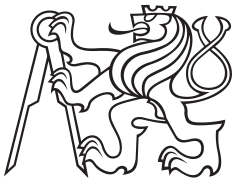
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra kybernetiky

Zrychlení lokalizace pomocí knihovny InLoc

Bc. Martin Sebera

Vedoucí: Ing. Michal Polic
Studijní program: Otevřená informatika
Specializace: Počítačové vidění a digitální obraz
Leden 2022

Poděkování

Následujícím děkuji:

- vyučujícím
- své rodině
- spolužákům
- bývalým spolužákům
- spolubydlícím
- ostatním přátelům
- kolejím Strahov

za to, že mi během mého studia pomáhali a tvořili příjemné prostředí.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 4. ledna 2022

Abstrakt

Lokalizace je úloha, která řeší odhad pózy kamery (tj. střed kamery a rotaci). Tato práce se zaměřuje na algoritmus InLoc, který implementuje vizuální lokalizaci pro zmapovaná vnitřní prostředí. Analyzuje původní implementaci InLocu, tj. jeho jednotlivé části a jejich výkonnost. Na závěr popisuje a nabízí zlepšení, konkrétně převod z výzkumné verze do produkční a zrychlení nejpomalejších částí pomocí přepisu do C++)

Klíčová slova: lokalizace, InLoc, C++, Matlab

Vedoucí: Ing. Michal Polic

Abstract

Localization is a task that solves camera pose estimation (e.g., camera center and rotation). This thesis focuses on the InLoc algorithm that implements visual localization for mapped indoor environment. It analyses the original implementation of InLoc, e.g. the performance of particular parts. In conclusion, it describes potential improvements, specifically reimplementation of research version into the production version and performance improvement (the least efficient part will be reimplemented in C++).

Keywords: localization, InLoc, C++, Matlab

Title translation: Speedup the localization by InLoc method

Obsah

Obrázky

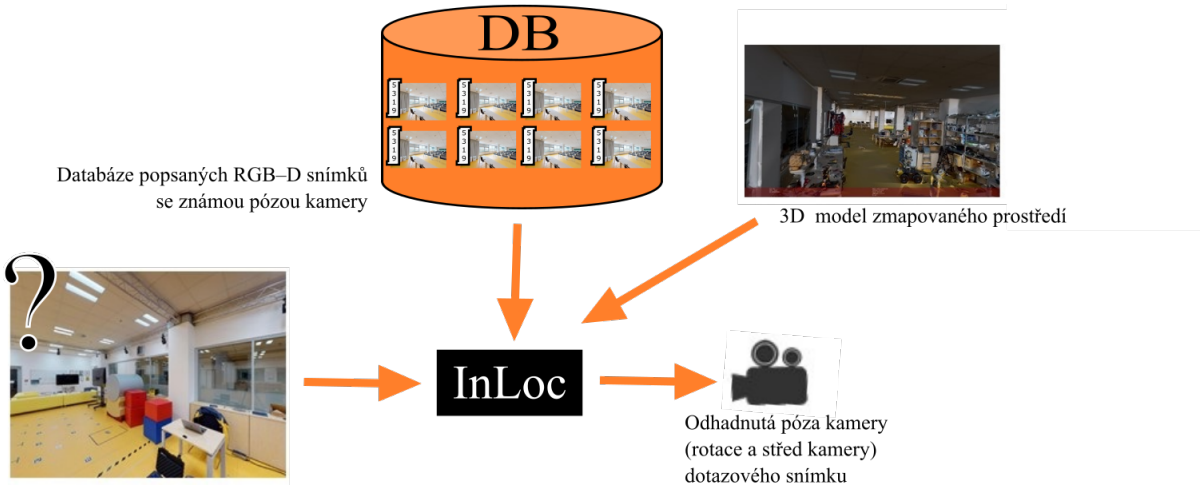
Tabulky

Pseudokódy

Kapitola 1

Úvod

InLoc je lokalizační algoritmus určený pro vnitřní prostředí odhadující pózu kamery (tj. souřadnice místa pořízení snímku a rotaci kamery) z jednoho (předem neznámého) dotazového snímku (query image). Odhad pózy kamery, tj. rotace a translace, je vypočítána na základě porovnání dotazového snímku s předem známými RGB-D snímky v databázi (RGB-D je RGB obraz + hloubka, tj. pro každý pixel známe jeho Eukleidovskou vzdálenost od středu kamery). U RGB-D snímků v databázi je póza kamery známa. Při výpočtu se nejdříve aplikuje konvoluční neuronová síť NetVLAD pro hledání podobných snímků.Husté korespondence se mezi snímky ověří pomocí homografií odhadnutých algoritmem DEGENSAC. Na základě vypočtených korespondencí se odhadne pomocí algoritmu PnP pozice kamery. Na závěr se vygenerují syntetické snímky a porovnají se s dotazovým snímkem. Tímto způsobem získáme finální pořadí, tj. vybereme pózu kamery vedoucí k nejmenší chybě DSIFT mezi dotazovým a syntetickým snímkem [1].



Obrázek 1.1: Vstupy a výstupy algoritmu InLoc.

1.1 Výhody a nevýhody lokalizace ze snímku

Lokalizace ze snímku má jen dva vstupy, tj. dotazový snímek a mapu prostředí reprezentovanou databází RGB-D snímků, pro něž je póza kamery známá. Proto je lokalizace ze snímku použitelná i tam, kde není dostupná GPS, zejména ve vnitřních prostorách. Prostory ani není potřeba pro lokalizační účely nijak upravovat. Oproti lokalizaci pomocí GPS nabízí lokalizace ze snímku podstatně lepší přesnost a navíc dokáže určit i rotaci kamery. Tyto algoritmy lze využít i pro jiné než lokalizační účely, např. v aplikacích virtuální reality [1] (promítání virtuálních objektů do skutečného prostoru vyžaduje velkou přesnost), nebo v aplikacích pro mapování prostředí do 3D mapy.

Lokalizace ze snímku má své limity a omezení. Je nutné vybudovat rozsáhlý dataset RGB-D snímků s přesnou pózou kamery. Tato lokalizace selhává v málo členitých prostředích (např. chodby, prázdné místnosti se stěnami bez textur apod.), může selhávat, pokud se dříve zmapované prostředí příliš změní (byly přemístěny některé předměty, nábytek apod.) a může být citlivá na změnu osvětlení (např. denní světlo se vymění za umělé osvětlení). Dále selhává v prostředích, kde se vyskytuje více vizuálně podobných míst (např. více chodeb se stejným rozmístěním dveří apod.) Lokalizace ze snímků pomocí algoritmu InLoc také není vhodná pro systémy reálného času, jako jsou například samořídící auta, jelikož současné implementace vyhodnocují několik časově náročných kroků.

1.2 Motivace

Algoritmus InLoc vyhodnocuje několik výpočetně náročných kroků, mimo jiné vyhledání podobných snímků, hledání tentativních korespondencí, popis dotazového snímku, nebo renderování 3D modelu. Algoritmus navíc klade důraz na přesnost, čímž výpočetní náročnost ještě stoupá (předpokládá výběr více podobných snímků z databáze a renderování více 3D modelů, čímž vzniká více výpočtů nutných pro odhad pózy kamery).

Současná implementace InLocu [2] byla s výjimkou několika funkcí napsána v Matlabu. Jedná se o interpretovaný jazyk, tj. kódy se nekompilují do spustitelných binárních souborů, ale jsou vyhodnocovány interpreterem při každém spuštění. Z toho vyplývá nižší výkon kvůli nemožnosti optimalizovat zdrojový kód, času potřebnému k interpretaci příkazů a správě pracovního prostoru. Na druhou stranu, Matlab interně využívá optimalizované matematické operace knihovny MKL. Matematické operace s velkými objemy dat dokáží nevýhody interpreteru vyvážit, protože práce s daty, která se provádí ve zkompilovaných knihovnách, zabere řádově více času než interpretace příkazů. Nicméně interpretované jazyky jsou pomalé, pokud je zapotřebí využít v kódu cykly, nebo mnoho příkazů s malými objemy dat.

Součástí práce je odhalení nejpomalejších částí kódu a jejich zrychlení vhodným přepisem do jazyka C++. Aby byl algoritmus připraven pro reálné využití, je nutné převést současnou testovací verzi na produkční verzi. Ta navíc může sloužit jako prototyp a předloha pro další implementace algoritmu InLoc v navazujících pracích.

1.2.1 Převod na produkční verzi

Současná implementace [2] , tak jak byla publikována, nesimuluje reálné využití algoritmu, a to z těchto důvodů:

1. Na vstupu jsou dvě předem známé množiny: databázové snímky RGB-D a rozsáhlá sada dotazových snímků. Při reálném využití jsou známy jen databázové snímky RGB-D a dotazový snímek je vždy předem neznámý. V produkční verzi tedy bude potřeba oddělit roli databázových a dotazových snímků.
2. V přípravném kroku pro popis snímků (skript buildFeatures) se ještě

před spuštěním algoritmu musí pomocí metody NetVLAD vypočítat popisné vektory pro všechny databázové snímky i pro všechny dotazové snímky. Při reálném využití je dotazový snímek vždy předem neznámý, a proto není možné jej předem popsat. Výpočet popisného vektoru dotazového snímku se musí stát součástí algoritmu, neboť dotazový snímek je vstupem algoritmu.

3. Pro každý předem známý dotazový snímek se spočítá jeho skóre podobnosti s každým databázovým snímkem již v přípravném kroku (skript buildScores) ještě před spuštěním samotného algoritmu. V reálném využití je výpočet podobností předem neznámého dotazového snímku nutnou součástí samotného algoritmu a následuje hned po spočítání popisu dotazového snímku.

4. Algoritmus je rozdělen do několika nezávislých kroků tak, že se v každému mezikroku ukládají mezivýsledky do souborů, aby bylo možné je v dalším kroku načíst. Proto je možné spustit jen některé části algoritmu. To je vhodné pro testovací a výzkumné účely, kde se zkouší jednotlivé části. V reálném využití poběží celý lokalizační algoritmus vždy od začátku do konce bez opakovaného ukládání a načítání mezivýsledků.

Implementace algoritmu bude ještě před měřením výkonu a následném zrychlení upravena podle těchto bodů tak, aby se více přiblížila reálnému využití.

1.2.2 Výkon Matlabu

Matlab je interpretovaný jazyk, tj. příkazy nejsou spouštěny přímo na procesoru, ale interpretace každého příkazu stojí nějaký čas. Samotný výpočet se často provádí pomocí zkompilovaných knihoven, které Matlab interně využívá. Z toho plyne fakt, že Matlab je optimalizován na hromadné zpracování dat, tj. maticové výpočty, při kterých interpretace příkazu trvá řádově kratší dobu než samotný výpočet ve zmíněných zkompilovaných knihovnách. Oproti tomu je využívání for cyklů, jejichž důsledkem je interpretace velkého množství drobných operací, řádově pomalejší. Experiment 7.3 porovnává výkon Matlabu s využitím maticových operací s použitím for cyklů.

Kapitola 2

Algoritmus InLoc

V této kapitole je popsán princip fungování algoritmu a původní testovací implementace v Matlabu.

2.1 Princip lokalizace

Dohoda: Označme počet dotazových snímků proměnnou q, počet databázových snímků proměnnou d. Množina databázových snímků bude značena ID a množina dotazových IQ.

Typicky bývá $d >> q$, protože pomocí databázových snímků je nutné kvalitně zmapovat celý prostor, kde se bude InLoc používat. V reálném využití uvažujeme $q = 1$. Algoritmus dostane na vstupu dotazové snímky $\{I_1, I_2, I_3, ..., I_q\}$, kde $I_i \in IQ$, u kterých je potřeba určit pózu kamery (tj. střed kamery C a rotaci R). Implementace se dá shrnout do následujících kroků:

1. InLoc pro každý dotazový snímek najde m nejpodobnějších snímků z databáze, takže každý dotazový snímek $I_i \in IQ$ bude mít přiřazenou množinu vybraných databázových snímků IDM_i tak, že $IDM_i \subseteq ID$ $(i \in 1, 2, ..., q)$ Podobnost je měřena pomocí $skóre_1$ 2.3. Výběr snímků se bude postupně zpřesňovat geometrickou verifikací a fotometrickou verifikací.)

2. Geometrická verifikace pro každý pár (I_{1i}, I_{2i}), kde $I_{1i} \in IQ$ a $I_{2i} \in IDM_i$ $(i \in 1, 2, ..., q)$, najde korespondující pixely pomocí porovnání popisů lokálních oblastí ve snímcích, které algoritmus dostal jako třetí a pátou vrstvu sítě VGG16 (podrobnější popis, jak jsou jednotlivé vrstvy použity, je v sekci 2.3.1). Mezi tentativními korespondencemi se najdou verifikované korespondence (inliery).

3. Algoritmus seřadí sestupně m vybraných nejpodobnějších snímků podle nového skóre (počtu inlierů z geometrické verifikace + $skóre_1$ (původní skóre z rovnice 2.3)) od nejvyššího skóre po nejnižší. Dále se zpřesní výběr snímků, tedy algoritmus vybere jen n nejpodobnějších snímků ($m > n$, v původní implementaci InLocu platí, že $m = 100$ a $n = 10$)

$$skóre_2 = skóre_1 + počet\_inlierů \tag{2.1}$$

   Na hodnotě nového skóre se podílí hlavně počet inlierů, protože $skóre_1 \in (0, 1]$ Výpočet $skóre_1$ je popsán v rovnici 2.3

4. Využije se toho, že databázové snímky jsou RGB-D (pro každý pixel je známa hloubka). Nalezené korespondující pixely dávají tedy 2D-3D korespondence. Díky tomu můžeme algoritmem P3P-RANSAC odhadnout pózu kamery. Algoritmus má v tomto kroku vybráno n nejpodobnějších databázových snímků, takže odhadne n póz kamery.

5. Dále je potřeba ohodnotit kvalitu n odhadnutých póz kamer a vybrat nejpřesnější odhad. Pro každou z n odhadnutých pozic kamer je renderován syntetický snímek. Syntetický snímek vznikne promítnutím 3D bodů z modelu prostředí do roviny odhadnuté kamery.

6. Vybere se jedna nejpřesněji odhadnutá póza kamery, pro kterou je syntetický snímek nejpodobnější dotazovému snímku. Podobnost syntetického snímku s dotazovým je určena pomocí $skóre_3$, které se vypočítá takto:

$$skóre_3 = 1/m_{err} \tag{2.2}$$

   kde m_{err} je medián vektoru chyb ($vektor\_chyb = ||M_q - M_s||$).

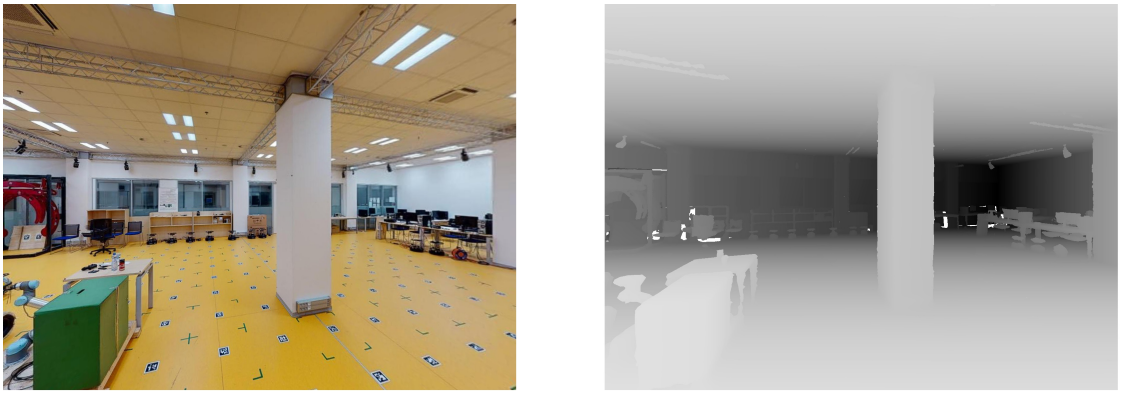
   M_q a M_s jsou popisné matice dotazového a syntetického snímku ($\mathbb{R}^{128\times10680}$) extrahované funkcí vl_phow z knihovny VLFeat [3].

   Čím vyšší $skóre3$, tím je menší rozdíl mezi dotazovým a syntetickým snímkem.

2.2 Databázové snímky (datasety)

Databáze snímků (také dataset) je množina RGB-D snímků, kde o každém snímku je známa póza kamery a eukleidovská vzdálenost každého pixelu od

středu kamery. Pomocí těchto snímků je tak možné reprezentovat 3D mapu známého prostředí. V této práci se používají snímky, které byly pořízeny v prostorách budovy CIIRC [1]. Algoritmus InLoc byl testován na čtyřech (v některých částech práce na devíti) datasetech s různou velikostí. Největší dataset obsahuje 2088 snímků, nejmenší 261 snímků. Větší dataset v sobě obsahuje vždy celý menší dataset a přidává další RGB-D snímky navíc. Každý dataset mapuje známé prostředí (prostory CIIRCu), ale kvalita zmapování se liší dle počtu obsažených snímků. V této práci bude popsán vliv velikosti databáze na přesnost odhadu pózy kamery.



Obrázek 2.1: Ukázka databázového snímku a jeho hloubkové mapy. Snímek pochází z datasetu [1]

2.3 Přehled kroků

Výše bylo ve zkratce popsáno šest kroků, jak InLoc funguje. V této části jsou detailně popsány skripty, ze kterých se InLoc skládá a které se spouští v uvedeném pořadí. Skripty víceméně korespondují s výše popsanými kroky.

2.3.1 Přípravné kroky

Nesouvisí přímo s algoritmem InLoc, ale připraví předpočítanou reprezentaci databázových snímků [4], která bude později využívána při hledání nejbližších sousedů.

1. **buildFeatures** - každý dotazový i databázový snímek se zpracuje neuronovou sítí NetVLAD. Výsledkem je vektor (feature), jehož cílem je jed-

noznačně popsat daný snímek vektorem reálných čísel délky $f = 32768$.

2. **buildScores** - spočítá skóre pro každou dvojici snímků ($I_1 \in IQ, I_2 \in ID$).

Popis snímků (skript buildFeatures)

Důležitým krokem algoritmu je hledání podobných snímků. Abychom mohli hledat snímky pořízené ze stejného místa, nemůžeme přímo porovnávat hodnoty intenzit pixelů, ale potřebujeme extrahovat důležitou informaci o podobnosti struktur ve snímcích obsažených. Této úloze se říká "image retrieval"a řeší jí řada algoritmů, např. BoW, VGG, VLAD, NetVLAD a další. InLoc využívá metodu NetVLAD [5] [6], která pracuje ve dvou krocích:

1. Konvoluční síť VGG16, kterou používá NetVLAD, vygeneruje pro každý snímek na šesti vrstvách popisné matice reálných čísel typu float, přičemž algoritmus InLoc využívá pouze třetí a pátou vrstvu. Na třetí vrstvě má matice rozměr 150x200x256, na páté vrstvě má rozměr 75x100x512.

2. Tyto popisné matice zpracuje VLAD (vectors of locally aggregated descriptors). Na výstupu je vektor o délce f (f je dimenze deskriptorů). Algoritmus InLoc standardně používá dimenzi $f = 32768$

Tento algoritmus je realizován ve skriptu buildFeatures. Skript nalezne popisný vektor stejné délky pro každý dotazový snímek i každý snímek z databáze. Podle popisných vektorů se bude měřit podobnost mezi snímky. Ke každému snímku (dotazovému i databázovému) se přiřadí deskriptivní vektor čísel stejné délky ($\mathbb{R}^f$). Tento vektor se následně použije pro výpočet obodování.

Vstupem jsou databázové snímky (d× soubor typu .jpg) a dotazové snímky (q× soubor typu .jpg)

Výstupem jsou extrahované popisné vektory o délce f pro každý dotazový i databázový snímek.

Výpočet matice skóre (skript buildScores)

Skóre dvojice (I_1, I_2), kde $I_1 \in IQ$ a $I_2 \in ID$, je reálné číslo z oboru $< 0; 1 >$. Čím vyšší skóre, tím podobnější snímky.

Matice skóre: Mějme množinu dotazových snímků $IQ = \{I_{Q1}, I_{Q2}, ..., I_{Qq}\}$ a množinu databázových snímků $ID = \{I_{D1}, I_{D2}, ..., I_{Dd}\}$.

Nechť $F_Q \in \mathbb{R}^{f \times q}$ a $F_D \in \mathbb{R}^{f \times d}$ jsou popisné vektory všech dotazových i databázových snímků extrahované sítí NetVLAD (normalizované vektory reálných čísel stejné délky f uspořádané do matic).

$$skóre_1 = softmax\_columnwise(F_Q^T F_D) \in \mathbb{R}^{q \times d} \tag{2.3}$$

Funkce softmax ováží každý sloupec matice skóre tak, že se budou skládat pouze z nezáporných čísel a hodnoty každého sloupce se budou sčítat do 1.

Vstupem jsou popisné vektory pro každý dotazový i databázový snímek.

Výstupem je matice skóre obsahující skóre pro každou dvojici snímků, což je matice $\mathbb{R}^{q \times d}$

2.3.2 Kroky algoritmu

Běh algoritmu byl rozdělen do tří navazujících skriptů, které jsou zde detailně popsány.

Výběr m nejpodobnějších snímků

Tuto úlohu vykonává skript ht_retrieval. Je potřeba znát matici obodování, kterou získáme voláním přípravného skriptu buildScores (2.3.1). Skript ht_retrieval pro každý dotazový snímek vybere m databázových snímků s nejvyšším obodováním. To znamená m nejpodobnějších databázových snímků ke každému dotazovému snímku. Jedná se o řazení řádků čísel (každý řádek

matice obodování je vektor, jehož čísla se seřadí, a indexy s nejvyšším skóre patří nejpodobnějším snímkům).

Vstupem je matice o rozměru počet dotazových snímků q krát počet databázových snímků d, tj. $M \in \mathbb{R}^{q \times d}$. Matice reprezentuje skóre mezi všemi dotazovými snímky a všemi databázovými snímky.

Výstupem je množina těchto proměnných:

1. ImgList = q× struktura s informacemi o dotazovém snímku a podobných nalezených snímcích. Přesněji:
   - queryname: název souboru dotazového snímku
   - topNname: m textových hodnot (názvy m nejpodobnějších snímků)
   - topNscore: m čísel float - obodování m nejpodobnějších snímků

Detaily funkce jsou popsány v pseudokódu [4].

