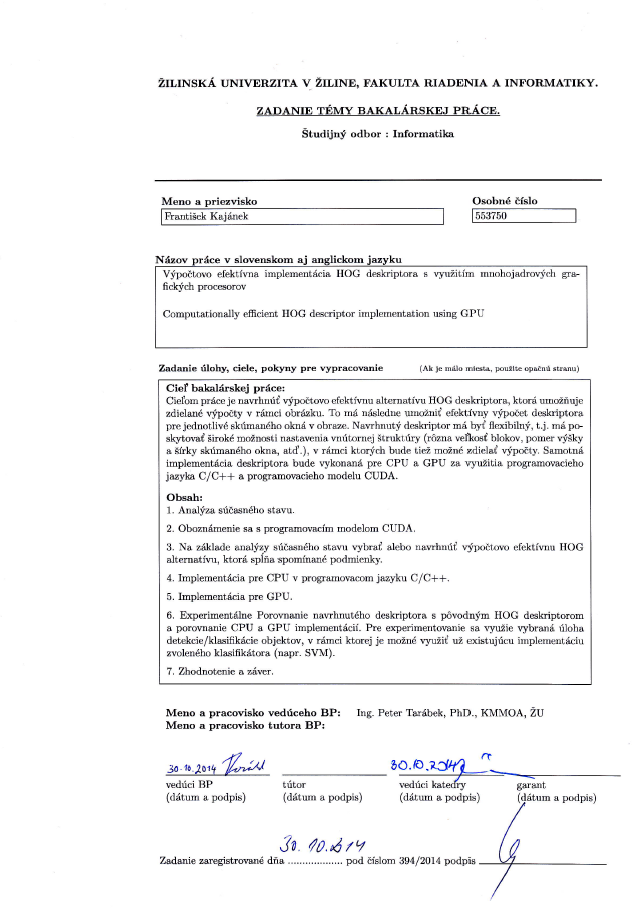
|  |  |
| --- | --- |
| Žilinská univerzita v Žilinežilinská univerzita v žiline  Fakulta riadenia a informatikyfakulta riadenia a informatiky | |
| bakalárska práca  Súčasné trendy v technológiách IP Fast Reroute | |
| František Kajánek  **Výpočtovo efektívna implementácia HOG deskriptora s využitím mnohojadrových grafických procesorov**  Vedúci práce:Ing. Peter Tarábek, Phd.  Evidenčné číslo: 28360720152394  Žilina, 2015Bakalárska práca | |
|  |  |

|  |  |
| --- | --- |
| Žilinská univerzita v Žilinežilinská univerzita v žiline  Fakulta riadenia a informatikyfakulta riadenia a informatiky | |
| bakalárska práca  Študijný odbor: InformatikaSúčasné trendy v technológiách IP Fast Reroute | |
| František Kajánek  **Výpočtovo efektívna implementácia HOG deskriptora s využitím mnohojadrových grafických procesorov**  Žilinská univerzita v Žiline  Fakulta riadenia a informatiky  Katedra matematických metód a operačnej analýzy  Žilina, 2015Bakalárska práca | |
|  |  |



ČESTNÉ VYHLÁSENIE

Čestne prehlasujem, že som prácu vypracoval samostatne s využitím dostupnej literatúry a vlastných vedomostí. Všetky zdroje použité v bakalárskej práci som uviedol v súlade s predpismi.

Súhlasím so zverejnením práce a jej výsledkov.

V Žiline, dňa 17.4.2015 František Kajánek

poďakovanie

Chcel by som poďakovať vedúcemu bakalárskej práce Ing. Petrovi Tarábkovi, PhD. za odbornú pomoc, pripomienky a usmerňovanie pri tvorbe práce.

Abstrakt

KAJÁNEK, František: ***Výpočtovo efektívna implementácia HOG deskriptora s využitím mnohojadrových grafických procesorov*** [Bakalárska práca] – Žilinská univerzita v Žiline, Fakulta riadenia a informatiky, Katedra matematických metód a operačnej analýzy. – Vedúci:Ing. Peter Tarábek Phd. – Stupeň odbornej kvalifikácie: Bakalár v odbore Informatika. Žilina: FRI ŽU v Žiline, 2015. – 63 s.

Cieľom predloženej bakalárskej práce je oboznámenie čitateľa s paralelným programovaním, HOG deskriptorom a s možnými spôsobmi paralelnej, a výpočtovo efektívnej aproximácie HOGu. Teoretická časť vysvetlí, prečo je zrýchlenie dosahované paralelizáciou, a ako rozhranie CUDA umožňuje dosiahnuť efektívnu paralelizáciu. Taktiež sa v tejto práci čitateľ oboznámi s algoritmom HOGa a zhodnotia sa spôsoby aproximácie, podľa ich urovne paralelizácie, rýchlosti a efektívnosti pri detekovaní objektov. Praktická časť porovná rôzne implementácie(CPU a GPU) a modifikácie tohto deskriptora. Pre porovnanie jednotlivých implementácii je použitá úloha detekcie chodcov s využitím SVM klasifikátora. Taktiež vysvetlí parametre paralelnej implementácie, ukáže vplyv parametrov na rýchlost/ úspešnosť a vyberie ideálne parametre pre daný testovaný dataset.

Kľúčové slová: CUDA, HOG, Paralelné výpočty, Support vector machine, CPU, GPU

AbstraCt

KAJÁNEK, František: **Computationally efficient HOG descriptor implementation using GPU** [Bachelor’s thesis] – University of Žilina, Faculty of management science and informatics, Department of mathematical methods and operations research. - Supervisor: Ing. Peter Tarábek Phd. – Academic qualification level: Bachelor of Informatics. Žilina: FRI ŽU v Žiline, 2015. – 63 p.

The goal of this bachelor thesis is to acquaint the reader with parallel programming, HOG descriptor and possibilities regarding a parallel and computationally effective approximation of HOG. The theoretical part deals with why parallelization offers a speedup and how CUDA enables us to reach an effective level of parallelization. The reader is going to acquaint himself with the HOG algorithm, and its possible approximations, ranking them based on degrees of parallelization, speed and effectiveness, whilst detecting objects. The practical part compares different implementations(CPU and GPU), and their modifications. To compare these implementations, pedestrian detection will be performed using an SVM classifier. Apart from that, parameters of these implementations will be explained, the effect of those parameters on performance/success rate will be discussed and a set of ideal parameters will be chosen.

Key words: CUDA, HOG, Parallel Computing, Support Vector Machine, CPU, GPU

Obsah

[Obsah 8](#_Toc419143852)

[Úvod 9](#_Toc419143853)

[1 CPU 10](#_Toc419143854)

[1.1 História procesorov 10](#_Toc419143855)

[1.2 Takt vs. počet jadier 13](#_Toc419143856)

[1.3 Moderné procesory a pamäť 14](#_Toc419143857)

[2 GPU 16](#_Toc419143858)

[2.1 História GPU 16](#_Toc419143859)

[2.2 Stavba GPU 19](#_Toc419143860)

[2.3 CUDA 21](#_Toc419143861)

[3 OpenCV 24](#_Toc419143862)

[4 HOG deskriptor 25](#_Toc419143863)

[4.1.1 Gradient 27](#_Toc419143864)

[4.1.2 Integrálny obraz 29](#_Toc419143865)

[4.1.3 Deskriptor 31](#_Toc419143866)

[4.2 SVM 33](#_Toc419143867)

[5 Rozbor Implementácii 34](#_Toc419143868)

[5.1 OPENCV implementácia 34](#_Toc419143869)

[5.2 CPU implementácia 36](#_Toc419143870)

[5.3 GPU – CUDA – implementácia 42](#_Toc419143871)

[5.3.1 Kernely 45](#_Toc419143872)

[5.4 Testovací dataset 49](#_Toc419143873)

[5.5 Vyhodnotenie 50](#_Toc419143874)

[5.6 Úvahy na zlepšenie 54](#_Toc419143875)

[6 Záver 56](#_Toc419143876)

[7 Zoznam Skratiek 57](#_Toc419143877)

[8 Referencie 59](#_Toc419143878)

[9 Zoznam použitých obrázkov 61](#_Toc419143879)

[10 Zoznam Tabuliek 62](#_Toc419143880)

[11 Zoznam príloh 63](#_Toc419143881)

Úvod

Zrýchlenie, optimalizácia a paralelizácia. Vždy ma tieto aspekty zaujímali a kvôli tomu som hľadal alternatívne riešenia k problémom, s ktorými som sa stretol. Taktiež som sa zaujímal o grafické procesory a samotnú prácu s grafikou alebo obrazom. Tieto dôvody okrem iných ma smerovali k výberu tejto bakalárskej práce.

Cieľom mojej práce je nájsť efektívnu paralelnú implementáciu HOG deskriptora. Samotný HOG deskriptor sa v niektorých krokoch zle paralelizuje(tzn. je pomalý), takže nedosahuje až tak veľmi veľké zrýchlenie. Častokrát chceme deskriptory používať v reálnom čase, či už na desktope alebo mobilnom zariadený pre rôzne účely. Preto je potrebná analýza algoritmu a hľadanie alternatívnych spôsobov počítania niektorých krokov.

V teoretickej časti sa čitateľ oboznámi s históriou procesorov a ich stavbou. Rozoberie sa, aké faktory najčastejšie ovplyvňujú výkon procesora. Nasledovať bude krátky rozbor stavby grafickej karty, CUDA modelu a OpenCV knižnice. Ako posledná je kapitola o samotnom HOG deskriptore, v ktorej sa vysvetlí čo je HOG, aké sú jeho základné kroky a oboznámenie s parametrami algoritmu. Tieto kapitoly vytvoria základ vedomostí potrebný na pochopenie praktickej časti práce.

V praktickej časti sa nachádza detailný rozbor trojice implementácii. Prvá implementácia je referenčná OpenCV implementácia, jej popis a približné fungovanie. Druhá implementácia je CPU implementácia. Pri tej sa nachádza detailný popis jej tvorbe, ako sa menila od prvého k poslednému návrhu a poznatky získané pri programovaní. Tretia implementácia je GPU implementácia. Tu sa čitateľ oboznámi so všetkými dôležitými aspektami, ktoré bolo treba brať na ohľad pri programovaní na CUDA modeli. Taktiež sa tu nájde detailný rozbor kernelov, a postup pri ich optimalizácií. Za tým nasleduje vyhodnotenie implementácii a na záver zhodnotenie možných budúcich vylepšení.

# CPU

## História procesorov

Prítomnosť počítadiel, či už mechanických alebo elektronických výrazne ovplyvnila vývoj ľudstva v dejinách. Či už to boli prvé grécke počítadlá, používané na jednoduché výpočty, alebo dešifrovacie stroje využívané v druhej svetovej vojne, bez nich by niektoré činy boli len sťažka dosiahnutelné.

Prvé elektromechanické programovateľné počítače sa vyskytli už na konci druhej svetovej vojny. Ich problém však bol, že neboli Turingovo úplné, alebo taktiež výpočtovo úplné. Takéto počítače boli preto limitované a nemohli byť používané na akýkoľvek výpočtový účel.

Nasledovalo obdobie prvých továrňovo vyrábaných počítačov, počas ktorého sa počítače zmestili do nákladného vozidla. Väčšina z nich používalo bubnovú magnetickú pamäť(jadrová pamäť bola dostupná, ale neuveriteľne drahá). Počítače čítali príkazy a dáta z magnetických pások. Taktiež sa objavili prvé binárne počítače ako poznáme dnes, ale napriek tomu boli veľmi odlišné od tých dnešných. Ako príklad môžeme uviesť spôsoby ukladanie záporných hodnôť. V tomto období boli používané metódy „sign magnitude“ a „one’s complement“, t.j. znamienková magnitúda a jednotkový doplnok. Znamienková magnitúda fungovala na princípe bežných čísel, ak máme záporné číslo dáme pred neho znamienko „-“. V binárnej sústave to znamenalo dať 1 pred číslo, tzn napríklad „1“ bola 00000001 a „-1“ bola 10000001. Jednotkový doplnok znamenalo, že ak chceme dostať negatívne číslo, invertujeme všetky 0 na 1 a naopak. Napríklad „1“ bola 00000001 a „-1“ bola 11111110. Názov jednotkový doplnok je z dôvodu, že zápornú verziu čísla dostaneme aj ak dané číslo odčítame od verzie -0 jednotkového doplnku. Tá je sekvenciou samých jednotiek, tzn číslo dostaneme doplnkom do samých jednotiek. V dnešnej dobe používame dvojkový doplnok, ten dostaneme inverziou čísla ako u jednotkového, a pričítaním jednotky. V tomto prípade „-1“ bude 11111111. Dôvodom prečo dvojkový doplnok je pokročilejšia metóda, je fakt, že pri operáciach nie je nutné kontrolovať hodnotu znakového bitu, a nula má len jednu reprezentáciu, narozdiel od znamienkovej magnitúdy a jednotkového doplnku.

Všetky tieto počítače ale mali jednu veľkú slabinu: program na jeden, nefungoval na druhom. V 60-tych rokoch IBM prišlo s možným riešením.  [**System/360**](http://en.wikipedia.org/wiki/System/360)Mal byť virtuálny počítač, ktorého inštrukčná sada mohla byť emulovaná na rôznych kusoch hardwaru. Tento, a ďalšie koncepty boli nazývané CISC(Complex Instruction Set Computer). V praxi to znamenalo, že jedna inštrukcia spustí viacero nízkoúrovňových príkazov. Toto zjednodušilo prácu programátorov, keďže bežný programátor si zapamätá maximálne desiatky inštrukcií a niekoľko spôsobov práce s pamäťou na nízkej úrovni. Bol to príklad prvých skoro ortogonálnych inštrukčných sád- t.j. sada, kde ľubovoľná inštrukcia môže používať ľubovoľný register/spôsob adresovania.

V 70-tych rokoch prišlo zmenšenie a bežnejšie používanie výpočtových a časovacích čipov. Prvý známejší mikroprocesor bol Intel 4004 v roku 1971, a o rok neskôr Intel 8008, priamy predchodca dnešných core i7/5/3 procesorov od Intelu. Každá inštrukcia 8008 má priamy ekvivalent v dnešných procesoroch, s výnimkou opkódov. Miniaturizácia umožnila umiestniť celé CPU na jednu PCB dosku. IBM pokračovalo vo výrobe rýchlych a vysokopamäťových počítačov. Taktovacia rýchlosť dosahovala niekoľko MHz a RAM bola už väčšia ako 1 MB.

