

Implementace Cannyho detektoru hran na GPU pomocí CUDA

Radek Cichra

24. dubna 2025

Obsah

1 Definice problému	2
2 Popis sekvenční verze algoritmu	2
2.1 Převod obrázku na odstín šedi	3
2.2 Aplikace Gaussova filtru (rozostření)	3
2.3 Výpočet gradientu intenzity a směru (Sobelův filtr)	3
2.4 Potlačení ne-maxim	4
2.5 Dvojité prahování	4
2.6 Sledování hran pomocí hystereze	5
3 Použité knihovny	5
4 GPU implementace a analýza	5
4.1 Změny v algoritmu při přechodu na GPU	5
4.2 Optimalizace a poznámky	6
4.3 Výsledky měření	6
5 Závěr	9
6 Použitá literatura	9
7 Příloha: zdrojové soubory a výstupy měření	9

1 Definice problému

Úkolem bylo implementovat algoritmus Cannyho detektoru hran pro zpracování obrazových dat běžící na GPU s využitím Nvidia CUDA. Zpracování obrazových dat na GPU je velmi praktické a běžně používané – extrakce hran je důležitou součástí pipeline v oblasti AI, počítačového vidění nebo např. rozpoznávání objektů v autonomních systémech.

Smyslem úkolu bylo

- analyzovat vhodnost algoritmu pro paralelizaci,
- navrhnout a implementovat jednotlivé fáze algoritmu,
- změřit a porovnat výkonnost mezi sekvenční a paralelní variantou.

2 Popis sekvenční verze algoritmu

Použitá sekvenční verze algoritmu implementuje klasické fáze Cannyho detektoru (implementace v příloze v adresáři `src/CPU`):

1. Převod obrázku na odstíny šedi
2. Aplikace Gaussova filtru (rozostření)
3. Výpočet gradientu intenzity a směru (Sobelův filtr)
4. Potlačení ne-maxim
5. Dvojité prahování
6. Sledování hran pomocí hystereze

Sekvenční kód byl postaven s `-O3`.

Vizualizace jednotlivých kroků algoritmu budou ilustrovány na demonstračním obrázku:



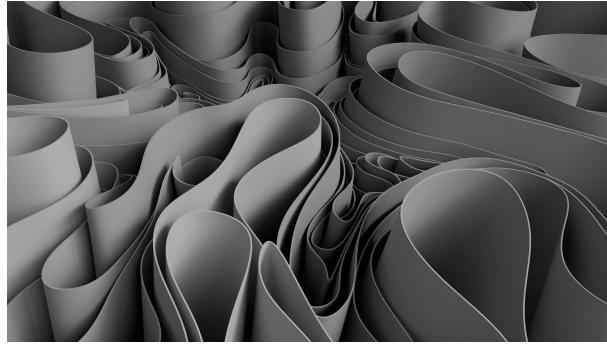
Obrázek 1: Odkaz

2.1 Převod obrázku na odstíny šedi

Každý pixel barevného obrázku (ve formátu BGR) je převeden na odstín šedi pomocí váženého průměru intenzit jednotlivých barevných kanálů. Použitý vzorec odpovídá standardu:

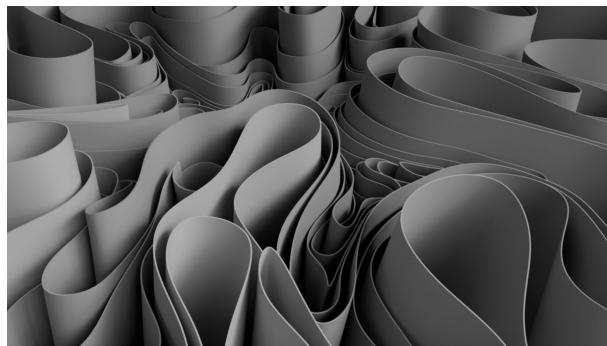
$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Tímto vznikne jednokanálový obraz, vhodný pro další zpracování.



