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 OBLIČEJŮ VE VIDEU NA GPU

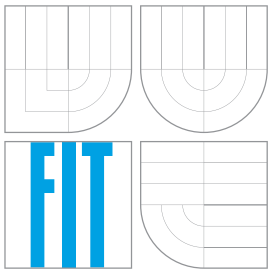
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR
AUTHOR

Bc. MARTIN TESAŘ

BRNO 2012

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 OBLIČEJŮ VE VIDEU NA GPU

FACE DETECTION IN VIDEO ON GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR
AUTHOR

Bc. MARTIN TESAŘ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LUKÁŠ POLOK

BRNO 2012

# Abstrakt

Tato práce se zabývá detekcí obličejů na grafickém procesoru. V první části je uveden přehled metod detekce obličejů se zaměřením na detektor Violy a Jonese. Dále jsou prostudovány možnosti mapování klíčových částí detektoru na grafickou kartu. Další část práce popisuje implementační detaily navržené aplikace. Na konci práce jsou zahrnuty výsledky a porovnání s CPU implementací. Poslední kapitola shrnuje celou práci a navrhuje budoucí možnosti vývoje.



# Abstract

This work deals with task of face detection on graphic card. First part is the introduction to face detection methods focusing on detector proposed by Viola and Jones. Further, this work studies the posibilities of mapping detector's key parts on graphic card. Next part describes implementation details of designed application. The end of work include results and comparison with CPU approach. The last chapter summarizes the whole work and proposes future possibilities of development.



# Klíčová slova

detekce obličejů, integrální obraz, GPGPU, OpenCL, paralelní primitiva

# Keywords

face detection, integral image, GPGPU, OpenCL, parallel primitives


# Citace

Martin Tesař: Detekce obličejů ve videu na GPU, diplomová práce, Brno, FIT VUT v Brně, 2012

# Detekce obličejů ve videu na GPU

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Lukáše Poloka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Tesař
21. května 2012

## Poděkování

Rád bych poděkoval svému vedoucímu panu Ing. Lukáši Polokovi, za jeho ochotu a cenné odborné rady, které mi při řešení práce poskytl.

# Obsah

# Kapitola 1

# Úvod

Detekce obličejů v obraze je vstupní branou pro další algoritmy analýzy obličeje (např. pro verifikaci obličeje, sledování obličeje ve videu, sledování výrazů obličeje a mnoho dalších). Z tohoto důvodu podléhá detekce obličejů neustálému výzkumu. Jsou objevovány nové metody, které se snaží zvyšovat úspěšnost detekce a samozřejmě i rychlost vyhodnocení.

Na detekci obličejů v obraze nahlížíme jako na úlohu, ve které zjišťujeme, v jakých podoblastech obrazu se obličej nachází, přičemž pozice jednoho obličeje nezávisí na pozicích obličejů ostatních. Proto můžeme na detekci nahlížet jako na problém, který lze paralelizovat. Na scénu se dostávají moderní grafické čipy, jejichž architektura je navrhnuta pro paralelní výpočty.

V práci se nejprve seznámíme s obecným přístupem k detekci objektů v obraze, jež spadá do oblasti rozpoznávání vzorů. Poté se zaměříme na rozdělení metod řešících detekci obličejů. Z dostupných metod si vybereme metodu založenou na boostovacím schématu, a další kapitolu věnujeme podrobnému studiu detektoru Violy a Jonese. Nastíníme si různé modifikace metody, které od doby navržení detektoru vznikly. Ve čtvrté kapitole uvedeme tři konkurentní rozhraní z oblasti obecných výpočtů na grafických kartách a shrneme jejich podobnosti a odlišnosti. Pro implementaci si vybereme OpenCL, což je otevřený standard pro paralelní programování napříč různými typy procesorů. Navážeme kapitolou pátou, ve které si probereme možnosti namapování stěžejních oblastí detektoru na grafickou kartu, jejíž základem je paralelizace vyhodnocení klasifikátoru, možnosti uložení klasifiktoru na grafické kartě a tvorba integrálního obrazu. V šesté kapitole navrhneme rozhraní pro naši budoucí knihovnu detekce. V další kapitole se seznámíme s implementačními detaily diplomové práce a zjistíme, jak jsme implementovali jednotlivé algoritmy navržené v páté kapitole. Navíc se objeví nové algoritmy, se kterými jsme se setkali až při řešení. Poté nás čeká krátká kapitola o detailech demonstrační aplikace. V následující kapitole se budeme zabývat testováním aplikace a hledáním nejlepších parametrů pro testovanou architekturu. Na konci kapitoly shrneme dosažené výsledky a porovnáme je s CPU implementací. V závěrečné kapitole si zrekapitulujeme vše čeho jsme dosáhli a nastíníme budoucí vylepšení.

# Kapitola 2

# Rozpoznávání vzorů

Pod procesy, jimiž dokážeme rozeznat obličej, číst knihu, poznat známou melodii, či vůni oblíbené květiny, leží rozpoznávání vzorů [4] – akt, kdy syrové informace jsou roztříděny do příslušných kategorií. Výzvou je naučit stroje rozpoznávat vzory také tak kvalitně a rychle jako lidský mozek, či jej dokonce překonat. Strojové rozpoznávání vzorů se uplatňuje v mnoha odlišných směrech: identifikaci otisků prstů, automatickém rozpoznávání řeči, rozpoznávání vzorů optického charakteru – příkladem je detektor obličejů zabudovaný v digitálním fotoaparátu.

## 2.1 Detekce objektů v obraze

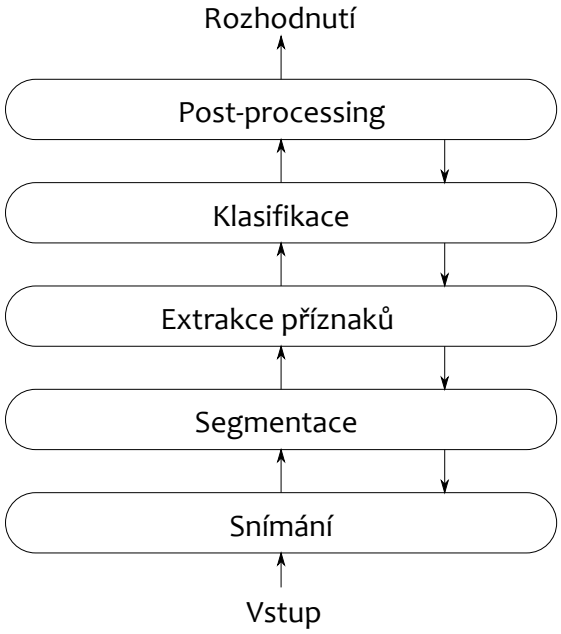
Detekci objektů v obraze chápeme jako klasifikaci, jíž sledujeme, zda se v určitém místě v obraze vyskytuje náš objekt zájmu (v našem případě obličej), nebo jestli se jedná o pozadí. Před klasifikací potřebujeme nalézt vhodné příznaky (*features*) [4], podle kterých budeme objekt detekovat. Příznaky dohromady tvoří příznakový vektor (*feature vector*) [4]. Všechny příznakové vektory pak leží v n-dimenzionálním příznakovém prostoru (*feature space*) [4]. Z příznaků vytvoříme model objektu, podle kterého budeme schopni rozhodovat, jestli se jedná o detekovaný objekt, nebo ne. Klasifikátor, který skenuje vstupní obraz, tohoto modelu využívá. Typický rozpoznávací systém se dělí do několika fází.

První částí je obvykle snímání. V této části převádíme analogový signál na diskrétní, kdy výstupem je pole hodnot (v případě obrazu pole pixelů). Každý snímač je limitován svou charakteristikou-zkreslením, rozlišením, latencí.. Na kvalitě snímání závisí kvalita rozpoznávání. Často snímání nelze ovlivnit, proto v systému musíme počítat s různě kvalitními vstupy.

Následuje segmentace, kdy se snažíme oddělit potencionální objekty zájmu od pozadí a mezi sebou. V detekci obličejů se často používá segmentace obrazu pomocí barvy kůže. Výsledek segmentace barvy kůže je binární obraz, který určuje pixely s barvou kůže. Segmentace není povinná a v úloze detekce se může vynechat. Ideální stav by byl, kdybychom ve fázi segmentace přímo oddělili pozadí od objektů našeho zájmu, detekce by byla hotova.

Úkolem extrakce příznaků je extrahovat příznakové vektory ze zkoumaného signálu a ty předávat klasifikátoru. Náš správný výběr příznaků je základem dobrého rozpoznávacího systému. Výhodné je, volíme-li příznaky, které jsou invariantní vůči všem relevantním transformacím (např. rotace, translace, změna měřítka).

Vstupem klasifikátoru je vektor příznaků. V klasifikátoru vektor porovnáváme a řadíme jej s určitou pravděpodobností do některé z možných tříd. Při detekci obličeje klasifikátor




Obrázek 2.1: *Fáze detekce objeků v obraze.*

určuje s jakou pravděpodobností se jedná o obličej.

Po klasifikaci následuje fáze post-processing. V této fázi můžeme poupravit výstup celého systému, na základě jiných znalostí (např. kontextu). Nebo v této fázi můžeme počítat chybovou míru systému. Použijeme-li více klasifikátorů, rozhodujeme se až po vzájemném sečtení jejich výstupů, které se provádí právě ve fázi post-processingu.

### 2.1.1 Shora dolů

Metody shora dolů [33], neboli také metody založené na znalostech (*knowledge-based methods*), jsou metody využívající naše vědomosti o struktuře lidského obličeje. Z našich znalostí odvozujeme pravidla pro detekci. Nejdříve v obraze hledáme kandidáty míst, kde by se mohli nacházet obličeje, a ty dále testujeme. Postupujeme z nejvyšší úrovně k detailům.

### 2.1.2 Zdola nahoru

Tento přístup je postaven na faktu, že lidé dokáží vnímat objekty při různých osvětleních a v odlišných pozicích. Výzkumníci se snažili nalézt rysy, které jsou vůči takovým podmínkám invariantní (*feature invariant approach*) [33]. Vyhledané části, například hrany dílčích částí obličeje, slučujeme a testujeme, zda tvoří celek. Dalšími vybranými rysy je například textura nebo barva kůže.

### 2.1.3 Porovnávání šablon (*Template matching*)

V metodě porovnávání šablon [33] hledáme míru shody částí obrazu s předem definovanými šablonami. Příkladem je model, který obsahuje několik dílčích šablon modelujících konturu očí, nosu, úst a tváře. Každá šablona nám definuje linii dané části. Linie ze vstupního obrazu extrahujeme na základě nejvyšší změny gradientu. K detekování možných pozic obličeje

vypočteme korelace mezi částmi obrazu a dílčími šablonami. V kandidátních pozicích poté hledáme shody s dalšími dílčími šablonami. Šablony nemusí být nutně statické, mohou být i deformovatelné. Takovým příkladem jsou šablony ASM (*active shape models*) [33].

### 2.1.4 Metody založené na vzhledu (*Appearance-based methods*)

Metody založené na vzhledu [33], na rozdíl od metod porovnávání šablon, nepoužívají předdefinované modely, ty jsou naučené. Využíváme strojového učení a statistické analýzy k nalezení chatekteristik obličejů. Celý proces je rozdělen na dvě hlavní části, trénování klasifikátoru a klasifikaci (testování částí obrazu klasifikátorem). Metody založené na vzhledu zaznamenaly v posledních deseti letech největší rozvoj díky zvětšujícímu se výpočetnímu výkonu počítačů a zvyšující se kapacitě úložných médií. Trénování klasifikátoru i klasifikace se tím staly dostupnějšími. Naše budoucí práce bude směřovat tímto směrem. Jelikož naším úkolem není popsat všechny dostupné metody, uvedeme pouze výčet z [33],[34]:

- EigenFaces,
- SVM (*Support Vector Machine*),
- Neuronové sítě,
- Bayesovské klasifikátory,
- Skryté Markovovy modely (*HMM – Hidden Markov model*),
- Boostovací schémata.

My se zaměříme na detektor, který je natrénován metodou boostování (bude osvětleno v následující kapitole).

# Kapitola 3

# Detektor Violy a Jonese

V roce 2001 Paul Viola a Michael Jones navrhli detektor obličejů pracující v reálném čase [27]. Pilíři jejich úspěšného detektoru se staly algoritmus AdaBoost, kaskáda klasifikátorů a integrální obraz. Jejich práce položila základy rozšíření detekce obličejů do rozdílných aplikací použitelných v běžném životě (v digitálních fotoaparátech či v softwarech pro správu forografií).

## 3.1 AdaBoost

Výzvou pro pány Violu a Jonese bylo z velkého množství příznaků v podoblasti[1], nalézt jen ty nejdůležitější, ze kterých by vytvořili finální efektivní klasifikátor. Vhodnou metodu spatřili v AdaBoostu. Algoritmus AdaBoost (*Adaptive Boosting*) [5] navrhli pánové Yoav Freund a Robert Schapire v roce 1996 [2].

Boosting [5], [22], [27] je obecná metoda, která kombinuje několik slabých klasifikátorů (*weak classifiers*) za účelem vytvoření jednoho silného (*strong classifier*). Vytvořit slabý klasifikátor má za úkol slabý učitel (*weak learner*). Viola a Jones si zvolili jako slabý klasifikátor jeden příznak. Úkolem slabého učitele je pak vrátit příznak, který nejlépe odlišuje pozitivní a negativní vzorky, a též pro slabý klasifikátor zvolit optimální prahovací funkci. V [27] je slabý klasifikátor *h(j)* definován vztahem 3.1,

$$h_j(x) = \begin{cases} 1, & p_j f_j(x) < p_j \theta_j \\ 0, & jinak \end{cases} \tag{3.1}$$

kde $p_j$ je parita, $f_j$ je příznak, a práh $\theta$, $x$ značí zkoumanou podoblast (např. 24 × 24px).

[1] Pro okno 24 × 24 je to 45396 příznaků [27].

[2] V roce 2003 Yoav Freund a Robert Schapire za algoritmus AdaBoost obdrželi Gödelovu cenu.

Algoritmus AdaBoostu upravený Violou a Jonesem ilustruje pseudokód .

**Algoritmus 3.1.1:** AdaBoost$((x_1, y_1), ..., (x_n, y_n))$

Vstupem jsou referenční obrazy $(x_1, y_1), ..., (x_n, y_n)$, kde $y_i \leftarrow 0{,}1$ pro negativní, respektive pozitivní vzorky.

Inicializujeme váhy: $w_{1,i} \leftarrow \frac{1}{2m}, \frac{1}{2l}$ pro $y_i \leftarrow 0{,}1$, kde $m$ a $l$ je počet negativních, resp. pozitivních vzorků.

**for** $t \leftarrow 1$ **to** $T$

**do**
- Normalizujeme váhy tak, aby $w_t$ bylo pravděpodobnostní rozložení: $w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j \leftarrow 0}^{n} w_{t,j}}$.
- Pro každý příznak $j$ natrénujeme slabý klasifikátor $h_j$, chybu klasifikátoru spočteme s ohledem na rozložení pravděpodobnosti $w_t$: $\epsilon_j \sum_i |h_j(x_i) - y_i|$.
- Vybereme takový slabý klasidikátor $h_i$, který disponuje nejmenší chybou $\epsilon_t$.
- Aktualizujeme váhy $\epsilon_t$: $w_{t+1,i} \leftarrow w_{t,i}\beta_t^{1-\epsilon}$, kde $\epsilon_i \leftarrow 0$, jestliže vzorek $x_i$ je zařazen správně, jinak je $\epsilon_i \leftarrow 1$, $\beta_t \leftarrow \frac{\epsilon_t}{1-\epsilon_t}$.

Sestavíme výsledný klasifikátor:
$h(x) \begin{cases} 1 & \sum_{t \leftarrow 1}^{T} \alpha h_t(x) \geq \frac{1}{2} \sum_{t \leftarrow 1}^{T} \alpha_t \\ 0 & jinak \end{cases}$, kde $\alpha_t \leftarrow \log \frac{1}{\beta_t}$.

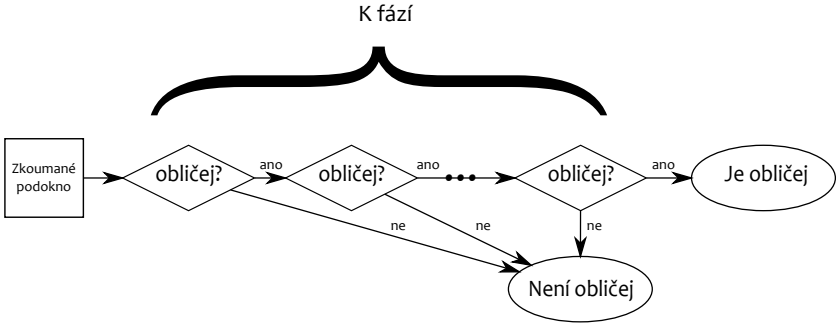
Z algoritmu 3.1.1 vidíme, že každý jeho krok trvá stejný čas, není velký časový rozdíl mezi prvním a stým krokem. Mezi další výhody patří jeho jednoduchost a minimum parametrů, které potřebuje pro různá nastavení (mění se pouze parametr T), často se nepřetrenovává. Nevýhodou algoritmu AdaBoost je jeho citlivost na šum.

