Experiences with Building Domain-Specific Compilation Plugins in Graal

Colin Barrett, Christos Kotselidis, Foivos S. Zakkak, Nikos Foutris, and Mikel Luján
Advanced Processor Technologies Group
School of Computer Science
The University of Manchester
Kilburn Building, Oxford Road
Manchester, UK
M13 9PL
first.last@manchester.ac.uk

ABSTRACT

In this paper, we describe our experiences in co-designing a domain-specific compilation stack. Our motivation stems from the missed optimization opportunities we observed while implementing a computer vision library in Java. To tackle the performance shortcomings, we developed **Indigo**, a computer vision API co-designed with a compilation plugin for optimizing computer vision applications.

Indigo exploits the extensible nature of the Graal compiler which provides invocation plugins, that replace methods with dedicated nodes, and generates machine code compatible with both the Java Virtual Machine (JVM) and the SIMD hardware unit. Our approach improves performance by up to 66.75x when compared to pure Java implementations and by up to 2.75x when compared to the original C++ implementation. These performance improvements are the result of low-level concurrency, idiomatic implementation of algorithms, and by keeping temporary objects in the wider vector unit registers.

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; *Domain specific languages*; • **Computing methodologies** → *Computer vision*;

KEYWORDS

SLAM, Graal, Compiler, JIT, Java

ACM Reference format:
Colin Barrett, Christos Kotselidis, Foivos S. Zakkak, Nikos Foutris, and Mikel Luján. 2017. Experiences with Building Domain-Specific Compilation Plugins in Graal. In *Proceedings of ManLang 2017, Prague,Czech Republic, September 27–29, 2017,* 12 pages.
https://doi.org/10.1145/3132190.3132207

1 INTRODUCTION

The advent of highly modular and efficient compilers [8, 23] creates new opportunities in introducing a number of aggressive optimizations. For example, Graal [8] along with Truffle [45] enable the implementation of multiple languages such as Ruby [39] and R [40] on top of a unified compilation and runtime stack served by Graal. In addition, the recent introduction of the JVM Compiler Interface (JVMCI) [20] enables the exposure of core VM components into the compiler stack seamlessly.

All the above result in compiler and runtime stacks that are more modular and efficient than monolithic compilers and virtual machines (VMs). Although such systems are high-performing, the level of optimization they perform is still bound by the fact that they need to work for every case; i.e., the compilation results need to adhere to the programming language specification and the runtime system must always be in a consistent state.

Currently, in the process of creating domain-specific optimizations, a compiler engineer must ensure that the optimization does not violate the language specifications. Therefore, a natural question risen is: Is there an easy way to create domain-specific optimizations without having to learn the whole compilation stack? In our case, that question formed while trying to implement a Java version of a computer vision algorithm.

Emerging applications from the computer vision domain are becoming mainstream in both the embedded and desktop systems' market [11]. Simultaneous Localization and Mapping (SLAM) applications aim to perform localization and mapping simultaneously for a sensor moving through an unknown environment [9, 10, 30]. SLAM algorithms are employed in all kinds of autonomous vehicles, like self-driving cars, drones, autonomous underwater vehicles, planetary rovers, etc. Nowadays, there is a great focus not only in analyzing the accuracy of such methods, but also on characterizing and systematically improving their performance [5].

In the process of creating a portable Java implementation of LSD-SLAM [9], a specific case of SLAM applications, we noticed some performance shortcomings related to the efficient execution of short vector types. We also noticed that we could achieve higher performance if we could instruct the compiler to force some optimizations even if it could not guarantee that they would be safe in the general case. Since we possess the domain-specific knowledge of the algorithm we are developing, we wanted to find a way to pass that semantic information down to the compiler and alter assumptions made while optimizing the code.

In particular, LSD-SLAM applications require vector abstractions that specialize for small vectors (two to seven elements) which can exploit SIMD instructions with idiomatic usages. To optimize such operations we create **Indigo** an API, for such short vectors,

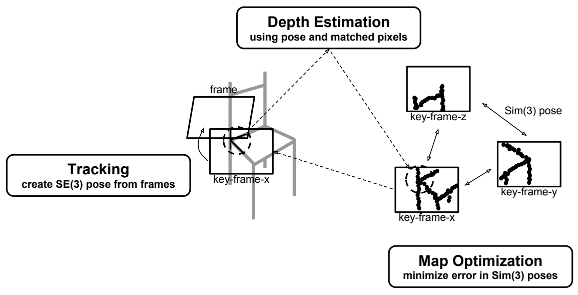

Figure 1: Overview of the LSD-SLAM application.

co-designed with *a lightweight compilation plugin for optimizing computer vision application specialized for classes of small vectors and matrices.* Our approach avoids algorithmic obfuscation through the short vectors API and allows the optimizing compiler to improve the execution time of the generated machine code through the co-designed plugin. Furthermore, we co-design vector and matrix operations in Java with the corresponding compiler optimizations in order to make them compatible with SIMD instructions. To achieve the above, we exploit Graal's *Invocation Plugin* mechanism which allows us to attach Indigo to the compilation process. Indigo is then able to extend Graal's intermediate representation (IR) to communicate the semantics of the abstraction through the compiler to the assembler.

In summary, the contributions of this paper are:

- We discuss compiler limitations regarding optimizing a computer vision library.
- We describe our experiences while building Indigo: a computer vision API co-designed with a plug-in for the Graal compiler to exploit SIMD instructions for commonly occurring computer vision data operations.
- We demonstrate that the optimized Java versions of the LSD-SLAM kernels can outperform commonly used Java libraries (by up to 66.75x). Furthermore, we showcase that Indigo, in most cases, outperforms the C++ implementations of the LSD-SLAM kernels by up to 2.75x.

The paper is organized as follows: Section 2 provides the details of the developed LSD-SLAM application and the shortcomings we encountered while implementing it in Java. Section 3 presents the Indigo API used for specializing matrix and vector operations of SLAM applications. Section 4 describes the Indigo compiler plugin, while Section 5 presents our performance evaluation. Finally, Sections 6 and 7 provide the related work and the conclusions respectively.

2 BACKGROUND

Simultaneous Location and Mapping (SLAM) is a key algorithm for robotics and other autonomous devices. SLAM applications, such as KinectFusion [30], SLAMBench [29], and Large-Scale Direct Monocular SLAM (LSD-SLAM) [9] perform a 3D reconstruction of an environment while tracking the location of a camera and calculating absolute positions of objects. Inputs include monocular cameras found in commodity web-cams, laser ranging sensors, or wide-angle time-of-flight cameras as found in the Microsoft Kinect [27]. SLAM applications are computationally demanding and often require state-of-the-art GPUs in order to perform the 3D reconstruction of the environment in real-time. Such algorithms enable robots and autonomous vehicles to *learn* the environment they move in and their position in it. That said, being able to run at real-time especially on less powerful devices than GPUs is highly desirable.

2.1 LSD-SLAM

In this work we focus on Large-Scale Direct Monocular (LSD) SLAM applications. LSD-SLAM in contrast to the feature-based SLAM applications, instead of using features (e.g., lines, edges, etc.), extracted by each captured frame, it operates directly on image intensities and creates point clouds. This approach enables LSD-SLAM to obtain a much denser map, since it is not limited on corners and straight line segments, as the feature-based algorithms.

LSD-SLAM, being monocular, takes input from a single camera and processes the images to reconstruct the 3D environment and estimate the relative position of the agent in the environment at real-time. LSD-SLAM models the environment as a pose-graph consisting of key frames with associated depth maps as nodes. A pose-graph is a graph where nodes are frames and directed edges contain the transformations (rotation, scaling, and translation) and the corresponding covariance matrix from the previous frame.

