Masarykova univerzita
Fakulta informatiky

Extrakce obrazových lokálních deskriptorů pomocí GPU

Bakalářská práce

Tomáš Surovec

Brno, 2011

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Tomáš Surovec

Vedoucí práce: RNDr. Tomáš Homola

Poděkování

Děkuji RNDr. Tomáši Homolovi za odborné vedení bakalářské práce a cenné rady při jejím zpracovávání.

Shrnutí

Cílem práce je implementovat algoritmus extrakce SIFT deskriptorů pro běh na grafické kartě. Teoretická část práce nejprve seznamuje s praktickým použitím SIFT deskriptorů, stručně a názorně popisuje algoritmus extrakce a zdůvodňuje vhodnost implementace algoritmu na GPU, poté je popsána problematika implementace algoritmů na GPU se zaměřením na architekturu CUDA. V poslední části se pak srovnává efektivita implementace klasické sériové verze algoritmu a výstupu praktické části.

Klíčová slova

Počítačové vidění, rozpoznávání obrazu, lokální deskriptor, bod zájmu, feature, SIFT, CUDA, GPGPU, extrakce deskriptorů, VLFeat

Obsah

1 **Úvod** . . . 2
1.1 *Obsah práce* . . . 2
1.2 *Použitá terminologie* . . . 3
2 **Algoritmus extrakce SIFT deskriptorů** . . . 5
2.1 *Použití v praxi* . . . 5
2.2 *Fáze algoritmu* . . . 5
2.2.1 Detekce lokálních extrémů ve scale-space . . . 6
2.2.1.1 Generování pyramidy . . . 6
2.2.1.2 Výpočet rozdílových obrázků . . . 6
2.2.1.3 Detekce lokálních extrémů . . . 6
2.2.2 Lokalizace klíčových bodů . . . 7
2.2.3 Přiřazení orientací klíčovým bodům . . . 7
2.2.4 Extrakce lokálních deskriptorů . . . 8
3 **Implementace algoritmů na GPU** . . . 12
3.1 *Motivace provádění obecných výpočtů na GPU* . . . 12
3.2 *CUDA* . . . 14
3.2.1 Specifikace architektury a programovacího modelu . . . 14
3.2.2 Architektura CUDA-enabled GPU . . . 14
3.2.3 Compute Capability . . . 14
3.2.4 Průběh výpočtu na hardwarové úrovni . . . 14
3.2.5 CUDA C . . . 14
4 **Srovnání korektnosti a efektivnosti CPU a GPU implementace** . . . 16
4.1 *Technická dokumentace* . . . 16
4.1.1 Parametry . . . 16
4.1.2 Návratová hodnota funkce . . . 17
4.1.3 Výstup . . . 17
4.2 *Poznámky k implementaci* . . . 18
4.3 *Poznámky k testování* . . . 18
4.3.1 Testovací konfigurace . . . 18
4.3.2 Průběh testování . . . 18
4.4 *Gaussián* . . . 19
4.5 *Rozdíl obrázků* . . . 19
4.6 *Downsampler* . . . 20
4.7 *Upsampler* . . . 20
4.8 *Výpočet gradientů* . . . 21
4.9 *Detektor lokálních extrémů* . . . 21
4.10 *Lokalizace klíčového bodu* . . . 22
4.11 *Výpočet orientací klíčových bodů* . . . 22
4.12 *Extrakce deskriptoru* . . . 23
4.13 *Celkový poměr časů* . . . 23
4.14 *Relativní poměr časů strávených jednotlivými fázemi* . . . 23
5 **Závěr** . . . 24
A **Obsah CD** . . . 26
B **Scénář použití demo programu** . . . 27

1 Úvod

Lidské bytosti vnímají svět jako množství trojrozměrných entit s mnoha objektivně zjistitelnými vlastnostmi. Z fyzikálního hlediska však příchozí data nevykazují žádnou vzájemnou souvislost - fotony odražené od dvou různých objektů se (kvalitativně) od sebe nijak neliší. Náš vizuální systém tedy musí určitým způsobem odfiltrovat nepodstatná a zavádějící data a z těch zbývajících vyextrahovat podstatné informace a souvislosti za použití určitých základních předpokladů a znalostí (např. čím je objekt dále od oka, tím se jeví menším a naopak). Tyto informace pak mozek dále zpracovává - identifikuje samostatné objekty, odhadne některé jejich vlastnosti (např. barvu, relativní polohu v prostoru, atd.) a na jejich základě činí rozhodnutí. [2]

Počítačové vidění (angl. *computer vision*) je vědecký a technologický obor zkoumající způsoby, jakými se z obrazových dat dají vyextrahovat informace potřebné pro řešení určitého problému. Obrazová data se mohou vyskytovat v mnoha různých formách, společné však mají zejména to, že se všechny dají považovat za n-rozměrná pole dat (od obyčejných dvou rozměrných obrázků či fotografií přes sekvenci obrazů v podobě videa až po prostorové záznamy z lékařských přístrojů). Počítačové vidění lze v jistém smyslu chápat i jako proces inverzní k počítačové grafice. Zatímco počítačová grafika vytváří z primitiv jako jsou body, úsešky, plochy, aj. výsledný obraz, počítačové vidění přebírá na vstupu obraz a jako výstup vrací ony původní primitiva, tzn. snaží se rekonstruovat scénu (nebo její část), ze které by obraz mohl být vytvořen.

Počítačové viděné jako odvětví informatiky je relativně mladé. Vzhledem k vysoké výpočetní intenzitě algoritmů zpracovávajích obraz bylo dlouhou dobu značně nepraktické rozsáhlejší použití počítačového vidění v praxi. Tento obor je celkem široký a proto není jednoduché přesně určit ani co je předmětem bádání, ani co se dá považovat za úspěšné řešení problematiky. Vpodstatě se dá říci, že jde o souhrn mnoha různých uzce specializovaných metod a jejich výběr a použití velice záleží na konkrétní aplikaci. Spousta z nich je doposud ve stádiu základního výzkumu, některé si však již našly cestu do produktů běžně dostupných na trhu. [8]

Pro jakýkoliv objekt na obrázku lze vybrat množinu bodů zájmu v jeho okolí, které určitým způsobem objekt charakterizují. Tento popis, získaný ze vzorového obrázku, pak může být použit při detekci tohoto objektu v jiných obrázcích. Ke spolehlivému rozpoznání je důležité, aby bylo objekt možné detekovat i při změnách velikosti, osvětlení, polohy či pootočení obrázku. [13]

V první kapitole blíže popisuji co je SIFT deskriptor, k čemu jej lze konkrétně použít a algoritmus extrakce z obrazových dat. V druhé kapitole charakterizuji implementaci algoritmů na GPU a architekturu CUDA. Třetí kapitola je pak srovnáním výsledné GPU implementace - rychlosti a korektnosti - s předlohou. Poslední kapitola je shrnutím a diskuzí o další možném vývoji práce.

1.1 Obsah práce

I přestože je výpočetní výkon dnešních procesorů vysoký, výpočet extrakce SIFT deskriptorů stále trvá řádově jednotky vteřin. Při hromadném zpracování velkého množství obrázků tak je rychlost stále omezujícím faktorem pro rozsáhlé praktické použití. Na druhé straně rozvoj počítačové grafiky a specializovaných zařízení zaměřených na její výpočet (grafických

karet) spolu s faktem, že se staly běžnou součástí osobních počítačů, vedl v posledních letech ke vzniku relativně levného hardwaru specializovaného na provádění výpočtů nad velkým množstvím prvků.