Pseudokód 1: ht_retrieval (původní výzkumná implementace)

```
Vstup  : Matice skóre S (q×d)
Výstup : imgList (pro každý dotazový snímek m nejpodobnějších
         snímků)
if existuje_soubor(topm_nazev_souboru) then
    načíst(topm_nazev_souboru);
else
    imgList[m];
    for každý i-tý dotazový snímek I ∈ IQ do
        skore, top_indexy = seřadit_databázové_snímky_dle_skóre(S,
          i);
        imgList[i].topNscore = top_m_nejvyšších_skóre(skore);
        imgList[i].topNname =
          názvy_snímků_s_top_m_nejvyšších_skóre(top_indexy);
    end
    uložit(imgList);
end
```

Odhad póz kamery

Tuto funkci vykonává skript ht_top100_densePE_localization. Pro každý dotazový snímek odhadne n póz kamery.

Načtou se předpočítané popisné vektory pro všechny dotazové a databázové snímky. Pro každý databázový snímek z vybrané podmnožiny m snímků provede geometrickou verifikaci, která zjistí počet inlierů mezi dotazovým snímkem a databázovým snímkem. Následně přičte ke skóre každého databázového snímku jeho počet inlierů. Vzhledem k tomu, že původní skóre je výstup ze softmaxu (a proto nemůže být žádné $skóre_1$ vyšší než 1), ve tvorbě $skóre_2$ je důležitější počet inlierů. Následně proběhne seřazení nejlepších m snímků dle tohoto nového skóre. Pro n databázových snímků s nejvyšším skóre odhadne pomocí algoritmu P3P pózu kamery.

Vstupem je ImgList, což je výstup z předešlého kroku. Přesněji:

- ImgList = q×struktura s informacemi o dotazovém snímku a podobných nalezených snímcích:
  - queryname: název souboru dotazového snímku
  - topNname: m textových hodnot (názvy m nejpodobnějších snímků)
  - topNscore: m čísel float - obodování m nejpodobnějších snímků

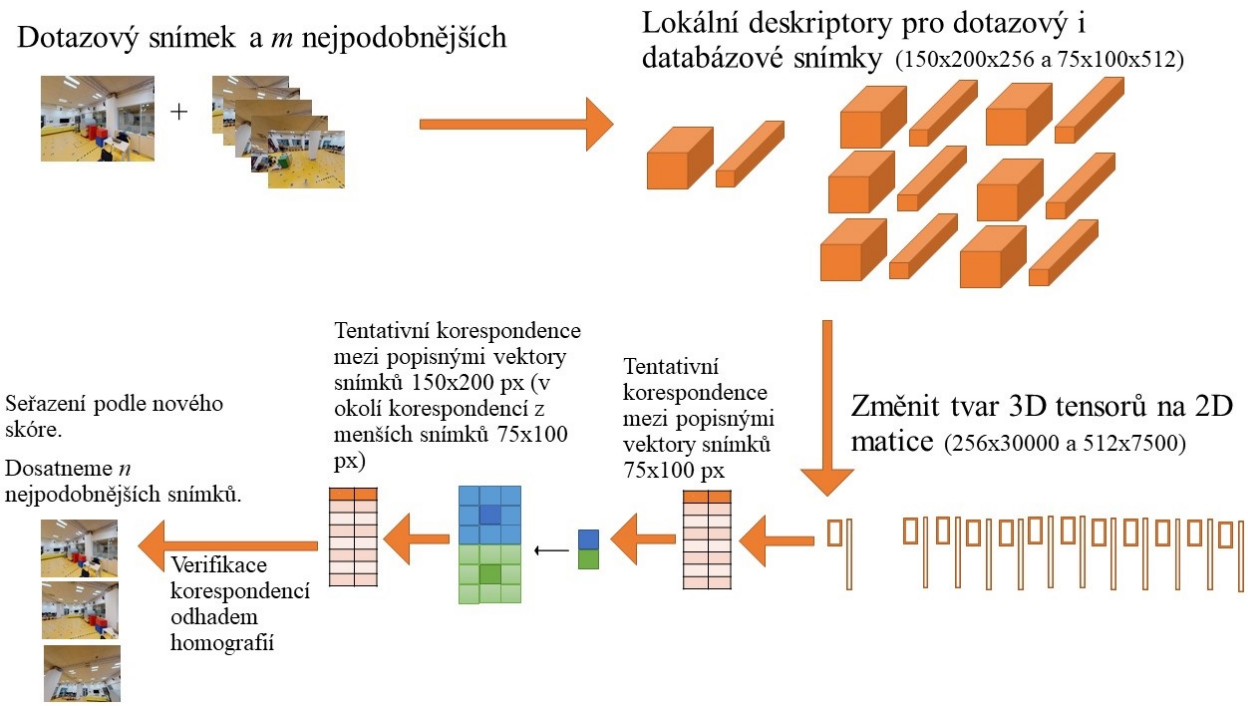
Výstupem je rozšířený ImgList:

- ImgList = q×struktura s informacemi o dotazovém snímku a podobných nalezených snímcích:
  - queryname: název souboru dotazového snímku
  - topNname: n textových hodnot (názvy n nejpodobnějších snímků)
  - topNscore: n čísel float - obodování n nejpodobnějších snímků
  - Ps: $n \times \mathbb{R}^{3\times4}$ - n odhadnutých póz

Skript je popsán v pseudokódu [2].

Pseudokód 2: ht_top100_densePE_localization (původní výzkumná implementace)

```
Vstup  : ImgList (pro každý dotazový snímek m nejpodobnějších
         snímků)
Výstup : ImgList (pro každý dotazový snímek n nejpodobnějších
         snímků a n odhadnutých póz)
if existuje_soubor(soubor_pro_imglist) then
    načíst(soubor_pro_imglist);
else
    for každý query snímek I_1 ∈ IQ do
        for každý i-tý (i=1...m) databázový nejpodobnější snímek
         I_2 ∈ IDM_i do
            gv = geometrická_verifikace (I_1, I_2);
            imgList[i].skóre += gv.počet_inlierů;
            imgList = seřadit_podle_skóre_sestupně(imgList);
            uložit(imgList);
            // zúžení výběru podle skóre m -> n
            imgList_topn = get_top_score(imgList, n);
            imgList_topn[i].Ps = p3p_odhadnout_n_poz(imgList, I_2);
            uložit(imgList_topn);
        end
    end
end
```



Obrázek 2.2: Ilustrace skriptu ht_top100_densePE_localization: hledání tentativních korespondencí mezi lokálními popisnými vektory pomocí popisných tensorů dvou různých rozlišení.

Verifikace odhadnutých póz

Tuto úlohu vykonává skript ht_top10_densePV_localization. Syntetizuje snímky ze 3D modelů prostředí dle odhadnutých póz kamery (krok pro odhad póz popsán v sekci 2.3.2). Upraví dosavadní ImgList tak, že seřadí vybrané databázové snímky podle kvality odhadnuté pózy. Kvalita je měřena pomocí výsledného skóre verifikace póz, které je popsáno v rovnici (2.2).

Vstupy.

- imgList spočítaný krokem pro odhad póz kamer (sekce 2.3.2)

Výstupy. Modifikovaný ImgList. Přesněji:

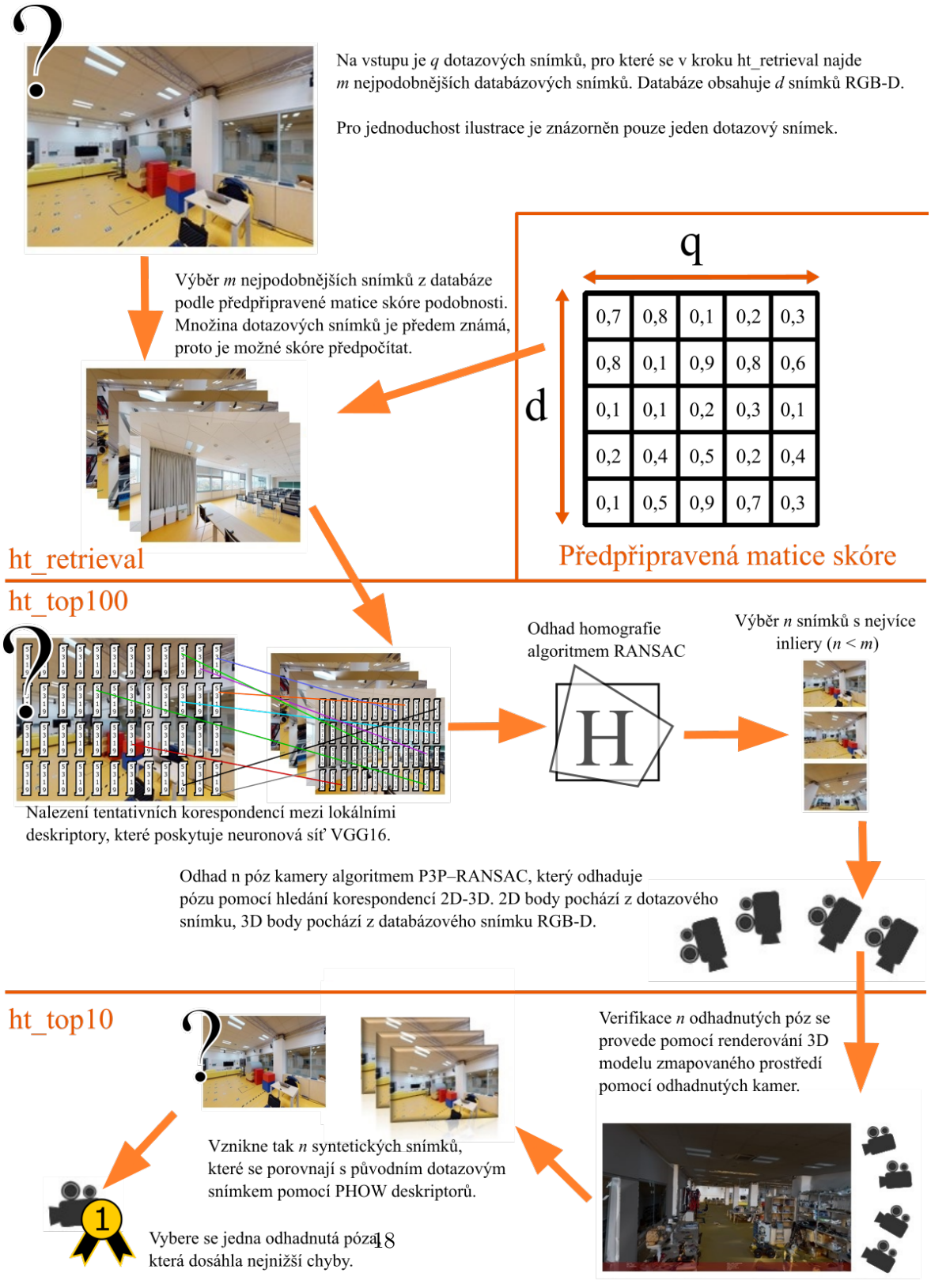
- queryname: název souboru dotazového snímku
- topNname: n×string (názvy n nejpodobnějších snímků)
- topNscore: n×float - obodování n nejpodobnějších snímků podle skóre vycházejícího z deskriptoru PHOW.
- Ps: $n \times \mathbb{R}^{3\times4}$ - n odhadnutých póz seřazených dle kvality

Detaily skriptu ht_top10_densePV_localization jsou popsány v pseudokódu [3]

Pseudokód 3: ht_top10_densePV_localization (původní výzkumná implementace)

```
Vstup  : ImgList (pro každý dotazový snímek n nejpodobnějších
         snímků a n odhadnutých póz)
Výstup : ImgList (pro každý dotazový snímek n nejpodobnějších
         snímků a n odhadnutých póz seřazeno podle přesnosti)
if existuje_soubor(soubor_pro_imglist) then
    načíst(soubor_pro_imglist);
else
    for každý query snímek I_1 ∈ IQ do
        for každý i-tý (i=1...n) nejpodobnější snímek I_2 ∈ ID do
            [rgb_synth, xyz] = render3D(meshPath, camera_params);
            // vektor rozdílů deskriptorů DSIFT
            err = get_dsift_error(I_1, rgb_synth);
            score = 1 / median(err);
        end
        seřazené_imgs =
          seřadit_podle_nového_skóre_sestupně(imgList);
        uložit(seřazené_imgs);
    end
end
```

2.4 Souhrnné schéma původní testovací verze



Obrázek 2.3: Souhrnné schéma původní testovací verze algoritmu InLoc

Kapitola 3

Převod testovací verze na produkční

Původní implementace algoritmu InLoc je vhodná pro výzkumné a testovací účely, ale špatně simuluje skutečné využití algoritmu. Tato kapitola popisuje nově vzniklou produkční implementaci, která simuluje reálný běh. Právě tato vzniklá verze bude spuštěna za účelem detailních profilací, což odhalí nejpomalejší části kódu a nabídne tak možné varianty k přepisu do C++. Tato práce tak poskytuje návazný bod pro budoucí produkční implementace InLocu mimo prostředí Matlab.

3.1 Oddělení role databázového a dotazového snímku

Ve výzkumné verzi byla množina dotazových snímků předem známá, což neodpovídalo potřebám reálného provozu, kde je známá pouze množina databázových snímků. V produkční verzi bude zvolen jiný přístup: množina dotazových snímků nebude z projektu odstraněna, bude se dále používat pro testování, ale změní se způsob práce s ní. Pro každý testovací běh algoritmu se zvolí právě jeden dotazový snímek, který bude považován za předem neznámý. Proto bude výpočet skóre podobnosti s databázovými snímky pro tento dotazový snímek zahrnut do algoritmu. Jinými slovy, skóre podobnosti mezi dotazovým a databázovými snímky nemůže být předpočítáno předem před spuštěním algoritmu.

3.2 Nový formát matice skóre

Ve výzkumné verzi dávalo smysl předpočítat matici skóre kvůli urychlení testování dalších kroků. Toto v reálném provozu není možné. Dotazový snímek je pouze jeden a je předem neznámý, proto výpočet skóre není přípravným krokem, ale součást prvního kroku algoritmu. Matice skóre už nemá rozměr $q \times d$, ale pouze $1 \times d$ (skóre je spočítáno pro jeden dotazový snímek s každým databázovým snímkem).

3.3 Reorganizace skriptů

3.3.1 buildScores

Skript buildScores, který spočítá skóre podobnosti pro každý dotazový snímek s každým databázovým snímkem, se již nebude používat. Důvodem je již zmíněné odlišení role databázových a dotazových snímků (skript buildScores předpokládal existenci předem známé množiny dotazových snímků a spouštěl se jako přípravný krok datasetu mimo samotný běh algoritmu.) Místo tohoto skriptu byla do projektu přidána funkce getScores1Query.

```
function [score] = getScores1Query(params, featuresPath,
    queryPath, cutoutFeatures)
```

Vstupy a výstupy:

- params jsou parametry běhu algoritmu vytvořené funkcí setupParams ještě před krokem ht_retrieval.
- featuresPath je cesta k popisným vektorům databázových snímků
- queryPath je cesta k dotazovému snímku

Funkce getScores1Query potřebuje extrahovat features předem neznámého dotazového snímku, proto interně volá funkci getFeatures1Query. Tato funkce byla do projektu přidána kvůli přechodu na produkční verzi.

3.3.2 getFeatures1Query

Funkce getFeatures1Query spočítá features pro jeden předem neznámý dotazový snímek. Tento skript v původní výzkumné verzi InLocu nebyl. K extrakci popisných features používá stejný algoritmus jako přípravný skript buildFeatures.

Funkce byla testována manuálně, zda vrací pro stejné snímky stejné popisné vektory jako funkce buildFeatures.

```
function [ queryFeatures ] = getFeatures1Query(params, queryPath)
```

Vstupy a výstupy:

- params jsou parametry běhu algoritmu vytvořené funkcí setupParams ještě před krokem ht_retrieval.
- queryPath je cesta k dotazovému snímku
- queryFeatures je jednotkový popisný vektor dotazového snímku.

3.4 Možnosti ladění

Pro testovací implementaci je typické odkládání mezivýsledků do souborů, čímž se proces lokalizace rozděluje na několik nezávislých kroků. Díky tomu se nemusí spouštět celý proces vždy od začátku, ale je možné začít z libovolného místa. Naopak produkční implementace předpokládá v rámci simulace reálného běhu vždy spouštění od začátku a nedělitelnost celého procesu. V reálném běhu tak odpadá potřeba odkládat mezivýsledky do souborů

Kvůli testování produkční verze, exportování grafů, profilací a tabulek ale není možné odkládání do souborů úplně zrušit. Simulaci reálného běhu a odkládání výsledků řídí tyto booleovské proměnné, které jsou definovány na začátku algoritmu ve funkci inloc_demo:

- USE_CACHE_FILES: umožňuje rozdělit proces lokalizace do nezávislých kroků, tj. pokud existují odložené mezivýsledky nějakého kroku algoritmu, tento krok se může přeskočit. Tato proměnná je při všech profilacích nastavená na hodnotu false, v opačném případě by algoritmus nesimuloval reálný běh.

- SAVE_SUBRESULT_FILES: umožňuje odkládat mezivýsledky do souborů. Tyto mezivýsledky jsou nezbytné pro generování grafů a tabulek, proto je tato možnost vždy zapnutá.

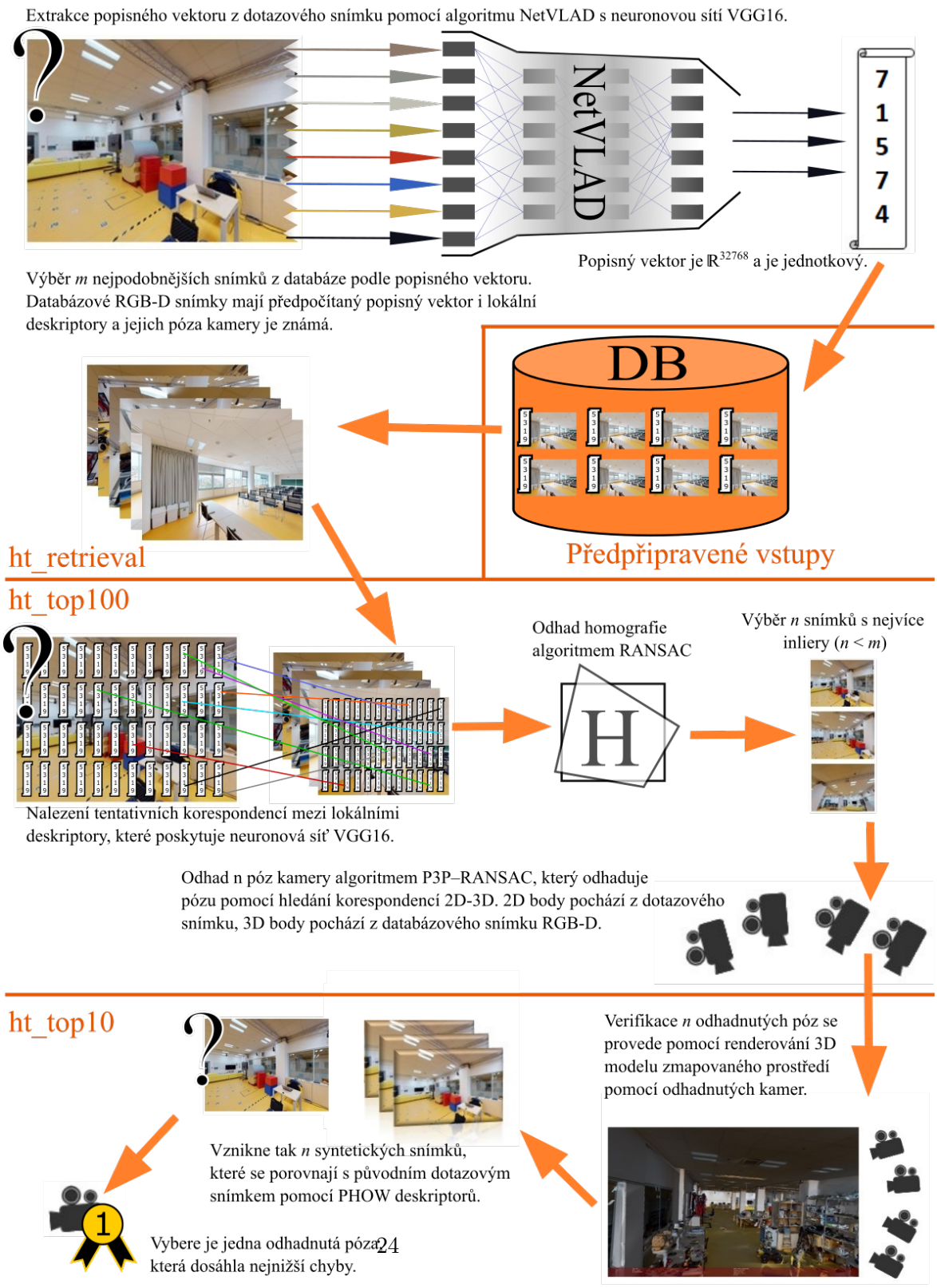
- USE_PAR: umožňuje spouštět vhodné podúlohy paralelně. To je ve většině případů žádoucí, ale pokud je potřeba pomocí profilace změřit výkon všech funkcí, je vhodné paralelní běh vypnout. Paralelní kód se v Matlabu profiluje jako atomická jednotka, takže ve výsledné profilaci není vidět, které funkce se v paralelních částech spouštěly a jaký mají výkon.

- USE_PROFIL: pokud je tato možnost zapnutá, kód se profiluje vestavěným profilačním nástrojem Matlabu a po skončení algoritmu se výsledky profilace uloží ve formátu html.

3.5 Význam produkční verze

Produkční verze je varianta algoritmu InLoc, která simuluje reálné využití ve skutečném provozu. Zatím se jedná jen o simulaci. Tato implementace ještě není na reálné využití připravena, protože je závislá na prostředí Matlabu, což s sebou nese nejen výkonnostní omezení, ale i omezení využívání hardwaru, zejména paralelismu, protože paralelní cykly vyžadují licenci na speciální komerční toolbox aplikace Matlab. Tato implementace může pro navazující práce (např. kompletní přepisování algoritmu do C/C++) sloužit jako předloha a prototyp.

