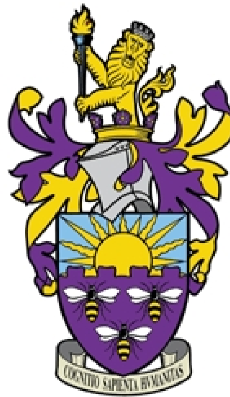
COMPILER AND RUNTIME SUPPORT FOR HETEROGENEOUS PROGRAMMING



A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2019

By
James Clarkson
School of Computer Science

Contents

Abstract **15**

Declaration **17**

Copyright **18**

Acknowledgements **19**

1 Introduction **20**

1.1 Challenges 22

1.1.1 Evolution of Devices 22

1.1.2 The Productivity Challenge 25

1.1.3 Evolving Use of Programming Languages 26

1.1.4 The Correctness Challenge 26

1.1.5 The Performance Challenge 26

1.2 Addressing These Challenges 27

1.3 Research Contributions 29

1.3.1 Publications 30

1.4 My Thesis 31

2 Background **34**

2.1 Processor Architectures 35

2.1.1 Single Instruction Single Data Stream 36

2.1.2 Multiple Instructions Multiple Data Streams 37

2.1.3 Single Instruction Multiple Data Streams 38

2.1.4 Single Instruction Multiple Threads 39

2.2 Programming Languages 40

2.2.1 Design Decisions 41

2.2.2 Assuming Closed and Open Worlds 44
2.3 The Software Gap 45
2.4 Programming Heterogeneous Systems 46
2.4.1 Low-level Heterogeneous Programming Languages 47
2.4.2 Programming SISD Architectures 50
2.4.3 Programming MIMD Architectures 50
2.4.4 Programming SIMD Architectures 51
2.4.5 Programming SIMT Architectures 54
2.4.6 High-level Heterogeneous Programming Languages 56
2.4.7 Emerging Heterogeneous Programming Languages 57
2.4.8 Examples of Prior At 57
2.4.9 Common Themes 59
2.5 Motivating Examples 60
2.5.1 Issues Due To Language Design 60
2.5.2 Inability To Express Coordination 60
2.6 Summary 63

3 Tornado API **65**
3.1 Programming Model 65
3.1.1 Tasks 67
3.1.2 Task Schedules 68
3.1.3 Informal Specification 68
3.1.4 Java Implementation 72
3.1.5 Composing Complex Pipelines 75
3.1.6 Design Rationale 83
3.2 Writing High-Performance Code 85
3.2.1 Parallelisation In Tornado 86
3.2.2 Collections API 87
3.3 Managing Uncertainty 87
3.3.1 Task Metadata 88
3.3.2 Dynamic Compilation 89
3.3.3 Task Tuning Parameters 89
3.3.4 Dynamic Configuration Examples 89
3.4 Summary 93

4 Tornado Virtual Machine **94**
4.1 A Nested VM 94
4.2 Overview 95
4.3 Architecture 97
4.3.1 Execution Context 99
4.3.2 Dynamic State 100
4.3.3 Internal State 100
4.3.4 Bytecode 101
4.3.5 Execution Engine 101
4.3.6 Device Interface 102
4.3.7 Object Cache 104
4.4 Bytecode Specification 104
4.4.1 Execution Model 106
4.5 Novel Features of The TVM 109
4.5.1 Virtualisation 109
4.5.2 Asynchronous Operations 110
4.5.3 Dynamic Configuration 113
4.6 Performance Model and Optimisation Criteria 115
4.6.1 Defining A Performance Model 115
4.6.2 Optimisations 118
4.7 Summary 119

5 Tornado Runtime System **121**
5.1 Architecture 122
5.1.1 Components For Compiling Coordination Logic 124
5.1.2 Tornado-IR Builder 124
5.1.3 Tornado-IR Optimiser 125
5.1.4 TVM Bytecode Assembler 126
5.1.5 Tornado Device-side Compilation Pipeline 126
5.1.6 Sketcher 126
5.1.7 Task Cache 128
5.1.8 TVM Client Compiler 128
5.2 Coordinating a Single Task 129
5.3 Coordinating Multiple Tasks 133
5.3.1 Variable Disambiguation 133
5.3.2 Handling Cross Device Data Flow 135

5.3.3 Strength Reduction . . . 135
5.3.4 Locality Disambiguation . . . 137
5.3.5 Redundancy Elimination . . . 138
5.3.6 Lowering . . . 139
5.3.7 Out-of-order Execution . . . 139
5.3.8 Scheduling . . . 142
5.4 Optimising Intra-task-schedule Data Movement . . . 146
5.4.1 Multi-device Task Schedules . . . 148
5.4.2 Object Caching Protocol . . . 149
5.5 Summary . . . 149

6 Tornado Virtual Machine Client 150
6.1 Overview . . . 152
6.2 Compiling Idiomatic Java . . . 158
6.2.1 Example . . . 159
6.2.2 Inlining . . . 160
6.2.3 Intrinsics . . . 160
6.2.4 Partial Escape Analysis And Allocation Removal . . . 161
6.2.5 Partial Evaluation . . . 163
6.3 Bytecode Coverage . . . 165
6.4 Aside: Jacc . . . 168
6.4.1 Jacc Architecture . . . 168
6.4.2 Shared Memory Atomics and Reductions . . . 169
6.4.3 Benchmarking Jacc . . . 170
6.4.4 Comparison with OpenMP and CUDA . . . 171
6.4.5 Comparison with APARAPI . . . 173
6.4.6 Why Is Tornado Based on OpenCL? . . . 174
6.5 Summary . . . 175

7 Evaluation 176
7.1 Limitations and Non-goals . . . 178
7.2 Experimental Setup . . . 180
7.2.1 System Configurations . . . 180
7.2.2 Tornado Software stack . . . 181
7.3 Real-world Application: Kinect Fusion . . . 181
7.3.1 Processing Pipeline . . . 184

7.3.2 Tracking Algorithm 186
7.3.3 Measuring Performance and Accuracy 189
7.4 Implementation 191
7.4.1 Serial Java 191
7.4.2 Tornado 194
7.5 Evaluating Portability 200
7.5.1 Aside: OpenCL Portability Bug 203
7.5.2 Aside: Issues Effecting Portability 205
7.6 Specialisation 207
7.6.1 Implementing the Reduction Kernel in OpenCL 210
7.6.2 Implementing the Reduction Kernel in Java 214
7.6.3 Performance Improvements 218
7.7 System Performance 222
7.7.1 Tornado Compiler Performance 224
7.7.2 Tornado Runtime System Performance 226
7.8 Dynamic Configuration and Optimization 228
7.8.1 Typical Optimization Process 229
7.8.2 Obtaining The Highest Kinect Fusion Performance 232
7.8.3 Blocking Versus Non-blocking OpenCL API Calls 234
7.8.4 Changing Parallelisation Schemes 236
7.9 Summary 238

8 Conclusion **240**
8.1 Tornado Design 240
8.2 Distinguishing Features of Tornado 241
8.3 Evaluation of Tornado 242
8.4 Limitations 244
8.4.1 Evaluation 244
8.4.2 Operating System and Hardware Diversity 244
8.4.3 Use of Multiple Accelerators 245
8.4.4 Limited Support For The Java Language 246
8.5 Final Remarks 246
8.5.1 Future Work 247
8.5.2 Extending Tornado 247
8.5.3 Development of a Programming Language 248

List of Tables

Table 1.1: Target System Configurations 30

Table 2.1: Existing Heterogeneous Programming Languages 61

Table 4.1: TVM Bytecode Specification 105
Table 4.2: Parameter Optimisation Rules 119

Table 5.1: Example Data-dependency Matrix 136

Table 6.1: Supported Java Bytecodes 167
Table 6.2: Jacc Benchmark Results 173

Table 7.1: System Configurations 181
Table 7.2: OpenCL Hardware Configuration 183
Table 7.3: Kinect Fusion: Pipeline Breakdown 186
Table 7.4: Kinect Fusion: Accuracy of Serial Implementations 190
Table 7.5: Kinect Fusion: Accuracy of Accelerated Implementations . . . 190
Table 7.6: Kinect Fusion: Serial C++ and Java Performance 192
Table 7.7: Kinect Fusion: Portability Results 202
Table 7.8: Kinect Fusion: Performance Results 219
Table 7.9: Tornado Kernel Profiling on the NVIDIA Tesla K20m 225

List of Figures

Figure 1.1: Example of a Homogeneous System 24
Figure 1.2: Example of a Heterogeneous System 24

Figure 2.1: Single Instruction Single Data (SISD) Architecture 36
Figure 2.2: Multiple Instruction Multiple Data (MIMD) Architecture 37
Figure 2.3: Single Instruction Multiple Data (SIMD) Architecture 38
Figure 2.4: Single Instruction Multiple Threads (SIMT) Architecture 39
Figure 2.5: Interprocedural Optimisations Example. 43

Figure 3.1: Using Scopes to Cache Variables 73
Figure 3.2: Java Implementation of Tasks And Task-schedules 74
Figure 3.3: Co-scheduling of Independent Tasks 76
Figure 3.4: Co-scheduling of Data-dependent Tasks 76
Figure 3.5: Co-scheduling of Both Independent and Data-dependent Tasks 77
Figure 3.6: Data-dependent Tasks with a Feedback Loop 78
Figure 3.7: Passing Data Between Different Task-schedules 79
Figure 3.8: Data-flow of Kinect Fusion Pipeline 82
Figure 3.9: Independent Task Execution 84
Figure 3.10: Optimised Task Execution 84

Figure 4.1: Comparing TVM Bytecode to Other HPLs 97
Figure 4.2: Tornado Virtual Machine Architecture 98
Figure 4.3: The Object Cache 103
Figure 4.4: Bytecode executing on the TVM 108
Figure 4.5: Synchronous Execution 111
Figure 4.6: Asynchronous In-order Execution 112
Figure 4.7: Asynchronous Out-of-order Execution 112
Figure 4.8: Dynamic Configuration in the TVM 114

Figure 4.9: Performance Measurement in Asynchronous In-order Mode . 116
Figure 4.10: Performance Measurements in Asynchronous Out-of-order Mode 116

Figure 5.1: Tornado Architecture 123
Figure 5.2: Tornado-IR: Example 124
Figure 5.3: Tornado Device-side Compilation Pipeline 127
Figure 5.4: Tornado-IR: Single Task 129
Figure 5.5: Tornado-IR: Lowering Example 130
Figure 5.6: Tornado-IR: Optimisation Example 130
Figure 5.7: Tornado-IR: Scheduling Example 132
Figure 5.8: Tornado-IR: Multi-task Schedule 134
Figure 5.9: Tornado-IR: After Variable Disambiguation 134
Figure 5.10: Tornado-IR: After Strength Reduction 137
Figure 5.11: Tornado-IR: After Locality Disambiguation 138
Figure 5.12: Example: Redundancy Elimination 139
Figure 5.13: Tornado-IR: After Redundancy Elimination 140
Figure 5.14: Tornado-IR: Asynchronous Nodes 141
Figure 5.15: Tornado-IR: Transforming for Out-of-Order Execution 142
Figure 5.16: Tornado-IR: After OOO Transformation 144
Figure 5.17: Tornado-IR: After Linearisation 145

Figure 6.1: GRAAL-IR: Example - `dot` 154
Figure 6.2: GRAAL-IR: After Lowering 155
Figure 6.3: GRAAL-IR: Control Flow Graph 156
Figure 6.4: GRAAL-IR: After Inlining 161
Figure 6.5: GRAAL-IR: Inlining Example 162
Figure 6.6: GRAAL-IR: Example - `dot3` 164
Figure 6.7: GRAAL-IR: After Partial Escape Analysis 164
Figure 6.8: Jacc System Overview. 169
Figure 6.9: Jacc Benchmark Results 172
Figure 6.10: Jacc Comparison with APARAPI 174

Figure 7.1: Kinect Fusion 182
Figure 7.2: Kinect Fusion Processing Pipeline 185
Figure 7.3: Data-flow in hardware accelerated multi-scale ICP algorithm . 188
Figure 7.4: Data-flow diagram of the build pyramid task-schedule. 197

Figure 7.5: Kinect Fusion Performance Breakdown: Time Spent in Each Pipeline Stage 209
Figure 7.6: Kinect Fusion Performance Breakdown: Per Stage Performance Comparision 209
Figure 7.7: OpenCL: Reduction Kernel 211
Figure 7.8: Tornado: Reduction Kernel 215
Figure 7.9: Performance Improvements Due to the Reduction Kernels . . 220
Figure 7.10: Tornado: Performance Against OpenCL 221
Figure 7.11: Tornado: Performance Against Serial Java 221
Figure 7.12: Tornado Performance Normalised to OpenCL. 223
Figure 7.13: Variation in Frame Rate on the NVIDIA Tesla K20m 227
Figure 7.14: Mean Tornado JIT Compile Times 228
Figure 7.15: Tornado Performance After Optimisation via Dynamic Configuration 232
Figure 7.16: Optimisation Steps for the NVIDIA Tesla K20m 233
Figure 7.17: Optimisation Steps for the NVIDIA GTX 500Ti 235
Figure 7.18: Optimisation Steps for the Intel i7-2600K 236

Listings

Listing 2.1 Example: For-loop 48
Listing 2.2 Example: CUDA - Kernel 48
Listing 2.3 Example: CUDA - Host Side Code 49
Listing 2.4 Example: Parallelisation for MIMD Architectures 51
Listing 2.5 Example: Loop After Partial Unrolling 52
Listing 2.6 Example: Vectorisation in OpenCL 52
Listing 2.7 Example: Vectorising an Irregular Sized Loop in OpenCL 53
Listing 2.8 Example: Vectorisation using Intel SSE Intrinsics 53
Listing 2.9 Example: SIMT Kernel – OpenCL 55
Listing 2.10 Example: More Robust SIMT Kernel – OpenCL 55
Listing 2.11 Example: Device-neutral OpenACC 56
Listing 2.12 Example: APARAPI 58
Listing 2.13 Example: Rootbeer – Kernel 58
Listing 2.14 Example: Rootbeer – Host 58
Listing 2.15 Issue 1: Library Calls 60
Listing 2.16 Issue 2: Coordination 62
Listing 2.17 Issue 3: Eliminating Unnecessary Data Transfers 63

Listing 3.1 Informal Specification: Task-schedule 69
Listing 3.2 Informal Specification: Task 70
Listing 3.3 Example: Task-schedule using the Informal Specification 70
Listing 3.4 Example: Execution of a Task-schedule 70
Listing 3.5 Example: Updating Properties of a Task 71
Listing 3.6 Example: Kinect Fusion Pipeline 81
Listing 3.7 Example: Fine-grained Parallelism 86
Listing 3.8 Example: Coarse-grain Parallelism 86
Listing 3.9 Using Dynamic Configuration from the Command Line 88

Listing 3.10 Using Dynamic Configuration to Migrate a Task-Schedule . . . 90
Listing 3.11 Using Dynamic Configuration to Migrate Individual Tasks 91
Listing 3.12 Using Dynamic Configuration to Randomly Select a Hardware Accelerator 92

Listing 4.1 Tornado Bytecode: Example – foo 96
Listing 4.2 Example: CUDA 97
Listing 4.3 Example: OpenCL 97
Listing 4.4 Example: Tornado Bytecode – foo 107
Listing 4.5 Example: Polymorphic Task Schedule 113
Listing 4.6 TVM Bytecode: Before Optimisation 119
Listing 4.7 TVM Bytecode: After Optimisation 120

Listing 5.1 TVM Bytecode: Final Output 132
Listing 5.2 Example: Task Schedule with Multiple Tasks 134
Listing 5.3 Example: Read-write Set 136
Listing 5.4 Example: Task Schedules 146
Listing 5.5 Issue 1: Executing Back-to-back Task Schedules 146
Listing 5.6 Issue 2: Iterative Execution 147
Listing 5.7 Issue 3: Iterative and Back-to-back Execution 147

Listing 6.1 Example: Java Loop 152
Listing 6.2 Example: Java Bytecode 153
Listing 6.3 Example: Generated OpenCL C - dot 157
Listing 6.4 Example: Dot Product 158
Listing 6.5 Example: Alternate Dot Product 163
Listing 6.6 Example: Generated OpenCL C - dot3 165
Listing 6.7 Example: Jacc - reduction 170

Listing 7.1 Example: Kinect Fusion Pipeline - Java 193
Listing 7.2 Example: Building the Image Pyramid - Java 194
Listing 7.3 Tornado: Task-schedule for Building the Image Pyramid 196
Listing 7.4 Tornado: Replacement Code for Building the Image Pyramid . . 196
Listing 7.5 Tornado: Parallelisation – depthToVertex 198
Listing 7.6 Tornado: Kinect Fusion Pipeline 199
Listing 7.7 OpenCL Bug: Kernel code 203
Listing 7.8 OpenCL Bug: Host code 204

Listing 7.9 OpenCL: Reduction Kernel 212
Listing 7.10 OpenCL: Reduction Task Schedule 213
Listing 7.11 Tornado Reduction Kernel 216
Listing 7.12 Tornado: Reduction Task Schedule 217
Listing 7.13 Tornado Dynamic Configuration For i7-4850HQ 231
Listing 7.14 Tornado Dynamic Configuration For Intel Iris Pro 231
Listing 7.15 Example: Block-cyclic Parallelisation Scheme 237
Listing 7.16 Example: Thread-cyclic Parallelisation Scheme 237

Abstract

Over the last decade computer architectures have changed dramatically leaving us in a position where nearly every desktop computer, laptop, server or mobile phone, has at least one multi-core processor. These systems are also likely to include at least one many-core processor that is used for accelerating graphical applications – called a General Purpose Graphics Processor Unit or GPGPU. By properly utilising these two different processors a software developer could achieve up to two orders of magnitude improvement in performance and/or energy efficiency. Unfortunately, these improvements in performance are often inaccessible to developers due to the combined complexity of understanding both the hardware architecture and the software needed to program them. It is this problem of inaccessibility that is explored within this thesis with the goal being to determine whether it is possible to develop a programming language that allows an application to dynamically adapt to the system it is executing on.

One of the salient issues is that a large amount of prior art is built atop of a *closed-world assumption*: that all the code and the devices it is to execute on are both known ahead of time and are fixed. An assumption that is becoming increasingly unworkable due to the proliferation of heterogeneous hardware. For instance, developers can now run applications in public clouds or mobile devices – contexts where it is difficult to anticipate what hardware an application is executing on, and where it likely that some form of hardware acceleration exists. Handling this uncertainty of not knowing what hardware is available until runtime is a fundamental problem of more statically compiled languages – like C, C++ and FORTRAN. In these languages, the closed-world assumption is obvious: a single processor architecture is assumed so that a single binary executable can be produced.

It is the aim of this thesis is to determine whether it is possible to create a programming language that is able to target modern heterogeneous systems without requiring

any closed-world assumptions about either the number or types of hardware accelerator contained within it. Consequently, this thesis introduces and evaluates Tornado: *the first truly dynamic programming framework for modern heterogeneous architectures*.