Cílem práce bylo implementovat algoritmus extrakce SIFT deskriptorů a využít výpočetního výkonu grafické karty pro kritické části algoritmu za účelem urychlení extrakce.

1.2 Použitá terminologie

- *Obrázek* Dvourozměrné pole pixelů. Rozlišením obrázku se myslí délky stran pole. Udává se ve tvaru $w \times h$, kde w (resp. h) je počet pixelů podél horizontální (resp. vertikální) osy. V rámci extrakce SIFT deskriptorů se pracuje pouze s černobílými obrázky. Každý pixel je vyjádřen jedním číslem v plovoucí čárce představující světlost v tomto bodě. Z teoretického pohledu obrázek představuje diskrétní reprezentaci spojitého signálu jako funkce dvou proměnných x, y (tzv. souřadnic). Dle konvence má levý horní roh obrázku souřadnice $x = 0, y = 0$.

- *Scale-space* Teorie umožňující reprezentaci signálu ve více měřítcích — Základní idea reprezentace signálu ve více měřítcích (angl. *multi-scale representation*) je zasadit původní signál (v kontextu této práce jde o vstupní obrázek) do parametrizované rodiny derivovaných signálů (tento parametr se nazývá *scale* neboli měřítko). [4]. S roustoucím *scale* jsou v odvozeném signálu více potlačeny jeho vysokofrekvenční složky, což umožňuje pracovat i s nízkofrekvenčními složkami signálu.

- *Feature* Obecný termín pro kus informace relevantní pro řešení daného problému. Může jít např. o specifické struktury přímo obsažené v obrázku (od jednoduchých jako jsou body či hrany až po komplexnější struktury jako jsou objekty) či výsledky operací.[9]

- *Klíčový bod*, angl. *keypoint*. V kontextu SIFT je to lokální extrém scale-space a definuje přesné umístění *feature* ve scale-space obrázku.

- *Lokální deskriptor* (též pouze *deskriptor*) Struktura vhodným způsobem popisující okolí klíčového bodu.

- *Oktáva* Posloupnost několika obrázků vzniklých postupnou aplikací Gaussovského rozostření.

- *Pyramida* Diskrétní reprezentace spojitého scale-space. Sestává z posloupnosti oktáv.

- *Gaussián* Obrázek, který je výsledkem konvoluce vstupního obrázku s funkcí Gaussovského rozostření.

- *Difference of Gaussians* (zkr. *DoG*) Rozdíl dvou Gaussiánů.

- *Downsampling* snížení rozlišení obrázku. Pokud není řečeno jinak, bere se jako faktor zmenšení 2 podél každé z os.

- *Upsampling* zvýšení rozlišení obrázku. Pokud není řečeno jinak, bere se jako faktor zvětšení 2 podél každé z os.

- *Graphics Processing Unit* (zkr. *GPU*) Komponenta zaměřená na výpočty související s počítačovou grafikou.
- *Compute Unified Device Architecture* (zkr. *CUDA*) Architektura pro obecné paralelní výpočty.
- *CUDA enabled* Přívlastek označující GPU anebo grafickou kartu, na které lze spouštět programy pro architekturu CUDA.
- *Kernel* Funkce, která je paralelně spouštěna na CUDA enabled grafické kartě [1].

2 Algoritmus extrakce SIFT deskriptorů

Scale-invariant feature transform (*SIFT*) je algoritmus počítačového vidění sloužící k detekci a popisu *features* v obrázku. V kontextu SIFT je třeba rozlišovat tzv. klíčové body, což jsou lokální extrémy scale-space, a features, které jsou tvořeny okolím klíčových bodů. Každá feature je popsána tzv. (lokálním) deskriptorem, který má v tomto případě podobu histogramu gradientů v okolí klíčového bodu. Takové deskriptory jsou invariantní vzhledem ke změnám polohy, měřítka a rotace obrázku a částečně invariantní vůči změnám osvětlení a polohy pozorovatele. Jsou navíc vysoce charakteristické, což zajišťuje vysokou přesnost správné identifikace features v rozsáhlých databázích. [5] Jako detektor features označujeme algoritmus, který v obrázku nalezne klíčové body, extraktor features je potom algoritmus, který pro daný klíčový bod z obrázku vyextrahuje deskriptor.[9]

2.1 Použití v praxi

Obecně lze SIFT deskriptory použít kdykoliv je třeba identifikovat nějaký objekt ve více obrázcích. Konkrétními aplikacemi jsou např.

- rozpoznávání objektů, kdy máme databázi trénovacích obrázků spolu s množinou jejich deskriptorů a po předložení obrázku složené scény se v něm snažíme nalézt objekty z databáze.
- panorama-stitching („slepování fotografií“) cílem této techniky je rekonstruovat obraz původní scény z více částečných obrázků, přičemž tyto dílčí obrázky mohou být různě velké, pootočené či posunuté. Hlavním problémem tak je zjistit ve kterých místech obrázky spojit a jak je předtím transformovat. Viz. obrázek 2.1
- robotické mapování a lokalizace, kdy se u klíčových bodů odhaduje i vzdálenost od pozorovatele (robota). Při pohybu robot průběžně porovnává obraz s částečnou mapou prostředí, čímž zjišťuje svoji polohu (lokalizace), a zároveň ji obohacuje o nové features, čímž fakticky mapuje neznámé prostředí. [13]
- rozpoznávání gest a obličejů [13]
- trasování objektu ve sledu chronologicky seřazených obrázků (např. video) [13]

2.2 Fáze algoritmu

Algoritmus extrakce SIFT deskriptorů lze rozdělit do čtyř základních fází: [5]

1. Detekce lokálních extrémů ve scale-space
2. Lokalizace klíčových bodů
3. Přiřazení orientací klíčovým bodům
4. Extrakce lokálních deskriptorů

