TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Ztrátové komprese obrazu v embedded zařízeních

Bakalářská práce

Studijní program: B2646 Informační technologie
Studijní obor: Informační technologie

Autor práce: **Jáchym Machat**
Vedoucí práce: Ing. Martin Rozkovec, Ph.D.
Ústav informačních technologií a elektroniky

Liberec 2021

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání bakalářské práce

Ztrátové komprese obrazu v embedded zařízeních

Jméno a příjmení: **Jáchym Machat**
Osobní číslo: M17000083
Studijní program: B2646 Informační technologie
Studijní obor: Informační technologie
Zadávající katedra: Ústav informačních technologií a elektroniky
Akademický rok: **2020/2021**

Zásady pro vypracování:

1. Seznamte se základními a pokročilými metodami ztrátové komprese obrazu. Dále se seznamte s platformou APSoC Zynq a balíkem vývojových nástrojů Xilinx Vivado, především pak s nástrojem vysokoúrovňové syntézy HLS.
2. Proveďte odhad náročnosti implementace algoritmů a vybraný implementujte na PC ve formě enkodéru a dekodéru.
3. Algoritmus enkodéru naprogramujte pro vnořená jádra ARM v Zynq a porovnejte rychlost s PC.
4. Vyberte části algoritmu vhodné pro implementaci v HW a upravte je tak, aby je bylo možné zpracovat pomocí nástroje HLS. Vyhodnoťte výsledky.

Rozsah grafických prací:	Dle potřeby dokumentace
Rozsah pracovní zprávy:	30-40
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] Khalid Sayood. Indtroduction to Data Compresssion, 5th Edition, 2017, Morgan Kaufmann, ISBN: 9780128094747
[1] Xilinx Inc., Vivado Design Suite User Guide: High-Level Synthesis (ug902), 2019, online <https://bit.ly/3428vjl>, 24. 10. 2109
[1] Topiwala, P.N.: Wavelet Image and Video Compression, 1998, Springer, ISBN: 978-0792381822

Vedoucí práce:	Ing. Martin Rozkovec, Ph.D. Ústav informačních technologií a elektroniky
Datum zadání práce:	19. října 2020
Předpokládaný termín odevzdání:	17. května 2021

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Ing. Ondřej Novák, CSc.
vedoucí ústavu

V Liberci dne 19. října 2020

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

16. května 2021 Jáchym Machat

Ztrátové komprese obrazu v embedded zařízeních

Abstrakt

Cílem této práce je seznámení studenta s metodami ztrátové komprese a jejich následnou implementací ve vestavných zařízeních. Konkrétně se jedná o naivní implementaci metody JPEG v APSoc Zynq. Kód komprese je napsán v jazyce C. Upravený kód je syntetizován pomocí nástroje Vivado HLS a integrován do programovatelného hradlového pole. Je porovnána složitost implementace a rychlost běhu algoritmu na platformě x64 a APSoC Zynq s využitím hardware akcelerátoru. Na závěr je rozhodnuto, zda úsilí, které je nutné k přizpůsobení kódu pro nástroj HLS, přinese významné zrychlení oproti plnému nasazení na procesoru.

Klíčová slova: Ztrátová komprese, JPEG, Zynq, Vivado, HLS, Xilinx, FPGA, Embedded, C

Lossy image compression in embedded devices

Abstract

The goal of this thesis is to acquaint the student with the methods of lossy compressions and their subsequent implementation in embedded devices. Specifically, it is a naive implementation of the JPEG method in APSoc Zynq. The compression code is written in plain C. The modified code is synthesized using the Vivado HLS tool and integrated into a programmable gate array. The complexity of implementation and performance of the algorithm are compared on the x64 and APSoC Zynq platforms. Finally, it is decided whe--ther the effort required to customize the code for HLS will result insignificant acceleration over full CPU deployment.

Keywords: Lossy compression, JPEG, Zynq, Vivado, HLS, Xilinx, FPGA, Embedded, C

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce panu Ing. Martinu Rozkovci, Ph.D. za odborné vedení, konzultace, cenné rady, upřímnost a především nekonečnou trpělivost při zpracování této práce.

Také musím poděkovat mé mámě MgA. Haně Hančové za podporu, korekturu a poskytnutí obrázků.

Obsah

Seznam zkratek 12

1 Úvod **13**

2 Použité prostředky **14**

2.1 Testovací obrázky 14
2.2 Code::Blocks 14
2.3 Vivado HLS 14
2.4 ZedBoard 15

3 Kompresní algoritmy **17**

3.1 Seznámení s kompresními algoritmy 17
3.2 Ztrátové komprese obrazu 20
3.2.1 Metoda JPEG 20
3.2.2 JPEG 2000 28
3.2.3 WebP 29
3.2.4 AV1 29
3.2.5 Shrnutí kompr. metod obrazu 29
3.2.6 Výběr vhodného kompresního algoritmu 29

4 Vlastní řešení **31**

4.1 Programování kompresního algoritmu 31
4.1.1 Programování metody JPEG 31
4.1.2 Programování pro vnořená jádra v Zynq 35
4.2 Programování pro programovatelné hradlové pole 36
4.2.1 Využití proměnných typu fixed 37
4.2.2 Práce s HLS 40
4.2.3 Práce s Vivado 44
4.2.4 Využití FPGA 46
4.3 Výsledky měření 49

4.3.1 Zynq s FPGA vs. PC 49
4.3.2 Zynq s FPGA vs. Zynq bez FPGA 50
4.3.3 Zynq s FPGA vs. Zynq libjpeg-turbo 51

5 Závěr 52
5.1 Souhrn práce 52
5.2 Názor na HLS 53

Použitá literatura 53

Přílohy 57

Seznam obrázků

2.1 Code::Blocks . . . 16
2.2 Deska ZedBoard . . . 16

3.1 Původní obrázek . . . 18
3.2 Zkomprimovaný obrázek . . . 18
3.3 Komprimovaný obrázek . . . 22
3.4 Přiblížení . . . 22
3.5 DCT blok . . . 22
3.6 Matice G . . . 23

4.1 Zynq profiling . . . 36
4.2 HLS Analysis . . . 42
4.3 Řádek macro bloků . . . 43
4.4 Blokové schéma zapojení . . . 45
4.5 Buňky v FPGA . . . 46

Seznam grafů

3.1 Porovnání velikostí 25

4.1 Porovnání fixed s kvalitou kompr. 80 39
4.2 Porovnání fixed s kvalitou kompr. 100 39
4.3 Porovnání PSNR metod fixed a double 40
4.4 Porovnání kompr. poměru s DC rozdílem a bez DC rozdílu 43
4.5 Poměr kompr. poměrů s DC rozdílem a bez DC rozdílu 43
4.6 Porovnání zrychlení doby běhu komprese v závislosti na počtu bloků 48
4.7 Porovnání času běhu komprese obrázku Lenna 49
4.8 Porovnání času běhu komprese obrázku Příroda 49
4.9 Porovnání poměrů časů 50
4.10 Porovnání času běhu komprese obrázku Lenna 50
4.11 Porovnání času běhu komprese obrázku Příroda 50
4.12 Porovnání poměrů časů 51

Seznam tabulek

3.1 Rozsahy hodnot 27

4.1 Hlavička enkódovaného obrázku 32
4.2 Porovnání libjpeg_jm s libjpeg-turbo 34
4.3 Hlavička obrázkového formátu 35
4.4 Porovnání PC s platformou Zynq 36
4.5 Parametry *encodeImage* funkce 42

Zdrojové kódy

4.1 Ukázka běhu programu 33
4.2 PC profiler 34
4.3 HLS Top Function 41

Seznam zkratek

DCT Diskrétní kosínová transformace
DC Stejnosměrná složka DCT matice
AC Střídává složka DCT matice
JPEG Referováno jako metoda ztrátové komprese obrazu (Joint Photographic Experts Group)
HLS Referován jako nástroj Vivado HLS (High-Level Synthesis)
PSNR Špičkový poměr signálu k šumu (Peak signal-to-noise ratio)
FPGA Programovatelné hradlové pole (Field-programmable gate array)
RLE Bezztrátový kompresní algoritmus (Run-length encodin)
HW Hardware
HDL Programovací jazyk sloužící pro popis HW (Hardware Description Language)
HLS High-Level Synthesis

1 Úvod

Tato práce se zabývá problematikou implementace ztrátového kompresního algoritmu v programovatelném hradlovém poli. Konkrétně tedy takovém, které se nachází na vývojové desce ZedBoard Zynq-7000 od firmy Xilinx. Algoritmus je syntetizován pomocí nástroje HLS a poté implementován do FPGA. Hlavním cílem práce je rozhodnout, zda úsilí, nutné k využití hardware akcelerátoru, přinese význačné zrychlení oproti optimalizovanému nasazení na procesoru.

Zprvu je nutné se seznámit se širokou škálou ztrátových algoritmů obrazu a rozhodnout se pro takový, který by nejlépe vyhovoval požadavkům této práce, případně si vymyslet algoritmus vlastní. Tento algoritmus bude poté naprogramován v jazyce C, a to bez použití externích knihoven. Pro kontrolu správnosti vytvořeného algoritmu bude včetně enkodéru (komprimuje obrázek do speciálního formátu) vytvořen i dekodér (vrátí komprimovaný obrázek do původního stavu).

Předpokládáme, že vstupní obrázky ke kompresi jsou pouze jednokanálové s 8bitovým rozsahem hodnot. Algoritmus by tedy měl jít použít např. ke kompresi infračervených obrázků.

Výsledný kód bude zprovozněn na vestavném zařízení zprvu výhradně na procesoru, poté i s využitím FPGA. Kód bude nutné uzpůsobit nástroji HLS, aby jej bylo možné syntetizovat. Z tohoto důvodu bude kód komprese psán co nejjednodušeji.

Na závěr bude porovnána rychlost algoritmu v závislosti na běžící platformě.

2 Použité prostředky

2.1 Testovací obrázky

Všechny obrázky, použité ke kompresi, jsou jednokanálové s 8bitovou hloubkou. Jsou uložené v bezztrátovém formátu. Aby se předešlo rozdílům dimenzí původních a komprimovaných obrázků a s tím spojeným potížím při následném analyzovaní, mají všechny obrázky šířku i výšku dělitelnou osmi. Obrázky naleznete v příloze.

2.2 Code::Blocks

Code::Blocks je open-source multiplatformní vývojové prostředí zaměřené zejména na programovací jazyky C a C++.

Mezi jeho výhody patří jednoduchost a s tím spojená rychlost a přehlednost grafického rozhraní. Svými nástroji se nijak význačně neliší od ostatních prostředí. Za zmínku stojí možnost snadného instalování a vytváření pluginů, kterými toto vývojové prostředí, díky početné komunitě, disponuje nemalým množstvím. Dále je možné např. jednoduše importovat projekty z programu Microsoft Visual Studio.

Code::Blocks podporuje většinu populárních kompilátorů, včetně GCC a Microsoft Visual C++.

Pro tuto práci je vybrané vývojové prostředí plně dostačující. 32bitová PC verze softwaru byla kompilována pomocí GCC s nastavením optimalizace *-o3*. Spouštěna byla na počítači s operačním systémem Windows 10 s čtyřjádrovým procesorem Intel(R) Core(TM) i5-9300H CPU @ 2.40Ghz. [10]

2.3 Vivado HLS

Vivado HLS (High-Level Synthesis) je vývojové prostředí převádějící kód C do RTL implementace, která lze posléze syntetizovat do tzv. programovatelného hradlového pole (FPGA). Jedná se, jak název vypovídá, fakticky o libovolně možně nastavitelný

hardware akcelerátor. FPGA poskytuje paralelní architekturu vhodnou zejména pro výpočetně náročné procesy, a tím pomáhá ke zlepšení celkového výkonu programu.

Jelikož se v HLS pracuje s C kódem, je možné pracovat ve vyšší abstrakci než při běžném používání jazyka HDL. Uživatelovi zároveň stačí jen minimální dovednosti s prací s HW, jak je také naznačeno v příručce HLS: "Consider Vivado HLS to be an expert designer who by default is given the task of finding the design with the highest throughput, lowest latency and minimum area." [1]

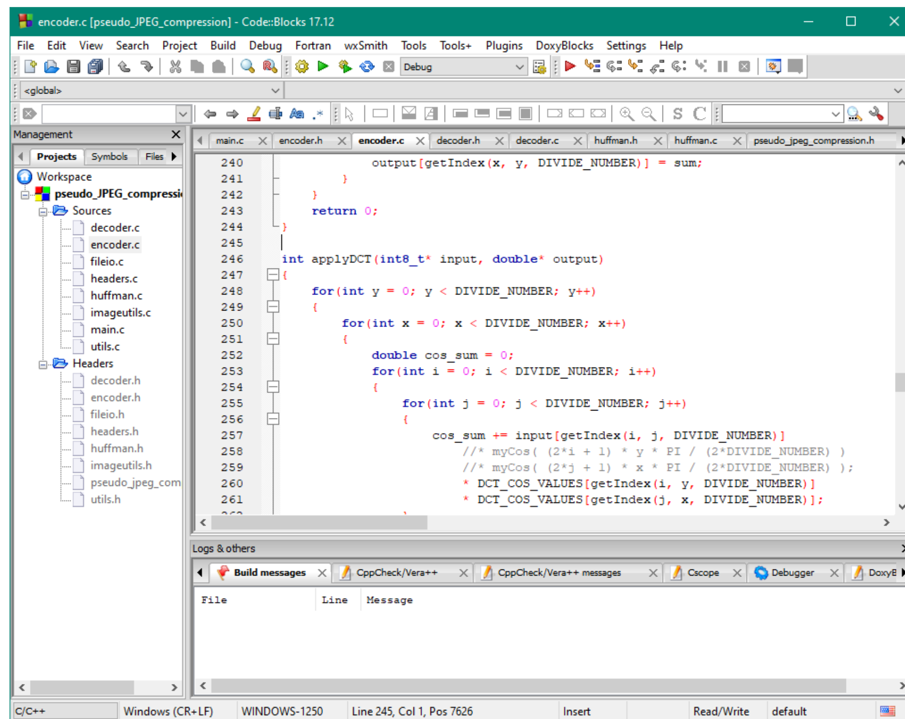
Nástroj HLS má potenciál být nezbytnou součástí arzenálu každého kodéra. Jedním z cílů této práce je tedy rozhodnout o skutečné relevanci jeho využití. [17]

2.4 ZedBoard

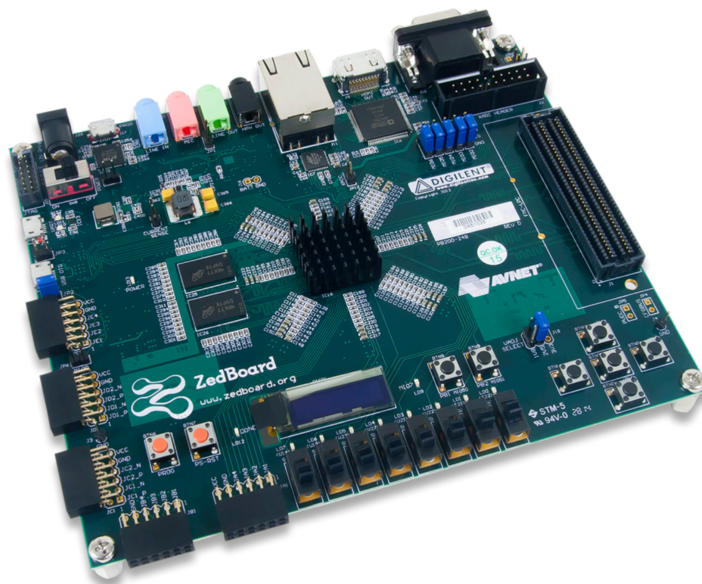
ZedBoard™ je vývojový výukový kit určený k seznámení s platformou Xilinx Zynq®-7000 AP SoC. Deska se hodí zejména pro rychlé prototypování a ověření konceptu řešení.

