtem ko obstaja pri drugi rešitvi dvostopenjsko prevajanje: najprej v vmesno in šele nato v izvajalsko kodo. Druge razlike so razvidne zgolj pri sintaksi programiranja. Te podrobnosti programerju precej olajšajo delo, saj ne zahtevajo dodatnega poglobljenega učenja novih tehnologij in ob tem omogočajo tudi enostaven prenos izvorne kode med obema rešitvama.

Poglavje 3

Simulacija N-teles

Simulacija N-teles je numerična aproksimacija razvoja sistema teles, v katerem vsako telo tega sistema vpliva na vsa ostala telesa sistema. Najbližji realni primer takega sistema najdemo v astrofiziki, kjer vsako telo predstavlja eno zvezdo v galaksiji in se vse zvezde med seboj ves čas privlačijo zaradi gravitacijske sile. Seveda se simulacija N-teles lahko uporablja tudi na drugih naravoslovnih področjih, predvsem v fiziki, kemiji in farmaciji [21].

Ker simulacija N-teles računa sile medsebojnega privlaka, je odličen kandidat za preizkus zmogljivosti naprav za vzporedno računanje. Hkrati pa za ta problem obstaja tudi *Barnes-Hutov algoritem* [22], s katerim se časovna zahtevnost algoritma dramatično zmanjša iz $O(n^2)$ na $O(nlog(n))$, kar nam omogoča analizo smiselnosti vpeljave visoko-vzporednih algoritmov glede na napredne zaporedne algoritme.

3.1 Opis problema

V diplomskem delu je realizirana simulacija N-teles za problem gibanja zvezd. Opravka imamo z računanjem vpliva gravitacijske sile med dvema telesoma za vse možne pare sistema zvezd, ki ga opazujemo. Torej lahko za izračun gravitacijske sile uporabimo Newtonov zakon univerzalne gravitacije (enačba 3.1) [20].

$$\vec{F}_{ij} = G\frac{m_i m_j}{\|\vec{r}_{ij}\|^2} \cdot \frac{\vec{r}_{ij}}{\|\vec{r}_{ij}\|} \tag{3.1}$$

F_{ij} predstavlja magnitudo gravitacijske sile, ki deluje na izbrano telo zaradi privlačnosti med obema opazovanima telesoma i in j. G je gravitacijska

konstanta, m_i in m_j masi obeh opazovanih teles, r pa predstavlja razdaljo med obema telesoma. V našem primeru so pozicije teles določene s krajevnimi vektorji $\vec{x}_i$, zato razdaljo izračunamo tako, da odštejemo vektor izbranega telesa od vektorja drugega telesa (enačba 3.2).

$$\vec{r}_{ij} = \vec{x}_j - \vec{x}_i \tag{3.2}$$

Tako izračunamo vpliv sile le enega telesa na izbrano telo. Da bi izračunali vpliv vseh sil, pa moramo to silo izračunati za vse možne pare za izbrano telo in jih sproti seštevati (enačba 3.3).

$$\vec{F}_i = \sum_{j=1, j\neq i}^{N} \vec{F}_{ij} = G m_i \sum_{j=1, j\neq i}^{N} \frac{m_j \vec{r}_{ij}}{\|\vec{r}_{ij}\|^3} \tag{3.3}$$

Da bi se pri računanju ognili singularnostim (deljenju z 0), do katerih prihaja pri trku dveh teles, saj je razdalja med njima takrat enaka 0, uvedemo nek majhen *faktor ublažitve* $\epsilon^2 > 0$ [20]. Hkrati s tem se ognemo pogoju $j \neq i$ (enačba 3.4).

$$\vec{F}_i \approx G m_i \sum_{j=1}^{N} \frac{m_j \vec{r}_{ij}}{\left(\|\vec{r}_{ij}\|^2 + \epsilon^2\right)^{\frac{3}{2}}} \tag{3.4}$$

Tekom izvajanja simulacije spreminjamo pozicijo teles zaradi trenutne hitrosti telesa, ki je opisana z vektorjem hitrosti v prostoru. Izračunan vpliv vsote gravitacijskih sil na telo povzroči spremembo trenutne hitrosti. Če smo natančni, spremembo hitrosti povzroči pospešek, ki ga lahko izračunamo kot $\vec{a}_i = \vec{F}_i / m_i$, kar nam poenostavi enačbo 3.5, tako da računamo samo pospešek.

$$\vec{a}_i \approx G \sum_{j=1}^{N} \frac{m_j \vec{r}_{ij}}{\left(\|\vec{r}_{ij}\|^2 + \epsilon^2\right)^{\frac{3}{2}}} \tag{3.5}$$

Po izračunanem pospešku izračunamo spremembo hitrosti in pozicije s pomočjo osnovnih enačb za enakomerno pospešeno gibanje (enačbi 3.6 in 3.7) [10].

$$x_{n+1} = x_n + v_n \delta t + \frac{a_{n+1} \delta t^2}{2} \tag{3.6}$$

$$v_{n+1} = v_n + a_{n+1} \delta t \tag{3.7}$$

3.2 Algoritem za vzporedno računanje

Izpeljava vzporednega algoritma iz zgoraj navedenih enačb je trivialna. Vemo, da računamo medsebojni privlak vseh teles, je takoj jasno, da se bo algoritem izvajal znotraj ugnezdene zanke (algoritem 1). Glede na napravo, ki je namenjena temu računanju, in število vzporednih tokov računanja, ki nam ga ta naprava omogoča, se lahko odločimo ali bomo vzporedno računali le končne pospeške za vsako telo (zunanja zanka) ali pa kar vsak pospešek vsakega para teles (notranja zanka).

Algoritem 1 Osnovni algoritem N-teles

for $i = 1$ *to* N **do**
  $vsota = 0$
  for $j = 1$ *to* N **do**
    izračunaj pospesek
    $vsota = vsota + pospesek$
  end for
  izračunaj novo pozicijo in hitrost telesa na podlagi vsote pospeškov
end for

Iz enačb 3.2 in 3.5 lahko izpeljemo postopek za računanje pospeška.

Algoritem 2 Izračun pospeška

$r = x[j] - x[i]$
$f = 1.0/sqrt(r * r + eps * eps)$
$da = m[j] * f * f * f$
$a = a + da$

Imena spremenljivk so enaka kot v enačbah 3.5, 3.6 in 3.7). Namesto grške črke δ je uporabljena predpona *d*.

Na koncu izpeljemo iz enačb 3.6 in 3.7 še postopek za popravljanje trenutne pozicije ter hitrosti (algoritem 3). Edini problem, ki ga moramo tukaj rešiti, je hranjenje trenutnih in spremenjenih pozicij in hitrosti teles hkrati. S tem zagotovimo, da je računanje novega pospeška neodvisno od računanja pospeškov vseh preostalih parov teles.

Ponovno pomislimo nazaj na Newtnov zakon univerzalne gravitacije (enačba 3.1) in se na podlagi tretjega Newtonovega zakona dinamike spomnimo še, da

Algoritem 3 Izračun nove pozicije in hitrosti telesa

$nov\_x[i] = x[i] + v[i] * dt + 1/2 * dt * dt * a$
$nov\_v[i] = v[i] + dt * a$

izračunana sila deluje vzajemno na obe telesi. To pomeni, da pri računanju medsebojnega privlaka računamo isto silo oziroma isti pospešek dvakrat. Če bi enkrat izračunan pospešek takoj prišteli k obema telesoma opazovanega para, bi s tem razpolovili število potrebnih izračunov pospeška. Vendar pa se moramo zavedati, da si s tako zasnovo vsi vzporedni računski tokovi podatke mesebojno spreminjajo med seboj. To dejstvo pa nas sili k uvedbi dodatnega kontrolnega mehanizma sinhronizacije, ki bi omogočal nadzorovano pisanje. S tem pa se zmogljivost programa ohromi veliko bolj, kot bi bilo za zmogljivost algoritma koristno[9].

Opisani algoritem zato nadgradimo v algoritem, ki omogoča vzporedno računanje tako, da v programskem jeziku C dodamo direktivo vmesnika OpenMP nad vrstico zunanje zanke:

```
#pragma omp parallel for private(j, a, r, f, da)
```

Tako navedemo, da želimo vzporedno izvajati zunanjo zanko, pri čemer so spremenljivke *j, a, r, f* in *da* lastne vsaki niti računanja, medtem ko so preostale spremenljivke med njimi deljene.

3.2.1 Testni program nbody

Z namenom preizkušanja algoritma za vzporedno računanje je bil narejen testni program *nbody*. Programiran je v programskem jeziku *C*, prevedemo pa ga lahko s prevajalniki *GCC*, *ICC*[1] ter *MSVC* tako za operacijski sistem *Microsoft Windows* kot tudi za operacijski sistem, ki je osnovan okrog jedra *Linux*. Program se zaganja iz ukazne vrstice in omogoča nastavljanje različnih parametrov okolja simulacije, računanja in prikaza. Uporaba programa nbody je podrobneje opisana v dodatku A.1.