LSD-SLAM comprises three main components; tracking, depth estimation, and map optimization. Figure 1 illustrates the interaction of the LSD-SLAM components. The *tracking* component estimates the location of the camera, processed from an image (*frame*), against *key frames* in a graph. The key frames are used as a reference, the first being the initial image and, after that, images that are too far apart in space or tracking has failed against the previous key frame. LSD-SLAM calculates depth information using *depth estimation*, which compares the difference in position of pixels in two separate frames. Each key frame maintains its own depth information and is able to create a point cloud to visualize the environment. The last component is *map optimization* that minimizes errors in the *map* of the environment. This includes loop closure in which disjoint, but neighboring, key frames are associated in the graph. The three components run in parallel and the feedback from the depth estimation and map optimization is used to improve tracking and to update key frame information. In more detail:

Tracking. The tracking phase calculates the relative position (*pose*) between frames, which is used to estimate depth and to build the graph that models the environment map. The tracking task uses an initial pose (the result of the previous tracked frame) to transform and project each point in the point cloud of the current key frame to a pixel in the image. Then, for each point, the transformed point and the gradient of the pixel are combined to form a vector that represents a pose. Following this, a *residual* is calculated from the photometric error (difference in corresponding pixels) between the key frame and the current frame. Finally, the calculated pose and residual are used to construct a system that uses the Levenberg-Marquardt iterative algorithm [26] to find the best pose. Once the

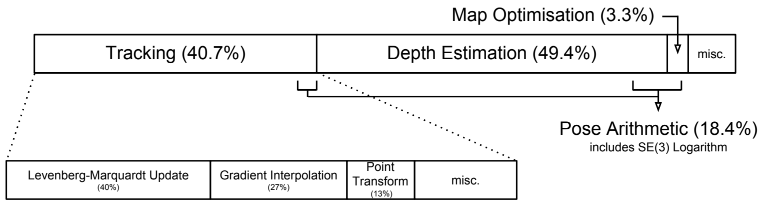

Figure 2: LSD-SLAM execution time breakdown.

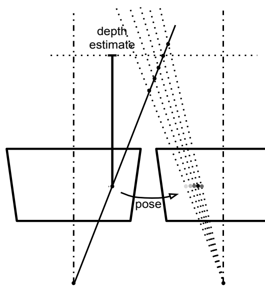

Figure 3: The depth estimation task in LSD-SLAM.

$$
\begin{aligned}
\xi &\in \mathfrak{se}(3) \\
\xi_{ki} &\equiv \xi_{kj} \circ \xi_{ji} \\
&\equiv \log(\exp(\xi_{kj}) \cdot \exp(\xi_{ji})) \\
E(\xi_{W_1} \dots \xi_{W_n}) &\equiv \sum_{(\xi_{ji},\Sigma_{ji})\in\varepsilon} \left(\xi_{ji} \circ \xi_{W_i}^{-1} \circ \xi_{W_j}\right)^T \Sigma_{ji}^{-1} \left(\xi_{ji} \circ \xi_{W_i}^{-1} \circ \xi_{W_j}\right) \\
Where:\ W_n &= world\ frame
\end{aligned}
$$

Figure 4: Cost-function of LSD-SLAM's map optimization.

best pose is found, the result is attached to the frame and then used in depth estimation.

Depth Estimation. Depth is computed using *stereo estimation*; using two frames and the pose as input to the algorithm. Figure 3 contains a simplified illustration of the process that is taking place. To achieve this, a point is selected from the point cloud of the key frame and the pose is used to estimate the equivalent point in the current frame. A vector is created representing the possible variation in depths for this second point. Points along with the length of the created vector are selected and during individual testing the best match is used to provide the depth. To achieve this, pixels around each point in both frames are compared and the pixel with the lowest photometric error (difference in pixel intensities) is selected. This process is computationally intensive because each point used has a unique vector based on the pose and the estimated depth of the point. The majority of the arithmetic in the implementation of the algorithm is in transforming and projecting points while testing whether they are within the bounds of the image. The depth information for each point in the key frame is used in the tracking algorithm, contributing to the residual.

Map Optimization. The mapping algorithm uses the pose information between key frames to track points representing the environment. The key frames, when created, are stored in a graph using the relative pose to the previous key frame as edges, acting as constraints during optimization. Over time the errors in tracking accumulate and may cause a duplication of points in the environment being mapped. Figure 4 contains the cost-function used to generate the error that is minimized during the optimization algorithm. It uses the pose concatenation operator ($\xi_{ij} \circ \xi_{jk}$ in Figure 4) that contains the mapping between Lie groups algebra [15] and the Hamilton product; two of the SLAM kernels. To reduce the amount of computation required, heuristics are implemented within LSD-SLAM to reduce the number of poses used in the cost-function.

2.2 SLAM Kernels

As described in the previous section, SLAM applications rely heavily on matrices and vectors. Vectors are used to represent points in the Euclidean space and also as a concise representation of poses with up to seven degrees of freedom. The vectors are translated, combined and normalized frequently in all phases of SLAM algorithms. Another use of vectors is in the quaternions, complex numbers with a real part and an imaginary vector which are used to encode rotations and scales. In addition, matrices are mainly used to store transformations.

In LSD-SLAM, and in SLAM applications in general, operations on matrices and vectors are frequent. The SLAM kernels mostly used in LSD-SLAM are presented below, while the contributions of each kernel to the total execution time of LSD-SLAM are illustrated in Figure 2.

Point Transform. Points are typically represented by three coordinates (x, y, z). However, there are cases where this is relative to the current key frame. When comparing pixels, the point is transformed and projected onto the frame image in order to determine the photometric error. This is achieved by rotating and translating the point and then projecting it with the camera model. The rotation is represented as a 3D matrix (or quaternion) while the translation is represented by a 3D vector. A point is first rotated with a matrix-vector multiplication resulting in a translation which is then added as an offset. A projection is a second matrix-vector multiplication which occurs during tracking and map optimization; each point of the point cloud is transformed in each iteration.

SE(3) Logarithm. An SE(3) pose is represented as a rotation and a translation. Since this is a complicated form used in linear algebra, the Lie group may be transformed to a vector with six elements. This is achieved by employing the logarithm, and thus, creating the SE(3) pose. It is a frequent operation, particularly in map optimization, within LSD-SLAM applications where it forms a central role in the cost-function.

Gradient Interpolation. Images are represented as a matrix of pixels; indexed by two co-ordinates (u, v) calculated during point transform in LSD-SLAM. The exact value is a real number so the options are to floor the values to use integer indices or to use sub-pixel resolution by applying interpolation. In the tracking algorithm of LSD-SLAM, used during the tracking and map optimization phases, the gradient and luminosity are interpolated to improve the precision. Convolution is used to obtain the result based on four vectors multiplied by coefficients derived from the fractional part of the co-ordinates.

Levenberg-Marquardt Update. LSD-SLAM uses the SE(3) pose estimation to solve the next estimate of the current frame relative to the key frame. The Levenberg-Marquardt algorithm [15] is used since it is a non-linear system that uses the partial differential of elements in the vector with photometric errors.

2.3 LSD-SLAM in Java

The vanilla implementation of LSD-SLAM contains hand-optimized code for two architectures [9, 38]; SSE intrinsics for x86 [16] and NEON assembly for ARM [24]. Furthermore, since SLAM applications are computationally intensive, they typically require hardware acceleration [28] on GPGPUs. Providing efficient implementations of SLAM applications for every GPU device available is a challenging task when implemented in statically compiled languages such as CUDA or OpenCL. In such languages, developers need to account for every possible configuration that the SLAM application might run on and include in their code base all the hand-tuned parameters suitable for each specific device; e.g., vector size, thread group sizes, parallelization schemes, etc.

