CO-DESIGNING SOFTWARE
ABSTRACTION AND OPTIMISATION FOR
PRODUCTIVITY AND PERFORMANCE

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2015

By
Colin Barrett
School of Computer Science
## Contents

Abstract .................................................. 11

Declaration .................................................. 13

Copyright .................................................. 14

Acknowledgements ........................................... 15

Abbreviations and Acronyms .................................. 16

1 Introduction ................................................. 19

  1.1 Areas of Interest .............................................. 20

    1.1.1 Java .................................................. 21

    1.1.2 Parallel Frameworks ....................................... 21

    1.1.3 Computer Vision .......................................... 22

  1.2 Motivation ................................................. 22

  1.3 Contributions .............................................. 23

  1.4 Thesis Structure ............................................ 25

2 Background: Parallelism ..................................... 26

  2.1 The Challenges ............................................ 27

    2.1.1 Deadlock ................................................. 27

    2.1.2 Livelock .................................................. 28

    2.1.3 Race Conditions ........................................... 29

  2.2 Hardware Evolution ......................................... 30

    2.2.1 Proliferation of Multi-Core ............................... 30

    2.2.2 Vector Acceleration ....................................... 31

    2.2.3 Heterogeneous Processing ................................. 32

  2.3 Parallel Software Abstraction ............................. 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.1</td>
<td>Low-level Parallel Support</td>
<td>33</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Explicit Parallel Abstractions</td>
<td>34</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Data Analytics and Clouds</td>
<td>37</td>
</tr>
<tr>
<td>2.4</td>
<td>Performance versus Programmability</td>
<td>38</td>
</tr>
<tr>
<td>2.4.1</td>
<td>MapReduce: A Case Study</td>
<td>40</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Map Abstraction Example</td>
<td>41</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Compiler Limitations</td>
<td>42</td>
</tr>
<tr>
<td>2.5</td>
<td>Measuring Performance</td>
<td>43</td>
</tr>
<tr>
<td>2.6</td>
<td>Summary</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>Co-design for MapReduce</td>
<td>46</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction to MapReduce</td>
<td>47</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Related Work</td>
<td>49</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Compromising Programmability</td>
<td>53</td>
</tr>
<tr>
<td>3.2</td>
<td>An Alternative Approach</td>
<td>55</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Managed Runtime Environment</td>
<td>56</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Optimisation</td>
<td>56</td>
</tr>
<tr>
<td>3.3</td>
<td>MapReduce for Java (MR4J)</td>
<td>57</td>
</tr>
<tr>
<td>3.3.1</td>
<td>MR4J</td>
<td>57</td>
</tr>
<tr>
<td>3.4</td>
<td>Semantically-Aware Optimisation</td>
<td>62</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Optimisation Options</td>
<td>62</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Implementation</td>
<td>63</td>
</tr>
<tr>
<td>3.5</td>
<td>Performance Evaluation</td>
<td>69</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Experimental Set-up</td>
<td>69</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Performance Results</td>
<td>72</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Optimisation Performance</td>
<td>75</td>
</tr>
<tr>
<td>3.6</td>
<td>Discussion</td>
<td>76</td>
</tr>
<tr>
<td>3.7</td>
<td>Summary</td>
<td>81</td>
</tr>
<tr>
<td>4</td>
<td>Background: SLAM</td>
<td>83</td>
</tr>
<tr>
<td>4.1</td>
<td>SLAM</td>
<td>84</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Overview</td>
<td>84</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Tracking</td>
<td>86</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Depth Estimation</td>
<td>87</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Map Optimisation</td>
<td>88</td>
</tr>
<tr>
<td>4.2</td>
<td>Existing SLAM Applications</td>
<td>89</td>
</tr>
</tbody>
</table>
4.2.1 Software Tools ........................................... 92
4.3 Recurring Algorithms ........................................ 94
  4.3.1 Poses .................................................. 94
  4.3.2 Image Representation ................................... 95
  4.3.3 Numerical Methods ..................................... 96
4.4 Summary ..................................................... 97

5 LSD-SLAM in Java ............................................ 98
  5.1 Motivation .................................................. 99
    5.1.1 Dynamic Compilation ................................... 99
    5.1.2 Heterogeneous Computing .............................. 100
  5.2 Implementing LSD-SLAM in Java ......................... 101
    5.2.1 Computer Vision in Java .............................. 102
    5.2.2 Design Considerations ................................. 102
  5.3 Compilation Using Graal ................................... 106
    5.3.1 Intermediate Representation .......................... 106
    5.3.2 Optimisation .......................................... 107
    5.3.3 Problems Arising in Compilation .................... 116
  5.4 Summary ..................................................... 117

6 Specialising for Small Vectors ............................. 118
  6.1 Small Vectors in SLAM .................................... 119
    6.1.1 Library Wrappers ...................................... 121
    6.1.2 Pure Java Libraries ................................... 122
  6.2 Co-designing Optimisation ................................ 123
    6.2.1 Removing Indirection ................................ 124
    6.2.2 Adding and Encapsulating Immutability ............. 124
    6.2.3 Semantically-Aware Optimisation .................... 129
  6.3 Implementation ............................................ 130
    6.3.1 Representing Small Vectors ......................... 130
    6.3.2 Extending the Optimiser ............................. 132
  6.4 Performance Evaluation ................................... 133
    6.4.1 Experimental Set-up ................................ 133
    6.4.2 Results .............................................. 138
  6.5 Discussion ................................................ 139
    6.5.1 Immutability in Levenberg-Marquardt ............... 141
6.5.2 Optimising LSD-SLAM in Java ........................................ 143
6.6 Summary ................................................................. 144

7 Accelerating Small Vectors .............................................. 146
7.1 Vectorisation ............................................................. 147
  7.1.1 Hardware ............................................................... 147
  7.1.2 Related Work ........................................................ 150
  7.1.3 Hand-optimisation in C++ ......................................... 152
7.2 Accelerating Small Vectors in Java .................................... 153
  7.2.1 Java Abstraction ...................................................... 154
  7.2.2 Vector Operations ................................................... 155
  7.2.3 Matrix Extensions .................................................... 157
7.3 Extending Graal for Vectors ............................................ 158
  7.3.1 Assumptions ........................................................... 159
  7.3.2 Adding Small Vectors to the IR ................................... 161
  7.3.3 Vector Operation Nodes ............................................. 162
  7.3.4 Code Generation ..................................................... 166
7.4 Performance Evaluation ................................................ 166
  7.4.1 Experimental Set-up ................................................ 167
  7.4.2 Results ................................................................. 169
7.5 Discussion ............................................................... 172
7.6 Summary ................................................................. 174

8 Conclusions ............................................................... 176
  8.1 MapReduce for Java ..................................................... 177
  8.2 Optimising Small Vectors ............................................. 178
  8.3 Future Work ............................................................. 178
    8.3.1 SIMD Instruction Selection .................................... 179
    8.3.2 Additional SIMD Unit Support ................................ 180
    8.3.3 Flexible Accuracy ................................................ 180

Bibliography .............................................................. 182

A MapReduce Word Count .................................................. 200
  A.1 Phoenix 2.0 ............................................................. 200
  A.2 Phoenix++ ............................................................. 202
  A.3 MR4J ...................................................................... 204
List of Tables

3.1 Configurations for MR4J evaluation. ........................................ 70
3.2 Input data sets for the benchmarks. ........................................ 72
3.3 MR4J server configuration performance groups. ......................... 72
3.4 Containers and combiners used in Phoenix++ benchmarks. .......... 77
3.5 MR4J optimisation speedup for GC configurations. .................... 80

4.1 Summary of existing SLAM applications. ................................. 92

6.1 Small vector data types used in computer vision. ....................... 120
6.2 Configurations for new abstraction evaluation. ......................... 134
6.3 Execution times for the SLAM kernels (in nanoseconds). .............. 136
6.4 Results of mutability in the Levenberg-Marquardt update. .......... 142

7.1 SSE instructions targeted for optimisation. .............................. 150
7.2 Computer vision libraries and optimisations. ........................... 151
7.3 Configurations for vectorisation performance evaluation. .......... 167
7.4 Execution time of vector operations (in nanoseconds). ............... 169
7.5 Execution time of matrix operations (in nanoseconds). ............... 170
7.6 Execution time and speedup of SLAM kernels using SIMD instruc-
tions. .................................................................................. 172

B.1 MR4J collector evaluation input sizes. ................................. 208

D.1 Accumulated latency of vector operation using SIMD. ............. 216
List of Figures

1.1 Co-design approaches across abstractions in this thesis. . . . . . . 20
2.1 Deadlock in the Philosophers Dining Problem. . . . . . . . . . . . 28
2.2 Potential data race in increment operation. . . . . . . . . . . . . . 29
2.3 Recursive Cilk method calculating Fibonacci numbers. . . . . . 35
2.4 Task graph for calculating Fibonacci numbers. . . . . . . . . . . . 36
2.5 Overview of Big Data systems. . . . . . . . . . . . . . . . . . . . . 39
2.6 Duplication of code in Phoenix 2.0. . . . . . . . . . . . . . . . . . 41
2.7 Word counting in Apache Flink. . . . . . . . . . . . . . . . . . . . 42
3.1 Using map and fold to find the variance of a sequence. . . . . . . 47
3.2 Word count example using a MapReduce framework. . . . . . . . 48
3.3 Phoenix 2.0 intermediate (key, value) pairs collector. . . . . . . 53
3.4 Phoenix++ intermediate (key, value) pairs collector. . . . . . . . 54
3.5 MR4J intermediate (key, value) pairs collector. . . . . . . . . . . . 59
3.6 Example implementation of word count in MR4J. . . . . . . . . . . 60
3.7 MapReduce using a combiner to eliminate the reduce phase. . . . 64
3.8 An alternative word count reducer in Phoenix 2.0. . . . . . . . . . 64
3.9 Generated methods for the Reducer class transform. . . . . . . . 66
3.10 Bytecode transform of the reducer for word count in MR4J. . . . 67
3.11 Idiomatic reducers for transformation. . . . . . . . . . . . . . . . 68
3.12 MR4J benchmark scalability on the workstation. . . . . . . . . . 73
3.13 MR4J benchmark scalability on the server. . . . . . . . . . . . . . 73
3.14 Relative performance against Phoenix++ on the workstation. . . 74
3.15 Relative performance against Phoenix++ on the server. . . . . . . 74
3.16 MR4J optimisation performance on the workstation. . . . . . . . 75
3.17 MR4J optimisation performance on the server. . . . . . . . . . . . 76
3.18 GC behaviour for MR4J word count on workstation. . . . . . . . 78
7.11 Relative performance of vector operations. . . . . . . . . . . . . . 170
7.12 Relative performance of matrix operations. . . . . . . . . . . . . . 171
7.13 Transpose of a $4 \times 4$ matrix using SIMD instructions. . . . . . 174

B.1 Relative performance of NonBlockingHashMap on workstation . 209
B.2 Relative performance of NonBlockingHashMap on server. . . . . . 210

C.1 Java code for the Point Transform SLAM kernel. . . . . . . . . . 211
C.2 Java code for the SE(3) Logarithm SLAM kernel. . . . . . . . . . 212
C.3 Java code supporting the SE(3) Logarithm SLAM kernel. . . . . . 213
C.4 Java code for the Gradient Interpolation SLAM kernel. . . . . . . 214
C.5 Java code for the Levenberg-Marquardt Update SLAM kernel. . . 215

D.1 Original and SIMD code generated for vector add. . . . . . . . . . 217
D.2 Original and SIMD code generated for vector cross product. . . . 218
D.3 Original and SIMD code generated for vector divide. . . . . . . . 219
D.4 Original and SIMD code generated for vector dot product. . . . . 220
D.5 Original and SIMD code generated for Hamilton product. . . . . . 222
D.6 Original and SIMD code generated for vector multiply. . . . . . . 223
D.7 Original and SIMD code generated for vector subtract. . . . . . . 224
Abstract

Co-designing software abstraction and optimisation for productivity and performance
Colin Barrett
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2015

Improving the execution time of applications is important, but there is a tendency to sacrifice programmability in its pursuit. This thesis investigates co-design approaches, in which APIs provide an abstraction that is strictly maintained using sound software engineering practices while performance is optimised within a managed runtime environment. Flexibility in APIs and weak encapsulation often results in hand-optimisation that restricts the effectiveness of performance improvements and obfuscates functionality. Domain specific applications contain semantics that general purpose languages cannot exploit during compilation. Hand-optimisation addresses this by manually improving the implementation of applications, requiring both expertise and time. Two application domains are used to demonstrate approaches for exploiting semantics to improve performance; MapReduce parallelism and SLAM in computer vision.

Creating correct parallel software is challenging and, thus, frameworks have been developed to exploit the available performance of commodity hardware. MapReduce is a popular programming framework to facilitate the development of data analytics applications. Implementations hide the complexities of data management, scheduling and fault tolerance from users; as MapReduce frameworks evolve, new specialisations and optimisations are introduced. However,
improvements often require manual integration into applications to enable performance gains. Hand-optimisation may be used because the semantics of the underlying abstraction or the scope of the compiler are unsuitable. This thesis demonstrates that the semantics of MapReduce may be used to extend the scope of the dynamic compiler. By analysing applications using a MapReduce framework with co-designed optimisation, it is possible to execute these applications in Java in a comparative time to hand-optimised C and C++. The benefits also include improved efficiency of memory management and reduction in the volume of the intermediate data generated. Hence, it is possible to speedup Java application performance twofold. Most importantly, it does not require any extension or rewriting of existing applications.

Computer vision, SLAM in particular, contains a mix of regular and irregular vector operations. These are not addressed directly, for this domain, by existing abstractions because many of the data types used represent small vectors (2–7 elements). An array is the natural choice to contain the elements of a vector, but it is not optimal for performance or productivity. This thesis presents a new class collection for small vectors in Java using sound software engineering practice. By co-designing the data-level implementation with its interaction with the dynamic compiler, overheads introduced by the strict API have been eliminated during optimisation. This results in kernels, frequently used in SLAM applications, with improved performance relative to a popular C++ SLAM library. In addition to this, it is possible to demonstrate how the small vector implementation may exploit SIMD instructions and registers to improve performance further.

When programmability is prioritised, performance should not be obtained by hand-optimisation because this tends to obfuscate application code. To compensate for this restriction, co-design approaches can extend the communication of application semantics. This thesis demonstrates that there is the potential for co-designed optimisations crossing abstraction boundaries for better performance without affecting productivity.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses
Acknowledgements

I would like to thank the APT research group in the University of Manchester School of Computer Science for housing me for the duration of my PhD study. I have to acknowledge the support provided by my supervisor Mikel Luján and Christos Kotselidis for keeping me going though the many periods of doubt.

Thanks to my parents as they have provided me with much support and they have loyally proof read nonsensical drafts of this thesis.

A special mention also goes to the members of the first cohort of the CDT in the school for keeping me distracted. Thank you for your friendship and encouragement over the last four years. We have shared many entertaining moments and made the most of being students with cash to drink away.
## Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Association for Computing Machinery</td>
</tr>
<tr>
<td>AMD</td>
<td>Advanced Micro Devices</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>AVX</td>
<td>Advanced Vector Extensions</td>
</tr>
<tr>
<td>BCEL</td>
<td>Byte Code Engineering Library</td>
</tr>
<tr>
<td>BLAS</td>
<td>Basic Linear Algebra Sub-Programs</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer Aided Design</td>
</tr>
<tr>
<td>CAS</td>
<td>Compare And Set (or Swap)</td>
</tr>
<tr>
<td>CLR</td>
<td>Common Language Runtime</td>
</tr>
<tr>
<td>CRTP</td>
<td>Curiously Recurring Template Pattern</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>CV</td>
<td>Computer Vision</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EISPACK</td>
<td>Eigenvalues and Eigenvectors Package</td>
</tr>
<tr>
<td>EKF</td>
<td>Extended Kalman Filter</td>
</tr>
<tr>
<td>FMA</td>
<td>Fused Multiply Add</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FPMR</td>
<td>FPGA MapReduce</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collector</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GFS</td>
<td>Google File System</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU’s Not Unix</td>
</tr>
<tr>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose GPU</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>HVX</td>
<td>Hexagon Vector Extensions</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IPP</td>
<td>Intel Performance Primitives</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>JAMA</td>
<td>Java Matrix</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Kit</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time</td>
</tr>
<tr>
<td>JNA</td>
<td>Java Native Access</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JRE</td>
<td>Java Runtime Environment</td>
</tr>
<tr>
<td>JVM</td>
<td>Java VM</td>
</tr>
<tr>
<td>LINPACK</td>
<td>Linear Systems Package</td>
</tr>
<tr>
<td>LIR</td>
<td>Low-Level IR</td>
</tr>
<tr>
<td>LSD-SLAM</td>
<td>Large Scale Direct Monocular SLAM</td>
</tr>
<tr>
<td>MARE</td>
<td>Multi-Core Asynchronous Runtime Environment</td>
</tr>
<tr>
<td>MMX</td>
<td>Multimedia Extension</td>
</tr>
<tr>
<td>MR4J</td>
<td>MapReduce for Java</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Architecture</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PARSEC</td>
<td>Princeton Application Repository for Shared-Memory Computers</td>
</tr>
<tr>
<td>PLASMA</td>
<td>Parallel Linear Algebra Software for Multicore Architectures</td>
</tr>
<tr>
<td>PS</td>
<td>Packed Scalar</td>
</tr>
<tr>
<td>PTAM</td>
<td>Parallel Tracking and Mapping</td>
</tr>
<tr>
<td>RGB</td>
<td>Red-Green-Blue</td>
</tr>
<tr>
<td>RGB-D</td>
<td>RGB-Depth</td>
</tr>
<tr>
<td>SE</td>
<td>Standard Edition</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SLAM</td>
<td>Simultaneous Location and Mapping</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multiprocessor System</td>
</tr>
<tr>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td>Single Scalar</td>
</tr>
<tr>
<td>SSE</td>
<td>SIMD Streaming Extensions</td>
</tr>
<tr>
<td>STM</td>
<td>Software Transactional Memory</td>
</tr>
<tr>
<td>SVD</td>
<td>Singular Value Decomposition</td>
</tr>
<tr>
<td>SVO</td>
<td>Semi-Direct Visual Odometry</td>
</tr>
<tr>
<td>TBB</td>
<td>Threading Building Blocks</td>
</tr>
<tr>
<td>UAV</td>
<td>Unmanned Aerial Vehicle</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>YARN</td>
<td>Yet Another Resource Negotiator</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Software abstractions are actively evolving and have been for decades, resulting in complex multi-layered software stacks. They simplify the implementation of applications, especially in domains where the full potential of modern hardware architectures is required. There is a compromise because performance is often sacrificed with added strictness in good software engineering practice; a compromise that is explored in this thesis. Application Programming Interfaces (API) are frequently extended or code is hand-optimised to bypass the restrictions imposed by the abstractions or to use low-level techniques to specialise for hardware to improve performance. These modifications can be seen as a means of communicating application semantics to a lower level abstraction to improve optimisation during compilation.

This thesis investigates the limitations of semantic communication across boundaries in software abstractions for managed runtime environments. In particular, parallel frameworks and computer vision are used to demonstrate how performance is lost when the software abstraction and application semantics do not align. It may be observed that this often leads to poor programming practice with optimisation added, by hand, to application code, complicating it or obfuscating its purpose. Bypassing the restriction of abstractions by using hand-optimisation often limits the portability and re-usability of code. Novel approaches to optimisation demonstrate how co-design in different abstractions may improve the performance of machine code generated. This is achieved by allowing application semantics to communicate across abstraction boundaries within managed runtime environments. Figure 1.1 shows the software abstraction layers and where the contributions in this thesis communicate over the boundaries. Co-designing
Figure 1.1: Co-design approaches across abstractions in this thesis.

layers of abstraction allows comparative performance to hand-optimised, statically compiled applications. Importantly, changes to the runtime environment leaves application code unchanged.

The Java Virtual Machine (JVM) offers portability for applications over different hardware architectures. However, additional layers of abstraction increases the challenge of generating efficient machine code. The research in this thesis demonstrates three approaches to improve the communication of semantics to optimise the performance of applications executing in a dynamic, managed environment.

1.1 Areas of Interest

The JVM is a managed runtime environment that, while popular for parallel frameworks in the Apache Foundation repertoire [1], is not commonly utilised for performance in high-performance computing. Its strengths are in the programmability and portability afforded by Java. It is used in this thesis as it provides an open source implementation [2], with support for runtime modification of classes [3] and a dynamic compiler [4] with an extensible optimiser [5]. In this research three areas are of interest: the interaction of Java with the JVM, especially the unnecessary allocation of objects on the heap; the challenges in maintaining a simple API for parallel frameworks; and the optimisations required by computer vision applications.
1.1.1 Java

Java is designed as a strongly typed, object-oriented programming language [6]. Its syntax and class file structure is stable, forwards-compatible and associated with execution of classes on the JVM. It is a general purpose language that strives for simple syntax. The Java Software Development Kit (JDK) [7] provides a mature, off-the-shelf library supporting many abstractions required by applications.

Within the JVM, memory is managed automatically by the Garbage Collector (GC) so developers do not need to worry about the life of objects. This differs from C++ where, without libraries [8], objects are allocated and referenced explicitly, whether on the stack or heap, and dynamic objects are tracked and deleted manually. The interaction of data and threads in Java is guaranteed only at well-defined synchronisation points ([6] pages 645–660) allowing flexibility in approaches for optimisation. The combination of object headers, the GC and the memory model leads to challenges in producing efficient applications; especially when performance is memory bound [9].

There are many implementations of the JVM specification [10] addressing a range of optimisations for specific applications and architectures. The Java HotSpot Server VM (version 1.8 [7]) is a popular and freely available JVM and is used for the evaluation of MapReduce for Java. However, this is a proprietary product from Oracle, Inc. and, therefore, cannot be modified for the purposes of research during computer vision optimisation. For this work into the specialisation and acceleration of small vectors the OpenJDK project is used [2]. This is a full, open source implementation but, in this thesis, it is used with the dynamic compiler replaced by the open source version from the Graal project [4]. Together these provide the flexibility required to co-design a class collection and optimisations for SLAM.

1.1.2 Parallel Frameworks

Parallel programming encompasses a diverse and ever-expanding set of software and hardware technologies. One approach to tackling the challenges of programming in such an environment is parallel frameworks. These provide an abstraction for applications, hiding data management, scheduling and fault tolerance in systems with many processing nodes. The frameworks target application development using many types of parallelism including data-parallel (Hadoop [11]),
task-parallel (Spark [12]) or data-streams (Flink [13]).

The APIs of parallel frameworks are frequently extended and configuration parameters are added to improve the communication of application semantics to lower levels of their software stack. This increases the complexity of parallel frameworks and increases the level of expertise required to produce efficient applications [14]. The purpose of these frameworks was to reduce the need for expertise of efficient parallelism; however the challenge now includes system configuration. An inconvenience in extending an API is that improved performance is only available by explicit inclusion within application code.

1.1.3 Computer Vision

The domain of computer vision is wide-ranging but has recurring computations. Simultaneous Location and Mapping (SLAM) applications are no exception and many competing implementations exist. Within SLAM there are implementations specialising for surface location (PTAM [15]), localisation (SVO [16]), mapping (LSD-SLAM [17]) or a combination (KinectFusion [18]).

SLAM applications target heterogeneous architectures, utilising SIMD units, GPGPUs and multi-core systems to maximise their performance. However the software abstractions used are frequently bypassed with hand-optimisations to improve the performance of generated machine code. This increases the difficulty of implementation, portability and maintainability of applications.

1.2 Motivation

MapReduce parallel frameworks for multi-core, shared-memory architectures and SLAM contain examples of applications that violate sound software engineering principles to improve performance. The semantics of the applications are not communicated to lower levels of abstraction and hand-optimisation is added to application code. Extending an API for added functionality is a good practice but this thesis is motivated by scenarios where APIs provide unnecessary exposure to implementation detail and permit hand-optimisation because:

- It is easier than generalising behaviour in a framework or library;
- It is not possible to communicate the exact semantics of an application; or
• Specialisation does not exist for the desired machine code.

Ada [19] is a very-strongly typed, object-oriented language. Its strength is the ability of the compiler to detect and prevent many runtime errors. It is highly modular and there is little room to bypass the strict object specifications. It is able to produce very reliable and deterministic machine code and is one reason that it is used in safety-critical systems [20]. The strict nature of this approach makes it a challenge for use in general purpose computing; although the lessons it provides are important. The specification of abstractions and their implementation are separated, by syntax and by source code files. The ability to isolate and specialise implementation while maintaining a strict interface is highly desirable for productivity [21].

Increasing the levels of software abstraction affects semantic communication. Rather than continually extending the API or permitting its bypass, this thesis investigates approaches to implement the optimisations targeted behind the API; leaving application code unchanged. This principle is demonstrated by co-designing frameworks, libraries and optimisers to focus on the communication of application semantics.

1.3 Contributions

This thesis demonstrates three approaches using co-design in a managed runtime environment. Figure 1.1 shows the different scopes of semantic communication crossing different abstraction boundaries. The approaches maintain a strict public API and use of existing instances of hand-optimisation to explore techniques to recover performance lost by restrictions imposed by using software engineering principles.

MapReduce for Java (MR4J) is an implementation of the popular parallel framework with only the two fundamental components, \textit{map} and \textit{reduce}. This differs from the state-of-the-art equivalent frameworks because intermediate data specialisations are not available. The contribution is an approach to overcome the restrictions imposed by Java due to the class data abstraction and the scope of the dynamic compiler. By using co-design to develop the framework and optimiser in co-operation, it is possible to develop semantically-aware transformations to application code at runtime. It demonstrates optimisation transformations that,
when applied, can improve performance by reducing both the execution time and interaction with the garbage collector.

In computer vision hand-optimisations are frequently used to improve the performance of algorithms. These affect the programmability of the application code and are undesirable. This thesis also contributes two alternative approaches to optimise an abstraction for small vectors for use in SLAM applications. An API has been implemented with optimisations co-designed in the JVM using the Graal dynamic Java compiler [5]. This extends the concept of ‘co-evolution’ of classes in the Java language and the JVM but it is aimed at domain specific optimisations.

The first approach is a demonstration of domain specific specialisation for small vectors that are frequently used in SLAM. Existing libraries with a suitable matrix abstraction tend to target scientific computing; however there is a mismatch in problem size between the two domains. The abstraction addresses small vectors by co-designing the implementation of the class collection with sound software engineering principle to maximise the effectiveness of the Graal compiler. The principles include encapsulation, immutability and validity checking of arguments. Semantic information added to the compiler is the nature of arithmetic used in SLAM applications. When the optimiser is aware of these assumptions it is able to improve the performance of frequently occurring SLAM kernels; at times Java is able to outperform C++.

The second approach generates code to utilise SIMD instructions and registers. Many computer vision applications contain assembly code to accelerate regular and idiomatic vector operations using SIMD instruction sequences in hardware. As they are hand-optimisations they restrict the portability and increase the code complexity in applications. Java is designed for productivity during application development and it is portable; existing approaches to vectorisation are not suitable for the abstraction discussed in this thesis. The Graal compiler is extended to support SIMD instruction sequences for the small vector class collection. Another important aspect of the contribution is the placement of intermediate data to be kept in SIMD registers, removing the need to allocate extra objects on the stack. This improves the performance as fewer instructions are required, there is low-level concurrency of arithmetic operations and there is less interaction with memory.
1.4 Thesis Structure

Java, the managed runtime environment of choice in this research, is common throughout the entire thesis, encompassing the two main themes; optimising parallel frameworks and computer vision algorithms.

The challenges of parallel programming and the need for frameworks to provide a software abstraction are introduced in Chapter 2. MR4J is introduced in Chapter 3 and describes the contribution made to the co-designed approach for MapReduce using multi-core, shared-memory architectures.

Computer vision is a domain with many interesting aspects, Chapter 4 introduces SLAM and the recurring computation observed in existing implementations. Chapter 5 contains an introduction to LSD-SLAM and its new implementation for Java developed for this thesis. It introduces the Graal compiler and the optimisations that are applied during execution and where opportunities are missed. Chapter 6 introduces a new small vector class collection co-designed for specialisation and optimisation. Chapter 7 explores the compilation of the classes to use SIMD units to provide further performance improvements.

The conclusions and possible future directions for this work, based upon the ability to improve the communication of application semantics, are found in Chapter 8.
Chapter 2

Background: Parallelism

Developing parallel software is not a new challenge; however since the introduction of multi-core to commodity computers, awareness has increased greatly [22]. This chapter provides an introduction to the challenges of developing parallel software using threads. A thread is a sequence of operations that is managed as an independent unit of execution. Many programs contain a single thread and thus, run to completion sequentially. More frequently programs are written with multiple threads sharing resources to progress to completion. On a single core the software context is switched to allow a thread to run. Parallelism enables threads to run concurrently on multiple cores.

Shared, mutable state refers to resources shared between software threads that may be modified. Common examples include files, peripherals or databases and are often unavoidable when creating applications. The challenge is that software will fail if shared, mutable state does not have adequate protection should it be modified by multiple software threads. Multi-core and many-core architectures are not a requirement to fail but, in the search to utilise the full capability of these architectures, low-level error mitigation is creating problems.

This chapter describes challenges arising from programming for multi-core architectures and some techniques existing to overcome them efficiently. It also illustrates what is meant by productivity, a trade-off between programmability and performance and includes the metrics to measure the impact of novel developments.
2.1 The Challenges

Parallel software has long established challenges arising from conditions that, while simple to describe, are difficult to protect against efficiently. The two manifestations of these in software are: a lack of progress to completion, due to deadlock or livelock; or incorrect and unreliable results, due to race conditions. A lack of progress generates a program that will run indefinitely and become unresponsive, potentially blocking an entire system. Incorrect results often go undetected and quietly corrupt values in memory. However there is always the chance of memory or program exceptions due to the corruption of references or stacks.

Unfortunately none of these manifestations will occur deterministically in general purpose languages due to the unpredictable scheduling of parallel software. Because of external timing dependencies and behaviours such as operating systems, control flow variations and data interaction it is possible for bugs to go undetected for long periods of time. However the consequences may be serious, as is evident from the deaths surrounding the Therac-25 radiation therapy machine [23]. A failure to understand the existence of race conditions led to several patients receiving lethal doses of x-ray radiation between 1985 and 1987.

2.1.1 Deadlock

A simple method to protect shared, mutable state is to allow only exclusive access to it. Blocking is when a software thread is waiting at a synchronisation point for access to a resource that is accessed by another. Deadlock occurs when two or more software threads are blocking each other and cannot make progress.

The classic illustrative example is the Dining Philosophers Problem [24] whereby five philosophers sit around a table, each with a meal. Unfortunately there are only five chopsticks but two per person are required to eat. A philosopher will only attempt to eat while not theorising; an activity that takes a non-deterministic, finite period of time. Deadlock occurs if all philosophers pick up one chopstick before any get a chance to pick up a second (Figure 2.1). Without a strategy to recover from this, each philosopher will continue to hold one chopstick and, as unable to eat, all will go hungry.

This problem is easily achievable with multiple software threads and multiple shared, mutable states. An unwelcome feature of deadlock is that, because it
Figure 2.1: Deadlock in the Philosophers Dining Problem.

requires rarely occurring conditions, detecting the error and its cause can be difficult. Often these errors go undetected for long periods of time or are attributed to other problems [25].

In parallel software, deadlock is further complicated by priority inversion [26]; the deadlock error forming from this requires two conditions. The first is that a low priority software thread has exclusive access to state required by another with a higher priority. The second condition, and cause of the deadlock, is when a medium or high priority software thread is executing and correctly preventing a low priority software thread from executing. In this case the low priority software thread dictates the control of the other tasks requiring access to the state. Their priorities have been inverted and progress is halted.

2.1.2 Livelock

Software threads in livelock repetitively execute but no useful work is achieved. Deadlock and livelock are closely related, software will stop responding for both, but they differ in their mechanism.

Another food-based example to illustrate the error in livelock is that of a
2.1.3 Race Conditions

A race condition is anomalous behaviour caused by the relative timing of software threads when accessing shared, mutable state. Software threads executing in parallel race to complete execution of their instructions. When left unmanaged the exact timing is non-deterministic and, at the system level, the order of instructions will appear interleaved.

Parallel correctness concerns errors that arise from the interleaving of instructions relating to a shared, mutable state from different software threads. Figure 2.2 demonstrates the introduced errors from an instruction that is commonly considered by programmers as atomic, occurring within the confines from a single instruction. The common syntax \texttt{i++}, when compiled or interpreted is, in fact, a load, increment and store instruction that may be interleaved with other software
CHAPTER 2. BACKGROUND: PARALLELISM

threads. Depending on the interleaving the result may be different from expected. In Figure 2.2, two threads are concurrently incrementing a shared index (i) and, depending on the interleaving, it is possible to produce different results.

It is not possible to predict when this inconsistency will occur and it appears as a transient error when it does. An array access out-of-bounds error may result if the shared, mutable state is an index manipulating an array. For low-level languages, a thread may attempt to access or call a partially modified or invalid address leading to a corruption in the program execution.

2.2 Hardware Evolution

The introduction of multi-core into commodity computing increased awareness of parallelism in the general population. This accompanied existing parallelism in the form of graphics and multimedia extensions, as both accelerators or separate co-processors. These have been made accessible to developers through better support in compilers, languages and libraries. The challenge in these is to maintain a stable yet flexible infrastructure to utilise cutting-edge technology, without over complicating the software interface.

By looking at the diversity and evolution of hardware available to improve application performance, it is understandable why there is a tendency to hand-optimise for the target architecture. However the nature and target of hardware parallelism change from device to device and is increasingly dynamic, making this approach infeasible.

2.2.1 Proliferation of Multi-Core

The priority during the evolution of commodity processors was to increase sequential performance by increasing clock speeds, optimising pipelines and improving memory architectures. In the early 2000s, clock speeds began to level off at about 2.2GHz. To maintain an increase in marketable performance, multiple cores were fabricated within the same processor, a multi-core system. In commodity hardware, this started with the AMD X2 range of processors [27] and the Intel Pentium D (or Extreme Edition) before being renamed Core [28]. These have developed into the server, workstation and personal computer processors used today. Features including power management, memory management and accelerator have evolved, but the underlying architecture is bound to the x86 instruction set [29].
CHAPTER 2. BACKGROUND: PARALLELISM

Homogeneous multi-core systems, like these early implementations, contain multiple copies of the same core with equal performance; a Symmetric Multi-Processing (SMP) system. Software threads may be assigned to execute on any core and will, if independent and uninterrupted, individually take a comparative length of time to complete. The challenge in these architectures is optimising applications to maximise the efficiency of software executing in the presence of shared, mutable state. To ensure correct parallel software, protection is generally required at some point; this has the effect of adding sequential code and reduces the efficiency of an application.

An alternative approach to increasing the number of cores in a ‘processing node’ is to increase the number of processors and connect them. This is achieved on a motherboard by using a high-speed interconnect to communicate data like HyperTransport [30]. In this configuration, each processor has direct access to its own bank of main memory. With global addressing, it is then possible for any core in any processor to access any memory location. Due to the difference in direct and indirect access to data, access times are variable. NUMA (Non-Uniform Memory Architectures) is the acronym used to describe these and the challenges are discussed by Laudon et al. [31].

Cache-consistency is also a restriction in these systems. A write to memory, if shared, must be viewed consistently by all other cores. Scalable cache coherence is a challenge in the transition from multi-core to many-core architectures [32]. Projects such as EUROSERVER [33] are innovating server architecture around the idea of Islands of Coherency to remove this restriction. It is a starting point to reduce the cost, while improving energy and software efficiency, in distributed computing.

2.2.2 Vector Acceleration

Floating point arithmetic, multimedia extensions, encryption and graphics are all possible in software but are aided by hardware acceleration. The use of additional, dedicated hardware reduces the time taken to handle special operations.

Floating Point Units (FPU) are tightly-coupled co-processors that accelerate mathematical operations in a binary environment. Dedicated hardware splits, transforms and builds values for operations that are expensive in software using integer pipelines. They apply basic arithmetic instructions but also contain special extensions such as Fused Multiply Add (FMA) to reduce the overhead of
rounding and data transfers. The overhead of floating point arithmetic is considerable when FPUs are not available. This makes them a shared resource and can create sequential code when floating point arithmetic is used, limiting scalability.

