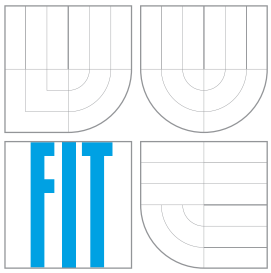
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DETEKCE OBJEKTŮ NA GPU

OBJECT DETECTION ON GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN JURÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ROMAN JURÁNEK, Ph.D.

BRNO 2015

Abstrakt

Tato práce je zaměřena na akceleraci detekce objektů v obraze metodou Random Forest. Detektor Random Forest se skládá ze souboru náhodných rozhodovacích stromů, které jsou na sobě nezávisle vyhodnocovány, čehož lze využít pro akceleraci na grafické jednotce. Vývoj a zvyšování výkonu grafických procesorů umožnilo použití GPU pro masivně paralelní obecné výpočty (GPGPU). Cílem této práce je popsat způsob implementace metody Random Forest na GPU s využitím standardu OpenCL.

Abstract

This thesis is focused on the acceleration of Random Forest object detection in an image. Random Forest detector is an ensemble of independently evaluated random decision trees. This feature can be used to acceleration on graphics unit. Development and increasing performance of graphics processing units allow the use of GPU for general-purpose computing (GPGPU). The goal of this thesis is describe how to implement Random Forest method on GPU with OpenCL standard.

Klíčová slova

Detekce objektů, GPGPU, GPU, OpenCL, Random Forest

Keywords

Object detection, GPGPU, GPU, OpenCL, Random Forest

Citace

Martin Jurák: Detekce objektů na GPU, diplomová práce, Brno, FIT VUT v Brně, 2015

Detekce objektů na GPU

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracoval samostatně pod vedením pana Ing. Romana Juránka Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Jurák
28. července 2015

Poděkování

Rád bych poděkoval Ing. Romanu Juránkovi, Ph.D. za poskytnutí již natrénovaného klasifikátoru pro detekci, odborné vedení a cenné rady při práci na diplomovém projektu. Mé poděkováni patří též Ing. Michalu Kulovi za rady a trpělivost při konzultacích ohledně obecných výpočtů na GPU.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Detekce objektů v obraze se v dnešní době používá v mnoha různých odvětvích: lékařství, bezpečnostní systémy, robotika, automobily, herní konzole a mnoho dalších. S tak širokým uplatněním a rychlým vývojem informačních technologií bylo představeno velké množství způsobů a metod pro detekci a rozpoznání objektů v obraze.

Jedním ze způsobů detekce objektů jsou klasifikační metody Random Forests[2][6], které jsou použity v této práci. Tyto metody používají soubor náhodných rozhodovacích stromů pro regresi a klasifikaci, kam spadá i detekce objektů. Jelikož je každý strom při trénování i klasifikaci vyhodnocován samostatně, je metoda vhodná pro akceleraci na paralelním hardwaru.

V mnoha zmíněných aplikacích je pro detekci objektů důležitá rychlost, kdy potřebujeme obraz zpracovávat real-time. Donedávna byly používány hlavně rychlé implementace pro CPU, nicméně během posledních let došlo k vývoji grafických procesorů. Ten umožňuje jejich použití v jiném odvětví než 3D grafice a zpracování obrazu. Pro tuto potřebu jsou vytvořeny aplikační rozhraní pro využití grafických procesorů pro obecné výpočty, jako OpenCL, CUDA a Direct Compute.

Tato práce se zabývá akcelerací na GPU metody Random Forests pro detekce objektů v obraze. V první části práce je popsána klasifikační metoda Random Forests a její využití pro detekci obrazu. Na tuto část navazuje kapitola s popisem možností akcelerace výpočtů na GPU, architektura standardu OpenCL a metody pro efektivní využití prostředků grafického procesoru. V kapitole 5 je popsán použitý klasifikátor pro detekci objektů a je zde navrhnuta a psána implementace na GPU. V závěrečné kapitole je provedeno testování rychlosti a úspěšnosti detekce na testovacích datech.

Kapitola 2

Detekce objektů v obraze

Detekce objektů se zabývá lokalizací hledaného objektu v obraze, kombinací metod zpracování obrazu a strojového učení. Typ detekovaného objektu záleží na natrénovaném detektoru, což umožňuje tyto metody použít ve velkém množství oblastí.

V průběhu let se našlo mnoho způsobů jak detekovat objekty v obraze. Yan, Kriegman a Ahuja [14] rozdělili metody do čtyř tříd, nicméně některé z těchto metod mohou spadat do různých tříd současně:

Knowledge-based metody Nebo také metody shora-dolů. Jsou založeny na znalosti hledaného objektu, kdy objekt popisujeme pomocí jednoduchých pravidel. Například obličej můžeme reprezentovat pozicí a vzdáleností úst, očí a nosu.
Problémem těchto metod je definice pravidel. Pokud jsou příliš obecné, tak dostáváme mnoho false positive detekcí. Naopak detailní pravidla způsobí velké množství false negative detekcí.

Feature invariant metody Metody zdola-nahoru. Hledají se takové vlastnosti/části objektu, které jsou nezávislé na změně pozice, světla nebo natočení.

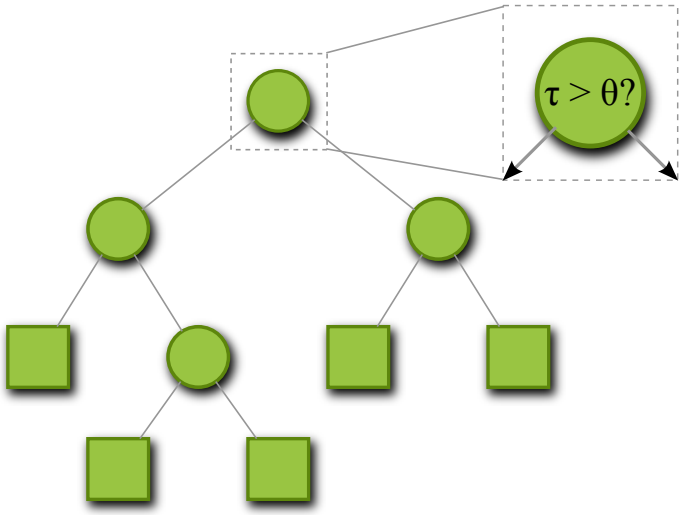
Template matching metody V těchto metodách je objekt reprezentován jako model, který popisuje tvar, barvu. V obraze se poté hledá oblast, která odpovídá danému modelu. Jednotlivé části objektu (oči, nos ústa) mohou být popsány zvlášť.
Tyto metody se obtížně vypořádávají s odlišným měřítkem, natočením nebo pozicí objektu.

Appearance-based metody Model objektu je vytvořen z velkého množství trénovacích obrazů, které postihnou různé změny při výskytu objektu.

2.1 Rozhodovací stromy

Strom je souvislý graf, který neobsahuje kružnici. Skládá se z hierarchické struktury uzlů a hran viz. obrázek (2.1). Uzly se dělí na vnitřní a koncové (listy stromu). Každý vnitřní uzel má jednu vstupní a několik výstupních hran. V tomto textu se omezím pouze na binární rozhodovací stromy, které mají pro každý vnitřní uzel vždy dvě výstupní hrany. U binárního rozhodovacího stromu obsahuje každý vnitřní uzel binární funkci, která určuje, zda pokračovat levým nebo pravým podstromem.

Vstupními daty stromu je vektor příznaků $v = (x1, x2, x3, \ldots, x_d) \in \mathbb{R}^d$, kde typ příznaku se odvíjí od vlastní aplikace a může to být například odezva na banku filtrů v daném bodě obrazu nebo přímo oblast obrazu.



Obrázek 2.1: Binární rozhodovací strom

Každý vnitřní uzel j obsahuje binární rozhodovací funkci, která na základě parametrů θ_j vyhodnotí vstupní data v a rozhodne, zda se pokračuje následujícím levým nebo pravým uzlem.

$$h(v, \theta_j) \in \{0, 1\} \tag{2.1}$$

Podle parametrů θ_j se vybírají vstupní vzorky z v a práh pro binární test.

Druh ohodnocení p v koncovém uzlu je závislý na daném použití. Může se jednat o:

- jednoduché "ano/ne", zda vektor v odpovídá/neodpovídá klasifikační třídě
- pravděpodobnost, že se jedná danou klasifikační třídu
- rozložení pravděpodobnosti pro více klasifikačních tříd

Fungování rozhodovacího stromu je rozděleno na dvě části:

Trénování Během procesu trénování se provádí rozdělování trénovacích dat na podmnožiny, které odpovídají větvení stromu. Dělení je prováděno na základě rozhodovací funkce, kterou se snažíme pro každý uzel optimalizovat pro co nejlepší rozdělení trénovací sady. Trénování stromu končí podle zvolených omezující podmínek, jako je hloubka stromu. [7]

Testování Vstupem stromu jsou neznámá data v. Průchod stromem začíná od počátečního uzlu a postupným vyhodnocováním vnitřních uzlů se získá ohodnocení p z koncového uzlu odpovídající tomuto vstupu.

2.2 Boosted trees

Jedná se o metody používané při trénování rozhodovacích stromů. Tyto algoritmy řeší nevýhody při použití stromů jako je:

Nestabilita Při malé změně trénovacích dat dojde ke značné změně struktury stromu.

Komplexnost Učící algoritmy vytváří velké stromy s mnoha větvením.

Overfitting Souvisí s předchozí nevýhodou. Komplexní stromy příliš odpovídají trénovací sadě a nedokáží dobře generalizovat.

2.2.1 Bagging

Bagging (Boostrap aggregating) poprvé představil Breiman [5]. Jedná se o metodu pro generování několika verzí rozhodovacích stromů z jedné trénovací sady.

Z trénovací sady se náhodně vybírá množina k vzorků, přičemž vzorky se v množinách mohou opakovat. Na každé množině je natrénován rozhodovací strom. Tím se dosáhne toho, že každý strom je náhodně odlišný, protože byla použita jiná trénovací sada.

Celkový výstup při detekci je pak dán průměrováním ohodnocení každého stromu (při regresi) nebo hlasováním každého stromu (klasifikaci).

Tato metoda je výhodná pro nestabilní učící metody, kde malá změna trénovací sady velmi ovlivní výsledný klasifikátor.

2.2.2 Random forests

Random forests jsou učící metody pro klasifikaci a regresi pomocí rozhodovacích stromů. Tuto metodu představil ve své práci Breiman [6], který vycházel z výzkumu náhodných rozhodovacích stromů Amita a Gemana[2].

Klasifikátory založené na metodách Random forests obsahují soubor náhodně trénovaných binárních rozhodovacích stromů, tzv. náhodný les.

Myšlenkou Random forest je natrénovat každý strom nezávisle na ostatních tak, že se budou navzájem náhodně lišit. K tomu je použita metoda Bagging (viz. kapitola 2.2.1), která umožní vytvořit více stromů ze stejné trénovací sady. Velikost náhodné podmnožiny vstupů má vliv na korelaci mezi jednotlivými stromy. Čím větší je vybraná podmnožina, tím je větší korelace mezi stromy (natrénované stromy jsou více identické). To vede k nízké obecnosti klasifikátoru a větší chybě při klasifikaci. Naopak pro malé podmnožiny je potřeba natrénovat více stromů.[7]

U Random Forest se navíc provádí náhodný výběr parametrů θ_j v každém uzlu trénovaného stromu, což dále zvýší korelaci mezi jednotlivými stromy.