Implementing algorithms without strict separation between abstraction and implementation impedes the ability to develop and evolve applications quickly, let alone dynamically. The motivation for an implementation of LSD-SLAM in Java is productivity and portability. Implementing SLAM applications in a VM-based language delegates all the decisions regarding hardware specific configurations to the runtime as showcased in [21, 22]. This way, we can achieve a single portable SLAM implementation that can run efficiently across many devices without requiring any code changes. To achieve this, VMs employ dynamic just-in-time (JIT) compilation which allows them to optimize the code at runtime.

Existing approaches that specialize in computer vision applications and matrices in Java use either library wrappers to existing compiled binaries or create pure Java libraries. However, as shown in Table 1, neither approach provides the performance of existing non-Java LSD-SLAM implementations. In Table 1, *Eigen* [12] is the original LSD-SLAM implementation written in C++ and optimized with SIMD instructions. Efficient Java Matrix Library (EJML) [2] is the basis for existing Java implementations of abstractions for computer vision, as GeoRegression [3], used by BoofCV [1]. EJML is used here to represent the pure Java SLAM libraries. *JEigen* [37] on the other hand represents the SLAM Java libraries implemented as wrappers to binary libraries using the Java native interface (JNI) and/or Java native access (JNA).

We observe that EJML is significantly slower than the original implementation, and JEigen is two orders of magnitude slower than the original implementation. We attribute EJMLs performance to the inability of the dynamic compiler to optimize the computation intensive parts to the same level as their hand-tuned counterparts. More details on this issue are provided in Section 3. Regarding JEigen's performance, we attribute it to the nature of JNI and JNA. The use of library wrappers through JNI or JNA comes with the cost of marshaling and un-marshaling of the data each time the execution transits from Java to *native code* and vice versa. Additionally, the contents of native methods cannot be inlined and optimized as part of the Intermediate Representation (IR) during compilation.

In this work we port LSD-SLAM to Java using pure Java libraries; we identify the dynamic compiler limitations and present a compiler plug-in that bypasses them and allows the compiler to perform equivalent, to the hand-tuned version, optimizations. The next section presents the specialized Java libraries, discusses the implementation decisions and their impact on performance, and identifies the dynamic compiler limitations.

3 INDIGO: SPECIALIZATION OF JAVA LIBRARIES

Specialization for computer vision libraries in Java is already available in the GeoRegression library [3] and is based on EJML [2]. EJML uses a `Matrix` interface, which it implements for various matrix and vector sizes. Each implementation is optimized depending on the size of the matrix or vector it represents. Small vectors are represented by classes with the data represented as public, non-final fields that allow efficient access without the indirection and boundary checks of Java arrays. However, our experience shows that the performance of such libraries is still not optimal (see Table 1). In this work we create a class collection for small vector (up to 8 elements) and matrix data types used in LSD-SLAM that is suitable for use in commonly observed SLAM algorithms. The design of our library increases the awareness of temporary objects and their values, thus reducing the interaction with the garbage collector and consequently improving performance.

Figure 5 presents the class hierarchy of the new library. The new abstractions introduce implementations for vectors of length four (`AbstractV128`) and eight (`AbstractV256`), as well as, for up to eight by eight matrices (`AbstractM256`). Larger vectors, used to store image data, use existing approaches with data abstraction based on matrices with meta-data in the form of properties. In our design we employ strict typing to increase the number of errors caught during compilation. For example, a point multiplied by a vector returns a point translated from the original while a vector multiplied by a vector is another vector. On the other hand, a point multiplied by a point has no meaning in SLAM applications and through strict typing can be prevented.

Table 1: Mean execution time and standard deviation for the SLAM kernels on different implementations.

Framework	**Point Transform**		**SE(3) Logarithm**		**Gradient Interpolation**		**L-M Update**	
	mean (ns)	*s.d.*	*mean (ns)*	*s.d.*	*mean (ns)*	*s.d.*	*mean (ns)*	*s.d.*
Eigen	13.342	0.128	131.138	3.046	9.847	0.309	152.376	2.789
EJML	77.411	8.383	415.924	8.450	84.479	1.277	308.412	5.648
JEigen	1356.498	38.164	1671.105	43.373	58.961	0.959	895.845	8.166

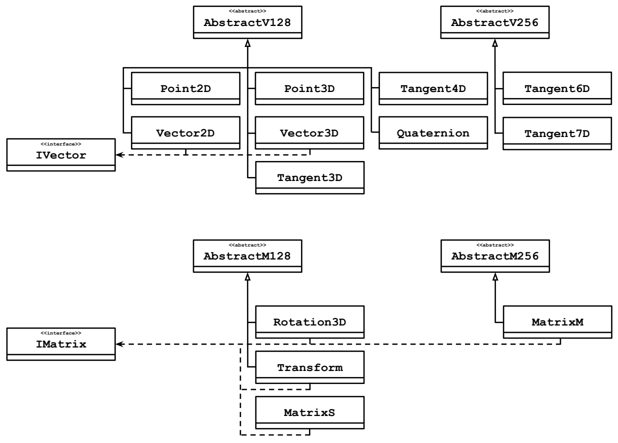

Figure 5: Class hierarchy of the new Java library.

3.1 Library Optimizations

To achieve increased performance we apply a number of optimizations on the library implementation. These optimizations enable the standard Graal compiler to better optimize the code at runtime. First, as in EJML we use fields instead of arrays to describe the small vectors. We further encapsulate (add `private` modifier) the fields and make them immutable (add `final` modifier). Encapsulation has no effect on the performance but allows for future revisions of the library without braking backwards compatibility. Note that mutable objects superficially improve performance by allowing operations, e.g. accumulation, without the need to instantiate a new object on the heap. However, the actual performance improvement is dependent on the scope of the object maintaining the new value. The machine code generated by the dynamic compiler will be the same if the value does not escape the scope of compilation and is thus placed on the stack instead of the heap. That said, the penalty of new object creation is eliminated in some cases, while enabling *constant folding*. This allows us to:

Reduce object allocation. An object encapsulating an array will hold at least one reference to an object on the heap (the array). There is also the likelihood that the vector also contains one or several objects to provide the necessary data abstraction. As each object is managed by the garbage collector there is an inherent cost in allocating memory for it (and later in its collection).

Reduce memory indirection. Each object reference is another level of indirection, the value of which is not fixed as it may move in physical memory. Therefore, it is not simply optimized by a base address and offset, the indirection must be followed each time. By expanding the array to primitive types, the elements of the vector become an offset from the object base address.

Enable constant folding. An array in Java is mutable; there is nothing preventing an array value from being modified. Our implementation enforces immutability, thus enabling constant folding which allows the propagation and replacement of variables with constant values in expressions, improving performance.

Enhance common sub-expression elimination. Establishing data dependencies when arrays are involved is not trivial and occasionally impossible. By eliminating the indirection and simplifying the values as primitive types, we enable existing optimization phases of the compiler to be more aggressive as primitive types are more predictable in their nature.

3.2 Compiler Limitations

Compilers perform several optimization passes where each pass may enable more optimizations. One of the most common enablers for further optimization is *inlining*. Due to the dynamic nature of Java, inlining of methods enables specialization and thus more optimizations. Not inlined methods cannot be optimized to the same extend since depending on the invocation context they might behave differently, especially in dynamic programming languages. In our case the main inhibitor in the dynamic compiler preventing the full inlining and optimization to use only registers in algorithms, as generated by C++, is the implementation of interfaces as these may be unknown at runtime. The best the compiler can do is to invoke a method directly but it is not possible to optimize across this boundary. As a result, opportunities for constant folding and sub-expression elimination are missed resulting in sub-optimal machine code generation. Although it is possible to write code that is efficiently compiled (e.g. manually inline some segments), it is not considered a good practice since functionality is obfuscated or specialized without due consideration for other uses.

