VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

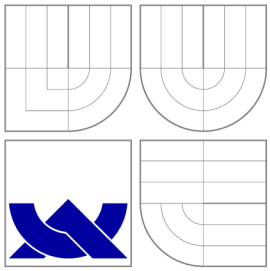
DETEKCE POHYBLIVÉHO OBJEKTU VE VIDEU NA CUDA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

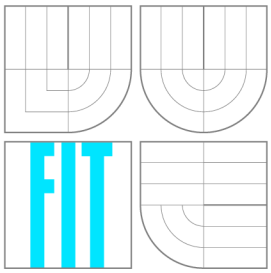
AUTOR PRÁCE
AUTHOR

FRANTIŠEK HORÁK

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DETEKCE POHYBLIVÉHO OBJEKTU VE VIDEU NA CUDA

MOVING OBJECT DETECTION IN VIDEO USING CUDA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — FRANTIŠEK HORÁK
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. ADAM HEROUT, Ph.D.
SUPERVISOR

BRNO 2011

Abstrakt

Práce se zabývá metodikami detekce pohybu na videozáznamech; výběrem a aplikací vhodných algoritmů za použití technologie CUDA. Tato technologie je dále rozebírána, zkoumána z hlediska její architektury a dalších součástí. Jsou sledovány obecné principy detekce pohybu a navrhovány systémy detekčních algoritmů. Systém je implementován a podroben testům.

Abstract

This thesis deals with methods of moving object detection in video; by selecting and aplying proper algorithms and implementing them with CUDA. Technology CUDA is described and wieved it's architecture and other parts. General principles of moving object detection are observed and modification are discussed. Final system is implemented and tested.

Klíčová slova

detekce pohyblivého objektu, odečítání pozadí, CUDA, modelování pozadí, detekce popředí.

Keywords

detection of moving object, background subtraction, CUDA, background modeling, foreground detection.

Citace

František Horák: Detekce pohyblivého objektu ve videu na CUDA, bakalářská práce, Brno, FIT VUT v Brně, 2011

Detekce pohyblivého objektu ve videu na CUDA

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Adama Herouta, Ph.D. . Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
František Horák
18. května 2011

Poděkování

Za odborné vedení, cenné rady a ochotu chci poděkovat vedoucímu bakalářské práce doc. Ing. Adamu Heroutovi, Ph.D. . Dále bych rád poděkoval své rodině za podporu.

Obsah

Kapitola 1

Úvod

Detekce pohyblivého objektu spadá do jedné z moderních vědních disciplín - *počítačové vidění*. Detekcí objektu získáme informaci, která nám v závislosti na svojí kvalitě slouží jako podklad pro další zpracování. Využití nalézáme v dohledových systémech firem a supermarketů, při monitorování dodržování pravidel silničního provozu, hledání odcizených vozidel, výběr mýtného na zpoplatněných úsecích silnic, sledování a rozpoznávání lidí, v různých vojenských aplikacích a v mnoha dalších oblastech.

Nový dech do tohoto odvětví přináší technologie CUDA. Jedná se o mladou a živou architekturu z dílny NVIDIE, která dovoluje provádět výpočty v několika vláknech současně. Právě výše zmíněný paralelismus může znamenat nárůst v rychlosti a potažmo i kvalitě zpracování obrazu.

Cílem této práce je nastudování, výběr a implementace vhodných algoritmů za současného využití technologie CUDA. Na detektor jsou kladeny především nároky na zpracování v reálném čase.

V kapitole 2 se věnujeme popisu technologie CUDA, rozebíráme její architekturu a všímáme si jejích součástí. V kapitole 3 uvádíme obecné principy detekce pohybu spolu s některými algoritmy. Navržení systému a úpravy detekčních algoritmů řešíme v kapitole 4. Kapitolu 5 jsme zasvětili implementaci a testování systému.

Kapitola 2

Popis technologie CUDA

V následujícím textu se budeme věnovat použité technologii a některým algoritmům. CUDA (Compute Unified Device Architecture) je vynález společnosti NVIDIA. Jedná se o technologii, která dovoluje vývojářům využít výpočetní schopnosti grafické karty zejména v oblasti paralelních výpočtů. Převedením výpočtu na grafické karty se jednak uspoří čas CPU, v mnohých případech se výpočty zrychlí v řádech desítek procent a některé složitější výpočty je možné provádět v reálném čase.

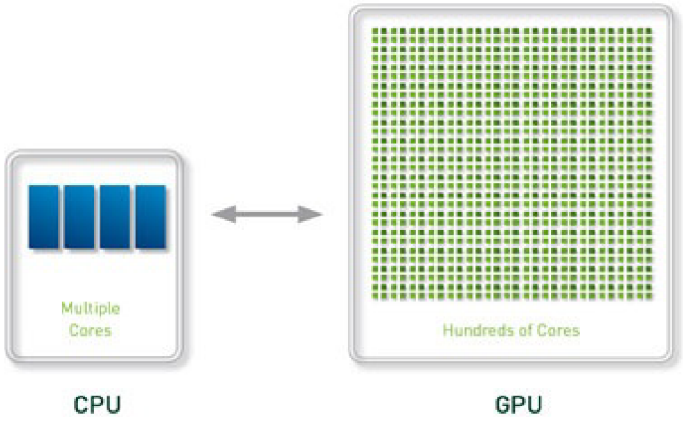
Jak uvádí NVIDIA na svých stránkách [12] CUDA nachází své uplatnění v mnohých oblastech vědeckého výzkumu. Například je pomocí CUDA urychlován AMBER, program pro dynamickou simulaci molekul, který slouží vědcům po celém světě pro výzkum nových druhů léčiv. CUDA své využití nalézá i v řadě finančních programů určených pro bankovní systémy. S nedávným vypuštěním operačních systémů Microsoft Windows 7 a Apple Snow Leopard přestává být GPU pouze grafickým procesorem, ale začíná být paralelním procesorem pro obecné účely dostupným všem aplikacím.

Výše zmíněná architektura je v dnešní době součástí grafických karet GeForce, ION, Quadro a Tesla. Existuje vícero způsobů v čem psát zdrojové kódy pro výpočty na GPU: CUDA C/C++, OpenCL, DirectCompute, CUDA Fortran a další.

2.1 Výpočty na GPU

Základní myšlenka výpočtů na GPU spočívá v současném užitím CPU a GPU v heterogenním výpočetním modelu (obrázek 2.1), kde sekvenční část aplikace běží na CPU a výpočetně náročná část je zrychlena GPU díky tomu, že GPU obsahuje stovky procesorových jader, na kterých může výpočet probíhat paralelně. CUDA programovací model je navržen tak, aby programátoři znalí standardních jazyků (C a podobných) měli co nejméně práce s jeho zvládnutím.

Jádrem celé věci jsou tři klíčové abstrakce - hiearchie vláken, sdílené paměti a synchronizace vláken. Kvůli těmto třem věcem je zvolený programovací jazyk obohacen o další sadu klíčových slov. Abstrakce v tomto směru poskytuje datový a vláknový paralelismus. Tyto nás vedou k tomu, abychom rozložili daný problém na podproblémy, které mohou být řešeny nezávisle v paralelních *blocích* vláken. Je tedy na nás - programátorech, abychom implementované algoritmy upravili pro paralelní zpracování. Architektura rovněž zachovává možnost vláken spolupracovat mezi sebou a zároveň dovoluje jejich zaměnitelnost. Jinými slovy, jakýkoliv blok vláken může být spuštěn na libovolném procesoru a v libovolném pořadí, ať už v sekvenčním nebo konkurentním režimu. Vytvořený program tedy může



Obrázek 2.1: Heterogenní model CPU a GPU, zdroj [12]

být spuštěn na obecném počtu jader, jak je znázorněno na obrázku 2.2. Jedinou věcí, kterou musí systém vědět, je počet fyzických jader k dispozici. Ten se totiž mezi jednotlivými grafickými kartami liší.

2.2 Architektura

Zde v jednoduchosti popíšeme jednotlivé části architektury tak, jak je uvádí průvodce pro programátora [9].

2.2.1 Kernely

CUDA C rozšiřuje klasický jazyk C o speciální funkce, tzv. *kernely*, které jsou při zavolání spuštěny N-krát v paralelním režimu na N různách CUDA vláknech. Nikoliv tedy pouze jednou, jak je to běžné u klasických C funkci.

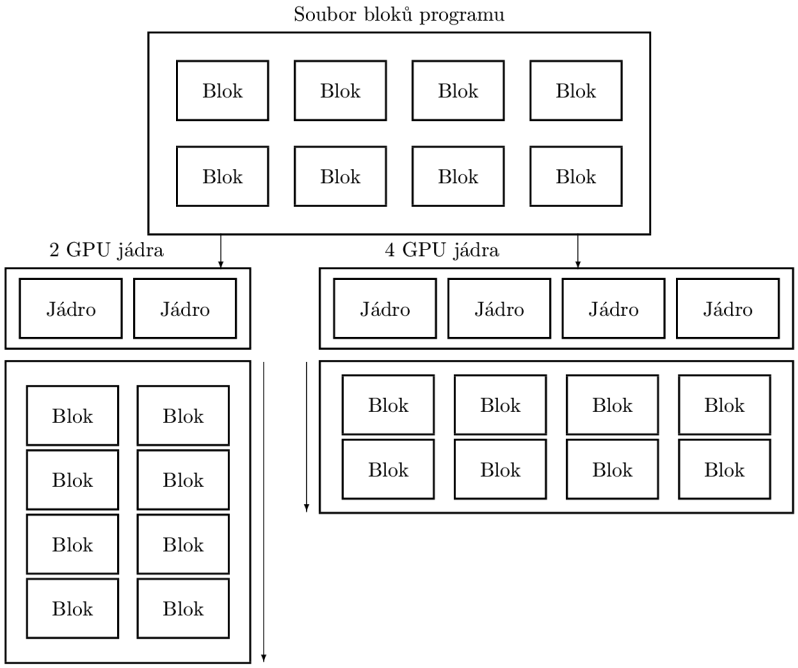
Kernel se definuje přidáním klíčového slova `__global__` před vlastní funkci a udáním počtu CUDA vláken, ve kterých bude spouštěn. Každému jádru cuda při spuštění přidělí unikátní *thread ID*, podle nějž je možné jednotlivá vlákna identifikovat. K tomuto ID lze ve vlastním kernelu přistupovat pomocí vestavěné proměnné `threadIdx`.

2.2.2 Hiearchie vláken

Pro pohodlnost je threadIdx 3 složkový vektor, takže vlákna můžeme identifikovat pomocí jednorozměrného, dvourozměrného, nebo třírozměrného indexu. Až trojrozměrně jsou pak tvořeny i bloky vláken. Tím je zaručen přirozený způsob práce s vektory, maticemi a objemem.

Index vlákna a jeho ID spolu přímo souvisí. Pro jednorozměrný blok jsou totožné. Pro dvourozměrný blok velikosti (D_x, D_y) určíme thread ID na pozici (x, y) jako $(x + y * D_x)$. A pro trojrozměrný blok velikosti (D_x, D_y, D_z) máme thread ID vlákna na pozici (x, y, z) dán vztahem $(x + y * D_x + z * D_x * D_y)$.

