Vysoké učení technické v Brně
Fakulta informačních technologií

Pokročilé assemblery – Projekt

Filtr pro zvýraznění barev založený na k-means

11. května 2021 Ondřej Ondryáš

Úvod

Cílem projektu je vytvořit program, který provede shlukovou analýzu barev obrázku pomocí algoritmu *k-means clustering* [1] a selektivně převede do stupňů šedi všechny body, které nepatří do vybraného shluku. Program tedy „roztřídí body“ obrázku do zvoleného počtu skupin podle dominantní barvy a „odbarví“ všechny tyto skupiny kromě jedné. Primárním záměrem bylo implementovat tento proces pomocí vektorových (SIMD) instrukcí z rozšíření instrukční sady z rodiny SSE a AVX [2] a docílit tím zrychlení celého procesu (oproti implementaci např. v jazyce C).

Popis řešení

Podle zadání byl proces rozdělen na dva samostatné podproblémy implementované ve dvou funkcích v assembleru architektury x86-64. První funkce, `algorithm_kmeans`, prochází obrazová data a pomocí algoritmu *k-means* přiřazuje jednotlivým bodům obrázku číslo shluku, do kterého daný bod patří. Druhá funkce, `algorithm_imgproc`, na základě vybraného shluku do výstupního obrázku uloží buď barvu vstupního bodu, pokud tento bod náleží vybranému shluku, nebo barvu v odstínech šedé[1], kterou podle vstupního bodu vypočítá pomocí vhodného převodního vztahu (viz dále).

Tyto dvě funkce jsou volány z programu v jazyce C++, který provádí pomocné činnosti, jako je parsování argumentů příkazové řádky, načítání, zobrazování a ukládání obrázků (pomocí knihovny OpenCV) nebo správa paměti. Tento program vychází z poskytnuté šablony. V tomto programu se také nachází referenční (neoptimalizované) implementace výše zmíněných funkcí[2]; v kapitole (REF) je provedeno srovnání obou verzí.

Je vhodné podotknout, že implementace využívá instrukcí ze sady AVX a AVX**2**. Ty se poprvé objevily u procesorů Intel v roce 2013 a podporuje je většina procesorů Intel i AMD vydaných po roce 2015; Intel je ovšem až donedávna neumožňoval použít na procesorech řady Pentium a Celeron, na takových strojích by tedy tento program nefungoval [3].

Spuštění

Přeložený program je možné spustit následujícím příkazem (argumenty ve složených závorkách jsou povinné, argumenty v hranatých závorkách jsou nepovinné):
`projekt.exe {vstupní soubor} {počet shluků} {cílový shluk} [seed] [výstupní soubor]`

Argument `cílový shluk` určuje *index* (číslovaný od nuly) shluku, jehož body mají zůstat vybarveny (ostatní budou převedeny do stupňů šedi). Počáteční hodnoty středů shluků jsou generovány (pseudo)náhodně, což může mít vliv na výsledek, proto je možné uvést číselný argument `seed` – dvě spuštění programu se stejnou hodnotou tohoto argumentu vygenerují shodné počáteční hodnoty středů. Pokud je uveden `výstupní soubor`, program výstup nezobrazuje v okně, ale pouze uloží. Výstupní soubor není možné uvést bez uvedení argumentu `seed`.

[1] takovou, která má všechny tři barevné složky nastavené na stejnou hodnotu

[2] Použitou verzi funkcí je možné změnit nastavením hodnoty maker `USE_C_KMEANS` a `USE_C_IMGPROC` v souboru `projekt.cpp`.

Implementace algoritmů

Funkce `algorithm_kmeans` dostává jako argumenty ukazatel na pole obrazových dat v paměti, ukazatel na pole záznamů o středech shluků v paměti a počet těchto záznamů, ukazatel na pole výsledků a celočíselné počty shluků a obrazových bodů. Algoritmus v cyklu prochází pole bodů, vždy načte čtyři z nich a pro tyto čtveřice prochází po jednom záznamy o středech shluků, přičemž najednou pro čtveřici bodů vyhodnocuje vzdálenost od jednotlivých středů, a tak i příslušnost k nim.

Po vyhodnocení všech obrazových bodů je pak iterováno nad záznamy o centrech shluků a je spočítána jejich případná nová poloha. Pokud žádné z center polohu nezmění, funkce je ukončena, v opačném případě se začíná od začátku a vyhodnocují se příslušnosti bodů k centrům shluků s nyní již upravenými polohami.