Multimedia extensions (MMX) were added to the x86 instruction set to allow the concurrent modification of data streams [34]. These instructions are Single Instruction Multiple Data (SIMD), so a single instruction can manipulate eight bytes in a single clock cycle. As multimedia applications evolved and the use of the accelerators increased, new instructions were added. In x86, Streaming SIMD Extensions (SSE) became the next standard in the set, followed by Advanced Vector Extensions (AVX) [29]. The registers for MMX were 64-bits wide and, as the demand for performance increased, these are now wider. SSE supports 128-bit, which equates to four single precision IEEE-754 floating point numbers, and AVX2 support 256-bit registers. Compilation to SIMD instructions is not well supported in general purpose programming environments and often appear as intrinsic methods [35] or inlined assembler in C and C++ applications. Xeon Phi [36] extends this by packaging many cores specialising in 512-bit SIMD operations in a hardware module.

Over the same period, dedicated hardware evolved to support larger scale parallelism for multimedia, including the pipeline for rendering computer graphics. Graphical Processing Units (GPU) began as external modules to enhance graphic intensive applications such as Computer Aided Design (CAD) or gaming. As power constraints and cost became a cross-cutting concern these were often moved to sit as tightly-coupled co-processors, as in the Intel Haswell architecture [28]. They have evolved to contain clusters of many-core rendering pipelines with specialised memory architectures. GPUs are now frequently used for general purpose programming; albeit restricted due to the nature of the architectures. Libraries have been developed [37, 38] to improve the programmability of software for this kind of hardware.

2.2.3 Heterogeneous Processing

Hardware has evolved from simple SMP architectures, mainly because of thermal constraints. Heterogeneous multi-core processing implies a single system with multiple cores, each with different capabilities. The challenge is greater for heterogeneous architectures as the locality of data and scheduling of software threads play a larger role in the performance of a given application.
In addition to GPUs, there are many different realisations of heterogeneity as they specialise for different applications. Xilinx [39] fabricates microprocessors with tightly-coupled FPGAs to provide configurable hardware acceleration. The Cell Broadband Engine Architecture [40] contains co-processors to accelerate vector and multimedia operations, targeting media applications and was used in the PlayStation 3 and smart televisions. Intel Core processors contain a Turbo Boost feature [28] that can increase the clock speed of a core if requested by the operating system; a more subtle version of heterogeneity because performance is no longer consistent. For the mobile device market, architectures such as the ARM big.LITTLE architecture [41] contain powerful cores to execute intensive applications and energy efficient cores to handle system maintenance.

The complexity for software abstraction increases as tasks compete for specialist resources or core availability due to power restrictions or reallocation. The challenge is to create an application code that is portable but exposes the full potential for speedup for devices on dynamic systems that are increasing battery powered devices. Productivity in heterogeneous processing is a difficult area to tackle for high-level abstractions.

### 2.3 Parallel Software Abstraction

Parallel software abstractions isolate software development from the direct control of the parallel hardware. However, this does not mean implementing parallel algorithms and correctness does not require thought. Software abstractions interact with one another creating complex software stacks for use by software developers.

#### 2.3.1 Low-level Parallel Support

Software threads are an execution context managed by the operating system or runtime environment that may be created, destroyed and controlled. Manual creation is achieved through wrapper libraries [42] or operating system calls providing handles. Coordinating shared access to mutable data is the greatest challenge in parallelism and a number of low-level abstractions offer protection against data races. Interacting with parallelism at such a low level may give rise to deadlock and livelock as the complexity of the application increases.
Atomic Instructions, at the lowest level, are hardware instructions that guarantee a given instruction to be atomically executed. The most common atomic operation is the Compare And Set (CAS) that will set a memory location only when the state of it is equal to an operand of the instruction. No change is made if this is not the case and no other instruction will have access while the comparison and write operations are executing, thus ensuring atomicity. The abstraction provides the fundamental mechanism for synchronisation and allows exclusive access to shared, mutable state.

Locks allow a single software thread exclusive access to a stream of instructions, a critical section, at any one time. They are usually used around sections that access shared, mutable state. They are considered pessimistic, or conservative, as only one thread may progress even if correct behaviour could occur; for example two or more threads modifying the same array but at different locations. Implementations of locks include mutexes from pthreads [42], the use of the Java monitor patterns ([43] pages 60–61) or explicit locks within Java ([43] pages 278–289).

Transactions [44] address the pessimistic nature of locks and allow optimistic access to instructions. A transaction managed in software, although now available in commodity processors [45], provides atomic access to shared, mutable state. Independent software threads may access the same stream of instructions and restrictions are only imposed if contention is detected when writing data. When this occurs one thread will continue, committing their transaction, while the others roll-back and retry their transactions. The overlapping of access assumes that there will not be many collisions, optimistically allowing access to the critical section.

These synchronisation abstractions are the building blocks for parallel software at all levels. They are simple to understand and apply, but to achieve correct and efficient code requires expert knowledge; lock-free data structures are still complex enough to be the topic of a PhD [46]. Parallel frameworks provide a software environment that removes this responsibility from software developers and allows them to focus on parallel algorithms.

2.3.2 Explicit Parallel Abstractions

To avoid the potential of deadlock and livelock, explicit parallelism provides the abstraction of tasks (light-weight threads). These, when implemented, contain
CHAPTER 2. BACKGROUND: PARALLELISM

... cilk int fib (int n) {
  if (n < 2) return 1;
  else {
    int x = spawn fib (n - 1);
    int y = spawn fib (n - 2);
    sync;
    return (x + y);
  }
}
...

Figure 2.3: Recursive Cilk method calculating Fibonacci numbers.

the functionality of an application and they are managed in a runtime environment. The runtime separates the programming interface from the complexities of thread management and scheduling. Explicit parallel abstractions provide support, implementations and tools that encourage focus on algorithm and application design.

The basic principle in many implementations is the idea of a fork/join pool; an executor containing worker threads that awaits tasks to run. The common implementations, and source of the name, contain constructs to encourage recursion. Fork splits execution, spawning new tasks, so that parts of the application execute in parallel. Join provides a point of synchronisation, where one or more tasks merge into the execution flow of their parent task. The number of forks and joins are not restricted, so it need not be a binary tree.

Several languages are built upon this principle and their performance is demonstrated over a range of benchmarks [47, 48]. X10 contains the finish and async keywords to achieve fork/join and Cilk uses spawn and sync. Figure 2.3 demonstrates how this works by giving an example of Cilk code to calculate a number in the Fibonacci series using recursion. The task is the fib method: it takes an integer as an argument and returns the number associated with that location in the sequence (1, 1, 2, 3, 5 ...). If the argument is 0 or 1, then it returns the value of the first two numbers, which are both 1. Anything greater forks execution into two independent tasks that find the two preceding numbers. This occurs recursively and is visualised in Figure 2.4. The values may only be used once
they are valid, as it is at this point execution is joined, the values are then added and returned to its parent. The conditional interaction of data between tasks is called data-flow dependency.

While the Fibonacci series is a useful example to explain the fork/join abstraction, it is in practice very inefficient. Furthermore the abstraction does not prevent data races; if the sync was omitted there would be no knowing what the result would be. The inefficiency stems from the nature of the algorithm, in that there are independent tasks executing on the same value argument. This duplication of effort is not needed in a sequential implementation where a loop only needs to calculate each number in the sequence once; and is one of many reasons why efficiency in parallel software is difficult.

By implementing the scheduling within fork/join frameworks, software developers are free to focus on creating correct and efficient parallel algorithms. The underlying framework is responsible for the parallelism as well as lower level details such as “load balancing, paging and communication protocols” [49]. There is support in Java since the inclusion of the java.util.concurrent.ForkJoinPool class [7]. It was originally ported to the language by Doug Lea [50] and then included officially in the Java SE 1.7 release. It provides abstract and concrete classes to implement tasks. The tasks may be grouped as a collection or executed individually, any of which may fork and join as per the framework.

Figure 2.4: Task graph for calculating Fibonacci numbers.
2.3.3 Data Analytics and Clouds

As the capabilities of computing have increased so the quantity and complexity of data have also. Madden [51] describes ‘big data’ as information that is too big, fast or hard for existing tools. There has been a revolution that has seen ‘big data’ analysed by a selection of tools hosted on distributed computing clusters (Clouds). These advocate a priority for productivity because deadlock, livelock or data races could corrupt applications that take weeks or months to execute.

These parallel frameworks create a programming environment with implicit parallelism. Within the bounds of the API provided, tasks are implemented in a style more akin to functional programming because there are no unmanaged side-effects, no shared, mutable state. The driving force behind this is the nature of the data; it is big, distributed and often irregular. The frameworks manage data in a way that prevents the application from making assumptions about the location of data that may tempt manual modification to abstractions; potentially resulting in bugs.

Google published details of its MapReduce framework [52] and the distributed file system [53] leveraged to abstract ‘big data’ storage and access. These provided a technique to simplify data analytics on a large scale with implicit data management, fault tolerance and, most importantly, parallelism on commodity computing clusters. Hadoop provides an open source implementation with its own distributed file system [11]. Research projects have also extended MapReduce to include frameworks for iterative algorithms [54, 55]. Furthermore wrapper frameworks have been developed to leverage the power of MapReduce for general applications (FlumeJava [56]) or specific domains (Mahout [57]).

Despite diversity in the applications and wrappers for MapReduce, limitations emerged as popularity grew: open source and commercial providers extending the tools concurrently. Google developed Percolator [58] and Pregel [59] to deal with data more efficiently using big databases and graphs respectively. These permitted a more incremental approach to analysis, increasing the throughput of repetitive actions on ‘big data’.

A similar series of developments emerged from the Apache Software Foundation repertoire. The scheduler (YARN) and file system (HDFS) from Hadoop where generalised and exposed for use in other tools [11]. The Spark [12] framework extends the capabilities of MapReduce to include graph-based and streaming
data sets. Flink [13] takes a similar approach but has its origins in the Stratosphere research project [60]. Figure 2.5 illustrates the scope and interaction of parallel frameworks forming projects within the ‘big data’ arena based on a presentation by Volker Markl [61]. It also highlights the multi-layered, data-flow approach to abstraction in creating software stacks implementing parallel frameworks.

Ranger et al. [62] investigated the suitability of MapReduce for multi-core shared-memory architectures. They successfully demonstrated that parallel frameworks also offer a simple abstraction for small scale parallelism. However as data is no longer distributed, there is nothing preventing shared, mutable state and data races emerging once again. Flink is able to specialise for multi-core systems as well as distributed systems [13], so it is possible to maintain the programming model, but there is an associated performance overhead in the abstraction.

2.4 Performance versus Programmability

Performance is one of any number of metrics used to quantify the success of an application. There are many possibilities including execution time, accuracy, resource utilisation and energy efficiency. All have their advantages but there is frequently a compromise required as they interact with one another. For example, improving accuracy may result in additional iterations or a different data abstraction, leading to an increase in execution time.

Programmability is an abstract concept encompassing the ideas surrounding the ease of programming. The experience gained over decades of software and abstraction development has led to guidance from software engineering literature. Libraries are full of advice on utilising programming languages in teams and individually for public consumption. Within the scope of this thesis, programmability is focused around:

- Clear and efficient APIs;
- Best practice in software design principles;
- Separation of concerns (including parallelism); and
- Suitability of abstractions (whether it is fit for purpose).
Figure 2.5: Overview of Big Data systems.
These concepts are not easily measured and will be justified, with examples, when applicable.

Performance and programmability are the cornerstones of *productivity*, the ability to produce a result efficiently for a given brief. There are many influences that complicate priorities including maintenance, re-use and expertise but the compromise is, at its most simple, measured by time. In a hypothetical, one-shot application, a poor performing program takes five days to create and five days to generate the result. For an improvement to the original program to be productive the sum of the time for new development and execution must be less that the five days used to generate the original result. Productivity is a judgement made by managers in software projects. When using third party tools, including parallel frameworks, the efficiency of implementation is linked to its programmability. The time of execution is a resource, especially in high performance or distributed computing, as are developers producing the applications.

### 2.4.1 MapReduce: A Case Study

MapReduce, described in detail in Chapter 3, is a simple parallel framework with two phases. Map creates intermediate (key, value) pairs from input data and reduce collects values with the same key and creates a single output value. There is a performance overhead associated with creating and distributing the intermediate data. To overcome this, a *combine* method provided by the developer locally reduces values as they are emitted from the map phase. This minimises the volume of data stored and communicated before between the map and reduce phases. However in practice the combiner is often duplicate code, as observed in code from Phoenix 2.0 [63], a MapReduce framework for shared-memory architectures. Lines 59–63 are identical to lines 68–72 in Figure 2.6 from the word count benchmark (the full source code is available in Appendix A). The duplication of code increases the likelihood of mistakes or inconsistencies and is made more likely by the use of void pointers and structures in the API.

Phoenix++ achieves a similar solution by introducing combinators as a modular class implementation. In this case, the complexity arises from the need to understand the internal operation of the framework to implement a bespoke combiner. There is no API and as such developers must build functionality directly into the framework. While this results in faster and more scalable applications, anything other than the simplest of applications requires expertise and time.
... 057  void reducer(void *key_, iterator_t *itr_) { 058  char *key = (char *)key_; 059  void *val; 060  intptr_t sum = 0; 061  while (iter_next(itr_, &val)) { 062    sum += (intptr_t) val; 063  } 064  emit(key, (void *) sum); 065 } 066
067  void *combiner(iterator_t *itr_) { 068  void *val; // duplicate of 59 069  intptr_t sum = 0; // duplicate of 60 070  while (iter_next(itr_, &val)) { // duplicate of 61 071    sum += (intptr_t) val; // duplicate of 62 072  } // duplicate of 63 073  return (void *) sum; 074 } ...

Figure 2.6: Duplication of code in Phoenix 2.0.

Chapter 3 demonstrates that the same functionality is achievable without the need for a combine method or module. By using the semantics of the application within the context of the framework, the classes may be rewritten at runtime to automatically generate the duplication observed in Phoenix 2.0. The result is comparative performance but without losing sight of the MapReduce abstraction or introducing the need for duplication.

2.4.2 Map Abstraction Example

The map phase from MapReduce is based on the map concept from functional programming where every value in a list is transformed by a unary function. It has a one-to-one mapping but there is also an alternative, the flat map, that has a one-to-zero or many mapping. Spark and Flink [12, 13] implement this so that it may be used for MapReduce like behaviour. Figure 2.7 contains code for word counting in Flink. It is the responsibility of developers to know when to use the map or flatMap data abstraction. The application contains the semantic information required in the mapping function but it is not exploited by the
...  

    DataSet<String> text = ... 

    DataSet<Tuple2<String, Integer>> wordCounts = text 
        .flatMap(new LineSplitter()) 
        .groupBy(0) 
        .sum(1); 

    ...

Figure 2.7: Word counting in Apache Flink.

framework. While not investigated in this thesis, it can be observed that a link between the application and runtime could remove the hand-optimisation existing in the framework. It is this semantic information that the research within this thesis aims to address.

2.4.3 Compiler Limitations

The map and reduce tasks are semantically linked, as is demonstrated by the introduction of the combiner methods in many MapReduce frameworks. There are data availability restrictions in place for distributed systems that do not apply for single-node systems. In shared memory implementations, such as Phoenix [62], data is available to all worker threads at all times. This permits optimisations to the extent that the reduce phase can be eliminated entirely [64]. However it has to be hand-optimised because the general purpose optimising compiler used is unaware of the framework specific opportunities for improvements in performance.

In Java, the dynamic compiler is given either a method or block of code to optimise and has no understanding beyond the bytecode contained within the snippet and invoked methods. Also, it cannot change the data abstraction provided by the developer and the JDK libraries because the implementations are defined by the JVM specification [10]. This creates a limitation to the effectiveness of the compiler for frameworks where two semantically linked tasks are temporally separated. In the case study discussed, these are the map and reduce tasks, the reduction will occur once the mapping is complete; therefore to the compiler they are unrelated and cannot be optimised. The research presented in Chapter 3 proposes an approach to address this limitation. It introduces semantically-aware optimisation through co-design (developing the framework and optimiser in co-operation).
2.5 Measuring Performance

One of the main aims in software parallelism is to reduce the execution time. Although there are also other metrics for performance including energy efficiency and memory utilisation, both are still correlated to time. Energy efficiency may be improved by race-to-halt, reducing static power usage by allowing deeper power saving whilst idle [65]. This requires execution time to be fast so that there are meaningful periods of idle where savings are available. Memory utilisation is also linked to execution time as longer access times are associated with physical distance from the core and, thus, power required to drive the data lines.

The metric for improved execution time is speedup and is the normalised increase in performance for an application. In parallel software this is a comparison between a reference execution time and a parallel implementation (Equation 2.1)

\[
\text{speedup} = \frac{t_{\text{ref}}}{t_N},
\]

where:

- \( t_{\text{ref}} \) = reference execution time, and
- \( t_N \) = time of parallel execution on \( N \) cores.

The exact nature of speedup is determined by the reference used. Relative speedup uses the parallel implementation running on a single thread and provides an insight into the efficiency and scalability of the parallelism. Absolute speedup uses a pre-determined reference time, that may be the performance of another tool or a bespoke implementation. The purpose is to compare alternative approaches or a comparison to a well implemented single core version. This final approach is rarely used as it highlights the inefficiency of parallelism. For example the Cilk Fibonacci code in Figure 2.3 will have a speedup of less than one against even a naïve iterative version.

The challenge is to optimise software to get as close as possible to linear, or super-linear, speedup. A linear speedup is achieved in strong scaling when an application doubles in speed when using double the number of cores. Super-linear speedup improves upon this and is normally possible due to the multiple instances of cache available to cores in multi-core architectures. As mentioned in the available low-level abstractions, there is frequently an element of sequential execution due to shared, mutable state that inhibits ideal improvements. Within this thesis relative performance is used to describe normalised execution time
CHAPTER 2. BACKGROUND: PARALLELISM

speedup in like-for-like comparisons. For example, the reference execution time on four cores divided by the experimental execution time on four cores.

Speedup has been discussed since the observation of Amdahl’s Law [66] (Equation 2.2). This acknowledges the relation between sequential and parallel execution and its effect on scalability; as the number of cores tends to infinity speedup tends towards the reciprocal of the sequential execution time. It is desirable to minimise the proportion of time spent executing sequential code.

\[
\text{speedup} = \frac{1}{(t_s + \frac{t_p}{N})},
\]

(2.2)

where:

- \(N\) = number of cores,
- \(t_s\) = time of sequential execution, and
- \(t_p\) = time of parallel execution.

Gustafson argues that when there is 0.4–0.8% sequential execution it is still possible to achieve speedups ranging from 1016 to 1024 for a 1024-core system [67]. He argues that to achieve these values, the sequential proportion would need to be close to 0%. In practice, these speedups are made possible by the ability to scale the problem with the level of parallelism. The time of parallel execution thus becomes \(t_pN\) and the speedup should be linear as predicted in Equation 2.3

\[
\text{speedup} = N + (1 - N)t_s,
\]

(2.3)

where:

- \(N\) = number of cores, and
- \(t_s\) = time of sequential execution.

Gustafson notes “it is important for the research community to overcome the “mental block” against massive parallelism” through the misuse of Amdahl’s Law [67]. However this actually describes and differentiates between strong and weak scaling. Strong scaling shows how execution time for a fixed problem size changes on increasing core counts. Weak scaling shows how execution time for a fixed problem size per core changes on increasing core counts. In this thesis performance measurements relate to strong scaling for relative and absolute speedups.
CHAPTER 2. BACKGROUND: PARALLELISM

2.6 Summary

Parallel software is challenging because it is difficult to avoid shared, mutable state in practical applications, whether these are physical resources or values in memory. The risk of deadlock, livelock or race conditions exist when protection is not correctly implemented. Low-level abstractions provide efficient means to achieve application integrity but require expertise to use effectively. Over time higher level abstractions, in the form of explicit parallelism or parallel frameworks, have improved the programmability. However there is a loss in ultimate performance and hardware support is often limited.

Hardware has evolved rapidly, creating an environment where powerful tools are at the disposal of commodity computing. In specialist domains many-core architectures, accelerators and GPUs are often used as a platform for software parallelism. Hand-optimisation may be used to specialise applications for a targeted architecture but the diversity of hardware is making this infeasible. Parallel frameworks have developed into complex multi-layered software stacks, with each layer providing a different abstraction. These frameworks generally target distributed computing; however there is assessment in utilising the same programming approaches for multi-core architectures.

The abstractions require a compromise between programmability and performance and rather than remove the requirement for expertise, it has just been shifted. Instead of solving the challenges of low-level parallelism, specialist knowledge is now required to efficiently leverage APIs and configure distributed computing systems. In this compromise, it may be observed that the semantics of applications are not communicated efficiently between layers in the stack. Developers need to declare this explicitly or extend the complex data structures in existence, risking bugs and affecting programmability in large systems.

This thesis will use performance metrics used in parallel computing to investigate the impact of automating the communication of application semantics on performance. MapReduce, as a popular parallel framework, is used to demonstrate the feasibility of modifying Java classes at runtime to improve the performance on applications without affecting the original application source code. This targets optimisations beyond the scope of the static and dynamic compilers in the existing JVM; demonstrating an approach for co-designing layers in the software stack.
Chapter 3

Co-design for MapReduce

MapReduce, as the name suggests, has two main phases of execution targeting data analytics; a map phase to find useful data and a reduce phase to consolidate it. It is possible to assume data independence so the framework is able to partition and parallelise data in each of the phases, reducing the time for execution. The programming model is exposed through a simple, clean API. However, there are inefficiencies in the framework if map and reduce are the only elements in the API.

The addition of combining duplicates reduce-like functionality, embedding it in the map phase. This improves performance measured by execution time and scalability. This chapter shows that this is a hand-optimisation, bypassing the restrictions of the parallel software abstraction. While this improves performance, it also has a negative impact on programmability.

The semantic information available in applications, using the MapReduce framework, are the data-dependencies between the input data and result. When exploited it allows application methods to be transformed to reduce the volume of intermediate data, adding combining without manual intervention by developers. Therefore this chapter applies the concept of co-design, by developing a framework and optimiser in co-operation, to improve performance while maintaining the existing clean, simple API.

The primary contribution in this chapter is an approach that uses the behaviour of MapReduce to analyse the semantics of applications. Code is identified and exploited by rewriting the class file so that the reduce functionality can be inlined in the map phase. Previous approaches have relied on developers to achieve this but as the MapReduce framework is clearly defined it is possible to
automate this process. By applying a novel bytecode transformation in the runtime environment is it possible to improve the ability of the optimising compiler to virtualise intermediate data. This improves performance by enhancing the code generated during compilation and reducing the interaction, and therefore overheads, with the GC.

3.1 Introduction to MapReduce

MapReduce frameworks build upon the functions map and fold, fundamental building blocks in functional programming. Map transforms all elements in a sequence into a new sequence of equal length and fold transforms a sequence into a single value by repeatedly applying a binary function to elements in the sequence. Figure 3.1 illustrates how this may be used to find the variance from an input sequence and its mean.

The mapping of every value is independent and may be executed in parallel,
CHAPTER 3. CO-DESIGN FOR MAPREDUCE

48

map(partition)
for word in partition
emit(word, 1)
reduce(key, values)
sum = 0
for value in values
sum = sum + value
emit(key, sum)

Figure 3.2: Word count example using a MapReduce framework.

the fold function may only be parallelised if it is commutative and associative. This combination is enough to inspire the implementation of the MapReduce framework. In MapReduce the map phase is used to extract relevant information as (key, value) pairs, a tuple of a key and an associated value, and the reduce phase folds the values associated with each unique key. Figure 3.2 contains pseudo code for a word count example that will be used as a running example throughout this chapter and forms the basis of one benchmark. MapReduce requires only these two methods to create an application and it is this simplicity that has made it popular in parallel computing.

Accompanying the pseudo code is an illustration of the data transformations through the map and reduce phases. The map phase finds words in the input string and emits a (key, value) pair to the framework containing each word and a count of one. The reduce phase folds all the emitted values associated with unique words by accumulating the counts and then emits the result. The (key, value) pairs are collected internally, between the map and reduce phases, using implementation specific mechanisms discussed later in this chapter and is represented by a grey dashed line in the figure.

Sorting the (key, value) pairs is another implementation specific activity that occurs in many MapReduce frameworks. In the cases where the (key, value) pairs are sorted, they use the natural ordering of the key, although this may be
overridden in applications. The word count example tests the flexibility of the framework as it is more useful to order the output by the occurrences of the words, sorting (key, value) pairs by the value in descending order. Figure 3.2 will output the tuples ("know", 3), ("i", 2) and ("you", 1) in this order if sorted using this method of comparison. In some frameworks, the sorting is a by-product of the internal collection of values and in others, it is optional; including the framework presented in this chapter (MR4J).

3.1.1 Related Work

There is a wide range of implementations of the MapReduce framework because it is a popular framework for data-parallel applications. This section contains an overview of the more important frameworks historically, and in the context of multi-core implementations.

Distributed Computing

MapReduce, as both a framework and a name, was popularised by Google, providing an API allowing it to automatically distribute data and process it across data centres [52]. The API, written in C++, hides many aspects of the parallelism for distributed computing (the scheduling, data distribution and fault tolerance) and relies on the Google File System (GFS) [53] to achieve this. The map and reduce methods are central to the framework; these are distributed to worker nodes to convert input files into a file or collection of files containing (key, value) pairs. Intermediate (key, value) pairs are stored as intermediate data files and are distributed in the partition phase to collect values for each key before the reduce phase is initiated.

Google has refined the MapReduce model to improve the efficiency of the reduce phase by adding two new methods; partition and combine. The partition method separates and sorts the intermediate (key, value) pairs into bins, applying a user-defined hash function. The different values associated with any given key are adjacent in these bins due to the implicit sort. Each bin is then processed by a single reduce worker producing the final (key, value) pairs. The combine method is used during the map phase and performs a local reduce to emitted (key, value) pairs before they are passed to the partition and reduce phases. This improves performance by reducing the size of data transferred after the map phase.
Another popular MapReduce framework for distributed computing is Hadoop [11], an open source implementation of a MapReduce framework written in Java, extended to allow map and reduce functionality implemented in other programming languages. It is implemented in a similar manner to the Google MapReduce framework, including the same refinements. The major differences are: the language in which it is written; the open source nature of the code and the Hadoop Distributed File System (HDFS) in lieu of the GFS. In addition to the HDFS, Hadoop relies on the application distribution and scheduling provided by Yet-Another-Resource-Negotiator (YARN). Hadoop uses Java interfaces enhanced by using generics to define the map and reduce methods; the map task implements the \textit{Mapper} interface and the reduce task the \textit{Reducer} interface. This allows flexible, yet typed, parameters for the (key, value) pairs.

Google and Hadoop take time to initialise an application before it can be executed and there are some tasks that require several iterations; it is these initialisation overheads that become problematic. Several novel MapReduce frameworks extend the Google and Hadoop platform to provide better support for iterative algorithms. HaLoop [55] extends the Hadoop framework by creating a loop-aware scheduler to support iterative applications. The operation allows for a fixed iteration count and also for iterations to continue until a condition is met (for example when there are no changes between iterations). Twister [54] also focuses on iterative algorithms but achieves this by implementing a flexible API that uses dynamic transfer of intermediate data using the Message Passing Interface (MPI) [68].

The limitations of MapReduce in distributed computing stem from two sources; the first is the execution time and the second is the compatibility with ‘big data’ collections. The overall execution time of a MapReduce application may be no greater than alternative methods. However, access to results is not possible until the application has completed execution, and minor updates to a data set cannot be reflected in the result unless the full application is re-run. This is a restriction that caused Google to investigate alternative solutions to the ‘big data’ challenges. Firstly Percolator, a framework that efficiently analyses only changes to a data set to update the final result [58] and secondly Pregel, by using different data structures providing coherent intermediate data, applications may provide useful data during execution [59]. The emerging de facto data structure in data centres is the graph that is not necessarily data parallel and can contain irregular
parallelism that is not compatible with the simplistic MapReduce model. Apache, the software foundation supporting the development of Hadoop, has addressed this issue by supporting alternative projects including Spark and Flink, graph and data-stream based parallel frameworks [12, 13].

**Multi-core**

Following the success of MapReduce for distributed computing Ranger *et al.* evaluated the suitability of the framework to simplify applications using multi-core, shared-memory architectures [62]. Phoenix [62, 63], a framework written in C, and Phoenix++ [64], an object-oriented framework written in C++, extract the principles of the MapReduce framework and substitute the communication strategies of a distributed system with shared-memory buffers. There are a number of deviations from the original, distributed computing MapReduce frameworks. Firstly, the sort is moved from the intermediate step and is executed on the final (key, value) results. Secondly due to the low-level language, the user must provide the method to split the data referenced and cast to and from the void pointers required by the framework. These add complexity to the implementation and maintainability of applications written for the framework. The main advantage of this framework is the optimisations included to reduce the overhead of data manipulation in memory and maximise data locality. The principle aim of the Phoenix project is to provide “an efficient implementation on shared-memory systems that demonstrates its feasibility” [62].

An extension to Phoenix is Ostrich [69] that changes the internal implementation of the MapReduce framework to use buffers to store (key, value) pairs between the map and reduce phases. From this, it claims additional speedup by beginning the reduce phase when workers executing the map function become idle. Over iterative applications, this small advantage builds and noticeable savings in execution time can be made.

The challenges when implementing applications using Phoenix stem from the decision to sacrifice programmability for performance. Exposing the internal workings of the framework to the developer, an important restriction of the original frameworks, has increased the need for expertise to eliminate mistakes. The design decisions in the different implementations of Phoenix are discussed in detail later in this section.
CHAPTER 3. CO-DESIGN FOR MAPREDUCE

A second change that has emerged in all MapReduce frameworks, but a fundamental part of multi-core implementations, is the combine functionality. In Phoenix++, many applications do not require a reduce phase as the intermediate (key, value) pairs are reduced when emitted from the map phase. Phoenix provides two methods to implement for this purpose and Phoenix++ provides a few Combiner class implementations and the option to implement new classes to achieve the desired goal. Immediate combining complicates some reduce operations that are not possible until all values associated with a key are available. A simple example is the average position of points in the k-means clustering algorithm used as a benchmark later in this chapter. A running total may be kept but a count of the elements is also required and the final mean value can only be calculated sequentially outside of the MapReduce framework. This change is evident in other implementations including the second iteration of Hadoop and in applications that build on it such as the JavaScript based MapReduce engine used in MongoDB [70].

Implementations Targeting Accelerators

The popularity of MapReduce has encouraged some novel implementations and these have been investigated because they offer a variety of interesting approaches.

Mars [71] is a MapReduce framework written for GPUs in NVIDIA CUDA. It aims to exploit the processing and memory bandwidth advantage of such hardware. There are several compromises made to simplify the core for the graphics rendering pipelines of the hardware. This includes simplification by using only the map, reduce and sort functions but having to duplicate the emits required in each stage to allow the allocation of memory in the output before the values are transmitted. The applications using the framework are written in C or C++ and use void pointers as with Phoenix. These demonstrate the benefits of MapReduce with its core abstraction; the principle is portable across architectures.

The MapReduce framework for the Cell Broadband Engine Architecture [72] successfully hides the complexities of the heterogeneous architecture, providing the user with a familiar programming model in C. Whilst similar, its enhancement over Phoenix is that the memory management is handled, through necessity, entirely within the runtime software of the MapReduce framework. FPMR is an FPGA accelerated MapReduce framework targeting machine learning and data mining applications [73]. The map phase uses concurrent execution nodes to
stream data faster than would be possible on a general purpose processor. These examples demonstrate the range of optimisations possible when using a simple API with only two compulsory phases and tuples.

### 3.1.2 Compromising Programmability

In 2007, Phoenix was created to evaluate the MapReduce programming model on multi-core systems. Written in C, it showed the speedup and scalability potential for several benchmarks [62]. Ranger et al. successfully argued that with “a careful implementation, MapReduce is a promising model for scalable performance on shared-memory systems with simple parallel code” [62]. At the centre of the implementation was the primary contribution, a buffer to collect intermediate (key, value) pairs from the map phase. The framework evolved to improve further the scalability and, again, the main area of change was the collector [63].

The collection of intermediate (key, value) pairs, for example (“HELLO”, 1) in a word count application is illustrated in Figure 3.3. Each worker thread has its own array of partitions that it may access asynchronously; within this, the hash of the key determines which partition is used. Each partition is a sorted array of key and value array pointer pairs. A binary search is used to order the keys and a memory copy is used to make space to insert the new pair. A partition
Figure 3.4: Phoenix++ intermediate (key, value) pairs collector.

is re-allocated with twice the capacity. The values are stored in a linked list, each node has a maximum capacity of ten values and, when full, a new node is allocated. The main difference between versions 1.0 and 2.0 of Phoenix is the introduction of a method that combines the values of a full node into a single value to make space for new values without the need of allocating new memory. This allows the removal of synchronisation and the bottlenecks associated with shared, mutable state. However it is possible to have uneven distributions of the (key, value) pairs, increasing the execution time because inserting a new pair involves shifting existing pairs.

Phoenix++ was created to provide a modular framework and to take advantage of the features of C++. Some of the benefits include the introduction of types for keys and values in the class template to allow type checking at compilation and eliminating void pointers used in the original Phoenix frameworks. The advantageous features of C++ include the support of the TBB libraries [74] for concurrent allocation of memory and static binding of overridden methods using the Curiously Recurring Template Pattern (CRTP) [75].

The modular approach targets two main features, the first is the container that holds the keys and associated combiner objects, the second of the modular components. These are defined in the class template and several are provided
as part of the Phoenix++ distribution [76]. While providing performance improvements, it destabilises the MapReduce programming model by exposing the implementation to developers, thus, requiring an understanding of the system, including the management of shared, mutable state. Figure 3.4 illustrates two common containers, a hash map for non-sequential or unknown keys and an array for sequential integer keys. A copy of the container is made for each thread and passed as an argument to the map method. The key is copied into the container and the value is passed to a combiner. A combiner is a running reduction of values that removes much of the need for a reduce phase, saving time in the execution of an application. An example Combiner is the sum combiner that takes the value and adds it to a running total, the value is then discarded and the sum is all that remains in the combiner.

As much of the container is pre-allocated as possible, for example the array container size is declared in the class definition and, as such, is fixed at compile time. This is a major restriction when creating applications with an unknown result array size. A problem with the modular approach is that to implement a new container, developers have to understand how the operators are overloaded, and how these and the methods required are used by the MapReduce framework. While this means specialised code and better performance are possible, the user is required to develop code at a level that the framework is there to abstract from. Appendix A contains the implementation of the word count benchmark for Phoenix and Phoenix++ including examples of code duplication, inconsistent and complex API components, poor encapsulation and limited compile time validation. These all impact on the programmability of the abstraction.

### 3.2 An Alternative Approach

The complexity of Phoenix is introduced by the desire to keep data static. There is an overhead associated with allocating and deallocating memory on the heap compared to communicating via arguments and return values on the stack. However the nature of the framework in multi-core implementations pushes the design responsibility to application developers. The advantage is that primitive types and pointers to existing data may be used in the framework, considerably improving performance [62, 63, 64].

Java, and later C#, have been popularised by the memory abstraction they
CHAPTER 3. CO-DESIGN FOR MAPREDUCE

bring to the programming arena. This removes the responsibility of design decisions entirely, improving the programmability of applications by the introduction of garbage collection. Ranger et al. [62] suggests that managed runtime environments may be suitable for equivalent frameworks and, in this chapter, this assumption is investigated.

3.2.1 Managed Runtime Environment

Managed runtime environments, such as the Java Virtual Machine (JVM) or the Common Language Runtime (CLR), are programs that manage the execution of applications. They offer a language agnostic abstraction that is supported by libraries. This allows portability of ideas and applications over hardware devices. There is also a reduction in programming concerns with the dynamic nature of compilation and in the memory model selected, specifically garbage collectors.

A Garbage Collector (GC) is a managed heap where objects are allocated. References to an object are monitored and, when there are no remaining uses, the object is considered garbage and discarded during a collection cycle. The performance of these is improving incrementally and Java provides six different configurations in Java SE 1.8 [7]. This means the developer does not need to explicitly manage objects so is unlikely to generate memory leaks, improving programmability.

