Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Gregor Vitek

Porazdeljevanje dela v heterogenih računalniških sistemih

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Raziščite možnosti učinkovite izrabe heterogenih računalniških sistemov za visoko zmogljivo računanje. Izdelajte splošno programsko ogrodje, osnovano na jeziku OpenCL, ki bo omogočalo enostavno dodajanje strategij za delitev dela med napravami v sistemu. Na računsko zahtevnem problemu preizkusite različne strategije za delitev dela in jih primerjajte med seboj.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Gregor Vitek sem avtor diplomskega dela z naslovom:

Porazdeljevanje dela v heterogenih računalniških sistemih

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Uroša Lotriča,
- 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 na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 15. septembra 2015 Podpis avtorja:

Rad bi se zahvalil svojemu mentorju izr. prof. dr. Urošu Lotriču za strokovno pomoč pri izdelavi diplomskega dela.

Prav tako se želim zahvaliti družini in prijateljem, ki so me vzpodbujali pri delu.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CPE	Central processing unit	Centralna procesna enota
GPE	Graphical processing unit	Grafična procesna enota
HPC	High performance computing	Visoko zmogljivo računanje
FLOPS	Floating point operations per second	Število operacij v plavajoči vejici na sekundo
TFLOPS	10^9 FLOPS	10^9 FLOPS

Povzetek

V diplomskem delu opisujemo zasnovo in izvedbo aplikacije, ki na heterogenem računalniškem sistemu razdeli, izračuna in združi vzporeden računski problem. Za izračun aplikacija izrabi vse računske naprave, ki so ji dosegljive. Naš cilj v delu je ugotoviti, kako najbolje razdeliti računski problem med naprave tako, da je izkoriščenost sistema čim večja. Predstavimo možne načine delitve problema, njihove prednosti in slabosti. Rešitve so testirane in med seboj primerjane. Testiranje je opravljeno z računanjem Mandelbrotove množice. Za dostop do naprav na sistemu aplikacija uporablja ogrodje OpenCL. Ugotovili smo, kako najbolje dodeljevati delo procesnim enotam in kako nastaviti velikosti delovnih skupin za čim boljši izkoristek naprav.

Ključne besede: heterogeni sistemi, OpenCL, vzporedno računanje, visoko zmogljivo računanje.

Abstract

The thesis explores the formulation and implementation of an application that divides, computes and merges a parallel computing problem on a heterogeneous system. The application uses all available compute devices. The goal is to determine how to divide a computing problem between devices to maximise the system's utilisation. The thesis presents possible solutions to the problem, their strengths, and weaknesses. Some of the solutions are benchmarked and compared. For benchmarking the Mandelbrot set was generated. Processing units are accessed and managed using the OpenCL framework. We found out how best to divide and allocate work, and how to set the size of work groups to imporve device utilisation.

Keywords: heterogeneous systems, OpenCL, parallel computing, high performance computing.

Poglavje 1

Uvod

Zaradi fizikalnih omejitev pri razvoju sodobnih procesorjev se je moderno računalništvo, namesto poviševanja procesorskega takta, usmerilo v paralelizem. S porastom grafičnih procesnih enot in njihovim razvojem iz fiksnih grafičnih cevovodov preko programirljivih senčilnikov (ang. *programmable shaders*) do splošno-namenskih procesnih enot, so programerji dobili veliko paralelne procesne moči. Moderni sistemi danes ne vključujejo le več-jedrnih procesorjev, ampak ponavadi tudi vsaj en ali več grafičnih procesorjev, nekateri pa celo dodatne koprocesorje in računske pospeševalnike.

Toda učinkovita izraba takih heterogenih sistemov je zelo težka. Pristopi za programiranje različnih vrst procesnih enot so povsem različni. Za programiranje na centralnih procesnih enotah obstaja mnogo programskih jezikov in razvojnih okolij in za mnoge od njih obstajajo odprtokodni in prosto dostopni prevajalniki. Centralne procesne enote običajno uporabljajo nekaj visoko zmogljivih jeder, ki imajo velik nabor ukazov namenjenih aritmetično-logičnim operacijam in upravljanju z zunanjimi napravami, navideznim pomnilnikom ter prekinitvami [13]. Za razliko, grafične procesne enote običajno tečejo z nižjim taktom in imajo veliko več jeder, ki so preprosta in namenjena le aritmetično-logičnim operacijam. Prav tako nimajo podpore za navidezni pomnilnik. Njihov pomnilnik je razdeljen v hierarhično arhitekturo, kjer imajo jedra poleg glavnega tudi svoj lokalni pomnilnik [25].

Kot konkurenco grafičnim procesnim enotam je Intel predstavil arhitekturo MIC (ang. *Many Integrated Core*), ki s pomočjo manjšega števila, bolj zmogljivih jeder lahko doseže podobne ali celo boljše rezultate kot visoko zmogljive grafične procesne enote. Poleg teh naprav obstaja še vrsta drugih, kot so programirljiva vezja (ang. *field programmable gate array, FPGA*) in mikroprocesorji za obdelavo signalov (ang. *digital signal processor, DSP*).

Programiranje zunanjih naprav navadno zahteva uporabo zaprte programske opreme izdelovalca naprave. Arhitektura in programska oprema se med tipi procesnih enot in proizvajalci zelo razlikujejo, kar dodatno otežuje njihovo učinkovito izrabo. Za poenoteno uporabo procesnih enot je leta 2009 Apple predstavil OpenCL [1]. Danes za njegovo specifikacijo skrbi skupina Khronos.

V tem delu smo želeli predstaviti zasnovo in izdelavo aplikacije, ki bi določen problem izračunala z učinkovito izrabo poljubnega heterogenega sistema. Taki sistemi so zelo razširjeni, vendar je uporaba celotnega sistema za računanje težavna. Za polno izrabo sistema je potrebno uporabiti več orodij in vmesnikov hkrati ali pa uporabiti splošno ogrodje, kot je OpenCL. Težave imamo tudi z delitvijo problema na podprobleme za različne procesne enote, saj le te niso enako zmogljive in učinkovite pri računanju različnih problemov.

V sledečih poglavjih najprej predstavimo heterogene sisteme in nekatere bolj znane arhitekture koprocesorjev, nato predstavimo OpenCL, ki smo ga uporabili za upravljanje s heterogenim sistemom. V četrtem poglavju predstavimo delitve dela na heterogenem sistemu in njihove lastnosti. V naslednjih dveh poglavjih predstavimo implementacijo aplikacije za delitev dela na sistemu ter meritve njene učinkovitosti.

Poglavje 2

Heterogeni sistemi in prenosljiva zmogljivost

Heterogeni računalniški sistemi so sistemi, ki uporabljajo več različnih procesnih enot za računanje [11]. Taki sistemi so danes zelo razširjeni na veliko področjih. Večina današnjih osebnih računalnikov je heterogen sistem z eno centralno procesno enoto (CPE) in eno grafično procesno enoto (GPE). Velika večina super-računalnikov na spisku super-računalnikov *top500.org* uporablja pospeševalnike ali koprocesorje, velikokrat Xeon Phi ali Nvidia Tesla [24]. Za programiranje koprocesorjev in pospeševalnikov obstaja več različnih programski ogrodij, kot so C++AMP, CUDA in OpenCL [5, 17, 7]. Vendar večina ogrodij ni prenosljivih ali niso primerna za programiranje heterogenih sistemov, kjer uporabljamo več raznolikih naprav. Izjema je OpenCL, ki omogoča izrabo večine bolj poznanih koprocesorjev in pospeševalnikov. Druga ogrodja pestijo problemi s prenosljivostjo med operacijskimi sistemi (C++AMP), prenosljivostjo med različnimi arhitekturami pospeševalnikov (CUDA, OpenMP) ali pa pomanjkanje podpore prevajalnikov (OpenACC). Najbolj razširjeni koprocesorji so danes brez dvoma GPE [18]. V zadnjem času je prav tako narasla popularnost arhitekturi MIC proizvajalca Intel [24, 2].

2.1 Računanje na GPE

Moderne grafične procesne enote (GPE) so visoko zmogljive in specializirane procesne enote [6, 19, 12, 4]. Zaradi njihove uporabe v namiznih aplikacijah in računalniških igrah so postale izjemno razširjene in eno od glavnih področij razvoja računalniške strojne opreme. Čeprav je večina GPE usmerjena v računanje s plavajočo vejico v enojni natančnosti, kar povsem zadostuje za potrebe računalniške grafike, so za potrebe računstva veliki proizvajalci začeli izdelovati tudi GPE z zmožnostjo računanja v dvojni natančnosti. Zaradi njihovega specializiranega namena, manjše potrebe po nizki zakasnitvi in namesto velikih količin predpomnilnika imajo lahko GPE večji delež vezja namenjen aritmetično logičnim enotam [8]. Z velikim številom vektorskih aritmetično logičnih enot lahko na GPE teče veliko število vzporednih in povsem neodvisnih niti. Zato so moderne GPE sposobne doseči več kot 5 milijard operacij v plavajoči vejici na sekundo (ang. *floating point operations per second, FLOPS*) v enojni natančnosti in čez 2 TFLOPS v dvojni natančnosti. Za primerjavo, moderni CPE so zmožni doseči le okoli 0,1 TFLOPS.

Računanje na grafičnih procesorjih (ang. *General purpose computing on GPU, GPGPU*) se je začelo že pred razvojem posebnih orodij zanj [18]. Prvi poskusi izrabe njihove moči so bili izvedeni že na GPE s fiksnim cevovodom. Ti cevovodi so dopuščali le zelo omejene operacije namenjene prikazovanju slik, zato njihova izraba za splošno računstvo ni bila zelo zanimiva. Z razvojem in popularizacijo grafičnih senčilnikov in strojne opreme, ki jih je podpirala, je med leti 2000 do 2005 nastala želja po izrabi programirljivih grafičnih cevovodov. Programer je moral razpon svojih vhodnih podatkov predstaviti kot geometrijo, vhodne podatke zakodirati v teksture, nato pa je GPE za vsak fragment pognal senčilnik fragmentov. Ta senčilnik je nato rezultate izrisal na teksturo, iz katere je moral programer dekodirati rezultate. Pospešitve so bile mogoče predvsem pri problemih z vektorji in matrikami, nad katerimi so operacije na GPE tekle hitro. Z razvojem novejših pristopov lahko programer direktno določi razpon problema, vhodne in izhodne podatke pa zapiše direktno v pomnilnik GPE.

2.2 Intel MIC

Trenutno najmočnejši super računalnik na svetu Tianhe-2 za pospeševalnike uporablja multiprocesorje Xeon Phi [26]. Xeon Phi pripada Intelovi arhitekturi MIC, ki je bila razvita kot alternativa zmogljivim GPE, namenjenim predvsem za računske naloge in je dosti bolj podobna arhitekturi CPE. Xeon Phi za razliko od GPE uporablja manjše število bolj zmogljivih jeder. Vsako jedro Xeon Phi vsebuje 512 bitno vektorsko enoto, ki je zmožna vseh osnovnih matematičnih operacij, podpira pa tudi operacijo sočasnega seštevanja in množenja naenkrat (ang. *fused multiply-add, FMA*) [21, 20]. Xeon Phi ima velike količine predpomnilnika. Vsako jedro ima 64 kB predpomnilnika na nivoju L1 in 512 kB predpomnilnika na nivoju L2. Xeon Phi lahko z uporabo tehnologije *hyper-threading* naenkrat izvaja štiri niti na enem jedru, vendar se v eni urini periodi izvedejo ukazi le iz ene niti. Za razliko od GPE je Xeon Phi dosti bolj učinkovit pri računanju, kadar se v programih pojavijo pogojni skoki.