Návrh paměťového modelu

Pro optimalizaci pomocí SIMD instrukcí je nutné zvolit vhodnou reprezentaci dat v paměti. V tomto případě bylo nutné způsob implementace přizpůsobit reprezentaci dat 2D obrázků, kterou využívá použitá knihovna OpenCV. Ta (ve výchozím nastavení) načítá obrázky jako sérii po sobě jdoucích bajtů, které reprezentují postupně složky B, G a R jednotlivých obrazových bodů. Z důvodu dosažení vyšší přesnosti výpočtů byla při použití vektorových registrů (technologie SSE) zvolena forma čtyř 32bitových čísel s pohyblivou desetinnou čárkou (*float*). Podstatnou *část* funkce tedy tvoří načítání takto *lineárně* uložených celých čísel do podoby, která je praktická pro paralelizaci.

Dalším klíčovou položkou v paměti jsou struktury, které obsahují informace o jednotlivých středech shluků. Ta je definována jako struktura 8 za sebou jdoucích 32bitových desetinných čísel (v C++ kódu je pojmenována `centroid`). První tři ukládají hodnoty složek R, G, B středu, čtvrté je prázdné (slouží pro zarovnání celé struktury), do dalších tří jsou v každé iteraci přičítány hodnoty složek R, G, B bodů, které středu nově připadají, a v posledním je uložen počet takových bodů. V závěru každé iterace jsou tedy pátá až sedmá složka vyděleny osmou, zjistí se, jestli došlo ke změně, a tyto složky se nastaví na pozice první až třetí složky.

Může se zdát kontraproduktivním, že jsou informace o středech také uloženy takto lineárně, ale dává to smysl: paralelizace probíhá na úrovni vstupních obrazových bodů; ve srovnání s nimi je středů řádově méně a stejné výpočty se opakují nad každým z nich zvlášť. Dává tedy smysl mít v paměti tato data u sebe a za pomoci několika instrukcí je při výpočtech s centry shluků distribuovat do registrů a poté je případě opět linearizovat pro uložení do paměti.

Zpracování bodů

Označíme-li sekvenci bajtů ve vstupním poli jako $[r0, g0, b0, r1, g1, b1, r2, g2, b2, r3, g3, b3]$, jednotlivé čtveřice bodů jsou tedy načteny tímto způsobem:

1. Z paměti jsou do *xmm* registru načteny čtyři bajty $[b0, g0, r0, b1]$,
2. tyto bajty jsou instrukcí změněny na čtyři 32b desetinná čísla (nyní s nimi můžeme pracovat pomocí vhodných instrukcí s postfixem `PS` – *packed single*),
3. obdobným způsobem jsou do dalších registrů *xmm* načteny čtveřice $[g1, r1, b2, g2]$ a $[r2, b3, g3, r3]$,
4. pomocí instrukcí přesunu jsou jednotlivé složky použitých registrů přeskládány do dalších *xmm* registrů do podoby $[r0, r1, r2, r3]$ a stejně pro ostatní dvě složky.

S takto uloženými složkami čtveřice bodů je pak jednoduché provádět požadované výpočty. Poté, co jsou pro čtveřici bodů zjištěny nové středy shluků, kterým body přísluší, je nutné barvy bodů přičíst k položkám ve strukturách popisujících tyto středy. Protože jsou uloženy lineárně a neznáme dopředu jejich množství[3], musíme pro každý bod ze čtveřice načíst z paměti odpovídající shluk, opět linearizovat barevné složky bodu, přičíst je k odpovídajícím položkám ve struktuře a tu uložit zpět do paměti.

Výstupem této funkce je naplněné pole celý čísel, kde každé číslo představuje index shluku, ke kterému patří bod na daném indexu tohoto pole. Toho využívá další implementovaná funkce.

Konverze do stupňů šedi

Funkce `algorithm_imgproc` přebírá ukazatel na zdrojová data, ukazatel na cílová data, ukazatel na pole výsledků popsané výše a dvě celá čísla vyjadřující index cílového shluku a počet pixelů. V této funkci byly využity 256b vektory *ymm* technologie AVX, což umožňuje vcelku elegantně zpracovávat dva celé pixely (tj. šest barevných složek) paralelně. Funkce převádí pixely, jejichž hodnota v poli výsledků neodpovídá cílovému indexu shluku, na stupně šedé pomocí známého vztahu, který pro vhodné smíchání jednotlivých složek vychází ze způsobu, jakým lidské oko vnímá barvy [4]:

$$grey = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Právě využití tohoto vztahu znamená, že je opět nutné body, které jsou uloženy lineárně po bajtech, vhodně načíst do registrů a konvertovat na čísla s pohyblivou desetinnou čárkou a zpět. Mohli bychom opět načítat čtveřice a distribuovat jednotlivé složky mezi tři registry, ale zde byl zvolen poněkud jiný přístup, který ponechává složky pixelů ve stejném pořadí a pouze upravuje jejich pozice v rámci vektoru[4].