Dynamic (or Just-In-Time (JIT)) compilation attempts to balance the benefits of compiling to native machine code against the expense of doing so. The JVM and CLR interpret bytecode instructions, a process that is slow but requires very lightweight compilation. As parts of the application are frequently executed (a hotspot) it may be useful to accelerate these portions of code. It is these that are compiled and, unlike statically compiled languages, additional assumptions may be drawn from the runtime properties of the abstractions. The properties are derived from the semantics of the application and may only emerge as the application runs.

3.2.2 Optimisation

Dynamic compilers for Java do not translate bytecode directly into machine code. They use many phases of optimisation that create efficient, fast generated machine code. MapReduce has two phases of execution and the semantic distance causes
an unnecessary overhead. This distance is created by the level of abstraction used by many compilers, the method. In existing JVMs, unless both phases appear in the same method they cannot be optimised and it is the combiner that is used to manually achieve this. Co-designing the framework and an optimiser with a different abstraction is an alternative approach that this chapter presents.

3.3 MapReduce for Java (MR4J)

Hadoop is the main MapReduce framework for Java and it operates using a complex and flexible scheduler (YARN) and scalable data storage (HDFS). MR-J [77] reuses the API of Hadoop and replaces the scheduler with threads for use on multi-core, shared-memory architectures. Both of these implementations use manual combining and, as such, a novel implementation is required to explore the applicability of an optimiser. Another problem, and a reason why there is a need for a specific MapReduce framework for Java, is that ForkJoinPool and streams in the JDK do not provide the functionality. The map method is closer to the functional map where it is a simple mapping of a unary function onto a value. However the reduce function is applicable to the intermediate values associated with each key.

3.3.1 MR4J

To evaluate the capabilities of MapReduce for managed memory, MR4J has been implemented as a MapReduce framework for Java. As the research presented aims to investigate the performance without sacrificing programmability there are three important design objectives.

- To maximise the use of standard Java libraries and exclude the use of native code to maintain portability across hardware architectures and operating systems.

- To create an API with the simplicity of the original Google implementation of MapReduce to encourage the user to focus on functionality.

- To maximise encapsulation to avoid distracting the user with implementation detail dependent on the framework and to permit future extensibility.
At the centre of the MR4J architecture are two elements, the scheduler and the collector for intermediate (key, value) pairs. The `ForkJoinPool` class in Java SE 1.8 provides a clean, off-the-shelf scheduler focusing on lightweight tasks executing on worker threads accessed via a work-stealing queue [50]. This emulates the scheduling approach of Phoenix and removes the need for the implementation of a new scheduler. However this prevents the transfer of both Phoenix and Phoenix++ intermediate (key, value) pair collectors. Phoenix demonstrates the flexibility of using hash tables in a generic framework, as does Phoenix++, for all but sequential integer keys. The JDK contains two hash table implementations suitable for parallel applications and a third class considered uses compare and set implementation of a concurrent hash map [78]. These are:

- **SynchronizedMap** is a wrapper for a standard hash map that locks the entire collection when a method is invoked. This cannot be used directly as it does not ensure atomic operation between checking a key exists in the map and inserting a new key. This means that the emit method requires synchronisation, creating serial access to the collector.

- **ConcurrentHashMap** is a class from the JDK that implements support for thread-safe and deterministic data insertions. It uses segments to split the hash map, each having their own lock, so unless two threads are accessing the same segment there may be concurrent access to the hash map.

- **NonBlockingHashMap** is an external library and alternative implementation for concurrency using atomic operations rather than synchronisation [78]. A linear-probing hash map is updated by using atomic operations updating one of two arrays holding keys and values.

Appendix B contains a performance evaluation of the three hash maps using representative data sets. The `ConcurrentHashMap` was selected based on its consistent performance and inclusion in the JDK and a synchronised `ArrayList` is used to collect the values. Following the map phase, the list is passed as a `List` interface argument to the reduce method. The use of the interface allows modification of the implementation in the future without changes required in the application. Figure 3.5 contains the MR4J collector architecture to compare to Phoenix and Phoenix++ implementations in Figures 3.3 and 3.4 respectively.
CHAPTER 3. CO-DESIGN FOR MAPREDUCE

ConcurrentHashMap

if ( values = get( key ) == null )
values = new list
putIfAbsent( key, values )
values.add( value )

Figure 3.5: MR4J intermediate (key, value) pairs collector.

MR4J Application

To demonstrate the use of MR4J, Figure 3.6 contains the example of word count. It assumes that the input document has already been divided into String chunks and collected in a list. The framework may then be instantiated with the map and reduce methods, (Line 23) providing types for the input and key (both String) and the value (an Integer). Primitive types cannot be used for these because of the restrictions of generics in Java.

The map phase is expected to emit each word as a key with a value of one; the implementation for the interface can be found on Lines 4–9; the Emitter interface provides the ability to pass the (key, value) pairs to the framework (Line 7).

In the word count example the reduce method sums the values for each key and in this framework it is provided as a list of references to the constant 1. The implementation on Lines 13–19 creates the sum in an intermediate int variable. At this point, the framework has both the input and the implementation of required methods. On Line 24 the run method is, subsequently, called in order to start execution and calculate the result. It then returns a list of (key, value) pairs containing the final data when complete.
public class WordCount {
    private static final Pattern WORD =
        Pattern.compile("[A-Z][A-Z']*");
    private final Mapper<String, String, Integer> mapper =
        new Mapper<String, String, Integer>() {
            public void map(String input,
                Emitter<String, Integer> emitter) {
                Matcher words = WORD.matcher(input.toUpperCase());
                while (words.find()) {
                    emitter.emit(words.group(), 1);
                }
            }
        };
    private final Reducer<String, Integer> reducer =
        new Reducer<String, Integer>() {
            public void reduce(String key,
                List<Integer> values,
                Emitter<String, Integer> emitter) {
                int sum = 0;
                for (Integer value : values) {
                    sum += value;
                }
                emitter.emit(key, sum);
            }
        };

    public List<KeyValue<String, Integer>>
        run(List<String> input) {
        MapReduce<String, String, Integer> mrj =
            new MapReduce<>((mapper, reducer));
        return mrj.run(input);
    }
}
Design Considerations

MapReduce was originally developed with a simple API, but this clean interface was lost in Phoenix and even more so in Phoenix++. MR4J contains two methods; the constructor and the run method to execute the framework. The map, reduce and sort methods are implemented within interfaces to enforce the correct syntax and types at compile time. This avoids the programming problems associated with void pointer usage in Phoenix 2.0 and the complexity of the declarations in Phoenix++. There are three generic type declarations in this Java implementation: the input, key and value types. This compares to none in Phoenix because it is not possible in C (relying on void pointers instead) and a minimum of five in Phoenix++, although the container requires an additional four types to be defined. This allows the Phoenix framework to be highly customised, but with some duplication and exposure to the implementation. It is not a trivial task to create a new application. The complexity introduced in the evolution of Phoenix is something that is avoided in MR4J and instead relies on the optimisations provided by the JVM. This solution does not prevent the possibility of using shared, mutable state but allows developers to bypass the data-parallel nature of MapReduce and introduce errors in the software. When the map function does not write to a shared object and developers follow the guidance the framework will execute as described in this thesis.

MR4J is accessed as a class API and the default constructor will use all available processors; however it can be limited by setting a parallelism argument. This enables several MapReduce applications to run in parallel although developers should be careful not to overload the cores; for example a 64 core system may run a word count on 16 cores and a matrix multiplication on the remaining 48. This is possible in Phoenix although the number of cores and the offset for core allocation must be specified, adding some complexity to an application.

Finally, the Java code is portable to any hardware architecture that supports a Java Runtime Environment (JRE). The class files may be copied to any system and used without modification. This is in contrast to Phoenix that contains a GNU Make script to build a copy for each operating system or architecture. New architectures will require support for atomic and processor assembly code for both C and C++. Due to the decision to use the concurrent hash map the MapReduce framework for Java is reliant on no other libraries other than those available in Java SE 1.8, improving the portability. Phoenix and, more so, Phoenix++ rely on
external libraries, such as pthreads [42], and intermediate additions to the C++ standard libraries before inclusion in C++11 [79]. The productivity of porting and running Java is an important feature and was a focus during the design of MR4J.

3.4 Semantically-Aware Optimisation

Accompanying the development of the MR4J is a novel approach to optimisation. As discussed, the semantic distance between the map and the reduce phases is too great for the JVM dynamic compiler to bridge. The optimisation should be aware of the semantics of both the framework and application, i.e. semantically-aware optimisation.

3.4.1 Optimisation Options

There are two mainstream options available to rewrite classes at runtime. Both have been evaluated informally for the scope at which they operate and the ease at which they may be used. After evaluating aspect-oriented programming, its use is targeting different class modifications but instrumentation is a more suitable approach for MR4J.

Aspect-Oriented Programming

Kiczales et al. [80] observed a separation in concerns in software. The functionality of an application is interspersed with common behaviours such as logging, data verification and validation. The cross-cutting concerns may be abstracted and form aspects in Aspect-Oriented Programming (AOP). An application without the cross-cutting concerns may be statically compiled (at compile or runtime) with aspects to allow a fully augmented application to run. In AspectJ [81], an AOP language for Java, aspects enable the re-writing of bytecode within classes statically or at run-time.

While exploring the potential for generating the combiner code, AspectJ fits the philosophy of splitting the concerns between performance and productivity. However it became clear that while it has a suitable level of abstraction, the low-level functionality of a method is not available. This tool is more useful for trimming or augmenting classes with features, not restructuring them.
CHAPTER 3. CO-DESIGN FOR MAPREDUCE

Instrumentation

Java provides an \texttt{java.lang.Instrumentation} interface that, when implemented and registered with the JVM, gives access to class files as they are loaded. The class may be rewritten before committed to the runtime as part of a Java agent \cite{3}. AspectJ uses this mechanism for runtime inclusion of aspects, other examples include profiling (JaCoCo \cite{82}) and Software Transaction Memory (STM) (Deuce STM \cite{83}).

Instrumentation using a Java agent allows a similar approach to AOP but with full access to bytecode and all features available in a class. There are tools available to edit class files such as ASM \cite{84} and BCEL \cite{85}. Kuleshov \cite{86} developed ASM to transform commonly occurring patterns when dealing with class files. However it uses the visitor pattern to achieve its objectives and is designed for like-for-like replacements. While instrumentation is the correct abstraction there are no tools available to achieve the aims of the MR4J optimisation directly.

3.4.2 Implementation

The concept of a combiner method to improve the locality of reducing data was first mentioned in the original Google MapReduce framework \cite{52}. Its purpose is to combine emitted values locally on a processing node in order to limit the data transferred, before and during the reduce phase. In the multi-core implementations, with direct access to all (key, value) pairs, it is possible to eliminate the reduce phase altogether. Figure 3.7 illustrates how the word count application can achieve this with a simple accumulator (an initial value of zero is assumed).

In related frameworks, this optimisation is manual and the user is responsible for implementing it for the framework. In many cases, such as all benchmarks used except K-Means Clustering, the combine method may be implemented directly in the method. The example in Figure 2.6 illustrates the duplication and may be rewritten as in Figure 3.8.

Rewriting a class to automatically move the combining functionality from the reduce method to the map phase is the objective in the implementation. The benefits are that it will reduce the source code written: eliminating duplication; limiting the scope for bugs; and improving the performance of benchmarks where a combine method has been overlooked. The principle of \textit{code motion} is to move
map(partition)
    for word in partition
        emit(word, 1)

combine(sum, value)
    return sum + value

Figure 3.7: MapReduce using a combiner to eliminate the reduce phase.

```cpp
void reducer(void *key_, iterator_t *itr_)
{
    emit(key, combiner(itr_));
}
```

Figure 3.8: An alternative word count reducer in Phoenix 2.0.

code beyond the boundaries of its abstraction and to allow better low-level optimisation in applications. This is the technique, with class file manipulation, this optimisation aims to exploit.

**MR4J Modifications**

Figure 3.9 illustrates the desired transformations in the context of MR4J to replace the reduce execution flow with combining. The primary change is to provide an intermediate (key, value) pair collector that is aware of combining values. To simplify the abstraction in Java, the intermediate value is held in a package private encapsulating object (a `Holder`). The same collector strategy is employed, the thread-safe hash map, but with a different implementation behind the public API (the `Emitter`).

Before the modification, a new key would instantiate a new list to collect values. In the optimised execution flow, a new key will instantiate a new holder and be stored in the collector. Each value will be combined with the intermediate value help in the holder. Finally, before the results are returned to the user, a finalisation method will convert the intermediate value into the resulting value.
This requires the runtime transformation to generate three methods and to set a flag enabling the optimisation.

**Runtime Transformation**

When the transformation is realised as a Java agent, it produces the bytecode shown in Figure 3.10. The reduce method is analysed to create an intermediate representation that identifies three bytecode blocks that will map onto the three methods required to implement the combiner in MR4J. The three blocks originating in the reduce method code body are before, within and after the loop iterating over all the values from the list provided as an argument. The code before the loop provides detail about the type of the intermediate value (object or primitive). The code within the loop, which is the body of the combiner, should be dependent only on the intermediate and current iteration value. The code after the loop provides the conversion code of the intermediate value into its final form. The purpose of each generated method is:

- **Holder initialise();**
  This method provides an initial intermediate representation for values as the holder. In the case of all types, it will provide a mutable boxing class. The initial value is formed by moving the code from the reduce method initialisation block (Line 2 in the example of Figure 3.9).

- **void combine(Holder, V);**
  This method contains the code from the reduce method that implements the combining. The mutable value in the holder is modified to include the information required from the emitted value. In Figure 3.9 (Line 4) this is the addition of the count to the accumulator.

- **V finalise(Holder);**
  This method converts the intermediate representation of the value into its final form. In Figure 3.9 this is the boxing of the int to an Integer auto-boxed in the code (Line 6).

Due to the implementation of Java generics, the combine and finalise methods require a generated synthetic bridge method to act as an interface due to type erasure. The methods ensure that type information is not erased from user code and the correct type is associated with objects on the stack during execution. These have been omitted from Figure 3.10 for brevity.
a) Existing Method

```java
public void reduce(String key,
        List<String> values,
        Emitter<String, Integer> emitter)
{
    int sum = 0;
    for (Integer value : values) {
        sum += value;
    }
    emitter.emit(key, sum);
}
```

b) Generated Methods

```java
public int initialise() {
    return 0;
}

public int combine(int sum, Integer value) {
    return sum + value;
}

public Integer finalise(int sum) {
    return sum;
}

public boolean isCombinable() {
    return true;
}
```

Figure 3.9: Generated methods for the Reducer class transform.
void reduce(String key, List<Integer> values, Emitter<> emitter) {
    int sum = 0;
    for (Integer value : values) {
        sum += value.intValue();
    }
    emitter.emit(key, Integer.valueOf(sum));
}

CHAPTER 3. CO-DESIGN FOR MAPREDUCE

Figure 3.10: Bytecode transform of the reducer for word count in MR4J.
a) Counting number of values, regardless of content (e.g. String Match)
    emitter.emit(key, values.size());

b) Assuming only a single value is stored per key (e.g. PCA)
    emitter.emit(key, values.get(0));

Figure 3.11: Idiomatic reducers for transformation.

Implementation

A Java agent [3] was chosen as the most suitable technique to generate the new methods since it is simple to identify implementations of the reduce method. The first step was to create an alternative execution flow in the MapReduce framework that uses the generated methods that are hidden from the user, i.e. they contain no functionality and cannot be accessed or overridden outside of the declared package. When the class loader loads the reduce class it rewrites the access to these methods so they can be overridden at runtime.

Before the more complex avenue of transformation is taken, the reduce method is tested against a few idioms (Figure 3.11) and transformed accordingly. The process for transforming the code follows the following steps:

1. Parse the reduce method to create an intermediate representation of the code in the control-flow dependency graph.

2. Identify the conditions of the loop iterating over the values to ensure coverage of all values provided by the framework.

3. Test that the initialisation block contains no external data dependencies. Determine the holder type required and copy adjusted bytecodes to the initialise method body.

4. Test the value iteration loop body for data dependencies (assuming that the operation is associative due to the semantics of the MapReduce framework). Copy adjusted bytecode to the combine method body.

5. Identify the original bytecode relating to the finalisation of the intermediate value, from the preparation of the stack for the emit method call. Copy adjusted bytecode to the finalise method body.
6. Set the flag to return a constant of true rather than false to enable the optimised combining execution flow in the MR4J implementation.

Should all these steps be followed successfully the reducer class for the word count will appear as in Figure 3.10. The adjustments to the bytecodes are to replace the access to the intermediate values from the method local with access to the field in the Holder and to update the value index to use the argument.

3.5 Performance Evaluation

MR4J is evaluated in two stages. The first stage explores: a) the scalability of MR4J on two different hardware configurations, and b) the comparative evaluation of MR4J against mature and hand-optimised state-of-the-art implementations in C and C++, Phoenix and Phoenix++ respectively. The second stage evaluates the performance benefits generated by the semantically-aware optimiser.

3.5.1 Experimental Set-up

Hardware Platforms

The experiments were run on two different hardware platforms in order to explore the performance on a multi-core workstation and a larger NUMA multi-socket, multi-core server. Table 3.1 presents the hardware and software configurations used to host the evaluations.

MapReduce Software Frameworks

The evaluation compares the presented lightweight MR4J with the co-designed optimiser framework against the hand-tuned Phoenix [63] and Phoenix++ [64] implementations. These are both configured manually using hardware specific parameters; e.g. the size of L1 cache and the number of desired threads. MR4J uses the same L1 cache size as its buffer size and the JVM is configured to use the default garbage collector (Parallel) with an initial and maximum heap size of 12GB. Furthermore, the -XX:+UseNUMA flag is set for the server configuration. Each benchmark is executed ten times (Java includes a five iteration warm-up) and the average execution time is used to generate results.
CHAPTER 3. CO-DESIGN FOR MAPREDUCE

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Workstation</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Core i7</td>
<td>AMD Opteron</td>
</tr>
<tr>
<td>4770 3.4GHz</td>
<td>6276 2.3Ghz</td>
<td></td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
<td>64 (4 × 16)</td>
</tr>
<tr>
<td>Hardware threads</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32kB per core</td>
<td>16kB per core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256kB per core</td>
<td>2MB per 2 cores</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8MB per 4 cores</td>
<td>8MB per 8 cores</td>
</tr>
<tr>
<td>Main memory</td>
<td>16GB</td>
<td>264GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Windows 8.1</td>
<td>Ubuntu 12.04</td>
</tr>
<tr>
<td>C/C++ compiler</td>
<td>gcc 4.8.3</td>
<td>gcc 4.6.4</td>
</tr>
<tr>
<td>Java</td>
<td>Java SE 1.8.0.20</td>
<td></td>
</tr>
<tr>
<td>JVM</td>
<td>Java HotSpot 64-Bit Server (build 25.20-b23)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Configurations for MR4J evaluation.

Benchmarks

The benchmarks distributed and used by Phoenix and Phoenix++ have been ported and validated against MR4J for a fair comparison. The benchmark suite consists of the following applications:

- **Histogram (HG)**
  Counts the occurrences of the 8-bit pixel intensities for each of the three colours (red, green and blue) within a bitmap image.

- **K-Means Clustering (KM)**
  An iterative algorithm to cluster points distributed in space. Initially a series of means, the centre of each cluster, are generated randomly in the same space. Each iteration then calculates the mean closest to each point and then sets the cluster mean for the next iteration to the centre of its associated points. This continues until an iteration is completed where no points change their associated mean.

- **Linear Regression (LR)**
  Provides a linear relationship between variables by analysing the input set containing a single dependent variable and a single independent variable.
• Matrix Multiply (MM)
  Two square matrices are multiplied together; a row per map task and the re-
  sultant matrix is then summarised by summing all the individual elements.

• Principal Component Analysis (PC)
  Provides a set of variables that are the most prominent in a set of results.
  This process occurs over two phases: the first calculates the mean of each row
  and the second establishes the correlation between each mean and the dataset.

• String Match (SM)
  Matches a predetermined set of keys against lines of a bespoke plain text file.
  Both the key and the input lines are hashed before comparison.

• Word Count (WC)
  Counts the occurrence of each unique word in a plain text document. The com-
  parison is case-insensitive and considers a word a letter followed by any other
  sequence of letters or apostrophes.

In order to ensure that the same algorithms are executed across all three
frameworks, modifications have been made to the original benchmarks. For His-
togram, Phoenix++ iterates over individual pixels; however due to performance
and memory constraints, Phoenix and MR4J iterate over chunks of data, emit-
ting values after partial combination in the map method. Histogram and Word
Count omit the requirement to sort the keys because this is testing the efficiency
of parallel sorting algorithms rather than the core of MapReduce.

The benchmarks demonstrate a variety of workloads, inputs, intermediate
and output results. All of these benchmarks, originally from the Phoenix paper
[63], contain combiner methods. For MR4J these combiners are all generated by
the optimiser described. The challenge for all three frameworks was to generate
a combiner for the K-Means Clustering benchmark because it requires state to
obtain the average (e.g. the total number of points in a cluster). In this case, the
combiner or the intermediate value contain the running sum of point co-ordinates.
The sum is divided by the number of points in the reducer for MR4J or in the
main body of the application for Phoenix and Phoenix++. Table 3.2 presents
the input data sets with an approximate categorisation of the volume of keys and
values.
Table 3.2: Input data sets for the benchmarks.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>HG</td>
<td>Medium</td>
<td>Large</td>
</tr>
<tr>
<td>KM</td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td>LR</td>
<td>Small</td>
<td>Large</td>
</tr>
<tr>
<td>MM</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>PC</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>SM</td>
<td>Small</td>
<td>Small</td>
</tr>
<tr>
<td>WC</td>
<td>Large</td>
<td>Large</td>
</tr>
</tbody>
</table>

Table 3.3: MR4J server configuration performance groups.

<table>
<thead>
<tr>
<th>Scalability</th>
<th>Benchmarks</th>
<th>Peak Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>KM, LR, MM</td>
<td>25.31</td>
</tr>
<tr>
<td>Moderate</td>
<td>PC, SM</td>
<td>17.72</td>
</tr>
<tr>
<td>Limited</td>
<td>HG, WC</td>
<td>4.85</td>
</tr>
</tbody>
</table>

3.5.2 Performance Results

The scalability of MR4J can be seen in Figures 3.12 and 3.13 for the workstation and server configurations, respectively. With a baseline of the execution time on a single core, the workstation shows a consistent scalability over all hardware threads, with an average of 2.85 on four cores and 3.73 on all eight hyper-threads. Regarding the scalability of MR4J on the server configuration (Figure 3.13), three groups of performance can be observed depending on their computational intensity and overhead of (key, value) pair generation. Table 3.3 contains the three groups of performance.

Figures 3.14 and 3.15 contain the speedup of MR4J and Phoenix relative to Phoenix++ on the workstation and server configurations respectively. Furthermore, Figures 3.16 and 3.17 take a more fine-grain approach and illustrate the relative speedup of MR4J against Phoenix++ with and without the implemented optimiser respectively per benchmark. Regarding the workstation configuration, a consistent performance behaviour can be observed between MR4J, Phoenix and Phoenix++. The performance of MR4J falls between the two hand-optimised frameworks with a median of 0.66 for MR4J and 0.39 for Phoenix for all hardware thread counts.

The server configuration reveals a different set of results, illustrating the
Figure 3.12: MR4J benchmark scalability on the workstation.

Figure 3.13: MR4J benchmark scalability on the server.
Figure 3.14: Relative performance against Phoenix++ on the workstation.

Figure 3.15: Relative performance against Phoenix++ on the server.
challenges of developing scalable software for multi-socket NUMA architectures. When using the same socket (1–16 threads) the performance of MR4J and Phoenix is comparative to Phoenix++ which consistently out-performs them (0.61 and 0.81 respectively). Scalability was a primary objective in the development of Phoenix++ [64] and the results are supported by this evaluation. The NUMA aware setting in the JVM is able to maintain a consistent level of performance, unlike Phoenix which employs only its locality optimisations. However, the speedups of MR4J and Phoenix are 0.76 and 0.20 compared to Phoenix++ when using all hardware threads.

### 3.5.3 Optimisation Performance

Figures 3.16 and 3.17 illustrate the relative speedup of MR4J against Phoenix++ before and after the optimiser is enabled for each of the benchmarks. Figure 3.16 contains the same plot at two different scales to illustrate the scale of performance for the Linear Regression and Matrix Multiplication benchmarks. The majority of the benchmarks on both configurations show a significant speedup, thus, closing the gap between MR4J and Phoenix++. String Match is an exception, exposing the overheads of instantiating and maintaining the intermediate value. This is
due to the nature of the benchmark which has few keys, few values and little computation that can be optimised.

The main overheads of the optimiser are when detecting classes that extend the `Reducer` and then generating the combining code. Since the optimiser instruments every Java class, the effect on the detection and transformation times are, on average per class, 81µs and 7.6ms respectively, which is negligible in comparison to the execution time of the benchmarks.

### 3.6 Discussion

MR4J is presented as a lightweight MapReduce framework based on the standard JDK classes. By using a simple API and by utilising Java interfaces it is possible to improve the framework whilst maintaining the backwards compatibility ethos of Java. The optimisation presented illustrates how a single map method can be used in two alternative execution flows (one to reduce values and the other to combine them) as a result of using the `Emitter` interface.

On a multi-core architecture, MR4J provides consistently better execution times than the hand-optimised C equivalent and, after optimisation, is within
CHAPTER 3. CO-DESIGN FOR MAPREDUCE

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Container</th>
<th>Combiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>array</td>
<td>sum</td>
</tr>
<tr>
<td>K-means Clustering</td>
<td>array</td>
<td>custom</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>array</td>
<td>sum</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>PCA</td>
<td>common array</td>
<td>one</td>
</tr>
<tr>
<td>String Match</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Word Count</td>
<td>hash table</td>
<td>sum</td>
</tr>
</tbody>
</table>

Table 3.4: Containers and combiners used in Phoenix++ benchmarks.

reach of the equivalent in C++. Phoenix and Phoenix++ offer powerful and scalable tools but with more complicated APIs that require manual configuration and tuning. The benchmarks where MR4J is superior are those where data is organised in arrays. The dynamic compiler is able to optimise better array accesses (by pre-fetching memory into cache before it is used) than pointer arithmetic alone in a static compiler. However, the benchmarks where heavy object creation is required, the ability of C and C++ to cast directly to data highlights the overhead of object allocation and management in Java. K-Means Clustering with Points and Word Count with Strings are such examples.

The gap in performance between MR4J and Phoenix++ for PCA can be explained by the use of pre-allocated memory to store the combiners for the means and covariances. The map methods for each task emit a value into a one_combiner that writes directly to memory, without the need for any synchronisation, using a common array container statically compiled into the application. The benchmark on C++ demonstrates that by communicating nearly all application semantics to the framework much of the overhead may be removed. The gap in execution time is the time required by Java to maintain the intermediate collector and objects.

Table 3.4 lists the containers and combiners used by the Phoenix++ benchmarks. The gap in MR4J performance in the histogram and k-means clustering benchmarks may be explained by the intensity of interaction with the collector. Phoenix++ is able to access each combiner directly as they are pre-allocated in arrays, with minimal synchronisation required due to the embedding of the implementation in the application by use of the CRTP. Linear regression, the final benchmark using an array container, is compute bound so is better optimised by Java and interacts less with the overheads introduced in MapReduce. There is
a compromise made in giving the user full control to produce efficient code but increasing complexity (although the benchmark is not an efficient algorithm for PCA and much of the effort is duplicated). MR4J is a simple, general purpose API without specialisation; therefore there is a restriction in performance that in the PCA benchmark is unsuitable for bypass.

The optimisation presented in this chapter changes the execution flow within the framework. Borrowing the notion of manual combining from existing MapReduce frameworks, the implemented optimiser automates this process at runtime. The optimiser uses the semantics of the framework and the structure of user code to eliminate the reduce phase and combines intermediate values as they are emitted from the map method. This has the effect of improving the execution time for the majority of the tested benchmarks.

The cause of the observed speedup is the improved interaction between the optimised executed code, the dynamic compiler and the GC. Figures 3.18 and 3.19 visualise the heap usage for the word count application without and with the optimiser respectively. The execution time axes are the same for a direct comparison. The heap usage is similar for both configurations showing a noticeable and steady increase in the size of the heap used because more references

Figure 3.18: GC behaviour for MR4J word count on workstation.
Figure 3.19: GC behaviour for optimised MR4J word count on workstation.

are stored for the intermediate values. The stark difference is in the secondary axis, the time spent in the GC. Without the optimisation, the inefficiency lies in the fact that Java must maintain (i.e. keep on the heap) all the object references for the intermediate values generated during the map phase. This results in their premature promotion into the older generations before they are marked for collection (and collected during minor collections). Consequently, this results in major collections that severely affects performance. The optimisation, in turn, increases performance by:

- Reducing the number of objects allocated which avoids unnecessary object promotions that lead to major GC cycles.
- Improving execution time by omitting completely the reduce phase.
- Enabling the dynamic compiler to introduce additional scalar replacements.
- Reducing the utilised heap size and, thus, enabling larger data sets to be used, increasing the potential for utilising smaller ‘big data’ jobs (as mentioned in the Hadoop job analysis [87]).
Figure 3.20: Relative performance of MR4J optimiser on GC configurations.

<table>
<thead>
<tr>
<th>GC</th>
<th>Default (4GB)</th>
<th>12GB Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>1.29</td>
<td>1.18</td>
</tr>
<tr>
<td>Parallel</td>
<td>1.97</td>
<td>1.19</td>
</tr>
<tr>
<td>OldParallel</td>
<td>1.94</td>
<td>1.18</td>
</tr>
<tr>
<td>CMS</td>
<td>1.39</td>
<td>1.37</td>
</tr>
<tr>
<td>CMS (parallel copy)</td>
<td>1.27</td>
<td>1.21</td>
</tr>
<tr>
<td>G1</td>
<td>1.40</td>
<td>1.30</td>
</tr>
</tbody>
</table>

Table 3.5: MR4J optimisation speedup for GC configurations.
The JVM, as publicly distributed by Oracle [7], contains a variety of GC algorithms allowing different tuning parameters and configurations. As with MR-J [77] the GC is the subject of performance evaluation for MapReduce in Java. Whereas they optimise performance by auto-tuning, MR4J optimisation reduces the interaction with the GC. In order to provide a better insight of how the proposed optimisation interacts with the memory allocator, all available GCs were tested. The six GCs in the Java SE 1.8 [7] distribution were all used twice, once with the default settings and the other with a large heap size to minimise collections and resizing. The results are presented in Table 3.5 and Figure 3.20.

Table 3.5 shows the relative average speedup achieved by the optimisation for different heap sizes per GC algorithm (1–8 threads). As expected, with smaller heap sizes (such as the default configuration with parameters \(-X\text{ms}256m\) and \(-X\text{mx}4g\)) the optimisation provides a better speedup over the baseline. The observed speedups vary from 1.27 (CMS-Parallel Copy) to 1.97 (Parallel). For larger heap sizes (where the GC pressure is smaller) the speedup varies from 1.18 (Serial) to 1.37 (CMS).

Figure 3.20 depicts the relative speedup (to the baseline unoptimised version) of each benchmark when all the combinations of GC algorithms, heap sizes, and number of hyper-threads are averaged. The figure also shows that the benchmarks with the greatest reliance on (key, value) pairs (HG and WC) are improved the most. String Match has four keys with 910 values; whereas Histogram has 768 keys and 1,406,250,000 values.

### 3.7 Summary

This chapter introduces MR4J, a lightweight Java based MapReduce framework for shared-memory, multi-core architectures built on standard JDK classes. MR4J is a lightweight Java based MapReduce framework for shared-memory multi-core architectures built on standard JDK classes. It focuses on ease-of-programming via a simple API in contrast to equivalent frameworks where performance is extracted via hand-optimisation and tuning by developers. By co-designing a semantically-aware optimiser it is possible to exploit the data-dependencies between the input and the result that are not available to the existing compiler in the JVM. The performance loss, due to its simplicity, is overcome by a novel optimiser built for the framework. The optimiser exploits semantic information
inherently contained within the parallel software framework transparently to the user. The design of MR4J aims to either supplement developers of large MapReduce algorithms, improve productivity or simply execute smaller applications.

The performance of MR4J is comparative to the equivalent state-of-the-art Phoenix framework, written and hand-optimised in C. Thanks to the expressiveness, type safety and portability of Java, it creates a more productive and portable framework with comparative performance. The original implementation of MR4J was positioned in between the two state-of-the-art MapReduce frameworks, Phoenix and Phoenix++, in terms of performance. The lack of a combiner phase was penalising performance and therefore the optimiser was implemented to supplement the framework. The co-designed optimiser presented automates the, previously hand-optimised, combining phase in order to improve performance. Without any modifications to user code, the optimised MR4J improves its performance up to 2.0 times, reducing the gap from the manually-tuned Phoenix++ to just 17%.

It shows that it is possible to re-think the way in which parallel frameworks may be developed. Co-designing allows an alternative communication path to reduce the semantic distance between phases in a parallel framework. This removes the need to extend and specialise the API and removes the need for additional expertise, or to upgrade existing applications. This chapter demonstrates an improvement in performance, especially for naïve configurations, without changing the programming interface, i.e. no hand-optimisation. MapReduce is simple in its semantics so it should be asked whether this novel approach may be extended to different layers in the software stack and a more general programming environment.
Chapter 4

Background: SLAM

Simultaneous Location and Mapping (SLAM) is a key algorithm for robotics and other autonomous devices; computing the location of an input sensor within an environment. Inputs include monocular cameras found in commodity web-cams, laser ranging sensors or wide-angle time-of-flight camera as is found in the Microsoft Kinect [88]. SLAM is used in this thesis because of the complexities and scale of the computational challenge. The volume of data processed requires many heuristics for efficient algorithms and implementations; resulting in a selection of SLAM applications. Captured inputs from a red-green-blue (RGB) standard-definition web-camera contain just over 900kB of data (24-bit colour, 640 × 480) whereas the Kinect is processing 2Gb of data per second to generate depth information. Hardware acceleration is used to increase the throughput of data and this is supported by a selection of libraries and algorithms. Existing benchmarks use computer vision applications to test systems; the PARSEC Benchmark Suite contains a body tracking application [89] and SLAMBench has been developed to evaluate many-core heterogeneous environments [90].

Many implementations of SLAM are written in C++ or MATLAB; the first targets performance, whether execution time or throughput; and the latter, productivity by relying on the mathematical expressiveness of the inherent matrix-based algorithms of SLAM. The libraries used in C++ implementations provide a data-level abstraction for commonly used numerical methods. They attempt to compile high-performance machine code with an expressive API. However as the data is publicly accessible it is possible to bypass encapsulation and many hand-optimisations appear in application code. The balance between expressiveness
and performance is what the remainder of this thesis explores, as means of addressing the loss in performance from a strict API that reduces opportunities for hand-optimisation. This thesis focusses on Large-Scale Direct Monocular SLAM (LSD-SLAM) [17] in particular, and the computational use of its key algorithms investigated to co-designed specialised optimisations.

4.1 SLAM

SLAM is a category of application within computer vision that, as the name suggests, maps an environment while tracking the location of the camera. SLAM applications are often extensions of visual odometry, which are techniques for tracking changes in position over time from input images. They use one of several input types including laser ranging equipment, monocular cameras, stereo cameras or depth cameras. In the case of KinectFusion, using the Microsoft Kinect [88], a monocular camera is combined with data from a wide-angle time-of-flight camera generating depth for the observed scene. LSD-SLAM uses a monocular camera with grey-scale values for each pixel, some SLAM applications extend existing implementations so may use the output from another as their input. The output from the applications depends on its use, although a secondary format is used to allow visualisation to demonstrate functionality. The main SLAM application outputs are:

- **Point clouds**, sparse set of points in a 3D environment representing surfaces, edges or corners;
- **grid**, is similar to point clouds but for 2D mapping;
- **a path** recording the trajectory of the camera; or
- **as with the KinectFusion algorithm and its variants**, the environment is a **volumetric model** constructed from small three-dimensional blocks (**voxels**).

