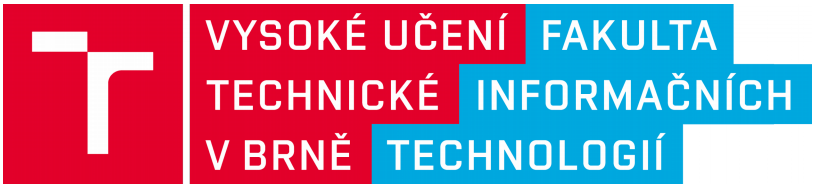


IPA - pokročilé assemblery
2017/2018

Grafický editor: Rotace obrázku s antialiasingem

Tomáš Pazdiora
login: xpazdi02

4. prosince 2017

Abstrakt

Tato práce se zabývá optimalizací rotace obrázku různých rozměrů s limitem minimální šířky 4px uloženého ve formátu 3x8bit, BGR, array of structures. Optimalizace je provedena na assembly úrovni na 64bit architektuře s využitím SIMD instrukcí z rozšíření SSE, AVX, AVX2 a FMA. Práce obsahuje matematický podklad pro transformaci souřadnic, rozbor algoritmu a příklad použití.

Abstract

This paper focuses on optimization of rotation of image with different sizes with limit of minimal width 4px stored in format 3x8bit, BGR, array of structures. Optimization is written on assembly level on 64bit architecture with use of SIMD instructions from SSE, AVX, AVX2 and FMA extensions. This paper contains mathematical basis for coordinate transformation, analysis of algorithm and use case.

Obsah

1 Zadání

Zadáním tohoto projektu bylo implementovat a následně optimalizovat rotaci obrázku bez aliasingu. Optimalizovaná část měla být zapsána v jazyce symbolických instrukcí s pomocí instrukcí SSE, AVX a případně AVX2.

1.1 Aliasing

Při rotaci či jiné transformaci obrázku může vznikat aliasing, a to právě pokud se pro každý zdrojový pixel hledá cílový (transformovaný) pixel. Pro odstranění aliasingu je nutné provést opačný postup a pro každý cílový pixel hledat pixel zdrojový.

Použitím následujícího kódu vzniká aliasing:

```
1  for(x = 0; x < width; x++){
2   for(y = 0; y < height; y++){
3    [tx,ty] = transform([x,y]);
4    Destination[tx,ty] = Source[x,y];
5   }
6  }
```

Následující kód aliasing eliminuje:

```
1  for(x = 0; x < width; x++){
2   for(y = 0; y < height; y++){
3    [tx,ty] = inverse_transform([x,y]);
4    Destination[x,y] = Source[tx,ty];
5   }
6  }
```

2 Řešení

Pro řešení byl využit výše popsaný algoritmus který nezpůsobuje aliasing. Tento algoritmus sestaví inverzní transformační matici, pomocí které mapuje souřadnice zdrojového obrázku na cílový, transformovaný. Poté už jen přesune pomocí těchto transformovaných souřadnic data ze zdrojového do cílového obrázku.

2.1 Transformační matice

Při otáčení obrázku je důležitý střed rotace (origin) a úhel otočení (angle). Případně můžeme obrázek i škálovat (scale). Dílčí transformační matice si tedy označíme jako M_T pro matici posunu ke středu rotace, M_R jako matici rotace a M_S jako matici škálování. Výslednou inverzní transformační matici M^{-1} tedy vytvoříme následovně:

$$M^{-1} = M_T^{-1} \cdot M_S^{-1} \cdot M_R^{-1} M_T$$

Matice můžeme rozepsat podrobněji jako:

$$M^{-1} = \begin{bmatrix} 1 & 0 & -T_x \\ 0 & 1 & -T_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x^{-1} & 0 & 0 \\ 0 & S_y^{-1} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

Vynásobením dílčích matic získáme následnou inverzní transformační matici:

$$M^{-1} = \begin{bmatrix} \frac{\cos(\alpha)}{S_x} & \frac{\sin(\alpha)}{S_x} & \frac{\cos(\alpha) \cdot T_x}{S_x} - T_x + \frac{T_y \cdot \sin(\alpha)}{S_x} \\ -\frac{\sin(\alpha)}{S_y} & \frac{\cos(\alpha)}{S_y} & \frac{\cos(\alpha) \cdot T_y}{S_y} - T_y - \frac{T_x \cdot \sin(\alpha)}{S_y} \\ 0 & 0 & 1 \end{bmatrix}$$

Samotná transformace souřadnic pak vypadá takto:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = M^{-1} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Vzhledem k tomu, že nás třetí složka vektoru nezajímá, tak je v programu transformační matice zapsána pouze jako **matice 2x3**.