AdaBoost není jediné boostovací schéma, další schéma je například RealBoost [22]. Ten se liší od Adaboostu reálným ohodnocením slabých klasifikátorů, na rozdíl od diskrétního Adaboostu (slabý klasifikátor hodnotí pouze 0 nebo 1). Od uvedení AdaBoostu vzniklo mnoho prací, které jej vylepšují v různých směrech: urychlení testování, urychlení trénování, použití výsledků z předchozích uzlů, ukončování silného klasifikátoru dříve, než jen na konci každé fáze (např. *WaldBoost*), rozcestník těchto přístupů je obsažen v [34].

## 3.2 Kaskáda klasifikátorů

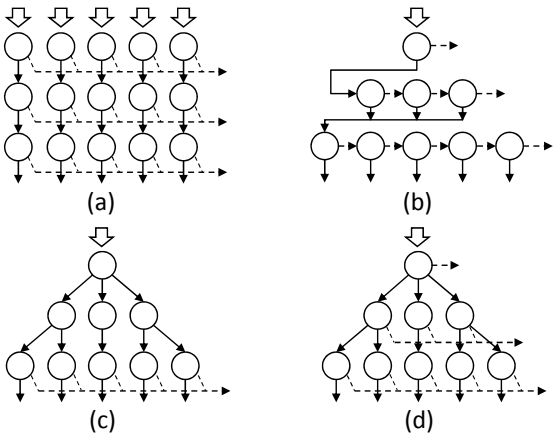
Viola a Jones formulovali kaskádu klasifikátorů [27] jako degenerovaný rozhodovací strom. Jejím účelem je urychlení výpočetního času klasifikace při zachování dobré míry detekce (80-95 procent) a nízké míry chybně akceptovaných vzorků (*false positive rates*) v řádu $10^{-5}$ nebo $10^{-6}$ [27]. Kaskáda je rozdělena na fáze, ty jsou natrénovány tak, aby minimalizovaly

počet chybně odmítnutých vzorků (*false negatives*). Každá fáze porovnává N příznaků, přičemž číslo N se postupně zvyšuje. Tímto uspořádáním kaskáda dosahuje vysoké rychlosti (většina vzorků neobsahující obličej je odmítnuta v prvních jednodušších fázích klasifikace).




Obrázek 3.1: *Princip kaskády klasifikátorů.*

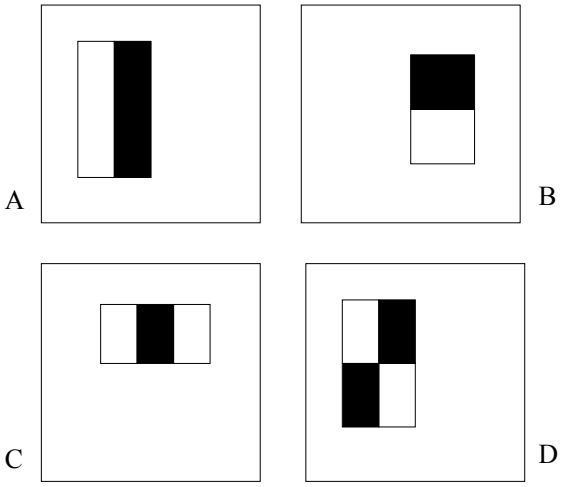
Kaskáda je navržena zejména pro frontální detekci obličeje (zepředu), selhává u detekcí, kdy je na obličej nahlíženo z více stran (*multi-view detection*). Cílem výzkumníků bylo nalézt podobnou taktiku, která podává lepší výsledky. Toho dosáhli obdobou metody "*rozděl a panuj*". Použili struktury jako paralelní kaskáda, pyramida a varianty rozhodovacích stromů. Struktury uvádíme pouze pro přehled, dále se jimi nebudeme zaobírat.




Obrázek 3.2: *a) paralelní kaskáda, b) pyramida, c,d) varianty rozhodovacího stromu, převzato z [34].*

## 3.3 Haarovy příznaky

Ukazuje se, že z hlediska rychlosti a rozlišitelnosti je výhodnější pracovat s příznaky, než přímo s pixely obrazu [27]. Navíc příznaky, na rozdíl od pixelů, dokáží zjistit nějakou zajímavou vlastnost obrazu, kterou bychom klasifikátor jen stěží učili. Viola a Jones navrhli příznaky podobné haarovým vlnkám [31]. Příznaky rozdělili do tří skupin, podle počtu oblastí nutných pro jejich kalkulaci: dvoj-obdelníkové, tří-odbdelníkové a čtyř-obdelníkové příznaky. Plocha všech obdelníků v jednom příznaku je identická.



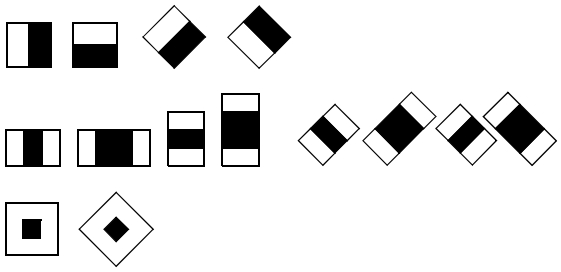

Obrázek 3.3: *Typy příznaků: A,B) dvoj-obdelníkové příznaky, C) troj-obdelníkový příznak, D) čtyř-obdelníkový příznak. Obrázek převzat z[27].*

Způsob výpočtu hodnoty příznaku ilustruje obrázek 3.4. Příznaky pracují jen s intenzitou obrazu. Haarovy příznaky reflektují základní vlastnosti lidského obličeje (například oblasti očí mají nižší intenzitu než oblasti tváří).

Obrázek 3.4: *Výpočet haarova příznaku (dvoj-obdelníkový příznak). Sumu intenzit pod bílým obdelníkem odečteme od sumy intenzit pod černým obdelníkem.*

Od doby existence haarových příznaků vzniklo mnoho jejich variací. Jednou z nich je přidání příznaků zrotovaných o 45 stupňů od pánů Lienhartda a Maydta [14]. Jejich modifikace snížila počet chybně akceptovaných vzorků průměrně o deset procent proti verzi detektoru Violy a Jonese. Počet příznaků v okně 24 × 24 se zvýšil na 117,941 – vzorec pro výpočet množství příznaků nalezneme taktéž v [14]. Toto zobecnění vyžaduje nový typ integrálního obrazu rotovaného o 45 stupňů, jehož výpočet je náročnější, tím pádem i pomalejší (jsou potřebné dva průchody obrazem zleva-doprava a shora-dolů)[14].

Další modifikací jsou například spojené haarovy příznaky (*joint haar-like features*) [34], jejichž funkce spočívá v souběžném výskytu více haarových příznaků v okně (odezva jednoho příznaku je buď 0 nebo 1, výsledek všech porovnání je binární výstup). Haarovy příznaky byly rozšířeny i pro video (*haar-like features on motion filtered video*) [34], ty jsou založeny na diferencích snímků aktuálního a různě posunutého předešlého snímku. Tím získáme směr



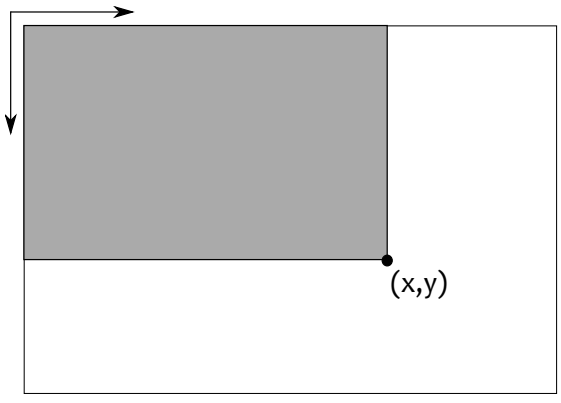
Obrázek 3.5: *Rozšířená sada příznaků o rotované příznaky. Všimněme si, že ačkoliv mnoho příznaků přibylo, byl odstraněn typ čtyř-obdelníkových příznaků. Převzato z [14].*

pohybu objektu (tento postup není invariantní vůči změně měřítka). Zajímavou modifikací je převedení 2D haarových příznaků do 3D a za jejich pomoci detekovat různé akce (např. mávání) ve videu [12].

Limitací haarovych příznaků je malá odolnost vůči různým světelným podmínkám. Příznaky, které řeší problémy se světelnými podmínkami, jsou například LBP (*Local Binary Patterns*) [34], nebo LRP (*Local Rank Patterns*), jež jsou vhodnou alternativou pro haarovy příznaky (příkladem je jejich použití v [8]). Spojení obou přístupů, haarových příznaků i LBP příznaků, nalezneme v práci [21](HLBP – *Haar Local Binary Patterns*). Dalšími typy jsou příznaky založené na statistice (*Statistics based features*) [34], ty pracují se statistickými informacemi obrazu, například histogramy. Geometrii obličeje vystihují lépe než haarovy příznaky, a navíc jsou invariantní vůči změně osvětlení. Do této skupiny spadají histogramy orientovaných hran nebo histogramy orientovaných gradientů (*HoG*) [34].

## 3.4 Integrální obraz

Integrální obraz [3], [27], je reprezentace obrazu, za pomocí které lze rychle vypočítat sumu hodnot obdelníkové podoblasti obrazu. Bod v integrálním obrazu na pozici (x,y) obsahuje sumu všech hodnot pixelů vlevo a nahoru včetně hodnoty pixelu na pozici x, y (viz obrázek 3.6). Pravý spodní pixel integrálního obrazu je tudíž sumou všech hodnot originálního obrazu.

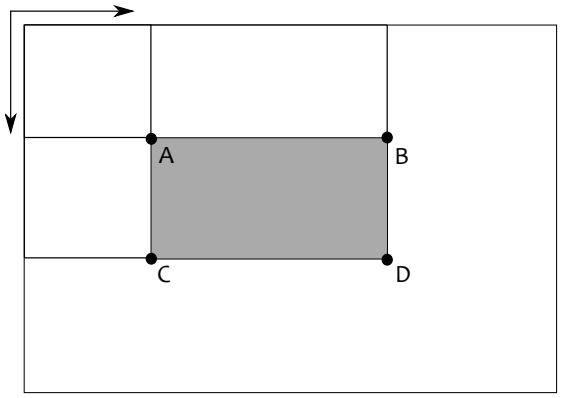



Obrázek 3.6: *Integrální obraz, bod na pozici (x,y) obsahuje sumu všech hodnot ze zvýrazněné části obrazu.*

Matematický zápis integrálního obrazu ukazuje vzorec 3.2,

$$ii(x,y) = \sum_{x' \leq x, y' \leq y} i(x', y') \tag{3.2}$$

kde *i(x,y)* představuje originální obraz a *ii(x,y)* je výsledný integrální obraz. Součet hodnot v obdelníkové podoblasti vypočítáme ze čtyř referencí v integrálním obrazu. Pozice referencí odpovídá pozici rohů obdelníku v originálním obrazu.




Obrázek 3.7: *V bodě D je uložena suma všech hodnot nalevo a nahoru, proto od ní odečteme sumu z bodu B a C. Následně přičteme sumu A, jelikož jsme tuto sumu odečítali dvakrát. Výsledný vzorec pro vyplněnou oblast je D - B - C + A.*

# Kapitola 4

# Dostupná rozhraní pro obecné výpočty na GPU

## 4.1 CUDA

CUDA (*Compute Unified Device Architecture*) [16], [28] je paralelní výpočetní architektura firmy NVIDIA. Byla uvedena v roce 2007. Do té doby se GPGPU (*General-Purpose computation on Graphics Processing Units*) výpočty prováděly různými triky, které využívali grafického vykreslovacího řetězce.

Celý výpočetní systém CUDA se skládá z hostitele (*host*), tím je tradiční centrální výpočetní jednotka (*CPU*), a z jednoho nebo více zařízení (*devices*) - paralelní grafické čipy. Model zpracování instrukcí je SIMT (*Single Instruction Multiple Threads*). Pro programování v CUDA se používá jazyk na bázi jazyka C s jistými omezeními a rozšířeními. Novější stroje podporující CUDA povolují i některé konstrukce z jazyka C++.

Program spouštěný na GPU se nazývá kernel. Před voláním kernelu je nutné alokovat grid, ten se skládá z 2D bloků vláken (*thread blocks*). Každý multiprocesor navíc rozděluje blok vláken do skupin vláken (*warp*) po 32 vláknech. Vlákna ve warpu jsou zpracovávana paralelně (každá instrukce běží ve stejný čas). Architektura CUDA určuje i hierarchii pamětí. Jednotlivé úrovně paměti se liší svojí latencí, velikostí, viditelností vzhledem k vláknu, životností a možností cache. Hierarchie pamětí zvyšuje propustnost celého systému. CUDA standardně nepodporuje operátory pro vektorové typy, jelikož se jedná o architekturu skalární (ovšem doporučuje se používat typ `float4` pro načítání dat do grafické paměti, CUDA načítá 128bitů v jednom načtení [16]).

Hlavním omezením CUDA je možnost programování výrobků pouze od firmy NVIDIA. Jinak se jedná o velmi silný a rozšířený nástroj pro GPGPU výpočty.

## 4.2 DirectCompute

DirectCompute [30] je rozhraní pro GPGPU výpočty od firmy Microsoft. DirectCompute spadá do skupiny rozhraní patřících pod DirectX (kolekce rozhraní věnovaná práci s multimédii). DirectCompute byl uveden spolu s DirectX 11, přesto je podporován s menšími limitacemi na zařízeních s DirectX 10. Pro výpočty na GPU rozhraní odhaluje nový shader nazvaný *Compute Shader* (pozn. shadery v DirectX používají syntaxi jazyka HLSL – *High Level Shader Language*). Programovací model rozděluje práci do skupin vláken (*Thread groups*), které leží v 3-dimenzionální mřížce (*Dispatch*). Obdobně jako CUDA, či OpenCL,

dovoluje programátorovi využít sdílenou (*shared*) paměť v rámci skupiny vláken. Sdílená paměť je limitována verzí Compute Shaderu (verze 4.x povoluje velikost 16KB a verze 5.0 32KB [26]). Sdílí i ostatní principy jako je například synchronizace vláken.

DirectCompute na rozdíl od CUDA není hardwarově omezen na grafické karty jednoho výrobce, jeho omezení je softwarové, jelikož je přístupný pouze pro operační systémy Windows firmy Microsoft.

## 4.3 OpenCL

OpenCL (*Open Computing Language*) [13] je otevřený standard pro paralelní programování napříč různými typy procesorů (CPU, GPU a další procesory, jakými jsou například signální procesory DSP). OpenCL spravuje konsorcium KHRONOS od listopadu 2008 [29]. Momentální verze specifikace je OpenCL 1.2 (vydána 15.11. 2011). Zahrnuje programové rozhraní a specifikaci jazyka OpenCL C. Abstraktní model OpenCL dle specifikace popisuje čtyří základní modely: model platformy (*platform model*), model paměti (*memory model*), exekuční model (*execution model*), programovací model (*programming model*).

Model platformy se skládá z hostovské aplikace (*host*) připojené k jednomu nebo více OpenCL zařízením (*OpenCL devices*). Každé zařízení je rozděleno na jednu nebo více výpočetních jednotek (*computing units*), které dále obsahují opět jeden či více procesních prvků (*processing elements*).

Exekuční model definuje spuštění kernelu. Běh kernelu je rozdělen do pracovních skupin (*work-groups*), které se skládájí z pracovních položek (*work-items*). Na rozdíl od modelu CUDA již nedefinuje, kolik pracovních položek se má zpracovat současně (to je záležitost hardwarového řešení, například u karet ATI toto číslo prezentuje *wavefront* [11], jež je roven 64, to je dvakrát více než definuje NVIDIA warp [16]).

OpenCL dělí paměti na globální (*global*), konstantní (*constant*), lokální (*local*) a privátní (*private*). Hierarchie pamětí je obdobná jako u CUDA. Terminologie CUDA nás může mást, jelikož v CUDA je lokální paměť pojmenována shared a privátní paměť užívá termín local.

Podporovaný programovací model je datově paralelní (*dat parallel*) i úlohově paralelní (*task parallel*). OpenCL se ovšem primárně zaměřuje na datově paralelní model.

Značná výhoda OpenCL je jeho softwarová a hardwarová nezávislost. Z hardwarové nevázanosti OpenCL plynou i nevýhody, zejména pro programátora, který by měl kód psát obecněji kvůli kompatibilitě s různými zařízeními.

# Kapitola 5

# Možnosti implementace na GPU

V této kapitole se zaměříme na možnosti implementace detektoru obličejů na grafické kartě. Nejdříve si osvojíme paralelní primitiva, která se nám budou hodit při psaní programu. Dozvíme se, jak paralelně počítat integrální obraz. Prozkoumáme možnosti procesu detekce. Probereme si strukturu klasifikátoru a budeme diskutovat jeho možnosti uložení do různých druhů paměti. Pro namapování detektoru na GPU si zvolíme OpenCL z důvodu hardwarové a softwarové nezávislosti. Z toho důvodu jsme v minulé kapitole podrobněji probrali standard OpenCL a jeho terminologii.