The implementation of Tornado is unique as it comprises of three co-designed components: (1) the Tornado API that is designed to decouple the application code that decides on which device code should execute – the co-ordination logic of the application – away from the code that defines the computation – the computation logic of the application; (2) the Tornado Virtual Machine that provides a layer of virtualisation between the application and the underlying architecture of the heterogeneous system; and (3) the Tornado Runtime System – a dynamic optimising compiler that converts code written using the Tornado API into a format consumed by the Tornado Virtual Machine. Tornado has a number of distinguishing features that are a direct result of combining these three key components together. One of these features is the optimisation of co-ordination logic by the Tornado Runtime System – this allows Tornado to automatically minimise the cost of data movement in complex processing pipelines that span multiple devices. Another is dynamic configuration: the ability to have the Tornado Runtime System dynamically re-compile the application at runtime to use a different hardware accelerator, parallelisation scheme, or device setting.

During the evaluation Tornado is tested across thirteen unique hardware accelerators: five multi-core processors, a discrete many-core accelerator, three embedded GPGPUs, and four discrete GPGPUs. In the evaluation it is shown that a complex real-world application, called Kinect Fusion, can be written in Tornado once and executed across all of these devices. Moreover, this portable implementation written in Tornado is able to achieve a maximum speed-up of 55× on a NVIDIA Tesla K20m GPGPU. However, if a little portability can be sacrificed more specialised code can be written that produces a speed-up of 166× on the same device. Tornado is also compared against OpenCL – the state-of-the-art in heterogeneous programming languages – where the specialised implementations of Kinect Fusion run 22% slower and in the best case experience a speed-up of 14× (although this is in an unrealistic scenario). This level of performance translates to speed-ups over the original Java application of between 18× and 150×. Finally, Tornado has been open-sourced so that the reader is able to verify the claims made by this thesis and start writing their own hardware accelerated Java applications – `https://github.com/beehive-lab/Tornado`.

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

This thesis is the culmination of a great deal of work and tears. I would like to thank a few people who helped me on the way...

My wife Rachel for all her words of encouragement and also putting up with me for so long. Edward, Thea, and Laurence for their collective wisdom and encouragement. Sam for arriving unexpectedly and disrupting the writing up process; albeit in an amazing way. My parents Pauline and Derek for providing a landing pad for my visits into University.

Mikel for providing me with the opportunity to undertake this endeavour and your help across the first three years. Christos for helping me get to the finish line with the endless advice, help, perseverance, and emotional support – this certainly would not have happened with out you.

Additionally, I would like to thank members of the EPRSC funded projects I was lucky enough to work with – Anyscale and Pamela – and the wider APT group for the support and advice – especially John and Thanos for tirelessly rebooting my crashed systems.

Finally, my Grandma Doreen who sadly passed away during the final weeks of the write up.

Thank you!

1 | Introduction

Imagine that a developer is writing code that could execute on a dedicated server, a desktop, a laptop, a mobile phone, a public cloud or even a robot. What they know is that an order-of-magnitude increase in performance is possible if a hardware accelerator is available. However, what they do not yet know is whether any of these systems will contain any form of hardware accelerator. Or even if they do how the code is best adapted to take advantage of them. How should they approach writing code for such an uncertain situation? Does the developer: use a high-level programming languages and aim for *portability* or try to use a low-level programming language to aim for the highest *performance*?

In this scenario, the developer has two mutually exclusive options open to them. However, what might not be apparent to them at the outset are the implications of choosing one approach over the other. On one hand, by sacrificing portability it is possible to achieve higher performance but in doing so the developer would become responsible for continually rewriting their application each time it encounters different hardware. Whilst on the other hand, obtaining portability is possible using a high-level programming language, such as Java [52] or JavaScript [69], but in doing so the application would become agnostic to the underlying hardware and leaving it unable to utilise any available hardware acceleration. How should a developer approach this trade-off?

The crux of this problem is that the programming of hardware accelerators is, at best, an afterthought in the design of programming languages. A situation that results in a lack of suitable abstractions to aid developers in programming such complicated systems. What makes this situation worse is that the languages which do support the programming of hardware accelerators all require the developer to pre-selecting the hardware accelerators they expect the application will encounter. A design decision that artificially restricts the final application into using the set of hardware accelerators

that the developer (or sometimes the compiler) predicted they may encounter. Consequently, this means that the final application will not be robust to any changes that happen in the system post-compilation; hence, if a hardware accelerator is added or upgraded it is quite likely that any application wishing to use it will need to be at worst re-written and at best re-compiled. Now ask yourself the question: do you expect to re-compile an application each time you upgrade your GPGPU or even change your system?

The aim of this thesis is to demonstrate that there is no need for a heterogeneous programming language to make any assumptions about either the number or types of components within a heterogeneous system. And by doing so demonstrating that it is possible for an application to be dynamically adapted to its execution context without the need for it to be explicitly re-compiled by the developer. Consequently, this thesis introduces and evaluates Tornado: *the first truly dynamic programming framework for modern heterogeneous architectures*.

By the end, this thesis will both describe how Tornado was implemented and demonstrate it is possible to write a complex real-world application once and then execute it across a range of hardware accelerators without requiring re-compilation. During Chapter 7 Tornado is tested across thirteen unique hardware accelerators: five multi-core processors, a discrete many-core accelerator, three embedded GPGPUs and four discrete GPGPUs. Initially, the Chapter starts by taking the real-world application, called Kinect Fusion, and that by using Tornado it is possible to execute across all thirteen devices. Moreover, this initial implementation achieves a maximum speed-up of 55$\times$ on a NVIDIA Tesla K20m GPGPU. However, later on it shows that if a little portability can be sacrificed then more specialised code can be written to produce a speed-up of 166$\times$ on the same device. Tornado is also compared against the state-of-the-art in heterogeneous programming languages, OpenCL, and that the specialised Tornado implementations of Kinect Fusion: run 21% slower than OpenCL in the worst case, and in the best case 14$\times$ faster. The latter result unfortunately happens in an unrealistic, but interesting, scenario that is discussed in Section 7.8.3. More importantly, so that the reader is able to verify the claims made by this thesis and start writing their own hardware accelerated Java applications, Tornado is available as open-source software: `https://github.com/beehive-lab/Tornado` – However, before jumping too deeply into this thesis the remainder of this Section aims to describe the challenges and research outcomes of this work.

1.1 Challenges

Currently, we are experiencing a large change in the way computers are being used: there is a move away from using them as tools for solving numerical problems and we are now starting to used them more as an assistive technology. For instance, there is large growth in fields – such as robotics, virtual reality, and machine-learning – that is being driven by having smaller more capable computers. In the HiPEAC Vision 2017 report [41] this is being called the beginning of the "centaur-era". Throughout this report a number of key technological challenges are highlighted for the coming years. Out of these challenges there are five that are directly relevant to this thesis and, as such, will be outlined in the next five Sections. Any interested reader can find these challenges in the following Sections of the report: [41, Section 2.5.7.1.4], [41, Section 2.5.7.2], [41, Section 2.5.7.3] and [41, Section 2.5.7.4].

1.1.1 Evolution of Devices

Over the last decade we have been experiencing the evolution of systems where computer architectures are becoming more complex. Today we have reached a point where simply increasing the number of processors in a system no longer provides the desired increases in performance. Subsequently, we are witnessing a shift away from systems containing many identical processors, referred to as homogeneous systems (see Figure 1.1), to systems that contain a mix of different processor types. It is now commonplace for a modern computer to contain at least one multi-core processor and a programmable General Purpose Graphics Processing Unit (or GPGPU). However, in addition to these two types of processor they may also contain some others like: Digital Signal Processors (DSPs), Field Programmable Field Arrays (FPGAs), or fixed-function accelerators (such as a cryptographic accelerator). As these systems contain a mix on non-identical processors they are considered to be *heterogeneous* (see Figure 1.2).

Now what makes heterogeneous systems complex to program is that sometimes the difference between processors can be stark – like the difference between a multi-core processor and a FPGA – and at other times it is very subtle – like having either an in-order or an out-of-order instruction pipeline (as is the case with big .LITTLE processors from ARM). This complexity is also not helped by the fact that these processors do not just exist on the same chip; they can exist as peripherals – like a GPGPU that is accessible over a PCIe bus – with their own disparate memories. Any programmer

wishing to make use of them needs to program them like a distributed system.

The advantage of a homogeneous system over a heterogeneous one is that each processor has exactly the same properties; hence, all processors behave in the same way. Coupling predictable behaviour with shared-memory means that developers do not need to contend with transferring data between different memories. Consequently, once an executable is loaded into memory it is instantly available to all other processors inside the same operating system process. As a result there is little complexity involved in migrating an application between cores and most of the time this is handled transparently by the operating system.

The drawbacks of heterogeneous systems are the cost, both in time and complexity, of choreographing the control of the application across multiple non-identical processor. For instance, as each processor may execute a different dialect of machine code a single logical program needs to be split into a series of sub-programs that can be compiled separately for each processor. Normally, the sub-program that contains the program entry point is called the *host-side code* (or *host-code*) and the others *kernels* (or *device-code*). As well as code-generation issues, larger problems are caused because heterogeneous systems are commonly distributed-memory machines. Typically, this means that a processor is not be able to transparently access data that resides is in the memory of another device. In this situation, it is common for a language to force a developer to manually intervene and explicitly copy data between the memories. However, if some of these issues can be overcome the clear advantage of using a heterogeneous systems is a one to two orders of magnitude increase in performance.

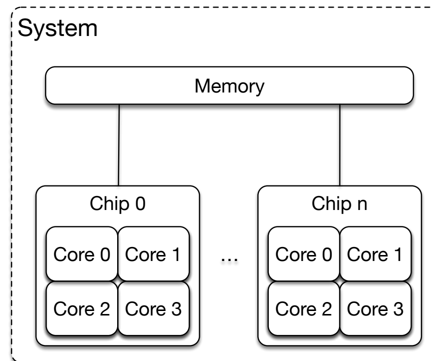


Figure 1.1: An example of a homogeneous system. For many this style of system architecture is synonymous with the computer and is characterised as being a standard Intel PC. Note that all the processor cores are identical – they have just been replicated lots of times. Programming this type of system is straightforward as all cores speak the same machine language and access the same memory.

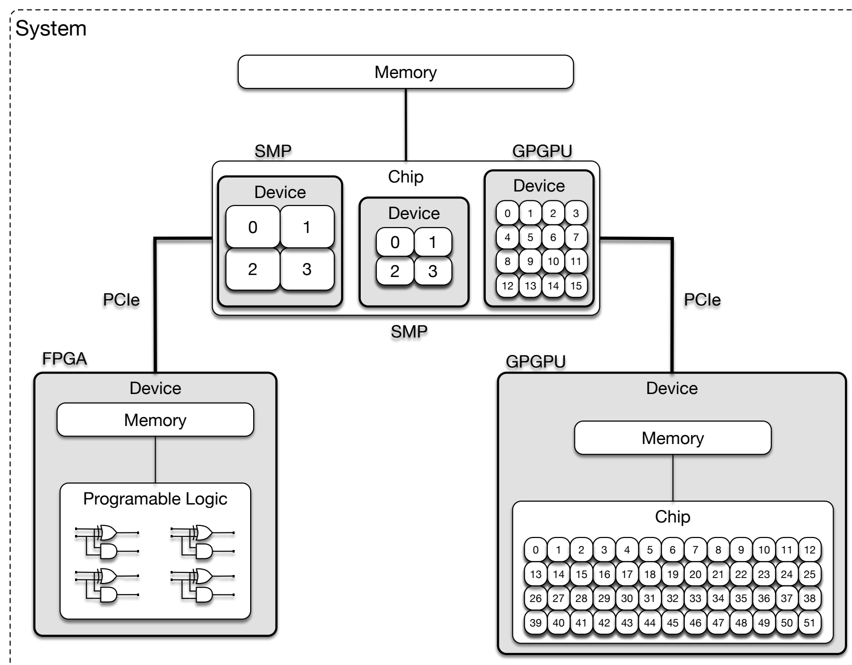


Figure 1.2: An example of a heterogeneous system. Notice that unlike the homogeneous system the cores are not all identical. For instance, some devices have few more capable cores whereas others have more simpler cores. Also take notice of the fact that there are multiple devices in the system that can each have their own memory. The complexity in programming this system is in keeping data synchronised between devices and understanding how best to utilise each type of core. If you are wondering, in the case of FPGAs there are no cores: a developer programs these by joining up logic gates to create circuits.

1.1.2 The Productivity Challenge

As computer systems become more heterogeneous they have a greater potential to increase both the performance and energy efficiency of an application. However, what many developers struggle with is knowing how to program these devices. This stems from the fact that the majority of specialised hardware accelerators, like GPGPUs, have to be programmed using a specialist programming language. Unfortunately for developers these languages are typically low-level languages that sit at the static end of the spectrum.

As these languages were created in the 1950s, 1970s and 1980s – FORTRAN [9], C [100], and C++ [116] respectively – the ability to program hardware accelerators, or even parallel programming, is at best an afterthought in their design. Meaning that developers using these languages often find that there is a lack of good abstractions to aid them in programming modern heterogeneous architectures. This leaves developers in a position where only an elite few with intricate hardware knowledge and the enthusiasm to program at a low-level are able to program these devices.

For the majority of developers, however, this is a wasted opportunity as the increasing ubiquity of hardware accelerators means that an application will never be executing more than a few centimeters away from a processor that has the potential to dramatically increase its performance.

The challenge now is to create a way for developers to program hardware accelerators without either needing expert knowledge of the hardware or the need to use a low-level programming language. This is not a trivial task, however, as the programming language needs the ability to: (1) partition the control of the application between devices; (2) transfer data between disparate memories; and (3) provide a means of for generating high performing code for the device. Here the the last point is especially important as it might require a new language to generate highly-parallel code or expose abstractions to the developer that provide a mechanism for obtaining high performance – like exposing low-level intrinsic operations. It is in meeting these three requirements without negatively impacting programming productivity that make the design of programming languages for heterogeneous systems difficult. From herein, the phrase "heterogeneous programming language" will be used to denote a programming language that is capable of being used to program heterogeneous systems.

1.1.3 Evolving Use of Programming Languages

One of the observable trends with respect to development tools is that developers are shifting away from using low-level languages, such as C, C++, and FORTRAN, and are now working with more high-level and dynamic languages, such as JavaScript [69], Ruby [103], and R [118]. A trend that is observable in the TIOBE Index [8] and in the IEEE programming language rankings [21]. However, a key problem with heterogeneous systems is that they often need to be programmed using a low-level programming language: a requirement that can often put them out of reach of ordinary developers. Therefore, a concerted effort is needed to try to enable newer and more dynamic programming languages to take advantage of hardware acceleration.

1.1.4 The Correctness Challenge

All programming languages make one very fundamental assumption: functional correctness. Every developer expects that when they write an application that the program will execute in a very specific way: they expect that the program executes their source code deterministically – line by line (this is called program order). This behaviour is important for two reasons: (1) it means that given an input a program will always produce the desired output, and (2) it provides developers with a way to reason about the execution of their application without actually having to run it.

Now, one of the biggest issues with heterogeneous systems is that they are effectively distributed systems. In a distributed system there are many disparate devices that are all running local copies of a program. The problem for the developer is in ensuring that all these disparate devices are synchronised correctly. If the devices are not synchronised then the application might not deliver the right data to the right device at the right time: leading to non-deterministic behaviour and likely a incorrect result.

From a programming language perspective this means that a heterogeneous programming language needs to provide features that allow an application to be divided up and executed on many disparate devices whilst also preserving correctness.

1.1.5 The Performance Challenge

The last challenge is the one of performance: that a programming language needs to be able to properly utilise the underlying hardware. This might mean being able to generate machine code for a specialised processor or the ability to exploit complex

instructions (like Intel SSE/AVX instructions [67]). Perhaps, one of the most fundamental challenges of heterogeneous programming languages is addressing the issue of generating highly parallel code. This feature is essential if an application is to execute on a GPGPU that is capable of having tens of thousands of parallel threads running simultaneously.

There is a second and more subtle performance challenge that is especially relevant to this thesis: the design of the programming language. For instance, many heterogeneous programming languages are derived from C as it is a relatively straightforward language to implement by today's standards. However, modern more dynamic languages tend to use more complex abstractions, such as object-orientation and dynamic typing, which sometimes have a detrimental impact on performance. There is an open question whether some of these language features fundamentally prohibit their use for programming hardware accelerators.

1.2 Addressing These Challenges

The vehicle used in this thesis to address these aforementioned challenges is a newly constructed heterogeneous programming framework called Tornado. The goal of Tornado is to demonstrate that it is possible for a hardware accelerated application to be constructed in a high-level programming language – Java. One of the advantages of Java is that applications are developed to be architecture-neutral – this means an application can be written once and executed on many different systems irrespective of the operating system or the underlying processor. Hence, a key outcomes of this thesis is to show that, in principle, applications can be constructed that are agnostic to the type of hardware acceleration that they use: i.e. that these applications are device-neutral. Achieving this outcome can be broken down into two parts: (1) allowing a developer to express, at an abstract level, what code needs to be run on a hardware accelerator, and (2) developing a framework that uses this abstraction along with runtime knowledge – of both the application and system – to transparently compile and execute the application using an available hardware accelerator. Thus, making it possible for an application to be compiled once and executed across many different types of hardware accelerators.

By meeting this goal this work will actively contribute to tackling key aspects of the challenges described in Section 1.1. For instance, this work is especially well aligned with the productivity challenge caused by the evolution of devices and the

performance challenges faced due to the evolving use of programming languages and development tools. Both of these challenges are partly addressed by demonstrating that it is possible for a modern language to support the programming of heterogeneous systems. One of the underlying themes throughout this thesis is that a heterogeneous programming framework is being developed using an object-orientated programming language; one that relies on the use of a complex runtime system in the form of the Java Virtual Machine [78]. Hence, this work will contribute to understanding how modern virtual machine based languages can be adapted to use hardware acceleration. Below is a detailed list of where these challenges are discussed in this thesis:

- Tornado introduces two key programming abstractions – tasks and task-schedules – to help decouple the coordination logic of an application from the computational logic. Their design and rationale are outlined in Section 3.1.

- Using the task abstractions the developer is able to compose complex multi-stage processing pipelines. Examples of how these are written can be found in Section 3.1.5 and an in-depth discussion on how Tornado is able to dynamically optimise these pipelines for a given system is discussed in Section 5.3.

- Tornado has programming abstractions that allow it to generate high performance code for hardware accelerators that have multiple cores. The discussion on this feature is found in Section 3.2.

- As Tornado does not know what hardware acceleration is available until runtime it needs a mechanism to handle this uncertainty. Hence, its runtime system is designed so that it is possible to dynamically configure and re-compile the application. A discussion of how this is exposed to the user is found in Section 3.3 and an example of an multi-state processing pipeline that on each invocation executes each stage on a randomly selected hardware accelerator can be found in Listing 3.12.

- The task and task-schedule abstractions in Tornado make it is possible for the system to transparently schedule data transfers and the execution of code on devices. An advantage of introducing these features is that ensuring the application executes correctly is handled transparently by Tornado – helping to address parts of the correctness and productivity challenges. A discussion on how this works is given in Section 5.2.

- Supporting this ability to dynamically compile an application for each execution context are three components: (1) the Tornado Virtual Machine which provides a virtualisation layer between the application and the hardware; (2) the Tornado Runtime System which provides a dynamic optimising compiler for the Tornado API; and (3) the Tornado Virtual Machine Client compiler that Just-In-Time (JIT) compiles Java bytecode for each device. The design rationale and key implementation details of these components are discussed in Chapters 4 to 6.

