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

SLEDOVÁNÍ POHYBLIVÉHO OBJEKTU VE VIDEU NA CUDA

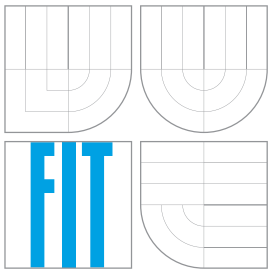
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MIROSLAV SCHERY

BRNO 2010

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

SLEDOVÁNÍ POHYBLIVÉHO OBJEKTU VE VIDEU NA CUDA

OBJECT TRACKING IN VIDEO USING CUDA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MIROSLAV SCHERY
AUTHOR

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

BRNO 2010

Abstrakt

Táto bakalářská práce se zabývá implementací algoritmu částicového filtru do technologie CUDA za účelem akcelerace jeho výpočtu. Věnuje se popisu metod sledování objektů ve videu a speciálně se zaměřuje na částicový filtr. Popisuje také architekturu CUDA. Vysvětleny jsou postupy implementace a optimalizace použité při vytváření aplikace. Práce je zakončena prováděním rychlostních testů a ověřením schopnosti algoritmu sledovat objekt na různých videích.

Abstract

This bachelors thesis is focused on implementation of particle filter algorithm in CUDA technology for the purpose of computation acceleration. It describes object tracking methods in video and is especialy aimed at particle filter. It also describes the CUDA architecture. Implementation and optimalization techniques used in application are explained. The work ends with performing speed tests and the tracking ability of the algorithm is verified on various videos.

Klíčová slova

sledování objektu, zpracování obrazu, částicový filtr, histogram , CUDA, GPU, paralelizace, optimalizace

Keywords

object tracking, image processing, particle filter, histogram, CUDA, GPU, parallelization, optimization

Citace

Miroslav Schery: Sledování pohyblivého objektu ve videu na CUDA, bakalářská práce, Brno, FIT VUT v Brně, 2010

Sledování pohyblivého objektu ve videu na CUDA

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Adama Herouta Ph.D.

.....
Miroslav Schery
19.05.2010

Poděkování

Děkuji vedoucímu práce Ing. Adamovi Heroutovi, Ph. D., za odbornou pomoc a cenné rady.

Obsah

Kapitola 1

Úvod

Informačná doba v poslednom desaťročí so sebou priniesla rozšírenie používania kamerových systémov do množstva odvetví ľudskej činnosti. Obraz však nemusí pozorovať len človek. S nástupom informačných technológii sa začalo rozvíjať spracovanie obrazových dát, zaoberajúce sa detekovaním a následným sledovaním objektov vo videu. Vzniklo niekoľko rôznych metód, žiadna však nie je univerzálna.

V súčasnosti nastal zlom, keď sa do popredia dostávajú technológie umožňujúce spracovanie veľkého množstva dát súčasne. Nazývajú sa GPGPU čo v preklade znamená „grafické procesory pre všeobecné využitie“. Jednou z týchto architektúr je CUDA od spoločnosti NVIDIA. Za necelé tri roky existencie na nej boli implementované tisícky algoritmov. Niektoré z nich dosiahli až stonásobné zrýchlenie oproti verziám počítaným na procesore. Medzi algoritmy ideálne k paralelizácií, patrí aj spracovanie obrazových dát. Vzniklo preto už niekoľko implementácii rozdielnych metód detekovania a sledovania objektov.

V tejto práci sa zameriame na metódu sledovania objektu pomocou Časticového filtra, pre ktorú neboli dostupné algoritmy pre paralelné zariadenia v čase výberu témy. Cieľom bude implementovanie algoritmu do technológie CUDA za účelom akcelerácie jeho výpočtu.

V druhej kapitole sa oboznámime s problematikou sledovania objektu vo videu. Popíšeme si niektoré algoritmy a zameriame sa na Časticový filter, ktorý bol vybraný k implementácií na CUDA.

Ďalšia kapitola sa venuje architektúre CUDA. Opísaný bude programový a pamäťový model a trochu obsiahlejšie sa venuje problému optimalizácie kódu.

V rámci návrhu analyzujeme vhodnosť paralelizácie jednotlivých celkov vybraného algoritmu. Získame základnú predstavu o štruktúre programu a použitých postupoch výpočtu. Naplánujeme konkrétne optimalizácie za účelom zistenia ich vplyvu na dobu spracovania jedného kroku algoritmu.

Časť Implementácia a testovanie popisuje použité funkcie a problémy, ktoré bolo treba riešiť. Pri jednotlivých optimalizáciach je diskutovaný dosiahnutý výsledný čas. Na konci tejto kapitoly vykonáme rýchlostné testy na výkonnostne rozdielnych zariadeniach a otestujeme funkčnosť časticového filtra na niekoľkých videách.

Kapitola 2

Algoritmy sledovania objektu

Táto kapitola čerpá informácie z [8].

Sledovanie pohybu objektu môžeme opísať ako problém určenia trajektórie objektu po jednotlivých snímkach videosekvencie. Niektoré algoritmy okrem polohy určujú aj iné vlastnosti ako orientáciu, plochu alebo tvar objektu. Väčšina algoritmov počíta s niekoľkými obmedzeniami pohybu objektu a zjednodušeniami jeho detekovania. Jedným z nich je predpoklad, že pohyb je plynulý bez náhlych prudkých zmien. Navyše môže byť očakávaný konštantný pohyb objektu alebo konštantná akcelerácia na základe informácie dostupnej pred začiatkom samotného sledovania.

Na algoritmy boli kladené rôzne kritéria. Vzniklo niekoľko rozdielnych metód sledovania objektov, pretože sa špecializovali na riešenie konkrétnych problémov. Používajú odlišné spôsoby reprezentácie sledovaného objektu a pracujú s inými vlastnosťami obrazu.

Odlišujú sa aj v schopnosti zvládať komplikácie, ktoré pri sledovaní vznikajú:

- šum a nepresnosti obrazu
- čiastočné alebo kompletné prekrytie objektu
- spracovanie obrazu v reálnom čase
- zmena v osvetlení scény
- zložitý tvar alebo pohyb objektu a iné

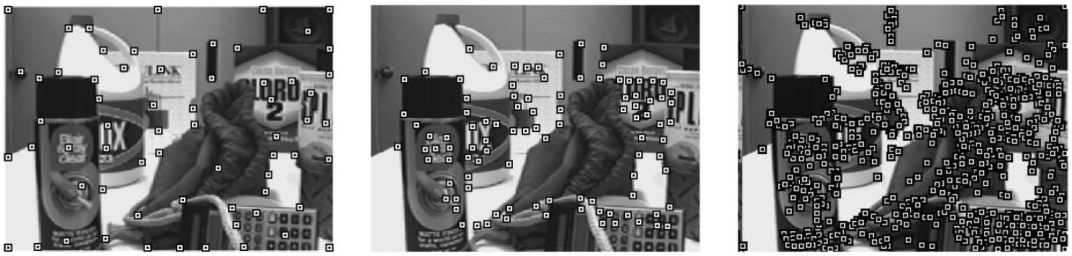
Metódy sledovania objektov rozdeľujeme do troch kategórií: sledovanie významných bodov (angl. *Point Tracking*), kernel tracking a sledovanie siluet (angl. Silhouette Tracking)

2.1 Sledovanie významných bodov

V tejto kapitole je objekt reprezentovaný skupinou významných bodov. Prepojenie detekovaných bodov do konkrétneho objektu je odvodené z jeho predchádzajúceho stavu, ktorého obsahom často býva informácia o pozícii a pohybe. Neoddeliteľnou súčasťou tejto metódy je detekovanie významných bodov v obraze. Medzi najznámejšie algoritmy patrí Moravec operátor, Harris operátor, KLT detektor a SIFT detektor.

Operátor Moravec hľadá významné body počítaním veľkosti zmeny intenzít farby obrázku. Ten rozdelí na okná o veľkosti 3x3, 5x5 alebo 7x7 pixelov a v nich sa v rámci výpočtu pohybuje v horizontálnom, vertikálnom a diagonálnom smere. Najnižšia hodnota z výsledných zmien intenzity sa vyberie pre každé okno. Významný bod je lokálnym maximom

v matici 12x12 zhotovenej z hodnôt reprezentujúcich jednotlivé okná. Zo spomenutých algoritmov je operátor Moravec najjednoduchší. Ostatné algoritmy sa venujú mimo iného aj zaisteniu odolnosti voči otáčaniu a posunu objektu. Snímku, na ktorej boli spustené spomínané algoritmy môžeme vidieť na obr. 2.1.



(a) (c) (c)

Obr. 2.1: Použitie operátorov (a) Harris (b) KLT a (c) SIFT (prevzaté z [8])

Sledovanie významných bodov objektu môžeme popísať ako hľadanie odpovedajúcich si objektov reprezentovaných významnými bodmi medzi jednotlivými snímkami. Metódy hľadania odpovedajúcich si bodov (angl. *point correspondence methods*) sa delia na deterministické a štatistické.

Deterministické metódy počítajú cenu spojenia všetkých detekovaných významných bodov v snímke v čase *t-1*, s jedným objektom zo snímky v čase *t*. Cieľom je nájsť riešenie, pri ktorom má každý bod priradený jeden odpovedajúci bod v novej snímke. Použiť môžeme niektoré z optimálnych metód pre priraďovanie, medzi ktoré patrí Maďarský algoritmus (angl. *Hungarian algorithm*) a tzv. *greedy search* metódy.

Štatistické metódy pri určovaní stavu objektu počítajú so šumom obsiahnutým vo videu a s nerovnomerným pohybom objektov. Pozíciu, rýchlosť či akceleráciu určujú pomocou vzorkovania obrazu či vytvorením funkcie hustoty pravdepodobnosti. Do tejto skupiny metód patrí Kalman filter a Časticový filter (angl. *Particle filter*).

Kalman filter sa používa pri počítaní stavu lineárneho systému. Predpokladá ale, že procesy systému sú Gausovské. Obsahuje dva kroky. Prvý krok je predpovedanie (angl. *prediction*), kde sa odhadne nový stav objektu na základe predchádzajúceho stavu. Krok opravy (angl. *correction*) následne spracovaním najnovších pozorovaní objektu, aktualizuje jeho stav. Každý z krokov sa môže vynechať, pokiaľ je to treba (napr. pri absencii nových pozorovaní sa prevedie prvý krok viackrát za sebou).

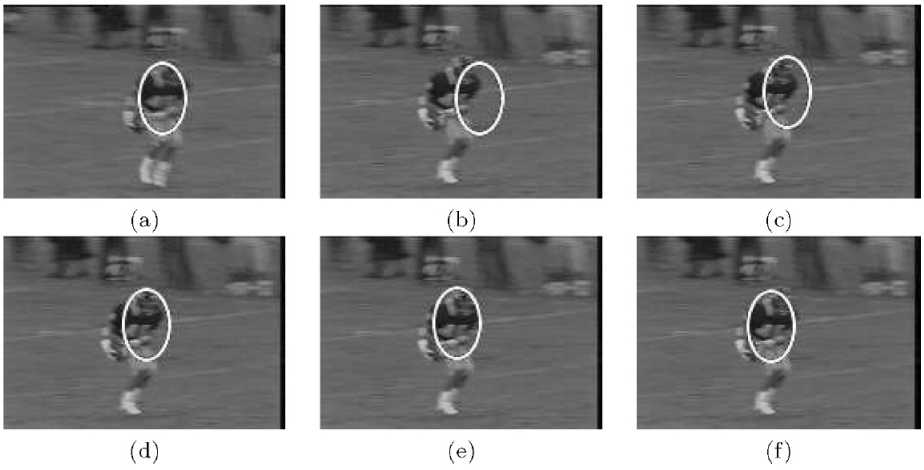
Časticový filter, na rozdiel od predchádzajúcej metódy, nie je obmedzený na počítanie s Gausovskými procesmi. Pomocou vytvorenia vzoriek (tzv. častíc), filter aproximuje hustotu pravdepodobnosti stavu systému. Viac v kapitole 2.4.

