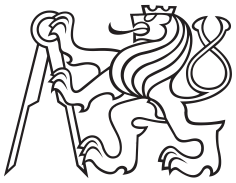
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra kybernetiky

Zrychlení lokalizace pomocí knihovny InLoc

Bc. Martin Sebera

Vedoucí: Ing. Michal Polic
Obor: Otevřená informatika
Zaměření: Počítačové vidění a digitální obraz
Květen 2021

Poděkování

Následujícím děkuji:

- vedoucímu a oponentovi mé práce
- vyučujícím, kteří nás po celou dobu studia provázeli
- své rodině
- spolužákům
- bývalým spolužákům
- spolubydlícímu
- ostatním přátelům
- kolejím Strahov
- kolegům
- městu Beroun
- městu Praha
- hotelu Bohemians

za to, že mi během mého studia pomáhali a tvořili příjemné prostředí (v rámci možností doby).

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, 19. května 2021

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 algoritmus odhadující pózu kamery z jednoho dotazového snímku (query image). Odhad pózy, tj. rotace a translace, je vypočítána na základě porovnání dotazového snímku s RGB-D snímky v databázi [1] (RGB-D je klasické RGB + hloubka, tj. pro každý pixel známe jeho Eukleidovskou vzdálenost od středu kamery). U RGB-D snímků v databázi je póza kamery vždy známa. Při výpočtu se aplikuje konvoluční neuronová síť NetVLAD pro hledání podobných snímků.

Husté korespondence se mezi snímky ověří pomocí homografií odhadnutých algoritmem DEGENSAC. Na základě vypočítaných korespondencí se odhadne pozice kamery. Na závěr se vygenerují syntetické snímky a porovnají se s dotazovým snímkem a vybere se pozice kamery vedoucí k nejmenším rozdílům mezi intenzitami jednotlivých pixelů.

1.1 Motivace

Lokalizace ze snímku má dva vstupy, tj. dotazový snímek a mapu 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. Oproti lokalizaci pomocí GPS nabízí lokalizace ze snímku podstatně lepší přesnost a navíc dokáže určit i rotaci kamery. Tyto lokalizační algoritmy lze využít např. v aplikacích virtuální reality [2] (promítání virtuálních objektů do skutečného prostoru vyžaduje velkou přesnost),

nebo v aplikacích pro mapování prostředí do 3D mapy.

Současná implementace [3] , tak jak byla publikována, se hodí spíše pro testování než pro reálné využití, a to z těchto důvodů:

1. S výjimkou několika funkcí je napsán celý algoritmus v jazyce Matlab, který má již ze své podstaty omezený výkon. Pro konečné softwarové produkty spíš určen není.
2. Mnoho mezivýsledků se ukládá do souborů, aby bylo možné celý dlouhý proces lokalizace snadno rozdělit do více nezávislých kroků, což je vhodné pro ladění, krokování a výzkumnou činnost. Nicméně kvůli mnoha souborovým operacím nezanedbatelně klesá výkon. V reálném využití nás zajímají zejména soubory týkající se finálních výsledků databázových snímků (extrahované části obrazu, pózy jejich kamer apod.)
3. Publikovaná implementace s reálným využitím nepočítá, je určena spíše pro výzkum. Např. není oddělena role databázových a dotazových snímků - obě množiny jsou totiž předem známy a všechny přípravné kroky se vykonávají současně pro obě množiny (místo toho, aby se předzpracovala data pouze pro databázové snímky).

Úkolem diplomové práce bude převést systém InLoc z dosavadní výzkumné verze do verze pro reálné využití. Znamená to hlavně reorganizaci jednotlivých kroků. Nová implementace bude spět k tomu, že dotazové snímky jsou předem neznámé a každý dotaz (typicky jen s jedním dotazovým snímkem) se bude zpracovávat oproti již kompletně připraveným databázovým snímkům. Část kódu bude přepsána do C++ (standard C++17) [4]. V tomto jazyce bude jednodušší optimalizovat výkon. Už ze své podstaty kompilovaného jazyka má oproti Matlabu a Pythonu výhodu v odstranění zbytečného kopírování dat a vytváření pomocných struktur pro běh kódu.

Kapitola 2

Implementace InLocu

Výzkumná implementace rozděluje proces lokalizace do více nezávislých kroků, využívajíc ukládání mezivýsledků do souborů. Toto rozdělení na kroky koresponduje s rozdělením kódu do několika skriptů, z nichž každý připraví své mezivýsledky pro další kroky. V této sekci jsou jednotlivé vysvětleny v tom pořadí, ve kterém jsou volány.

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 množina databázových snímků větší.

Algoritmus dostane na vstupu dotazový snímek $I \in IQ$, u kterého je potřeba určit pózu kamery (tj. střed kamery C a rotaci R). Implementace se dá shrnout do následujících kroků:

1. Pro každý dotazový snímek najde n nejpodobnějších snímků z databáze. Podobnost je měřena pomocí skóre (výpočet viz. rovnice 2.3). Výběr snímků se bude postupně zpřesňovat geometrickou verifikací a fotometrickou verifikací.

2. Geometrická verifikace pro každý pár (I_1, I_2), kde $I_1 \in IQ$ a $I_2 \in ID$, 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 2.2.1). Mezi korespondující pixely se najdou inliery pomocí algoritmu RANSAC, který odhaduje homografie.

3. Seřadit n vybraných nejpodobnějších snímků podle nového skóre (počtu inlierů z geometrické verifikace + původní skóre z rovnice (2.3)) od nejvyššího skóre po nejnižší. Dále se zpřesní výběr snímků, tedy algoritmus vybere jen m nejpodobnějších snímků ($m < n$)

$$\text{skóre2} = \text{skóre} + \text{počet\_inlierů} \tag{2.1}$$

4. Využije se toho, že databázové snímky jsou RGB-D (tedy pro každý pixel je známa hloubka). Nalezené korespondující pixely jsou tedy 2D-3D, díky tomu můžeme algoritmem P3P-RANSAC odhadnout pózu kamery.

5. Pro každou z m odhadnutých pozic kamer je renderován syntetický snímek. Syntetický snímek vznikne promítnutím 3D bodů do roviny odhadnuté kamery.

6. Vybere se pozice kamery, pro kterou je syntetický snímek nejpodobnější dotazovému snímku. Podobnost je určena pomocí skóre, které se vypočítá takto:

$$\text{skóre3} = 1 \ / \ \text{m}_{err} \tag{2.2}$$

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 [5].
Čím vyšší *skóre*3, tím je menší rozdíl mezi dotazovým a syntetickým snímkem.

