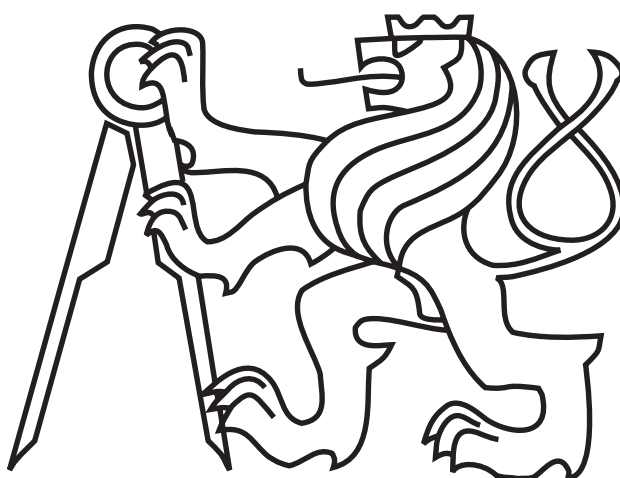


ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE  
FAKULTA ELEKTROTECHNICKÁ

## BAKALÁŘSKÁ PRÁCE



Jan Šváb

**Akcelerace zpracování obrazu hradlovým polem**

Katedra kybernetiky

Vedoucí bakalářské práce: **Ing. Tomáš Krajník**

Praha, 2009

### **Prohlášení**

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady ( literaturu, projekty, SW atd.) uvedené v příloženém seznamu.

V Praze dne 12. 6. 2009



.....  
Podpis



## ZADÁNÍ BAKALÁŘSKÉ PRÁCE

**Student:** Jan Š v á b

**Studijní program:** Elektrotechnika a informatika (bakalářský), strukturovaný

**Obor:** Kybernetika a měření

**Název tématu:** Akcelerace zpracování obrazu hradlovým polem

### Pokyny pro vypracování:

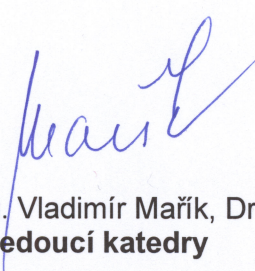
1. Prostudujte metody počítačového vidění používané pro navigaci mobilního robotu.
2. Vyberte vhodný hardware a odpovídající algoritmus.
3. Specifikujte nutné modifikace tohoto algoritmu pro jeho implementaci na daném hardware.
4. Algoritmus implementujte a otestujte.

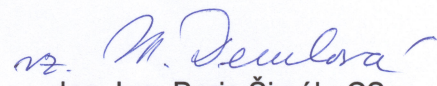
**Seznam odborné literatury:** Dodá vedoucí práce.

**Vedoucí bakalářské práce:** Ing. Tomáš Krajník

**Platnost zadání:** do konce zimního semestru 2009/2010



  
prof. Ing. Vladimír Mařík, DrSc.  
vedoucí katedry

  
doc. Ing. Boris Šimák, CSc.  
děkan

### *Abstrakt*

Bakalářská práce prezentuje návrh hardwarově akcelerované implementace algoritmu Speeded Up Robust Features (SURF). Algoritmus SURF je složen ze dvou částí – detekce významných bodů a tvorba deskriptorů významných oblastí – z čehož první je vhodnější pro řešení pouze logikou programovatelného hradlového pole (FPGA). Druhá část algoritmu je implementována v jazyce C pro běh na standardních procesorech – nejlépe s podporou operací v plovoucí řádové čárce. V práci jsou kromě popisu algoritmu a popisu implementace zahrnuty i výsledky dosažené praktickou aplikací této implementace na FPGA rodiny Virtex-5 FXT, včetně rychlosti zpracování. Podmínky této testovací aplikace simulují použití při úloze navigace mobilního robota. Výsledky jsou srovnány s výsledky jedné z již existujících softwarových implementací, čímž je prokázána praktická použitelnost této implementace.

### *Abstract*

This thesis presents a design of hardware-accelerated implementation of the SURF algorithm (Speeded Up Robust Features). The original SURF algorithm can be divided into two parts – interest point detection and feature descriptor extraction – first part is more suitable for implementation using logic of field-programmable-gate-array (FPGA). Second part of algorithm is implemented in ordinary C programming language suitable for execution by ordinary CPU – optimally with FPU support. The thesis contains, apart from algorithm description and implementation description, results of a practical application using described implementation on FPGA from Virtex-5 FXT family, including timing information necessary to evaluate performance. Practical application setup is similar to the task of outdoor navigation of a mobile robot. Results are compared with existing software-based implementation to prove this implementation's practical usability.

Děkuji vedoucímu práce za důvěru a podporu v průběhu vývoje tohoto projektu. Děkuji své rodině za podporu v průběhu studia.

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 Stručný přehled algoritmů pro obrazovou navigaci mobilních robotů</b>	<b>3</b>
1.1 Algoritmy pracující s mapou prostředí	4
1.2 Algoritmy pracující bez mapy prostředí	4
1.3 Navigace ve strukturovaném prostředí	5
1.4 Navigace v nestrukturovaném prostředí	5
<b>2 Popis algoritmu SURF</b>	<b>6</b>
2.1 Integrální obraz	7
2.2 „Fast-Hessian“ detekce významných bodů	7
2.2.1 Hessián	7
2.2.2 Invariance vůči zvětšení	8
2.2.3 Lokalizace významných bodů	9
2.3 Deskriptor významných oblastí	11
2.3.1 Invariance vůči rotaci	11
2.3.2 Konstrukce deskriptoru	12
<b>3 Implementace</b>	<b>13</b>
3.1 Blok hardware	14
3.1.1 Generátor integrálního obrazu	17
3.1.2 Použití MATLAB pro optimalizaci průběhu výpočtu	17
3.1.3 Blok MasterController	20
3.1.4 Blok TripleMAC	22
3.1.5 Blok HessianCalc	23
3.1.6 Blok LocMaxFinder	24
3.2 Blok software	25
<b>4 Experimenty</b>	<b>27</b>
4.1 Specifika testovací aplikace	27
4.2 Metoda použitá ke zpracování výsledků	28
4.3 Výsledky testovací aplikace	30

<b>5 Závěr</b>	<b>32</b>
<b>Příloha A: Obsah CD</b>	<b>35</b>



# Seznam tabulek

3.1	Velikosti Gaussovských filtrů dle čísel intervalu a oktávy . . . . .	15
3.2	Příklad rozmístění koeficientů pro výpočet odezvy Gaussovských filtrů . . .	19
3.3	Hodnoty koeficientů použité pro výpočet determinantů blokem <i>HessianCalc</i>	23
3.4	Pořadí determinantů v elementárním výpočetním bloku velikosti 2x2 . . .	25
3.5	Přiřazení kódů oktáv pozicím pixelů ve výpočetním bloku 2x2 . . . . .	25
4.1	Základní parametry testovací aplikace . . . . .	28
4.2	Úspěšnosti hledání stejných obrazových detailů . . . . .	30
4.3	Shrnutí výkonnosti testovací aplikace . . . . .	30

# Seznam obrázků

2.1	Ilustrace využití integrálního obrazu . . . . .	7
2.2	Diskretizované a oříznuté Gaussovské kernely a jejich aproximace 2-D filtry . . . . .	8
2.3	Konstrukce prostoru obrazů s různým zvětšením . . . . .	9
2.4	Postup zvětšování 2-D filtrů . . . . .	10
2.5	Demonstrace principu přiřazování dominantního směru orientace deskriptoru . . . . .	11
2.6	Ilustrace postupu výpočtu SURF deskriptoru . . . . .	12
3.1	Blokové schéma rozdělení implementace algoritmu SURF . . . . .	14
3.2	Blokové schéma akcelérátoru výpočtu Hessiánů . . . . .	15
3.3	Ideové schéma generátoru integrálního obrazu . . . . .	18
3.4	Ideové schéma bloku <i>MasterController</i> . . . . .	21
3.5	Ideové schéma bloku <i>TripleMAC</i> . . . . .	22
3.6	Ideové schéma bloku <i>LocMaxFinder</i> . . . . .	24
4.1	Sedmý obraz testovací sekvence . . . . .	31

# Úvod

Rychlé a přesné zpracování obrazu je základním krokem při realizaci aplikací využívajících strojové vidění. Především díky zlevňování výpočetních technologií se strojové vidění používá ve stále větším množství především průmyslových aplikací. Algoritmů používaných k tomuto účelu je celá řada a pro každou aplikaci můžeme většinou vybírat hned z několika již známých. Téměř všechny algoritmy zpracování obrazu však mají jednu společnou vlastnost, kterou je velká výpočetní náročnost. Důvodem je velký objem obrazových dat, které je nutné zpracovat v krátkém časovém úseku. Charakter úkonů při zpracovávání obrazových dat většinou umožňuje vysoký stupeň paralelizace procesů a tím i dosažení dostatečného výpočetního výkonu pro danou aplikaci. Klasickým příkladem tohoto přístupu jsou grafické procesory (GPU) v osobních počítačích. Pro některé aplikace (především z embedded oblasti) však může být použití GPU nevhodné, například z důvodu spotřeby nebo velikosti, a zároveň použití CPU s jedním nebo s několika málo jádry výkonově nedostačující. V případě takovýchto aplikací se jeví použití programovatelné logiky jako velmi vhodná volba, protože poskytuje prostor pro vysokou paralelizaci a optimalizaci úloh a zároveň netrpí neduhy běžných GPU. Tato práce popisuje implementaci algoritmu SURF [1] do hradlového pole, včetně porovnání jejích výsledků s již existujícími implementacemi. Výběr algoritmu byl proveden především na základě srovnání výkonu existujících algoritmů určených k popisu významných vlastností obrazu a faktu, že vybraný algoritmus zatím s největší pravděpodobností implementaci programovatelnou logikou nemá.

Algoritmus SURF byl poprvé představen u příležitosti 9. Evropské konference o počítačovém vidění (ECCV 2006, Graz, Rakousko). Jedná se tedy o algoritmus relativně nový, který se navíc, dle výsledků nejrůznějších testů ([2], [3]), jeví jako nejlepší volba z algoritmů tohoto typu. Tento typ algoritmů má mnoho možností využití – rekonstrukce 3D objektů, rozpoznávání objektů a navigace jsou jen některé z nich. SURF a jiné podobné algoritmy (např. [8]) určené k popisu významných vlastností obrazu se obvykle skládají ze dvou hlavních částí:

**Hledání významných bodů** – Pojmeme významný bod můžeme rozumět například nějaké viditelně kontrastní místo ve zkoumaném obraze — například černý bod na bílém pozadí nebo obecně ostrý roh předmětu kontrastní barvy vůči pozadí. Mnou vybraný algoritmus pro tento účel využívá lokálních maxim determinantů Hessových matic. Z pohledu reálné aplikace je dále důležité, aby tento algoritmus byl schopen hledat významné vlastnosti obrazu nezávisle na jeho zvětšení (přiblížení kamery) k danému detailu. Tohoto efektu je obvykle dosahováno hledáním významných bodů v několika

úrovních zmenšení obrazu.

**Popis okolí významných bodů** – Úkolem této části algoritmu je vytvořit relativně krátký, snadno zpracovatelný a stabilní deskriptor. Stabilitou rozumíme, že stejné objekty na dvou odlišných obrazech (lišících se v osvětlení scény, posunutí, zvětšení a rotaci) budou popsány stejnými, nebo alespoň velmi podobnými, deskriptory. V praktických aplikacích, po získání, deskriptory porovnáváme se sadou již známých nebo v minulosti získaných deskriptorů. Efektivita tohoto kroku je podmíněna snadnou zpracovatelností deskriptoru – tedy například, aby deskriptory byly tvořeny  $n$ -rozměrnými vektory, mezi kterými poté v konečné aplikaci postačuje počítat (např. Euklidovské) vzdálenosti, aby bylo získáno relevantní měřítko podobnosti obrazových detailů. Pro téměř všechny aplikace je v tomto kroku nutno zajistit invarianci vůči zvětšení (přiblížení kamery) k danému detailu a změně osvětlení scény. Některé aplikace vyžadují i invarianci deskriptoru vůči rotaci obrazu.

# Kapitola 1

## Stručný přehled algoritmů pro obrazovou navigaci mobilních robotů

Obrazová navigace je primární způsob navigace člověka a většiny živočichů. Země je osvětlena Sluncem a na základě odrazu jeho paprsků od objektů okolo nás lze tyto velmi efektivně rozpoznat a lokalizovat. Využití metod rozpoznávání obrazu se proto nabízí jako nejvhodnější metoda navigace i pro roboty. Tato kapitola ve stručném přehledu mapuje existující přístupy k tomuto problému a rozdílů mezi nimi. Kapitola čerpá hlavně z obsahu článku [6].

Navigace je úloha, při které podle znalosti cíle a současného bezprostředního okolí stanovujeme akce vedoucí k dosažení tohoto cíle. Úloha navigace se skládá ze dvou podúloh, kterými jsou vyhýbání se překážkám a stanovení směru k cíli. Její řešení závisí především na charakteru prostředí, kterým se robot pohybuje. V tomto ohledu rozlišujeme mezi prostředím *strukturovaným* a *nestrukturovaným*. Navigace uvnitř budov je obvykle navigací ve strukturovaném prostředí.

Navigační algoritmy lze z hlediska používání mapy rozdělit do dvou skupin: algoritmy pracující s mapou prostředí – viz. 1.1 a algoritmy pracující bez mapy prostředí – viz. 1.2. Jakýmsi přechodem mezi oběma přístupy jsou algoritmy, které si mapu prostředí vytvářejí za chodu. Tyto budou popsány v rámci podkapitoly o bezmapové navigaci, protože v podstatě odpovídají intuitivní definici této skupiny. Rozřazení algoritmů do těchto skupin je stále méně jednoznačné, protože mnohé algoritmy kombinují více přístupů.

