Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matevž Markovič

Izboljšana hitra Gaussova transformacija v OpenCL

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in matematika

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo: Izboljšana hitra Gaussova transformacija v OpenCL.

Tematika naloge:

Diskretna različica Gaussove transformacije se precej uporablja na področju numerične analize podatkov. Velik problem pri njeni uporabi predstavlja sam izračun, ki pri večjih količinah podatkov postane zelo zamuden. Poiščite ustrezno aproksimacijo diskretne Gaussove transformacije in jo z uporabo ogrodja OpenCL prilagodite za izvajanje na grafičnih karticah. Naredite primerjalno analizo hitrosti računanja in natančnosti izračunov za vašo in obstoječe rešitve.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matevž Markovič, z vpisno številko 63100280, sem avtor diplomskega dela z naslovom:

Izboljšana hitra Gaussova transformacija v OpenCL (angl. *Improved Fast Gauss Transform in OpenCL*)

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 25. januarja 2016 Podpis avtorja:

Zahvaljujem se svojemu mentorju izr. prof. dr. Urošu Lotriču ter asistentu Davorju Slugi za pomoč pri izdelavi diplomskega dela.

Svojim staršem ter svoji sestri.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DGT	Discrete Gauss Transform	diskretna Gaussova transformacija
FGT	Fast Gauss Transform	hitra Gaussova transformacija
IFGT	Improved Fast Gauss Transform	izboljšana hitra Gaussova transformacija
DFGT	Dual-tree Fast Gauss Transform	dvodrevesna hitra Gaussova transformacija
DIFGT	Dual-tree Improved Fast Gauss Transform	dvodrevesna izboljšana hitra Gaussova transformacija
SIFGT	Single-tree Improved Fast Gauss Transform	enodrevesna izboljšana hitra Gaussova transformacija
FMM	fast multipole method	hitra večpolna metoda
ITL	information theoretic learning	informacijsko teoretično učenje
IP	information potential	informacijski potencial

Povzetek

Naslov: Izboljšana hitra Gaussova transformacija v OpenCL

Diskretna Gaussova transformacija (ang. Discrete Gauss Transform, DGT) se pogosto pojavlja na področjih strojnega učenja, informatike, fizike ter ekonomije. Zaradi njene časovne neučinkovitosti se pri problemih večjih razsežnosti pogosto posega po aproksimativnih metodah, ki nam omogočajo hiter izračun njenega približka. Ena od teh metod je metoda izboljšane hitre Gaussove transformacije (ang. Improved Fast Gauss Transform, IFGT). V pričujočem diplomskem delu je predstavljena implementacija IFGT na platformi OpenCL, ki omogoča uporabo procesnih zmogljivosti grafične kartice za pohitritev celotnega izračuna. Implementacijo smo testirali z različnimi testnimi podatki tako na procesorju kot na grafični kartici. Primerjali smo jo z implementacijo DGT, ki je bila prav tako izvedena na procesorju in na grafični kartici z uporabo OpenCL. Implementacijo IFGT smo preverili s pomočjo izračuna Rényijeve entropije.

Ključne besede: OpenCL, grafična kartica, diskretna Gaussova transformacija, izboljšana hitra Gaussova transformacija, implementacija, paralelnost, Rényijeva entropija .

Abstract

Title: Improved Fast Gauss Transform in OpenCL

Discrete Gauss Transfrom (DGT) commonly appears in areas such as artificial learning, informatics, physics and economy. Due to its inefficiency in terms of speed, especially when we start considering larger problems, faster approximative methods such as Improved Fast Gauss Transform (IFGT) are frequently used instead. This thesis discusses the implementation of IFGT on the OpenCL platform, which enables us to use processing capabilities of the GPU to accelerate the computation. The implementation is tested using different sets of test data both with and without the graphics card. We compare it with the implementation of DGT, which is also implemented on the CPU and on the GPU using OpenCL. We test our implementation in the computation of continuous Rényi's entropy.

Keywords: OpenCL, graphics card, Discrete Gauss Transform, Improved Fast Gauss Transform, implementation, parallelism, Rényi entropy.

Poglavje 1

Uvod

Diskretna Gaussova transformacija (ang. Discrete Gauss Transform, DGT) je diskretna različica *gaussove transformacije*. Uporablja se na primer na področju statistike za konstrukcijo verjetnostnih porazdelitev na osnovi podatkov [20, str. 22], na področju informatike za računanje entropije [6] ter na področju statističnega strojnega učenja za izdelavo funkcij, ki se bodo najbolj prilegale dosedanjim odzivom sistema in jih lahko uporabimo za napovedovanje novih odzivov sistema [20, str. 28]. Izračun DGT je pri večjih količinah podatkov v praksi izredno zamuden, zaradi česar si ga želimo pohitriti, četudi na račun natančnosti samega izračuna.

Naš namen je implementacija in paralelizacija izboljšane hitre Gaussove transformacije (ang. Improved Fast Gauss Transform, IFGT) na platformi OpenCL, ki omogoča hitro računanje približka DGT s pomočjo grafične kartice.

V tem delu bomo najprej predstavili IFGT kot način izračuna DGT, ki nam omogoča hiter izračun približka slednje do poljubne natančnosti. Na kratko bomo spoznali tudi hitro Gaussovo transformacijo (ang. Fast Gauss Transform, FGT), dvodrevesno hitro Gaussovo transformacijo (ang. Dual-Tree Fast Gauss Transform , DFGT) ter dvodrevesno izboljšano hitro Gaussovo transformacijo (ang. Dual-Tree Fast Gauss Transform, DIFGT). Nato se bomo posvetili standardu OpenCL, poizkušali bomo ugotoviti v čem so

posebnosti programiranja grafičnih kartic ter kako nam te posebnosti lahko pomagajo pri pohitritvi naše implementacije IFGT. Sledil bo opis same implementacije IFGT, vseh podpornih orodij, ki so bila razvita v namen njenega preizkušanja ter njeno vrednotenje s pomočjo različnih testov. Poizkušali bomo ugotoviti, kako se odreže v primerjavi z implementacijo IFGT, ki teče le na centralni procesni enoti, ter z implementacijo DGT, ki teče na centralni procesni enoti in na grafični kartici. Kot primer uporabe implementacije IFGT bomo slednjo izkoristili za izračun približka informacijskega potenciala, ki ga bomo uporabili za izračun Rényijeve entropije danih podatkov. V sklepnem delu bomo povzeli rezultate vrednotenja naše implementacije, podali mnenje glede njene uporabnosti in predlagali možne prihodne izboljšave.

Poglavje 2

Predstavitev IFGT

V tem poglavju se bomo posvetili teoretični predstavitvi IFGT. Najprej si bomo pogledali DGT, predstavili bomo potrebna orodja in koncepte, ki nam bodo pomagala pri razumevanju delovanja IFGT, nato se bomo posvetili predstavitvi IFGT in jo na koncu primerjali tudi z nekaterimi drugimi aproksimativnimi metodami izračuna DGT.

2.1 Predstavitev DGT

Diskretna Gaussova transformacija je diskretna različica *Gaussove transformacije*. Naj je $f(x)$ poljubna zvezna funkcija, Γ neka podmnožica $\Re^d$ in x,$y \in \Gamma$. Tedaj zapišemo Gaussovo transformacijo z enačbo

$$G_\delta f(x) = \int_\Gamma e^{-|x-y|^2/\delta} f(y)\, dy \quad (\delta > 0)\,, \tag{2.1}$$

kjer je h pasovna širina, ki določa moč vpliva točk y iz okolice x na vrednost v točki x v odvisnosti od njihove oddaljenosti, ter $\delta = h^2$. Pojavlja se v veliko problemih praktične matematike [5, str. 1], kot so Cauchyjev problem[1] ter pri glajenju funkcije $\psi(x)$[2]. Drugi primeri uporabe Gaussove transformacije

[1]Cauchyjev problem je problem iskanja rešitev parcialne diferencialne enačbe, ki zadošča nekaterim omejitvam.

[2]Poljubno zvezno funkcijo $\psi(x)$ se da aproksimirati z družino gladkih in hitro padajočih

so v neparametrični statistiki [17, str. 13] ter v tomografiji [4, str. 1].

Pri računanju Gaussove transformacije z računalnikom moramo le-to diskretizirati. Pričnemo z množico izvornih točk $s_i \in \Re^d$, ki jim priredimo uteži q_i. Le-te so enake vrednostim $f(s_i)$. Izračunu diskretne Gaussove transformacije v ponorni točki t_j lahko tedaj pravimo Gaussovo polje (*Gaussian field*) [4, str. 2], sestavljeno iz izvornih točk s_i, od katerih ima vsaka svojo moč q_i. Moč polja je izračunana v ponornih točkah t_j glede na pasovno širino h. Pri manjši pasovni širini imajo velik vpliv na vrednost ponorne točke le njej bližnje izvorne točke, medtem ko imajo pri veliki pasovni širini velik vpliv na vrednost ponorne točke tudi izvorne točke, ki so od nje bolj oddaljene.

Pred začetkom raziskovanja metod računanja diskretne Gaussove transformacije je potrebno vzpostaviti notacijo, ki se je bomo držali skozi celotno diplomsko delo. Notacijo povzamemo po [20]. Naj so

- $t_j \in \Re^d$ za $j = 1, \ldots, M$ **ponorne točke**, kjer M je število ponornih točk,
- $s_i \in \Re^d$ za $i = 1, \ldots, N$ **izvorne točke**, kjer N je število izvornih točk,
- $q_i \in \Re^+$ za $i = 1, \ldots, N$ **uteži** izvornih točk ter
- $h \in \Re^+$ je pasovna širina.

Tedaj lahko **diskretno Gaussovo transformacijo** oziroma DGT zapišemo v obliki

$$\left(G(t_j) = \sum_{i=1}^{N} q_i \cdot e^{-||t_j - s_i||^2/h^2} \right)_{j=1,\ldots,M}, \tag{2.2}$$

po kateri izračunamo vrednosti transformacije v vseh ponornih točkah t_j. Opazimo lahko, da je časovna kompleksnost izračuna DGT $O(N \cdot M)$, kar je izredno slabo za probleme z velikim številom bodisi ponornih bodisi izvornih točk. V naslednjih poglavjih si bomo pogledali aproksimativne algoritme za

funkcij oblike

$$\psi_\delta(x) = (\pi\delta)^{-d/2} G_\delta \psi(x)\,,$$

ki konvergirajo proti $\psi(x)$ ko $\delta \to 0$.

izračun DGT, katerih časovna kompleksnost se giblje v okolici $O(M + N)$.

Ponorne ter izvorne točke pred računanjem uvrstimo v prostor $[0,1]^d$, pri čemer moramo seveda ustrezno spremeniti tudi pasovno širino h.

2.2 Potrebna orodja

Za zmanjšanje časovne kompleksnosti izračuna DGT potrebujemo orodja, s katerimi problem lahko razbijemo na več manjših problemov, ki nam omogočajo hiter izračun približka rezultata znotraj podane zahtevane natančnosti ter nam omogočajo reševanje problema v poljubno dimenzionalnem prostoru.

2.2.1 Tehnike FMM

Zmanjšanje časovne kompleksnosti iz $O(N \cdot M)$ na $O(N + M)$ nam omogoča uporaba metod, ki gradijo na tehnikah FMM (ang. fast multipole method) [20, str. 41]. To je zbirka aproksimacijskih tehnik za pohitritev večkratne evaluacije funkcije $f : \Re^d \to \Re$ oblike

$$f(t) = \sum_{i=1}^{N} q_i \cdot k(t,s_i) \tag{2.3}$$

v ponornih točkah $t = t_j$ za $j = 1, \ldots, M$. Funkcija $k(t,s_i) : \Re^d \times \Re^d \to \Re$ je funkcija jedra (ang. kernel function)[3], v našem primeru je to seveda Gaussovo jedro (pomen spremenljivk je isti kot pri DGT)

$$k(t,s_i) = k_h(t,s_i) = e^{-||t-s_i||^2/h^2}\,, \tag{2.4}$$

ki je popularno zaradi njegovih pozitivnih analitičnih lastnosti kot sta pozitivna definitnost in gladkost ter zaradi njegove vloge v verjetnosti[4].

[3]Funkcija jedra oziroma jedro (ang. kernel) je v splošnem funkcija oblike $k : \chi \times \chi \to \Re$ kjer je χ množica analiziranih objektov. Navadno funkcije jeder uporabljamo za merjenje podobnosti med objekti v χ.

[4]Centralni limitni izrek pravi, da je vsota vrednosti neodvisnih slučajnih spremenljivk približno normalno porazdeljena.

Aproksimacijske tehnike FMM lahko strnemo v naslednjih točkah.

- Uporaba aproksimativnega izračuna jedrne funkcije $k(t,s_i)$, pri katerem le-to predstavimo z neskončno vrsto, kot je na primer Taylorjeva vrsta. Pri samem računanju lahko seveda seštejemo le končno mnogo členov te vrste, pri čemer želimo, da je napaka izračuna pod mejo za napako (ang. error bound).

- Hierarhično deljenje (*particioniranje*) prostora, pri katerem izvorne ter ponorne točke razdelimo v več skupin na več nivojih na podlagi porazdelitve točk, želene meje za napako, ter lastnosti jedra $k(t,s_i)$.

- Uporaba translacijskih operatorjev, ki omogočajo prenos rezultatov izračuna med nivoji hierarhično deljenega prostora.

.

Poznamo dve vrsti tehnik FMM:

1. večnivojske (ang. multi-level) tehnike FMM ter

2. enonivojske (ang. single-level) tehnike FMM, kjer je prostor deljen le na enem nivoju, ter zaradi tega ne potrebujemo translacijskih operatorjev.

Uporaba tehnik FMM pri pohitritvi izračuna DGT ni enostavna, saj zahteva natančno poznavanje danega problema (porazdelitev izvornih ter ponornih točk, pasovna širina, meja napake) in preudarno uporabo teh tehnik v skladu z značilnostmi tega problema. Na primer, pri enakomerni porazdelitvi točk je smiselno uporabiti deljenje prostora na približno enako velike kvadrate. Pri točkah, ki so nagnetene na nekaj kupih v prostoru, pa je bolj smiselno uporabiti deljenje prostora, ki skupaj nagnetene točke uvrsti v isto skupino.

2.2.2 Večdimenzionalni indeks

Pri uporabi metod za izračun približka DGT bomo pogosto morali prečesavati zaporedja večdimenzionalnih vektorjev, ki bodo enolično določali posame-

zne koeficiente v vrstah za aproksimativen izračun jedrne funkcije (glej poglavje 2.2.1). V ta namen potrebujemo nekakšno posplošitev koncepta indeksa koeficienta na več dimenzij.

Predpostavimo, da je $d \geq 1$ je dimenzija našega problema ter naj so $\alpha_1, \ldots, \alpha_d$ nenegativna cela števila. Tedaj je $\alpha = (\alpha_1, \ldots, \alpha_d)$ večdimenzionalni indeks in velja:

$$|\alpha| := \alpha_1 + \ldots + \alpha_d \tag{2.5a}$$

$$\alpha! := \alpha_1! \cdot \ldots \cdot \alpha_d! \tag{2.5b}$$

$$x^\alpha := x_1^{\alpha_1} \cdot \ldots \cdot x_d^{\alpha_d} \qquad (x \in \Re^d) \tag{2.5c}$$

V kolikor napišemo $\alpha > p$ za naravno število p, tedaj to pomeni da je $\forall i = 1, \ldots, d : \ \alpha_i > p$.

2.3 IFGT

Izboljšana hitra Gaussova transformacija (ang. Improved Fast Gauss Transform, IFGT) je ϵ-natančen aproksimativni algoritem za izračun diskretne Gaussove transformacije (DGT), ki omogoča izračun slednje do poljubne natančnosti ϵ [19]. Ob podani želeni napaki $\epsilon > 0$ izračuna približek $\tilde{G}(t_j)$ natančne vrednosti $G(t_j)$ ki bi jo dobili z DGT, in sicer takšen, da je maksimalna absolutna napaka glede na absolutno vsoto vseh uteži $Q = \sum_{i=1}^{N} |q_i|$ omejena navzgor z ϵ,

$$\max_{t_j} \left[\frac{\tilde{G}(t_j) - G(t_j)}{Q} \right] \leq \epsilon \,. \tag{2.6}$$

Algoritem IFGT je bil prvič predstavljen leta 2003 v [21], 12 let po prvi predstavitvi algoritma FGT v [4] leta 1991. Tako IFGT kot FGT uporabljata enonivojske tehnike FMM (glej sekcijo 2.2.1), saj dosegata pohitritev izračuna DGT preko aproksimativnega izračuna jedrne funkcije s pomočjo vrste do poljubne natančnosti, prostor v katerem ležijo izvorne ter ponorne

točke pa delita enonivojsko - zaradi česar rezultatov izračuna med nivoji prostora ni potrebno prevajati s pomočjo translacijskih operatorjev. Predpona "izboljšan" ("*improved*") v imenu IFGT je zavajajoča, saj gre v resnici za algoritem z drugačnim pristopom k aproksimaciji DGT, s katero se izogne večini slabosti algoritma FGT. Več o razlikah med IFGT ter FGT bomo spoznali v poglavju 2.4. Algoritem IFGT in njegove podrobnosti bomo povzeli po [20].

Osnovna ideja algoritma je obravnava izvornih točk v okviru gruč katerim pripadajo, medtem ko pa ponorne točke še vedno obravnavamo ločeno. Ob računanju DGT neke ponorne točke bomo lahko upoštevali le doprinose gruč in ne doprinose posameznih izvornih točk, saj bomo doprinose izvornih točk že prej zbrali v okviru Taylorjevega razvoja v vrsto okoli centra gruče. Smiselnost tega pristopa bomo obravnavali v poglavju 2.3.2.

2.3.1 Matematično ozadje algoritma IFGT

Algoritem IFGT uporablja razvoj v Taylorjevo vrsto. Pričnimo s preoblikovanjem DGT-ja tako da uporabimo poljubno točko $c_k \in \Re^d$.

$$\begin{aligned} G(t_j) &= \sum_{i=1}^{N} q_i \cdot e^{-||t_j - s_i||^2/h^2} \\ &= \sum_{i=1}^{N} q_i \cdot e^{-||(t_j - c_k) - (s_i - c_k)||^2/h^2} \\ &= \sum_{i=1}^{N} q_i \cdot e^{-||t_j - c_k||^2/h^2} \cdot e^{-||s_i - c_k||^2/h^2} \cdot e^{2(t_j - c_k)\cdot(s_i - c_k)/h^2}. \end{aligned} \tag{2.7}$$

Izračun prvega faktorja za vseh M ponornih točk t_j ima časovno kompleksnost $O(M \cdot d)$, izračun drugega faktorja za vseh N izvornih točk s_i ima časovno kompleksnost $O(N \cdot d)$, medtem ko ima izračun tretjega faktorja časovno zahtevnost $O(M \cdot N)$. Prav tretji faktor je tisti, ki ga moramo razviti s pomočjo Taylorjeve vrste in najti način za njegovo pohitritev, saj ima njegov izračun ogromno preveliko časovno zahtevnost.

Razvijmo funkcijo $e^{2(t_j - c_k)\cdot(s_i - c_k)/h^2}$ v Taylorjevo vrsto okoli točke c_k (s

pomočjo [20, str. 64]). Razvoj opravimo po vseh členih, za katere velja, da je $|\alpha| < p$, kjer α je večdimenzionalni indeks kot omenjen v poglavju 2.2.2. Tedaj je

$$e^{2(t_j-c_k)\cdot(s_i-c_k)/h^2} \approx \sum_{|\alpha|<p} \frac{2^{|\alpha|}}{\alpha!} \left(\frac{t_j - z}{h}\right)^{\alpha} \left(\frac{s_i - z}{h}\right)^{\alpha}. \tag{2.8}$$

Ob uporabi enačb 2.7 ter 2.8 lahko napišemo približek za DGT, ki ga lahko uporabimo za osnovo algoritma IFGT:

$$\begin{aligned}\tilde{G}(t_j) &= \sum_{s_i} q_i \cdot e^{-||t_j-z||^2/h^2} \cdot e^{-||s_i-z||^2/h^2} \cdot \sum_{\alpha\geq 0} \frac{2^{|\alpha|}}{\alpha!} \left(\frac{t_j - z}{h}\right)^{\alpha} \left(\frac{s_i - z}{h}\right)^{\alpha} \\ &= \sum_{\alpha\geq 0} \frac{2^{|\alpha|}}{\alpha!} \sum_{s_i} q_i \cdot e^{-||s_i-z||^2/h^2} \left(\frac{s_i - z}{h}\right)^{\alpha} \cdot e^{-||t_j-z||^2/h^2} \left(\frac{t_j - z}{h}\right)^{\alpha}.\end{aligned} \tag{2.9}$$

2.3.2 Delitev izvornih točk v gruče

Naslednji dve ugotovitvi nam bosta pokazali smiselnost grupiranja izvornih točk v gruče in ignoriranja interakcij med ponornimi točkami in centri gruč, ki so oddaljene bolj od neke določene razdalje.