Obrázek 2.1: Výsledek techniky *panorama stitching*

Zdroj: fredriksalomonsson.wordpress.com

2.2.1 Detekce lokálních extrémů ve scale-space

2.2.1.1 Generování pyramidy

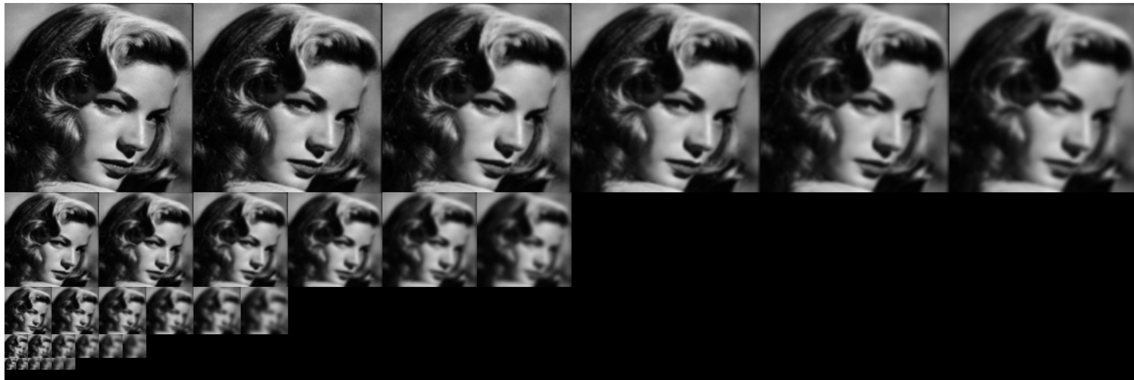
Nejprve je vytvořena tzv. *pyramida*, což je diskrétní aproximace spojitého scale-space. Pyramida sestává z několika *oktáv*, jejichž počet a velikost lze měnit. Platí, že čím více oktáv pyramida má a čím větší jsou samotné oktávy, tím je aproximace přesnější (a tím více lokálních extrémů scale-space se nám podaří nalézt), zároveň ale roste výpočetní náročnost této fáze. Samotné generování pyramidy spočívá v opakované konvoluci Gaussovského jádra s obrázkem, čímž se získá posloupnost postupně se rozostřujících obrázků. Gaussovské rozostření funguje jako nízkofrekvenční filtr, tzn. potlačuje vysokofrekvenční informace v obrázku, což nám umožňí nalézt i nízkofrekvenční extrémy a navíc konvolucí nevznikají extrémy, které v původním signálu nebyly. Obrázek oktávy, který má dvojnásobnou hodnotu rozostření σ oproti prvnímu obrázku oktávy je pak downsamplován (kvůli snížení výpočetní náročnosti) a použit jako vstupní obrázek pro generování následující oktávy. Viz. obrázek 2.2

2.2.1.2 Výpočet rozdílových obrázků

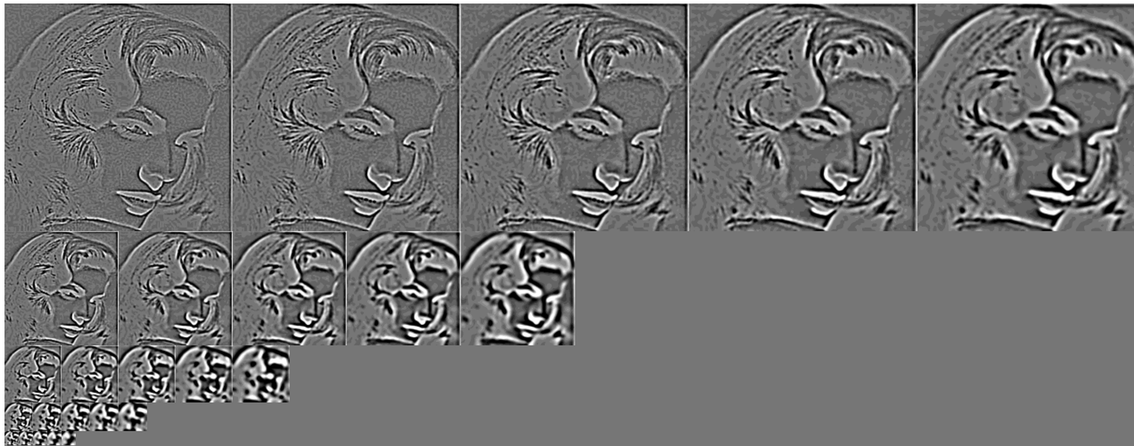
Po vygenerování pyramidy je proveden rozdíl každých dvou sousedních gaussiánů v každé oktávě (angl. *Difference of Gaussians*, zkr. *DoG*). Viz. obrázek 2.3

2.2.1.3 Detekce lokálních extrémů

Následně jsou v rozdílových obrázcích nalezeny lokální extrémy. To se provádí tak, že pro každý pixel rozdílového obrázku je zjištěno, zda je jeho hodnota větší (resp. menší) než hodnoty pixelů v jeho bezprostředním okolí stejného obrázku a též ve stejném okolí v sousedních



Obrázek 2.2: Scale-space pyramida s pěti oktávami



Obrázek 2.3: Pyramida rozdílových obrázků

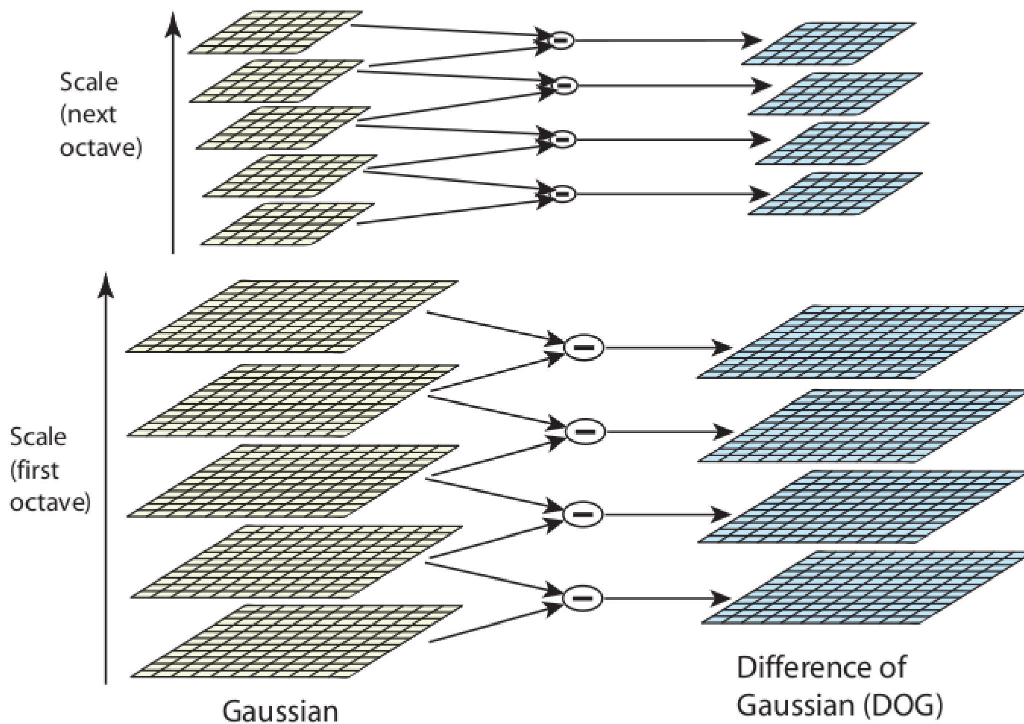
obrázcích (viz. obrázek 2.5). Pokud ano, je nalezen lokální extrém.