Section 4 addresses inefficiencies arising from compiler sensitivity by implementing a compiler plugin, co-designed with our library, that maximizes the compiler's capabilities. Our approach also augments the dynamic compiler with semantics of small vectors allowing it to utilize hardware support for vector operations, further improving performance. These transformations result in dynamically generated machine code equivalent to that generated from the hand-optimized C++ libraries.

4 INDIGO: SPECIALIZATION OF DYNAMIC COMPILERS

In this section, we present the co-designed, with our library, modifications to the optimizer. The presented optimizations remove the overheads associated with the encapsulation and indirection of operations suitable for vectorization. We also extend the co-designed methodology to allow Java to utilize the vector unit for simple operations and also instruction sequences developed over time in the SIMD application research.

4.1 The Graal Compiler

The Graal compiler [8], currently integrated in JDK 9 through the JVM Compiler Interface (JVMCI) [20], is a highly modular and efficient compiler used not only for the compilation of Java programs but also for other dynamic programming languages running on top of Truffle [45]. Graal is written in Java and employs a hierarchical way of optimizing code through its tiers; High, Mid, and Low. Initially, an Intermediate Representation (IR) graph is created by parsing the bytecodes from a class file. The Graal IR contains both control and data dependencies and utilizes the method as its level of compilation abstraction. Within Graal, the IR is maintained as a structured graph with nodes representing actions or values while edges represent their dependencies. Consequently, the generated IR graph is being optimized and *lowered* iteratively until final machine code emission.

A key feature of Graal is the *Invocation Plugins*. These allow the addition or replacement of method invocations with IR subgraphs created during the graph building phase in Graal. We could potentially leverage Invocation Plugins directly through Graal in order to optimize our Indigo API. However, this was complex since:

- There is no publicly accessible SIMD assembler in Graal.
- Linear scan based register allocation leads to sub-optimal usage in SLAM-like algorithms.
- Since the JVM does not support SIMD instructions, it cannot handle them during register spillage.

The aforementioned reasons necessitated significant engineering effort in order to reach our implementation targets since we would have to extend a production quality JVM to augment it with all the necessary functionality. In contrast, our objective was to achieve a fast and lightweight implementation of our compilation chain, optimizing our computer vision application without having to worry about completeness issues. As a result, we implemented the Indigo compilation stack explained below (Section 4.1).

4.2 Extending The Graal Compiler

Indigo uses a single node plugin that contains its own domain-specific compiler stack. The major benefit of this approach is that it has runtime independence from Graal. Therefore, it can be downloaded and used as a standalone library that, should the JVM implement Graal on top of the JVMCI, SIMD instruction emission can be generated. The compiler stack contains a basic graph builder, optimizer, register allocator, and code generator with a scope limited for its target domain; SLAM applications. The outline of Indigo as well as its interactions with Graal and the JVM, through the JVMCI, are illustrated in Figure 6.

As shown in Figure 6, the LSD-SLAM application has been implemented by using the specialized Indigo API. Upon invoking the application, the Indigo plugin is launched and registers its Invocation plugins through the Graal Specialization class. This class, calls the JVMCI and registers “magic” Indigo Nodes to be added in Graal’s IR. The added Inigo Nodes, are then retrieved and expanded by the Indigo plugin. Typically, each “magic” Indigo Node represents an operation to be performed on the short vector and matrix classes of the Indigo API.

Consequently, the Indigo Nodes are being expanded by the Indigo plugin and specialized IR graphs are being appended to Graal’s IR. The expansion of the Indigo Nodes, create more specialized Indigo Nodes which extend typical node types found in Graal’s IR (e.g. `FixedWithNextNode`). The newly added nodes, are then treated like typical Graal IR nodes and are lowered iteratively until machine code is produced.

Indigo currently uses a number of phases of standard Graal. All Indigo nodes for example inherit from `FixedWithNextNode` node while implementing the lowering interfaces. Regarding optimizations, Indigo uses its own internal ones such as `DeadCodeElimination`, `DotProductRewrite` etc. However IndigoâĂŹs optimizations can not be used from within standard Graal.

The Indigo node is generated during the graph building of the compilation or indirectly during inlining. Figure 7 illustrates how this appears; all nodes in the figure are inserted during the Invocation Plugin. Once a graph is constructed, it is transformed during the optimization phases by exploiting canonicalization and simplification to merge with other nodes. This allows us to maximize the number of operations in the node and eliminate new instance nodes from the graph, leaving the data in registers. A simplification phase traverses the operand edges of the Indigo node to detect other Indigo nodes and merges the internal operation graphs together and with Graal.

4.3 Extending Graal for Vectors

Implementing Indigo as a separate compilation plugin for Graal, helped us optimize the Indigo API by enabling all the optimizations described in Section 3.1. In addition, Indigo has been designed to inline all methods implemented in sub-classes of the `AbstractV128`, `AbstractV256`, `AbstractM128` and `AbstractM256` abstract classes. A key motivation factor of the presented research was the absence of a publicly available SIMD code generator for Graal. Therefore, as depicted in Figure 6, Indigo employs a pluggable SIMD compiler backend that currently implements the SSE instruction set.

Figure 8 contains a simplified overview of the graph within the Indigo node and the transformations that take place until SIMD code is generated. In the given example, `lhs = P(1)`, `rhs = P(2)` and the result are new instances of the `Vector3D` class. When the Indigo node is lowered to the low-level IR used by Graal, the node must claim virtual registers from Graal. At this point we lower the operation to a generic SIMD instruction to be scheduled while profiling the register requirements. In order to maintain the vanilla implementation of Graal, we indirectly use its register allocator to provide general purpose and vector registers by claiming LIR values to satisfy the requirements of the profile. Later, these will be converted into physical registers during the back end phases. The use of profiling enables us to offload the allocation algorithms to Graal, while ensuring that no vector registers are spilled to the stack. This technique guarantees that the JVM will not enter unrecoverable states while being spatially more efficient.

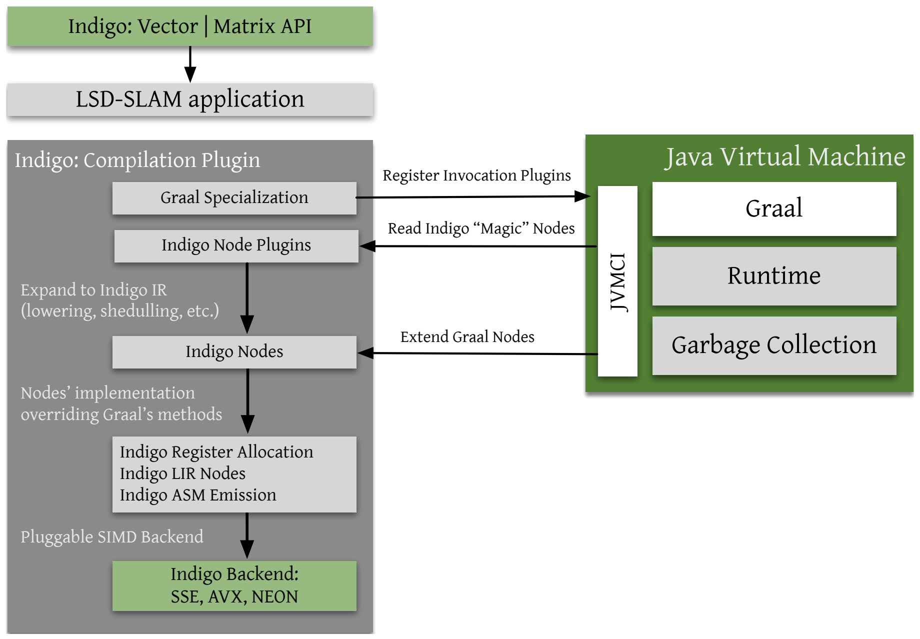
Indigo: Vector | Matrix API
LSD-SLAM application
Indigo: Compilation Plugin
Graal Specialization
Indigo Node Plugins
Expand to Indigo IR (lowering, shedulling, etc.)
Indigo Nodes
Nodes' implementation overriding Graal's methods
Indigo Register Allocation
Indigo LIR Nodes
Indigo ASM Emission
Pluggable SIMD Backend
Indigo Backend: SSE, AVX, NEON
Register Invocation Plugins
Read Indigo "Magic" Nodes
Extend Graal Nodes
Java Virtual Machine
JVMCI
Graal
Runtime
Garbage Collection

Figure 6: Indigo compilation plugin outline.