2.2 Přehled kroků

Výše bylo popsáno šest kroků, jak InLoc funguje. Zde je popsáno několik skriptů, které se spouští v uvedeném pořadí a víceméně korespondují s výše popsanými kroky.

2.2.1 Přípravné kroky

Nesouvisí přímo s algoritmem InLoc, ale připraví předpočítanou reprezentaci databázových snímků [6], 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 jednoznačně 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 [7] [8], 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 známou 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).

$$\text{skóre} = \text{softmax}(\text{F}_Q^T F_D) \in \mathbb{R}^{q \times d} \tag{2.3}$$

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 $\mathbb{R}^{q \times d}$

2.2.2 Kroky algoritmu

1. **Výběr 100 top snímků (skript ht_retrieval)** - pro každý dotazový snímek vybere 100 databázových snímků s nejvyšším skóre. Matice skóre

již byla předpočítána přípravnými kroky.

2. **Odhad póz (skript ht_top100_densePE_localization)** - pro každý dotazový snímek se odhadne 10 potenciálních póz kamery.

3. **Verifikace odhadnutých póz (skript ht_top10_densePV_localization)** Pro každou odhadnutou pózu kamery vygeneruje syntetický snímek ze 3D modelu zmapovaného prostředí. Následně se vyberou pouze nejpřesněji odhadnuté pózy, tj. takové pózy, jejichž syntetický snímek je nejpodobnější příslušenému dotazovému snímku.

Následující podsekce popisují detailně jednotlivé jmenované funkce algoritmu InLoc.

Výběr 100 top snímků

Tuto úlohu vykonává skript ht_retrieval. Je potřeba znát matici obodování, kterou získáme voláním buildScores (2.2.1). Skript ht_retrieval pro každý dotazový snímek vybere 100 databázových snímků s nejvyšším obodováním. To znamená 100 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. cutouts_imgnames_all = $d\times$ string, názvy všech databázových snímků
2. ImgList = q$\times$ struktura s informacemi o dotazovém snímku a podobných nalezených snímcích. Přesněji:
   - queryname: název souboru dotazového snímku
   - topNname: 100 textových hodnot (názvy 100 nejpodobnějších snímků)
   - topNscore: 100 čísel float - obodování 100 nejpodobnějších snímků
3. query_imgnames_all = q$\times$ string, názvy souborů dotazových snímků

4. score = q×d čísel float (obodování pro všechny dvojice dotazový snímek-databázový snímek)
5. top100_matname (string) název souboru, kam se uloží output

Detaily funkce jsou popsány v pseudokódu [1].

Pseudokód 1: ht_retrieval (původní výzkumná implementace)

```
Vstup  : Matice skóre S (q×d)
Výstup : imgList (pro každý dotazový snímek 100 nejpodobnějších
         snímků)
if existuje_souor(top100_nazev_souboru) then
    načíst(top100_nazev_souboru);
else
    imgList[100];
    for každý i-tý dotazový snímek I ∈ IQ do
        skore, top_indexy = seřadit_databázové_snímky_dle_skóre(S,
          i);
        imgList[i].topNscore = top_100_nejvyšších_skóre(skore);
        imgList[i].topNname =
          názvy_snímků_s_top_100_nejvyšších_skóre(top_indexy);
    end
    uložit(imgList);
end
```

Odhad póz kamery

Tuto funkci vykonává skript ht_top100_densePE_localization. Pro každou dvojici (I_1, I_2) (kde $I_1 \in IQ$ a $I_2 \in ID$) odhadne až 10 póz kamery.

Načtou se předpočítané popisné vektory pro všechny dotazové a databázové snímky. Pro každý databázový snímek z vybrané podmnožiny sta 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 původní skóre vyšší než 1), ve tvorbě skóre je důležitější počet inlierů. Následně proběhne seřazení nejlepší stovky snímků dle tohoto nového skóre. Pro 10 databázových snímků s nejvyšším upraveným 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: 100 textových hodnot (názvy 100 nejpodobnějších snímků)
  - topNscore: 100 čísel float - obodování 100 nejpodobnějších snímků

Výstupem je rozšířený ImgList:

- ImgList = q×struktura s informacemi o dotazovém snímku a podobných nalezených snímcích:
  - queryname: název souboru dotazového snímku
  - topNname: 100 textových hodnot (názvy 100 nejpodobnějších snímků)
  - topNscore: 100 čísel float - obodování 100 nejpodobnějších snímků
  - P: 10x matrix(3,4) - 10 odhadnutých póz

Skript je popsán v pseudokódu [2].

2.2.3 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 2.2.2). Upraví dosavadní ImgList tak, že ke každému dotazovému snímku vybere pouze 10 nejvhodnějších databázových snímků místo původních 100. Podobnost je měřena pomocí výsledného skóre verifikace póz, které je popsáno v rovnici (2.2).

Detaily skriptu ht_top10_densePV_localization jsou popsány v pseudokódu [3]

Vstupy.

- imgList spočítaný krokem pro odhad póz kamer (sekce 2.2.2)

Pseudokód 2: ht_top100_densePE_localization (původní výzkumná implementace)

```
Vstup  : ImgList (pro každý dotazový snímek 100 nejpodobnějších
         snímků)
Výstup : ImgList (pro každý dotazový snímek 100 nejpodobnějších
         snímků a 10 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...100) databázový nejpodobnější snímek
         I_2 ∈ ID do
            gv = geometrická_verifikace (I_1, I_2);
            uložit(gv);
        end
        for každý i-tý (i=1...100) databázový nejpodobnější snímek I_2
         do
            gv = načíst_i_ty_gv();
            imgList[i].skóre += gv.počet_inlierů;
        end
    end
    seřazený_imgs = seřadit_podle_nového_skóre_sestupně(imgList);
    uložit(seřazený_imgs);
end
```

Výstupy. ImgList_densePV je modifikovaný ImgList_densePE - u každého dotazového snímku je jen 10 nejpodobnějších snímků a jejich skóre (namísto původních 100 v ImgList_densePE). Přesněji:

- queryname: název souboru dotazového snímku
- topNname: 10×string (názvy 10 nejpodobnějších snímků)
- topNscore: 10×float - obodování 10 nejpodobnějších snímků
- P: 10x matrix(3,4) - 10 odhadnutých póz

Pseudokód 3: ht_top10_densePV_localization (původní výzkumná implementace)

```
Vstup  : ImgList (pro každý dotazový snímek 100 nejpodobnějších
         snímků a 10 odhadnutých póz)
Výstup : ImgList (pro každý dotazový snímek 10 nejpodobnějších
         snímků a 10 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...10) nejpodobnější snímek I_2 ∈ ID do
            verifikace_pózy(I_1, I_2, imgList.odhadnute_pozy);
        end
        for každý i-tý (i=1...10) nejpodobnější snímek I_2 ∈ ID do
            imgList[i].skóre = načíst_skóre_po_verifikaci(I_1 I_2);
        end
        seřazený_imgs =
          seřadit_podle_nového_skóre_sestupně(imgList);
        uložit(seřazený_imgs);
    end
end
```

2.3 Produkční implementace

Přepsání algoritmu z testovací do produkční verze s sebou nese rozsáhlé reorganizace a přepisování. Cílem je zvýšit výkon a posunout projekt od testování k reálným úlohám.