80-te roky priniesli mnoho vylepšení. Zistilo sa, že zmenšením ortogonality sa počítače zároveň zrýchlia a zlacnia. Začali sa pridávať cache pamäte do procesorov, kvôli zrýchleniu prístupu do pamäte. Taktiež sa súbežne vyskytovali počítače Harvardskej a Von Neumann architektúry. Harvardská architektúra mala oddelené inštrukcie a dáta, zatiaľ čo Von Neumann architektúra mala jeden adresný priestor, kde boli uložené aj inštrukcie a dáta. V dnešnej dobe sa používa modfikovaná Harvardská architektúra, tá má jednu hlavnú pamäť a následne separátne cache pamäte v procesore pre inštrukcie a dáta. Výsledkom je využitie kladných stránok oboch architektúr: jednoduchosť a rýchlosť prístupu do pamäte.

Čím viac pamäte bolo treba, tým väčšie adresy boli potrebné. Kvôli jednoduchosti sa ustanovilo, že „slová“ budú násobky znakov. Takto vznikli 8,16,32,64-bitové adresné priestory. Dnes nám to príde smiešne, ale v minulosti veľkosť adries musela byť dostatočná pre adresovanie a zároveň malá tak aby nezaberala príliš miesta.

Už vtedy sa objavovali prvé operačné systémy, ktoré urobili prácu s počítačom viac transparentnú. Programátor mohol používať vysokoúrovňové programovacie jazyky a nemusel sa vždy starať o nízkoúrovňové operácie. Procesory mali viacero ALU, čo im umožňovalo vykonávať niekoľko na sebe nezávislých mikroinštrukcií naraz. Toto sa nazýva superskalárny dizajn.

V 90-tych rokoch a prakticky až do dnes nastal rozmach procesorov rôznych typov, inštrukčných sád a architektúr. „Bežné“ počítače sú prakticky len rýchlejšie verzie týchto počítačov. Nastala masovejšia výroba PC, prenosné počítače sú čoraz populárnejšie a miniaturizácia umožnila výrobu mobilných zariadení, ktoré sú rýchlejšie ako mnohé len o dekádu staršie počítače. A vďaka programovacím jazykom, je možné v jednom jazyku(po menších úpravách) spustiť kód na širokej škále zariadení. Ako príklad môžeme uviesť PC a mobilné zariadenia Android/iOS.

## Takt vs. počet jadier

Čím rýchlejšie a čím rozmanitejšie procesory sa začali vyrábať, tým väčší problém bol v meraní a porovnávaní výkonu. Drvivá väčšina výrobcom klasifikovala rýchlosť v MHz/GHz, tzn. v inštrukciách za sekundu. Do určitého bodu tento indikátor fungoval, ale potom prišli procesory ktoré vyhodnocovali naraz viac mikroinštrukcií.

V 80-tych rokoch Intel poukazoval na omnoho vyššiu rýchlosť taktu oproti všetkým iným výrobcom. Súperiaci Apple toto označil ako „megaherthový mýtus“. Vtedajší procesor Applu 6502 spracoval inštrukciu LDA za 2 μs na takte 1 MHz. Zatiaľčo tá istá inštrukcia na Intel 8088 mal skoro 5-násobný takt 4.7 MHz a samotná inštrukcia trvala 0.87 μs. Tzn samotná inštrukcia bežala trochu viac ako 2x tak rýchlo na Apple procesori.

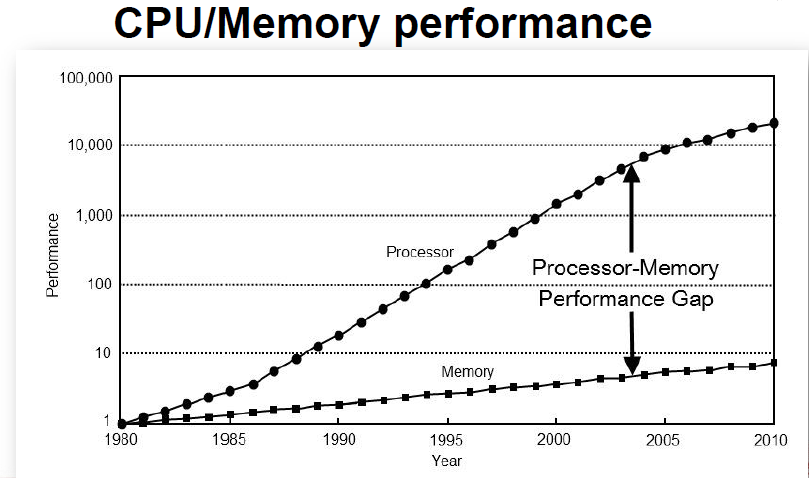
Okolo roku 2000 nárast taktu bol tak výrazný, že sa predpokladali taktovacie rýchlosti okolo 10-15 GHz vo veľmi krátkom období. Nárast sa zastavil v roku 2005, kedy bol dosiahnutý takt 4 GHz na procesore Pentium Extreme Edition. Bolo očividné, že bez veľmi značných zmien v chladení procesora sú vyššie taktovacie rýchlosti nemožné.

O rok neskôr prišiel Intel s úplne novým konceptom. Intel Core 2 bola architektúra, ktorá priniesla viacjadrové procesory do bežných desktopových počítačov. Oproti minulej generácií sa spotreba energie znížila o polovicu, zatiaľ čo výkon zostal viac menej rovnaký. Takt sa znížil na priemerných 2.5 GHz, čo malo za výsledok omnoho menšie teploty.

Nasledujúce roky pre Intel aj AMD boli viac menej o „vyšší takt, viac jadier“. Výrobné procesy sa zmenšovali, taktiež sa zvyšoval počet mikroinštrukcií na jeden takt, a výsledkom bolo zrýchlenie na celej čiare. Dnešný Intel i7 procesor(rok 2014) má 3-4 GHz takt a 4-8 jadier(a 8-16 logických jadier). Tzn. 8 jadier \* 14-19 mikroinštrukcií počas jedného cyklu [1] je veľmi vysoká úroveň paralelizácie. Trend, ktorým sa vyvíjajú procesory je jasný, pokus o čoraz väčšiu paralelizáciu na CPU.

## Moderné procesory a pamäť

Už vieme, že procesory sú schopné tisíciek operácií za sekundu. Pri takom náraste rýchlosti nastáva ale úplne iný problém. Pamäť už nie je taká rýchla ako procesor. Pred 30-timi rokmi procesor pristupoval do pamäte ihneď, pamäť aj procesor mali rovnaký takt a stavba bola taktiež jednoduchá: jeden procesor, jedna RAM pamäť.

V dnešnej dobe je pamäť segmentovaná v bežných prípadoch na nasledovné: CPU register, L1 Cache, L2 Cache, L3 Cache a RAM. V tomto poradí je taktiež ich prístupová doba, a v opačnom poradí ich veľkosť, tzn. čím väčšia tým pomalšia. Zatiaľ čo registre sú prístupné ihneď, tzn. za jeden cyklus, RAM pamäť môže byť prístupná niekedy až o 200 CPU cyklov neskôr.

Nato aby sa obišli tieto obmedzenia, navrhli sa zložité algoritmy, ktoré predpovedajú používanie pamäte a podľa toho ju skopírujú to bližšej, rýchlejšie prístupnej pamäte. Pri viacúrovňovej cache nastáva potreba o vysoký hit-rate. Tento údaj hovorí ako často nájde procesor daný údaj v danej úrovni Cache. Ak máme napríklad 99% hit rate, prístup do L1 cache 1 nanosekundu a prístup do L2 cache 10 nanosekúnd, znamená to že procesor 99 operácií urobí za 99 nanosekúnd, a zvyšnú jednu za 10. Celkové trvanie teda je 109 ns a to znamená že 1% miss rate nám zníži rýchlosť o 10%. V reálnom svete je L1 hit-rate okolo 95% – 97% [1] a vplyv na rýchlosť CPU je okolo 14%. Ak by sme predpokladali, 95-97% hit rate a procesor by musel pristupovať až do hlavnej pamäte, tieto 2% rozdiel by mohli znamenať až zdvojnásobenie času na výpočet daného problému.

Obrázok -Rozdiel medzi nárastom výkonu CPU a pamäte

Zdroj:Computer Architecture, a quantitative approach; Hennessy,Patterson,Arpaci-Dusseau

Väčšina programátorov tieto malé rozdiely ani nezaregistruje, ale ak ide o veľmi vysokú úroveň paralelizácie, kde máme zdieľané zdroje a veľa súbežných výpočtov, treba prístupy do pamäte minimalizovať.

# GPU

## História GPU

Počiatky grafických procesorov sa dajú sledovať do 70-tych rokov, kedy sa začal prechod z obrazu spracovávaného sekvenčne na procesore k jednotkám medzi procesorom a obrazovkou. Medzi prvými bol RCA video čip, ktorý dokázal produkovať video signál o rozlíšení 64x128 pixelov. Nasledovník tohto čipu bol napríklad použitý v Atari 2600 v roku 1977. Konkurenčná firma Motorolla taktiež vyvinula svoj video čip v roku 1978, ktorý bol predchodcom čipu používaného v prvých osobných počítačoch.

Ďalší vývoj grafických čipov bol skôr v profesionálnej sfére. Grafický čip Intelu 82720 bol použitý ako základ pre iSBX 275 Video Graphics Controller Multimode Board, ktorá vedela vykresľovať 8 farebné dáta o rozlíšení 256x256 a jej 32kb pamäť bola dostatočná pre kreslenie geometrických útvarov a bitových máp.

Pokračovali roky počas ktorých boli firma ATI(rok 1985) firma Nvidia(rok 1993) a firma 3Dfx Interactive(1994) založené. ATI bola jedným zo zakladateľov Video Electronics Standards Association (VESA), ktorá mala na starosti unifikáciu rozhraní a zariadení. Bol taktiež problém v používanom API. Zatiaľ čo v roku 1992 prišiel na trh OpenGL 1.0, ktorý podporoval aj 2D aj 3D grafiku, Microsoft vyvíjal svoju vlastnú verziu grafického API Direct3D. Kvôli tomu ani nelicencoval OpenGL Mini-Client Driver(MCD) na Windows 95. Ako odpoveď na odmietnutie, firma SGI(zakladateľ OpenGL) vytvorila vlastný inštalovateľný driver pre túto platformu, ktorý podporoval omnoho viac funkcí ako pôvodne predložený MCD.

Začala sa výroba 3D grafík, a čoraz viac výrobcov sa pridávalo do konkurencie, dokým neprišla na trh karta 3Dfx Voodoo Graphics, ktorá poskytovala na tú dobu bezkonkurečný výkon a cenu. Bola to čisto 3D grafika a pokiaľ bolo treba 2D operácie, bola spojená káblom s inou 2D grafikou od 3Dfx. Výkon tejto karty spôsobil nárast záujmu o 3D grafiku aj u výhradne 2D vývojárov. Predpokladá sa, že počas vrcholu 3Dfx kariet, ich podiel na trhu 3D grafík bol okolo 80-85%.

Intel vyvinul port AGP, ktorý bol používaný až do roku 2004 väčšinou grafických kariet, a jeho nasledovníkom bol PCIe(Peripheral Component Interconnect Express) a ten je používaný v jeho tretej iterácií 3.0 dodnes.

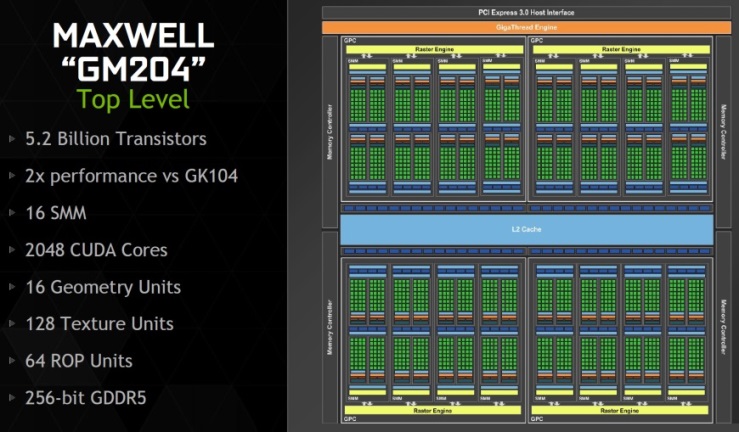
Po vydaní Voodoo 2 sa firma 3Dfx rozhodla starať o predaj sama, a toto rozhodnutie bohužiaľ spôsobilo jej pomalý zánik v roku 2002. Toto bolo začiatkom obdobia dominancie ATI a NVIDIA, ktoré trvá až dodnes. Predtým ale vyvinuli SLI(Scan-Line Interleave) rozhranie ktoré umožňovalo spoluprácu dvoch Voodoo 2 grafík tak, že každá spracovávala polovicu zobrazovaných riadkov.

V roku 1999 prišla na trh karta GeForce 256, označovaná ako prvé GPU. Hlavnou zmenou bolo pridanie hardware enginu na T&L, t.j. transformáciu 3D obrazu na 2D. Táto karta bola podporovaná renderovacími aplikáciami až do roku 2006.

Nasledovalo obdobie znižovania výrobného procesu, rozširovania zberníc a všeobecného zrýchlenia grafický procesorov. Firma 3Dfx bola kúpená firmou NVIDIA. Microsoft prišiel so svojim grafickým rozhraním DirectX, ktoré počas verzí 9-11 bolo dominantným renderovacím rozhraním. Firma ATI prišla v roku 2005 s Crossfire, alternatívou k SLI, tzn. využitím viacerých grafických kariet na spracovávanie videa. Na začiatku bolo zamýšľané spojenie pomocou Y kábla, ale neskôr boli použité mostíky medzi grafikami, a najnovšia generácia komunikuje výhradne cez PCIe 3.0 slot.

V roku 2007 NVIDIA vydala beta verziu CUDA platform SDK(Software development toolkit), založenej na jazyku C. Toto umožnilo veľmi vysokú paralelizáciu a distribuované výpočty na grafickej karte v rozsahu, aký predtým nebol dostupný.Konkurenčná firma ATI, sa ako odpoveď na CUDA rozhodla plne podporovať OpenCL, narozdiel od in-house riešenia. OpenCL je založené na C99, je to otvorený štandard založený a udržiavaný neziskovou organizáciou Kronos Group. Konformné implementácie sú dostupné od Altera, AMD, Apple, ARMHoldings, Creative Technology, IBM, Imagination Technologies, Intel, Nvidia, Qualcomm, Samsung, Vivante, a ZiiLABS. [2] [3] Tieto SDK boli základom pre GPGPU(General Purpose GPU).