2.2 Kernel tracking

Kategória *Kernel tracking* nemá zaužívaný preklad. Jej metódy sledujú pohyb objektu, ktorého vzhľad sa dá popísať pomocou jednoduchých geometrických tvarov. Pohyb je väčšinou vyjadrený pomocou posunov, natočení a zmene veľkosti. V tejto sekcii sa algoritmy odlišujú hlavne použitou reprezentáciou vzhľadu a metódami pre určenie pohybu objektu.

Podľa reprezentácie vzhľadu ich rozdeľujeme do dvoch skupín. V prvej sa nachádzajú modely vzhľadu popísané šablónami (angl. *templates*) a tiež modely založené na hustote (angl. *density–based*, napr. pomocou histogramu). Druhá zahŕňa modely vzhľadu definovaného viacerými pohľadmi na objekt (angl. *multiview appearance models*)

Medzi najznámejšie algoritmy z prvej skupiny patrí mean–shift sledovanie objektu. Algoritmus sa približuje k čo najlepšej zhode so vzorovým objektom pomocou iteratívneho porovnávania histogramov vzorky objektu Q a miesta predpokladaného výskytu objektu P. Pomocou Bhattacharya koeficientu $\sum_{u=1}^{b} P(u)Q(u)$ vypočítame mieru zhody histogramov. Cieľom je dosiahnutie konvergencie podobnosti histogramov a to niekoľkými iteráciami výpočtu. Aby mean-shift metóda pracovala správne, musí sa objekt nachádzať vo vnútri regiónu vybraného pri inicializácii algoritmu. Pokiaľ je táto podmienka splnená, objekt bude zameraný za približne 6 iterácií. Veľkou výhodou algoritmu je absencia hľadania objektu pomocou hrubej sily. V obrázku 2.2 sú znázornené výsledky jednotlivých iterácií algoritmu mean–shift.



Obr. 2.2: Mean–shift algoritmus: (a) poloha objektu v čase $t-1$ (b) odhad polohy v čase t (c)(d)(e) iterácie algoritmu (f) konečná poloha objektu (prevzaté z [8])

Ďalším algoritmom na sledovanie objektov z prvej skupiny je KLT. Jeho úlohou je sledovať významné body detekované napríklad pomocou rovnomenného KLT detektora. Pracuje na princípe iteratívneho počítania posunu malých regiónov, ktorých centrom je významný bod. Ak je zistený príliš veľký rozdiel medzi súčasnou plochou a odpovedajúcou plochou v novej snímke, tento bod sa prestane sledovať.

Do druhej skupiny sa zaraďuje algoritmus Support Vector Machine (ďalej len SVM). Pomocou pozitívnych a negatívnych trénovacích vzoriek algoritmus určí črty odlišujúce tieto dve triedy. Medzi pozitívne sa zaraďujú snímky sledovaného objektu a do negatívnych spadajú časti pozadia, ktoré sa podobajú objektu. Výhodou *SVM* oproti ostatným algoritmom na sledovanie objektu je zahrnutie informácie o pozadí do rozhodovania.

2.3 Sledovanie siluet

Metódy zahrnuté v tejto kategórii sú určené na sledovanie objektov so zložitými tvarmi a výsledkom bývajú ich presné popisy. Delia sa na metódy porovnávajúce tvary (angl. *shape matching*) a sledovanie kontúr (angl. *contour tracking*).

Metódy porovnávajúce tvary sú podobné porovnávaniu šablón (viď 2.2). Na základe predchádzajúcej snímky sa vytvorí silueta objektu, podľa ktorej sa následne porovnávaním hľadá objekt v novej snímke. Model sa po nájdení objektu reinicializuje, vďaka čomu je schopný zniesť aj zmeny vo vzhľade či tvare objektu. Sú preto odolné voči slabej zmene osvetlenia a natáčaniu.

Sledovanie kontúr pracuje princípom iteratívneho vývoja kontúry okolo objektu. Na úspešné vytvorenie kontúry sa musí výskyt objektu prekrývať na snímkach idúcich po sebe.

2.4 Časticový filter

Pre túto sekciu som čerpal informácie z [3].

Istú formu časticového filtra prvýkrát použili na sledovanie objektu vo videu Isard a Blake a nazvali ho Condensation [5]. Je to skratka z *Con*ditional *Den*sity Propag*ation*. Časticové filtre (angl. *Particle filters*) sú tiež označované ako Sekvenčné Monte Carlo metódy (*SMC*). Medzi ich základné výhody patrí odolnosť proti pohybu kamery a tiež schopnosť sledovania negausouského dynamického systému.

K popisu dynamického systému, akým je aj sledovanie objektu, potrebujeme dva modely. Jeden model popisujúci vývoj stavu systému v závislosti na čase (rovnica (2.1), ďalej len systémový model) a druhý, ktorý popisuje merania vykonané na systéme (rovnica (2.2), ďalej len merací model). Na systémový aj merací model vplýva procesný šum v_{k-1} resp. šum spôsobený meraním n_k.

$$x_k = f_k(x_{k-1}, v_{k-1}) \tag{2.1}$$

$$y_k = h_k(x_k, n_k) \tag{2.2}$$

Ak budeme predpokladať vyjadrenie modelov v pravdepodobnostnej forme, môžeme na prácu s dynamickým systémom použiť rekurzívny Bayes filter. S využitím všetkých dostupných informácií o systéme je schopný vytvoriť aposteriórnu funkciu hustoty pravdepodobnosti (angl. *probability density function*, ďalej len *pdf*) na vyjadrenie nového stavu systému. Vďaka použitiu rekurzie je umožnené sekvenčné zapracovanie nových meraní do systému, preto nemusíme ukladať všetky dostupné dáta.

Výpočet nového stavu pomocou Bayes filtra zahŕňa dve fázy: predikciu a aktualizáciu (angl. *prediction* a *update*). V prvom kroku je pomocou Chapman–Kolmogorov rovnice (2.3) počítaná apriórna *pdf* $p(x_k|z_{1:k-1})$.

$$p(x_k|z_{1:k-1}) = \int p(x_k|x_{k-1})p(x_{k-1}|z_{1:k-1})\mathrm{d}x_{k-1} \tag{2.3}$$

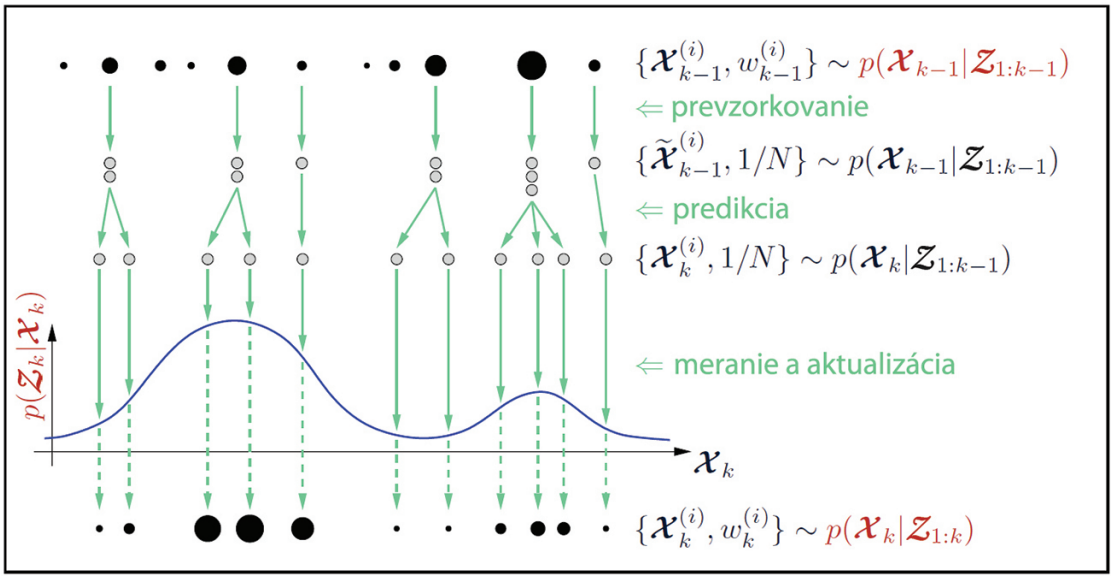
Použitím systémového modelu, viď rovnica (2.1), ktorý je v rovnici zastúpený ako $p(x_k|x_{k-1})$, je odhadnutá apriórna *pdf* na základe predchádzajúceho stavu systému $p(x_{k-1}|z_{1:k-1})$ v čase $k-1$. S príchodom meraní v čase k sa na základe Bayes pravidla (2.4) vypočíta aposteriórna *pdf*. Nasledujúca rovnica predstavuje aktualizačnú fázu.

$$p(x_k|z_{1:k}) = \frac{p(z_k|x_k)p(x_k|z_{1:k-1})}{p(z_k|z_{1:k-1})} \tag{2.4}$$

Pravdepodobnosť (angl. likelihood) $p(z_k|x_k)$ zastupuje merací model, viď rovnica (2.2) a predstavuje sadu nameraných hodnôt v čase k. Filter v tejto fáze modifikuje nepresnú apriórnu *pdf* pomocou meraní z_k. $p(z_k|z_{1:k-1})$ je normalizačná konštanta.

Rekurentné vzťahy medzi fázami vytvárajú optimálne Bayesove riešenie (angl. *optimal Bayes solution*). Analytickým postupom sa však rekurzívne šírenie aposteriórnej *pdf* nedá vyjadriť. Časticový filter rieši tento problém aproximovaním optimálneho Bayesovho riešenia. Aposteriórnu *pdf* vyjadríme pomocou náhodne generovaných vážených vzoriek (angl. *weighted samples*) inak nazývaných aj *častice*. S rastúcim počtom použitých častíc sa približujeme k skutočnému aposteriornému *pdf*.

Sadu častíc vyjadríme ako $\{X_{k-1}^{(i)}, w_{k-1}^{(i)}, i = 1, \ldots, N\}$, kde $w^{(i)}$ predstavuje váhu konkrétnej častice. Túto sadu potom používame pri výpočte v jednotlivých krokoch algoritmu. Na obr. 2.3 je zobrazený jeden priebeh časticového filtra popisujúci výpočet apriórnej a aposteriórnej *pdf*.



Obr. 2.3: Reprezentácia základného kroku Časticového filtru (prevzaté z [6])

Veľkosť kruhov reprezentujúcich častice v schéme, závisí od ich váhy. Cieľom prevzorkovania je zaručiť zachovanie častíc s najväčšou váhou. Algoritmus sa tým sústredí len na miesta s najvyššou pravdepodobnosťou výskytu objektu. V časti označenej ako predikcia je k jednotlivým časticiam pripočítaný šum. Nasleduje meranie a aktualizácia, kde pre každú časticu vypočítame váhu, čím získame aproximáciu aposteriórnej *pdf* $p(x_k|z_{1:k})$.

Na prevzorkovanie bol u väčšiny časticových filtrov používaný algoritmus SIS (angl. *Sequential Importance Sampling*). Objavuje sa však pri ňom tzv. degeneratívny problém, keď po niekoľkých iteráciách má množstvo častíc zanedbateľnú váhu. Preto vznikla nová metóda nazvaná SIR (angl. *Sampling Importance Resampling*), ktorá v sebe zahŕňa elimináciu častíc s najmenšími váhami čím v sade zostávajú len tie s relevantnou váhou.