Vytvoření pyramidy obrazů je naopak část, kterou budeme implementovat na GPU nejspíše za pomoci některé z hardwarových funkcí grafické karty.

## 5.1 Paralelní primitiva

Paralelní primitiva jsou datově paralelní algoritmy, jež jsou hojně využívány při tvorbě širokého spektra složitějších algoritmů na grafických kartách. Osvětlíme si tři základní primitiva, která v detektoru budeme potřebovat.

### 5.1.1 Suma prefixů

Suma prefixů (neboli sken) je operace, která z vektoru čísel vytvoří nový vektor, jehož každý prvek je sumou předešlých prvků. Pokud máme na vstupu vektor a binární asociativní operaci $\oplus$:

$$[x_0, x_1, .., x_{n-1}] \tag{5.1}$$

aplikací sumy prefixů dostáváme nový vektor:

$$[x_0, (x_0 \oplus x_1), .., (x_0 \oplus x_1 \oplus .. \oplus x_{n-1})]. \tag{5.2}$$

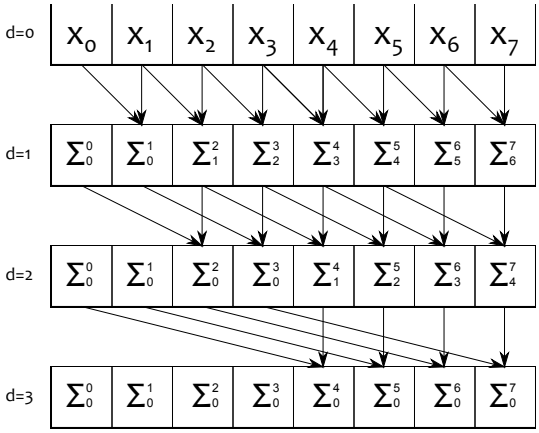
Suma prefixů je ze své podstaty sekvenční, nicméně pro ni existuje efektivní paralelní algoritmus.

**Algoritmus 5.1.1:** PREFIX SUM($x[0..N]$)

$s \leftarrow x[0]$
**for** $k \leftarrow 1$ **to** $N-1$
**do** $\begin{cases} s \leftarrow s + x[k] \\ x[k] \leftarrow s \end{cases}$
**return** $(x[0..N])$

### Paralelní algoritmus

První datově paralelní algoritmus byl popsán již v roce 1986 pány Hillisem a Steelem v [9]. Obrázek 5.1 nám dobře ilustruje vykonání algoritmu. Pro úplnost připojíme i psedokód algoritmu. Tento postup má časovou složitost $O(Nlog_2N)$, která na rozdíl od sekvenčního algoritmu není optimální.




Obrázek 5.1: *Paralelní průběh výpočtu sumy prefixů podle Hillise a Steela.*

---

**Algoritmus 5.1.2:** HILLIS-STEEL$(x[0..N])$

---

$s \leftarrow x[0]$
**for** $d \leftarrow 1$ **to** $\log_2 N$
**do** { **for each** $k$ **in parallel**
**do** { **if** $k \leq 2^d$
**then** $x[k] \leftarrow x[k - 2^d] + x[k]$
**else** $x[k] = x[k]$
**return** $(x[0..N])$

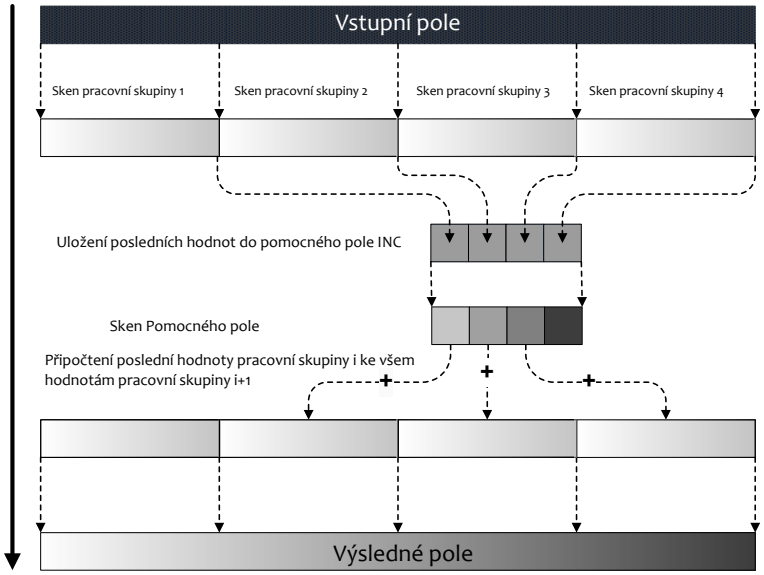
---

Efektivní algoritmus byl v roce 1990 prezentován panem Blellochem pro architekturu PRAM [2]. Hlavní myšlenka tohoto algoritmu je založena na procházení vyváženého binárního stromu. Celý proces se skládá ze dvou průchodů (*Up-sweep* a *Down-sweep*). První průchod je *Up-sweep*, neboli redukce, při kterém procházíme stromovou strukturu od listů ke kořenu a v uzlech uchováváme částečné sumy, dokud nenarazíme na kořen. Ve druhém průchodu začínáme u kořene, do kterého vložíme neutrální prvek a míříme k listům. V každém kroku pak hodnotu z každého uzlu z aktuálně zpracovávané úrovně předáme jeho levému potomku. Do pravého potomka putuje hodnota získaná operací $\oplus$ mezi původní hodnotou levého potomka a hodnotou otcovského uzlu. Složitost tohoto algoritmu je $O(N)$, takže je efektivní.

Naneštěstí efektivní algoritmus je na moderních GPU pomalejší, jak uvádí práce [23], [24]. Proto použijeme první algoritmus pánů Hillise a Steela.

### Generalizace pro velká pole hodnot

Uvedené algoritmy v praxi pracují pouze pro pole hodnot o velikosti pracovní skupiny. Pro generalizaci je nutné výsledky ze skupin sloučit dohromady. Budeme postupovat identicky, jak je popsáno v [6], s rozdílem převedení terminologii pojmů z CUDA na OpenCL. Předpokládejme, že pracujeme s binární asociativní operací sčítání.



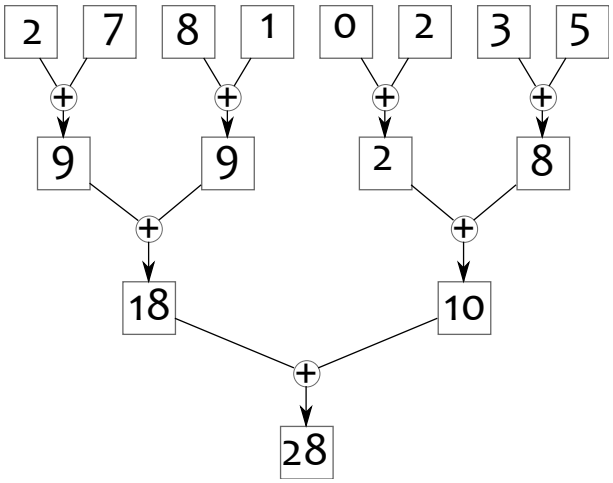

Obrázek 5.2: *Sken pole o velikosti čtyř pracovních skupin. Převzato z [6] – upravena terminologie.*

Rozdělíme si pole hodnot na úseky o velikosti pracovní skupiny. Na nich provedeme scan. Poslední hodnota scanu udává sumu všech hodnot v pracovní skupině, proto si ji uchováme v dodatečném poli sum pracovních skupin *SUM*. Na pole *SUM* znovu aplikujeme scan a výsledek uložíme do dalšího pole *INC*. Přírustky pole v *INC* sečteme s hodnotami v pracovní skupině (v nichž se nachází částečné sumy prefixů), čímž dostaneme výsledek. Může se stát, že počet polí v SUM bude větší než velikost pracovní skupiny. Tuto situaci vyřeší rekurzivní zavolání scanu z hostovské aplikace jak je navrhováno v [23], [24], nebo použijeme iterativně jednu pracovní skupinu na zpracování zbylých hodnot.

### 5.1.2 Redukce

Redukce je paralelní primitivum, jehož úkolem je zredukovat vektor hodnot na jednu hodnotu (například výpočet sumy vektoru). Opět na ni nahlížíme jako na průchod binárním stromem, myšlenka tohoto průchodu je shodná s *Up-sweep* průchodem, kde ji Blelloch využil. Abychom si udělali lepší představu, můžeme si prohlédnout obrázek.

Stejně jako u sumy prefixů algoritmus funguje na GPU pouze pro pole hodnot menší než nebo rovno velikosti pracovní skupiny. Jeho zobecnění je obdobné jako u sumy prefixů, kdy se nejprve redukují bloky pracovních skupin a následně jejich výsledky.




Obrázek 5.3: *Princip redukce. (Pozn.: Redukce je obecné paralelní primitivum, které lze využít kromě sumy hodnot pole i na zjištění minima, nebo maxima).*

### 5.1.3 Stream compaction

Účelem primitiva *Stream compaction* je vybrat z řídkého vektoru jen ty důležité hodnoty a tím velikost vektoru zmenšit. Práce s menším vektorem je efektivnější, a menší vektor navíc šetří čas při paměťových přenosech. Nám se primitivum bude hodit při předávání výsledků (pozic obličejů) hostovské aplikaci.

Paralelní algoritmus *Stream compaction* se skládá ze tří kroků:

1. Vytvoříme dočasný vektor, který bude maskou původního vektoru. Pokud je prvek vektoru objektem našeho zájmu, nastavíme v masce položku na 1, v opačném případě na 0.

2. Aplikujeme primitivum sumy prefixů na dočasný vektor. Pomocný vektor bude nyní obsahovat indexy pro hodnoty, které jsou v našem zájmu.

3. Přesuneme hodnoty, které jsou indikovány maskou, na pozici obsaženou v pomocném vektoru do výsledného vektoru.

Tyto kroky jsou textovým popisem pseudokódu, který nám pomůže detailně objasnit ne-

srovnalosti.

**Algoritmus 5.1.3:** STREAM COMPACTION($x[0..N]$)

**comment:** Krok 1.

**for each** $i \in [0..N]$ **in parallel**
**do** { **if** ISINTERESTING($x[i]$)
  **then** $mask[i] \leftarrow 1$
  **else** $mask[i] \leftarrow 0$

**comment:** Krok 2.

$index[0..N] \leftarrow$ PREFIXSUM($mask[0..N]$)

**comment:** Krok 3.

**for each** $i \in [0..N]$ **in parallel**
**do** { $a \leftarrow index[i]$
**if** $mask[i]$
  **then** $result[a] \leftarrow x[i]$
**return** ($result[0..N]$)

## 5.2 Paralelizace výpočtu integrálního obrazu

Algoritmus 5 představuje pseudokód sekvenčního výpočtu integrálního obrazu. Jeho složitost je O(N2). V této podkapitole se dozvíme možnosti jeho implementace na grafické kartě. Použijeme přitom již nabyté znalosti o algoritmu prefixů sum.

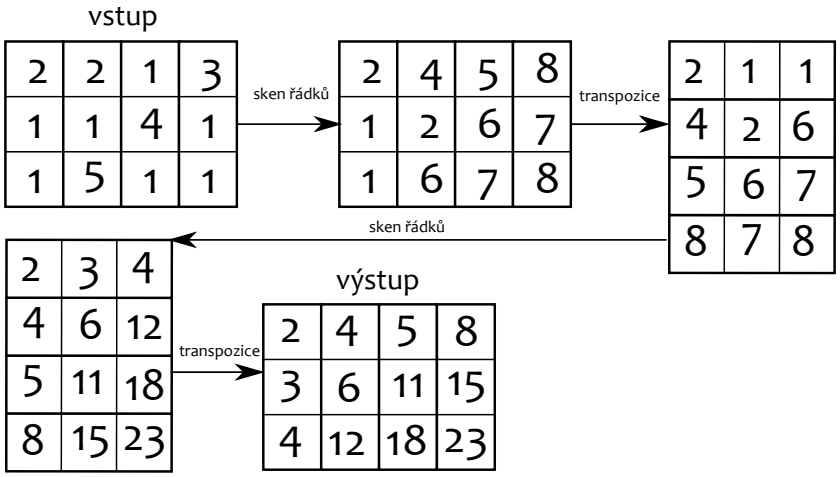
**Algoritmus 5.2.1:** CREATE INTEGRAL IMAGE($img[], w, h$)

$s \leftarrow x[0]$
**for** $y \leftarrow 0$ **to** $h - 1$
**do** { $s \leftarrow 0$
**for** $x \leftarrow 0$ **to** $w - 1$
  **do** { $s \leftarrow s + img[x + y \cdot w]$
  **if** $y = 0$
    **then** $i\_img[x] \leftarrow s$
    **else** $i\_img[x + y \cdot w] = i\_img[x + (y - 1) \cdot w] + s$
**return** ($i\_img[]$)

### 5.2.1 Výpočet integrálního obrazu pomocí prefixů sum

Hojně využívaná metoda pro výpočet integrálního obrazu na grafických čipech je metoda založená na sumě prefixů [1], [6], [10], [15], [20], [25]. Nejprve rozdělíme obraz na řádky a vytvoříme jejich skeny, druhá operace je sken po sloupcích, čímž vznikne žádaný integrální obraz.

Nevýhodou tohoto přístupu je sken po sloupcích, protože přistupuje k hodnotám v paměti, které jsou od sebe poměrně vzdálené. Řešením je po prvním skenu paralelně transponovat obraz a znovu provést sken po řádcích [1], [6], [10], [20]. Abychom dostali odpovídající integrální obraz, aplikujeme další operaci transpozice.




Obrázek 5.4: *Posloupnost akcí pro paralelní výpočet integrálního obrazu. Pro všechny akce existuje paralelní algoritmus.*

## 5.3 Možnosti uložení kaskády klasifikátorů v paměti GPU

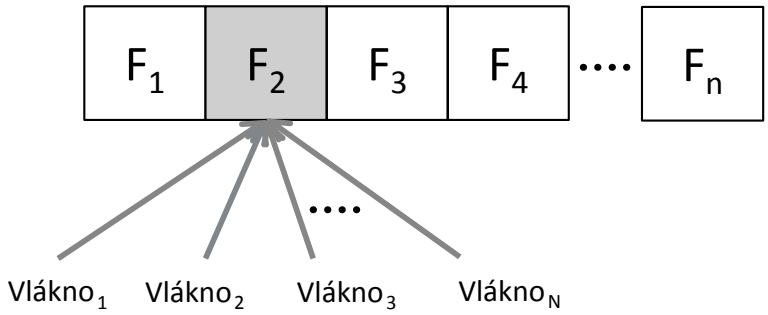
Grafické čipy obsahují hierarchii pamětí. Z toho nám vyplývají i možnosti uložení klasifikátoru:

- **Globální paměť** - práce [7] ji používá pro uložení kaskády klasifikátorů. U starších grafických karet neobsahuje žádnou vyrovnávací pamět, tudíž má vysokou latenci. Z toho důvodu práce [7] přesouvá aktuální fáze kaskády do lokální paměti. Tato situace se ovšem mění s nástupem novějších grafických karet (Nvidia od architektury Fermi), u kterých již vyrovnávací paměť globální paměti je přítomna [17].

- **Texturovací paměť** - je umístěna v globální paměti, přístup k ní je optimalizován vyrovnávací pamětí (i u starších grafických karet). Je optimalizována pro plošné přístupy, čímž se liší od globální paměti (ta dosahuje největšího výkonu při sekvenčních přístupech). Je použita například v práci [25], která též používá pro zvýšení výkonu lokální paměť obdobně jako v [7].

- **Konstantní pamět** - podle specifikace OpenCL je její minimální kapacita 64KB (viz [13]). Obsahuje vyrovnávací pamět, a pro uložení kaskády klasifikátorů by mohla být výhodná, jelikož je navržena ke sdílení hodnot mezi pracovními položkami. Protože je její kapacita omezena, museli bychom se pokusit komprimovat vstupní hodnoty. Konstantní paměť je použita v práci [20]. Tento přístup se nám zdá výhodný, zejména kvůli hardwarové optimalizaci přenosu hodnoty v konstantní paměti více vláknům souběžně (viz obrázek 7.3). Klasifikátor totiž vyhodnocuje na každém podokně příznaky ve stejném pořadí.

- **Lokální paměť** - specifikace OpenCL určuje 32KB (viz [13]) jako její minimální kapacitu. Nemůžeme ji proto použít jako úložiště pro celý klasifikátor, na druhou stranu je možné ji využít jako vyrovnávací paměť pro jednotlivé fáze klasifikátoru.




Obrázek 5.5: *Konstantní paměť má hardwarově optimalizovaný vícenásodný přenos z paměti, dopsat převzato z [20].*

### 5.3.1 Struktura OpenCV klasifikátoru

Abychom si udělali představu, jak pracuje detekce a jak přesně vypadá struktura souboru klasifikátoru. Prostudujeme si takový soubor na příkladu. Soubor klasifikátoru OpenCV [32] je ve formátu XML, na disku zabíra poměrně mnoho prostoru (klasifikátor natrénovaný na okno 24 ×24 má velikost 1.23MB, a pro okno 20 × 20 má velikost 898kB). Jeho velikost plyne již z formátu XML, který je koncipován spíše pro lidskou čitelnost, než pro strojové využití.