Existuje tu ovšem limit počtu vláken na jeden blok, protože očekáváme, že vlákna v rámci jednoho bloku poběží na stejném jádře a budou tedy muset sdílet omezené množství paměti. U současných GPU se udává 512 jako maximální počet vláken na blok. Avšak mějme



Obrázek 2.2: Způsob vypořádání s různým počtem fyzických jader. Na GPU s více jádry jsou bloky zpracovány v rychlejším čase.

na paměti, že vytvořený program může být složen z mnoha bloků, takže celkový počet jader je roven součinu počtu jader na blok a počtu bloků.

Bloky řadíme do jednorozměrné, dvourozměrné, nebo třírozměrné mřížky. Struktura 2D uspořádání je patrná z obrázku 2.3. Počet bloků v mřížce odvozujeme nejčastěji od dat určených ke zpracování, nebo od počtu procesorů v systému. Bloků může být i výrazně více než procesorů.

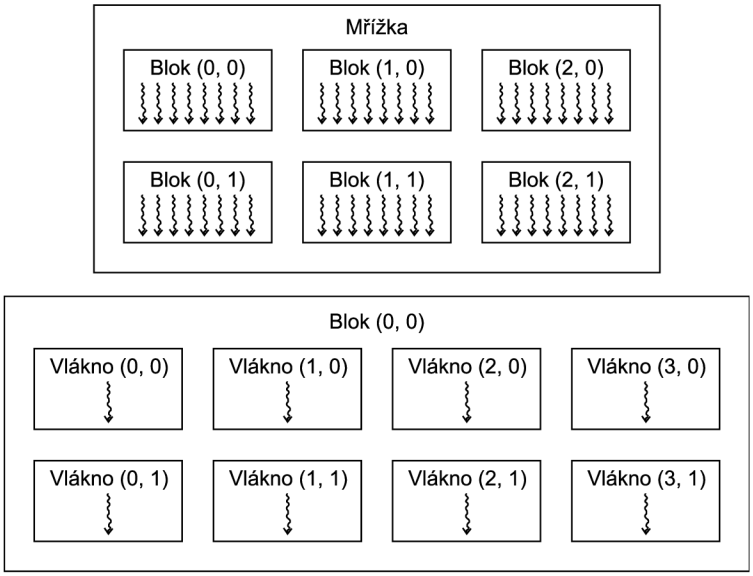
Oba údaje, počet vláken na blok a počet bloků v mřížce, musíme předat kernelu při jeho volání, přičemž podporované typy parametrů jsou buď prostý `int` nebo třísložkový vektor `dim3`.

Každý blok může být identifikován 1 až 3 rozměrným indexem, který v kernelu získáme z vestavěné proměnné `blockIdx`. Rozměr bloku nalezneme pod proměnnou `blockDim`.

Lepší spolupráce mezi vlákny jednoho bloku dosáhneme tím, že použijeme *sdílenou paměť* a sychnronizační nástroje pro přístup k ní. Abychom byli přesnější, můžeme přímo v kódu specifikovat synchronizační body, ve kterých se vykonávání programu zastaví a čeká se tak dlouhou, dokud všechna vlákna nedospějí k tomuto bodu.

2.2.3 Uspořádání paměti

Vlákna během svého běhu smí využívat data z vícero druhů paměti. Každé vlákno má k dispozici svou vlastní, lokální paměť. Každý blok má vlastní paměť, která je sdílená vrámci všech vláken daného bloku a vzniká a zaniká současně s blokem. Všechna vlákna mají přístup do společné globální paměti. Další dvě paměti jsou určeny pouze pro čtení a jsou taktéž přístupné všem vláknům. Setkáme se s nimi pod názvy konstantní paměť a



Obrázek 2.3: Složení mřížky a bloku v 2D prostoru

texturová paměť.

Rozdíly mezi globální, konstantní a texturovou pamětí jsou především v optimalizaci pro jiné účely. Texturová pamět se od zbylých liší také tím, že nabízí odlišný způsob adresace paměti a filtrování specifického formátu dat. Výše zmíněné 3 druhy paměti přetrvávají v dané aplikaci a můžeme k nim přistupovat i z různých kernelů.

Pokud bychom zhrnuli popsané druhy paměti, měli bychom jich celkem pět. Lokální paměť vláken, sdílená bloková paměť, globální paměť, konstatní pamět a texturová paměť.

Musíme ještě zmínit, že programovací model CUDA předpokládá běh kernelů na fyzicky odděleném zářízení (grafické kartě). S tím souvisí i předpoklad rozdělení paměťového prostoru na paměť přístupnou procesoru a paměť přístupnou zařízení. Což zahrnuje i alokování a uvolňování paměti na zařízení a přesouvání dat mezi procesorem a zařízením. Musíme mít na paměti i to, že kernely mohou přistupovat pouze do paměti zařízení. Proto, chceme-li k nějakému výpočtu použít kernel, musíme všechna data potřebná k výpočtu zkopírovat do paměti zařízení.

2.2.4 Sdružený přístup do paměti

Na závěr této kapitoly zmiňme sdružený přístup do paměti (coalesced memory acces). Informace byly čerpány z [13]. Globální paměť není kešovaná a nachází se daleko od čipu, což má za následek pomalou práci s ní. Abychom přístup do této paměti alespoň trochu urychlili, zavádí Nvidia sdružený přístup do paměti. Na konkrétním čipu je v jednu chvíli obsluhováno 32 vláken, ty nazýváme *warp*. Šestnáct vláken, tedy *half-warp*, může přistupovat do globální paměti najednou a při dodržení určitých podmínek se tím práce s pamětí zrychlí.

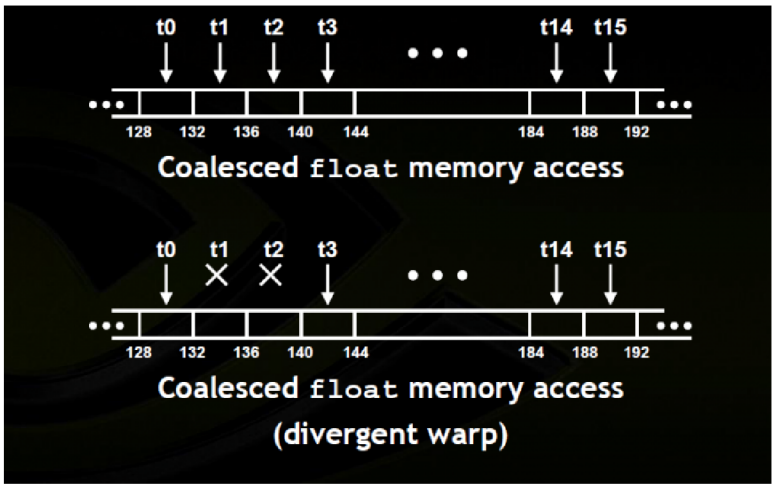
Oněmi podmínkami jsou:

1. Vlákna přistupují k paměti sekvenčně. K n-tému elementu pouze n-té vlákno.

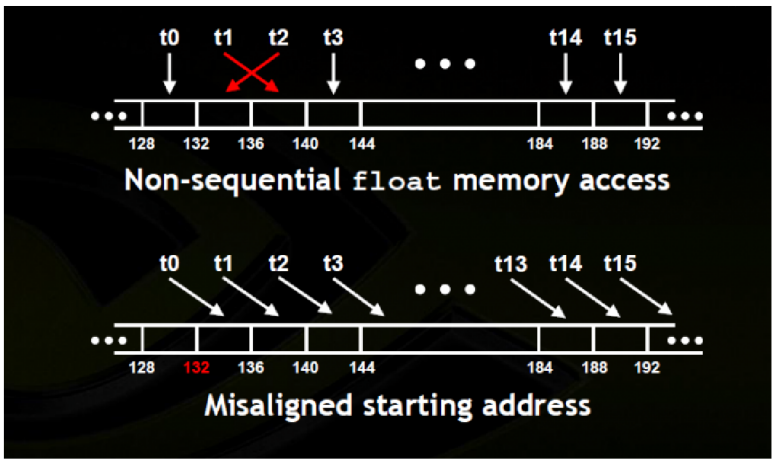
2. Přípustné velikosti přistupovaných elementů jsou 4B, 8B, 16B.

3. Přistupované elementy globální paměti jsou ve stejném segmentu, přičemž adresa prvního elementu musí být zarovnána k 16násobku velikosti elementu.

Pokud jsou výše uvedené podmínky splněny, jedná se o sdílený přístup do paměti (16 vláken souběžně přistupuje do globální paměti). V opačném případě je transakce zúžena pouze na jediné vlákno (v jeden okamžik přistupuje pouze jedno vlákno do sdílené paměti). Rozdíl uvedených transakcí je samozřejmě znatelný a proto se vyplatí dodržovat výše popsaná pravidla. Pro lepší názornost sdíleného přístupu nám poslouží obrázek 2.4 a nesdíleného přístupu obrázek 2.5.



Obrázek 2.4: Znázornění sdruženého přístupu do paměti. Zdroj [14]



Obrázek 2.5: Znázornění nesdruženého přístupu do paměti. Zdroj [14]

Kapitola 3

Detekce pohybu

Zmínili jsme technologii CUDA a nyní přejdeme k postupu detekce pohyblivých objektů. Již běžným přístupem k detekci je oddělení pozadí od popředí, kde jako pozadí bereme tu část obrazu, která zůstává neměnná, statická a bez pohybu a za popředí pak považujeme hýbající se objekty. Pozadí v reálných záběrech ale nemáme vždy statické, ba naopak, stává se z menších i větších změn (pohybující se listí na stromech, náhodný šum, změna intenzity osvětlění jedná-li se o venkovní záběry, rozsvícení/zhasnutí v interiérech, ...), které při vyhodnocování detektory chceme stále považovat za pozadí. Oproti tomu popředí nemáme vždy k dispozici celistvé a přesně ohraničené, ale často rozšířené o stíny, nebo ochuzené o nějakou část. Nacházíme celou řadu metod, které s vyšší, či nižší účinností řeší daný problém, avšak mají své nároky na výpočetní výkon. Zde zmíníme některé z často používaných metod.

3.1 Odečítání pozadí

Metoda odečítání pozadí je jednou z nejběžněji používaných metod detekce pohybu. Za její popularitu vděčíme především díky její rychlosti a jednoduché implementaci. Snad právě proto se dnes už můžeme setkat s desítkami různých variant. Gró metody spočívá, jak již název napovídá, v odečítání zpracovávaného snímku od pozadí. Při použití této metody odečítáme pixely jeden po druhém od referenčního snímku (pozadí), a to co nám zbyde, je popředí. Přestože existuje nepřeberné množství variant algoritmů pracujících na principu odečítání pozadí, jedno mají vždy společné. V článku [1] autoři zmiňují čtyři body: preprocessing, background modeling, foreground detection a data validation.

