Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tadej Ciglarič

# Stiskanje podatkov na grafični procesni enoti

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana, 2016

To delo je ponujeno pod licenco Creative Commons Attribution-ShareAlike International 4.0 (CC BY-SA 4.0). To pomeni, da se tako besedilo, slike, grafi in druge sestavine dela kot tudi rezultati diplomskega dela lahko prosto distribuirajo, reproducirajo, uporabljajo, priobčujejo javnosti in predelujejo, pod pogojem, da se jasno in vidno navede avtorja in naslov tega dela in da se v primeru spremembe, preoblikovanja ali uporabe tega dela v svojem delu, lahko distribuira predelava le pod licenco, ki je enaka tej. Podrobnosti licence so dostopne na spletni strani `creativecommons.org`.

*Besedilo je oblikovano z urejevalnikom besedil LaTeX.*

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Zaradi nenehno naraščajoče količine podatkov je njihovo stiskanje vedno bolj pomembno na področjih, kjer lahko omejitve pasovne širine pri prenosih ali prostora za shranjevanje negativno vplivajo na delovanje sistema. Učinkoviti algoritmi za stiskanje podatkov pa so lahko računsko zelo zahtevni. Ogromno računskih kapacitet je na voljo v modernih grafičnih procesnih enotah. V delu raziščite možnosti za paralelizacijo algortima deflate na grafični procesni enoti in eno od njih izvedite z ogrodjem OpenCL. Vašo rešitev primerjajte s sekvenčno različico in z obstoječimi rešitvami.

# Kazalo

**Povzetek**

**Abstract**

**1 Uvod 1**

1.1 Metodologija . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Pregled področja 3**

2.1 Kompresija . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2.2 Splošno procesiranje na grafičnih procesnih enotah . . . . . . . 10

2.3 Obstoječe paralelne implementacije algoritma deflate za izvajanje na grafičnih karticah . . . . . . . . . . . . . . . . . . . . 15

**3 Izvedba 19**

3.1 Sekvenčni algoritem . . . . . . . . . . . . . . . . . . . . . . . 19

3.2 Paralelizacija . . . . . . . . . . . . . . . . . . . . . . . . . . 22

**4 Testiranje in rezultati 27**

4.1 Testiranje sekvenčne implementacije . . . . . . . . . . . . . . . 27

4.2 Testiranje paralelne implementacije . . . . . . . . . . . . . . . 28

4.3 Primerjava sekvenčne in paralelne implementacije . . . . . . . 32

4.4 Primerjava naše paralelne implementacije z obstoječo . . . . . 35

4.5 Možnosti za nadaljne izboljšave . . . . . . . . . . . . . . . . . 37

**5 Zaključek 39**

**Literatura** **42**

# Seznam uporabljenih kratic

| kratica | angleško | slovensko |
|---|---|---|
| **API** | application programming interface | aplikacijski programski vmesnik |
| **CPU** | central processing unit | centralna procesna enota |
| **C++ AMP** | C++ Accelerated massive parallelism | pospešen masovni paralelizem |
| **CUDA** | Compute unified device architecture | poenotena arhitektura računskih naprav |
| **GPU** | graphical processing unit | grafična procesna enota |
| **GPGPU** | general processing on graphical processing units | splošno procesiranje na grafičnih procesnih enotah |

# Povzetek


**Naslov:** Stiskanje podatkov na grafični procesni enoti

**Avtor:** Tadej Ciglarič

Učinkoviti algoritmi za stiskanje podatkov so lahko dokaj počasni. Namen tega dela je poizkus paralelizacije kompresijskega algoritma za izvajanje na grafičnih karticah. Ker grafične kartice vsebujejo zmogljivo vzporedno računsko enoto, bi pričakovali, da je mogoče izvajanje algoritma močno pohitriti. Zato smo v tem delu pregledali algoritem deflate in obstoječe paralelne implementacije za izvajanje tega algoritma na grafičnih procesnih enotah. Implementirali smo sekvenčni algoritem in ga na dva načina paralelizirali z uporabo ogrodja OpenCL. Vse implementacije smo preizkusili na korpusu datotek za testiranje algoritmov za stiskanje podatkov. Primerjali smo rezultate naših implementacij z obstoječimi paralelnimi in sekvenčnimi implementacijami.



**Ključne besede:** GPGPU, OpenCL, stiskanje podatkov, algoritem deflate, paralelizacija.

# Abstract

**Title:** Data Compression on Graphics Processing Unit

**Author:** Tadej Ciglarič

Efficient data compression algorithms can be slow. The purpose of this work is an attempt of efficient parallelization of compression algorithm for execution on graphics processing units. Since graphics processing units contain an efficient parallel computing unit, it is reasonable to expect speedup from such parallelization of the algorithm. This work contains an overview of deflate algorithm and its existing parallel implementations intended for graphics processing units. We sequentially implemented the algorithm and parallelized it in two different ways using OpenCL framework. The implementations were tested on a corpus of files, intended for testing of compression algorithms. We compared the results with existing sequential and parallel implementations.

**Keywords:** GPGPU, OpenCL, data compression, deflate algorithm, paralellization.

# Poglavje 1

# Uvod

V današnjem času generiramo ogromne količine podatkov. Te podatke je potrebno shranjevati, pogosto pa se jih pošilja prek mreže. Diski imajo omejeno kapaciteto, mrežne povezave pa omejeno pasovno širino. Nadgradnje strojne opreme so lahko drage.

Količino podatkov lahko zmanjšamo s stiskanjem. Podatki so pogosto zapisani redundantno – na daljši način, kot bi bilo nujno potrebno. To izkoriščajo kompresijski algoritmi, ki podatke zapišejo na bolj zgoščen način. Zato je pred shranjevanjem ali pošiljanjem podatkov smiselna uporaba kompresije.

Kompresija je računsko dokaj zahtevna. Kadar kompresijo uporabljamo hkrati z zapisovanjem na diske ali pošiljanjem po mreži, lahko ta predstavlja ozko grlo – omejuje hitrosti zapisovanja ali pošiljanja. Zato bi bilo koristno, če bi lahko algoritme za stiskanje podatkov poganjali na grafičnih karticah, namesto na centralnih procesorjih.

Grafične kartice so primarno namenjene izrisovanju grafike. Sodobne modele grafičnih kartic je možno programirati, kar pomeni, da lahko na njih izvajamo poljuben program. Za primerne algoritme – take, ki jih je mogoče napisati v obliki hkratnega izvajanja enakih operacij na veliki količini podatkov, so lahko grafične kartice mnogo hitrejše od centralnih procesorjev.

## 1.1 Metodologija

Pregledali bomo specifikacije algoritma deflate in njegovih komponent, Huffmanovega kodiranja in iskanja ponovitev LZSS.

Nato bomo implementirali sekvenčni algoritem deflate. Za preverjanje pravilnosti delovanja bomo implementirali tudi inflate – algoritem, ki raztegne z algoritmom deflate stisnjene podatke. Implementacijo bomo profilirali, da ugotovimo, v katerem delu se porabi največ časa.

Pregledali bomo, kakšni pristopi k paralelizaciji tega algoritma že obstajajo. Analizirali bomo različne možnosti za paralelizacijo in ovrednotili njihove prednosti in slabosti. Eno bomo izbrali in jo implementirali.

Paralelno implementacijo bomo testirali na različnih količinah podatkov in merili čas izvajanja. Testi bodo izvedeni na podatkih iz javno dostopnega korpusa za testiranje algoritmov za stiskanje podatkov. Vsak test bomo izvedli desetkrat in izračunali povprečni čas izvajanja.

Našo paralelno implementacijo bomo primerjali z obstoječimi sekvenčnimi, našo sekvenčno in obstoječimi paralelnimi implementacijami za grafične kartice.

# Poglavje 2

# Pregled področja

## 2.1 Kompresija

Kompresija je kodiranje podatkov z namenom skrajšanja njihovega zapisa. Koliko se zapis skrajša se meri s kompresijskim razmerjem. To je razmerje med količino podatkov, potrebno za kompresiran zapis, in količino podatkov, potrebno za nekompresiran zapis. Kompresijsko razmerje je odvisno od kompresijskega algoritma in podatkov, ki jih stiska. Popolnoma naključnih podatkov ni mogoče učinkovito stisniti, a podatki, ki se jih stiska, običajno niso naključni. Pogosto imajo lastnosti, kot so ponavljanje delov podatkov in različne frekvence ponovitev znakov, s katerimi so zapisani. Predvsem ti dve lastnosti izkoriščajo kompresijski algoritmi, da podatke zapišejo na krajši način.

Poleg kompresijskega razmerja so pomombne lastnosti kompresijskih algoritmov tudi hitrost stiskanja, hitrost raztegovanja in količina pomnilnika, ki ga potrebujejo. Hitrost stiskanja ali razširjanja je razmerje med količino nestisnjenih podatkov, ki jih algoritem stisne ali razširi, in časom, ki ga za to porabi. Če želimo izboljšati eno izmed teh štirih lastnosti, lahko to običajno storimo le na račun ene ali večih izmed preostalih.

Obstajajo izgubni in brezizgubni kompresijski algoritmi. Brezizgubni kompresijski algoritmi zapišejo podatke tako, da v podatkih po stiskanju

in raztegovanju ni nobenih sprememb. Izgubna kompresija pa lahko podatke nekoliko spremeni – pojavijo se izgube. Zato izgubna kompresija ni primerna za vse vrste podatkov, pač pa le za take, kjer so manjše spremembe nemoteče. Obstajajo specializirani izgubni algoritmi za stiskanje določenih vrst podatkov, na primer za zvok, slike in video posnetke, ki običajno dosegajo boljša kompresijska razmerja od brezizgubnih.

### 2.1.1 LZ77 in LZSS

LZ77 [14] je kompresijski algoritem, ki deluje tako, da nizov, ki se v vhodnih podatkih ponovijo večkrat, ne zapiše vsakič v celoti, ampak v drugi ali kasnejši ponovitvi istega niza zapiše niz z referenco na prejšnjo ponovitev. Znake, ki niso del ponovljenega niza, zapiše dobesedno. Ime LZ77 je sestavljeno iz začetnic priimkov avtorjev – Lempel in Ziv ter letnice objave 1977.

Zakodirani podatki so sestavljeni iz vhodnih znakov in referenc na ponovitve. Referenca je sestavljena iz para (*odmik*, *dolžina*). To pomeni, da je začetek ponovljenega niza oddaljen za *odmik* nekodiranih znakov in je dolg *dolžina* znakov.

Originalni algoritem predvideva, da v kompresiranih podatkih referenci vedno sledi znak vhodne abecede in vhodnemu znaku referenca. Če sta v vhodnih podatkih dva zaporedna znaka, ki nista del ponovitev, bosta v kompresiranih podatkih zapisana dobesedno, med njima pa bo referenca (*0*,*0*). Na ta način se izognemo problemu, ki bi drugače nastal pri dekodiranju, če ne bi bilo določeno, ali določen del kompresiranih podatkov predstavlja referenco ali nekodirane znake. Če je v vhodnih podatkih malo ponovitev, lahko dodatne reference povzročijo, da kompresija podaljša zapis, namesto da bi ga skrajšala.

