VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

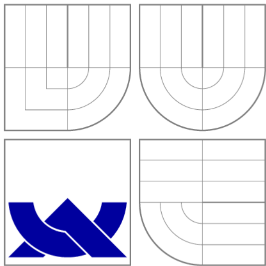
AKCELEROVANÁ DEKOMPRESE OBRÁZKŮ VE FORMÁTU JPEG NA GRAFICKÝCH KARTÁCH

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

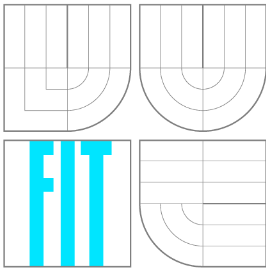
AUTOR PRÁCE
AUTHOR

ONDŘEJ JANOŠÍK

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

AKCELEROVANÁ DEKOMPRESE OBRÁZKŮ VE FORMÁTU JPEG NA GRAFICKÝCH KARTÁCH

ACCELERATED JPEG DECOMPRESSION ON GRAPHICS PROCESSING UNITS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
ONDŘEJ JANOŠÍK

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. PEČIVA JAN, Ph.D.

BRNO 2014

Abstrakt
Tato práce řeší problematiku dekomprese JPEG, návrh algoritmů pro provedení dekomprese na grafické kartě a jejich implementaci. Podrobněji se pak zabývá konkrétním postupem implementace včetně možných alternativ a optimalizací. Postup je popisován podle pořadí jednotlivých kroků JPEG dekomprese. Cílem této práce je snaha o redukci času, který je potřeba pro nahrání textury do paměti grafické karty použitím JPEG komprimované textury a dekomprese na straně grafické karty. Kromě úspory času je další výhodou takového přístupu snížení zátěže procesoru, což může být v některých případech žádoucí.

Abstract
This thesis addresses the issue of JPEG decompression, design of algorithms for decompression on graphics cars and their implementation. In more details, it describes specific process of implementation, including possible alternatives and optimizations. It is described in order of JPEG decompression algorithm steps. The aim of this thesis is the reduction of time consumed by uplading textures into graphics unit's memory using JPEG compressed textures and decompression od the side of graphics card. In addition to time savings, this access reduces processor load, which may be in some cases beneficial.

Klíčová slova
dekomprese, gpu, jpeg, jfif, grafická karta, OpenGl, gpgpu, computing, compute shader

Keywords
decompression, gpu, jpeg, jfif, graphics processing unit, OpenGl, gpgpu, computing, compute shader

Citace
Ondřej Janošík: Akcelerovaná dekomprese obrázků
ve formátu JPEG na grafických kartách, bakalářská práce, Brno, FIT VUT v Brně, 2014

Akcelerovaná dekomprese obrázků ve formátu JPEG na grafických kartách

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Pečivy, Ph.D.

....
Ondřej Janošík
19. května 2014

Poděkování

Tímto bych rád poděkoval Ing. Janu Pečivovi, Ph.D. za trpělivost a ochotu, kterou mi v průběhu zpracování bakalářské práce věnoval.

© Ondřej Janošík, 2014.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **2**

2 Použité technologie **3**
2.1 OpenGL . . . 3
2.1.1 Historie OpenGL . . . 3
2.1.2 OpenGL pipeline . . . 3
2.1.3 Compute shader . . . 5
2.2 JPEG . . . 6
2.2.1 Kompresní algoritmus . . . 7
2.2.2 Další varianty JPEG komprese . . . 14
2.2.3 JPEG JFIF . . . 14
2.3 Knihovna libjpeg . . . 15
2.3.1 libjpeg-turbo . . . 15

3 Implementace **16**
3.1 JPEG dekodér . . . 16
3.2 Použití fragment shaderu . . . 16
3.2.1 Převzorkování a převod barevného modelu . . . 17
3.2.2 Inverzní diskrétní kosinová transformace . . . 17
3.3 Použití compute shaderu . . . 17
3.3.1 Inverzní diskrétní kosinová transformace a dekvantizace . . . 18
3.3.2 Nahrávání nenulových koeficientů . . . 18
3.3.3 Dekódování Huffmanova kódu a zig-zag uspořádání . . . 19
3.4 Konverzní utilita . . . 22

4 Vyhodnocení **23**

5 Závěr **26**

A Obsah CD **29**

Kapitola 1

Úvod

Výpočetní výkon moderních grafických karet dnes několikanásobně převyšuje výkon procesorů. Toho se začalo využívat v různých odvětvích a místo drahých programovatelných jednotek převažují spíše levnější grafické karty. Jejich výhodou je vysoká úroveň paralelizace, čehož se využívá především u složitých výpočtů a simulací.

Tato práce se zaměřuje na návrh algoritmu pro dekompresi JPEG obrázků na grafické kartě, identifikaci problematických částí, jejich možná řešení a pokud je to možné, provedení implementace kompletní dekomprese na grafické kartě. Kompresní algoritmus JPEG se skládá z několika kroků, přičemž některé mohou být s vhodným přístupem k paralelizaci velice rychlé. Výsledkem tedy je knihovna, umožňující rychlou dekompresi JPEG obrázků pro další využití v grafických aplikacích či hrách.

V následujícím textu budu popisovat použité technologie, problematiku JPEG komprese spolu s dalšími variantami a způsob jakým probíhá uložení komprimovaných dat. Dále vysvětlím postup implementace, popíši další možnosti optimalizace a provedu srovnání s CPU či případnými GPU implementacemi. Na závěr práce provedu vyhodnocení výsledků a popíši výhody či nevýhody oproti ostatním řešením. Výslednou práci jsem zveřejnil pod open source licencí *LGPL v3.0* na adrese `https://sourceforge.net/projects/ogljpeg/`.

Kapitola 2

Použité technologie

Během vývoje knihovny jsem využil technologie, které budou postupně popsány v této kapitole. Veškeré programy jsou psány v jazyce C, a shadery v jazyce GLSL.

2.1 OpenGL

OpenGL je v dnešní době nejrozšířenější standard pro práci s GPU [11], používané především v interaktivních 2D a 3D aplikacích. Podporuje širokou škálu operačních systémů a díky variantám jako OpenGL ES či WebGL je dostupné i na mobilních zařízeních a webových prohlížečích. Standard OpenGL je spravován skupinou Khronos, která byla v roce 2000 založena předními společnostmi zabývající se multimédii za účelem vytvoření otevřených standardizovaných API[10].

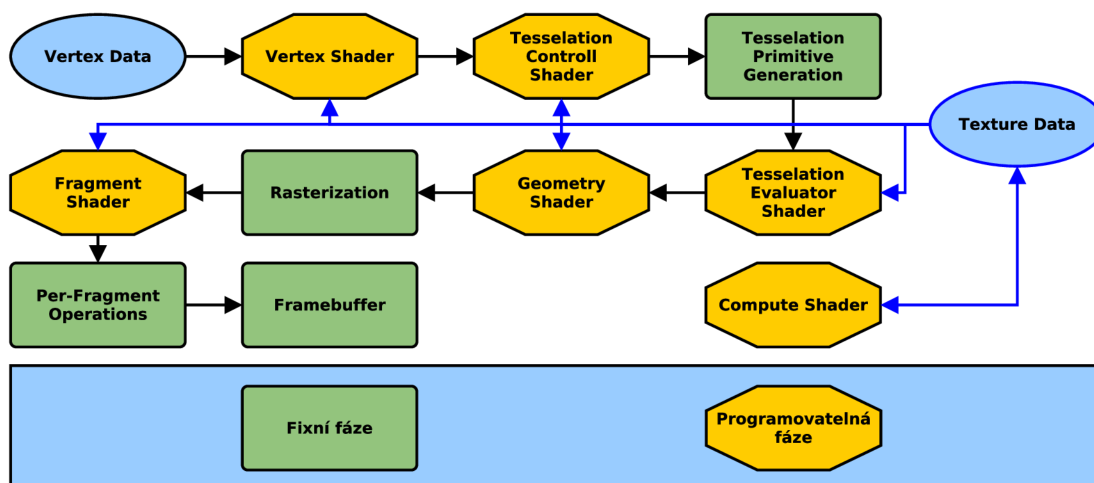
2.1.1 Historie OpenGL

První verze OpenGL byla vydána roku 1992 jako otevřená alternativa k IrisGL, což bylo proprietární grafické API na pracovních stanicích Silicon Graphics. Ačkoliv bylo zpočátku OpenGL podobné IrisGL, nedostatek formální specifikace a testů shody způsobil nevhodnost IrisGL pro větší rozšíření [1].

OpenGL prošlo mnoha revizemi, které byly ve většině případů dílčí dodatky, kdy byly rozšíření k základnímu API začleněny do jádra API. Asi nejvýznamnější změna přišla s OpenGL 2.0, kdy byl představen jazyk pro popis shaderů *OpenGL Shading Language* známý pod zkratkou GLSL. Tento jazyk podobný jazyku C umožňoval naprogramování vertex shaderu a fragment shaderu OpenGL pipeline. S dvouletým odstupem přišla verze 2.1, která příliš mnoho změn nepřinesla. V roce 2008 byla vydána verze 3.0. Jednalo se o vůbec první úpravu API v takovém rozsahu. Bylo třeba přepracovat způsob, jakým OpenGL pracuje a to si vyžádalo zásadní změny API. Během těchto úprav došlo k několika problémům. Výsledkem bylo téměř roční zpoždění oproti původnímu plánu. S touto verzí byl zaveden koncept zastarávání, kdy byly některé vlastnosti označeny jako zastaralé a určeny k odstranění v dalších verzích. Aktuální verze OpenGL specifikace je 4.4 [3].

2.1.2 OpenGL pipeline

OpenGL pipeline se skládá z několika fází, které jsou zřetězeny za sebou. Výstupní data jedné fáze jsou tedy vstupními daty fáze následující. Zjednodušené zobrazení OpenGL pipeline pro verzi 4.0 je na obrázku 2.1.