Dalším možným hlediskem dělení je charakter prostředí pro které jsou algoritmy určeny: algoritmy určené pro navigaci ve venkovním prostředí a algoritmy pracující ve vnitřním prostředí. Dělení podle tohoto hlediska ovlivňuje například jak se tyto algoritmy jsou schopné vypořádat se změnou v osvětlení nebo viditelnosti. Dále lze říci, že pro algoritmy pracující ve vnitřním prostředí je obecně výhodné používat ke své práci mapu, protože ji díky strukturovanosti prostředí lze snadno vytvořit.

## 1.1 Algoritmy pracující s mapou prostředí

Pojmem mapa lze rozumět velmi široké spektrum informačních celků – od téměř přesných CAD modelů až po grafy znázorňující prostorové vztahy mezi objekty prostředí. Její použití je také velmi univerzální – je vhodná jak pro vyhýbání se překážkám, tak pro stanovení cesty k cíli.

Prvním nejčastějším přístupem v této oblasti je *mřížka obsazenosti* (orig. Occupancy Map). Prostředí je rozděleno do diskrétních prostorových bloků, přičemž každý z těchto bloků má přiřazenu informaci o pravděpodobnosti úspěšnosti projetí tohoto bloku robotem. Tento typ mapy je vhodný především pro plánování cesty k cíli.

Dalším typem mapy je tzv. *landmarková mapa*. V této jsou zaneseny objekty, které je robot schopen identifikovat. Tento typ mapy je vhodný především pro určení pozice robota v prostoru relativně k cíli.

Pro plánování cesty mapou typu *mřížka obsazenosti* je nejčastěji používána metoda „virtuálních silových polí“ (orig. VFF – Virtual Force Fields [10]). Tato metoda vytvoří okolo překážek v mapě silové pole, které robota odpuzuje. Naproti tomu cíl cesty emituje silové pole přitahující robota. Součtem působení těchto silových polí v současné pozici robota získáme kýžený směr, kudy se má robot ubírat. Další často používanou metodou plánování cesty je algoritmus  $A^*$ .

Obě metody řízení robota však pracují na základě zjištěné pozice robota v mapě – tento úkol samotný je však mnohem obtížnější. Obecně se skládá ze 4 kroků – získání sensorické informace, extrakce invariant ze sensorických dat, stanovení korespondencí pozorovaných a zaznamenaných objektů, výpočet pozice. Problém lokalizace robota má dva základní přístupy – *globální* a *relativní*. *Globální* lokalizace je intuitivní přístup, při kterém se robot snaží přímo spočítat svou pozici na základě pozice viděných známých objektů. Naproti tomu *relativní* lokalizace očekává, že na začátku robot ví, kde se nachází a tuto pozici musí v průběhu své jízdy pouze „opravovat“. Robot při své jízdě kumuluje nejistoty své pozice (způsobené například prokluzováním kol). Když chce robot poté v nějakém bodě cesty přesně zjistit svoji pozici, vytvoří na základě mapy očekávaný obraz (respektive pozice očekávaných významných objektů v obraze), který je doplněn o regiony nejistot. Z viděného obrazu jsou pak v rámci očekávaných regionů nejistot stanoveny přesné pozice významných objektů, ze kterých je vypočtena přesná poloha.

## 1.2 Algoritmy pracující bez mapy prostředí

Nejjednodušším způsobem bezmapové navigace je navigace reaktivní. Při reaktivní navigaci je akce robota určena pouze na základě aktuálně pořízených obrazů. Velké množství metod reaktivní navigace je inspirováno vzory z přírody - například: *Navigace s pomocí obrazového toku* (orig. Optical Flow Navigation) zpracovává obrazy z laterálně umístěných kamer a na základě jejich rozdílů určí akci robota. Jedním z nejjednodušších postupů v tomto případě je určit, z které kamery se obraz mění rychleji a zatočit na opačnou stranu – tento konkrétní postup modeluje chování včel – viz. [11].

*Navigace s pomocí srovnávání scén* (orig. Appearance-Based Matching Navigation) je způsob, kdy si jedoucí robot pamatuje informace o prostředí, kterým se pohyboval. Na základě srovnávání uložených informací z předchozího učení cesty řídí svůj pohyb k cíli. Příkladem tohoto způsobu je VSRR (orig. View-Sequenced Route Representation) koncept – robot si zapamatuje obrazy posbírané v průběhu učení cesty, viz. [12]. Robot poté při svém pohybu srovnává sledovaný obraz s databází. Jakmile dojde k nalezení obrazu reprezentujícího současnou scénu, je spočten posuv mezi viděným obrazem a obrazem v databázi, podle kterého je dán příkaz řízení motorů. Tento přístup se dá z určitého pohledu označit za algoritmus vytvářející si mapu prostředí za chodu, nicméně forma ukládání informací se mapě podobá pouze velmi okrajově.

*Navigace s pomocí rozpoznávání objektů* (orig. Object Recognition Navigation) využívá zadání cesty pomocí symbolických informací – viz [13]. Na rozdíl od předchozího případu, kdy si robot pamatoval a porovnával obrazy bez interpretace jejich obsahu, tento přístup nejdříve prozkoumá aktuální obrazová data a na základě symbolického příkazu (charakteru „Jdi ke stolu.“) v nich vyhledá chtěný cíl cesty a spočte její trajektorii. V případě, že objekt není v přímém dohledu, je nutno opět použít jakousi zjednodušenou mapu lokálního okolí a na jejím základě určit cestu. Mapu lokálního okolí si robot musí udržovat sám na základě své předchozí cesty – tento přístup se také dá, do jisté míry, považovat za vytváření mapy za chodu.

### 1.3 Navigace ve strukturovaném prostředí

Navigační algoritmy z této oblasti jsou motivovány především aplikacemi řízení autonomních automobilů a proto většina algoritmů implementuje princip *následování cesty* (orig. road-following). Průkopníky v této oblasti jsou projekty autonomních vozidel „NAVLAB“ s navigačním systémem „ALVINN“ pracujícím na bázi neuronové sítě, viz. [14], [15]. Práce těchto algoritmů je založena na předchozích obecných informacích o struktuře prostředí, například že na silnici jsou namalované bílé pruhy a tráva okolo silnice je zelená atd.

### 1.4 Navigace v nestrukturovaném prostředí

Typickým příkladem navigace v nestrukturovaném prostředí je navigace autonomních vozidel pro průzkum cizích planet (např. v rámci mise „Mars Pathfinder“). Základní úlohou algoritmu navigujícího pohyb robota k danému cíli v tomto prostředí je vyhýbání se hazardním situacím – tyto jsou především nepřekonatelné překážky, ale například i neočekávané vlastnosti povrchu (např. „tekutý písek“). Nestrukturovaná scéna je zkoumána především technikami kamerového sterea nebo promítáním známého vzoru do zorného pole kamery.

# Kapitola 2

## Popis algoritmu SURF

V této kapitole bude detailně popsána architektura algoritmu SURF tak, jak je definována v původním článku [1]. Jak již bylo řečeno v úvodu, algoritmus je možno rozdělit do několika funkčních částí. Pro účely této kapitoly však použijeme jemnější dělení než jaké je popsáno v úvodu. Stručný popis obsahu jednotlivých podkapitol:

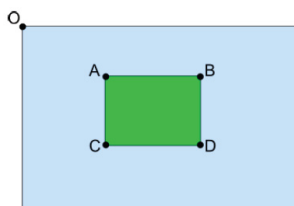
- 2.1 Integrální obraz** – První funkční blok algoritmu. Integrální obraz používají pro zrychlení své práce všechny následující bloky.
- 2.2 Detekce významných bodů** – Blok zajišťující detekci významných bodů v obraze. V originální literatuře pojmenován „Fast-Hessian Detector“.
  - 2.2.1 Hessián** – Konstrukce determinantu Hessovy matice a její optimalizace a zjednodušení pro účely popisovaného algoritmu.
  - 2.2.2 Invariance vůči zvětšení** – Konstrukce prostoru zvětšení (orig. „Scale-Space“).
  - 2.2.3 Lokalizace významných bodů** – Hledání lokálních maxim (významných bodů) a interpolace jejich pozic.
- 2.3 Deskriptor významných oblastí** – Blok zajišťující výpočet vlastních deskriptorů. Algoritmus SURF ve své základní podobě poskytuje invarianci jak vůči zvětšení, tak vůči rotaci obrazu. V mnoha aplikacích ale mají rozpoznávané obrazy stále stejnou orientaci, a proto se používá zjednodušení algoritmu nazývané U-SURF (orig. Upright SURF), které nepoužívá princip popsaný v kapitole 2.3.1.
  - 2.3.1 Invariance vůči rotaci** – Zajišťuje výpočet orientace výsledného deskriptoru (kompenzuje natočení obrazu v ose kamery).
  - 2.3.2 Konstrukce deskriptoru** – Popisuje matematický princip konstrukce deskriptoru okolo nalezeného významného bodu.



## 2.1 Integrální obraz

Integrální obraz v každém svém pixelu v podstatě obsahuje hodnotu součtu hodnot obrazových pixelů od počátku obrazu až do daného bodu. Máme-li obraz  $I$ , můžeme hodnotu bodu integrálního  $I_\Sigma$  obrazu vyjádřit takto:

$$I_\Sigma(x, y) = \sum_{i=1}^x \sum_{j=1}^y I(i, j) \quad (2.1)$$



Obrázek 2.1: Ilustrace využití integrálního obrazu

V okamžiku, kdy je k dispozici integrální obraz, zabere výpočet libovolně velké sumy přes hodnoty původního obrazu vždy pouze 4 přístupy do paměti. Například pro výpočet hodnoty sumy hodnot pixelů uvnitř zeleného obdélníku na obrázku 2.1 nám stačí znát hodnoty pixelů A, B, C a D odpovídajícího integrálního obrazu. Hodnota sumy je poté  $\Sigma = I_\Sigma(A) + I_\Sigma(D) - I_\Sigma(C) - I_\Sigma(B)$ .

## 2.2 „Fast-Hessian“ detekce významných bodů

### 2.2.1 Hessián

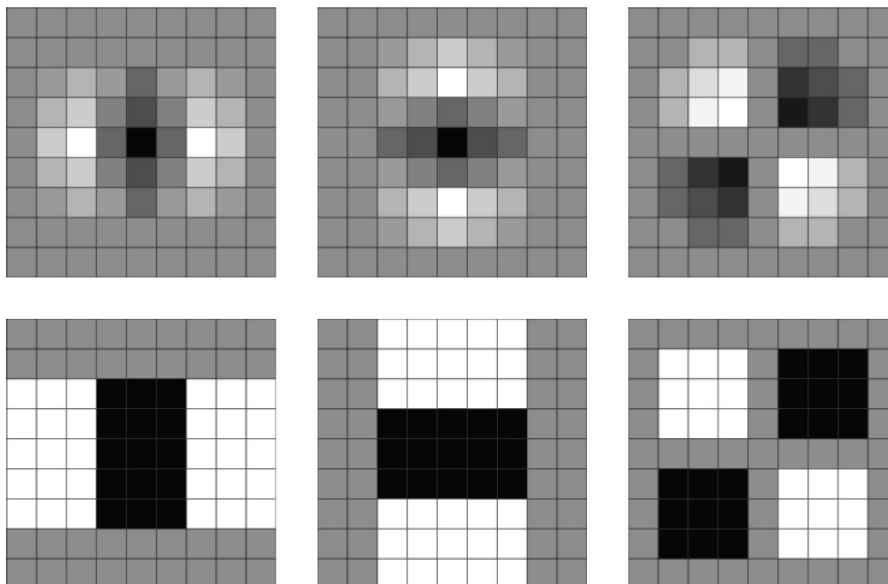
Hessián je zkrácený název pro determinant Hessovy matice. Hessova matice je čtvercovou maticí druhých parciálních derivací skalární funkce. Pro skalární funkci  $f(x, y)$  dvou proměnných má rozměr 2x2 a její determinant má následující tvar:

$$\mathcal{H}(x, y) = \det \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix} = \frac{\partial^2 f}{\partial x^2} \frac{\partial^2 f}{\partial y^2} - \left( \frac{\partial^2 f}{\partial x \partial y} \right)^2 \quad (2.2)$$

Pro praktické využití tohoto determinantu jako detektoru významných bodů obrazu stačí funkci  $f(x, y)$  nahradit hodnotami jasu obrazu a druhé parciální derivace nahradit konvolucí obrazu a vhodného Gaussovského kernelu. Gaussovské kernely snadno vytvoříme pro aproximaci derivací ve směru  $x$ ,  $y$  a kombinovaném směru  $xy$ . Hessovu matici tedy napíšeme ve tvaru:

$$H(x, y, \sigma) = \begin{pmatrix} L_{xx}(x, y, \sigma) & L_{xy}(x, y, \sigma) \\ L_{xy}(x, y, \sigma) & L_{yy}(x, y, \sigma) \end{pmatrix} \quad (2.3)$$

Kde  $L_{xx}(x, y, \sigma)$  značí konvoluci druhé derivace Gaussovy funkce  $\frac{\partial^2 g(\sigma)}{\partial x^2}$  s obrazem (obdobně pro  $L_{yy}$  a  $L_{xy}$ ). Použití konvoluce s Gaussovským kernelem umožňuje změnou parametru  $\sigma$  v podstatě měnit zvětšení, na kterém je determinant počítán.