ZedBoard je vybaven dvou jádrovým ARM Cortex-A9 procesorem s integrovaným 28nm Artix-7 FPGA. [11]

[1]Xilinx Inc., Vivado Design Suite User Guide: High-Level Synthesis (ug902). [online]. 2019 [cit. 2021-4-28]. Dostupné z: https://bit.ly/3428vjl s. 66.



Obrázek 2.1: Code::Blocks



Obrázek 2.2: Deska ZedBoard, zdroj: [20]

3 Kompresní algoritmy

3.1 Seznámení s kompresními algoritmy

Komprese je způsob zpracování počítačových dat za účelem snížení jejich celkové velikosti. To se hodí hlavně pro zmenšení potřebného místa k uložení (např. různé archivátory) nebo pro snížení potřebného datového toku k jejich přenosu (např. streamování videa přes internet). Data se zakódují do speciální binární podoby, která v závislosti na korelaci (nenáhodnosti) původních dat disponuje menší velikostí. Aby bylo možné se zakódovanými daty jakkoliv manipulovat, je nutné data dekódovat zpátky do své podoby původní.

Kompresní algoritmy můžeme dělit na dva základní typy: bezeztrátové a ztrátové.

Bezeztrátovou kompresí zakódovaná data dekódujeme do původního stavu bez jakékoliv změny. O žádná data nepřijdeme. Je využívána všude, kde není možné přijít ani o jediný bit, což je nutné např. u počítačových programů, textů či velmi citlivých a důležitých dat. Její nevýhoda spočívá v nepříliš velkém kompresním poměru, což je poměr velikosti původních dat s daty zakódovanými.

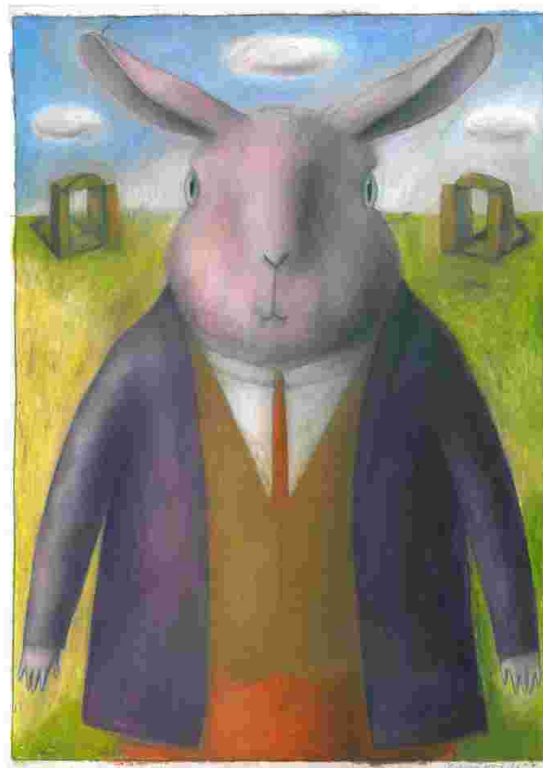
Naopak tomu je u komprese ztrátové, kde je za cenu ztráty části informace původních dat, dosaženo velkého kompresního poměru. Tato komprese je používána především na zpracování obrázků, videí a jiných médií, kde je možné díky nepříliš dokonalým smyslům člověka vynechat podstatnou část původních dat, aniž by to bylo na první pohled znát.

Lidské zrakové ústrojí je jedním z našich nejdůležitějších smyslů. Lidské oči obsahují čočku, která promítá odraz na sítnici. Tato sítnice obsahuje dva druhy receptorů, a to tyčinky a čípky. Tyčinky jsou citlivé na změnu jasu/intenzity dopadajícího světla. Čípky jsou citlivé na konkrétní vlnové délky viditelného spektra, a to červené, modré a zelené regiony viditelného spektra. Čípky jsou koncentrované v tzv. žluté skvrně. Žlutá skvrna je místem nejostřejšího vidění. Čili i když je v oku více tyčinek, tak čípky poskytují lepší „rozlišení" signálu pro mozek. Svaly oka roztahují a smršťují čočku tak, aby obraz dopadal přímo na žlutou skvrnu. V malém osvětlení

je tedy paradoxně lepší koukat na okolí zkoumaného objektu, aby obraz padal spíše na tyčinky. Čípků reagující na zelené světlo je v oku nejvíce. To hraje zásadní roli ve snímacích čipů digitálních fotoaparátů, které obsahují rovněž větší množství bodů na snímání zeleného světla. Této skutečnosti se dá využít při implementaci ztrátové komprese.

Obrázky komprimované ztrátovou metodou se mohou jevit po přiblížení lehce rozmazaně. Pokles kvality dekódovaných dat závisí na druhu a zvolené kvalitě komprese. Jedná se tedy o kompromis minimálního zkreslení a maximálního poměru komprese.

Obrázek 3.1: Původní obrázek



Obrázek 3.2: Zkomprimovaný obrázek

Zkreslení způsobené kompresí se dá měřit různými způsoby. Mnohdy bývá nejlepší použít lidské smysly jako takové, tedy nechat intenzitu zkreslení posoudit člověkem. Zkušený fotograf si nejlépe všimne artefaktů či změn barev komprimovaného obrázku. Vášnivý audiofil zase nejlépe určí, jak moc se liší čistota zvuku u komprimované audio nahrávky. Tento způsob je samozřejmě vhodný jen pro takové média, která jsou určena pro běžnou konzumaci ve formě filmů, apod. Nehledě na to, že každý vnímá kvalitu trochu jinak. Objektivní kritérium zkreslení se dá měřit např. metodou PSNR.

PSNR (peak signal-to-noise ratio) je špičkový poměr signálu k šumu způsobeného zkreslením. Této metody se využívá zejména u komprese obrázku a jiných zkvantizovaných médií. Toto kritérium bohužel příliš neodpovídá tomu, jak vnímá zkreslení člověk. Existuje mnoho jiných složitějších metod, které se snaží tento problém vyřešit. Pro splnění požadavků této práce ale PSNR bohatě stačí.

Mezi nejpoužívanější formáty využívající ztrátovou kompresi patří JPEG a MP3. Jejich princip spočívá v transformování původního modelu dat do takových modelů, které lépe popisují vlastnosti zdroje. Např. z domény časové do domény frekvenční. Poté je možné potlačit vyšší frekvence z původních dat, jejichž absenci naše lidské smysly většinou ani nezaznamenají. Takto pozměněná data se nakonec většinou zkomprimují i bezeztrátově. Např. pomocí tzv. Huffmanovo kódování. [1] [2]

3.2 Ztrátové komprese obrazu

Pokud se bavíme o kompresi statických rastrových obrázků, tak se v dnešní době používají zejména metody založené buď na kosínové, waveletové nebo fraktálové transformaci. Tyto metody, jak už bylo řečeno, transformují model dat do takových modelů, které, podle dané situace, lépe popisují vlastnosti původních dat.

3.2.1 Metoda JPEG

Metoda JPEG (Joint Photographic Experts Group) je ztrátová kompresní metoda vytvořená v roce 1992. Její princip spočívá v redukci barevné informace a použití tzv. diskrétní kosínové transformace (DCT), díky které může data obrázku zbavit o nepotřebné vysokofrekvenční složky. Data jsou následně zpracována pomocí bezeztrátové komprese.

JPEG je ale pouze jen standard, resp. „návod“, který říká, co přesně s daty udělat. Je z něho odvozeno několik různých souborových typů. Mezi nejpoužívanější patří formát JFIF (JPEG File Interchange Format), který je obecně označován jako JPEG/JPG.

JPEG metoda se skládá z několika kroků, a to:

1. Převedení dat do barevného modelu YCbCr
2. Podvzorkování barevných složek
3. Provedení DCT
4. Provedení kvantizace
5. Zig-zag vyčítání
6. Run-length encoding
7. Huffmanovo kódování

Existuje více variant procesu, kde se např. Huffmanovo kódování nahradí kódováním aritmetickým. [6] [7]

Převedení dat do barevného modelu YCbCr

Barevný model popisuje barvu jednotlivého bodu obrazu pomocí kombinace specifických barev či jiných parametrů.

Jedním z takovýchto modelů je barevný model RGB (Red Green Blue). Ten, jak název vypovídá, popisuje body obrazu pomocí červené, zelené a modré barvy.

U obrázků se musí určit i jeho bitová hloubka, neboli kolik bitů bude použito k uložení jednotlivých barevných informací. Standardní bitovou hloubkou pro RGB model bývá 24 bitů, tedy 8 bitů na každý barevný kanál.

Barevný model YCbCr je složen z luminance/jasu (Y) a modré a červené chrominanční komponenty (Cb, Cr). Složka Y je v podstatě plněhodnotná černobílá složka, což je také hlavním záměrem tohoto modelu. Byl využíván k přenosu barevného televizního signálu, tak aby mohly staré černobílé televize i nadále zobrazovat nezkreslený obraz. Zároveň je možné využít skutečnosti, že lidské oko vnímá rozdíl jasu mnohem intenzivněji než rozdíl barevných složek. Chrominančním komponentám lze tedy výrazně snížit bitovou hloubku bez zřetelného poklesu kvality.

Ve většině případů je nutné převádět z modelu RGB. To se dělá pomocí následujících vzorců.

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B, \tag{3.1}$$

$$Cb = -0.1687 \cdot R - 0.3313 \cdot G + 0.5 \cdot B + 128, \tag{3.2}$$

$$Cr = 0.5 \cdot R - 0.4187 \cdot G - 0.0813 \cdot B + 128. \tag{3.3}$$

Podvzorkování barevných složek

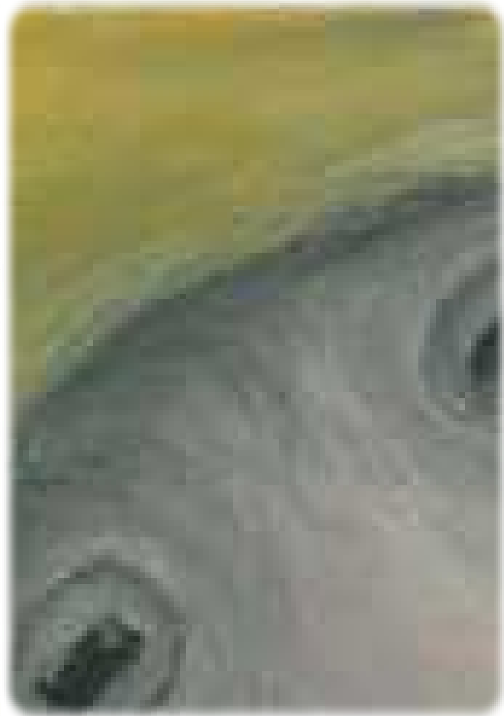
Barevné kanály bývají většinou podvzorkovány na čtyřbitovou hloubku. Tímto dokážeme značně snížit výslednou velikost zkomprimovaného obrázku.

Podvzorkování se provádí zanedbáním 4 nejméně významných bitů, resp. vydělením číslem 16.

Provedení DCT

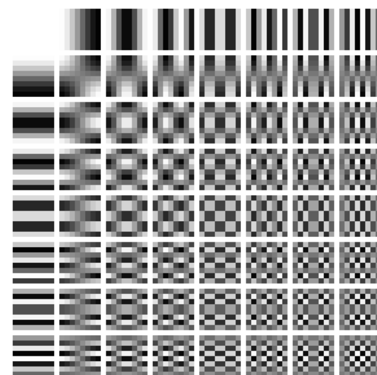
DCT (Diskrétní kosínová transformace) je obdoba Fourierovy transformace, jejíž hlavní princip spočívá v transformaci původních dat do frekvenční domény. DCT tedy vlastně popisuje, z jakých a jak silných frekvencí se skládá např. obrázek nebo zvuková nahrávka. U obrázků se kosínová transformace provádí na každý barevný kanál zvlášť. Aby bylo možné jednoduše využít paralerismu, obrázek ke zpracování je rozdělen na malé bločky o výšce a šířce 8 pixelů, které lze nezávisle na sobě zkomprimovat.

