Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Sodja Anže

Evalvacija polinomov na podatkovno-pretokovnih računalnikih

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2017

Zahvala

Zahvaljujem se družbi Maxeler, da mi je omogočila razvoj in testiranje, ter mentorju doc. dr. Juriju Miheliču za podporo in nasvete pri izdelavi tega dela.

Sodja Anže, 2017

Delo posvečam svoji družini in prijateljem.

"The only reason for time is so that everything doesn't happen at once."

— Albert Einstein

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DFE	data-flow engine	podatkovno-pretokovna enota
CFE	central processing unit	centralna procesna enota
FPGA	field programmable gate array	programabilno vezje
LMEM	large memory	veliki pomnilnik
FMEM	fast memory	hitri pomnilnik
DFT	discrete Fourier transform	diskretna Fourierova transformacija
FFT	fast Fourier transform	hitra Fourierova transformacija
DCT	discrete cosine transform	diskretna kosinusna transformacija
DSP	digital signal processor	digitalni signalni procesor
LUT	look-up table	preslikovalna tabela
BRAM	block ram	blokovni pomnilnik
FF	flip-flop	bistabilni multivibrator, tudi flip-flop

Povzetek

Naslov: Evalvacija polinomov na podatkovno-pretokovnih računalnikih

V magistrskem delu smo implementirali algoritme za evalvacijo polinomov na podatkovno-pretokovni arhitekturi. Čeprav je evalvacija polinomov enostaven problem za današnje centralne procesne enote, pa z večjim številom točk tudi ta postane počasna. Tako smo implementirali algoritme za evalvacijo redkih in gostih polinomov v eni in več točkah na podatkovno-pretokovnem računalniku družbe Maxeler. Naše algoritme smo eksperimentalno preizkusili na realnih in kompleksnih polinomih. Dosegli smo do dvajsetkratne pospešitve za goste polinome v več točkah in do sedemdesetkratne pospešitve za redke polinome v več točkah. Poleg tega smo naše algoritme prilagodili tudi za evalvacijo podproblema gručenja točk in diskretne Fourierove transformacije. Vse rezultate smo analizirali in grafično predstavili.

Ključne besede

podatkovno-pretokovna arhitektura, evalvacija polinomov, algoritmi, Maxeler

Abstract

Title: Polynomial Evaluation on Data-Flow Computers

In this master thesis we implemented algorithms for polynomial evaluation on data-flow architecture. Polynomial evaluation is relatively simple problem for today's central processing units. However with an increasing number of points in which we evaluate polynomial, time of evaluation can become a problem. We implemented algorithms for evaluation of sparse and dense polynomials on Maxeler data-flow computers. We tested our algorithms on real polynomials as well as on complex polynomials. We have achieved up to 20-fold speedup for dense and up to 70-fold speedup for sparse polynomials. Additionally, we customised our algorithms for evaluation of subproblem of point clustering and also for evaluation of Discrete Fourier transform. We analysed our results and presented them graphically.

Keywords

data-flow architecture, polynomial evaluation, algorithms, Maxeler

Poglavje 1

Uvod

1.1 Motivacija

Današnji računalniki izvedejo na milijarde operacij na sekundo. Vendar obstajajo problemi, za katere niso dovolj hitri. Pogosto se ob takih problemih opremo na paralelizacijo. Paralelizacijo lahko izvajamo kar na sodobnih centralnih procesnih enotah, saj imajo te vedno več jeder. Z uporabo aplikacijskih programskih vmesnikov, kot sta CUDA ali OpenCL, lahko algoritem prenesemo tudi na grafično procesno enoto. Ta je zaradi velikega števila jeder primerna za paralelizacijo. Lahko pa implementiramo algoritme tudi na sprecializirani arhitekturi, kot je podatkovno-pretokovni računalnik Maxeler.

Evalvacija polinomov ne spada med težke probleme za klasične računalnike s centralno procesno enoto. Vendar pa z rastjo polinomov in/ali številom točk, v katerih evalviramo polinom, tudi ta postane počasna. V magistrski nalogi bomo predstavili implementacijo problema evalvacije polinomov na alternativni arhitekturi, to je na podatkovno-pretokovnem računalniku družbe Maxeler [1]. Ta omogoča paralelizacijo operacij, zaradi česar se lahko algoritmi izvedejo hitreje [2]. Dodatno pa so lahko ti računalniki tudi energijsko učinkovitejši, kar omogoča tudi prihranke pri porabi energije [3]. Več o samem podatkovno-pretokovnem računalniku sledi v razdelku 2.

Opisane algoritme, implementirane na podatkovno-pretokovnem raču-

nalniku Maxeler, smo primerjali s primerljivo implementacijo na centralni procesni enoti, rezultate in opažanja pa smo grafično predstavili, opisali in kritično ovrednotili.

1.2 Polinomi

V magistrski nalogi bomo reševali problem evalvacije polinomov. Polinomi so pogosto eno izmed orodij za modeliranje problemov. Preprost primer je modeliranje prostega pada brez upora v fiziki, ki ga lahko zapišemo kot [4]:

$$y(t) = -\frac{1}{2}gt^2 + v_0t + y_0,$$

kjer je g gravitacijski pospešek, t predstavlja čas, v_0 začetno hitrost in y_0 začetno višino. Drug tak primer je grafično modeliranje v računalništvu z Bézierovimi krivuljami [5]. S polinomi pa se srečujemo tudi drugje, na primer v ekonomiji, kjer predstavljajo stroškovne funkcije [6].

Polinom definiramo kot izraz iz spremenljivk in koeficientov, ki povezujejo samo operacije množenja in seštevanja ter potence števil s celimi nenegativnimi eksponenti [7]. Ponavadi ga zapišemo kot vsoto členov in ga predstavimo s splošno enačbo:

$$p(x) = \sum_{i=0}^{d} k_i x^i,$$

kjer je d stopnja polinoma.

V razširjenem zapisu polinoma člene z ničelnimi koeficienti ponavadi izpustimo, kot je prikazano v spodnjem primeru:

$$p(x) = 4 + x + x^3 + \frac{1}{3}x^7.$$

1.3 Predstavitev polinomov

Polinome lahko glede na predstavitev in obliko razdelimo na goste in redke polinome.

Goste polinome bomo zapisali kot [8]:

$$p(x) = \sum_{i=0}^{d} k_i x^i. \tag{1.1}$$

Tak zapis bomo imenovali *predstavitev gostih polinomov*. V predstavitvi gostih polinomov vedno zapišemo vse člene polinoma, eksponenti si sledijo eden za drugim v naraščajočem vrstnem redu, dolžina polinoma pa je za ena večja od največjega eksponenta oziroma stopnje. Stopnja je v enačbi označena z d.

Predstavitev gostih polinomov v računalništvu običajno podamo kot seznam koeficientov. Zaradi enakomerno naraščajočih eksponentov te izpustimo. Zanjo obstajajo učinkoviti algoritmi za evalvacijo, kot je Hornerjev algoritem [9].

Če je število ničelnih koeficientov v polinomu veliko, govorimo o redkih polinomih. Za redke polinome lahko uporabimo naslednjo predstavitev [8]:

$$p(x) = \sum_{i=0}^{n} k_i x^{e_i}. \tag{1.2}$$

Tak zapis bomo imenovali *predstavitev redkih polinomov*. V predstavitvi redkih polinomov predstavimo $n + 1$ členov z neničelnim koeficientom. Člene z ničelnim koeficientom izpustimo.

Eksponenti med seboj niso odvisni, lahko so razvrščeni po velikosti ali ne. Dovolili bomo tudi polinome, kjer se eksponenti ponavljajo.

V računalništvu redke polinome predstavimo kot seznam parov koeficientov in eksponentov. Predstavitev redkih polinomov je primerna tudi za polinome več spremenljivk. V predstavitvi gostih polinomov število členov s številom spremenljivk namreč eksponentno raste, s tem pa tudi število ničelnih koeficientov [8].

1.4 Evalvacija polinoma

Če zapišemo preprost polinom, na primer:

$$p(x) = 5 + 3x^3 + x^4 + 20x^{15},$$

potem vidimo, da ga lahko razčlenimo na tri operacije:

- potenciranje spremenljivke x,
- množenje rezultata potenciranja s koeficientom,
- seštevanje rezultatov množenja.

Zaradi lastnosti komutativnosti in asociativnosti seštevanja in množenja so členi polinoma medseboj neodvisni, prav tako sta neodvisna vrstni red in način evalvacije členov. To je pomembno pri sami implementaciji na podatkovno-pretokovnem računalniku, saj nam pri implementaciji ni treba paziti, kateri člen najprej izračunamo, kar je dobra iztočnica za paralelizacijo. Zaradi teh lastnosti obstajajo učinkoviti sekvenčni algoritmi za evalvacijo polinomov. Tak algoritem je Hornerjev algoritem [9], ki izračuna polinom z optimalnim številom operacij [10]. Obstajajo pa tudi dobri algoritmi za paralelno evalvacijo, kot so opisani v [11].

Tako na primer naivna evalvacija gostega polinoma, podana v algoritmu 1, kjer zaporedno evalviramo člene polinoma, zahteva $3(n-1)$ operacij: $2(n-1)$ množenj in $(n-1)$ seštevanj. Z n smo označili dolžino polinoma. V algoritmu 1 so koeficienti podani kot seznam ks, d pa je stopnja polinoma.

Algoritem 1 Naivni algoritem evalvacije polinoma

$pow \leftarrow 1.0$
$result \leftarrow ks[0]$
for $i \in \{1, \ldots, n-1\}$ **do**
  $pow \leftarrow pow * x$
  $result \leftarrow result + ks[i] * pow$
end for

Medtem Hornerjev algoritem, podan v algoritmu 2, zahteva $2(n-1)$ operacij: $(n-1)$ množenj in $(n-1)$ seštevanj [9], pri čemer je n število členov polinoma. Algoritem je tako še vedno enakega asimptotičnega reda. Vendar pa lahko ob velikem številu operacij, na primer pri evalvaciji velikega števila točk, manjše število operacij pomeni veliko pohitritev.

Algoritem 2 Hornerjev algoritem

$result \leftarrow ks[d]$
for $i \in \{n-2, \ldots, 0\}$ **do**
  $result \leftarrow result * x + k[i]$
end for

Čeprav je razdelitev polinoma na operacije trivialna, pa je bila ta razdelitev v našem miselnem procesu razvoja implementacije osnova za implementirane algoritme.

1.5 Uporaba polinomov

Kot že omenjeno, so polinomi pogosto eno izmed orodij za modeliranje problemov. V razdelku 1.2 smo podali primer prostega pada.

Drug primer so Bézierove krivulje v grafičnem modeliranju. Spodaj je podan zapis kvadratne Bézierove krivulje [5]:

$$B(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2,$$

kjer $0 \leq t \leq 1$, P_0, P_1 in P_2 pa so podane kontrolne točke.

Omenili smo tudi stroškovne funkcije v ekonomiji. V [6] se rešuje problem optimizacije stroškov električnih generatorjev. Poskuša se minimizirati funkcijo:

$$C = \sum_{j \in J} F_j(P_j),$$
$$F_j(P_j) = a_j + b_j P_j + c_j P_j^2,$$

kjer je C celoten strošek, P_j proizvedena elektrika generatorja, J množica generatorjev, a_j, b_j in c_j pa so stroškovni koeficienti.

Kot polinom bi lahko obravnavali tudi računanje razdalje med točkami. Če želimo točke klasificirati v razrede, kot v primeru gručenja z metodo voditeljev (angl. k-means clustering) [12], potem lahko kot metriko za gručenje

uporabimo kvadrat evklidske razdalje, in sicer:

$$d(X,Y)^2 = (x_0 - y_0)^2 + (x_1 - y_1)^2 + \cdots + (x_n - y_n)^2.$$

Če eno izmed točk fiksiramo, dobimo poseben primer polinoma več spremenljivk.

Podobni primeri so diskretna Fourierova transformacija (krajše DFT) ter sorodni diskretna kosinusna transformacija in diskretna sinusna tranformacija. Čeprav ti problemi po definiciji niso pravi polinomi, pa so polinomom sorodni in jih lahko evalviramo na podoben način kot polinome. Spodaj je zapisan primer diskretne kosinusne transformacije [13]:

$$X_k = \sum_{n=0}^{N-1} x_n cos(k\frac{\pi}{N}(n + \frac{1}{2})),\ k = 0, ...,N - 1.$$

Več o diskretni Fourierovi transformaciji je napisano v razdelku 4.3.

1.6 Pregled vsebine

V razdelku 2 sta opisani podatkovno-pretokovna arhitektura na splošno in njena implementacija na računalnikih Maxeler. Opisana sta tudi način pisanja kode za podatkovno-pretokovni računalnik Maxeler in njeno izvajanje.

V razdelku 3 je nato opisana implementacija algoritmov na podatkovno-pretokovni enoti; najprej za evalvacijo gostih polinomov v razdelkih 3.1 in 3.2, nato algoritmi za evalvacijo redkih polinomov v razdelkih 3.3 in 3.4. Razširitev algoritmov s paralelizacijo pa je opisana v razdelku 3.5.

V razdelku 4 nato sledi razširitev algoritmov za računanje kompleksnih polinomov, opisani sta tudi implementaciji podproblema gručenja točk in diskretne Fourierove transformacije. V razdelku 5 pa sledijo še analiza rezultatov, kritično vrednotenje in zaključek.

Poglavje 2

Podatkovno-pretokovni računalnik

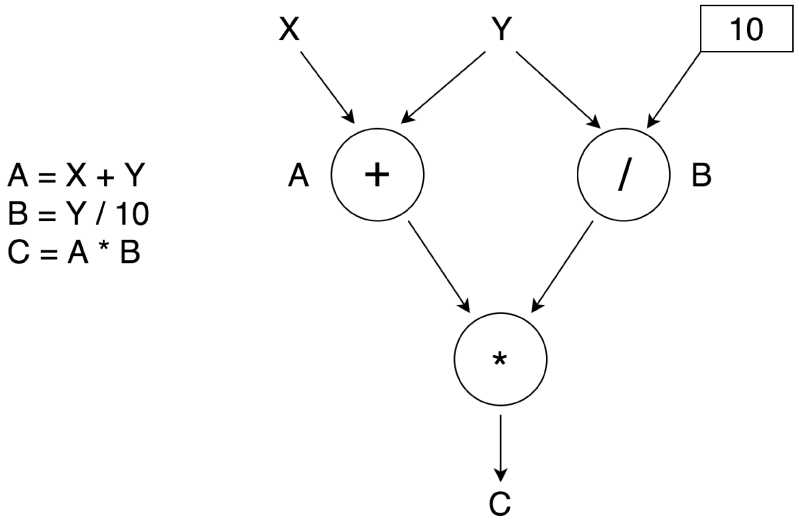
2.1 Podatkovno-pretokovna arhitektura

Algoritme za evalvacijo polinomov bomo v magistrski nalogi implementirali na podatkovno-pretokovni arhitekturi, ki se je razvila z namenom, da bi omogočila visoko stopnjo paralelizacije. Glavna kritika von Neumannove arhitekture je namreč bila, da uporablja programske števce in globalni pomnilnik, ki omejujeta paralelizacijo [14, 15].

Na podatkovno-pretokovni arhitekturi tečejo algoritmi, ki jih lahko predstavimo kot graf operacij, kjer so operacije predstavljene kot vozlišča, povezave pa predstavljajo tok podatkov. Podatki se tako pretakajo po grafu, vse operacije oziroma vozlišča s podatkom na vhodu pa se lahko izvedejo naenkrat.

Klasična predstavitev podatkovno-pretokovnega algoritma je graf izračuna matematičnega izraza, kot je prikazano na sliki 2.1. Nad vhodnima podatkoma X in Y se paralelno izračunata izraza A in B, nato pa se izračuna še C. Če imamo tok vhodnih podatkov X in Y, lahko vsa vozlišča paralelno izvajajo operacije, kot je prikazano v tabeli 2.1.

Podatkovno-pretokovni graf se nato preslika na čip, kar pomeni, da je



Slika 2.1: Primer izvajanja podatkovno-pretokovnega algoritma [14]

vsaka operacija fizično prisotna na čipu. Maxeler zato tako računanje označuje kot računanje v prostoru (angl. computing in space) [16].

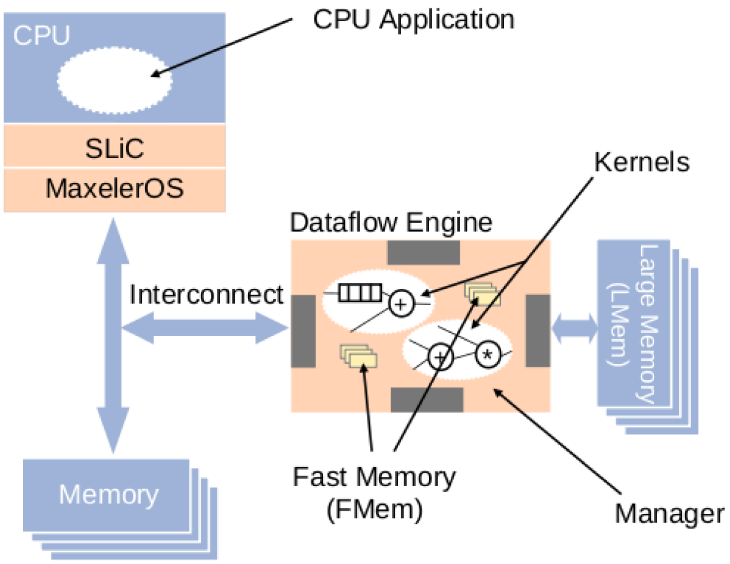
t	A_t	B_t	C_t
1	$X_1 + Y_1$	Y_1 / 10	-
2	$X_2 + Y_2$	Y_2 / 10	$A_1 * B_1$
3	$X_3 + Y_3$	Y_3 / 10	$A_2 * B_2$
4	-	-	$A_3 * B_3$

Tabela 2.1: Izvajanje operacij v vozliščih po času

2.2 Izvedba Maxeler

Naše algoritme bomo izvajali na podatkovno-pretokovnem računalniku podjetja Maxeler Technologies [1, 17]. Glavni del tega računalnika je Maxelerjeva podatkovno-pretokovna enota. Ta enota je pravzaprav razširitvena kartica, ki je z računalnikom povezana prek vmesnika PCIe.

Arhitektura je predstavljena na sliki 2.2. Razširitvena kartica je sestavljena iz vezja FPGA (angl. field-programmable gate array) in vsebuje dve



Slika 2.2: Arhitektura Maxelerjevega računalnika [18, 19]

vrsti pomnilnika - hitri pomnilnik (FMEM) direktno na čipu in veliki pomnilnik (LMEM) izven čipa. Hitri pomnilnik ima velikost nekaj megabajtov in dosega hitrosti pisanja in branja do 10 TB/s [18], zato je primeren za shranjevanje vmesnih rezultatov med računanjem, do katerih moramo pogosto dostopati. Veliki pomnilnik pa je večji, dosega velikost nekaj gigabajtov, a je zato počasnejši in primeren za shranjevanje večjih količin podatkov, do katerih med računanjem ne dostopamo pogosto ali takšnih, ki so preveliki za shranjevanje v hitri pomnilnik. [2, 18]

Enota za računanje v Maxelerjevi podatkovno-pretokovni enoti je ena urina perioda (angl. tick) [18], za katero bomo v nadaljevanju uporabljali izraz *tik*. Med enim tikom Maxelerjeva podatkovno-pretokovna enota izvede en korak računanja, prebere en podatek iz vhoda in zapiše en podatek na izhod. Pri tem moramo vedeti, da nekatere operacije trajajo več tikov. Tako na primer množenje in seštevanje trajata okoli 12 tikov. Zato je pri programiranju na Maxelerjevi podatkovno-pretokovni enoti pogosto izziv, da med

čakanjem na rezultat ene operacije izvajamo druge operacije algoritma in ne blokiramo branja vhodnih podatkov.

Podjetje Maxeler Technologies ponuja različne razširitvene kartice, ki so prilagojene glede na namen računanja, dodatno pa ponuja tudi podatkovno-pretokovno računanje v oblaku s tehnologijo MaxCloud [20, 19].

2.3 Programiranje

Program, ki se izvaja na Maxelerjevi podatkovno-pretokovni enoti, je sestavljen iz treh delov [18, 19]:

- programa za centralno procesno enoto (v nadaljevanju glavni program),
- ščepca (angl. kernel) za izvajanje operacij nad podatki znotraj podatkovno-pretokovne enote,
- nadzornega programa (angl. manager) za usmerjanje podatkov znotraj podatkovno-pretokovne enote.

2.3.1 Glavni program

Programer mora najprej napisati kodo za centralno procesno enoto, ki služi kot vstopna točka programa. Imenovali jo bomo glavni program. To je običajno program v programskem jeziku C, prek katerega se prebere podatke in se jih pošlje na podatkovno-pretokovno enoto za obdelavo. Ko se podatki obdelajo na podatkovno-pretokovni enoti, se rezultat vrne v glavni program za nadaljnjo uporabo. Koda 2.1 prikazuje primer enostavnega glavnega programa [18].

Poleg programskega jezika C so podprti tudi nekateri drugi programski jeziki, kar je opisano v dokumentaciji [18]. Tako lahko za glavni program uporabimo tudi programske jezike Python, Matlab ali R, vendar je treba programske vezave (angl. binding) posebej prevesti. Zaradi tega smo se pri programih za to magistrsko nalogo odločili za programski jezik C.