V posledných rokoch sa taktiež objavila popularita zrýchľovania grafických procesorov na mobilných zariadeniach. Rôzne mobilné aplikácie používajú paralelné výpočty na spracovávanie obrazu, rendering a iné.

Stavba GPU

Obrázok - Dizajnový nákres Maxwell architektúry GPU

Zdroj:http://www.extremetech.com/wp-content/uploads/2014/09/maxwell-gm-204-top-level-diagram.jpg

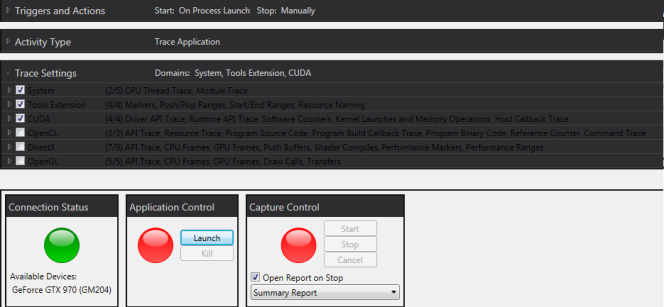
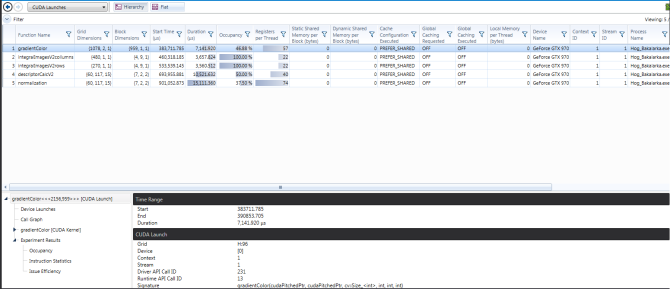
Grafické procesory sa podstatne líšia od normálnych procesorov. Zatiaľ čo normálny moderný i7 procesor má 4-8 jadier, okolo 1.4 miliardy tranzistorov, L1-L3 cache a RAM, bežná grafická karta GTX 970 má 1664 stream procesorov, a 4GB vlastnej GDDR5 pamäte. Takt jadra CPU je omnoho vyšší, až 4GHz kým GPU má v bežnom stave okolo 1,1GHz.

Grafická karta má globálnu pamäť, zdieľanú celou kartou a adresovanú hlavnou zbernicou. Následne sú procesory rozdelené na skupiny vlákien(nazývané warpy u NVIDIe a wavefront u AMD), viacero skupín zdieľa niektoré funkcie, ako napríklad rastrovací engine, texture a L2 cache. L1 cache je vo forme zdieľanej pamäte v rámci skupiny procesorov. Samotná skupina vlákien má u najnovších grafík zdieľaný súbor pre registre, zdieľanú pamäť, plánovač(scheduler), a mnoho ďalších, ktoré sa líšia podľa architektúry a výrobcu.

Pre porovnanie, bežné výkonné CPU core i7 5820K, a grafická karta Maxwell architektúry GTX 980 majú nasledovné špecifikácie: [4] [5]

1. CPU má 4 jadrá, a možnosť spúšťania naraz 8 vlákien, GPU má 2048 stream procesorov, I keď tieto jadrá sú značne jednoduchšie a majú zdieľané elementy, rozdiel v paralelizácií je značný.
2. CPU má 15MB dedikovanej L3 cache, zatiaľčo grafická karta má 2048 L2 cache, a vlastnú globálnu 4GB pamäť. CPU využíva viac úrovní pamäte ako GPU, + RAM je viac menej 4tá úroveň pamäte pre CPU, zatiaľčo GPU má k dispozícií len 3.
3. Pamäť GPU má 7000 MHz takt, zatiaľčo bežná RAM pamäť má takt okolo 1600MHz(DDR4 pre CPU, GDDR5 pre GPU). Maximálna priepustnosť pamäte pre toto CPU je 68 GB/s, a pre GPU 224GB/s. Takt jadra pri GPU je okolo 1200 MHz(záleží od výrobcu) a pri CPU 3.3GHz.
4. GFLOP/s(koľko operácií s posunutou desatinnou čiarkou za sekundu môže čip urobiť) pre CPU je okolo 142 zatiaľ čo GPU má okolo 4612, ak zaručíme maximálnu úroveň paralelizácie.

Z uvedených štatistík je jasné, že ak ide o sekvenčné výpočty, GPU bude maximálne zaostávať za CPU keďže jedno jadro GPU je omnoho jednoduchšie. Ale ak príde na čistú výpočtovú rýchlosť pri zabezpečení maximálnej úrovne paralelizácie, CPU môže byť viac ako 2,5x pomalšie [7]. Taktiež je vhodné spomenúť, že CPU model je omnoho náročnejší na maximálnu optimalizáciu, keďže nebol navrhnutý výhradne s úmyslom paralelizácie. Taktiež narozdiel od CPU operácie s dvojitou presnosťou(double precision operations), sú omnoho pomalšie. Každý multiprocesor GPU zdieľa double precision jednotku s viacerými jadrami, a preto priepustnosť double precision operácií je častokrát viac ako 10-násobne menšia.

CUDA

Obrázok -Nsight profiler v móde Profile Cuda Application

Obrázok -Nastavenia Nsight profilera

**Compute Unified Device Architecture** je paralelná výpočtová platforma a programovací model vytvorený firmou NVIDIA. Umožňuje využitie grafických kariet pre všeobecné výpočty(nie nutne len grafiku), a je implementovaná na zariadeniach, ktoré vyrába NVIDIA. CUDA umožňuje programátorom priamy prístup k virtuálnej inštrukčnej sade GPU, ktoré podporujú CUDA model.

Platforma je dostupná pomocou CUDA akcelerovaných knižníc, kompilátorových direktív, a taktiež pomocou rozšírení pre bežne programovacie jazyky ako C, C++ a Fortran. Zatiaľčo C/C++ sa kompiluje pomocou kompilátora vyíjaného firmou NVIDIA, ktorý je založený na LLVM, Fortran sa kompiluje pomocou PGI CUDA kompilátora, vyvíjaným The Portland Group. Na ostatné jazyky ako Python, Perl, Java, Haskell, Matlab a ďalšie, sú dostupné wrappery vyvíjané inými firmami.

Využitia CUDA modelu sú rôzne, používa sa na biologické simulácie(hľadanie artériálneho trombusu) [8], kryptografii(SSL/AES akcelerácia), analýza dopravy, fyzikálne modely(simulácia molekúl). V týchto prípadoch dokážu GPU dosahovať viac ako 2,5-násobné zrýchlenie [7].

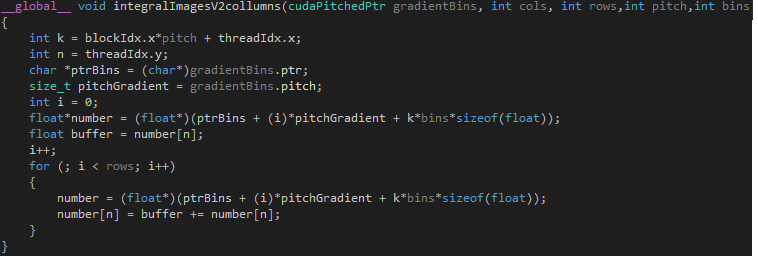
CUDA toolkit je voľne dostupný pre vývojárov po celom svete. Obsahuje NVCC kompilátor pre Linux a Windows. Vo Windowse sa používa v spojení s Visual C++ kompilátorom a v linuxe v spojení s GCC kompilátorom. Ďalej sú v balíku rôzne nástroje, ako GPU/CUDA debugger, memory checker(kontroluje nesprávnu prácu s pamäťou a hlási ju), vizuálny profiler výkonu, pamäte, časovania a paralelizácie. Tie sú dostupné buď samostatne, alebo v spojení s prostredím Visual Studio.

Na obrázkoch 3 a 4 môžeme vidieť GUI spúšťania profilera, dostupné sú rôzne nastavenia ako monitorovanie CPU, GPU, CUDA, OpenCL, OpenGL a DirectX. Taktiež môžeme zobraziť hlbokú analýzu trvania kernelov, a čo obmedzuje ich zvýšenú paralelizáciu(obrázok 4).

CUDA používa kombináciu dvoch paralelizačných metód, SIMD a SMT. SIMD znamená, že viacej vlákien spracúva krátky vektor dát. SMT znamená, že viacero vlákien s rôznymi inštrukciami sa vykonávajú paralelne, a je rozšírením sekvenčného CPU modelu. SIMT je kompromis medzi týmito dvoma. Narozdiel od SIMD, umožňuje prácu s dlhými vektormi, je umožnené používať viac registrových setov(registre na GPU sú iné ako registre na CPU), rôzne adresy, a rôzne toky(flow paths). Naopak SIMT je menej flexibilné oproti SMT, veľká divergencia vlákien(každé vlákno robí úplne rôzne operácie), malé obsadenie(occupancy) jadier veľmi redukujú výkon, a je dostupná iba jednu primitívu synchronizácie \_\_syncthreads(). Táto metóda zabezpečuje, že kód za touto metódou nebude vykonávaný skôr ako všetky vlákna opustili kód pred touto metódou. To nebráni v prepínaní kontextu na iné vlákno, zatiaľčo iné čaká na ostatné vlákna.

Prevedením do príkladu, SIMT má oproti SIMD lacnejšie(menej náročné na zdroje) vykonávanie miest, kde sa musí o niečom rozhodovať(if-else bloky), a oproti SMT má SIMT lacnejšie pracovanie s vláknami, ale za cenu omnoho problematickejšieho kódu(ak všetky warpy nie sú obsadené, klesá výkon, a tak isto ak nastáva divergencia toku, tzn. časté rozhodovanie).

Prvá voľne dostupná verzia CUDA platformy bola vydaná v roku 2007 pre Microsoft Windows a Linux. CUDA funguje na všetkých jadrách od grafickej karty rady G8x. Každá grafická karta má určitú „compute capability“, tzn. verziu CUDA platformy, ktorú podporuje. Najnovšia verzia počas písania tejto práce je 5.2 dostupná na Maxwell architektúre. Oproti prvej verzii sa zvýšil počet súbežných vlákien, zvýšenie dimenzií blokov(na 3D), počet súbežných warpov(malých sekcií kernelu, ktoré zdieľajú niektoré prvky, a sú na jednom multiprocesore), počet registrov, atď.Jedna z novších podporovaných funkcií je dynamická paralelizácia, možnosť vytvárať vlákna vo vláknach GPU.

Je taktiež dôležité spomenúť veľmi podobné názvy. Zatiaľčo warp je základná skupina vlákien v hardwari u NVIDIA grafík, CUDA má taktiež označenie „thread block“. Tento blok sa vzťahuje na spúšťanie kernelu, voči vláknam, warpom a kernelu nasledovne: Jeden kernel môže obsahovať niekoľko blokov. Každý block môže obsahovať niekoľko warpov, a samotný warp sa vykonáva na jednom fyzickom warpe. Preto treba brať do úvahy či sa hovorí o fyzickej alebo logickej stránke CUDA platformy.

Na obrázku 5 môžeme vidieť bežný kernel kód, ktorý je vykonávaný na GPU. Direktíva \_\_*global*\_\_ označuje funkciu, ktorá bude vykonávaná na GPU. Kód sa inak veľmi nelíši od bežného CPU kódu. Rozdiel oproti bežnému kódu sú blockIdx a threadIdx štruktúry. Tie obsahujú pozíciu vlákna v bloku a bloku v gride vlákien, podľa ktorej je možné počítať prístupy do pamäte, ktoré budú rôzne pre každé vlákno. Až na niektoré pokročilejšie možnosti (napríklad virtuálne funkcie v triedach alebo výnimky), podporuje CUDA väčšinu C/C++ štandardu. Pri volaní kernelu sa špecifikujú rozmery gridu(počet blokov) a rozmery blokov(počet vlákien) v tvare <<<grid dimenzie, blok dimenzie>>>.

Obrázok -Príklad kernel kódu

# OpenCV

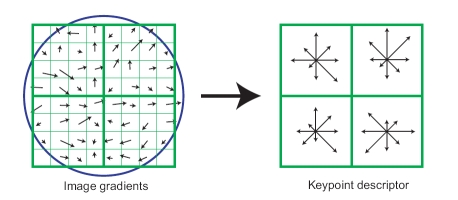
OpenCV je knižnica programovacích funkcií, hlavne cielená na real-time prácu s obrazom. Je voľne dostupná pod BSD licenciou. Možno ju používať na viacerých platformách vrátane Windows, Linux, Android, OS X, Blackberry 10 a mnoho ďalších. Využitia knižnice sa nachádzajú v oblastiach rozpoznávania objektov, tvárí, sledovanie pohybu, rozšírenej reality, segmentácie obrazu, mobilnej robotiky a ďalších oblastiach počítačového videnia.

Prvotne bola knižnica vyvíjaná pobočkou Intelu v Nižnom Novgorode, odvtedy sa k podpore pridala Willow Garage a Itseez. V auguste 2012 prevzala vývoj knižnice nezisková organizácia OpenCV.org [7], ktorá udržuje stránku pre developerov a používateľov, vrátane online dokumentácie [8].

Väčšina OpenCV knižnice je napísaná v jazyku C++, v ktorom je napísaný celý primárny interface. Knižnica obsahuje taktiež starší, menej prehľadný ale napriek tomu úplný C interface. Úplné interfacy sú taktiež dostupné v jazykoch Java, Python, MATLAB/OCTAVE. API týchto interfacov sú dostupné v online dokumentácií. Wrappery pre ďalšie jazyky ako C#, Perl, Ruby a Ch boli vyvinuté za účelom rozšírenia záujmu o knižnicu. Od roku 2010 je vo vývoji GPU CUDA interface, a od roku 2012 je vo vývoji GPU OpenCL interface.

# HOG deskriptor

Metóda využitia HOGa [11] alebo taktiež histogramu orientovaných gradientov(Histogram of oriented Gradients) na rozpoznávanie objektov je založená na viacerých poznatkoch. Hlavná myšlienka je, že lokálny vzhľad a tvar v obraze sa dá dobre charakterizovať lokálnymi intenzitami gradientu, aj bez znalosti presných hodnôt alebo presných pozícií hrán.

V praxi sa tento princíp aplikuje ako rozdelenie obrazového okna na bunky, Každá bunka naakumuluje 1D histogram gradientových smerov alebo hranových orientácií cez pixely danej bunky. Kombináciou všetkých buniek dostaneme reprezentáciu histogramu(obrázok 6).