Výpis 5.1: *XML zdroj klasifikátoru*

```
<opencv_storage>
<haarcascade_frontalface_alt2 type_id="opencv-haar-classifier">
    <size>20 20</size>
    <stages>
        <!-- stage 0 -->
        <trees>
            <!-- tree 0 -->
            <!-- root node -->
            <feature>
                <rects>
                    <_>2 7 16 4 -1.</_>
                    <_>2 9 16 2 2.</_>
                </rects>
                <tilted>0</tilted>
            </feature>
            <threshold>4.3272329494357109e-003</threshold>
            <left_val>0.0383819006383419</left_val>
            <right_node>1</right_node>

            <!-- node 1 -->
            <feature>
                <rects>
                    <_>8 4 3 14 -1.</_>
```

```
                    <_>8 11 3 7 2.</_>
                </rects>
                <tilted>0</tilted>
            </feature>
            <threshold>0.0130761601030827</threshold>
            <left_val>0.8965256810188294</left_val>
            <right_val>0.2629314064979553</right_val></_></_>
        ...
    <stage_threshold>0.3506923019886017</stage_threshold>
    <parent>-1</parent>
    <next>-1</next>

    <!-- stage 1 -->
       ...
    <!- stage 19 -->
       ...
    <stage_threshold>53.7555694580078130</stage_threshold>
    <parent>18</parent>
    <next>-1</next>
  </stages>
</haarcascade_frontalface_alt2>
</opencv_storage>
```

Ze zdrojového kódu klasifikátoru vidíme jeho rozdělení do fází (*stages*). Uvnitř každé fáze je množina stromů (*trees*), ty se skládají z příznaků (*features*). Značka size nám určuje velikost podokna, na které je klasifikátor natrénován. Hodnota tilted u příznaku značí rotovaný příznak.

Strom příznaků je binární rozhodovací strom s jednou výstupní hodnotou. Vyhodnocení příznaku počítáme na základě sumy hodnot intenzit pixelů pod obdelníky (*rects*), jež jsou násobeny svými váhami (tj. poslední hodnota na řádku struktury obdelníku, první dvě hodnoty udávají pozici obdelníku v podoblasti, další dvě rozměr výšky a šířky). Hodnotu příznaku porovnáme s prahem (*threshold*). Pokud je hodnota příznaku menší než prahová hodnota, zvolíme levou větev stromu, jinak zvolíme větev pravou. Musíme rozlišit, zda je levá či pravá větev uzlem, pokud není, jejich hodnoty jsou výstupní (*left_val*, *right_val*). Naopak, pokud se jedná o uzly(označení *left_node*, *right_node*), pokračujeme v nich rekurzivním vyhodnocením jejich podstromů.

Výsledky rozhodovacích stromů sčítáme a ke konci fáze porovnáme s prahem fáze (*stage_threshold*). Pokud je suma z fáze větší nebo rovna prahu, pokračujeme klasifikací další fáze, v opačném případě končíme svoji činnost neúspěchem.

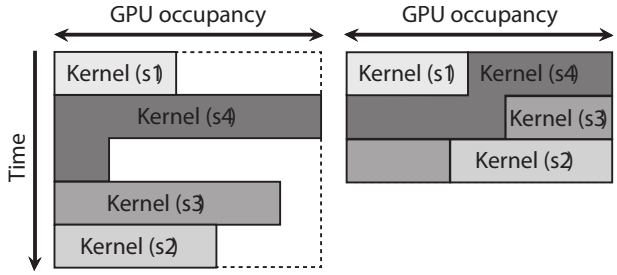
Pro implementaci se dále omezíme jen na klasifikátory typu *stump*. Jejichž stromy jsou jednoduché (kořen se dvěma listy). V kořenu obsahují práh, tím určí, který ze dvou listů s hodnotami vybrat. Toto omezení vykazuje efektivnější namapování na architekturu GPU, jelikož tak dochází k menší divergenci výpočetních vláken.

## 5.4 Možnosti namapování procesu detekce na GPU

Vyhodnocení kaskády klasifikátorů je separovatelný proces (výsledky mezi podoblastmi obrazu na sobě víceméně nezávisí), a proto jej lze jednoduše paralelizovat. Práce [7] uvádí tři základní přístupy:

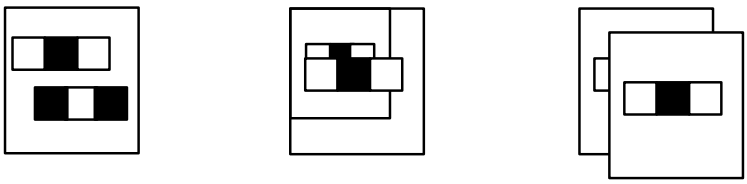
- **souběžný výpočet všech příznaků v jedné podoblasti** - vlákna paralelně počítají příznaky uvnitř podoblasti. Nevýhoda přístupu spočívá ve zbytečném počítání příznaků, ke kterým se klasifikace nemusí dostat. Částečné řešení přináší [19], kde se navrhuje paralelně počítat příznaky jen po skupinách, a vyhodnocovat průběžně po fázích.

- **souběžný výpočet jednoho příznaku v rozdílných měřítkách** - zde se nevýhoda metody skrývá v nevyváženém přidělení práce mezi rozdílnými měřítky oken (větších podoblastí je v obraze o dané dimenzi zákonitě méně, tudíž potřebují méně času na celkový výpočet). Jedním řešením je vytvořit pyramidu škálovaných obrazů, kterou uložíme do jedné textury (textura tak není optimálně zaplněna, vznikají neobsazená místa). Celý obraz pak testujeme jednotnou velikostí podokna. Další možností je spouštění kernelů konkurentně, které moderní grafické karty podporují [17]. Tímto přístupem dosáhneme rovnoměrného obsazení prostředků grafické karty.




Obrázek 5.6: *Konkurentní vyhodnocení kernelů, převzato z [20].*

- **souběžný výpočet podoblastí** - každé podoblasti přidělíme vlákno, výpočet tím pádem probíhá paralelně na více podoblastech. Přístup selhává zejména na podoblastech bez obličeje, které jsou brzy odmítnuty (hned v první fázi je odmítnuto nejvíce podoblastí). Vlákna jsou pak zbytečně blokovaná a dochází k hladovění.

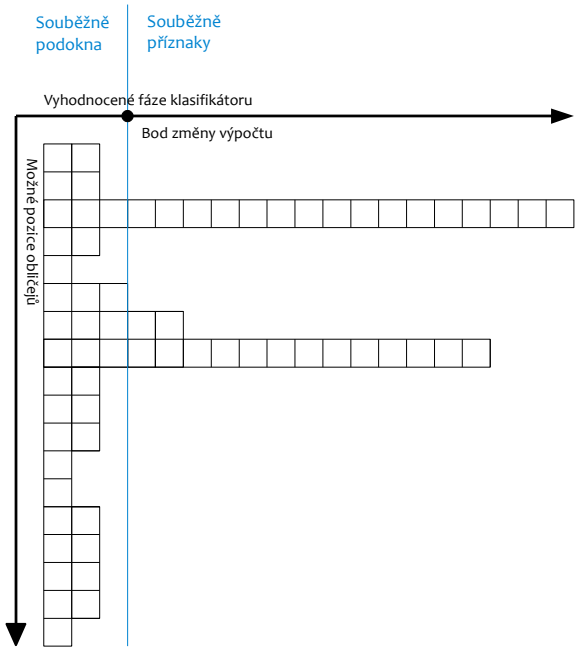


Obrázek 5.7: *Různé možnosti paralelního vyhodnocení klasifikátoru. Vlevo – souběžný výpočet příznaků v podoblasti, uprostřed – souběžný výpočet v rozdílných měřítkách, vpravo – souběžný výpočet podoblastí. Převzato z [7].*

Problém posledního přístupu částečně řeší A. Obukhov [18]. Navrhuje rozdělit silný klasifikátor na úseky po fázích a na nich spouštět kernely. Po každém shluku fází jsou místa potencionálních kandidátů vrácena zpět a je na nich volán další shluk fází klasifikátoru. Z tohoto přístupu vyplývá i častější přenos dat. Abychom snížili objem dat, můžeme použít algoritmus *Stream compaction* [6], jehož nedílnou součástí je algoritmus sumy prefixů.

Obrázek 5.8: *Hladovění vláken po první fázi klasifikátoru (bílé tečky zobrazují levou vrchní souřadnici potencionálního obličeje). Převzato z [18]. Nepodařilo se nám zjistit, ke kterému klasifikátoru tato data náleží. Nicméně situaci hladovění ilustruje dostatečně.*

Je zřejmé, že první fáze klasifikátoru je nutné počítat pro všechny nebo pro většinu pixelů v obraze. Proto je pro ně výhodné použít přístup souběžného výpočtu podoblastí s kombinací uložení klasifikátoru v konstantní paměti (stejné přiznaky jsou počítány současně - využijeme vestavěné možnosti distribuce hodnot více vláknům současně). Po dalších fázích klasifikace již dochází k úbytku potencionálních podoken, jak lze vidět na obrázku 5.8. Zde pak můžeme zapojit souběžný výpočet příznaků v jedné podoblasti, a tím lépe využít všechny prostředky grafické karty. Nastává otázka, kdy je ten nejvhodnější okamžik přepnutí mezi těmito přístupy. Toto dilema nám ilustruje obrázek 5.9.

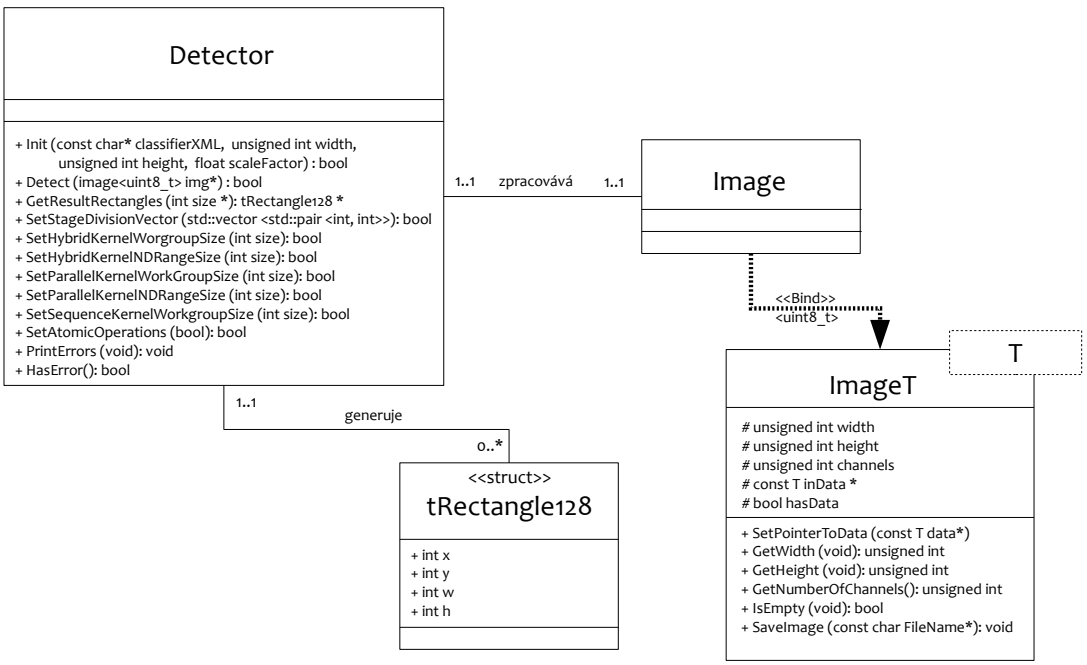



Obrázek 5.9: *Princip rozdělení výpočtu při klasifikaci.*

# Kapitola 6

# Návrh rozhraní detektoru

Navrhli jsme rozhraní knihovny. Hlavní třídou navrženého rozhraní je třída *Detector*. Ta předpokládá volání detekce ve smyčce po snímcích. Můžeme ji ovšem použít i na statické obrázky. Třída obsahuje základní metody pro nastavení detektoru. Inicializace je jednoduchá, požaduje se velikost vstupního obrazu, na kterém se bude detekce provádět, dále umístění xml souboru s klasifikátorem, a v poslední řadě škálovací faktor, udávající podíl mezi dvěma velikostmi podoken (1.0+). Vstupem detektoru je objekt šablonové třídy typu `uint_8t`. Šablonu jsme zvolili z důvodu, protože detektor bude během výpočtu pracovat i s jinými typy obrazu (např. na integrální obraz). Výstupem detektoru bude pole obdelníků. Jeden obdelník reprezentuje jednu kladnou odezvu klasifikátoru. Třída detektor bude dále obsahovat různé metody pro nastavení GPU zařízení, čímž budeme moci dosáhnout optimálního využití dané grafické karty. Navržené rozhraní si můžeme prohlédnout v uml diagramu tříd (viz 6.1).




Obrázek 6.1: *Navržené rozhraní knihovny.*

# Kapitola 7

# Implementace detektoru

V této kapitole se dozvíme podrobnosti implementace naší knihovny pro detekci obličejů na GPU. Projdeme si základní stavbu knihovny, podrobnosti výpočtu integrálního obrazu, struktury uložení kaskády klasifikátorů, detaily navržených kernelů pro detekci, zobecnění detekce ve více měřítkách, a v neposlední řadě i algoritmy a struktury, jež slouží pro předávání výsledků.

K implementaci hostovské aplikace jsme si vybrali jazyk C/C++ a pro programování grafického čipu jazyk OpenCL C (jak již bylo avizováno v předchozích kapitolách). Na volání funkcí z hostovské aplikace na zařízení jsme použili navázání knihovny OpenCL pro jazyk C++ [1]. K vývoji aplikace jsme použili Microsoft Visual Studio 2010.

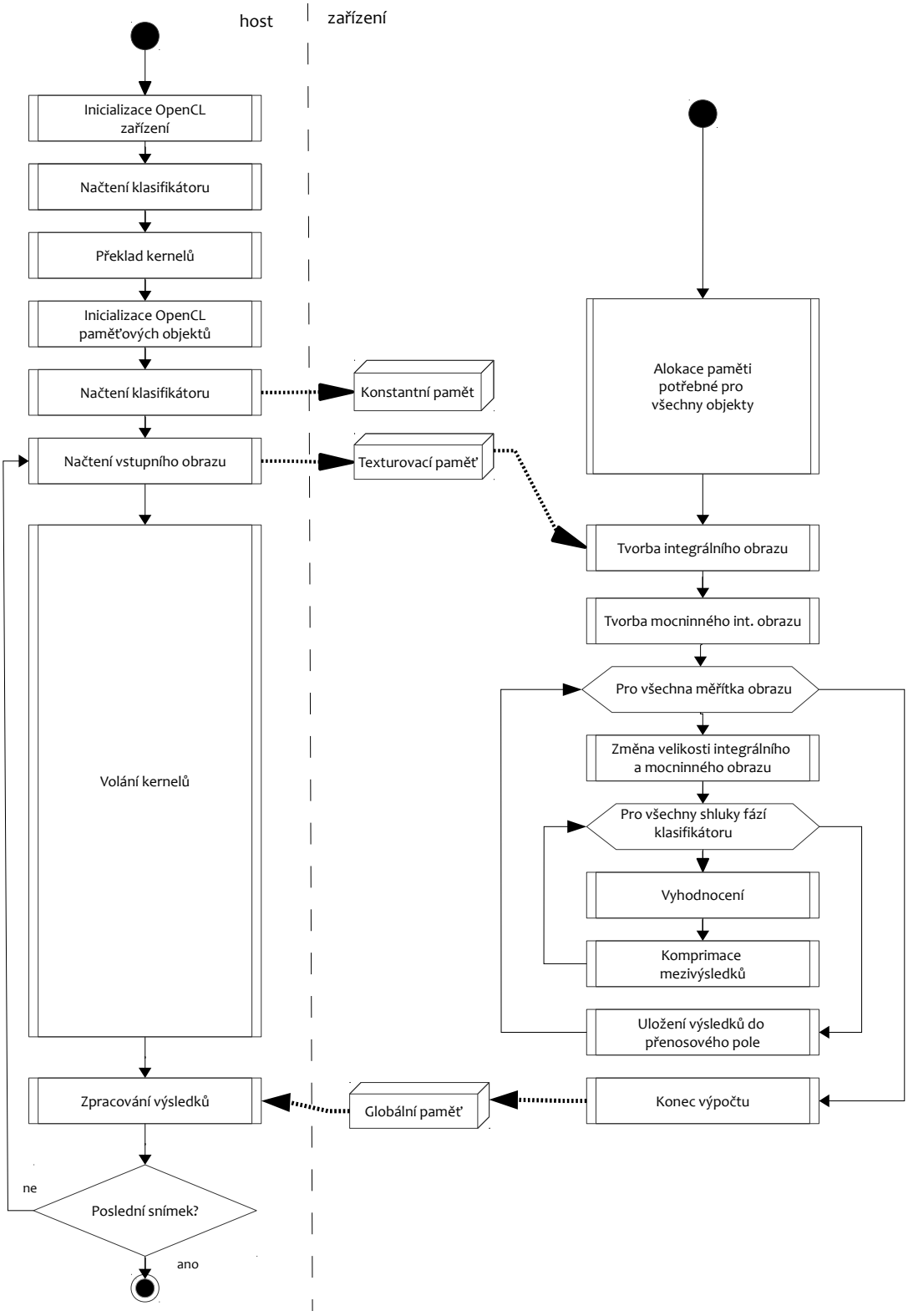
## 7.1 Stavba knihovny

Celou knihovnu jsme implementovali několika třídami, každou z nich lze zařadit do jedné z těchto skupin:

- třídy pro práci s OpenCL,
- třída detekce,
- pomocné třídy.