Preprocessing

Než začneme rozhodovat, který objekt ve videu se hýbe, je dobré vstupní data nějakým způsobem upravit. Ve většině systémů počítačového vidění využíváme k potlačení šumu způsobeného kamerou jednoduché rozmazání Gaussovým filtrem (Gaussian smooth). Rozmazání také můžeme použít pro odstranění vlivu prostředí, jakým jsou déšť nebo sníh. V systémech pracujících v reálném čase si dovolíme zmenšit velikost jednotlivých snímků například převodem do stupňů šedi, nebo některé snímky zcela vypustit. Tím docílíme toho, že nemusíme zpracovávat velké množství dat. Pokud videozáznam pochází z pohyblivé kamery, nebo z více kamer na různých místech musíme zajistit správné překrytí následujících snímků, než přistoupíme k modelování pozadí.

Background modeling

Modelování pozadí je stěžejní částí algoritmu odečítání pozadí. Nejvíce výzkumu bylo věnováno vývoji takového modelu pozadí, které by bylo dostatečně odolné vůči změnám prostředí a současně natolik citlivé, aby pokrylo všechny oblasti zájmu. Podle doby, kdy k modelování pozadí dochází, můžeme jej rozdělit na statické a dynamické.

Statické pozadí vytváříme v počáteční fázi algoritmu a během dalšího postupu algoritmem již toto pozadí neupravujeme. Výhodou je jednoduchost implementace a nenáročnost na výpočetní výkon. Největším nedostatkem této metody je, že neuvažuje možnost zastavení objektu v zorném poli. Například osoba prohlížející výlohu je stále vyhodnocována jako pohyblivá, ačkoliv je v rámci snímku nehybná. Dalším příkladem budiž nějaký odložený předmět. V modelující fázi se v záběru nenacházel, v detekční fázi ho někdo umístí do záběru a ten je vůči statickému pozadí stále vyhodnocován jako pohyb. Druhým významným neduhem, který tato metoda trpí je citlivost na stálost jasu. Změní-li se byť jen nepatrně osvícení celé scény, rozsvítí se v detektoru celý snímek jako pohyblivý. Tento nešvar je ale možné omezit, nebo zcela odstranit použitím vhodné metody detekce popředí.

Oproti tomu stojí modelování dynamické, ve kterém je model pozadí neustále updatován. To má zpravidla za následek vyšší výpočetní a režijní náklady, které jsou ovšem vykoupeny odolnosti vůči změně jasu a zapomenutým předmětům. V této práci se budeme věnovat dvěma způsobům modelování dynamického pozadí 3.1.2.

Foreground detection

Při detekci popředí porovnáváme vstupní snímek videa s modelem pozadí a identifikujeme pixely, které náleží do popředí. V algoritmech [2] se setkáváme s použitím jednoho konkrétního snímku modelujícího pozadí. Tento může být buď neměnný pro celou dobu vykonávání algoritmu, nebo proměnlivý, jak jsme již uvedli. Nejpoužívanější metodou pro detekci popředí je kontrola, zda se vstupní pixel nějak významě liší od odpovídajícího modelu pozadí. Tomu odpovídá rovnice 3.1:

$$|I_t(x,y) - B_t(x,y)| > T \tag{3.1}$$

V další možnosti detekce popředí využíváme normalizované statistiky 3.2:

$$\frac{|I_t(x,y) - B_t(x,y) - \mu_d|}{\sigma_d} > T_s \tag{3.2}$$

kde μ_d a σ_d jsou střední hodnota a standardní odchylka $I_t(x,y) - B_t(x,y)$ pro všechny pixely (x,y). Většina systémů určuje hodnoty prahů T, T_s experimentálně nebo z histogramů. V ideálním případě bychom získavali hodnoty prahů z funkce souřadnic pixelů (x,y). Kupříkladu oblasti s nižším kontrastem by měli získat nižší hodnotu prahu a naopak. Získanou masku popředí po provedení prahování nazýváme též *binární maskou*, protože obraz je zde reprezentován pouze dvěmi hodnotami; nulou pro pixel pozadí a jedničkou pro pixel popředí.

Data validation

Za potvrzování dat považujeme proces vylepšování binární masky zakládající se na informacích získaných z modelu pozadí. U modelu pozadí máme hned několik nedostatků.

Zanedbávají jakoukoliv korelaci sousedících pixelů, rychlost adaptace modelu nemusí stíhat rychlost pohybujících se objektů a drobné pohyby listí nebo stínů mohou být chybně vyhodnoceny jako popředí.

Projevem prvního problému jsou malé oblasti chybně určené jako popředí, nebo pozadí rozmístěné náhodně po celé masce. Použitím morfologického filtru lze osamocené pixely odstranit a pospojovat blízké oblasti. Také můžeme aplikovat erozní filtr, který osamocené pixely odstraní, a následně diletační filtr, který který vrátí objetkům pixely odstraněné erozním filtrem a částečně vyplní díry v objektech. Pro úplné vyplnění děr objektů je potřeba aplikovat diletační filtr několikrát, ale musíme si uvědomit, že tím se zvětší i objekt samotný a původní informace o hranicích objektu bude znehodnocena.

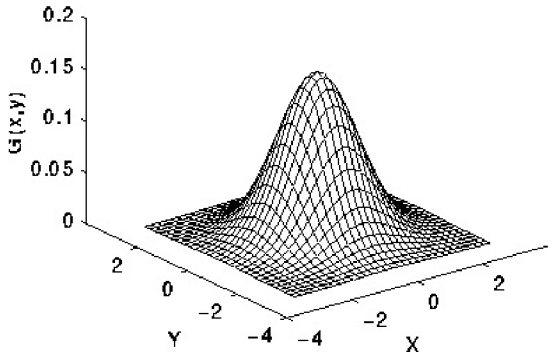
Když se model pozadí přizpůsobuje pomaleji, než se mění popředí, objevují se v binární masce oblasti chybně označené jako popředí. Naopak, přizpůsobuje-li se model pozadí rychleji dochází k ignorování částí pohybujících se objektů. Tento nežádoucí efekt můžeme odstranit použitím více modelů pozadí a následným porovnáním získaných masek popředí. Pokud máme k dispozici původní snímek, můžeme rozšířit binárni masku o pixely podobné barvy, jako mají již označené pixely, jelikož je tu vysoká pravděpodobnost, že náleží stejnému (pohyblivému) objektu. Avšak musíme počítat s patřičnými režijními nároky.

3.1.1 Gaussovo rozmazání

Jak je uvedeno v [4] Gaussův rozmazávací operátor je dvourozměrný konvoluční operátor, který používáme k odstranění šumu v obraze. V tomto smyslu je podobný průměrovému filtru, kde je hodnota pixelu určena na základě devítiokolí. Na rozdíl od průměrového filtru Gaussův nepoužívá pro výpočet průměrnou hodnotu devítiokolí, ale zapracovává vzdálenost od počítaného bodu. Vzdálenější body se na výsledné hodnotě nepodílí tak významně jako body bližší, nebo bod sám. Výpočet hodnoty Gaussova filtru pro pixel (x, y) je dán vztahem 3.3:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.3}$$

Kde σ je směrodatná odchylka distribuce. Grafické vyjádření je na obrázku 3.1.



Obrázek 3.1: Grafické vyjádření rovnice Gaussova filtru. Zdroj [3]

Pro názornost uvedeme konvoluční matici vzniklou aproximací rovnice 3.3 pro $\sigma = 1.0$ a $(x, y) = (0, 0)$.

$$\frac{1}{273}\begin{vmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{vmatrix}$$

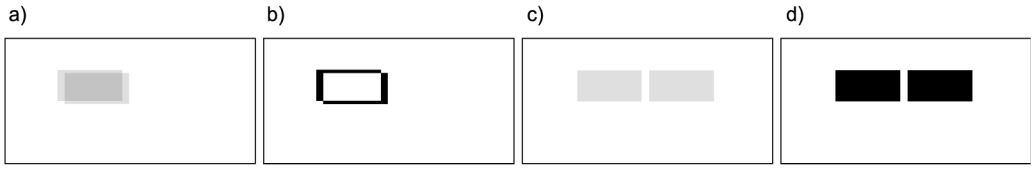
3.1.2 Metoda s bufferem pozadí

Jednou z věcí, které můžeme při modelování dynamického pozadí použít, je buffer snímků. Do tohoto bufferu vkládáme vstupní snímky a za pozadí prohlásíme průměrnou hodnotu ze všech vložených snímků. Druhou variantou téhož může být vybrání mediánu.

První nevýhoda je jasná: buffer snímků zabírá sám o sobě hodně paměti. Uvažujeme-li snímek o velikosti 640x480 pixelů, a pro každý pixel 32 bitů RGBA, pak se dostáváme na 1200 KiB, nebo na 300 KiB při překódování snímku na stupně šedi.

3.1.3 Metoda bez bufferu

Snad ještě jednodušší než metoda s bufferem je porovnávání dvou po sobě jdoucích snímků. Zde pozadí získáme pouhým odečtením těchto snímků. Tado metoda má hned několik nedostatků. Pokud se sledovaný objekt pohybuje příliš pomalu, je jako pohyb vyhodnocena pouze přední a zadní část objektu, ve směru pohybu. Druhým případem je situace, kdy se objekt pohybuje příliš rychle. Pak jsou vyhodnoceny jako pohyblivé 2 objekty. Oba dva případy jsou pro ilustraci na obrázku 3.2.



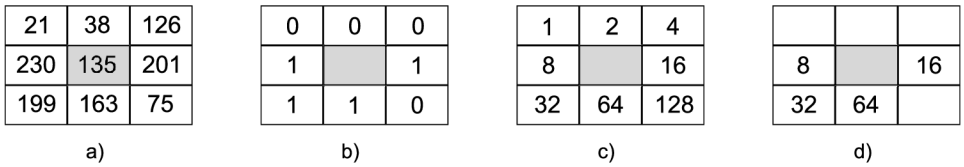
Obrázek 3.2: Odečítání metodou bez bufferu.
a) příliš pomalý objekt, b) po odečtení, c) příliš rychlý objekt, d) po odečtení

Na druhou stranu, při využití této metody nám stačí si pamatovat pouze dva snímky. Také režijní náklady jsou nižší než u předchozí metody.

3.1.4 Local Binnary Patterns

Local binary patterns (LBP) je další z metod modelování popředí. První zmínky o ní nalézáme v článku finských vědců [10]. Jejím zaměřením je především rozpoznávání objektů v obraze. Uplatnění i modifikací je mnoho. Například k rozpoznání obličejů byla použita v [8] a [6]. Mezi její výhody patří tolerance ke změně jasu, vysoká rychlost a v neposlední řadě nízké nároky na systém. Změna jasu je nežádoucí. Setkáme se s ní jak v interiérech, tak i exteriérech. LBP snese změnu slunečního svitu ať už se jedná o pokles způsobený mraky, nebo denní dobou. Toho při použití metody dosáhneme tak, že jednotlivé pixely snímku popisujeme jako funkci jejich sousedících pixelů. Dojde-li k nějaké změně, a je-li tato změna plošná, pak se určitý pixel změní velice podobně jako jeho okolí, a funkce, kt. daný pixel

hodnotí, mu přiřadí stejnou, nebo velice podobnou hodnotu jako v době před změnou. V algoritmu pracujeme s jednou sloužkou obrazu. Může se jednat o jasovou složku, nebo o barevé složky převedené do stupňů šedi. Na obrázku 3.3 je znázorněn proces hodnotící funkce.