Obrázek 3.3: Komprimovaný obrázek



Obrázek 3.4: Přiblížení

Na obrázku 3.4 můžeme vidět, po přiblížení na oblast červeného obdélníčku, že JPEG komprese opravdu pracuje s 8×8 bločky. Hrany jednotlivých 8×8 čtverečků jsou znatelně viditelné.



Obrázek 3.5: DCT blok, zdroj: [19]

Na obrázku 3.5 lze vidět frekvenční složky, které popisují jeden blok 8×8 pixelů.

Vlevo nahoře je stejnosměrná (DC) složka s nulovou frekvencí. Určuje intenzitu celého bloku, a proto je tato hodnota ze všech nejvýznamnější. Všechny ostatní složky jsou střídavého charakteru (AC). Vpravo dole mají nejvyšší frekvenci a z lidského hlediska jsou tedy nejméně významné.

Pokud není výška či šířka obrázku dělitelná osmi, běh komprese se lehce zkomplikuje. Hraniční bločky totiž nebudou mít správné rozměry. Tento problém se dá řešit různými způsoby. Nejjednodušším z nich je k obrázku přidat nadbytečné černé pixely.

Diskrétní kosínovou transformaci lze spočítat pomocí tzv. Fast DCT metody. K té nám stačí dvě předem připravené matice. Vzorec je následující

$$F = G \cdot f \cdot G^T, \tag{3.4}$$

kde F je blok vypočítaných DCT složek, G je předpřipravená matice, f je matice hodnot pixelů a G^T je inverzní matice G

Matice G se vypočítá způsobem zvýrazněným na obrázku 3.6.

$$\boldsymbol{G} = \frac{1}{2}\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ \cos\frac{\pi}{16} & \cos\frac{3\pi}{16} & \cdots & \cos\frac{15\pi}{16} \\ \vdots & \vdots & & \vdots \\ \cos\frac{7\pi}{16} & \cos\frac{21\pi}{16} & \cdots & \cos\frac{105\pi}{16} \end{bmatrix}.$$

Obrázek 3.6: Matice G, zdroj: [3]

Celá operace diskrétní kosínové transformace je redukovaná na pouhé násobení tří matic, což značně zkrátí výsledný čas celého kompresního algoritmu. [3]

Provedení kvantizace

Kvantizace spočívá ve zmenšení či úplném vynulování jednotlivých DCT složek. To se provádí celočíselným dělením prvky ze speciální kvantizační matice. Takových matic existuje nespočet a každá je optimalizovaná pro jiný účel. Existují ale standardní, které se dají použít univerzálně pro většinu případů.

Právě touto metodou dochází k největšímu poklesu kvality, resp. k nejmenšímu zkreslení, a tudíž i k zmenšení velikosti obrázku.

Následující kvantizační matice 3.5 má největší čísla vpravo dole, abychom vynulováním přišli především o vysokofrekvenční DCT složky.

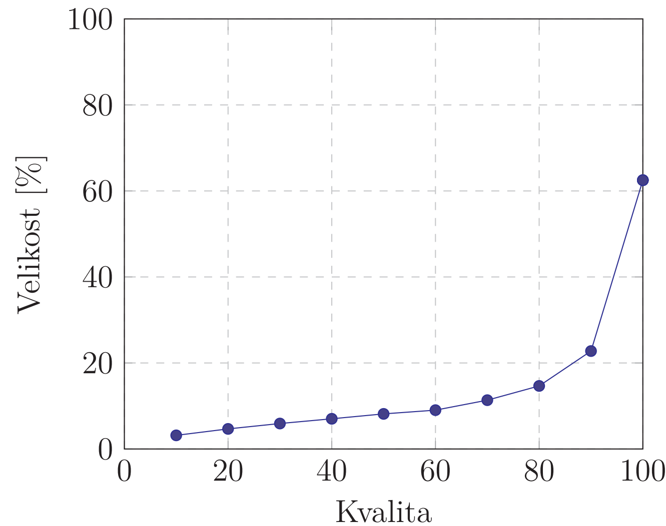
$$quantMatrix = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix} \tag{3.5}$$

Vynásobením této matice specifickým číslem můžeme měnit kvalitu výsledné ztrátové komprese. Čím větší budou jednotlivé prvky kvantizační matice, tím méně kvalitní bude výstupní obrázek. Škála kvality je od nuly do sta. To ale neznamená, že pokud budeme obrázek komprimovat s kvalitou 50, bude výstupní obrázek 2krát méně kvalitní či 2krát menší. Cílem je zpravidla zvolit takovou hodnotu, při které nebude příliš znatelné zkreslení obrázku a zároveň bude co největší kompresní poměr.

Komprimovat obrázek s kvalitou v rozmezí 95–100 je ve většině případů, vzhledem k výslednému kompresnímu poměru, znatelně nevýhodné. Komprimovaný obrázek je sice víceméně totožný s originálem, ale také většinou zabírá mnohem více místa než při použití nižší kvality. Zvolit hodnotu kvality v tomto rozmezí je nutné pro obrázky s gradientem, jako je např. západ slunce. Důležité je si uvědomit, že i když budeme obrázek komprimovat s kvalitou 100, tak nebude výstup, kvůli zaokrouhlování ve výpočtech, naprosto totožný s originálem. Pokud toto provedeme několikrát po sobě na jeden a ten samý obrázek, začne docházet k zřetelnému zkreslení vůči originálu. Zkreslení začíná být viditelné u většiny obrázků pod hodnotou 70. Pod hodnotou 30 už je zkreslení do očí bijící.

Obvykle se doporučuje používat kvalitu v rozmezí 80–90, což nám ve valné většině případů zajistí pěkný kompromis mezi nízkým zkreslením a vysokým kompresním poměrem. [5]

Komprese v grafu 3.2.1 byla provedena na obrázek Lenna. Graf velikosti v závislosti na kvalitě je víceméně lineární v rozmezí 0–80, zato v rozmezí 80–100 začne výrazně stoupat, tedy kompresní poměr začne výrazně klesat.



Graf 3.1: Porovnání velikostí

Zig-zag vyčítání

Zkvantizované DCT složky vyčítáme od prvků s nejnižší frekvencí po prvky s největší, resp. podle jejich významnosti. Přesné pořadí je zvýrazněno pomocí indexů v následující matici 3.6.

$$zigzagIndex = \begin{pmatrix} 0 & 1 & 5 & 6 & 14 & 15 & 27 & 28 \\ 2 & 4 & 7 & 13 & 16 & 26 & 29 & 42 \\ 3 & 8 & 12 & 17 & 25 & 30 & 41 & 43 \\ 9 & 11 & 18 & 24 & 31 & 40 & 44 & 53 \\ 10 & 19 & 23 & 32 & 39 & 45 & 52 & 54 \\ 20 & 22 & 33 & 38 & 46 & 51 & 55 & 60 \\ 21 & 34 & 37 & 47 & 50 & 56 & 59 & 61 \\ 35 & 36 & 48 & 49 & 57 & 58 & 62 & 63 \end{pmatrix}. \quad (3.6)$$

Díky tomuto výčtu se nám sdruží velké množství nul, a to zejména na konci sekvence. To se dá využít při tzv. Run-length encoding.

Run-length encoding

Run-length encoding (RLE) je bezztrátový kompresní algoritmus, který se hodí na sekvence znaků, ve kterých se opakuje velké množství stejných znaků za sebou. Jako výstup produkuje sekvenci dvojic 3.7, kde jeden z páru značí hodnotu/druh znaku

a druhý jeho kvantitu.

$$aabccaaaa > 2a, 1b, 2c, 4a \tag{3.7}$$

V případě metody JPEG vznikne po zig-zag výčtu velké množství sdružených nul. Jiné hodnoty se takto za sebou příliš neopakují. Proto se používá speciální případ RLE, kdy se vytváří dvojice kvantity po sobě jdoucích nul a samotné hodnoty, která po nich následuje.

Tyto dvojice jsou zapsány pomocí tzv. Huffmanova kódování.

Huffmanovo kódování

Huffmanovo kódování je další metodou bezeztrátového komprese. Podstatou tohoto algoritmu je zjištění nové možné reprezentace dat, ve kterých budou nejpočetnější znaky zapsány pomocí menšího počtu bitů než u znaků původních.

Nejprve spočítá četnost znaků, předem určené délky, v datech, které se chystáme komprimovat. Pomocí těchto četností vytvoří strom, který určí novou reprezentaci znaků. Počet bitů nutný k reprezentaci znaků o velké četnosti je zpravidla co nejmenší. Naopak u znaků o malé četnosti je tento počet bitů největší.

Zakódované znaky jsou v tzv. prefixovém kódu, tzn. že lze každý z nich jednoznačně dekódovat bez nutnosti použití oddělovače.

V metodě JPEG se tento algoritmus používá pouze na určitou část předzpracovaných dat. Produktem předešlého kroku je dvojice počtu nul a za nimi následované nenulové hodnoty.

Pro počet nul jsou rezervovány 4 bity. Tzn. že pokud je počet nul rovný 16, tak se jako druhá hodnota ve dvojici určí nula. Pokud je počet nul vyšší jak 16, tak se dvojice rozdělí na tolik dalších dvojic, až bude počet nul v každé z nich menší jak 17.

Pro nenulovou hodnotu se určí tzv. informační entropie, resp. počet bitů nutný pro její zakódování, podle následující tabulky 3.1:

Tabulka 3.1: Rozsahy hodnot

Počet bitů	**Hodnota**
0	0
1	$\{-1; 1\}$
2	$\{-3;-2; 2; 3\}$
3	$<-7;-4> \cup <4; 7>$
...	...
11	$<-2047;-1024> \cup <1024, 2047>$

Pro informační entropii jsou také určeny 4 bity, tudíž nám vznikne pár počtu nul a počtu bitů potřebných k zapsání hodnoty, tedy např. při počtu 5 nul a hodnotě 1028 vychází bajt 01011011_2. Až tato výsledná dvojice se zakóduje pomocí Hufmmanova kódování.

Samotná nenulová hodnota se určí podle počtu určených bitů a jím přiřazeným rozsahem hodnot, tedy např. číslo -6 zapíšeme jako 001_2, jelikož je to druhé číslo v pořadí v rozsahu $<-7; -4> \cup <4; 7>$. Číslo 3 zapíšeme jako 11_2. Tento proces je analogický k výpočtu dvojkového doplňku.

Pro konec výčtu hodnot (End Of Block) se používá zakódovaná dvojice $\{0; 0\}$, která logicky značí, že už žádné další hodnoty nenásledují.

Jelikož je hodnota DC složky zpravidla mnohem větší než zbylé AC složky, je kódovaná samostatně podle vlastních Huffmanovo tabulek. Z tohoto důvodu není nutné tvořit dvojici počtu nul a potřebných bitů k zapsání. Stačí nám pouze informační entropie, kterou zakódujeme pomocí Huffmanova kódování, a za ní zapíšeme samostatnou hodnotu v daném rozsahu.

Jak lze vidět, čím menší hodnoty v metodě JPEG kódujeme, tím potřebujeme menší počet bitů k zapsání jejich reprezentace v daném rozsahu. Toho můžeme využít u DC složek, které se v běžných obrázcích v určitých úsecích příliš neliší. Jak už bylo řečeno, DC složka určuje intenzitu celého bločku, tudíž pokud např. zpracováváme obrázek oblohy, bude mít velké množství bločků za sebou totožnou či sobě hodně blízkou jasovou hodnotu. U DC složek se tedy provádí rozdíl s hodnotou z předešlého bločku. Pokud je DC hodnota 1. bločku 13, hodnota 2. bločku 14 a hodnota 3. bločku 11, vyjdou nám nové hodnoty 13, 1 a -3. Tento krok ale znamená, že je kódování jakéhokoliv bločku (krom prvního) závislé na výpočtu bločku předešlého. Z tohoto důvodu se značně zkomplikuje možné paralelní zpracování jednotlivých bločků. Jelikož tento proces nepřináší příliš velké zlepšení, co se výsled-

né velikosti zakódovaného obrázku týče, je možné tento krok vynechat za účelem možného zrychlení algoritmu.

Huffmanovo kódování je poměrně náročným algoritmem. V každé iteraci algoritmu je nutné setřídit pole prvků, což je obzvláště v hardwaru nelehkým úkolem. Proto je u JPEG komprese většinou využíváno předem vypočítaných Huffmanových tabulek, ze kterých se dá jednoduše a rychle určit reprezentace výsledných znaků. Zároveň není nutné Huffmanovy tabulky přikládat do výstupního souboru, jelikož jsou vždy stejné. Ve většině případů je využíváno standardních JPEG tabulek optimalizovaných pro co nejuniverzálnější použití.

Mimo Huffmanova kódování se dá použít např. kódování aritmetické, se kterým se dá dosáhnout lepšího kompresního poměru. [15] [16]

3.2.2 JPEG 2000

JPEG 2000 je iterací metody JPEG. Byla vytvořena, jak název vypovídá, v roce 2000. Je založená na vlnkové transformaci. Vlastnosti, kterými tato metoda disponuje, byly odpovědí na nedostatky předešlé metody JPEG. Mezi ně patří:

- Progresivní dekódování
  - Díky tomu, jak probíhá metoda JPEG 2000 a jak kódovaná data řadí, je možné dekódovaný obrázek zobrazovat s postupně zvyšující se kvalitou, např. při stahování obrázku přes internet. Tato vlastnost je podporována i původní metodou JPEG, ale zdaleka ne tak účinně. K tomu je navíc redundantní (zbytečné data navíc), což neplatí v JPEG 2000.
- Možnost volby mezi ztrátovou a bezeztrátovou kompresí
- Podpora více barevných modelů i s možností kanálu pro průhlednost
- Mnohem lepší kompresní poměr o stejné kvalitě jako JPEG
- Lze editovat pouze vybrané regiony obrázku bez nutnosti rekomprese všech ostatních
- Podpora variabilní bitové hloubky
- Absence bločkových artefaktů

Pro všechny tyto a mnoho jiných výhod má tato metoda mnohem vyšší implementační, časovou a paměťovou náročnost. I kvůli těmto důvodům se JPEG 2000

nikdy nestal populárním. Využití nachází např. ve filmovém průmyslu a zdravotnictví. [13]

3.2.3 WebP

WebP je open-source obrázkový formát vyvinutý firmou Google v roce 2010. Formát byl postupně vylepšován a v roce 2018 vyšla verze 1.0. V současné době je podporován většinou populárních internetových prohlížečů, včetně prohlížeče Safari. Popularita a využití tohoto formátu znatelně roste. Využívají ho např. Facebook a Youtube.

