6월 24일 최종 결과 발표

1. 최종 개발 목표 및 결과 요약

1-1. 최종 개발 목표 상세

> 💡 **① 다양한 알고리즘을 적용하여 실험을 할 수 있는 모듈러 아키텍쳐 VSLAM 프레임워크 만들기**
> **② 해당 프레임워크를 타 OpenSource SLAM과 성능 비교하기**

- 프레임워크 카메라 종류
  - Stereo 카메라를 지원할 것
- 실험 대상이 될 기능 조건
  - Feature 관련: ORB, AKAZE를 포함하여 최소 3개의 실험 가능한 기능 개발
  - Frontend 관련: 최소 5개 이상의 실험 가능한 조합 개발
  - Backend 관련: 최소 5개 이상의 실험 가능한 조합 개발
  - Loop closure 관련: 최소 5개 이상의 실험 가능한 조합 개발 (활성화/비활성화 기능 필수)
- CI/CD에서 자동 빌드를 지원할 것
- GitHub repository 링크를 공유할 것

1-2. 최종 개발 결과 요약

- 개발 기간: 2022.06.13.월 ~ 2022.06.24.금 (총 2주)
- 개발 인원: 4인 (한은기(PM), 이창준, 이현진, 유희평)
- ProSLAM을 기반으로 프레임워크를 제작함. 선정 이유는 아래와 같음.
  - 구현과 관련한 내용이 논문에 상세하게 기술되어 있음.
  - 각 구현부마다 상세한 주석이 포함되어 있음
  - 한 Thread 에서 모든 과정이 진행되어 다소 delay가 발생할 수는 있으나 복잡도가 낮음
  - 더 나은 정확도를 위하여 Monocular 대신 Stereo 카메라 및 RGB-D 카메라를 지원함.
- 실험을 위한 각 알고리즘 구현 내용을 아래에 간단히 기술함. 더욱 자세한 내용은 [**2. 개발 결과 상세**]에서 계속.
  - Feature 관련: Feature Detector와 Descriptor를 구분하여 각각 4개씩의 선택이 가능
    - Detector Type [4개]: AKAZE, FAST, KAZE, ORB
    - Descriptor Type [4개]: AKAZE, BRISK, KAZE, ORB
  - Frontend 관련: Matching 방식을 선택할 수 있으며, 좋은 matching을 찾을 시 Homography의 이용 여부 및 파라미터를 설정할 수 있음
    - Matching 방식 [6개]: FLANNBASED, BRUTEFORCE, BRUTEFORCE_L1, BRUTEFORCE_HAMMING, BRUTEFORCE_HAMMINGLUT, BRUTEFORCE_SL2
    - Homography 사용 여부 [2개]: Homography 사용, lowe's ratio 사용
    - Homography 사용 시 파라미터 [3개]: RANSAC, RHO, LMEDS
  - Backend 관련
    - Optimization Algorithm [3개]: Gauss-Newton, Levenberg, Dogleg
    - Linear Solver Type [3개]: CHOLMOD, CSPARSE, DENSE
  - Loop closure 관련
    - local map aligner [2개]: ICP, FAST-ICP

    - parameters [6개]: error_delta_for_convergence, maximum_error_kernel, maximum_number_of_iterations, minimum_number_of_inliers, minimum_inlier_ratio, anderson_m
    - on-off 가능
    - 추후 local map aligner를 추가하기 용이하게 코드를 리펙터링함
- GitHub Repository에서 CI/CD 구현 내용을 확인할 수 있음

---

2. 개발 결과 상세

> 💡 모든 파트의 실험은 설정 파일(예: configuration/configuration_kitti.yaml)의 인자를 바꿈으로써 선택 구동이 가능함.

2-1. ImageProcessing

입력받은 이미지로부터 (1) feature을 추출하고, 비교를 할 수 있는 (2) descriptor을 만드는 방법 각각을 선택해 실험할 수 있음. 실험 가능한 알고리즘은 아래와 같이 구현해 두었으며 각 알고리즘에 대한 간단한 설명도 기술하였음

2-1-1. Feature 추출 방식

detector_type	설명
ORB	(default) Oriented FAST and Rotated BRIEF로, OpenCV에서는 해리스 코너 검출을 기본값으로 함
AKAZE	Accelerated-KAZE로, KAZE와 마찬가지로 Nonlinear diffusion filter를 이용함
FAST	Features from accelerated segment test로, 특징점 추출 속도가 매우 빠름
KAZE	Nonlinear diffusion filter를 이용하여 비선형 스케일 스페이스를 구축하여 특징점을 검출함

