Diplomová práce

České
vysoké
učení technické
v Praze

F3 **Fakulta elektrotechnická**
Katedra telekomunikační techniky

Metody zpracování obrazu pro měření a třídění bižuterních kamenů

Bc. Maria Nasyrova

Vedoucí: Ing. Stanislav Vítek, Ph.D.
Obor: Sítě elektronických komunikací
Studijní program: Komunikace, multimédia a elektronika
Květen 2016

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra telekomunikační techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Maria Nasyrova**

Studijní program: Komunikace, multimédia a elektronika
Obor: Sítě elektronických komunikací

Název tématu: **Metody zpracování obrazu pro měření a třídění bižuterních kamenů**

Pokyny pro vypracování:

Cílem diplomové práce je studium a aplikace metod zpracování obrazové informace, se zaměřením na hodnocení rozměrů, tvarů a kvality bižuterních kamenů. Předpokládá se aplikování metod rozpoznávání geometrických tvarů kamenů a návrh optické soustavy. Hlavní důraz bude kladen na optimalizaci algoritmů (hledání hran, bodů, významných vad apod.) s ohledem na rychlost zpracování. Předpokládá se využití nástrojů prostředí Matlab a cílově v programovém prostředím LabVIEW, případně Vision Assistent.

Seznam odborné literatury:

[1] Hlaváč, V.; Šonka, M.: Počítačové vidění, GRADA Praha 1992, ISBN: 80-85424-67-3.
[2] Hlaváč, V.; Sedláček, M.: Zpracování signálů a obrazu, skripta FEL ČVUT, Praha 2007; ISBN: 978-80-01-03110-0.
[3] Šonka, M.; Hlaváč, V.; Boyle, R.: Image Processing, Analysis, and Machine Vision, Thomson Learning, Toronto 2008, ISBN 0-495-08252-X, starší vydání dostupné na http://www.icaen.uiowa.edu/~dip/LECTURE/lecture.html [on-line].
[4] Vlach, J. a kol.: Začínáme s LabVIEW, BEN Praha 2008, ISBN: 978-80-7300-245-9. Dostupné na http://shop.ben.cz/cz/121299-zaciname-s-labview.aspx [on-line].
[5] Vlach, J.: Metody zpracování obrazu pro časově náročné úlohy, disertační práce, FM TU Liberec 2012.
[6] Materiály dostupné na http://www.ni.com/czech [on-line].

Vedoucí: Ing. Stanislav Vítek, Ph.D.

Platnost zadání: do konce letního semestru 2016/2017

L.S.

prof. Ing. Boris Šimák, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 21. 12. 2015

Poděkování

Ráda bych poděkovala Ing. Stanislavu Vítkovi, Ph.D. za trpělivost a odborné vedení v průběhu práce.

Prohlášení

Prohlašuji, že jsem zadanou diplomovou práci vypracovala samostatně s přispěním vedoucího práce a používala jsem pouze literaturu v práci uvedenou. Dále prohlašuji, že nemám námitek proti půjčování nebo zveřejňování mé diplomové práce nebo její části se souhlasem katedry.

V Praze, 25. května 2016

Abstrakt

Zvyšující se požadavky na efektivitu výroby bižuterních kamenů tlačí výrobce k hledání nových metod kontroly kvality. Jednou z možností je využití metod zpracování obrazu, kterými se zabývá tato práce.

V práci je navržen algoritmus pro měření a třídění bižuterních kamenů. V teoretickém rozboru jsou analyzovány jednotlivé kroky algoritmu a stanovena kritéria pro implementaci - rychlost a přesnost vyhodnocení obrazu. Implementace algoritmu je provedena v prostředí MATLAB. Problém časové náročnosti je řešen pomocí výpočtu na grafickém procesoru NVIDIA podporující CUDA architekturu. Dílčí bloky algoritmu je možné v podobě knihovny využít i v prostředí LabVIEW.

V závěru práce je provedeno subjektivní hodnocení výsledků rozpoznávání geometrických tvarů kamenů a hodnocení rychlosti zpracování dat.

Klíčová slova: bižuterní kámen, rozpoznávání geometrických tvarů, hodnocení kvality povrchu, CUDA, CUDA v LabVIEW, CUDA v MATLAB, Cannyho detektor hran na GPU

Vedoucí: Ing. Stanislav Vítek, Ph.D.

Abstract

The increasing demand for efficiency in the fabrication of rhinestones pushes producers to look for new methods of the quality assessment. One way is to use the image processing methods. This thesis investigates it.

The algorithm for the measuring and sorting of rhinestones is suggested. The steps of the algorithm are analysed and implementation criteria are defined. It is rate processing and the evaluation accuracy. The implementation of the algorithm in MATLAB is performed. Time-consuming problems are solved by computing through the NVIDIA graphic processor which supports CUDA architecture. The particular blocks of the algorithm can be used in LabVIEW as an external library.

In addition, subjective assessments of the results are given as they pertain to the geometric shape recognition of rhinestones. Assessments of the procesing rate are also included.

Keywords: rhinestones, jewellery, recognizing geometric shapes, quality assessment of surface, CUDA, CUDA in MATLAB, CUDA in LabVIEW, Canny Edge Detection on GPU

Title translation: Image Processing Methods for Measuring and Sorting of Rhinestones

Obsah

1 Úvod **1**

2 Teoretický rozbor **3**
2.1 Vstupní data 3
2.2 Postup zpracování vstupních dat . 5
2.3 Rozbor bloků algoritmu 6
2.3.1 Filtrace obrazu konvolucí 6
2.3.2 Segmentace 12
2.3.3 Určení vlastností kamenů . . . 15

3 Implementace algoritmu rozpoznávání geometrických tvarů kamenů **20**
3.1 Implementace algoritmu v prostředí MATLAB 20
3.2 Možnosti implementace algoritmu v prostředí LabVIEW 24

4 Zrychlení zpracování dat pomocí výpočtů na grafickém procesoru **27**
4.1 Technologie paralelních výpočtů na GPU 27
4.2 Instalace NVIDIA CUDA Toolkit 29
4.3 Programovací model CUDA 29
4.4 Příklad implementace CUDA funkce v prostředí MATLAB 30
4.5 Implementace Cannyho detektoru hran pomocí výpočtů na GPU 33
4.6 Využití externích knihoven 39
4.7 Implementace CUDA funkce v prostředí LabVIEW 39

5 Výsledky **41**
5.1 Implementace uživatelského rozhraní 41
5.2 Hodnocení rozpoznávání geometrických tvarů kamenů 43
5.3 Hodnocení časové náročnosti . . . 44

6 Závěr **46**

A Instalace NVIDIA CUDA Toolkit **48**

B Implementace CUDA funkce v LabVIEW **50**

C Hodnocení výsledků určení vlastností kamenů **55**

D NVIDIA Visual Profiler **57**

E Literatura **58**

Obrázky

1.1 Kontrola kvalitativních vlastností výrobku . . . 1

2.1 Obrazová data získaná pomocí řádkové kamery . . . 3
2.2 Obrazová data získaná pomocí plošné kamery . . . 4
2.3 Výřezy obrazových dat získaných pomocí řádkové kamery . . . 4
2.4 Výřezy obrazových dat získaných pomocí plošné kamery . . . 4
2.5 Tvar bižuterního kamene [1] . . . 5
2.6 Postup zpracování dat . . . 5
2.7 Porovnání výkonností FFT a konvoluce [2] . . . 7
2.8 Vyhlazování obrazu . . . 8
2.9 Detekce hran na obrázku . . . 9
2.10 Cannyho algoritmus . . . 11
2.11 Porovnání metod detekce hran . . . 11
2.12 Metoda hledaní kružnic . . . 13
2.13 Lokalizace kamínků pomocí algoritmu hledání kružnic . . . 14
2.14 Segmentace pomocí dělení obrazu na homogenní oblasti . . . 14
2.15 Výsledky použití algoritmu K-means . . . 15
2.16 Houghova transformace . . . 16
2.17 Detekce přímek na obrázku . . . 16
2.18 Výsledek distanční transformace v případě kamene bez vad . . . 17
2.19 Výsledek distanční transformace v případě kamene s poškozšným povrchem . . . 18
2.20 Porovnávání průměrné vzdálenosti do nejbližšího nenulového bodu v případě kamene s vadami a bez vad . . . 18
2.21 Hodnocení vzdálenosti do nejbližšího nenulového bodu v okolí lemu . . . 19
2.22 Hodnocení vzdálenosti do nejbližšího nenulového bodu v okolí lemu . . . 19

3.1 Výsledky předzpracování a segmentace vstupních dat . . . 22
3.2 Detekce hran faset . . . 23
3.3 Porovnání s maskou . . . 23

4.1 Porovnání početu operací v plovoucí řádové čárce za sekundu na CPU a GPU [3] . . . 27
4.2 Porovnávání architektury CPU a GPU [3] . . . 28
4.3 SIMD architektura . . . 28
4.4 Programátorský pohled na architekturu CUDA [3] . . . 30
4.5 Výsledný vzkaz úspěšné kompilace CUDA kódu . . . 32
4.6 Ověření implementace funkce `AddVectorsCuda` . . . 33
4.7 Kompilace spustitelného souboru CUDA funkce . . . 33
4.8 Dělení vstupních dat na bloky . . . 36
4.9 Transformace směru hrany do indexů bodů na obrázku [4] . . . 37
4.10 Porovnávání výsledků detekce hran . . . 39
4.11 Implementace externí CUDA funkce v prostředí LabVIEW . . . 40
4.12 LabVIEW aplikace využívající paralelní výpočty . . . 40

5.1 Volení vstupních dat v GUI . . . 42
5.2 Hodnocení kvality detekce v GUI . . . 42
5.3 Časový profil algoritmu detekce hran na GPU . . . 44
5.4 Porovnávaní času zpracovávání dat na CPU a GPU . . . 45

A.1 Zjištění modelu GPU ve Windows 7 . . . 48
A.2 Vyvolávání `mex -setup` . . . 49
A.3 Verifikace instalace CUDA 7.5 . . . 49

B.1 Nastavení implecitního využití CUDA 7.5 . . . 51
B.2 Nastavení vlastností CUDA souboru . . . 51
B.3 Nastavení vlastností projektu . . . 51
B.4 Nastavení vlastností projektu . . . 52
B.5 Soubory vznikaící po kompilaci DLL projektu . . . 53
B.6 Nastavení vlastností bloku Call Library Function . . . 53

B.7 Nastavení vlastností bloku Call Library Function 54
B.8 Využití externí knihovny v prostředí LabVIEW 54

C.1 Kvalita vstupních obrazových dat získaných pomocí řádkové kamery 55
C.2 Kvalita vstupních obrazových dat získaných pomocí plošné kamery . . 55
C.3 Kvalita detekce hran faset při použití řádkové kamery 56
C.4 Kvalita detekce hran faset při použití plošné kamery 56

D.1 Nastavení NVIDIA Visual Profiler 57

Tabulky

2.1 Metriky distanční transformace [5] 17

3.1 Vliv parametru *threshold* na kvalitu detekce 21
3.2 Nastavení parametrů detekce . . . 22
3.3 Implementace detektoru hran . . 24
3.4 Implementace detektoru kružnic 25
3.5 Implementace detektoru kružnic pomocí MATLAB kódu 25
3.6 Implementace Houghovy transformace 25
3.7 Implementace 2D korelace 26

5.1 Hodnocení kvality vstupních dat 43
5.2 Hodnocení kvality lokalizace hran faset 43
5.3 Výsledky subjektívního hodnocení lokalizace hran faset 43
5.4 Výsledky subjektívního hodnocení lokalizace hran faset 44

Seznam zkratek

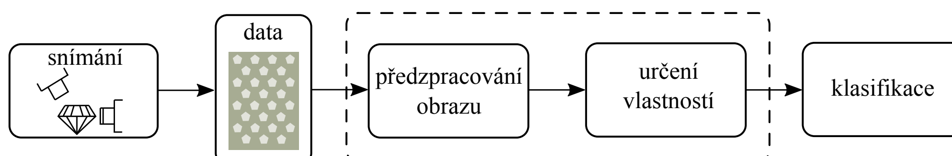
ALU	Arithmetic Logic Unit
API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DLL	Dynamic-link Library
DRAM	Dynamic Random Access Memory
FLOPs	Floating-point Operations per Second
GPGPU	General-purpose Computing for Graphics Processing Units
GPU	Graphics Processing Unit
GUI	Graphical User Interface
GUIDE	Graphical User Interface Design Environment
PNG	Portable Network Graphics
SIMD	Single Instruction Multiple Data
VI	Virtual Instrumentation

Kapitola 1

Úvod

Během procesu výroby bižuterních kamenů je nutné zajistit kontrolu kvalitativních vlastností výrobku. Při tom se mohou hodnotit různé parametry: stav povrchu kamene, jeho obrys (velikost a symetrie hran, poměr délky a šířky kamene), profil (poměr výšky a šířky kamene, poměr koruny a pavilonu, konkávnost koruny), atd. Zvyšující se požadavky na efektivitu výroby bižuterních kamenů tlačí výrobce k hledání nových metod kontroly. Za tímto účelem lze využít metody zpracování obrazu. A proto vzniká tato diplomová práce, která se zabývá metodami měření a třídění bižuterních kamenů, jež hrají významnou roli v návrhu aplikace zajišťující kontrolu kvality.

Celý proces posuzování kvalitativních vlastností výrobků lze rozdělit na několik kroků (viz obrázek 1.1). Nejprve je nutno zajistit sbíraní obrazových dat vhodných k následujícímu zpracovávání. Pak se provádí předzpracování získaných dat s cílem zvýraznit určité vlastností obrazu. A na základě těchto dat se provádí určení vlastností kamenů a jejich klasifikace.



Obrázek 1.1: Kontrola kvalitativních vlastností výrobku

Jednou z prací věnovaných metodám zpracování obrazových dat pro zajištění kontroly kvalitativních vlastností bižuterních kamenů je disertační práce Ing. Jaroslava Vlacha [1]. V ní je velká pozornost věnována rozboru možností aplikace nástrojů fuzzy logiky, fuzzy transformace a obrazové fúze jako alternativy pro kvalitativní hodnocení vlastností bižuterních výrobků. Metodami snímání skleněných kamenů pro jejich následné měření a třídění se zabývala společnost ATEsystem [6]. Jinak v oblasti bižuterního průmyslu není velké množství dostupných zdrojů z důvodu ochrany obchodního tajemství.

V průběhu řešení diplomové práce jsem se po dohodě s vedoucím rozhodla více zaměřit na implementaci algoritmů zpracování obrazu na platformě CUDA a návrhem optické soustavy se nezabývat. Byly stanoveny dva dílčí cíle. První z nich je vyzkoušení různých metod zpracování obrazu v prostředí MATLAB a stanovení kriterií pro implementaci algoritmu rozpoznávání

geometrických tvarů. Druhým cílem je řešení problému časové náročnosti s tím, aby dílčí bloky algoritmu bylo možné v podobě knihovny využívat v prostředí LabVIEW.

Kapitola 2 je teoretickým úvodem do navržené problematiky. Zde jsou probrány postup zpracování obrazových dat a metody realizace jednotlivých kroků vhodných k řešení úlohy.

Kapitola 3 je věnovaná možnostem implementace navrženého postupu v prostředí MATLAB a LabVIEW.

V kapitole 4 se řeší problematika časové náročností zpracování dat. Zde je popsána technologie paralelních výpočtů na GPU a možností využití grafického procesoru NVIDIA podporujícího CUDA architekturu k implementaci Cannyho detektoru hran.

Kapitola 5 je věnována hodnocení časové náročnosti navrženého algoritmu a subjektivnímu testování výsledků rozpoznávání geometrického tvaru kamenů

Kapitola 6 shrnuje dosažené výsledky a uvádějí se další možné postupy vývoje tohoto tématu.

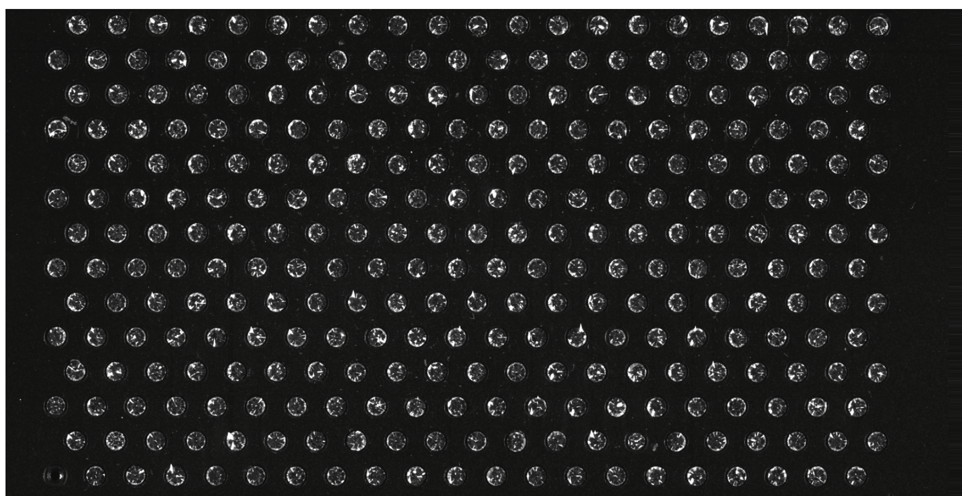
Kapitola 2

Teoretický rozbor

Tato kapitola je teoretickým úvodem do problematiky zpracování obrazových dat během procesu kontroly kvality výroby bižuterních kamenů. Je v ní proveden rozbor vstupních dat, návrh algoritmu jejich zpracování a také jsou probrány metody realizace jednotlivých kroků tohoto algoritmu.

2.1 Vstupní data

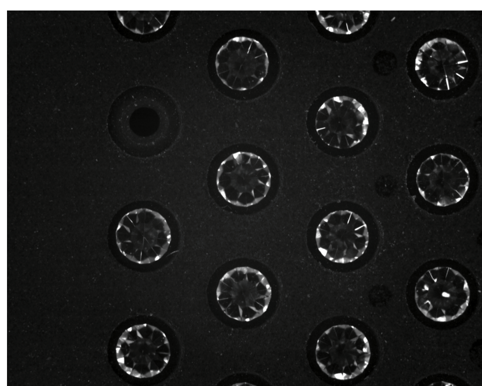
Na obrázku 2.1 a 2.2 jsou obrazová data[1], která byla určena k zpracování. Jsou získána pomocí dvou typů kamer. V prvním případě se používala řádková kamera, v druhém případě plošná kamera Basler acA2500-60um, jejíž charakteristiky lze najít v [7]. Porovnání vlastností těchto typů kamer z pohledu snímání skleněných kamenů lze najít v [6].



Obrázek 2.1: Obrazová data získaná pomocí řádkové kamery

Poskytnutá data jsou ve formátu PNG [8], jenž je určen pro bezeztrátovou kompresi rastrové grafiky. Barevná hloubka je 8 bitů, a v tomto případě se používá 256 odstínů šedi. Obrazová data získaná řádkovou kamerou mají

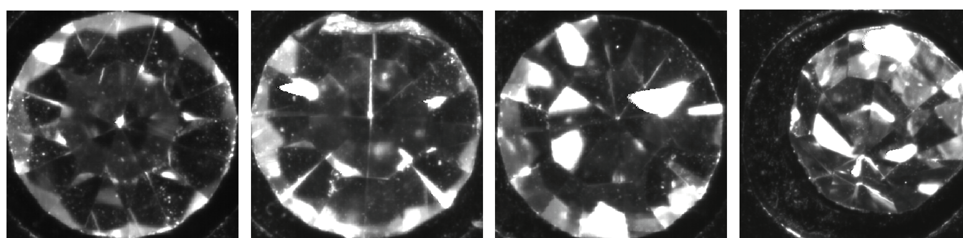
[1] Vstupní data jsou poskytnuta firmou PRECIOSA



Obrázek 2.2: Obrazová data získaná pomocí plošné kamery

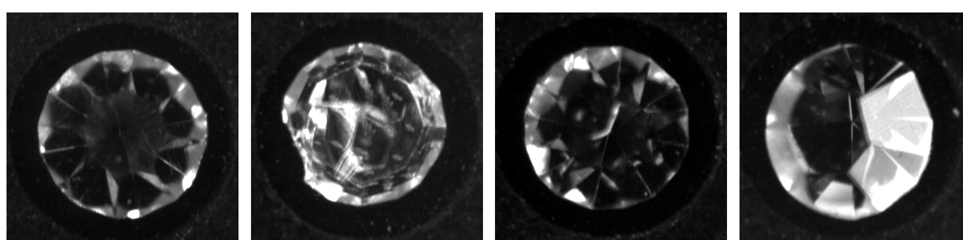
rozlišení 8160x16000 pixelů, a z toho vyplývá, že v jakékoli aplikaci zpracování těchto obrázků je nezbytně nutné klást velký důraz na rychlost zpracování dat. Rozlišení obrazových dat získaných plošnou kamerou je 2590x2048.

