Rychlá detekce rohů FAST-N

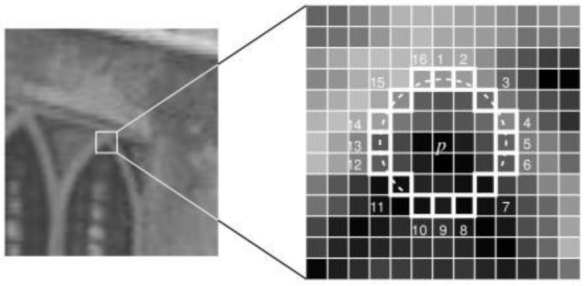
Zdeněk Rozsypálek
rozsyzde@fel.cvut.cz

Závěrečná zpráva projektu k předmětu B4M39GPU
zimní semestr 2018/2019

***Abstrakt*—Semestrální práce se zaměřuje na paralelní implementaci hledání rohů v obrázku za pomoci algoritmu FAST. Jak název napovídá cílem tohoto algoritmu bylo zrychlit hledání rohů, tak aby bylo použitelné v reálném čase i na pomalejším hardwaru. Tento algoritmus se používá zejména pro tvz. *feature detection*, které má mnoho dalších využití. Například algoritmus ORB, který je vhodný pro odometrii z obrazu, používá skóre z algoritmu FAST pro odhad kvality rohu.**

I. Popis algoritmu

Algoritmus se dá rozdělit na tři hlavní části - rychlý test, komplexní test a potlačení sousedů. Algoritmus se provádí pro každý pixel obrázku zvlášť, zde je právě největší prostor pro paralelizaci. Algoritmus je založený na porovnávání hodnot mezi vyšetřovaným pixelem p a pixely na kružnici k kolem něj (viz. obr. 1). Obrázek musí být nejdříve převeden do stupňů šedi a pak můžou následovat tři už zmíněné kroky.



Obrázek 1. Pixely na obklopující kružnici

A. Rychlý test

Tato část algoritmu má vyřadit co největší množství pixelů, které nejsou rohy. Vyberou se čtyři pixely z kružnice (obyčejně 1, 5, 9, 13 na obr. 1) a aspoň pro tři pixely zároveň musí platit podmínka (1) nebo pro tři zároveň podmínka (2):

$$k > p + t \quad (1)$$

$$k < p - t, \quad (2)$$

kde porovnáváme intenzity jednotivých pixelů a hodnota t je uživatelem zvolený *threshold*. Pokud je jedna z podmínek splněna, pixel je vystaven dalšímu testování.

B. Komplexní test

Po vyřazení nevhodných kandidátů je potřeba udělat složitější test, zda je daný pixel rohem. Aby byl pixel rohem, musí na kružnici ležet alespoň π po sobě jdoucích pixelů splňujících podmínku (1) nebo π po sobě jdoucích pixelů splňujícíh podmínku (2). Zde se algoritmus dělí na více druhů, např. pro $\pi = 9$ se algoritmus značí FAST-9. FAST-9 je nejvyužívanější, protože produkuje nejstabilnější rohy, ale někdy se využívá i FAST-12. Pokud je pixel rohem, můžeme mu přiřadit skóre S, které je sumou skóre jednotlivých elementů kružnice splňujících jednu z našich dvou podmínek. Skóre s pro element kružníce vypočítáme jako:

$$s = abs(k - (p + t)) \quad (3)$$

$$S = \sum_{1}^{\pi} s. \quad (4)$$

C. Potlačení sousedů

Obzvláště pro obrázky s vysokým rozlišením je pravděpodobné, že bude vždy více rohů těsně vedle sebe. V tomto kroku algoritmu potlačíme všechny rohy, které vedle sebe mají roh s větším skóre. Obyčejně se používá 3x3 filtr, který vynuluje tyto rohy.

II. CPU implementace

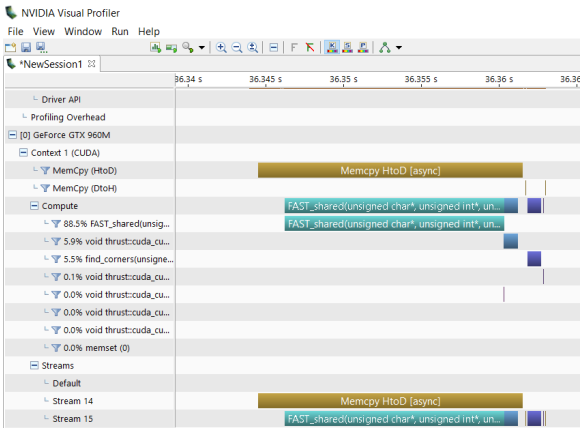
Naivní implementace na CPU je poměrné přímočará. Obrázek budeme iterovat přes každý pixel a to pro každý krok algoritmu. Velkou výhodou tohoto algoritmu je právě rychlý test, který vyřadí opravdu velké množství kandidátů a uvolní tak výpočetní kapacitu, pro jiné potenciální rohy. Komplexnější řešení je použité v *open-source* knihovně OpenCV, která algoritmus implementuje za pomocí bitových operací a SIMD instrukcí. Tato implementace je minimálně o řád rychlejší než naivní implementace.

