Semestrální projekt MI-PRC.16 2017/2018
Cannyho detekce hran na platformě CUDA

Martin Černáč, cernama9

11. května 2017

1 Definice problému a popis programu

Úvodní kapitola shrnuje definici řešeného problému a myšlenku sekvenčního přístupu k řešení úlohy. Zabývá se rovněž implementačními detaily, jako je formát vstupu, formát výstupu a další.

1.1 Cannyho detekce hran

Detekce hran je postup v digitálním zpracování obrazu, který slouží k nalezení oblastí pixelů, ve kterých se podstatně mění jas. Cannyho hranový detektor je algoritmus zahrnující několik kroků pro získání co nejlepšího výsledku při detekci hran v dvourozměrném diskrétním obraze. Byl formulován na začátku osmdesátých let J. F. Cannym, který jako cíle ideální hranového detektoru (požadavky detekce) stanovil [1]:

- Minimální počet chyb (musí být detekovány všechny hrany, nesmí být detekována místa, která hranami nejsou)
- Přesnost (poloha hrany musí být určena co nejpřesněji)
- Jednoznačnost (odezva na jednu hranu musí být jedna, nesmí docházet ke zdvojení)

Algoritmus, který Canny navrhl, tyto požadavky splňuje a má dnes velké zastoupení v řešení úlohy detekce hran v obraze. Své uplatnění nachází zejména v oblasti počítačového vidění, kde se jedná o jeden z prvních kroků zpracování obrazu. Samotný algoritmus se skládá z několika navazujících kroků:

1. Potlačení šumu (Gaussův filtr)
2. Výpočet gradientu (derivace)
3. Hledání lokálních minim (thinning)
4. Eliminace nevýznamných hran (thresholding / hystersis)

1.2 Vstup programu

Program umí zpracovávat obrázky ve formátu BMP[1] – tedy s 24-bitovou barevnou hloubkou. Toto omezení je aplikováno z důvodů nezávislosti na knihovnách pro podporu obrazu – formát BMP je velmi známý a jednoduchý formát.

Samotná detekce hran probíhá na modifikovaném vstupním obrázku (převod na stupně šedi). Převod RGB kanálů do odstínů šedi probíhá dle normy[2, BT.709]:

$$R_{coef} = 0.2126, G_{coef} = 0.7152, B_{coef} = 0.0722$$

Konverze je aplikována už při načítání dat, což šetří paměťové nároky (na třetinu, vezmeme-li v potaz velikost vstupního obrázku).

[1] Windows Bitmap Image

1.3 Výstup programu

Výstupem programu je opět obrázek ve formátu BMP – rovněž s 24-bitovou hloubkou (pro zachování co největší kompatibility). Obrázek však využívá jen 8 bitů možné informace na pixel – a to pro zvýraznění hran v místech původního obrázku. Hrany jsou znázorněny bílou barvou, zbytek obrazu je vyplněn černou barvou.

1.3.1 Ukázka výstupu programu

Pro zhodnocení výstupu jsem si nemohl odpustit použít velmi známý obrázek pro testování (nejen) komprese. Následující dva obrázky tedy zobrazují vstup 1 a výstup 2 programu (parametry hystereze $t_{min} = 40, t_{max} = 80$.

Obrázek 1: Lena Söderberg, 1972

Obrázek 2: detekované hrany v obrázku

2 Popis sekvenčního algoritmu a jeho implementace

Tato kapitola popisuje klasický (sekvenční) algoritmus Cannyho detekce hran a jeho implementaci v jazyce C (sekvenční).

2.1 Rozostření obrazu

Jako první krok algoritmu je vhodné aplikovat Gaussův filtr pro jemné rozostření obrazu a tím pádem potlačení šumu v obraze. Algoritmus je implementován jako konvoluce obrazu s 5×5 kernelem3

$$\frac{1}{159} \cdot \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Obrázek 3: 5×5 kernel Gausovského filtru (odpovídá parametru $\sigma = 1.4$)

Větší velikostí kernelu, nebo případně opakovaným použitím filtru, lze dosáhnout rovnoměrnějšího rozostření – což ovšem není cílem pro funkci algoritmu. Naopak, přílišné rozostření může způsobit, že ani výrazné hrany nebudou správně detekovány. Myšlenka přímočarého algoritmu je zachycena v následujícím pseudokódu 1:

```
1  for y = 0 to vyska do
2    for x = 0 to sirka do
3      suma = 0
4      for i = -filtr_vyska to filtr_vyska do
5        for j = -filtr_sirka to filtr_sirka do
6          //je-li mozne cist obrazek[x - j][y - i]
7          suma += filtr[j][i] * obrazek[x - j][y - i]
8      end
9      vystup[x][y] = suma / vaha
10   end
11 end
```

Listing 1: Pseudokód přímočaré konvoluce

2.1.1 Ukázka kroku algoritmu

Prvním krokem algoritmu je pouhé rozostření, takže výsledek prvního kroku algoritmu není nikterak překvapivý.