2.3 Prenosljiva zmogljivost

Pri prenosljivih programih težko dosežemo dobro izkoriščenost heterogenega sistema. Programska ogrodja, kot so OpenCL, nam omogočajo, da programe izvajamo na raznolikih napravah, a ne zagotavljajo, da bodo ti programi na vseh napravah tekli dobro [22]. Zaradi raznolikosti arhitektur programi, ki na nekateri napravah tečejo dobro, na drugih delujejo počasi, saj ne izrabljajo vseh zmogljivosti strojne opreme. Pri pisanju kode, ki bo tekla na več različnih procesnih enotah, se morajo programerji odločiti, kako naj svojo izvorno kodo napišejo. Lahko napišejo preprosto kodo, ki na vseh arhitekturah teče dobro, a na nobeni odlično, lahko pa jo specializirajo za vse naprave, ki so jim na voljo. Prva rešitev je prenosljiva, a slabše izkoristi naprave. Pri drugi rešitvi, ki dobro izkoristi naprave, pa trpi prenosljivost, saj je pri zamenjavi ali nadgradnji sistema potrebno ponovno dopisati specializacije za nove naprave. Srednja pot je, da za najmočnejše naprave pripravimo osnovne

specializacije, saj tako najhitreje dobimo večje pohitritve.

Poglavje 3

Izbrana tehnologija

3.1 OpenCL

Za izrabo heterogenih sistemov smo uporabili OpenCL (ang. *Open Computing Language*). OpenCL je odprt standard za izdelavo paralelnih programov, ki tečejo na heterogenih platformah in sistemih. Omogoča izrabo vseh procesnih enot v sistemu prek enotnega vmesnika. Implementacije OpenCL obstajajo za širok nabor računalniških arhitektur, od vgrajenih sistemov do visoko zmogljivih CPE in GPE. Programsko ogrodje OpenCL sestavljajo programski vmesnik (ang. *OpenCL API*), programski jezik OpenCL C, knjižnice in gonilniki za razvoj programske opreme. [9] [7] [23] [15]

Namen OpenCL je, da lahko programerji pišejo učinkovito kodo, ki je prenosljiva tudi med napravami s povsem različno arhitekturo. Tako lahko izvajamo isti program sočasno na več napravah in polno izkoristimo računalniški sistem. Ukazi vmesnika OpenCL so pri zunanjih procesnih enotah ukazi gonilnikom naprav, na CPE, kjer poženemo aplikacijo, pa so ukazi le klici funkcij v knjižnici, s katero je aplikacija povezana. Kadar v tem besedilu omenjamo ukaze vmesnika, se na njih sklicujemo z imeni iz vmesnika za C.

Jezik OpenCL C je prilagojen za uporabo podatkovnega paralelizma in nam olajša eksplicitno uporabo vektorskih ukazov, saj so vgrajeni v jezik. Pri navadnih prevajalnikih za CPE je potrebno za eksplicitno uporabo vektorskih

ukazov uporabiti zbirni jezik, ki ni prenosljiv med različnimi arhitekturami. Prevajalniki za OpenCL C so prilagojeni za uporabo na paralelnih arhitekturah in tako programsko kodo prevajajo drugače kot navadni prevajalniki za CPE. [14]

Heterogene sisteme OpenCL opiše s štirimi glavnimi modeli:

- model računskih naprav,
- model izvajanja,
- pomnilniški model,
- programski model.

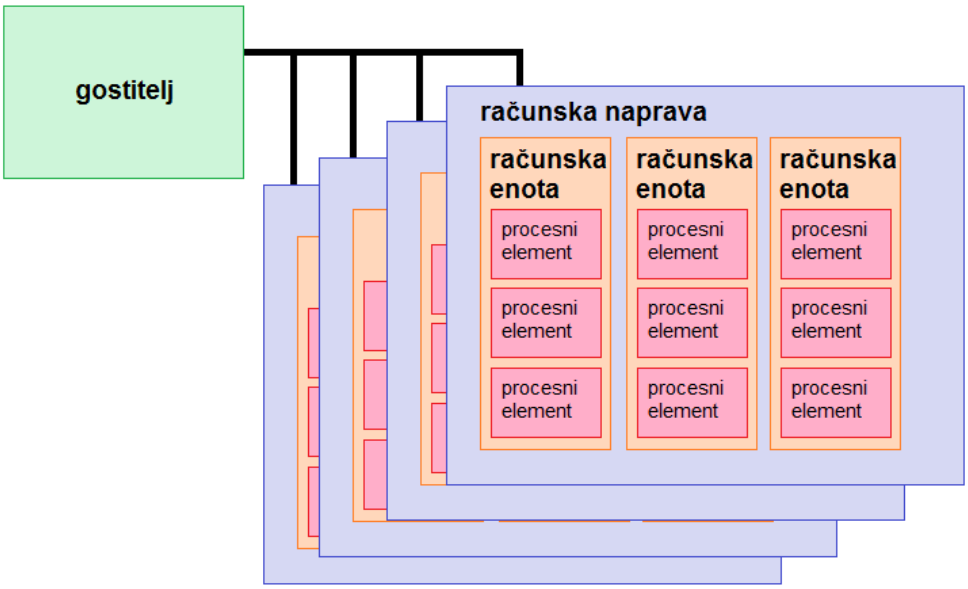
3.1.1 Model računskih naprav

Model računskih naprav OpenCL je sestavljen iz gostitelja (ang. *host*), povezanega na eno ali več računskih naprav (ang. *device*). Vsaka naprava je nadalje razdeljena na več računskih enot (ang. *compute units*), ki vsebujejo procesne elemente (ang. *process element*), glej sliko 3.1. Aplikacija, ki uporablja OpenCL, teče na gostitelju in napravam pošilja ukaze, ki naj se izvedejo.

3.1.2 Model izvajanja

Model izvajanja OpenCL je razdeljen na dva dela: na ščepce (ang. *kernels*) in gostiteljski program (ang. *host program*). Ščepec je program, ki ga nameravamo izvajati na računskih napravah. Napisan je v programskem jeziku OpenCL C.

N-dimenzionalni razpon (ang. *NDRange*) je koncept, ki ga OpenCL uporablja za predstavitev problemskih prostorov. Vsak N-dimenzionalni razpon ima N dimenzij, kjer je N med 1 in 3. Za predstavitev problema moramo v vsaki dimenziji določiti tudi velikost razpona. Kadar v tem delu opisujemo velikosti N-dimenzionalnih razponov uporabimo notacijo A x B x C, kjer so



Slika 3.1: Model računskih naprav OpenCL

A, B in C velikosti razpona v posamezni dimenziji. Če razpon ni tridimenzionalen, lahko velikost neuporabljene dimenzije izpustimo, na primer A x B.

Za vsak ščepec se ob zagonu vzpostavi problemski prostor, ki mu pravimo globalni razpon (ang. *global range*). Globalni razpon je N-dimenzionalni razpon. Ščepec se izvede na vsaki točki tega razpona. Vsaka nit, ki izvaja ščepec, se imenuje delavec (ang. *work-item*) in se izvaja na procesnem elementu. Delavci so razdeljeni v delovne skupine (ang. *work-group*), ki se izvajajo na računskih enotah. Velikost delovne skupine je tudi N-dimenzionalni razpon, ki mu pravimo lokalni razpon (ang. *local range*). Velikosti lokalnega razpona v vsaki od dimenzij morajo deliti velikosti globalnega razpona v vsaki istoležni dimenziji.

Vsak ščepec, ki se izvaja, ima tako globalno in lokalno identifikacijsko številko ter številko delovne skupine (ang. *global ID, local ID, work-group ID*).

Gostiteljski program, ki ga poženemo na CPE, upravlja z vmesnikom OpenCL, pripravi naprave, prevede in poganja ščepec. Prevajalnik za OpenCL

C mora zagotoviti izdelovalec gonilnika naprave, na kateri se bo izvajal ščepec. Ščepce prevajamo za vsako napravo posebej, ko zaženemo gostitelja (ang. *run-time compilation*).

Priprava naprave zahteva nekaj korakov. Gostitelj iz datoteke najprej prebere izvorno kodo ščepca. Nato je potrebno pridobiti vse obstoječe platforme (ang. *platform*). Vsaka platforma predstavlja drugo implementacijo ogrodja OpenCL na našem sistemu. Vsaka platforma vsebuje eno ali več naprav (ang. *device*). Naprava je objekt, ki v vmesniku OpenCL predstavlja fizične naprave na našem sistemu, na gostitelju pa dobimo le njeno identifikacijsko številko. Preden lahko naprave uporabimo, moramo za njih ustvariti kontekst (ang. *context*). Vse naprave, ki se nahajajo v isti platformi, lahko vključimo v en kontekst. Za nadaljnje izvajanje moramo v kontekstu ustvariti nekaj objektov OpenCL. Naprave v istem kontekstu si lahko delijo nekatere objekte OpenCL iz tega konteksta. Na primer: v sklopu konteksta izvorno kodo, ki jo je naložil gostitelj, prevedemo in s tem v njem dobimo programski objekt (ang. *program object*), ki ga lahko za izvajanje uporabijo vse naprave v kontekstu.

3.1.3 Objekti OpenCL

Vsi objekti OpenCL so narejeni v kontekstu in so vezani nanj, lahko so vezani tudi na specifično napravo. Uporabili smo naslednje objekte OpenCL.

Ukazna vrsta

Ukazna vrsta (ang. *command queue*) je objekt, s katerim dajemo ukaze napravi. Ukazna vrsta je vezana na napravo. V vrsto lahko uvrstimo tri glavne vrste ukazov:

- ukaze za izvedbe ščepca,
- ukaze za upravljanje s pomnilnikom,
- ukaze za sinhronizacijo.

Če ne zahtevamo drugače, ukazna vrsta izvaja ukaze v istem vrstnem redu, kot so uvrščeni v vrsto.

Programski objekt

Programski objekt (ang. *program object*) je objekt, ki predstavlja prevedeno izvorno kodo ščepca. Vsak programski objekt lahko vsebuje prevedeno kodo za več kot eno napravo.