Koda 2.1: Primer kode glavnega programa

```
1  int main() {
2      // Podatki, nad katerimi delamo izracun
3      float* numbers = generateInputNumbers();
4
5      // Spremenljivka, v katero se bo zapisal rezultat
6      float* result = malloc(sizeof(float) * 4);
7
8      // Klic podatkovno-pretokovne enote, ki izvede
9      // izracun in zapise rezultat v "result"
10     CalculatateOnDataflow(numbers, result);
11
12     // Ostala koda glavnega programa
13     doSomethingUsefulWithResult(result);
14
15     return 0;
16 }
```

2.3.2 Ščepec

Programer mora nato definirati vsaj en ščepec (angl. kernel). Ta služi za izvajanje operacij nad podatki na podatkovno-pretokovni enoti. Podatke program prebere kot tok (angl. stream), ki mu jih poda glavni program. Nato podatke obdela, izračuna nove vrednosti in jih vrne na izhod. Primer ščepca, ki pomnoži tok števil s koeficientom, je podan v kodi 2.2. Za ta primer bi v glavnem programu tok podali kot tabelo števil, na izhodu pa bi dobili novo tabelo spremenjenih vrednosti.

Za ščepec se uporablja programski jezik MaxJ [17, 18], ki je nekoliko prirejena različica Jave 6. Znotraj ščepca lahko uporabljamo standardno knjižnico Jave, dodatno pa tudi ostale knjižnice za podatkovno-pretokovno enoto Maxeler. Sintaksa jezika je enaka Javi, torej lahko uporabljamo vse elemente Jave kot so dedovanje, abstraktni razredi in razredni vmesniki. Dodani pa so nekateri sintaktični elementi, kot je operator "<==" za povezanje

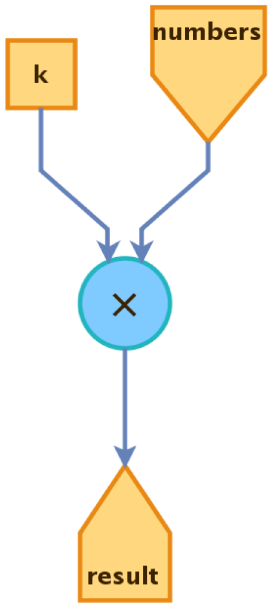
Koda 2.2: Primer kode ščepca

```
1  public class NumberTimesKKernel extends Kernel {
2
3      NumberTimesKKernel(KernelParamaters params) {
4          // Preberemo koeficient kot skalar
5          DFEVar k = io.scalarInput("k", dfeFloat(8, 24));
6          // Preberemo trenutno stevilo iz toka stevil
7          DFEVar number = io.input("numbers", dfeFloat(8, 24));
8
9          // Pomnozimo stevili
10         DFEVar result = k * number;
11
12         // Rezultat zapisemo na izhod
13         io.output("result", result, dfeFloat(8, 24));
14     }
15
16 }
```

podatkovnih tokov.

Ker so vsi podatki, ki jih obdelujemo, posebnega tipa DFEVar in so tudi rezultati operacij med podatki tipa DFEVar, obstajajo določene omejitve pri programiranju. Tako na primer ne moremo uporabljati zank z besedo *for*, če število obhodov ni znano v času prevajanja. Prav tako ne moremo uporabljati besede *if*. Za dinamične zanke moramo uporabljati podatkovne tokove, za dinamične *if* stavke pa multiplekserje, ki jih implementiramo s trikomponentnim operatorjem *?:*.

Koda ščepca predstavlja graf podatkov, ki se izvaja na FPGA-ju. Ta graf zaradi direktne predstavitve na FPGA-ju ne moremo dinamično spreminjati med izvanjanjem, pot podatkov pa mora biti vnaprej znana. Podan ščepec 2.2 se tako prevede v graf, prikazan na sliki 2.3. Z grafa lepo vidimo pot podatkov, kako se tok k množi s tokom *numbers* in po končani operaciji zapiše na izhod.



Slika 2.3: Graf ščepca, podanega v kodi 2.2

2.3.3 Nadzorni program

Za povezavo med glavnim programom in ščepci ter za vodenje tokov podatkov skrbi nadzorni program [17, 18]. V njem določimo parametre, ki jih sprejmejo ščepci pred izračunom in katere podatke po izračunu ščepci vrnejo. V nadzornem programu lahko tudi nastavimo frekvenco pomnilnika podatkovno-pretokovne enote ipd. V kodi 2.3 je predstavljen primer nadzornega programa, ki povezuje dva ščepca. Pri tem se najprej izvede prvi ščepec, njegov rezultat tega pa se uporabi v drugem ščepcu. Celoten rezultat se nato vrne v glavni program.

2.4 Prevajanje in izvajanje

Vse te komponente se pri prevajanju povežejo in prevedejo v izvršljiv program. Povezave med komponentami so prikazane na sliki 2.4. S slike lahko vidimo, da se ščepec našega programa najprej poveže z nadzornim programom in prevede v konfiguracijsko datoteko podatkovno-pretokovne enote s

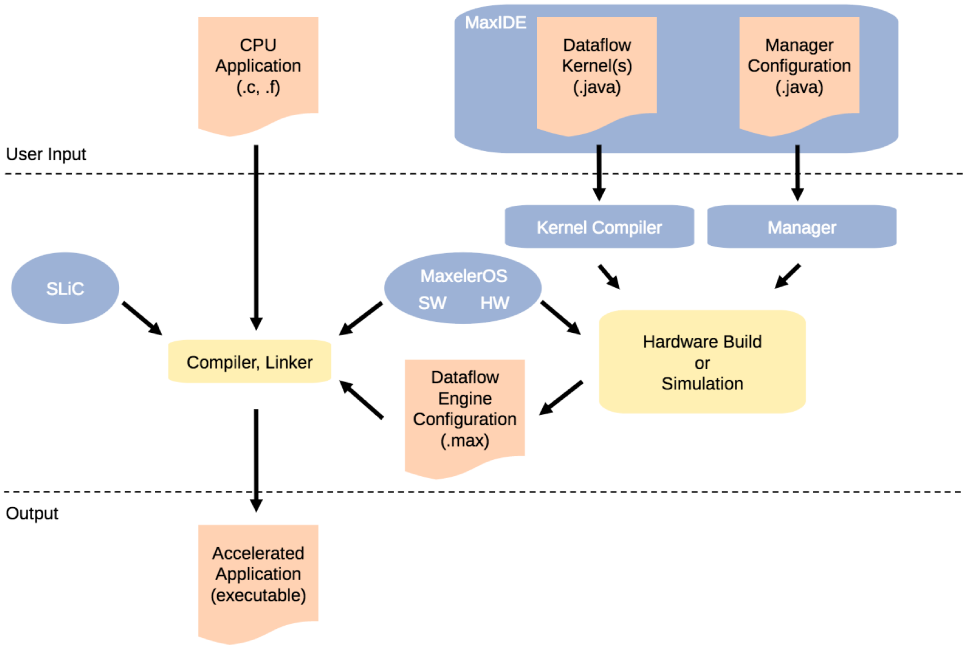
končnico *.max*. Konfiguracijska datoteka se nato v drugem koraku poveže z glavnim programom in prevede v izvršljiv program, ki je primeren za izvajanje [21].

Koda 2.3: Primer kode nadzornega programa

```
1  public class Manager extends CustomManager {
2
3      public Manager(EngineParameters params) {
4          super(params);
5          // Inicializiramo oba scepca
6          KernelBlock kernel1 = new FirstKernel();
7          KernelBlock kernel2 = new SecondKernel();
8
9          // Tok glavnega programa podamo kot vhod v prvi scepec
10         kernel1.getInput("nums") <== addStreamFromCPU("nums");
11         // Rezultat prvega scepca podano kot vhod v drugi scepec
12         kernel2.getInput("nums") <== kernel1.getOutput("res");
13
14         // Rezultat drugega podamo kot izhod v glavni program
15         addStreamToCPU("result") <== kernel2.getOutput("res");
16     }
17
18 }
```

Za namene implementacije nam Maxeler ponuja orodje MaxIDE [22], ki temelji na razvojnem okolju Eclipse [23]. Maxeler nam ponuja tudi simulator podatkovno-pretokovne enote, ki nam omogoča hitro prevajanje in izvajanje naše kode. Vendar simulator ni primeren za testiranje na velikih vhodnih podatkih, zato mora del testiranja še vedno potekati na pravi podatkovno-pretokovni enoti. Če se koda prevede na simulatorju namreč še ne pomeni, da se bo tudi na podatkovno-pretokovni enoti. Podobno velja, če se naš program uspešno izvede na simulatorju; ni nujno, da se bo tudi na podatkovno-pretokovni enoti.

To se na primer lahko zgodi, če ima naš program več operacij, kot jih lahko čip naenkrat izvaja. Zaradi tega je običajno priporočljivo, da naš program



Slika 2.4: Povezava med komponentami in prevajanje

najprej implementiramo na centralni procesni enoti in ga postopoma v več iteracijah prenesemo na podatkovno-pretokovno enoto. Primerni koraki in problemi pri prenosu algoritmov so opisani v članku [24]; nasvete v njem smo poskušali upoštevati tudi mi.

Poglavje 3

Evalvacija polinomov

3.1 Gosti polinomi v eni točki

Najprej smo implementirali evalvacijo gostih polinomov v eni točki, kot je definirano v enačbi 1.1. Implementacija evalvacije takega polinoma v eni točki v ukazno-pretokovnem programu je podana v kodi 3.1. Ukazno-pretokovna rešitev uporablja Hornerjev algoritem.

V ukazno-pretokovnem programu je koda preprosta. Sprehodimo se od zadnjega koeficienta proti prvemu. V vsakem koraku iteracije množimo točko x z rezultatom iz prejšnega koraka in mu prištejemo koeficient. Ta ukazno-pretokovni program nam bo služil kot opora za implementacijo na Maxeler-

Koda 3.1: Ukazno-pretokovna implementacija evalvacije gostega polinoma v eni točki

```
1 float evaluatePolynomial(float x, float* ks, int n) {
2     float result = 0.0f;
3     for (int i = n - 1; i >= 0; i--) {
4         result += result * x + ks[i];
5     }
6     return result;
7 }
```

jevi podatkovno-pretokovni enoti. Z implementacijo evalvacije gostih polinomov v eni točki smo se spoznali z Maxelerjevo podatkovno-pretokovno enoto in njeno arhitekturo, ter jo poizkušali razumeti. Pridobljeno znanje je služilo kot osnova za razvoj nadaljnjih algoritmov.

Dana implementacija ni najbolj optimalna, vendar je dobro izhodišče za implementacijo evalvacije v več točkah. Izboljšana rešitev evalvacije v eni točki bo opisana v razdelku 3.3.

3.1.1 Podatkovno-pretokovna implementacija

Implementacija evalvacije gostih polinomov v eni točki je služila kot učni model. Spoznali smo osnovne konstrukte programa na podatkovno-pretokovni enoti:

- števce,
- branje podatkov iz vhoda,
- dinamične zanke,
- pisanje na izhod.

Ščepec implementacije evalvacije je prikazan v kodi 3.2.

Števci so posebni konstrukti, ki so namenjeni sledenju toka podatkov. V danem ščepcu imamo dva števca, *tickCount* in *counter*, definirana v vrsticah od 11 do 14. Ob vsakem koraku izračuna oziroma tiku se njuna vrednost poveča za ena, kot bi se v algoritmu ukazno-pretokovnega programa. Tako lahko nadziramo tok podatkov, simuliramo ukazno-pretokovne zanke, inicializiramo začetne vrednosti ipd. Poleg tega lahko konstruiramo tudi bolj zapletene števce, na primer take, ki so medsebojno odvisni, števce, ki se jim zmanjšuje vrednost ali take, katerim se vrednost povečuje s korakom, različnim od 1, ipd.

Drugi konstrukt je branje podatkov, prikazan v vrsticah od 16 do 18. V danem ščepcu sta prikazana branje toka koeficientov k in branje skalarja x.

Branje toka podatkov lahko nadziramo s podanim pogojem. Pogoj podamo kot tretji argument funkcije *io.input()*. V dani implementaciji tako iz toka koeficientov beremo, ko je števec operacij *counter* enak 0, kar vidimo v vrstici 17. V vseh ostalih primerih pa podatki čakajo na branje.

Tretji konstrukt so dinamične zanke. Maxelerjeva podatkovno-pretokovna enota ne pozna klasičnih dinamičnih zank, kjer število iteracij ni znano v času prevajanja programa. Zato uporabimo posebne dinamične zanke, zapisane v obliki grafa. Ta zapis je prikazan v vrsticah od 20 do 31. Tako najprej inicializiramo spremenljivko *result* v vrstici 21, v katero ciklično zapisujemo delni rezultat v vrstici 31. Pri tem moramo upoštavati dolžino izvajanja operacij seštevanja in množenja, zato rezultat zapisujemo z odmikom *offset*. Ta odmik je enak času izvajanja množenja in seštevanja.

V vrstici 35 je nato prikazano pisanje na izhod. Pri dani implementaciji smo na izhod pisali kar vsak vmesni rezultat. Tako je bil rezultat celotne evalvacije zapisan v zadnjem elementu izhoda, vsi ostali elementi pa so pravzaprav delne vsote evalvacije.

Dodatna posebnost Maxelerjeve podatkovno-pretokovne enote, ki bi jo radi predstavili s to implementacijo, je, da rezultat operacij množenja in seštevanja ni takoj na voljo. Kot že prej omenjeno, zaradi tega dinamično zanko izvajamo z zamikom nekaj tikov. Operacije na podatkovno-pretokovni enoti namreč trajajo nekaj tikov. V primeru kartice Vectis, ki smo jo imeli na voljo za testiranje, je tako čas množenja 12 tikov. Zaradi tega tudi podatke iz vhoda v dani implementaciji beremo šele, ko vemo, da so se predhodne operacije izvedle v celoti. Samo tako bo rezultat pravilen in program se bo izvedel do konca. S tem izgubimo veliko časa, zato je to ena od možnosti za optimizacijo. V vrstici 9 in 10 sta podani spremenljivki, ki nosita informacijo o dolžini operacij. Potrebno število korakov za izračun lahko prevajalnik Maxeler predvidi sam. Velikokrat pa to vrednost definiramo kar sami na tako vrednost, ki se najbolje prilagaja razporeditvi ali velikosti vhodnih podatkov. Za primer, velikokrat je za optimalno implementacijo pomembno, da je ta vrednost deljiva z velikostjo našega vhodnega toka.

Koda 3.2: Evalvacija gostega polinoma v eni točki

```
1  public class EvaluationKernel extends Kernel {
2
3      static final DFEType FLOAT = dfeFloat(8, 24);
4
5      EvaluationKernel(KernelParameters parameters) {
6          super(parameters);
7
8          // Stevci zanke
9          OffsetExpr offset = stream.makeOffsetAutoLoop("offset");
10         DFEVar opsLength = offset.getDFEVar(this, dfeUInt(64));
11         // Stevec, s katerim sledimo trenutnemu taku
12         DFEVar tickCount = control.count.simpleCounter(64);
13         // Stevec, ki steje od 0 do stevila operacij
14         DFEVar counter = control.count.simpleCounter(0, opsLength);
15
16         // Branje iz toka podatkov vsakic ko je stevec na 0
17         DFEVar kIn = io.input("k", FLOAT, counter === 0);
18         DFEVar xIn = io.scalarInput("x", FLOAT);
19
20         // Zanka evalvacije
21         DFEVar loopResult = FLOAT.newInstance(this);
22         DFEVar loopX = FLOAT.newInstance(this);
23         DFEVar loopPowX = FLOAT.newInstance(this);
24         DFEVar x = tickCount === 0 ? xIn : loopX;
25         DFEVar powX = tickCount === 0 ? 1.0 : powX * x;
26         DFEVar result = tickCount === 0
27             ? kIn
28             : (kIn * powX + loopResult);
29         loopX <== stream.offset(x, -offset);
30         loopPowX <== stream.offset(powX, -offset);
31         loopResult <== stream.offset(result, -offset);
32
33         // Zapis rezultata na izhod,
34         // ki je enake dolzine kot vhod podatkov
35         io.output("result", result, FLOAT);
36     }
37
38 }
```

3.2 Gosti polinomi v več točkah

Poleg evalvacije polinoma v eni točki smo implementirali tudi evalvacijo v več točkah. To pomeni, da bomo za različne vhodne točke x izračunali polinom, za vsako točko posebej. Pri tem bomo uporabili vse tehnike implementacije, ki smo jih spoznali v razdelku 3.1. Evalvacijo v več točkah smo implementirali za goste polinome. Kasneje bomo v razdelku 3.4 pokazali, kako jo lahko enostavno razširimo tudi na redke polinome.

Ukazno-pretokovni algoritem lahko zapišemo kot preprosto zanko evalvacij za enojno točko iz razdelka 3.1 in kode 3.1. Ukazno-pretokovna implementacija je predstavljena v kodi 3.3.

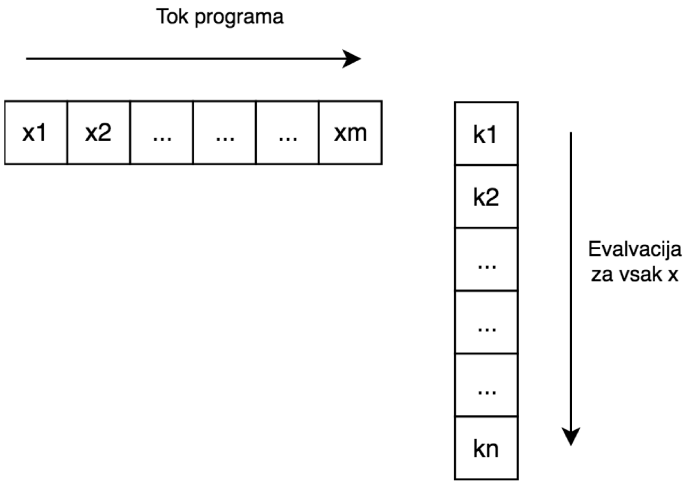
Koda 3.3: Ukazno-pretokovna implementacija evalvacije v več točkah

```
1 float* evaluateMultiPoint(float* xs, float* ks, int n, int m) {
2     float* results = malloc(sizeof(float) * m);
3     for (int i = 0; i < m; i++) {
4         results[i] = evaluatePolynomial(xs[i], ks, n);
5     }
6     return results;
7 }
```

Podobno bi lahko implementirali rešitev, kjer bi klicali podatkovno-pretokovni program iz razdelka 3.1 in kode 3.2. Znotraj glavnega programa bi se sprehodili čez točke x in vsakič poklicali vmesnik podatkovno-pretokovnega programa. Ta bi evalviral polinom za vsako točko posebej. Vendar pa bomo v nasprotju s tem v tem razdelku opisali učinkovitejšo implementacijo, ki bolje izkoristi podatkovno-pretokovno arhitekturo.

3.2.1 Podatkovni graf

V razdelku 3.1 smo spoznali principe podatkovno-pretokovnega programiranja, pri problemu evalvacije gostega polinoma v več točkah pa smo se problema najprej lotili s podatkovnega vidika. Pred samo implementacijo smo najprej poskušali ugotoviti, kako najbolje izkoristiti podatkovno-pretokovno



Slika 3.1: Smer podatkovnih tokov evalvacije v več točkah

arhitekturo, in si zamisliti same tokove podatkov. Pri programiranju pa smo se poskušali držati nasvetov programiranja na podatkovno-pretokovni enoti, podanih v [24].

Iz ukazno-pretokovne implementacije v kodi 3.3 hitro vidimo, da so vstopni podatki štirje:

- točke *xs*, za katere evalviramo polinom,
- koeficienti *ks* danega polinoma,
- velikost n danega polinoma,
- število točk m.

Opazimo lahko, da se ves čas premikamo po x-ih, medtem ko tok koeficientov zaokroži za vsak x posebej. V ukazno-pretokovnem programu to predstavimo kot dve zanki, grafično pa bi lahko predstavili dva podatkovna tokova, pravokotna eden na drugega, kot na sliki 3.1. Če pogledamo predstavitev, tok programa teče v smeri toka x-ov. Medtem pa teče tudi tok koeficientov, ki se pretoči za vsak x. Lahko bi tudi obrnili situacijo, vendar pa bi bilo zaradi implementacijskih posebnosti, kot je shranjevanja rezultatov

za vse x-e, to težje. Tok programa, kjer sledimo toku podatkov x, bo osnova za implementacijo, za razumevanje implementacije pa si moramo zapomniti, da se premikamo po točkah x, medtem ko morajo koeficienti k zaokrožiti za vsak x.