Klasifikace pro vstup v je provedena pro všechny stromy T v náhodném lesu. Výstupní ohodnocení ze všech stromů se poté kombinují pro získání výsledného ohodnocení. Nejčastější metodou je jednoduché zprůměrování výsledků dané vztahem:

$$p(v) = \frac{1}{T} \sum_{t=1}^{T} p_t(v) \tag{2.2}$$

$p(v)$ je výsledné ohodnocení lesu, $l_t(v)$ je výstupní ohodnocení t-ého stromu. Alternativou může být násobení všech ohodnocení. [7]

2.2.3 Boosting

jedná se o algoritmy pro učící metody, kde výsledkem je soubor tzv. *slabých klasifikátorů*. Při detekci je vyhodnocován každý jednotlivý klasifikátor a jejich výsledky jsou kombinovány do celkového ohodnocení.

V trénovací sadě má každý vzorek přiřazenou váhu. Trénování se provádí v iteracích, kde výsledkem po každé iteraci je slabý klasifikátor, se kterým se vyhodnotí trénovací sada. Poté se podle chybně/správně detekovaných vzorků upraví jejich váha. Nejznámějším algoritmem z této třídy je Adaboost, který představil Freund a Schapire [9].

2.3 Detekce posuvným okénkem

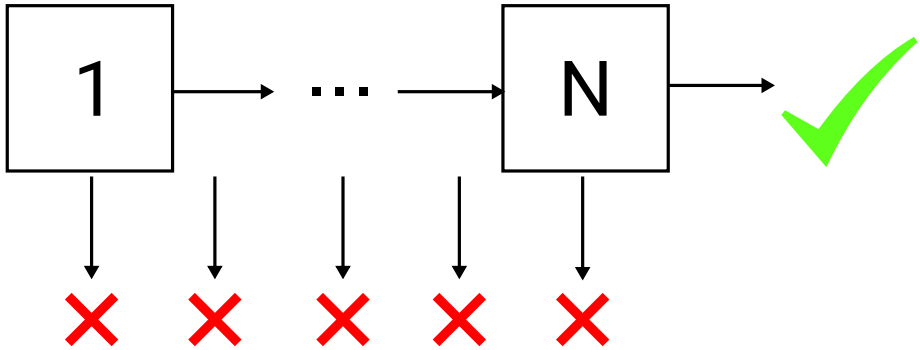
Jde o techniku pro identifikaci a lokalizaci objektů v obraze. Nejprve je natrénován klasifikátor na datasetu s obrazy o velikosti $m \times n$. Při detekci následně procházíme obrazem po pixelu nebo se zvoleným krokem a pro každou tuto pozici vyhodnotíme posuvné (detekční) okénko s velikostí $m \times n$ klasifikátorem. Posuvné okénka označené jako pozitivní obsahují hledaný objekt, naopak označené jako negativní objekt neobsahují.

Jedním problémem přístupu je, že s fixní velikostí detekčního okénka lze detekovat pouze objekty o určité velikosti. Toto lze vyřešit vytvořením pyramidy obrazů [3], při které se generují obrazy s menším rozlišením. Následně se provádí detekce nad každým zmenšeným obrazem zvlášť. [8]

Při detekci je pro jeden objekt několik překrývajících se detekčních okének označeno jako pozitivní. Tzn. že objekt je detekován několikrát. K vybrání pouze jedné detekce se použije metoda *non-maxima suppression*, která ponechá pouze okénko s nejvyšším ohodnocením. [8]

2.4 Kaskáda klasifikátorů

Tuto techniku poprvé použil Viola a Jones [21][22]. Jedná se o detektor, který obsahuje více klasifikátorů seřazených hierarchicky za sebou v kaskádě, jak je vidět na obrázku 2.2.



Obrázek 2.2: Schéma kaskády.

Vstupem jsou detekční okénka nad daným obrazem. Detekce probíhá od prvního stupně kaskády, kdy se vyhodnotí jeho klasifikátor a rozhodne zda jde o pozitivní nebo negativní detekci. V případě pozitivní se pokračuje stejným způsobem následujícím stupněm. Průchodem všemi stupni a dosažením konec kaskády značí, že jde o detekovaný objekt. Pokud je kterýkoli stupeň vyhodnocen negativně, je detekce detekce daného okénka ukončena.

U tohoto typu detektoru se využívá toho, že drtivá většina detekčních okének neobsahuje hledaný objekt. Z toho důvodu mohou být v prvních stupních kaskády jednodušší klasifikátory, které velmi brzo zamítnou většinu negativních detekcí. Složitější klasifikátory, které přesněji detekují objekt, jsou v pozdějších stupních kaskády.

Vytvoření takového detektoru zahrnuje vyřešení několika kompromisů:

- počet stupňů kaskády
- složitost klasifikátorů ve stupních
- hodnota prahu každého stupně pro rozhodnutí, zda jde o pozitivní nebo negativní detekci

Viola a Jones [22] použili pro stupeň kaskády klasifikátor s Haarovými příznaky natrénovaný metodou Adaboost. Detektor obsahuje 38 stupňů s celkově 6060 příznaky. První stupeň obsahuje pouze 2 příznaky a zamítne již 50% negativních detekcí. Další stupně obsahují klasifikátory s více příznaky pro přesnější detekci.

2.4.1 Kaskádovaný Random Forest

V této práci je pro detekci použita kaskády s Random Forest klasifikátorem. Každý stupeň obsahuje Random Forest klasifikátor, který se skládá ze souboru rozhodovacích stromů, jak je popsáno v kapitole 2.2.2. Vstupní data jsou vyhodnocována pro jednotlivé stromy v každé kaskádě. Tím dojde k rozšíření původního vztahu 2.2 na:

$$p(v) = \sum_{s=1}^{S} \frac{1}{T} \sum_{t=1}^{T} p_t(v) \tag{2.3}$$

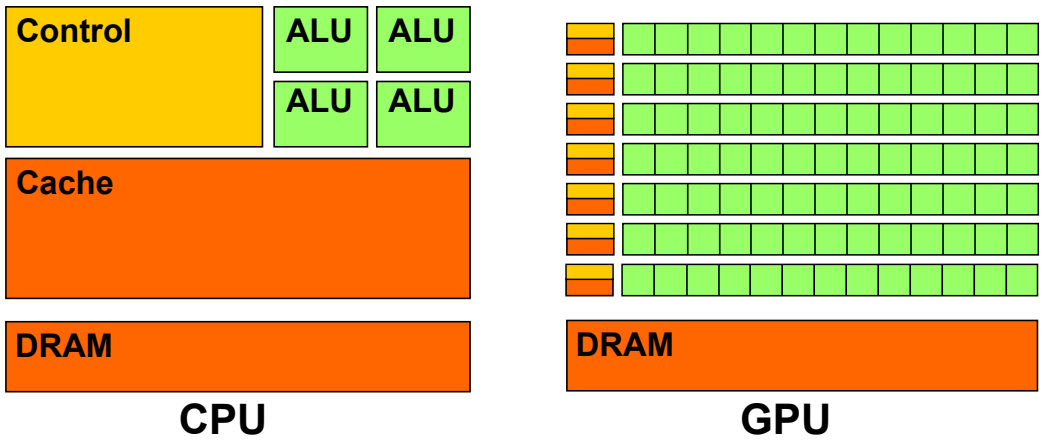
Tento detektor použil Baumann [4] pro detekci aut a obličejů v obraze. Kaskáda obsahovala 400 stupňů, každý s jedním rozhodovacím stromem. Pro vyhodnocení kaskády použil několik variant:

- každému stupni je přiřazena váha vzhledem k jeho přesnosti detekce
- pro pozitivní detekci musí být provedeny všechny stupně, jak je popsáno výše
- mezi jednotlivé stupně je vložena další kontrola, která se provádí pro všechny předešlé stupně, pokud jí většina stupňů projde, pokračuje se dále

Kapitola 3

Akcelerace na GPU

Z důvodu zpracování obrazu, 3D grafiky se GPU (grafický procesor, Graphics Processing Unit) vyvinul do procesoru s mnoha výpočetními jádry, vysokým výkonem a velkou paměťovou propustností. GPU je na rozdíl od CPU uzpůsobeno k paralelním výpočtům, což je dáno rozdílnou architekturou. Na obrázku (3.1) porovnání obou architektur, kde GPU je navrženo spíše pro zpracování mnoha dat, než pro řízení toku a uchování dat ve vyrovnávací paměti.



Obrázek 3.1: Architektura CPU a GPU

GPU je optimalizováno pro paralelní zpracování dat, kdy stejný program se provádí na mnoha různých datech zároveň. Z toho důvodu nepotřebuje složitou řídící jednotku a přístup do paměti může být skryt během výpočtu, proto nepotřebuje velké vyrovnávací paměti.

Příchod programovatelných shaderů a implementace čísel s pohyblivou řádovou čárkou umožnilo využít GPU pro obecné výpočty (GPGPU, General-purpose computing on graphics processing units). Aplikační rozhraní využívající shadery (OpenGL, DirectX) byly určeny pro zpracování počítačové grafiky. Z toho důvodu bylo potřeba převést výpočetní problém do grafických primitiv. Dalším krokem bylo vytvoření aplikačních rozhraní, která umožňují obecné výpočty na grafickém procesoru. Mezi nejznámější patří OpenCL (Khronos Group), CUDA (Nvidia) a DirectCompute (Microsoft).

3.1 OpenCL

OpenCL (Open Computing Language) je standard pro programování heterogenních systémů, který umožňuje využít výpočetního výkonu grafického procesoru nebo jiných koprocesorů k obecným výpočtům. Program vytvořený podle specifikace OpenCL je přenosný, nezávislý na značce a zařízení a je možné ho použít na mnoha různých hardwarových platformách.

Aplikační rozhraní používá jazyk C s možností rozšíření pro C++. Podpora dalších jazyků (Java, Python, .NET a další) je tvořena rozhraními od třetích stran. Programovací jazyk pro OpenCL zařízení (OpenCL C) je odvozen od jazyka C99 s rozšířeními pro paralelní programování.

3.1.1 Architektura OpenCL

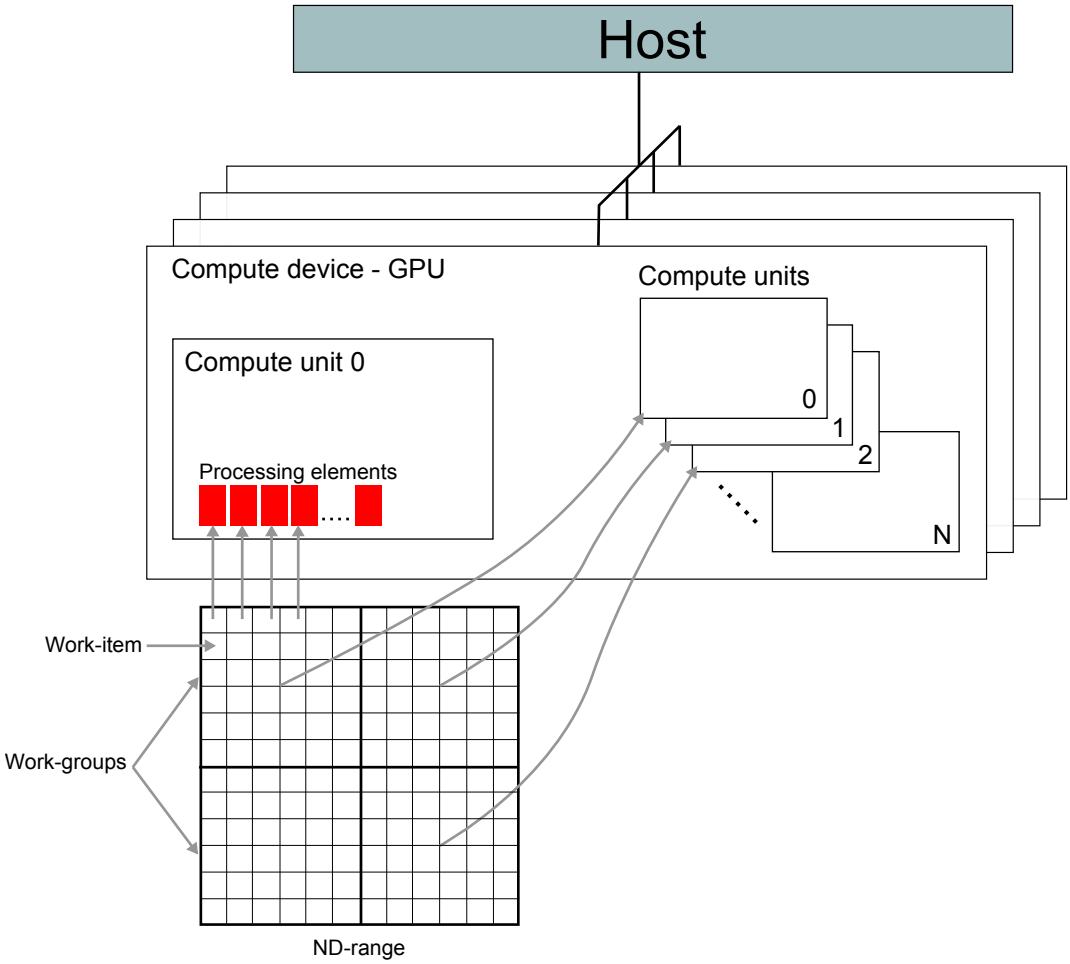
Na obrázku (3.2) je zobrazena architektura OpenCL. Skládá se z jednoho hostitelského systému (*host*), což je standardní operační systém běžící na CPU. K tomuto systému je připojeno jedno nebo více výpočetních zařízení (*compute devices*) jako GPU nebo DSP. Výpočetní zařízení obsahuje několik výpočetních jednotek (*compute units*, výpočetní jádro) a každá z těchto jednotek se dál skládá z prvků pro zpracování dat (*processing elements*). Tyto prvky pracují jako SIMD (Single Instruction, Multiple Data - prvky zpracují data synchronně) nebo SPMD (Single Program, Multiple Data - prvky zpracují data nezávisle na sobě). [13]