- Moč Gaussovega jedra postaja z razdaljo vse šibkejša [15, str. 11], kar nam omogoča, da lahko ignoriramo interakcije med ponornimi in izvornimi točkami, ki so si med seboj oddaljene za dovolj veliko razdaljo.

- Taylorjev razvoj za posamezno točko s_i bi bilo dobro računati okoli točke c_k, ki je blizu s_i, saj je Taylorjeva vrsta z malo členi veljavna le v majhni okolici točke okoli katere je razvita [15, str. 11]. Za računanje Taylorjevega razvoja s_i okoli preveč oddaljene točke c_k bi morali računati in seštevati ogromno členov vrste.

Izvorne točke lahko sedaj razdelimo v gruče $Z_{k=1,\ldots,K}$. Vsaka od njih ima svoj center c_k, radij r_k, ki je enak največji oddaljenosti izvorne točke v tej gruči od centra gruče ter interakcijski radij r_k^{int}, ki je enak maksimalni razdalji med

centrom gruče c_k in ponorno točko t_j, v kateri se še upošteva doprinos te gruče.

Deljenje točk v gruče izvedemo s pomočjo algoritma za grupiranje na podlagi najbolj oddaljene točke (ang. Farthest-point clustering), ki je bil predstavljen leta 1985 [3]. Gre za aproksimativen algoritem za reševanje problema K centrov (ang. K-center problem)[5]. ki zagotavlja rešitev (konfiguracijo gruč) z maksimalnim radijem gruče največ dvakrat večjim od optimalne rešitve.

Psevdokoda za grupiranje na podlagi najbolj oddaljene točke je predstavljena v algoritmu 2.1.

Algoritem 2.1 Algoritem za grupiranje na podlagi najbolj oddaljene točke

Vhod: Izvorne točke $s_{i=1,\ldots,N}$, naravno število $K \leq N$.

Izhod: K gruč Z_k ki predstavljajo particijo izvornih točk s centri c_k ter radiji r_k, kjer je maksimalni radij $\max_k r_k$ največ dvakrat večji od optimalnega.

1: Izberi naključno točko s_i za center c_1 prve gruče Z_1. Vse ostale točke naj tudi pripadajo tej gruči.
2: Za vsako točko v Z_1 izračunaj razdaljo do c_1 in si zapomni maksimalno razdaljo do centra v tej gruči kot radij te gruče r_1
3: **Za** $k = 1 \ldots K$ **naredi**
4: Ustvari gručo Z_{k+1}
5: Pojdi skozi vse do sedaj ustvarjene gruče in si zapomni gručo z maksimalnim radijem. Loči točko, ki povzroča ta radij, od te gruče in jo postavi za center gruče Z_{k+1}.
6: Pojdi skozi vse točke s_i in primerjaj njihove razdalje do centra gruče kateri trenutno pripadajo in centra nove gruče Z_{k+1}. V kolikor je sledeča manjša, naj se točka s_i premakne v gručo Z_{k+1}.
7: **Konec Za**

[5]Problem K centrov (*K-center problem*) je problem deljenja N točk $s_i \in \Re^d$ v K različnih gruč Z_k s centri c_k, kjer je maksimalni radij gruče $\max_k \max_{s_i \in Z_k} ||s_i - c_k||$ minimalen. Problem je NP-poln.

2.3.3 Izpeljava algoritma

S konceptom gruč lahko sedaj izpopolnimo enačbo 2.9 v obliko, ki jo bomo lahko uporabili pri dejanskem izračunu približka DGT. Naj je Z_k neka gruča s centrom c_k in t_j ponorna točka, v kateri želimo izračunati doprinos gruče ter p_k neko naravno število, ki je lastno vsaki gruči ter kateremu pravimo **število odreza vrste** (ang. truncation number). Uvedimo **IFGT-Taylorjev koeficient** za gručo Z_k:

$$C_\alpha(Z_k) = \frac{2^{|\alpha|}}{\alpha!} \sum_{s_i \in Z_k} q_i \cdot e^{-||s_i - c_k||^2/h^2} \left(\frac{s_i - c_k}{h} \right)^\alpha . \tag{2.10}$$

Tedaj je **IFGT-Taylorjev razvoj (*l1*-različica)**

$$\tilde{G}_{Z_k}(t_j) = \sum_{|\alpha| < p_k} C_\alpha(Z_k) \cdot e^{-||t_j - c_k||^2/h^2} \left(\frac{t_j - c_k}{h} \right)^\alpha . \tag{2.11}$$

Zaradi uporabe l1-norme[6] v $|\alpha| < p_k$ je IFGT-Taylorjevih koeficientov $C_\alpha(Z_k)$ za gručo Z_k ravno $\binom{p_k+d}{d}$ [20, str. 66]. Med ponorno točko t_j ter gručo Z_k se izračuna vpliv le v kolikor je $||t_k - c_k|| \leq r_k^{\text{int}}$, kjer je r_k^{int} interakcijski radij (ang. interraction radius) gruče Z_k.

Število odreza vrste p_k izračunamo za vsako gručo Z_k tako, da poiščemo tak najmanjši $p \geq 1$, da je

$$p_k = \min_{p \geq 1} \frac{1}{p!} \left(\frac{2 \cdot r_k^{\text{int}} \cdot r_k}{h^2} \right)^p \leq \epsilon \,, \tag{2.12}$$

medtem ko lahko interakcijski radij r_k^{int} izračunamo s pomočjo enačbe

$$r_n^{\text{int}} = \min\{\max_{i,j} ||t_j - s_i||, r_k + h \cdot \sqrt{\ln(\epsilon^{-1})}\} \,. \tag{2.13}$$

Prišli smo do točke, ko lahko zapišemo algoritem IFGT.

$$\tilde{G}_{\text{IFGT}}(t_j) = \sum_{||t_j - c_k|| \leq r_k^{\text{int}}} \sum_{|\alpha| < p_k} C_\alpha(Z_k) \cdot e^{-||t_j - c_k||^2/h^2} \left(\frac{t_j - c_k}{h} \right)^\alpha . \tag{2.14}$$

[6] l1-norma vektorja $(x_1, \ldots, x_d)$ je $|(x_1, \ldots, x_d)|_1 = \sum_{i=1}^d |x_i|$.

Psevdokoda IFGT je predstavljena v algoritmu 2.2. Časovna kompleksnost algoritma IFGT je

$$O((N+M)\cdot\binom{p_{\max}-1+d}{d})+O(N\cdot K\cdot d)\,, \tag{2.15}$$

kjer je N število izvornih točk, M število ponornih točk, $p_{\max}$ maksimalno število odreza vrste po vseh gručah, K število gruč ter d število dimenzij [20, str. 76].

Algoritem 2.2 IFGT

Vhod: $s_{i=1,\ldots,N} \in [0,1]^d$, $t_{j=1,\ldots,M} \in [0,1]^d$, $q_{i=1,\ldots,N} \in \Re^+$, $h > 0$, $\epsilon > 0$.

Izhod: $\forall j = 1,\ldots,M : \tilde{G}_{\mathrm{IFGT}}(t_j)$.

1: Izračunaj parametre: število gruč K ter maksimalno število odreza vrste $p_{\max}$.

2: S pomočjo *algoritma za grupiranje na podlagi najbolj oddaljene točke* (algoritem 2.1) razdeli izvorne točke v K gruč. Vsaka gruča naj ima center $c_k \in [0,1]^d$, radij $r_k > 0$, število odreza vrste $p_k \leq p_{\max}$ ter interakcijski radij r_k^{int}.

3: Za vsako gručo Z_k izračunaj koeficiente C_α^k v skladu z enačbo 2.10 in njenim številom odreza vrste p_k.

4: Za vsako ponorno točko t_j najdi gruče Z_k kjer $||t_j - c_k|| \leq r_k^{\mathrm{int}}$ in izračunaj doprinos gruče k $\tilde{G}_{\mathrm{IFGT}}(t_j)$ v skladu z enačbo 2.11.

2.4 Primerjava IFGT z ostalimi metodami aproksimativnega izračuna DGT

Poleg IFGT poznamo še druge metode aproksimativnega izračuna DGT. Nekatere izmed teh so hitrejše in bolj prilagodljive od IFGT, vendar pa so zaradi njihovih načinov delovanja neprimerne za grafične kartice v kombinaciji z OpenCL. Primeri teh metod so:

- hitra Gaussova transformacija (ang. Fast Gauss Transform, FGT) [4],
- dvodrevesna hitra Gaussova transformacija (ang. Dual-tree Fast Gauss Transform, DFGT) [11],
- dvodrevesna izboljšana hitra Gaussova transformacija (ang. Dual-tree Improved Fast Gauss Transform, DIFGT) [20],
- enodrevesna izboljšana hitra Gaussova transformacija (ang. Single-tree Improved Fast Gauss Transform, SIFGT) [20].

Za vse metode aproksimativnega izračuna DGT je značilno, da uporabljajo tehnike FMM (glej poglavje 2.2.1), ki so bodisi enonivojske (single-level) ali večnivojske (multi-level). Enonivojske tehnike FMM uporabljata FGT ter IFGT, večnivojske pa DFGT, DIFGT in SIFGT. Slednje nadgradijo FGT ter IFGT metodi z uporabo (najpogosteje binarnih) dreves, ki na vsakem nivoju predstavljajo delitev vseh (izvornih ter ponornih) točk, ter katere se rekurzivno obdeluje. Pri dimenzijah $d > 3$ se metode, ki uporabljajo drevesne strukture, izkažejo kot veliko hitrejše od metod IFGT ter FGT, še posebej pri velikih pasovnih širinah h (tipično $h \geq 10$). Najbolje se odreže algoritem DIFGT [20, str. 107]. Prav uporaba drevesnih struktur pa onemogoča učinkovito implementacijo metod DFGT, DIFGT ter SIFGT na grafični kartici, saj standard OpenCL ne dovoljuje uporabe rekurzije (več v poglavju 3.2). Tudi metoda FGT zahteva uporabo rekurzije, saj se vrednost Hermitske funkcije iz enačbe 2.17 izračuna ravno preko rekurzivno definiranega Hermitskega

polinoma. Metoda IFGT je tako pravzaprav edina, ki smo jo lahko enostavno implementirali za izvajanje na grafični kartici v sklopu standarda OpenCL.

Razlika med IFGT in FGT ter metodami, ki uporabljajo drevesa, je tudi v meji za napako, ki jo jamčijo.

- FGT ter IFGT jamčita: $|\tilde{G}(t_j) - G(t_j)| < \epsilon \cdot \sum_{i=1}^{N} |q_i|$.
- DFGT, DIFGT ter SIFGT jamčijo: $|\tilde{G}(t_j) - G(t_j)| < \epsilon \cdot |G(t_j)|$.

Opazimo lahko, da metodi IFGT ter FGT jamčita le absolutno napako, odvisno od uteži izvornih točk, ne glede na končni rezultat same transformacije $G(t_j)$. Metode, ki uporabljajo drevesne strukture, pa jamčijo relativno napako, odvisno od rezultata same transformacije - parameter ϵ ima v tem primeru veliko oprijemljivejšo in intuitivnejšo vlogo v primerjavi s parametrom ϵ pri FGT ter IFGT, saj nam omogoča enostavno izbiro natančnosti, do katere želimo izračunati rezultat $G(t_j)$.

Mi se bomo posvetili primerjavi IFGT s starejšim algoritmom FGT.

2.4.1 Primerjava IFGT ter FGT

Tako kot IFGT je tudi FGT ϵ-natančen aproksimativen algoritem za izračun DGT. Prostor $[0,1]^d$ razdeli enakomerno na d-dimenzionalne škatle s stranico dolžine $l = \sqrt{2}rh$ (kjer je konstanta $r \leq \frac{1}{2}$ in h pasovna širina) in vanje uvrsti ponorne ter izvorne točke. Pri računanju vrednosti v ponorni točki y upoštevamo izvorne točke v $(2n+1)^d$ sosednjih škatlah za neko naravno število n. Vse ostale izvorne točke v prostoru k vrednosti y namreč prispevajo manj kot $Q\epsilon$ kjer $Q = \sum_{i=0}^{N} |q_i|$ in ϵ napaka, medtem ko lahko vedno izberemo tako naravno število n na podlagi ϵ ter r, da je napaka v vrednosti ponorne točke manjša od $Qe^{-2r^2n^2}$ [4].

FGT metode FMM iz poglavja 2.2.1 aplicira neposredno na naslednji dve razvitji Gaussovega jedra v vrsto:

$$e^{-||y-x_i||^2/h^2} = \sum_{n=0}^{p-1} \frac{1}{n!} \left(\frac{x_i - x_{\text{center}}}{h} \right)^n h_n \left(\frac{y - x_{\text{center}}}{h} \right) + \epsilon(p)\,, \qquad (2.16a)$$

$$e^{-||y-x_i||^2/h^2} = \sum_{n=0}^{p-1} \frac{1}{n!} h_n \left(\frac{x_i - y_{\text{center}}}{h} \right) \left(\frac{y - y_{\text{center}}}{h} \right)^n + \epsilon(p)\,, \qquad (2.16b)$$

kjer je $h_n(x)$ Hermitska funkcija

$$h_n(x) = (-1)^n \frac{d^n}{dx^n} \left(e^{-x^2} \right)\,, \qquad (2.17)$$

ter ju uporabi za združevanje vpliva izvornih točk v izvornih škatlah preko Hermitske vrste okoli centra izvorne škatle x_{center} ter preko Taylorjeve vrste v okolici centra ponorne škatle y_{center}. Pri računanju vpliva izvorne škatle na ponorno točko imamo štiri možnosti.

1. V kolikor je tako v izvorni kot v ponorni škatli malo točk, se doprinos izvornih točk k vrednosti ponorne točke izračuna neposredno preko DGT (formula 2.2).

2. V kolikor je v izvorni škatli veliko in v ponorni škatli malo točk, se izračuna Hermitsko vrsto okoli centra izvorne škatle ter nato doprinos centra izvorne škatle k vrednosti ponorne škatle preko formule 2.2.

3. če je v ponorni škatli veliko točk in v izvorni škatli malo točk, se tedaj izračuna Taylorjeva vrsta okoli centra ponorne škatle, medtem ko se doprinosi ponornih točk k tej Taylorjevi vrsti upoštevajo za vsako ponorno točko posebej. Na koncu se izračuna vpliv centra ponorne škatle na posamezne ponorne točke znotraj te škatle,

4. V kolikor pa je tako v ponorni kot v izvorni škatli veliko točk, se uporabi Taylor-Hermitski razvoj tako okoli centrov ponorne in izvorne škatle. Ta možnost je najkompleksnejša.

V primerjavi f FGT imamo pri IFGT le en razvoj, in sicer v Taylorjevo vrsto. To odstrani potrebo po uporabi translacijskih operatorjev med vrstami (pri FGT sta to Taylorjeva ter Hermitska vrsta) ter s tem zelo poenostavimo algoritem.

Pri izračunu vrednosti obeh vrst izračunamo vsoto p^d členov, kar je pri večjih dimenzijah d ogromno več kot pri IFGT-jevemu število $\binom{p+d}{d}$ seštetih

koeficientov pri izračunu vrednosti vrste v ponorni točki za število odreza vrste p. IFGT je prav tako precej fleksibilnejši algoritem od FGT-ja zaradi lokalnega izračuna radija r_k, interakcijskega radija r_k^{int} in števila odreza vrste p_k za vsako gručo posebej. To mu omogoča določeno mero prilagajanja lastnostim porazdelitve izvornih in ponornih točk, medtem ko je število odreza vrste p ter soseščina škatle določena pri algoritmu FGT globalno.

Nefleksibilnost algoritma FGT izhaja tudi iz njegovega načina enakomerne porazdelitve prostora $[0,1]^d$ na škatle, kar še posebej pri $d > 3$ nanese potencialno ogromno število škatel (reda 10^{10} ali več), ki so povrhu še skoraj povsem prazne. V splošnem velja, da je FGT neuporaben ko je (število škatel) $> C \times \max(M,N)$ za neko majhno naravno število C.

Tako IFGT kot FGT se odrežeta dobro pri srednjih in visokih pasovnih širinah ($h > 1$) [20, str. 107], medtem ko je pri višjih dimenzijah od 3 FGT neuporaben, IFGT pa postane zelo odvisen od pasovne širine.

Največje šibkosti IFGT so:

- individualna obravnava ponornih točk ter zamudno iskanje vseh gruč v r_k^{int} soseščini teh ponornih točk,
- ne-trivialna izbira parametrov algoritma (število gruč K, število odreza vrste p_k, ...),
- ne odreže se dobro pri problemih z nizko pasovno širino (meja uporabnosti IFGT je pri $h = 0{,}03$ za $\epsilon = 10^{-2}$ ter pri $h = 0{,}5$ za $\epsilon = 10^{-6}$) [20, str.110],
- zagotovljena le absolutna meja napake.

Poglavje 3

Predstavitev OpenCL

Soustanovitelj podjetja Intel Gordon E. Moore je leta 1965 postavil napoved, da se bo število tranzistorjev na čipu v prihodnosti podvojilo vsako leto [13, 10]. Pri napovedi je bil previden in se je omejil le na obdobje naslednjih 10 let. Leta 1975 je popravil svojo napoved, in sicer da se bo število tranzistorjev na čipu podvojilo vsaki dve leti - slednjo poznamo pod imenom Moorov zakon. Proizvajalcem procesorjev je vse do sredine prvega desetletja našega tisočletja uspelo ob podvajanju števila tranzistorjev v vsaki novi generaciji procesorjev dosegati 50% pohitritev ob konstantni porabi energije [2, str. vii]. Ta trend se je tedaj ustavil in nadaljnje povečevanje zmogljivosti (oziroma frekvence delovanja procesorjev) je bilo možno le še bo povečevanju porabe energije in toplote pri delovanju. Proizvajalci procesorjev so:

- namesto povečevanja frekvence delovanja procesorja začeli v procesor vgrajevati več jeder, ki so omogočala izvajanje več nalog hkrati, niso pa pohitrila izvajanja obstoječih nalog,
- pri snovanju novih procesorskih arhitektur se začeli osredotočati na energijsko učinkovitost.

Omenjena trenda sta javnost vzpodbudila k preučevanju *heterogenih sistemov*[1]. Pisanje programske opreme za heterogene sisteme pa je zaradi njihove

[1]Heterogeni sistemi so sestavljeni iz različnih podsistemov, od katerih ima vsak svoje lastnosti, prednosti ter slabosti, ter je zaradi tega primeren le za določeno podmnožico

raznolikosti izredno težko, saj ima vsaka vrsta podsistemov takšnega heterogenega sistema svoj način delovanja, svojo paradigmo programiranja ter zaradi tega tudi drugačna orodja, prevajalnike in razhroščevalnike.

Težnje po iskanju rešitev za standardizacijo programiranja heterogenih sistemov so se v industriji pojavljale že dolgo pred predstavitvijo standarda OpenCL. Implementirane so bile rešitve kot so AMD CTM (ang. Close To Metal), CAL (ang. Compute Abstraction Layer) ter Nvidia CUDA (ang. Compute Unified Device Architecture), vendar pa je bila glavna pomanjkljivost vseh teh sistemov njihova nesplošnost, saj so bila usmerjena le v podporo strojni opremi posameznega proizvajalca. Industrija je potrebovala nek standard programiranja heterogenih sistemov, ki bi bil tako splošen, kot je na primer okolje Java, da bi omogočal izvajanje istih programov na še tako različnih procesorskih arhitekturah in sistemih različnih proizvajalcev. Odgovor je prišel v obliki standarda OpenCL.

OpenCL (ang. The Open Computing Language) je odprt in za uporabo brezplačen standard splošno-namenskega paralelnega programiranja, ki omogoča programerjem pisanje prenosljivih in učinkovitih programov za uporabo na heterogenih sistemih [8], od mobilnih naprav do superračunalnikov. Predstavljen je bil leta 2008 (različica 1.0) s strani neprofitnega konzorcija Khronos Group[2], ki so ga ustanovila podjetja IBM, Intel, AMD ter Apple. Trenutna različica standarda je 2.1 (November 2015).

Uporabnik lahko programe, napisane v skladu s standardom OpenCL, izvaja na vsakem heterogenem sistemu, za katerega obstaja izvajalno okolje OpenCL. S pomočjo OpenCL lahko programer na primer piše splošnonamenske programe, ki se nato izvajajo na grafični kartici, brez potrebe po prirejanju programa uporabi 3D programskih vmesnikov kot sta DirectX in OpenGL. Standard OpenCL definira:

nalog. Primeri teh podsistemov so grafične kartice, navadni procesorji, procesorji DSP (ang. Digital Signal Processor), namenska vezja FPGA, ...

[2] `https://www.khronos.org/`

- programski vmesnik (ang. Application Programming Interface, API),
- izvajalno okolje,
- programski jezik OpenCL C, ki izhaja iz standarda ISO C99[3].

OpenCL podpira tako podatkovni[4] kot funkcijski paralelizem[5] [8], uporablja modificirano različico IEEE 754 standarda za implementacijo aritmetike v plavajoči vejici ter omogoča souporabo OpenGL, OpenGL ES in ostalih grafičnih vmesnikov.

