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

VYUŽITÍ GRAFICKÉHO PROCESORU JAKO AKCELERÁTORU - TECHNOLOGIE OPENCL

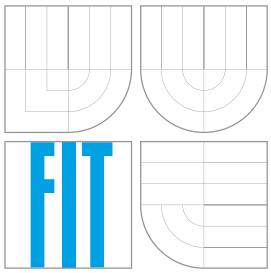
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL HRUBÝ

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYUŽITÍ GRAFICKÉHO PROCESORU JAKO AKCELERÁTORU - TECHNOLOGIE OPENCL

EXPLOITATION OF GRAPHICS PROCESSOR AS ACCELERATOR - OPENCL TECHNOLOGY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MICHAL HRUBÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Dr. Ing. PAVEL ZEMČÍK

BRNO 2011

Abstrakt

Tato práce se zabývá technologií OpenCL a jejím využitím pro detekci objektů. První část je zaměřená na popis principů technologie OpenCL a základní teorii o detekci objektů. Následuje kapitola analýzy, kde je navržená metoda zpracování s přihlédnutím na možnosti OpenCL. Další část popisuje samotnou implementaci detekční aplikace a experimentálně vyhodnocuje výkon detektoru. Poslední kapitola shrnuje dosažené výsledky.

Abstract

This work deals with the OpenCL technology and its use for the task of object detection. The introduction is devoted to description of OpenCL fundamentals, as well as basic theory of object detection. Next chapter of the work is analysis, with design proposal which takes into consideration the possibilities of OpenCL. Further, there's description of implementation of detection application and experimental evaluation of detector's performance. The last chapter summarizes the achieved results.

Klíčová slova

OpenCL, grafická karta, detekce objektů, klasifikátor, AdaBoost, WaldBoost, Local Binary Patterns, paralelní algoritmus, kernel

Keywords

OpenCL, graphics card, object detection, classifier, AdaBoost, WaldBoost, Local Binary Patterns, parallel algorithm, kernel

Citace

Michal Hrubý: Využití grafického procesoru jako akcelerátoru - technologie OpenCL, diplomová práce, Brno, FIT VUT v Brně, 2011

Využití grafického procesoru jako akcelerátoru - technologie OpenCL

Prohlášení

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pana Doc. Dr. Ing. Zemčíka.

.....
Michal Hrubý
24. mája 2011

Poděkování

Chcel by som poďakovať vedúcemu práce pánovi Doc. Dr. Ing. Zemčíkovi, za poskytnutie odbornej pomoci.

Obsah

Kapitola 1

Úvod

V dnešnom svete nás takmer všade obklopujú počítače, pričom každým rokom rastie ich výkon. V posledných rokoch však nárast výkonu už nie je taký zreteľný pre bežného užívateľa, čoho dôvodom je, že už sa nezvyšuje rýchlosť procesoru, ale pribúdajú výpočetné jadrá, ktoré umožňujú spracovávať niekoľko rôznych úloh súbežne.

Okrem hlavného procesora obsahujú dnešné počítače aj špecializované obvody na urýchlenie niektorých výpočtov, a jednou z takýchto komponent je grafický akcelerátor, pre ktorý tiež platí tento trend. Donedávna sa grafické akcelerátory používali najmä na grafické operácie - zobrazovanie 3D scén. S postupom času však začali pribúdať aj iné úlohy a ako výkon grafických procesorov rástol, stávali sa postupne rovnako flexibilné ako hlavné procesory, pričom poskytujú väčšiu hrubú výpočtovú silu.

Aby bolo možné využiť túto masívnu výpočtovú silu aj na iné ako grafické výpočty, objavil sa nový koncept (nazývaný obecné výpočty na grafickej jednotke - anglicky *General purpose computing on graphics processing unit*, alebo *GPGPU*), ktorý umožňuje využiť grafický akcelerátor na výpočty bez nutnosti použitia špeciálnych grafických programovacích rozhraní. Vďaka *GPGPU* sa začali grafické akcelerátory používať v rôznych aplikáciách ktoré spracovávajú veľké množstvo dát za použitia vektorových operácií ako napríklad skladanie proteínov, kryptografia, spracovanie signálu, simulácie využívajúce genetické algoritmy alebo inteligenciu roju, zobrazovanie medicínskych dát, spracovanie videa a mnohé iné.

Táto práca sa bude zaoberať práve spracovaním videa, konkrétne detekciou objektov na grafickom akcelerátore. Pri tejto úlohe sa zisťuje, či sa objekt danej triedy (chodec, ľudská tvár, atď.) nachádza v snímke z videa a jeho presná pozícia. Detekcia objektov sa využíva na robotické videnie, v rôznych sledovacích systémoch, ale aj na interakciu človeka s počítačom a mnohé iné aplikácie.

V súčasnej dobe existuje viacero aplikačných programovacích rozhraní, ktoré umožňujú pristupovať k výpočtovým prostriedkom grafických akcelerátorov, pričom medzi novšie patrí OpenCL, ktoré je otvoreným štandardom a okrem grafických procesorov je schopné spúšťať výpočty aj na iných zariadeniach dostupných danému počítaču (napríklad DSP, Cell BE ale aj na štandardných procesoroch).

Táto práca je štruktúrovaná tak, aby čitateľa oboznámila s technológiou OpenCL, jej základnými myšlienkami a upozornila na techniky optimalizujúce rýchlosť spracovávania algoritmov na grafických kartách.

Ďalšia kapitola popisuje metódy, ktoré sa používajú na detekciu objektov vo videu, algoritmy umožňujúce trénovanie klasifikátorov, a príznaky, ktoré je možné extrahovať z obrazu, a teda použiť v klasifikátore.

Štvrtá kapitola sa venuje súčasným prístupom k detekcií objektov vo videu a je v nej navrhnutý paralelný detekčný algoritmus vhodný na použitie na grafickom akcelerátore a v neposlednom rade sa zaoberá vstupmi, ktoré bude takýto algoritmus vyžadovať.

V piatej kapitole sa nachádza popis samotnej implementácie navrhnutého algoritmu pomocou technológie OpenCL. Detailne je rozvedená implementácia hostiteľskej časti ako aj kernelov vykonávaných na grafickom akcelerátore, pričom veľký dôraz sa kladie na vysokú optimalizáciu výslednej aplikácie.

Posledná kapitola experimentálne vyhodnocuje výsledný detektor, zaoberá sa hľadaním optimálnych parametrov a porovnáva OpenCL s existujúcimi implementáciami, ktoré využívajú na detekciu hlavný procesor počítača.

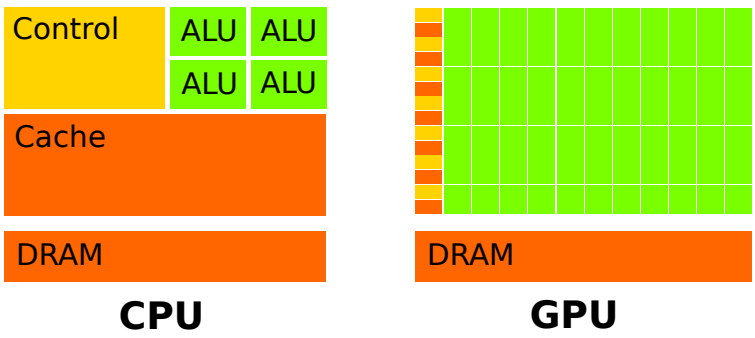
Kapitola 2

Technológia OpenCL

V tejto kapitole sa nachádza stručný popis princípov technológie OpenCL, vývoj predchádzajúci jej vzniku, krátke porovnanie s podobnými aplikačnými rozhraniami a optimalizačné techniky, ktoré sa používajú na akceleráciu výpočtov na grafických akcelerátoroch. Poskytuje teda zhrnutie súčasného stavu potrebné pre prácu.

2.1 Historický vývoj

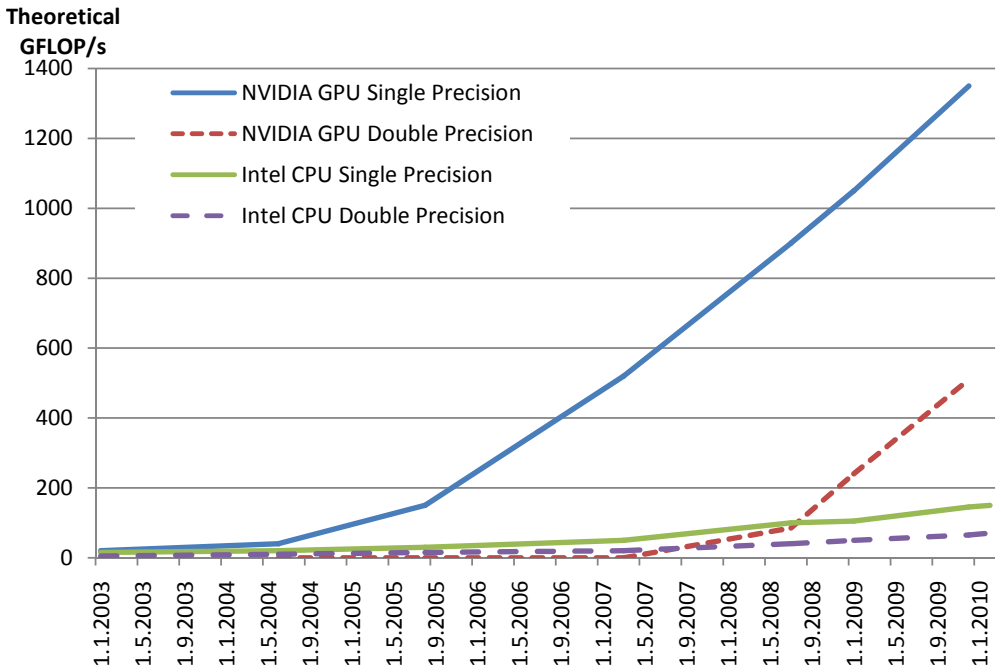
V šesťdesiatych rokoch dvadsiateho storočia si spoluzakladateľ firmy Intel Gordon E. Moore všimol, že každý rok sa dvojnásobne zvýši počet tranzistorov, ktoré možno umiestniť na integrovaný obvod[15] (neskôr opravil túto dobu na každé dva roky), a tento trend, ktorý bol neskôr nazvaný Moorov zákon, platí až do dnešnej doby. Spočiatku s týmto súviselo pravidelné zvyšovanie výkonu nielen samotných obvodov ale aj programov, keďže nové generácie procesorov používali stále vyššie hodinové frekvencie, a teda novšie procesory boli schopné vykonať aplikačný kód rýchlejšie bez akýchkoľvek zmien v samotných programoch. Od roku 2003 sa však tento trend spomalil, pretože ďalšie zvyšovanie hodinovej frekvencie by spôsobovalo vysokú spotrebu energie a tepelné straty[13]. Preto takmer všetci výrobcovia mikroprocesorov prešli na zvyšovanie počtu výpočtových jednotiek, čo stále umožňuje zvyšovať výkon, avšak aplikácie je nutné pozmeniť aby využívali viacero vlákien a boli teda schopné využiť tejto dodatočnej výpočtovej sily. Ovládanie paralelného programovania sa takto stáva veľmi dôležitým pre softvérových vývojárov.



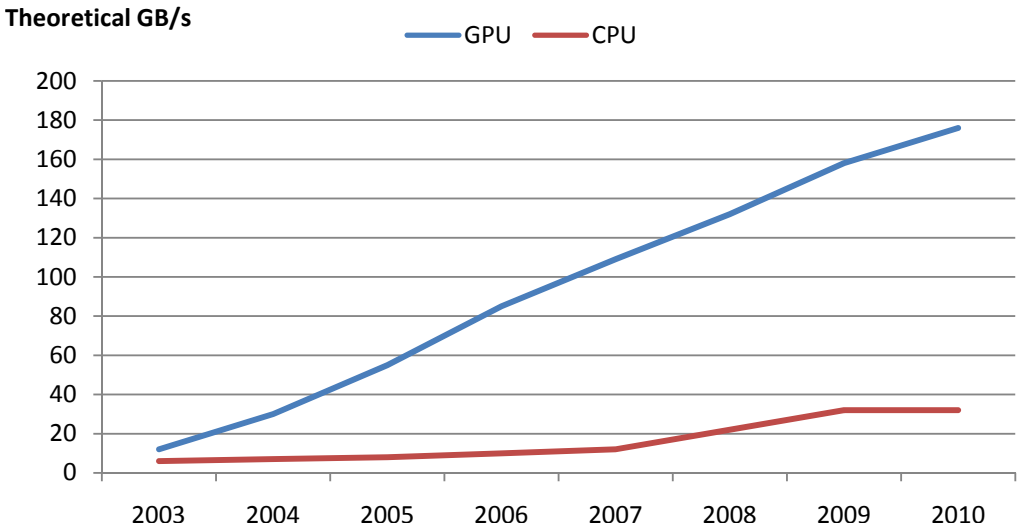
Obr. 2.1: Využitie tranzistorov v CPU a GPU čipoch (prevzaté z [17])

Polovodičový priemysel sa vydal dvoma trajektóriami - viac-jadrová (anglicky *multi-core*), ktorá sa snaží udržať vysokú rýchlosť sekvenčných programov s využitím viacerých

jadier, ktoré vykonávajú inštrukcie mimo poradia (*out-of-order*). Príkladom takéhoto mikroprocesoru je Intel Core i7-970 so šiestimi jadrami implementujúci plnú inštrukčnú sadu x86. Druhá trajektória sa nazýva mnoho-jadrová (*many-core*), vykonávajúca inštrukcie v poradí a optimalizovaná na priepustnosť[13]. Jedným zo zástupcov tohto smeru je GPU od spoločnosti NVIDIA GeForce GTX 580 ktoré obsahuje 512 jadier. Keďže GPU venujú viac tranzistorov na čipe aritmetickým jednotkám (znázornené na obrázku 2.1), má *many-core* architektúra omnoho vyšší výkon vo floating-point operáciách a vysokú pamäťovú priepustnosť (viď obrázok 2.2 a 2.3).



Obr. 2.2: Výkon vo floating-point operáciách (prevzaté z [17])



Obr. 2.3: Pamäťová priepustnosť (prevzaté z [17])

Porovnanie s inými GPGPU technológiami

S príchodom GPGPU sa postupne objavilo niekoľko aplikačných rozhraní, vďaka ktorým je možné využiť výpočtovú silu grafických procesorov. V dnešnej dobe sa používajú napríklad tieto rozhrania:

- BrookGPU
- CUDA
- DirectCompute

Hlavné rozdiely medzi týmito rozhraniami sú najmä v podporovaných zariadeniach a platformách, a jazyku, ktorý sa používa na implementáciu kernelov.

BrookGPU podporuje viacero platforiem a GPU ako aj CPU *backend*, avšak už sa niekoľko rokov nevyvíja a na grafických kartách spoločnosti ATI používa staršie rozhranie CTM (*Close-To-Metal*), ktoré ATI už nepodporuje a odporúča používanie OpenCL.

CUDA, je síce platformovo-nezávislá, ale je podporovaná iba na GPU od spoločnosti NVIDIA[3]. Kernely sa implementujú pomocou *C for CUDA*, jazyku podobnému C s určitými obmedzeniami ale aj rozšíreniami, pričom novšie verzie podporujú aj podmnožinu funkcionality jazyka C++.

DirectCompute je podporovaný na všetkých grafických kartách kompatibilných s DirectX verzie 10, avšak funguje iba na platforme Microsoft Windows.

OpenCL, ako otvorený štandard, podporuje rôzne typy zariadení - CPU, GPU, DSP procesory atď., aj keď záleží najmä na dodávateľoch jednotlivých zariadení aké platformy budú podporované - v súčasnosti je možné využívať OpenCL na grafických kartách spoločností NVIDIA aj AMD na platformách Microsoft Windows, Mac OS X aj Linux.

2.2 Princípy OpenCL

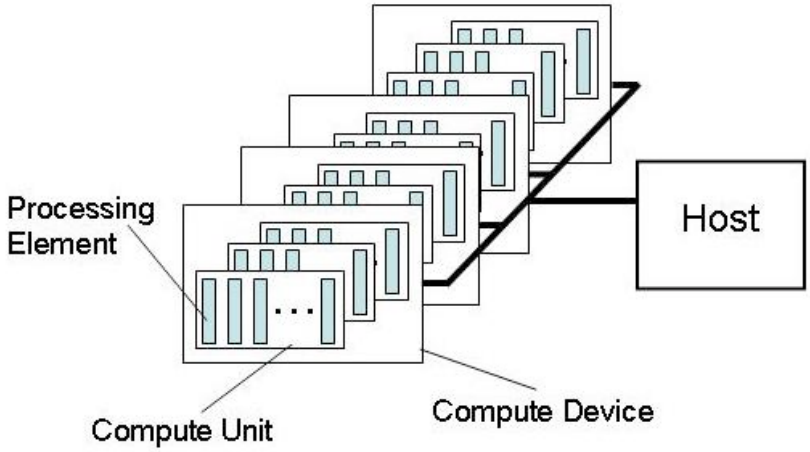
Aby bolo možné využiť stále sa rozširujúci paralelizmus, ktorý ponúkajú mikroprocesory rôznych architektúr, vydalo v roku 2009 technologické konzorcium *Khronos* otvorený štandard OpenCL (*Open Computing Language*). OpenCL zahrňuje framework pre paralelné programovanie, čiže ovládacie aplikačné programovacie rozhrania (*API*) a programovací jazyk *OpenCL C* (založený na C99 s niekoľkými rozšíreniami aj obmedzeniami) pre programovanie samotných kernelov, teda funkcií vykonávaných na OpenCL zariadeniach[2].