V sklopu nastavitev za okolje simulacije nam program dovoljuje spreminjanje privzetih lastnosti simulacije. Tukaj mislimo na število teles v simulaciji,

[1]Zaradi hrošča v prvi različici standarda OpenCL, se program s prevajalnikom ICC da prevesti, samo če pri prevajanju ne definiramo spremenljivke 'OPENCL'

njihovo največjo dovoljeno maso in hitrost. Privzeto je, da se v sklopu vsake simulacije telesa na začetku naključno razvrsti znotraj definiranega prostora in se jim ob tem določi tudi naključne hitrosti ter mase. Na željo uporabnika pa se lahko namesto tega uporabi vhodno datoteko, ki navaja pozicije, mase ter hitrosti vseh delcev, ki jih želimo simulirati. Sintaksa datoteke je sledeča:

- **Komentar** je vsaka vrstica, ki ima '#' kot prvi znak.
- Takoj za komentarji in praznimi vrsticami mora biti navedeno **število delcev**, ki jih bomo simulirali. To število se lahko razlikuje od dejanskega števila kasneje opisanih teles. Če je število večje, se določi programu več pomnilnika, kot ga dejansko potrebuje, se prazen prostor ne upošteva pri sami simulaciji in računanju. Če je število manjše, se za računanje uporabi samo prvih n opisov teles.
- **Opis telesa** mora v eni vrstici navesti vse omenjene parametre telesa z racionalnimi števili:
  - vse tri prostorske komponente vektorja pozicije telesa,
  - maso telesa, in
  - vse tri prostorske komponente vektorja hitrosti telesa.

Primer take datoteke za dve telesi:

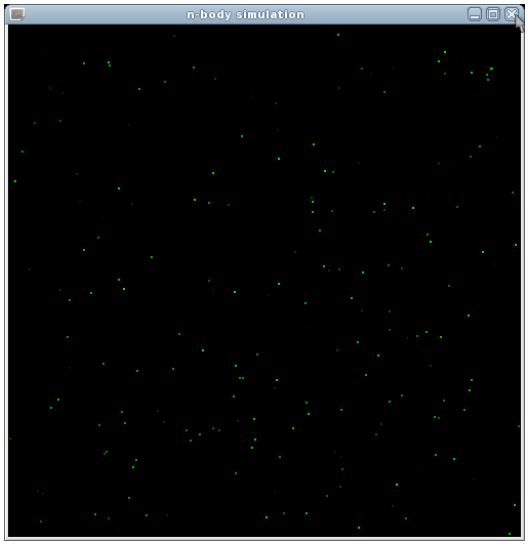
```
2
#x    y      z   m    vx  vy   vz
0.2   -0.1   0.5 0.85 1.3 0.5  3.2
-0.6  -0.04 -0.1 0.12 0.5 0.1 -0.79
```

V sklopu nastavitev za računanje lahko izberemo način računanja, kjer lahko izbiramo med:

- navadnim vzporednim računanjem s pomočjo vmesnika OpenMP,
- vzporednim računanjem z vmesnikom OpenMP in vektorskimi ukazi SSE,
- vzporednim računanjem na grafični kartici s pomočjo knjižnic za arhitekturo CUDA, in
- vzporednim računanjem na procesorju ali grafični kartici s pomočjo knjižnic za ogrodje OpenCL.

Prva dva načina se izvajata izključno na procesorju, tretji izključno na grafični kartici, medtem ko lahko pri četrtem izbiramo med obema. Zaradi tega nam program daje možnost tipa naprave, s katero bomo računali. V primeru, da imamo v sistemu več naprav, si lahko ogledamo njihov seznam in med njimi poljubno izberemo. Nastavljivo je tudi število vzporednih računskih tokov, kar nam pride prav, ko želimo uporabljati omejeno število jeder večjedrnega procesorja, ali ko je jedro preobsežno za izračun v eni niti arhitekture CUDA.

Nenazadnje program omogoča sprotni prikaz poteka simulacije. S pomočjo knjižnice *OpenGL* se odpre dodatno okno (slika 3.1), v katerem se izrisujejo trenutne pozicije teles po vsakem opravljenem izračunu trenutnih pozicij. Seveda pa nam grafični prikaz odžira določen del razpoložljivih virov, kar lahko vodi do anomalij pri opravljanju meritev, zato ga lahko tudi izklopimo.



Slika 3.1: Testni program nbody zagnan v grafičnem načinu.

3.2.2 Optimizacije algoritma

Omenjeni različni načini računanja izkoriščajo različne tehnologije, kar posledično pomeni, da se implementacije algoritma zanje razlikujejo in zaradi tega zahtevajo različne načine optimizacije. Kljub temu pa so si enotne v načinu hranjenja podatkov o telesu.

Vsako telo je predstavljeno s sedmimi spremenljivkami, ki so shranjene v dveh poljih spremenljivk dolžine 4 in so tipa število v plavajoči vejici z enojno natančnostjo. Tako so vse tri komponente prostorskega vektorja pozicije telesa ter masa telesa shranjene v prvem, preostale tri komponente prostorskega vektorja hitrosti pa v drugem polju. Četrta spremenljivka v drugem polju je nedefinirana in se je ne uporablja. Tak način predstavitve spremenljivk izhaja iz uporabe vektorskih tipov spremenljivk, ki jih uporabljajo tehnologije (*_m128* pri SSE, *float4* pri CUDA in OpenCL). Obe polji se nahajata znotraj dveh polj, ki vsebujeta bodisi pozicije ali hitrosti vseh teles.

Kot je razvidno že iz osnovnega algoritma, se nove vrednosti pozicije in hitrosti hranijo drugje kot trenutne vrednosti. To pomeni, da imamo obe omenjeni polji podvojeni – eno, iz katerega beremo, drugo, da vanj pišemo. Vsi načini računanja podpirajo tudi uporabo kazalcev, s čimer se ognemo nepotrebnemu kopiranju podatkov iz polja z novimi vrednostmi v polje s trenutnimi vrednostmi, saj lahko sedaj le zamenjamo kazalce na obeh poljih.

Vpeljava vektorskih ukazov SSE

Pri nadgradnji osnovnega vzporednega algoritma z vektorskimi ukazi SSE pride do prvih sprememb osnovnega algoritma. Do problema pride, ker je masa telesa shranjena skupaj z njegovo pozicijo. Zaradi tega moramo paziti, da vrednosti mase telesa ne spreminjamo med računanjem, pa tudi splošno jo potrebujemo za računanje, zato jo prenesemo v drug vektorski register s pomočjo ukaza *_mm_shuffle* [4], ki omogoča prenos komponent vektorja s poljubno medsebojno zamenjavo komponent. Tukaj uporabimo kopiranje, ki četrto komponento vektorskega registra pozicije prenese v vse komponente novega vektorskega registra. Izračun razdalje med vektorjema pozicij dveh opazovanih teles je na tem mestu trivialen. A vendar ne smemo pozabiti, da je vrednost mase še zmeraj četrta komponenta vektorja pozicije. Za izračun vsote kvadratov komponent razdalje potrebujemo kopijo že obstoječega vektorskega registra razdalje, a smo tu še dodatno premeteni in to kopiranje izvedemo s pomočjo maske, ki vrednost mase v prekopiranem vektorskem registru postavi na nič. Tako se pri kvadriranju (množenju obeh registrov) vrednost mase izniči. Seveda pa vseh težav s tem še nismo rešili. Največji izziv predstavlja seštevanje kvadratov, da bi dobili vsoto. Ker se vse vrednosti nahajajo znotraj enega vektorskega registra, zanj žal ni posebnega ukaza, saj se lahko ukazi SSE izvajajo le med različnimi vektorskimi registri. Ponovno posežemo po ukazu *_mm_shuffle* in dvojno kopiranje komponent izvedemo tako, da imamo v vsa-

kem vektorskem registru komponente na različnih mestih. Sedaj lahko med njimi izvedemo navadno seštevanje in stvar je rešena. Vse ostale ukaze nadomestimo z ekvivalentnimi ukazi SSE.

Optimizacija pri računanju s procesorjem grafične kartice

Pri prehodu na računanje z grafično kartico iščemo možnosti optimizacije povsem drugje. Zaradi drugačne zasnove pomnilniške hierarhije, kjer nimamo več samo enega pomnilnika, vsebovanega v podmnožicah predpomnilnikov, mora programer ves čas skrbeti, kako in kdaj najbolj uporabljane podatke prenesti v pomnilnik, ki je čim bližje nitim oz. delovnim predmetom. Vsaka nit je tukaj zadolžena za izračun spremembe pozicije enega telesa glede na vse možne pare teles z njim. Zato si na začetku izvajanja jedra naloži v lokalni pomnilnik podatke o svojem telesu. Ker obdeluje vse možne pare glede na to telo, si preostala telesa ves čas pobira iz glavnega pomnilnika. Skupine niti so združene v bloke in vse izvajajo isto nalogo, zato se odločimo, da bodo vse niti znotraj istega bloka v nekem trenutku obdelovale isto podmnožico teles glede na njihovo opazovano telo. To lastnost izkoristimo za to, da si v nekem trenutku en blok niti v skupen pomnilnik prenese določeno podmnožico podatkov o preostalih telesih. Ko vse niti izračunajo vse možne kombinacije parov za podatke v skupnem pomnilniku, si prenesejo novo podmnožico podatkov. S tem občutno zmanjšamo število prenosov podatkov iz počasnega glavnega pomnilnika.

Druga pohitritev računanja na grafični kartici se posveča pogojnim skokom. Zaradi narave izračunov, ki jih izvajajo, grafične kartice ne premorejo napredne enote za napovedovanje skokov, zaradi česar so le-ti zelo nezaželeni. Pogojni skoki dodatno poslabšujejo situacijo, saj izvajanje v nitih zahteva, da vse niti v bloku izvajajo isti ukaz, česar pa pri pogojnih skokih ne moremo zagotoviti. Zato pri pogojnih skokih, do katerega pride v našem primeru na koncu obhoda zanke, uporabimo trik, ki ga zelo dobro poznamo iz prevajanja izvorne kode za procesorje – t.i. razvoj zank. Naš algoritem zagotavlja razvoj zanke za 0, 4, 8 ali 16 obhodov.