2.3.1 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 testovací běh algoritmu se zvolí jeden dotazový snímek, který bude považován za předem neznámý a proto pro něj nebude předpočítáno skóre.

2.3.2 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 předem neznámý, proto výpočet skóre není přípravným krokem, ale součást prvního kroku algoritmu. Matice skóre už nemá rozměr $q \times d$, ale pouze $1 \times d$.

2.3.3 Přepsání nejpomalejší části

Analýza profilace a kódu ukázala, že vhodným kandidátem na přepis do C++ a zkompilování do MEX souboru je funkce at_dense_tc a všechny funkce, které jsou volány o úroveň níž. Důvodů je několik a jsou popsány v sekci (3.2)

2.3.4 Reorganizace skriptů

Produkční implementace si vynutila změnit strukturu některých skriptů. Úplně zanikl skript buildScores, protože tento přípravný krok již nemá smysl. V mnoha skriptech se využívalo ukládání mezivýsledků do souborů, aby bylo možné algoritmus rozdělit na nezávislé kroky. V nové implementaci je většina I/O operací odstraněna a skript není určen k rozdělení do nezávislých kroků.

2.3.5 Přípravné kroky v produkční verzi

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

1. **buildFeatures** - každý databázový snímek zpracuje neuronovou sítí NetVLAD. Výsledkem je vektor (feature), jehož cílem je jednoznačně popsat daný snímek vektorem reálných čísel. Dotazové snímky se samozřejmě neřeší, protože dotazový snímek je předem neznámý a zpracuje se až po startu algoritmu.
2. **buildScores** - skript byl úplně odstraněn.

2.3.6 Kroky algoritmu v produkční verzi

1. **Výběr 100 top snímků (skript ht_retrieval)**. Úloha tohoto skriptu je v produkční implementaci širší, jelikož příprava informací o dotazovém snímku byla začleněna do algoritmu. Musí extrahovat z dotazového snímku popisný vektor, k čemuž se používají stejné algoritmy jako pro popis databázových snímků. Po výpočtu popisného vektoru spočítá skóre pro jeden dotazový snímek oproti všem databázovým snímkům. Podle tohoto skóre vybere 100 nejpodobnějších databázových snímků.
2. **Odhad póz (skript ht_top100_densePE_localization)** - pro jeden dotazový snímek odhadne 10 potenciálních póz.

   Načte předpočítané popisné vektory pro databázové snímky. Pro každý databázový snímek z podmnožiny sta snímků provede geometrickou verifikaci, která zjistí počet inlierů mezi jedním dotazovým snímkem a databázovým snímkem. Následně přičte ke skóre každého databázového

snímku jeho počet inlierů z geometrické verifikace. Vzhledem k tomu, že původní skóre je výstup ze softmaxu (a proto nemůže být vyšší než 1), ve tvorbě skóre je důležitější počet inlierů. Následně proběhne seřazení nejlepší stovky snímků dle tohoto nového skóre. Následně pro 10 databázových snímků s nejvyšším upraveným skóre odhadne pomocí algoritmu P3P pózu kamery.

3. **Verifikace odhadnutých póz (skript ht_top10_densePV_localization)** Tato část algoritmu prodělala při převodu InLocu na produkční verzi nejméně změn. Byly např. odebrány souborové operace v souvislosti s mezivýsledky předešlých kroků.

Kapitola 3

Nová implementace v C++17

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++ a zkompilovat je do souboru MEX, což je dynamická knihovna, ve které je definována funkce mexFunction a je spustitelná z prostředí Matlabu.

3.1 Výkon Matlabu

Matlab je interpretovaný nekompilovaný jazyk, tj. kódy se nepřevádí na strojové instrukce, ale jsou spouštěny interpreterem. 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 zabere řádově více času než interpretace příkazů. Matlab je ale pomalý, pokud je zapotřebí využít v kódu cykly, nebo mnoho příkazů s malými objemy dat.

Následující tabulka ukazuje příklady operací Matlabu a jejich výkon při použití maticových operací v porovnání s rozepsáním těchto operací do for cyklů.

operace	trvání maticové operace [ms]	trvání for cyklu [ms]
Vytvoření 10^7 náhodných 3D bodů	430	1190
Projekce 10^7 bodů kamerou	382	28720
Normalizace 10^7 promítnutých bodů	129	8522

Tabulka 3.1: 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 zhruba 75x více času než jedna maticová operace.

Normalizace bodů je vydělení výsledných promítnutých 2D bodů jejich třetí homogenní souřadnicí. Jedna vektorová operace si vyžádala zhruba 65x méně času.

3.2 Výběr části k přepisu

Pro výběr vhodné části k reimplementaci bylo nutné provést profilace kódu vzhledem k různě rozsáhlým datasetům databázových snímků. Profilace ukázala, že většina částí algoritmu má vzhledem k velikosti datasetu databázových snímků konstantní složitost, protože algoritmus pracuje s podmnožinou 100 nejpodobnějších snímků. Pouze přípravný krok buildFeatures a první krok algoritmu (ht_retrieval), který vybírá podmnožinu 100 nejpodobnějších snímků, nemají vůči velikosti datasetu konstantní složitost. Krok buildFeatures slouží k přípravě datasetu a v samotném algoritmu InLoc se nespouští, proto není vhodný k reimplementaci. Krok ht_retrieval je v porovnání s ostatními kroky menší zátěží, a proto nebyl vybrán k přepsání.

Hlavní kritéria pro výběr části k reimplementaci jsou:

Počet databázových snímků	250	500	1000	2000
Délka běhu ht_retrieval [s]	14	15	16	17
Podíl kroku ht_retrieval	2,40 %	2,40 %	2,50 %	2,70 %

Tabulka 3.2: Délka běhu kroku ht_retrieval a jeho procentuální podíl na celkové době běhu algoritmu InLoc v závislosti na počtu databázových snímků.

1. Procentuální podíl dané části na celkové době běhu. 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.

Reimplementace se týká funkce at_dense_tc a všech funkcí, které jsou volány pod ní.