2-1-2. Descriptor 방식

descriptor_type	설명
ORB	(default) BRIEF descriptor를 사용함
AKAZE	크기 및 회전 변화에 강하고 이진 기술자이기 때문에 계산이 빠름
BRISK	Binary Robust Invariant Scalable Keypoints로, 크기 및 회전 변화에 강하고 이진 기술자이기 때문에 계산이 빠름
KAZE	크기 및 회전 변화에 강하지만 실수 기술자이기 때문에 계산이 오래 걸림

2-1-3. 실행 영상

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/4ce688bb-2b20-4b06-86b2-9877ea7c3f4d/feature_detector_ORB.mp4

ORB

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/dccd78ba-c156-4943-90de-fc0453d63054/feature_detector_ORB_with_WARNING.mp4

ORB with WARNING

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/16f0b176-6083-401d-a8af-ff887cfe6adf/feature_detector_FAST.mp4

FAST

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/556bbbaa-d924-4b7a-822f-6a5e2f002900/feature_detector_AKAZE.mp4

AKAZE

2-2. Frontend

Frontend에서는 Stereo Camera Data에서 받아온 왼쪽과 오른쪽 이미지를 서로 비교하여 matching을 찾는 방식을 실험할 수 있으며, 크게 세 가지로 분류할 수 있음. (1) Matching 방식 (좌우 이미지에서 서로 대응하는 점을 찾기)과 (2) Matching을 찾기 위한 Homography 사용 여부 (대응하는 점 간의 대응 정확도를 판단), 그리고 (3) Homography를 사용할 시 선택할 수 있는 파라미터임.

2-2-1. Matching 방식

use_opencv_match	설명

select_descriptor_matcher	설명

use_opencv_match	설명
true	OpenCV를 사용하여 matching을 시행함
false	기존 ProSLAM 방식을 이용해 matching을 시행함 (왼쪽과 오른쪽 점의 row/column 위치 비교)

select_descriptor_matcher	설명
FLANNBASED	(default) 모든 디스크립터를 찾지 않고, 서로 이웃하는 디스크립터끼리 비교를 하는 알고리즘.
BRUTEFORCE	queryDescriptors와 trainDescriptors를 하나하나 확인해 매칭되는지 판단하는 알고리즘
BRUTEFORCE_L1	BRUTEFORCE의 유클리드 거리 측정 방법을 이용, = sigma(abs(a-b))
BRUTEFORCE_HAMMING	BRUTEFORCE의 해밍 거리 측정 방법을 이용, = sigma(a==b ? 1: 0)
BRUTEFORCE_HAMMINGLUT	BRUTEFORCE의 해밍 거리 측정 방법을 이용, XOR기법을 사용함.
BRUTEFORCE_SL2	BRUTEFORCE의 해밍 거리 측정 방법을 이용, root를 씌워서 값을 비교함.

2-2-2. Homogrphy 사용 여부

matching_disable_findhomography	설명
true	lowe's ratio 방식을 사용함
false	OpenCV의 `findHomography()` 함수를 사용함 lowe's ratio 방식을 사용함

양쪽 이미지에서 matching을 찾았다면 각 포인트 쌍에 대해 좋은 matching인지, 즉 잘 matching되었는지를 확인함.

2-2-3. Homography 파라미터

findhomography_method	설명
RANSAC	(default) 즉 가장 많은 수의 데이터들로부터 지지를 받는 모델을 선택하는 방법 - 랜덤으로 포인트를 잡아서 모델을 만듦.
RHO	PROSAC, RANSAC에서 개선한 모델. - 피쳐 디스크립터간 디스턴스를 정렬하여 반복을 할 때 마다 디스턴스가 낮은 매쳐들을 보다가 점점 높은 매쳐들을 봄. 디스턴스가 적은 매칭일수록 모델 추론때 더 정확히 추론될 가능성(inlier 가능성)을 볼 수 있음.
LMEDS	에러 분포들의 중앙값(에러 분포들을 순서대로 정렬했을 때 순서상 가운데에 있는 값)이 최소화되는 모델을 찾는다.

`matching_disable_findhomography` = `true`일 경우, 어떤 방식을 사용할지 결정함.

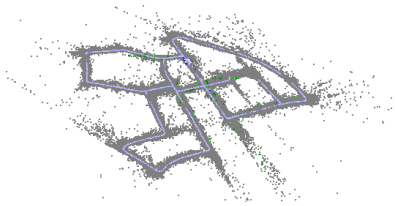
`RANSAC`, `RHO`를 선택할 경우, `findHomography()` 함수의 파라미터로서 설정 파일에서 [maximum_ransac_reproject](RANSAC 재투영 허용 개수), `maximum_iters`(iteration 횟수 (RANSAC)), `maximum_confidence`(정확도)를 세부 조정을 할 수 있음