Obrázek 2.2: Diskretizované a oříznuté Gaussovské kernely a jejich aproximace 2-D filtry

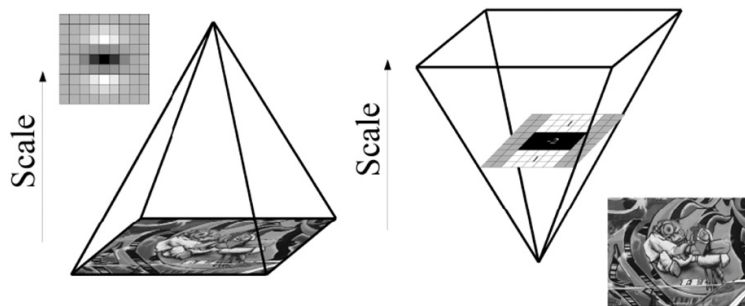
Pro zjednodušení výpočetních operací aproximujeme Gaussovské kernely 2-D filtry s celočíselnými koeficienty. Tato aproximace je znázorněna na obrázku 2.2. Horní řádek představuje diskretizované a oříznuté Gaussovské kernely -  $L_{xx}$ ,  $L_{yy}$ ,  $L_{xy}$ , dolní řádek jejich aproximace 2-D filtry -  $D_{xx}$ ,  $D_{yy}$ ,  $D_{xy}$ . Černé čtverečky v obrázcích pro  $D_{xx}$ ,  $D_{yy}$  představují váhování daného obrazového pixelu koeficientem -2. Černé čtverečky v obrázku pro  $D_{xy}$  představují váhování daného obrazového pixelu koeficientem -1. Bílé čtverečky představují váhování daného obrazového pixelu koeficientem 1, šedé 0. Tato aproximace Gaussovských kernelů samozřejmě zkresluje hodnotu Hessiánu, pro kompenzaci tohoto vlivu je možno dle [1] použít následující vztah:

$$\mathcal{H} = D_{xx}D_{yy} - (0,9D_{xy})^2 \quad (2.4)$$

Tento vztah zajišťuje znatelné zvýšení výpočetní rychlosti při zachování použitelné přesnosti výpočtu.

## 2.2.2 Invariance vůči zvětšení

Jak již bylo zmíněno v úvodu, pro mnoho praktických aplikací tohoto algoritmu je klíčové, aby algoritmus byl schopen detekovat identické významné body i při změně zvětšení obrazu. Pro dosažení této vlastnosti algoritmus musí tyto body pokaždé hledat v několika



Obrázek 2.3: Konstrukce prostoru obrazů s různým zvětšením – standardní přístup, algoritmus SURF

úrovních zvětšení obrazu. Tohoto efektu je dosaženo zavedením třetího rozměru (vedle standardních plošných souřadnic bodu  $x, y$ ), který reprezentuje úroveň zvětšení. Hledání lokálních maxim je poté prováděno v tomto třírozměrném prostoru (orig. Scale-Space). Většina algoritmů v tomto místě provádí iterativní zmenšování obrazu převzorkováním. Zvětšení plochy Gaussovského kernelu, vyjadřujícího jednu z derivací, je matematicky ekvivalentní výpočtu této derivace na úměrně zmenšeném obraze. Jelikož algoritmus SURF používá k výpočtu konvolucí obrazu s Gaussovskými kernely integrální obraz, je výpočet stejně dlouhý pro libovolnou velikost Gaussovského kernelu, tudíž i úroveň zvětšení, na které je počítán. Zvětšování Gaussovského kernelu ilustruje obr. 2.3. Díky této optimalizaci dosahuje algoritmus SURF značného zrychlení oproti svým konkurentům.

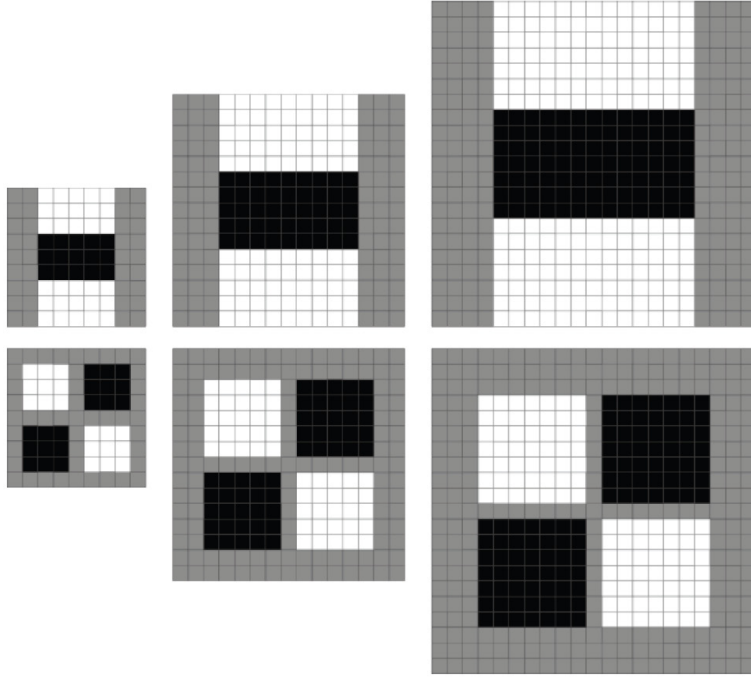
Nejmenší velikost 2-D filtru je  $9 \times 9$  a odpovídá Gaussovskému kernelu s  $\sigma = 1.2$ , pro potřeby dalšího textu ji prohlásíme za „základní zvětšení“  $s = 1.2$  (další úrovně zvětšení lze vypočítat přímo úměrně zvětšování hrany filtru – např. pro  $27 \times 27$  je  $s = 3 \times 1.2 = 3.6$ ). Pro zachování správného poměru velikostí jednotlivých oblastí je nutno při každém kroku zvětšení filtru přidat 6 pixelů každé jeho hraně.

Při zvětšení 2-D filtru dojde ke zvětšení hodnoty jeho odezvy. Pro kompenzaci tohoto jevu je výsledná odezva filtru zmenšena přímo úměrně ploše filtru. Pro další optimalizaci práce algoritmu je zavedeno dělení úrovní zvětšení na oktávy a intervaly. Mezi jednotlivými oktávami je zdvojnásoben krok zvětšování filtru a vzorkovací krok pro extrakci významných bodů. Interval 1 v oktávě 1 má tedy velikost filtru  $9 \times 9$  a vzorkovací krok 1, interval 2  $15 \times 15$  atd. . . Interval 1 v oktávě 2 má velikost filtru  $15 \times 15$  a vzorkovací krok 2, interval 2  $27 \times 27$  atd. . . Příklad meziintervalového zvětšování 2-D filtrů je znázorněn na obr. 2.4.

### 2.2.3 Lokalizace významných bodů

Jak vyplývá z předchozího textu, nyní máme k dispozici hodnoty Hessiánů pro jednotlivé body obrazu v různých zvětšeních. Postup detekce významných bodů může být rozdělen do tří kroků.

Prvním krokem je prosté oříznutí prahovou hodnotou. Úroveň prahové hodnoty významně ovlivňuje počet detekovaných významných bodů. Tato hodnota by měla být nastavena



Obrázek 2.4: Postup zvětšování 2-D filtrů

podle potřeb finální aplikace.

Po aplikaci filtrování prahovou hodnotou se provádí hledání lokálních maxim. Toto hledání je prováděno nejen v plošném okolí bodu, ale i na dvou sousedících intervalech zvětšení. Hodnota determinantu je tedy vždy porovnávána s 26 nejbližšími sousedícími hodnotami (porovnávání je nutno provádět vždy pouze mezi sousedícími intervaly, nikoliv oktávami).

Posledním krokem při detekci významných bodů je interpolace pozice maxim do sub-pixelové přesnosti. Pro provedení této interpolace nejdříve vyjádříme Taylorův rozvoj 2. stupně Hessiánu, centrováný kolem zkoumaného bodu, a to i s přihlédnutím k souřadnici zvětšení.

$$\mathcal{H}(\mathbf{x}) = \mathcal{H} + \frac{\partial \mathcal{H}^T}{\partial \mathbf{x}} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \frac{\partial^2 \mathcal{H}}{\partial \mathbf{x}^2} \mathbf{x} \quad (2.5)$$

Interpolovaná lokace extrému  $\hat{\mathbf{x}} = (x, y, \sigma)$  je nalezena jako řešení rovnice, kde je derivace Taylorova rozvoje (2.5) rovna nule.

$$\hat{\mathbf{x}} = - \frac{\partial^2 \mathcal{H}^{-1}}{\partial \mathbf{x}^2} \frac{\partial \mathcal{H}}{\partial \mathbf{x}} \quad (2.6)$$

Jelikož hodnoty Hessiánů máme již spočteny z minulých částí algoritmu, dosazujeme do vztahu (2.6) už jen difference těchto hodnot. Původní Taylorův rozvoj byl centrován

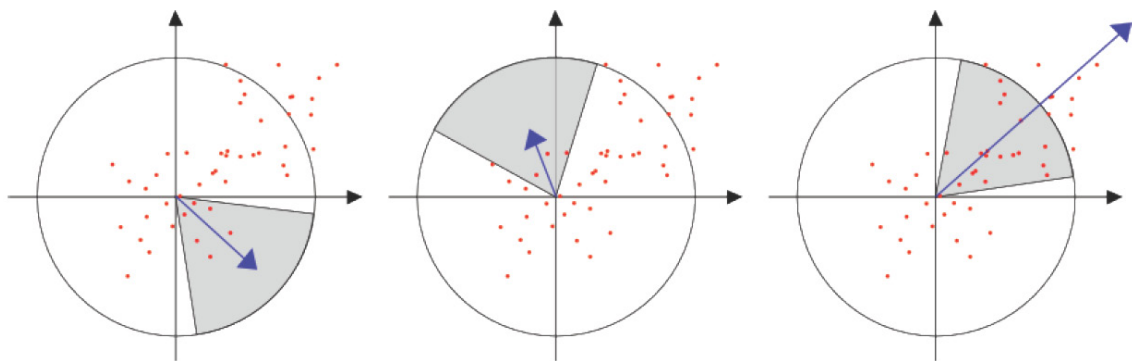
kolem interpolovaného bodu, proto je za správný výsledek považováno, vychází-li  $\hat{\mathbf{x}}$  v absolutní hodnotě každé své složky menší než 0,5. V opačném případě polohu bodu opravíme (po zaokrouhlení) o  $\hat{\mathbf{x}}$  a interpolaci provedeme znovu, dokud nevyčerpáme předem pevně stanovený počet interpolačních kroků a nebo nenajdeme vyhovující bod.

## 2.3 Deskriptor významných oblastí

Deskriptor popsáný specifikací algoritmu SURF v podstatě představuje 64-rozměrný vektor hodnot spočtený na okolí detekovaného významného bodu. Velikost tohoto okolí závisí na měřítku, na kterém byl daný bod detekován. Orientace tohoto okolí je určena před vlastním procesem výpočtu deskriptoru, čímž je dosaženo invariance vůči rotaci. Ve zjednodušené verzi algoritmu SURF (orig. U-SURF) se orientace okolí nepřizpůsobuje vlastnostem obrazu, ale zůstává vždy stejná.

### 2.3.1 Invariance vůči rotaci

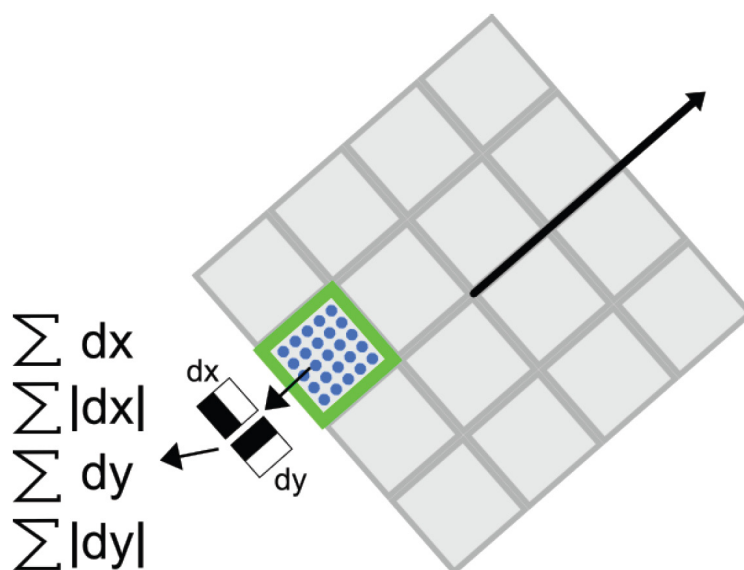
V okolí každého významného bodu je stanovena kruhová oblast s poloměrem  $6s$  ( $s$  viz. kap. 2.2.2). V této oblasti jsou vyčísleny odezvy Haarových vlnkových filtrů (orig. Haar Wavelets) ve směrech os  $x$  a  $y$  s velikostí hrany  $4s$  a krokem  $s$ . Odezvy vlnkových filtrů v dané oblasti jsou váhovány Gaussovou funkcí se směrodatnou odchylkou  $\sigma = 2,5s$ . V každém bodě kruhové oblasti tedy získáme váhovanou odezvu vlnkových filtrů ve směrech  $x$  a  $y$ . Tyto dvě odezvy prohlásíme za vektor. Okolo středu stanoveného kruhu orotujeme kruhovou výseč velikosti  $\pi/3$ , ve které sčítáme Euklidovské normy vektorů. Dle pozorování autorů je algoritmus stabilní pro změny orientace zhruba v rozmezí  $\pm 15^\circ$ . Pro zajištění stability a přesnosti výpočtu je tedy vhodné volit krok rotace výseče na cca  $10^\circ$ . Úhel vektoru, daného součtem odezev filtrů ve výseči, která vykazuje největší součet obsažených vektorů, je poté prohlášen za dominantní a deskriptor je dále počítán ve směru tohoto úhlu.