Obrázek 4: vstupní obrázek

Obrázek 5: aplikace rozostření

2.2 Výpočet gradientů (intenzit) obrazu

V tomto kroku jsou na obraz aplikovány Sobelovy směrové filtry (opět konvoluce, 3×3) – v horizontálním a vertikálním směru.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Obrázek 6: 3×3 kernel Sobelova filtru pro horizontální a vertikální směry

Intenzitu gradientu je pak možné nalézt pomocí předpisu

$$G = \sqrt{G_x^2 + G_y^2}$$

Směr gradientu nalezneme pomocí přepisu

$$\Theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Směr gradientu je užitečná informace pro hledání lokálních maxim. Z hlediska přesnosti výpočtu nám postačí rozlišovat mezi osmi "kvadranty" (tedy intervaly mezi 0 a 45 stupni, 45 a 90, ...).

Sekvenční kód opět využívá algoritmu konvoluce. K výpočtu intenzity gradientu byla použita funkce `hypot(x,y)`, kterou lze matematicky definovat jako

$$hypot(x, y) := \sqrt{x^2 + y^2}$$

Pro výpočet směru gradientu byla použita matematická funkce `atan2(x,y)`, která opět přesně splňuje požadavky algoritmu. Její výstup je však v intervalu $(-\frac{\pi}{2}; \frac{\pi}{2})$, a proto je za účelem rozdělení do 8 "kvadrantů" její výstup přeškálovat. Tuto operaci zachycuje řádka kódu 2.

```
1 const float dir = fmod(atan2(gradientY, gradientX) + M_PI, M_PI) / M_PI) * 8;
```

Listing 2: Přeškálování výstupu výpočtu směru gradientu

2.2.1 Ukázka kroku algoritmu

Tento krok vypočítává intenzitu a směr gradientu. Vizuálně přiložen je pouze obrázek intenzity gradientu a složek pro jeho výpočet.

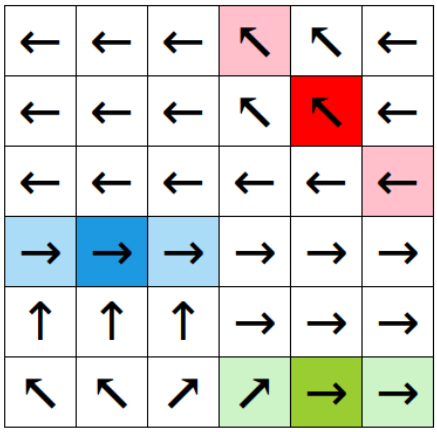
Obrázek 7: $G_x gradient$

Obrázek 8: $G_y gradient$

Obrázek 9: Intenzita gradientu

2.3 Výpočet gradientů (intenzit) obrazu

V tomto kroku jsou za pomoci nalezených intenzit a směrů gradientu z předchozího kroku potlačeny všechny body, které nejsou lokálním maximem. Díky informaci o směru gradientu a jeho diskretizaci na 8 možných směrů tak stačí porovnat hodnotu každého bodu obrazu s dvěma sousedními body podle směru gradientu. Tuto operaci znázorňuje obrázek 10:



Obrázek 10: Potlačení lokálních non-maxim – operace porovnání. Sytěji vybarvené body jsou porovnány se slaběji vybarvenými body stejné barvy

Sekvenční kód této operace je triviální (jedná se o 4 podmínky a zápis do výstupního pole obrazu)- Pseudokód pro tento krok je tedy z technické zprávy vynechán.

Má-li porovnávaný bod největší gradient ze všech tří porovnávaných bodů, je zachován. V opačném případě je namísto hodnoty gradientu zapsána hodnota 0. Tuto operaci lze nalézt pod pojmem “thinning”, tedy *prořídnutí*. Název vychází z efektu operace – z dat jsou odděleny *zavádějící* hrany, nebo-li body s nedostatečně intenzivním gradientem.

2.3.1 Ukázka kroku algoritmu

Efekt *thinningu* ilustrují následující obrázky, vstup a výstup operace potlačení lokálních non-maxim.

Obrázek 11: Vizualizace intenzity gradientů (z minulého kroku algoritmu)

Obrázek 12: Vizualizace obrazu po aplikaci operace *thinningu*

2.4 Hystereze obrazu

Posledním krokem algoritmu je hystereze za pomoci dvou prahů (horního a dolního). Operace pro každý bod obrazu na základě intenzity jeho gradientu rozhodne, zda se jedná o součást hrany, nebo pouze o šum (a bod jako takový bude potlačen). Operace se řídí třemi jednoduchými pravidly:

1. Pokud je intenzita gradientu zkoumaného bodu **vyšší**, než **horní práh**, bod je přijat jako součást hrany
2. Pokud je intenzita gradientu zkoumaného bodu **nižší**, než **dolní práh**, bod je potlačen (není součástí hrany)
3. Pokud je intenzita gradientu zkoumaného bodu **v intervalu mezi dvěma prahy**, bude přijat **pouze** pokud sousedí s jiným již přijatým bodem. Každý bod obrazu má nejvýše 8 sousedů.