Tretjo pohitritev deluje le, ko simulacije ne poganjamo v grafičnem načinu. V tem primeru ne izvajamo kopiranja podatkov iz globalnega pomnilnika grafične kartice v glavni pomnilnik računalnika. Podatke v slednjem potrebujemo zato, da jih lahko uporabimo za izris s knjižnicami OpenGL.

Zadnja mikrooptimizacija algoritma izhaja iz dejstva, da ko kličemo funkcijo za računanje, moramo zmeraj podati vse vrednosti spremenljivk, ki jih potrebujemo za računanje. Ker so med njimi konstante, jih namesto kot vhodne parametre funkcije uporabimo v kodi kot spremenljivke, katerih vrednosti dobimo že ob prevajanju.

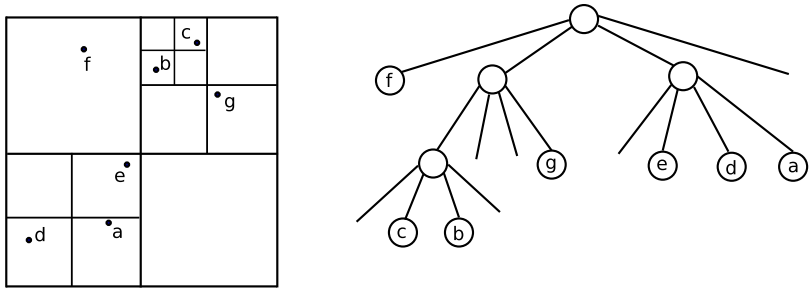
Popolnoma enake pohitritve algoritma so uporabljene tudi za računanje po standardu OpenCL, a je pa med njimi zanimiva razlika. Ker so pri standardu OpenCL vse pomnilne strukture objekti pomnilnika, enako velja tudi za prostor v skupnem pomnilniku. Pri standardu OpenCL moramo tako ustvariti ta poseben objekt skupnega pomnilnika in ga predati instanci jedra kot dodatno spremenljivko, medtem ko je v arhitekturi CUDA skupen pomnilnik predstavljen kot poseben tip spremenljivke, ki se po potrebi dodeli na začetku izvajanja jedra. Zaradi tega imamo ob začetku izvajanja jedra prenos ene spremenljivke manj. Dodatna zanimiva razlika med obema je tudi vpeljava konstant. Pri standardu OpenCL vrednosti konstant določimo kot definicije prevajalnika, medtem ko nam arhitektura CUDA dovoljuje uporabo datoteke v glavi, ki smo jo uporabljali v preostalem delu programa.

3.3 Barnes-Hutov algoritem

V želji, da bi lahko realno ocenili učinek prehoda na vzporedno računanje, je to oceno potrebno podati glede na dejanske hitre rešitve iste simulacije z naprednim algoritmom za zaporedno računanje. Napredni algoritmi za zaporedno računanje zmanjšujejo časovno zahtevnost osnovnih algoritmov, kar postane najbolj očitno šele pri večjem obsegu problema, ki ga z njimi rešujemo.

Eden izmed bolj znanih naprednih algoritmov za zaporedno računanje simulacije N-teles, je Barnes-Hutov algoritem [23], ki sta ga leta 1986 definirala Josh Barnes in Piet Hut. Algoritem uspešno zmanjša časovno zahtevnost reševanja problema na $O(nlog(n))$, kar je v primerjavi s časovno zahtevnostjo osnovnega algoritma $O(n^2)$, precejšna pohitritev.

Algoritem zmanjša število operacij s tem, da skupine bolj oddaljenih teles predstavi kot eno telo s skupno maso in pozicijo. Algoritem hrani telesa in skupine teles v skupni podatkovni strukturi – drevesu (slika 3.2). Zaradi uporabe drevesa algoritem delimo v dve fazi izvajanja: **gradnja drevesa** in **sprehod po drevesu**.



Slika 3.2: Razporejanje teles v drevo.

Učinkovitost Barnes-Hutovega algoritma izhaja iz načina združevanja teles, ki so si dovolj blizu. Obstoječo dvodimenzionalno površino, v kateri izvajamo simulacijo, definiramo kot kvadrat. Nato se držimo pravila, da je v enem kvadratu lahko kvečjemu eno telo. V primeru, da se v kvadratu nahaja več kot eno telo, kvadrat razdeli na štiri manjše enake kvadrate in to ponavlja za vsak nastali kvadrat, dokler ni zagotovljeno prej omenjeno pravilo. Iz tega je tudi razvidna struktura drevesa. Prvi velik kvadrat predstavlja koren drevesa, vsak njegov manjši kvadrat pa njegove štiri sinove in tako rekurzivno naprej. Kvadrati, ki ne vsebujejo telesa, so prazni in jih iz praktičnih razlogov ne ustvarimo. Tako ima lahko vsako vozlišče najmanj enega in največ štiri sinove. Ob tem pa vozlišča v drevesu niso samo prazni kazalci na sinove (zunanje vozlišče), ampak se prav ta izkoriščajo za predstavitev sinov v drevesu kot eno telo s povprečno maso in središčno pozicijo (notranje vozlišče).

Iz vsega povedanega lahko izpeljemo pravila za rekurzivno gradnjo drevesa (slika 3.2). Če je trenutno vozlišče v drevesu:

1. **prazno**, vstavi trenutno telo kot zunanje vozlišče;
2. **notranje**, upoštevaj in popravi povprečno maso in pozicijo tega vozlišča (enačbi 3.8 in 3.9), nato ugotovi v kateri manjši kvadrat trenutnega vozlišča ga lahko uvrstimo, in izberi sina trenutnega vozlišča, ki predstavlja ta manjši kvadrat, kot novo trenutno vozlišče;
3. **zunanje**, ustvari kopijo trenutnega vozlišča, kot sina tega vozlišča in označi trenutno vozlišče za notranje.

$$m = m_1 + m_2 \tag{3.8}$$

$$\vec{x} = \frac{\vec{x}_1 m_1 + \vec{x}_2 m_2}{m} \tag{3.9}$$

Kot primer iz slike 3.2 si lahko ogledamo vstavljanje telesa g v drevo. Prvo izbrano vozliššče je koren drevesa. Ker so vsa ostala telesa že v drevesu je koren notranje vozlišče. Popravimo mu povprečno maso in središčno pozicijo, nato ugotovimo, da izberemo drugega sina korena drevesa. To izbrano vozlišče je pravtako notranje, saj ima sina na prvi poziciji (katerega sinova sta telesi b in c). Ponovno izvedemo popravljanje povprečne mase izbranega vozlišča ter središčne pozicije. Ugotovimo tudi, da nadaljujemo s četrtim sinom izbranega vozlišča. Novo izbrano vozlišče je prazno, zato lahko telo g vstavimo na njegovo mesto.

Zgoraj opisani postopek velja za primer, ko so telesa razporejena v dvodimenzionalnem prostoru, to pa zgolj zato, ker ga je lažje opisati in narisati na papir. Seveda vse deluje popolnoma enako, ko delamo v tridimenzionalnem prostoru, le da se tam celoten prostor razdeli v osem manjših kock, zaradi česa ima drevo lahko do osem sinov.

Ko so vsa telesa razporejena v drevo, lahko začnemo z računanjem oz. s fazo sprehoda po drevesu. Kako najti telesa (zunanja vozlišča) v drevesu, za katera računamo novo pozicijo in hitrost – izbrana telesa, ni predpisano in je zato prepuščeno kreativnosti programerja. Za vsako najdeno telo nato ponavljamo isto rekurzivno zaporedje, ki ga začnemo pri korenu drevesa:

1. Če je opazovano vozlišče v drevesu zunanje (in ni trenutno izbrano telo), izračunaj pospešek med telesoma in ga prištej k že izračunanemu pospešku po enačbi 3.5.

2. Če je razmerje med razdaljo s od trenutno izbranega telesa do opazovanega vozlišča in dolžino stranice kocke kocke d, v kateri se opazovano vozlišče nahaja, manjše od parametra θ ($s/d < \theta$), obravnavaj izbrano vozlišče kot zunanje (saj le-to predstavlja vse sinove v poddrevesu) in enako kot prej izračunaj pospešek.

3. Sicer izvajaj celotno proceduro rekurzivno za vse sinove opazovanega vozlišča.

Parameter θ določa, kolikšna mora biti relativna razdalja med telesoma, da je dovolj velika, za uporabo središčnega telesa za izvajanje izračunov. Večinoma je nastavljena na vrednost 0,5. Nižja kot je, večjo relativno razdaljo zahtevamo, s tem pa je potrebno obiskati tudi več vozlišč in obratno. Če je $\theta = 0$, se vozlišča nikoli ne upoštevajo.

Za izračun spremembe pozicije in hitrosti uporabimo izračunane pospeške na isti način kot pri osnovnem algoritmu. Za neprestano izvajanje simulacije obe fazi izmenično.