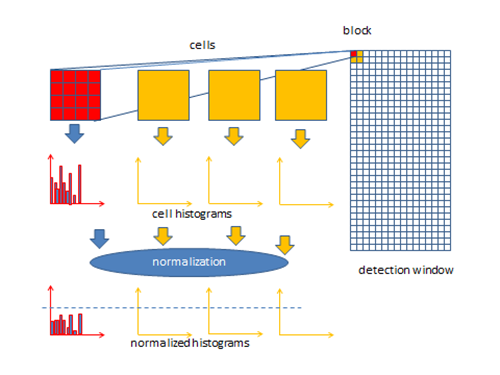
Následne z dôvodov rôzneho osvetlenia v obraze, je potrebné kontrastovo normalizovať lokálne hodnoty pred ich použitím. Toto dosiahneme akumuláciou „energie“ vo väčších priestoroch, v tomto prípade blokoch, a následnou normalizáciou pomocou výsledku akumulácie. Týmto normalizovaným deskriptorovým blokom budeme hovoriť Histogram of oriented Gradients(HOG).

Obrázok - Transformácia pixelov na bunkový histogram

Zdroj: “Distinctive Image Features from Scale-Invariant Keypoints,” Lowe, IJCV, 2004

Pokiaľ skombinujeme tento prístup s prelínaním blokov, a následným použitím výsledného „feature“ vektoru v bežnom SVM klasifikátore, dostávame postup pre rozpoznávanie objektov, napríklad ľudí v obraze.

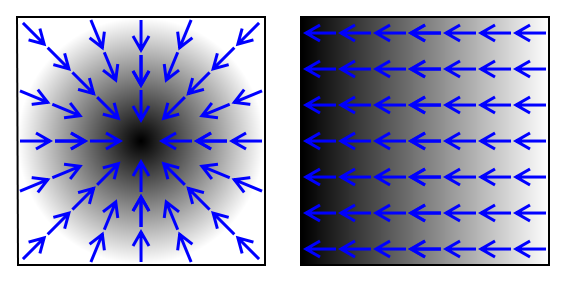
V prípade implementácie sa preto dá segmentovať tento algoritmus na nasledujúce kroky(obrázok 7):

* + 1. Výpočet gradientu pre každý bod, váženie magnitúdy gradientu do najbližších košov(binov) podľa uhla gradientu.
    2. Výpočet buniek histogramu, a uloženie do výsledného histogramu.
    3. Normalizácia buniek v rámci blokov.

Obrázok - Ilustrácia fungovania normalizácie

Zdroj: “Distinctive Image Features from Scale-Invariant Keypoints,” Lowe, IJCV, 2004

Gradient

Gradient pri práci s obrazom, je zmena intenzity farby v určitom smere. Je pravidelne používaný pri práci s obrazom na získavanie informácií o obraze. Matematicky možno gradient popísať v každom bode obrazu, ako 2D vektor, ktorého komponenty dostaneme deriváciami v horizontálnom a vertikálnom smere. V každom bode obrazu ukazuje gradient smer najväčšieho zvýšenia intenzity a dĺžka vektora gradientu korešponduje s rýchlosťou zmeny v danom smere. V obrázku 7 môžeme vidieť smer aký nám udáva gradient pre daný obraz. Tak ako šípky znázorňujú, gradient je v smere zmeny farby od svetlej k tmavšej.

Obrázok - Vizualizácia smeru gradientu pri zmene farby

Zdroj: http://commons.wikimedia.org/wiki/File:Gradient2.svg;Sarang, Cweiske

Ak máme spojitú funkciu intenzity i(*x*,*y*) tak gradient v smere x vypočítame nasledovne:

Vzorec

Ak však máme diskrétnu funkciu, ako napríklad obraz, tak môžeme zmenu sledovať len v pixelových intervaloch. Preto v tomto prípade bude vzorec vyzerať nasledovne:

Vzorec

Rovnakou logikou dostaneme vzorec pre gradient v smere y:

Vzorec

Spojením týchto dvoch funkcií dostaneme výslednú hodnotu gradientu v obraze nasledovným vzorcom:

Vzorec

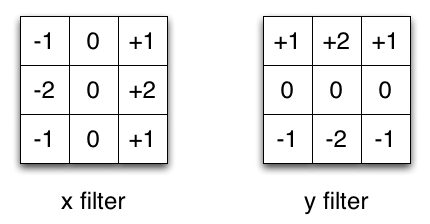
je gradient v smere x(horizontálnom), je gradient v smere y(vertikálnom).

Smer gradientu možno vypočítať nasledovne:

Vzorec

Gradient v obraze môžeme vypočítať pomocou nejakej derivačnej masky/gradientového filtru napríklad [-1,0,1] v horizontálnom a [-1,0,1]T vo vertikálnom smere. Výpočet gradientu vyzerá nasledovne: máme bod (*x*,*y*), v horizontálnom smere dostaneme gradient rovnicou -(*x*-1,*y*)+(*x*+1,*y*) a podobne vo vertikálnom -(*x*,*y*-1)+(*x*,*y*+1). Taktiež sa dajú používať iné masky napríklad [-1,1] alebo sobel maska(obrázok 9).

Gradient v obraze má mnoho využití. Hlavné oblasti, v ktorých sa stretneme s gradientom sú počítačová grafika a počítačové videnie. V počítačovej grafike s používa na vizuálnu úpravu ako napríklad úprava ostrosti, rozmazanie alebo zvýraznenie črtov. V oblasti počítačového videnia sa používa na extrahovanie informácií z obrazu, ako napríklad detekcia hrán/objektov.



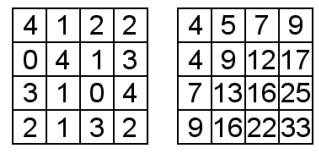
Obrázok - Sobel Maska

Integrálny obraz

Integrálny obraz(taktiež nazývaný „summed-area table) je algoritmus, ale aj datová štruktúra používaná na rýchly výpočet súčtu hodnôt v obdĺžnikovej ploche. Prvýkrát bol tento princíp použitý v počítačovej grafike v roku 1984 v spojení s mipmapami. Prvá známejšia zmienka bola pri použití Viola-Jones object detection frameworku [9], na rýchly výpočet oblastí v spojení s haarovými vlnkami.

Samotný vzorec výpočtu integrálneho obrazu je nasledovný:

Vzorec

To znamená, že každý bod integrálneho obrazu je súčtom všetkých bodov naľavo a hore od daného bodu vrátane daného bodu. Ak si vezmeme bod na súradniciach 2,2(obrázok 8, napravo), tak v integrálnom obraze 9 = 4 + 1 + 0 + 4(obrázok 8, naľavo), tzn. všetky hodnoty naľavo a hore od daného bodu, vrátane toho bodu sú výslednou hodnotou v tom bode.

Obrázok - Ilustrácia hodnôt integrálneho obrazu(vpravo) k pôvodnej matici(vľavo)

Samotných algoritmov na výpočet integrálneho obrazu je niekoľko, spomenieme však dva použité v tejto práci. Prvý z nich je sekvenčný výpočet, ktorý jedným prechodom vypočíta integrálny obraz pre všetky body. Máme funkcie I(*x*,*y*), funkcia intenzity v integrálnom obraze a i(*x*,*y*), funkcia intenzity pôvodného obrazu. Vzorec je nasledovný: I(*x*,y)=i(*x*,*y*) + I(*x*-1,*y*) + I(*x*,*y*-1) - I(*x*-1,*y*-1), tj v každom bode zoberieme intenzitu v danom bode, a pričítame k nej body, ktoré sú nad, naľavo a naľavo hore od samotného bodu.

Druhý algoritmus je základný paralelný algoritmus, ktorý dvoma prechodmi vypočíta integrálny obraz, a úroveň paralelizácie je podľa nižšej hodnoty z počtu stĺpcov a počtu riadkov. Prvý prechod vypočíta vo vertikálnom smere pre každý bod nasledovný vzorec: I(*x*,*y*)=i(*x*,*y*)+I(*x*,*y*-1) a druhý prechod vypočíta v horizontálnom smere pre každý bod vzorec: I(*x*,*y*)=I(*x*-1,*y*)+I(*x*,*y*). Ak to popíšeme podľa obrázku 9, najprv v bode 2,2 po prvom prechode bude 1+4=5 a v bode 2,1 bude 4+0=4. Po druhom prechode bude v bode 2,2 4+5=9 tzn. správna hodnota finálneho integrálneho obrazu.

Deskriptor

Samotný HOG má široké spektrum nastavení. V práci Dalal a Triggsa [11], bol vypracovaný dlhý zoznam ideálnych parametrov pre detekciu ľudí v obraze. Taktiež porovnali vplyv daných parametrov na FPPW(počet falošných detekcií na okno), a taktiež na rýchlosť sekvenčného algoritmu.

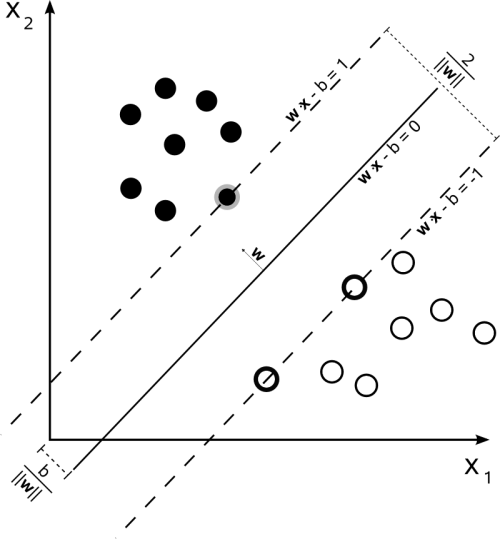
Efektívna detekcia ľudí v obraze závisí od veľkosti zvoleného okna, a taktiež od rôznych veľkostí buniek a blokov. Povedzme, že máme obrázok veľkosti 96x160. Osoba v danom obrázku je približne veľkosti 64x128. Ideálny prípad pri výbere trénovacieho datasetu bude, ak osoba bude v strede a okolo seba bude mať približne 16 pixelov miesta z každej strany(tzn. presne vycentrovaná). Tieto testy bol vypracované v pôvodnej práci Dalal a Triggsa, a bolo zistené, že v takomto prípade ideálna veľkosť bloku je 3x3 buniek a veľkosť bunky 6x6 pixelov. Ak okolo daného objektu nie je tento okraj, a okno nie presne obkolesuje osobu/objekt v obraze, počet FPPW rastie.

Prvý krok algoritmu spočíva vo voľbe správnej derivačnej masky, práci s farbami, ich normalizáciou a vyhladzovaním. V tomto kroku podľa práce[11], najlepšie výsledky poskytuje práca s farebným obrazom, bez úpravy farieb, alebo vyhladzovania. Najlepšia derivačná maska je [-1,0,1], zatiaľčo všetky zložitejšie masky ako Sobel alebo Gausova zhoršujú výslednú detekciu. Taktiež použitie necentrovaných [-1,1] masiek zhoršuje výsledok. Pre RGB obraz vezmeme gradient každého farebného kanálu, vyberieme ten ktorý má najväčšiu normu(napríklad absolútnu hodnotu, v tomto prípade). Taktiež máme k dispozícií dve metódy ukladania magnitúdy do košov, diskrétna a vážená. Vážená metóda uloženia magnitúdy do dvoch najbližších košov poskytuje najlepšie výsledky.

Druhý krok algoritmu spočíva v akumulácií gradientov do jedného histogramu v rámci bunky a v tomto prípade je postup priamočiary. Jediná možná variácia je ak predpokladáme obdĺžnikové bloky a kruhové bloky. Zatiaľčo kruhové bloky ponúkajú lepší výsledok, obdĺžnikové bloky sú rýchlejšie a omnoho jednoduchšie na implementáciu.

Tretí krok spočíva v normalizácií buniek v rámci blokov. Testované boli 4 metódy normalizácie a to L2(euklidova normalizácia, ), L2 nasledovaná osekávaním(limitovaním maximálnej hodnoty na 0.2), L1 normalizácia () a nakoniec L1 normalizácia nasledovaná odmocninou. Epsilon je ľubovoľná malá konštanta. Keďže následne používame SVM, a bloky sa prelínajú, je potrebná určitá úroveň regularizácie. [11] Všetky metódy okrem jednoduchej L1 normalizácie ponúkajú porovnateľné výsledky, bez znateľného rozdielu pri porovnávaní rýchlosti algoritmu.

SVM

Je metóda strojového učenia. SVM slúži na klasifikáciu príznakov, a ich rozdelenie do dvoch tried(napríklad pozitívne príznaky, negatívne príznaky). Ako prvý krok musia byť SVM dodané trénovacie data, ktoré sú klasifikované ako jedna, alebo druhá trieda. SVM následne hľadá nadrovinu ktorá v priestore príznakov rozdeľuje trénovacie data na dve skupiny. Optimálne riešenie je také, kedy triedy sú v opačných polpriestoroch a keď veľkosť minimálnej vzdialenosti od rozdeľujúcej roviny je čo najväčšia. Na obrázku 6 môžeme vidieť ilustráciu problému, ktorý rieši SVM. Biele a čierne body reprezentujú dáta rôznych tried, *w* je rozdeľujúca rovina, a prvky na prerušovanej čiare sú prvky najbližšie k rovine.

Obrázok - Príklad optimalizačnej úlohy, ktorú sa snaží SVM riešiť.

Zdroj:http://commons.wikimedia.org/wiki/File:Svm\_max\_sep\_hyperplane\_with\_margin.png

Dôležitou časťou techniky SVM je jadrová transformácia(kernel transformation), tá umožňuje transformovať príznaky dát na príznaky vyššej dimenzie. Toto umožňuje preniesť úlohu, ktorá bola pôvodne lineárne neseparovateľná, na úlohu lineárne separovateľnú, a na ktorú možno následne aplikovať optimalizačný algoritmus na nájdenie rozdeľovacej nadroviny. Výhodou tejto metódy je, že transformácia sa dá aplikovať nielen na body v *Rn* ale taktiež na grafy, stromy, postupnosti DNA atď.

SVM objekt implementovaný v OpenCV umožňuje natrénovať dáta, dodané cez pole Mat pomocou metódy train. Následne je možné pomocou metódy predict zistiť, či dáta ktoré boli poslané taktiež ako pole Mat patria do jednej alebo druhej skupiny.

# Rozbor Implementácii

OPENCV implementácia

Táto implementácia bola používaná ako referenčná pre zobrazenie výsledkov mojej implementácie. Voči nej sa porovnáva rýchlosť, úspešnosť a modifikovateľnosť pre rôzne situácie. Je to priama implementácia HOGa podľa práce Dalala a Triggsa [10]. Priebeh algoritmu je identický s popisom implementácie v práci. Taktiež je možné skompilovať algoritmus s podporou knižnice IPP od Intelu, ktorá zabezpečuje paralelizáciu ak sa použijú ekvivalenty metód OPENCV z tejto knižnice.