Obrázek 2.1: Zjednodušené zobrazení OpenGL pipeline

Vertex shader

Vertex shader je první programovatelná fáze OpenGL pipeline. Provádí operace nad daty obsahující souřadnice vrcholů geometrických primitiv. Jedná se hlavně o transformaci pomocí transformačních matic, především o modelovou matici, pohledovou matici a projekční matici.

Tesselace

Tesselace je volitelná fáze OpenGL pipeline, která následuje po vertex shaderu. Skládá se celkem ze tří dalších fází, z nichž je jedna fixní a dvě programovatelné. Pomocí tesselace dochází k navýšení míry detailů geometrie bez potřeby zpracování velkého objemu dat procesorem a jejich následného přenosu do paměti GPU.

Tesselation control shader První fází tesselace je Tesselation control shader (dále jen TCS). Jedná se o volitelnou část pipeline, která je přeskočena, pokud není naprogramována. Vstupem TCS je speciální primitiv *patch*. Patch je univerzální primitivum, u kterého každý n vrchol je nové primitivum. TCS rozhoduje o míře tesselace daného primitiva. Může k primitivu přidat vrcholy a také je ubrat, ale v této fázi nelze přímo primitivum zahodit.

Tesselation primitive generation Jedná se o fixní fázi tesselace, která zpracovává výstup TCS, pouze pokud je aktivní Tesselation evaluation shader (dále jen TES). V takovém případě vezme patch primitivum a vytvoří z nich novou množinu základních geometrických primitiv (body, přímky, nebo trojúhelníky). Tyto nová primitiva jsou vytvořeny dělením původních primitiv na základě parametrů nastavených v TCS, pokud je aktivní, nebo podle výchozích hodnot.

Tesselation evaluation shader TES fáze ze souřadnic každého vrcholu primitiva vytvořeného v předchozí fázi a vygeneruje vrchol s danou pozicí a atributy. V TES je možné přistoupit k jakémukoliv vrcholu patch primitiva, které bylo vytvořeno TCS (pokud je aktivní), nebo je přímo výstupem vertex shaderu. TES je volitelná fáze pipeline.

Geometry shader

Geometry shader (dále jen GS) je poslední volitelná fáze pipeline, která podobně jako fáze tesselace upravuje geometrii objektů. Mezi GS a tesselací jsou však jisté rozdíly. Vstupními daty GS jsou jednotlivá primitiva a na rozdíl od tesselace je lze i zcela zahodit. Následně jsou z těchto dat vytvořeny nová primitiva, které mohou být i jiného typu, než primitiva vstupní.

Rasterizace

Rasterizace je proces, během kterého dochází ke konverzi geometrických primitiv na 2D obraz. Každý bod takového obrazu obsahuje informace jako barva a hloubka. Spolu s dalšími informacemi tvoří fragment, který je dále zpracován fragment shaderem.

Fragment shader

Fragment shader zpracovává fragmenty a produkuje sadu barev a hloubkovou hodnotu. Tyto hodnoty jsou následně zapsány do framebufferu a dále pak použity jako textura či přímo prezentovány uživateli.

2.1.3 Compute shader

Compute shader je programovatelný shader, který slouží primárně pro výpočet složitých paralelizovatelných algoritmů. Vzhledem ke své povaze a odlišném účelu je tento shader umístěn zcela mimo OpenGL pipeline. Na rozdíl od ostatních shaderů nemá žádné předem definované vstupy či výstupu a záleží tedy pouze na tom, jak je daný shader naprogramovaný. Pokud je potřeba načíst data, lze to provést čtením z textury, nebo z shader storage bufferu. Stejně tak pro výstup dat je potřeba vypočítané hodnoty do textury či shader storage bufferu zapsat.

Výpočetní prostor

Výpočetní prostor compute shaderu se dělí na *pracovní skupiny* (*work groups*), což jsou nejmenší, uživatelem spustitelné jednotky. Prostor těchto skupin je tří-dimenzionální a jeho rozměry určuje uživatel. Minimální rozměr každé z dimenzí je 1, takže je možné spustit i 2D či 1D pracovní skupiny. To se hodí pro například pro zpracování obrazových dat, nebo lineárních polí.

Pracovní skupina má dále svůj lokální 3D prostor s definovatelnými rozměry. Právě velikost pracovní skupiny ovlivňuje míru paralelizace a i výkonnost shaderu. Podle rozměrů pracovní skupiny se spustí počet vláken, které provádějí výpočet souběžně. Dle OpenGL specifikace jsou požadavky na podporovaný minimální rozměr 1024 vláken v `X` a `Y` osách, zatímco v ose `Z` je to pouhých 64 vláken. Limit na konkrétním zařízení lze získat pomocí vhodné varianty `glGet` v kombinaci s konstantou `GL_MAX_COMPUTE_WORK_GROUP_SIZE`. Dalším omezením je celkový počet vláken ve skupině. Jedná se tedy o produkt hodnot ve všech osách a minimální podporovaný rozměr zde musí být 1024. Konkrétní hodnotu získáme pomocí `GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS`. Posledním omezením je celkový počet pracovních skupin. Minimální požadovaná hodnota je 65535 ve všech osách, takže zde je dostatek prostoru na většinu výpočetních úloh. Pro získání přesné hodnoty slouží `GL_MAX_COMPUTE_WORK_GROUP_COUNT`.

Optimální velikost pracovní skupiny

Grafické karty mají obecně vyšší počet výpočetních jader, v některých případech i tisíce. Většinou se však jedná o desítky či stovky. Při běhu compute shaderu obsadí každá výpočetní skupina jedno výpočetní jádro. Výpočetní jádro je vlastně SIMD[1] procesor, který zpracovává jednou instrukcí větší množství dat. U grafických procesorů je vhodné aby velikost pracovní skupiny byla násobkem 32, v ostatních případech dochází k neefektivnímu využívání SIMD procesoru.

Sdílená paměť

V rámci své skupiny mohou vlákna komunikovat pomocí sdílené paměti. OpenGL specifikuje minimální velikost sdílené paměti 32kB pro jednu skupinu. Pro zjištění této hodnoty na konkrétním zařízení lze využít `GL_MAX_COMPUTE_SHARED_MEMORY_SIZE`.

Paměť je rozdělena po 4 bajtových slovech. Každé slovo náleží jedné paměťové bance, kterých se na moderních grafických kartách nachází 32. Které paměťové bance náleží dané slovo zjistíme tím, že si vypočítáme zbytek po celočíselném dělení index slova počtem paměťových bank. Je nutné tyto slova indexovat od nuly. Nulté slovo tedy náleží do nulté paměťové banky, první slovo náleží do první paměťové banky atd. až po jednatřicáté slovo, které náleží do jednatřicáté banky. Dvaatřicáté slovo pak opět náleží do nulté banky [13].

Pokud vlákna najednou přistupují k různým slovům v jedné bance, pak dochází ke konfliktu paměťových bank. Podle počtu různých slov, ke kterým vlákna přistupují se jedná o `N`-cestný konflikt bank. V případě konfliktu je nutné čtení z paměti serializovat a to má negativní vliv na výkon.

Když ale všechna vlákna přistupují ke stejnému slovu, pak se hodnota načte pouze. Jedná se o techniku *broadcast*. Pokud pouze několik vláken přistupuje ke stejnému slovu pak se jedná o techniku *multicast* [9].

Větvení a iterace

Vzhledem ke způsobu zpracování instrukcí na GPU mohou větvení a iterace značně snížit výkon algoritmu. Pokud během větvení některé z vláken prochází jinou větví než ostatní vlákna, je potřeba vykonat instrukce v obou větvích, přičemž část vláken, která prochází druhou větví, je během této doby neaktivní. O stejný problém se jedná, pokud je v případě cyklu počet iterací každého vlákna jiný [6]. Je tedy vhodné se pokud možno vyhnout větvením a iteracím specifickým pro konkrétní vlákna.

2.2 JPEG

JPEG[2] je původně metoda ztrátové komprese standardizovaná normou ISO. Ačkoliv se jedná o kompresní metodu, tak dnes často bývá zaměňována se souborovým formátem JPEG JFIF[3], který slouží právě pro uchování a přenos obrazu zkomprimovaného metodou JPEG [8].

[1]SIMD – Single Instruction Multiple Data
[2]JPEG – Joint Photographic Experts Group
[3]JFIF – JPEG File Interchange Format

2.2.1 Kompresní algoritmus

Základní kompresní algoritmus JPEG se skládá z několika kroků.

1. Převod do barevného modelu YCbCr
2. Podvzorkování barvonosných složek
3. Rozdělení jednotlivých složek do bloků 8×8
4. Dvourozměrná diskrétní kosinová transformace (2D DCT)
5. Kvantizace
6. Zig-zag kódování
7. Run-length kódování
8. Huffmanovo kódování

Při dekompresi se akorát celý postup obrátí. Jedná se o původní baseline algoritmus. Existují však další varianty, které jsou popsány v kapitole 2.2.2.

Konverze barevného modelu

Barevný model YCbCr se skládá ze 3 složek.

Y Luminance[4]

Cb Blue chrominance[5]

Cr Red chrominance

Luminační složka má hodnoty v intervalu $< 0, 255 >$, hodnoty chrominance se běžně vyskytují v intervalu $< -128, 127 >$, v tomto případě jsou ale dodatečně posunuty do stejného intervalu jako luminační složka. Samotný převod je vyjma zaokrouhlovacích chyb bezeztrátový a hodnoty pro potřeby JPEG komprese lze vypočítat podle vzorců 2.1.