Umožňuje ztrátovou i bezztrátovou kompresi, tedy dokáže nahradit jak JPEG, tak i formát PNG. WebP nabízí oproti JPEG okolo 30% zmenšení velikosti komprimovaných obrázků se stejným zkreslením. Podporuje tvorbu animací a přidání alpha kanálu (průhlednosti).

V současnosti se chystá nová generace formátu s názvem WebP 2. [9]

3.2.4 AV1

AV1 (AOMedia Video 1) je video kodek původně určený pro streamování videa přes internet. Vytvořen byl v roce 2018 neziskovou organizací Aliance for Open Media. Jeho hlavní výhoda spočívá v tom, že ho lze použít bez licenčích poplatků. Dosahuje podobných výsledků jako jeho placená varianta H.265.

Z AV1 je odvozen obrázkový formát AVIF (AV1 Image File Format), jehož finální verze byla vydána v roce 2019. Podporuje prakticky stejné možnosti komprese jako výše uvedené metody. Poskytuje jeden z nejlepších možných kompresních poměrů. Výsledné obrázky jsou bez viditelných artefaktů. [8]

3.2.5 Shrnutí kompr. metod obrazu

Jak lze vidět, tak i když nové kompresní metody, téměř ve všech případech, převálcují stařičkou kompresi JPEG, tak je tato metoda stále jednou z nejpopulárnějších, jelikož je její podpora zaručená prakticky ve všech digitálních zařízeních a softwarech.

3.2.6 Výběr vhodného kompresního algoritmu

Pro splnění zadání této práce se hodí taková metoda komprese, která je dostatečně přímočará, tedy nepříliš implementačně složitá pro návrháře. Metoda by měla být

časově nenáročná a zároveň lehce paralelisovatelná, aby bylo možné plně využít programovatelného hradlového pole.

Pro svou relativní jednoduchost, v porovnání s ostatními kompresemi, byla vybrána metoda JPEG. Z důvodu možnosti paralelizace necháme uživatele vybrat, zdali chce kompresi s DC diferencí, nebo bez DC diference.

4 Vlastní řešení

4.1 Programování kompresního algoritmu

Metoda JPEG je určena primárně na obyčejné barevné obrázky. Podle požadavků této práce bude výsledný program zpracovávat obrázky černobílé. Bude tedy nutné zpracovávat pouze jeden barevný kanál, což značně zrychlí výslednou kompresi.

Jelikož jsme omezení výpočetním výkonem a architekturou vestavného zařízení, nebude možné jednoduše vytvářet Huffmanův strom pro individuální snímky. Budeme tedy používat předzpracované standardní JPEG Huffmanovy tabulky, což lehce sníží výsledný kompresní poměr.

4.1.1 Programování metody JPEG

Metoda JPEG se skládá z několika přímočarých kroků, které lze od sebe jednoduše oddělit. Jelikož nám jde o snadnou přenositelnost do nástroje HLS, neměly by se tyto kroky, za cenu jejich čitelnosti a složitosti, příliš optimalizovat. V HLS nelze dynamicky alokovat paměť, proto by se v kódu enkodéru neměla nikde vyskytovat.

Nově napsaná knihovna nese název *libjpeg_jm*.

Pro načítání vstupních obrázků byla využita externí knihovna *Std_image*. Pixely vstupních obrázků musí být seřazeny řádek po řádku ze shora, tedy první pixel v poli je nejvíce horní pixel vlevo a poslední pixel v poli je nejvíce dolní pixel vpravo. Barevné obrázky jsou převáděny do černobílé podoby pomocí lineární aproximace konverze jasové a luminanční percepce, tak jako je tomu v prvním kroku metody JPEG. Ta se provádí podle následujícího vzorce

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B. \tag{4.1}$$

Jak lze vidět ve vzorci, největší důraz se klade na barvu zelenou. To je, jak bylo v úvodu naznačeno, zejména kvůli tomu, že zelenou barvu vnímáme nejvíce.

Pro uložení enkódovaného obrázku v paměti je vytvořeno pole *bit_stream_buffer*

typu integer, jehož maximální velikost je nutné měnit v závislosti na rozměrech a obsahu vstupního obrázku. Velikost zajišťující bezproblémový běh programu se dá vypočítat jako

$$buffer_size = width \cdot height \cdot \frac{51}{64}, \tag{4.2}$$

kde $buffer_size$ je velikost pole, $width$ je šířka a $height$ je výška vstupního obrázku. Velikost se samozřejmě dá nastavit i na hodnotu značně převyšující potřebné místo, aby nebylo nutné program kompilovat při každé změně vstupního obrázku. Obrázky nemusí mít rozměry dělitelné osmi, jelikož jim program automaticky přidá, v případě potřeby, řádky a sloupce černých pixelů.

Před samotným enkódováním je nutné inicializovat globální proměnné pomocí funkce *initializeGlobalVariables*.

Binární podoba enkódovaného obrázku obsahuje následující hlavičku 4.1.

Tabulka 4.1: Hlavička enkódovaného obrázku

Jméno	**Typ**	**Hodnota**
Identifikátor	char[5]	'BOREC'
Šířka	int	-
Výška	int	-
Kvalita	char	<0; 100>
Fixed	char	<−1; 32>
DC rozdíl	char	{0; 1}

K tomu, co značí hodnota $Fixed$, se dostaneme později. Po hlavičce následují zakódované data. Výstupem programu je soubor *output.bor*, pokud není specifikováno jinak.

Pro snadné ovládání výsledného programu bylo vytvořeno jednoduché konzolové rozhraní. Argumentem --*help* se uživateli zobrazí nápověda se všemi možnými způsoby ovládání programu. Program disponuje enkodérem i dekodérem, aby se dala ověřit funkčnost a účinnost komprese.

Ukázka běhu programu 4.1:

```
c:\Jachym>Encoder.exe -e -i lenna.bmp -q 80 -r
--------IMAGE PROPERTIES-------
 Width:                 512
 Height:                512
 Bytes:                 262144
------------RESULTS------------
 ENCODING took 0.007000 seconds to execute
 Original image size:   2097152 bits
 Encoded image size:    298389 bits + 128 bits
 Compress ratio:        7.03
```

Zdrojový kód 4.1: Ukázka běhu programu

Jak lze vidět, tak enkódování obrázku Lenna 5.3, s kvalitou komprese 80 na procesoru Intel Core i5 2.40Ghz, trvala 7 ms. Kompresní poměr mezi původním a výstupním obrázkem je 7,03.

Výsledky komprese 4.2 dvou obrázků s různým nastavením porovnáme s knihovnou libjpeg-turbo, která v současné době dosahuje nejrychlejších výsledků, co se komprese JPEG týče. Knihovnu libjpeg-turbo lze optimalizovat pro zvolené zařízení. Kvůli složitosti a s tím spojené časové náročnosti byl tento krok vynechán. Proto jsou výsledky měření spíše jen orientační. Změřené časy měří čas strávený procesorem, a to pouze samotný běh komprese bez režie, tedy bez načtení a předzpracování obrázku a zapsání komprimovaného obrázku do souboru. Jako kompilátor bylo použito GCC s nastavením optimalizace *-o3*.

Tabulka 4.2: Porovnání libjpeg_jm s libjpeg-turbo

Obrázek	Rozměry	Typ kompr.	Kvalita	Čas	KP	PSNR
Lenna	512 × 512	libjpeg_jm	80	0,007 s	7,03	38,559 dB
-	-	-	90	0.008 s	4,49	40,786 dB
-	-	libjpeg-turbo	80	0,001 s	6,91	38,521 dB
-	-	-	90	0,001 s	4,39	40,653 dB
Rytíř	3504 × 4960	libjpeg_jm	80	0,461 s	5,76	44.773 dB
-	-	-	90	0,502 s	4.47	47.608 dB
-	-	libjpeg-turbo	80	0.043 s	5.70	44.662 dB
-	-	-	90	0.046 s	4.35	47.346 dB

V případě porovnávání kompresního poměru je důležité si uvědomit, že knihovna libjpeg-turbo výstupní formát má mnohem větší hlavičku a různé popisné značky. Nehledě na to, že může mít mnohem jiné kvantizační tabulky, tudíž hodnota kvality může být zavádějící. I přes to je zajímavé, že má můj formát výrazně větší kompresní poměr i hodnotu PSNR, tedy že dochází k menšímu zkreslení. Ve výsledku lze vidět, že je libjpeg-turbo přibližně 10krát rychlejší.

Program jsem analyzoval pomocí nástroje Code profiler 4.2. Obrázek Lenna jsem enkódoval ve smyčce o 20 iteracích s kvalitou 80. Funkce pro násobení matic trvaly nejdéle, což není nic překvapivého, jelikož kód pro maticové násobení není vůbec optimalizovaný, resp. se skládá pouze ze tří jednoduchých smyček. Tato skutečnost bude velkou výhodou při syntetizování. Přesně k takovýmto kalkulacím je FPGA vhodné.

```
1 Each sample counts as 0.01 seconds.
2   \%   cumulative   self              self     total
3  time   seconds   seconds    calls  ms/call  ms/call  name
4  26.67      0.08      0.08    81920     0.00     0.00  multiplySquareMatrices
5  20.00      0.14      0.06    81920     0.00     0.00  multiplySquareMatrices_double
6  16.67      0.19      0.05    81920     0.00     0.00  quantize
7  13.33      0.23      0.04  2119820     0.00     0.00  push_bits
```

Zdrojový kód 4.2: PC profiler

4.1.2 Programování pro vnořená jádra v Zynq

K vývoji použijeme vývojové prostředí Xilinx SDK, dnes již nahrazené prostředím Vitis. Jelikož se v Xilinx SDK pracuje s jazykem C, nemělo by být problémem použít knihovnu libjpeg_jm. Stačí linknout knihovnu math argumentem *-m*.

Abychom zajistili přenositelnost programu vytvoříme jednoduchý obrázkový formát, který snadno v Zynq načteme. V tabulce 4.3 můžeme vidět, jsou obrazová data ve formátu uložena. PC verzi programu přidáme možnost konvertování obrázků do této podoby. Argumentem *--binarydump* přepneme program do režimu konvertování. Použitím argumentu *-dc* konverzi otočíme, tedy budeme náš nový formát exportovat do formátu běžného. Pomocí argumentu *--little-endian* a *--big-endian* můžeme přepnout endianitu, která definuje pořadí zápisu jednotlivých bytů. Big-endian značí, že se zapisuje od nejvýznamnějšího bytu, tedy že číslo zapisujeme tak, jak jsme zvyklí na papíře. S little-endian je to opačně.

Tabulka 4.3: Hlavička obrázkového formátu

Jméno	**Typ**
Šířka	int
Výška	int
Kanály	uint8_t
Bit. hloubka	uint8_t
Pixely	uint8_t [ ]

Obrázky budeme načítat z SD karty. Výsledek budeme ukládat rovněž na SD kartu. K tomu využijeme knihovnu xillfs, která poskytuje FatFs souborový systém, je tedy potřeba dávat pozor na používání zakázaných znaků v názvech souborů.

Z tabulky 4.4 lze vyčíst, že algoritmus je přibližně 60krát rychlejší na PC než na Zynq s ARM Cortex-A9 866MHz procesorem.

Algoritmus jsem analyzoval pomocí nástroje TCF Profiler o stejných podmínkách jako v případě na PC 4.2. Na obrázku 4.1 můžeme vidět, že je zátěž funkcí prakticky totožná. Nejvíce náročné jsou funkce na násobení matic, resp. funkce na výpočet DCT.

Tabulka 4.4: Porovnání PC s platformou Zynq

Obrázek	Rozměry	Zařízení	Kvalita	Čas
Lenna	512 × 512	Zynq	80	0,450 s
-	-	-	90	0,479 s
-	-	PC	80	0,007 s
-	-	-	90	0,008 s
Rytíř	3504 × 4960	Zynq	80	30,548 s
-	-	-	90	31,303 s
-	-	PC	80	0,461 s
-	-	-	90	0,502 s

Profiler running. 151 samples

Address	% Exc...	% Incl...	Function	File
00102980	32.4		multiplySquareMatrices	encoder.c
00102eac	29.8		multiplySquareMatrices_double	encoder.c
001049a0	10.5		push_bits	huffman.c
00137800	7.28		XSdPs_WritePolled	xsdps.c
00103264	5.96		quantize	encoder.c
00126f0c	3.97		__fixdfdi	
001037a4	1.98		zeroRle	encoder.c
00121c58	1.98		getImagePart	utils.c
00121d2c	1.32		convertToInt8	utils.c
00126eb0	1.32		__aeabi_l2d	

Obrázek 4.1: Zynq profiling

4.2 Programování pro programovatelné hradlové pole

K tomu, abychom mohli kód spustit na FPGA, potřebujeme nejdříve využít programu Vivado HLS. Nástroj HLS nám umožní kód transformovat do RTL implementace, aniž bychom museli práci s HDL jakkoliv znát. HLS podporuje jazyky ANSI-C, C++ a SystemC. Nepodporuje dynamické alokování paměti a systémové operace. Tato omezení by neměla být překážkou, jelikož s nimi bylo počítáno již od počátku psaní práce. Potíží bude ale jakékoliv dělení a práce s proměnnými typu float, které se v kódu vyskytují. Syntetizované bločky, které tyto operace provádí, potřebují po-

měrně velké množství zdrojů. Proto bychom měli kód předělat takovým způsobem, aby se v něm tyto operace nevyskytovaly. K tomu můžeme využít proměnných typu fixed.

