Experiences with Building Domain-Specific Compilation Plugins in Graal

ManLang'17, 28 Sep 2017

Colin Barrett Christos Kotselidis Foivos S. Zakkak Nikos Foutris
Mikel Luján

MANCHESTER
1824
The University of Manchester

Is there a way to create domain-specific compiler optimizations without having to learn the whole compilation stack?

Yes! Modular JIT compilers (e.g. Graal)

Is there a way to create domain-specific compiler optimizations without having to learn the whole compilation stack?

Yes! Modular JIT compilers (e.g. Graal)

Introduction

- Computer vision applications becoming mainstream (e.g. autonomous vehicles, virtual reality)
- Both on embedded and desktop environments
- Ongoing effort to:
  - Increase accuracy
  - Optimize performance

Background

Simultaneous **L**ocalization **A**nd **M**apping (SLAM) Applications

Input

Stream of frames from cameras moving in an unknown environment

Output

- 3D reconstruction of environment
- Cameras' location in the environment
- Absolute positions of objects in the environment

Background

Simultaneous **L**ocalization **A**nd **M**apping (SLAM) Applications

Input

Stream of frames from cameras moving in an unknown environment

Output

- 3D reconstruction of environment
- Cameras' location in the environment
- Absolute positions of objects in the environment

Background

Simultaneous **L**ocalization **A**nd **M**apping (SLAM) Applications

Input

Stream of frames from cameras moving in an unknown environment

Output

- 3D reconstruction of environment
- Cameras' location in the environment
- Absolute positions of objects in the environment

Our case

Large-**S**cale **D**irect monocular SLAM (LSD-SLAM)

- Monocular: uses a single camera for input
- Non feature-based, operates on image densities
- Uses pose-graphs

Pose-graph

A graph where:

- nodes are frames
- directed edges contain the transformations (rotation, scaling, and translation) and the corresponding covariance matrix from the previous frame

Our case

Large-**S**cale **D**irect monocular SLAM (LSD-SLAM)

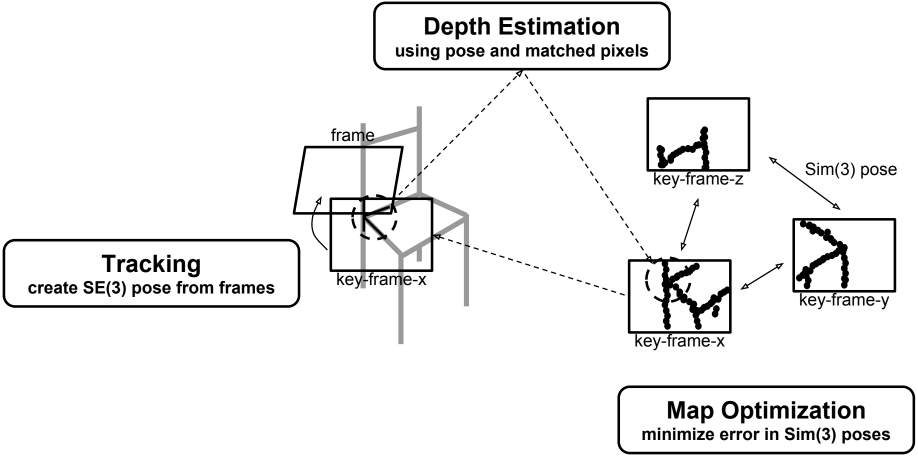
- Monocular: uses a single camera for input
- Non feature-based, operates on image densities
- Uses pose-graphs

Pose-graph

A graph where:

- nodes are frames
- directed edges contain the transformations (rotation, scaling, and translation) and the corresponding covariance matrix from the previous frame

LSD-SLAM overview

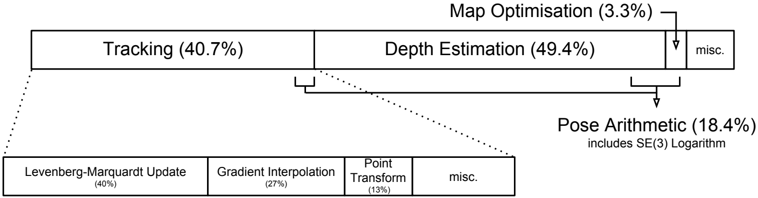


Depth Estimation
using pose and matched pixels

Tracking
create SE(3) pose from frames

Map Optimization
minimize error in Sim(3) poses

LSD-SLAM breakdown



Map Optimisation (3.3%)

Tracking (40.7%)

Depth Estimation (49.4%)

misc.