3.0.2 OpenCL na grafičnih karticah

Uporaba grafičnih kartic v namene vzporednega splošnega računanja je bil eden od glavnih motivacijskih dejavnikov za uvedbo splošno-namenskega računanja na grafičnih karticah oziroma t.i. GPGPU (ang. general-purpose computing on graphics processing units). Le-te sicer pogrešajo mehanizme pospeševanja hitrosti izvajanja sekvenčnih programov, ki so prisotni na sodobnih centralnih procesnih enotah[6], vendar tovrstne pomanjkljivosti upravičijo s svojo surovo hitrostjo, zmožnostjo paralelnega izvajanja velikega števila niti [18], strojno podprto sinhronizacijo med nitmi ter strojno podprto delitvijo in uravnovešanju dela med nitmi [2, str.128].

[3]C99 oziroma ISO/IEC 9899:1999 je ena izmed starejših različic standarda programskega jezika C. Njegov naslednik je standard ISO C11, ki je bil izdan leta 2011.

[4]Podatkovni paralelizem označuje način programiranja, kjer se ukazi istega programa izvajajo nad različnimi podatki. Fokus podatkovnega paralelizma je porazdelitev podatkov med procesorji.

[5]Funkcijski paralelizem oziroma paralelizem nalog označuje način programiranja, kjer se vzporedno izvajajo različne naloge. Fokus funkcijskega paralelizma je porazdelitev nalog med procesorji.

[6]Primeri mehanizmov pospeševanja hitrosti izvajanja sekvenčnih programov, ki manjkajo na grafičnih karticah, so napoved skokov (ang. branch prediction) ter izvajanje ukazov zunaj vrstnega red (ang. out-of-order execution).

3.1 Opis OpenCL

V namen predstavitve OpenCL bomo slednjega predstavili skozi naslednje štiri modele:

1. platformni model,
2. pomnilniški model,
3. izvajalni model,
4. programski model.

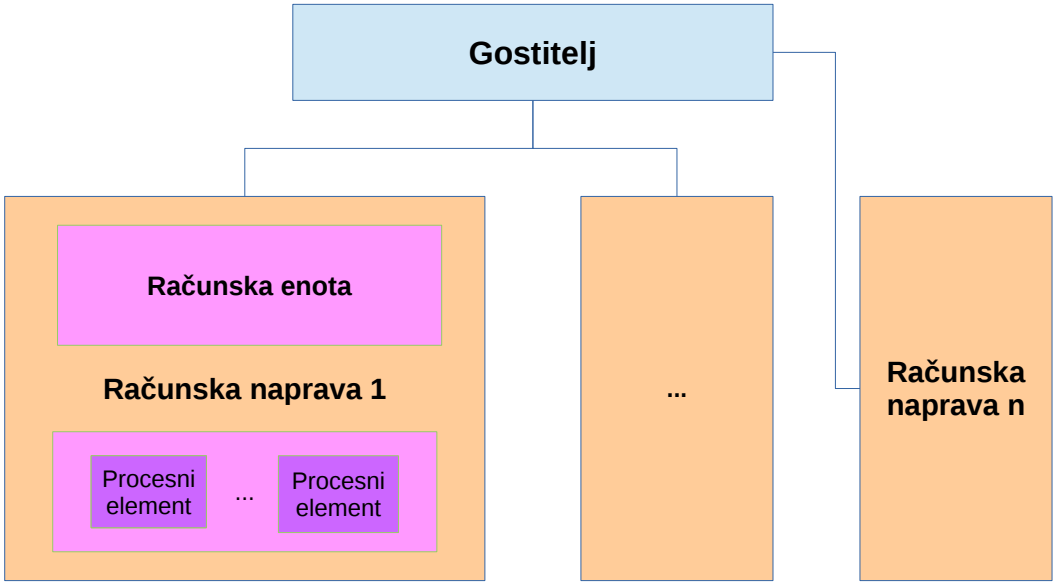
3.1.1 Platformni model

Platformni model definira abstraktni pogled na strojno opremo, ki ga uporablja programer, ko piše funkcije OpenCL C[7], ki se izvajajo na napravi. Določa, da na heterogenem sistemu, ki izvaja program OpenCL, obstaja:

- **gostitelj** OpenCL (ang. host), ki koordinira izvajanje,
- vsaj ena **naprava** OpenCL (ang. device), ki je sposobna izvajati kodo OpenCL C.

Primeri naprav OpenCL so centralni procesorji ter grafične kartice. To so naprave, ki izvajajo prevedene programe OpenCL C. Vsaka naprava OpenCL se deli naprej v **računske enote** (ang. compute units). Pri centralnih procesorjih so to navadno jedra, pri grafičnih karticah pa tokovni procesorji (ang. stream processors). Računske enote se delijo naprej v **procesne elemente** (ang. processing elements), ki pri centralnih procesorjih predstavljajo jedra, pri grafičnih karticah pa tokovne enote (ang. stream units). Pri centralnih procesorjih ima vsaka računska enota torej le en procesni element, medtem ko pri grafični kartici vsaka računska enota vsebuje ogromno procesnih elementov. Platformni model je ponazorjen tudi na sliki 3.1.

[7]Kasneje bom takim funkcijam OpenCL C rekli ščepci (ang. kernel).



Slika 3.1: Platformni model OpenCL.

Samo procesiranje se izvaja na procesnih elementih, in sicer tako, da vsak procesni element izvaja en tok podatkov kot enota SIMD[8] ali kot enota SPMD[9].

Naenkrat lahko na gostitelju obstaja nameščenih več implementacij OpenCL, od katerih lahko vsaka programu OpenCL ponudi več različnih platform za izvajanje. Znotraj teh platform lahko soobstajajo naprave različnih vrst

[8]SIMD (ang. Single Instruction, Multiple Data) je programski model, kjer se ščepec OpenCL izvaja na vseh procesnih enotah OpenCL sočasno, in sicer tako da vsaka obdeluje svoje podatke, vse pa sočasno izvajajo iste ukaze.

[9]SPMD (ang. Single Program, Multiple Data) je programski model, kjer se ščepec OpenCL izvaja na vseh procesnih enotah OpenCL sočasno. Izvajanje poteka tako, da vsaka procesna enota obdeluje svoje podatke, vendar pa ima vsak procesni element svoj programski števec, kar zaradi vejitev v programu povzroči, da njihovo izvajanje divergira.

in tudi različic OpenCL.

3.1.2 Izvajalni model

Nalogo definiranja OpenCL okolja na gostitelju ter koordiniranja izvajanje ščepcev (ang. kernels) na napravah opravlja izvajalni model. To vključuje ustvarjanje konteksta OpenCL (ang. context) na gostitelju, komunikacijo in sinhronizacijo med gostiteljem in napravami ter definiranje izvajalnega okolja za hkratno izvajanje ščepcev na napravah.

Pri izvajalnem modelu se srečamo z naslednjima dvema pomembnima pojmoma.

- **Ščepec** oziroma ang. *kernel* je funkcija, ki je deklarirana v *programu* (program je zbirka ščepcev, konstantnih podatkov ter podpornih funkcij ki jih ščepci kličejo) ter izvajana na napravi OpenCL. Prepoznamo ga lahko preko predpone "*__kernel*" ali "*kernel*" pred imenom funkcije v izvorni kodi. Primer deklaracije ščepca lahko vidimo v kodi 3.1.

- **Kontekst** (ang. *context*) je okolje, znotraj katerega se izvajajo ščepci. Znotraj njega se upravlja tudi s pomnilnikom naprav ter izvaja sinhronizacija. Vključuje množico naprav OpenCL, pomnilnik dostopen tem napravam, pripadajoče pomnilniške podatke ter eno ali več ukaznih vrst (ang. *command queues*), ki se uporabljajo za vodenje izvajanja ščepcev, pomnilniških in sinhronizacijskih operacij na napravah.

  Pri konfiguraciji ukazne vrste lahko zahtevamo izvajanje ukazov po vrsti (ang. in-order) ali pa vrsti red izvajanja prepustimo sami napravi (ang. out-of-order) V prvem primeru se ukazi izvedejo v zaporedju kot pridejo v ukazno vrsto, medtem ko se v drugem primeru ukazi izvedejo takoj, ko je možno - čeprav se prejšnji ukazi še niso zaključili. Programer je v slednjem primeru primoran uporabljati eksplicitne sinhronizacijske ukaze.

Z oddajo ščepca v izvajanje s strani gostitelja se definira indeksni prostor (ang. index space), ki odraža naravo našega problema. Za vsako točko v

tem indeksnem prostoru se zažene primerek ščepca, ki mu pravimo **delovna enota** (ang. *work item*,WI), medtem ko točka v indeksnem prostoru določa **globalni indeks** te delovne enote. Vse delovne enote izvajajo isto programsko kodo, vendar pa se zaradi razlik v podatkih, ki jih obdelujejo, njihova pot izvajanja lahko spreminja. Delovne enote organiziramo v **delovne skupine** (ang. *work group*, WG), vsaki delovni enoti je dodeljen **lokalni indeks** znotraj njene delovne skupine ter **indeks delovne skupine**, ki ju lahko uporabimo za izračun globalnega indeksa delovne enote

$$\mathrm{ID}^{\mathrm{WI}}_{\mathrm{globalni}} = (N_{\mathrm{WG}}) \times \mathrm{ID}^{\mathrm{WG}} + \mathrm{ID}^{\mathrm{WI}}_{\mathrm{lokalni}}\,, \tag{3.1}$$

kjer je $\mathrm{ID}^{\mathrm{WI}}_{\mathrm{globalni}}$ globalni indeks delovne enote, N_{WG} velikost delovne skupine, $\mathrm{ID}^{\mathrm{WG}}$ indeks delovne skupine ter $\mathrm{ID}^{\mathrm{WI}}_{\mathrm{lokalni}}$ lokalni indeks delovne enote.

Indeksni prostor v OpenCL je N-dimenzionalen, kjer je $N \in \{1, 2, 3\}$. Več o podrobnostih indeksnega prostora si lahko preberete v [8, str. 24].

V nadaljevanju bomo delovnim enotam včasih rekli kar niti, delovnim skupinam pa skupine.

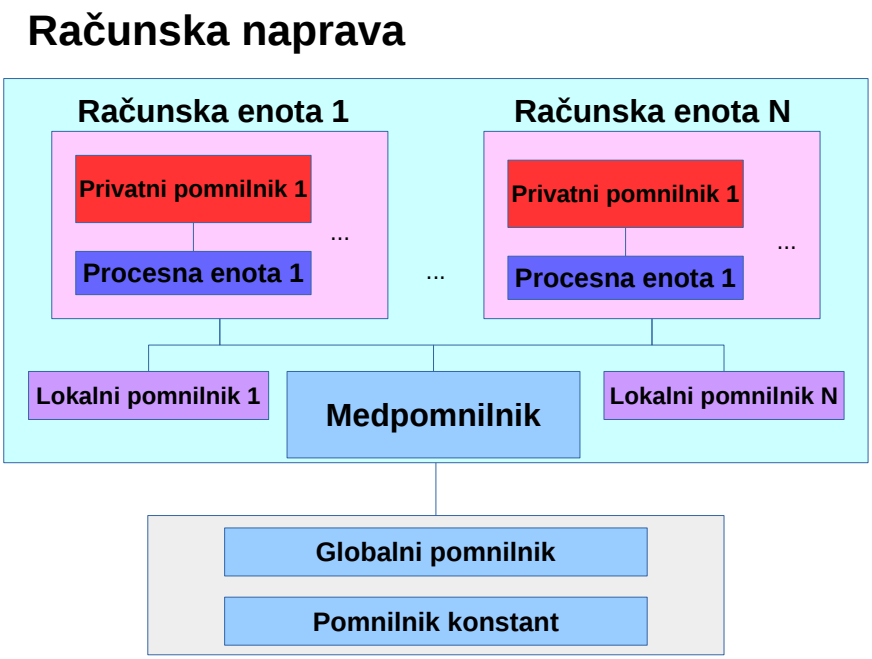
3.1.3 Pomnilniški model

Pomnilniški model določa abstraktno pomnilniško hierarhijo, ki se uporablja med izvajanjem kode OpenCL C na napravi. Ta model spominja na pomnilniške arhitekture grafičnih kartic, kar sicer ni omejilo uporabnost modela s strani drugih vrst naprav kot so raznovrstni pospeševalniki in procesorji.

OpenCL standard pozna štiri tipe pomnilnikov.

- **Globalni pomnilnik**. Vanj lahko pišejo ter iz njega lahko berejo niti iz vseh skupin. Prenosi podatkov so lahko predpomnjeni v skladu z zmožnostmi strojne opreme. Dostop do tega pomnilnika je praviloma najpočasnejši.

- **Pomnilnik konstant**. Del glavnega pomnilnika, ki ostane konstanten skozi izvajanje programa ter iz katerega lahko niti le berejo. Vrednosti vanj vpiše gostitelj pred izvajanjem.



Slika 3.2: Pomnilniški model OpenCL.

- **Lokalni pomnilnik**. Je lokalen za vsako skupino. Uporablja se ga za shranjevanje podatkov, ki si jih delijo vse niti v skupini. Uporablja se tudi za sinhronizacijo med nitmi v skupini.

- **Privatni pomnilnik**. Je privaten za vsako nit. Dostop do tega pomnilnika je praviloma najhitrejši.

Relacija med platformnim ter pomnilniškim modelom OpenCL je ponazorjena na sliki 3.2. Prve implementacije grafičnih kartic s podporo OpenCL so imele vse vrste pomnilnika (tudi privatnega) realizirane v istem fizičnem pomnilniku, medtem ko imajo sodobne grafične kartice vse vrste pomnilnika tipično realizirane fizično ločeno.

Interakcija med pomnilniškim prostorom gostitelja ter OpenCL naprav

poteka bodisi preko kopiranja podatkov med njima bodisi preko rezervacij pomnilnika. Kopiranje podatkov je lahko blokirajoče (ang. blocking) ali neblokirajoče (ang. non-blocking). V primeru prvega klic za kopiranje podatkov vrne nadzor programu po končanem kopiranju, v primeru drugega pa vrne nadzor programu takoj (četudi kopiranje podatkov še ni bilo izvršeno do konca).

Potencialni pomnilniški problemi na grafičnih karticah Pri programiranju grafičnih kartic se moramo zavedati, da pomnilniški dostop na grafičnih karticah ni zaščiten, kar pomeni da se napake zaradi dostopa do nedovoljenih ali neobstoječih pomnilniških naslovov ne preprečijo in ne poročajo uporabniku [18]. Na ta način lahko neka nit prepiše del operacijskemu sistemu namenjenega pomnilnika grafične kartice in povzroči vse od nepravilnega izrisa zaslona do zrušitve sistema.

Drugo nevarnost pri programiranju grafičnih kartic predstavlja dejstvo, da ima operacijski sistem prednost pred OpenCL pri rezervaciji in uporabi pomnilnika na grafični kartici. V kolikor se, na primer ob zvišanju ločljivosti zaslona, potrebna količina pomnilnika poveča preko količine pomnilnika, ki je na razpolago, si bo operacijski sistem dodelil del pomnilnika, ki je sicer na voljo OpenCL. Gonilnik grafične kartice bo takšne dogodke prepoznal in blokiral vsa nadaljnja izvajanja programov OpenCL.

Na nekaterih operacijskih sistemih kot je na primer Mac OS X Lion se utegne zgoditi tudi, da se prevelika količina statično rezerviranega pomnilnika[10] ne javi kot napaka pri prevajanju programa, zaradi česar je delovanje tako prevedenega programa negotovo [18]. Primer programa, ki znotraj ščepca rezervira preveliko količino statičnega pomnilnika, je predstavljen v kodi 3.1.

[10]Pri statično rezerviranemu pomnilniku se količina le-tega določi ob prevajanju, medtem ko se pri dinamično rezerviranem pomnilniku količina le-tega določi v času izvajanja.

Izpis 3.1: Nevaren program na Mac OS X Lion

```
__shared__ int staticnaShramba[NEKA_PREVELIKA_STEVILKA];
//Na Mac OS X Lion se prevede!

__kernel void mojeNezanesljivoJedro()
{
    int id = get_global_id(0);
    staticnaShramba[id] = -1;
}
```

3.1.4 Programski model

Programski model definira sočasnost izvajanja na OpenCL napravah (bodisi preko podatkovnega ali preko funkcijskega paralelizma) ter postavi dva načina sinhronizacije (sinhronizacija med nitmi znotraj ščepca ter sinhronizacija med ukazi znotraj ukazne vrste).

3.1.4.1 Podatkovni ter funkcijski paralelizem

OpenCL je bil ustvarjen za uporabo **podatkovnega paralelizma** (ang. data parallel programming model), ki računanje definira v smislu izvajanja zaporedja ukazov nad več elementi nekega pomnilniškega objekta [8, str. 29]. Indeksni prostor, ki je bil predstavljen v poglavju 3.1.2, definira delovne enote ter preslikavo med delovnimi enotami in podatki, ki jih želimo obdelati. V strogem podatkovnem paralelizmu bi bila preslikava bijektivna, ena delovna enota bi dobila natanko en del podatkov oziroma eno točko v indeksnem prostoru, medtem ko pa OpenCL dovoljuje tudi preslikave, kjer posamezna delovna enota dobi v obdelavo več delov podatkov oziroma točk v indeksnem prostoru.

OpenCL implementira hierarhičen podatkovni paralelizem, ki je lahko bo-

disi ekspliciten ali impliciten. V primeru prvega uporabnik specificira koliko delovnih skupin naj se ustvari ter koliko delovnih enot naj bo znotraj teh skupin, v primeru drugega pa uporabnik specificira le skupno število vseh delovnih enot in jih OpenCL sam porazdeli po delovnih skupinah.

Funkcijski paralelizem (ang. task parallel programming model) v OpenCL definira model, kjer se izvaja le ena instanca ščepca brez kakršnegakoli indeksnega prostora. Logično je ekvivalenten izvajanju ščepca v eni delovni skupini z le eno delovno enoto. Programerji s pomočjo tega modela paralelnost dosežejo s pomočjo izvajanja več ščepcev istočasno ali z uporabo vgrajenih vektorskih podatkovnih tipov.

3.1.4.2 Podpora za sinhronizacijo

OpenCL podpira dva načina sinhronizacije.

- Sinhronizacija med delovnimi enotami znotraj posamezne delovne skupine. To dosežemo s pomočjo **pregrad delovne skupine** (ang. work-group barriers), kjer se morajo vse delovne enote počakati preden nadaljujejo z izvajanjem. Mehanizma za sinhronizacijo med delovnimi skupinami OpenCL ne pozna.

- Sinhronizacija med ukazi v ukazni vrsti znotraj istega konteksta. Realizacija slednjega je možna na dva načina.

    1. S pomočjo **pregrad ukazne vrste**(ang. command-queue barrier) ob katerih se novi ukazi ne smejo izvršiti pred zaključkom starih ukazov in pred zaključkom vseh pomnilniških transakcij.
    2. S pomočjo ukazov za čakanje na dogodke. Vsi ukazi OpenCL, ki operirajo z ukazno vrsto (poganjanje ščepcev, ukazi za prenos podatkov, ...) imajo dodeljen dogodek, ki se zgodi ob njihovem zaključku, ter na katerega lahko čakajo preostali ukazi v ukazni vrsti.

3.2 Jezik OpenCL

Jezik OpenCL C je osnovan na standardu ISO/IEC 9899:1999 (bolj znanem kot C99) z določenimi dodatki in omejitvami [9]. Omenjenemu standardu so bili dodani spodaj predstavljeni dodatki.

- Dodatki za izboljšanje paralelizma:
  - vektorski podatkovni tipi (*uint4*, *double2*, ...),
  - funkcije za upravljanje z nitmi in skupinami (glej poglavje 3.1.2) kot je *uint get_work_dim()*, ki vrne število dimenzij indeksnega prostora,
  - sinhronizacija med nitmi s pomočjo pregrad in sinhronizacije dostopa do pomnilnika s pomočjo pomnilniških pregrad.
- Kvalifikatorji naslovnega prostora (*__global*, *__local*, *__constant*, *__private*).
- Dodatne vgrajene funkcije za skalarne in vektorske operande (veliko le-teh nadomešča funkcije iz standardnih C knjižnic, ki manjkajo v OpenCL C).

Omejitve OpenCL C pa so:

- uporaba kazalcev na funkcije ni dovoljena,
- uporaba polj, katerih število elementov se med izvajanjem lahko spreminja, ni dovoljena (velikost vseh polj in vseh struktur, ki jih vsebujejo, mora biti znana že ob času prevajanja),
- ni bitnih polj (*bit fields*),
- uporaba rekurzije ni mogoča,
- ni standardnih knjižnic C (ter s tem tudi ne funkcij, ki jih ponujajo knjižnice kot sta *errno.h* ter *math.h*) - izgubljene funkcionalnosti nadomešča OpenCL s svojimi vgrajenimi funkcijami.