2.2 Interpolace

Zadání nijak nespecifikovalo interpolaci a tak pro jednoduchost byla zvolena interpolace nejbližším sousedem (nearest-neighbour). Tato metoda zaokrouhluje souřadnici na nejbližší souřadnici pixelu. Ostatní metody mohou poskytnout kvalitnější výsledky, jako například bikubická nebo alespoň bilineární interpolace. Tyto metody ale zase o něco více komplikují implementaci a požadují více přístupů do paměti.

2.3 Omezení

Vzhledem k povaze algoritmu jsem přistoupil na limitaci, která požaduje, aby **šířka** zpracovávaného obrázku byla **minimálně 4 pixely**. Tato limitace zjednoduší a urychlí algoritmus.

Uvažoval jsem ještě o limitaci, která by požadovala aby byl násobek šířky a výšky obrázku dělitelný 8mi. Tato limitace by také zjednodušila a urychlila algoritmus. Od této limitace jsem ale nakonec ustoupil, jelikož mi přišla příliš svazující.

3 Optimalizace

Pro optimalizaci kódu byly použity instrukce SSE, AVX, AVX2 a **FMA**. Vzhledem k tomu, že použití FMA instrukcí nebylo v zadání, je možné je v případě nekompatibility při kompilaci pomocí NASM vypnout přepínačem *WITHOUT_FMA*.

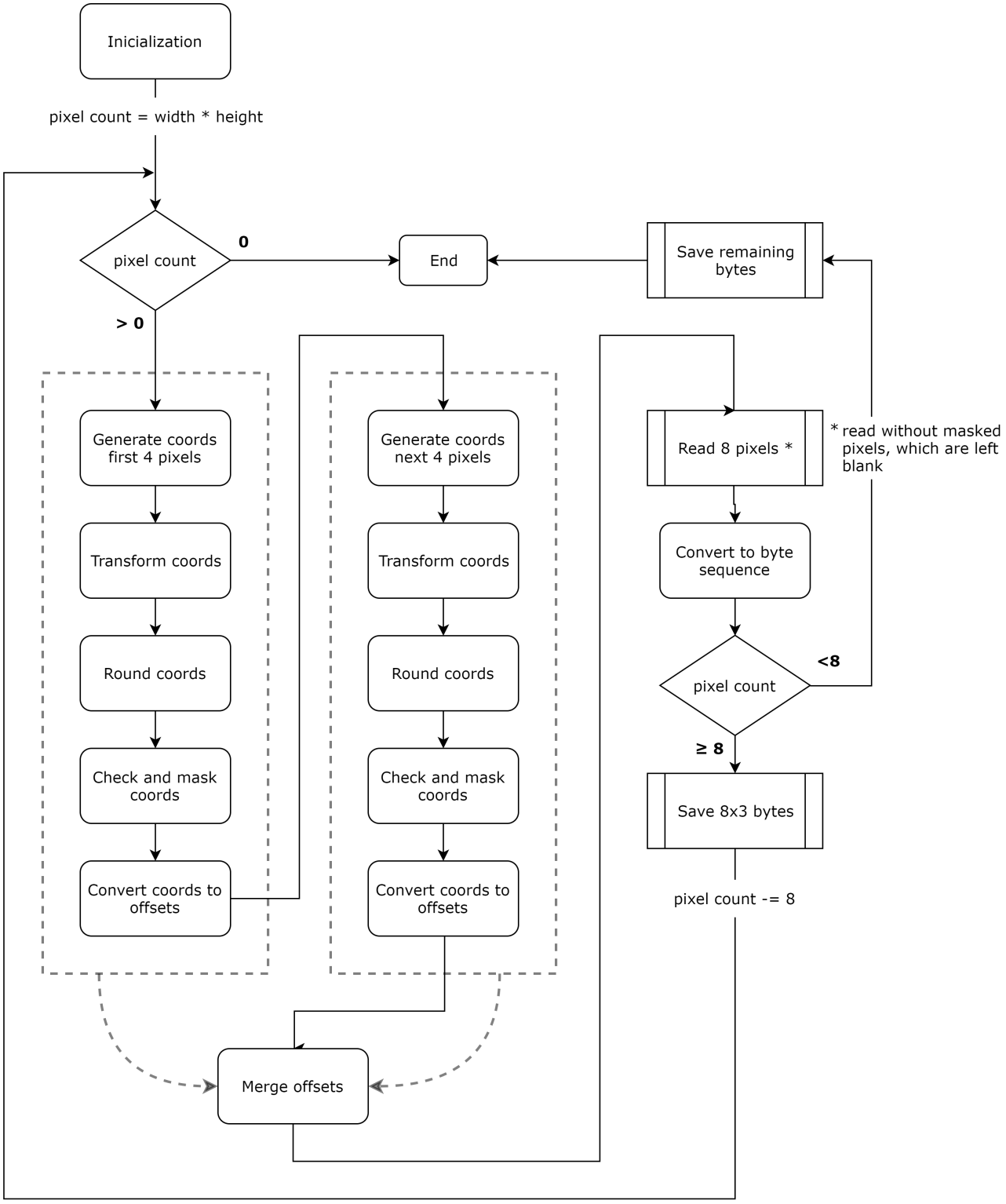
Kód byl po domluvě s vedoucím napsán pro 64bitovou architekturu, namísto původně zadané 32bitové.