Důvody pro výběr této funkce k přepisu jsou následující:

1. 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), čímž se implementace vyhnula použití for cyklů, které by představovaly ještě větší zátěž.
2. Funkce je spouštěna více než 30000x a celková doba běhu převyšuje dvě minuty, což představuje necelou pětinu z celkové doby běhu algoritmu.
3. Tato funkce nachází tentativní korespondence mezi normovanými vektory reálných čísel. Experimenty ukázaly, že úspornější implementace s použitím vhodných 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. Pro výběr knihovny byla zpracována rešerše, jejíž závěry jsou uvedeny pro každou knihovnu zvlášť.

Počet databázových snímků	250	500	1000	2000
Délka běhu at_dense_tc [s]	136	136	137	135
Podíl at_dense_tc na době běhu	22,07 %	22,22 %	22,42 %	21,92 %
Délka běhu nové funkce get_tcs [s]	40	40	40	40

Tabulka 3.3: Délka běhu funkce at_dense_tc a její procentuální podíl na celkové době běhu algoritmu InLoc v závislosti na počtu databázových snímků. Nová implementace stejných výpočtů (MEX soubor get_tcs) vykazuje zhruba 3,3x lepší výkon.

3.3 Knihovna OpenBLAS

Knihovna OpenBLAS [9] [10] implementuje funkce pro mnoho základních matematických operací, přičemž se zaměřuje na operace s vektory a maticemi. OpenBLAS 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.) OpenBLAS 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 velmi urychlit výpočty. Podporovány jsou jazyky C (C++) a Fortran.

Funkce knihovny OpenBLAS jsou rozděleny do několika úrovní:

1. Level 1 poskytuje pouze operace s vektory (výpočet normy, skalární součty, násobení a 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.

Nevýhody knihovny OpenBLAS.

1. Operace jsou implementovány nízkoúrovňově - programátor tak není odstíněn od práce s hardwarem a pamětí. Špatná manipulace s knihovnou proto může vést k únikům paměti, neplatným přístupům do paměti apod.

3.3.1 Knihovna cuBLAS

Knihovna cuBLAS (CUDA Basic Linear Algebra Subroutine) [11] je implementací rozhraní BLAS na CUDA, kterou vyvinula společnost NVIDIA. Tato knihovna využívá výpočetního výkonu GPU. Při využívání cuBLAS je potřeba nahrávat data do paměti GPU a poté spouštět požadované operace. Tato knihovna může být využita pro urychlení operací s velkým množstvím dat (např. násobení velkých matic).

Výhody knihovny cuBLAS.

1. Tato implementace poskytuje vyšší výkon než jiné implementace BLAS využívající pouze výkonu CPU.
2. Knihovnu je možné využít i bez znalosti technologie CUDA - její využití může zůstat skryto v implementaci knihovních funkcí.
3. Teoreticky ji můžeme paralelně využívat společně s výpočty na CPU

Nevýhody knihovny cuBLAS.

1. Nutnost přenášet data mezi pamětí GPU a pamětí RAM.
2. Operace jsou implementovány nízkoúrovňově - programátor tak není odstíněn od práce s hardwarem a pamětí. Špatná manipulace s knihovnou proto může vést k únikům paměti, špatně zarovnaným alokacím paměti, neplatným přístupům do paměti apod.
3. Více paměťových operací snižuje čitelnost kódu.

Tato knihovna nakonec nebyla do projektu zahrnuta. Testy ukázaly, že knihovna MKL dosáhla lepšího výkonu.

3.4 Knihovna MKL

MKL vznikla jako implementace rozhraní BLAS od společnosti Intel [12], 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 aj. V tomto projektu byly využity pouze její funkce pro lineární algebru.

Nová implementace algoritmu InLoc využívá MKL kvůli optimalizovaným funkcím lineární algebry. Nevyužívá se ale vždy přímo - její využití je většinou ukryto pod knihovnou Eigen. Knihovna Eigen je plnohodnotná sama o sobě, ale můžeme zvýšit její výkon tím, že ji nastavíme, aby interně využívala knihovnu MKL. (Eigen potom automatizuje práci s pamětí, tj. alokace, kopírování apod., čímž snižuje komplexnost kódu a zvyšuje jeho robustnost proti chybám spojeným s pamětí.)

Výhody knihovny MKL.

1. Používání MKL je přehlednější než používání cuBLAS (nemusí se přenášet data mezi GPU a CPU).
2. MKL poskytuje jednoduché funkce pro hromadné zpracování dat na více vláknech, přičemž není potřeba zarovnávat alokovanou paměť jako v případě cuBLAS.

Nevýhody knihovny MKL.

1. Operace jsou implementovány nízkoúrovňově - programátor tak není odstíněn od práce s hardwarem a pamětí. Špatná manipulace s knihovnou proto může vést k únikům paměti, neplatným přístupům do paměti apod.

Testy ukázaly, že knihovna MKL dosáhla nejvyššího výkonu ze všech testovaných knihoven a nabízí mnoho užitečných funkcí. V tomto projektu byla MKL využita jako základ pro Eigen. Byla využita i funkce sgemm_batch pro paralelní zpracování 100 nezávislých maticových násobení.

3.5 Knihovna Eigen

Knihovna Eigen implementuje operace lineární algebry. Na rozdíl od zmíněných knihoven je však implementována jinak. Jedná se o soubor tříd a šablon

tříd jazyka C++, které zaobalují nízkoúrovňové operace.

Výhody knihovny Eigen.

1. Objektový kód jazyka C++ je mnohem více paměťově bezpečný a mnohem více automatizovaný než operace MKL. Programátor je tak odstíněn od práce s pamětí a hardwarem.
2. Objektový kód je přehlednější. Umožňuje používat přetížené operátory a členské funkce objektů, jejichž pojmenování je intuitivní.
3. Díky využití šablon typů může být mnoho důležitých informací známo už při kompilaci. Pokud např. dojde k násobení matic nekompatibilních rozměrů nebo různých číselných 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 napodobit požadované chování.
6. Eigen je implementována jako množina hlavičkových souborů. Do projektu se zařazuje velmi jednoduše, není potřeba nic instalovat.
7. Eigen se dá zkompilovat tak, aby interně využívala knihoven MKL, OpenBLAS apod.

Nevýhody knihovny Eigen.

1. Využívání elegantních a přehledných zápisů Eigenu vždy nemusí být nejvýkonnější varianta a snadno se do kódu dostanou neoptimální postupy. Knihovna Eigen např. nenabízí analogii k funkci sgemm_batch, kterou MKL poskytuje.