- As Java is an object-orientated language Tornado needs the ability to compile it into highly performing code for hardware accelerators. Section 6.1 describes how compilation works and Section 6.2 discusses how Tornado is able to compile idiomatic Java code.

1.3 Research Contributions

The key contribution of this thesis, demonstrated throughout Chapter 7, is that Tornado can be used to construct a complex real-world application that is executed across thirteen hardware accelerators: five multi-core processors, a discrete many-core accelerator, three embedded GPGPUs and four discrete GPGPUs. (The full list of devices used are shown in Table 1.1.) By demonstrating the feasibility of an approach like Tornado makes a direct and timely attempt to tackle some of the main challenges faced by the evolution of devices (as outlined earlier in Section 1.1).

Perhaps, one of the most fundamental research contributions is this thesis is that the problems addressed by Tornado are solved using existing techniques that are commonly used to implement dynamic programming languages – such as bytecode interpreters (the Tornado Virtual Machine in Chapter 4), dynamic compilation (the Tornado Runtime System in Chapter 5 and the Tornado Virtual Machine Client in Chapter 6), and dynamic de-optimisation (the Tornado Runtime System in Chapter 5) – it is just that they are being applied to a new context: the programming of heterogeneous systems.

By combining these technologies together to form Tornado, a novel programming framework is created that can *dynamically* adapt an application to a specific hardware context without the need to change the source code or even re-compile the application. Finally, Tornado has been released as an open-source project to benefit the research community and is available at `https://github.com/beehive-lab/Tornado`.

System	OS	Accelerator	Type
Laptop	OSX 10.11.6	Intel i7-4850HQ Intel Iris Pro 5200 NVIDIA GT 750M	mutli-core processor integrated GPGPU external GPGPU
Desktop 1	Fedora 21	AMD A10-7850K AMD Radeon R7	multi-core processor integrated GPGPU
Desktop 2	Fedora 25	Intel i7-2600K NVIDIA GTX 550 Ti	multi-core processor external GPGPU
Server 1	CentOS 6.8	Intel Xeon E5-2620 Intel Xeon Phi 5110P NVIDIA Tesla K20m	multi-core processors external many-core device external GPGPU
Server 2	CentOS 7	Intel Xeon E3-1285 Intel Iris Pro P6300 AMD Radeon HD 6970	multi-core processor integrated GPGPU external GPGPU

Table 1.1: Target System Configurations

1.3.1 Publications

Alongside this thesis there are a number of peer reviewed publications that support this research. Below is a list of publications relating to Tornado where I am the first author:

- James Clarkson et al. "Boosting Java Performance Using GPGPUs". In: *Architecture of Computing Systems - ARCS 2017*. Ed. by Jens Knoop et al. Cham: Springer International Publishing, 2017, pp. 59–70. ISBN: 978-3-319-54999-6

- James Clarkson et al. "Towards Practical Heterogeneous Virtual Machines". In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. Programming 2018. Nice, France: ACM, 2018, pp. 46–48. ISBN: 978-1-4503-5513-1. DOI: 10.1145/3191697.3191730. URL: http://doi.acm.org/10.1145/3191697.3191730

- James Clarkson et al. "Exploiting High-performance Heterogeneous Hardware for Java Programs Using Graal". In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. ManLang '18. Linz, Austria: ACM, 2018, 4:1–4:13. ISBN: 978-1-4503-6424-9. DOI: 10.1145/3237009.3237016. URL: http://doi.acm.org/10.1145/3237009.3237016

There is also additional publications that relate to Tornado where I am not the first

author:

- Christos Kotselidis et al. “Heterogeneous Managed Runtime Systems: A Computer Vision Case Study”. In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’17. Xi’an, China: ACM, 2017, pp. 74–82. ISBN: 978-1-4503-4948-2. DOI: 10.1145/3050748.3050764. URL: http://doi.acm.org/10.1145/3050748.3050764
- S. Saeedi et al. “Navigating the Landscape for Real-Time Localization and Mapping for Robotics and Virtual and Augmented Reality”. In: *Proceedings of the IEEE* (2018), pp. 1–20. ISSN: 0018-9219. DOI: 10.1109/JPROC.2018.2856739

1.4 My Thesis

The aim of this final Section is to outline both my thesis and how each Chapter contributes to proving it. Firstly, a reminder of what my thesis is: that there is no need for a heterogeneous programming language to make any assumptions about either the number of or type of components within a heterogeneous system.

This thesis will be proven by creating a heterogeneous programming framework – Tornado – that allows an application to be dynamically adapted to its execution context without the need for it to be explicitly re-compiled by the developer.

The ability of Tornado to prove the thesis will be evaluated by demonstrating that a complex real-world application can be written using Tornado and transparently executed across the thirteen devices listed in Table 1.1. Additionally, extra focus will be placed on ensuring that the performance of Tornado is comparable with a state-of-the-art heterogeneous programming language: OpenCL. To understand how Tornado is able to achieve this goal a discussion on the design and implementation of each of the major components of Tornado will be provided. Finally, to help navigate this document a brief summary of each of the Chapters is included below:

Chapter 2: Background provides a refresher on common processor architectures – like SISD (in Section 2.1.1), MIMD (in Section 2.1.2), SIMD (in Section 2.1.3) and SIMT (in Section 2.1.4) – and explains the notion of a heterogeneous programming language. It also includes a breakdown of the prior-art in Section 2.4 and summarises the key themes in emerging heterogeneous programming languages Section 2.4.9. It concludes by providing a set of motivating examples,

Section 2.5, that cause problems in existing heterogeneous programming languages but are solved by Tornado.

Chapter 3: Tornado Programming API describes how a programmer interacts with the Tornado framework. In Section 3.1.2 it introduces the programming model and two key programming abstractions – tasks and task schedules – that decouple an applications coordination logic from its computational logic. Thereby, allowing programmers to compose complex processing pipelines that can be transparently mapped onto different system topologies; for example to use GPGPU acceleration. Examples of how this is done are provided in Section 3.1.5. Next, this Chapter describes how Tornado implements support for generating parallel code in Section 3.2.1 – addressing one of the fundamental challenges of heterogeneous programming (see Section 1.1.5). Finally, the notion of dynamic configuration is introduced in Section 3.3 that allows an application to be adapted to make use of different hardware accelerators without the need to explicitly re-compile or modify the application.

Chapter 4: Tornado Virtual Machine describes the component that provides the virtualisation layer between the application and the physical components of the heterogeneous system. It is this component that provides an abstraction upon which the Tornado API is built. Moreover, by avoiding making assumptions about the components with a heterogeneous system Section 4.5.3 will describe how it is possible to dynamically adapt each application – in terms of which hardware accelerator is used – at runtime. The Tornado Virtual Machine is a bytecode interpreter and its design is covered in Section 4.3 with explanations of its key features in Section 4.5.

Chapter 5: Tornado Runtime System describes how the task and task-schedule abstractions are dynamically compiled to target the Tornado Virtual Machine. Initially the architecture of the Tornado Runtime System is described in Section 5.1 where the incoming application is split so that task-schedules are compiled and optimised by the Tornado Runtime System (Section 5.1.1) and tasks are sent down a separate compilation pipeline (Section 5.1.5) to produce machine code for the hardware accelerator. Finally, one of the most important aspects of this Chapter is how the Tornado runtime system optimises the execution of task schedules, by allowing tasks to be executed out-of-order (Section 5.3.7), whilst also minimising the amount of data transferred (Section 5.4).

Chapter 6: Tornado Virtual Machine Client describes how tasks are compiled from Java bytecode into OpenCL C for each device. One of the key aspects of this Chapter is that it explains how idiomatic Java code is compiled in Section 6.2. It also includes a discussion on which language features are supported by Tornado (Section 6.3) and how the choice to target OpenCL C limits which features can be implemented (Section 6.4).

Chapter 7: Evaluation describes how a complex real-world application can be written once using Tornado and executed across thirteen unique hardware accelerators: five multi-core processors, a discrete many-core accelerator, three embedded GPGPUs and four discrete GPGPUs. Section 7.5 evaluates this ability of Tornado to execute an application across multiple accelerators. Demonstrating that use Tornado on a NVIDIA Tesla K20m GPGPU the performance of the application was increased by a maximum of $55\times$ over its serial Java implementation. Following on from this, Section 7.6 evaluates the ability to specialise Tornado applications to improve their performance and outlines how using the same GPGPU a maximum speed-up of $167\times$ over serial Java is achievable. Finally, the impact of dynamic configuration is evaluated to show that in two scenarios applications can be further specialised – by using different thread configurations or OpenCL driver features – to improve performance by between 14-17%.

Chapter 8: Conclusion provides critical analysis of the work presented in this thesis, an overview of the limitations of what has been presented (Section 8.4) and concludes by outlining possible future directions of Tornado and other heterogeneous programming languages (Section 8.5.1).

2 | Background

Hardware accelerators are not new: ever since the first general purpose processor appeared, people have been working on additional hardware to provide higher levels of performance. Perhaps, one of the most common examples is the drive to increase the performance of floating-point arithmetic for scientific computation. A path that lead to the development of math coprocessors like the Intel x87 [97]. However, this was not the only domain where performance was problematic and throughout this period hardware accelerators were also developed to target applications within other domains: such as Computer Aided Design [10, 60, 112], Computer Graphics [45, 83, 121], and even speech recognition [6]. These pioneering accelerators still influence how computers are designed today. For instance, the early work accelerating computer graphics paved the way for the modern general purpose graphics processing units (GPGPUs) [35, 84, 85] that can also accelerate non-graphical workloads. The underlying trend has been for these standalone accelerators to be amalgamated into larger processor architectures, like x86 or ARM, where they now no longer exist in the own right.

Historically, one of the problems of using specialised hardware accelerators has been their cost. A issue that forced chip designers to carefully trade-off what hardware is included in a system against both the area it consumes and its monitory cost. Therefore, if an accelerator was deemed to be too specialised then it would not appear in commodity computers as it would be unnecessary for the majority of users. Due to this trade-off, it was more preferential to look towards architectural innovation to improve performance in the general case. Notable innovations between the early-1970s to the late-1990s include: data-flow processors [55], decoupled access-execute architectures [109], Very Long Instruction Word (VLIW) processors [43], and the vector processor [11, 105, 123]. The latter was popularised by the eponymous line of Cray supercomputers that continued this innovation into the late 90's and ended with the Cray XMT – a supercomputer that contained a novel highly-threaded processor architecture [110]. Eventually, the commoditisation of the x86 processor and it's Streaming

SIMD Extensions (SSE) [67] lead to the demise of bespoke processor designs.

Fast-forward to today, and a resurgence of hardware accelerated systems is under way. In enterprise computing processors now contain a range of hardware accelerators that improve the performance of cryptography, compression, regular expression matching, SQL queries, and parsing XML. Prime examples of this are the Sparc M7 [4] and PowerEN [79] chips. What has been more noticeable is the commoditisation of many-core and FPGA accelerators [31, 84, 111]. The consequences of which is that accelerators, like GPGPUs, are now usable in a wide range of commodity computer systems such as mobile phones, tablets, laptops, PCs, and servers.

Both of these strands of innovation – hardware accelerators and architectural innovation – have the *potential* to provide order-of-magnitude increases in performance. These promises of improvements to performance (or energy efficiency) are warmly received by developers but only on the premise that they are readily realisable. From the perspective of the developer, these benefits should come cheaply – as for many the cost of changing programming language or even re-writing an application is prohibitive.

The resulting problem is that the actual value of these innovations is only realisable once developers have the capability of exploiting them in software. Now as there are relatively few programming languages capable of programming these devices a software crisis exists – from which the challenges outlined earlier in Section 1.1 are bourne. However, before tackling any challenges it is first prudent to cover two fundamental topics: what heterogeneous architectures exist and how they are currently programmed.

2.1 Processor Architectures

Heterogeneous systems (or architectures) comprise of multiple cores (or processors) that come in a variety of forms. Over the next few Sections a high-level overview is given of the typical architectures that a developer might encounter in a modern computer. For these purposes, Flynn's taxonomy [44] will be used to highlight the differences between processor architectures (with a few amendments to address more modern architectures).

Flynn's taxonomy classifies architectures according to the amount of instruction and data streams available to each *processing element* (or PE). An *instruction stream* is simply a series of machine instructions that are being executed by the processor. Note that in some literature, an instruction stream may be referred to as a program but

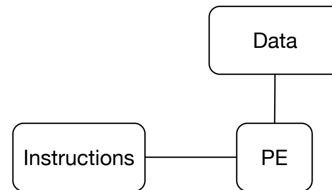


Figure 2.1: A Single Instruction Single Data Stream architecture is a single PE that has a single data and instruction stream. This architecture is not able to exploit either TLP or DLP on its own but can be used as a building block to create more complex processor architectures that can.

this thesis will not use this terminology. A *data stream* can be thought of as a register file or main memory depending on whether the processor exists inside a chip or is the chip. This definition is a little more ambiguous due to the ability to integrate many processors within a single chip.

Even before describing any architectures, it should be clear how performance improvements are to be made: by either increasing the number of instruction and data streams. Hence, these changes will allow *parallelism* to be exploited between streams. In the situation where multiple instruction streams exist *thread-level* (TLP) parallelism is used to process multiple streams of instructions. Whereas in the situation where multiple data streams exist *data-level parallelism* (DLP) is used so that multiple data streams can be operated on together. An important point is that these two types of parallelism are not mutually exclusive and are exploitable simultaneously. For completeness, there is also a third form of parallelism – *instruction-level parallelism* – where multiple instructions within the same instruction stream can be executed in parallel. Although using ILP can improve performance it is often impractical to exploit and is less likely to result in the same level of performance gains that are achieved through the use of TLP or DLP [61].

2.1.1 Single Instruction Single Data Stream

A Single Instruction Single Data Stream (SISD) architecture, shown in Figure 2.1, is the one that most developers envision they are programming when writing code. They are designed to serially execute each program instruction by instruction. As a result, SISD architectures are ideally suited to problems where a low-latency is desirable. For instance, they work well with programs that are control-flow heavy, use a lot of indirection, or have strict real-time constraints. A practical example would be a single

.

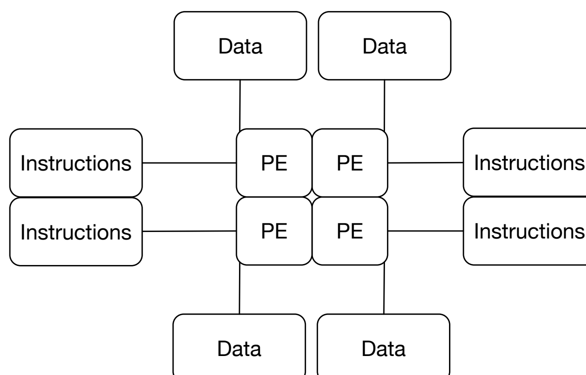


Figure 2.2: A Multiple Instruction Multiple Data Stream architecture. Here a Processing Element (PE) could equally be a single processor core or an entire chip. Notice how four SISD architectures are combined to form a single MIMD architecture.

processing core within a modern x86 processor. From a developers perspective, they are the most programmable because there is no need to explicitly exploit any form of parallelism. SISD architectures commonly exist in two forms: either an in-order or out-of-order design [119]. The advantage of an in-order PE is its simplicity: it can be implemented using less area and are generally more power efficient. Consequently, this makes them an ideal as a building block for larger more complex architectures. An out-of-order PE, however, is able to achieve higher performance by exploiting ILP. For instance, they have the ability to execute other instructions while waiting for long-latency memory operations to complete. A real-world example of how these two designs can be exploited is the ARM big .LITTLE architecture [2] that uses a combination of in-order and out-of-order cores to improve the energy efficiency of mobile phones.

2.1.2 Multiple Instructions Multiple Data Streams

A natural way to extend a SISD architecture is to replicate it multiple times within the same chip – exactly how a Multiple Instruction Multiple Data stream (MIMD) architecture is created. Whereas a SISD architecture is a single x86 core, a MIMD architecture can be thought of as either a multi-core x86 processor within a chip or multiple x86 processors on separate chips. An example of a MIMD architecture is shown in Figure 2.2. Typically, applications like web-servers benefit from MIMD architectures as

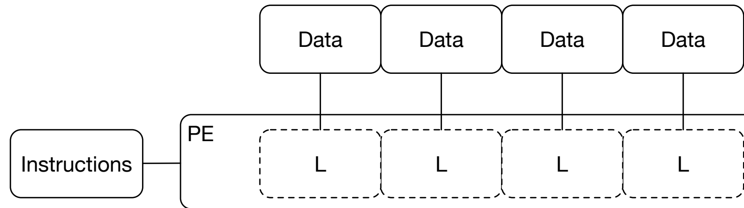


Figure 2.3: A Single Instruction Multiple Data Stream architecture. Here a processing element is composed of multiple lanes (marked L) that allow different data streams to be processed in parallel. In this situation a single instruction, e.g. an addition, is performed across all lanes simultaneously meaning that a program can produce four results for each instruction issued.

they can improve throughput by starting an extra instance of themselves on a different core. By using this approach there is no need to parallelise the web-server. More generally, MIMD architectures are suited to problems where a large number of tasks can be executed in isolation. Although MIMD architectures are very versatile and can exploit all forms of parallelism. The only drawback is that some applications can suffer performance degradation when data needs to move frequently between processing elements. Typically, this occurs in shared-memory machines when different processing elements access different parts of the same cache line (the technical term for this is *false sharing*).

2.1.3 Single Instruction Multiple Data Streams

A limitation of MIMD architectures is that they are very costly to implement both in terms of complexity and the area they consume on a chip. Most of the complexity stems from the need to manage the execution of multiple threads. Therefore, to avoid this complexity a Single Instruction Multiple Data stream (SIMD) architecture can be employed to exploit data-level parallelism from within a single instruction stream. Normally, SIMD units are more energy efficient than their MIMD counterparts as they can be embedded inside a larger processor architecture to improve its performance. Examples of this form of SIMD architectures are Intel SSE/AVX[67], ARM NEON [88] and the ARM Scalable Vector Extensions [113]. Note that sometimes SIMD architectures are commonly known as vector processors.

