VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

GENEROVÁNÍ KÓDU PRO ZPRACOVÁNÍ SUROVÉHO OBRAZU NA GRAFICKÉM ZAŘÍZENÍ

GENERATING CODE FOR PROCESSING THE RAW IMAGE ON A GRAPHICS DEVICE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Jakub Pojsl**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Martin Appel**
SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadaní bakalářské práce

Ústav:	Ústav mechaniky těles, mechatroniky a biomechaniky
Student:	**Jakub Pojsl**
Studijní program:	Aplikované vědy v inženýrství
Studijní obor:	Mechatronika
Vedoucí práce:	**Ing. Martin Appel**
Akademický rok:	2020/21

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Generování kódu pro zpracování surového obrazu na grafickém zařízení

Stručná charakteristika problematiky úkolu:

Vývojová deska NVIDIA Jetson Nano je levné zařízení s dobrým grafickým výkonem, vhodným na provádění grafických operací. Cílem práce je prozkoumat možnosti generování CUDA code z MATLAB GPU coder pro zpracování surových obrazových dat z kamery. Výsledné zařízení by mělo být schopné snímat obrazová data, poté provést různé grafické úpravy a následně výsledný obraz promítnout na snímaný povrch s dostatečnou rychlostí (cca 30FPS).

Cíle bakalářské práce:

1) Seznámit se s NVIDIA Jetson Nano, MATLAB GPU coder.
2) Vytvořit funkce pro zpracování obrazových dat.
3) Vytvořit aplikaci pro NVIDIA Jetson Nano za pomoci MATLAB GPU coder.
4) Po konzultaci s vedoucím práce vytvořit ukázkovou aplikaci pro NVIDIA Jetson Nano, která zpracuje surová obrazová data a výsledný obraz promítne na snímanou plochu.

Seznam doporučené literatury:

NVIDIA Developer [online]. [cit. 2020-10-23]. Dostupné z: https://developer.nvidia.com/

NVIDIA: CUDA Programming Guide, Dostupné z:

http://developer.nvidia.com/object/cuda.html

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2020/21

V Brně, dne

L. S.

prof. Ing. Jindřich Petruška, CSc. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

Abstrakt

Tato bakalářská práce se zabývá možností generování CUDA code pomocí Matlab GPU Coder na grafické zařízení Nvidia Jetson Nano a zpracování hloubkových obrazových dat z Intel RealSence kamery takto vygenerovanými funkcemi. Teoretická část je věnována seznámení s CUDA code, Intel RealSence D415, Jetson Nano a Matlab GPU Coder. Praktická část práce je zaměřena na ukázku generování CUDA code a spustitelného souboru pro Jetson Nano pomocí Matlab GPU Coder a nástroje CMake. Následuje vytvoření aplikace pro získání hloubkových dat z Intel RealSence kamery, jejich zpracování ve vygenerované CUDA funkci a zobrazení zpracovaných dat. Práce je zakončena analýzou běhu aplikace a demonstrací výhod CUDA code při náročných výpočtech.

Summary

This bachelor thesis explores generating CUDA code using Matlab GPU Coder, deploying code to Jetson Nano, and processing depth data from Intel RealSence depth camera with generated CUDA functions. The theoretical part introduces CUDA code, Intel RealSence D415, Jetson Nano, and Matlab GPU Coder. The practical part describes generating CUDA code from simple Matlab functions and generating executables using Matlab GPU Coder and CMake compiler. This is followed by the description of developing the application that gathers depth data from Intel RealSence depth camera, processes the data, and displays the processed data on the screen. Finally, the developed application is analyzed and CUDA code advantages in raw computation are demonstrated.

Klíčová slova

CUDA code, Matlab GPU Coder, Intel RealSence, Nvidia Jetson Nano, CMake

Keywords

CUDA code, Matlab GPU Coder, Intel RealSence, Nvidia Jetson Nano, CMake

Bibliografická Citace

POJSL, J. *Generování kódu pro zpracování surového obrazu na grafickém zařízení.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2021. 44 s., Vedoucí bakalářské práce: Ing. Martin Appel.

Prohlašuji, že tato práce je mým původním dílem, zpracoval jsem ji samostatně pod vedením Ing. Martina Appela a s použitím informačních zdrojů uvedených v seznamu.

Jakub Pojsl

Brno . . . .

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Martinu Appelovi za ochotu a cenné rady při psaní této závěrečné práce. Také bych chtěl poděkovat rodině a blízkým za neutuchající podporu ve studiu.

Jakub Pojsl

Obsah

1 Úvod

Příchod CUDA code, tedy hardwarové a softwarové nadstavby, umožňující spustit C/C++ kód na grafických procesorech, výrazně pomohl využití grafických procesorů pro náročné výpočty a jejich schopnost pracovat s velkými objemy dat dala prostor k vývoji deep learning a AI aplikací. Díky tomu lze v reálných aplikacích využít levnějšího a menšího hardwaru a tím snížit rozměr a cenu výsledného produktu. Při neznalosti programování CUDA code lze využít generátorů kódu. Společnost Mathworks dodává ke svému produktu, programovému prostředí Matlab, rozšíření právě pro generování CUDA code z programovacího jazyka Matlab, Matlab GPU Coder.

Cílem této práce je prozkoumání možnosti využití rozšíření Matlab GPU Coder pro program Matlab při generování CUDA kódu z Matlab funkcí. Běh vygenerovaného CUDA kódu pak bude probíhat na grafickém zařízení Nvidia Jetson Nano Developer Kit.

Jetson Nano je malý, výkonný vývojářský počítač, jehož hlavní využití je na poli AI, IoT a embedded aplikací. S cenou $99 je tak velice dostupnou vstupní bránou do světa těchto aplikací.

Dalším cílem práce je prozkoumat možnosti zpracování hloubkových dat z kamery Intel RealSence D415. Intel RealSence D415 je kamera se standardním zorným polem určená pro vysoce přesné aplikace jako například 3D skenování.

Výstupem by měla být aplikace, která získá data z RealSence kamery, pomocí vygenerovaných CUDA funkcí data zpracuje a zpracovaná data zobrazí na snímaný povrch.

2 Rešerše

2.1 CUDA Code

Mezi lety 2001 a 2003 se frekvence procesoru *Intel Pentium 4* zdvojnásobila, zatímco mezi lety 2003 a 2005 se zvýšila jen o čtvrtinu. Hlavním způsobem, jak zvyšovat procesorům výkon bylo zvýšit jejich operační frekvenci. Jelikož výrobci dosahovali tepelných a výkonových limitů, nemluvě o limitu fyzické velikosti tranzistoru, museli najít jiné způsoby, jak by každá nová generace měla přinést vyšší výkon. V roce 2005 začali do procesorů přidávat další jádra – dvoujádrové procesory. V průběhu následujících let se počty jader začaly zvyšovat, dnes jsou pro běžného spotřebitele k dispozici i procesory 16jádrové [1].

Tyto více jádrové procesory spoléhaly na paralelizaci. Při paralelizaci výpočtů více jader procesoru pracuje na jednom výpočtu souběžně. Komplikovaný výpočet se rozloží na jednodušší parciální výpočty, které jsou souběžně zpracovávány [3].

Paralelizace využívají také grafické karty. Jsou sestaveny z až tisíců jader a určeny k rychlým aritmetickým výpočtům a práci s velkým množstvím dat [4]. Díky této specializaci se více hodí na řešení algebraických problémů, deep learning a AI aplikace.

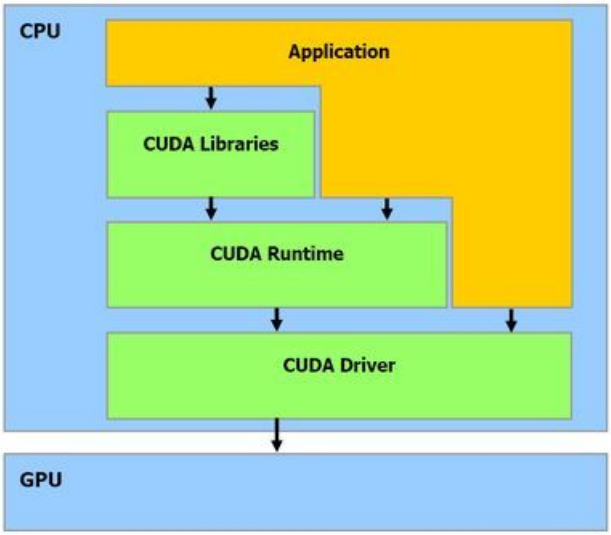
První pokusy o akcelerování matematických výpočtů grafickými akcelerátory jsou z devadesátých let minulého století. S příchodem BrookGPU v roce 2003 se velice usnadnil přístup k výpočetnímu potenciálu GPU. Brook byl set rozšíření pro programovací jazyk C, který měl zjednodušit využití GPU pro náročné a datově rozsáhlé úkoly [5]. Obsahoval kompilátor, který zkompiloval .br soubor do standardního C++ kódu spojený s run-time knihovnou s různými backendy jako DirectX a OpenGL.

Problém BrookGPU byla ale kompatibilita. Výrobci grafických karet často optimalizují jejich drivery. To ale znamenalo, že s novou aktualizací driveru byl narušen přístup Brook k GPU.

Úspěch BrookGPU zaznamenali výrobci grafických čipů ATI a Nvidia. Uviděli příležitost pro různorodé využití svých grafických čipů a začali pracovat na naplnění jejich potenciálu. V roce 2007 společnost Nvidia vydala první verzi CUDA. CUDA je rozšíření pro programovací jazyk C++ a poskytuje tři vrstvy pro komunikaci s GPU, viz obrázek 2.1, – CUDA libraries, CUDA runtime API a CUDA Driver API. CUDA Driver API a CUDA runtime API mohou být zaměnitelné, ale existují mezi nimi klíčové rozdíly. Při používání runtime API je zdrojový kód jednodušší, zato driver API nám poskytuje větší kontrolu nad chodem aplikace. Třetí vrstvou jsou CUDA knihovny [11]:

- **Matematické knihovny**
  - cuBLAS – základní lineární algebra,
  - cuFFT – rychlá Fourierova transformace,
  - CUDA Math library – standardní matematické funkce,
  - cuRAND – generování pseudonáhodných čísel,
  - cuSOLVER – řešení řídkých matic,
  - cuTENSOR – tenzorová lineární algebra,

- **Obrazové a video knihovny**
- **Deep learning knihovny**
  - cuDNN – základní knihovna pro deep neural networks,
  - TensorRT – rozšířená knihovna pro deep neural networks,
- **a další ...**



Obrázek 2.1: Vrstvy pro komunikaci s GPU [1]

Nadále dochází k přidávání dalších knihoven ze strany Nvidia v nových vydáních CUDA, stejně jako podpora nového hardwaru a přidávání nových funkcí. CUDA používá vlastní kompilátor *nvcc*, který je součástí NVIDIA CUDA Toolkitu, přes který probíhá instalace CUDA.

2.2 Intel RealSence D415

Intel RealSence je technologie počítačového vidění zaměřená na sledování objektů a vnímání hloubky. Toho je dosaženo zejména kamerami s hloubkovým senzorem. Nejaktuálnější produkty s touto technologií spadají do řady D400. Kamery používají metodu aktivního stereo snímání, jejímž funkčním základem je zpracování obrazu ze dvou kamer ve stejné výšce se známou vzdáleností od sebe. Díky tomu je kamera schopna získat vzdálenost jednotlivých pixelů a na jejich základě vytvořit hloubkovou mapu. Pro zlepšení výsledků kamery také obsahují infračervený projektor, který promítá na snímanou plochu známý vzor, který poté pomáhá při zpracování dvou obrazů [23] [22].

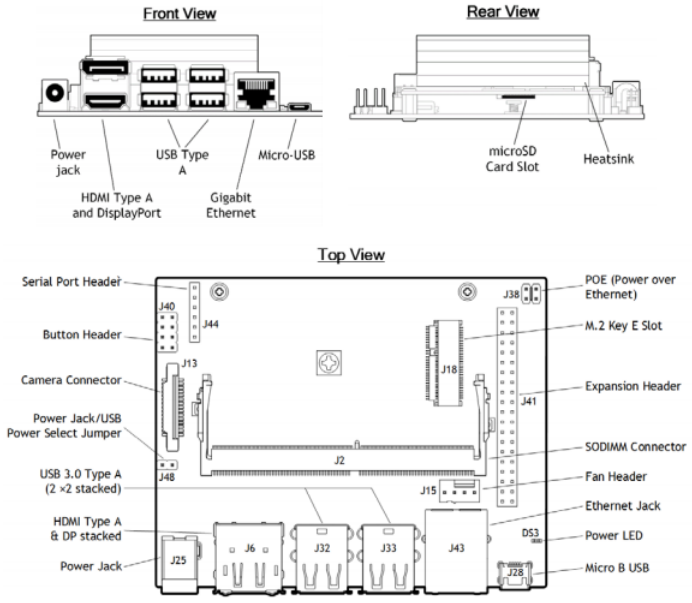
Pro práci s kamerou Intel nabízí Intel RealSence SDK 2.0. Tato knihovna, dostupná na Linux, Windows a Mac OS, umožňuje stream hloubkových a barevných dat z kamery a kalibraci kamery. Knihovna také nabízí syntetický stream, tedy stream pointcloud dat a zarovnaných hloubkových a barevných dat kamery. Knihovna nativně podporuje programovací jazyky C a C++, ale nabízí rozšíření pro Python, Matlab a další. Existuje mnoho příkladů a dokumentace pro práci s knihovnou [21].