3.6 Souhrnné schéma produkční verze



Obrázek 3.1: Souhrnné schéma produkční verze algoritmu InLoc

Kapitola 4

Zrychlení produkční verze pomocí C++ a GPU

Součástí této práce je zvýšit výkon algoritmu InLoc. Dosavadní implementace je napsána ve skriptovacím jazyce Matlab, což s sebou nese výkonnostní omezení. Nabízí se proto přepsat některou pomalejší část do C++. Vybraná přepsaná část bude potřebovat propojení s prostředím Matlab, ve kterém poběží ostatní implementace. Kvůli nutné vazbě na Matlab se nabízí zkompilovat kód C++ jako knihovnu MEX.

4.1 Knihovny MEX

Soubory Matlab executable (zkráceně MEX) představují způsob, jak spouštět v prostředí Matlabu kódy, které jsou napsány v jazyce C nebo C++. Jedná se o dynamické knihovny, což jsou binární soubory obsahující zkompilovaný kód, které se načítají za běhu aplikace. Soubory MEX tak fungují na podobném principu jako dynamické knihovny .dll v systému Windows či .so v systému Linux. Knihovna MEX obsahuje právě jeden vstupní bod (tj. hlavní funkci), který se volá z prostředí Matlabu a poskytuje základní rozhraní pro spolupráci Matlabu s nativním kódem. Každá knihovna MEX tak zastupuje jednu funkci. V Matlabu se knihovna volá vlastním názvem. Např. pokud máme soubor knihovny my_mex_library.mexa64, v Matlabu ji můžeme zavolat stejně jako běžnou funkci, tj. příkazem my_mex_library(*parametry*); . Tímto příkazem se zavolá hlavní funkce knihovny, které se předají ukazatele na vstupní a výstupní parametry.

Knihovna MEX se dá naimplementovat v jazyce C nebo C++, přičemž způsob implementace se značně liší. Implementace C obsahuje hlavní funkci, C++ obsahuje třídu reprezentující funkci Matlabu. Tyto rozdíly se týkají jen API Matlabu, takže je samozřejmě možné použít API jazyka C a zbytek knihovny napsat v C++. Podpora pro C++ MEX API byla do Matlabu přidána až ve verzi R2018a. Pro tuto práci byla vybrána implementace C, protože je kompatibilní i se staršími verzemi Matlabu. Každá knihovny MEX vydaná v rámci této práce tak bude obsahovat hlavní funkci mexFunction.

4.1.1 Implementace MEX v C

Potřebná rozhraní pro komunikaci s Matlabem jsou definována v hlavičkovém souboru mex.h. Vstupním bodem knihovny je funkce mexFunction, která má následující definici:

```
void mexFunction(int nlhs, mxArray* plhs[],
    int nrhs, const mxArray* prhs[])
```

nlhs je počet výstupních parametrů, které jsou přístupné pomocí ukazatelů *plhs*. Obdobně jsou přístupné i vstupní parametry pomocí parametrů *nrhs* a *prhs*.

4.1.2 Implementace MEX v C++

Potřebná rozhraní pro komunikaci s Matlabem jsou definována v hlavičkovém souboru mex.hpp. Vstupním bodem knihovny je třída knihovny, což je třída nesoucí název funkce a je odvozena od základní třídy matlab::mex::Function. Třída musí definovat operátor () následujícím způsobem:

```
void operator()(matlab::mex::ArgumentList vystupy,
   matlab::mex::ArgumentList vstupy) {...}
```

Třída ArgumentList je kontejner obsahující hodnoty typu matlab::data::Array. Kontejner je možné použít s běžným operátorem indexování a navíc si pamatuje vlastní délku, díky čemuž má operator() pouze dva parametry (tj. oproti implementaci C chybí parametry popisující délku polí).

4.1.3 Kompilace MEX

Existuje více příkazů Matlabu, které zkompilují soubory C/C++. V této práci byly použity příkazy mex a mexcuda. Příkaz mexcuda byl použit pro kompilaci implementace s knihovnami cuBLAS a CUDA. Dokáže kompilovat soubory .cu, které obsahují kernely CUDA. Každý z příkazů spustí interně nějaký kompilátor pro C/C++. Pro tuto práci byl použit G++ pro příkaz mex a nvcc pro příkaz mexcuda.

Oba příkazy byly použity ve stejném tvaru:

```
mex <soubory .cpp> -I "<hlavičkové soubory .h>"
CXXFLAGS="<parametry kompilátoru>" <statické knihovny>

mexcuda <soubory .cpp> -I "<hlavičkové soubory .h>"
CXXFLAGS="<parametry kompilátoru>" <statické knihovny>
```

Příkaz definuje použité zdrojové a hlavičkové soubory C++ a seznam nalinkovaných knihoven. Parametry kompilátoru nastavují např. použitou verzi C++.

4.2 Výběr vhodných částí k přepisu do C++

Pro výběr vhodné části k reimplementaci v C++ bylo nutné provést profilace celého běhu algoritmu s různě rozsáhlými datasety databázových snímků. Detailní výsledky všech profilací jsou v kapitole Experimenty v sekci 7.6. Většina kroků algoritmu InLoc pracuje s předem definovaným množstvím dat (např. 100 nejpodobnějších snímků, 10 syntetických snímků apod.). Proto by velikost databáze snímků neměla mít vliv na časovou náročnost těchto kroků. Velikost databáze snímků má vliv pouze na časovou náročnost skriptů, které pracují s celou databází. Profilace tento fakt potvrdila. Jediné skripty, které pracují s celou databází, je přípravný krok buildFeatures, který popíše databázové snímky metodou NetVLAD, a první krok algoritmu ht_retrieval, který z databáze vybere množinu nejpodobnějších snímků. Krok buildFeatures slouží k přípravě datasetu a v samotném algoritmu InLoc se nespouští, proto není vhodný k přepsání. Krok ht_retrieval je v porovnání s ostatními kroky menší zátěží, a proto nebyl vybrán k přepsání.

Hlavní kritéria pro výběr skriptu k přepsání do C++ jsou:

1. Procentuální podíl doby běhu daného skriptu na celkové době běhu algoritmu. Dle profilace mají obvykle často volané funkce větší podíl na celkové době běhu, a proto je jejich optimalizace důležitější.
2. Způsob původní implementace vybrané části - např. kódy s velkým množstvím for cyklů se budou přepisovat jednodušeji, protože jazyk Matlab není pro tyto operace optimalizován. Oproti tomu je složitější optimalizovat kód, který využívá převážně optimalizovaných maticových operací.

K přepisu byla vybrána funkce at_dense_tc. Důvody pro výběr této funkce k přepisu jsou následující:

1. Jednovláknová profilace (viz. sekce 7.6) ukázala, že funkce je spouštěna zhruba 27000x a celková doba běhu převyšuje dvě minuty, což představuje zhruba šestinu z celkové doby běhu algoritmu.
2. Funkce využívá funkci yael_nn z knihovny Yael, která řeší problém nejbližších sousedů. V dosavadní implementaci je ale yael_nn volána dvakrát se stejnými daty, jen s opačným pořadím parametrů. Tím se implementace vyhnula použití for cyklů, které by představovaly ještě větší zátěž.
3. Tato funkce nachází tentativní korespondence mezi normovanými vektory reálných čísel, přičemž často pracuje s velkými objemy dat. Experimenty ze sekce 7.7 ukázaly, že nové implementace v C++ s využitím GPU, vhodných matematických knihoven a paralelizace cyklů vykazuje násobně lepší výkon než původní implementace.

Funkce at_dense_tc bude přepsána do jazyka C++ s využitím optimalizovaných matematických knihoven. Funkce byla napsána ve více verzích s použitím různých matematických knihoven. Poté byla otestována rychlost běhu každé varianty funkce na stejných vstupních datech.

4.3 Výběr matematických knihoven

Pro nově vzniklý projekt C++, který obsahuje zrychlenou implementaci pro vybranou část algoritmu InLoc, je potřeba vybrat vhodnou matematickou

knihovnu pro maticové operace. Bude vybrána taková knihovna, která v dané úloze poskytne nejlepší výkon.

4.3.1 Specifikace BLAS

Specifikace BLAS [7] implementuje funkce pro mnoho základních matematických operací, přičemž se zaměřuje na operace s vektory a maticemi. BLAS je zaměřena na výkon - je implementována nízkoúrovňově (tj. pracuje se s alokací paměti, se kterou se pracuje pomocí ukazatelů. Zcela chybí automatizace na úrovni objektů.) BLAS je navržena tak, aby mohla těžit z výkonu hardwaru. K největším výhodám patří využívání SIMD instrukcí (single instruction, multiple data), což je datový paralelismus, který dokáže násobně urychlit výpočty. Např. je možné pracovat najednou s 256 bity, což je 8 čísel typu float najednou. Podporovány jsou jazyky C (C++) a Fortran.

Funkce BLAS jsou rozděleny do několika úrovní:

1. Level 1 poskytuje pouze operace s vektory (výpočet normy, skalární součin, sčítání aj.).
2. Level 2 poskytuje funkce matematických operací mezi maticemi a vektory (např. násobení), dále implementuje algoritmy pro řešení lineárních rovnic aj.
3. Level 3 implementuje matematické operace mezi maticemi. Typickou úlohou je součin dvou matic.

4.3.2 Knihovna cuBLAS

Knihovnu cuBLAS vyvinula společnost NVIDIA. Jedná se o kombinaci rozhraní BLAS a knihovny CUDA, která je určena pro programování výpočtů na GPU [8].

Výhody knihovny cuBLAS.

1. Rozsáhlé maticové výpočty běží na GPU podstatně rychleji

2. Knihovna poskytuje funkce napsané v jazyce C a technologie CUDA je využívána interně. Proto není nutné ji znát.
3. cuBLAS je možné doplnit o vlastní kernely CUDA, nebo o libovolné jiné funkce, které CUDA poskytuje.
4. Správnost manipulace s pamětí je možno ověřit nástrojem cuda-memcheck
5. cuBLAS poskytuje jednoduché funkce pro hromadné zpracování dat na více vláknech.

Nevýhody knihovny cuBLAS.

1. Je nutné kopírovat data do GPU a z GPU.
2. Programy obsahující knihovnu cuBLAS mohou plnohodnotně běžet pouze na strojích s grafickou kartou.
3. Knihovna je napsána v C, proto je nutné buď spravovat paměť manuálně (alokovat a uvolňovat paměť, kopírovat paměť, udržovat rozměry matic apod.), nebo vyvinout vlastní sadu tříd C++, které mohou tuto nutnou administrativu automatizovat.

4.3.3 Knihovna MKL

MKL vznikla jako implementace rozhraní BLAS od společnosti Intel [9], ale využití této knihovny je širší. Stejně jako BLAS a cuBLAS poskytuje operace lineární algebry ve třech úrovních (vektor a vektor, matice a vektor, matice a matice). Navíc poskytuje funkce pro vědecké výpočty, Fourierovy transformace, statistiku, generátory pseudonáhodných čísel aj. V tomto projektu byly využity pouze její funkce pro lineární algebru.

Nová implementace algoritmu InLoc využívá MKL kvůli optimalizovaným funkcím lineární algebry.

Výhody knihovny MKL.

1. Používání MKL je jednodušší než používání knihovny cuBLAS, kde se musí přenášet data mezi GPU a CPU, zarovnávat paměť atd.

2. MKL poskytuje jednoduché funkce pro hromadné zpracování dat na více vláknech.

Nevýhody knihovny MKL.

1. Stejně jako knihovna cuBLAS, MKL počítá s manuální správou paměti. Špatná manipulace s knihovnou proto může vést k únikům paměti, neplatným přístupům do paměti apod.

4.3.4 Knihovna Eigen

Knihovna Eigen implementuje operace lineární algebry. Na rozdíl od zmíněných knihoven postavených na BLAS je implementována jinak. Jedná se o soubor tříd a šablon tříd jazyka C++, které zaobalují nízkoúrovňové operace.

Výhody knihovny Eigen.

1. Objektový kód C++ knihovny Eigen je více automatizovaný, např. volání destruktorů objektů automaticky uvolňuje alokovanou paměť. Programátor je tak odstíněn od práce s pamětí a hardwarem.
2. Objektový kód poskytuje intuitivnější zápisy. Umožňuje používat přetížené operátory a členské funkce objektů.
3. Díky využití šablon typů může být mnoho důležitých informací známo už při kompilaci. Pokud např. dojde k násobení matic nekompatibilních rozměrů nebo nekompatibilních typů, program se vůbec nezkompiluje.
4. Objekty matic si uchovávají informace o vlastním rozměru. Při operacích násobení není nutné zadávat je manuálně a alokovat pro výsledek pole správného rozměru.
5. Operace vyšší úrovně nabízí širší možnosti optimalizace. Pokud např. provedeme transpozici matice, nebo replikaci matice, tato operace se ve skutečnosti vůbec provést nemusí, ale systém šablon tříd umí simulovat požadované chování.
6. Eigen implementuje líné vyhodnocování, což umožňuje neprovádět náročné operace, dokud to není nutné.

7. Eigen je implementována jako množina hlavičkových souborů. Do projektu se zařazuje velmi jednoduše, není potřeba nic instalovat.
8. Eigen se dá zkompilovat tak, aby interně využívala knihoven MKL, OpenBLAS apod.

Nevýhody knihovny Eigen.

1. Knihovna Eigen nenabízí analogii ke každé funkci BLASu. Např. nenabízí analogii k funkci sgemm_batch, kterou MKL a cuBLAS poskytují.

4.4 Popis funkce at_dense_tc

Tato funkce byla vybrána jako vhodný kandidát na přepis do C++. Pro odlišení dostala nová implementace této funkce nové jméno. Nová funkce se vyskytuje ve třech variantách: get_tcs_mkl, get_tcs_eigen a get_tcs_cublas. Každá varianta obsahuje v názvu použitou matematickou knihovnu.

Funkce at_dense_tc hledá tentativní korespondence mezi dvěma snímky. Funkce je napsána v Matlabu s následujícími vstupními a výstupními parametry:

```
function match = at_dense_tc(desc1,desc2)
```

*desc*1 a *desc*2 jsou matice typu $\mathbb{R}^{7500\times512}$ a obsahují 7500 lokálních popisných vektorů oblasti snímku, kde každý vektor je normovaný vektor typu $\mathbb{R}^{512}$.

match je matice $\mathbb{R}^{2\times t}$, kde t je počet nalezených tentativních korespondencí mezi dvěma snímky. Každý sloupec obsahuje dva indexy popisných vektorů, které tvoří tentativní korespondence. Platí, že dva popisné vektory tvoří tentativní korespondenci, pokud jsou si vzájemně nejpodobnější. Podobnost dvou popisných vektorů $d_1, d_2 \in \mathbb{R}^{512}$ se měří skalárním součinem:

$$podobnost = d_1^T d_2 \in \mathbb{R} \tag{4.1}$$

d_1 a d_2 jsou normované vektory, takže jejich podobnost je ekvivalentní jejich úhlové vzdálenosti.

Odlišnosti v implementaci at_dense_tc a get_tcs_*

Není pravda, že funkce get_tcs_* by byla naprosto přesnou analogií k funkci at_dense_tc.

Původní funkce at_dense_tc se dala použít pro vektory s libovolnou normou, zatímco všechny nové implementace počítají s normovanými vektory na vstupu. Při porušení této podmínky budou nové funkce get_tcs_* vracet chybné výsledky. Tato odlišnost v implementaci se v algoritmu InLoc nijak neprojeví, protože lokální deskriptory (tj. vstupní parametry) jsou normované vektory.

Důležitý rozdíl je, že původní funkce at_dense_tc hledá tentativní korespondence mezi jedním dotazovým a jedním databázovým snímkem. Všechny nové funkce hledají tentativní korespondence mezi jedním dotazovým a libovolným počtem databázových snímků. Tato úprava implementace přispívá ke zvýšení výkonu efektivním využitím paralelismu, ale vyžádala si reorganizace algoritmu.

Paměťové testy s cuda-memcheck

Při použití knihoven cuBLAS a CUDA je nutná manuální správa paměti pomocí ukazatelů a knihovních funkcí. Pro alokaci a uvolnění paměti GPU jsou k dispozici funkce cudaMalloc a cudaFree. Data se kopírují do a z paměti GPU funkcí cudaMemcpy. Protože zmíněné knihovny nezaručují bezpečnou manipulaci s pamětí, je nutné využít nástroj cuda-memcheck, který spustí zkompilovaný kód v testovacím prostředí a za běhu detekuje chybné paměťové operace. Mezi detekované problémy patří úniky paměti, dvojité uvolnění (double free), porušení hranice pole, chybné zarovnání paměti a různé problémy hardwaru. Tyto chyby se detekují jak pro paměť GPU, tak pro paměť RAM. Nástroj CUDA-memcheck se spouští v příkazové řádce příkazem:

```
cuda-memcheck <spustitelný soubor>
```

Pro účely testování paměti je vhodné zkompilovat implementaci get_tcs_cublas tak, aby nebyla závislá na Matlabu, tedy místo knihovny MEX je vhodné zkompilovat spustitelný program. Pro tuto testovací kompilaci byl ignorován soubor get_tentative_corresps_MEX.cpp, který zajišťoval napojení na prostředí Matlab. Byl nahrazen souborem test_cuda_code.cpp, který obsahuje funkci main a po spuštění vygeneruje náhodné matice, které se použijí jako vstup do testované funkce get_tcs. Následující výstup paměťového testu ukazuje úspěšný výsledek:

```
$ cuda-memcheck test_mem_get_tcs_cublas

========= CUDA-MEMCHECK
Result len: 4
========= ERROR SUMMARY: 0 errors
```

Paměťové testy s valgrindem

V implementaci get_tcs_mkl se používá manuální správa paměti. Samotná knihovna MKL je napsaná v jazyce C. Navíc se v implementaci vyskytují dynamické alokace polí. Proto je nutné zkontrolovat korektnost paměťových operací pomocí nástroje valgrind.

```
$ valgrind test_mem_get_tcs_mkl
```

Pomocí tohoto nástroje byl testován každý build implementace. Stejně jako při testování varianty get_tcs_cublas, i varianta MKL byla pro testovací účely zkompilována do spustitelného programu.

CI/CD

CI/CD do projektu přidáno nebylo, a to z těchto důvodů

- Na Git byl nahráván pouze kód, který se úspěšně přeložil na knihovnu MEX. Překlad provádí Matlab pomocí příkazů mex a mexcuda.

- Šířit Matlab pro potřeby CI/CD je nepraktické a navíc nelegální. Pro běh Matlabu je potřebná platná licence.

- Testy lze spustit manuálně. Stačí spustit jednou pro každé sestavení.

4.4.1 Test správnosti převodu do C++

Všechny varianty nové funkce (tedy get_tcs_cublas, get_tcs_mkl, get_tcs_eigen) musí být otestovány, zda vrací ty stejné, nebo alespoň podobné výsledky jako at_dense_tc. Malé zaokrouhlovací chyby jsou přirozené a dají se tolerovat. Větší odlišnosti v počtu nalezených tentativních korespondencí či odlišnosti v jejich spárování jsou nepřípustné.

Použitá testovací data jsou reálné popisné matice použitých snímků, konkrétně 100 párů popisných matic dotazového snímku s databázovým. Každá popisná matice má rozměr 7500×512, což odpovídá 7500 lokálním popisným vektorům typu $\mathbb{R}^{512}$. Výsledkem funkce at_dense_tc i každé její nové alternativy je množina párů reprezentovaná jako dvouřádková matice.

Pro testování rovnosti výsledků byla vytvořena v Matlabu funkce TEST_MEX_FILES (umístnění: _InLoc_PROD_Speedup/functions/at_netvlad_function). Funkce nepřijímá žádné parametry a nevrací žádné výsledky, jen vypíše nalezené chyby. Tentativní korespondence nalezené původní funkcí Matlabu at_dense_tc se považují za referenční správné řešení a za chybu se považuje, pokud nová implementace vrátí nějakou korespondenci navíc, pokud nějaká korespondence

chybí a pokud je nějaká korespondence jinak přiřazena.

Pseudokód 4: Způsob vyhodnocení tentativních korespondencí mezi jedním dotazovým a mnoha databázovými snímky

```
desc_q, desc_dbs[] = načíst_tent_korespondence();
mkl_errors = 0; cublas_errors = 0; eigen_errors = 0;
for každý desc_db ∈ desc_dbs do
    // Funkce to7500 přijímá páry indexů deskriptorů, které tvoří
      tentativní korespondence. Funkce vrácí vektor o délce 7500, kde na
      každém indexu i je celé číslo j (index deskriptoru databázového
      snímku, se kterým se páruje i-tý deskriptor dotazového snímku).
      Pokud j=0, tak i-tý deskriptor nemá přiřazenou žádnou
      korespondenci.
    tcs_original = to7500(at_dense_tc(desc_q, desc_db));
    tcs_mkl = to7500(get_tcs_mkl(desc_q, desc_db));
    tcs_eig = to7500(get_tcs_eigen(desc_q, desc_db));
    tcs_cbl = to7500(get_tcs_cublas(desc_q, desc_db));
    // Počet chyb = počet jinak přiřazených indexů
    mkl_errors += sum(tcs_original != tcs_mkl);
    eigen_errors += sum(tcs_original != tcs_eig);
    cublas_errors += sum(tcs_original != tcs_cbl);
end
report_errors();
```

Protože tentativních korespondencí může být maximálně 7500, je snadné převést úlohu spočítání rozdílů v tentativních korespondencích na úlohu porovnání dvou vektorů u, v o délce 7500, kde se spočítá množství indexů i, pro které platí: $u[i] \neq v[i]$. Výstup testu nových implementací je následující:

```
    >> TEST_MEX_FILES
Pocet nalezenych TC se v 55. paru lisi [Matlab, MKL, cuBLAS, Eigen]:
   393   393   394   393

cuBLAS: Ve dvojici 55 je jinak prirazenych korespondenci: 1
Pocet nalezenych TC se v 57. paru lisi [Matlab, MKL, cuBLAS, Eigen]:
   328   328   329   328

cuBLAS: Ve dvojici 57 je jinak prirazenych korespondenci: 1
Pocet nalezenych TC se v 99. paru lisi [Matlab, MKL, cuBLAS, Eigen]:
   287   287   286   287

cuBLAS: Ve dvojici 99 je jinak prirazenych korespondenci: 1
TEST skoncil
```

Z toho vyplývá, že 97 testovaných případů vrací stejný počet stejně přiřazených tentativních korespondencí. Pouze ve třech případech vrátila implementace s knihovnou cuBLAS mírně odlišné výsledky. V případě č.55 došlo k navýšení počtu tentativních korespondencí z 393 na 394. To samé navýšení se vyskytlo ještě v případě č.57, naopak v případě č.99 došlo ke snížení tentativních korespondencí z 287 na 286.