Základné princípy OpenCL je možné popísať štyrmi základnými modelmi[2]:

- Model platformy
- Model vykonávania
- Model pamäte
- Programovací model

Model platformy

OpenCL platforma pozostáva z hostiteľského zariadenia ku ktorému je pripojené jedno alebo viac OpenCL zariadení. Tieto sa ďalej delia na jednu alebo viac výpočtových jednotiek (*compute units*), ktoré obsahujú spracovávacie elementy (*processing elements*)[12]. Znázornenie tohto modelu je možné vidieť na obrázku 2.4. Aplikácia využívajúca OpenCL odosiela príkazy na vykonávanie výpočtov na processing elementoch jednotlivých zariadení.



Obr. 2.4: Model platformy (prevzaté z [12])

Model vykonávania

Vykonávanie OpenCL programu má dve časti:

- hostiteľský program - definuje kontext a spravuje vykonávanie kernelov
- kernel - program, ktorý sa vykonáva na OpenCL zariadení

Pred vykonaním samotného kernelu sa vytvorí jedno- až troj-rozmerný priestor indexov, pričom jednotlivé inštancie kernelu pracujú na bodoch v tomto priestore a nazývajú sa pracovné položky (*work-item*). Každá pracovná položka vykonáva rovnaký kód (aj keď vzhľadom na vetvenie presná cesta kódom sa nemusí zhodovať) na inej časti vstupných dát.

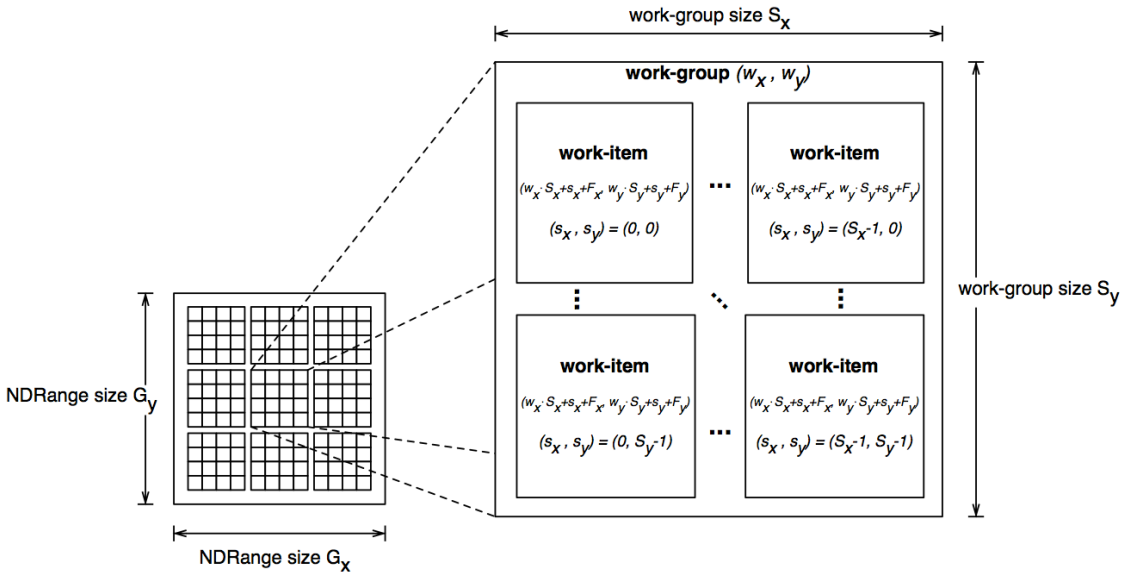
Pracovné položky sú organizované do pracovných skupín (*work-group*). Pracovná skupina sa vykonáva súčasne na processing elementoch jednej výpočtovej jednotky a OpenCL umožňuje synchronizáciu výpočtu pracovných položiek vrámci jednej skupiny.

Model pamäte

OpenCL kernel má prístup ku štyrom rôznym pamäťovým oblastiam:

- globálna pamäť - všetky pracovné položky môžu čítať z a zapisovať do tejto pamäte, zodpovedá pamäti zariadenia
- konštantná pamäť - prístupná všetkým pracovným položkám, avšak iba na čítanie
- lokálna pamäť - môžu z nej čítať a zapisovať do nej len pracovné položky vrámci rovnakej pracovnej skupiny
- privátna pamäť - prístupná iba jednej pracovnej položke

Všetky vyššie uvedené pamäťové oblasti patria OpenCL zariadeniu, aplikácia bežiaca na hostiteľskom počítači používa OpenCL API na vytvorenie pamäťových objektov v globálnej pamäti a pridáva do fronty pamäťové príkazy (ako čítanie, zápis, kopírovanie alebo mapovanie) na prácu s nimi[12].



Obr. 2.5: Znázornenie dvojrozmerného indexového priestoru a jeho rozdelenie na pracovné skupiny a pracovné položky (prevzaté z [12])

Programovací model

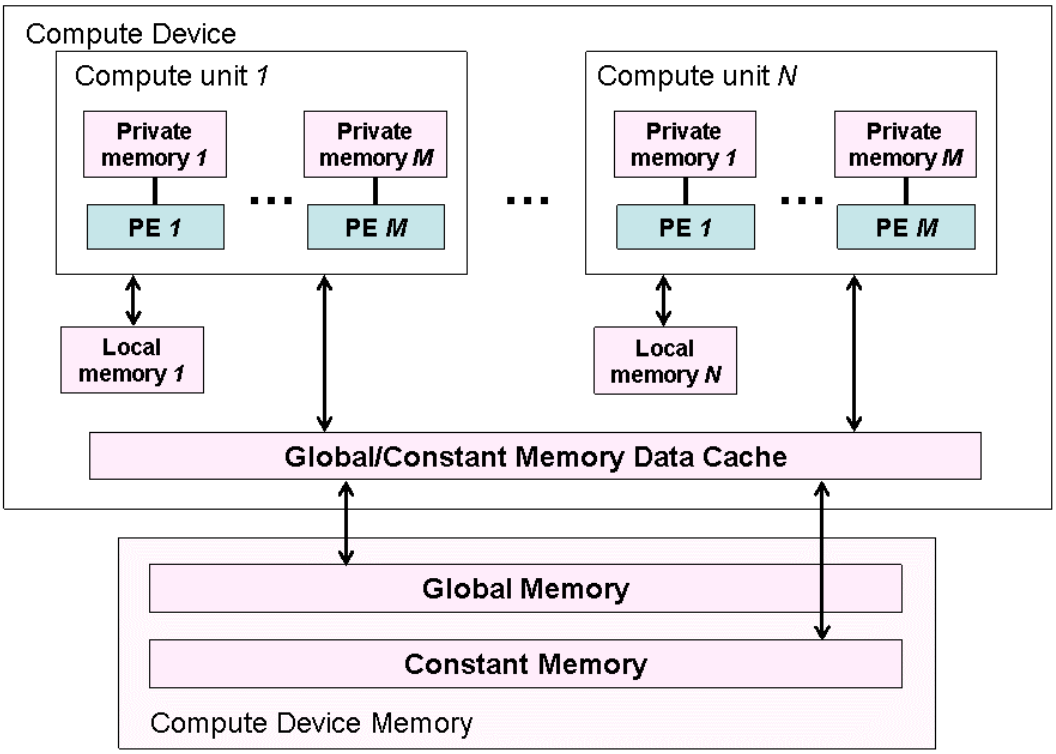
Model vykonávania v OpenCL podporuje dva programovacie modely - dátovo paralelný a úlohovo paralelný, poprípade aj miešanie oboch, aj keď primárne sa OpenCL zameriava na dátový paralelizmus.

Pri využití dátového paralelizmu sa výpočet definuje ako sekvencia inštrukcií aplikovaná na jednotlivé prvky pamäťového objektu, pričom priestor indexov (ako bolo popísané v sekcií o modele vykonávania 2.2) definuje presné mapovanie dát na pracovné položky. Úlohový paralelizmus definuje model v ktorom sa inštancia kernelu vykonáva na výpočtovej jednotke bez indexového priestoru. Pri použití tohto modelu sa paralelizmus vyjadruje používaním vektorových typov, spúšťaním viacerých úloh alebo kernelov natívnych pre dané zariadenie[12].

2.3 Akceleračné techniky

Vzhľadom na architektúru grafických mikroprocesorov je nutné dodržiavať určité zásady aby bol OpenCL program schopný využiť potenciál grafických kariet. Keďže sa architektúry grafických kariet spoločností NVIDIA a AMD mierne odlišujú, optimalizačné techniky sú v niektorých ohľadoch odlišné, avšak základné princípy sú rovnaké:

- maximalizovanie paralelného vykonávania aby sa dosiahlo maximálneho využitia
- optimalizovanie využitia pamäte
- optimalizovanie inštrukčnej priepustnosti



Obr. 2.6: Model pamäte (prevzaté z [12])

Maximalizovanie využitia grafického procesoru

Keďže OpenCL je zameraný na paralelizmus, je vhodné snažiť sa rozdeliť sériové úlohy na hostiteľský počítač a paralelnú záťaž na grafický procesor. Avšak zároveň treba brať ohľad aj na množstvo dát ktoré by bolo nutné zdieľať medzi hostiteľským počítačom a grafickým akcelerátorom, a preto je niekedy výhodné aby aj algoritmy ktoré sú sériové spracoval grafický akcelerátor.

Pokiaľ určitý paralelný algoritmus má časti, kde je paralelizmus porušený, je možné zdieľať dáta medzi jednotlivými pracovnými položkami pomocou bariér pokiaľ sú tieto dáta dostupné v rovnakej pracovnej skupine, inak je nutné využiť globálnu pamäť a dva kernely, čo má však vyššiu réžiu, a preto je najvhodnejšie mapovať algoritmus na OpenCL model tak aby využíval synchronizáciu vrámci pracovnej skupiny.

Zároveň je nutné brať do úvahy, že z hardvérového hľadiska sú pracovné položky vykonávané v skupinách, pričom u grafických kariet NVIDIA je v každom okamihu aktívnych 32 pracovných položiek (NVIDIA nazýva túto základnú skupinu *warp* [17]), grafické karty ATI vykonávajú v jednom okamihu 64 pracovných položiek (ATI označuje túto skupinu *wavefront* [1]). Preto je vhodné keď je veľkosť pracovnej skupiny násobkom jednej z týchto konštánt pre dosiahnutie vysokého využitia výpočtových prostriedkov grafickej karty.

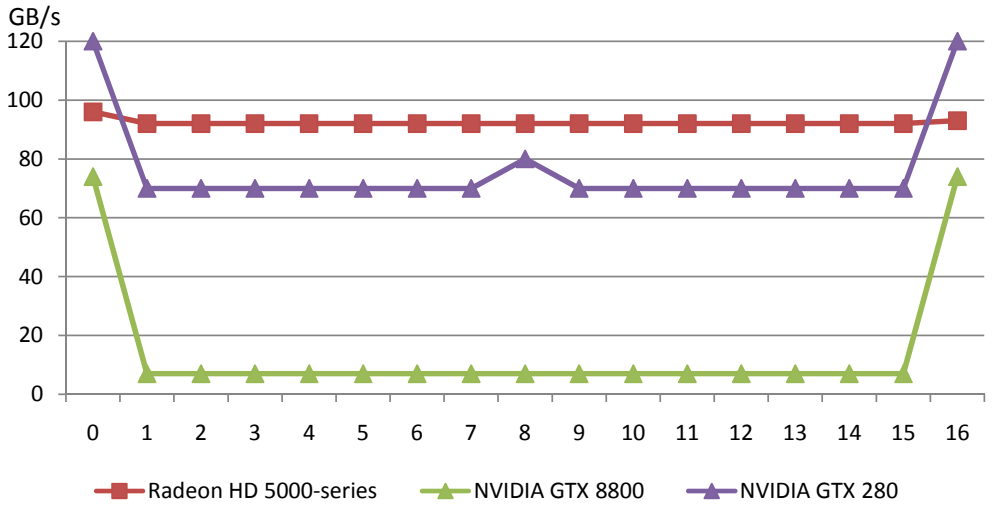
Optimalizovanie pamäťovej priepustnosti

Na maximalizovanie pamäťovej priepustnosti treba obmedziť dátové prenosy s nízkou šírkou pásma, teda najmä prenosy z hostiteľského počítača do grafickej karty[16]. Vyššiu šírku pásma pri týchto prenosoch možno dosiahnuť zlúčením menších prenosov do jedného väčšieho a tiež používaním pamäte so zamknutým stránkovaním.

Ďalej je tiež vhodné minimalizovať prenosy medzi globálnou pamäťou a zariadením, a snažiť sa využiť pamäť na čipe - čiže najmä lokálnu a konštantnú pamäť. Typické paradigma na efektívne používanie pamäte je nasledovné[17]:

- načítanie dát z globálnej do lokálnej pamäte
- synchronizácia vrámci pracovnej skupiny, aby všetky pracovné položky mohli čítať pamäť načítanú ostatnými pracovnými položkami
- spracovanie dát v lokálnej pamäti
- uloženie výsledkov do lokálnej pamäte a ďalšia synchronizácia
- zápis do globálnej pamäte

Jedno z veľmi dôležitých hľadísk optimalizácie prístupu do globálnej pamäte najmä na grafických kartách spoločnosti NVIDIA podporujúcich *CUDA Compute Capability 1.0* alebo *1.1* je zlučovanie (anglicky *coalescing*) prístupov do globálnej pamäte. Zlúčenie prístupu do pamäte nastáva keď sa použije zarovnaný a sekvenčný prístup, pričom nie všetky pracovné položky sa musia zúčastniť čítania/zápisu[16]. Vplyv samotného zarovnania na pamäťovú priepustnosť je vidieť na obrázku 2.7.



Obr. 2.7: Vplyv zarovnania na pamäťovú priepustnosť (prevzaté z [16] a [1])

Ďalšie veľmi dôležité hľadisko je efektívne využívanie lokálnej pamäte, ktorá býva až 100-krát rýchlejšia ako globálna pamäť[16], pokiaľ nedochádza ku konfliktom v pamäťových bankách, a naviac lokálna pamäť nevyžaduje zlučovanie prístupu ako globálna pamäť.

Pamäťové banky sú moduly rovnakej veľkosti, ktoré sú využívané skupinou pracovných položiek a je do nich možné pristupovať súbežne. Pokiaľ sa však viacero adries z pamäťovej požiadavky mapuje na jednu pamäťovú banku, prístupy k tejto banke sú serializované.

Výnimkou je tzv. *broadcast*, čiže prístup zoskupených pracovných položiek k rovnakej banke, ktorý taktiež prebehne súčasne.

Na minimalizovanie konfliktov v pamäťových bankách je nutné uvedomiť si mapovanie pamäťových adries na banky a podľa toho rozvrhnúť prístup k bankám aby nenastávali konflikty[17]. Grafické karty spoločnosti NVIDIA ako aj ATI používajú bity 5:2 (poprípade 6:2 u novších kariet)[1] pamäťovej adresy na mapovanie na pamäťové banky. Celkom je teda k dispozícii 16 (alebo 32) bánk.

Výhodné je tiež využívanie textúrovej a konštantnej pamäte, ktoré sú síce iba na čítanie, no keďže sa do nich pristupuje cez vyrovnávaciu pamäť, poskytujú vyššiu šírku pásma. Samozrejme pokiaľ nastane výpadok vyrovnávacej pamäte, je prístup do nich rovnako rýchly ako do globálnej pamäte. V prípade textúrovej pamäte sa dá počet výpadkov minimalizovať zachovaním lokality prístupov.

Optimalizovanie inštrukčnej priepustnosti

Na optimalizáciu inštrukčnej priepustnosti môže aplikácia[17]:

- používať natívne matematické funkcie, ktoré síce neposkytujú plnú presnosť požadovanú OpenCL špecifikáciou, ale ich vykonanie je omnoho rýchlejšie
- minimalizovať počet rozdielnych kódových ciest vrámci jedného warpu / wavefrontu
- znížiť počet inštrukcií odstránením nadbytočných bariér, typových konverzií alebo používaním obmedzených ukazovateľov (*restricted pointer*)

Kapitola 3

Detekcia objektov vo videu

Jednou z úloh vhodných na paralelné spracovanie je detekcia objektov vo videu. Pri tejto úlohe má detektor zistiť, či sa určitý objekt nachádza v snímku zo vstupného videa. Kvalitný detektor je pritom schopný poradiť si s rôznymi ťažkosťami ako napríklad rôzne osvetlenie objektov, rotácia, tiene z iných objektov, šum z kamery atď.