Realizacija Barnes-Hutovega algoritma pa v našem primeru ni strogo zaporedne narave. Druga faza se namreč izvaja vzporedno s pomočjo vmesnika OpenMP. Vseh osem poddreves korena drevesa se ločeno uporabi za iskanje teles, za katere želimo izračunati pospešek. Resda se tako za vseh osem poddreves nato sprehajamo po celotnem drevesu, a ker gre tukaj zmeraj samo za branje, do kritičnih sekcij ne pride.

3.3.1 Testni program barneshut

Testni program *barneshut* je bil izdelan z namenom testiranja zmogljivosti Barnes-Hutovega algoritma. Narejen je po vzoru testnega programa nbody, zato ima realizirane tudi vse smiselne funkcije tega testnega programa. Prav tako je podprt na enakih operacijskih sistemih in ga je moč prevesti z istimi prevajalniki. Še zadnja podobnost med njima je, da se nove pozicije in hitrosti zapisujejo v polje spremenljivk, kar nam koristi pri izrisovanju pozicij elementov. Uporaba programa barneshut je podrobneje opisana v dodatku A.2.

Če je testni program barneshut navzven skoraj identičen testnemu programu nbody, pa tega ne moremo trditi za najpomembnejši, a uporabniku prikriti del – izvajanje algoritma.

Osnova algoritma predvideva uporabo podatkovne strukture drevesa. Struktura tega drevesa je v vsakem trenutku nepredvidljiva, zato potrebujemo dinamičen način večanja dodeljenega prostora. Pravtako vozlišče sedaj ni več predstavljeno kot polje spremenljivk, ampak kot *posebna struktura vozlišča*, ki hrani podatke o telesu (pozicija, masa in hitrost), o kocki, v kateri se nahaja (koordinate stične točke vseh osmih sorodnih kock ter dolžino stranice kocke), in nenazadnje še osem kazalcev za osem možnih sinov vozlišča. Da bi se ognili neprestanemu dodeljevanju in sproščanju prostora za strukture vozlišča drevesa, uvedemo povezan *seznam prostih struktur*. Ob zagonu programa si tako vnaprej pripravimo $8n$ prostih struktur. Vsakič ko potrebujemo prosto strukturo, vzamemo prvo s seznama, kazalec na njo pa premaknemo na naslednjo. V primeru, da prostih struktur zmanjka, se ustvari $n/2$ dodatnih struktur. Po opravljenem računanju, se drevo podre in neuporabljene strukture so vrnjene nazaj na seznam prostih struktur.

Drevo gradimo po pravilih iz prejšnjega poglavja 3.3. Zunanja vozlišča prepoznamo po tem, da imajo vsi kazalci na sinove vrednost *NULL*. Zaradi iskanja napak je bila strukturi dodana spremenljivka, v kateri hranimo število teles v podvozliščih izbranega vozlišča. Tako lahko zunanja vozlišča prepoznamo tudi po tem, da imajo to vrednost postavljeno na 0.

Ko je drevo zgrajeno, se vseh osem vozlišč pod korenom drevesa preda v vzporedno izvajanje. Sedaj vsaka od izvajalskih poti izvaja dvojno rekurzijo – s prvo išče realna telesa v prostoru (zunanja vozlišča) v njej določenem poddrevesu, z drugo pa za vsako najdeno telo izvede sprehod po celotnem drevesu (od korena naprej), zato da najde vse možne pare po pravilih za računanje iz prejšnjega poglavja 3.3. Vsako najdeno telo s prvo rekurzijo si s seznama prostih struktur vzame eno strukturo, ki jo uporablja za shranjevanje novih vrednosti pozicije in hitrosti. Po končanem računanju, strukturo doda v seznam *razrešenih teles*. Ta se po opravljenem računanju uporabi za ponovno gradnjo drevesa. Tu bi strukturo morda lahko kar direktno vstavili v novo drevo, namesto da posegamo po vmesnem dodajanju v seznam. Vendar bi takšna rešitev lahko pripeljala do večjih zapletov s kritičnimi sekcijami, ko bi v nekem trenutku tudi do osem vzporednih tokov gradilo drevo. Medtem ko bi se čakanje zaradi kritičnih sekcij lahko pojavljalo ob vsakem pisanju v drevo, imamo sedaj le eno tako kritično sekcijo, ko strukturo pripnemo v seznam.

V primeru, da program zaganjamo v grafičnem načinu, se ob shranjevanju v novo strukturo vrednosti pišejo tudi v posebej dodeljeno polje spremenljivk, preko katere je telesa lažje izrisati[2].

Po končani fazi računanja drevesa trenutnih vrednosti ne potrebuje več, zato se vsa njegova vozlišča vrnejo na seznam prostih struktur. Tako nam po uspešnem podiranju drevesa ostaneta samo še seznam prostih struktur in seznam razrešenih struktur. Slednji se uporabi za ponovno gradnjo drevesa, tako da z njega vzamemo vsako strukturo in jo vstavimo v novo drevo. Kar še nismo omenili, je, da je seznam razrešenih struktur ustvarjen, še preden izvedemo prvo računanje in vsebuje telesa z naključnimi vrednostmi.

[2]Prve različice programa so za izris teles uporabljale kar seznam razrešenih teles, ki se je na koncu izkazal celo kot nekoliko hitrejši od dodatnega pisanja v poseben pomnilnik. Ideja o vpeljavi slednjega izhaja predvsem iz prvotne želje, da bi bilo možno združiti oba testna programa v enega, a do tega zaradi časovnih omejitev nikoli ni prišlo.

Poglavje 4

Meritve

Z namenom, da bi izdelane testne programe kar najtemeljiteje preizkusili in izvedli karseda natančne meritve, smo na testni sistem namestil posebno različico okrnjenega operacijskega sistema z jedrom *Linux* [24]. Nameščena programska oprema je bila precej omejena, zato da bi zmanjšali morebiten vpliv na meritve zaradi kake dodatne potrebne režije za razvrščanje procesiranja, ki ga na takem testnem sistemu ne potrebujemo. Vse meritve so bile opravljene v tem okolju, z izjemo meritev opravljenih z računsko kartico *NVIDIA Tesla C1060* [25]. Dodatno je bil v operacijskem sistemu *Windows* opravljen test z grafično kartico *AMD Radeon 5850* [26] z namenom, da bi lahko primerjali vpliv operacijskega sistema pri računanju z grafičnimi karticami.

Oba programa sta bila prevedena tako s prevajalnikom *GCC* [27] kot tudi *ICC* [28], da bi se ognili nenapovedanim anomalijam zaradi morebitnih hroščev v prevajalnikih, deloma pa tudi zato, da smo iz algoritmov, ki tečejo na procesorju, iztisnili dodaten procent zmogljivosti. VNa ta način smo lahko za naš problem podali realno oceno uspešnosti uporabljanih algoritmov.

Posamezno končano računanje pozicij in hitrosti označimo z izrazom *operacija*. Tako rezultate meritev predstavimo s povprečnim številom opravljenih operacij v sekundi. Povprečno število operacij v sekundi dobimo tako, da po vsaki opravljeni operaciji števec povečamo za ena. Po pretečenem času več kot sekundo število operacij delimo s pretečenim časom in tako dobimo število operacij na sekundo, oziroma krajše **op/s**. Da bi dobili natančnejšo povprečno število operacij na sekundo, smo program poganjali več kot minuto in tako dobili več kot šestdeset meritev za isti poskus.

Za vsako napravo smo pri istih nastavitvah za način računanja in istem številu vzporednih tokov računanja izvedli meritve za 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 in 65536 teles, oziroma vse dokler je bilo število operacij na sekundo višje od 1,0.

Meritve smo opravili na sledeči strojni opremi:

- procesor AMD Phenom II x4 955 3.2GHz [29],
- grafična kartica AMD Radeon 5850 [26], in
- računska kartica NVIDIA Tesla C1060 [25].

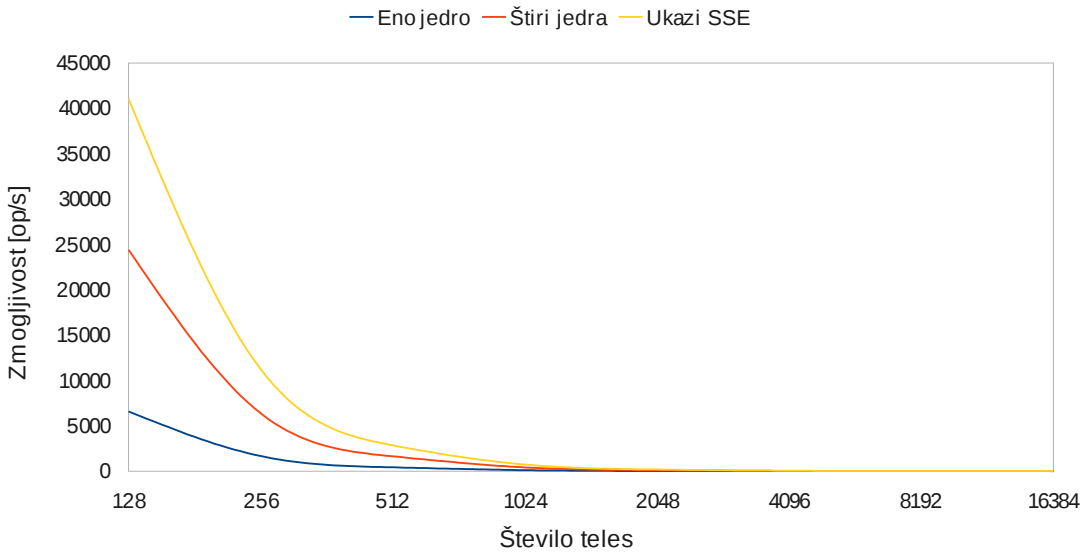
4.1 Meritve zmogljivosti računanja s procesorjem

Najprej smo se želeli prepričati, kako uspešne so različne tehnologije za vzporedno izvajanje na procesorju. V ta namen smo primerjali izvajanje algoritma na enem jedru, na štirih jedrih in na štirih jedrih ob uporabi ukazov SSE. Tabela 4.1 prikazuje primerjavo omenjenih treh načinov računanja, pri čemer številke pomenijo število operacij v sekundi. Navedene so samo vrednosti za testni program, preveden s prevajalnikom ICC, saj je bil v primerjavi s programom, prevedenim s prevajalnikom GCC v povprečju za 70%.