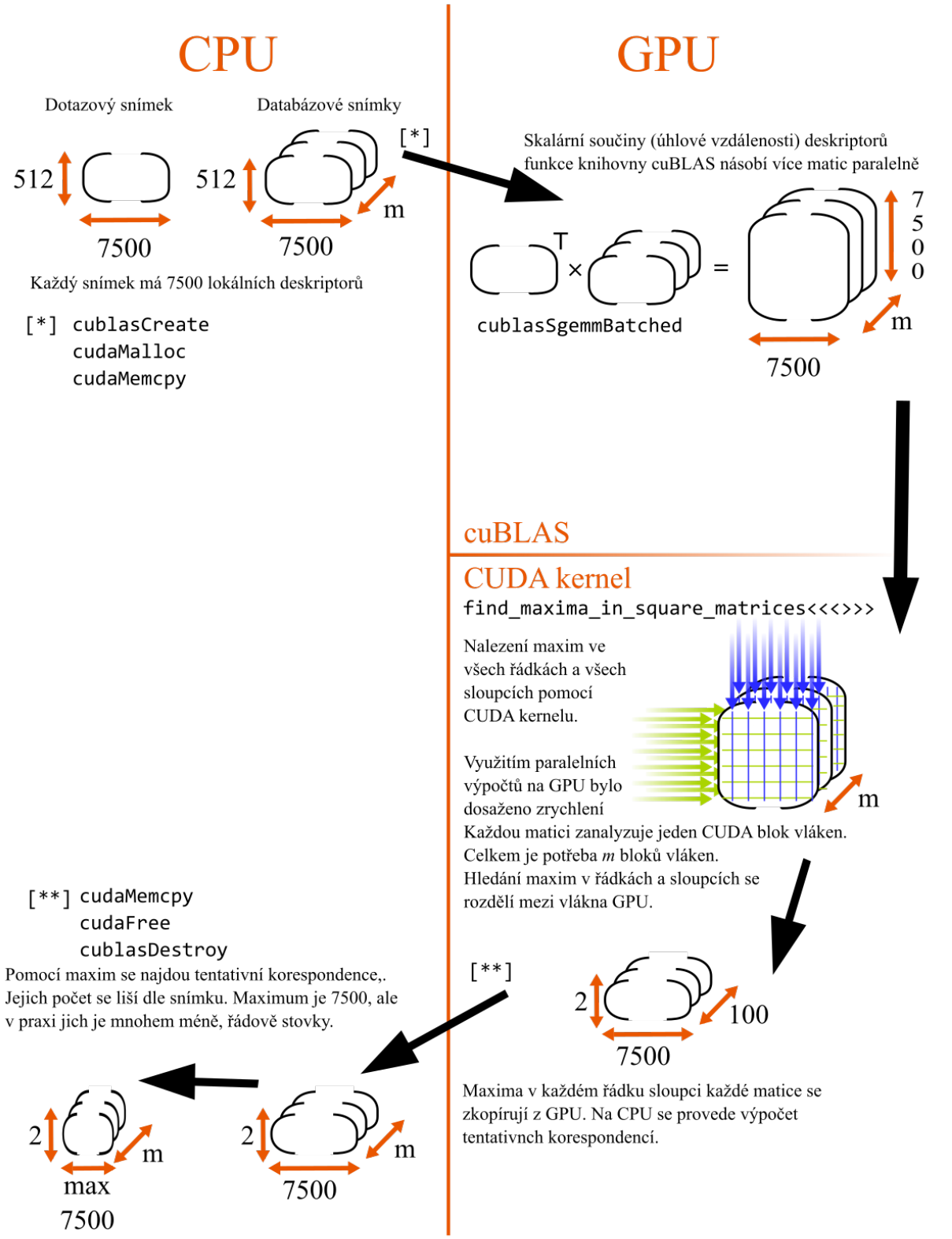
Implementace s knihovnou cuBLAS je nejvíce zatížena zaokrouhlovacími chybami, což ve 3% testovaných případů změnilo počet nalezených tentativních korespondencí o 1. Jedná se o zanedbatelnou odchylku, tato implementace byla zařazena do projektu.

4.5 CUDA

CUDA byla použita v projektu dvěma způsoby, což přineslo významné zrychlení předělaných částí algoritmu, o čemž svědčí grafy 7.1, 7.2, 7.3 a 7.4. Pro maticové operace byla použita knihovna cuBLAS, která využívá technologii CUDA interně. Dále byl naimplementován kernel CUDA, který analyzuje matice podobnosti lokálních deskriptorů (z čehož se poté odvodí tentativní korespondence, viz. schématický obrázek 4.1.

Knihovna **cuBLAS** byla použita pro hromadné vynásobení m matic o rozměru 7500×512, což přineslo významné urychlení. Výsledkem této operace je m matic o rozměru 7500×7500, což je velký objem dat, konkrétně $m \times 7500 \times 7500$ čísel float, to je např. pro $m = 100$ přesně 22,5 GB dat. Jedná se o podobnosti mezi jednotlivými lokálními deskriptory, mezi nimiž se hledají maximální podobnosti, ze kterých se poté odvodí tentativní korespondence.

Bylo by neefektivní kopírovat gigabajty dat z GPU a provést hledání maxim pomocí výpočtu na CPU. Matice maximálních podobností je řádově menší, konkrétně $7500 \times 2 = 15000$ indexů integer, tedy 60 kB pro jeden pár snímků, tedy pro $m = 100$ přesně 6 MB. Nabízí se proto naimplementovat **kernel CUDA**, který tato maxima najde mnohem rychleji než výpočet CPU (jedná se o dobře paralelizovatelnou úlohu do mnoha vláken, která GPU nabízí) a navíc řádově sníží objem kopírovaných dat (přesně 3750×). Schématický obrázek 4.1 zobrazuje využití CUDA a cuBLAS.



Obrázek 4.1: Rozdělení výpočtu tentativních korespondencí mezi CPU, cuBLAS a kernel CUDA

Kapitola 5

Chyby a nevýhody současného InLocu

V rámci experimentálního ověření dosažené rychlosti a přesnosti byly nalezeny zásadní nedostatky algoritmu, které mají negativní dopad na přesnost. Tato kapitola se zabývá nalezenými chybami a rovněž diskutuje jejich nápravu. Nalezené chyby se vyskytují relativně často a mají souvislost s nemožností spolehlivého zvýšení rychlosti běhu InLocu pomocí zvětšení datasetu a snížení m (počet nejpodobnějších snímků, pro které se hledají korespondence s dotazovým snímkem). To bude tématem další kapitoly.

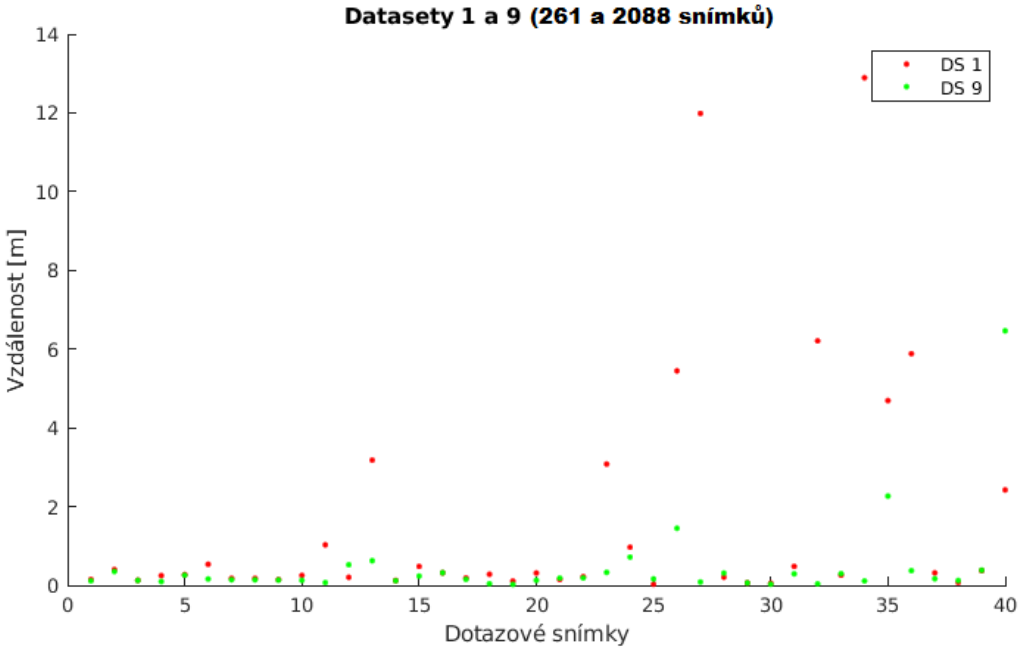
5.1 Chybné odhady s různými datasety

Detailním porovnáním přesnosti algoritmu s různými datasety se zabývá sekce 6.2.5. Tato sekce se zabývá výskytem typických chyb algoritmu.

Vliv velikosti datasetu na přesnost odhadu byl zkoumán na devíti datasetech snímků RGB-D, které mapují místnosti CIIRCu (velikosti: 261, 391, 522, 783, 1044, 1305, 1566, 1827, 2088). Platí, že každý větší dataset je rozšířením menšího datasetu. InLoc se spouštěl s parametry $m = 100$ a $n = 10$. V této sekci se přesnost odhadu ukazuje na eukleidovské vzdálenosti skutečného středu kamery od odhadnutého středu. Zlepšení přesnosti mezi nejmenším a největším datasetem je viditelné na první pohled. Rozdíly mezi po sobě jdoucími datasety jsou ale nenápadné, přesto většinou dojde ke zlepšení přesnosti. To odpovídá závěrům 24/7 image place recognition paper [5]. Dotazy, kde navzdory zvětšení datasetu nedošlo ke zlepšení přesnosti, jsou

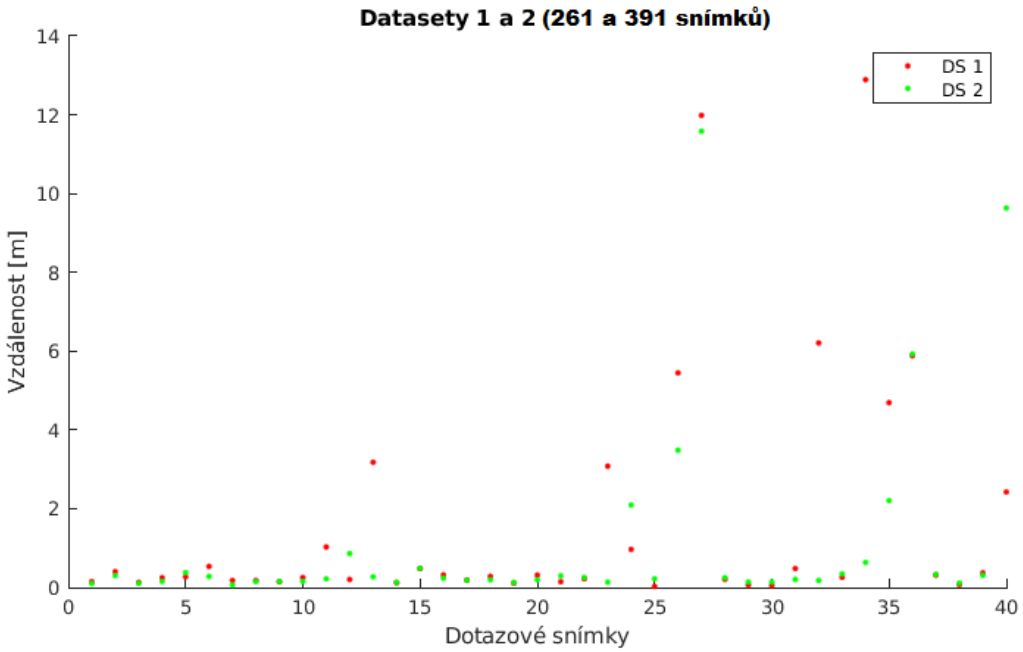
zatížené chybou. V rámci testování bylo nalezeno několik druhů chyb, kterých se InLoc dopouští. Níže jsou popsané objevené chyby a návrhy na jejich odstranění nebo zredukování.

Dohoda: Bylo zmíněno, že při zvětšení datasetu se může přesnost lokalizace zhoršit. Většina zhoršení je však zanedbatelná, řádově v jednotkách centimetrů. Budeme zmiňovat jen významná zhoršení přesnosti. Řekněme, že **významné zhoršení přesnosti** nastává, když se přesnost při zvětšení datasetu zhorší alespoň o 0,2 m.



Obrázek 5.1: Porovnání přesnosti pro první a devátý dataset (261 a 2088 snímků). Chyba pro velký dataset je u většiny dotazů nižší. Velký dataset se také mnohem méně dopouští selhání, při kterých nepřesnost lokalizace dosáhne několika metrů.

Na obrázku 5.1 je zlepšení přesnosti při zvětšení datasetu dobře patrné. Významné zhoršení přesnosti (tedy více než 20cm) nastalo u 5% dotazů. K nejnápadnějšímu zhoršení přesnosti došlo u dotazového snímku 40, který vede ke specifické chybě a je diskutován v další části kapitoly. Níže jsou ukázána postupná zpřesňování na po sobě jdoucích datasetech. Při postupném zvětšování datasetu je lépe vidět, jak se InLoc dopouští chyb.

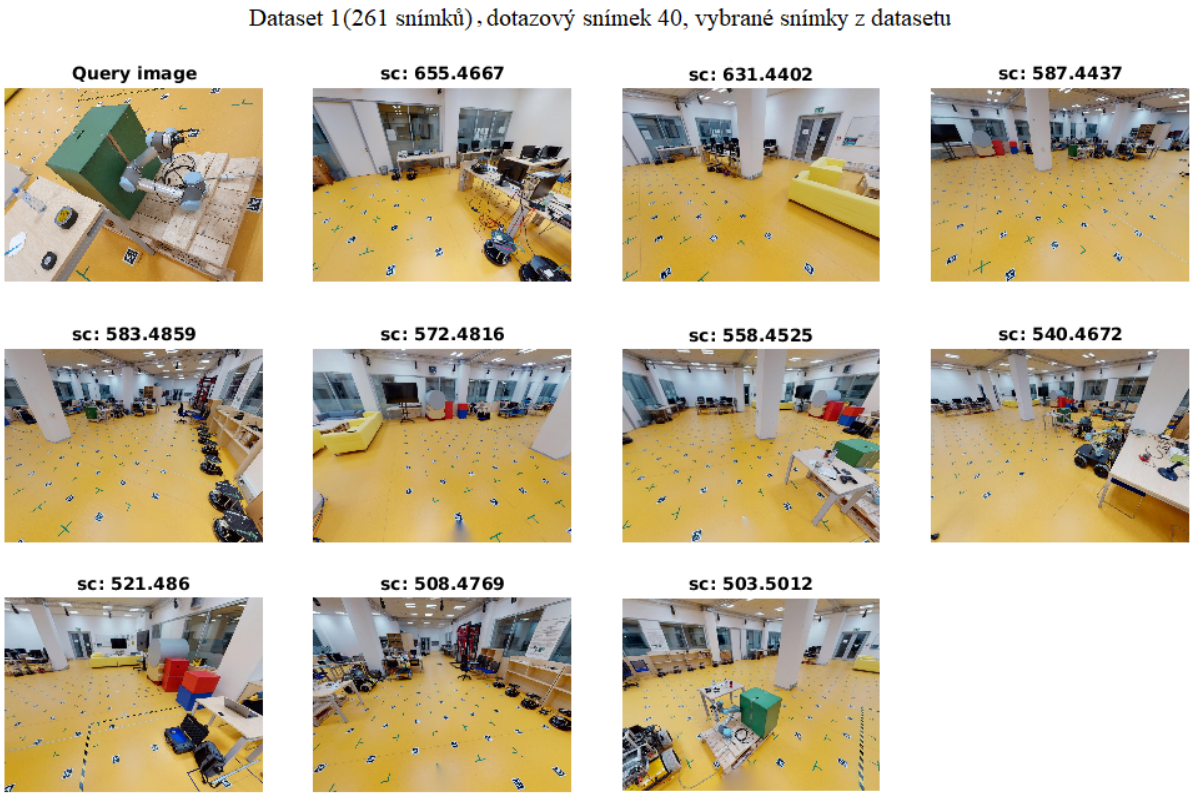


Obrázek 5.2: Nejmenší dataset (261 snímků) se dopustil nejvíce selhání. U druhého datasetu (391 snímků) došlo k nižšímu počtu selhání, nebo selhání dosáhla alespoň o něco nižší nepřesnosti. Významné zhoršení přesnosti nastalo u 7,5% dotazových snímků. Dotaz č. 40 dosáhl násobného zhoršení, proto je diskutován níže.

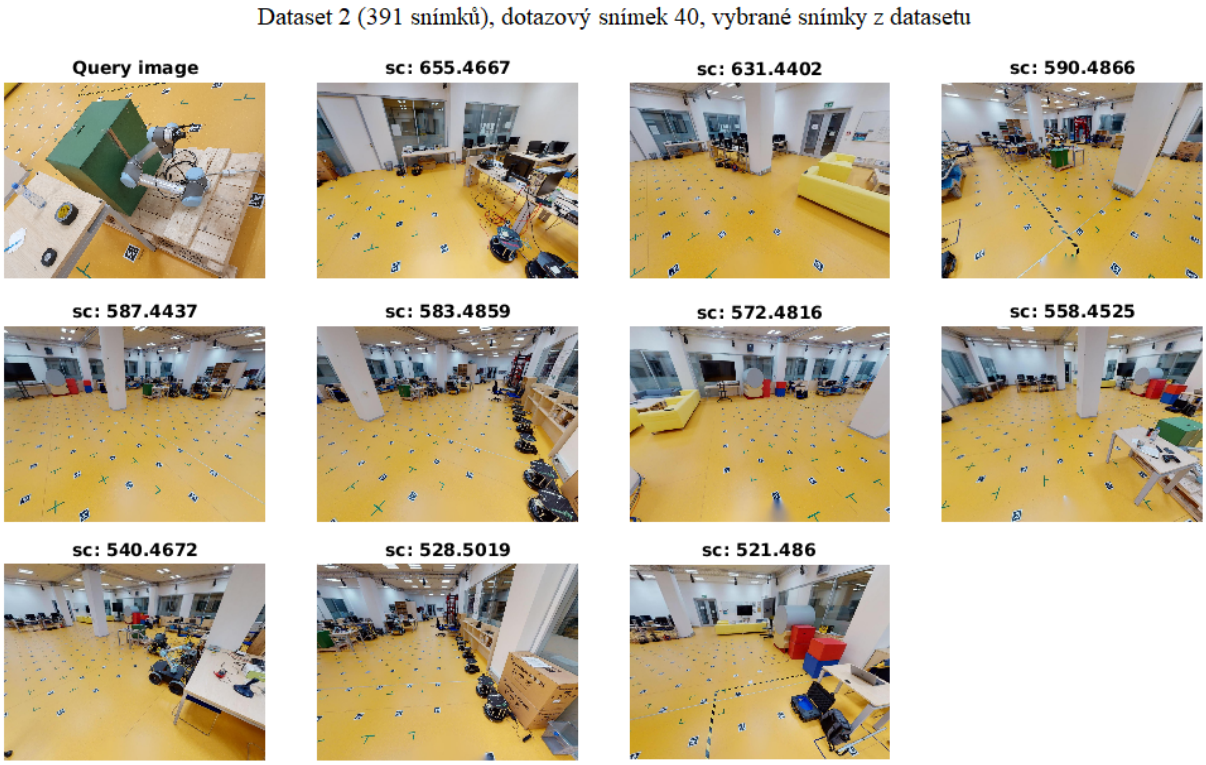
Dotaz 40 nebyl spolehlivě lokalizován pomocí žádného datasetu, proto ho rozeberme detailněji. Pro první dataset byla odhadnuta pozice [3,66; 1,36; 4,91]. Pro druhý dataset [-1,7; 0,33; 12,45]. Ani jeden odhad není přesný, skutečná pozice je [2.19, 1.52, 2.99]. Obrázky 5.3 a 5.4 ukazují výsledek kroku geometrické verifikace, tedy vybrané databázové snímky s nejvyšším počtem inlierů. Některé vybrané databázové snímky jsou skutečně pořízeny z podobného místa jako dotazový snímek (vyobrazují zelené zařízení, které je na dotazovém snímku) a dají se považovat za rozumný výstup. Mnoho vybraných snímků je ale z jiného místa. Už tento krok má potíže s výběrem snímků ze správného místa, protože v databázi patrně chybí snímek daného zeleného zařízení zblízka, nebo ze správného úhlu. V databázi také chybí snímky míst a předmětů z výrazného podhledu či nadhledu. Proto takové dotazové snímky mají v databázi méně podobných snímků, ze kterých by se dala přesně vypočítat póza kamery.

Obrázky 5.5 a 5.6 ukazují další krok, tedy nejlepší syntetické snímky pro tento dotaz. Je vidět, že se algoritmus snaží odhadnout takovou pózu, aby napodobil pohled shora na podlahu, což vidíme na dotazovém snímku, ale bohužel došlo k chybě odhadu pozice (snímky vyobrazují jen podlahu).

