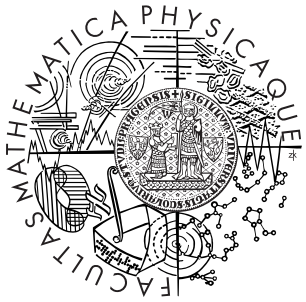
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Zbyněk Falt

Zobrazování obrázků ve formátu JPEG

Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš
Studijní program: Informatika, obor Programování

2008

Chtěl bych poděkovat Mgr. Martinu Marešovi za vedení bakalářské práce, cenné rady, podnětné připomínky a jazykové korektury k této práci.
Za korektury textu bych také velmi rád poděkoval Elišce Lacinové.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 29. května 2008 Zbyněk Falt

Obsah

1 Úvod **6**
- 1.1 Komprimace obrazových dat . . . 6
  - 1.1.1 Bezeztrátové a ztrátové metody komprimace . . . 6
- 1.2 Algoritmus JPEG . . . 7
- 1.3 Cíl práce . . . 8

2 Popis sekvenčních souborů JPEG **10**
- 2.1 Kroky prováděné při komprimaci . . . 10
  - 2.1.1 Převod a podvzorkování barvonosných složek . . . 10
  - 2.1.2 Podvzorkování barvonosných složek . . . 10
  - 2.1.3 MCU bloky . . . 11
  - 2.1.4 Diskrétní kosinová transformace . . . 12
  - 2.1.5 Kvantizace koeficientů . . . 12
  - 2.1.6 Linearizace bloků . . . 12
  - 2.1.7 Kódování hodnot koeficientů . . . 13
  - 2.1.8 Kódování posloupnosti koeficientů . . . 14
  - 2.1.9 Huffmanovo kódování . . . 14
- 2.2 Formát souborů . . . 15

3 Metody urychlující dekódování souborů JPEG **16**
- 3.1 Architektura IA-32 . . . 16
  - 3.1.1 Instrukční sada SSE2 . . . 16
- 3.2 Načtení struktury souboru . . . 17
- 3.3 Načítání dat jako proudu bitů . . . 17
  - 3.3.1 Řešení bez bytů `0xff` . . . 17
  - 3.3.2 Plnohodnotné řešení . . . 18
- 3.4 Dekódování Huffmanových kódů . . . 21
- 3.5 Interpretace symbolů . . . 22
- 3.6 Inverzní diskrétní kosinová transformace pomocí SSE2 instrukcí . . . 22
- 3.7 Transpozice matice 16 bitových hodnot pomocí SSE2 instrukcí . . . 24
  - 3.7.1 Transpozice čtyř matic 2×2 se 16-bitovými prvky . . . 25
  - 3.7.2 Transpozice dvou matic 4×4 se 16-bitovými prvky . . . 26
  - 3.7.3 Transpozice jedné matice 8×8 se 16-bitovými prvky . . . 27
- 3.8 Převod modelu YC_bC_r do modelu RGB . . . 27
- 3.9 Smíchání barevných kanálů . . . 28

4 Metody urychlující prohlížení obrázků **30**
4.1 Zobrazování obrázků s vysokým rozlišením 30
4.1.1 Dekódování výřezu obrázku 31
4.1.2 Změna měřítka obrázku 31
4.2 Optimalizace prohlížení více souborů 32
4.2.1 Výběr obrázků k přednačítání 33
4.2.2 Přednačítání obrázků 34

5 Popis přiloženého programu **35**
5.1 Popis práce s programem 35
5.1.1 Překlad a instalace 35
5.1.2 Spuštění a nastavení 35
5.1.3 Ovládání programu 36
5.2 Popis souborů se zdrojovými kódy 36
5.2.1 Soubor `bit_reader.c` 36
5.2.2 Soubor `decoder.c` 36
5.2.3 Soubor `idct.c` 36
5.2.4 Soubory `jpeg.c` a `decode.c` 38
5.2.5 Soubor `image.c` 38
5.2.6 Soubor `threaded_image.c` 38
5.2.7 Soubor `viewer.c` 39
5.2.8 Soubor `resize.c` 39
5.2.9 Soubor `my_api.c` 39

6 Závěr **40**
6.1 Porovnání rychlosti jinými programy 40
6.2 Shrnutí 40

Literatura **42**

Název práce: Zobrazování obrázků ve formátu JPEG
Autor: Zbyněk Falt
Katedra (ústav): Katedra aplikované matematiky
Vedoucí bakalářské práce: Mgr. Martin Mareš
E-mail vedoucího: mj@ucw.cz

Abstrakt: Cílem předložené práce je popsat algoritmy a techniky, které umožňují co nejrychleji zobrazovat sekvenční JPEG obrázky. Toho je dosaženo eliminací počtu aritmetických výpočtů, odstraňováním podmíněných skoků v kritických sekcích, využitím vektorových instrukcí a přednačítáním souborů v několika vláknech. Kromě použití vektorových instrukcí SSE2, které jsou podporovány pouze architekturou IA-32, jsou tyto techniky nezávislé na platformě. Přiložený program pak demonstruje implementovatelnost a účinnost těchto postupů.

Klíčová slova: JPEG, JFIF, SSE2

Title: Displaying images in JPEG format
Author: Zbyněk Falt
Department: Department of Applied Mathematics
Supervisor: Mgr. Martin Mareš
Supervisor's e-mail address: mj@ucw.cz

Abstract: The content of this work is the description of techniques and algorithms, which accelerate loading of sequential JPEG images. To reach this, special methods, such as the elimination of arithmetic computation, the reduction of conditional jumps in the most critical sections, the use of the vector instruction and the multithreaded preloading of images, are used. These techniques are platform independent, except the usa of SSE2 vector instructions, which are supported only on the IA-32 platform. The attached computer program demonstrates their real implementation and their efficiency.

Keywords: JPEG, JFIF, SSE2

Kapitola 1

Úvod

1.1 Komprimace obrazových dat

Komprimace obrazových dat v poslední době nabývá velkého významu, neboť digitální záznam obrazu nahrazuje čím dál více klasický způsob. To zapřičiňuje hlavně zvyšující se popularita digitální fotografie, digitalizace stávajících obrazových dat, rozšíření aplikací typu Google Earth[8] a dalších. Kromě počtu obrázků dochází navíc ke zvyšování jejich rozlišení a k jejich spojování do větších celků, ať už se jedná např. o panoramatické snímky nebo o satelitní fotografie zemského povrchu. Vše dohromady pak způsobuje, že se zvyšují nároky na kapacitu úložišť pro tato data. A jejich komprimace nabízí způsob jak tyto nároky snižovat.

Motivací ke komprimaci obrázků ale není jenom omezená kapacita úložišť, ale i omezená přenosová kapacita počítačových sítí. Ty sice slouží k přenosu různých druhů dat, ale podíl těch obrazových poslední dobou narůstá, neboť roste počet obrázků na webových stránkách a objevují se služby jako jednou zmíněná Google Earth. Zde již nehrají tak velkou roli nároky na kapacitu úložiště, jako spíše doba přenosu dat do počítače. Jinými slovy, šetření času uživatele je další důvod, proč obrázky komprimovat.

1.1.1 Bezeztrátové a ztrátové metody komprimace

Je zřejmé, že komprimace obrázků je poměrně důležitá a hojně používaná operace. Od doby, kdy se poprvé začala obrazová data zpracovávat elektronicky, se vyvinulo mnoho algoritmů komprese, přičemž nové metody se stále hledají. Právě z toho důvodu existuje nepřeberné množství různých komprimačních technik a algoritmů. Zatímco některé jsou specializované pouze pro konkrétní oblasti, jiné pokrývají větší množinu typů komprimovaných dat. Každopádně pro ně platí, že se všechny dají jednoznačně rozdělit do dvou hlavních skupin:

Algoritmy bezeztrátové komprimace

Do této kategorie spadají všechny algoritmy, které umí snížit objem vstupních dat, aniž by se z nich ztratila jakákoliv informace. To jednoduše znamená, že originální

obrázek a obrázek, který vznikne dekomprimací jeho zkomprimované formy, jsou bod po bodu zcela identické. Tyto metody se používají při komprimaci obrázků, u kterých by ztráta informace (např. rozmazání hran nebo zavedení šumu) byla velmi nežádoucí (např. u technických nákresů, geografických map atd.).

Mezi nejznámější formáty implementující tyto algoritmy patří např. GIF (Graphics Interchange Format) [3], který používá algoritmus LZW, nebo PNG (Portable Network Graphics) [12], který byl navržen jako náhrada patenty zatíženého GIFu a který používá komprimační algoritmus deflate [2].

Algoritmy ztrátové komprimace

Na druhou stranu existují obrázky, u kterých drobná ztráta informace není nežádoucí, a naopak je upřednostněn vysoký kompresní poměr. Mezi taková data patří například digitální fotografie. Pro jejich komprimaci se používá tato třída algoritmů, které využívají nedokonalostí lidského zraku a specifických vlastností těchto dat.

Mezi nedokonalosti lidského zraku patří například to, že oko vnímá mnohem citlivěji jasovou složku obrázku než složky barvonosné. To je způsobeno tím, že sítnice obsahuje přibližně desetkrát více tyčinek (které mají na starosti vnímat intenzitu světla) než čípků (které jsou zodpovědné za vnímaní barvy světla). Takže je možné ukládat barvonosné složky v nižším rozlišení než složku jasovou, aniž by to člověk postřehl.

Co se týče specifických vlastností obrazových dat, tak například fotografie jsou charakteristické tím, že zpravidla obsahují minimální počet ostrých barevných nebo jasových přechodů a naopak velké množství plynulých změn. Toho se dá využít tak, že je možné vysokofrekvenční část obrázku ukládat ve snížené kvalitě či úplně zanedbat. Právě této vlastnosti využívá algoritmus JPEG [6].

Díky těmto skutečnostem mohou tyto algoritmy dosahovat mnohem vyšších kompresních poměrů než metody předchozí, neboť se informace, jejichž ztrátu člověk nepostřehne nebo jejichž výskyt se na vstupu nepředpokládá, vůbec neukládají. Navíc je zpravidla možné volit kvalitu, s jakou bude obrázek zakódován, a tím ovlivnit výslednou velikost.

Na druhou stranu nejsou tyto algoritmy univerzální (hlavně díky předpokladům o charakteru dat), takže se nehodí pro komprimaci libovolných obrázků.

1.2 Algoritmus JPEG

Tento algoritmus primárně určený pro komprimaci digitálních fotografií byl navržen konsorciem JPEG (Joint Photographic Experts Group), podle něhož získal jméno. Celý postup komprimace využívá vlastností diskrétní kosinové transformace, která převádí data z časové roviny do roviny frekvenční. To způsobí, že se od sebe oddělí vysokofrekvenční část dat od nízkofrekvenční. A právě proto, že vysokofrekvenční složky jsou ve fotografiích zastoupeny velmi málo, jsou koeficienty příslušné těmto frekvencích velmi nízké. V tom spočívá ona ztráta informace, neboť tyto koeficienty se následně kvantizují, neboli celočíselně vydělí tzv. kvantizační konstantou. Hodnoty

těchto konstant zpravidla rostou směrem k vyšším frekvencím, což vyplývá z toho, že lidské oko drobné detaily příliš nevnímá, a z toho, že se výskyt jemných detailů nepředpokládá. Po tomto kroku zůstane většina hodnot vysokofrekvenčních koeficientů nulová, čehož se využívá v následné RLE (Run-Length Encoding) komprimaci doplněné Huffmanovým nebo aritmetickým kódováním [6].

Samotný standard JPEG specifikuje celkem 4 způsoby komprimace:

1. Sekvenční kódování, které obrázek zpracovává postupně zleva doprava a shora dolů.

2. Progresivní kódování, které obrázek zpracovává několika průchody, a v každém průchodu zvyšuje množství uložených informací o obrázku.

3. Bezeztrátové kódování, které zachovává věrný obraz originálu.

4. Hierarchické kódování, které ukládá obrázek sekvenčně několikrát za sebou, pokaždé s jiným rozlišením, čehož lze využít např. při změnách měřítka.

Druhé dva způsoby se příliš neujaly, neboť na bezeztrátové kódování existují jiné a rozšířenější algoritmy. Hierarchické ukládání obrázků se zase řeší tak, že se jednotlivé složky fyzicky oddělují, takže každá bude v samostatném souboru.

Nicméně první dva postupy dosáhly obrovské popularity. Sekvenční kódování má tu výhodu, že samotný kodér si vystačí s velmi malým množství pracovní paměti, čehož se využívá v digitálních fotoaparátech. Naopak progresivní kódování přináší možnost zobrazit hrubý náhled obrázku bez nutnosti načíst celý soubor. Toho se využívá převážně u webových stránek a obrázků přenášených internetem.