To težavo močno omili nadgrajeni algoritem LZSS [13]. Ta določa, da je v kompresiranih podatkih pred vsakim znakom ali referenco en bit, ki pove, ali mu sledi referenca ali znak vhodne abecede. S tem se odpravi potreba po fiksnem vrstnem redu znakov vhodne abecede in frekvenc, zato

se reference uporablja le tam, kjer porabi za njihov zapis manj bitov, kot bi jih za ponovljen niz.

Za kompresijo LZSS uporabljamo drseče okno (angl. *sliding window*). Drseče okno sestavljajo zadnji vhodni znaki, ki so že bili skompresirani, dolgo pa je toliko, kolikor je največji dovoljeni odmik pri zapisu reference na ponovitev.

V drsečem oknu algoritem išče ponovitve niza, ki se začnejo s prvim še ne skompresiranim znakom. Če je najdaljša ponovitev krajša, kot bi bil njen zapis z referenco, na izhod zapiše trenutni znak in se premakne na naslednji znak. Za eno mesto naprej premakne tudi drseče okno. Če pa je ponovitev daljša od reference, se na izhod zapiše referenca. Algoritem se premakne naprej za toliko znakov, kot je dolga ponovitev. Za enako število znakov premakne tudi drseče okno. Ta postopek se ponavlja, dokler niso skompresirani vsi podatki.

Iskanje ponovitev z izčrpnim preiskovanjem vseh možnih podnizov v drsečem oknu je računsko zelo zahtevno. Zato za iskanje ponovljenih nizov večinoma uporabljamo podatkovne strukture, ki omogočajo, da niz primerjamo le z nekaterimi izmed podnizov v drsečem oknu, večino pa jih lahko vnaprej zavržemo.

Primer take strukture je zgoščevalna tabela (angl. *hashtable*). Za iskanje ponovitev je implementirana kot tabela, v kateri so shranjeni kazalci na mesta, kjer se začnejo posamezni nizi. Ponovitev niza poiščemo tako, da izračunamo vrednost zgoščevalne funkcije za prvih nekaj bajtov tega niza. Ta vrednost je indeks, kje v tabeli se nahaja kazalec na prejšnji niz z enakim začetkom. Da je to mogoče, je potrebno vrednost v zgoščevalni tabeli posodobiti vsakič, ko se drseče okno premakne naprej. Za vse nize, ki se začnejo z znaki, ki so se premaknili v drseče okno, je potrebno vnesti kazalce v tabelo.

Z uporabo zgoščevalne tabele je mogoče hitro najti lokacijo zadnjega niza z istim začetkom. Če želimo iskati tudi prejšnje nize z istimi začetnimi znaki, lahko uporabimo verižno tabelo (angl. *chain array*). V tej je en vnos za vsak znak (ali več zaporednih znakov) v drsečem oknu. V tabeli je na mestu, ki

ustreza temu znaku, kazalec na prejšnje mesto v drsečem oknu, kjer se niz začne z enakimi znaki.

Raztegovanje podatkov, zapisanih s kodom LZSS, je preprosto. Če je v stisnjenih podatkih naslednji nekodiran znak, ga prepišemo na izhod. Če je naslednja ponovitev, na izhod skopiramo *dolžina* znakov, ki so v izhodni tabeli oddaljeni za *odmik*.

### 2.1.2 Huffmanov kod

Huffmanov kod [9] vsakemu možnemu vhodnemu znaku priredi ne nujno za vse znake enako dolgo zaporedje bitov, s katerim se ga zakodira – kodno zamenjavo. Je prefiksni kod, kar pomeni, da za vse kodne zamenjave, ki predstavljajo znake, velja, da se nobena kodna zamenjava ne ponovi na začetku druge. To je zadosten pogoj, da je mogoče ob poznavanju koda zakodirane podatke enolično dekodirati brez dodatnih podatkov [11].

Algoritem kot vhodne podatke prejme pogostosti vhodnih znakov in vsakemu znaku določi kodno zamenjavo tako, da dobijo pogostejši znaki krajše zamenjave. Ob predpostavkah, da vsak znak kodiramo s celim številom bitov in da verjetnost pojavitve posameznega znaka ni odvisna od predhodnih znakov, je Huffmanov kod optimalen – ne obstaja drug kod, ki bi sporočilo zakodiral z manj biti [11]. V praksi predpostavka, da verjetnost pojavitve posameznega znaka ni odvisna od predhodnih znakov večinoma ne drži, obstajajo pa tudi kodi, ki znakov ne kodirajo nujno s celim številom bitov, a je Huffmanov kod kljub temu uporaben.

Za konstrukcijo koda potrebujemo frekvence ponovitev vseh vhodnih znakov. V vsakem koraku poiščemo dva znaka, ki imata najmanjši (neničelni) frekvenci, in ju združimo v nov znak. To storimo tako, da oba znaka odstranimo iz seznama, vanj pa dodamo nov znak s frekvenco, ki je vsota frekvenc združenih znakov. Za nov znak si označimo iz katerih dveh znakov je narejen. Postopek ponavljamo, dokler ne dobimo enega samega znaka. Tako dobimo Huffmanovo drevo.

Znaku v korenu določimo kodno zamenjavo, dolgo 0 bitov. Nato dodelimo

znakoma, iz katerih je bil sestavljen, za en bit daljši kodni zamenjavi. Biti v teh dveh zamenjavah so, razen zadnjega, enaki zamenjavi znaka, ki ga sestavljata. Zadnji bit je v zamenjavi enega izmed znakov enak 0, v zamenjavi drugega 1. Vsakič ko sestavljenemu znaku dodelimo kodno zamenjavo, na njem ponovimo isti postopek. Tako dobimo kodne zamenjave za vse vhodne znake.

Kodiranje s Huffmanovim kodom je enako kot s katerimkoli prefiksnim kodom. Za vsak znak v vhodnih podatkih se na izhod zapiše kodno zamenjavo, ki ustreza temu znaku.

Za dekodiranje podatkov zapisanih s Huffmanovim kodom potrebujemo Huffmanovo drevo, s katerim so bili podatki zakodirani. Pri dekodiranju začnemo v korenu drevesa in podatke beremo bit po bit. Vsak bit določa v katero vejo drevesa se premaknemo. Ko pridemo do lista, smo dekodirali znak v tem listu. Ponovno se premaknemo v koren drevesa in nadaljujemo postopek.

Ker pri dekodiranju potrebujemo Huffmanovo drevo, ga je potrebno zapisati zraven zakodiranih podatkov. Količino podatkov, ki jih porabimo za zapis, lahko zmanjšamo, če uporabumo kanonični Huffmanov kod [11]. V splošnem za neko sporočilo obstaja več različnih Huffmanovih kodov. Kanonični kod ima dodatne omejitve, ki ga za podane frekvence ponovitev vhodnih znakov enolično definirajo. Za zapis kanoničnega Huffmanovega koda ni potreben zapis celotnega Huffmanovega drevesa, ampak ga je mogoče konstruirati le iz dolžin kodnih zamenjav vhodnih znakov.

Za konstrukcijo kanoničnega koda uporabimo isti algoritem, kot za poljuben Huffmanov kod, vendar ne uporabimo dobljenih kodnih zamenjav, ampak le njihove dolžine. Znake sortiramo najprej po dolžini kodnih zamenjav in nato po abecedi. V tem vrstnem redu znakom po naraščajočem vrstnem redu dodeljujemo kodne zamenjave. Prvi znak dobi zamenjavo sestavljeno iz samih ničel. Naslednjemu znaku začasno dodelimo naslednjo kodno zamenjavo iste dolžine. Operacija naslednika nad kodnimi zamenjavami je definirana, če na njih gledamo kot na števila v dvojiškem zapisu. Če mora imeti znak

daljšo kodno zamenjavo od prejšnjega, njegovi kodni zamenjavi z desne dodamo ustrezno število ničel. Če gledamo na kodne zamenjave kot na binarna števila, je to enakovredno operaciji zamika v levo za toliko bitov, kolikor se dolžina zamenjav zaporednih znakov razlikuje.

### 2.1.3 Deflate

Deflate [7] je algoritem za stiskanje podatkov, ki združuje algoritma LZSS in Huffmanovo kodiranje. Razvit je bil za stiskanje splošnih podatkov v datoteke zip. Še vedno je najpogosteje uporabljen algoritem v datotekah zip. Uporablja se tudi v datotekah png. Uporaba je opcijska v datotekah pdf in pri komunikaciji z uporabo protokola SSL/TLS ali HTTP. Na formatu zip je osnovan tudi zapis v datoteke jar (java arhiv), odt (OpenOffice dokument) in swf (Shockwave Flash).

Vhodne podatke se najprej zakodira z algoritmom LZSS, pri katerem je dolžina ponovitev omejena na najmanj 3 in največ 258 znakov. Za en znak se uporabi en bajt podatkov. Dobljene ponovitve in znake se zakodira z dvema kanoničnima Hufmanovima kodoma. En se uporabi za kodiranje odmikov ponovitev, drugi pa za kodiranje dolžin in znakov, ki niso del ponovitev. Huffmanovo drevo za dolžine in znake se sestavi iz 286 vhodnih znakov. Prvih 256 jih predstavlja osnovne znake, ki niso del ponovitev. Ena znak predstavlja konec bloka, preostale pa se uporabli za zapis dolžin. Ker je možnih dolžin več kot je znakov za njihov zapis, se jih z istim znakom zapiše več. Za ločevanje med dolžinami, ki se jih zapiše z istim znakom, se uporabi dodatne, nekodirane bite, ki sledijo znaku. Število dodatnih bitov je odvisno od kodiranega znaka in je manjše za znake, ki predstavljajo krajše dolžine. Tako se za zapis krajših dolžin porabi manj dodatnih bitov. Prav tako se na kratko zapiše najdaljšo dolžino, ki se uporablja, če so v vhodnih podatkih daljši nizi enakih znakov. Za njen zapis se ne uporabi dodatnih bitov.

Za kodiranje odmikov se uporabi Huffmanovo drevo, sestavljeno iz 30 znakov. Tudi za določitev natančnega odmika se uporablja dodatne bite, podobno, kot pri kodiranju dolžin. Za razliko od dolžin ni najdaljši odmik

nič posebnega in nima manjšega števila dodatnih bitov. Natančno število dodatnih bitov za posamezen znak in katere dolžine in odmiki se zapišejo z istimi znaki je razvidno iz tabel v [7]. Če se v bloku uporabi samo en odmik, se tega zapiše s kodo 0 (dolgo 1 bit), koda 1 pa ni uporabljena.

Huffmanove kode v algoritmu deflate so omejene na 15 bitov. S tem, da je njihova dolžina omejena, se zagotovi, da se za zapis posamezne kode lahko uporabi fiksna količina pomnilnika.

Kodirani podatki so razdeljeni v bloke. V vsakem bloku se za kodiranje lahko uporabita drugačen Huffmanov kod. Prvi bit bloka pove, ali je trenutni blok zadnji. Nato sledita dva bita, ki določata tip kompresije v bloku. Možnosti so tri, četrta kombinacija bitov se ne uporablja.