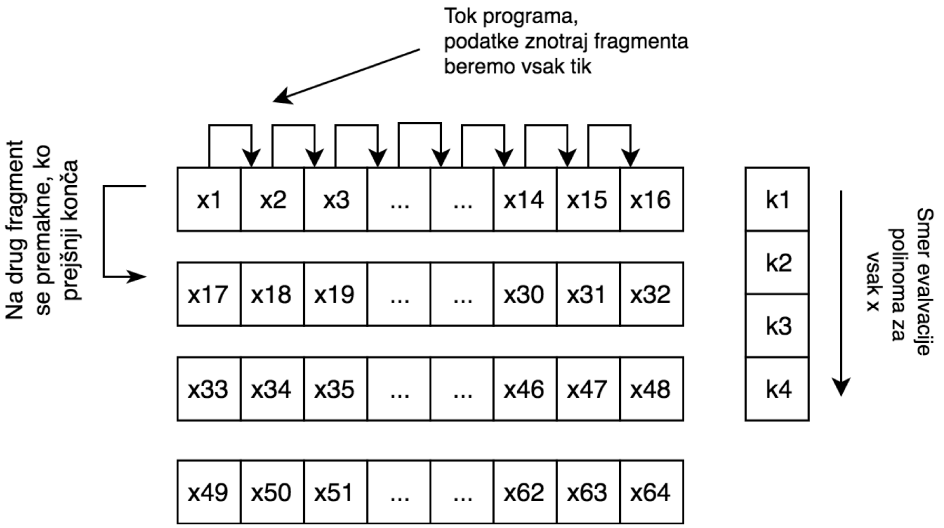
3.2.2 Implementacija

Skozi implementacijo smo spoznali, da ima Maxelerjeva podatkovno-pretokovna enota omejitev pri toku podatkov, ki ga ne moremo ponavljati oziroma se po njem ne moremo krožno sprehajati. Ta del pa je nujen za večkratno evalvacijo polinoma. Zato smo morali za pravilno delovanje programa polinom shranjevati v hitri pomnilnik, ki ga Maxeler označuje kot FMEM [18].

Pri uporabi hitrega pomnilnika pa pridemo do omejitve števila elementov, ki jih lahko shranimo. Hitri pomnilnik ima namreč omejeno velikost na nekaj megabajtov. Zaradi tega smo omejili naš algoritem na maksimalno 8192 koeficientov polinoma. Zgornje meje števila točk nismo omejili. Izbrana meja števila koeficientov je zelo nizka, saj 8192 števil v enojni natančnosti zavzema le okrog 32 kB. Če bi potrebovali večje polinome, bi lahko to število še nekajkrat povečali. Če pa bi hoteli evalvirati polinome z več milijoni koeficientov, bi morali izkoristiti višjenivojski počasnejši pomnilnik, kamor lahko shranimo nekaj gigabajtov podatkov. Za večino problemov pa naša izbrana meja zadostuje.

Problem evalvacije v več točkah je očitno samo razširjen problem evalvacije v eni točki. Vendar preprosta razširjena implementacija algoritma, kjer bi evalvirali točke zaporedno, najprej prvo točko, nato drugo itd., ne bi bila nujno najbolj optimalna. Zaradi zakasnitve operacij množenja in seštevanja bi namreč za vsako točko izgubili nekaj tikov. Pri več milijonih točk pa nekaj tikov na točko pomeni veliko izgubljenega časa. Zato je rešitev, da algoritem prilagodimo in točke izmenično evalviramo.

Za namene izmenične evalvacije smo ubrali pristop navidezne fragmentacije podatkov. Tako točke razdelimo v več fragmentov določene velikosti, znotraj fragmenta pa se izvaja izmenjujoča evalvacija točk. Na sliki 3.2 je



Slika 3.2: Podatkovni tok evalvacije v več točkah in fragmentacija

predstavljena razdelitev točk na fragmente, kjer vsak fragment vsebuje 16 točk. Dodatno je prikazan tudi potek programa, ki naenkrat evalvira samo točke v enem fragmentu izmenjajoče. To pomeni, da najprej množimo točko *x1* za koeficient *k1*, nato točko *x2* za koeficient *k1* vse do točke *x16* in šele nato zopet točko *x1* za koeficient *k2*. Ko se evalvacija za celoten fragment zaključi, pa se premaknemo na naslednji fragment. Algoritem ima tako naslednje korake:

1. preberi fragment in vzporedno zapiši polinom v hitri pomnilnik,
2. evalviraj polinom za vsak element fragmenta izmenjujoče,
3. preberi naslednji fragment,
4. ponavljaj korake od 2-4 do konca fragmentov.

Implementacija je podana v kodi 3.4 in kodi 3.5. V kodi 3.4 so podani števci in vhodni podatki. Iz vhoda beremo število točk in velikost polinoma ter koeficiente in točke. Uporabljamo veriženje števcev, kjer se sprehajamo po točkah od 0 do velikosti fragmenta, kar je prikazano v vrstici 6. Ko števec po točkah pride do konca, se poveča še števec trenutnega koeficienta. Ko oba števca prideta do konca, začnemo od začetka za naslednji fragment. Na ta način kontroliramo tok in se premikamo skozi točke in polinom.

V vrsticah 12 do 16 sta prikazana branje in pisanje koeficientov v pomnilnik. Koeficiente beremo vsak tik. Ker pisanje traja eno periodo, beremo trenutni koeficient iz pomnilnika šele, ko so vsi elementi že zapisani. Predtem pa koeficiente polinoma beremo kar iz toka podatkov. To lahko vidimo v vrsticah od 14 do 16.

V kodi 3.5 je prikazana glavna dinamična zanka programa, v kateri krožijo točke, njihove potence in delni rezultat. Inicializacija spremenljivk dinamične zanke se izvede v vrsticah od 2 do 4, evalvacija pa v vrsticah od 5 do 10. Posebnost imamo pri prvem členu polinoma, kjer spremenljivke dinamične zanke še nimajo vrednosti.

Ko števec trenutnega koeficienta pride do velikosti polinoma, rezultat pošljemo na izhod v vrstici 16. Izhodni podatki se potem pošljejo glavnemu programu, ki lahko podatke naprej obdeluje.

Koda 3.4: Števci evalvacije gostih polinomov v več točkah

```
1  // Input in stevci
2  DFEVar n = io.scalarInput("n", UINT_32);
3  DFEVar m = io.scalarInput("m", UINT_32);
4  CounterChain chain = control.count.makeCounterChain();
5  DFEVar kIndex = chain.addCounter(n, 1);
6  DFEVar xIndex = chain.addCounter(fragmentSize, 1);
7  DFEVar tickCount = control.count.simpleCounter(64);
8
9  DFEVar readAndWriteK = tickCount < n;
10 DFEVar kIn = io.input("coefficients", FLOAT, readAndWriteK);
11 DFEVar xIn = io.input("xs", FLOAT, kIndex === 0);
12 Memory<DFEVar> memory = mem.alloc(FLOAT, maxPolynomialLength);
13 memory.write(kIndex.cast(addressType), kIn, readAndWriteK);
14 DFEVar k = (readAndWriteK)
15     ? kIn
16     : memory.read(kIndex.cast(addressType));
```

Koda 3.5: Zanka evalvacije gostih polinomov v več točkah

```
1  // Glavna zanka
2  DFEVar loopX = FLOAT.newInstance(this);
3  DFEVar loopPowX = FLOAT.newInstance(this);
4  DFEVar loopResult = FLOAT.newInstance(this);
5  DFEVar x = (kIndex === 0) ? xIn : loopX;
6  DFEVar powX = (kIndex === 0) ? 1.0 : x * loopPowX;
7  DFEVar multiply = powX * k
8  DFEVar result = (kIndex === 0)
9                  ? multiply
10                 : multiply + loopResult;
11 loopX <== stream.offset(x, -fragmentSize);
12 loopPowX <== stream.offset(powX, -fragmentSize);
13 loopResult <== stream.offset(result, -fragmentSize);
14
15 // Pisanje na izhod
16 io.output("result", result, FLOAT, kIndex === n - 1);
```

3.3 Redki polinomi v eni točki

Implementacijo evalvacije gostega polinoma v eni točki iz razdelka 3.1 bi radi prilagodili za evalvacijo redkega polinoma. V razdelku 3.1 smo omenili, da dana implementacija evalvacije v eni točki ni najbolj optimalna. Implementacijo za redke polinome bi radi optimizirali in primerjali z ukazno-pretokovno implementacijo na centralni procesni enoti. Pri tem lahko identificiramo nekaj problemov algoritma iz razdelka 3.1, ki bi potrebovali prilagoditev. Problemi so:

- podatki se ne berejo vsak tik,
- med računanjem ene operacije ne opravljamo nobene druge operacije,
- v enem tiku ne moremo izračunati poljubne potence,
- potence morajo biti zaporedne, razporejene po velikosti.

Če se podatki ne berejo vsak tik in med računanjem ene operacije ne opravljamo nobene druge operacije, izgubljamo veliko tikov, kjer del čipa ne dela ničesar in ga ne izkoristimo popolno. Zato za čim boljšo izkoriščenost čipa stremimo k računanju v vsakem tiku. Če ta pogoj ni izpolnjen, se bo program izvedel v več tikih in bo trajal dlje.

Potrebno bi bilo računanje poljubne potence v enem tiku, saj so lahko pri redkih polinomih eksponenti neenakomerno razporejeni glede na velikost. Učinkovita implementacija potenciranja bi pripomogla k hitrejšemu izvajanju in manjšemu številu korakov evalvacije. Če bi poleg tega dovolili, da potence niso razporejene po velikosti, pa bi se izognili predhodnemu urejanju elementov glede na velikost eksponenta.

Poleg tega, če bi hoteli razviti algoritem za evalvacijo redkega polinoma v več točkah, bi nujno morala obstajati možnost evalvacije potenc za več točk. Zato bi radi implementirali učinkovit algoritem potenciranja, ki bi zadostil tudi temu pogoju.

3.3.1 Potenciranje

Naivna rešitev

Pri implementaciji smo se najprej lotili izračuna potence danega števila x za dan eksponent. Naivna rešitev je vsebovala poseben ščepec, rezultat katere pa je bil tok potenc števila x od 0 do 1024. Algoritem lahko zapišemo kot rekurzivno relacijo:

$$y_0 = 1,$$
$$y_n = y_{n-1}x.$$

Ukazno-pretokovni algoritem bi lahko zapisali kot:

Koda 3.6: Ukazno-pretokovni algoritem izračuna prvih 1024 potenc števila x

```
1 float* exp(x) {
2     float result[1024] = { 1.0 };
3     for (int i = 1; i < 1024; i++) {
4         result[i] = x * result[i - 1];
5     }
6     return result;
7 }
```

To ukazno-pretokovno rešitev smo pretvorili v ščepec, ki ustreza Maxelerjevi podatkovno-pretokovni enoti, kar je prikazano v kodi 3.7. Podobno kot v implementaciji evalvacije gostega polinoma v eni točki (koda 3.2) pridobimo informacijo o dolžini trajanja operacij, ki jih potrebujemo za pravilno izvedbo algoritma, in sicer v vrstici 9. Nato iz vhoda preberemo x v vrstici 13. Potem zopet ustvarimo dinamično zanko kot graf, v katerega zapisujemo potence x, kar je prikazano v vrsticah od 15 do 18. Nazadnje pa ob vsakem izračunu rezultat zapišemo na izhod v vrstici 22.

V primeru uporabe te implementacije bi se najprej izvedel opisan ščepec za potenciranje, tok izračunanih potenc pa bi se nato podal drugemu ščepcu za evalvacijo polinoma. Ščepec za evalvacijo polinoma bi priredili tako, da

Koda 3.7: Ščepec za računanje potenc števila x

```
1  public class PowX1024Kernel extends Kernel {
2
3      static final DFEType FLOAT = dfeFloat(8, 24);
4
5      PowX1024Kernel(KernelParameters parameters) {
6          super(parameters);
7
8          // Stevci zanke
9          OffsetExpr offset = stream.makeOffsetAutoLoop("offset");
10         DFEVar tickCount = control.count.simpleCounter(64);
11
12         // Vhod X
13         DFEVar x = io.scalarInput("x", FLOAT);
14
15         // Zanka evalvacije
16         DFEVar result = FLOAT.newInstance(this);
17         DFEVar powX = tickCount === 0 ? 1.0 : (x * result);
18         result <== stream.offset(powX, -offset);
19
20         // Zapis rezultata na izhod,
21         // ki je enake dolzine kot vhod podatkov
22         io.output("result", powX, FLOAT);
23     }
24
25 }
```

bi sprejel tok izračunanih potenc in tok eksponentov, ki mu jih poda uporabnik. Eksponent bi služil kot indeks potence, sej je tok izračunanih potenc razvrščen v naraščajočem vrstnem redu od x^0 do x^{1024}. Do izračunane potence bi nato program dostopal z odmikom toka; potenca x^0 bi bila dostopna na odmiku 0, potenca x^1 na odmiku 1 itd. Rezultat bi nato na enak način kot v implementaciji evalvacije ene točke za goste polinome zapisali na izhod.

Optimizirana rešitev

V prejšnjem razdelku smo implementirali naivno potenciranje, ki nam ne ustreza. Implementacija je namreč kompleksna, saj zahteva dodaten ščepec, zato bi radi našli boljšo rešitev.

Kaj sploh je potenciranje? Potenciranje v našem primeru lahko zapišemo kot:

$$x^n = \begin{cases} 1, \text{ če je n} = 0 \\ \prod_{i=1}^{n} x, \text{ če je n} > 0 \ . \end{cases} \tag{3.1}$$

V magistrski nalogi obravnavamo samo evalvacijo polinomov, zato smo izpustili primere, ko je $n < 0$ in ko n ni celo število. Zelo enostavno pa bi lahko naše algoritme prilagodili tudi takim primerom. Enačbo iz 3.1 bi v ukazno-pretokovnem jeziku zapisali kot:

Koda 3.8: Preprosto ukazno-pretokovno potenciranje

```
1 float pow(float x, int n) {
2     float result = 1.0f;
3     while(n--- > 0) {
4         result = result * x;
5     }
6     return result;
7 }
```

Hitro opazimo, da lahko zmanjšamo število operacij, če pri množenju uporabimo že izračunane vrednosti. Tako lahko naprimer x^4 izračunamo kot $x^4 = x*x*x*x$, za kar porabimo štiri množenja. Lahko pa najprej izračunamo $x^2 = x * x$, si rezultat zapomnimo in končno izračunamo še $x^4 = x^2 * x^2$. S

tem smo člen x^2 izračunali samo enkrat in namesto štirih množenj uporabili dve.

Izkaže se, da lahko za cele eksponente n izračunamo potenco števila s kompleksnostjo $O(\log n)$ [25]. Algoritem temelji na lastnosti, da lahko potenco s celim eksponentom izrazimo kot potenco nižjih potenc na naslednji način [25]:

$$x^n = \begin{cases} x(x^2)^{\frac{n-1}{2}}, \text{ če n je lih} \\ (x^2)^{\frac{n}{2}}, \text{ če n je sod .} \end{cases} \tag{3.2}$$

Lastnost iz enačbe 3.2 lahko uporabimo za algoritem potenciranja števil s celim eksponentom, ki ga lahko v ukazno-pretokovnem jeziku zapišemo kot:

Koda 3.9: Učinkovit izračun potence v ukazno-pretokovnem jeziku

```
1  float pow(float x, int n) {
2      float result = 1.0f;
3      while (n > 0) {
4          if (n % 2 != 0) {
5              result = x * result;
6          }
7          x = x * x;
8          n = n / 2;
9      }
10     return result;
11 }
```

Algoritem bomo prevedli kot *ponavljajoče kvadriranje* (angl. repeated squaring, tudi exponentiation by squaring). Ta algoritem smo uporabili tudi v naši podatkovno-pretokovni implementaciji.

Za podatkovno-pretokovni računalnik ga lahko implementiramo kar kot statično zanko; implementacija je prikazana v kodi 3.10. Statična zanka je drugačen konstrukt kot dinamična zanka, ki smo jo spoznali v razdelku 3.1.1. Pri statični zanki poznamo število obhodov v času prevajanja, rezultat prevajanja pa so podvojene operacije na čipu. Grafična ponazoritev statičnih zank je podana kasneje v razdelku 3.5.

Koda 3.10: Učinkovit izračun potence na podatkovno-pretokovni enoti

```
1  DFEVar powX(DFEVar x, DFEVar exp) {
2      DFEVar powX = constant.var(FLOAT, 1.0);
3      for (int i = 0; i < 10; ++i) {
4          powX = ((exp > 0) & ((exp & 1) === 1))
5              ? powX * x
6              : powX;
7          exp = exp >> 1;
8          x = x * x;
9      }
10     return powX;
11 }
```

Ker moramo za statično zanko vedeti število obhodov v času prevajanja, smo se mi odločili za maksimalni eksponent 1024. To pomeni desetkratno ponavljanje grafa istih operacij na podatkovno-pretokovni enoti, oziroma deset obhodov zanke. Za maksimalni eksponent smo se odločili na podlagi velikosti števil. Tako je naprimer 2^{1023} enako $8{,}988466 \cdot 10^{307}$, kar še lahko zapišemo s predstavitvijo števila v plavajoči vejici z dvojno natančnosto, kjer je maksimalna vrednost $1{,}7976931348623157 \cdot 10^{308}$. Potence 2^{1024} pa ne moremo več zapisati v dvojni natančnosti. V naših implementacijah smo tako ali tako uporabljali enojno natančnost, ki ima še nižjo mejo. Uporaba večjih potenc bi bila smotrna le, če bi evalvirali samo števila zelo blizu 1.

Uporaba statične zanke dodatno pomeni, da se vse potence, ne glede na eksponent, pretočijo po podatkovno-pretokovnem grafu iste dolžine. To posledično pomeni, da potenco z eksponentom 1 računamo enako dolgo kot potenco z eksponentom 1024. Če pa beremo podatke vsak tik in število podatkov presega dolžino grafa, to ne predstavlja težave. V takem primeru se namreč v vsakem vozlišču grafa vsak tik nahaja podatek, graf pa je polno izkoriščen.

Recimo, da za potenciranje enega števila potrebujemo 100 tikov. V primeru polno izkoriščenega grafa imamo torej izračun prvega števila na voljo

100. tik, izračun drugega 101. tik, izračun tretjega pa 102. tik itd. To pomeni, da je maksimalna zakasnitev 100 tikov za celoten program pri polno izkoriščenem grafu. Pri tisoč ali milijon vhodnih podatkih pa 100 tikov bistveno ne vpliva na čas izvajanja celotnega programa.

Kot že omenjeno, naša implementacija podpira samo evalvacijo celih eksponentov. Ker ima knjižnica prevajalnika Maxeler že vgrajeni funkciji izračuna eksponente funkcije e^x in naravnega logaritma, bi lahko v primeru računanja racionalne potence uporabili tudi enakost:

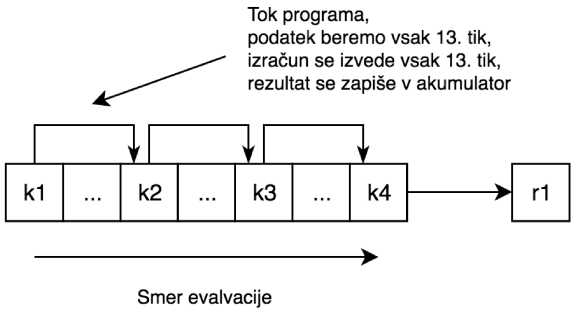
$$x^y = e^{y \ln x}.$$

3.3.2 Množenje in seštevanje

V razdelku 3.3.1 smo rešili problem potenciranja. Kot je opisano, je za učinkovito potenciranje treba brati podatke vsak tik, da popolnoma izkoristimo podatkovno-pretokovni graf. V tem delu bomo poskusili rešiti problem branja podatkov. Obenem pa bomo prilagodili tudi vrstni red evalvacije, kar bo vplivalo na hitrost izvajanja celotnega algoritma, in ne samo potenciranja.

Za implementacijo evalvacije gostega polinoma v eni točki smo v razdelku 3.1 zapisali, da se podatki iz vhodnega toka berejo samo, ko se operaciji množenja in seštevanja izvedeta do konca in smo prepričani, da bo rezultat pravilen. Iz simulatorja smo razbrali, da za evalvacijo enega člena polinoma potrebujemo 12 tikov na Maxelerjevi podatkovno-pretokovni enoti *Vectis*. To pa pomeni, da se vsak vhodni podatek prebere vsak 13. tik. Kako pa zagotoviti, da se vhod prebere vsak tik?

Izkaže se, da to lahko izvedemo s pomočjo spremembe koreografije podatkov [2, 18], podobno, kot smo to storili v večtočkovni implementaciji evalvacije gostega polinoma v razdelku 3.2. V enotočkovni implementaciji evalvacije gostih polinomov iz razdelka 3.1 so se podatki brali eden za drugim in seštevali v istem vrstem redu, kot so se brali. Mi bomo izrabili asociativne in komutativne lastosti seštevanja in množenja, ki smo jih omenili v razdelku 1.4. Prilagodili bomo vrstni red računanja in s tem omogočili branje podatkov vsak tik.



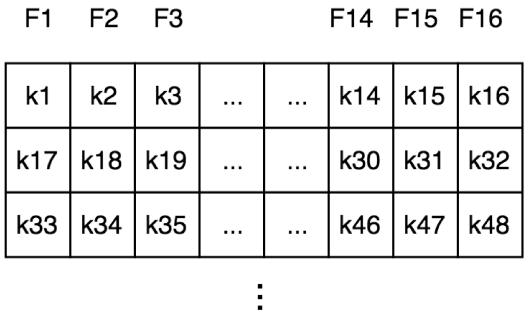
Slika 3.3: Graf toka naivne evalvacije v eni točki

Če pogledamo implementacijo iz razdelka 3.1, potem zanjo lahko prikažemo graf toka podatkov, kot je prikazan na sliki 3.3. Z grafom smo hoteli prikazati, kako slabo smo izkoristili podatkovno-pretokovni računalnik, saj beremo podatke in izvedemo en izračun vsak 13. tik. Kot glavni problem lahko označimo zapisovanje rezultata v en sam akumulator. Zaradi zapisovanja v en akumulator moramo namreč vedno čakati, da se prejšnje operacije nad akumulatorjem izvedejo do konca, šele nato lahko izvedemo naslednje.