Vstupní data jsou expozicí velkého množství skleněných kamenů na černém pozadí. V obraze jsou zachyceny jak objekty bez patrných vad (viz obrázek 2.3 a, 2.4 a), tak i ty, které mají vady způsobené mechanickým poškozením (viz obrázek 2.3 b, 2.4 b). Také se zde objevují objekty, které nejsou vhodné k zpracování. Nevhodnost může být způsobena velkým množstvím nežádoucích odrazů vznikajících nehomogenitou osvětlení (viz obrázek 2.3 c, 2.4 c), nebo špatným umístěním kamínků, jejichž velikost nedovoluje přesné usazení do žlábku (viz obrázek 2.3 d, 2.4 d).



a - Bez vad | b - S vadami | c - Problém osvětlení | d - Problém umístění

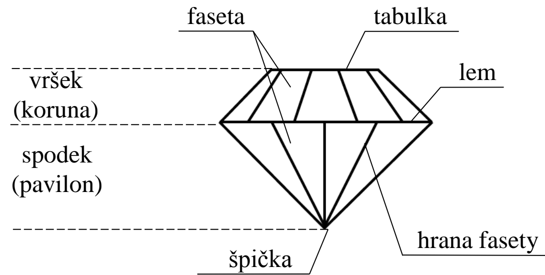
Obrázek 2.3: Výřezy obrazových dat získaných pomocí řádkové kamery



a - Bez vad | b - S vadami | c - Problém osvětlení | d - Problém umístění

Obrázek 2.4: Výřezy obrazových dat získaných pomocí plošné kamery

Je velmi důležité si upřesnit tvar zkoumaných kamínků a jejich rozměry (viz obrázek 2.5), protože tyto apriorní znalosti mohou hrát významnou roli během navrhování algoritmu.

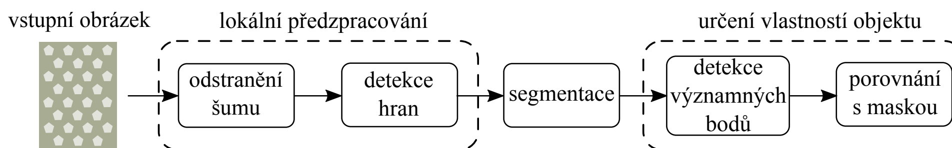


Obrázek 2.5: Tvar bižuterního kamene [1]

K významným bodům skleněného kamene patří špička, hrany faset, lem (viz obrázek 2.5), podle nichž lze soudit o tvaru objektu a jeho rozměrech na obrázku. Například obrys lemu kamene lze využit k určení rozměru a na základě polohy hran relativně ke špičce se může vyhodnotit jeho tvar.

2.2 Postup zpracování vstupních dat

Postup zpracování navržených dat je uveden na obrázku 2.6. Na vstupu algoritmu je šedotónový obrázek.



Obrázek 2.6: Postup zpracování dat

Vstupní data obsahují náhodný šum. Pro odstranění šumu se obvykle používá vyhlazovací filtr, detailní popis lze najít v [9], [10]. Operace odstranění šumu může být samostatným krokem v předzpracování dat, a nebo součástí algoritmu detekce hran (Cannyho detektor). Výsledkem vyhlazování obrazu je potlačení vyšších frekvencí obrazové funkce.

Za blokem odstranění šumu následuje blok hranového detektoru. Detekce hran je velmi důležitou součástí algoritmu zpracování obrazových dat, neboť hrany nesou často daleko více informací, než jiná místa v obraze. Algoritmů detekce hran je věnována kapitola 2.3.1, obsáhlý přehled lze nalézt například v [9].

Zpracování vstupních dat lze provádět buď globálně nebo lokálně. Globální zpracování je časově náročné, proto se často používá lokální zpracování dat, při kterém pracujeme pouze s dílčími částmi obrazu, tzv. oblastmi zájmu (ang. ROI - Region of Interest). K určení oblastí zájmu, ve kterých očekáváme výskyt hledaných objektů, se používají segmentační algoritmy [11].

U nalezených objektů provedeme lokalizaci významných bodů bižuterního kamene, zejména špičky a hrany faset (viz obrázek 2.6). Další analýzou pak určíme natočení kamene, jeho velikost a případné vady povrchu.

2.3 Rozbor bloků algoritmu

Počet metod, jež lze použít pro implementaci jednotlivých kroků navrženého algoritmu, je značný. Proto jsou v této části probrány jen ty, jichž analýza predikuje vhodnost pro zpracování navržených obrázků.

2.3.1 Filtrace obrazu konvolucí

V prvních krocích navrženého postupu (viz obrázek 2.6) se provádí lokální předzpracování vstupního obrázku.

Lokální operace předzpracování dat je filtrací [12]. Metody filtrace lze rozdělit do dvou skupin. První je určená k vyhlazování obrazu. V tomto případě jde o potlačení vyšších frekvencí obrazové funkce. Žádaným výsledkem vyhlazování je potlačení náhodného šumu. Současně dochází k potlačování ostatních náhlých změn jasové funkce, jakými jsou hrany, které nesou významnou informaci. Druhá skupina realizuje gradientní operace, které naopak vedou k zdůraznění vyšších frekvencí. Současně jsou zvýrazněny ty obrazové elementy, ve kterých se jasová funkce náhle mění. A v tomto případě žádaným výsledkem je zvýraznění hran v obraze.

Při realizaci lokálního předzpracování se obvykle jedná o konvoluční filtraci v prostorové oblasti nebo kmitočtovou masku ve spektrální oblasti.

Konvoluce je pro dvourozměrné funkce definována jako:

$$g(x,y) = (f * h)(x,y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(a,b)h(x-a,y-b)dadb \qquad (2.1)$$

V případě použití lineárních filtrů při lokálním předzpracování jde o diskrétní konvoluci tohoto obrazu s konvolučním jádrem:

$$g(i,j) = \sum_{(m,n)} \sum_{\in O} h(i-m,j-n)f(m,n) \qquad (2.2)$$

Tato operace počítá výsledný jas v bodě (i,j) jako lineární kombinaci jasů v okolí O vstupního obrazu f s váhovými koeficienty h.

Někde je vhodné provést filtraci ve spektrální oblasti. Přechod do této oblasti v případě dvourozměrné funkce lze provést pomocí Fourierovy Transformace:

$$F(u,v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y)e^{-2\pi i(xu+yv)}dxdy \qquad (2.3)$$

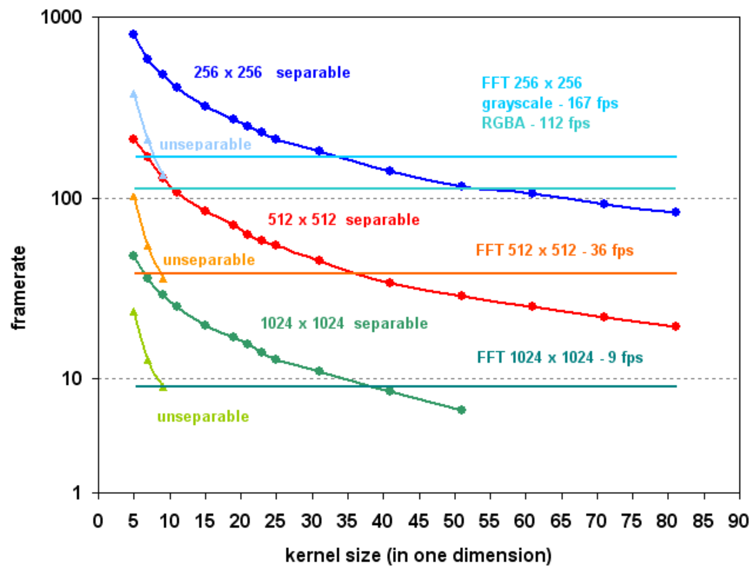
V případech zpracování obrazů se používá Diskrétní Fourierova Transformace:

$$F(k,l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n)e^{-j2\pi(\frac{km}{M}+\frac{ln}{N})} \qquad (2.4)$$

Pak filtrace ve spektrální oblasti:

$$G(u, v) = F(u, v)H(u, v) \tag{2.5}$$

Jak je popsáno v [2] a je vidět na obrázku 2.7 kvůli časové náročnosti filtraci ve spektrální oblasti má smysl provádět jen v případě velkých konvolučních jader.



Obrázek 2.7: Porovnání výkonností FFT a konvoluce [2]

Odstranění šumu

Při snímání, přenosu a zpracování obrazu může vzniknout šum, který je obvykle popsán pravděpodobnostními charakteristikami a je reprezentován různými modely [12], [10]. Idealizovaný šum, který má konstantní výkonové spektrum, se označuje jako bílý a často se používá pro simulaci nejhorších degradací obrazu. Speciálním případem bílého šumu je Gaussův šum, který má hustotu pravděpodobnosti dánu vztahem:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\dfrac{-(x-\mu)^2}{2\sigma^2}}, \tag{2.6}$$

kde μ je střední hodnota a σ je směrodatná odchylka. Gaussův šum je vhodným pro aproximaci degradace obrazu ve většině praktických případů.

Při přenosu obrazu v přenosovém kanále vzniká šum, který je obvykle nezávislý na obrazovém signálu. Takový šum má aditivní charakter a může být popsán modelem:

$$f(x, y) = g(x, y) + v(x, y), \tag{2.7}$$

kde šum v a vstupní obraz g jsou nezávislé veličiny.

V většině případů je šum signálově závislý:

$$f(x, y) = g + vg \tag{2.8}$$

Model popsaný tímto vztahem se nazývá multiplikativní šum.

Dalším druhem šumu je kvantizační šum. Ten se objevuje v případě nedostatečného počtu jasových úrovní. Impulzním šumem se nazývá šum, který vzniká v ojedinělých pixelech.

Pro účely vyhlazování šumu v obraze se používá filtrace. Volba vhodného filtru je závislá na typu šumu. V případě aditivního šumu se používají zejména lineární filtry.

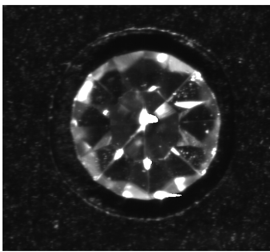
Jedním ze způsobů vyhlazování šumu je obyčejné průměrování, které filtruje obraz tím, že jako nový jas bodu přiřadí aritmetický průměr jasu bodů obdélníkového okolí. Tato filtrace se provádí pomocí diskrétní konvoluce vstupního obrazu a masky:

$$h = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.9}$$

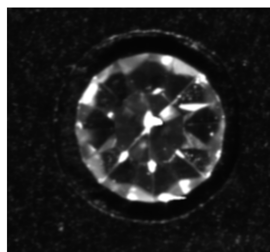
Někdy se zvětšuje váha středového bodu masky nebo jeho 4-sousedů, aby se lépe aproximovaly vlastnosti šumu s Gaussovým rozdělením:

$$h = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.10}$$

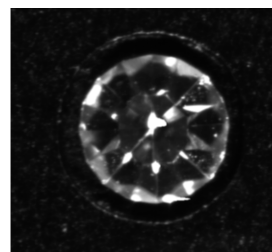
Gaussova filtrace je velice často používanou metodou, která je součástí této práce (viz Cannyho detektor hran v kapitole 2.3.1) . Na obrázku 2.8 jsou vidět výsledky vyhlazování obrazu průměrováním s pomocí Gaussovy filtrace, kde směrodatná odchylka σ je $\sqrt{2}$ a rozměr konvoluční masky je 5x5.



a - Originální obrázek



b - Průměrování

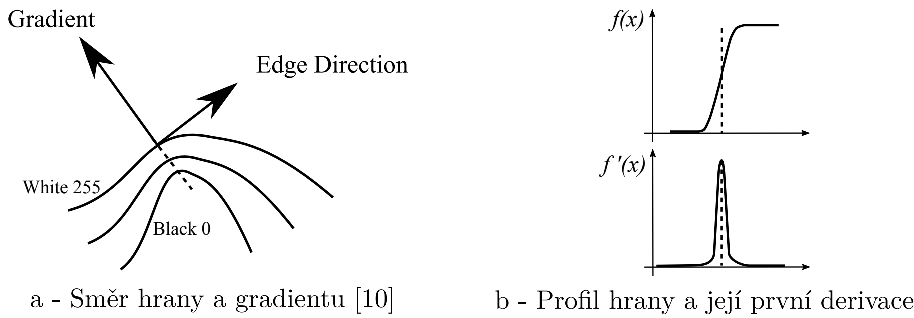


c - Gaussova filtrace

Obrázek 2.8: Vyhlazování obrazu

Detekce hran

Existuje celá řada metod určených k detekci hran, o nichž lze najít další informace v [11], [10]. V podstatě jsou všechny tyto techniky zaměřeny na detekci skokové změny obrazové funkce (viz obrázek 2.9).



a - Směr hrany a gradientu [10]   b - Profil hrany a její první derivace

Obrázek 2.9: Detekce hran na obrázku

Hrana má charakter vektoru a je určena magnitudou a směrem. Ohodnotit ji lze pomocí gradientu. Gradient funkce $f(x, y)$ je definován jako:

$$\nabla \mathbf{f} = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \\ \dfrac{\partial f}{\partial y} \end{bmatrix} \tag{2.11}$$

Magnituda gradientu $\nabla \mathbf{f}$ nese informaci o rychlostí nárůstu $f(x, y)$ na jednotku vzdálenosti ve směru $\nabla \mathbf{f}$:

$$\nabla f = |\nabla \mathbf{f}| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \tag{2.12}$$

Směr nárůstu lze určit jako:

$$\psi = arg\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \tag{2.13}$$

Výpočet gradientu lze provést pomocí konvoluce obrazu a masky, která aproximuje derivaci v různých směrech. Nejznámějšími konvolučními maskami jsou Robertsův, Prewittův a Sobelův operátor [11]. Nejjednodušším příkladem je Robertsův operátor, který používá masku 2x2:

$$h_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} h_2 = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{2.14}$$

Detekce hran pomocí tohoto operátoru je velmi rychlá, ale v případě zatížení obrazu šumem má horší výsledky, než ostatní operátory. Prewittův operátor zpracovává větší okolí zkoumaného bodu a používá masku 3x3:

$$h_1 = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} h_2 = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.15}$$

Sobelův operátor má oproti předchozím tu základní výhodu, že kromě vlastní diferenciace provádí i vyhlazování. A protože derivace obecně zvýrazňují šum,

je zmíněný efekt vyhlazování velmi užitečný:

$$h_1 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} h_2 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.16}$$

Další známe algoritmy detekce hran využívají vlastnosti druhé derivace, kterou lze zjistit pomocí Laplaceova operátoru:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{2.17}$$

Konvoluční maska Laplaceova operátoru:

$$h = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{2.18}$$

Laplaceův operátor je značně citlivý na šum a není možné detekovat směry hrany. Proto je Laplaceův operátor používán k nalezení výskytu hran s využitím průchodů nulou (zero-crossing). Tento postup je založen na konvoluci vstupního obrazu s dvourozměrnou Gaussovou funkcí:

$$h(x, y) = \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right), \tag{2.19}$$

na kterou je aplikován Laplaceův operátor, kde σ je směrodatná odchylka. Celkový výsledek je následující:

$$\nabla^2 h = \frac{r^2 - \sigma^2}{\sigma^4} \exp\left(-\frac{r^2}{2\sigma^2}\right) \tag{2.20}$$

Nejpoužívanějším algoritmem detekce hran je v současnosti Cannyho detektor [13]. Tento detektor je optimálním v případě zkreslení obrazu bílým šumem. Jeho optimalita je vztažena k třem kritériím:

1. kvalita detekce: významné hrany nesmějí být ztraceny a nemají být označeny části obrazu jako hrany, které jimi nejsou;

2. lokalizace: vzdálenost mezi skutečnou hranou a detekovanou má být minimální;

3. jednoznačnost: vyžaduje minimalizaci vícenásobné odezvy na jednu hranu. Tento požadavek je částečně pokrýt prvním kriteriem, protože jsou-li dvě odezvy na jednu hranu, jedna z nich je falešná. Ale toto kriterium zaměřen na řešení problémů vznikajících pří zkreslení šumem.

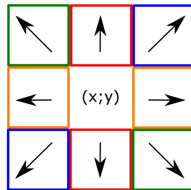
Cannyho algoritmus se skládá z následujících kroků:

1. konvoluce obrazu s Gaussovým filtrem;

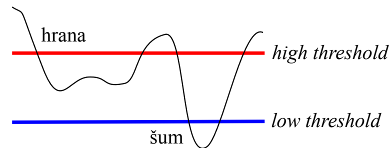
2. výpočet magnitudy a směru gradientu;

3. nalezení lokálních maxim gradientu v zavilostí na jeho směru. Pro to směr gradientu je zaokrouhlován buď do $0^0, 45^0, 90^0$ a nebo do 135^0 (viz obrázek 2.10 a). A pak hledaní lokálního maxima se provádí jen v směru orientace gradientu, buď ve vertikálním, horizontálním a nebo diagonálním;

4. prahování s hysterezí. Tady se provádí eliminace šumu pomocí dvou prahů - *high threshold* a *low threshold* (viz obrázek 2.10 b). Magnituda gradientu větší než *high threshold* znamená, že tento bod patří do hrany. Na druhou stranu magnituda gradientu menší, než *low threshold* ukazuje, že tento bod je šumem. Pokud posuzujeme bod, magnituda gradientu kterého leží mezi nastavenými prahy, pak je uznán jako bod hrany jedině pokud sousedí s bodem který už byl jako hrana označen dříve (sousedství může být 4-bodovým a 8-bodovým).



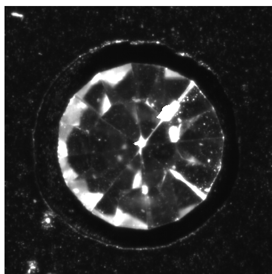
a - Možné směry gradientu



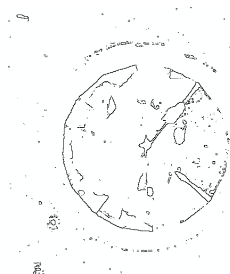
b - Prahování

Obrázek 2.10: Cannyho algoritmus

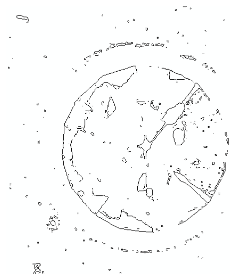
Porovnání výsledku použití různých operátorů je uvedeno na obrázku 2.11.



a - Originální obrázek

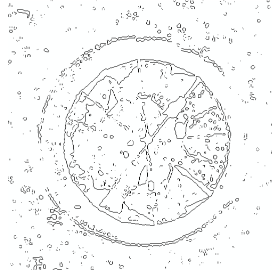


b - Robertsův operátor

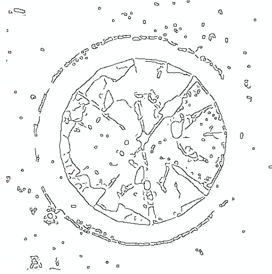


c - Prewittův operátor

d - Sobelův operátor



e - Zero-crossing



f - Cannyho detektor

Obrázek 2.11: Porovnání metod detekce hran

2.3.2 Segmentace

Segmentace obrazu je jedním z nejdůležitějších kroků vedoucích k analýze obsahu zpracovávaných obrazových dat. Obecná definice segmentace říká, že je to proces dělení obrazu do částí, které korespondují s konkrétními objekty v obraze. Jinými slovy, každému obrazovému pixelu je přiřazen index segmentu vyjadřující určitý objekt v obraze [14]. Podrobný popis metod segmentace lze najít v [9], [12], [10]. Metody segmentace lze podle přístupu k řešení této úlohy rozdělit do několika skupin [14]:

1. metody vycházející z detekce hran (edge-based) jsou orientované na detekci významných hran v obraze. Sem patří segmentační techniky, které využívají operátorů detekce hran a Houghovy transformace určené pro hledání přímek popsaných parametricky. Další technikou v této skupině je metoda aktivních kontur, která je založena na postupném tvarování kontur až k hraně objektu. Obdobný přístup využívá metoda level-set segmentace;

2. metody orientované na regiony v obraze (region-based) využívají identifikace hran, které by teoreticky měly ohraničovat regiony. Zde lze změnit metodu region growing, kde pixely, které jsou si podobné vzhledem k nějakému kriteriu, jsou seskupovány k sobě a vytváří segmentovanou oblast. Další technikou je split and merge segmentace, která je založena na quad-tree prezentaci dat, kdy podobraz je rozdělen na čtyři kvadranty pokud jsou jeho měřené atributy nehomogení;

3. statistické metody jsou založeny na statistické analýze obrazových dat, nejčastěji hodnot jasové funkce. Typickými zástupci této skupiny jsou prahovácí technika a její variace, amplitudová projekce, shluková analýza. Taky sem patří techniky využívající Kohonenovy mapy, Markovská náhodná pole a metody s fuzzy přístupen;

4. hybridní metody obsahují prvky každé z předchozích metod, taky sem patří metody založené na matematické morfologii. Jednou z takových technik je Watershed transformace. V tomto případě je obraz chápán jako térén nebo topografický reliéf, který je postupně zaplavován vodou. V místech, kde by se voda ze dvou různých povodí mohla slít jsou vytvořeny hráze. Výsledkem je obraz rozdělený do regionů, jednotlivých povodí oddělených hrázemi. Jinou technikou patřící do této skupiny je segmentace s využitím neuronových sítí.