4.2.1 Využití proměnných typu fixed

Fixed proměnné mají pevně daný počet míst na celou a desetinnou část čísla, na rozdíl od proměnných typu *float*, které mají pohyblivou řádovou čárkou. Konverzi mezi těmito dvěma typy provedeme prostým vynásobením libovolným číslem. Určíme si typ *fixed* v binární soustavě, ve kterém budeme mít 6 bitů na celou část čísla a 2 bity na desetinnou část čísla, tedy dohromady 8 bitů, které se vejdou do proměnné typu *char*. Posléze můžeme např. číslo 5,625, které v binární soustavě zapíšeme jako

$$101,101_2, \tag{4.3}$$

konvertovat do podoby

$$00010110_2, \tag{4.4}$$

ve které poslední dva bity značí desetinnou část, tedy jsme přišli o část původního čísla, a to konkrétně o hodnotu 0,125. Výsledné číslo je po převedení 5,5.

Tuto operaci konverze můžeme provést prostým vynásobením specifickým číslem. Např. v binární soustavě budeme násobit mocninami dvou a v desítkové soustavě mocninami desítky. S tímto číslem můžeme dál libovolně manipulovat. Pro následné navrácení čísla do původní podoby stačí číslo vydělit stejnou hodnotou, jakou jsme násobili. To ale nekoresponduje s naším původním úmyslem převodu do typu *fixed*. Naštěstí existuje speciální operace binárního posuvu, díky které budeme moct nahradit násobení i dělení.

Binární posuv číslo v binární podobě posouvá mezi bity takovým způsobem, že buď číslo vynásobí, nebo vydělí dvěma, tedy vlastně násobí mocninami dvou, jako je znázorněno v následujícím příkladu

$$00101111_2 << 2 = 47 \cdot 2^2 = 10111100_2 = 188, \tag{4.5}$$

kde $<< 2$ značí binární posuv vlevo o 2 bity. Analogicky můžeme využít binárního posuvu vpravo o 2 bity pomocí $>> 2$. Tato operace je výpočetně nenáročná.

Tímto jsme se tedy zbavili problému použití proměnných typů *float*, ale stále je zde potíž ohledně dělení jako takového. To lze vyřešit inverzní hodnotou, tedy např. číslo 16, kterým chceme dělit, převedeme na číslo $\frac{1}{16}$ a místo abychom jím

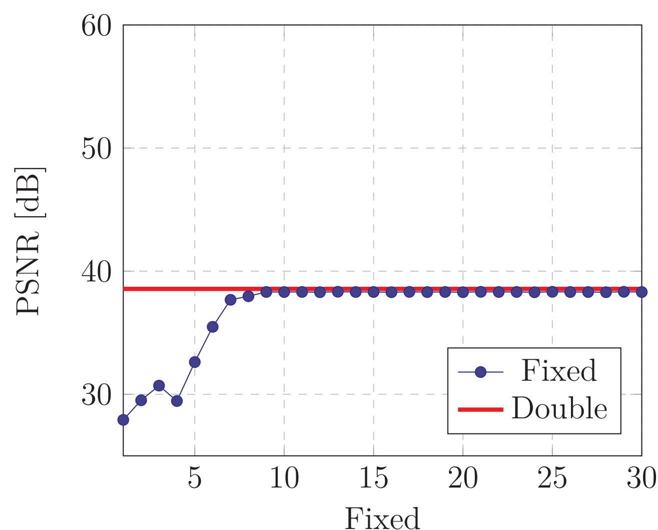
dělili, budeme jím násobit. Číslo 0,0625 z typu *float* převedeme na typ *fixed* a tím je problém vyřešen.

Matice G 3.6, používané na provedení DCT, převedeme na typ *fixed* s počtem 16 bitů k uchovávání desetinné části 4.2.1, tedy matice vynásobíme číslem 2^{16}, aby hodnoty plně využívaly proměnné typu *short*, jelikož všechny tyto hodnoty mají pouze desetinnou část, resp. jsou v rozmezí $(-1; 1)$. Kvantizační matici 3.5, kterou se dělí, převedeme hodnoty do inverzní podoby a také vynásobíme číslem 2^{16}.

$$G_M_F = \begin{pmatrix} 23170 & 23170 & 23170 & 23170 & 23170 & 23170 & 23170 & 23170 \\ 32138 & 27246 & 18205 & 6393 & 6393 & -18205 & -27246 & -32138 \\ 30274 & 12540 & -12540 & -30274 & -30274 & -12540 & 12540 & 30274 \\ 27246 & -6393 & -32138 & -18205 & 18205 & 32138 & 6393 & -27246 \\ 23170 & -23170 & -23170 & 23170 & 23170 & -23170 & -23170 & 23170 \\ 18205 & -32138 & 6393 & 27246 & -27246 & -6393 & 32138 & -18205 \\ 12540 & -30274 & 30274 & -12540 & -12540 & 30274 & -30274 & 12540 \\ 6393 & -18205 & 27246 & -32138 & 32138 & -27246 & 18205 & -6393 \end{pmatrix} \tag{4.6}$$

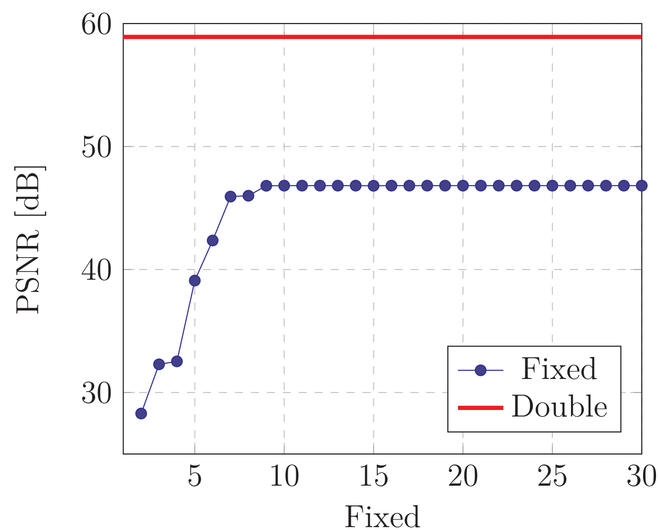
Aby nebylo nutné při každém volání funkce kalkulovat nové hodnoty kvantizační tabulky v závislosti na zvolené kvalitě, předpřipravíme matice s hodnotou kvality 10, 20, ... , 100.

Práce s *fixed* proměnnými nám logicky bude lehce měnit výstup algoritmu, jelikož zmenšujeme přesnost čísel s desetinnou čárkou. Abychom mohli jednoduše porovnat závislost množství zkreslení na velikosti *fixed*, tak knihovně libjpeg_jm přidáme možnost komprese ve *fixed* formátu. Možnost přidáme enkodéru i dekodéru. Argumentem --*fixed* <*hodnota*> přepneme program do tohoto režimu.



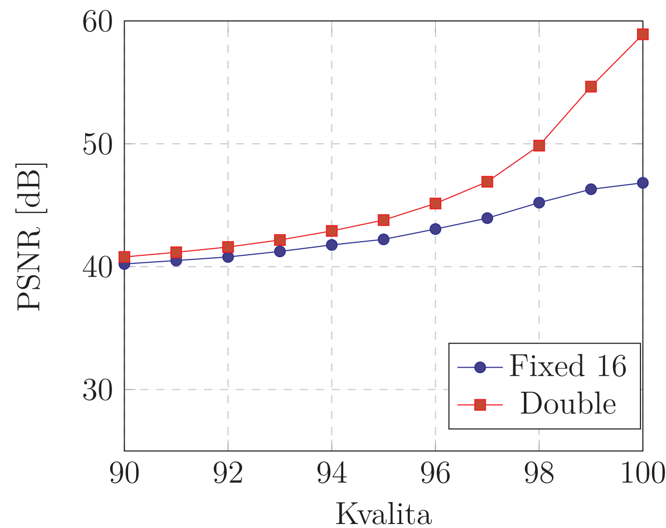
Graf 4.1: Porovnání fixed s kvalitou kompr. 80

Porovnání *fixed* nastavení 4.1 bylo provedeno na obrázku Lenna s kvalitou komprese 80. V grafu lze vidět, že se hodnota PSNR ustálí od hodnoty *fixed* 9. PSNR obrázku komprimovaného s proměnnými typu *double* se prakticky vůbec neliší od obrázku komprimovaného s hodnotu *fixed* 9 a více.



Graf 4.2: Porovnání fixed s kvalitou kompr. 100

Další porovnání *fixed* nastavení 4.2 bylo porovnáno na obrázku Lenna s kvalitou komprese 100. Hodnota PSNR se ustálí rovněž od *fixed* 9. PSNR obrázku komprimovaného s proměnnými typu *double* je značně rozdílné od metody s typem *fixed*. Provedeme další měření, abychom zjistili, od jaké hodnoty kvality komprese se PSNR začíná lišit.



Graf 4.3: Porovnání PSNR metod fixed a double

Hodnota PSNR se podle grafu 4.3 začne výrazně lišit v bodě kvality 95 a dále. To značí, že pokud chceme obrázek zkomprimovat s co nejmenším zkreslením, musíme použít metody s typy *double*. Jak již bylo řečeno, komprese obrázků s kvalitou v rozsahu 95 až 100 je ve většině případů zbytečná, tudíž by nepřesnost metod *fixed* v tomto rozsahu neměla být příliš velkým problémem.

4.2.2 Práce s HLS

Nástroj HLS poskytuje snadné testování výsledků syntetizované funkce. V tzv. Test Bench vytvoříme testovací metodu, která zavolá funkci k syntetizování a výstup porovná s přiloženými referenčními daty. Tento proces je nezbytný k zajištění správnosti výsledné RTL implementace, jelikož kroky nutné k finálnímu spuštění algoritmu na FPGA jsou časově velmi náročné.

Pomocí HLS můžeme využít tzv. pipelining (též zřetězené zpracování), pomocí kterého si můžeme práci s daty připodobnit zpracování produktů na běžícím páse. Tedy pokud máme proces skládající se z několika kroků, jako např.:

- polož brambor na pás,
- umyj brambor,
- oloupej brambor,
- nakrájej brambor

a máme takto zpracovat velké množství bramborů, tak je samozřejmě výrazně rychlejší hlízy zpracovávat bezprostředně za sebou, než čekat, až tímto procesem projde jeden brambor, a až poté zpracovat tímtéž procesem brambor další, resp. aby se krájela jedna hlíza a zároveň loupala druhá atd. Analogicky můžeme proces připodobnit zpracování čísel v FPGA. Pipelining samozřejmě nejde využít v každé situaci, např. když je výpočet jednoho čísla závislý na výpočtu čísla druhého. [14]

Tato a mnoho jiných optimalizací lze v kódu provést jednoduchým přidáním direktiv. V případě zřetězeného zpracování se jedná o direktivu *#pragma HLS PIPELINE*, kterou vložíme do smyčky, ve které chceme data tímto způsobem zpracovat.

Kód knihovny libjpeg_jm bylo, až na výjimky, velice snadné uzpůsobit nástroji HLS. Jelikož má HLS problém s nepřímou adresací druhého řádu (ukazatele, které ukazují na jiné ukazatele), bylo nutné přepsat funkce pracující s bit streamem, které této skutečnosti využívaly. Nejen že stačilo po přepsání těchto metod kód prakticky jen zkopírovat a vložit, ale i jejich rychlost na PC se mnohonásobně zrychlila (veškeré měření v této práci bylo prováděno s finální verzí kódu). Příklad jedné z těchto metod můžete shlédnout v příloze 5.1. Kód dekomprese kvůli nadbytečnosti změněn nebyl, proto se v kódu knihovny vyskytují dva druhy práce s bit streamem.

V HLS se musí určit tzv. Top Function 4.3, která bude fungovat jako rozhraní RTL bloku. Funkce *encodeImage* přijímá 5 parametrů popsaných v tabulce 4.5. Pátou proměnnou *ouput_length* není nutné nijak nastavovat. Po dokončení funkce stačí z proměnné přečíst hodnotu.

```
1  void encodeImage(volatile unsigned char *input, volatile uint32_t* output, uint32_t image_width, uint8_t ...
       quality, uint32_t* output_length)
2  #pragma HLS INTERFACE m_axi depth=4096 port=input offset=slave bundle=AXIM_1
3  #pragma HLS INTERFACE m_axi depth=4096 port=output offset=slave bundle=AXIM_1
4  #pragma HLS INTERFACE s_axilite port=return bundle=AXI_Lite_1
5  #pragma HLS INTERFACE s_axilite port=image_width bundle=AXI_Lite_1
6  #pragma HLS INTERFACE s_axilite port=quality bundle=AXI_Lite_1
7  #pragma HLS INTERFACE s_axilite port=output_length bundle=AXI_Lite_1
```

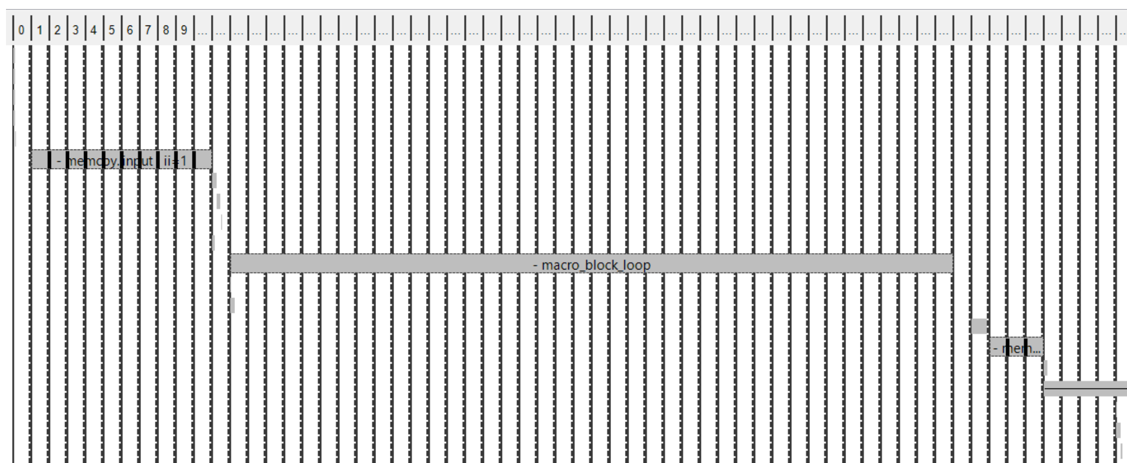
Zdrojový kód 4.3: HLS Top Function

Pod deklarací funkce lze vidět direktivy, které určují druh rozhraní pro daný parametr. Rozhraní AXIM je takové rozhraní, které má přímý přístup do externí paměti (DMA), tedy je např. vhodné pro co nejrychlejší přenos velkého množství dat. Rozhraní AXI-lite je vhodné zejména pro přenos individuálních řídících dat.