Da bi rešili ta problem, smo polinom navidezno razdelili na 16 fragmentov. Razdelitev koeficientov polinoma je prikazana na sliki 3.4. Vsak stolpec predstavlja en fragment, označen z F1 do F16, koeficienti pa so podani kot k1, k2 itd. V fragmentu, označenim z F1, imamo torej koeficiente k1, k17 in k33, v fragmentu, označenim z F2, so k2, k18, k34 itd. Enako smo razdelili tudi podane eksponente polinoma, da bi sovpadali s pripadajočimi koeficienti.

Tako lahko celotno razdelitev zapišemo kot:

$$
\begin{aligned}
F_1(x) &= k_1 x^{y_1} + k_{17} x^{y_{17}} + k_{33} x^{y_{33}} + ... \\
F_2(x) &= k_2 x^{y_2} + k_{18} x^{y_{18}} + k_{34} x^{y_{34}} + ... \\
F_3(x) &= k_3 x^{y_3} + k_{19} x^{y_{19}} + k_{35} x^{y_{35}} + ... \\
&\vdots \\
F_{16}(x) &= k_{16} x^{y_{16}} + k_{32} x^{y_{32}} + k_{48} x^{y_{48}} + ...
\end{aligned}
$$



Slika 3.4: Fragmentacija koeficientov polinoma

Rezultat celotne evalvacije pa kot:

$$p(x) = \sum_{i=1}^{16} F_i$$

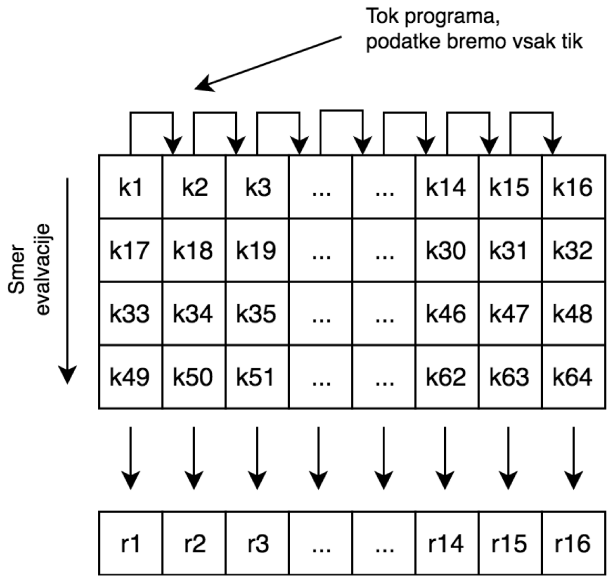
Ker evalviramo vsak fragment posebej in je vsota celotne evalvacije vsota fragmentov, podatki v fragmentu pa so oddaljeni za 16 enot, lahko implementiramo ščepec, ki bere tok podatkov vsak tik izmenično za fragmente in rezultat vsak tik izračuna. Tako smo ohranili tok programa, spremenili pa smer evalvacije. Tok programa in smer evalvacije sta prikazana na sliki 3.5.

3.3.3 Implementacija

Končna implementacija evalvacije redkih polinomov v eni točki je podana v kodi 3.11.

Najprej inicializiramo števec tikov programa *counter* in preberemo vhodne podatke v vrsticah od 10 do 14. Dolžino polinoma predstavlja spremenljivka n, k predstavlja trenutni koeficient iz toka podatkov, *exp* trenutni eksponent iz toka podatkov in x točko, v kateri evalviramo polinom. Števec tikov programa *counter* služi za inicializacijo vsote členov fragmentov in za pravilen zapis na izhod.

V vrsticah od 17 do 24 izračunamo potenco po algoritmu iz razdelka 3.3.1. Rezultat potenciranja shranimo v spremenljivko *powX*, ki jo kasneje upora-



Slika 3.5: Smer evalvacije in smer programa optimizirane evalvacije. Z *ri* so označeni rezultati za vsak fragmentent posebej

bimo pri množenju s koeficientom.

Nato sledi dinamična zanka v vrsticah od 27 do 30. V dinamični zanki izračunamo zmnožek koeficienta in rezultata potenciranja in ga prištejemo k vsoti členov fragmetna. V prvem obhodu fragmentov inicializiramo vsoto na 0 v vrstici 28. V vseh nadaljnih obhodih pa vzamemo vsoto iz zanke. Vsoto fragmentov beremo iz zanke z odmikom 16 tikov, kakršno je število fragmentov, kar predstavlja vrstica 30.

Nazadnje pa v vrstici 33 v zadnjih 16 tikih na izhod zapišemo vsote členov za vsak fragment posebej. Te vsote lahko nato seštejemo kar v glavnem programu in tako dobimo končni rezultat polinoma v dani točki.

Koda 3.11: Evalvacija redkega polinoma v eni točki

```
1  public class EvaluationKernel extends Kernel {
2
3      static final DFEType FLOAT = dfeFloat(8, 24);
4      static final DFEType UINT_32 = dfeUInt(32);
5
6      EvaluationKernel(KernelParameters params, int fragments) {
7          super(params);
8
9          // Stevci
10         DFEVar counter = control.count.simpleCounter(64);
11         DFEVar n = io.scalarInput("n", UINT_32);
12         DFEVar k = io.input("coefficients", FLOAT);
13         DFEVar x = io.scalarInput("x", FLOAT);
14         DFEVar exp = io.input("exponents", UINT_32);
15
16         // Izracun pow(x, exp)
17         DFEVar powX = constant.var(FLOAT, 1.0);
18         for (int i = 0; i < 10; ++i) {
19                 powX = ((exp > 0) & ((exp & 1) === 1))
20                     ? powX * x
21                     : powX;
22                 exp = exp >> 1;
23                 x = x * x;
24         }
25
26         // Evalvacijska zanka
27         DFEVar loopSum = FLOAT.newInstance(this);
28         DFEVar sum = (counter < fragments) ? 0.0 : loopSum;
29         sum += (k * powX);
30         loopSum <== stream.offset(sum, -fragments);
31
32         // Pisanje na izhod
33         io.output("result", sum, FLOAT, counter >= (n - fragments));
34     }
35 }
```

Koda 3.12: Zanka evalvacija redkega polinoma v več točkah

```
1  // Glavna zanka
2  DFEVar loopX = FLOAT.newInstance(this);
3  DFEVar loopResult = FLOAT.newInstance(this);
4  DFEVar x = kIndex === 0 ? xIn : loopX;
5  DFEVar powX = powX(x, currentExp);
6  DFEVar multiply = powX * currentK;
7  DFEVar currentResult = kIndex === 0
8                       ? multiply
9                       : multiply + loopResult;
10 loopX <== stream.offset(x, -fragmentSize);
11 loopResult <== stream.offset(currentResult, -fragmentSize);
12
13 // Pisanje na izhod
14 io.output("result", currentResult, FLOAT, kIndex === n - 1);
```

3.4 Redki polinomi v več točkah

Tako kot pri gostih polinomih smo tudi za redke implementirali večtočkovno različico. Za osnovo smo uporabili večtočkovno implementacijo gostih polinomov iz razdelka 3.2.

Če želimo večtočkovno implementacijo evalvacije gostih polinomov spremeniti v evalvacijo redkih polinomov, moramo prilagoditi računanje potenc. Uporabili bomo enak algoritem ponavljajočega kvadriranja kot pri redkih polinomih v eni točki iz razdelka 3.3.

Poleg drugačnega načina računanja potenc si moramo v primerjavi z večtočkovno implementacijo gostih polinomov poleg koeficientov polinoma, zapomniti še eksponente polinoma. Vse ostalo lahko ostane enako. Implementacija glavne zanke je podana v kodi 3.12.

Glavno razliko zanke lahko opazimo v vrstici 5. Funkcija *powX* izvede ponavljajoče kvadriranje iz kode 3.10. Vrednost *currentExp* dobimo iz toka eksponentov na enako kot smo dobili *currentK* v kodi 3.4. Ostala koda pa je enaka kot pri implementaciji glavne zanke gostih polinomov v kodi 3.5.

3.5 Paralelizacija algoritmov

Do sedaj smo pisali le algoritme, ki čim bolje izkoristijo tok podatkov. To pomeni, da smo vsak tik podatkovno-pretokovne enote izkoristili za operacije nad podatki. Nismo pa konkretno izkoristili paralelizacije, ki nam jo ponuja podatkovno-pretokovna enota. Zato smo naše algoritme poskušali izboljšati še s paralelizacijo.

Na Maxelerjevi podatkovno-pretokovni enoti lahko paralelizacijo izvedemo na naslednje načine [16]:

1. uporaba vektorjev podatkov znotraj ščepca,
2. podvajanje računskih enot s statičnimi zankami znotraj ščepca,
3. podvajanje ščepcev,
4. uporaba povezave MaxRing več podatkovno-pretokovnih enot.

Mi smo uporabili paralelizacijo z vektorji podatkov in podvajanje računskih enot s statičnimi zankami. Obe paralelizaciji se na nivoju strojne opreme prevedeta v več množilnikov in seštevalnikov. Na ta način lahko naenkrat izvedemo več operacij nad več podatki. Tako se na primer koda 3.13 brez paralelizacije prevede v graf na sliki 3.6.

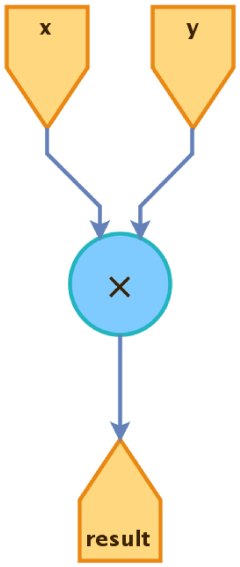
Koda 3.13: Sekvenčno množenje na Maxelerjevi podatkovno-pretokovni enoti

```
1 DFEVar result[i] = x * y;
```

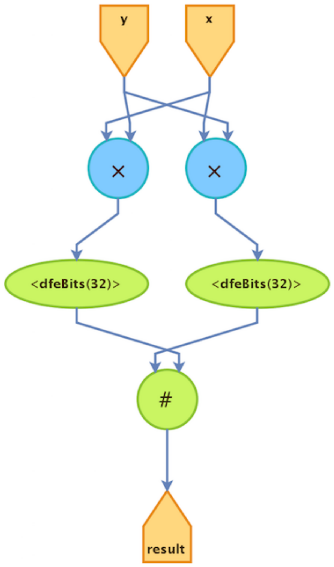
Koda 3.14 pa se s statično zanko prevede v graf, kjer imamo množilnik podvojen, kakor je prikazano na sliki 3.7. Podobno se prevede koda ob uporabi vektorjev podatkov.

Koda 3.14: Paralelno množenje na Maxelerjevi podatkovno-pretokovni enoti

```
1 DFEVar[] result = new DFEVar[2];
2 for (int i = 0; i < 2; i++) {
3     result[i] = x * y;
4 }
```



Slika 3.6: Sekvenčno množenje



Slika 3.7: Paralelno množenje

Z grafa lahko opazimo, da s paralelizacijo ustvarimo več vozlišč grafa, zaradi česar pa porabimo več razpoložljivih virov našega čipa. Tako je zmožnost paraleliziranja operacij navzgor omejena z razpoložljivimi viri čipa. Paziti moramo tudi, da izkoristimo hitrost pretoka podatkov med računalnikom in podatkovno-pretokovno enoto. Ob nezadostni paralelizaciji se lahko zgodi, da je prenos podatkov med računalnikom in podatkovno-pretokovno enoto prepočasen, da bi lahko dobro izkoristili paralelizacijo, kakor je bilo ugotovljeno tudi v članku [2]. Zato je treba naše podatke prenesti v pomnilnik podatko-pretokovne enote ali pa spremeniti vrstni red operacij nad podatki.

Podvajanja ščepcev in povezavo MaxRing v naši implementaciji nismo uporabili. Za dodatne pohitritve bi bila zanimiva uporaba povezave MaxRing. Pri njej lahko namreč naš program dodatno paraleliziramo tako, da se izvaja na več podatkovno-pretokovnih enotah. Kljub temu pa se zaradi dodatne kompleksnosti programa in omejenih možnostih testiranja nismo odločili za ta korak.

Koda 3.15: Paralelizirana evalvacija redkega polinoma v eni točki

```
1  // Sprememba pri branju podatkov
2  DFEVector<DFEVar> k = io.input("constants", floatVec);
3  DFEVector<DFEVar> exp = io.input("exponents", uint16Vec);
4
5  // Spremembe evalvacijske zanke
6  DFEVar loopSum = FLOAT.newInstance(this);
7  DFEVar sum = (counter < fragments) ? 0.0 : loopSum;
8  DFEVar[] summands = new DFEVar[vecSize];
9  for (int i = 0; i < vecSize; i++) {
10         DFEVar powX = powX(x, exp[i]);
11         summands[i] = k[i] * powX;
12 }
13 sum += FloatingPointMultiAdder.add(summands);
14 loopSum <== stream.offset(sum, -fragments);
```

3.5.1 Paralelizacija v eni točki

Implementacijo evalvacije redkih polinomov v eni točki iz razdelka 3.3 smo implementirali z vektorji. Spremembe kode 3.11 so podane v kodi 3.15. Pohitritve enotočkovne evalvacije gostih polinomov se nismo lotili, saj smo predvideli, da za goste polinome ne bomo uspeli prehiteti implementacije na centralni procesni enoti.

Glavna sprememba je, da koeficiente in eksponente beremo kot vektorje iz vhoda, kar prikazujeta vrstici 2 in 3. Zaradi tega tudi drugače evalviramo vsoto členov polinoma. Vsak tik imamo namreč na voljo več koeficientov in eksponentov. S statično zanko paralelno izračunamo rezultat in na koncu seštejemo izračunane člene s funkcijo *FloatingPointMultiAdder.add()* iz knjižnice Maxeler max power [26]. Paralelni izračun je prikazan v vrsticah od 8 do 12, seštevanje členov pa v vrstici 13. Funkcija *powX* izvede ponavljajoče kvadriranje iz kode 3.10. Spremenljivka *vecSize* pa predstavlja velikost vektorjev, oziroma število členov, ki jih naenkrat evalviramo.

Zaradi paralelizacije smo zmanjšali čas izvajanja našega programa. Im-

plementacija enotočkovne evalvacije redkih polinomov iz razdelka 3.3 se je izvajala n tikov, kjer je n število členov polinoma. Vsak tik smo namreč evalvirali samo en člen. Za paralelizirano različico pa je število tikov enako *n / dolžina vektorja*, kar teoretično pomeni, da lahko prvotno implementacijo pohitrimo tolikokrat, kot je dolžina našega vektorja.

3.5.2 Paralelizacija v več točkah

V razdelkih 3.2 in 3.4 smo implementirali evalvaciji v več točkah brez paralelizacije. Naše podatke smo že razdelili na fragmente, kot smo prikazali na sliki 3.2. Sedaj moramo te fragmente samo še paralizirati.

Naš algoritem lahko s statičnimi zankami in vektorji paraleliziramo na tri načine, tako dobimo tri različice:

1. vsak tik evalviramo čim več členov za eno točko (slika 3.8),

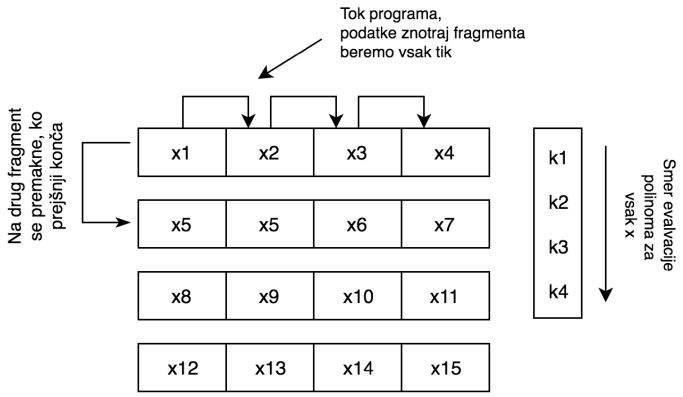
2. vsak tik evalviramo čim več točk za en koeficient (slika 3.9),

3. vsak tik evalviramo več točk za več členov (slika 3.10).

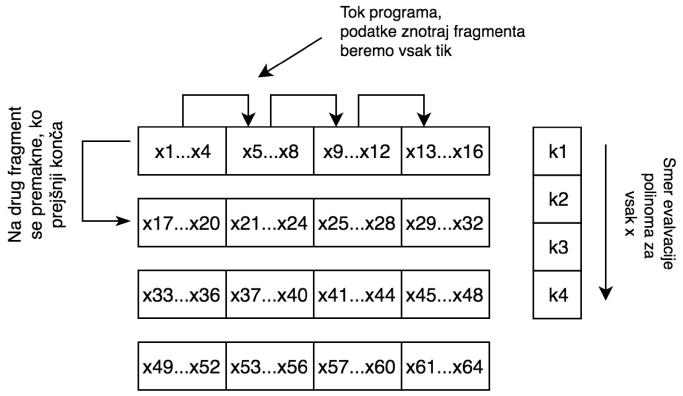
Na slikah 3.8, 3.9 in 3.10 vsak kvadrat vsebuje podatke, ki jih evalviramo naenkrat.

V prvi različici moramo paralelizacijo omejiti na predefinirano dolžino polinoma. To pomeni, da če bi na primer vsak tik naenkrat evalvirali 64 členov, naš polinom pa bi bil dolžine 16, velikega dela čipa sploh ne bi izkoristili. Podobno bi veljalo tudi za vse dolžine polinoma, ki niso večkratnik števila 64. Teoretično je ta razdelitev najbolj optimalna, kadar evalviramo polinome z enakim številom členov, kot jih lahko naenkrat maksimalno evalviramo, in manj optimalna, ko to število ni enako. To pa smo videli kot pomanjkljivost proti drugima različicama.

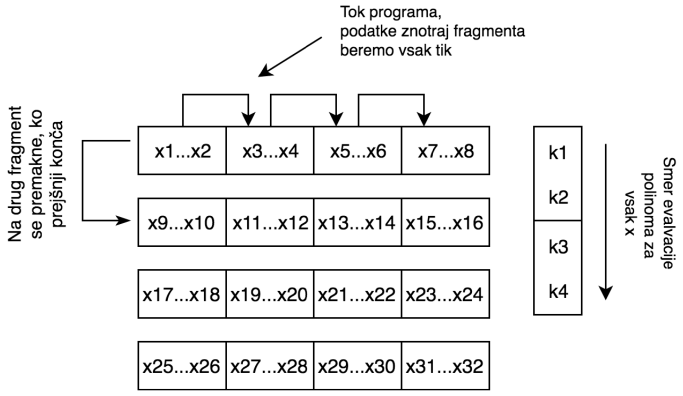
Pred samim testiranjem algoritmov smo predpostavili, da bo tretja različica najbolj optimalna. Pri drugi različici smo namreč na podlagi rezulatov iz [2] sklepali, da bo prišlo do ozkega grla med prenosom toka točk med glavnim



Slika 3.8: Prva različica paralelne evalvacije v več točkah



Slika 3.9: Druga različica paralelne evalvacije v več točkah



Slika 3.10: Tretja različica paralelne evalvacije v več točkah

programom in podatkovno-pretokovno enoto in podatkov ne bomo zmogli prenesti tako hitro, kot jih lahko računamo. Tretja različica bi tako bila nek hibrid med prvo in drugo, ki bi to ozko grlo omilila. Skozi implementacijo smo ugotovili, da je tretja različica kompleksnejša od druge, z vmesnim testiranjem pa se je tudi izkazalo, da ne nudi nobene prednosti.

Zato bomo predstavili drugo različico, ki je najbolj preprosta in s katero smo dosegli tudi najboljše rezultate. Spremembe algoritma za evalvacijo gostih polinomov v več točkah brez paralelizacije iz razdelka 3.2 prikazuje koda 3.16.

Glavna sprememba je, da namesto ene točke x, naenkrat iz vhoda preberemo več točk. Večje število točk predstavimo kot tip *DFEVector*. Ker naenkrat iz vhoda preberemo več točk, posledično naenkrat tudi evalviramo več točk.

Poleg tega lahko opazimo, da smo nekoliko spremenili glavno zanko. V neparalelizirani različici iz razdelka 3.2 smo evalvirali polinom brez uporabe Hornerjevega algoritma. Ker nam Hornerjev algoritem prihrani nekaj množenj, s tem prihranimo tudi na virih čipa, posledično pa lahko povečamo našo paralelizacijo. Kajti večji kot je vektor prebranih točk, več fizičnih operacij bomo imeli na čipu in več virov bo porabil naš algoritem. Zato se hočemo znebiti vsake nepotrebne operacije, kar pa ni bilo potrebno pri neparalelizirani različici.