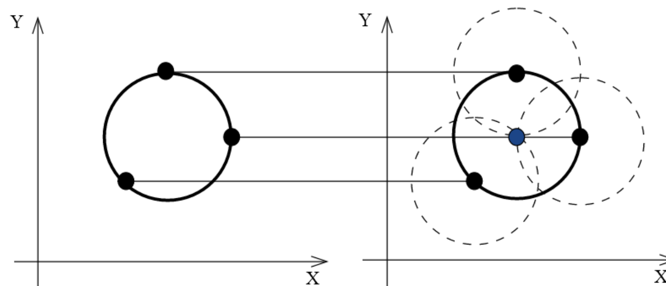
5. znalostní (knowledge-based) metody jsou založeny na znalostech vlastností segmentovaných objektů (tvar, barva, struktura, apod.). Zde lze vydělit segmentaci pomocí Active Appearance Model.

Segmentace pomocí Houghovy transformace

Počet algoritmů segmentace je velmi velký a volba záleží na konkrétní úloze. V případě, že obrázek zahrnuje objekty známého tvaru a rozměrů, segmentace může vypadat jako hledání objektů specifického tvaru. Při zkoumání

navržených obrazových dat si lze všimnout, že pohled shora bižuterního kamene je skoro kružnice. Proto k jeho lokalizaci je vhodné použit algoritmus detekce kružnic. Efektivní metodou určenou k řešení tohoto problému je Houghova transformace. Detailní popis různých variací algoritmu vyhledávaní kružnic pomocí Houghovy transformace a způsobů jejich implementace lze najít v [15], [16], [17].

Z [10] a [18] vyplývá, že metoda detekce tmavé kružnice na světlém pozadí začíná hledáním tmavých pixelů, které jsou potenciálními středy pomocných kružnic nutných k zjištění středa kružnice na obrázku (viz obrázek 2.12).



Obrázek 2.12: Metoda hledaní kružnic

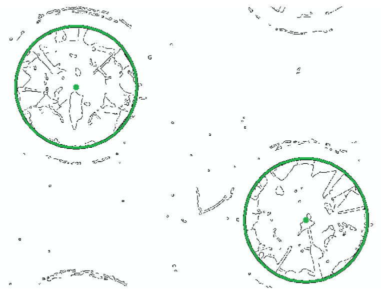
Na obrázku 2.12 je vidět, že střed kružnice je určen bodem, kterým prochází maximální počet pomocných kružnic (jsou označeny tečkovou čárou na obrázku 2.12). Ale je to případ, kde odhad poloměru kružnice je správný. V případě hledání kružnice odlišného poloměru, než je na obrázku, se vyskytuje velká množina falešně detekovaných kružnic. Proto se k zpracovávání nalezených dat používá Houghova transformace, která umožňuje transformaci nalezených kružnic v kartézských souřadnicích do jednoho bodu v parametrickém prostoru. V podstatě, Houghova transformace v případě hledání kružnic je velice podobná Houghově transformaci pro hledání přímek (viz kapitolu 2.3.3) s jediným rozdílem v tom, že se zde používá tří dimenzionální prostor, protože je kružnice v kartézských souřadnicích vyjádřena vztahem:

$$r^2 = (x - a)^2 - (y - b)^2 \tag{2.21}$$

kde (a, b) jsou souřadnice středa kružnice a r je její poloměrem.

To znamená, že se při vyhledávání kružnic pomocí Houghovy transformace definuje prostor $A(a, b, r)$. A tento prostor se dělí na políčka, na nichž se lze obrátit na základě znalostí parametrů a, b a r. A pak do políčka (a_i, b_i, r_i) se uchovává počet pomocných kružnic, které byly použity k detekci kružnice popsanou parametry (a_i, b_i, r_i). Následující lokalizace maxim v tomto prostoru dovoluje minimalizovat chybu detekce a získat středy všech kružnic na obrázku a jejích poloměry. Na základě znalosti poloměrů nalezených kružnic lze posoudit o velikostech kamenů.

Výsledky hledání kružnic pomocí Houghovy transformace jsou na obrázku 2.13.



Obrázek 2.13: Lokalizace kamínků pomocí algoritmu hledání kružnic

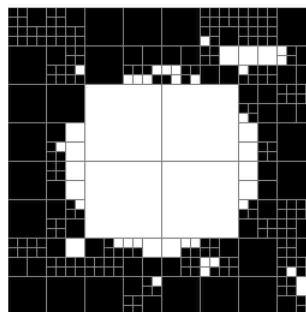
Segmentace na základě homogenity

Hledání polohy kamenů pomocí Houghovy transformace dává docela přesné výsledky, ale tento algoritmus má jednu zásadní nevýhodu, to je čas nutný na zpracování obrazu. Proto byla vyzkoušena jiná metoda segmentace navržených vstupních dat.

Z hlediska segmentace je významnou vlastností oblasti její homogenita. Proto lze rozčlenit obraz do maximální souvislých oblastí tak, aby byly z hlediska zvoleného způsobu popisu homogenní. Kriterium homogenity se může opírat o jasové vlastnosti, o komplexnější způsoby popisu, jako je například textura, nebo dokonce i o vytvářený model segmentovaného obrazu - o jeho sémantickou interpretaci.

Pro vydělení oblastí bižuterních kamenů byla provedena segmentace na základě postupného dělení obrazu na 4 kvadranty stejné velikosti a hodnocení homogenity jasové funkce. Při tom se řídilo následující logikou. V případě homogenity jednotlivého kvadrantu, celý zkoumaný kvadrant bude označen buď jako objekt nebo jako pozadí a se pokračuje v hodnocení dalšího kvadrantu. V případě nehomogenity jednotlivého zkoumaného kvadrantu probíhá jeho další dělení dokud nebude nalezen homogenní kvadrant a nebo nebude dosažena nejmenší možná velikost tohoto kvadrantu. Příklad postupného dělení obrazu na kvadranty a výsledek takové segmentace je vidět na obrázku 2.14.

a - Dělení obrazu na kvadranty

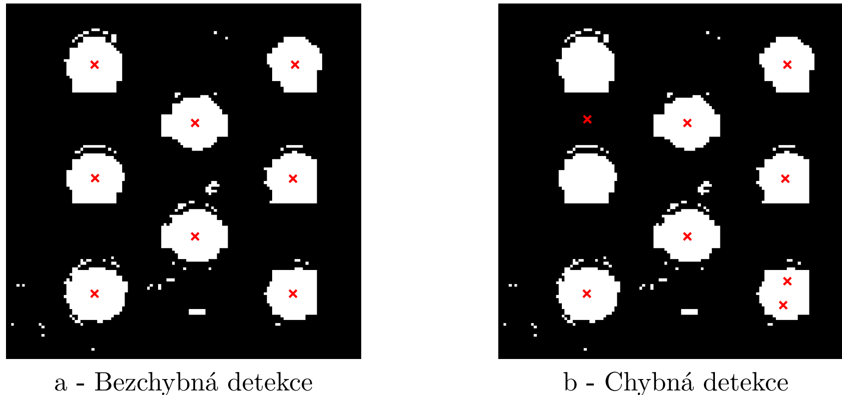


b - Výsledek segmentace

Obrázek 2.14: Segmentace pomocí dělení obrazu na homogenní oblasti

Při zpracování obrazových dat je důležité určit nejen oblasti zahrnující kamínky, ale i jejich střed. Pro tento účel lze využít algoritmus K-means. Na základě apriorní znalosti počtu kamínků na obrázku lze najít K centroid

obrazu. Ale v tomto případě se může vyskytnout velká chyba, viz obrázek 2.15.



a - Bezchybná detekce b - Chybná detekce

Obrázek 2.15: Výsledky použití algoritmu K-means

2.3.3 Určení vlastností kamenů

Na základě obrazové informace lze provést nejen hodnocení velikostí kamenů (viz kapitolu 2.3.2), ale i jeho tvaru. Za tímto účelem se používá lokalizace hran faset a další porovnávání s maskou. Také lze provést hodnocení kvality povrchu kamene a detekci patrných vad lemu.

Lokalizace hran faset

Techniky detekce hran v ideálním případě poskytují jako výstup obrazové body, které leží na hranici mezi jistými oblastmi. V praktických úlohách tato množina obrazových bodů zřídka kdy charakterizuje kompletní hranici. Proto je každý algoritmus detekce hran následován spojováním či formováním smysluplných hranic [11]. Existují široké spektrum takových metod, které lze najít v [10].

V této úloze je důležité upřesnit tvar kamene. Jednou z užitečných metod je detekce přímek na obrázku, protože na základě těchto přímek lze lokalizovat hrany faset a tuto informaci použít k klasifikaci kamene. Při hledání přímek na obrázku se obvykle používá Houghova transformace [19], [20]. Houghova transformace umožňuje transformaci přímky v kartézských souřadnicích do jednoho bodu v parametrickém prostoru. Přímka v kartézských souřadnicích může být vyjádřena následujícím vztahem:

$$\rho = x\cos\theta + y\sin\theta, \tag{2.22}$$

kde ρ je normálová vzdálenost přímky od počátku a θ je úhel mezi normálou a osou x. Houghova transformace takové uvažované přímky je bod mající souřadnice (ρ, θ).

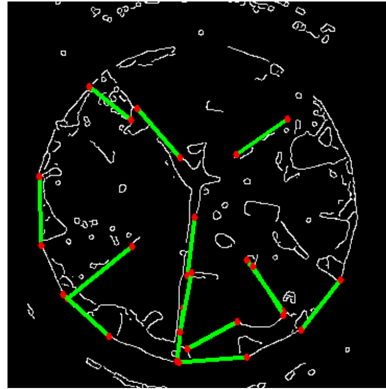
Při použití Houghovy transformace se definuje prostor $A(\rho, \theta)$. A je-li definována množina všech možných θ a ρ, pak pro každý bod obrázku detekovaný jako hrana postupnou změnou parametru θ lze spočítat ρ. A těmito dvěma

parametry se určují všechny přímky, které mohou procházet zkoumaným bodem. Inkrementace políček odpovídajících nalezeném parametrům ρ a θ dovoluje získat množinu přímek, které mají největší pravděpodobnost výskytu na obrázku (viz obrázek 2.16).

	0.5	1.0	1.5	2.0	2.5	3.0	…	... ρ
1	1	2	1	1	1	2	2	
2	3	1	1	1	1	1	1	
3	1	1	2	115	1	1	1	
…	2	1	1	1	1	2	1	
	1	2	1	1	1	2	2	
θ								

Obrázek 2.16: Houghova transformace

Výsledky detekce přímek jsou na obrázku 2.17.



Obrázek 2.17: Detekce přímek na obrázku

Výhodou použití této metody je v dané konkretní úloze to, že lze snadno rozdělit detekované přímky na ty, které patří do hran faset (jsou směrovány k středu) a na ty, které patří k lemu. A pak na základě těchto znalostí posoudit o tvaru nalezeného kamene.

Hodnocení kvality povrhu a vad lemu

Při zkoumání obrázku získaných pomocí plošné kamery byly zaznamenány zajímavé výsledky využití distanční transformaci, o níž lze najít v [21], [22]. Konverze vstupního obrazu P na výstupní obraz D_p je dána vztahem:

$$D_p(p) = \min \parallel p - q \parallel, p, q \in P \tag{2.23}$$

Jinými slovy pomocí této transformace se v každém bodu binárního obrázku (získaného například z šedotónového obrazu pomocí detekce hran) určuje vzdálenost do nejbližšího nenulového bodu. Takovým pádem, na výstupním

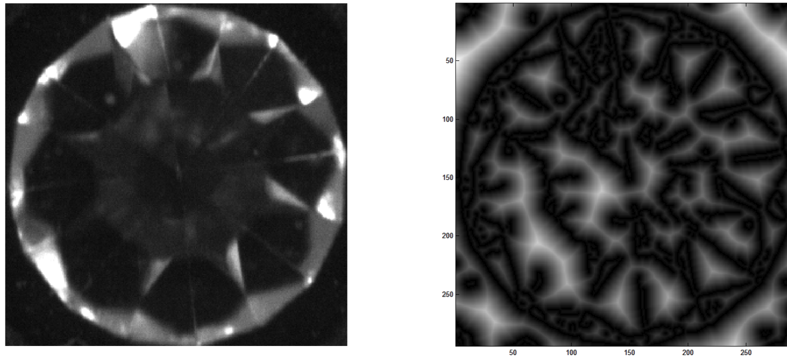
obrazu D_p je vidět tmavější oblasti, které reprezentují okolí hran, a světlejší, které ukazují na vzdálená od hran místa (viz obrázek 2.18).

Pro aproximaci vzdálenosti bodů P a Q v prostoru $\Re_n$ se používají různé metriky, jsou uvedeny v tabulce 2.1.

Metrika	Popis	Vstupní a výstupní matice
Euklidovská	geometrická vzdálenost mezi dvěma body	0 0 0 0 1 0 0 0 0 → 1.4 1.0 1.4 1.4 0 1.4 1.4 1.0 1.4
Manhattan (cityblock)	počet jednotkových kroků nutných pro přesun z výchozího do cílového bodu na základě 4-sousedství	0 0 0 0 1 0 0 0 0 → 2 1 2 1 0 1 2 1 2
Chessboard	počet jednotkových kroků nutných pro přesun z výchozího do cílového bodu na základě 8-sousedství	0 0 0 0 1 0 0 0 0 → 1 1 1 1 0 1 1 1 1

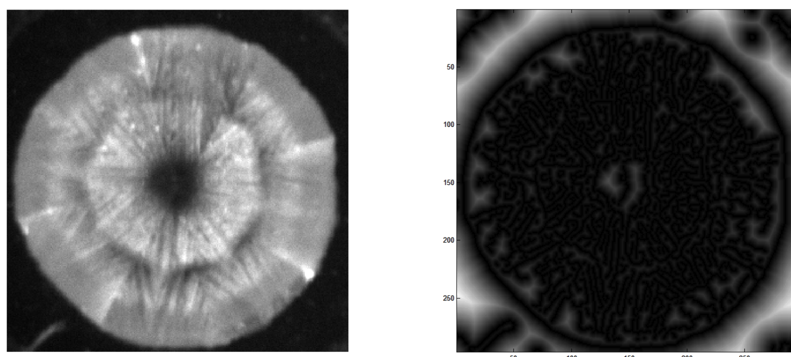
Tabulka 2.1: Metriky distanční transformace [5]

Na obrázku 2.18 je výsledek distanční transformace v Euklidovském prostoru v případě kamene bez vad. A tento se příliš liší od výsledku na obrázku 2.19 získaného v případě kamene s poškozeným povrchem.



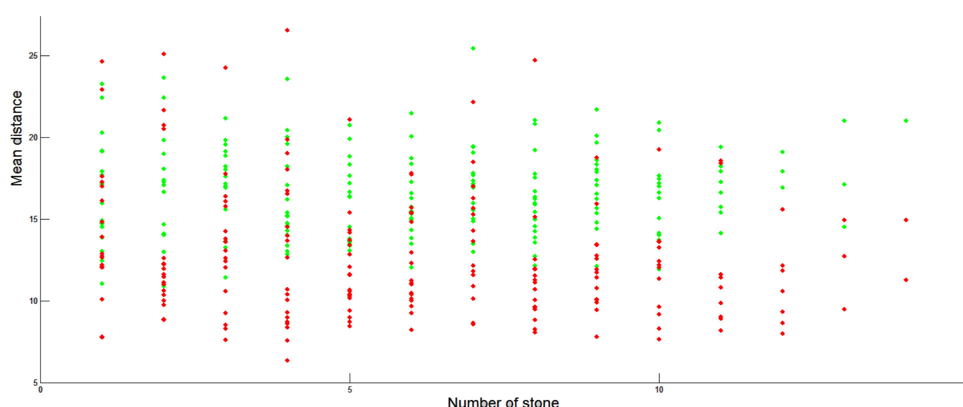
Obrázek 2.18: Výsledek distanční transformace v případě kamene bez vad

Na základě porovnávání množiny výsledků distanční transformace kamenů bez vad a s vadami byl udělán předpoklad o tom, že průměrná vzdálenost do nejbližšího nenulového bodu v případě kamene bez vad je větší, než v případě kamene s poškozeným povrchem. Tento předpoklad byl vyhodnocen na základě množiny obrázků zahrnujících kameny s vadami a bez vad. Výsledky jsou uvedeny na obrázku 2.20.



Obrázek 2.19: Výsledek distanční transformace v případě kamene s poškozšným povrchem

Zde jsou červeným označeny průměrné vzdálenosti do nejbližšího nenulového bodu na kamenech s poškozením a zeleným - na kamenech bez patrných vad. Na základě těchto dat lze přijít na závěr, že informace o průměrné vzdálenosti do nejbližšího nenulového bodu lze využít k posouzení je-li kamen s vadami povrchu nebo není.

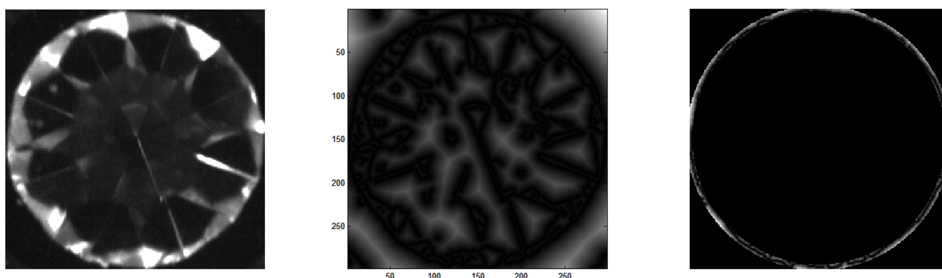


Obrázek 2.20: Porovnávání průměrné vzdálenosti do nejbližšího nenulového bodu v případě kamene s vadami a bez vad

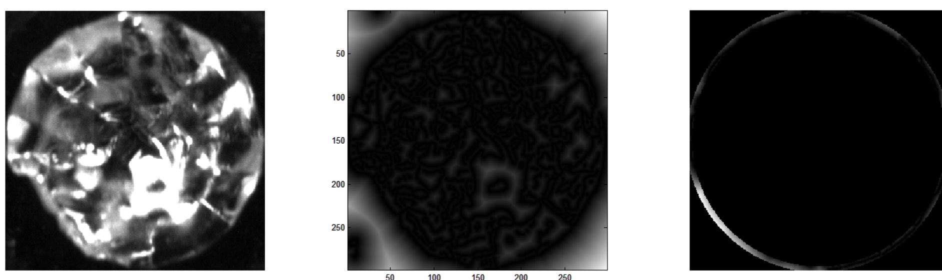
Také bylo zaznamenáno, že tuto transformaci lze využít k lokalizaci patrných vad kamene. Obrys lemu kamene bez vad je skoro kružnice. Proto v její okolí po aplikaci distanční transformace se mají objevit tmavé body (viz obrázek 2.21). V případě poškození (viz obrázek 2.22) na místech, kde se má objevit hrana, jsou body, které jsou vzdálené od detekované hrany. Což ukazuje na odchylku od normy.

Po použití masky, která nechává jen okolí lemu, na výsledný obraz distanční transformace, je vidět, že místa poškození mají mnohem větší jasovou hodnou, než ostatní místa v okolí lemu (viz obrázek 2.22).

Takovým pádem na základě hodnoty vzdálenosti do nejbližšího nenulového bodu v okolí lemu lze posoudit o patrném poškození kamene.



Obrázek 2.21: Hodnocení vzdálenosti do nejbližšího nenulového bodu v okolí lemu



Obrázek 2.22: Hodnocení vzdálenosti do nejbližšího nenulového bodu v okolí lemu

Kapitola 3

Implementace algoritmu rozpoznávání geometrických tvarů kamenů

V této kapitole je probrána implementace algoritmu rozpoznávání geometrických tvarů kamenů v prostředí MATLAB. Dále jsou uvedeny možnosti implementace jednotlivých kroků navrženého algoritmů v prostředí LabVIEW.

3.1 Implementace algoritmu v prostředí MATLAB

Na základě navrženého postupu (viz kapitolu 2.2) a rozboru jednotlivých bloků algoritmu zpracování obrazových dat (viz kapitolu 2.3), byla provedena implementace algoritmu zpracování obrazů v prostředí MATLAB. Tento algoritmus se při zpracovávání vstupních dat řídí logikou uvedenou v pseudokódu 3.1.