Tabulka 4.5: Parametry *encodeImage* funkce

Název	Typ	Popis
input	unsigned char*	vstupní hodnoty pixelů obrázku
output	unsigned int*	ukazatel na pole k uložení enk. obrázku
image_width	unsigned int	šířka obrázku
quality	unsigned char	kvalita komprese v rozsahu 1, 2, ..., 10
output_length	unsigned int*	délka enk. řádku macro bloků

Kód jsem po syntetizování analyzoval pomocí HLS nástroje Analysis 4.2. Načítání obrazových dat, které je znázorněno prvním obdélníčkem, zabírá poměrně značnou část procedury. To samé lze říci o zapisování dat výstupních. Je důležité si uvědomit, že čas režie přesunu těchto dat není zanedbatelný. Proto by bylo vhodné na vstup posílat více než jeden 8×8 blok (macro blok) pixelů. Funkce tedy byla implementována tak, aby zpracovávala celý řádek macro bloků vstupního obrázku 4.2.2. Očíslované čtverečky značí jednotlivé 8×8 macro bloky. V tomto případě se tedy jedná o obrázek o šířce a výšce 40 pixelů.

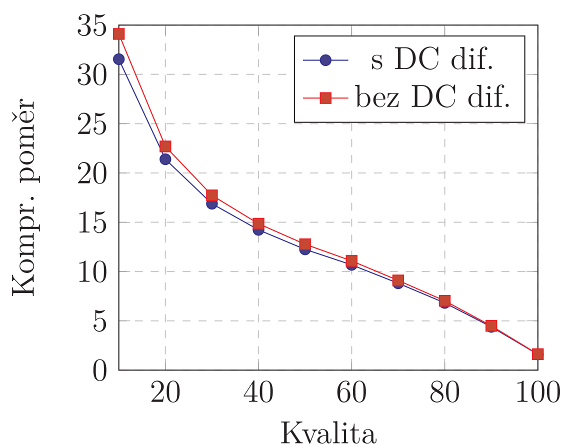


Obrázek 4.2: HLS Analysis

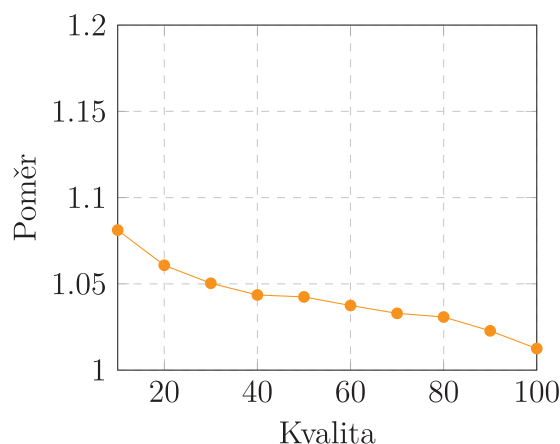
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Obrázek 4.3: Řádek macro bloků

Kdybychom zpracovávali celý obrázek najednou, nebylo by možné jednoduše využít paralelizace. To samé platí, jak již bylo řečeno, i s DC diferencí, kde je vstup pro Huffmanovo kódování jednoho macro bloku závislý na výpočtu macro bloku předešlého. Proto bude v kódu HLS DC diference vynechána.



Graf 4.4: Porovnání kompr. poměru s DC rozdílem a bez DC rozdílu



Graf 4.5: Poměr kompr. poměrů s DC rozdílem a bez DC rozdílu

Na grafu 4.2.2 lze vidět komprese obrázku Lenna s DC rozdílem a bez DC rozdílu. Na grafu 4.2.2 je zobrazen poměr mezi kompresí s DC rozdílem a bez DC rozdílu. Je patrné, že zlepšení kompresního poměru výrazně klesá se zvyšující se kvalitou komprese. To je zejména kvůli tomu, že se při vyšších kvalitách vynuluje méně AC složek, na které DC diference nemá vliv.

Jelikož není možné dynamicky alokovat paměť, je nutné předem nastavit velikost globálních bufferů. Velikost bufferu na uložení vstupních dat 4.7 se vypočítá jako

$$LOCAL_IMAGE_BUFFER_LENGTH = 8 \cdot image_width, \tag{4.7}$$

kde $image_width$ je šířka komprimovaného obrázku. Zároveň je nutné nastavit velikost bufferu na uložení výstupních enkódovaných dat, která se vypočítá jako

$$LOCAL_STREAM_LENGTH = \frac{51}{8} \cdot image_width. \tag{4.8}$$

Stejně jako v případě na PC 4.2 se jedná o nejhorší možnou maximální velikost, tedy je možné velikost přizpůsobit v závislosti na entropii vstupních dat a zvolené kvalitě komprese.

Několik kroků metody JPEG bylo v HLS spojeno dohromady kvůli optimalizaci, konkrétně normalizace, dct, kvantizace a zig-zag vyčítání.

4.2.3 Práce s Vivado

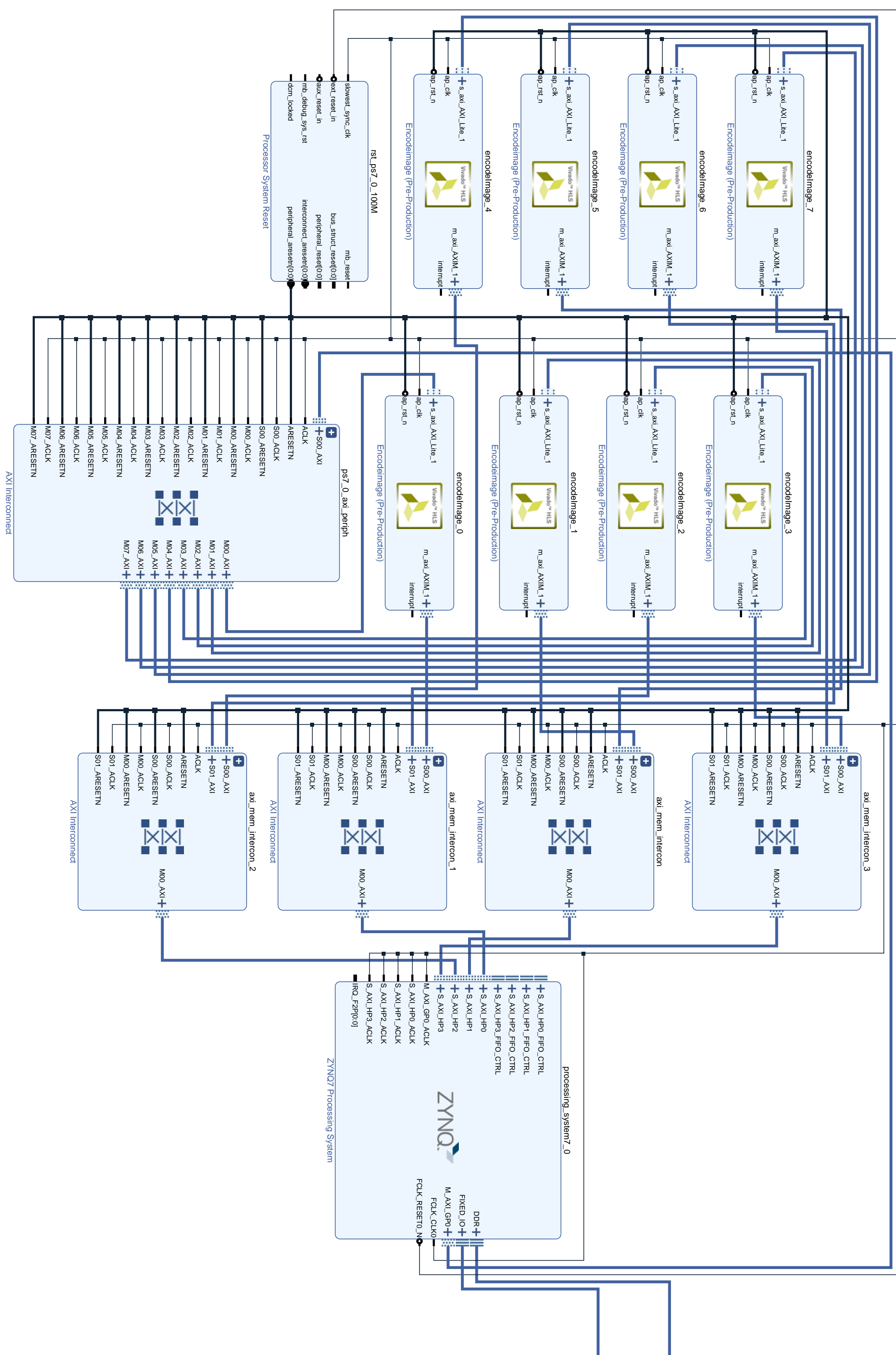
Exportovaný IP blok z HLS vložíme do programovatelné části obvodu Zynq pomocí programu Vivado. Program poskytuje intuitivní grafické rozhraní, pomocí kterého lze IP jádra jednoduše zapojovat. Díky možnosti automatického propojení stačí uživateli opravdu minimální znalosti práce s hardwarem. Do tzv. Block Design stačí vložit bloček komprese a bloček zařízení Zynq a Vivado se postará o vše nutné nutné k jejich zapojení, a to i včetně sběrnicových a jiných řídících bločků. Pokud změníme kód HLS a následně znovu exportujeme RTL, program Vivado nám umožní bločky v designu jednoduše aktualizovat.

Abychom mohli využít paralelizace, musíme bločku sběrnice nastavit počet Slave rozhraní a následně bloček v libovolném množství zkopírovat. Vivado znovu nabídne možnost automatického propojení. Finální schéma lze vidět na obrázku 4.2.3.

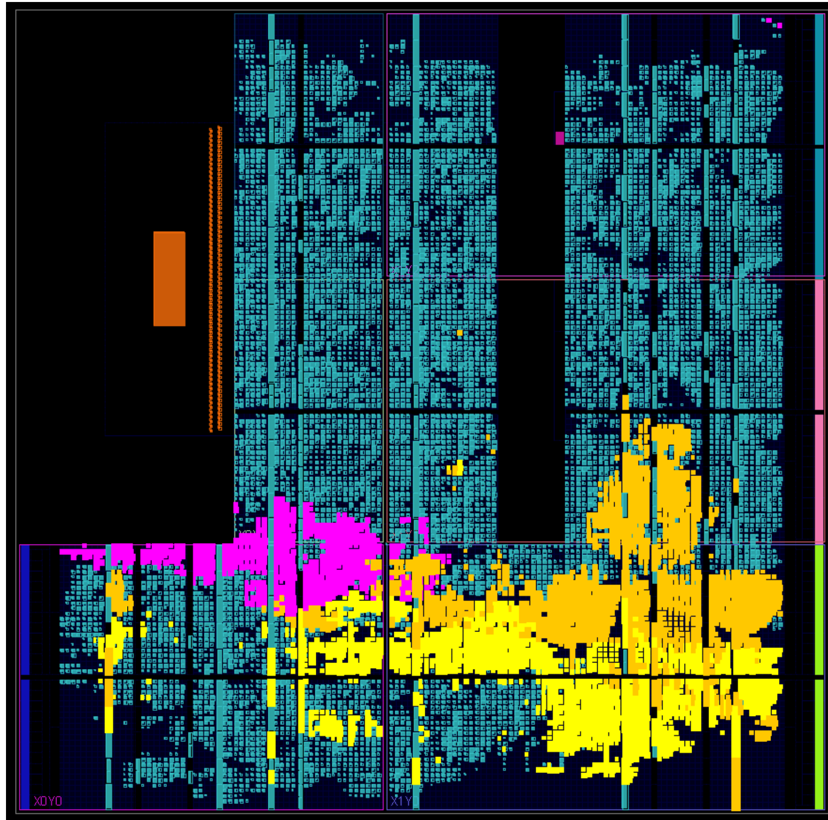
Posledním krokem je generace Bitstreamu, pomocí kterého budeme moct naprogramovat FPGA v Zynq. Na obrázku 4.5 lze vidět finální formu programovatelného hradlového pole. Tento proces, v závislosti na složitosti projektu, může trvat až několik desítek minut, jelikož se hledá nejlepší možné propojení buněk na FPGA. Z tohoto důvodu je zásadní využívat HLS Test Bench.

S osmi *encodeImage* bločky zapojenými do 4 sběrnic je využito prakticky celé FPGA. Růžovou barvou je zvýrazněna 1. AXIM sběrnice. Žlutě a oranžově jsou zvýrazněny buňky 1. a 2. *encodeImage* bločku. Každý *encodeImage* design je složen z 306 buněk a 251 propojů.

Jelikož chceme využít co nejvíce možnosti paralelizace, bylo potřeba omezit maximální možnou šířku řádku macro bloků na 1600 pixelů, neboli nastavit $LOCAL_IMAGE_BUFFER_LENGTH$ na hodnotu $64 \cdot 200$. Při vyšších hodnotách totiž nestačilo množství paměťových bločků, které má FPGA v Zynq k dispozici.



Obrázek 4.4: Blokové schéma zapojení



Obrázek 4.5: Buňky v FPGA

4.2.4 Využití FPGA

Z programu Vivado exportujeme hardwarové specifikace včetně Bitstreamu a volbou *launch SDK*, nacházející se pod exportem, spustíme prostředí Xilinx SDK. Vytvoříme nový aplikační projekt a v případě nutnosti (při implicitním nastavení by mělo být všechno potřebné již zvoleno) zvolíme námi požadovanou hardware platformu. Projekt dále nastavíme tak, jako tomu bylo v implementaci bez využití FPGA 4.1.2.