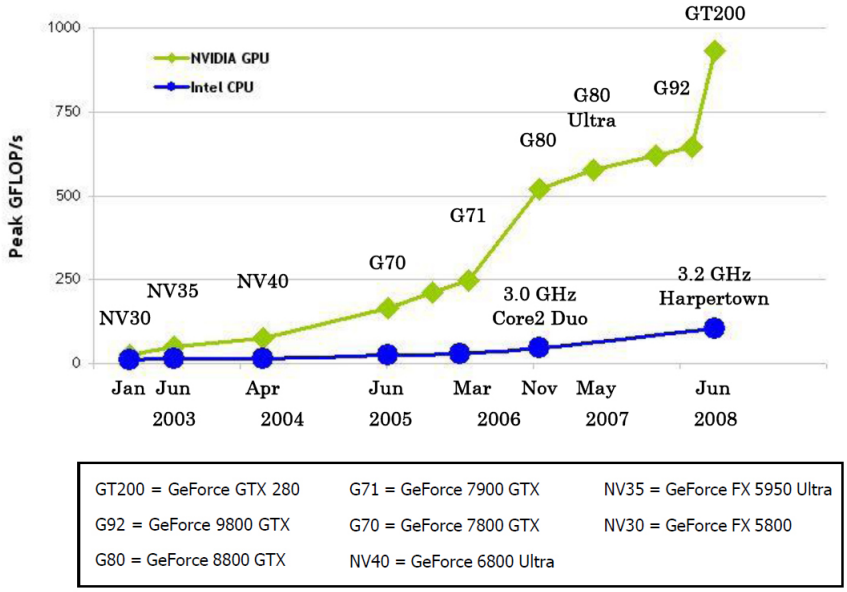
Kapitola 3

Architektúra CUDA

Táto kapitola vychádza a obrázky boli prevzaté z oficiálnej dokumentácie [2] a [1].

3.1 Vývoj grafických akcelerátorov

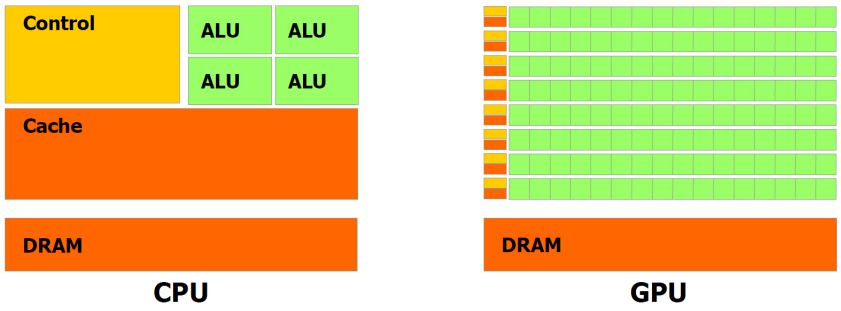
Grafické karty sa vyvíjajú pre zobrazovanie 3D grafiky v čo najväčších rozlíšeniach a aby sa tak dialo v reálnom čase. K tomuto cieľu sú vybavené viacjadrovými procesormi, ktoré pracujú paralelne s veľkým počtom vlákien. CPU zvyšovalo frekvencie, ale v súčasnosti sa vývoj týmto smerom zastavil a tiež pridávajú viac jadier. GPU však vo výpočtovej sile vedie nad CPU už niekoľko rokov, ako môžeme vidieť na obr. 3.1.



Obr. 3.1: Graf počtu floating point operácií za sekundu na CPU a GPU (prevzaté z [2])

V súčasnej dobe sa objavil dopyt po možnosti mohutnej paralelizácii a preto spoločnosť nVidia vytvorila univerzálnu paralelnú architektúru, ktorú pomenovala CUDA (Compute Unified Device Architecture).

GPU sú určené na intenzívne, vysoko paralelné výpočty a preto sú navrhované takým spôsobom, že viac tranzistorov je vyčlenených na spracovanie dát ako na ukladanie dát do vyrovnávacej pamäte (angl. caching) a kontrolu toku inštrukcií (angl. flow control). Na obr. 3.2 je zobrazený rozdiel medzi GPU a CPU z pohľadu vyčlenenia tranzistorov pre rôzne úlohy pri výpočte.

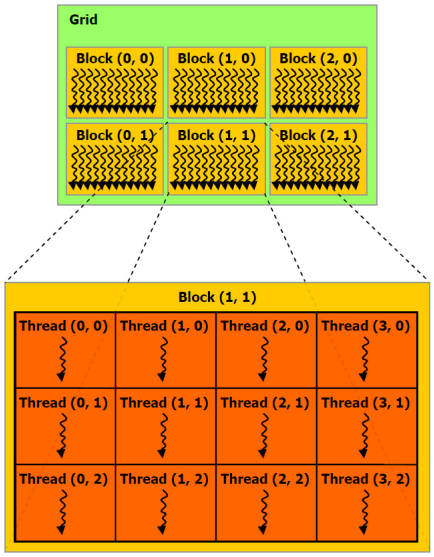


Obr. 3.2: Pomer využitia tranzistorov na CPU a GPU(prevzaté z [2])

3.2 Programový model

CUDA C ako rozšírenie jazyka C umožňuje programátorovi definovať funkciu, ktorá je nazvaná *kernel*. Táto funkcia sa na rozdiel od obyčajnej C funkcie spustí N-krát paralelne na N rôznych CUDA vláknach.

Každému vláknu vykonávajúcemu kernel je pridelené unikátne *thread ID*, ktoré je prístupné vo vnútri kernelu pomocou vstavanej premennej. Táto premenná je trojprvkový vektor, takže vlákna môžu byť indexované až trojrozmerným indexom. Skupina vlákien tvorí tzv. *blok vlákien* (angl. *thread block*, ďalej len *blok*). Z dôvodu hardwarového obmedzenia, môže v súčasných GPU jeden blok používať maximálne 1024 vlákien. CUDA však umožňuje spúšťať kernel na viacerých blokoch s rovnakými rozmermi. Bloky sú organizované do jednorozmernej alebo dvojrozmernej *mriežky blokov* (angl. *grid*, ďalej len *mriežka*) ako je ilustrované na obr. 3.3. Mriežky sú indexované vstavanou premennou *block ID*. Každý rozmer ich môže obsahovať až 65 536.



Obr. 3.3: Mriežka (Grid) a Blok vlákien (Block) (prevzaté z [2])

3.3 Pamäťový model

Z pohľadu Cuda je pamäť rozdelená na dve hlavné oblasti a to *host* a *device*. Ako *host* je označovaný systém, na ktorom je vykonávaný C program. *Device* je označenie pre GPU, ktorý vo vzťahu s *hostom* plní funkciu koprocesora a sú na ňom spúšťané kernely. CPU je zodpovedné za správu vlastných pamäťových priestorov v DRAM a do pamäte grafickej karty nemôže priamo pristupovať. Programu je preto umožnené túto pamäť ovládať pomocou volaní CUDA funkcií. Tieto volania zahŕňajú alokáciu a dealokáciu pamäte, ako aj prenos dát medzi *host* (systém) a *device* (grafická karta).

Pamäť na GPU môžeme rozdeliť na niekoľko druhov:

- *Globálna pamäť* (angl. *global memory*) je vlastne DRAM inštalovaná na grafickej karte. Použitá je kvôli jej veľkej kapacite. Zároveň je ale najpomalšia spomedzi pamätí a čítanie a ukladanie dát do nej by sa malo čo najviac minimalizovať. Ako jediná však môže prenášať dáta z a na systémovú DRAM. Každé vlákno môže pristúpiť na hocijakú jej položku.

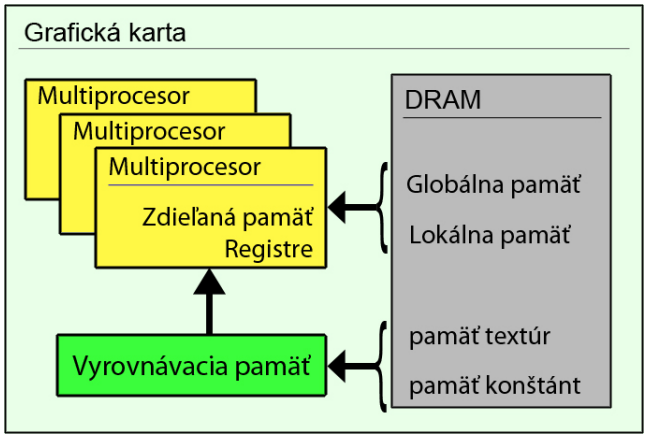
- *Pamäť textúr* (angl. *texture memory*) je definovaná nad globálnou pamäťou. Používa *cache*, ktorá je optimalizovaná pre využívanie dvojrozmernej adresácie prvkov. Umožňuje tiež filtrovanie dát pre textúry, ktoré majú návratovú hodnotu typu `float`. Je prístupná každému vláknu, ale výhradne pre čítanie.

- *Pamäť konštánt* (angl. *constant memory*) je tiež definovaná nad globálnou pamäťou. Používa *cache* optimalizovanú pre *broadcast* dát a je určená iba na čítanie. Pristúpiť k nej môže každé vlákno.

- *Lokálna pamäť* (angl. *local memory*) je mapovaná do globálnej pamäte. Pre každé vlákno je vyhradená samostatná oblasť pamäte a navzájom si do nich nemôžu pristupovať. Slúži na zápis a čítanie lokálnych premenných vlákien.

- *Zdieľaná pamäť* (angl. *shared memory*) slúži na uchovanie a zdieľanie dát v rámci jedného bloku. Pri správnom používaní môže dosahovať rýchlosť registrov. Jej kapacita na zariadeniach s *Compute Capability* (viď. 3.4.1) menšou ako 2.0 je 16kB inak 32kB.
- *Registre* (angl. *registers*) poskytujú najrýchlejšie čítanie a zápis dát. Každému vláknu je priradený konkrétny počet podľa potreby a prístupné sú len pre toto vlákno.

Pre lepšiu predstavu pamäťového priestoru je zahrnutý nasledujúci obrázok 3.4.



Obr. 3.4: Schéma pamäťových priestorov na CUDA

3.4 Hardwarová implementácia

Základ architektúry CUDA tvorí prispôsobiteľné pole multiprocesorov. Tie sú navrhnuté na súbežné vykonávanie inštrukcií stoviek vlákien. Pre zvládnutie takého množstva, využíva špeciálnu architektúru nazvanú *SIMT* (angl. *Single-Instruction, Multiple-Thread*), teda „*jedna inštrukcia, viacero vlákien*“.

Multiprocesor vytvára, spravuje, plánuje a spúšťa vlákna v skupinách po 32, nazvanej *warps*. Vlákna tvoriace *warp* sú spustené spoločne na rovnakej adrese programu, ale majú vlastný čítač adresy inštrukcie a vlastné registre, preto môžu vykonávať skoky a prevádzať inštrukcie nezávisle na sebe, avšak za cenu nutnosti serializácie. O rozdelenie blokov v multiprocesore do jednotlivých *warpov* sa stará jednotka nazvaná *warp scheduler*. Okrem toho prideľuje každému vláknu *thread ID* postupným indexovaním od nuly. *Warp* vykonáva naraz len jednu spoločnú inštrukciu, takže je ideálne, ak sa všetky vlákna zhodnú na vykonávaní rovnakej postupnosti inštrukcií. Pre každé odklonenie sa od spoločných inštrukcií musí *warp* vykonať tieto príkazy, aj keby s nimi pracovalo len jedno vlákno, čím sa strácajú výhody paralelizácie. Na rozdiel od vlákien, sú *warpy* vykonávané nezávisle na tom, či nasledujú spoločnú alebo rozdielnu postupnosť inštrukcií.

Multiprocesor umožňuje spustiť viacero blokov a následne medzi nimi prepínať. Tým sa dosiahne efektívneho využitia času, počas ktorého jeden blok čaká na načítanie dát, vykonávaním inštrukcií iného bloku. Každý *warp* spracovávaný na multiprocesore, má po celú dobu výpočtu uložený svoj kontext na čipe. Prepínanie z jedného kontextu na druhý, preto prebehne bez réžie navyše. *Warp scheduler* pri vystavovaní inštrukcie zároveň vyberá *warp*, ktorý má vlákna pripravené na spustenie.