Prvním a velmi důležitým krokem, který určuje přesnost a kvalitu následujících kroků zpracování vstupních dat, je detekce hran. Na základě zkoumání možných metod realizace tohoto kroku byl zvolen Cannyho hranový detektor, jenž v sobě zahrnuje Gaussovou filtraci. Implementace detektoru v prostředí MATLAB má několik parametrů. Jedním z nejdůležitějších je parametr *threshold*, který určuje vysoký práh při prahování s hysterezí a pak se na základě tohoto parametru přepočítá nízký práh. Dalším parametrem je σ určující směrodatnou odchylku Gaussova filtru (viz Cannyho detektor v kapitole 2.3.1). Proto bylo provedeno zkoumaní vlivu změn těchto parametrů na kvalitu detekce přímek a kružnic na obrazových datech.

Během tohoto zkoumání byl zaregistrován velký vliv parametru *threshold*, na základě kterého se rozhoduje, jestli lze určitý bod přiřadit k hraně nebo ne. Nejprve bylo rozhodnuto prozkoumat vliv tohoto parametru na počet detekovaných přímek. Za tímto účelem se na obrazcích získaných pomocí řádkové kamery prováděla Cannyho detekce hran s měnícím se parametrem *threshold* od 0.1 do 0.6. Registroval se počet kamenů, na kterých se detekoval maximální počet čar. Ukázalo se, že nejlepších výsleků lze dosáhnout při *threshold*, jenž se rovná 0.2 a 0.3 (viz tabulku 3.1).

Podobná analýza byla provedena pro obrazová data získaná pomocí plošné kamery. A ukázalo se, že za účelem detekce hran faset nejlépe použít parametr *threshold*, který se rovná 0.04 (viz tabulku 3.1).

Dále bylo provedeno hodnocení chybné detekce kružnic minimálního polo-

Input: $I \leftarrow$ image, $imtemp \leftarrow$ mask
Result: určení vlastností detekovaných objektů: velikost, úhel otáčení
$BW \leftarrow$ Cannyho detekce hran na I
$[centers, radii] \leftarrow$ hledání kružnic na BW;
for *rr = 1:length(radii)* **do**
    $BW_area \leftarrow$ vyříznout část BW obsahující kámen;
    $[H, T, R] \leftarrow$ Houghova transformace na BW_area;
    $peaks \leftarrow$ hledání maxim v H ;
    $lines_all \leftarrow$ hledání přímek na BW_area na základě znalosti množin $T, R, peaks$;
    for *l = 1:length(lines_all)* **do**
        $lines \leftarrow$ hledání přímek, které se směrují ke středu kružnici;
    end
    for *k = 1:length(lines)* **do**
        $BW_ht \leftarrow$ nechat na obrázku jen ty části, které patří k detekovaným přímkám;
    end
    for *i = 1:360* **do**
        $imtemp_rot \leftarrow$ otáčení $imtemp$ na úhel i;
        $similar(i) \leftarrow$ korelace mezi BW_ht a $mask$;
    end
    $c \leftarrow$ maximum korelace;
end

Algoritmus 3.1: Zpracování dat v prostředí MATLAB

měru 150 a maximálního 160 pixelů na obrázcích získaných pomocí řádkové kamery a poloměrem od 145 do 155 pixelů na obrázcích získaných pomocí plošné kamery (viz tabulku 3.1).

Výsledky těchto zkoumání uvedené v tabulce 3.1 jsou dány v 95%-intervalu spolehlivosti.

	řádková kamera		plošná kamera
	threshold=0.2	*threshold*=0.3	*threshold*=0.04
Procento kamenů, na nichž se detekovalo maximum čar	$(35.7 \pm 5.5)\%$	$(37.8 \pm 5.6)\%$	$(54.5 \pm 14.9)\%$
Procento chybné detekce kružnic	$(1.6 \pm 1.2)\%$	$(2.6 \pm 1.5)\%$	$(6.1 \pm 7.15)\%$

Tabulka 3.1: Vliv parametru *threshold* na kvalitu detekce

Po aplikaci hranového detektoru na vstupní obrázek se provádí detekce kružnic s cílem určit polohu objektů a jejich velikost. Zde je velmi důležité upřesnit minimální a maximální poloměr kamenů, aby omezit zbytečné hledání zvyšující časovou náročnost algoritmu a zmenšit chybovost při lokalizaci objektu (viz segmentaci pomocí Houghovy transformace v kapitole 2.3.2).

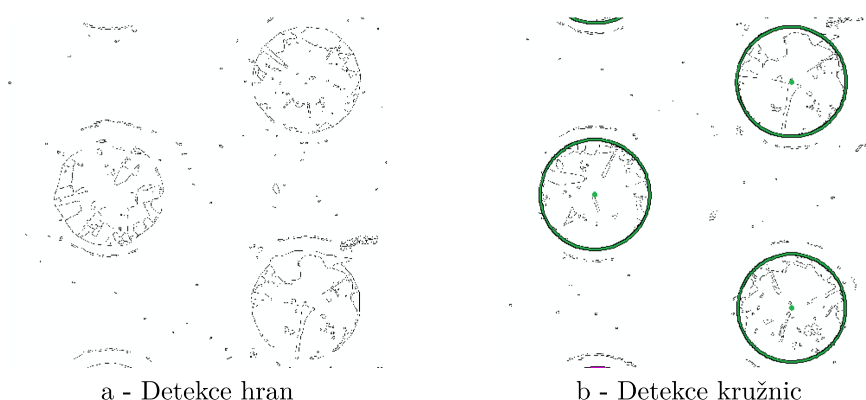
Výsledné nastavení parametrů Cannyho detektoru hran a detektoru kružnic

jsou uvedeny v tabulce 3.2.

	Detekce hran		Detekce kružnic	
	práh	směrodatná odchylka	minimální poloměr	maximální poloměr
řádková kamera	0.2	1.41	150	160
plošná kamera	0.04	1.41	145	155

Tabulka 3.2: Nastavení parametrů detekce

Výsledky aplikace detektoru hran a algoritmu hledání kružnic na vstupní data jsou uvedeny na obrázku 3.1.



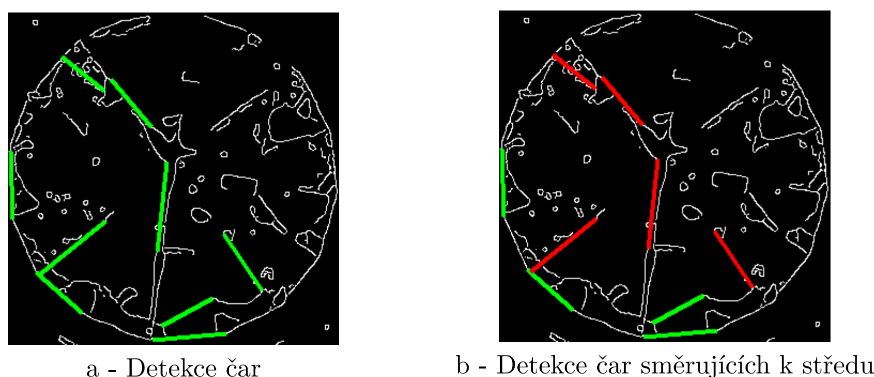
a - Detekce hran   b - Detekce kružnic

Obrázek 3.1: Výsledky předzpracování a segmentace vstupních dat

V dalších krocích na základě znalosti souřadnic středů detekovaných kružnic a jejich poloměrů lze pracovat jen se segmenty obrazu, které obsahují kameny. Nad každým z těchto segmentů se za účelem detekce přímek provádí Houghova transformace. Výsledkem je matice H, která je určena parametry T a R, kde T je množina všech možných úhlů mezi normálou k přímce a osou x, a R je množina všech možných vzdáleností této přímky od počátku (viz Houghovou transformaci v kapitole 2.3.3). Každé pole matice H zahrnuje počet přímek určených parametry T a R, které se vyskytují na obrázku. Pak se v této matici určují lokální maxima. V dalším kroku se rozhoduje na základě minimální možné délky přímky a maximální nespojitosti, jestli nalezená maxima popisují výskyty přímek, které se mohou objevit na obrázku.

Při realizaci dalšího kroku je nutné rozhodnout, které přímky jsou nejvýznamnější pro určení vlastností kamenů. Předpokládá se, že úhel natočení bude definován na základě porovnávání významných částí obrazů s maskou. Na základě znalosti tohoto úhlu budou odhadnuty polohy všech hran faset, pomocí čehož pak lze provést rozpoznávání geometrického tvaru objektu. Proto bylo provedeno zkoumání možností použití různých typů masek v závislosti na typu detekovaných hran (jsou-li hranami faset koruny, spodku, a nebo patří lemu). Nejlepší výsledky byly dosaženy při použití masky, která má tvar jedné spodní fasety.

V důsledku toho, že se nejlíp detekují hrany spodních faset bylo rozhodnuto pro další porovnávání s maskou nechat jen přímky, které se směrují k středu.



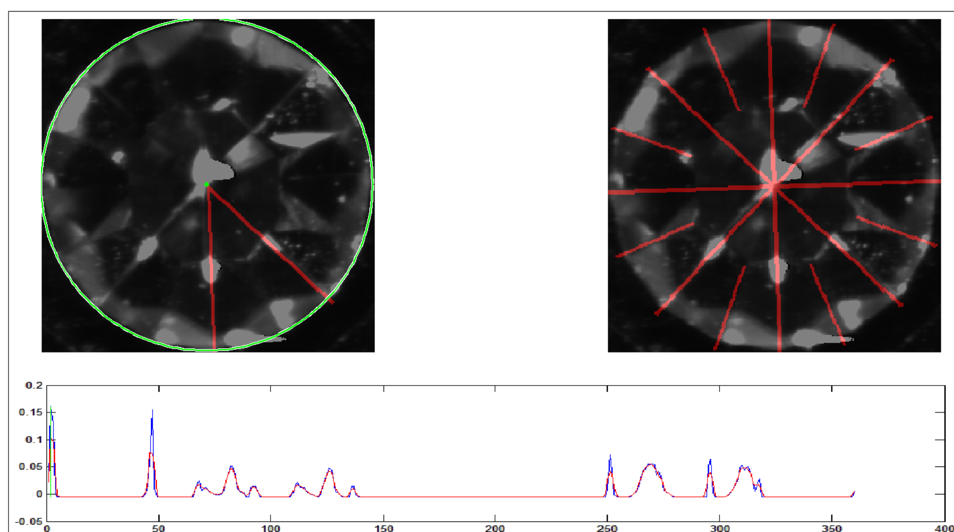
a - Detekce čar

b - Detekce čar směrujících k středu

Obrázek 3.2: Detekce hran faset

Toto rozdělení lze provést na základě znalosti rovnic každé přímky a souřadnic středu kamene. Výsledky určení přímek směrujících k středu je vidět na obrázku 3.2.

V posledním kroku probíhá určení úhlu natočení kamenu. Za tímto účelem se provádí porovnávání s maskou jen těch částí obrazu, které patří ke přímkám definovaným v předchozích krocích. A to tím, že se zjišťuje průběh korelace při otáčení masky kolem středu kamene a určuje se úhel, při kterém bude dosažena největší podobnost (viz obrázek 3.3).



Obrázek 3.3: Porovnání s maskou

Navržený algoritmus je schopen určit velikost a natočení detekovaného kamene. Tyto vlastnosti lze dále využít k lokalizaci všech hran faset a rozpoznání geometrického tvaru kamene.

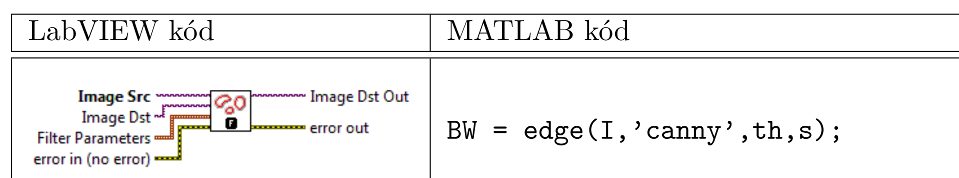
3.2 Možnosti implementace algoritmu v prostředí LabVIEW

Programové prostředí MATLAB je vhodným nástrojem při zkoumaní metodik zpracování obrazů, protože jeho knihovna Image Processing Toolbox zahrnuje velké množství funkcí, s jejichž pomocí lze rychle a snadno ohodnotit vhodnost použití nějakého algoritmu k řešení určité úlohy.

V moderní době se ve řadě průmyslových testovacích, měřicích a řídicích aplikací častěji používá vývojové prostředí LabVIEW, jež bylo vyvinuto společností National Instruments. Cílem National Instruments bylo zajistit technikům a vědcům možnost rychlého a efektivního použití měřicího a řídicího hardwaru, analýzy dat a sdílení výsledků. V prostředí LabVIEW se používá grafický programovací jazyk založený na architektuře datových toků. To znamená, že posloupnost vykonání operátorů není určena pořadím, jako v klasických programovacích jazycích, ale dostupností dat na vstupech těchto operátorů. Operátory, které mají nezávislá vstupní data se vykonávají v libovolném pořadí [23].

Z analýzy tohoto programovacího prostředí vyplývá, že LabVIEW je vhodným nástrojem pro implementaci již navrženého algoritmu. A je důležité se rozhodnout, jakým způsobem implementovat jednotlivé kroky navrženého algoritmu, protože zde lze využít interní virtuální nástroje (VI), případně realizovat nějaký nástroj interními operatory, implementovat MATLAB kód nebo využít externí DLL knihovny. Proto v této části jsou probrány možnosti implementace navrženého algoritmu v prostředí LabVIEW.

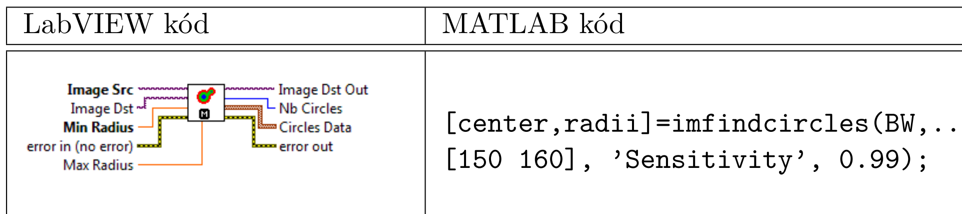
Detekci hran v prostředí LabVIEW lze provést pomocí virtuálního nástroje IMAQ EdgeDetection, případně, IMAQ CannyEdgeDetection (viz tabulku 3.3). IMAQ CannyEdgeDetection VI dovoluje nastavit různé parametry detekce (High Threshold, Low Threshold, velikost jádra Gaussova filtru a směrodatnou odchylku), měnit je během vykonávání programu a okamžitě vidět vliv těchto parametrů.

LabVIEW kód	MATLAB kód
	`BW = edge(I,'canny',th,s);`

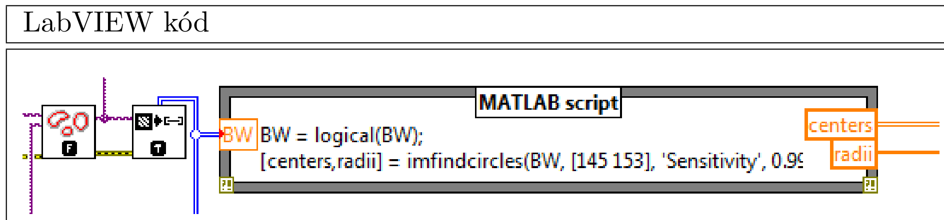
Tabulka 3.3: Implementace detektoru hran

Navržený algoritmus určuje polohu kamene hledáním kružnic. Virtuální nástroj IMAQ Find Circles dovoluje vydělit objekty okrouhlého tvaru na obrázku (viz tabulku 3.4). Ale nevýhodou tohoto nástroje je možnost jej aplikovat jen po provedení segmentace obrázku, a to s tím, že celé zkoumané objekty mají být odděleny od pozadí. A v případě nedokonalé segmentace se objevuje velké množství chybně detekovaných objektů.

Proto za účelem realizace detekce kamínků na obrázku lze využít MATLAB kód v prostředí LabVIEW (viz tabulku 3.5).

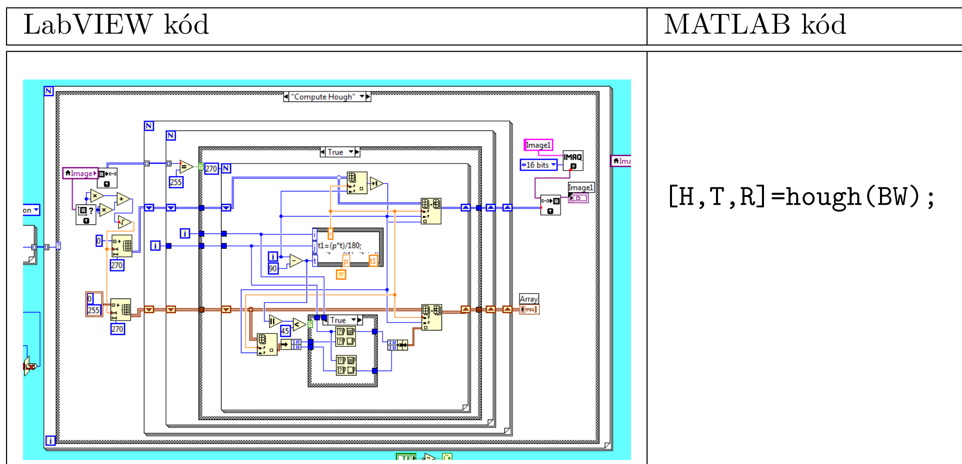
LabVIEW kód	MATLAB kód
	`[center,radii]=imfindcircles(BW,...` `[150 160], 'Sensitivity', 0.99);`

Tabulka 3.4: Implementace detektoru kružnic

LabVIEW kód


Tabulka 3.5: Implementace detektoru kružnic pomocí MATLAB kódu

Dalším krokem podle navrženého algoritmu je detekce přímek na obrázku, které lze využít k určení úhlu natočení kamínku. Prostředí LabVIEW má na to vlastní nástroj, ale ten vykazuje horší výsledky v porovnání s využitím Houghovy transformace v prostředí MATLAB. Proto realizaci tohoto kroku lze provést pomocí implementace MATLAB kódu v LabVIEW. A nebo realizovat tuto transformaci interními operátory prostředí LabVIEW (viz tabulku 3.6). Příklad této realizace byl nalezen v [24].

LabVIEW kód	MATLAB kód
	`[H,T,R]=hough(BW);`

Tabulka 3.6: Implementace Houghovy transformace

V posledním kroku navrženého algoritmu probíhá porovnávání masky a významných části obrázku a pro to lze využít virtuální nástroj IMAQ Correlate (viz tabulku 3.7).

Při realizaci algoritmů v prostředí LabVIEW je nutné brát v úvahu to, že externí zpracovávání dat se vykonává pomaleji v porovnávání s využitím interních operátorů. A proto zrychlení zpracovávání dat lze někde dosáhnout

LabVIEW kód	MATLAB kód
Optional Rectangle, Image Src, Image Template, Image Dst, error in (no error), Image Dst Out, error out	`r = corr2(A,B);`

Tabulka 3.7: Implementace 2D korelace

realizací jednotlivých kroků algoritmu interními operátory namísto využití externích knihoven nebo MATLAB kódu.

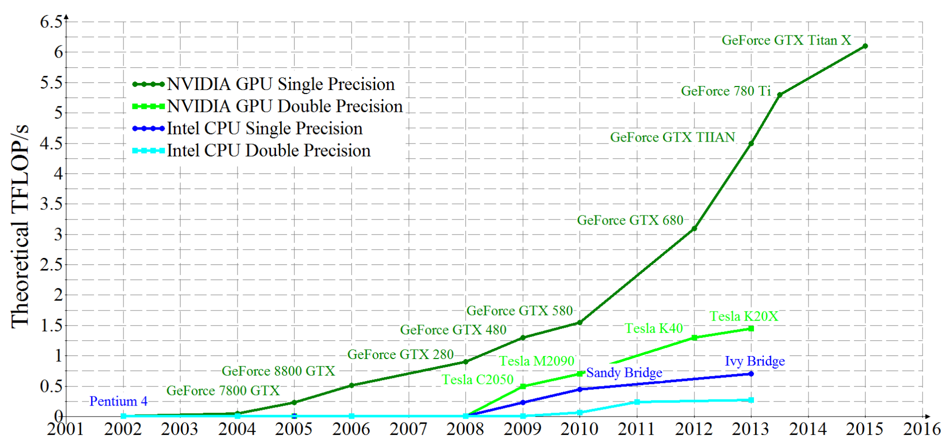
Kapitola 4

Zrychlení zpracování dat pomocí výpočtů na grafickém procesoru

Tato kapitola je úvodem do problematiky zpracování obrazových dat pomocí výpočtů na grafických procesorech (GPU), což dovoluje snížit časovou náročnost algoritmů. Největší pozornost je soustředěna na programovacím modelu CUDA, jenž byl navržen společností NVIDIA. Zde jsou uvedeny praktické nápovědy při instalaci NVIDIA CUDA Toolkit, je probrána implementace klasického příkladu CUDA programu v prostředí MATLAB, je popsána realizace detektoru hran, jež využívá výpočty na GPU, a také je popsána možnost využití výpočtů na GPU v prostředí LabVIEW.