Je nutné podotknout, že pracujeme s šesti prvky vektoru, který je (v tomto způsobu použití) osmiprvkový, obecně pracujeme s šesti prvky na systému, který je přirozeně založen na mocninách dvojky. Kvůli způsobu, jakým se data ukládají zpět do paměti (uloží se dvojice pixelů, tj. 6 bajtů, a to jako 8B číslo, které má na horních dvou bajtech nuly, načež se ukazatel posune o 6 B doprava), je tedy nutné, aby výstupní pole mělo velikost dělitelnou osmi. To zařizuje obslužný kód v C++.

Testování a srovnání variant

Pro testování programu byl využit především přiložený vstupní obrázek `input2.png`. Pro srovnání variant byl vytvořen jednoduchý skript, který postupně spouští obě přeložené varianty programu (s optimalizacemi i bez nich) s vzrůstajícím počtem shluků a vypisuje počty cyklů ve formátu CSV. *Část* výstupu jednoho takového testovacího běhu je pro demonstraci uvedena v tabulce 1, celý výstup je přiložen v souboru `results.xlsx`.

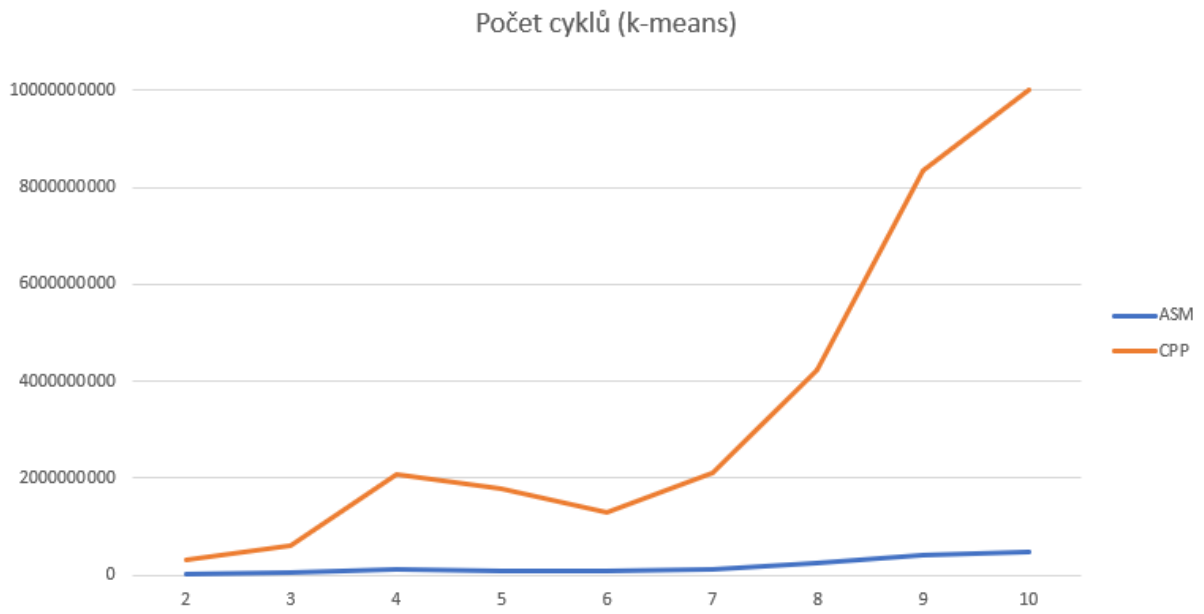
Graf na obrázku 1 ukazuje závislost průměrného počtu cyklů, který program strávil výpočtem shlukování ve funkci `algorithm_kmeans` v jednotlivých variantách (optimalizované a neoptimalizované), na počtu shluků. I když je na něm vidět jistá odchylka pro 4 a 5 shluků, není překvapivé, že se vzrůstajícím počtem cyklů vzrůstá i čas potřebný ke zpracování. Z grafu je ale zjevné, že varianta využívající paralelní zpracování program výrazně urychluje (alespoň při těchto menších počtech shluků, které jsou ale pro tento druh zpracování obrazu vcelku přirozené).