Pose Arithmetic (18.4%)
includes SE(3) Logarithm

Levenberg-Marquardt Update (40%)

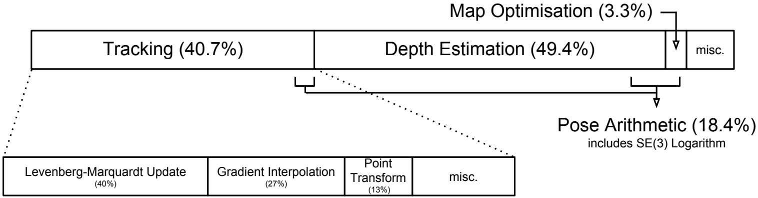
Gradient Interpolation (27%)

Point Transform (13%)

misc.

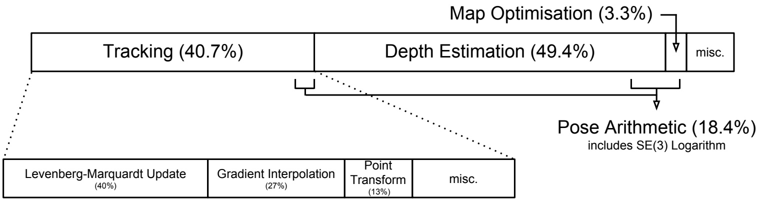
Framework	Point Trans.	SE(3) Log.	Gradient Inter.	L-M Update
	mean (ns)	*mean (ns)*	*mean (ns)*	*mean (ns)*
Eigen (C++)	13.342	131.138	9.847	152.376
EJML (Java)	77.411	415.924	84.479	308.412
JEigen (JNI)	1356.498	1671.105	58.961	895.845

LSD-SLAM breakdown



Framework	Point Trans.	SE(3) Log.	Gradient Inter.	L-M Update
	mean (ns)	*mean (ns)*	*mean (ns)*	*mean (ns)*
Eigen (C++)	13.342	131.138	9.847	152.376
EJML (Java)	77.411	415.924	84.479	308.412
JEigen (JNI)	1356.498	1671.105	58.961	895.845

LSD-SLAM breakdown



Framework	Point Trans. *mean (ns)*	SE(3) Log. *mean (ns)*	Gradient Inter. *mean (ns)*	L-M Update *mean (ns)*
Eigen (C++)	13.342	131.138	9.847	152.376
EJML (Java)	77.411	415.924	84.479	308.412
JEigen (JNI)	1356.498	1671.105	58.961	895.845

Performance Characterization

JIT compiler generated code worse than the hand-tuned Eigen

- JIT compiler fails to inline some methods in the critical path
- Opportunities for constant folding and sub-expression elimination are missed
- No SIMD

Indigo: Our Approach

A small vector and matrix library

- Up to 8 elements and 8x8 cells

Accompanied by a Graal plugin

- Force inline methods of the library
- Custom register allocation
- SIMD backend

Encapsulated and immutable

- Reduces object allocation
- Reduces memory indirection
- Enables constant folding
- Enhances common sub-expression elimination