Objekt ščepca

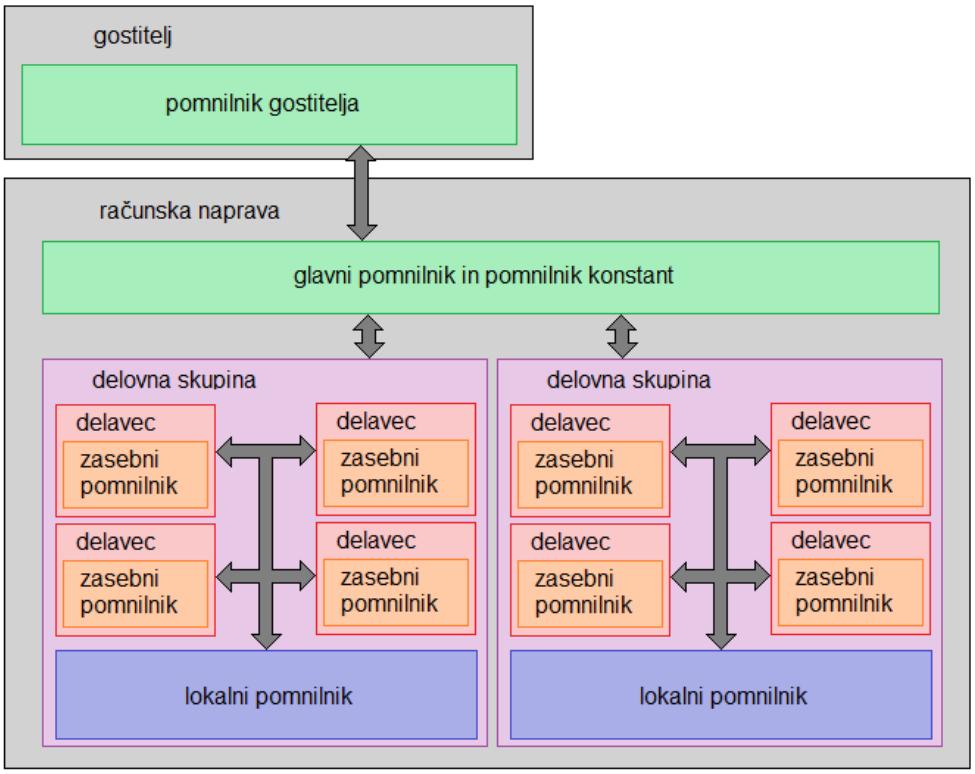
Vsak ščepec, ki ga želimo zagnati na napravi, mora biti predstavljen z objektom (ang. *kernel object*) na gostitelju. S pomočjo tega objekta lahko ščepcu določimo argumente. Ta objekt nato uvrstimo v ukazno vrsto skupaj z ukazom za zagon ščepca.

Dogodkovni objekt

Dogodkovni objekt (ang. *event object*) je objekt za sledenju dogodkov, na primer za spremljanje stanja ukazov v ukazni vrsti. Če želimo slediti dogodkom, moramo dogodkovni objekt povezati z ukazom v ukazni vrsti. S pomočjo dogodkov lahko ugotovimo stanje ukaza, izmerimo njegov čas izvajanja in ga sinhroniziramo z drugimi ukazi.

Pomnilniški objekt

Pomnilniški objekt (ang. *memory object*) je objekt za upravljanje s pomnilnikom. Pomnilniški objekt je dostopen vsem napravam v kontekstu. Za vsak pomnilniški objekt lahko prepovemo pisalne ali bralne dostope. Če želimo da naprava piše ali bere iz pomnilnika, moramo ta objekt podati kot argument ščepcu.



Slika 3.2: Pomnilniški model OpenCl

3.1.4 Pomnilniški model

Pomnilniški model loči štiri vrste pomnilnika, do katerega lahko dostopajo ščepci: globalni pomnilnik, pomnilnik konstant, lokalni pomnilnik in zasebni pomnilnik. Pomnilnik konstant je del pomnilnika, ki se med izvajanjem ščepcev ne spreminja in je dostopen vsem delavcem. Globalni pomnilnik je prav tako dostopen vsem delavcem, ki lahko vanj pišejo in iz njega berejo. Vsaka delovna skupina ima svoj lokalni pomnilnik, vsak delavec pa ima še svoj zasebni pomnilnik. Gostitelj lahko dostopa samo do globalnega pomnilnika in pomnilnika konstant. Shemo modela prikazuje slika 3.2.

3.1.5 Programski jezik OpenCL C

Programski jezik OpenCL C je jezik v katerem pišemo ščepce. Njegova osnova je programski jezik C, bolj natančno standard C99. Vsebuje določene omejitve in razširitve. Omejitve vsebujejo:

- omejeno delo s kazalci,
- kazalci na funkcije so prepovedani,
- ni bitnih polj,
- ni tabel s spremenljivo velikostjo,
- nekatere ključne besede in zaglavja iz standarda C99 niso dostopna,
- rekurzija ni dovoljena,
- brez razširitev se v polja z elementi manjšimi od 32 bitov ne sme pisati.

Tipi

Za razliko od standarda C99, kjer imajo celoštevilski tipi določeno le spodnjo mejo velikosti, imajo osnovni tipi OpenCL C natančno določeno velikost. OpenCL C podpira večino standardnih osnovnih tipov: *char, short, int, long in float*. Vsi celoštevilski tipi imajo tudi nepredznačene oblike. Tip *long long* ne obstaja zaradi spremenjenih omejitev velikosti. OpenCL C dodatno pozna naslednje osnovne tipe:

- *bool* - logični tip,
- *half* - tip za 16 bitno plavajočo vejico po standardu IEEE 754.

Poleg teh ima OpenCL C še nekaj osnovnih tipov, ki nimajo določene velikosti: *size_t, intptr_t, uintptr_t, void*. Opcijsko lahko implementacije podpirajo tudi tip *double*.

Poleg tega ima dodane še vektorske tipe, ki so lahko velikosti 2, 4, 8, 16, na primer: `char2`, `float4`, `int8`, `short16`. Vse tabele, ki niso v zasebnem

pomnilniku, morajo imeti tudi specificirano dostopnost: globalno, konstantno ali lokalno s ključnimi besedami `__global`, `__constant`, `__local`.

Vgrajene funkcije

OpenCL pozna veliko vgrajenih funkcij, ki omogočajo uporabo ščepcev in olajšajo računanje. Najprej so tu funkcije za lociranje ščepca v računskem prostoru in so opisane v tabeli 3.1. Standard nadalje vsebuje funkcije za sinhronizacijo, delo s pomnilnikom, delo s celimi števili, delo s prostorskimi vektorji, primerjave med vektorji in funkcije za matematične operacije nad tipi *float* in *half*, podobne tistim, ki se nahajajo v zaglavju `<math.h>` standarda C99, le da podpirajo tudi vektorske tipe. Veliko matematičnih funkcij se lahko na primernih napravah prevede direktno v en strojni ukaz.

Opcijsko lahko implementacije OpenCL vključujejo tudi funkcije za atomarne operacije in matematične operacije za plavajočo vejico z dvojno natančnostjo.

Tabela 3.1: Funkcije za osnovno delo s ščepci

Funkcija	opis
get_work_dim()	Vrne število dimenzij
get_global_size(D)	Vrne velikost dimenzije D
get_global_id(D)	Vrne ID številko ščepca v dimenziji D
get_local_size(D)	Vrne velikost dimenzije D delovne skupine
get_local_id(D)	Vrne lokalno ID številko ščepca v dimenziji D
get_num_groups(D)	Vrne število delovnih skupin v dimenziji D
get_group_id(D)	Vrne ID številko skupine v dimenziji D

Preprost primer za ilustracijo delovanja OpenCL C in primerjava s C na primeru seštevanja vektorjev vidimo v izsekih 3.1 in 3.2.

```
void vecAdd(int* a,
            int* b,
            int* c,
            int size) {
    for(int i=0;i < size; i++){
         c[i] = a[i] + b[i];
    }
}
```

Izsek kode 3.1: Seštevanje vektorjev v C

```
__kernel void vecAdd(__global int* a,
                     __global int* b,
                     __global int* c) {
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Izsek kode 3.2: Seštevanje vektorjev v OpenCL C

3.2 Programski jezik C++

Gostiteljski program

Za razvoj gostiteljskega programa smo uporabili programski jezik C++. Vmesnik OpenCL je v osnovi zasnovan za programski jezik C, vendar obstajajo ovojnice za večino popularnih jezikov. Odločili smo se za C++, ker je jezik dovolj nizko-nivojski, da dosežemo visoke hitrosti pri razdeljevanju dela na naprave, a dopušča lažje delo z objekti OpenCL, ki jih lahko uporabljamo kot prave programske objekte. Ker je naš program uporabljal vse dosegljive naprave, je bilo pomembno, da je bil jezik programa na gostitelju dovolj varčen s procesorskim časom in s pomnilnikom, saj bi drugače oviral izvajanje ščepcev.

Prevajali smo s prevajalnikom `gcc`, natančneje z verzijo `4.4.7 20120313`, ki je bila nameščena na razvojnem okolju. Ukaz za prevajanje lahko vidimo