2.3 Jetson Nano

V březnu 2019 Nvidia představila Jetson Nano, čtvrtý počítač z rodiny Jetson. Jedná se o kompletní systém SOM s Tegra SOC, jehož hlavní uplatnění se nachází ve vývoji nových systémů a aplikací. Jetson Nano je nejlevnější, s doporučenou cenou 99 USD za verzi s 4GB paměti RAM. Technické specifikace jsou zobrazeny v tabulce 2.1.

GPU	128-core Maxvell
CPU	Quad-core ARM A57 @ 1.43 GHz
RAM	4 GB 64-bit LPDDR4 25.6 GB/s
Úložiště	microSD slot
Video enkódování	4K @ 30 | 4x 1080p @ 30 | 9x 720p @ 30 (H.264/H.265)
Video dekódování	4K @ 60 | 2x 4K @ 30 | 8x 1080p @ 30 | 18x 720p @ 30 (H.264/H.265)
Kamera	2x MIPI CSI-2 DPHY lanes
Konektivita	Gigabit Ethernet, M.2 Key E
Výstup videa	HDMI and display port
USB	4x USB 3.0, USB 2.0 Micro-B
Ostatní	GPIO, I^2C, I^2S, SPI, UART
Rozměry	100mm x 80mm x 29mm

Tabulka 2.1: Technické specifikace Jetson Nano



Obrázek 2.2: Schéma Jetson Nano [25]

Nvidia k Jetsonům dodává Nvidia JetPack SDK. JetPack obsahuje Jetson Linux drivery, linuxovou distribuci Ubuntu, CUDA akcelerované knihovny a API pro deep learning a další vývojářské nástroje. Také obsahuje dokumentaci a ukázkové projekty.

Pro instalaci Nvidia Jetpack je potřeba vypálit na micro SD kartu, např. pomocí programu Rufus, image JetPack pro Jetson Nano, který je dostupný na webových stránkách developer.nvidia.com.

Napájení Jetsonu je možné zajistit čtyřmi způsoby – PoE, Micro B USB, Power Jack, nebo pomocí pinů 4+6. Napájecí zdroj se s Jetson Nano nedodává, v případě Micro B USB je potřeba

zdroj, který dodává až 2A při 5V, pro zdroj Power Jack konektorem je doporučeno 4A při 5V a je potřeba zkratovat dvojici pinů J48 [6].

Po vložení SD karty s JetPackem, spuštění Jetsonu a připojení periferií jako monitoru, klávesnice a myši následuje průvodce prvním spuštěním systému Ubuntu.

2.4 Matlab GPU Coder

Matlab GPU Coder je rozšíření programu Matlab generující optimalizovaný CUDA kód z Matlab kódu a Simulink modelů. Jeho hlavní využití je v oblasti deep learning, embedded aplikacích, computer vision a autonomních systémech. Vygenerovaný kód využívá optimalizované CUDA knihovny, jako cuDNN, cuSOLVER, cuBLAS a TensorRT. Umožňuje generování zdrojového CUDA kódu, statických či dynamických knihoven, MEX funkcí, nebo přímo spustitelných souborů. Pomocí MEX souborů můžeme přímo v Matlabu akcelerovat náročné výpočty [9].

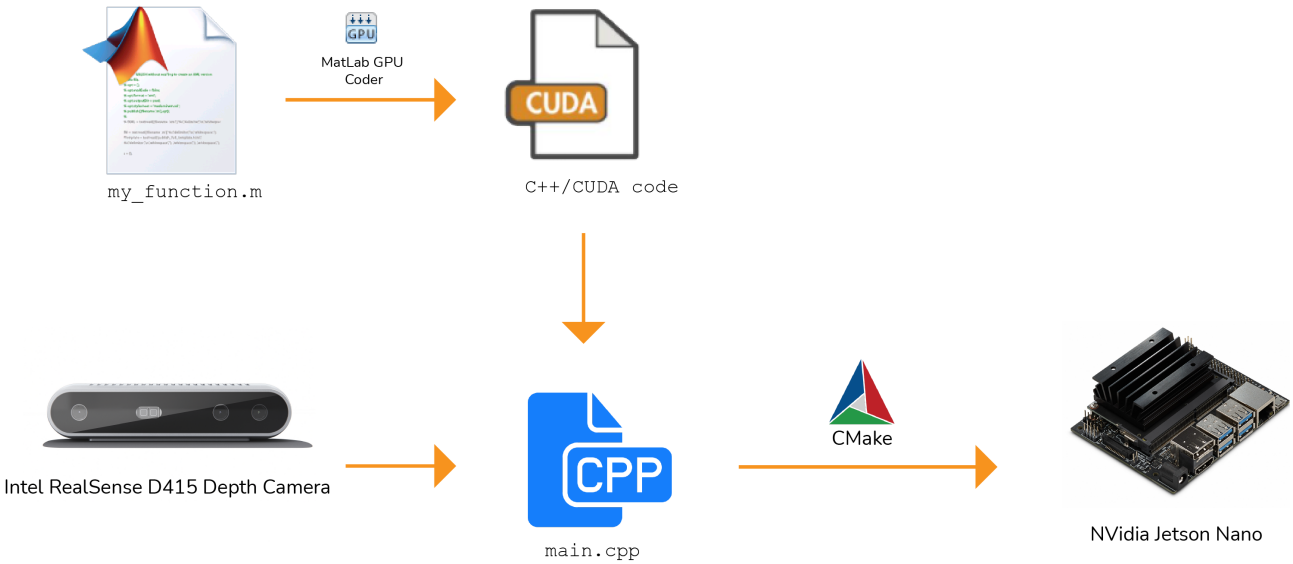
3 Cíle řešení

3.1 Generování CUDA kódu pomocí Matlab GPU Coder

V první řadě je nutné připravit Jetson Nano na generování kódu, nainstalovat potřebné knihovny a připojit jej k programu Matlab. Dále je třeba prozkoumat vzájemné fungování programu Matlab a Jetson Nano. Proběhne seznámení s Matlab GPU coderem, demonstrace generování CUDA code na jednoduchých funkcích a spuštění vygenerovaného kódu na Jetson Nano.

3.2 Ukázková aplikace

Cílem aplikace je získání dat z hloubkové kamery Intel RealSence D415 a zobrazení vrstevnic snímaného terénu. Terénem se myslí nádoba s pískem, ve kterém budou vytvořeny určité terénní nerovnosti. Vrstevnice bychom poté na písek promítali.



Obrázek 3.1: Schéma vytvoření ukázkové aplikace

Pro získání dat z kamery je zapotřebí knihovna `librealsence2`, dostupná na vývojářské platformě Github. Jelikož mi není znám způsob, jak do Matlab GPU Coder přidat externí knihovnu, nemůžeme použít pro vygenerování executable souboru na Jetson Nano přímo Matlab GPU Coder, ale budeme muset pro finální vygenerováni aplikace použít nástroj CMake. V programu budeme volat CUDA funkci vygenerovanou aplikací Matlab GPU Coder. Schéma postupu vytvoření aplikace je zobrazeno na obrázku 3.1.

4 Postup řešení

4.1 Konfigurace Jetsonu Nano

Abychom mohli s Jetsonem Nano pracovat v prostředí Matlab, musíme mít nainstalované následující Matlab balíčky:

- Matlab Coder,
- Parallel computing toolbox,
- GPU Coder Support package pro Nvidia GPU's,
- Deep Learning toolbox.

Na Jetsonu Nano musíme mít nainstalovaný Nvidia JetPack a Simple DirectMedia Layer knihovnu. Knihovnu nainstalujeme pomocí následujícího příkazu v terminálu [8]:

```
$ sudo apt-get install libsdl1.2debian
$ sudo apt-get install libsdl1.2-dev
```

Musíme správně nastavit proměnné prostředí (environmental variables):

```
$ sudo nano .bashrc
```

Přidáme řádky 4-6 [10]:

```
1 case $- in
2     *i*) ;;
3       *)
4             export PATH=${PATH}:/usr/local/cuda-10.2/bin
5             export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/cuda-10.2/lib64
6             export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/lib/aarch64-linux-gnu
7           return;;
```

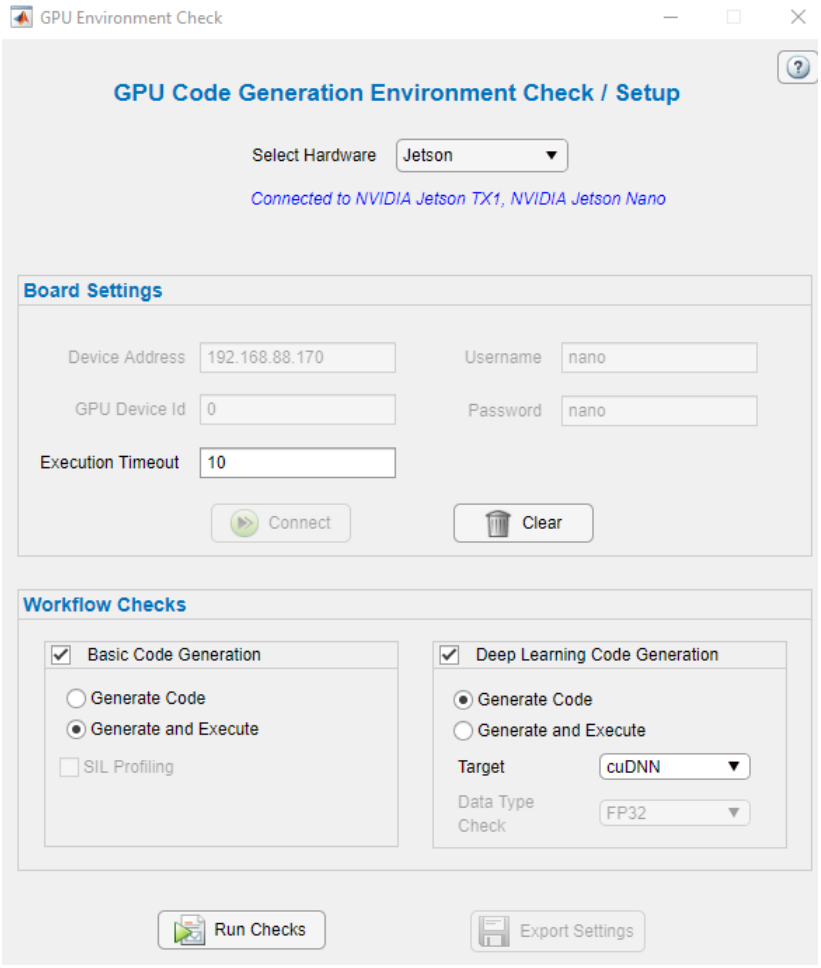
Na počítači, na kterém běží Matlab, musíme mít nainstalovaný C/C++ kompilátor, například Microsoft Visual Studio 2019. Počítač také musí mít Nvidia grafickou kartu podporující CUDA. Připojení Jetsonu Nano k počítači, na kterém máme spuštěný Matlab, probíhá přes SSH. Pro připojení tedy potřebujeme IP adresu Jetsonu v lokální síti, v Matlabu:

```
1 ipaddress = '192.168.88.170';
2 username = 'nano'; % jmeno uctu Ubuntu
3 password = 'nano'; % heslo k uctu
4 hwobj = jetson(ipaddress, username, password);
```

Pomocí těchto příkazů v Matlabu ověříme, zda jsou oba systémy připraveny na generování kódu:

```
gpuEnvObj = coder.gpuEnvConfig('jetson');
gpuEnvObj.BasicCodegen = 1;
gpuEnvObj.BasicCodeexec = 1;
gpuEnvObj.DeepLibTarget = 'tensorrt'; % tensorrt nebo cudnn
gpuEnvObj.DeepCodeexec = 1;
gpuEnvObj.DeepCodegen = 1;
gpuEnvObj.HardwareObject = hwobj;
results = coder.checkGpuInstall(gpuEnvObj);
```

Alternativně můžeme použít příkaz `coder.checkGpuInstallApp`, který otevře okno aplikace (viz obrázek 4.1), ve kterém zadáme potřebné parmetry pro připojení k Jetsonu a zvolíme, jaké testy chceme provést.



Obrázek 4.1: GPU Environment Check v prostředí Matlab

4.2 Generování CUDA code

Generování v Matlab GPU Coderu můžeme uskutečnit dvěma způsoby. Můžeme použít grafické prostředí (GUI) Matlab GPU coderu, nebo můžeme kód generovat pomocí několika příkazů. Je ale značně jednodušší použít GUI, které nám nakonec může příkazy pro generování vytvořit.