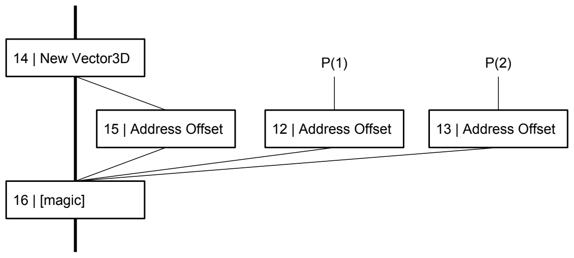

Figure 7: Illustration of an Indigo node within the Graal IR.

The other change is to use a macro-substitution for the get methods of the Indigo API. This allows accessed values to be used directly if constant indices are used and offset-based access if the index is variable. This moves the implementation for the field access away from software but still retaining direct access when compiled.

The vanilla implementation of LSD-SLAM contains hand-tuned code for x86 and ARM architectures, relying on SSE intrinsics for x86 and NEON assembly for ARM. Each manual optimization uses the 128-bit vector unit registers, directly or indirectly, in source code to increase the performance of regular data algorithms. The use of vector instructions is limited to the tracking phase of LSD-SLAM. Although, the tracking phase is greatly influencing performance, it is observed from Java profiling that other parts of the application take advance of SIMD execution. The main restriction of this approach is that in order to apply it manually, developers have to change the source code in three different locations using different syntax; thus, increasing the risk of errors.

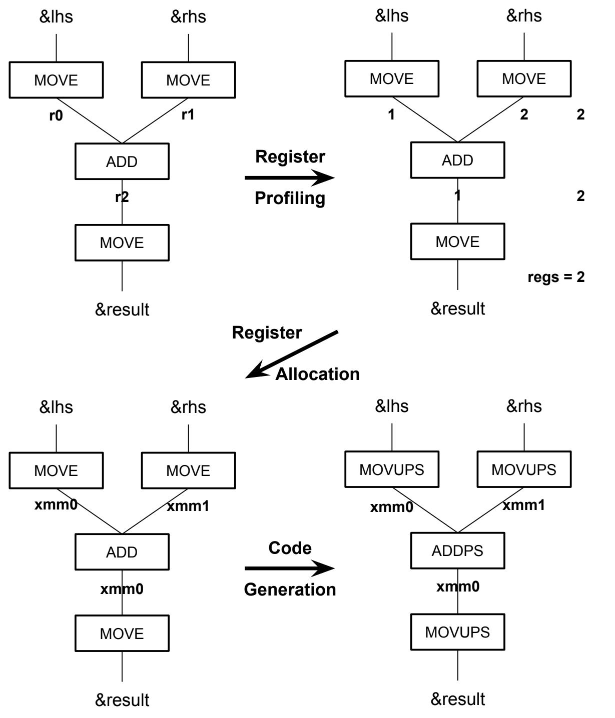

Figure 8: Simplistic illustration of the internal state of the Indigo node from initial creation to the code generation.

The objective of vectorization is to reduce the distance between vector operations in the IR enabling further optimizations through virtualization[1]. With the use of virtualization, provided by Graal's escape analysis, we can maintain temporary vectors entirely in the registers of the targeted architectures. The addresses of the vectors are being used for reading and writing enabling us to break free from the primitive Java types and, more importantly, from the use of arrays. However, since this is not an inherent safe usage of the Java semantics we made the following assumptions:

- Hardware supports 128-bit vector operations, true for ARM NEON and Intel SSE implementations.
- The implementations of the abstract classes `AbstractV128` and `AbstractM128` contain four 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.

While the assumptions dictate the implementation of the base classes in the library, they allow some of the restrictions in Java to be eliminated (e.g, zeroing the fields of a newly created object). This enables the IR to be extended and optimized more aggressively since the semantics are now within the vector abstraction and not within the general purpose language.

5 EVALUATION

The objectives of this section are to validate the correctness and assess the performance of Indigo by following a two-stage evaluation process.

First, we evaluate Indigo as a generic small vectors and matrices Java library, and compare it against the Apache Commons Mathematics Library (CML) [42] which is a library of mathematics and statistics components complementary to the built-in Java libraries. Second, we evaluate Indigo as a SLAM specific library and compare it against alternative implementations using the SLAM kernels extracted from LSD-SLAM.

Evaluation Setup: Table 2 presents the hardware and software configurations used for the evaluation of Indigo. The Java Virtual Machine (JVM) used is the standard version distributed by Oracle [33] built with Graal [32] as its optimizing compiler. Each test is using the Java Micro-benchmark Harness (JMH) [34] with the default settings (10 forks run sequentially with 20 warm-up iterations and 20 measurement iterations). The performance is measured in operations (time to execute the annotated methods) per second (op/s) and contains the standard error also generated by JMH. Finally, the Apache CML version used is 3.6.

5.1 Arithmetic Operations

To evaluate Indigo as a small vectors and matrices library we employ an extended set of arithmetic operations on small vectors and matrices. Namely, the full set of vector operations are:

- Addition ($\vec{a} + \vec{b}$),
- Cross Product ($\vec{a} \times \vec{b}$),
- Scalar Divide ($\vec{a}b^{-1}$),
- Dot Product ($\vec{a} \cdot \vec{b}$),
- Hamilton Product ($Q_1 * Q_2$),
- Scalar Multiply ($\vec{a}b$), and
- Subtraction ($\vec{a} - \vec{b}$).

While the matrix operations are:

- Addition ($A + B$),
- Scalar Division (Ab^{-1}),
- Scalar Multiplication (Ab),
- Vector Multiplication ($A\vec{b}$),
- Matrix Multiplication (AB), and
- Subtraction ($A - B$).

These operations are used throughout the LSD-SLAM implementation of our case study as well as other SLAM applications and the bodytrack benchmark in the PARSEC benchmark suite [4].

Table 3 contains the results of our evaluation on arithmetic operations. Regarding the tested configurations:

- **Apache CML** represents the execution times of JMH using Apache CML compiled with the Graal compiler.
- **Indigo** represents the execution times of JMH using Indigo compiled with the Graal compiler with SIMD generation **deactivated** in the compiler plugin.
- **Indigo-SIMD** represents the execution times of JMH using Indigo with the Graal compiler with SIMD generation **activated** in the compiler plugin.

5.1.1 Vector Operations. Figure 9 illustrates the relative performance of the Indigo library, with and without SIMD instructions against Apache CML, during vector processing. Regarding Indigo without SIMD enabled, performance exceeds Apache CML (from 1.15x up to 7.68x) in all operations except division. This is because in Apache CML there is no divide operation so we essentially perform a multiplication of the inverse of the scalar. This reflects the difference in latency between the division and multiplication in the Intel SIMD SSE instruction set [17] and a reason why division is more generally avoided. Regarding cross and dot products, Indigo outperforms Apache CML by a great margin (6.13x and 7.68x respectively) because Apache CML prioritizes precision over performance [31] (double versus single precision). In LSD-SLAM the

Table 2: Configurations used during performance evaluation.