3.1.2 Programovací model

OpenCL aplikace má dvě části, program pro hostitelský systém a *kernel* pro výpočetní zařízení. OpenCL API na hostitelském systému umožňuje správu provádění OpenCL příkazů, jako přesuny dat mezi hostem a OpenCL paměťovými strukturami nebo nastavení výpočetního prostředí pro kernel.

Kernel je část programu, která je napsána v jazyce OpenCL C a provádí se na výpočetním zařízení. Pro paralelní programování se v OpenCL používá N-rozměrný indexovací prostor (*NDRange*), který je na obrázku (3.2). Tento indexovací prostor může být 1,2 nebo 3-rozměrný. Nezávislá jednotka, která vykonává jednu instanci kernelu se nazývá *work-item*. Každá work-item provádí stejný kernel, ale na jiných datech. Work-itemy mohou být seskupovány do tzv. *work-group*. Výsledná velikost celého indexovacího prostoru musí být násobek velikosti work-group. Všechny work-itemy ve work-group jsou prováděny na stejném výpočetním zařízení, což umožňuje sdílet společnou lokální paměť a provádět synchronizaci výpočtu. Synchronizace je možná pouze mezi work-itemy ve stejné work-group.[10][13]

GPU NVIDIA architektury Fermi obsahuje několik multiprocesorů (*Streaming multiprocesor, SM*), které v OpenCL odpovídají výpočetním jednotkám (compute units). Každý multiprocesor má 32 výpočetních jader (v OpenCL jsou to prvky pro zpracování dat), dále pak plánovací jednotku a sdílenou paměť[15]. Každému multiprocesoru je přiřazena work-group, ve které se sdílí lokální paměť a je možná synchronizace. Ve výpočetních jádrech jsou prováděny work-itemy, jak je naznačeno na obrázku (3.2). Multiprocesor vykonává skupinu 32 work-item (*warp*) paralelně. Z důvodu architektury SIMD je vhodné v kernelu omezit větvení programu, protože při provádění větví s různou výpočetní náročností musí některé work-itemy z warpu čekat na dokončení.



Obrázek 3.2: Architektura OpenCL. K jednomu *host* je připojeno jedno nebo více výpočetních zařízení (*compute device*), která obsahují výpočetní jednotky (*compute units*) s prvky zpracovávající data (*processing elements*).

3.1.3 Paměťový model

OpenCL definuje abstraktní paměťový model, na který výrobci mapují paměťové moduly hardwaru. Model se skládá z několika paměťových oblastí, které se liší velikostí a rychlostí přístupu.

Privátní paměť je malá oblast paměti přístupná pouze pro jednotlivé work-itemy. Každá work-item má svou vlastní privátní paměť, kde se mohou definovat proměnné přístupné pouze pro tuto work-item. V praxi je tato paměť mapována na registry výpočetní jednotky a má nejrychlejší přístup při čtení a zápisu. [10]

Další úrovní je *lokální paměť*, která je sdílená pro celou work-group. Všechny work-item ze stejné work-group sdílejí lokální paměť a mohou číst/zapisovat data. U GPU (AMD, NVIDIA) je lokální paměť rozdělena na stejně velké části, tzv. *banky*, ke kterým lze přistupovat souběžně. Maximální šířka pásma je využita v případě, že se přistupuje zároveň ke všem bankám. Pokud se vyskytne více požadavků pro čtení ze stejné banky a stejné adresy, jsou data poskytnuta všem bez paměťových konfliktů. V případě požadavků pro čtení ze stejné banky, ale rozdílné adresy v bance, dochází k paměťovým konfliktům a požadavky

jsou vyřízeny sekvenčně a tím dojde ke snížení propustnosti paměti. Počet bank a jejich velikost je rozdílná podle výrobce a verze GPU. [1][16]

Globální paměť je nejpomalejší ze všech nabízených paměťových prostorů. Je viditelná pro všechny work-itemy ze všech work-group. Data přenášená z hostitelského systému na výpočetní zařízení jsou uložena v globální paměti. Stejně tak data z výpočetního zařízení musí být nejprve uložena do globální paměti a po dokončení výpočtu mohou být načtena na hostitelském systému. Jedná se tedy o jedinou možnost jak komunikovat mezi hostem a zařízením. Jelikož má globální paměť mnohem vyšší latenci než provádění instrukce na GPU a zároveň má omezenou šířku pásma, je potřeba co nejefektivněji využívat tyto paměťové operace. Když work-itemy z jednoho warpu požadují data z globální paměti a požadované adresy jsou zarovnány za sebou, tak se neprovede každý požadavek zvlášť, ale požadavky se sloučí do jednoho a přenese se pouze blok paměti, který obsahuje všechny potřebná data. Toto chování souvisí se vhodným návrhem kernelů a indexovacího prostoru. [10][16]

Konstantní paměť je část globální paměti pro uložení dat, která se v průběhu výpočtu nemění (konstanty) a je určena pouze pro čtení. Tato paměť na rozdíl od globální paměti používá vyrovnávací paměť.

OpenCL umožňuje využít texturovací paměť na GPU ke čtení/zápisu z/do vícerozměrných polí uložených v globální paměti. Jedná se o globální paměť s vyrovnávací pamětí. Pole je pak interpretováno jako textura, což přináší některé výhody:

- Výpočet adresy neprovádí work-item, ale je provedena vyhrazeným hardwarem výpočetní jednotky.
- Je vhodné ji použít při nezarovnaném přístupu do globální paměti, kdy se čte z různých míst pole.

Tato paměť je vhodná pro uložení obrázků, ale jen v případě, kdy provádíme operace s náhodnými pixely. [1][16]

Kapitola 4

Rozbor GPU detektorů

V této kapitole jsou popsány implementace GPU detektorů používající kaskádu klasifikátorů, rozhodovací stromy a posuvné okénko. Popsané GPU detektory obsahují ve stupni kaskády jiné klasifikátory než rozhodovací stromy, nicméně samotný detektor pracuje stejně až na vyhodnocení samotného klasifikátoru.

Sharma aj.[18] představili GPU implementaci kaskádového klasifikátoru s posuvným okénkem. Používá zde Haarovy příznaky a integrální obraz dle Viola a Jonese [22]. Při detekci zde použil pyramidu obrazů pro detekování libovolně velkých obličejů. Celá pyramida i kaskáda je uložena v texturovací jednotce, přičemž kaskáda je při výpočtu nejprve nahrána do sdílené paměti skupiny vláken. Každé vlákno zde vyhodnocuje kaskádu pro jednu pozici posuvného okénka.

Oro aj.[17] stejně jako předchozí implementovali GPU detektor na základě [22]. Na rozdíl od Sharma aj.[18] použili pro uložení Haarových příznaků konstantní paměť, protože během výpočtu jsou příznaky neměnné a dochází tedy pouze ke čtení z paměti.

Herout aj.[11] představili GPU detector založený na metodě Waldboost [23] s LRP příznaky [12]. Stejně jako předchozí zde použili pyramidu obrazů, která je uložena v texturovací jednotce GPU. Neměnná kaskáda klasifikátorů je uložena v konstatní paměti. Popsali zde problém nízkého obsazení výpočetních jednotek při vyhodnocování kaskády a jeho řešení pomocí přeskupení aktivních vláken během výpočtu.

Sharp[19] vytvořil GPU implementaci Random Forest detektoru v Direct3D s použitím textur a programovatelných shaderů. Ukázal zde způsob uložení rozhodovacích stromů v paměti s jejich vyhodnocením bez použití větvení programu.

Kapitola 5

Implementace

Implementace detektoru je provedena v C++ s použitím knihovny OpenCV pro načtení obrazu a videa. Pro akceleraci na GPU je použita knihovna OpenCL s aplikačním rozhraním v jazyce C. Implementace je provedena pro detektor poskytnutý od pana Ing. Romana Juránka, Ph.D. Detektor používá kaskádu klasifikátorů (viz. 2.4) a je natrénován pro detekci automobilů. V této kapitole se zaměřím na návrh a implementaci Random Forest na GPU. V podkapitole 5.1 se zaměřím na reprezentaci rozhodovacího stromu v paměti. Ve 5.2 popíši použitý detektor, jeho vstupní data a průběh detekce. V poslední podkapitole 5.6 popíši návrh implementace na GPU s využitím OpenCL.

5.1 Reprezentace rozhodovacího stromu v paměti

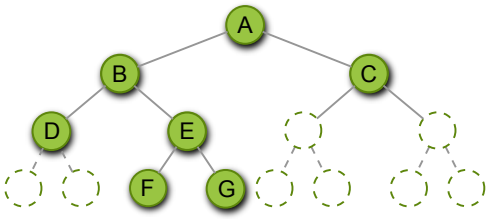
Klasická implementace stromové struktury v počítačovém programu je pomocí dynamické struktury, kde jsou hrany reprezentovány ukazateli do paměti. Každý uzel stromu obsahuje ukazatele na svoje následníky, případně ukazatele na sousední uzly stejné úrovně nebo rodičovský uzel. Takováto struktura je vhodná při trénování detektoru, kdy přidáváme nové uzly. Stačí vytvořit nový uzel a správně propojit ukazatele. Naopak při testování chceme mít následující uzly v blízkém okolí, z důvodu použití vyrovnávací paměti[20]. Proto se používá lineární pole pro uložení stromové struktury.

Nejjednodušší možností je uložení do lineárního pole, kdy každý prvek reprezentuje jeden uzel a prvním prvkem je kořenový uzel, jak je zobrazeno na obrázku (5.1 a, b). Toto uspořádání umožňuje kompaktní uložení bez nutnosti přidávat informace o okolních uzlech. Pro úplný strom, kdy všechny koncové uzly mají stejnou hloubku, je tento přístup paměťově efektivní. Každý uzel je uložen do jednoho prvku pole. V případě neúplného stromu, který je na obrázku (5.1 a), dojde k vytvoření prázdných oblastí v poli. Proto tato metoda ukládání není vhodná pro neúplné stromy.

