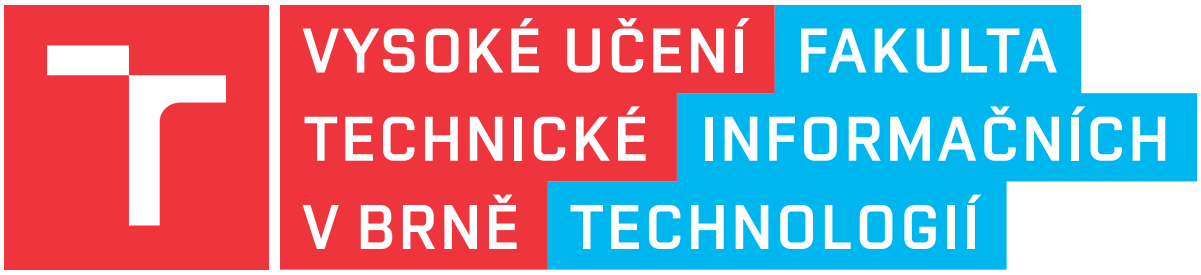

Pokročilé asemblery (IPA)

Graficky editor: Zvýraznění barvy v obraze s využitím K-means

v Brně
10. prosince 2017

Václav Martinka
xmarti76

Obsah

1 Zadání

Cílem projektu je implementovat v asembleru s pomocí SSE, AVX nebo AVX2 instrukcí algoritmus pro zvýraznění jedné z majoritních barev.

Použijte k tomu algoritmus *k-means* pomocí kterého určíte dominantní odstíny barev. Například budete mít tři barevné složky a jednu z nich vykreslíte, zatímco zbylé pixely zůstanou v odstínech šedi.

2 Návrh řešení

Nejdříve je nutné nastudovat princip algoritmu *k-means*. Mimo to je potřeba zjistit fungování dodané šablony programu. Na závěr zbývá převést část obrázku do stupňů šedi a zbytek vykreslit.

2.1 Algoritmus k-means

Tento algoritmus vytváří předem daný počet skupin, tzv. *shluků*, bodů v euklidovském prostoru. Pro každý tento shluk je definován centroid. Ten lze na začátku algoritmu buď generovat náhodně nebo staticky. V případě obrázku by bylo možné ho taktéž generovat jako náhodnou (popř. statickou) pozici v obrázku. Barva na této pozici by pak byla použita jako centroid.

Každý bod (pixel) je přiřazen do jednoho ze shluků na základě vzdálenosti k jeho centroidu, přičemž se vybírá ten nejbližší. Je dobré pro tento výpočet vytvořit funkci. Protože bude volána pro každý pixel obrázku, bylo by vhodné ji co nejvíce akcelerovat pomocí vektorizace.

V případě barvy se bavíme o třírozměrném euklidovském prostoru, kdy jednotlivé osy představují složky barvy R, G a B. Vzorec pro výpočet vzdálenosti je následující:

$$\sqrt{(R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2}$$

Po roztřídění všech bodů se vypočtou nové centroidy, jakožto nové geometrické středy jednotlivých shluků. Taktéž lze provést výpočet pokaždé, když dojde k přidání bodu do shluku. Opět se jedná o často prováděnou činnost, tudíž je vhodné ji vektorizovat.

Algoritmus končí v případě, že nové centroidy se od těch předchozích neliší, nebo se liší jen nepatrně. Vzhledem k možným zaokrouhlovacím chybám bude lepší neočekávat přesnou shodu. Případně lze začlenit počítadlo iterací a stanovit maximální hranici, která zajistí, že se algoritmus vždy ukončí. [1]

2.2 Převod do stupňů šedi

Šedá barva je taková, která nabývá pro všechny tři složky R, G a B stejnou hodnotu. Tu lze vypočítat jako průměr jednotlivých složek. Protože lidské oko je různě citlivé na různé barevné složky, tak nelze použít pouze aritmetický průměr, nýbrž je nutné provést tzv. vážený průměr: [3]

$$Y = 0,299 * R + 0,587 * G + 0,114 * B$$

Jednotlivé konstanty nejsou zrovna vhodné pro dělení v CPU, proto jsem se rozhodl je mírně upravit:

$$Y = 0,25 * R + 0,5 * G + 0,25 * B$$

Takto lze výsledku dosáhnout pouhými bitovými posuny.

2.3 Který shluk vybrat

Na závěr se nabízí otázka který z výsledných shluků vybrat k vykreslení? Možnosti jsou v podstatě čtyři: vybrat vždy ten n-tý, určit ho náhodně, vybrat ten největší/nejmenší shluk nebo nechat vybrat uživatele.