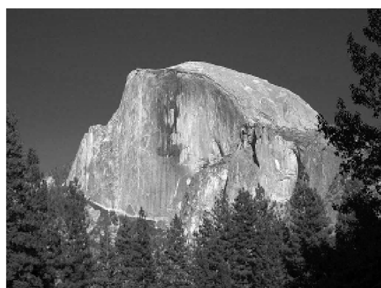
$$\begin{aligned} Y &= 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\ C_b &= -0.1687 \cdot R - 0.3313 \cdot G + 0.5 \cdot B + 128 \\ C_r &= 0.5 \cdot R - 0.4187 \cdot G - 0.0813 \cdot B + 128 \end{aligned} \tag{2.1}$$

Zpětný převod z YcbCr na RGB probíhá dle vzorců 2.2.

$$\begin{aligned} R &= Y + 1.402 \cdot (C_r - 128) \\ G &= Y - 0.34414 \cdot (C_b - 128) - 0.71414 \cdot (C_r - 128) \\ B &= Y + 1.772 \cdot (C_b - 128) \end{aligned} \tag{2.2}$$

[4] Luminance – jas
[5] Chrominance – barevná sytost

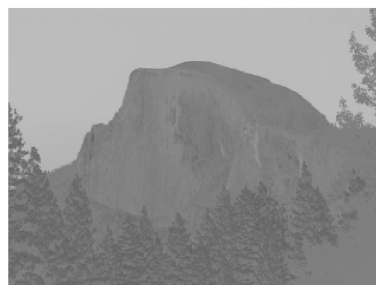
Původní RGB obrázek



R složka RGB modelu

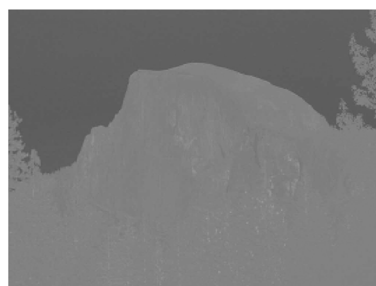
Y složka YCbCr modelu

G složka RGB modelu



Cb složka YCbCr modelu

B složka RGB modelu



Cr složka YCbCr modelu

Obrázek 2.2: Srovnání barevných modelů RGB a YCbCr.

Podvzorkování barvonosných složek

Ze sady obrázků 2.2 lze vyvodit, že nejvíce informací o obsahu obrazu nese právě luminační složka, zatímco chrominační složky obsahují informací podstatně méně. Spolu s faktem, že lidské oko je nejcitlivější právě na změny jasu, se využívá tato znalost k redukci objemu dat tím, že dojde k podvzorkování barvonosných složek. Obě barvonosné složky jsou vždy podvzorkovány stejně. Možnosti vzorkování jsou tyto:

Plné
: Je zachováno plné rozlišení barvonosných složek.

Poloviční
: Jsou zprůměrovány hodnoty sousedních pixelů. Redukce objemu na 66%.
Poloviční podvzorkování může být horizontální, nebo vertikální.

Čtvrtinové
: Jsou zprůměrovány čtveřice 2×2 pixelů. Redukce objemu na 50%.

Diskrétní kosinová transformace

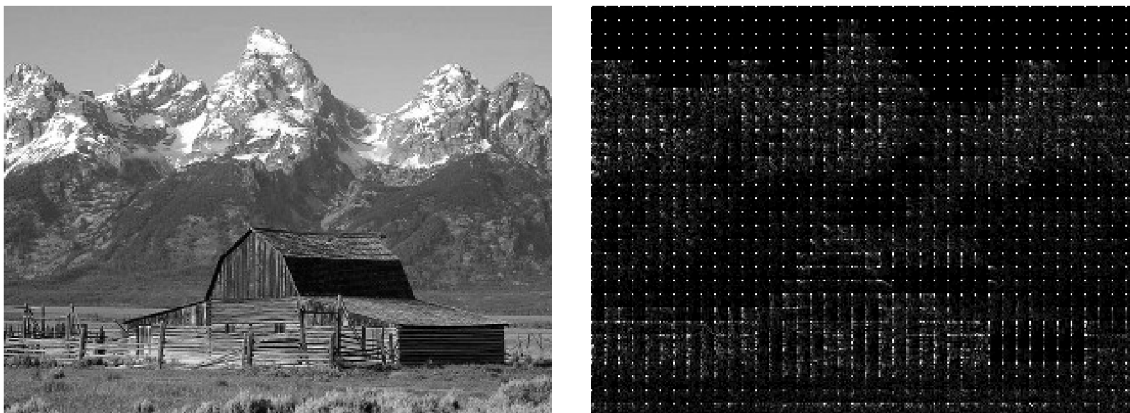
Po rozdělení jednotlivých barevných složek na bloky o rozměrech 8×8 je na každý tento blok aplikován 2D DCT algoritmus. Tato transformace pracuje s faktem, že mezi je sousedními pixely jistá úroveň korelace a proto má výhodnější využití v případě fotografií, kde se běžně nevyskytují ostré přechody, ale naopak obsahují spíše oblasti vyplněné podobnou barvou. Algoritmus DCT tedy provádí dekorelaci obrazu, jinými slovy jej převádí z prostorové domény do domény frekvenční [7]. Podobně funguje i diskrétní Fourierova transformace, ale pro účely JPEG je DCT vhodnější, jelikož produkuje pouze reálné koeficienty a nejvíce energie je koncentrováno na nižších frekvencích. Toho se využívá v dalších krocích komprese.

Za předpokladu, že S je původní obraz v prostorové doméně a F je transformovaný obraz ve frekvenční doméně, pak platí 2.3.

$$
\begin{aligned}
C_u, C_v &= \begin{cases} \frac{1}{\sqrt{2}} & \text{pro } u, v = 0 \\ 1 & \text{jinak} \end{cases} \\
\text{2D DCT:} \qquad F_{uv} &= \frac{1}{4} C_u C_v \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} S_{xy}\, cos\left(u\pi \frac{2x+1}{2N}\right) cos\left(v\pi \frac{2x+1}{2N}\right) \\
\text{2D IDCT:} \qquad S_{uv} &= \frac{1}{4} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} C_u C_v F_{xy}\, cos\left(u\pi \frac{2x+1}{2N}\right) cos\left(v\pi \frac{2x+1}{2N}\right)
\end{aligned}
\tag{2.3}
$$

Výhodou DCT algoritmu je jeho vlastnost separability, takže lze místo 2D DCT se složitostí $O(n^4)$ aplikovat 1D DCT vertikálně pro každý sloupec a horizontálně pro každý řádek, čímž dosáhneme složitosti $O(n^3)$.

První, tak zvaný DC koeficient se souřadnicemi $(0,0)$ je stejnosměrná složka signálu a určuje průměrnou hodnotu vzorku. Je možné jí využít k rychlejšímu vytvoření malého náhledu na obrázek. Ostatní hodnoty jsou tak zvané AC koeficienty a reprezentují zastoupení frekvencí ve vzorku.



Obrázek 2.3: Obrázek o rozměrech 320×240px před a po DCT aplikované na bloky 8×8px.

Příklad výstupu DCT algoritmu je zobrazen na obrázku 2.3. Na tomto obrázku je možné pozorovat , že se hodnoty nejvíce koncentrují právě v levém horním rohu jednotlivých bloků, který zastupuje nižší frekvence. Nejsvětlejší body jsou právě DC koeficienty.

Stejně jako existuje rychlá Fourierova transformace, tak i IDCT má své optimalizované varianty. Mezi nejrychlejší patří *Loeffler, Ligtenberg, Moschytz* (LLM) metoda, používající 28 operací sčítání a 11 operací násobení pro transformaci 8 prvků. O něco rychlejší je pak metoda *Arai, Agui, Nakajima* (AAN), které pro stejný počet prvků stačí 29 operací sčítání a jen 5 operací násobení. Algoritmus lze nadále urychlit použitím fixed-point[6] aritmetiky[14].

Kvantizace

Proces kvantizace je způsob redukce větší množiny hodnot na menší množinu hodnot. Jedná se tedy o ztrátový proces. V případě JPEG komprese se kvantizace provádí celočíselným dělením matice koeficientů DCT s kvantizační maticí po prvcích. Ta je navržena tak, aby z obrazu odstranila vysoké frekvence, protože lidské oko není příliš citlivé na změny s vysokou frekvencí, a tyto informace mohou být odstraněny, aniž by došlo k vytvoření zjevných vizuálních artefaktů. Artefakty vzniklé kvantizací se v obraze vyskytují převážně v místech s ostrými přechody.

Hodnoty v kvantizační matici (2.4) jsou určeny mírou nastavení kvality a to zároveň ovlivňuje míru komprese. Všeobecně by se však dalo říci, že čím vyšší jsou souřadnice buňky matice, tím vyšší je také hodnota této buňky. Celočíselným dělením matice DCT koeficientů (2.5) následně vznikají v oblasti vysokých frekvencí nulové hodnoty a celkově se rozsah hodnot v matici redukuje (2.6).

JPEG algoritmus používá dvě kvantizační matice. Jednu pro luminační složku a druhou pro složky chrominační. Pro účely dekomprese je nutno tyto matice přenášet zároveň s komprimovanými daty.

[6] Fixed-point aritmetika, tedy aritmetika s pevnou desetinnou čárkou využívá pouze celočíselných instrukcí a bitových posunů, na rozdíl od floating-point aritmetiky nevyžaduje přítomnost aritmetického koprocesoru a díky tomu je rychlejší. Dosahuje však nižší přesnosti výpočtu.