Vylepšením této metody je přidání indexu(posunutí) prvního následníka. Pokud uvažujeme, že pravý a levý následník jsou řazeny za sebou, můžeme použít pouze index(posunutí) jednoho uzlu a druhý je spočítán přičtením jedničky. Tato struktura je na obrázku (5.1 c).

V předchozích dvou přístupech se neřešily rozdílná data ve vnitřních a koncových uzlech. Vnitřní uzly obsahují parametry pro rozhodnutí, kterým následujícím uzlem pokračovat, koncové uzly uchovávají výsledné ohodnocení. Pokud je použita jedna datová struktura reprezentující oba uzly, pak všechny uzly obsahují nepotřebná data navíc. Tento problém lze vyřešit rozdělením vnitřních a koncových uzlů do dvou polí, jak je zobrazeno na obrázku (5.1 d). K vnitřnímu uzlu jsou přidány indexy jeho následovníků s příznakem, zda se jedná

a)



b)

Adresa	List/Uzel
0	A
1	B
2	C
3	D
4	E
5	---
6	---
7	---
8	F
9	G
10	---
11	---
12	---
13	---
14	---

c)

Adresa	List/Uzel	1. Následník
0	A	1
1	B	3
2	C	NULL
3	D	NULL
4	E	5
5	F	NULL
6	G	NULL

d)

Adresa U	List/Uzel	Následníci
0	A	U1, L1
1	B	U2, L1
2	E	L2, L3

Adresa L	List/Uzel
0	C
1	D
2	F
3	G

Obrázek 5.1: Uložení binární stromové struktury v paměti. a)Příklad binárního rozhodovacího stromu. b)Nejjednodušší uložení stromu v poli. c)Efektivní uložení v případě neúplného stromu d) Rozdělení vnitřních uzlů a koncových uzlů do dvou polí. [20]

o vnitřní nebo koncový uzel. [20]

OpenCL neumožňuje použít dynamické datové struktury v globální paměti, proto je potřeba některý z výše zmíněných přístupů.

5.2 Detektor

Jak bylo zmíněno na začátku kapitoly, jedná se o detektor s kaskádou klasifikátorů. V každém stupni kaskády je soubor binárních rozhodovacích stromů (1–N), kdy každý strom může mít různou hloubku a může být neúplný. Jako vstupní obrazové data pro detektor je použita oblast daná posuvným okénkem. Část obrazu daná posuvným okénkem je vstup v pro rozhodovací funkci stromu ze vztahu (2.1).

V každém stupni jsou vyhodnoceny všechny stromy a výsledky jednotlivých stromů jsou zprůměrovány dle (2.2). Výsledek daného stupně kaskády je přičten k akumulovanému hodnocení z předešlých stupňů a porovná se s prahem daného stupně. Pokud je hodnocení vyšší než práh (zatím se jedná o detekovaný objekt), pokračuje se dalším stupněm kaskády, jinak je detekce v daném okénku ukončena.

Jako rozhodovací funkce vnitřního uzlu stromu je použito jednoduché odečtení dvou pixelů náležících posuvnému okénku a porovnání s natrénovaným prahem:

$$h(v, \theta_j) = \begin{cases} 1 & pokud\,(v[\theta_1] - v[\theta_2]) > \theta_3 \\ 0 & jinak \end{cases} \tag{5.1}$$

Kaskáda stromů pro detekci je uložena v CSV souborech. Data pro rozhodovací stromy jsou rozdělena do několika souborů, které mají stejnou strukturu, kde každý řádek odpovídá jednomu stromu a sloupec je uzel daného stromu. Stupně kaskády jsou poté specifikovány indexem počátečního stromu (tj. číslo řádku), indexem koncového stromu a prahem daného stupně.

Při spuštění programu jsou CSV soubory parsovány do datových struktur. Datová struktura uzlu je na obrázku (5.2), obsahuje parametry pro vnitřní uzel s rozhodovací funkcí, výsledné ohodnocení koncového uzlu a index prvního následníka daného uzlu. Celý uzel má velikost 16B (mocnina dvou), což zajišťuje bezproblémovou interpretaci v paměti GPU. Položky `z1` a `z2` jsou zde z důvodu případného použití více vrstev obrazu.

```
typedef struct Node{
float prob;     // pokud se jedná o list, je zde výsledné hodnocení
float thr;      // práh pro rozhodovací funkci
short children; // levý následník uzlu, -1 pokud je to list
uchar y1;       // řádek prvního bodu
uchar x1;       // sloupec prvního bodu
uchar y2;       // řádek druhého bodu
uchar x2;       // sloupec druhého bodu
uchar z1;       // vrstva obrazu prvního bodu
uchar z2;       // vrstva obrazu druhého bodu
} Node;
```

Rozhodovací strom je tvořen polem uzlů, které má pevnou velikost danou stromem s nejvyšším počtem uzlů.

Datová struktura pro specifikaci stupně kaskády je zobrazena v (5.2). Stejně jako předchozí, má velikost mocniny dvou a to 8B. Obsahuje počáteční, koncový strom stupně a práh pro testování akumulovaného hodnocení.

```
typedef struct Stage{
float thr;    // práh stupně
ushort start; // první strom - index do pole všech stromů
ushort end;   // poslední strom - index do pole všech stromů
} Stage;
```

5.3 Inicializace

Před samotným zpracováním obrazu a detekcí je provedena inicializace. Do paměti GPU jsou přeneseny všechny neměnné informace potřebné pro detekci. Pro zpracování obrazu je to maska pro konvoluci, pro detekci je to detektor, který je načten z CSV souborů do datových struktur popsaných v předchozí kapitole. Stromy jsou vloženy do globální paměti a informace o stupních kaskády do konstantní paměti GPU.

Obraz je načten knihovnou OpenCV jako obraz ve stupních šedi a hodnoty pixelů obrazu jsou normalizovány do rozsahu $[0 \dots 1]$. Obraz poté nahrán na GPU jako textura, ze které je vygenerována pyramida obrazů.

5.4 Předzpracování obrazu

První částí, která je potřeba provést je předzpracování obrazu pro detekci. Použitý detektor je natrénován pro obraz ve stupních šedi, který je předzpracován podle postupu níže.

1. Původní obraz je rozmazán konvolučním filtrem.
2. Je vytvořena pyramida obrazů, podle zadaného měřítka.
3. Pro každý obraz z pyramidy je udělán součet $M \cdot M$ pixelů, kde M je zvolená velikost. Tím se obraz zmenší M-krát.
4. Nakonec se pro každý obraz v pyramidě provede konvoluce stejným filtrem, jako v kroku jedna.

5.4.1 Konvoluce na GPU

Konvoluce je provedena jako první a poslední část přípravy obrazu. Jako konvoluční maska je zde použita $[0,25 \quad 0,5 \quad 0,25]$ a výsledný obraz je uložen jako typ *Image2D*. Uložení jako textura je výhodné, protože lze jednoduše provést jeho zmenšení při generování pyramidu obrazů.

Konvoluce je provedena separabilně, zvlášť pro řádky a sloupce. Na obrázku (5.2 b)) je znázorněno fungování řádkové konvoluce. Nejprve je do lokální paměti zkopírována hlavní část obrazu (zelená část v obrázku), která odpovídá velikosti work-group, následně je zkopírován přesah konvolučního jádra (modrá část v obrázku). Takto odpovídá každá work-item jednomu pixelu a provede na této pozici řádkovou konvoluci. Výsledek zapíše do globální paměti. Pro sloupcovou konvoluci je rozdíl pouze v tom, že je přesah konvolučního jádra ve vertikálním směru.

5.4.2 Pyramida obrazů na GPU

Každý zmenšený obraz se vytváří z původního zdrojového obrazu. OpenCL umožňuje číst obraz Image2D s normalizovanými souřadnicemi mezi $[0 \ldots 1]$, kdy výpočet bodu v obraze se provede bilineární interpolací. Souřadnice hodnoty z originálního obrazu, která bude odpovídat bodu ve zmenšeném obrazu, je počítána podle vztahů (5.2) a (5.3), kde x_S a y_S jsou souřadnice zdrojového obrazu, x_R a y_R jsou souřadnice výsledného zmenšeného obrazu a w_R a h_R je výška a šířka zmenšeného obrazu.

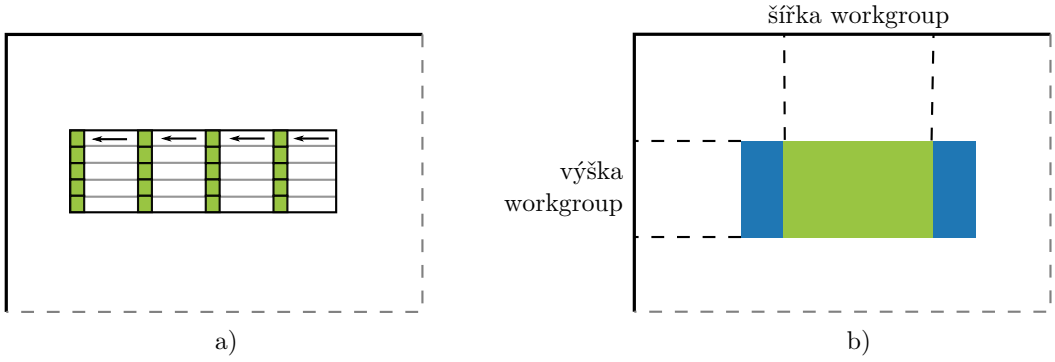
$$x_S = x_R / w_R \tag{5.2}$$
$$y_S = y_R / h_R \tag{5.3}$$

Měřítko pro každou zmenšeninu je spočítáno podle vztahu $(2^{\frac{1}{N}})^k$, kde k je k-tý obraz v pyramidě. Pyramida je generována, dokud je obraz větší než velikost posuvného okénka.

5.4.3 Součet bloku pixelů na GPU

Dalším krokem předzpracování obrazu je součet $M \cdot M$ pixelů. To je rozděleno do dvou součtů, nejprve po řádcích a poté následuje součet po sloupcích. Výpočetní prostor NDRange odpovídá velikosti výstupního obrazu z každé části, pak je tedy každá work-item bodem výstupního obrazu.

Pro součet v řádku je do lokální paměti nejprve zkopírována část vstupního obrazu odpovídající M-krát šířce work-group, protože každá work-item sčítá M pixelů. Součet po řádcích je na obrázku (5.2 a)). Součet po sloupcích je proveden obdobně, pouze se změní výška lokální paměti, která bude odpovídat M-krát výška work-group.



Obrázek 5.2: a) Součet pixelů v řádku na GPU. b) Řádková konvoluce, zelená část jsou hlavní data, modrá přesah konvolučního jádra.

5.5 Paměťová struktura

OpenCL a grafické procesory nabízejí několik druhů paměťových prostorů, které jsou optimalizované k různému použití a jejich výběr a správné využívání je jednou z nejdůležitějších částí návrhu. Pro detekci je potřeba vyřešit uložení obrazu a detektoru.

Obraz je uložen jako textura, jak pro předzpracování, tak pro detekci. Při předzpracování se s výhodou využije pro generování pyramidy obrazů, jak bylo popsáno v 5.4.2. Při detekci se pro vyhodnocení uzlu stromu přistupuje pouze ke dvěma bodům v obrazu. Pro tento účel je vhodná textura, která je optimalizována pro náhodný přístup do paměti.

Detektor se skládá ze dvou částí: posloupnost rozhodovacích stromů a popis každého stupně kaskády. Informace o každém stupni jsou neměnné a přistupují k nim všechny workitemy při výpočtu. Z toho důvodu jsou uloženy v konstantní paměti, která je určena pouze pro čtení a využívá vyrovnávací paměť.

Stejně jako předchozí i rozhodovací stromy jsou během výpočtu neměnné, nicméně použití konstantní paměti není možné. Použitý detektor obsahuje 2048 stupňů, kde v každém stupni je jeden strom se 7-mi uzly, to je celkově 230KB (uzel má velikost 16B). NVIDIA GPU mají velikost konstantní paměti 64KB, což nedostačuje. Proto jsou stromy uloženy v globální paměti, pro kterou je potřeba provádět zarovnaný přístup při čtení a zápisu.