Na detekciu objektov je možné použiť niekoľko prístupov[20]:

- *knowledge-based* (založené na znalostiach) - tieto metódy sa zakladajú na ľudských znalostiach toho, ako vyzerá daný objekt - pravidlá zachytávajú vzťahy medzi jednotlivými časťami objektu a najčastejšie sa tieto metódy používajú na lokalizáciu objektu

- *feature-invariant* (príznakovo invariantné) - taktiež sa používajú najmä na lokalizáciu objektu, pričom sa zameriavajú na štrukturálne rysy, ktoré sa nemenia aj keď sa zmení uhol pohľadu, osvetlenie a pod.

- *template-matching* (porovnávanie vzorov) - využívajú korelácie medzi vstupným obrázkom a uloženými vzormi, ktoré popisujú objekt ako celok alebo jeho jednotlivé rysy, a využívajú sa ako na detekciu tak aj na lokalizáciu

- *appearance-based* (založené na vzhľade) - na rozdiel od metód porovnávania vzorov sa pri týchto metódach využíva strojové učenie na získanie jednotlivých modelov z množiny trénovacích dát a následne sa využíva klasifikátor pri skenovaní obrázku

Appearance-based metódy je možné ďalej rozdeliť podľa dvoch rôznych stratégií[18]:

- lokálne - hľadajú význačné oblasti ktoré sú charakterizované rohmi, hranami alebo entropiou a v neskorších štádiách sú charakterizované riadnym deskriptorom

- globálne - narozdiel od lokálnych modelujú informácie z celého obrázku

Jedným so spôsobov akým sa detekujú objekty v obraze je použitie klasifikátorov, ktoré sa trénujú strojovým učením a sú schopné určiť či sa daný objekt nachádza v (zväčša malom) okne s ktorým pracujú. Detekcia objektov s takýmto klasifikátorom prebieha tak, že sa postupne skenuje snímok videa a klasifikátor sa aplikuje na každú možnú pozíciu obrázku [6]. V nasledujúcich podkapitolách sú predstavené algoritmy strojového učenia používané na trénovanie klasifikátorov a niekoľko nízko-úrovňových obrazových príznakov, ktoré je možné extrahovať z klasifikovaného okna.

3.1 AdaBoost

Na úlohu detekcie objektov sa veľmi často používajú klasifikátory natrénované pomocou algoritmu AdaBoost[19]. Tento algoritmus bol predstavený v roku 1995, pričom jeho hlavná výhoda je schopnosť exponenciálne znižovať chybu výsledného klasifikátoru. Základný variant algoritmu kombinuje niekoľko slabých klasifikátorov do jedného silného tak, aby výsledná klasifikačná funkcia bola presnejšia ako všetky použité klasifikátory. Slabé klasifikátory môžu mať ľubovoľnú zložitosť, ale zväčša sa volia jednoduché funkcie. AdaBoost priradí jednotlivým klasifikátorom váhu na základe ich chyby.

Vstupom algoritmu je sada trénovacích vzoriek x and ich ohodnotenie y - teda dvojice $(x_1, y_1)...(x_m, y_m)$, $x \in X, y \in Y$. V základnom variante tohto algoritmu platí, že $Y \in \{-1, 1\}$. Hlavnou myšlienkou algoritmu je uchovávanie distribúcie váh trénovacích vzoriek, vďaka ktorej sa môže prispôsobiť ťažko klasifikovateľným vzorkám z trénovacej sady a aj tieto klasifikoval správne. Váha vzorku v kroku t je $D_t(i)$, pričom na začiatku algoritmu sú váhy všetkých vzoriek inicializované na rovnakú hodnotu. V každom kroku algoritmu sa hľadá slabý klasifikátor z danej množiny, ktorý pre distribúciu $D_t(x)$ najlepšie klasifikuje vzorky v trénovacej sade. Vhodnosť klasifikátoru h_t sa zisťuje pomocou chybovej funkcie 3.1, ktorá je sumou váh zle klasifikovaných vzoriek. Pre každý slabý klasifikátor h_t sa vypočíta jeho dôležitosť α_t (3.2) - čím nižšia je jeho chyba, tým je α_t vyššie[9].

$$\epsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i) \tag{3.1}$$

$$\alpha_t = \frac{1}{2} ln \frac{1 - \epsilon_t}{\epsilon_t} \tag{3.2}$$

Následne sa aktualizuje distribúcia pre ďalší krok algoritmu 3.3, kde Z_t je normalizačný faktor, čo má za úlohu zvýšiť dôležitosť zle klasifikovaných vzoriek a znížiť správne klasifikovaných.

$$D_{t+1}(i) = \frac{D_t(i) exp(-\alpha_t y_i h_t(x_i))}{Z_t} \tag{3.3}$$

Výsledkom je silný klasifikátor 3.4.

$$H(x) = sgn\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right) \tag{3.4}$$

3.2 WaldBoost

Algoritmus WaldBoost je kombináciou AdaBoostu a Waldovou skúškou pomeru sekvenčnej pravdepodobnosti (*sequential probability ratio test - SPRT*)[14]. Vstupom tohto algoritmu sú, ako v prípade AdaBoostu, dvojice trénovacích vzoriek a ich ohodnotení $(x_1, y_1)...(x_m, y_m)$, ale naviac aj parametre ktoré udávajú požadované pomery falošných negatív α (*false negative rate* - FNR) a falošných pozitív β (*false positive rate* - FPR). Z týchto parametrov sa vypočítajú hodnoty A a B podľa vzťahu 3.5.

$$\begin{aligned} A &= \frac{1 - \beta}{\alpha} \\ B &= \frac{\beta}{1 - \alpha} \end{aligned} \tag{3.5}$$

Váhy jednotlivých vzoriek sa inicializujú na rovnakú hodnotu a následne prebieha cyklus (pre $t = 1, ..., T$) pozostávajúci z nasledujúcich krokov[14]:

1. Nájdi najlepší slabý klasifikátor h_t pomocou AdaBoost vyhľadávania.
2. Odhadni pravdepodobnostný pomer R_t pomocou vzťahu 3.6.
3. Priamo aplikuj SPRT na odhadnutý pravdepodobnostný pomer R_t a nájdi prahové hodnoty $\theta_A^{(t)}$ a $\theta_B^{(t)}$, kde každá zodpovedá jednej z podmienok vzťahu sekvenčnej stratégie 3.7 a sú jednoznačne určené medzami A a B.
4. Na základe prahových hodnôt odstráň z trénovacej sady vzorky pre ktoré $H_t \geq \theta_B^{(t)}$ alebo $H_t \leq \theta_A^{(t)}$.
5. Vytvor novú trénovaciu sadu pomocou náhodného výberu zo vzoriek ktoré ešte neboli priradené do žiadnej z tried.

$$\hat{R}_t(x) = \frac{p(H_t(x)|y = -1)}{p(H_t(x)|y = +1)} \tag{3.6}$$

$$S_m^* = \begin{cases} +1 & R_m \leq B \\ -1 & R_m \geq A \\ \# & B < R_m < A \end{cases} \tag{3.7}$$

Výstupom trénovania je teda silný klasifikátor H_T a prahy $\theta_A^{(t)}$ a $\theta_B^{(t)}$.

Vďaka prahom $\theta_A^{(t)}$ a $\theta_B^{(t)}$ nie je nutné pri samotnej klasifikácií vyhodnocovať všetky slabé klasifikátory aby sa získala výsledná odozva. Pokiaľ suma silného klasifikátoru presiahne jeden z prahov, vyhodnocovanie je ukončené, inak sa pokračuje. Detekčné úlohy zväčša používajú hodnotu β (falošné pozitíva) nastavenú na 0, čo znamená že θ_B bude ∞ pre všetky stupne klasifikácie a teda vzorku je možné iba odmietnuť predčasným ukončením. Prijať ju je možné iba vykonaním všetkých stupňov klasifikácie[10]. Z tohto dôvodu budem v nasledujúcich kapitolách používať pre premennú θ_A iba označenie θ.

Samotný proces klasifikácie vzorky x silným klasifikátorom H_T teda pozostáva z jednoduchého cyklu (opäť pre $t = 1, ..., T$)[14]:

1. Ak $H_t \geq \theta_B^{(t)}$ klasifikuj x ako člena triedy $+1$ a skonči.
2. Ak $H_t \leq \theta_A^{(t)}$ klasifikuj x ako člena triedy -1 a skonči.

Pokiaľ cyklus skončí bez toho, aby bola vzorka klasifikovaná do ktorejkoľvek triedy, porovnaj H_t s užívateľsky definovanou hodnotou γ, a klasifikuj x do triedy $+1$ ak $H_t > \gamma$, inak do -1.

3.3 Obrazové príznaky

Úloha detekcie objektu vo videu najčastejšie využíva *appearance-based* prístup, a preto boli navrhnuté rôzne metódy ako extrahovať vlastnosti textúr a tieto použiť pri trénovaní silných klasifikátorov.

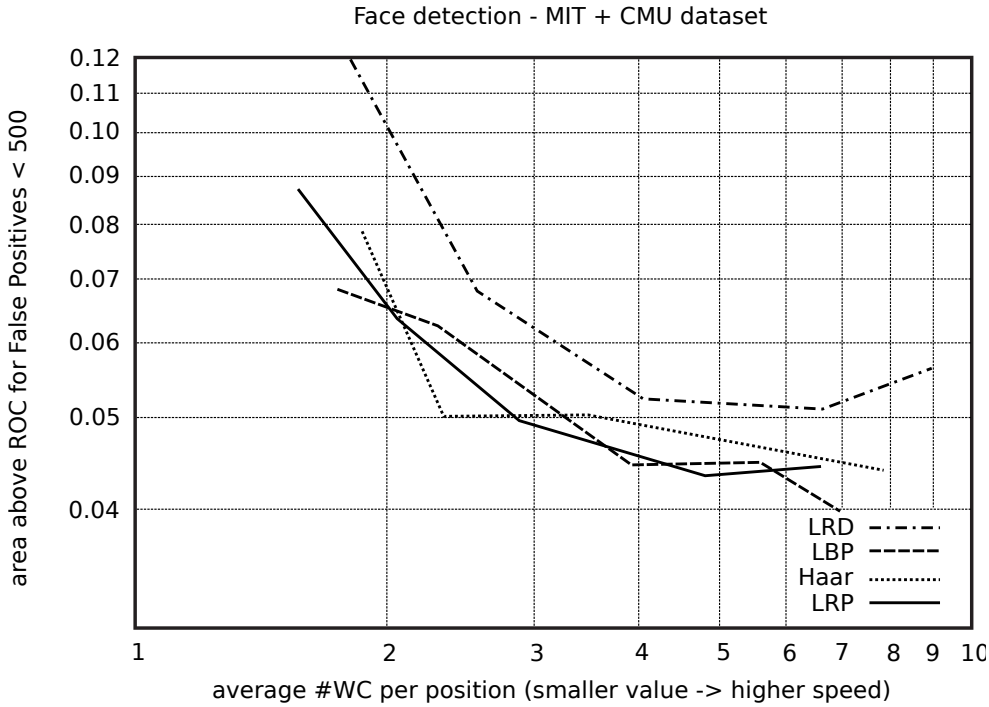
Rýchlosť detektoru veľmi záleží na type príznakov ktoré sú použité. Ideálne príznaky sú výpočtne nenáročné a do istej miery invariantné k zmenám osvetlenia a geometrii objektu.

Tiež však bolo ukázané že aj jednoduché obrazové filtre dosahujú dobré výsledky na rôznych typoch objektov[7].

Medzi často používané príznaky patria napríklad:

- Haarove príkazy
- Local Binary Patterns
- Local Rank Patterns
- Local Rank Differences

Vyhodnocovaniu výkonnosti jednotlivých príznakov sa venovali autori [7]. Ako vidieť na obrázku 3.1, Haarove príznaky, LBP aj LRP majú podobne dobrý výkon pri úlohe detekcie tvárí, a trochu horšie výsledky majú LRD. Situácia je trochu iná pri detekcií očí, kde LBP boli najlepšie, nasledované LRP, LRD a najhoršie dopadli Haarove príznaky.



Obr. 3.1: Porovnanie výkonnosti obrazových príznakov na úlohe detekcie tvárí. Graf ukazuje plochu nad ROC ako funkciu priemernej rýchlosti klasifikátoru (nižšie sú teda presnejšie a vľavo rýchlejšie klasifikátory). Klasifikátory boli trénované algoritmom WaldBoost pre rôzne FNR - 1%, 2%, 5%, 10% a 20%. (prevzaté z [7])

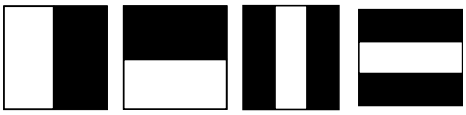
Haarove príznaky

Na obrázku 3.2 je možné vidieť 4 typy Haarových báz, z týchto sa podľa vzorca 3.8, kde x je vzorka dát a W a B sú množiny pixlov patriacich bielej resp. čiernej oblasti báze, vypočíta hodnota konkrétneho príznaku[9].

$$f(x) = \sum_{w \in W} x(w) - \sum_{b \in B} x(b) \tag{3.8}$$

Okrem obrazového príznaku obsahuje tento slabý klasifikátor aj parametre - klasifikačný prah θ a polaritu p. Tieto slabé klasifikátory sú vyhodnocované podľa vzorca 3.9.

$$h_j(x) = \begin{cases} 1 & ak\ p_j\ f_j(x) < p_j\ \theta_j \\ 0 & inak \end{cases} \tag{3.9}$$



Obr. 3.2: Niektoré tvary Haarových báz

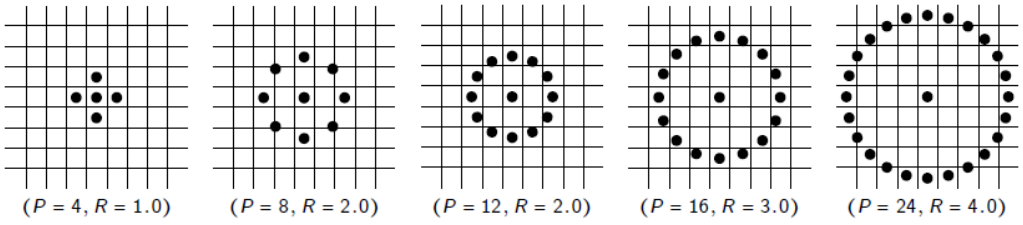
Local Binary Patterns

Local binary patterns sú veľmi jednoduchým obrazovým príznakom a sú založené na binárnom kódovaní prahovaných hodnôt intenzity[18]. V základnej forme pracujú na okolí 3x3 $(p_1...p_8)$ a používajú intenzitu stredného bodu $I(p_0)$ ako prah. Okolité pixle p_i sú následne ohodnotené vzťahom 3.10.

$$S(p_0, p_i) = \begin{cases} 1 & ak\ (I(p_i) - I(p_0)) \geq 0 \\ 0 & inak \end{cases} \tag{3.10}$$

Umocnením a následným sčítaním týchto hodnôt sa vytvorí hodnota LBP deskriptoru 3.11. Túto definíciu je možné rozšíriť na akýkoľvek počet susedných bodov pomocou bilineárnej interpolácie intenzít. LBP príznaky sú invariantné k monotónnym zmenám intenzity, ale nie sú rotačne invariantné, aj keď toto sa dá docieliť rotáciou okolných bodov[18].

$$LBP(p_0) = \sum_{i=1}^{8} S(p_0, p_i) 2^{i-1} \tag{3.11}$$



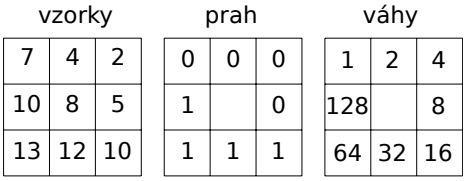
Obr. 3.3: Príklady rôznych okolí pri vyhodnocovaní LBP príznakov

Local Rank Patterns

Príznaky *Local Rank Patterns*, ktoré vychádzajú z *Local Rank Differences*, sú invariantné k monotónnym zmenám v osvetlení a zachovávajú amplitúdu lokálnych zmien.

Na skalárnom obrázku $f : \mathbb{Z}^2 \to \mathbb{R}$ je možné nadefinovať vzorkovaciu funkciu 3.12, kde $(x, u \in \mathbb{Z}^2, g : \mathbb{Z}^2 \to \mathbb{R})$[8].

$$S_x^g(u) = (f * g)(x + u) \tag{3.12}$$



Obr. 3.4: Vyhodnotenie LBP príznaku pre okolie s ôsmymi vzorkami (tvorí LBP = 11110000b = 240)

Táto vzorkovacia funkcia je parametrizovaná konvolučným jadrom g, ktoré sa aplikuje pred samotným vzorkovaním a vektorom x určujúcim počiatok vzorkovania. Ďalej sa definuje vektor relatívnych súradníc 3.13, určujúci okolie ľubovolného tvaru a spolu so vzorkovacou funkciou sa použije na získanie vektoru 3.14 (nazývaného tiež maska) popisujúceho okolie tohto tvaru na pozícií x v obrázku.

$$U = [u_1 u_2 ... u_n], u_i \in \mathbb{Z}^2, n \in \mathbb{N} \tag{3.13}$$

$$M = [S_x^g(u_1) S_x^g(u_2) ... S_x^g(u_n)] \tag{3.14}$$

Pre každý element k tohto vektoru sa definuje jeho *rank* vzťahom 3.15.

$$R_k = \sum_{i=1}^{n} \begin{cases} 1 & ak\ M_k < M_i \\ 0 & inak \end{cases} \tag{3.15}$$

Rank je teda poradím daného členu masky v zaradenej postupnosti všetkých členov masky. Táto hodnota je tiež nezávislá na lokálnej energií obrázku, čo je užitočná vlastnosť tohto príznaku, definovaného ako 3.16 ($a, b \in \{1, ..., n\}$), kde n je počet vzoriek vzatých z okolia, čo znamená že výsledok LRP je unikátny pre každú kombináciu hodnôt R_a a R_b, takže LRP môžeme definovať aj alternatívnym zápisom 3.17 [8].

$$LRP(a, b) = R_a \cdot n + R_b \tag{3.16}$$

$$LRP(a, b) = [R_a\ R_b] \tag{3.17}$$

Kapitola 4

Analýza

Táto kapitola rekapituluje doterajšie metódy používané na detekciu objektov, na základe ktorých sú vybrané metódy a príznaky vhodné pre OpenCL implementáciu. Zároveň popisuje návrh paralelných algoritmov vhodných na detekciu objektov spolu s dátami ktoré budú potrebovať.