Uporaba Hornerjevega algoritma pomeni, da moramo evalvirati polinom v obratnem vrstnem redu, tj. od zadnjega člena proti prvemu. Ker smo za večtočkovno evalvacijo evalvirali polinoma z maksimalno dolžino 1024 členov, smo ta korak implementirali v glavnem programu na centralni procesni enoti in obrnjen polinom podali podatkovno-pretokovni enoti. Ker je število točk, v katerih evalviramo, veliko večje od dolžine polinoma, ta korak ne upočasni našega algoritma. Za redke polinome uporaba Hornerjevega algoritma ni smiselna, zato smo v paralelizirani večtočkovni implementaciji za redke polinome ta korak izpustili.

Ker naenkrat beremo več točk vektorsko, mora biti število vhodnih točk

večkratnik dolžine tega vektorja. Če ta pogoj ni izpolnjen, se naš program ne bo izvedel. V takem primeru je najbolj preprosto, da vhodnim podatkom dodamo nekaj točk, s čimer vhod postane večkratnik tega vektorja.

Za paralelizacijo algoritma evalvacije redkih polinomov, bi enako spremenili branje podatkov, točke bi brali kot vektorje, za glavno zanko pa bi uporabili kodo 3.12, kjer bi zamenljali tip spremenljivk iz *DFEVar* v *DFEVector*.

Neparalelizirani implementaciji evalvacije v več točkah iz razdelkov 3.2 in 3.4 zahtevata $n \cdot m$ tikov evalvacije, kjer je n dolžina polinoma, m pa število točk, ki jih evalviramo, medtem ko podani paralelizirani rešitvi zahtevata $n \cdot m$ / *dolžina vektorja* tikov. Tako lahko naš algoritem teoretično pospešimo tolikokrat, kolikorkrat lahko naenkrat preberemo število točk.

Koda 3.16: Paralelizirana evalvacija gostega polinoma v več točkah

```
1  // Sprememba pri branju podatkov
2  DFEVectorType<DFEVar> floatVec = new DFEVectorType<>(FLOAT, xsPerTick);
3  DFEVector<DFEVar> xIn = io.input("xs", floatVec, kIndex === 0);
4
5  // Glavna zanka evalvacije
6  DFEVector<DFEVar> loopX = floatVec.newInstance(this);
7  DFEVector<DFEVar> loopResult = floatVec.newInstance(this);
8  DFEVector<DFEVar> x = kIndex === 0 ? xIn : loopX;
9  DFEVector<DFEVar> multiply = kIndex === 0
10                 ? constant.vect(xsPerTick, FLOAT, 0.0)
11                 : x * loopResult;
12 DFEVector<DFEVar> result = multiply + k;
13 loopX <== stream.offset(x, -fragmentSize);
14 loopResult <== stream.offset(result, -fragmentSize);
15
16 // Pisanje na izhod
17 io.output("result", result, floatVec, kIndex === n - 1);
```

Poglavje 4

Uporaba algoritmov

V tem razdelku bomo opisali nekatere razširitve in prilagoditve naših algoritmov tudi na druge probleme, ki niso direktno polinomska evalvacija.

4.1 Evalvacija kompleksnih polinomov

Nekatere naše algoritme smo hoteli razširiti tudi na evalvacijo v kompleksnem obsegu, natančneje, ko so koeficienti in točke lahko kompleksna števila. Kompleksno število v računalništvu predstavimo kot par realnega in kompleksnega dela. Tako bi število

$$5 + 6i$$

predstavili kot par $(5, 6)$. Realni del tega kompleksnega števila je $Re(5+6i) = 5$, imaginarni del pa število $Im(5 + 6i) = 6$.

Za izvedbo seštevanja in množenja v kompleksnem obsegu moramo izvesti več operacij kot v realnem. Tako za seštevanje dveh kompleksnih števil, ki je definirano v [27] kot:

$$(a + bi) + (c + di) = (a + c) + (b + d)i,$$

potrebujemo dve operaciji realnega seštevanja namesto ene. V primeru množenja, ki je definirano v [28] kot:

$$(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i,$$

pa štiri realna množenja, eno realno seštevanje in eno realno odštevanje. Kadar sta operaciji realnega seštevanja in odštevanja hitrejši od operacije realnega množenja, pa lahko kompleksna števila množimo samo s tremi realnimi množenji. Tako lahko izračunamo realni del, kar je navedeno v [28] kot:

$$Re((a + bi) \cdot (c + di)) = ac - bd,$$

imaginarni del pa izračunamo kot:

$$Im((a + bi) \cdot (c + di)) = (a + b) \cdot (c + d) - ac - bd,$$

kjer vrednosti ac in bd izračunamo samo pri realnem delu, pri kompleksnem delu pa jih samo ponovno uporabimo. Tako potrebujemo tri realna množenja, dve realni seštevanji in tri realna odštevanja. Računanje s kompleksnimi števili je tako v primerjavi z računanjem v realnem računsko bolj zahtevno, saj za izračun porabimo več množenj in seštevanj. Poleg tega pa je tudi prostorsko bolj zahtevno; n kompleksnih števil namreč zavzema toliko prostora kot $2n$ realnih števil.

Maxelerjeva podatkovno-pretokovna enota že ima implementirano podporo za operacije nad kompleksnimi števili, zato nam samih operacij ni treba implementirati. Moramo pa spremeniti tip naših spremenljivk iz *DFEVar* v *DFEComplex*. S tem prevajalniku povemo, da mora uporabiti kompleksne operacije množenja in seštevanja. Primer uporabe kompleksnih tipov je predstavljen v kodi 4.1, kjer je prikazano preprosto branje kompleksnih števil iz vhoda.

Koda 4.1: Branje kompleksnih števil

```
1 DFEComplexType COMPLEX_FLOAT = new DFEComplexType(dfeFloat(8, 24));
2 DFEComplex x = io.input("xs", COMPLEX_FLOAT);
```

Prav tako moramo prilagoditi tudi vhodne podatke. Vhodna kompleksna števila podamo programu kot seznam realnih števil, kjer se izmenjujejo realne in imaginarne komponente. Če bi evalvirali števila $5 + 6i$, $7 + 4i$, $9 + 2i$, bi na vhod podali seznam: $5, 6, 7, 4, 9, 2$.

4.2 Gručenje točk

Eden od algoritmov za gručenje točk je Lloydov algoritem [29]. Ta uspešno rešuje problem gručenja m točk v n skupin oz. gruč (angl. je to k-means clustering problem). Lloydov algoritem je iterativen algoritem, ki ima naslednje korake [30]:

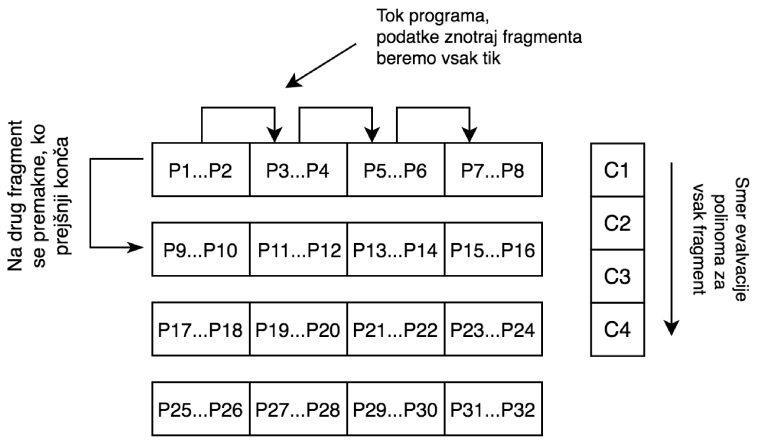
1. naključno izberi n točk, ki jim bomo rekli centri gruč,
2. za vsako točko izračunaj razdaljo do vsakega centra,
3. za vsako točko izberi najbližji center in jo dodeli v njegovo gručo,
4. za vsako gručo izračunaj nov center kot srednjo vrednost vseh točk gruče,
5. ponavljaj korake od 2 do 4 dokler nobena točka ne zamenja gruče ali dokler ne naredimo točno določenega števila iteracij.

Mi smo z DFE pospešili koraka 2 in 3. V nadaljevanju bomo ta dva koraka imenovali *gručenje*. V našem algoritmu bomo namesto razdalje, ki se računa v drugem koraku, uporabljali njen kvadrat. Pri tem z izrazom *razdalja* označujemo *evklidsko razdaljo*.

V razdelku 1.5 smo omenili, da si lahko kvadrat razdalje zamislimo kot poseben primer polinoma. Če fiksiramo eno točko, to postane polinom d spremenljivk, kjer d predstavlja dimenzijo točk. Kot primer je naveden izračun razdalje med točko in centrom v dveh dimenzijah, kjer smo vrednosti centra fiksirali:

$$p(x_1, x_2) = (x_1 - c_1)^2 + (x_2 - c_2)^2.$$

Gručenje tako lahko prevedemo na večtočkovno evalvacijo polinoma z nekaj spremembami. Tok podatkov in programa za gručenje točk je podan na sliki 4.1 in je zelo podoben toku programa paralalelne večtočkovne evalvacije polinoma iz razdelka 3.5.2 in slike 3.9. Na sliki 4.1 oznaka P predstavlja točke, C pa centre.



Slika 4.1: Tok podatkov in programa pri gručenju točk

Glavna razlika med večtočkovno evalvacijo polinoma in gručenjem je v predstavitvi podatkov. V primeru gručenja so točke in centri predstavljeni z več spremenljivkami oz. dimenzijami, medtem ko smo pri večtočkovni evalvaciji polinoma imeli samo točke v eni dimenziji.

Sam potek algoritma za gručenje za dano fragmentacijo na sliki 4.1 pa se izvaja takole: paralelno evalviraj razdaljo med točko *P1* in centrom *C1* ter točko *P2* in centrom *C1*. Potem paralelno evalviraj razdaljo med točko *P3* in *C1*, ter točko *P4* in centrom *C1*, in na ta način do konca fragmenta. Nato paralelno evalviraj razdaljo med točko *P1* in centrom *C2*, ter točko *P2* in centrom *C2*. Dodatno posebej za vsako točko *P1* in *P2* primerjaj razdalji do centrov *C1* in *C2* ter izberi najkrajšo. Enako nato evalviraj tudi vse ostale točke znotraj fragmenta. To ponavljaj dokler ne prideš do konca vseh centrov. Takrat na izhod za vsako točko zapiši najbližji center. Enak postopek nato ponavljaj še za vse ostale fragmente.

4.2.1 Implementacija

Za osnovo smo uporabili implementacijo večtočkovne evalvacije iz kode 3.16. Spremembe implementacije so podane v kodi 4.2. Nekoliko smo prilagodili poimenovanje programskih spremenljivk. V kodi 3.16 imamo *xIn* kot vhod

x-ov, tukaj je *pIn* kot vhod točk. V kodi 3.16 je k kot koeficient, tukaj uporabljamo c kot center. V kodi 3.16 imamo *kIndex* kot indeks trenutnega koeficienta, tukaj imamo *cIndex* kot indeks trenutnega centra.

Koda 4.2: Glavna zanka gručenja točk

```
1  // Sprememba branja podatkov.
2  // Tip Point predstavlja DFEVector<DFEVar> z velikostjo D
3  DFEVectorType<DFEVar> pType = new DFEVectorType<>(FLOAT, D);
4  DFEVectorType<Point> pVec = new DFEVectorType<>(pType, ptsPerTick);
5  DFEVector<Point> pIn = io.input("points", pVec, cIndex === 0);
6  Point c = cMemory.read(cIndex.cast(addressType));
7
8  // Glavna zanka
9  DFEStruct closestCenters = structType.newInstance(this);
10 DFEVector<Point> loopPoints = pointVec.newInstance(this);
11 DFEVector<Point> points = cIndex === 0 ? pIn : loopPoints;
12 DFEVector<Point> difference = points - c;
13 DFEVector<Point> sqrs = difference * difference;
14 List<Point> sqrsList = rowsToColumns(sqrs).getElementsAsList();
15 DFEVector<DFEVar> sqDists = FloatingPointMultiAdder.add(sqrsList);
16 DFEStruct closest = findClosest(closestCenters, sqDists, cIndex);
17 loopPoints <== stream.offset(points, -fragmentSize);
18 closestCenters <== stream.offset(closest, -fragmentSize);
19
20 // Izhod
21 DFEVector<DFEVar> closestIndexes = closest.get(CENTER_KEY);
22 io.output("result", closestIndexes, uint64Vec, cIndex === n - 1);
```

Ker imamo pri gručenju točke v več dimenzijah, bomo točke predstavili kot vektor *DFEVector<DFEVar>*. V zgornji kodi smo zaradi poenostavitve kode ta zapis zamenjali z zapisom *Point*. Definiranje tipa točke je predstavljeno v vrstici 3, kjer D predstavlja število spremenljivk oziroma dimenzijo. Tip vektorja več točk pa je definiran v vrstici 4.

V vrsticah od 8 do 18 je nato podana glavna zanka. V spremenljivki *closestCenters* nosimo informacijo o indeksih najbližjih centrov in njihov razdalj do točk. V spremenljivki *loopPoints* prenašamo točke, za katere trenutno

računamo razdalje.

Logika glavne zanke je nekoliko spremenjena v primerjavi z glavno zanko iz implementacije večtočkovne evalvacije polinoma iz 3.16. Razlikuje se pri izračunu. Tu za točke izračunamo kvadrat razdalje od trenutnega centra, medtem ko smo pri večtočkovni evalvaciji izračunali zmnožek potence in koeficienta. Izračun kvadrata razdalje se izvede v vrsticah od 12 do 15, nato pa v vrstici 16 poiščemo najbližji center s pomočjo funkcije *findClosest()*; njena koda je podana v 4.3. Na koncu v vrstici 22 na izhod po točkah vrnemo najbližje indekse centrov.

Koda 4.3: Izračun najbližjih centrov za trenutne točke

```
1  private DFEStruct findClosest(DFEStruct oldDistance,
2      DFEVector<DFEVar> newSqDist, DFEVar cIndex) {
3      DFEVar[] updatedSqDist = new DFEVar[ptsPerTick];
4      DFEVar[] updatedCIndexes = new DFEVar[ptsPerTick];
5      DFEVector<DFEVar> oldSqDist = oldDistance.get(SQR_KEY);
6      DFEVector<DFEVar> oldCIndexes = oldDistance.get(CENTER_KEY);
7      DFEVar isFirstCenter = cIndex === 0;
8      for (int i = 0; i < ptsPerTick; i++) {
9          DFEVar isCloser = isFirstCenter | newSqDist[i] < oldSqDist[i];
10         updatedSqDist[i] = isCloser ? newSqDist[i] : oldSqDist[i];
11         updatedCIndexes[i] = isCloser ? cIndex : oldCIndexes[i];
12     }
13     return buildStruct(updatedCIndexes, updatedSqDist);
14 }
```

4.3 Diskretna Fourierova transformacija

Diskretna Fourierova transformacija (krajše DFT) je poseben primer Fourierove transformacije [31] in je pomembno orodje v analizi signalov. Za izračun diskretne Fourierove transformacije obstaja učinkovit algoritem, imenovan Fast Fourier transform (krajše FFT) [32]. FFT je že implementiran za Maxelerjevo podatkovno-pretokovno enoto v [26], implementacija pa je opisana tudi v [17].

Mi se bomo lotili naivne implementacije DFT ter jo primerjali z naivno ukazno-pretokovno implementacijo DFT in optimizirano ukazno-pretokovno implementacijo FFT knjižnice FFTW [33]. Algoritem bi se lahko uporabil za primere, kjer nam FFT ne bi zadoščal. Tak primer bi bil, če bi želeli računati DFT točno določene dolžine, ki ne bi bila enaka 2^k za neko naravno število k. Namen implementacije pa je tudi testiranje zmogljivosti podatkovno-pretokovne enote proti namenski programski opremi za računanje, kakršna je FFTW [33].

Algoritem bi bilo enostavno razširiti na druge transformacije, sorodne Fourierovi, kot je diskretna kosinusna transformacija (krajše DCT), za katero že obstaja implementacija na Maxelerjevi podatkovno-pretokovni enoti za dvodimenzionalne slike [34]. Morda bi lahko algoritem prilagodili tudi za namene drseče DFT opisane v [35].

DFT po definiciji lahko zapišemo kot:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}}.$$

Pri tem to lahko prevedemo na polinome tako, da si mislimo, da so x_n koeficienti, $e^{\frac{-i2\pi kn}{N}}$ pa je točka, v kateri računamo polinom.

Ideja našega algoritma je, da vsak tik za nek k evalviramo naenkrat čim več točk x_n. Tako na primer v prvem tiku evalviramo čim več točk za $k = 0$ kot:

$$X_0 = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi 0}{N}},$$

v drugem tiku evalviramo čim več točk za $k = 1$ kot:

$$X_1 = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi n}{N}}$$

ipd. Pri tem si lahko potence $e^{\frac{-i2\pi kn}{N}}$ preračunamo vnaprej in shranimo v pomnilnik. Do njih pa dostopamo z odmikom, ki ga dobimo kot zmnožek $k * n$ po modulu N. Tako je $e^{\frac{-i2\pi 0}{N}}$ dostopen na indeksu 0, $e^{\frac{-i2\pi 1}{N}}$ na indeksu 1, $e^{\frac{-i2\pi 2}{N}}$ na indeksu 2 itd.

Za večje DFT jasno ni mogoče naenkrat evalvirati vseh točk, zato je treba te evalvacije, podobno kot v razdelku 3.2, razbiti na fragmente. Tako na primer v prvem tiku evalviramo samo prvih P točk za $k = 0$, kjer je P število točk, ki smo jih zmožni paralelizirati. V naslednjem tiku evalviramo prvih P točk za $k = 1$ itd. vse do konca fragmenta. Šele nato evalviramo naslednjih P točk za $k = 0$.

4.3.1 Implementacija

Za osnovo implementacije DFT smo vzeli implementacijo osnovne večtočkovne evalvacije, ki je podana v kodi 3.5. Glavna zanka DFT je podana v kodi 4.4. Za lažje razumevanje pa jo je najbolje primerjati s kodo 3.5 osnovne večtočkovne evalvacije.

Kot smo že omenili, lahko pri DFT vnaprej poračunamo potence $e^{\frac{-i2\pi kn}{N}}$, jih shranimo v pomnilnik in ob računanju beremo iz pomnilnika. Dodatno iz pomnilnika beremo tudi vse točke. V vrstici 3 tako preberemo vektor x-ov za naš DFT iz pomnilnika. Nato v vrstici 4 preberemo vse ustrezne potence števila $e^{\frac{-i2\pi kn}{N}}$ iz pomnilnika s pomočjo metode *pows()* glede na trenutni k in n. Na koncu pomnožimo x-e s potencami $e^{\frac{-i2\pi kn}{N}}$ in člene skupaj seštejemo v vrsticah 5 in 6.

V kodi števec i predstavlja trenutne i-te P točke, ki smo jih zmožni evalvirati paralelno za nek k. Število vseh iteracij P točk, ki jih moramo evalvirati, pa je enako is. Ko smo v zadnji iteraciji P točk oziroma na koraku $is - 1$, rezultat za k-to točko zapišemo na izhod, kar je prikazano v vrstici 14.

V razdelku 3.5.2 smo govorili o treh različicah paralelizacije polinoma. Implementacija DFT pravzaprav predstavlja evalvacijo čimvečjega števila koeficientov za točko oziroma prvo različico paralelizacije, predstavljeno na sliki 3.8. Pri tem so naši koeficienti točke x_n, točka, v kateri evalviramo, pa $e^{\frac{-i2\pi kn}{N}}$. Za tako implementacijo smo se odločili predvsem zaradi enostavnosti. Posledično naš algoritem najbolje deluje, ko je dolžina DFTja večkratnik števila koeficientov, ki jih naenkrat evalviramo.

Koda 4.4: Glavna zanka evalvacije DFT

```
1  // Glavna zanka
2  DFEComplex loopResult = CPLX_FLOAT.newInstance(this);
3  DFEVector<DFEComplex> x = xBuf.read(xsAddress);
4  DFEVector<DFEComplex> powE = getPows(params);
5  DFEVector<DFEComplex> multiply = powE * x;
6  DFEComplex sum = FloatingPointMultiAdder.add(multiply.getElsAsList());
7  DFEComplex result = (i === 0)
8                      ? sum
9                      : sum + loopResult;
10 loopResult <== stream.offset(result, -fragmentSize);
11
12 // Pisanje na izhod
13 DFEVar out = (i === is - 1);
14 io.output("result", result, CPLX_FLOAT, out);
```

Za primerjavo z ukazno-pretokovno implementacijo in s knjižnico FFTW smo našo implementacijo izvedli tako, da ji lahko podamo večje število DFT za izračun. Izračun ene DFT zaradi začetne zakasnitve podatkovno-pretokovne enote ne bi bil konkurenčen.

Našo kodo DFT bi v DCT enostavno spremenili tako, da bi spremenili vrstico 4 in brali vnaprej izračunane kosinuse namesto potenc števila e. Dodatno bi pri implementaciji DCT namesto kompleksnih lahko uporabili tudi realna števila in s tem prihranili na virih čipa. Enak postopek bi seveda lahko uporabili tudi za diskretno sinusno transformacijo.