5.6 Detekce na GPU

Detekce je provedena pro každou pozici posuvného okénka a je označeno jako pozitivní, pokud úspěšně projde celou kaskádou. V každém stupni jsou vyhodnoceny stromy, jejich ohodnocení je z průměrováno a přičteno k akumulovanému hodnocení. Výpočet je ukončen, pokud je akumulované hodnocení menší než práh stupně. Na obrázku (5.6) je diagram průběhu detekce. V levé části je znázorněn průběh hostitelského programu, v pravé části je průběh kernelu. Uprostřed diagramu jsou znázorněny použité typy pamětí pro uložení vstupních a výstupních dat.

Strom je v paměti reprezentován metodou (5.1c), takže lze provést výpočet následujícího uzlu ve stromu podle vztahu:

$$\text{leftChild} = \text{parentNode.children} \tag{5.4}$$

$$\text{rightChild} = \text{parentNode.children} + 1 \tag{5.5}$$

Algorithm 1: Algoritmus pro vyhodnocení pozice posuvného okénka

```
Data: pozice posuvného okénka
for každý stupeň kaskády do
    for každý strom v kaskádě do
        načti uzel stromu;
        while dokud není koncový uzel do
            vyhodnoť rozhodovací funkci uzlu;
            podle výsledku načti levý nebo pravý uzel;
            přičti výsledek hodnocení k výsledku stupně;
        end
    end
    zprůměruj výsledek stupně;
    přičti výsledek stupně k akumulovanému hodnocení;
    if akumulované hodnocení < práh stupně then
        ukonči výpočet pro danou pozici;
    end
end
```

Velikost indexovacího prostoru je zvolena následovně. Work-group je dvourozměrná o velikosti 32x8 a celkový indexovací prostor je spočítán podle vztahu 5.6, kde w_I a h_I je šířka a výška obrazu, w_W a h_W je šířka a výška posuvného okénka, w_N a h_N je šířka a výška indexovacího prostoru.

$$w_N = \lceil (w_I - w_W)/32 \rceil \cdot 32 \tag{5.6}$$

$$h_N = \lceil (h_I - h_w)/8 \rceil \cdot 8 \tag{5.7}$$

OpenCL vyžaduje, aby velikost celého indexovacího prostoru byla násobkem work-group, což tento vztah zaručí. Velikost indexovacího prostoru je ještě zmenšena o velikost posuvného okénka, protože posuv se neprovádí přes okraj obrazu, viz obrázek (5.3). Při takto zvolené velikosti výpočetního prostoru na počátku výpočtu odpovídá každá work-item jedné pozici posuvného okénka.

Pro detekci jsou vytvořeny čtyři verze implementací:

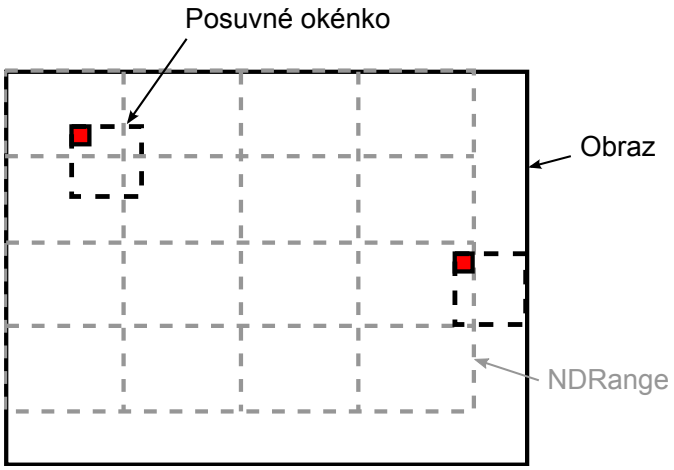
Globální paměť základní implementace, která čte kaskádu přímo z globální paměti

Lokální paměť část rozhodovacích stromů z kaskády je nejprve přesunuto do lokální paměti

Mipmapa pyramida obrazu je uložena jako mipmapa a zároveň je použita lokální paměť pro rozhodovací stromy

Přeskupení vláken vychází z předchozí implementace mipmapy a navíc je zde provedeno přeskupení vláken během výpočtu

V referenční implementaci se pracuje s kaskádou pouze v globální paměti. Každá work-item vyhodnocuje kaskádu pro svoji pozici v obraze. Strom se prochází tak, že si každá work-item zkopíruje aktuální uzel stromu do své privátní paměti a provede jeho vyhodnocení. Pokud se projdou všechny stupně kaskády, tak se jedná o pozitivní vzorek obrazu a výsledek se zapíše do globální paměti. Aby nemohlo dojít k přepisování výsledků jednotlivých work-item, je použita globální funkce *atomic_inc()*, která inkrementuje index do pole výsledků



Obrázek 5.3: Schéma indexovacího prostoru pro detekci. 2D výpočetní prostor, který odpovídá velikosti obrazu.

a zároveň udává jejich celkový počet. Výsledkem je pozice posuvného okénka, jeho velikost a výsledné ohodnocení.

Detekovanému objektu většinou odpovídá několik výsledných pozic s menším posunutím, proto je poté použita funkce z OpenCV `groupRectangles()`, která tyto pozice sloučí do jedné výsledné a navíc odstraní samostatné pozice, které jsou většinou falešnou pozitivní detekcí.

5.6.1 Lokální paměť

V referenční implementaci je strom čten přímo z globální paměti po uzlech, což sebou nese nevýhody při práci s pamětí. Každá work-item musí číst z globální paměti každý uzel, který potřebuje k výpočtu. Tím dochází k situaci, kdy work-itemy ze stejné work-group kopírují stejný uzel (čte se dvakrát stejná oblast paměti) nebo různé uzly (do paměti přistupují nezarovnaně). Obě tyto možnosti zvyšují počet paměťových požadavků do globální paměti a tím se snižuje propustnost paměti. U moderních GPU je vliv nezarovnaného přístupu nebo opakované čtení stejných dat snížen použitím vyrovnávacích pamětí L1/L2.

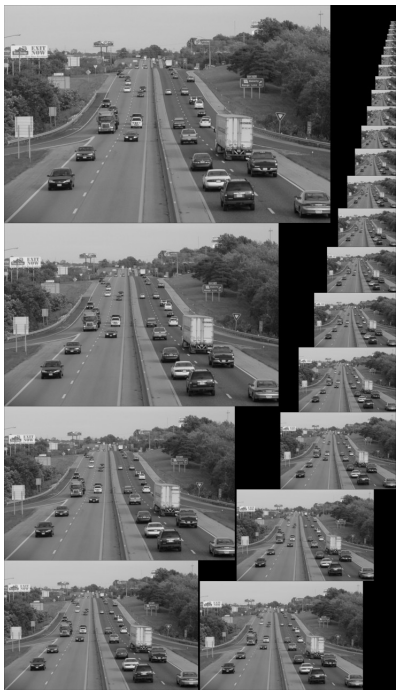
Oba tyto problémy lze vyřešit použitím lokální paměti, která je rychlejší než globální paměť, ale je sdílená pouze pro jednu work-group. Každá wokr-group si tedy nejprve přesune část kaskády do lokální paměti, se kterou poté pracuje při detekci. Jelikož se kopíruje blok paměti, je možné číst paměť zarovnaným způsobem, jak je popsáno v kapitole 3.1.3. Aby se předešlo bankovým konfliktům, čte se kaskáda z globální paměti jako pole `int`. Nevýhodou použití tohoto řešení je, že je nutné synchronizovat všechny work-itemy ve work-group po každém čtení z globální paměti, aby pracovaly s aktuálními rozhodovacími stromy.

5.6.2 Mipmapa

Pro detekci objektů v různém měřítku je vytvořena pyramida ze zpracovávaného obrazu. V referenční implementaci je na každém obrazu z pyramidy prováděna detekce zvlášť, z toho důvodu je nutné po každém obrazu ukončit výpočet, přesunout další obraz na grafickou kartu a znovu spustit detekci. U tohoto řešení se nevyužijí plně všechny výpočetní jednotky a s každým menším obrazem je tato situace horší. Navíc dochází ke zbytečné režii při

opakovaném spouštění a inicializaci detekce.

Výhodnějším řešením je vložit všechny obrazy z pyramidy do jednoho obrazu a vytvořit tak mipmapu pro daný snímek, jak je znázorněno na obrázku 5.4. To umožní spustit detekci pouze jednou nad všemi velikostmi obrazu. Mipmapa je navíc větší o prostor mezi jednotlivými obrazy, který je vyplněn konstantní barvou. Čas strávený detekcí této části obrazu je minimální, protože se výpočet ukončí již během prvních stupňů kaskády.



Obrázek 5.4: Pyramida obrazů je uložena v do jedné mipmapy nad kterou je provedena detekce.

5.6.3 Přeskupení vláken

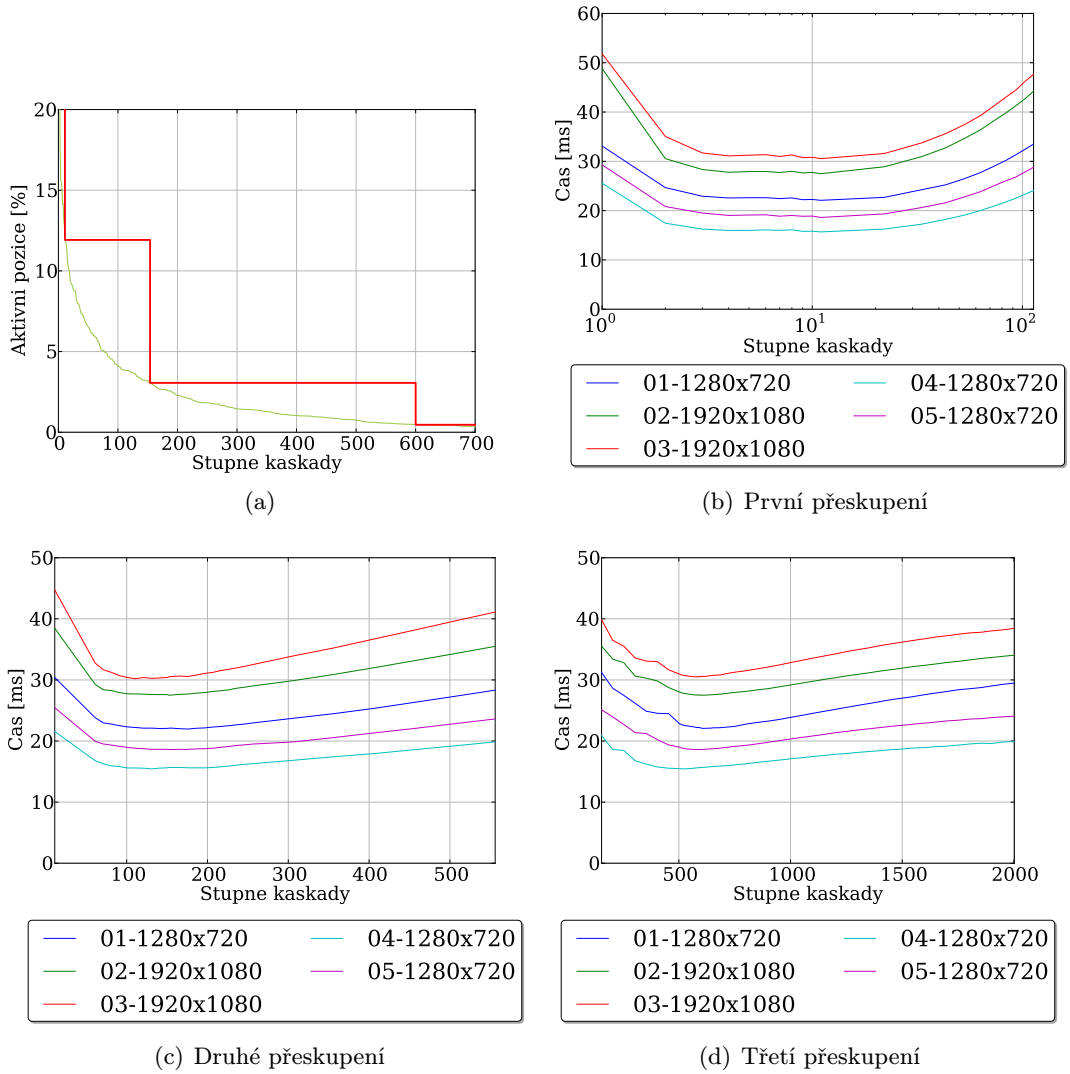
Moderní GPU pracují jako SIMD, kde skupina vláken (warp/wavefront) současně zpracovává stejné instrukce s různými daty. Na začátku detekce provádí všechny work-itemy stejný kód – načte se část kaskády do lokální paměti, vyhodnotí se stromy v kaskádě atd. V průběhu detekce však mnoho work-item ukončí svůj výpočet dříve s negativním hodnocením dané pozice. Tím dochází k větvení v prováděném programu, kdy ve warp/wavefront část work-item pokračuje ve vyhodnocování kaskády a zbylé pouze čekají na jejich dokončení.