Nejprve vytvoříme ukázkovou funkci `test_fcn.m`, která vrací $cos(a)$ vstupní hodnoty.

```
1 function [b] = test_fcn(a)
2     coder.gpu.kernelfun();
3     b = cos(a);
4 end
```

Kód 4.1: Ukázková Matlab funkce pro vygenerování CUDA kódu

Za povšimnutí stojí funkce `coder.gpu.kernelfun()` na řádku 2, v kódu 4.1, která se snaží pro všechny výpočty ve funkci použít GPU.

Pro otevření aplikace Matlab GPU Coder můžeme použít příkaz `gpucoder`. Aplikace obsahuje průvodce generováním kódu. V prvním kroku zvolíme soubor funkce, pro kterou chceme CUDA kód generovat.

V dalším kroku zvolíme vstupy funkce. V jazyku C++ musí mít všechny funkce pevně stanovené typy vstupů. Pro automatické zvolení vstupů funkce GPU Coderem funkci zavoláme v příkazovém řádku GPU Coderu.

```
>> test_fcn(1:100)
```

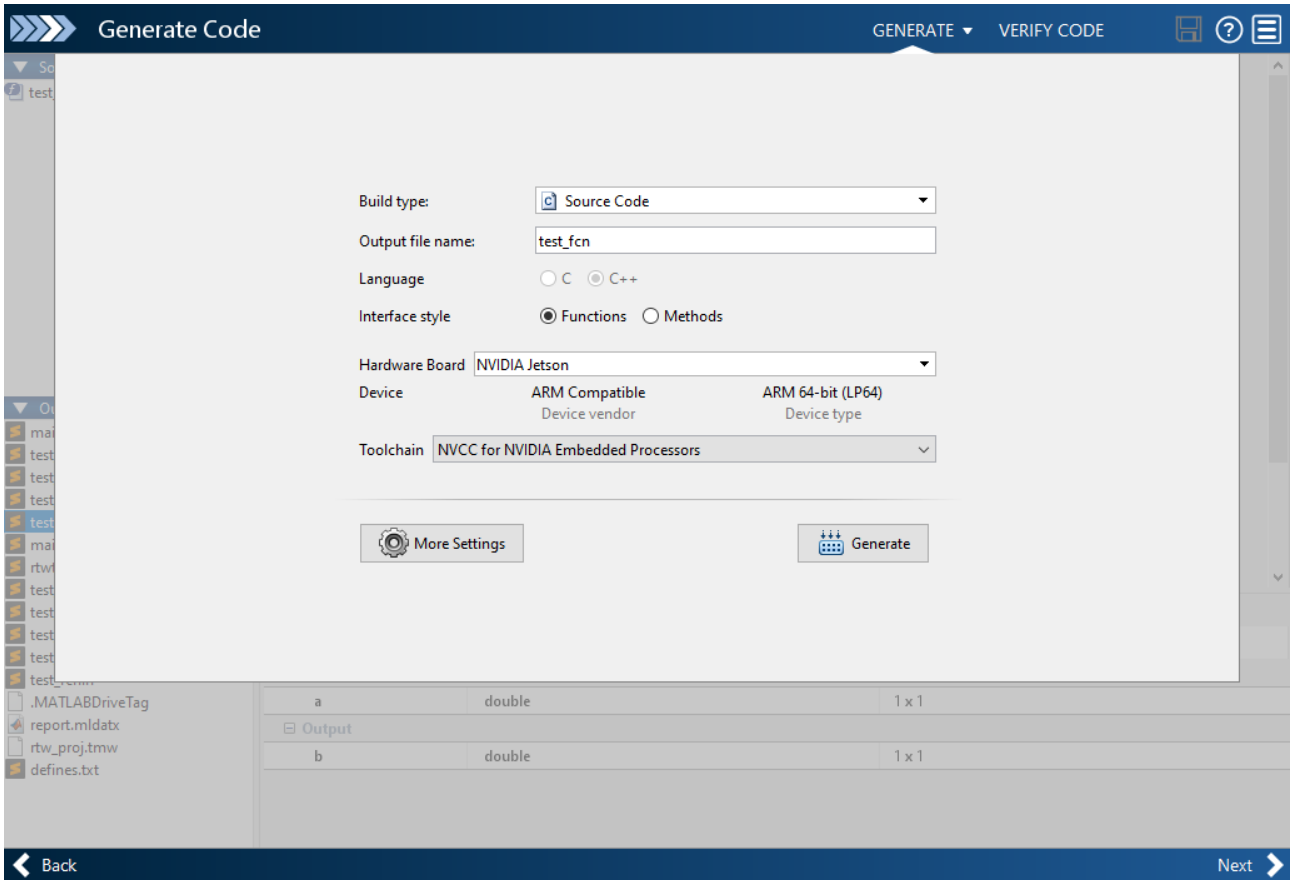
Vstup jsme tedy definovali jako vektor obsahující 100 prvků. GPU Coder vstup CUDA funkce zvolil typu double(1x100).

V dalším kroku Matlab GPU Coder otestuje na MEX funkci, kterou vygeneruje, jak funkce na GPU běží. Stejným příkazem jako v předchozím kroku funkci zavoláme.

V dalším kroku, viz obrázek 4.2, volíme typ generování. Můžeme zvolit MEX soubor, který použijeme v Matlabu pro akcelerování výpočtů GPU, statickou, nebo dynamickou knihovnu, zdrojový kód, ze kterého následně vygenerujeme spustitelný soubor, např. pomocí programu CMake, což je popsáno v kapitole 4.2.2. Nakonec můžeme generovat přímo spustitelný soubor.

Nejprve musíme vygenerovat zdrojový kód CUDA a zvolíme jako *Hardware Board* NVIDIA Jetson a jako *Toolchain* NVCC for NVIDIA Embedded Processors. Dále zadáme přes tlačítko *More Settings* a *Hardware* IP adresu Jetson Nano, uživatelské jméno a heslo a adresář, ve kterém Matlab bude na Jetson Nano pracovat. Nyní můžeme vygenerovat zdrojový kód CUDA pro zvolenou Matlab funkci tlačítkem *Generate*.

Zdrojový kód nyní nalezneme ve vytvořeném adresáři *codegen* v aktuální pracovní složce Matlabu. V ní nalezneme i ukázkový soubor `main.cu`, který, po určitých úpravách, použijeme pro sestavení spustitelného souboru.



Obrázek 4.2: Generování kódu v prostředí Matlab

```
1  static void argInit_1x100_real_T(double result[100])
2  {
3    for (int idx1 = 0; idx1 < 100; idx1++) {
4      result[idx1] = idx1;
5    }
6  }
7
8  ...
9
10 static void main_test_fcn()
11 {
12   double a[100];
13   double b[100];
14
15   argInit_1x100_real_T(b);
16   test_fcn(b, a);
17   writeToFile(a);
18 }
```

Kód 4.2: Část ukázkového souboru `main.cu`, vygenerovaného Matlab GPU coderem

4.2.1 Sestavení pomocí Matlab GPU Coder

K sestavení spustitelného souboru na Jetson Nano můžeme použít přímo Matlab GPU Coder. Pro jeho vytvoření musíme dodat soubor `main.cu`, ve kterém budeme vygenerovanou funkci volat. Upravíme tedy vygenerovaný ukázkový soubor `main.cu`, který jsme vygenerovali v kapitole 4.2.

Matlab v ukázkovém `main.cu`, viz kód 4.2, vytvoří funkci `argInit_1x100_real_T()`, která ve `for()` cyklu dosadí do vstupního vektoru *b* funkce `test_fcn()` číselné hodnoty. Do vektoru *b* tímto způsobem dosadíme např. hodnoty 1 až 100. Poté se zavolá Matlabem vygenerovaná CUDA funkce `test_fcn()`, u které je výstupem vektor *a*. Nakonec můžeme uložit hodnoty ve vektoru *a* do souboru.

Nyní v Matlab GPU Coderu při volbě *Build type* zvolíme možnost *Executable* a přes tlačítko *More Settings* a *Custom Code* v kolonce *Additional source files* přidáme námi upravený main soubor, který uložíme do aktuálního pracovního adresáře Matlabu.

Po vygenerování se na Jetson Nano v adresáři `/home/nano/`, kde *nano* je název uživatelského účtu, ve složce, kterou jsme definovali v minulém kroku, objeví vygenerovaný spustitelný soubor. Můžeme tedy v adresáři na Jetson Nano otevřít příkazové okno a pomocí příkazu `./test_fcn.elf` soubor spustit.

Kód 4.3 ukazuje možnost soubor spustit přímo z prostředí Matlab. Na prvním řádku se připojíme k Jetson Nano. Vložíme v proměnné *exe* spustitelný soubor na Jetsonu Nano a funkcí `runExecutable()` jej spustíme. Výsledné hodnoty se nám uloží do souboru `test_fcn.bin`, který funkcí `getFile()` překopírujeme z Jetson Nano do aktivního adresáře Matlabu. Do proměnné *Out* uložíme hodnoty ze souboru `test_fcn.bin` [17].

```
1 hwobj = jetson('192.168.88.170','nano','nano');
2 exe = [hwobj.workspaceDir '/test_fcn.elf'];
3 procID = runExecutable(hwobj,exe);
4 outputFile = [hwobj.workspaceDir '/test_fcn.bin'];
5 getFile(hwobj,outputFile);
6 fId = fopen('test_fcn.bin', 'r');
7 Out = fread(fId, 'double');
```

Kód 4.3: Script pro spuštění vygenerované aplikace na Jetson Nano v prostředí Matlab

4.2.2 Kompilace nástrojem CMake

Pro kompilaci zdrojového kódu z Matlab GPU Coderu využijeme nástroj CMake. CMake je open-source sada nástrojů, která zajišťuje build systému na různých platformách [12].

CMake nejdříve konfiguruje vstupy potřebné pro vygenerování build systému. Jedná se o různé volby a přístup ke knihovnám. V další fázi generuje build systému na základě nakonfigurovaných informací. Vstupy CMake určujeme v souboru `CMakeLists.txt` v kořenovém adresáři projektu [14].

Nejdříve CMake nainstalujeme na Jetson Nano, např. pomocí návodu z příspěvku [13].

Vytvoříme kořenovou složku projektu a vygenerovaný zdrojový kód z Matlab GPU Coder do ní přesuneme. Při generování kódu, při volbě vstupu funkce zadáme vstup skalár, Matlab tedy jako vstup vygenerované funkce zvolí typ double(1x1). Vytvoříme soubor `CMakeLists.txt`.

```
1 cmake_minimum_required(VERSION 3.1)
2 project(cos_test)
3 find_package(CUDA REQUIRED)
4 set(CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS} -gencode arch=compute_53,code=sm_53)
5 file( GLOB  cu  *.cu)
6 cuda_add_executable(${CMAKE_PROJECT_NAME} main.cpp ${cu})
```

Kód 4.4: `CMakeLists.txt`

Na řádku 2 v kódu 4.4 zvolíme jméno projektu. Na dalším řádku najdeme CUDA knihovnu na zařízení. Řádek 4 určuje, jakým způsobem budeme CUDA soubory kompilovat. Pro každé zařízení se proměnné *arch* a *code* nastavují jinak. Pro Jetson Nano mají uvedené hodnoty [15]. Na řádku 5 hledáme všechny CUDA soubory, které mají příponu *.cu*. Tyto soubory se kompilují zvlášť pomocí *nvcc* kompilátoru. Na řádku 6 přidáme main soubor aplikace a už zkompilované CUDA soubory.

Vytvoříme main soubor aplikace. Jako vzor použijeme ukázkový main, který vygeneroval Matlab GPU Coder. V kódu 4.5 vidíme upravenou funkci `main_test_fcn()`.

```
1 #include <iostream>
2 static void main_test_fcn()
3 {
4   double a, b;
5   std::cin >> a;
6   b = test_fcn(a);
7   printf("Jeho cos je %f\n", b);
8 }
```

Kód 4.5: Upravená `main_test_fcn()` v `main.cu`

Vyžádáme od uživatele číselný vstup, který použijeme jako vstup do vygenerované CUDA funkce. Výsledek vypíšeme pomocí `printf()`, musíme tedy přidat knihovnu `<iostream>`.

Nyní otevřeme příkazový řádek v kořenovém adresáři projektu, vytvoříme složku build a v ní vytvoříme spustitelný soubor:

```
~/cos_test$ mkdir build
~/cos_test$ cd build
~/cos_test/build$ cmake ..
~/cos_test/build$ make
~/cos_test/build$ ./cos_test
3.14159264
Jeho cos je -1.000000
```

Posledním příkazem soubor spustíme.