Poglavje 5

Analiza rezultatov

Naše algoritme smo primerjali z ustreznimi ukazno-pretokovnimi implementacijami na centralni procesni enoti v jeziku C, ki smo jih prevajali s prevajalnikom *gcc* in optimizacijsko ravnjo *O3*. Kritično smo ovrednotili naše algoritme in navedli zaključke o primernosti prenosa na podatkovno-pretokovno enoto. Naše algoritme smo izvajali na delovni postaji s specifikacijo, podano v tabeli 5.1.

Pri tem smo merili realni čas izvajanja na različno velikih podatkih in porabo čipa. Vrednosti porabe čipa smo dobili iz poročila o porabi virov, ki ga zgenerira prevajalnik Maxeler. Poročilo vsebuje elemente, predstavljene v tabeli 5.2.

Poročilo o porabi virov vsebuje odstotek porabe elementov čipa in je pri implementaciji ključnega pomena, saj smo prek porabe lahko sklepali,

CPE	Intel(R) Core(TM) i7-6700K CPU @ 4.00 GHz
RAM	4 DIMM enot, vsaka velikosti 16384 MB - skupaj 64 GB
DFE	DFE kartica Vectis - 297600 preslikovnih tabel (LUT), 2 x 297600 flip-flopov, 2016 signalnih procesorjev (DSP), 2128 celic po 18 kbitov BRAM-a

Tabela 5.1: Specifikacije delovne postaje, na kateri smo izvajali algoritme

Oznaka	Opis
LUTs	preslikovalne tabele (angl. look-up tables)
FF1	primarni flip-flopi (angl. primary flip-flops)
FF2	sekundarni flip-flopi (angl. secondary flip-flops)
DSP	signalni procesorji oz. množilniki, ki so uporabljeni ob množenju
BRAM	blokovni pomnilnik za daljše hranjenje vrednosti

Tabela 5.2: Elementi čipa Maxelerjeve podatkovno-pretokovne enote

koliko operacij še lahko dodamo na čip in katera implementacija ni dovolj učinkovita. Pogosto se naša implementacija ni uspešno prevedla, saj je bila preobsežna za prevedbo na čip. Poročilo o porabi virov pa nam je pomagalo prilagoditi algoritem, da se je ta uspešno prevedel in izvedel.

5.1 Redki polinomi v eni točki

V primeru gostih polinomov nismo preizkusili algoritma v eni točki. Za redke polinome v eni točki pa smo videli priložnost za pohitritve v zelo dolgih polinomih. Pri tem so vhodni polinomi imeli več milijonov členov, katerim so se eksponenti lahko ponavljali. Najprej smo implementirali in preizkusili algoritme za evalvacijo realnih polinomov. Ker rezultati niso bili obetavni, se testiranja kompleksne rešitve nismo lotili. Tako smo preizkusili algoritme, podane v tabeli 5.3.

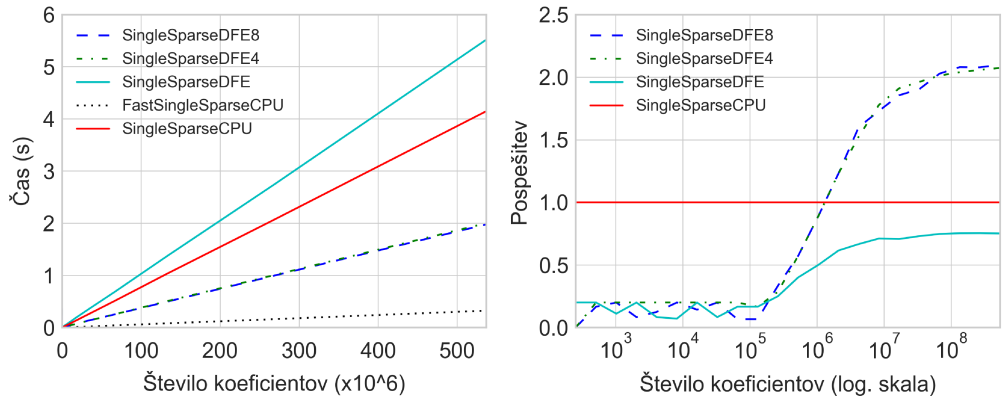
5.1.1 Rezultati za realne polinome

Na sliki 5.1 so predstavljeni časi in pospešitve. Opazimo lahko, da smo z algoritmi na DFE hitrejši od prehiteli ekvivalentno ukazno-pretokovno implementacijo *SingleSparseCPU*.

Vidimo, da sta paralelni različici podatkovno-pretokovnega algoritma hitrejši od ukazno-pretokovne rešitve, ko je število koeficientov večje od 10^6.

Algoritem	Opis
SingleSparseCPU	Ukazno-pretokovna implementacija, ki je ekvivalentna DFE algoritmu, podanem v razdelku 3.3.
FastSingleSparseCPU	Ukazno-pretokovna implementacija, kjer najprej združimo koeficiente z isto potenco in nato evalviramo s Hornerjevim algoritmom.
SingleSparseDFE	Naš algoritem brez paralelizacije na DFE, iz razdelka 3.3.
SingleSparseDFE4	Naš algoritem s paralelizacijo z vektorji, kjer smo naenkrat evalvirali štiri koeficiente polinoma na DFE. Paralelizacija je opisana v razdelku 3.5.1.
SingleSparseDFE8	Naš algoritem s paralelizacijo z vektorji, kjer smo naenkrat evalvirali osem koeficientov polinoma na DFE. Paralelizacija je opisana v razdelku 3.5.1.

Tabela 5.3: Preizkušeni algoritmi za redke polinome v eni točki



Slika 5.1: Časi (levo) in pospešitve (desno) za redke polinome v eni točki

Poleg tega opazimo, da s paralelizacijo nekaj pridobimo v primerjavi z neparaleliziranou različico DFE. Vendar pa med obema paraleliziranima različicama ni razlik v hitrosti izvajanja, kar pomeni, da se podatki prepočasi pretakajo iz računalnika na podatkovno-pretokovno enoto.

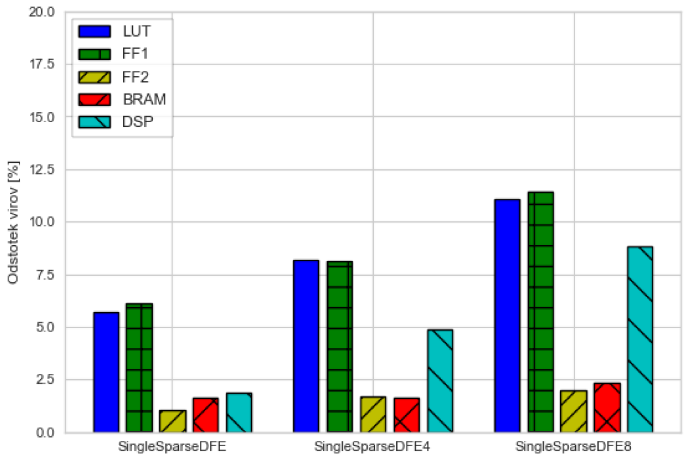
Tako smo iz hitrosti evalvacije *SingleSparseDFE4* in *SingleSparseDFE8* preračunali, da se na podatkovno-pretokovno enoto vsako sekundo prenese približno 2 GB podatkov. Če namreč pogledamo podatke pri 500 milijonih koeficientov, je čas izvajanja približno 2 sekundi. 500 milijonov koeficientov pomeni približno 2 GB podatkov v enojni natančnosti. S tem da imamo še 2 GB eksponentov, kajti pri tej evalvaciji algoritmu podajamo tudi eksponente, za katere smo uporabili tip *uint32_t*, ki zavzame 4 bajte. Iz tega sledi, da se 4 GB podatkov prenese v dveh sekundah oziroma 2 GB v eni, kar je zgorna meja prenosa PCIe, ki jo je uporabljala naša testna delovna postaja. Za večje pospešitve bi lahko optimizirali velikost podatkov eksponentov in uporabili tip, ki zavzame manj bajtov, na primer *uint16_t*. Za dodatne pospešitve pa bi tako potrebovali hitrejši prenos PCIe.

Na ukazno-pretokovni enoti smo uspeli implementirati hitrejši algoritem *FastSingleSparseCPU*. Ta algoritem deluje tako, da redek polinom pretvori v gost polinom, saj koeficiente združi za vsak eksponent posebej, nato pa izvede Hornerjev algoritem.

FastSingleSparseCPU je okoli šestkrat hitrejši od podatkovno-pretokovnih algoritmov. Poleg tega tako dolgih polinomov v realnosti ne srečamo, zato smo zaključili, da podatkovno-pretokovna enota ni primerna za evalvacijo polinomov v eni točki, tudi če bi naprej optimizirali naš algoritem.

5.1.2 Poraba virov čipa

Tabela 5.4 in slika 5.2 prikazujeta, da je poraba virov izredno nizka in je še veliko prostora za druge operacije. Vendar zaradi nizke prepustnosti podatkov od računalnika do podatkovno-pretokovne enote večja paralelizacija algoritma ni smiselna. Zaradi nizke porabe pa bi algoritem lahko uporabili kot del nekega drugega algoritma.



Slika 5.2: Poraba čipa za algoritme enotočkovno evalvacije redkih polinomov

Algoritem	LUT	FF1	FF2	BRAM	DSP
SingleSparseDFE	5,70	6,09	1,02	1,64	1,88
SingleSparseDFE4	8,17	8,14	1,68	1,60	4,86
SingleSparseDFE8	11,10	11,40	1,98	2,35	8,83

Tabela 5.4: Odstotki porabe čipa FPGA za implementacijo evalvacije redkih polinomov v eni točki

5.2 Gosti polinomi v več točkah

Za preizkus implementacij evalvacije gostih polinomov smo vzeli testne polinome dolžin od 16 do 1024, in sicer 16, 32, 128 in 1024. Za število točk smo vzeli vrednosti od 16 do 67108864, kjer smo vmesna števila dobili tako, da smo prejšno vrednost množili z 2.

Ovrednoteni algoritmi so podani v tabeli 5.5.

Algoritem	Opis
MultiDenseRealCPU	Ukazno-pretokovni Hornerjev algoritem v programskem jeziku C
MultiDenseRealDFE	Naš algoritem iz razdelka 3.2 brez paralelizacije na DFE
MultiDenseRealDFE64	Naš algoritem s paralelizacijo z vektorji iz razdelka 3.5.2, kjer smo naenkrat evalvirali 64 točk na DFE
MultiDenseRealDFE128	Naš algoritem s paralelizacijo z vektorji iz razdelka 3.5.2, kjer smo naenkrat evalvirali 128 točk na DFE
MultiDenseComplexCPU	Ukazno-pretokovni algoritem v programskem jeziku C, ki uporablja kompleksna števila
MultiDenseComplexDFE	Naš algoritem iz razdelka 3.2 brez paralelizacije, ki uporablja kompleksna števila
MultiDenseComplexDFE64	Naš algoritem s paralelizacijo z vektorji iz razdelka 3.5.2, kjer smo naenkrat evalvirali 64 točk, ki uporablja kompleksna števila

Tabela 5.5: Preizkušeni algoritmi za goste polinome v več točkah

5.2.1 Rezultati za realne polinome

Slika 5.3 prikazuje graf časa izvajanja algoritmov za realne polinome. Ker so časi implementacije brez paralelizacije dosegali veliko slabše čase in so zmanjšali preglednost naših grafov, smo implementacijo brez paralelizacije odstranili z grafa časov. Slika 5.4 nato prikazuje pospešitve naših algoritmov.

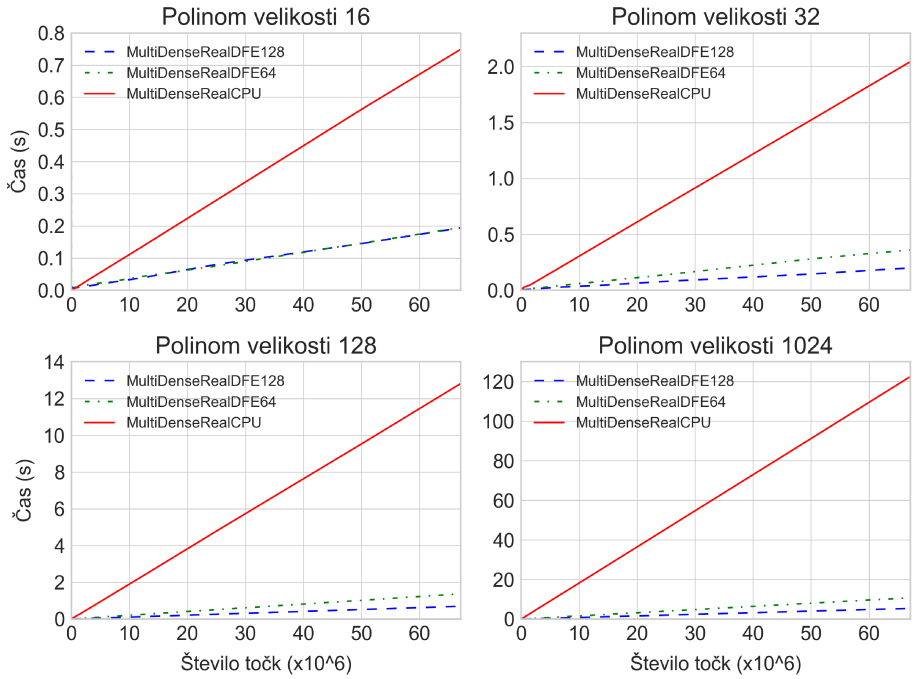
Vidimo, da so paralelizirani algoritmi na DFE hitrejši od ukazno-pretokovne implementacije na centralni procesni enoti. S slike pospešitev 5.4 vidimo, da to ne velja za neparalelizirano različico, saj je ta okoli petkrat počasnejša od ukazno-pretokovne različice.

Opazimo tudi, da sta pri velikosti polinoma do stopnje 16 obe paralelizirani različici enako hitri. Predpostavimo lahko, da predstavlja povezava med računalnikom in podatkovno-pretokovno enoto ozko grlo in se točke prepočasi pretočijo na podatkovno-pretokovno enoto, zaradi česar ovirajo samo evalvacijo.

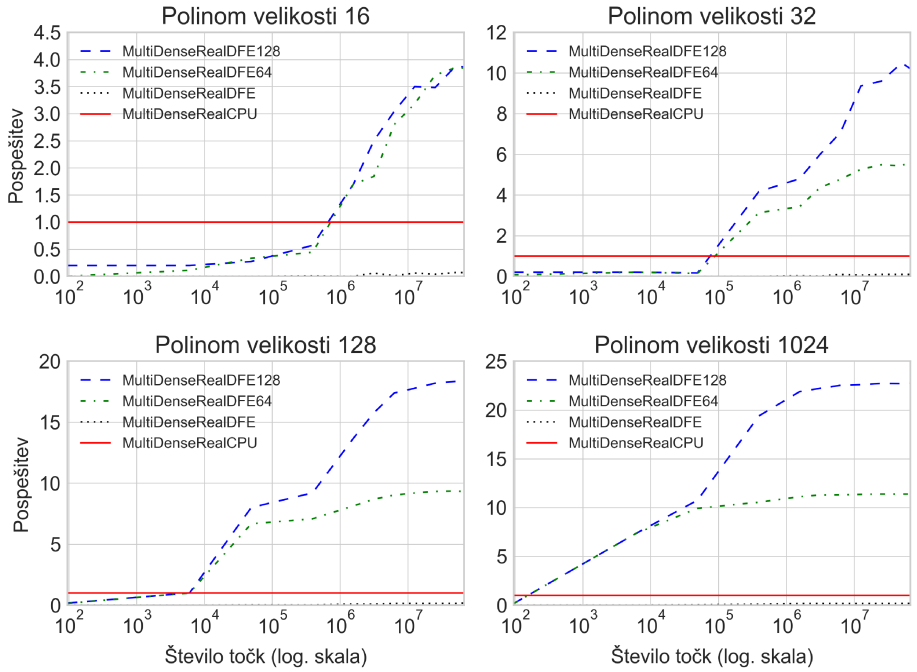
Ob večjih polinomih zaradi daljše evalvacije polinoma teh težav ni več. Tako evalvacija, kjer naenkrat evalviramo 128 točk, počasi pridobiva z velikostjo polinoma proti tisti, kjer evalviramo samo 64 točk.

Poleg tega lahko iz naklona opazimo, da ob daljših polinomih dobimo tudi večjo pospešitev v primerjavi z ukazno-pretokovno implementacijo. To potrjuje tudi graf pospešitev na sliki 5.4. Za nizke stopnje polinomov imamo samo okoli štirikratne pospešitve, za večje pa več kot dvajsetkratne. Tako lahko sklepamo, da je podatkovno-pretokovna evalvacija hitrejša od ukazno-pretokovne evalvacije, vendar je omejena s povezavo med računalnikom in podatkovno-pretokovno enoto.

Na podlagi rezultatov lahko sklepamo, da je DFE primerna za evalvacijo gostih realnih polinomov v več točkah, najbolj pa se izplača pri dolgih polinomih.



Slika 5.3: Časi paraleliziranih algoritmov za goste polinome



Slika 5.4: Pospešitve paraleliziranih algoritmov za goste polinome

5.2.2 Rezultati za kompleksne polinome

Slika 5.5 prikazuje čase evalvacije kompleksnih polinomov. Podobno kot pri evalvaciji v realnem tudi tukaj ne dobimo pospešitev pri neparalelizirani različici. Zaradi predolgega časa izvajanja se evalvacija pri polinomih, ki so večji od 32 celo ni končala pravočasno pri več točkah. Za podatkovno-pretokovno enoto je bil namreč čas izvajanja enega testnega primera omejen na 60 sekund. Zaradi tega pri večjih evalvacijah tudi nismo pridobili vseh podatkov za neparalelizirani različici.

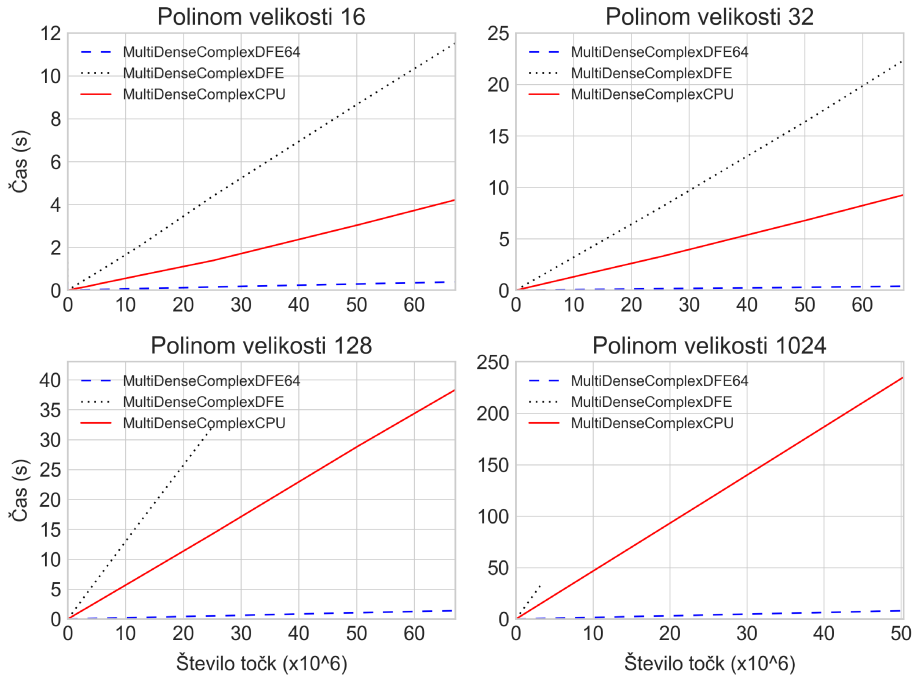
S slike 5.6 lahko razberemo, da je osnovna implementacija počasnejša od ukazno-pretokovne implementacije na centralni procesni enoti, in sicer približno trikrat. Pri paralelni različici pa dosežemo od 10 do skoraj 30-kratne pospešitve, odvisno od velikosti polinoma.

Tako kot v realnem lahko zopet opazimo, da ob večjih polinomih dobimo večje pospešitve algoritma. Vzrok je tudi tokrat ozko grlo med računalnikom in podatkovno-pretokovno enoto. Obenem moramo vedeti tudi, da kompleksna števila zavzamejo dvakrat več prostora, zato je to ozko grlo še izrazitejše. Toda počasnejša evalvacija kompleksnih števil ukazno-pretokovne implementacije odtehta počasnejši pretok podatkov.