Samotný algoritmus je naprogramovaný objektovo, ako vstupné parametre akceptuje všetky testované a monitorované parametre z práce o HOGu. Medzi hlavné parametre patria: rozmery buniek, rozmery blokov, rozmery okien, odsadenie, posun okien/blokov(kolko sa bloky prelínajú navzájom) atd. Nachádza sa tu viacero nízkoúrovňových optimalizácií, ako napríklad priama práca s dátami(narozdiel od práce s objektami), „unrolled“ cykly(direktíva pre kompilátor, ktorá rozloží cykly na sekvenčné príkazy, narozdiel od skokov), prípadne špecifické správanie na Intel procesoroch, za účelom paralelizácie niektorých činností. Taktiež treba spomenúť zrýchlenie výpočtu blokov a ich normalizácie. V tejto implementácii sa každý blok počíta práve raz, keďže sa pri spracovaní obrazu bloky vždy prelínajú, algoritmus znovu použije už vypočítanú hodnotu bloku(v triviálnej implementácii sa jeden blok bežne počíta aj 4x). Táto zmena celkovo zníži počet potrebných prístupov do pamäte, a taktiež

Taktiež tu nájdeme vlastnú implementáciu SVM, s dvoma už natrénovanými možnosťami a to pre 48x96 okno a pre 64x128 okno. Keďže vnútorné SVM by nám neposkytlo správne dáta pre porovnanie, budeme používať metódu compute, ktorá poskytne vektory HOGa. Bežný čas hardwarovo nezávislej verzie metódy(sekvenčná, kompilovaná bez podpory IPP) je 3,4ms pre 96x160 obrázok a 651ms pre 1920x1080 obrázok. Ak sa užívateľ rozhodne využiť aj vnútorné SVM, je tu metóda detectMultiScale, ktorá popíše obrázok HOGom, zhodnotí či boli nájdené osoby, a vráti súradnice obdĺžnikov, v ktorých by sa mal nachádzať človek. V tejto práci sa ako náhrada za túto metódu používa metóda compute, pomocou ktorej natrénujeme objekt cv::SVM a následne metódu detect, ktorá má rovnaký účel, a to rozpoznávať objekty v obraze(bez obmedzenia na ľudí). Nepoužívame metódu detectMultiScale, pretože potrebujeme testovať rozpoznávanie pre rovnaký vstupný dataset ako GPU/CPU implementáciu, aby sa dali zaručiť rovnaké podmienky.

CPU implementácia

Ako prvý krok k paralelnej implementácii v CUDA som zvolil sekvenčnú implementáciu na CPU. Tento krok umožní jednoducho naprogramovať základné časti algoritmu bez nízkoúrovňovej réžie potrebnej pri programovaní na GPU. Taktiež umožní čiastočné vyhodnotenie výsledkov(pomocou výsledkov detekcie a podľa rýchlosti), ešte pred začatím náročnej nízkoúrovňovej optimalizácie. Ďalším cieľom je verzia CPU hoga(separátna od normálnej) v ktorom je úpravený spôsob prístupov do polí podobne ako to je na GPU. Zatiaľčo na CPU by bol cyklus for(i=0; i<stĺpce ;i+=4), tzn. iterovalo by sa po každom štvrtom prvku a i by reprezentovalo finálny index kam pristúpiť v poli, na GPU sa počítajú indexy podľa indexov vlákna/bloku vlákien. Preto som každý cyklus iteroval po jednom prvku, a následne vypočítal výsledné miesto v poli, kam chcem pristupovať aritmetickými operáciami. Tento krok výrazne zjednoduší prvotné písanie kernelov v CUDA, a ich následný debug.

Za vstupné parametre som zvolil obrázok a bežné parametre HOGa ako veľkosť bunky, bloku, okna, odsadenie, posun bloku a okna, a počet gradientových košov(gradient bins). Tieto parametre budú uložené ako atribúty objektu. Taktiež tu máme cv::SVM objekt ako atribút, pre zjednodušenú prácu s testovaním detekcie. Výstup je cv::Mat pole veľkosti *n* x *m*, kde *n* je počet okien obrázku a *m* je počet feature daného okna.

Hlavná metóda, ktorá popíše obraz, sa volá hogCompute. Ako vstupné parametre berie obrázok, ktorý chceme popísať, a cv::Mat pole, do ktorého má vložiť vektory okien(pre každý obrázok niekoľko okien, napr. pre 96x160 obrázok, okná 64x128 a posun o 16px ich bude 9). Máme tu taktiež alternatívu k tejto funkcii, ktorá obsahuje rovnaký kód, ale umožňuje zápis trvania častí algoritmov. Toto zľahčuje postup pri identifikácií časovo kritických bodov algoritmu, tzn. miest, ktoré potrebujú najviac optimalizácie. Ďalej bude nasledovať jednotlivý rozbor krokov algoritmu v nasledovnom poradí: Alokácia polí, výpočet gradientu, výpočet integrálneho obrazu, výpočet buniek a blokov a následná normalizácia.

****

Obrázok - Obrázok chodcov

Obrázok - Gradient k obrázku 12

**Prvý krok** algoritmu je alokácia polí. Triviálna implementácia by používala 7 polí, jedno pre uloženie každého medzivýpočtu. Tieto polia sú gradientx a gradienty, pre medzivýpočet samotného gradientu v horizontálnom a vertikálnom smere. Potom to je pole finalGradient a direction, do ktorých by sa uložila magnitúda gradientu, a uhol gradientu. Ďalšie by bolo pole gradientBins, kam sa ukladá diskrétna alebo vážená hodnota gradientu uložená do košov(bins), vypočítaná pomocou uhla gradientu(kôš do ktorého patrí) a magnitúdy(aká hodnota do daného koša pôjde). Pole integralImage, v ktorom uložíme integrálny obraz pre každý smer/gradientového koša. Ak máme napríklad 9 košov, tak budeme mať 9 integrálnych obrazov. Posledné pole descriptorValues by obsahovalo všetky vektory okien gradientov. Napríklad pre okno 64x128 bude každé okno obsahovať 3780 vektorov. Počet vektorov závisí od posunu bloku, veľkosti bloku a bunky, a veľkosti okna. Počet okien závisí od veľkosti obrázku, odsadenia, veľkosti okna a posunu okna.

**Druhým krokom** algoritmu je výpočet gradientu a zadelenie do gradientových košov. Triviálna implementácia vypočíta pre každý bod v obraze veľkosť gradientu(okrem okrajov, kde sa predpokladá nulový gradient). V každom bode sa zistí hodnotu gradientu *x* a *y*, pomocou derivačnej masky [-1,0,1] v horizontálnom smere a [-1,0,1]T masky vo vertikálnom smere.

Keďže pracujem s farebným obrazom, metóda výpočtu je trochu komplikovanejšia ako pri greyscale obraze. Derivačnú masku aplikujem na všetky tri farebné kanály v oboch smeroch(x a y), a v každom bode rozhodnem, ktorá z farieb má najväčšiu absolútnu hodnotu. Následne použijem danú hodnotu ako magnitúdu pri vážení.

 Pomocou funkcie atan(y gradient,x gradient) dostanem uhol gradientu a pomocou rovnice |gradient x + gradient y| dostanem magnitúdu v každom bode. Rozhodnem, ktoré uhlové koše sú k danému uhlu najbližšie. Predpokladám 9 košov, a uhlové koše 10,30,50,70,90,110,130,150,170. Ak dostanem napríklad uhol 37®, najbližšie koše sú 30 a 50. Rozdiel košov je 20®, takže do 30 koša pôjde 13/20 magnitúdy, a do 50 koša pôjde 7/20 magnitúdy ak predpokladáme váženú verziu, v prípade diskrétnej pôjde celá magnitúda do koša 50.

Prvá iterácia tohto kroku bola identická s triviálnou diskrétnou verziou. Táto verzia bola vcelku neefektívna, ale dala sa na nej porovnať správnosť výpočtov vizualizáciou. Vizualizácia magnitúdy gradientu má ukazovať znázornenie hrán v obraze. Ako vidíme na obrázku 11, presne to sa podarilo pomocou [-1,0,1] masky docieliť.

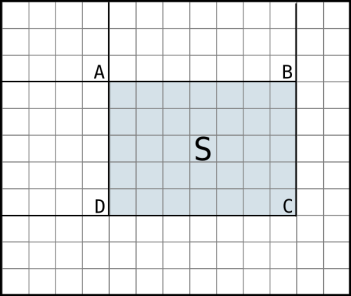
Druhá iterácia používala váženú verziu algoritmu. Táto verzia podľa práce o HOGu dosahuje lepšie výsledky po finálnej detekcii, bez výrazného ovplyvnenia rýchlosti. Taktiež som zistil, že polia gradientx, gradienty, finalGradient a direction sa dajú nahradiť lokálnými premennými. Žiaden z tých medzivýpočtov nie je potrebný v neskoršej časti algoritmu a preto nie je potrebné ho ukladať do pamäti. Táto zmena výrazne ovplyvnila potrebu operačnej pamäte, keďže tieto polia tvorili viac ako polovicu používanej pamäte.

Tretia iterácia bola finálne zrýchlenie kroku. Ako prvé som preskupil premenné tak, aby deklarácia, a ich využitie bolo až priamo v bode kde sú potrebné(tento krok na CPU výkon neovplyvnil ale pomôže neskôr na GPU). Ďalej som podmienky, ktoré sú zodpovedné za zadelenie magnitúdy do správnych košov, zredukoval na ternárne operátory „?“. Predpokladáme 9 gradientových košov, tzn. 20 stupňov medzi každým košom. Predtým sa zadelenie počítalo podľa intervalov 0-10, 10-170 a 170-180. Používal som posunuté koše o 10, tzn koše boli napríklad 10,30,50 namiesto 0,20,40. Krajné koše sa obsluhovali rôzne, kvôli pretečeniu(0vý uhol má ísť do koša 170 a 10, čo by bežnou aritmetikou pristupovalo do prvku poľa v bode -1). Nový spôsob výpočtu používa neposunuté koše(aby sa musela ošetrovať len jedna strana intervalu <0,180> a nie obe). Prvý ternárny operátor ošetruje možnosť uhlu 180, ten má byť zadelený do koša 0 a 20, keďže posledný kôš je 160 stupňov. Druhý ternárny operátor ošetruje možnosť uhlov (160,180), ktoré sa zadeľujú do košov 160 a 0. Celkovo táto zmena zmenšila potrebný počet premenných o 2 a jednoduché ternárne operátory sú rýchlejšie ako komplikovanejšie if bloky. Táto zmena znížila trvanie tohto kroku o 10%.

**Tretí krok** je výpočet integrálneho obrazu. Keďže sa jedná o sekvenčnú CPU implementáciu, použil sa jednoduchý sekvenčný výpočet, ktorý prebieha v troch cykloch, najprv pre každý bod v obraze(okraje majú hodnotu 0, tzn. žiaden gradient) a následne pre každý gradientový kôš. Jedinou zmenou tohto kroku bolo prispôsobenie pre vážené hodnoty magnitúdy gradientu, keďže tie sú uložené v type float.

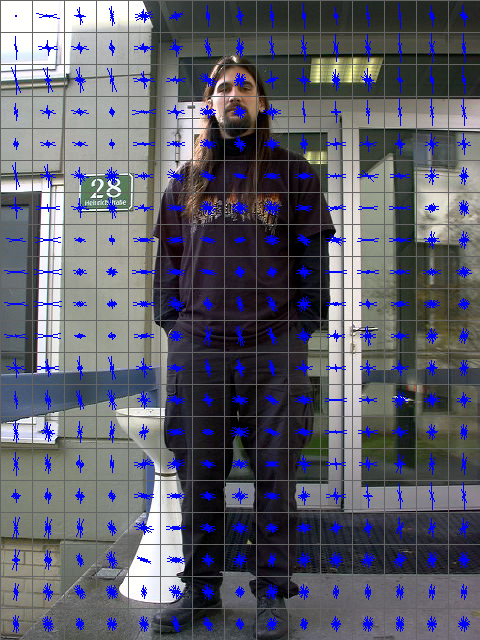
Obrázok - Ilustrácia výpočtu obdĺžnika pomocou integrálneho obrazu

Zdroj:http://www.nongnu.org/rapp/doc/rapp/integral.html

**Štvrtý krok** je výpočet blokov a ich normalizácia. Je to finálny krok výpočtu HOG deskriptoru. Triviálna verzia vypočíta všetky bunky všetkých blokov, a potom v rámci každého bloku znormalizuje vektory do intervalu <0,1> pomocou zvolenej normy. Realizácia tohto kroku je 7 vnorených for cyklov. Prvé dve úrovne iterujú cez všetky okná deskriptora v obraze. Ďalšie dve iterujú cez všetky bloky v okne. Na tejto úrovni sa počíta hodnota bloku, ktorá je neskôr využitá na L1 normalizáciu , x je daný vektor, ktorý chceme znormalizovať, ||x||(norma) je suma absolútnych hodnôt všetkých vektorov v bloku a epsilon je ľubovoľná malá konštanta(napríklad mnou použitá 0,00001). Ďalšie dve úrovne iterujú cez všetky bunky, a posledná cez všetky orientácie, keďže každá bunka je vo výslednom okne práve raz pre každú orientáciu.

Tento krok sa delí na dve väčšie časti, výpočet buniek a normalizáciu. Výpočet buniek je veľmi jednoduchý keďže sa v predošlom kroku vypočítal integrálny obraz. Každá bunka sa vypočíta pomocou bodov A,B,C,D ako je znázornené v obrázku 6. Samotný vzorec je C+A-B-D. Táto časť algoritmu od úplne prvého návrhu prešla len jednou zmenou. Pri prvej iterácií sa index každého prvku počítal pri každom prístupe do pamäte. Každý bod má určitú časť výpočtu indexu rovnakú. Napríklad bod A má rovnakú *x* súradnicu ako bod D, tak isto bod A *y* súradnicu ako bod B. Preto som predpočítal spoločnú časť indexu do dvoch premenných(jednu pre *x*, druhú pre *y*) a pomocou toho som následne počítal index v poliach. Výrazne to zlepšilo čitateľnosť kódu, aj zjednodušilo následné hľadanie chýb pri návrhu GPU algoritmu, kde sa dali použiť identické výpočty indexov.