Formát JPEG navíc umožňuje uložit obrázky s 8 nebo 16-bitovou hloubkou na jeden kanál. Šestnáctibitové byly původně zamýšleny pro oblast např. medicíny, kde větší barevná hloubka hraje významnou roli. Nicméně takový formát se prakticky nepoužívá.

Dále může být u každého způsobu použito buď Huffmanovo nebo aritmetické kódování. Bohužel aritmetické kódování (přestože dosahuje vyšších kompresních poměrů) je zatíženo patenty, takže se v praxi používá téměř výhradně Huffmanovo kódování, na které se žádná omezení nevztahují.

1.3 Cíl práce

Cílem této práce je představit techniky, které mají za úkol co nejvíce urychlit běžné operace s obrázky uloženými v sekvenčních souborech JPEG.

Celý text je rozdělen do čtyř hlavních kapitol. Kapitola 1 čerpá ze specifikace [6] a popisuje sekvenční algoritmus JPEG v takovém rozsahu, aby to postačovalo ke snadnému pochopení dalšího obsahu. Zbylá část práce je již původní a zaměřuje se na hlavní téma. Techniky urychlující samotný proces dekódování souborů popisuje kapitola 3. V kapitole 4 jsou pak popsány algoritmy použité k urychlení běžných operací s obrázky, jako je prohlížení většího množství souborů, posun obrázku po obrazovce a změna jeho měřítka. Poslední kapitola pak stručně popisuje ovládání

a strukturu přiloženého programu, který všechny techniky představené v této práci implementuje.

Kapitola 2

Popis sekvenčních souborů JPEG

Jak již bylo naznačeno v úvodu, tato práce se zabývá dekódováním sekvenčních souborů JPEG. Právě tento formát byl vybrán proto, že téměř všechny digitální fotografie a všechny obrázky s vyšším rozlišením jsou uloženy touto metodou. Navíc zde představené techniky jsou snadno použitelné i v progresivních souborech JPEG.

K tomu, abychom mohli obrázky komprimované touto metodou správně dekódovat, je nutné znát posloupnost kroků, které vykonává kodér při ukládání obrázku. Během dekódování totiž probíhají stejné kroky pouze v opačném pořadí.

2.1 Kroky prováděné při komprimaci

2.1.1 Převod a podvzorkování barvonosných složek

První krok komprimace spočívá v tom, že se obrázek převede z 8-bitového barevného prostoru RGB do prostoru YC_bC_r, které místo tří barvonosných složek používá pouze dvě (C_bC_r) a navíc jednu složku jasovou (Y). Tento převod se řídí následujícími vztahy [14]:

$$\begin{aligned} Y &= 0{,}299\cdot R + 0{,}587\cdot G + 0{,}114\cdot B \\ C_b &= -0{,}1687\cdot R - 0{,}3313\cdot G + 0{,}5\cdot B + 128 \\ C_r &= 0{,}5\cdot R - 0{,}4187\cdot G - 0{,}0813\cdot B + 128 \end{aligned}$$

Barevný prostor YC_bC_r je oproti RGB výhodný právě proto, že je blízký tomu, jak člověk vnímá světlo, neboť téměř ekvivalentním způsobem odděluje barvonosné složky od složky jasové. Následně je pak možné využít toho, že lidské oko citlivěji vnímá intenzitu světla než jeho barvu, takže zatímco jasová složka je uložena v plném rozlišení, barvonosné se zpravidla podvzorkují a dále se s nimi pracuje už jenom ve sníženém rozlišení.

2.1.2 Podvzorkování barvonosných složek

Před podvzorkováním je zvolen tzv. vzorkovací faktor, který udává, v jakém poměru je v jednotlivých směrech rozlišení sníženo. Vzorkovací faktor jsou tři dvojice čísel,

přičemž každá dvojice je určena pro jednu složku v pořadí Y, C_b a C_r. První číslo z dvojice vždy popisuje horizontální směr a druhé číslo vertikální. Např. vzorkovací faktor 211111 říká, že v horizontálním směru připadají dva vzorky jasové složky na jeden vzorek barvonosných složek. Ve vertikálním směru se žádné podvzorkování neprovádí, neboť na jeden vzorek jasové složky připadá jeden vzorek barvonosný.

Ač specifikace JPEG podporuje vzorkovací faktor téměř libovolný, vyskytují se v praxi pouze následující 4 možnosti:

Faktor	**Popis**
111111	Nedochází k podvzorkování barvonosných složek.
221111	Rozlišení barvonosných složek je v obou směrech sníženo dvakrát.
211111	Rozlišení barvonosných složek je v horizontálním směru sníženo dvakrát.
121111	Rozlišení barvonosných složek je sníženo dvakrát ve vertikálním směru.

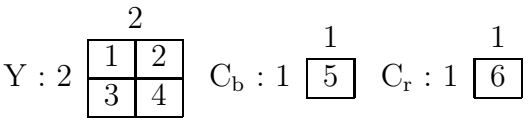
2.1.3 MCU bloky

V dalším kroku je obrázek dlaždicovitě rozdělen na tzv. MCU bloky (Minimum coded unit). MCU blok je jednotka, která pokrývá určitý výřez obrázku. Platí, že v jednom MCU bloku jsou zastoupeny všechny složky celočíselným počtem bloků o rozměru 8×8 bodů a že tento počet je minimální možný. Takže např. pro vzorkovací faktor 111111 je MCU blok tvořen třemi bloky 8×8 bodů – každá složka má právě jeden blok. Pro faktor 221111 ale jeden MCU blok obsahuje 6 bloků 8×8 bodů. Čtyři pro jasovou složku a po jednom pro barvonosné složky.

Pokud obrázek nemá rozměry, které by byly násobkem rozměru MCU bloku, je jeho velikost dorovnána na nejbližší vyšší násobek v obou směrech. Tyto body navíc jsou při dekomprimaci ořízntuty, takže se součástí obrázku nestanou.

Nyní se MCU bloky sekvenčně zpracovávají zleva doprava a shora dolů tak, že se každý MCU blok serializuje, neboli převede na posloupnost bloků 8×8 bodů. Pořadí serializovaných bloků je opět zleva doprava a shora dolů. Navíc bloky náležející jasové složce předcházejí barvonosným a složka C_b předchází složce C_r.

Například pro vzorkovací faktor 221111 je MCU blok serializovaný tak, jak je naznačeno na obrázku 2.1.



Obrázek 2.1: Serializace MCU bloku

Nyní se již data mohou dále zpracovávat jako proud navzájem téměř nezávislých bloků 8×8 čísel.

2.1.4 Diskrétní kosinová transformace

Před dalším zpracováním se vstupní blok 8×8 bodů upraví tak, že se od každého vzorku odečte konstanta 2^{p-1}, kde p je počet bitů použitých na uchování jeho hodnoty. Tedy pro 8-bitové vzorky se odečte číslo 128 a tím se hodnoty vzorků zobrazí do intervalu $\langle -128, 127 \rangle$.

Poté se na každý blok provede dvoudimenzionální diskrétní kosinové transformace dle vztahu:

$$S_{uv} = \frac{1}{4} C(u) \cdot C(v) \cdot \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} \cdot \cos \frac{(2x+1)u\pi}{16} \cdot \cos \frac{(2y+1)v\pi}{16},$$

kde

$$C(x) = \begin{cases} \frac{1}{\sqrt{2}} & \text{pro } x = 0 \\ 1 & \text{pro } x \neq 0. \end{cases}$$

Tato transformace je sama o sobě bezeztrátová, nicméně vlivem zaokrouhlování a nepřesné aritmetiky v počítači již nelze s jistotou původní blok zcela obnovit. Transformace je důležitá proto, že převede vzorky z časové roviny do roviny frekvenční. Nízké frekvence jsou soustředěny v levém horním rohu a rostou směrem k pravému dolnímu rohu.

Koeficient vlevo nahoře se nazývá stejnosměrný koeficient (DC-koeficient), neboť mu odpovídá v obou směrech nulová frekvence, a všechny ostatní se nazývají střídavé koeficienty (AC-koeficienty).

A protože obrázky, pro které je formát JPEG určen, jsou charakteristické plynulými přechody, jsou největší hodnoty soustředěny právě v oblasti nízkých frekvencí.

2.1.5 Kvantizace koeficientů

Následně se hodnoty koeficientů kvantizují. To je proces, při kterém se koeficienty celočíselně vydělí příslušnou kvantizační konstantou. Tyto konstanty jsou voleny tak, aby hodnoty nízkofrekvenčních koeficientů byly co nejvíce zachovány, zatímco vysokofrekvenční koeficienty jsou kvantizovány nejvíce. Blok 8×8 kvantizačních konstant (jedna konstanta pro jeden koeficient) se nazývá kvantizační tabulka a tyto tabulky jsou pro každý obrázek dvě různé. Jedna pro jasovou a jedna společná pro barvonosné složky.

Po kvantizaci je zpravidla většina hodnot koeficientů (převážně vysokofrekvenčních) nulová, čehož se využívá během kódování koeficientů.

2.1.6 Linearizace bloků

Předpřípravou samotného kódování ale ještě je převedení koeficientů na tzv. cik-cak sekvenci. Ta převádí dvojrozměrnou tabulku 8×8 na jednorozměrnou posloupnost 64 hodnot tak, že je vypisuje směrem od nejnižších frekvencí. Prvním koeficientem v posloupnosti je tedy DC-koeficient.

Tabulka 2.1 ukazuje pro každý koeficient, na jaké pozici (číslováno od nuly) se ve výsledné posloupnosti bude vyskytovat, a snad i ozřejmí, proč se sekvence nazývá cik-cak.

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

Tabulka 2.1: Převod na cik-cak sekvenci

Toto pořadí je zvoleno právě proto, že se ve výsledné posloupnosti budou nacházet vysokofrekvenční koeficienty vedle sebe, takže je pravděpodobné, že se ve výsledku budou nacházet dlouhé řetězce nul.

2.1.7 Kódování hodnot koeficientů

Posledním krokem, který se provede před samotným zakódováním hodnot koeficientů, je odečtení předchozí hodnoty DC-koeficientu odpovídající složky od aktuální hodnoty. Důvodem této operace je předpoklad, že v obrázku jsou plynulé změny barev, takže rozdíl dvou po sobě jdoucích DC-koeficientů může být relativně nízký. Ve výsledku to znamená, že pro uložení jeho hodnoty bude potřeba zpravidla méně bitů.

Nyní k samotnému kódování hodnot koeficientů. Ty se totiž neukládají přímo v binární soustavě. Sice platí, že čísla s menší absolutní hodnotou si vystačí s nižším počtem bitů než čísla větší. Nicméně takový postup by přinesl zbytečnou redundanci. Např. pomocí 3 bitů je možné vyjádřit čísla v intervalu $\langle -4, 3\rangle$. Pomocí 4 bitů pak čísla v intervalu $\langle -8, 7\rangle$. Sice jsme přidali jeden bit, ale získali jsme možnost uložit pouze čísla v intervalech $\langle -8, -5\rangle$ a $\langle 4, 7\rangle$ (čísla v intervalu $\langle -4, 3\rangle$ bychom vyjádřili pomocí tří bitů). Jinými slovy, použití 4 bitů přineslo možnost kódovat pouze 8 nových čísel i přesto, že 4 bity mohou nabývat 16 různých stavů.

Tato redundance je odstraněna zavedením tzv. kategorií, přičemž každá kategorie udává, jaké hodnoty lze zakódovat pomocí daného počtu bitů. Důležité je to, že kategorie pro p bitů obsahuje vždy 2^p hodnot, takže nedochází ke zbytečné redundanci. Rozsah hodnot pro každou kategorii shrnuje tabulka 2.2.

Ve skutečnosti platí, že žádný z koeficientů nebude větší než 2^{10}. Kategorie pro 11 bitů je přidána proto, že se ukládá rozdíl DC-koeficientů, který může v nejhorším případě být až dvojnásobkem jejich maximální hodnoty.

Počet bitů	Příslušné hodnoty
0	$\{0\}$
1	$\{-1\} \cup \{1\}$
2	$\langle -3, -2\rangle \cup \langle 2, 3\rangle$
3	$\langle -7, -4\rangle \cup \langle 4, 7\rangle$
4	$\langle -15, -8\rangle \cup \langle 8, 15\rangle$
5	$\langle -31, -16\rangle \cup \langle 16, 31\rangle$
6	$\langle -63, -32\rangle \cup \langle 32, 63\rangle$
7	$\langle -127, -64\rangle \cup \langle 64, 127\rangle$
8	$\langle -255, -128\rangle \cup \langle 128, 255\rangle$
9	$\langle -511, -256\rangle \cup \langle 256, 511\rangle$
10	$\langle -1023, -512\rangle \cup \langle 512, 1023\rangle$
11	$\langle -2047, -1024\rangle \cup \langle 1024, 2047\rangle$

Tabulka 2.2: Kategorie pro kódování hodnot koeficientů

2.1.8 Kódování posloupnosti koeficientů

Protože zakódovaná čísla mohou mít rozdílnou délku, je nutné ukládat i jejich délku. To řeší tato fáze algoritmu a navíc konečně využívá toho, že se v posloupnosti mohou vyskytnout dlouhé řetězce nulových hodnot.