2.2.2 Lokalizace klíčových bodů

Každý lokální extrém je dále vyšetřen, aby byly vyloučeny body, které leží na hranách a body s nízkým kontrastem a aby se pomocí kvadratické interpolace zpřesnila poloha kandidátního klíčového bodu v obrázku a ve scale-space, což zvyšuje stabilitu výsledného deskriptoru. [5]

2.2.3 Přiřazení orientací klíčovým bodům

Tato fáze je nutná kvůli zajištění invariance vůči rotaci. Pro každý klíčový bod je vytvořen histogram orientací gradientů v okolí klíčového bodu. Maxima histogramu odpovídají převažujícím orientacím lokálních gradientů v okolí klíčového bodu. Klíčovému bodu jsou přiřazeny až 4 různé orientace (maximum histogramu a další lokální maxima histogramu s alespoň 80% hodnoty maxima - viz. obrázek 2.8). [5]

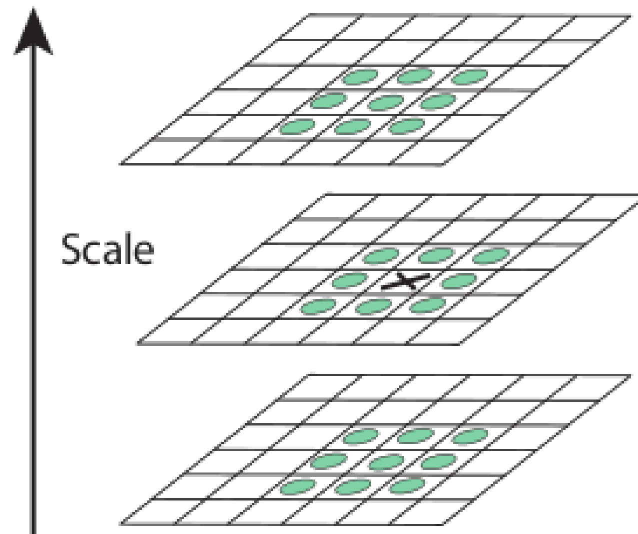


Obrázek 2.4: Výpočet rozdílových obrázků

Zdroj:[5]

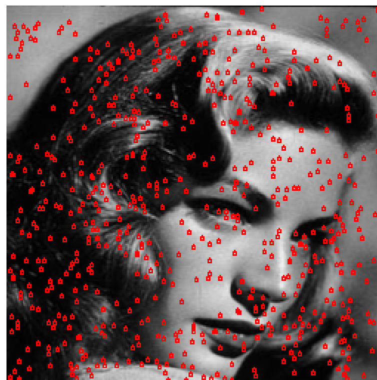
2.2.4 Extrakce lokálních deskriptorů

Předchozí kroky přiřadily každému klíčovému bodu polohu v obrázku, pozici ve scale-space a orientaci. To zajišťuje znovu vytvořitelný 2D systém souřadnic (což poskytuje invarianci vzhledem k těmto parametrům), ve kterém se popíše deskriptor. Dalším krokem je spočítat lokální deskriptor, který je vysoce charakteristický a současně tak invariantní, jak je jen možné vůči změnám osvětlení či poloha pozorovatele. Pro každou orientaci každého klíčového bodu je okolí klíčového bodu rozděleno do $n \times n$ oblastí. Pro každou oblast je pak spočten histogram gradientů o p směrech. SIFT deskriptor je tak tvořen 3D histogramem o $n \times n \times p$ složkách. [5]

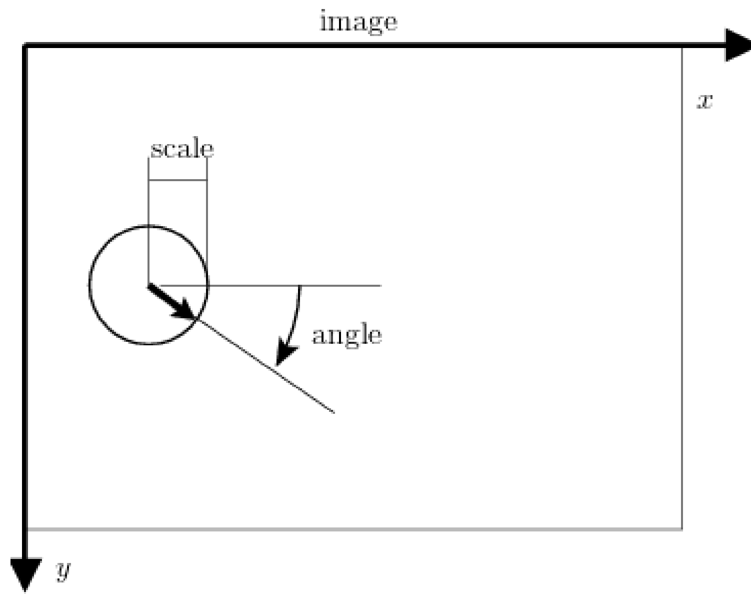


Obrázek 2.5: Lokální extrém v rozdílových obrázcích

Zdroj: [5]

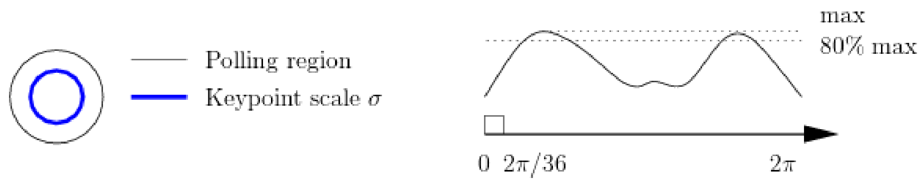


Obrázek 2.6: Lokalizované klíčové body



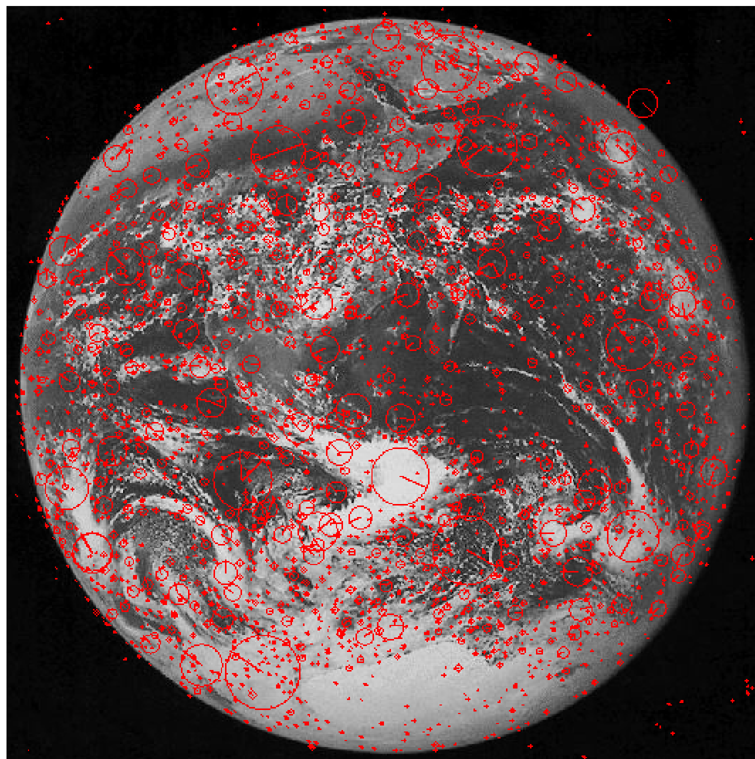
Obrázek 2.7: Schéma klíčového bodu

Zdroj: [7]



Obrázek 2.8: Histogram orientací gradientů

Zdroj: [7]



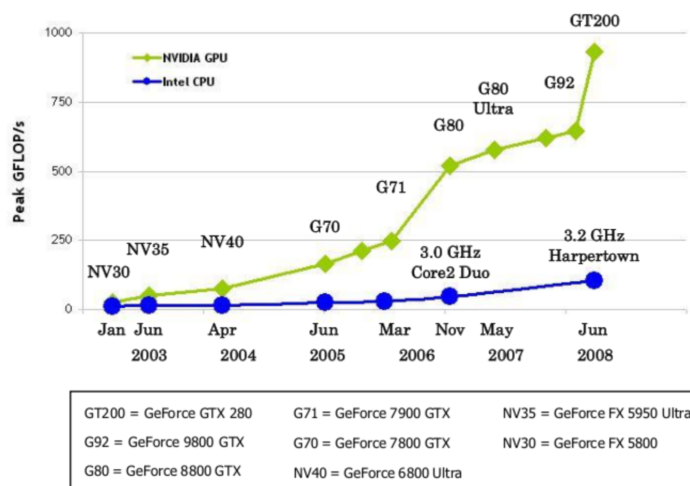
Obrázek 2.9: Klíčové body s přiřazenými orientacemi