Obrázek 3.3: Princip výpočtu hodnotící funkcí

Na obrázku (a) vidíme situaci na snímku v rámci jedné matice. Ve zvýrazněném poli je pixel, který budeme hodnotit. Okolo něj se nachází osmiokolí a to se na hodnocení také podílí. Hodnocení provedeme tak, že na každou pozici osmiokolí umístíme 1, pokud je hodnota v daném místě větší než hodnota porovnávaného pixelu, nebo 0, pokud je nižší. Tuto situaci vidíme na obrázku (b). Nyní těmito hodnotami vynásobíme matici, která se skládá z mocnin 2^n, kde n vyjadřuje pořadí prvku v matici (vyjma prostředního prvku). Vzniklá matice se skrývá pod (c). Výsledky násobení (d) sečteme a součet zapíšeme jako novou hodnotu hodnoceného pixelu. Tuto operaci provedeme pro všechny pixely obrazu s tím, že okrajové pixely můžeme ignorovat.

Se snímky vytvořenými pomocí LBP nakládáme dále jako s plnohodnotnými obrazovými snímky. To znamená, že z nich modelujeme pozadí, provádíme odečet a určujeme popředí. Oproti standardní metodě odečtu má LBP výhodu ve výrazně nižší citlivosti vůči změně osvětlení.

3.1.5 Technologie štítkování

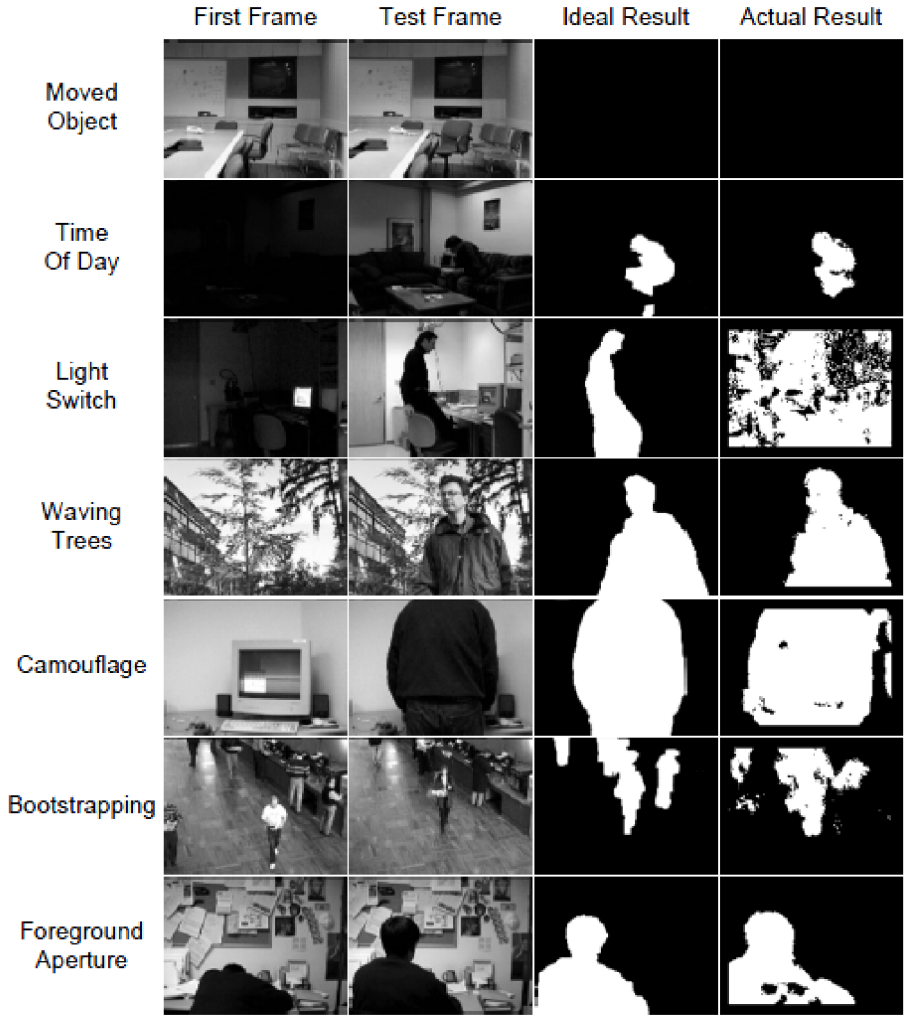
Jednou z nejběžnějších operací v počítačovém vidění je nalezení spojitých oblastí ve snímku. Body ve spojité oblasti tvoří objekty. Pokud se ve snímku nachází pouze jediný objekt, nepotřebujeme hledat spojité oblasti. Avšak daleko častěji dochází k opačnému jevu. V tom případě jsme nuceni rozhodnout, ke které oblasti patří ten či onen pixel.

Algoritmus štítkování nachází všechny spojité oblasti snímku a přiřazuje jedinečný znak - štítek každému pixelu dané oblasti. V literatuře [7] se můžeme setkat s rozdělením na dva typy algoritmů: rekurzivní a sekvenční.

3.1.6 Sekvenční metoda

Sekvenční verze algoritmu obvykle vyžaduje 2 běhy snímkem. V prvním běhu algoritmus sleduje okolí pixelu a přiřazuje mu stejný štítek, jako má jeden z jeho sousedů. Pro případ, že jeho dva sousedi mají štítek odlišný, máme připravenu *tabulka rovnosti*, ve které uchováváme údaj o rovnosti těchto dvou štítků. Výše zmíněnou tabulku pak používáme v druhém běhu, kdy ekvivalentní štítky sdružujeme pod jediný, jedinečný v rámci spojité oblasti.

V algoritmu máme tři případy zájmu při průchodu zleva doprava a zeshora dolů. V prvním běhu sledujeme pouze dva sousedy ze čtyřokolí pixelu. Sousedy nad a vlevo od zkoumaného pixelu. Povšimněme si, že tito sousedé již museli být algoritmem označeni. Pokud ani jeden ze sousedů není označem, přidělíme pixelu novou značku a zároveň zapíšeme údaj



Obrázek 3.4: Demonstrace LBP, zdroj [5]

o novém štítku do tabulky rovností. Pokud byl označen pouze jeden soused, přidělíme pixelu právě jeho značku. Jestliže jsou označeni oba sousedé stejnou značkou, pixelu dáme stejnou, pokud se ale sousedé ve značce lišili, vybereme jednu ze značek, tou označíme pixel a do tabulky rovnosti uvedeme údaj o tom, které dva štítky jsou ekvivalentní.

Tabulka rovností nám obsahuje informace o tom, jaké štítky patří jedné spojité oblasti. V druhém běhu vybereme jeden ze štítků jako vzor a prohlásíme jej za znak spojité oblasti. Po té, co jsme prošli všechny oblasti, přečíslujeme oblasti tak, aby jsme se zbavili mezer ve značení a projdeme celý snímek znovu. Tentokrát přiřazujeme pixelům přímo značku oblasti, pod kterou spadají.

3.1.7 Rekurzivní metoda

Rekurzivní algoritmus je na sekvenčních procesorech velmi neefektivní, ale použijeme-li jej na paralelních procesorech, oceníme jeho sílu. Princip algoritmu je následující. Procházíme

snímek tak dlouho, dokud nenarazíme na pixel popředí. Tomuto pixelu přiřadíme štítek S a rekurzivně všem jeho sousedům. Pokud se v okolí již nenachází neoznačený pixel, pokračujeme v průchodu snímkem. V práci budeme vycházet z obou metod.

Kapitola 4

Návrh systému

Tato kapitola se zabývá návrhem aplikace. Účelem aplikace je demonstrace funkčnosti detekčních algoritmů. V návrhu aplikace je kladen důraz na zpracování v reálném čase. Uživatelské rozhraní proto stojí v pozadí zájmu. Pro názornost funkce detekčních algoritmů jsou při běhu programu poskytovány video výstupy demostrující některé činnosti algoritmů.

K implementaci navrhované aplikace jsou zvoleny jazyky C a C++, spolu s volně šiřitelnou knihovnou OpenCV, za současného použití knihoven CUDA. Knihovna OpenCV je použita pro práci s obrazovými daty videosekvence, ať už je zdrojem soubor ve formátu .avi, nebo je vstupem videostream z webkamery. Knihovny CUDA zajišťují práci s grafickou kartou a především rozšiřují jazyk C o sadu klíčových slov k tomu určených.

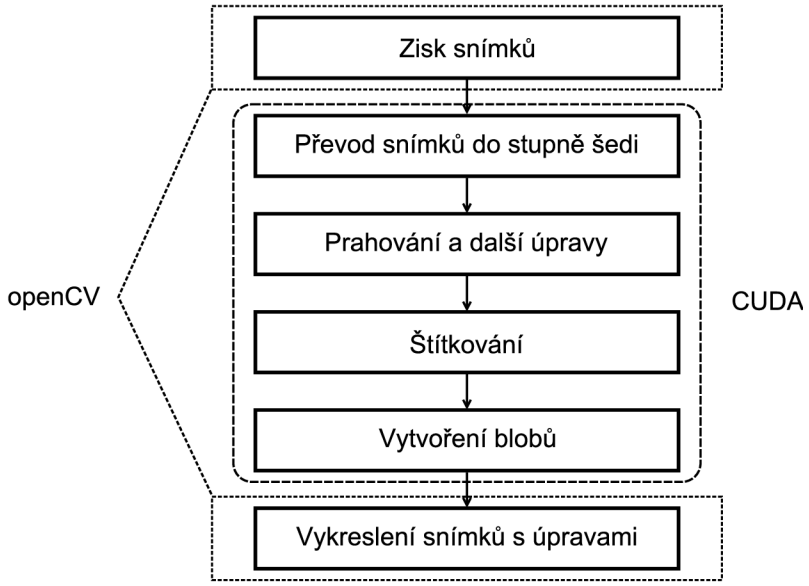
4.1 Struktura aplikace

Aplikace je koncipována tak, aby byla schopna zpracovávat požadavky ohledně získání, práce a zobrazování vstupních/výstupních dat v reálném čase. V návrhu aplikace řešíme problémy přístupu k obrazovým informacím, efektivní zpracování obrazových dat, úpravu algoritmů pro paralelní zpracování a vizualizaci zprocesovaných dat. Na základě poznatků uvedených v 3.1 byla navržena struktura aplikace. Ta je znázorněna na obrázku 4.1 spolu s vyznačením, které knihovny jsou použity v které části aplikace.