```
g++ -O3 -std=c++0x -Lpath-OpenCL-include
 -Lpath-OpenCL-libdir -lOpenCL -o Main.out
 Main.cpp lodepng.cpp
```

Izsek kode 3.3: Ukaz za prevajanje

v izseku 3.3. Pri prevajanju smo eksplicitno uporabili prevajalnik za C++ `g++`, ki po prevajanju avtomatično poveže še *libstdc++*, kar je enako kot če bi `gcc` podali zastavico `-lstdc++`. Sledita mu zastavici za vklop optimizacij in določitev verzije C++ standarda. Zastavica `-std=c++0x` specificira najnovejšo verzijo standarda za naš prevajalnik. Naslednje trije argumenti se uporabljajo za povezovanje s knjižnicami za OpenCL. Prvi dva dodata imenike, ki jih preišče povezovalnik, `-lOpenCL` pa določi s katero knjižnico naj se program poveže. Na koncu še določimo izhodno datoteko s končnim programom in vhodni datoteki z izvorno kodo.

Lodepng

Lodepng je odprtokodna in prosto dostopna knjižnica za branje in pisanje slikovnih datotek. Programska koda se zanaša le na standardno knjižnico in je zato zelo prenosljiva. Uporabljena je le za preverjanje rezultatov,natančneje ali vse naprave pravilno računajo in pravilno zapisujejo v pomnilnik.

3.3 Uporabljena strojna oprema

Razvoj in meritve so potekale na računskem strežniku, ki se nahaja na FRI. Strežnik teče na dveh Intel Xeon E5-2620. Poleg tega ima vgrajen ko-procesor Xeon Phi 5110P in dve GPE Tesla K20m.

3.3.1 Intel Xeon E5-2620

Xeon E5-2620 je CPE s 6 fizičnimi jedri, ki z vklopljeno tehnologijo *hyper-threading* predstavljajo 12 navideznih jeder. Naš testni strežnik je imel na

voljo dva takšna procesorja, tako da smo imeli na voljo 24 navideznih jeder. Vsakič ko je v besedilu omenjen ta procesor, sta v resnici mišljena oba procesorja skupaj. Njegova teoretična zmogljivost pri enojni natančnosti je 96 GFLOPS. Pri testiranju je bil uporabljen kot naprava gostitelja ter kot manj zmogljiva računska naprava.

3.3.2 Nvidia Tesla K20m

Nvidia Tesla K20m je GPE, namenjena za računska opravila. Tesla ima 2496 jeder CUDA - *Compute Unified Device Architecture*. Odlikuje jo, da je zmožna dela v plavajoči vejici z dvojno natančnostjo. Pri dvojni natančnosti je največja teoretična zmogljivost 1,17 TFLOPS, pri enojni natančnosti pa 3,5 TFLOPS [16]. Pri testiranju smo uporabili dve Tesli, ki sta služili kot primer normalne arhitekture uporabljene v HPC.

3.3.3 Xeon Phi 5110P

Xeon Phi 5110P je multiprocesor Intelove arhitekture MIC (Many Integrated Core). Arhitektura je podrobneje predstavljena v razdelku 2.2. Ima 60 jeder, vsak od njih ima zmogljivo vektorsko procesno enoto. Z vklopljeno tehnologijo *hyper-threading* ima na voljo 240 navideznih jeder. Pri enojni natančnosti ima teoretično zmogljivost 2,0 TFLOPS [3]. Pri testiranju je bil uporabljen kot zmogljiva alternativa arhitekturi grafičnih kartic.

Poglavje 4

Delitev dela v heterogenem sistemu

Heterogene sisteme sestavljajo različno močne naprave, zato je pri računanju na njih glavni problem učinkovita delitev dela. V nadaljevanju so predstavljene nekatere možne delitve dela.

Glavni cilji delitve so:

- da se delo razdeli uravnoteženo med naprave,
- da se naprave čim bolje izkoristi,
- da zasedenost naprav z drugim delom čim manj vpliva na našo aplikacijo,
- da pri računanju ves čas sodelujejo vse naprave.

4.1 Enakomerna delitev

Najpreprostejši tip delitve je enakomerna delitev na D enakih delov, kjer D predstavlja število naprav. Glavna prednost take delitve je, da je program na gostitelju najbolj preprost in najhitreje pripravljen. Gostitelj mora namreč samo razdeliti celotno nalogo na enake dele in nato počakati do konca izvajanja. Kritična slabost je, da bo program tekel tako dolgo, kot bo potrebovala

najpočasnejša naprava. Medtem hitrejše naprave stojijo. Pri nekaterih problemih, ki so zelo primerni za računanje na GPE, kot je tudi Mandelbrotova množica, se lahko zgodi, da v konfiguraciji ene CPE in ene GPE, GPE zaključi z računanjem tudi več kot desetkrat hitreje od CPE.

4.2 Utežena delitev

Naslednji tip delitve dela je podoben kot prejšnji, kjer delo razdelimo na N delov, le da te dele otežimo s utežmi. Uteži naprav določimo glede na njihov tip. Nas v OpenCL zanimajo le trije tipi naprav:

- CPE,
- GPE,
- pospeševalniki.

Vsakemu tipu naprave pripišemo uteži kot konstanto v gostitelju. Uteži določimo tako, da ocenimo kako primerne so arhitekture naprav za naš problem. Ta delitev ima zopet prednost preprostega programa na gostitelju. Če nastavimo ugodne uteži, lažje dosežemo večjo hitrost in izkoristek naprav. Če uteži nastavimo nepravilno, se pri daljših izračunih zelo lahko zgodi, da hitrejše naprave končajo opazno pred počasnejšimi. Problem je tudi pravilno oceniti uteži, še posebej če nismo dobro seznanjeni s strojno opremo na kateri delujemo.

4.2.1 Utežena delitev s preizkusom

Preproste delitve niso zadovoljive, saj temeljijo na ugibanjih o zmogljivosti strojne opreme ali pa je sploh ne upoštevajo. Najboljša rešitev se zdi, da zmogljivost naprav sami preizkusimo in se nato na podlagi rezultatov odločimo o delitvi dela. Strojno opremo preizkusimo na majhnem deležu problema. Ko je preizkus končan, pridobimo čas dela naprave. S primerjavo teh časov napravam določimo uteži, nato pa glede nanje razdelimo preostalo

delo. Taka porazdelitev je zadovoljiva, dokler vsi ščepci potrebujejo približno enako ukazov, da dokončajo svoje delo in dokler je naš program edini, ki teče na sistemu. Če poleg našega programa teče še kak drug zahteven program, ki uporablja katero od izbranih naprav, se lahko zgodi, da je izračunana utež zanjo povsem napačna. Utež prav tako lahko napačno določimo v primeru, da vsi ščepci niso enako zahtevni. V tem primeru se lahko zgodi, da katera od naprav dobi znatno lažje delo od ostalih in tako dobi utež, ki ne odraža dejanskega stanja naprave. Paziti moramo, da začetni preizkus traja tako dolgo, da polno izkoristi naprave, a je dovolj kratek, da naprave ne čakajo predolgo, da ostale naprave končajo s testiranjem.

4.3 Dinamična delitev blokov

Rešitev, ki omili probleme pri delitvi dela napravam, je dinamična delitev dela. Kot pri prejšnji rešitvi, na začetku dodelimo majhen del potrebnega dela vsem napravam. Ko naprave končajo, s pomočjo njihovih časov računanja, dodelimo napravam novo delo. Za razliko od ostalih pristopov tu dodelimo napravi le delež celotnega dela, ki mu pravimo blok. Vsakič ko naprava konča s svojim delom, jo ponovno ocenimo in ji dodelimo nov primerno velik blok. Tako se izognemo problemom z napačno ocenjenimi utežmi zaradi ostalih aplikacij.

Taka oblika dodeljevanja dela potrebuje spremembo preverjanja in ravnanja z dogodki. Pri ostalih načinih dodeljevanja smo lahko na dogodke čakali tako, da smo uporabili funkcijo `clWaitEvents()`. Ta ukaz blokira izvajanja gostiteljskega programa, dokler se vsi dogodki, ki jih podamo funkciji, ne izvedejo. To sprosti CPE, ki izvaja gostiteljski program, za izvajanje ščepca. Žal pa OpenCL ne podpira funkcije, ki bi blokirala izvajanje le do sprožitve prvega dogodka. Zato je potrebno, da se gostitelj vrti v zanki in pregleduje stanje dogodkov. Ko opazi, da se je na neki napravi izračun zaključil, ji lahko dodeli novo delo.

Alternativa utežem je lahko tudi, da velikost blokov določimo s pomočjo

časovnih intervalov. Na začetku programa določimo nek časovni interval T, ki se lahko v teku programa tudi zvišuje. Vsakič, ko naprava obdela svoj blok, izračunamo koliko dela potrebuje, da bo naslednji blok izračunala približno v času T. To nam dovoljuje, da določimo časovne intervale po katerih se naprave zopet ocenijo.

Pri tej delitvi dela je velika slabost to, da v primeru da težavnost problemske funkcije ni enakomerno porazdeljena, naše ocene ne temeljijo samo na hitrosti opravljenega dela, ampak tudi na težavnosti dela, ki pa nam je med tekom programa neznana. Zato ocene naprav in njihovih uteži postanejo nezanesljive.

4.4 Preprosta delitev blokov