3 Implementace algoritmů na GPU

S postupným masovým rozšiřováním osobních počítačů rostla nutnost vytvářet stále složitější a sofistikovanější uživatelská prostředí, navíc jejich výpočetní výkon neustále rostl, a tak nacházely využití ve stále širším okruhu aplikací. Počítače se také stále běžněji používaly i k zábavě, která byla primárně zaměřena na vytváření efektních grafických výjevů. V roce 1985 vstoupil na trh Commodore Amiga, první osobní počítač využívající GPU urychlující vykreslování (v tomto případě hlavně 2D operace související s rasterizací). V průběhu 90. let se dalé rozvíjela akcelerace 2D operací a čím dál zřetelnější byla i nutnost akcelerovat 3D grafiku, což na konci 90. let vyústilo v nástup první generace 3D akcelerátorů na trh.[12] Ty byly zprvu schopny provádět jen základní operace (transformace, práce s osvětlením, rasterizace, apod.) a navíc měly uzavřený (neprogramovatelný) model výpočtu. V roce 2001 se pak objevila GeForce 3, první grafická karta poskytující vývojářům možnost zasahovat do některých fází vykreslování obrazu, což (kromě poskytnutí mnohem větší flexibility programům pracujícím s grafikou) byl první krok v proměně GPU z vysoce specializovaného kusu hardwaru ve flexibilní výpočetní jednotku se širokým polem působnosti. [10]

3.1 Motivace provádění obecných výpočtů na GPU

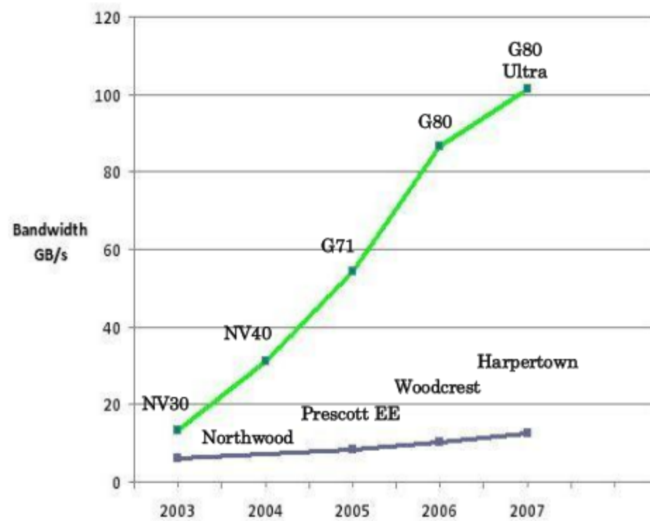
Nenasytná poptávka po stále rychlejších a složitějších grafický výpočtech (motivovaná požadavkem na stále realističtější obraz ve vysokém rozlišení) měla za následek vývoj GPU ve vysoce paralelní vícevláknový mnohojádrový procesor s obrovským výpočetním výkonem a velmi vysokou paměťovou propustností, která řádově překonává dnešní CPU stejných cenových relací.[1]



Obrázek 3.1: Výpočetní výkon CPU vs. GPU

Zdroj: [1]

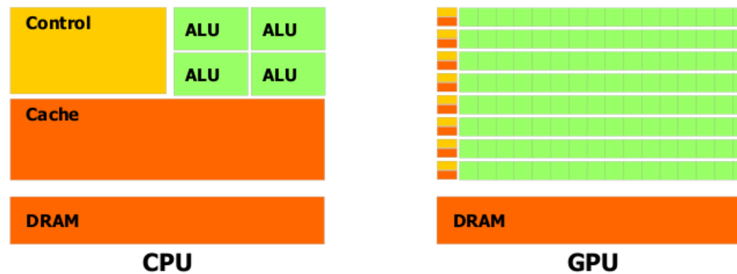
Důvodem tak propastného rozdílu mezi výpočetním výkonem CPU a GPU je to, že CPU ma relativně mnohem více tranzistorů vyčleněných pro funkce, které přímo nesouvisí s prováděním výpočetních operací (řízení toku, caching, predikce příští instrukce, aj). [1] GPU jsou totiž díky svému původnímu zaměření na zpracovávání velkého množství dat v reálném



Obrázek 3.2: Paměťová propustnost CPU vs. GPU

Zdroj: [1]

čase (pixely obrazu) architektonicky navrženy jako SIMD (Single Instruction Multiple Data).



Obrázek 3.3: Symbolické znázornění plochy, kterou zabírají tranzistory pro výpočet

Zdroj: [1]

Postupem času se grafické karty s výkonnými procesory staly naprosto běžnou součástí osobních počítačů. Vyvstala otázka, zda lze (a pokud ano, tak jak) využít GPU i pro výpočty nesouvisející s grafikou. Tak vznikla technika zvaná *General-purpose computing on graphics processing units* (zkr. *GPGPU*) zabývající se použitím GPU pro obecné výpočty. [11] V prvopočátcích GPGPU (kolem roku 2003) nebyly dostupné žádné specializované nástroje pro využívání GPU k obecným výpočtům, a tak se muselo využívat tehdy dostupných způsobů, tzn. využití API a knihoven, které primárně používaly GPU k akceleraci grafických výpočtů a vykreslování. Takový způsob výpočtu je velice těžkopádný a pro některé problémy takřka nepoužitelný. Data musely mít např. tvar obrázku či textury v paměti grafické karty, kód pracující s daty byl psán např. v GLSL či Cg, jazycích uzpůsobených na provádění per-pixel operací, a výpočet proběhl vykreslením scény. Poptávka po prostředích a architekturách umožňující snadný vývoj programů využívajících GPU postupně vedl až ke vzniku dnes

dostupných architektur jako jsou AMD Stream, CUDA či OpenCL. [3]

3.2 CUDA

Computation Unified Device Architecture (CUDA) byla poprvé představena roku 2006 společností NVIDIA. Je to architektura pro obecné paralelní výpočty využívající CUDA enabled grafické karty (duben 2011 - takové GPU jsou vyráběny jen společností NVIDIA). [1]

3.2.1 Specifikace architektury a programovacího modelu

Základní jednotkou výpočtu je *kernel*, což je funkce v jazyce C, která je paralelně spuštěna N-krát N různými CUDA vlákny (*CUDA threads*). Vlákna jsou sdružována do trojrozměrných *bloků*, přičemž v rámci bloku je každé vlákno identifikováno jednoznačnými souřadnicemi `x, y, z`. Bloky jsou pak organizovány do dvojrozměrné *mřížky* (angl. *grid*) a opět platí, že každý blok je v rámci mřížky jednoznačně identifikován souřadnicemi `x, y`. Rozměry bloků a mřížky určuje uživatel explicitně při volání kernelu v tzv. *konfiguraci kernelu* [1]

3.2.2 Architektura CUDA-enabled GPU

CUDA-enabled GPU obecně sestává z N multiprocesorů (angl. *streaming multiprocessor, SM*), z nichž každý obsahuje M jader (angl. *scalar processor, SP*). Celkově je tedy na kartě přítomno $N \times M$ jader (např. GeForce GTX 295 má 30 multiprocesorů, každý s 8 jádry, což dává dohromady celkem 240 jader).