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 124 & 140 & 151 & 161 \\ 12 & 12 & 14 & 19 & 126 & 158 & 160 & 155 \\ 14 & 13 & 16 & 24 & 140 & 157 & 169 & 156 \\ 14 & 17 & 22 & 29 & 151 & 187 & 180 & 162 \\ 18 & 22 & 37 & 56 & 168 & 109 & 103 & 177 \\ 24 & 35 & 55 & 64 & 181 & 104 & 113 & 192 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 199 \end{bmatrix} \tag{2.4}$$

$$C = \begin{bmatrix} 2102 & 1576 & 1432 & 1103 & 902 & 887 & -635 & -334 \\ 731 & -186 & 5 & -289 & 177 & -393 & 422 & -137 \\ 216 & -96 & -56 & 66 & -239 & 310 & -264 & 146 \\ -32 & 98 & -15 & -32 & 90 & -92 & 237 & -108 \\ 149 & -60 & 121 & 13 & 74 & 70 & -78 & 74 \\ 22 & 14 & -51 & 87 & -24 & -16 & 14 & -22 \\ 189 & -55 & -4 & 6 & 44 & 11 & -7 & 32 \\ 44 & -29 & -29 & -21 & -19 & 12 & 16 & 19 \end{bmatrix} \tag{2.5}$$

$$\text{round}\left(\frac{C}{Q}\right) = \begin{bmatrix} 131 & 143 & 143 & 69 & 7 & 6 & -4 & -2 \\ 61 & -16 & 0 & -15 & 1 & -2 & 3 & -1 \\ 15 & -7 & -4 & 3 & -2 & 2 & -2 & 1 \\ -2 & 6 & -1 & -1 & 1 & 0 & 1 & -1 \\ 8 & -3 & 3 & 0 & 0 & 1 & -1 & 0 \\ 1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 4 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.6}$$

Zig-zag kódování

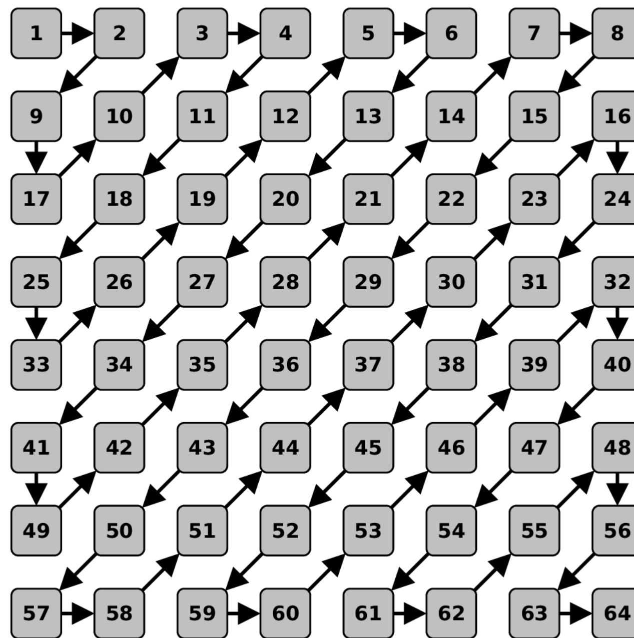
Po kvantizaci je pro další potřeby komprese data v matici vhodně přeuspořádat do lineární podoby a právě to je účelem kódovací metody zig-zag. Ta prochází matici způsobem, který je zobrazen na obrázku 2.4. Výstupem tohoto kódování je zig-zag sekvence s nízkou mírou entropie, protože ve většině případů obsahuje dlouhé řetězce nul, které vznikly v pravé spodní oblasti matice během kvantizace.

Run-length kódování

Run-length kódování (dále jen RLE) je bezeztrátová kompresní metoda, vhodná pro kompresi dlouhých sekvencí stejných hodnot. Její princip spočívá v tom, že kóduje posloupnosti stejných hodnot do dvojice *délka sekvence* a *hodnota*. Její nevýhoda je, že pro nevhodná vstupní data může být objem výstupních dat až dvojnásobný. Právě proto se touto metodou v případě JPEG komprese kódují pouze nulové koeficienty. V případě JPEG formátu je run-length kódování svým způsobem součástí Huffmanova kódu. Tento fakt je podrobněji popsán v kapitole 2.2.1

Huffmanovo kódování

Huffmanovo kódování je algoritmus používaný pro bezeztrátovou kompresi. Využívá kódu proměnné délky pro zakódování symbolu na základě jeho četnosti v souboru. K reprezentaci



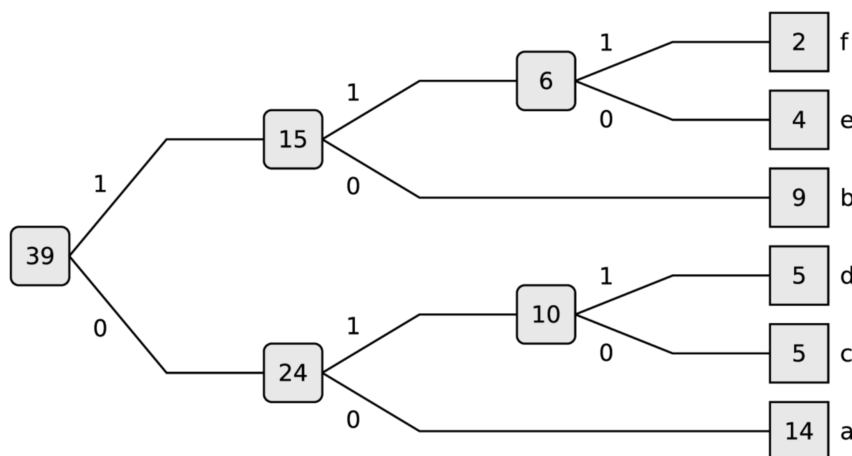
Obrázek 2.4: Znázornění průchodu maticí při zig–zag kódování.

zakódovaného symbolu je použita metoda, která vytváří tak zvaný prefixový kód. Jedná se o takový bitový řetězec, který není prefixem (předponou) řetězce reprezentující jiný symbol. Nejčastěji se vyskytující symboly jsou ve výsledku reprezentovány kratšími bitovými řetězci, zatímco symboly objevující se zřídka jsou reprezentovány řetězci delšími.

Algoritmus Algoritmus Huffmanova kódování probíhá ve třech fázích.

1. Statistická analýza dat
2. Vytvoření binárního kódovacího stromu
3. Zakódování dat

Při statistické analýze se průchodem vytvoří statistika výskytu jednotlivých symbolů ve vstupních datech. Za pomocí této statistiky vytvoří binární kódovací strom. Nejjednodušší algoritmus využívá prioritní frontu. Symbol s nejvyšším počtem výskytů má nejnižší prioritu.



Obrázek 2.5: Příklad binárního stromu sestaveného za účelem vytvoření Huffmanova kódu.

Symbol	Kód
a	11
b	01
c	101
d	100
e	001
f	000

Tabulka 2.1: Huffmanův kód pro jednotlivé symboly.

1. Pro každý znak ve statistice vytvoř listový uzel a přidej jej do fronty.
2. Dokud je ve frontě více něž jeden uzel:
   a. Vyjmi dva uzly s nejnižším počtem výskytů
   b. Vytvoř vnitřní uzel, který má za potomky dva vyjmuté uzly a počet výskytů je roven součtu výskytů těchto dvou uzlů
   c. Zařaď nově vytvořený uzel do fronty
3. Poslední uzel ve frontě je kořenem stromu

Je běžnou konvencí, že levé větve mají bitovou hodnotu `0` a pravé větve hodnotu `1`. Bitovou reprezentaci znaku lze získat průchodem skrze strom od znaku do kořene stromu, přičemž si při každém přechodu je třeba si zaznamenat hodnotu větve.

Na obrázku 2.5 je znázorněn binární strom pro data obsahující symboly $\{a; b; c; d; e; f\}$ s počtem výskytů $\{14; 9; 5; 5; 4; 2\}$. Tomuto stromu odpovídá Huffmanův kód v tabulce 2.1.

Huffmanovo kódování v JPEG kompresi V případě JPEG komprese je Huffmanovo kódování ještě trochu složitější. Po dekódování symbolu získáme 8 bitovou hodnotu. Pro přehlednost si tuto hodnotu můžeme představit jako následující řetězec bitů `RRRRSSSS`. Hodnota `SSSS`, tedy 4 nejméně významné bity určuje tzv. diferenční kategorii, kterých je pro DC koeficienty 11 a pro AC koeficienty pouze 10. Bavíme se zde však pouze v případě, kdy je

použita 8 bitová přesnost. JPEG totiž nabízí také možnost uložení s 12 bitovou přesností, tato možnost se však příliš nepoužívá. Diferenční kategorie zároveň určuje následující počet bitů, které je nutno dále načíst. Pokud je tato hodnota nulová, pak není třeba nic dále načítat a výsledná hodnota je rovněž nula. Pokud však nulová není, pak je třeba načíst další bity, které interpretujeme jako další hodnotu `V`. Pokud platí, že `V < (V << (SSSS-1))`, pak se jedná o záporné číslo a musíme provést následující operaci `V += ((-1)<< V) + 1`.

Co se týče DC koeficientů, tak zde je hodnota `RRRR` ignorována. Pokud se jedná o AC koeficienty, pak je hodnota těchto bitů interpretována jako tzv. *zero run-length*, neboli délka řetězce nulových koeficientů, po kterém následuje právě dekódovaný koeficient. Speciálním případem je pak *end of block* symbol, což je případ, kdy je hodnota `RRRRSSSS` rovna 0.

2.2.2 Další varianty JPEG komprese

JPEG nabízí další možnosti komprese. Většinou se jedná o nahrazení části algoritmu účinnější metodou. Huffmanovo kódování je možno nahradit například kódováním aritmetickým, které vykazuje o 5-10% vyšší účinnost komprese. Dále se nabízí možnost využít vlnkové transformace, čehož využívá novější formát JPEG 2000, který měl nahradit starší JPEG, to se však nestalo. Také existuje varianta JPEG-LS, která umožňuje bezztrátovou kompresi. Tu rovněž nabízí i JPEG 2000.