Poglavje 4

Implementacija in rezultati meritev

V prvem poglavju si bomo najprej pogledali implementacijo IFGT v programskem jeziku C. Še posebej se bomo osredotočili na podrobnosti algoritma, ki niso bile omenjene v poglavju 2.3. Nato bomo opisali paralelizacijo implementacije na platformi OpenCL. Ločena obravnava sekvenčne implementacije nam bo dala boljši in jasnejši vpogled v delovanje implementacije IFGT. Boljši vpogled bomo imeli tudi v spremembe, ki jih bomo morali postoriti z namenom paralelizacije v OpenCL. Na koncu se bomo na kratko posvetili še opisu implementacije DGT, saj nam bo le-ta služila kot pomoč pri vrednotenju implementacije IFGT.

V drugem delu poglavja bomo najprej opisali testno okolje in proces testiranja implementacij, nato pa se bomo lotili samega testiranja in vrednotenja rezultatov.

4.1 Implementacija IFGT

V okviru diplomske naloge smo napisali program za izračun IFGT nad danimi podatki. Program je bil napisan najprej sekvenčno, nato pa je bil paraleliziran s pomočjo OpenCL. Tako sekvenčna kot paralelna implementacija

programa uporabljata vhodne in izhodne datoteke iste strukture, ki so lahko tako tekstovne kot binarne.

4.1.1 Delovanje implementacije ter formati datotek

Program za izračun IFGT se zažene iz ukazne vrstice s pomočjo ukaza 4.1, pri čemer ročno navedemo datoteko z vhodnimi podatki ter datoteko z izhodnimi podatki. Parameter `-b` na koncu ukaza pomeni, da sta tako vhodna kot izhodna datoteka binarni. V tej diplomski nalogi bomo uporabljali tekstovne datoteke, saj nam omogočajo enostavno preverjanje delovanja programa in pravilnost rezultatov. Program deluje tako, da najprej prebere vsebino vho-

Izpis 4.1: Ukaz za zagon IFGT

```
$./IFGT ./vhod/rocna_testna_3d_data.in \
./izhod/rocna_testna_3d_data.out -b
```

dne datoteke in si prebrane podatke shrani v pomnilnik, nato prebrane podatke obdela v skladu s parametri (pasovna širina h, meja napake ϵ oziroma tau, ...), ki so prav tako bili zapisani v vhodni datoteki. Rezultat izračuna shrani v izhodno datoteko.

Formata vhodnih ter izhodnih datotek sta enaka skozi vse programe, ki so bili implementirani v tem diplomskem delu. Primer vhodne datoteke se lahko vidi v izpisu 4.2, medtem ko se lahko primer izhodne datoteke (ki pripada omenjeni vhodni datoteki) vidi v izpisu 4.3. Znak `#` na začetku vrstice v vhodni ali izhodni datoteki pomeni komentar. Komentarji so namenjeni razumevanju vsebine datoteke in imajo smisel le v tekstovnih datotekah. Pri branju vhodne datoteke se komentarji prezrejo. V prvi vrstici vhodne datoteke, ki ni komentar, so zapisani sledeči podatki od leve proti desni:

1. dimenzionalnost problema (naravno število d),

2. število izvornih točk (naravno število N),

Izpis 4.2: Primer vhodne datoteke za IFGT

```
#Avtomatsko generirana datoteka vhodnih podatkov.
#Razlicica generatorja je bila 9py_rocna!
#ID vhodnih podatkov: 2015-10-21 16:01:50
#Vhodne tocke so enakomerno porazdeljene
#Utezi so enakomerno porazdeljene
#Izhodne tocke so enake vhodnim tockam

#dim stIzvTock stPonTock     h       tau
  3       4          4        0.4   2.2204e-06

#Izvorne tocke (dimenzije med (0,1), na koncu utez > 0)
0.417022004703 0.720324493442 0.000114374817345 1.0
0.302332572632 0.146755890817 0.0923385947688 1.0
0.186260211378 0.345560727043 0.396767474231 1.0
0.538816734003 0.419194514403 0.685219500397 1.0

#Ponorne tocke
0.417022004703 0.720324493442 0.000114374817345
0.302332572632 0.146755890817 0.0923385947688
0.186260211378 0.345560727043 0.396767474231
0.538816734003 0.419194514403 0.685219500397
```

3. število ponornih točk (naravno število M),

4. pasovna širina (pozitivno neničelno realno število h),

5. meja napake oziroma natančnost (pozitivno neničelno realno število ϵ (označujemo ga tudi s spremenljivko `tau`)).

V nadaljnjih N vrsticah vhodne datoteke, ki niso komentarji, so zapisane izvorne točke. V vsaki vrstici je opisana ena izvorna točka s sledečimi podatki,

Izpis 4.3: Primer izhodne datoteke za IFGT

```
#++++++++++++++++++++++++++++++++
#d=3 N=4 M=4 h=0.400000
#x1 x2 ... xd vrednost ;
#IFGT ver (2.1) ; Fri Dec 4 09:58:38 2015
#Za pripravo in racunanje sem sem porabil 0.0006 sekund.
0.417022 0.720324 0.000114 1.250739
0.302333 0.146756 0.092339 1.563364
0.186260 0.345561 0.396767 1.778090
0.538817 0.419195 0.685220 1.341065
#++++++++++++++++++++++++++++++++
```

ki si sledijo od leve proti desni:

1. d koordinat izvorne točke (pozitivna realna števila $x_i \in (0, 1)$ za $i = 1 \dots d$),

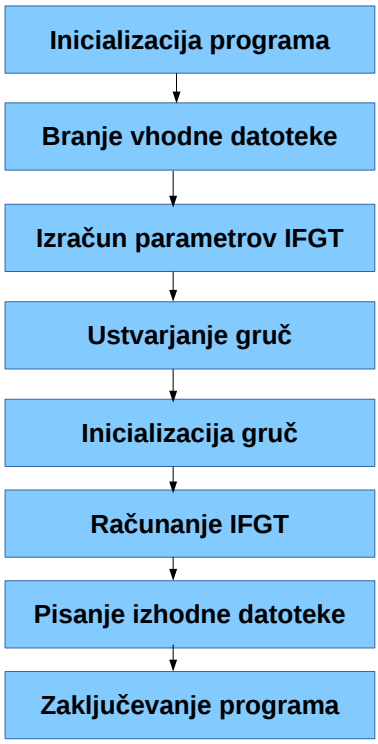
2. utež izvorne točke (pozitivno realno število q_i).

V zadnjih M vrsticah vhodne datoteke, ki niso komentarji, so zapisane ponorne točke, in sicer po ena ponorna točka na vrstico. Format zapisa ponorne točke je izpeljan iz zapisa izvorne točke, le da pri ponorni točki ni uteži.

Format zapisa izhodne datoteke je izpeljan iz formata zapisa vhodne datoteke. Prav tako kot pri branju vhodne datoteke se tudi pri branju izhodne datoteke prezrejo vse vrstice, ki predstavljajo komentarje. V izhodni datoteki so zapisane ponorne točke ter vrednosti transformacije IFGT v teh ponornih točkah, in sicer ena ponorna točka ter vrednost v tej ponorni točki na eno vrstico. Parametri, kot so število ponornih točk in pasovna širina, v izhodni datoteki niso zapisani. Čas trajanja izračuna je prav tako zapisan v izhodni datoteki.

4.1.2 Sekvenčna implementacija IFGT

Implementacija IFGT je osnovana na algoritmu 2.2. Postopek sekvenčnega delovanja implementacije IFGT v programskem jeziku C je orisan na sliki 4.1. V tem poglavju se bomo sprehodili skozi vse faze tega postopka.



Slika 4.1: Postopek delovanja sekvenčne implementacije IFGT.

Faza inicializacije programa V fazi inicializacije programa se obdelajo parametri ukazne vrstice, ki programu povedo lokacijo vhodne in izhodne datoteke ter format (tekstovni ali binarni) v skladu s poglavjem 4.1.1. V kolikor je program zagnan brez parametrov, se prikažejo navodila za uporabo.

Faza branja vhodne datoteke V fazi branja vhodne datoteke se obdela njena vsebina v skladu s formatom zapisanim v poglavju 4.1.1. Dimenzija,

pasovna širina, število izvornih točk, število ponornih točk ter meja napake se zapišejo v spremenljivke `d`, `h`, `N`, `M` ter `tau`. Izvorne točke, uteži izvornih točk ter ponorne točke se zapišejo v polje struktur `struct IzvornaTocka* izvorneTocke` in `struct PonornaTocka* ponorneTocke`, kjer ima vsaka izvorna in ponorna točka svojo lastno strukturo (`struct IzvornaTocka` ali `struct PonornaTocka`). Obe strukturi sta predstavljeni v izpisu 4.4. Nekatera polja strukture izvorne točke ostanejo v tej fazi izvajanja še nezapolnjena (`IDmojeGruce`, `lokacijaVGruci` ter `razdaljaDoCentra`).

Izpis 4.4: Strukturi ponorne in izvorne točke

```
struct IzvornaTocka
{
    double q;
    double *dimenzija;

    //ID gruce, ki ji tocka pripada
    int IDmojeGruce;

    //Moja lokacija znotraj gruce
    int lokacijaVGruci;

    //Razdalja do centra moje gruce
    double razdaljaDoCentra;
};

struct PonornaTocka
{
    double vrednost;
    double *dimenzija;
};
```

Faza izračuna parametrov IFGT V fazi izračuna parametrov IFGT se izračuna doseg podatkov `R`[1] ter število gruč `K`.

Doseg podatkov `R` lahko izračunamo neposredno iz podatkov (prebranih izvornih in ponornih točk v prejšnji fazi), vendar pa ima ta izračun časovno kompleksnost $O(M \times N)$, zaradi česar raje za vrednost `R` vzamemo kar njegovo zgornjo mejo $\sqrt{d}$, kjer je `d` število dimenzij. Ta pristop je bil uporabljen tudi v izvorni kodi implementacije IFGT [19].

Število gruč `K` izračunamo s pomočjo algoritma 4.3 predstavljenega v [15, str. 19] in izboljšanega v izvorni kodi [19]. Implementiran je v obliki funkcije `int izracunK(int d, double h, double tau, double R)`. Ta algoritem je optimiziran za enakomerno porazdeljene izvorne točke, v [20, str. 72] pa je predstavljen bolj prilagodljiv algoritem za izračun števila točk `K`, ki pa je tudi težji za implementacijo.

Faza ustvarjanja gruč Naslednja faza delovanja implementacije IFGT vključuje rezervacijo prostora za `K` gruč, uvrščanje izvornih točk v gruče z algoritmom za grupiranje na podlagi najbolj oddaljene točke (psevdo-koda je v algoritmu 2.1) ter optimizacijo gruč.

Struktura gruče je predstavljena v izpisu 4.5. Nekatera polja te strukture kot so `**monomi` ter `*CfKoeficienti` bodo zapolnjena šele v naslednji fazi. Vse gruče so shranjene v polju gruč `struct Gruca* tabelaGruc`.

Algoritem za grupiranje na podlagi najbolj oddaljene točke je implementiran v obliki funkcije `void farthestPointClustering(int N,int K,struct IzvornaTocka *izvorneTocke,struct Gruca *tabelaGruc,int d)`. Pred koncem izvajanja funkcije se pokliče funkcijo `void defragmentacijaGruc( struct Gruca *tabelaGruc, int N, int K)`, ki optimizira zasedenost polj točk po gručah in s tem zniža časovno kompleksnost preiskovanja vseh točk v posamezni gruči iz $O(N)$ na $O(\mathrm{StTock}_k)$, kjer StTock_k označuje število točk v k-ti gruči.

[1] Doseg podatkov `R` je enak maksimalni razdalji med katerimakoli dvema podatkovnima točkama (tako izvornima kot ponornima) znotraj prostora $[0, 1]^d$.

Algoritem 4.3 Algoritem za izračun števila gruč `K`

Vhod: Dimenzija `d`, pasovna širina `h`, meja napake ϵ (v programu `tau`) ter doseg podatkov `R`.

Izhod: Število gruč `K` ter maksimalno število odseka gruče $p_{\max}$.

1: $r \leftarrow \min(R, h\sqrt{\log(1/\epsilon)})$.
2: $K_{\text{meja}} \leftarrow \lceil 20R/h \rceil$.
3: Minimalna zrnatost $c_{\min} \leftarrow 10^{-16}$.
4: **Za** $k \leftarrow 1 : K_{\text{meja}}$ **naredi**
5: Izračunaj približek maksimalnega števila gruč $r_x \leftarrow k^{-1/d}$.
6: Izračunaj približek števila sosedov $n \leftarrow (r/r_x)^d$.
7: $p \leftarrow 1$, napaka $e \leftarrow 1$, vmesni člen izračuna $v \leftarrow 1$.
8: **Dokler** $e > \epsilon$ **naredi**
9: $a \leftarrow r_x$.
10: $b \leftarrow \min((a + \sqrt{(a^2 + 2 * p * h^2)})/2, r + r_x)$.
11: $v = v \times ((2 \times a \times b)/h^2)/p$.
12: $e = v \times e^{-((a-b)^2)/h^2}$.
13: $p \leftarrow p + 1$.
14: **Konec Dokler**
15: Shrani si vrednost $p_k \leftarrow p$.
16: Shrani si vrednost $c_k \leftarrow k + \log(k) + (1 + n) \times \binom{p-1+d}{d}$.
17: Če je vrednost $c_k < c_{\min}$, tedaj naj je $c_k \leftarrow c_{\min}$.
18: **Konec Za**
19: Izberi tak `K` $\leftarrow k$, za katerega je c_k minimalen.
20: Za isti k si shrani še $p_{\max} \leftarrow p_k$.

Izpis 4.5: Struktura gruče

```
struct Gruca
{
    int ID;
    //Indeks tocke ki je center gruce
    int centerGruce;
    //Radij gruce
    double Rn;
    //Tocka z najdaljso oddaljenostjo od centra gruce
    int tockaRn;
    //Tabela indeksov tock v tej gruci
    int *tockeVGruci;
    //Stevilo tock v tej gruci.
    int stTockVGruci;
    //Interakcijski radij
    double RnInt;
    //Stevilo odreza vrste te gruce
    int Pn;
    //Stevilo monomov te gruce
    int stMonomov;
    //Seznam monomov dimenzije d
    int **monomi;
    //Seznam koeficientov Cf(Zn)
    //sovpada z monomi
    double *CfKoeficienti;
};
```

Faza inicializacije gruč V fazi inicializacije gruč se najprej izračuna interakcijski radij vseh gruč `tabelaGruc[i].RnInt`. V ta namen uporabimo formulo tabelaGruc[i].RnInt $\leftarrow \min(R, \text{tabelaGruc[i].Rn} + h * \sqrt{\log(1/\text{tau})})$.

Sledi izračun odreza vrste `tabelaGruc[i].Pn` za vsako gručo s pomočjo algoritma, povzetega iz [20, str. 69], ter implementiranega v obliki funkcije `void racunanjeStevilaOdrezaVrste(struct Gruca *tabelaGruc,double tau, int K, double h)`. V algoritmu uporabimo približen izračun faktoriele $n!$ poljubno velikega števila n v plavajoči vejici, imenovanega *luschnyCF4* po njegovem avtorju Petru Luschnyju, ki ga dobimo v [12, Algoritem luschnyCF4].

Sledi rezervacija prostora za $\binom{\text{tabelaGruc[i].Pn}-1+d}{d}$ koeficientov in monomov[2], ki sovpadajo s koeficienti gruče `tabelaGruc[i].CfKoeficienti`. Vsak monom je predstavljen v obliki polja d nenegativnih celih števil (potenc posameznih spremenljivk monoma), katerih vsota je manjša ali enaka odrezu vrste gruče `tabelaGruc[i].Pn`. V algoritmu 2.2 monom označimo s spremenljivko α. Monome ustvarimo s pomočjo funkcije `void ustvarjanjeMonomov( struct Gruca *tabelaGruc, int K, int d)`.

V zadnjem koraku faze izračunamo koeficiente C_α^k k-te gruče s pomočjo funkcije `void racunanjeKoeficientovZn(struct Gruca *tabelaGruc,int K, double h, int N, struct IzvornaTocka *izvorneTocke, int d)`, ki sledi enačbi 2.11. Ta korak izračuna koeficientov gruč je zaradi neodvisne obravnave gruč in izvornih točk primeren za paralelizacija.

Faza računanja IFGT V fazi računanja IFGT uporabimo enačbo 2.11, implementirano v obliki funkcije `void racunanjeIFGT(struct Gruca *tabelaGruc, struct IzvornaTocka *izvorneTocke, struct PonornaTocka *ponorneTocke, int K, double h, int d, int N, int M)`, ki vrednosti ponornih točk shrani v polje struktur `ponorneTocke` za vsako ponorno točko posebej (v polje `ponorneTocke[i].vrednost`). Tako kot korak izračuna koeficientov gruč prejšnje faze je tudi ta faza primerna za paralelizacijo, saj ponorne točke obravnava povsem neodvisno in ločeno.

[2]Monom ali enočlenik je produkt spremenljivk od katerih ima vsaka svojo lastno potenco. Primeri monomov so xyz, x^2y^3z ter $-4xz^3$. Polinom je vsota monomov.

Faza pisanja izhodne datoteke V fazi pisanja izhodne datoteke se vrednosti ponornih točk iz polja struktur `ponorneTocke` (vsebuje spremenljivko `vrednost` ter polje `*dimenzija` za vsako ponorno točko posebej) zapišejo v izhodno datoteko.

Faza zaključevanja programa Namen zadnje faze je zapiranje datotek (ukaza `fclose(vhodnaDatoteka)` in `fclose(izhodnaDatoteka)`) ter sprostitev pomnilnika s pomočjo sistemskega klica `free()`.

4.1.3 Paralelizacija IFGT v OpenCL

V poglavju 4.1.2 smo obravnavali sekvenčno implementacijo IFGT, ki ne izkorišča nobenih dodatnih procesorskih jeder ter nobenih prednosti heterogenih sistemov, kot je na primer programirljiva grafična kartica. V tem poglavju predelamo obstoječo sekvenčno implementacijo s pomočjo OpenCL v obliko, ki je primerna za izkoriščanje prednosti heterogenih sistemov. Pri temu se poslužujemo pojmov in znanja o OpenCL iz poglavja 3.

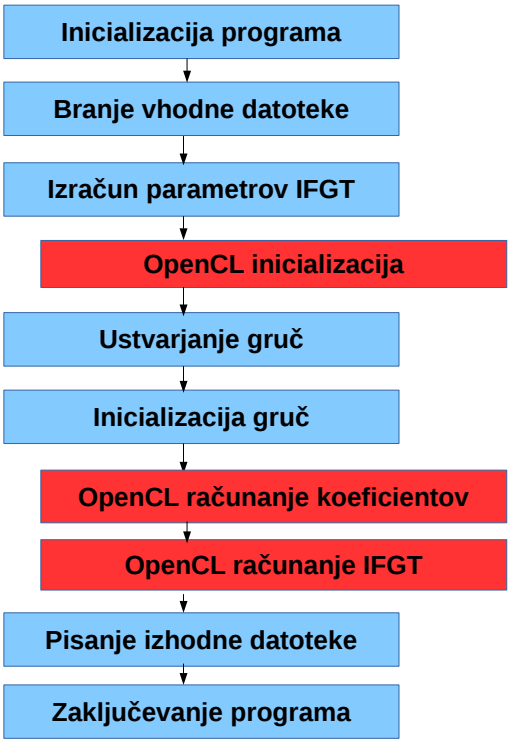
Sam postopek predelave obstoječe sekvenčne implementacije v OpenCL zaobsega:

- vgradnjo dodatnih faz v delovanje programa z namenom vzpostavljanja in upravljanja okolja OpenCL,
- predelavo podatkovnih struktur (struktur točk in gruč), tako da se jih lažje uporabi v ščepcih (napisanih v OpenCL C),
- pisanje ščepcev, ki se izvedejo na napravi OpenCL.

4.1.3.1 Dodatne faze implementacije OpenCL

Predelava implementacije v OpenCL zahteva poleg pisanja ščepcev, ki se izvajajo na napravah OpenCL, tudi vzpostavljanje in vzdrževanje ustreznega izvajalnega okolja OpenCL na gostitelju. V ta namen so bile fazam sekvenčnega programa iz poglavja 4.1.2 dodane še dodatne faze, katerih namen je skrb za okolje OpenCL.

Slika 4.2 prikazuje faze delovanja implementacije IFGT v OpenCL. Na njej so prikazane tudi dodatne faze glede na delovanje sekvenčne različice IFGT s slike 4.1. Prikazane faze OpenCL so pomaknjene na desno ter obarvane z rdečo barvo.



Slika 4.2: Postopek delovanja paralelne implementacije IFGT v OpenCL.