3.4.1 Compute Capability

Compute capability je číslo skladajúce sa z dvoch revíznych čísiel a označuje vývojový stupeň zariadenia, od ktorého sa odvíjajú vlastnosti a prípadne aj nové funkcie pribúdajúce s novšími verziami. Prvé číslo je rovnaké pre všetky zariadenia s rovnakou architektúrou jadra. Pre najnovšiu *Fermi* architektúru je to číslo 2. Druhé číslo sa zvyšuje s postupnými vylepšeniami architektúry jadra a môže znamenať aj pridanie novej vlastnosti. Napríklad zariadenia s *compute capability* 1.2 získali oproti predchádzajúcim verziám funkcie pre atomický prístup do zdieľanej pamäte.

3.5 Optimalizácie

3.5.1 Riadenie toku

Každá z inštrukcií na riadenie toku (`if`, `switch`, `do`, `for`, `while`) môže výrazne ovplyvniť množstvo vykonaných príkazov v jednom *warpe* tým, že spôsobí rozdelenie vlákien, ktoré potom sledujú rôzne postupnosti inštrukcií. Keď sa tak stane, vykonávanie kódu musí byť serializované pre každú odlišujúcu sa skupinu vlákien, zvyšujúc tak množstvo inštrukcií prevedených jedným *warpom*. Aby sa zaistil čo najvyšší výkon, podmienky týchto inštrukcií by sa mali vzťahovať k celým *warpom* a nie k jednotlivým vláknam. V rámci jednotlivých *warpov* sa totiž postupnosť vykonávaných inštrukcií môže líšiť bez vplyvu na výkon.

Inštrukcie prístupu do pamäte zahŕňajú všetky inštrukcie, ktoré čítajú alebo zapisujú do zdieľanej, lokálnej alebo globálnej pamäte. Čas potrebný na vystavenie pamäťovej inštrukcie je 8 hodinových taktov. Pri prístupe do globálnej alebo lokálnej pamäte sa k tomu pripočítava od 400 do 600 hodinových cyklov oneskorenia. Veľká časť tohto času sa dá skryť prepnutím na vykonávanie ďalšieho bloku spustenom v mikroprocesore. Avšak najlepšie je vyhnúť sa prístupu do globálnej pamäte, pokiaľ je to možné.

3.5.2 Globálna pamäť

Globálna pamäť sa nachádza mimo čipu mikroprocesoru, preto je jej čítanie a zápis pomalý. Jedna z najdôležitejších optimalizácií v CUDA architektúre je zarovnaný prístup do globálnej pamäte. Tá sa skladá zo zarovnaných segmentov po 32, 64 a 128 Bajtov. Tým že sa zaistí, aby jeden *half-warp* (prvá alebo druhá polovica *warpu*) pristúpil k zarovnanému segmentu, dôjde iba k jednej pamäťovej transakcii. Zariadenia s *compute capability* väčšou ako 1.2 majú voľnejšie podmienky pre zarovnaný prístup. Stačí pokiaľ sú požadované dáta obsiahnuté vo väčšom zarovnanom bloku pamäte a nemusia byť umiestnené na jeho začiatku.

3.5.3 Pamäť textúr

Pamäť textúr používa *cache* a je určená len na čítanie. Je optimalizovaná na prístup k dátam pomocou dvojrozmerného systému, takže vlákna z jedného *warpu*, ktoré čítajú z adries blízko seba, dosiahnu najlepší výkon. Čítanie z textúry potrebuje jeden prístup do globálnej pamäte, iba ak sa potrebné dáta nenachádzajú v *cache*. V opačnom prípade sa použije vyrovnávacia pamäť a nároky na priepustnosť globálnej pamäte sa znížia.

3.5.4 Pamäť konštánt

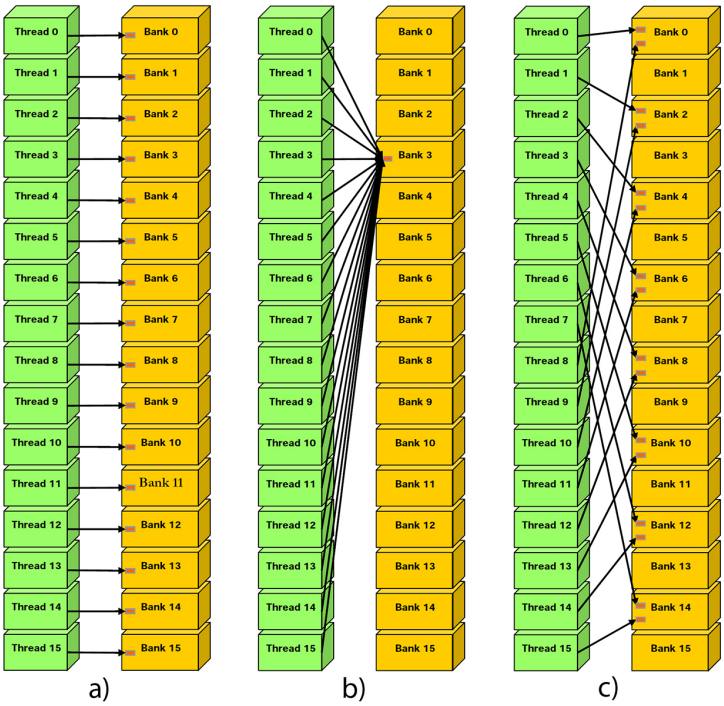
Na zariadení sa nachádza 64kB pamäte konštánt. Obsahuje *cache* a čítanie z nej zaberie čas jedného prístupu do globálnej pamäte v prípade, že sa požadovaná položka nenachádza

v *cache*. Ak všetky vlákna *warpu* pristupujú k rovnakej položke, rýchlosť čítania je rovnaká ako rýchlosť registrov. V prípade nezhody bude načítavanie serializované.

3.5.5 Zdieľaná pamäť

Zdieľaná pamäť je vďaka umiestneniu na čipe spoločne s multiprocesormi oveľa rýchlejšia, než lokálna alebo globálna pamäť. Pri jej optimálnom využívaní dosahuje viac ako stokrát menšiu latenciu.

Základný stavebný prvok zdieľanej pamäte sa nazýva *bank* a tvorí najmenší adresovateľný blok. Má veľkosť 32 bitov a po sebe idúce slová sú odpovedajúco usporiadané do *banks* idúcich po sebe. Žiadosť o prístup do zdieľanej pamäte od jedného *warpu* je rozdelená na dve, teda pre každý *half-warp*. Ak chceme dosiahnuť najrýchlejšiu odozvu, musíme zaistiť, aby každé vlákno *half-warpu*, pristupovalo do iného *banku* alebo všetky čítali z toho istého. Pri druhej z možností sa využije *broadcast*. V opačnom prípade by sa prístupy museli serializovať. Na obr. 3.5 sú znázornené prípady čítania zo zdieľanej pamäte.



Obr. 3.5: Čítanie zd. pamäte: (a) zarovnaný prístup - žiadny konflikt (b) využitie *broadcast* (c) konflikt niektorých vlákien (prevzaté z [2])

Zapisovanie pomocou neatomickej funkcie viacerými vláknami do rovnakého *banku* spôsobí, že iba jednému vláknu v rámci *half-warp* sa podarí pristúpiť do pamäte. Atomické funkcie pre prístup do zdieľanej pamäte sú však dostupné až v zariadeniach s *compute capability* 1.2 a viac. V ostatných prípadoch treba prístup explicitne serializovať.

3.5.6 Prenos dát medzi host a device

Prenos dát zo systémovej DRAM do grafickej karty a naopak je časovo náročná operácia. Zariadenia musia čakať, kým budú mať k dispozícii zdroje potrebné k ďalšiemu výpočtu. Aplikácie by sa preto mali snažiť obmedziť výmenu dát medzi *host* a *device*. Jedným spôsobom, ako to dosiahnuť, je preniesť čo najväčšie množstvo výpočtov na GPU.

Spájanie množstva malých prenosov do jedného veľkého spôsobí zníženie počtu režijných operácií potrebných k inicializácii prenosu.

3.5.7 Prepojenie s OpenGL

CUDA dokáže namapovať do svojho pamäťového priestoru zdroje z OpenGL. Medzi ne patrí OpenGL buffer, textúra a renderbuffer objekt. Čítanie a zápis následne prebieha priamo z kernelu. Výsledky výpočtov preto nemusia byť kvôli zobrazeniu prenášané do systémovej DRAM, ale môžeme ich vykresľovať priamo z GPU pamäte.

3.5.8 Konfigurácia kernelu a zaťaženie mikroprocesoru

Základom pre dobrý výkon aplikácie je zaťaženie multiprocesorov na maximum. V CUDA sú inštrukcie pre vlákna vykonávané sekvenčne, preto jediný spôsob ako skryť latencie prístupov do pamäte, je mať niekoľko aktívnych *warpov*, medzi ktorými bude multiprocesor prepínať.

Množstvo spustených blokov na multiprocesore je ovplyvnené možnosťami hardwaru zariadenia a náročnosti jedného bloku na jeho zdroje. Prvým faktorom vplývajúcim na počet spustených blokov je dostupnosť registrov použitých pre ukladanie lokálnych premenných. Ich počet je obmedzený a všetky vlákna nachádzajúce sa na multiprocesore ich zdieľajú. Každý blok si pri spustení alokuje určitý počet registrov, ktoré využije a používa ich až do svojho ukončenia. Vyčlenenie veľkého množstva registrov na jeden blok tak spôsobí zníženie počtu aktívnych blokov na multiprocesor. Ten istý prípad nastáva aj pri rozdeľovaní zdieľanej pamäte. Jej nedostatok zamedzí spusteniu ďalšieho bloku až do doby, kým sa neuvoľní požadované množstvo pamäte.

Kapitola 4

Návrh

Program bude písaný v programovacom jazyku C++, C a C pre CUDA. Vyvíjaný bude v prostredí operačného systému Windows 7 za použitia Microsoft Visual Studia 2008. Rýchlostné testy budú vykonávané na dvoch zostavách. Prvá z nich je notebook, na ktorom je program vyvíjaný a prekladaný. Obsahuje grafickú kartu nVidia GeForce 9600M GT (*Compute Capability* 1.1) a Intel Core2Duo taktované na 2,5 GHz. Druhá zostava obsahuje GeForce GTX 285 (*Compute Capability* 1.3) a Intel Core2Duo taktované na 2,66 GHz.

Pri vytváraní paralelných aplikácii je správny návrh veľmi dôležitý, preto budú popisy jednotlivých algoritmov v tejto kapitole podrobnejšie.

4.1 Výber a analýza metódy sledovania objektu

Ako vhodný algoritmus sledovania objektu vo videu k implementácii na CUDA bol vybraný Časticový filter (viď. sekciu 2.4 na str. 6).

Pred začatím práce na bakalárskej práci neboli na internete dostupné implementácie časticového filtra na architektúre CUDA. Keďže sa tento algoritmus pohybuje na hranici výpočtu v reálnom čase, jeho paralelizáciou by sa dalo dosiahnuť užitočného zrýchlenia.

Algoritmus implementujeme v dvoch rôznych verziách s odlišnými hodnotiacimi funkciami.

Časticový filter sa skladá z troch logicky oddelených celkov. Prvým je vzorkovanie, nasleduje počítanie váh a nakoniec výber častice s najlepšou zhodou so vzorkou obrazu (ďalej len vzorkou).

Pri vzorkovaní sa počíta s malým množstvom častíc, a preto by sa paralelizáciou nedosiahlo relevantného zrýchlenia. Navyše tu prebieha generovanie náhodných čísiel, ktorého implementácia na paralelnom stroji nedosahuje uspokojivé výsledky, alebo je príliš zložitá.