DC-koeficient je vždy uložen tak, že samotnému kódu předchází symbol udávající číslo, které je rovno počtu bitů nutných k jeho uložení bez ohledu na to, zda je nulový nebo ne. Každý AC-koeficient je pak uvozen symbolem, který se skládá z dvojice čísel. První číslo udává, kolik AC-koeficientů je před ním nulových, zatímco druhé říká, kolik bitů je třeba na uložení jeho hodnoty.

Takže např. posloupnost:

$$-10, 1, 2, 0, 3, 0, 0, 4 \ldots$$

bude zakódována takto:

$$(4)(-10)(0,1)(1)(0,2)(2)(1,2)(3)(2,3)(4)\ldots$$

Na vyjádření počtu bitů vždy stačí 4 bity. Na vyjádření počtu nulových koeficientů by teoreticky bylo potřeba až 6 bitů. Ve skutečnosti se ale používají rovněž 4 bity a větší počet nulových koeficientů než 15 je řešen následovně: Pokud jsou všechny zbývající koeficienty nulové, je zapsán speciální symbol (0,0), pokud tato situace nenastala a počet nulových koeficientů je větší než patnáct, zapíše se symbol (15,0), který znamená 16 nulových koeficientů, a dále se pokračuje v kódování stejným způsobem.

2.1.9 Huffmanovo kódování

Jednotlivým symbolům jsou v tomto kroku přiřazeny Huffmanovy kódy vygenerované klasickým algoritmem podle četnosti jejich výskytů ve výsledném souboru.

Existují celkem 4 Huffmanovy tabulky. Zvlášť pro délky DC-koeficientů, zvlášť pro symboly kódující AC-koeficienty a zvlášť pro jasovou a barvonosné složky.

Protože spočítat četnost všech symbolů a následně vygenerovat k nim příslušné kódy může být poměrně náročné, používají se, hlavně u malých zařízení, již předpočítané tabulky.

Tímto krokem jsme konečně získali ze vstupního obrázku zakódovanou posloupnost bitů, nyní zbývá ji jenom uložit do souboru.

2.2 Formát souborů

Způsob, jakým jsou data v souboru uložena, udává standard JIF (JPEG Interchange Format). Tento standard je ale velmi obecný a ponechává mnoho věcí nevyřešených. Například neříká nic o tom, v jakém barevném modelu mají být obrázky uloženy nebo jaký je vzájemný vztah jednotlivých barevných složek. Je to spíše zárodek pro další standardy.

Z toho zárodku vychází standard JFIF (JPEG File Interchange Format), který zmíněné nedostatky řeší. Přesně vymezuje, jakým způsobem a v jakém vzájemném vztahu jsou data uložena. Obrázky s příponou .jpg, respektive .jpeg vycházejí právě z tohoto standardu.

Nebudeme zabíhat příliš do podrobností, neboť samotný formát dat v souboru již není pro další text příliš důležitý. Nicméně je nutné zmínit alespoň základní strukturu souboru.

Soubor je tvořen tzv. bloky, přičemž každý (kromě toho, ve kterém jsou uložena zakódovaná data) je uvozen hlavičkou obsahující typ bloku a jeho délku. V těchto blocích jsou uloženy informace nutné k dekódování souboru – rozlišení obrázku, vzorkovací faktor, kvantizační tabulky, definice Huffmanových kódů, EXIF (Exchangeable image file format) a další. Každá hlavička vždy začíná bytem `0xff` a teprve druhý byte udává její typ.

Bohužel se může stát, že v zakódovaných datech bude byte `0xff`, který jakoby uvozuje neexistující blok, a dekodér by jej mohl chybně interpretovat. Aby se tomuto efektu předešlo, předepisuje standard, aby za každý takový byte byl vložen byte `0x00`, který signalizuje, že se nejedná o začátek bloku. Tento přidaný byte pak musí dekodér ignorovat, což přináší nemalé rychlostní problémy.

Kapitola 3

Metody urychlující dekódování souborů JPEG

Nyní víme, jak probíhá proces, kterým z obrázku vytvoříme soubor obsahující zkomprimovaná data. Můžeme se tedy zaměřit na hlavní cíl této práce – dekódování těchto souborů.

Mohlo by se zdát, že tato kapitola bude identická s kapitolou předchozí, jenom v obráceném pořadí. Cílem však nebude popis algoritmu na dekódování, neboť ten je velmi podrobně popsaný ve standardu JPEG [6], nýbrž popis technik použitých k tomu, aby tento proces byl co nejrychlejší.

3.1 Architektura IA-32

Techniky, kromě použití vektorových instrukcí SSE2, popsané v této kapitole jsou platformově nezávislé. Nicméně vzhledem k tomu, že práce s obrázky převažuje u uživatelů domácích počítačů, a k tomu, že v této oblasti převažuje architektura IA-32 (Intel Architecture, 32 bit), jsou tyto techniky optimalizovány právě pro ni.

Z toho plyne mimo jiné to, že pokud bylo možné vyřešit daný problém více způsoby, byl vybrán takový, který co nejvíce respektoval přímo doporučení dvou největších výrobců těchto procesorů [1] a [5].

Některé operace při dekódování souborů je možné vektorizovat, a tím urychlit jejich vykonávání. Bohužel neexistuje možnost, jak nezávisle na platformě vykonávat vektorové výpočty. Z důvodů popsaných výše byla vybrána možnost provádět tyto výpočty pomocí vektorových instrukcí SSE2, které jsou podporovány všemi novějšími procesory architektury IA-32. Pro ostatní platformy je v programu určen alternativní kód, který žádnou vektorizaci nepoužívá.

3.1.1 Instrukční sada SSE2

Instrukční sada SSE2 nabízí možnost používat vektorové 128-bitové registry nazvané `xmm0` až `xmm7`. Programátor si může vybrat na kolik navzájem nezávislých částí tyto registry rozdělí. Každý registr tak může představovat buď jedno 128-bitové číslo, dvě 64-bitová, čtyři 32-bitová, osm 16-bitových nebo šestnáct 8-bitových.

Instrukční sada SSE2 umožňuje vykonávat vektorové výpočty nad těmito registry. Navíc nabízí instrukce, které umožňují rozličným způsobem přeskládat nebo kombinovat obsahy těchto registrů, čehož lze při řešení některých situací s výhodou využít.

Pozn.: Veškeré ukázky kódu pomocí SSE2 instrukcí budou zapisovány tak, že jednotlivé instrukce budou chápány jako funkce, které mají vstupní parametry a návratovou hodnotu, přičemž vstupní parametry ponechají beze změny. Tento způsob je zvolen s ohledem na názornost ukázek. Je ale velmi snadné převést tento zápis do korektního kódu pro IA-32.

Například zápis pro vektorový součet dvou registrů:

```
xmm2 = paddw(xmm0, xmm1);
```

Lze do asembleru přepsat takto:

```
movdqa xmm2, xmm1
paddw  xmm2, xmm0
```

3.2 Načtení struktury souboru

První operací, kterou je nutné vykonat, je načtení jednotlivých bloků souboru a jejich interpretace. Vzhledem k tomu, že samotná hlavička souboru má v porovnání se zbytkem souboru zanedbatelnou velikost a že její načtení se ve výsledném čase prakticky neprojeví, stačí se v této části držet postupů specifikovaných a JPEG specifikaci [6] a dále se jimi nezabývat.

V dalším textu tedy již budeme předpokládat, že je hlavička souboru korektně načtena, takže lze začít se samotným dekódováním datové části.

3.3 Načítání dat jako proudu bitů

První problém, na který narazíme a který je třeba vyřešit, je způsob, jakým načítat zakódovaná data jako proud bitů. Z definice formátu plyne, že posloupnost bitů, která vystupuje z kodéru je kódována jako posloupnost bytů, kde za každým bytem s hodnotou `0xff` musí následovat byte `0x00`, který ale není součástí dat a který se má ignorovat (viz kapitola 2.2). Tato zdánlivě jednoduchá operace je touto skutečností značně zkomplikovaná, neboť samotné čtení dat patří mezi časově nejkritičtější operace.

3.3.1 Řešení bez bytů `0xff`

Pro začátek předpokládejme, že se ve vstupní posloupnosti žádné byty `0xff` nevyskytují. Pak lze problém řešit vytvořením okénka po posloupnosti bitů a vždy, kdy by bylo potřeba n bitů, by byly k dispozici na začátku okénka. Po jejich zpracování pak stačí okénko posunout o n bitů dále. Toto okénko bude v dalším textu označováno jako bitové.

Nabízí se otázka, jak velké by takové okénko mělo být. Určitě by mělo mít alespoň tolik bitů, kolik nejvíce budeme v libovolném kroku potřebovat. Využijeme znalosti, že žádný z Huffmanových kódů nemůže být ze specifikace delší než 16 bitů. Navíc žádná hodnota nemůže být zakódovaná větším počtem bitů než 13. Takže je zřejmé, že velikost 16 bitů by měla být dostačující. Na druhou stranu je nutné si uvědomit, že operace posunu okénka bude zřejmě časově náročná, takže je určitě výhodné volat ji co nejméněkrát. Proto se nabízí mít bitové okénko velké tak, aby bylo možné z něj bylo čerpat bity rovnou na dva kroky – načtení Huffmanova kódu a načtení zakódovaného koeficientu. Proto zvolíme velikost na 32 bitů, což navíc přináší podstatnou výhodu v tom, že architektura IA-32 je optimalizovaná pro práci s proměnnými právě této velikosti.

Jak by vypadala samotná implementace posunu bitového okénka? Ve skutečnosti by se udržovala okénka dvě. Již zmíněné bitové okénko, které by se umělo posunovat po jednotlivých bitech, a zásobníkové, které by sloužilo jako určitý zdroj bitů, kterými by se doplňovalo bitové okénko po jeho posunutí. Toto okénko by mělo velikost rovněž 32 bitů a posunovalo by se po násobcích 4 bytů, což se implementuje velmi snadno, a navíc je možné udržovat čtení z paměti zarovnané na adresy, které jsou násobky 4. Čtení ze zarovnaných adres procesory vykonávají rychleji než čtení nezarovnaná.

Při požadavku na posun okénka by se okénko posunulo o počet požadovaných bitů vlevo a zprava by se za pomocí bitových operací vzniklá mezera zaplnila bity ze zásobníkového okénka. Kdyby v zásobníkovém okénku nebyl dostatečný počet bitů, posunulo by se toto okénko na další 4 byty a operace doplnění bitů by se zopakovala. Tento postup má tu výhodu, že k této operaci není potřeba žádných cyklů, neboť se stačí na začátku rozhodnout, jestli je v zásobníkovém okénku dostatečný počet bitů, a podle toho postupovat.

K implementaci si stačí pamatovat počet bitů použitých z bitového okénka, počet nepoužitých bitů ze zásobníkového okénka a pozici zásobníkového okénka v souboru.

3.3.2 Plnohodnotné řešení

Jak nyní postupovat, když odstraníme předpoklad o tom, že se ve vstupní posloupnosti nevyskytují byty `0xff`? Nabízí se možnost nastavit velikost zásobníkového okénka na 8 bitů a vždy, když bychom do něj načetli byte `0xff`, posunuli bychom ukazatel příště nikoliv o jeden, nýbrž o dva byty dále. Tím by se nulový byte následující za bytem `0xff` jednoduše přeskočil.

Tato metoda ale má dvě nevýhody:

1. Čtení z paměti po jednotlivých bytech je poměrně pomalé. Pomalejší než čtení po 32-bitových částech.

2. V jednom kroku by bylo do bitového okénka přidáno maximálně 8 bitů, takže by se operace posunu okénka neobešla bez cyklu, který by dále zpomalil kód.

Proto by bylo vhodné ponechat velikost zásobníkového okénka na 32 bitech, aby výhody předchozího řešení zůstaly zachované. Jedno z řešení je mít okénko široké 32 bitů (4 byty) a v případě, že žádný z bytů není `0xff`, pracovat s ním jako

v předešlém řešení. V okamžiku, kdy by některý z bytů měl způsobit problém, vyvolala by se zvláštní procedura, která by problém vyřešila, a program by pak mohl pokračovat dále. Naštěstí platí, že výskyt bytu `0xff` je nepravděpodobný, takže obslužná procedura nebude volána příliš často.

Zbývá vyřešit dvě otázky:

1. Jak co nejrychleji detekovat byte `0xff` v zásobníkovém okénku?
2. Jak ošetřit jeho výskyt?

Detekce bytu `0xff` ve 32-bitové proměnné

Je zřejmé, že tato operace by měla probíhat co nejrychleji, neboť se bude provádět pro každou čtveřici přečtených bytů. Proto by bylo vhodné rozhodnout o výskytu nezávisle na velikosti proměnné pouze za pomoci elementárních aritmetických a bitových operacích.