Jedným z prvých algoritmov ktorý bol schopný detekcie objektov v reálnom čase bol publikovaný autormi Viola a Jones v [19] a využíval mierne modifikovaný AdaBoost algoritmus spolu s príznakmi ktoré sa podobali Haarovým bázam. Aby bol dosiahnutý dostatočne rýchly čas detekcie využívala sa kaskáda klasifikátorov, kde prvé klasifikátory kaskády používali minimálny počet príznakov a ich hlavnou úlohou bolo rýchlo vylúčiť pozície v obraze kde sa nemôže nachádzať hľadaný objekt (vylúčenie pozadia atď.). Nevýhodou veľkej kaskády klasifikátorov je, že jednotlivé klasifikátory musia mať vysoký pomer detekcie (aj keď zároveň stačí nízky pomer falošných pozitív). Ďalšie urýchlenie bolo dosiahnuté použitím integrálneho obrázku, čo umožnilo veľmi rýchle vyhodnocovanie príznakov.

Nedostatkom kaskády bola však strata informácie medzi jej jednotlivými úrovňami, napriek tomu že výsledok jednej úrovne tvorí veľmi dobrý základ na vyhodnotenie ďalšej, čo ju robí suboptimálnou. Jedným z vylepšení pôvodného detektoru bolo teda nahradenie kaskády klasifikátorov algoritmom WaldBoost[14], vďaka ktorému sa stal klasifikátor monolitický, pričom je stále schopný predčasného ukončenia detekcie.

Autori [5] dokonca predstavili implementáciu WaldBoost klasifikátoru na grafickej karte pomocou architektúry CUDA. Použitými príznakmi boli Local Rank Patterns, ktoré sú vhodné na hardvérovo akcelerované implementácie. Problémom v tejto implementácií bola však práve vlastnosť WaldBoost algoritmu - predčasné ukončovanie detekcie - aj keď je táto vlastnosť veľmi prospešná z hľadiska rýchlosti algoritmu, na masívne paralelných architektúrach, akými sú grafické procesory, vytvára problém kedy sa niektoré pracovné položky zo skupiny ukončia skôr ako iné, čo vedie k divergujúcim *warpom* a nižšiemu využitiu výpočetného výkonu grafickej karty, keďže sa stáva, že celá pracovná skupina čaká na vyhodnotenie malého počtu kandidátnych pozícií. Toto viedlo autorov k použitiu časovo náročného preusporiadávania vlákien.

Iný prístup bol zvolený v [11], kde sa autori rozhodli použiť viac-stupňovú implementáciu, kde prvý stupeň detekcie je spustený na programovateľnom hradlovom poli, využíva sa pri ňom klasický AdaBoost bez predčasného ukončovania s viac-blokovými príznakmi Local Binary Patterns. Po ukončení tohto kroku je zoznam kandidátnych pozícií zo vstupného obrázku značne obmedzený a zvyšné pozície je možné vyhodnotiť komplexnejším klasifikátorom napríklad aj na samotnom CPU.

Po zvážení doterajších implementácií som sa rozhodol implementovať detektor založený

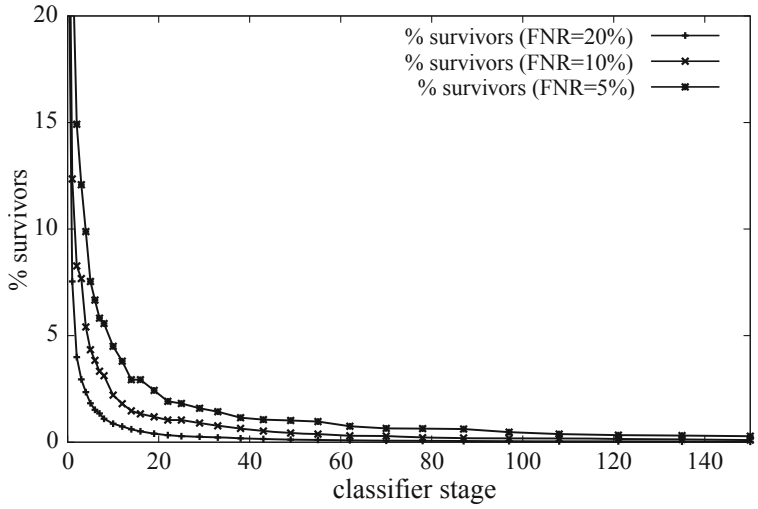
na algoritme WaldBoost podobný [5], pričom sa budem snažiť minimalizovať negatívny dopad predčasného ukončovania detekcie. Ako príznaky použijem viac-blokové Local Binary Patterns rovnako ako v [11], vzhľadom na dostupnosť textúrovacích jednotiek na grafických procesoroch, ktoré umožňujú jednoduché načítavanie a zároveň bilineárnu interpoláciu spracovávaných obrázkov. Keďže klasifikátor je natrénovaný na detekciu objektov iba určitého rozlíšenia, bude taktiež nutné budovať pyramídu vstupných obrázkov s rôznymi rozlíšeniami, vďaka čomu sa dosiahne nezávislosť detekcie od rozlíšenia vstupného obrázku.

4.1 Návrh algoritmu

Efektívna paralelná implementácia musí riešiť dva hlavné problémy:

- detekciu objektov na bloku určitej veľkosti
- paralelné spustenie takýchto blokov na celom vstupnom obrázku

Po analýze správania sa WaldBoost klasifikátoru, ktoré je zobrazené na obrázku 4.1, je možné vidieť určitú podobnosť so správaním paralelných redukčných algoritmov. Uvedené správanie klasifikátoru je však globálne pre celý obrázok, čo prináša problém, keďže po rozdelení obrázku na samostatne spracovávané bloky môže byť iné. Odhadujem však, že vyhodnotením väčšieho počtu slabých klasifikátorov sa bude blížiť aj lokálne správanie globálnemu.



Obr. 4.1: Pomer oblastí ktoré treba vyhodnocovať po vyhodnotení určitého počtu príznakov (prevzaté z [5])

S princípu funkcie klasifikátoru naviac nie je možné definovať koľko slabých klasifikátorov je pre daný blok obrázku nutné vyhodnotiť aby sa znížil počet skúmaných pozícií na určitý pomer. Toto prináša ďalšie problémy so synchronizáciou, keďže klasifikátor detekuje v každej skupine iné množstvo pozitívnych odoziev. Naviac OpenCL, ako aj iné paralelné architektúry, neposkytuje možnosť synchronizácie na úrovni viacerých pracovných skupín, kvôli vysokej cene zabudovania takejto funkcionality do hardvéru grafických kariet s vysokým množstvom procesorov. Preto sa paralelné redukčné algoritmy zväčša dekomponujú do viacerých kernelov a ako globálny synchronizačný bod slúži spustenie kernelu. Takéto

riešenie má zanedbateľnú réžiu v hardvéri, pričom aj softvérová réžia je veľmi nízka[4]. Postupnou redukciou sa tiež bude zväčšovať oblasť s ktorou sa pracuje, čím sa bude správanie klasifikátoru ďalej blížiť globálnemu. Preto som sa rozhodol pre zostavenie detekčného algoritmu z viacerých kernelov, kde každý kernel redukuje počet detekčných okien na predom daný pomer, takže výsledkom detekcie bude pole s rozptýlenými hodnotami (*sparse array*).

Implementácia aplikácie využívajúcej štatistické klasifikátory, akým je klasifikátor natrénovaný WaldBoost algoritmom, bude pozostávať z nasledujúcich častí:

- príprava dát klasifikátoru
- predspracovanie vstupného obrázku
- detekcia objektov
- spracovanie výsledkov

V nasledujúcich podkapitolách sú bližšie popísané postupy, ktorými som sa rozhodol jednotlivé časti vyriešiť.

4.2 Dáta klasifikátoru

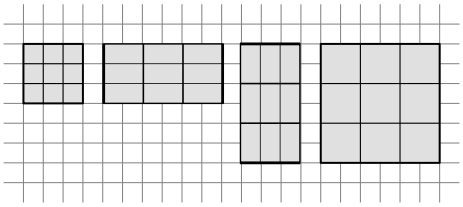
Ako bolo spomínané v kapitole 3.2, silný klasifikátor natrénovaný algoritmom WaldBoost obsahuje dáta ako sú hodnoty θ jednotlivých slabých klasifikátorov a veľkosť detekčného okna. Každý slabý klasifikátor má naviac priradené hodnoty posunutia a veľkosti bloku z ktorého sa vypočítavajú LBP príznaky a zároveň tabuľku hodnôt α zodpovedajúcu jednotlivým hodnotám LBP príznaku.

Táto práca sa nezaoberá samotným trénovaním klasifikátorov, budú teda prebrané z existujúcich implementácií vo forme XML súboru, ktorý obsahuje parametre klasifikátoru, ako aj všetky hodnoty slabých klasifikátorov. Ukážka takéhoto súboru je znázornená na obrázku 4.2.

```
<WaldBoostClassifier classifierName="" minStdDev="0"
  imageSizeX="24" imageSizeY="24" type="LBP">
  <stage posT="1e+50" negT="-0.0657414">
    <HistogramWeakHypothesis
      predictionValues="-1.1776324758022 -0.55194005503571...">
      <LBPFeature positionX="7" positionY="5"
        blockWidth="2" blockHeight="2"
        binMappingType="NONE"/>
    </HistogramWeakHypothesis>
  </stage>
  <stage posT="1e+50" negT="0.0839644">
    <HistogramWeakHypothesis
      predictionValues="-0.9500509672182 -0.30672576045967...">
      <LBPFeature positionX="11" positionY="4"
        blockWidth="2" blockHeight="2"
        binMappingType="NONE"/>
    </HistogramWeakHypothesis>
  </stage>
  ...
</WaldBoostClassifier>
```

Obr. 4.2: Ukážka XML súboru s natrénovaným klasifikátorom s LBP príznakmi

Autori [7] ukázali, že na detekciu objektov, ktorá je zároveň efektívne implementovaná v hardvéri, stačí použiť príznaky s veľkosťami bloku 1×1, 1×2, 2×1 a 2×2, ktoré sú vidno na obrázku 4.3. Takéto obmedzenie pritom vôbec neznižuje presnosť výsledného klasifikátoru.



Obr. 4.3: Konfigurácie LBP príznakov (prevzaté z [6])

4.3 Predspracovanie

Pred samotnou detekciou objektov je potreba predspracovať vstupný obrázok a vytvoriť pyramídu s rôznymi rozlíšeniami. Vzhľadom na to, že klasifikátor spracováva vstupný obraz iba v odtieňoch šedi, prevedú sa troj-kanálové RGB snímky na jedno-kanálové podľa známeho vzťahu 4.1. Prevod je možné vykonať ešte pred prenosom snímky na grafickú kartu, vďaka čomu stačí preniesť iba štvrtinu dát (OpenCL nepodporuje 24-bitové troj-kanálové obrázky, preto by bol nevyhnutný prevod do štvor-kanálového RGBA formátu).

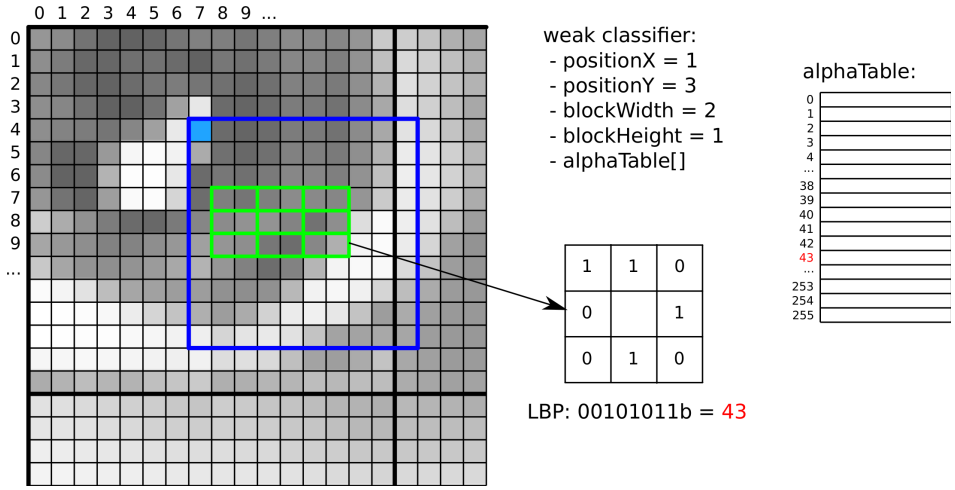
$$L = 0.30 \cdot R + 0.59 \cdot G + 0.11 \cdot B \tag{4.1}$$

Po prevode snímky na odtiene šedi sa skonštruuje pyramída podvzorkovaných obrázkov (viď obrázok 4.4). Aj keď v tejto reprezentácií nie sú využité všetky pixely, po ich vyplnení rovnakou farbou je klasifikátor schopný vylúčiť takéto pozície po vyhodnotení veľmi malého počtu slabých klasifikátorov a dopad na rýchlosť detekcie je teda minimálny [5].

Obr. 4.4: Obrazová pyramída (pôvodný snímok je vľavo)

4.4 Detekcia objektov

Akonáhle je vstup pripravený na samotnú detekciu, rozdelí sa pyramídový snímok na bloky, ktoré sa budú samostatne spracovávať vrámci kernelov. Pracovná skupina spracovávajúca jeden blok pozostáva z pracovných položiek, kde každá začne vyhodnocovať silný klasifikátor na inej pozícií daného bloku, pričom globálny index pracovnej položky korešponduje ľavému hornému rohu detekčného okna. Každý kernel vykonáva algoritmus 4.1, v ktorom prebieha vyhodnocovanie jednotlivých slabých klasifikátorov v cykle ako je naznačené na obrázku 4.5. Po vyhodnotení slabého klasifikátoru sa získa hodnota α, ktorá sa pričíta k celkovej odozve silného klasifikátoru. Pokiaľ je táto hodnota nižšia ako θ^t, detekovaný objekt sa v danom detekčnom okne nenachádza a aktuálna pozícia detekčného okna sa vyradí zo zoznamu kandidátnych.



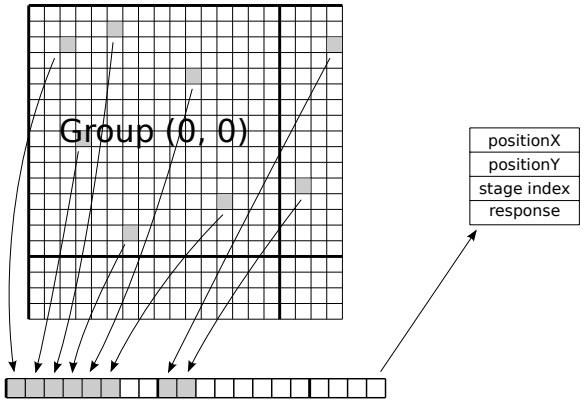
Obr. 4.5: Znázornenie vyhodnocovania slabého klasifikátoru pre pracovnú položku s lokálnym indexom [7, 4] (modrou farbou je znázornené detekčné okno zodpovedajúce tejto pracovnej položke)

Algoritmus 4.1 Detekčný kernel #1

1: $r \leftarrow 0$
2: $p \leftarrow$globálny index pracovnej položky
3: **for** $t = 1 \rightarrow N$ **do**
4: $\quad \alpha \leftarrow$ vyhodnoť slabý klasifikátor h_t pre pozíciu p
5: $\quad r \leftarrow r + \alpha$
6: $\quad$ **if** $r < \theta_t$ **then**
7: $\quad\quad$ vyraď danú pozíciu zo zoznamu kandidátnych a ukonči cyklus
8: $\quad$ **end if**
9: $\quad$ **if** počet kandidátnych pozícií v skupine $\leq$ maximálny počet **then**
10: $\quad\quad$ ukonči cyklus
11: $\quad$ **end if**
12: **end for**
13: ulož kandidátne pozície do pamäte

Vyššie uvedený cyklus pokračuje až kým slabé klasifikátory nevyradia určitý počet kandidátnych pozícií, vtedy sa zostávajúce pozície spolu s indexom nasledujúceho slabého kla-

sifikátoru a doterajšou odozvou uložia do globálnej pamäte, čo je znázornené na obrázku 4.6. Po tomto kroku je možné kernel ukončiť a spustiť ďalší, ktorý bude pracovať s menším počtom kandidátnych pozícií. Funkcia takéhoto kernelu bude takmer identická s prvým detekčným kernelom, s výnimkou odlišných vstupných dát, ktoré bude nutné načítať z pamäte, narozdiel od ich priameho odvodenia z globálnych indexov pracovných položiek. Naviac, keďže vstup aj výstup bude totožný, je možné spustiť takýto kernel n-krát.