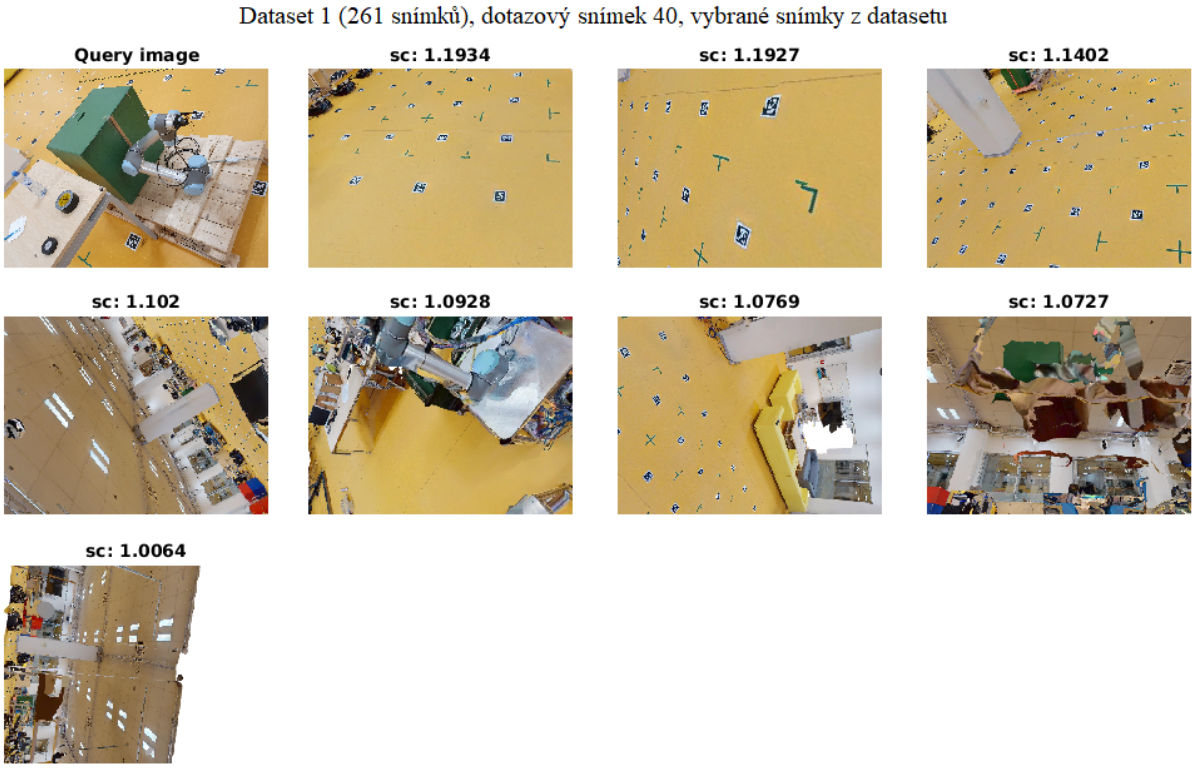
Žádný syntetický snímek nenabídl přesný odhad, což může souviset s problémy předešlého kroku geometrické verifikace (databáze neposkytuje vhodné snímky, ze kterých by se odhadla póza pro tak netypický dotazový snímek). Mezi vybranými syntetickými snímky se vyskytuje další chyba: odhadnutá pozice kamery se nachází mimo zmapovaný prostor, což se projeví rozlehlým bílým pozadím.



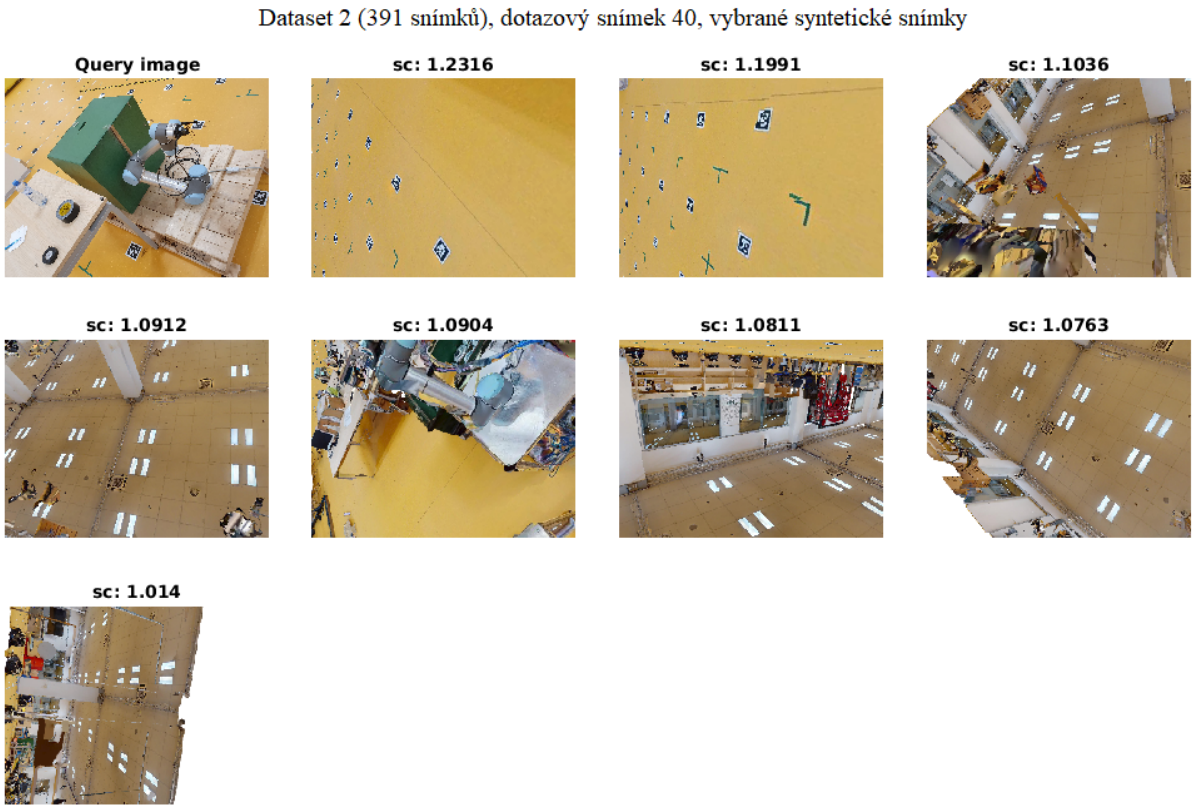
Obrázek 5.3: Výběr nejlepších databázových snímků RGB-D pro dotaz 40 při použití datasetu o velikosti 261 snímků. U každého snímku je uvedeno skóre podobnosti s dotazovým snímkem. Nejvíce se na tvorbě skóre podílí počet inlierů. Výpočet tohoto skóre je popsán v sekci 2.1



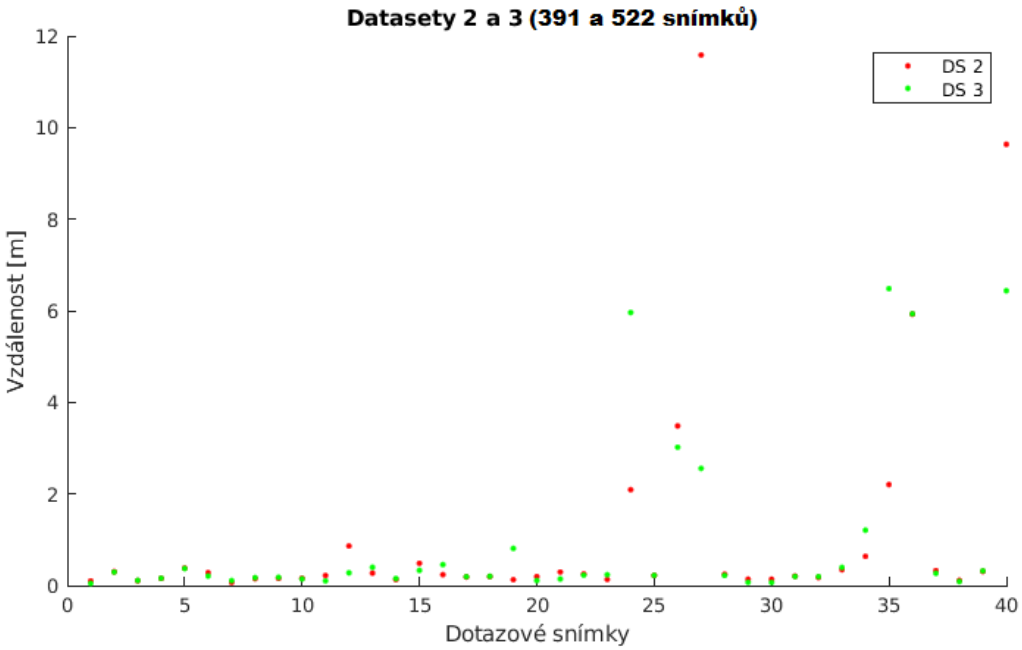
Obrázek 5.4: Výběr nejlepších databázových snímků RGB-D pro dotaz 40 při použití datasetu o velikosti 391 snímků. U každého snímku je uvedeno skóre podobnosti s dotazovým snímkem. Nejvíce se na tvorbě skóre podílí počet inlierů. Výpočet tohoto skóre je popsán v sekci 2.1



Obrázek 5.5: Výběr nejlepších syntetických snímků pro dotaz 40 při použití datasetu o velikosti 261 snímků. U každého syntetického snímku je uvedeno skóre podobnosti s dotazovým snímkem. Výpočet skóre podobnosti syntetického snímku s dotazovým je popsán v rovnici 2.2

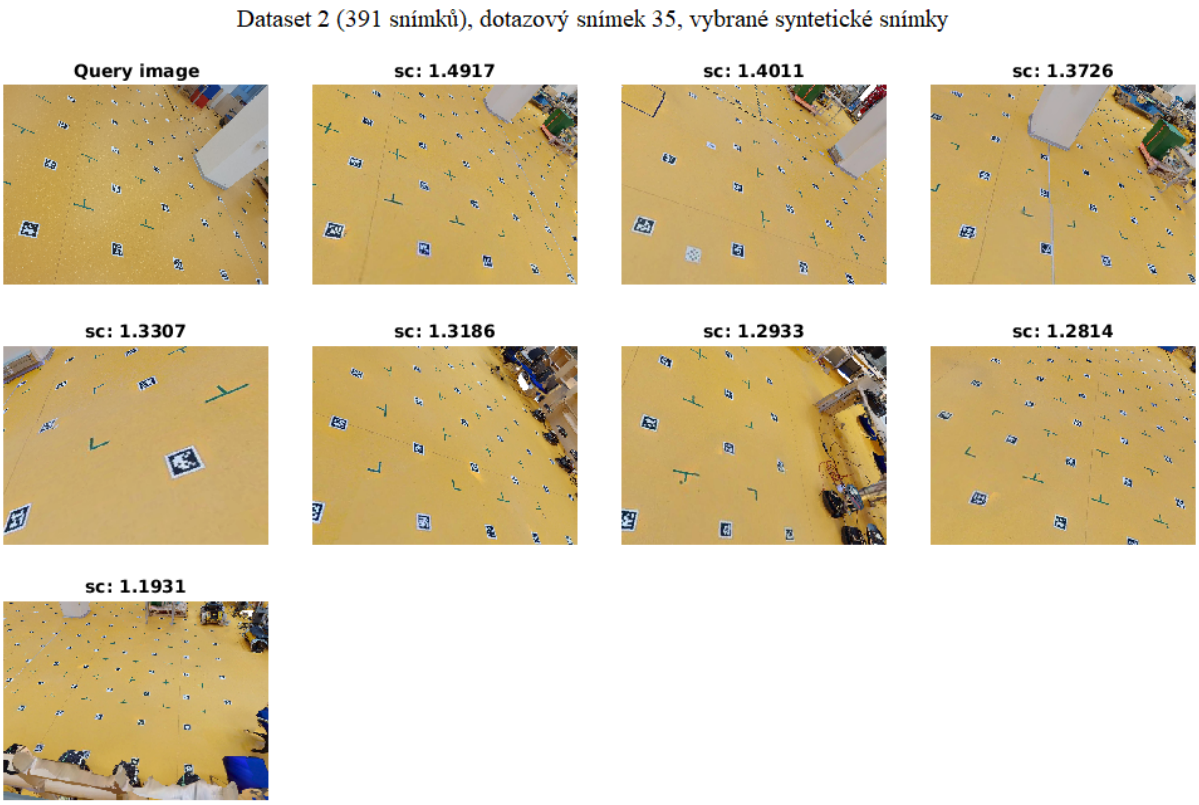


Obrázek 5.6: Výběr nejlepších syntetických snímků pro dotaz 40 při použití datasetu o velikosti 391 snímků. U každého syntetického snímku je uvedeno skóre podobnosti s dotazovým snímkem. Výpočet skóre podobnosti syntetického snímku s dotazovým je popsán v rovnici 2.2

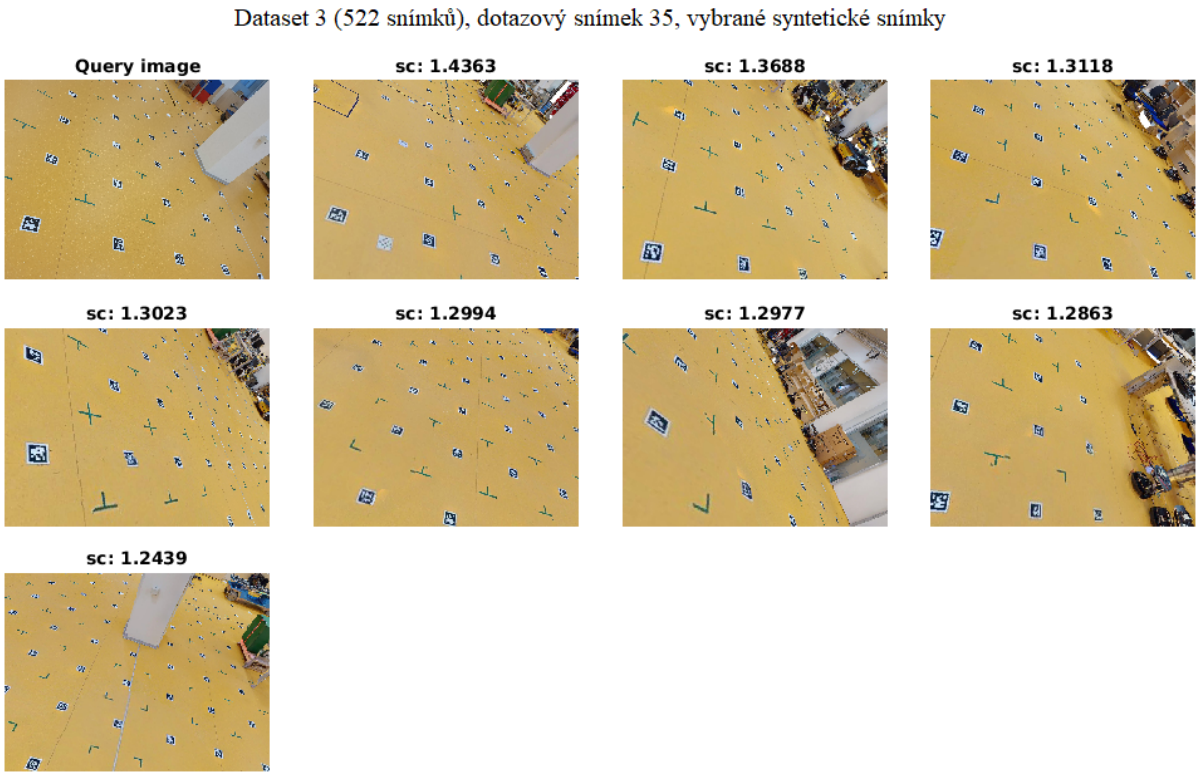


Obrázek 5.7: Třetí dataset (522 snímků) opět snížil počet selhání a některá selhání alespoň výrazně zredukoval. 12,5% dotazových snímků bylo lokalizováno s významným zhoršením přesnosti.

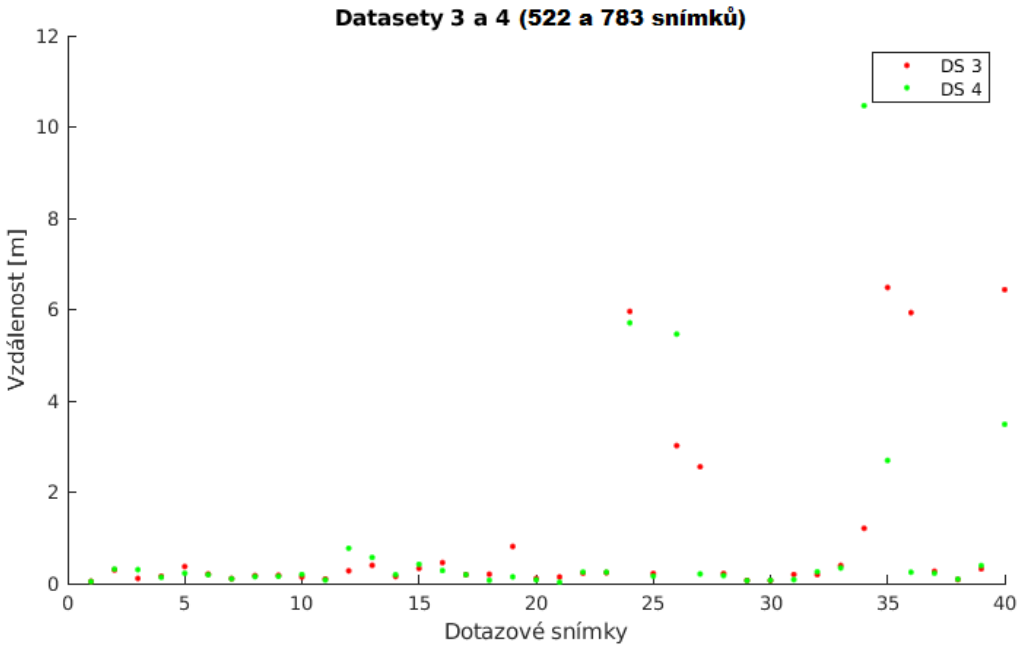
Lokalizace dotazu 35 dopadla pro druhý i třetí dataset špatně. Tento dotaz dopadl špatně i pro všechny ostatní datasety. Nabízí se otázka, v čem je problém. Obrázky 5.8 a 5.9 ukazují, že dotazový snímek je velmi nejednoznačný. Zobrazuje podlahu s bílým sloupem. Problém je, že zmapované prostředí obsahuje více takto vypadajících míst. Selhání lokalizace jsou pro nejednoznačné dotazy přirozená.



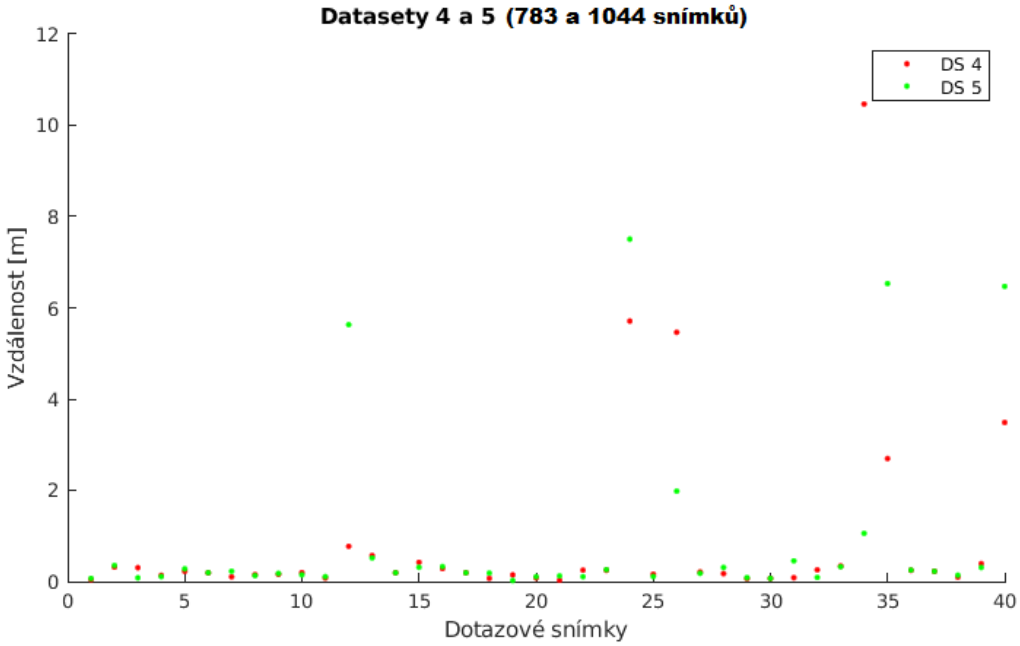
Obrázek 5.8: Dotazový snímek 35 a nejpodobnější syntetické snímky při použití datasetu o velikosti 391 snímků.



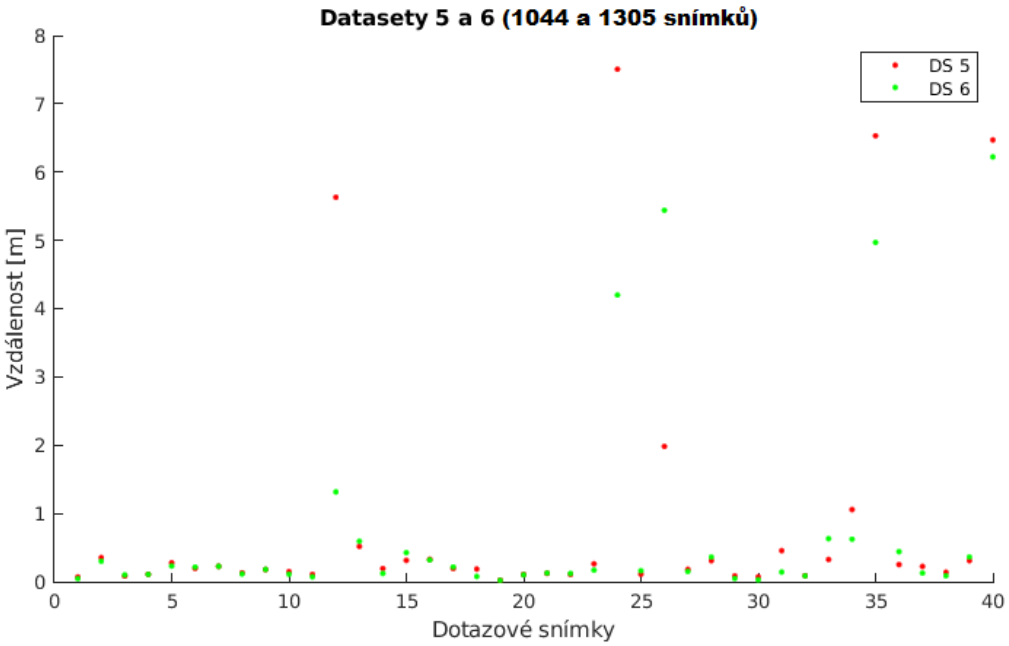
Obrázek 5.9: Dotazový snímek 35 a nejpodobnější syntetické snímky při použití datasetu o velikosti 522 snímků.