Při reimplementaci vybraných částí algoritmu se osvědčila kombinace Eigenu a MKL. Pro složitější operace je výhodnější a přehlednější využívat Eigen. Pro jednoduché, ale obsáhlé operace je výhodnější zvolit vhodnou funkci pro hromadné zpracování z knihovny MKL. Knihovna MKL se musí správně nalinkovat, ať už staticky nebo dynamicky. Toto linkování se liší dle kompilátoru a systému.

3.6 Nová implementace hledání tentativních korespondencí

Funkce at_dense_tc vyhledává tentativní korespondence mezi dvěma množinami normovaných reálných vektorů. Tyto vektory jsou extrahovány ze třetí a páté vrstvy sítě VGG16 a popisují části snímků.

Tato funkce byla reimplementována v C++ a pod názvem get_tcs. Na vstupu jsou tyto parametry:

1. const Eigen::Map<MatrixXf>& descriptorsQ
   - Matice, jejíž sloupce tvoří reálné vektory. Jedná se o popis jednoho dotazového snímku
2. const vector<Eigen::Map<MatrixXf> >& descriptorsDBs
   - Pole Matic, jejichž sloupce tvoří reálné vektory. Jedná se o popisy vybraných 100 databázových snímků. Funkce přebírá všechny tyto matice najednou kvůli využití hromadných výpočetních operací knihovny MKL.

Poznámka k šabloně Eigen::Map<_>. Spuštěný MEX soubor má přístup k maticím Matlabu pomocí ukazatelů typu float*. Každý tento ukazatel ukazuje na začátek jednorozměrného bloku alokované paměti, v němž je uložena matice o libovolném počtu dimenzí. Tato šablona knihovny Eigen umí adoptovat klasický ukazatel a zacházet s ním jako s jednorozměrnou či dvourozměrnou maticí, aniž by bylo nutné kopírovat paměť. Šablona navenek nabízí příslušné maticové operace. Samozřejmou nutností je předat v konstruktoru informace o rozměru matice.

Návratové hodnoty get_tcs.

1. vector<map<size_t, size_t> >
   - Velikost pole vector<_> je rovna počtu databázových snímků, které byly funkci předány. Uvažujeme jeden dotazový snímek.
   - Prvek vektoru je množina dvojic korespondujících indexů popisných vektorů (map<size_t, size_t>) pro jednu dvojici ($I_1 \in IQ$, $I_2 \in ID$ (dotazový a databázový snímek)

Pseudokód [4] popisuje novou implementaci v C++. Tento pseudokód neukazuje napojení nativního kódu C++ na data Matlabu ani detaily volání nízkoúrovňových operací MKL.

3.6.1 Kontrola výstupu

Novou funkci get_tcs je nutné otestovat, jestli vrací stejné tentativní korespondence jako původní funkce at_dense_tc. Kvůli používání čísel typu float dochází při výpočtu k numerickým chybám. Z toho důvodu lze tolerovat odchylky ve vzdálenostech mezi vektory popisující oblasti snímku, které jsou v řádu cca. $\epsilon = 10^{-7}$. Pro zjednodušení testuji, jestli je zachován počet korespondencí i korespondující indexy.

Tyto dvě funkce byly spuštěny se stejnými vstupy a jejich výstupy byly porovnány. Původní funkce vrací k tentativních korespondencí v matici o rozměru $3 \times k$, kde první a druhý řádek jsou indexy vektorů, které spolu tvoří korespondující pár. Třetí řádek je vzdálenost mezi korespondujícími vektory. Protože se vzdálenost vektorů neuplatňuje v žádných dalších výpočtech, nová funkce get_tcs ji nevrací. Tím se šetří paměť a čas. Pro otestování zachování počtu korespondencí a korespondujících indexů byla v Matlabu napsána testovací funkce test_matches.

Testovací funkce přebírá dva objekty typu cell. Každý cell obsahuje seznamy tentativních korespondencí pro každou dvojici snímků. Počet dvojic snímků při testování byl 100 (1 dotazový snímek a 100 databázových snímků). První cell obsahuje tentativní korespondence, které vrátila původní funkce at_dense_tc, druhý cell obsahuje korespondence vrácené funkcí get_tcs. Dvojice seznamů korespondencí vzešlé ze stejných snímků musí mít stejný počet prvků i obsahovat stejná čísla. V opačném případě se inkrementuje počet chyb.

Funkce vrací booleovskou hodnotu, zda se seznamy rovnají a počet nalezených chyb. Kód této funkce je popsán v pseudokódu [5] .

Testování ukázalo, že pouze jedna dvojice snímků neměla stejné výsledky. Korespondence nalezené novou funkcí byly sice správné, ale jejich počet se lišil o 1 (místo původních 374 bylo nalezeno jen 373 korespondencí). Tato chyba vzešla z numerických chyb číselného typu float a byla považována za přijatelnou. Proto považuji novou implementaci za validní a využívám ji v dalších experimentech.

3.7 Reorganizace funkce parfor_denseGV

Jedná se o funkci Matlabu provádějící geometrickou verifikaci. Její kód musel být upraven tak, aby mohl pracovat s novou funkcí get_tcs.

Funkce parfor_denseGV hledá inliery pro každý pár dotazového snímku s databázovým. Má k dispozici jejich popisné matice (což jsou data ze třetí a páté vrstvy sítě VGG16). Sloupce těchto matic jsou reálné vektory, které tvoří tentativní korespondence a mezi nimi funkce hledá pomocí algoritmu ransac inliery.

V původní implementaci funkce parfor_denseGV zpracovávala jen jednu dvojici dotazového snímku s databázovým. Výhoda byla, že uchovávání jediné dvojice popisných matic má malé nároky na paměť RAM.

Tato funkce byla kvůli zvýšení výkonu předělána tak, aby mohla zpracovávat co největší množství dat najednou. Proto přijímá všechny matice popisující databázové snímky najednou, což umožňuje spouštět vícevláknové hromadné operace knihovny MKL. Nevýhoda je, že je potřeba alokovat přibližně 20GB paměti RAM a také je nutno přeorganizovat původní kódy Matlabu, a to jak původní funkci parfor_denseGV, tak i funkce, které tato funkce volá. V praxi to znamená méně for cyklů při volání funkcí, protože popisné matice databázových snímků se již nepředávají každá zvlášť, ale všechny najednou. Na druhou stranu se objevily nové for cykly, které načítají a připravují argumenty pro funkce.

Původní implementace geometrické verifikace je v pseudokódu [6].

Nově upravená implementace je v pseudokódu [7].

3.8 Reorganizace at_coarse2fine_matching

Tato funkce je volána funkcí parfor_denseGV a dále volá nově napsanou funkci get_tcs. Proto musela být pozměněna její původní implementace, aby dostávala data nutná pro spuštění funkce get_tcs - efektivního vyhledání tentativních korespondencí.