Možen je nekodiran blok. To možnost uporabimo, če bi se zaradi kodiranja količina podatkov povečala. V tem primeru naslednje bite do konca bajta ignoriramo. Tako so nadaljnjni podatki poravnani na bajt. Sledi 16-bitno nepredznačeno število in negacija tega števila. To število pove koliko bajtov nekodiranih podatkov je v tem bloku. Nato so v bloku sami podatki. Ker njihovo dolžino zapišemo s 16 bitnim številom, jih je lahko največ 64 kilobajtov.

Naslednja možnost je blok kodiran s statičnim kodom. V specifikaciji [7] ga imenujejo Huffmanov kod, a gre le za vnaprej določen prefiksni kod. To možnost uporabimo predvsem za zapis kratkih blokov, kjer bi z zapisom Huffmanovega drevesa pridobili več podatkov, kot bi jih prihranili z bolj učinkovitim sproti izračunanim Huffmanovim kodom. Sledijo podatki zapisani z statičnim kodom, ki jim sledi znak za konec bloka.

Zadnja in najuporabnejša možnost je kodiranje z dinamičnim Huffmanovim kodom. Pri uporabi te možnosti na začetek bloka zapišemo Huffmanovi drevesi - drevo za kodiranje dolžin in znakov ter drevo za kodiranje odmikov. Dolžine kodnih zamenjav, ki sestavljajo drevesi, kodiramo s kanoničnim Hufffmanovim kodom. Poleg dolžin kodnih zamenjav, so v vhodni abecedi še trije dodatni znaki. Prvi znak predstavlja ponovitev prejšnjga znaka 3-6 krat, natančna vrednost pa je določena z dvema dodatnima bitoma. Pre-

ostala dva znaka uporabljamo za zapis večih zaporednih dolžin, enakih 0. Prvega, skupaj s tremi dodatnimi biti uporabimo za 3-10 ponovitev, drugega, skupaj s 7 dodatnimi biti, pa za 11-128 ponovitev. Nato izračunamo frekvence ponovitev vhodnih znakov, iz njih zgeneriramo novo Huffmanovo drevo, ki ima dolžino kodnih zamenjav omejeno na 7 bitov.

Na začetku zapisa drevesa so tri števila. Prvo, 5-bitno število sešteto z 257, pove koliko je zakodiranih dolžin drevesa za zapis dolžin in vhodnih znakov, ki niso del ponovitev. Če jih je manj kot 286, je to prvih nekaj dolžin, preostale so enake 0. Naslednje, 5-bitno število sešteto z 1, pove koliko je zakodiranih dolžin drevesa za zapis odmikov. Če jih je manj kot 30, so preostale enake 0. Zadnje, 4-bitno število sešteto s 4, pove koliko je dolžin, s katerimi je zapisano drevo za zapis drugih dveh Huffmanovih dreves. Če jih je manj kot 19, so preostale dolžine enake 0. Sledijo s 3-bitnimi števili zapisane dolžine kodnih zamenjav koda za zapis Huffmanovih dreves. Na koncu sta še s tem kodom kodirani Huffmanovi drevesi, najprej drevo za zapis dolžin in nekodiranih znakov, nato pa drevo za zapis odmikov. Po drevesih so zapisani zakodirani podatki, ki jim sledi znak za konec bloka.

## 2.2 Splošno procesiranje na grafičnih procesnih enotah

### 2.2.1 Arhitektura grafičnih procesnih enot

Prvotni namen grafičnih kartic je izrisovanje 3D grafike in temu so tudi strojno prilagojene. Za izisovanje predvsem kompleksnejših scen velja, da se enake računske operacije ponavljajo na veliki količini podatkov. Najprej je potrebno 3D koordinate oglišč trikotnikov, iz katerih je sestavljena scena, preslikati na dvodimenzionalen zaslon. Trikotnike, ki so povsen izven ekrana, izločimo, od tistih, ki pa so vidni le delno, obdržimo le vidni del. Nato trikotnike rasteriziramo – določimo katere piksle na zaslonu pokrivajo in izločimo tiste piksle na trikotniku, ki niso vidni, ker jih prekriva drug trikotnik. Na

koncu za vsak piksel na ekranu izračunamo barvo glede na osvetlitev in material, ki ga predstavlja.

Sodobni monitorji imajo večinoma več kot milijon pikslov, kompleksne scene pa imajo lahko hkrati vidnih več milijonov trikotnikov. Za tekočo uporabniško izkušnjo je potrebno izrisati novo sliko vsaj tridesetkrat, še raje pa šestdesetkrat na sekundo. Zato, da lahko grafične kartice dosežejo toliko izrisanih slik v eni sekundi, je njihova strojna zasnova ustrezno prilagojena.

Najpomembnejši del grafičnih kartic je grafična procesna enota. Osnovna enota grafičnih procesnih enot, ki lahko izvršuje ukaze, je procesni element (angl. *processing element*, *PE*). Vsak procesni element ima svoj blok registrov, ki pa jih je lahko mnogo več, kot je običajno za eno jedro v centralnih procesnih enotah. Več procesnih elementov skupaj sestavlja računsko enoto (angl. *compute unit*). Vsaka računska enota vsebuje blok lokalnega pomnilnika, kontrolno enoto in še nekatere druge enote, ki so pomembnejše za izrisovanje grafike, kot za splošno računanje. Ker si procesni elementi v računski enoti delijo eno kontrolno enoto, ne morejo hkrati izvrševati različnih ukazov [2, 1]. V novejših grafičnih karticah vsebujejo računske enote tudi manjšo količino predpomnilnika, ki se uporablja pri dostopih do glavnega pomnilnika. Grafične kartice, ki niso vgrajene v centralne procesorje, imajo svoj globalni pomnilnik, vgrajene pa si globalni pomnilnik delijo s pomnilnikom centralnega procesorja.

### 2.2.2 OpenCL

OpenCL [6] je odprtokodno ogrodje za pisanje paralelnih programov, ki ga vzdržuje skupina Khronos. Programski vmesnik (angl. *aplication programming interface*, *API*) je standardiziran, implementacija pa je prepuščena proizvajalcem računskih naprav. Na ta način je doseženo, da je mogoče na enak način delati z zelo raznolikimi računskimi napravami. OpenCL podpirajo mnoge naprave, predvsem večina novejših grafičnih kartic, centralni procesorji in še nekatere druge naprave.

Skupina Khronos določa programsko knjižnico za jezika C in C++, obsta-

jajo pa tudi neuradne knjižnice za večino razširjenih programskih jezikov. Od knjižnice program izve, za katere platforme so na računalniku prisotni gonilniki OpenCL. Prek knjižnice od gonilnikov izve, katere računske naprave so na voljo in specifikacije teh naprav. Glede na te podatke lahko program izbere eno ali več naprav na katerih bo poganjal program OpenCL – ščepec (angl. *kernel*). Za izbrane naprave ustvari kontekst OpenCL in ukazno vrsto ter zanje prevede ščepec. Za prevajanje poskrbi gonilnik OpenCL za napravo, za katero se prevaja. Obstaja tudi možnost uporabe vnaprej prevedenega ščepca, a podpora te opcije s strani proizvajalcev naprav ni obvezna.

Ob zagonu ščepca mu določimo dimenzije izvajanja. Dimenzije določajo skupno število niti – poimenovanih delovne enote (angl. *work item*) in kako so razdeljene v delovne skupine. Razdelitev niti v skupine je pomembna, ker se niti znotraj posamezne skupine izvajajo na isti računski enoti. Niti v isti delovni skupini si delijo tudi lokalni pomnilnik, prek katerega je mogoča hitrejša komunikacija med nitmi, kot prek globalnega pomnilnika.

Če je ščepec zagnan na centralnem procesorju, je ena računska enota eno procesorsko jedro [5]. Prevajalnik lahko vektorizira ščepec tako, da lahko eno procesorsko jedro z uporabo vektorske enote hkrati izvaja računske operacije večih delovnih enot. Zato je izvajanje najbolj učinkovito, če je v posamezni delovni skupini vsaj toliko niti, kot je v eni računski enoti procesnih elementov, oziroma kolikor je širina vektorske enote v centralnem procesorju.

Na grafični kartici eno delovno skupino izvaja ena računska enota. Posamezne delovne enote se izvajajo na procesnih elementih.

Niti v različnih delovnih skupinah se ne izvajajo nujno hkrati, na primer če je delovnih skupin več kot računskih enot. Hkrati se izvajajo le niti, ki so v eni podskupini (angl. *subgroup*), vrstni red izvajanja podskupin ni predpisan. Običajno se na grafičnih karticah začne izvajati druga podskupina, ko prva čaka, da bodo podatki, ki jih je zahtevala iz globalnega pomnilnika pripravljeni. Zato običajno večja števila delovnih enot v delovni skupini pripomorejo k hitrejšemu izvajanju. Velikost podskupine ni nastavljiva in je odvisna od strojne opreme, na kateri se izvaja ščepec. Zato ogrodje OpenCL

podpira sinhronizacijo niti znotraj posamezne delovne skupine. Na grafičnih karticah bi bila sinhronizacija vseh niti v ščepcu časovno zahtevna, zato je ogrodje OpenCL ne podpira.

Ščepec mora biti napisan v jeziku OpenCL C. To je jeziku C podoben programski jezik, osnovan na standardu C99 z nekaj omejitvami in razširitvami. Glavne omejitve so, da ne podpira kazalcev na funkcije, rekurzije in standardne knjižnice. Razširjen pa je z vektorskimi spremenljivkami in matematičnimi funkcijami, ki podpirajo skalarne in vektorske argumente. Podpira tudi funkcije za atomične dostope do pomnilnika in nove ključne besede, s katerimi se spremenljivkami določi v katerem nivoju pomnilnika so zapisane.

Poleg obveznih obstajajo tudi opcijske razširitve. Nekatere so predpisane s strani skupine Khronos, druge lahko definirajo proizvajalci računskih naprav sami. Ena izmed neobveznih razširitev je podpora 64-bitnih števil v zapisu s plavajočo vejico.

### 2.2.3 CUDA

CUDA [3] je ogrodje za programiranje grafičnih kartic, razvita s strani podjetja Nvidia. Zato za razliko od ogrodja OpenCL kot računske naprave podpira le grafične kartice podjetja Nvidia.

Uradno obstaja programska knjižnica CUDA za jezike C, C++ in Fortran, neuradne verzije pa tudi za večino drugih razširjenih programskih jezikov. Podobno kot pri ogrodju OpenCL, program od knjižnice izve katere računske naprave so na voljo. Razlika pa je, da ogrodje CUDA vedno podpira vnaprej prevedene programe. Pri prevajanju je potrebno le določiti računsko sposobnost (angl. *compute capability*) ciljnih naprav in ni potrebno ločeno prevajanje za vsako napravo.

Ščepci CUDA so napisani v jeziku CUDA C++. Osnovan je na jeziku C++ in ima manj omejitev kot jezik OpenCL C. Natančne zmogljivosti in omejitve so odvisne od računske sposobnosti ciljne grafične kartice.

Terminologija CUDA se nekoliko razlikuje od terminologije OpenCL. Delovno enoto imenujejo nit (angl. *thread*), delovni skupini se reče blok niti

(angl. *thread block*), podskupini ovoj (angl. *wrap*), za ščepec se uporablja ista beseda. Za globalni pomnilnik se uporablja isti izraz, lokalnemu pomnilniku rečejo deljeni, privatnemu pa lokalni.