V případě, že jsou prvotní centroidy určeny skutečně náhodně, lze prohlásit první a druhou možnost za identické. Čtvrtá možnost naproti tomu bude pro uživatele nejpříjemnější, proto jsem se vydal touto cestou. Pokud uživatel žádnou barvu nezadá, použije se první možnost, i když čistějším řešením by byla možnost třetí.

3 Implementace

Pro začátek jsem měl k dispozici kostru programu, která měla na starosti načtení obrázku, volání funkce na jeho zpracování a zobrazení výsledku. Tudíž stačilo implementovat čistě samotnou funkci. Jako první bylo třeba nastudovat v jakém tvaru je obrázek načten. Experimentálně jsem zjistil, že se jedná o pole struktur, kde jsou složky uloženy v pořadí B, G a R. Pro akceleraci algoritmu by bylo zajisté vhodnější načítat obrázek jako strukturu polí, ale to by si vyžádalo větší zásah do kódu šablony, což bylo v rozporu se zadáním, tedy implementovat pouze danou funkci.

3.1 Implementace v jazyce C

Po nastudování principů jednotlivých algoritmů a fungování šablony jsem se rozhodl algoritmus nejdříve implementovat v jazyce C. Nesnažil jsem se o nijak optimální implementaci. Hlavní cílem bylo vytvořit funkční algoritmus, který bude co nejjednodušší s důrazem na přehlednost kódu. Naprogramovat funkční prototyp zabralo cca jedno odpoledne.

3.2 Implementace v assembleru

Poté, co se mi podařilo dokončit funkční implementaci v jazyce C, bylo možné začít přepisovat program do assembleru. Na začátek jsem se nesnažil o žádnou vektorizaci a pro výpočty jsem využíval FPU koprocesor.

Cílem bylo dosáhnout vůbec funkčního kódu a navrhnout strukturu programu. Nakonec jsem došel k výsledku, že je program složen ze 6 procedur:

`getUserColor` Se stará o načtení uživatelem zadané barvy z příkazové řádky. Pro svůj běh využívá proceduru `hex2dec`, které převádí hexadecimální stringovou hodnotu na číslo.

Výsledkem této procedury je uživatelem zadaná barva v lokální proměnné `userColor` na zásobníku nebo -1 v případě chyby (popř. nezadané barvy).

`generateRandomColors` Náhodně vygeneruje 4 centroidy. Barvy se generují náhodně bez ohledu na vstupní obrázek (možná by bylo lepší generovat náhodnou pozici do obrázku, což by mohlo zajistit rychlejší konvergenci k výsledku). Jako zdroj náhodných čísel je využita AVX2 instrukce `rdrand`.

V případě, že uživatel zadal barvu, kterou chce vykreslit, je tato barva použita jako jeden z centroidů, čímž by se mělo dosáhnout lepších výsledků.

`clustering` Implementuje k-menas algoritmus. Pro svůj běh volá proceduru `distance`, která spočte vzdálenost aktuálního pixelu k jednotlivým centroidům. Tato procedura je vektorizována, tudíž všechny 4 vzdálenosti jsou spočteny naráz.

Číslo nejbližšího shluku si poznačím do B složky výstupního obrázku. Tím šetřím místo v paměti za cenu většího počtu čtení z paměti při ukončení algoritmu. Dále se vypočte nový centroid (ten

ale aktuální centroid nahradí až po té, co se ukončení právě probíhající iterace). Tento výpočet je opět vektorizován a tak se pomocí jedné instrukce počítají všechny 3 složky barvy zároveň.

Na závěr dojde k porovnání starých a nových centroidů. Pokud se součet absolutních rozdílů jednotlivých barev liší méně jak o 5, dojde k zastavení algoritmu. Pro zajištění konvergence jsem algoritmus doplnit o počítadlo iterací, kdy při 50 iteraci dojde natvrdo k ukončení.

`toGray` Na závěr je obrázek převeden do stupňů šedi. Nejdříve si zjistím, do kterého shluku spadá uživatelem zadaná barva (pokud byla zadána). Následně index tohoto shluku očekávám v B složce výstupního obrázku. V případě shody obnovím tuto složku ze vstupního obrázku. V opačném případě vypočtu odstín šedé a nahradím touto hodnotou všechny tři RGB složky. Tento algoritmus **nebyl** vektorizován i když by to jistě bylo možné. Ovšem vzhledem k tomu, že tato procedura se za běh programu spustí pouze jednou, nejednalo by se o nijak významnou úsporu.

3.3 Použití aplikace

Výsledkem projektu je jednoduchá aplikace. Ta ke svému běhu vyžaduje *OpenCV* knihovny a knihovnu *Student_DLL* obsahující samotnou implementaci. Příklad spuštění:

```
Text_editor.exe <image> [color]
```

image Vstupní obrázek. V závislosti na operačním systému jsou podporovány tyto formáty: [2]

- Windows bitmaps (*.bmp, *.dib)
- JPEG files (*.jpeg, *.jpg, *.jpe)
- JPEG 2000 files (*.jp2)
- Portable Network Graphics (*.png)
- Portable image format (*.pbm, *.pgm, *.ppm)
- Sun rasters (*.sr, *.ras)
- TIFF files (*.tiff, *.tif)

color Uživatel může definovat, kterou barvu chce vykreslit. Očekávaný vstupní formát je ve tvaru `#RRGGBB`, zkrácený formát `#RGB` **není** podporován.