Myšlenka implementovaného sekvenčního algoritmu je zachycena v následujícím pseudokódu 3:

```
1  hrany = queue.init()
2  for y = 0 to vyska do
3    for x = 0 to sirka do
4      if vstup[x][y] < tmax || vystup[x][y] != 0 then continue
5      vystup[x][y] = 1
6      hrany.push({x,y})
7      while (!hrany.isEmpty()) do
8        bod = hrany.pop()
9        for s : bod.sousede() do
10         if vstup[s.x][s.y] >= tmax && vystup[s.x][s.y] == 0 then
11           vystup[x][y] = 1
12           hrany.push(s)
13         end
14       end
15     end
16   end
17 end
```

Listing 3: Pseudokód přímočaré hystereze

2.4.1 Ukázka kroku algoritmu

Finální krok algoritmu vyčistí obraz od *slabých* hran a ponechá pouze dostatečně výrazné hrany, dle zvoleného dolního a horního prahu.

Obrázek 13: Vizualizace obrazu po aplikaci operace *thinningu*

Obrázek 14: Finální podoba obrazu po aplikaci operace hystereze

3 Popis paralelního algoritmu a jeho implementace

Tato kapitola shrnuje volby a implementaci prvků algoritmu. Zachycuje mou snahu o přizpůsobení algoritmu platformě CUDA a o využití jejich výhod pro běh algoritmu.

3.1 Datové typy a struktury

V rámci přizpůsobení algoritmu platformě CUDA jsem nebyl nucen vytvářet žádné nové struktury. Ve vyjádření dat zpracovávaného obrazu jsem se omezil na datové typy `char` a případně `float`. Algoritmus jako takový nepotřebuje velkou přesnost výpočtů v plovoucí desetinné čárce a vzhledem k faktu, že Sobelovský operátor je grayscale operátor, není nutný ani velký rozsah hodnot pro bod obrazu (postačí 8-bitů, tedy `char`). Řada intristik je navíc dostupná jen pro výpočty v `single` přesnosti a volba datového type `float` je tím dále podložena.

3.2 Použité typy pamětí

V implementaci jsem se pokusil využít různých typů pamětí v závislosti na jejich vlastnostech vhodných ke konkrétnímu účelu práce programu.

3.2.1 Texture memory

Paměť pro textury je přístupná pro všechna vlákna a pro hostující stroj. Z pohledu grafické karty se jedná se o paměť pouze pro čtení. Paměť je během běhu algoritmu značně utilizována, protože je do ní po krocích ukládán zpracovávaný obraz. Hlavním důvodem pro použití texturovací paměti pro data zpracovávaného obrazu jsou:

- "2D" Cache optimalizovaná pro grafické operace (*2D spatial locality*)
- implicitní schopnosti pro zacházení s okraji obrázku
- bez nutnosti koherenčního protokolu (pouze čtení)

Což splňuje naše požadavky pro běh algoritmu. Dostupná literatura tento závěr potvrzuje [3, Článek z konference, COMMUNICATION-MINIMIZING 2D CONVOLUTION].

Pro další snížení paměťové náročnosti programu jsou v paměti textur uloženy celkem dvě položky – obě vyjadřují stav zpracovávaného obrazu, ale liší se datovým typem. Pro předzpracování obrazu (Gaussovské rozostření a počítání gradientů) je postačující datový typ `char`, který snižuje paměťovou náročnost uložení obrazu na čtvrtinu, protože `sizeof(char) = 1` a `sizeof(float) = 4`.

Protože texturovací paměť slouží pouze ke čtení, je třeba aktualizovat její obsah ze strany CPU. Data však lze zapsat i z paměti grafické karty:

```
1 void CUDARebindTextureFloat(float *devIn, const unsigned int size) {
2   cudaUnbindTexture(devImageTextureFloat);
3   cudaMemcpyToArray(devImageFloat, 0, 0, devIn, size, cudaMemcpyDeviceToDevice);
4   cudaBindTextureToArray(devImageTextureFloat, devImageFloat);
5 }
```

Listing 4: Funkce pro aktualizaci položky v texturovací paměti – zápis z device memory do device memory

Texturovací paměť umožňuje definovat chování při pokusu o čtení dat z oblasti mimo obrázek. Tato funkcionalita je velice vhodná pro implementaci konvoluce, protože tím pádem eliminuje nutné podmínky v kódu vlákna a tím pádem snižuje divergence warpů. V současné době je možno použít jeden ze čtyř možných módů zacházení s požadavky na čtení mimo obrázek, a to sice: [5]

- `border` – neexistující data jsou nahrazena nulami
- `wrap` – simuluje čtení z dlaždic vytvořených z původních dat

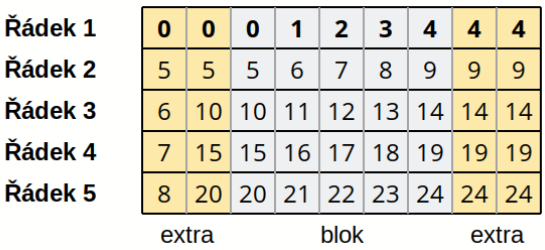
- `mirror` – zrcadlově se vrací zpět do původních dat obrázku
- `clamp` – neexistující data jsou nahrazena nejbližší platnou hodnotou

Jako adresní mód jsem v práci zvolil `clamp`, protože tím pádem nevznikne ostrý přechod na hranách obrazu (důležité u výpočtu gradientů).

3.2.2 Shared memory

Sdílenou paměť se mi v práci podařilo efektivně využít v implementaci separovaného Gaussova filtru – kdy se první 1D konvoluce počítá z paměti textur a druhý průchod 1D konvoluce z dat napočítaných prvním průchodem (uloženo v globální paměti) nakešovaných do sdílené paměti.