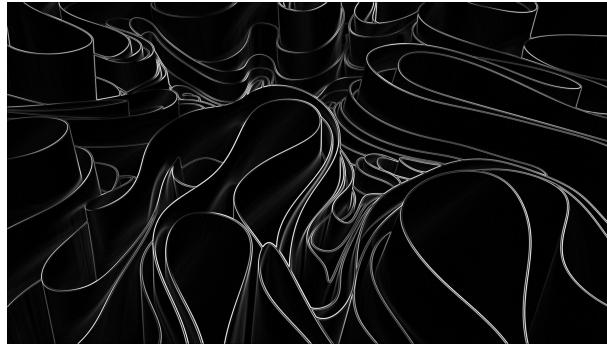
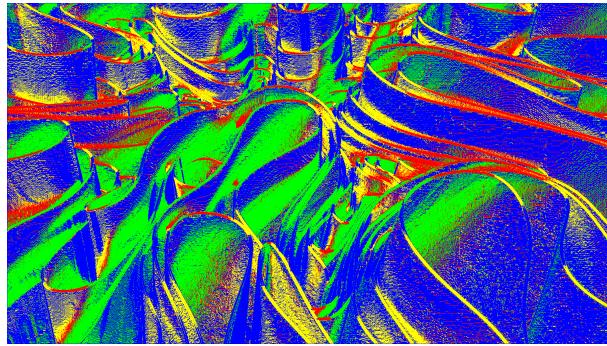
2.2 Aplikace Gaussova filtru (rozostření)

Gaussův filtr je použit k odstranění vysokofrekvenčního šumu, který by mohl ovlivnit detekci hran. V každém bodě obrazu se počítá vážený průměr okolních pixelů podle Gaussova rozdělení (v našem případě 5×5 kernel). Výsledkem je hladší obraz se zachováním hlavních struktur.



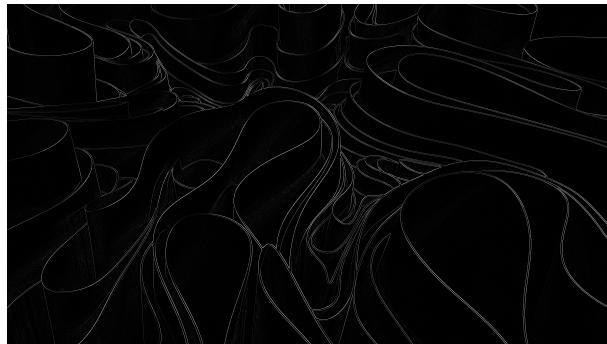
2.3 Výpočet gradientu intenzity a směru (Sobelův filtr)

Sobelův filtr slouží k detekci změn intenzity – tedy hran. Ve vodorovném a svislém směru jsou aplikovány konvoluční masky (Sobelovo jádro), které udávají směr a velikost gradientu v každém bodě. Z těchto hodnot se pak vypočítá velikost a kvantizovaný směr gradientu. Rozeznáváme 4 směry gradientu, které lze ilustrovat různým zabarvením.



2.4 Potlačení ne-maxim

V této fázi dochází k tenkému zvýraznění hran. V každém bodě se porovnává velikost gradientu s jeho sousedy ve směru gradientu. Pokud není v daném směru lokálním maximem, je hodnota nastavena na nulu. Tím se výrazně omezí šířka detekovaných hran.



2.5 Dvojité prahování

Dvojité prahování klasifikuje pixely na:

- **silné hrany** (nad horním prahem),
- **slabé hrany** (mezi oběma prahy),
- **nehrany** (pod dolním prahem).

Tím se oddělí jasně definované hrany od šumu a nejistých přechodů. Je třeba podotknout, že pro reálné použití je vhodné až nutné implementovat nějaký dynamický způsob nastavení prahů v závislosti na histogramu z hodnot pixelů vstupního obrázku (Otsu method). V naší implementaci jsme nastavili pevné, obecně dobře fungující hodnoty.