2-3. Backend & Loop Closure

2-3-1. Loop Closure

aligner_type	설명
ICP(Classic-ICP)	(default) 두 cloud point를 겹치게 만드는 translation과 rotation을 iterative하게 구하는 방법
FAST-ICP	ICP를 fixed-point iteration 문제로 다루어 Anderson Acceleration을 적용하여 수렴을 가속하는 방법

option_disable_relocalization	설명
true	Loop Closure 기능이 off되며 pose-graph optimization을 수행하지 않음



option_disable_relocalization	설명
false	Loop Closure 기능이 on되며 다음과 같은 옵션을 선택할 수 있음

aligner's parameter	설명
error_delta_for_convergence	수렴으로 판단하기 위한 iteration간 error의 차이
maximum_error_kernel	inlier와 outlier를 판단하기 위한 커널의 값
maximum_number_of_iterations	최대 iteration 횟수
minimum_number_of_inliers	유효한 closure인지 판단하기 위한 최소 inlier 수
minimum_inlier_ratio	유효한 closure인지 판단하기 위한 최소 inlier 비율
anderson_m	Anderson Acceleration 기법을 사용하는데, 이때 과거 iteration의 정보를 얼마나 사용할지에 대한 파라미터

2-3-2. Pose-Graph Optimization

optimization_algorithm	설명
GAUSS_NEWTON	(default) 비선형 최소자승문제를 해결하기 위한 알고리즘으로 에러의 오차 제곱을 최소화하는 방법이다.
LEVENBERG	Gauss-Newton 방법에서 Gradient descent를 결합한 방법
DOGLEG	Levenberg 방법에서 trust region을 도입한 방법

linear_solver_type	설명
CHOLMOD	(default) sparse 숄레스키 분해(Cholesky decomposition)를 이용
CSPARSE	Csparse library에서 구현된 solver를 이용
DENSE	dense 숄레스키 분해(Cholesky decomposition)를 이용

▼ Refactoring

```
//aligner_type을 추가하여 relocalizer가 다른 여러 local map aligner를 받을 수 있게 만듦
if (_parameters->aligner_type == "ICP")
  _aligner = XYZAlignerPtr(new XYZAligner(_parameters->aligner));
else if (_parameters->aligner_type == "FAST-ICP")
  _aligner = FastAlignerPtr(new FastAligner(_parameters->aligner));
else{
  _parameters->aligner_type = "ICP";
  LOG_INFO(std::cerr << "Relocalizer::aligner type | invalid aligner_type, force to use ICP" << std::endl)
  _aligner = XYZAlignerPtr(new XYZAligner(_parameters->aligner));
}
LOG_INFO(std::cerr << "Relocalizer::aligner type | " << _parameters->aligner_type << std::endl)
```

```
// 기존에 local map aligner를 가리키는 포인터를 다른 포인터에도 적용하기 위해 XYZAligner의 상위 클래스인 BaseLocalMapAligner에 AlignerPtr을 정의
typedef std::shared_ptr<XYZAligner> XYZAlignerPtr;
// --->
typedef std::shared_ptr<BaseLocalMapAligner> AlignerPtr;
```

2-4. 성능 비교(1): 기존 ProSLAM과의 비교

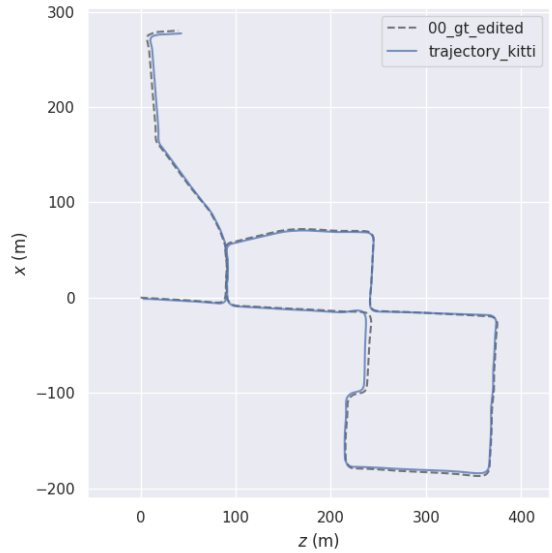
> 타 SLAM과의 비교는 [3-1. 타 SLAM과의 성능 비교]에서 이어짐

> **변경 내용**
> *** 이미지 처리:**
> detector_type: `FAST`
> *** Frontend:**
> use_opencv_match: `true` → select_descriptor_matcher: `FLANNBASED`
> matching_disable_findhomography: `false` → findhomography_method: `LMEDS`