Hardware	
Processor	Intel Core i7 4770 3.4GHz
Cores	4
Hardware threads	8
L1 Cache	32kB per core
L2 Cache	256kB per core
L3 Cache	8MB per 4 cores
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

[1]The act of performing scalar replacement to object fields.

Table 3: Throughput, as 10^6 op/s, of matrix and vector operations of Apache CML, Indigo, and Indigo-SIMD.

Type	Operation	Apache CML		Indigo		Indigo-SIMD		Indigo / Apache	Indigo-SIMD / Apache	Indigo-SIMD / Indigo
		mean	*s.d.*	*mean*	*s.d.*	*mean*	*s.d.*			
Vector	Addition	220.702	0.310	263.794	0.406	281.439	0.456	1.20	1.28	1.07
	Cross Product	43.509	0.635	266.653	0.389	281.125	0.340	6.13	6.46	1.05
	Scalar Division	216.712	0.302	139.626	0.104	282.410	0.309	0.64	1.30	2.02
	Dot Product	78.231	0.034	600.595	1.733	769.142	6.723	7.68	9.83	1.28
	Hamilton Product	169.256	0.290	194.842	0.251	275.617	0.706	1.15	1.63	1.41
	Scalar Multiplication	221.741	0.314	269.654	0.550	282.133	0.311	1.22	1.27	1.05
	Subtraction	220.868	0.327	262.915	0.377	281.634	0.398	1.19	1.28	1.07
Matrix	Addition	3.451	0.012	66.397	0.720	80.999	3.274	19.24	23.47	1.22
	Scalar Division	3.426	0.007	46.370	0.919	71.073	1.685	13.53	20.74	1.53
	Scalar Multiplication	3.343	0.012	65.068	1.514	74.791	2.650	19.46	22.37	1.15
	Vector Multiplication	3.610	0.010	104.745	1.270	240.963	5.484	29.02	66.75	2.30
	Matrix Multiplication	2.538	0.017	47.892	0.184	77.673	2.910	18.87	30.61	1.62
	Subtraction	3.431	0.011	63.996	1.811	82.935	3.142	18.65	24.17	1.30

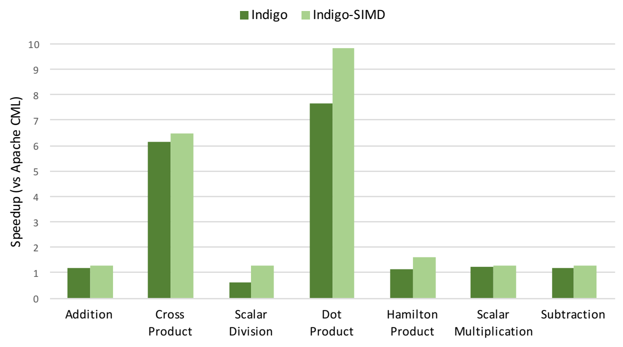

Figure 9: Speedup of Indigo vectors vs Apache CML.

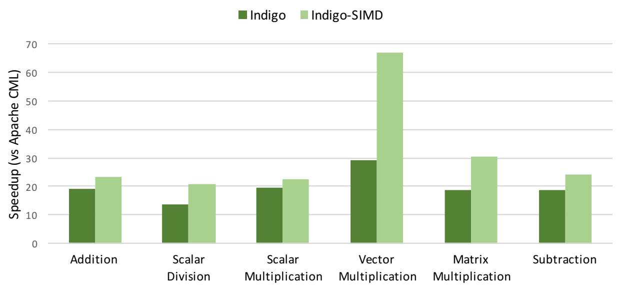

Figure 11: Speedup of Indigo matrices vs Apache CML.

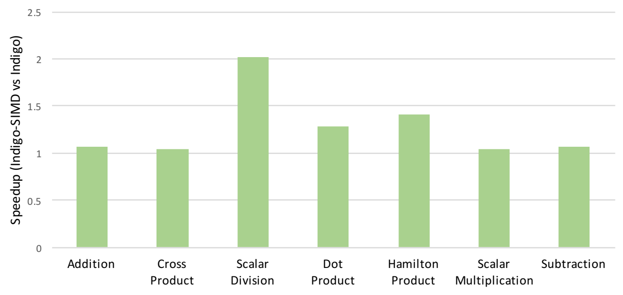

Figure 10: Speedup of Indigo vectors with SIMD instructions.

usage of double precision instead of single precision does not affect the accuracy of the algorithm and since Indigo has been designed for that use case, we use single precision arithmetic.

Regarding Indigo with SIMD enabled, in all cases Indigo-SIMD outperforms the Apache CML (from 1.27x up to 9.83x). Even in the vector division operation, Indigo-SIMD is 1.30x faster than Apache CML. Both configurations (Indigo with/without SIMD enabled) have exactly the same number of object instantiations during execution (in our benchmark we instantiate only one object). The relative performance of Indigo-SIMD against Indigo, as shown in Table 3 and Figure 10, extends from 1.05x up to 2.02x, with division and the Hamilton product to be the best performing.

5.1.2 Matrix Operations. Figure 11 illustrates the relative performance of the Indigo library, with and without SIMD instructions against Apache CML, during matrix processing. Regarding Indigo with SIMD disabled, performance exceeds Apache CML in all cases by great margins, from 13.53x up to 29.02x. The main reason behind that is the data abstractions employed. Since Apache CML uses arrays of arrays (`double[][]`) to store data, more indirection (when reading elements) is being added in comparison to the field-based approach of Indigo. This limits the efficiency of arithmetic operations due to the use of loops during data accesses. This inefficiency highlights the fact that a general solution to a specific domain is not always applicable. Small vector and matrix operations, heavily utilized in the computer vision domain, require a more domain-specific approach as showcased by Indigo. When enabling SIMD code generation, the performance gains become even higher ranging from 20.74x up to 66.75x.

Furthermore, as shown in Table 3 and Figure 12, the advantages of using SIMD generation are significant ranging from 1.15x up to

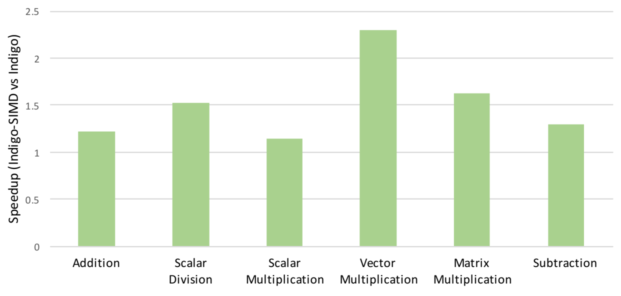

Figure 12: Speedup of Indigo matrices with SIMD instructions.

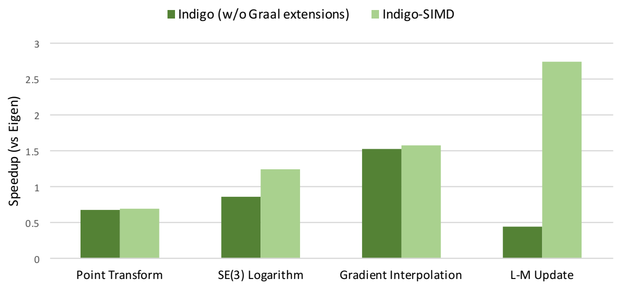

Figure 13: Speedup of Indigo over Eigen.