Podobna rešitev kot dinamična delitev blokov je preprosta delitev blokov. Pri takšni delitvi razdelimo bloke med naprave. Vsakič, ko naprava konča, ji dodelimo nov blok dela. Vendar za razliko od dinamične delitve tu velikosti blokov ne spreminjamo, ampak jo določimo na začetku in je enaka za vse naprave. Delo se porazdeli enakomerno, saj hitrejše naprave večkrat prevzamejo bloke in tako opravijo več dela. Tako se izognemo problemom, ko naprave dobijo različno težko delo in nato dobijo neprimerne velikosti dela. Če na koncu najpočasnejša naprava še vedno zaostaja, lahko njeno delo dodelimo prosti hitrejši napravi. Tista, ki konča prej, zapiše podatke v pomnilnik, nato pa obe napravi zaključita z delom. To pomaga le, če so hitre naprave nekajkrat hitrejše od počasnih. Taka delitev je lahko uporabna tudi, če želimo da gostitelj opravi manj dela. V našem primeru to ni bilo tako pomembno, saj je bila naprava gostitelja več kot desetkrat šibkejša od ostalih naprav.

Poglavje 5

Implementacija aplikacije

5.1 Potek programa

5.1.1 Priprava gostitelja

Pred delitvijo dela med napravami je potrebno vzpostaviti ogrodje za delo. Delitev dela opravljamo na CPE. Poleg tega, da razdeljuje delo, je CPE prav tako računska naprava, ki sama sebi dodeli delo.

Najprej pridobimo vse platforme, ki nam jih OpenCL ponudi. Naložimo tudi izvorno kodo ščepca iz datoteke. Nato je potrebno iz vsake platforme pridobiti vse naprave, ki so na voljo. Na tem koraku lahko poimensko izločimo naprave, za katere ne želimo, da sodelujejo pri računanju. To je uporabno, če vemo, da je kakšna naprava zasedena z drugimi opravili zunaj našega programa in ne želimo, da to ovira delovanje programa. Pri izločanju naprav je potrebno tudi preveriti, ali med napravami na različnih platformah obstajajo duplikati, in jih odstraniti. Standard OpenCL zagotavlja, da se ista naprava ne pokaže več kot enkrat na eni platformi. Če se naprava pojavi v dveh platformah, jo lahko zaznamo tako, da njeno ime primerjamo z imeni ostalih naprav. Če je ime enako, je naprava duplikat.

Za vsako platformo ustvarimo kontekst z napravami, ki ji pripadajo, za vsak kontekst pa ustvarimo tudi programski objekt, ki ga prevedemo in

povežemo.

Za vsako napravo, ki jo uporabljamo, ustvarimo objekt imenovan delovna naprava (ang. *working device*). Ta objekt vsebuje identifikacijsko številko naprave, trenutne ukaze naprave, ukazno vrsto, programski objekt, pomnilnik, ščepec, kazalec na njen kontekst in podatke o napravi sami. Ko ustvarimo ta objekt, nam ni več treba skrbeti v kateri kontekst ali platformo spada naprava. Vse naprave lahko od tu naprej obravnavamo enako.

V naših testnih primerih, ko smo imeli na voljo dve Tesli, Xeon Phi in Xeon E5-2620, nam je sistem ponudil dve platformi, eno proizvajalca Nvidia, ki je vsebovala obe Tesli in eno proizvajalca Intel, ki je vsebovala Xeon Phi in Xeon E5-2620. Zato je bilo potrebno ustvariti dva konteksta.

Ko je ogrodje pripravljeno, izračunamo začetne parametre za računske ukaze. Računski ukazi so podatkovna struktura na gostitelju, ki smo jo ustvarili za lažje upravljanje z nalogami, ki jih izvajajo naprave. Ta struktura je drugačna za vsak algoritem. V splošnem vsebuje podatke o delu, ki je dodeljeno napravi, ter kazalce na vhodne podatke in kazalce na pomnilnik za izhodne podatke.

5.1.2 Delitev dela

Za vsako napravo je za začetek delitve dela potrebno pripraviti naslednje strukture OpenCL:

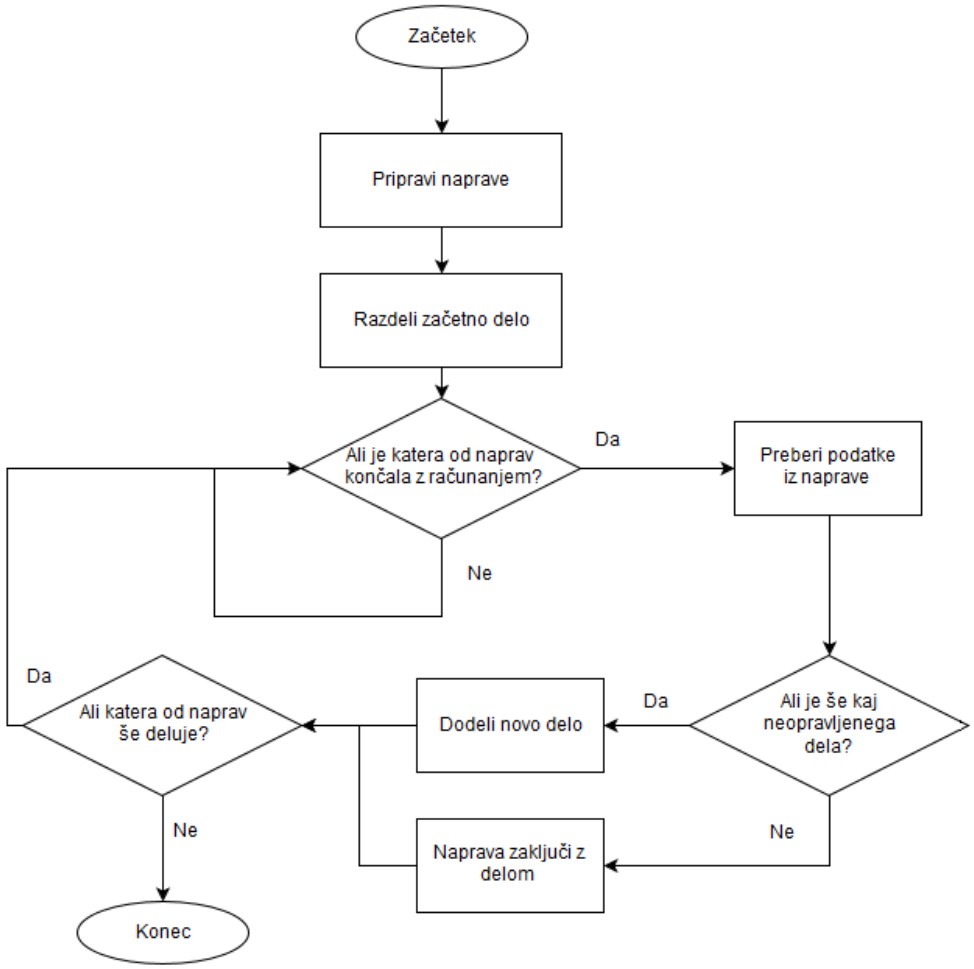
- ukazno vrsto,
- ščepec,
- dva dogodkovna objekta,
- pomnilniški objekt.

Pred začetkom izvajanja moramo določiti začetne argumente za ščepec. Ti veljajo za vse niti, ki se izvajajo na napravi.

Pri enakomerni delitvi dela razdelimo naš problem na D kosov, kjer D predstavlja število naprav v sistemu, nato pa vsaki napravi v ukazno vrsto uvrstimo ukaz za izvajanje ščepca s funkcijo `clEnqueueNDRangeKernel()`. Po zagonu ščepcev moramo v vrsto uvrstiti še ukaz za branje iz medpomnilnika na napravi v glavni pomnilnik s funkcijo `clEnqueueReadBuffer()`. Ta dva ukaza povežemo z dogodkovnima objektoma, ki ju uporabimo zato, da lahko ugotovimo, kdaj je naprava končala z delom in koliko časa je za delo porabila. Prvi dogodkovni objekt je namenjen sledenju izvajanja ščepcev, drugi pa sledenju prenosa podatkov v glavni pomnilnik. Ko so ukazi uvrščeni v vrstah, moramo počakati na konec njihovega izvajanja. To storimo z ukazom `clWaitForEvents()`, ki povzroči, da gostitelj zaspi, dokler se ne sprožijo dogodki, ki sledijo ukazom. Ko se vsi ukazi izvedejo do konca, izmerimo čas izvajanja in zaključimo.

Če delo delimo s pomočjo uteži, se moramo odločiti za velikost dela, na katerem bomo testirali naprave. Temu delu pravimo testni blok. Testni blok ponavadi obsega od 1 do 5 procentov celotnega dela. Testni blok razdelimo na D enakih delov, ki jih dodelimo napravam. Nato z ukazom za čakanje dogodkov počakamo, da vse naprave zaključijo z delom. Tedaj iz dogodkovnih objektov dobimo podatke o času računanja in pisanja v pomnilnik. Za vsako napravo nato količino opravljenega dela delimo s časom računanja. Izračun predstavlja hitrost naprave, ki jo uporabimo kot utež. Z utežmi naprav lahko določimo porazdelitev preostalega dela. Preden uvrstimo nove ukaze je potrebno obnoviti dogodkovna objekta in preveriti, ali pomnilniški objekt dovolj velik za preostalo delo. Nato v ukazne vrste uvrstimo ukaze za izračun preostalega dela. Nato zopet počakamo, da se sprožijo vsi dogodki, izmerimo čas in zaključimo.

Diagram poteka delitve dela z bloki vidimo na sliki 5.1. Pri takih delitvah program ne more čakati na naprave s preprostimi ukazi, ampak mora program uporabiti povpraševanje. Zato moramo delujočim napravam dodeliti zastavice, ki nam povedo, ali so naprave zaključile z delom. Naprava zaključi z delom, ko nima dodeljenega bloka, hkrati pa ni več dela, ki bi ga bilo po-



Slika 5.1: Diagram poteka delitve dela z dodeljevanjem blokov

trebno opraviti. Ko program razpošlje ukaze za začetek dela, čaka v zanki, da vse naprave zaključijo z delom. V tej zanki s funkcijo `clGetEventInfo()` pregleduje dogodke vseh naprav in za vsako napravo preveri, ali je že zaključila z delom. Ko funkcija vrne vrednost `CL_COMPLETE`, program ve, da je delo končano. Iz naprave prebere podatke in, če je potrebno opraviti še kaj dela, vstopi v funkcijo za dodelitev dela.

Če delimo delo s preprostim dodeljevanjem blokov, je funkcija za dodelitev dela preprosta. Napravi dodeli tako velik blok, kot je določeno na začetku programa. Če je blok večji od preostalega dela, dodelimo celoten preostanek.

Če delimo delo na bloke dinamične velikosti, je potrebno določiti velikost bloka. Najprej je potrebno iz dogodkov pridobiti podatke o času izvajanja prejšnjega bloka in nato, kot pri delitvi z utežmi, izračunamo hitrost naprave. Pred začetkom izvajanja programa določimo časovni interval T. Velikost naslednjega bloka določimo tako, da s pomočjo hitrosti naprave izračunamo potrebno velikost bloka, da bo naprava zaposlena za T sekund.

Po izstopu iz funkcije za dodelitev dela mora program najprej obnoviti dogodkovne objekte, preveriti, da ima dovolj rezerviranega pomnilnika za delo na napravi, nato pa dodeliti delo napravi. Zatem se vrne na čakanje v zanki. Zanka na koncu vsake iteracije zaspi za nekaj milisekund, zato da gostiteljski program ne preobremenjuje CPE z nepotrebnim delom. Ko vse naprave zaključijo z delom, se zanka prekine in zaključimo z izvajanjem programa.

5.2 Upravljanje s pomnilnikom

Pomnilniški objekti na gostitelju predstavljajo pomnilnik, ki ga uporabljajo računske naprave. Vsaka naprava ima dodeljen svoj pomnilniški objekt. Ker vsak tak objekt predstavlja fizični pomnilnik, to pomeni, da ustvarjanje novih pomnilniških objektov sproži postopke dodeljevanja pomnilnika aplikaciji. Če ob vsakokratni dodelitvi dela sprostimo star pomnilnik in dodelimo novega, lahko pride do upočasnitev gostiteljskega programa. Dodeljevanje in sproščanje pomnilnika aplikacijam je namreč na nekaterih napravah draga operacija, ki je ne želimo ponavljati. Če je pomnilnika na napravi veliko, lahko na začetku programa napravi dodelimo veliko pomnilnika, ki ga nato uporablja do konca programa. Vendar to ni dobra strategija v primeru, da aplikacija potrebuje veliko pomnilnika in si naprave deli z drugimi aplikacijami. Če je namreč na napravi dodeljenega preveč pomnilnika, se lahko pomnilnik začne preslikovati na disk, ali pa naprava ne pusti več dodeljevanja spomina in postane neuporabna. V takih primerih moramo omejiti velikost dela, ki ga dodelimo napravam, kar lahko posledično pomeni počasnejše