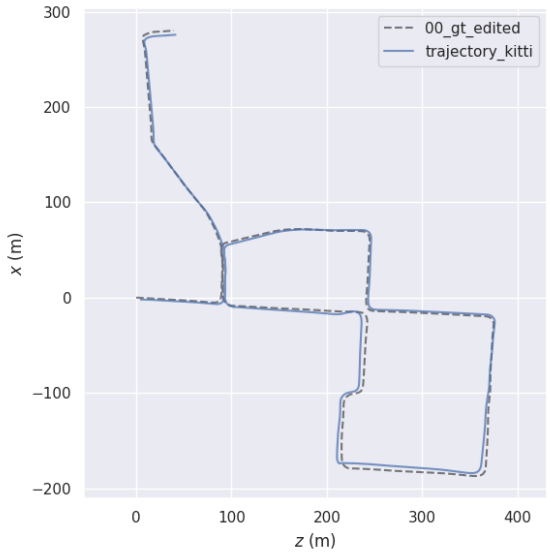
2-4-1. 정확도

> **점선**: Ground Truth (실제) / **실선**: SLAM 결과 경로

기존 ProSLAM



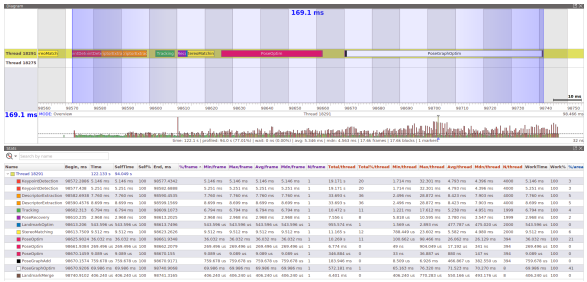
모듈러 아키텍처



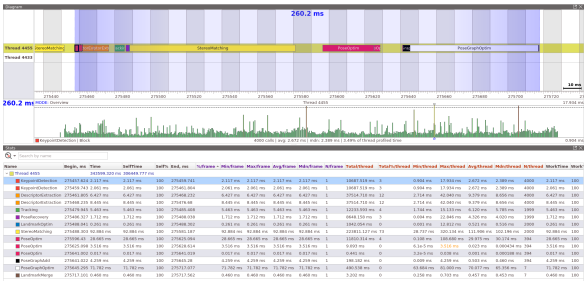
2-4-2. 속도

> 다양한 연산이 수행되는 한 프레임의 내용을 보임

기존 ProSLAM

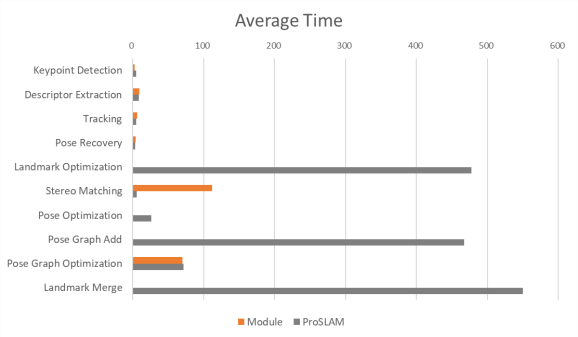


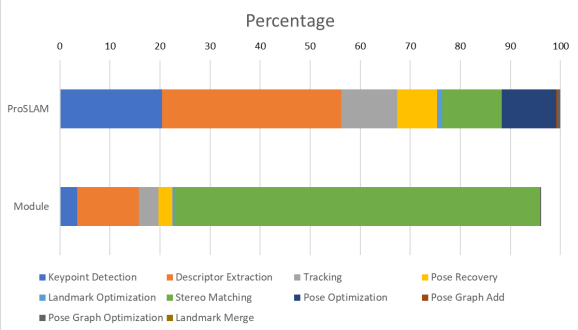
모듈러 아키텍처



block	calls	avg (ms)	mdn (ms)	block	calls	avg (ms)	mdn (ms)
Keypoint Detection	4000	4.794	4.396	Keypoint Detection	4000	2.672	2.2389
Descriptor Extraction	4000	8.423	7.903	Descriptor Extraction	4000	9.379	8.656
Tracking	1999	5.238	4.951	Tracking	1999	6.120	5.785
Pose Recovery	1999	3.780	3.547	Pose Recovery	1999	4.326	4.020
Landmark Optimization	2000	477.787	475.020	Landmark Optimization	2000	0.521	0.516
Stereo Matching	2000	5.582	4.980	Stereo Matching	2000	111.906	102.196
Pose Optimization	394	26.062	26.129	Pose Optimization	2	0.423	0.316
Pose Graph Add	394	466.867	382.550	Pose Graph Add	394	0.503	0.460
Pose Graph Optimization	8	71.523	70.270	Pose Graph Optimization	7	70.077	65.356

block	calls	avg (ms)	mdn (ms)	block	calls	avg (ms)	mdn (ms)
Landmark Merge	8	550.166	493.176	Landmark Merge	7	0.457	0.453