Zatímco baseline varianta JPEG komprese není patenty zatížena. Na některé volitelné možnosti JPEG jako jsou aritmetické kódování, nebo hierarchické ukládání se již vztahují jisté patenty a z toho důvodu se příliš nepoužívají.

2.2.3 JPEG JFIF

JPEG JFIF je souborový formát pro přenos obrazu komprimovaného JPEG algoritmem vytvořený skupinou IJG[7]. Dnes patří mezi nejpoužívanější formáty pro ukládání obrazu.

Od vydání první implementace prošel tento formát značným vývojem a nabízí různé metody JPEG komprese. Například progresivní ukládání přeuspořádá informace, aby po stažení části souboru byl namísto malé části obrázku vidět obrázek v nižší kvalitě, ale celý. Tento rozdíl je zobrazen na obrázku 2.6.

Struktura souboru

Struktura souborů JPEG JFIF je navržena tak, aby byly hodnoty zarovnány na 16 bitů (výjimkou je část obsahující Huffmanův kód). 16 bitová hodnota je vždy uložena způsobem známým jako *big-endian*[8]. Důležitou součástí tohoto formátu jsou pak tzv. 2 bajtové *markery*, které začínají bajtem s hodnotou `0xFF` a označují různé segmenty souboru. Druhý bajt pak určuje konkrétní typ markeru. Protože je hodnota `0xFF` použita jako marker, je veškeré tyto hodnoty escapovat jako `0xFFFF`. Dalším speciálním případem je pak sekvence `0xFF00`, která se používá pouze během Huffmanova či aritmetického kódování. Tato technika je označována jako *byte stuffing* a takové sekvence jsou ignorovány.

Markery s rozsahem hodnoty `0xFFC0`–`0xFFCF` jsou nazývány *Start of Frame*, a určují způsob komprese a použité kódování. V tomto projektu se budeme zabývat pouze obrázky s hodnotou tohoto markeru `0xFFC0`, tedy baseline uspořádání dat a Huffmanovo kódování. Další důležité markery jsou `0xFFC4` (*Define Huffman Table(s)*), `0XFFDB` (*Define quantization*

[7]IJG – Independend JPEG Group

[8]Big-endian (big end first) je způsob uložení hodnot v paměti takovým způsobem, že se významnější bajty nachází na nižší adrese než bajty méně významné.

Baseline Progresivní

Obrázek 2.6: Rozdíly mezi baseline a progresivním formátem JPEG JFIF při částečném stažení souboru.

table(s)), `0xFFD8` (*Start of image*), `0xFFDA` (*Start of scan*) a `0xFFDD` (*Define restart interval*), které označují segmenty obsahující potřebná data pro dekódování jako Huffmanovy a kvantizační tabulky a pak samotná data.

Pro tuto práci je velice důležitý segment definující velikost restart intervalu. Pokud je délka restart intervalu nenulová, tak lze využí paralelního dekódování Huffmanova kódu. V tom případě jsou ještě důležité samotné *restart markery*, které jsou v rozsahu `0xFFD0`–`0xFFD7` a periodicky se opakují. Právě tyto markery umožňují najít v kódovaných datech začátek bloku a kód tak rozdělit pro využití paralelizace.

2.3 Knihovna libjpeg

Jedná se o volně dostupnou knihovnu[9], napsanou v jazyce C, která slouží ke kompresi a dekompresi JPEG souborů a zároveň umožňuje konverzi mezi různými formáty. Tuto knihovnu spravuje a distribuuje IJG[10]. Aktuální verze této knihovny (*libjpeg v9*) by tedy měla implementovat všechny formáty JPEG zahrnuté ve standardu ITU-T.81 [4].

2.3.1 libjpeg-turbo

Tato knihovna[11] je odnoží knihovny *libjpeg*, která využívá SIMD instrukcí pro urychlení komprese a dekomprese JPEG. Díky tomu vykazuje 2-4 násobné zrychlení[2] oproti knihovně *libjpeg*.

libjpeg-turbo je kompatibilní s tradičním *libjpeg API*, a navíc nabízí i jednodušší *TurboJPEG API*. Zároveň podporuje téměř všechny funkce jako *libjpeg* ovšem s výjimkou funkcí spojených s nestandardním bezeztrátovým formátem *SmartScale*.

[9] Stránky projektu *libjpeg*: `http://libjpeg.sourceforge.net/`
[10] Stránky IJG: `http://www.ijg.org/`
[11] Stránky projektu *libjpeg-turbo*: `http://www.libjpeg-turbo.org/`

Kapitola 3

Implementace

Během implementace programu jsem postupoval od konce algoritmu pro JPEG dekompresi. Díky zvolenému postupu nebylo třeba implementovat celou dekompresi najednou, nýbrž stačilo postupně přesouvat implementaci z procesoru na grafickou kartu. Výhodou byla jistota, že předchozí kroky dekomprese fungují správně a stačilo se zabývát pouze právě implementovanou fází. U některých z kroků jsem experimentoval s různými přístupy řešení daného problému. Některé varianty se ukázaly jako vhodné a jiné byly naprosto neefektivní a dobu dekomprese naopak navýšily.

3.1 JPEG dekodér

Na počátku práce jsem si vystačil s možnostmi, které poskytovala knihovna *libjpeg*. Ta umožňovala jak dekompresi celého obrázku včetně převodu barevného modelu, tak přístup k pouhým DCT koeficientům a kvantizačním tabulkám. V pozdější fázi možnosti této knihovny bohužel přestávaly dostačovat a bylo třeba přistoupit k implementaci vlastního CPU dekodéru nad kterým bych měl plnou kontrolu.

Vzhledem k široké škále možností JPEG komprese jsem se rozhodl použít pouze nejzákladnější baseline formát který je popsán v dokumentu ITU-T.81. Kód je částečně inspirován implementací knihovny *libjpeg*, velice mi však pomohla knihovna *nanojpeg*[1], díky které jsem byl schopen identifikovat své chyby a také použít některé optimalizace.

Během implementace dekodéru jsem hlouběji pochopil strukturu JFIF souboru a lépe jsem porozuměl jednotlivým krokům JPEG algoritmu. Nově nabyté poznatky mi rozšířily možnosti řešení daných problému a zjednodušily jejich následnou implementaci na GPU.

3.2 Použití fragment shaderu

Ze začátku se nejvýhodněji jevil fragment shader vzhledem k možnostem kompatibility s širokou škálou zařízení, která podporovala pouze starší verze OpenGL. Fragment shader v kombinaci se zápisem do framebufferu je pro operace s rastrem jako dělaný. Později se však ukázalo, že fragment shader je pro složitější operace vzhledem k nízké míře kontroly a možnosti zapisovat pouze do výstupní textury nevhodný.

[1] Domovská stránka projektu *nanojpeg*: `http://keyj.emphy.de/nanojpeg/`

3.2.1 Převzorkování a převod barevného modelu

Mezi poslední kroky JPEG dekomprese patří případné nadvzorkování barvonosných složek a následný převod barevného modelu. Pokud budu uvažovat pouze vertikální, či horizontální podvzorkování, tak dochází k redukci objemu dat, které je nutné přenést přes sběrnici na 66%. V případě podvzorkování v obou směrech se jedná o 50% redukci (viz. 2.2.1).

Dalším pozitivem je fakt, že fragment shader je velice vhodný pro bitmapové operace a tedy i převzorkování či převod barevného modelu. Jelikož se během procesu nadvzorkování textury zvětšují na dvojnásobný rozměr v dané ose, vystačíme si s proximální interpolací. Jedná se o prosté zkopírování bodu. I přesto je pro dosažení lépe vypadající výsledků vhodnější alespoň lineární interpolace. V některých případech se totiž v obraze s proximální interpolací mohou vyskytovat mdlé barvy. Pokud ale přihlédnu k tomu, že se jedná o textury pro použití v grafických aplikacích pak bych očekával, že budou uloženy bez podvzorkování. Vhodnější proces nadvzorkování by mohl být předmětem dalšího rozšíření knihovny.

3.2.2 Inverzní diskrétní kosinová transformace

Následující krokem JPEG dekomprese je IDCT. Nejjednodušší implementace se skládá z čtveřice vnořených cyklů což samo o sobě vypovídá o vysoké časové složitosti. Na grafických kartách jsou většinou kvůli způsobu zpracování SIMD větvení a iterace zatíženy určitou penalizací. To však pouze v případě, že všechna vlákna jedné výpočetní jednotky neputují stejnou cestou, což ovšem není tento případ. Navíc se jedná o iterace s konstantní velikostí, která je známá již v době kompilace, takové smyčky by měl kompilátor rozbalit. Zde pouze vyvstává otázka, zda to provede v případě několika vnořených iterací.

Jak se však dalo očekávat algoritmus s tak vysokou složitostí nemohl proces dekomprese urychlit ani když byl paralelizován. Proto byl nahrazen metodou výpočtu AAN IDCT pro blok o velikosti 8×8. Ačkoliv byl předchozí výpočet nahrazen výpočtem mnohem rychlejším stále byl celý algoritmus pomalejší než jeho předchozí varianta. Toto zpomalení způsobují faktory popsané v následujících odstavcích.

Samotný AAN algoritmus nepočítá jednotlivé hodnoty, ale celý blok. Nevýhodou fragment shaderu je to, že není možné při jednom průchodu zapsat hodnoty texelů na více místech, ale pouze do místa, pro které byl fragment shader spuštěn. Jedním průchodem fragment shaderem tedy sice získáme 64 hodnot, ale zachovat je možné pouze jednu. Zbylých 63 hodnot je zahozeno a vzniká zde vysoká redundance výpočtů. Dále je pro jeden blok nutno spustit 64 fragment shaderů, to znamená, že z celkového počtu $64 \times 64 = 4096$ je využit pouze 64 hodnot a zbylých $64 * 63 = 4032$ hodnot je zahozeno. Efektivní využití výpočtu je $64 \div 4096 = 1.5625\%$.