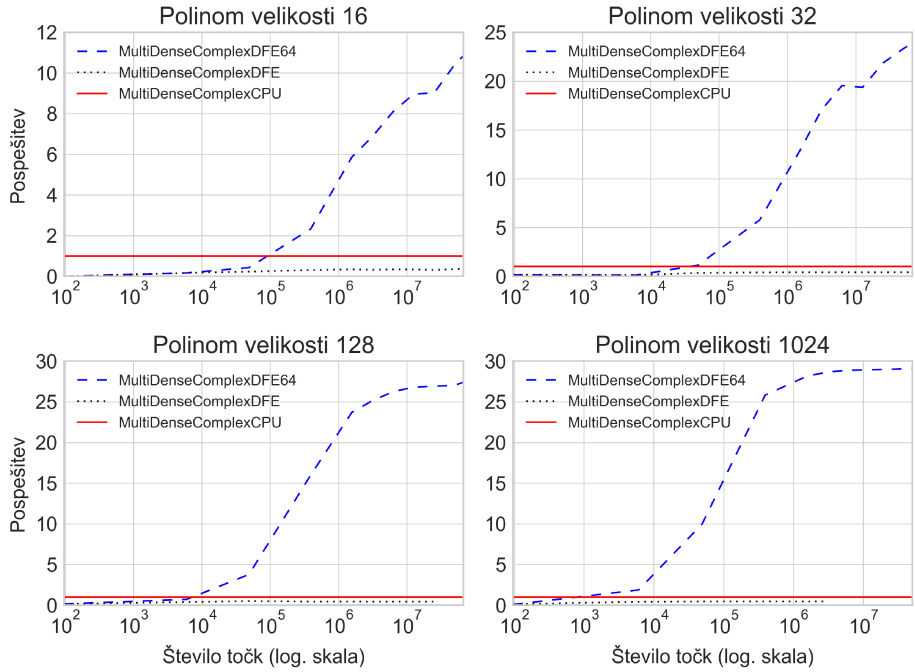
5.2.3 Poraba virov čipa

V tabeli 5.6 in na sliki 5.7 je prikazana poraba čipa, ki po pričakovanju raste z velikostjo paralelizacije. Naše implementacije so najbolj intenzivne za primarne flip-flope, ki so v tabeli označeni s FF1. Ti tudi najbolj omejujejo nadaljnje možnosti pospešitev. Največja implementacija za realne polinome *MultiDenseRealDFE128* tako porabi okoli 45 % primarnih flip-flopov, medtem ko paralelna različica za kompleksne polinome *MultiDenseRealDFE64* porabi okoli 60 % primarnih flip-flopov.

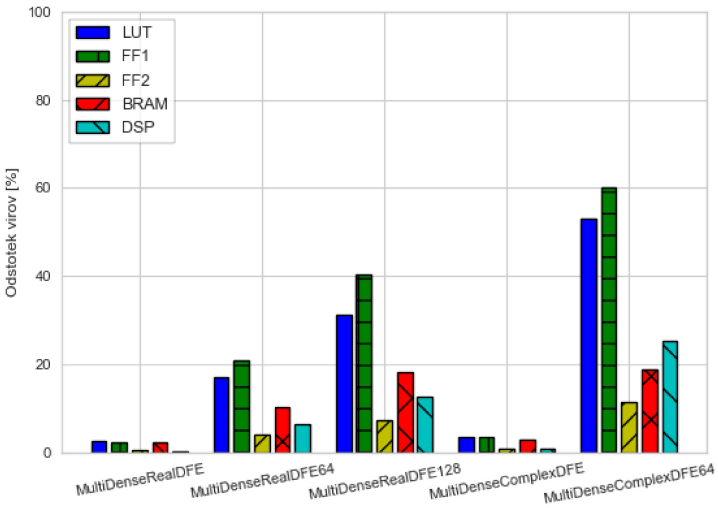
Sklepamo lahko, da bi velikost paralelizacije naših algoritmov lahko še povečali. Tako bi na primer lahko algoritem *MultiDenseRealDFE128* za realne polinome še enkrat povečali in evalvirali 256 točk naenkrat.



Slika 5.5: Časi algoritmov za kompleksne goste polinome



Slika 5.6: Pospešitve algoritmov za kompleksne goste polinome



Slika 5.7: Poraba čipa za algoritme evalvacije gostih polinomov

Algoritem	LUT	FF1	FF2	BRAM	DSP
MultiDenseRealDFE	2,41	2,38	0,51	2,11	0,20
MultiDenseRealDFE64	17,03	20,91	4,12	10,15	6,35
MultiDenseRealDFE128	31,19	40,19	7,24	18,19	12,70
MultiDenseComplexDFE	3,49	3,54	0,71	2,91	0,79
MultiDenseComplexDFE64	53,02	60,06	11,39	18,75	25,40

Tabela 5.6: Odstotki porabe čipa za algoritme evalvacije gostih polinomov

5.3 Redki polinomi v več točkah

Za algoritme evalvacije redkih polinomov smo evalvirali kar enake goste polinome kot v prejšnjem razdelku 5.2, le da smo jih predstavili kot seznam parov koeficientov in eksponentov. Za vrednotenje smo vzeli polinome dolžin od 16 do 1024, in sicer 16, 32, 128 in 1024. Za število točk smo vzeli vrednosti od 16 do 67108864, pri čemer smo vmesna števila dobili tako, da smo prejšnjo vrednost množili z 2.

Gosti polinomi so podmnožica redkih polinomov in so znotraj množice

Algoritem	Opis
MultiSparseRealCPU	Ukazno-pretokovna implementacija za realne polinome, ki za poteciranje vsakega števila uporablja algoritem iz kode 3.9
MultiSparseRealDFE32	Paralelna različica iz razdelka 3.5.2 za realne polinome, ki naenkrat evalvira 32 točk
MultiSparseComplexCPU	Ukazno-pretokovna implementacija za kompleksne polinome, ki za potenciranje vsakega števila uporablja algoritem iz kode 3.9
MultiSparseComplexDFE8	Paralelna različica iz razdelka 3.5.2 za kompleksne polinome, ki naenkrat evalvira 8 točk

Tabela 5.7: Preizkušeni algoritmi za redke polinome v več točkah

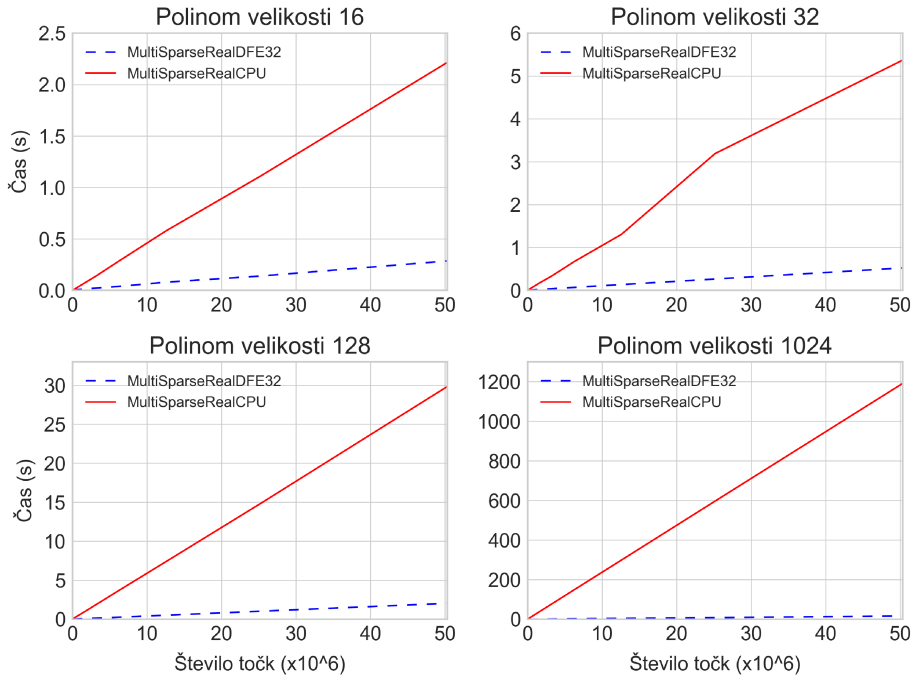
redkih polinomov tisti, katerih skupna vsota eksponentov je najnižja možna. S tem pa ukazno-pretokovna implementacija izvede najmanjše možno število operacij pri evalvaciji potenc. Zaradi paralelne evalvacije potenc pa velikost eksponentov ne vpliva na čas izvajanja algoritmov na podatkovno-pretokovni enoti. Na ta način smo tako dobili okvirno spodnjo mejo pospešitev podatkovno-pretokovne implementacije proti ukazno-pretokovni.

Za redke polinome v več točkah smo na DFE evalvirali samo paralelne različice algoritmov. Preizkušeni algoritmi so podani v tabeli 5.7.

5.3.1 Rezultati za realne polinome

Na sliki 5.8 so prikazani časi algoritmov za redke realne polinome, na sliki 5.9 pa še pospešitve. Vidimo, da je naš paraleliziran podatkovno-pretokovni algoritem veliko hitrejši od ekvivalentega ukazno-pretokovnega algoritma.

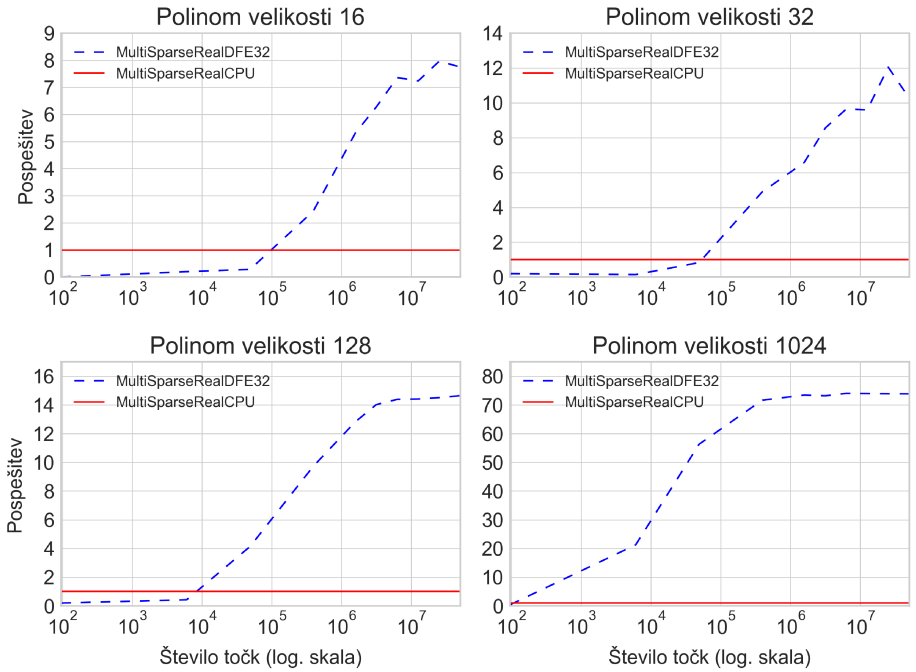
Na sliki 5.9 lahko vidimo, da dosežemo do osemkratne pospešitve za polinome do velikosti 16 in do sedemdesetkratne pospeštive za polinome velikosti 1024. Do tako velikih pospešitev pride, ker mora ukazno-pretokovni program



Slika 5.8: Časi algoritmov evalvacije redkih realnih polinomov

z večanjem polinoma evalvirati vedno večje potence in več členov. Velikost potence pa ne vpliva na čas izvajanja podatkovno-pretokovne implementacije, nanj vpliva le število členov.

Iz rezultatov tako lahko sklepamo, da je prenos algoritma na DFE primeren za vse velikosti polinomov.



Slika 5.9: Pospešitve algoritmov evalvacije redkih realnih polinomov

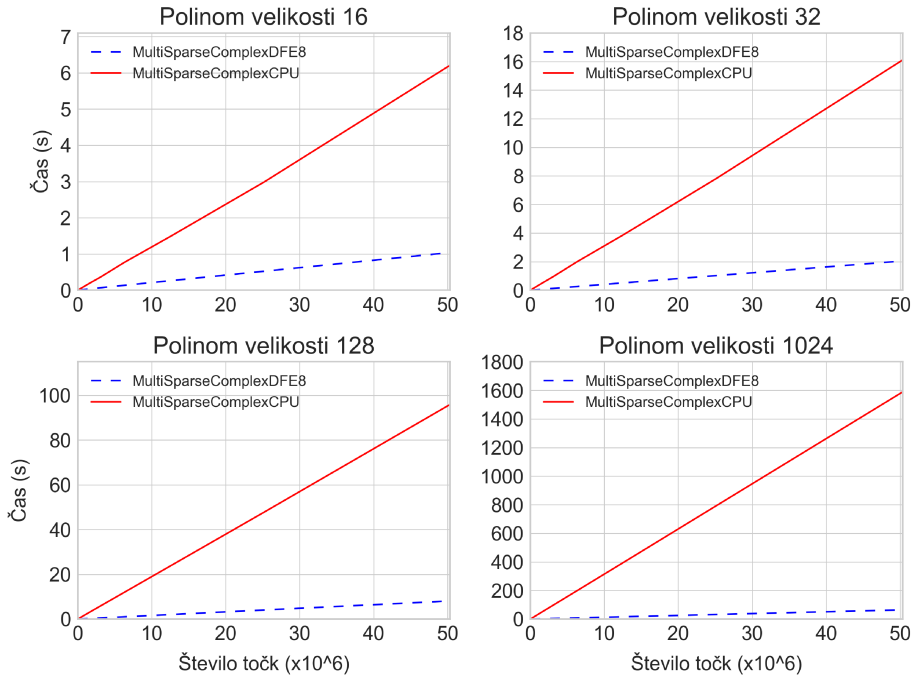
5.3.2 Rezultati za kompleksne polinome

Na sliki 5.10 so prikazani časi algoritmov za redke kompleksne polinome, na sliki 5.11 pa so podane pospešitve.

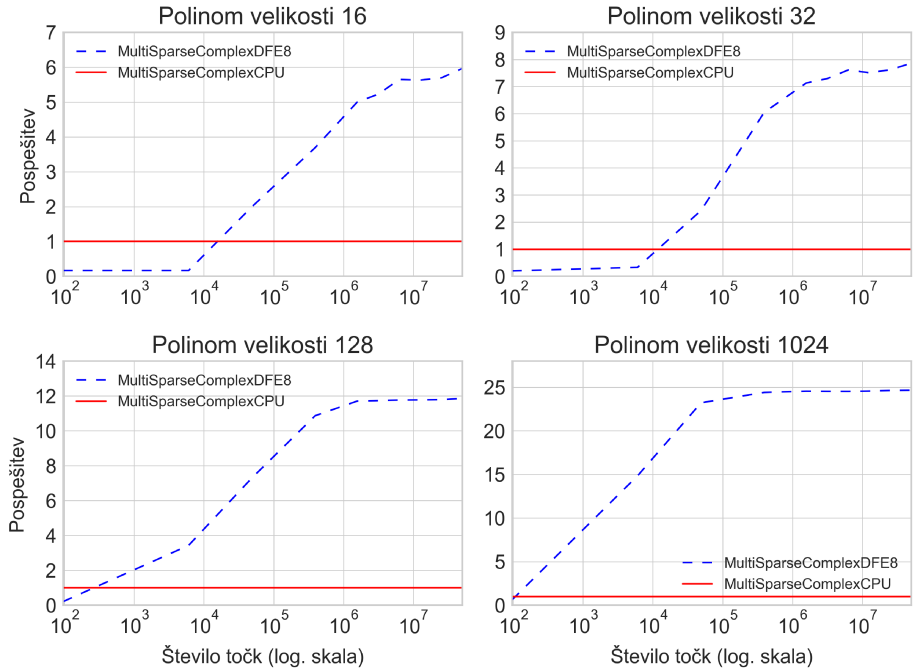
Podobno kot za algoritme v realnem tudi tu dobimo pospešitve na DFE, vendar dobimo tu nekoliko nižje pospešitve. Za polinome velikosti 16 do šestkratne, za polinome velikosti 1024 pa do petindvajsetkratne.

Nižje pospešitve so predvsem rezultat manjše paralelizacije. Pri algoritmu za realne polinome smo tako naenkrat evalvirali do 32 točk, algoritem za kompleksne polinome pa naenkrat evalvira samo 8 točk.

Čeprav so pohitritve manjše, pa so še vedno zadovoljive. Tako kot pri realnih redkih polinomih lahko zato tudi pri kompleksnih zaključimo, da je evalvacija redkih polinomov primerna za prenos na DFE.



Slika 5.10: Časi algoritmov evalvacije redkih kompleksnih polinomov



Slika 5.11: Pospešitve algoritmov evalvacije redkih kompleksnih polinomov

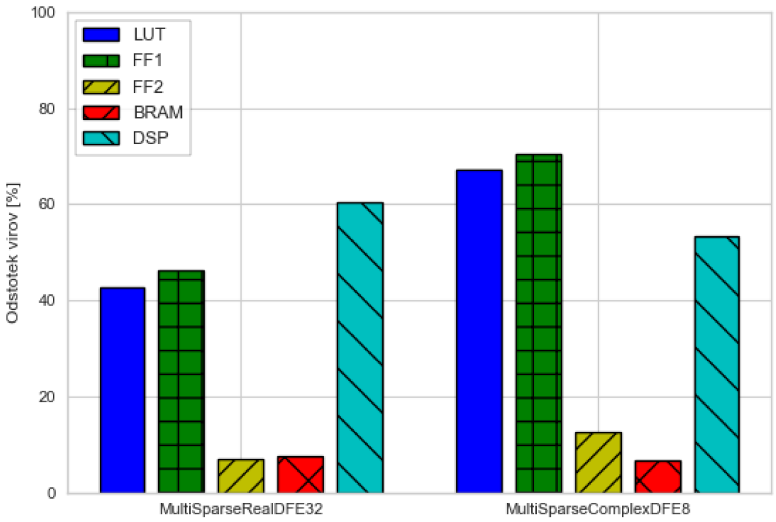
5.3.3 Poraba virov čipa

Poraba virov čipa je predstavljena v tabeli 5.8 in na sliki 5.12. Opazimo lahko, da smo pri obeh algoritmih, tako pri kompleksnih kot realnih polinomih, presegli 50 odstotkov deleža množilnikov (DSP). Podobno pa smo porabili tudi velik delež preslikovalnih tabel (LUT) in primarnih flip-flopov (FF1). Najverjetneje na relativno visoko porabo najbolj vpliva funkcija za izračun potenc. Vidimo tudi, da delež preslikovalnih tabel in primarnih flip-flopov v primeru kompleksnih polinomov celo presega delež porabe množilnikov. To lahko pripišemo načinu množenja kompleksnih števil, pri katerem se poleg realnega množenja uporabljajo tudi realna seštevanja, kot smo opisali v razdelku 4.1.

Če bi želeli povečati paralelizacijo, bi tako morali prilagoditi funkcijo izračuna potenc, da ta ne bi porabila toliko virov. Kot optimizacijo bi lahko zmanjšali najnižjo potenco, ki jo dovolimo izračunati, ali vpeljali kakšno drugo optimizacijo.

Algoritem	LUT	FF1	FF2	BRAM	DSP
MultiSparseRealDFE32	42,60	46,11	7,02	7,66	60,32
MultiSparseComplexDFE8	67,30	70,42	12,68	6,53	53,17

Tabela 5.8: Odstotki porabe čipa FPGA za algoritme evalvacije redkih polinomov



Slika 5.12: Poraba čipa za algoritme evalvacije redkih polinomov

5.4 Gručenje točk

Odločili smo se, da algoritem gručenja iz razdelka 4.2 preizkusimo v dveh dimenzijah. Preizkusili smo gručenje točk za 32 in 128 gruč. Za število točk smo vzeli vrednosti od 16 do 67108864, kjer smo vmesna števila dobili tako, da smo prejšno vrednost množili z 2.

Preizkušeni algoritmi so podani v tabeli 5.9.

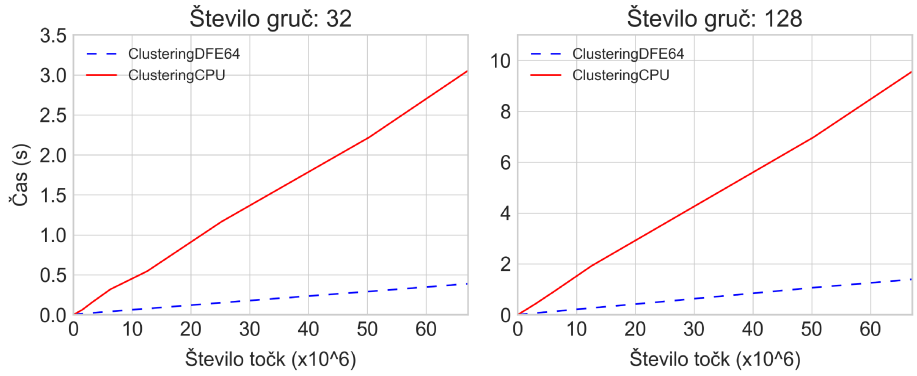
Algoritem	Opis
ClusteringCPU	Ukazno-pretokovna različica korakov gručenja iz razdelka 4.2
ClusteringDFE64	Podatkovno-pretokovna različica korakov gručenja iz razdelka 4.2, ki naenkrat evalvira 64 točk

Tabela 5.9: Preizkušeni algoritmi gručenja

5.4.1 Rezultati

Časi algoritmov so podani na sliki 5.13, pospešitve pa na sliki 5.14. Opazimo lahko, da smo v obeh primerih prehiteli ukazno-pretokovno implementacijo. Za obe velikosti gruč smo dosegli do okoli osemkratno pospešitev. Za 32 gruč dobimo pohitritve pri nekje 10^5 točk, medtem ko prečkamo mejo za 128 gruč malo prej, in sicer med 10^4 in 10^5 točk.