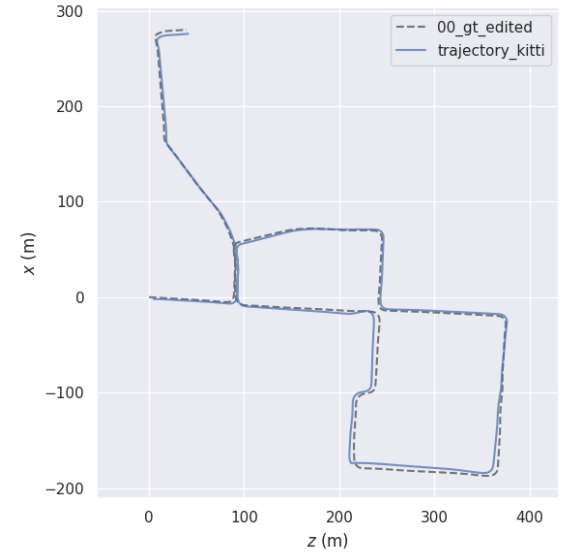
2-5. 성능 비교(2): 파라미터 변경 결과 비교

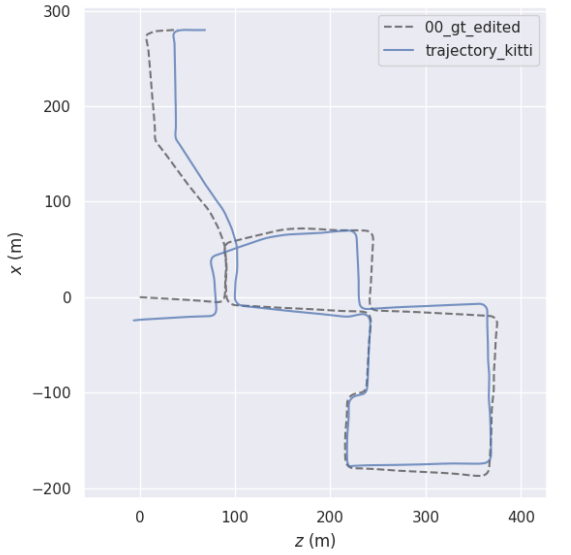
이전 설정

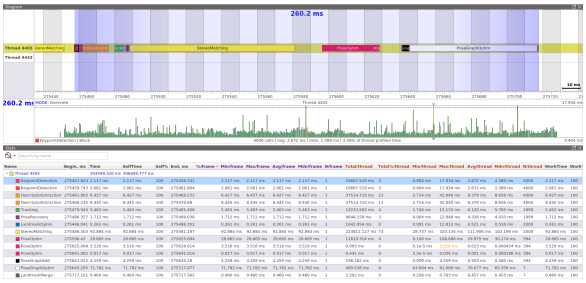
> aligner_type : `ICP`
> optimization_algorithm: `GAUSS_NEWTON`

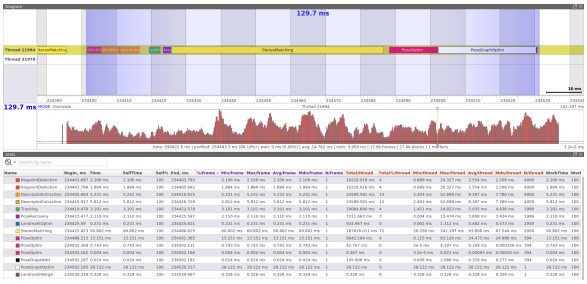
Backend 방식 변경

> aligner_type : `FAST-ICP`
> optimization_algorithm: `DOGLEG`







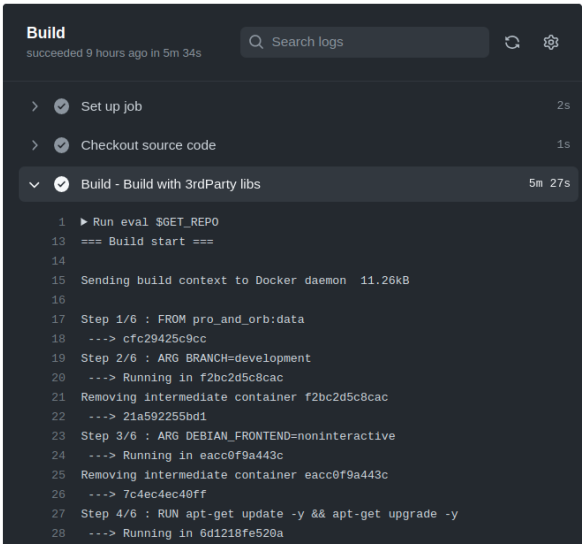


block	calls	avg (ms)	mdn (ms)	block	calls	avg (ms)	mdn (ms)
Keypoint Detection	4000	2.672	2.2389	Keypoint Detection	4000	2.554	2.206