Faza inicializacije OpenCL Faza inicializacije OpenCL je na vrsti takoj ko je znano število gruč `K` iz faze izračuna parametrov. V sklopu te faze se najprej iz datoteke `IFGT_kernel_racunanjeCfKoeficientov.cl` prebere ščepec za računanje koeficientov C_α^k ter ustrezen ščepec za izračun IFGT - v skladu s številom gruč `K` je to bodisi ščepec `IFGT_kernel_racunanjeIFGT.cl` bodisi ščepec `IFGT_kernel_racunanjeIFGT_alternativa.cl`. Izvorna koda

obeh ščepcev se prebere v polje `char *source_str[2]`. Sledi poizvedba o platformi OpenCL (funkcija `clGetPlatformIDs()`) ter pridobitev ustrezne računske naprave OpenCL (funkcija `clGetDeviceIDs()`, ki jo pri nas uporabimo za pridobivanje grafične kartice). Pred prevajanjem ščepcev ustvarimo še kontekst (funkcija `clCreateContext()`) ter ukazno vrsto (funkcija `clCreateCommandQueue()`). Ščepce prevedemo s pomočjo funkcije `clBuildProgram()` v program[3], ki ga pred tem ustvarimo s klicem funkcije `clCreateProgramWithSource()`. Ščepci so sedaj pripravljeni za uporabo.

Faza paralelnega izračuna koeficientov Naslednja faza uporabe OpenCL, v kateri vidimo potencial za paralelizacijo, je faza izračuna koeficientov. Fazo inicializacije gruč iz sekvenčne implementacije IFGT zaključimo takoj po rezervaciji pomnilnika za koeficiente, saj bomo sedaj izračun koeficientov izvedli na grafični kartici.

Fazo paralelnega izračuna koeficientov C_α^k na grafični kartici pričnemo z rezervacijo prostora na grafični kartici s pomočjo funkcije `clCreateBuffer()`. Na grafično kartico prenesemo vse potrebne podatke (monome, izvorne točke, ...). Sledi definicija indeksnega prostora s pomočjo spremenljivk `local_item_size` ter `global_item_size`. Za vsako od `K` gruč ustvarimo svojo delovno skupino (*work group*), medtem ko si niti znotraj delovne skupine (*work item*) porazdelijo izračun vpliva vseh izvornih točk v gruči. Sledi ustvarjanje ščepca s pomočjo klica funkcije `clCreateKernel()`, nastavljanje parametrov ščepca (funkcija `clSetKernelArg()`) ter dokončno oddajanje ščepca v izvajanje s pomočjo klica funkcije `clEnqueueNDRangeKernel()`. Zagotovitev izvedbe ščepca ter počiščenje ukazne vrste zagotovimo s klicem funkcije `clFlush()`. Na koncu ščepec izbrišemo iz pomnilnika s pomočjo funkcije `clReleaseKernel()`.

Faza paralelnega izračuna IFGT Faza paralelnega izračuna IFGT popolnoma nadomesti fazo sekvenčnega izračuna IFGT iz poglavja 4.1.2. Njeno

[3]Beseda *program* v kontekstu standarda OpenCL pomeni zbirka ščepcev. Več o programih OpenCL je zapisano v poglavju 3.1.2.

delovanje je isto kot delovanje faze paralelnega izračuna koeficientov, le da si sedaj izberemo ščepec za izračun IFGT in indeksni prostor, znotraj katerega se izvaja v skladu s številom gruč `K`. V obeh primerih ustvarimo delovno skupino za vsako ponorno točko, do razlike pa pride pri deljenju dela med niti znotraj delovne skupine. Ščepec, ki ga uporabimo v primeru da je število gruč `K`> 32, porazdeli med niti izračun vpliva gruč na vrednost ponorne točke, medtem ko ščepec, ki ga uporabimo v nasprotnem primeru, porazdeli med niti izračun doprinosa koeficientov C_α^k vseh ustreznih gruč k vrednosti ponorne točke. Za to rešitev se odločimo, ker želimo čim bolj učinkovito zaposliti niti znotraj delovnih skupin. Več o tej rešitvi piše v poglavju 4.1.3.3.

4.1.3.2 Predelava podatkovnih struktur

Ščepci OpenCL so napisani v jeziku OpenCL C, ki ima v primerjavi s standardnim C kar nekaj omejitev (primeri omejitev so bili našteti v poglavju 3.2). Največji problem je uporaba polj, katerih število elementov ni znano ob času prevajanje[4], kot so `int **monomi`, `double *CfKoeficienti` ter `int *tockeVGruci` (izpis 4.5), ki v ščepcih ni mogoča.

Ta problem smo rešili tako, da smo vse strukture `struct Gruca`, `struct IzvornaTocka` ter `struct PonornaTocka` nadomestili s polji fiksne dolžine, ki jih bomo lahko uporabili pri računanju na grafični kartici. Na novo ustvarjena polja vidimo na izpisu 4.6. Polja več-dimenzionalnih polj (kot je polje dimenzij vseh točk) nadomestijo eno-dimenzionalna polja, kjer se do vrednosti i-tega elementa (npr. dimenzij i-te točke) dostopa preko indeksa $i \times d + j$, kjer j označuje j-to dimenzijo točke ter je d število dimenzij.

V polju `int* tockeVGruci` so shranjeni indeksi točk po vseh gručah. Ker število točk po posameznih gručah ni znano vnaprej, smo bili za vsako gručo primorani rezervirati prostor za zgornjo mejo števila točk, to je N. Poraba pomnilnika takšnega polja je torej $M(K \times N)$. Pri velikem številih izvornih točk `N` utegne to postati problem zaradi velikega števila podatkov, ki bi se moralo kopirati na grafično kartico pred izvajanjem ščepca.

[4]Pravi se jim tudi polja spremenljive dolžine (ang. variable length array, VLA).

Po končanem postopku ustvarjanja gruč zato indekse točk vseh gruč premaknemo v polje `int* optimiziraneTockeVGruci`, ki ima veliko manjšo in priročnejšo velikosti $M(N)$, saj vemo da je skupno število vseh točk `N`, na indekse točk znotraj posamezne gruče pa kažemo s pomočjo polja `int* odmikiOptimiziraneTockeVGruci`. Podobno taktiko uberemo tudi pri polju `double* CfKoeficienti` (katerega velikost je vsota števila koeficientov po vseh gručah) ter pri polju `int* monomi` (katerega velikosti je maksimalno število koeficientov po gručah).

4.1.3.3 Ščepci za paralelne dele programa

Kot smo videli v poglavju 4.1.3.1, na grafični kartici izvajamo fazo paralelnega izračuna koeficientov ter fazo paralelnega izračuna IFGT. Za izračun koeficientov uporabljamo en ščepec, za izračun IFGT pa imamo na voljo kar dva, med katerima izbiramo glede na število gruč `K`.

V vseh ščepcih vklopimo podporo za delo s števili v plavajoči vejici v dvojni natančnosti. To storimo s pomočjo direktive[5] `#pragma OPENCL EXTENSION cl_khr_fp64:enable`. Ščepci, ki ne uporabljajo števil v plavajoči vejici v dvojni natančnosti, se na grafičnih karticah sicer izvajajo hitreje, vendar bomo v poglavju 4.3.1 spoznali, da enojna natančnost števil v plavajoči vejici za naš problem ne zadostuje.

Glava funkcije ščepca za paralelni izračun koeficientov gruč je napisana v izpisu 4.7. S pomočjo `__constant` prevajalniku povemo, da podatkov ne bomo spreminjali in mu s tem omogočimo uporabo hitrejšega pomnilnika konstant v kolikor je le-ta na voljo. Končne vrednosti koeficientov gruč se bodo zapisale v glavni pomnilnik `global double* CfKoeficienti`, do katerega bomo imeli dostop tudi iz gostitelja. Koeficiente vsake gruče računa svoja delovna skupina, medtem ko si delovne enote znotraj delovnih skupin med seboj porazdelijo računanje koeficientov gruče s pomočjo zanke `for j`. Vsaka delovna enota gre nato skozi vse izvorne točke v gruči s pomočjo zanke `for t` in na koncu zapiše vrednost koeficienta v pomnilnik s pomočjo ukaza

[5]Direktiva je navodilo predprocesorju programskega jezika OpenCL C.

`CfKoeficienti[zacetniKoeficient[groupID]+j] = rez`.

Pri izračunu IFGT imamo na voljo dva ščepca. V skladu s številom gruč `K` izberemo primernejšega. Vsaki ponorni točki je dodeljena ena delovna skupina, ščepca pa se med seboj razlikujeta v načinu delitve dela med delovne enote znotraj delovne skupine. V primeru, da je gruč več kot 32, se uporabi ščepec, ki med delovne enote razdeli računanje doprinosov gruč k vrednosti ponorne točke. Sicer se uporabi alternativni ščepec, ki med delovne enote razdeli računanje doprinosov koeficientov gruč za vse gruče, ki so dovolj blizu ponorni točki (ponorna točka je v R_k^{int} okolici gruče). Delo se v tem primeru razdeli med delovne enote s pomočjo zanke `for j`. Razlog za uporabo alternativnega ščepca pri majhnem številu gruč je, da je število monomov oziroma koeficientov posamezne gruče tipično precej veliko (do 10^4 ali več) ter bo zaradi tega delitev izračuna le-teh bolje zaposlila delovne enote, kot bi jih delitev izračuna doprinosov gruč, ki bi zaposlila manj kot 32 delovnih enot na delovno skupino.

V primeru ščepcev za izračun IFGT uporabljamo tudi lokalni pomnilnik delovnih skupin in pregrade (omenjene v poglavju 3.1.4.2). Lokalni pomnilnik v glavi ščepca rezerviramo z `__local double* lokalniPomnilnik`. Uporabili ga bomo za redukcijo[6] doprinosov k vrednosti ponorne točke med delovnimi enotami. Delni rezultati niti se shranijo v lokalni pomnilnik s pomočjo `lokalniPomnilnik[localID] += doprinosGruce;`, kjer je `localID` indeks delovne enote znotraj delovne gruče.

Redukcija je realizirana s pomočjo pregrade `barrier(CLK_LOCAL_MEM_FENCE)`, ki sinhronizira dostop delovnih enot do lokalnega pomnilnika tako, da jih ustavi in prisili, da se počakajo med seboj. Šele ko zadnja delovna enota pride do pregrade, lahko delovne enote nadaljujejo z izvajanjem. Kodo redukcije prikazuje izvorna koda 4.8. Številka delovne enote je `localID`, `globalID` pa številka delovne skupine.

[6]Redukcija je proces kombiniranja (v našem primeru seštevanja) delnih rezultatov v končni rezultat.

Izpis 4.6: Polja nadomestijo strukture pri prehodu na OpenCL

```
//Izvorne tocke ter ponorne tocke;
double* izvorneTockeKoordinate;
double* qIzvorneTocke;
int* IDmojeGruceIzvorneTocke;
int* lokacijaVGruciIzvorneTocke;
double* razdaljaDoCentraIzvorneTocke;
double* ponorneTockeKoordinate;
double* vrednostPonorneTocke;

//Gruce; (i-ta gruca => i-ti indeks)
int* centerGruce;
double* Rn;
int* tockaRn;

//Tocke v vseh grucah so shranjene v enem polju.
//Indeksi tock posamezne gruce i --> [i*N...(i+1)*N)
//Vsaka gruca ima lahko najvec N tock!
int* tockeVGruci;
int* stTockVGruci;

//Defragmentacija gruc!
//Namesto M(K*N) je poraba pomnilnika M(N)
int* optimiziraneTockeVGruci;
int* odmikiOptimiziraneTockeVGruci;
double* RnInt;
int* Pn;
int* stMonomov;
int* monomi;
double* CfKoeficienti;
//Indeks zacetnega koeficienta za gruco i
int* zacetniKoeficient;
```

Izpis 4.7: Glava ščepca za izračun koeficientov

```
__kernel void IFGT_kernel_racunanjeCfKoeficientov(
    int K, double h, int N, int d, __constant int*
       stMonomov,
    __constant int* monomi, __constant int*
       stTockVGruci,
    __constant int* centerGruce, __constant double*
       izvorneTockeKoordinate,
    __constant double* qIzvorneTocke, global double*
       CfKoeficienti,
    __constant int* zacetniKoeficient, __constant int*
       optimiziraneTockeVGruci,
    __constant int* odmikiOptimiziraneTockeVGruci
    )
```

Izpis 4.8: Redukcija v ščepcu za izračun IFGT

```
//Sedaj imas doprinos posameznih delovnih enot WI
    shranjen v lokalnem pomnilniku
barrier(CLK_LOCAL_MEM_FENCE);

//log2(NumWorkItem) korakov
for(i=steviloNiti/2; i>=1; i=i/2)
{
    //Vsaka nit ki je znotraj intervala [0,i)
        sesteje localID in localID+i ter shrani v
        lokalniPomnilnik[localID]
    if(localID < i)
    {
        lokalniPomnilnik[localID] =
            lokalniPomnilnik[localID] +
            lokalniPomnilnik[localID+i];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}
//Glavna nit pise v glavni pomnilnik
barrier(CLK_LOCAL_MEM_FENCE);
if(localID == 0)
{
    vrednostPonorneTocke[groupID] =
        lokalniPomnilnik[0];
}
```

4.2 Implementacija DGT

Implementacija DGT deluje na isti način in uporablja iste formate vhodnih ter izhodnih datotek kot so bili opisani v poglavju 4.1.1. Namen takšne rešitve je poenostavljenje testiranja in primerjanja implementacije DGT z implementacijo IFGT, saj lahko obe uporabljati isto testno okolje ter iste vhodne datoteke, prav tako pa je mogoča neposredna primerjava njunih izhodnih datotek.

Sekvenčna implementacija DGT deluje neposredno po enačbi 2.2 in uporablja isti dve prvi ter zadnji fazi sekvenčne implementacije IFGT: inicializacijo programa, branje vhodne datoteke, pisanje izhodne datoteke ter fazo zaključevanja programa. Osrednja faza programa je faza izračuna DGT (izvorna koda 4.9).

Izpis 4.9: Izračun DGT

```
for(i=0;i<M;i++)
{
    vsota=0.0;
    for(j=0;j<N;j++)
    {
        vsota += izvorneTocke[j].q * exp( (-1.0)*pow(
            razlikaVektorskaNorma(ponorneTocke[i],
            izvorneTocke[j],d) ,2)/(pow(h,2)));
    }
    ponorneTocke[i].vrednost = vsota;
}
```

OpenCL implementacija izračuna DGT doda še fazo vzpostavljanja okolja OpenCL in prestavi izračun DGT v ščepec. Tako kot implementacija IFGT v OpenCL tudi implementacija DGT v OpenCL za vsako ponorno točko ustvari svojo delovno skupino, medtem ko si delovne enote med seboj

porazdelijo računanje vpliva vseh N izvornih točk na vrednost ponorne točke. Tako kot izračun IFGT tudi izračun DGT v OpenCL uporablja redukcijo in pregrade za seštevanje delnih rezultatov vrednosti ponornih točk (podobno kot v izvorni kodi 4.8).

4.3 Testiranje

V tem poglavju se bomo posvetili testiranju sekvenčne in paralelne implementacije IFGT in DGT, ki sta bili opisani v poglavju 4.1. Najprej si bomo pogledali testno okolje in orodja s katerimi smo si pomagali, nato pa se bomo posvetili predstavitvi in analizi rezultatov.

4.3.1 Testno okolje

Testno okolje implementacij zaobsega *testni direktorij*, ki vsebuje mape *vhod* (vsebuje vhodne datoteke s podatki) in *izhod* (po koncu izračuna bo vsebovala izhodne datoteke s podatki), datoteke z izvorno kodo implementacij ter *testno skripto*, napisano v programskem jeziku Python. Le-ta rekurzivno preišče direktorij *vhod*, za vsako najdeno vhodno datoteko požene vsako od štirih implementacij (DGT in IFGT, sekvenčno ter v OpenCL) in ustvari ustrezno izhodno datoteko v direktoriju *izhod*.

Implementacije so bile prevedene s pomočjo sledečih ukazov:

- DGT: `gcc DGT.c -O3 -o DGT -lm`,
- DGT v OpenCL: `gcc -lOpenCL -O3 -o DGT_OpenCL DGT_OpenCL.c`,
- IFGT: `gcc IFGT.c -O3 -o IFGT -lm`,
- IFGT v OpenCL: `gcc -lOpenCL -O3 -o IFGT_OpenCL IFGT_OpenCL.c`.

Meritve so bile izvedene preko povezave SSH na računalniku v Laboratoriju za adaptivne sisteme in paralelno procesiranje Fakultete za računalništvo in informatiko v Ljubljani. Testni računalnik je bil opremljen s procesorjem

Intel(R) Xeon(R) CPU E5-2620 z 2×6 jedri, tehnologijo večnitenja (ang. hyperthreading), frekvenco delovanja 2.0GHz ter 64GB pomnilnika. Grafična kartica je bila NVIDIA Tesla K20m s 5GB grafičnega pomnilnika, 13 SIMD jedri ter podporo za standard OpenCL različice 1.1. Pri testiranju smo uporabljali sledeče konfiguracije OpenCL ščepcev:

- računanje koeficientov: 256 delovnih enot na delovno skupino,
- računanje IFGT (osnovno): 256 delovnih enot na delovno skupino,
- računanje IFGT (alternativno): če $K < 32$, je bilo 32 delovnih enot na delovno skupino, sicer je bilo $32 * 2^a$ delovnih enot na delovno skupino (kjer je a bilo največje tako nenegativno celo število, da neenačba $32 * 2^a < K$ še vedno velja),
- računanje DGT: 256 delovnih enot na delovno skupino.

Časi izvajanja implementacij so se začeli meriti pred začetkom faze izračuna parametrov algoritma ter končali po koncu izračuna IFGT. Vključevali so tudi vse vmesne rezervacije pomnilnika s pomočjo `malloc()`, vzpostavljanje okolja OpenCL ter prenose podatkov med napravo OpenCL in gostiteljem. V kolikor pri meritvah časov izvajanja programa uporabimo besedno zvezo na procesorju, to pomeni da so se vse faze izračuna izvedle na procesorju, v kolikor pa uporabimo besedno zvezo na grafični kartici, pa to pomeni da so se ustrezne faze OpenCL izvedle na grafični kartici, medtem ko so se vse ostale faze izračuna seveda še vedno izvedle na procesorju.

Pri testiranju smo si pomagali še z dvema skriptama, napisanima v programskem jeziku Python. Prva med njima je služila kot *primerjalnik izhodov*, ki primerja vrednosti vsake ponorne točke med dvema različnima implementacijama in izpiše, ali je največja razlika med njima manjša od meje napake ϵ ali ne. V kolikor obstaja ponorna točka, v kateri je razlika med implementacijama večja od meje napake, se izpiše tudi največja najdena napaka.

Določimo dve različni metriki, s katerima lahko med seboj primerjamo izhodne datoteke implementacij DGT ter IFGT in s tem vrednotimo pravilnost implementacije:

- *največja absolutna napaka implementacije*

$$\text{napaka}_{\max} = \max_{j=1\ldots M} |\tilde{G}(t_j) - G(t_j)| \quad , \tag{4.1}$$

- *relativna kvantitativna napaka implementacije* - število napak večjih od meje napake ϵ deljeno s številom vseh ponornih točk

$$\text{KvantNapaka}_{\text{relativna}} = \frac{\#\{t_j : |\tilde{G}(t_j) - G(t_j)| > \epsilon\}}{M} \quad . \tag{4.2}$$

Metriko relativna kvantitativna napaka implementacije je zaradi meje napake, ki jo garantira metoda IFGT (poglavje 2.4), smiselno uporabljati le v primeru, ko je vsota uteži vseh izvornih točk enaka 1. To najlažje jamčimo tako, da postavimo uteži vseh izvornih točk na enako vrednost, in sicer na $q_i = \frac{1}{N}$.

Druga skripta je bila *generator testnih podatkov*. Le-ti so ustrezali naši notaciji DGT, omenjeni v poglavju 2.1. Ob uporabi knjižnice `numpy` je znala izvorne točke ustvariti naključno porazdeljene na več načinov, ki so opisani spodaj.

- Enakomerna porazdelitev v $[0,1]^{\dim}$.

- Porazdelitev, kjer so točke zgoščene okoli več centrov z uporabo večrazsežne normalne porazdelitve. Uporabnik vnese standardni odklon in število centrov. V kolikor obstajajo generirane izvorne točke $x \notin [0,1]^{\dim}$, se jih vstavi v $[0,1]^{\dim}$ s pomočjo deljenja z največjo koordinato po absolutni vrednosti in negiranja vseh negativnih koordinat.

  Pri testiranju smo uporabljali dva različna primera. V prvem so bile izvorne točke večdimenzionalno normalno porazdeljene okoli od 1 do 10 centrov, v drugem pa so bile porazdeljene okoli 5 centrov, medtem ko smo spreminjali standardni odklon $\sigma \in [0{,}001, 0{,}1]$.

- Porazdelitev po krogu okoli točke (0.5,0.5), kjer uporabnik vnese polmer kroga[7].

[7]Formula je bila povzeta po `http://stackoverflow.com/questions/9879258/how-can-i-generate-random-points-on-a-circles-circumference-in-javascript`.

Pri testiranju smo spreminjali polmer kroga r.

- Porazdelitev po sferi okoli točke (0.5,0.5,0.5), kjer uporabnik vnese polmer sfere[8].

  Pri testiranju smo spreminjali polmer sfere r.