Protože z hostujícího stroje je nutné nakopírovat počáteční podobu obrazu pro zpracování, zvolil jsem pro první průchod 1D konvoluce Gaussova filtru texturovací paměť. Experimentoval jsem s nahráním obrazu do globální paměti a kešováním pomocí sdílené paměti podobně jako při druhém průchodu, ale tento postup nepřinesl výkonnostní zlepšení.



Obrázek 15: Diagram znázorňující kešování bodů obrazu a *apron* (extra) bodů pro konvoluci (čísla označují konkrétní vlákno)

Kešování při průběhu separované konvoluce znázorňuje diagram 15. Během implementace jsem experimentoval i s načítáním pouze jednoho extra bodu na vlákno, ale jako výkonněji se ukázala varianta, kde pouze krajní vlákna načítají dva extra body. Předpokládám, že při větší velikosti kernelu by rozdělení režie extra bodů přineslo výkonnostní zlepšení.

3.2.3 Registers

V implementaci používám deklaraci `register` u proměnných, se kterými se hodně pracuje. Vliv každé takové deklarace jsem ověřil měřením času (průměr ze sta). V programu jsem filtry pro vyšší výkon zadrátoval přímo do kódu, namísto uložení konvoluční masky do registrů vláken – data použitých masek jsou k tomu vhodná, protože se jedná o řádově desítky položek hodnot velmi malých rozsahů. Dle [3] lze tímto postupem dosáhnout vyššího výkonu pro zpracování obrazu.

Užití registrů jsem zkontroloval pomocí kompilace s přepínačem `-Xptxas="-v"` (a při měření v profileru).

3.3 Single Precision Intrinsics

CUDA API má k dispozici celou řadu méně přesných, ale výkonnějších operací nad čísly pohyblivé desetinné čárky.

Například při výpočtu směru gradientu je efektivnější nespoléhat na standardní funkci `atan2f`, ve tvaru

```
1    atan2f(accY,accX)
```

Listing 5: Použití funkce `atan2f`

Oproti tomu je výrazně výkonnější použít volání ve tvaru

```
1    atanf(__fdividef(accY,accX)
```

Listing 6: Vydělení veličin pomocí intristiky a následné použití funkce `atanf`

protože v Cannyho algoritmu na přesnosti výstupu příliš nezáleží, výstupní úhel totiž stejně uvažujeme jen v intervalech 45 stupňů.

Obdobně při následném zpracování výstupu je použita celá řada intristik:

```
1    __fmul_rd(__fdiv_rd(__fadd_rd(atanf(__fdividef(accY,accX)),M_PI),M_PI),8);
```

Listing 7: Kód pro *binning* výstupu atanf do osmi disjunktních intervalů

3.4 Separable filter

Pro některé filtry platí, že je lze rozložit na součin dvou matic. Příklad separable filtru je Sobelův filtr, tedy: Platí, že G_x a G_y lze rozepsat na součin dvou matic dle přepisu

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} +1 \\ +2 \\ +1 \end{bmatrix} \cdot \begin{bmatrix} +1 & 0 & -1 \end{bmatrix}$$

a analogicky

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = \begin{bmatrix} +1 \\ 0 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} +1 & +2 & +1 \end{bmatrix}$$

Výsledkem je rozložení 2D konvoluce na dvě 1D konvoluce. Výhodou postupného výpočtu pomocí separovaného Sobelova filtru je menší velikost filtru, navíc i zmenšení uvažovaných okrajů zpracovávaného obrazu a tím pádem výrazně nižší počet aritmetických operací na jeden bod obrazu.

Stejným způsobem je možno rozložit kernel Gaussova filtru[4, On-line kalkulačka]. V programu pro výpočet 2D konvoluce Gaussova filtru používám konstanty `0.113318, 0.236003, 0.30136, 0.236003, 0.113318` – jedná se o separovaný filtr Gaussova rozostření pro parametr $\sigma = 1.4$.

V kódu jsem ponechal i původní implementaci neseparované 2D konvoluce, na které jsem aplikoval loop unrolling, zadrátování kernelu a použití intristiky.

3.5 Metody transformace zdrojových kódů

Při implementaci algoritmu na grafickou kartou jsem se transformací zdrojových kódů snažil dosáhnout vyššího výkonu.

Loop unrolling Namísto direktivy `pragma unroll` jsem cykly rozbaloval sám, protože jsem chtěl mít jistotu, že smyčka opravdu bude rozbalena. Takto jsem postupoval například u prvního kroku algoritmu (Gaussův filtr) a druhém (výpočet gradientů, aplikace Sobelova operátoru). Vyšší volnost v pořadí vykonávání příkazů navíc může pomoci s optimalizací čtení z paměti (u texturovací paměti funguje cache).

Odstranění konstant Zadrátováním konstant se sice snižuje čitelnost kódu, ale experimentálně jsem ověřil výkonnostní zisk. Jednalo se především o konvoluční masky, které jsem před tímto krokem ukládal do konstantní paměti grafické karty (pro snazší celkovou modifikaci programu).

Loop fusion Sloučení cyklů jsem aplikoval u výpočtu gradientů, kdy se v kódu napočítávají oba směry gradientu současně, což díky nakešování dat obrazu do registrů snižuje počet čtení z paměti mimo čip na polovinu.