Program potřebuje udržovat kontext a příkazovou frontu se zařízením, to nám zajišťuje třída `CLCore`. Ta v metodě `InitCLCore` vybere vhodné zařízení, vytvoří kontext a k němu přiřadí příkazovou frontu (*Command Queue*), pomocí níž se pak komunikuje se zařízením. Fronta je nastavena na zpracování příkazů popořadě (*In-Order Execution*). Tato třída dále obsahuje metody pro vytvoření výsledného programu z jednotlivých kernelů a jeho kompilaci. Další třídou je `CLKernel`. Objekty této třídy spravují objekty podprogramů, které pak jsou z hostitelského zařízení volány, aby se vykonaly. Každý kernel má svůj zdrojový kód v odděleném souboru a při své inicializaci se registruje třídě `CLCore`, ta jejich zdrojové soubory spojuje dohromady v jeden řetězec (tzn. záleží na pořadí kernelů jak jsou nahrávány). Metodou `MakeAndBuildAllKernels` zdojové soubory zkompiluje. Dále knihovna obsahuje dvě třídy pro práci s paměťovými objekty. Jelikož paměťové objekty OpenCL mohou být různého typu (`uint8_t`, `uint32_t`, `float`), navrhli jsme pro ně šablony tříd. Jedna obaluje třídu `cl::Image2D` a druhá třídu `cl::Buffer`. Tato modifikace nám usnadňuje práci a přehlednost programu. Ukázku použití inicializace objektu `Image` vidíme níže.

[1] Dostupné ze stránek konsorcia Khronos spravujícího specifikaci OpenCL na adrese http://www.khronos.org/registry/cl/




Obrázek 7.1: *Tok příkazů knihovny.*

Výpis 7.1: *Inicializace objektu textury*

```
1 CLImage<uint8_t> * image = new CLImage<uint8_t> (w, h, channels);
2 image->Init (CL_MEM_READ_WRITE)
```

Nejrozsáhlejší a nejdůležitější třída knihovny je `gpuDetector`, která poskytuje volání všech kernelů detekce na grafickém zařízení. Pro testování a hledání chyb v ranných fázích vývoje jsme vytvořili třídu `cpuDetector`, která obsahuje ekvivalentní neparalelní kód.

Objekt je validní až po jeho inicializaci. Formát textury je určen typem objektu a parametrem `channels`, podle kterého se vybere jeden z ekvivalentních typů OpenCL (`CL_R`, `CL_RG`, `CL_RGB` a implicitní `CL_RGBA`). Pro odeslání dat do textury nám slouží metoda `ToDevice (cl_bool)`, její parametr určuje buď blokující zápis, nebo zápis neblokující. Komplementární metodou je `FromDevice (cl_bool)`. Obdobně můžeme pracovat i se třídou `CLBuffer`.

V poslední řadě aplikace obsahuje pomocné třídy. Třída `Utils` obsahuje pomocné funkce (například načtení souboru), které jsou využívány v různých částech celé knihovny. Další pomocnou třídou je `cvParser`, ta slouží pro naplnění objektu třídy `HaarClassifier` daty ze vstupního xml souboru obsahujícího klasifikátor. Správu chyb (generovaných OpenCL zařízením, nebo knihovnou) obstarává třída `Errors`.

## 7.2 Integrální obraz

Pro výpočet integrálního obrazu potřebujeme vstupní obraz reprezentovaný ve stupních šedi (bude zapotřebí jednokanálové textury s rozsahem hodnot 0-256). Výstupem bude jednokanálový obraz s vyšším rozsahem hodnot, který vyplývá z definice integrálního obrazu jako sumy všech pixelů nahorů a doleva. Aby nedocházelo k přetečení, jeví se pro HD obraz jako dostatečné použít 32bitů na jeden pixel. Přetečení nastane nejdříve po 16 777 216 pixelech, což odpovídá obrazu o rozměru $4096 \times 4096$.

O výpočet se starají celkem čtyři kernely (`scan_rows_step1`, `scan_rows_step2`, `scan_rows_step3`, `transpose`). Úlohou prvních tří je provést scan po řádkách. Ten je implementován ve třech krocích. Poprvé jsme jej implementovali v jednom kroku (kernelu) tak, že každá pracovní skupina měla za úkol provést paralelní sken jednoho řádku. Celkově bylo naplánováno N pracovních skupin, kde N se rovnalo výšce obrazu. Tento postup se neosvědčil, jelikož byl pomalý a neefektivní. Rozdělili jsme původní kernel do tří kroků (obdobně je koncipován sken v NVidia SDK). První kernel je naplánován na dvou-dimenzionální mřížce odpovídající rozměrům vstupního obrazu. Každá pracovní skupina provede paralelní sken bloku (velikost bloku je rovna velikosti pracovní skupiny, ta je jednorozměrná) hodnot v určitém řádku. Druhý kernel provede sken nad posledními hodnotami v bloku a uchová jej v dodatečném poli. Třetí kernel spojí výsledky z kernelu druhého a třetího, čímž vznikne výsledek. Poté je volána transpozice, po ní znovu sken řádků a další transpozice, kterou dostaneme očekávaný integrální obraz.

### 7.2.1 Mocninný integrální obraz

Před evaluací příznaku bychom jej měli vynásobit standardní odchylkou (tak je detektor natrénován). Standardní odchylka nám udává, jak se liší pixely v integrálním obraze. Nízkou standardní odchylku mají podoblasti s celistvou intenzitou. Výpočet standardní odchylky

podokna se počítá podle vzorce 7.1

$$\sigma = \sqrt{\left(\frac{1}{N}\sum_{i=1}^{N} x_i^2\right) - \overline{x}^2} \tag{7.1}$$

Potřebujeme spočítat průměr hodnot podokna a průměr druhých mocnin hodnot. Sumu podokna spočteme v konstantním čase stejným způsobem jako haarovy příznaky. Abychom nemuseli počítat opakovaně mocniny z integrálního obrazu, předpočítáme si je do nové textury (v naší aplikaci kernel `squared_image`). Nevystačíme si již s typem `uint32_t` na pixel (je potřeba 64b bezznaménkové číslo). Můžeme použít typ `float`, ovšem tím se nám sníží přesnost. Nebo můžeme vytvořit dvou-kanálovou texturu (s parametry `CL_RG`, `CL_UNSIGNED_INT32`) a 64-bitové číslo tím emulovat. Vyzkoušeli jsme obě možné cesty a zůstali jsme u druhé, protože by měla být přesnější. Pro kernely, které vyhodnocují jedno podokno více vlákny, byla vytvořena nová tabulka. Ta předpočítávala standardní odchylky pro všechna okna. Její význam však nebyl dostatečný.

## 7.3 Reprezentace kaskády klasifikátoru

Pro uložení kaskády klasifikátoru jsme vybrali konstantní pamět. Naše volba byla podmíněna její možností HW vestavěné distribuce hodnot více vláknům v jednom okamžiku. Konstantní paměť je omezena specifikací na minimální velikost 64KB. Museli jsme navrhnout struktury tak, abychom nepřekročili daný limit. Při návrhu je dobré brát ohled na velikost sruktury, aby data na grafické kartě byla dobře zarovnána, měly by mít velikost 4, 8 nebo 32 bajtů. Vytvořili jsme 3 struktury.

Výpis 7.2: *Struktura klasifikátoru*

```
1  // struktura obdelníkového příznaku
2  typedef struct {
3      uchar x;         // x-ová počáteční souřadnice v podokně
4      uchar y;         // y-ová počáteční souřadnice v podokně
5      uchar w;         // šířka
6      uchar h;         // výška
7      short weight;    // váha
8  } t_haar_rectangle;
9
10 // struktura příznaku
11 typedef struct {
12     float threshold;          // práh příznaku
13     float left_val;           // levá hodnota příznaku
14     float right_val;          // pravá hodnota příznaku
15     ushort n_haar_rects;      // počet obdelníkových příznaků
16     ushort haar_rect_offset;  // adresa prvního obdelníkového příznaku
17 } t_haar_feature;
18
19 // struktura fáze klasifikátoru
20 typedef struct {
21     float threshold;          // práh jedné fáze klasifikátoru
22     ushort n_haar_features;   // počet příznaků fáze
23     ushort haar_feature_offset; // adresa prvního příznaku
24 } t_stage;
```

Pozorný čtenář si všimne, že první struktura není zarovnána ani na jednu z doporučených velikostí. Ideální by bylo, kdybychom váhu uchovávali v typu `float`, jak je uváděno ve specifikaci OpenCV. Naneštěstí by struktura celého klasifikátoru přesáhla stanovený limit konstantní paměti. Typ `short` jsme zvolili, jelikož hodnoty vah v souboru použitého klasifikátoru jsou celočíselné a nepřesahují rozsah (-3,3). Pro větší komprimaci a zarovnání struktury na čtyři bajty jsme vyzkoušeli obdelník zakódovat zapomocí bitových operací (na jednu souřadnici nám postačí pět bitů, pro podokna do velikosti 31 × 31). Tento postup vedl ke zpomalení detekce, zejména kvůli režiji na dekódování. Též jsme vyzkoušeli zakódovat obdelník podle [19], kde navrhují zkomprimovat dvojici (počáteční pozice x, šířka) do osmi bitů. V tomto postupu se vychází z faktu, že souřadnice jsou k sobě komplementární – jejich součet je menší nebo roven celkové šířce podokna. Tento způsob také vedl ke zpomalení detekce.

Pro kernely které počítají více různých příznaků najednou, není výhodné konstantní paměť použít, protože nedochází k přístupu více vláken k jednomu bloku v paměti naráz. Přístupy jsou potom serializovány. Od architektury Fermi je globální paměť kešována, takže pro nenáhodné přístupy vykazuje lepší latenci. Proto jsme ji přidružili k těmto paralelním kernelům.

## 7.4 Proces detekce

Část detekce (vyhodnocení klasifikátoru) je časově nejnáročnější částí. V kapitole 5 jsme prostudovali možné přístupy. V této podkapitole si probereme detaily implementace tří různých přístupů.

### 7.4.1 Sekvenční

První přístup, který jsme implementovali, je souběžný výpočet podoken, kdy každému podoknu je přiděleno jedno vlákno. I když se jedná o paralelní přístup, dále jej budeme v práci označovat štítkem *Sekvenční*. Vyhodnocení klasifikátoru je implementováno kernelem `detection`. Každé položka pracovní skupiny se řídí pseudokódem 7.4.1 (v pseudokódu je zahrnuto pouze vyhodnocení podokna, tzn. bez načtení pozice a uložení výsledku).

**Algoritmus 7.4.1:** DETECTION($pos, stages, features, rectangles$)

```
std_dev ← COMPUTESTDDEVIATION(pos)
for each stage ∈ stages                                          (i)
do { stage_sum ← 0
     for each feature ∈ stage                                    (ii)
     do { feature_sum ← 0
          for each rect ∈ feature                                 (iii)
            do feature_sum+ = EVALRECT(pos, rect)
          if feature_sum < feature.threshold · std_dev
            then stage_sum+ = feature.left_val
            else stage_sum+ = feature.right_val
     if stage_sum < stage.threshold
       then return ( false )
return ( true )
```

Nejvnitřnější smyčku (iii) algoritmu 7.4.1, to jest vyhodnocení jednotlivých obdelníků v příznaku, jsme rozepsali (*odrolovali*). To jsme si mohli dovolit, jelikož příznak obsahuje dva až tři obdelníky. Smyčku vyhodnocení příznaků (ii) `for` jsme nahradili smyčkou `while`, čímž jsme snížili počet registrů potřebný pro výpočet, a tím zvýšili rychlost algoritmu. Nevýhodou tohoto přístupu je vysoká divergence vláken.

### 7.4.2 Hybridní

Další přístup jsme nazvali *Hybridní*. Tento kernel rozdělí pracovní skupinu na podskupiny po 32 pracovních položkách (velikost warpu). Každá takto rozdělená podskupina pak zpracovává jedno potencionální podokno. Pracovní položky z podskupiny poté paralelně počítají příznaky spadající do jedné fáze. Tím pádem nám odpadne smyčka (ii) ze *sekvenčního* kernelu a je nahrazena paralelní smyčkou. Takže pokud poslední fáze klasifikátoru obsahuje 216 příznaků, skupina 32 pracovních položek v sedmi cyklech spočte všechny dílčí hodnoty příznaků fáze. Spočtené příznaky ukládáme do lokálního pole, abychom na nich mohli provést redukci (paralelní sečtení všech příznaků z lokálního pole). Výsledek redukce je suma všech spočtených příznaků (též je uložena v lokálním poli), která se porovnává s prahem fáze. Pokud je výsledek porovnání kladný, stejným způsobem se počítá další fáze. V opačném případě je celý warp ukončen, a může počítat další potencionální pozici, pokud nějaká zbývá.

Velikost 32 pracovních položek jedné podskupiny jsme zvolili záměrně. Vlákna pak nemusíme v celé skupině synchronizovat, a též tím nedochází k divergenci pracovních položek uvnitř warpu. Celá pracovní skupina o velikosti N tudíž počítá paralelně N / 32 podoken současně. Jelikož každému vláknu je přidělen jiný příznak, přesunuli jsme klasifikátor z konstantní paměti do paměti globální, není potřeba distribuovat data klasifikátoru více položkám najednou, a naopak by docházelo k serializaci.

### 7.4.3 Paralelní

Naopak *Paralelní* program nejdříve vyhodnotí paralelně všechny příznaky, a teprve potom zjišťuje, zda se klasifikace prošla všemi fázemi nebo ne. Tento program používá na jedno podokno všechna vlákna pracovní skupiny. Očekávali jsme, že bude rychlejší než předchozí *Hybridní* kernel, jelikož počet příznaků v jedné fázi není násobkem warpu, a tudíž některá vlákna jsou ve stavu čekání. Nicméně očekávání se nenaplnilo. Úzkým hrdlem aplikace je totiž čtení hodnot z integrálního obrazu a v tomto přístupu musíme přečíst všechny příznaky pro každý pixel v obraze. Celý kernel se skládá ze tří kroků:

1. výpočet příznaků a uložení do lokálního pole,
2. sken lokálního pole,
3. vyhodnocení kaskády,

V prvním kroku se program spočte všechny příznaky (cca 2000 příznaků) obdobně jako v předešlém přístupu. Poté pracovní skupina o velikosti N provede sken hodnot za pomoci funkce `scan_workgroup`, která je zavolána (N / velikost pracovní skupiny) krát. Lokální pole pak obsahuje sumu prefixů. Ve třetím kroku první warp skupiny vybere hraniční hodnoty fází z lokálního pole a odečte je od sebe, čímž získá sumu všech hodnot fáze a tu porovná s prahovou hodnotou fáze klasifikátoru. Pokud všechny fáze uspějí prahovým testem, je výsledek hodnocen kladně (lokální proměnnou ve sdílené paměti nastavíme před

vyhodnocením na 1, a každé vlákno warpu pak atomickou operací `atomic_and` při neúspěchu nuluje hodnotu). Bez ztráty na výkonu byla implementována i modifikace vyhodnocení, kdy výsledky vyhodnocuje jen jedno vlákno z pracovní skupiny (pak atomická operace není nutná).

### 7.4.4 Neúspěšné optimalizace

Během vývoje detekčního kernelu jsme narazili i na slepé uličky. Práce [7] navrhuje přesunout klasifikátor do lokální paměti, pracovní skupina pak sdílí klasifikátor. Tato varianta je na architektuře Fermi pomalejší přibližně o dobu načítání dat do sdílené paměti. Navíc žádný z pixelů v pracovní skupině se nemusí dostat až k posledním fázím klasifikátoru. Některá data jsou proto načtena zbytečně. Dále jsme omezeni velikostí lokální paměti, a tudíž není možné celý klasifikátor do ní uložit naráz. Další varianta byla přesunout pixely podoken odpovídajícím vláknům v pracovní skupině do lokální paměti, tím urychlit vyhodnocení. Tato varianta též zklamala. Důvodem může být nepotřebnost všech pixelů (obvykle se jedná o krajní hodnoty), dalším důvodem je fakt kešování textury. Následující neúspešnou optimalizací bylo použití konkuretních kernelů. Při hledání dokumentace pro použití v OpenCL jsme narazili na diskusi[2], z níž jasně vyplývá, že použití konkurentních kernelů v OpenCL nefunguje. Diskuse obsahuje i zdrojové kódy (těmi jsme skutečnost potvrdili). Tudíž pro kernely v aplikaci, které pracují s menší globální pracovní velikostí nedochází k plnému využití grafické karty.