3.2.3 Compute Capability

Každé CUDA-enabled GPU má explicitně definováno tzv. *Compute Capability* (zkr. *CC*), což je číslo tvaru $A.B$ blíže specifikující vlastnosti GPU (např. instrukční sadu, množství sdílené paměti na multiprocesoru, maximální dimenze bloku a mřížky, aj).

3.2.4 Průběh výpočtu na hardwarové úrovni

Vlákna jednoho bloku jsou prováděna paralelně na jednom multiprocesoru, což implikuje omezenou velikost bloku (pro CC $\leq$ 1.3 je maximální velikost bloku 512 vláken, pro CC $\geq$ 2.0 je to 1024 vláken). Naproti tomu ale tato skutečnost umožňuje vláknům bloku vzájemnou synchronizaci a komunikaci přes sdílenou paměť multiprocesoru (*shared memory*). Na každém multiprocesoru může současně běžet výpočet až 8 bloků. Vlákna bloku jsou multiprocesorem sdružována v tzv. *warpy* o velikosti 32 vláken. Vlákna warpu se spouštějí současně a vykonávají stejnou instrukci. Pokud výpočet v rámci warpu diverguje (různá vlákna chtějí vykonávat různé instrukce), pak se výpočet serializuje: vlákna se rozdělí do skupin podle následující instrukce, které budou provádět, a jednotlivé skupiny jsou pak zpracovávány jedna po druhé. [6]

3.2.5 CUDA C

CUDA umožňuje vyvíjet programy, které spouštějí kód na grafické kartě, v CUDA C, což je jazyk C obohacený o několik konstrukcí, jmenovitě jde hlavně o:

- explicitní definice, pro která zařízení se má funkce kompilovat. Kernely jsou definovány jako `__global__ void identifikátor(parametry)` a jsou volatelné jen z hostitelského kódu. Funkce, které jsou volány GPU kódem, jsou definovány klíčovým slovem `__device__`
- specifikaci konfigurace volání kernelu, která se provádí:
  `jménokernelu<<<dim3(Bx, By, Bz), dim3(Gx, Gy)>>>(...)` Kde `Bx, By, Bz` jsou dimenze bloků a `Gx, Gy` jsou dimenze mřížky.
- funkce pro správu paměti grafické karty (alokace, dealokace, přesuny mezi hlavní systémovou pamětí a pamětí GPU)
- aj.

Typické schéma kódu pro výpočet na GPU:

- Příprava vstupních dat výpočtu
- Alokace paměti na GPU a přesun vstupních dat na GPU
- Výpočet konfigurace volání kernelu
- Asynchronní volání kernelu
- Vykopírování výsledků z paměti GPU do systémové paměti
- Zpracování výsledků

4 Srovnání korektnosti a efektivnosti CPU a GPU implementace

4.1 Technická dokumentace

Výstupem praktické části je sdílená knihovna pro systém Linux, která zpřístupňuje jedinou základní funkci:

```
int extract_sifts(const float *img,
                  unsigned int imgw,
                  unsigned int imgh,
                  int octaveSize,
                  int nOctaves,
                  int firstOctave,
                  int sMin,
                  int sMax,
                  float sigmaN,
                  float sigmaK,
                  float sigma0,
                  float dSigma0,
                  float peakThreshold,
                  float edgeThreshold,
                  float magnification,
                  float windowSize,
                  char **outbuffer,
                  int *outbuffersize,
                  bool diagnosticMode = false);
```

4.1.1 Parametry

- `const float *img` ukazatel na pole čísel v plovoucí čárce o velikosti $imgw \times imgh$ představujících pixely vstupního obrázku
- `unsigned int imgw` šířka vstupního obrázku v pixelech
- `unsigned int imgh` výška vstupního obrázku v pixelech
- `int octaveSize` velikost oktávy - počet obrázků, které bude každá oktáva obsahovat
- `int nOctaves` počet oktáv pyramidy
- `int firstOctave` index první oktávy; pokud je index záporný, bude vstupní obrázek upsamplován, pokud je kladný, bude obrázek downsamplován
- `sMin` minimální scale obrázků oktávy
- `sMax` maximální scale obrázků oktávy
- `float sigmaN` hodnota σ rozostření vstupního obrázku

- `float sigmaK, float sigma0, float dSigma0` jiné parametry ovlivňující generování pyramidy
- `float peakThreshold` parametr ovlivňující potlačení nízkokontrastních lokálních extrémů
- `float edgeThreshold` parametr ovlivňující potlačení lokálních extrémů na hranách
- `float magnification, float windowSize` parametry ovlivňující velikost okolí, ze kterého se vypočítává deskriptor
- `char **outbuffer` ukazatel na ukazatel, který bude odkazovat na výstupní data
- `int *outbuffersize` velikost výstupních data
- `bool diagnosticMode` pokud je tento parametr nastaven na `true`, výpočet bude proveden zároveň na CPU i GPU, výsledky budou porovnány a program na standardní chybový výstup vytiskne ladící informace; při nastavení `false` výpočet probíhá jen na GPU a na standardní výstup se netiskne nic

4.1.2 Návratová hodnota funkce

- < 0 při chybě
- $>= 0$ počet nalezených deskriptorů

4.1.3 Výstup

Výstupní data extrakce se nacházejí v paměti na adrese, na kterou ukazuje ukazatel `char *outbuffer`. Paměť je alokována C++ příkazem `new char[...]`, její dealokaci provádí aplikační programátor příkazem `delete[]`. Výstup je tvaru textového řetězce o $2N$ řádcích, kde N je počet nalezených deskriptorů. Data pro každý deskriptor jsou popsána dvěma řádky tohoto tvaru:

```
X,Y,Ori,Scale
v0,v1,v2, ...,v126,v127
```

Význam jednotlivých polí:

- **X, Y** normalizované souřadnice klíčového bodu. Dle úzu je $[0, 0]$ levý horní a $[1, 1]$ pravý dolní roh obrázku.
- **Ori** orientace klíčového bodu z intervalu $[0, 2\pi]$.
- **Scale** scale klíčového bodu.
- **v*n*** n-tá složka samotného vektoru popisující deskriptor. Všechny složky jsou celá čísla z intervalu $[0, 255]$.

4.2 Poznámky k implementaci

Jako referenční implementaci jsem bral knihovnu VLFeat 0.9.9 (dále jen VLFeat), která je napsána v jazyce C. S tou sem srovnával rychlost i korektnost výsledků. Vstupem je pole čísel v plovoucí čárce představující hodnoty jasu pixelů obrázku. Data obrázku jsou po celou dobu výpočtu uchovávána v globální paměti grafické karty, aby se zamezilo zbytečným přenosům mezi grafickou kartou a systémovou pamětí. Data jsou na/z grafické karty přenášena až když je to bezprostředně nutné.