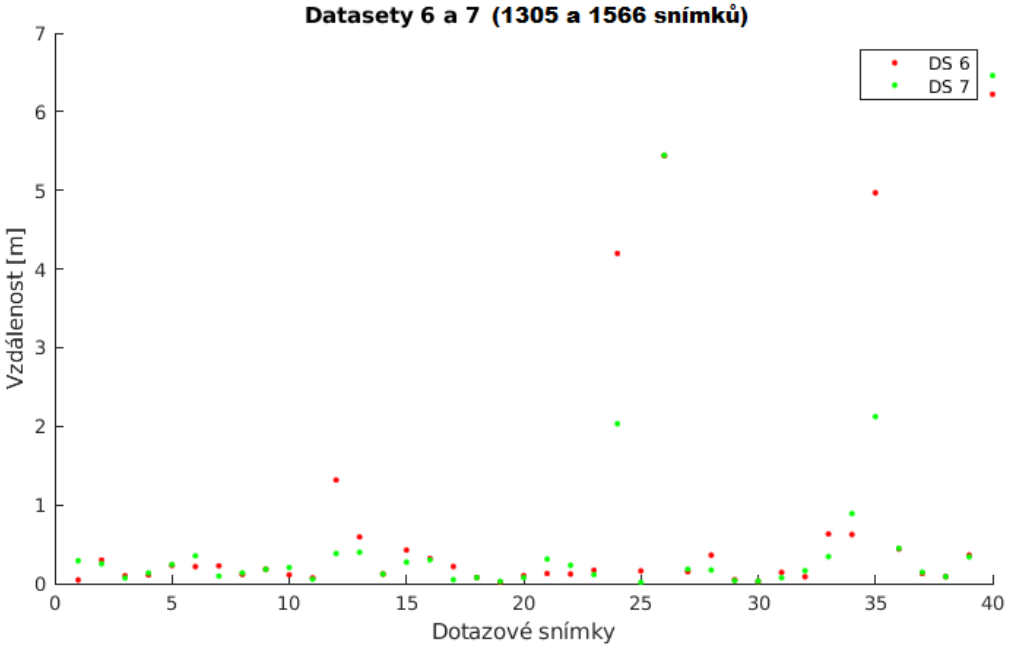
Obrázek 5.10: Čtvrtý dataset (783 snímků). Významné zhoršení přesnosti nastalo u 7,5% dotazových snímků.



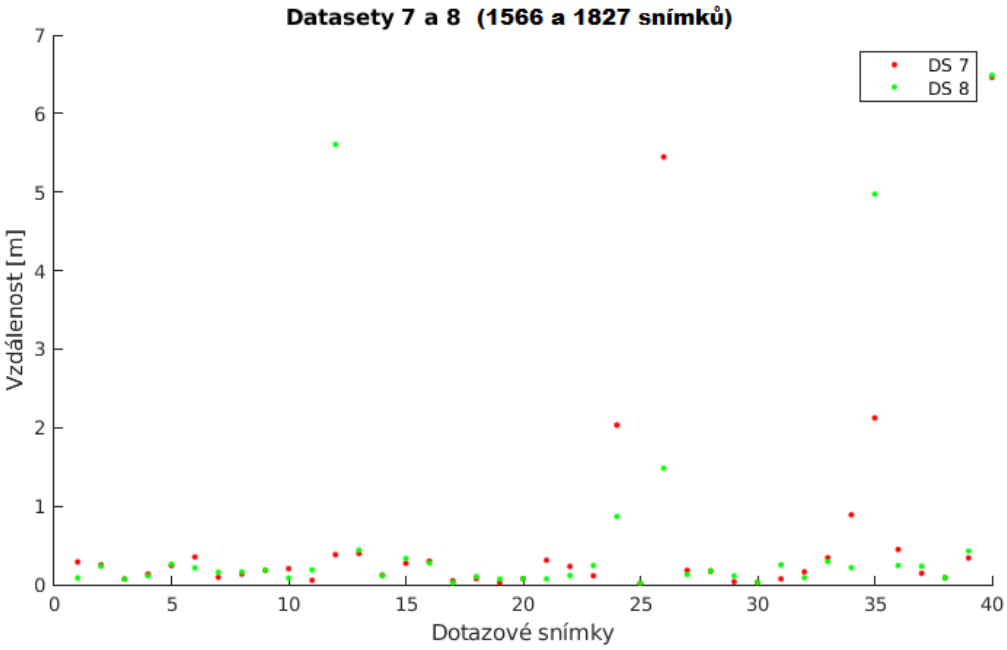
Obrázek 5.11: U pátého datasetu (1044 snímků) se projevilo významné zhoršení chyb u 12,5% dotazových snímků.



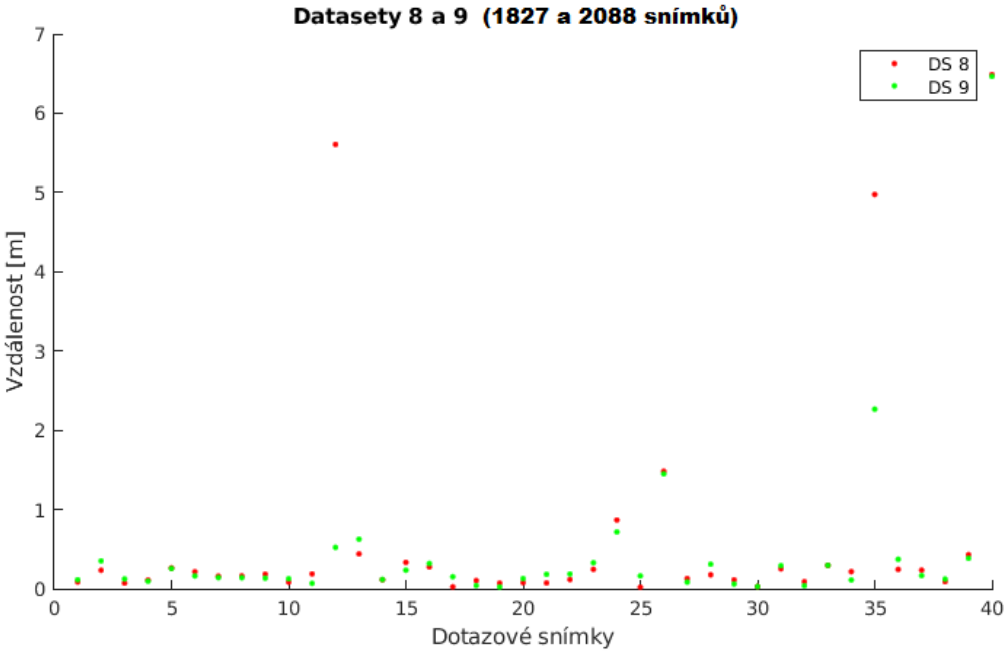
Obrázek 5.12: Šestý dataset (1305 snímků). Významné zhoršení přesnosti nastalo u 5% dotazových snímků.



Obrázek 5.13: Sedmý dataset (1566 snímků). Významné zhoršení přesnosti nastalo u 7,5% dotazových snímků.



Obrázek 5.14: Osmý dataset (1827 snímků). Významné zhoršení přesnosti nastalo u 5% dotazových snímků.



Obrázek 5.15: Devátý dataset (2088 snímků) zlepšil celou řadu dotazů. Významné zhoršení přesnosti nenastalo u žádného dotazového snímku.

Časté chyby, které způsobují výše popsané selhávání lokalizace, jsou detailně popsány v sekci níže.

5.2 Různé chyby

Selhání lokalizace znamená, že odhadnutá póza kamery je buď lokalizována na zcela jiném místě, nebo na správném místě, ale např. se špatnou rotací kamery. Při selháních se dá pozorovat několik speciálních případů chyb.

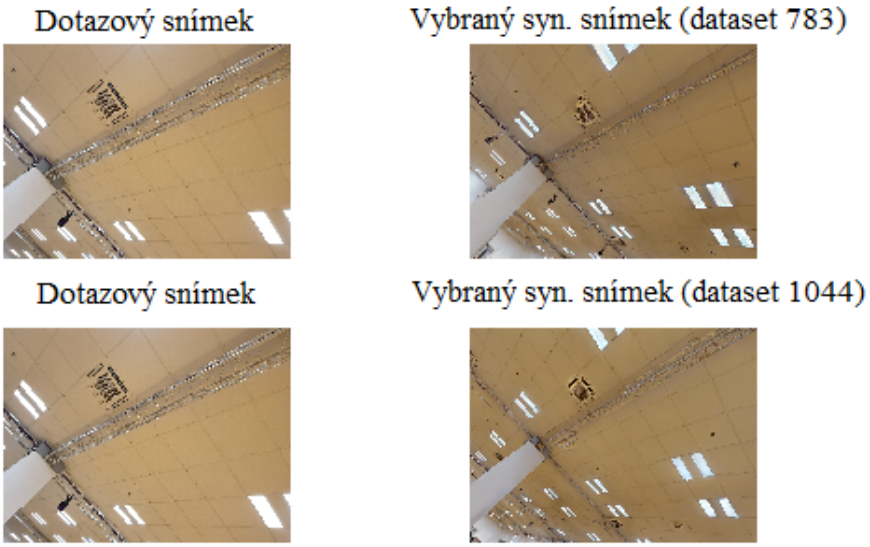
Neexistence databázového snímku z podobné pozice

Aby byl InLoc spolehlivý, musí mít každý snímek, který chceme lokalizovat, korespondující snímek s dostatečně podobným úhlem pohledu a podobnou pozicí kamery v databázi. Obrázky 5.3 a 5.5 ukazují situaci, kdy se pro dotazový snímek s netypickým úhlem záběru nenašel dostatečně podobný databázový snímek, kvůli čemuž lokalizace selhala.

Tento problém se dá řešit doplněním datasetu o databázové snímky s různými úhly záběru (typickými i neobvyklými), tj. zvětšením počtu databázových snímků.

Záměna dvou vizuálně podobných míst

Pokud se ve zmapovaném prostředí vyskytuje více navzájem podobných míst, může se InLoc dopouštět chyb. Výběr nejpodobnějších snímků (databázových i syntetických) může obsahovat podobné snímky, ale velmi odlišné souřadnice a výsledný odhad pózy může být nepoužitelný. Jedná se o přirozenou vlastnost lokalizace ze snímku, proto tuto chybu není možné zcela odstranit, dá se však zredukovat. Níže jsou vyobrazeny příklady lokalizace v prostorách CIIRCu, kde k této chybě došlo.



Obrázek 5.16: Pro stejný dotazový snímek byl v každém běhu algoritmu s jiným datasetem vybrán jiný syntetický snímek s nejnižším mediánem chyby deskriptoru DSIFT.

Obrázek 5.16 ukazuje situaci, kdy pro stejný dotazový snímek byly ve dvou spuštěních InLocu vybrány dva různé syntetické snímky s nejnižší chybou. Při detailním pohledu je vidět, že vybrané syntetické snímky jsou sice velmi podobné, ale z úplně jiného místa. V případě datasetu 783 odhad pózy uspěl, ale pro dataset 1044 úplně selhal (nepřesnost v řádech metrů). Tato konkrétní chyba vznikla z důvodu nejednoznačnosti dotazového snímku. Ve zmapovaném prostředí je více stejných sloupů a stejný strop.

V současné verzi InLocu může dojít k situaci, kdy by měl být dotazový snímek jednoznačný (např. pohled do místnosti s rozmístěným nábytkem), ale lokalizace selže z důvodu upřednostnění syntetického snímku z jiného místa, kde je rozmístění předmětů sice podobné, ale na první pohled se jedná o jiný nábytek. Současná metoda obodování podobnosti syntetického snímku s dotazovým (výpočet popisuje rovnice 2.2) může přehlížet detaily, protože rozhodující je pro ni pouze medián chyb. Obrázek 5.17 jednu takovou situaci ukazuje.

Obrázek 5.17: Stejný dotazový snímek, ale zcela odlišné syntetické snímky ze dvou různých míst. Pro dataset o velikosti 1827 se lokalizace zdařila, pro dataset o velikosti 2088 selhala. Vybraný syntetický snímek dosáhl nižšího mediánu chyb deskriptoru DSIFT, a proto byl vybrán.

U této situace se nabízí otázka, zda selhání lokalizace má na svědomí jen špatný výběr syntetického snímku kvůli špatnému změření podobnosti s dotazovým snímkem, nebo selhal už některý z předchozích kroků. Obrázek 5.18 ukazuje výsledek předchozího kroku, tedy geometrické verifikace (počítání inlierů mezi lokálními deskriptory algoritmem RANSAC). Výsledky geometrické verifikace byly pro datasety o velikostech 1827 a 2088 velmi podobné. Seznamy vybraných RGB-D snímků se sice lišily, ale prvních několik snímků bylo ve stejném pořadí a se stejným skóre. Krok geometrické verifikace vrátil pro oba datasety korektní výsledky. Chyba musela nastat až později – ve verifikaci syntetických snímků.

Obrázek 5.18: Krok geometrické verifikace dopadl úspěšně. Pro datasety 1827 a 2088 byly vybrány téměř stejné databázové snímky RGB-D. Prvních několik snímků bylo v identickém pořadí se stejným skóre (obrázek ukazuje první tři). RGB-D snímky s nejvíce inliery pochází ze stejného místa jako dotazový snímek

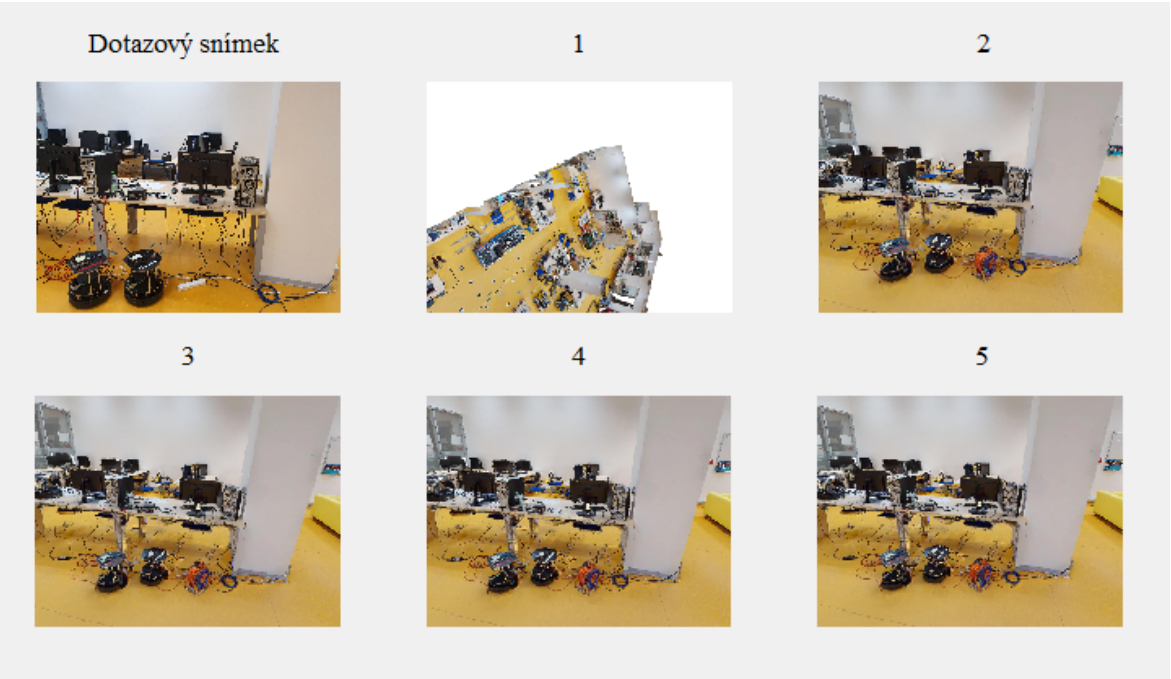
Záměna podobných míst při lokalizaci se dá minimalizovat několika způsoby:

- Zvýšit rozlišení syntetických snímků. Současná verze InLocu používá malé rozlišení 378x504. Proto popis snímku pomocí DSIFT přehlíží některé detaily.
- Nepoužívat nejednoznačné dotazy (např. zmíněný snímek světla na stropě)
- Použít jinou metodu než medián chyb deskriptorů DSIFT. Měření podobnosti dotazového snímku se syntetickým pomocí deskriptorů DSIFT funguje spolehlivě jen tehdy, když je odhadnutá póza velice přesná. I malá změna rotace či souřadnice středu kamery může zásadně zvýšit medián chyb, což může způsobit zavrhnutí syntetického snímku ve prospěch jiného, který byl ale pořízen úplně jinde (takovou situaci ukazuje obrázek 5.17).

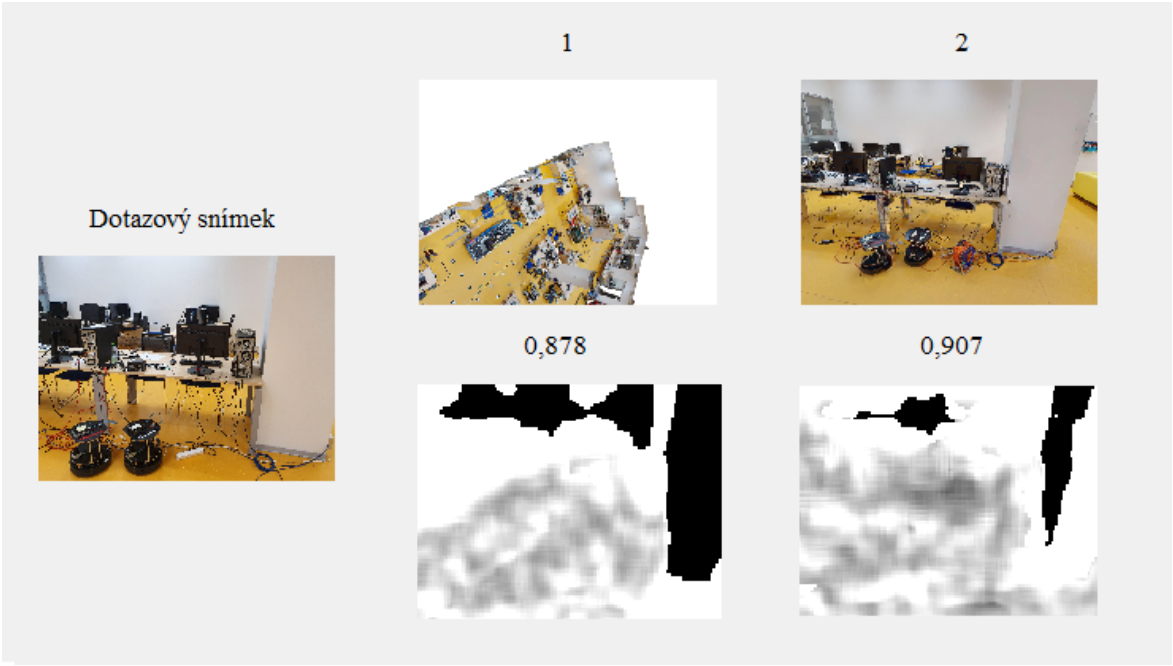
Odhadnutá póza kamery mimo zmapovaný prostor

Občas se vyskytují případy, kdy se v kroku PE (pose estimation) odhadne taková póza kamery, jejíž střed má souřadnice mimo zmapovaný prostor. Syntetický snímek vzniklý z této chybné pózy mívá obvykle bílé pozadí. Ve verifikačním kroku se takovým syntetickým snímkům obvykle přiřadí nízké skóre, ale obrázek 5.19 ukazuje výjimku, kdy byl tento chybný snímek vybrán jako nejpřesnější odhad. Odhadnutá pozice kamery byla v tomto případě X=-1.1368, Y=7.9556, Z=-4.5940. Chybný syntetický snímek je pohled na

zmapované prostředí skoro z osmimetrové výšky. Tento snímek byl vybrán jako nejlepší, protože tvar bílého pozadí je podobný tvaru bílých stěn na dotazovém snímku. Pro tyto rozsáhlé oblasti ve verifikačním kroku byla vypočtena téměř nulová chyba deskriptoru DSIFT a celkový medián chyby byl ze všech snímků nejnižší. Právě podle mediánu chyb se verifikuje podobnost syntetických snímků s dotazovým. Obrázek 5.20 ukazuje mapu chyb (čím tmavší barva, tím menší chyba)



Obrázek 5.19: Dotazový snímek a výběr pěti syntetických snímků s nejvyšším skóre podobnosti.



Obrázek 5.20: Dotazový snímek, mediány chyb a mapy chyb pro dva nejlépe hodnocené syntetické snímky. Medián chyb pro chybný syntetický snímek je 0,878, zatímco pro správný syntetický snímek 0,907.

Tato chyba by šla vyřešit několika způsoby:

1. Omezit souřadnice kamery dle velikosti zmapovaného prostředí. Pokud by bylo známo, které souřadnice jsou mimo prostředí, bylo by možné všechny odhadnuté kamery s takovými souřadnicemi vyloučit.
2. Na chybném snímku není vidět bílý strop místnosti, protože 3D model prostoru nepředpokládá renderování stropu shora. Kdyby byla textura stropu oboustranná, syntetický snímek by byl celý bílý a nemohl by být vybrán verifikací.
3. Místo bílého pozadí použít např. barevný šum, který se ve zmapovaném prostředí nevyskytuje.

Kapitola 6

Zrychlení algoritmu zvětšením datasetu a snížením m

V této kapitole je popsán experiment, který má ukázat, že zvětšením množiny databázových snímků stoupá přesnost odhadu pózy kamery, a proto s větší databází můžeme snížit m (počet vybíraných nejpodobnějších snímků k dotazovému snímku) a přitom zachovat přesnost výsledku a navíc významně zrychlit běh algoritmu, protože se sníží počet snímků, se kterými se provádí náročné výpočty (např. vyhledávání tentativních korespondencí mezi lokálními deskriptory). Cílem této práce není zvýšení rychlosti běhu triviálním snížením nároků na přesnost výsledku, proto je zachování přesnosti důležité.

Zvětšení databáze má pozitivní vliv na přesnost algoritmu, protože budou k dispozici RGB-D snímky z více pozic. Roste pravděpodobnost, že mezi m vybranými databázovými snímky s nejvyšším skóre podobnosti budou snímky pořízené z místa pořízení dotazového snímku a výsledný odhad bude moct dosáhnout vyšší přesnosti.

6.1 Měření přesnosti odhadu pózy

Odhad pózy kamery v praxi nemůže být úplně přesný. Můžou za to zaokrouhlovací chyby, přirozené zkreslení objektivu, omezené rozlišení i nepřesnost měření. V této kapitole se přesnost odhadnuté pózy měří dvěma čísly:

- eukleidovská vzdálenost skutečného středu kamery od odhadnutého středu kamery
- rotační vzdálenost mezi skutečnou rotací kamery a odhadnutou rotací kamery

Rotační vzdálenost mezi rotačními maticemi R_1 a R_2 je nezáporný úhel mezi R_1v a R_2v, kde v je libovolný nenulový vektor. Výsledná vzdálenost je vždy kladné číslo z intervalu <0; 180>, tedy rotace větší než 180° se převedou na doplněk do 360° (např. 270° se převede na 90°) Rotační vzdálenost se měří tímto algoritmem:

```
R = R2*R1';
rot_distance = acos(0.5 * (trace(R)-1));
rot_distance = abs(rad2deg(rot_distance));
```

Pro každý dataset byl algoritmus spuštěn se čtyřiceti dotazovými snímky (tzn. 40 běhů algoritmu). Pro každý tento běh byl zjištěn průměr chyb (středu kamery i rotace), medián, minimum a maximum.