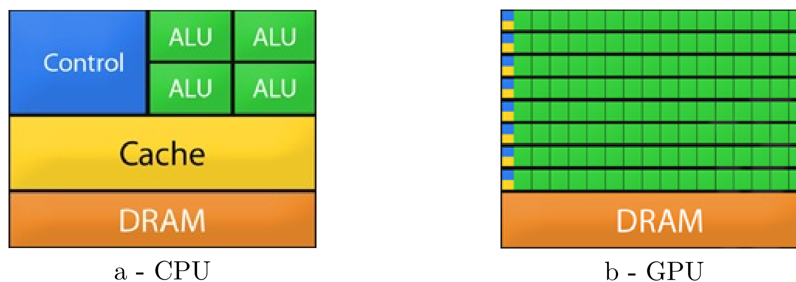
4.1 Technologie paralelních výpočtů na GPU

Kvůli rostoucím požadavkům na rychlost a objem zpracovávání dat, výpočty na grafických procesorech získávají větší význam. Grafické procesory se během několika let rozvinuly do vícevláknových, vícejádrových procesorů s paralelními výpočty, jež mají obrovský výpočetní výkon (viz obrázek 4.1) v porovnání s výkonností centrální procesorové jednotky (CPU).



Obrázek 4.1: Porovnání počtu operací v plovoucí řádové čárce za sekundu na CPU a GPU [3]

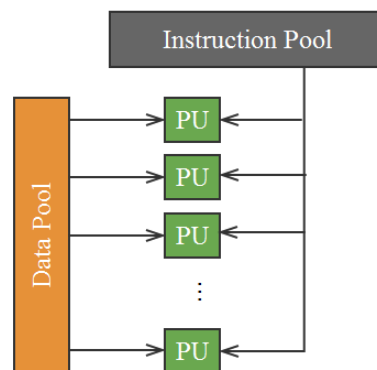
Důvodem velkého rozdílu výkonnosti CPU a GPU je jejích odlišná architektura, o tom lze najít v [3], [25]. Klasické vícejádrové procesory jsou optimalizovány pro zpracování sekvenčního kódu, obsahují tudíž složitou řídící logiku zajišťující efektivní zpracování instrukcí v ALU a přísun dat k výpočtům. Pro snížení latence přístupu do paměti je využívána složitá hierarchie pamětí včetně sdílených cache pamětí. Na rozdíl od CPU v procesu rozvoje grafického procesoru vznikla potřeba velkého množství jader na čipu. Jádra GPU mají menší výkon než jádra CPU, ale počet těchto jader na GPU mnohem větší než na CPU (viz obrázek 4.2). Například, GPU NVIDIA Tesla K20X má 2688 jader, na rozdíl od CPU Intel Xeon, jenž má do 10 jader.



a - CPU b - GPU

Obrázek 4.2: Porovnávání architektury CPU a GPU [3]

Jinými slovy, GPU se obzvlášť hodí k úlohám, které mohou být řešeny prostřednictvím paralelních výpočtů. To znamená, že se celá řada datových prvků zpracovává na základě jediného seznamu instrukcí, což se vyznačuje jako SIMD architektura (Single Instruction Multiple Data), jež je na obrázku 4.3. Výsledkem takového přístupu k zpracování dat je, za prvé, nižší požadavek na sofistikované řízení toku. A za druhé, vliv latence přístupů k pamětí je kompenzován vypočteným výkonem procesoru se SIMD architekturou.



Obrázek 4.3: SIMD architektura

Techniky všeobecného použití paralelních výpočtů (GPGPU) se vyvíjejí velmi rychle a jsou v této oblasti dva významných výrobce grafických procesorů: NVIDIA a AMD. Grafické procesory AMD podporují framework OpenCL, jenž specifikuje programovací jazyk (založený na standardu programovacího jazyka C) a rozhraní pro programování aplikací (API). A v

současné době OpenCL framework je hlavním open source frameworkem, který podporuje heterogenní výpočty na CPU a GPU [26].

Společnost NVIDIA v roce 2006 představila proprietární platformu pro všeobecné použití paralelních výpočtů CUDA. Tato platforma byla vyvinuta na základě několika zásad. První z nich je to, že CUDA nemá být samostatným programovacím jazykem, ale má poskytovat rozšíření standardních programovacích jazyků, jako C/C++, Fortran, Java, Python atd., jež umožňují přímou realizaci paralelních algoritmů. A další zásadou je nutnost podporování heterogenních výpočtů, kde aplikace používá CPU a GPU. Sériové části aplikace jsou spuštěny na CPU, a paralelní části jsou převedeny na GPU.

Další výzkum proběhl s použitím procesoru NVIDIA GeForce GT 635M podporujícím CUDA architekturu, jehož teoretická výkonnost je 290 GFLOP/s.

4.2 Instalace NVIDIA CUDA Toolkit

CUDA Toolkit je ve volném dostupu, ale jeho instalace a kompilace prvního poměrně snadného programu není jednoduchou úlohou. CUDA 7.5 podporuje různé operační systémy: Windows, Mac OS X a různé distribuce Linux. Podrobné návody k instalaci v těchto operačních systémech lze najít v [27]. Na základě osobních zkušeností, návodu dostupného na webové stránce NVIDIA [27] a v [28] v příloze A je uveden algoritmus instalace CUDA 7.5 v operačním systému Windows 7 a konfigurace prostředí MATLAB k použití výpočtů na GPU.

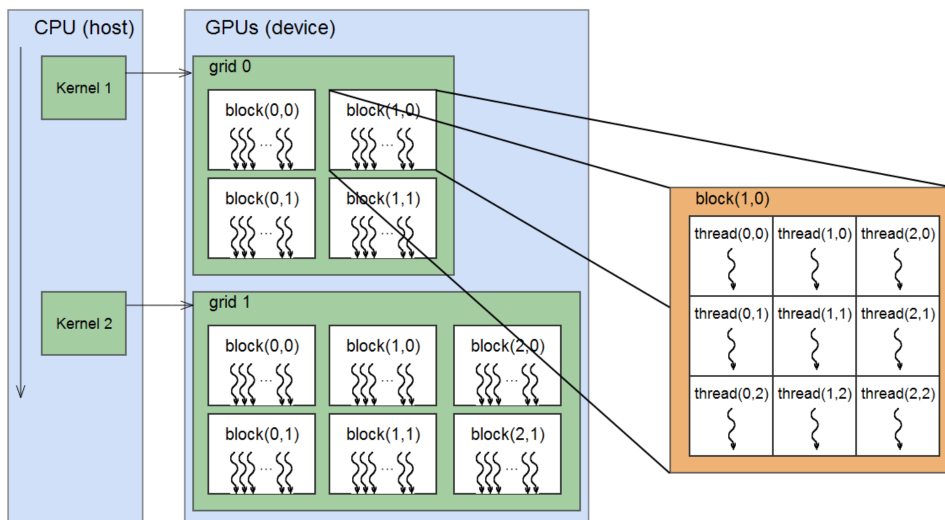
4.3 Programovací model CUDA

Realizace CUDA funkcí v prostředí MATLAB není možná bez porozumění programovacího modelu CUDA. Proto je nutno pochopit principy realizace CUDA programů, o nichž lze najít v [3], [25].

Základem programování architektury CUDA je tzv. *kernel*, který si lze představit jako část kódu, která bude spuštěna paralelně nad datovými elementy. Kernel je definován jako funkce v jazyce C/C++, spouští se na straně CPU, které je v terminologii CUDA nazýváno *hostitel* (*host*), ale provádí se na *zařízení* (*device*), což je označení pro grafický procesor.

Pro spuštění kernelu je na zařízení vytvořena množina *vláken* (*threads*), a každé vlákno zpracovává svůj datový element. Protože obvykle při použití výpočtů na GPU je velké množství vstupních datových prvků, které obvykle přesahuje počet vláken dovolených k využití, je potřeba organizovat vlákna do bloků a bloky do mřížky (viz obrázek 4.4). Konkretní konfiguraci (počet bloků a jejích velikost) si volí programátor při spouštění kernelu.

Kernely jsou spouštěny asynchronně, tj. po spuštění kernelu se na straně hostitele nečeká na dokončení výpočtu na zařízení. Místo toho se pokračuje v provádění sekvenčního kódu. Předpokládá se, že hostitel a zařízení jsou fyzicky oddělené jednotky, které mají svůj vlastní oddělený adresní prostor a díky tomu lze vlastně docílit překrývání výpočtů na hostiteli a zařízení. Tak



Obrázek 4.4: Programátorský pohled na architekturu CUDA [3]

vzniká heterogenní systém, ve kterém lze efektivně kombinovat klasické vícevláknové aplikace běžící na vícejádrových procesorech s paralelními výpočty vyžadujícími velké množství operací v plovoucí řádové čárce.

Bloky vláken musí být vzájemně nezávislé, aby je bylo možné spouštět v libovolném pořadí, ať již sériově či paralelně. Požadovaná nezávislost umožňuje naprosto libovolně rozmístit bloky na jednotlivé multiprocesory, což usnadňuje programátorům vývoj aplikací, které jsou dobře škálovatelné vzhledem k počtu multiprocesorů.

Kernely mohou pracovat pouze nad daty, která jsou umístěna v globální paměti grafické karty a proto samostatnému spuštění kernelů musí předcházet nakopírování všech dat se kterými bude kernel pracovat do zařízení. Kopírovaní dat je poměrně drahou operací a proto je vhodné data v globální paměti držet co nejdéle, bez zbytečného kopírování zpět do hlavní paměti hostitele. Stejně tak je potřeba spustit dostatečné množství vláken, aby bylo zařízení dostatečně vytížené, čímž lze velmi snadno eliminovat latence způsobené přístupy do pomalé globální paměti, jinak bude efektivita kódu velmi rychle klesat [25].

4.4 Příklad implementace CUDA funkce v prostředí MATLAB

Seznámení s implementací CUDA funkce v prostředí MATLAB lze provést na základě jednoduchého příkladu uvedeného v [28]. Zde se vytváří funkce `addVectors`, která prostřednictvím výpočtů na GPU sečte dva vektory stejné délky. To znamená, že jsou vstupními parametry této funkce dva vektory libovolné stejné délky. Na výstupu je také vektor, jehož délka se rovná délce vstupních vektorů.

V prvním kroku je nutné vytvořit složku `AddVectors.h`, která obsahuje direktivy podmíněného překladu, v prostředí MATLAB:

```
1  #ifndef __ADDVECTORS_H__
2  #define __ADDVECTORS_H__
3
4  extern void addVectors(float*A, float*B, float*C, int size);
5
6  #endif // __ADDVECTORS_H__
```

Výpis 4.1: Soubor AddVectors.h

Dále implementace funkce `addVectors` probíhá v souboru `AddVectors.cu`, kde přípona `*.cu` representuje CUDA soubor. Za tímto účelem se vytvoří nový soubor `AddVectors.cu` v prostředí MATLAB a ukládá se tam níže uvedený kód. Zde, v případě, že soubor `mex.h` není v pracovním adresáři, při použití preprocesoru `♯include` je nutno ukázat celou cestu k tomuto souboru, například:

`♯include "C:\Program Files\MATLAB\R2013b\extern\include\mex.h"`

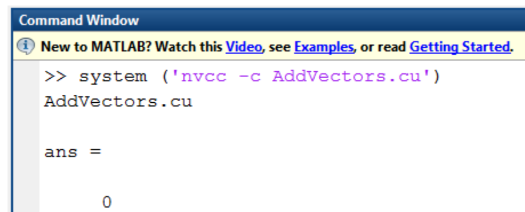
```
1  #include "AddVectors.h"
2  #include "mex.h"
3
4  __global__ void addVectorsMask (float*A, float*B, float*C, ...
       int size)
5  {
6      int i = blockIdx.x;
7      if (i≥size)
8      return;
9
10     C[i]=A[i]+B[i]; }
11
12 void addVectors(float*A, float*B, float*C, int size)
13 {
14     float*devPtrA=0, *devPtrB=0, *devPtrC=0;
15
16     cudaMalloc(&devPtrA, sizeof(float)*size);
17     cudaMalloc(&devPtrB, sizeof(float)*size);
18     cudaMalloc(&devPtrC, sizeof(float)*size);
19
20     cudaMemcpy(devPtrA, A, sizeof(float)*size, ...
           cudaMemcpyHostToDevice);
21     cudaMemcpy(devPtrB, B, sizeof(float)*size, ...
           cudaMemcpyHostToDevice);
22
23     addVectorsMask<<<size,1>>>(devPtrA, devPtrB, devPtrC, ...
           size);
24
25     cudaMemcpy(C, devPtrC, sizeof(float)*size, ...
           cudaMemcpyDeviceToHost);
26
27     cudaFree(devPtrA);
28     cudaFree(devPtrB);
29     cudaFree(devPtrC); }
```

Výpis 4.2: Soubor AddVectors.cu

V dalším kroku se provádí kompilaci CUDA kódu s cílem vytvořit objektový soubor, jenž se potřebuje pro překlad a linkování `mex` kódu. Pro to se do příkazového řádku v prostředí MATLAB píše:

```
» system('nvcc -c AddVectors.cu')
```

V případě úspěšné kompilace se objeví vzkaz jako je na obrázku 4.5.



Obrázek 4.5: Výsledný vzkaz úspěšné kompilace CUDA kódu

Dále je nutno vytvořit `mex` funkci, která bude vyvolávat CUDA funkci. Pro to v novém souboru `AddVectorsCuda.cpp` vytvářeném v prostředí MATLAB uložíme následující kód:

```
1  #include "mex.h"
2  #include "AddVectors.h"
3
4  void mexFunction(int nlhs, mxArray*plhs[], int nrhs, mxArray*...
       prhs[])
5
6  {
7       int numRowsA=(int)mxGetM(prhs[0]);
8       int numColsA=(int)mxGetN(prhs[0]);
9       int size = numRowsA*numColsA;
10
11      float*A=(float*)mxGetData(prhs[0]);
12      float*B=(float*)mxGetData(prhs[1]);
13
14      plhs[0]=mxCreateNumericMatrix(numRowsA, numColsA, ...
            mxSINGLE_CLASS, mxREAL);
15      float*C=(float*)mxGetData(plhs[0]);
16
17      addVectors(A,B,C,size);
18 }
```

Výpis 4.3: Soubor AddVectorsCuda.cpp

V posledním kroku se kompiluje `mex` soubor pomocí příkazu:

```
» mex AddVectorsCuda.cpp AddVectors.obj -lcudart -L"C:\ Program
Files\ NVIDIA GPU Computing Toolkit \ CUDA \ v7.5 \ lib \ x64"
```

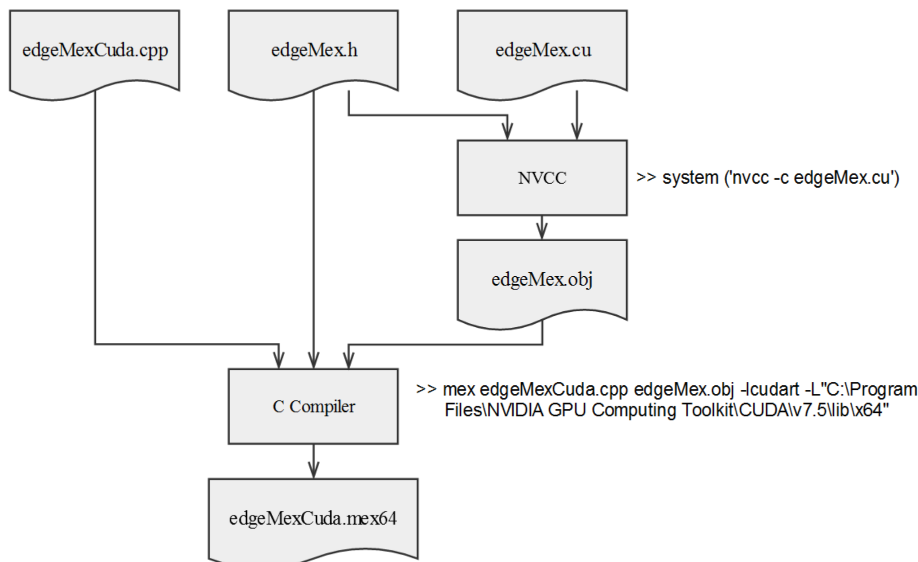
V případě úspěšného vykonání kompilaci se objeví nový soubor s příponou `*.mexw64`. Ověření správnosti realizace funkce `AddVectorsCuda` je na obrázku 4.6.

```
Command Window
New to MATLAB? Watch this Video, see Examples, or read Getting Started.
>>
>> A = single([1 2 3 4]);
>> B = single([4 3 2 1]);
>> C = AddVectorsCuda(A,B)

C =

     5     5     5     5
```

Obrázek 4.6: Ověření implementace funkce `AddVectorsCuda`

4.5 Implementace Cannyho detektoru hran pomocí výpočtů na GPU

Implementace Cannyho detektoru v prostředí MATLAB byla provedena na základě programovacího modelu CUDA. Tato funkce se vyvolává příkazem `edgeMexCUDA(I,threshold)`, kde `I` je vstupní obrázek a `treshold` je parametr odpovídající vysokému prahu při prahování s hysterezi (viz Cannyho detektor hran v kapitole 2.3.1). Implementace je provedena prostřednictvím tří souborů: `edgeMex.cu, edgeMex.h, edgeMexCuda.cpp`. Proces kompilace spustitelného souboru je na obrázku 4.7. Dále jsou probrány nejdůležitější části kódu. Za účelem usnadnit pochopení logiky implementace jsou uvedené výpisy kódu zjednodušeny a zkráceny.



Obrázek 4.7: Kompilace spustitelného souboru CUDA funkce

Část kódu v souboru `edgeMexCuda.cpp` (viz výpis 4.4) je odpovědná za získávání vstupních dat, která budou zpracována CUDA podprogramem, a za vracení výstupních dat. Zde na začátku je nutné zajistit všemožnou kontrolu dat (počet vstupních a výstupních proměnných, jejich datový typ, atd.). Pak pomocí funkce `mxGetData` lze získat vstupní obrázek a pomocí `mxGetM,`

mxGetN zjistit jeho velikost (počet řádků a sloupců). Tato získaná data jsou vstupními funkce edgeMex, která realizuje zpracovávání dat na GPU.

```
1  if (!mxIsSingle(prhs[0])&&!mxIsSingle(prhs[1]))
2  mexErrMsgTxt("Input image and mask type must be single");
3
4  float *image = (float*)mxGetData(prhs[0]);
5
6  int numRows = (int)mxGetM(prhs[0]);
7  int numCols = (int)mxGetN(prhs[0]);
8
9  plhs[0] = mxCreateNumericMatrix(numRows, numCols, ...
       mxSINGLE_CLASS, mxREAL);
10 float *out = (float*)mxGetData(plhs[0]);
11
12 edgeMex(image,out,numRows,numCols);
```

Výpis 4.4: Získávání vstupních dat

Zpracování dat na GPU probíhá v souladu s instrukcemi podprogramu v souboru edgeMex.cu (viz výpis 4.5).

```
1  int totalPixels = numRows*numCols;
2  float *deviceSrc, *deviceDstSob, *deviceDstDir, *deviceDstFil...
       , *deviceDstLoc, *deviceDstThr;
3
4  cudaMalloc(&deviceSrc, sizeof(float)*totalPixels);
5  ...
6  cudaMemcpy(deviceSrc, src, sizeof(float)*totalPixels, ...
       cudaMemcpyHostToDevice);
7  cudaMemset(deviceDstFil, 0, sizeof(float)*totalPixels);
8  ...
9  dim3 blockSize(32,16);
10 dim3 gridSize((numRows+31)/blockSize.x, (numCols+15)/...
       blockSize.y);
11
12 filterMexCuda<<<gridSize,blockSize>>>(deviceSrc,deviceDstFil,...
       numRows,numCols);
13 cudaDeviceSynchronize();
14 sobeledgeMexCuda<<<gridSize,blockSize>>>(deviceDstFil,...
       deviceDstSob,deviceDstDir,numRows,numCols);
15 cudaDeviceSynchronize();
16 localMexCuda<<<gridSize,blockSize>>>(deviceDstSob,...
       deviceDstDir,deviceDstLoc,numRows,numCols);
17 cudaDeviceSynchronize();
18 thresholdMexCuda<<<gridSize,blockSize>>>(deviceDstLoc,...
       deviceDstThr,numRows,numCols);
19 cudaDeviceSynchronize();
20
21 cudaMemcpy(dst, deviceDstThr, sizeof(float)*totalPixels, ...
       cudaMemcpyDeviceToHost);
22 cudaFree(deviceSrc);
23 ...
```

Výpis 4.5: CUDA podprogram

Kernel může zpracovávat pouze data uložená na zařízení, proto musí být nejprve všechna data překopírována do globální paměti grafického procesoru. Vlastnímu kopírování dat do zařízení musí předcházet alokace paměti a po skončení kernelu je paměť obvykle opět uvolněna. V podstatě funkce `cudaMalloc(void *devPtr, size_t size)` slouží k alokaci bloku globální paměti, kde `devPtr` je ukazatel na alokovaný blok a `size_t size` je velikost tohoto bloku v bytech. Alokace paměti má být provedena pro všechny bloky dat, s kterými se předpokládá pracovat [25].