III. GPU IMPLEMENTACE

Pro implementaci na GPU je vhodné paralelizovat výpočty pro každý pixel zvlášť. Při implementaci jsme zjistili, že při vysoké míře paralelizace je rychlý test naopak nevýhodou a algoritmus pouze zpomaluje. Při vyřazení rohu totiž nejsou uvolněny žádné výpočetní prostředky a vlákno pouze čeká na synchronizaci s těmi, které provádějí komplexní test.

IV. IMPLEMENTAČNÍ DETAILY – CUDA

Pro GPU máme celkem dvě verze kernelu FAST. Jedna verze využívá pouze globální paměť a druhá verze využívá paměť sdílenou. Výstupem obou verzí tohoto kernelu je pole, které obsahuje skóre jednotlivých pixelů. Na toto pole jsme použili paralelní scan, abychom získali indexy jednotlivých rohů. Ty jsme dále paralelně seřadili a zakreslili do obrázku. Zajímavější je ovšem implementace pro videosoubor. Hlavním problémem GPU je, že se data musí z RAM nahrát do paměti GPU. Abychom zajistili větší konkurenci kernelu a kopírování paměti, využili jsme CUDA *stream*. Takto je možné asynchroně hledat rohy a zároveň plnit paměť GPU.



Obrázek 2. Screenshot z Nvidia profiler, kde můžeme vidět vysokou konkurenci kernelu a kopírování paměti.

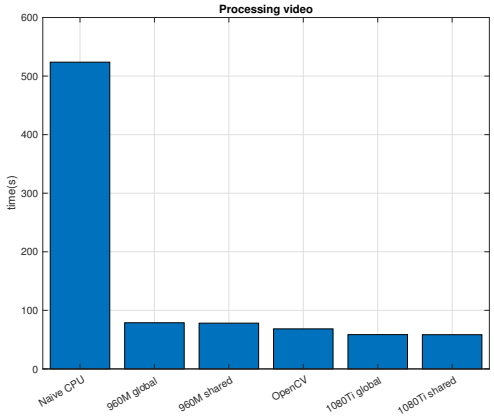
Další výhodou naší implementace je, že můžeme přímo získat skóre rohu. V aplikaci je to znázorněno barvou rohu. Červený roh získal nízké skóre, zelený roh získal skóre vysoké. Program se dá spouštět s velkým množstvím přepínačů a v různých módech, což je zdokumentováno v *readme.md* na verzovacím systému projektu.

V. NAMĚŘENÉ VÝSLEDKY

Náš program jsme testovali na dvou strojích, jeden na přenosném PC s GPU 960M a druhý na podstatně výkonějším stroji s GPU 1080Ti. Při vývoji jsme používali nejrůznější obrázky abychom zjistil, že detekce rohů

Obrázek 3. Příklad výstupu programu.

funguje správně. Dále jsme použili 4k video s necelými 600 snímky jako benchmark rychlosti. Během testování se ukázalo, že verze s globální pamětí a sdílenou pamětí se rychlostně příliš neliší. Verze se sdílenou pamětí je zanedbatelně rychlejší. Dále je vidět, že GPU implementace je řádově rychlejší než naivní CPU implementace, ovšem optimální CPU implementace z knihovny OpenCV byla rychlejší než GPU verze na kartě 960M. Další zajímavostí je, že rychlost obou CPU implementací klesá s rostoucím počtem nalezených rohů, zatímco na GPU je rychlost více stabilní.



Obrázek 4. Doba běhu aplikace na našem benchmarku.

VI. ZÁVĚR

Paralelní implementace přinesla požadované zrychlení oproti naivní GPU implementaci. Pokud bychom video nezapisovali na disk, bylo by možné na výkonějším GPU zpracovávat i 4k video v reálném čase. Jelikož naše GPU implementace vychází z naivní CPU implementace, je zde pravděpodobně hodně prostoru pro zlepšení. Jak je vidět z OpenCV implementace, většina výpočtů v kernelu se dá nahradit bitovými operacemi.

REFERENCE

[1] Edward Rosten: https://arxiv.org/pdf/0810.2434.pdf
[2] OpenCV: https://github.com/opencv/opencv/blob/master/modules/features2d/src/fast.cpp