Ideálny algoritmus k paralelizácii prevádza rovnaké inštrukcie nad množstvom rôznych dát. Výpočet váhy má preto veľký potenciál dosiahnuť markantného zrýchlenia jeho implementovaním na grafickej karte. Okrem iného je to aj výpočtovo najnáročnejšia časť a akákoľvek zmena v rýchlosti spracovania sa výrazne prejaví.

Vzhľadom na relatívne malý počet častíc, nebude výber tej s najlepším ohodnotením potreba implementovať do CUDA, pretože by bolo zrýchlenie zanedbateľné.

4.2 Ovládanie programu

Program bude umožňovať zadanie počiatočnej konfigurácie pomocou argumentov príkazového riadku. Táto konfigurácia by mala zahŕňať zadanie názvu video súboru, ktorý sa má načítať a taktiež názov súboru pre uloženie výsledkov sledovania. Pri použití video súboru bude možné zadať pozíciu a rozmery oblasti, v ktorej sa nachádza objekt na sledovanie a tiež kernel, ktorý bude použitý pri sledovaní.

Tento spôsob nastavenia bude umožňovať aj pri snímaní videa z web kamery. Zapne sa v prípade ak sa neuvedie názov video súboru. Pre použiteľnosť tejto metódy bude potrebné pridať možnosť výberu objektu na sledovanie priamo pri zobrazovaní snímaného obrazu. Kernel bude tiež možné zmeniť pomocou priradených kláves pred samotným výberom objektu.

4.3 Práca s videom

Pre načítavanie videa použijeme knižnicu OpenCV [4], ktorá obsahuje funkcie umožňujúce prístup k jednotlivým snímkam z video súboru vo formáte `avi` a web kamery. Použitím funkcií pre kreslenie do snímky bude zobrazovaná nová poloha sledovaného objektu a následne ukladaná do samostatného video súboru.

Obraz bude snímaný v troch farebných kanáloch, každý o veľkosti 8 bitov. Grafická karta dokáže načítať najmenšiu položku o veľkosti 4 Bajty, preto pre väčšiu efektivitu pri načítaní dát, sa každý pixel rozšíri o 8 bitov.

Na zobrazenie priebehu sledovania objektu bude použitá knižnica OpenGL [7]. Odpadne tým povinnosť prenosu dát na vykreslenie späť do RAM pamäte. Využijeme *Pixel Buffer Object* (ďalej len *PBO*), ktorý slúži na uloženie a prenos dát vo forme pixelov. Hlavná výhoda *PBO* je v rýchlom dátovom prenose do a z grafickej karty cez *DMA* (*Direct Memory Access*) bez použitia CPU a tiež skutočnosť, že prenos je asynchrónny.

4.4 Kostra programu

Z prvej snímky, v ktorej bude vyznačený objekt na sledovanie, sa inicializačným kernelom získa vzorka objektu. Pokiaľ prebieha sledovanie objektu, pomocou ďalšieho CUDA kernelu sa vypočíta váha jednotlivých častíc. Procesor vyberie časticu s váhou, ktorá odpovedá najbližšej zhode so vzorkou a do snímky vykreslíme novú pozíciu objektu. V okolí posledného výskytu objektu vygenerujeme častice a aplikujeme na ne šum. Ich pozície sa na grafickú kartu prenesú pred spustením ďalšieho kernelu. Algoritmus ukončíme načítaním ďalšej snímky videa. Kostru programu popisuje pseudokód 1.

Algorithm 1 Pseudokód kostry programu

```
1: while tracking do
2:   if objectSelected then
3:     createSamples_CUDA();
4:   else
5:     evaluateWeights_CUDA();
6:     chooseBestParticle();
7:   end if
8:   showObjectPosition();
9:   resampleParticles();
10:  loadNewImage();
11: end while
```

4.5 Určovanie váhy počítaním rozdielu pixelov častice a vzorky

Táto metóda bude ďalej v práci označovaná ako *prvá metóda*.

4.5.1 Základný návrh

Prvá verzia programu bude najzákladnejšia implementácia s požadovanou funkcionalitou, ktorá sa bude v následných krokoch optimalizovať.

Pre výpočet váhy sa budú volať 2 kernely. U prvého bude jeden blok využívať 8 vlákien, pričom jedno vlákno vypočíta rozdiel jednotlivých farebných kanálov odpovedajúcich pixelov častice a vzorky. Tieto rozdiely následne umocní, aby sa mohli sčítať a výsledok uloží do pamäte. Druhý kernel načíta medzivýsledky a paralelne ich sčíta. V tomto prípade každý blok kernelu bude počítať s 512 vláknami a dokáže sčítať 1024 čísiel. Vo väčšine prípadov sa na sčítanie medzivýsledkov jednej častice bude musieť použiť viacero blokov a teda znova dostaneme len časti výsledku. Tieto ale budú vo veľmi malom množstve a neoplatí sa spočítavať ich na grafickej karte pomocou ďalšieho kernelu. Dáta sa predajú procesoru, ktorý vyberie časticu s najlepšou zhodou so vzorkou.

4.5.2 Optimalizácia konfigurácie kernelu

Aj keď multiprocesor umožňuje prepínanie medzi viacerými blokmi, blok s ôsmimi vláknami nevyužíva možnosti paralelizácie veľmi efektívne. Použitie 512 vlákien na jeden blok umožní spracovať 64-krát viac pixelov, čím sa zníži počet vykonávaných blokov a teda aj množstvo réžie potrebnej pre ich spúšťanie.

4.5.3 Použitie jedného kernelu a zdieľanej pamäte

Spojenie predchádzajúcich dvoch kernelov do jedného umožní zrýchlenie z dôvodu absencie réžie druhého kernelu. Okrem toho sa výsledky nemusia ukladať do globálnej pamäte, ktorá má veľké oneskorenie odpovede. Použitím zdieľanej pamäte sa podarí obmedziť použitie globálnej pamäte len na načítanie obrazových dát a následné uloženie výsledkov.

4.5.4 Hľadanie najefektívnejšej konfigurácie

Keďže testy budú prebiehať na dvoch výkonnostne veľmi rozdielnych zostavách, rôzne konfigurácie kernelu môžu mať rôzne doby vykonávania algoritmu. Základný rozdiel je v počte

vlákien, ktoré dokáže grafická karta spustiť v jednom mikroprocesore. U GeForce 9600M GT je to 768 a u GeForce GTX 285 je to 1024 vlákien. S prihliadnutím k tejto skutočnosti sa javí ako najideálnejšie vytvoriť blok s počtom vlákien, ktoré sú násobkom čísla 128. Konkrétne 256, 384 a 512 vlákien na jeden blok.

4.5.5 Použitie pamäte textúr

Častice filtra sa prekrývajú, preto bude vhodné otestovať prínos pamäte pre textúry, ktorá využíva rýchlu vyrovnávaciu pamäť. Častíc je však veľa a pri 64kB vyrovnávacej pamäte je dosť pravdepodobné, že sa bude častejšie prepisovať ako triafať do už načítanej oblasti dát.

4.6 Určovanie váhy počítaním rozdielu histogramov

Každá farebná zložka môže nadobudnúť 256 úrovní intenzity farby. Z dôvodu úspory pamäte sa použije 128 prvkový histogram s tromi farebnými kanálmi. Po skúsenostiach s predchádzajúcim algoritmom bude hneď v prvej verzii algoritmu priamo zahrnuté využívanie zdieľanej pamäte a pamäte textúr pre načítavanie obrazových dát.

Počítanie histogramu na paralelnom stroji prináša niekoľko problémov. Pokiaľ viacero vlákien, ktoré pracujú paralelne, načítajú rovnakú farbu, pokúsia sa naraz zvýšiť hodnotu položky odpovedajúcej tejto farbe v histograme. V prípade, že grafická karta nepodporuje atomické prístupy do zdieľanej pamäte (zariadenie s *compute capability* menšou ako 1.2), k položke v pamäti pristúpi len jedno vlákno. Z toho dôvodu musí byť zaistená serializácia zápisu, ktorá však výrazne spomalí výpočet.

Ďalším problémom je nedostatok zdieľanej pamäte a registrov. Pokiaľ by sme chceli obísť serializáciu algoritmu spôsobenú súčasným zápisom viacerých vlákien na jedno miesto, museli by sme pre každé vlákno alokovať vlastný histogram. Tie by sa nakoniec spočítali. Na tento postup však nie je dostatok zdieľanej pamäte. Musí sa nájsť kompromis medzi rozsahom serializácie výpočtu a množstvom využitej pamäte, od ktorého závisí aj počet blokov spustených na mikroprocesore.

Na základe predchádzajúcej analýzy problému budú výpočet prevádzať dva kernely.

Prvý kernel

Samostatný blok bude pracovať s 256 vláknami rozdelenými do skupín po 32. Každá skupina bude mať k dispozícii vlastný histogram. Maximálne môže nastať 32 súbežných pokusov o zápis, ktoré sa budú musieť serializovať.

- Pre zariadenie umožňujúce atomický prístup do zdieľanej pamäte bude jedna položka histogramu uložená na 8 bitoch, čo znamená, že jeden trojkanálový histogram zaberie 384 Bajtov. Jeden blok teda potrebuje 3072 Bajtov na uloženie všetkých čiastkových histogramov. Na zariadení s *compute capability* väčšou ako 1.1, môže jeden mikroprocesor použiť 1024 vlákien a k dispozícii má 16kB zdieľanej pamäte. Preto bude schopný súčasne spustiť až 4 bloky a medzi nimi prepínať (Použije sa 1024/1024 vlákien a 12/16 kB zdieľanej pamäte).

- Pre ostatné zariadenia sa použije iný postup. Vlákna budú v cykle spolu s dátami zapisovať do zdieľanej pamäte aj svoje poradové číslo. Následne otestujú, či sa toto číslo zhoduje s ich poradovým číslom. Ak áno, znamená to, že vláknu sa zápis podaril a môže opustiť cyklus.

Prvý kernel vytvorí čiastkový histogram pre 256 pixelov častice a uloží ho do globálnej pamäte, pričom jedna položka bude uložená na dvoch Bajtoch kvôli zníženiu zaťaženia dátovej priepustnosti globálnej pamäte. Pre každú časticu vznikne niekoľko čiastkových histogramov.

Druhý kernel

Jeden blok druhého kernelu spočíta histogramy jednej farebnej zložky pre celú časticu. Pracovať bude s 256 vláknami rozdelenými do 4 skupín po 64 vlákien. Všetky skupiny budú postupne načítať odpovedajúce histogramy a nakoniec sa sčítajú navzájom, čím vznikne jeden výsledný histogram pre jednu farebnú zložku častice. Následne sa vypočíta rozdiel oproti vzorke a uloží sa do globálnej pamäte.

4.7 CPU verzie váhových funkcií

Pre objektívne porovnanie výkonu algoritmov na CUDA s CPU verziami budú implementované dve funkcie na výpočet váhy na procesore. Komplexnosťou budú odpovedať implementáciam na grafickej karte.

Kapitola 5

Implementácia a testovanie

Pri popise niektorých funkcií sa na mieste predávaných parametrov nachádzajú tri bodky. Tie nepredstavujú súčasť programovacieho jazyka, ale nahrádzajú popisy skutočných parametrov, ktoré nie sú potrebné pre pochopenie podstaty funkcie a zabrali by veľa miesta.

Pri implementovaní jednotlivých optimalizácií sú spomenuté aj rýchlostné výsledky popisovaných verzií algoritmu. Tie sú následne zhrnuté v sekcii 5.6.

5.1 Ovládanie programu

V inicializačnej časti programu bola použitá funkcia `cutGetCmdLineArgumenti(...)` na získanie argumentu z príkazového riadku typu `int` a `cutGetCmdLineArgumentstr(...)` pre argument vo forme textového reťazca. Ak bola definovaná pozícia objektu, sledovanie sa spustí automaticky, inak sa čaká na vyznačenie objektu pomocou myši.