Knihovna vytvořená programem Vivado nese název *x<název projektu>.h*. V našem případě se jedná o knihovnu *xencodeimage.h*. Veškeré funkce pro komunikaci s FPGA začínají prefixem *XEncodeimage_*.

Vytvoříme tolik instancí objektu *XEncodeimage*, kolik jsme jich zapojili v blokovém schématu. Objekty inicializujeme, nastavíme ukazatele na vstup a výstup a nastavíme šířku obrázku.

Nesmíme zapomenout, že produktem tohoto objektu je pole, ve kterém je uložen bitstream komprimovaného řádku macro bloků, tedy pracujeme s přesností na bity. Jelikož chceme využít paralelizace, resp. komprimování více macro bloků zároveň, musíme, podobně jako v případě HLS 4.8, ukazatele na výstup od sebe dostatečně

oddělit. To ale samozřejmě znamená, že výsledný bitstream celého obrázku bude obsahovat veliké mezery. Proto bude nutné před zápisem na SD kartu rozdělené bitstreamy sloučit dohromady. Abychom nemuseli přesouvat jednotlivé bity, což by bylo časově velmi náročné, budeme slučovat k nejbližšímu možnému integeru.

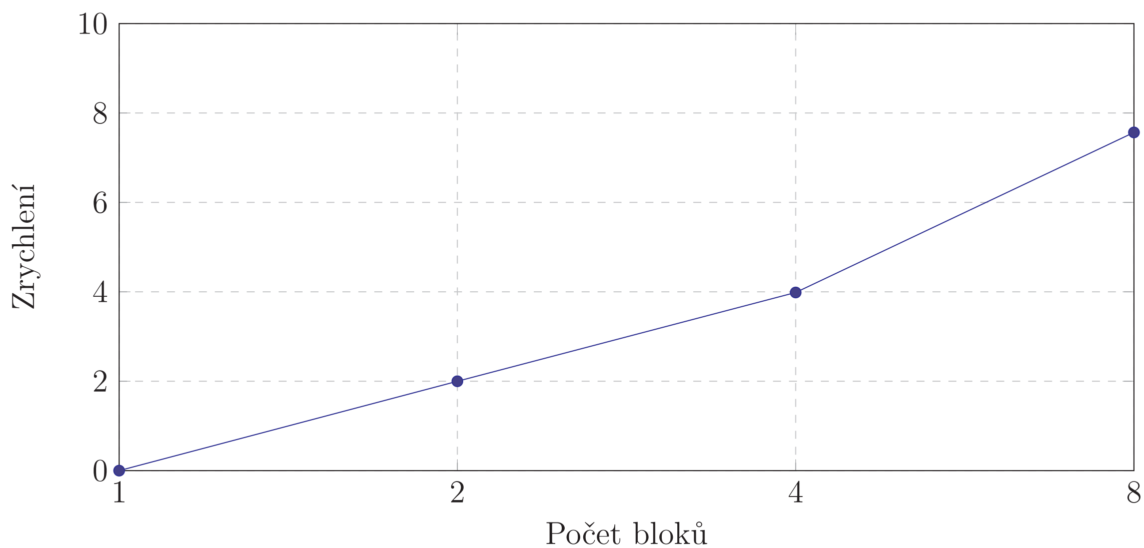
Tímto jsme sice mezeru značně snížili, ale nezbavili jsme se jí. Proto dekodéru na PC přidáme možnost čtení zakódovaných dat ve speciálním režimu, kde po přečtení jednoho řádku macro bloků přeskočí k začátku následujícího integeru. Tento režim zapneme použitím argumentu *-fp*.

Jednotlivé instance *XEncodeImage* spustíme ve speciální smyčce, která jim bude postupně přiřazovat řádky macro bloků v závislosti na tom, zda už svou kompresi řádku dokončili, tedy konečně využijeme možnosti paralelizace.

Jako poslední je nutné vyřešit problém tzv. cachování. Cache (též mezipaměť) je rychlá paměť umístěná mezi procesorem a operační pamětí. Používána je ke zlepšení výkonu tak, že ukládáním již používaných a místně souvisejících dat snižuje latenci přístupu do paměti. Jelikož si rozhraní AXIM sahá přímo do operační paměti bez ohledu na to, co dělá procesor, musíme manuálně provést tzv. cache invalidation a flushing. Invalidace značí, že všechna data v cache v označené oblasti jsou zastaralá, a tedy neplatná, resp. že pokud s nimi chce procesor jakkoliv manipulovat, musí o ně požádat operační paměť. Funkce flush data z cache uloží do operační paměti. Před spuštěním FPGA komprese, resp. po načtení obrázku, zavoláme funkci *Xil_DCacheFlushRange*. Po zkomprimování obrázku zavoláme funkci *Xil_DCacheInvalidateRange*. Pro ušetření času ji můžeme zavolat během samotné komprese.

Stejně jako v předešlých případech se musí manuálně nastavit velikost globálních bufferů k uložení obrázku *RD_BUFFER_SIZE* a *BIT_STREAM_SIZE*.

Před spuštěním naprogramujeme FPGA pomocí dříve vytvořeného Bitstreamu. To lze provést vložením Bitstreamu na SD kartu nebo přímým naprogramováním z prostředí SDK.



Graf 4.6: Porovnání zrychlení doby běhu komprese v závislosti na počtu bloků

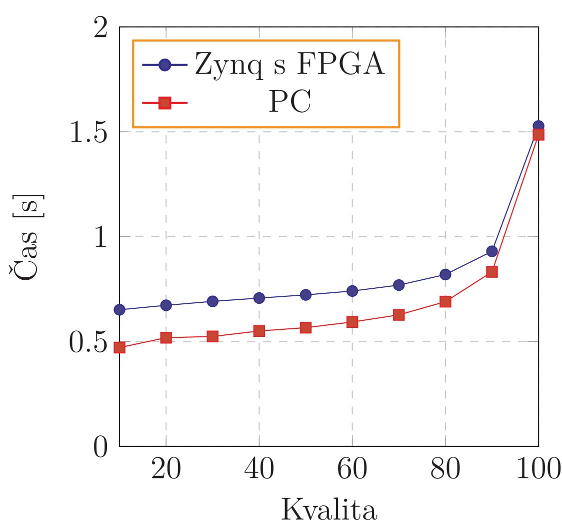
Na grafu 4.6 lze vidět poměr zrychlení v závislosti na počtu použitých *encodeImage* bloků k paralelizaci. Jelikož má Zynq k dispozici pouze 4 sběrnice, bylo nutné pro každý případ vytvořit nové schéma, a tedy vygenerovat nový Bitstream. Z grafu lze vypozorovat, že se zdvojnásobením počtu bloků přibližně zdvojnásobí i výsledné zrychlení komprese. Podle Amdahlova zákona by poměr zlepšení výkonu začal značně klesat s přirůstajícím počtem nových bloků, proto není příliš zásadní, že nemůžeme přidat více jak 8 bloků. [12]

4.3 Výsledky měření

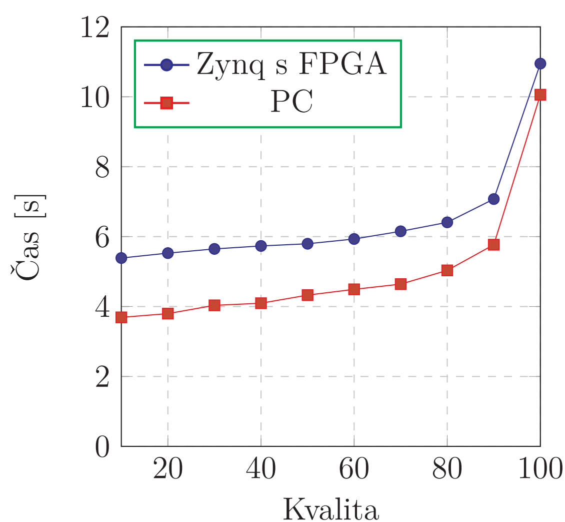
4.3.1 Zynq s FPGA vs. PC

V grafu 4.7 můžeme vidět výsledky doby běhu komprese na Zynq s využitím FPGA (8 *encodeImage* bloků) a na PC s *double* metodami. Komprimován byl obrázek Lenna bez DC diference ve smyčce o 100 iteracích. Čas běhu na Zynq byl měřen včetně invalidování cache a složení Bitstreamu. Jak lze vidět, tak se doba komprese na Zynq s využitím FPGA příliš neliší od doby komprese na PC.

V grafu 4.8 můžeme vidět výsledky komprese obrázku Příroda o stejných podmínkách jako v případě 4.7.



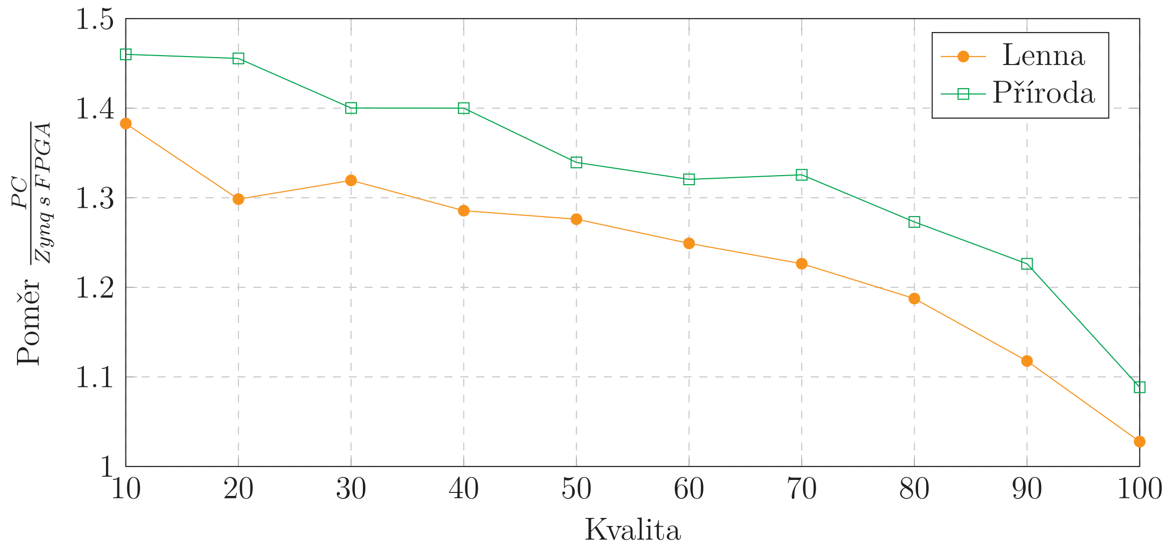
Graf 4.7: Porovnání času běhu komprese obrázku Lenna



Graf 4.8: Porovnání času běhu komprese obrázku Příroda

Pokud bychom čas doby běhu vydělili 100, dostali bychom přibližný čas komprimování jednoho obrázku. Lze tedy tvrdit, že komprese na Zynq s využitím FPGA dokáže zpracovat přibližně 122 snímků za vteřinu, pokud se jedná o obrázky s rozměry 512 × 512 a s kvalitou komprese 80. Obrázky s rozměry 1920 × 1080 zpracuje přibližně 15krát za vteřinu.

Poměry mezi dobou na Zynq s využitím FPGA a PC jsou zobrazeny v grafu 4.9. Můžeme vidět, že čím je zvolena vyšší kvalita komprese, tím dochází k menšímu zrychlení. Zároveň lze vypozorovat, že si verze na PC poradí lépe s většími obrázky. V běžných případech použití komprese, tedy s použitím kvality v rozsahu 80–90 na obrázek menších až středních rozměrů, bude doba běhu komprese na PC pouze přibližně 1,2krát rychlejší.

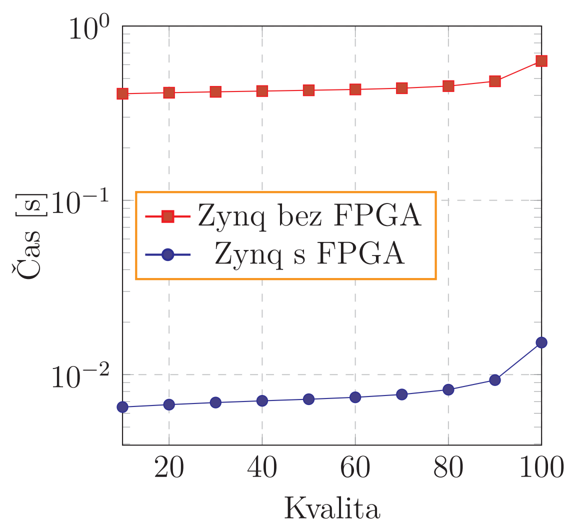


Graf 4.9: Porovnání poměrů časů

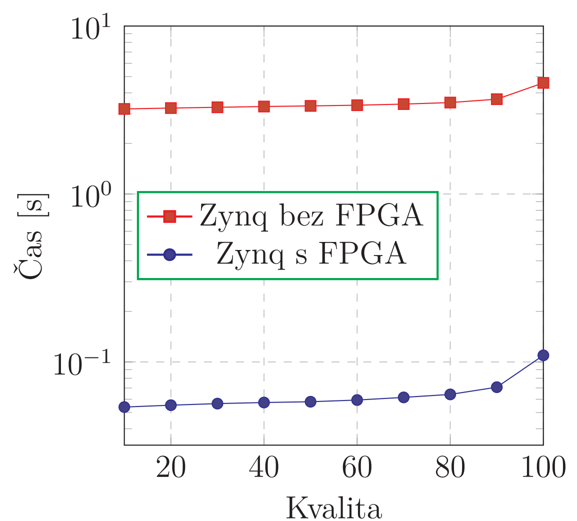
4.3.2 Zynq s FPGA vs. Zynq bez FPGA

Obrázek Lenna jsem komprimoval na Zynq bez využití FPGA, tedy pomocí knihovny libjpeg_jm, a s využitím FPGA. Tentokrát komprese proběhla pouze jednou. Komprese s libjpeg_jm byla provedena metodami *double* bez DC diference. Na grafu 4.10 lze vidět, že se výsledné časy komprese výrazně liší.