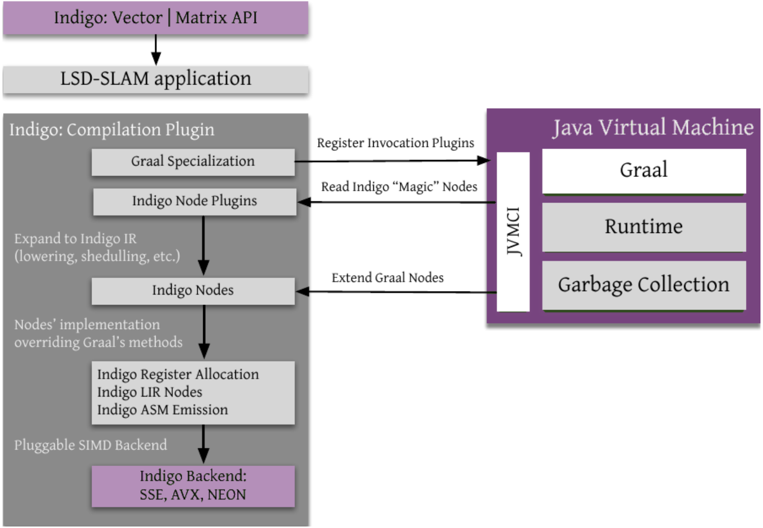
Why a new backend and register allocator?

1. There is no publicly accessible SIMD assembler in Graal
2. The JVM does not support SIMD registers
3. The JVM cannot handle SIMD registers during register spillage

Indigo: Assumptions for SIMD acceleration

- Hardware supports 128-bit vector operations
- Indigo's classes/subclasses contain single-precision floating point numbers suitable for vector operations in SLAM
- Unused elements of a vector are zero
- The elements of a vector are contiguous in memory
- Once constructed, a vector is immutable

Indigo Compilation Plugin Outline



Methodology

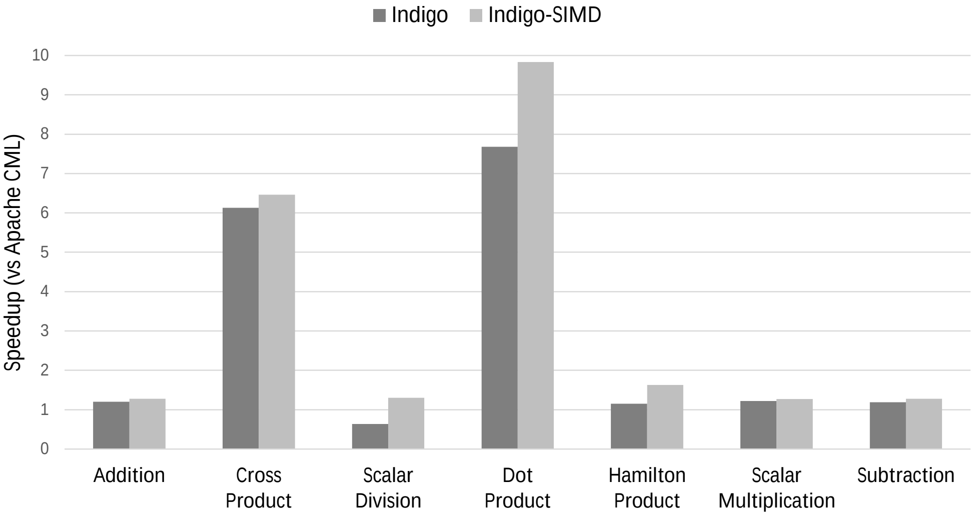
Comparison

1. Indigo vs Apache CML
   as a generic small vectors and matrices Java library
2. Indigo vs Eigen
   as a SLAM specific library