Platí, že přičtení jedničky k binárnímu číslu nastaví nejnižší nenulový bit na jedničku a všechny nižší bity nastaví na nulu. To mimo jiné znamená, že když přičteme k bytu `0xff` jedničku, tak se jeho nejvyšší bit změní z jedničky na nulu. Pokud jedničku přičteme k jakémukoliv jinému bytu, tak tento jev nenastane (jediná hodnota bytu, která ještě změní přičtením jedničky jeho nejvyšší bit je `0x7f`, kdy se změní z nuly na jedničku).

Stačí tedy přičíst ke zkoumanému bytu jedničku a podívat se, jestli se nejvyšší bit změnil z jedničky na nulu. Tato metoda se dá aplikovat pro všechny 4 byty najednou.

Takže výraz[1]

$$x \;\&\; \texttt{0x80808080} \;\&\; \texttt{\~{}}(x + \texttt{0x01010101})$$

je nenulový, pokud má v 32-bitové proměnné x alespoň jeden byte hodnotu `0xff`.

Je pravda, že se při operaci sčítání mohou jednotlivé byty ovlivňovat, což by zdánlivě mohlo ovlivnit výsledek operace. To ale může nastat pouze tehdy, pokud by se v proměnné vyskytoval byte, který by obsahoval samé jedničky, a tedy měl hodnotu `0xff`. Jenže takový byte potřebujeme detekovat takže to, že ovlivní své okolí, již nevadí.

Vyhodnocování výrazu lze ale ještě urychlit. Z definice dvojkového doplňku platí, že $-y = \texttt{\~{}}y + 1$, neboli $\texttt{\~{}}y = -y - 1$.

Takže část $\texttt{\~{}}(x + \texttt{0x01010101})$ lze přepsat takto: $-x - \texttt{0x01010101} - 1$

Protože ale platí $\texttt{\~{}}\texttt{0x01010101} = -\texttt{0x01010101} - 1$, lze původní výraz zapsat jako:

$$x \;\&\; \texttt{0x80808080} \;\&\; (\texttt{\~{}}\texttt{0x01010101} - x)$$

Takto jsme ještě ušetřili instrukci bitové negace při vyhodnocování.

[1] Výrazy jsou zapsány v konvenci jazyka C.

Ošetření výskytu bytu `0xff` v zásobníkovém okénku

Když zjistíme, že se byte `0xff` vyskytuje v zásobníkovém okénku, zavoláme obslužnou proceduru, která tento problém vyřeší. Procedura by měla odstranit z okénka byty `0x00`, které následují po bytu `0xff`, což provede jednoduše tak, že nulové byty přeskočí, chybějící byty doplní zleva nulami a bokem si zapamatuje, že se ze zásobníkového okénka již použilo vynechaný počet bitů.

Velký problém ovšem nastává v okamžiku, kdy je byte `0xff` posledním bytem okénka a nulový byte je prvním bytem okénka následujícího. Pak nelze tuto operaci jednoduše provést, neboť v dalším kroku by se do bitového proudu dostalo navíc 8 nulových bitů, které tam správně být neměly.

Sice se nabízí řešení, že v tomto případě by se okénko v příštím kroku posunulo pouze o 3 byty a to současné by se o byte zkrátilo, jenže tím by se narušila významná výhoda toho, že všechna čtení z paměti jsou korektně zarovnána.

Proto nezbývá nic jiného, než udělat zásobníkové okénko dvojité. První okénko bude to, o kterém jsme mluvili dosud a druhé okénko bude vždy o krok před prvním. Pomocí tohoto triku bude možné ošetřit vzniklou situaci tak, že první okénko zkrátíme o poslední byte a do prvního bytu druhého okénka vložíme místo nuly číslo `0xff`. Tím jsme sice vyřešili problém, ale když provedeme posun okénka, tak prvním bytem bude `0xff`, a je nutné si pamatovat, že druhý byte, ačkoliv následuje po `0xff`, se nesmí vynechávat.

Např. tuto situaci, která by nastala těsně po posunu zásobníkového okénka:

První okénko				Druhé okénko				Korektní	Použito
`0xff`	`0x00`	`0xff`	`0x00`	`0xff`	`0x00`	`0xff`	`0x00`	ano	32 bitů

procedura řeší takto:

První okénko				Druhé okénko				Korektní	Použito
`0x00`	`0x00`	`0xff`	`0xff`	`0xff`	`0x00`	`0xff`	`0x00`	ano	16 bitů

Zatímco následující situaci, ve které dochází právě k tomu, že byte `0xff` je posledním bytem prvního zásobníkového okénka:

První okénko				Druhé okénko				Korektní	Použito
`0x11`	`0xff`	`0x00`	`0xff`	`0x00`	`0xff`	`0x00`	`0xff`	ano	32 bitů

vyřeší procedura takto:

První okénko				Druhé okénko				Korektní	Použito
`0x00`	`0x00`	`0x11`	`0xff`	`0xff`	`0xff`	`0x00`	`0xff`	ne	16 bitů

Položka „Korektní“ říká, zda se v druhém okénku jedná o korektní stav či nikoliv. Nekorektním stavem přitom rozumíme stav takový, že ač je na první pozici byte `0xff`, nemá se druhý byte vynechat.

Je zřejmé, že implementace tohoto postupu nemůže být zcela přímočará, neboť by znamenala značný nárůst počtu podmíněných příkazů, kterým je nutné se co nejvíce vyhýbat. V přiloženém programu je proto tento postup implementován tak, že většina z podmíněných příkazů je řešena pouze pomocí aritmetických a bitových operací.

3.4 Dekódování Huffmanových kódů

Nejprostší možnost, jak dekódovat symboly, je vytvořit si prefixový strom nad abecedou $\{0, 1\}$ a jeho listy ohodnotit příslušnými symboly. Tento přístup má zřejmou nevýhodu v tom, že každý zpracovávaný bit znamená přístup do paměti, ze které je třeba načíst, kde je uložen příslušný potomek současného uzlu. Zbytečným přístupům do paměti bychom se ale chtěli co nejvíce vyhnout, neboť zpomalují běh programu.

Problém s počtem přístupů do paměti lze jednoduše řešit zvětšením abecedy stromu a nezpracovávat jednotlivé bity, ale rovnou celé jejich skupiny.

S tímto postupem se pojí dvě otázky:

1. Jak velké skupiny bitů najednou zpracovávat?
2. Jak vyřešit situaci, kdy velikost Huffmanova kódu není násobkem?velikosti skupiny bitů

Odpověď na první otázku je trochu komplikovanější než na druhou, tak začněme s ní.

Definice formátu zaručuje, že délka nejdelšího kódu je 16 bitů, takže nemá smysl, aby velikost skupiny byla větší než 16 bitů. Ze stejného důvodu též platí, aby velikost skupiny dělila číslo 16.

Dále by celá struktura měla mít relativně malou paměťovou náročnost. Hlavní motivací k tomuto požadavku je to, že se s ní bude intenzivně pracovat, takže by bylo vhodné, kdyby si ji mohl procesor celou uložit do své vyrovnávací paměti. Z té by pak k ní měl mnohem rychlejší přístup.

A nakonec by velikost skupin bitů neměla být příliš malá, neboť nejen, že by se počet přístupů do paměti příliš nesnížil, ale navíc by jinak bylo nutné v každém okamžiku testovat, zda jsme již nebylo dosaženo listu v Huffmanově stromě. To by ale znamenalo mnoho vykonávaných podmíněných příkazů ve velmi kritické části programu.

Jako jediná rozumná velikost skupin tak zbývá pouze 8 nebo 16 bitů. Jenže pro 16 bitů by první (a zároveň jediný) uzel měl 2^{16} položek, přičemž každá položka musí obsahovat alespoň jeden byte, neboť se kódují symboly o velikosti 8 bitů. Z toho vyplývá, že velikost struktury by byla nejméně 64 kB. To je bohužel pro současné procesory, kde vyrovnávací paměť první úrovně má velikost řádově desítky kilobytů, příliš mnoho. Navíc je třeba vzít v úvahu, že existují 4 různé kódové tabulky.

Takže zbývá 8 bitů. Tato velikost je prakticky ideální, neboť strom bude mít maximálně dvě úrovně, přičemž s větší pravděpodobností se do druhé úrovně nebude třeba ani podívat. Navíc velikost celé struktury je poměrně malá. Jeden uzel má alespoň 256 záznamů. A vzhledem k tomu, že delší kódy mají stejný prefix, tak uzlů ve druhé úrovni příliš mnoho nebude. Průměrně do 10 uzlů, takže celá struktura obsahuje méně než $11 \cdot 256$ položek. Navíc k načtení zakódovaného symbolu stačí pouhý jeden podmíněný příkaz.

Odpověď na druhou otázku je již jednoduchá. Stačí si kromě symbolu, který je reprezentovaný příslušným kódem pamatovat, kolik bitů daný kód zabírá. Tak snadno zjistíme, jak velký počet bitů musíme odebrat z bitového okénka.

3.5 Interpretace symbolů

Interpretace symbolů je v okamžiku, kdy získáme jejich posloupnost dekódováním Huffmannových kódů, takřka triviální. Stačí vyjít pouze z jejich definice (viz kapitola 2.1.8). Drobná potíž však nastává u dekódování hodnot koeficientů.

Jak jsme ukázali v kapitole 2.1.7, koeficienty nejsou zakódovány v binární soustavě, ale pomocí tzv. kategorií. Binární hodnota kódu, který přečteme, představuje pořadí (číslováno od nuly) skutečné hodnoty v příslušné kategorii. Např. 3-bitový kód s hodnotou 2, představuje číslo 3. číslo v kategorii pro 3 bity. Kategorie pro 3 bity obsahuje hodnoty $-7, -6, -5, -4, 4, 5, 6, 7$, takže skutečná hodnota koeficientu bude -5. Mohli bychom se držet doporučeného postupu, který se nachází ve specifikaci JPEG, ale to by znamenalo použít opět podmíněný příkaz v časově velmi kritické části.

Stačí si ale všimnout toho, že kódy, které mají nastavený nejvyšší bit, odpovídají číselně přímo hodnotě koeficientu (např. 3-bitový kód s hodnotou 5 představuje koeficient s hodnotou 5). Pokud nejvyšší bit nastavený není, pak výsledná hodnota je tvořena součtem hodnoty kódu a nejnižším číslem v příslušné kategorii. Hodnota nejnižšího čísla se snadno spočte jako $1 - (1 << p)$, kde p je počet bitů příslušné kategorie.

Takže pokud by byla v proměnné x jednička tehdy, když je nejvyšší bit zakódovaného čísla nulový, a nula v opačném případě, pak by hodnota koeficientu byla rovna:

$$koeficient = kod + x - (x << delka\_kodu)$$

Zbývá vyřešit, jak nastavit proměnnou x. To ale naštěstí není složité, neboť nejvyšší bit kódu je 32. bitem v bitovém okénku (viz kapitola 3.3), takže stačí bitové okénko bitově znegovat a posunout o 31 bitů doprava. Takže platí:

$$x = (\tilde{}\, bitove\_okenko) >> 31$$

3.6 Inverzní diskrétní kosinová transformace pomocí SSE2 instrukcí

Nyní umíme načítat jednotlivé MCU bloky. Další krok, který přijde na řadu je provedení inverzní kosinové transformace na každý blok 8x8 koeficientů v načteném MCU bloku.

Výpočet inverzní kosinové transformace je dán tímto vztahem:

$$S_{uv} = \frac{1}{4} \sum_{x=0}^{7} \sum_{y=0}^{7} C(u) \cdot C(v) \cdot s_{yx} \cos \frac{(2x+1)u\pi}{16} \cdot \cos \frac{(2y+1)v\pi}{16}$$

kde

$$C(x) = \begin{cases} \frac{1}{\sqrt{2}} & \text{pro } x = 0 \\ 1 & \text{pro } x \neq 0. \end{cases}$$

Výpočet této transformace přímo z definice je ale časově velmi náročný. Pro transformaci bloku $N \times N$ je totiž zapotřebí N^4 operací, neboť výpočet jednoho členu trvá N^2 kroků.

Lze však využít skutečnosti, že dvojrozměrnou transformaci je možné spočítat tak, že se provede jednorozměrná transformace nejdříve na řádky (řádky odpovídají indexu u v uvedeném vztahu) a poté na sloupce (index v). Tento trik zredukuje počet kroků na $2\dot{N}^3$, neboť výpočet jednoho řádku trvá N^2 kroků. Řádků je dohromady N a totéž se provede i pro sloupce, takže výsledný počet operací je dvojnásobný.

Naštěstí existují rychlé algoritmy na výpočet jednorozměrné inverzní transformace, pomocí kterých lze požadovaný počet kroků dále snížit. Tyto algoritmy zpravidla vycházejí z metod pro rychlý výpočet Fourierovy transformace [11], ze které je kosinová transformace odvozena.