Tudi za komponente grafične kartie se uporablja nekoliko drugačne izraze. Procesni element imenujejo pretočni procesor (angl. *streaming processor*, *SP*), računski enoti se reče pretočni multiprocesor (angl. *streaming multiprocessor*, *SM*).

### 2.2.4 Druge platforme za splošno procesiranje na grafičnih procesnih enotah

Preden sta obstajali ogrodji OpenCL in CUDA smo lahko programe za grafične kartice pisali samo kot senčilnike (angl. *shader*). Senčilniki so namenjeni prvenstveno izrisovanju, kljub temu pa je mogoče v obliki senčilnikov zapisati tudi splošne, z grafiko nepovezane programe. Novejše verzije knjižnic OpenGL in DirectX omogočajo uporabo računskih senčilnikov (angl. *compute shader*). Te naredijo pisanje z grafiko nepovezanih programov nekoliko lažje, kot je uporaba senčilnikov fragmentov (angl. *fragment shader*). Če se uporablja senčilnike fragmentov, je potrebno vse vhodne in izhodne podatke zapisati v teksture – polja niso podprta. Poleg tega senčilniki fragmentov ne dopuščajo eksplicitne uporabe lokalnega pomnilnika in sinhronizacije med nitmi v isti delovni skupini. Zato je težje optimizirati program - uporaba lokalnega pomnilnika je odvisna od optimizacij, ki jih naredi gonilnik grafične kartice. Vseeno senčilnike fragmentov lahko uporabljamo, če moramo podpirati tudi starejše grafične kartice, ki ne podpirajo računskih senčilnikov.

Za pisanje senčilnikov uporabljamo programski jezik GLSL, če uporabljamo knjižnico OpenGL ali jezik HLSL pri uporabi knjižnice DirectX. Oba sta podobna jeziku C in sta namenjena pisanju senčilnikov – predvsem za izrisovanje grafike.

Microsoftova tehnologija C++ AMP (angl. *accelerated massive parallelism* – pospešen masovni paralelizem) omogoča pisanje paralelnih programov za grafične kartice kot del običajnega programa C++. Funkcije, ki so name-

njene izvajanju na grafični kartici, se prevedejo v računski senčilnik DirectX, ki se izvaja na grafični kartici. Podpira sinhronizacijo niti v delovni skupini – ki jo imenujejo ploščica (angl. *tile*) – ne pa eksplicitne uporabe deljenega pomnilnika. Zaradi uporabe senčilnikov DirectX je uporaba omejena na operacijski sistem Windows.

Poleg naštetih obstajajo še druge platforme za računanje na grafičnih procesnih enotah. Večinoma so eksperimentalni projekti, ki prevedejo višjenivojski jezik v jezik OpenCL C ali jezik CUDA C++, lahko pa tudi v senčilnik. Večinoma ne dopuščajo tako natančnega nadzora nad izrabo strojne opreme kot ogrodji OpenCL in CUDA ter računski senčilniki, kar povzroči nekoliko počasnejše izvajanje programov, ki jih uporabljajo.

## 2.3 Obstoječe paralelne implementacije algoritma deflate za izvajanje na grafičnih karticah

Implementacij algoritma deflate za izvajanje na grafičnih karticah ni veliko, obstaja pa več implementacij njegovih komponent – algoritma LZSS in Huffmanovega kodiranja.

Implementacija algoritma LZSS, poimenovana CULZSS [12] je narejena v ogrodju CUDA. Algoritem so paralelizirali na dva načina. Prva verzija podatke razdeli med niti, vsaka nit stisne svoj del in zapiše rezultat v svoj del globalnega pomnilnika. Rezultate posameznih niti združi sekvenčno. Druga verzija razdeli podatke med delovne skupine. Med niti znotraj delovne skupine je delo razdeljeno tako, da vsaka nit išče ponovitve nizov, ki se začnejo z zaporednimi vhodnimi znaki.

Obe implementaciji za iskanje ponovitev uporabljata preiskovanje vseh podnizov v drsečem oknu. V testih so uporabljali velikost drsečega okna 128 bajtov. Testirani sta bili na grafični kartici GeForce GTX 480 na petih sklopih podatkov velikih 128 MB.

Odvisno od tipa podatkov je prva implementacija dosegla hitrost stiskanja 17-260 MB/s, druga pa 8-37 MB/s. Primerjali so jih z svojo sekvenčno in paralelno implementacijo istega algoritma ter programom bzip2, ki so se izvajali na centralnem procesorju Intel Core i7 920. Odvisno od tipa podatkov, so pohitritve med paralelno impementacijo na procesorju in grafični kartici do 160 kratne. V najslabšem primeru obe implementaciji dosegata približno enake hitrosti. V primerjavi z bzip2 so pohitritve še večje. A program bzip2 uporablja popolnoma drug kompresijski algoritem, ki dosega bistveno boljša kompresijska razmerja.

Algoritem GLZSS [15] je prav tako izvedba algoritma LZSS, narejena v ogrodju CUDA. Za iskanje ponovljenih nizov uporablja zgostitveno tabelo. Za vsako možno vrednost zgoščevalne funkcije je v tabeli prostor za en vnos. Tako je za vsak niz možno poiskati samo zadnji niz, katerega začetek se zgosti v isto vrednost. Tako je možno da je bil vnos za daljši ponovljen niz prepisan z vnosom za krajšega, ali celo z vnosom za niz, ki sploh ni ponovitev, ampak se le njegovi prvi 4 bajti zgostijo v isto vrednost.

Drseče okno, v katerem se išče ponovitve, ima nastavljivo velikost do 64 kilobajtov. Podatke razdelijo med podskupine. Na ta način dosežejo, da med nitmi ni potrebe po sinhronizaciji – niti, ki delajo na istem kosu podatkov, se vedno izvajajo hkrati. To je možno, ker ogrodje CUDA podpira samo eno platformo – grafične kartice Nvidia, ki imajo velikost podskupine vedno enako 32. V ogrodju OpenCL tak program verjetno ne bi pravilno deloval na vseh implementacijah, saj so lahko velikosti podskupin različne. Kako natančno si niti znotraj ovoja razdelijo delo, iz zapisanega v članku [15] ni popolnoma jasno.

Zgoraj opisani osnovni algoritem nadgradijo v verzijo, poimenovano označevalni GLZSS (angl. *GLZSS-tagging*). V tej verziji zmanjšajo število vejitev, v katerih bi lahko niti v isti podskupini izbrale različne veje izvajanja. Na ta račun ima označevalna verzija nekoliko slabše kompresijsko razmerje, a večjo hitrost stiskanja v primerjavi z osnovno.

Oba algoritma so testirali na grafični kartici GeForce GTX 590. Upo-

rabili so testne datoteke velikosti 32 do 200 megabajtov, z različno vsebino. Osnovni algoritem je dosegel hitrosti stiskanja od 110 do 160 MB/s. Označevalna verzija je dosegla od 140 do 210 MB/s.

Članek [16] opisuje implementacijo algoritma deflate v ogrodju CUDA. Osnovana je na implementaciji GLZSS, ki ji je dodano paralelno Huffmanovo kodiranje.

Histograme vhodnih znakov za Huffmanovo kodiranje izračuna sproti, ko kodira vhodne podatke z LZSS. Pri gradnji Huffmanovih dreves histograme sortira z uporabo paralelnega bitoničnega sortirnega algoritma (angl. *bitonic sorting algorithm*). S sortiranimi histogrami lahko zgradi Huffmanovi drevesi v linearnem času glede na število vhodnih znakov.

Huffmanovo kodiranje poteka tako, da vsaki niti dodeli en simbol. Ta je lahko referenca na ponovitev ali nekodiran znak. Nit poišče ustrezno kodno zamenjavo in če gre za referenco, združi zamenjavi, s katerima je predstavljena. Niti izračunajo odmike, na katerih se začnejo posamezne kodne zamenjave in jih zapišejo z atomičnimi operacijami v lokalni pomnilnik. Nato prepišejo podatke iz lokalnega pomnilnika v globalnega.

Algoritem je testiran na enaki strojni opremi in podatkih kot GLZSS. Dosega hitrosti stiskanja med 125 in 160 MB/s.

# Poglavje 3

# Izvedba

## 3.1 Sekvenčni algoritem

Pri iskanju ponovitev se izčrpno pregledovanje vseh možnih podnizov v drsečemu oknu izkaže za računsko prezahtevno. Zato uporabljamo zgoščevalno tabelo z veriženjem. Prvih nekaj znakov niza, katerega ponovitve iščemo, uporabimo kot ključ v zgoščevalni tabeli, vrednost v njej pa je indeks, ki kaže na najbližji niz v drsečemu oknu, ki se začne z istimi znaki.

Število prvih znakov niza, ki se uporablja kot ključ v zgoščevalni tabeli, je nastavljivo. Naša implementacija število omeji na največ štiri znake. Z večanjem števila znakov zmanjšujemo število nizov, ki se jih primerja s trenutnim nizom, hkrati pa se veča poraba pomilnika za zgoščevalno tabelo. Tabela mora biti velika $2^{8 \times \textit{število znakov}}$. V testiranju so uporabljeni trije znaki. Ta vrednost omogoča pregledovanje majhnega števila nizov in ne poveča pretirano tabele. Hkrati je to najmanjše število znakov, ki se ga pri uporabi algoritma deflate lahko zapiše kot ponovitev.

Za iskanje prejšnjih ponovitev nizov z istim začetkom uporabljamo verižno tabelo. V njej je za vsak znak v drsečem oknu zapisan kazalec na prejšnji niz, ki se začne z istimi znaki, kot niz, ki se začne na trenutnem znaku. Verižna tabela bi lahko imela enako število vnosov kot drseče okno. V naši implementaciji ima namesto tega toliko vnosov, kot je vhodnih podatkov. S

tem se močno poveča poraba pomnilnika, a se tudi pohitri algoritem, saj se lahko za iskanje prejšnjega niza z istim začetkom uporabi kar indeks prvega znaka trenutnega niza. S krajšo verižno tabelo pa bi bilo potrebno indeks izračunati.

Z uporabo zgoščevalne in verižne tabele dosežemo, da ni potrebno izčrpno pregledovanje drsečega okna za ponovitve, ampak lahko pregledamo le nize, ki se začnejo s tremi enakimi znaki kot niz, katerega ponovitve iščemo. S tem močno zmanjšamo skupno količino dela.

Ko iščemo ponovitve nekega niza, najprej preberemo prve tri znake tega niza. Na tri znake, ki so dolgi vsak osem bitov, lahko gledamo kot na 24-bitno celo število. To število uporabimo kot indeks v zgoščevalni tabeli, da najdemo indeks zadnjega niza z enakimi prvimi tremi znaki.

Najprej preverimo, če je indeks veljaven – če kaže na znak, ki je v drsečem oknu. Če ni veljaven, za trenutni niz ni več ponovitev v drsečem oknu. Če kaže na znak v drsečem oknu, preverimo koliko znakov ima niz, ki se začne s tem znakom, enakih s trenutnim nizom. Tudi v primeru, če je število skupnih znakov manjše od tri, indeks ni veljaven. Če so vsaj trije enaki znaki, imamo veljaven indeks, ki ga lahko uporabimo, da iz ustreznega mesta v verižni tabeli preberemo indeks prejšnje ponovitve z enakimi prvimi tremi znaki. Nato z novim indeksom ponovimo postopek. Ko naletimo na neveljaven indeks, zapišemo niz z največ enakimi znaki kot referenco.