Obr. 4.6: Ukončenie prvého stupňa detekcie a uloženie zostávajúcich kandidátnych pozícií do pamäte

Počet kandidátnych pozícií klesá vyhodnotením každého ďalšieho slabého klasifikátoru, avšak ich nízky počet by viedol k veľmi malému využitiu výpočetných prostriedkov grafického akcelerátoru. Preto budem uvažovať, že pre silný klasifikátor H_T platí vzťah 4.2, a teda po vyhodnotení určitého počtu slabých klasifikátorov nie je nutné kontrolovať hodnotu θ_t a vyraďovať kandidátne pozície na jej základe. Vďaka tomu je možné pre dokončenie detekcie použiť kernel implementujúci algoritmus 4.2. Každá z pracovných skupín vykonávajúca tento kernel bude spracovávať jednu zo zostávajúcich kandidátnych pozícií a pracovné položky budú predstavovať jednotlivé slabé klasifikátory, ktoré je ešte potrebné vyhodnotiť. Po dokončení tohto kernelu sčíta každá skupina hodnoty α jednotlivých pracovných položiek, a túto hodnotu pričíta k doterajšej sume uloženej v pamäti, čím sa získa výsledná hodnota odozvy silného klasifikátoru.

$$\exists i, i \in \{1, ..., T\}, \theta_i \leq -\infty \wedge \forall j > i, \theta_j \leq \theta_i \tag{4.2}$$

Algoritmus 4.2 Detekčný kernel #2

1: $p, i, r \leftarrow$ načítaj pozíciu, index slabého klasifikátoru a odozvu z pamäte
2: $t \leftarrow i +$ lokálny index pracovnej položky
3: $\alpha \leftarrow$ vyhodnoť slabý klasifikátor h_t pre pozíciu p
4: **for all** položky z danej skupiny **do**
5: $\quad r \leftarrow r + \alpha$
6: **end for**
7: aktualizuj hodnotu odozvy r v pamäti

Akonáhle sú výsledné odozvy známe, prenesú sa dáta späť do hlavnej pamäte, kde sa spracujú a na základe prahu γ sa rozhodne o prítomnosti objektu v detekčnom okne na jednotlivých kandidátnych pozíciách.

Kapitola 5

Implementácia

Táto kapitola sa bude bližšie venovať implementácií samotnej aplikácie a objasní detaily, vďaka ktorým bolo možné dosiahnuť vysokú rýchlosť detekcie na grafickom akcelerátore.

Aplikácia je implementovaná v jazyku C za pomoci knižníc *glib*, ktorá poskytuje základné funkcie ako jednoduchú prácu s reťazcami, multiplatformnú implementáciu práce so súbormi a zdrojmi udalostí, jednoduché spracovanie XML súborov a iné. Taktiež som využil knižnicu *OpenCV*, ktorá umožňuje načítanie jednotlivých snímkov z videa alebo kamery pripojenej k počítaču a ich ďalšie spracovanie. Samozrejmosťou je použitie knižnice OpenCL. Kritériom tiež bolo, aby výsledná aplikácia fungovala aj na zariadeniach, ktoré nepodporujú všetky možnosti OpenCL - napríklad zdieľanie kontextu s *OpenGL*.

Jednotlivé podkapitoly ozrejmia beh aplikácie na hostiteľskom počítači, vysvetlia usporiadanie dát v pamäti aby bolo možné s nimi efektívne pracovať, a ukážu samotné mapovanie paralelného algoritmu navrhnutého v kapitole 4.4 na OpenCL kernely.

5.1 Hostiteľská časť

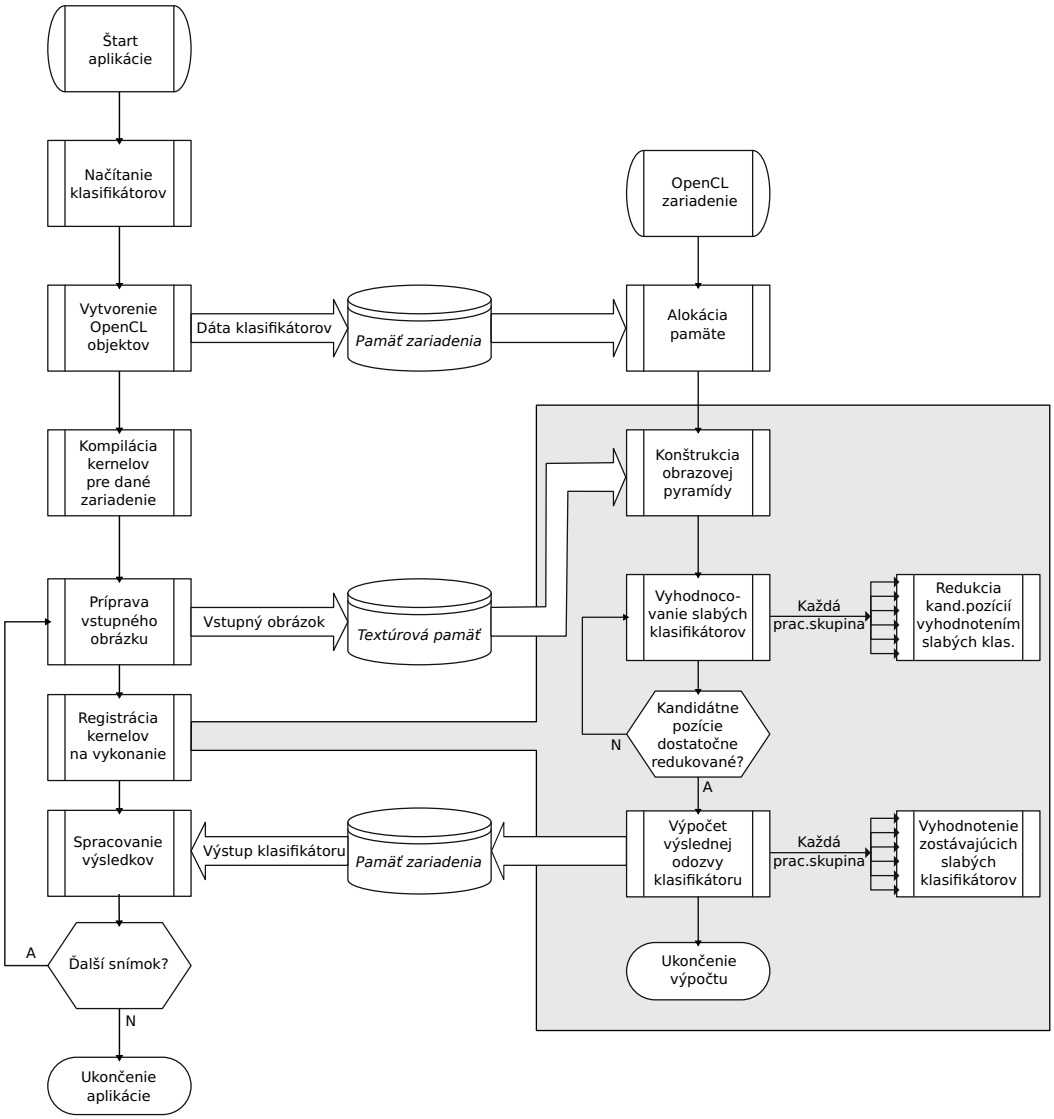
Napriek tomu, že celý detekčný algoritmus je spustený na grafickom akcelerátore, beh aplikácie je stále kontrolovaný hostiteľským počítačom, a preto je dôležité správne navrhnúť aj túto časť programu. Kompletný tok aplikácie je znázornený na obrázku 5.1. Táto podkapitola sa ďalej zaoberá detailnejším popisom implementácie jednotlivých procesov vykonávaných na hostiteľskom počítači s dôrazom na použité volania aplikačného programovacieho rozhrania OpenCL.

Po spustení aplikácie sú spracované parametre príkazového riadku, kde je špecifikované vstupné video pre samotnú detekciu, parametre detekčného algoritmu, XML súbor s natrénovaným klasifikátorom atď. V prípade, že nie je špecifikovaný klasifikátor, použije sa interný klasifikátor, ktorého dáta sú súčasťou samotnej aplikácie.

Keď je pripravený klasifikátor, vytvoria sa jednotlivé OpenCL objekty - najskôr sa nájdu nainštalované platformy volaním `clGetPlatformIDs` a vrámci nich dostupné zariadenia pomocou `clGetDeviceIDs`. Potom, čo je známe zariadenie s ktorým sa bude pracovať, je vytvorený OpenCL kontext:

```
context = clCreateContext (properties, 1, &device, NULL, NULL, &error);
```

Kontext používa OpenCL *runtime* na spravovanie pamäte, fronty príkazov, programových objektov a kernelov, a zároveň na vykonávanie kernelov na zariadeniach špecifikovaných pri jeho vytvorení. Ďalší krok je teda vytvorenie jednotlivých pamäťových objektov,



Obr. 5.1: Graf toku aplikácie - ľavá časť predstavuje procesy na hostiteľskom počítači a pravá časť znázorňuje procesy vykonávané OpenCL zariadením

ktoré budú reprezentovať slabé klasifikátory, vstupný snímok, pyramídový obrázok a výstupné dáta klasifikátorov. Tieto objekty vytvoríme volaním `clCreateImage2D`, poprípade `clCreateBuffer`, pričom dôležité sú príznaky, ktoré použijeme pre vytvorenie jednotlivých objektov - napríklad príznak `CL_MEM_READ_ONLY` značí, že pamäťový objekt je možné umiestniť do pamäte konštánt (samozrejme iba pokiaľ veľkosť tohto objektu nepresahuje celkovú veľkosť konštantnej pamäte). Pamäťový objekt pre slabé klasifikátory teda vytvoríme:

```
flags = CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR;
classifiers = clCreateBuffer (context, flags, stages_size, stages,
                              &error_code);
```

Nutnosťou je tiež vytvorenie fronty do ktorej sa budú vkladať príkazy na čítanie a zápis

z/do pamäte zariadenia a samozrejme na samotné vykonanie kernelov:

```
queue = clCreateCommandQueue (context, device, properties, &error);
```

Ďalším krokom je načítanie zdrojového súboru s kernelmi, pomocou ktorého vytvoríme programový objekt:

```
program = clCreateProgramWithSource (context, 1, &file_contents,
                                     NULL, &error_code);
```

Predtým ako môžeme získať objekty reprezentujúce jednotlivé kernely, je nutné najskôr zostaviť program, ktorý sme práve vytvorili. Naviac pri samotnom zostavení môžeme určiť hodnoty konštánt, typy s ktorými budú kernely pracovať, zapínať alebo vypínať časti kernelov a iné možnosti definovaním rôznych makier a tieto predať ako parametre pre zostavenie programu:

```
options = "-D STEP_ONE_SIZE_DIVISOR=8";
...
clBuildProgram (program, 1, &device, options, NULL, NULL);
```

Vďaka takémuto dynamickému zostaveniu môže kompilátor ďalej optimalizovať samotný program pre dané zariadenia. Pokiaľ je hostiteľská aplikácia skompilovaná s makrom *HAVE_GIO* (definícia tohto makra vyžaduje knižnicu *gio*[1]), monitoruje sa obsah zdrojového súboru s kernelmi a pri jeho zmene sa znovu zostaví celý program, čo umožňuje jednoduché testovanie zmien v kerneloch za behu aplikácie.

Po úspešnom zostavení programu je už možné získať objekty reprezentujúce jednotlivé kernely definované v zdrojovom súbore využitím funkcie `clCreateKernel` a nastaviť parametre kernelov pomocou funkcie `clSetKernelArg`.

```
lbp_kernel = clCreateKernel (program, "lbp_kernel", &error_code);
```

Potom ako sú pripravené kernely na vykonanie, nasleduje hlavný cyklus aplikácie. V ňom sa dekomprimuje snímok z videa pomocou funkcií knižnice *OpenCV* a získa sa RGB obrázok ktorý sa ďalej spracováva - najskôr sa prevedie na jedno-kanálový obrázok úrovní šedi a následne sa tiež vytvorí časť pyramídového obrázku, ako je znázornené na obrázku 5.2. Dôvodom vytvorenia polovice pyramídy na CPU je zjednodušenie kernelu, ktorý vytvára pyramídu na grafickom akcelerátore. Keďže kernel bude mať k dispozícií iba obrázok v plnom rozlíšení, bilineárnou interpoláciou je možné jednoducho vytvoriť iba obrázok polovičného rozlíšenia, teda pre vytvorenie celej pyramídy by bolo nutné manuálne interpolovať hodnoty alebo invokovať kernel viac-krát, pričom obe varianty by boli časovo náročné. Naviac, keďže sa vytvára iba menšia časť pyramídy, aj CPU zvládne túto operáciu dostatočne rýchlo.

Oba tieto obrázky sa prenesú na grafický akcelerátor volaním `clEnqueueWriteImage`. Akonáhle sú obrázky prenesené, zaregistrujú sa jednotlivé kernely na vykonanie, pričom prvý kernel doplní chýbajúce stupne pyramídy, čím vytvorí kompletnú pyramídu, ktorú je možné vidieť na obrázku 4.4, a ostatné vykonajú detekciu objektov. Individuálne kernely sa zaradia do fronty príkazov volaním:

[1] Knižnica *gio* je súčasťou knižnice *glib* od verzie 2.14.

```
clEnqueueNDRangeKernel (queue, kernel_obj, num_dimensions, NULL,
                        global_size, local_size, 0, NULL, &event);
```

Obr. 5.2: Dva obrázky, ktoré sa prenášajú na grafický akcelerátor pred detekciou

Keď sa ukončí vykonávanie všetkých kernelov, prenesú sa dáta s výstupom klasifikátoru späť do hlavnej pamäte (pomocou funkcie `clEnqueueReadBuffer`), kde sa spracujú - súradnice pyramídy sa mapujú na oblasti vstupného snímku, a tento sa zobrazí alebo uloží do výstupného videa. Pokiaľ vstupné video obsahuje ďalšie snímky, celý cyklus sa opakuje, inak sa aplikácia ukončí.

Jednu z optimalizácií je možné vykonať v hlavnom cykle - keďže väčšina OpenCL funkcií pracuje asynchrónne, je možné využiť čas, kedy sa čaká na dokončenie kernelov a prenos výstupných dát, na dekódovanie ďalšieho snímku a jeho spracovanie, keďže tieto operácie sú značne časovo náročné. Takto je možné dosiahnuť omnoho vyšší počet spracovaných snímkov za sekundu.

5.2 Usporiadanie dát v pamäti

Veľmi dôležitým faktorom pre rýchle spracovanie dát je tiež efektívne usporiadanie dát v pamäti. V tejto podkapitole je vysvetlené aké usporiadanie bolo zvolené pre jednotlivé typy dát s ktorými detekčný algoritmus pracuje.

Slabé klasifikátory

V kapitole 4.2 bolo spomenuté, že každý slabý klasifikátor obsahuje nasledujúce dáta:

- posunutie v detekčnom okne
- veľkosť bloku z ktorého sa vypočítavajú príznaky
- tabuľka hodnôt α

- prah θ pre predčasné ukončenie detekcie

Analýzou detekčného algoritmu môžeme vidieť, že všetky pracovné položky pristupujú k rovnakým hodnotám slabého klasifikátoru okrem hodnôt α, ku ktorým sa pristupuje náhodne. Preto je dobré oddeliť tabuľky α hodnôt od ostatných vlastností klasifikátoru. Dátová štruktúra ktorá bude obsahovať slabý klasifikátor musí teda zahŕňať štyri celočíselné premenné a jedno desatinné číslo. Grafické akcelerátory dokážu najefektívnejšie pracovať s pamäťovými oblasťami o veľkosti 16 bajtov, pričom premenné typu *int* ako aj *float* vyžadujú 4 bajty a typ *short* 2 bajty. Preto je najvýhodnejšia reprezentácia slabého klasifikátoru:

```
typedef struct _TStage {
  int offsetX, offsetY;
  short blockWidth, blockHeight;
  float theta;
} TStage;
```

Najčastejšie používané silné klasifikátory obsahujú približne 1000 slabých klasifikátorov, čo znamená, že celková veľkosť pamäte nutnej na uchovanie všetkých slabých klasifikátorov je menej ako 16kB, a keďže OpenCL špecifikácia definuje veľkosť konštantnej pamäte minimálne 64kB, máme príležitosť uložiť klasifikátory do konštantnej pamäte, do ktorej sa pristupuje cez vyrovnávaciu pamäť a poskytuje teda vysokú šírku pásma.