Funkce `cudaMemcpy(void *dst, cons void *src, size_t count, enumMemcpyKind)` umožňuje kopírovaní bloků dat mezi hostitelem a zařízením. Zdrojovou oblast odkud se budou data kopírovat definuje ukazatel `src`, cílovou oblast do které se budou data ukládat určuje ukazatel `dst` a počet kopírovaných bytů je dán parametrem `count`. Směr kopírování dat popisuje výpočtový typ `cudaMemcpyKind`, který udává z jakého adresního prostoru a do jakého se bude kopírovat [25]. Povolené kombinace:

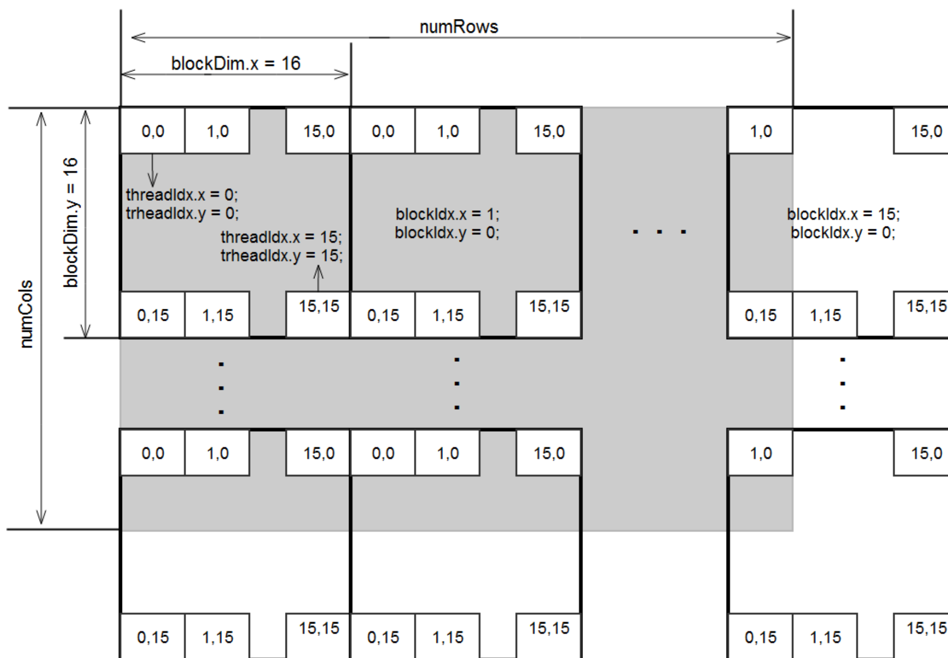
1. `cudaMemcpyHostToHost`: kopírování dat v rámci hostitele (CPU);
2. `cudaMemcpyHostToDevice`: kopírování dat z CPU na GPU;
3. `cudaMemcpyDeviceToHost`: kopírování dat z GPU na CPU;
4. `cudaMemcpyDeviceToDevice`: kopírování dat v rámci globální paměti.

Funkce `cudaMemset(void *devPtr, int value, size_t count)` nastaví část globální paměti na zadanou hodnotu `value`. Začátek nastavovaného bloku je určen ukazatelem `devPtr` a jeho velikost v bytech parametrem `size_t count`.

Protože velikost vstupního obrázku je mnohem větší, než maximální počet vláken v CUDA modelu, je nutné řešit dělení vstupního obrázku na bloky (viz obrázek 4.8).

Bloky mohou být jedno, dvou nebo třírozměrné, stejně tak mřížka může mít jednu, dvě nebo tři dimenze. Každé vlákno má přidělený jednoznačný identifikátor v rámci bloku `threadIdx` a každý blok má svůj index `blockIdx` v rámci mřížky. Uvedené indexy slouží k odlišení vláken, ale hlavně si na jejich základě musí každé vlákno rozhodnout, k jakým datům bude přistupovat a kam se bude zapisovat výsledek. Třírozměrné identifikátory jsou zavedeny pro zjednodušenou indexaci dat, neboť spousta problémů je přirozeně vícerozměrných. V rámci bloků si mohou vlákna vyměňovat data přes sdílenou paměť a také je lze vzájemně synchronizovat pomocí bariéry [25].

V rámci dané úlohy jedno vlákno je zodpovědné za zpracování jednoho bodu vstupního obrázku. A na základě specifikace uvedenou v [3] byly zvoleny parametry bloků a mřížky, podle nichž se zpracovává vstupní data. Podle této specifikace maximální počet vláken v jednom bloku nemůže být větší než 1024. Obvykle se volí blok s 512 nebo 1024 vlákny. V této úloze se hodí zvolit blok rozměry 32x16 a takovým pádem definovat tří dimenzionální proměnnou `blockSize(32,16,1)`. Dále je nutno definovat mřížku - počet bloků na ose X a Y: `gridSize((numRows+31)/blockSize.x, (numCols+15)/blockSize.y)`.



Obrázek 4.8: Dělení vstupních dat na bloky

Pak probíhá zpracovaní každého datového prvku funkcemi, které realizují detekci hran na základě Cannyho algoritmu. Funkce `filterMexCuda` (viz algoritmus 4.1) realizuje první krok tohoto algoritmu (viz kapitolu 2.3.1) a provádí konvoluce vstupního obrázku a jádra Gaussova filtru.

Input: $src \leftarrow$ vstupní obrázek, $numRows, numCols$
Output: $dstFil$
//zjištění řádku datového elementu
$row = blockIdx.x * blockDim.x + threadIdx.x$;
//zjištění sloupečku datového elementu
$col = blockIdx.y * blockDim.y + threadIdx.y$;
//zjištění indexu datového elementu
$dstIndex = col * numRows + row$;
$ker \leftarrow$ jádro Gaussova filtru;
$kerIndex = 8$;
for *kc=-1:2* **do**
  //index okolního datového elementu
  $srcIndex = (col + kc) * numRows + row$;
  for *kr=-1:2* **do**
    //výpočet konvoluce
    $dstFil[dstIndex] + = ker[kerIndex - -] * srcFil[srcIndex + kr]$;
  end
end

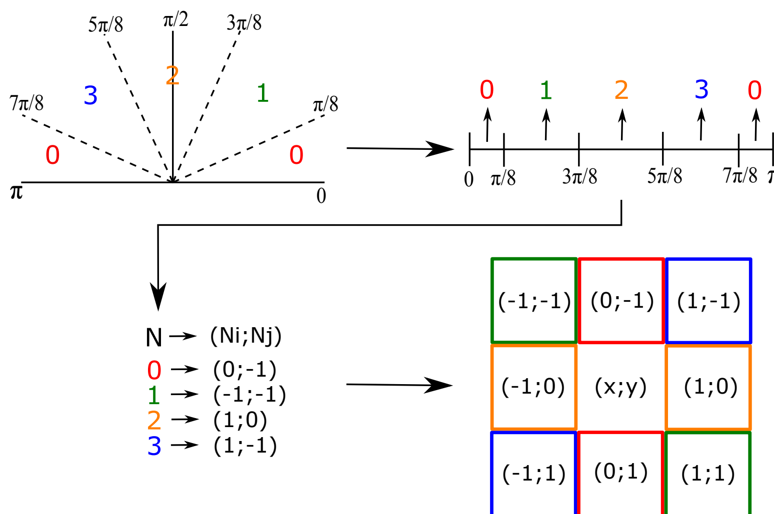
Algoritmus 4.1: Gaussova filtrace

Funkce `sobeledgeMexCuda` (viz algoritmus 4.2) určuje velikost gradientu a jeho směr ve každém obrazovém bodě (viz kapitolu 2.3.1).

Input: $src \leftarrow$ vstupní obrázek, $numRows, numCols$
Output: $dstMag, dstDir$
$row \leftarrow blockIdx.x * blockDim.x + threadIdx.x$;
$col \leftarrow blockIdx.y * blockDim.y + threadIdx.y$;
$dstIndex \leftarrow col * numRows + row$;
$maskX \leftarrow$ jádro filtru;
$maskY \leftarrow$ jádro filtru;
$kerIndex \leftarrow 8$;
for *kc=-1:2* **do**
  $srcIndex \leftarrow (col + kc) * numRows + row$;
  for *(kr=-1:2)* **do**
    $edgeX+ = maskX[kerIndex] * src[srcIndex + kr]$;
    $edgeY+ = maskY[kerIndex - -] * src[srcIndex + kr]$;
  end
end
//výpočet gradientu
$dstMag[dstIndex] = sqrt(pow(edgeX, 2) + pow(edgeY, 2))$;
//výpočet směru hrany
$dstDir[dstIndex] = atan(edgeY/edgeX) + pi/2$;
if $(dstDir[dstIndex] > 7 * pi/8)$ **then**
  $dstDir[dstIndex]- = 7 * pi/8$;
end

Algoritmus 4.2: Určení velikosti gradientu a jeho směru

Dále podle Cannyho algoritmu je nutno provést hledání lokálních maxim v směru kolmém k směru gradientu (viz algoritmus 4.3). Pro to směr určený v předchozím kroku se zaokruhuje a provádí se transformace tohoto směru na indexy lokálních bodů, mezi nimiž se bude určovat lokální maximum (viz obrázek 4.9) .



Obrázek 4.9: Transformace směru hrany do indexů bodů na obrázku [4]

Input: $srcM \leftarrow$ magnituda gradientu, $srcM \leftarrow$ směr hrany, $numRows, numCols$
Output: $dstLoc$
$row \leftarrow blockIdx.x * blockDim.x + threadIdx.x$;
$col \leftarrow blockIdx.y * blockDim.y + threadIdx.y$;
$dstIndex = col * numRows + row$;
$N = ceilf(4 * srcD[dstIndex]/pi - 0.5)$;
$N_i = 1 - (N == 0) + ((N == 1) > 1)$;
$N_j = 1 - (N == 2)$;
$dstIndex1 = (col + Ni) * numRows + row + Nj$;
$dstIndex2 = (col - Ni) * numRows + row - Nj$;
if $(srcM[dstIndex] > srcM[dstIndex1]$ & $srcM[dstIndex] > srcM[dstIndex2])$ **then**
| $dstLoc[dstIndex] = srcM[dstIndex]$
end

Algoritmus 4.3: Určení lokálního maxima v směru kolmém směru gradientu

Posledním krokem Cannyho detektoru hran je provádění prahování s hysterezi (viz algoritmus 4.4).

Input: $src \leftarrow$lokální maximy obrázku,$h_th \leftarrow$ vysoký práh, $l_th \leftarrow$ nízký práh, $numRows, numCols$
Output: $dstThr$
$row = blockIdx.x * blockDim.x + threadIdx.x$;
$col = blockIdx.y * blockDim.y + threadIdx.y$;
$dstIndex = col * numRows + row$;
if $(src[dstIndex] <= l_th)$ **then**
| $dstThr[dstIndex] = 0$;
end
if $(src[dstIndex] > l_th)\&(src[dstIndex] < h_th)$ **then**
| **if** $(src[dstIndex-1] > h_th)||(src[dstIndex+1] > h_th)||(src[(col+1)* numRows + row] > h_th)||(src[(col-1)*numRows+row] > h_th)$ **then**
| | $dstThr[dstIndex] = 255$; **else**
| | | $dstThr[dstIndex] = 0$;
| | **end**
| **end**
end

Algoritmus 4.4: Prahování s hysterezi

Po provádění všech kroku Cannyho algoritmu nad všemi datovými elementy bloky výstupních dat se kopírují ze zařízení na hostitel a uvolní se dříve alokované bloky globální paměti pomocí funkce `cudaFree(void *devPtr)`.

Výsledky zpracovávání dat navrženou funkcí jsou uvedeny na obrázku 4.10.

a - MATLAB funkce

b - CUDA funkce

Obrázek 4.10: Porovnávání výsledků detekce hran

4.6 Využití externích knihoven

Externí knihovny mohou rychle usnadnit implementaci výpočtů na GPU. Během této práce byla vyzkoušena knihovna ArrayFire, která zahrnuje funkce využívající paralelní výpočty a je skvělým nástrojem v případě, kde není možnost realizace vlastních CUDA funkcí. Program s využitím funkcí této knihovny je velmi jednoduchý:

```
1 using namespace af;
2
3 array myfunc(array input)
4 {
5     array Gx, Gy;
6     sobel(Gx,Gy,input,3);
7     array output = sqrt(pow(Gx,2)+pow(Gy,2));
8     return (output);
9 }
```

Výpis 4.6: Realizace Sobelova detektoru hran pomocí knihovny ArrayFire

Tato knihovna je ve volném dostupu, ale množství funkcí dostupných pro zpracování obrázků není v tuto chvíli tak veliké. Knihovna ArrayFire je vhodná v případě nutnosti rychle realizace nějakého základního algoritmu s využitím paralelních výpočtů, jakými jsou filtrace, segmentace, morfologické operace atd.

4.7 Implementace CUDA funkce v prostředí LabVIEW

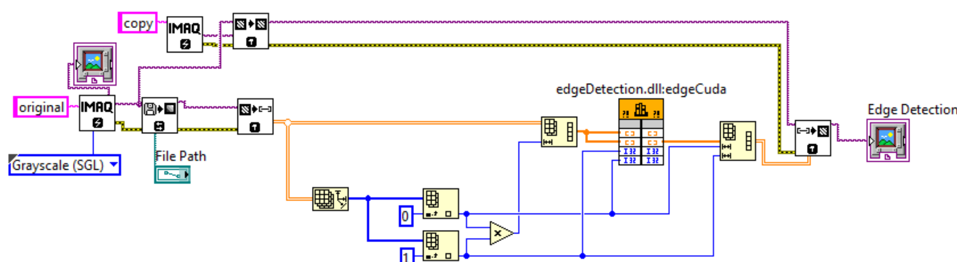
LabVIEW GPU Analysis Toolkit umožňuje komunikaci LabVIEW aplikace s NVIDIA CUDA grafickým procesorem. Tato knihovna zajišťuje možnost alokace pamětí, přenosu dat s hostitele na zařízení a zpátky, a kontrolu plnění

GPU kódu. LabVIEW GPU Analysis Toolkit taky umožňuje využití CUDA knihoven navržených společnosti NVIDIA. Mezi nimi jsou:

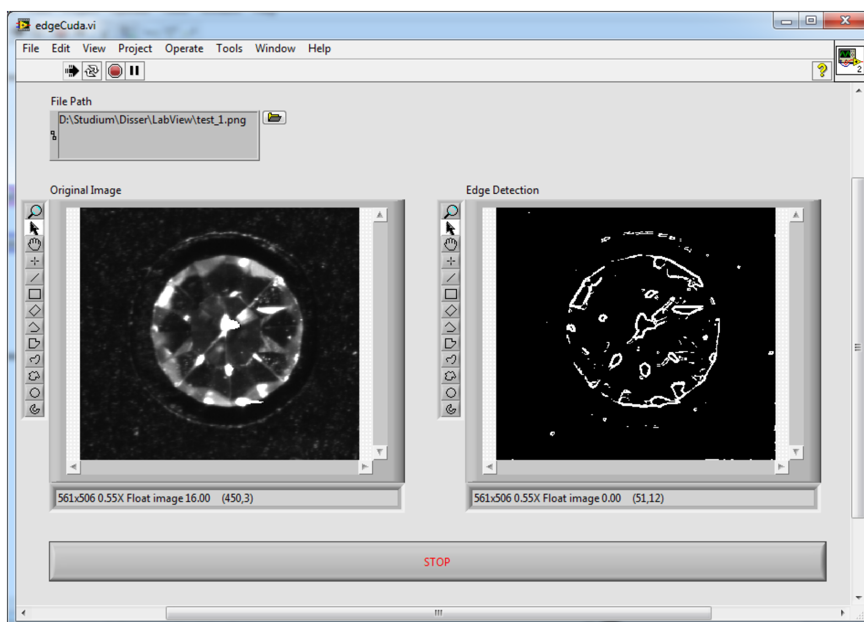
1. CUDA Basic Linear Algebra Subroutines (CUBLAS);
2. CUDA FFT knihovna (CUFFT).

Pro pokročilé operace, kód, který již byl vytvořen v CUDA, může být vložen do LabVIEW kódu jako funkce externí knihovny. Zde je důležité si všimnout, že GPU Analysis Toolkit nesestavuje LabVIEW kód pro použití GPU, ale spíš umožňuje zabalení CUDA funkce nebo CUDA jader definovaných uživatelem, které mají být použity v LabVIEW [29].

Algoritmus implementace CUDA kódu vytvářeného uživatelem v prostředí LabVIEW je popsán ve příloze B. A na základě tohoto algoritmu v prostředí LabVIEW byl implementován CUDA kód, který realizuje detekci hran (viz obrázek 4.11). Výsledky zpracovávání vstupních obrazových dat LabVIEW aplikací jsou na obrázku 4.12.



Obrázek 4.11: Implementace externí CUDA funkce v prostředí LabVIEW



Obrázek 4.12: LabVIEW aplikace využívající paralelní výpočty

Kapitola 5

Výsledky

V této kapitole je popsáno implementované uživatelské rozhraní, pomocí kterého lze ohodnotit kvalitu rozpoznávání geometrických tvarů kamenů a mezivýsledky zpracování dat. Jsou uvedeny výsledky subjektivního hodnocení rozpoznávání geometrických tvarů kamenů a jsou představeny výsledky hodnocení časové náročnosti zpracování dat při využití výpočtů na CPU a GPU v prostředí MATLAB a LabVIEW.

5.1 Implementace uživatelského rozhraní

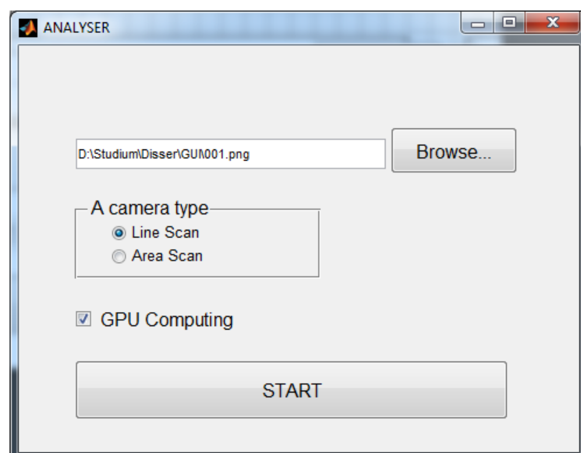
Uživatelské rozhraní (GUI) poskytuje snadné ovládání softwarové aplikace, což eliminuje potřebu se vyznávat v kódu programu pro spuštění aplikace nebo pro nastavení dalších parametrů. Proto za účelem pohodlného hodnocení výsledků zpracování obrazových dat bylo rozhodnuto implementovat grafické uživatelské rozhraní.

Prostředí MATLAB poskytuje možnost vytváření vlastního uživatelského rozhraní pomocí vývojového prostředí GUIDE. GUIDE Layout Editor dovoluje navrhnout samostatný design rozhraní. Pak GUIDE vygeneruje MATLAB kód, kterým se budou řídit objekty rozhraní. Úprava tohoto kódu definuje vlastnosti a chování objektů navrženého rozhraní. Na základě této logiky a navrženého algoritmu rozpoznávání geometrických tvarů kamenů bylo implementováno uživatelské rozhraní v prostředí MATLAB.

Při spuštění aplikace se na začátku vyvolává okénko jako je na obrázku 5.1. Zde si uživatel musí zvolit soubor k zpracovávání, jenž má příponu `*.png`. Také je nutné ukázat typ kamery, která byla použita k získávaní obrazových dat. Tento krok je důležitým, protože na typu kamery záleží volení parametrů detekce a špatné stanovení toho typu může být zdrojem velké množiny chyb. Taky je nutné ukázat, jestli technické prostředky systému dovolují využití výpočtů na grafickém procesoru.

Po nastavení všech vstupních parametrů a stisknutí tlačítka "START"probíhá předzpracování obrazových dat a vyvolává se další okénko (viz obrázek 5.2).