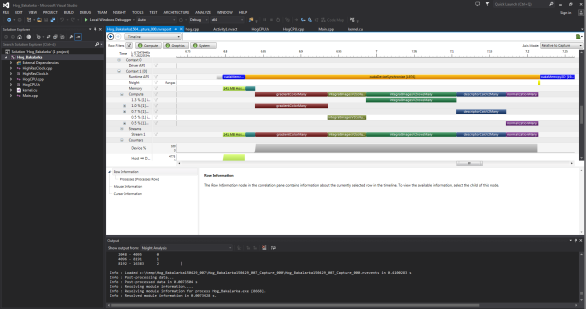
Druhý podkrok bola normalizácia. Prvá iterácia bola naprogramovaná pomocou cyklu, v ktorom sa sčítali hodnoty všetkých vektorov pre blok, uložených vo výslednom poli predošlým krokom. Tento spôsob mal výhodu jednoduchosti a vďaka tomu bolo ľahšie naprogramovať následnú implementáciu, v ktorej sa používa integrálny obraz. Zatiaľčo u tejto môžeme vypočítať aj L1 aj L2(suma odmocnin vektorov) normalizáciu, pri druhej omnoho rýchlejšej verzii sa dá vypočítať iba L1 normalizácia, ak sa nevenuje čas ďalším výpočtom. L1 normalizácia v prvej iterácií vypočíta sumu všetkých vektorov daného bloku, pričíta k tomu malú konštantu(0,00001) a touto hodnotou následne predelí každý prvok v bloku. V druhej iterácií pristúpi do 4ch bodov v integrálnom obraze, ktoré korešpondujú s okrajmi bloku(podľa obrázku je rovnica A+C-B-D), sčíta ich pre všetky orientácie a tým predelí každý prvok bloku. Ak máme napríklad v každom bloku 4 bunky(minimálny počet) tak počet prístupov do pamäte je v bežnom prípade identický. Zrýchlenie nastáva vo viacero momentoch. Prvý je, že pre hociaký parameter HOGa, výpočet cez integrálny obraz budú vždy 4 prístupy. Ďalej na hornom a ľavom okraji obrázku sa počet prístupov zredukuje na 1-2, keďže body A a D, respektíve A a B(obrázok) budú nulové. A nakoniec, výpočet cez integrálny obraz sa môže implementovať ako 4 sekvenčné výpočty, zatiaľčo ten bez integrálneho obrazu sa môže jedine implementovať cyklom a ten je pri porovnaní asi o 10% pomalší(ak v oboch prípadoch počítame L1 normalizáciu, ktorá produkuje identický výsledok).

Na obrázku 7 môžeme vidieť vykreslené vektory HOG deskriptoru. Každý segment reprezentuje jednu bunku. Touto metódou vizualizácie [8], každá hrana v obraze je reprezentovaná vektormi pravoúhlymi na danú hranu. Problémom so správnou vizualizáciou HOGa je prelínanie blokov a správne váženie buniek. Preto každý vektor je súčtom danej bunky v rôznych blokoch, a následne je každá bunka spriemerovaná počtom výskytov danej bunky v blokoch(na okrajoch sa bunka môže vyskytovať len raz, zatiaľčo v strede niekoľkokrát). Iná metóda vizualizácie je dostupná pre MatLab [9]. V tejto práci sa pojednáva popísanie obrazu HOGom, a následné invertovanie obrazu. Na príkladoch je predložené, prečo v niektorých prípadoch je pri použití detekcie HOGu nájdený objekt aj na nezmyselnom mieste.

Obrázok - Vizualizácia buniek HOG dekritptora

GPU – CUDA – implementácia

Zatiaľčo CPU implementácia mala za účel naprogramovať „len“ sekvenčnú verziu HOG deskriptora, ktorá spĺňa požiadavky a otestuje aproximácie, táto mieri na podstatné zlepšenie využiteľnosti HOG deskriptora. Narozdiel od CPU implementácie, programovanie GPU pomocou CUDA nebolo triviálne. Ako prípravu na niektoré zmeny oproti CPU programovaniu, bola predošlá implementácia navrhnutá s účelom väčšej podobnosti k modelu GPU programovania.

Ako ďalšie boli dôležité nastavenia samotného prostredia. Jedna z modifikácií, ktoré som urobil vzhľadom na CUDu je nastavenie kompilátora na compute\_52,sm\_52. Toto nastaví „compute capability“, t.j. špecifikáciu rýchlosti akou je schopná grafická karta spúšťať kód. Zjednodušene sa dá povedať, že čím vyššia, tým lepšia, ale taktiež čím vyššia, tým novší hardware je potrebný na danú špecifikáciu. Samotné toto nastavenie znížilo čas vykonávania kódu na polovicu.

Obrázok -Timeline view CUDA profilera

Keďže sa pracovalo s paralelným výpočtovým modelom, normálny debugging aký sa používa na procesore je nepoužiteľný. Presne tento problém rieši Nsight plugin pre Visual Studio. Je tu dostupný debugger pre: GPU/CUDA debugging, debugging grafiky, analýzu výkonu a kontrolu pamäte. Počas programovania som použil CUDA debugger, kontrolu pamäte, a neskôr pri optimalizácií aj analýzu výkonu. Na obrázku môžeme vidieť timeline, ktorý znázorňuje časy a následnosť volaní CUDA API, prípadne kernelov HOG deskriptora.

Tak ako u predošlej implementácie, prvá časť algoritmu je venovaná práci s pamäťou. Zatiaľčo na CPU postačuje alokácia dynamického poľa a práca s ním, na GPU je potrebná prídavná réžia. Prvým krokom je zvolenie správnej alokačnej funkcie a datového typu pre pointer. CUDA API poskytuje nielen bežnú cudaMalloc funkciu, ale taktiež pokročilejšie cudaMallocPitch a cudaMalloc3D(a veľa ich ďalších variácií). Prvá metóda naalokuje súvislý blok pamäte v pamäti GPU, pokiaľ je dostatok voľnej pamäte. Druhá metóda naalokuje 2D pole vo fyzickej pamäti, každý riadok má na konci „pitch“- odsadenie, ktoré zabezpečuje aby riadky bol presne pod sebou zarovnané. Posledná metóda rovnakým princípom naalokuje 3D pole, a samotná 3tia dimenzia poľa rovnako ako pri predošlej metóde je presne zarovnaná v pamäti.

Ja som zvolil cudaMallocPitch ako funkciu, ktorú budem používať. Dôvodom je, že polia, ku ktorým pristupujem, nevyžadujú 3D alokáciu(ktorá je podstatne zložitejšia) ale taktiež paralelizácia tak ako som ju navrhol môže čerpať zo zarovnania riadkov. Tento krok sa môže zdať ako príliš komplikovaný, problém ale je v spôsobe ako GPU scheduler optimalizuje prístupy do pamäte. Ako vieme, vlákna sa zodeľujú na bloky a bloky sa ešte na hardwarovej úrovni delia na warpy. Keďže warpy zdieľajú plánovacú jednotku, táto plánovacia jednotka sa snaží dáta alokovať naraz, pomocou vektorov. Zatiaľčo normálne pri prístupe do pamäte berie plánovač ten prvok ktorý potrebujeme, warp plánovač sa snaží alokovať všetky prvky, ktoré vlákna potrebujú naraz. Toto sa dá jedine v prípade ak dáta, ktoré má naalokovať sú fyzicky vedľa seba a môže ich vziať ako jeden vektor. Ak sa mu to nepodarí, musí omnoho pomalším spôsobom pristupovať na izolované miesta v pamäti, a toto môže zablokovať výkon kvôli rýchlosti pamäte.

Ako datový typ som použil cudaPitchedPtr. Je to štruktúra, ktorá obsahuje pointer na pole, pitch, tzn. odsadenie, a šírku(v bytoch) a výšku poľa. Pri použití tejto štruktúry s metódou cudaMallocPitch, volania API sú optimalizované, a preto trvajú kratšie.

Pri CPU implementácii sa nám podarilo zredukovať počet potrebných polí na 4, gradientBins(pole na vážené magnitúdy gradientu), integralImage(pole pre integrálny obraz), vstupné a výstupné pole. Tu budeme potrebovať dve ďalšie polia, a to vstupné pole na CPU, a výstupné pole na CPU. V neskoršej fáze sa mi podarilo nahradiť pole integralImage znovupoužitím poľa gradientBins a to vďaka „in-place“(na mieste) výpočtu integrálneho obrazu. Táto redukcia zvýšila rýchlosť algoritmu o 3-5% pri obrázkoch s rozlíšením 1920x1080, a pri obrázkoch veľkosti 96x160 dokonca až o 15%, keďže pri malých rozlíšeniach, volania API predstavujú viac ako polovicu výpočtu deskriptora. Taktiež to znížilo potrebnú pamäť o 40%. Samotnému in-place algoritmu sa budem venovať pri popise optimalizácie výpočtu integrálneho obrazu.

Následný krok je kopírovanie vstupu na GPU, to je pomocou metódy cudaMemcpy2D, keďže používame 2D pole. Potom som použil metódu cudaMemset2D aby som inicializoval pole gradientBins na 0. Je to kvôli charakteru výpočtu gradientu. Predpokladáme 9 gradientových košov, výpočet gradientu napríklad zapíše hodnoty do miest s indexom 2 a 3, a miesta 0,1,4,5,6,7,8 budú naďalej obsahovať náhodnú hodnotu. Ak tieto miesta nie sú nastavené na 0, tak pri výpočte integrálneho obrazu dodávajú nežiadúce výsledky.

Zvyšok hlavného tela volania CUDA API, sú volania kernelov, kopírovanie výstupnej pamäte a uvoľnenie pamäte. Toto ukončuje časť implementácie, ktorá je na CPU strane. Okrem už spomínaného použitia štruktúry a vyradenia používania poľa, sa na tejto strane nemá moc čo optimalizovať, ak to nemá dočinenia s volaním kernelov.

Keďže zarovnanie pamäte je veľmi dôležitý aspekt pri optimalizácií, treba spomenú spôsob akým ukladám dáta do polí. Vstupné pole je obrázok, ktorý chceme popísať deskriptorom. Na CPU strane je to matica mxn, ktorá sa pri kopírovaní na GPU vloží do 2D poľa s rozmermi mxn, s tým rozdielom, že vďaka použitej metóde alokácie, riadky matice sú zarovnané po seba aj vo fyzickej pamäti.

Ďalšie je pole pre gradientové koše, do ktorých sa vkladá vážený/diskrétny gradient v prvom kroku algoritmu, a po použití in-place algoritmu pre výpočet integrálneho obrazu, aj samotný integrálny obraz. Toto pole má toľko riadkov čo vstupný obraz, a počet stĺpcov je počet stĺpcov vstupného obrazu\*počet košov(binov). Preto v riadku poľa budú za sebou uložené hodnoty všetkých orientácií pre daný bod, pre každý bod v riadku.

Posledné pole je pole okien deskriptora pre daný obraz. Toto je výsledné pole, ktoré sa po normalizácií kopíruje späť na CPU. Toto pole má počet riadkov = počtu okien a počet stĺpcov = počet blokov\* počet buniek\* počet košov(to je vlastne počet vektorov jedného okna deskriptora).

Kernely

Samotné kernely sú členené rovnakým spôsobom ako kroky CPU implementácie. Sú to výpočet gradientových košov, výpočet integrálneho obrazu, výpočet buniek, a normalizácia v rámci blokov.

**Prvý kernel** je výpočet gradientu. Samotný priebeh algoritmu je identický s popisom pri CPU implementácii. Hlavný rozdiel je štruktúra priebehu kroku. Zatiaľčo pri CPU prebieha v dvoch cykloch, kernel musí byť rozdelený na nezávislé časti, ktoré môžu prebiehať paralelne. Kvôli charakteru algoritmu(každý bod sa počíta osobitne) sa tento krok dá paralelizovať veľmi jednoducho. Každý bod v obraze bude mať jedno vlákno, počet vlákien v bloku je počet stĺpcov a počet blokov je počet riadkov.

V prípade ak počet stĺpcov je väčší ako 1024(maximálny počet vlákien na jeden blok), pri volaní kernelu je najvyššia mocnina 2, ktorá delí počet stĺpcov, ten sa pridá do rozmerov gridu(počtu blokov), a delenec sa použije ako počet vlákien v bloku. Ako príklad pre obraz rozmerov 1920x1080, 1920-2(na každom okraji sa jeden pixel nepočíta)=1918, 2 je najvyššia mocnina dvojky ktorá delí toto číslo, tzn. rozmery gridu budú (1078,2,1) a rozmery blokov(1918/2,1,1).

Optimalizačné kroky v tomto kerneli boli vykonávané súbežne s krokmi popísanými na CPU verzií. Oproti CPU verzií sa výpočet líši v použijej arkus tangens funkcii, zatiaľčo na CPU je použitá bežná atan funkcia z knižnice math.h, na GPU používam atanf. Tá počíta s datovým typom float, tzn. bežnou presnosťou narozdiel od dvojitej. Je to z dôvodu pomalšej rýchlosti vykonávania operácií s dvojitou presnosťou na GPU(každý warp má menej „double precision units“ ako samotných jadier).

**Druhý a tretí kernel** je výpočet integrálneho obrazu. Prvý návrh tohto kroku je identický so sekvenčnou verziou algoritmu tak ako je popísaný pri CPU verzií. Problém je, že výpočty v tomto kroku, tak ako je navrhnutý na CPU nemožno paralelizovať(okrem paralelného počítania každej orientácie, 9 vlákien na GPU je ale príliš málo). Preto bolo potrebné navrhnúť druhú verziu, tak ako je popísaná v kapitole **4.1.2.** Prvý kernel vypočíta prechod pre všetky stĺpce, rozmery gridu sú (počet stĺpcov/najväčšia mocnina 2, ktorá delí stĺpce;1;1) a rozmery bloku sú (najväčšia mocnina 2, ktorá delí stĺpce;počet binov;1). Druhý kernel vypočíta finálnu hodnotu, prechodom cez všetky riadky. Rozmery gridu a blokov sú rovnaké ako pri prvom kerneli len tentokrát pre počet riadkov. Vstupné pole je pole gradientových košov z prvého kernelu.

Tento kernel sa dal nahradiť in-place verziou, takže vstupné pole sa dá využiť aj na následné výpočty a aj na finálny výsledok. Toto umožnilo zmenšiť spotrebu pamäte, potrebný čas na alokáciu dát a na uvoľnenie dát(najviac poznať pri obrázkoch malých rozmerov).

Pre porovnanie, prvá sekvenčná verzia algoritmu mala pre 96x160 až 10x pomalšie vykonávanie ako výpočet integrálneho obrazu na CPU. Druhá verzia už produkovala omnoho lepšie výsledky. Pre rovnaký obrázok CPU výpočet integrálneho obrazu trvá 0.6 ms a pri GPU 0.3ms.