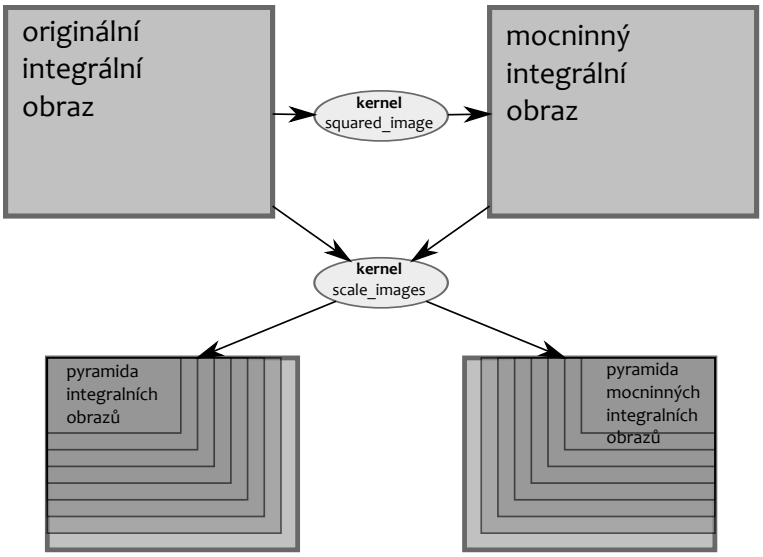
## 7.5 Detekce nad obrazovou pyramidou

V podkapitole 5 jsme diskutovali možnosti detekce ve více měřítkách. Vybrali jsme si prostřední cestu, kdy detekce bude probíhat s konstantní velikostí zkoumaného podokna a budeme pouze měnit velikost integrálního obrazu (nicméně ten budeme generovat jen jednou). Naše volba byla podmíněna časem, jež je potřebný pro tvorbu integrálního obrazu (v prvních verzích byl opravdu pomalý, okolo 10ms pro rozlišení 1280 $\times$720). Rozhodovali jsme se mezi změnou velikosti okna, či zůstat u konstantní velikosti. Vybrali jsme si druhou variantu, ta totiž zachovává blízkost pixelů v podokně. Tento přístup má i své nevýhody. Jednou z nich je méně přesnější detekce menších obličejů, jelikož měřítky jsou pouze přirozená čísla a faktor zmenšení mezi dvěmi měřítky bývá typu `float`, největší skok je mezi originální velikostí a druhou menší následnou velikostí, která je vždy minimálně rovna dvěma. Další měřítka jsou již aproximována lépe.

Tento postup jsme implementovali následovně. Nejdříve si vytvoříme vektor `_scaleVector`, který pak obsahuje faktory zmenšení [3] vzhledem k originálnímu obrazu. Poté procházíme ve smyčce daný vektor a zmenšujeme původní integrální obraz a mocninný integrální obraz o dané měřítko. O zmenšení se stará kernel `scale_images`, jehož parametry jsou dva vstupní obrazy, a výstupem dva zmenšené obrazy. Dříve jsme kernel volali dvakrát, ale osvědčilo se tyto dvě operace sloučit. Všechny objekty jsou již předem inicializované. Jednotlivá rozlišení integrálního obrazu jsou uchovávána ve vektoru `_scaledIntegralImageVec` (podobně i pro mocninný integrální obraz). Na každém rozlišení počítáme ještě buffery pro předávání výsledků mezi kernely, ty použijeme neměnné pro všechna rozlišení, pouze měníme jejich velikost.

[2] http://forums.nvidia.com/index.php?showtopic=207195

[3] Pro rozlišení 1280 $\times$ 720, velikost podokna 20 $\times$ 20 a faktor zmenšení mezi dvěma měřítky je mohutnost vektoru rovna patnácti.

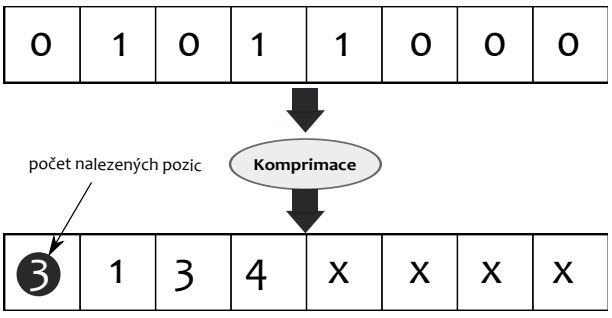



Obrázek 7.2: *Tvorba pyramidy obrazů.*

## 7.6 Reprezentace a distribuce výsledků

Výsledky detekce se předávají mezi hostovskou aplikací i mezi kernely. Oba spolu souvisí. Detekci můžeme rozdělit na bloky po N fázích, jejichž mohutnosti uchovává hostovská aplikace ve vektorovém kontejneru. Například pokud vektor obsahuje hodnoty (2,6), znamená to rozdělení detekce po druhé a osmé fázi. Pro nás to v praxi znamená, že budeme vyhodnocovat první dvě fáze, pak dalších šest fází a následně zbytek. Výsledky klasifikátoru potřebujeme předat dalšímu volání detekčního kernelu.

Uvažujme jedno-dimenzionální pole hodnot o velikosti obrazu. Každý pixel by zapsal na pozici buď 1, pokud by prošel klasifikací, nebo 0 v opačném případě. Takto bychom dostali pole s rozptýlenými hodnotami. To ovšem není žádané.




Obrázek 7.3: *Vstup a výsledek komprimace.*

Naimplementovali jsme dvě možnosti jak zkomprimovat toto rozptýlené pole:

- komprimace atomickou operací,
- algoritmem stream compaction.

Jakmile máme pole zkomprimované, předání výsledků je mezi kernely jednoduché. Kernel detekce dekóduje ze zkomprimovaného pole souřadnice (udávající pozici pixelu v integrálním obrazu), které určují podokno ke zpracování. Výsledek zapisují buď pomocí atomické operace již do zkomprimovaného pole, nebo do pole s rozptýlenými položkami (0, nebo 1). Mezi kernely se pak provede komprimace, anebo pouhé přehození adres bufferů (zkomprimovaného vstupu a výstupu). Jedno pole použít nemůžeme, protože bychom si přepisovali vstupní hodnoty.

### 7.6.1 Tvorba komprimovaného pole za pomocí atomické operace

První možností je takové pole vůbec nevytvářet a vytvořit rovnou pole zkomprimované s pomocí atomických operací. Na první pozici pole uchováváme velikost zkomprimovaného pole (není předem známa). Na začátku je inicializována na 0. Atomická operace `atomic_add` vrátí původní hodnotu a zvýší nastávající hodnotu o jedna, navíc nám zajistí, že žádné jiné vlákno v danou chvíli nezapisuje též do pole. Vrácená hodnota + 1 nám značí index zápisu nynější pozice. Postup provádíme pouze pokud detekce vrátí kladnou odezvu, je-li odezva záporná, není nic potřeba zapisovat.

### 7.6.2 Komprimace algoritmem Stream compaction

Druhou možností je použít algoritmus stream compaction. O ten se v knihovně starají čtyři kernely (`scan_step1`, `scan_step2`, `scan_step3` a `compact`), neboli dvě primitiva.

První tři kernely se starají o sken, jak je popsáno v kapitole 5. Sken je z velké části převzat z Nvidia SDK, nicméně jsme jej museli upravit. Správně fungoval pro pole o maximální velikosti 262 144 [4]. Toto číslo je limitováno druhým krokem skenu, kterému je přidělena jen jedna pracovní skupina skenující výsledky předchozího kroku. Správně by maximální číslo mělo být rovno $256 \cdot 256$, ale první kernel je optimalizován pro práci nad vektorovým polem `int4`), kdy každé vlákno provede sken navíc nad jedním čtyř-složkovým vektorem a paralelně se skenuje až výsledek. Upravili jsme prostřední krok, aby provedl postupný sken potřebných čísel jednou pracovní skupinou. Mohli jsme spustit rekurzivně celý sken znovu, nicméně tento druhý krok je nejrychlejší částí skenu (několikanásobně než zbylé kroky), takže jsme si tuto úpravu mohli dovolit a zjednodušit režiji hostovské aplikaci.

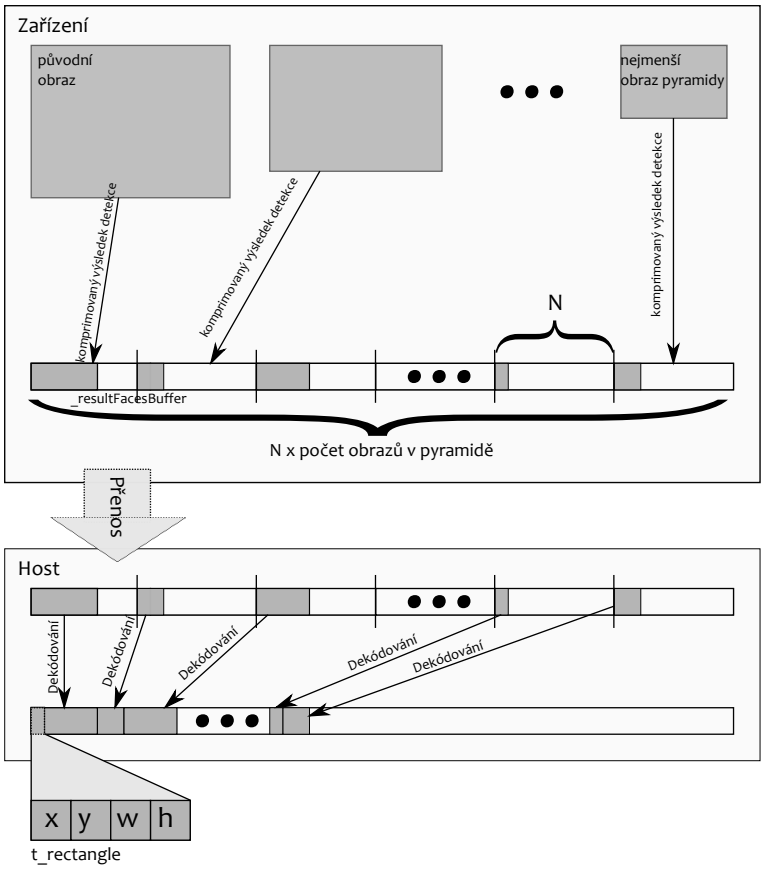
Kernel `compact` je jednoduchý. Na vstupu má původní pole (pouze nuly a jedničky) a jeho sken. Kernel je spuštěn nad jedno-dimenzionálním polem. Pokud se v původním poli nachází hodnota 1, vezmeme odpovídající hodnota na stejném indexu ze skenu (ta udává adresu do výsledného pole). Na tuto adresu je pak zapsán globální identifikátor vlákna (`get_global_id(0)`), který určuje pozici z původního pole.

### 7.6.3 Předávání výsledků hostovské aplikaci

Doporučuje se slučovat více přenosů do jednoho přenosu a minimalizovat objem přenesených dat. Proto nejsou výsledky v aplikaci předávány po každém vyhodnoceném rozlišení, ale až po vyhodnocení celého snímku.

Pro reprezentaci výsledků jsme vytvořili nový buffer (`_resultFacesBuffer`), do kterého se ukládají mezivýsledky (již zkomprimované) po každé poslední fázi detekce na daném rozlišení. Jednomu výsledku z jednoho rozlišení připadá jeden spojitý blok hodnot v bufferu `_resultFacesBuffer`, index bloku je roven indexu měřítka z vektoru měřítek

[4] tj. $(4 \cdot 256) \cdot 256$, velikost pracovní skupiny je 256




Obrázek 7.4: *Předávání výsledků hostovské aplikaci a dekódování.*

(`_scaleVector`). Velikost bloku se definuje v OpenCL programu makrem `MAX_RESULTS_PER_SCALE`. To znamená, že předem omezíme maximální počet obličejů. A ostatní obličeje, jež překročí `MAX_RESULTS_PER_SCALE` budou zahozeny. Při dostatečně velké hodnotě tohoto makra nedojde k žádné ztrátě (osvědčila se hodnota 4096 na jedno měřítko). Tento přístup je odůvodněný. Nemusíme předávat aplikaci celé pole o velikosti obrazu pro každé měřítko. Mohli bychom nahrávat jen potřebný počet dat, a to tak, že z paměti grafické karty nejdříve přeneseme první hodnotu ze zkomprimovaného pole (ta nám udává počet výsledků) a načteme pouze tolik prvních hodnot. Nicméně tím dochází k přenosou dvakrát, a tudíž je tato varianta pomalejší než námi implementovaný postup.

Hostovská aplikace toto pole dekóduje (využívá znalosti struktury bufferu s výsledky) a vytvoří pole struktur `t_rectangle`, jež přesně definují pozici a velikost obličeje.

# Kapitola 8

# Demonstrační aplikace

Vytvořili jsme jednoduchou aplikaci, která demonstruje použití naší knihovny. Implementovaná aplikace pracuje s těmito komponentami:

- navržený detektor,
- kodek,
- Fltk (Fast light toolkit),
- POSIX threads.

K dekódování videa jsme použili kodek převzatý od Ing. Lukáše Poloka, a mírně upravili (přepsali formát výstupu a zastaralé volání funkcí nahradili novými) k našim potřebám. Kodek využívá následujících komponent FFmpeg: knihovnu libavformat (dekódování videa), libavutil (pomocná knihovna FFmpeg) a libswscale (změna barevného modelu videa). Práce s kodekem je jednoduchá, vystačíme si se třemi funkcemi ( `OpenVideoFile`, `GetVideoFrame` a `CloseVideoFile`), jejich názvy jsou sebevysvětlující. Video pro zjednodušení dekódujeme přímo ve stupních šedi. Tak upravený obraz předpokládá naše knihovna na vstupu.

O okno aplikace a posílání zpráv se stará Fltk, což je multiplatformní knihovna grafického uživatelského rozhraní. Tu jsme si vybrali pro její jednoduchost. Výsledky a snímek videa vykreslujeme do objektu vytvořené třídy `FltkDrawBox`, která dědí třídu `Fl_Box` a přidává jí metody pro zobrazení výsledků, snímku videa a textových informacích (např. počet zpracovaných snímků za sekundu).

Pro zvýšení rychlosti zpracování snímků videa jsme rozdělili aplikaci do tří vláken. Hlavní vlákno programu obsluhuje grafické uživatelské rozhraní, další vlákno detekci a poslední dekódování videa. Tudíž detekce a dekódování probíhají paralelně, a jsou synchronizovány vždy s následujícím snímkem.

Program se spouští přes příkazovou řádku, přes kterou se předávají programu základní argumenty (např. cesta k video souboru). Po spuštění se objeví okno s běžícím videem, v němž jsou detekovány obličeje. Aplikace se ukončuje buď křížkem okenní aplikace nebo klávesou `Esc`.

# Kapitola 9

# Testování a výsledky

V této kapitole budeme prezentovat dosažené výsledky. Budeme se věnovat i nejlepšímu výběru parametrů, porovnáváme implementovanané přístupy detekce, a porovnáme naše výsledky s CPU implementací v knihovně OpenCV, ze které byl klasifikátor převzat. Použitý klasifikátor je uložen v souboru *haarcascade_frontalface_alt.xml*. Obsahuje 22 fází a je natrénován na podokna 20 × 20. Celkový počet příznaků klasifikátoru je roven 2135 (každý příznak se navíc skládá dvou nebo tří haarových obdelníků).

| Operační systém | Typ | Microsoft Windows 7 64b |
|---|---|---|
| Procesor | Typ | Intel Core i3-2100 |
| | Počet jader | 2 |
| | Takt procesoru | 3.1GHz |
| Paměť RAM | Velikost | 8GB |
| Grafická karta | Typ | GeForce GTX 550 Ti |
| | Počet SM jader | 4 |
| | Velikost paměti | 1GB |

Tabulka 9.1: *Testovaná architektura.*

Testování a vývoj probíhal na sestavě, již ukazuje tabulka 9.1. Testovaná grafická karta GeForce GTX 550 Ti nepatří mezi nejvýkonější dostupné karty, spíše se řadí do nižší třídy. Proto je velice pravděpodobné, že na grafických kartách vyšší třídy s podobnou architekturou (Fermi) poběží aplikace několikanásobně rychleji. Aplikaci jsme testovali pouze na grafické kartě firmy Nvidia.