Obrázek 2.5: Demonstrace principu přiřazování dominantního směru orientace deskriptoru

## 2.3.2 Konstrukce deskriptoru

Při konstrukci deskriptoru je nejprve nutno okolo zvoleného významného bodu vymezit čtvercovou oblast o hraně  $20s$  ( $s$  viz. kap. 2.2.2) a natočit ji podle výsledku předešlé části algoritmu. V případě varianty algoritmu U-SURF je natáčení čtverce samozřejmě přeskočeno. Tato čtvercová oblast je rozdělena na 16 stejně velkých čtvercových oblastí, z nichž každá obsahuje 25 rovnoměrně rozložených vzorkovacích bodů. V každém z těchto bodů jsou spočteny odezvy Haarových vlnkových filtrů o hraně velikosti  $2s$  pro směry  $d_x$  a  $d_y$ . Zmíněné směry  $d_x$  a  $d_y$  nerepresentují směry obrazových os  $x$  a  $y$ , nýbrž  $d_x$  představuje směr odpovídající orientaci deskriptoru určený dle 2.3.1 a  $d_y$  je směr na něj kolmý. Pro zvýšení robustnosti vůči chybám lokalizace a deformacím jsou odezvy vlnkových filtrů váhovány Gaussovou funkcí se směrodatnou odchylkou  $\sigma = 3,3s$ . Každá z 16-ti zmíněných oblastí je poté popsána 4-rozměrným vektorem  $\mathbf{v} = [\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|]$ . Tyto vektory nám poskytnou celkem  $4 \times 4 \times 4 = 64$  hodnot, které tvoří základ SURF deskriptoru. Po jejich získání jsou tyto hodnoty normalizovány tak, aby výsledný 64-rozměrný vektor měl délku 1.



Obrázek 2.6: Ilustrace postupu výpočtu SURF deskriptoru

Dalším významným prvkem v SURF deskriptoru je znaménko stopy Hessovy matice (2.3). Toto znaménko reprezentuje rozdíl mezi situacím, kdy je významný bod reprezentován světlým „bodem“ na tmavém pozadí a obráceně. Výhodou samozřejmě je, že tato znaménka jsou pro účely tvorby deskriptoru pouze zachována z výpočtu Hessiánů (viz. kap. 2.2.1).

# Kapitola 3

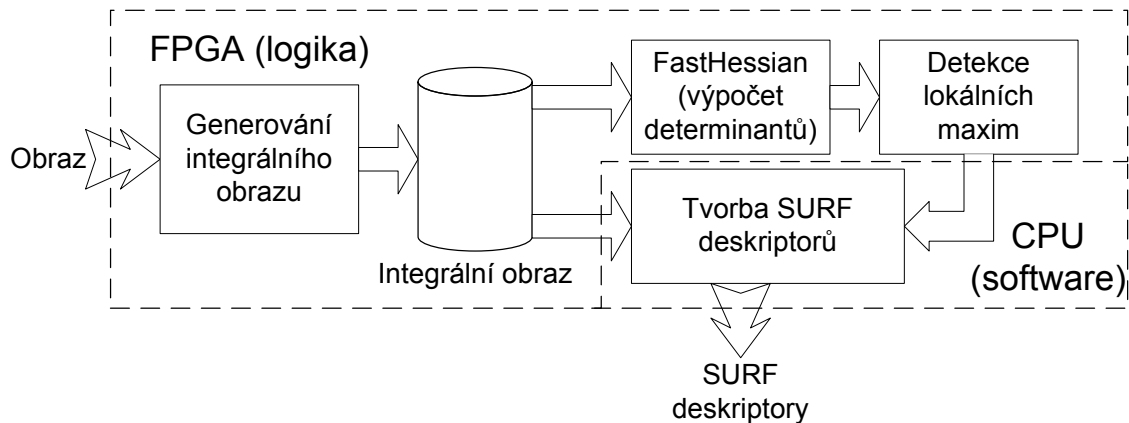
## Implementace

Způsob implementace algoritmu do programovatelného hradlového pole (FPGA) je oproti implementaci na jakýkoli běžný procesor velmi odlišný. Při programování kódu musí mít vývojář stále na paměti, že jím psaný kód bude v konečném důsledku převeden na zapojení logických obvodů a nebude se tedy provádět řádku po řádce jako je tomu u běžných CPU. Můžeme zde vysledovat jakousi paralelu s neprocedurálním programováním, ale přirovnání ke kreslení elektrického schématu považuji za výstižnější. Konec konců editor schémat je jedním z možných způsobů „programování“ FPGA.

Klasickým příkladem využití předností FPGA je implementace digitálního filtru. V běžném CPU by byl výpočet jedné hodnoty odezvy filtru složen z cyklického běhu mnoha instrukcí násobení a sčítání na jedné aritmeticko-logické jednotce, které by trvaly mnoho hodinových taktů. V FPGA je tato odezva vypočtena jedna na každém hodinovém taktu, protože potřebné násobičky a sčítačky se jednoduše zapojí do kaskády. Jeden hodinový takt bude v FPGA znamenat  $n$  matematických operací, kdežto v CPU pouze jednu – tato vlastnost charakterizuje způsob paralelizace výpočtu v FPGA. Díky architektuře moderních FPGA může  $n$  být nejen v řádu desítek, ale i stovek a navíc při použití dedikovaných funkčních bloků může frekvence synchronizačního hodinového signálu sahát až ke stovkám megahertz. Ze zmíněného příkladu vychází FPGA jako velmi výkonná platforma pro realizaci výpočetních operací, protože kombinací vysokých  $n$  a vysokých hodinových kmitočtů lze dosáhnout téměř libovolně velkého výpočetního výkonu. Pokud se ale nad příkladem zamyslíme z pohledu programátora, zjistíme, že se jedná o velmi triviální úlohu, která má složitost i paměťovou náročnost  $n$  a navíc běží na kontinuálním jednorozměrném datovém proudu. Naproti tomu při zamýšlení z pohledu obvodáře jakékoli zesložité úlohy bude znamenat znatelné zesložité realizujícího logického obvodu a tudíž i jeho větší zpoždění a tudíž i snížení meze synchronizační frekvence, do které bude obvod fungovat spolehlivě. Tato pozorování jsou výrazným limitujícím faktorem pro architekturu úloh, které je FPGA schopno efektivně (a tudíž rychle) řešit.

Velkou část algoritmů zpracovávajících obraz nelze takto triviálně (a tedy optimálně) v FPGA implementovat a to již jen z důvodu, že obraz je v lepším případě signál rozměru  $r \times c$  (uvažujeme-li pouze obraz ve stupních šedi nebo jen jednu barevnou složku). Po přečtení kapitoly 2 je již bez pochyby zřejmé, že implementace algoritmu SURF vyžaduje značně

netriviální způsob řešení (triviálním způsobem bychom totiž získali nereálně velký obvod). Při dalším zamyšlení nad architekturou algoritmu SURF dojdeme k závěru, že realizace některých jeho částí pouze logikou by byla neúnosně složitá a ani by nepřinesla významné navýšení výkonu oproti softwarovému řešení. Algoritmus byl tedy rozdělen na část, která je realizována pouze logikou (hardwarově) a část, kterou realizuje program vykonávaný na běžném CPU (software) (obr. 3.1).



Obrázek 3.1: Blokové schéma rozdělení implementace algoritmu SURF

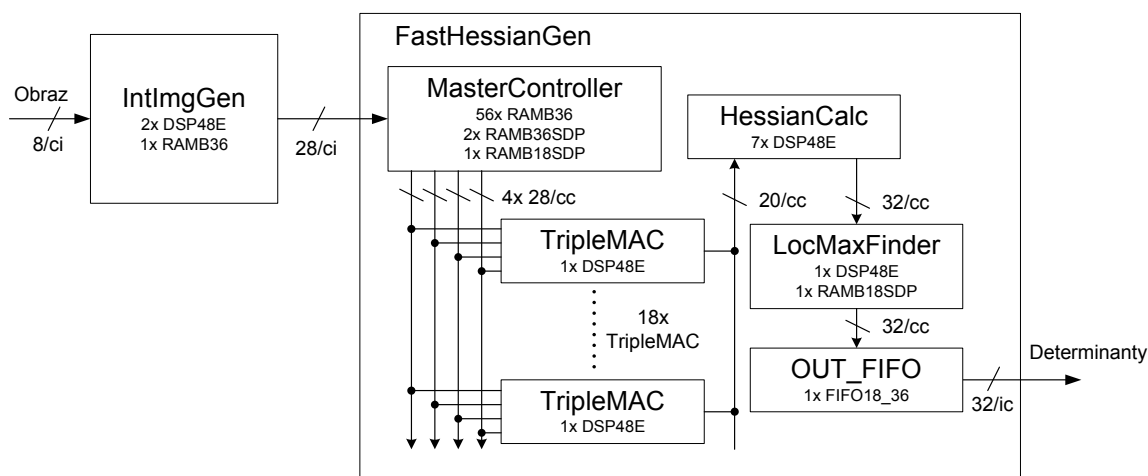
Od počátku projektu bylo jedním z kýžených cílů dosáhnout vyšší rychlosti zpracování obrazu než u implementace na procesoru architektury x86<sup>1</sup>. Díky složitosti výpočtu determinantů (a jejich množství) jsem dlouho nevěděl, zda (a pokud ano, jaké) FPGA bude schopno této rychlosti dosáhnout. Z tohoto důvodu jsem přistoupil k volbě hradlového pole z rodiny Virtex-5. Tato rodina poskytuje (pro optimální implementaci výpočtu determinantů nezbytné) hard-core primitivní bloky DSP48E (48-bitová sčítačka a 25x18-bitová násobička) a dvouportovou blokovou paměť v dostatečné velikosti. Tato rodina zároveň byla i nejvyšší třídou FPGA produkovaných spol. Xilinx v době volby – proto její logické bloky a primitivní hard-core bloky umožňují práci na nejvyšších dostupných kmitočtech hodinových signálů realizovatelných hradlovým polem v době volby. Volba mi také zajistila jistotu, že pokud by implementace nedosahovala požadovaného výkonu, nebudu moci spekulovat o případném vlivu použití FPGA vyšší třídy.

### 3.1 Blok hardware

Obrázek 3.2 znázorňuje blokové schéma akcelerátoru výpočtu Hessiánů – *FastHessian-Gen*. Úloha výpočtu Hessiánů vyžaduje nejdříve provést konvoluce obrazu s Gaussovskými filtry a poté z jejich odezev vypočítat determinanty dle vzorce 2.4. Počet determinantů závisí na počtu úrovní zvětšení, na kterých se počítají. Pro redukci počtu přístupů do

<sup>1</sup>Pro obraz 1024x768x8bit zpracuje 3GHz x86 procesor cca 3 snímky za vteřinu.





Obrázek 3.2: Blokové schéma akcelérátoru výpočtu Hessiánů. Jména funkčních bloků odpovídají názvům VHDL entit, u každého bloku je uvedena „spotřeba“ dedikovaných hard-core funkčních bloků. Čísla před lomítkem značí šířku sběrnice, dvojice písmen za lomítkem odpovídá: cc = „core clock“; ci = „interface clock“.

paměti se odezvy Gaussovských filtrů počítají z hodnot integrálního obrazu. K vypočtení odezvy filtrů  $D_{xx}$  a  $D_{yy}$  je třeba přečíst 8 hodnot integrálního obrazu, pro filtr  $D_{xy}$  16. Jak je vysvětleno v kapitole 2.2.2, počet těchto přístupů je nezávislý na úrovni zvětšení, na které se daná odezva počítá. Na zvětšení však závisí velikost okolí, ze kterého je nutno tyto hodnoty číst. Například pro výpočet odezvy na úrovni zvětšení oktáva 1 a interval 1 (základní zvětšení) pro pozici determinantu  $[0;0]$  je nutno mít přístup k hodnotám z intervalu sloupců  $\langle -5, 4 \rangle$  (pro řádky stejně). Pro úroveň zvětšení oktáva 2, interval 4 (největší, pro které je současná verze určena) je tento interval požadovaného počtu řádků a sloupců již velikosti  $\langle -26, 25 \rangle$ .

Číslo oktávy	Číslo intervalu			
	1	2	3	4
1	9	15	21	27
2	15	27	39	51

Tabulka 3.1: Velikosti Gaussovských filtrů dle čísel intervalu a oktávy použité v mé implementaci.

Jak vidíme z tabulky 3.1, je v mnou zvolené konfiguraci 6 unikátních velikostí filtru. V kombinaci s faktem, že výpočet determinantů v oktávě 2 se provádí s vzorkovacím krokem 2 pixely, snadno spočteme, že v bloku 2x2 pixely je nutno spočítat 18 determinantů (4x4 pro oktávu 1 a chybějící 2 pro levý horní pixel pro oktávu 2). Blok velikosti 2x2

byl zvolen jako jednotka pro zpracování determinantů, protože díky faktu, že v oktávě 2 se determinanty počítají s krokem 2 pixely, je to nejmenší blok obrazu, ve kterém se všechny operace výpočtu opakují. Z předchozího je dále zřejmé, že pro výpočet všech determinantů v tomto bloku je nutno mít přístup k intervalu  $\langle -26, 24 \rangle$  sloupců a řádků okolo tohoto bloku. Determinanty se tedy nevyčísľují v blocích, které jsou vzdáleny méně než 26 pixelů od některé z hran obrazu. Dále je zřejmé, že k výpočtu 18 determinantů je nutno provést 576 čtení hodnot integrálního obrazu <sup>2</sup>. Velkou nevýhodou algoritmu je, že čtení těchto hodnot z okolí bodu (které má v případě největší úrovně zvětšení velikost 52x52) je značně nesequenční. Obraz je v paměti uložen jako posloupnost posloupností hodnot jasu pixelů v jednotlivých řádcích. Pokud chceme přečíst hodnotu jasu pixelu v sousedním sloupci, jedná se o změnu adresy o 1, pokud ale chceme číst ze sousedního řádku, změna adresy je rovná počtu sloupců. Je proto zřejmé, že při výpočtu jednoho bloku determinantů se adresa čtení z paměti změní až o 52 x počet sloupců. Synchronní počítačové paměti jsou svojí architekturou optimalizovány pro sekvenci přístup. Způsob čtení hodnot integrálního obrazu nutný k výpočtu bloku determinantů je pro ně tudíž krajně neefektivní. Pro efektivně pracující výpočet je tedy nutné mít v paměti, umožňující nesequenční čtení bez penalizace výkonu, uloženo 52 řádek obrazu. Toto FIFO integrálního obrazu je společně s logikou řízení výpočtu determinantů implementováno v bloku *MasterController* (kap. 3.1.3).