4.1.1 Overview

To introduce SLAM, this chapter explains LSD-SLAM as way of example; it is a state-of-the-art application to map small and large environments with the same algorithm. It contains many recurring algorithms that are common to
other SLAM applications and these are used to evaluate the performance of co-designed optimisation presented in Chapters 6 and 7. The focus of LSD-SLAM is to provide relative position, with scale, in a self-optimising graph representing the current environment [17]. In contrast Semi-dense Visual Odometry (SVO) is optimised for rapid positional tracking [16] and the KinectFusion algorithm [18], the basis for SLAMBench [90], provides absolute position in an precise model of the environment. LSD-SLAM is of interest as it combines the techniques used by many of the 3D SLAM applications introduced in Table 4.1. The application has been parallelised to improve its performance; despite this being an opportunity for improvement in the Java implementation discussed in Chapter 5, this is not addressed in this thesis. The Java implementation will be used to assess optimisation and acceleration of LSD-SLAM using co-design with the Graal optimising compiler [5].

Figure 4.1 illustrates the interaction of the three key algorithms forming LSD-SLAM. The algorithms are implemented as independent threads that execute when data is available. The tracking algorithm 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 must calculate its own depth information and does this using depth estimation, comparing the difference in position of pixels in two separate frames. Each key-frame maintains its own depth information and is
\(\xi \leftarrow \text{initial pose estimate}\)
\(lgs \leftarrow \text{Levenberg-Marquardt}\)
\(\text{for point in keyframe}\)
  \(p \leftarrow \text{project(camera,} \xi, \text{point)}\)
  \(\text{residual} \leftarrow \text{error}_{\text{photometric}}(\text{keyframe point, frame}_p)\)
  \(\xi_p \leftarrow \text{pose}(p, \text{frame}_p)\)
  \(\text{update}(lgs, \xi_p, \text{residual})\)
\(\xi \leftarrow \text{solve}(lgs)\)

Figure 4.2: The tracking task used in estimate poses in LSD-SLAM.

able to create a point cloud to visualise the environment. The last algorithm is map optimisation that minimises errors in the map of the environment, this includes loop closure in which disjoint, but neighbouring, key-frames are associated in the graph. The three key algorithms run in parallel and the feedback from the depth estimation and map optimisation is used to improve tracking and to update key-frame information.

4.1.2 Tracking

The camera is assumed to be moving so new frames are tracked against the active key-frame. The relative position (pose) between frames is important as it builds the structure of the graph containing the map and it is used to estimate depths. LSD-SLAM is a direct method so it does not detect features in the image, instead to reduce the computation, only pixels that are different to their neighbours are used (the points in the cloud). Other applications may detect edges or corners and these provide the features used to track the current location of the camera. Figure 4.2 shows the processes at work in the tracking task of LSD-SLAM.

The tracking task uses an initial pose (the result of the previous tracked frame) to transform and project a point to a pixel in the image (see Figure 5.1 for the implementation). 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. Together these are used to construct a system that uses the Levenberg-Marquardt algorithm [91] to find the best pose iteratively. Once the pose is no longer changing, the estimate is attached to the frame and then used in depth estimation.
The Levenberg-Marquardt algorithm [91] is summarised by Equation 4.1

\[(J^T J + \lambda \text{diag}(J^T J)) \delta = J^T [y - f(\beta)] \] (4.1)

where:
- \(J\) = Jacobian matrix (weighted pose estimates),
- \(\lambda\) = dampening factor, and
- \(f(\beta)\) = estimated pose (mean of all estimates).

It is used to construct a system that may then be solved to find \(y\), the pose best-fitting the current frame. It is an extension of the Gauss-Newton algorithm ([92] pages 173–227) and is used to solve non-linear problems. The system constructed is a positive definite matrix so may be solved with \(LDL^T\) factorisation, a specialisation of Cholesky factorisation ([93] pages 76–69) removing the need to use square roots. The solution is a six-dimensional vector that may be mapped to an \(se(3)\) Lie group [94] to provide rotation and translation of the frame to its associated key-frame.

### 4.1.3 Depth Estimation

Depth is computed using stereo estimation; using two frames and the pose as the input to the algorithm. Figure 4.3 contains a simplified illustration of the process that is taking place. 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 variation in depths possible for this second point, each is tested and the best match is selected as the depth. To test the match, 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. Each point in the the point cloud is initialised with a random value between 0.5 and 1.5 and uses a large variance (0.125) that increase the length of the vector used to match pixels. The majority of the arithmetic in the implementation of the algorithm is transforming and projecting points and testing 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.
4.1.4 Map Optimisation

The mapping algorithm uses the pose information between key-frames to track points representing the environment. The key-frames are stored in a graph, when they are created, and linked to the previous key-frame. The edges are annotated with the relative pose as a constraint used during optimisation. Over time the errors in tracking accumulate and may cause a duplication of points in the environment being mapped. OpenSLAM [95] contains a selection of libraries for both on-line and off-line map optimisation. LSD-SLAM uses the \textit{g}^2\textit{o} library for optimisation by minimising errors in the graph of key-frames on-line.

Figure 4.4 contains the cost-function used to generate the error that is minimised during the optimisation algorithm. The pose concatenation operator ($\xi_{ij} \circ \xi_{jk}$ in Figure 4.4) uses the mapping between Lie groups and algebra [94] (Figure 4.5). This is done for the poses between neighbouring key-frames using the reference to the world. To reduce the amount of computation required, heuristics are implemented within LSD-SLAM to reduce the number of poses used in the cost-function. There is also another process used during map optimisation to merge key-frames that are observing the same points. It uses the algorithms and
\[ \xi \in \mathfrak{se}(3) \]
\[ \xi_{ki} \equiv \xi_{kj} \circ \xi_{ji} \equiv \log(\exp(\xi_{kj}) \cdot \exp(\xi_{ji})) \]
\[ E(\xi_{W_1}, \ldots, \xi_{W_n}) \equiv \sum_{(\xi_{ji}, \Sigma_{ji}) \in \mathcal{E}} (\xi_{ji} \circ \xi_{W_i}^{-1} \circ \xi_{W_j})^T \Sigma_{ji}^{-1} (\xi_{ji} \circ \xi_{W_i}^{-1} \circ \xi_{W_j}) \]

*Where*:  \( W_n = \text{world frame} \)

Figure 4.4: Cost-function used in the map optimisation for LSD-SLAM.

library based on FABMAP [96], an implementation to achieve loop closure.

## 4.2 Existing SLAM Applications

The variety of SLAM applications (summarised in Table 4.1) provide different specialisations for different problem types. The table is ordered by date of publication, from oldest to newest, and the description contains the input and output category along with the pose used to transform features. These form the inspiration for SLAM kernels used in the performance evaluation in Chapters 6 and Chapters 7.

<table>
<thead>
<tr>
<th>Application</th>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEKF-SLAM [97]</td>
<td>2001</td>
<td>MATLAB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses features to generate a path.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Implements a Compressed Extended Kalman Filter (C-EKF) using a vehicle model to create a path based on 2D points.</td>
</tr>
<tr>
<td>GridSLAM [98]</td>
<td>2003</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses laser ranging data to build a grid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extends Rao-Blackwellized particle filters to maintain a map of 2D points and locality.</td>
</tr>
</tbody>
</table>
### CHAPTER 4. BACKGROUND: SLAM

<table>
<thead>
<tr>
<th>Application</th>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP-SLAM [99]</td>
<td>2003</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses laser ranging data to build a grid. Distances are used to create particles in a 2D point grid using probabilistic techniques to overcome uncertainty in the system.</td>
</tr>
<tr>
<td>GMapping [100]</td>
<td>2007</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses laser ranging data to build a grid. Extends Rao-Blackwellized particle filters to maintain map and locality but reduces the number of 2D points re-sampled and maintained.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses images to create a point cloud. Visual odometry targeting augmented reality using features tracked with SE(3) poses with bundle adjustment.</td>
</tr>
<tr>
<td>2D I-SLSJF [101]</td>
<td>2008</td>
<td>MATLAB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optimises features in a grid by using a Sparse Local Submap Joining Filter (SLSJF) to iteratively match duplicate 2D points.</td>
</tr>
<tr>
<td>EKF MonoSLAM [102]</td>
<td>2008</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses images to create a point cloud. This implements Extended Kalman Filter (EKF) with SE(3) poses using visual odometry with inverse depth and sampling optimisations.</td>
</tr>
<tr>
<td>ro-slam [103]</td>
<td>2008</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optimises features in a grid by adding probabilistic range only data to Rao-Blackwellized particle filters.</td>
</tr>
<tr>
<td>Application</td>
<td>Year</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>UFastSLAM [104]</td>
<td>2008</td>
<td>MATLAB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses <strong>laser ranging data</strong> to build a <strong>grid</strong>. Extends Rao-Blackwellized particle filters with additional unscented filters to maintain a map of <strong>2D points</strong> and locality.</td>
</tr>
<tr>
<td>OpenRatSLAM [105]</td>
<td>2010</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Neural network SLAM application based on the hippocampus of a rat.</td>
</tr>
<tr>
<td>tinySLAM [106]</td>
<td>2010</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses <strong>laser ranging data</strong> to generate a <strong>path</strong>. Implementation of SLAM for <strong>2D points</strong> with very small code base.</td>
</tr>
<tr>
<td>KinectFusion [18]</td>
<td>2011</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses image and depth from a <strong>Kinect</strong> to create a <strong>volumetric model</strong>. Builds a model of a scene from a monocular camera augmented with depth information tracked with <strong>SE(3)</strong> poses.</td>
</tr>
<tr>
<td>OpenSeqSLAM [107]</td>
<td>2012</td>
<td>MATLAB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visual odometry optimisation to provide the best frames from <strong>image</strong> sequences to condition the input.</td>
</tr>
<tr>
<td>RGBDSlam [108]</td>
<td>2012</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses image and depth from a <strong>Kinect</strong> to create a <strong>point cloud</strong>. A colour camera with augmented depth information is used to track and map and is based on <strong>transform</strong> matrices (equivalent to SE(3) poses).</td>
</tr>
<tr>
<td>COP-SLAM [109]</td>
<td>2013</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Closed-form Online Pose-chain (COP) providing graph optimisation for <strong>transform</strong> chains by interpolating corrections.</td>
</tr>
<tr>
<td>Application</td>
<td>Year</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Linear SLAM [110]</td>
<td>2013</td>
<td>MATLAB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Local submap joining of features in a grid using linear algebra to minimise errors.</td>
</tr>
<tr>
<td>LSD-SLAM [17]</td>
<td>2014</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses images to create a point cloud. Direct visual odometry and depth estimation with errors distributed during optimisation using Sim(3) poses that encode the scale of the environment.</td>
</tr>
<tr>
<td>SVO [16]</td>
<td>2014</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses images to create a point cloud. Semi-dense Visual Odometry (SVO) uses features to create a point cloud with an emphasis on speed, the camera is tracked with SE(3) poses.</td>
</tr>
<tr>
<td>InfiniTAM [111]</td>
<td>2015</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses image and depth from a Kinect to create a volumetric model. Extends fusion techniques with a more efficient method for storing volumetric data but still using SE(3) poses.</td>
</tr>
<tr>
<td>ORB-SLAM [112]</td>
<td>2015</td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses images to create a point cloud. Visual odometry based feature tracking with transforms optimised with ‘survival of the fittest’ bundle adjustment.</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of existing SLAM applications.

4.2.1 Software Tools

There are two main languages used to implement SLAM applications introduced in Table 4.1. The first uses libraries based on the statically compiled language C++. This provides access to libraries developed for different computer vision
and numerical applications. The second uses the expressiveness of MATLAB due to the extensive use of matrices and numerical methods used in SLAM applications. In both cases, domain specific knowledge is encapsulated within the libraries supporting implementation and which have evolved over time.

**C and C++**

C++ is popular as it is the language of choice for many support libraries and is ‘close to the metal’ so hand-optimisation is possible. Libraries, such as OpenCV [113], g2o [114], Eigen [115] and Sophus [116], have been implemented and optimised by developers with domain knowledge and expertise. Therefore they contain a rich, expressive API for numerical and computer vision algorithms. These are used as the basis for many SLAM algorithms that target different uses of the applications. Using frameworks based on Eigen, a matrix abstraction, and derived frameworks (e.g. Sophus, a Lie algebra [94] abstraction) provide portability over different platforms (desktop and mobile computing) but hardware acceleration is not immediately available. OpenCV provides an interface with compiled binaries for computer vision kernels, which allows specialisation for hardware. However application-wide optimisation is not available with statically or dynamically linked libraries as the methods cannot be inlined during compilation.

In these frameworks numeric and syntactic techniques are exploited for performance but often at the expense of code maintainability. The implementations are community developed to add functionality and extend hardware support for new architectures, features or accelerators. Despite this, there are occasions when abstractions are bypassed for additional performance and ‘portable libraries’ are specialised for hardware architectures. OpenCV uses Intel Performance Primitives (IPP) [117] and Intel SSE intrinsics [35] to improve performance but there are implementations and compiled binaries for other target hardware. Proprietary libraries also exist for specific hardware; one example is FastCV [118] that is developed by Qualcomm for Snapdragon architectures. Eigen is able to override implementations of tasks specialising for vector units (although Intel SSE intrinsics are the extent of specialisation). Performance is further achieved by using the template meta-language and applying deferred execution. These techniques allow the communication of semantics, albeit statically at compile time, and pre-empting optimisations such as loop unrolling.

The danger in using C++ is that pointers to data are always accessible and
this permits hand-optimisation. Another use of pointers allows ‘quick and dirty’
casting between different abstractions, such as casting between matrix data rep-
resentations in Eigen and OpenCV. Complications may arise because any change
in the data layout used in the matrix library could lead to unforeseen errors. An
example change could be the choice of storing the matrix as row or column major.

MATLAB

MATLAB is a domain specific language for technical computing [119]; its pri-
mary abstraction is the matrix. Numerical methods are supported and highly
optimised when executing in its runtime environment. There is a ‘plug and play’
approach to supporting additional capabilities and external hardware interfaces.
This allows a range of input and output configurations with interoperability with
other applications and languages. The power of MATLAB for SLAM applications
is the ability to develop quickly and prototype algorithms. A computer vision
toolbox [120] provides functions that implement frequently used transforms and
algorithms. Applications may be exported to C, although there may be some
restrictions, for use as standalone applications. The SLAM applications in Ta-
ble 4.1 that use MATLAB are generally demonstrating the properties of their
algorithms.

4.3 Recurring Algorithms

Throughout the implementations of SLAM applications in Table 4.1 there are
recurring themes in the algorithms used. These are introduced with a justification
to the origins of the SLAM kernels used in the performance evaluation of Chapters
6 and 7.

4.3.1 Poses

The transformation, from one image to the next, representing the relative camera
position is, in LSD-SLAM, managed through a pose (rigid body transform). Table
4.1 lists the exact pose used in each implementation, and the transform and SE(3)
Lie group are ubiquitous. The SE(3) pose contains six degrees of freedom, rotation
around and translation in three axes. It is possible to represent the group as a
single vector of six elements (se(3)) by applying a logarithm. Poses contain a
rotation matrix but may instead use a quaternion as the representation as it
requires only four elements, as opposed to nine for the matrix. A quaternion is
a complex number with the imaginary part represented by a 3D vector, in SE(3)
it is always a unit quaternion. This is an operation that has been specialised for
Intel architectures in Eigen [115] and is used as an optimisation in Chapter 7.

The se(3) representation of the pose is useful as it may be used in numerical
methods, in the case of LSD-SLAM this is the Levenberg-Marquardt algorithm
[91]. The mapping between SE(3) and se(3) uses matrix-matrix, matrix-vector
and matrix-scalar multiplication and is affected by the efficiency of the abstrac-
tion used. The vector (Tangent6D in the abstraction used in LSD-SLAM for Java)
is calculated for each tracking algorithm invocation and is constructed from the
transformed points and gradients during estimation iterations. Map optimisa-
tion frequently invokes the logarithm and exponential (Figure 4.5 contains the
mapping) and is why the algorithm has formed one of the SLAM kernels.

4.3.2 Image Representation

The input to LSD-SLAM and other applications listed in Table 4.1 are images in a
range of resolutions. The luminosity of RGB colours or grey-scale are represented
as elements of vectors with values ranging from 0–255, 0.0–255.0 or normalised
(0.0–1.0). As the input data is large there is a tendency to use a smaller, com-
pressed versions of the image to obtain initial estimates. Image pyramids are
created and each layer contains half the resolution of the previous: convolution
is used to compress four pixels into one. The number of layers varies but five are generated in LSD-SLAM, although the layers used changes depending on the algorithm executed.

The operations used to generate the pyramid is regular: convolution is used to blur and resize the images. As data is stored contiguously in memory, they are suitable for processing either on GPGPUs or in vector units. Convolution is perfectly parallel and are ubiquitous in graphics processing; however most operations are implemented manually in C++ as the data abstraction and desire for portability restricts access to these technologies. Jacc [121] is an optimising compiler, based on Soot [122], that allows portable Java code to be specialised for GPGPUs. Within the context of this research, pyramids are important as they affect the iterations used in tracking. Frames are assigned an initial pose, relative to the current key-frame and this is improved for each layer of the pyramid. This reduces the computation required to estimate a pose relatively compared to using an image at its original resolution. Each iteration contains three of the four SLAM kernels and these are executed many times per frame and consume much of the time used during tracking.

4.3.3 Numerical Methods

Linear algebra is used to estimate poses in SLAM applications and to optimise the graph representing the point cloud in LSD-SLAM. Systems are constructed from pose estimations and the residual calculated from a cost-function. LSD-SLAM uses $LDL^T$ factorisation to estimate poses whereas KinectFusion implemented is SLAMBench [90] uses Singular Value Decomposition (SVD) factorisation. When implemented in Java (Chapter 5) there is a disparity between the scale of the system and existing libraries available. JAMA [123], Colt [124] and Parallel Colt [125] assume large inputs for high-performance benchmarking such as LINPACK [126]. The overhead of the matrix and vector abstractions used in these frameworks is inefficient for SLAM applications and so Chapter 6 reduces the overhead of numerical methods in small vectors.

LSD-SLAM constructs a system to find an estimated pose based on the $\mathfrak{so}(3)$ Lie algebra for tracking and $\text{Sim}(3)$ for map optimisation. The groups are equivalent to a $4 \times 4$ matrix but are transformed into a vector when the logarithm is applied. Two of the SLAM kernels are based on this process as they test their execution using existing matrix abstractions. The kernels contain the logarithm
for SE(3) and the construction of the system during the Levenberg-Marquardt algorithm [91].

4.4 Summary

This chapter has introduced SLAM as a sub-domain of computer vision with a variety of applications solving different problems. SLAM implementations are written in C++ or MATLAB, using libraries to provide an abstraction of the types used and commonly used algorithms, including linear algebra. However, existing approaches in C++, as used by LSD-SLAM, expose the implementation and permit hand-optimisation. Table 4.1 describes implementation of SLAM for a range of applications leading to the identification recurring computations. Poses and linear algebra are frequently used and form the basis for SLAM kernels used in the performance evaluation of Chapters 6 and 7. Chapter 5 introduces an implementation of LSD-SLAM in Java to investigate the inefficiencies in execution of SLAM applications in a managed runtime environment.
Chapter 5

LSD-SLAM in Java

This chapter provides the motivation for creating a new implementation of LSD-SLAM, porting the application from C++ to Java. It is motivated by the trend towards heterogeneous architectures with dynamic availability of resources and also the benefits associated with dynamic compilation. Graal is used as the optimising compiler for the JVM executing the application. This allows the exploration of semantic communication of domain specific knowledge that is not currently used during the optimisation process. The Java implementation of LSD-SLAM relies on only two libraries: a vector abstraction developed for this thesis used to encapsulate data and methods used in SLAM; and Parallel Colt [125] to implement the LSD-SLAM map optimisation. This chapter contains the design considerations used in the development of the application and the small vector abstraction used.

By profiling the execution of the LSD-SLAM application for Java and visualise Graal IR, it is possible to observe the transformations made during the compilation process. There is a running example to introduce the optimisations applied and how they interact with LSD-SLAM. The compiler is sensitive to the implementation details and this affects the efficiency of the generated machine code. The understanding gained from this exploration is used to develop the co-designed approach for the vector abstraction implementation (Chapter 6) and further evolution targeting SIMD units available in hardware (Chapter 7).
CHAPTER 5. LSD-SLAM IN JAVA

5.1 Motivation

Computer vision is computationally intensive and a balance needs to be found between performance and the expression of algorithms. Compromises are made during application implementation and these are the reasons why there are many SLAM variants. The lack of strict separation between abstraction and implementation impedes the ability to develop and evolve applications quickly, let alone dynamically. The motivation in this thesis is productivity, the ease at which algorithms may be expressed while reducing inefficiencies introduced by additional abstraction. Fortunately, there is domain specific knowledge (semantics) in SLAM that could be used to improve performance but are not used during compiler optimisation. There is potential for dynamic compilation to address the communication of these semantics from an application; enabling specialisation for current hardware architectures.

There is a trend for SLAM applications to execute on technology incorporated into smaller, mobile systems where assumptions of the past no longer hold. These assumptions are ingrained in statically compiled languages and the mentality of application developers, SLAM in particular. Systems are assumed to be static because once running there will be no changes to the environment the application exists; neither is resource competition expected. LSD-SLAM is auto-tuned and hand-optimised for two architectures [17, 127]. This reduces the portability of code and performance and is the motivation for separating the abstraction representing the algorithm from the implementation specialising for hardware.

5.1.1 Dynamic Compilation

Dynamic compilation (or Just-In-Time (JIT) compilation) improves the performance of an application as it is running and the technique is used in managed runtime environments. The benefit is that there is often more information available to optimise code during execution that is unavailable in static compilation. An example is the value of configuration parameters, in C++ they are either compiled into an application or are variable and have to be read for every use. The JVM and Java language have ‘co-evolved’ to provide classes that may be configured during the initialisation of an application and then optimised as constants during execution.
Dynamic compilation is able to overcome the optimisation limitations of libraries for statically compiled languages, whether linked statically or dynamically. Without optimisation during static linking, there is a boundary at the method invocation. It requires the current method frame state to be saved and a new frame created for use by the library method. This boundary may be removed if the invocation and library are both in Java. This allows more efficient use of registers and eliminates the overhead of managing frames on the stack. It is hypothesised that if the compiler is able to generate machine code similar to C++ for the functionality, dynamic compilation will allow Java to surpass the performance of native code.

5.1.2 Heterogeneous Computing

Project Tango [128], a SLAM development kit from Google, and LSD-SLAM for smartphones [127] indicate a trend to move computer vision capabilities to mobile devices. This is mirrored in the use of visual tracking in robotics, automation (aerial and land vehicles) and augmented reality. Heterogeneity is a key component of modern processors in these devices and exists for performance, energy efficiency, security and flexibility. This means that while a device may contain a standard instruction set, for which applications may be statically compiled, there is no guarantee for a consistent architecture during execution. In the LSD-SLAM publications [17, 127], there is a 24-hour phase of auto-tuning; ‘magic-parameters’ improve the performance of the application on different platforms. This is suitable for research but not for the variability of the targeted mobile devices.

Hardware Accelerators

General purpose processing is flexible but uses more energy that specialised hardware. Ubiquitous in desktop, high-performance and, now, mobile computing, GPUs are difficult to avoid; these are utilised with specialised compilers, e.g. OpenGL, OpenCL and CUDA. GPUs contain pipelines that are not easily manipulated by general purpose code but can be utilised by runtime environments such as Jacc [121]. In statically compiled applications, specialisation is implemented manually with targeted code for the system architecture. There is a compromise between hardware specialisation and software portability, a problem managed runtime environments could address.
Another technology commonly found in commodity technology is SIMD units with accelerated floating point operations. Java SE [7] and OpenJDK [2] virtual machines both make use of the SSE instructions to implement floating point arithmetic because it is faster than using the integer pipeline and the FPU coprocessor. ARM processors contain SIMD co-processors that are available using the NEON instruction set [129]. Both of these are used by LSD-SLAM to accelerate data-parallel portions of code and indirectly for some specialised vector operations.

Dynamic Availability

LSD-SLAM makes an assumption about the configuration of a system during auto-tuning and compilation; the consistent availability of resources. However, there are several factors in mobile computing that invalidate this. Performance may change by migrating applications onto another core with different capabilities or by changing the frequency of the core to change its energy characteristics. It is unusual to use only a single application as SLAM is generally an input for other applications; this means that there may be contention for resources or memory. Finally, in a dynamic system, the availability of hardware accelerators may change based on either of the previous two behaviours.

Qualcomm have addressed this with a runtime system called Multi-core Asynchronous Runtime Environment (MARE) [130]. It provides an API to create data-driven tasks and submit them to the runtime. Scheduling, data migration and fault tolerance are abstracted from developers allowing dynamic reconfiguration of applications based on the current hardware availability. While targeting Qualcomm Snapdragon architectures, similar problems exist with ARM based big.LITTLE architectures [41] because processing may also be migrated between processors with different capabilities. Providing a high-level abstraction separating functionality and runtime is important and it is tackled by different specialisations of data-flow scheduling. These allow optimisation for runtime conditions, architectures and application properties, providing better utilisation of resources.

5.2 Implementing LSD-SLAM in Java

Large-Scale Direct Monocular SLAM (LSD-SLAM) [17] is a state-of-the-art application to map small and large environments with the same algorithm. A pure
Java implementation is developed and used to explore optimisation and acceleration of SLAM applications using Graal [5]. It uses existing approaches for computer vision support in Java as inspiration but tackles the problem from a software engineering, rather than an algorithmic, approach.

5.2.1 Computer Vision in Java

There are two main approaches to support computer vision in Java. The first is to wrap C++ libraries using the Java Native Interface (JNI) [131, 132, 133] and the second is to develop pure implementations [134]. JNI [135] is the current, official interaction with native code in Java but it is an expensive route because there is an overhead associated with an unmanaged heap and the interaction with data on the heap. Projects such as JNA [136] are community led projects to improve the performance of native code invocation over JNI. The data footprint of matrices is often large, therefore there is an overhead in copying data between the native environment and Java. An obstacle for optimisation is the inability to cross the native code boundary, methods cannot be inlined and so the extent of transformations available is limited. As data types are small in SLAM, this boundary is inefficient and as such there are few SLAM applications based on these C++ libraries using JNI.

The alternative method to support is to implement a framework in Java. BoofCV [134] is a pure library created to be used to create computer vision applications in Java. This approach contains the restrictions of Java and these have to be addressed to improve performance in SLAM applications.

5.2.2 Design Considerations

The design considerations aim to create an API using sound software engineering principles. This is achieved by creating a new vector abstraction, as a collection of classes, that uses a strict API that hides the implementation for exploration into co-design (the same methodology behind the Emitter interface in MR4J presented in Chapter 3). This is a justified approach as Joshua Bloch has advised: “don’t force users to choose between safety/abstraction and performance” [137]. Rather than initially considering performance the implementation strives for algorithmic clarity.
a) Java implementation

```
A01 // Transform point with rotation and translation
A02 Point3D wxp = rotation.multiply(point).add(translation);
A03
A04 // Project point with camera
A05 Point3D pixel = camera.pointToPixel(wxp);
```

```
A11 public final class PinholeCamera implements Camera {
A12
A13 private final Matrix3D k // intrinsic parameter matrix
A14
A21 public Point3D pointToPixel(Point3D point) {
A22 return k.multiply(point.divide(point.z()));
A23 }
```

b) LSD-SLAM implementation

```
B01 // Transform point with rotation and translation
B02 Eigen::Vector3f Wxp = rotMat * (*refPoint) + transVec;
B03
B04 // Project point with camera
B05 float u_new = (Wxp[0]/Wxp[2])*fx_l + cx_l;
B06 float v_new = (Wxp[1]/Wxp[2])*fy_l + cy_l;
```

Figure 5.1: Manual inlining of point to pixel in LSD-SLAM.

The vector abstraction for LSD-SLAM in Java addresses some of the challenges associated with understanding how the LSD-SLAM publication [17] relates to the implementation in C++. Improved discipline during development of SLAM applications in Java is desired, using strict typing in the API. The critique presented is used as the inspiration for developing the Java implementation that is further optimised by applying co-design.

Figure 5.1 contains an example use of vectors in LSD-SLAM. It contains the algorithm to transform a point in the Euclidean space using a rotation matrix and a translation vector and then to determined using the location of the corresponding pixel (pixel or (u_new, v_new)) projected using the camera model. This
operation occurs for every point used by the application and, since it is frequently performed, it is the target of hand-optimisation. Libraries, such as OpenCV [113], contain camera models but this has not been used in LSD-SLAM. The software engineering principles that the Java implementation aims to maximise are:

- **Consistent use of abstraction**: Figure 5.1 lines B2 and B4–B5 use different techniques to apply a matrix-vector multiplication. The latter has been hand-optimised to propagate constants as they cannot be removed during compilation. Java addresses this with consistency across Figure 5.1 lines A2 and A22 and optimisation is handled by the co-design of the implementation.

- **Strict typing**: a vector and a point contain different meanings and are used for different purposes. LSD-SLAM [17] uses matrices through type definitions of the base abstraction available from the Eigen library [115]. A point in 3D space is represented by an instantiation of a $3 \times 1$ matrix of single precision floating point values. However, so is a 3D vector, both use `Vector3f` as the type name (Figure 5.1, lines B2). By using strict typing, errors such as the incorrect use of variables can be detected during compilation.

- **Strict Encapsulation** avoids duplication of code and reduces hand-optimisation that may obscure the exact nature of an algorithm. LSD-SLAM uses a pinhole camera model to project points between the modelled environment and the frame data. The points are projected using hand-optimised code (Figure 5.1 line B5–B6). The Java implementation of LSD-SLAM has implemented a `PinholeCamera` class with a method to implement projection (`pointToPixel`). This reduces duplication (and the associated risk of making mistakes) and also improves the maintainability. Strict encapsulation also enables many of the Graal optimisation described later in this chapter.

- **Immutability** of objects means that once fields are set they cannot be changed. This allows a consistent view of data throughout an application, especially with parallelism. Programming choices are simplified because there is no distinction between arithmetic operations and in-place arithmetic operations; a new object is created if there would be a side-effect.
Matrix operations in Eigen contain operators for both (e.g. scaling may be written as $A = A \times 0.5$; and $A *= 0.5$). This means that care is required; copies of a matrix must be made if it is scaled in-place and the original is required later in the algorithm. Enforcing immutability of type removes this confusion but it also enables the propagation of constants during optimisation, improving the generated machine code for applications.

The first rule of optimisation is “don’t do it” [138]. However, this is not always true and there are profiling handles in LSD-SLAM that, when used, will provide information regarding where execution time is being spent. This is a sensible approach in targeting optimisations; however it has been done in a manner that removes portability, maintainability and obfuscates the algorithm. Hand-optimisation is achieved without encapsulation and adds verbosity to code. Specialisation for hardware platforms should be separated from the application code, separated as methods and inlined if performance is critical. LSD-SLAM implements the SE(3) tracking algorithm in a single file containing the original C++ code, but the file also contains verbose specialisation for SSE and raw assembly for NEON. As each exists as a single method, a change to the algorithm would be difficult to update in all implementations. This is a motivation for the research in the following chapters as it aims to optimise performance while leaving the source code of the application unchanged.

There is an attempt to improve the performance of LSD-SLAM in C++ by parallelising the algorithms but the result is C++ code that is complex and exhibits deadlock (Chapter 2 describes the challenges). In Java, LSD-SLAM is a sequential application with the three key algorithms executing one after the other. An input is loaded from a file and tracked, the depth is updated for the key-frame and then an optimisation phase is applied to the map. This simplifies the application because it removes many of the locks protecting shared, mutable state in the original implementation.

The output of the application is a visualisation that displays the current key-frame with the depth of colourised points in the point cloud. This provides a basis to assess the progress through the key algorithms in LSD-SLAM for Java. Finally, the implementation does not contain any code for sparse matrix linear algebra; it uses Parallel Colt [125] to achieve this. This forms the bulk of the work executed by the map optimisation task. The constraints and cost-functions, in this, contain the data types developed in the vector abstraction (Sim3Group
The principle consideration is to address the observed algorithm obfuscation in the C++ implementation of LSD-SLAM by creating a strict API. Co-design is used to maintain software engineering principles and to recover the performance lost by removing the hand-optimisation.

5.3 Compilation Using Graal

There are two stages of compilation in Java. The first statically compiles Java to bytecode class files off-line using the `javac` application that are executed by the JVM. Very few optimisations are applied during this stage. The second stage in Java is dynamic compilation using a profiler directed optimiser to improve performance by generating machine code. This means that methods that are regularly executed (10,000 times by default) are dynamically compiled.

JVMs [7, 2] are often written in statically compiled languages, frequently C++, as this permits efficient utilisation of resources during runtime. There are implementations such as Jikes VM [139] and Maxine VM [140] that are written in Java allowing code introspection and extension for research purposes. They also have the advantage of being modular to help developing new approaches in supporting Java and languages using bytecode. As part of the Maxine project the optimising compiler formed its own project Graal [4] that may be used to augment existing JVMs. Graal uses a representation of applications that may be derived from bytecode or from Truffle [141], which allows generated abstract syntax trees (AST) to be optimised and hosted on JVMs. This allows languages not traditionally associated with bytecode to run on the JVM (e.g. C [142]) and this permits interoperability and optimisation over language boundaries [143]. This thesis uses Graal as it provides a modular, extensible optimising compiler for development purposes. Improvements made in Graal could also be more generally applicable to other languages using the mechanisms available in Truffle.

5.3.1 Intermediate Representation

Graal is based on a ‘sea of nodes’ approach to optimisation with vertices and edges in the graph forming the intermediate representation (IR) used to compile application code [5]. A `StructuredGraph`, tracking compilation tasks dispatched from the VM, is initially built from bytecode. It combines control-flow and data-flow, followed using different mechanisms (field and annotations), to define edges.
The graph is optimised using three tiers (high, mid and low) and each has a different emphasis. Tiers contain optimisation phases that perform a variety of different transformations to the graph. Once optimisation is complete the IR is lowered in a final phase to a low-level intermediate representation (LIR). The LIR is optimised further and the nodes emit machine code representing the compiled method; this is returned to the VM for use in execution.

Exceptions are not represented directly in the IR and instead they are replaced by guards. Guards are inserted by replacing exceptions that defined explicitly (as in the example Matrix class Figure 5.2 line 17) or implicitly (e.g. array indexing errors in Figure 5.2 lines 12, 21 and 36). Guards have a condition assigned to them and are linked to the de-optimisation process. This is initiated should the guard fails. De-optimisation identifies the current state of the stack and returns to the equivalent instruction in bytecode and begins to interpret them directly to handle the exception. These guards add inefficiencies during execution and are explained further in Chapter 7.

5.3.2 Optimisation

Optimisation phases are varied in the transformations they apply but aim to improve the quality of the generated machine code. The phases of interest to the co-designed optimisation for SLAM applications are introduced in this chapter by using a matrix-vector multiplication as a running example. The classes used for the matrix and vector are contained in Figure 5.2. The phases described maintain Java syntax throughout to illustrate the transformations. Arrays are used to store the data in Matrix and Vector objects.

The running example used to demonstrate the optimisation phases is the multiplication of a vector (point \(p\)) by an identity matrix \(\text{eye}\), this results a new vector with the same values as the original. The initial code for the method is contained in Figure 5.3. The transformations in the phases are described and result in the desired code which is equivalent to returning \(p\) from the method.

Inlining

After creating the IR in Graal for a method one of the first transformations applied is inlining. Inlining is the process of integrating the bodies of invoked methods directly into the IR of the target method. It improves performance
public class Matrix {
    private final int rows, cols;
    private final float[] data;
    
    public Matrix(int rows, int cols, float... data) {
        this.rows = rows;
        this.cols = cols;
        this.data = data;
    }
    
    public float get(int row, int col) {
        return data[row * cols + col];
    }
    
    public Vector multiply(Vector vector) {
        if (vector == null || vector.length() != cols)
            throw new IllegalArgumentException();
        float[] temp = new float[cols];
        for (int i = 0; i < cols; i++) {
            for (int j = 0; j < rows; j++) {
                temp[i] += get(i, j) * vector.get(j);
            }
        }
        return new Vector(temp);
    }
}

public class Vector {
    private final float[] data;
    
    public Vector(float... data) {
        this.data = data;
    }
    
    public float get(int i) {
        return data[i];
    }
    
    public int length() {
        return data.length;
    }
}

Figure 5.2: Matrix and vector classes used in the optimisation examples.
public Vector example() {
    Matrix eye = new Matrix(3, 3,
        1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0);
    Vector p = new Vector(0.1, 0.2, 0.3);
    return eye.multiply(p);
}

Figure 5.3: Example method used to demonstrate optimisation.

directly by removing the need for the invocation and the created stack frame but its true strength is delivered as an enabler of further optimisations. Inlining cannot be controlled directly in Java unlike statically compiled languages, in which keywords may be used to direct the compiler.