Ustvarjene ponorne točke so bile bodisi enake izvornim točkam bodisi enakomerno porazdeljene. Uteži so bile bodisi enake 1, $1/N$, bodisi enakomerno porazdeljene. Pasovno širino h je vnesel uporabnik, lahko pa se je izračunala tudi s pomočjo Silvermanovega pravila[9].

4.3.2 Meritve

Meritve časov izvajanja implementacij smo izvajali na več različnih načinov. V prvem delu tega poglavja bomo predstavili rezultate meritev, v katerih so bile izvorne točke enakomerno porazdeljene, v drugem delu pa rezultate meritev, v katerih so bile izvorne točke porazdeljene tudi na druge načine.

V namen opravljanja meritev smo vsak posamezen primer uporabe implementiranih metod opisali s pomočjo pasovne širine h, števila izvornih in ponornih točk $N = M$, meje napake ϵ ter števila dimenzij d. Pri meritvah smo naenkrat spreminjali eno samo spremenljivko, nato pa smo preverjali odvisnosti časa izvajanja od spreminjanja vrednosti te spremenljivke in na podlagi videnega podali opažanja. Ta opažanja smo nato poizkušali potrditi ali ovreči s pomočjo dodatnih meritev.

[8]Formula je bila povzeta po `http://vpython.org/contents/contributed/vspace.py` (C) Ronald Adam 3/20/2005; ron@ronadam.com.

[9]Silvermanovo pravilo (*Silverman's rule of thumb* oziroma *normal distribution approximation*) nam omogoča izračun optimalne pasovne širine h za naše podatke preko formule

$$h = \left(\frac{4\sigma^5}{3n}\right)^{\frac{1}{5}} \approx 1.06\hat{\sigma}n^{-\frac{1}{5}},$$

kjer je $\hat{\sigma}$ standardna deviacija podatkov.

4.3.2.1 Meritve na enakomerno porazdeljenih izvornih točkah

V poglavju 2.4.1 smo postavili trditve o naravi IFGT, ki so bile povzete iz literature. Le-te pravijo, da naj bi se IFGT odrezal slabo pri problemih z majhno pasovno širino, in sicer bi naj se meja pasovne širine, pri kateri je še uporaben[10] dvigala z nižanjem meje napake ϵ (pri $\epsilon = 10^{-6}$ se omenja meja uporabnosti pri $h = 0{,}5$). Z naraščanjem dimenzionalnosti problema naj bi uporabnost IFGT postajala zelo odvisna od višine pasovne širine.

Po izkušnjah iz preizkušanja delovanja IFGT že med samim programiranjem je zelo pomemben faktor pri določanju časa njegovega delovanja število odreza vrste p_k (enačba 2.12), ki za vsako gručo Z_k določi število monomov in koeficientov gruče. Na podlagi opažanj smo v namene meritev razdelil pasovne širine h v dve skupini:

- majhne pasovne širine pri $h < 1$,
- velike pasovne širine pri $h \geq 1$.

Pomemben pojem pri preučevanju meritev je *pohitritev* paralelnega programa glede na sekvenčni program. Pri nas jo izračunamo s pomočjo enačbe

$$S = \frac{t_{\mathrm{CPE}}}{t_{\mathrm{GPE}}}\,, \tag{4.3}$$

kjer S označuje pohitritev, t_{CPE} označuje čas trajanja delovanja programa na procesorju, t_{GPE} pa čas trajanja delovanja programa, pri katerem so se ustrezne faze OpenCL izvedle na grafični kartici namesto na procesorju.

Meritve pri fiksni pasovni širini in meji napake V prvem sklopu meritev smo fiksirali pasovno širino na $h = 1{,}0$ ter mejo napake na $\epsilon = 10^{-3}$. V skladu z zgoraj navedenimi opažanji iz literature bi pričakovali, da se bo pri naraščanju števila izvornih in ponornih točk $N = M$ čas izvajanja IFGT pri nizkih dimenzijah le počasi večal, medtem ko bo pri višjih dimenzijah čas izvajanja postal zelo dolg.

[10] Uporaben v smislu razumnega časa izvajanja.

d	N	DGT (s)	DGT (OpenCL) (s)	IFGT (s)	IFGT (OpenCL) (s)
1	25000	31,1233	0,3130	0,0667	0,3300
	50000	124,5467	0,5050	0,1300	0,4550
	75000	280,3733	0,8440	0,1800	0,5840
	100000	497,5267	1,3220	0,2067	0,7150
	125000	777,8500	1,9390	0,2900	0,8420
	150000	1122,1767	2,6970	0,3133	0,9710
	175000	1519,2667	3,5380	0,3633	1,1010
	200000	1992,3633	4,5540	0,4200	1,2440
2	50000	131,9033	0,6200	1,9000	0,5070
	75000	296,4433	1,0640	2,8500	0,6660
	100000	527,3100	1,6900	3,7600	0,8240
	125000	823,4433	2,5100	4,6867	0,9550
	150000	1187,3633	3,4910	5,6200	1,1090
	175000	1616,3733	4,6770	6,5500	1,2800
	200000	2110,4767	6,0170	7,5267	1,4240
3	50000	131,9900	0,9610	33,5900	1,4170
	75000	297,7733	1,8450	50,4100	2,1020
	100000	527,6933	3,1100	67,3900	2,6670
	125000	826,5833	4,6890	84,0600	3,2900
	150000	1189,5367	6,6660	100,5433	3,9200
	175000	1614,8400	8,9460	117,0867	4,5250
	200000	2110,6533	11,6200	133,8500	5,1530
4	50000	135,7867	1,4590	362,9567	8,1420
	75000	305,0333	2,9460	568,5033	12,4730
	100000	542,6233	5,0520	791,2900	16,7180
	125000	915,7433	7,7260	988,3233	20,7970
	150000	1221,5800	11,0060	1233,8533	25,2520
	175000	1674,1567	14,8920	1503,8967	29,9050
	200000	2180,0767	19,3710	1716,2033	34,3150

Tabela 4.1: Rezultati meritev časov izvajanja pri $d = 1 \ldots 4$, $\epsilon = 10^{-3}$ ter $h = 1{,}0$.

V tabeli 4.1 lahko vidimo rezultate meritev časov izvajanja vseh naših implementacij. Meritve potrjujejo naše pričakovanje glede zviševanja časa trajanja izvajanja IFGT z višanjem dimenzionalnosti problema. Pohitritev implementacije IFGT v OpenCL v primerjavi s sekvenčnim IFGT pri $N = 200000$ je pri $d = 1$ enaka 0,3376, pri $d = 2$ 5,3139, pri $d = 3$ 25,9751, pri $d = 4$ pa že kar 50,0312. V primeru $d = 1$ je torej IFGT na procesorju še hitrejši, v višjih dimenzijah pa več ne. Pri $d = 4$ sicer implementacija DGT v OpenCL deluje že enkrat hitreje kot implementacija IFGT v OpenCL.

Pri $h = 1{,}0$ je implementacija IFGT v OpenCL torej spodobno hitra, še posebej v primerjavi s sekvenčno implementacijo IFGT. Zanima nas, kako se odreže še pri ostalih pasovnih širinah.

Meritve pri fiksnem številu točk in meji napake Poizkusimo sedaj napraviti meritve pri spreminjajoči-se pasovni širini $h \in \{0{,}1, 0{,}5, 1{,}0, 2{,}5, 5{,}0\}$ ter pri spreminjajoči dimenzionalnosti problema $d \in \{1, 2, 3, 4, 5\}$. Meja napake ϵ naj bo 10^{-3}, število točk pa $N = M = 100000$.

Iz tabele 4.2 je jasno razvidno, da z višanjem pasovne širine implementaciji IFGT (tako sekvenčna kot paralelna) postajata čedalje hitrejši tudi v višjih dimenzijah. Pri pasovni širini $h = 0{,}1$ je implementacija IFGT v OpenCL uporabna v problemih z $d = 1$ ter $d = 2$, pri $h = 0{,}5$ postane uporabna tudi v $d = 3$. Največjo učinkovitost v primerjavi s sekvenčno implementacijo IFGT doseže pri $h = 0{,}5$ v $d = 3$, ko je pohitritev paralelizacije v OpenCL kar 121-kratna. Pri $d = 5$ in $h = 5{,}0$ postane IFGT v OpenCL počasnejša od sekvenčne IFGT.

Meritve pri različnih mejah napak Do sedaj smo naše meritve izvajali pri konstantni meji napake ϵ, sedaj pa si poglejmo še kako se naše implementacije obnašajo pri različnih mejah napake ϵ. Dimenzija testnih podatkov je $d = 1$.

V tabeli 4.3 so zbrani rezultati meritev. Ko konvergira $h \to \infty$ gre čas izvajanja IFGT v OpenCL proti 0,86 ter je tako vedno boljši od časa izvajanja DGT v OpenCL. Čas izvajanja IFGT se veča skupaj z manjšanjem meje

h	d	DGT (s)	DGT (OpenCL) (s)	IFGT (s)	IFGT (OpenCL) (s)
0.1	1	546,8167	1,3220	1,5767	0,4490
	2	575,0300	1,6860	106,3167	3,0540
	3	576,9467	3,0890	7738,0733	422,8760
	4	589,6933	/	/	/
	5	602,5433	/	/	/
0.5	1	497,9833	1,3360	0,6200	0,5930
	2	524,7533	1,7150	21,1533	0,7460
	3	526,4467	3,1160	1091,1767	9,0090
	4	541,0900	5,0520	/	187,5400
	5	553,7767	6,1760	/	/
1.0	1	497,5133	1,3280	0,2400	0,7150
	2	526,3967	1,7160	3,7733	0,8280
	3	528,2833	3,1250	67,0633	2,6730
	4	540,5067	5,0530	789,1633	16,7420
	5	553,4933	6,1880	6834,5900	251,1080
2.5	1	545,6567	1,3570	0,1100	0,8820
	2	573,6933	1,7120	0,4633	1,3490
	3	573,7733	3,1280	2,0200	1,8130
	4	589,8567	5,0460	7,9567	1,7800
	5	601,7833	6,1910	23,5500	5,2300
5.0	1	546,3133	1,3480	0,0667	0,8910
	2	573,9667	1,7020	0,1633	1,3450
	3	574,2367	3,1260	0,4267	1,3650
	4	589,0500	5,0520	1,8333	1,7730
	5	600,3600	6,2130	2,3400	2,7500

Tabela 4.2: Rezultati meritev časov izvajanja pri $N = 100000$ ter $\epsilon = 10^{-3}$.

ϵ	h	DGT (OpenCL) (s)	IFGT (s)	IFGT (OpenCL) (s)
10^{-2}	0,1	1,3430	0,9233	0,4590
	0,25	1,3420	0,8500	0,4800
	0,5	1,3360	0,4667	0,5330
	0,75	1,3270	0,2900	0,7110
	1,0	1,3220	0,1933	0,7120
	2,5	1,3160	0,0667	0,8810
	5,0	1,3310	0,0467	0,8810
	7,5	1,3290	0,0433	0,8690
	10,0	1,3250	0,0433	0,8920
10^{-4}	0,1	1,3120	2,0167	0,4510
	0,25	1,3260	1,5467	0,4820
	0,5	1,3190	0,7133	0,6000
	0,75	1,3180	0,4200	0,7090
	1,0	1,3350	0,3033	0,7340
	2,5	1,3270	0,1100	0,9090
	5,0	1,3260	0,0633	0,8860
	7,5	1,3330	0,0633	0,8810
	10,0	1,3360	0,0433	0,8870
10^{-6}	0,1	1,3290	2,8100	0,4790
	0,25	1,3250	1,9467	0,5240
	0,5	1,3290	0,8867	0,6040
	0,75	1,3510	0,5367	0,7160
	1,0	1,3400	0,3767	0,7030
	2,5	1,3220	0,1767	0,8880
	5,0	1,3300	0,1067	0,8750
	7,5	1,3200	0,0933	0,8680
	10,0	1,3270	0,0500	0,8730

Tabela 4.3: Rezultati meritev časov izvajanja pri $N = 100000$ ter $d = 1$.

napake ϵ, vendar pa je pri $h \geq 0{,}5$ vedno boljši od časa izvajanja DGT v OpenCL. Velika večina časa izvajanja IFGT v OpenCL se porabi v ščepcu za izračun IFGT.

Razlog za višanje časa izvajanja v implementaciji OpenCL je zelo majhna dimenzija testnega problema $d = 1$. V višjih dimenzijah bi časi izračunov v ščepcih OpenCL zasenčili čas prenosa podatkov po vodilu ter čas prevajanja samih ščepcev.

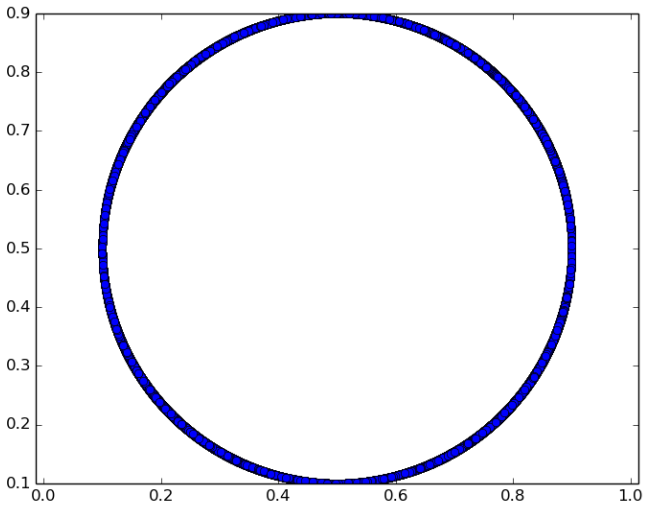
4.3.2.2 Meritve na drugih porazdelitvah izvornih točk

V tem poglavju bomo predstavili rezultate meritev časov delovanja naših implementacij na različnih porazdelitvah izvornih točk, ki so bile predstavljene v poglavju 4.3. Ponorne točke so bile enake izvornim točkam. Meja napake je bila $\epsilon = 10^{-5}$.

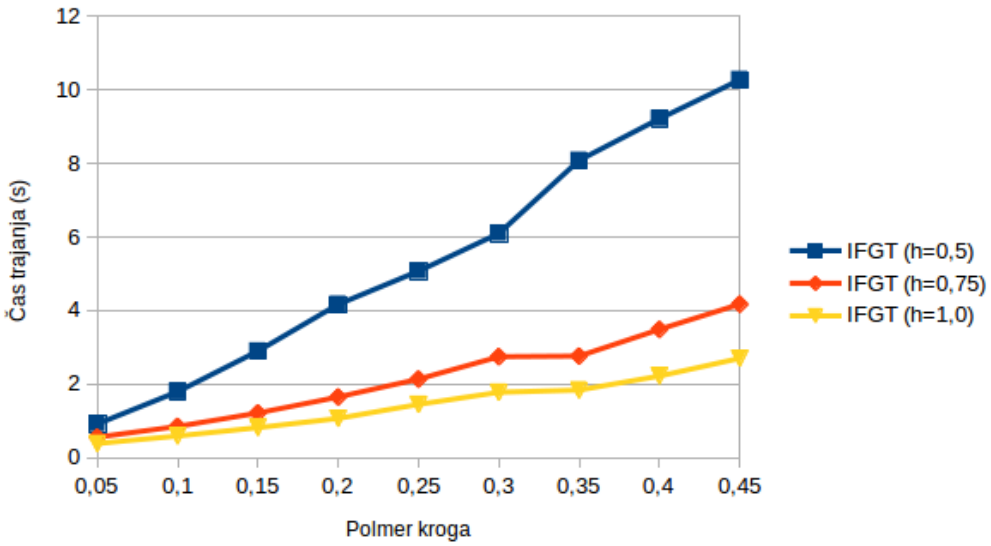
Porazdelitev izvornih točk po robu kroga Ustvarili smo 50000 izvornih točk, ki so bile porazdeljene po robu kroga s polmerom $r \in \{0{,}05, \ldots 0{,}45\}$. Primer porazdelitve pri $r = 0{,}4$ lahko vidite na sliki 4.3.

Rezultati meritev so predstavljeni na slikah 4.4 ter 4.5. Rezultata obeh slik sta kombinirana na sliki 4.6, v kateri je pohitritev S izračunana po formuli 4.3. Pri sekvenčni implementaciji IFGT lahko opazimo višanje časa izvajanja ob višanju polmera kroga. Do tega pride, ker se pri vseh polmerih kroga ustvari isto število gruč, poveča pa se radij samih gruč. Večji radij gruč povzroči tudi večje število odseka vsake posamezne gruče P_k in s tem tudi večje število koeficientov C_α^k v posameznih gručah.

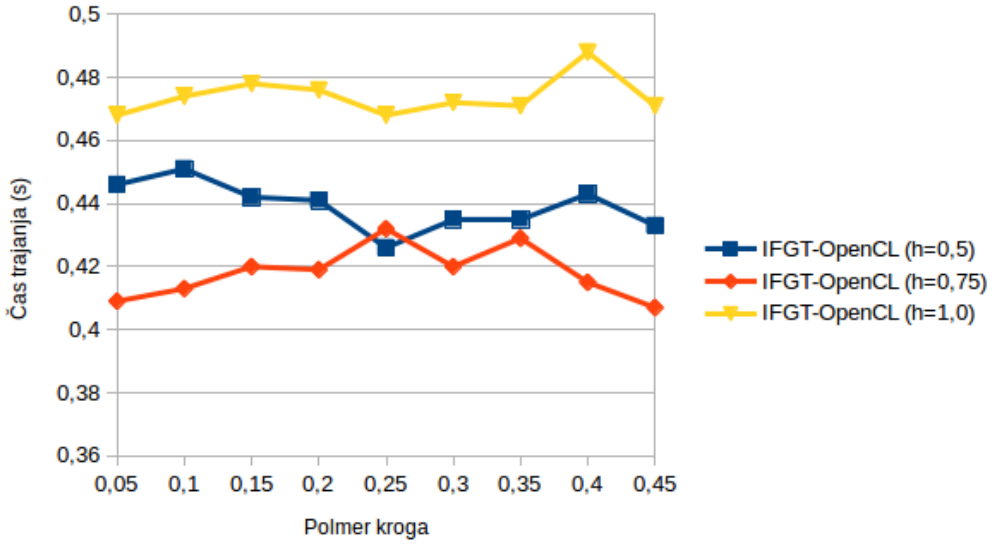
Pri implementaciji IFGT v OpenCL ne opazimo nobenega očitnega višanja časa izvajanja v odvisnosti od polmera kroga, vendar pa to lahko pripišemo premajhnemu številu samih izvornih točk. Prevajanje ščepcev OpenCL in prenosi po vodilu med grafično kartico in pomnilnikom bi pri večjih številih izvornih točk predstavljali le majhen delež opravljenega dela in časa v primerjavi s samim računanjem IFGT na grafični kartici.



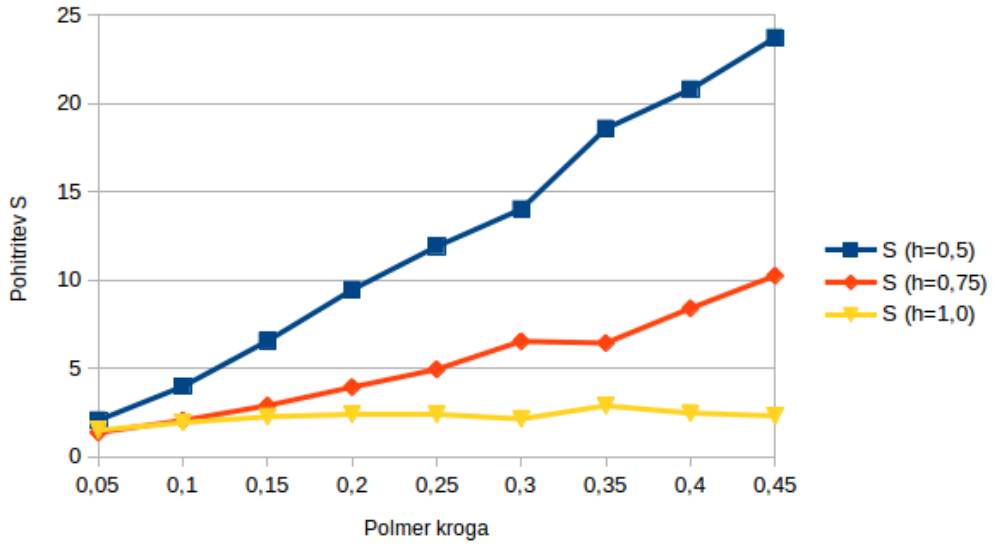
Slika 4.3: 50000 izvornih točk, porazdeljenih po robu kroga s polmerom 0,4.



Slika 4.4: Rezultati meritev izvajanja časa pri porazdelitvi izvornih točk po robu kroga v sekvenčni implementaciji IFGT.