Blok *MasterController* realizuje FIFO integrálního obrazu pomocí blokové RAM, která umožňuje dvouportový asynchronní přístup. Tohoto je s výhodou využito ke změně hodinové domény signálů z *clk\_if* („interface clock“) do *clk\_core* („core clock“) <sup>3</sup>. Ze zřejmých důvodů je totiž výhodné, aby jádro výpočtu běželo na vyšší frekvenci než rozhraní přenosu obrazu. Hodnoty integrálního obrazu jsou z bloku *MasterController* předávány k výpočtu do dalších bloků přes čtyři 28-bitové sběrnice synchronizované dle *clk\_core*. Jejich pracovní název ve VHDL kódu je *sPixelBus0-3*. I s použitím veškerých optimalizací popsaných v kapitole 3.1.2 je totiž nutno pro výpočet determinantů ve 2 oktávách a 4 intervalech na jednom obraze velikosti 1024x768 po těchto sběrnicích přenést zhruba 271MB dat.

Bloky *TrippleMAC* (kap. 3.1.4) realizují výpočty odezev jednotlivých Gaussovských filtrů. Jeden blok počítá odezvy 3 různých filtrů a pro výpočet jednoho bloku determinantů je nutno znát 54 těchto odezev. Zvolit počet těchto jednotek na 18 je tedy s ohledem na mnou zvolenou konfiguraci algoritmu neoptimálnější volba.

Výsledky z bloků *TrippleMAC* jsou přenášeny do *HessianCalc* (kap. 3.1.5) po 20-bitové sběrnicí pracující také v doméně *clk\_core*. Do budoucna ovšem zvažují převedení této sběrnice na nižší doménu, jelikož je její kapacita využita zhruba pouze na 50% – toto lze zjistit z poměru počtu hodinových cyklů nutných k vyčtení všech hodnot integrálního obrazu a počtu odezev Gaussovských filtrů nutných pro výpočet jednoho bloku determinantů.

Blok *LocMaxFinder* (kap. 3.1.6) provádí prahování hodnot determinantů a vyhledávání

<sup>2</sup>576 = 18 × (16 + 8 + 8) kde 18 je počet unikátních determinantů v bloku 2 × 2 pixely při výpočtu pro 2 oktávy a 4 intervaly a 16 a 8 jsou počty přístupů do paměti nutné k vyčíslení odezev filtrů  $D_{xy}$  a  $D_{yy}$  nebo  $D_{xx}$

<sup>3</sup>V mnou provedené testovací aplikaci je *clk\_if* 100MHz a *clk\_core* 200MHz.

lokálního maxima. Hodnota prahování je softwarově nastavitelná a tento blok je možno, především pro účely ladění, přemostit. Blok *OUT\_FIFO* již pouze instanciuje jednu z primitivních komponent FPGA – asynchronní FIFO – které se používá pro přechod zpět do hodinové domény *clk\_if* a vyrovnává nekonzistence toku determinantů.

*FastHessianGen* musí samozřejmě kromě datových signálů distribuovat i signály řídicí průběh výpočtu. Mezi tyto patří především signály významu „Pixel Done“ a „Row Done“ – signály jsou předávány z jednoho funkčního bloku do druhého a proto nemohou mít v rámci *FastHessianGen* stejné pojmenování – význam však zůstává stejný. Posledním významným signálem je signál *rst*, který je synchronizován zvláště v obou hodinových doménách a v doméně *clk\_core* je z důvodu snížení logického zatížení 2x zpožděn. Různé bloky v doméně *clk\_core* poté pracují s jeho různě zpožděnými verzemi podle pořadí, v jakém jimi procházejí data.

### 3.1.1 Generátor integrálního obrazu

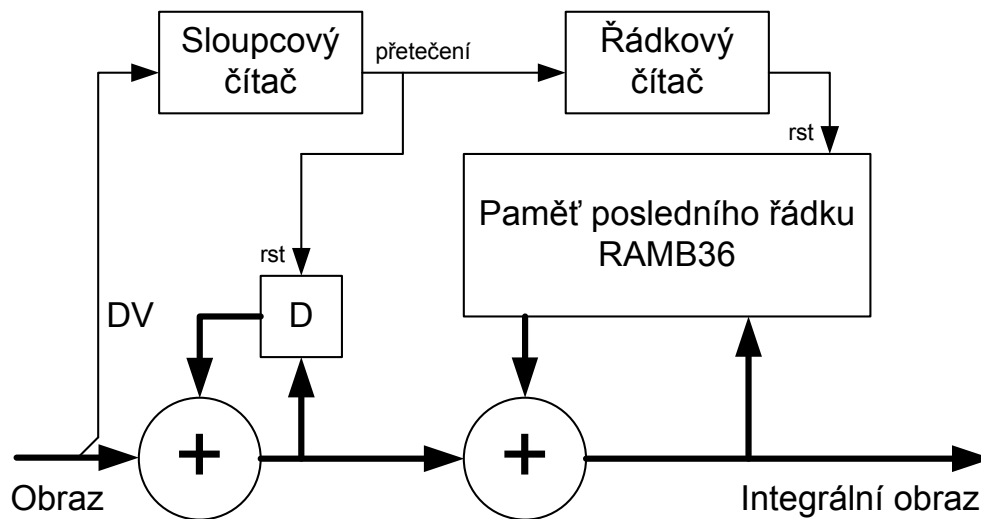
Tento blok je z celé architektury zřejmě jediný, který má zcela intuitivní implementaci. Definice výpočtu integrálního obrazu (kap. 2.1) a fakt, že obraz je přenášen po řádcích, umožňuje funkci tohoto bloku řešit s pomocí jedné blokové RAM, dvou sčítaček a dvou čítačů tak, jak ilustruje obr. 3.3. Od tohoto ideového schématu je k realizaci již pouze malý krok – je nutno doplnit vstupy a výstupy synchronizačních signálů pro řádek a snímek. Touto poslední úpravou umožníme práci bloku na obrazech proměnlivé velikosti. Posledním významným parametrem tohoto bloku jsou šířky sběrnic – uvažujeme-li vstupní obraz maximálního rozměru 1024x768 v 256 stupních šedi, je zřejmé, že vstupní sběrnice bude mít 8 bitů. S použitím funkce *log2* a uvažováním nejextrémnějšího případu (tj. všechny pixely obrazu mají hodnotu 255) snadno spočítáme, že výstupní sběrnice musí být šířky 28 bitů<sup>4</sup>.

### 3.1.2 Použití MATLAB pro optimalizaci průběhu výpočtu

Jak je řečeno v kapitole 3.1, pro výpočet jednoho bloku determinantů je podle základní definice algoritmu, při mnou použité konfiguraci, nutno přechít 576 hodnot integrálního obrazu, ze kterých se spočte 54 odezev Gaussovských filtrů. V tabulce 3.2 můžeme vidět rozložení koeficientů pro výpočet odezev filtrů v základní úrovni zvětšení. Tyto tabulky zapíšeme jako matice do MATLAB společně s dalšími podle zvoleného rozsahu oktáv a intervalů zvětšení, které sestavíme dle popisu v 2.2.2. Menší matice doplníme okolo okrajů nulami, aby měli stejný rozměr jako matice největšího filtru (52x52). Důležité je, aby pro všechny takto vygenerované matice byla pozice, ve které počítáme odezvu daného filtru (a tedy i determinant), na stejném místě. Tímto postupem vytvoříme základních 18 matic odpovídajících 6 unikátním velikostem filtrů (viz tab. 3.1). Z prvních 12 matic (odpovídajících 4 intervalům v oktávě 1, pro každý ze 3 filtrů) vytvoříme tři nové 12-ice matic, v nichž nenulové hodnoty budou posunuty postupně o sloupec doprava, o řádek

---

<sup>4</sup>Jak obrazová data, tak integrální obraz mají pouze kladné hodnoty – znaménkový bit není třeba.



Obrázek 3.3: Ideové schéma generátoru integrálního obrazu - tučné šipky znázorňují tok obrazových dat

dolů a obě posunutí dohromady. Tímto postupem jsme získali 54 matic, každou pro jeden filtr a jednu úroveň zvětšení, jejichž každý nenulový prvek symbolizuje jedno čtení hodnoty integrálního obrazu. Nyní z těchto matic můžeme snadno v MATLAB vytvořit logické matice, ve kterých hodnota prvku 1 bude symbolizovat čtení daného pixelu integrálního obrazu daným filtrem. Sečtením těchto matic získáme matici, která bude obsahovat počet čtení jednotlivých pixelů integrálního obrazu nutný k vyčíslení hodnot daného bloku determinantů. Tímto postupem zjistíme, že pro výpočet bloku determinantů je nutno provést 405 čtení různých pozic integrálního obrazu s tím, že nejvíce 18 filtrů zároveň požaduje čtení jednoho stejného pixelu (pozice [27, 27]).

Každá jednotka *TripleMAC* má připojení na všechny 4 sběrnice *sPixelBus* a v jednom hodinovém cyklu může na svém vstupu zachytit jednu hodnotu z vybrané sběrnice, která bude poté započítána (s příslušným koeficientem) do odezvy jednoho filtru. Je tedy zřejmé, že nejvyšší optimalizace dosáhneme, když v co nejvíce cyklech budou použity všechny sběrnice *sPixelBus*. Díky vygenerovaným maticím koeficientů víme, které filtry požadují čtení jakých pozic integrálního obrazu. Na základě těchto informací je třeba vytvořit adresní sekvenci čtení paměti integrálního obrazu, která bude co možná neoptimálnějším způsobem posílat data na sběrnice *sPixelBus*.

Tato sekvence musí přihlížet i k hardwarovým limitacím obvodu – jak již bylo řečeno, je třeba mít pro účely tohoto výpočtu v paměti 52 řádek integrálního obrazu – jeden řádek integrálního obrazu má v podmínkách mého řešení nejvíce zhruba 4kB<sup>5</sup>. Tato hodnota je zároveň i velikostí jednoho bloku blokové RAM v FPGA Virtex-5. Proto buffer

<sup>5</sup>Maximální délka řádky obrazu je 1024 a bitová šířka hodnoty integrálního obrazu je 28 bitů.

musí být realizován alespoň 52 těmito bloky a jejich výstup musí být multiplexován. Kdybychom požadovali možnost mít na každé *sPixelBus* libovolný pixel integrálního obrazu, museli bychom realizovat sběrnicový přepínač 52x4. Logika tohoto přepínače by byla velká a pomalá, a proto v mém řešení po každé sběrnicí mohou být čteny pouze některé řádky bufferu<sup>6</sup>. Další omezení je v provedení jednotky *TripleMAC*. Ta totiž potřebuje na zpracování jedné hodnoty s koeficientem  $\pm 1$  dva hodinové cykly a na hodnotu s koeficientem  $\pm 3$  dokonce čtyři. Každá jednotka navíc obsahuje na vstupu 2-úrovňové FIFO. Výše zmíněné hardwarové limitace úlohu tvorby optimální adresní sekvence znatelně zesložitují.

0	0	0	0	0	0	0	0	0	0
0	1	0	0	-1	-1	0	0	1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	-1	0	0	1	1	0	0	-1	0
0	-1	0	0	1	1	0	0	-1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	1	0	0	-1	-1	0	0	1	0
0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
1	0	0	-3	0	0	3	0	0	-1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
-1	0	0	3	0	0	-3	0	0	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

0	0	1	0	0	0	0	-1	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	-3	0	0	0	0	3	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	3	0	0	0	0	-3	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	-1	0	0	0	0	1	0	0

Tabulka 3.2: Příklad rozmístění koeficientů pro výpočet odezvy Gaussovských filtrů z integrálního obrazu v základním zvětšení (oktáva 1, interval 1). Odshora v pořadí  $D_{xy}$ ,  $D_{xx}$  a  $D_{yy}$ . Orámečkováná hodnota značí pozici, na které je počítán determinant.

Další důležitou úlohou řešenou s pomocí MATLAB na základě matic koeficientů je mapování filtrů do bloků *TripleMAC*. Jak již bylo zmíněno, jedna jednotka *TripleMAC* spravuje výpočet odezvy tří filtrů. Pokud se v adresní sekvenci chceme vyhnout nutnosti opakovaného čtení, je nutno mít k dispozici minimálně tolik jednotek *TripleMAC* jako je nejvyšší počet filtrů požadujících čtení stejného pixelu integrálního obrazu. Dále je od úlohy mapování požadováno, aby žádná *TripleMAC* nespravovala výpočet odezvy dvou filtrů,

<sup>6</sup>Volba příslušnosti řádek bufferu ke sběrnicím je provedena na základě spodních 2 bitů řádkové adresy.

kteřé potřebují započítat hodnotu stejného pixelu (důvod je taktěž eliminace opakovaného čtení).

Poslední úloha řešená v MATLAB je generování řídicí sekvence komponent *HessianCalc* a *LocMaxFinder*. Tyto spolu úzce souvisí a zároveň jsou poměrně triviální, protože tyto bloky zpracovávají výsledky Gaussovských filtrů, na jejichž zpracování je třeba podstatně méně cyklů než na jejich získání. Tyto sekvence pouze určují v jakém pořadí jsou zpracovávány odezvy jednotlivých filtrů a tudíž v jakém pořadí jsou generovány a vyhodnocovány determinanty. Jejich úloha bude popsána blíže v kapitolách zabývajících se funkcí příslušných bloků.