Protože DCT koeficienty je nutno uložit minimálně na dvou bajtech, dostáváme se k problému dvojnásobného objemu dat, který je třeba přenést oproti případu, kdy nahráváme přímo RGB texturu. Většina bloků DCT koeficientů má však vlastnost, že značná část těchto koeficientů je nulová. Vyskytuje se zde tedy možnost nahrát pouze nenulové koeficienty, tou se ale budeme zabývat až v kapitole 3.3.2.

3.3 Použití compute shaderu

V předchozí kapitole jsme si ukázali, že pro další postup je fragment shader z důvodu nemožnosti preciznější kontroly nevhodný. Je tedy jasné, že potřebujeme lépe ovládat jednotlivé výpočetní jednotky a jejich vlákna. Právě v takovém případě nám docela dobře poslouží

compute shader, který umožňuje výše zmíněné a navíc je možné zapisovat do více textur na více různých míst při jednom průchodu.

3.3.1 Inverzní diskrétní kosinová transformace a dekvantizace

S využitím compute shaderu získáváme větší kontrolu nad výpočtem a lze tedy odstranit negativní vlastnost výpočtu IDCT, kdy bylo třeba zahazovat hodnoty. Aktuální implementace využívá floating-point aritmetiku a AAN implementaci IDCT.

Jedna skupina o velikosti $2 \times 2 \times 8$ vláken zpracuje najednou 4 bloky DCT koeficientů. Zde by stačily skupiny skládající se z 8 vláken, které by zpracovaly jeden blok. Pak by ale nebyla plně využita výpočetní síla procesoru, a proto byl výpočet rozšířen na 4 bloky, takže konečný počet vláken je 32. Výpočet probíhá pro každou složku odděleně. Konkrétní postup výpočtu bude popsán pouze pro jeden blok.

Nejprve je třeba načíst potřebná data. Každé z 8 vláken načte jeden řádek matice DCT koeficientů, příslušné hodnoty kvantizační matice a tyto hodnoty spolu vynásobí. Po dekvantizaci se na řádek aplikuje výpočet 1D IDCT. Nyní je potřeba aplikovat 1D IDCT také na sloupec, ale v různých řádcích mají ostatní vlákna a je třeba provést předání vypočtených hodnot. Využijeme toho, že v rámci jedné skupiny můžeme použít rychlou sdílenou paměť a není třeba výsledek mezi výpočtu ukládat pomalé do globální paměti. Každé vlákno tedy uloží do sdílené paměti jeden řádek mezivýsledků. Během ukládání nedochází k žádnému konfliktu bank. Během načítání bohužel pracujeme se sloupci během toho dochází ke dvou-cestnému konfliktu bank. Zde by bylo vhodné upravit přístupový vzor a tím konflikty odstranit. Díky absenci větvení je přístup do sdílené paměti synchronní a není jej třeba synchronizovat explicitně. Po výměně dat mezi vlákny provedeme 1D IDCT na sloupce a výsledky zapíšeme do textury v globální paměti.

Zde by bylo možné urychlit výpočet například užitím fixed-point aritmetiky. Jelikož jsou však floating-point operace na GPU vysoce optimalizované a mnohdy rychlejší než na CPU [12], je tedy otázkou, zda by fixed-point aritmetika dosahovala znatelně vyššího výkonu.

3.3.2 Nahrávání nenulových koeficientů

V tomto bodě jsem stále před problémem, který způsobuje znatelné zpoždění při přenosu dat po sběrnici do paměti GPU. Odstraněním nulových koeficientů je možné redukovat objem dat ve většině případů o 30–60%[2]. Bohužel existují i případy, kdy se jedná o méně než 10%. Navíc vše závisí také na kvalitě obrázků, takže se ve výsledku jedná o velice pohyblivý faktor a není možné na něj příliš spoléhat.

Nyní jsou věci poněkud komplikovanější. Díky vlastnímu dekodéru bylo možné vygenerovat data, které jsou stále v zig-zag pořadí spolu s délkou řetězce nenulových koeficientů. Negativním faktorem byla pomalejší implementace dekódování Huffmanova kódu, než obsahují knihovny *libjpeg* a *libjpeg-turbo*. Kdybych chtěl naopak využít funkce knihoven, bylo by třeba opět přeuspořádat data do zig-zag pořadí a zjistit délku řetězce nenulových koeficientů, což by opět zvýšilo potřebnou režii. Nakonec bych se pravděpodobně v obou případech dostal na přibližně stejnou časovou hranici. Co ale mělo velice negativní dopad na celkovou dobu, je velice časté volání OpenGL funkce `glTexSubImage2D`.

Existuje možnost vhodně uspořádat nenulové koeficienty a nahrát je najednou jako celou texturu, pak by ale bylo navíc potřeba dalších podpůrných dat obsahující informaci

[2]Tato informace byla získána experimentálním měřením na sérii běžně používaných fotografií

o začátku jednotlivých bloků. Jedná se opět o další režii navíc a vzhledem k velice pohyblivému poměru nulových koeficientů na konci zig-zag sekvence ku jejich celkovému počtu se již dále nevyplatí tomuto problému věnovat vyšší pozornost. Vhodnější je zaměřit se na potenciálně rychlejší alternativy.

3.3.3 Dekódování Huffmanova kódu a zig-zag uspořádání

Huffmanovo kódování a dekódování je sekvenční algoritmus. Je zde tedy jisté omezení v možnostech paralelizace. Ačkoliv má Huffmanův kód vlastnost auto korekce[5], tak není možné této vlastnosti využít. To z důvodu, že se jedná o kód s proměnlivou délkou znaku. I kdybych rozdělil kód do vzájemně přesahujících intervalů mezi jednotlivé skupiny a spoléhal na auto korekci, není zcela jisté, že by se podařilo správně detekovat začátek bloku a už vůbec ne jeho správné umístění. Tento přístup je zcela nevhodný a nespolehlivý.

Jelikož jsou ale součásti JFIF specifikace také restart markery (viz. 2.2.3), které slouží pro synchronizaci dekodéru a v případě poškození souboru, je možné rychle přeskočit poškozená data a tyto markery vyhledat. Vyhledání markerů zabere velice krátký čas a při průchodu si buduji mapovací tabulku, která později poslouží při určování začátku jednotlivých intervalů. Při spuštění je pak potřebný pouze jeden přístup do globální paměti navíc. Restart markery jsou však volitelné a pokud nejsou v souboru přítomny, pak není možné tento proces vhodně paralelizovat. Proto vznikla utilita pro bezeztrátový převod JFIF souborů do vhodného formátu. Tato utilita je dále popsána v kapitole 3.4.

Pro efektivní paralelní dekódování bylo třeba najít vhodný počet vláken pracovní skupiny a také optimální délku restart intervalu. Experimentováním se mi osvědčila hodnota 128 vláken. Během proces dekódování Huffmanova kódu totiž jednotlivá vlákna na místech větvení relativně často divergují. Tato hodnota vyvažuje tyto divergence počtem dekódovaných intervalů. Vyšší hodnoty už se příliš nevyplácí, jelikož naopak snižují počet jednotek na kterých je možno dekódování provádět.

U délky restart intervalu jsem očekával, že optimální hodnota bude především záviset na rozměrech obrazu. Menší obrazy by měly kratší restart interval zatímco obraty velkých rozměrů by měly restart interval delší. Nakonec se ovšem ukázalo, že pro výpočet nejoptimálnější hodnoty by bylo třeba předem znát počet dostupných výpočetních jednotek. Nabízí se zde možnost uvažovat například průměr dostupný ve většině grafických karet. Nejspíše by se jednalo o přibližně 400 jednotek. Dalším problémem byl výpočet, pokud bychom podělili počet MCU bloků v obrázku počtem jednotek, tak by u výkonnějších grafických karet zůstaly některé jednotky nevyužity. Nakonec jsem se rozhodl určit obecně nejvýhodnější délku restart intervalu. Prostým experimentováním jsem došel k délce intervalu 8 MCU bloků.

Binární dekódovací strom

Specifikace formátu JFIF povoluje maximální počet 255 kódů na jednu Huffmanovu tabulku. Pokud budeme uvažovat, že tyto hodnoty budou v poslední úrovni úplného binárního stromu, pak takový strom bude mít 9 úrovní. Maximální počet uzlů je zde 511. Tuto hodnotu jsem kvůli zarovnání zaokrouhlil na nejbližší mocninu dvou, tedy 512. Pro dekódování je třeba znát 4 tyto binární stromy a jeden uzel vyžaduje 2 bajty. Celková velikost binárních dekódovacích stromů nepřesáhne 4096 bajtů a lze tyto stromy umístit do sdílené paměti. Tím jsem redukoval přístupy do globální paměti.

V ukázce 3.1 je naznačen postup dekódování při použití binárního stromu. Funkce `prepareBits` zajišťuje, aby ve podpůrné struktuře `huff`, bylo načten minimálně `n` bitů

Ukázka kódu 3.1: Dekódování Huffmanova kódu pomocí binárního stromu.

```
void huffDecode(in shared HuffNode tree[512], out uint bits, out int value,
    out uint code, inout HuffStruct huff) {
    HuffNode node;
    ...
    node = tree[0];
    bits = 0;
    value = 0;
    do {
            prepareBits(1, huff);

            bits = bits << 1 | GET_BITS(1);
            RM_BITS(1);

            node = tree[node.code + (bits & 1)];
    }
    while (node.type == TREE_NODE);
    ...
}
```

Huffmanova kódu, přičemž `n` je prvním parametrem funkce. Samotné načítání ale probíhá po bajtech a data jsou získávána z globální paměti. Globální paměti se v tomto případě není možné vyhnout kvůli obecně vyššímu objemu dat. Makro `GET_BITS` pak vrací počet bitů specifikovaných parametrem, zatímco makro `RM_BITS` tento počet bitů označí jako přečtené.