Právě tento postup je zvolen v knihovně The Independent JPEG Group's JPEG Library [13] a výpočet transformace v přiloženém programu vychází z algoritmu použitého v této knihovně. Jenom se jej snaží implementovat pomocí SSE2 instrukcí, díky kterým se výpočet znatelně urychlí.

Lze využít toho, že hodnoty koeficientů se vždy vejdou do 11 bitů, a tak je možné rozdělit 128-bitové XMM registry na osm 16-bitových.

Pak už je snadné provést jednorozměrnou transformaci na všechny sloupečky najednou. To lze udělat tak, že budeme provádět stejné operace jako při výpočtu transformace jednoho sloupečku, jenom všechny operace vykonáme vektorově na všech 8 hodnot najednou.

Pro sloupečky bloku 8×8 je tedy transformace snadná, ale pro jeho řádky nastane problém. SSE2 instrukce jsou totiž pouze vertikální, tzn. že lze např. sčítat pouze 1. složku vektoru s 1. složkou jiného vektoru. Jenže pro výpočet transformace pro řádky bychom potřebovali instrukce horizontální, neboli sčítat např. 1. složku vektoru s 3. složkou téhož vektoru. Nezbývá tedy nic jiného než celý blok transponovat, čímž se z řádků stanou sloupečky, na ně provést opět stejnou transformaci jako na sloupečky, pak provést transpozici ještě jednou, a tím vrátit blok do původního stavu.

Naštěstí nezáleží na tom, zda se nejdříve provede transformace na sloupečky a teprve poté na řádky, nebo zda to bude naopak. Díky tomu můžeme ušetřit jednu transpozici, neboť jednotlivé bloky můžeme načítat rovnou transponované. Stejně je zapotřebí načtené koeficienty ukládat do bloku v pořadí inverzním k cik-cak sekvenci. Stačí tedy vycházet místo z normální cik-cak sekvence ze sekvence pro transponovanou matici, takže tato operace nebude mít žádný vliv na rychlost načítání.

Otázkou však zůstává, jak provést co nejrychleji transpozici matice. Nabízí se sice možnost obsah registrů uložit do paměti a přeskládat klasickými instrukcemi jednotlivé složky vektorů tak, aby byly transponované, a poté obsah registrů z paměti opět načíst. To je ale velmi pomalý postup a samotní výrobci od podobných technik odrazují.

Pokud tedy budeme chtít blok transponovat co nejrychleji, budeme to muset provést pomocí instrukcí SSE2. A o tom pojednává kapitola 3.7.

Zbývá dořešit už jenom poslední věc. Transformace totiž pracuje s reálnými čísly, ale přitom jsme 128-bitový registr rozdělili na osm 16-bitových celočíselných hodnot.

Tato skutečnost nevadí u součtů a rozdílů, ale začne se projevovat u násobení

celého čísla číslem reálným, bez kterého se transformace přirozeně neobejde. Naštěstí jsou všechna taková reálná čísla konstantami, což tuto operaci usnadňuje. Navíc instrukční sada SSE2 takovou operaci nepřímo umožňuje.

Sada totiž obsahuje instrukci `pmulhw`, která provádí vektorový součin 16-bitových hodnot a jako výsledek použije horních 16 bitů 32-bitového výsledku. Neboli pro X a Y celá platí:

$$\texttt{pmulhw}(X, Y) = (X * Y) >> 16$$

A jak tato instrukce pomůže? Všimněme si toho, že pro A celé a B reálné platí:

$$\lfloor A * B \rfloor = \lfloor (A * 2^{16} * B)/2^{16} \rfloor = \lfloor (A * 2^{16} * B) \rfloor >> 16$$

Pokud označíme $C = \lfloor B * 2^{16} \rfloor$, tak platí:

$$\texttt{pmulhw}(A, C) = \lfloor A * B \rfloor$$

To je ale správně pouze za předpokladu, že `pmulhw` násobí bezznaménkově, a navíc pokud pro B platí následující nerovnost: $0 \leq B < 1$ (v opačném případě není hodnota C, vzhledem k jejímu 16-bitovému rozsahu, definována). Pokud ale ohlídáme, aby platilo, že B bude v intervalu $\langle -0{,}5; 0{,}5)$, pak tento trik funguje i pro znaménkové násobení.

Problém s omezeným intervalem povolených hodnot B opravíme poměrně snadno. Stačí si uvědomit, že platí:

$$\lfloor A * B \rfloor = \lfloor ((A * 2^p) * 2^{16} * (B/2^p)) \rfloor >> 16$$

Konstanta p pak může být zvolena tak, aby výraz $(B/2^p)$ byl v požadovaném intervalu. Pokud tedy označíme

$$\overline{C} = \lfloor 2^{16} * (B/2^p) \rfloor,$$

tak platí:

$$\texttt{pmulhw}(A << p, \overline{C}) = \lfloor A * B \rfloor$$

pro „rozumně velké“ reálné B.

3.7 Transpozice matice 16 bitových hodnot pomocí SSE2 instrukcí

Tato operace je potřebná pro výpočet inverzní kosinové transformace popsané v kapitole 3.6. Protože se v dalším textu ukáže, že transformaci budeme provádět pro jeden blok 8×8, pro dva bloky 4×4 a pro čtyři bloky 2×2, ukážeme algoritmy pro všechny transpozice, které během výpočtu budeme potřebovat.

Bohužel instrukční sada SSE2 nevychází příliš vstříc programátorovi, který by pomocí ní chtěl transponovat matice. Takže je nutné použít instrukce, které sice pro tuto operaci nejsou určené, ale její provedení usnadňují.

3.7.1 Transpozice čtyř matic 2×2 se 16-bitovými prvky

Mějme 4 matice o rozměru 2×2:

A_1	B_1	A_2	B_2	A_3	B_3	A_4	B_4
C_1	D_1	C_2	D_2	C_3	D_3	C_4	D_4

Abychom provedli jejich transpozici je nutné navzájem vyměnit prvky C_x a B_x. Je ale nutné počítat s tím, že architektura IA-32 ukládá data v kódování little-endian. To znamená, že méně významný byte vícebytového čísla se ukládá do paměti na nižší adresu než významější byte. Registry SSE2 v tomto ohledu netvoří výjimku, takže pokud načteme tyto dva řádky z paměti např. do registrů `xmm0` a`xmm1` bude jejich obsah následující:

`xmm0`	B_4	A_4	B_3	A_3	B_2	A_2	B_1	A_1
`xmm1`	D_4	C_4	D_3	C_3	D_2	C_2	D_1	C_1

Nyní lze využít toho, že kdyby se podařilo přeuspořádat obsah registrů takto:

`xmm0`	A_4	A_3	A_2	A_1	B_4	B_3	B_2	B_1
`xmm1`	C_4	C_3	C_2	C_1	D_4	D_3	D_2	D_1

stačilo by použít instrukci `punpck`, která umí zkombinovat obsah horních nebo dolních polovin dvou registrů do jednoho registru tak, že střídavě promíchá jejich složky. Jinými slovy stačí provést:

```
xmm2 = punpckhwd(xmm0, xmm1);
xmm3 = punpcklwd(xmm0, xmm1);
```

abychom v registrech `xmm2` a `xmm3` získali toto uspořádání:

`xmm2`	C_4	A_4	C_3	A_3	C_2	A_2	C_1	A_1
`xmm3`	D_4	B_4	D_3	B_3	D_2	B_2	D_1	B_1

Uložením těchto registrů do paměti se pořadí hodnot opět obrátí, takže získáme uspořádání přesně takové, jako jsme potřebovali.

Jak ale obsah registrů `xmm0` a `xmm1` přeuspořádat? Žádná instrukce, která by uměla najednou zpřeházet všech osm 16-bitových hodnot neexistuje. Je ale možné využít instrukcí `pshuf`, které umožňují přeskládat určité části registru podle zadané bitové masky. Instrukcí `pshuflw` lze přeuspořádat dolní polovinu registrů tak, aby se k sobě dostala stejná písmenka, pomocí `pshufhw` se stejným způsobem upraví horní polovina a nakonec instrukcí `pshufd` dostaneme jednotlivé prvky celého registru do požadovaného pořadí. Kód pak bude vypadat takto:

```
xmm0 = pshuflw(xmm0, 0xD8);
xmm1 = pshuflw(xmm1, 0xD8);
xmm0 = pshufhw(xmm0, 0xD8);
xmm1 = pshufhw(xmm1, 0xD8);
xmm0 = pshufd (xmm0, 0x8D);
xmm1 = pshufd (xmm1, 0x8D);
```

Pozn.: Místo tohoto postupu je možné použít podobný trik jako v následující části. Bohužel pro dva registry by měl kód velké množství závislostí, a proto procesor nemůže výpočet paralelizovat, takže ačkoliv kód vyjde na stejný počet instrukcí, je pomalejší.

3.7.2 Transpozice dvou matic 4×4 se 16-bitovými prvky

Nyní máme 2 matice o rozměru 4×4. Viz obrázek:

A_1	B_1	C_1	D_1	A_2	B_2	C_2	D_2
E_1	F_1	G_1	H_1	E_2	F_2	G_2	H_2
I_1	J_1	K_1	L_1	I_2	J_2	K_2	L_2
M_1	N_1	O_1	P_1	M_2	N_2	O_2	P_2

Po načtení do registrů např. `xmm0` až `xmm3` získáme:

`xmm0`	D_2	C_2	B_2	A_2	D_1	C_1	B_1	A_1
`xmm1`	H_2	G_2	F_2	E_2	H_1	G_1	F_1	E_1
`xmm2`	L_2	K_2	J_2	I_2	L_1	K_1	J_1	I_1
`xmm3`	P_2	O_2	N_2	M_2	P_1	O_1	N_1	M_1

V této situaci lze získat výsledné matice tak, že je jakoby budeme číst po sloupcích zdola nahoru a zprava doleva (operace osově souměrná s klasickou transpozicí). Tato operace rovněž nejde provést najednou a je nutné ji rozložit na několik elementárních kroků, ve kterých využijeme vlastností sady instrukcí začínající prefixem `punpck`. Nejdříve se sdruží jednotlivé hodnoty po dvojicích touto posloupností instrukcí:

```
xmm4 = punpcklwd(xmm0, xmm1);
xmm5 = punpckhwd(xmm0, xmm1);
xmm6 = punpcklwd(xmm2, xmm3);
xmm7 = punpckhwd(xmm2, xmm3);
```

Po této operaci bude obsah registrů `xmm4` až `xmm7` následující:

`xmm4`	H_1	D_1	G_1	C_1	F_1	B_1	E_1	A_1
`xmm5`	H_2	D_2	G_2	C_2	F_2	B_2	E_2	A_2
`xmm6`	P_1	L_1	O_1	K_1	N_1	J_1	M_1	I_1
`xmm7`	P_2	L_2	O_2	K_2	P_2	L_2	O_2	K_2

Nyní je možné sdružit dvojice do celých čtveřic těmito instrukcemi:

```
xmm0 = punpckldq(xmm4, xmm6);
xmm1 = punpckldq(xmm5, xmm7);
xmm2 = punpckhdq(xmm4, xmm6);
xmm3 = punpckhdq(xmm5, xmm7);
```

Což způsobí, že obsah registrů bude následující:

xmm0	N_1	J_1	F_1	B_1	M_1	I_1	E_1	A_1
xmm1	N_2	J_2	F_2	B_2	M_2	I_2	E_2	A_2
xmm2	P_1	L_1	H_1	D_1	O_1	K_1	G_1	C_1
xmm3	P_2	L_2	H_2	D_2	O_2	K_2	G_2	C_2

Tím jsme získali jednotlivé řádky výsledné matice, ale ve špatném pořadí. Proto je nutné provést ještě:

```
xmm4 = punpcklqdq(xmm0, xmm1);
xmm5 = punpckhqdq(xmm0, xmm1);
xmm6 = punpcklqdq(xmm2, xmm3);
xmm7 = punpckhqdq(xmm2, xmm3);
```

Nyní jsou v jednotlivých registrech konečně uloženy prvky v požadovaném pořadí:

xmm4	M_1	I_1	E_1	A_1	M_2	I_2	E_2	A_2
xmm5	N_1	J_1	F_1	B_1	N_2	J_2	F_2	B_2
xmm6	O_1	K_1	G_1	C_1	O_2	K_2	G_2	C_2
xmm7	P_1	L_1	H_1	D_1	P_2	L_2	H_2	D_2

Uložení těchto registrů do paměti způsobí, že se pořadí prvků horizontálně překlopí, což dá přesně požadovaný výsledek.