računanje.

5.3 Velikost globalnega razpona in delovnih skupin

V splošnem moramo naš problem razdeliti na veliko število točk v eno ali večdimenzijskem prostoru. Za vsako točko našega problema moramo zagnati svojo nit. Če želimo sami izbrati velikost delovnih skupin, je možno, da moramo zagnati kakšno nit več, zaradi omejitev pri velikost lokalnega razpona. Velikost delovnih skupin je predvsem pomembna na GPE, saj so delovne skupine implementirane v strojni opremi.

V pomnilniku so točke razvrščen tako, da so si sosednje, če so njihove realne komponente sosedne. To je pomembno na napravah kot je Xeon Phi, ki za hitro delovanje uporablja predpomnilnik. Na to GPE ni tako pomembno, saj GPE navadno ne uporabljajo veliko predpomnilnika.

Če želimo določiti velikost delovnih skupin, potrebujemo nekaj podatkov o napravah. S funkcijama `clGetKernelWorkGroupInfo()` in `clGetDeviceInfo()` lahko pridobimo nekaj predlogov. Prva funkcija nam pri pravih argumentih vrne vrednost M. Velikost delovne skupine na izbrani napravi mora biti mnogokratnik vrednosti M. Če velikost globalnega razpona ni deljiva z velikostjo naše delovne skupine, lahko velikost globalnega razpona v dimenziji, ki je neustrezna, povečamo za toliko, da postane deljiva. Vse dodatne niti, ki jih s tem ustvarimo, problema ne računajo ampak iz programa takoj izstopijo.

Velikost delovnih skupin navadno ne sme biti enaka M. To število je namig, ki na različnih naprava pomeni različne stvari. Na primer: na napravah proizvajalca Nvidia ta funkcija skoraj vedno vrne 32. To je v CUDA velikost snopa (ang. *warp*), kar je število niti, ki se hkrati izvajajo na eni računski enoti GPE, ki pa ni nujno enaka računski enoti OpenCL. Na GPE proizvajalca AMD je ekvivalent snopa *wavefront*, in normalno sestoji iz 64 niti. OpenCL ekvivalenta temu konceptu nima, vendar se morajo progra-

merji kljub temu zavedati te lastnosti, saj lahko le tako dosežejo dobro izrabo naprav.

Dejanska primerna velikost delovne skupine je na GPE nekajkrat večja kot M, ponavadi vsaj 128, lahko pa tudi več. Na CPE velikost delovnih skupin ni tako pomembna, saj arhitekture CPE niso tako podobne modelu OpenCL kot arhitekture GPE. Za velikost delovnih skupin na Xeon Phi proizvajalec svetuje, da so mnogokratnik števila 16, ali pa da to odločitev prepustimo gonilniku [10].

Iz naprav lahko tudi izvemo število njihovih računskih enot. Vendar nam to zopet ne pove veliko, saj računske enote pri različnih proizvajalcih pomenijo različne stvari. Na primer: na napravah proizvajalca Intel je število računskih enot enako številu navideznih jeder, ki so na voljo; na GPE proizvajalca Nvidia pa nam to število pove, koliko procesorjev SP (ang. *Streaming Multiprocesor*) je dostopnih. Na vsakem procesorju SP lahko teče več sto niti naenkrat. Iz števila računskih enot lahko ocenimo, koliko delovnih skupin najmanj potrebujemo, saj želimo, da na GPE vsaka računska enota dobi vsaj 4 do 8 delovnih skupin. To število skupin ponavadi z lahkoto presežemo, zato ta podatek ni tako uporaben.

Če velikosti delovnih skupin ne določimo, jih pred zagonom ščepca določi gonilnik sam. Gonilnik velikosti delovnih skupin izbere le glede na arhitekturo, kar ponavadi prinese zadovoljive rezultate. Vendar se na gonilnik ne smemo preveč zanašati, saj lahko zaradi nepoznavanja našega ščepca, ali pa zaradi neprimernosti našega globalnega razpona, izbere povsem neustrezne velikosti, kar pripelje do slabe izrabe naprav. Če se na primer zgodi, da sta velikosti obeh dimenzij globalnega razpona praštevili in pustimo odločitev o velikosti delovnih skupin gonilniku, ta velikosti delovnih skupin nastavi na 1. V tem primeru je izkoristek GPE zelo slab.

Če želimo GPE čim bolje izrabiti, je bolje, da izbire velikosti delovnih skupin ne prepustimo gonilniku. Pomembno je, da je velikost delovnih skupin res mnogokratnik vrednosti M.

5.4 Problemi pri razvoju splošne delitve

Če na našem sistemu aplikacija ni sama, ampak si rabo naprav deli z drugimi aplikacijami, moramo poskrbeti, da naša aplikacija naprave še vedno čim bolje izkoristi. Tak problem je težko rešiti v splošnem. Nekatere naprave imajo gonilnike, ki dopuščajo deljenje procesorske moči med programi. To ponavadi velja za vse CPE. Več težav imamo z gonilniki za GPE, saj nekateri ne dopuščajo sočasnega izvajanja ščepcev iz različnih kontekstov. Na primer: OpenCL je na GPE proizvajalca Nvidia implementirana kot ovojnica okoli platforme CUDA, ki pa po standardu ne dopušča sočasnega izvajanja ščepcev iz različnih računskih kontekstov [17]. Ko želimo ustvariti nov kontekst na taki GPE, funkcija `clCreateContext()` čaka, dokler naprava ni prosta. OpenCL ne omogoča zaznavanja takih situacij, zato je potrebno pred zaganjanjem aplikacije preveriti trenutno stanje naprav z orodji kot so `nvidia-smi`, torej zunaj aplikacije.

Razvoj splošnih rešitev otežuje tudi to, da imajo različni gonilniki pomanjkljivosti v svojih implementacijah OpenCL. Naš razvoj je predvsem oviralo to, da gonilnik Tesle K20m ne podpira asinhronega branja iz pomnilnika. Pri funkciji `clEnqueueReadBuffer()`, tudi ko ima `blocking_read` vrednost CL_FALSE, lahko gonilnik Tesle funkcijo vseeno izvede sinhrono. To pomeni, da je potrebno v vrsto uvrstiti ukaz branja šele po koncu izračuna na napravi. To upočasni delovanje, saj bi drugače lahko gostitelja zbudili šele, ko bi bili podatki že prepisani.

Poglavje 6

Rezultati

Našo aplikacijo smo testirali z računanjem pripadnosti Mandelbrotovi množici. Množica je ugodna za testiranje, saj je preprosta za izračun, izračun množice pa lahko poteka na vseh točkah povsem neodvisno.

6.1 Mandelbrotova množica

Mandelbrotova množica je množica števil c v kompleksni ravnini, za katere velja, da je neskončno zaporedje, podano kot $z_{n+1} = z_n^2 + c$ omejeno ($n \in \mathbb{N}, z_0 = 0$) . Izkaže se, da zaporedje divergira, če je element zaporedja $|z_n| > 2$, zato v tem primeru c ni element Mandelbrotove množice.

Algoritem 1 prikazuje psevdokodo za določanje pripadnosti množici. Računanje členov zaporedja ponavljamo toliko časa, da z preseže 2. Pri elementih množice bi to pomenilo, da bi računanje teklo v nedogled. S konstanto `število_iteracij`, ki se pojavi v algoritmu, določimo, kolikokrat največ bomo ponovili računanje. To pomeni, da lahko pri nekaterih točkah iz zanke izstopimo veliko hitreje kot pri drugih, kar posledično pomeni, da se čas računanja na različnih točkah lahko zelo razlikuje.

Kot lahko vidimo iz definicije, je vsaka točka povsem neodvisna od drugih. To pomeni, da je problem trivialno paralelen, saj lahko za vsako točko neodvisno poženemo izračun in da lahko izračuni potekajo v naključnem vr-

Algoritem 1 Pripadnosti točke mandelbrotovi množici

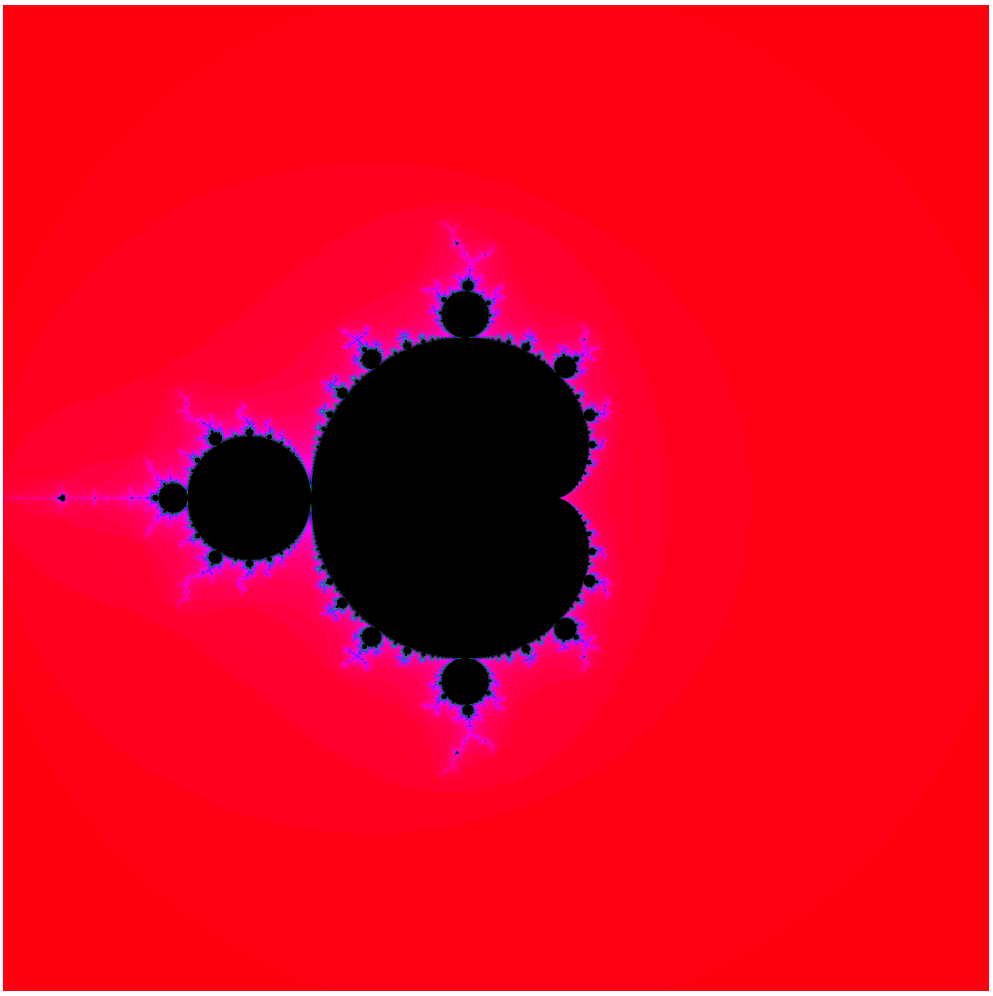
```
c ← trenutna točka
z ← 0
iter ← 0
for iter < število_iteracij do
    z ← z^2 + c
    if |z| > 2 then
        return true
    end if
end for
return false
```

stnem redu, če je to potrebno. Ker smo pri nalogi imeli na razpolago različno hitre naprave, je to zelo pomembna lastnost.

6.1.1 Implementacija algoritma

Množico smo računali na dvodimenzionalnem kvadratnem razponu od -2,0 do 2,0 in od -2,0i do 2,0i. V smeri realne osi smo ga razdelili na X delov, v smeri imaginarne osi pa na Y delov in izračunali pripadnost množici na vseh ogliščih te mreže $x + iy$. Tu x predstavljajo realne, y pa imaginarne komponente kompleksnih števil pri katerih teče račun. Pri risanju slike množice vsaka točka predstavlja en piksel izhodne slike, barvo piksla pa določa število iteracij, potrebnih za izstop iz zanke. Primer množice, naslikan z programom vidimo na sliki 6.1. Za vsako točko razpona smo zagnali svojo nit.