Ko za neko mesto v vhodnih podatkih najdemo ponovitev, jo shranimo v tabelo ponovitev kot par (*odmik*, *dolžina*), kjer je odmik število znakov med začetkoma ponovljenih podnizov, dolžina pa število znakov, ki se ponovijo. Če ugotovimo, da v drsečem oknu ponovitve ni, zapišemo v tabelo ponovitev par (*0*, *trenutni znak*). Ker 0 ni veljaven odmik, vemo, da je na trenutnem mestu v tabeli ponovitev zapisan znak, ne pa ponovitev – na ta način lahko v isto podatkovno strukturo zapišemo tako ponovitve, kot znake.

Sproti ko shranjujemo znake, gradimo tudi histograma za izgradnjo Huffmanovega koda. Z vsak znak ali dolžino ponovitve iz ustreznih tabel izvemo, kateri vhodni znak se uporabi za Huffmanovo kodiranje. Ker je možnih od-

mikov več (32000), se za odmike vhodni znak izračuna sproti. Nato se poveča ustrezno polje v tabeli, ki predstavlja histogram dolžin in znakov, ter če gre za ponovitev, tudi ustrezno polje v tabeli, ki predstavlja histogram dolžin.

Nato iz obeh histogramov izračunamo najprej dolžine Huffmanovih kod, nato pa iz dolžin same kode. Izračun dolžin poteka po postopku, opisanem v poglavju 2.1.2. V vsakem koraku poiščemo dve najmanjši, neničelni vrednosti iz histograma. V novo polje histograma zapišemo njuno vsoto, v ločene tabele pa zapišemo, da sta vrednosti že uporabljeni in iz katerih dveh vrednosti je dobljena vsota. To ponavljamo, dokler ni v histogramu le še ene neuporabljena vrednost. Tej vrednosti priredimo dolžino 0. Nato za vsako sestavljeno vrednost iz histograma pogledamo, iz katerih dveh vrednosti je sestavljena in jima priredimo dolžino, za 1 daljšo od dolžine trenutne vrednosti. Tako dobimo dolžine kodnih zamenjav za vse neničelne vrednosti iz histograma. Zamenjavam za znake, ki se v vhodnih podatkih ne pojavijo, priredimo dolžino 0.

S tem postopkom dobimo dolžine Huffmanovih kod, ki pa še ne zadoščajo nujno pogoju, da imajo dolžino največ 15 bitov. Če obstaja vsaj ena daljša koda, pripadajoče vrednosti v histogramu podvojimo in ponovimo postopek izračuna dolžin kod. Ponavljamo dokler niso vse kode dovolj kratke.

Za zapis kodnih zamenjav uporabljamo 16 bitna cela števila. Zamenjava se začne na najmanj pomembnem bitu (angl. least significant bit). Koliko bitov predstavlja kodno zamenjavo, zapišemo v ločeno spremenljivko. Najprej izračunamo histogram uporabljenih dolžin kod. Za vsako dolžino izračunamo prvo kodo te dolžine. Nato vsakemu vhodnemu znaku določimo naslednjo kodo ustrezne dolžine.

V izhodno tabelo se najprej zapišeta Huffmanovi drevesi. Na enak način kot zgoraj zračunamo kod za zapis dolžin kodnih zamenjav, le da je dolžina kodne zamenjave omejena na 7 bitov. Na izhod zapišemo, s koliko dolžinami je zapisano posamezno Huffmanovo drevo, nato v ustreznem vrstnem redu zapišemo 3 bitna števila – dolžine kod za zapis Huffmanovih dreves. Na koncu sledijo še s tem kodom zapisane dolžine, ki predstavljajo Huffmanovi

drevesi za zapis vhodnih podatkov.

S tem je glava bloka zaključena. Nato znake in ponovitve iz tabele ponovitev zakodiramo s Huffmanovih kodom in zapišemo v izhodno tabelo. Za znake iz ustreznih tabel preberemo dolžino kode in samo kodo ter kodo zapišemo v izhodno tabelo. Za ponovitve najprej zapišemo dolžino, nato pa na skoraj enak način odmik. Iz ustreznih tabel preberemo, s katerim znakom se kodira trenutna dolžina, in koliko dodatnih bitov potrebujemo. Za izbrani znak preberemo dolžino kode in kodo ter kodo zapišemo v izhodno tabelo. Nato zapišemo še dodatne bite. Na koncu bloka zapišemo še na enak način zakodiran znak za konec bloka.

## 3.2 Paralelizacija

Parareliziratí je smiselno tiste dele algoritma, ki se izvajajo dolgo časa. V algoritmu deflate se največ časa porabi za primerjanje enakosti nizov pri iskanju ponovitev v algoritmu LZSS. Pri datotekah, ki jih algoritem LZSS ne more dobro stisniti, se skoraj enako časa porabi za Huffmanovo kodiranje. Zato je paralelizacija teh dveh delov najpomembnejša, med tem ko posodabljanje zgoščevalne in verižne tabele manj pomembno, gradnja Huffmanovih dreves in izračun kanoničnega koda pa sta skoraj nepomembna.

Pri paralelizaciji za grafične procesne enote je pomembno, da lahko niti večino časa hkrati izvajajo enake ukaze.

### 3.2.1 Možni načini paralelizacije

Najbolj preprost način paralelizacije algoritma deflate bi bil vnaprej razdeliti vhodne podatke na veliko število blokov in dodeliti vsaki niti enega ali več blokov. Tak način paralelizacije se uporablja v implementacijah, ki tečejo na centralnem procesorju. Prednost tega pristopa je, da paraleliziramo celoten algoritem – noben del ni tak, da bi ga lahko izvajala samo ena nit, ostale pa bi morale čakati, da konča. Poleg tega sinhronizacija med nitmi ni potrebna.

Vendar tak pristop ni najbolj primeren za grafične kartice. Na grafičnih

karticah običajno poganjamo tisoče niti hkrati, da popolnoma izkoristimo njihove strojne zmogljivosti. To pomeni, da bi lahko grafične kartice dobro izkoristili samo z večjimi količinami podatkov, ali če bi podatke razdrobili na majhne bloke, s čimer bi poslabšali kompresijsko razmerje. Druga slabost tega pristopa se pokaže, če blokov ni možno enako učinkovito stisniti. V tem primeru se pogosto dogaja, da niti v programu izbirajo različne vejitve in izvajajo zanke različno število iteracij. To negativno vpliva paralelnost, ker niti znotraj delovne skupine ne morejo hkrati izvajati različnih ukazov. Tudi dostopi do glavnega pomnilnika so naključni, kar onemogoči uporabo usklajenih (angl. *coalesced*) [2] dostopov in zmanjša hitrost prenosa podatkov iz glavnega pomnilnika.

Drug možen pristop je, da bloke razdelimo med delovne skupine in ena delovna skupina skupaj stiska blok. Prednost tega pristopa je, da je potrebnih manj blokov, da izkoristimo strojne zmogljivosti grafičnih kartic in lahko niti znotraj posamezne delovne skupine večji del časa izvajajo iste ukaze. Slabost tega pristopa je, da je potrebno vsak korak algoritma paralelizirati ločeno.

### Paralelizacija Huffmanovega kodiranja

Paralelizacija kodiranja s Huffmanovim kodom je relativno preprosta. Vsaka nit iz delovne skupine prebere en vnos iz tabele ponovitev. S tem, da niti hkrati berejo zaporedne vnose iz globalnega pomnilnika, dosežemo, da se na napravah, ki to podpirajo, uporabijo usklajeni dostopi do globalnega pomnilnika. Vsaka nit na enak način kot sekvenčni algoritem zakodira ponovitev. Tabele, ki jih pri tem uporabljamo, so dovolj majhne, da jih lahko hranimo v lokalnem pomnilniku, do katerega so dostopi hitrejši, kot do globalnega. Pri tem gredo lahko niti po dveh različnih poteh izvajanja: lahko kodirajo ponovitev ali pa znak. To ni idealno, ker znotraj procesne enote procesni elementi ne morejo hkrati izvajati različnih vej, ampak se morata veji izvesti ena za drugo, a se temu ni mogoče na preprost način izogniti. Nato je potrebno zapisati izračunane kode v tabelo z zakodiranimi podatki.

Ker so kode različnih dolžin, se pogosto v en bajt zapiše več kot eno kodno zamenjavo. Niti kodne zamenjave zapisujejo hkrati, zato je potrebno uporabiti atomične operacije. Ker so atomične operacije nad lokalnim pomnilnikom hitrejše kot nad globalnim, je smiselno Huffmanove kode zapisovati v del lokalnega pomnilnika. Ko se ta napolni, niti znotraj delovne skupine skopirajo podatke v glavni pomnilnik. Pri tem ponovno dostopajo do zaporednih lokacij in s tem uporabijo usklajene dostope do glavnega pomnilnika, če jih računska naprava podpira.

#### Paralelizacija algoritma LZSS znotraj delovne skupine

Za paralelizacijo algoritma LZSS znotraj delovne skupine je možnih več opcij. Ena možnost je, da paraleliziramo le najbolj notranjo zanko – primerjanje enakosti nizov. Vsaka nit primerja enakost enega bajta – niti dostopajo do zaporednih lokacij globalnega pomnilnika, kar omogoči uporabo usklajenih dostopov. Ko vsaka nit ugotovi enakost znakov na svojem indeksu, si morajo izmenjati informacijo o tem, kateri indeksi so enaki. To lahko storijo vsakič po tem, ko posamezna nit primerja dva znaka, ali pa najprej preverijo enakost vseh znakov do največje dovoljene dolžine ponovitve.

V obeh primerih dolžino ponovitve izračunajo skupaj, z uporabo lokalnega pomnilnika. Pri prvi možnosti vsaka nit v tabelo v lokalnem pomnilniku na indeks, enak zaporedni številki niti znotraj delovne skupine, zapiše številko, ki je odvisna od tega, ali sta znaka, ki jih je primerjala enaka. Če nista enaka, zapiše zaporedno številko znakov, ki jih primerja. Če sta enaka, zapiše številko, večjo od števila niti v delovni skupini. Nato niti nad tabelo izvedejo paralelno redukcijo z uporabo operacije manjši (angl. *min*). S tem dobijo na prvem mestu v tabeli najmanjši indeks niti, ki nima enakih znakov, ali število, večje od števila niti, če imajo vse niti enaka znaka. Če dobijo zaporedno številko niti, je ta enaka dolžini ponovitve; v nasprotnem primeru izvedejo še eno iteracijo.

V drugi možnosti vsaka nit v tabelo zapiše indeks prvih elementov v ponovitvi, ki nista bila enaka. Niti nad tabelo izvedejo pararelno redukcijo z

uporabo operacije manjši in na prvem mestu dobijo indeks prvih znakov, ki se nista primerjala kot enaka. Ker indeksiramo z osnovo 0 (angl. *zero-based indexing*) je to hkrati dolžina ponovitve.