Na grafu (5.5a) je zelenou křivkou zobrazeno, kolik procent aktivních pozic zbývá v jednotlivých stupňů kaskády. Lze vidět, že během několika prvních stupňů klesne jejich počet pod 20 % od počátku detekce, tzn. přes 80 % work-item neprovádí žádný užitečný výpočet. V dalších stupních se rychlost odpadávání aktivních pozic snižuje, až se ustálí na konečném počtu výsledných detekcí.

Počet a pozice přerušení v kaskádě závisí na natrénovaném klasifikátoru, protože je u každého rozdílná rychlost odpadávání negativních pozic v různých částech kaskády. Optimální pozice je možné zjistit experimentálně nebo použitím heuristiky pro jejich odhad.

Obecně je vhodné provést první přerušení již během začátku kaskády z důvodu vysokého počtu negativních rozhodnutí.

Pro použitý detektor byly experimentálně zjištěny přerušení v stupních kaskády 11, 154 a 600. Vyšší počet přerušení přinášel pouze minimální zrychlení. V grafu (5.5a) je červenou křivkou znázorněn počet spuštěných vláken pro detekci vzhledem k aktivním pozicím (zelená křivka) v každé stupni kaskády. V grafech (5.5b,c,d) je experimentální zjištění ideálního stupně pro přeskupení vláken.



(a) (b) První přeskupení

(c) Druhé přeskupení (d) Třetí přeskupení

Obrázek 5.5: a) Počet vláken (červená křivka) vzhledem k aktivním pozicím (zelená křivka). b)c)d) Experimentální zjištění ideálních stupňů pro přeskupení vláken na různých videích.

Tuto situaci lze vyřešit přeskupením výpočetního prostoru během detekce [11]. V průběhu detekce se provede ukončení výpočtu po zvoleném stupni kaskády, přepočítá se výpočetní prostor a detekce pokračuje od tohoto stupně. Počet zbývajících aktivních pozic a pozice samotné jsou zapsány do globální paměti a nový výpočetní prostor NDRange je spočítán podle vztahů 5.9, 5.10, kde S je počet aktivních pozic, $NDRange_X$ a $NDRange_Y$ je šířka a výška výpočetního prostoru. Tento postup se může několikrát opakovat po různých

částech kaskády.

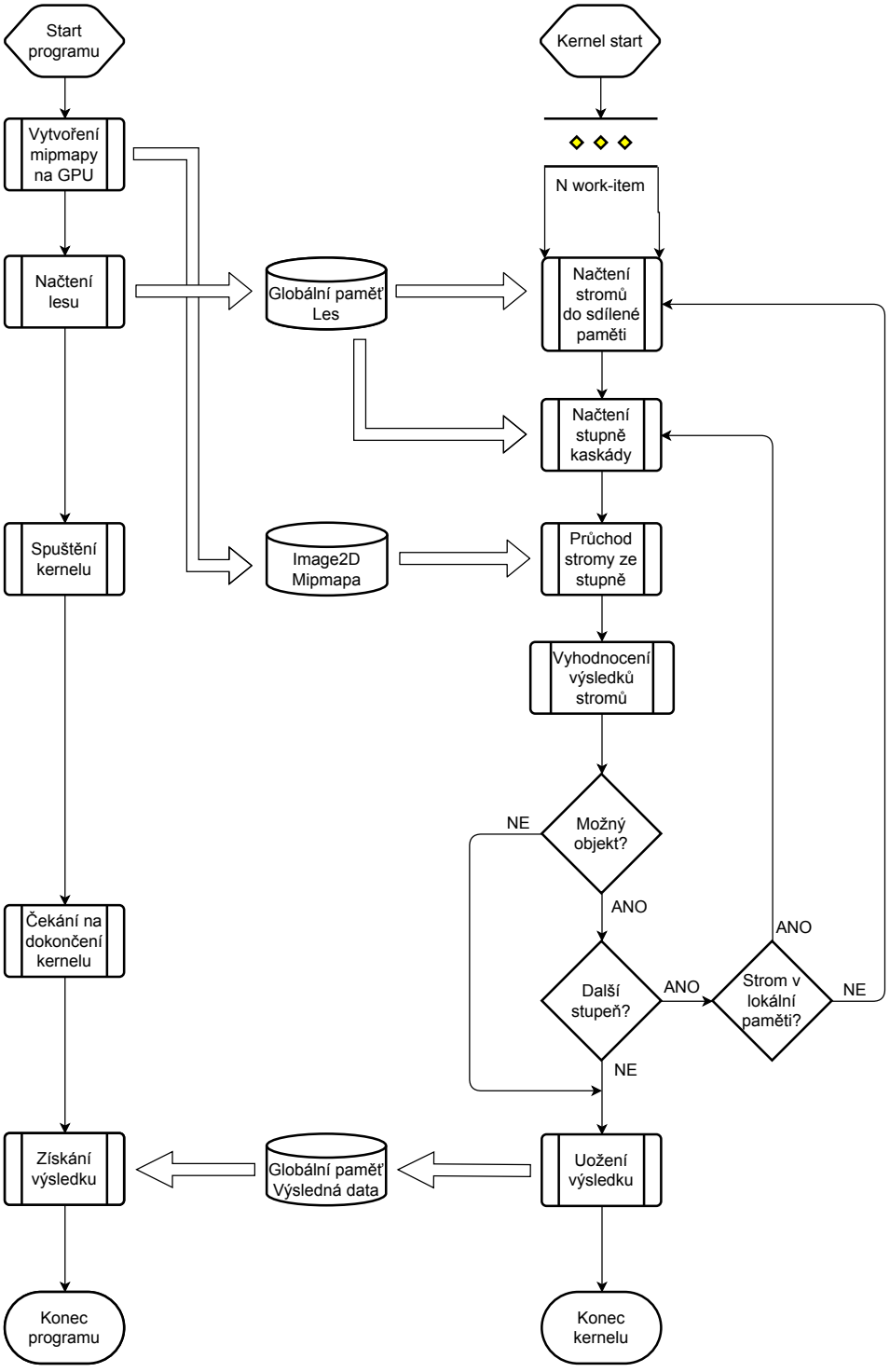
$$w_x = \sqrt{S} \tag{5.8}$$

$$NDRange_X = \left\lceil \frac{w_x}{32} \right\rceil \cdot 32 \tag{5.9}$$

$$NDRange_Y = \left\lceil \frac{\frac{S}{NDRange_X}}{8} \right\rceil \cdot 8 \tag{5.10}$$

Zbývající aktivní pozice jsou nejprve počítány samostatně v každé work-group pomocí sdílené proměnné. Ta je na začátku inicializována na počet work-item ve work-group a pro každé ukončení výpočtu s negativním hodnocením je dekrementována atomickou instrukcí `atomi_cdec()`. Po dosažení zvoleného stupně v kaskádě reprezentuje tato proměnná zbývající počet aktivních pozic. Přičtením toho lokálního počtu k celkovému globálnímu počtu aktivních pozic v globální paměti (`atomic_add()`) se navíc získá index do globálního pole zbývajících pozic, kde každá aktivní work-item zapíše svou pozici a akumulované hodnocení.

V digramu 5.6 je znázorněn průběh detekce pro implementaci s mipmapu, lokální pamětí a přeskupením vláken.



Obrázek 5.6: Diagram detekce objektů na GPU. Vlevo je část programu prováděná na hostitelském systému. Vpravo průběh kernelu.

Kapitola 6

Testování

V této kapitole je popsáno testování implementovaného programu z hlediska rychlosti a úspěšnosti detekce. V tabulce 6.1 je seznam GPU a CPU použitých pro testování výkonnosti implementace. Jsou zde zastoupeny GPU od různých výrobců, s různými architekturami a s různým výkonem.

GPU zařízení						
GPU	**Architektura**	**Výpočetní jednotky**	**Propustnost paměti [GB/s]**	**L2 cache [KB]**	**Sdílená paměť [KB]**	**Texturovací paměť [KB]**
AMD 6970	VLIW4	1536 (24 CU)	176	512	32	8
AMD 7970 GHZ	GCN	2048 (32 CU)	288	512	64	16
GeForce GT 630M	Fermi	96 (2 SM)	32	768	64	L2 + L1
GeForce GTX 580	Fermi	512 (16 SM)	192	768	64	L2 + L1
GeForce GTX 780	Kepler	2880 (15 SM)	336	1536	64	L2 + L1
GeForce GTX TITAN	Maxwell	3072 (24 SM)	336,5	3072	96+24	96+24

Hostitelský systém			
CPU	**Frekvence CPU [GHz]**	**RAM**	**Frekvence RAM [MHz]**
Intel Core i7-4790K	4	16GB DDR3	1866
Intel Core i7-3612QM	2,1	8GB DDR3	800

Tabulka 6.1: Seznam použitých GPU a CPU.

Testování rychlosti detekce bylo prováděno na pěti videích stažených ze serveru Youtube s velikostí 1280x720px a 1920x1080px.

Pro vyhodnocení úspěšnosti detekce byla ručně vytvořena sada 23 anotovaných obrázků ze zmíněných videí. Ke každému obrazu je vytvořen CSV soubor se seznamem objektů v obraze, které jsou určeny detekčním oknem s danou pozicí a velikostí.

6.1 Výkonnostní testy

Testování je provedeno na zmíněných pěti videích. Na každém GPU je provedena detekce nad každým videem se všemi implementacemi (s globální pamětí, lokální pamětí a přeskupením vláken). Parametry detekce byly zvoleny podle natrénovaného detektoru:

- velikost posuvného okénka 20x20px
- shrinkFactor 4, pak minimální detekovaný objekt je 80x80px
- scaleFactor 4
- konvoluční jádro je $[0,25 \quad 0,5 \quad 0,25]$
- přeskupení vláken ve stupních kaskády 11, 154, 600

Výsledkem je doba zpracování v jednotlivých fázích detekce:

- předzpracování obrazu, rozdělené na zmenšení obrazu, agregaci obrazu a konvoluci
- detekce na GPU

6.1.1 Lokální vs globální paměť

V tabulce 6.2 je srovnání použití globální a lokální paměti při detekci. Použití lokální paměti zrychlí detekci u většiny architektur. Největší zlepšení je u architektur používající pouze L2 cache při přístupu do globální paměti (GTX TITAN X, GTX 780 Ti). U GPU využívající L1/L2 cache (GTX 580, GT 630M) není potřeba tak často číst přímo z globální paměti a tím je zrychlení nižší.