Število teles	Eno jedro [op/s]	Štiri jedra [op/s]	Ukazi SSE [op/s]
128	6580,69	24387,93	41026,61
256	1657,58	6366,89	11215,86
512	417,54	1631,16	2844,53
1024	103,97	409,65	722,87
2048	26,13	102,74	180,18
4096	6,53	25,85	45,13
8192	1,63	6,47	11,18
16384	-	1,68	2,79

Tabela 4.1: Primerjava zmogljivosti algoritmov izvajanih na procesorju.

Iz izmerjenih rezultatov vidimo, da je zaradi uporabe štirih jeder, pohitrimo za 3,7-krat oziroma se faktor pohitritve z višanjem števila teles celo



Slika 4.1: Izmerjena zmogljivost vektorskih ukazov in večjedrnih procesorjev.

približuje magični številki 4 (pri 8192 telesih je faktor enak 3.97). Dodatna vpeljava ukazov SSE sicer ne prinese štirikratne pohitritve, ki bi jo glede na to, da omogočajo izvajanje istega ukaza za štiri podatke hkrati, morda lahko pričakovali. Z uporabo ukazov SSE smo računanje še dodatno pohitrili. V razdelku 3.2.2 smo navedli določene kompromise, ki smo jih bili prisiljeni sprejeti zaradi uporabe ukazov SSE, tako da je faktor dodatne pohitritve (glede na računanje s štirimi jedri procesorja) 1,7 še zmeraj odličen rezultat. Vidimo tudi, da s podvajanjem števila teles zmogljivost pada skoraj za faktor 4, kar pa je pričakovano iz kvadratne časovne zahtevnosti algoritma.

4.2 Meritve zmogljivosti računanja z grafično kartico

Preostala načina računanja, ki ju omogoča testni program, *cuda* in *opencl*, smo uporabili za merjenje računskih zmogljivosti dveh grafičnih kartic. Prva je *AMD Radeon 5850*, ki je predstavnica dražjega segmenta grafičnih kartic za namizne računalnike. Druga kartica je predelana grafična kartica podjetja NVIDIA, imenovana *Tesla C1060*, in je namenjena izključno računanju, zato jo raje imenujemo kar *računska kartica*. Samo slednja je omogočala izvajanje

simulacije v načinu *cuda*, medtem ko smo meritve s prvo kartico opravili samo v načinu *opencl*.

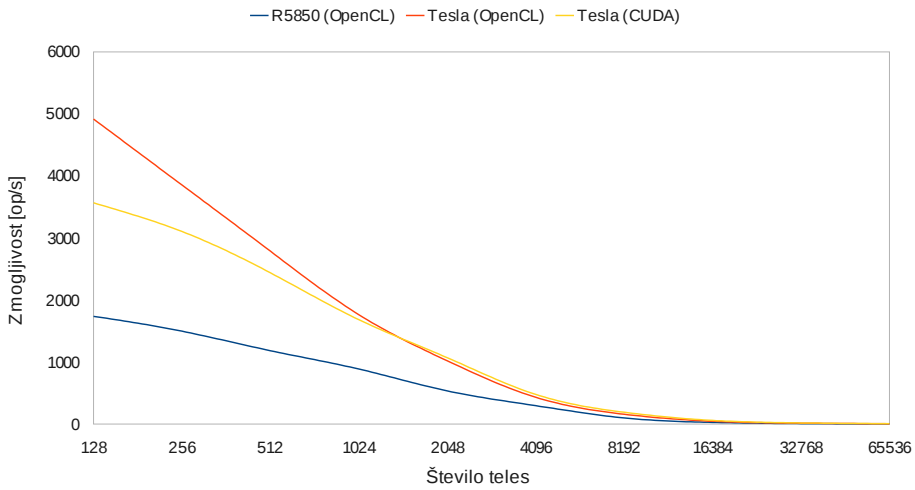
Število teles	OpenCL		CUDA
	Radeon [op/s]	Tesla [op/s]	Tesla [op/s]
128	1736,69	4918,94	3565,28
256	1496,06	3853,92	3104,10
512	1181,44	2786,97	2437,85
1024	885,89	1762,44	1682,12
2048	533,33	1018,51	1068,37
4096	294,98	430,15	476,64
8192	97,20	156,39	188,40
16384	25,36	43,16	57,18
32768	6,49	12,65	16,57
65536	1,75	3,35	4,17

Tabela 4.2: Primerjava zmogljivosti algoritmov izvajanih na grafičnih karticah.

Prva stvar, ki nam pri izmerjenih rezultatih pade v oči je, da zmogljivost ne pada s kvadratom faktorja večanja števila teles kot pri izvajanju algoritmov na procesorjih, kljub temu, da je časovna zahtevnost algoritma ostala $O(n^2)$. Zmogljivost ne pada niti za faktor večanja števila teles, ampak za manj. To anomalijo gre pripisat slabi izkoriščenosti vseh računskih enot, ki jih imata grafični kartici na voljo. Komaj pri številu teles, ki presega vrednost 4096, zmogljivost pada po pričakovanjih.

Hkrati opazimo, da je Tesla pri 128 telesih, v načinu računanja *opencl* za 38% zmogljivejša kot v načinu *cuda*, a je pri največji obremenitvi za 20% manj zmogljiva. Ponovno gre za anomalijo, in sicer zaradi premajhnega obsega problema. Zaradi hitre ponovne vrnitve v funkcijo računanja, procesor precej časa posveti režiji za dostavo vseh ukazov grafični kartici. To lahko potrdimo s tem, da so nihanja zmogljivosti pri teh obsegih problema precejšna, medtem ko pri dovolj velikem številu teles zmogljivost ostaja konstantna v celotnem enominutnem intervalu izvajanja meritev. Iz izmerjenih zmogljivosti vidimo tudi, da je kartica Tesla, v primerjavi s kartico Radeon približno dvakrat zmogljivejša zaradi česar je za Radeon ta mejna vrednost že pri 2048 telesih.

Iz izmerjenih zmogljivosti vidimo tudi, da daje uporaba knjižnic za OpenCL za 20% slabše rezultate kot uporaba knjižnic za arhitekturo CUDA. Težko je



Slika 4.2: Zmogljivosti računanja z grafičnimi karticami.

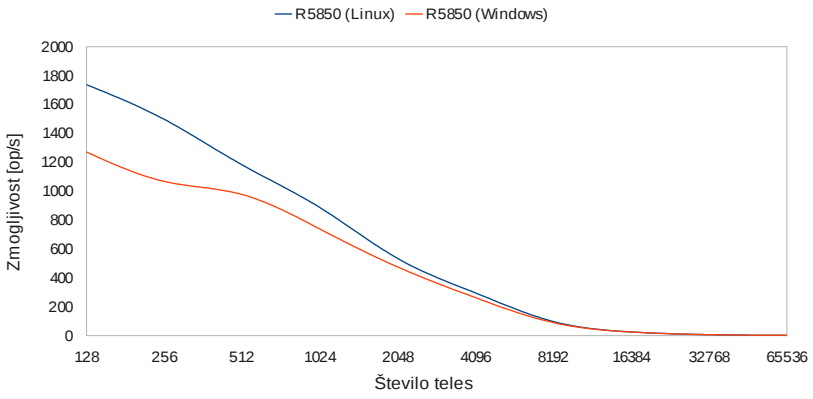
natančno določiti ali gre vzrok iskati v dejstvu, da je arhitektura CUDA že 3 leta dlje na tržišču, je posledično zrelejša in daje zato boljše rezultate, ali pa večji obseg naprav, ki jih lahko vključimo v isto platformo pri knjižnicah za OpenCL, zahteva bolj zapletene postopke, zaradi katerih se več časa porabi za operacije, ki niso namenjene računanju.

Seveda pa pri vsem tem ne smemo spregledati dejstva, da smo meritve za obe kartici izvajali v različnih operacijskih sistemih. Pojavi se vprašanje, koliko so ti rezultati sploh primerljivi. A če dobro razmislimo, delata obe knjižnici za izvajanje operacij na grafični kartici po istem postopku. Izvorna koda programa z algoritmom za računanje se prevede v zbirno kodo za grafično kartico, ki je enaka v vseh operacijskih sistemih. Nato se grafični kartici pošilja zaporedje ukazov iz zbirne kode v predpomnilnik ukazov, medtem ko sama nemoteno izvaja računanje. Iz tega je moč sklepati, da sam operacijski sistem ne bi smel vplivati na zmogljivost izvajanja računov. Omenjeno trditev smo preverili z meritvami v obeh podprtih operacijskih sistemih (tabela 4.3), in sicer z grafično kartico Radeon.