4.3 Vytvoření aplikace

4.3.1 Instalace knihovny librealsence2 na Jetson Nano

Pro sestavování programů s knihovnou realsence2 je doporučeno použít nástroj CMake, který také použijeme k sestavení knihovny na Jetson Nano. Nejdříve stáhneme zip archiv knihovny z vývojářské platformy Github. Soubor extrahujeme do složky a v ní vytvoříme složku `/build`. V terminálu poté librealsence2 sestavíme [16]:

```
~/build$ cmake ../ -DFORCE_RSUSB_BACKEND=ON -DBUILD_PYTHON_BINDINGS:bool=true
-DCMAKE_BUILD_TYPE=release -DBUILD_EXAMPLES=true -DBUILD_GRAPHICAL_EXAMPLES=true
-DBUILD_WITH_CUDA:bool=true
~/build$ make -j4
~/build$ sudo make install
```

Pomocí příkazu `realsence-viewer` spustíme aplikaci, která je schopná zobrazit data z kamery a ověříme úspěšnou instalaci knihovny.

4.3.2 Získání dat z RealSence kamery

Ve funkci `main` nejdříve definujeme `rs2::pipeline`, nastavíme rozlišení hloubkového senzoru a zahájíme stream dat z kamery:

```
1    rs2::pipeline p;
2    rs2::colorizer color_map;
3    rs2::config cfg;
4    cfg.enable_stream(RS2_STREAM_DEPTH, 1280, 720);
5    rs2::pipeline_profile profile = p.start(cfg);
```

Dále zjistíme měřítko, ve kterých nám kamera bude posílat hodnoty vzdálenosti jednotlivých pixelů. Nyní můžeme uložit metodou `get_depth_frame()` první hloubkový snímek:

```
1    float depth_scale = get_depth_scale(profile.get_device());
2    rs2::frameset frames = p.wait_for_frames();
3    rs2::depth_frame depth = frames.get_depth_frame();
```

Hloubkový snímek chceme uložit do matice 1280x720, ve které budou prvky matice reprezentovat vzdálenost kamery od jednotlivých pixelů. Jelikož C++ neumí úplně dobře pracovat s maticemi, hodnoty budeme ukládat do vektoru typu double `pixel_dist[921600]`, který bude mít řádky matice poskládané v řadě za sebou. Metodou `get_width()` a `get_height()` získáme rozlišení streamovaných dat z kamery:

```
1    int width = depth.get_width();
2    int height = depth.get_height();
3    float max_distance = 5;
4    double pixel_distance[921600];
```

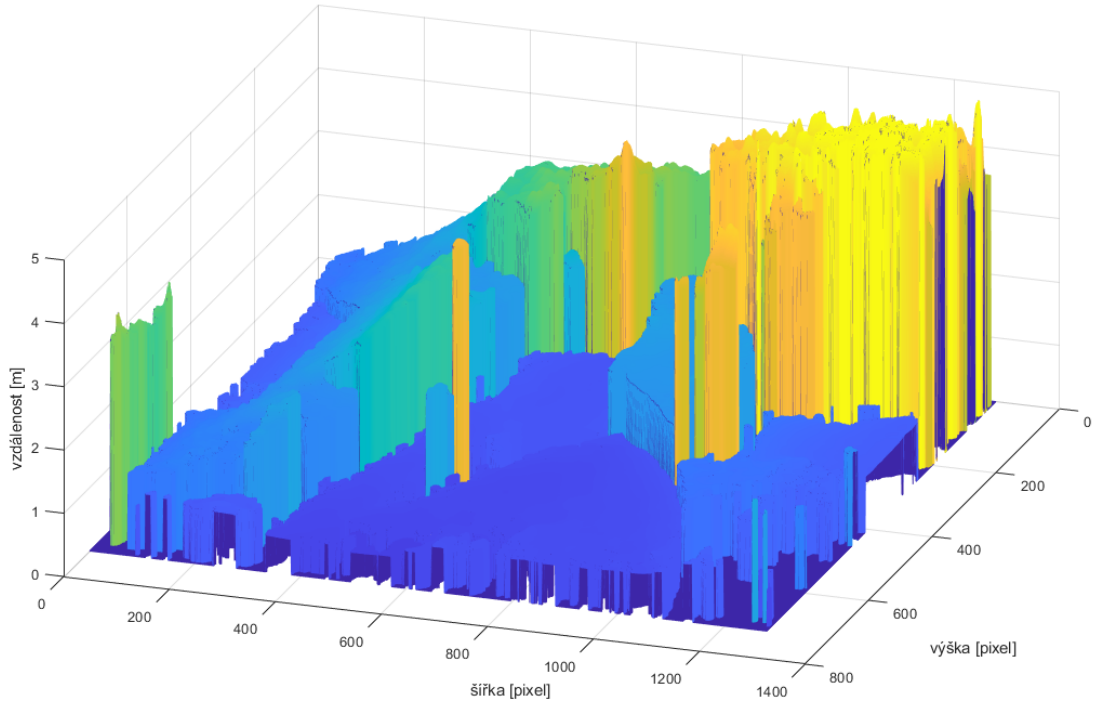
Nakonec budeme iterovat přes prvky vektoru `pixel_dist` a uložíme do něj hloubková data ze získaného snímku. Navíc filtrujeme hodnoty vzdáleností, které nepatří do intervalu $\langle 0; 5 \rangle$:

```
const uint16_t* p_depth_frame = reinterpret_cast<const uint16_t*>(depth.get_data());
for (int y = 0; y < height; y++){
    auto pixel_index = y * width;
    for (int x = 0; x < width; x++, ++pixel_index)
    {
        pixel_distance[pixel_index] = depth_scale * p_depth_frame[pixel_index];
        if (pixel_distance[pixel_index] < 0.f || pixel_distance[pixel_index] > max_distance)
        {
            pixel_distance[pixel_index] = 0;
        }
    }
```

Vektor `pixel_distance` uložíme do souboru, a načteme jej v Matlabu, abychom měli data, se kterými můžeme v Matlabu pracovat. Vektor `pixel_distance` bude vstupem do CUDA funkce, kterou v Matlab GPU Coder vygenerujeme. Kompletní zdrojový kód pro zapsání hloubkových dat z kamery do souboru je v příloze 9.1.

4.3.3 Zpracování dat z RealSence kamery

Hodnoty, které jsme získali v kapitole 4.3.2 nahrajeme do Matlab Workspace a můžeme je vykreslit. Vektor `A`, do kterého hodnoty ve Workspace uložíme musíme převést, např. pomocí funkce `reshape()`, na matici 1280x720. Následně matici vykreslíme funkcí `mesh()`, jako na obrázku 4.3.



Obrázek 4.3: Zobrazení surových dat z hloubkové kamery pomocí funkce `mesh()`.

Cílem zpracování dat je rozdělit body podle jejich výšky do určitých intervalů a každý interval zobrazit jinou barvou. Matlab má vestavěnou funkci `contour()` pro vytvoření vrstevnic, Matlab GPU Coder ji ale nepodporuje, budeme ji muset tedy vytvořit. Zdrojový kód funkce `opencv_contour()` je v příloze 9.2. Vstupem funkce v Matlabu bude vektor `depth_data`, do kterého se při zavolání uloží hodnoty vzdáleností pixelu od kamery daného snímku.

Ve funkci, kterou nazveme `opencv_contour()`, nejdříve definujeme počet intervalů, do kterých budeme výšky pixelů přiřazovat. Jelikož filtrujeme hodnoty z kamery větší než 5 m, přiřadíme proměnné `levels` hodnotu 20, což odpovídá 19 intervalům.

Nyní vygenerujeme samotné intervaly výšek, které budeme porovnávat s hodnotami vektoru `depth_data`. Toho dosáhneme funkcí `linspace()` a budeme generovat od minimální do maximální hodnoty z vektoru `depth_data`. Výstup funkce `linspace()` uložíme do proměnné `height`.

Dále musíme definovat barvy, které pro daný interval vykreslíme. Funkce `flip(turbo(20))` vrátí matici 20x3, ve které každý řádek odpovídá barvě a každý sloupec jednotlivé složce RGB spektra. Dostaneme tedy barevnou mapu s dvaceti barvami, které se nějakým způsobem mění. Tento způsob je daný typem *Matlab colormap array*. Matlab nabízí 18 přednastavených barevných map, ze kterých se pro naši aplikaci nejvíce hodí barevná mapa *Turbo*. Funkci `flip()` použijeme z důvodu, který zmíním v kapitole 4.3.4.

Vytvoříme novou funkci `contour_colormap()`, která vrátí matici, kterou jsme vygenerovali funkcí `turbo()`. Přímo funkci `turbo()` ve funkci `opencv_contour()` použít nemůžeme, protože ji Matlab GPU Coder nepodporuje. Jelikož Matlab používá pro složky RGB hodnoty z intervalu $\langle 0; 1 \rangle$, celou matici barevné mapy vynásobíme po prvcích číslem 254, abychom dostali hodnoty RGB z intervalu $\langle 0; 254 \rangle$. Barevnou mapu uložíme ve funkci `opencv_contour()` do proměnné `color_map`.

Výstupem funkce bude vektor `cont_mat`, který pro každý pixel z hloubkového snímku kamery bude mít RGB informaci o barvě daného pixelu. Výstupní vektor proto musí být 3x větší než vstupní, tedy $1280 \times 720 \times 3 = 2764800$ prvků, kde trojice prvků vedle sebe obsahuje RGB informaci o jednom pixelu.

```
1     j = 1;
2     for i = 1:3:len*3
3         if(depth_data(j) == 0)
4             cont_mat(i:(i+2)) = [0 0 0];
5         else
6             for k = 1:levels
7                 if(height(k) >= depth_data(j))
8                     cont_mat(i:(i+2)) = color_map(k, :);
9                     break;
10                end
11            end
12        end
13        j = j + 1;
14    end
```

Kód 4.6: Cyklus ve funkci `opencv_contour()`

Při přiřazování barev jednotlivým pixelům budeme iterovat přes délku `cont_mat` ob tři prvky v cyklu `for()`, jako v ukázce kódu 4.6. Vytvoříme podmínku, pokud bude hodnota ve snímku 0, nastavíme barvu na černou. Poté budeme v pod-cyklu iterovat přes úrovně výšek

a porovnávat každou s hodnotou v konkrétním pixelu. Výšku každého pixelu tedy porovnáme s horní hranicí intervalů výšek. Jakmile bude podmínka splněna, hodnota z barevné mapy pro odpovídající výškový interval se uloží do odpovídajícího pixelu výstupního vektoru `cont_mat`, a zároveň příkazem `break` ukončíme exekuci pod-cyklu.

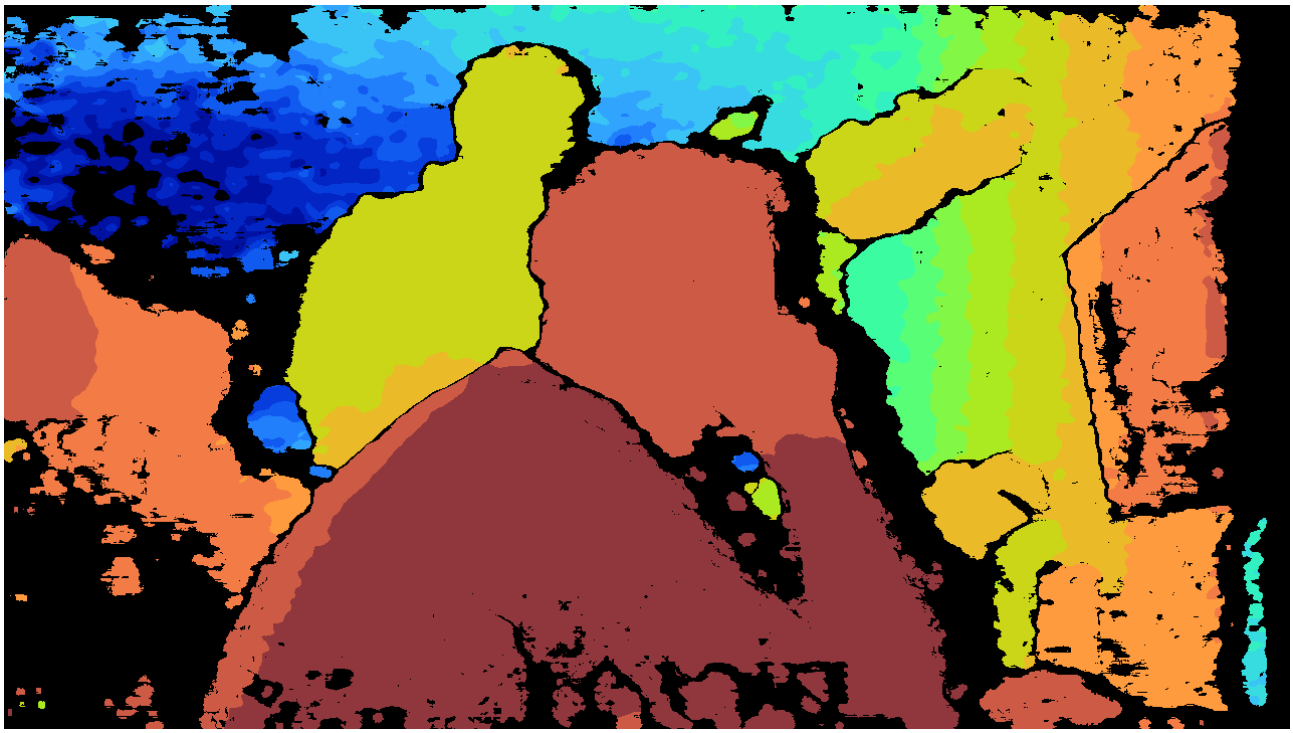
Pro otestování funkčnosti kódu v prostředí Matlab vytvoříme skript, který načte testovací snímek a zavolá funkci `opencv_contour()`. Abychom mohli v Matlabu obrázek zobrazit, musíme ještě vytvořit funkci, která hodnoty barev pixelů převede z intervalu $\langle 0; 255 \rangle$ do intervalu $\langle 0; 1 \rangle$ a přehodí modrý a červený barevný kanál.