block	calls	avg (ms)	mdn (ms)	block	calls	avg (ms)	mdn (ms)
Descriptor Extraction	4000	9.379	8.656	Descriptor Extraction	4000	8.397	7.780
Tracking	1999	6.120	5.785	Tracking	1999	5.035	4.836
Pose Recovery	1999	4.326	4.020	Pose Recovery	1999	3.608	3.434
Landmark Optimization	2000	0.521	0.516	Landmark Optimization	2000	0.462	0.473
Stereo Matching	2000	111.906	102.196	Stereo Matching	2000	93.808	87.544
Pose Optimization	2	0.423	0.316	Pose Optimization	2	0.347	0.345
Pose Graph Add	394	0.503	0.460	Pose Graph Add	394	0.370	0.277
Pose Graph Optimization	7	70.077	65.356	Pose Graph Optimization	1	28.122	28.122
Landmark Merge	7	0.457	0.453	Landmark Merge	1	0.328	0.328

2-6. CI/CD

GitHub의 main branch로 Push하거나 Pull Request할 때 자동적으로 Build를 실행함



추가적 Actions로는 아래와 같은 것들이 있음

Workflow	Trigger	Description
Build Base docker image	workflow_dispatch (수동 수행)	KITTI dataset을 제외한 ProSLAM 모듈러 이미지를 만듦
Build Platform	workflow_dispatch (수동 수행)	ARM64에서 빌드함 (실패)
Run	workflow_dispatch (수동 수행)	도커 이미지를 도커 컨테이너에서 실행함 (실패)
Test	workflow_dispatch (수동 수행)	Unit Test를 작동시킴 (절반의 성공)
clang-format	workflow_dispatch (수동 수행)	모든 소스코드를 clang format으로 전환함 (사용하지 않음)

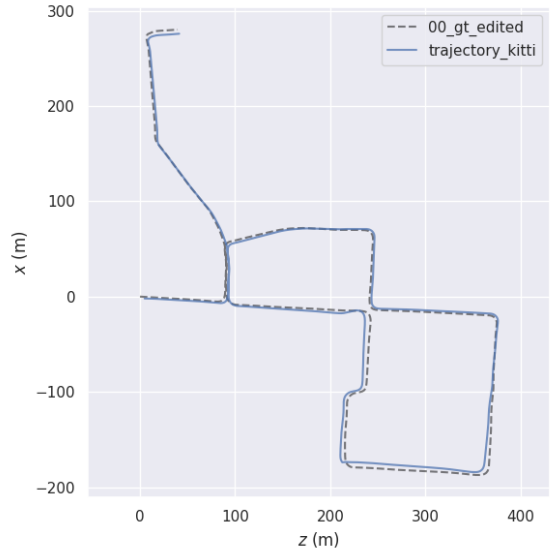
3. 추가적 결과 (갑작스러운 요구사항)

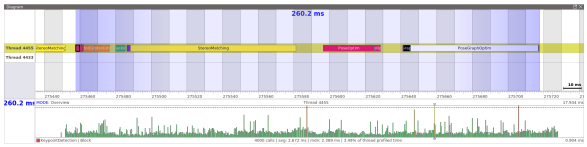
3-1. 타 SLAM과의 성능 비교

3-1-1. ORB SLAM2

- 한 개의 Thread에서 동작하는 ProSLAM과는 다르게, ORB SLAM2는 Local Mapping과 Tracking, Loop Closure가 각각의 Thread에서 실행됨
- 동일한 KITTI Dataset(00)에서 2000 frame에 대해 SLAM을 수행한 결과 아래와 같은 차이를 보임

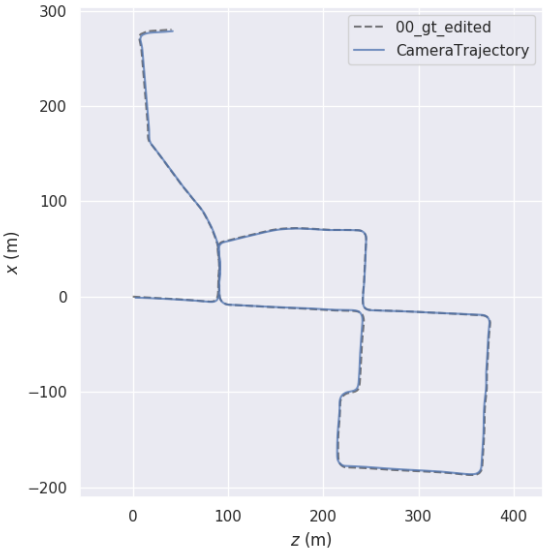
ProSLAM (모듈러 아키텍처)