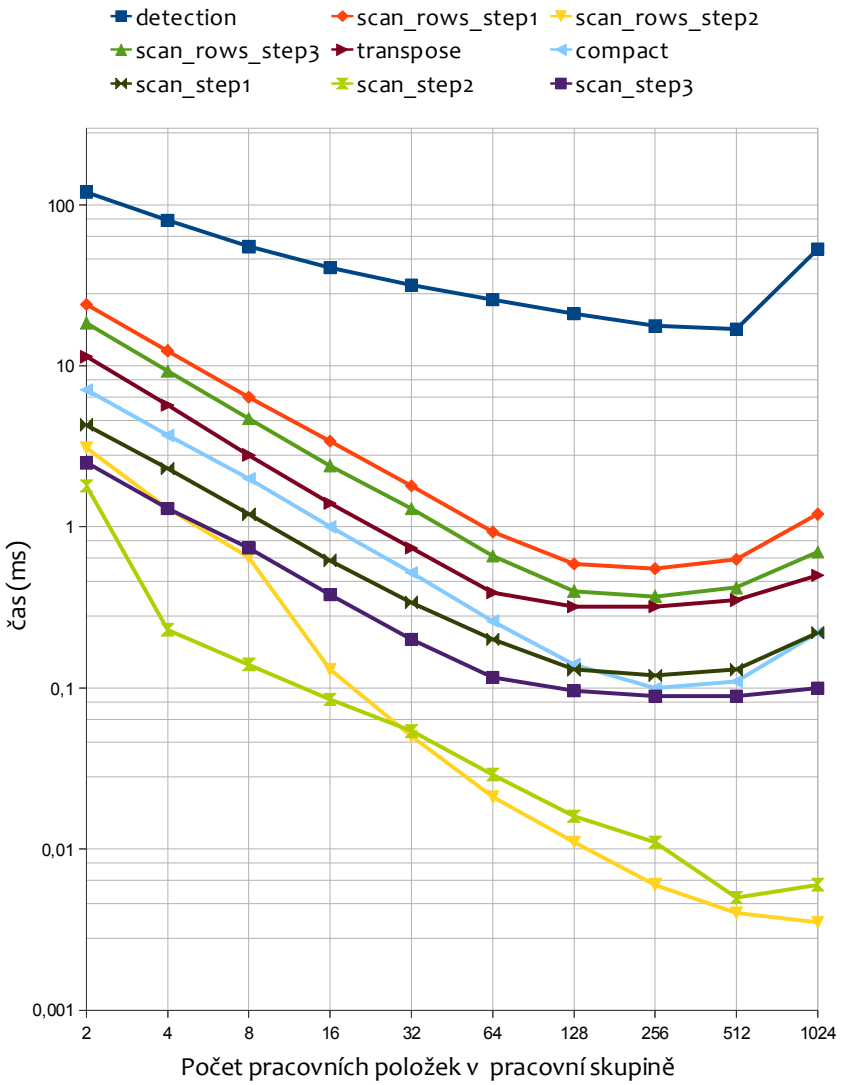
## 9.1 Hledání optimální velikosti pracovních skupin

Pro nejefektivnější využití prostředků testované grafické karty provedeme experimentální měření. Budeme se zabývat zejména vlivem velikosti pracovních skupin na rychlost kernelu. Hledáme tak nejrychlejší řešení.

### 9.1.1 Měření 1

První experimentální měření jsme provedli na kernelech, u nichž nelze měnit globální pracovní velikost na které se spouští, jelikož ta je proměnná podle velikosti dat vstupujícího do kernelu. Vstupem měření byl statický obraz o velikosti 1280 × 720 pixelů. Zkoumali jsme

vliv pouze se sekvenčním detekčním kernelem, který prováděl detekci podoknem o velikosti $20 \times 20$. Naše výsledky měření jsme zanesli do grafu 9.1.

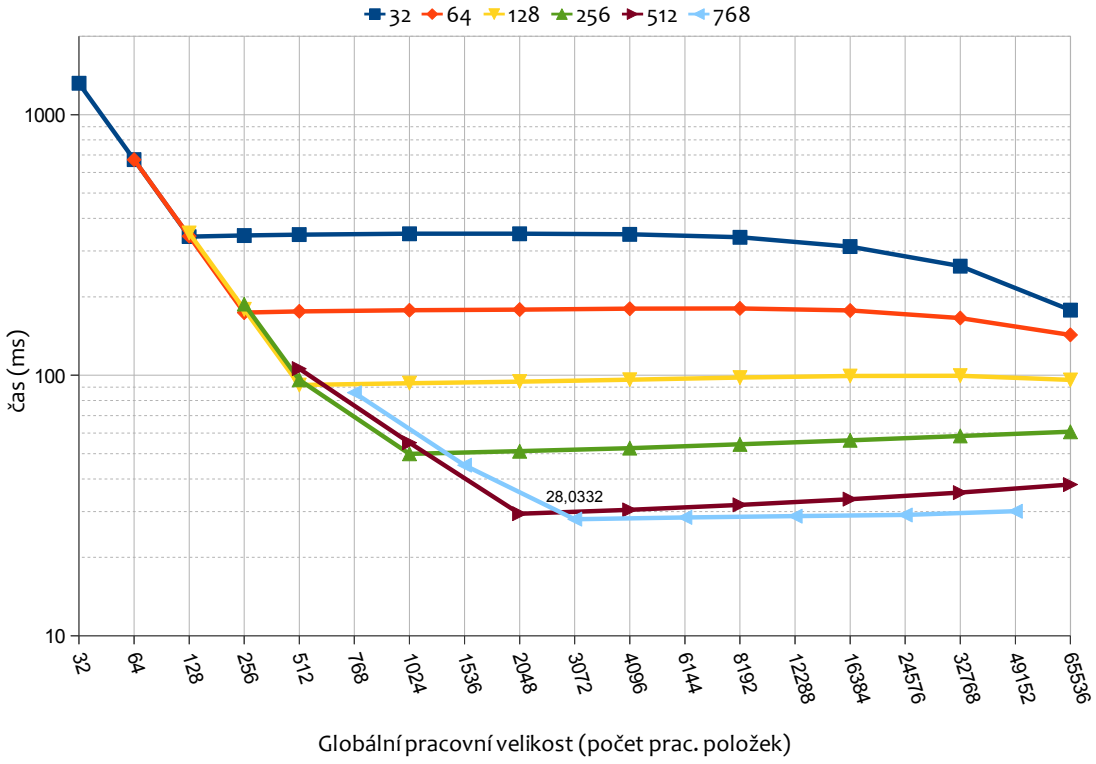



Obrázek 9.1: *Graf závislosti rychlosti zpracování kernelů na velikosti pracovní skupiny.*

Z grafu je patrné, že většina kernelů má minimální čas při použití velikosti pracovní skupiny o 256 pracovních položkách. Výjimku tvoří kernely druhých kroků skenů (`scan_rows_step2` a `scan_step2`), které dosahují minima pro větší velikosti pracovní skupiny. Tento výsledek u nich není zarážející, jelikož pro svůj výpočet používají jen jednu pracovní skupinu. Překvapující je výsledek detekčního kernelu, jehož minimální čas je dosažen při velikosti pracovní skupinu o 512 pracovních položek, i přesto, že Visual profiler určil jeho obsazenost (*occupancy*) na 66.7 procent. Naproti tomu při použití pracovní skupiny o velikosti 256 prac. položek dosahuje obsazenosti 83,3 procent. Z důvodu větší obsazenosti jsme zůstali u 256 položek (rozdíl v rychlosti není výrazný).

### 9.1.2 Měření 2

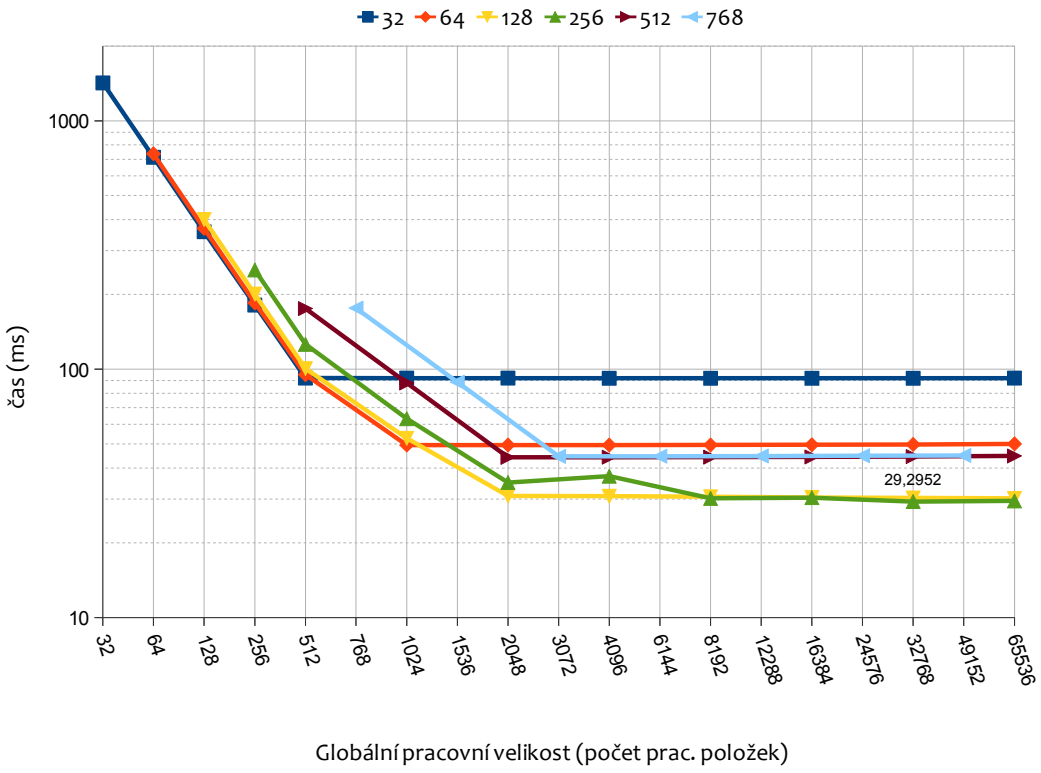
Druhé experimentální měření jsme provedli na detekčních kernelech, u nichž lze měnit globální pracovní velikost. Ta má totiž u detekčních kernelů, které zpracovávají více příznaků současně, signifikantní podíl. Zkoumali jsme změnu zpracovaného času detekčních kernelů v závislosti na změně globální pracovní velikosti a lokální pracovní velikoti (pozn. oba zkoumané kernely byly spouštěné na 1 dimenzionální mřížce). Vstupem měření bylo video o velikosti 1280 × 720. Výsledky měření jsme zprůměrovali z 1000 snímků. První dvě fáze počítal sekvenční kernel, zbytek (dvacet fází) testovaný kernel. Právě časy testovaného kernelu jsme zanesli do grafu.




Obrázek 9.2: *Graf ovlivnění rychlosti výpočtu kernelu* `detection_hybrid` *různou globální a lokální pracovní velikostí.*

Graf 9.2 obsahuje výsledky měření pro kernel `detection_hybrid`. Tento graf ukazuje, že kernel dosahuje minima při velikosti pracovní skupiny 768 pracovních položek a pro globální pracovní velikost 3072 položek. To odpovídá čtyřem pracovním skupinám. Tyto čtyři pracovní skupiny zpracovávají všechny podokna klasifikátoru. Větší velikost mocniny dvou než 512 pracovních položek v jedné pracovní skupině nelze na testované architektuře spustit. Překladač nepovolil velikost 1024 pracovních položek pro tento kernel. Proto jsme jako další testovanou velikost zvolili 768.

Další graf 9.2 ukazuje výsledky měření pro kernel `detection_parallel`. U tohoto měření jsme provedli malou obměnu v počtu fází, které měřený kernel klasifikoval. Sekvenční kernel klasifikoval prvních osm fází namísto dvou, tudíž na paralelní kernel zbylo šestnáct fází.




Obrázek 9.3: *Graf ovlivnění rychlosti výpočtu kernelu* `detection_parallel` *různou globální a lokální pracovní velikostí.*

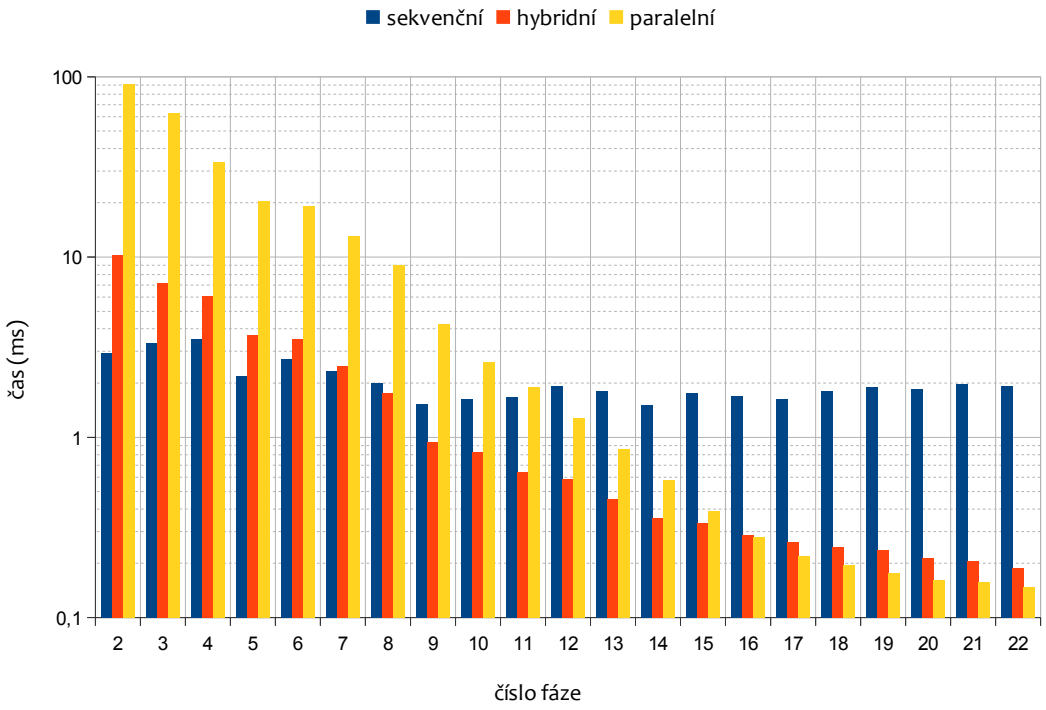
Modifikaci v měření jsme zavedli kvůli rychlosti vyhodnocení, jelikož paralelní klasifikátor vyhodnocuje všechny fáze. V tomto grafu již minimum není plně zřetelné, je označeno příslušným časem. Testovaný kernel je nejrychlejší v případě, kdy počítá 256 (tj. velikost pracovní skupiny) příznaků paralelně.

## 9.2 Hledání bodu zlomu klasifikátoru

Na konci kapitoly 5 jsme uvedli, že je vhodné po určité fázi výpočet klasifikátoru rozdělit a počítat příznaky jednoho podokna paralelně. Naimplementovali jsme tři různé detekční kernely, v ideálním případě existují dva body, ve kterých klasifikátor rozdělit a počítat odlišným způsobem. V minulé podkapitole jsme určili nejlepší konfigurace velikosti pracovních skupin detekčních kernelů. S těmito maximy jsme provedli měření, ve kterém jsme se zaměřili na hledání těchto bodů zlomu klasifikátoru. Test probíhal na videu se stejnými parametry jako minulé měření. Spouštěli jsme program po fázích. Zkoumaný kernel počítal vždy jen jednu fázi, zbytek výpočtu probíhal sekvenčním kernelem. Výsledky sekvenčního kernelu pro jednu fázi jsme získali odečtem časů získaných při měření komplementárních kernelů.

Výsledky jsme zanesli do grafu 9.4. Z grafu je patrné který kernel je pro dané fáze rychlejší. K prvnímu zlomu dochází v osmé fázi, kdy rychlost kernelu `detection_hybrid`

překračuje sekvenční kernel. Další zlom je ve dvanácté fázi, kdy paralelní kernel překračuje kernel sekvenční, nicméně je stále pomalejší než hybridní kernel. Paralelní kernel předčí hybridní až v šestnácté fázi. Avšak rozdíl v rychlosti obou přístupů je v těchto fázích minimální (v řádu setin milisekund).




Obrázek 9.4: *Graf času výpočtu jednotlivých fází pro různé kernely detekce.*

| Fáze rozdělení | Celkový čas |
|---|---|
| 1 | 21,46ms |
| 2 | 20,30ms |
| 3 | 20,80ms |
| 4 | 22,98ms |
| 5 | 24,77ms |
| 6 | 27,19ms |
| 7 | 29,92ms |

Tabulka 9.2: *Změna celkového času vyhodnocení osmi fází klasifikátoru sekvenčním kernelem v závislosti na rozdělení po určité fázi.*

Dostali jsme se k takové konfiguraci, kdy první sekvenční kernel má vyhodnocovat prvních osm fází. To je poměrně velká část klasifikátoru, která odmítá vysoké procento potencionálních obličejů i před osmou fází, čímž může být mnoho pracovních položek ve stavu čekání (bude docházet ke hladovění jak jsme uvedli v kapitole 5). U paralelního a hybridního

kernelu k němu nedochází, protože skupiny pracovních položek, které vyhodnotily pozici, jsou znovu použity na vyhodnocení dalších podoken. U sekvenčního přístupu je tak výhodné v určité fázi rozdělit klasifikátor a znovu na něm spustit sekvenční kernel. Tím obnovíme nevyužité zdroje grafické karty. Proto jsme provedli další experiment, kterým jsme zjistili, po které fázi klasifikátor rozdělit. Průměrný čas běhu sekvenčního kernelu bez dělění fází byl změřen na 32ms. Ostatní časy vidíme v tabulce 9.2.