2.6 Sledování hran pomocí hystereze

Slabé hrany, které jsou spojeny se silnými hranami, jsou zpětně přeměněny na silné. Ostatní slabé hrany jsou odstraněny. Tímto způsobem se eliminuje falešná detekce hran, která není spojena s jasnou strukturou v obrazu. Výstup z této fáze je finálním výstupem algoritmu.



3 Použité knihovny

Všechny části algoritmu byly implementovány bez použití knihovních funkcí (kromě I/O zajištěného přes OpenCV). OpenCV bylo použito pouze pro načítání a ukládání obrázků – veškeré operace (konvoluce, filtry, prahování) byly implementovány ručně.

4 GPU implementace a analýza

Díky dobře rozdělené struktuře jednotlivých fází bylo poměrně snadné algoritmus převést do podoby GPU verze, kde každá fáze byla převedena jako samostatný CUDA kernel.

4.1 Změny v algoritmu při přechodu na GPU

Na teoretické úrovni nebylo třeba zásadně měnit strukturu algoritmu – každá fáze pracuje s vlastní částí dat a může být zpracována paralelně. Výpočetně nejnáročnější části (konvoluce, filtry, prahování) jsou přirozeným kandidátem na masivní paralelizaci – každý pixel lze počítat nezávisle.

Prakticky se implementace převedla do jednotlivých CUDA kernelů pro každou fázi. Gridy a bloky byly zvoleny na základě rozměrů obrázku. Například:

- Konvoluční operace (Gauss, Sobel) byly implementovány jako 2D jádra s přístupem do okolních pixelů.
- Grayscale konverze byla provedena jako 1D kernel, který iteroval přes BGR vstup.

Celá implementace je v příloze v `src/GPU`.

4.2 Optimalizace a poznámky

CUDA kód byl komplikován s přepínačem `-use_fast_math`, který umožňuje použití zrychlených matematických funkcí (např. approximace `sqrtf`, `atan2f`) za cenu potenciálních nepřesností.

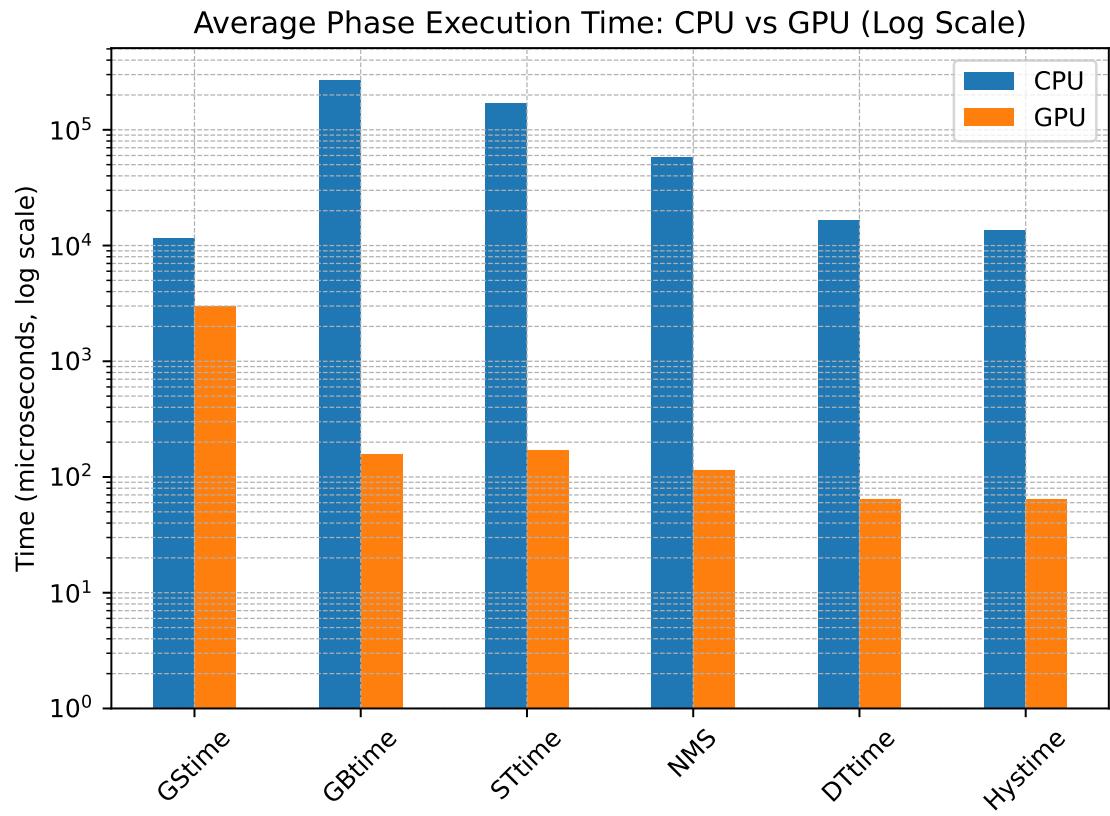
V naší implementaci to však není problém:

- Výpočty neakumulují chybu – každý pixel se počítá nezávisle.
- Neprovádí se iterované sumace, které by byly citlivé na asociativitu.
- Jediným citlivějším místem je kvantizace směru gradientu, který se ale zaokrouhuje k jednomu ze 4 rozeznávaných směrů, tedy i celkem velká chyba (až 45 stupňů) se stejně neprojeví.

4.3 Výsledky měření

Bylo testováno 126 obrázků různé velikosti (náhodné rozměry $100 - 15000 \times 100 - 15000$ pixelů) získaných s pomocí služby Lorem Ipsum a Unsplash. Měření probíhalo odděleně pro každou fázi algoritmu.

Následující graf ukazuje porovnání času stráveném v každé části algoritmu zprůměrovaného ze všech 126 obrázků. Výsledky potvrdily výrazné zrychlení GPU varianty – typicky o cca 3 řády oproti CPU.



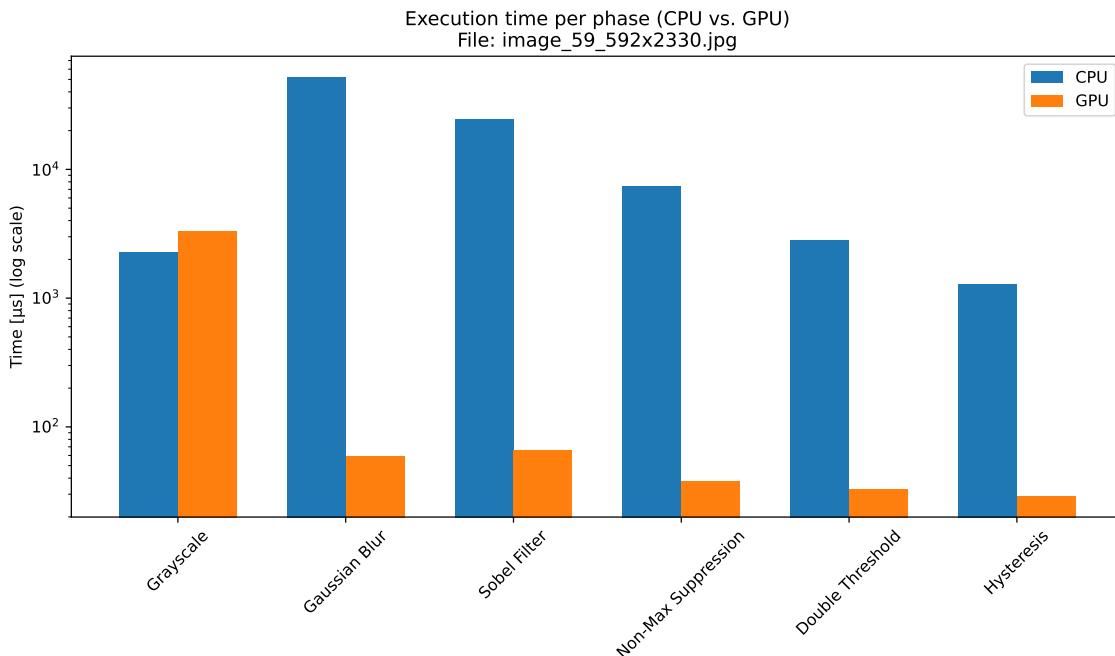
Obrázek 2: Porovnání průměrných časů jednotlivých fází algoritmu na CPU a GPU.