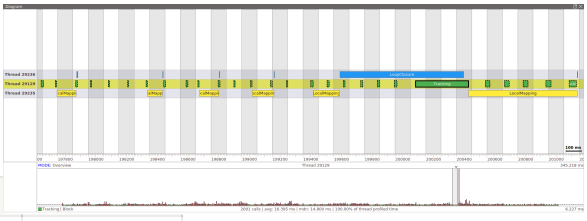




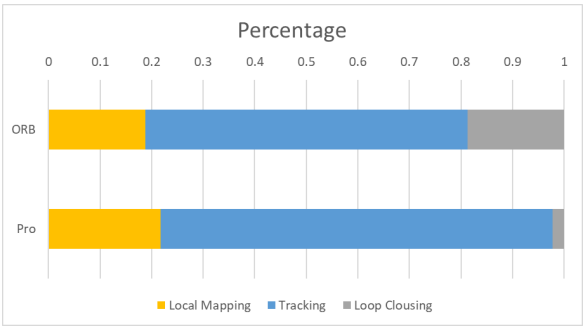
block	calls	avg (ms)	mdn (ms)
Keypoint Detection	4000	2.672	2.2389
Descriptor Extraction	4000	9.379	8.656
Tracking	1999	6.120	5.785
Pose Recovery	1999	4.326	4.020
Landmark Optimization	2000	0.521	0.516
Stereo Matching	2000	111.906	102.196
Pose Optimization	2	0.423	0.316
Pose Graph Add	394	0.503	0.460
Pose Graph Optimization	7	70.077	65.356
Landmark Merge	7	0.457	0.453

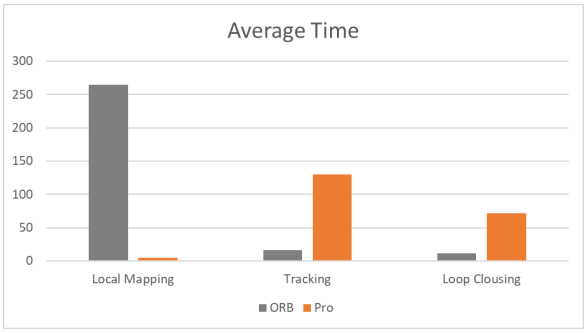
ORB SLAM2





block	calls	avg (ms)	mdn (ms)
Local Mapping	601	264.266	255.698
Tracking	2001	16.395	14.8
Loop Closure	600	11.372	9.075

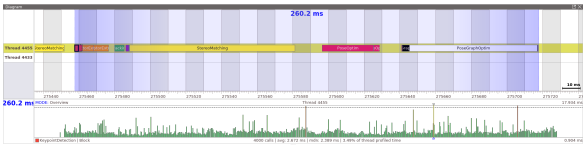




3-1-2. LSD SLAM

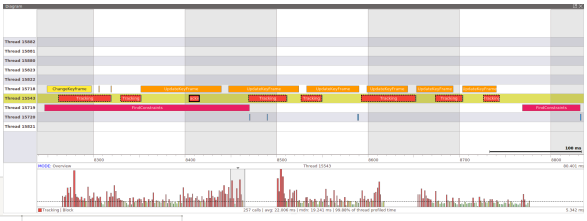
- 한 개의 Thread에서 동작하는 ProSLAM과는 다르게, LSD SLAM은 Tracking, Mapping, Constraint Search, Optimization이 각각의 Thread에서 실행됨
- LSD SLAM은 Tracking을 Lost할 경우 모든 Thread를 수동으로 재시작함. 따라서 아래 Profiler 결과에서도 Thread가 여러 개로 나타남. 그러나 각 Thread의 종류와 개수는 동일하게 유지됨.
- LSD SLAM에서는 SLAM 결과의 Pose 정보가 저장되지 않아 경로 정확도 결과는 비교할 수 없없음
- 동일한 KITTI Dataset(00)에서 2000 frame에 대해 SLAM을 수행한 결과 아래와 같은 차이를 보임

ProSLAM (모듈러 아키텍처)



block	calls	avg (ms)	mdn (ms)	
Keypoint Detection	4000	2.672	2.2389	2.40
Descriptor Extraction	4000	9.379	8.656	12.24
Tracking	1999	6.120	5.785	3.99
Pose Recovery	1999	4.326	4.020	2.62
Landmark Optimization	2000	0.521	0.516	0.34
Stereo Matching	2000	111.906	102.196	73.03
Pose Optimization	2	0.423	0.316	0.00
Pose Graph Add	394	0.503	0.460	0.06
Pose Graph Optimization	7	70.077	65.356	0.16
Landmark Merge	7	0.457	0.453	0.00