Na obrázku 4.4 je výstup funkce `imshow()`, kterou jsme v Matlabu pro zobrazení obrázku použili. Červenou barvou jsou zobrazeny plochy nejblíže kameře a přes přes zelenou se dostaneme k nejvzdálenějším plochám, které jsou zobrazeny modrou barvou. Poměrně velká část obrázku je černá. Černou barvu jsme přiřadili pixelům, ze kterých jsme dostali špatná data. Černé jsou také ty pixely, u který kamera nebyla schopna určit vzdálenost. Podrobnosti o vylepšení obrazu post-processingem jsou v kapitole 4.3.6.

Kód vygenerujeme pomocí Matlab GPU Coderu a zvolíme generování zdrojového kódu. Po vygenerování v posledním kroku generování zvolíme možnost *Package*. Tato možnost nám dovolí uložit všechny soubory, které potřebujeme pro vygenerování spustitelného souboru v archivu.

V C++ budeme funkci volat a ukládat data do proměnné `cont_mat`:

```
1 double cont_mat[2764800];
2 opencv_contour(pixel_distance, cont_mat);
```



Obrázek 4.4: Zpracovaná ukázková data v Matlabu.

4.3.4 Zobrazení obrazu v OpenCV

Zobrazení vrstevnic z funkce `opencv_contour()` provedeme čistě v C++ a použijeme na to knihovnu OpenCV (Open Source Computer Vision Library). OpenCV je open-source knihovna, která obsahuje několik stovek algoritmů pro počítačové vidění [18]. Knihovna OpenCV je součástí systému Jetpack na Jetson Nano, takže ji nemusíme instalovat.

V OpenCV se obrázky ukládají do struktury `cv::Mat`, což je matice, která předepsaným způsobem ukládá informace o pixelech obrázků. Pro uložení vektoru `cont_mat` do `cv::Mat` použijeme způsob uložení `CV_8UC3`, který říká, že se jedná o 8-bitový unsigned integer BGR obraz. OpenCV má kanály pro modrou a červenou, na rozdíl od Matlabu, prohozené, proto jsme museli naši mapu barev funkcí `flip()` v kapitole 4.3.3 převrátit.

Abychom mohli do `cv::Mat` uložit obraz, musíme vektor `cont_mat` převést z typu double na typ uchar. Voláním funkce `Mat()` a přidáním informací o rozměrech obrázku vytvoříme strukturu `cv::Mat src`. Obrázek zobrazíme funkcí `imshow()`. Funkce `waitKey()` čeká 30 milisekund na stisk libovolného tlačítka. Používá se současně s funkcí `imshow()`, aby program měl čas zobrazit a vykreslit okno obrázku, viz příloha 9.3 na řádku 83 a 94.

V okně budeme zobrazovat počet snímků, který se zobrazí za sekundu – FPS. Na řádcích 51 až 54 v příloze 9.3 definujeme knihovnu `std::chrono`, kterou budeme měřit exekuci programu. Na řádku 59 definujeme proměnnou `t1`, do které uložíme čas před získáním snímku. Na řádku 85 definujeme proměnnou `t2`, která drží čas po uložení snímku do `cv::Mat`. Rozdílem časů t2 a t1 získáme čas, za který program zpracoval jeden snímek. Převrácená hodnota času zpracování snímku nám říká, kolik by se zvládlo vytvořit snímků za sekundu – FPS. Text vložíme do obrázku funkcí `putText()` na řádku 92.

4.3.5 Vytvoření spustitelného souboru aplikace

Opět začneme vytvořením souboru `CMakeLists.txt`. Vyjdeme ze souboru, který jsme vytvořili v kapitole 4.2.2. Projekt pojmenujeme *contour*, ale musíme přidat dvě nově použité knihovny. V kódu 4.7, na řádku 3 až 5, najdeme knihovny OpenCV a realsence2 a na řádcích 10 a 11 odkazujeme spustitelný soubor na přidané knihovny.

Získávat, zpracovávat a zobrazovat snímky budeme v cyklu `while()`. V příloze 9.3 nejdříve definujeme všechny proměnné, následně v cyklu `while()` získáme snímek a převedeme jej na vektor typu double, který následně zpracujeme v Matlabem vygenerované CUDA funkci. Poté zpracovaný snímek zobrazíme pomocí OpenCV funkce `imshow()`.

```
1  cmake_minimum_required(VERSION 3.1)
2  project(contour)
3  find_package( OpenCV REQUIRED )
4  include_directories( ${OpenCV_INCLUDE_DIRS} )
5  find_package(realsense2 REQUIRED)
6  find_package(CUDA REQUIRED)
7  set(CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS} -gencode arch=compute_53,code=sm_53)
8  file( GLOB  cu  *.cu)
9  cuda_add_executable(${CMAKE_PROJECT_NAME} main.cpp ${cu})
10 target_link_libraries(${CMAKE_PROJECT_NAME} ${realsense2_LIBRARY})
11 target_link_libraries(${CMAKE_PROJECT_NAME} ${OpenCV_LIBS} )
```

Kód 4.7: `CMakeLists.txt` pro aplikaci

Spustitelný soubor vygenerujeme stejným způsobem jako v kapitole 4.2.2. Na obrázku 4.6d je zobrazený výstupní snímek aplikace. Objekty nejblíže kamery jsou zobrazeny červenou barvou, s rostoucí vzdáleností přecházejí do žluté, zelené a nejvzdálenější pixely jsou modré.

4.3.6 Post-processing

Kvalitu získaných snímků se můžeme pokusit zlepšit použitím post-processing filtrů. Post-processing získaného snímku můžeme provést dvěma způsoby. Knihovna realsence2 obsahuje algoritmy pro úpravu získaného snímku. Druhou možností je filtr vytvořit přímo v Matlabu. Snížením rozlišení pomocí filtru bychom mohli mírně zvýšit FPS aplikace. Většinou ale filtry spíše snižují FPS, kvůli jejich výpočetní náročnosti.

Filtry knihovny librealsence2

Knihovna realsesnce2 obsahuje post-processing filtry pro snížení rozlišení snímku, zvýšení kvality hloubkových dat získaných ze senzoru, filtry pro potlačení šumu a filtry pro vyplnění mezer ve snímku.

- **Decimation filtr** snižuje rozlišení získaného snímku a tím i jeho komplexitu. Filtr má volitelné velikosti kernelu od [2,2] do [8,8]. To znamená, že pro kernel např. [2,2] se z hodnot 4 pixelů získá hodnota jednoho. Rozlišení se tímto způsobem sníží dvakrát. Pro kernel 2 a 3 se používá mediánová hodnota, pro větší kernely se používá průměr z důvodu výpočetní náročnosti.
- **Spatial Edge-Preserving filtr** odstraňuje šum hloubkových dat a vyhlazuje plochy při zachování ostrých hran. Také částečně vyplňuje díry v hloubkových datech. Nastavují se u něj čtyři parametry, které ovlivňují: (1) intenzitu vyhlazení ploch, (2) zachování hran při použití filtru, (3) vyplnění mezer, (4) počet iterací při aplikaci filtru.
- **Tempolar filtr** je ideální pro scénu, která se náhle nemění. Využívá nenulové hodnoty z předchozích snímků a porovnává je s hodnotami aktuálního snímku. Díky tomu dojde k vyhlazení hloubkových dat a zaplnění mezer ve snímku. U Temporal filtru se nastavují stejné parametry alfa a delta jako u Spatial Edge-Preserving filtru. Navíc se nastavuje index vytrvalosti, což je soubor pravidel pro vyplnění chybějícího pixelu poslední platnou hodnotou.
- **Holes Filling filtr** několika způsoby odhaduje hodnoty pixelů, pro které neznáme hodnoty ze senzoru kamery. Používá hodnotu sousedních pixelů nebo data z RGB kamery. Holes Filling filtr přijímá jeden parametr, definující způsob práce filtru. Může používat pro doplnění chybějící hodnoty buď hodnotu pixelu nalevo od něj, hodnotu sousedního pixelu, která je nejvzdálenější od senzoru kamery, nebo hodnotu sousedního pixelu nejblíže senzoru kamery [24][19].

V příloze 9.3 nejdříve definujeme filtry, které chceme použít – řádky 36 až 40. Na řádcích 42-46 heuristicky definujeme parametry jednotlivých filtrů. V cyklu `while()` duplikujeme získaný snímek typu `rs2::depth_frame` do proměnné `filtered` typu `rs2::frame`, na který můžeme aplikovat filtry, což uděláme na řádcích 77 až 81. Filtrovaný snímek následně zpracujeme stejným způsobem jako snímek, na který jsme filtry nepoužili. V obrázku 4.6 je srovnání snímků s použitými filtry. Při daném nastavení realsence filtrů se zmenší rozlišení snímků na 640x360 pixelů. Na snímkovou frekvenci použití filtrů vliv nemělo, pořád se pohybujeme mezi 3 a 4 FPS. Holes Filling filtr na obrázku 4.6b velice agresivně vyplňuje prázdná místa a dochází

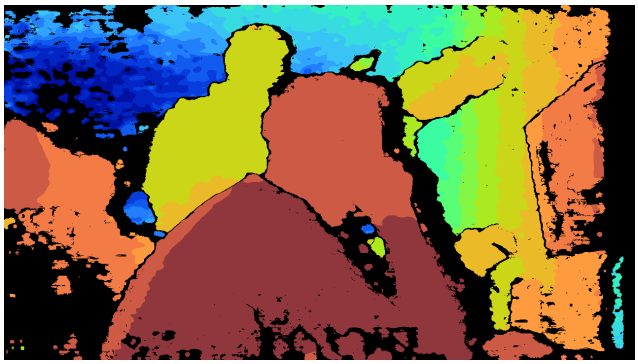
k doplnění nepřesných hodnot, zejména kolem obrysu postavy. Levá horní část snímku je ale poměrně dobře vyplněná a při použití na nádobě s pískem by výsledky mohly býti uspokojivé.

Matlab filtr

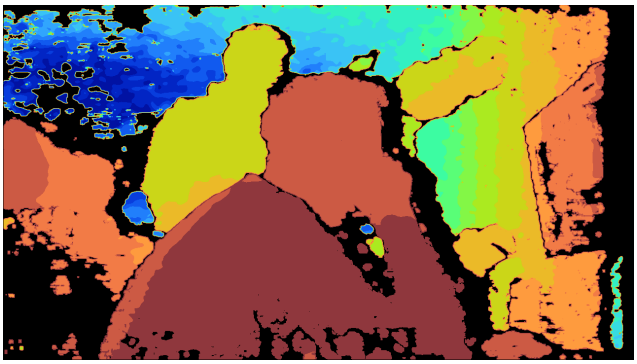
Do funkce `opencv_contour` v příloze 9.2 přidáme skalární vstup, kterým budeme moci zvolit použití filtru. Vytvoříme novou funkci `contour_filter()` se třemi vstupy – vektorem dat snímku a skalárními hodnotami rozlišení snímku.

Na hloubková data budeme aplikovat filtr Gaussovského rozostření. Tímto způsobem vyhladíme data a zbavíme se lokálních extrémů. V příloze 9.4 v kódu funkce `contour_filter()` nejdříve transformujeme vektor dat snímku na matici. V proměnné `g` uložíme kernel, který na data budeme aplikovat. V cyklu `for()` iterujeme přes všechny pixely. Matici kernelu násobíme po prvcích se stejně velikou maticí okolí pixelu. Pro kernel 3x3 tedy vybereme matici 3x3, v jejímž středu bude pixel, jehož hodnotu měníme a jeho sousedních osm pixelů. Tuto matici po prvcích násobíme maticí kernelu. Suma výsledných hodnot po násobení je nová hodnota pixelu. Jelikož suma hodnot kernelu je jedna, nedojde ke zkreslení hodnot snímku. Abychom mohli kernel aplikovat i na krajní hodnoty matice snímku, funkcí `padarray()` zvětšíme rozměr matice o 1 v každém směru a do vzniklých prvků matice dosadíme hodnotu 0. Pro kernel 3x3 máme upravený snímek na obrázku 4.6c.