Vstupem aplikace je videosekvence. Tu získáme buď ze souboru uloženého pomocí běžného kodeku ve formátu .avi, nebo můžeme považovat za zdroj připojenou webkameru. Vzhledem k tomu, že při testování budeme měnit různé parametry detekčních algoritmů a modifikovat použité metody, nebo je dokonce měnit, je výhodnější mít stálý videozáznam. Proto budeme preferovat vstup z videosouboru před vstupem z kamery.

Než na načtených snímcích budeme provádět jakoukoliv detekci, je vhodné, abychom ony snímky upravili. K tomu, abychom redukovali zašuměné pixely použijeme jednoduchého Gaussova rozmazání popsaného v 3.1.1. Dalším postupem je vytvoření modelu pozadí. Zde vyzkoušíme obě varianty, jak variantu s bufferem, tak bez něj 3.1.2, přičemž pozadí nebude statické, ale dynamické. Vzhledem k tomu, že hlavní program běží na CPU a nikoliv na GPU, budeme pozadí modelovat bez využití možností CUDA.

Po vytvoření pozadí můžeme přistoupit k detekci popředí. Tu už budeme provádět na CUDA zařízení. Jednou z věcí, kterou musíme vyřešit je přesun aktuálního snímku a modelu pozadí do paměti zařízení. Můžeme přitom přenášet obraz s původní barevnou hloubkou, nebo se pokusit o kompresi převodem do stupňů šedi, nebo redukcí barevného prostoru z 24 bitů RGB, na 8 bitů RGB. Redukci můžeme provádět pomocí knihoven OpenCV, nebo



Obrázek 4.1: Struktura systému

už můžeme využít výpočetního výkonu grafické karty. Při využití grafické karty existují dva možné způsoby: implementace funkce, která obdrží barevný obraz a bude vracet obraz redukovaný; nebo druhý způsob, kdy se redukce bude provádět až v rámci detekce pohybu. První zmíněná varianta má velkou nevýhodu v tom, že paměť mezi hostem a zařízením kopírujeme až čtyřikrát. Kopie barevných obrazů na zařízení a redukovaných zpátky a kopie redukovaných obrazů na zařízení za účelem detekce popředí a zpět. Zatímco tento přístup je programátorsky čistší, druhá výše zmíněná varianta je efektivnější a proto ji zvolíme.

4.1.1 Redukce barevného prostoru

Jelikož detekčních algoritmy vytváří detekované popředí jako binární masku je vhodné použít takovou datovou strukturu, ze které je vytvoření této masky snadné a efektivní. Za tímto účelem převedeme původní barevné snímky na stupně šedi.

Algoritmus převodu je notoricky známý, a tak zde jen připomeneme vzorec 4.1, kterým se převod řídí:

$$I = 0,299R + 0,587G + 0,144B \tag{4.1}$$

Kde R je hodnota červené složky, G je hodnota zelené složky a B je hodnota modré složky.

Sekvenční zpracování probíhá tak, že je výše uvedený vzorec aplikován na všechny pixely obrazu. V paralelním zpracování to nebude jinak. Avšak zde nebude existovat pouze jedno vlákno pro celý obraz, ale naopak, každý pixel bude mít své vlákno. Jediné, co musíme vymyslet je, jak tyto pixely (respektive vlákna) sdružíme do bloků. Obraz můžeme považovat za dvourozměrnou matici a tak i bloky mohou být dvourozměrné. Pokud zvolíme blok jako čtverec o délce strany 16, pak se v jednom bloku nachází $16^2 = 256$ vláken, což je jedna z ideálních možností počtu vláken na blok z hlediska efektivity zpracování.

Na šedotónových obrazech ještě můžeme provést úpravu LBP metodou, jejíž princip je popsán v 3.1.4. I zde můžeme vlákna sdružovat do bloků $16x16$ vláken, kde jedno vlákno reprezentuje jeden konkrétní pixel.

4.1.2 Detekce pohybu

Ať už máme oba porovnávané snímky (aktuální snímek a model pozadí) ve stupních šedi, nebo upraveny LBP metodou. Metodou prostého odečtění těchto snímků od sebe získáme popředí, tedy pohyblivé objekty. V této fázi také přistupujeme k jednotlivým pixelům obrazu, takže jednotlivá vlákna počítají jedny konkrétní pixely.

Výsledkem odečítání je další snímek ve stupních šedi, kde vystupují pouze ty objekty, které byly vyhodnoceny jako pohyb. Pro další práci musíme výsledný snímek převést na binární masku. Toho dosáhneme tak, že provedeme prahování s vhodně zvoleným prahem. Vzhledem k tomu, že jednotlivá vlákna nemají představu o dění mimo svůj blok, volíme jeden globální práh pro celý snímek. Ten stanovíme empiricky, na základě testování. Prahování můžeme také provést již v rámci odečítání pozadí.

Výsledná maska bude pravděpodobně obsahovat osamocené pixely způsobené šumem ve snímku, který nebyl Gaussovým filtrem zcela odstraněn. Téměř úplně odstraníme osamocené pixely užitím erozního filtru. Na druhou stranu, maska bude obsahovat i objekty s díramy, které se při aplikaci erozního filtru ještě zvětší, proto po použití erozního filtru bude ihned následovat diletační filtr. Pokud budou díry v objektech větších rozměrů, provedeme diletaci i vícekrát.

V těchto algoritmech můžeme opět ponechat 256 vláken na dvourozměrný blok.

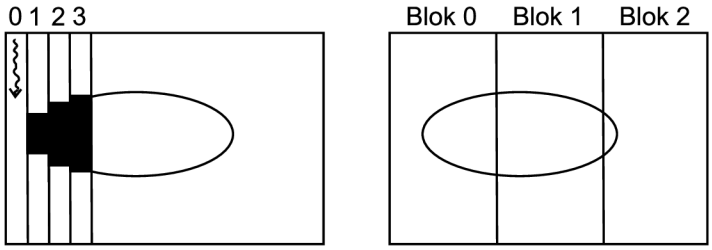
4.1.3 Štítkování

Již jsme získali binární masku. Náš další postup bude vytvoření znalosti o objektech v binárni masce obsažených. K tomu použijeme štítkování (labeling). Jak jsme zmínili v 2.1, je nutné algoritmus štítkování z 3.1.5 upravit tak, aby mohl běžet ve vláknech a jeho paralelním zpracováním došlo ke zrychlení původního sekvenčního přístupu. Způsobů, jak toho dosáhnout, bychom jistě pár vymysleli. Zde popíšeme ten, který byl předmětem konzultace.

Původní algoritmus prochází snímek po řádcích zleva doprava a přiděluje aktuálním pixelům značky. Tento přístup je při dvojím sekvenčním zpracování dostačující. Uvažujme, že sledované objekty se budou ve snímku pohybovat převážně zleva doprava, nebo zprava doleva. Pak by při paralelním zpracování, kde bychom v jednom vláknu procházeli jediný řádek, mohlo docházet k tomu, že některá vlákna by zpracovávala velké množství objektů a naopak některá vlákna žádné. Proto přistoupíme na průchod snímku po sloupcích, kde budeme očekávat rovnoměrnější rozložení zátěže. Rozložení vláken a bloků vláken je symbolicky znázorněno na obrázku 4.2.

V původním algoritmu probíhá zapisování značek přímo do snímku (binární masky popředí). To se během implementace a testování ukázalo jako nevýhodné především kvůli tomu, že se maska popředí nachází v globální paměti karty a přístup do této paměti patří k nejpomalejším. Proto vznikla potřeba vytvořit datovou strukturu ve sdílené blokové paměti, která je rychlejší.

Obrázek 4.3 ilustruje dvě varianty datové struktury určené jednomu vláknu. V obou variantách vidíme čtveřici tabulek, které reprezentují jednotlivé shluky pixelů. Čtveřice proto, že uvažujeme až 4 shluky detekované v rámci jednoho vlákna. Do řádků budeme ukládat údaje o začátku a konci shluku. Pokud bychom uvažovali 1B na jeden údaj, uchovali bychom informaci pouze o $2^8 = 256$ řádcích videa. Dnešní kamery dosahují rozlišení daleko



Obrázek 4.2: Schéma funkce algoritmu

většího a proto zvolíme 2B na jeden údaj, což nám poskytne prostor pro video s až $2^16 = 65536$ řádky. To je pro naše účely více než dostačující. Údaj o šířce videa můžeme zanedbat, protože ten nám setřou bloky, které mají pevnou šířku 128 vláken (potažmo sloupců).

Velikost datové struktury jednotlivých bloků je v obou případech 8B. Tento rozměr volíme záměrně, kvůli sdruženému přístupu do paměti. Ve variantě *a*) do pole *y min* vkládáme informaci o začátku shluku, do pole *y max* o konci shluku v rámci jednoho sloupce. Do pole *label* uvádíme informaci o přiděleném štítku. Pole *align* slouží pouze k doplnění struktury na požadovanou velikost osmi bytů.

Pokud bychom se rozhodli pro variantu *a*), setkali bychom se s několika problémy. Zaprvé, musela by existovat funkce nadřazená všem vláknům, která by obstarávala řádnou distribuci štítků. Vlákna totiž mezi sebou komunikují prostřednictvím sdílené paměti a režie tímto způsobem by mohla být náročná. Zadruhé, spojování jednotlivých shluků v objekty by vyžadovalo další datovou strukturu, ve které by se uchovávaly informace o tom, jaké štítky jsou obsazeny konkrétním objektem spolu s údaji o umístění objektu v obraze (jeho souřadnice).