Pri testiranju aplikacije se je pokazalo, da s povečevanjem števila točk za izračun, ozko grlo postane prepustnost pomnilnika. Ker smo želeli, da so testi računsko dovolj zahtevni, smo vsako izbrano točko v množici nadalje razdelili še na več točk, ki smo jih nato povprečili. Ta povprečja so tako določila ali točka pripada množici ali ne. Vsaka nit je tako izračunala vrednosti več točk in barvo le enega piksla. Tak postopek računanja v nadaljevanju imenujemo *normalen*.



Slika 6.1: Izris Mandelbrotove množice

Ker smo želeli testirati našo aplikacijo tudi na problemu, kjer je težavnost računanja porazdeljena enakomerno, smo poleg normalne verzije implementirali tudi ne-optimiziran algoritem, ki tudi v primeru, da je kriterij za divergenco izpolnjen, izvede vse iteracije do konca. Takemu postopku računanja v nadaljevanju imenujemo *enakomeren*.

6.2 Meritve

Meritve smo opravili s 100.000.000 nitmi, kar predstavlja dvodimenzionalni globalni razpon 10.000 x 10.000. Merili smo na obeh verzijah postopka računanja. Opravljena je bila tudi daljša meritev pri normalnem postopku za testiranje dinamične in preproste delitve blokov na težjem problemu. Testni parametri vseh testiranj so zapisani v tabeli 6.1.

Tabela 6.1: Parametri testov

parameter	normalen	enakomeren	daljša meritev
število iteracij	1000	1000	1000
število točk na ščepec	1600 (40^2)	324 (18^2)	10000 (100^2)

Za primerjavo hitrosti naprav so v tabeli 6.2 časi računanja celotnega problema na samo eni napravi. Razmerje med hitrostmi naprav je podobno kot razmerje med njihovimi optimalnimi FLOPS, ki so zapisani v razdelku 3.3. Iz te tabele lahko ocenimo največjo računsko hitrost sistema, ki pove koliko časa bi bilo potrebno za rešitev problema, če bi problem razdelili idealno. To izračunamo tako, da izmerimo hitrost izračuna ene vrstice razpona, nato iz hitrosti določimo idealno razdelitev dela, nato pa od tu potreben čas dela. Rezultati izračunov so predstavljeni v tabelah 6.3 in 6.4.

Tabela 6.2: Časi računanja mandelbrotove množice s samo eno napravo

naprava	normalen [s]	enakomeren [s]
Tesla K20m	393,0	605,3
Xeon Phi	814,4	1114,3
Xeon E5-2620	6075,8	5422,5

Tabela 6.3: Optimalna razdelitev dela, normalen postopek

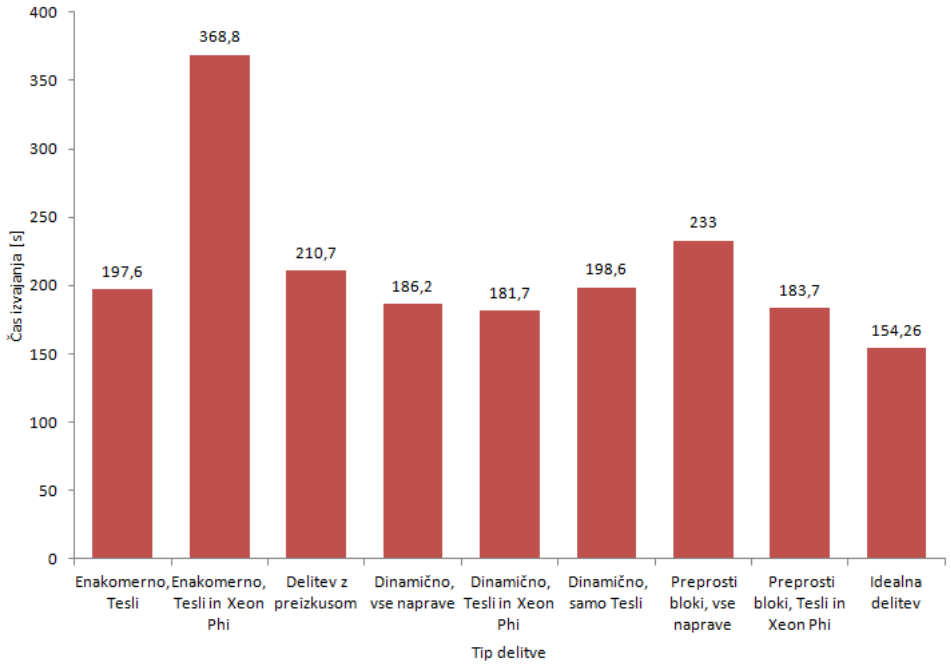
naprava	število vrstic na sekundo	velikost dela [%]	pričakovan čas [s]
Tesla K20m	25,44	39,2	154,3
Tesla K20m	25,44	39,2	154,3
Xeon Phi	12,28	18,9	154,3
Xeon E5-2620	1,64	2,5	154,3

Tabela 6.4: Optimalna razdelitev dela, enakomeren postopek

naprava	število vrstic na sekundo	velikost dela [%]	pričakovan čas [s]
Tesla K20m	16,52	37,7	228,0
Tesla K20m	16,52	37,7	228,0
Xeon Phi	8,97	20,5	228,0
Xeon E5-2620	1,84	4,2	228,0

Tabela 6.5: Časi računanja Mandelbrotove množice glede na delitev

porazdelitev	naprave	normalen [s]	enakomeren [s]
Enakomerno	Tesli	197,6	284,9
Enakomerno	Tesli in Xeon Phi	368,8	355,3
Delitev s preizkusom	vse	210,7	243,1
Dinamično	vse	186,2	232,7
Dinamično	Tesli in Xeon Phi	181,7	233,7
Dinamično	Tesli	198,6	290,0
Preprosti bloki	vse	233,0	260,6
Preprosti bloki	Tesli in Xeon Phi	183,7	236,8



Slika 6.2: Graf časov računanja Mandelbrotove množice, normalen postopek, glede na delitev

6.3 Komentar rezultatov

6.3.1 Enakomerni postopek

Če v tabeli 6.5 primerjamo rezultate testov, ki uporabljajo samo Tesli, vidimo, da je dinamično porazdeljevanje blokov le nekoliko počasnejše od enakomerne delitve na dva dela. To je zelo ugoden rezultat, saj pokaže, da dinamična delitev blokov deluje zelo učinkovito in je v dolgotrajnih izračunih zelo uporabna.

Če je problemska funkcija enakomerna, naše naprave pa imajo različne moči, lahko iz rezultatov na sliki 6.3 vidimo, da je dinamično dodeljevanje blokov zelo učinkovito, saj se zelo približa idealni delitvi. Ta način dodeljevanja deluje dobro tudi, kadar eno od naprav vmes obremeni drug program. Pri njej je potrebno določiti približno trajanje blokov, torej interval T. Kadar pričakujemo, da bo izračun celotnega problema trajal nekaj minut, je

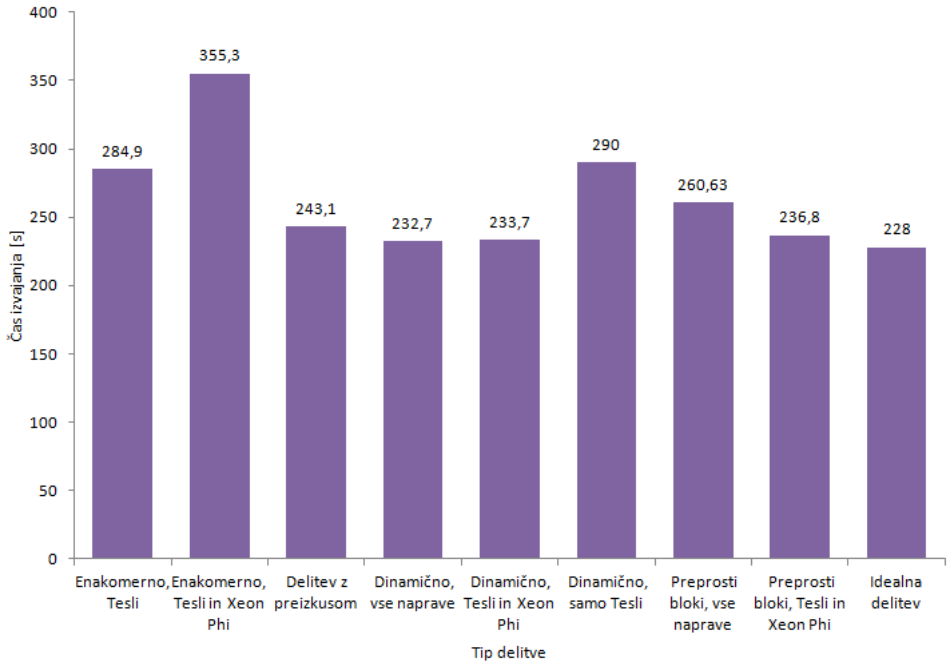
ugoden čas T nekje od 10 do 20 sekund, za daljše izračune pa lahko tudi več minut. Ker je opravilo razdeljevanja blokov v vsakem primeru zelo kratko, je dobro, da bloki ne trajajo predolgo, saj želimo čim prej opaziti napake v utežeh naprav.

Idealni delitvi se približa tudi utežena delitev s preizkusom. Vendar je pri interpretaciji teh rezultatov treba biti previden, saj so lahko zavajajoči. Implementacija delitve, ki smo jo testirali, je preveč prilagojena testni strojni opremi. Izkušnje iz testiranja kažejo, da ima GPE Tesla K20m počasnejši zagonski čas in da deluje hitreje na večjih problemih. Meritve časa iz začetnega testiranja so zato Teslam namenile premajhen kos dela. To smo popravili tako, da smo namesto časov pri računanju, upoštevali kvadrate časov. Tako hitre naprave dobijo še več dela v primerjavi s počasnimi. Te spremembe so preveč specifične in bi lahko škodovale v splošnem primeru. V splošnem je ta problem težko rešljiv, saj ne moremo napovedati, kakšne so razlike v zmogljivosti naprav, glede na velikost problema.

Od delitev, ki ne upoštevajo zmogljivosti strojne opreme, ampak še vedno uporabijo vse naprave, je najboljša preprosta delitev z bloki. Ostale delitve so neprimerno počasnejše, saj hitrejše naprave vedno čakajo na počasnejše. Vendar je tudi ta delitev slabša od dinamične delitve blokov. Pri tej delitvi je še bolj pomembno, da pravilno določimo velikost blokov. Pri krajših računanjih se velikokrat zgodi, da hitre naprave na koncu čakajo počasnejše, če so bloki preveliki. To ni tako velik problem ko je razmerje med velikostjo izračuna in velikostjo bloka veliko. Bloki tudi ne smejo biti premajhni, saj to pomeni, da naprave niso dovolj izkoriščene.

6.3.2 Normalni postopek

Pri neenakomerni problemski funkciji iz tabele 6.5 vidimo glavno slabost enakomerne porazdelitve. Razlika med uporabo samo dveh Tesel, ki si enakomerno razdelita problem in uporabo Tesel in Xeon Phi, kjer je problem neenakomerno porazdeljen med neenake naprave, je ogromna. Na sliki 6.2 vidimo tudi, da se delitve dosti manj približajo idealni delitvi problema, kot



Slika 6.3: Graf časov računanja Mandelbrotove množice, enakomeren postopek, glede na delitev

pri enakomerni funkciji.

Pri taki problemski funkciji se zaradi variacij v težavnosti funkcije velikokrat zgodi, da dobimo boljše rezultate, če ne uporabimo zelo počasnih naprav. Lahko se namreč zgodi, da naprava dobi preveliko utež in jo nato druge naprave čakajo zelo dolgo časa. Tak primer vidimo v rezultatih pri obeh delitvah blokov. Če primerjamo dinamično dodeljevanje brez uporabe CPE in tisto brez CPE in Xeon Phi, vidimo da s Xeon Phi pridobimo le 20 sekund, kar je okoli 10%. Primerjavo med dinamičnim in preprostim dodeljevanjem blokov, prikazuje tabela 6.6. V njej vidimo, da se pri daljših računanjih bolj obnese dinamično porazdeljevanje z vsemi napravami. Pri porazdeljevanju preprostih blokov vidimo, da CPE tudi pri daljših izračunih upočasni delo.

Tabela 6.6: Časi računanja Mandelbrotove množice, daljša meritev

porazdelitev	naprave	normalen [s]
dinamično	vse	1046,7
dinamično	Tesli in Xeon Phi	1091,0
preprosti bloki	vse	1261,7
preprosti bloki	Tesli in Xeon Phi	1171,2

6.3.3 Velikosti delovnih skupin

Tabela 6.7: Časi dinamične porazdelitve z enakomernim postopkom, glede na velikost delovnih skupin, vse naprave

razpon	enakomeren [s]
32x1	340,1
64x1	339,8
128x1	227,8
192x1	227,8
256x1	227,8
512x1	228,4
Ni nastavljen	232,2