[3] celý tento proces by mohl být výrazně jednodušší, kdybychom měli malý a fixní počet shluků – informace o nich by se tak mohly uchovávat pouze jako složky vektorových registrů

[4] Jedním z důvodů byla také možnost vyzkoušet si i práci s 256b registry.

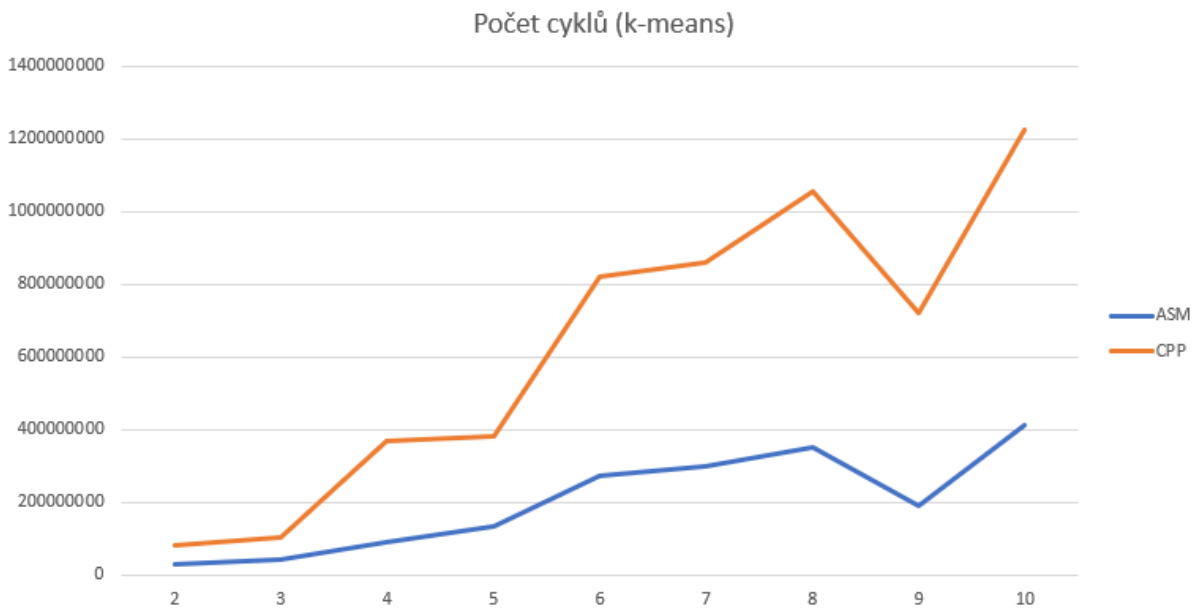
Výpočty nad naměřenými daty ukazují, že optimalizovaná funkce běží průměrně 18,6krát rychleji než neoptimalizovaná varianta (medián je dokonce 19,03). Můžeme tedy předpokládat, že kromě zrychlení způsobeného paralelním výpočtem čtyř prvků najednou jsme dosáhli zrychlení také v jiných oblastech, například díky efektivní práci s pamětí[5].

Pro zajímavost jsme se pokusili obě varianty projektu přeložit také s přepínačem `-O3`, který aktivuje pokročilé optimalizace překladače, a porovnat je stejným způsobem. Výsledek tohoto běhu je na obrázku 2. Je zjevné, že optimalizační schopnosti překladače GCC jsou opravdu výborné – ačkoliv naše ručně optimalizovaná varianta stále podává přibližně třikrát lepší výsledky[6], rozdíl se výrazně zúžil, a pro druhou funkci, `algorithm_imgproc` GCC podává dokonce lepší výsledky, než kterých dosahuje naše varianta.



Obrázek 1: Průměrný počet cyklů strávených ve funkci `algorithm_kmeans` v optimalizované (ASM) a neoptimalizované (CPP) variantě.

[5]To je samozřejmě pouze domněnka a měla by být ověřena jinými metodami, které jsou nad rámec tohoto projektu.
[6]podrobnější srovnání opět v přiložené tabulce `results.xlsx`



Obrázek 2: Průměrný počet cyklů strávených ve funkci `algorithm_kmeans` v optimalizované (ASM) a neoptimalizované (CPP) variantě přeložené s přepínačem `-O3`.