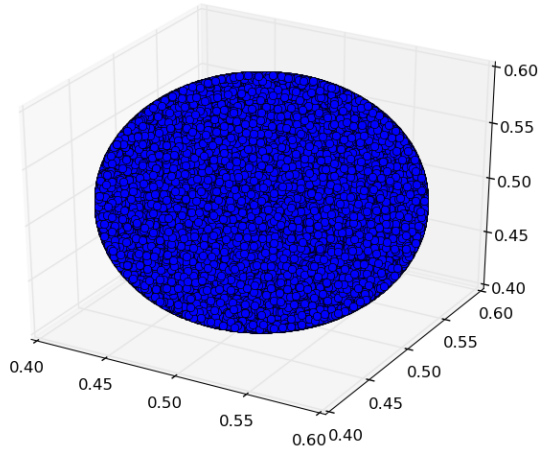


Slika 4.5: Rezultati meritev izvajanja časa pri porazdelitvi izvornih točk po robu kroga v paralelni implementaciji IFGT (OpenCL).



Slika 4.6: Pohitritve S pri porazdelitvi izvornih točk po robu kroga.

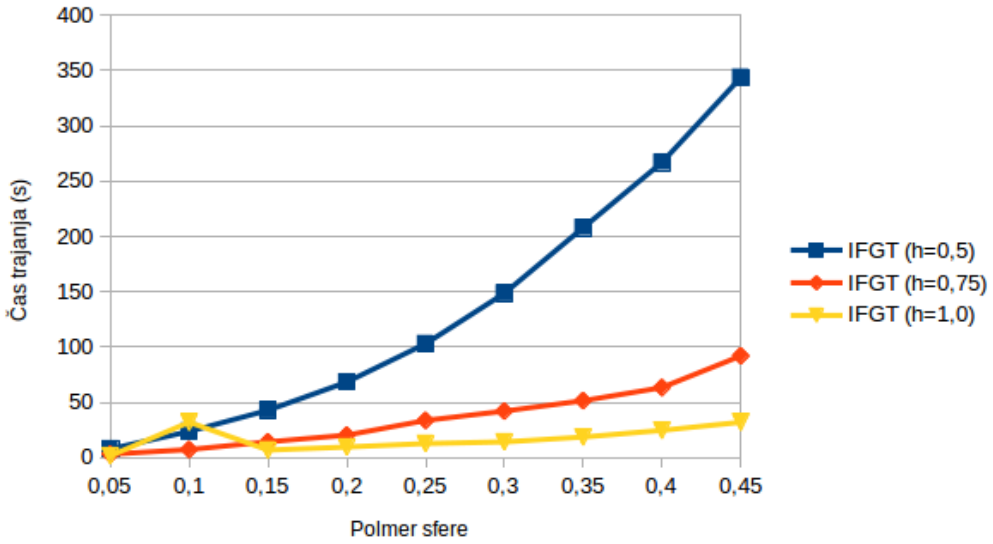
Porazdelitev izvornih točk po robu sfere Tako kot pri porazdelitvi po robu kroga smo tudi tu ustvarili 50000 izvornih točk, ki pa so sedaj bile porazdeljene po sferi s polmerom $r \in \{0{,}05, \ldots 0{,}45\}$. Primer porazdelitve pri $r = 0{,}1$ lahko vidite na sliki 4.7.



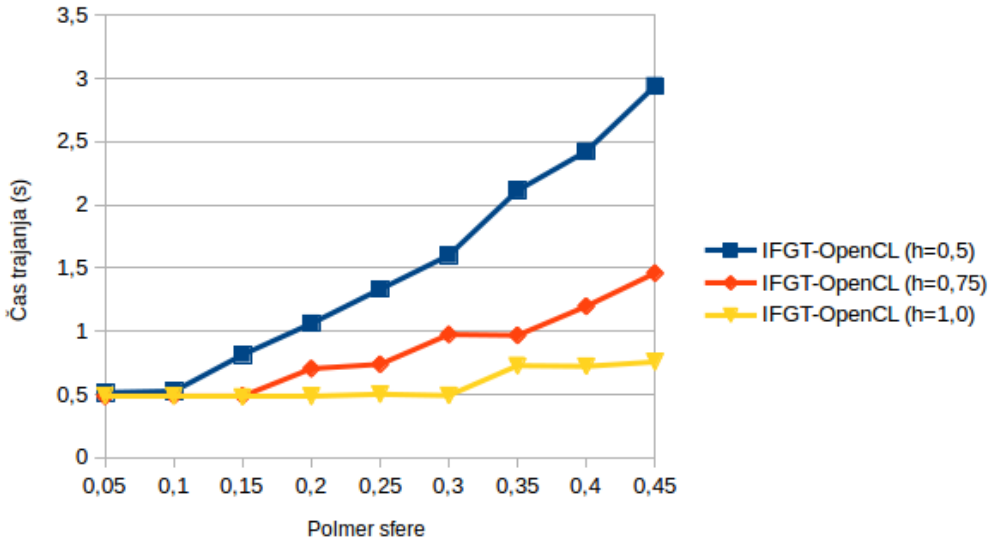
Slika 4.7: 50000 izvornih točk, porazdeljenih po sferi s polmerom 0,1.

Časi trajanja so zbrani na slikah 4.8 ter 4.9. Rezultata obeh slik sta kombinirana na sliki 4.10, v kateri je pohitritev S izračunana po formuli 4.3. Ugotovitev je ista kot pri porazdelitvi po robu kroga - z večanjem polmera sfere se veča tudi čas izvajanja. Za razliko od točk porazdeljenih po robu kroga pa se pri sferi jasno pokaže vpliv večje dimenzije $d = 3$, saj se sedaj tudi pri meritvah v OpenCL vidi trend naraščanja časa izvajanja skupaj z naraščanjem polmera sfere.

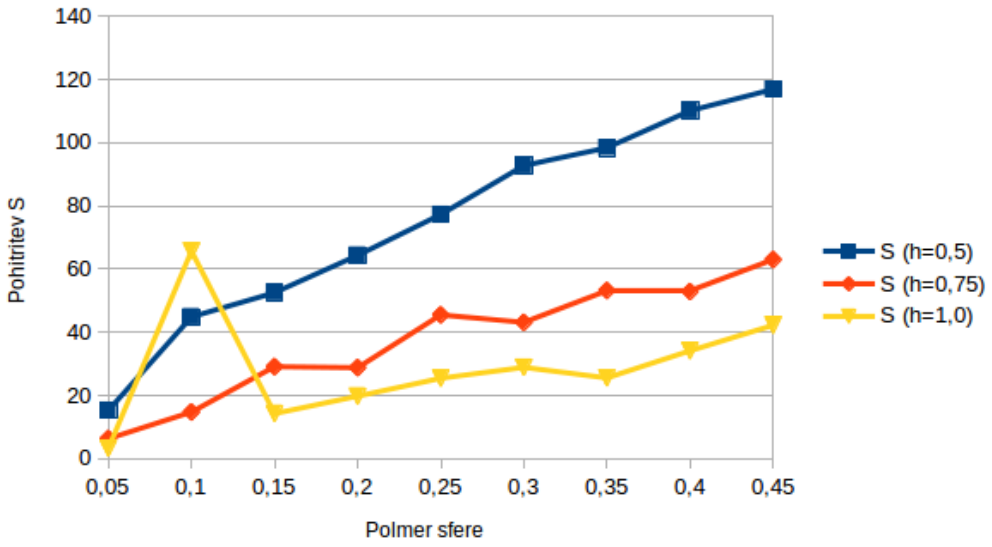
Porazdelitev izvornih točk okoli več centrov - spreminjanje števila centrov Zanima nas tudi čas izvajanja v kolikor točke zgostimo okoli več centrov z uporabo večdimenzionalne normalne porazdelitve. Pripravili smo 10 vhodnih datotek ter v njih zgostili točke okoli 1 do 10 centrov. Standardni



Slika 4.8: Rezultati meritev izvajanja časa pri porazdelitvi izvornih točk po sferi v sekvenčni implementaciji IFGT.

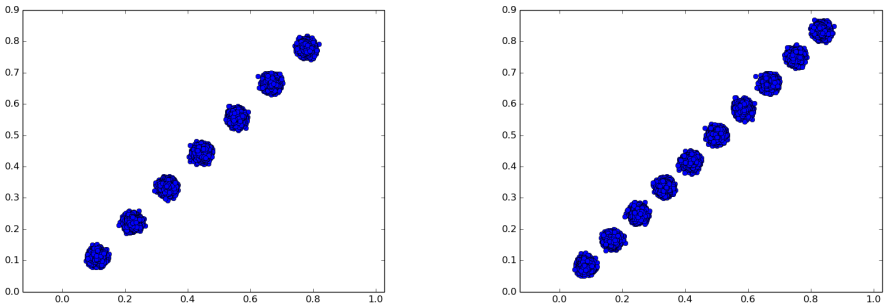


Slika 4.9: Rezultati meritev izvajanja časa pri porazdelitvi izvornih točk po sferi v paralelni implementaciji IFGT (OpenCL).



Slika 4.10: Pohitritve S pri porazdelitvi izvornih točk po robu sfere.

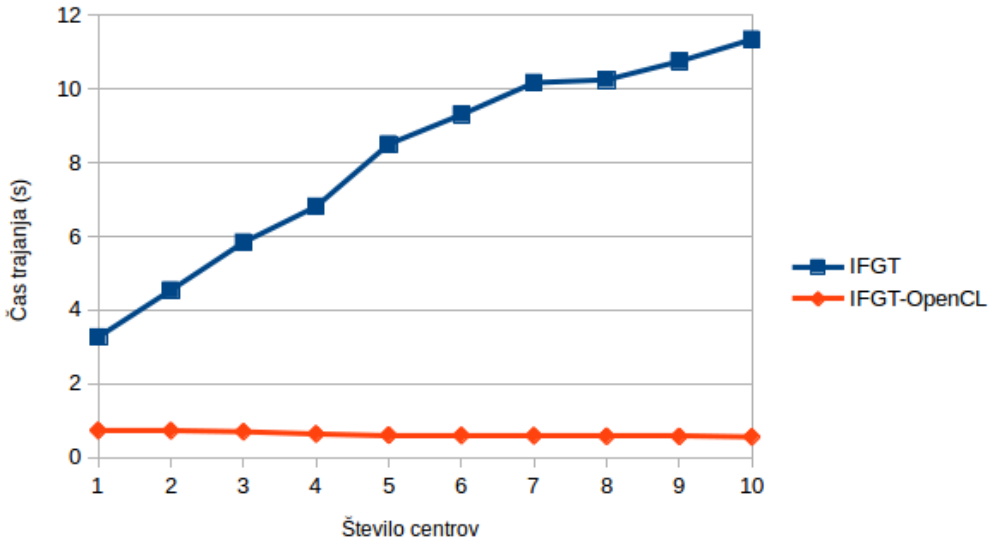
odklon $\sigma = 0{,}01$, število dimenzij $d = 2$ ter pasovna širina $h = 0{,}25$. Primera dobljenih porazdelitev točk lahko vidite na slikah 4.11.



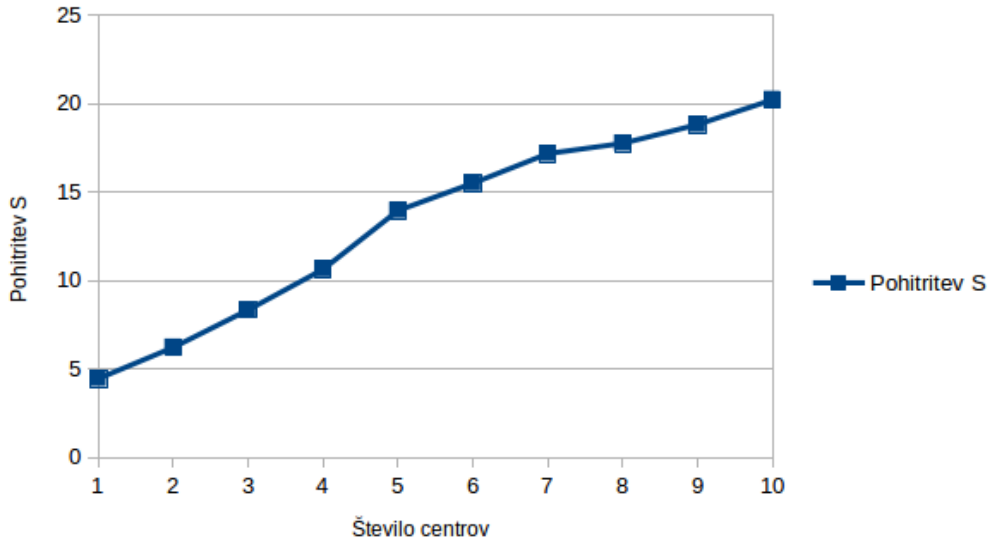
(a) 7 centrov (b) 10 centrov

Slika 4.11: Primera 50000 točk zgoščenih okoli več centrov s standardnim odklonom $\sigma = 0{,}01$ in uporabo večdimenzionalne normalne porazdelitve.

Rezultati meritev so zbrani na sliki 4.12, iz katere izračunane pohitritve S po enačbi 4.3 pa so predstavljene na sliki 4.13. Jasno se vidi, da čas trajanja



Slika 4.12: Rezultati meritev izvajanja časa pri spreminjanju števila centrov okoli katerih zgostimo 50000 točk.



Slika 4.13: Pohitritev pri spreminjanju števila centrov okoli katerih zgostimo 50000 točk.

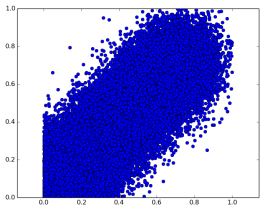
Število centrov	Odseki vrst gruč P_k
1	6
2	5 do 7
3	7 do 8
4	8 do 9
5	9
6	9 do 11
7	9 do 11
8	9 do 11
9	9 do 11
10	9 do 11

Tabela 4.4: Števila odseka vrst gruč v odvisnosti od števila centrov okoli katerih smo zgostili točke. Število gruč $K = 20$.

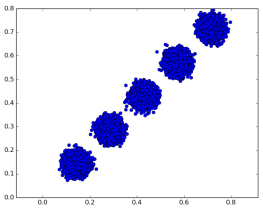
IFGT narašča s številom centrov. Razlog za to lahko vidimo v tabeli 4.4, kjer opazimo, da z naraščanjem števila centrov narašča tudi razpon števil odsekov vrst gruč P_k in s tem število koeficientov C_α^k, ki se izračunajo za vsako posamezno gručo. Zaradi majhnega števila dimenzij $d = 2$ ne pride do naraščanja časa izvajanja v implementaciji OpenCL, vendar pa bi pri višjih dimenzijah do njega vsekakor prišlo.

Porazdelitev izvornih točk okoli več centrov - spreminjanje standardnega odklona Poizkusimo sedaj še s spreminjanjem standardnega odklona σ. Naj je 50000 točk zgoščenih okoli 5 centrov ter naj je pasovna širina $h = 0{,}5$. Primere dobljenih porazdelitev točk lahko vidimo na slikah 4.14.

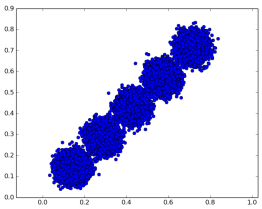
Slika 4.15 prikazuje rezultate meritev. Iz nje so izračunane pohitritve S po enačbi 4.3, ki so predstavljene na sliki 4.16. Z višanjem standardnega odklona se viša tudi čas izvajanja IFGT. Razlogi za to so podobni kot pri prejšnjem primeru, ko smo spreminjali število centrov - višji standardni odklon povzroči višanje števil odreza vrst gruč P_k.



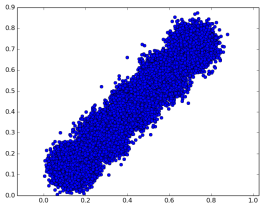
(a) $\sigma = 0{,}1$



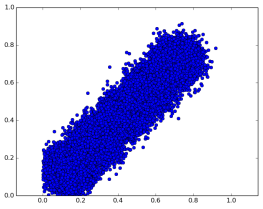
(b) $\sigma = 0{,}02$



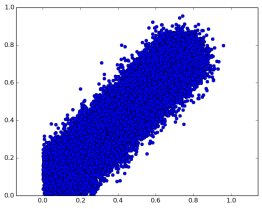
(c) $\sigma = 0{,}03$



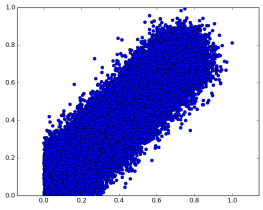
(d) $\sigma = 0{,}04$



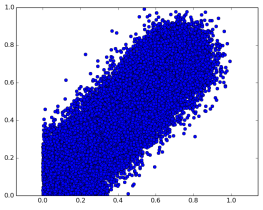
(e) $\sigma = 0{,}05$



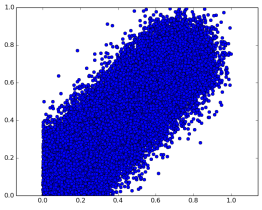
(f) $\sigma = 0{,}06$



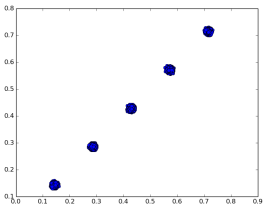
(g) $\sigma = 0{,}07$



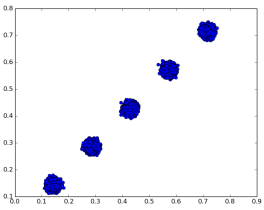
(h) $\sigma = 0{,}08$



(i) $\sigma = 0{,}09$

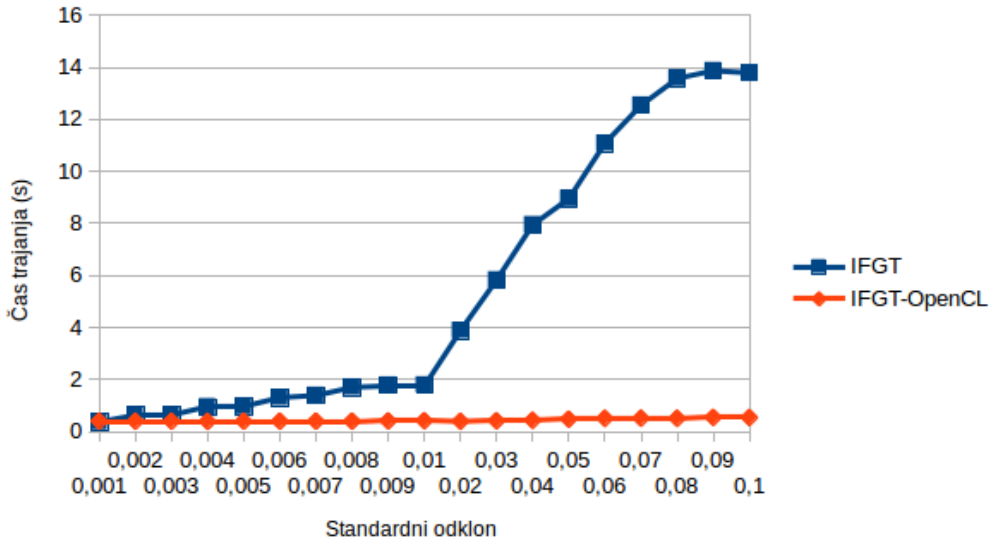


(j) $\sigma = 0{,}004$

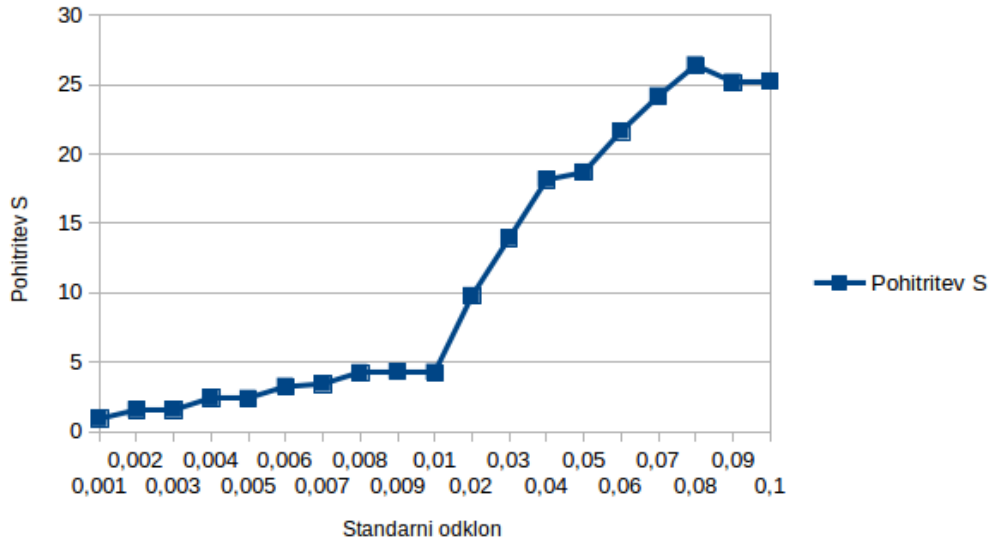


(k) $\sigma = 0{,}009$

Slika 4.14: Primeri 50000 točk zgoščenih okoli 5 centrov z različnim standardnim odklonom σ in uporabo večdimenzionalne normalne porazdelitve.