V závěru této kapitoly shrnuji řídicí a adresní sekvence, které je nutno sestavit před syntézou <sup>7</sup>:

**Adresní sekvence sběrnic *sPixelBus*** – každému cyklu sběrnic (v rámci cyklu výpočtu bloku determinantů) přiřazuje adresy čtených pixelů integrálního obrazu.

**Sekvence volby koeficientů bloků *TripleMAC*** – v každém cyklu sběrnic *sPixelBus* přiřazuje každému bloku *TripleMAC* řídicí signály určující koeficient, se kterým se má započítat hodnota na vstupu tohoto bloku.

**Sekvence volby sběrnic bloků *TripleMAC*** – v každém cyklu sběrnic *sPixelBus* přiřazuje každému bloku *TripleMAC* řídicí signály určující, která ze sběrnic *sPixelBus* se má připojit na vstup bloku.

**Sekvence volby filtru bloků *TripleMAC*** – v každém cyklu sběrnic *sPixelBus* přiřazuje každému bloku *TripleMAC* řídicí signály určující do odezvy jakého filtru (jehož výpočet je spravován daným blokem) se má započítat současná hodnota na vstupu tohoto bloku. Jednou z možných voleb je i do žádného.

**Sada sekvencí řízení výpočtu bloku *HessianCalc*** – tyto sekvence určují pořadí výčtu hodnot spočtených bloky *TripleMAC* v předchozím cyklu výpočtu bloku determinantů. Určují také přiřazení normalizačních koeficientů odezvám jednotlivých filtrů.

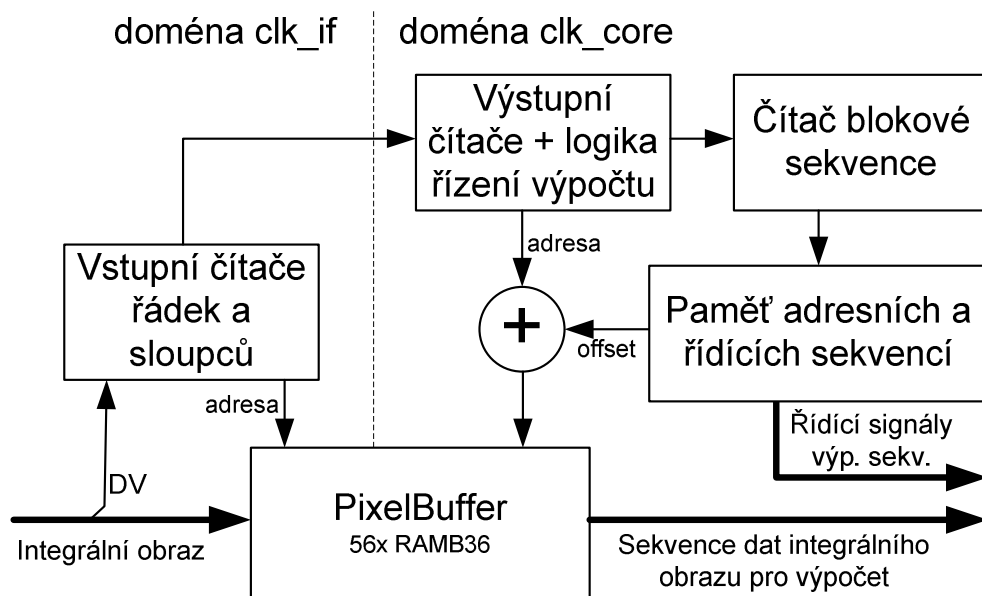
**Sada sekvencí řízení práce bloku *LocMaxFinder*** – určují, kde v toku determinantů má daný determinant své sousedy, které je nutno použít při porovnávání za účelem hledání lokálního maxima.

### 3.1.3 Blok MasterController

Tento blok realizuje řízení stěžejní části funkce celého akcelérátoru. Z ideového schématu 3.4 je vidět základní způsob funkce tohoto bloku – zjednodušeně řečeno, blok pro každý 4. pixel obrazu na výstupních sběrnicích vytvoří posloupnost hodnot integrálního obrazu a řídicích signálů, podle kterých ostatní bloky počítají odezvy Gaussovských filtrů a potažmo i determinanty. V následujících odstavcích bude princip jeho funkce popsán detailněji.

---

<sup>7</sup>Sekvence, které v předchozím textu nebyly zmíněny, jsou intuitivním způsobem generovány z dostupných dat.



Obrázek 3.4: Ideové schéma bloku *MasterController* - tučné šipky znázorňují tok obrazových dat a výstupních řídicích signálů.

Blok *PixelBuffer* slouží jako FIFO integrálního obrazu, které umožňuje nesequenční čtení bez penalizace rychlosti, které probíhá zároveň po 4 sběrnicích – zdroj dat sběrnic *sPixelBus0-3*. Taktéž umožňuje přechod mezi hodinovými doménami *clk\_if* a *clk\_core*.

Hodinová doména *clk\_if* realizuje pouze ty jednodušší funkce tohoto bloku – inkrementace zápisové adresy a řízení vstupního toku dat. Kromě obvyklých řádkových a sloupcových čítačů k adresaci zápisu se zde zapisuje i vektorový signál *sInRowCnt* signalizující počet řádek obrazu od jeho začátku. Vektorové signály *sImgLastCol* a *sImgLastRow* signalizují offset posledního sloupce a řádku, na kterém se ještě má provádět výpočet determinantů. Jejich hodnoty jsou aktualizovány na základě řádkových a sloupcových synchronizačních signálů. Vektorový signál *sImgFirstRow* ukazuje na jakém řádku bufferu začíná nový integrální obraz. V této hodinové doméně je též generován výstupní signál *oBuffLineFree*, který signalizuje alespoň jednu volnou řádku v bloku *PixelBuffer*.

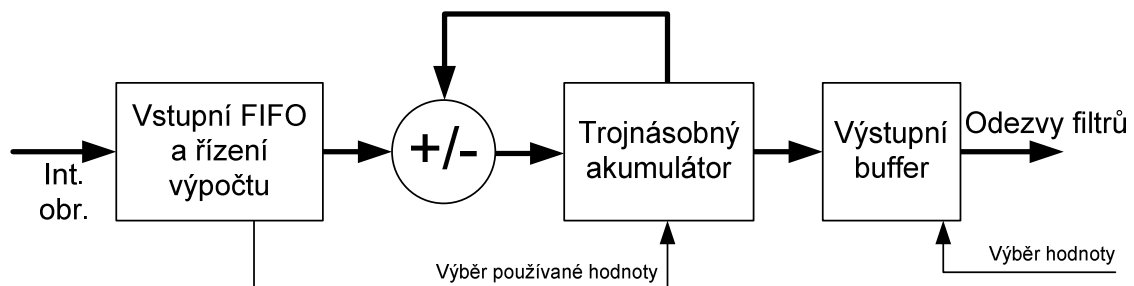
Logika řízení výpočtu a výstupní čítače v hodinové doméně *clk\_core* mají podstatně komplikovanější úlohu. Jak bylo vysvětleno na začátku kapitoly 3.1, k výpočtu determinantů na bloku 2x2 pixely je zapotřebí mít k dispozici 52x52 pixelů integrálního obrazu. V intencích bufferu řádek to znamená začínat čtení až když je v bufferu alespoň 52 řádek obrazu, čtení řádky ukončit po přečtení 52. sloupce od jejího konce, čtení obrazu ukončit (a zbytek bufferu až do začátku nového obrazu zahodit) po přečtení 52. řádky od konce obrazu. Dále díky velikosti elementárního bloku výpočtu determinantů 2x2 pixely se čtečí řádkový i sloupcový čítač (*sPixBuffRRP*, *sPixBuffCRP*) inkrementují vždy o 2. Po dosažení hodnot *sImgLastRow* a *sImgLastCol* se tyto čítače reloadují hodnotami *sImgFirstRow* a

0. Oproti vstupní straně má však tato strana o jednu úroveň iterace navíc. Vektorový signál  $sPixSeqCnt$  reprezentuje současnou pozici v sekvenci výpočtu elementárního bloku determinantů a dosažení jeho konečné hodnoty teprve znamená inkrementaci  $sPixBuffCRP$  nebo na konci řádku i  $sPixBuffRRP$ . Po dosažení konečné hodnoty se také inkrementuje  $sOutRowCnt$ , který reprezentuje počet zpracovaných řádek současného snímku. Rozdíl hodnot  $sOutRowCnt$  a  $sInRowCnt$  reprezentuje počet řádků v bufferu. V době před koncem zpracování obrazu, když se již do bufferu zapisuje nový obraz, by tato hodnota vycházela záporně, a proto se za této situace počítá odečítáním od celkového počtu řádek obrazu.

Vektorový signál  $sPixSeqCnt$  slouží především k adresaci paměti sekvence kontrolních signálů a paměti adresní sekvence. Hodnoty adresní sekvence jsou poté připočítávány k hodnotám  $sPixBuffRRP$ ,  $sPixBuffCRP$  a teprve podle těchto se adresuje čtení z *PixelBuffer*. Výpočet adresy je díky nutnosti korekce přetečení z konce na začátek bufferu nutno provádět ve více hodinových cyklech. Paměť adresní sekvence musí na jeden hodinový takt vydat adresní offsety pro všechny 4 sběrnice, stejně tak paměť kontrolní sekvence musí najednou vydat všechny řídicí signály pro všechny bloky *TripleMAC*. Uvážíme-li šířku jednoho adresního offsetu 16 bitů a 6 řídicích signálů pro jeden blok *TripleMAC*, dostaneme nutnou šířku výstupního slova paměti adresní a kontrolní sekvence 172 bitů, což mě v kombinaci s délkou této sekvence dovedlo k nutnosti použít 2 bloky RAMB36SDP a jeden RAMB18SDP, které mají oproti běžným blokovým RAM zdvojnásobenou šířku datových sběrnic.

Inicializační data paměti adresní a kontrolní sekvence musí být před spuštěním syntézy předpřipravena v několika textových souborech.

### 3.1.4 Blok TripleMAC



Obrázek 3.5: Ideové schéma bloku *TripleMAC* - tučné šipky znázorňují tok obrazových dat.

Blok *TripleMAC* je určen k výpočtu odezev tří Gaussovských filtrů. Na vstupní straně je ovládán dvěma bitovými signály  $iMACSelect$  a  $iCoefSelect$ .  $iMACSelect$  vybírá, ke které ze 3 hodnot odezev filtrů počítaných blokem vstupní data patří (nulová hodnota značí



ignorování vstupu). *iCoefSelect* volí jeden z koeficientů 1, -1, 3, -3, který se má použít k váhování hodnoty. Výstup bloku je realizován 3-stavovým budičem, který umožňuje takovýchto výstupů spojit více a číst výsledky z bloků postupně. Tento budič je ovládán signálem *iOutSelect*, jehož hodnota je interpretována stejně jako v případě *iMACSelect*, s tím rozdílem, že nulová hodnota značí vysokou impedanci výstupu. Vstupní data tohoto bloku jsou hodnoty integrálního obrazu, které mají bitovou šířku stanovenou předchozími bloky na 28 bitů. Pro zajištění imunity proti přetékání v průběhu výpočtu se interní matematické operace a mezivýsledky uchovávají v bitové šířce o 4 vyšší než jakou má vstupní integrální obraz. Blok násobičky obsažený v primitivním bloku DSP48E v FPGA Virtex-5 je schopen pracovat s vstupními daty šířky 25 a 18 bitů. Vzhledem k bitové šířce mezivýsledku (32 bitů) by tato vlastnost primitivního bloku znamenala nutnost použít pro realizaci více pospojovaných bloků DSP48E. Protože nejvyšší používaný váhovací koeficient má hodnotu 3, zvolil jsem namísto použití funkce násobení tři po sobě jdoucí sčítání/odčítání. Po každém výpočtu je nutno uložit mezivýsledek do akumulátoru a před každým výpočtem tento mezivýsledek načíst. Tento fakt způsobuje, že zpracování vstupní hodnoty s koeficientem  $\pm 1$  trvá dva hodinové cykly a s koeficientem  $\pm 3$  čtyři hodinové cykly. Při aktivním vstupním signálu *iSampleResults* dochází k přenosu mezivýsledků do výstupních bufferů a k jejich nulování.

### 3.1.5 Blok HessianCalc

Tento blok realizuje čtení hodnot spočtených bloky *TripleMAC* a výpočet determinantů dle definičního vztahu 2.4. Výpočet je spouštěn vstupním signálem *iPixSeqDone*, který vytváří blok *MasterController* po dokončení čtení adresní sekvence výpočtu elementárního bloku determinantů. Jak bylo zmíněno v kapitole 2.2.2, odezva jednotlivých filtrů musí být váhována nepřímo úměrně jejich ploše, navíc dle vztahu 2.4 je odezva filtru  $D_{yy}$  násobena koeficientem 0,9. Tento blok tedy počítá podle následujícího vztahu:

$$\mathcal{H}_{HC} = k_{xx}D_{xx}D_{yy} - k_{xy}D_{xy}^2 \quad (3.1)$$

$k_{xx}$

$k_{xy}$

Číslo oktávy	Číslo intervalu				Číslo oktávy	Číslo intervalu			
	1	2	3	4		1	2	3	4
1	4095	530	138	50	1	3316	429	111	40
2	530	50	11	3	2	429	40	9	3

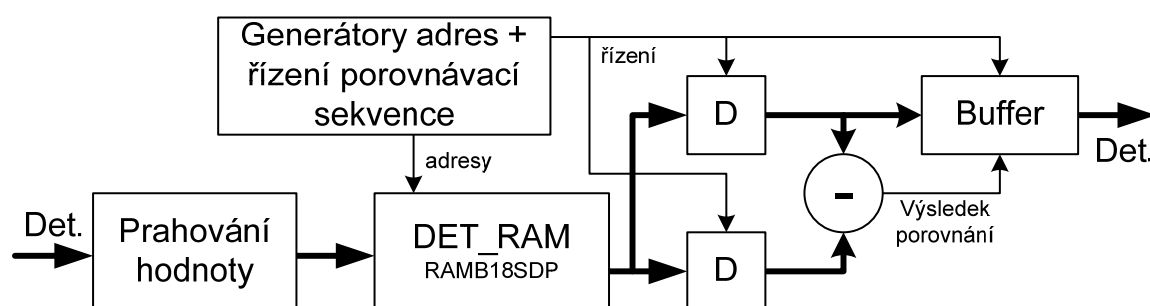
Tabulka 3.3: Hodnoty koeficientů použité pro výpočet determinantů blokem *HessianCalc*

Koeficienty  $k_{xx}$  a  $k_{xy}$  mají hodnoty dané v závislosti na úrovni zvětšení (oktávě a intervalu) dané tabulkou 3.3. Činnost tohoto bloku je řízena kontrolní sekvencí, která má délku shodnou s počtem odezev filtrů nutných k výpočtu všech determinantů v elementárním

bloku determinantů (tj. 54). Blok pracuje ještě s jednou sekvencí třetinové délky, kterou používá k volbě normalizačních koeficientů  $k_{xx}$  a  $k_{xy}$  pro současnou úroveň zvětšení počítaného determinantu.