3.7.3 Transpozice jedné matice 8×8 se 16-bitovými prvky

Transpozice matice 8×8 používá téměř stejný algoritmus jako transpozice matice 4×4, a proto nebude popsaný tak podrobně. Myšlenka spočívá opět v tom, že nejdříve vytvoříme v korektním pořadí dvojice, poté čtveřice a nakonec celý výsledek. Nastává ale problém, že se pracuje se všemi registry najednou, takže nezbývají žádné volné na zazálohování jednoho z registrů před dvojicí operací `punpck`. Nelze se tedy vyhnout tomu, aby bylo nutné odložit hodnotu některých registrů do paměti a zpět. Protože se jedná o relativně pomalou operaci, je dobré počet jejích výskytů minimalizovat. Ukazuje se ale, že nejrychlejší řešení je takové, kdy se nejdříve předzpracují první čtyři řádky, poté poslední čtyři řádky a nakonec se výsledek spojí z obou částí.

3.8 Převod modelu YC_bC_r do modelu RGB

Když je provedena inverzní transformace na všechny bloky MCU bloku, tak krok který by měl následovat, je převod složek barevného modelu YC_bC_r na barevný model RGB, neboť data do obrazové paměti musí být ukládána právě tímto způsobem. Samotný převod se řídí těmito vztahy:

$$\begin{aligned}
R &= Y &&&&&+ \quad 1{,}402 \cdot (C_r - 128)\\
G &= Y &- \quad 0{,}34414 \cdot (C_b - 128) &- \quad 0{,}71414 \cdot (C_r - 128)\\
B &= Y &+ \quad 1{,}772 \cdot (C_b - 128)
\end{aligned}$$

Tento postup nelze bohužel nijak urychlit, ale lze výpočet alespoň vektorizovat. Je tedy možné spočítat převod pro několik bodů zároveň. Jediná potíž tak vzniká u násobení reálnými čísly. To ale lze provést stejným způsobem jako v kapitole 3.6, neboť se opět jedná o násobení celého čísla reálnou konstantou.

Drobný problém ale nastává v okamžiku, kdy budou barvonosné složky podvzorkovány. Pak je totiž nutné převzorkovat je na jejich původní rozlišení. Pokud jsou složky podvzorkované ve vertikálním směru, jedná se o jednodušší situaci, neboť stačí každý řádek 8 hodnot složek C_b a C_r zdvojit.

Obtížnější varianta nastává v případě, že jsou barvonosné složky podvzorkovány v horizontálním směru. Pak je hodnoty nutné opět zduplikovat, ale v rámci jednoho řádku. To je naštěstí možné provést poměrně snadno již mnohokrát použitými instrukcemi `punpcklwd` a `punpckhwd`, tentokrát však se stejným vstupním a cílovým registrem. Pokud bude např. v registru `xmm0` uloženo těchto 8 hodnot:

xmm0	A	B	C	D	E	F	G	H

budou po provedení instrukcí

```
xmm1 = punpcklwd(xmm0, xmm0);
xmm2 = punpckhwd(xmm0, xmm0);
```

v registrech `xmm1` a `xmm2` tyto hodnoty:

xmm1	A	A	B	B	C	C	D	D
xmm2	E	E	F	F	G	G	H	H

Pokud jsou složky podvzorkovány v obou směrech, oba zmíněné postupy se zkombinují.

Ještě je třeba vyřešit skutečnost, že hodnoty, které jsme celým postupem získali, jsou 16-bitové, zatímco pro zobrazení jsou třeba převést na 24-bitový model, kde na jednu složku připadá 8-bitů. K tomu ale slouží instrukce `packuswb`, která převede osm 16-bitových hodnot ze dvou registrů na šestnáct hodnot 8-bitových do registru jednoho. Navíc toto provede s tzv. saturací, která ze záporných čísel vytvoří nulu a z čísel, která se nevejdou do rozsahu jednoho bytu, udělá číslo 255. To je velmi důležité proto, že vlivem kvantizace a nepřesné aritmetiky se může stát, že výsledná hodnota se nevejde do požadovaného rozsahu.

3.9 Smíchání barevných kanálů

Konečně nastává situace, kdy je MCU blok dekódovaný a máme 8-bitové hodnoty jednotlivých bodů v barevném prostoru RGB. Bohužel ještě nelze tato data uložit do obrazové paměti, neboť jednotlivé složky jsou od sebe odděleny a je třeba je přerovnat tak, aby každý bod byl posloupností 3 bytů – jeden pro složku R, druhý pro G a třetí pro B. Tato operace však nelze jednoduše provést pomocí SSE2 instrukcí. Proto si operaci zjednodušíme tak, že doplníme složky RGB o alfa-kanál. Tento kanál

udává průhlednost bodu, neboli s jakou průhledností tento bod překrývá obrázek v pozadí. Pokud tuto složku nastavíme na absolutní neprůhlednost, získáme stejný výsledek jako v případě modelu RGB, s tím, že v modelu RGBA připadají 4 byty na jeden bod, což operaci smíchání složek velmi usnadní.

Jsou-li tedy 4 registry naplněné těmito hodnotami,

xmm0	R_{15}	R_{14}	R_{13}	R_{12}	.....	R_3	R_2	R_1	R_0
xmm1	G_{15}	G_{14}	G_{13}	G_{12}	.....	G_3	G_2	G_1	G_0
xmm2	B_{15}	B_{14}	B_{13}	B_{12}	.....	B_3	B_2	B_1	B_0
xmm3	A_{15}	A_{14}	A_{13}	A_{12}	.....	A_3	A_2	A_1	A_0

můžeme použít podobný trik jako při transpozici matic a provedeme instrukce:

```
xmm4 = punpcklbw(xmm0, xmm1);
xmm5 = punpcklbw(xmm2, xmm3);
xmm6 = punpckhbw(xmm0, xmm1);
xmm7 = punpckhbw(xmm2, xmm3);

xmm0 = punpcklwd(xmm4, xmm5);
xmm1 = punpckhwd(xmm4, xmm5);
xmm2 = punpcklwd(xmm6, xmm7);
xmm3 = punpckhwd(xmm6, xmm7);
```

Poté bude v registrech uložen následující obsah:

xmm0	A_3	B_3	G_3	R_3	.....	A_0	B_0	G_0	R_0
xmm1	A_7	B_7	G_7	R_7	.....	A_4	B_4	G_4	R_4
xmm1	A_{11}	B_{11}	G_{11}	R_{11}	.....	A_8	B_8	G_8	R_8
xmm1	A_{15}	B_{15}	G_{15}	R_{15}	.....	A_{12}	B_{12}	G_{12}	R_{12}

Nyní lze konečně tyto registry uložit na příslušné místo v obrazové paměti a pokračovat načítáním dalšího MCU bloku.

Kapitola 4

Metody urychlující prohlížení obrázků

V předchozí kapitole byly vysvětleny metody, jak co nejrychleji dekódovat JPEG soubory, za využití nejrůznějších technik a instrukční sady SSE2. I přesto, že pomocí nich lze dosáhnout znatelného urychlení vlastního dekódování, zůstává tato operace relativně pomalá a v případě větších obrázků i paměťově náročná.

Tato kapitola již proto nepopisuje metody, které by měly dále optimalizovat proces dekódování, ale přibližuje, jak co nejvíce urychlit běžné operace prováděné s obrázky, jako například změna měřítka, vykreslování výřezů, procházení většího množství souborů a zobrazování obrázků s vysokým rozlišením.

4.1 Zobrazování obrázků s vysokým rozlišením

Při dekódování velkých obrázků nastává zpravidla problém, že se dekódovaná data ani nemusí vejít do paměti. Operační systém takovou situace začne obvykle řešit odkládáním operační paměti na disk, což neúnosně zpomaluje proces otevírání a občas se ani kvůli nedostatku paměti nemusí celá operace zdařit. Navíc je manipulace s takto velkými obrázky neúměrně pomalá. Z těchto důvodů je nutné modifikovat dekódovací algoritmus tak, aby tyto potíže nenastávaly.

Stačí si ale všimnout, že uživatel nepotřebuje mít dekódovaný celý obrázek, neboť stačí, když bude dekódováno právě tolik informací, kolik je třeba k tomu, aby se požadovaný výsledek zobrazil na obrazovce. Existují pouze dvě základní operace, které při prohlížení obrázku může uživatel požadovat:

1. Zobrazení určitého výřezu.

2. Změna měřítka obrázku.

Protože každá z těchto operací vyžaduje poněkud odlišný přístup, bude rozebrána každá z nich zvlášť.

4.1.1 Dekódování výřezu obrázku

Největším a téměř jediným problémem, který nastává při dekódování výřezů je skutečnost, že soubor je sekvenční. Neexistuje tak možnost, jak dekódovat dat od určité pozice v souboru, aniž by bylo nutné začít od začátku.

Můžeme ale využít toho, že program při otevření obrázku ho stejně dekóduje celý proto, aby mohl zobrazit jeho celkový náhled. Během tohoto úvodního dekódování si tak můžeme vytvořit seznam záchytných bodů, ve kterých si uložíme aktuální stav celého dekodéru.

Budeme-li chtít dekódovat od libovolné pozice v souboru, stačí začít z odpovídajícího záchytného bodu a spustit dekódování až z něj. S touto myšlenkou se bohužel pojí poněkud nepříjemná skutečnost, neboť stav dekodéru vyžaduje k uložení několik desítek bytů, a kdybychom chtěli vytvořit záchytný bod pro každý MCU blok, může velikost seznamu i mnohonásobně přerůst velikost vstupního souboru.

Takže není možné pamatovat si všechny záchytné body, ale pouze jejich podmnožinu. Když pak program bude chtít dekódovat od určité pozice, stačí začít od nejbližšího předchozího bodu, přeskočit několik prvních MCU bloků, a tak se lze dostat na požadovanou pozici.

S tímto mechanismem je tedy možné začít dekódování velmi rychle v libovolné pozici v souboru, takže z dekódování výřezů obrázků se stává jednoduchá záležitost.

4.1.2 Změna měřítka obrázku

Požadavek na změnu měřítka nastane již při otevření souboru, neboť v ten okamžik zpravidla uživatel požaduje zobrazení náhledu celého obrázku. Je tedy nutné změnit měřítko tak, aby se obrázek akorát vešel na obrazovku.

Dekódovat celý obrázek a pak ho převzorkovat na požadovanou velikost jsme již zavrhli, neboť obrázek se nemusí vejít do operační paměti. První vylepšení, které se nabízí, je využití skutečnosti, že lze dekódovat výřezy obrázku. Bylo by tak možné číst obrázek po menších částech, ty rovnou zmenšovat a poslepovat z nich náhled celého obrázku. Tento přístup by byl sice možný, ale mohly by vzniknout artefakty na spojích jednotlivých částí vlivem převzorkování. Navíc čtení obrázku po úsecích by také načítání nepříjemně zpomalilo. Rovněž by se zbytečně dekódovalo mnoho dat, která by se nijak nevyužila, takže by dekodér vykonal mnoho práce navíc.

Nejlepší by tedy bylo načíst celý obrázek najednou a rovnou zmenšený. Toto je sice poměrně nereálný požadavek, ale pokud omezíme povolené velikosti měřítka pouze na určité hodnoty, pak lze tuto operaci provést. Lze totiž využít toho, že data jsou uložena v blocích 8×8. Pokud bychom načítali tyto bloky rovnou zmenšené na 4×4, 2×2 nebo 1×1 dosáhneme dvoj, čtyř a osminásobného zmenšení rovnou při dekódování.

Nejdříve se ale musí vyřešit problém, které koeficienty je vlastně třeba načítat. Kdyby totiž byla data uložena v klasickém časovém prostoru, bylo by logické načítat např. pro zmenšený blok 4×4 ty hodnoty, které se vyskytují na sudých řádcích a sudých sloupcích. Díky použití kosinové transformace jsou ale koeficienty uloženy ve frekvenčím prostoru, takže je třeba načíst ty, které nesou největší množství infor-

mace. Tento požadavek splňují koeficienty nízkofrekvenční, které se navíc se nejméně kvantizují. Tyto koeficienty se ale nacházejí v levém horním rohu, takže pro bloky 4×4 načteme horní levou podmatici o rozměrech 4×4, pro bloky 2×2 bude postup analogický a pro blok 1×1 načteme pouze DC-koeficient.

Při načítání lze s výhodou využít toho, že pořadí koeficientů je uloženo v cik-cak sekvenci. Takže např. pro načtení horní podmatice 4×4 není nutné načítat celé 3 řádky a polovinu čtvrtého (28 hodnot), ale pouze prvních 19 hodnot, pro matici 2×2 stačí načíst pouhých 5 hodnot a pro matici 1×1 stačí načíst první hodnotu. Bohužel není možné po načtení příslušného počtu hodnot rovnou přeskočit na další blok, neboť není jasné, kde začíná. Ale i přesto, že je stále nutné dekódovat celé bloky, je možné dosáhnout určitého urychlení. Není totiž třeba přepočítávat kódy hodnot na skutečnou číselnou hodnotu koeficientu (viz kapitola 2.1.7) a ukládat je do paměti.