Slika 4.15: Rezultati meritev izvajanja časa pri spreminjanju standardnega odklona σ zgostitve 50000 točk okoli 5 centrov.



Slika 4.16: Pohitritev pri spreminjanju standardnega odklona σ zgostitve 50000 točk okoli 5 centrov.

Standardni odklon σ	Odseki vrst gruč P_k
0,001	4
0,005	6
0,01	7 do 8
0,05	16
0,1	17 do 21

Tabela 4.5: Števila odseka vrst gruč v odvisnosti od števila centrov okoli katerih smo zgostili točke. Število gruč $K = 5$.

Z višanjem števila dimenzij d bi se videl tudi vpliv na čas izvajanja implementacij v OpenCL.

4.3.3 Uporaba števil v enojni natančnosti v ščepcu OpenCL

V poglavju 4.1.3.3 smo navedli, da v naših ščepcih implementacij DGT ter IFGT uporabljamo števila v plavajoči vejici dvojne natančnosti. V specifikaciji OpenCL [8] piše, da je implementacija števil dvojne natančnosti le opcijska, zaradi česar raziščimo tudi možnost uporabe števil v plavajoči vejici enojne natančnosti.

Zanima nas, ali se ob uporabi števil v enojni natančnosti naše implementacije pohitrijo glede na implementacije, ki uporabljajo števila v dvojni natančnosti, ter ali s tem izgubimo kaj na natančnosti rezultata.

Test natančnosti Pripravili smo 6 vhodnih datotek (dim = 2, h = 1,0, število točk $N = M = 10000$, uteži so enake $1/N$), ki se razlikujejo v mejah napake $\epsilon \in \{10^{-2},10^{-3},10^{-4},10^{-5},10^{-6},10^{-7}\}$. Te vhodne datoteke bomo uporabili v testu implementacij IFGT ter DGT, ki v ščepcih uporabljajo števila v enojni natančnosti (*float*).
Rezultati testa natančnosti so zbrani v tabeli 4.6. Primerjali smo izhodno datoteko implementacije DGT s ščepcem, ki uporablja dvojno natančnost, z

ϵ	(float) DGT	(float) IFGT
10^{-2}	prestal	prestal
10^{-3}	prestal	prestal
10^{-4}	prestal	prestal
10^{-5}	prestal	prestal
10^{-6}	$\text{KvantNapaka}_{\text{relativna}} = 0{,}9722 > 0$	$\text{KvantNapaka}_{\text{relativna}} = 0{,}6778 > 0$
10^{-7}	$\text{KvantNapaka}_{\text{relativna}} = 0{,}9625 > 0$	$\text{KvantNapaka}_{\text{relativna}} = 0{,}5275 > 0$

Tabela 4.6: Test natančnosti implementacije OpenCL ščepcev IFGT ter DGT, ki uporabljajo števila v plavajoči vejici enojne natančnosti.

izhodnimi datotekami DGT in IFGT v OpenCL s ščepci, ki so uporabljali števila v enojni natančnosti. Beseda prestal pomeni, da so bila odstopanja vrednosti vseh ponornih točk med implementacijami manjša od ϵ. Metrika $\text{KvantNapaka}_{\text{relativna}}$ je bila predstavljena v poglavju 4.3.1.

Natančnost implementacije je padla pod dovoljeno mejo pri $\epsilon = 10^{-6}$. Takrat je relativna kvantitativna napaka iz enačbe 4.1 postala večja od 0 - torej ko so začele obstajati ponorne točke, v katerih so bila odstopanja vrednosti med implementacijami večja od ϵ. V enačbi 4.1 smo omenili še metriko največje absolutne napake. Največja absolutna napaka $\text{napaka}_{\text{max}}$ implementacije IFGT in DGT s ščepci, ki so uporabljali števila v enojni natančnosti, pa je bila malo večja od 10^{-6}.

Test hitrosti Ugotovili smo, da lahko v naših implementacijah uporabljamo ščepce, ki uporabljajo števila v plavajoči vejici v enojni natančnosti, do meje natančnosti $\epsilon = 10^{-6}$. Zanima nas, kakšen je čas trajanja delovanja takšnih implementacij v primerjavi s tistimi, ki uporabljajo števila v dvojni natančnosti. Izvedli smo test hitrosti ($\text{dim} = 1$, $h = 1.0$, $\epsilon = 10^{-4}$), kjer smo spreminjali število izvornih ter ponornih točk $N = M$.
Rezultati testov trajanja delovanja so zbrani v tabeli 4.7. Iz podatkov lahko razberemo, da so implementacije, ki uporabljajo števila v enojni natančnosti, precej hitrejše od tistih, ki uporabljajo števila v dvojni natančnosti. V kolikor

N	DGT (s)	DGT (s)	IFGT (s)	IFGT (s)
Podatkovni tip	double	float	double	float
50000	0,72	0,58	0,70	0,72
75000	1,06	0,73	0,94	0,82
100000	1,68	1,12	0,96	0,89
125000	2,23	1,18	1,12	0,91
150000	2,98	1,14	1,21	0,96
175000	3,74	1,18	1,48	1,05
200000	4,72	2,58	1,46	1,16

Tabela 4.7: Časi trajanja delovanja ščepcev IFGT ter DGT.

je želena meja napake ϵ dovolj majhna (večja od 10^{-6}), je uporaba števil enojne natančnosti v ščepcih OpenCL smiselna izbira.

Poglavje 5

Računanje Rényijeve entropije

Eden od primerov uporabe algoritma IFGT je pohitritev izračuna zvezne različice Rényijeve kvadratne entropije. V tem poglavju bomo predstavili Rényijevo kvadratno entropijo, nato bomo opisalo prilagojeno različico implementacije IFGT v OpenCL (predstavljene v poglavju 4) namenjeno izračunu zvezne Rényijeve kvadratne entropije ter jo tudi testirali.

5.1 Rényijeva kvadratna entropija

Za lažje razumevanje predstavimo najprej diskretno različico Rényijeve kvadratne entropije, nato pa se posvetimo še zvezni različici le-te.

Rényijeva entropija označuje najbolj splošno parametrično družino mer količine informacije, ki še ohranja aditivnost neodvisnih dogodkov[1], ter je združljiva z verjetnostnimi aksiomi (povzetimi po [16, str. 528]):

- naj je Ω je množica vseh dogodkov, verjetnost $P(\Omega) = 1$,
- verjetnost nekega dogodka $P(X) > 0$ za $X \in \Omega$,
- naj so $X_1, \ldots, X_k$ paroma nezdružljivi dogodki, tedaj je $P(X_1 \cup \ldots \cup X_k) = \sum_{i=1}^{k} P(X_i)$.

[1] Aditivnost neodvisnih dogodkov nam pravi, da za neodvisna dogodka X_1 in X_2 velja, da je $I(X_1X_2) = I(X_1) + I(X_2)$ za neko mero količine informacije I.

Diskretna Rényijeva α-entropija se izračuna s formulo [14]

$$H_\alpha(X) = \frac{1}{1-\alpha} \log(V_\alpha(x)) \,, \tag{5.1}$$

kjer je X diskretno porazdeljena slučajna spremenljivka, p_k za $k = 1 \ldots N$ pa verjetnosti vrednosti X. $V_\alpha(x) = \sum_{k=1}^{N} p_k^\alpha$ pravimo *informacijski potencial* (ang. Information Potential, IP). Shannonova entropija[2] je poseben primer Rényijeve entropije, in sicer ko $\lim \alpha \to 1$. Drug poseben pogosto uporabljan primer Rényijeve entropije je Rényijeva kvadratna entropija, in sicer pri $\alpha = 2$. Izračuna se jo s pomočjo enačbe

$$H_2(X) = -\log(\sum_{k=1}^{N} p_k^2) \,. \tag{5.2}$$

Poznamo tudi zvezno različico Rényijeve entropije. Zapišemo jo kot

$$H_\alpha(Y) = \frac{1}{1-\alpha} \log \int_{-\infty}^{\infty} p^\alpha(y) \, \mathrm{d}y \,, \tag{5.3}$$

kjer je Y zvezno porazdeljena slučajna spremenljivka, funkcija $p(y)$ pa njena gostota verjetnosti. Pri $\alpha = 2$ dobimo zvezno različico Rényijeve kvadratne entropije, ki jo bomo aproksimirali z našo implementacijo IFGT v OpenCL. Njen zapis je

$$H_2(Y) = -\log \int_{-\infty}^{\infty} p^2(y) \, \mathrm{d}y \,. \tag{5.4}$$

5.1.1 Aproksimacija zvezne Rényijeve kvadratne entropije

Zvezno različico Rényijeve kvadratne entropije lahko aproksimiramo z uporabo Parzenove aproksimacije gostote (ang. Parzen Density Estimation, PDE) [6, 1]. Ideja le-te je aplikacija nekega jedra (na primer Gaussovega) nad podatki ter normaliziran seštevek tako dobljenih rezultatov. Izračunamo jo lahko s pomočjo formule

[2] Shannonovo entropijo se zapiše kot $H(X) = -\sum_{k=1}^{N} p_k \log(p_k)$, kjer je p_k verjetnost dogodka X_k.

$$\hat{p_\chi}(y) = \frac{1}{N\delta} \sum_{i=1}^{N} K(\frac{\chi - y_i}{\delta}) \,, \tag{5.5}$$

kjer znak χ pomeni velikost jedra, K pa označuje samo jedro. Uporabimo **Gaussovo jedro** z velikostjo $\delta\sqrt{(2)}$. Tako dobimo aproksimacijo zvezne Rényijeve kvadratne entropije

$$\hat{H}_2(Y) = -\log \int_{-\infty}^{\infty} (\frac{1}{N} \sum_{i=1}^{N} G_\delta (y - y_i)^2) \, \mathrm{d}y \,, \tag{5.6}$$

katere za nas uporaben približek je

$$\hat{H}_2(Y) = -\log(V_2(y)) = -\log(\frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} G_{\delta\sqrt{2}}(y_j - y_i)) \tag{5.7}$$

za informacijski potencial $V_2(y)$, ki se izračuna preko formule

$$V_2(y) = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1}^{N} G_{\delta\sqrt{2}}(y_j - y_i) \,. \tag{5.8}$$

5.2 Izračun Rényijeve kvadratne entropije s pomočjo IFGT v OpenCL

Informacijsko teoretično učenje (ang. Information Theoretic Learning, ITL) je zbirka metod, s pomočjo katerih ocenimo entropijo neposredno iz podatkov, brez uporabe statističnih pojmov kot sta varianca in kovarianca. Metode za aproksimativen izračun Rényijeve kvadratne entropije podatkov brez eksplicitne ocene funkcije gostote verjetnosti spadajo v samo jedro ITL [6] in se uporabljajo predvsem na področju strojnega učenja. Izračun informacijskega potenciala po formuli 5.8 ima časovno kompleksnost $O(N^2)$, ki jo želimo zmanjšati. To nam omogoča uporaba IFGT.

Imamo adaptiven sistem, ki deluje po principu *FIR*[3]. Ta sistem lahko

[3]Kratica FIR (ang. Finite Impulse Response) označuje način odziva sistema na impulze iz zunanjega sveta, pri katerem je čas trajanja odziva končen - oziroma je po nekem končnem času enak 0.

opišemo s k-dimenzionalnim vektorjem uteži w_k, v katerega iz zunanjega sveta zajamemo vektor k vhodnih podatkov u_k. Želimo, da se sistem odzove s stanjem d_k, napako med dejanskim ter želenim stanjem sistema pa označimo kot e_k. Velja $e_k = d_k - w_k^T u_k$.

Mi računamo entropijo množice vektorjev e_k za $k = 1 \dots N$. Uporabimo že predstavljeno notacijo DGT. Na vhod algoritma uvedemo:

- dimenzionalnost d,
- N število ponornih in izvornih točk $e_k \in R_d^+$ (le-te sovpdajo),
- uteži izvornih točk, ki so na začetku nastavljene na 1,
- pasovno širino $h = \sqrt{(}\delta\sqrt{2})$, kjer se δ izbere z uporabo Silvermanovega pravila (omenjenega v poglavju 4.3.1).

Najprej izračunamo informacijski potencial po enačbi 5.8. Za pohitritev izračuna uporabimo IFGT na zgoraj navedenih vhodnih podatkih, nato seštejemo vrednosti vseh ponornih točk ter seštevek zmnožimo z $\frac{1}{N^2}$. Sedaj uporabimo prvi del enačbe 5.7, preko katerega dokončamo izračun aproksimacije Rényijeve kvadratne entropije $\hat{H}_2(X)$.

5.2.1 Implementacija

Za implementacijo postopka hitrega izračuna Rényijeve kvadratne entropije, kot je bil opisan v poglavju 5.2, moramo predelati obstoječo implementacijo IFGT v OpenCL.

- Iz vhodne datoteke preberemo le N izvornih točk. V kolikor $N \neq M$, se izvajanje prekine.
- V kolikor uteži vseh izvornih točk niso enake 1, prekinemo izvajanje.
- Prebrane izvorne točke znotraj programa zapišemo tudi v tabelo ponornih točk.

- Po izračunu IFGT izračunamo vsoto vrednosti vseh ponornih točk, jo zmnožimo z $1/N^2$, vzamemo naravni logaritem te vrednosti ter jo pomnožimo z (-1).

- Dobljeno vrednost Rényijeve kvadratne entropije izpišemo tako na zaslon kot tudi v izhodno datoteko.

Vse ostale podrobnosti implementacije IFGT v OpenCL iz poglavja 4 ostanejo iste, vključno s ščepci, ki se izvajajo na grafični kartici. Poleg različice izračuna Rényijeve kvadratne entropije, ki uporablja IFGT v OpenCL, implementiramo tudi različico, ki uporablja DGT v OpenCL.

V namen primerjanja obeh implementacij izračuna Rényijeve kvadratne entropije smo ustvarili 7 enakih testnih datotek (z $N = 50000$ enakimi izvornimi točkami, $h = 1{,}0$ ter $d = 1$), ki so se razlikovale le v mejah napake $\epsilon \in \{10^{-3}, \ldots, 10^{-9}\}$. Zanimala nas je absolutna napaka izračunane entropije med implementacijo, ki uporablja DGT, ter implementacijo, ki uporablja IFGT.

ϵ	DGT	IFGT	absolutna napaka	relativna napaka
10^{-3}	0,1495917168	0,1495943558	0,00000264	0,000017648
10^{-4}		0,149591889	0,000000172	0,000011497
10^{-5}		0,1495917278	0,000000011	0,000000073
10^{-6}		0,1495917174	0,0000000006	0
10^{-7}		0,1495917169	0,0000000001	0
10^{-8}		0,1495917168	0	0
10^{-9}		0,1495917168	0	0

Tabela 5.1: Absolutna napaka implementacije rényijeve kvadratne entropije s pomočjo IFGT

Rezultati so zbrani v tabeli 5.1. Iz njih je razvidno, da je bila absolutna napaka manjša od ϵ - kar je bilo tudi za pričakovati, saj IFGT že sam po sebi garantira določeno absolutno napako, operacije seštevanja vrednosti ponornih točk ter množenje te vrednosti z $1/N^2$ pa so elementarne operacije v plavajoči

vejici. Njihovo približno vrednost izračunamo preko funkcije $\phi(a \operatorname{op} b) = (1 + \omega)(a \operatorname{op} b)$, kjer je $(a \operatorname{op} b)$ želena natančna operacija v plavajoči vejici ter $\omega = 1{,}11 \times 10^{-16}$ napaka dvojne natančnosti [7]. Pri nas izvršimo N seštevanj in dve deljenji. Na ta način dobljena napaka bo zagotovo manjša od najmanjše vrednosti ϵ.

Poglavje 6

Sklepne ugotovitve

V okviru diplomskega dela smo uspešno implementirali metodo IFGT v OpenCL ter jo izkoristili za izračun Rényijeve kvadratne entropije. Testiranje implementacije je potrdilo pričakovanja glede učinkovitosti metode iz literature. Narejena implementacija se je izkazala za veliko hitrejšo od implementacije IFGT, ki je tekla le na procesorju. V večini primerov se je izkazala tudi hitrejša od implementacije DGT, ki je tekla na procesorju. V nekaterih primerih je bila hitrejša celo od DGT, ki je tekla na grafični kartici v OpenCL.

Izdelana implementacija IFGT v OpenCL nam za določene narave vhodnih podatkov omogoča zelo hiter izračun približka DGT - predvsem pri velikih pasovnih širinah h ter pri nizkih dimenzijah d. Pri majhnih pasovnih širinah ter pri višjih dimenzijah se implementacija izkaže za neuporabno, vendar pa te slabosti izhajajo iz samih omejitev metode IFGT. V slednjih primerih je nujna uporaba metod kot so DIFGT, DFGT ter SIFGT, ki so precej boljše od IFGT, vendar pa zaradi odsotnosti podpore za rekurzijo v sedanjih različicah standarda OpenCL še niso najbolj primerne za implementacijo na grafični kartici. Podprtje uporabe rekurzije v OpenCL bi omogočilo veliko lažjo implementacijo teh metod na grafični kartici, brez ročne implementacije sklada ter imitacije rekurzije s pomočjo zank. To bi omogočilo še hitrejši izračun približka DGT, kot smo ga dosegli v tej diplomski nalogi.

Literatura

[1] Suyash P. Awate. Parzen-window density estimation. dostopno na naslovu Univerze v Utahu `https://www.cs.utah.edu/~suyash/Dissertation_html/node11.html` (dostopano dne 12.12.2015), 2007. Spletna stran.

[2] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Waltham : Morgan Kaufmann ; Amsterdam [etc.] : Elsevier, cop. 2012, 2013.

[3] Teofilo F Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[4] Leslie Greengard and John Strain. The fast gauss transform. *SIAM Journal on Scientific and Statistical Computing*, 12(1):79–94, 1991.

[5] Leslie Greengard and Xiaobai Sun. A new version of the fast gauss transform. *Documenta Mathematica*, ICM 1998(III):575–584, 1998.

[6] Seungju Han, Sudhir Rao, and Jose Principe. Estimating the information potential with the fast gauss transform. In *Independent Component Analysis and Blind Signal Separation*, pages 82–89. Springer, 2006.

[7] Prof. W. Kahan. Ieee 754 standard for binary floating-point arithmetic. dostopno na naslovu Univerze Kalifornije `http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps` (dostopano dne 12.12.2015), 1996. Zapiski predavanj.

[8] Khronos OpenCL Working Group. *The OpenCL Specification (različica 1.2*, 2012. na voljo na naslovu `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf` (dostopano dne 28.10.2015).

[9] Khronos OpenCL Working Group. *The OpenCL C Specification (različica 2.0*, 2015. na voljo na naslovu `https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf` (dostopano dne 28.10.2015).

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

[11] Dongryeol Lee, Andrew W Moore, and Alexander G Gray. Dual-tree fast gauss transforms. In *Advances in Neural Information Processing Systems*, pages 747–754, 2005.

[12] Peter Luschny. Approximation formulas for the factorial function. dostopno na naslovu `http://www.luschny.de/math/factorial/approx/SimpleCases.html` (dostopano dne 15.10.2015), 2015. Spletna stran z zbirko algoritmov za približen izračun vrednosti faktoriele n! v plavajoči vejici.

[13] Gordon E Moore. Lithography and the future of moore's law. In *SPIE's 1995 Symposium on Microlithography*, pages 2–17. International Society for Optics and Photonics, 1995.

[14] Jose C. Principe. Renyi's entropy. dostopno na naslovu Univerze Florida `http://www.cnel.ufl.edu/courses/EEL6814/renyis_entropy.pdf` (dostopano dne 23.11.2015), 2009. Zapiski predavanj.

[15] Vikas Chandrakant Raykar, Changjaing Yang, Ramani Duraiswami, and Nail Gumerov. Fast computation of sums of gaussians in high dimensions (tehnično poročilo). Technical report, Perceptual Interfaces and Reality Laboratory, Department of Computer Science and Institute for

Advanced Computer Studies, University of Maryland, November 2005. CS-TR-4767.

[16] Alfred Rényi. Some fundamental questions of information theory. *Selected Papers of Alfred Renyi*, 2(174):526–552, 1976.

[17] Bernard W Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

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

[19] Changjiang Yang in Ramani Duraiswami Vikas Chandrakant Raykar. *Improved fast Gauss transform.* Department of Computer Science, University of Maryland, CollegePark, MD 20783, 3 edition, Oktober 2006. Navodila za uporabo implementacije IFGT dostopne preko `http://www.umiacs.umd.edu/labs/cvl/pirl/vikas/Software/IFGT/IFGT_code.htm`.

[20] Daniel Wissel. Die diskrete gauß-transformation – schnelle approximationsverfahren und anwendunged in hohen dimensionen (diplomsko delo). Master's thesis, Mathematisch-Naturwissenschaftliche Fakult´at der Rheinischen Friedrich-Wilhelms Universit´at Bonn, April 2008.

[21] C Yang, R Duraiswami, and NA Gumerov. Improved fast gauss transform, dept. of computer science, university of maryland. Technical report, CS-TR-4495, 2003.