**Štvrtý kernel** je výpočet buniek pomocou integrálneho obrazu. Každá bunka obsahuje jeden vektor pre každú orientáciu integrálneho obrazu. Vstupné pole obsahuje integrálny obraz, a výstup je pole vektorov HOG deskriptora. Tento kernel má rovnaké rozmery gridu a blokov ako piaty kernel, tzn. normalizácia. Grid je (počet okien v stĺpci, počet okien v riadku, počet blokov v stĺpci jedného okna) a blok je (počet blokov v riadku jedného okna, počet buniek v stĺpci jedného bloku, počet buniek v riadku jedného bloku). Tieto rozmery zaručujú jednoduchú hierarchiu pohybu výpočtu, najprv v rámci bloku, následne v rámci okna, a nakoniec v rámci celého obrazu.

V prípade tohto kroku, CPU verzia už obsahuje optimalizovaný algoritmus(oproti pôvodnému HOGu kde sa bunky počítajú z pixelov, a tu sa počítajú integrálnym obrazom). Prevedenie CPU verzie na GPU poskytovalo 4násobné zrýchlenie. Keďže každá bunka je pomocou integrálneho obrazu nezávislý výpočet, výpočet každej bunky prebieha v jednom vlákne. Preto sa prístup k optimalizácií tohto kroku uberal iným smerom ako u predošlého(výber lepšie paralelizovateľnej úlohy), keďže tento krok má dostatočnú úroveň paralelizácie.

Boli testované tri verzie tohto kernelu, prvá bol výpočet jednej bunky v jednom vlákne pre jednu orientáciu. Druhá bol výpočet jednej bunky v jednom vlákne pre všetky orientácie, a tretia výpočet niekoľko buniek v jednom vlákne pre jednu orientáciu. Najrýchlejšia verzia bola druhá, voči prvej zrýchlenie poskytovalo vytvorenie menej kernelov a celkové zredukovanie výpočtov potrebných na prístup do pamäte, keďže orientácie sú v pamäti uložené hned za sebou. V tretej verzii sa síce zmenšil počet potrebných vlákien, ale výpočty pre presun medzi bunkami sú omnoho zložitejšie ako pre presuny medzi orientáciami, v prípade ak nepreorganizujeme celú štruktúru dát. Druhá verzia oproti prvej ponúka 4násobné zrýchlenie pre 9 orientácií(binov).

**Piaty kernel** je normalizácia buniek. Narozdiel od CPU verzie, som oddelil tento krok od výpočtu buniek. Napriek tomu, že tieto dva kroky spolu súvisia, oba potrebujú veľa réžie ohľadom výpočtom indexov do polí. Vtedy začína počet dostupných registrov obmedzovať úroveň paralelizácie. Vieme, že každý fyzický warp má určitú veľkosť súboru na registre. Preto ak nejaké vlákno používa viac ako povedzme 50 registrov, nastáva problém so súbežným vykonávaním. Ak by sme naimplementovali výpočet buniek a normalizáciu v jednom kerneli, potrebný počet registrov presiahne 100. Tento fakt má následne negatívny vplyv na výkon kernelu. V prípade rozdelenia, používa výpočet buniek 50 registrov a normalizácia 74.

Následné testovanie ukázalo, že s využitým spôsobom ukladania dat, je najlepšie naraz normalizovať 9 vektorov v jednom vlákne kernelu normalizácie. Testoval som kernel postupným zvyšovaním vektorov na kernel(začal som u 1) a čím viac som zvyšoval počet vektorov od 9, tým viac sa výkon kernelu zhoršoval.

Následovala fáza globálnych zmien v kerneloch. Tak ako u CPU sa osvedčila náhrada if blokov jednoduchšími ternárnymi operátormi. Ako príklad môžem uviesť normalizáciu. Normalizujeme blok len ak daný blok je nenulový, tzn. ak sa norma rovná nule, preskočím normalizáciu keďže sa chceme vyhnúť deleniu nulou.

Rýchlosť kernelov taktiež ovplyvnila práca s parametrami kernelov. Vyčistenie kódu, minimalizovanie počtu potrebných parametrov a taktiež posielanie už spracovaných parametrov pomohlo zrýchliť kód. Pod spracovaním parametrov chápeme nasledovný príklad: do kernelu sa posiela parameter počet košov a parameter počet buniek. V kerneli sa používa parameter počet buniek len v prípade vzorca koše\*početBuniek. Preto do kernelu posielam priamo tento predpočítaný vzorec a tým sa dosiahne zredukovania potrebných registrov a zároveň sa zrýchli výpočet vlákien.

Posledná značná globálna optimalizácia bola štrukturovanie kódu okolo cyklov for. V prípade skokov pri cykli for sa ukladajú registre do pamäte, a toto môže spôsobovať väčšie využitie súboru registrov a taktiež môže spomaliť vlákno pri ukončení cyklu. Preto som každý výpočet v kóde, presunul presne tam kde ho je treba. Ak napríklad potrebujem výpočet indexu, kam uložiť vektor pri normalizácií, tento výpočet počítam až pred cyklom, ktorý normalizuje, a nie na úplnom začiatku, ešte pred výpočtom normalizačnej hodnoty. Taktiež som vždy zredukoval ukladanie medzivýpočtov(napríklad indexov), hlavne v prípadoch keď ten medzivýpočet sa používa len na jednom mieste. Tým som presunul optimalizáciu na kompilátor, ktorý prepíše kód omnoho efektívnejšie.

Testovací dataset

Na testovanie algoritmov bol použitý INRIA Person Dataset [7], použitý pri testovaní práce Dalala a Triggsa [10], pôvodnej práci o HOG deskriptore. Tento dataset obsahuje niekoľko skupín obrázkov. Ako prvá je trénovacia skupina obrázkov, obsahujúca 1218 negatívnych obrázkov rôznych veľkosti a 2416 pozitívnych obrázkov s rozmerom 96x160. Ďalej tu máme testovaciu skupinu obsahujúcu 453 negatívnych obrázkov rôznych rozmerov, a 1126 pozitívnych obrázkov s rozmerom 70x134(jedno 64x128 okno a dačo naviac). Následne sú tu ešte obrázky rôznych rozmerov pre iné nastavenia HOG deskritprora.

Parametre HOGa pre tento dataset sú nasledovné:

* Veľkosť okna: 64x128
* Veľkosť bloku: 16x16
* Veľkosť bunky: 8x8
* Veľkosť posunu okna: 16x16
* Veľkosť posunu bloku: 8x8
* Počet gradientových košov(binov): 9

Pre tento dataset som používal lineárne SVM, ktoré sa zastavuje po minimálnej zmene ďalšieho kroku o 0,01 alebo po minimálnom počte krokov(podľa rôznych hodnôt).

Vyhodnotenie

Hardware ktorý bol použitý pri testovaní:

CPU: Intel Core i7 4770

GPU: MSI GTX 970 4G

RAM: 24GB 1600 MHz CL9(4,4,8,8GB)

HDD: 1TB Caviar Black Western Digital

Základná doska: Gigabyte Z87-HD3

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Verzia\Sekcia | Alokácia pamäte | Výpočet Gradientu | Integrálny obraz | | Výpočet buniek | Normalizácia | Dealokácia | |
| Prvá CPU(96x160) | 2000 | 1100 | 230 | | 250 | | 1250 | |
| Prvá GPU(96x160) | 1050 | 165 | 132 900 | | 420 | 135 | 460 | |
| Posledná CPU(96x160) | 1200 | 780 | 230 | | 230 | | 650 | |
| Posledná GPU(96x160) | 520 | 82 | 200 | 115 | 40 | 55 | 159 | |
| Prvá CPU(1920x1080) | 308600 | 120500 | 33500 | | 215000 | | 182000 | |
| Posledná CPU(1920x1080) | 192900 | 85300 | 33500 | | 196700 | | 91300 | |
| Posledná GPU(1920x1080) | 3266 | 1170 | 1917 | 4500 | 7900 | 7200 | 34000 | 2279 |

Pre porovnanie zrýchlenia som zvolil vždy prvú a poslednú verziu daného algoritmu. Niektoré zmeny v boli velmi nepatrné a odlišovaii sa povedzme iba v desatinách milisekúnd. Taktiež bolo pri časovaní nutné prestať používať profiler, pretože overhead ktorý pridával k vykonávaciemu času GPU metód v posledných štádiách, zdvojnásoboval vykonávací čas.

Tabuľka - Porovnanie vykonávacieho času jednotlivých časti algoritmu(v μs)

Ako prvé máme dáta o vykonávaní samotných sekcií algoritmu(tabuľka 1). Tieto hodnoty boli namerané kombináciou CUDA profilera a QueryPerformanceFrequency z knižnice windows. V oboch prípadoch časovanie pridalo určitý overhead a preto celkový čas algoritmov je vyšší ako pri bežnom chode(20%+- pre 96x160 rozmer pre GPU časovanie, pri 1920x1080 zanedbatelné). Pri GPU verzií je výpočet integrálneho obrazu a výpočet buniek a ich normalizácia rozdelená na dve časti, dohromady dávajú ekvivalent kroku u CPU. Taktiež prvá GPU verzia pre 1920x1080 trvala dlhšie ako 5 minút, preto som neuviedol jej časy.

Ak všeobecne porovnáme časy pre rovnaké rozmery, dostávame celkové zrýchlenie až na pár výnimiek Zatiaľčo finálna GPU implementácia poskytuje zrýchlenie v skoro každej časti algoritmu, výpočet integrálneho obrazu je v CPU implementácii rýchlejší ako na GPU. V prvej implementácii bol algoritmus pre výpočet integrálneho obrazu identický, a to produkovalo obrovské spomalenie. Vo finálnej implementácii porovnávame sekvenčný CPU algoritmus, a paralelný GPU algoritmus. Tu už možno vidieť, že CPU algoritmus je len o niečo rýchlejší ako pri GPU, hlavne ak berieme do úvahy, že časovanie na GPU má určitý overhead. Pri rozmere obrázkov 1920x1080 je ešte výraznejšia zmena v rýchlosti medzi CPU a GPU verziami.

V oboch prípadoch je práca s pamäťou veľmi drahá časť algoritmu. V prípade obrázkov 96x160 je pomer výpočtov/pamäte veľmi malý(pre vybrané parametre počítame iba 9 HOG okien). Preto samotný výpočet deskriptora je polovičný oproti správe pamäte. Ak však vezmeme čas pre rozmer 1920x1080, samotná alokácia pamäte je omnoho menšia. Dealokácia sa v tomto prípade rozdeľuje na kopírovanie deskriptoru na CPU, a samotnú dealokáciu. Pri tomto rozmere, viac ako polovica času potrebného na výpočet, zaberá kopírovanie do CPU pamäte(rýchlosť približne 2,5GB/s).

Ďalšie máme porovnanie globálneho času metód, voči referenčnej implementácii.Tieto hodnoty boli namerané iba pomocou QueryPerformanceFrequency, čiže overhead je veľmi malý a na hodnotách sa skoro vôbec nezobrazí.

Pre ilustráciu som tu ponechal údaje prvej implementácie pre CPU aj GPU. Tak ako bolo vidieť na predošlých dátach(tabuľka 1), aj tu je vidieť že prvotný výpočet integrálneho obrazu na GPU bol veľmi pomalý, a časové hodnoty boli veľmi vysoké.

Môžeme si všimnúť, že pri rozmere 96x160 finálna implementácia má rovnaký výkon ako referenčná implementácia. Tu je vidieť, aký vplyv má pri malých obrázkoch overhead volaní CUDA API. Zatiaľčo OpenCV verzia nemusí kopírovať dáta z pamäte do pamäte, a alokuje dáta priamo na CPU.

Tento pomer výkonu sa zmení pri výpočte veľkých obrázkov, napríklad Full HD. V tomto prípade je GPU implementácia 2,5 rýchlejšia ako referenčná implementácia. V tomto prípade je využitá väčšia časť potenciálu GPU, keďže sa vytvára vysoké množstvo vlákien. Pri malých obrázkoch pri výpočte integrálneho obrazu sa stáva, že počet vlákien je menší ako počet dostupných procesorov(pre použité GPU, obrázok 96x160, využíva naraz 96 vlákien, a GPU má 1536 jadier), tzn. využitie hardwaru nie je perfektné.

|  |  |  |  |
| --- | --- | --- | --- |
| Implementácia\verzia | OpenCV Hog | CPU | GPU |
| Prvá verzia 96x160 | 0,75ms | 5ms | 150ms |
| Finálna verzia 96x160 | 3,4ms | 0,75ms |
| Prvá verzia 1920x1080 | 145ms | 740ms | 10000ms |
| Finálna verzia 1920x1080 | 600ms | 60ms |
| Finálna verzia 70x134 | 0,4ms | 2ms | 0,9ms |

Tabuľka - Porovnanie celkového času pre rôzne implementácie HOG deskriptora

Ako posledný údaj sú obrázky najmenšieho testovaného rozmeru. Tu možno vidieť, že referenčná implementácia je o viac ako polovicu rýchlejšia ako GPU, a taktiež CPU verzia je výkonom bližšie. Taktiež je si vhodné všimnúť,že rozmer 70x134 má vyšší čas ako rozmer 96x160 pri GPU implementácii. V tomto prípade nastáva problém so zvolením veľkosti bloku. Heuristika, ktorú som používal(delenie čo najvyššou mocninou 2), je pri tomto rozmere neefektívna, najväčšiu mocninu, ktorú nájde je 2 zatiaľčo najväčšia mocnina pri 96x160 je 4. Tento poznatok ilustruje, ako veľmi záleží na analýze CUDA modelu, keďže to spôsobuje klesanie výkonu.

Spotreba pamäte algoritmov je v podstate identická aj pre CPU a GPU implementácii, s pár rozdielmi. Vstupné a výstupné pole pri GPU implementácii musí byť aj v RAM aj v GPU pamäti. Taktiež treba brať do úvahy obmedzenia GPU pamäte, keďže CPU pamäť je ľahko rozšíriteľná. Pre obrázok veľkosti 96x160 je spotreba pamäte GPU 1,4 MB a pre obrázok 1920x1080 to je 210MB. Toto obmedzuje počet súbežných výpočtov, ktoré môžu naraz bežať na GPU, v prípade ak by sme chceli počítať viac operácií naraz.