Figure 5.4 contains the result of inlining the method forming the running example. The first step replaces the multiplication invocation (Figure 5.3 line 7) with the body of the instance method (Figure 5.2 lines 16–24). This results in the code in lines A7–A15 in Figure 5.4. Following this, the access to the encapsulated field is exposed by inlining the vector length usage (line A7 transforms to B7) and the matrix and vector get method (line A12 transforms to B12). The maintainability of the equivalent code increases and the algorithm is obfuscated, the method name (multiply) is descriptive whereas the nested loops hide this. This is what the original LSD-SLAM implementation in C++ is doing to improve performance as part of the hand-optimisation observed.

The inlining phase is iterative as illustrated in Figure 5.4 using a top-down approach. Each iteration presents more opportunity for transformation. In Graal, the inlining is done in the opposite direction, bottom-up, so it may restrict the size of the graph. There are a number of restrictions that result in methods in LSD-SLAM not being inlined. The causes are the number of nodes in the sub-graphs occasionally being too large to insert into the current graph. Although the main obstruction is the dynamic nature of methods in classes, as a single invocation may execute code in several different locations.

This dynamic dispatch is a result of polymorphism where a method in a class or interface may be overridden by sub-classes. In the example, the Matrix class may have sub-classes but in the scope of the compilation task it is possible to determine the exact implementation so it may be inlined. The same could not
a) inline the `multiply()` method

```java
public Vector example() {
    Matrix eye = new Matrix(3, 3,
        1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0);
    Vector p = new Vector(0.1, 0.2, 0.3);
    if (p == null || p.length() != eye.cols)
        throw new IllegalArgumentException();
    float[] temp = new float[eye.cols];
    for (int i = 0; i < eye.cols; i++) {
        for (int j = 0; j < eye.rows; j++) {
            temp[i] += eye.get(i, j) * p.get(j);
        }
    }
    return new Vector(temp);
}
```

b) inline the `length()` and `get()` methods

```java
public Vector example() {
    Matrix eye = new Matrix(3, 3,
        1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0);
    Vector p = new Vector(0.1, 0.2, 0.3);
    if (p == null || p.data.length != eye.cols)
        throw new IllegalArgumentException();
    float[] temp = new float[eye.cols];
    for (int i = 0; i < eye.cols; i++) {
        for (int j = 0; j < eye.rows; j++) {
            temp[i] += eye.data[i * eye.cols + j] * p.data[j];
        }
    }
    return new Vector(temp);
}
```

Figure 5.4: Result of inlining the example method.
public Vector example() {
    int eye.rows = 3, eye.cols = 3;
    float eye.data$0 = 1.0, eye.data$1 = 0.0, eye.data$2 = 0.0;
    float eye.data$3 = 0.0, eye.data$4 = 1.0, eye.data$5 = 0.0;
    float eye.data$6 = 0.0, eye.data$7 = 0.0, eye.data$8 = 1.0;
    float p.data$0 = 0.1, p.data$1 = 0.2, p.data$2 = 0.3;
    if (p == null || p.data$length != eye.cols)
        throw new IllegalArgumentException();
    float[] temp = new float[eye.cols];
    for (int i = 0; i < eye.cols; i++) {
        for (int j = 0; j < eye.rows; j++) {
            temp[i] += eye.data[i * eye.cols + j] * p.data[j];
        }
    }
    return new Vector(temp);
}

Figure 5.5: Example method with virtualised objects.

Virtualisation of Object Location

During the execution of the example method, six objects are allocated on the heap; the three matrix and vector objects along with the arrays encapsulated within. Escape Analysis [144] identifies objects that are confined to the targeted scope using a generally applicable algorithm. By analysing the method it may be asserted that only two of these escape from the method and so the eye and p objects may be allocated on the stack. The objects are said to be virtualised. This allows more efficient memory access during the execution of the compiled methods because heap access is accompanied by a larger overhead than when accessing the stack. As with inlining, the virtualisation of objects enables other optimisations. These are represented by primitive types defined in Figure 5.5, eye is virtualised as lines 2–5 and p becomes line 6. During compilation objects that are returned may also exist on the stack until they escape, at which point
01 public Vector example() {
02     int eye.rows = 3, eye.cols = 3;
03     float eye.data$0 = 1.0, eye.data$1 = 0.0, eye.data$2 = 0.0;
04     float eye.data$3 = 0.0, eye.data$4 = 1.0, eye.data$5 = 0.0;
05     float eye.data$6 = 0.0, eye.data$7 = 0.0, eye.data$8 = 1.0;
06     float p.data$0 = 0.1, p.data$1 = 0.2, p.data$2 = 0.3;
07     if (p == null || p.data$length != eye.cols)
08         throw new IllegalArgumentException();
09     float[] temp = new float[eye.cols];
10     temp[0] += eye.data[0 * eye.cols + 0] * p.data[0];
11     temp[0] += eye.data[0 * eye.cols + 1] * p.data[1];
12     temp[0] += eye.data[0 * eye.cols + 2] * p.data[2];
13     temp[1] += eye.data[1 * eye.cols + 0] * p.data[0];
14     temp[1] += eye.data[1 * eye.cols + 1] * p.data[1];
16     temp[2] += eye.data[2 * eye.cols + 0] * p.data[0];
19     return new Vector(temp);
20 }

Figure 5.6: Example method with nested loops unrolled.

they are allocated on the heap. This allows the same transformations that follow
but has to exist as an object when it is materialised.

In C++, virtualisation is implemented directly by the developer. An object
may be allocated on the stack, Vector v1; but it may only be safely used
within the scope of its declaration. Allowing it to escape by passing its address
as a pointer will result in unpredictable behaviour as the stack is re-used during
subsequent execution. Alternatively it may be allocated on the heap, Vector *v2
= new Vector(); if it is required elsewhere and copying the value is inefficient.
Removing this choice from developers can improve productivity and reduce errors
but does contain the overhead associated with the GC. The aim of co-designed
abstractions and optimisations is to provide the programmability of Java but the
control and performance possible in C++.
Loop Unrolling

Loops are inefficient because they contain a jump instruction to repeat or skip instructions in their body. Loop unrolling reduces the number of jump instructions by peeling iterations from the loop body. The scope of the compilation must be able to infer the length of the loop to unroll it fully, as is possible in the running example. If this is not possible, the compiler will peel a deterministic number of iterations surrounding them in the original loop and alter the increment used (partial unrolling). Figure 5.6 illustrates this process by fully unrolling the loop body multiplying elements of each element of each row by elements in the point (Figure 5.5 line 12). The loops defined in Figure 5.5 lines 10 and 11 are transformed into Figure 5.6 lines 10–18. The indices (i and j) are replaced with the constants representing the values of them during the iteration peeled. In partial unrolling this will not be possible and the will remain variables, they would become i, i + 1, i + 2 and so on.

Common Sub-expression Elimination

There is often repeated arithmetic expressions in algorithms caused by low-level duplication and this is removed in common sub-expression elimination. In the running example, the indices to the identity matrix data array contain duplication on lines 10–12 in Figure 5.6. The expression 0 * eye.cols can be assigned as a temporary variable and then this is used in its place. In Figure 5.7 the variable is created and initialised on line 10 and then used on line 11. The same transformation is applied for lines 12–15 resulting in the elimination of six common sub-expressions. The example has taken this opportunity to combine the accumulation of resulting elements to a single expression. This is a liberty to simplify the example; however in reality, without this, it may prevent the next optimisation being applied in the same manner described.

Constant Folding

Constant folding involves the propagation of constants through variables and numeric operations. This simplifies arithmetic and reduces the amount of code generated after the optimisation is complete. It is an iterative process but Graal is able to achieve this by traversing the IR of the current compilation task. There are two sources of constants for propagation in the running example. The first is
public Vector example() {
    int eye.rows = 3, eye.cols = 3;
    float eye.data$0 = 1.0, eye.data$1 = 0.0, eye.data$2 = 0.0;
    float eye.data$3 = 0.0, eye.data$4 = 1.0, eye.data$5 = 0.0;
    float eye.data$6 = 0.0, eye.data$7 = 0.0, eye.data$8 = 1.0;
    float p.data$0 = 0.1, p.data$1 = 0.2, p.data$2 = 0.3;
    if (p == null || p.data$length != eye.cols)
        throw new IllegalArgumentException();
    float[] temp = new float[3];
    int temp0 = 0 * eye.cols;
    temp[0] = eye.data[temp0 + 0] * p.data[0]  
        + eye.data[temp0 + 1] * p.data[1]  
        + eye.data[temp0 + 2] * p.data[2];
    int temp1 = 1 * eye.cols;
    temp[1] = eye.data[temp1 + 0] * p.data[0]  
        + eye.data[temp1 + 1] * p.data[1]  
        + eye.data[temp1 + 2] * p.data[2];
    int temp2 = 2 * eye.cols;
    temp[2] = eye.data[temp2 + 0] * p.data[0]  
        + eye.data[temp2 + 1] * p.data[1]  
        + eye.data[temp2 + 2] * p.data[2];
    return new Vector(temp);
}
CHAPTER 5. LSD-SLAM IN JAVA

01 public Vector example() {
02 int eye.rows = 3, eye.cols = 3;
03 float eye.data$0 = 1.0, eye.data$1 = 0.0, eye.data$2 = 0.0;
04 float eye.data$3 = 0.0, eye.data$4 = 1.0, eye.data$5 = 0.0;
05 float eye.data$6 = 0.0, eye.data$7 = 0.0, eye.data$8 = 1.0;
06 float p.data$0 = 0.1, p.data$1 = 0.2, p.data$2 = 0.3;
07 if (p == null || 3 != 3)
08 throw new IllegalArgumentException();
09 float[] temp = new float[3];
10 int temp0 = 0;
11 temp[0] = 0.1; // 1.0 * 0.1 + 0.0 * 0.2 + 0.0 * 0.3
12 int temp1 = 3;
13 temp[1] = 0.2; // 0.0 * 0.1 + 1.0 * 0.2 + 0.0 * 0.3
14 int temp2 = 6;
15 temp[2] = 0.3; // 0.0 * 0.1 + 0.0 * 0.2 + 1.0 * 0.3
16 return new Vector(temp);
17 }

Figure 5.8: Example method with constants fully propagated.

01 public Vector example() {
02 return new Vector(0.1, 0.2, 0.3);
03 }

Figure 5.9: Example method with dead-code eliminated.

the floating point values assigned to the identity matrix and point co-ordinates in Figure 5.8 lines 2–6. The second is the array index constants created during loop unrolling in Figure 5.6 and then merged during common sub-expression elimination (Figure 5.7 lines 10, 12 and 14). These propagate so the index to the arrays storing the matrix and vector elements are constants allowing access to the constants values initially assigned to eye and p. It is then possible to fold the constants to a single value as shown in the comments on lines 11, 13 and 15 in Figure 5.8.

Dead-Code Elimination

A frequently applied phase during the optimisation process is dead-code elimination. This is a phase that traverses the IR and detects nodes that are not used anywhere and deletes them from the graph. The deletion of a node may remove
the last usage from another node and this will also be deleted. Figure 5.9 shows the final step of optimisation applicable to the example method. The virtualised object fields in Figure 5.8 lines 2–6 are no longer used as the constants have been folded. The condition in Figure 5.8 line 7 is true as it is possible to assert that \( p \) can never be null and that \( 3 \neq 3 \) will always be false. This results in the exception contained within the associated if statement being unreachable so this may also be deleted. Finally the array creation and population in Figure 5.8 lines 9–15 has been prettified resulting in Figure 5.9 line 2.

5.3.3 Problems Arising in Compilation

The running example, introduced in Figure 5.3 and transformed by way of example in Figures 5.4–5.9, demonstrates how transformations may produce efficient machine code. It is sensitive to uncertainty and is conservative in its approach. For example the transformation in constant folding in the running example is made possible by taking a liberty in the preceding transformation. These are the semantics that are possible to observe but difficult to automate and are often opportunities for optimisation that are missed. This was demonstrated by the hand-optimisation in C++ in Figure 5.1 where constant folding is implemented manually. Another example of missed semantics, potentially improving performance, is the symmetry in the construction of the Levenberg-Marquardt matrix in the tracking algorithm of LSD-SLAM. The compiler is not able to reduce the multiplication efficiently; it does not attempt common sub-expression elimination because the method is deemed too large for inlining.

It is possible to write Java that compiles efficiently but there is a risk that functionality is obfuscated or specialised without due consideration of other uses. In the running example, it is known from linear algebra that a matrix (or vector) multiplied by an identity matrix is itself (\( I_m A = AI_n = A \) where \( A \) is an \( m \times n \) matrix). This knowledge may be encoded in libraries for matrix abstractions but in the example given the optimising compiler is unaware and so could fail to reach the optimal solution. The sensitivity of the compiler suggest that there is a need to provide some domain specific semantics into the optimisation phases. This would reduce the general applicability of the optimisations but allow improved performance for the target domain. In Chapters 6 and 7, SLAM applications are targeted based on the experience gained from implementing LSD-SLAM in Java.
5.4 Summary

Using SLAM applications to assess the performance of processing systems is constructive because of the compromise between programming and performance. This thesis uses LSD-SLAM as an application that needs performance improvement when running in a managed runtime environment. A critique of the original implementation reveals that poor encapsulation of the vector abstractions allows direct access to data that is then used in hand-optimisation. This leads to duplication of values and functionality that is manually compiled to SIMD instructions using intrinsics or assembly. The Java platform provides tools to observe where performance is lost and how bytecode is compiled to machine code. This is used to guide the development of co-design approaches to specialise Java for SLAM applications.

The compilation approach used by Graal has been introduced to describe the nomenclature used, and the transformations that occur, in LSD-SLAM. There is a sensitivity to programming techniques encouraged in Java that lead to inefficiencies in the generated machine code for compiled methods. Chapter 6 addresses this by co-designing an implementation of a small vector abstraction, maximising the capabilities of the compiler. This involves small modifications to the compiler and is applicable to irregular and repetitive arithmetic on data in LSD-SLAM. An alternative approach in Chapter 7 extends the Graal compiler to utilise SIMD units. These transformations can result in machine code equivalent to code written in C++ libraries that have been hand-optimised but obfuscating the algorithms used.
Chapter 6

Specialising for Small Vectors

The natural storage of values in vectors is contiguously in memory and in Java this is made possible with arrays. Vectors are a well defined linear algebra concept and are used to represent many entities in SLAM applications, chiefly points and transformations in the Euclidean space. The objective in this chapter is to specialise classes for small vectors as they are required in SLAM applications. It presents a class collection for small vector data types used in LSD-SLAM that is suitable for use in commonly observed SLAM algorithms. Programmability is an important design consideration and, as such, software engineering principle of encapsulation and immutability are used. The benefits extend to optimisation, especially with the scope of the dynamic compiler in Java. Unfortunately, encapsulation and immutability, in combination, can add an overhead because it increases the method indirection and the number of objects instantiated in an application.

During bytecode interpretation these overheads add to the overall execution time but this is addressed during dynamic compilation. Graal is used to detect the scope of objects and may virtualise them on the stack or use only accessed fields. Once stripped to the machine code, compiled Java methods still contain inefficiencies because of the assumptions used during optimisation. Co-design is used to provide a compilation context for SLAM abstractions so that further improvement is possible. The ability to communicate the semantic information about the immutability and structure of both data and algorithms should improve the performance of the abstraction without the need to violate the software engineering principles.

This chapter presents a SLAM vector abstraction implementation based on
specialisation for observations of data usage in LSD-SLAM \cite{17}. There is a description of the effects of this specialisation on the optimisations applied and the improvement in performance. The new implementation is co-designed with the phases of optimisation in the Graal compiler. This improves the performance by increasing the awareness of temporary objects and their values, thus reducing the interaction with the garbage collector. The dynamic compiler is also able to inline methods that are boundaries in statically compiled code, allowing Java to exceed the performance of C++ libraries used by LSD-SLAM in some cases.

The primary contribution in this chapter is an approach to co-design that exploits the nature of the arithmetic used in a specific domain (SLAM). A new class collection is designed using sound software engineering principles and an intimate understanding of how the optimisation phases are applied during compilation. At the same time the optimising phases have been modified to remove further inefficiencies by embedding the semantics of the class collection in the dynamic compiler. These target inlining, constant folding and common sub-expression elimination to generate machine code specialised for small vector and matrix operations.

6.1 Small Vectors in SLAM

In SLAM applications small vectors contains up to seven elements (representing the degrees of freedom). The types represented in the vector abstraction are summarised in Table 6.1. There are multiple uses of these types:

- **Composite Objects**
  The SE(3) and Sim(3) Lie groups representing poses encapsulate an SO(3) and $\mathbb{R} \times \text{SO}(3)$ group respectively, along with a 3D vector containing the translation.

- **Vector Arithmetic**
  Points, vectors and quaternions are subject to both regular (e.g. scaling) and irregular (e.g. cross product) vector arithmetic.

- **Numerical methods**
  The SE(3) and Sim(3) representation of poses are used in a Levenberg-Marquardt \cite{91} (an extension to Gauss-Newton) using matrix decomposition to
CHAPTER 6. SPECIALISING FOR SMALL VECTORS

<table>
<thead>
<tr>
<th>Type</th>
<th>Data Type Names in Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D point</td>
<td>Point2D</td>
</tr>
<tr>
<td>2D vector</td>
<td>Vector2D</td>
</tr>
<tr>
<td>3D point</td>
<td>Point3D</td>
</tr>
<tr>
<td>3D vector</td>
<td>Vector3D</td>
</tr>
<tr>
<td>Quaternion</td>
<td>Quaternion</td>
</tr>
<tr>
<td>SO(3)</td>
<td>SO3Group = { Quaternion }</td>
</tr>
<tr>
<td></td>
<td>Tangent3D</td>
</tr>
<tr>
<td>$\mathbb{R} \times SO(3)$</td>
<td>RxSO3Group = { Quaternion }</td>
</tr>
<tr>
<td></td>
<td>Tangent4D</td>
</tr>
<tr>
<td>SE(3)</td>
<td>SE3Group = { SO3Group, Vector3D }</td>
</tr>
<tr>
<td>se(3))</td>
<td>Tangent6D</td>
</tr>
<tr>
<td>Sim(3)</td>
<td>Sim3Group = { RxSO3Group, Vector3D }</td>
</tr>
<tr>
<td>sim(3)</td>
<td>Tangent7D</td>
</tr>
</tbody>
</table>

Table 6.1: Small vector data types used in computer vision.

solve the system for the next estimated pose in the tracking and optimisation tasks.

Within LSD-SLAM the the exact types from the abstraction used varies for each of the four key algorithms. Tracking applies the logarithm to SE(3) pose to obtain the se(3) representation used in the numerical methods to estimate the camera location relative to the current key-frame. Estimating the depth of pixels using stereoscopic differences is regular in its generation of points to test, translating them iteratively along vectors derived from planes. The optimisation process accumulates poses from the key-frame graph using vector arithmetic combined with manipulation of Sim(3) and sim(3) poses to determine photometric errors.

Scientific computing traditionally uses Fortran to develop applications with vectors and matrices supported by the Basic Linear Algebra Sub-program (BLAS) abstraction. BLAS provides an API for commonly used kernels representing operations used to build algorithms in applications. The Fortran implementation is used as the basis for benchmarking computing, these benchmarks develop so the algorithms using them may take advantage of new numerical theorems and computational thinking. An example given by Jack Dongarra in the evolution of EISPACK [145] to PLASMA [146] was the ability to reduce the effort calculating the bidiagonal form by using the properties of eigenvalues in a sub-matrix. At the same time, the singular value decomposition (SVD) was rearranged to use blocking, this increased the number of floating point operations. However these
are contained within matrix-matrix multiplication which utilises the cache better than the matrix-vector multiplication preceding the modification. This is an active demonstration of the co-evolution of application and abstraction; it is not always best to assume less is better.

Existing approaches in Java use two different techniques to provide abstractions for numerical computation and computer vision applications. C++ libraries may be invoked directly from Java through interaction with wrappers for native code. This contains an overhead as data within the native environment is unmanaged so must interact with the Java heap through a gateway. Also it is not able to optimise over the boundary, so benefits following on from inlining are not possible using this approach. The second approach is to create a pure implementation in Java, this is where existing efforts create expressive APIs but contain hand-optimisations in source code.

6.1.1 Library Wrappers

The library used to represent matrices and vectors in LSD-SLAM and the underlying $g^2o$ map optimisation is Eigen [115]. It is developed as a portable library for matrix-based numerical methods in C++, with support for the geometry in computer vision algorithms. It is implemented in C++ header files so it must be compiled for each use, there is no binary that may be linked statically or dynamically. The Sophus library [116] extends Eigen to provide the abstractions for poses and their associated Lie groups and algebra [94]. Over its history, it has been hand-optimised with aggressive use of the meta-language of templates and use of Intel SSE intrinsic methods to specialise for commodity hardware. Eigen contains efficient implementations of many algorithms used in scientific computing with an expressive, yet highly customisable data abstraction.

JEigen [133] is a wrapper for the Eigen library, compiled as a binary, for Java. It accesses the library using Java Native Access (JNA) [147] as a way to interact with native code without using JNI with its deficiencies. Row and column counts, along with the data reference to an array are encapsulated with the DenseMatrix class. Operations are split between Java (matrix data creation) and native code (matrix operations) but, as with JNI, the ability to optimise over the boundary is not possible. As objects are not contained within the scope of compilation, they have to exist on the heap and interaction with the GC is unavoidable.

Many optimisations added to Eigen rely on the static nature of application
code, compiling and optimising classes with known meta-data, such as matrix size. An example of the resulting restriction from this is that operations such as matrix-matrix multiplication cannot unroll loops fully so contain unnecessary, and inefficient, jump instructions. The wrapper documentation claims an overhead of 27% over C++ for interaction with matrix methods through JNA for $10 \times 10$ matrices [133]. This overhead proportion decreases as the size of the matrix increases; however this chapter is interested in small vectors where higher overheads are not compensated by the hand-optimised libraries. While it is advantageous to utilise the performance of these libraries for data application with complex, large-scale scientific computing, it is not suitable for all uses of numerical methods.

6.1.2 Pure Java Libraries

There is an underlying prejudice against Java for scientific computing because it needs to ‘warm up’ and it was evaluated while JVMs were still in their infancy. Boisvert et al. [148] 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 followed the implementation of JAMA [123], 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 [149] 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 specialisation 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 specialised execution paths. It also allows extensions for blocking and distributed execution as in Colt [124]. Parallel Colt [125] enables parallelism for the operations defined in Colt for Java.

The catch is that in all these cases the abstraction adds an overhead that, while adding efficiency in calculation, is not compensated for by small vectors. As with the proliferation of ‘big-data’ frameworks in Chapter 3, there is also an attitude of forward-looking in scientific computing for bigger data. Specialisation
for computer vision in Java, with its specialist abstractions for small vectors, is available in the GeoRegression library [150] used by larger frameworks. It is based on a pure implementation of a numerical computing library, Efficient Java Matrix Library (EJML) [151], and is utilised in computer vision application framework such as BoofCV [134]. It uses a Matrix interface that is implemented in a different manner depending on the matrix or vector size. Small vectors are represented by classes with the data represented as public, non-final fields that allow efficient access without the indirection of arrays used in larger matrices.

There is a performance benefit from specialising matrices as fields in an object and this will be demonstrated in this chapter. It also provides a critique of the use of public, non-final implementation and why this is disadvantageous for the compiler. The programmability is affected by the public nature of the field. Once this library is made public the API is fixed so future improvements to the class would be hindered by the direct interaction between application and data implementation. This is an aspect of programmability that is an important consideration and why encapsulation and immutability are used in the co-design of an abstraction for small vectors.

6.2 Co-designing Optimisation

The vector abstraction used by the Java implementation of LSD-SLAM introduced in Chapter 5 uses a matrix library that is inefficient for SLAM applications. The implementation developed in this chapter has the express purpose of allowing specialised support during compilation to improve the generated code. Using the model developed during the evolution of numerical computing in Java and the field-based implementations of computer vision libraries as a starting point. The hypothesis is that by first addressing the limitations in the existing approaches and co-designing the optimiser to be semantically-aware, the execution time of the generated machine code will improve in Java. Improving encapsulation improves the flexibility of the implementation for future enhancements (as demonstrated by the Emitter interface for MR4J in Chapter 3). Introducing immutability simplifies the semantics of developed code and the assumptions made in the compiler. This mirrors the evolution of the JDK in which in-consistent mutable types have been deprecated in favour of immutable equivalents (e.g. LocalDate superseded the Date class in Java SE 1.8 [7]).
6.2.1 Removing Indirection

Arrays are special objects in Java with dedicated bytecodes associated with their usage. A major restriction is that they cannot be sub-classed (or extended) so there is no ability to add new methods to the existing constructors, length attribute or access via indexing. Class abstractions using arrays as their storage may provide additional attributes and methods, but as operators cannot be overloaded the use of square bracket syntax is lost. Furthermore this introduces indirection, as demonstrated in Figure 6.1.b and has to rely on the optimising compiler within the JVM to transform the data to improve performance. Compared to C++ (Figure 6.1.a) libraries, the resulting machine code is sub-optimal due to its structure in memory.

There is room for improvement as the C++ approach to data abstraction is flexible and concise. However there are still advantages to the managed memory model in the JVM that will prevent the object header from being removed. It is required for the identification of its type and information for use in the GC algorithm. BoofCV [134] uses specialisations for each vector size and type because of the incompatibility of primitive types and generics in Java. For small vectors, the overhead of indirection is significant and so moving the array elements to primitive type fields in the class is beneficial.

6.2.2 Adding and Encapsulating Immutability

Encapsulation of fields is important for programming; separating the public interface from the implementation of a class allowing evolution over time. For example, the encapsulation of the fields within ArrayList has allowed its implementations to be significantly changed from Java SE 1.7 to 1.8. The original array that its implementation contained, and its name derived, has been replaced by size dependent data structures that are compatible with streams. Had the array been exposed as a public field, this would have been impossible without invalidating the principles of version compatibility in Java. This encapsulation is addressed by the new implementation of the small vectors abstraction for the LSD-SLAM application.

Figure 6.2 contains a comparison between the abstraction for the base class used for 2D vectors and points. The use of four values is because of the research presented in Chapter 7, where the SIMD unit is used for further acceleration.
a) direct mapping of an object in C++

\[
\text{Matrix<float,3,1> vec;}
\]

b) indirection and overhead of an object in Java

\[
\text{SimpleMatrix vec = new SimpleMatrix (3, 1);}
\]

Figure 6.1: Memory layout of a Vector in C++ and Java.
CHAPTER 6. SPECIALISING FOR SMALL VECTORS

a) GeoRegression implementation of an abstract 2D vector

```java
public abstract class GeoTuple2D_F32<T extends GeoTuple2D_F32>
        extends GeoTuple_F32<T> {
    public float x, y;

    public GeoTuple2D_F32(float x, float y) {
        this.x = x;
        this.y = y;
    }

    public void plusIP(GeoTuple2D_F32 a) {
        x += a.x;
        y += a.y;
    }

    public T plus(GeoTuple2D_F32 a) {
        T ret = createNewInstance();
        ret.x = x + a.x;
        ret.y = y + a.y;
        return ret;
    }
}
```

b) New implementation of abstract vector

```java
abstract class AbstractV128 {
    private final float v0, v1, v2, v3;

    protected AbstractV128(float v0, float v1, float v2, float v3) {
        this.v0 = v0;
        this.v1 = v1;
        this.v2 = v2;
        this.v3 = v3;
    }

    protected final <T extends AbstractV128>
            T addImpl(AbstractV128 other, T type) {
        return type.instantiate(v0 + other.v0,
                                v1 + other.v1,
                                v2 + other.v2,
                                v3 + other.v3);
    }
}
```

Figure 6.2: Implementations for small vectors in Java.
The important change is to set the modifier of fields to `private`, which keeps the implementation separate from the developer and sub-classes; changes can be made in a single place, even sub-classes have to access data through methods. Generic types are supported in Java and allow a method to be specialised when it is used, the vector additions in lines A21 and B21 in Figure 6.2 are flexible in their return type. Historical decisions led to generics being implemented using type erasure [152], this removes type information at runtime so cannot be used to instantiate information (`new T()` is illegal as of Java SE 1.8). This complicates the implementation because a factory method is required; it is common to both implementations as observed in lines A22 and B22 in Figure 6.2. The debate and details surrounding type reification in Java is ongoing [153] and may make it into the language specification. The implementation presented in Figure 6.2.b hides this behind the API so advances in the language may be used; should it become a reality.

Figure 6.3 contains the class hierarchy used in the vector abstraction provided in the new library and used in SLAM. It takes some inspiration from the very strict typing in Ada [19] to increases the number of errors caught during compilation.
public Rotation3D rotation() {
    return quaternion.normalise().rotation();
}

public float scale() {
    return quaternion.norm();
}

public Rotation3D transform() {
    return rotation().multiply(scale());
}

Figure 6.4: Derivation of transform matrix from the RxSO3Group type.

A benefit is to maintain correct abstractions because a single vector abstraction would not highlight any mistakes resulting in this kind of calculation. For example a point multiplied by a vector returns a point translated from the original, 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 now can be prevented.

Immutability is the other important addition; in code this is just the addition of the final modifier to the fields. The restriction it creates is demonstrated in Figure 6.2; the GeoRegression implementation of a 2D tuple is able to implement in-place operations that immutable classes cannot. Superficially it improves performance by allowing operations without the need to instantiate a new object on the heap. This creates side-effects that are the source of transient bugs in parallel programming as introduced in Chapter 2. However the actual performance improvement is dependent on the scope of the object maintaining the accumulated value. The resulting machine code will be the same if the value does not escape the scope of compilation and is virtualised on the stack. The danger of mutable types occurs if it does escape, as it becomes difficult to reason about its contents, especially in a multi-core environment. Immutability dictates that any reference to the original object will observe a consistent view.

The benefits of immutability become clearer as algorithms increase in their complexity. Figure 6.4 illustrates three methods from the RxSO3Group class. The encapsulated quaternion is immutable so the normalisation and norm calculation in the rotation and scale methods respectively produce the same intermediate
result. Once inlined by the dynamic compiler, the norm need only be calculated once. The assurance of this simplifies the implementation making the code more readable, therefore improving its programmability. In the equivalent C++ code, observed in the Sophus library [116]. A new quaternion is explicitly created for normalisation so as not to corrupt the original values due to the in-place operations. The inconsistency in implementation of operator overloading is one of the reasons it has not been used in Java, but here the importance of immutability in programmability is demonstrated.

6.2.3 Semantically-Aware Optimisation

The specialisation for matrix and vector sizes allows more aggressive optimisation in commodity JVMs. In MR4J (Chapter 3), the optimiser associates separated functionality with semantic understanding of the methods to enable more transformations. This chapter provides additional information that cannot be inferred from an array or the abstractions introduced in the library. The semantics of immutability, consistency and encapsulation of the data affect the scope of the compiler and target the following transforms:

- **Reduced 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 this (and later in their collection).

- **Reduced 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 optimised by a base address and offset, the indirections 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. The vector abstraction enforces immutability in software
and constant folding ([154] pages 329–331) allows the propagation and replacement of variables with constant values in expressions, simplifying code. In Figure 6.2, as the fields are final floating point values, the optimiser is able to use the values directly in generated code if the instantiation is within scope or they are runtime invariants (e.g. final class variables).

- **Enhance common sub-expression elimination**
  As mentioned in the justification for array expansion in high-performance computing [155], there are challenges in establishing data dependencies when arrays are involved. By eliminating the indirection and simplifying the values as primitive types, these challenges are removed. The existing optimisation phases in Graal are then able to be more aggressive as primitive types are more predictable in their nature.

The main inhibitor in Graal preventing the full inlining and optimisation 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 the boundary cannot be optimised across. It is this process of inlining for which this chapter is providing additional information, ensuring a suitable scope for SLAM applications and removing uncertainty.

### 6.3 Implementation

There are two parts to the implementation creating the data representation for small vectors (and small matrices to support them) and awareness in the optimising compiler. The representation is designed to maximise the ability of compiler to optimise code, but maintaining the programmability of the classes. Graal, the optimising compiler used, is modified to remove the restrictions that cannot be addressed from Java alone.

#### 6.3.1 Representing Small Vectors

The data types representing small vectors are based on the class hierarchy introduced in Figure 6.3 are provide specialised methods for the different types. The abstract classes specialise the data representation and contain all method implementations used by sub-classes, the fields are not available to these to allow
future modifications in the implementation. The methods made public through
the API have to check the validity of the input and then they may use the super
class implementation to execute the operation.

The new abstractions introduce implementations for four (AbstractV128) and
eight length vectors (AbstractV256). Larger vectors used to store image data use
existing approaches, with data abstraction based on matrices with meta-data in
the form of properties. Figure 6.2.b contains a snippet from the new class showing
the constructor and add method. Each contains final primitive types that are set
in, and only in, the constructor. Static methods are used to generate frequently
used instances, e.g. a unit quaternion or identity matrix. An abstract class is
used to implement the base for all data types as it allows final implementation
of methods that cannot be overridden in sub-classes. The benefit of this is that
the method can be clearly defined so it may be inlined rather than having to
dynamically dispatch to whichever type is given as an argument. There needs to
be flexibility in the type returned from the operations and that is why there is a
type argument (Figure 6.2.b line B21). There are two public interface (IVector
and IMatrix) that may be implemented optionally by types in the abstraction if
they are compatible with vector and matrix arithmetic. The interfaces allow the
types to be used in packages that implement numerical methods such as matrix
decomposition.

Although not mentioned so far, but following the same approach, is the spe-
cialisation for commonly used square matrices used in SLAM. GeoRegression
Java library [150] contains the same specialisation, up to a six by six matrix to
support the accumulation of partial differentials of se(3) poses for estimation in
tracking. The implementation in this chapter supports up to an eight by eight
matrix (AbstractM256) and is able to do the same for the sim(3) used in LSD-
SLAM. There are additional semantics available that have not been addressed
in this thesis but would be to use meta-data to reduce calculations and storage.
The meta-data can hold mathematical knowledge allowing specialisation of im-
plementations. For example the multiplication of a vector by its transpose creates
a symmetric matrix, the lower and upper triangular matrices contain duplicate
information. The basis of this can be found in the OoLALA [149] research but
is not extended in this research. The removal of duplication should be addressed
by the optimisation phases during compilation.
The vector abstraction sub-classes of the vector data implementation guarantees that all vectors that are four values are an AbstractV128. This means that there is no need to handle operations differently based on vector size. The padding in this situation will be removed as part of the dead-code elimination as the final values will not be used or constant folding be applied. There is a compromise made as this padding increases the size of objects in memory but this is justified by the simplified implementation, improved optimisation possible through co-design and for use in SIMD units in Chapter 7.

6.3.2 Extending the Optimiser

The implementation of the abstract classes has improved the machine code generated but there is still one main obstruction during compilation. This is caused by the verbosity required in the Java source code for getters because an if statement is used to access fields. Alternative implementations could use unsafe access to fields based on the offset within the object or by using reflection. These were not used because the verbosity is easier to follow during debugging or to allow for any future extensions to the classes.

The verbosity of the field access, and some methods (e.g. matrix multiplication), create IR sub-graphs that are deemed too large to inline. This results in the methods being compiled separately and accessed by a direct method invocation; this is not ideal for the optimisations targeted. The semantic information the optimising compiler does not have is that many of the operations will simplify because of properties of the usage. For example the Levenberg-Marquardt update matrix is symmetric so many sub-expressions will be eliminated. This chapter is using the assumption that the arithmetic density of node in Graal IR [5] will increase if methods provided in the abstraction are inlined. Therefore the Graal compiler has been modified to inline all methods implemented in sub-classes of the AbstractV128, AbstractV256, AbstractM128 and AbstractM256 abstract classes.

The other change is to use a macro-substitution for the get methods (Chapter 7 explains this mechanism for vectorisation). This allows values accessed 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.
6.4 Performance Evaluation

6.4.1 Experimental Set-up

The benchmarks used in this evaluation are four performance critical SLAM kernels used and that are also observed in other SLAM applications. The hardware and software configuration represents a commodity computer and the small vector class collection presented in this chapter is compared to Eigen [115] (C++) and three rival Java libraries.