4 Testování a experimenty

Program byl testován po celou dobu svého vývoje. Při programování v assembleru velmi snadno dochází k chybám (často velmi významným), proto by to bez testování ani nešlo.

4.1 Chyby během vývoje

Velmi dobrým pomocníkem bylo krokování s přímým pohledem do paměti. Nejčastější chybou byl právě zápis na špatnou pozici v paměti. Docházelo buď k přepisu vlastních hodnot, v horším případě přístupu mimo povolený rozsah. Další chyby vznikali spíše nepozorností (překlepy, kopírování částí kódu bez přepsání návěstí, apod.). Jedna z chyb byla záměna funkcí `min` a `max`. Program byl tak sémanticky a syntakticky správně, nedocházelo k pádům a dokonce i výsledky vypadali přijatelně. Jedinou chybou tak bylo, že algoritmus nekonvergoval k řešení, místo toho se zacyklil a vzájemně prohazoval centroidy jednotlivých shluků.

Dalším problémem během testování byli náhodně generované centroidy. Ze začátku jsem je tvořil staticky. Ale po té, co již byl algoritmus odladěn a začal jsem centroidy generovat dynamicky, docházelo občas k pádům aplikace. Četnost nebyla vysoká (např. 1 za 20 spuštění). Hledání takové chyby taktéž není jednoduché.

4.2 Experimenty

Na závěr implementace jsem provedl 10 spuštění a porovnal výsledný počet taktů v případě implementace C bez optimalizací, s maximální optimalizací (`/Ox`) a assembleru.

C		**C /Ox**		**Assembler**	
Instrukcí	**Iterací**	**Instrukcí**	**Iterací**	**Instrukcí**	**Iterací**
863 354 145	5	312 212 740	5	658 465 229	10
2 082 090 312	12	176 182 598	4	819 640 141	12
4 377 146 204	26	242 332 530	5	999 989 864	15
2 198 241 208	13	677 543 109	13	810 430 594	12
5 560 222 146	33	231 411 985	5	743 860 003	11
1 217 740 299	7	607 917 354	14	938 293 936	14
874 202 758	5	324 320 304	6	675 029 275	10
865 152 700	5	522 411 366	11	612 949 219	9
513 273 310	3	476 837 612	11	342 883 684	5
692 215 900	4	450 031 011	10	539 962 083	8
Průměrně:					
1 924 363 898	11	402 120 061	8	714 150 403	11
Průměr na 1 iteraci					
170 297 690		**47 871 436**		67 372 680	

Prvotní měření vypadalo povzbudivě, algoritmus vykazoval téměř sedminásobné zrychlení. Bohužel se ukázalo, že na vině bylo povolení generování debugových informací, které výrazně zpomalili C-čkový kód.

Po-té, co jsem generování debugových informací zakázal a provedl nové měření (viz tabulka), kleslo zrychlení na 2,5-násobek. Po zapnutí optimalizací došlo dokonce ke zpomalení o 40 %.

Ještě možná stojí za zmínku, že prvotní implementace bez vektorizací pouze s využitím FPU a omezením pouze na jednu iteraci byla několikanásobně pomalejší oproti plně funkční C-čkové variantě. Nemám sice konkrétní čísla, ale je patrné, že používání FPU značně zpomaluje výsledný program.

5 Závěr

Z výsledků je patrné, že implementace algoritmu v assembleru s ohledem na její časovou náročnost, nižší přehlednost a horší budoucí rozšiřitelnost se nevyplatí.

Oproti neoptimalizovanému kódu v jazyce C došlo pouze k 2,5-násobnému zrychlení. V případě zapnutí optimalizací došlo dokonce ke zpomalení o 40 %. Jistě příznivějších výsledků by šlo dosáhnout pomocí 64-bitového režimu a použitím struktury polí místo pole struktur. Ovšem dá se předpokládat, že by i v tomto případě optimalizátor ještě zvýšil výkon aplikace a tudíž by stále předčil assembler.

Reference

[1] *K-means* [Wkipedia]. Dostupné z: <https://cs.wikipedia.org/wiki/K-means>.

[2] *Reading and Writing Images and Video* [OpenCV]. Dostupné z: <https://docs.opencv.org/2.4/modules/highgui/doc/reading_and_writing_images_and_video.html\#imread>.

[3] *Základy počítačové grafiky* [VUT Brno]. 2016/2017.