Tato operace nás naštěstí ani nepřipravuje o možnost používat stávající kód v SSE2 instrukcích. Lze totiž data načítat tak, že vždy vyplníme blok 8×8 menšími bloky. Např. blok 4×4 se do bloku 8×8 vejde čtyřikrát, blok 2×2 už šestnáctkrát a blok 1×1 šedesátčtyřikrát. Stačí tak pouze upravit kód pro počítání inverzní kosinové transformace tak, aby se neprováděla na celý blok 8×8, ale na jednotlivé podbloky.

Taková úprava je velmi jednoduchá, neboť algoritmus použijeme stejný jako v případě bloku 8×8, jenom se dosadí místo nenačtených koeficientů nuly. Tím se transformace velmi zjednoduší, takže dosáhneme dalšího zrychlení. Navíc nehrozí, že by kapacita XMM registrů nebyla plně využita. Pokud totiž tyto registry budou stále rozděleny na osm 16-bitových prvků, lze provádět např. 4 transformace bloků 2×2 najednou.

Jestliže budou podbloky poskládány v šikovném pořadí, není třeba dokonce vůbec modifikovat kód na převzorkování barvonosných složek a převod barevných prostorů. Jediné v čem se kód bude trochu lišit, jsou adresy v obrazové paměti, na které budou ukládány výsledky těchto operací.

Nyní umíme zmenšit obrázek dvakrát, čtyřikrát a osmkrát. Libovolného měřítka lze nyní dosáhnout jednoduše tím, že obrázek se dekóduje na nejbližší vyšší možnou velikost a teprve pak se procedurou na převzorkování dosáhne požadovaného měřítka.

Mohlo by se zdát, že osminásobné snížení rozlišení může být pro veliké obrázky stále málo, ale je třeba si uvědomit, že velikost paměti nutné k uložení obrázku klesá s druhou mocninou velikosti zmenšení. Takže např. obrázek s rozměry 30000×30000 by v paměti v měřítku 1:1 obsadil 2,6GB, zatímco v měřítku 1:8 už to bude 64-krát méně, tedy pouhých 40MB, takže osminásobné zmenšení je zřejmě dostatečné.

4.2 Optimalizace prohlížení více souborů

Vzhledem k tomu, že data jsou uložena v souboru sekvenčně, nelze dekódování paralelizovat spuštěním výpočtu ve více vláknech. Naštěstí to ale neznamená, že by se více vláken k prohlížení obrázků použít nedalo. Lze totiž s výhodou využít toho, že uživatel zpravidla neprohlíží jeden obrázek, ale celou posloupnost. Proto je možné

dekódovat ve zvláštních vláknech obrázky dopředu tak, aby se na další nemuselo dlouho čekat. K tomu, aby tento postup fungoval, je ale nutné nalézt odpovědi na následující otázky:

1. Které obrázky přednačítat?
2. Jak provádět samotné přednačítání?

4.2.1 Výběr obrázků k přednačítání

Nejtriviálnější řešení, které se nabízí, je vytvořit si frontu obrázků o pevné, předem zvolené velikosti. Na začátku fronty bude vždy obrázek, který se v dalším kroku má zobrazit na obrazovce. Na ostatních pozicích pak v příslušném rostoucím pořadí fronta obsahuje přednačítané obrázky. Při požadavku na vykreslení dalšího obrázku se z fronty odebere její první prvek a zobrazí se na obrazovku, zatímco na její konec se přidá další obrázek k přednačítání.

Toto řešení ale funguje pouze za předpokladu, že se obrázky budou procházet pouze jedním směrem.[1] Pokud by nastal požadavek na vykreslení obrázku o jednu pozici zpět, muselo by se počkat opět na jeho dekódování, neboť jeho náhled byl zapomenut. Ještě větším problémem by bylo procházení obrázků v opačném směru, než pro který je určená fronta. Pak by prohlížení vůbec neurychlovala a navíc by zabírala příliš mnoho zbytečného místa v paměti.

Ale při některých činnostech není žádný z obou směrů dominantní, takže pro takové účely je třeba vymyslet řešení nezávislé na směru procházení. Takovým řešením jsou například fronty dvě – každá pro jeden směr.

Následující obrázek ukazuje situaci, kdy obě fronty mají velikost 2 prvky a právě se zobrazuje obrázek `obr3.jpg`.

Fronta pro směr zpět			Fronta pro směr vpřed	
Konec	Začátek	Aktuální obrázek	Začátek	Konec
`obr1.jpg`	`obr2.jpg`	`obr3.jpg`	`obr4.jpg`	`obr5.jpg`

Při posunu například o jeden obrázek vpřed se s frontou pro směr vpřed provede operace popsaná výše. U fronty zpět se jakoby obrátí směr, neboť se odebere prvek z jejího konce a na začátek se vloží obrázek, který byl v tuto chvíli na obrazovce. Tedy `obr3.jpg`. Pro opačný směr by pak operace proběhla pouze naopak.

Takže po výše popsané operaci by situace vypadala následovně:

Fronta pro směr zpět			Fronta pro směr vpřed	
Konec	Začátek	Aktuální obrázek	Začátek	Konec
`obr2.jpg`	`obr3.jpg`	`obr4.jpg`	`ob5.jpg`	`obr6.jpg`

Tento postup zřejmě odstraňuje nedostatky, které obsahuje řešení s jednou frontou. Na druhou stranu není příliš paměťově šetrné, neboť pokud bude jeden ze směrů dominantní, nebude se fronta pro druhý směr téměř využívat.

[1] Například při prezentaci fotografií

Takže se nabízí použití adaptivního algoritmu, který sleduje jakým směrem uživatel soubory prochází a podle toho je přednačítá. Takové řešení může vycházet z předcházejícího postupu se dvěma frontami. Pouze jejich velikost nebude stále stejná, ale bude se měnit podle chování uživatele. Vždy ale bude platit, že součet jejich velikostí bude po celou dobu konstantní, aby se alokovaná paměť plně využila.

Když pak uživatel prochází soubory např. od začátku do konce, tak se zvětšuje fronta pro přednačítání dopředu na úkor fronty pro přednačítání dozadu. Pro opačný směr pak operace bude probíhat přesně naopak.

4.2.2 Přednačítání obrázků

Samotné přednačítání je velmi jednoduché. Stačí vytvořit nové vlákno, ve kterém se spustí dekódování příslušného souboru. Problematické je spíše to, že kdyby se pustilo vláken příliš mnoho, může se chod programu celkově zpomalit. Pokud bude vláken málo, nevyužije se naplno výkon, který počítač nabízí. Navíc je nutné počítat s tím, že běh více vláken bude zvyšovat latenci programu při prohlížení obrázků.

Je tedy nutné nějakým způsobem vyřešit přidělování času procesoru jednotlivým vláknům. Zde se již budou lišit strategie na přidělování procesoru pro počítače s jedním procesorem a pro počítače s více procesory.

Strategie určená pro vícejádrové nebo víceprocesorové systémy může být taková, že se program bude snažit co nejvíce vytížit výpočetní kapacitu. Bude se tedy snažit, aby v jednom okamžiku běželo právě N vláken, kde N bude např. počet procesorů resp. jader v systému. Problém ale nastává ve chvíli, když všech N vláken přednačítá obrázky a my bychom chtěli překreslit obrázek např. v jiném měřítku. Pak by ostatní vlákna nepříjemně zpomalovala tuto operaci, takže by se na výsledek muselo čekat zbytečně dlouho. Situaci lze vyřešit například tak, že se vlákno, které přednačítá obrázek s nejnižší prioritou (je od toho aktuálně zobrazeného nejvzdálenější) pozastaví a místo něj se spustí vlákno, které překreslí obrázek. Po ukončení jeho práce se opět spustí pozastavené vlákno.

Tento postup sice vytěžuje výpočetní výkon počítače v největší míře, ale není příliš šetrná k okolním procesům. Proto lze z této strategie odvodit takovou, která by byla méně agresivní k ostatním procesům. Můžeme totiž snížit prioritu vláken, která přednačítají obrázky dopředu, zatímco překreslovací vlákno poběží s prioritou normální.

Obě výše popsané strategie však nemusí dobře fungovat na počítačích s jedním procesorem. Přednačítající vlákna totiž blokují překreslovací vlákno, a tak zvyšují latenci celého programu. Proto poslední, v přiloženém programu, implementovaná strategie je taková, že přednačítací vlákna běží s nízkou prioritou a v okamžiku, kdy je puštěno vlákno překreslovací, jsou všechna vlákna, která přednačítají obrázky, pozastavena do doby, než bude výsledný obrázek zobrazen. Tím se uvolní prostředky procesoru a samotné překreslování není ničím zpomalováno.

Kapitola 5

Popis přiloženého programu

Přílohou k této práci je program, který všechny popsané optimalizace implementuje a ukazuje jejich reálnou použitelnost. Tento program je napsaný v jazyku C s rozšířeními překladače GCC. Ke grafickému výstupu a komunikaci s uživatelem využívá knihovnu Gtk+ [9] a k práci s vlákny a synchronizačními primitivy knihovnu Glib [9].

Zdrojové kódy jsou určeny pro překladač GCC [7] verze alespoň 4.3, neboť tato verze nabízí možnosti, jak pohodlně pracovat se všemi požadovanými SSE2 instrukcemi i bez použití asembleru.

Podporovanými operačními systémy jsou Unix a Windows, na kterých je nainstalovaná knihovna Gtk+ verze 2.0 a vyšší.

Program je napsán tak, aby šel přeložit a správně fungoval pod libovolnou architekturou, nicméně vyzkoušeny byly pouze IA-32 a AMD64.

5.1 Popis práce s programem

5.1.1 Překlad a instalace

Program lze snadno přeložit za pomoci přiloženého `makefile` souboru. Podporované parametry překladu jsou popsané v dokumentaci u tohoto souboru.

Pro prostředí Windows je přiložen přeložený spustitelný soubor včetně instalačního balíčku knihovny Gtk+ verze 2.12.9.

5.1.2 Spuštění a nastavení

Veškerá nastavení programu lze ovládat pomocí parametrů předaných na příkazové řádce. Základní syntaxe pro spuštění programu je tato:

```
jpeg [přepínače] [jméno_souboru|jméno_adresáře]
```

Pokud je zadáno jméno souboru, zobrazí se na obrazovce tento soubor jako první. Zároveň s ním se ale načte obsah adresáře, ve kterém se nachází, takže je možné postupně procházet všechny soubory v tomto adresáři. Pokud je zadáno jméno adresáře,

načte se jeho obsah a je zobrazen první podporovaný soubor v tomto adresáři. Pokud není zadané jméno souboru ani adresáře, je načten obsah adresáře, ze kterého byl program spuštěn.

Seznam podporovaných přepínačů je uveden v tabulce 5.1.

5.1.3 Ovládání programu

Protože je program navržen pouze pro prohlížení obrázků a veškerá nastavení jsou předávána pomocí příkazové řádky při spuštění, nemá žádné grafické uživatelské rozhraní. Program je ovládán pouze prostřednictvím klávesnice a myši.

Při stisknutém levém tlačítku myši kopíruje obrázek pohyby kurzoru. Tím je možné dosáhnout jeho plynulého pohybu po obrazovce. Kolečkem pak lze nastavit měřítko, ve kterém obrázek bude zobrazen.

Seznam podporovaných kláves je uveden v tabulce 5.2.

5.2 Popis souborů se zdrojovými kódy

Zdrojové kódy jsou rozděleny do několika souborů, přičemž téměř každý ze souborů implementuje některou z popsaných technik. Tento přehled by tak měl usnadnit orientací v těchto souborech a stručně popsat jejich účel.

5.2.1 Soubor `bit_reader.c`

Soubor `bit_reader.c` implementuje operace nutné k načítání datové části vstupního souboru jako proudu bitů. Jedná se o přímý přepis postupu zmíněného v kapitole 3.3. Obsahuje metodu pro inicializaci z libovolné pozice ve vstupním souboru a metodu pro posun okénka.

5.2.2 Soubor `decoder.c`

Soubor obsahuje implementaci třídy `decoder`. Ta implementuje metody pro načítání celých MCU bloků ze vstupního souboru. Všechny metody obsahuje ve dvojím provedení. První varianta načítá bloky přesně tak, jak by měla, zatímco druhá varianta ukládá všechna data transponovaně. Druhá varianta je implementovaná z důvodu zmíněného v kapitole 3.6. Metody navíc umí načítat MCU bloky v plném rozlišení a podvzorkované 2, 4 a 8-krát, přesně v duchu kapitoly 4.1.2. Rovněž obsahuje metody, které umožňují uložit a znovu načíst stav objektu. To je důležité proto, aby bylo možné vytvořit index záchytných bodů (viz kapitola 4.1.1).