Historically, SIMD processors have been considered difficult to use because of their inflexibility to cope with unknown or irregular lengths of data. This problem stems from the fact that a single processing element is composed of several lanes – one

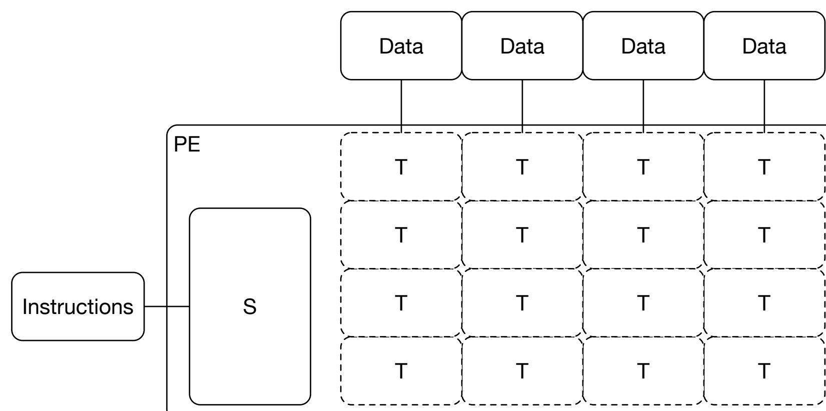


Figure 2.4: A Single Instruction Multiple Thread architecture. A single instruction stream is fed into a scheduler (marked S). The scheduler is responsible for issuing instructions to a group of threads (in this example there are four threads per group and each thread is labelled T). The schedule then multiplexes the execution of multiple thread groups across multiple physical cores. In this example, there are four thread-groups being multiplexed across four physical cores. The result is a architecture that has high-throughput and the ability to amortise the cost of lengthy memory operations. Take note at how similar SIMT is to SIMD – it is also organised into lanes but has more flexibility to disable them when needed.

to process each data stream – that execute unconditionally. However, recent advances in SIMD architecture design now relax this constraint and allow lanes to be disabled using per-lane predication [113]. As a result a new form of more programmable SIMD unit is starting to emerge.

2.1.4 Single Instruction Multiple Threads

A Single Instruction Multiple Thread (SIMT) architecture, shown in Figure 2.4, aims to overcome the historical issues of the SIMD design and exploit data-level parallelism on a larger scale. SIMT architectures typically exist in modern GPGPUs [84] that render complex computer graphics at high frame rates by using tens of thousands of parallel threads. Internally, they are structured similarly to a SIMD architecture where a processing element has multiple lanes to process multiple data streams. However, instead of applying instructions to multiple lanes they are applied to multiple *threads*. Here the distinction is that each thread has its own program counter that allows it to make forward progress in the situation where the execution paths of threads diverge – i.e. it is not a strict requirement for threads to execute in lock-step with each other.

Using a SIMT processor requires a developer to provide both a stream of instructions and specify how many threads should be used to execute them. Internally, a hardware scheduler will create batches of threads and issue instructions to each batch. Once a batch completes a new batch is then scheduled. What allows a SIMT processor to be utilised more efficiently that a SIMD architecture is that each batch is further subdivided into groups of threads – these are commonly referred to as *warps* on NVIDIA GPGPUs. Normally, the size of a warp is equal to the number of processing lanes in the processing element. By scheduling in terms of batches and warps, the hardware scheduler can overlap the execution of multiple threads within a single processing lane. Unlike some threaded architectures switching between threads in a SIMT processor is performed very quickly – typically on a per clock cycle basis – which allows forward progress to be made while awaiting results of long-latency operations.

The outcome is that SIMT architectures are ideally suited to amortising the cost of memory accesses. The other advantage of issuing instructions to threads is that SIMT architectures, unlike traditional SIMD architectures, are able to conditionally execute individual threads (or each lane). This aids the processing of irregular problem sizes but also allows them to handle control-flow divergence. As a result, this architecture is regarded as being easier to program than a SIMD architecture. However, SIMT architectures often struggle when there is not enough data-parallelism available to keep all the lanes utilised. Typically, this happens when an application exhibits a lot of control-flow divergence. As SIMT architectures are very complex to understand interested readers are directed to [75] for more information.

2.2 Programming Languages

The term programming language is often ambiguous as it is used interchangeably to refer to either: the syntax and semantics of a particular language; or the components that implement a particular language – such as the compiler and runtime system. Although a language can exist in an abstract form of syntax and semantics, it only becomes useful for a developer once an ecosystem of components exist that implement the desired behaviour of the language. For instance, the core components of both the C [100] and C++ [116] programming languages is a compiler – that turns source code into a binary executable or library – and a set of standard libraries. As there are no hard and fast rules as to how a programming language is implemented each language may be implemented using a unique set of components. For example, languages such as Self [26]

and Java [52] have a complex runtime system, called a virtual machine, that amongst other things provides support for memory management and Just-In-Time (JIT) compilation. Others like R [118] are purely interpreted and as such do not have the need for a compiler.

In this thesis, the term programming language will be used to encompass everything needed to implement a particular language: syntax, semantics, compiler and the runtime system. And the salient point that is made throughout this thesis is that it is often the fundamental assumptions made during the design of a programming language that dictate how difficult it is to adapt to allow the programming of heterogeneous systems.

2.2.1 Design Decisions

It is very common for programming languages (or more specifically their implementing components) to be constructed according to a series of fundamental design decisions; these are necessary for a variety of reasons but some examples are:

1. That a compiler generates either an executable or library in a machine executable format.
2. That all compiler outputs uses the same machine language.
3. That all code and libraries are available to the compiler at compile-time.

Despite looking innocuous even these simple decisions can have a profound impact on how a programming language can operate, as will be highlighted in the next three sections.

2.2.1.1 Binary Artefacts

The decision made in the first example seems straightforward: that a compiler should generate its artefacts in a machine executable format. And is a common decision taken when implementing a programming language, as it has a number of important repercussions:

- that the binary artefact can be executed directly, albeit with some minimal operating system and library support.
- that the source code needs to be translated (or compiled) into a specific machine language, such as x86 or Aarch64.

- that the executable is only valid for a single processor architecture, and cannot execute on processors that have different architectures.

Generally, this is a common decision made by statically compiled languages, such as C, C++ and FORTRAN, where there is emphasis on keeping the language implementation simple so that only a compiler needs to be implemented.

2.2.1.2 Code Generation

The second decision is made to help simplify the implementation of the compiler by requiring that: all artefacts that it produces target the same machine language. Although this decision does not preclude the use of the compiler to target different machine languages, it does make an an underlying assumption that the application will execute in its entirety on a single type of processor. An assumption that is clearly going to be broken when compiling for heterogeneous systems (see Section 1.1.1).

One of the direct consequences of making this decision is the fact that a machine language needs to be specified in order for the compiler to generate an executable. For some programming languages – like C, C++, FORTRAN – this is a decision made at compile-time. Whereas in other languages, like Java, this decision can be made at run-time. The important point to note here is that if an application is to remain portable across different machine languages, then the decision on which machine language to target needs to made at the last possible moment. Otherwise, the application will either need to be continually re-compiled if the machine language needs to changed or the compiler adapted to produce a monolithic executable containing all possible machine languages. Neither of these options are particularly scalable or workable in the long term and are a key reason why Tornado is implemented to use a JIT compiler.

2.2.1.3 Interprocedural Optimisations

The third decision – that the compiler is designed to require knowledge of all the code and libraries used by an application – is key to allowing compiler optimisations that exploit information about how data and control flow through the application. These optimisations are known as *interprocedural* optimisations and to show their impact Figure 2.5 shows how two such optimisations – constant propagation and inlining – can be applied.

What is important to note is that because the compiler is able to determine that all calls to `add` pass the constant `1` to variable `b` it is able to use constant propagation

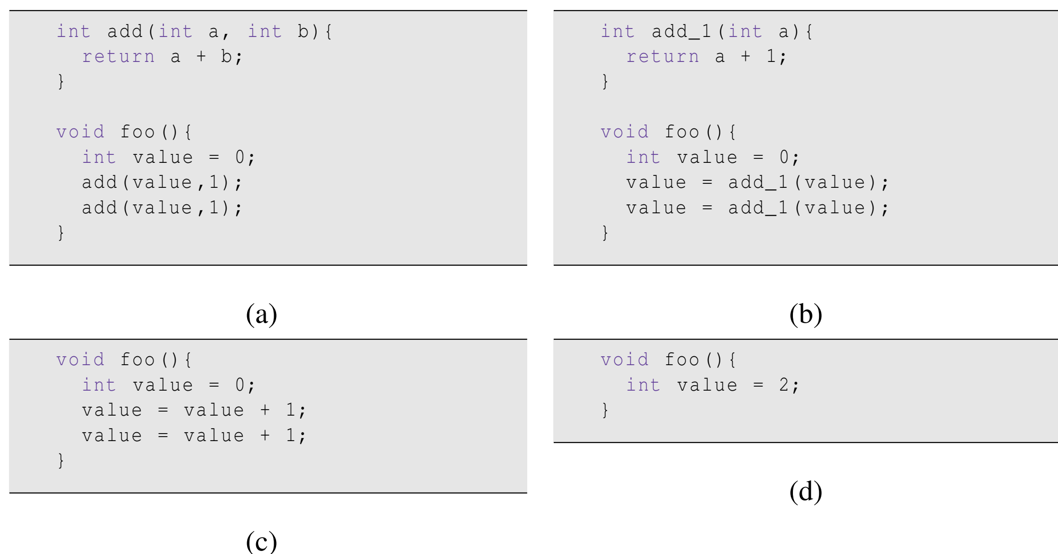


Figure 2.5: Impact of applying interprocedural optimisations. The initial program is shown in (a). The result of applying interprocedural constant propagation is shown in (b). Next `add_1` is inlined (c). Finally, constant propagation and partial evaluation are applied within `foo` to produce (d).

to optimise the function `add`. This is shown in Figure 2.5b where the `add` function is replaced with a specialisation called `add_1`. Moreover, because the compiler knows exactly what code is being called from `foo`, and that `add_1` is a small function, it is possible for the compiler to replace all the function calls with a copy of the function body – an optimisation called inlining. Figure 2.5c shows the example after inlining has been performed.

One of the most important aspects of interprocedural optimisations is that, like in the above example, they often create more opportunities for *intraprocedural* optimisation – where the focus of the optimisation shifts onto the code within each function. Figure 2.5d shows that by applying intraprocedural optimisations – such as constant propagation and partial evaluation. As a result the compiler has managed to eliminate: two function calls and two additions. Ultimately, the remaining code would be optimised away entirely as `value` is never read but this example shows how giving compilers full information about an application allows them to produce high performance machine code.

2.2.2 Assuming Closed and Open Worlds

The preceding Sections have tries to illustrate how a few fundamental programming language design decisions impact how a language is implemented. Generally, speaking it is possible to categorise programming languages into two categories: ones that assume a *closed-world* or ones that assume an *open-world*. In a closed-world it is assumed that all the information about the world can be provided to the compiler and that it is static – i.e. it never changes. Typically, assuming a closed-world requires the compiler to be provided with the source code for the whole application and any addition information it needs like the machine language to target and external libraries. Usually, it is statically compiled languages – like C, C++ and FORTRAN – that fall into the category of assuming a closed-world. By assuming a closed-world the compiler is able to precisely infer information such as locating all the callers of a particular method, find all the implementations of a single class, and determine exactly the flow of control throughout the application. Thus, enabling them to reason about the application with full knowledge to produce highly performing machine code (as can been seen in Figure 2.5). However, a direct consequence of assuming a closed-world means that these languages are not very robust to changes in their environment, for example if the underlying processor architecture or operating system changes a full re-compilation is required.

The alternative approach to assuming a closed-world is to assume an *open-world*. In this scenario, the compiler is limited in what it can assume about the world. For example, a compiler may only be given individual functions or classes to compile at a time. As such it cannot determine how many times a function is called or by who. Similarly, it also cannot assume that it knows about every type used by the application or which processor architecture it is to be executed on. These are all reasons why programming languages that assume an open-world have more complex implementations. They are generally composed of a runtime system that is responsible for resolving issues related to the open-world – like dynamically loading code, classes or invoking a Just-In-Time compiler. Examples of programming languages that fall into this category are Java [52] and Self [26]. The advantage of making an open-world assumption is that these languages are extremely portable. For instance, Java is distributed in an architecture-neutral bytecode format that is able to be efficiently converted into machine code at runtime by the Java Virtual Machine [78].

2.3 The Software Gap

So far in this Chapter, it has been highlighted that processors can be organised in many different ways and that each organisation can improve performance by exploiting various types of parallelism – data-level and thread-level parallelism. However, the problem is that there is not a single architecture capable of exploiting every type of parallelism equally well. Therefore, to improve performance, it is often best to solve the problem using the most ideally suited architecture. The issue for developers, as will be seen in Section 2.4, is that utilising a different hardware accelerator is not a transparent process as it often requires the developer to tailor an application to particular architecture. Clear examples of this are given in Sections 2.4.3 to 2.4.5 that show how the same code needs to be re-written to take advantage of each architecture.

The next Section will examine some of the programming languages and tools that can be used to program heterogeneous systems in order to demonstrate how much effort a developer expends reworking their application. One of the most interesting aspects of language design, in the context of programming heterogeneous systems, are the different approaches that can be used to enable a developer to extract the maximum performance from each device. Here there is a trade-off between providing developers with the ability to write low-level device-specific code or providing pre-written hand-optimised libraries. Depending on where a tool lies in this spectrum correlates directly with developer productivity.

For instance, the LAPACK [70], and the more recent MAGMA [120], linear algebra libraries are prime examples of commonly used libraries that allow developers to extract very high levels of performance from hardware accelerated systems by just linking against a library. Although this is highly productive, it has the downside that these libraries cannot be used to generate new functionality, and so a developer is restricted to only using the code provided by the library. Conversely, if a developer can write low-level device code they also have the ability to both write new functions for themselves and extract high-levels of performance; albeit at the cost of learning the idiosyncrasies of each hardware accelerator. The hidden cost in this situation is that it requires the developer to become intimate with the low-level details of the hardware and leaves them with a specialised application that cannot be easily ported to a radically different architecture.

2.4 Programming Heterogeneous Systems

The aim of this Section is to introduce the reader to some languages that allow us to program heterogeneous systems. It will start by looking at low-level programming languages and provide examples of how they can be used to adapt a simple example to the range of processor architectures introduced in Section 2.1. As these languages are used to program non-identical processor architectures they will be referred to as *heterogeneous programming languages* or HPLs for short.

Recall from Section 1.1.1 that programming a heterogeneous system is very similar to programming a distributed system. Well as a consequence, one of the fundamental ways in which heterogeneous programming languages differ from other languages is through their ability to coordinate the execution of an application across disparate devices. Typically, this involves support for asynchronous programming or having the ability to transfer data between devices. Additionally, they also provide the capability to generate high performance code through the exploitation of parallelism and/or vectorisation.

Nearly, all HPLs adopt a programming model where work is offloaded from a host onto an accelerator (referred to as a device); mirroring how the rendering of complex computer graphics is offloaded from a traditional processor onto a graphics accelerator. The problem with this programming model is that it is geared towards the programming of systems that have both a fixed configuration and a low-degree of heterogeneity: typically a Intel multi-core processor and a GPGPU. Tornado strives to solve this problem by adopting a task-based programming model that can be adapted to any heterogeneous system – examples of the types of complex processing pipelines that can be produced can be seen in Section 3.1.5.

Presently, the most dominant heterogeneous programming languages are CUDA [89], OpenACC [94], OpenCL [73], and OpenMP [16]. All of these languages are centred around the C programming language, with some also supporting C++ and FORTRAN. This choice has been driven out of the necessity to efficiently exploit GPGPUs from pre-existing scientific applications written in these languages. Consequently, these languages do not benefit from advances made in modern programming language implementations, such as dynamic compilation [36, 65], dynamic typing, reflection [108] or dynamic deoptimisation [64].

Heterogeneous programming languages are categorisable as either low-level or high-level depending on the abstractions that they employ. For instance, CUDA and

OpenCL are considered low-level languages as they only provide a superficial abstraction of a device. Hence, it is up to the developer to understand what that device is and the best way to program it. OpenACC and OpenMP, on the other hand, are considered to be high-level languages as they do a better job of abstracting away the concept of a device. In such languages, a developer does not need to worry about writing code for a specific device as the compiler applies the necessary code transformations for each device. In practice, the advantage here is that the developer does not need to manually parallelise their application. However, they may still need to ensure that data is correctly synchronised between devices. The disadvantage is that they still make a closed-world assumption about what device the application is going to target; leading to an application that is not robust to changes in the execution environment.

2.4.1 Low-level Heterogeneous Programming Languages

The advantage of the lower-level languages is that they provide all the tools necessary to extract maximum performance from each device. This is helped, in part, by the fact that these languages expect the developer to specialise an application for a specific device; opposed to targeting a range of devices. However, as the developer has full control over all aspects of the device the resultant code tends to be verbose.

To show what a low-level programming language looks like an example CUDA application is provided in Listings 2.2 and 2.3 that demonstrates how CUDA can be used to accelerate an element-wise addition operation, shown in Listing 2.1, using a GPGPU. The first Listing 2.2 is the code that runs on the GPGPU and the second Listing 2.3 is the code that runs on the host. Together they show how the developer partitions the application between the host and device in CUDA.

On the device-side the developer needs to use some CUDA specific syntax to define that `add` is the entry point to a kernel – via the `__global__` keyword. One of the distinguishing features of this kernel is the lack of control flow. On GPGPUs, the approach is to use large numbers of simple threads to process large amount of data. This kernel is executed in SIMT fashion by each of these threads, hence, the code only needs to process a single element of an array. For those wondering how the right number of elements are processed: this done through a combination of launching the correct number of threads in the host-side code and including an if-statement inside the kernel to detect cases where more threads than elements are launched.

Although, the device-side code is succinct the CUDA host-side code tends to be verbose. Looking through Listing 2.2 it is easy to see why: the developer has to

```
for(int i=0;i<num_elements;i++){
  c[i] = a[i] + b[i];
}
```

Listing 2.1: An example of a for-loop that is ameanable to parallelisation. This code performs the element-wise addition of two arrays.

```
__global__ void add(const int *a, const int *b,
      int *c, int array_len){
        int tid = threadIdx.x;
        if(tid < array_len){
                c[tid] = a[tid] + b[tid];
        }
}
```

Listing 2.2: An example CUDA kernel that performs element-wise addition on two arrays. Notice that there is no loop; instead, each addition is performed by a seperate thread. The kernel indexes into the input arrays using its unique thread id that is obtained with `threadIdx.x`.

manually allocate device side memory, transfer data to the device, execute the kernel, transfer the result back, and then free up the device side memory. One of the primary reasons for this verbosity is the lack of programming abstractions for supporting the programming of hardware accelerators. This lack of language support forces the developer to handle all aspects of device usage explicitly: from the parallelisation scheme to use; how memory is allocated and de-allocated; when data is transferred; and which code to execute. A direct consequence of this approach is that because there is so much for the developer to contend with the resultant application will end up very specialised. Another common issue in CUDA is that code often has to be separated according to where it is to execute: either on the host or a device. Although this is necessary, the way it is implemented in CUDA precludes the ability to run the code on the host and often leads to the duplication of code.

```
// allocate memory on the host
int *a = (int *) malloc(sizeof(int) * ARRAY_SIZE);
int *b = (int *) malloc(sizeof(int) * ARRAY_SIZE);
int *c = (int *) malloc(sizeof(int) * ARRAY_SIZE);

// select target device
cudaSetDevice(target_gpu);

// allocate memory on the GPGPU
int *dev_a,*dev_b,*dev_c;
cudaMalloc((void**) &dev_a, sizeof(int) * ARRAY_SIZE);
cudaMalloc((void**) &dev_b, sizeof(int) * ARRAY_SIZE);
cudaMalloc((void**) &dev_c, sizeof(int) * ARRAY_SIZE);

// copy the data from host to GPGPU
cudaMemcpy(dev_a, a, ARRAY_SIZE * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, ARRAY_SIZE * sizeof(int), cudaMemcpyHostToDevice);

// calculate how many threads to launch
int blocks = (ARRAY_SIZE + 1024 - 1) / 1024;
dim3 grid = dim3(blocks, 1);
dim3 threads = dim3(1024, 1);

// launch the kernel on the GPGPU
add<<<grid, threads>>>(dev_a, dev_b, dev_c, ARRAY_SIZE);

// copy back the results
cudaMemcpy(c, dev_c, ARRAY_SIZE * sizeof(int), cudaMemcpyDeviceToHost);

// free up memory on the GPGPU
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);

// free up memory on the host
free(a);
free(b);
free(c);
```

Listing 2.3: The host-side code needed to execute Listing 2.2 in CUDA. Notice how verbose the code is due to the developer having such fine-grained control over the device. For example, the developer has to manually allocate memory on both the host – via `malloc` – and the GPGPU – via `cudaMalloc`. Then schedule data transfers – via `cudaMemcpy` – and launch the kernel with the correct thread configuration – via the «`...`» operator. Finally, the result needs to be copied back and both sets of memory freed.