6.2 Postupné snižování m

Připomeňme, že m je počet nejpodobnějších snímků, které vrací první krok algorítmu - ht_retrieval. Pro tento počet snímků se v kroku ht_top100_densePE_localization provede vyhledání korespondencí s dotazovým snímkem a geometrická verifikace. InLoc poběží pro 9 datasetů různých velikostí pro každé vybrané m. Zpočátku nastavme původní hodnotu $m = 100$. Postupně se bude snižovat na 80, 60, 40 a 20.

U každého experimentu jsou k dispozici tabulky s detailními výsledky. Tabulky 6.11 a 6.12 shrnují mediány chyb všech experimentů pro všechny použité hodnoty m.

Způsob měření výkonu je popsán v sekci 7.2. Uvedené časy běhu jsou aritmetickým průměrem všech čtyřiceti běhů skriptu ht_top100_densePE_localization. Doba běhu tohoto skriptu právě na volbě hodnoty m závisí.

6.2.1 Experiment - vliv velikosti datasetu na přesnost (m=100)

V tomto experimentu byl spuštěn algoritmus InLoc celkem 9x pro různě velké datasety. Pro každý dataset bylo použito 40 dotazových snímků. Experiment ukázal, že kvůli chybám, kterých se InLoc dopouští, není vždy navýšení velikosti datasetu zárukou vyšší přesnosti.

velikost databáze	průměr [m]	medián [m]	minimum [m]	maximum [m]
261	1,61	0,27	0,02	12,89
391	1,08	0,22	0,06	11,58
522	0,97	0,21	0,04	6,48
783	0,88	0,20	0,02	10,46
1044	0,89	0,19	0,02	7,50
1305	0,73	0,16	0,01	6,22
1566	0,58	0,18	0,01	6,46
1827	0,63	0,17	0,01	6,48
2088	0,44	0,16	0,02	6,46

Tabulka 6.1: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje vzdálenost středu odhadnuté kamery od skutečného středu kamery. Mezi nejmenším a největším datasetem skutečně došlo ke zlepšení, ale při postupném zvětšování datasetu se občas přesnost zhoršila. Mohou za to náhodné chyby popsané v předešlé kapitole.

velikost databáze	průměr [°]	medián [°]	minimum [°]	maximum [°]
261	18,71	2,74	0,53	178,17
391	10,65	2,40	0,37	171,60
522	10,28	2,17	0,36	103,54
783	18,51	2,23	0,24	177,74
1044	13,83	2,24	0,47	177,32
1305	15,47	2,60	0,26	178,11
1566	9,47	1,98	0,32	177,98
1827	11,21	2,19	0,38	178,45
2088	4,57	1,86	0,29	89,30

Tabulka 6.2: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje rotační vzdálenost skutečné rotace kamery a odhadnuté rotace kamery.

Je vidět, že mezi nejmenším a největším datasetem skutečně ke zlepšení přesnosti došlo, ale vývoj přesnosti s rostoucím datasetem je spíše náhodný. Může za to výskyt výše popsaných chyb. Toto chování je viditelné i pro výběr jiné hodnoty m.

Výběr hodnoty m má vliv na dobu běhu skriptu ht_top100_densePE_localization. Průměrná doba běhu této části algoritmu pro zvolených 9 datasetů a pro $m = 100$ je 99,2 sekund. Budeme sledovat zkracování této doby běhu při postupném snižování hodnoty m.

6.2.2 Experiment - vliv velikosti datasetu na přesnost (m=80)

Tento experiment je stejný, jen pro něj platí $m = 80$. Nižší počet vybíraných podobných snímků má zvýšit rychlost, ale potenciálně snížit přesnost.

velikost databáze	průměr [m]	medián [m]	minimum [m]	maximum [m]
261	2,05	0,28	0,02	11,71
391	1,02	0,21	0,06	11,70
522	1,30	0,22	0,03	15,09
783	1,20	0,20	0,02	10,48
1044	0,86	0,18	0,01	7,41
1305	0,80	0,17	0,01	10,09
1566	0,71	0,18	0,02	7,33
1827	0,45	0,17	0,01	6,43
2088	0,51	0,17	0,01	6,44

Tabulka 6.3: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje vzdálenost středu odhadnuté kamery od skutečného středu kamery.

velikost databáze	průměr [°]	medián [°]	minimum [°]	maximum [°]
261	27,62	2,34	0,73	177,64
391	12,95	2,18	0,76	178,11
522	17,92	2,09	0,85	172,25
783	18,48	2,97	0,27	177,42
1044	15,91	2,39	0,61	178,11
1305	11,5	2,07	0,32	179,16
1566	10,91	2,08	0,25	167,21
1827	6,70	1,96	0,21	90,47
2088	4,66	1,79	0,14	88,84

Tabulka 6.4: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje rotační vzdálenost skutečné rotace kamery a odhadnuté rotace kamery.

Je vidět, že mezi nejmenším a největším datasetem skutečně ke zlepšení přesnosti došlo, ale vývoj přesnosti s rostoucím datasetem je spíše ná-

hodný. Může za to výskyt výše popsaných chyb. Průměrná doba běhu skriptu ht_top100_densePE_localization pro $m = 80$ je 79,2 sekund.

6.2.3 Experiment - vliv velikosti datasetu na přesnost (m=60)

Tento experiment je stejný, jen pro něj platí $m = 60$. Nižší počet vybíraných podobných snímků má zvýšit rychlost, ale potenciálně snížit přesnost.

velikost databáze	průměr [m]	medián [m]	minimum [m]	maximum [m]
261	2,08	0,30	0,02	14,08
391	0,92	0,20	0,05	6,45
522	1,08	0,21	0,02	11,73
783	1,14	0,20	0,03	10,45
1044	1,22	0,16	0,02	10,58
1305	0,89	0,19	0,02	9,58
1566	0,81	0,17	0,02	10,48
1827	0,50	0,18	0,04	6,45
2088	0,57	0,16	0,02	7,85

Tabulka 6.5: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje vzdálenost středu odhadnuté kamery od skutečného středu kamery.

velikost databáze	průměr [°]	medián [°]	minimum [°]	maximum [°]
261	28,16	2,84	0,91	175,85
391	12,03	2,44	0,93	178,14
522	17,82	2,35	0,67	179,25
783	20,19	2,49	0,28	178,41
1044	22,29	1,85	0,25	179,95
1305	15,69	2,12	0,21	177,59
1566	11,43	2,09	0,38	174,46
1827	6,97	2,08	0,62	92,29
2088	6,25	1,83	0,32	92,41

Tabulka 6.6: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje rotační vzdálenost skutečné rotace kamery a odhadnuté rotace kamery.

Průměrná doba běhu skriptu ht_top100_densePE_localization pro $m = 60$ je 62,5 sekund.

6.2.4 Experiment - vliv velikosti datasetu na přesnost (m=40)

Tento experiment je stejný, jen pro něj platí $m = 40$. Nižší počet vybíraných podobných snímků má zvýšit rychlost, ale potenciálně snížit přesnost.

velikost databáze	průměr [m]	medián [m]	minimum [m]	maximum [m]
261	1,84	0,33	0,04	11,79
391	1,67	0,26	0,01	10,74
522	1,63	0,18	0,03	15,29
783	1,60	0,21	0,01	15,78
1044	1,36	0,17	0,04	10,49
1305	0,88	0,18	0,02	9,87
1566	0,85	0,18	0,01	9,52
1827	0,76	0,19	0,06	7,81
2088	0,72	0,18	0,02	9,75

Tabulka 6.7: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje vzdálenost středu odhadnuté kamery od skutečného středu kamery.

velikost databáze	průměr [°]	medián [°]	minimum [°]	maximum [°]
261	25,07	2,85	0,54	177,65
391	23,89	2,36	0,45	177,16
522	20,28	2,11	0,59	175,67
783	23,99	2,66	0,32	179,85
1044	19,43	2,46	0,24	176,82
1305	12,29	1,97	0,34	176,83
1566	15,96	2,08	0,15	179,02
1827	11,18	2,00	0,20	172,72
2088	10,78	1,83	0,67	170,42

Tabulka 6.8: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje rotační vzdálenost skutečné rotace kamery a odhadnuté rotace kamery.

Je vidět, že mezi nejmenším a největším datasetem skutečně ke zlepšení přesnosti došlo, ale vývoj přesnosti s rostoucím datasetem je spíše náhodný. Může za to výskyt výše popsaných chyb. Průměrná doba běhu skriptu ht_top100_densePE_localization pro $m = 40$ je 50,2 sekund.

6.2.5 Experiment - vliv velikosti datasetu na přesnost (m=20)

Tento experiment je stejný, jen pro něj platí $m = 20$. Nižší počet vybíraných podobných snímků má zvýšit rychlost, ale potenciálně snížit přesnost.

velikost databáze	průměr [m]	medián [m]	minimum [m]	maximum [m]
261	5,12	4,48	0,93	19,11
391	5,32	3,77	0,94	21,26
522	3,69	3,47	1,25	9,71
783	4,92	3,88	1,44	16,97
1044	1,40	0,17	0,01	13,07
1305	0,95	0,22	0,02	7,79
1566	0,82	0,18	0,02	8,22
1827	0,72	0,19	0,01	6,82
2088	0,90	0,18	0,02	7,79

Tabulka 6.9: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje vzdálenost středu odhadnuté kamery od skutečného středu kamery.

velikost databáze	průměr [°]	medián [°]	minimum [°]	maximum [°]
261	61,90	39,60	4,78	178,66
391	23,45	52,14	3,77	178,71
522	46,47	20,74	2,58	179,98
783	68,35	58,85	5,48	179,94
1044	20,61	1,98	0,24	177,56
1305	12,68	2,65	0,27	172,83
1566	11,80	2,06	0,14	175,10
1827	8,72	2,12	0,70	86,41
2088	13,13	2,49	0,17	177,35

Tabulka 6.10: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje rotační vzdálenost skutečné rotace kamery a odhadnuté rotace kamery.

Pokud zvolíme $m = 20$, objevují se vyšší nepřesnosti v odhadu. To je viditelné u nejmenších datasetů, zatímco největší datasety poskytují stále velmi přesné odhady. U takto nízkých hodnot m je velikost a kvalita datasetu ještě důležitější než u vysokých hodnot m. Pro $m = 20$ je medián odchylek odhadnutých středů kamery u největšího datasetu téměř $25\times$ nižší než u nejmenšího datasetu. U vyšších hodnot m nebyly tyto rozdíly tak propastné. Průměrná doba běhu skriptu ht_top100_densePE_localization pro $m = 20$ je 42,1 sekund.

6.3 Vyhodnocení

Následující dvě tabulky shrnují výsledky předchozích experimentů pro různé hodnoty m. Uvádějí medián chyb při použití různých datasetů a různých hodnot m.

Mediány chyb pro různé datasety	m=100 [m]	m=80 [m]	m=60 [m]	m=40 [m]	m=20 [m]
261	0,27	0,28	0,30	0,33	4,48
391	0,22	0,21	0,20	0,26	3,77
522	0,21	0,22	0,21	0,18	3,47
783	0,20	0,20	0,20	0,21	3,88
1044	0,19	0,18	0,16	0,17	0,17
1305	0,16	0,17	0,19	0,18	0,22
1566	0,18	0,18	0,17	0,18	0,18
1827	0,17	0,17	0,18	0,19	0,19
2088	0,16	0,17	0,16	0,18	0,18

Tabulka 6.11: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje vzdálenost skutečného středu kamery od odhadnutého v metrech.

Mediány chyb pro různé datasety	m=100 [°]	m=80 [°]	m=60 [°]	m=40 [°]	m=20 [°]
261	2,74	2,34	2,84	2,85	39,60
391	2,40	2,18	2,44	2,36	52,14
522	2,17	2,09	2,35	2,11	20,74
783	2,23	2,97	2,49	2,66	58,85
1044	2,24	2,39	1,85	2,46	1,98
1305	2,60	2,07	2,12	1,97	2,65
1566	1,98	2,08	2,09	2,08	2,06
1827	2,19	1,96	2,08	2,00	2,12
2088	1,86	1,79	1,83	1,83	2,49

Tabulka 6.12: Vliv velikosti datasetu na přesnost odhadu pózy. Tabulka ukazuje rotační vzdálenost skutečné rotace kamery a odhadnuté rotace kamery.

Není překvapivé, že výběr nižšího počtu snímků urychlí běh algoritmu. Hledání tentativních korespondencí patří mezi výpočetně náročné operace. Průměrné doby běhu skriptu ht_top100_densePE_localization pro vybrané hodnoty m jsou: **99,2; 79,2; 62,5; 50,2 a 42,1 sekund**. Průměrují se doby běhů algoritmu pro 40 různých dotazových snímků.

Výběr hodnoty m má sice na přesnost a na počet selhání určitý vliv, jak naznačují tabulky výše, ale mnohem důležitější je velikost datasetu. Průměrné chyby rotační vzdálenosti a vzdálenosti odhadnutého středu kamery od skutečného jsou často mezi nejmenším a největším datasetem několikanásobné (rozdíly v mediánech chyb u malých a velkých datasetů nejsou ale tak propastné). Např. největší dataset poskytoval pro $m = 40$ přesnější odhady než nejmenší dataset pro $m = 100$, samozřejmě také v kratším čase. Větší datasety než 2088 snímků nebyly v době provádění těchto experimentů k dispozici, nicméně lze tušit další zlepšení přesnosti.

Experimenty ukázaly, že nejpodstatnější je kvalitní a rozsáhlý dataset. **Je možné snížit čas běhu algoritmu snížením hodnoty** m. Vzniklé zhoršení přesnosti lze poměrně snadno kompenzovat rozšířením (a zkvalitněním) datasetu. Samozřejmě pro každý dataset platí, že vyšší hodnota m zvýší přesnost, ale vždy to značně prodlouží čas běhu algoritmu. Pro každou reálnou aplikaci algoritmu InLoc lze výběrem hodnoty m stanovit buďto důraz na přesnost, nebo důraz na výkon.

Kapitola 7

Experimenty

7.1 Prostředí běhu

Všechny experimenty byly spouštěny na clusteru CIIRC. Jedná se o sestavu sedmnácti výpočetních nodů s různou hardwarovou výbavou. Paměť RAM jednotlivých nodů dosahuje 512GB. Některé nody jsou vybaveny osmi GPU (NVIDIA Tesla V100 SXM2 32 GB). Softwarová výbava je poskytována nástrojem ml a výpočetní výkon se rezervuje nástrojem Slurm.

7.1.1 Výhody práce na clusteru CIIRC

1. Na clusteru je možno pomocí programu Slurm rezervovat přesně definovaný výpočetní výkon (velikost RAM, počet jader CPU, počet GPU aj.), což umožňuje spouštět experimenty v různých podmínkách.

2. Pomocí systému ml je poskytováno velké množství modulů, aplikací a knihoven v různých verzích. Cluster poskytuje několik stovek modulů, mimo jiné matematické knihovny (např. Eigen), kompilátory (např. GCC), nástroje pro CUDA, Matlab a další.

3. Do clusteru je zapojeno velké množství výkonných strojů, jejichž výpočetní výkon se dá rezervovat. Proto je možné spouštět paralelně více úloh, nebo práce mnoha uživatelů najednou.

7.1.2 Nevýhody práce na clusteru

1. Výpočetní nody neimplementují X forwarding, proto na nich lze pracovat s příkazovým řádkem, ale ne s aplikacemi využívajícími GUI. Práci s GUI podporuje pouze "head node", který pro změnu není určen pro náročné výpočty.
2. Kvůli bezpečnosti clusteru je vždy nutné pracovat bez sudo práv.

7.1.3 Rezervace výpočetního výkonu pomocí Slurm

Po připojení na cluster se uživatel nachází na stroji head, který slouží jako vstupní brána do clusteru a není určen na spouštění náročných výpočtů, neboť by mohly zneprůchodnit přístup na cluster. Pro náročné výpočty je nutno alokovat výpočetní výkon pomocí systému Slurm. Pro práci se slurmem se používají mimo jiné dva základní příkazy: srun a sbatch.

Příkaz srun

Příkaz srun uživatele podle zadaných požadavků přepojí na výpočetní node. Tento příkaz je interaktivní a blokující, tedy uživatel dále pracuje na příkazovém řádku cílového stroje a výsledky se vypisují do konzole. Výhodou příkazu je interaktivita. Nevýhodou je závislost rezervovaného výkonu a právě probíhajících operací na síťovém spojení vzdáleného nodu s vlastním počítačem uživatele. Např. při výpadku sítě nebo při restartu počítače uživatele může automaticky dojít k uvolnění výpočetního výkonu a ukončení probíhajících operací. Příkaz srun se hodí pro manuální úlohy vyžadující přítomnost uživatele, naopak pro automaticky běžící úlohy se nehodí.

Příkaz srun, který byl použit pro alokaci výpočtů pro některé zde uvedené experimenty, vypadá takto:

```
srun --mem=64G --time=0-35:00:00 --partition=gpu --gres=gpu:1
     --cpus-per-gpu=16 --job-name=MyJob_%j --pty bash
```

Tento příkaz alokuje paměť RAM 64GB. Časový limit na dokončení výpočtů je 35 hodin. Po vypršení limitu budou všechny běžící procesy ukončeny a

alokovaný výpočetní výkon bude uvolněn. Bude rovněž umožněno využívat výkon GPU.

Příkaz sbatch

Příkaz sbatch spustí na výpočetním stroji zadaný skript. Tento příkaz není ani interaktivní ani blokující, tedy uživatel není na cílový výpočetní stroj vůbec přesměrován. Výsledky se nevypisují do konzole, ale do souboru. Tento příkaz se hodí pro automaticky běžící úlohy, ale pro manuální práci nelze použít. Příkaz sbatch, který byl použit pro alokaci výpočtů pro některé zde uvedené experimenty, vypadá takto:

```
sbatch runall.sh
```

Příkaz sbatch spustí zadaný skript. Součástí skriptu je i definice rezervovaného výpočetního výkonu pro slurm, seznam načítaných knihoven a spuštění samotných výpočetních úloh:

```
#!/bin/bash
#SBATCH  --mem=128G
#SBATCH --time=0-35:00:00
#SBATCH --partition=gpu
#SBATCH --gres=gpu:1
#SBATCH --cpus-per-gpu=16
#SBATCH --job-name=MyJob_%j.log

source activate inlocul
ml load MATLAB
ml load CUDA/9.1.85
ml load SuiteSparse/5.1.2-foss-2018b-METIS-5.1.0
ml load cuDNN/7.0.5-CUDA-9.1.85
# samotný příkaz spuštění výpočetní operace
matlab -batch run_inloc_demo_100_10
```

7.2 Způsob měření výkonu

Všechny experimenty byly spouštěny na clusteru a byl jim přidělen stejný výpočetní výkon a stejný hardware. Všechny experimenty byly spouštěny pomocí sbatch nebo srun s výše zmíněnými parametry, tedy 128 GB RAM, CPU (Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz) a jedno zařízení GPU (NVIDIA Tesla V100 SXM2 32 GB).

Měří se **celková doba běhu v sekundách**. Pro tento algoritmus je to vhodnější než měření procesorového času, protože součástí algoritmu je několik I/O operací, např. načítání snímků a jejich popisných matic z databáze. Tyto souborové operace by nebyly při měření procesorového času zaznamenány.

Každý experiment se spouští opakovaně (počet opakování je vždy uveden v popisu experimentu) a doba běhu se spočítá **aritmetickým průměrem** všech běhů.

7.3 Výkon Matlabu

Kód Matlabu se nekompiluje, ale je spouštěn interpreterem. Výkon je proto omezován nutností interpretovat jednotlivé příkazy a spravovat pracovní prostor. Nelze ani zdrojový kód nijak optimalizovat. Z toho plyne fakt, že při využívání Matlabu se vyplatí využívat hromadné maticové operace, pro které interpretace příkazu zabere mnohem kratší dobu než samotné provedení příkazu. Maticové výpočty se navíc interně provádějí ve zkompilovaných knihovnách. V některých situacích je ale nutné využít místo hromadných operací např. cykly, což má za následek interpretování mnoha drobných operací, které při spouštění skriptů způsobují velké latence. Pokud záleží na výkonu, je nutné tyto části přepsat do C/C++ a zkompilovat jako knihovnu MEX.

Následující tabulka ukazuje příklady operací Matlabu a jejich výkon při použití maticových operací v porovnání s rozepsáním těchto operací do for cyklů. Tento experiment byl spuštěn a změřen dle metodologií popsaných v 7.2 a 7.1.3. Byl opakován 100x a uvedené časy jsou aritmetickým průměrem všech běhů dané funkce. Kód experimentu je k nahlédnutí v souboru test_matlab_performance.m.

operace	trvání maticové operace [ms]	trvání for cyklu [ms]
Vytvoření 10^7 náhodných 3D bodů	235	870
Projekce 10^7 bodů kamerou	171	20746
Normalizace 10^7 promítnutých bodů	58	6108

Tabulka 7.1: Ukazuje výkony vybraných operací, pokud jsou provedeny maticově (tj. zpracují se všechna data najednou), nebo jsou provedeny pomocí for cyklu.

Vytvoření náhodných 3D bodů je nejjednodušší operace s minimem maticových operací. V experimentu uvažujeme 10^7 bodů, tj. stejný počet cyklů v kódu. Porovnání běží v prostředí bez dalších proměnných a ukazuje primárně zpoždění způsobené interpretací.

Projekce bodů vyžaduje převedení bodů do homogenních souřadnic. Je mnohem efektivnější rozšířit celou matici bodů než každý bod převádět jednotlivě. Deset milionů těchto drobných operací vyžaduje řádově více času než jedna maticová operace.

Normalizace bodů je vydělení výsledných promítnutých 2D bodů jejich třetí homogenní souřadnicí. Jedna vektorová operace si vyžádala 105,3× méně času.