Meritve kažejo, da izbira operacijskega sistema vendarle ni tako nepomembna za zmogljivost računanja grafičnih kartic. Predvsem pri simulacijah, kjer imamo nizko število teles in je potrebno več režije operacijskega sistema, so rezultati slabši za skoraj 40% na operacijskem sistemu *Microsoft Windows*. Po

Število teles	Linux [op/s]	Winodws [op/s]
128	1736,69	1270,44
256	1496,06	1067,09
512	1181,44	976,39
1024	885,89	737,52
2048	533,33	474,44
4096	294,98	262,36
8192	97,20	89,92
16384	25,36	24,62
32768	6,49	6,39
65536	1,75	1,74

Tabela 4.3: Primerjava zmogljivosti algoritmov v različnih operacijskih sistemih.



Slika 4.3: Izmerjena zmogljivost vektorskih in večjedrnih procesorjev.

drugi strani pa se z višanjem števila teles zmogljivosti skoraj izenačita, saj je razlika zmogljivosti pri 8102 telesih manj kot 10% in pri zadnji simulaciji, s 65536 telesi, celo manjša kot 1%. Tako lahko trdimo, da so rezultati, pridobljeni za kartico Tesla, primerljivi z ostalimi rezultati za simulacije z večjim številom teles, ki je za nas bolj zanimivo.

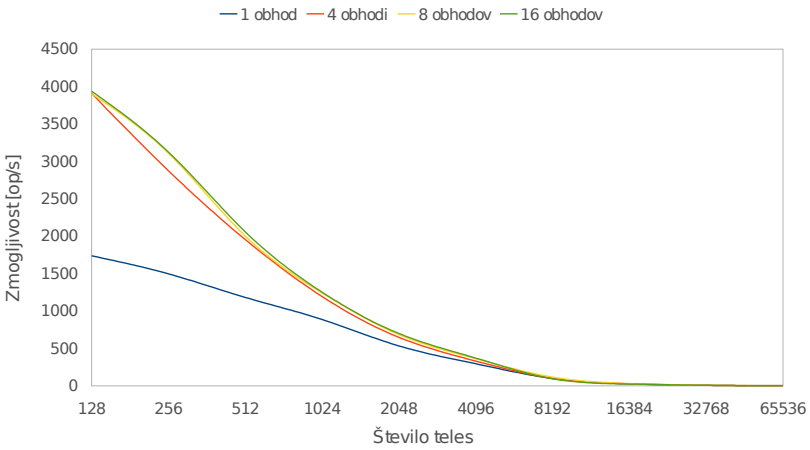
4.2.1 Razvitje zank

V prejšnjem razdelku 4.2 so bile prikazane meritve zmogljivosti izvajanja algoritmov na grafični kartici. Vendar te meritve še niso vključevale ene izmed rešitev za izboljšanje zmogljivosti računanja, omenjene že v razdelku 3.2.2, namreč razvoj zank. Od razvoja zank si obetamo precejšno pohitritev računanja, saj se s to tehniko zmanjša število pogojnih skokov. Zanko smo tako razvili za 1, 4, 8 in 16 obhodov in meritve iz prejšnjega razdelka ponovili (razvoj za 1 obhod v resnici pomeni brez razvoja zanke).

Število teles	Število razvitih obhodov			
	1 [op/s]	4 [op/s]	8 [op/s]	16 [op/s]
128	1736,69	3906,82	3903,64	3937,01
256	1496,06	2873,11	3108,08	3122,63
512	1181,44	1954,38	2001,92	2057,30
1024	885,89	1192,63	1241,75	1250,59
2048	533,33	647,94	685,59	698,97
4096	294,98	330,27	362,81	370,29
8192	97,20	99,54	116,54	94,38
16384	25,36	26,97	29,93	23,68
32768	6,49	6,68	7,62	6,76
65536	1,75	1,77	2,05	1,69

Tabela 4.4: Zmogljivost algoritma z razvitimi zankami za kartico Radeon 5850.

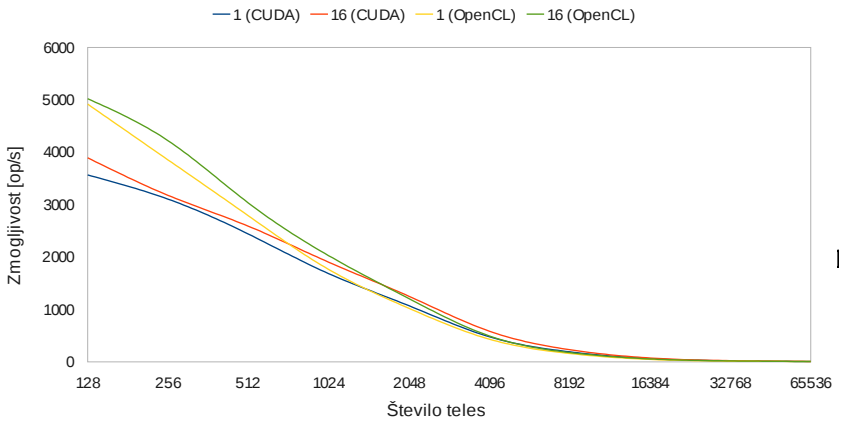
Iz izmerjenih rezultatov lahko vidimo, da razvitje zank dejansko prinese vidne pohitritve pri računanju. Najočitnejša pohitritev je vidna pri majhnem številu teles, kjer znaša pohitritev skoraj 230%. Očitno razvoj zanke zelo koristi v območju, preden uspemo zasesti večino virov kartice. Z večanjem števila teles ta pohitritev pada in je pri zadnji simulaciji le še 17%, a vendar o njej še zmeraj lahko govorimo. Iz rezultatov je prav tako razvidno, da razvoj zanke



Slika 4.4: Zmogljivost pri razvoju zanke za Radeon 5850.

dodaja tudi neko mero dodatne zapletenosti in s tem rabo več prostih virov kartice. Pri kartici Radeon, smo to mejo dosegli pri razvoju zanke za 8 obhodov. Pri razvoju zanke za 16 obhodov je namreč pri zadnji simulaciji (z največjim številom teles) zmogljivost celo slabša kot pri tisti brez razvoja zanke.

Iste teste smo izvedli tudi za računsko kartico Tesla C1060 (tabeli 4.5 in 4.6).



Slika 4.5: Zmogljivost pri razvoju zanke za Tesla C1060.

Tudi pri računski kartici Tesla opazimo, da razvitje zanke vodi do izboljšane

Št. teles	CUDA		OpenCL	
	1 obhod [op/s]	4 obhodi [op/s]	1 obhod [op/s]	4 obhodi [op/s]
128	3565.28	3808.32	4918.94	5443.48
256	3104.10	3202.94	3853.92	5073.76
512	2437.85	2621.72	2786.97	3133.86
1024	1682.12	1885.44	1762.44	2079.05
2048	1068.37	1201.62	1018.51	1232.91
4096	476.64	556.74	430,15	483.01
8192	188.40	219.01	156.39	170.64
16384	57.18	67.78	43.16	52.06
32768	16.57	19.47	12.65	14.61
65536	4.17	4.90	3.35	3.66

Tabela 4.5: Zmogljivost algoritma z razvitimi zankami za kartico Tesla C1060.

Št. teles	CUDA		OpenCL	
	8 obh. [op/s]	16 obh. [op/s]	8 obh.v [op/s]	16 obh. [op/s]
128	3552.15	3894.01	5057.39	5020.25
256	3383.72	3177.16	4229.26	4221.30
512	2612.56	2587.37	3113.88	3031.80
1024	1880.06	1900.83	2031.66	2025.03
2048	1228.78	1249.26	1201.92	1204.02
4096	576.77	583.47	478.99	492.27
8192	225.19	228.27	172.12	174.01
16384	69.86	70.72	52.27	52.92
32768	20.04	20.30	14.76	14.90
65536	5.04	5.11	3.70	3.74

Tabela 4.6: Zmogljivost algoritma z razvitimi zankami za kartico Tesla C1060.

zmogljivosti algoritma. Zanimivo je, da tukaj tudi z razvitjem zanke za 16 obhodov zmogljivost raste iz česar sklepamo, da kljub višjim potrebam po prostih virih za računanje, še nismo dosegli meje. A vendar iz številk mogoče sklepati, da zaradi zgolj manjših izboljšav v primerjavi z razvitjem na 8 obhodov s še daljšim razvitjem zanke boljših rezultatov verjetno ne bi dosegli. V primerjavi z nerazvito zanko je bila pohitritev, za način računanja *cuda*, 23%, medtem ko je bila za način računanja *opencl* le 12%.