Druga možnost v primeru kratkih ponovitev opravi nekaj odvečnega dela, a pri dolgih ponovitvah opravi isto delo z manj sinhronizacije med nitmi. Tudi v prvi možnosti opravimo nekaj dodatnega dela, ki v sekvenčnem algoritmu ni potrebno. Če dolžina ponovitve ni deljiva s številom niti, nekaj niti naredi primerjave, ki jih sekvenčni algoritem ne bi.

Naslednja možnost bi bila, da vsaka nit primerja niz, ki se začne s trenutnim znakom, z drugim nizom iz drsečega okna. Ko iščemo ponovitve niza, najprej iz zgoščevalne in verižne tabele preberemo vse lokacije prejšnjih nizov z istim začetkom kot ga ima trenutni niz. Tega dela brez spremembe podatkovnih struktur, v katerih hranimo prejšnje nize z enakimi začetki, ne moremo paralelizirati. Nato vsaka nit primerja trenutni niz z enim od nizov z istim začetkom v drsečem oknu. Ko vsaka nit preveri dolžino svoje ponovitve, niti v lokalni pomnilnik zapišejo dolžine in z redukcijo z operacijo večji (angl. *max*) poiščejo najdaljšo ponovitev. V tem primeru niti ne opravljajo nič več dela, kot sekvenčni algoritem, sinhronizacija med nitmi pa tudi ni tako pogosta, kot pri prejšnjemu načinu.

Vendar so lahko ponovljeni nizi različno dolgi in morajo niti, ki preverijo krajše ponovitve, nato čakati na tiste z daljšimi – v isti delovni skupini niti ne morejo hkrati izvajati različnih ukazov. Do še slabše izkoriščenosti strojne opreme pride, če je nizov iz istim začetkom, kot ga ima trenutni niz v drsečem oknu, manj, kot je število niti v delovni skupini. V tem primeru del niti čaka, dokler ostale niti ne najdejo najdaljše ponovitve trenutnega niza.

Možen je tudi pristop, ki združuje prejšnja dva. Na začetku vsaka nit primerja trenutni niz z drugim iz drsečega okna. Ko polovica niti s svojimi primerjavami konča, lahko primerjave, ki še niso zaključene, delata po dve niti skupaj. Ko je vedno več primerjav končanih, lahko dela na eni primerjavi vedno več niti. Niti, ki primerjajo isti niz, se po vsaki primerjavi uskladijo enako, kot se vse niti v prvem pristopu. Nato s še eno paralelno reduk-

cijo seštejemo število niti brez dela in če jih je dovolj, za naslednjo iteracijo povečamo število niti, ki delajo eno primerjavo.

Če se uporabi ta pristop, niti manj časa čakajo brez dela, kot pri uporabi prejšnjega in opravijo manj nepotrebnega dela kot pri uporabi predprejšnjega. Vendar to zahteva večjo količino komunikacije med nitmi.

Z vsemi pristopi k deljenju dela znotraj delovne skupine paraleliziramo zanko, v kateri porabimo največ časa, preostali algoritem pa se v vsaki delovni skupini izvaja v eni niti.

#### Izbira

Kljub analizi pristopov je nemogoče vnaprej zanesljivo vedeti, kateri bi bil najbolj učinkovit. Vsak ima določene prednosti in slabosti. Katere najbolj vplivajo na končno hitrost izvajanja, lahko preverimo samo tako, da implementiramo in izmerimo hitrost izvajanja na različnih grafičnih karticah. Ker se strojne zasnove krafičnih kartic različnih proizvajalcev in serij razlikujejo, je možno, da bi se za različne kartice izkazali različni pristopi kot najbolj optimalni.

Izbral sem pristop z razdelitvijo blokov med delovne skupine, v kateri niti znotraj skupine skupaj primerjajo enakost nizov. Ocenil sem, da je ta najprimernejši za izvajanje na grafičnih procesnih enotah. Pristop paralelizira najbolj kritični del kode tako, da niti ta del opravljajo skupaj, hkrati pa uporabljajo usklajene dostope do pomnilnika. Implementiral sem oba opisana pristopa h komunikaciji med nitmi.

# Poglavje 4

# Testiranje in rezultati

Meritve smo izvajali na datotekah iz prosto dostopnega korpusa [8]. Sestavljen je iz šestih datotek. Datoteka *sources* vsebuje izvorno kodo iz odprtokodnih projektov Linux in GCC. Datoteka *pitches* vsebuje zapise tonov iz prostodostopnih datotek MIDI. Datoteka *proteins* vsebuje zapise zapise beljakovin, v katerih je vsaka aminokislina predstavljena z eno črko. Datoteka *dna* vsebuje zapise DNK. Datoteka *english* vsebuje angleška besedila. Datoteka *dblp.xml* vsebuje bibliografske zapise računalniških revij v formatu xml.

Vsaka meritev smo izvedli desetkrat. Rezultati so povprečne vrednosti s standardnim odklonom. Ker so datoteke iz korpusa dolge, smo za nekatere teste uporabili le začetni del vsake datoteke.

## 4.1 Testiranje sekvenčne implementacije

Za sekvenčne teste smo uporabili računalnik s procesorjem Intel Core i5 2500. Primerjavo z obstoječo implementacijo algoritma deflate smo naredil na datoteki *sources*. Pri stiskanju celotne datoteke naša implementacija doseže hitrost 12,7 $\pm$ 0,1 MB/s in kompresijsko razmerje 0,235.

Za primerjavo smo uporabili program 7-zip z nastavitvami stiskanja v datoteko zip, z uporabo algoritma deflate. Preostale nastavitve smo nastavili

na privzete vrednosti. Ta doseže hitrost 7,46 ± 0,01 MB/s in kompresijsko razmerje 0,216. Naša implementacija je skoraj dvakrat hitrejša, a dosega slabše kompresijsko razmerje.

Če kompresijsko razmerje in hitrost primerjamo tako, da je dvakrat večja hitrost enakovredna deset odstotkov boljšemu razmerju [4], sta implementaciji primerljivi.

Sekvenčni algoritem smo profilirali z orodjem, vgrajenim v razvojno okolje Visual Studio. Na datoteki *sources* porabi iskanje ponovitev 76,5 % časa, posodabljanje zgostitvene in verižne tabele 7,1 % časa, izgradnja Huffmanovega drevesa 1,7 % in Huffmanovo kodiranje 5,4 % časa. Preostali čas se porabi za izvajanje drugih operacij. Očitno je kodiranje z algoritmom LZSS najbolj računsko zahteven del algoritma deflate. Huffmanovo kodiranje porabi bistveno manj časa, izgradnja dreves pa je skoraj zanemarljiva.

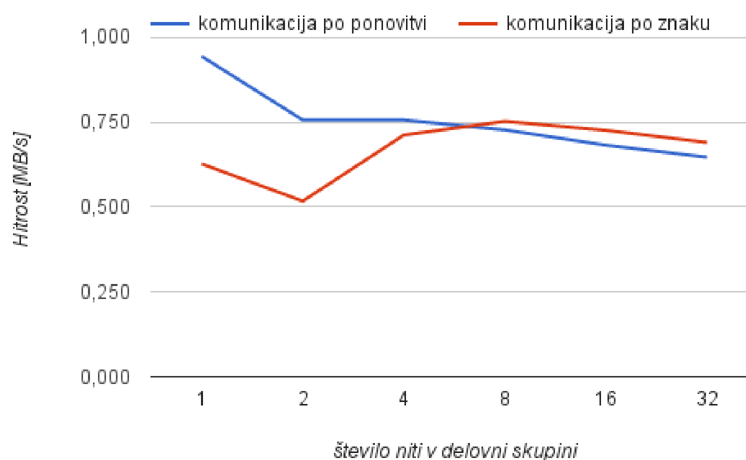
## 4.2 Testiranje paralelne implementacije

Paralelno implementacijo smo testirali na grafični kartici Nvidia GeForce GTX 550 Ti. Najprej smo testirali, kako se hitrost izvajanja algoritma spreminja s številom niti v delovni skupini. Testirali smo obe možnosti za komunikacijo med nitmi – po vsakem znaku in enkrat na vsako ponovitev. Za test smo uporabili prvih 8 MB datoteke *sources*, ki smo jo stiskali z 32 delovnimi skupinami. Rezultati so v tabeli 4.1 in na grafu na sliki 4.1.

Komunikacija enkrat na ponovitev doseže večjo hitrost. Vendar je to pri eni niti v delovni skupini, z večanjem števila niti pa se hitrost zmanjšuje. Razlog za upočasnitev z večanjem števila niti je, da s tem zmanjšujemo delež tistih niti, ki računajo operacije, ki so dejansko potrebne. Na primer, če je dolžina ponovitve pet znakov, bo prvih pet niti v delovni skupini primerjalo znake ponovitve, šesta bo ugotovila, da se niza v šestem znaku ne ujemata več, vse ostale pa bodo primerjale znake, ki jih v sekvenčni implementaciji ne bi bilo potrebno. Hkrati se povečuje količina potrebne komunikacije med nitmi.

| velikost delovne skupine | hitrost s komunikacijo po ponovitvi [MB/s] | hitrost s komunikacijo po znaku [MB/s] |
|---|---|---|
| 1 | 0,943±0,001 | 0,627±0,001 |
| 2 | 0,756±0,001 | 0,517±0,000 |
| 4 | 0,756±0,001 | 0,712±0,001 |
| 8 | 0,727±0,021 | 0,752±0,001 |
| 16 | 0,682±0,001 | 0,726±0,001 |
| 32 | 0,647±0,001 | 0,690±0,001 |

Tabela 4.1: Hitrost stiskanja v odvisnosti od velikosti delovne skupine in pogostosti komunikacije




Slika 4.1: Hitrost stiskanja v odvisnosti od velikosti delovne skupine in pogostosti komunikacije

Komunikacija po vsakem znaku dosega največjo hitrost pri osmih nitih v delovni skupini. Pri večjem številu niti se hitrost zmanjša. Z večanjem števila niti se zmanjša količina dela, ki ga opravi posamezna nit. A hkrati se

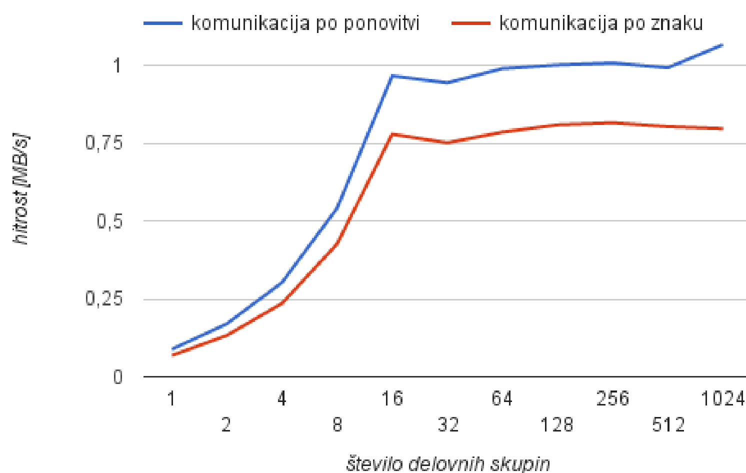
| število delovnih skupin | hitrost s komunikacijo po ponovitvi [MB/s] | hitrost s komunikacijo po znaku [MB/s] |
|---|---|---|
| 1 | 0,088±0,000 | 0,068±0,000 |
| 2 | 0,170±0,000 | 0,133±0,000 |
| 4 | 0,302±0,003 | 0,235±0,000 |
| 8 | 0,541±0,000 | 0,426±0,000 |
| 16 | 0,965±0,001 | 0,778±0,025 |
| 32 | 0,943±0,000 | 0,751±0,000 |
| 64 | 0,989±0,040 | 0,785±0,001 |
| 128 | 1,000±0,003 | 0,808±0,002 |
| 256 | 1,007±0,039 | 0,815±0,027 |
| 512 | 0,992±0,004 | 0,803±0,003 |
| 1024 | 1,065±0,072 | 0,796±0,003 |

Tabela 4.2: Hitrost stiskanja v odvisnosti od števila delovnih skupin in pogostosti komunikacije

povečuje čas, ki ga niti porabijo za medsebojno komunikacijo pri združevanju rezultata. Teoretično se čas potreben za izračun paralelne redukcije med nitmi povečuje z logaritmom števila niti v delovni skupini. Izkaže se, da je pri osmih nitih količina dela, ki ga opravlja posamezna nit, še dovolj majhna, hkrati pa komunikacije še ni preveč, da dosežemo največjo hitrost.