7.4 Profilace produkční verze (bez paralelního kódu)

Algoritmus InLoc se obvykle spouští na více vláknech, protože některé jeho části lze jednoduše rozdělit do nezávislých podúloh. Vestavěná profilace kódu v Matlabu ale neprofiluje paralelní kód detailně. Výsledné profilace dokumentují paralelní kód jako nedělitelnou jednotku, tedy jednotlivé části paralelního kódu nejsou změřeny odděleně. Aby bylo možné vybrat nejpomalejší části, které jsou vhodné k přepisu, bylo nutné spustit celý algoritmus na jednom vlákně (tj. nahradit všechny paralelní cykly parfor za cykly for). Tabulka ukazuje výsledky této upravené implementace určené k profilaci všech funkcí.

funkce / velikost databáze	261	522	1044	2088	počet volání
Celkový čas běhu InLocu [s]	933	948,5	1026,0	983,2	-
ht_retrieval	12,9	13,3	15,3	17,7	1
» getScore1Query	12,6	12,8	14,2	15,7	1
»» getFeatures1Query	12,2	12,0	12,9	13,3	1
»»» at_serialAllFeats_convfeat	8,1	8,0	8,4	8,6	1
ht_top100_densePE_localization	486,0	504,3	535	478,4	1
» parfor_dense_GV	440,2	459,7	486,7	425,4	10
»» at_coarse2fine_matching	251,6	268,9	286,4	219,0	10
»»» at_dense_tc	142,9	163,8	176,2	130,6	*27657
»» at_denseransac	181,2	179,0	189,6	194,6	100
»»» at_ransacH4	181,0	178,9	189,5	194,5	200
» parfor_dense_PE	25,6	24,6	29,1	31,5	10
»» ht_lo_ransac_p3p	25,3	23,1	27,2	28,6	10
» at_serialAllFeats_convfeat	7,0	6,9	7,4	7,5	1
ht_top10_densePV_localization	434,2	430,7	474,7	487,0	1
» parfor_densePV	433,8	430,5	474,5	486,8	10
»» projectMesh	426,0	422,55	466,4	478,5	10
»»» system	425,9	422,4	466,3	478,4	10

Tabulka 7.2: Profilace nezrychlené produkční verze, jedno vlákno

* Počet volání označený hvězdičkou je průměrný počet volání. Pro každý dotaz se počet volání liší.

7.5 Profilace produkční verze (s paralelním kódem)

funkce / velikost databáze	261	522	1044	2088	počet volání
Celkový čas běhu InLocu [s]	267,1	254,3	259,3	260,4	-
ht_retrieval	13,6	13,6	13,5	12,5	1
» getScore1Query	13,3	13,2	13,1	11,4	1
»» getFeatures1Query	13,0	12,8	12,7	10,5	1
»»» at_serialAllFeats_convfeat	7,9	7,7	8,7	6,7	1
ht_top100_densePE_localization	127,6	125,5	128,2	120,7	1
»» parallel_function	91,0	90,1	91,2	88,1	1
ht_top10_densePV_localization	125,85	125,0	118,4	117	1
» parallel_function	100,3	100,2	108,6	90,6	1

Tabulka 7.3: Profilace produkční verze.

7.6 Profilace zrychlené produkční verze (s paralelním kódem)

funkce / velikost databáze	261	522	1044	2088	počet volání
Celkový čas běhu InLocu [s]	**244,9**	**239,3**	**241,9**	**244,9**	-
ht_retrieval	13,5	12,5	14,0	14,6	1
» getScore1Query	12,8	11,8	12,8	12,9	1
»» getFeatures1Query	11,5	11,3	11,7	11,5	1
»»» at_serialAllFeats_convfeat	7,6	7,2	9,7	9,2	1
ht_top100_densePE_localization	**105,8**	**93,1**	**95,2**	**100,8**	**1**
» parfor_dense_GV	74,6	70,9	72,4	80,3	1
»» at_coarse2fine_matching	30,3	28,8	28,0	32,5	1
»»» parallel_function	28,1	27,8	26,1	27,5	1
»»» get_tcs_cublas [*]	1,4	1,2	1,2	1,4	1
»»» at_cnnfeat2vlfeat	4,5	3,5	3,6	4,3	202
»» parallel_function	33,6	27,8	27,6	32,6	1
ht_top10_densePV_localization	123,5	128,5	127,3	122,7	1
» parallel_function	106,9	101,6	110,2	109,2	1

Tabulka 7.4: Profilace zrychlené produkční verze. Tučným písmem jsou vyznačeny části, kde došlo díky nové implementaci ke zrychlení.

[*] Tato funkce je mnohonásobně rychlejší než původní funkce at_dense_tc díky využití C++ (knihovny MEX), grafické karty a hromadného zpracování dat.

7.7 Výběr nejlepší implementace get_tcs

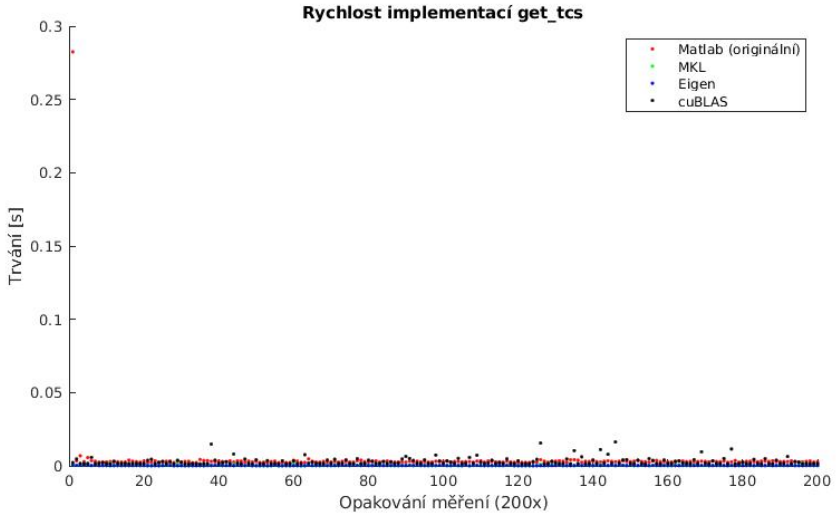
Funkce get_tcs, která byla napsána v C++ a má nahradit funkci Matlabu at_dense_tc, byla napsána ve více variantách, aby se vyzkoušelo více knihoven. Byly vyzkoušeny již zmíněné knihovny MKL, Eigen a cuBLAS. Knihovna cuBLAS provádí výpočty na GPU spolu s ní byl použit i kernel CUDA, který rovněž zrychluje běh funkce a minimalizuje objem přenášených dat z GPU do CPU. Všechny čtyři implementace využívají paralelní běh na více vláknech.

Původní at_dence_tc a všechny tři vatianty get_tcs byly spuštěny 200x na třech různě velkých úlohách. V nové implementaci se get_tcs volá ve funkci at_coarse2fine_matching na dvou místech – jednou pro nalezení tentativních

korespondencí mezi jedním dotazovým a m databázovými snímky, poté pro nalezení tentativních korespondencí v okolí jednoho bodu. Jsou to různě velké instance úlohy. Pro detailnější porovnání výkonu budou všechny čtyři varianty profilovány s různě velkými instancemi úlohy.

Malá úloha

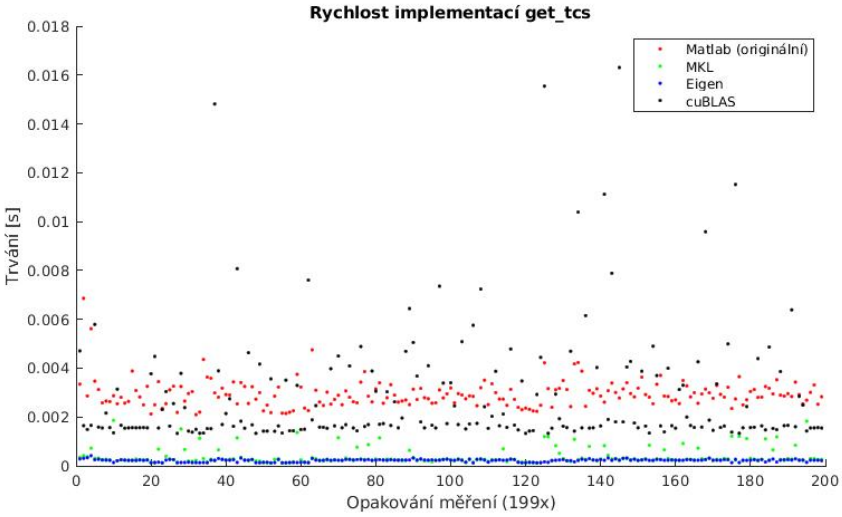
V této úloze se hledají tentativní korespondence mezi dvěma maticemi o rozměru 100×100. Úloha má ukázat, zda je použití cuBLAS výhodné pro menší instance úlohy hledání tentativních korespondencí, nebo zda se v těchto případech hodí jiná knihovna, která provádí výpočty na CPU. Pro tuto úlohu



Obrázek 7.1: Porovnání výkonu at_dense_tc s variantami get_tcs na malé úloze

Tento graf zobrazuje velmi nevyvážené výsledky. Graf ukazuje, že první spuštění původní funkce Matlabu at_dense_tc vykazuje násobně horší výkon než všechna další spuštění (viz. červená tečka vlevo nahoře). Je to způsobeno tím, že Matlab po prvním vstupu do paralelního cyklu (parfor) inicializuje paralelní běhové prostředí. Je to počáteční investice umožňující v dalších průchodech používat paralelismus. Jedná se o jednu z nevýhod používání Matlabu.

V dalších grafech pro větší názornost odstraňme zmíněné první volání. Další grafy tak zobrazují o jedno opakování experimentu méně.

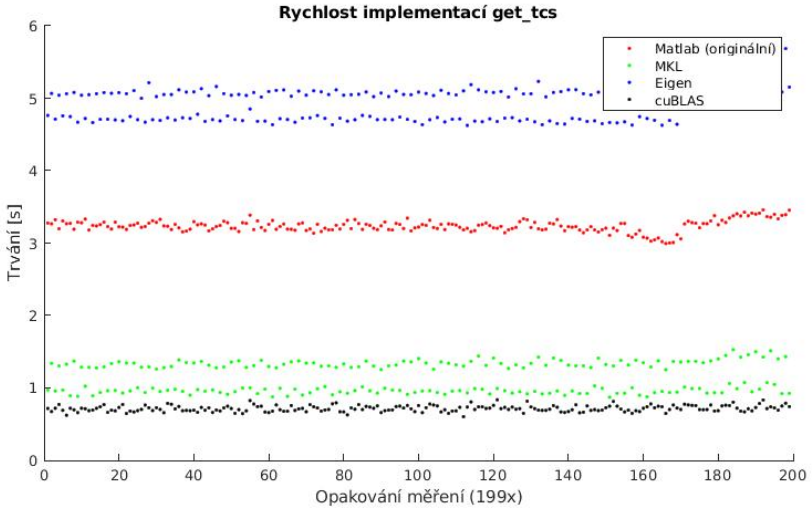


Obrázek 7.2: Porovnání výkonu at_dense_tc s variantami get_tcs na malé úloze

V této úloze dosáhla nejlepšího výkonu varianta get_tcs_eigen. Varianta get_tcs_mkl je také stále velmi výkonná, přibližně dvakrát rychlejší než původní funkce Matlabu at_dense_tc. cuBLAS dosáhl ve většině volání lepšího výkonu než původní funkce at_dense_tc, ale v některých voláních dopadl nejhůře. Pro menší úlohy nevykazuje implementace s GPU stabilní výkon, proto v těchto případech nebude využívána.

Střední úloha

V této úloze se hledají tentativní korespondence mezi maticí 7500×512 oproti stovce matic téhož rozměru. Simuluje to situaci, kdy máme jeden dotazový snímek (512 deskriptorů délky 7500) a 100 databázových snímků (pro každý 512 deskriptorů délky 7500). Tato situace se v algoritmu InLoc nevyskytuje a slouží jen k porovnání výkonu jednotlivých implementací.

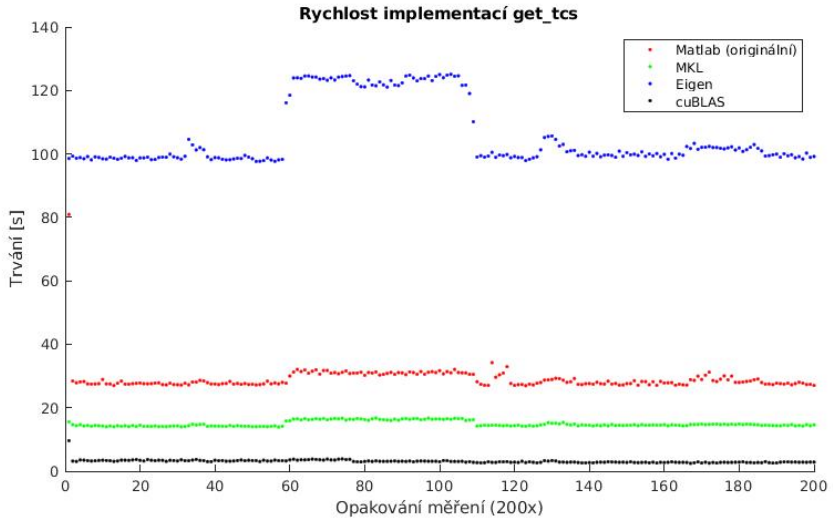


Obrázek 7.3: Porovnání výkonu at_dense_tc s variantami get_tcs na středně velké úloze

Větší úlohy hledání tentativních korespondencí ukazují, že je pro ně výhodnější využít výpočetní síly GPU, tedy použít variantu get_tcs_cublas. Varianta get_tcs_mkl je pomalejší, ale stále podstatně lepší než původní funkce Matlabu at_dense_tc. Nejnižšího výkonu dosáhla varianta get_tcs_eigen. Graf také ukazuje, že všechny čtyři testované funkce mají ve větších instancích úlohy jen malé výkyvy ve výkonu.

Velká úloha

V této úloze se hledají tentativní korespondence mezi maticí 512×7500 oproti stovce matic téhož rozměru. Tato úloha se skutečně v algoritmu InLoc objevuje a její výsledky jsou pro výběr implementace zásadní. Simuluje to situaci, kdy máme jeden dotazový snímek (7500 deskriptorů délky 512) a 100 databázových snímků (pro každý 7500 deskriptorů délky 512)



Obrázek 7.4: Porovnání výkonu at_dense_tc s variantami get_tcs na velké úloze

Velká úloha hledání tentativních korespondencí ukázala, že implementace get_tcs_eigen, tak jak byla zkompilována, dosahuje nejnižšího výkonu a úloha trvá obvykle 100 sekund a více. Varianta get_tcs_mkl a původní at_dense_tc dosahují násobně lepšího výkonu. Běh get_tcs_cublas, který využívá výpočetního výkonu GPU, trvá méně než 5 sekund, což je mnohem méně než u ostatních implementací.

Výsledek experimentu

Měření ukázalo, že pro velké instance úlohy hledání tentativních korespondencí je využití GPU zásadní, neboť se řádově zkrátí doba běhu. Pro malé instance se naopak vyplatí nechat výpočet na CPU. Původní funkce at_dense_tc v žádném experimentu nedosáhla nejvyššího výkonu, proto je na všech místech nahrazena.

Kapitola 8

Závěr

8.1 Úpravy implementace algoritmu

Implementace algoritmu byla upravena tak, aby simulovala reálný běh, což je posun od původní testovací verze směrem k reálnému využití. Díky tomu bylo možné určit vhodné části k úpravě či přepsání do C++. Upravená implementace si zachovala ukládání mezivýsledků do souborů pro testování a ladění. Je však určena k simulaci reálného běhu algoritmu. Úprava implementace zahrnovala:

1. Oddělení role databázového a dotazového snímku:
   - Dotazový snímek je jen jeden, v algoritmu se již nikde nepracuje se seznamy dotazových snímků
   - Matice skóre je předem neznámá, proto byly výpočet skóre podobnosti a extrakce features zařazeny do algoritmu.
   - Z přípravných kroků byly odstraněny všechny analýzy dotazových snímků
2. Není nutné ukládat mezivýsledky do souborů. Tato možnost sice nebyla odstraněna z důvodu nutnosti extrahovat různé informace a grafy, nicméně ukládání je možné vypnout.
3. Je možno vypnout možnost načítání mezivýsledků ze souborů – pak je nutné spustit algoritmus od začátku, což simuluje reálný běh.

8.2 Zrychlení upravené implementace

Variant zrychlení algoritmu bylo vyzkoušeno několik:

1. Přepis některých částí do C++
   - Výkony všech variant přepsané funkce popisují obrázky 7.1, 7.2, 7.3, 7.4
2. Nalezení typických úloh pro InLoc (a jemu podobné algoritmy), které lze zpracovávat na GPU.
   - Tento bod souvisí i s přepisem do C++. Pomocí cuBLAS a CUDA je možné přesunout vhodné úlohy na GPU a tím novou implementace ještě více zrychlit.
3. Vybráním nižšího počtu nejpodobnějších snímků a renderování nižšího počtu syntetických snímků.
   - Rozsáhlejší databáze snímků RGB-D umožní snížit počty nejpodobnějších snímků a renderů, ale zachová se přesnost.

Výkon nezrychlené produkční verze je detailně rozepsán v tabulce 7.3, výkon zrychlené v tabulce 7.4.

Nemá smysl zrychlovat algoritmus pro přípravu databáze (skript buildFeatures), protože příprava databáze není součástí algoritmu. Databáze se připravuje pouze jednou při instalaci InLocu a nepředpokládají se žádné její změny. Algoritmus při každém běhu už předpokládá existenci připravené databáze.

8.3 Typické chyby InLocu

Byly prodiskutovány typické chyby algoritmu, které se projevily v kapitole 6. Práce rovněž nabízí a diskutuje jejich nápravu.

8.4 Souvislost mezi velikostí databáze a snížením hodnoty m

Je možné rychlost běhu efektivně zkrátit tím, že se sníží hodnota m, nicméně to má negativní dopad na přesnost. Experimenty ukázaly, že tento dopad lze poměrně snadno vykompenzovat zvětšením datasetu, protože velikost datasetu má na přesnost větší vliv. Samozřejmostí je zahrnout každé místo ve zmapovaném prostředí z různých úhlů pohledu.

Příloha A

Literatura

[1] P. Lučivňák, "Visual localization with hololens," Master's thesis.

[2] P. Lučivňák, "Výzkumná implementace inloc." https://github.com/lucivpav/InLocCIIRC_demo. [cit. 19.5.2021].

[3] VLFeat, "Vlfeat, phow descriptors." http://3dvision.princeton.edu/pvt/depthImproveStructureIO/lib/vlfeat/doc//overview/dsift.html. [cit. 18.5.2021].

[4] P. Lučivňák, "Přípravné kroky pro výzkumnou implementaci algoritmu inloc." https://github.com/lucivpav/InLocCIIRC_dataset. [cit. 18.5.2021].

[5] R. Arandjelović, P. Gronat, A. Torii, T. Pajdla, and J. Sivić, "Netvlad: Cnn architecture for weakly supervised place recognition." https://arxiv.org/abs/1511.07247. [cit. 18.5.2021].

[6] Y. W. Yeong, "Netvlad: Cnn architecture for weakly supervised place recognition." https://towardsdatascience.com/netvlad-cnn-architecture-for-weakly-supervised-place-recognition-ce64b08bebaf. [cit. 18.5.2021].

[7] "Dokumentace rozhraní blas." http://www.netlib.org/blas. [cit. 13.12.2021].

[8] NVIDIA, "Dokumentace knihovny cublas." https://docs.nvidia.com/cuda/cublas/index.html. [cit. 17.5.2021].

[9] Intel, "Dokumentace knihovny mkl." https://software.intel.com/content/www/us/en/develop/articles/intel-math-kernel-library-documentation.html. [cit. 17.5.2021].

Příloha B

Zdrojové kódy

Veškeré kódy jsou dohledatelné na Gitu.

Algoritmus InLoc

Všechny tři varianty InLocu, kterými se tato práce zabývala, se nachází na této adrese v různých větvích:

```
https://github.com/dubenma/localization_service.git
```

1. Původní výzkumná/testovací verze:
   - branch: sebera
   - commit: Initial commit; f340ac4703bd753ee5b057a7acd87c9966228fd3
   - Tato větev nebyla touto prací nijak pozměněna a ukazuje stav InLocu před zahájením této práce.
2. Nová produkční verze:
   - branch: sebera_produkcni
   - commit: 671ada49cb799d8d428734567cc522bc3bc680c7
3. Nová zrychlená produkční verze:

- branch: sebera_produkcni_speedup
- commit: 6b5f0289c4b9da4310dac31aa1fcfce087a90964

Implementace get_tcs_*

Všechny tři implementace get_tcs_* jsou zabaleny do přílohy této práce. Všechny tři varianty get_tcs_*, kterými se tato práce zabývala, se nachází na této adrese v různých větvích:

```
https://gitlab.com/xmartin.sebera/inloc-get_tcs
```

1. MKL:
   - branch: MKL
   - commit: 1a91472b4ffa0937005199d22959586c569d55fc
2. cuBLAS:
   - branch: cuBLAS
   - commit: 82aa341f3d2459086d83f972c36244c3ad61f4ca
3. Eigen:
   - branch: EigenOMP
   - commit: 45253a2d6bd20fa07a751494673a62a17685d787

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Sebera** Jméno: **Martin** Osobní číslo: **466120**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra kybernetiky**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové vidění a digitální obraz**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Zrychlení lokalizace pomocí knihovny InLoc

Název diplomové práce anglicky:

Speedup the Localization by InLoc Method

Pokyny pro vypracování:

1. Seznamte se s lokalizační metodou InLoc.
2. Vyhodnoťte rychlost předzpracování dat a běhu pro jeden dotaz vzhledem k velikosti databáze.
3. Najděte nejpomalejší části kódy a navrhněte postup pro jejich zrychlení (např. přepis do jazyka C++).
4. Aplikujte navržené metody pro urychlení lokalizace.
5. Porovnejte čas potřebný pro přípravu databáze a samotného vyhledávání pozice před a po Vašich vylepšeních.

Seznam doporučené literatury:

[1] Taira, Hajime, et al. "InLoc: Indoor visual localization with dense matching and view synthesis." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.
[2] Arandjelovic, Relja, et al. "NetVLAD: CNN architecture for weakly supervised place recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.
[3] Torii, Akihiko, et al. "24/7 place recognition by view synthesis." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Michal Polic, aplikovaná algebra a geometrie CIIRC

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **17.01.2021** Termín odevzdání diplomové práce: **04.01.2022**

Platnost zadání diplomové práce: **30.09.2022**

Ing. Michal Polic	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Podpis studenta

Datum převzetí zadání