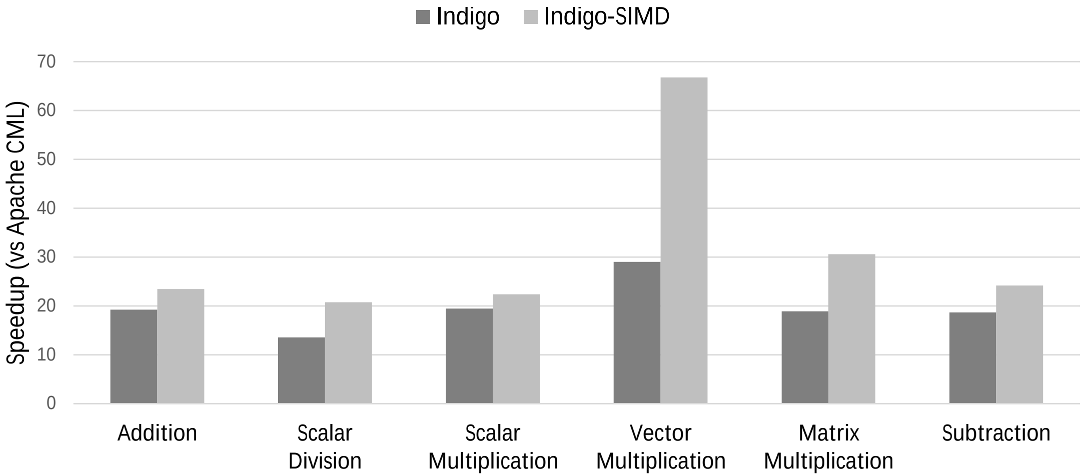
Evaluation Setup

Hardware	
Processor	Intel Core i7 4770 3.4GHz
Cores	4
Hardware threads	8
Main memory	16GB
Vector Units	SSE 4.2 and AVX2
Software	
OS	Windows 8.1
C++ compiler	MSVC 17.00.61030 (x64)
JVM	Java SE 1.8.0_72 64-Bit JVMCI VM
Baseline	Apache CML 3.6

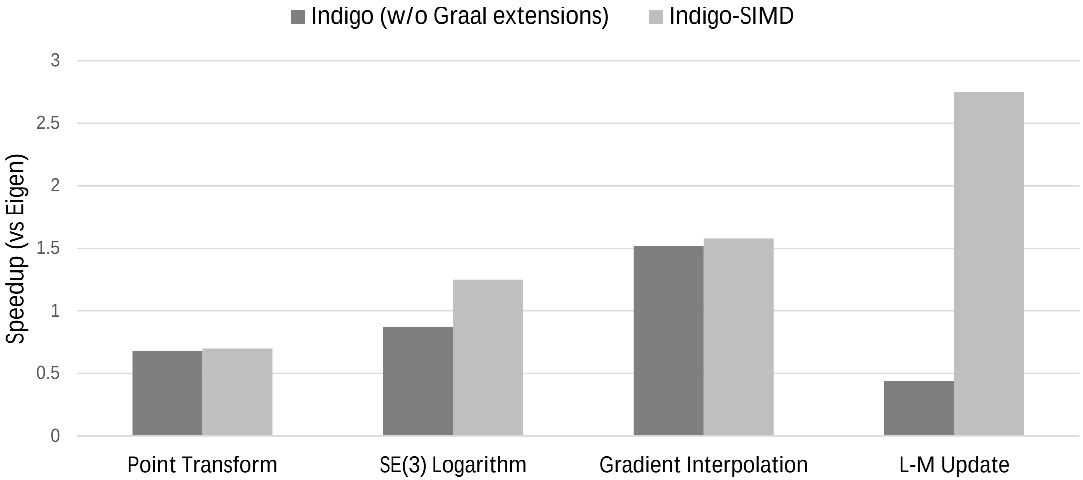
Indigo vs Apache CML: Vector Operations



Indigo vs Apache CML: Matrix Operations



Indigo vs Eigen: SLAM kernels



Conclusions

- Domain-specific optimizations have significant impact on the performance of domain-specific applications
- Modular JIT compilers like Graal ease such optimizations through plugins
- Indigo demonstrates that SLAM applications written in Java can be significantly optimized using this approach

Thank You!

Experiences with Building Domain-Specific Compilation Plugins in Graal

ManLang'17, 28 Sep 2017

Colin Barrett   Christos Kotselidis   Foivos S. Zakkak   Nikos Foutris
Mikel Luján

MANCHESTER
1824
The University of Manchester