Funkce at_coarse2fine_matching přijímá extrahovaná data ze sítě VGG16. Konkrétně se pracuje se třetí vrstvou, která má rozlišení 150x200x256 (150x200 je výška x šířka a 256 je hloubka) a s pátou vrstvou, která má rozlišení 75x100x512. Pátá vrstva není tolik detailní, ale oproti třetí vrstvě má dvojnásobnou hloubku.

Funkce at_coarse2fine_matching nejprve provede pomocí nově přepsané funkce get_tcs nalezení tentativních korespondencí mezi dvěma popisnými maticemi z páté vrstvy VGG16, což slouží k rychlému nalezení korespondencí důležitých bodů (snímek z páté vrstvy má nižší rozlišení). Tyto tentativní korespondence tvoří dvojice pixelů. Funkce spočítá souřadnice těchto pixelů a přepočítá, kde se tyto pixely nachází na snímcích vyššího rozlišení (150x200x256). Z těchto detailnějších dat se extrahují okolí těchto bodů (čtverce o délce hrany 3 pixely). Na těchto extrahovaných částech opět proběhne vyhledání tentativních korespondencí a tyto korespondence funkce vrací.

Originální implementace zpracovávající jen jednu dvojici ($I_1 \in IQ$, $I_2 \in ID$), viz. pseudokód [8].

Nová implementace zpracovávající T databázových snímků se dá shrnout do pseudokódu [9]:

3.9 Kontrola odhadu póz kamery

Kdyby byla nová funkce na výpočet tentativních korespondencí napsána chybně, muselo by se to projevit na odhadu póz kamery, protože výpočet pózy je závislý na tentativních korespondencích. Stejně tak by mohlo změnit výsledné pózy, kdyby reorganizované kódy Matlabu zpracovávaly data chybným způsobem. Na verifikaci póz kamery, což je poslední krok algoritmu InLoc, nemají tyto změny kódu přímý vliv, proto stačí zkontrolovat jen správnost pózy kamery.

Upravené implementace funkcí Matlabu spolu s novým výpočtem tentativních korespondencí vrátily ty stejné pózy kamery jako původní implementace. Proto považuji novou implementaci za korektní.

Pseudokód 4: get_tcs - nová funkce na výpočet tentativních korespondencí v C++

```
Vstup  : descriptorsQ (popis jednoho snímku I ∈ IQ),
         descriptorsDBs (popis pro T snímků I_D1...I_DT ∈ ID)
Výstup : Páry korespondujících indexů vektorů pro každou dvojici
         I, I_Di

q_norm = normalizovat_sloupce_matice(descriptorsQ);
dbs_norms[] = normalizovat_sloupce_matice(descriptorsDBs[]);
if počet(descriptorsDBs) > 1 then
    // Vícevláknové násobení knihovny MKL
    for i ∈ 0...počet(dbs_norms) do
        matice_podobnosti[i] = q_norm^T * dbs_norms[i];
else
    matice_podobnosti[0] = q_norm^T * dbs_norms[0];
// Tentativni korespondence (jeden dotazový snímek s
   každým databázovým)
tc[počet(descriptorsDBs)];
index_paru_IqId = 0;
// Paralelní for
for mp ∈ matice_podobnosti[] do
    lokalni_tc = [];
    // součin matic složených z normovaných vektorů dá
       matici "skóre"podobnosti. Čím vyšší skóre, tím
       více jsou si vektory podobné.
    sloupce_max_indexy_hodnoty = [];
    radky_max_hodnoty = [];
    for c, c_hodnota ∈ sloupce(matice_podobnosti[i]) do
        max_ve_sloupci = -nekonečno;
        max_idx_ve_sloupci = 0;
        for r, r_hodnota ∈ radky_ve_sloupci(c) do
            if max_ve_sloupci < r_hodnota then
                max_ve_sloupci = r_hodnota;
                max_idx_ve_sloupci = r;
            radky_max_hodnoty[r] = max(radky_max_hodnoty[r],
              r_hodnota);
        sloupce_max_indexy_hodnoty[c] = (max_ve_sloupci,
          max_idx_ve_sloupci);
    idx_sloupce = 0;
    for max_hodnota, max_index ∈ sloupce_max_indexy_hodnoty
      do
        if radky_max_indexy[max_index] == max_hodnota then
            lokalni_tc.pridat_tc(max_index, idx_sloupce);
        ++idx_sloupce;
    tc[index_paru_IqId] = lokalni_tc;
    ++index_paru_IqId;
```

Pseudokód 5: test_matches: kontrola správnosti tentativních korespondencí

Vstup : cell1, cell2 (2x pole tentativních korespondencí, kde cell1 jsou tentativní korespondence vrácené původní funkcí at_dense_tc a ce112 jsou tentativní korespondence vrácené novou funkcí get_tcs)

Výstup : -počet_chyb (počet dvojic snímků, pro které nebyly nalezeny identické tentativní korespondence).
-stejné (bool) (určuje, jestli jsou cell1 a cell2 stejné, tj. jsou to stejně dlouhá pole a obsahují stejná čísla)

```
počet_chyb = 0;
stejné = true;
if délka(cell1) == délka(cell2) then
    for i ∈ 1...délka(cell1) do
        // kor1 a kor2 jsou matice se dvěma řádky
           (seznamy dvojic, tj. indexů korespondujících
           vektorů)
        kor1= cell1[i][1:2, :];
        kor2= cell2[i][1:2, :];
        if kor1 != kor2 then
            stejné = false;
            počet_chyb+=1;
else
    stejné = false;
    return;
```

Pseudokód 6: parfor_denseGV (produkční implementace bez get_tcs)

Vstup : query_vgg16data, název_db_snímku

Výstup : inliers mezi dvěma snímky (jeden dotazový a jeden databázový)

```
db_vgg16data = načíst(název_db_snímku);
tentativni_k = matching(query_vgg16data, db_vgg16data);
inliers = ransac(tentativni_k);
```

Pseudokód 7: parfor_denseGV přijímající data pro T dvojic snímků (produkční implementace s get_tcs)

Vstup : query_vgg16data, název_db_snímku[T]
Výstup : inliers pro T dvojic dotazového snímku s databázovým

```
dbs_vgg16data[T] = načíst(dbnames);
// Funkce at_coarse2fine_matching byla upravena k
   přijímání všech dat najednou
tentativni_k[T] = matching(query_vgg16data, dbs_vgg16data);
for tc ∈ tentativni_k do
    // Funkce ransac nebyla upravena k přijímání všech
       dat najednou. Zůstala původní implementace
    inliers[i] = ransac(tc);
```

Pseudokód 8: at_coarse2fine_matching (produkční implementace bez get_tcs)