Hardware and Software Configuration

The experiments were conducted on a single hardware platform, exploring the performance on a multi-core workstation. Table 6.2 presents the hardware and software configuration used to run the benchmarks. The JDK used to compile the benchmarks has been downloaded and built using the source from the OpenJDK project [2]. The same project is also used to provide the source for the JVM used, but in this case the C++-based server compiler is replaced by Graal [4]. Each test contains 1,000,000 executions of the SLAM kernels. The results are generated from 25 tests following two un-timed cycles of 10 tests to ‘warm-up’ the JVM and force compilation. Care has been taken to ensure that the GC does not influence the results, with the execution times generated from code without ‘stop-the-world’ interruptions.

SLAM Kernel Benchmarks

The SLAM kernels used as the benchmarks have been selected to explore some of the performance shortfalls of the LSD-SLAM implementation in Java and represent frequently occurring algorithms that use small vector data types. They aim to demonstrate the overhead and optimiser inefficiencies when dealing with mutable types that are not well encapsulated. The implementation of the kernels in Java can be found in Appendix C, the contribution of each kernel to the execution time of LSD-SLAM is illustrated in Figure 6.5. The SLAM kernels used are:

- Point Transform
  Points are represented by three co-ordinates \((x, y, z)\) but in many uses this is relative to the current key-frame. When comparing pixels, the point is
CHAPTER 6. SPECIALISING FOR SMALL VECTORS

<table>
<thead>
<tr>
<th>Hardware</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Core i7</td>
</tr>
<tr>
<td></td>
<td>4770 3.4GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
</tr>
<tr>
<td>Hardware threads</td>
<td>8</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32kB per core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256kB per core</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8MB per 4 cores</td>
</tr>
<tr>
<td>Main memory</td>
<td>16GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Windows 8.1</td>
</tr>
<tr>
<td>C++ compiler</td>
<td>MSVC 17.00.61030 for x64</td>
</tr>
<tr>
<td>Java</td>
<td>OpenJDK 1.8.0_internal</td>
</tr>
<tr>
<td>JVM</td>
<td>OpenJDK 64-Bit Graal VM</td>
</tr>
</tbody>
</table>

Table 6.2: Configurations for new abstraction evaluation.

Figure 6.5: Contribution of SLAM Kernels to LSD-SLAM execution time.
transformed and projected onto the frame image to determine the photometric error. This is achieved by rotating and translating the point and then projecting it with the camera model. Rotation is represented as a 3D matrix (or quaternion) and the translation is a 3D vector. The point is first rotated with a matrix-vector multiplication and then the translation is added as an offset, projection is a second matrix-vector multiplication. It occurs during tracking and map optimisation, each point in the point cloud is transformed in each iteration.

- **SE(3) Logarithm**
  An SE(3) pose is represented as a rotation and a translation. This is a complicated form to use in linear algebra and so the Lie group may be transformed to a vector with six elements. This is achieved by taking the logarithm creating the $\mathfrak{se}(3)$ pose. It is a frequent operation, particularly in map optimisation 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 the 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 in the tracking and map optimisation 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 $\mathfrak{se}(3)$ pose estimation to solve the next estimate of the current frame relative to the key-frame. The Levenberg-Marquardt algorithm [94] is used as it is a non-linear system and it uses the partial differential of elements in the vector with photometric errors.

**Comparison**

The performance evaluation is performed with six frameworks implemented with identical procedures within the bounds of the provided API. *Eigen* [115] is used
CHAPTER 6. SPECIALISING FOR SMALL VECTORS

<table>
<thead>
<tr>
<th>Framework</th>
<th>Point Transform</th>
<th>SE(3) Logarithm</th>
<th>Gradient Interpolation</th>
<th>L-M Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eigen</td>
<td>13.342</td>
<td>131.138</td>
<td>9.847</td>
<td>152.376</td>
</tr>
<tr>
<td></td>
<td>(0.128)</td>
<td>(3.046)</td>
<td>(0.309)</td>
<td>(2.789)</td>
</tr>
<tr>
<td>Arrays</td>
<td>57.084</td>
<td>819.950</td>
<td>48.743</td>
<td>980.302</td>
</tr>
<tr>
<td></td>
<td>(0.226)</td>
<td>(44.076)</td>
<td>(0.824)</td>
<td>(3.696)</td>
</tr>
<tr>
<td>JEigen</td>
<td>1356.498</td>
<td>1671.105</td>
<td>58.961</td>
<td>895.845</td>
</tr>
<tr>
<td></td>
<td>(38.164)</td>
<td>(43.373)</td>
<td>(0.959)</td>
<td>(8.166)</td>
</tr>
<tr>
<td>EJML</td>
<td>77.411</td>
<td>415.924</td>
<td>84.479</td>
<td>308.412</td>
</tr>
<tr>
<td></td>
<td>(8.383)</td>
<td>(8.450)</td>
<td>(1.277)</td>
<td>(5.648)</td>
</tr>
<tr>
<td>FloatVector</td>
<td>98.746</td>
<td>437.434</td>
<td>60.431</td>
<td>561.156</td>
</tr>
<tr>
<td></td>
<td>(9.433)</td>
<td>(8.219)</td>
<td>(0.730)</td>
<td>(6.072)</td>
</tr>
<tr>
<td>New Impl.</td>
<td>20.784</td>
<td>142.797</td>
<td>6.634</td>
<td>361.892</td>
</tr>
<tr>
<td></td>
<td>(0.336)</td>
<td>(3.430)</td>
<td>(0.180)</td>
<td>(4.045)</td>
</tr>
<tr>
<td>Co-design</td>
<td>20.253</td>
<td>104.564</td>
<td>6.464</td>
<td>55.977</td>
</tr>
<tr>
<td></td>
<td>(0.201)</td>
<td>(2.157)</td>
<td>(0.137)</td>
<td>(2.700)</td>
</tr>
</tbody>
</table>

Table 6.3: Execution times for the SLAM kernels (in nanoseconds).

for C++ and it provides the reference for Java performance. The code is compiled using the Microsoft C/C++ optimising compiler, optimising for speed over the whole application. Java arrays are used as a demonstration of the performance achievable using arrays directly; the programmability is not a concern for benchmarking. JEigen [133] is used to demonstrate the overhead associated with native calls in Java using JNA. EJML [151] is the basis for existing Java implementation of abstractions for computer vision, as GeoRegression [150], used by BoofCV [134], does not contain a consistent implementations for data types used by the SLAM kernels. The FloatVector class, part of an in-house class collection, uses an array to store data and an Index interface to provide flexible re-use of data (e.g. it is possible to view a matrix and its diagonal from separate objects sharing the same data). It has been developed as the original abstraction for vectors in LSD-SLAM for Java and is used to store all numeric types. It is similar to EJML but with added immutability. The new implementation for the API used in LSD-SLAM is reported separately from the co-designed approach, in which the optimisation is enabled.
Figure 6.6: Relative performance of small vector abstractions vs Eigen.

Figure 6.7: Relative performance of small vector abstractions vs arrays.
6.4.2 Results

The transform benchmark contains four small vector operations: two 3D matrix multiplications; a vector add; and a vector division. One of the matrices (the intrinsic parameter matrix for the camera) is a static field in Java but exists on the stack in C++. The results shown in Figure 6.6 are relative to Eigen and in Figure 6.7 are relative to Java arrays. The pure Java classes that are designed for large matrices and vectors demonstrate their unsuitability for small vector data types (5.82 and 7.40 times slower than C++). JEigen is over 100 times slower than Eigen and demonstrates the overhead associated with native code interaction in Java when the problem size is too small. When the co-designed approach to Eigen in Figure 6.6 it has a speedup of 0.86, meaning C++ still outperforms Java.

The SE(3) logarithm in Figure 6.6 shows the widest range of results. Arrays, EJML and FloatVector have poor performance (all over twice as slow as Eigen) because of the issues of indirection and the inability of the optimiser to virtualise data. In this algorithm the vector libraries outperform pure Java arrays and the poorer performance may be attributed to indirection of the array of arrays used in the benchmark. JEigen must interact with native code a few times in this benchmark so the mismatch in its target application is amplified, running over 12 times slower than Eigen, the source for its native code. The new implementation is 12% slower than the implementation in the Sophus library [116] but this result changes when the optimisation in applied. When this occurs the co-designed abstraction that addresses these issues is able to execute faster than C++ and has a relative speedup of 1.25.

The gradient interpolation in Figure 6.6 demonstrates the inlining capability of Java for algorithms using only small vectors. The difference in performance between the new implementation, the co-designed approach and Eigen is small because the same optimisations are applied. Arrays, EJML, JEigen and FloatVector have poor performance (all over five times slower than Eigen) because of the issues of indirection and the inability of the optimiser to virtualise data. The relative speedups are 7.54 and 1.52 times faster than Java arrays and Eigen respectively. However this algorithm is taken in isolation, ignoring the access of input data containing the original gradients and luminosity. This means that the observed speedup will have a smaller influence on the performance of LSD-SLAM as a whole when compared to the original C++ implementation.
Finally, the Levenberg-Marquardt update provides a speedup against Eigen but, more importantly, also against all other Java libraries used. This can be observed in Figures 6.6 and 6.7. The improvement of the co-designed implementation of small vector data types is 17.51 and 2.72 times relative to Java arrays and Eigen respectively. In this case the overhead of JEigen is reduced as the matrix-multiplication executed in native code is larger than in the other kernels (6 × 6 compared to 3 × 3). The performance of EJML has a speedup of 1.82 times over the \texttt{FloatVector} classes due to the restrictions imposed in creating immutability in Java with arrays.

In three of the four SLAM kernels used as benchmarks, the original implementation of vectors for LSD-SLAM and two other matrix libraries available were slower than the C++-based Eigen library. The use of encapsulation and immutability led to poorer performance than EJML, a comparable library. This chapter introduced an implementation of the same small vector abstraction with specialisation for the data types used and with an understanding of the compilation process. This improved the performance relative to Eigen but it was still slower. However, by modifying the the Graal compiler to understand the nature of arithmetic used in SLAM applications it was possible to extend inlining and constant folding so that performance surpasses C++ in all SLAM kernels.

### 6.5 Discussion

The Point Transform SLAM kernel was used as an example of hand-optimisation in the original C++ implementation of LSD-SLAM (Figure 5.1.b). The implementation of the benchmark in this performance evaluation uses the intrinsic parameter matrix to project the point using the pinhole camera model. This has a mean execution time of 13.342ns in C++ using the Eigen library, when the hand-optimisations are added to the benchmark, the mean execution time reduces to 5.390ns (0.128 s.d.). This demonstrates the benefits of hand-optimisation, but it is not necessarily the correct step to take in pursuit of performance. The cause of this is that the density of the arithmetic increases in the initial matrix-vector multiplication, as no simplification is possible. Using SIMD instructions to accelerate this in Java is explored in Chapter 7.

It can be observed in Figure 6.7 that JEigen is very inefficient, despite matrix multiplication being offloaded to Eigen, which has superior performance to many
Java abstractions. This is because of the incompatibility in problem size between SLAM and that for which Eigen is designed. The overheads introduced by off-loading arrays to native code through the JNA is not recovered by the ‘better’ execution time of C++. This is one of the motivating factors for creating a pure Java approach, especially as this restriction is also true to existing vectorisation approaches for Java introduced in Chapter 7.

The indirection in EJML and JEigen is from an array field in the SimpleMatrix and DenseMatrix abstractions respectively. They both use a one-dimensional array to store matrices to limit the indirection. The array index is calculated from the row and column indices before elements are accessed. The FloatVector class encapsulates a FloatStore object that may be shared between matrices and vectors. This allows the reuse of data in large data structures and is possible because the data is immutable; a sub-matrix contains a different Index object but a reference to the original array. This includes indirection to the index that specialises the data abstraction and provides matrix and vector properties. However, this results in a noticeable overhead when small vectors are used. This is in part because of indirection (for reading elements) but also because additional objects must be created (whether virtualised on the stack or allocated on the heap). The FloatVector performance is affected in all the benchmarks because of this.

The Java arrays performance is poor because there are many object references in the more complex benchmarks (SE(3) logarithm and Levenberg-Marquardt update). It uses an array of arrays (the naive but natural choice when using pure JDK classes and Java syntax) and each row is a new object. Therefore for Levenberg-Marquardt update there are a total of 34 arrays allocated per iteration. Even with virtualisation of object location on the stack this overhead affects the execution time. In contrast, the co-designed abstraction creates two objects per iteration in the benchmark. In this way, and thanks to other optimisations previously not possible, co-design can improve performance by 17.51 times over arrays. Removing indirection for small vectors improves performance by allowing better optimisations after inlining and also reduces the interaction with the GC as fewer objects are allocated.

The potential inlining for the Java-based matrix abstraction is limited as the vector size cannot be detected during dynamic compilation of the benchmarks. The compiler cannot infer the size and so any size matrix or any length vector could be provided as an argument. Therefore, it cannot be certain about
the internal structure and cannot virtualise them. This results in an IR graph that contains a series of runtime method invocations, rather than optimising the method. The size is known in the co-designed abstract classes, therefore these problems are non-existent and it is possible to force inlining.

The co-designed abstraction specialises for small vectors and matrices, allowing Java to utilise optimisations available for primitive types that have different semantics to arrays and object types. Arrays provide the functionality for a basic vector and may be used to implement many of the benchmarks and have good performance; the challenge is in their use during programming. There is no checking for null and compatibility between arguments, e.g. a 10 and 18 element array may be used to store a 2D point, with undefined use of the remaining data. This is disadvantageous for program integrity but it provides reasonable performance, as demonstrated in Table 6.3. Figure 6.7 contains the relative performance of the Java array abstractions against using `float[]` types alone. These checks are simplified in the new abstraction with guards being null checks on arguments. The data types add the bound checks that would have to be added to Java array helpers. Co-design adds further performance improvements by enhancing the assumptions made in the compiler and is 7.84 times faster than arrays in SE(3) logarithm and 17.51 times faster when updating the intermediate matrix and vector for Levenberg-Marquardt algorithm estimating the pose in LSD-SLAM. When using an abstraction designed for larger, regular data structures capable of GPU offloading [121] (`FloatVector`, the improvement in performance for the update is 10.02 times. This speedup is because of the mismatch in implementation of the abstraction and SLAM kernels which this chapter addresses. In maintains the same API but implements the classes in co-operation with the Graal optimising compiler to surpass the performance of hand-optimisations in equivalent C++.

### 6.5.1 Immutability in Levenberg-Marquardt

The Levenberg-Marquardt benchmark is implemented differently in Eigen (C++) and the other Java abstractions. Immutability is assumed in all Java abstractions so several new objects are allocated on the heap during each iteration. However, the LSD-SLAM application uses a mutable result matrix and vector that are updated with each new estimated pose. Table 6.4 contains the execution times of the benchmarks using mutability to update the system matrix and vector. JEigen and EJML provide in-place multiplication and the `FloatVector` may be replaced...
CHAPTER 6. SPECIALISING FOR SMALL VECTORS

<table>
<thead>
<tr>
<th>Framework</th>
<th>Execution Time</th>
<th>Relative Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>s.d.</td>
</tr>
<tr>
<td>Egien (C++)</td>
<td>133.211</td>
<td>2.088</td>
</tr>
<tr>
<td>Arrays</td>
<td>767.108</td>
<td>14.104</td>
</tr>
<tr>
<td>JEigen</td>
<td>869.585</td>
<td>10.053</td>
</tr>
<tr>
<td>EJML</td>
<td>197.406</td>
<td>7.439</td>
</tr>
<tr>
<td>FloatVector</td>
<td>261.166</td>
<td>7.076</td>
</tr>
<tr>
<td>Co-design</td>
<td>36.777</td>
<td>1.911</td>
</tr>
</tbody>
</table>

Table 6.4: Results of mutability in the Levenberg-Marquardt update.

with the FloatVectorBuilder class used to construct vectors from smaller vectors or individual elements. A class has been developed for this investigation that provides mutability for the co-designed approach using the same structure and methods. The performance restriction of the Levenberg-Marquardt update is important for the Java LSD-SLAM implementation as 40.7% of the execution for tracking is spent maintaining the intermediate values during tracking. The relative performance of the co-designed abstraction increases from 2.28 to 3.62 times the Eigen algorithm that uses the template meta-language of C++.

The debate here is whether the additional performance is worth the degraded programmability. Immutability is useful when parallelising this code; however, so far it has only been considered within a sequential algorithm. A simple approach to utilising multiple cores is to find calculations with no shared, mutable state. In the LSD-SLAM tracking algorithm this is the calculation of poses for each point. The complexity arises in combining the estimated poses in the Levenberg-Marquardt update, which is shared. Without immutability, a critical section would have to be made containing the update kernel as there will be the opportunity for data races as elements of the matrix and vector are updated. In the co-designed implementation with immutability, the options are more easily tested, improving productivity. All options introduced in Chapter 2 are open to exploration; atomic operations could use the AtomicReference class; transactions with Deuce STM [83]; or using locks, whether a re-entrant lock or a Java monitor. Beyond this short exploratory research, the immutability of the small vector abstraction is all that is provided by the API.
6.5.2 Optimising LSD-SLAM in Java

LSD-SLAM is highly dependent on the input, initial conditions and order of execution. To demonstrate the use of the co-design associated with the data types presented in this chapter, the tracking algorithm has been isolated with its executions timed. This uses an iterative technique to estimate the pose (or transform) applied between input frames. The depth is used to project points between the image and the 3D model of the environment. It is initialised with random values but in this demonstration all depths are initially assumed to be 1.0. The tracking algorithm dictates the throughput of data to the other key algorithm, the depth estimation and map optimisation. Only once the pose is estimated can the depth of points in the key-frame be calculated and optimisation between key-frames begin.

Before the optimisations are enabled, the LSD-SLAM tracking task has a mean execution time of 327.7ms per frame (13.5 s.d.) over 25 frames after a five frame ‘warm-up’ period. The abstraction without modification to the compiler, the baseline performance in the evaluation, reduces the average execution time to 126.5ms (4.7 s.d.). Using only a software engineering approach to optimisation in Java for the LSD-SLAM implementation produces a speedup of 2.59 times. Enabling semantic understanding of the vector abstraction implementation in Graal reduces the execution time further to 57.9ms (3.8 s.d.). This is a improvement in performance of 2.18 times, giving a speedup of 5.66 over the original Java implementation. This demonstrates the advantage of specialisation and the use of co-design in the optimising compiler.

However, by way of comparison, the original C++ implementation tracks with a mean execution time of 14.3ms per frame (0.5 s.d.). This means that the relative performance of the Java implementation has improved from 0.04 to 0.25. The C++ is still able to run four times faster. The reason for this is because of other optimisations that have not been implemented in Java. The main hand-optimisation the creation of buffers to store intermediate data that are traversed to generate results used in tracking. This improves memory access as the data is static and accessed sequentially. Java does not address this and continues to use objects allocated on the heap as they are required in the algorithm. The rewriting of the application for this behaviour in not addressed in this thesis but is the target of optimisations in Jacc [121].
CHAPTER 6. SPECIALISING FOR SMALL VECTORS

6.6 Summary

The natural storage of values in small vectors is contiguous in memory and in Java this is made possible with arrays. This chapter presented a new class collection for small vector data types. It uses the software engineering principles of encapsulation and immutability to simplify programming. The benefits of immutability are for programming but also extend to optimisation, especially with the scope of the dynamic compiler in Java. Unfortunately, encapsulating Java arrays contain additional indirection and object allocation in Java. This is addressed by creating the AbstractV128 base class the contains primitive type fields and methods required in SLAM applications.

When compared to existing libraries in C++ and Java it is possible to observe and understand why the implementations perform better than an encapsulated mutable array. Once stripped to the machine code, compiled Java methods contain inefficiencies due to the uncertainty of invocations and data abstraction, including size. The ability to communicate the semantic information about the immutability of both data and algorithms improved the performance of the class collection. Co-design provided a compilation context for SLAM abstractions to generate machine code equivalent to the C++ compiler. It was observed that immutability has its limitations in iterative environments where data escapes. Testing demonstrates that if mutability is re-inserted, Java can further outperform C++. However adding mutability should be considered an unnecessary amendment as the performance is still improved with immutability.

This chapter presents an approach to co-design abstractions for specific domains, in this case it is SLAM application. A new class collection for small vectors is co-designed with the phases of optimisation in the Graal compiler. This improves the performance by increasing the semantic information inferred from temporary objects and their values. It is achieved by improving the effect of constant folding and common sub-expression elimination and reducing the interaction with the garbage collector.

The dynamic compiler is also able to inline methods that are seen as boundaries in statically compiled code. As compilation occurs at runtime there are configuration parameters that may be seen as constants. This differs from static compilation where values are hard coded in source to achieve the same. This improves frequently executed SLAM kernels such as point transform where a camera configuration, used in projection, is discovered after the application is
started. This enables Java to exceed the performance of C++ libraries in some benchmarks derived from the Java implementation of LSD-SLAM.
Chapter 7

Accelerating Small Vectors

Computer Vision applications, in particular SLAM, contain many data abstractions based on vectors. Chapter 6 introduced a new Java implementation for vectors types which removes indirection to arrays with an emphasis on accelerating irregular operations. Vectors in SLAM are also used in regular operations that can make use of hardware acceleration. In C++ this is achieved by hand-optimisation; applications contain examples of vectorisation as intrinsic methods or inline assembly for each hardware architecture targeted. This affects the portability and programmability during development. The availability of SIMD units is ubiquitous but their use is not specialised in Java to benefit SLAM applications.

The focus of this chapter is to make the vector and matrix abstraction, co-designed for Java, compatible with SIMD instructions. It will use an approach based on the hand-optimisations observed in the applications surveyed but without explicit use in application code or classes. The existing hand-optimisations make use of commodity SIMD units with instructions for SSE (for x86 architectures) and NEON (for ARM based Android devices). Pure C++ is used if the architecture does not contain the supported SIMD operations, the same approach used in the Java implementation.

This chapter investigates extending the IR in Graal to communicate the semantics of the abstraction, deeper in the compiler, to the assembler. The vector abstraction in Java will be optimised to generate efficient instructions sequences for architectures containing SIMD units. In addition to the use of regular data operations, the optimisation utilises registers to store intermediate vector data. The outcome is an approach for co-design between a portable abstraction and improved performance on the available hardware.
The primary contribution in this chapter is an approach to accelerate data structures suitable for manipulation by SIMD instructions. Small vectors and matrices are ubiquitous in SLAM applications and SIMD is capable of accelerating the associated arithmetic. Existing approaches introduce vectorisation to Java but they are unsuitable for such small data structures because the overheads outweigh the benefits. As such, this chapter modifies Graal to further exploit the new class collection presented in Chapter 6. Basic vector and matrix operations are targeted for hardware acceleration; these are then used to compose SLAM kernels improving their performance.

7.1 Vectorisation

SIMD instructions permit concurrent application of operations to vector elements. The degree of concurrency is dependent upon the width of the registers used. For a 32-bit IEEE-574 single precision floating point value, four will fit in a 128-bit register, so up to four operations can be applied. Figure 7.1 illustrates this, with the addition of an array to another with the result stored in a third array in memory. Vector data is handled in single instructions as packed data (operations in x86 differentiate between a single scalar (SS) and four packed scalars (PS)). For example the packed addition is applied as a single instruction for all values (addps), as opposed to a single value addition currently used in Graal (addss).

7.1.1 Hardware

SIMD units are ubiquitous in commodity hardware with Intel introducing MMX instructions in Pentium processors for 64-bit SIMD operations. This instruction set has developed over the years to increase the register widths and concurrency but is common to many architectures. ARM based processors frequently contain a SIMD unit accessed via the NEON instruction set and it is available to many low-power devices including mobile phones. The availability is dependent on the architectural design of fabricated microprocessors.

MMX, SSE and AVX

Intel introduced multimedia extensions (MMX) to the Pentium series of processors in 1997 [28]. This instruction set provided new 64-bit registers with support
a) scalar float operation in 32-bit register

b) vector float operation in 128-bit vector unit register

Figure 7.1: Adding floating point values from an array.
for integer SIMD operations; eight byte operations could be applied concurrently. These instructions have been exploited in many applications, including visual odometry for input filtering [156]. Data could be moved from memory or existing registers into the new \( \text{mm} \) registers within the tightly-coupled MMX co-processor. As the hardware evolved, new instructions were added and new data types supported, notably adding support for floating point arithmetic.

Streamed SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) are co-processor updates that have increased the instructions available and the width of the registers. 128-bit support is available with the \( \text{xmm} \) registers; 256-bits with the \( \text{ymm} \) registers; and 512-bits with the \( \text{zmm} \) registers. Some instructions for the SIMD units are specialist and not useful for general purpose languages. For example the availability of saturation arithmetic (operations on numbers with a fixed range) increases the importance of correctly ordering arithmetic operations. However existing approaches and the work investigated in this chapter look at packed integer or floating point values. Table 7.1 contains the basic operations available in SSE that are targeted for the optimisation of vectors in SLAM applications.

For contiguous data, vectorisation is frequently applied with examples found in SLAM projects. However the alignment of data in memory and its length does require attention. The aligned move instruction (\text{movaps}) is faster in SSE than the unaligned equivalent (\text{movups}) but data must be aligned on 16-byte boundaries. The challenge in hand-optimisation is to make algorithms efficient and portable over the many subtly different SIMD unit architectures and data abstractions.

**NEON**

The ARM NEON instruction set provides 128-bit SIMD operations and is supported in the Cortex-A processors [41]. Optimisation using NEON in SLAM applications target ARM based Android devices. The use of mobile devices for computer vision applications stems from the increase in demand for augmented reality and object detection and identification in everyday situations. This is illustrated in the implementation of PTAM and LSD-SLAM for cameras on mobile devices [157, 127]. Unmanned Aerial Vehicles (UAV) or ‘drones’ use SLAM applications for their visual localisation, requiring real-time results with limited processing capability and energy supplies [16].
Table 7.1: SSE instructions targeted for optimisation.

The ARM NEON pipeline is a co-processor with data transfers between register sets. There are instructions for data transfer, manipulation, bitwise and arithmetic operations. While not targeted in this thesis, the methodology applies to the use of NEON and other architectures, for example Xeon Phi. Once there is full support in Graal for these architectures, the abstraction discussed in this chapter can be used to accelerate applications. There is also no need to re-write the application to support current and future developments in SIMD operations in processors.

7.1.2 Related Work

Despite SIMD units being available in commodity processors since 1997, there is still limited support in general purpose languages. C++-based SLAM applications manually specialise for hardware. There are a number of approaches taken to vectorise SLAM applications for commodity hardware.

Libraries

There are three concerns involved in programming with vectors for computer vision applications; a consistent abstraction, optimisation and hardware utilisation. To achieve all three is difficult, particularly the support for optimisation of a high-level abstraction for a specific hardware platform. Table 7.2 contains a summary of some computer vision libraries pertinent to SLAM applications with the Java equivalents for completeness. Within this list, it may be observed that
CHAPTER 7. ACCELERATING SMALL VECTORS

<table>
<thead>
<tr>
<th>Framework</th>
<th>Type</th>
<th>H/W Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCV [131]</td>
<td>Source and Library</td>
<td>GPU and Vector Unit</td>
</tr>
<tr>
<td>FastCV [118]</td>
<td>Library</td>
<td>GPU and Vector Unit</td>
</tr>
<tr>
<td>Eigen [115]</td>
<td>Template</td>
<td>Vector Unit</td>
</tr>
<tr>
<td>Sophus [116]</td>
<td>Template</td>
<td>Vector Unit</td>
</tr>
<tr>
<td>TooN [158]</td>
<td>Template</td>
<td>None</td>
</tr>
<tr>
<td>$g^2o$ [114]</td>
<td>Template</td>
<td>Vector Unit</td>
</tr>
<tr>
<td>Java arrays</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>JAMA [123]</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>(Parallel) Colt [124, 125]</td>
<td>Source</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 7.2: Computer vision libraries and optimisations.

the success in optimising for specific hardware is limited to OpenCV and FastCV; however performance is achieved by hand-optimised source code. The optimisations make use of abstract numerical properties and preprocessor macros. This leads to code that is difficult to understand for developers not directly involved in the implementation.

The Java Software Development Kit (JDK) does not contain support for numerical computing. In the past, the Java Numerics Working Group [148] investigated the inclusion of support for new hardware and improved programming techniques to encourage the use of Java within the scientific community. This failed to gain momentum and JAMA [123], along with other products in this initiative, became obsolete. Colt and Parallel Colt [124, 125] are high-performance libraries for numerical computing, the latter using Java thread pools for its concurrency. However, this is for data abstraction larger than the few elements representing dimensions in a point in the Euclidean space.

GeoRegression [150] is a pure Java library built upon EJML [151] designed for the computer vision and robotics domain. Points, both 2D and 3D, are represented as a class containing a field for each dimension. This approach improves the performance but each class requires an explicit implementation with explicit support in utility methods. Ideally there would be a single abstraction that may be optimised for the given hardware architecture and context of execution; in short, taking advantage of dynamic compilation.

Support for numerical computing in C++ is ubiquitous and the history more extensive. In the domain of computer vision Eigen [115], TooN [158] and OpenCV
[113] provide a class representation of vectors and matrices for numerical computing. They are commonly used as the building blocks for SLAM applications and libraries. Eigen is implemented entirely as a template library making full use of the meta-language to aggressively optimise for performance, including loop unrolling and deferred evaluation. What it cannot do, however, is to make use of new hardware, as it is designed and implemented as header files to be hardware independent.

OpenCV is a library for computer vision applications with support for Intel architectures with compilation options to make use of SIMD and the Integrated Performance Primitives (IPP) library in addition to GPU acceleration. This is commonly used as the base for a range of computer vision algorithms extending beyond the scope of SLAM. OpenCV has spawned a number of wrappers for Java interoperability [132, 131] as well as inspiring the creation of BoofCV [134], a pure Java computer vision library. Besides OpenCV, which is closely tied to proprietary modules, there are few attempts to exploit the advantages of vector units; this is why there are many instances of hand-optimisation in SLAM applications.

**Vectorisation in Java**

The most common approach to make use of SIMD instructions in Java is to create a native library and use JNI to interact with this directly. Parri et al. [159] 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 this. However this is fixed behaviour and does not make use of the hardware available to the JVM. The real restriction of this approach is that the contents of native methods cannot be inlined and optimised as part of the IR during compilation. There is no ability to prevent allocation of the vector on the heap and the vector. This is another source of inefficiencies in vectorising libraries accessed using wrappers and why they are not suitable for SLAM with its small vectors.

**7.1.3 Hand-optimisation in C++**

LSD-SLAM and other applications use a type definition for the matrix abstraction provided by Eigen [115] (as shown in Figure 6.1.a). In SLAM applications observed and implemented using C++, Eigen is frequently used as a library to
provide a matrix and vector abstraction for algorithms. This allows the definition
of data types for vectors; including position, gradients, rotation and rigid body
transforms. Rather than extending the APIs used there is evidence of hand-
optimisation obfuscating algorithms used in the application. Figures 5.1.b and
7.2 illustrate the issues when an abstraction does not provide support for the
desired use case. The code of algorithms is verbose and its purpose in not quickly
discernible.

Table 7.2 contains an overview of the libraries available for use in SLAM ap-
plications. In the compromise between programming and performance, Eigen has
been designed as a header library and therefore it may be used on any platform;
however this restricts hardware specialisation. Over the years it has evolved and
the template meta-language has been used to force optimisations such as deferred
evaluation and loop unrolling. This is an optimisation for performance, produc-
ing efficient machine code from a pure software abstraction. A downside to this
is that it becomes very difficult to debug due to the complex data structures,
method indirection, casting and static nature of templates during runtime. Eigen
is an expertly developed library and well supported; however it is imperative that
it is extended and clean abstractions developed for it in SLAM applications.

7.2 Accelerating Small Vectors in Java

Modern commodity processors contain vector units to accelerate arithmetic op-
erations for contiguous data and streams. Often associated with multimedia ap-
plications, SIMD instructions are ubiquitous in hardware. The implementation
of LSD-SLAM contains hand-optimised code for two architectures; SSE intrinsics
for x86 based architectures and NEON assembler for ARM microprocessor based
mobile devices. Each hand-optimisation uses the 128-bit vector unit registers
directly or indirectly in source code.
In programming SLAM applications there is a distinction between small vector types; despite this not being evident in many implementations. A single vector type is used in LSD-SLAM extending the \texttt{Matrix} template class provided in Eigen. This means that all types have access to the same methods but this is problematic as any length vector can access the cross product despite it only provides a meaningful result for 3D vectors. This approach does permit limited specialisation for vector units using SSE intrinsics and is implemented as architecture specialisations (only SSE is supported).

Another benefit of vectorisation in LSD-SLAM is the increase in performance of regular data algorithms. In LSD-SLAM the main technique used to achieve this is to insert optional source code using intrinsics, or assembly, protected by preprocessor directives. By profiling Java it is possible to observe performance limitations elsewhere but tracking, where the optimisations appear, is the algorithm that affects data throughput. The main restriction of this approach is that algorithm changes must be implemented in three locations using different syntax, increasing the risk of errors.

Chapter 6 presented a new abstraction for small vectors in Java and demonstrated that, with a co-designed modifications to the dynamic compiler to make it aware of the semantics, it is able to exceed C++ performance for irregular algorithms: the $\text{se}(3)$ logarithm and Levenberg-Marquardt update. The optimisation removes the overheads of encapsulation and indirection for operations suitable for vectorisation. This chapter uses co-design to allow Java to utilise the vector unit with instruction sequences developed over time in the SIMD application research.

### 7.2.1 Java Abstraction

Chapter 6 presented an implementation of the \texttt{AbstractV128} abstraction class, e.g. \texttt{Vector3D}. This provides a specialisation by removing the indirection used in larger vectors that may be accelerated using a different mechanism (e.g. Jacc [121]). Java is able to treat the individual elements as primitive types and virtualise them in registers rather than placing them on the stack, improving performance. The elements, or values, in this class are laid out side-by-side and, as it was co-designed with the JVM, it is possible to assert that they are contiguous in memory starting at the first element ($v_0$). SIMD registers, in targeted hardware, are at least 128-bits wide which, by design, matches the four floats stored in the class. This allows \texttt{AbstractV128} sub-class instances to be virtualised on the
stack or, better, contained within registers. The aim of this chapter is to make use of these wide registers to store all data in the same way Java does currently for instances of primitive types.

The implementation for the abstraction has been presented in Chapter 6 and incorporated into the port of LSD-SLAM to Java. It has been possible to demonstrate the potential for speedup by making the optimiser aware of the class structure and properties. This chapter now looks at moving the semantics of this class deeper, extending the communication from the high tier phase of the dynamic compiler to the assembler that generates the machine code. As with many compromises in optimisation, the ability to utilise the vector unit is not free because the abstraction and hardware are co-designed to interact closely. This means that padding is required for 2D and 3D vectors to fit in the 128-bit registers without corrupting data when writing back to objects. This affects the efficiency of objects allocated in memory. As there is the possibility of virtualisation of object placement, many vectors in optimisation will never be allocated and reduce this overhead once compiled.

### 7.2.2 Vector Operations

Vector arithmetic operations are directly translatable to SSE or NEON instructions, e.g. the `addps` will add two vectors together. The other operations that need to be supported are move and shuffle (Table 7.1 contains a list of SSE instructions targeted by the optimiser for the new abstraction). The move provides the interface between memory and registers, loading from and storing to the stack. This can then be materialised as an object allocated on the heap. The implementation in this chapter does not extend to the garbage collector or stack allocator so it is not possible to align vectors on the boundaries required for an aligned move (always using `movups`). The other operation, shuffle, is very useful in manipulating the elements within a vector or shuffling the elements of two vectors. It uses an immediate value in the instruction to control the re-ordering of elements in the vector unit register targeted. Figure 7.3 illustrates how the cross product may be accelerated using the vector unit based on shuffle and arithmetic operations. A full list of vector operations supported in this chapter can be found in Appendix D. While it may not make a large impact on the execution time alone it may use virtualised vectors in registers and leave the result as a virtual vector in registers for use elsewhere. Without vectorisation this must always exist.
a) implementation using Java arrays

...  
                        a[2] * b[0] - a[0] * b[2],  
...

b) implementation using SSE instructions

...

    movups xmm0, XMM PTR &a  ; Load vector a into xmm0
    movups xmm1, XMM PTR &b  ; Load vector b into xmm1
    pshufd xmm2, xmm0, 0xC5 ; { a1, a2, a0, 0.0f }
    pshufd xmm3, xmm1, 0xD2 ; { b2, b0, b1, 0.0f }
    mulps xmm2, xmm3 ; Multiply temporary vectors
    pshufd xmm0, xmm0, 0xD2 ; { a2, a0, a1, 0.0f }
    pshufd xmm1, xmm1, 0xC5 ; { b1, b2, b0, 0.0f }
    mulps xmm0, xmm1 ; Multiply temporary vectors
    subps xmm2, xmm0 ; Subtract temporary vectors
    movups XMM PTR &result, xmm2 ; Store result from xmm2
...