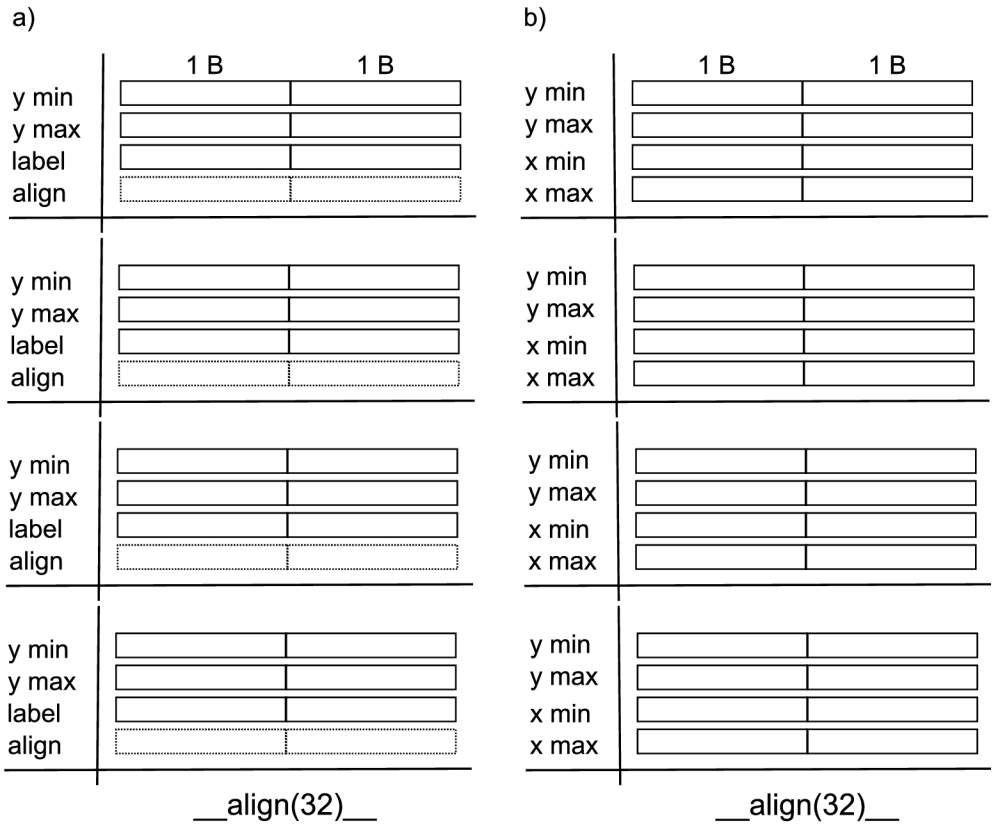
Ve variantě *b*) vkládáme po řadě: informaci o začátku shluku, informaci o konci shluku, informaci o začátku shluku ve sloupci a informaci o konci shluku ve sloupci. V posledních dvou zmíněných uvádíme v této fázi detekce stejný údaj - index vlákna v rámci celého obrazu. Je to logické, protože v této části detekce shluk začíná i končí ve stejném sloupci. Tento způsob nám nabídne jednodušší práci v etapě slučování, kdy nebudeme potřebovat žádnou další datovou strukturu, všechny informace jsme schopni získat již z těchto polí. Navíc jsme přišli o štítky, takže slučovat budeme na základě blízkosti shluků.

Jak jsme již zmínili, uvažujeme pouze čtyři shluky ve sloupci, jakýkoliv další shluk je ignorován. V případě potřeby by bylo možné strukturu rozšířit o několik dalších shluků, ale tím bychom mohli přijít o výhody plynoucí ze sdruženého přístupu do paměti (ze sdílené paměti musíme informace překopírovat do globální paměti a odtud do paměti hosta) a velikost sdílené paměti je limitovaná. Její maximální přípustná velikost se mezi kartami liší, proto zde neuvedeme přesné číslo, pouze podotkneme, že současně navrhovaná velikost sdílené paměti nepřevyšuje nejmenší maximální velikost. Aby nám ve struktuře nezůstávali malé shluky na úkor velkých, můžeme zaznamenávat pouze shluky například větší než pět pixelů.

4.2 Slučování v bloby

Jakmile vlákna naplní všechny struktury, můžeme přistoupit ke slučování těchto shluků. Na obrázku 4.4 je znázorněn pricip slučování.

Jednotlivá vlákna přistupují ke struktuře a prochází ji v několika iteracích. V každé



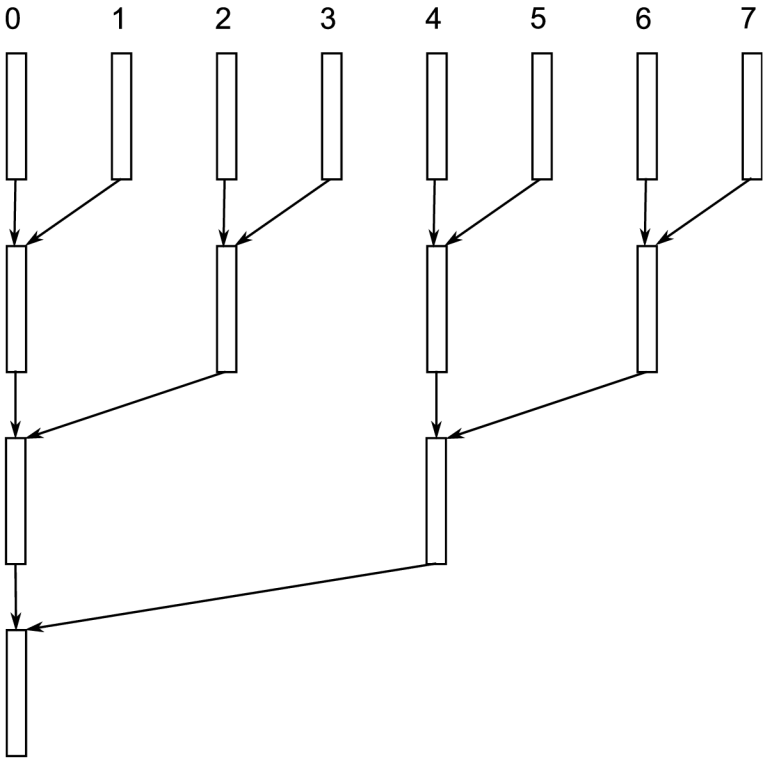
Obrázek 4.3: Návrhy struktury pro uchování informace o blobech.

iteraci jsou porovnávany shluky dvou sloupců mezi sebou podle pravidla každý s každým. V první iteraci jsou porovnávány shluky bezprostředně sousedících sloupců. V druhé iteraci sloupců vzdálených o 2^2, v třetí o 2^3, až do maxima 2^8. Maximum 2^8 není zvoleno náhodně, je to počet vláken na blok. Za tuto mez již vlákna nemají přístup a v rámci jednoho bloku jsou již všechny shluky porovnány.

Při porovnávání vyhodnocujeme vzdálenost blobů. Zaprvé, jestli se překrývají ve své y souřadnici a za druhé, jestli jsou poblíž z pohledu x souřadnice. Oba tyto údaje parametrizujeme. Samotné sloučení provádíme tak, že pokud jsou shluky vyhodnoceny jako sousedící, převedeme informace z pravého shluku do levého následujícím způsobem:

- Je-li počátek shluku v ose y pravého sloupce menší než počátek shluku v ose y levého sloupce, zkopírujeme tento údaj z pravého shluku do levého.
- Je-li konec shluku v ose y pravého sloupce větší než konec shluku v ose y levého sloupce, zkopírujeme tento údaj zprava do leva.
- Konec shluku v ose x levého sloupce nastavíme na konec shluku v ose x pravého sloupce.
- Všechny údaje pravého sloupce smažeme.

Tato pravidla nám zaručí správné sloučení stejně tak dobře v první iteraci, jako v jakékoliv další. Dovolíme si ještě jistou optimalizaci.



Obrázek 4.4: Princip slučování blobů.

Během porovnávání může dojít k situaci, kdy shluk levého sloupce je prázdný, zatímco shluk pravého sloupce nikoliv. V takovém případě je vhodné, abychom informace z pravého shluku přesunuli do levého shluku. Principem celého slučování je setřepávání shluků co nejvíce vlevo, tak toto pravidlo dodržíme. Může se také stát, že v průběhu slučování dojde k tomu, že se shluky překrývají i v rámci jednoho sloupce. Proto na konec etapy slučování provedem ještě sjednocení v rámci každého sloupce.

4.3 Vykreslení

Máme-li hotovou fázi slučování, přistoupíme k vykreslení detekovaných objektů. Strukturu ještě naposledy celou projdeme. U každého shluku známe informaci kde začíná a kde končí. Tyto informace můžeme považovat za levý horní roh a pravý spodní roh obdélníka, který vykreslíme.

Kapitola 5

Implementace

Aplikace je implementována v jazyce C s využitím jedné funkce C++ v OS Windows a ve vývojovém prostředí Microsoft Visual Studio 2008. Aplikace je přenositelná i na operační systém Linux. Pro práci je použita volně šiřitelná knihovna OpenCV, určená pro alikace počítačového vidění. Další použitou knihovnou je CUDA. Její služby jsou využity při práci s grafickou kartou.

5.1 OpenCV

OpenCV (Open Source Computer Vision) je knihovna programových funkcí pro počítačové vidění v reálném čase [11]. Je vydána pod licencí BSD a je volně použitelná pro komereční i nekomerční účely. Knihovna je napsána v jazyce C a C++. Výrazně usnadňuje práci s obrazem a videem, takže není nutné zdlouhavým způsobem vypracovávat rutiny pro práci s videozáznamem. Aktuální verze je OpenCV 2.2, pro implementaci je použita verze 2.1. Knihovna je neustále vyvíjena, ale zachovává si kompatibilitu nových verzí knihovny s programy, napsanými ve staré verzi.

5.2 CUDA

Pro implementaci funkcí běžících na grafickém zařízení jsou využity knihovny CUDA. Tato knihovna je volně ke stažení všem programátorům. Umožňuje psát v mnoha jazycích, z nejběžnějších C, C++, Fortran, Python. Aplikace byla implementována při využití knihovny verze 3.1, později 3.2. V době dokončování práce existuje verze 4.0 RC, dostupná zatím jen pro registrované programátory.

5.3 Aplikace

Vlastní implementace systému spočívá v rozdělení kódu na tři části podle vzoru v Nvidia GPU Computing SDK.

V první části (main.cpp) je řešeno načítání obrazu, jednoduché GUI aplikace, označování zjištěných blobů a vstupně výstupní záležitosti.

V druhé části (wrapper.cu) je prováděna režie v rámci grafické karty, alokace a uvolnění paměti, volání kernelů.

Ve třetí části (md_kernel.cu) jsou definovány kernely, které běží na grafické kartě. Také se zde nachází definice některých pomocných struktur a definice parametrů běhu programu.

5.4 Komunikace CUDA - OpenCV

Knihovna OpenCV používá k uložení snímku vlastní strukturu nazvanou IplImage. Pro výpočty na GPU je nutné přenést obrazová data na GPU. Toho dosáhneme buď tím, že CUDU naučíme strukturu IplImage, nebo budeme kopírovat samotná data. Struktura IplImage obsahuje totiž navíc větší množství přidružených informací, které jsou užitečné v různých jiných aplikacích, avšak v této aplikaci nám stačí, budeme-li znát rozměry snímku a jeho data. Z tohoto důvodu jsou na GPU kopírována pouze obrazová data, ke kterým je přidána informace o rozměrech. Tím dosáhneme nižšího objemu dat k přenosu a tím i rychlejšího přenosu host - zařízení.

5.5 Ovládání aplikace

Aplikace je koncipována jako konzolová aplikace ovládaná pomocí příkazového řádku. Aplikace se spouští příkazem:

```
>md.exe [flags][filename]
```

Kde `flags` představují možnosti:

- `-f filename` program bere jako vstupní videosekvenci data obsažená v souboru `filename`
- `-h` program vypíše nápovědu a ukončí se
- `-o filename` program ukládá výstup do souboru `filename`
- pokud je program spuštěn bez parametrů, pokusí se připojit ke straemu z kamery, není-li tato k dispozici, program vypíše chybovou hlášku a ukončí se

Kapitola 6

Testování systému

Pro účely testování budeme pracovat se záznamy. Ty mají oproti živému streamu neoddiskutovatelnou výhodu v tom, že jsou při každém spuštění testu stejné a nemůže u nich tedy docházet k různým náhodným jevům. Videozáznamy určené k testování pochází z okolí školy, některé záběry jsou z okolí dálnice D1 a sjezdařské dráhy Fajtův kopec. Jako záznamová kamera byla použita digitální kamera Sony Handycam s rozlišovací schopností 5 MPix. Získaný záznam byl z DVD formátu převeden na formát PAL (rozlišení 720 x 576). Pro převod byl použit software dodávaný s kamerou. Kamera byla umístěna staticky ve volné scéně, takže světelné podmínky nemohly být ovlivněny. Zabíraný provoz je náhodný.