Optimalizovaná byla jen zásadní část algoritmu. Načtení argumentů, ověření limitů a sestavení inverzní transformační matice, které se provádí pouze jednou, je napsáno v jazyce C v modulu „ipa_algorithm_c.c“. Optimalizovaný kód je pak k nalezení v modulu „xpazdi02.s“.

3.1 Struktura algoritmu

V inicializační fázi algoritmus načítá potřebné hodnoty do registrů. Tyto hodnoty zahrnují zdroj, cíl, šířku a výšku obrázku, inverzní transformační matici a pomocné konstanty. Další kód už probíhá ve smyčce. V první části se nachází dva identické bloky kódu v sérii za sebou. V těchto blocích se generují souřadnice 4 pixelů, transformují se inverzní transformační maticí, provede se zaokrouhlení na souřadnici nejbližšího pixelu, následně se kontroluje rozsah souřadnic a neplatné jsou maskovány. Nakonec se převádí souřadnice pixelu na jeho offset v obrázku. Po zpracování těchto dvou bloků se výsledné čtveřice offsetů spojí v osmici offsetů. Pomocí této osmice offsetů se načtě ze zdrojové paměti 8 pixelů na požadovaných souřadnicích, nejsou-li maskovány. Pixely na maskovaných souřadnicích jsou nahrazeny prázdným pixelem. Těchto 8 pixelů se sloučí do posloupnosti bytů odpovídající výstupnímu formátu. Tato posloupnost je uložena do cílové paměti. Pokud nejsou zpracovány všechny pixely, pokračuje se ve vykonávání této smyčky.

Struktura algoritmu je znázorněna na následujícím Obrázku 3.1.



Obrázek 3.1: Diagram optimalizovaného algoritmu rotace obrázku

3.2 Optimalizace transformace vektoru maticí

V algoritmu je potřeba transformovat vektor $(u, v, 1)$ maticí M_{3x3}. Výpočet vypadá následovně:

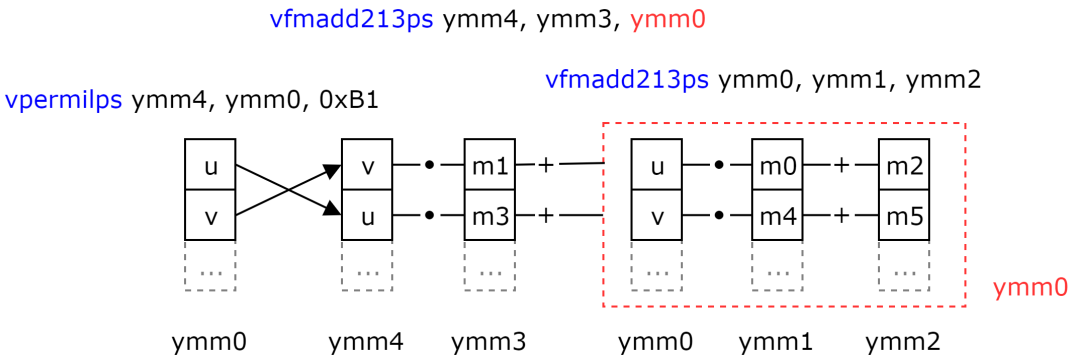
$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = M_{3x3} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} M_0 & M_1 & M_2 \\ M_3 & M_4 & M_5 \\ M_6 & M_7 & M_8 \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} M_0 \cdot u + M_1 \cdot v + M_2 \cdot 1 \\ M_3 \cdot u + M_4 \cdot v + M_5 \cdot 1 \\ M_6 \cdot u + M_7 \cdot v + M_8 \cdot 1 \end{bmatrix}$$