2.4.2 Programming SISD Architectures

By default, nearly all developers write their application with a SISD architecture in mind. As this architecture cannot exploit either data-level or thread-level parallelism, the only exploitable parallelism is instruction-level parallelism and even this is handled transparently by the hardware. Thus, the developer does not need to re-work their applications for a SISD architecture. However, the importance of programming for a SISD architecture is that it represents the simplest and, consequently, most productive way to program a computer. This productivity stems from the fact that code is easy to understand: a property that makes it simpler for developers to reason about the semantics of the code in their minds and thus making it more practical to debug. In Section 2.4.6, it will be seen that many programming languages – like OpenACC in Listing 2.11 – try to exploit the SISD representation of a program to improve productivity by automatically transforming it into a form more suitable for a different type of architecture.

2.4.3 Programming MIMD Architectures

MIMD architectures are unique as they can execute multiple independent streams of instructions (called threads). The ability to run multiple threads concurrently means that an application can have different parts of an application running in different threads simultaneously – exploiting task-level parallelism – or have multiple threads cooperatively execute the same code but with different data – exploiting data-level parallelism. It is this ability to exploit both TLP and DLP that makes MIMD architectures versatile enough to exploit coarse-grained, irregular, and non-uniform parallelism – something that other architectures cannot do.

In order to accelerate the example in Listing 2.1 on a MIMD architecture it needs to be re-written to use threads. MIMDs typically have only a small number of very capable processing elements. This means that they prefer to have work distributed across a small number of threads, unlike the CUDA example that was previously encountered. The difference is that now the number of threads used to process the array is significantly less than the size of the array. Therefore, the processing of arrays is normally divided up evenly into contiguous blocks that are assigned to threads. Listing 2.4 shows the kernel optimised for a MIMD architecture written in OpenCL. Notice that compared to Listing 2.2 there is an addition loop inside the kernel that allows each thread to process multiple elements of the array. Consequently, this kernel is considered to be

```
int my_thread_id = get_global_id(0);
int num_pes = get_global_size(0);
int blockSize = (num_elements + num_pes - 1) / num_pes;
int start = blockSize * my_thread_id;
int end = min(start + blockSize,num_elements);
for(int i=start;i<end;i++){
  c[i] = a[i] + b[i];
}
```

Listing 2.4: To optimise the for-loop in Listing 2.1 for a MIMD architecture a block-cyclic mapping is used. This example uses OpenCL to parallelise the original code so that each processing element processes a contiguous block of the input data.

coarser than the CUDA kernel.

2.4.4 Programming SIMD Architectures

As SIMD architectures apply one operation to multiple data lanes, therefore, to use them efficiently they need to exploit fine-grained data parallelism in a fixed shape (an array of items equal to the number of SIMD lanes). It is this last requirement that leads to the rigidity that makes SIMD architectures hard to utilise. To demonstrate this the remainder of this Section will go through the steps that a developer (or a compiler) has to take to accelerate Listing 2.1 with a SIMD architecture.

In its initial form Listing 2.1 is not amenable to vectorisation as it does not contain four operations that operate on consecutive elements of the array. Fortunately, this issue is easily resolved by unrolling the loop by a multiple of the vector length (as shown in Listing 2.5).

After being unrolled the loop body now contains four statements that apply the same operation to different data elements. As these operations, all process consecutive elements they can be replaced with vector equivalents as shown in Listing 2.6.

Notice that it is not just the arithmetic addition operator (+) that are vectorised, the array accesses (`array[index]`) – used to load and store each element – are also vectorised. Although this transformation looks like vectorisation is straightforward, this last example has some serious issues. Most importantly it assumes that `num_elements` is an exact multiple of the vector length. If this is false then this loop will fail to process the last few elements correctly. Therefore, to ensure that this never happens an additional loop needs to be appended to the example (as shown in Listing 2.7).

Despite being a functional implementation this example still has a very subtle, but serious, issue that can prevent it from executing on some SIMD architectures: memory alignment. As SIMD architectures are designed to operate on multiple elements of

```
for(int i=0;i<num_elements;i+=4){
  c[i] = a[i] + b[i];
  c[i+1] = a[i+1] + b[i+1];
  c[i+2] = a[i+2] + b[i+2];
  c[i+3] = a[i+3] + b[i+3];
}
```

Listing 2.5: Above shows Listing 2.1 after the loop has been unrolled four times. Notice how the induction variable of the loop, `i`, is now incremented by four and there are four operations within the loop body that process consecutive array elements.

```
for(int i=0;i<num_elements;i+=4){
  float4 vec_a = vload4(i,a);
  float4 vec_b = vload4(i,a);
  float4 vec_c = vec_a + vec_b;
  vstore4(vec_c,i,c);
}
```

Listing 2.6: Listing 2.5 re-written to use OpenCL's short vector extensions. Notice that the memory accesses are also vectorised by using the `vload` and `vstore` instructions.

data their memory operations are sometimes required to be cache aligned. In OpenCL C, as used in the examples, this is not an issue as the `vload` and `vstore` instructions do not impose any memory alignment restrictions. However, this does not mean that misaligned memory accesses do not degrade performance or that other programming models or architectures do not.

The Intel SSE instruction set, for instance, has stricter rules governing memory alignment and to accommodate them the example code would need to be modified. The change is to introduce another loop to handle the first few elements in the case that they not memory aligned – shown in Listing 2.8. Now looking at the code in this final SIMD example it is clear why SIMD architectures are considered difficult to program: what started as a single loop in Listing 2.1 is now implemented using three loops.

```
for(int i=0;i<num_elements;i+=4){
  float4 vec_a = vload4(i,a);
  float4 vec_b = vload4(i,b);
  float4 vec_c = a + b;
  vstore4(vec_c,i,c);
}

for(int i=num_elements >> 2;i<num_elements;i++){
  c[i] = a[i] + b[i];
}
```

Listing 2.7: One of the problems with Listing 2.6 is that it relies on `num_elements` being an exact multiple of the vector length (which is four in this example). To make this code more robust a second loop is added to handle irregular sizes of arrays.

```
int offset = 0;
for(;offset < num_elements && offset % alignment != 0; offset++){
  c[offset] = a[offset] + b[offset];
}

__m128* a_src = (__m128*) &a[offset];
__m128* b_src = (__m128*) &b[offset];
__m128* c_dest = (__m128*) &c[offset];
for(int i=offset;i<num_elements;i+=4){
  *c_dest = _mm_add_ps(a_src,b_src);
  a_src++;
  b_src++;
  c_dest++;
}

for(int i=num_elements >> 2;i<num_elements;i++){
  c[i] = a[i] + b[i];
}
```

Listing 2.8: An example showing how Listing 2.1 would be accelerated using Intel SSE Intrinsics. Notice how three loops are needed: one to handle memory alignment, one to perform vectorised operations, and one to handle irregular sizes of array.

2.4.5 Programming SIMT Architectures

A SIMT architecture is designed for throughput: it has large numbers of threads working on very small problems. They are like MIMD architectures as they require parallel code to extract performance, but unlike MIMD all threads are required to execute the same instructions. The result is that SIMT architectures are adept at exploiting data-parallelism within a single memory hierarchy as they amortise the cost of lengthy memory accesses by having tens of thousands of threads in-flight. Moreover, SIMT architectures are designed to be easier to utilise than SIMD architectures as they do not require code to be vectorised; instead, it is parallelised.

To convert the example in Listing 2.1 for a SIMT architecture is a matter of assiging each iteration of the for-loop to an individual thread. The transformation for doing this is straightforward: the loop-header is removed and the loop-body is modified so that the original induction variable, in this case `i`, is replaced with the unique id of each thread – resulting in Listing 2.9. This code can be executed by specifying that the processor executes the code using `num_elements` threads.

One of the limitations of SIMT architectures is that they are not well suited to executing an arbitrary number of threads. This issue stems from the fact that they are designed to execute the threads in parallel across several lanes. Therefore, to achieve good performance threads need to be scheduled in work-groups and not individually. Hence, the developer needs to specify either: the size of a work-group and how many work-groups to execute (CUDA) or the total number of threads and the work-group size (OpenCL). This creates a requirement that the total number of threads scheduled for execution is an exact multiple of the work-group size. Sometimes it is desirable to create more threads than are required as this helps when processing problems of an irregular size. Therefore, to avoid accessing invalid array indices, the application code needs to be modified – as shown in Listing 2.10.

```
__kernel void add(
    __global float* A,
    __global float* B,
    __global float* C,
    int num_elements)
{
    int my_thread_id = get_global_id(0);
    c[my_thread_id] = a[my_thread_id] + b[my_thread_id];
}
```

Listing 2.9: Transforming the for-loop in Listing 2.1 into a SIMT kernel is a matter of extracting the loop-body and replacing the induction variable with the unique id of a thread. However, this makes the assumption that it is possible to launch the same number of threads as there are elements in the arrays.

```
__kernel void add(
    __global float* A,
    __global float* B,
    __global float* C,
    int num_elements)
{
  int my_thread_id = get_global_id(0);
  if(my_thread_id < num_elements){
    c[my_thread_id] = a[my_thread_id] + b[my_thread_id];
  }
}
```

Listing 2.10: A more robust implementation that handles the situation where more threads are launched than there are elements in the array.

```
// allocate memory on the host
int *a = (int *) malloc(sizeof(int) * ARRAY_SIZE);
int *b = (int *) malloc(sizeof(int) * ARRAY_SIZE);
int *c = (int *) malloc(sizeof(int) * ARRAY_SIZE);

#pragma acc kernels
for(int i=0;i<num_elements;i++){
   c[i] = a[i] + b[i];
}

// free up memory on the host
free(a);
free(b);
free(c);
```

Listing 2.11: Listing 2.1 accelerated in a device-neutral form using OpenACC. Notice how there is no explicit device management: no memory allocations, data transfers, or kernel launches. All a developer has to do is annotate the loop with OpenACC syntax and the compiler will take care of parallelising the code for whatever architecture is required.

2.4.6 High-level Heterogeneous Programming Languages

High-level heterogeneous programming languages address a lot of the issues faced by low-level HPLs. For example, the entire CUDA application, in Listings 2.2 and 2.3, can be shortened to Listing 2.11 using OpenACC[94].

High-level HPLs aim to improve developer productivity by introducing abstractions and language support for heterogeneous programming. One of the key changes that is made is to shift the responsibility of applying the code transforms – like in Sections 2.4.3 to 2.4.5 – into the compiler. By doing this it makes it unnecessary to for a developer to change the code to target a SIMD, SIMT or MIMD device. What aids productivity further is that these languages transparently handle offloading execution on to the device. Typically, no explicit data transfers, memory allocations, or kernel launches are required. Therefore, using a high-level HPL an application can now be written in a device-agnostic manner. However, the problem that most of these languages have is that they are based on languages that make closed-world assumptions about the target device. Which in turn means that although the source code is device-agnostic, it still needs to be explicitly compiled for each different architecture before it can be used. Creating a limitation that the final application will not be robust against changes in the execution environment. This is one of the primary problems that Tornado seeks to solve by introducing: a virtualisation layer – the Tornado Virtual Machine (described in Chapter 4); a runtime system – the Tornado Runtime System (described in Chapter 5); and a parallelising JIT compiler (described in Chapter 6).

2.4.7 Emerging Heterogeneous Programming Languages

Currently, most of the focus of heterogeneous programming languages is on improving the four established and community-driven languages: CUDA, OpenACC, OpenCL, and OpenMP. However, there is a significant amount of prior work that exists in the research community that focuses on using heterogeneous hardware from dynamic languages, typically these aim to exploit GPGPUs, but other attempts have also been made to target FPGAs, vector processors, and multi-core processors. Languages that have been targeted are Java [1, 7, 37, 46, 58, 98, 124, 125, 127], Python [12, 22, 76, 102], Haskell [25, 62, 86], Scala [23, 93], MATLAB [12, 34], JavaScript [63] and Lua [34].

Until Tornado there were no JVM-based solutions capable of both competing with CUDA and OpenCL in terms of performance and the capability to create complex multi-stage processing pipelines. (This will be demonstrated later in Section 7.6). The primary reason for this is that the majority of prior art focuses on exposing an underlying low-level heterogeneous programming language in to a dynamic language. Examples of this problem are seen in Section 2.4.8 where both APARAPI [1] and Rootbeer [98] expose low-level HPL features to Java. Typically, this is done for expediency and to avoid writing a parallelising compiler and has the effect of constraining the new language to also being a low-level HPL. Tornado does not take this approach and implements a parallelising JIT compiler (see Section 3.2). As a direct result Tornado is able to differentiate between these languages by becoming a high-level HPL based on a more dynamic language.

2.4.8 Examples of Prior At

In order to highlight some of the differences between Tornado an the prior art this Section provides code examples from APARAPI [1] and Rootbeer [98]. APARAPI was a precursor to the now defunct OpenJDK Project Sumatra[95] and, like Tornado, is built on top of OpenCL. An APARAPI implementation that accelerates the earlier example from Listing 2.1 is shown in Listing 2.12 with a Rootbeer implementation split across Listings 2.13 and 2.14. The major difference between these Listings and Tornado is that both expose a low-level programming model to the developer. Contrast this with the Tornado API discussed in Section 3.2 and it is clear that Tornado is more capable. as neither of these frameworks have attempted to tackle the execution of real-world applications such as Kinect Fusion (see Section 7.3), what really distinguishes Tornado is its ability to compose complex processing pipelines (see Section 3.1.5).

```
final float[] a = new float[size];
final float[] b = new float[size];
final float[] c = new float[size];

Kernel kernel = new Kernel(){
      @Override
      public void run() {
            int gid = getGlobalId();
            c[gid] = a[gid] + b[gid];
      }
};
```

Listing 2.12: Code to accelerate Listing 2.1 in APARAPI. Note how the OpenCL C language is exposed to the developer and that the developer has to implement a new class for each kernel.

```
public class VectorAddKernel implements Kernel {

  private int[] a;
  private int[] b;
  private int[] c;
  private int index;

  public VectorAddKernel(int[] a, int[] b, int[] c, int index){
    this.a = a;
    this.b = b;
    this.c = c;
    this.index = index;
  }

  public void gpuMethod(){
  c[index] = a[index] + b[index];
  }
}
```

Listing 2.13: Device-side code to accelerate Listing 2.1 in Rootbeer. Like APARAPI the developer is expected to implement a new class for each kernel that needs to be executed on the GPGPU.

```
int[] a = new int[length];
int[] b = new int[length];
int[] c = new int[length];

List<Kernel> tasks = new ArrayList<Kernel>();
  for(int index = 0; index < a.length; ++index){
    tasks.add(new VectorAddKernel(a,b,c, index));
}

Rootbeer rootbeer = new Rootbeer();
rootbeer.run(tasks);
```

Listing 2.14: Rootbeer also split its application into host and device side code. Notice how the developer has to manually create a new task for each device-side thread.

2.4.9 Common Themes

As shown in Table 2.1, there is a plethora of prior art focusing on improving the state-of-the-art in heterogeneous programming languages. Naturally, there are some fundamental similarities and over-arching themes that run through both the prior art and this work. For instance, all work aims at exploiting array-oriented data-parallelism for numerical applications. Another theme is that many projects adopt a functional programming style. The advantage this brings is the ability to express code in an implicitly parallel form that can be mapped easily onto highly threaded architectures like GPGPUs. These techniques stem from work such as: NESL [14], HiDP [87] and Data Parallel Haskell [24].

Using a functional programming approach provides developers with a set of higher-order functions that can be used to compose larger applications. Some of the most common functions are *map*, *reduce*, *combine*, *scatter*, *gather*, *scan* and *filter*. Each one of these operations has well-defined semantics that permits them to be implemented in parallel. The approach taken is to allow expert programmers to implement these operators for each specific device and have the programming framework select the implementation based on the device the application is targeting. The downside of this approach is the same as the library approach mentioned earlier in Section 2.3 – that these operators are highly useful when they align with the needs of the code but they do not provide a mechanism for generating new operators. Hence, developers are restricted to using a specific set of hardware accelerated operators.

Despite the commonality of high-level programming models, the prior art differs in many ways, but the three most important ways are:

Developer Exposure: this is how a developer is expected to interact with heterogeneous hardware. This interaction may be direct, such as using a device-specific APIs to manage or generate code for a heterogeneous device – PyOpenCL/PyCUDA [76], indirectly via calls to a domain-specific library e.g. Torch 7 [34], or even extending the original language, e.g. Habanero-Java [58].

Code Generation: by definition, programming heterogeneous hardware involves generating machine code for multiple devices. To solve this problem a number of techniques can be used: Just-In-Time (JIT) compilation, e.g. APARAPI [1]; Ahead-of-Time (AOT) compilation, e.g. Rootbeer [98]; or providing a library of pre-existing operations to avoid code generation. e .g. ViperVM [62].

```
#pragma acc parallel loop
for(int i=0;i<n;i++){
  foo(i)
}
```

Listing 2.15: The following OpenACC code is only compilable for hardware accelerators if the source of `foo` is available to the compiler.

Parallelism: one of the most common problems developers face is writing highly-parallel code to target a heterogeneous device. Many different techniques are used to make this possible: exposing low-level CUDA/OpenCL API calls to the developer, e.g. JOCL [127]; directive driven parallelisation, e.g. JCUDAMP [37]; or exploiting the implicit parallelism in functional-style operators, e.g. River Trail [63].

2.5 Motivating Examples

After introducing a range of different heterogeneous programming languages and their benefits, this Section aims to provide some motivating examples of the types of programming language issue that is solved by Tornado.

2.5.1 Issues Due To Language Design

There are some fundamental issues with HPLs like OpenCL that stem from them making closed-world assumptions. Firstly, it is difficult for developers to use libraries or reuse existing code. This problem is shown in Listing 2.15 where the code is only compilable if the source of `foo` is available to the compiler. This is an example of where Tornado is more capable than OpenACC: as it compiles from Java bytecode it is able to compile methods that exist in third-party libraries.

2.5.2 Inability To Express Coordination

One of the most critical omissions for heterogeneous programming languages is that they often have no abstractions to help express what code should run where. Most languages either require a developer to schedule device-side code on a device by appending it to a different queue – CUDA and OpenCL – or by changing a global environment variable – OpenACC and OpenMP. Listing 2.16 is an example of how this is achieved in OpenACC.