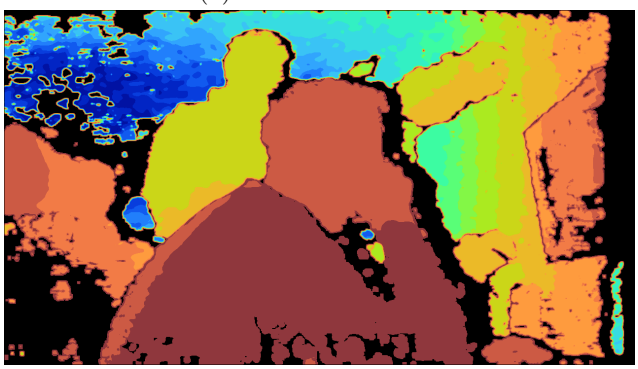
Při použití kernelu velikosti 5x5 a 7x7 upravujeme hodnotu jednotlivých pixelů na základě většího množství dat, viz srovnání velikosti kernelů na obrázku 4.5. Použitím filtrů jsme schopni v malé míře zaplnit prázdná místa (černá místa), dojde ale k zkreslení hran, kdy se ostré přechody zjemní. To není tak znatelné při přechodu mezi sousedícími výškovými intervaly, ale u ostřejších přechodů, kdy je výška mezi sousedícími pixely větší než jeden výškový interval. To by ale neměl být problém, kdybychom snímali právě nádobu s pískem, kde by k ostrým výškovým přechodů nemělo docházet.



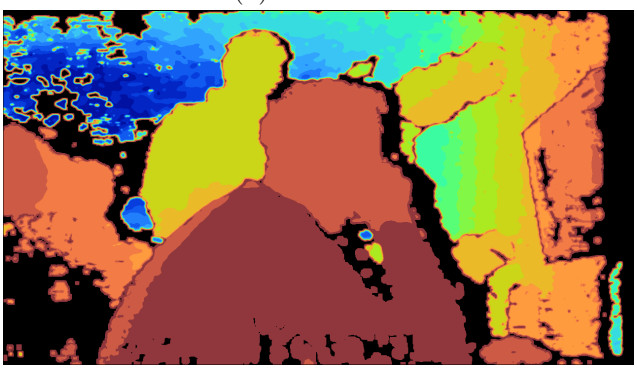
(a) Snímek bez filtru



(b) Kernel 3x3

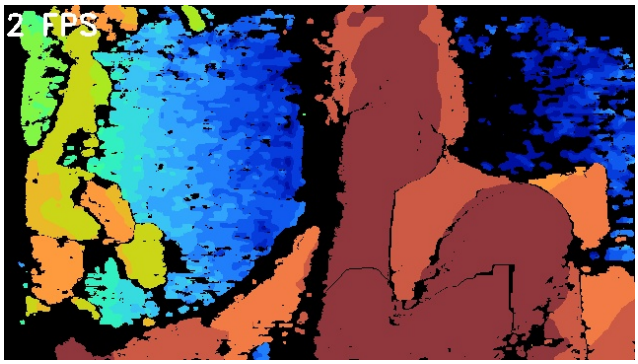


(c) Kernel 5x5

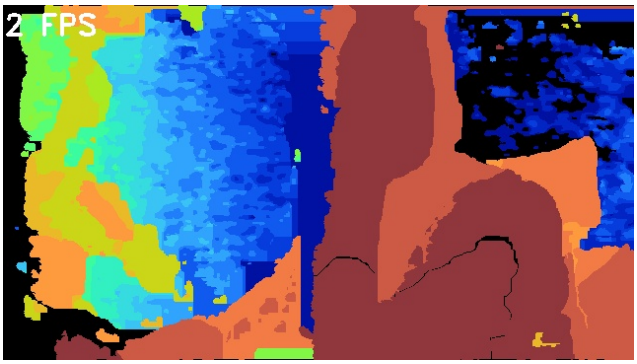


(d) Kernel 7x7

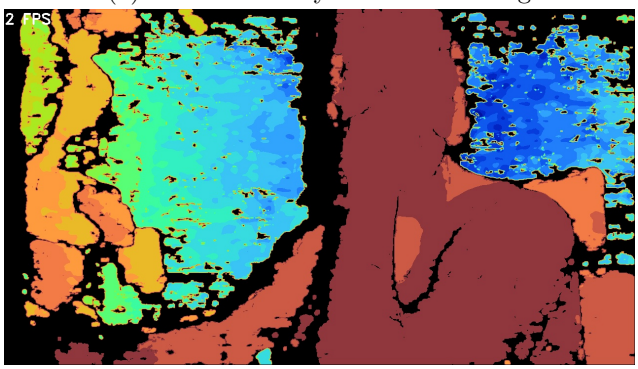
Obrázek 4.5: Použití různých velikostí kernelů



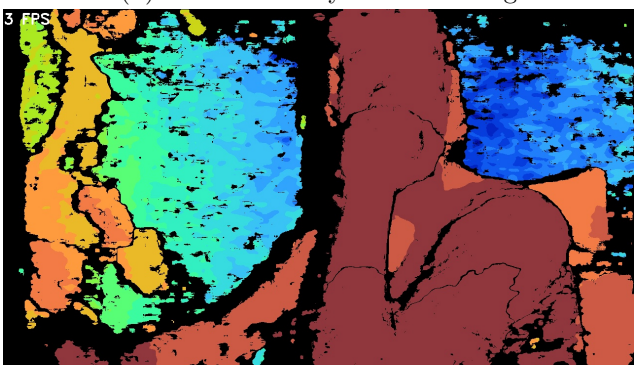
(a) Realsence filtry bez Holes Filling



(b) Realsence filtry s Holes Filling



(c) Matlab filtr s kernelem 3x3



(d) Snímek bez použití filtrů

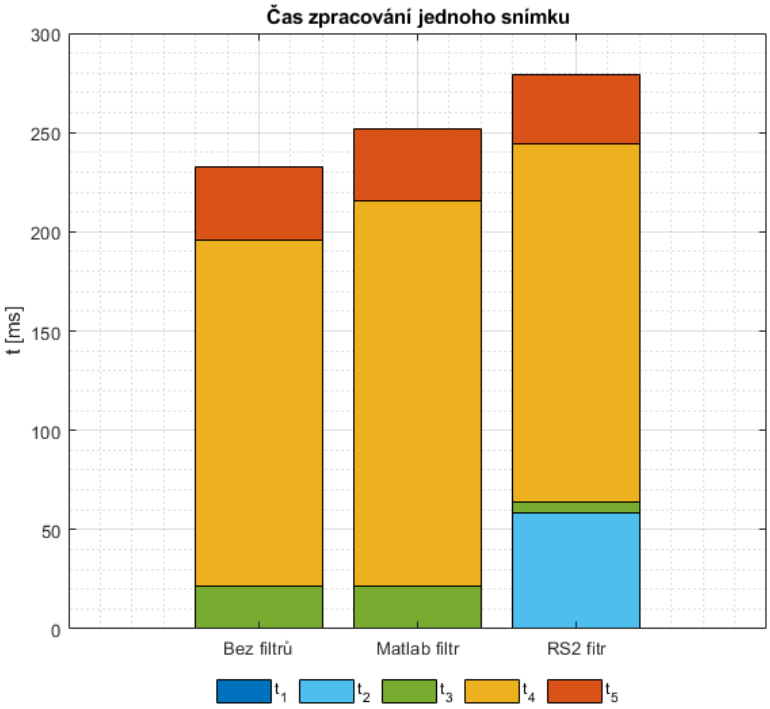
Obrázek 4.6: Ukázky snímků při použití filtrů

5 Výsledky

5.1 Analýza běhu aplikace

Pro analýzu běhu aplikace budeme měřit čas běhů jednotlivých částí aplikace. Použijeme již přítomnou knihovnu `std::chrono` stejným způsobem jako v kapitole 4.3.4. Budeme tedy měřit pět různých částí běhu aplikace:

- čas t_1 – doba získání snímku z kamery,
- čas t_2 – doba zpracování snímku librealsence2 filtry,
- čas t_3 – doba převodu snímku na vektor,
- čas t_4 – doba zpracování dat CUDA funkcí,
- čas t_5 – doba vykreslení snímku.



Obrázek 5.1: Doba zpracování jednoho snímku.

V grafu 5.1 vidíme průměrné časy z 1000 snímků pro 5 dříve zmíněných operací programu s použitím Matlab filtrů, realsence2 filtrů a bez filtrů.

	t_1 [ms]	t_2 [ms]	t_3 [ms]	t_4 [ms]	t_5 [ms]	$\sum_{i=1}^{5} t_i$ [ms]
Bez filtrů	0.5779	0.0002	20.9887	174.3589	37.0183	232.9440
Matlab filtr	0.5761	0.0001	20.6377	194.2994	36.3122	251.8255
RS2 fitr	0.5806	58.1322	4.9267	180.3357	35.4021	279.3773

Tabulka 5.1: Doba zpracování jednoho snímku

Průměrný čas získání snímku z kamery byl 0,58 milisekund pro všechny tři případy.

Čas pro použití realsence2 filtrů je 58 milisekund, což odpovídá asi 21% celé aplikace.

Převod snímku na vektor trval nejméně při použití realsence2 filtrů, jelikož jejich použitím bylo zmenšeno rozlišení snímku. V obou zbylých případech převod snímku trval asi 20 milisekund, tedy asi 9% respektive 8% procent celé aplikace.

Nejnáročnější operací je zpracování dat, které probíhá na grafické kartě Jetsonu Nano. Nejdéle zpracování dat trvalo při použití Matlab filtrů – 194 milisekund, při použití realsence2 filtrů 180 milisekund a bez použití filtrů 174 milisekund. Průměrně tedy zpracování snímku zabralo 72% času celé aplikace.

Zobrazení snímku zabralo ve všech třech případech asi 36 milisekund tedy průměrně 14% z celé aplikace.

Snímková frekvence je při použití realsence2 filtrů 3,58 FPS, při použití Matlab filtrů 3,97 FPS a bez použití filtrů 4,29 FPS.

5.2 LU rozklad

Jak bylo zmíněno v kapitole 2.1, díky paralelizaci jsou GPU využívány při náročných matematických výpočtech. Tuto výhodu budeme demonstrovat na příkladu LU rozkladu a srovnáním s dalšími dostupnými vývojářskými platformami. Princip metody LU rozkladu (trojúhelníkový rozklad) spočívá v rozkladu čtvercové matice n-tého řádu na součin tzv. trojúhelníkových matic. Tento pojem vymezujeme podle následující definice: Jestliže jsou všechny prvky čtvercové matice U ležící pod hlavní diagonálou rovny nule, nazývá se matice U horní trojúhelníková matice. Jestliže jsou všechny prvky čtvercové matice L ležící nad hlavní diagonálou rovny nule, nazývá se matice L dolní trojúhelníková matice. Metoda LU rozkladu se používá při numerickém řešení systémů lineárních rovnic [20].

```
1  function [x] = LU_dec(n)
2     coder.gpu.kernelfun();
3     A = rand(n);
4     tic
5     [L U] = lu(A);
6     x = toc;
7  end
```

Kód 5.1: Zdrojový kód funkce `LU_dec()`

V Matlabu obsahuje funkci `lu()`, která provádí LU rozklad. Vytvoříme funkci `LU_dec()`, která bude mít skalární vstup definující řád matice. Výstupem funkce bude skalární hodnota času, která udává dobu výpočtu LU rozkladu. V kódu 5.1 nejdříve vygenerujeme matici řádu n, a budeme měřit čas potřebný pro její LU rozklad.

C++ funkce LU rozkladu je v příloze 9.5.

Při generování spustitelného souboru Matlab GPU Coderem přidáme funkci z přílohy 9.5

do main souboru aplikace, budeme tak mít srovnání s ryze CUDA funkcí, která bude využívat cuSOLVER a cuBLAS knihovny.