4 Naměřené výsledky a vyhodnocení

Měření jsem provedl na platformě s grafickou kartou **nVidia GeForce 940M**, která disponuje architekturou **Maxwell**. Velikost DDR3 paměti je 2GB@2000MHz a šířka připojené sběrnice 64-bitů. Jádrem je GM108 včetně 3 SMM. CPU měření byla provedena na platformě Intel Core i5-7500 CPU@3.40GHz, 16GB RAM.

Všechna uvedená měření jsou **průměrem ze 100 opakovaných samostatných měření** stejné instance.

Překlad zdrojového kódu jsem prováděl ve tvaru:
`$ nvcc main.cu --gpu-architecture=sm_50 --use_fast_math --optimize 3`

4.1 Výkon jednotlivých operací CPU a GPU verze

Následující tabulka shrnuje naměřené údaje z CPU (sekvenční) a GPU verze. V obou případech byly zanedbány potřebné paměťové alokace a přesuny před samotným výpočtem. Naměřené časové úseky reprezentují pouze danou operaci.

Data	CPU Gauss	GPU Gauss	CPU grads	GPU grads	CPU NMS	GPU NMS
lenna 256x256	9.6593ms	0.10675 ms	18.6814 ms	0.078 ms	2.4806 ms	0.073 ms
lenna 512x512	33.0502ms	0.28356 ms	74.4757 ms	0.19832 ms	19.05 ms	0.34502 ms
lenna 1024x1024	132.828ms	1.01958 ms	348.668 ms	0.76962 ms	74.6243 ms	1.30825 ms

Tabulka 1: Porovnání GPU a CPU časů zpracování obrazu

Z tabulky 1 je patrné zrychlení GPU verze o několik řádů oproti sekvenční verzi na CPU.

Data	Gauss	Gradients	Non-maxima supp.
lenna 256x256	90.485	239.505	33.981
lenna 512x512	116.555	375.533	55.214
lenna 1024x1024	130.277	453.039	57.041

Tabulka 2: Zrychlení operací GPU verze oproti operacím CPU verze

Tabulka 2 zachycuje zrychlení oproti CPU verzi. Z dostupných údajů je vidět, že GPU řešení oproti CPU řešení efektivně škáluje.

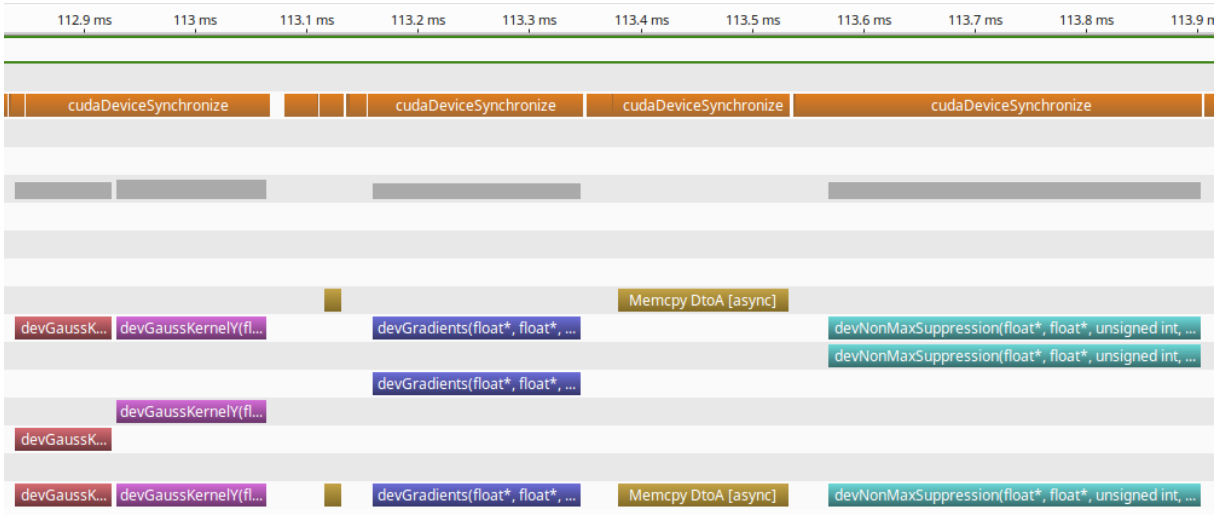
4.2 Výkon jednotlivých operací GPU v závislosti na velikosti bloku

Následující tabulka shrnuje naměřené údaje z celkého běhu GPU verze v závislosti na velikosti bloku (počtu vláken v bloku). Opět byly zanedbány potřebné paměťové alokace a přesuny před samotným výpočtem. Naměřené časové úseky reprezentují pouze čas strávený zpracováváním dat v krocích Cannyho algoritmu.

Data \ Blok	8x8	16x16	24x24	32x32
lenna 256x256	0.60817ms	0.59504ms	0.59207ms	0.57205ms
lenna 512x512	2.17823ms	2.18418ms	2.15115ms	2.13322ms
lenna 1024x1024	6.90006ms	6.85357ms	6.69436ms	6.56066ms

Tabulka 3: Porovnání GPU a CPU časů zpracování obrazu

Z tabulky je lze vyčíst nepatrné zrychlení GPU verze při použití větších bloků vláken.