Tabuľky α hodnôt sú značne väčšie (jeden megabajt pre 1000 slabých klasifikátorov), preto ich nie je možné umiestniť do konštantnej pamäte. Zostáva nám teda použitie globálnej alebo textúrovej pamäte. Globálna pamäť nevyužíva vyrovnávaciu pamäť, a keďže prístup do tabuľky do istej miery zachováva lokalitu (rôzne pracovné položky vyhodnocujúce rovnaký slabý klasifikátor pristupujú do toho istého riadku tabuľky v prípade, že použijeme usporiadanie 256×1000 - teda jedna tabuľka má rozmery 256×1), je vhodné použiť textúrovú pamäť. Textúru, ktorá bude obsahovať desatinné čísla je možné vytvoriť použitím jedno-kanálového obrázkového formátu:

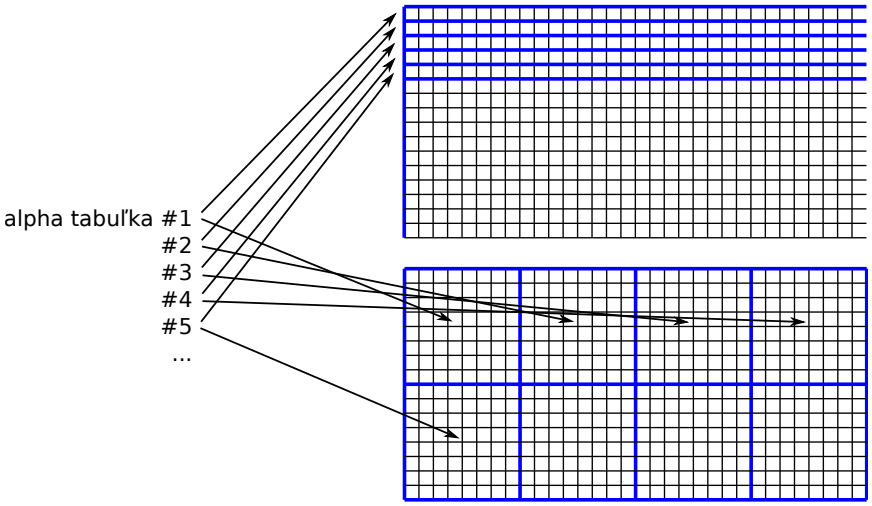
```
cl_image_format format = { CL_INTENSITY, CL_FLOAT };
```

Takýto formát síce patrí do nepovinných v OpenCL špecifikácií, avšak dostupné OpenCL implementácie dodávateľov grafických akcelerátorov ho podporujú.

Experimentálne som implementoval aj usporiadanie, kde α hodnoty jedného slabého klasifikátoru boli uložené v blokoch o veľkosti 16×16 a celá textúra mala veľkosť 160×1600, avšak ukázalo sa, že takéto usporiadanie neprináša výrazne vyššiu úspešnosť textúrovej vyrovnávacej pamäte. Naviac mapovanie súradníc pri použití tohto usporiadania znamenalo nutnosť vykonávania väčšieho počtu inštrukcií a preto bol takýto prístup celkovo menej efektívny oproti jednoduchému 256×1 usporiadaniu. Rovnaký problém malo taktiež usporiadanie 512×500, a preto som nakoniec použil textúru, kde jeden riadok predstavuje jednu tabuľku (teda variantu 256×1000). Jednotlivé usporiadania je možné vidieť na obrázku 5.3.

Výstupné dáta klasifikátoru

Veľmi dôležitými sú tiež dáta, ktoré sa ukladajú do pamäte grafického akcelerátoru pred dokončením kernelov, a slúžia na výmenu informácií medzi jednotlivými kernelmi, ako aj

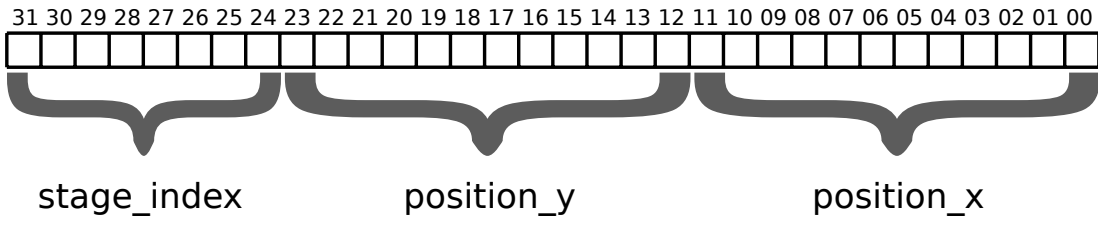


Obr. 5.3: Jedny z možných usporiadaní tabuliek α hodnôt v textúre

medzi grafickým akcelerátorom a hostiteľskou aplikáciou. Tieto dáta musia obsahovať:

- pozíciu detekčného okna v obrázku
- index slabého klasifikátoru, ktorý treba najbližšie vyhodnotiť
- doterajšiu sumu hodnôt α

Vzhľadom na to, že načítanie dát z globálnej pamäte trvá stovky cyklov na grafickom akcelerátore, je vhodné minimalizovať ich veľkosť. Preto som sa rozhodol využiť na všetky tieto dáta iba 8 bajtov pamäte, pričom 4 bajty spotrebuje suma α hodnôt, a teda zvyšné 4 bajty (32bitov) využívam na uloženie pozície a aj indexu ďalšieho slabého klasifikátoru. Toto som dosiahol zavedením obmedzení na veľkosť pyramídového obrázku a počtu klasifikátorov ktoré vykonávajú kernel popísaný algoritmom 4.1. Na súradnice pozície som vyhradil 24bitov (12bitov na zložku), kvôli čomu vzniká obmedzenie na maximálnu veľkosť pyramídového obrázku 4096×4096 pixelov, čo však znamená že algoritmus je stále schopný spracovávať videá vo *Full HD* rozlíšení (*Full HD* značí videá s rozlíšením 1920×1080, pričom po skonštruovaní obrazovej pyramídy bude mať snímok veľkosť približne 1920×3940 pixelov). Na index slabého klasifikátoru teda zostáva 8 bitov, čo spresňuje obmedzenie uvedené vo vzťahu 4.2 - $i \in \{1, ..., 255\}$. Rozdelenie bitov je možné vidieť na obrázku 5.4.



Obr. 5.4: Rozdelenie bitov na uloženie jednotlivých hodnôt výstupu klasifikátoru

5.3 Mapovanie algoritmu na OpenCL kernely

Na vykonanie detekčných algoritmov navrhnutých v kapitole 4.4 boli implementované štyri kernely v jazyku *OpenCL C*. Prvé dva kernely (`red_kernel_step1` a `red_kernel_step2`) vykonávajú algoritmus 4.1, pričom prvý pracuje priamo so vstupným obrázkom a globálne indexy pracovných položiek zodpovedajú súradniciam ľavého horného rohu detekčného okna. Druhý kernel už načítava kandidátne pozície a ostatné dáta z pamäte, takže tento kernel je možné spustiť n-krát a predstavuje hlavný redukčný kernel. Tretí kernel (`red_kernel_step3`) funguje podobne ako druhý, avšak jeho cieľom je zaručiť, že všetky kandidátne pozície majú vyhodnotený rovnaký počet slabých klasifikátorov, čím ich pripraví na vykonanie posledného kernelu (`red_kernel_step4`), ktorý implementuje algoritmus 4.2.

Každý z prvých troch kernelov má definované makro (`STEP_ONE_SIZE_DIVISOR`, ...), ktoré určuje, ako bude daný kernel redukovať kandidátne pozície. Práve preto, že sú tieto hodnoty pevne definované, je možné na CPU zaregistrovať spustenie všetkých kernelov naraz a nie je nutná ďalšia komunikácia pred dokončením výpočtu. Napríklad pre prvý kernel je východzia hodnota tohto makra 8, čo znamená, že kernel bude vyhodnocovať slabé klasifikátory, až kým nezostane len 1/8 kandidátnych pozícií (teda pre pracovnú skupinu veľkosti 256 položiek zostane 32 kandidátnych pozícií). Samozrejme môže sa stať, že po vyhodnotení niekoľkých slabých klasifikátorov zostane menej kandidátnych pozícií ako je daný pomer, a v tom prípade sa uloží do pamäte hodnota 0, ktorá by značila kandidátnu pozíciu so súradnicami [0, 0] a indexom ďalšieho slabého klasifikátoru 0 (viď obrázok 5.4), lenže prvý kernel vždy vyhodnotí minimálne jeden slabý klasifikátor, a preto sa takáto hodnota nemôže za iných okolností v pamäti vyskytnúť. Napriek tomu, že sú redukčné pomery určené makrami, hostiteľská aplikácia môže meniť ich hodnoty pri zostavovaní programu ako bolo ukázané v kapitole 5.1, takže ich hodnoty závisia na parametroch predaných hostiteľskej aplikácií.

Jedným z problémov prvých dvoch kernelov, ktoré implementujú algoritmus 4.1, je nutnosť zisťovania počtu zostávajúcich kandidátnych pozícií. Táto operácia vyžaduje jednak využitie atomických inštrukcií v lokálnej pamäti, ako aj synchronizáciu položiek vrámci celej pracovnej skupiny, ktorý je v jazyku *OpenCL C* implementovaný volaním funkcie:

```
barrier (CLK_LOCAL_MEM_FENCE);
```

Synchronizácia celej pracovnej skupiny po vyhodnotení každého slabého klasifikátoru by bola veľmi pomalá, preto som pridal parameter pre tieto kernely, pomocou ktorého sa vyhodnocovanie rozdeľuje na dve fázy - najskôr sa vyhodnotí stanovený počet slabých klasifikátorov a až následne prebieha synchronizácia medzi vyhodnocovaním. Zavedenie tohto parametru poskytuje jednoduchú možnosť ako vyladiť výkon celej detekcie.

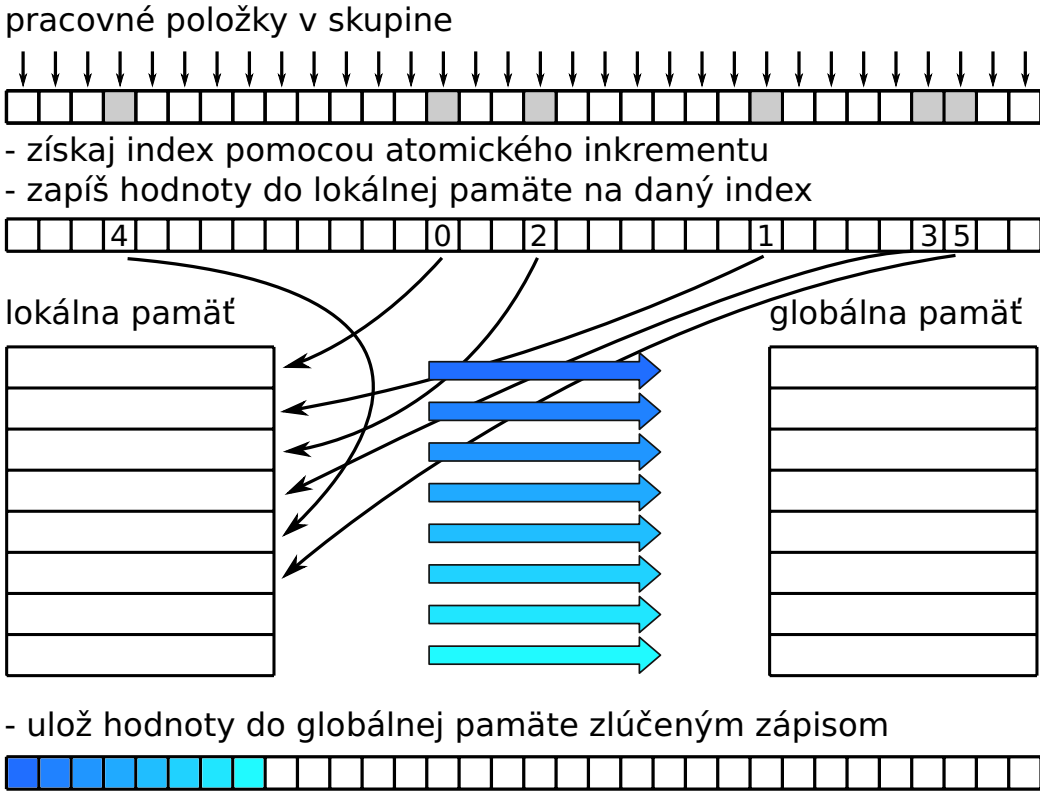
Aby bola zachovaná kompatibilita so zariadeniami ktoré nepodporujú OpenCL rozšírenie pre atomické inštrukcie v lokálnej pamäti, teda napríklad grafické karty značky NVIDIA podporujúce iba *CUDA Compute Capability 1.1*, bola tiež implementovaná metóda, ktorá nahrádza atomický inkrement využitím nízko-úrovňových vlastností takýchto grafických akcelerátorov - konkrétne využitie kombinovania zápisu keď pracovné položky jedného *warpu* zapisujú rôzne hodnoty do rovnakej pamäťovej bunky. Na riadku číslo 6 algoritmu 5.1 vidno, že všetky položky z *warpu* sa snažia zapísať do rovnakej bunky pamäte, avšak hardvér zabezpečí, že iba jedna z hodnôt bude zapísaná. Takáto funkcia je pritom niekoľko-násobne rýchlejšia ako využitie atomického inkrementu v globálnej pamäti.

Všetky implementované kernely používajú zlúčené načítanie aj zápis do pamäte, s výnimkou posledného kernelu, kde sa s východzím nastavením počíta odozva iba pre jednu

Algoritmus 5.1 Náhrada atomického inkrementu

```
1: t ← bunka lokálnej pamäte patriaca danému warpu
2: tag ← ID vlákna vrámci warpu bitovo posunuté vľavo
3: repeat
4:     count ← spodné bity bunky t (bez ID vlákna)
5:     count ← (count + 1) | tag
6:     t ← count
7: until t = count
```

pozíciu vrámci celej pracovnej skupiny. Zlúčený zápis je riešený tým spôsobom, že po redukcií na daný počet kandidátnych pozícií sa pomocou atomického inkrementu (poprípade algoritmu podobnému 5.1) priradia jednotlivým pracovným položkám indexy do lokálnej pamäte na ktoré sa uložia kandidátne pozície a následne je možné previesť zlúčený zápis. Princíp tohto spôsobu je zobrazený na obrázku 5.5.



Obr. 5.5: Zlúčený zápis zostávajúcich pozícií (pracovné položky, ktoré ich vyhodnocovali sú znázornené šedou farbou) po redukcií

Pri práci s lokálnou pamäťou používajú jednotlivé pracovné položky pre prístup k pamäti krok s veľkosťou 4 bajty, takže nedochádza ku konfliktom v pamäťových bankách.

Kapitola 6

Dosiahnuté výsledky

V tejto kapitole popisujem kroky, ktoré boli vykonané na vyladenie rýchlosti OpenCL detektoru, experimentálne vyhodnotím celkový výkon implementácie a porovnám ho so známymi optimalizovanými implementáciami pracujúcimi na CPU.

Aplikácia bola testovaná na počítači s konfiguráciou uvedenou v tabuľke 6.1, okrem samotného grafického akcelerátoru uvádzam aj verziu grafického ovládača, ktorý obsahuje samotný kompilátor jazyka *OpenCL C* a zároveň aj implementáciu hostiteľského rozhrania OpenCL. Aplikácia nebola testovaná na grafických kartách ATI, preto nie je zaručený optimálny výkon aj na týchto zariadeniach.