LSD SLAM



block	calls	avg (ms)	mdn (ms)
Update KeyFrame	158	22.945	25.712
Change KeyFrame	13	47.544	45.796
Tracking	257	22.006	19.241
Map Estimation	255	27.084	27.891
Find Constraints	14	80.842	69.093
Optimization	61	433.345	242.261

각 SLAM 별 기능 비교

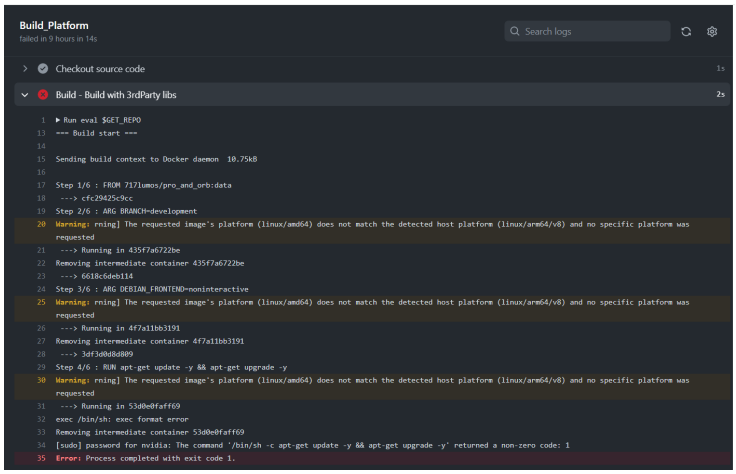
구분 (ProSLAM 논문 기준)	ProSLAM	ORB SLAM2	LSD SLAM
Triangulation	Keypoint Detection Descriptor Extraction Stereo Maching	Tracking	Tracking
Incremental Motion Estimation	Tracking	Tracking	Tracking
Map Management	Pose Recovery Landmark Optimization	Local Mapping	Map Estimation Update KeyFrame

구분 (ProSLAM 논문 기준)	ProSLAM	ORB SLAM2	LSD SLAM
Relocalization	Pose Optimization Pose Graph Add Pose Graph Optimization Landmark Merge	Loop Closure	Optimization Find Constraints Change KeyFrame

3-1-3. RTap map

빌드 및 Docker images 제작에는 성공하였으나, 각 모듈 별 해석을 시간 관계상 진행하지 못해 성능 비교를 진행하지 못함.

3-2. 각 SLAM의 Xycar에서의 CI 실행 및 작동



Xycar의 Jetson TX2의 경우 플랫폼이 ARM64/v8이나,
작업이 진행되어 Docker 이미지를 만들었던 플랫폼은
AMD로 상이했음.

따라서 Docker Container가 작동하지 못함.

직접 Xycar에서 모든 라이브러리와 소스코드를 빌드하거나,
처음부터 Docker image를 다시 만들어야 했으나,
시간 부족으로 진행하지 못함.

4. 기본 및 보너스 포인트 정산

4-1. 기본 목표

- [x] ~~프로젝트 종료 시기에 맞춰 정해진 개발 목표 달성시키기 **(기술개발) (+150)**~~
- [x] ~~프로젝트 종료 시기에 맞춰 개발 내용 및 사용한 프레임워크의 알고리즘에 대해 발표 **(이론 이해) (+150)**~~
- [x] ~~중간 점검 시기에 맞춰 개발 중인 내용 보고 (프로젝트 매니징) **(+150)**~~

4-2. 보너스 포인트

Aa 이름	# 점수	달성 여부	비고
KITTI 데이터셋에서 돌 수 있게 프레임워크 개량	50	☑	
PC 웹캠/리얼센스 카메라로 실시간 데모가 가능하게 함	100	☐	기존 ProSLAM 모델에 존재함
자이카에서 돌 수 있게 프레임워크 개량	100	☐	시도했으나 실패
정확도 개선	50	☐	
속도 개선	50	☑	
오프라인 시각화 가능	50	☑	
실시간 시각화 가능	100	☑	
아키텍처/알고리즘 재사용성 개선	100	☑	이미지 처리 ~ 백엔드 선택적 구동 가능
안정성 확보: CI/CD 유닛테스트	100	☑	시도했으나 실패
다른 팀에게도 도움이 될 수 있는 자료 정리 및 공유	50	☑	
오픈소스를 참고해 직접 VSLAM 파이프라인을 설계 및 구현(최소 2 모듈 이상 변경)	150	☑	ProSLAM의 feature detection, aligner 등 변경

Aa 이름	# 점수	달성 여부	비고
고객의 갑작스러운 요구사항1 달성	100	☑	ORB SLAM, LSD SLAM 2개는 만족, RTAB Map은 실패
고객의 갑작스러운 요구사항2 달성	200	☐	시도했으나 실패

450 + 750