Language		Hosting Language	Compiler	Devices
CUDA	[89]	C/C++	AOT	NVIDIA GPGPUs
OpenACC	[94]	C/C++/FORTRAN	AOT	Multi-core and Many-core
OpenMP	[15]	C/C++/FORTRAN	AOT	Multi-core, Many-core
OpenCL	[73]	C/C++	AOT	Multi-core, Many-core, FPGA, DSP
Apricot	[99]	C/C++/FORTRAN	AOT	Intel MIC
River Trail	[63]	Javascript	JIT	Multi-core, Many-core
SkelCL	[114]	C++	AOT	Multi-core, Many-core
Lime	[7, 38]	Lime	JIT	CPU, GPU, FPGA
Firepile	[93]	Scala	JIT	OpenCL GPGPUs
FastR-OCL	[47]	R	JIT	OpenCL GPGPUs
Accelerate	[25]	Haskell	JIT	GPGPUs
Nikola	[86]	Haskell	JIT	GPGPUs
Torch7	[34]	Lua	?	GPGPUs
Theano	[12]	Python	JIT	NVIDIA GPGPUs
Copperhead	[22]	Python	JIT	NVIDIA GPGPUs
Parakeet	[102]	Python	JIT	NVIDIA GPGPUs
PyGPU	[82]	Python	JIT	NVIDIA GPGPUs
Dandelion	[101]	.NET	JIT	CPUs, GPGPUs, FPGAs
MaJIC	[5, 49]	MATLAB	JIT	GPGPUs
Chesnut	[115]	-	AOT	NVIDIA GPGPUs
NOVA	[33]	-	-	NVIDIA GPGPUs
JCUDA	[124]	Java	AOT	NVIDIA GPGPUs
JCUDAMP	[37]	Java	AOT/JIT	NVIDIA GPGPUs
JOCL	[127]	Java	JIT	OpenCL compatible
JaBEE	[125]	Java	JIT	NVIDIA GPGPUs
Habanero-Java	[54, 58]	Java	AOT	OpenCL compatible
RootBeer	[98]	Java	AOT	NVIDIA GPGPUs
APARAPI	[1]	Java	JIT	Multi-core, Many-core
IBM J9	[66, 68]	Java	JIT	NVIDIA GPGPUs
Sumatra	[95]	Java	JIT	NVIDIA GPGPUs, HSA Compatible
Jacc	[28]	Java	JIT	NVIDIA GPGPUs
Tornado	[77]	Java	JIT	OpenCL compatible
ViperVM	[62]	JIT	Haskell	CUDA/OpenCL compatible
Velocoraptor	[50]	JIT	VRIR	OpenCL compatible
Polly-Acc	[53]	AOT	LLVM	CUDA/OpenCL compatible
Delite	[17]	AOT	-	CPUs or GPGPUs

Table 2.1: A summary of prominent heterogeneous programming languages. The first group of languages are state-of-the-art and are of industrial quality. The second group of languages are non-JVM based. The third group are the JVM-based languages, of which Tornado and its predecessor Jacc are members. The final group are projects that are not languages in their own right – e.g. compiler optimisations or new forms of intermediate representations that can be used to generate code for hardware accelerators – but share the aim of enabling heterogeneous programming. Key: Compiler – represents the type of compiler used: either Ahead-of-Time (AOT) or Just-In-Time (JIT).

```
acc_set_device_type(device_type);
acc_set_device_num(device_id);

#pragma acc parallel loop
for(int i=0;i<n;i++){
  c[i] = a[i] + b[i];
}
```

Listing 2.16: The following OpenACC example shows how kernels are migrated between devices.

This approach suffers from two problems: (1) that there is little support for choosing which device to use, and (2) the decision of where the code is going to execute is decoupled from the programming language. With regards to the first issue, this makes it difficult for a developer to express a selection preference like: *I want to use the GPGPU with the most processing elements* or *I want to schedule this code on the device where variable X resides*. This is a problem solved in Tornado by using dynamic configuration. The examples in Section 3.3.4 show how the Tornado API can be used to programmatically adapt a complex processing pipeline so that each stage can be freely mapped onto different devices.

Moreover, the loose coupling of the coordination logic with a HPL leaves most languages unsuited to writing complex multi-device multi-kernel applications. A first problem is the composition of concise multi-kernel codes – Listing 2.17 shows some OpenACC code that has been re-factored to allow code reuse and back-to-back kernel executions. Despite there being clear scope for optimisation of this code, the structure of the language means that the compiler cannot optimise the data movement between kernels. The problem is that the coordination logic and the compute logic are dispersed through the source code. Therefore, when the compiler starts to compile the `mult` function it is unsure about what happens *before* and *after* the kernel is executed on the remote device. Therefore, it will be conservative and generate data transfers into and out of the device for every variable. Although this is not problematic in terms of correctness this application will severely under perform because of the extra data movement that is needed. As will be seen later in Section 7.6 the cost of moving large amounts of data between devices can take longer than executing the code itself. Hence, developers that wish to write complex codes are required to manually optimise the coordination of data between devices – reducing their productivity. The salient point here is that optimisation of data-movement needs to be performed in the coordination logic rather than from the perspective of the compute logic. Solving this issue is a

```
#pragma acc routine
void mult(int n, float *a, float *b, float *c);

#pragma acc routine
void add(int n, float *a, float *b, float *c);

void mult(int n, float *a, float *b, float *c){
#pragma acc loop
  for(int i=0;i<n;i++){
    c[i] = a[i] * b[i];
  }
}

void add(int n, float *a, float *b, float *c){
#pragma acc loop
  for(int i=0;i<n;i++){
    c[i] = a[i] + b[i];
  }
}

#pragma acc kernels
{
  mult(n,a,b,c);
  add(n,c,d,e);
}
```

Listing 2.17: Back-to-back kernel execution in OpenACC. In this example the compiler is unable to optimise the data movement between kernels and so always generates data transfers to copy data to the device and off the device before and after a kernel is executed.

key aspect of Tornado: the Tornado API provides abstractions that allow complex processing pipelines to be constructed (see Section 3.1.5) and the Tornado Runtime System is able to use these abstractions to minimise the amount of data transferred between devices (see Section 5.3).

2.6 Summary

Initially, this Chapter looks at the range of different processor architectures that are appearing in modern computers (Section 2.1). For instance, it introduced the SPSD, MIMD, SIMD, and SIMT architectures and described how a simple a example needs to be adapted for each one (see Sections 2.4.2 to 2.4.5). The important point is that the same program needs to be adapted differently depending on which one of these architectures that it is to use.

In order to make use of these different hardware architectures a developer needs to be able to program them. Section 2.3 discusses this gap that needs to be bridged by software: that programming languages need to support the ability to target each of

these different types of architectures. However, a programming languages ability to do this is often restricted because of the closed-world assumptions that they make (see Section 2.2.2).

To understand the problem further, the concept of a heterogeneous programming language is introduced and the differences between low-level and high-level heterogeneous programming languages are discussed in Section 2.4.

Now to place this thesis in context, a brief survey of prior art is provided in Section 2.4.9 to highlight the common themes that are occurring in the field. Additionally, some examples from relevant prior art is discussed in Section 2.4.8.

Finally, the Chapter concludes with some motivating examples of where the prior art fails and where Tornado is able to address these shortcomings in Section 2.5. In the next Chapter the focus will shift onto the design and implementation of the Tornado API and how it addresses a lot of these shortcomings.

3 | Tornado Application Programming Interface

The role of this Chapter is to describe the developer-facing interface of the Tornado heterogeneous programming framework and highlight how it has been designed to address the shortcomings of prior art in three ways. First, that it enables the composition of complication processing pipelines from multiple kernels (or tasks) that run on multiple devices. Second, that device-side code in a device-neutral manner that is portable across a wide range of devices. Third, that by integrating Tornado closely with Java allows developers to write code using a more productive language.

3.1 Programming Model

One of the most challenging aspects of creating Tornado was in developing a programming interface that allows an application to be portable across a wide range of system topologies and device architectures. A task that is made more difficult due to the knowledge that to exploit each system or device adequately both the compiler and runtime system require system-specific knowledge. As these last two goals seemingly conflict, a clear question emerges about how best to design a programming framework that is system-neutral but also allows the application to exploit system-specific information.

A clear example of how these goals are difficult to satisfy is seen in the design of OpenACC: a heterogeneous programming language that many developers already consider to be system-neutral. The problem with OpenACC stems from fact that there is a closed-world assumption being made with respect to the hardware accelerator(s) that it is targeting. At compile-time the compiler expects that the developer has provided it full knowledge of what hardware the application is to execute on. So although

the application can be written in a system-neutral form once it is compiled the compiler embeds system specific information into the generated machine code. As a result of this specialisation the application is not robust to changes in its execution environment, like running on a different system or having hardware dynamically added into the system. To a certain extent, this situation can be remedied by having the compiler speculate on what devices the application might encounter to produce several versions of the same application. However, this approach is not scalable as it is often a range of configuration options that need to be tuned on a per device basis. For instance, OpenACC clauses such as `device`, `async`, `worker`, `gang`, and `tile` are highly context specific. Hence, this became the motivation for Tornado to support *dynamic configuration*: where each of these parameters can be tuned at runtime without the need to re-compile the application each time it is executed in a different environment. Now these parameters can be specified: on a per system basis in a configuration file, on a per execution basis using command line parameters or programmatically by the application itself. Something that is covered in more detail later in Section 3.3 with the benefits being evaluated in Section 7.8.

An important point to note about Tornado is that it is novel because the focus is not on generating high-performance low-level code for specific hardware accelerators but on the development of a framework that allows an application to dynamically adapt (or optimise) itself to use a particular hardware accelerator. Therefore, the majority of innovation in Tornado surrounds development of a programming model that supports: transparent data movement, asynchronous execution, dynamic compilation and automatic memory management. Now what makes Tornado stand out from the four dominant heterogeneous programming languages – CUDA, OpenACC, OpenCL and OpenMP – is that it is designed to be system-neutral. This means that a Tornado application is not critically dependent on a particular heterogeneous system design, operating system or type of hardware. Remember that this is the overall aim of the thesis stated in Section 1.4 and that the ability to achieve this goal will be evaluated later on in Chapter 7. Where it be tested by trying to implement a real-world application once and execute it across thirteen different hardware accelerators: comprising of multi-core processors, GPGPUs, and a discreet many-core accelerator.

Tornado does not follow this conventional wisdom. Instead, it has a programming API that allows developers to express the computations – to execute on hardware accelerators – and how they are connected. By knowing information about both the computation and the relationships between successive computations, Tornado is able

to minimise the cost of data movement within a heterogeneous system. The key principle is that Tornado avoids making system-specific choices ahead-of-time; instead, decisions are delayed until runtime when the exact system configuration is known. Thereby eliminating the need for either the developer or the compiler to speculate. If system specific information needs to be provided for tuning, the developer is free to provide the runtime system with information either from the command line, a configuration file or dynamically generate it from within the application itself. Tornado aims to add value to developers by enabling them to quickly, and efficiently create heterogeneous applications.

3.1.1 Tasks

In Tornado a task represents the smallest unit of computation that is schedulable on a device. It provides an abstraction that separates the execution of the computation away from the computation itself. Formally, there are three parts to a task: the code to be executed, the parameters to invoke the code with, and some meta-data that is specified by the developer or collected dynamically by the runtime system. However, it might help to think of as tasks as being a schedulable method invocation akin to a continuation [59].

The reason for using tasks is that they provide an abstraction that allows extra properties – such as locality – to be attached to the program state which is required invoke a method correctly. Or more formally, they capture the minimum possible subset of the running application that is needed to execute the computation on a remote device. What is special about Tornado is that by because it is implemented using Java it is possible to use reflection to retrieve the bytecode associated with each method. This allows Tornado to work with existing applications – unlike the OpenACC example in Section 2.5.1 – and re-compile the application on demand.

A major feature of Tornado is that Tasks are compiled dynamically each time a request is made to execute a task on a different device or if the task is dynamically re-configured – for example to change the parallelisation scheme. This ability of Tornado is not generally unique as it is a common technique used by many dynamic languages, however, it is unique amongst the established heterogeneous programming languages – CUDA, OpenACC, OpenCL and OpenMP – and the majority of emerging languages mentioned in 2.1. It should be noted that invoking a JIT compiler at runtime – as is the case with OpenCL – is not the same as dynamic compilation in Tornado. The key differences are: that dynamic compilation is transparent to the developer unlike OpenCL

where an explicit API call to the compiler is needed; Tornado is able to perform interprocedural optimisations between the host-side and device-side code; and Tornado is able to dynamically change the parallelisation scheme used in the device-side code opposed to OpenCL that requires the developer to do this.

One of the more pragmatic and, perhaps, commercial problems with the mainstay of heterogeneous programming languages – CUDA, OpenACC, OpenCL and OpenACC – is the expectation that applications are fine to be distributed as source code. In these languages this is required as the end-user is likely required to re-compile the application each time they hardware changes. Tornado avoids this problem as it works directly with Java bytecode and not the applications source code.

3.1.2 Task Schedules

It is rare for real-world code to centre around the acceleration of a single task, what is much more likely is that multiple tasks will need accelerating for the application to become viable. Therefore Tornado introduces a novel language feature – *task-schedules* – to tackle this problem.

Normally a task-schedule represents all the tasks that span a computationally critical part of an application and as such should be thought of as a collection of tasks (or continuations) that are executed atomically. From the developers perspective task-schedules provide them with the ability to define the critical parts of their application and have Tornado increase their performance by transparently using hardware accelerators to execute the tasks. Thus, alleviating the need for the developer to explicitly: coordinate data movement, perform any device management activities, or parallelise their code. However, from an implementation perspective task-schedules have two roles: first they capture the flow of control and data through its constituent tasks and secondly, they define how to treat variables before and after their execution. This information is then used by the Tornado Runtime System to analytically determine the most optimal way of executing the task-schedule for the current system. A discussion can be found later in Section 4.6 that describes the the metrics used to drive optimisation in Tornado.

3.1.3 Informal Specification

Before introducing the Java specific Tornado API in the next Section an informal specification will be is introduced to highlight the semantics of a task-schedule in a language

```
schedule <identifier>
  [volatile <variable0>,<variable1>,...,<variableN>]
  {
    <task0>,
    <task1>,
    ...,
    <taskN>
  }
  [sync <variable0>,<variable1>,...,<variableN>]
```

Listing 3.1: Informal specification of a task-schedule.

neutral way. As described in the previous section, a Tornado application needs to be defined in terms of tasks – the smallest unit of work that can be scheduled on a device – and task-schedules – a collection of tasks. Listings 3.1 and 3.2 show the syntax used to define task-schedules and tasks respectively. An explanation of the keywords and their meaning is provided below:

schedule defines a new task-schedule associated with a unique identifier that can be used by the developer to reference either the task-schedule indirectly or access its meta-data.

volatile specifies an optional list of variables that change between subsequent task-schedules executions. Hence, they need to be synchronised – copied from the host-side into device-side memory – before they are first used within the task-schedule.

sync specifies an optional list of variables that are used by other code outside the task-schedule. These variables need to be sychronised – copied from device-side memory into host-side memory – before the task-schedule is considered to be executed.

task defines a new task within a task-schedule with a unique identifier. It also contains a reference to some code and the parameters that should be provided to it.

An example of a task-schedule is shown in Listing 3.3 that defines a task-schedule `schedule0` that executes a single task `task0`. When `task0` is executed it will call method `foo` and pass in parameter `a`. Once a task-schedule has been defined it can be executed in the application either synchronously or asynchronously (as demonstrated in Listing 3.4).

```
task <identifier> <method>(<parameter0>, <parameter1>,...,<parameterN>)
```

Listing 3.2: Informal specification of a task.

```
schedule schedule0 {
  task task0 foo(a);
}
```

Listing 3.3: An example task-schedule that executes a single task.

3.1.3.1 Termination Criteria

A task-schedule terminates once all tasks have been successfully executed and any variables included in the `sync` list have been updated in host-side memory. From the developers perspective, the task schedule executes the tasks contained within the its body in program order – i.e. it preserves sequential consistency. If synchronous execution is used then the task-schedule will block until the termination criteria are met. However, if the task-schedule is executed asynchronously using `submit` the task-schedule will return immediately and the task-schedule will be executed in the background. To determine whether the task-schedule has completed a developer can use the `wait` method that will block until the criteria is met. In the event that an exception occurs during the execution of a task-schedule Tornado will try to propagate the exception back to the application code.

3.1.3.2 Properties

By expressing the heterogeneous code using the tasks and task-schedules abstractions organises the coordination logic into a hierarchical structure. Now as each task and task-schedule has a unique identifier they can be addressed directly inside this structure. This feature allows properties (or the meta-data) to be defined at both

```
// synchronous
schedule0.execute();

// asynchronous execution
schedule0.submit();
...
schedule0.wait();
```

Listing 3.4: An example to show how the task-schedule from Listing 3.3 is executed either synchronously or asynchronously.

```
schedule0.device = <value>;
schedule0.task0.device = <value>;
```

Listing 3.5: As each element of a task-schedule is uniquely identifiable it is possible to update its meta-data as shown. Here the meta-data of the task-schedule and task defined in Listing 3.3 is updated. The first line assigns a device to the task-schedule – so all tasks inside it will be also assigned to this device. In the second line a single task is updated to execute on a specific device (overriding any devices specified in the task-schedules meta-data).

the task and task-schedule granularity and addressed using the dot notation: *task-schedule*[*.task*]*.property*. Hence, it is possible for a developer to query or modify the properties associated with different instances of work. One of the most typical uses for properties is to define which device the task-schedule or task should use (shown in Listing 3.5). Hence, properties are fundamental in providing Tornado the ability to dynamically configure applications (see Section 3.3).

3.1.3.3 Variable Scopes

In the informal specification a the body of a task-schedule is enclosed by parenthesis that define the optimisation scope of Tornado. Any variable that is captured in this lexical scope is by default cached on the device-side. However, there are two situations where a variable should not be cached: (1) when a variable is modified before a task-schedule is executed, and (2) when a variable is used after being modified by a task-schedule. Both of these situations occur because of the existence of read-after-write data dependencies that span the host-side and device-side. At present Tornado is unable to automatically identify these types of data-dependencies, and so the developer is responsible for managing them. The resolution is to place a variable within either the `volatile` or `sync` statements attached to the task-schedule. The `volatile` statement that has the effect of invalidating all cached versions of a variable before the task-schedule is executed – this way it will be transferred from the host-side the first time it is used on a device. Whereas the `sync` statement has the effect of transferring the latest version of the specified variables back to host-side memory at the end of the task-schedule. Hence, any variable that is modified on the host-side before a task-schedule is executed should be placed in the `volatile` to ensure a clean copy is transferred to the device on every invocation of the task-schedule. Similarly, any variable that is updated on the device and then used on the host afterwards should be put in the `sync` statement to ensure the latest version is written back to the host. Figure 3.1

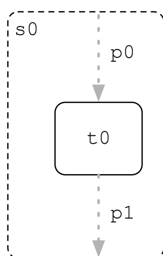
shows the semantic differences between task-schedules that use the `volatile` and `sync` statements.

One of the main sources of performance degradation that is common across the majority of heterogeneous programming languages is the cost of transferring data between devices. A problem that will be encountered later in Section Section 7.6. One of the key features of Tornado is its ability to optimise the amount of data that moves between devices (see Sections 5.3 and 5.4). However, this optimisation is only possible due to the design of task-schedules and is explained in Section 5.3.