CPU (počet jadier)	Intel Core2 Duo E8200 @ 2.66GHz (2)
Veľkosť pamäte	2 GB
GPU (počet výpočetných jednotiek)	NVIDIA GeForce GTX 285 (30)
Veľkosť GPU pamäte	2 GB
Verzia ovládača GPU	270.41.06

Tabuľka 6.1: Konfigurácia testovacieho systému

6.1 Výber redukčných parametrov

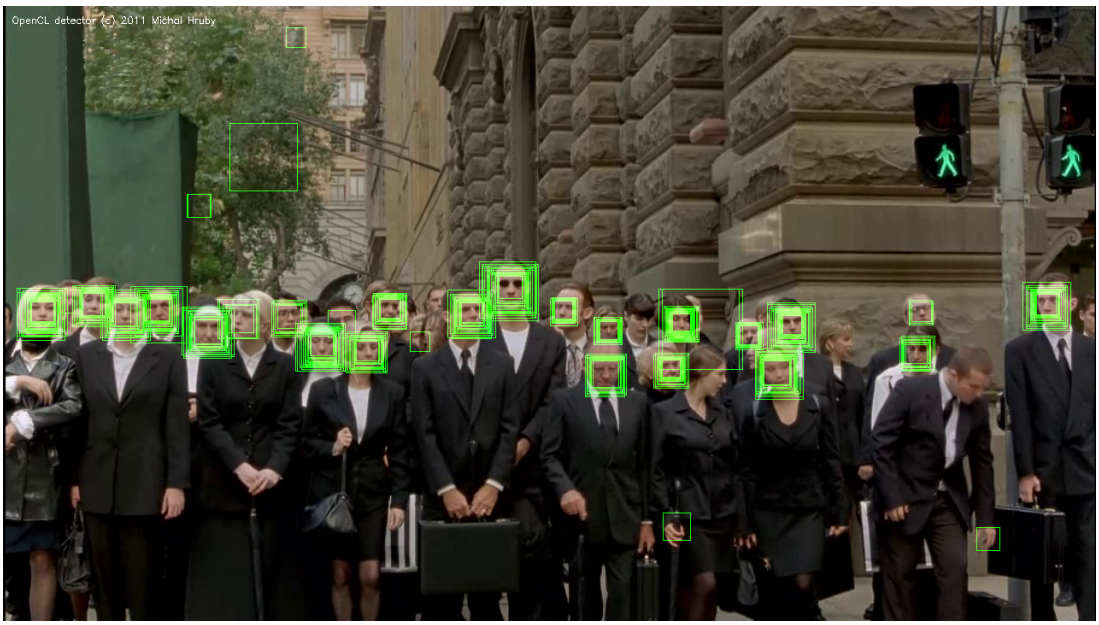
Jedným z najdôležitejších parametrov implementovaného algoritmu sú práve hodnoty redukčných parametrov. Ich výber som realizoval analýzou chovania WaldBoost klasifikátoru (zobrazený na obrázku 4.1) a testovaním klasifikátoru na videách rôznych veľkostí. Testovanie ukázalo, ako vidno v tabuľke 6.2, že po vyhodnotení všetkých slabých klasifikátorov zostala v priemere približne 1/14000 všetkých pozícií klasifikovaných ako objekt danej triedy. Aby sa však nestrácali niektoré kandidátne pozície je vhodnejšie založiť odhad na základe maximálneho pomeru, ktorý je približne 1/4600. Netreba však zabudnúť, že tieto pomery sú globálne pre celý snímok videa, pričom implementovaný algoritmus paralelne spracováva jednotlivé bloky, a teda pamäť vyhradená na ukladanie kandidátnych pozícií musí byť väčšia. Preto som ako východzí celkový pomer redukcie zvolil 1/512, pričom tento pomer je možné jednoducho meniť a aplikácia má naviac špeciálny mód v ktorom je možné zistiť koľko kandidátnych pozícií bolo vyradených práve kvôli príliš agresívnej redukcií a tým pádom nedostatku pamäte vyhradenej pre uloženie pozícií.

Testovaním tohto redukčného pomeru sa ukázalo, že v ojedinelých prípadoch stále môže nastať situácia, kedy sa niekoľko kandidátnych pozícií vyradí. Tento prípad však nastáva

	pomer kandidátnych pozícií ku všetkým možným pozíciám detekčného okna	
počet vyhodnotených slabých klasifikátorov	priemerný	maximum
1	0.19280335	0.26000109
10	0.01130559	0.02106337
80	0.00006931	0.0002181

Tabuľka 6.2: Testovanie redukčných vlastností klasifikátoru ($\alpha = 0.2$) na rôznych videách

iba v scénach s veľkým počtom detekcií ktoré sú blízko pri sebe a naviac v rovnakej úrovni obrazovej pyramídy. Príklad takéhoto snímku je zobrazený na obrázku 6.1. V tomto snímku bolo zahodených 11 kandidátnych pozícií, pričom celkovo bolo klasifikátorom vyhodnotených 726 pozícií ako obsahujúce hľadaný objekt. Z princípu funkcie klasifikátoru však plynie, že v oblastiach kde sa nachádza hľadaný objekt bude viacero pozitívnych odoziev, preto takýto nízky počet vyradených pozícií nemá žiadny vplyv na úspešnosť detekcie, teda redukčný pomer 1/512 je vhodný ako východzia hodnota.



Obr. 6.1: Snímok z videa znázorňujúci okná v ktorých mal klasifikátor pozitívnu odozvu

Ako bolo spomenuté v kapitole 5.3, redukcia je vykonávaná viacerými kernelmi, a preto je potreba vybrať správne parametre redukcie pre každý z nich, pričom treba mať na pamäti, že príliš vysoký redukčný pomer v niektorom z nich zvýši pravdepodobnosť zahodenia niektorých kandidátnych pozícií a naopak vyhodnocovanie vysokého počtu kernelov spomalí celú detekciu. Preto bolo nutné nájsť vhodné rozdelenie redukcie, pri zachovaní celkového redukčného pomeru 1/512.

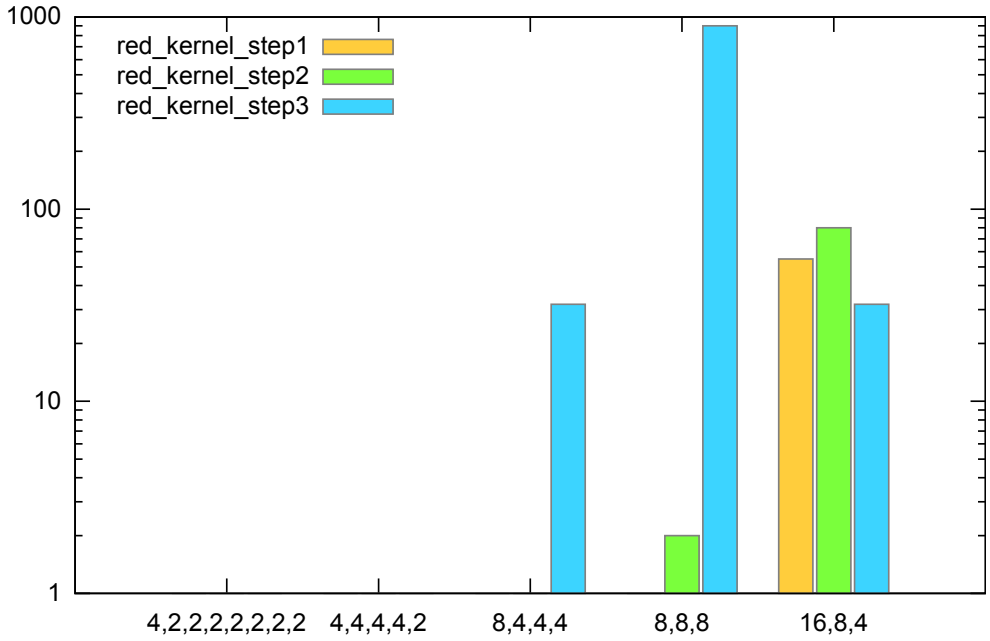
Niektoré možnosti rozdelenia sú vyhodnotené v tabuľke 6.3, kde testovanie prebiehalo práve na videu zostavenom z tisícky snímkov podobných 6.1, aby sa ukázal vplyv redukčných parametrov na počet vyradených pozícií. Ako vidno v tabuľke, čím viac kernelov

bolo použitých a mali teda nižšie redukčné konštanty, tým bolo menej pozícií zahodených. Na obrázku 6.2 je naviac vidieť presne ktorý kernel vyradil koľko kandidátnych pozícií. Z týchto výsledkov je zrejmé, že použitím viacerých kernelov sa správanie klasifikátoru viac blíži globálnemu správaniu z obrázku 4.1. Zároveň, ako východzie hodnoty pre redukčné konštanty som zvolil 8, 4, 4, 4, ktoré poskytujú veľmi dobrú rovnováhu medzi počtom vyradených pozícií a časom potrebným na vyhodnotenie všetkých kernelov.

počet kernelov	redukčné konštanty[a]	zahodené pozície	priemerný GPU čas[μs][b]
8	4, 2, 2, 2, 2, 2, 2, 2	0	8066
5	4, 4, 4, 4, 2	0	7516
4	8, 4, 4, 4	32	6874
3	8, 8, 8	901	7551
3	16, 8, 4	167	8494

[a]Redukčné konštanty sú prevrátené hodnoty redukčných pomerov.
[b]Zahŕňa budovanie pyramídy, detekciu a režijný čas na komunikáciu s CPU medzi spustením jednotlivých kernelov.

Tabuľka 6.3: Hľadanie vhodného rozdelenia redukčných konštánt pre jednotlivé kernely



Obr. 6.2: Počet zahodených pozícií podľa kernelu pre rôzne veľkosti redukčných konštánt (kernelu *red_kernel_step*1 zodpovedá ľavý parameter redukčných konštánt, kernelu *red_kernel_step*3 pravý parameter, a ostatné náležia kernelu *red_kernel_step*2)

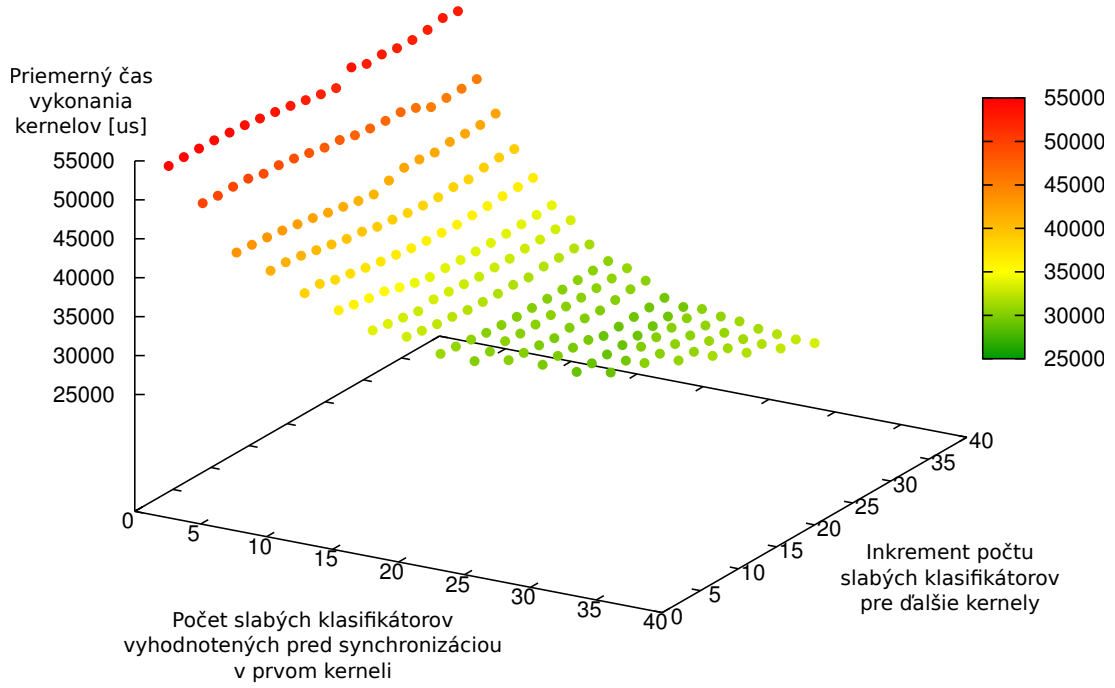
6.2 Vplyv synchronizácie na rýchlosť detekcie

Pri implementácií detekčných kernelov bol zavedený parameter, ktorý rozdeľuje vyhodnocovanie na dve fázy (viď kapitola 5.3) - fázu bez synchronizácie, kde sa vyhodnotí presný počet slabých klasifikátorov daný týmto parametrom a druhú fázu so synchronizáciou, kedy

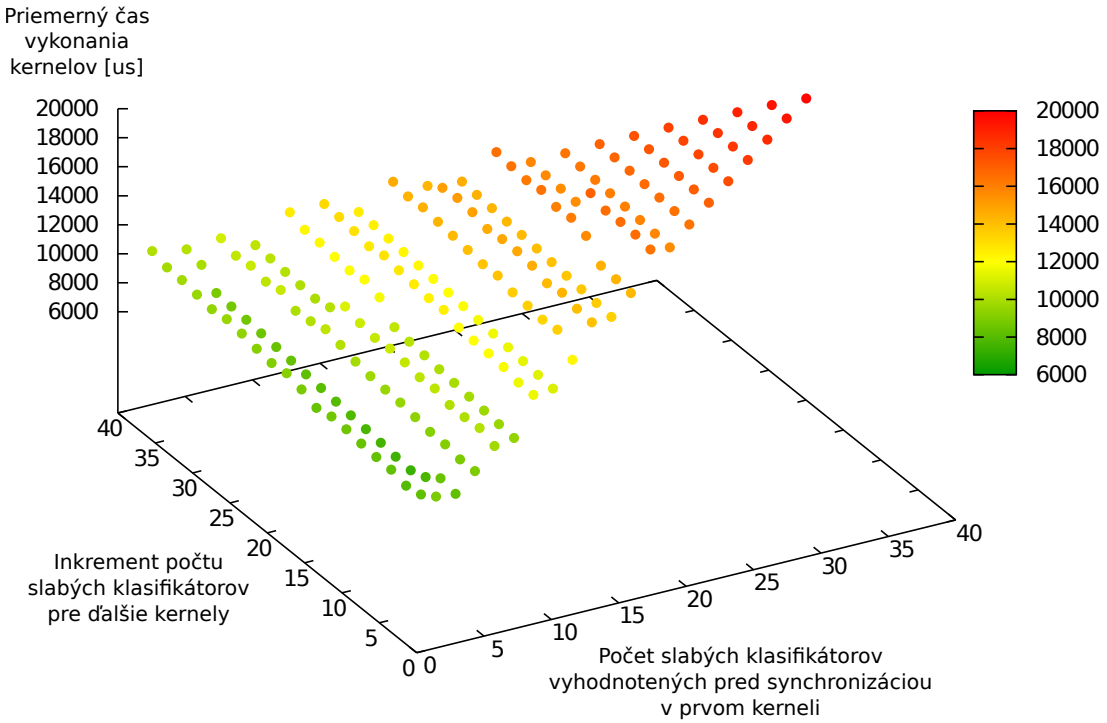
sa vyhodnocujú ďalšie slabé klasifikátory, kým počet zostávajúcich kandidátnych pozícií nedosiahne hodnotu určenú redukčným pomerom.

Mojim cieľom bolo nájsť optimálne hodnoty týchto parametrov, pričom kritériom bol čo najnižší priemerný čas vykonávania detekčných kernelov na snímok videa. Samozrejmosťou je, že hodnoty týchto parametrov sú závislé od použitého silného klasifikátoru, preto som hľadal optimálne parametre pre dva veľmi odlišné klasifikátory - jeden z nich bol natrénovaný s hodnotou α parametru (pomer falošných negatív) 0.01, a druhý 0.2, čo má zároveň veľký vplyv na rýchlosť vyhodnotenia jednotlivých klasifikátorov. Keďže pri použití štyroch redukčných kernelov je nutné zadať tri rôzne parametre, čo sa zle vynáša do grafu, redukoval som ich počet na dva tým spôsobom, že prvý parameter značí počet slabých klasifikátorov vyhodnotených v prvom kerneli a druhý parameter je inkrement tohto počtu pre ďalšie kernely. Výsledky testovania je možné vidieť na obrázkoch 6.3 a 6.4.

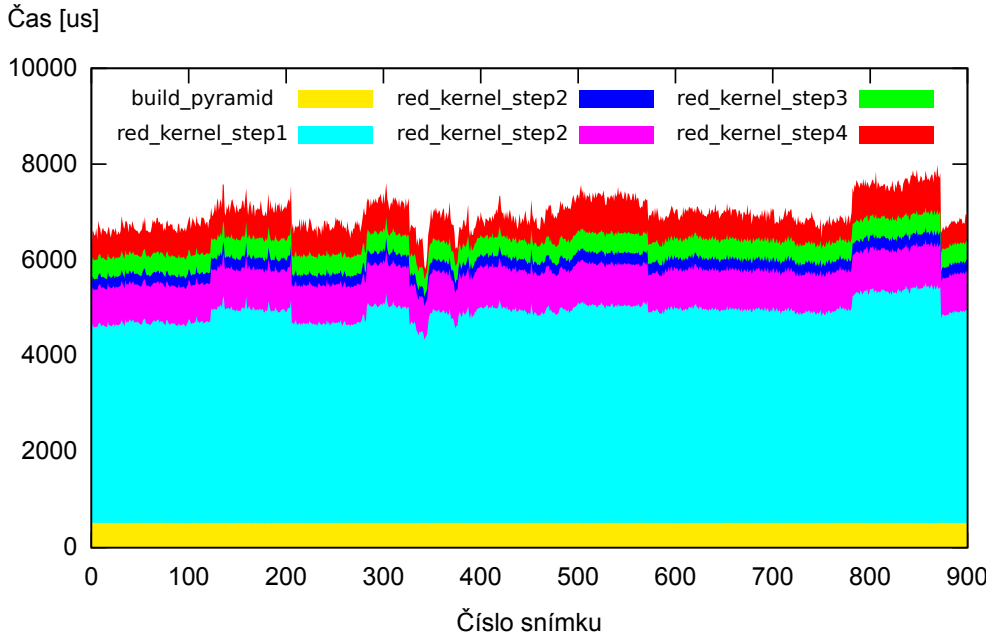
V grafoch sú zreteľné parametre minimálnej hodnoty priemerného času vykonávania kernelov, pričom je tiež vidno, že oveľa dôležitejší je počet slabých klasifikátorov vyhodnotených pred synchronizáciou v prvom kerneli ako táto hodnota pre ostatné kernely. Prečo je tomu tak vysvetľuje obrázok 6.5, ktorý znázorňuje čas potrebný na vykonanie jednotlivých kernelov pri spracovávaní videa. V tohto obrázku si môžeme všimnúť, že vykonanie prvého detekčného kernelu je časovo omnoho náročnejšie ako vykonanie ostatných kernelov. Pri používaní rozličných klasifikátorov je teda vhodné hľadať optimálnu hodnotu aspoň pre prvý kernel, inak riskujeme aj viac ako dvakrát horší výkon detektoru.