Uživatel tuto aplikaci může využít k sbírání statistik kvality lokalizace hran faset v závislosti na kvalitě vstupního obrázku. Jinými slovy si uživatel na začátku má rozhodnout podle jakého kriteria bude hodnotit vstupní obrázek a kvalitu detekce. Pak postupným zkoumáním všech detekovaných

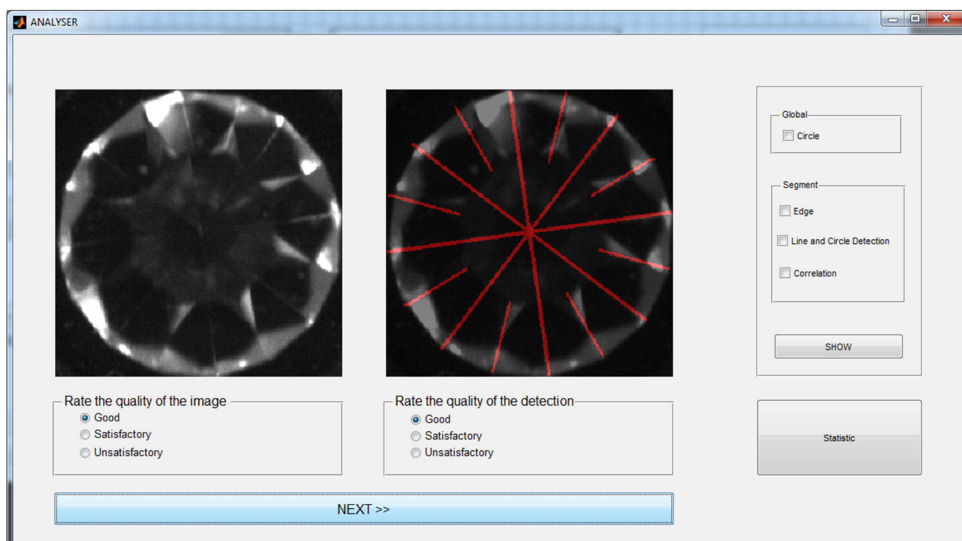


Obrázek 5.1: Volení vstupních dat v GUI

objektů ohodnotit, jestli podle těchto kriterií vstupní obrázek či detekce je příjemná, uspokojivá nebo není. Pomocí tlačítka "STATISTIC"lze získat procento příjemné a uspokojivé detekce na kvalitních obrazcích a procento nepříjemných vstupních dat či detekcí.

Pomocí panelu v pravé částí okénka uživatel může vykreslit mezivýsledky zpracování vstupního obrázku. Mezi nimi jsou množina všech detekovaných objektů, výsledky zpracování zkoumaného segmentu detektorem hran, množina nalezených čar v tomto segmentu a taky průběh korelace při otáčení masky kolem detekovaného středu kamenu.

Pomocí tlačítka "NEXT"lze přejít na hodnocení dalšího detekovaného objektu na vstupním obrázku. Po ukončení hodnocení všech objektů tlačítko "NEXT"bude neviditelným.



Obrázek 5.2: Hodnocení kvality detekce v GUI

5.2 Hodnocení rozpoznávání geometrických tvarů kamenů

Existují objektivní a subjektivní metody hodnocení kvality zpracování obrazových dat. Subjektivní testy připadají nejvíc důvěryhodnými, protože jejich výsledky odpovídají názoru pozorovatele.

To znamená, že v případě hodnocení výsledků určení vlastností kamínků, tento proces může spoléhat na fakt, že lidský zrakový zdroj dokáže provést rozpoznávání hran faset na vstupním obrázku a ohodnotit výslednou lokalizaci těchto hran. Za tímto účelem pozorovateli může být nabídnuto rozhodnout jestli vstupní data jsou příjemní, uspokojivá a nebo nepříjemní na základě parametrů uvedených v tabulce 5.1. Příklady vyhovující různým stupňům kvality vstupních dat jsou uvedeny v příloze na obrázcích C.1 a C.2.

	Hodnocení kvality
Správné umístění kamínku a nejsou velké odrazy	příjemná
Správné umístění kamínku, ale jsou odrazy	uspokojivá
Špatné umístění kamínku nebo velké odrazy	nepříjemná

Tabulka 5.1: Hodnocení kvality vstupních dat

Následující hodnocení lokalizace hran faset lze provést na základě parametrů uvedených v tabulce 5.2. Příklady vyhovující uvedeným stupňům kvality jsou uvedeny v příloze na obrázku C.3 a C.4.

	Hodnocení kvality
Správně detekována špička a hrany faset	příjemná
Správně detekována špička a většina hran faset	uspokojivá
Špatně detekovaná špička nebo většina hran faset	nepříjemná

Tabulka 5.2: Hodnocení kvality lokalizace hran faset

Na základě této logiky byla ohodnocena navržená obrazová data a výsledky jejich zpracování algoritmem rozpoznávání tvarů kamenů. Získaná data jsou uvedena v tabulce 5.3.

	Řádková kamera	Plošná kamera
Celkový počet ohodnocených objektů	168	78
Kvalitní vstupní data	100	68
Správně detekována špička a hrany faset	22	36
Správně detekována špička a většina hran faset	51	18

Tabulka 5.3: Výsledky subjektívního hodnocení lokalizace hran faset

Na základě těchto dat lze spočítat 95%-intervaly spolehlivosti, v nichž bude pohybovat procento příjemné a uspokojivé kvality lokalizace hran faset pří použití různých vstupních dat (viz tabulku 5.4).

	Řádková kamera	Plošná kamera
Příjemná kvalita lokalizace hran faset	$(22 \pm 8.16)\%$	$(52.94 \pm 12.19)\%$
Uspokojivá kvalita lokalizace hran faset	$(51 \pm 9.85)\%$	$(26.47 \pm 10.78)\%$
Celková příjemná kvalita lokalizace hran faset	$(73 \pm 8.75)\%$	$(79.41 \pm 8.75)\%$

Tabulka 5.4: Výsledky subjektívního hodnocení lokalizace hran faset

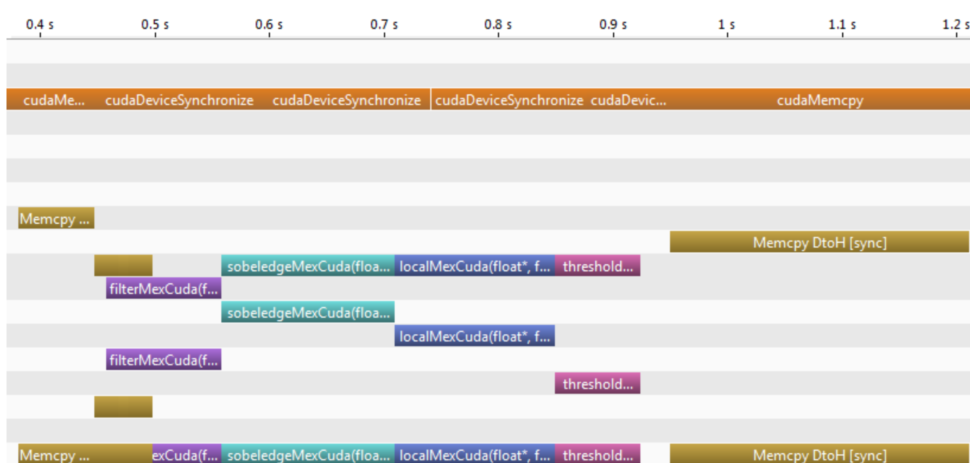
5.3 Hodnocení časové náročnosti

Časová náročnost zpracování dat navrženým algoritmem je velmi důležitým parametrem dané úlohy. Proto bylo provedeno hodnocení času zpracovávání obrazových dat velikostí 8160x8160 pixelů v prostředí MATLAB a LabVIEW.

Analýza časové náročnosti v prostředí MATLAB se prováděla pomocí MATLAB Profiler. Zpracovávání obrazových dat navrženým algoritmem zabírá 480 sekund. Nejvíc času zabírají předzpracování obrázku (detekce hran), segmentace obrazu pomocí Houghovy transformace a porovnávání s maskou při určení úhlu natočení kamenu.

Implementace Cannyho detektoru hran pomocí výpočtu na grafickém procesoru značně zrychlovává navržený algoritmus. Zpracování dat celým algoritmem probíhá skoro 5 krát rychleji (99 sekund) a detekce hran 12 krát rychleji (viz obrázek 5.4 a).

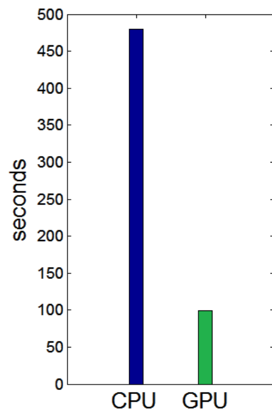
Analýza implementace detekce hran s využitím paralelních výpočtů byla provedena pomocí NVIDIA Visual Profiler (viz přílohu D). Na obrázku 5.3 je vidět, že při provádění výpočtů na GPU předávání dat z hostu na zařízení a zpátky je časově náročnou operací, proto nemá smysl využívat grafický procesor při zpracování malých objemů dat.



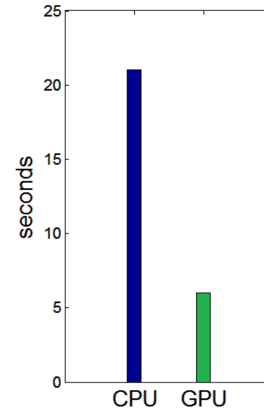
Obrázek 5.3: Časový profil algoritmu detekce hran na GPU

Hodnocení času zpracovávání dat v prostředí LabVIEW bylo zrealizováno

pomocí operátoru Tick Count. Zpracování navržených obrazových dat virtuálním nástrojem IMAQ CannyEdgeDetection v prostředí LabView trvá 21.3 sekundy. Využití paralelních výpočtů v tomto prostředí zrychlovává detekci hran do 6.39 sekund (viz obrázek 5.4 b).



a - prostředí MATLAB



b - prostředí LabVIEW

Obrázek 5.4: Porovnávaní času zpracovávání dat na CPU a GPU

Kapitola 6

Závěr

Během práce jsem si stanovila dva dílčí cíle. První se týkal stanovení kriterií pro implementaci algoritmu rozpoznávání geometrických tvaru a zkoumaní různých metod zpracování obrazu. Provedený teoretický rozbor ukázal, že nejdůležitějšími kriterií jsou rychlost zpracování dat a přesnost vyhodnocení obrazu. Aplikování metod rozpoznávání geometrických tvarů kamenů v prostředí MATLAB dovolilo zrealizovat algoritmus lokalizace hran faset a špičky kamene za účelem rozpoznaní tvaru objektu.

Dalším cílem bylo řešení problému časové náročnosti s tím, aby dílčí části algoritmu bylo možné využít i v prostředí LabVIEW. Proto jsem implementovala Cannyho hranový detektor využívající paralelní výpočty na grafickém procesoru NVIDIA GeForce GT 635M podporující CUDA architekturu. Výsledkem je knihovna, které lze využit jak v prostředí MATLAB, tak i LabVIEW.

Pro snadné ovládání jsem vytvořila grafické uživatelské rozhraní. V tomto rozhraní může uživatel aplikace hodnotit kvalitu rozpoznávání geometrických tvarů a mezivýsledky zpracování obrazových dat.

Navíc jsem provedla subjektivní hodnocení výsledků rozpoznávání geometrických tvarů kamenů při použití vstupních dat získaných řádkovou a plošnou kamerou. Subjektivní hodnocení ukázalo, že lepších výsledků lze dosáhnout při použití plošné kamery.

Také jsem provedla hodnocení rychlosti zpracování dat při využití výpočtu na GPU a CPU. Ukázalo se, že při využití paralelních výpočtů na grafickém procesoru probíhá zpracování dat mnohokrát rychleji, než při výpočtech na centrálním procesoru.

Hlavní přínos této práce spočívá v úspěšné aplikaci metod rozpoznávání tvarů bižuterních kamenů s využitím paralelních výpočtů na GPU. Dílčí částí navrženého algoritmu lze využit v podobě knihovny i v prostředí LabVIEW.

Z pohledu návaznosti na tuto práci by byl docela užitečný návrh objektivních a subjektivních kriterií hodnocení vstupních obrazových dat a rozpoznávání geometrických tvarů kamenů. Protože na základě těchto kriterií lze porovnat různé metody rozpoznávání geometrických tvarů.

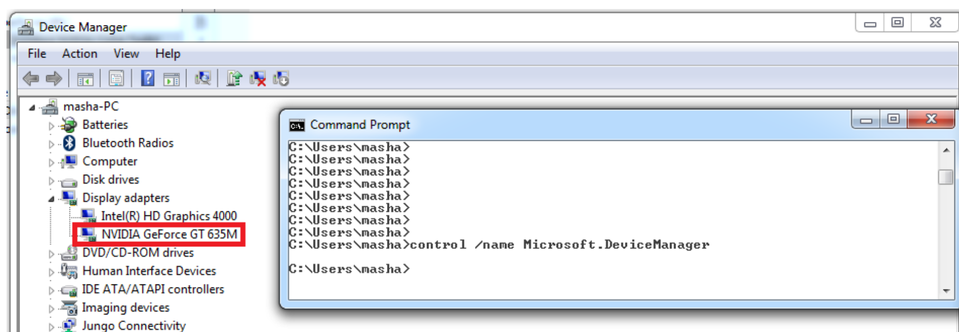
Dále by bylo zajímavé rozvíjet myšlenku využití distanční transformace pro hodnocení kvality povrchu kamene a detekci významných vad. Zde je nutné získat roztříděná vstupní data podle toho, jaké vady se na kamenech vyskytují. A pak na základě analýzy těchto dat navrhnout kritéria klasifikace

kamenů, mezi nimiž se mohou vyskytnout nedoleštěný kámen, rozbitý kámen, přetažené simili (kovový povlak dolních faset) a nebo nežádoucí obrácené simili (kovový povlak horních faset).

Příloha A

Instalace NVIDIA CUDA Toolkit

Prvním důležitým krokem je ověření toho, že používaný grafický procesor opravdu podporuje NVIDIA CUDA Toolkit. Model GPU lze zjistit pomocí Windows Device Manageru (viz obrázek A.1). A pak plný seznam procesorů podporujících NVIDIA CUDA Toolkit je uveden na http:// developer.nvidia.com /cuda-gpus.



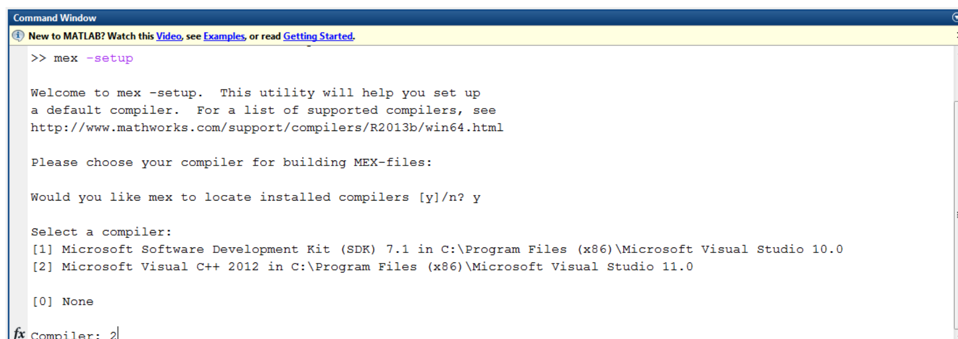
Obrázek A.1: Zjištění modelu GPU ve Windows 7

Pak má smysl zkontrolovat jaké operační systémy podporuje verze CUDA, která bude nainstalována. Tuto informaci lze najít ve [27].

Dalším nutným před instalací CUDA 7.5 krokem je instalace C/C++ kompilátoru. Kompilátory, které jsou podpořené v CUDA, taky lze najít v [27]. Dříve stačilo nainstalovat MS Visual Studio 2012, avšak od roku 2014 Microsoft změnil způsob distribuce produktu Visual Studio, který je teď dostupný v třech verzích: Community Edition, Professional a Enterprise Edition (poslední dvě jsou plácené). Verze Comunity odpovídá Express verzí Visual Studio a nezahrnuje všechny knihovny nutné pro CUDA 7.5. Jedním z řešení problémů je nainstalovaní Professional verze MS Visual Studio. Po instalaci je dostupná 30-denní zkušební doba, po které program MS Visual Studio nebude spustitelný, avšak knihovny budou k dispozici i po teto době.

Po instalaci MS Visual Studio 2012 Professional Version je nutné nastavit implicitní použití tohoto kompilátoru v prostředí MATLAB. Pro to se spustí MATLAB a v příkazovém řádku se zavolá `mex -setup` (viz obrázek A.2). Pak by se měl objevit uvítací vzkaz. Pokračování v nastavení se zajišťuje stisknutím `[y]` a zvolením kompilátoru, který bude používán implicitně. V

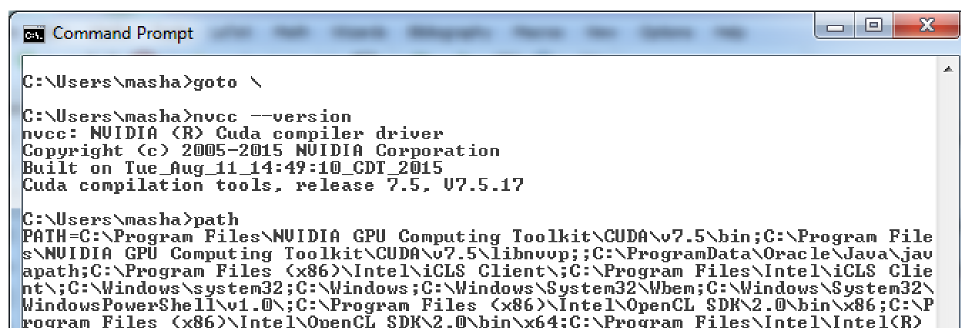
tomto případě je to Microsoft Visual C++ 2012, který lze zvolit stisknutím `[2]`.



Obrázek A.2: Vyvolávání `mex -setup`

Po úspěšném vykonávání předchozích kroků lze začít s instalací CUDA 7.5. Stáhnout NVIDIA CUDA Toolkit lze z http://developer.nvidia.com/cuda-downloads. Instalace CUDA softwaru je možná pomocí CUDA instalatéru.

Posledním krokem je verifikace instalace. Za tímto účelem lze zkusit vykonat příkaz `nvcc -version` a nebo provést kontrolu proměnné `path` (viz obrázek A.3), která by měla zahrnovat PATH=C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v7.5\bin; C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v7.5\libnvvp.



Obrázek A.3: Verifikace instalace CUDA 7.5

Příloha B

Implementace CUDA funkce v LabVIEW

Implementaci CUDA funkce v LabVIEW lze provést pomocí vytváření dynamicky linkované knihovny DLL z CUDA programu v C/C++ kompilátoru [30], [31]. Při použití MS Visual Studio je nutné vytvořit nový projekt: Win32 Console Application, v dalších krocích zvolit typ aplikace DLL a opci empty project.

Tuto implementaci si ukážeme na příkladu funkce, která má za úkol složení dvou vektorů stejné délky. V tomto projektu je nutné vytvořit header file, který obsahuje následující kód:

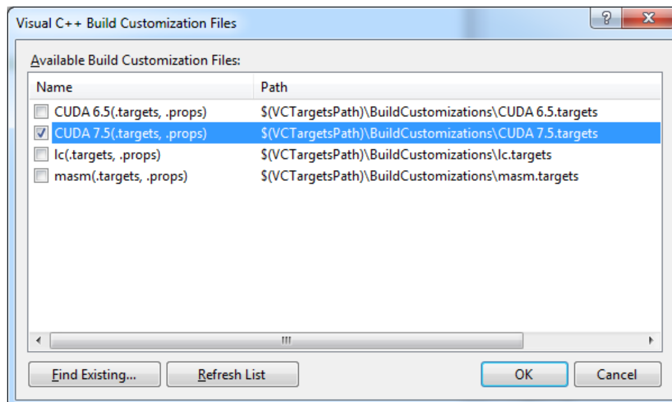
```
1  #ifndef __ADDVECTORS_H__
2  #define __ADDVECTORS_H__
3
4  #if defined DLL_EXPORT
5  #define DECLDIR __declspec(dllexport)
6  #else
7  #define DECLDIR __declspec(dllimport)
8  #endif
9
10 extern "C"
11 {
12 DECLDIR void AddVectors(float*A,float*B,float*C,int size);
13 }
14
15 //extern void addVectors(float*A,float*B,float*C,int size);
16
17 #endif // __ADDVECTORS_H__
```

Výpis B.1: Header file

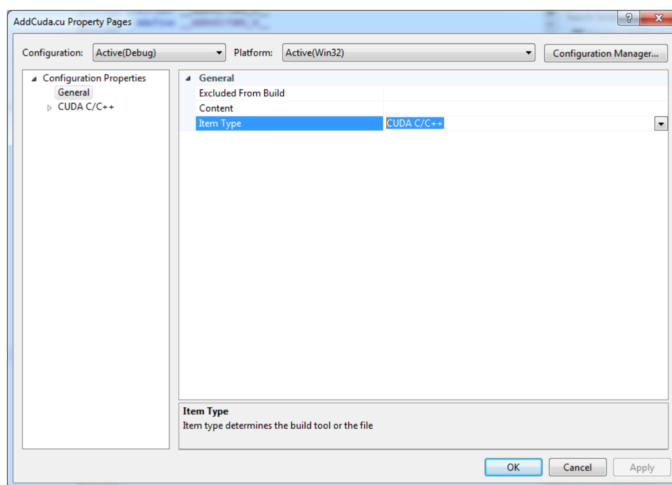
Pak je nutné zavolat Build Customization z menu k tomuto projektu a tam nastavit použití CUDA 7.5 (viz obrázek B.1).