4.3 Poznámky k testování

4.3.1 Testovací konfigurace

- **Hardware** 4 x CPU Intel(R) Core(TM)2 Quad CPU Q8400 @ 2.66GHz, 8 GB RAM, 6 x GeForce GTX 295 (30 multiprocesorů, 240 jader, 1 GB GDDR3, Compute Capability 1.3)
- **Software** OS Linux version 2.6.26-2-amd64 (Debian 2.6.26-26lenny1), NVIDIA CUDA Toolkit 3.1 NVIDIA UNIX x86-64 Kernel Module 256.35 GCC version 4.1.3

Nastavení kompilátoru pro překlad a linkování sdílené knihovny .so:

```
nvcc -arch=sm_13 -G --compiler-options '-fPIC' -shared
```

4.3.2 Průběh testování

Rychlost: Absolutní čas výpočtu v mikrosekundách změřen jako rozdíl mezi systémovým časem před výpočtem a po něm. U GPU výpočtu je navíc bezprostředně před každým odečítáním času provedena synchronizace GPU a CPU, aby bylo zajištěno ukončení všech asynchronních GPU operací v momentě odečítání času. Korektnost výpočtu: U algoritmů na per-pixel bázi (gaussián, rozdíl obrázků, downsampler, upsampler, výpočet gradientů) se porovnává absolutní rozdíl hodnot výsledného pixelu GPU a CPU výpočtu. Jako stejný výsledek se pak bere, když je pro každý pixel tato hodnota menší než $\epsilon = 1e - 2$. U ostatních algoritmů (detektor lokálních extrémů, lokalizace klíčového bodu, výpočet orientací klíčového bodu, extrakce deskriptoru) individuálně, viz. příslušná sekce.

4.4 Gaussián

Při výpočtu Gaussiánu je, stejně jako v knihovně VLFeat, využito separability konvoluce a výpočet probíhá dvoufázově - konvoluce s jádrem ve směru horizontální osy a pak konvoluce jádra s mezivýsledkem ve vertikálním směru, díky čemž je časová složitos lineární.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
13×7	0.04	0.19	0.21×
16×16	0.09	0.20	0.46×
123×177	6.93	0.49	14.24×
191×191	11.57	0.77	15.09×
256×256	20.81	1.31	15.89×
533×533	90.35	5.29	17.06×
625×450	89.47	5.16	17.34×
1024×768	266.03	14.17	18.77×
1200×800	304.42	17.22	17.68×
1600×1200	609.16	34.32	17.75×
3264×1832	1921.70	108.82	17.66×

4.5 Rozdíl obrázků

Operace počítající rozdíl dvou obrázků stejných rozměrů v každém pixelu.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
13×7	0.001	0.024	0.03×
16×16	0.001	0.024	0.06×
123×177	0.111	0.033	3.40×
191×191	0.179	0.041	4.40×
256×256	0.326	0.056	5.85×
533×533	1.488	0.160	9.31×
625×450	1.394	0.159	8.77×
1024×768	3.900	0.380	10.27×
1200×800	5.061	0.454	11.15×
1600×1200	10.085	0.875	11.53×
3264×1832	38.878	2.662	14.60×

4.6 Downsampler

Operace, která zmenšuje obrázek z původní velikosti $[w, h]$ na velikost $[\frac{w}{2}, \frac{h}{2}]$.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
13 × 7	0.001	0.032	0.02×
16 × 16	0.001	0.031	0.03×
123 × 177	0.054	0.033	1.63×
191 × 191	0.092	0.044	2.08×
256 × 256	0.144	0.056	2.55×
533 × 533	0.925	0.147	6.29×
625 × 450	0.818	0.147	5.58×
1024 × 768	5.892	0.336	17.53×
1200 × 800	2.991	0.408	7.34×
1600 × 1200	6.803	0.777	8.75×
3264 × 1832	25.451	2.324	10.95×

4.7 Upsampler

Upsampling s lineární interpolací, který zvětšuje obrázek z velikosti $[w, h]$ na velikost $[2w, 2h]$. Používá se jen pokud je index první oktávy záporný.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
13 × 7	0.004	0.037	0.10×
16 × 16	0.010	0.036	0.28×
123 × 177	0.829	0.078	10.69×
191 × 191	1.444	0.108	13.42×
256 × 256	3.537	0.201	17.60×
533 × 533	15.136	0.571	26.49×
625 × 450	15.112	0.572	26.44×
1024 × 768	169.743	1.476	114.99×
1200 × 800	58.962	1.791	32.92×
1600 × 1200	229.286	3.540	64.76×
3264 × 1832	1317.712	10.938	120.47×

4.8 Výpočet gradientů

Operace, která pro každý pixel obrázku vypočte velikost a orientaci gradientu v tomto bodě.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
13×7	0.012	0.060	0.19×
16×16	0.024	0.065	0.37×
123×177	1.842	0.117	15.80×
191×191	3.087	0.173	17.82×
256×256	5.520	0.288	19.15×
533×533	24.141	1.032	23.39×
625×450	23.658	1.089	21.73×
1024×768	67.855	2.779	24.42×
1200×800	69.993	3.287	21.29×
1600×1200	154.044	6.480	23.77×
3264×1832	537.864	19.649	27.37×

4.9 Detektor lokálních extrémů

Operace, která na vstupu přebírá tři obrázky a na výstupu vrací seznam pixelů, které jsou lokálními extrémy. Korektnost je kontrolována ujištěním, že oba výpočty označí stejnou množinu extrémů.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
13×7	0.01	0.12	0.06×
16×16	0.02	0.19	0.12×
123×177	1.89	1.45	1.30×
191×191	3.20	2.12	1.51×
256×256	5.93	3.56	1.66×
533×533	26.04	14.99	1.74×
625×450	25.16	14.89	1.69×
1024×768	73.55	41.19	1.79×
1200×800	85.44	50.62	1.69×
1600×1200	174.03	101.03	1.72×
3264×1832	181.13	104.14	1.74×

4.10 Lokalizace klíčového bodu

Výpočetně intenzivní operace, který kvadraticky interpoluje polohu lokálních extrémů a odstraňuje lokální extrémy ležící poblíž hran. Korektnost zajištěna ujištěním, že CPU i GPU výpočty odstranily stejné klíčové body.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
16 × 16	0.003	0.437	0.01×
123 × 177	0.042	1.038	0.04×
191 × 191	0.065	1.526	0.04×
256 × 256	0.120	1.396	0.09×
533 × 533	0.549	1.578	0.35×
625 × 450	0.582	1.579	0.37×
1024 × 768	1.487	1.669	0.89×
1200 × 800	2.524	1.858	1.36×
1600 × 1200	3.715	2.232	1.66×
3264 × 1832	3.522	1.902	1.85×

4.11 Výpočet orientací klíčových bodů

Operace, která pro každý klíčový bod vypočítá histogram gradientů v jeho okolí a na jeho základě pak vrací (až 4) orientaci(-e) klíčového bodu. Korektnost kontrolována ujištěním, že pro každý klíčový bod CPU i GPU výpočet našel stejný počet orientací, přičemž absolutní rozdíl orientací nesmí být větší než $\epsilon = 1e - 2$.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
16 × 16	0.047	0.089	0.53×
123 × 177	0.942	0.461	2.04×
191 × 191	1.444	0.642	2.25×
256 × 256	3.961	1.779	2.23×
533 × 533	18.191	7.781	2.34×
625 × 450	17.703	7.413	2.39×
1024 × 768	67.226	31.314	2.15×
1200 × 800	67.465	30.205	2.23×
1600 × 1200	124.066	61.020	2.03×
3264 × 1832	130.073	77.358	1.68×