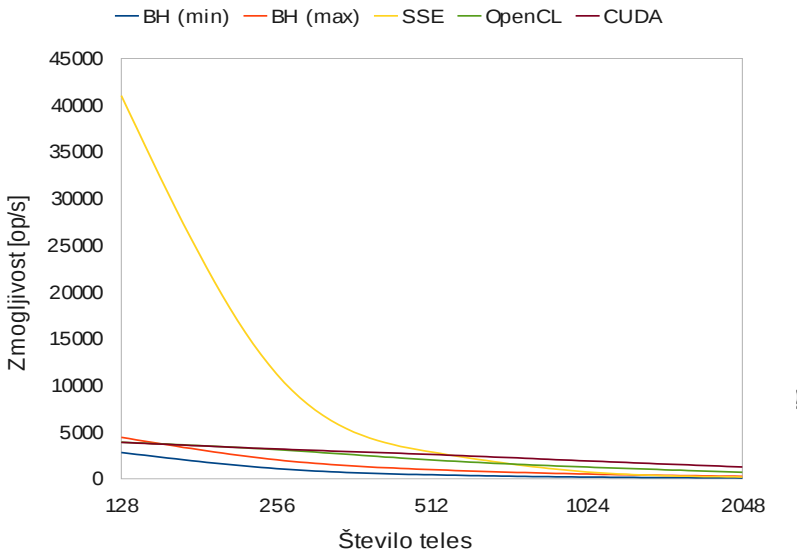
4.3 Primerjava vzporednih in zaporednih algoritmov

Po opravljenih meritvah vzporednih načinov računanja nas je zanimalo, kako se ti obnesejo v primerjavi z naprednim algoritmom za zaporedno računanje. Za primerjavo z Barnes-Hutovim algoritmom smo uporabili:

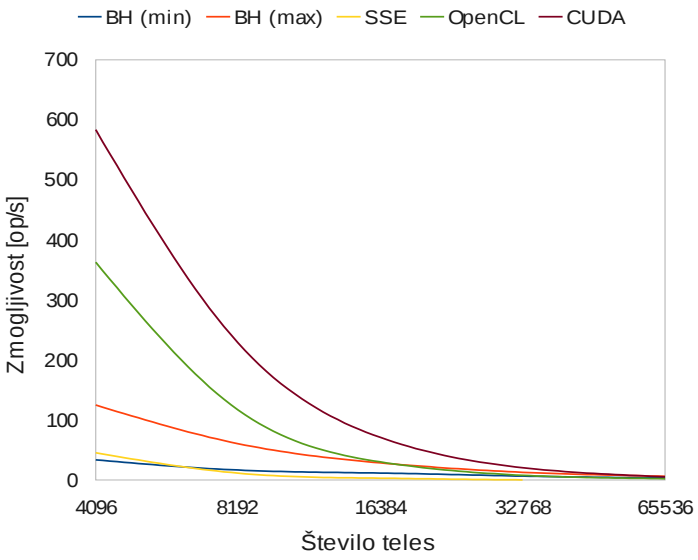
- računanje na procesorju z vsemi štirimi jedri in uporabo SSE ukazov;
- računanje na grafični kartici namiznega računalnika z uporabo knjižnjice za OpenCL; in
- računanje z namensko računsko kartico s knjižnicami za arhitekturo CUDA.

Za razliko od prejšnjih meritev imamo tukaj za Barnes-Hutov algoritem v tabeli 4.7 kar dve vrednosti. To pa zato, ker je zmogljivost algoritma odvisna od trenutne strukture drevesa, ki ga uporabljamo. Najboljše vrednosti za zmogljivost dosežemo največkrat na začetku, ko so telesa enakomerno razdeljena po prostoru. V primeru, ko se telesa zaradi privlačnosti gibljejo vse bližje drugemu, pa zmogljivost pade, saj se drevo izrodi. Do tega pride, ker mora algoritem prostor, v katerem se telesa nahajajo, razdeljevati na manjše kocke, zato da bi se v vsaki kocki nahajalo le eno telo.

Ob pregledu številk opazimo, da je pri majhnem številu teles (do 1024) še smiselno uporabljati algoritme, pospešene z več jedri procesorja ter SSE ukazi. Kakor hitro začnemo posegati po številih nad 1000 teles, pa računanje na grafični kartici prevzame pobudo. Tako je pri računanju z grafično kartico zmogljivost kar 32-krat večja od vzporednega računanja s procesorjem. Pri vsem tem pa ne smemo pozabiti na Barnes-Hutov algoritem, ki deluje kljub uporabi istega procesorja, veliko boljše tudi za visoka števila teles, vsaj če



Slika 4.6: Pred zasičenjem



Slika 4.7: Po zasičenju

Število teles	Barnes-Hut		SSE	OpenCL	CUDA
	min [op/s]	max [op/s]	[op/s]	Radeon [op/s]	Tesla [op/s]
128	2797,91	4426,56	41026,61	3903,64	3894,01
256	1078,92	2019,94	11215,86	3108,08	3177,16
512	412,66	965,12	2844,53	2001,92	2587,37
1024	170,85	495,90	722,87	1241,75	1900,83
2048	80,25	249,89	180,18	685,59	1249,26
4096	33,32	124,74	45,13	362,81	583,47
8192	16,38	60,37	11,18	116,54	228,27
16384	11,39	28,31	2,79	29,93	70,72
32768	6,29	12,71	-	7,62	20,30
65536	3,28	5,93	-	2,05	5,11

Tabela 4.7: Zmogljivosti vseh algoritmov.

ga primerjamo po največjem številu operacij. Sicer so te najvišje številke le redko dosegljive – večinoma le v prvi sekundi računanja, ko so še vsa telesa enakomerno porazdeljena. Vendarle pa je tudi najnižje število operacij višje kot pri uporabi grafične kartice za namizni računalnik, kar nas sili v premislek o smiselnosti uporabe grafičnih kartic. Seveda pa samo številke o zmogljivosti ne dajejo popolne slike o smiselnosti vpeljave naprednih algoritmov.

Ob vsem povedanem ne moremo mimo še ene velike pomanjkljivosti Barnes-Hutovega algoritma. Algoritem od nas že v osnovi zahteva, da simulacijo omejimo v nek določen prostor. V primeru, da katero od teles dobi takšen pospešek, da se uspe odtrgati od vpliva vseh drugih teles, bo v vsakem primeru zašlo izven tega prostora. Za tak primer osnovni algoritem ne predvideva nobene rešitve. Seveda dinamičnega prilagajanja velikosti prostora ni težko dodati, a je bila pomanjkljivost ugotovljena šele med testiranjem algoritma. Preseganja okvirja prostora, pri vzporednih algoritmih skoraj ni mogoče doseči, saj je rob prostora največja možna vrednost števila v plavajoči vejici z enojno natančnostjo.

Če odmislimo to pomanjkljivost, še zmeraj ostaja nekaj argumentov v prid uporabi grafičnih kartic za izvajanje računanja. Izjemne rezultate za algoritem smo dosegli v posebno prirejenem operacijskem sistemu, kjer smo uporabili najboljši prevajalnik ter simulacijo poganjali večkrat in med vsemi izbrali najboljšo. Pri tem ne smemo pozabiti, da je izvedba algoritma precej zapletena.

Pri računanju z grafično kartico lahko govorimo o prav nasprotnem. Dokazali smo, da dobimo ekvivalentne rezultate tudi v normalnem uporabniškem operacijskem sistemu, ki so ob vsem zelo konsistentni čez vse pognane simulacije. Tudi programiranje algoritma je povsem preprosto, saj delujemo po načelu: "preberi podatke, dodeli del pomnilnika in izvajaj računanje po osnovni enačbi". Ne smemo pozabiti niti dejstva, da je zaradi prenosa računanja na grafično kartico procesor prost.

Po vseh pridobljenih rezultatih in po vseh premletih argumentih je težko podati črno-bel odgovor na smiselnost uporabe grafičnih kartic za izvajanje vzporednega računanja. Odgovor na to zavisi od trenutne okoliščine. Če izdelujemo algoritem, namenjen izvajanju v povprečnem uporabniškem okolju (npr. dodajanje posebnega filtra na sliko), kjer želimo z malo truda doseči veliko, je računanje z grafičnimi karticami idealna rešitev. Če imamo na voljo strojno opremo srednje zmogljivosti, so grafične kartice prav tako boljša rešitev. V primeru, ko pa hočemo izvajati res ene in iste in ob tem tudi zahtevne operacije na ogromni količini podatkov in imamo za to na voljo zelo zmogljive računalnike, pa je boljše poseči po algoritmih z manjšo časovno zahtevnostjo.

Poglavje 5

Zaključek

V sklopu diplomskega dela smo iskali odgovor na vprašanje o smiselnosti uporabe grafične kartice za reševanje računsko zelo intenzivnega problema. Pri tem smo zmogljivost takega načina računanja postavili ob bok navadnemu osnovnemu vzporednemu računanju na procesorju ter računanju s pomočjo časovno manj zahtevnega algoritma za zaporedno računanje. V primerjavi z obema se je tak način računanja zmeraj izkazal kot konkurenčen in v večini primerov tudi precej dominanten. Seveda se moramo zavedati, da smo te meritve izvajali zgolj na nekoliko zmogljivejših namiznih računalnikih in bi rezultat lahko bil pri kakšni še bolj specializirani opremi popolnoma drugačen. A nenazadnje so knjižnice za delo z grafičnimi karticami zaenkrat delane prav za takšno okolje.

S stališča programerja je uporaba teh knjižnic še toliko bolj zanimiva, ker so koncepti realizacije algoritma precej enostavni in analogni temu, kar že poznamo. Največ težav je tako povzorčala postavitev osnovnega razvojnega okolja, saj je zaradi narave načina računanja v večini primerov zelo težko ugotoviti zakaj nekatere stvari ne delujejo.

Kljub vsemu pa se moramo zavedati, da smo v diplomskem delu v primeru knjižnic za standard OpenCL preizkušali samo del funkcij ki nam jih omogočajo. Glede na to, da so namenjene združevanju več naprav sistema v en računski stroj, bi bilo zanimivo preizkušati tudi, kako se obnese računanje na grafični kartici in procesorju hkrati ali celo računanje na več grafičnih karticah ipd.