Tabulka ukazuje, že náš odhad byl správný. Rozdělením vyhodnocení klasifikátoru můžeme ušetřit až třetinu času. Nejvýhodnější je rozdělit výpočet sekvenčního kernelu již po druhé fázi klasifikace.

## 9.3 Dosažené výsledky

V předchozích podkapitolách jsme určili potřebné optimální parametry vhodné pro testovanou architekturu. S těmito parametry jsme spustili aplikaci a výsledné časy zapsali do tabulky 9.3. Testovali jsme dvě různá videa ve třech různých rozlišeních, do tabulky jsme zanesli průměrné časy. Aplikaci jsme spustili s nastavením bez atomických operací. Paralelní kernel nebyl zahrnut do spouštění, jelikož jeho výsledky jsou jen o málo lepší než časy kernelu hybrid. Tento rozdíl je menší než čas potřebný na režii předání výsledků mezi těmito dvěma kernely.

Z tabulky vidíme, že operace předzpracování obrazu trvají poměrně dlouhou dobu. To je způsobeno použitím objektu textury s jedním kanálem. Sken proto není efektivní jako u algoritmu *stream compaction*. Nepracuje totiž s polem čtyř-rozměrných vektorů, ale zpracovává pouze jednu hodnotu na jedno vlákno. Z výsledků je patrné, že knihovna dokáže pracovat s HD videem v reálném čase.

| Zařazení | Kernel | 640×360 | 1280×720 | 1920×1080 | Obsazenost |
|---|---|---|---|---|---|
| Detekce | detection | 8,04 | 19,54 | 31,43 | 83,3% |
| | detection_hybrid | 2,24 | 4,88 | 6,77 | 50% |
| Mezisoučet | | 10,28 | 24,41 | 38,19 | – |
| Pyramida obrazů | transpose | 0,17 | 0,64 | 1,45 | 100% |
| | scan_rows_step1 | 0,33 | 1,14 | 2,63 | 100% |
| | scan_rows_step2 | 0,01 | 0,02 | 0,02 | 100% |
| | scan_rows_step3 | 0,21 | 0,76 | 1,17 | 100% |
| | squared_image | 0,11 | 0,40 | 1,08 | 100% |
| | scale_image | 0,19 | 0,61 | 1,43 | 100% |
| Mezisoučet | | 1,03 | 3,60 | 7,78 | – |
| Strem compaction | scan_step1 | 0,29 | 0,75 | 1,48 | 100% |
| | scan_step2 | 0,18 | 0,54 | 1,17 | 100% |
| | scan_step3 | 0,21 | 0,54 | 1,08 | 100% |
| | compact | 0,28 | 0,79 | 1,58 | 100% |
| Mezisoučet | | 0,96 | 2,63 | 5,31 | – |
| **Celkem** | | **12,28** | **30,63** | **51,29** | – |

Tabulka 9.3: *Dosažené výsledky pro různá rozlišení videa. Hodnoty udané v tabulce jsou v milisekundách.*

## 9.4 Srovnání s implementací OpenCV

Natrénovaný klasifikátor jsme převzali z knihovny OpenCV zaměřené na algoritmy počítačového vidění. Tudíž máme dostupnou CPU implementaci, se kterou můžeme srovnávat naše výsledky. Hlavní rozdíl mezi OpenCV verzí spočívá ve změně velikosti příznaků. V naší knihovně jsou příznaky škálovány pouze celočíselným faktorem. Výsledné porovnání tudíž nebude zcela přesné. Nicméně zbytek vyhodnocení by měl být ekvivalentní. Použili jsme knihovnu OpenCV ve verzi 2.1.

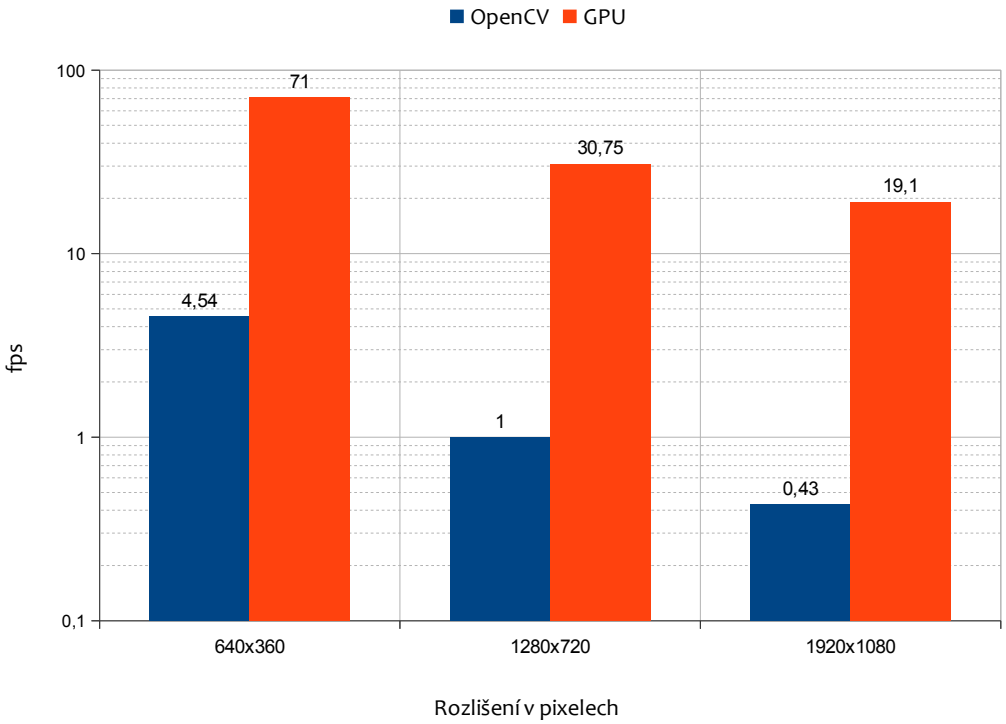
Výpis 9.1: *Volání detekce v OpenCV.*

```
1  CvSeq* faces = cvHaarDetectObjects (img, cascade, storage,
2                                      1.2, 1, 0, cvSize(20, 20) );
```

K srovnávacímu měření nám posloužil statický obraz (viz obrázek 9.5) ve třech různých rozlišeních. Parametry volání funkce detekce jsme zvolili ekvivalentní naší implementaci (zaznamenány jsou ve výpisu 9.1). Měřili jsme volání pouze této funkce. Naopak v naší implementaci jsme měřili celý proces zpracování, to znamená i s předáním a vykreslením výsledků a také s dekódováním obrazu (vytvořili jsme video se statickým obrazem).


Obrázek 9.5: *Testovaný obraz s vyznačenou odezvou naší aplikace.*

Naměřené hodnoty si můžeme prohlédnout na grafu 9.6. Zrychlení není překvapující, hodnoty jsou spíše informativní. Přesto lze vidět zásadní rozdíl v sekvenčním a paralelním přístupu řešení.




Obrázek 9.6: *Porovnání naší GPU implementace s CPU implementací knihovny OpenCV. Osa Y udává teoretický počet podobných snímků za sekundu.*

# Kapitola 10

# Závěr

Úspěšně jsme naimplementovali knihovnu pro detekci obličejů schopnou zpracovávat snímky HD videa v reálném čase. Vytvořili jsme jednoduchou aplikaci, která demonstruje použití knihovny. Prostudovali jsme jak problematiku týkající se detekce obličejů, tak využití grafických akcelerátorů pro obecné výpočty. Diskutovali jsme různé možnosti namapování procesu detekce na GPU. Pro detekci obličejů jsme zvolili natrénovaný klasifikátor, jež pracuje s haarovými příznaky a je volně dostupný v knihovně OpenCV. K vývoji programu jsme vybrali standard OpenCL.

V jazyku OpenCL jsme naimplementovali tři detekční kernely. Každý z nich slouží pro jiné fáze klasifikace. Z testování aplikace je zřejmé, že nejvíce času připadá na první spouštěný kernel. Ten je nutné spustit nad všemi pixely obrazu. Snažili jsme se o co nejvyšší obsazenost prvního detekčního kernelu, proto musel být jednoduchý. Další dva implementované kernely jsou složitější, nicméně vyhodnocují příznaky pod jedním podoknem paralelně, tím pádem dosahují lepších výsledků u mohutnějších fází klasifikátoru. Detekce pracuje s integrálním obrazem. Ten jsme prostudovali a naimplementovali tak, aby též těžil z paralelního prostředí grafické karty. Vytvořili jsme pomocné kernely sloužící k distribuci výsledků i mezivýsledků. Snažili jsme se minimalizovat počet přenesených dat z a do grafické karty. I proto naše implementace předpokládá vstup již šedotónového obrazu. Ten může být paralelně předzpracován hostovským programem.

Vytvořenou aplikaci jsme otestovali na dostupné architektuře. Použitá grafická karta nepatří mezi špičku a je pravděpodobné, že na výkonějších grafických kartách poběží detekce rychleji. Aplikaci jsme vyvíjeli a testovali pouze na architektuře Fermi firmy NVIDIA. Nemůžeme zaručit, že pro jiný typ bude optimální. Provedli jsme experimentální měření. Těmi jsme zjistili nejoptimálnější parametry pro testovanou grafickou kartu. Výsledky jsme zanesli do grafů a tabulek. Porovnali jsme naše GPU řešení s CPU řešením (OpenCV). Zrychlení je patrné. Výsledek jsme očekávali, jelikož se jedná o paralelní úlohu.

Program je možné vylepšit. První vylepšení vyplývá z použitého klasifikátoru, kdy jsme se omezili pouze na slabé klasifikátory typu *stump*, které nejsou plnohodnotnými stromy. Avšak zobecnění by mohlo přivézt zpomalení, jelikož by docházelo k vyšší divergenci vláken, která je patrná ze stromové struktury. Dalším vylepšením je plánování kernelů konkurentně, které jsme zavrhli při hledání použití této funkce v OpenCL. Zjistili jsme, že není dostupné, ačkoliv CUDA jej podporuje. Nicméně do budoucnu by mělo být dostupné. Zajímavým rozšířením by bylo provádět výpočet na více GPU jednotkách tím stylem, že každá GPU jednotka by se starala o detekci na různých rozlišeních obrazové pyramidy.

# Literatura


[1] Bilgic, B.; Horn, B.; Masaki, I.: Efficient Integral Image Computation on the GPU. In *Intelligent Vehicles Symposium*, ročník 4, 2010, s. 528–533.

[2] Blelloch, G. E.: Prefix Sums and Their Applications. *Synthesis of Parallel Algorhitms*, 1991.

[3] Crow, F. C.: Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, New York, NY, USA: ACM, 1984, ISBN 0-89791-138-5, s. 207–212.

[4] Duda, R.; Hart, P. E.; Stork, D. H.: *Pattern classification*. New York: J. Wiley, druhé vydání, 2001, iSBN 0-471-05669-3.

[5] Freund, Y.; Schapire, R. E.: A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, ročník 55, č. 1, Srpen 1997: s. 119–139, ISSN 0022-0000.

[6] Harris, M.; Sengupta, S.; Owens, J.: Stream Reduction Operations for GPGPU Applications. In *GPU Gems 2*, editace H. Ngyen, Addison Wesley, 2007, s. 413–442.

[7] Hefenbrock, D.; Oberg, J.; Thanh, N. T. N.; aj.: Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, Washington, DC, USA: IEEE Computer Society, 2010, ISBN 978-0-7695-4056-6, s. 11–18.

[8] Herout, A.; Jošth, R.; Juránek, R.; aj.: Real-time object detection on CUDA. *J. Real-Time Image Process.*, ročník 6, č. 3, Září 2011: s. 159–170, ISSN 1861-8200.

[9] Hillis, W. D.; Steele, G. L., Jr.: Data parallel algorithms. *Commun. ACM*, ročník 29, č. 12, Prosinec 1986: s. 1170–1183, ISSN 0001-0782.

[10] Huang, W.; Wu, L.-D.; Zhang, Y.-G.: GPU-Based Computation of the Integral Image. *International Conference on Virtual Reality and Visualization*, ročník 0, 2011: s. 243–246.

[11] Advanced Micro Devices Inc: Programming Guide ATI Stream Computing. *Micro*, August 2010.

[12] Ke, Y.; Sukthankar, R.; Hebert, M.: Efficient Visual Event Detection Using Volumetric Features. In *Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1 - Volume 01*, ICCV '05, Washington, DC, USA: IEEE Computer Society, 2005, ISBN 0-7695-2334-X-01, s. 166–173.

[13] Khronos OpenCL Working Group: The OpenCL Specification, version 1.1 [online]. http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf, Leden 2011.

[14] Lienhart, R.; Maydt, J.: An extended set of Haar-like features for rapid object detection. In *Image Processing. 2002. Proceedings. 2002 International Conference on*, ročník 1, IEEE, 2002, ISBN 0-7803-7622-6, ISSN 1522-4880, s. I–900–I–903 vol.1.

[15] Messom, C.; Barczak, A.: High Precision GPU based Integral Images for Moment Invariant Image Processing Systems. In *Electronics New Zealand Conference (ENZCON'08)*, 2008.

[16] NVidia: CUDA FAQ [online]. http://developer.nvidia.com/cuda-faq, 2008-11-01 [cit. 2008-11-28].

[17] NVidia: NVIDIA's Next Generation CUDA Compute Architecture: Fermi [White paper]. http://cs.jhu.edu/~jason/papers/#istv91, 2009.

[18] Obukhov, A.: Face detection with CUDA. In *2009 GPU Technology Conference (GTC 2009)*, 2009.

[19] Obukhov, A.: Haar Classifiers for Object Detection with CUDA. In *GPU Computing Gems*, editace W.-M. W. Hwu, Elsevier Inc, 2011, s. 517–543.

[20] Oro, D.; Fernandez, C.; Saeta, J. R.; aj.: Real-time GPU-based face detection in HD video sequences. In *ICCV Workshops*, 2011, s. 530–537.

[21] Roy, A.; Marcel, S.: Haar Local Binary Pattern Feature for Fast Illumination Invariant Face Detection. In *Proc. BMVC*, 2009, ISBN 1-901725-39-1, s. 19.1–19.12.

[22] Schapire, R. E.; Singer, Y.: Improved Boosting Algorithms Using Confidence-rated Predictions. *Mach. Learn.*, ročník 37, č. 3, Prosinec 1999: s. 297–336, ISSN 0885-6125.

[23] Sengupta, S.; Harris, M.; Garland, M.: Efficient Parallel Scan Algorithms for GPUs. Technická Zpráva NVR-2008-003, NVIDIA Corporation, Prosinec 2008.

[24] Sengupta, S.; Harris, M.; Garland, M.; aj.: Efficient Parallel Scan Algorithms for Many-core GPUs. In *Scientific Computing with Multicore and Accelerators*, editace J. Kurzak; D. A. Bader; J. Dongarra, Chapman & Hall/CRC Computational Science, kapitola 19, Taylor & Francis, Leden 2011, ISBN 978-1-4398253-6-5, s. 413–442.

[25] Sharma, B.; Thota, R.; Vydyanathan, N.; aj.: Towards a robust, real-time face processing system using CUDA-enabled GPUs. In *HiPC*, 2009, s. 368–377.

[26] Tyanyun, N.: Direct Compute [online]. http://www.nvidia.com/content/GTC/documents/1015_GTC09.pdf, 2009 [cit. 2011-12-28].

[27] Viola, P.; Jones, M.: Robust Real-time Object Detection. In *International Journal of Computer Vision*, 2001.

[28] Wikipedia: CUDA. — Wikipedia, The Free Encyclopedia [online]. http://en.wikipedia.org/wiki/CUDA, 2011, [cit. 27-11-2011].

[29] Wikipedia: CUDA. — Wikipedia, The Free Encyclopedia [online]. http://en.wikipedia.org/wiki/OpenCL, 2011, [cit. 21-12-2011].

[30] Wikipedia: Direct Compute. — Wikipedia, The Free Encyclopedia [Online]. http://en.wikipedia.org/wiki/DirectCompute, 2011, [cit. 21-12-2011].

[31] Wikipedia: Haar wavelet. — Wikipedia, The Free Encyclopedia [online]. http://en.wikipedia.org/wiki/Haar_wavelet, 2011, [cit. 22-11-2011].

[32] WWW stránky: OpenCV [online]. http://opencv.willowgarage.com, 2008-11-01 [cit. 2011-12-27].

[33] Yang, M.-H.; Kriegman, D. J.; Ahuja, N.: Detecting Faces in Images: A Survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, ročník 24, č. 1, Leden 2002: s. 34–58, ISSN 0162-8828.

[34] Zhang, C.; Zhang, Z.: A Survey of Recent Advances in Face detection. 2010.

# Příloha A

# Obsah DVD

Obsah přiloženého DVD:

- zdrojové soubory,
- dokumentace,
- manuál,
- plakát,
- ukázková videa.