4.12 Extrakce deskriptoru

Operace, která pro daný klíčový bod vyextrahuje deskriptor. Korektnost kontrolována porovnáním příslušných složek výsledného deskriptoru CPU a GPU výpočtu.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
123 × 177	47	16	2.849×
191 × 191	87	30	2.816×
256 × 256	126	48	2.633×
533 × 533	653	253	2.582×
625 × 450	626	221	2.833×
1024 × 768	1136	421	2.696×
1200 × 800	1873	674	2.777×
1600 × 1200	3502	1412	2.479×
3264 × 1832	16800	6060	2.772×

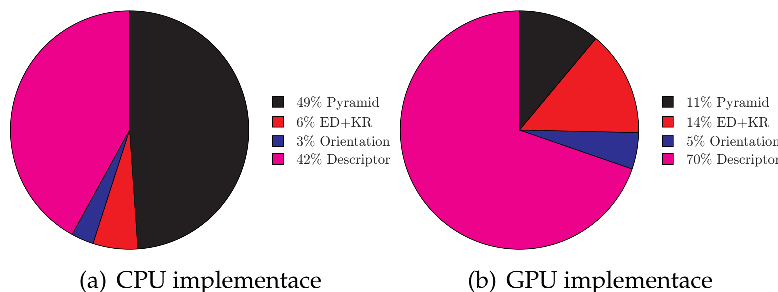
4.13 Celkový poměr časů

Poměr celkového času výpočtu na CPU a GPU.

Rozlišení obrázku	Čas na CPU [ms]	Čas na GPU [ms]	Poměr časů CPU/GPU
123 × 177	107	45	2.377×
191 × 191	187	65	2.880×
256 × 256	305	90	3.379×
533 × 533	1435	381	3.764×
625 × 450	1400	347	4.028×
1024 × 768	3341	695	4.804×
1200 × 800	4477	1023	4.373×
1600 × 1200	8704	2071	4.203×
3264 × 1832	32465	8263	3.929×

4.14 Relativní poměr časů strávených jednotlivými fázemi

Pro obrázek s rozlišením 2592×1944:



(a) CPU implementace (b) GPU implementace

Obrázek 4.1: Srovnání relativních časů výpočtu jednotlivých fází

5 Závěr

Cílem práce bylo implementovat extrakci SIFT deskriptorů z obrázku s využitím GPU. Výstupem je plně funkční dynamická knihovna pro systém Linux spolu se všemi zdrojovými kódy. Další vývoj práce by se měl ubírat hlavně směrem zefektivnění a optimalizace kódu vykonávaného grafickou kartou, především části, která provádí samotnou extrakci deskriptoru, protože ta stále představuje časově nejnáročnější fázi. Je možno využít možností poskytovaných novějšími CUDA-enabled kartami s vyšší Compute Capability, rozšířit výpočet o možnost použití OpenCL, vyzkoumat dopad použítí texturové paměti.

Literatura

[1] NVIDIA Corporation. Nvidia cuda programming guide version 3.0. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf, 2010.

[2] C. M. Brown D. H. Ballard. *Computer vision*. Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[3] GPGPU.org. Gpgpu. http://gpgpu.org/developer, May 2011.

[4] Tony Lindeberg. Scale-space theory: A basic tool for analysing structures at different scales. *Journal of Applied Statistics*, pages 225–270, 1994.

[5] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, pages 91–110, 2004.

[6] John Nickolls, Ian Buck, Michael Garland, NVIDIA, Kevin Skadron, and University of Virginia. Scalable parallel programming with cuda. ACM Queue March/April 2008, 2008.

[7] VLFeat.org. Sift c api documentation. http://www.vlfeat.org/api/sift_8h.html, May 2011.

[8] Wikipedia.org. Computer vision. http://en.wikipedia.org/wiki/Computer_vision, April 2011.

[9] Wikipedia.org. Feature (computer vision). http://en.wikipedia.org/wiki/Feature_%28Computer_vision%29, April 2011.

[10] Wikipedia.org. Geforce 3. http://en.wikipedia.org/wiki/GeForce_3, May 2011.

[11] Wikipedia.org. Gpgpu. http://en.wikipedia.org/wiki/GPGPU, May 2011.

[12] Wikipedia.org. Graphics processing unit. http://en.wikipedia.org/wiki/Graphics_processing_unit, May 2011.

[13] Wikipedia.org. Scale-invariant feature transform. http://en.wikipedia.org/wiki/Scale-invariant_feature_transform, April 2011.

A Obsah CD

Zdrojové kódy ve složce *srcs*

Soubor	Poznámky
cputests.cpp	
cputests.h	
cudamacros.h	
cudasift.h	Deklarace rozhraní sdílené knihovny
descriptor.cu	
descriptor.h	
deviceimage.cu	
deviceimage.h	
difference.cu	
difference.h	
gaussian.cu	
gaussian.h	
gpuapi.cu	
gpuapi.h	
gradient.cu	
gradient.h	
keypointdetector.cu	
keypointdetector.h	
log.cpp	
log.h	
octave.cpp	
octave.h	
orientations.cu	
orientations.h	
pyramid.cpp	
pyramid.h	
samplers.cu	
samplers.h	
solib.cpp	Interní implementace rozhraní sdílené knihovny
solib.h	Hlavičkový soubor pro interní implementaci rozhraní sdílené knihovny
timers.cpp	
timers.h	
types.h	

Skripty a ostatní ve složce *demo*

Soubor	Poznámky
makeso	Skript pro zkompilování sdílené knihovny
maketest	Skript pro zkompilování demo programu
testso.cpp	Zdrojový kód demo programu
pic1.pgm	Testovací obrázek pro demo program
pic2.pgm	Testovací obrázek pro demo program
pic3.pgm	Testovací obrázek pro demo program

B Scénář použití demo programu

Zdrojový kód demo programu se nachází na CD ve složce `demo`. Kroky nutné k jeho spuštění:

1. Spustit skript `makeso` ve složce `demo`. Ten vykopíruje zdrojové kódy knihovny do složky `cudasift` v domovském adresáři uživatele a zkompiluje zde knihovnu.
2. Provést příkaz `export LD_LIBRARY_PATH=~/cudasift:$LD_LIBRARY_PATH`
3. Spustit skript `maketest` ve složce `demo`, který zkompiluje demo program a jeho spustitelný soubor `testso.o` umístí do složky `cudasift` v domovském adresáři uživatele.
4. Spuštění demo programu s odpovídajícími parametry.

Demo program se spouští takto: `./testso.o imagefile diagmode`
Parametry demo programu:

- `imagefile` Vstupní obrázek formátu PGM. Povinný parametr.
- `diagmode` Nepovinný parametr (implicitně 0). Možné hodnoty:
  - 0 - výpočet proběhne jen GPU výpočet, deskriptory budou vytištěny na standardní výstup
  - 1 - proběhne GPU i CPU výpočet, ladící informace se vytisknou na standardní chybový výstup, deskriptory se vytisknou na standardní výstup, uloží vygenerovanou pyramidu do série PGM obrázků `pyramid_o*_s*.pgm`, uloží obrázek s vyznačenými klíčovými body bez orientace do souboru `refinedkey.bmp`, uloží obrázek s vyznačenými klíčovými body a orientacemi do souboru `orikey.bmp`