6.1 Určení prahu

Získat přesnou hodnotu prahu použitelnou pro všechny záznamy není úplně jednoduchá věc. Při jeho určování musíme brát v potaz různé náhodné jevy (stíny, pohyblivé listí, šum v obraze, odrazy, objekty podobné barvy jako má pozadí) a četnost jejich výskytu ve videu. Proto se také setkáváme s problémy, kdy odstranění šumu pomocí prahování vede k dalšímu nepříjemnému jevu, a to ke ztrátě části objektů, která měla podobnou barvu jako pozadí. Během testování bylo zjištěno, že ideální hranice prahu se nachází mezi 30 až 40 v závislosti na zabírané scéně.

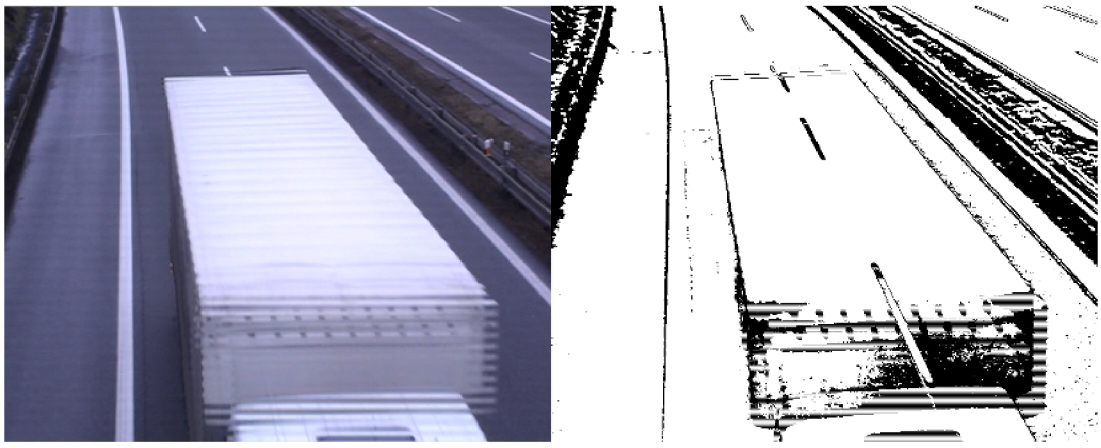
Obrázek 6.1: Demonstrace nízkého prahu.

Například u záznamu, kde se často mění jasová složka, hranici přibližujeme hornímu prahu a někdy i přesáhneme. Jak vypadala scéna při použití malého prahu je znázorněno na obrázku 6.1. Naopak obrázek 6.2 ilustruje variantu s vysokým prahem.

Obrázek 6.2: Ukázka použití vysokého prahu.

6.2 Detekce pohybu

Na obrázku 6.3 je vyobrazen jeden z běžných problémů plynoucích z použití bufferu pozadí. Jelikož je pro snímání použita kamera coby inteligentní zařízení, koriguje si automaticky jas. V případě větší změny ve scéně dojde při odečtu pozadí k vychodnocní celého snímku jako pohybu. Velikost bufferu se na tom úměrně podílí. V obrázku představený nákladní vůz nejen že nebyl detekován, ale znemožnil detekci pro několik dalších snímků, než se model pozadí vrátil do normálu. U tohoto konkrétního případu došlo k eliminaci zmíněného jevu až v případě, kdy byla velikost buffer nastavena na 2, což jej ve své podstatě degradovalo na metodu bez bufferu.



Obrázek 6.3: Ukázka neduhu plynoucího korekcí jasu.

Proto byl zvolen pro další testování přístup úplného vypuštění modelování pozadí s bu-

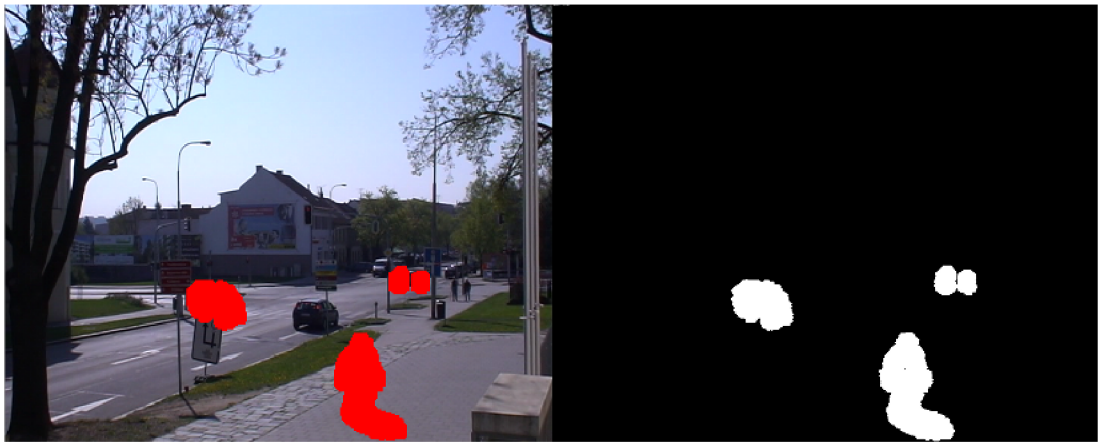
fferem a přešlo se na metodu bez bufferu.

Na obrázku 6.4 je znázorněn případ, kdy vznikla v objektu díra, tento jev můžeme omezit použitím diletačního filtru. V případě potřeby jej použijeme i vícekrát.

Obrázek 6.4: Díra v objektu.

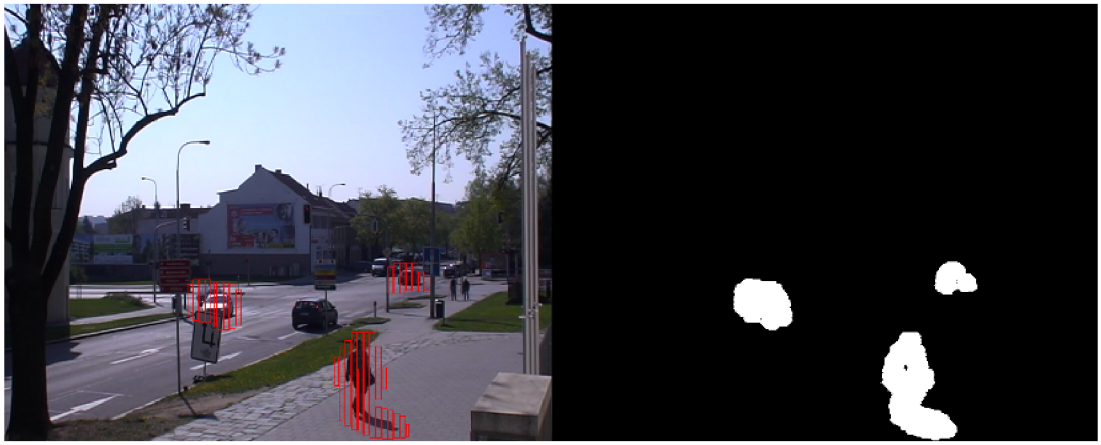
6.3 Test slučovacího algoritmu

V následné sérii obrázků 6.5, 6.6, 6.7, 6.8 je demonstrován test slučovacího algoritmu. V obrázcích je patrný princip slučování. Mimo jiné si v nich můžeme také povšimnout nedostatku ve formě označeného stínu. Ten by se ve zcela dokonalém detektoru nevyhodnotil jako pohyb. Implementace takového detektoru ale vyžaduje velké množství zkušeností.

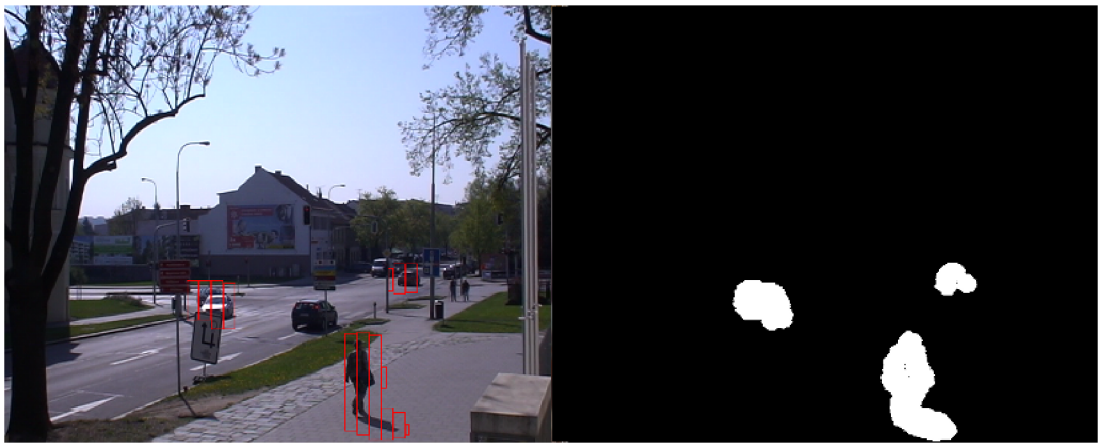


Obrázek 6.5: Slučovací algoritmus v první iteraci.

Obrázek 6.6: Slučovací algoritmus v druhé iteraci.



Obrázek 6.7: Slučovací algoritmus v třetí iteraci.



Obrázek 6.8: Slučovací algoritmus ve čtvrté iteraci.

6.4 Test citlivosti

Další z testů, který si předvedeme je test citlivosti. Smyslem je vyzkoušet, jak malé objekty je možné ještě detekovat jako pohyb, aniž by byly chybně vyhodnoceny jako pozadí. K tomuto účelu nám poslouží kotvy lanovky na Fajtově kopci ilustrované obrázkem 6.9.

Obrázek 6.9: Detekce velmi malých objektů.

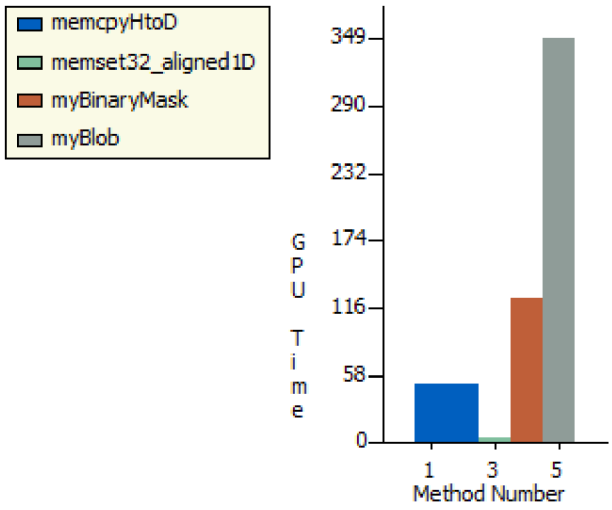
V obrázku 6.9 si rovněž můžeme povšimnout netečnosti vůči mrakům, nebo kývajícím se větvím stromů.