Tato část kódu má však na celkový výkon celkem negativní vliv a to hned ze dvou důvodů. První faktor snižující výkon je smyčka do-while. Zde dochází kvůli rozdílné délce různých kódu k vysoké divergenci vláken. Druhým faktorem je přístup do sdílené paměti. Zde není možné vhodně upravit přístupový vzor, protože je závislý na obsahu Huffmanova kódu. Během přístupů do sdílené paměti tedy dochází ke konfliktu paměťových bank. Pořád se ale jedná o lepší variantu než užití globální paměti.

Před počítaná dekódovací tabulka

Alternativa k předchozí metodě je využití tzv. lookup tabulky. Velikost tabulky je závislá na délce kódu, který chce programátor vyhledávat, což je v tomto případě maximální délka Huffmanova kódu, tedy 16. Z implementačního hlediska se jedná o pole. Pokud použiji 16 následujících bitů Huffmanova kódu jako index tohoto pole, pak získám délku platného Huffmanova kódu a jeho hodnotu.

V ukázce 3.2 se nachází část použité implementace. Výhodou tohoto řešení je, že odstraňuje divergenci vláken, kterou způsobuje průchod binárním stromem. Nevýhodou je, že pro lookup tabulku pro 16 bitů je třeba pole o velikosti $2^16 = 65536$ prvků, přičemž jeden prvek má velikost 2 bajty (délka kódu a hodnota). Tyto tabulky jsou potřeba 4 a tím se dostávám na velikost 0,5MB. Takto velký objem dat se už do sdílené paměti nevleze a je třeba jej ponechat v globální paměti. Ve výsledku je tento postup znatelně rychlejší než průchod binárním stromem.

V aktuální implementaci probíhá vytvoření lookup tabulky na procesoru. Ačkoliv tento proces z celkové doby dekomprese zabírá minimum času, v rámci optimalizace je možné ho provést rovněž na grafické kartě.

Ukázka kódu 3.2: Dekódování Huffmanova kódu pomocí lookup tabulky.

```
void huffDecode(in int table, out uint bits, out int value, out uint code,
    inout HuffStruct huff) {
    ...
    value = 0;

    prepareBits(16, huff);
    bits = GET_BITS(16);

    load = imageLoad(huffLookup, int(bits + LOOKUP_SIZE*table));

    len = load[0];
    code = load[1];

    RM_BITS(len);
    ...
}
```

Kombinace binárního dekódovacího stromu a před počítané dekódovací tabulky

Kombinace binárního stromu a lookup tabulky využívá toho, že nejčetnější znaky Huffmanova kódu jsou nejkratší. Pomocí této techniky je implementováno také dekódování v knihovně *libjpeg*. Urychlení spočívá v tom, že si před generuji lookup tabulku pro omezený počet bitů (např. 8). Pokud zde použiji daný počet bitů z Huffmanova kódu jako index do této tabulky, získám délku platného kódu v bitech a jeho hodnotu, popřípadě informaci, od kterého uzlu je třeba pokračovat dále při průchodu binárním stromem. Tabulky jsou rovněž potřeba 4 a velikost tabulky pro 8 bitů je dostatečně malá na to, aby se vlezla do sdílené paměti spolu s binárními stromy.

Tuto možnost jsem ale neimplementoval, a to z toho důvodu, že pokud by byl pouze jeden kód delší než 8 bitů, opět dochází k divergenci vláken, ačkoliv ne v takové míře jako v prvním případě. I přesto je ale možné, že by tato metoda mohla dosahovat lepších výsledků, než předchozí metoda, a to z důvodu nižšího počtu přístupů do globální paměti.

Reverzní zig-zag uspořádání

Pro reverzní zig-zag uspořádání se nevyplatí kvůli nutnosti přístupu ke globální paměti psát separátní shader. Tento proces bylo nejvýhodnější provést právě během dekódování Huffmanova kódu.

V ukázce 3.3 se nachází definice lookup tabulky pro reverzní zig-zag uspořádání. Tato tabulka specifikuje index `i` prvku v matici DCT koeficientů. Vyhledání příslušného indexu se pak nachází v ukázce 3.4.

Nulování textury

Run-length kódování je v případě JPEG komprese součásti Huffmanova kódu. Během dekódování není potřeba zapisovat nulové hodnoty. Pro lepší paralelizaci je tento přístup výhodnější, protože nedochází k tak vysoké divergenci vláken. To bohužel přináší jednu negativní vlastnost. Některé body v textuře s DCT koeficienty mohou mít nedefinovanou hodnotu, a je téměř jisté, že takové budou. Proto je třeba všechny potřebné textury předem

Ukázka kódu 3.3: Lookup tabulka pro reverzní zig-zag uspořádání.

```
const int zigZagRevOrd[DCTSIZE2] = {
    0,   1,  8, 16,  9,  2,  3, 10,
    17, 24, 32, 25, 18, 11,  4,  5,
    12, 19, 26, 33, 40, 48, 41, 34,
    27, 20, 13,  6,  7, 14, 21, 28,
    35, 42, 49, 56, 57, 50, 43, 36,
    29, 22, 15, 23, 30, 37, 44, 51,
    58, 59, 52, 45, 38, 31, 39, 46,
    53, 60, 61, 54, 47, 55, 62, 63
};
```

Ukázka kódu 3.4: Získání korektního indexu v matici po reverzním zig-zag uspořádání.

```
j = zigZagRevOrd[i];
```

vynulovat. K tomu slouží jednoduchý shader, který nedělá nic jiného, než že všude nastaví hodnotu 0.

3.4 Konverzní utilita

Pro optimální využití paralelizačních schopností grafických karet jsem vytvořil utilitu pro bezeztrátovou konverzi obrázků do vhodného formátu. Z velké části je založena na nástroji `jpegtran`, která je součástí balíku *libjpeg*. Bezeztrátovost převodu je zajištěna tím, že se neprovádí celá dekomprese, ale pouze extrakce DCT koeficientů. Jedná se o převod do baseline formátu s podporou restart markerů.

Kapitola 4

Vyhodnocení

Měření jsem prováděl na sérii fotografií v různých velikostech, s nastavením faktoru kvality na 90%, bez podvzorkování a za klidového stavu počítače. Všechny obrázky byly předem upraveny konverzní utilitou. Během měření tedy nebyly spuštěny žádné další náročné procesy, které by mohly výrazně ovlivnit výsledek. Avšak kvůli různému plánování úloh na procesoru je možné zpozorovat drobné rozdíli mezi jednotlivými časy dekomprese. Proto je výsledný čas dekomprese jedné fotografie získán průměrem z 50 vzorků. Výsledný čas je pak průměrem časů dekomprese jednotlivých fotografií.

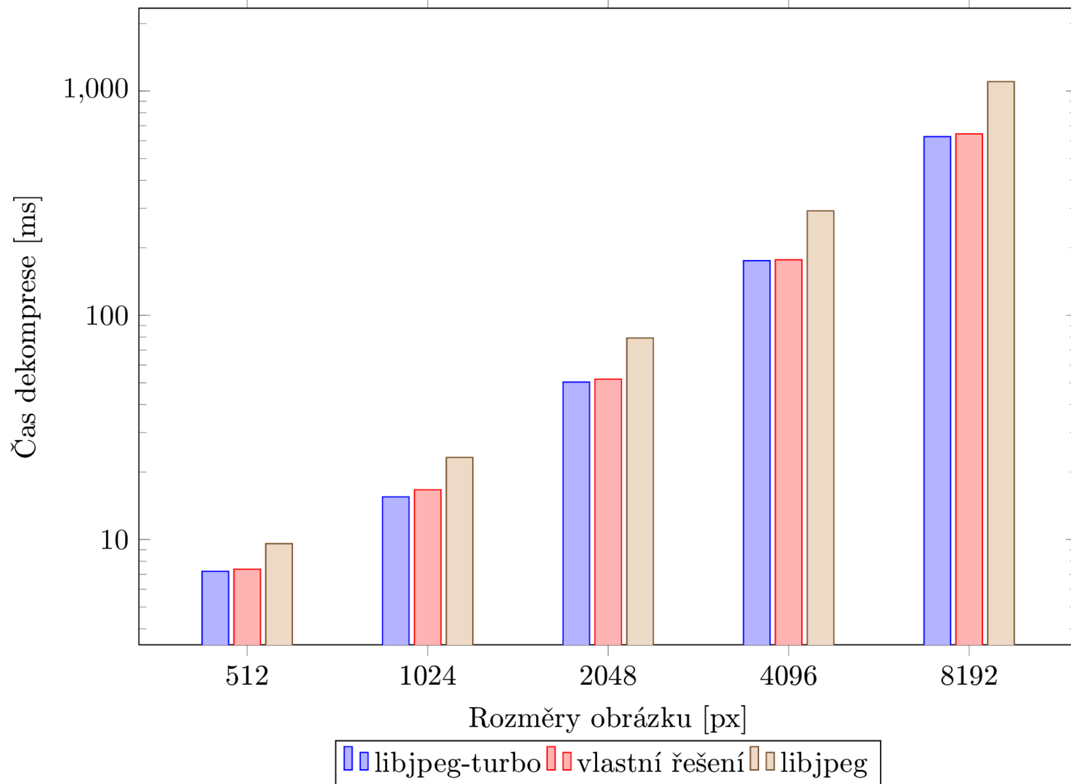
Čas je měřen od začátku dekomprese až po okamžik, kdy je obrázek dostupný jako textura na grafické kartě. Aby bylo zajištěno, že v okamžiku ukončení dekomprese na grafické kartě byly dokončeny veškeré operace, je využito funkcí `glFinish()` a `glMemoryBarrier()`, které dohromady zajistí, že do textury byly zapsány veškerá potřebná data a zároveň byly dokončeny všechny odeslané OpenGL příkazy. Dále aby nebylo měření ovlivňováno rychlostí načítání souborů z disku, tak jsou soubory před začátkem měření načteny do paměti počítače.