Vstup : dotazové_vgg16data, db_vggdata
Výstup : tentativni_k_list

```
// Extrakce dat z páté vrstvy VGG16 a změna tvaru
   matic do formátu knihovny vlfeat
desc1 = vgg_převést_na_vlfeat(dotazové_vgg16data[5]);
desc2 = vgg_převést_na_vlfeat(db_vgg16data[5]);
// Extrakce detailnějších dat ze třetí vrstvy VGG16
desc1fine = vgg_převést_na_vlfeat(dotazové_vgg16data[3]);
desc2fine = vgg_převést_na_vlfeat(db_vgg16data[3]);
// původní funkce na výpočet korespondencí
tent_k = at_dense_tc(desc1, desc2);
tentativni_k_list = [];
for tk ∈ tent_k do
    okolí1 = okolí_bodu_3x3(tk[1]);
    okolí2 = okolí_bodu_3x3(tk[2]);
    // původní funkce na výpočet korespondencí
    tentativni_k_list[i] = at_dense_tc(okolí1, okolí2);
```

Pseudokód 9: at_coarse2fine_matching (produkční implementace s get_tcs)

```
Vstup  : dotazové_vgg16data, db_vggdata[T]
Výstup : tentativní_k_list[T]
// Extrakce dat z páté vrstvy VGG16 a změna tvaru
   matic do formátu knihovny vlfeat
desc1 = vgg_převést_na_vlfeat(dotazové_vgg16data[5]);
for i ∈ 1...T do
    descs2[i] = vgg_převést_na_vlfeat(db_vgg16data[i][5]);
// Extrakce detailnějších dat ze třetí vrstvy VGG16
desc1fine = vgg_převést_na_vlfeat(dotazové_vgg16data[3]);
for i ∈ 1...T do
    desc2fine[i] = vgg_převést_na_vlfeat(db_vgg16data[i][3]);
// nová funkce na výpočet korespondencí, která přijímá
   T databázových snímků najednou
tent_k[T] = get_tcs(desc1, descs2);
tentativní_k_list = [];
for i in 1...T do
    for tk ∈ tent_k[i] do
        okolí1 = okolí_bodu_3x3(tk[1]);
        okolí2 = okolí_bodu_3x3(tk[2]);
        // Nová funkce na výpočet korespondencí
        tentativní_k_list[i] = get_tcs(okolí1, okolí2);
```

Kapitola 4

Experimenty

Tato kapitola vyšetřuje výkon dvou různých produkčních implementací.

4.1 Produkční verze bez nových implementací

Tabulka [4.1] analyzuje výkon jednotlivých kroků produkční implementace bez nové funkce get_tcs a souvisejících úprav. Kód této produkční verze vychází z implementace původní výzkumné verze InLocu. Zdrojový kód byl přepsán, aby splňoval požadavky produkční implementace, ale záměrem ještě nebylo zvyšovat výkon algoritmu přepisem jeho částí do C++.

4.2 Produkční verze s novými implementacemi

Tabulka [4.2] analyzuje výkon jednotlivých kroků produkční implementace, která v sobě zahrnuje novou optimalizovanou funkce get_tcs a úpravy souvisejících kódů Matlabu. V této implementaci byly zvoleny úspornější maticové operace, což snižuje časové i paměťové nároky. Tato implementace uplatňuje více hromadných operací, které běží na více vláknech.

funkce / velikost databáze	250	500	1000	2000	počet volání
Celkový čas běhu InLocu	616	612	611	616	-
ht_retrieval	14	15	16	17	1
» getScoreForQuery	14	15	15	17	1
»» at_serialAllFeats_convfeat	9	9	9	9	1
ht_top100_densePE_localization	441	437	435	425	1
» parfor_dense_GV	427	423	426	425	1
»» at_coarse2fine_matching	238	238	237	237	100
»»» at_dense_tc	136	136	137	135	34549
»»» at_dense_hashtable	79	80	79	80	200
»»» at_retrieve_fineposition	13	13	13	13	68898
»» at_denseransac	180	176	174	179	100
» parfor_dense_PE	13	14	14	14	10
»» ht_lo_ransac_p3p	13	13	13	13	10
ht_top10_densePV_localization	160	161	161	160	1
» parfor_dense_PV	160	161	160	160	10
»» projectMesh (python)	153	153	153	152	10

Tabulka 4.1: Profilace běhu produkční implementace algoritmu InLoc bez nové funkce get_tcs a bez upravené implementace souvisejících funkcí Matlabu pro jeden dotaz v závislosti na velikosti databáze, tj. počet snímků RGB-D se známou pozicí.

funkce / velikost databáze	250	500	1000	2000	počet volání
Celkový čas běhu InLocu	**538**	**523**	**530**	**533**	-
ht_retrieval	15	15	16	17	1
» getScoreForQuery	15	15	15	17	1
»» at_serialAllFeats_convfeat	9	9	9	9	1
ht_top100_densePE_localization	**341**	**327**	**333**	**334**	**1**
» parfor_denseGV	**327**	**314**	**319**	**320**	**1**
»» at_coarse2fine_matching	**141**	**139**	**140**	**140**	**1**
»»» get_tcs	**40**	**40**	**40**	**40**	**34449**
»»» at_dense_hashtable	79	79	79	79	200
»»» at_retrieve_fineposition	15	14	15	14	68896
»» at_denseransac	166	167	171	173	100
» parfor_dense_PE	14	14	14	14	10
»» ht_lo_ransac_p3p	13	14	13	13	10
ht_top10_densePV_localization	181	182	181	181	1
» parfor_dense_PV	178	178	177	178	10
»» projectMesh (python)	170	171	170	170	10

Tabulka 4.2: Profilace běhu produkční implementace algoritmu InLoc s novou funkcí get_tcs a s upravenou implementací souvisejících funkcí Matlabu pro jeden dotaz v závislosti na velikosti databáze, tj. počet snímků RGB-D se známou pozicí. Části algoritmu zrychlené díky funkci get_tcs jsou zvýrazněny tučně.

4.3 Popis vedlejších funkcí

Významné skripty a funkce byly popsány v předchozích kapitolách. Další funkce [3] zmíněné v profilační tabulce jsou popsány zde.

4.3.1 getScoreForQuery

Nahrazuje skript buildScores z výzkumné implementace. Na rozdíl od skriptu buildScores, který spočítal matici skóre mezi všemi dotazovými a všemi databázovými snímky, getScoreForQuery extrahuje popisný vektor z jednoho předem neznámého dotazového snímku a spočítá pro něj skóre vůči všem databázovým snímkům. Spouští se jako součást algoritmu InLoc ve skriptu ht_retrieval.