Obr. 6.3: Rýchlosť detekcie pre klasifikátor s $\alpha = 0.01$ pri použití rôznych parametrov ovplyvňujúcich synchronizáciu vrámci jednotlivých kernelov. Minimum sa nachádza v bode [26, 15], čo znamená, že detekcia je najrýchlejšia keď počet slabých klasifikátorov vyhodnotených pred synchronizáciou v prvom kerneli je 26, v druhom kerneli musí byť minimálny počet vyhodnotených klasifikátorov 41 a v treťom 56.



Obr. 6.4: Rýchlosť detekcie pre klasifikátor s $\alpha = 0.20$ pri použití rôznych parametrov ovplyvňujúcich synchronizáciu vrámci jednotlivých kernelov. Minimum sa nachádza v bode [4, 8].



Obr. 6.5: Čas potrebný na vykonanie jednotlivých kernelov pri spracovávaní snímkov videa (redukčný kernel *red_kernel_step*2 sa vykonáva dvakrát, preto sú oba behy znázornené inou farbou)

6.3 Porovnanie s optimalizovanou CPU implementáciou

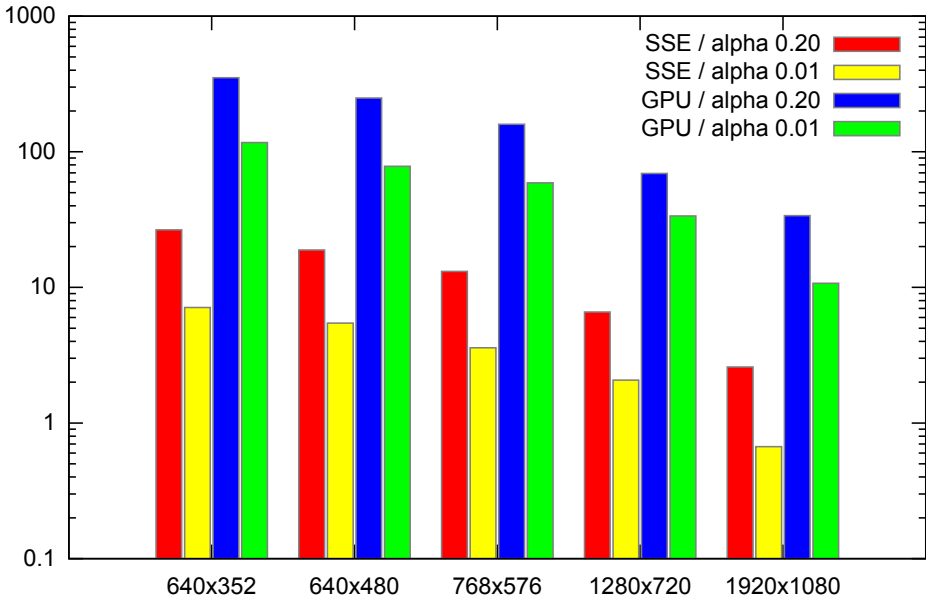
Potom, čo boli nájdené optimálne hodnoty jednotlivých parametrov algoritmu, je možné porovnať ho s existujúcimi optimalizovanými CPU implementáciami, ktoré využívajú SSE inštrukcie, ako bolo popísané napríklad v [6]. Takáto implementácia je voľne dostupná z adresy `http://www.fit.vutbr.cz/research/view_product.php.cs?id=107`.

Jasná výhoda OpenCL oproti SSE implementácií je, že má k dispozícií nielen CPU prostriedky, ale aj GPU, naopak nevýhoda, ktorú toto prináša, je nutnosť prenosu každého snímku po zbernici. V tabuľke 6.4 sú preto uvedené priemerné časy samotnej detekcie potrebné na vyhodnotenie jedného snímku pre videá rôznych veľkostí, bez operácií predspracovania a prenosu dát. Ako je z tabuľky vidno, detekcia je na GPU pätnásť až sedemnásť-krát rýchlejšia ako na CPU.

	Klasifikátor s $\alpha = 0.20$		Klasifikátor s $\alpha = 0.01$	
Veľkosť videa	SSE [ms]	GPU [ms]	SSE [ms]	GPU [ms]
640×352	32.9	2.1	135.5	7.9
640×480	46.8	2.9	177.5	11.9
768×576	68.3	4.0	270.2	15.8
1280×720	132.8	7.8	463.9	27.6
1920×1080	359.4	19.1	1474.9	87.7

Tabuľka 6.4: Porovnanie priemerného času detekcie (bez predspracovania) na rôznych videách

Pre získanie predstavy o cene operácií mimo samotnej detekcie je na obrázku 6.6 ukázaný priemerný počet snímkov spracovaných za sekundu jednotlivými implementáciami. Z tohto grafu vidno, že aj po započítaní režijných operácií je implementácia využívajúca GPU dvanásť až šestnásť-krát rýchlejšia na všetkých rozlíšeniach videa.



Obr. 6.6: Počet snímkov spracovaných za sekundu pre videá rôznej veľkosti pri použití rôznych klasifikátorov (všimnite si logaritmické merítko osy y)

6.4 Výsledky profilovania

V tabuľke 6.5 sú uvedené časy jednotlivých fáz behu OpenCL implementácie, spolu s hodnotami výsledného počtu spracovaných snímkov za sekundu, ktoré zahŕňajú vykonanie všetkých CPU operácií (dekódovanie videa, prevod na jedno-kanálový obrázok, budovanie časti pyramídy, zahájenie prenosu textúr, registrácia kernelov na vykonanie a spracovanie výsledkov klasifikátoru) ako aj všetkých GPU kernelov.

		Veľkosť videa				
		640 × 352	640 × 480	768 × 576	1280 × 720	1920 × 1080
Predspracovanie	pyramída GPU čas [μs]	157	205	300	514	1133
	pyramída CPU čas [μs]	1226	1789	2441	4905	10926
	celkový CPU čas [μs][a]	2590	3780	5785	13322	27226
Klasifikátor s 20% FNR	detekcia [μs]	2113	2935	4010	7804	19092
	celkový GPU čas [μs]	2270	3140	4310	8318	20225
	snímkov za sekundu	**351.96**	**249.19**	**159.38**	**69.27**	**33.80**
Klasifikátor s 1% FNR	detekcia [μs]	7860	11935	15778	27630	87676
	celkový GPU čas [μs]	8017	12140	16078	28144	88809
	snímkov za sekundu	**117.08**	**78.18**	**59.09**	**33.63**	**10.72**

[a] Zahŕňa dekódovanie snímku videa, prevod na úrovne šedi a budovanie časti pyramídy.

Tabuľka 6.5: Priemerné časy jednotlivých fáz spracovania snímkov videí rôznych veľkostí spolu s výslednými počtami spracovaných snímkov za sekundu

Z tabuľky vidno, že v prípade klasifikátoru s hodnotou $\alpha = 0.20$ je čas potrebný na predspracovanie snímku videa na CPU dokonca o niečo vyšší ako čas samotnej detekcie,

preto najmä v prípade videí s vysokým rozlíšením by bolo možné dosiahnuť o niečo väčší počet snímkov spracovaných za sekundu už len optimalizovaním CPU operácií. V prípade videí s nízkym rozlíšením by to zrejme nepomohlo, keďže riadok *celkový CPU čas* nezahŕňa prenos snímkov medzi CPU a GPU, a rozdiely medzi časmi detekcie a predspracovania nie sú až také vysoké. Dôvodom nezahrnutia času potrebného na uskutočnenie prenosov je použitie asynchrónnych volaní, vďaka čomu CPU vykonáva iné operácie kým sa na pozadí dokončuje kopírovanie dát.

V tabuľke 6.6 je zobrazené využitie prostriedkov grafickej karty pri vykonávaní jednotlivých kernelov. Vidieť, že prvý a druhý detekčný kernel nedosahujú príliš vysokej obsadenosti, čo je spôsobené vysokým počtom registrov potrebných pre každú pracovnú skupinu. Novšie grafické karty (s podporou *CUDA Compute Capability 2.0*) sú schopné dosiahnuť ešte vyššiu obsadenosť aj pri takýchto vysokých počtoch potrebných registrov, takže sa dá predpokladať, že by detekcia bola vykonaná ešte rýchlejšie na novších grafických kartách. Naviac urýchlenie by tiež mohla priniesť nová verzia grafického ovládača, pokiaľ by optimalizovala alokácie registrov pri kompilácií kódu kernelov, vďaka čomu by detekcia mohla v budúcnosti fungovať rýchlejšie aj na testovanej grafickej karte.

Kernel	potrebných registrov	dosiahnutá obsadenosť	veľkosť skupiny
build_pyramid	15	0.938	320
red_kernel_step1	25	0.5	32×8
red_kernel_step2	23	0.5	256
red_kernel_step3	18	0.75	256
red_kernel_step4	17	0.75	256

Tabuľka 6.6: Využitie prostriedkov grafickej karty jednotlivými kernelmi

Výsledky ukazujú, že detekcia je takmer vo všetkých prípadoch (až na spracovanie *Full HD* videa klasifikátorom s $\alpha = 0.01$) schopná spracovávať viac ako 25 snímkov z videa za sekundu, pričom pri videách v nižších rozlíšeniach je možné vykonať detekciu aj niekoľko sto krát za sekundu.

Kapitola 7

Záver

Cieľom diplomovej práce bolo preštudovanie literatúry s témou OpenCL, výber algoritmu vhodného na spracovanie na grafickom akcelerátore, návrh metódy akcelerácie a jej následná implementácia. Všetky body zadania sa podarilo splniť.

Úloha, ktorá bola zvolená na spracovanie na grafickom akcelerátore bola detekcia objektov vo videu. Bol navrhnutý nový algoritmus pracujúci s WaldBoost klasifikátorom vhodný na paralelné spracovanie, ktorý pracuje na základe postupnej redukcie počtu pozícií detekčného okna, ktoré je nutné vyhodnotiť. Následne bola implementovaná aplikácia využívajúca technológiu OpenCL, ktorá používa navrhnutý algoritmus na detekciu objektov. Tiež boli identifikované techniky vďaka ktorým bolo možné maximalizovať výkon detektoru na grafickom akcelerátore a tieto zahrnuté do implementácie.

Výsledná aplikácia dosahuje veľmi vysoký výkon detektoru, pričom v porovnaní s efektívnymi CPU implementáciami dosahuje 12 a viac-násobné zrýchlenie detekcie na testovanej grafickej karte. Naviac, vzhľadom na paralelný charakter algoritmu sa dá očakávať, že výkon detektoru bude na modernejších grafických procesoroch ešte lepší. Experimentálne bolo tiež ukázané, že spôsob redukcie, ktorý algoritmus používa, neznižuje kvalitu detekcie.

Technológia OpenCL sa naviac postupne stáva podporovanou viacerými výrobcami, čo urýchľuje jej adopciu a OpenCL kompilátory pre CPU implementácie začínajú dokonca automaticky využívať vektorové rozšírenia procesorov akými sú SSE alebo AVX na urýchlenie kernelov, čo zvyšuje atraktivitu tejto technológie.

Jedna z možných budúcich prác by sa mohla zamerať na odvodenie redukčných pomerov už pri trénovaní klasifikátoru, alebo ich spočítať na základe štatistík trénovania, čím by sa dosiahla istota, že zvolené redukčné pomery sú správne. Ďalšia z možností budúcich vylepšení by bola snaha preniesť ešte väčšiu časť výpočtu na grafický akcelerátor - konkrétne aj dekódovanie videa pomocou rozhraní ako sú VDPAU (*Video Decode and Presentation API for Unix*) alebo DxVA (*DirectX Video Acceleration*) čím by bolo možné dosiahnuť nižšiu spotrebu energie a zároveň aj vyšší počet spracovaných snímkov za sekundu.

Literatúra

[1] Advanced Micro Devices: *ATI Stream Computing - OpenCL Programming Guide*. 2010-06, rev 1.03.

[2] Apple: *OpenCL Programming Guide for Mac OS X*. 2009-06-10.

[3] Gohara, D. W.: OpenCL Fundamentals [online]. `http://macresearch.org/opencl`, 2009.

[4] Harris, M.: Optimizing Parallel Reduction in CUDA. Technická zpráva, NVIDIA Coproration, 2010.

[5] Herout, A.; Jošth, R.; Juránek, R.; aj.: Real-time Object Detection on CUDA. *Journal of Real-time Image Processing*, 2010.

[6] Herout, A.; Juránek, R.; Zemčík, P.: Implementing the Local Binary Patterns with SIMD Instructions of CPU. In *Proceedings of WSCG 2010*, Plzeň, CZ: University of West Bohemia in Pilsen, 2010, s. 39–42.

[7] Herout, A.; Zemčík, P.; Hradiš, M.; aj.: *Low-Level Image Features for Real-Time Object Detection*. Vienna, AT: IN-TECH Education and Publishing, 2010, s. 111–136.

[8] Hradiš, M.; Herout, A.; Zemčík, P.: Local Rank Patterns - Novel Features for Rapid Object Detection. In *Proceedings of the International Conference on Computer Vision and Graphics*, 2008.

[9] Juránek, R.: *Rozpoznávaní vzorů v obraze pomocí klasifikátorů*. Diplomová práce, VUT Brno, 2007.

[10] Juránek, R.: Detection of Dogs in Video Using Statistical Classifiers. In *Proceedings of International Conference on Computer Vision and Graphics*, Heidelberg, Germany, 2008, iSSN 0302-9743.

[11] Juránek, R.; Zemčík, P.; Kadlček, F.; aj.: Automatic Synthesis of AdaBoost Classifier, zaslané k publikácií.

[12] Khronos OpenCL Working Group: *The OpenCL Specification*. 36 vydání, 2010-09-30.

[13] Kirk, D. B.; mei W. Hwu, W.: *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers, 2010, iSBN 978-012-3814-722.

[14] Matas, J.; Šochman, J.: Wald's Sequential Analysis for Time-constrained Vision Problems. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, ročník 2, 2005: s. 150–156.

[15] Moore, G. E.: Cramming more components onto integrated circuits. *Electronics*, ročník 38, č. 8, 1965-04-19: s. 114–119.

[16] NVIDIA Coproration: *nVidia OpenCL Best Practices Guide*. 2010-05-27.

[17] NVIDIA Coproration: *nVidia OpenCL Programming Guide for the CUDA Architecture*. 2010-08-16, version 3.2.

[18] Roth, P. M.; Winter, M.: Survey of Appearance-based Methods for Object Recognition. Technická zpráva, Inst. for Computer Graphics and Vision, Graz University of Technology, Austria, 2008-01-15.

[19] Viola, P.; Jones, M.: Robust Real-time Object Detection. *International Journal of Computer Vision*, ročník 57, č. 2, 2001: s. 137–154.

[20] Yang, M.-H.; Kriegman, D. J.; Ahuja, N.: Detecting Faces in Images: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, ročník 24, 2002: s. 34–58.

Zoznam príloh

Príloha A

Obsah DVD

DVD priložené k technickej správe obsahuje:

- zdrojové texty aplikácie
- dokumentáciu
- testovacie videá

Príloha B

Popis parametrov príkazového riadku

Parameter	skratka	popis
–show	-s	Zobrazí video v okne
–gpu		Vynúti použitie OpenCL zariadenia typu *CL_DEVICE_TYPE_GPU*
–cpu		Vynúti použitie OpenCL zariadenia typu *CL_DEVICE_TYPE_CPU*
–list	-l	Vypíše informácie o dostupných OpenCL platformách a skončí
–stages=N1,N2,...	-n	Počet slabých klasifikátorov vykonaných v jednotlivých kerneloch pred synchronizáciou
–start-frame=N		Počet snímkov videa ktoré budú preskočené
–timeout=T	-t	Čas v sekundách po ktorom aplikácia skončí
–profile	-p	Profiluje čas potrebný na vykonanie jednotlivých kernelov a zapíše výsledky do textového súboru
–step1-divisor=D1		Redukčná konštanta pre kernel *red_kernel_step1*
–step2-divisor=D2		Redukčná konštanta pre kernel *red_kernel_step2*
–step3-divisor=D3		Redukčná konštanta pre kernel *red_kernel_step3*
–step2-exec-count=N		Počet vykonaní kernelu *red_kernel_step2*
–step4-items-per-wg=M		Počet pozícií spracovávaných jednou pracovnou skupinou kernelu *red_kernel_step4*
–threshold=T	-r	Detekčný prah (hodnota γ silného klasifikátoru)
–output=FILE	-o	Názov výstupného súboru s vyznačenými pozíciami kde boli objekty detekované
–classifier-xml=XML	-x	Názov XML súboru s dátami klasifikátoru
–show-pyramid		Ladiaci mód kedy sa zobrazuje obrazová pyramída
–verbose	-v	Zobrazí rôzne informácie o priebehu spracovania
–count-discarded		Ladiaci mód kedy sa sčítavajú vyradené kandidátne pozície v jednotlivých kerneloch