GPU	Globální paměť [ms]	Lokální paměť [ms]	Zrychlení
AMD 6970	164,5	169,0	0,97
AMD 7970	49,0	47,5	1,03
Geforce GT 630M	95,6	133,5	0,72
GeForce GTX 580	32,1	31,9	1,01
GeForce GTX 780 Ti	44,5	27,2	1,64
GeForce GTX TITAN X	44,7	18,8	2,38

Tabulka 6.2: Doba detekce v milisekundách při použití globální a lokální paměti na různých GPU.

6.1.2 Mipmapa

V tabulce 6.3 je srovnání doby detekce bez použití mipmapy a s použitím mipmapy. Mipmapa se nejvíce projeví u GPU s velkým počtem výpočetních jednotek (viz. tabulka 6.1), protože je obraz větší a tím pádem je i více pozic posuvného okénka k vyhodnocení, to umožní těmto GPU zaplnit většinu výpočetních jednotek. Horší výsledek u AMD 6970 je způsoben malou texturovací pamětí, která způsobí časté čtení z globální paměti. U GPU GT 630M se použití mipmapy tolik neprojeví z důvodu malého počtu výpočetních jednotek, které se zaplní i při jednom obrazu z pyramidy.

GPU	Lokální paměť [ms]	Mipmapa [ms]	Zrychlení
AMD 6970	169,0	176,8	0,96
AMD 7970	47,5	14,3	3,33
Geforce GT 630M	133,5	115,5	1,16
GeForce GTX 580	31,9	14,3	2,24
GeForce GTX 780 Ti	27,2	10,7	2,54
GeForce GTX TITAN X	18,8	4,9	3,81

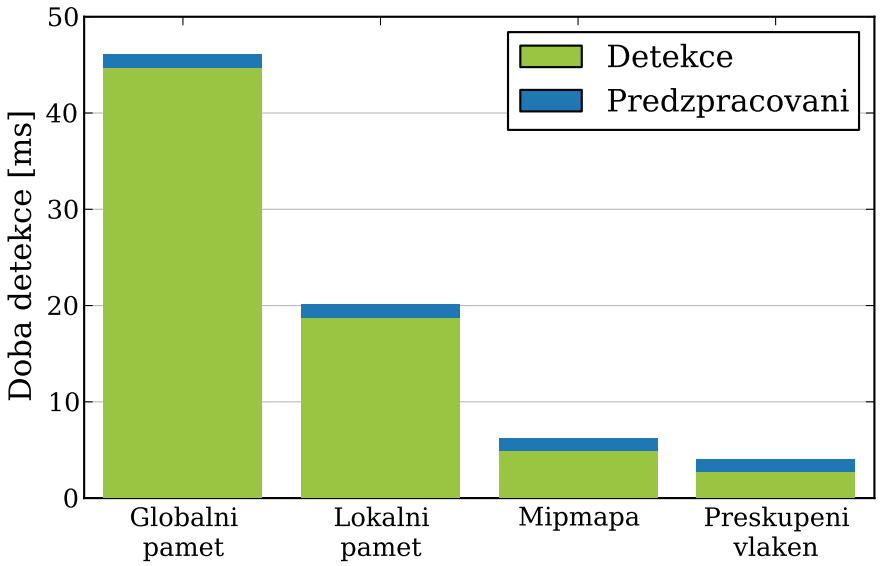
Tabulka 6.3: Doba detekce v milisekundách při použití implementace s lokální pamětí s využitím mipmapy a bez mipmapy.

6.1.3 Přeskupení vláken

V tabulce 6.4 je srovnání doby detekce s přeskupením vláken a bez přeskupení vláken. Přeskupení vláken bylo provedeno v 11, 154, 600 stupni kaskády. U všech grafický karet došlo k několika násobnému zrychlení detekce.

GPU	Bez přeskupení vláken [ms]	Přeskupení vláken [ms]	Zrychlení
AMD 6970	176,82	14,48	12,21
AMD 7970	14,27	6,48	2,20
Geforce GT 630M	115,50	31,10	3,71
GeForce GTX 580	14,27	6,07	2,35
GeForce GTX 780 Ti	10,69	4,73	2,26
GeForce GTX TITAN X	4,93	2,68	1,84

Tabulka 6.4: Doba detekce v milisekundách při použití implementace s lokální pamětí, mipmapou a přeskupením vláken a bez přeskupení vláken



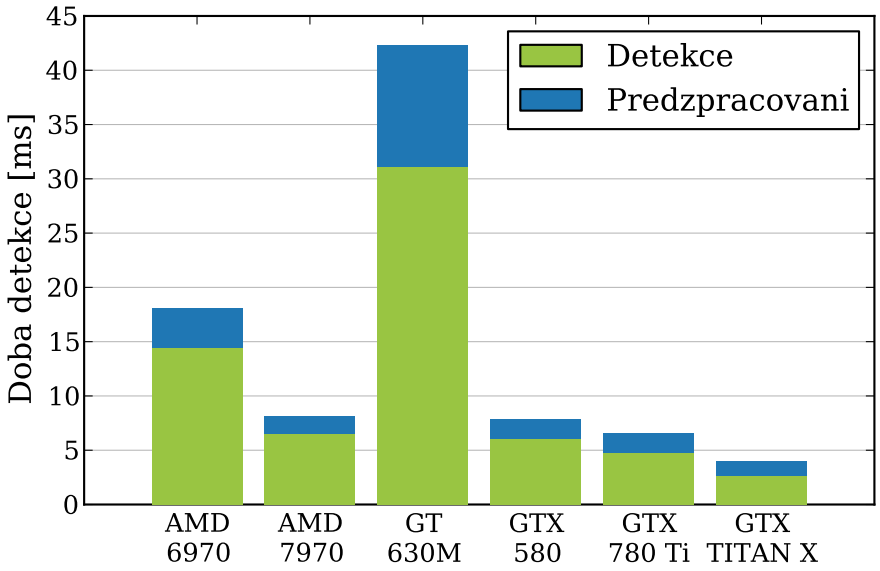
Obrázek 6.1: Srovnání doby detekce jednotlivých implementací, včetně doby předzpracování pro video 02-1920x1080 (Geforce GTX TITAN X).

6.1.4 Srovnání rychlosti

V grafu 6.1 je zobrazeno srovnání doby zpracování jednotlivých implementací pro nejvýkonnější použitou GPU Geforce GTX TITAN X. Každý sloupec v grafu je rozdělen na dvě části, kde zelená část je čas strávený detekcí na GPU a modrá část je doba strávená předzpracováním obrazu. Lze vidět, že každá optimalizace zrychlí detekci několika násobně, přičemž nejvýraznější skok je u mipmapy, díky které se plně využijí všechny výpočetní jed-

notky. Implementace se všemi optimalizacemi je na GPU GTX TITAN X téměř 9**x** rychlejší než referenční s globální pamětí.

V grafu 6.2 je porovnání výkonosti jednotlivých GPU pro implementaci se všemi optimalizacemi s vyznačením času stráveného předzpracováním. Rozdíly ve výkonu odpovídají daným GPU, kdy Geforce GT 630M je nejslabší s časem detekce42ms a nejvýkonější Geforce GTX TITAN X s časem 4ms.



Obrázek 6.2: Srovnání doby detekce testovaných GPU).

6.2 Profilování

V předchozí části byly popsány dosažené výsledky rychlosti detekce. Pro vylepšení implementace nebo zjištění důvodů, které způsobují špatnou výkonost, se využívá profilování na GPU. AMD i NVIDIA mají profilovací nástroje, které poskytují velké množství informací a statistik o běhu programu na grafické kartě. AMD CodeXL plně podporuje standard OpenCL a podává mnoho informací o GPU, včetně limitujícího faktoru dané implementace. Novější verze NVIDIA Visual Profiler již nepodporují profilování OpenCL, nicméně lze použít starší verzi 3.2, která zprostředkovává alespoň hlavní statistiky o běhu programu. Nejdůležitější informace, které je potřeba znát pro vyhodnocení implementace, je obsazenost výpočetních jednotek, hit rate vyrovnávací paměti a konflikty při přístupu do sdílené paměti.

V následující části jsou popsány jednotlivé informace pro nejrychlejší implementaci s lokální pamětí, mipmapou a přeskupením vláken. Pro detekci jsou na GPU použity dva kernely:

- `rfDetector`, který je spuštěn od na začátku detekce až po první přeskupení vláken a
- `rfDetectorSurvived`, který provádí detekci dále

Obsazenost GPU Ukazuje kolik procent z celkového množství warpů/wavefrontů je obsazeno na GPU.

Pro AMD 6970 je obsazenost 95% pro oba kernely, cože je pro toto GPU 20 wavefront z 21. Neobsazenost je zde způsobena množstvím použitých registrů, lokální paměti a velikosti work-group.
AMD 7970 GHZ má obsazenost 90%. Omezení je zde způsobené použitým množstvím skalárních registrů, z důvodu čehož se využije 36 wavefront ze 40.
NVIDIA GT 630M má nejnižší obsazení 83% také z důvodu velkého množství použitých registrů.
Snížit použití registrů lze zefektivněním kódu prováděných kernelů, což je náročný úkol a může to vyžadovat rozsáhlou úpravu kernelů.

Hit rate vyrovnávací paměti V tabulce 6.5 jsou hodnoty úspěšnosti nalezení dat ve vyrovnávací paměti. Pro GT 630M jsou rozděleny do dvou částí, úspěšnost čtení textury a L1 cache.

	AMD 6970 [%]	**AMD 7970 GHZ** [%]	**NVIDIA GT 630M** [%]	
			Tex. hit rate	**L1 hit rate**
`rfDetector`	88	86	86	99
`rfDetectorSurvived 11`	96	87	32	64
`rfDetectorSurvived 154`	76	70	33	64
`rfDetectorSurvived 600`	85	95	32	64

Tabulka 6.5: Úspěšnost čtení dat z cache v [%].

Bankové konflikty sdílené paměti Je to procento z celkového času, kdy je zpomalena sdílená paměť konflikty při čtení/zápisu.
Všechny GPU nemají téměř žádné bankové konflikty, které by zpomalily výpočet viz. tabulka (6.6).

	AMD 6970 [%]	**AMD 7970 GHZ** [%]	**NVIDIA GT 630M** [%]
`rfDetector`	1,07	0,83	0
`rfDetectorSurvived 11`	0,04	0,02	0
`rfDetectorSurvived 154`	0,01	0	0
`rfDetectorSurvived 600`	0	0	0

Tabulka 6.6: Bankové konflikty v [%] při přístupu do sdílené paměti.

6.3 Úspěšnost detekce

Vyhodnocení úspěšnosti detekce je realizováno na vytvořené anotované testovací sadě obrázků. Na každém obrázku je provedena detekce automobilů a výsledné pozitivní pozice jsou zapsány do CSV souboru. Výsledky detekce byly porovnány s anotovanou verzí podle vztahu 6.1, kde D poměr překrytí detekčních oken, vzhledem velikosti oken, S_I je obsah překrytí, S_A je plocha anotovaného detekčního okna a S_B je plocha detekčního okna.

$$D = \frac{S_I}{S_A + S_B - S_I} \tag{6.1}$$

Podle porovnání D se zvoleným prahem, v tomto případě $0,65$, jsou detekce rozděleny do tří kategorií:

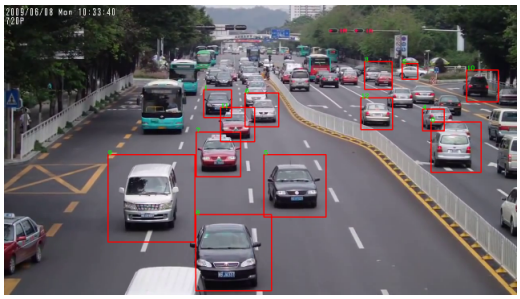
- True positive (TP) – detekční okno odpovídá anotovanému objektu
- False positive (FP) – detekční okno neodpovídá žádnému anotovanému objektu v obraze
- False negative (FN) – anotovaný objekt neodpovídá žádnému detekčnímu oknu

V tabulce 6.7 je porovnání vlivu *shrinkFactoru*, tj. počet sčítaných pixelů v řádku a sloupci během předzpracování obrazu a *scaleFactoru*, který určuje počet zmenšenin v pyramidě obrazu. První parametr ovlivňuje, jaké nejmenší objekty je možné detekovat. Pokud zvolíme detekční okénko o velikosti 20x20, tak se shrinkFactorem 4 je možné detekovat objekty s minimální velikostí 80x80. Oba tyto parametry navíc ovlivňují počet obrazů v pyramidě. V tabulce lze vidět, že při snižujícím se shrinkFactoru se zvyšuje TP až na 65%, ale zároveň roste počet chybných detekcí FP.

shrinkFactor [px]	**scaleFactor**	**TP** [%]	**FN** [%]	**FP**
4	2	11	89	30
	4	28	72	51
	6	32	68	77
2	2	41	59	201
	4	62	38	288
	6	65	35	367
1	2	65	35	464
	4	61	39	508
	6	65	35	610

Tabulka 6.7: Vliv změny shrinkFactoru a scaleFactoru na úspěšnost detekce.