Díky omezení bitové šířky násobiček v blocích DSP48E, kterými je realizována matematická funkce tohoto bloku, se v průběhu výpočtu musí dvakrát zmenšovat bitová šířka zpracovávané hodnoty. Toto omezení má za následek značné snížení přesnosti výpočtu, které se viditelně projevuje i ve výstupních datech. Vliv nepřesnosti zkracováním bitové šířky je sice viditelný, ale nezpůsobuje chyby tak velké, aby měli fatální vliv na výsledky práce celého algoritmu. Nicméně architektura tohoto bloku bude před případným praktickým nasazením změněna tak, aby se tato nepřesnost omezila maximálním možným způsobem.

### 3.1.6 Blok LocMaxFinder



Obrázek 3.6: Ideové schéma bloku *LocMaxFinder* - tučné šipky znázorňují tok obrazových dat.

Tento blok slouží k označování determinantů, u kterých je podezření na přítomnost lokálního maxima. Jak již bylo zmíněno několikrát, výpočet determinantů probíhá po elementárních blocích velikosti  $2 \times 2$  obrazových pixelů, které obsahují 18 hodnot determinantů na všech úrovních zvětšení. Dle popisu v kapitole 2.2.3 má proces lokalizace významného bodu 3 části – prahování, hledání lokálních maxim porovnáváním hodnot z 3-rozměrného okolí bodu a interpolace pozice. Poslední krok – tj. interpolace – ke svému výpočtu vyžaduje použití floating-point aritmetiky, a proto je přenechán zcela na softwarovém řešení. Naproti tomu prahování je triviální úkol, který je logikou velmi jednoduše implementován. Hledání lokálních maxim je sice také implementovatelné logikou, ale problém nastává v okamžiku, kdy chceme determinant porovnávat s hodnotou na sousedících řádcích – v paměti by musely být uloženy dva celé řádky elementárních bloků determinantů<sup>8</sup>. Z důvodu omezení

<sup>8</sup>Jeden řádek představuje zhruba  $34,2kB = (1024 - 50)/2 \times 18 \times 4/1024$  – význam těchto čísel v uvedeném pořadí je: počet pixelů na řádku; počet nevyužitých pixelů při maximální velikosti filtru  $52 \times 52$ ; délka hrany x elementárního bloku determinantů; počet determinantů v elementárním bloku; počet byte na jeden determinant; počet byte v kB.

paměťové náročnosti tohoto bloku jsem se rozhodl pro omezení jeho působnosti na determinanty ze dvou sousedních elementárních bloků. K úplnému porovnání by bylo třeba mít k dispozici všech 26 sousedních hodnot. Při porovnávání s determinanty pouze ze sousedních bloků na jedné řádce <sup>9</sup> z tohoto okolí chybí vždy 9 hodnot pro případ oktávy 1 a 18 pro případ oktávy 2. Tyto chybějící porovnání musí dodatečně provést software.

kód oktávy	1	1	1	1	2	2	3	3	3	3	4	4	4	4	5	5	5	5
interval	1	2	3	4	3	4	1	2	3	4	1	2	3	4	1	2	3	4

Tabulka 3.4: Pořadí determinantů v elementárním výpočetním bloku velikosti 2x2 - kód oktávy 1 a 2 odpovídá oktávám 1 a 2, kódy 3-5 odpovídají oktávě 1 s posunutou pozicí dle tabulky 3.5

1,2	3
4	5

Tabulka 3.5: Přiřazení kódů oktáv pozicím pixelů ve výpočetním bloku 2x2

Ani po omezení na adresně blízké sousedy se tento úkol příliš nezlehčí. Jak je vidět z tabulek 3.4 a 3.5, determinanty se naskládají do paměti v pořadí, které příliš nepřeje jednoduchému řešení této úlohy. Navíc v 9 z determinantů v každém bloku se lokální maximum nehledá, protože se jedná o hraniční polohy. Dále bylo nutno vzít v úvahu, že maximum nelze hledat na pozicích pixelů, které ohraničují oblast, v níž jsou determinanty počítány – k těmto totiž vždy chybí několik hodnot jejich okolí. S pomocí MATLAB byla tedy pro každý determinant v elementárním bloku vygenerována adresovací sekvence, která zajistí jeho porovnání s dostupným okolím. Stavový automat v bloku *LocMaxFinder* provádí pro každý determinant v bloku jemu příslušící sekvenci, která se navíc může předběžně ukončit v případě nalezení vyšší hodnoty než je současná porovnávaná.

## 3.2 Blok software

Jak bylo řečeno na začátku kapitoly 3, implementace některých částí algoritmu SURF byla ponechána na softwarovém řešení. Jedná se o části interpolace pozice lokálního maxima a tvorby deskriptorů. Tyto jsou reprezentovány funkcemi *SSA\_get\_SD\_pos* a *SSA\_get\_SD*. Aby softwarové zpracování nebrzdilo hardwarový výpočet, spouští se funkce pouze pro obrazové body, které byly blokem *LocMaxFinder* označeny jako lokální maximum. Způsob, kterým softwarová část algoritmu získává data z bloku hardwarového akcelérátoru, je do značné míry závislý na specifikách konečné aplikace. Řešení tohoto „interface“, pokud

<sup>9</sup>Zde se jedná fakticky o 2 obrazové řádky, protože velikost elementárního bloku výpočtu determinantů je 2x2.

chceme využít plný výpočetní potenciál hardwarového akcelerátoru, je samo o sobě komplikovaný úkol, který je mimo rámec obsahu této práce. Kód popsany v následujících odstavcích potřebuje ke své práci mít v paměti připraven vypočtený integrální obraz a determinanty.

Funkce *SSA\_get\_SD\_pos* musí před provedením vlastní interpolace polohy lokálního maxima spočítat pozici determinantu v prostoru zvětšení. Funkce má na vstupu k dispozici vždy jednu pozici (od začátku toku determinantů daného obrazu) determinantu, který byl vyhodnocen v hardware jako lokální maximum. Z tohoto offsetu je na základě znalostí velikosti obrazu a hodnot tabulek 3.4 a 3.5 vypočten řádek, sloupec, oktáva a interval daného determinantu. Poté se testuje, zda tento determinant neleží na některé z hraničních pozic, kde by k výpočtu interpolace chyběla některá část okolí. Blok *LocMaxFinder* sice neoznačuje jako lokální maxima determinanty na začátcích a koncích řádků, ale nemá informace o začátku a konci obrazu – determinanty, které se nacházejí na první a poslední řádce musí ze zpracování vyloučit software. Dalším krokem je vyčíst z paměti determinantů všechny hodnoty z okolí tohoto determinantu. K tomuto je opět nutno využít znalosti hodnot tabulek 3.4 a 3.5. Protože blok *LocMaxFinder* z důvodu paměťové limitace nemůže provést kontrolu maxima na celém okolí daného determinantu, je třeba zkontrolovat, že daný determinant je skutečně lokální maximum s přihlédnutím k celému svému okolí. Poté, co jsou hodnoty okolí předchozím krokem koncentrovány do jednoho pole, je tento úkon triviální. Nyní zbývá v souladu s 2.2.3 vypočítat na tomto okolí daného determinantu interpolaci jeho pozice. Výpočet je proveden přesně podle definice. Z důvodu dosažení co nejvyšší rychlosti jeho průběhu je proveden přechod od celočíselné aritmetiky do plovoucí řádové čárky jak nejpozději je to možno, aby nedocházelo k přetečení při celočíselném výpočtu.

Funkce *SSA\_get\_SD* má k dispozici již interpolované pozice významných bodů obrazu. V tomto okamžiku jsem však v popisu přeskočil část algoritmu popsanou v kapitole 2.3.1. Má současná implementace nepodporuje invarianci vůči rotaci a všechny deskriptory jsou tedy počítány se shodnou orientací. Originální specifikace algoritmu SURF [1] neříká jasně, kterým směrem má tato orientace směřovat. Použil jsem tedy směr použitý v implementaci [9]. Všechny deskriptory jsou počítány, jako kdyby jejich orientace směřovala v kladném směru osy x obrazu. V první části funkce *SSA\_get\_SD* je nutno ověřit, zda se významný bod nachází dostatečně daleko od okraje obrazu, aby byly k dispozici všechny hodnoty nutné k výpočtu jeho deskriptoru. Dále již nic nebrání provedení výpočtu deskriptoru tak, jak je specifikován v kapitole 2.3.2. Tato funkce je připravena pro práci i za předpokladu, že deskriptor bude mít nenulovou orientaci (tj. nepůjde o variantu algoritmu U-SURF). Pro úplné naplnění vlastností algoritmu SURF stačí doplnit funkci, která bude pro každý deskriptor počítat jeho orientaci.

# Kapitola 4

## Experimenty

### 4.1 Specifika testovací aplikace

Posledním bodem zadání mé bakalářské práce bylo mnou navrženou implementaci otestovat. Z důvodu složitosti akcelérátoru výpočtu determinantů by funkční simulace jeho běhu na jednom obrazu velikosti 1024x768 trvala velmi hrubým odhadem okolo 6000 hodin<sup>1</sup>. Navíc jedním z důležitých parametrů této implementace je, jakou rychlostí dokáže běžet v reálném FPGA. Délka trvání případné simulace a požadavek na zjištění výkonu implementace mě dovedly k nutnosti provádět experimenty na reálné aplikaci mé implementace.

Rozdělení algoritmu na část realizovanou hardwarově a softwarově vznáší na takovouto reálnou aplikaci požadavek na přítomnost rychlého CPU realizujícího softwarovou část. Proto volba konkrétního FPGA padla na XC5VFX70T. Toto FPGA je z nejvybavenější podskupiny (FXT) rodiny Virtex-5 – hlavním specifikem podskupiny FXT je široké spektrum dedikovaných funkčních hard-core bloků, mezi kterými bych zmínil především bloky CPU PowerPC-440 s APU a DDR kontrolérem, Tri-Mode ethernet MAC a PCI Express endpoint<sup>2</sup>. Jako hardwarová platforma pro testovací aplikaci byla zvolena vývojová deska ML-507<sup>3</sup> postavená na XC5VFX70TFFG1136-1C. Tato deska poskytuje všechna důležitá komunikační rozhraní, která mohou být využita v různých finálních aplikacích, a zároveň i přijatelné rozměry pro případnou budoucí testovací implementaci do mobilního robotu.

Jak je zřejmé z popisu mé implementace, její nasazení do reálné aplikace vyžaduje vyřešení mnoha pomocných problémů. Z těch, které jsou zmíněny v minulé kapitole je to především vygenerování řídicích a adresních sekvencí tak, jak je popsáno v podkapitole 3.1.2. Jak je zmíněno v kapitole 3.2, softwarová část implementace vyžaduje, aby integrální obraz a spočtené determinanty byly přítomny v paměti dostupné z procesoru,

---

<sup>1</sup>Tato hodnota byla stanovena na základě pozorování rychlosti běhu simulátoru Modelsim-XE při debugování funkce bloku výpočtu determinantů. Bohužel mám k dispozici pouze volnou verzi tohoto simulátoru, jejíž běh je výrobcem pro takto velké simulace záměrně zpomalen. Nicméně neočekávám, že plná verze by byla o tolik rychlejší, aby trvání simulace bylo únosné.

<sup>2</sup>Termíny a zkratky použité v této větě vycházejí z produktové dokumentace FGPA rodiny Vitex-5 FXT [4].

<sup>3</sup>Výrobce Xilinx, Inc.

na kterém tato běží. Z předchozí věty vyplývá nutnost vytvořit interface mezi generátorem integrálního obrazu a systémovou sběrnicí a mezi akcelerátorem výpočtu determinantů a systémovou sběrnicí po které tyto bloky budou zapisovat do paměti výsledky svých výpočtů. Dalšími pomocnými problémy jsou například způsob, jak dopravit obrazová data na vstup výpočetního akcelerátoru a jak dopravit výsledné vygenerované deskriptory do PC k provedení evaluace. Vyřešení těchto pomocných problémů, některých alespoň provizorní formou, bylo nutné k otestování funkce mé implementace. Tyto problémy, svázané s konkrétní aplikací mé implementace, jsou však mimo rámec mého zadání a proto se jim v této práci nevěnuji.