Keďže na zobrazovanie videa je použité OpenGL, na vykreslenie ohraničenia objektu pri jeho vymedzovaní bola použitá funkcia `glVertex2f(...)`. Jej osemnásobným použitím je poskladaný štvorhran, ktorý sa obnovuje pri pohybe myši. Po uvoľnení tlačidla automaticky začína sledovanie objektu.

Na zobrazenie aktuálne detekovanej pozície, je použitý jednoduchý `color_kernel(...)`, ktorý zvýrazní jednu farebnú zložku obrazu. Detailný popis ovládania sa nachádza v dodatku A.

5.2 Práca s videom

Na načítanie videa z video súboru použijeme funkciu `cvCreateFileCapture(const char*)` knižnice OpenCV. Ako parameter sa predáva názov súboru ktorý musí byť vo formáte avi.

K videu z web kamery pristúpime cez funkciu `cvCreateCameraCapture(int)`, ktorej sa predáva index kamery zapojenej v systéme. Indexujú sa od nuly, preto ako parameter zadáme číslo 0.

Od oboch funkcií dostaneme ukazovateľ na štruktúru `CvCapture`, z ktorej za použitia `cvQueryFrame(...)` získame nasledujúci ukazovateľ na štruktúru IplImage na nasledujúci snímok videosekvencie.

Na ukladanie jednotlivých snímok videa do nového videosúboru si vytvoríme objekt CvVideoWriter a použijeme funkciu `cvWriteFrame(...)`. Pred samotným uložením musíme do obrázku zakresliť pozíciu detekovaného objektu pomocou funkcie `cvRectangle(...)`.

5.3 Kostra programu

Program využíva kostru založenú na knižnici OpenGL. Obsahuje *callback* funkcie pre myš, klávesnicu, pohyb myši a zobrazovaciu funkciu `displayFunc(void)`, ktorá bude bližšie popísaná.

Pokiaľ bola práve vyznačená oblasť výskytu objektu, program musí vymazať vzorku objektu sledovaného v predchádzajúcom kroku z DRAM a z grafickej karty. Následne sa nastaví nová pozícia a rozmery objektu. Pred samotným volaním funkcií, ktoré zaistia vytvorenie vzorky objektu, sa musí pomocou `cudaGLRegisterBufferObject(...)` zaregistrovať *PBO* objekt (viď 4.3) aby sa mohol používať v CUDA a následne ho namapovať cez `cudaGLMapBufferObject(...)` do pamäťového priestoru CUDA. Pre zaistenie možnosti prepínať medzi metódami sledovania sa vytvoria vzorky objektu zavolaním funkcií pre obidva spôsoby: `launch_init_kernel(...)` pre sledovanie pomocou počítania rozdielu medzi obrazmi a `init_hist_kernel(...)` pre sledovanie použitím histogramov. Po vytvorení vzoriek je nutné zrušiť mapovanie a objekt odregistrovať pomocou `cudaGLUnmapBufferObject(...)` a `cudaGLUnregisterBufferObject(...)`.

Ak sú vzorky objektu k dispozícii a prebieha sledovanie objektu, ako prvá sa zavolá funkcia `translateParticles()`, ktorá vygeneruje náhodné rozloženie pre častice. Následne je vybraná jedna z troch funkcií `launch_hist_kernel(...)`, `launch_kernel(...)` a `launch_one_kernel(...)`, ktorá spúšťa druh kernelu vopred vybraný užívateľom. Rovnako ako bolo popísané v predchádzajúcom odseku, treba registrovať *PBO* a namapovať ho do pamäťového priestoru CUDA.

Po detekcii objektu nasleduje vykreslenie výsledku do okna použitím OpenGL funkcií. Ak je zapnuté ukladanie videa, tak sa uloží aktuálna snímka, do ktorej sa najskôr pomocou `cvRectangle(...)` zakreslí nová pozícia objektu. Pokiaľ prebieha sledovanie, alebo je video snímané z web kamery, načíta sa nová snímka. Nakoniec sa do grafickej karty cez funkciu `glBufferDataARB(...)` skopíruje aktuálna snímka videosekvencie.

5.4 Určovanie váhy počítaním rozdielu častice a vzorky

Súčasne s popisom optimalizácie implementácie budú uvádzané časy výpočtu algoritmu. Budeme tak postupne diskutovať úspešnosť vykonaných zmien. Využijeme výsledky testov 1 a 2 z tabuľky 5.3 na str. 26. Parametre testov viď podkapitola 5.6.

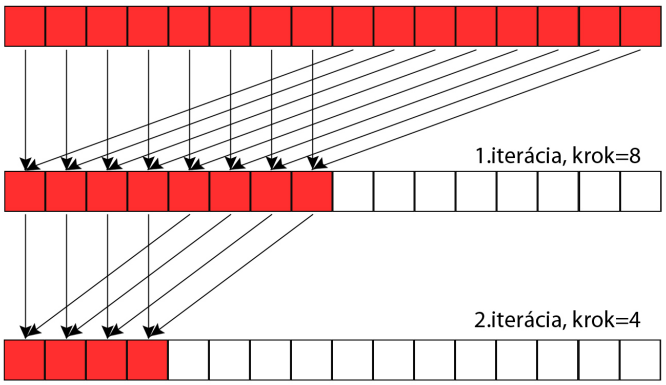
5.4.1 Prvá implementácia

Na spúšťanie kernelov, alokovanie pamäte a prenos dát medzi systémovou DRAM a pamäťou grafickej karty, bola implementovaná funkcia `launch_kernel(...)`. Najskôr skopíruje pole s pozíciami častíc na GPU cez `cudaMemcpy()`, alokuje miesto pre medzivýsledky pomocou funkcie `cudaMalloc()` a za použitia príkazu `switch()` zaistí správne spustenie toho kernelu, ktorý bol vybraný užívateľom. Nastaví sa konfigurácia kernelu a pomocou funkcií Event modulu od CUDA sa zmeria čas výpočtu.

Prvý implementovaný kernel sa volá `kernelT8` a v jednom bloku využíva 8 vlákien. Každé vlákno si na začiatku vypočíta pozíciu odpovedajúcich pixelov obrazu vo vzorke objektu (`frame_pos`) a v aktuálnej snímke (`frame_x`). Kernel bude väčšinou spúšťaný s viac vláknami ako má objekt pixelov (`max_size`), preto musíme použiť podmienku `if (frame_pos < max_size)`, v ktorej sa po jej splnení odčítajú všetky farebné zložky, umocnia a sčítajú.

Takto získame celkový rozdiel odpovedajúcich pixelov, ktorý uložíme do globálnej pamäte pre medzivýsledky.

Pre druhý kernel, nazvaný `add_kernel` sa alokuje ďalšie miesto, na ktoré sa zapíšu výsledky sčítania výstupov z predchádzajúceho kernelu. Do premennej `result_x` si každé vlákno uloží index na číslo s ktorým bude pracovať. Keďže posledný blok bude pracovať s vopred neznámym počtom čísiel, musí sa pomocou podmienky ošetriť, aby nepočítal s prvkami navyše. Potom môžeme použiť sčítanie s krokom mocniny dvojky, ktorý sa postupne znižuje o polovicu (viď obr. 5.1). Zložitosť tejto operácie je logaritmická.



Obr. 5.1: Zobrazenie 2 iterácií paralelného sčítania

Výstup z druhého kernelu skopírujeme do pamäte RAM pomocou `cudaMemcpy(...)`, kde procesor vyberie časticu s najmenším rozdielom oproti vzorke. Následne sa zavolá `color_kernel(...)`, ktorý pozmení zložky farby vybranej častice pre jej zvýraznenie.

Prvá implementácia neberie ohľad na optimalizácie a preto aj časy kernelu nie sú uspokojivé. Na notebooku zabral výpočet obidvoch kernelov v priemere 54,065ms a na desktope 4,980ms. Verzia na procesore pre porovnanie dosiahla čas 70,216 ms.

5.4.2 Optimalizácia konfigurácie kernelu

Pre otestovanie tejto optimalizácie bol vytvorený `KERNEL_T512`. Spúšťa sa rovnakým spôsobom ako predchádzajúci kernel, iba konfigurácia a výpočet pozície vlákna boli upravené pre využívanie 512 vlákien na blok. Keďže sa pracuje s oveľa viac vláknami, rýchlosť na notebooku dosiahla 40,6 ms a desktope 2,033 ms.

5.4.3 Použitie jedného kernelu a zdieľanej pamäte

Pri prechode na využívanie jedného kernelu sa musela zmeniť aj spúšťacia funkcia. Implementovaná bola funkcia `launch_one_kernel(...)`, v ktorej bolo vypustené alokovanie miesta pre druhý kernel a jeho spúšťanie. Ostatné prvky boli zachované.

Vytvorený bol `kernel_T512_V2`, ktorý vychádzal z predchádzajúcej verzie a teda prijíma rovnaké parametre. Na začiatku je alokácia zdieľanej pamäte pre dáta aktuálnej snímky `dataImg`, dáta vzorky objektu `dataRef` a výsledky výpočtu `results`. Vypočítajú sa indexy do pamäte a pred samotným počítaním, si každé vlákno načíta hodnoty pixelu do zdieľanej pamäte, s ktorou bude pri výpočte pracovať. Za výpočtom je synchronizácia vlákien pomocou príkazu `__syncthreads()`, ktorý zaistí aby vlákna, ktoré neboli použité na výpočet

v poslednom bloku, nespôsobili čiastočnú serializáciu výpočtu. Výsledky sa ukladajú do poľa `results` a nakoniec do globálnej pamäte. V tejto fáze dostávame rovnaký výstup, ako po druhom kerneli z predchádzajúcej implementácie.

Vďaka týmto zmenám sa prístup do globálnej pamäte obmedzil na minimum a kernel to zrýchlilo na 22,780 ms na notebooku a 1,987 ms na desktope.

5.4.4 Hľadanie najefektívnejšej konfigurácie

Pre tento test boli vytvorené kernely `kernel_T384_V1` a `kernel_T256_V1` s 384 a 256 vláknami na jeden blok. Vznikli jednoduchými úpravami konfigurácie kernelu `kernel_T512_V2`. Výsledné časy môžeme vidieť v tabuľke 5.1.

	kernel_T512_V2	kernel_T384_V1	kernel_T256_V1
GeForce 9600M GT	22,780 ms	28,863 ms	**21,751** ms
GeForce GTX 285	1,987 ms	2,003 ms	**1,800** ms

Tabuľka 5.1: Časy rôznych konfigurácií

Ako najrýchlejší sa ukázal byť kernel s 256 vláknami na blok. Umožňuje multiprocesoru prepínať medzi viacerými aktívnymi blokmi aby zakryl latenciu prístupu do globálnej pamäte. Naopak použitie 384 vlákien je na notebooku oproti počtu 512 viditeľne pomalšie.

5.4.5 Použitie pamäte textúr

Po vytvorení referencie na textúru, ktorú budeme používať (`texture<...> texRef`), ju pomocou funkcie `cudaBindTexture(...)` pripojíme k snímke videa uloženej na grafickej karte pred samotným volaním kernelu. V ňom je na prístup k jednotlivým položkám použitá funkcia `tex1Dfetch(texture<...> texRef, int position)`. Nakoniec musíme zrušiť previazanie s textúrou k čomu slúži `cudaUnbindTexture(texture<...> texRef)`.

Ukázalo sa, že k využitiu dát z vyrovnávacej pamäte nedochádza tak často, pretože zrýchlenie nebolo výrazné. Na otestovanie tejto optimalizácie bol prispôsobený kernel s 512 a 256 vláknami na blok. Pri prvom spomínanom bolo zrýchlenie algoritmu u notebooku na 20,777 ms a desktope na 1,951 ms. Kernel s 256 vláknami sa na desktope zrýchlil na 1,746 ms, čo predstavuje najrýchlejší čas tejto zostavy. Na notebooku však došlo k spomaleniu na 23,673 ms. To mohlo byť spôsobené malým množstvom opätovného používania dát z vyrovnávacej pamäte, pričom čas spotrebovaný na réžiu načítavania a prepisovania dát spomalil výpočet.