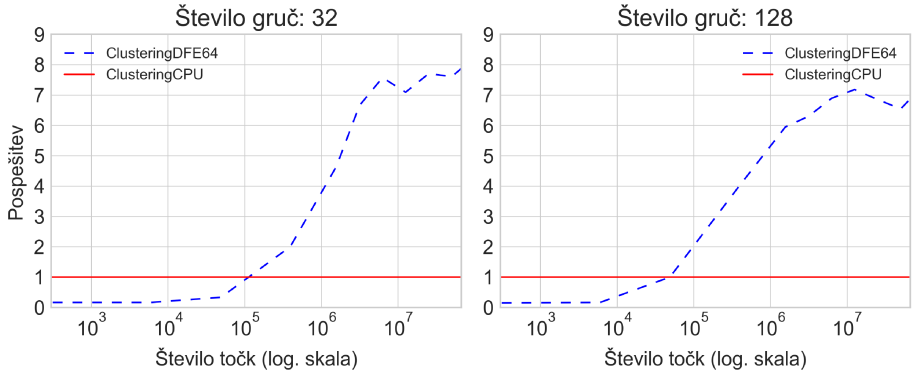
Ko smo primerjali čase algoritmov, smo ugotovili, da časi algoritma gručenja *ClusteringDFE64* na DFE sovpadajo s časi algoritma za goste polinome *MultiDenseRealDFE64* iz rezultatov v razdelku 5.2. Tako so časi gručenja v 32 gruč enaki časom evalvacije polinoma dolžine 32 za enako število točk. Podobno velja tudi za gručenje v 128 gruč in evalvacijo polinoma dolžine 128. To pa je pravzaprav pričakovan rezultat, saj smo za osnovo algoritma gručenja vzeli večtočkovno evalvacijo gostega polinoma.



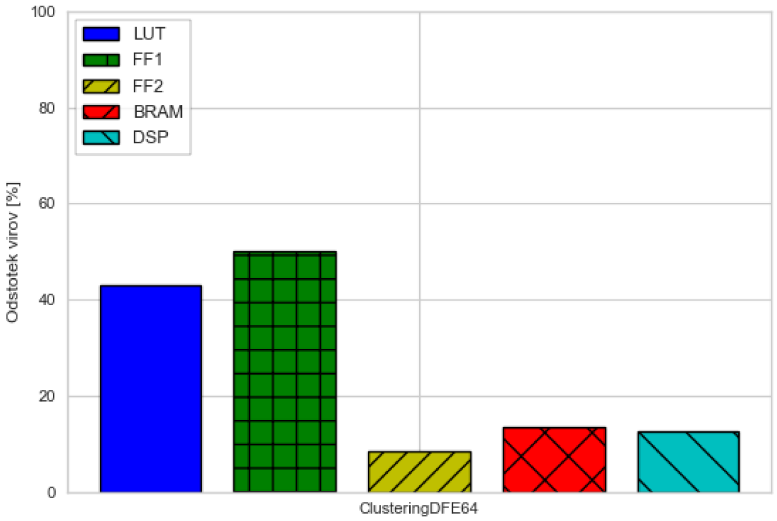
Slika 5.13: Časi algoritmov gručenja točk

5.4.2 Poraba virov čipa

Poraba čipa je prikazana v tabeli 5.10 in na sliki 5.15. Iz porabe lahko vidimo, da naš algoritem porabi največ preslikovnih tabel (LUT), okoli 43 %, in primarnih flip-flopov (FF1), okoli 50 %. Z nekaj optimizacije bi tako lahko še enkrat povečali paralelizacijo oziroma število točk, ki jih naenkrat evalviramo, s tem pa tudi pohitrili naš algoritem.



Slika 5.14: Pospešitve algoritmov gručenja točk



Slika 5.15: Poraba čipa za gručenje točk

Algoritem	LUT	FF1	FF2	BRAM	DSP
ClusteringDFE64	43,05	50,02	8,56	13,44	12,70

Tabela 5.10: Odstotki porabe čipa za gručenje točk

5.5 Diskretna Fourierova transformacija

Evalvacija diskretne Fourierove tranformacije je služila tudi kot primerjalni test s programsko opremo, ki je namenjena računanju. Našo implementacijo smo primerjali s knjižnico FFTW [33], ki je ena hitrejših knjižnic za računanje diskretne Fourierove transformacije za centralne procesne enote in uporablja hitro Fourierovo transformacijo za računanje, torej je njen algoritem asimptotično hitrejši od naše implementacije.

Zaradi omejitev testiranja smo implementacijo FFTW preizkusili na drugem računalniku s centralno procesno enoto Intel(R) Core(TM) i7-4770HQ CPU @ 2.20 GHz, ki je po naših testih približno 1.25-krat počasnejši od delovne postaje, opisane v uvodu v razdelku 5. Vendar je bil za primerjavo asimptotično različnih algoritmov dovolj hiter.

Testirali smo različne dolžine diskretne Fourierove tranformacije in različno število transformacij. Preizkušeni algoritmi so podani v tabeli 5.11.

Algoritem	Opis
DFTCPU	Implementacija na centralni procesni enoti po definiciji DFT
CPUFFTW	Algoritem FFT s knjižnico FFTW3 na centralni procesni enoti
DFEDFT64	Algoritem za evalvacijo DFT na DFE, kjer smo lahko naenkrat za eno točko evalvirali 64 koeficientov/točk, iz poglavja 4.3

Tabela 5.11: Preizkušeni algoritmi evalvacije diskretne Fourierove transformacije

5.5.1 Rezultati

Rezultati so podani na slikah 5.16 in 5.17. Vidimo, da smo z implementacijo na podatkovno-pretokovni enoti presegli ukazno-pretokovno implementacijo

DFT po definiciji, a smo bili počasnejši od implementacije FFT knjižnice FFTW. Vendar pa se moramo zavedati, da smo uporabili algoritem, ki je asimptotično kvadratnega reda, medtem ko je FFT reda $O(n \log n)$.

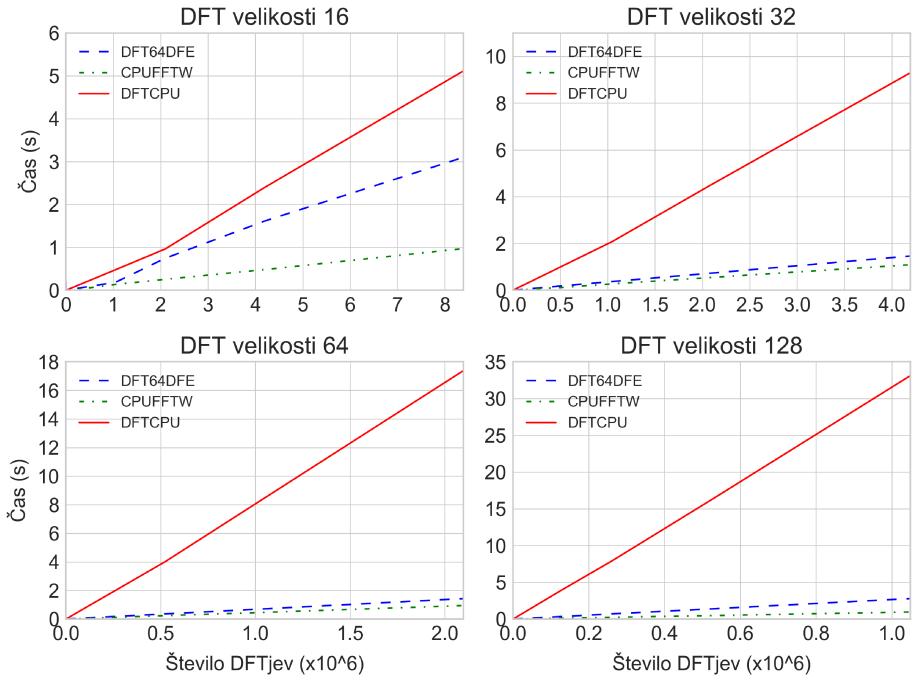
Kot smo predvidevali v opisu implementacije DFT v razdelku 4.3, se naš algoritem na podatkovno-pretokovni enoti najbolje obnese za tiste dolžine DFT, ki ustrezajo številu naenkrat evalviranih koeficientov. To je bilo v naši implementaciji 64. Vidimo, da implementacijo DFT po definiciji pospešimo do desetkrat, algoritmu FFT pa se približamo na 0.7-kratno hitrost, kar je še vedno zadovoljivo. Vendar pa v primeru večjih DFT (npr. 128 členov) asimptotični nižji red algoritma FFT prevlada in postane naš algoritem na podatkovno-pretokovni enoti opazno počasnejši v primerjavi z njim. Zato je naš algoritem na podatkovno-pretokovni primeren samo za DFT z manj kot 128 členi ali v primeru, ko hočemo izračunati DFT, ki niso dolžine 2^k za neko naravno število k.

Vendar pa so rezultati vseeno spodbudni, saj lahko vidimo, da se naši algoritmi s paralelizacijo v določenih okvirjih kosajo tudi z asimptotično boljšimi algoritmi.

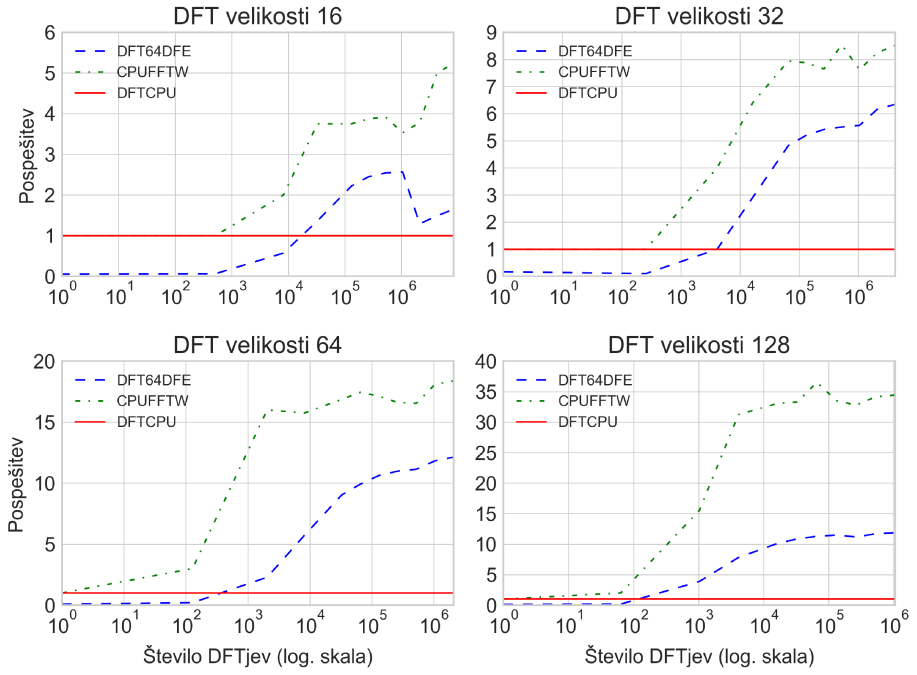
5.5.2 Poraba virov čipa

Poraba čipa je predstavljena v tabeli 5.12 in na sliki 5.18. Vidimo lahko, da naša implementacija porabi velik delež BRAM-a. Razlog je ta, da opravljamo veliko paralelnega branja iz pomnilnika, kar povzroči, da podatkovno-pretokovna enota pomnoži osnovni pomnilnik, da se podatke lahko bere paralelno.

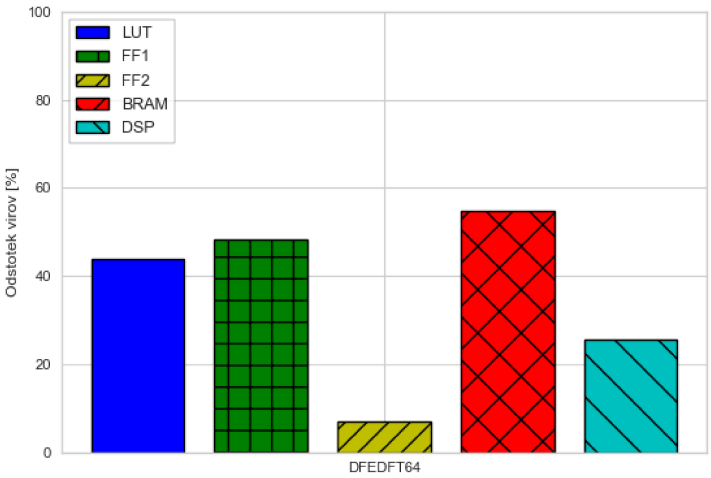
Z optimizacijo bi naš algoritem morda lahko še enkrat povečali. S tem bi bolje deloval tudi za DFT velikosti 128. Tabela in slika prikazujeta, da zaradi množenja kompleksnih števil porabimo veliko množilnikov. Sklepamo lahko, da bi v primeru evalvacije realnih števil porabili manj množilnikov in BRAM-a in bi lahko algoritem zagotovo vsaj še enkrat povečali.



Slika 5.16: Časi evalvacije DFT na DFE



Slika 5.17: Pospešitve DFT na DFE



Slika 5.18: Poraba čipa za evalvacijo DFT

Algoritem	LUT	FF1	FF2	BRAM	DSP
DFEDFT64	43,79	48,39	6,99	54,79	25,64

Tabela 5.12: Odstotki porabe čipa FPGA za evalvacijo DFT

Poglavje 6

Sklepne ugotovitve

V magistrski nalogi so bili prikazani algoritmi za evalvacijo polinomov na podatkovno-pretokovni arhitekturi. Natančneje povedano, implementirali smo algoritme za podatkovno-pretokovne računalnike Maxeler. Implementirali smo algoritme za goste in redke polinome v eni spremenljivki za evalvacijo ene ali več točk. Poleg tega smo naše algoritme prilagodili za reševanje podproblema gručenja točk in evalvacijo diskretne Fourierove transformacije.

Za goste polinome v več točkah smo dobili okoli dvajsetkratne pohitritve, za redke v več točkah pa tudi do sedemdesetkratne, medtem ko za enotočkovno evalvacijo nismo dobili opaznejših pohitritev. Podatkovno-pretokovne algoritme bi lahko še nekoliko izboljšali, saj v nobenem primeru nismo porabili vseh virov čipa. Verjetno pa bi lahko izboljšali tudi ukazno-pretokovne algoritme, vendar bi težko dosegli dvajsetkratne pospešitve za goste oziroma sedemdesetkratne za redke polinome.

Algoritmi, ki smo jih implementirali v magistrski nalogi, bi lahko dodatno izboljšali in povečali velikost paralelizacije, s čimer bi jih tudi pohitrili. Zanimivo pa bi jih bilo prilagoditi tudi za reševanje drugih problemov. Naše algoritme bi tako lahko prilagodili za evalvacijo polinomov več spremenljivk. Lahko bi se poigrali tudi z natančnostjo izračunov. Mi smo uporabljali enojno natančnost v plavajoči vejici, zanimivo pa bi bilo videti, kakšne pohitritve dobimo, če bi to natančnost zmanjšali ali povečali. Poleg tega bi se lahko

osredotočili tudi na točno določene probleme evalvacije polinomov, na primer simetričnih polinomov.

Podatkovno-pretokovni računalnik Maxeler je namenjen reševanju specifičnih problemov. Za programerja, vajenega ukazno-pretokovnih programov, predstavlja podatkovno-pretokovni računalnik izziv, zato je za učinkovite rešitve potrebno kar nekaj vaje in iteracij implementacij algoritmov. Poleg tega tudi ni primeren za prenos vseh algoritmov. Za specifične probleme, ki nudijo velike možnosti paralelizacije in se izvajajo na veliko podatkih, pa z njim dobimo pospešitve, ki jih z ukazno-pretokovnimi programi na centralni procesni enoti ne bi dosegli, obenem pa lahko prihranimo tudi na energiji [3]. Zato sta raziskovanje in implementacija algoritmov na podatkovno-pretokovni arhitekturi pomembna tako za popularizacijo podatkovno-pretokovne arhitekture kot za namene reševanja specifičnih problemov v znanstvenem računanju. Popularizacija in uporaba podatkovno-pretokovne arhitekture pa omogočata nadaljnji razvoj arhitekture in njen napredek.

Literatura

[1] Maxeler Technologies, http://maxeler.com, Dostopano: 29.5.2017.

[2] Uroš Čibej, Jurij Mihelič, Adaptation and Evaluation of the Simplex Algorithm for a Data-flow Architecture, in: Advances in Computers, Vol. 106, 2017.

[3] L. Gan, H. Fu, C. Yang, W. Luk, W. Xue, O. Mencer, X. Huang, G. Yang, A Highly-Efficient and Green Data Flow Engine for Solving Euler Atmospheric Equations, in: Field Programmable Technology (FPT) Internetional Conference, 2015.

[4] Free fall - Wikipedia, https://en.wikipedia.org/wiki/Free_fall, Dostopano: 29.5.2017.

[5] G. Farin, J. Hoschek, M.-S. Kim, Handbook of Computer Aided Geometric Design, North-Holland, 2002.

[6] Jong-Bae Park, Ki-Song Lee, Joong-Rin Shin, Kwang Y. Lee, A Particle Swarm Optimization for Economic Dispatch With Nonsmooth Cost Functions, in: IEEE Transactions on Power Systems, Vol. 20, 2005, pp. 34–42.

[7] Polynomial - Encyclopedia of Mathematics, https://www.encyclopediaofmath.org/index.php/Polynomial, Dostopano: 29.5.2017.

[8] Stephen C. Johnson, Sparse Polynomial Arithmetic, in: ACM SIGSAM, Vol. 8, 1974, pp. 63–71.

[9] Arnold Neumaier, Introduction to Numerical Analysis, Cambridge University Press, 2001.

[10] C. W. Ueberhuber, Numerical Computation 1: Methods, Software, and Analysis, Springer, 1997.

[11] Ian Munro, Michael Paterson, Optimal Algorithms for Parallel Polynomial Evaluation, in: Journal of Computer and System Sciences, Vol. 7, 1973, pp. 189–198.

[12] D. J. MacKay, An Example Inference Task: Clustering, in: Information Theory, Inference, and Learning Algorithms, Cambridge University Press, Cambridge, 2003, Ch. 20, pp. 284–292.

[13] Discrete Cosine Transform - Wikipedia, `https://en.wikipedia.org/wiki/Discrete_cosine_transforml`, Dostopano: 29.5.2017.

[14] R. J. M. Wesley M. Johnston, J.R. Paul Hanna, Advances in Dataflow Programming Languages, in: ACM Computing Surveys, Vol. 36, 2004, pp. 1–34.

[15] Walid A. Najjar, Edward A. Lee, Guang R Gao, Advances in the Dataflow Computational Model, in: Parallel Computing - Special Anniversary issue, Vol. 25, 1991, pp. 1907–1929.

[16] Maxeler technologies, Maxeler DFE Debugging and Optimization Tutorial, 2016.

[17] Veljko Milutinović, Jakob Salom, Nemanja Trifunovic, Roberto Giorgi, Guide to DataFlow Supercomputing, Springer, 2015.

[18] Maxeler technologies, Compiler tutorial, 2016.

[19] M. Žnider, Matrični algoritem na podatkovno-pretokovnih računalnikih, Magistrsko delo (2016).

[20] Maxcloud Maxeler Technologies, `https://www.maxeler.com/products/maxcloud`, Dostopano: 29.5.2017.

[21] M. Eleršič, Prevajanje grafne vmesne kode za Maxeler podatkovno-pretokovne enote, Diplomsko delo (2015).

[22] Maxcompiler Maxeler Technologies, `https://www.maxeler.com/products/software/maxcompiler`, Dostopano: 29.5.2017.

[23] Eclipse - The Eclipse Foundation Open Source Community Website., `https://www.eclipse.org`, Dostopano: 29.5.2017.

[24] Jurij Mihelič, Uroš Čibej, Experimental Algorithmics for the Dataflow Architecture: Guidelines and Issues, in: IPSI BgD Transactions on Advanced Research, Vol. 13, 2017, pp. 1–8.

[25] Exponentiation by squaring - Wikipedia, `https://en.wikipedia.org/wiki/Exponentiation_by_squaring`, Dostopano: 29.5.2017.

[26] Maxeler standard library, `https://github.com/maxeler/maxpower`, Dostopano: 29.5.2017.

[27] Complex Addition – from Wolfram MathWorld, `http://mathworld.wolfram.com/ComplexAddition.html`, Dostopano: 20.8.2017.

[28] Complex Multiplication – from Wolfram MathWorld, `http://mathworld.wolfram.com/ComplexMultiplication.html`, Dostopano: 20.8.2017.

[29] S. Lloyd, Least Squares Quantization in PCM, IEEE Transactions on Information Theory 28 (2) (1982) 129–137.

[30] k-Means clustering - Algorithm and Examples, `http://www.onmyphd.com/?p=k-means.clustering`, Dostopano: 20.8.2017.

[31] Discrete Fourier Transform – from Wolfram MathWorld, http://mathworld.wolfram.com/DiscreteFourierTransform.html, Dostopano: 29.5.2017.

[32] James W. Cooley, John W. Tukey, An Algorithm for the Machine Calculation of Complex Fourier Series, in: Mathematics of Computation, Vol. 19, 1965, pp. 297–301.

[33] FFTW Home Page, http://www.fftw.org, Dostopano: 29.5.2017.

[34] Maxeler/DCTApp, https://github.com/maxeler/DCTApp, Dostopano: 29.5.2017.

[35] Eric Jacobsen, Richard Lyons, The Sliding DFT, in: IEEE Signal Processing Magazine, Vol. 20, 2003, pp. 74–80.