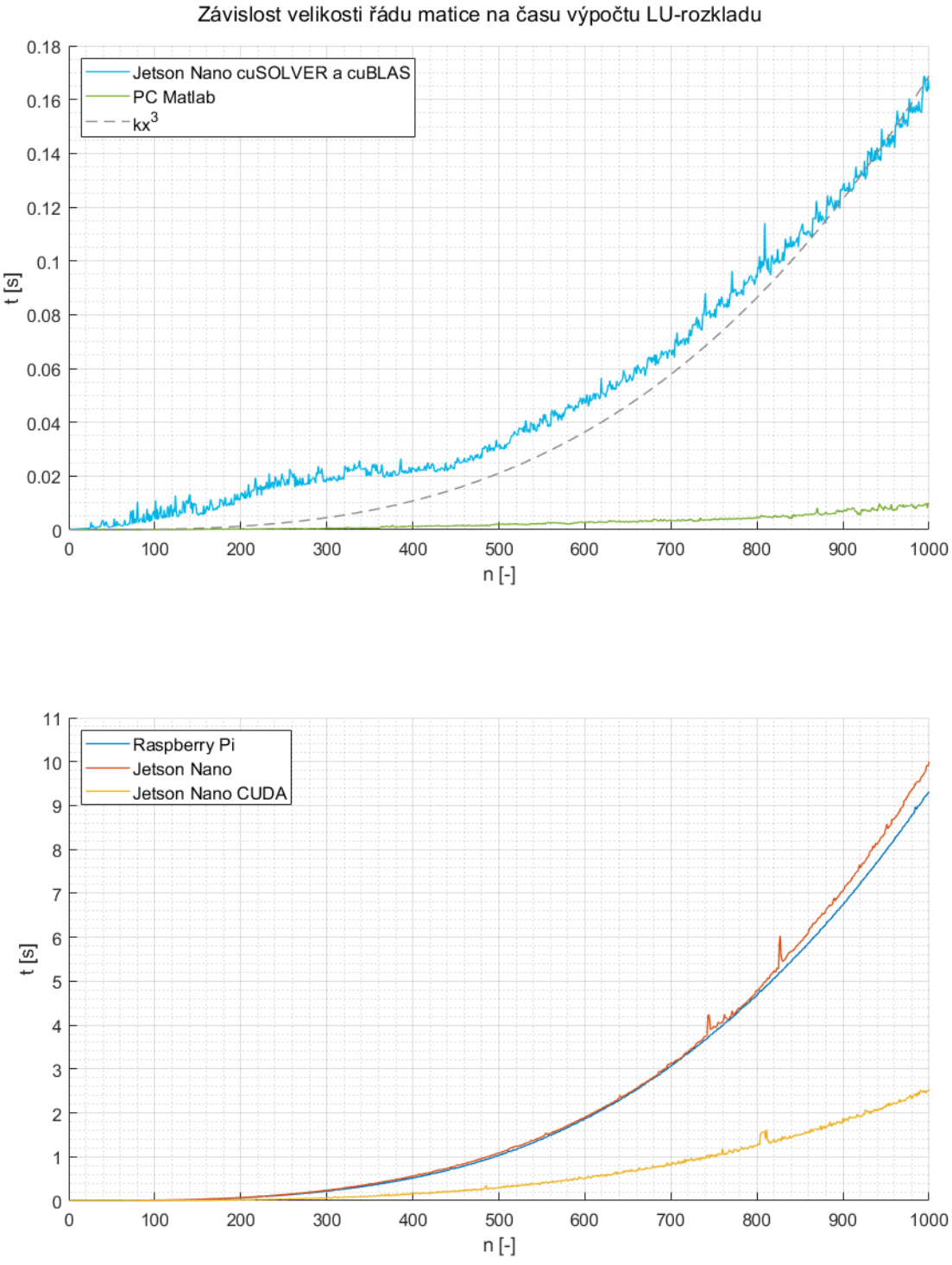
Jako příklad další dostupné vývojářské platformy budeme LU rozklad počítat na Raspberry Pi 4. Jedná se o malý počítač podobný Jetsonu Nano s tím rozdílem že jeho grafický procesor je mnohem méně výkonný, než u Jetsonu Nano. Raspberry Pi nepodporuje CUDA knihovny, program tedy poběží čistě na CPU. Pro generování spustitelného souboru využijeme nástroj CMake stejným způsobem jako na Jetsonu.

Provedeme tedy výpočet LU rozkladu náhodné matice s řádem 1 až 1000 a pro každý řád budeme měřit čas výpočtu. Výpočet budeme provádět několika způsoby:

- V prostředí Matlab na stolním počítači,
- na Jetsonu Nano v CUDA kódu s knihovnami cuSOLVER a cuBLAS,
- na Jetsonu Nano v CUDA kódu bez dalších knihoven,
- na Jetsonu Nano v C++,
- na Raspberry Pi 4 v C++.

Na grafu 5.2 jsou zobrazeny naměřené hodnoty času výpočtu LU rozkladu pro matici o řádu n. Výpočetní čas pro řád matice 1000 byl na třikrát levnějším Raspberry Pi 4 a Jetsonu Nano srovnatelný. Pokud jsme ale na Jetsonu Nano použili paralelizaci výpočtu na GPU díky CUDA kódu, dosáhli jsme čtyřnásobně rychlejšího výpočtu. Využitím CUDA knihoven cuSOLVER a cuBLAS jsme na Jetsonu Nano dosáhli 15krát rychlejšího výpočtu. Podle očekávání jsme na stolním počítači dosáhli nejlepšího času, jeho cena a možnosti použití se ale nedají s Jetsonem Nano srovnávat.

Z grafu je také zřetelná kubická závislost výpočetního času na řádu matice.



Obrázek 5.2: Graf závislosti velikosti řádu matice na času výpočtu LU rozkladu.

6 Závěr

Cílem práce bylo prozkoumat možnosti generování CUDA code pomocí Matlab GPU Coder pro zpracování surových dat z kamery.

První část práce je zaměřena na představení CUDA code, Matlab GPU Coder, použitého grafického zařízení Nvidia Jetson Nano a použité hloubkové kamery Intel RealSence D415.

Druhá část práce je zaměřena na samotné generování CUDA code pomocí Matlab GPU Coder a vytvoření CUDA funkcí pro aplikaci, která zpracovává a zobrazuje surová data získaná z kamery.

První problém při vytváření aplikace byla skutečnost, že Matlab GPU Coder nepodporuje knihovnu librealsence2, která je nutná pro získání hloubkových dat z kamery. Bylo tedy potřeba pomocí Matlab GPU Coder generovat jen funkce pro zpracování obrazu a získání hloubkových dat z kamery a jejich zobrazení bylo provedeno v C++, které vygenerované funkce využívalo. Nemožnost využít Matlab GPU Coder k získání dat z kamery vedla k vytváření spustitelného souboru nástrojem CMake.

Při vytváření CUDA code funkcí pro zpracování obrazových dat se projevily hlavní nedostatky Matlab GPU Coder. Matlab GPU Coder podporuje omezenou skupinu Matlab funkcí. Nepodporuje např. funkci `contour()`, která by vytvořila z matice výškových dat vrstevnice. Za zmínku stojí, že Matlab GPU Coder nepodporuje funkci `turbo()`, která vrací barevnou mapu. Pokud si ale zobrazíme zdrojový kód této funkce příkazem `edit turbo` v Matlab příkazovém okně a celý zdrojový kód zkopírujeme do vlastní funkce, jíž změníme jméno, aby nedošlo ke kolizi jmen funkcí, Matlab GPU Coder s takovou funkcí problém nemá.

Podle dokumentace Matlab GPU Coder nepodporuje příkaz `break` a doporučuje ho nahradit podmínkou a funkcí `while()`. Při generování kódu mě na to Matlab GPU Coder jednou upozornil a nedovolil mi generovat kód ani s funkcí `for()`. Jak je ale vidět ze zdrojového kódu 9.2, příkaz `break` je zde použit a generování proběhlo bez kompilací.

Pokud ale používáme jen vybrané funkce a generujeme přímo executable např. na Jetsnon Nano, Matlab GPU Coder funguje dobře.

Další problém bylo vytváření spustitelného souboru pomocí nástroje CMake. Pro správnou konfiguraci souboru `CmakeLists.txt` jsem se musel obrátit na vývojářské fórum forums.developer.nvidia.com. Nepodařilo se mi ale pomocí nástroje CMake vygenerovat spustitelný soubor z CUDA code funkcí, které využívaly další CUDA knihovny, jako např. cuSOLVER nebo cuBLAS. To se podařilo pouze vygenerováním spustitelného souboru v Matlab GPU Coderu.

Při vytváření zdrojového kódu CUDA pomocí Matlab GPU Coder jsem narazil na problém, kdy ve složce, kam se vygenerované soubory ukládaly chyběly některé hlavičkové soubory, na které se vygenerované hlavičkové soubory odkazovali. Ty se většinou nacházejí v instalační složce programu Matlab a je třeba je například pomocí Průzkumníka souborů ve Windows podle názvu hledat. Pokud se rozhodneme uložit vygenerované soubory do zip archivu tlačítkem *Package* v posledním kroku průvodce generováním CUDA code, měl by archiv obsahovat všechny potřebné hlavičkové soubory.

Knihovna librealsence2 je dobře zdokumentovaná na vývojářské platformě Github, kde jsou i příklady kódu. U kamery D415 je výrazný šum dat a pro mnoho bodů kamera data vůbec

nezměří. To do jisté míry řeší filtry, ale výsledek není dokonalý.

Výsledná aplikace na Jetsonu Nano vytvoří 3-4 snímky za sekundu. Nejvíce výpočetního času zabírá CUDA funkce, která zpracovává data. Tento malý výkon může být zapříčiněn způsobem kompilování spustitelného souboru. Výhody použití CUDA code jsou demonstrovány v kapitole 5.2, kde využitím CUDA code a CUDA knihoven výrazně snížíme výpočetní čas LU rozkladu.

Aplikace nebyla testována na zmíněné nádobě s pískem, kdy měla vrstevnice na písek promítat. Práce byla více zaměřena na generování a spouštění CUDA code a vzájemnou spolupráci Jetsonu, Matlabu, Matlab GPU Coder a Intel RealSence D415 kamery. Připojením projektoru k Jetson Nano a spuštění aplikace by se tohoto cíle dalo dosáhnout.

Aplikace by se dala vylepšit jiným způsobem generování kódu, zejména využitím CUDA knihoven, které by mohly snížit výpočetní čas CUDA funkce, která zpracovává data. Jednou možností je přímo replikovat způsob generování spustitelného souboru z Matlab GPU Coder.

Nakonec proběhlo srovnání rychlosti výpočtu LU rozkladu pomocí CUDA code, přímo v Matlabu a čistě v C++. Použitím CUDA code na Jetson Nano proběhne výpočet LU rozkladu 15x rychleji, než čistě v C++ na stejném zařízení. Srovnání výpočetního času konkurenční vývojářskou deskou, která CUDA code nepodporuje, dopadlo stejným způsobem, kdy výpočet v CUDA code proběhl asi 15krát rychleji než na Raspberry Pi.

K programování byl použit program Matlab R2020b. Na Jetson Nano byl nainstalován operační systém Linux 4 Tegra 32.4.4, Ubuntu 18.04.5, s verzí CUDA 10.2. Pro kompilování byla použita verze CMake 3.10.2. Použitá librealsence2 knihovna byla ve verzi 2.41.0.

7 Seznam použitých zkratek a symbolů

IoT internet of things

CPU centrální procesorová jednotka, anglicky central processing unit

GPU grafická procesorová jednotka, anglicky graphical processing unit

AI umělá inteligence, anglicky artificial intelligence

API rozhraní pro programování aplikací, anglicky application programming interface

SOM systém na modulu, anglicky system on module

SOC integrovaný obvod, který obsahuje všechny součásti elektronického systému na jednom čipu, anglicky system on chip

RAM paměť s náhodným přístupem, anglicky random access memory

PoE napájení pomocí datového síťového kabelu, anglicky power over ethernet

SSH anglicky secure shell

IP anglicky internet protocol

GUI grafické uživatelské prostředí, anglicky graphical user interface

FPS frames per second

8 Literatura

[1] ABI-CHAHLA, Fedy. *Nvidia's CUDA: The End of the CPU?* [online]. June 18, 2008, , 9-10 [cit. 2021-03-11]. Dostupné z: https://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954.html

[2] HARRIS, Mark. *An Easy Introduction to CUDA C and C++* [online]. 31 October 2012 [cit. 2021-03-18]. Dostupné z: https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/

[3] *How CPU and GPU Work Together* [online]. In: . [cit. 2021-03-18]. Dostupné z: https://www.omnisci.com/technical-glossary/cpu-vs-gpu

[4] SEDLÁČEK, Filip. *Zpracování obrazu s velkými datovými toky - využití CUDA/OpenCL.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce Ing. Petr Honec, Ph.D.