Závěr

Provedli jsme úspěšnou implementaci algoritmu *k-means clustering* pro rozdělování obrázku do volitelného počtu barevných shluků s využitím vektorových instrukcí z rodiny SSE a AVX. Ověřili jsme, že ručně optimalizovaný algoritmus svou rychlostí výrazně předčí stejný algoritmus implementovaný ve vyšším programovacím jazyce (C++), a to i se zapnutými pokročilými optimalizacemi překladače (ačkoliv v tomto případě už není rozdíl tak výrazný).

Reference

1. ZBOŘIL, František V. *Základy umělé inteligence: Studijní text*. Brno: FIT VUT v Brně, 2021.
2. INTEL CORP. *Intel® 64 and IA-32 Architectures Software Developer's Manual (Volume 1: Basic Applications)* [online]. 2016 [cit. 2021-05-11].
   Dostupné z: `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf`.
3. WIKIPEDIA CONTRIBUTORS.
   Advanced Vector Extensions — Wikipedia, The Free Encyclopedia [online]. 2021 [cit. 2021-05-11].
   Dostupné z: `https://en.wikipedia.org/w/index.php?title=Advanced_Vector_Extensions&oldid=1022634846`.
4. HELLAND, Tanner.
   Seven grayscale conversion algorithms (with pseudocode and VB6 source code) [online].
   2011 [cit. 2021-05-11].
   Dostupné z: `https://tannerhelland.com/2011/10/01/grayscale-image-algorithm-vb6.html`.

c	i	asmKmeans	asmProc	cppKmeans	cppProc	kmeansImprov	procImprov
4	0	119353312	2545600	2107733152	7272448	17,66	2,86
4	1	127110177	2521792	2013084128	6824960	15,84	2,71
4	2	119669296	2485088	2089777920	6799520	17,46	2,74
4	3	116190945	2296896	2034701120	6913568	17,51	3,01
5	0	89413088	2537664	1773302432	6886176	19,83	2,71
5	1	108080737	2493568	1818679104	6726208	16,83	2,70
5	2	112225153	2560576	1737597456	6707232	15,48	2,62
5	3	90354848	2482080	1822180271	10727296	20,17	4,32
5	4	90521952	3427776	1745389920	7298432	19,28	2,13
6	0	77759104	2560704	1330999616	7390080	17,12	2,89
6	1	83303680	2611456	1275911296	7431519	15,32	2,85
6	2	69702785	2497376	1288142624	9461216	18,48	3,79
6	3	59510560	2534176	1322309519	7647520	22,22	3,02
6	4	59317152	2711840	1260821952	7352352	21,26	2,71
6	5	61789536	2502080	1262353984	7195392	20,43	2,88
7	0	113382753	2522720	2165495296	7662144	19,10	3,04
7	1	137879248	2868000	2088086336	7221984	15,14	2,52
7	2	110357600	2472128	2126363104	6727488	19,27	2,72
7	3	137058274	2486240	2071466401	8761824	15,11	3,52
7	4	116142144	2539776	2061301853	7289920	17,75	2,87
7	5	111659153	2586560	2094461057	7379264	18,76	2,85
7	6	129763328	2652160	2102010800	7489440	16,20	2,82
8	0	237237376	2493440	4217048128	7434976	17,78	2,98
8	1	227208127	2603712	4353401504	10186560	19,16	3,91
8	2	225098368	2517248	4138898144	10466112	18,39	4,16
8	3	252723297	2467200	4336100480	7324704	17,16	2,97
8	4	246429792	2521152	4261609648	10678208	17,29	4,24
8	5	232909632	2481216	4230968208	7345024	18,17	2,96
8	6	231136771	2504928	4200525200	7568256	18,17	3,02
8	7	256973024	3037377	4209411584	6843200	16,38	2,25
9	0	388320257	2989664	8162693792	7784288	21,02	2,60
9	1	392638657	2510144	8440066528	7311264	21,50	2,91
9	2	376002337	2485280	8177251488	7590432	21,75	3,05
9	3	438527456	2516832	8670391696	7462400	19,77	2,96
9	4	398321281	2547456	8258632607	7689760	20,73	3,02
9	5	423653059	2267552	8273403856	8238208	19,53	3,63
9	6	382823087	2490496	8707215584	7560096	22,74	3,04
9	7	378694113	2530528	8124693616	6867360	21,45	2,71
9	8	423911504	2551008	8213190657	7551520	19,37	2,96

Tabulka 1: Část výstupu testovacího běhu. Sloupec *c* označuje počet shluků, sloupec *i* označuje zvolený shluk, sloupce *asm** a *cpp** označují počet cyklů, které provádění funkce zabralo v jednotlivých implementacích, a sloupce **improv* ukazují podíl hodnot *cpp* a *asm*.