3.1.4 Java Implementation

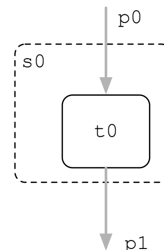
The reason for having two ways for specifying task-schedules is due to the idiosyncrasies introduced when implementing the API in Java. For instance, the Java implementation of the API is implemented as a user-space library opposed to being integrated into the language itself. By doing this Tornado can work on a stock Java Virtual Machine but sacrifices some of the expressiveness available in the informal specification. For the most part, the code is similar, however, it is easier to understand the semantics of task-schedules when they are expressed using the informal specification.

Figure 3.2 compares the differences of the informal specification and the Java implementation. The major differences between the two are that in the Java implementation it is not possible to use parenthesis to capture the task-schedule and the way properties are accessed.



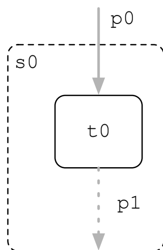

```
schedule s0 {


  task t0 method(p0, p1);

}
```

(a) Default Tornado behaviour.



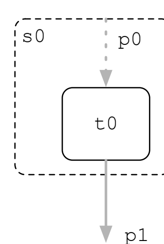

```
schedule s0
  volatile(p0)
  {
    task t0 method(p0, p1);
  }
  sync(p1)
```

(b) Bidirectional streaming pipeline.




```
schedule s0
  volatile(p0)
  {
    task t0 method(p0, p1);
  }
```

(c) Unidirectional streaming pipeline (in).




```
schedule s0
  {
    task t0 method(p0, p1);
  }
  sync(p1)
```

(d) Unidirectional streaming pipeline (out).

Figure 3.1: The optimisation scope of Tornado is defined by the parenthesis that enclose the body pf the task-schedule (represented by the dashed black line). Grey arrows depict data transfers and show the flow of data into and out of tasks. A dotted arrow indicates that a data transfer is only performed if the variable is not already present (or cached) on the device. Figures (a) to (d) show simple examples of how data flows into and out of task-schedules when various combinations of `volatile` and `sync` are used.

```
schedule s0
  volatile (p1)
  {
    task t0 Class.method(
    p0, p1, ..., pn);
  }
  sync (p1);

s0.device = value;
s0.t0.device = value;

s0.execute();

s0.submit();
s0.wait();
```

(a) Informal specification

```
s0 = new TaskSchedule("s0")
  .volatile(p0)

  .task("t0", Class::method,
        p0, p1, ..., pn)

  .sync(p1);

s0.mapAllTo(device);
s0.getTask("t0").mapTo(device);

s0.execute();

s0.submit();
s0.waitOn();
```

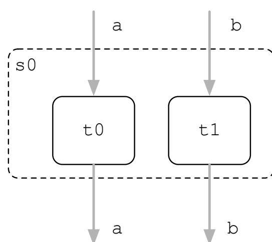
(b) Java implementation

Figure 3.2: Side-by-side comparison of the idiosyncrasies between the informal specification and Java implementation.

3.1.5 Composing Complex Pipelines

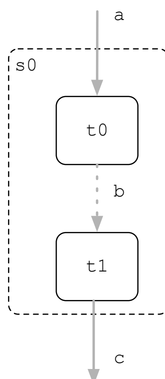
A use case that developers encounter when writing real-world application is implementing a multi-stage processing pipeline that contains many tasks. Our example application, Kinect Fusion (described in Section 7.3), has twelve distinct kernels that are invoked between 18 and 54 times on each invocation of the pipeline. It is this kind of application that is difficult to port between hardware accelerators because the porting might require these kernels to be either ported to another heterogeneous programming language or re-cast to target a different device. Tornado's strength lies in its ability to make this process transparent to the developer. By using task-schedules a developer can capture complex organisations of tasks succinctly and this Section provides some examples of how complex pipelines are composed using Tornado.

In each of the following examples notice how Tornado can identify and optimise data-flow between different tasks. Other than ensuring that variables are included in the appropriate `volatile` or `sync` parameter list, data movement is transparent to the developer.



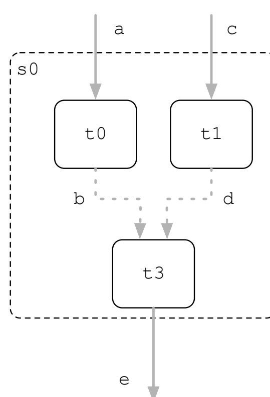

```
new TaskSchedule("s0")
  .streamIn(a)
  .task("t0", Example::op1, a)
  .task("t1", Example::op2, b)
  .streamOut(b);
```

Figure 3.3: Co-scheduling of independent tasks. In this situation Tornado is able to determine that there is no data-dependence between tasks `t0` and `t1` and so will try and overlap their execution. This is useful if these tasks are assigned for execution on different devices.



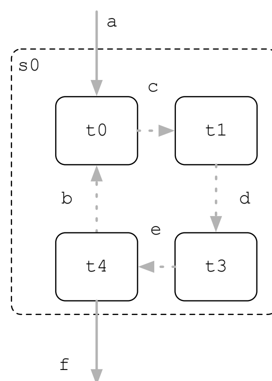

```
new TaskSchedule("s0")
  .streamIn(a)
  .task("t0", Example::op1, a, b)
  .task("t1", Example::op2, b, c)
  .streamOut(c);
```

Figure 3.4: Co-scheduling of data-dependent tasks. In this example two tasks are called back-to-back and are data-dependent on `b`. In this situation Tornado is able to recognise that both these kernels are in the same optimisation scope (or task-schedule) and there is no need to synchronise `b` with the host in between executing these tasks. This optimisation is useful for multi-stage pipelines as it avoids the need to perform any data-transfers between each of the stages.



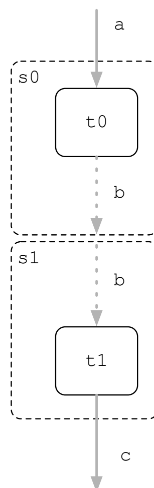

```
new TaskSchedule("s0")
  .streamIn(a, c)
  .task("t0", Example::op1, a, b)
  .task("t1", Example::op2, c, d);
  .task("t2", Example::op3, b, d, e)
  .streamout(e);
```

Figure 3.5: Co-scheduling of both independent and data-dependent tasks. In this example tasks `t0` and `t1` produce two intermediate results that are immediately consumed by task `t3`. As Tornado is able to again recognise that these intermediate results only exist within the optimisation scope it is able to eliminate any data transfers of `b` and `d`. However, what Tornado is also able to realise is that as there is no data-dependency between `t0` and `t1` it is possible to overlap their execution.




```
new TaskSchedule("s0")
  .streamIn(a)
  .task("t0", Example::op1, a, b, c)
  .task("t1", Example::op2, c, d);
  .task("t3", Example::op3, d, e)
  .task("t4", Example::op4, e, b, f)
  .streamOut(f);
```

Figure 3.6: Data-dependent tasks with a feedback loop. This example shows a complex pipeline configuration that has a feedback loop, i.e it has a data-dependency that spans successive pipeline executions. In this example, Tornado will allocate `b` to be a device-side variable and avoid ever transferring it to the host. However, Tornado is designed so that the lifetime of device-side variables is tied to the lifetime of the task that uses them; hence, in this case as long as `t4` lives `b` will also.




```
new TaskSchedule("s0")
  .streamIn(a)
  .task("t0", Example::op1, a, b)

new TaskSchedule("s1")
  .task("t1", Example::op2, b, c)
  .streamIn(c)
```

Figure 3.7: Passing data between different task-schedules. This final example illustrates how Tornado optimises data movement between different task-schedules. In the case task-schedule `s0` produces a value, `b`, that is consumed by `s1`. By default variables are cached on the device something that is advantageous in the situation. As the developer has not specified in `s0` to `sync` variable `b` or specified that `b` is `volatile` in `s1` then `b` is left on the device in between executing both task-schedules. Consequently, Tornado is able to automatically detect that these task-schedules use the same variable `b` and that it is already present on the device. Therefore, Tornado will not generate any data transfers.

3.1.5.1 Real-world Example From Kinect Fusion

Listing 3.6 shows an example taken directly from the Kinect Fusion application (see Section 7.3). It highlights how task-schedules are used in real-world applications. In these twenty lines, the first three stages of the Kinect Fusion pipeline is defined that accounts for between 12 and 30 kernel invocations of six distinct kernels each time the pipeline is run. A visualisation of the pipeline is shown in Figure 3.8.

```
pp = new TaskSchedule("pp")
  .volatile(depthImageInput)
  .task("mm2meters", ImagingOps::mm2metersKernel, scaled_image, input_image,...)
  .task("bFilter", ImagingOps::bilateralFilter, pDepth[0], scaled_image, ...);

ePose = new TaskSchedule("estimatePose");
for (int i = 1; i < iterations; i++) {
  ePose.task("resizeImage" + i,
            ImagingOps::resize,
            pDepth[i],
            pDepth[i - 1],
            ...);
}

for (int i = 0; i < iterations; i++) {
  ePose.task("d2v" + i, GraphicsMath::depth2vertex, pVertex[i], pDepth[i], ...)
       .task("v2n" + i, GraphicsMath::vertex2normal, pNormal[i], pVertex[i]);
}

for (int i = 0; i < iterations; i++) {
  icp[i] = new TaskSchedule("icp" + i)
    .volatile(pPose)
    .task("track" + i, ICP::track, result[i], pVertex[i], pNormal[i], pPose);
}
```

Listing 3.6: Example code taken from the Tornado implementation of Kinect Fusion (see Section 7.3). This code creates three task-schedules – `pp`, `ePose`, and `icp` – that between them accounts for between 12 and 30 task executions each time the Kinect Fusion pipeline is called.

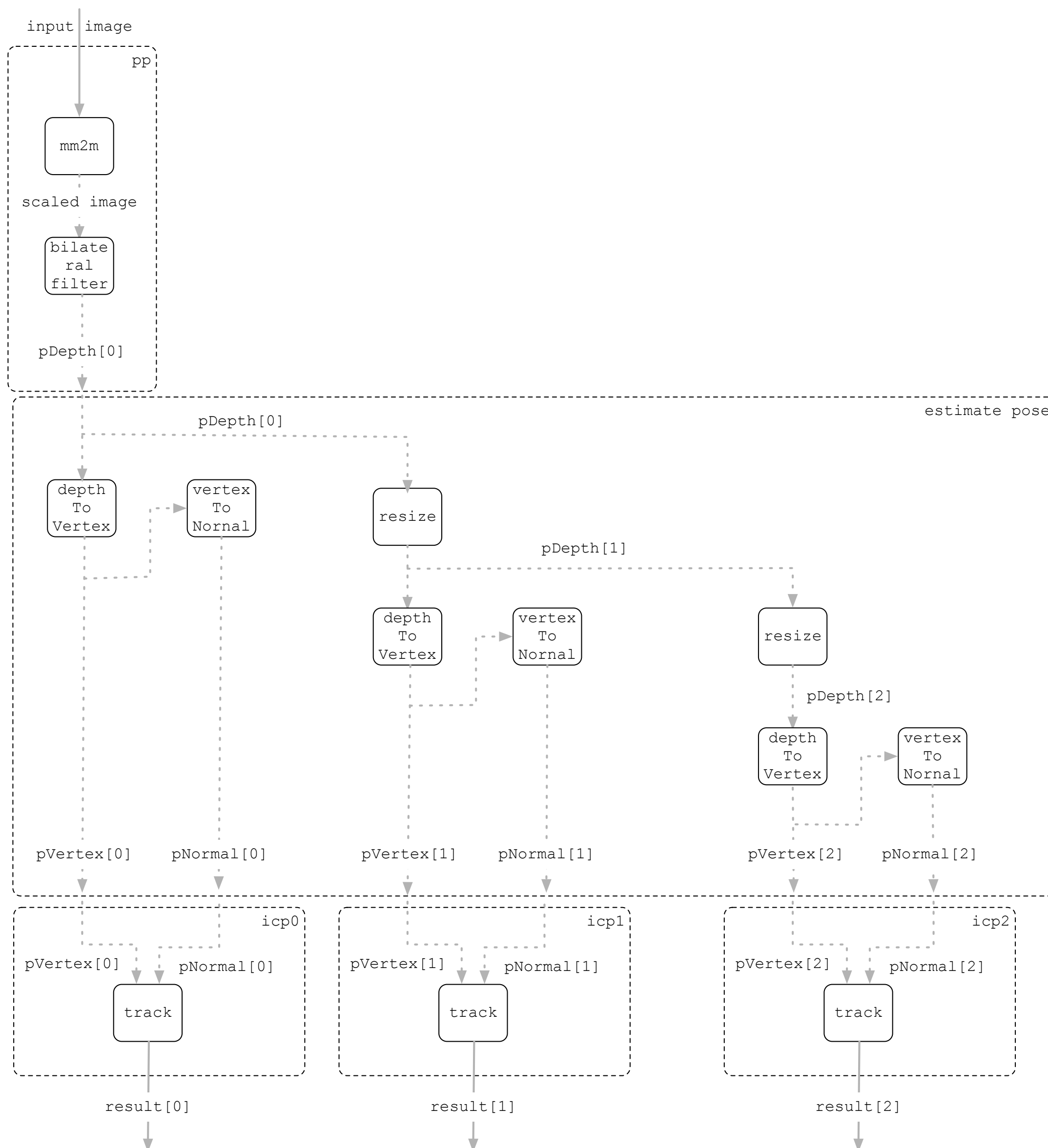
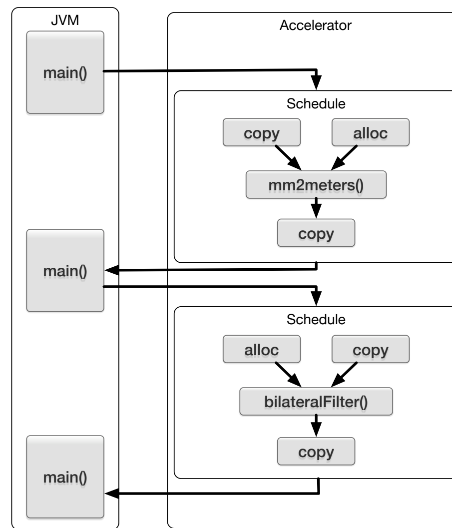


Figure 3.8: Diagram shows the first three stages of the Kinect Fusion pipeline: preprocessing (pp), estimate pose, and ICP. Notice the optimisation boundaries of Tornado (dashed black boxes) and how it is able to eliminate a large amount of data movement between tasks (data grey lines).

3.1.6 Design Rationale

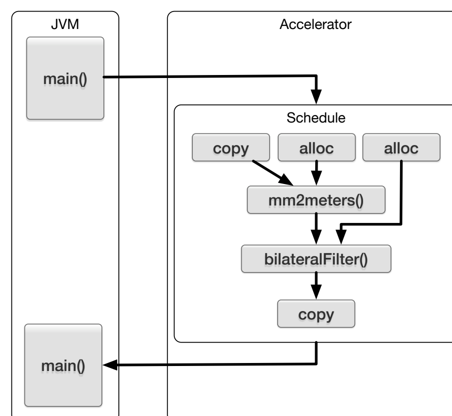
Up to now, the focus has been on describing the Tornado API and how it can be used. This Section will explain the rationale behind the design of the Tornado API with use of code taken from the real-world application described in Section 7.3. One of the first questions a developer might ask is why can I not execute tasks directly?

The answer to this question is that by doing this Tornado not know what the application does before and after executing each task. Thus, leaving it with a very small scope for optimising data movement and making it impossible for it to adequately optimise pipelines like the one in Figure 3.8. To see why this is the case consider Figure 3.9 that takes the first stage of the Kinect Fusion processing pipeline. It has two kernels `mm2meters` and `bilateralFilter` with a read-after-write data-dependency between them – `scaledDepthImage`. If this is implemented as shown in the Listing below Figure 3.9 then this code will execute as two separate tasks as shown in the diagram. There are two problems with this: (1) control bounces back and forth between the host-side code running on the JVM and the device-side code; and (2) Tornado cannot optimise away the data-transfer for `scaledDepthImage` because it cannot guarantee that this variable is not modified on the host in between invoking the task-schedules. Instead, by scheduling the two tasks within a task-schedule (shown in Figure 3.10) Tornado is able to automatically eliminate this redundant work because it knows that no host-side code can modify the variables while the task-schedule is executing.




```
new Task("t0", mm2meters,scaledDepthImage, inputDepthImage, ...).execute();
new Task("t1", bilateralFilter, ouputDepthImage, scaledDepthImage, ...).execute();
```

Figure 3.9: Executing tasks independently results in sub-optimal task execution. Notice how control ping-pongs between the host and the device, blocking the host from making progress. Secondly, notice how there are data-transfers (black-arrows) for the data-dependency between executions.




```
new TaskSchedule("pp").volatile(inputDepthImage)
 .task("t0", mm2meters,scaledDepthImage, inputDepthImage, ...)
 .task("t1", bilateralFilter, ouputDepthImage, scaledDepthImage, ...);
```

Figure 3.10: Executing a task-schedule allows Tornado to optimise data movement. By changing the optimisation scope of Tornado both these tasks can be executed asynchronously on the device and eliminating the need to synchronise `scaledDepthImage`.

3.2 Writing High-Performance Code

For many developers, heterogeneous computing is synonymous with parallel programming as many devices require developers to employ parallelism to obtain high performance. However, this causes a problem when implementing Tornado as Java is not overly suited to exploiting a fine granularity of data-parallelism. Although there is excellent support for concurrent and multi-threaded programming: using Java 8 Streams, the Fork-Join API [81], Executors or even `java.lang.Thread`. However, bar Java 8 Streams, these approaches to parallelism are unsuitable for heterogeneous applications as they work at too coarse a granularity and require explicit application decomposition.

One option that would provide Tornado with the ability to support data-parallelism is to integrate Tornado with the Java 8 Stream API – an approach already trialled by [68]. In principle, this would allow a developer to provide the API with a lambda function and Tornado would be responsible for applying the function to all elements within a data stream in parallel.

This is similar concept to data-parallel functional operators [14, 87] and to a lesser extent algorithmic skeletons [32]. There is also a large amount of prior work that uses this approach: [18, 19, 24, 25, 27, 33, 42, 46, 47, 63, 68, 114]. However, there are some drawbacks to this approach: (1) existing code will need to be reimplemented to use the new API, (2) developers can only use the patterns that are available to them, and (3) developers are restricted to only using systems or devices where patterns have been implemented.

Therefore, Tornado takes a similar approach to parallelism as OpenACC and OpenMP, by providing compiler support for parallelisation of loop-nests. Hence, developers have more freedom to express parallel code, and the compiler will take care of parallelising their code in a device specific way. Moreover, by not using pre-written templates, many different parallelisation schemes and parameters can be tried at runtime using dynamic configuration (see Section 3.3).