V následující tabulce shrnuji některé hlavní architektonické rysy testovací aplikace <sup>4</sup>:

Použitá architektura CPU	PowerPC-440
Frekvence základního CPU	400MHz
Frekvence FPU	200MHz
Frekvence systémové paměti	200MHz
Frekvence systémové sběrnice	100MHz
Bitová šířka systémové sběrnice	128 bitů
Komunikační rozhraní k přenosu dat	1G ethernet
Použitá BSP <sup>5</sup>	Standalone v2.00

Tabulka 4.1: Základní parametry testovací aplikace

Softwarová část aplikace musí zajišťovat řízení toku obrazových dat a jejich synchronizaci s výsledky zpracování. Problém synchronizace zpracování obrazu hardwarovou a softwarovou částí byl vyřešen implementací fronty úloh, do které ukazuje několik ukazatelů určujících v jakém stádiu zpracování ta, která úloha je. Jak vyplývá z tabulky 4.1, není v testovací aplikaci použit operační systém – pro řízení ethernetového rozhraní je použita pouze jednoduchá knihovna volně dostupného TCP/IP stacku – tato se ukázala být slabým místem celé aplikace, protože je svojí architekturou určena především pro provoz na velmi malých systémech.

## 4.2 Metoda použitá ke zpracování výsledků

Ze záznamu pohybu mobilního robota je vybrána sekvence 10-ti snímků, které byly pořízeny s odstupem cca 1m. Tyto snímky jsou postupně odesílány do testovací desky ke zpracování. Jako nejvhodnější prostředek pro analýzu takto velkého množství výsledků se ukázal být program MATLAB – vytvořil jsem tedy několik JAVA tříd, které jsou schopny řídit komunikaci s testovací deskou, odeslat do ní obrazová data a přijmout a rozřadit

<sup>4</sup>Ostatní specifiká hardware jsou dány použitou deskou – ML-507 viz. [5]

<sup>5</sup>Board Support Package – tento parametr v podstatě určuje jaký operační systém byl použit – viz [4], pozn. hodnota „Standalone“ značí absenci operačního systému a použití pouze sady knihoven usnadňujících ovládání hardware.

výsledky. Po získání výsledků máme k dispozici ke každému obrazu zhruba 120 deskriptorů<sup>6</sup>.

Pro sadu deskriptorů ke každému obrazu spočteme euklidovské vzdálenosti deskriptorů z této sady se všemi deskriptory z tří následujících obrazů. Takto získáme pro každý obraz 3 matice, jejichž prvky reprezentují euklidovské vzdálenosti deskriptorů tohoto obrazu a jednoho ze tří následujících obrazů. Každá řádka jedné z takto získaných matic reprezentuje vzdálenosti jednoho deskriptoru od všech deskriptorů jednoho ze tří následujících obrazů. Na každé řádce poté nalezneme dvě nejmenší hodnoty. Pokud je ta větší z těchto dvou hodnot větší než 1,3x menší hodnota, prohlásíme deskriptor odpovídající řádce matice a deskriptor odpovídající sloupci matice, kde se nachází menší hodnota, za shodné (tj. nalezení shodného místa – korespondence – na dvou daných obrazech). Tato relativní metoda hledání korespondencí je principiálně shodná s metodou používanou autory originální specifikace SURF. Relativní porovnávání je nutné kvůli nutnosti rozlišit situaci, kdy hledané místo z prvního obrazu na druhém již není. Zaznamenáme počet takto nalezených korespondencí. Za úspěšně nalezenou korespondenci považujeme tu, pro kterou vzdálenost pozice korespondujících deskriptorů není příliš velká – při zkoumání sekvence obrazů z pohybu robota je totiž zřejmé, že pozice jednotlivých shodných míst by se neměla měnit více než je očekáváno vzhledem ke změně pozice robota. Pro dvojici porovnávaných obrazů dáme počet úspěšně nalezených korespondencí do poměru s počtem celkově nalezených korespondencí.

Takto zjištěné poměry zatím ale příliš nevypovídají o korektnosti funkce mé implementace. Postup popsáný v minulém odstavci je tedy proveden znovu pro ty samé snímky, ale s použitím jiné implementace algoritmu SURF. Výsledky mé implementace jsou dány do poměru s výsledky této „referenční“ implementace<sup>7</sup>. Za předpokladu, že tuto jinou implementaci považujeme za správnou, by v případě správné funkce mé implementace měly tyto poměry vyjít 1. Popsaná metoda testování je velmi hrubé měřítko, které však postačuje pro prvotní zjištění správnosti mé implementace a ověření možnosti využití této implementace pro navigaci mobilního robota.

Evaluační výkon mé implementace byla provedena s použitím timeru v bloku procesoru PowerPC-440. Timer běží na kmitočtu procesoru a poskytuje tedy velmi dobré časové rozlišení. Jeho hodnota je jednoduše zaznamenána vždy při začátku/skončení zpracovávání obrazu blokem hardware a při vstupu/výstupu z funkcí *SSA\_get\_SD\_pos* a *SSA\_get\_SD*. Na základě takto zaznamenaných údajů lze zjistit dobu trvání zpracování obrazu v hardwarové části implementace a v softwarové části implementace.

---

<sup>6</sup>Počet nalezených deskriptorů je ovlivněn především nastavením prahové hodnoty v bloku *LocMaxFinder* – toto nastavení bylo provedeno experimentálně.

<sup>7</sup>Referenční implementace, respektive její výsledky pro stejnou sadu snímků, byly dodány vedoucím práce.

### 4.3 Výsledky testovací aplikace

S ohledem na jednu z možných koncových aplikací byla jako testovací data vybrány záběry z kamery umístěné na mobilním robotu pohybujícím se městským parkem. Toto venkovní prostředí je velmi nestrukturované, v průběhu pohybu robota se mění (např. díky foukání větru) a obraz z kamery je navíc zatížen chybami, které jsou typické pro malou kameru a objektiv (např. soudkovitost obrazu). Tato testovací data lze tedy považovat za reprezentativní vzorek reálného prostředí. V tabulce 4.2 jsou zachyceny poměry počtu úspěšně nalezených korespondencí mé implementace a referenční implementace pro prvních sedm obrazů sekvence a vždy následující tři obrazy. Přesný význam těchto čísel a způsob jejich výpočtu je popsán v kapitole 4.2. V ideálním případě (stejných výsledků mého a referenčního algoritmu) by tyto čísla měla být rovna jedné. Průměr poměrů ze všech deseti obrazů sekvence je 1,021.

Tabulka 4.3 shrnuje výkonnostní parametry testovací aplikace. Časy zpracování obrazu hardwarově realizovanou částí implementace jsou odděleny od časů běhu softwarové části, protože vhodnou architekturou aplikace lze dosáhnout zcela paralelního běhu těchto částí. Proto pro stanovení výkonu implementace (počet zpracovaných snímků za vteřinu) lze uvažovat pouze vyšší z těchto časů. Naproti tomu pro stanovení latence zpracování obrazu je nutno tyto časy sečíst.

Poměr úspěšnosti pro obraz	Pořadí počátečního obrazu v sekvenci, $p$						
	1	2	3	4	5	6	7
$p + 1$	0.9700	1.0389	1.1310	1.0809	0.9489	1.0000	0.8378
$p + 2$	0.9677	0.8889	0.9663	1.1515	1.1071	1.2169	0.9167
$p + 3$	0.6277	0.4765	1.1765	1.2402	0.9383	1.4359	0.9744

Tabulka 4.2: Úspěšnosti hledání stejných obrazových detailů – poměry mé implementace k referenční implementaci.

Parametry snímků	1024x768, 256 stupňů šedi
Průměrná doba běhu hardware akcelérátoru pro jeden snímek	102,37ms
Průměrná doba běhu neakcelero- vané části algoritmu pro jeden snímek	99,31ms
Klidový příkon testovací desky	7,766W
Průměrný příkon testovací desky v průběhu výpočtu	8,55W

Tabulka 4.3: Shrnutí výkonnosti testovací aplikace.





Obrázek 4.1: Sedmý obraz testovací sekvence – malé „x“ značí významný bod s kladným znaménkem stopy Hessovy matice, malé „o“ záporným, čtverečky značí oblast, ze které byl spočten deskriptor, jejich barva je inverzní k barvě písmenka značícího významný bod.

# Kapitola 5

## Závěr

V průběhu práce na tomto projektu jsem se dlouho obával, zda v této kapitole nebudu nucen vysvětlovat příčiny nezdaru svého původního záměru. Naštěstí se tak nestalo. Akcelerace části funkce algoritmu SURF logikou FPGA se ukázala být vhodnou (i když poněkud pracnou) cestou v případech, kdy nedostačuje výkon standardních CPU a kdy je použití systému s GPU nevhodné, ať už z rozměrových důvodů, nebo kvůli velkému příkonu. Současná verze implementace se i přes všechny nedostatky ukázala být svými parametry dostačující pro nasazení v aplikaci navigace mobilního robota. Pokud budou vhodné podmínky k pokračování práce na tomto projektu, chci se především zaměřit na řešení následujících témat:

- Opravy nedostatků stávající implementace – k tomuto bodu patří předně zvýšení přesnosti výpočtu bloku *HessianCalc* a dále například lepší optimalizace tvorby adresních a řídicích sekvencí bloku *MasterController*.
- Optimalizace architektury vedoucí k možnosti zvýšení pracovní frekvence výpočetního jádra akcelérátoru.
- Nalezení možností hardwarové akcelerace druhé části algoritmu – tento bod v podstatě vyžaduje tvorbu akcelérátoru vektorových operací v plovoucí řádové čárce, který by však významnou měrou přispíval i k akceleraci práce se získanými deskripty.
- Architektonické změny umožňující použití FPGA nižší třídy.

# Literatura

- [1] H. Bay, T. Tuytelaars, L. V. Gool:  
„SURF: Speeded up robust features,“ prezentováno při *9th European Conference on Computer Vision (ECCV'06)*, *Lecture Notes in Computer Science vol. 3951.*, A. Leonardis, H. Bischof, and A. Pinz, Graz, Austria: Springer-Verlag, Květen 2006, str. 404–417.
- [2] J. Bauer, N. Sünderhauf, P. Protzel:  
„COMPARING SEVERAL IMPLEMENTATIONS OF TWO RECENTLY PUBLISHED FEATURE DETECTORS,“ prezentováno při *International Conference on Intelligent and Autonomous Systems, IAV 2007*
- [3] K. Mikolajczyk, C. Schmid:  
„A Performance Evaluation of Local Descriptors,“ *IEEE Transactions on Pattern Analysis and Machine Intelligence*, svazek 27, str. 1615–1630, Říjen 2005
- [4] Xilinx, Inc. (různí autoři):  
Sada pdf katalogových listů a uživatelských manuálů FPGA rodiny Virtex-5. Volně dostupné na [www.xilinx.com](http://www.xilinx.com) vyhledáváním dle document ID: ds100, ds202, ug190, ug193, ug194, ug197, ug200
- [5] Xilinx, Inc. (různí autoři): Dokumentace testovací platformy ML-507  
schéma: [http://www.xilinx.com/support/documentation/boards\\_and\\_kits/ml50x\\_schematics.pdf](http://www.xilinx.com/support/documentation/boards_and_kits/ml50x_schematics.pdf),  
Dále vyhledání Xilinx document ID: ug347, ug349
- [6] G. N. DeSouza, A. C. Kak:  
„Vision for Mobile Robot Navigation: A Survey,“ *IEEE Transactions on Pattern Analysis and Machine Intelligence*, svazek 24, Únor 2002
- [7] T. B. Terriberry, L. M. French, J. Helmsen:  
„GPU Accelerating Speeded-Up Robust Features,“ Argon ST, Inc., 12701 Fair Lakes Circle, Fairfax, VA 22033, dodáno vedoucím práce
- [8] D. G. Lowe:  
„Object recognition from local scale-invariant features,“ *Computer Vision*, svazek 2, 20-27. Září 1999, stránky 1150-1157

- [9] Ch. Evans: „Notes on the OpenSURF Library,“  
url: <http://www.cs.bris.ac.uk/Publications/Papers/2000970.pdf>,  
url abstrakt: [http://www.cs.bris.ac.uk/Publications/pub\\_master.jsp?id=2000970](http://www.cs.bris.ac.uk/Publications/pub_master.jsp?id=2000970),  
Leden 2009
- [10] J. Borenstein and Y. Koren:  
„Real-Time Obstacle Avoidance for Fast Mobile Robots,“ *IEEE Trans. Systems, Man, and Cybernetics*, svazek 19, číslo 5, strany 1179-1187, 1989
- [11] J. Santos-Victor, G. Sandini, F. Curotto, S. Garibaldi:  
„Divergent Stereo for Robot Navigation: Learning from Bees,“ *Proc. IEEE CS Conf. Computer Vision and Pattern Recognition*, 1993
- [12] Y. Matsumoto, M. Inaba, H. Inoue:  
„Visual Navigation Using View-Sequenced Route Representation,“ *Proc. IEEE Int'l Conf. Robotics and Automation*, svazek 1, strany 83-88, Duben 1996
- [13] D. Kim, R. Nevatia:  
„Symbolic Navigation with a Generic Map,“ *Proc. IEEE Workshop Vision for Robots*, strany 136-145, Srpen 1995
- [14] D.A. Pomerleau:  
„ALVINN: An Autonomous Land Vehicle in a Neural Network,“ *Technical Report CMU-CS-89-107*, Carnegie Mellon Univ., 1989
- [15] D.A. Pomerleau:  
„Efficient Training of Artificial Neural Networks for Autonomous Navigation,“ *Neural Computation*, svazek 3, strany 88-97, 1991

# Obsah CD

Přiložené CD obsahuje zdrojové kódy hardwarové i softwarové části implementace, text diplomové práce ve formátu PDF a zdrojové kódy celého textu práce pro systém L<sup>A</sup>T<sub>E</sub>X. Zdrojové kódy implementace jsou zahrnuty především pro lepší pochopení její funkce. V následující tabulce je popsána struktura přiloženého CD.

Adresář	Popis
<code>src_hw</code>	zdrojové kódy hardwarové části implementace
<code>src_sw</code>	zdrojové kódy softwarové části implementace
<code>doc</code>	zdrojové kódy textu bakalářské práce
<code>thesis.pdf</code>	text bakalářské práce