5.5 Implementácia histogramovej hodnotiacej funkcie

Na spúšťanie výpočtu váhy častice pomocou porovnania histogramov je určená funkcia `launch_hist_kernel(...)`. Po spustení sa pomocou funkcie `cudaMemcpy(...)` skopírujú pozície jednotlivých častíc na grafickú kartu. Alokujeme miesto pre čiastkové histogramy a výsledný rozdiel histogramov pre každú časticu. Po nastavení konfigurácie a pripojení pamäte textúr cez funkciu `cudaBindTexture(...)`, sa spustí prvý z dvoch kernelov, ktorý vytvorí jeden čiastkový histogram pre každých 256 pixelov (viď 5.5.1). Odpojíme pamäť textúr pomocou `cudaUnbindTexture(...)` a upravíme konfiguráciu pre druhý kernel (viď 5.5.2).

Ten sčíta čiastkové histogramy a na výstupe dostaneme rozdiel histogramov pre každú časticu. Výsledky sa skopírujú do RAM pamäte kde procesor vyberie časticu s najnižším rozdielom oproti vzoru. Na jej zvýraznenie v snímke je použitý `color_kernel(...)`.

5.5.1 Prvý kernel

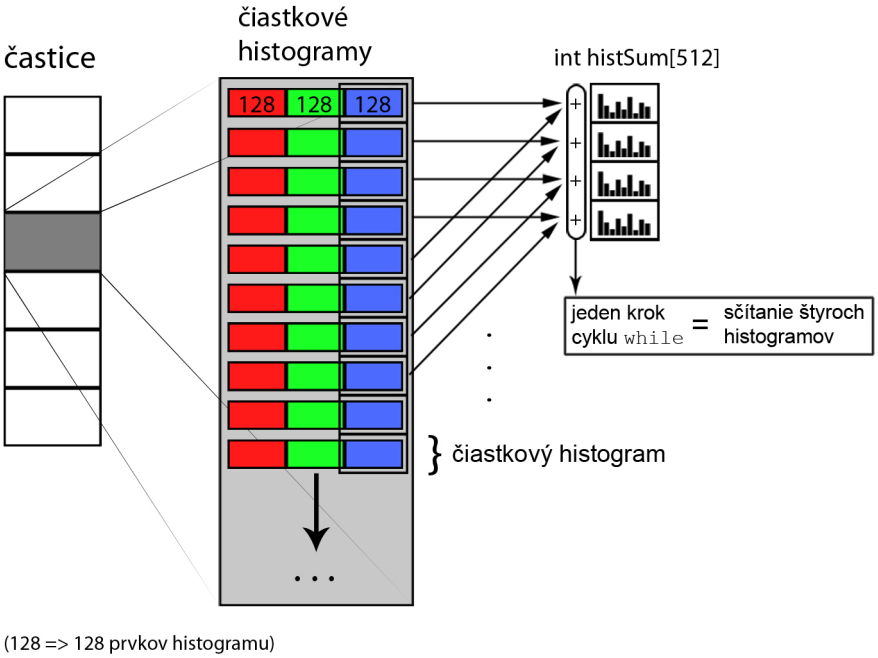
Hlavným problémom v tomto kerneli bol atomický prístup do zdieľanej pamete. Kód bolo treba rozdeliť pomocou makra `SMEM_ATOMICS`, podľa ktorého rozhodneme, či použijeme vstavané atomické funkcie zariadenia alebo explicitné serializovanie prístupov.

V časti pre zariadenia s atomickými funkciami bola použitá funkcia `atomicAdd(int* address, int val)`, ktorá pripočíta do konkrétneho *bank* zdielanej pamäte hodnotu `val`. Z dôvodu úspory kapacity pamäte je jedna položka histogramu uložená na jednom Bajte. Keďže jeden `bank` má 4 Bajty, musíme bitovým posunom upraviť premennú `val`, aby sme boli schopní inkrementovať všetky štyri položky. Po vypočítaní histogramov ich sčítame a jeden výsledný uložíme do globálnej pamäte. Na uloženie jednej položky použijeme 2 Bajty, preto ich musíme upraviť pomocou binárnych posunov.

Zariadenia, ktoré nemajú k dispozícii spomínané atomické funkcie, použijú explicitnú serializáciu. Pri pokuse o zápis do zdieľanej pamäte sa dostanú do cyklu `do{}while()`. Na 5 najvýznamnejších bitov položky sa uloží identifikácia vlákna a zároveň sa aj inkrementuje hodnota v položke. Vlákno si následne skontroluje identifikáciu uloženú v zdieľanej pamäti a ak sa zhodujú, znamená to, že zápis sa mu podaril a môže opustiť cyklus. Spomenutá metóda je implementovaná vo funkcii `addByte(...)`.

5.5.2 Druhý kernel

Jeden blok kernelu `process_semihist_kernel(...)` spočíta jednu farebnú zložku histogramu, jednej častice. Na začiatku si vymedzíme pole `histSum` s 512 prvkami typu `int` v zdieľanej pamäti a vynulujeme ho. Každé vlákno si vypočíta pozíciu začiatku čiastkových histogramov pre časticu ktorú bude sčítať (`glmemBlockPos`) a odskok vypočítaný podľa spracovávaného farebného kanálu (`glmemOffset`). Jedna farebná zložka histogramu je uložená na 256 Bajtoch pričom jeden prvok zaberá 2 Bajty. Na spracovanie týchto dát je použitých 64 vlákien, kde jedno vlákno načíta 4 Bajty dát, teda 2 prvky a tie pripočíta na odpovedajúce miesta v poli `histSum`. Kernel pracuje s 256 vláknami a preto dokáže spracovať 4 čiastkové histogrami v jednom priechode cyklu `for`. Cyklus sa opakuje dovtedy, kým nespracuje dáta jednej farebnej zložky pre celú časticu. V tomto momente sú v poli `histSum` 4 histogramy, ktoré sčítame do jedného a z neho vypočítame rozdiel oproti histogramu vzorky objektu. Časť výpočtu druhého kernelu ilustruje obr. 5.2.



Obr. 5.2: Časť výpočtu druhého kernelu

5.6 Testovanie programu

K otestovaniu aplikácie sú dispozícii tieto zostavy:

- *Notebook* - Hardware: nVidia GeForce 9600M GT (*Compute Capability* 1.1) a Intel Core2Duo taktované na 2,5 GHz.
- *Stolový počítač* (ďalej len *desktop*) - Hardware: nVidia GeForce GTX 285 (*Compute Capability* 1.3) a Intel Core2Duo taktované na 2,66 GHz.

Na účely výkonnostného testovania použijeme dve videá. Obidve majú štandardné rozmery 640 na 480 pixelov. Hlavný rozdiel medzi nimi je vo veľkosti objektu, ktorý v nich budeme sledovať. Druhé testovacie video bude sledovať viac ako desaťnásobne menšiu plochu. Na jednu snímku sa použilo 300 častíc, pričom na úspešné sledovanie objektov väčšinou postačuje 200.

Súhrnné informácie o videách sú uvedené v tabuľke 5.2.

	Rozmery videa Šírka x výška (pixel)	Rozmery objektu Šírka x výška (pixel)	Počet snímkov videa
Znacka.avi	640 x 480	94 x 124	166
Lampa.avi	640 x 480	32 x 33	358

Tabuľka 5.2: Videá použité k testovaniu výkonu

K testovaniu použijeme 10 rôznych implementácií hodnotiacej funkcie. K dispozícii máme osem funkcií počítajúcich podľa *prvej metódy*, pričom jedna je počítaná na procesore a ostatné na CUDA. Počítanie rozdielu histogramov častice a vzorky je zastúpené jednou implementáciou pre procesor a jednou pre CUDA.

Vykonáme 5 rôznych výkonnostných testov. Prvý bude testovať rýchlosť programu na notebooku. Použijeme video *Znacka*, kde budeme sledovať objekt s viac ako jedenásť tisíc pixelmi. Následne otestujeme rýchlosť na desktope s tým istým videom. Odskúšame dve verzie programu preložené pre zariadenia s *compute capability* 1.1 a 1.3. Zistíme tak výkonnostný rozdiel medzi implementáciou algoritmu s atomickými funkciami a bez nich. Tiež uvidíme či verzia prekladača 1.3 prináša zlepšenie rýchlosti oproti 1.1.

Na posledné dva testy použijeme video *Lampa* v ktorom budeme sledovať objekt s relatívne malými rozmermi.

	Číslo testu, Zostava Video Compute capability	1) notebook Znacka 1.1	2) desktop Znacka 1.3	3) desktop Znacka 1.1
0	CPU, R	78,277 ms	70,216 ms	70,216 ms
1	GPU, R, 2K, T8, SM	54,065 ms	4,980 ms	4,646 ms
2	GPU, R, 2K, T512, SM	40,600 ms	2,033 ms	2,041 ms
3	GPU, R, 1K, T512, SM	22,780 ms	1,987 ms	1,990 ms
4	GPU, R, 1K, T512, SM, TM	**20,777** ms	1,951 ms	1,952 ms
5	GPU, R, 1K, T256, SM	21,751 ms	1,800 ms	1,803 ms
6	GPU, R, 1K, T256, SM, TM	23,673 ms	**1,746** ms	**1,760** ms
7	GPU, R, 1K, T384, SM	28,863 ms	2,003 ms	2,001 ms
8	GPU, H, 2K, T512, SM, TM	27,239 ms	2,013 ms	3,629 ms
9	CPU, H	30,729 ms	27,735 ms	27,735 ms

Tabuľka 5.3: Časová náročnosť jednotlivých implementácii na rôznych zostavách a videách (Legenda: 5.4)

CPU	=	Algoritmus prevádzaný na procesore
GPU	=	Algoritmus prevádzaný na CUDA
R	=	Hodnotiaca funkcia počíta rozdiel hodnôt pixelov častice a vzorky
H	=	Hodnotiaca funkcia počíta rozdiel histogramov častice a vzorky
#K	=	# vyjadruje počet kernelov, z ktorých sa skladá funkcia
T#	=	# vyjadruje počet vlákien použitých na jeden blok kernelu
SM	=	Značí použitie zdieľanej pamäte
TM	=	Značí použitie pamäte textúr

Tabuľka 5.4: Legenda k tabuľke 5.3 a 5.5

Test 1. Po prvom teste je vidno, že kernel s 512 vláknami a všetkými optimalizáciami je na notebooku najrýchlejší. Zaujímavý je výsledný čas pre verzie s 256 vláknami, kde použitie pamäte textúr spomalilo kernel. Spôsobené je to nedostatočným opätovným využívaním načítaných dát. Implementácie na CUDA boli rýchlejšie vo všetkých prípadoch v porovnaní s CPU, ale pri histogramových hodnotiacich funkciách bol rozdiel malý.

Test 2. Na základe porovnania s prvým testom je grafická karta na desktope 10 až 20 krát rýchlejšia. Rozdiel medzi CPU a GPU je pri najrýchlejšom z kernelov počítajúcim rozdiel pixelov častice a vzorky až 40 násobný. Taktiež sme získali viac ako 13–krát rýchlejšie spracovanie histogramovej hodnotiacej funkcie.

Test 3. Týmto testom sme zistili výrazný rozdiel medzi implementáciou používajúcou atomické inštrukcie zdieľanej pamäte a s explicitným serializovaním prístupov do pamäte. Okrem toho boli časové rozdiely verzií programu 1.3 a 1.1 štatisticky zanedbateľné a k zrýchleniu tu nedochádza.