Počítač na kterém měření probíhalo obsahuje procesor Intel Core i5 3210M Ivy Bridge, s frekvencí 2,5GHz. Použitá grafická karta je NVIDIA GeForce GTX660M, která disponuje 384 výpočetními jádry. Výsledky měření jsou zobrazeny v tabulce 4.1, dále jsou tyto data zobrazena v semilogaritmickém grafu 4.1. Z těchto dat je patrné, že mé řešení dosahuje výsledků srovnatelných s knihovnou *libjpeg-turbo*. Čas dekomprese také záleží na obsahu obrázků. Během testování jsem zjistil, že mé řešení dosahuje lepších výsledků, pokud se v obrázku nachází větší plochy s konstantní barvou. V takových případech dokonce rychlostí převyšuje knihovnu *libjpeg-turbo*. Stejně tak dosahuje lepších výsledků při nižším faktoru kvality.

Mezi open source projekty zabývající se problematikou JPEG dekomprese na grafických

Rozlišení [px]	Průměrný čas dekomprese [ms]		
	libjpeg	libjpeg-turbo	vlastní řešení
512×512	9.59	7.22	7.38
1024×1024	23.24	15.51	16.68
2048×2048	79.26	50.40	51.93
4096×4096	292.31	175.25	176.84
8192×8192	1101.39	626.04	644.19

Tabulka 4.1: Srovnání časů dekomprese různých řešení pro kvalitu obrázků 90%.



Obrázek 4.1: Srovnání časů dekomprese s CPU implementacemi pro kvalitu obrázků 90%.

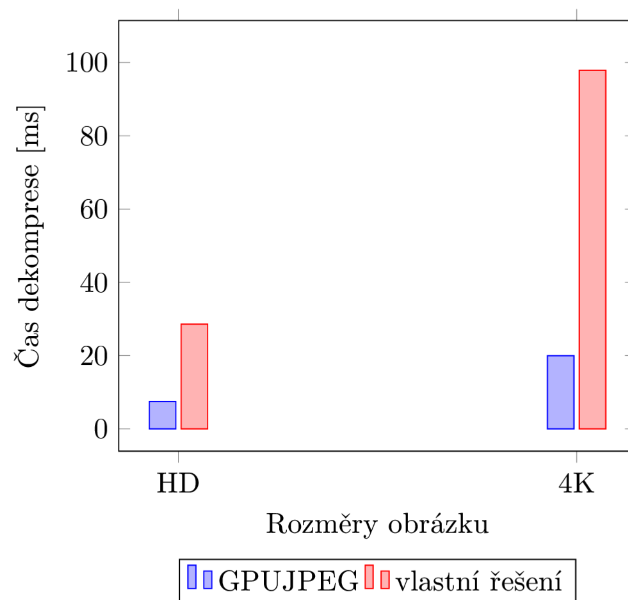
Rozlišení [px]	Průměrný čas dekomprese [ms]	
	vlastní řešení	GPUJPEG
1920 × 1080 (HD)	28.58	7.48
4096 × 2160 (4K)	97.87	19.99

Tabulka 4.2: Srovnání časů dekomprese s GPUJPEG pro kvalitu obrázků 90%.

kartách jsem nalezl knihovnu *GPUJPEG* [1], která je založena na technologii *CUDA*[2]. Dle slov autorů knihovny probíhalo měření podobným způsobem, pouze na jiné sadě obrázků. Pro porovnání jsem vytvořil obrázky stejných velikostí a provedl měření na této sadě. Srovnání se nachází v tabulce 4.2 a v grafu 4.2. Z naměřených hodnot je možno vyčíst, že rychlost knihovny *GPUJPEG* je v tomto případě přibližně 4 až 5 krát rychlejší.

[1] Adresa projektu *GPUJPEG*: https://sourceforge.net/p/gpujpeg/

[2] CUDA je technologie pro paralelní výpočty s využitím grafických karet vytvořena firmou NVIDIA. Tuto technologii ostatní výrobci grafických karet nepřijali a je dostupná pouze na grafických kartách NVIDIA.



Obrázek 4.2: Srovnání časů dekomprese s GPU implementacemi pro kvalitu obrázků 90%.

Kapitola 5

Závěr

Cílem této práce byla akcelerace dekomprese využitím výkonu grafických karet, s ohledem na minimalizaci datových přenosů přes sběrnici. Dle mého názoru se mi tohoto cíle dosáhnout podařilo. Dekomprese dosahuje podobných výsledků jako optimalizované CPU verze a stále je zde prostor pro optimalizace. Přenosy přes sběrnici z této doby zabírají téměř nepatrnou část.

V textu byla popsána problematika paralelizace výpočtů na grafických kartách. Na první pohled se sice nejedná o příliš složitou problematiku, ale dosáhnout maximálního využití výkonu, který grafická karta dosahuje nemusí být vůbec jednoduché. Dále jsme se seznámili s algoritmem JPEG komprese a stručně jsme probrali i jeho různé varianty. Byl popsán postup implementace, alternativy k jednotlivým přístupům a také možnosti dalšího snížení času dekomprese. Následovalo srovnání s dostupnými nástroji pro JPEG dekompresi, přičemž se výsledné řešení umístilo přibližně v průměru. Ačkoliv výsledky nedosahují hodnot, kterých jsem očekával, tak tato práce si může najít své uplatnění v případech, kdy je výhodnější výpočet přenechat grafické kartě a tím snížit zátěž procesoru.

V rámci této práce jsem se blíže seznámil s technologií OpenGL, která poslední dobou získává díky propagaci Linuxových distribucí herní společností *Valve* na popularitě a stává se rozšířenější než dříve. Mimo jiné jsem si vyzkoušel v open source projektech často užívané nástroje jako *autoconf* a *automake*. A v poslední řadě jsem zjistil informace o struktuře grafických karet a možnostech, jak co možná nejlépe využívat jejich výpočetní výkon. Protože mi práce hlavně v závěru přinesla cenné znalosti a docela mě i zaujala, rád bych se v budoucnu chtěl pokusit o další optimalizace.

Literatura

[1] History of OpenGL. [online], Září 2013 [cit. 2014-05-04].
URL http://www.opengl.org/wiki/History_of_OpenGL

[2] Libjpeg-turbo – Performance Study. [online], 2014 [cit. 2014-05-08].
URL http://www.libjpeg-turbo.org/About/Performance

[3] AKELEY, Kurt a SEGAL, Mark: The OpenGL® Graphics System: A Specification. [online], Březen 2014.
URL http://www.opengl.org/registry/doc/glspec44.core.pdf

[4] CCITT: ITU-T Recommendation T.81 – Digital compression and coding of continuous-tone still images. Září 1992.
URL http://www.w3.org/Graphics/JPEG/itu-t81.pdf

[5] FERGUSON, J., T. a RABINOWITZ: Self-synchronizing Huffman codes (Corresp.). *Information Theory, IEEE Transactions on*, ročník 30, č. 4, Červenec 1984: s. 687–693, ISSN 0018-9448, doi:10.1109/TIT.1984.1056931.

[6] HAN, T. David a ABDELRAHMAN Tarek S.: [online].
URL http://www.eecis.udel.edu/~cavazos/cisc879/papers/a3-han.pdf

[7] KHAYAM, S. A.: The Discrete Cosine Transform (DCT): Theory and Application. [online], Březen 2003.
URL http://www.egr.msu.edu/waves/people/Ali_files/DCT_TR802.pdf

[8] LILLEY, Chris: JPEG JFIF. [online], Leden 2003 [cit. 2014-05-08].
URL http://www.w3.org/Graphics/JPEG/

[9] NGUYEN, Hubert: *Gpu Gems 3*. Addison-Wesley Professional, první vydání, 2007, ISBN 9780321545428.

[10] The Khronos Group: About The Khronos Group. [online], 2014 [cit. 2014-05-04].
URL http://www.khronos.org/about/

[11] The Khronos Group: OpenGL Overview. [online], 2014 [cit. 2014-05-04].
URL http://www.opengl.org/about/

[12] WILT, Nicholas: *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, první vydání, 2013, ISBN 9780321809469.

[13] YOUNG, Eric: DirectCompute Optimizations and Best Practices. [online], Září 2010.
URL http://www.nvidia.com/content/GTC-2010/pdfs/2260_GTC2010.pdf

[14] ZHU, P.P. a LIU, J. G. a DAI, S. K. a WANG G. Y.: Scaled AAN for Fixed-Point Multiplier-Free IDCT. *EURASIP Journal on*, ročník 2009, Únor 2009: str. 9, doi:10.1155/2009/485817.
URL
http://asp.eurasipjournals.com/content/pdf/1687-6180-2009-485817.pdf

Příloha A

Obsah CD

- `/dokumentace.pdf` – Textová část bakalářské práce ve formátu PDF
- `/dokumentace-print.pdf` – Textová část bakalářské práce ve formátu PDF určená pro tisk
- `/dokumentace/` – Zdrojové kódy a obrázky textové části práce v programu LaTeX
- `/dokumentace.tar.gz` – Komprimovaná verze zdrojových kódu v programu LaTeX
- `/ogljpeg` – Zdrojové kódy aplikace
- `/ogljpeg.tar.gz` – Komprimovaná verze zdrojových kódů aplikace
- `/video/`
  - `ogljpeg.mp4` – Prezentační video
  - `ogljpeg.jpg` – Náhled videa
  - `ogljpeg.xml` – Popis videa ve formátu XML
- `readme.txt` – Popis obsahu CD