Vstup

1. 1 dotazový snímek
2. d databázových snímků

Výstup

1. Matice skóre o rozměru $1 \times d$

4.3.2 at_dense_hashtable

Algoritmus InLoc využívá data ze třetí a páté vrstvy sítě VGG16, přičemž popisné matice ze třetí vrstvy mají rozměr 150x200x256 (výška x šířka x hloubka), popisné matice z páté vrstvy mají rozměr 75x100x512. Pokud ignorujeme hloubku, pak má matice ze třetí vrstvy 4x větší plochu (tedy na jeden vektor z menší matice připadají ve větší matici 4 vektory).

Tato funkce pro každou souřadnici x,y z menší popisné matice vrátí 4 souřadnice, na kterých se nachází tentýž bod ve větší matici.

4.3.3 at_retrieve_fineposition

Jedná se o pomocnou funkci, která dostane na vstupu souřadnici v rámci popisné matice o rozměru 75x100x512, přepočítá, které body odpovídají této souřadnici v popisné matici o rozměru 150x200x256, a extrahuje z okolí těchto bodů popisné vektory oblasti snímku.

4.3.4 at_denseransac

Implementace algoritmu ransac odhadující až Nh homografií pro účely geometrické verifikace.

Vstup

1. f1, f2 - všechny kombinace souřadnic x, y
2. match - dvojice indexů souřadnic, které tvoří tentativní korespondencí
3. Nh - cílový počet odhadnutých homografií

Výstup

1. H - odhadnuté homografie
2. inliers - indexy inlierů

4.3.5 ht_lo_ransac_p3p

Slouží k odhadu pózy kamery z korespondujících 3D bodů reálného světa a jejich 2D průmětů na snímek.

Vstup

1. u - souřadnice bodů na snímku

2. X - souřadnice 3D bodů v reálném světě

Výstup

1. P - odhadnutá póza kamery
2. inliers - množina použitých 2D-3D korespondencí

4.3.6 projectMesh

Tato funkce spouští skript pythonu jako externí proces, který vyrenderuje 3D model zmapovaného prostředí pomocí odhadnuté perspektivní kamery.

Vstup

1. meshPath - cesta k 3D objektu
2. f - ohnisková vzdálenost objektivu
3. R - rotace kamery
4. t - translace kamery
5. sensorSize - výška a šířka snímače kamery
6. ortho - pokud je true, použije se k renderování ortografická kamera, jinak se použije perspektivní
7. mag - zvětšení podle os x a y (číselný parametr pro ortografickou kameru)
8. projectMeshPyPath - cesta ke spouštěnému skriptu
9. headless - tento parametr nebyl v současné implementaci použit.

Výstup

1. RGBcut - vytvořený syntetický snímek ve formátu RGB
2. XYZcut - Souřadnice x,y,z reálného světa pro každý pixel snímku RGBcut
3. depth - hloubková mapa syntetického snímku RGBcut

Kapitola 5

Závěr

Výstupem této práce je nová implementace lokalizační metody InLoc. Cílem byl převod výzkumné verze na produkční verzi, což obnášelo následující úkoly:

1. Oddělení role dotazových a databázových snímků.
   - Ve výzkumné verzi byly obě množiny předem známy.
   - V produkční verzi se pracuje jen s jedním dotazovým snímkem, který je předem neznámý. To obnášelo odstranit některé přípravné kroky týkající se výpočtu skóre a implementovat je do samotného algoritmu. Zjistit skóre pro jeden dotazový snímek zabere cca 14-17 sekund (viz. kapitola 4).
2. Další předělání dosavadního kódu Matlabu
   - Produkční verze algoritmu InLoc není určena rozdělení do více nezávislých kroků pomocí ukládání mezivýsledků do souborů. Algoritmus je více uzpůsoben běhu v reálném využití. Další rozsáhlejší úpravy kódu si vyžádalo zavedení nové funkce get_tcs, jelikož pracuje s větším množstvím dat najednou.
   - Část kódu, kde byla použita knihovna pyrender (používaná k vytvoření syntetických snímků), nefungovala stabilně. Důvodem byla chybná konfigurace na straně Matlabu. To si vynutilo implementaci nastavování některých hodnot systémového prostředí.
3. Vyhodnocení výkonu jednotlivých kroků a zrychlení vybrané části přepisem do C++.

- Analýza výkonu jednotlivých částí algoritmu umožnila vybrat vhodnou funkci k přepisu. To dále obnášelo provést rešerši dostupných matematických knihoven.
- K přepisu byla vybrána funkce pro vyhledání tentativních korespondencí, jejíž počet volání je v řádu desítek tisíc. Funkce je nyní implementována jako dynamická knihovna MEX a dosahuje zhruba 3,5x vyššího výkonu než původní implementace.
  - Rozsáhlé maticové operace provádí knihovna MKL na více vláknech. Knihovna MKL byla vybrána na základě rešerše dostupných řešení.

5.1 Porovnání výkonu přepsané části v C++ s původní implementací

V kapitole [4] je porovnán výkon dvou produkčních verzí algoritmu InLoc, přičemž starší verze nespouštěla nově napsanou funkci na výpočet tentativních korespondencí. Běh původní funkce at_dense_tc průměrně trval 135 sekund. Nová funkce get_tcs je násobně rychlejší, běh trvá průměrně jen 40 sekund. To přispělo k tomu, že produkční verze algoritmu InLoc, která používá get_tcs, je zhruba o 90 sekund rychlejší.

Příloha A

Literatura

[1] H. Taira, M. Okutomi, T. Sattler, M. Cimpoi, M. Pollefeys, J. Sivic, T. Pajdla, and A. Torii, "Inloc: Indoor visual localization with dense matching and view synthesis," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7199–7209, 2018.

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

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

[4] Cppreference.com, "Specifikace standardu c++ 17." `https://en.cppreference.com/w/cpp/17`. [cit. 18.5.2021].

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

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

[7] 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].

[8] 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].

[9] OpenBLAS, "Dokumentace knihovny openblas." https://www.openblas.net/. [cit. 19.5.2021].

[10] Z. Xianyi, "Knihovna openblas." https://github.com/xianyi/OpenBLAS.git.

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

[12] 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

1. Projekt na GitLab s kódy nové funkce get_tcs a přidružených knihoven pro tvorbu MEX (commit fb7392b4):

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

2. Projekt na GitLab s produkční implementací algoritmu InLoc (commit bde1f4b6):

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

   Neobsahuje celý systém InLoc. Zveřejněné soubory nahrazují původní implementace. Jsou uzpůsobeny pro běh s jedním dotazovým snímkem a k využívání nové MEX knihovny get_tcs.

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: **21.05.2021**

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