Při pohledu na závislost velikosti obrázku na čase můžeme vidět, že pro malé obrázky je CPU lepší, neboť nemá overhead spojený s GPU kernel launchy apod. Jak je ale z grafu vidět, škálování do velikosti je mnohem lepší pro GPU.



Obrázek 3: Porovnání časů běhů na CPU a GPU v kontextu velikosti obrázků.

Poslední graf ukazuje porovnání na jednom náhodně vybraném obrázku (v `src/scripts/` možnost vygenerovat jiný).



Obrázek 4: Porovnání časů běhů na CPU a GPU na jednom obrázku.

Zajímavostí je rozdílný bottleneck:

- **CPU:** Nejvíce času zabere Gaussova konvoluce – opakované čtení a sumace přes okolí každého pixelu.
- **GPU:** Nejdelší trvání má paradoxně konverze do odstínů šedi – jediná operace, která zpracovává celý obrázek ve 3 kanálech (BGR). Všechny další fáze už operují pouze s jedním kanálem.

Uvažováno bylo i o optimalizaci grayscale převodu pomocí většího počtu threadů (např. 3 per pixel pro každý kanál), ale overhead s tím spojený (alokace a agregace – nutně další kernel launch) by pravděpodobně výhodu negoval.

5 Závěr

Projekt implementace algoritmu Cannyho detektoru hran na GPU pomocí CUDA byl velmi přínosný. Šlo o typický průchod reálnou problémovou pipeline, kde bylo možné uplatnit paralelizaci a výkonnostní optimalizace.

Projekt potvrdil, že struktura algoritmu je velmi dobře paralelizovatelná. Přímý přepis z CPU verze (překlopení smyček do threadů) přinesl výrazné zrychlení, přičemž i jednoduchá varianta bez sdílené paměti nebo komplikovaných optimalizací vedla k výsledku, který je prakticky využitelný.

Zvláště přínosné bylo sledování odlišného chování mezi CPU a GPU – např. jiného výkonového bottlenecku.

6 Použitá literatura

- Canny Edge Detector – Wikipedia: https://en.wikipedia.org/wiki/Canny_edge_detector
- OpenCV – oficiální dokumentace: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html
- Lorem Picsum – náhodné testovací obrázky: <https://picsum.photos>
- Unsplash – reálné obrázky ve vysokém rozlišení: <https://unsplash.com>

7 Příloha: zdrojové soubory a výstupy měření

K reportu je přiložen kompletní projekt ve formě adresářové struktury, která obsahuje jak zdrojové kódy implementace, tak i výsledky měření v podobě výstupních tabulek.

Struktura příloh

- **src/** – adresář obsahující zdrojové kódy implementace algoritmu:

- **src/CPU/** – sekvenční implementace algoritmu pro běh na CPU v jazyce C++ s využitím OpenCV pouze pro vstup a výstup obrázků.
 - **src/GPU/** – paralelní implementace pomocí CUDA, optimalizovaná s využitím `-use_fast_math`, obsahuje jednotlivé CUDA kernele odpovídající jednotlivým fázím algoritmu.
 - **src/scripts/** - scripty použité pro zpracování dat, scrape obrázku atd.
- **out_logs/** – adresář obsahující výstupní logy a měření běhu obou verzí algoritmu:
 - `canny_cpu.csv` – tabulka obsahující naměřené časy všech fází pro jednotlivé obrázky při běhu na CPU.
 - `canny_gpu.csv` – analogická tabulka pro běh na GPU.

Každý CSV soubor obsahuje následující sloupce: název souboru, velikost vstupního obrázku, časy jednotlivých fází (grayscale, gauss, sobel, non-max suppression, double threshold, hysteresis) a celkový čas běhu. Hodnoty jsou uvedeny v mikrosekundách.