Posledné máme porovnanie detekcie. Táto časť bola komplikovaná, keďže síce moja a OpenCV implementácia majú rovnaký základ ale napriek tomu sú dostatočne rozdielne na to, aby potrebovali rôzne nastavenie pri trénovaní. Bolo testovaných okolo 5 konfigurácií pre môj HOG(CPU aj GPU produkujú rovnaké vektory), a okolo 25 konfigurácií pre OpenCV HOG(pre rôzne počty trénovacích obrázkov a neskôr pre rôzne počty iterácií).

Hodnoty, ktoré som nameral(tabuľka 3), boli namerané na súbore 1126 pozitívnych obrázkov(1126 deskriptorových okien) a 453 negatívnych obrázkov(227037 deskriptorových okien). Dáta sú uvedené v tvare počet nájdených pozitívnych okien v pozitívnych obrázkoch/počet nájdených false positives v negatívnych obrázkoch.

|  |  |  |  |
| --- | --- | --- | --- |
| Trénovacie parametre\verzia | OpenCV Hog(vlastné SVM) | OpenCV Hog(vstavané SVM) | CPU/GPU |
| 300 poz/100 neg | 1074/5301 | 754/13 | 827/16867 |
| 420 poz/100 neg/40000 iterácií | - | 857/15919 |

Tabuľka -Výsledky trénovania SVM(nájdené osoby/false positives) pre 1126 pozitívnych a 453 negatívnych obrázkov.

Zatiaľčo môj HOG je čistý výpočet HOG deskriptora a jeho vektorov, OpenCV HOG bol dizajnovaný v spojení so vstavaným SVM. Pre rovnaké nastavenia parametrov trénovania SVM, môj HOG podáva lepšie výsledky ako OpenCV HOG.

Problém s týmto údajom je, že OpenCV HOG môže mať úplne iný vzťah k pretrénovaniu. Zatiaľčo môj algoritmus bol jednoduchší na nastavenie, a trénovanie pre rovnaké parametre trvalo niekedy až o 2,5x menej času, je veľmi pravdepodobné, že OpenCV HOG nie je možné až tak efektívne prispôsobiť k použitiu v spojení s vonkajším SVM.

Ak porovnáme moju implementáciu so vstavaným OpenCV HOG SVM, tak dostávame možno na prvý pohľad rôzne údaje. Zatiaľčo moja implementácia má lepšiu detekciu ľudí, OpenCV HOG produkuje rádovo menej false positives. V práci o HOG deskriptore [11] bolo spomenuté, že ak zvýšime nastavením parametrov detekciu ľudí, taktiež zvýšime počet false positives. Taktiež treba brať do úvahy, že vstavané SVM bolo trénované komplikovanejším spôsobom(napríklad v práci spomenutým pretrénovávaním). Preto ak zvážime všetky tieto poznatky, môžeme povedať, že mnou implementovaný HOG má podobnú úroveň detekcie.

Úvahy na zlepšenie

Navrhnutý algoritmus ponúka ešte niekoľko smerov, ktorým je potenciálne možné dosiahnuť zrýchlenie výkonu. Prvým je využitie iného algoritmu na výpočet integrálneho obrazu. Ako príklad je práca o GPU efektívnych integrálnych obrazoch [15]. Práca hovorí o výpočte integrálneho obrazu pre sekcie 2x2, a následnými operáciami ako efektívne a paralelne spočítať tieto hodnoty do kompletného integrálneho obrazu.

Ďalším smerom, ktorým by sa dalo uberať je hlbšia analýza modelu CUDA. V práci som spomínal aký je vzťah medzi warpmi, blokmi a zarovnaním pamäte. Pokiaľ by sa niektoré sekcie algoritmu mohli upraviť tak, aby lepšie pracovali na týchto najzákladnejších jednotkách hardwaru, určite by to poskytlo značné zrýchlenie. Ako príklad môžeme brať zdieľané výpočty buniek, pri výpočte blokov, alebo využitie zdieľanej pamäte pri výpočte normalizácie(kernely normalizujú v jednom vlákne toľko vektorov koľko je košov, ale hodnota ktorou normalizujú je rovnaká pre celý blok). Taktiež je možné navrhnúť nejaký dynamický systém počtu vlákien v bloku(ideálny počet vlákien na jeden blok je 256, ale vyžaduje to dodatočnú réžiu), ktorý by zrýchlil vykonávanie kernelov. Problém nastáva ak vlákna divergujú, tzn. inštrukcie, ktoré sú v danom momente potrebné sú rôzne.

Taktiež je možné preorganizovať spôsob uloženia dát v poliach, s ktorými pracujeme. Tento krok vyžaduje veľa experimentovania, a častokrát prepísanie celého kernelového výpočtu indexov.

Ako ďalšiu možnosť môžeme uviesť rozdelenie spôsobov výpočtu pre malé a pre veľké obrazy. Keďže API volania predstavujú značnú časť času potrebného na výpočet HOGa, mohla by sa navrhnúť nejaká verzia, ktorá by znovu používala polia pre vstupný obraz na integrálny obraz. Taktiež by bolo možné navrhnúť výpočet gradientu tak aby nebolo potrebné nastavovať pred výpočtom pole na 0(memset pri malých obrázkoch je celkom drahá operácia). Tieto zmeny by poskytovali značné zrýchlenie pre malé rozmery(tam API volania predstavujú viac ako polovicu) a pre veľké rozmery by poskytli aspoň inkrementálnu optimalizáciu.

Na koniec treba spomenúť možnosť obmedzenia flexibility algoritmu. Pôvodný hog algoritmus obsahuje mnoho parametrov ktoré boli testované. Môj algoritmus síce používal dané parametre pre zvolený dataset, ale napriek tomu je nastaviteľný pre rôzne veľkosti blokov/buniek atď. Ak by sa táto flexibilita obmedzila/odstránila, tzn. algoritmus by bol použiteľný iba na danú množinu parametrov, bola by možnosť optimalizovať oblasti, ktoré sa normálne až tak efektívne nedajú(napríklad veľkosti blokov vlákien, gridu blokov, alebo aj „unroll“ cyklov).

# Záver

Cieľom tejto práce bolo zhodnotiť možnosti aproximácii HOG deskriptora a ich možnú implementáciu na CPU a následne na GPU pomocou CUDA modelu. Pre porovnanie algoritmov sme zvolili porovnanie trvania implementácii, a porovnanie detekcie pomocou SVM.

Na pochopenie vnútorných mechanizmov CPU a GPU sme sa venovali ich vývoju v histórií. Prešli sme rôzne faktory, ktoré obmedzujú výpočtovú rýchlosť samotného hardwaru a ako možno navrhnúť algoritmus tak aby sme využívali čo najvyššie percento výkonu hardwaru. Oboznámili sme sa s CUDA modelom a ponúkanými možnosťami paralelizovať algoritmy na grafických kartách. Následne sme popísali knižnicu OpenCV, v ktorej sú implementované rôzne algoritmy počítačového videnia. Na konci teoretickej časti sme sa oboznámili s HOG deskriptorom, jeho fungovaním, a taktiež s ním spojenými pojmami.

V praktickej časti sme sa oboznámili s OpenCV HOG implementáciou. Detailne sme prešli návrhové rozhodnutia pri oboch implementáciách a ako tieto rozhodnutia ovplyvnili výkon algoritmu. Následne sme uviedli INRIA dataset, na ktorom boli testované všetky časti algoritmov, a taktiež hardware, na ktorom boli implementácie spúšťané. Vo vyhodnotení sme porovnali časové hodnoty implementácii pre rôzne rozmery obrázkov. Taktiež sme prešli pamäťové nároky GPU implementácie. Ako poslednú sme zhodnotili kvalitu deskriptorov pomocou vyhodnotenia výsledkov detekcie na INRIA datasete. Dospeli sme k záveru, že detekcia všetkých implementácii dosahuje podobné výsledky a z pohľadu tohto testovania sú všetky naše implementácie porovnateľné s referenčnou implementáciou. Nasledovalo zhodnotenie možných vylepšení algoritmu, a ktorým smerom sa môžu ďalšie optimalizácie uberať.

S prihliadnutím na podané výsledky práce, môžeme povedať, že paralelizácia výpočtov na GPU pri použití HOG deskriptora, je správny prístup k optimalizácií tohto algoritmu. S vývinom GPGPU sa možnosti zrýchlenia budú len rozširovať.

# Zoznam Skratiek

AGP- Accelerated Graphics Port

AMD- Advanced Micro Devices

API- Application Programming Interface

ATI- Array Technology Inc.

BSD- Berkeley Software Distribution

CPU- Central Processing Unit

CUDA- Compute Unified Device Architecture

GDDR5- Graphics Double Data Rate memory, generation 5

GPU- Graphics Processing Unit

GPGPU- General Purpose Graphics Processing Unit

HOG- Histogram of Oriented Gradients

IBM- International Business Machines

INRIA- Institut national de recherche en informatique et en automatique

IPP- Intel Performance Primitives

LDA- Load Accumulator

LLVM- Low Level Virtual Machine

MatLab- Mathematic Laboratory

OpenCL- Open Computing Language

OpenCV- Open Source Computer Vision

PCB- Printed Circuit Board

RAM- Random Access Memory

RCA- Radio Corporation of America

SGI- Silicon Graphics Inc.

SIMD- Single Instruction, Multiple Data

SIMT- Single Instruction, Multiple Threads

SMT- Simultaneous Multithreading

SVM- Support Vector Machine

# Referencie

|  |  |
| --- | --- |
| [[1] | J. Laird, „Haswell: everything you need to know about Intel's latest Core processors,“ techradar.com, 26 6 2014. [Online]. Available: http://www.techradar.com/news/computing-components/processors/haswell-everything-you-need-to-know-about-intel-s-latest-core-processors-1156004/3. [Cit. 5 5 2015]. |
| [[2] | [Online]. Available: http://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips/2. |
| [[3] | „Open source OpenCL implementácie,“ Kronos Group, [Online]. Available: https://www.khronos.org/opencl/resources/opencl-open-source-opencl-implementations. |
| [[4] | K. Group, „komerčné OpenCL implementácie,“ [Online]. Available: https://www.khronos.org/opencl/resources/opencl-commercial-implementations. |
| [[5] | „Intel® Core™ i7-5820K Processor,“ Intel Corporation, [Online]. Available: http://ark.intel.com/products/82932. [Cit. 2 5 2015]. |
| [[6] | „GeForce GTX 980 Specifications,“ NVIDIA, [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications. [Cit. 2 5 2015]. |
| [[7] | C. K. J. C. M. D. K. A. D. N. N. S. M. S. C. P. H. R. S. a. P. D. Victor W Lee, „Debunking the 100X GPU vs. CPU Myth:An Evaluation of Throughput Computing on CPU and GPU,“ Intel Corporation, Saint-Malo, France, 2010. |
| [[8] | „Tesla GPUs aid early detection of dangerous arterial plaque earlier,“ NVIDIA, [Online]. Available: http://www.nvidia.co.uk/object/harvard-brigham-uk.html. [Cit. 2 5 2015]. |
| [[9] | OpenCV.org, „OpenCV,“ OpenCV.org, [Online]. Available: opencv.org. [Cit. 4 5 2015]. |
| [[10] | OpenCV.org, „OpenCV Documentation,“ OpenCV.org, [Online]. Available: http://docs.opencv.org. [Cit. 4 5 2015]. |
| [[11] | N. D. a. B. Triggs, „Histograms of Oriented Gradients for Human Detection,“ INRIA Rhˆone-Alps, 655 avenue de l’Europe, Montbonnot 38334, France, 2005. |
| [[12] | M. J. J. Paul Viola, „Robust Real-Time Detection,“ Kluwer Academic Publishers, Redmond, Cambridge USA, 2001. |
| [[13] | J. Brauer, „HOG descriptor computation and visualization,“ [Online]. Available: http://www.juergenwiki.de/work/wiki/doku.php?id=public:hog\_descriptor\_computation\_and\_visualization. [Cit. 4 5 2015]. |
| [[14] | A. K. T. M. A. T. C. Vondrick, „HOGgles: Visualizing Object Detection Features,“ International Conference on Computer Vision (ICCV), Sydney, Australia, 2013. |
| [[15] | N. Dalal, „INRIA Person Dataset,“ [Online]. Available: http://pascal.inrialpes.fr/data/human/. [Cit. 4 5 2015]. |
| [[16] | A. M. R. S. H. H. Diego Nehab, „GPU-Efficient Recursive Filtering and Summed-Area Tables,“ ACM Trans. Graphics, 2011. |

# Zoznam použitých obrázkov

[Obrázok 1-Rozdiel medzi nárastom výkonu CPU a pamäte 14](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419142995)

[Obrázok 2- Dizajnový nákres Maxwell architektúry GPU 19](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419142996)

[Obrázok 3-Nsight profiler v móde Profile Cuda Application 21](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419142997)

[Obrázok 4-Nastavenia Nsight profilera 21](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419142998)

[Obrázok 5-Príklad kernel kódu 23](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419142999)

[Obrázok 6- Transformácia pixelov na bunkový histogram 25](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143000)

[Obrázok 7- Ilustrácia fungovania normalizácie 26](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143001)

[Obrázok 8- Vizualizácia smeru gradientu pri zmene farby 27](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143002)

[Obrázok 9- Sobel Maska 28](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143003)

[Obrázok 10- Ilustrácia hodnôt integrálneho obrazu(vpravo) k pôvodnej matici(vľavo) 29](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143004)

[Obrázok 11- Príklad optimalizačnej úlohy, ktorú sa snaží SVM riešiť. 33](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143005)

[Obrázok 12- Obrázok chodcov 37](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143006)

[Obrázok 13- Gradient k obrázku 12 37](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143007)

[Obrázok 14- Ilustrácia výpočtu obdĺžnika pomocou integrálneho obrazu 39](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143008)

[Obrázok 15- Vizualizácia buniek HOG dekritptora 41](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143009)

[Obrázok 16-Timeline view CUDA profilera 42](file:///D:\Users\killerwife\C%20source\Cuda%20shtuff\Bakalarka%20words\FrantišekKajánekBakalárskaPrácaHOG.docx#_Toc419143010)

# Zoznam Tabuliek

[Tabuľka 1- Porovnanie vykonávacieho času jednotlivých časti algoritmu(v μs) 50](#_Toc419143011)

[Tabuľka 2- Porovnanie celkového času pre rôzne implementácie HOG deskriptora 52](#_Toc419143012)

[Tabuľka 3-Výsledky trénovania SVM(nájdené osoby/false positives) pre 1126 pozitívnych a 453 negatívnych obrázkov. 53](#_Toc419143013)

# Zoznam príloh

**Príloha A -** CD médium – bakalárska práca, zdrojové kódy a súbory implementácii HOG deskriptora a CUDA profiler dáta nazbierané počas programovania