Rezultati izvedeni meritev kažejo na to, da uporaba grafičnih kartic za izvajanje vzoprednega računanja je smiselna, saj smo v določenih primerih računanje pohitrili do 32-krat, v primerjavi z vzporednim računanjem na računalniškem

procesorju. Slednji je boljši le pri manjših obesgih problema, kjer do izraza pride višja frekvenca ure s katero se podatki obdelujejo. Hkrati pa se moramo zavedati, da je stopnja vzporednosti na grafični kartici tudi omejena navzgor. Ko obseg problema enkrat preseže to vrednost, je bolj smiselno poseči po algoritmu z manjšo časovno zahtevnostjo. Seveda to le pod pogojem, da so omejite takega algoritma sprejemljive za naše računanje.

Pri primerjavi obeh tehnologij, ki sta nam na voljo: arhitektura CUDA in standard OpenCL, se prva izkaže kot boljša. V prvi vrsti zato ker na isti strojni opremi zaenkrat omogoča hitrejše računanje. Ob tem se izkaže tudi kot bolj zanesljiva in kar je najbolj pomembno, dokumentacije in primerov izvorne kode je zanjo veliko več. Po drugi strani pa standarda OpenCL ne gre tako hitro odpisati. Glede na to, da gre novo tehnologijo, so težave na začetku pričakovane. Hkrati zmogljivostno zaostaja le za približno 10%. Če se ob tem zavedamo, da nam omogoča računanje z vsemi računsko-sposobnih komponentami računalnika hkrati in ne samo z grafično karico, se tehtnica lahko prevesi njemu v prid.

Dodatek A

Uporaba testnih programov

A.1 Testni program nbody

Testni program nbody poženemo iz ukazne vrstice. Pri tem lahko za nastavotev različnih parametrov izvajanja simulacije uporabimo poljubno kombinacijo stikal. V primeru, da nastavimo kakšno nemogočo kombinacijo stikal, se nelogični parametri simulacije postavijo na privzete smiselne vrednosti.

Izbiramo lahko med sledečimi stikali:

- **-h**
  Stikalo za pomoč. Z uporabo tega stikala se ignorirajo vsa ostala, program pa izpiše seznam stikal z njihovimi opisi.
- **-f**
  Stikalo zažene aplikacijo v celozaslonskem načinu v primeru, da nismo izklopili grafičnega prikaza simulacije.
- **-b**
  S tem stikalom se v grafičnem načinu izpisuje število operacij v sekundi.
- **-a**
  Stikalo onemogoči grafični način izvajanja programa.
- **-c** <način računanja>
  S tem stikalom izberemo način računanja. Izbiramo lahko med:
  - **openmp** – navadno vzporedno računanje,
  - **openmp_sse** – vzporedno računanje z uporabo vektorskih ukazov,

  - **cuda** – računanje s pomočjo grafične kartice s knjižnicami za arhitekuro CUDA, in
  - **opencl** – računanje z uporabo knjižnic za OpenCL.

- **-n** <število teles>
  Stikalo nastavi število teles za katera želimo izvajati računanje. Privzeto število teles je 256.

- **-v** <najvišja hitrost>
  Stikalo nastavi najvišjo dovoljeno začetno hitrost gibanja teles. Privzeta vrednost je 1,0.

- **-m** <največja masa>
  Stikalo nastavi največjo dovoljeno maso teles. Privzeta vrednost je 1,0.

- **-i** <vhoda datoteka>
  Stikalo določi vhodno datoteko, iz katere program ugotovi lastnosti teles, za katera so bo izvajala simulacija. Sintaksa datoteke je navedena v podpoglavju 3.2.1. Z uporabo tega stikala se onemogočijo stikala -n, -v in -m.

- **-o** <izhodna datoteka>
  Stikalo določi izhodno datoteko, v katero se izpisujejo izmerjene vrednosti operacij v sekundi, ko program poganjamo v konzolnem načinu.

- **-t** <število vzporednih izvajanj>
  Stikalo določi, koliko vzporednih tokov naj se uporabi za računanje. To število je navzgor omejeno s številom jeder procesorja pri prvih dveh načinih računanja, in s številom dovoljenih niti oziroma delovnih predmetov v bloku oziroma delovni skupini pri preostalih dveh načinih računanja.

- **-d** <naprava>
  V primeru uporabe načina računanja *opencl*, s tem stikalom določimo napravo, s katero želimo izvajati račune – **cpu** izbere procesor, **gpu** pa grafično kartico.

- **-l** [<številka naprave>]
  S tem stikalom lahko izberemo natančno določeno napravo, s katero bi radi izvajali računanje. Ta opcija nam pride prav, če imamo v sistemu več naprav, in ker program privzeto zmeraj izbere prvo, ki je na voljo, mu s tem stikalom določimo drugo. V primeru, da ne navedemo zaporedne

številke naprave, program izpiše seznam naprav, ki so nam na voljo, skupaj z njihovimi zaporednimi številkami.

- **-u** <število obhodov>
  Stikalo določi, za koliko obhodov zanke se naj razvije najbolj notranja zanka algoritma. To stikalo je veljavno samo pri *cuda* in *opencl* načinu računanja.

- **-p** <ime platforme>
  V primeru, da imamo na sistemu nameščene knjižnice različnih podjetji za računanje po standardu OpenCL, nam to stikalo omogoča izbiro med njimi. Dovoljeni opciji sta **nvidia** in **amd**.

A.2 Testni program barneshut

Testni program barneshut je za uporabnika videz zelo podoben testnemu programu nbody, le da ima le nekaj stikal za različne parametre:

- **-h**
  Stikalo za pomoč. Z uporabo tega stikala se ignorirajo vsa ostala, program pa izpiše seznam stikal z njihovimi opisi.

- **-a**
  Stikalo onemogoči grafični način izvajanja programa.

- **-n** <število teles>
  Stikalo nastavi število teles, za katera želimo izvajati računanje. Privzeto število teles je 256.

Slike

Tabele

Algoritmi

Literatura

[1] D. Kodek, *Arhitektura računalniških sistemov*, Ljubljana: BI-TIM, 2000, pogl. 3., 7 in 8.

[2] (2010) OpenMP - Wikipedia, the free encyclopedia. Dostopno na: http://en.wikipedia.org/wiki/OpenMP

[3] (2010) About OpenMP and OpenMP.org. Dostopno na: http://openmp.org/wp/about-openmp/

[4] (2009) Intel C++ Compiler 11.1 User and Reference Guides. Dostopno na: http://software.intel.com/sites/products/documentation/hpc/compilerpro/-en-us/cpp/lin/compiler_c/index.htm

[5] (2010) AlitVec - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/AltiVec

[6] (2010) Visual Instruction Set - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Visual_Instruction_Set

[7] (2010) Intrinsic function - Wikipedia, the free encyclopedia. Dostopno na: http://en.wikipedia.org/wiki/Intrinsic_function

[8] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture,"IEEE Micro, march 2008

[9] L. Ridgway Scott, T. Clark, B. Bagheri, "Scientific parallel computing", Princeton and Oxford: Princeton University Press, 2005, pogl. 5.

[10] H. Gould, J. Tobochnik, W. Christian, "An introduction to computer simulation methods,"*Applications to physical systems*, San Francisco: Addison Wesley, 2006, pogl. 8.

[11] (2008) AMD R600-Family Instruction Set Architecture. Dostopno na: http://developer.amd.com/gpu_assets/r600isa.pdf

[12] NVIDIA GPU Computing Developer Home Page. Dostopno na: http://developer.nvidia.com/object/gpucomputing.html

[13] ATI Stream SDK — AMD Developer Central. Dostopno na: http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx

[14] DirectCompute - Wikipedia, the free encyclopedia Dostopno na: http://en.wikipedia.org/wiki/DirectCompute

[15] (2010) Nvidia Compute, PTX: Parallel Thread Execution, ISA Version 2.1. Dostopno na: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ptx_isa_2.1.pdf

[16] (2010) NVIDIA CUDA C Programing Guide. Dostopno na: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

[17] (2009) OpenCL Specification 1.0. Dostopno na: http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf

[18] (2010) About the Khronos Group. Dostopno na: http://www.khronos.org/about/

[19] P. Bulić, "Osnovni postopki v računalniški grafiki", Ljubljana, 2006, pogl. 7

[20] (2009) Fast N-Body Simulation with CUDA. Dostopno na: http://developer.download.nvidia.com/compute/cuda/sdk/website/C/src/nbody/doc/nbody_gems3_ch31.pdf

[21] E. Elsen, V. Pande, V. Vishal, M. Houston, P. Hanrahan, E. Darve, "N-Body Simulations on GPUs", Stanford, 2007

[22] J. Barnes, Piet Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, Št. 324, 1986

[23] J. Dubinski, "A Parallel Tree Code", Santa Cruz, 1996

[24] The Linux Home Page at Linux Online. Dostopno na: http://www.linux.org/

[25] Tesla Tech Specs. Dostopno na: http://www.nvidia.co.uk/page/tesla_supercomputer_tech_specs.html

[26] ATI Radeon™ HD 5850 graphics. Dostopno na: http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5850/Pages/ati-radeon-hd-5850-overview.aspx#2

[27] GCC, the GNU Compiler Collection. Dostopno na: http://gcc.gnu.org/

[28] Intel® Compilers - Intel® Software Network. Dostopno na: http://software.intel.com/en-us/intel-compilers/

[29] AMD Phenom™ II Processors. Dostopno na: http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii.aspx