2.30x compared to non-SIMD Indigo. The matrix-vector multiplication is the best performing operation. This is because it has been targeted by design, specifically for SIMD instructions, in part to optimize point project, a very frequently executed SLAM kernel. Therefore, the data is column major so that matrix-vector multiplication may use a more efficient sequence of SIMD instructions.

5.2 SLAM Kernels

In order to factor out parts of LSD-SLAM that do not exercise vector or matrix operations, we extracted the kernels of interest (see Section 2.2) from the application and thoroughly tested their performance characteristics. These SLAM kernels explore a number of the performance shortfalls of the LSD-SLAM implementation in Java, and represent frequently occurring algorithms that use small vector data types commonly found in the computer vision domain. They aim to demonstrate the overhead and optimizer inefficiencies when dealing with mutable types that are not well encapsulated.

In order to demonstrate the effectiveness of Indigo, without the benefits of SIMD execution, we compare it against the C++ based Eigen library using the SLAM kernels. The results are presented in Figure 13, where the y-axis is the speedup of Indigo over Eigen. For more detail we present the speedup of both Indigo as a pure Java library (just the API), without the Graal extension, and as the presented co-designed approach. In three out of the four tested SLAM kernels, the original implementation of vectors for LSD-SLAM were slower than the C++-based Eigen library. The use of encapsulation and immutability led to lower performance than EJML, a comparable library. However when used with the Graal extension, Indigo manages to outperform Eigen in three out of the four kernels.

Table 4 contains the absolute throughput of the SLAM kernels when built using Indigo. The results can be divided into two groups: those that utilize constants, *SE(3) Logarithm* and *Levenberg-Marquardt Update*; and those that utilize variables, *Point Transform* and *Gradient Interpolation.* As seen, only the second group exhibits performance improvements ranging from 1.33x up to 1.77x.

Regarding the first group of kernels, by inspecting the code generated by the Graal compiler for the *SE(3) Logarithm* we discovered that the main inefficiency lies in the creation of the hat of the SO(3) Lie group. It is a 3×3 matrix containing only three unique values (ignoring signs). Despite the low-level concurrency by using SIMD instructions, constant folding and common sub-expression elimination out-performs the approach of using SIMD. This is because the arithmetic is simplified and therefore Indigo can not employ its optimizations. This, in part, is the reason behind the numerous hand-optimizations found in C++ implementations of SLAM applications [9, 10, 12, 30].

5.3 Discussion

In conclusion, all operations for 128-bit based data structures exhibit performance improvements. In comparison to the Apache CML, Indigo demonstrates the advantage of specializing for a specific domain; computer vision in particular. Indigo allowed the exploitation of domain-specific knowledge in order to provide a set of specific optimizations as a novel compiler plugin. Building compilation plugins for domain-specific languages, can potentially create a collection of pluggable and customizable compilers built for specific purposes packaged as downloadable components.

6 RELATED WORK

Accelerating software with SIMD instructions has been implemented for Java in a number of different ways. For computer vision, offloading computation to native code and to GPUs are subject to ongoing research while there is also work in adding automatic vectorization in optimizing compilers.

6.1 Pure Java Implementations

There is an underlying prejudice against Java for scientific computing because it needs to 'warm up'. However, such studies have been performed while JVMs were still in their infancy. Boisvert et al. explored best practices for, and proposed uses of Java in this environment. However, as JVMs were running as interpreters for bytecode, the performance fell short of what was expected by Fortran and C/C++ developers in scientific computing. This overview

Table 4: Throughput, as 10^6 op/s, of SLAM Kernels in Indigo.

Operation	Indigo		Indigo-SIMD		$\frac{\text{Indigo-SIMD}}{\text{Indigo}}$
	mean	*s.d.*	*mean*	*s.d.*	
Point Trans.	132.86	1.03	234.52	7.26	1.77
SE(3) Log	9.46	0.03	4.07	0.04	0.43
Gradient	157.56	0.88	209.13	6.73	1.33
L-M Update	1.72	0.05	1.64	0.00	0.95

followed the implementation of JAMA [13], a small library that demonstrated matrix decomposition with the encapsulation of an array of arrays in a `Matrix` class. The matrix abstraction introduced by JAMA evolved into implementations that provided a more comprehensive set of features for the scientific computing domain. OoLaLa [25] is a library for matrix-based numerical methods in Java with additional data abstraction, separating a matrix from its data representation and properties. The added abstraction allows implicit specialization for numerical methods, enabling better performance without burdening the user. The separation of storage also allows packed data for special matrices, such as lower triangular or bi-diagonal, that are exploited by specialized execution paths. It also allows extensions for blocking and distributed execution as in Colt [14] and Parallel Colt [44].

6.2 Native Libraries

The library used to represent matrices and vectors in LSD-SLAM and the underlying g^2o map optimization is Eigen [12]. It is developed as a portable library for matrix-based numerical methods in C++, with support for the geometry in computer vision algorithms. Since it is implemented as C++ header files, it must be compiled for each use as there is no binary that may be linked statically or dynamically. The Sophus library [41] extends Eigen to provide the abstractions for poses and their associated Lie groups and algebra [15]. Over its history, it has been hand-optimized with aggressive use of meta-language templates and use of Intel SSE intrinsic methods to specialize for commodity hardware. Eigen contains efficient implementations of many algorithms used in scientific computing with an expressive, yet highly customizable data abstraction. JEigen [37] is a wrapper for the Eigen library, compiled as a binary, for Java. It accesses the library using Java Native Access (JNA) [43] as a way to interact with native code without using Java Native Interface (JNI) with its deficiencies. Another frequently used library in computer vision is OpenCV [19]. Similarly to Eigen, there is a wrapper for the binaries to allow access from within Java [18].

6.3 Vectorization and GPU Offloading

There are three concerns involved in programming with vectors for computer vision applications; a consistent abstraction, optimizations, and hardware utilization. To achieve all three is extremely difficult, particularly the support for optimization of a high-level abstraction for a specific hardware platform. The most common approach to exploit SIMD instructions in Java is to create a native library and use JNI to interact with it directly. Parri et al. created a re-targetable library to achieve this; however the peak speedup is reported for vectors of length 50,000 and above.

As OpenCV makes use of SSE instructions, all wrappers for Java naturally make use of them. However, this is a fixed behaviour and does not exploit the hardware available to the JVM. The real restriction of this approach is that the contents of native methods cannot be inlined and optimized as part of the Intermediate Representation (IR) during compilation. Also, there is no ability to prevent allocation of the vector in the heap which is another source of inefficiency in vectorizing libraries and, therefore, not applicable to SLAM applications.

Another approach used by Project Sumatra [35] and Jacc [7] use the Java dynamic compiler to write kernels for GPUs using OpenCL, CUDA, PTX or HSAIL. These are able to transform Java code into applications that may be applied to data accessible to the GPU or other supported hardware accelerators. Again the limitation is the overhead in transferring data to the computational resources where it is needed. Another restriction is complex control flow with unpredictable behavior, a feature commonly found in SLAM applications.

7 CONCLUSION AND FUTURE WORK

SLAM applications are of interest within the computer vision domain. LSD-SLAM, demonstrated on workstations and mobile devices, is a type of SLAM application that maps an environment whilst calculating the location of a camera. To optimize such applications we introduce **Indigo**; an API, for small vectors and matrices, co-designed with *a lightweight compilation plugin for optimizing computer vision application specialized for classes of small vectors and matrices*. We showcased that Indigo accelerates the performance of SLAM kernels compared to off-the-shelf general libraries (by up to 66.75x). To achieve that, two novel techniques were proposed: (a) a carefully designed library that supports a specific domain with the knowledge of how the existing optimizing compiler operates; and (b) exploitation of the vector units to further accelerate kernels by emitting SIMD instruction from within a novel compiler plugin. This demonstrates the applicability to add domain-specific knowledge to a runtime, simplifying the abstraction and improving the performance.