Figure 7.3: Cross product using SIMD instructions.
on the stack or heap.

### 7.2.3 Matrix Extensions

SIMD instruction sequences are used for matrix operations and the `AbstractV128` abstract class is accompanied by the `AbstractM128` abstract class with a specialisation for up to $4 \times 4$ matrices, targeting rotation and transform matrices for SLAM applications. The four rows are stored as a concrete sub-class of `AbstractV128`, the operations are affected by the same optimisations that apply to vectors. Figure 7.4 demonstrated how a SIMD unit may use SSE instruction to perform a matrix-vector multiplication, as used in the point transform SLAM kernel. The code in Figure 7.5 contains the implementation for the matrix-vector multiplication that may use the capabilities of the SIMD unit. The alternative approach is to store the matrix as column major (Figure 7.5 is row major) as this allows the multiplication with a vector to use multiplications and adds.
... 01 protected final <V extends AbstractV128> 02 V multiplyImpl(AbstractV128 vector, V type) { 03 return type.instantiate( 04 row0.dotImpl(vector), 05 row1.dotImpl(vector), 06 row2.dotImpl(vector), 07 row3.dotImpl(vector)); 08 } ...

Figure 7.5: Java implementation for small matrix-vector multiplication.

... 01 vector.multiply(a).add(b);
...

Figure 7.6: Snippet to demonstrate a compound vector operation in Java.

7.3 Extending Graal for Vectors

Graal is a meta-circular optimising compiler for Java; meaning it is written in Java, generating machine code from Java bytecode [5]. It is designed for extensibility and is suitable to evaluate the co-design of abstraction and optimiser. It uses an IR that contains both control and data dependencies and is linked to other research projects because it has the ability to represent the language agnostic philosophy of the JVM [10]. It is possible to use Graal as the compiler for the HotSpot VM and, as such, is focused on the method as its level of compilation abstraction. Within Graal, the IR is maintained as a structured graph with nodes representing actions or values and the edges represent their the dependencies. The graph is initially generated by parsing the bytecode from the class file. Figure 7.7 contains the initial graph generated for the example in Figure 7.6, a compound vector operation.

It is possible to observe how limited the opportunities for optimisation are before inlining. The frame states are omitted for clarity but these maintain information about the liveness of locals and the contents of the stack during execution. These are used in subsequent optimisations but also de-optimisation, so they are an important complication in the modifications proposed for Graal.
CHAPTER 7. ACCELERATING SMALL VECTORS

An objective in this chapter is to reduce the distance between vector operations in the IR to enable further optimisation through virtualisation. Using partial escape analysis, temporary vectors may be maintained entirely within the SIMD registers (xmm0 – xmm15 when using SSE). The challenge is in constructing a set of assumptions that are safe and used to extend Graal without affecting existing compilation techniques used for general programming.

Figure 7.8 contains the ideal final sub-graph transformed from the original by the optimisation phases. It can be observed that the distance between the packed scalar multiplication and addition has been reduced so they are connected with a single edge. The address of the vector for the read and write is treated as a primitive Java type. However this is not an inherently safe usage of the Java semantics so the assumptions made must be justified to enable the production of this target graph.

7.3.1 Assumptions

The main assumptions made to justify the data abstraction and to extend the compiler to support SIMD code generation are:

- Hardware supports 128-bit vector operations; this is true for ARM NEON and Intel SSE implementations. This restriction allows the inclusion of all values for the dimensionality of 2D points, 3D points, quaternions and
Figure 7.8: Desired IR sub-graph of compound vector operation.
translation vectors. It provides a domain specific optimisation so that it is possible to be more aggressive in transforms than for general purpose applications.

- The class contains four single-precision floating point numbers. This is what fits in hardware and also supports all the targeted operations for the computer vision domain. This allows optimisation based on a constant length.

- Unused elements in the vector are zero. This simplifies the operations for vectors that do not fill the entire width of the register. Zero will be propagated through all operations except division; in which an error condition could be generated and propagated to further instructions.

- The elements in the vector are contiguous in memory. Graal may be built into the HotSpot VM. One implementation is the OpenJDK project [2] and this has been selected for this research. The advantage is that it is possible to inspect the implementation of object fields. For the environment selected this assumption is true but cannot be guaranteed for other JVMs.

- Once constructed, the vector is immutable. It is not possible to enforce immutability in Java, especially with the availability of theUnsafe proprietary class from Sun but distributed with the JDK. However with a considered approach to programming, it is possible to restrict any write access with the following steps:
  - The elements are written once, in the constructor,
  - The constructor for the specialisations ensure validity, and
  - The class will be implemented in the JDK to prevent unsafe access.

Together the assumptions dictate the implementations within theAbstractV128 class but allow some of the restrictions in Java to be eliminated. This enables the IR to be extended and be more aggressive in the optimisation, as the semantics are now within the vector abstraction, not the general purpose language.

### 7.3.2 Adding Small Vectors to the IR

The Graal IR [5] is designed to be extensible and there are two mechanisms for substitution in Graal. Method-substitution provides an alternative method
body to replace any invocation of a specific target method. Java does not contain support for SIMD operations so there is no equivalent code to substitute for the method, so it is not possible to use this approach. Macro-substitution replaces the method invocation with a special IR node that is later lowered to a specialised implementation. An example in Graal is the substitution of the System.arraycopy(...) method, creating a node that may be specialised for the type being copied.

Macro-substitution has been chosen because it enables the replacement of a method invocation with a predefined node or sub-graph, when lowered, in the IR. This enables the augmentation of the optimising compiler for Java without modifying any components directly related to Java. The optimisation may be isolated for the transforms associated with the vectorisation of the AbstractV128 class methods. The main separation of concerns, as shown in Figure 7.9, is the validity checking of the argument. By design, the vector contains a non-null array with a matching field to hold data for use in reads and arithmetic.

The use of separate instance methods and implementation of the arithmetic allows the isolation of input verification and functionality. This is important to the process of de-optimisation as the frame state must be available where exceptions (explicit or implicit) could arise. The implementation of concrete classes (e.g. Vector3D) must check inputs to simplify the use of macro-substitution in the optimisation. The add implementation in Figure 7.9.b will be substituted as line A5 in Figure 7.9.a without affecting the guard introduced in line A3. The 128-bit wide data is not understood in Java so, once optimised and compiled, it must be completed and the frame state restored before any de-optimisation guards are reached. The verification ensures that the failure conditions (null or incompatible vectors) are handled before Java behaviour is modified. The use of the protected final arithmetic methods provides frame state encapsulating the vectorisation without the need to modify Graal.

7.3.3 Vector Operation Nodes

The vector operations are represented as nodes that replace the associated method invocations in the IR graph. Using the knowledge from the implementation of the class and operation of the JVM it is possible to reference the vector as a single group based on the first element of the vector. This is made possible because the values are immutable and contiguous in memory. The movement of vector
a) add interface in Vector3D class

```java
@Override
public Vector3D add(Vector3D other) {
    Objects.requireNonNull(other);
    return addImpl(other, this);
}
```

b) add implementation in AbstractV128 class

```java
protected final <V extends AbstractV128> V addImpl(
    AbstractV128 other,
    V type)
{
    return type.instantiate(v0 + other.v0,
                            v1 + other.v1,
                            v2 + other.v2,
                            v3 + other.v3);
}
```

Figure 7.9: Implementation of Vector add method.
Figure 7.10: Sub-graph used to replace vector nodes in IR.
values and the vector arithmetic operations may then utilise SIMD instructions. Figure 7.10 contains the sub-graph inserted in the IR during optimisation of the vector methods. Appendix D lists all operations supported by the nodes added to Graal, each description includes the instructions emitted during code generation. There are two phases of optimisation once the vector nodes are inserted into the IR graph. The first replaces the node with the sub-graph during the first lowering phase. The second removes the need for intermediate vectors to be moved from the registers. The steps involved in achieving this are:

1. Create a new structured graph applying the desired vector operation to the values accessed using the first element in the vector using a field access node. A new instance of the vector type is created to write the result of the vector operation to. The exact type is obtained from the type argument provided to each of the operation method implementations.

2. Lower the graph; this has the effect of replacing the field access nodes with an object read or write node using the displacement of the field and instance address as the address.

3. Traverse the data-flow and control-flow graph identifying vectors that do not escape and may remain as virtual objects. Delete the read node and the associated write and new instance nodes, connecting the output of the preceding vector operation to the usage in the current sub-graph. This results in the data-dependency like that shown between vector operations in the example in Figure 7.8.

The lowering method of the macro node is extended in the vector nodes to achieve this. This differs from existing virtualisation in Graal as it is possible to store the entirety of the object in the SIMD unit registers. At this point the optimisation in the front end of the compiler is complete and no further transformations are made until the LIR nodes are generated and machine code emitted. Each of the nodes added to Graal is used to generate machine code but first they are lowered to the LIR used in Graal. This representation is used to ensure control flow, implement low-level optimisations and also allocate registers. The vector operation nodes generate sub-graphs for inclusion in the LIR graph declaring the number of variables required. Registers are allocated based on the liveness of variables in the graph and assigned registers are used in the final phase of code generation.
7.3.4 Code Generation

Nodes in the LIR emit machine code and Figure 7.3 contains the output from the cross product node. Two registers are allocated to the vector read values (xmm0 and xmm1) and two registers are allocated for temporary values (xmm2 and xmm3). The write node uses the vector address and its input data dependency to the result (xmm2). In the cross product there is a shuffle operation (pshufd) used to re-arrange the values in a temporary register. In addition to the arithmetic operations, two further LIR nodes have been added to Graal to support multiplication, division, the cross product and Hamilton product. xorps is used to negate values in Hamilton product by flipping the sign of floating point numbers using a mask. pshufd used an immediate operand to determine the order of elements in the destination register. This instruction is used to extend a single scalar into a packed scalar for multiplication and division; the shuffle repeats the value so the operation may be applied to all values in the vector.

7.4 Performance Evaluation

The objective of this performance evaluation is to establish the benefits of using SIMD instructions to accelerate SLAM kernels. The optimisation targets small vectors and matrices that extend the AbstractV128 and AbstractM128 classes. Operations used in these classes and added to the Graal IR, using the macro-substitution mechanism, are also evaluated in isolation.

There is a limitation to the code generated because the mechanism by which registers are made available is managed in the JVM. The runtime environment in the OpenJDK project is unable to parse SIMD instructions when installing compiled methods. This means that it is not aware of the data size that needs to be reserved for the persistence of SIMD register data spilled to the stack. Therefore, it is not possible to evaluate the SLAM kernels using the JVM. Despite this there is the opportunity to isolate and evaluate the SIMD code generated by Graal in a C-based harness. The evaluations of the operations and SLAM kernels use the code generated by the optimisations introduced in Chapter 6 as reference.
CHAPTER 7. ACCELERATING SMALL VECTORS

### Hardware

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Core i7 3.4GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
</tr>
<tr>
<td>Hardware threads</td>
<td>8</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32kB per core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256kB per core</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8MB per 4 cores</td>
</tr>
<tr>
<td>Main memory</td>
<td>16GB</td>
</tr>
<tr>
<td>Vector Units</td>
<td>SSE4.2 and AVX2</td>
</tr>
</tbody>
</table>

### Software

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Windows 8.1</td>
</tr>
<tr>
<td>C/C++ compiler</td>
<td>gcc 4.8.3</td>
</tr>
<tr>
<td>Java</td>
<td>OpenJDK 1.8.0_internal</td>
</tr>
<tr>
<td>JVM</td>
<td>OpenJDK 64-Bit Graal VM</td>
</tr>
</tbody>
</table>

Table 7.3: Configurations for vectorisation performance evaluation.

### 7.4.1 Experimental Set-up

#### Hardware and Software Configuration

The experiments run on a single hardware platform, exploring the performance on a workstation that supports the SSE instruction set. Table 7.3 presents the hardware and software configuration used to host the evaluations.

The JVM from the OpenJDK project [2] is used to compile the machine code used in this preliminary performance analysis. The C++-based server compiler is replaced by Graal [4] with the modification to exploit SIMD instructions. Each test is the mean execution time of 1,000,000 iterations of the machine code being tested. The time taken by the test harness to execute a `nop` is used to derive the execution time of the instruction sequences implementing the operations and SLAM kernels. Relative performance is calculated by using the execution times after this overhead has been taken into account.

#### Vector and Matrix Operations

The operations used in the SLAM kernels manipulating small vectors and matrices are basic arithmetic operations on data in the Euclidean space. For the
purposes of testing these are separated into two categories: the first vector operations; and the second matrix operations including interaction with vectors. The operations are:

- **Vector Operations**
  - **Add** – element-wise addition between two vectors (also represents the execution time for subtraction).
  - **Divide** – elements in a vector are divided by a scalar value.
  - **Multiply** – elements in a vector are multiplied by a scalar value.
  - **Cross Product** – an operation on 3D vectors calculating the normal to a plane containing the two input vectors.
  - **Dot Product** – the sum of element-wise multiplication between two vectors, it calculates the product of the magnitudes of the two vectors.
  - **Hamilton Product** – an operation to multiply two quaternions representing rotation and scale in computer vision algorithms.

- **Matrix Operations**
  - **Add** – element-wise addition between two matrices (also represents the execution time for subtraction).
  - **Divide** – elements in a matrix are divided by a scalar value.
  - **Multiply** – elements in a matrix are multiplied by a scalar value.
  - **Multiply Vector** – multiplication between a matrix and a vector, results in a new vector, this operation benefits from the matrix being stored as column major and is ubiquitous in SLAM algorithms.
  - **Multiply Matrix** – multiplication between two matrices, results in another matrix.

The operations are executed with the code generated by compiling the Eigen library for each operation; the Java code emitted from Graal; and the new SIMD acceleration. The operations have also been evaluated with the data assumed to be, and accessed, *aligned* on and *unaligned* across 128-bit boundaries. The GC has not been modified so the data is unaligned in all operations in Graal, accessing this using `movaps` or as an argument to an arithmetic operation causes an error.
CHAPTER 7. ACCELERATING SMALL VECTORS

<table>
<thead>
<tr>
<th>Operation</th>
<th>nop</th>
<th>Eigen</th>
<th>Java</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>unaligned</td>
</tr>
<tr>
<td>Add</td>
<td>1.973</td>
<td>3.330</td>
<td>3.937</td>
<td>2.475</td>
</tr>
<tr>
<td>Divide</td>
<td>1.879</td>
<td>5.355</td>
<td>7.070</td>
<td>2.590</td>
</tr>
<tr>
<td>Multiply</td>
<td>1.943</td>
<td>3.112</td>
<td>3.561</td>
<td>2.616</td>
</tr>
<tr>
<td>Cross Product</td>
<td>1.981</td>
<td>3.008</td>
<td>3.729</td>
<td>2.793</td>
</tr>
<tr>
<td>Dot Product</td>
<td>1.910</td>
<td>2.912</td>
<td>3.298</td>
<td>2.711</td>
</tr>
</tbody>
</table>

Table 7.4: Execution time of vector operations (in nanoseconds).

condition that affects execution time. The vectors and matrices used during these operations are 3D, except the Hamilton product that uses a quaternion with four elements.

SLAM Kernel Benchmarks

The SLAM kernels introduced in Chapter 6 will be used to evaluate the performance of the accelerated optimisations as they are composed of the vector and matrix operations. The results generated from these use the code emitted from Graal and tested in the C-based test harness as with the operations.

The exception is the Levenberg-Marquardt Update because this uses a vector length greater than the implementation in Graal. It could be extended to use two SSE instructions for each operation but the type system in Graal has not been extended to be aware of this. The kernel is hypothesised not to benefit from the SIMD acceleration as the performance improvements presented in Chapter 6 are mainly from constant folding and common sub-expression elimination. The SIMD approach does not consider the values of elements in the vectors or matrices so cannot exploit the symmetry of the Jacobian matrix or the use of the weighting in both parts of the system.

7.4.2 Results

Tables 7.4 and 7.5 contains the execution times in nanoseconds for each of the nodes added to Graal. The vector operations that are improved the most are division and the Hamilton product, the SIMD variant accelerates the Java by 7.30 and 5.46 times respectively. The improvement in performance of matrix operations between Java and the SIMD variant is 2.76–3.84 times except for
Table 7.5: Execution time of matrix operations (in nanoseconds).

<table>
<thead>
<tr>
<th>Operation</th>
<th>nop</th>
<th>Eigen</th>
<th>Java</th>
<th>SIMD unaligned</th>
<th>SIMD aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide</td>
<td>1.912</td>
<td>16.219</td>
<td>29.231</td>
<td>7.154</td>
<td>7.135</td>
</tr>
<tr>
<td>Multiply</td>
<td>1.933</td>
<td>5.670</td>
<td>12.130</td>
<td>3.608</td>
<td>3.536</td>
</tr>
<tr>
<td>Multiply Matrix</td>
<td>1.906</td>
<td>10.070</td>
<td>19.724</td>
<td>8.351</td>
<td>8.214</td>
</tr>
</tbody>
</table>

Figure 7.11: Relative performance of vector operations.
scalar multiply and division where the instruction latencies are larger, the relative performance is 5.21 and 6.08 respectively.

Figures 7.11 and 7.12 contain the relative performance of the code generated by Eigen and that of the new SIMD accelerated Java operations. All SIMD operations are able to outperform the Eigen code; although there no significant difference for the Hamilton product operation because the Graal node emits very similar machine code. The difference is the bit that is flipped to change the sign of one of the vector elements during the algorithm, Java is able to load a scalar to achieve this but Eigen loads a constant 128-bits wide to achieve the same. The matrix operations are more consistent in their relative performance except matrix multiplication which is only improved by 1.27 times although the padding means that Java is executing a $4 \times 4$ matrix but Eigen, because it uses templates, operates on a $3 \times 3$ matrix during the evaluation.

Table 7.6 contains the result for using the C-based test harness to execute three of the four SLAM kernel benchmarks from the co-designed implementation and using SIMD instructions. The results show an improvement for Point Transform and Gradient Interpolation of just over 60%. These are small kernels with no control flow so make full use of the SIMD operations. The SE(3) Logarithm,
and hypothetically the Levenberg-Marquardt Update, are slower using SIMD instructions. The co-design is 1.12 faster than the work for this chapter, this can be attributed to the failure to exploit constant folding when using SIMD instructions.

### 7.5 Discussion

The improvement in basic vector operations (add, subtract, multiply and division) is noticeable against both Eigen and Java; reflecting the improvement in latencies introduced in Appendix D. These are also the operations that are used to compose the matrix operations, thus, the results are similar for these as well. All operations make use of the low-level concurrency and the reduction in instructions executed that move data between memory and registers.

The cross, dot and Hamilton products are specialised operations for SLAM and, as such, are optimised in Eigen. In these operations the more significant improvement is for Java, upgrading the co-design to exploit the hardware acceleration available. The machine code generated for the cross and Hamilton products use SIMD instructions in Eigen. Using the C++ framework requires manual inclusion, especially for the cross product because it is applicable to 3D vectors but requires a $4 \times 1$ matrix to allow for padding. This is hidden in the abstraction in Java but affects the programmability when using the Eigen library.

The SLAM kernels can be divided into two groups: those with constants, SE(3) Logarithm and Levenberg-Marquardt Update; and those with variables, Point Transform and Gradient Interpolation. The first is improved by co-design and the latter by SIMD instructions and it would be useful to select the best approach during compilation. Unfortunately, it is not possible to execute the

<table>
<thead>
<tr>
<th></th>
<th>Point Transform</th>
<th>SE(3) Logarithm</th>
<th>Gradient Interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>2.185</td>
<td>1.922</td>
<td>2.821</td>
</tr>
<tr>
<td>Co-design</td>
<td>10.976</td>
<td>29.319</td>
<td>8.265</td>
</tr>
<tr>
<td>SIMD</td>
<td>7.508</td>
<td>32.536</td>
<td>6.204</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.65</td>
<td>0.89</td>
<td>1.61</td>
</tr>
</tbody>
</table>

Table 7.6: Execution time and speedup of SLAM kernels using SIMD instructions.
kernels in LSD-SLAM but it does indicate the potential for additional performance improvements.

By inspecting the code generated by Graal for the SE(3) Logarithm, it reveals that the main inefficiency is in the creation of the hat of the SO(3) Lie group in the algorithm. It is a $3 \times 3$ matrix but it only contains three unique values (ignoring signs). So despite the low-level concurrency by using SIMD instruction, constant folding and common sub-expression elimination out-performs the approach of this chapter. This is because the arithmetic is simplified and explains why there is a lot of hand-optimisation in C++ implementations of SLAM applications.

As an extension to the evaluation, a benchmark was created for the matrix-vector multiplication operation illustrated in Figure 7.4. The original code generated by Graal, before any of the vector operation nodes were added, has an execution time of 5.104ns. The benchmark uses a 3D rotation matrix and vector, making use of the constant folding of padding presented in Chapter 6. The code that is generated by Graal after the optimisation is slower and has an execution time of 6.906ns. It would appear that the mulss instruction used to multiply two floating point numbers (in this case with operands in a register and memory) is more efficient than the dot product instruction (dpps) used in the emitted code. By rewriting the machine code manually to first transpose the matrix (using the algorithm in Figure 7.13) the matrix-vector multiplication may be achieved with SIMD multiplications and adds. The result of this in an execution time of 2.826ns; this is a speedup of 1.81 times.

The results demonstrate a reduction in execution time for all vector operation node equivalent implementations that are unable to make any additional compilation assumptions. However when creating SIMD instructions for larger code, there are obstacles to overcome and it becomes difficult to maintain the performance improvements initially observed. The SIMD nodes in the LIR of Graal need to be aware of the best instructions to select to address the inefficiency of some instructions (e.g. the dot product or division). An approach that could be applied to address this is instruction selection [160]. This technique matches patterns in the LIR and uses cost-functions to select the closest selection to optimal it can find. For this work this is used as the justification to storing matrices as column major as the transpose, despite being available, is no longer required.

There are still elements missing in the implementation that have not yet been addressed. The single most important limitation in implementation is the lack
Tier 1
\[
\begin{align*}
\text{shufps} & \quad \text{row0, row1, 0x44} \\
\text{shufps} & \quad \text{row1, row0, 0xEE} \\
\text{shufps} & \quad \text{row2, row3, 0x44} \\
\text{shufps} & \quad \text{row3, row2, 0xEE}
\end{align*}
\]

Tier 2
\[
\begin{align*}
\text{shufps} & \quad \text{row0, row2, 0x88} \\
\text{shufps} & \quad \text{row0, row2, 0xDD} \\
\text{shufps} & \quad \text{row1, row3, 0x88} \\
\text{shufps} & \quad \text{row1, row3, 0xDD}
\end{align*}
\]

Figure 7.13: Transpose of a $4 \times 4$ matrix using SIMD instructions.

of extension of the packed scalar type into the OpenJDK runtime environment itself. This is essential for complex algorithms like the SE(3) Logarithm as there are not enough registers to contain all intermediate data. When this occurs data is transferred to the stack for persistence while the registers are used for other calculations. This change would need to be made to the assembler in the OpenJDK project to parse SSE instructions and augment the JVM with an understanding of the 128-bit data type. Another cause of this behaviour is the register allocation algorithm used in Graal. OpenJDK and Graal and their derived JVMs (including HotSpot) use linear scan to allocate registers \[161\] and so in SE(3) Logarithm there is not an optimal utilisation of registers, thus, values are spilled to the stack. However, this still provides better performance than using SIMD instructions for the same.

7.6 Summary

As a general purpose programming language Java does not have to perform domain specific optimisations. SLAM applications provide many opportunities for specialised optimisation on commodity hardware that are not currently exploited. Potential optimisations are manually developed for C++ SLAM applications but this is at the expense of programmability and portability. Existing approaches
to using SIMD units from Java target acceleration of large data structures that are not ubiquitous in SLAM applications. The co-design has made vectorisation possible for small vectors by moving the semantics of the application to the implementation of the class collection. The optimising compiler is then able to use this to reduce the volume of arithmetic executed (Chapter 6) or accelerate execution with SIMD instructions.

The AbstractV128 and AbstractM128 classes are supported by additional co-designed optimisations in Graal to support SIMD instructions. As there is no native code, the optimisations are as portable as the Graal and OpenJDK projects and their internal assemblers. The new approach to support vectors demonstrated potential for SLAM applications introduced in Chapter 6. This is different to existing approaches incorporating vectorisation in Java as it does not unroll loops or streams to a SIMD unit.

The introduction of SIMD instructions improves the execution time of vector and matrix operations in Java (1.73–7.30 times faster). These show similar or higher performance than the Eigen library used by the original C++ implementation of LSD-SLAM. Using these operations to implement the SLAM kernels improves performance over the co-designed approach, with 1.61 and 1.65 times speedup for two of the kernels. The work presented in this chapter is not a complete solution; it has successfully communicated the application semantics to the LIR in Graal, but further work is required to produce improved SIMD instruction sequences and work seamlessly with the JVM. It demonstrates improved machine code but a challenge remains in how best to implement instruction selection to benefit the widest audience. The modification could be added directly to the Graal IR for this specific case but this constitutes hand-optimisation and is not an ideal solution.
Chapter 8
Conclusions

This thesis presents co-designed approaches for optimisation to improve performance of applications whilst maintaining productivity. In static compilers, a frequent compromise exists between program size and execution time. For example, unrolling a for loop may reduce execution time by reducing conditional jumps, but will generate more instructions. This decision is static and made at compile time so frameworks explicitly communicate potential optimisations by extending their APIs, often unnecessarily, or leading to complex source code. MapReduce and SLAM applications demonstrate that optimisation opportunities are missed when programmability is prioritised.

When looking at these particular domains, there are application semantics that are not exploited for optimised execution. MapReduce cannot optimise between tasks and so cannot improve the efficiency of intermediate values and reduce the execution time. Despite the data-dependencies, the compiler cannot view both the map and reduce tasks. Computer vision application optimisation prioritises execution time and, increasingly, energy efficiency. Hand-optimisation is used to communicate application semantics to improve the execution time of the machine code generated. This restricts the maintainability of code and reduces its portability to new, quickly evolving hardware architectures.

Productivity concerns the compromise between the resources spent developing an application and the improvement in its execution time. Programmability is of particular interest in this thesis, primarily striving towards the engineering principles of object-oriented software. Three approaches of co-design explore and address the reasons for hand-optimisation for the two domains used.
8.1 MapReduce for Java

MapReduce is a parallel software framework to reduce the execution time of data analytics. An intermediate data set is introduced to facilitate two phases of perfectly parallel execution. The map phase transforms the input into a set of intermediate (key, value) pairs, grouping them by the key. The reduce phase generates the output by combining all values associated with each key to a single value. Resulting in an output of (key, value) pairs, the running example is counting the words in a document with (word, count) pairs.

This thesis discusses the three Phoenix MapReduce frameworks developed for shared-memory, multi-core computing [62, 63, 64]. These have evolved with hand-optimisations used to reduce the execution time. Each framework improves the execution time and scalability of its predecessor. However the complexity of implementing benchmarks increases, as does the risk of runtime exceptions.

MapReduce for Java (MR4J) is a new parallel framework implementation presented in Chapter 3. It used standard Java collections and concurrency and runs the same seven benchmarks as Phoenix (ported to Java). Over these, it achieved comparative execution time to Phoenix 2.0 but ran 1.5 times slower than Phoenix++. On closer inspection, time is lost managing the intermediate values within the garbage collector. All versions of MapReduce, distributed and multi-core alike, introduce a combiner sub-phase. Combining reduces local intermediate values before the reduce phase; limiting the volume of data transferred over networks or eliminating the reduce phase completely.

A co-designed optimiser was developed to automatically rewrite reduce methods, inlining them into the map phase. The benchmarks remain untouched, yet it is possible to restructure the execution to improve garbage collector interaction and reduce execution time. The optimiser is able to make assumptions about implementations despite the distance between the map and reduce phases due to the semantics of the MapReduce framework. Phoenix hand-optimises benchmarks to achieve the same, but introduces duplication of code, assumes knowledge of the framework and violates encapsulation. The potential improvement in execution time of MR4J with the optimiser is twofold. Overall the improvement is a speedup to within 17% of Phoenix++ [64]; but importantly, it now uses less memory, allowing the analysis of larger input data.
8.2 Optimising Small Vectors

LSD-SLAM [17] is a tool to map an environment whilst calculating the location of the camera and is demonstrated on workstations and mobile devices [127]. Its algorithms use vectors to represent the arithmetic but these are not specialised for the application and abstractions are borrowed from scientific computing. There are arithmetic properties that cannot be assumed in a general purpose compiler. Hand-optimisation is used to communicate these properties to improve the execution time of the machine code generated.

This thesis presents a co-designed class collection (Chapter 6) for small vector data types with an understanding of the compilation transforms applied. The Graal compiler has been modified to enable additional assumptions to be made based on the arithmetic properties made. One such example is the inlining of data during the Levenberg-Marquardt algorithm. This improves performance because it uses a symmetric matrix and, thus, common sub-expression elimination is important. The co-design of the Java class collection allows performance of this algorithm to be 2.72 times faster than the equivalent C++ code used in LSD-SLAM. When applied to the LSD-SLAM tracking algorithm for Java the performance improves 5.66 times.

Chapter 7 presents an approach to use the structure and immutability of the data implementation of small vectors to utilise SIMD instructions. It adds new vector operation nodes to the Graal IR [5] that may be optimised and emit SIMD instruction sequences. Modifications have been made to Graal but the code generated is not compatible with the JVM. Therefore, the machine code has been tested in isolation and demonstrates an improvement in Java for all vector and matrix operations that form the SLAM kernels. The kernels themselves demonstrate the advantage of SIMD instructions with two showing an improvement of 1.61 and 1.65 times. However, the SE(3) Logarithm is slower so there should be future work into the selection of the optimisation strategy employed and the instructions used.

8.3 Future Work

This thesis provides approaches to implement specialisation that opposes the trend to evolve existing abstractions by unnecessarily extending their APIs. This
allows communication of semantics of an application to a lower level, leading to potential optimisations without explicit inclusion by developers. The approaches can be extended to provide additional support or different compromises during compilation.

Within the research into SLAM applications the JVM has been modified to specialise for small vectors. This has been achieved with improved optimisation phases targeting the nature of the arithmetic used and its suitability for acceleration using SIMD instructions. This is the beginning of a domain specific language and is diverging from the original aims of achieving this within a managed runtime environment.

The aim of the future work is to specialise a library, exploiting the metacircular nature of Graal. This will allow Java to remain intact but allow additional semantics, that may violate the specifications of the virtual machine [10], to augment runtime behaviour. These semantics will be encapsulated within the library and exploited by directly interacting with the dynamic compiler. An example would be to propagate constant zeroes, found in small vectors and matrices in SLAM, through constant folding ignoring the specification for NaN. This may be exploited as SLAM applications need to check for these as they are often used to indicate a point is outside the current view. The checks are used to avoid array index out of bound errors, thus, they should not appear in arithmetic and can be considered undefined behaviour.

By removing the specialisation from the JVM and encapsulating it in a library, it may be possible to improve performance using techniques that would be difficult to generalise. The options include improved selection of instructions, types and to increase the range of architectures supported.

8.3.1 SIMD Instruction Selection

The latency of different instruction sequences is critical in producing improved executions time. A general approach to instruction selection is provided by using pattern matching in graphs [160]. However, matrix and vector data may be more aggressively transformed because of the immutability used in the class collection presented in this thesis. Specialising an SIMD instruction selection for vector arithmetic in the context of SLAM applications would be beneficial. A similar approach was implemented for stream-based algorithms in C [162]. It would also provide the basis for techniques that could be applied more generally.
8.3.2 Additional SIMD Unit Support

The size of vectors used in Chapter 7 are 128-bits wide. Desktop computers based on the Intel Haswell architecture and above support 256-bits and the Xeon Phi support 512-bits. A benefit to dynamic compilation is the awareness of the architecture targeted so there is an opportunity to see how SIMD units computing can accelerate Java. The wider registers would also allow larger intermediate objects to be kept in registers with the potential for more execution time improvements.

Another development is the introduction of the Hexagon Vector Extensions (HVX) for the Qualcomm Snapdragon DSP with 1024-bit wide registers [163]. This further extends the challenges of heterogeneity in modern processors but opens up additional opportunities for optimisation. It does not support floating point arithmetic but it does support some fixed point operations. SLAM is based on estimation and there is little evidence that accuracy of the data types used is considered. An abstraction for primitive types, similar to what is available in Ada, may permit the use of fixed point arithmetic unlocking this type of devices for acceleration.

8.3.3 Flexible Accuracy

In control theory, there is a problem that the faster something is sampled the more accurately it needs to be measured. This is because as the sampling rate increases the changes between samples become smaller. The aim in demonstrating performance by taking an algorithm and hoping that it will just speed up indefinitely is naïve. There must be a consideration as to what accuracy is required, what numeric representations are available and how good performance needs to be.

Another point of note is that LSD-SLAM uses numeric methods to reduce the effect of anomalies in data. This stems from image problems, include changing light levels, glare and occlusion. There is little evidence in the implementation of LSD-SLAM of the quantisation noise (0.4% for pixels) affecting the thresholds used. The numerical methods are then applied with double precision floating point numbers. The Java implementation of uses single precision values but without any noticeable difference, what impact is there in propagating this uncertainty through a system to bound calculation. It may possible to use accuracy
constraints to change the representation of data in SIMD units, e.g. using half-precision floating point numbers, to increase the throughput of data in vector arithmetic.
Bibliography

   http://www.apache.org/.  
   Online; last accessed 27-July-2015.

   http://openjdk.java.net/projects/jdk8/.  
   Online; last accessed 24-May-2015.

   Instrumentation. In Proceedings of the 5th International Symposium on  

   http://openjdk.java.net/projects/graal/.  
   Online; last accessed 24-May-2015.

   Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative  
   Programming Languages and Compilers Workshop, 2013.

[6] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The  

   Online; last accessed 24-February-2015.


[18] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. KinectFusion: Real-Time Dense Surface


[28] Intel, Inc. Intel Website.
   http://www.intel.co.uk/.
   Online; last accessed 22-July-2013.


   http://www.hypertransport.org/.
   Online; last accessed 08-July-2013.


[33] EUROSERVER. EUROSERVER Project.
   Online; last accessed 06-May-2015.


   https://software.intel.com/sites/landingpage/
   IntrinsicsGuide/.
   Online; last accessed 31-August-2015.

   Online; last accessed 06-May-2015.

[37] NVIDIA Corporation. CUDA Parallel Computing Website.
html.  
Online; last accessed 23-July-2013.

[38] Khronos Group. OpenCL Website.  
http://www.khronos.org/opencl/.  
Online; last accessed 23-July-2013.

Online; last accessed 06-August-2013.


[41] ARM Limited. big.LITTLE Website.  
Online; last accessed 08-July-2013.

[42] The IEEE and The Open Group. POSIX Website.  
http://pubs.opengroup.org/onlinepubs/9699919799/.  
Online; last accessed 23-September-2015.


Online; last accessed 23-July-2013.


[61] Volker Markl. Big Data Looks Tiny From the Stratosphere. 


[63] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-Memory System. In


    http://www.eclemma.org/jacoco/.  
    Online; last accessed 11-May-2015.

[83] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with  
    Java STM. In *Proceeding of the Third Workshop on Programmability Issues  

[84] OW2 Consortium. ASM Project Website.  
    http://asm.ow2.org/.  
    Online; last accessed 11-May-2015.

    Project Website.  
    https://commons.apache.org/proper/commons-bcel/.  
    Online; last accessed 11-May-2015.

[86] Eugene Kuleshov. Using the ASM framework to implement common Java  
    bytecode transformation patterns. *Aspect-Oriented Software Development*,  
    2007.