Obrázek Příroda jsem komprimoval za stejných podmínkách. Výsledky jsou zobrazeny na grafu 4.11.

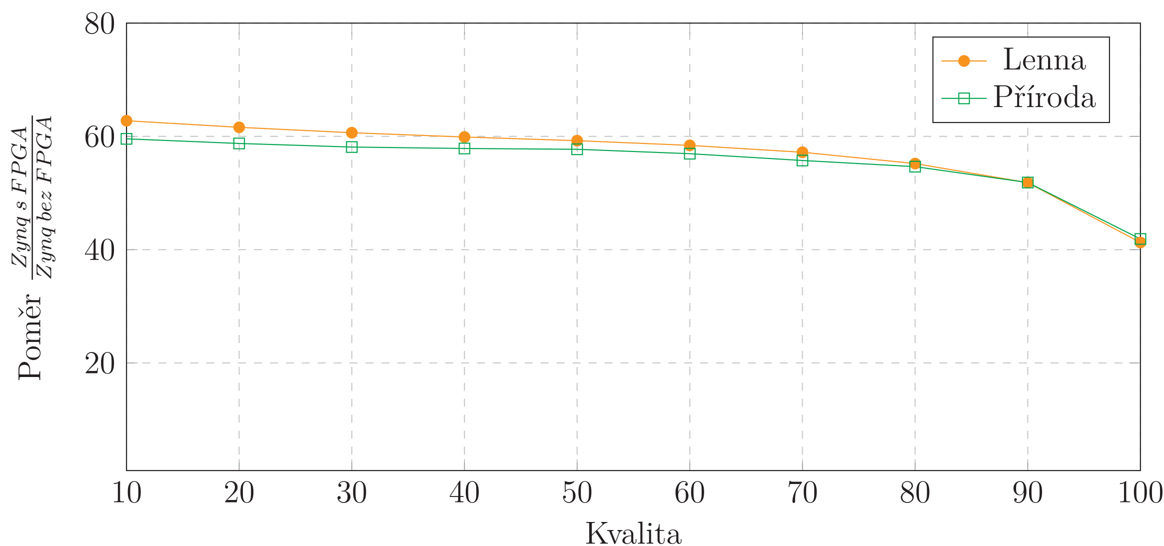


Graf 4.10: Porovnání času běhu komprese obrázku Lenna



Graf 4.11: Porovnání času běhu komprese obrázku Příroda

V grafu 4.12, kde je porovnán poměr doby běhu komprese na Zynq s využitím a bez využití FPGA, lze vidět, že je druhá varianta přibližně 60krát rychlejší, když nepočítáme případ s kvalitou komprese 100. Na rozdíl od PC 4.9, poměr zrychlení nezáleží na velikosti obrázku.



Graf 4.12: Porovnání poměrů časů

4.3.3 Zynq s FPGA vs. Zynq libjpeg-turbo

Dle zkušeností mého vedoucího zvládá Zynq-7000 PS s využitím libjpeg-turbo komprimovat obrázky 640 × 480 s 8bitovou hloubkou přibližně 25–30krát za sekundu, tedy jeden obrázek za 33 ms. Když obrázek 640 × 480 přepočítáme na počet obrazových bodů, vyjde nám, že je 1,17krát větší než obrázek Lenna s rozměry 512 × 512. Ten je podle měření 4.10 možné komprimovat s využitím FPGA 122krát za vteřinu, tedy jeden obrázek za 8 ms. Po přepočtu na rozměry 640 × 480 nám vychází, že by trvalo jeden obrázek o těchto rozměrech komprimovat přibližně 9,3 ms. Z toho bychom mohli usoudit, že je komprese na Zynq s využitím FPGA až 3,5krát rychlejší než na Zynq s využitím libjpeg-turbo.

5 Závěr

5.1 Souhrn práce

Hlavní myšlenka práce byla proveditelná bez výrazných překážek, i přes mé minimální zkušenosti s vestavěnými zařízeními, natož s programovatelnými hradlovými poli. Samozřejmě se musí brát v potaz možnost konzultace s expertem, bez kterého by bylo velice těžké nemálo kroků práce dokončit. Firma Xilinx poskytuje dostatečný počet kvalitně napsaných manuálů a tutoriálů pro práci na platformě Zynq. Dotazů a rad na internetových fórech je také hojné množství, ale většinou se jedná o příspěvky lidí s dlouholetými zkušenostmi s prací s HW, tedy je často těžké z těchto textů něco vyčíst.

Kód z PC do HLS stačilo, až na výjimky, víceméně zkopírovat a vložit. To se samozřejmě nedá říct o následné optimalizaci kódu, ale pokud by měl vývojář nedostatek času, má možnost kód, s ohledem na omezení HLS, prakticky okamžitě převést do HW podoby. Prostředí Vivado HLS se liší jen minimálně od běžných vývojových prostředí. Proces syntetizování výsledného RTL bločku je přímočarý.

S největšími problémy jsem se setkal při psaní kompresního algoritmu jako takového, zejména co se práce s jednotlivými bity a testování správných výstupů programu týče. S HW jsem měl zprvu největší potíže způsobené špatným pracováním s cache pamětí, která se ale dá pro testování kódu vypnout. Programování pro HW má výrazně méně „záchranných sítí“, tedy je mnohem snazší udělat v kódu chybu formou přetečení pole, volbou špatného typu proměnné, atd.

Po několika neúspěšných pokusech zprovoznit algoritmus na FPGA jsem zjistil, jak důležité je využít možnosti HLS Test Bench. Vygenerování Bitstreamu může trvat až desítky minut, tudíž pokud se algoritmus nechová správně, je nutné celý proces opakovat znovu. Za zmínku také stojí problémová komunikace se ZedBoard přes rozhraní UART, při které jsem často musel v operačním systému Windows restartovat příslušné ovladače, aby mohla komunikace znovu probíhat.

5.2 Názor na HLS

Algoritmus komprese knihovny libjpeg_jm 4.2 spuštěný na PC byl přibližně 10krát pomalejší než varianta s knihovnou libjpeg-turbo. Z toho můžeme usoudit, že kdybychom libjpeg-turbo zprovoznili na platformě Zynq, byla by komprese přibližně 10krát rychlejší než s využitím libjpeg_jm. Tedy pokud bychom čas, strávený zprovozněním hardware akcelerátoru, využili na optimalizaci algoritmu knihovny libjpeg_jm, dosáhli bychom v nejlepším případě (berme v potaz, že libjpeg-turbo je, jak název vypovídá, jeden z nejrychlejších algoritmů na kompresi ve formátu JPEG) desetinásobného zrychlení. Zynq-7000 disponuje dvoujádrovým ARM procesorem, tedy teoreticky můžeme algoritmus paralelizovat a přibližně 2krát zrychlit.

To je v porovnání s využitím FPGA, který byl podle měření 4.12 60krát rychlejší než bez využití FPGA, až 3krát pomalejší. Tento odhad se shoduje s odvozením v sekci 4.3.3. A to se bavíme o teoreticky nejlepší možné variantě optimalizace kódu na procesoru. Kód HLS byl dokonce jen minimálně optimalizován, což bylo způsobené zejména nedostatečnými znalostmi s prací s HW.

V závěru tedy vyplývá, že se zlepšení výkonu algoritmu pomocí FPGA, v konkrétním případě naší práce, rozhodně vyplatilo. Dalo by se předpokládat, že by došlo ke stejným výsledkům i při tvorbě jiného vhodného algoritmu.

Použitá literatura

[1] SAYOOD, Khalid. Introduction to Data Compression. 5th Edition. Morgan Kaufmann, 2017. ISBN 9780128094747.

[2] IFEACHOR, Emmanuel C. a Barrie W. JERVIS. Digital signal processing: a practical approach. 2nd ed. Harlow: Prentice-Hall, c2002. ISBN 0201596199.

[3] JI, XiuHua, CaiMing ZHANG, JiaYe WANG a S. H. BOEY. Fast 2-D 8×8 discrete cosine transform algorithm for image coding. Science in China Series F: Information Sciences [online]. 2009, 52(2), 215-225 [cit. 2021-04-23]. ISSN 1009-2757. Dostupné z: doi:10.1007/s11432-009-0038-4

[4] CCITT. Information technology - Digital compression and coding of continuous-tone still images [online]. In: . 1992 [cit. 2021-04-28]. Dostupné z: https://www.w3.org/Graphics/JPEG/itu-t81.pdf

[5] FRIEDL, Jeffrey. An Analysis of Lightroom JPEG Export Quality Settings. Jeffrey Friedl's Blog [online]. 2010 [cit. 2021-04-12]. Dostupné z: http://regex.info/blog/lightroom-goodies/jpeg-quality

[6] TIŠNOVSKÝ, Pavel. JPEG - král rastrových grafických formátů? Root.cz - informace nejen ze světa Linuxu [online]. 2006 [cit. 2021-05-01]. ISSN 1212-8309. Dostupné z: https://www.root.cz/clanky/jpeg-kral-rastrovych-grafickych-formatu/

[7] TIŠNOVSKÝ, Pavel. Ztrátová komprese obrazových dat pomocí JPEG. Root.cz - informace nejen ze světa Linuxu [online]. 2006 [cit. 2021-04-28]. Dostupné z: https://www.root.cz/clanky/ztratova-komprese-obrazovych-dat-pomoci-jpeg/

[8] JEŽEK, David. Změní AV1 a AVIF svět? Root.cz - informace nejen ze světa Linuxu [online]. 2018 [cit. 2021-04-25]. ISSN 1212-8309. Dostupné z: https://www.root.cz/clanky/zmeni-av1-a-avif-svet/

[9] ŻÓŁCIAK, Antoni. Why WebP Is The Rockstar of Image Formats for Web Designers. In: Insane Lab [online]. 2018 [cit. 2021-04-28]. Dostupné z: https://insanelab.com/blog/web-development/webp-web-design-vs-jpeg-gif--png/

[10] Code::Blocks. Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2021 [cit. 2021-04-28]. Dostupné z: https://en.wikipedia.org/wiki/Code::Blocks

[11] ZedBoard Zynq-7000 ARM/FPGA SoC Development Board. Xilinx - Adaptable. Intelligent. [online]. Copyright © 2021 Xilinx [cit. 2021-05-09]. Dostupné z: https://www.xilinx.com/products/boards-and-kits/1-elhabt.html

[12] ZAPLETAL, Jan. Amdahlův a Gustafsonův zákon. Homel.vsb.cz [online]. 2009 [cit. 2021-05-09]. Dostupné z: https://homel.vsb.cz/ zap150/pa/ref/pa_ref.htm

[13] ARCHAMBAULT, Michael. JPEG 2000: The Better Alternative to JPEG That Never Made it Big. PetaPixel [online]. 2015 [cit. 2021-05-01]. Dostupné z: https://petapixel.com/2015/09/12/jpeg-2000-the-better-alternative-to-jpeg--that-never-made-it-big/

[14] PETERKA, Jiří. Co (ne)najdete ve slovníku: Pipelining. Computerworld [online]. 1992, 92(11) [cit. 2021-05-13]. Dostupné z: https://www.earchiv.cz/a92/a211c120.php3

[15] HASS, Calvin. JPEG Huffman Coding Tutorial. ImpulseAdventure [online]. 2009 [cit. 2021-04-28]. Dostupné z: https://www.impulseadventure.com/photo/jpeg-huffman-coding.html

[16] CCITT, INFORMATION TECHNOLOGY – DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES – REQUIREMENTS AND GUIDELINES, T.81 [online]. 1992 [cit. 2021-04-29]. Dostupné z: https://www.w3.org/Graphics/JPEG/itu-t81.pdf

[17] Xilinx Inc., Vivado Design Suite User Guide: High-Level Synthesis (ug902). [online]. 2019 [cit. 2021-04-28]. Dostupné z: https://bit.ly/3428vjl

[18] Lenna. Rice University Electrical and Computer Engineering [online]. 2003 [cit. 2021-5-14]. Dostupné z: https://www.ece.rice.edu/ wakin/images/lena512.bmp

[19] DEVCORE. 8x8 DCT (discrete cosine transformation). Wikipedia: the free encyclopedia [online]. 2012 [cit. 2021-05-01]. Dostupné z: https://upload.wikimedia.org/wikipedia/commons/2/24/DCT-8x8.png

[20] XILINX INC. ZedBoard. Xilinx - Adaptable. Intelligent. [online]. [cit. 2021-05-01]. Dostupné z: https://www.xilinx.com/content/dam/xilinx/imgs/prime/ZedBoard-obl--1000.png

Přílohy

Příloha A: Obrázky použité ke kompresi

A. 5.1: Lenna (512 x 512), zdroj: [18]

A. 5.2: Rytíř (3504 x 4960)

A. 5.3: Příroda (1920 x 1080), zdroj: www.signatureedits.com

Příloha B: Ukázka části zdrojového kódu

```
1  int push_bits(BitStream* bit_stream, uint32_t val, uint32_t n_bits, uint32_t ...
       opposite)
2  {
3      int source_index = 0;
4      int *buffer_value = &bit_stream->buffer[bit_stream->length];
5      for(; bit_stream->bit_position < 32; )
6      {
7          if(source_index ≥ n_bits)
8          {
9              break;
10         }
11         if( (!opposite && BIT_CHECK(val, 31-source_index)) || (opposite && ...
               BIT_CHECK(val, n_bits-1-source_index)) )
12         {
13             BIT_SET(*buffer_value, 31-bit_stream->bit_position);
14         }
15         else
16         {
17             BIT_CLEAR(*buffer_value, 31-bit_stream->bit_position);
18         }
19         source_index++;
20         bit_stream->bit_position++;
21         if(bit_stream->bit_position > 31)
22         {
23             (bit_stream->length)++;
24             buffer_value = &bit_stream->buffer[bit_stream->length];
25             bit_stream->bit_position = 0;
26         }
27     }
28     return 0;
29 }
```

B. 5.1: Funkce na přidávání bitů do bit streamu