Pri testiranju, kako na hitrost vpliva število delovnih skupin je število niti v posamezni skupini tako, kot se je izkazalo za najboljše. Pri komunikaciji po vsakem znaku je to osem, pri komunikaciji enkrat na ponovitev ena. Rezultati so v tabeli 4.2 in na grafu na sliki 4.2.

Testna grafična kartica ima štiri računske enote, ki izvajajo ukaze v štiristopenjskem cevovodu. Vsaka računska enota v cevovodu izmenično izvršuje ukaze iz štirih delovnih skupin. Zato se z večanjem števila delovnih skupin do 16 skoraj linearno povečuje hitrost stiskanja. Šele pri 16 in več delovnih skupinah izkoristi vse strojne zmogljivosti računskih enot.




Slika 4.2: Hitrost stiskanja v odvisnosti od števila delovnih skupin in pogostosti komunikacije

Tudi za večja števila je opazno manjše povečanje hitrosti s povečevanjem števila delovnih skupin. Do tega pride, ker se delovnim skupinam vnaprej dodeli podatke, ki jih kompresirajo. Možno je, da dobi ena delovna skupina slabše stisljive podatke, kot druga. Ker je čas izvajanja odvisen od podatkov, lahko ena skupina za stiskanje porabi več časa – s tem se dalj časa izvaja ščepec, kljub temu, da na koncu čaka le eno delovno skupino.

Če podatke razdelimo med več skupin, ima vsaka skupina manj podatkov, skupin pa je več. V tem primeru lahko grafična kartica dinamično določa, katera računska enota bo izvedla katere delovne skupine, s čimer se izvajanje bolje razporedi med enote.

## 4.3 Primerjava sekvenčne in paralelne implementacije

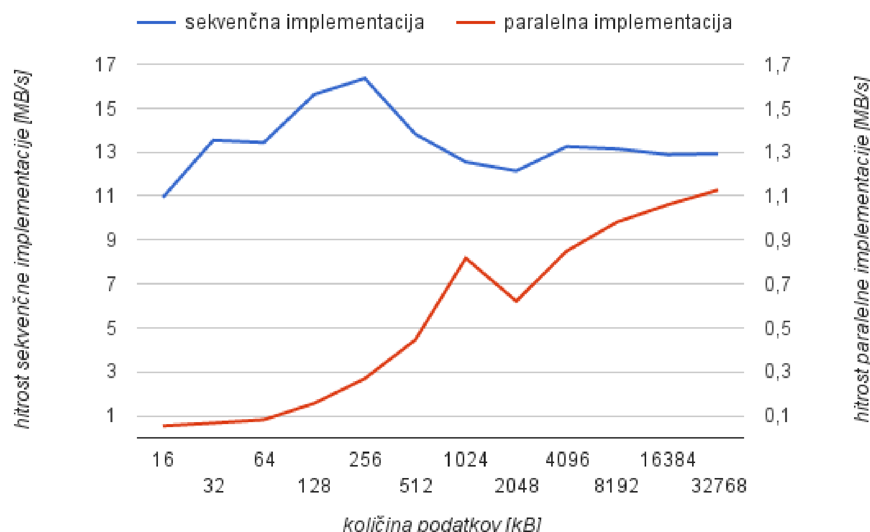
Hitrost stiskanja je odvisna tudi od količine podatkov. Test smo izvedli na datoteki *sources*. S sekvenčno implementacijo smo primerjali paralelno implementacijo s komunikacijo enkrat na ponovitev, ki se je izkazala za hitrejšo. Pri testiranju paralelne implementacije smo uporabili najhitrejšo kombinacijo nastavitev – 1 nit na delovno skupino in 1024 delovnih skupin. Iz grafa na sliki 4.3 se vidi, da se pri manjši količini podatkov hitrost povečuje z večanjem količine podatkov. Sekvenčni algoritem doseže največjo hitrost stiskanja pri 256 kB, pri paralelnemu pa hitrost narašča do 32 MB. Razlog za znižanje hitrosti sekvenčnega algoritma pri več kot 256 kB podatkov je verjetno v samih podatkih – hitrost kompresijskih algoritmov je v splošnem zelo odvisna od podatkov. Izgleda, da del podatkov po 256 kB v datoteki *sources* upočasni sekvenčni algoritem, podatke malo pred to mejo pa lahko zelo hitro stisne.

V tabeli 4.3 so poleg hitrosti sekvenčnega in paralelnega algoritma tudi kompresijska razmerja. Paralelni algoritem podatke vedno stisne na popolnoma enak način kot sekvenčni, tako da dosega enako kompresijsko razmerje. Odvisnost kompresijskega razmerja od količine podatkov je vidna na grafu na sliki 4.4. Kompresijsko razmerje niha dokaj naključno, kar je posledica variacij v stisljivosti vhodnih podatkov.

Razlog, da smo testirali le do velikosti podatkov 32 MB je v funkcionalnosti operacijskega sistema Windows, poimenovani detekcija in okrevanje po prekoračitvi časovnih omejitev (angl. *timeout detection and recovery, TDR*). Ta komponenta operacijskega sistema zazna, če se gonilnik grafične kartice neha odzivat in ga ponovno zažene. Problem nastane, ker se grafični gonilnik med tem, ko se na grafični kartici izvaja ščepec, ne odziva na klice operacijskega sistema. Zato ta ponovno zažene gonilnik, s čimer prekine izvajanje ščepca. Privzeta nastavitev za časovno omejitev je dve sekundi, a jo je mogoče s spremembo določenih vrednosti v registru operacijskega sistema

| količina podatkov [kB] | kompresijsko razmerje | hitrost sekvenčne implementacije [MB/s] | hitrost paralelne implementacije [MB/s] |
|---:|---:|---:|---:|
| 16 | 0,279 | 10,93$\pm$4, 03 | 0,054$\pm$0, 005 |
| 32 | 0,296 | 13,54$\pm$2, 68 | 0,067$\pm$0, 000 |
| 64 | 0,251 | 13,43$\pm$1, 50 | 0,082$\pm$0, 000 |
| 128 | 0,216 | 15,62$\pm$0, 00 | 0,157$\pm$0, 005 |
| 256 | 0,208 | 16,35$\pm$0, 50 | 0,269$\pm$0, 009 |
| 512 | 0,228 | 13,82$\pm$0, 34 | 0,446$\pm$0, 002 |
| 1024 | 0,255 | 12,54$\pm$0, 18 | 0,817$\pm$0, 021 |
| 2048 | 0,266 | 12,14$\pm$0, 07 | 0,622$\pm$0, 002 |
| 4096 | 0,243 | 13,25$\pm$0, 11 | 0,850$\pm$0, 006 |
| 8192 | 0,238 | 13,14$\pm$0, 04 | 0,982$\pm$0, 001 |
| 16384 | 0,248 | 12,88$\pm$0, 05 | 1,060$\pm$0, 024 |
| 32768 | 0,247 | 12,92$\pm$0, 11 | 1,127$\pm$0, 002 |

Tabela 4.3: Hitrosti stiskanja v odvisnosti od količine podatkov in implementacije

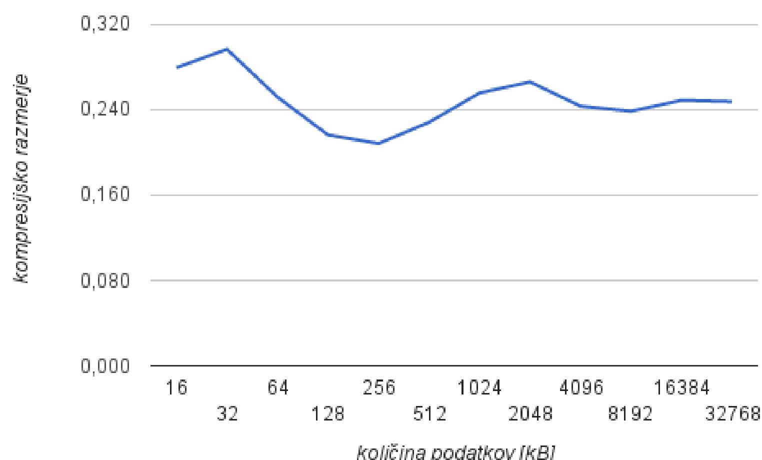



Slika 4.3: Hitrosti stiskanja v odvisnosti od količine podatkov in implementacije

spremeniti. Vendar pri velikih vrednostih operacijski sistem te nastavitve ne upošteva, ampak že prej ponovno zažene gonilnik. Zato se je ščepec uspešno zaključil le, če je izvajanje končal v približno dveh minutah.

Zadnji test je bil primerjava izvajanja paralelnega in sekvenčnega algoritma na različnih vhodnih podatkih. Testirali smo jih na prvih osmih megabajtih vsake datoteke iz korpusa. Kompresijska razmerja in hitrosti izvajanja so v tabeli 4.4. Testirali smo učinkovitejšega od paralelnih algoritmov – s komunikacijo med nitmi znotraj delovne skupine enkrat na vsako ponovitev. Nastavili smo 1024 delovnih skupin z eno nitjo v vsaki. Najboljše kompresijsko razmerje dosežeta na podatkih v formatu xml. Do tega pride, ker so v tem formatu začetne in končne značke skoraj enake in se pogosto ponavljajo. Za najmanj sitisljiva se izkažejo angleška besedila. Oba algoritma dosežeta največjo hitrost na podatkih v formatu xml. Sekvenčni algoritem doseže najmanjšo hitrost na zapisih DNK, paralelni pa na zapisih beljakovin.

Za vse vrste podatkov je sekvenčni algoritem za vsaj faktor 10 hitrejši od




Slika 4.4: Kompresijsko razmerje v odvisnosti od količine podatkov

paralelnega. Razlog za to je, da je delo razdeljeno med niti tako, da je v obeh paralelnih implementacijah za združevanje delnih rezultatov niti potrebnega preveč dodatnega dela in je hitreje, če v vsaki delovni skupini dela le ena nit. Zato so računske enote na grafični kartici slabo izkoriščene. Vsaka ima 32 računskih elementov, a se uporablja le en na enkrat. Posamezen računski element je bistveno manj zmogljiv od enega jedra centralnega procesorja, zato se sekvenčni algoritem izvede hitreje.