The Indigo library is the first step towards achieving higher-performing domain-specific compilers. Regarding future work, we are planning to develop several new optimizations within the Indigo compiler plug-in. Such as avoiding field initialization writes and further escape analysis opportunities. Finally, we plan to explore the effectiveness of Indigo on other application domains.

ACKNOWLEDGMENTS

This work is partially supported by the EPSRC grants Anyscale EP/L000725/1, PAMELA EP/K008730/1, and the EU Horizon 2020 ACTiCLOUD 732366 grant. Mikel Luján is funded by a Royal Society University Research Fellowship.

REFERENCES

[1] Peter Abeles. 2017. BoofCV Project Website. http://boofcv.org/index.php?title=Main_Page. (2017).
[2] Peter Abeles. 2017. Efficient Java Matrix Library Project Website. https://github.com/lessthanoptimal/ejml. (2017).
[3] Peter Abeles. 2017. Geometric Regression Library Project Website. http://georegression.org/. (2017).
[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 72–81.
[5] Bruno Bodin, Luigi Nardi, M. Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Lujan, Björn Franke, Paul H.J. Kelly, and Michael O'Boyle. 2016. Integrating Algorithmic Parameters into Benchmarking and Design Space Exploration in 3D Scene Understanding. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 57–69. https://doi.org/10.1145/2967938.2967963
[6] Ronald F. Boisvert, José Moreira, Michael Philippsen, and Roldan Pozo. 2002. Java and Numerical Computing.

http://www.javagrande.org/leapforward/cacm-ron.pdf. (2002).

[7] James Clarkson, Christos Kotselidis, Gavin Brown, and Mikel Luján. 2017. Boosting Java Performance using GPGPUs. In *Proceedings of the 30th International Conference on Architecture of Computing Systems (ARCS '17)*.

[8] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Second Asia-Pacific Programming Languages and Compilers Workshop*.

[9] Jakob Engel, Thomas Schöps, and Daniel Cremers. 2014. LSD-SLAM: Large-Scale Direct Monocular SLAM. In *Computer Vision – ECCV 2014*. Lecture Notes in Computer Science, Vol. 8690. 834–849.

[10] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. 2014. SVO: Fast Semi-Direct Monocular Visual Odometry. In *Proceedings of the 2014 IEEE International Conference on Robotics and Automation*. 15–22.

[11] Google. 2017. Google Project Tango. https://get.google.com/tango/. (2017).

[12] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen Website. http://eigen.tuxfamily.org. (2010).

[13] Joe Hicklin, Cleve Moler, Peter Webband, Ronald F. Boisvert, Bruce Miller, Roldan Pozo, and Karin Remington. 2000. Jama: A Java Matrix Package Project Website. http://math.nist.gov/javanumerics/jama/. (2000).

[14] Wolfgang Hoschek. 2004. Colt Project Website. http://dst.lbl.gov/ACSSoftware/colt/. (2004).

[15] James E. Humphreys. 1994. *Introduction to Lie Algebras and Representation Theory*. Springer, Inc.

[16] Intel. 2017. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com. (2017).

[17] Intel, Inc. 2015. Intel Intrinsics Guide Website. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. (2015). Online; last accessed 31-August-2015.

[18] Itseez, Inc. 2015. OpenCV for Java Website. http://opencv.org/opencv-java-api.html. (2015).

[19] Itseez, Inc. 2015. OpenCV Project Website. http://opencv.org/. (2015).

[20] JEP 243: Java-Level JVM Compiler Interface 2016. JEP 243: Java-Level JVM Compiler Interface. http://openjdk.java.net/jeps/243. (2016). [Online; last accessed 1-Feb-2016].

[21] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*. ACM, New York, NY, USA, 74–82. https://doi.org/10.1145/3050748.3050764

[22] Christos Kotselidis, Andrey Rodchenko, Colin Barrett, Andy Nisbet, John Mawer, Will Toms, James Clarkson, Cosmin Gorgovan, Amanieu d'Antras, Yaman Cakmakci, et al. 2015. Project Beehive: A Hardware/Software Co-designed Stack for Runtime and Architectural Research. *arXiv preprint arXiv:1509.04085* (2015).

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[24] ARM Ltd. 2017. NEON. https://developer.arm.com/technologies/neon. (2017).

[25] Mikel Luján, T. L. Freeman, and John R. Gurd. 2000. OoLaLa: an Object Oriented Analysis and Design of Numerical Linear Algebra. In *Proceedings of the 15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*. 229–252.

[26] Donald W. Marquardt. 1963. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *Journal of the Society for Industrial and Applied Mathematics* 11, 2 (1963), 431–441.

[27] Microsoft, Inc. 2015. Kinect Website. https://www.microsoft.com/en-us/kinectforwindows/. (2015).

[28] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham Riley, Nigel Topham, and Steve Furber. 2015.. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *ICRA*.

[29] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham D. Riley, Nigel Topham, and Steve Furber. 2015. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *Proceedings of the 2015 IEEE International Conference on Robotics and Automation*. 5783–5790.

[30] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. 2011. KinectFusion: Real-Time Dense Surface Mapping and Tracking. In *Proceedings of the 10th IEEE International Symposium on Mixed and Augmented Reality*. 127–136.

[31] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. 2005. Accurate Sum and Dot Product. *SIAM J. Sci. Comput* 26, 6 (2005), 1955–1988.

[32] Oracle, Inc. 2015. Graal Project Website. http://openjdk.java.net/projects/graal/. (2015).

[33] Oracle, Inc. 2015. Java SE. http://www.oracle.com/technetwork/java/javase/overview/index.html. (2015).

[34] Oracle, Inc. 2016. Java Microbenchmark Harness Website. http://openjdk.java.net/projects/code-tools/jmh/. (2016).

[35] Oracle, Inc. 2016. Project Sumatra Website. http://openjdk.java.net/projects/sumatra/. (2016).

[36] Jonathan Parri, John-Marc Desmarais, Daniel Shapiro, Miodrag Bolic, and Voicu Groza. 2010. Design of a Custom Vector Operation API Exploiting SIMD Intrinsics within Java. In *Proceedings of the 23rd Canadian Conference on Electrical and Computer Engineering*. 1–4.

[37] Hugh Perkins. 2015. JEigen Website. https://github.com/hughperkins/jeigen. (2015).

[38] Thomas Schöps, Jakob Engely, and Daniel Cremers. 2014. Semi-Dense Visual Odometry for AR on a Smartphone. In *Proceeding of the 13th IEEE International Symposium on Mixed and Augmented Reality*. 145–150.

[39] Chris Seaton. 2015. *Specializing Dynamic Techniques for Implementing the Ruby programming language*. Ph.D. Dissertation. University of Manchester.

[40] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R Language Execution via Aggressive Speculation. *SIGPLAN Not.* 52, 2 (Nov. 2016), 84–95. https://doi.org/10.1145/3093334.2989236

[41] Hauke Strasdat. 2015. Sophus Project Website. https://github.com/strasdat/Sophus. (2015).

[42] The Apache Software Foundation. 2017. Commons Math: The Apache Commons Mathematics Library. https://commons.apache.org/proper/commons-math/. (2017).

[43] Timothy Wall. 2015. Java Native Access (JNA) Project Website. https://github.com/twall/jna. (2015).

[44] Piotr Wendykier and James G. Nagy. 2010. Parallel Colt: A High-Performance Java Library for Scientific Computing and Image Processing. *ACM Transactions on Mathematical Software* 37, 3 (2010), 31:1–31:22.

[45] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! '13)*.