Obrázek 16: Timeline z Visual Profiler, zachycující běh programu

4.3 Výstup z profileru

4.3.1 Timeline

Timeline na obrázku 20 zachycuje běh programu. Není zachyceno kopírování zpracovaného obrazu zpět z *device* na hostující počítač. Všechny kernely jsou dále ve zprávě postupně detailně zobrazeny.

4.3.2 Kernel `devGaussKernelX`

devGaussKernelX(float*, unsigned int, unsigned int)

Start	112.838 ms (112,838,147 ns)
End	112.925 ms (112,925,253 ns)
Duration	87.106 µs
Stream	Default
Grid Size	[16,16,1]
Block Size	[32,32,1]
Registers/Thread	15
Shared Memory/Block	0 B
▼ Efficiency	
Global Load Efficiency	n/a
Global Store Efficiency	100%
Shared Efficiency	n/a
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficiency	98.3%
▼ Occupancy	
Achieved	82.6%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	64 KiB
Shared Memory Executed	64 KiB
Shared Memory Bank Size	4 B

Obrázek 17: Vlastnosti kernelu `devGaussKernelX` z Visual Profiler

Kernel `devGaussKernelX` aplikuje 1D konvoluci na počáteční podobu zpracovávaného obrazu. Ta je nahrána v texturovací paměti. Jak bylo popsáno v předchozí kapitole, kernel používá registrové proměnné pro uchování některých proměnných, jako například `gRow` a `gCol`.

4.3.3 Kernel `devGaussKernelY`

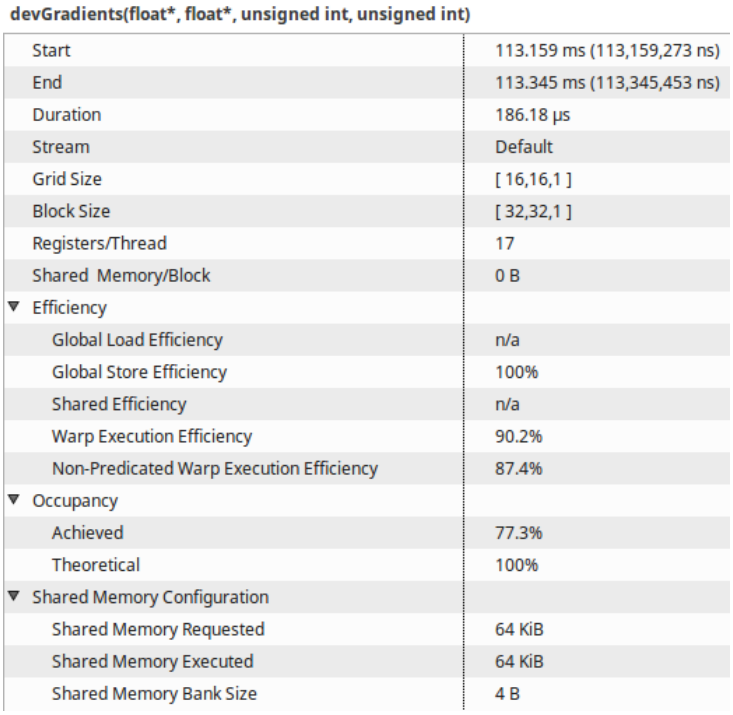
devGaussKernelY(float*, unsigned char*, unsigned int, unsigned int)	
Start	112.929 ms (112,929,285 ns)
End	113.064 ms (113,063,911 ns)
Duration	134.626 µs
Stream	Default
Grid Size	[16,16,1]
Block Size	[32,32,1]
Registers/Thread	14
Shared Memory/Block	4.5 KiB
▼ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	100%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficiency	95.9%
▼ Occupancy	
Achieved	93%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	64 KiB
Shared Memory Executed	64 KiB
Shared Memory Bank Size	4 B

Obrázek 18: Vlastnosti kernelu `devGaussKernelY` z Visual Profiler

Kernel `devGaussKernelY` aplikuje druhou 1D konvoluci na výstup obrazu z kernelu `devGaussKernelX`. Ta je nahrána v globální paměti. Používá se proto sdílená paměť jako cache globální paměti a kernel rovněž používá registrové proměnné pro uchování některých proměnných, jako například části zpracovávaného obrazu, `gRow` a `gCol`. Kernel dosahuje dobrých hodnot *occupancy* a *efficiency*.

Při ladění kernelu jsem se snažil vyhnou konfliktům bank experimentováním s adresací keší, nakonec se ale ukázalo, že současný způsob "2D" adresace ve výsledku není špatný.