5.2.3 Soubor `idct.c`

Tento soubor se vyskytuje ve dvou variantách. V jedné jsou funkce provádějící inverzní kosinovou transformaci implementované nezávislé na cílové architektuře. Ve druhé jsou pak tytéž funkce implementované za použití instrukcí ze sady SSE2.

Přepínač	**Popis**
`--help`	Zobrazí nápovědu pro použití programu.
`-a`, `--agressive`	Všechna vlákna v programu běží s normální prioritou.
`-s`, `--standard`	Vlákna, která přednačítají další soubory, běží se sníženou prioritou.
`-i`, `--interactive`	Při překreslování obrázku jsou všechna přednačítací vlákna pozastavena. V opačném případě běží vlákna se sníženou prioritou.
`-f`, `--fullscreen`	Program je spuštěn na celou obrazovku.
`-hq`, `--high_quality`	Je použitý kvalitnější algoritmus na převzorkování.
`-qp`, `--quick_preview`	Nejdříve je zobrazen rychlý náhled, který je později překreslen kvalitnějším obrázkem.
`-lq`, `--low_quality`	Používá rychlý algoritmus na převzorkování obrázku.
`-t N`, `--threads=N`	Určuje maximální počet zároveň běžících vláken.
`-b N`, `--buffer_size=N`	Počet položek, které slouží k přednačítání obrázků dopředu. Ty jsou rozděleny na přednačítací fronty pro oba směry.
`--forward_min=N`	Minimální velikost fronty k přednačítání v dopředném směru.
`--backward_min=N`	Minimální velikost fronty k přednačítání obrázků směrem vzad.
`-w N`, `--width=N`	Šířka okna vytvořeného k prohlížení obrázků.
`-h N`, `--height=N`	Výška okna pro prohlížení obrázků.
`-d`, `--debug`	Zapne výpis ladících informací na konzoli.
`-p`, `--progress`	Zapne zobrazování aktuálního stavu načítání do titulku hlavního okna.
`-u`, `--unsupported`	Pokud narazí během prohlížení na obrázek JPEG, který neumí otevřít, použije knihovnu libjpeg [13]. To ale může způsobit problémy, neboť program pak nemá kontrolu nad jeho dekódováním. Takové obrázky nejsou načítány dopředu a není možné je zobrazit jinak, než na celou obrazovku.

Tabulka 5.1: Podporované přepínače programu

Klávesa	Popis
`Enter`	Přepnutí mezi zobrazením v okně a zobrazením na celou obrazovku.
`PgUp`	Přesun na předchozí obrázek v adresáři.
`PgDn`	Přesun na další obrázek v adresáři.
`Šipky`	Posun obrázku v příslušném směru o 100 pixelů.
`+,-`	Změna měřítka obrázku.
`Esc`	Ukončení programu.

Tabulka 5.2: Seznam kláves pro ovládání programu

5.2.4 Soubory `jpeg.c` a `decode.c`

V těchto souborech je implementována hlavní třída celého programu – `jpeg`. Tato třída umožňuje otevřít soubor a vykreslit jeho libovolný výřez v libovolném měřítku. Během dekódování a převzorkovávání pak volá pravidelně notifikační funkci, na níž dostane ukazatel při inicializaci objektu. Této funkci předává kolik procent výřezu již dekódovala. Navíc lze vhodnou návratovou hodnotou funkce celý výpočet přerušit. Tento mechanismus tak umožňuje mimo zobrazování průběhu nejen celý proces ukončit, ale případně i pozastavit.

5.2.5 Soubor `image.c`

Soubor `image.c` obsahuje implementaci třídy `image`. Ta rozšiřuje možnosti třídy `jpeg` o operace, které uživatel běžně provádí při prohlížení obrázků. Jedná se o otevírání souborů, posun obrázků v libovolném směru a změnu měřítka. Navíc se tato třída snaží minimalizovat množství operací nutných k zobrazení výsledku, takže například při posunu obrázku nedekóduje celý výřez, ale recykluje části, které na obrazovce po posunu zůstanou. Rozhraní je navrženo tak, aby bylo možné tuto třídu používat ve vícevláknovém prostředí.

5.2.6 Soubor `threaded_image.c`

V tomto souboru se nachází implementace třídy `threaded_image`, která navazuje na třídu `image` a rozšiřuje ji o možnost pustit dekódování v jiném vlákně a toto vlákno pohodlně ovládat. Takže obsahuje metody pro ukončení, pozastavení, znovuspuštění a změnu priority dekódovacího vlákna.

Princip práce se třídou je takový, že veškeré operace jako posun obrázku či změna měřítka jsou prováděny bez jakéhokoliv efektu na výsledném obrázku. Teprve, když se spustí samotné překreslování, začne se dekódovat část obrázku, která odpovídá předchozím provedeným operacím. Během dekódování je pak možné stále volat metody na posun a změnu měřítka obrázku, které se ale projeví až při příštím překreslení. Tento postup efektivně zabraňuje přehlcení množstvím požadovaných operací s obrázkem, které mohou nastat například při posunu obrázku myší.

Ukončení a zastavování/znovuspouštění dekódovacího vlákna využívá notifikační funkci z třídy `image` a `jpeg`. Při požadavku na ukončení překreslování nastaví návratovou hodnotu na „false“, zatímco při požadavku na pozastavení počká s návratem z notifikační funkce až do požadavku na znovuspuštění.

Třída zavádí další notifikační funkci, která je volána vždy při ukončení dekódování obrázku.

5.2.7 Soubor `viewer.c`

Třída `viewer` dále rozšiřuje funkčnost třídy `threaded_image`. Využívá možnosti pouštět překreslování ve vlastním vlákně a díky tomu může implementovat metody popsané v kapitole 4.2. Tato třída obsahuje vlastní smyčku zpráv, do které se přeposílají zprávy jednak ze smyčky zpráv knihovny Gtk+ a jednak zprávy notifikační o aktuálním stavu dekódování a o ukončování překreslovacích vláken. Na základě těchto zpráv řídí překreslování aktuálního obrázku a přednačítání obrázků dle zvolené strategie.

5.2.8 Soubor `resize.c`

V tomto souboru jsou implementované funkce na změnu rozlišení obrázku. Obsahuje dvě funkce. První slouží pro vytváření rychlých náhledů a druhá, která vychází z algoritmu Lanczos [4], je určena pro kvalitnější zobrazování.

5.2.9 Soubor `my_api.c`

Tento soubor obsahuje velké množství pomocných funkcí použitých ve zbytku programu a odstiňuje rozdíly mezi operačními systémy Windows a Unix. Díky funkcím z tohoto souboru je možné procházet soubory ve zvoleném adresáři a paměťově mapovat soubory. Navíc zapouzdřují funkce knihovny Glib, aby jejich rozhraní bylo konzistentní se zbytkem programu.

Kapitola 6

Závěr

6.1 Porovnání rychlosti jinými programy

Program byl testován na počítači s procesorem Intel®Core™2 Duo se sníženou taktovací frekvencí na 1 GHz, s operační pamětí 1 GB RAM typu DDR2 667 MHz a diskem s 5400 otáčkami za minutu. Za testovací data posloužilo 107 digitálních fotografií s rozlišením 2272×1704.

První porovnání proběhlo s knihovnou libjpeg [13] používanou k načítání obrázků v knihovně Gtk+. Obrázky byly otevírané v plném rozlišení, aby se ve výsledku neprojevil čas nutný k převzorkování obrázku. S knihovnou libjpeg trvalo dekódování všech obrázků 50,3 vteřiny. Přiloženému programu trvala tato operace 34,7 vteřin pokud bylo povolené nejvýše jedno vlákno a 25,8 vteřin, pokud byla povolená vlákna dvě. To znamená urychlení 31% pro jedno vlákno a 49% pro vlákna dvě.

Druhou aplikací, se kterou byla porovnávána rychlost programu, je IrfanView [10] ve verzi 4.10, což je freewareový program na prohlížení obrázků v prostředí Windows. Tentokrát byly obrázky při otevření okamžitě zmenšeny tak, aby se vešly na obrazovku s rozlišením 1400×1050. V obou případech byl použit stejný základní podvzorkovávací algoritmus založený na přímém výběru bodů. Programu IrfanView zabralo dekódování všech souborů 50 vteřin, zatímco přiloženému programu s využitím jednoho vlákna 22,6 a s využitím dvou vláken pouhých 16 vteřin. Tomu odpovídá zrychlení o 55% v případě jednoho vlákna a 68% v případě dvou vláken.

Pozn.: Skutečnost, že dekódování stejných obrázků včetně změny jejich velikosti je rychlejší než jejich dekódování a zobrazení v plném rozlišení, je způsobena tím, že není nutné přesunovat tak velké množství dat do obrazové paměti.

6.2 Shrnutí

Techniky představené v této práci umožnily vytvořit program, který výrazně urychluje prohlížení obrázků zkomprimovaných sekvenční metodou JPEG. Použití 32-bitového načítaní dat (i za cenu velkých komplikací) společně s dvojúrovňovým dekódováním Huffmanových symbolů urychluje samotné dekódování struktury souboru. Instrukce SSE2 pak umožňují provádět navazující operace vektorově, což dále

přispívá ke zvýšení rychlosti načtení obrázku do obrazové paměti.

Použití více vláken k přednačítání dalších obrázků spolu s adaptivním algoritmem, který určuje, které obrázky budou nejpravděpodobněji v dalších krocích zobrazeny, způsobuje, že je uživateli proces dekódování (a tím i doba potřebná k otevření souboru) ve většině případů zcela skrytý.

Techniky umožňující zobrazování výřezů obrázků a změny měřítka pak umožňují pohodlnou práci i se soubory, k jejichž celkovému dekódování nemusí počítač disponovat potřebným množstvím paměti.

Navíc je struktura programu navržena tak, aby téměř všechny jeho části byly použitelné samostatně. Díky tomu by bylo možné snadno vytvořit např. platformově nezávislou knihovnu, která by umožňovala využívat všechny použité techniky i v jiných programech, a tak rozšířit jejich funkčnost.

Literatura

[1] Advanced Micro Devices, Inc.: *AMD Athlon^{TM}Processor x86 Code Optimization Guide* [online, PDF].
Revize K z února 2002 [cit. 2008-05-26].
URL: <http://www.amd.com/
us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf>.

[2] Antaeus Feldspar. *An Explanation of the Deflate Algorithm* [online].
Verze z 13. dubna 2002. [cit. 2008-05-26].
URL: <http://zlib.net/feldspar.html>

[3] CompuServe Incorporated: *Graphics interchange format* [online].
Verze 89a z 31 července 1990 [cit. 2008-05-26].
URL: <http://www.w3.org/Graphics/GIF/spec-gif89a.txt>.

[4] Ken Turkowski: *Filters for Common Resampling Tasks* [online, PDF].
Verze z 10. dubna 1990 [cit. 2008-05-26].
URL: <http://www.worldserver.com/
turk/computergraphics/ResamplingFilters.pdf>

[5] Intel Corporation: *Intel®Architecture Optimization – Reference manual* [online, PDF].
Revize 001 z února 1999 [cit. 2008-05-26].
URL: <http://download.intel.com/design/PentiumII/manuals/24512701.pdf>.

[6] International telecommunication union: *Digital compression and coding of continuous-tone still images, requirements and guidelines* [online, PDF]
Revize ze září 1992 [cit. 2008-05-26].
URL: <http://www.w3.org/Graphics/JPEG/itu-t81.pdf>

[7] *GCC, the GNU Compiler Collection* [online].
Revize z 24. května 2008 [cit. 2008-05-26].
URL: <http://gcc.gnu.org/>

[8] *Google Earth* [online].
URL: <http://earth.google.com/>
A virtual globe program.

[9] *The Gtk+ project* [online]
©2007-2008 [cit. 2008-05-26].
URL: <http://www.gtk.org/>
A toolkit for creating graphical user interfaces.

[10] Irfan Skiljan *IrfanView* [online].
Verze 4.10, ©2008 [cit. 2008-05-26].
URL: <http://www.irfanview.com/>
A freeware (for non-commercial use) graphic viewer for Windows.

[11] Jan Čapek, Peter Fabian: *Komprimace dat, principy a praxe*.
1. vydání, Praha: Computer Press, 2000. ISBN 80-7226-231-9.

[12] *PNG (Portable Network Graphics) Home Site* [online].
Revize z 16. dubna 2008 [cit. 2008-05-26].
URL: <http://www.libpng.org/pub/png/>

[13] *The Independent JPEG Group's JPEG Library* [online].
Verze 6b z 27. března 1998.
URL: <ftp://ftp.simtel.net/pub/simtelnet/msdos/graphics/jpegsr6.zip>

[14] Pavel Tišnovský: *Ztrátová komprese obrazových dat pomocí JPEG* [online].
Revize z 14. prosince 2006, [cit. 2008-05-26].
URL: <http://www.root.cz/clanky/
ztratova-komprese-obrazovych-dat-pomoci-jpeg/>