Tabeli 6.7 in 6.8 prikazujeta hitrosti izračuna pri dinamični porazdelitvi pri enakomernem postopku glede na velikost in obliko delovnih skupin na GPE. Rezultate vidimo tudi na sliki 6.4. Iz tabele 6.7 lahko sklepamo, da je zelo pomembno, da so delovne skupine dovolj velike, da učinkovito izkoristijo GPE. Pri GPE proizvajalca Nvidia je zaradi zasnove, ki dopušča višjo zakasnitev pri pomnilniških dostopih, pomembno, da je v vsaki skupini na čakanju zadostno število niti, čeprav se jih sočasno iz ene skupine izvaja le 32. Iz testov vidimo, da zadostuje, da je velikost delovne skupine vsaj 128 niti. Navodila proizvajalca Nvidia omenjajo, da je priporočena velikost delovnih skupin 128-256 za polno zasedenost blokov in nad 192 za zakritje zakasnitve

Tabela 6.8: Časi dinamične porazdelitve z enakomernim postopkom, glede na obliko delovnih skupin

razpon	enakomeren [s]
128x1	227,8
64x2	238,5
32x4	237,8
16x8	238,4
8x16	228,7
4x32	231,7
2x64	234,0
1x128	242,5
20x10	248,4
43x13	239,0
257x1	255,5

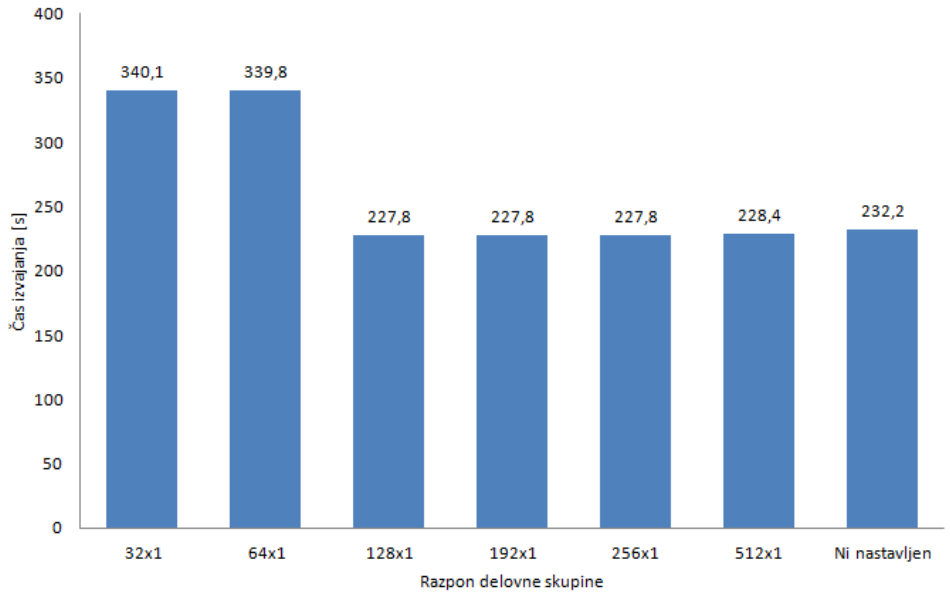
pri dostopih do pomnilnika. Ker naš ščepec dostopa do pomnilnika le na koncu izvajanja, ta vrednost za nas ni tako pomembna. Vidimo tudi, da je v teh primerih ročno nastavljanje velikosti delovnih skupin boljše kot če pustimo, da velikost določi gonilnik.

Še huje je v nekaterih posebnih primerih. Na primer: če je velikost obeh dimenzij globalnega razpona praštevilo, gonilnik nima druge možnosti, kot da velikost delovnih skupin nastavi na 1. Takrat postanejo GPE povsem neuporabne, saj je naša aplikacija na istem problemu z razponom 10.007 x 10.007 potrebovala 5025 sekund, kar je približno dvajsetkrat dlje od časa, potrebnega za delo na razponu 10.000 x 10.000. To je seveda preprosto rešljiv problem, saj lahko globalni razpon v takem primeru razširimo, podobno kot je to opisano v razdelku 5.3.

V tabeli 6.8 so predstavljeni rezultati glede na obliko delovnih skupin. Opazimo, da so razlike med enako velikimi skupinami zelo majhne, le okoli 5%. Vidimo pa tudi, da izračun traja dlje, kadar velikost delovnih skupin ni

deljiva z 32.

Iz teh rezultatov vidimo, da je za delo GPE potrebno, da med dodeljevanjem, odločitev o velikosti delovnih skupin ne prepuščamo gonilniku ampak jo določimo sami. Rezultati kažejo, da je primerna velikost za delovne skupine 192 ali 256. Iz rezultatov se tudi kaže, da je najboljša oblika delovne skupine enodimenzionalna in vsebuje točke, ki so v pomnilniku sosednje, torej velikost 256x1.



Slika 6.4: Graf časov dinamične porazdelitve z enakomernim postopkom, glede na velikost delovnih skupin na GPE.

Poglavje 7

Zaključek

V delu smo zasnovali in razvili aplikacijo za delitev dela na heterogenih sistemih. Omogoča učinkovito izrabo poljubnih heterogenih sistemov. Aplikacijo bi bilo preprosto pretvoriti za reševanje drugih vzporednih problemov, saj je napisana modularno, ne pa specifično za naš testni primer.

Prikazali smo nekaj možnih načinov deljenja dela, ki bi bili uporabni za različne vrste problemov. Ugotovili smo, da je najboljše deliti problem z dinamično delitvijo, ki napravam dodeli manjše kose dela, nato pa jim na podlagi njihovih časov računanja dodeli novo delo. Taka delitev je izboljšala hitrost delovanja tudi za več kot 25 % v primerjavi z drugimi delitvami. Rezultati so pri problemih, kjer težavnost ni enakomerna, pokazali tudi, da uporaba počasnih naprav velikokrat upočasni delo, namesto da bi ga pohitrila. Ugotovili smo tudi, da odločitve glede velikosti delovnih skupin ni pametno prepustiti gonilniku temveč je bolje, da jih nastavimo sami. Pokazali smo primerne velikosti delovnih skupin za grafične procesne enote proizvajalca Nvidia. Pregledali smo tudi, kakšne oblike delovnih skupine so primerne in ugotovili, da so najboljše delovne skupine enodimenzionalne ter vsebujejo točke, ki pišejo v sosednje naslove v pomnilniku.

Kljub temu, da je naše orodje namenjeno splošni uporabi na kateremkoli heterogenem sistemu, bi bilo potrebno izvesti več testiranj na različnih heterogenih sistemih. Tega nismo storili, saj smo imeli dostop le do enega sistema,

ki je vseboval več kot le centralno procesno enoto in eno grafično procesno enoto. Dodatno bi bilo potrebno implementirati in testirati možne optimizacije za Xeon Phi kot so prilagoditev globalnih in lokalnih razponov, ter oblike delovnih skupin. Prav tako bi bilo potrebno testirati še grafične procesorje drugih proizvajalcev in prilagoditi pripravo naprav, da bi upoštevala tudi njihove lastnosti in prednosti. Ni mogoče, da bi naša aplikacija pokrila vse naprave, lahko pa bi bolj podrobno poznala in upoštevala lastnosti vsaj še nekaj bolj razširjenih arhitektur.

Literatura

[1] Apple. Opencl: Taking the graphics processor beyond graphics.

[2] Sze-Hang Chan, Jeanno Cheung, Edward Wu, Heng Wang, Chi-Man Liu, Xiaoqian Zhu, Shaoliang Peng, Ruibang Luo, and Tak-Wah Lam. Mica: A fast short-read aligner that takes full advantage of intel many integrated core architecture (mic). *arXiv preprint arXiv:1402.4876*, 2014.

[3] George Chrysos and Senior Principal Engineer. Intel xeon phi coprocessor (codename knights corner). In *Proceedings of the 24th Hot Chips Symposium*, 2012.

[4] Slo-Li Chu and Chih-Chieh Hsiao. Methods for optimizing opencl applications on heterogeneous multicore architectures. *Appl. Math*, 7(6):2549–2562, 2013.

[5] Microsoft Corporation. C++ amp: Language and programming model, 2012.

[6] Peter N. Glaskowsky. Nvidia's fermi: The first complete gpu computing architecture, 2009.

[7] Khronos OpenCL Working Group et al. The opencl specification. *version*, 1(29):1+, 2008.

[8] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In

ACM SIGARCH Computer Architecture News, volume 37, pages 152–163. ACM, 2009.

[9] Intel. Writing optimal opencl code with intel opencl sdk, 2011.

[10] Intel. Opencl optimization guide for intel xeon phi coprocessor and intel xeon processor, 2015.

[11] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 341–352. ACM, 2012.

[12] Volodymyr V Kindratenko, Jeremy J Enos, Guochun Shi, Michael T Showerman, Galen W Arnold, John E Stone, James C Phillips, and Wen-mei Hwu. Gpu clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.

[13] Dušan Kodek. *Arhitektura in organizacija računalniških sistemov*. Bitim, 2008.

[14] Joo Hwan Lee, Kaushik Patel, Nimit Nigania, Hyojong Kim, and Hyesoon Kim. Opencl performance evaluation on modern multi core cpus. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1177–1185. IEEE, 2013.

[15] Simon McIntosh-Smith, James Price, Richard B Sessions, and Amaurys A Ibarra. High performance in silico virtual drug screening on many-core processors. *International Journal of High Performance Computing Applications*, page 1094342014528252, 2014.

[16] Nvidia. Tesla k20 gpu accelerator, 2013.

[17] Nvidia. Cuda c programming guide, 2015.

[18] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[19] James C Phillips, John E Stone, and Klaus Schulten. Adapting a message-driven parallel application to gpu-accelerated clusters. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–9. IEEE, 2008.

[20] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *Parallel Processing and Applied Mathematics*, pages 559–570. Springer, 2014.

[21] Davor Sluga, Tomaz Curk, Blaz Zupan, and Uros Lotric. Heterogeneous computing architecture for fast detection of snp-snp interactions. *BMC bioinformatics*, 15(1):216, 2014.

[22] John A Stratton, Hee-Seok Kim, Thoman B Jablin, and Wen-Mei W Hwu. Performance portability in accelerated parallel kernels. *Center for Reliable and High-Performance Computing*, 2013.

[23] Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. *Person Education*, 2012.

[24] TOP500.org. Top500 list, 2015.

[25] Shigeyoshi Tsutsui and Noriyuki Fujimoto. Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2523–2530. ACM, 2009.

[26] Wei Xue, Chao Yang, Haohuan Fu, Xinliang Wang, Yangtong Xu, Lin Gan, Yutong Lu, and Xiaoqian Zhu. Enabling and scaling a global shallow-water atmospheric model on tianhe-2. In *Proceedings of the*

2014 IEEE 28th International Parallel and Distributed Processing Symposium, pages 745–754. IEEE Computer Society, 2014.