4.3.4 Kernel `devGradients`



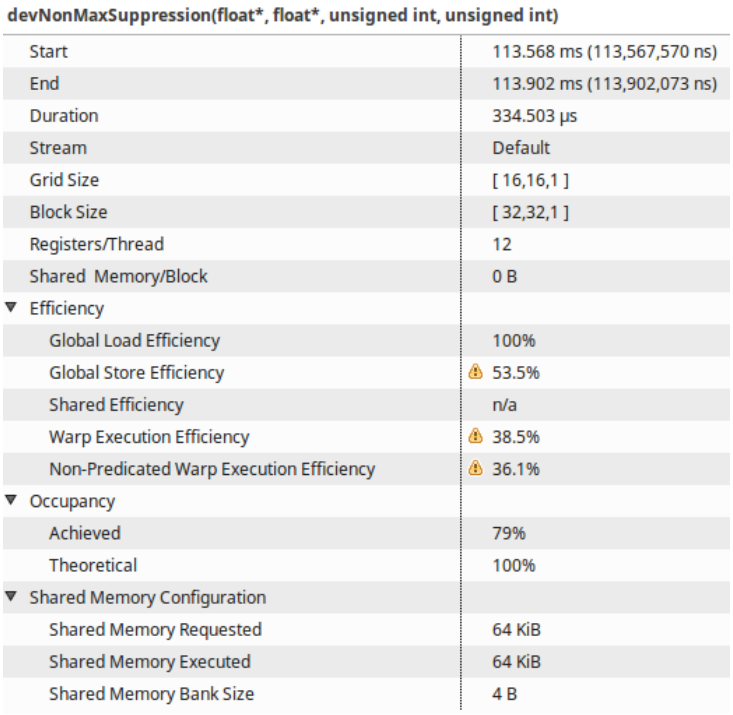
devGradients(float*, float*, unsigned int, unsigned int)	
Start	113.159 ms (113,159,273 ns)
End	113.345 ms (113,345,453 ns)
Duration	186.18 µs
Stream	Default
Grid Size	[16,16,1]
Block Size	[32,32,1]
Registers/Thread	17
Shared Memory/Block	0 B
▼ Efficiency	
Global Load Efficiency	n/a
Global Store Efficiency	100%
Shared Efficiency	n/a
Warp Execution Efficiency	90.2%
Non-Predicated Warp Execution Efficiency	87.4%
▼ Occupancy	
Achieved	77.3%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	64 KiB
Shared Memory Executed	64 KiB
Shared Memory Bank Size	4 B

Obrázek 19: Vlastnosti kernelu `devGradients` z Visual Profiler

Kernel `devGradients` aplikuje oba směrové Sobelovy operátory a zároveň vypočítává intenzitu a směr gradientu. Data, která zpracovává (rozostřený původní obraz) jsou nahrána v texturovací paměti, kvůli 2D keším. Většina divergence warpů je způsobena výpočtem směru gradientu, kde je nutné použít řadu matematických operací jednu po druhé. Výpočet intenzity a směru gradientu je největší *brzdou* kernelu. Kernel rovněž používá registrové proměnné pro uchování některých proměnných, jako například části zpracovávaného obrazu, `gRow` a `gCol`.

4.3.5 Kernel `devNonMaxSuppression`

Kernel `devNonMaxSuppression` zpracovává data z výstupu kernelu pro výpočet gradientu. Data, která zpracovává jsou nahrána v texturovací paměti, znovu kvůli 2D keším. Kernel dle profileru není optimálně odladěný. Bohužel z podstaty operace potlačení lokálních non-maxim plyne velká divergence warpů. Ta je způsobena především velkým větvením kódu. V mojí práci se mi nepodařilo provedením několika podmínek nijak vyhnout. I přes neefektivní kernel z pohledu profileru je výpočet o dva řády rychlejší, než jeho CPU varianta a dobře škáluje. Kernel opět používá registrové proměnné pro uchování některých proměnných, jako například intenzity gradientu zkoumaného prvku, protože je opakovaně použit při vyhodnocování podmínek v kódu.



devNonMaxSuppression(float*, float*, unsigned int, unsigned int)

Start	113.568 ms (113,567,570 ns)
End	113.902 ms (113,902,073 ns)
Duration	334.503 µs
Stream	Default
Grid Size	[16,16,1]
Block Size	[32,32,1]
Registers/Thread	12
Shared Memory/Block	0 B
▼ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	⚠ 53.5%
Shared Efficiency	n/a
Warp Execution Efficiency	⚠ 38.5%
Non-Predicated Warp Execution Efficiency	⚠ 36.1%
▼ Occupancy	
Achieved	79%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	64 KiB
Shared Memory Executed	64 KiB
Shared Memory Bank Size	4 B

Obrázek 20: Vlastnosti kernelu `devNonMaxSuppression` z Visual Profiler

4.4 Jiná dostupná řešení

Cannyho detekce hran je velmi rozšířený algoritmus a jeho implementaci i s řadou dalších úprav lze nalézt v celé řadě produktů. Mezi nejznámější patrně patří Matlab a knihovna OpenCV. Dokumentace OpenCV je velmi kvalitní podklad pro seznámení se s algoritmem a jeho následnou implementací "na zelené louce". Sručně a přehledně shrnuje jednotlivé kroky a jejích účel.

Vzhledem k naměřeným časům jednotlivých kroků algoritmu, lze algoritmu považovat jako konkurenceschopný (i když samozřejmě ne tak dobrý) vzhledem k uvedeným alternativám. Schopnost zpracovat milion obrazových bodů během 6ms znamená teoretickou schopnost zpracování 160 snímků za vteřinu. Oproti CPU sekvenční verzi (necelá vteřina) se jedná o velké zlepšení.