## 4.4 Primerjava naše paralelne implementacije z obstoječo

Ker so v [16] testirali na datotekah istega korpusa, lahko rezultate naše implementacije primerjamo z njihovimi. Vendar so testirali na drugačnih dolžinah datotek, zato lahko med našimi in njihovimi razultati pride do manjših odstopanj.

| datoteka | kompresijsko razmerje | hitrost sekvenčne izvedbe [MB/s] | hitrost paralelne izvedbe [MB/s] |
|---|---|---|---|
| *sources* | 0,238 | 12,351±1, 71 | 0,981±0, 003 |
| *dna* | 0,294 | 5,490±0, 10 | 0,543±0, 012 |
| *pitches* | 0,398 | 10,550±0, 45 | 0,768±0, 003 |
| *dblp.xml* | 0,188 | 18,741±1, 06 | 1,357±0, 001 |
| *english* | 0,402 | 6,653±0, 18 | 0,567±0, 012 |
| *proteins* | 0,379 | 7,338±0, 28 | 0,450±0, 000 |

Tabela 4.4: Kompresijsko razmerje in hitrost stiskanja v odvisnosti od vrste podatkov

Direktna primerjava med hitrostnimi rezultati naše implementacije in implementacije, osnovane na algoritmu GLZSS, ni mogoča, saj so meritve narejene na različni strojni opremi. Vendar je grafična kartica Nvidia GeForce GTX 590, na kateri so testirali algoritem deflate, osnovan na algoritmu GLZSS, iz iste serije, kot ta, na kateri smo testirali. Ker imata enako strojno arhitekturo, lahko naredimo grobo primerjavo med njima na podlagi števila procesnih elementov in frekvence, pri kateri delujejo. GTX 590 ima 1024 procesnih elementov, ki delujejo pri frekvenci 1350 MHz. GTX 550 Ti ima 192 procesnih elementov, ki delujejo pri frekvenci 1800 MHz. Iz tega lahko sklepamo, da je GTX 590 približno štirikrat zmogljivejša. V tabeli 4.5 je primerjava med kompresijskim razmerjem in hitrostjo algoritmov. Za algoritem deflate, osnovan na algoritmu GLZSS, so hitrosti preračunane na kolikšne bi bile na kartici GeForce GTX 550 Ti. Kompresijska razmerja tudi niso pretirano natančna, saj v [16] niso direktno napisana, ampak smo jih prebrali iz grafa.

Tudi pri rezultatih, preračunanih na enako strojno opremo, se izkaže algoritem, osnovan na algoritmu GLZSS, kot bistveno hitrejši od našega. Vendar zato podatke stisne nekoliko slabše. Razlog za to je, da ne uporablja verižne tabele, torej za vsako mesto v vhodnih podatkih preveri največ eno pono-

| | naša implementacija | | implementacija GLZSS | |
|---|---|---|---|---|
| datoteka | kompresijsko razmerje | hitrost [MB/s] | kompresijsko razmerje | ocena hitrosti [MB/s] |
| *sources* | 0,238 | 0,981 | 0,303 | 49,23 |
| *dna* | 0,294 | 0,543 | 0,370 | 59,96 |
| *pitches* | 0,398 | 0,768 | N/A | N/A |
| *dblp.xml* | 0,188 | 1,357 | 0,227 | 46,33 |
| *english* | 0,402 | 0,567 | 0,476 | 40,67 |
| *proteins* | 0,379 | 0,450 | 0,400 | 31,41 |

Tabela 4.5: Primerjava naše implementacije in implementacije, osnovane na algoritmu GLZSS

vitev. Če zadnja ponovitev ni najdaljša, ampak pred njo v drsečem oknu obstaja daljša, je ne bo našel. S tem pregleda manj potencialnih ponovitev in je nekoliko hitrejši. Glavni razlog za veliko razliko v hitrosti je, da naš slabo izkoristi računske enote ali pri večjem številu niti v delovni skupini opravi preveč dodatnega dela.

Orodje, ki bi omogočalo profiliranje ščepca OpenCL na grafičnih karticah Nvidia, ne obstaja. Približno oceno, koliko se porabi za kateri del algoritma, smo dobili tako, da smo dele algoritma preskočili in merili izvajanje ščepca, ki naredi le del operacij. Izkaže se, da se večina časa porabi za kodiranje LZSS. Preostali algoritem v primerjavi z iskanjem ponovitev LZSS porabi zanemarljivo količino časa.

## 4.5 Možnosti za nadaljne izboljšave

Glavni problem obeh preizkušenih paralelnih implementacij je prevelika količina dodatnega dela v primerjavi s sekvenčnim algoritmom. V eni implementaciji se za komunikacijo med nitmi v delovni skupini porabi toliko časa, da se ščepec izvaja najhitreje, če je v vsaki delovni skupini samo ena nit,

s tem pa slabo izkoristi grafično kartico. Druga opravlja komunikacijo med nitmi redkeje, zaradi česar je hitrejša, a zaradi tega niti opravijo veliko dela, ki v sekvenčni implementaciji ni potrebno. Zato bi verjetno bila možnost, v kateri bi bloke podatkov razdelili med posamezne niti, boljša. Sicer bi niti znotraj delovne skupine veliko časa izvajale različne ukaze, zaradi česar bi se izvajale zaporedno, a bi bilo mogoče vsaj del časa popolnoma izkoristiti strojne zmogljivosti grafične kartice.

Trenutna implementacija uporablja za konstrukcijo Huffmanovega koda z omejeno dolžino kodnih zamenjav algoritem, ki lahko v določenih situacijah sestavi neoptimalno drevo. Z uporabo algoritma združevanja paketov (angl. *package-merge*) [10] bi vedno zgradili optimalno drevo z omejenimi dolžinami Huffmanovih kodnih zamenjav. To bi malo izboljšalo kompresijsko razmerje, a bi bila gradnja Huffmanovega drevesa nekoliko počasnejša.

Implementacije stiskanja podatkov z algoritmi, osnovanimi na algoritmu LZSS, pogosto uporabljajo bolj napreden algoritem, kot je zgolj za vsako mesto v vhodnih podatkih poiskati najdaljšo ponovitev. Ko za neko mesto v vhodnih podatkih najdejo nize, ki so ponovitev trenutnega niza, ne izberejo takoj najdaljšega niza za zapis za referenco. Namesto tega poizkusijo poiskati tako kombinacijo ponovitev, ki pokrije čim večji del vhodnih podatkov in zapiše čim manj vhodnih znakov dobesedno.

Z uporabo takega algoritma bi lahko nekoliko izboljšali kompresijsko razmerje, a upočasnili stiskanje. Večji kot je poudarek na iskanju čim boljše kombinacije, bolj je algoritem počasen. Potrebno bi bilo poiskati primerno kombinacijo hitrosti stiskanja in kompresijskega razmerja.

# Poglavje 5

# Zaključek

V tem delu smo opisali, kaj je stiskanje podatkov in katere lastnosti kompresijskih algoritmov so pomembne. Pregledali smo podrobnosti kompresijskega algoritma deflate, platforme za izvajanje programov na grafičnih procesnih enotah in obstoječe paralelne implementacije tega algoritma.

Implementirali smo sekvenčni algoritem deflate. Profilirali smo ga, da smo ugotovili, kje so ozka grla, ki jih je najpomembneje pohitriti. Ugotovili smo, da se večino časa porabi za iskanje ponovitev v vhodnih podatkih. Našo implementacijo smo primerjali s programom 7-zip in ugotovili, da sta primerljiva.

Za paralelizacijo smo izbrali ogrodje OpenCL. Algoritem smo paralelizirali na dva različna načina, ki se med sabo razlikujeta v tem, kako niti znotraj delovne skupine komunicirajo med sabo. Za obe implementaciji smo poiskali parametre, ki dajejo največjo hitrost stiskanja. Implementaciji smo primerjali med sabo in boljšo izmed njiju tudi s sekvenčnim algoritmom in obstoječo paralelno implementacijo algoritma deflate za izvajanje na grafičnih karticah. Obe naši paralelni implementaciji sta počasnejši tako od sekvenčne, kot od obstoječe paralelne, ker sta pralelizirani tako, da združevanje delnih razultatov posameznih niti zahteva veliko dodatnega dela. Zato se hitreje izvajata z manj nitmi znotraj delovne skupine, s čimer slabo izkoristita grafično procesno enoto.

V nadaljnejm delu je algoritem deflate najprej potrebno paralelizirati na način, ki ne bo bistveno povečal količine dela. Kot osnovo je mogoče uporabiti pristop, v katerem vsaka nit neodvisno dela na svojem bloku podatkov. Nato je smiselno spremeniti izvedbo tako, da bodo niti znotraj iste delovne skupine čim manj časa hkrati izvajale različne vejitve v programu. Vendar komunikacija med nitmi ne sme biti prepogosta. Če za ta problem ne najdemo dobre rešitve, lahko delo razdelimo med centralno in grafično procesno enoto. Ponovitve lahko išče centralna procesna enota, s Huffmanovim kodom pa jih kodira grafična procesna enota. Čeprav s paralelizacijo za grafično procesno enoto nismo uspeli pohitriti stiskanja, smo našli možne težave pri stiskanju podatkov na grafični procesni enoti.

# Literatura

[1] OpenCL optimization guide. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/opencl-optimization-guide/, povzeto avgust 2016.

[2] Nvidia OpenCL best practices guide, 2009. http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/nvidia_opencl_bestpracticesguide.pdf, povzeto avgust 2016.

[3] Nvidia CUDA reference manual, 2010. http://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/CUDA_Toolkit_Reference_Manual.pdf, povzeto avgust 2016.

[4] Summary of the multiple file compression benchmark tests, 2011. http://www.maximumcompression.com/data/summary_mf2.php, povzeto avgust 2016.

[5] Intel® SDK for OpenCL* applications 2012 OpenCL* optimization guide, 2012. https://software.intel.com/sites/landingpage/opencl/optimization-guide/, povzeto avgust 2016.

[6] The OpenCL specification, 2012. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[7] L. P. Deutsch. Deflate compressed data format specification version 1.3. 1996. https://www.ietf.org/rfc/rfc1951.txt, povzeto avgust 2016.

[8] P. Ferragina and G. Navarro. Pizza Chili corpus, 2005. `http://pizzachili.dcc.uchile.cl/texts.html`, povzeto avgust 2016.

[9] D. A. Huffman et al. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[10] L. L. Larmore and D. S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM (JACM)*, 37(3):464–473, 1990.

[11] D. Luenberger. *Information Science*. Princeton University, 2006.

[12] A. Ozsoy and M. Swany. CULZSS: LZSS lossless data compression on cuda. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 403–411. IEEE, 2011.

[13] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.

[14] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[15] Y. Zu and B. Hua. GLZSS: LZSS lossless data compression can be faster. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, page 46. ACM, 2014.

[16] Y. Zu and B. Hua. Parallelizing the deflate compression algorithm on GPU. In *Journal of Computational Information Systems 11*, pages 6159—-6170, 2015.