Dále je nutné v tomto projektu přidat nový C++ soubor. Změnit příponu souboru C++ souboru *.cpp na *.cu a zavolat okénko vlastností souboru s příponou *.cu. Ve Confifuration Properties → General → Item Type zmenit na CUDA C/C++ (viz obrázek B.2).

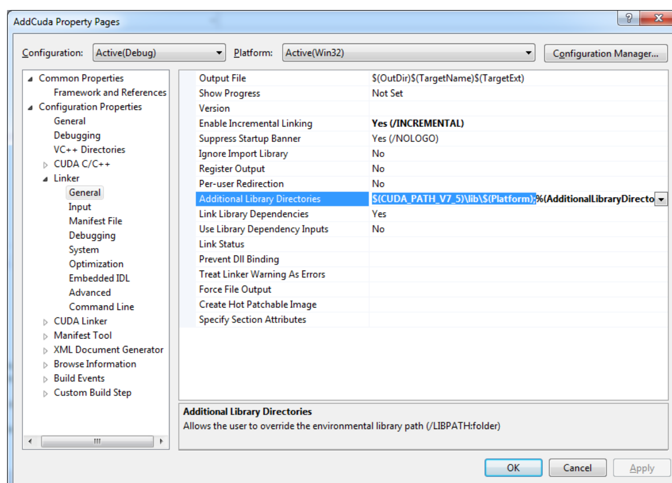
Dále se nastavují vlastnosti projektu. Ve Configuration Properties → Linker → General → Additional Library Directories (viz obrázek B.3) se přidává: $(CUDA_PATH_V4_0)\lib\$(Platform);



Obrázek B.1: Nastavení implecitního využití CUDA 7.5

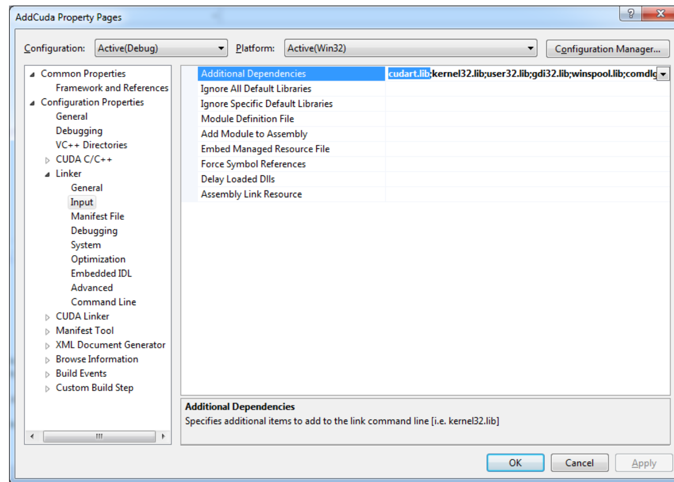


Obrázek B.2: Nastavení vlastností CUDA souboru



Obrázek B.3: Nastavení vlastností projektu

Ve Configuration Properties → Linker → Input → Additional Dependencies (viz obrázek B.4) je nutné přidat cudart.lib.



Obrázek B.4: Nastavení vlastností projektu

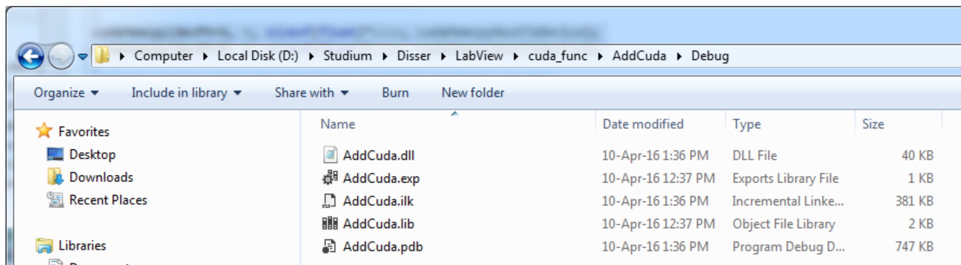
CUDA soubor má zahrnovat kód funkce:

```
1  #define DLL_EXPORT
2  #include "AddCuda.h"
3  #include <cuda_runtime.h>
4  #include <cuda.h>
5
6  extern "C"
7  {
8
9  __global__ void addVectorsMask (float*A, float*B, float*C, ...
       int size)
10
11 {
12 int i = blockIdx.x;
13 if (i≥size)
14 return;
15
16 C[i]=A[i]+B[i];
17 }
18
19 DECLDIR void addVectors(float*A, float*B, float*C, int size)
20 {
21 float*devPtrA=0, *devPtrB=0, *devPtrC=0;
22
23 cudaMalloc(&devPtrA, sizeof(float)*size);
24 cudaMalloc(&devPtrB, sizeof(float)*size);
25 cudaMalloc(&devPtrC, sizeof(float)*size);
26
27 cudaMemcpy(devPtrA, A, sizeof(float)*size, ...
       cudaMemcpyHostToDevice);
28 cudaMemcpy(devPtrB, B, sizeof(float)*size, ...
       cudaMemcpyHostToDevice);
```

```
29
30 addVectorsMask<<<size,1>>>(devPtrA, devPtrB, devPtrC, size);
31
32 cudaMemcpy(C, devPtrC, sizeof(float)*size, ...
       cudaMemcpyDeviceToHost);
33
34 cudaFree(devPtrA);
35 cudaFree(devPtrB);
36 cudaFree(devPtrC);
37 }
38
39 }
```

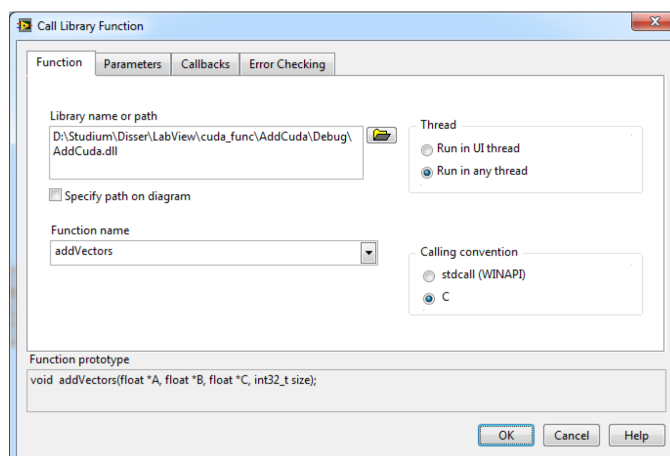
Výpis B.2: CUDA soubor

Po vyvolávání Build the Solution má vzkniknout soubory jako na obrázku B.5.

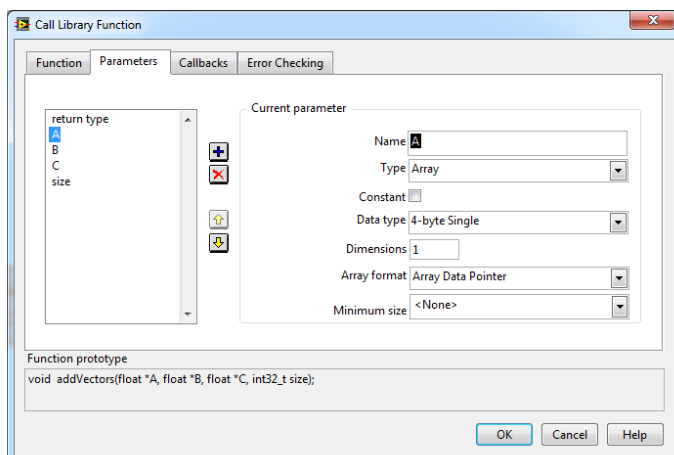


Obrázek B.5: Soubory vznikaící po kompilaci DLL projektu

Dále je nutné zajistit možnost použití této zkompilované knihovny v prostředí LabVIEW. S tímto účelem lze použit blok Connectivity → Library & Executable → Call Library Function Node. Po zavolání vlastností tohoto bloku nutno ukázat cestu do zkompilovaného DLL souboru a jméno funkce, která se bude používat (viz obrázek B.6). Pak v tomto okénku je nutné uvést vstupní a výstupní proměnné a jejich typ (viz obrázek B.8).

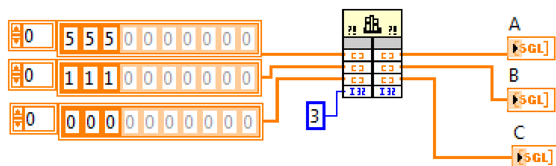


Obrázek B.6: Nastavení vlastností bloku Call Library Function



Obrázek B.7: Nastavení vlastností bloku Call Library Function

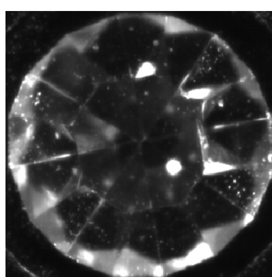
Výsledný kód v prostředí LabVIEW je uveden na obrázku B.8.

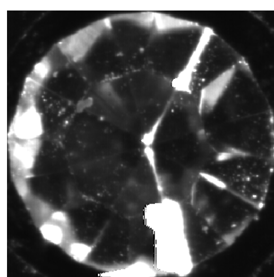


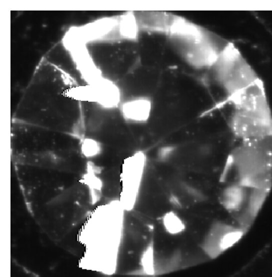
Obrázek B.8: Využití externí knihovny v prostředí LabVIEW

Příloha C

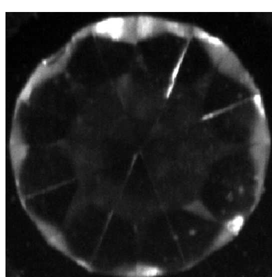
Hodnocení výsledků určení vlastností kamenů


a - Příjemná

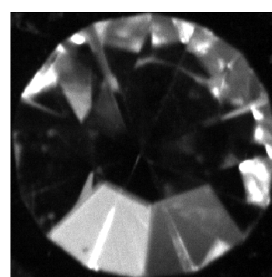

b - Uspokojivá


c - Nepříjemná

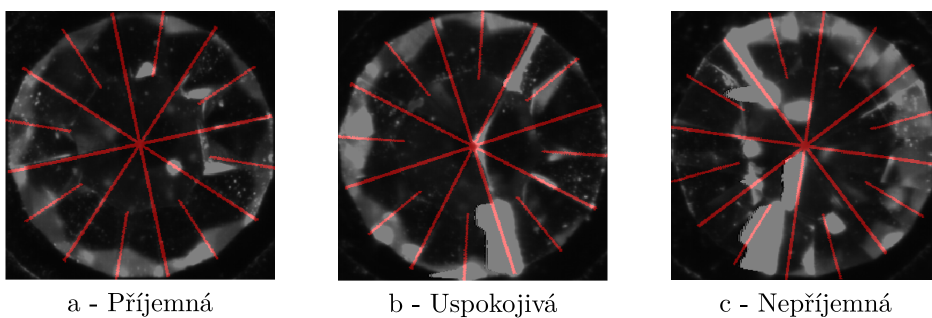
Obrázek C.1: Kvalita vstupních obrazových dat získaných pomocí řádkové kamery


a - Příjemná

b - Uspokojivá

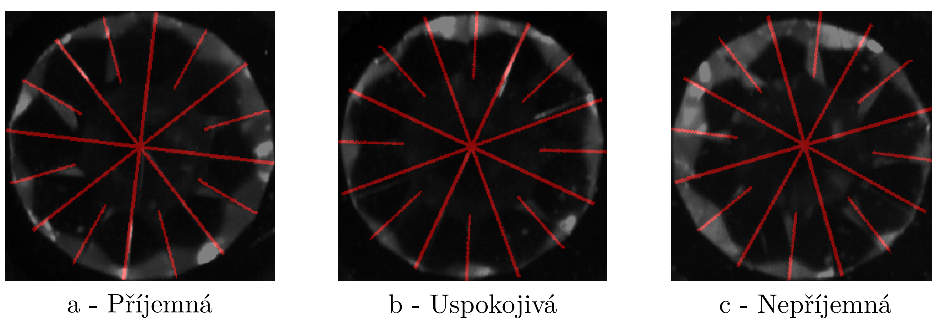

c - Nepříjemná

Obrázek C.2: Kvalita vstupních obrazových dat získaných pomocí plošné kamery



a - Příjemná b - Uspokojivá c - Nepříjemná

Obrázek C.3: Kvalita detekce hran faset při použití řádkové kamery



a - Příjemná b - Uspokojivá c - Nepříjemná

Obrázek C.4: Kvalita detekce hran faset při použití plošné kamery

Příloha D

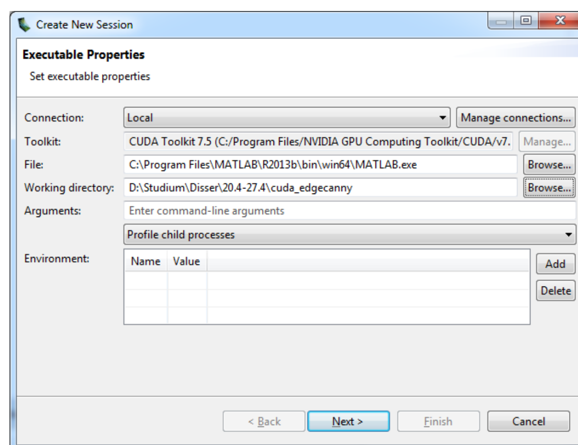
NVIDIA Visual Profiler

V případě nutnosti využití analýzy času zpracování vstupních dat pomocí NVIDIA Visual Profiler je nutné lehce upravit kód souboru *.cpp (viz kapitolu 4.4). Zde je nutné zajistit vložení souboru `cuda_runtime.h` a zavolání funkce `cudaDeviceReset()`:

```
1  #include ...
2  #include "cuda_runtime.h"
3
4  void mexFunction() {
5  ...
6  addVectors(A,B,C,size);
7  cudaDeviceReset();}
```

Výpis D.1: *.cpp soubor

NVIDIA Visual Profiler se obvykle instaluje spolu s CUDA, a lze ho najít v C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v7.5\libnvvp \nvcc.exe. Zde je nutné vyvolat File → New Session a nastavit spuštění MATLAB a pracovní direktorium spustitelného souboru (viz obrázek D.1):



Obrázek D.1: Nastavení NVIDIA Visual Profiler

Po stisknutí Finish se otevře MATLAB, kde je nutné pustit program, který bude analyzovat NVIDIA Visual Profiler.

Příloha E

Literatura

[1] Ing. Jaroslav Vlach. *Metody zpracování obrazu pro časově náročné úlohy*. PhD thesis, Technická Univerzita v Liberci, 2012.

[2] O.Fialka, M. Čadík. FFT and Convolution Performance in Image Filtering on GPU. *In Proceedings of the Tenth International Confetence on Information Visualisation. Los Alamitos: IEEE Computer Society*, 2006.

[3] NVIDIA Corporation. *CUDA C Programming Guide. Version 7.5*, 2015.

[4] Luis H.A Lourenco, D. Weingaerther, E. Told. Efficient implementation of Canny Edge Detection Filter for ITK using CUDA. *WSCAD-SSC '12 Proceedings of the 2012 13th Symposium on Computing Systems*, 2012.

[5] Distance Transform - MATLAB&Simulink. Dostupné na http://www.mathworks.com/help/images/distance-transform.html [on-line] (cit. 24.5.2016).

[6] Způsoby snímání sklaněných kamenů pro jejich následné měření a třídění. Dostupné na http://www.mti.tul.cz/files/obr15 _ atesystem_ volny.pdf [on-line] (cit. 24.5.2016).

[7] Basler ace acA2500-60um - Area Scan Camere. Dostupné na http://www.baslerweb.com/en/products/cameras/area-scan-cameras/ace/aca2500-60um [on-line] (cit. 24.5.2016).

[8] Portable Network Graphics - Wikipedie. Dostupné na https://cs.wikipedia.org/wiki/Portable_Network_Graphics [on-line] (cit. 24.5.2016).

[9] R.C. Gonzalez, R.E. Woods, S.L. Eddins. *Digital Image Processing Using MATLAB*. Pearson Education, Inc., 2004.

[10] M.Sonka, V.Hlavac, R.Boyle. *Image Processing, Analysis, and Machine Vision*. Brooks/Cols Publishing Company, 1999.

[11] M.Klíma, M.Bernas, J.Hozman, P.Dvořák. *Zpracovámí obrazové informace*. Vydavatelství ČVUT, 1999.

[12] M.Šonka, V.Hlaváč. *Počítačové vidění*. Grada, 1992.

[13] J. Canny. A computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-8, No.6*, November 1986.

[14] M. Španěl, V. Beran. Obrazové segmentační techniky. Dostupné na http://www.fit.vutbr.cz/ spanel/segmentace/#_Toc125769325 [on-line] (cit. 24.5.2016).

[15] W.C.Y.Lam, S.Y.Yuen. Efficient technique for circle Detection using hypothesis filtering and Hough transform. *IEE Proc.-Vis. Image Signal Processing, Vol.143, No.5,*, October 1996.

[16] V.K.Yadav, S.Batham, A.K.Acharya, R.Paul. Approach to Accurate Circle Detection: Circular Hough Transform and Local Maxima Concept. *International Conference on Electronics and Communication Systems*, 2014.

[17] H.K.Yuen, J.Princen, J.Illingworth, J.Kittler. A Comparative Study of Hough Transform Methods for Circle Finding. *AVC*, 1989.

[18] C.Kimme, D.Ballard, J.Sklansky. Finding Circle by an Array of Accumulators. *Communications of the ACM, No.2*, 1975.

[19] Tetsuo Asano, Naoki Katoh. Variants for the Hough transform for line detection. *Computational Geometry: Theory and Applications, No.6*, 1996.

[20] Richard O. Duda, Peter E. Hart. Use of the Hough TransTransform to Detect Line and Curves in Pictures. *Graphics and Image Processing*, 1972.

[21] Calvin R. Maurer, Vijay Raghavan. A Linear Time Algorithm for Computing Exact Euclidean Distance Transform of Binary Image in Arbitrary Dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 25, No.2,*, 2003.

[22] K. Horák. Defektoskopie a klasifikace. Dostupné na http://midas.uamt.feec.vutbr.cz/ROZ/Lectures/ [on-line] (cit. 24.5.2016).

[23] Learn LabVIEW: intro to graphical programming in NI LabVIEW. Dostupné na http://www.ni.com/getting-started/labview-basics/ [on-line] (cit. 24.5.2016).

[24] A. Bovik, G. Panayi. SIVA - The Image Processing Gallery. Dostupné na http://live.ece.utexas.edu/class/siva/siva_dip/siva_dip.htm [on-line] (cit. 24.5.2016).

[25] I. Šimeček, J. Sloup. *Programování grafických akcelerátorů*. Vydavatelství ČVUT, 2013.

[26] OpenCL vs. CUDA: Which has better application support? Dostupné na http://create.pro/blog/open-cl-vs-cuda-amd-vs-nvidia-better-application-support-gpgpugpu-acceleration-real-world-face/ [on-line] (cit. 24.5.2016).

[27] NVIDIA Corporation. *CUDA Installation Guide for Microsoft Windows. Version 7.5*, 2015.

[28] Jung W. Suh, Youngmin Kim. *Accelerating MATLAB with GPU Computing.* Elsevier Inc., 2014.

[29] Introduction to GPU Computing with LabVIEW. Dostupné na http://www.ni.com/white-paper/14077/en/ [on-line] (cit. 24.5.2016), 2013.

[30] Create a VC++ DLL in Visual Studio 2012 for mlRC with CUDA. Dostupné na https://github.com/CymatiCorp/CyCuda [on-line] (cit. 24.5.2016).

[31] DLL Tutorial For Beginners. Dostupné na http://www.codeguru.com /cpp/cpp/cpp_mfc/tutorials/article.php/c9855/DLL-Tutorial-For-Beginners.htm [on-line] (cit. 24.5.2016).