[87] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hod-  
    son, and Antony Rowstron. Nobody ever got fired for buying a cluster.  
    2013.

    Online; last accessed 28-July-2015.

[89] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The  
    PARSEC Benchmark Suite: Characterization and Architectural Implica-  
    tions. In *Proceedings of the 17th International Conference on Parallel Ar-  

[90] 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. Introducing SLAM-  
    Bench, a performance and accuracy benchmarking methodology for SLAM.


Online; last accessed 09-July-2015.


Online; last accessed 15-September-2015.

Online; last accessed 31-July-2015.

http://uk.mathworks.com/products/computer-vision/?s_tid=hp_fp_list.  
Online; last accessed 31-July-2015.

[121] James Clarkson, Christos Kotselidis, Gavin Brown, and Mikel Luján.  

[122] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam,  
and Vijay Sundaresan. Soot – a Java Bytecode Optimization Framework.  
In Proceedings of the 1999 Conference of the Centre for Advanced Studies  

[123] Joe Hicklin, Cleve Moler, Peter Webband, Ronald F. Boisvert, Bruce Miller,  
http://math.nist.gov/javanumerics/jama/.  
Online; last accessed 27-April-2015.

http://dst.lbl.gov/ACSSoftware/colt/.  
Online; last accessed 27-April-2015.

[125] Piotr Wendykier and James G. Nagy. Parallel Colt: A High-Performance  
Java Library for Scientific Computing and Image Processing. ACM Trans-  

[126] Jack J. Dongarra. The LINPACK Benchmark: An explanation. In Super-  
computing, volume 297 of Lecture Notes in Computer Science, pages  

https://www.google.com/atap/project-tango/.  
Online; last accessed 29-July-2015.

[129] ARM Limited. NEON Website.  
Online; last accessed 29-July-2015.

[130] Qualcomm Technologies, Inc. MARE Parallel Computing SDK Website.  
Online; last accessed 29-July-2015.

[131] Itseez, Inc. OpenCV for Java Website.  
Online; last accessed 31-August-2015.

https://github.com/bytedeco/javacv.  
Online; last accessed 31-August-2015.

[133] Hugh Perkins. JEigen Website.  
https://github.com/hughperkins/jeigen.  
Online; last accessed 06-August-2015.

[134] Peter Abeles. BoofCV Project Website.  
Online; last accessed 27-April-2015.

[135] Oracle, Inc. JNI APIs and Developer Guide Website.  
http://docs.oracle.com/javase/8/docs/technotes/guides/jni/.  
Online; last accessed 31-August-2015.

Online; last accessed 06-August-2015.


[146] The University of Tennessee. PLASMA Project Website. 
http://icl.cs.utk.edu/plasma/. 
Online; last accessed 10-August-2015.

Online; last accessed 06-August-2015.

[148] Ronald F. Boisvert, José Moreira, Michael Philippsen, and Roldan Pozo. 
Java and Numerical Computing. 
Online; last accessed 27-April-2015.

[149] Mikel Luján, T. L. Freeman, and John R. Gurd. OoLALA: an Object 
Oriented Analysis and Design of Numerical Linear Algebra. In *Proceedings 
of the 15th ACM SIGPLAN conference on Object-Oriented Programming, 

[150] Peter Abeles. Geometric Regression Library Project Website. 
Online; last accessed 27-April-2015.

https://github.com/lessthanoptimal/ejml. 
Online; last accessed 07-August-2015.

[152] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Mak- 
ing the Future Safe for the Past: Adding Genericity to the Java Program- 

[153] Maurizio Cimadamore and Mirko Viroli. On the reification of Java wild- 

[154] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan 


Appendix A

MapReduce Word Count

This appendix contains the source code for the word count benchmark used throughout Chapter 3. The main method and ceremony surrounding the code has been omitted for brevity. Each contains the same algorithm based on the Phoenix 2.0 benchmarks [63]; allowing a direct comparison of the frameworks.

A.1 Phoenix 2.0

```c
typedef struct {
    int offset;
    int length;
    char *input_data;
    int unit_size;
} wc_data_t;

int key_cmp(const void *key1_, const void *key2_) {
    return strcmp((const char *) key1_, (const char *) key2_);
}

int splitter(void *data_in_, int units_, map_args_t *out_) {
    wc_data_t *data = (wc_data_t *) data_in_;
    if (data->offset >= data->length) {
        return 0;
    }
    out_->data = (void *)&data->input_data[data->offset];
    out_->length = units_ * data->unit_size;
    if ((data->offset + out_->length) > data->length) {
        out_->length = data->length - data->offset;
    }
    data->offset += out_->length;
    while (data->offset < data->length && data->input_data[data->offset] != ' ') {
        data->offset++;
        out_->length++;
    }
} 
```
APPENDIX A. MAPREDUCE WORD COUNT

```c
027     return 1;
028 }
029
030 void *locator(map_args_t *task_) {  
031     return task_->data;
032 }
033
034 void mapper(map_args_t *args_) {  
035     char *data = (char *)args_->data;
036     int length = args_->length;
037     int start, i;
038     for (i = 0; i < length; i++) {  
039         data[i] = (char) toupper(data[i]);
040     }
041     i = 0;
042     while (i < length) {  
043         while (i < length && (data[i] < 'A' || data[i] > 'Z')) {  
044             i++;
045         }
046         start = i;
047         while (i < length && ((data[i] >= 'A' && data[i] <= 'Z') || data[i] == '\n')) {  
048             i++;
049         }
050         if (i > start) {  
051             data[i] = '\0';
052             emit_intermediate(&data[start], (void *)1, i - start);
053         }
054     }
055 }
056
057 void reducer(void *key_, iterator_t *itr_) {  
058     char *key = (char *)key_;  
059     void *val;  
060     intptr_t sum = 0;  
061     while (iter_next(itr_, &val)) {  
062         sum += (intptr_t) val;
063     }
064     emit(key, (void *) sum);
065 }
066
067 void *combiner(iterator_t *itr_) {  
068     void *val;  
069     intptr_t sum = 0;  
070     while (iter_next(itr_, &val)) {  
071         sum += (intptr_t) val;
072     }
073     return (void *) sum;
074 }
075
076 void run(char *input_data_, int input_length_) {  
077     final_data_t wc_vals;
078     wc_data_t wc_data;
079     wc_data.unit_size = 5; // approx 5 bytes per word
```
APPENDIX A. MAPREDUCE WORD COUNT

081    wc_data.offset = 0;
082    wc_data.length = input_length;
083    wc_data.input_data = input_data;
084
085    map_reduce_args_t map_reduce_args;
086    memset(&map_reduce_args, 0, sizeof(map_reduce_args_t));
087    map_reduce_args.task_data = &wc_data;
088    map_reduce_args.map = mapper;
089    map_reduce_args.reduce = reducer;
090    map_reduce_args.combiner = combiner;
091    map_reduce_args.splitter = splitter;
092    map_reduce_args.locator = locator;
093    map_reduce_args.key_cmp = key_cmp;
094    map_reduce_args.unit_size = wc_data.unit_size;
095    map_reduce_args.partition = NULL;
096    map_reduce_args.result = &wc_vals;
097    map_reduce_args.data_size = input_length;
098    map_reduce_args.L1_cache_size = atoi(GETENV("MR_L1CACHESIZE"));
099    map_reduce_args.num_map_threads = atoi(GETENV("MR_NUMPROCS"));
100    map_reduce_args.num_reduce_threads = atoi(GETENV("MR_NUMPROCS"));
101    map_reduce_args.num_merge_threads = atoi(GETENV("MR_NUMPROCS"));
102    map_reduce_args.num_procs = atoi(GETENV("MR_NUMPROCS"));
103    map_reduce_args.key_match_factor = 2;
104
105    map_reduce_init();
106
107    map_reduce(&map_reduce_args);
108
109    map_reduce_finalize();
110 }

A.2 Phoenix++

001 struct wc_string {
002     char* data;
003     uint64_t len;
004 };  
005
006 struct wc_word {
007     char* data;
008
009     bool operator<(wc_word const& other) const {
010         return strcmp(data, other.data) < 0;
011     }
012
013     bool operator==(wc_word const& other) const {
014         return strcmp(data, other.data) == 0;
015     }
016  
017  }
018
019  

size_t operator()(wc_word const& key) const {
  char* h = key.data;
  uint64_t v = 14695981039346656037ULL;
  while (*h != 0)
    v = (v ^ (size_t) (*(h++))) * 1099511628211ULL;
  return v;
}
};

class WordCount : public MapReduce<WordCount,
    wc_string, wc_word, uint64_t,
    hash_container<wc_word, uint64_t, sum_combiner, wc_word_hash> > {

class WordCount : public MapReduce<WordCount,
    wc_string, wc_word, uint64_t,
    hash_container<wc_word, uint64_t, sum_combiner, wc_word_hash> > {
  char* data;
  uint64_t data_size;
  uint64_t chunk_size;
  uint64_t splitter_pos;
  public:
    explicit WordCount(char* _data, uint64_t length, uint64_t _chunk_size) :
      data(_data), data_size(length), chunk_size(_chunk_size), splitter_pos(0) {
    }
    void* locate(data_type* str, uint64_t len) const {
      return str->data;
    }
    void map(data_type const& s, map_container& out) const {
      for (uint64_t i = 0; i < s.len; i++) {
        s.data[i] = toupper(s.data[i]);
      }
      uint64_t i = 0;
      while (i < s.len) {
        while (i < s.len && (s.data[i] < 'A' || s.data[i] > 'Z')) {
          i++;
        }
        uint64_t start = i;
        while (i < s.len && (s.data[i] >= 'A' && s.data[i] <= 'Z')
            || s.data[i] == '\\') {
          i++;
        }
        if (i > start) {
          s.data[i] = 0;
          wc_word word = { s.data+start }; w
          emit_intermediate(out, word, 1);
        }
      }
      int split(wc_string& out) {
        if ((uint64_t) splitter_pos >= data_size) {
          return 0;
        }
        uint64_t end = std::min(splitter_pos + chunk_size, data_size);
while (end < data_size && data[end] <= ' ') {
    end++;
}
out.data = data + splitter_pos;
out.len = end - splitter_pos;
splitter_pos = end;
return 1;
}
};
void run(char *input_data_, int input_length_) {
std::vector<WordCount::keyval> result;
WordCount mapReduce(input_data_, input_length_, 1024 * 1024);
mapReduce.run(result) < 0);
}

A.3 MR4J

public class WordCount {
private final MapReduce<String, String, Integer> mrj;
private final Integer one = 1;
public WordCount() {
    mrj = new MapReduce<>(mapper, reducer);
}
private final Mapper<String, String, Integer> mapper = new Mapper<String, String, Integer>() {
    @Override
    public void map(String input, Emitter<String, Integer> emitter) {
        String data = input.toUpperCase();
        int i = 0, start;
        int length = data.length();
        while (i < length) {
            while (i < length && (data.charAt(i) < 'A' || data.charAt(i) > 'Z')) {
                i++;
            }
            start = i;
            while (i < length && (data.charAt(i) >= 'A' && data.charAt(i) <= 'Z') || data.charAt(i) == '\n')) {
                i++;
            }
            if (i > start) {
                emitter.emit(data.substring(start, i), one);
            }
        }
    }
};

private final Reducer<String, Integer> reducer = new Reducer<String, Integer>() {
    @Override
    public void reduce(String key,
            List<Integer> values,
            Emitter<String, Integer> emitter) {
        int sum = 0;
        for (Integer value : values) {
            sum += value;
        }
        emitter.emit(key, sum);
    }
};

public List<Entry<String, Integer>> run(List<String> input, int parallelism) {
    return mrj.run(input, parallelism)
}
Appendix B

MR4J Collector Selection

The intermediate (key, value) pair collection is essential to the speed and scalability of the MapReduce frameworks for multi-core architectures. Phoenix 2.0 explored the design space and tuning parameter to improve this feature [63]. The advantage of shared-memory architectures with a global address space is that any task may access any data, increasing the potential for optimisation over distributed parallel frameworks.

A key design consideration is the type of keys emitted during the map phase. Google began with strings because much of the data and intermediate storage format is textual [52]. Hadoop leveraged the programmability of Java and introduced flexible typing to minimise the type conversion required in applications [11]. Phoenix started life as an implementation in C using void pointers to introduce flexibility [62, 63]. This is detrimental to the programmability and increases the likelihood of runtime errors. However it does allow the casting of primitive types or pointers to be emitted without any additional memory allocation. This feature drastically increased performance but the user must still provide a method to compare keys for the collector to work. Phoenix++ makes full use of template classes and can specify primitive types as the key type achieving the same but with programming errors caught at compile time [64]. As with Phoenix the user must provide code to hash a key (unless it is used as an index to an array container).

MR4J must allow the use of flexible keys and as demonstrated in Phoenix and Phoenix++ a hash table is the sensible choice. It enables any object type to be accessed with constant time access (average). What is more, the base class in Java, the Object, implements a hash method so all classes may be used in a Map,
good programming dictates the `equals` and `hashCode` methods are overridden for new classes and exist for standard classes such as `String`. The motivation behind this evaluation is to investigate which Java hash table supports parallelism as the best implementation within MR4J.

### B.1 Alternative Implementations

There are two main options for the hash table in MR4J both implement the `ConcurrentMap` interface so may be used interchangeably. The implementation of MR4J is also encapsulated so the selection may be updated in the future should better implementations emerge. A synchronised hash map, created in the `Collections` class, is not used as it does not support the `ConcurrentMap` interface and has been superseded.

#### B.1.1 ConcurrentHashMap

Doug Lea has been instrumental in developing the concurrency library in the JDK, this includes a concurrent hash table. To achieve thread-safe behaviour the table uses a series of synchronized segments to hold the (key, value). Within these is either a traditional array of linked lists or, if large enough, a tree to support Streams that were introduced for Java SE 1.8. This class is available with the standard Oracle Java distribution [7].

#### B.1.2 NonBlockingHashMap

Cliff Click noted that the concurrent hash table implementation in Java uses synchronization for thread-safety. This has the potential to limit the scalability if accessed by many threads and went about implementing a version that uses purely atomic operations [78]. The concept behind the design is a finite state machine that uses atomic compare and set instructions to transition between states. It contains two linear probing hash tables that form a buffer for (key, value) pairs when resizing, pairs are copied to the active table during subsequent accesses.
APPENDIX B. MR4J COLLECTOR SELECTION

<table>
<thead>
<tr>
<th></th>
<th>Number of Keys</th>
<th>Number of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiny</td>
<td>10</td>
<td>1,000</td>
</tr>
<tr>
<td>Small</td>
<td>1,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>Medium</td>
<td>1,000</td>
<td>1,000,000,000</td>
</tr>
<tr>
<td>Large</td>
<td>100,000</td>
<td>1,000,000,000</td>
</tr>
</tbody>
</table>

Table B.1: MR4J collector evaluation input sizes.

B.2 Performance Evaluation

The performance evaluation is a series of micro-benchmarks to ascertain the execution overhead and the scalability of the two hash tables. The micro-benchmarks cover the main key types used in the Phoenix benchmarks [63].

B.2.1 Experimental Configuration

The hardware and software configuration used for the evaluation are the same as the workstation and server configuration used for MR4J (Section 3.5).

B.2.2 Micro-Benchmarks

There are three micro-benchmarks used in this evaluation, each with four input sizes. They aim to emulate the use cases for the benchmarks used to compare Phoenix and Phoenix++ to their predecessors. All three configuration use an integer type as the value that is incremented by one for each associated key that is generated. The difference is in the key type: a string (String) used to demonstrate data types that cannot be used as an index; and an integer that is randomly (Random) and sequentially (Sequential) generated to understand behaviour equivalent to arrays in Phoenix++. Table B.1 contains the four input size configurations used.

To evaluate the run time each micro-benchmark has a warm-up period to encourage dynamic compilation in the JVM. To arrive at the final execution time the average of 10 runs for all frameworks, micro-benchmarks and input sizes. The number of hardware threads is also varied so the scalability may also be explored.
B.2.3 Results

Figure B.1 and Figure B.2 contain the micro-benchmark results for the workstation and server configuration. They contain the relative performance for the three data types of the NonBlockingHashMap using the ConcurrentHashMap as the reference.

B.3 Conclusions

The results from the micro-benchmark indicate the NonBlockingHashMap is slower and no better in scalability than the standard ConcurrentHashMap. Due to these findings MR4J uses the latter to implement intermediate (key, value) pair collector. This selection also has the benefit that it is distributed as part of the Java runtime environment, promoting the design principle of re-use. The choice is also beneficial as the hash table is maintained and developed by a community so any future improvements will be incorporated without extra effort for MR4J. Finally as it could be possible to introduce MR4J as a standard class in the JDK.
APPENDIX B. MR4J COLLECTOR SELECTION

Figure B.2: Relative performance of NonBlockingHashMap on server.

the re-use of existing classes reduces the duplication of code minimising the scope for error and effort required for maintenance.
Appendix C

SLAM Kernels

This appendix contains the Java implementation of the three SLAM kernels used to evaluate the performance of co-design. They are based on the implementation their uses in LSD-SLAM [17].

C.1 Point Transform

The current pose and camera are used to project points in the point cloud onto pixels in the image. This is achieved by transforming the point, from the pose rotation and translation, and then projecting it using a pinhole camera model. Figure C.1 contains the Java code used in the performance evaluations. Between these two transforms the depth ($z()$) of the point is normalised so it may be multiplied by the intrinsic parameter matrix ($k$) of the camera.

```java
01 private final Matrix3D k; // intrinsic parameter matrix
...
11 public Point3D test(Rotation3D rotation,
12     Vector3D translation,
13     Point3D point) {
14     Point3D wxp = rotation.multiply(point).add(translation);
15     return k.multiply(wxp.divide(wxp.z()));
16 }
...
```

Figure C.1: Java code for the Point Transform SLAM kernel.
C.2 SE(3) Logarithm

Poses are represented as a Lie group that may be transformed using exponential mapping [94]. The code in Figure C.2 encapsulates the mapping to a vector containing six elements and is used in LSD-SLAM to estimate a pose. The methods it invokes from the S03Group class can be found in Figure C.3. It contains methods from two classes (S03Group and S03Group) and is based on the implementation from Sophus [116].

01 public final class SE3Group {
   ...
   11 public static Tangent6D log(SE3Group se3) {
       Pair<Tangent3D, Float> log
           = S03Group.logWithTheta(se3.quaternion());
       Tangent3D omega = log.getFirst();
       float theta = log.getSecond();
       Rotation3D hat = S03Group.hat(omega);
       float constant;
       if (Math.abs(theta) < 1e-5f) {
           constant = 1.0f / 12.0f;
       } else {
           constant = (1.0f - theta
               / (2.0f * (float) Math.tan(theta / 2.0f)))
               / (theta * theta);
       }
       Rotation3D v = Rotation3D.identity()
           .subtract(hat.multiply(0.5f))
           .add(hat.multiply(hat).multiply(constant));
       Vector3D upsilon = v.multiply(se3.translation());
       return new Tangent6D(upsilon, omega);
   } ...

Figure C.2: Java code for the SE(3) Logarithm SLAM kernel.
public final class SO3Group {
    ...
    public static Rotation3D hat(Tangent3D tangent) {
        return new Rotation3D(
            0.0f, -tangent.z(), tangent.y(),
            tangent.z(), 0.0f, -tangent.x(),
            -tangent.y(), tangent.x(), 0.0f);
    }
    ...
    static Pair<Tangent3D, Float> logWithTheta(Quaternion quaternion) {
        float norm = quaternion.imaginary().norm();
        float real = quaternion.real();
        float scale;
        if (norm < 1e-5f) {
            scale = 2.0f / real
                     - 2.0f * (norm * norm)
                     / (real * real * real);
        } else {
            if (Math.abs(real) < 1e-5f) {
                scale = (float) Math.PI / norm;
                if (real < 0) {
                    scale = -scale;
                }
            } else {
                scale = 2.0f * (float) Math.atan(norm / real)
                           / norm;
            }
        }
        Vector3D temp = quaternion.imaginary().multiply(scale);
        return new Pair<>(
            new Tangent3D(temp.x(), temp.y(), temp.z()),
            scale * norm);
    }
    ...

Figure C.3: Java code supporting the SE(3) Logarithm SLAM kernel.
C.3 Gradient Interpolation

The gradient is interpolated in LSD-SLAM to obtain sub-pixel resolution, improving the accuracy of the algorithms. The fractions of the indices (dx and dy) are used to weight the kernel for the convolution of pixel gradients in the image (t00–t11). Figure C.4 contains the Java code used in the performance evaluations.

```java
public Vector3D test(float dx, float dy, Vector3D t00, Vector3D t01, Vector3D t10, Vector3D t11) {
    float dxdy = dx * dy;
    return t11.multiply(dxdy).add(t10.multiply(dy - dxdy)).add(t01.multiply(dx - dxdy)).add(t00.multiply(1.0f - dx - dy + dxdy));
}
```

Figure C.4: Java code for the Gradient Interpolation SLAM kernel.
C.4 Levenberg-Marquardt Update

The poses in LSD-SLAM use the Levenberg-Marquardt algorithm to best fit the estimated pose between frames. The system is created by accumulating a matrix constructed of the pose vector (the Jacobian matrix) multiplied by its transpose and the vector itself. The matrix diagonal is scaled and the system is later solved by using matrix decomposition but these do not form part of the micro-benchmark as they are rarely executed. Figure C.5 contains the Java code used to test this SLAM kernel.

```java
01 public Pair<IMatrix, Tangent6D> test(
            Pair<IMatrix, Tangent6D> system,
            Tangent6D pose,
            float residual,
            float weight) {
02            IMatrix m = AnonMatrix.create(pose);
03
04            IMatrix matrix = system.first().add(                  
            m.multiply(m.transpose()).multiply(weight));        
05            Tangent6D vector = system.second().subtract(
            pose.multiply(residual * weight));
06
07            return new Pair<matrix, vector> system;
08        }
...```

Figure C.5: Java code for the Levenberg-Marquardt Update SLAM kernel.
Appendix D

Vector Operation Nodes

There are seven macro-substitutions implemented to exploit SIMD instructions for the nodes added to Graal IR to support the co-design presented in this thesis. Table D.1 contains the latencies of the instruction sequences, with an approximate speed-up, for each of the operations. The overhead of memory access has been ignored and the table shows the approximate number of clock cycles simplified by assuming a single instruction at a time.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original</td>
<td>SIMD</td>
</tr>
<tr>
<td>Add</td>
<td>36</td>
<td>12</td>
</tr>
<tr>
<td>Cross Product</td>
<td>68</td>
<td>26</td>
</tr>
<tr>
<td>Divide</td>
<td>79</td>
<td>23</td>
</tr>
<tr>
<td>Dot Product</td>
<td>44</td>
<td>23</td>
</tr>
<tr>
<td>Hamilton Product</td>
<td>176</td>
<td>51</td>
</tr>
<tr>
<td>Multiply</td>
<td>47</td>
<td>15</td>
</tr>
<tr>
<td>Subtract</td>
<td>36</td>
<td>12</td>
</tr>
</tbody>
</table>

Table D.1: Accumulated latency of vector operation using SIMD.
D.1 Add

a) original code generated

\begin{verbatim}
movss  xmm0, &lhs.v0  ; scalar load
addss  xmm0, &rhs.v0  ; scalar add
movss  &result.v0, xmm0  ; result.v0 = lhs.v0 + rhs.v0
movss  xmm0, &lhs.v1  ;
addss  xmm0, &rhs.v1  ;
movss  &result.v1, xmm0  ; result.v1 = lhs.v1 + rhs.v1
movss  xmm0, &lhs.v2  ;
addss  xmm0, &rhs.v2  ;
movss  &result.v2, xmm0  ; result.v0 = lhs.v2 + rhs.v2
movss  xmm0, &lhs.v3  ;
addss  xmm0, &rhs.v3  ;
movss  &result.v3, xmm0  ; result.v0 = lhs.v3 + rhs.v3
\end{verbatim}

b) code after SIMD aware node substitution

\begin{verbatim}
movups  xmm0, &lhs  ; vector load (unaligned)
movups  xmm1, &rhs  ;
addps  xmm0, xmm1  ; vector add
movups  &result, xmm0  ; result = lhs + rhs
\end{verbatim}

Figure D.1: Original and SIMD code generated for vector add.
D.2 Cross Product

a) original code generated

```assembly
movss  xmm0, &lhs.v1 ; lhs.y (scalar load)
movss  xmm1, &rhs.v1 ; rhs.y
movss  xmm2, &rhs.v0 ; rhs.x
movss  xmm3, &lhs.v2 ; lhs.z
movss  xmm4, &lhs.v0 ; lhs.x
movss  xmm5, &rhs.v2 ; rhs.z
movaps xmm6, xmm1 ; take a copy of rhs.y
movaps xmm7, xmm2 ; take a copy of rhs.x
mulss xmm6, xmm4 ; rhs.y * lhs.x
mulss xmm7, xmm0 ; rhs.x * lhs.y
subss xmm6, xmm7 ; z = rhs.y * lhs.x - rhs.x * lhs.y
mulss xmm0, xmm5 ; lhs.y * rhs.z
mulss xmm1, xmm3 ; rhs.y * lhs.z
subss xmm0, xmm1 ; x = lhs.y * rhs.z - rhs.y * lhs.z
mulss xmm2, xmm3 ; rhs.x * lhs.z
mulss xmm4, xmm5 ; lhs.x * rhs.z
subss xmm2, xmm4 ; y = rhs.x * lhs.z - lhs.x * rhs.z
movss &result.v0, xmm0 ; result.v0 = x
movss &result.v1, xmm2 ; result.v1 = y
movss &result.v2, xmm6 ; result.v2 = z
```

b) code after SIMD aware node substitution

```assembly
movups xmm0, &lhs ; vector load (unaligned)
movups xmm1, &rhs ;
pshufd xmm2, xmm0, 0xC5 ; a = [ lhs.y lhs.z lhs.x 0.0 ]
pshufd xmm3, xmm1, 0xD2 ; b = [ rhs.z rhs.x rhs.y 0.0 ]
mulps xmm2, xmm3 ;
pshufd xmm0, xmm0, 0xD2 ; d = [ lhs.z lhs.x lhs.y 0.0 ]
pshufd xmm1, xmm1, 0xC5 ; e = [ rhs.y rhs.z rhs.x 0.0 ]
mulps xmm0, xmm1 ;
subps xmm2, xmm0 ;
movups &result, xmm2 ; result = a * b - d * e
```

Figure D.2: Original and SIMD code generated for vector cross product.
D.3 Divide

a) original code generated

```
movss  xmm0, &lhs.v0    ; scalar load
movss  xmm1, &a         ;
divss  xmm0, xmm1       ; scalar divide
movss  &result.v0, xmm0 ; result.v0 = lhs.v0 / a
movss  xmm0, &lhs.v1    ;
divss  xmm1, xmm1       ;
movss  &result.v1, xmm0 ; result.v1 = lhs.v1 / a
movss  xmm0, &lhs.v2    ;
divss  xmm1, xmm1       ;
movss  &result.v2, xmm0 ; result.v2 = lhs.v2 / a
movss  xmm0, &lhs.v3    ;
divss  xmm1, xmm1       ;
movss  &result.v3, xmm0 ; result.v3 = lhs.v3 / a
```

b) code after SIMD aware node substitution

```
movups  xmm0, &lhs         ; vector load (unaligned)
movss  xmm1, &a            ; scalar load
pshufd  xmm1, xmm1, 0x00    ; b = [ a a a a ]
divps  xmm0, xmm1          ; vector divide
movups  &result, xmm0       ; result = lhs / b
```

Figure D.3: Original and SIMD code generated for vector divide.
APPENDIX D. VECTOR OPERATION NODES

D.4 Dot Product

a) original code generated

```
movss  xmm0, &lhs.v0    ; scalar load
mulss  xmm0, &rhs.v0   ; a = lhs.v0 * rhs.v0
movss  xmm1, &lhs.v1
mulss  xmm1, &rhs.v1   ; b = lhs.v1 * rhs.v1
addss  xmm0, xmm1     ; dot = a + b
movss  xmm1, &lhs.v2
mulss  xmm1, &rhs.v2   ; c = lhs.v2 * rhs.v2
addss  xmm0, xmm1     ; dot = a + b + c
movss  xmm1, &lhs.v3
mulss  xmm1, &rhs.v3   ; d = lhs.v3 * rhs.v3
addss  xmm0, xmm1     ; dot = a + b + c + d
movss  &result, xmm0  ; result = dot
```

b) code after SIMD aware node substitution

```
movups xmm0, &lhs    ; vector load (unaligned)
movups xmm1, &rhs

dpps  xmm0, xmm1     ; dot product
movss  &result, xmm0 ; result = lhs . rhs
```

Figure D.4: Original and SIMD code generated for vector dot product.
APPENDIX D. VECTOR OPERATION NODES

D.5 Hamilton Product

a) original code generated

```assembly
movss  xmm0, &lhs.v0  ; lhs.w (scalar load)
mulss xmm0, &rhs.v0 ; a = lhs.real * rhs.real
movss xmm1, &lhs.v1 
mulss xmm1, &rhs.v1 ; b = lhs.i * rhs.i
subss xmm0, xmm1 
mulss xmm1, &lhs.v2 
mulss xmm1, &rhs.v2 ; c = lhs.j * rhs.j
subss xmm0, xmm1 
movss xmm0, &lhs.v0 ; a = lhs.real * rhs.i
mulss xmm0, &rhs.v1 ; a = lhs.real * rhs.i
movss xmm1, &lhs.v1 
mulss xmm1, &rhs.v0 ; b = lhs.i * rhs.real
addss xmm0, xmm1 
mulss xmm1, &rhs.v3 ; c = lhs.j * rhs.k
addss xmm0, xmm1 
movss xmm0, &lhs.v2 
mulss xmm0, &rhs.v2 ; d = lhs.k * rhs.j
subss xmm0, xmm1 
movss &result.v0, xmm0 ; result.v0 = a - b - c - d
movss xmm0, &lhs.v0 
mulss xmm0, &rhs.v0 ; a = lhs.real * rhs.real
movss xmm1, &lhs.v1 
mulss xmm1, &rhs.v0 ; b = lhs.i * rhs.real
addss xmm0, xmm1 
mulss xmm1, &rhs.v3 ; c = lhs.j * rhs.k
addss xmm0, xmm1 
movss xmm0, xmm1 
mulss xmm0, &rhs.v2 ; d = lhs.k * rhs.j
subss xmm0, xmm1 
movss &result.v1, xmm0 ; result.v0 = a + b + c - d
movss xmm0, &lhs.v0 
mulss xmm0, &rhs.v2 ; a = lhs.real * rhs.j
movss xmm1, &lhs.v1 
mulss xmm1, &rhs.v1 ; b = lhs.i * rhs.k
mulss xmm1, &rhs.v3 ; c = lhs.j * rhs.real
addss xmm0, xmm1 
movss xmm0, &lhs.v3 
mulss xmm1, &rhs.v1 ; d = lhs.k * rhs.i
addss xmm0, xmm1 
movss &result.v2, xmm0 ; result.v2 = a - b + c + d ...
```
APPENDIX D. VECTOR OPERATION NODES

movss xmm0, &lhs.v0 ;
mulss xmm0, &rhs.v3 ; \( a = \text{lhs}.real \times \text{rhs}.k \)
movss xmm1, &lhs.v1 ;
mulss xmm1, &rhs.v2 ; \( b = \text{lhs}.i \times \text{rhs}.j \)
addss xmm0, xmm1 ;
movss xmm1, &lhs.v2 ;
mulss xmm1, &rhs.v1 ; \( c = \text{lhs}.j \times \text{rhs}.i \)
subss xmm0, xmm1 ;
movss xmm1, &rhs.v3 ;
mulss xmm1, &lhs.v3 ; \( d = \text{lhs}.k \times \text{rhs}.real \)
addss xmm0, xmm1 ;
movss &result.v3, xmm0 ; \text{result.v2} = a - b - c + d

b) code after SIMD aware node substitution

movups xmm0, &lhs ; a (vector load (unaligned))
movups xmm1, &rhs ; b
mov edx, 0x80000000 ;
movd xmm2, edx ; mask to flip sign
pshufd xmm3, xmm0, 0x7B ; \[ \text{ak,
 aj,} \text{ak, ai} \]
pshufd xmm4, xmm1, 0x9F ; \[ \text{bk,}
 bk, \text{bi, bj} \]
mulps xmm3, xmm4 ;
xorps xmm3, xmm2 ; \( c = [ -\text{ak} \cdot \text{bk} \ \text{aj} \cdot \text{bk} \ \text{ak} \cdot \text{bi} \ \text{ai} \cdot \text{bj} ] \)
pshufd xmm4, xmm0, 0x02 ; \[ \text{aj, ar, ar, ar} \]
pshufd xmm5, xmm1, 0xE6 ; \[ \text{bj, bi, bj, bk} \]
mulps xmm4, xmm5 ;
xorps xmm4, xmm2 ; \( d = [ -\text{aj} \cdot \text{bj} \ \text{ar} \cdot \text{bi} \ \text{ar} \cdot \text{bj} \ \text{ar} \cdot \text{bk} ] \)
pshufd xmm2, xmm1, 0x00 ; \[ \text{br, br, br, br} \]
mulps xmm2, xmm0 ; \( e = [ \text{ar} \cdot \text{br} \ \text{ai} \cdot \text{br} \ \text{aj} \cdot \text{br} \ \text{ak} \cdot \text{br} ] \)
pshufd xmm0, xmm0, 0x9D ; \[ \text{ai, ak, ai, aj} \]
pshufd xmm1, xmm1, 0x79 ; \[ \text{bi, bj, bk, bi} \]
mulps xmm0, xmm1 ; \( f = [ \text{ai} \cdot \text{bi} \ \text{ak} \cdot \text{bj} \ \text{ai} \cdot \text{bk} \ \text{aj} \cdot \text{bi} ] \)
subps xmm2, xmm0 ;
addps xmm3, xmm4 ;
addps xmm2, xmm3 ;
movups &result, xmm2 ; \text{result} = e - f + c + d

Figure D.5: Original and SIMD code generated for Hamilton product.
APPENDIX D. VECTOR OPERATION NODES

D.6 Multiply

a) original code generated

```assembly
movss  xmm0, &lhs.v0              ; scalar load
movss  xmm1, &a                   
mulss  xmm0, xmm1                 ; scalar multiply
movss  &result.v0, xmm0           ; result.v0 = lhs.v0 * a
movss  xmm0, &lhs.v1              
mulss  xmm1, xmm1                 
movss  &result.v1, xmm0           ; result.v1 = lhs.v1 * a
movss  xmm0, &lhs.v2              
mulss  xmm0, xmm1                 
movss  &result.v2, xmm0           ; result.v2 = lhs.v2 * a
movss  xmm0, &lhs.v3              
mulss  xmm0, xmm1                 
movss  &result.v3, xmm0           ; result.v3 = lhs.v3 * a
```

b) code after SIMD aware node substitution

```assembly
movups xmm0, &lhs                  ; vector load (unaligned)
movss xmm1, &a                     ; scalar load
pshufd xmm1, xmm1, 0x00            ; b = [ a a a a ]
mulps xmm0, xmm1                   ; vector multiply
movups &result, xmm0               ; result = lhs * b
```

Figure D.6: Original and SIMD code generated for vector multiply.
APPENDIX D. VECTOR OPERATION NODES

D.7 Subtract

a) original code generated

movss xmm0, &lhs.v0 ; scalar load
subss xmm0, &rhs.v0 ; scalar subtract
movss &result.v0, xmm0 ; result.v0 = lhs.v0 - rhs.v0
movss xmm0, &lhs.v1 ;
subss xmm0, &rhs.v1 ;
movss &result.v1, xmm0 ; result.v1 = lhs.v1 - rhs.v1
movss xmm0, &lhs.v2 ;
subss xmm0, &rhs.v2 ;
movss &result.v2, xmm0 ; result.v0 = lhs.v2 - rhs.v2
movss xmm0, &lhs.v3 ;
subss xmm0, &rhs.v3 ;
movss &result.v3, xmm0 ; result.v0 = lhs.v3 - rhs.v3

b) code after SIMD aware node substitution

movups xmm0, &lhs ; vector load (unaligned)
movups xmm1, &rhs ;
subps xmm0, xmm1 ; vector subtract
movups &result, xmm0 ; result = lhs - rhs

Figure D.7: Original and SIMD code generated for vector subtract.