[5] *BrookGPU: Introduction* [online]. [cit. 2021-03-18]. Dostupné z: http://graphics.stanford.edu/projects/brookgpu/intro.html

[6] , dusty_nv. *Power supply considerations for Jetson Nano Developer Kit* [online]. 18 March 2019n. l. [cit. 2021-03-18]. Dostupné z: https://forums.developer.nvidia.com/t/power-supply-considerations-for-jetson-nano-developer-kit/71637

[7] KANGALOW. *$59 Jetson Nano 2GB* [online]. 5 October 2020 [cit. 2021-03-18]. Dostupné z: https://www.jetsonhacks.com/2020/10/05/59-jetson-nano-2gb/

[8] Deep Learning For Object Detection. *Mathworks* [online]. 8 Apr 2020 [cit. 2021-03-18]. Dostupné z: https://www.mathworks.com/matlabcentral/fileexchange/73954-deep-learning-for-object-detection?s_eid=PSM_15028

[9] *GPU Coder Generate CUDA code for NVIDIA GPUs* [online]. [cit. 2021-03-18]. Dostupné z: https://www.mathworks.com/products/gpu-coder.html

[10] IRFAN, Ahmed. *MATLAB GPU coder for jetson* [online]. In: . 23 Oct 2020 [cit. 2021-03-18]. Dostupné z: https://www.mathworks.com/matlabcentral/answers/591646-matlab-gpu-coder-for-jetson

[11] *NVIDIA CUDA-X GPU-Accelerated Libraries* [online]. [cit. 2021-03-18]. Dostupné z: https://developer.nvidia.com/gpu-accelerated-libraries

[12] *About CMake* [online]. [cit. 2021-03-18]. Dostupné z: https://cmake.org/overview/

[13] ZCY. *Can't install "cmake"* [online]. In: . [cit. 2021-03-18]. Dostupné z: https://forums.developer.nvidia.com/t/can-t-install-cmake/75299/2

[14] *CMake 3.4* [online]. [cit. 2021-03-18]. Dostupné z: http://www.fit.vutbr.cz/ imilet/shared/seminar/CMake.pdf

[15] DUSTY_NV. *Using CMake with Matlab GPU Coder generated CUDA code* [online]. In: . [cit. 2021-03-18]. Dostupné z: https://forums.developer.nvidia.com/t/using-cmake-with-matlab-gpu-coder-generated-cuda-code/169803/4

[16] DRKSTR. *Running pyrealsence2 on JetsonNano* [online]. In: . Oct 13 2020 [cit. 2021-03-25]. Dostupné z: https://github.com/IntelRealSense/librealsense/issues/6964#issuecomment-707501049

[17] MATHWORKS, INC. *GPU Coder User's Guide* [online]. March 2021. [cit. 2021-03-26]. Dostupné z: https://www.mathworks.com/help/pdf_doc/gpucoder/gpucoder_ug.pdf

[18] *Introduction* [online]. [cit. 2021-03-28]. Dostupné z: https://docs.opencv.org/4.5.1/d1/dfb/intro.html

[19] MALOEL. *Librealsense Post-Processing Filters* [online]. Sep 25 2020 [cit. 2021-04-02]. Dostupné z: https://github.com/IntelRealSense/librealsense/blob/master/doc/post-processing-filters.md

[20] HASÍK, Karel. *Numerické metody* [online]. [cit. 2021-04-23]. Dostupné z: https://www.slu.cz/file/cul/3f2481de-0419-42ec-91b1-ab2cdde89f45. Matematický ústav Slezské univerzity v Opave.

[21] MALOEL. *Intel® RealSense SDK* [online]. [cit. 2021-04-08]. Dostupné z: https://github.com/IntelRealSense/librealsense

[22] DAQRI. *DEPTH CAMERAS FOR MOBILE AR: FROM IPHONES TO WEARABLES AND BEYOND* [online]. 26.4.2018 [cit. 2021-04-09]. Dostupné z: https://medium.com/@DAQRI/depth-cameras-for-mobile-ar-from-iphones-to-wearables-and-beyond-ea29758ec280

[23] *Intel® RealSense Depth Camera D415* [online]. [cit. 2021-04-09]. Dostupné z: https://www.intelrealsense.com/depth-camera-d415/

[24] GRUNNET-JEPSEN, Anders, John N. SWEETSER a John WOODFILL. [online]. [cit. 2021-04-09]. Dostupné z: https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/BKMs_Tuning_RealSense_D4xx_Cam.pdf

[25] *Seeed Studio NVIDIA® Jetson Nano Developer Kits* [online]. 13-05-2019 [cit. 2021-04-23]. Dostupné z: https://cz.mouser.com/new/seeed-studio/seeed-nvidia-jetson-nano-dev-kit/

9 Přílohy

9.1 Zdrojový kód pro získání hloubkových dat kamery

```
#include <librealsense2/rs.hpp> // includes RealSense Cross Platform API
#include <fstream> //includes file operations
//Get depth scale
float get_depth_scale(rs2::device dev)
{
  for (rs2::sensor& sensor : dev.query_sensors())
  {
    if (rs2::depth_sensor dpt = rs2::depth_sensor(sensor))
    {
      return dpt.get_depth_scale();
    }
  }
  throw std::runtime_error("Device does not have a depth sensor");
}

int main(int argc, char * argv[])
{
    //Define and start pipeline
    rs2::pipeline p;
    rs2::colorizer color_map;
    rs2::config cfg;
    cfg.enable_stream(RS2_STREAM_DEPTH, 1280, 720);
    rs2::pipeline_profile profile = p.start(cfg);
    //Define variables
    float depth_scale = get_depth_scale(profile.get_device());
    rs2::frameset frames = p.wait_for_frames();
    rs2::depth_frame depth = frames.get_depth_frame();
    int width = depth.get_width();
    int height = depth.get_height();
    float max_distance = 5;
    double pixel_distance[921600];
    //rs2::depth_frame to double
    const uint16_t* p_depth_frame = reinterpret_cast<const uint16_t*>(depth.get_data());
    for (int y = 0; y < height; y++){
        auto pixel_index = y * width;
        for (int x = 0; x < width; x++, ++pixel_index)
        {
            pixel_distance[pixel_index] = depth_scale * p_depth_frame[pixel_index];
```

```
39              if (pixel_distance[pixel_index] < 0.f || pixel_distance[pixel_index] > max_distance)
40              {
41                  pixel_distance[pixel_index] = 0;
42              }
43          }
44      }
45      //save vector to file
46      FILE *f = fopen("points.bin", "wb");
47      fwrite(pixel_distance, sizeof(double), sizeof(pixel_distance), f);
48      fclose(f);
49
50      return 0;
51  }
```

9.2 Zdrojový kód funkce pro zpracování dat

```
function [cont_mat] = opencv_contour(depth_data, filter)

    coder.gpu.kernelfun();
    z0 = 0.5;
    levels = 20;
    depth_data(depth_data == 0) = NaN;
    depth_data = z0 - depth_data;
    depth_data(isnan(depth_data)) = 0;

    if filter
        depth_data = contour_filter(depth_data, 720, 1280);
    end

    len = numel(depth_data);
    height = linspace(min(depth_data), max(depth_data), levels);
    color_map = contour_colormap(levels);
    cont_mat = zeros(1, len*3);
    j = 1;
    for i = 1:3:len*3
        if(depth_data(j) == 0)
            cont_mat(i:(i+2)) = [0 0 0];
        else
            for k = 1:levels
                if(height(k) >= depth_data(j))
                    cont_mat(i:(i+2)) = color_map(k, :);
                    break;
                end
            end
        end
        j = j + 1;
    end
end
```

9.3 Zdrojový kód aplikace

```
#include <librealsense2/rs.hpp> // includes RealSense Cross Platform API
#include <fstream> // load text file
#include <string> //includes copy
#include <stdio.h> //for printf
#include "opencv_contour.h" //includes Matlba generated CUDA function
#include <opencv2/opencv.hpp> //includes OpenCV
#include <time.h> // includes clock_t and CLOCKS_PER_SEC

using namespace cv;
using namespace std;
//Get depth scale
float get_depth_scale(rs2::device dev)
{
    // Go over the device's sensors
    for (rs2::sensor& sensor : dev.query_sensors())
    {
        // Check if the sensor is a depth sensor
        if (rs2::depth_sensor dpt = rs2::depth_sensor(sensor))
        {
            return dpt.get_depth_scale();
        }
    }
    throw std::runtime_error("Device does not have a depth sensor");
}

int main()
{
    //Define and start pipeline
    rs2::pipeline p;
    rs2::colorizer color_map;
    rs2::config cfg;
    cfg.enable_stream(RS2_STREAM_COLOR, 1280, 720);
    cfg.enable_stream(RS2_STREAM_DEPTH, 1280, 720);
    rs2::pipeline_profile profile = p.start(cfg);
    // Declare filters
    rs2::decimation_filter dec_filter;
    rs2::spatial_filter spat_filter;
    rs2::temporal_filter temp_filter;
    rs2::hole_filling_filter hole_filter;
    rs2::threshold_filter tres_filter;
    // Configure filter parameters
    rs2::decimation_filter(3.f);
    rs2::spatial_filter(0.25f, 8.f, 2.f, 5.f);
    rs2::temporal_filter(0.f, 5.f, 1);
    rs2::hole_filling_filter(2);
    rs2::threshold_filter(0.1f, 5.f);
```

```
47      //Define variables
48      float depth_scale = get_depth_scale(profile.get_device());
49      static double pixel_distance[921600];
50      static double cont_mat[2764800];
51      static uchar bum[2764800];
52      float max_distance;
53      double filter;
54      int rs2_filter;
55      //Load variables from file
56      std::ifstream settings("settings.txt");
57      settings >> max_distance;
58      settings >> filter;
59      settings >> rs2_filter;
60      //Define variables for timer
61      using std::chrono::high_resolution_clock;
62      using std::chrono::duration_cast;
63      using std::chrono::duration;
64      using std::chrono::milliseconds;
65
66      while (1)
67      {
68          //Start timer
69          auto t1 = high_resolution_clock::now();
70          // Wait for frames
71          rs2::frameset frames = p.wait_for_frames();
72          // Try to get a frame of a depth image
73          rs2::depth_frame depth = frames.get_depth_frame();
74          rs2::frame filtered = depth;
75          //rs2 filter
76          if(rs2_filter){
77              filtered = dec_filter.process(filtered);
78              filtered = spat_filter.process(filtered);
79              filtered = temp_filter.process(filtered);
80              //filtered = hole_filter.process(filtered);
81              filtered = tres_filter.process(filtered);
82          }
83          //Array from depth frame
84          const uint16_t* p_depth_frame = reinterpret_cast<const uint16_t*>(filtered.get_data());
85          const int width = filtered.as<rs2::video_frame>().get_width();
86          const int height = filtered.as<rs2::video_frame>().get_height();
87          for (int y = 0; y < height; y++)
88          {
89              auto pixel_id = y * width;
90              for (int x = 0; x < width; x++, ++pixel_id)
91              {
92                  pixel_distance[pixel_id] = depth_scale * p_depth_frame[pixel_id];
93                  if (pixel_distance[pixel_id] < 0.f || pixel_distance[pixel_id] > max_distance)
94                  {
95                      pixel_distance[pixel_id] = 0;
96                  }
```

```
97                  }
98              }
99              //Get contour
100             opencv_contour(pixel_distance, filter, cont_mat);
101             //Double to uchar
102             copy(begin(cont_mat), end(cont_mat), begin(bum));
103             //Copy image to cv::mat
104             Mat src = Mat(height, width, CV_8UC3, bum);
105             //Stop timer
106             auto t2 = high_resolution_clock::now();
107             duration<double, std::milli> ms_double = t2 - t1;
108             //Display FPS
109             double ex_time = ms_double.count()/1000;
110             int FPS = 1/ex_time;
111             string FPS_str = to_string(FPS);
112             FPS_str = FPS_str + " FPS";
113             putText(src, FPS_str, Point(1, 30), cv::FONT_HERSHEY_DUPLEX, 1.0, CV_RGB(255,255,255),2);
114             //Show image
115             imshow("Contour", src);
116             waitKey(30);
117             //Print execution time
118             printf("%f", ex_time);
119         }
120         return 0;
121 }
```

9.4 Zdrojový kód Matlab filtru

```
function [out] = contour_filter(in, h, w)
    mat = reshape(in, [w h]);
    g = [1 2 1; 2 4 2; 1 2 1]/16;
%     g = [1 2 4 2 1; 2 4 8 4 2; 4 8 16 8 4; 2 4 8 4 2; 1 2 4 2 1]/100;
%     g = [1     2     4     8     4     2     1
%          2     4     8    16     8     4     2
%          4     8    16    32    16     8     4
%          8    16    32    64    32    16     8
%          4     8    16    32    16     8     4
%          2     4     8    16     8     4     2
%          1     2     4     8     4     2     1]./484;
    n = 1;
    mat = padarray(mat, [n n]);
    out = zeros(w, h);
    for i = n+1:w
        for j = n+1:h
            x = mat(i-n:i+n, j-n:j+n).*g;
            out(i-1, j-1) = sum(x(:));
        end
    end
    out = out(:)';
end
```

9.5 C++ kód LU rozkladu

```
void LUdecomposition(float a[1000][1000], float l[1000][1000], float u[1000][1000], int n) {
   int i = 0, j = 0, k = 0;
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         if (j < i)
         l[j][i] = 0;
         else {
            l[j][i] = a[j][i];
            for (k = 0; k < i; k++) {
               l[j][i] = l[j][i] - l[j][k] * u[k][i];
            }
         }
      }
      for (j = 0; j < n; j++) {
         if (j < i)
         u[i][j] = 0;
         else if (j == i)
         u[i][j] = 1;
         else {
            u[i][j] = a[i][j] / l[i][i];
            for (k = 0; k < i; k++) {
               u[i][j] = u[i][j] - ((l[i][k] * u[k][j]) / l[i][i]);
            }
         }
      }
   }
}
```