(a) ShrinkFactor = 4, scaleFactor=4



(b) ShrinkFactor = 2, scaleFactor=4

Obrázek 6.3: Porovnání vlivu shrinkFactoru na detekci objektů

Kapitola 7

Závěr

Cílem diplomové práce bylo navrhnou a vytvořit program používající GPU pro detekci objektů metodou Random Forests. V úvodní části se zabývám metodou Random Forests, která používá ke klasifikaci rozhodovací stromy se slabými klasifikátory. V další části je popsáno využití GPU pro obecné výpočty, jeho programovací a paměťový model a metody pro efektivní využití jeho výkonu.

V této práci se nezabývám trénováním klasifikátoru Random forest, ale pouze detekcí. Z toho důvodu používám klasifikátor poskytnutý od pan Ing. Roman Juránek Ph.D. Tento klasifikátor je odlišný od klasické metody Random Forests, protože má rozhodovací stromy rozděleny do kaskády, která je postupně vyhodnocována a po každém stupni může být detekce ukončena.

V hlavní části práce je proveden návrh implementace detektoru na GPU s použitím standardu OpenCL pro obecné výpočty na GPU. Je zde navrhnuta referenční implementace, která je pak následně vylepšována různými technikami pro efektivnější využití prostředků grafické karty. Pro každý návrh zlepšení je vytvořena implementace a je provedeno srovnání a vyhodnocení rychlosti s předchozími verzemi.

Testování je provedeno na různých grafických kartách s různým výkonem a architekturou pro objektivní porovnání rychlosti a nezávislosti implementace na architektuře. Pro obraz 1920x1080 se rychlost detekce pohybuje mezi 4–42ms podle výkonu GPU. Takto velká rychlost je z části dána malou hloubku rozhodovacích stromů v kaskádě a zmenšením obrazu před samotnou detekcí. V poslední části je vyhodnocena úspěšnost detekce poskytnutého detektoru, která velmi závisí na zmíněném zmenšení obrazu, které ovlivňuje minimální velikost detekovaného objektu.

Další vylepšení, které by mohlo poskytnout zrychlení je zpracování více pozic posuvného okénka v obraze jedním vláknem.

Literatura

[1] Advanced Micro Devices, Inc.: AMD Accelerated Parallel Processing. OpenCL User Guide. 2014.
URL http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide.pdf

[2] Amit, Y.; Geman, D.: Shape Quantization and Recognition with Randomized Trees. *Neural Computation*, ročník vol. 9, č. issue 7, 1997: s. 1545–1588.

[3] Anderson, C. H.; Bergen, J. R.; Burt, P. J.; aj.: Pyramid Methods in Image Processing. 1984.

[4] Baumann, F.; Ehlers, A.; Vogt, K.; aj.: Cascaded Random Forest for Fast Object Detection. In *Image Analysis*, *Lecture Notes in Computer Science*, ročník 7944, editace J.-K. Kämäräinen; M. Koskela, Springer Berlin Heidelberg, 2013, ISBN 978-3-642-38885-9, s. 131–142, doi:10.1007/978-3-642-38886-6˙13.
URL http://dx.doi.org/10.1007/978-3-642-38886-6_13

[5] Breiman, L.: Bagging Predictors. *Machine Learning*, ročník 24, č. 2, 1996: s. 123–140, ISSN 0885-6125, doi:10.1023/A:1018054314350.
URL http://dx.doi.org/10.1023/A%3A1018054314350

[6] Breiman, L.: Random forests. *Machine Learning*, ročník vol. 45, č. issue 1, 2001: s. 5–32.

[7] Criminisi, A.; Shotton, J.; Konukoglu, E.: Decision Forests for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning. Technická Zpráva MSR-TR-2011-114, Microsoft Research, Oct 2011.

[8] David, A. F.: *Computer vision a modern approach*. New Jersey: Prentice-Hall, vyd. 1 vydání, 2003, ISBN 0131911937.

[9] Freund, Y.; Schapire, R. E.: A Short Introduction to Boosting. 1999.

[10] Gaster, B.: *Heterogeneous Computing with OpenCL*. Boston: Morgan Kaufmann Publishers Inc, druhé vydání, 2012.
URL http://www.sciencedirect.com.ezproxy.lib.vutbr.cz/science/book/9780124058941

[11] Herout, A.; Jošth, R.; Juránek, R.; aj.: Real-time object detection on CUDA. *Journal of Real-Time Image Processing*, ročník 6, č. 3, 201109: s. 159–170, ISSN 18618200.

[12] Hradiš, M.; Herout, A.; Zemčík, P.: Local Rank Patterns - Novel Features for Rapid Object Detection. In *In: Proceedings of ICCVG 2008*, 2008.

[13] Khronos OpenCL Working Group: The OpenCL Specification. 2014.

[14] Kriegman, D.; Ahuja, N.: Detecting faces in images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, ročník 24, č. 1, 200201: s. 34–58, ISSN 01628828, doi:10.1109/34.982883.

[15] NVIDIA Corporation: Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi. 2009.
URL http://www.nvidia.co.uk/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[16] NVIDIA Corporation: OpenCL Programming Guide for the CUDA Architecture. 2012.
URL http://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Programming_Guide.pdf

[17] Oro, D.; Fern'ndez, C.; Segura, C.; aj.: Accelerating Boosting-Based Face Detection on GPUs. In *Parallel Processing (ICPP), 2012 41st International Conference on*, Sept 2012, ISSN 0190-3918, s. 309–318, doi:10.1109/ICPP.2012.12.

[18] Sharma, B.; Thota, R.; Vydyanathan, N.; aj.: Towards a robust, real-time face processing system using CUDA-enabled GPUs. In *High Performance Computing (HiPC), 2009 International Conference on*, Dec 2009, s. 368–377, doi:10.1109/HIPC.2009.5433189.

[19] Sharp, T.: Implementing Decision Trees and Forests on a GPU. In *ECCV (4)*, *Lecture Notes in Computer Science*, ročník 5305, Springer, 2008, ISBN 978-3-540-88692-1, s. 595–608.
URL http://research.microsoft.com/apps/pubs/default.aspx?id=71445

[20] Shotton, J.; Robertson, D.; Sharp, T.: *Efficient Implementation of Decision Forests*. Springer, 2013.
URL http://research.microsoft.com/apps/pubs/default.aspx?id=194045

[21] Viola, P.; Jones, M.: Rapid Object Detection using a Boosted Cascade of Simple Features. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, ročník 1, 2001: str. 511, ISSN 1063-6919, doi:http://doi.ieeecomputersociety.org/10.1109/CVPR.2001.990517.

[22] Viola, P.; Jones, M. J.: Robust Real-Time Face Detection. *International Journal of Computer Vision*, ročník vol. 57, č. issue 2, 2004: s. 137–154, ISSN 09205691, doi:10.1023/B:VISI.0000013087.49260.fb.
URL http://link.springer.com/10.1023/B:VISI.0000013087.49260.fb

[23] Šochman, J.; Matas, J.: WaldBoost - learning for time constrained sequential detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, ročník 2, June 2005, ISSN 1063-6919, s. 150–156 vol. 2, doi:10.1109/CVPR.2005.373.

Příloha A

Obsah CD

Přiložené CD obsahuje:

- technickou zprávu v PDF
- zdrojové kódy technické zprávy v LaTeX
- zdrojové kódy a knihovny pro vytvořenou aplikaci
- použitý detektor
- sadu testovacích obrázků a videí
- dokument s popisem programu a manuál spuštění a ovládání aplikace
- plakát

Příloha B

Tabulky

Globální paměť	**AMD** [ms]		**NVIDIA** [ms]			
	6970	7970 GHZ	GT 630M	GTX 580	GTX 780	GTX TITAN X
01-1280x720	133,7	50,5	83,9	31,7	44,8	46,0
02-1920x1080	164,5	49,0	95,6	32,1	44,5	44,7
03-1920x1080	209,5	68,6	99,3	41,3	59,5	60,1
04-1280x720	99,1	42,4	53,7	26,1	38,4	39,5
05-1280x720	115,3	46,4	71,3	29,3	42,5	43,7
Lokální paměť	**AMD** [ms]		**NVIDIA** [ms]			
	6970	7970 GHZ	GT 630M	GTX 580	GTX 780	GTX TITAN X
01-1280x720	132,0	49,0	110,1	30,0	26,2	19,0
02-1920x1080	169,0	47,5	133,5	31,9	27,2	18,8
03-1920x1080	210,0	65,6	132,8	39,3	35,0	24,6
04-1280x720	96,4	40,5	66,9	23,7	22,1	16,0
05-1280x720	112,7	44,8	91,2	27,5	24,5	17,8
Mipmapa	**AMD** [ms]		**NVIDIA** [ms]			
	6970	7970 GHZ	GT 630M	GTX 580	GTX 780	GTX TITAN X
01-1280x720	114,8	13,1	96,4	12,1	9,2	4,4
02-1920x1080	176,8	14,3	115,5	14,3	10,7	4,9
03-1920x1080	191,1	14,3	120,7	14,7	12,4	5,1
04-1280x720	91,3	8,9	59,0	8,1	6,6	3,2
05-1280x720	100,5	11,2	80,4	10,4	8,4	4,0

Tabulka B.1: Naměřené časy detekce pročistě GPU detekci.

AMD 6970	**Zmenšení**	**Shrink**	**Konv.**	**Celkem**
01-1280x720	0,5	0,6	0,8	1,9
02-1920x1080	1,0	1,0	1,5	3,6
AMD 7970 GHZ	**Zmenšení**	**Shrink**	**Konv.**	**Celkem**
01-1280x720	0,2	0,2	0,4	0,8
02-1920x1080	0,4	0,4	0,7	1,6
GT 630M	**Zmenšení**	**Shrink**	**Konv.**	**Celkem**
01-1280x720	2,0	1,2	1,8	5,1
02-1920x1080	4,6	2,6	4,0	11,2
GTX 580	**Zmenšení**	**Shrink**	**Konv.**	**Celkem**
01-1280x720	0,3	0,2	0,3	0,9
02-1920x1080	0,7	0,4	0,7	1,8
GTX 780 Ti	**Zmenšení**	**Shrink**	**Konv.**	**Celkem**
01-1280x720	0,3	0,3	0,4	1,0
02-1920x1080	0,6	0,5	0,7	1,9
GTX TITAN X	**Zmenšení**	**Shrink**	**Konv.**	**Celkem**
01-1280x720	0,2	0,2	0,3	0,8
02-1920x1080	0,5	0,4	0,5	1,4

Tabulka B.2: Naměřené časy v milisekundách pro předzpracování obrazu.