	Číslo testu, Zostava Video Compute capability	4) desktop Lampa 1.3	5) notebook Lampa 1.1
0	CPU, R	6,563 ms	7,584 ms
1	GPU, R, 2K, T8, SM	0,518 ms	4,442 ms
2	GPU, R, 2K, T512, SM	0,263 ms	3,361 ms
3	GPU, R, 1K, T512, SM	0,274 ms	2,897 ms
4	GPU, R, 1K, T512, SM, TM	0,268 ms	2,610 ms
5	GPU, R, 1K, T256, SM	**0,211** ms	**2,342** ms
6	GPU, R, 1K, T256, SM, TM	0,218 ms	2,547 ms
7	GPU, R, 1K, T384, SM	0,235 ms	2,884 ms
8	GPU, H, 2K, T512, SM, TM	0,261 ms	2,721 ms
9	CPU, H	3,276 ms	3,522 ms

Tabuľka 5.5: Časová náročnosť jednotlivých implementácii na rôznych zostavách a videách (Legenda: 5.4)

Test 4 a 5. Množstvo spracovávaných dát pri videu z testu 4 je 11–krát menší oproti videu z testu 2, výsledky sa však pohybujú pod 8 násobným zrýchlením. Spôsobuje to pravdepodobne priblíženie sa k hardwarovým možnostiam zariadenia, hlavne taktovanie multiprocesorov tu začína mať silnejší vplyv.

Test fungovania algoritmu

Algoritmus bol testovaný na niekoľkých videách k demonštrovaniu jeho funkčnosti. Prvé video s názvom *Znacka* obsahuje modrú dopravnú značku, čiže stacionárny objekt. V pozadí je okno budovy, ktoré predstavuje rušivý prvok, pretože odráža modrú oblohu. Pri snímaní videa sa pohybovalo kamerou do strán. Vznikla tak scéna so statickým prvkom na dynamickom pozadí. *Prvá metóda* sledovala objekt presne počas celej videosekvencie. Na obr. 5.3 je zobrazený jej priebeh. Pri použití funkcie porovnávania histogramov, vyznačenie objektu v niektorých momentoch presahovalo mimo značku, nikdy z nej však nespadlo. Sledovanie v tomto videu môžeme prehlásiť za úspešné.

Obr. 5.3: Sledovanie statického objektu na dynamickom pozadí s využitím *prvej metódy*

Vo videu *Lampa* bol sledovaný malý kruhový objekt na kontrastnom zelenom pozadí. Kamerou sa rýchlo pohybovalo a prudko sa menil smer. *Prvá metóda* po celý čas úspešne sledovala pohyb objektu. Druhý algoritmus využívajúci histogramy mal pri prudkej zmene smeru tendenciu preskočiť na objekt s podobnou farbou.

Ďalšie video s názvom *Chodec* zobrazuje postavu pohybujúcu sa po ulici. Objekt sledovania je v tomto prípade dynamický. Zároveň sa pohybuje aj kamerou, takže máme aj dynamické pozadie. Postava kráča a mení tak tvar a mierne aj rozmery. V tomto prípade je vhodnejšie použiť histogramy, pretože tento algoritmus sa udržal na objekte až do konca, pričom *prvá metóda* sa stratila po pár snímkach. Výsledok úspešnej metódy je zobrazený na obr. 5.4. Medzi treťou a štvrtou snímkou postava prechádza poza lampu, algoritmus s týmto chvíľkovým prekrytím však nemal problém.

Obr. 5.4: Sledovanie s využitím porovnávania histogramov

Vo videu *Curling* bol sledovaný objekt veľmi malých rozmerov (14 x 17 pixelov). Navyše je v obraze viacero objektov, ktoré sú podobné tvarom aj farbou. Obidve metódy mali tendenciu preskakovať na podobné objekty, pokiaľ bol rozptyl pre generovanie šumu na vysokých hodnotách. Po jeho znížení sa podarilo sledovať objekt počas celej videosekvencie. Ak by však bolo kamerou prudko hýbané, objekt by sa mohol algoritmu stratiť.

Posledné video *Chodec2* zobrazuje ďalšiu pohybujúcu sa tmavú postavu na bledom pozadí. Algoritmus používajúci histogramy začal pri prechode postavy popred trávnatú plochu kmitať smerom dohora, nikdy však z postavy neodskočil úplne. *Prvá metóda* bola v tomto prípade veľmi úspešná, keďže aj napriek rušivému pohybu nôh sa udržala na postave až do konca videa (obr. 5.5).

Obr. 5.5: Sledovanie s využitím *prvej metódy*

Kapitola 6

Záver

Cieľom tejto práce bolo implementovať algoritmy sledovania objektu vo videu na architektúre CUDA. Naprogramoval som metódu časticového filtra, pre ktorú som implementoval dve hodnotiace funkcie spracovávané na grafickej karte. Vytvoril som základnú implementáciu funkcie počítajúcej rozdiel pixelov častice a vzoru. Postupne som na ňu aplikoval optimalizácie a sledoval ich vplyv na výkon. V porovnaní s verziou počítanou na procesore som na CUDA dosiahol 40 násobné zrýchlenie na výkonnej grafickej karte. Vytvoril som tiež hodnotiacu funkciu pracujúcu s histogramami. Pri jej implementácii som sa stretol s problémom absencie špecifických inštrukcií na niektorých starších grafických kartách, ktorý som vyriešil vytvorením dvoch verzií programu. Paralelný výpočet histogramu je náročnejší na možnosti hardwaru, dokázal som však urýchliť výpočet až 13–krát.

Úspešnosť implementovaného algoritmu časticového filtra v sledovaní objektu je veľmi dobrá. Verzia s porovnávaním korešpondujúcich pixelov veľmi presne sleduje objekty, ktoré nemenia tvar. Histogramy sú v tomto smere odolnejšie, ale majú tendenciu kmitať okolo objektu. Ako rozšírenie by preto mohlo byť sľubné pokúsiť sa spojiť tieto dve metódy pri počítaní váhy častice.

Ďalší vývoj aplikácie by mohol pokračovať smerom implementovania odolnosti voči transformáciám objektu ako je zmena veľkosti či rotácia.

Pri vytváraní tejto práce som získal cenné skúsenosti s návrhom a implementáciou paralelných algoritmov. Naučil som sa vytvárať programy pre architektúru CUDA a následne aplikovať optimalizačné metódy. Získal som prehľad v oblasti spracovania obrazu. Oboznámil som sa s niektorými metódami na detekovanie a sledovanie objektov vo videu a hlbšie som sa ponoril do problematiky časticových filtrov.

Literatúra

[1] NVIDIA CUDA Best Practices Guide [online]. 2010-02-04 [cit. 2010-05-09]. URL http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf

[2] NVIDIA CUDA Programming Guide [online]. 2010-02-20 [cit. 2010-05-09]. URL http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf

[3] Arulampalam, S.; Maskell, S.; Gordon, N.; aj.: A Tutorial on Particle Filters for On-line Non-linear/Non-Gaussian Bayesian Tracking. *IEEE Transactions on Signal Processing*, ročník 50, č. 2, February 2002: s. 174–188.

[4] Bradski, G.; Kaehler, A.: *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008, ISBN 978-0-596-516130.

[5] Isard, M.; Blake, A.: CONDENSATION – Conditional Density Propagation for Visual Tracking. *International Journal of Computer Vision*, ročník 29, č. 1, August 1998: s. 5–28, ISSN 0920-5691.

[6] Lehmann, E.: Particle Filtering [online]. 2003-06-06 [cit. 2010-05-08]. URL http://users.cecs.anu.edu.au/~hartley/Vision-Reading-Course/Particle-filtering.pdf

[7] Song, H. A.: OpenGL [online]. [cit. 2010-05-12]. URL http://www.songho.ca/opengl/index.html

[8] Yilmaz, A.; Javed, O.; Shah, M.: Object tracking: A survey. *ACM Comput. Surv.*, ročník 38, č. 4, 2006: str. 13, ISSN 0360-0300.

Zoznam príloh

Príloha A

Ovládanie programu

Program je ovládaný cez príkazový riadok alebo priamo za behu.

Ovládanie cez príkazový riadok

Pri spustení aplikácie bez zadania argumentov sa pokúsi pristúpiť k webkamere. Po výbere oblasti v ktorej sa nachádza objekt, sa automaticky začne sledovanie.

Pomocou nasledujúcich argumentov sa programu na mieste <input> zadávajú potrebné informácie.

--video=<string>
slúži na zadanie názvu vstupného videa vo formáte `avi`,
pri jeho neuvedení sa program pokúsi pristúpiť k webkamere

--save=<string>
slúži na zadanie názvu videa pre uloženie výsledkov programu,
video sa uloží vo formáte `avi`,
pri jeho neuvedení sa výsledky ukladať nebudú

Pri zadávaní informácií o polohe objektu musia byť použité všetky štyri prepínače:

--x=<number>
pozícia objektu na osy x, zadáva sa ľavý okraj oblasti opisujúcej objekt

--y=<number>
pozícia objektu na osy y, zadáva sa spodný okraj oblasti opisujúcej objekt

--w=<number>
šírka oblasti opisujúcej objekt

--h=<number>
výška oblasti opisujúcej objekt

--kernel=<number>
výber kernelu na výpočet váhy častíc, vstupom je číslo 0 až 9,
popis odpovedajúcich kernelov je nižšie

--dev=<number>
rozptyl normálneho rozloženia so stredom v nule, zadaný v pixeloch,
používa sa pri generovaní šumu aplikovaného na častice

--np=<number>
počet použitých častíc k sledovaniu objektu
-cv
zapína zobrazenie okna v OpenCV v ktorom sa zobrazujú obrysy častíc

Príklady použitia:
`cudaPF.exe --video=Test.avi --x=100 --y=50 --w=40 --h=30 --kernel=3 -cv`
použije sa video Test.avi, vysledky sa neukladajú,
hneď po spustení sa začne sledovanie objektu
na zvolenej pozícii a použije sa kernel č. 3,
v ďalšom okne sa vyznačia jednotlivé častice
`cudaPF.exe --dev=25 -cv`
použije sa video z webkamery, vysledky sa neukladajú,
v ďalšom okne sa vyznačia jednotlivé častice, rozptyl je nastavený na 25
`cudaPF.exe --save=Result.avi --kernel=8 --dev=15`
použije sa video z webkamery, vysledky sa ukladajú do súboru Result.avi,
prednastavený je kernel č. 8, rozptyl je nastavený na 15

Identifikácie kernelov:

č.	miesto výpočtu	počítanie váhy	počet vlákien	popis kernelu v...
0	CPU	rozdielom	-	4.7
1	GPU	rozdielom	8	4.5.1
2	GPU	rozdielom	512	4.5.2
3	GPU	rozdielom	512	4.5.3
4	GPU	rozdielom	512	4.5.5
5	GPU	rozdielom	256	4.5.4
6	GPU	rozdielom	256	4.5.5
7	GPU	rozdielom	384	4.5.4
8	GPU	histogramom	256	4.6
9	CPU	histogramom	-	4.7

Ovládanie za behu programu

Počas behu programu je možné kedykoľvek zmeniť použitý kernel pomocou kláves 0 až 9 (značenie – viď predchádzajúca tabuľka). Taktiež je implementovaný výber objektu v okne OpenGL pomocou myši.

Príloha B

Obsah DVD

\bin
obsahuje binárne súbory aplikácie a knižnice potrebné k ich spusteniu, nachádzajú sa tu dávkové súbory pre spúšťanie programu s prednastavenými hodnotami

\video
obsahuje zdrojové videosúbory použité pri testovaní, zahnrnuté sú aj výsledky sledovania objektu použitím obidvoch implementovaných funkcií

\source
- **\manual**
  obsahuje zdrojové kódy pre manual v LaTeXu
- **\program**
  obsahuje zdrojové kódy programu
- **\sprava**
  obsahuje zdrojové kódy technickej správy v LaTeXu

\doc
obsahuje tecnickú správu a manuál vo formáte pdf

README.txt
obsahuje dôležité informácie k obsahu DVD