Použitelnost CUDA řešení je samozřejmě podmíněna užitím karty s podporou této technologie (pro vývoj a testování jsem musel použít notebook namísto své pracovní stanice, protože ta je vybavena kartou AMD).

Některé implementace při výpočtu síly gradientu "podvádějí", protože namísto vztahu

$$G = \sqrt{G_x^2 + G_y^2}$$

aplikují pouze vztah

$$G = |G_x| + |G_y|$$

který je samozřejmě výpočetně jednodušší, ale má výrazně nižší přesnost – dost nízkou na to, aby se projevila při detekci hran a musela být částečně kompenzována volbou konstant t_{min} a t_{max} pro hysterezi.

Data	Octave 4.0.3	CUDA
lenna 256x256	12.184ms	12.505ms
lenna 512x512	39.78ms	39.40ms
lenna 1024x1024	152.95ms	115.81ms

Tabulka 4: Porovnání délky zpracování dat s Octave 4.0.3

Z tabulky 4 je vidět, že CUDA řešení má velmi srovnatelné časy s jistě dobře odladěnou verzí Cannyho detekce hran ze software Octave ve verzi 4.0.3. Navrch se řešení CUDA dostává v případě většího objemu dat (1024x1024 BMP má velikost přes milion bodů a 3MB), což není překvapivé, zejména z tabulky zachycující zrychlení2.

5 Zhodnocení a závěr

V semestrální práci jsem implementoval Cannyho detekci hran na platformě CUDA. Z počátečního sekvenčního CPU řešení se mi podařilo vytvořit GPU řešení, které při zpracovávání obrazu je o několik řádů rychlejší a s velikostí obrázku škáluje. Porovnání se **sekvenční** verzí na CPU není zcela férové, neboť i CPU verzi lze do velké míry paralelizovat. Úloha hezky demonstruje, že ne všechny úlohy lze na GPU provádět efektivně.

Z povahy algoritmu (jedná se o algoritmus zpracovávající obraz) plyne, že některé části jsou pro paralelizaci na GPU velice vhodné (konvoluce), protože zpracování obrazu je hlavní doménou GPU jednotek. Naopak úlohy, které vyžadují během zpracování úrovně větvení nejsou pro zpracování na GPU vhodné (ale to samé do jisté míry platí i pro CPU zpracování). Na GPU platformě je však tento efekt umocněn divergencí celého warpu a tím pádem zpomalováním výpočtu.

Z pohledu algoritmu Cannyho detekce hran jsou vhodnými kandidáty na GPU zpracování konvoluce – kroky 1 a 2 (včetně výpočtu gradientu v jednoduché přesnosti). Krok 3 (potlačení lokálních non-maxim) není pro GPU ideální, ale i přesto lze z jeho implementace vytěžit výkonnostní zisk. Poslední krok, hystereze, není pro GPU implementaci vhodná vůbec a nepodařilo se mi ji naimplementovat tak, aby byla lepší než CPU verze. Tento můj závěr potvrzuje celá řada autorů odborných článků (vybrané z nich jsem v textu práce odkazoval). Autoři často hysterezi neimplementují vůbec, nebo nekorektně [7, Canny Edge Detection on NVIDIA CUDA]. V odkazované práci autoři během zpracování obrazu provedou **fixně čtyři** iterace hystereze a prohlásí výsledek za konečný. Takový přístup nepovažuji za správný, protože se podle mého názoru jedná o princip "sice špatně, ale rychle".

Práce na semestrální práci mne bavila, mrzí mne jen, že jsem si souběžně nezapsal předmět `MI-PAP` (vzhledem k částečnému překryvu s CUDA). Různorodost kroků algoritmu činila práci zajímavou, protože stejný postup pro zrychlení nefungoval na všech místech algoritmu a bylo třeba věnovat větší pozornost jednotlivým krokům. Také jsem se konečně s něčím *hands-on* seznámil v oblasti počítačového vidění.

Reference

[1] Cannyho hranový detektor. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-05-09]. Dostupné z: `https://cs.wikipedia.org/wiki/Cannyho_hranový_detektor`

[2] Rec. 709: ITU-R Recommendation BT.709. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-05-09]. Dostupné z: `https://en.wikipedia.org/wiki/Rec._709`

[3] IANDOLA, F. COMMUNICATION-MINIMIZING 2D CONVOLUTION IN GPU REGISTERS. 2013. [cit. 2017-05-09]. Dostupné z: `http://parlab.eecs.berkeley.edu/publication/899`

[4] Gaussian Kernel Calculator. A graphics programmer's blog [online]. 2014 [cit. 2017-05-10]. Dostupné z: `http://dev.theomader.com/gaussian-kernel-calculator/`

[5] The different addressing modes of CUDA textures. Stack Overflow [online]. 2013 [cit. 2017-05-10]. Dostupné z: `http://stackoverflow.com/questions/19020963/the-different-addressing-modes-of-cuda-textures`

[6] OpenCV: Canny Edge Detection. OpenCV [online]. [cit. 2017-05-11]. Dostupné z: `http://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html`

[7] DURAISWAMI, Ramani a Luo YUANCHENG. Canny Edge Detection on NVIDIA CUDA. 2008. [cit. 2017-05-11]. Dostupné z: `http://www.umiacs.umd.edu/~ramani/pubs/luo_gpu_canny_fin_2008.pdf`