Jak již bylo uvedeno výše, tak vzhledem k tomu, že nás třetí složka vektoru nezajímá, ji nepočítáme. Znovu bych rád upozornil, že v programu je proto také matice zapsána jako **matice 2x3**.

Pro transformaci využíváme rozšíření AVX2 a FMA. Rozšíření AVX2 nám dovoluje zpracovávat až 8 čísel najednou, v našem případě 4 dvousložkové vektory. Transformovat maticí tyto dvě složky čtyř vektorů nás ve výsledku při výpočtu pomocí následujícího kódu stojí přibližně 10 instrukčních cyklů na architektuře Intel Haswell[1]. Což je statisticky 2.5 instrukčních cyklů na jednu transformaci.

```
1  ;ymm1 = m4  m0  m4  m0  m4  m0  m4  m0
2  ;ymm2 = m5  m2  m5  m2  m5  m2  m5  m2
3  ;ymm3 = m3  m1  m3  m1  m3  m1  m3  m1
4
5  ;ymm0 = v3  u3  v2  u2  v1  u1  v0  u0
6
7  vpermilps ymm4, ymm0, 0xB1
8  vfmadd213ps ymm0, ymm1, ymm2
9  vfmadd213ps ymm4, ymm3, ymm0
10
11 ;ymm4 = v3' u3' v2' u2' v1' u1' v0' u0'
```

Na následujícím Obrázku 3.2 můžete vidět vizualizaci výše uvedeného kódu.



Obrázek 3.2: Vizualizace výpočtu transformace vektoru maticí

1 Odvozeno dle údajů ze stránek *Intel Intrinsics Guide*[1]

4 Spuštění programu

TODO: Parametry

5 Závěr

Při testu na procesoru Intel® Core™ i5-5200U s architekturou Intel Broadwell bylo oproti referenčnímu řešení napsaném v jazyce C docíleno zrychlení TODO: Zrychlení.

TODO: Github???

6 Reference

[1] Intel, *Intel Intrinsics Guide*, 3.4, September 7, 2017, https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[2] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Volume 1 – 4, October 2017, https://software.intel.com/en-us/articles/intel-sdm#combined

[3] Félix Cloutier, *x86 Instruction Set Reference*, 2014, http://www.felixcloutier.com/x86/

[4] René Jeschke, *x86 Instruction Set Reference*, 2016, http://x86.renejeschke.de/

[5] Rich Skorski, In-depth: Windows x64 ABI: Stack frames, *Gamasutra*, October 1, 2012, https://www.gamasutra.com/view/news/178446/Indepth_Windows_x64_ABI_Stack_frames.php

6.1 Použitý software

[6] The NASM development team, *NASM - The Netwide Assembler*, 2.13.01, May 1, 2017, http://www.nasm.us/

[7] OpenCV team, *OpenCV*, library, 3.3.1, October 23, 2017, https://opencv.org/

[8] Microsoft Corporation, *Visual Studio Community 2017*, 15.4.3, November 8, 2017 https://www.visualstudio.com/cs/vs/community/

[9] Henk-Jan Lebbink, *Asm-Dude*, Visual Studio extension, 1.9.3.2, November 21, 2017 https://github.com/HJLebbink/asm-dude

[10] Junio Hamano and others, *Git*, 2.14.1, August 10, 2017 https://git-scm.com/

[11] Henk Westhuis, *Git Extensions*, 2.50.02, September 6, 2017 https://github.com/gitextensions/gitextensions

[12] The Document Foundation, *LibreOffice*, 5.4.3.2, November 3, 2017, https://www.libreoffice.org/

[13] JGraph Ltd., *draw.io*, 2017, https://www.draw.io/