6.5 Test rychlosti

Vzhledem k povaze práce je jasné, že kód běžící na CPU je pomaleji zpracováván, nežli kód běžící na GPU. Dokladem toho budiž například Gaussův filtr, který byl nejdříve implementován s využitím knihovny OpenCV a běžel tedy na CPU. Aplikace Gaussova filtru způsobovala znatelné zpomalení programu, proto bylo při testování rozhodnuto o jeho odstranění. Vzniklé nedostatky v obraze byly kompenzovány použitím erozního filtru a ná-

sledně diletačního filtru, které již byly implementovány s využitím knihoven CUDA a tím pádem na GPU. Toto řešení se ukázalo jako dostatečné a běh programu to urychlilo i přes to, že diletační filtr je použit několikrát.

Pro testování byl také použit měřící nástroj z dílny Nvidie. Řeč je o Compute Visual profileru (VP). Tento nástroj je schopný otestovat program napsaný pomocí knihoven CUDA. Po zadání cesty k vytvořenému programu a zadání argumentů, s kterými se má program spouštět, je benchmark připraven k běhu. VP postupně několikrát spustí testovaný program a poté nabídne výsledky. Využitím VP získáme pro každou použitou metodu informaci o tom, jakou dobu byla obsluhována CPU, jakou dobu GPU, kolik bloků se nacházelo v mřížce, kolik vláken bylo v jednom bloku a jak byla uspořádána, kolik bytů sdílené paměti bylo použito dynamicky a kolik staticky a mnoho dalších informací. V případě práce s pamětí se také dozvíme propustnost pamětí a šířku přenosového pásma. Výsledky jsou prezentovány formou tabulek, ale také i formou grafů. Zde uvedeme graf 6.10 časové obsazenosti GPU jednotlivými metodami.



Obrázek 6.10: Detekce velmi malých objektů.

V grafu je jistě matoucí to, že jsou zobrazeny jen 4 metody. Jedná se totiž o graf vytvořený přibližně v polovině práce. VP po dokončení projektu odmítnul spolupracovat a tak je zde pro názornost a pro budoucí pokračovatele alespoň zmíněn.

Kapitola 7

Závěr

Cílem práce bylo vytvoření detekčních algoritmů, které by běželi na grafické kartě. Technologie CUDA je poměrně mladá, a tak bylo nutné navrhnout a aplikovat úpravy známých detekčních algoritmů. Vzhledem k obsáhlosti celého odvětví byla práce věnována pouze podmožině těchto algoritmů, konkrétně odečtu pozadí. Byly zde popsány metody modelování pozadí, zisku popředí a úpravy vstupních snímků, spolu s náhledem do použité architektury CUDA.

Použitím algoritmu odečtu pozadí byly dosaženy průměrné výsledky. Systém se setkával s řadou běžných chyb při občasné detekci stínů nebo odrazů. V dohledových systémech by mohlo docházet záměně stínů za reálné objekty, což je nežádoucí. Na druhou stranu odrazy se skutečně pohybují s reálnými objekty a rozlišit je, se i člověk musí naučit. Dalším problémem byly mizející objekty. Například auto zastavující na semaforech. Po dobu, kdy auto stálo, bylo správně vyhodnocováno jako pozadí. V momentě, kdy se zase rozjelo, byl to opět pohyblivý objekt. V dohledových systémech by mohlo být užitečné uchovat si informaci o tomto objektu i přestože se nehýbe, kvůli automatické identifikaci. V opačném případě by se totiž jednalo o rozdílné objekty, což je nežádoucí. Rozšíření práce v tomto směru by tedy mohlo být o modul sledující pohyb objektů.

Odstraněním Gaussova filtru z části programu, která byla zpracovávána procesorem, došlo k výraznému urychlení zpracování videa. Tato skutečnost byla kompenzována použitím erozního a diletačního filtur, které již běželi na grafické kartě a nezpůsobovali takové zpomalení. Aplikace modelu pozadí s využitím bufferu byla v některých případech, kvůli své pomalé přizpůsobivosti, vyhodnocena jako nevyhovující. Na straně druhé, použití metody pouhého odečtení dvou po sobě jdoucích snímků nebylo také ideální, protože byly zachyceny pouze okraje objektů. I to šlo kompenzovat diletačním filtrem, který objekty uzavřel do jednoho celku. Významnou částí projektu byla úprava algoritmu štítkování pro paralelní zpracování. Na to navazuje i následné zpracování štítků, taktéž paralelně. Vhodným vylepšením těchto metod by mohl být subsystém, který by si pamatoval stav v předchozím snímku a byl by schopen předvídat trajektorii pohybu objektů.

Z hlediska dalšího vývoje projektu se nabízí možnost přesunout hlavní běh programu z procesoru na grafické jádro. Současný stav je takový, že program je spuštěn na procesoru a pro každý snímek si volá obsluhu grafické karty, která snímek zpracuje a vrátí informace. To je nevýhodné především z hlediska neustálého kopírování paměti z hosta na zařízení a zpátky. Oním přesunutím na grafické jádro by došlo k tomu, že veškeré datové struktury by zůstali uloženy v paměti grafiky a ta by si volala obsluhu procesoru, aby dostala nový snímek. I zde je patrné, že dochází ke kopírování paměti host - zařízení, ale v tomto případě se kopíruje pouze aktuální snímek. Snímek obsahující pozadí zůstává na grafické kartě.

Jiným pohled na věc by mohlo vrhnout uvolnění knihovny CUDA verze 4.0, která podle referencí optimalizuje rychost datových přenosů mezi hostem a zařízením. Použití toho, či onoho přístupu stojí za zvážení.

Literatura

[1] Cheung, S.-C. S.; Kamath, C.: Robust techniques for background subtraction in urban traffic video. *Visual Communications and Image Processing 2004*, ročník 5308, 2004: s. 881–892.

[2] Davies, E.: *Machine vision Theory, Algorithms, Practicalities.* Massachusetts: Morgan Kaufmann Publishers, 2005, ISBN 0-12-206093-8.

[3] Fisher, R.: Gaussian Smoothing. `http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm`.

[4] Gonzalez, R.: *Digital Image Processing.* New Jersey: Upper Saddle River, 2002, ISBN 0-20-118075-8.

[5] Heikkilä, M.; Pietikäinen, M.: A Texture-Based Method for Modeling the Background and Detecting Moving Objects. `http://www.ee.oulu.fi/mvg/files/pdf/pdf_662.pdf`.

[6] Hopjan, T.: *Identifikace osob pro kamerový systém.* Bakalářská práce, FIT VUT, Brno, 2010.

[7] Jain, R.; Kasturi, R.; G.Schunck, B.: *Machine vision.* New York: MCGraw-Hill, 1995, ISBN 0-07-032018-7.

[8] Jelínek, T.: *Detekce pohybujících se objektů ve video sekvenci.* Diplomová práce, FIT VUT, Brno, 2010.

[9] NVIDIA Corporation: Scalable Parallel Programming with CUDA. *ACM Queue*, ročník 6, 2008.

[10] Ojala, T.; Pietikäinen, M.; Harwood, D.: Performance evaluation of texture measures with classification based on Kullback discrimination of distributions. *Proceedings of the 12th IAPR International Conference on Pattern Recognition (ICPR 1994)*, ročník 1, 1994: s. 582–585.

[11] OpenCV: OpenCV Library Wiki. `http://opencv.willowgarage.com/wiki/`.

[12] WWW stránky: GPU Computing. `http://www.nvidia.com/object/GPU_Computing.html`.

[13] Zaorálek, L.: CUDA: optimalizace přístupu do globální paměti. `http://www.root.cz/clanky/cuda-optimalizace-pristupu-do-globalni-pameti/`.

[14] Zeller, C.: Tutorial CUDA. `http://people.maths.ox.ac.uk/gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf`.

Dodatek A

Obsah CD

- Binaries - přeložený binární soubor pro operační systém Windows
- Source - zdrojové kódy programů
- Video - videa získaná za účelem testování
- Tested - demonstrace detekčních algoritmů
- OpenCV - knihovny nutné pro běh aplikace
- Doc - zdrojový text bakalářské práce
- Text bakalářské práce
- Plakát

Dodatek B

Plakát

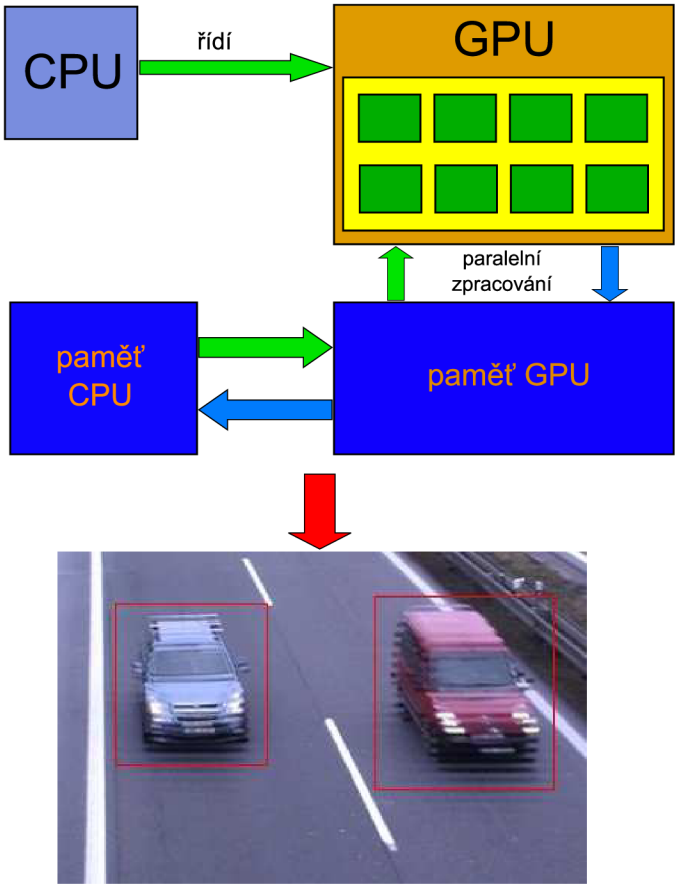
Bakalářská práce

Detekce pohyblivých objektů ve videu na CUDA

Cíle:

Implementovat detekční algoritmy na grafickou kartu.

Zpracování v reálném čase.



František Horák, xhorak06@stud.fit.vutbr.cz, Vysoké učení technické v Brně, Fakulta informační technologií