As parallelism and the parallel programming of heterogeneous architectures is a vast subject, this thesis is going to concentrate on the bare essentials. Therefore, it will focus on optimising loops without data-dependencies. However, this does not mean that Tornado cannot support more complex codes or parallelisation schemes – this is left as future work.

```
for (int i = get_global_id(0); i < c.length; i += get_global_size(0)) {
  c[i] = a[i] + b[i];
}
```

Listing 3.7: Fine-grain Parallel Schedule

```
int id = get_global_id(0);
int block_size = (c.length + get_thread_size - 1) / get_thread_size(0);
int start = id * block_size;
int end = min(start + bs, c.length);
for (int i = start; i < end; i++) {
  c[i] = a[i] + b[i];
}
```

Listing 3.8: Coarse-grain Parallel Schedule

3.2.1 Parallelisation In Tornado

As discussed earlier in Section 2.4.1 hardware accelerators all behave differently and often they perform better when the code is parallelised in an amenable way. Therefore, Tornado allows code to be parallelised differently for each device by supporting the three most common options: no parallelism (for SISD devices), fine-grained data-parallelism (for SIMT devices), and coarse-grained data-parallelism (for MIMD devices).

The fine-grained schedule is designed to execute each loop-iteration in different threads – an approach targeted towards highly-parallel devices like GPGPUs (see Section 2.4.5). An example is shown in Listing 3.7. Whereas the coarse-grained schedule is designed to execute a block of loop-iterations in different threads and is targeted at latency-orientated processors like x86 (see Section 2.4.3). An example is shown in Listing 3.8.

The key features of these parallel schedules are that they can cope with irregular iteration-spaces and the work performed by each thread is adaptable by launching more or fewer threads on the device. Generating code like this introduces a slight overhead by including a loop structure inside the generated code. However, this cost is amortised because these kernels can be tuned to use different thread-group configurations via dynamic configuration (see Section 3.3). In Section 7.8.2 this feature is used to obtain a 14% increase in performance without having to re-compile the task.

3.2.2 Collections API

To enable developers write efficient code, Tornado provides a number of types such as short-vector types (`Float3` and `Byte4`), and containers of short-vector types (`ImageFloat3` and `ImageByte4`). These types are heavily used in Computer Vision applications as they are a natural fit for storing images in RGBA format or 3D coordinates of a point in space. The Collections API is implemented entirely in Java and can be used outside of the Tornado framework. However, since the Collections API is co-designed with Tornado's dynamic compiler, it can optimise the layout of both types and collections of types. Moreover, the compiler is also able to map certain functions directly onto OpenCL intrinsics - examples of this are the `dot` and `cross` functions. Later in Section 6.2.1 will describe how the example code that uses the Collections API is JIT compiled.

3.3 Managing Uncertainty

One of the most challenging aspects of developing, or even porting, heterogeneous applications is that every hardware accelerator behaves differently. Developers using OpenACC, OpenMP and OpenCL, for example, are forced through a cycle of re-compilations, at best, to port their applications onto each new accelerator. Additionally, it is not uncommon for them to be forced into implementing an entirely new OpenCL kernel for each new device. Typically, this because a different parallelisation scheme needs to be used or to re-work an algorithm to better fit the characteristics of the device. This costly cycle becomes an impasse when more than a couple of devices need to be supported. The underlying problem is the closed-world assumptions that these languages make (see Section 2.2.2) and as a result developers are likely to embed multiple sets of distinct settings and optimisations within the source code. A good example of this is highlighted later in Section 7.5.1. Essentially, developers are required to specify optimal configurations for each data-parallel kernel on a per-device basis. For example, if an application has ten kernels and targets ten devices then up to 100 different configurations may need to be identified and recorded. However, the likelihood of the application using more than a couple of these configurations is low. Therefore, this process becomes wasteful for the developer while simultaneously reducing the maintainability of the application.

```
# assigning an entire task-schedule to a specific device
$ tornado -Dpreprocessing.device=0:0 <application>

# assigning individual tasks within a task-scheduleto specific devices
$ tornado -Dpreprocessing.t0.device=0:1 \
          -Dpreprocessing.t1.device=1:0 \
          <application>

# defining the workgroup and blocksizes used for a specific task
$ tornado -Dpreprocessing.t0.workgroup=16,16 \
          -Dpreprocessing.t0.blocksize=8 \
          <application>

# specifying where to find the OpenCL implementation of a specific task
$ tornado -Dpreproceesing.t1.source=./opencl/t1.cl <application>
```

Listing 3.9: Dynamic configuration is designed so that an application can be configured after it has been compiled. Above are some examples of how a Tornado application can be optimised by passing configuration flags on the command line when the application is executed. These examples are taken from the Kinect Fusion application described in Chapter 7.

Tornado solves this problem by dynamically compiling code for each specific device. This means that there is no need to embed assumptions on a per-device basis. Instead, tuning parameters can be provided on the command line (or in a configuration file) on a per system basis. Hence, the application is now optimisable without having to either modify the source code or recompile the application. Importantly, this also applies to choices like the parallelisation scheme that should be used; thus allowing the application to be quickly adjusted on an individual accelerator basis. Moreover, if a range of possible parameters exist then the application can be easily adapted to explore the different options dynamically. Overall, these changes now make it possible for the end-user, and not the developer, to optimise performance for each system. Listing 3.9 shows an example of how configuration parameters can be provided on the command line. Finally, this ability of Tornado is evaluated in Section 7.8 where a speed-up of 14$\times$ was achieved over OpenCL as a result of being able to quickly experiment with different settings in the OpenCL driver.

3.3.1 Task Metadata

One of the key drivers behind the task abstraction is that it allows meta-data to be attached to a method invocation. This allows the developer and Tornado to store and retrieve task specific information. Perhaps one of the most common uses of task metadata is to set device specific optimisation flags on a per task basis. Typically, this

can be done either programmatically by the developer inside the application or on the command line when a Tornado application is executed. Listing 3.9 shows how to dynamically configure tasks and task-schedules from the command line. However, what is more powerful is that the developer is able to modify meta-data both dynamically and from within the running application. A clear example of this is given later in Listing 3.12 where the hardware accelerator used for each task invocation is being determined randomly.

3.3.2 Dynamic Compilation

Tornado is unlike the majority of other heterogeneous programming languages as it employs fully dynamic compilation. Compilation occurs at runtime, the first time a task-schedule is executed or if the meta-data changes on a task. The advantage of dynamic compilation is that the compiler does not need to generate multiple different versions of the code – one for every possible target device – ahead of time; instead, it generates code on demand.

3.3.3 Task Tuning Parameters

After compilation, the performance of code can be altered using a range of different parameters. Typically, this might include the dimensions of the thread groups used to execute device-side code. These parameters exist in the task-meta data and, subsequently, can be updated dynamically. In Section 7.8.2 this ability is used to obtain a 14% increase in performance without having to re-compile any code.

3.3.4 Dynamic Configuration Examples

One of the key advantages of Tornado is that it allows a wide range of options to be configured dynamically. Therefore, unlike most other heterogeneous programming languages, an application can be adapted to its system configuration in-situ. However, by simplifying how coordination logic is expressed, Tornado can also perform some very powerful dynamic transformations.

For example, consider Listing 3.12 where a task-schedule with two tasks is running over multiple accelerators. On each iteration, it then randomly selects two different accelerators to use. Although this example is quite synthetic, it demonstrates the basis of how an application might dynamically optimise task placement on an unknown system. Apart from varying task-locality, it is also possible to dynamically modify other

```
float[] array = {0, 1, 2, 3};

// define a multi-stage pipeline
// each task performs a[i] *= 1 on the array
TaskSchedule schedule = new TaskSchedule("s0");
for (int i = 0; i < numKernels; i++) {
  schedule.task("sscal" + i, BLAS::sscal, a, 1.0);
}

// ensure that each task works on the lastest version of array
schedule.volatile(a);

TornadoDriver driver = getTornadoRuntime().getDriver(0);

// execute on device 0
schedule.mapAllTo(driver.getDevice(0));
schedule.execute();

// execute on device 1
schedule.mapAllTo(driver.getDevice(1));
schedule.execute();
```

Listing 3.10: Tornado allows meta-data to be specified at two different granularities: for an entire task-schedule or for each task. This example shows how a multi-stage processing pipeline can be migrated onto a different hardware accelerator by updating the meta-data associated with the task-schedule.

task meta-data. For example, to change the parallelisation scheme applied to a particular task – as is done in Section 7.8.4 to improve performance by 17%. Listings 3.10 to 3.12 provide some examples of how dynamic configuration can be used to migrate entire task-schedules and tasks between devices.

```
float[] array = {0, 1, 2, 3};

// define a multi-stage pipeline
// each task performs a[i] *= 1 on the array
TaskSchedule schedule = new TaskSchedule("s0");
for (int i = 0; i < numKernels; i++) {
  schedule.task("sscal" + i, BLAS::sscal, a, 1.0);
}

TornadoDriver driver = getTornadoRuntime().getDriver(0);

// assign each task to a specific device
schedule.getTask("sscal0").mapTo(driver.getDevice(0));
schedule.getTask("sscal1").mapTo(driver.getDevice(1));
schedule.getTask("sscal2").mapTo(driver.getDevice(2));

// execute the task-schedule once
schedule.execute();
```

Listing 3.11: This example shows how each task within a multi-stage processing pipeline can be executed by different hardware accelerators. Here the code is updating the per-task meta-data to override where each task should execute.

```
// define a two stage pipeline
TaskSchedule schedule = new TaskSchedule("s0")
  .volatile(a)
  .task("t0", SimpleMath::vectorMultiply, a, b, c)
  .task("t1", SimpleMath::vectorAdd, c, b, d)
  .sync(d);


// query the number of devices attached to the system
TornadoDriver driver = getTornadoRuntime().getDriver(0);
int maxDevice = driver.getDeviceCount();
final Random rand = new Random(7);
final int[] devices = new int[2];

// invoke the pipeline multiple times
for (int i = 0; i < num_iterations; i++) {

  // randomly select a device for each task
  devices[0] = rand.nextInt(maxDevice);
  devices[1] = rand.nextInt(maxDevice);

  // update the task meta-data
  schedule.getTask("t0").mapTo(driver.getDevice(devices[0]));
  schedule.getTask("t1").mapTo(driver.getDevice(devices[1]));

  // execute the pipeline
  schedule.execute();
}
```

Listing 3.12: One of the key design goals of Tornado is to afford the developer with greatest amount of flexibility when executing in heterogeneous systems. This example shows how it is possible to create a processing pipeline with two stages – `multiply` and `add` – and have each stage execute on a randomly selected accelerator. One of the key observations is that the developer does not explicitly know how many devices are being used and it is not necessary to perform any explicit data transfers between the devices. The complexity of ensuring that data is moved between devices is handled entirely by Tornado.

3.4 Summary

In this Chapter, the task-based programming model used by Tornado has been introduced in Section 3.1. There are two abstractions at the core of the Tornado API – the task and the task-schedule. These abstractions allow the decoupling of an applications coordination logic from its computation logic. Thereby, making it possible for developers to construct complex processing pipelines. One such pipeline can be seen in Listing 3.12 where two tasks are moved between randomly selected devices on each invocation of the pipeline.

In Section 3.2 Tornado's ability to generate parallel code for devices is introduced. The parallelisation schemes supports aim to target SISD, MIMD, SIMD and SIMT devices. Moreover, Section 3.3 introduces the notion of dynamic Configuration. As Tornado supports dynamic compilation then it is possible for a task to be configured without having to modifying the source code. For example, the application is able to inspect all the devices in the system and assign one to each task (Listing 3.11). Or a user can override the choice of parallelisation scheme that is used for a device (see Section 7.8.4). Over the next Chapters the machinery that implements these features of the Tornado API will be introduced.

4 | Tornado Virtual Machine

One of the key aspects of heterogeneous programming is the need to coordinate the execution of an application across multiple devices. This is a key challenge that is highlighted in Section 1.1. To solve this problem Tornado aims to provide a clean separation between the code that defines a computation and the code that coordinates its execution – this code will be referred to as the computation logic and coordination logic respectively. This Chapter describes the Tornado Virtual Machine (TVM) a novel component that provides a virtualisation layer between an application and the hardware accelerators available to it. Those familiar with programming language implementation will immediately recognise the TVM as a bytecode interpreter.

Tornado aims to improve the state-of-the-art regarding the programming of heterogeneous systems, something that it achieves this via the co-design of its compiler, language, and runtime infrastructure. However, all of the advancements are made possible by the virtualisation support that the TVM provides. Most notably, this allows an applications coordination logic to become a series of operations applied to an implicitly defined device. The benefits of this parameterisation have a profound impact on the ability to effectively program heterogeneous systems. For instance, Section 4.5.3 describes how the TVM can dynamically migrate tasks between devices.

The remainder of this Chapter describes the design and operation of the TVM. Additionally, it highlights some of the main features that improve developer productivity, such as its asynchronous execution modes and how it manages variables that reside in multiple memories. Finally, it concludes with discussions on measuring performance and the limitations of the TVM.

4.1 A Nested VM

The Tornado Virtual Machine (TVM) is responsible for providing a virtualisation layer between the application and any hardware accelerators that are attached to the system;

much in the same way as the Java Virtual Machine does for the underlying processor architecture and operating system. It fulfils this role by by being able to translate a stream of bytecodes into calls to the underlying software driver for each hardware accelerator – in the case of this thesis this is the OpenCL runtime system. What should be noted is that one of the aims of the TVM is to decouple the application from being linked against a specific system libraries like OpenCL – as this is not portable. Hence, by implementing the TVM as a bytecode interpreter gives Tornado the ability to dynamically discover and use different system libraries. Therefore, there is no need for any Tornado application to be linked directly against the OpenCL runtime and allowing applications to be distributed entirely as Java bytecode. Moreover, this design also makes it possible for the TVM to interoperate with multiple system libraries via its device interface (see Section 4.3.6) – a feature that might allow both CUDA and OpenCL devices to be utilised within a single application in the future.

One of the frequent points of confusion in Tornado is that the TVM is not actually a fully functional virtual machine (VM) – like the Java Virtual Machine (JVM). Instead it has been designed to sit on top of the JVM: resulting in a novel VM-in-VM or a nested VM design. There are two reasons for doing this: (1) that it avoids having to write or modify an entire VM; and (2) that it allows the TVM to be portable across different JVM implementations.

Moving forward, it is suggested that the reader thinks about the TVM and TVM bytecode as being a superset of both the JVM and Java bytecode. In reality, the TVM is designed with this in mind and in the future it might be possible for the TVM to be absorbed by the JVM to resolve some of the issues mentioned in Section 6.3.

4.2 Overview

Developers often prefer to write code at a high-level of abstraction, like the following code that defines a high-level task, `t0`, in Tornado that can execute on a hardware accelerator:

```
task t0 foo(a, b, c)
```

The syntax is very similar to a method or a function call – it specifies that the method `foo` should be invoked with parameters `a`, `b` and `c`. The problem is that the high-level of abstraction of this task prevents it from being executed directly on a hardware accelerator. Instead, it needs to be broken down and executed as a series of smaller low-level tasks that interact directly with the device. In this instance, low-level tasks

```
      setup          1, 1 , 0              ; create new execution context:
                                           ;   - 1 device stack
                                           ;   - a 1 entry device list
                                           ;   - no event queues
foo_entry:
      begin                                ; start of executable bytecodes
      copy_in        0                     ; copy_in arg[0]
      copy_in        1                     ; copy_in arg[1]
      copy_in        2                     ; copy_in arg[2]
      launch         0, 0, 0, 3, 0         ;
      push_arg_ref   0                     ; push arg[0] onto device stack
      push_arg_ref   1                     ; push arg[1] onto device stack
      push_arg_ref   2                     ; push arg[2] onto device stack
      copy_out       0                     ; copy_out arg[0]
      copy_out       1                     ; copy_out arg[1]
      copy_out       2                     ; copy_out arg[2]
      sync                                 ; block until all bytecodes in
                                           ; context are complete
      end                                  ; terminate TVM
```

Listing 4.1: Tornado Bytecode: Example – `foo`

need to be generated to:

- Copy any variables that need to be passed by-reference onto the device. In this case all three variables `a`, `b` and `c`.
- Build a device-side parameter list that will be passed to the device-side version of `foo`.
- Submit the device-side version of `foo` for execution.
- Copy back any variables that have been modified on the device.
- Ensure that all low-level tasks have completed before returning control back to the application.

The Tornado Virtual Machine is designed to virtualise the execution of each of these low-level tasks. A programming language or API can utilise the TVM by generating TVM bytecode, as shown in Listing 4.1. Notice how similar the bytecode is to other low-level heterogeneous programming languages like CUDA and OpenCL shown in Figures 4.2 and 4.3 respectively. In fact, TVM bytecode is a generalisation of these two listings that can be implemented using different programming languages and frameworks, like CUDA, OpenCL, Vulkan or HSA. The key difference is that TVM bytecode does not directly specify information like the number of threads to

```
// allocate memory on the GPGPU
int *dev_a,*dev_b,*dev_c;
cudaMalloc((void**)&dev_a,SIZE);
cudaMalloc((void**)&dev_b,SIZE);
cudaMalloc((void**)&dev_c,SIZE);

// copy the data from host to GPGPU
cudaMemcpy(dev_a,a,SIZE,
  cudaMemcpyHostToDevice);
cudaMemcpy(dev_b,b,SIZE,
  cudaMemcpyHostToDevice);

// launch the kernel on the GPGPU
foo<<<...>>>(dev_a,dev_b,dev_c);

// copy back the results
cudaMemcpy(c,dev_c,SIZE,
  cudaMemcpyDeviceToHost);

// free up memory on the GPGPU
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
```

Listing (4.2) CUDA

```
// allocate memory on the device
cl_mem a_d, b_d, c_d;
a_d = clCreateBuffer(ctx,flg,SIZE,...);
b_d = clCreateBuffer(ctx,flg,SIZE,...);
c_d = clCreateBuffer(ctx,flg,SIZE,...);

// copy the data from host to device
clEnqueueWriteBuffer(q, a_d, false,
  0, SIZE, a, ...);
clEnqueueWriteBuffer(q, b_d, false,
  0, SIZE, b, ...);

// launch the kernel on the device
clEnqueueNDRangeKernel(q, foo_k, ...);

// copy back the results
clEnqueueReadBuffer(q, c_d, false,
  0, SIZE, c, ...);

// free up memory on the device
clReleaseMemObject(a_d);
clReleaseMemObject(b_d);
clReleaseMemObject(c_d);
```

Listing (4.3) OpenCL

Figure 4.1: Executing `foo(a,b,c)` on an accelerator via CUDA and OpenCL.

use or even the code to execute. This indirection makes it possible to configure heterogeneous application at runtime dynamically. An example of where this is useful is changing the device on which code should execute – something that can even be varied dynamically.

4.3 Architecture

The Tornado Virtual Machine (TVM) is an abstract machine that executes the coordination logic of a heterogeneous application. Its role is to virtualise the execution of the low-level tasks that are necessary to program a heterogeneous system. For those familiar with implementing programming languages it can be considered to be a bytecode interpreter. The TVM is comprised of three components: an execution engine, a device interface, and an object cache. The execution engine interprets the bytecode, issuing commands to dynamically loadable clients via the device interface and the object cache is responsible for tracking the state of variables across different devices. Internally, the TVM operates on an execution context that contains: the bytecode to execute, the data used by device-side code, and per task meta-data. Once an execution context is provided to the TVM, the TVM will execute all of the bytecode instructions

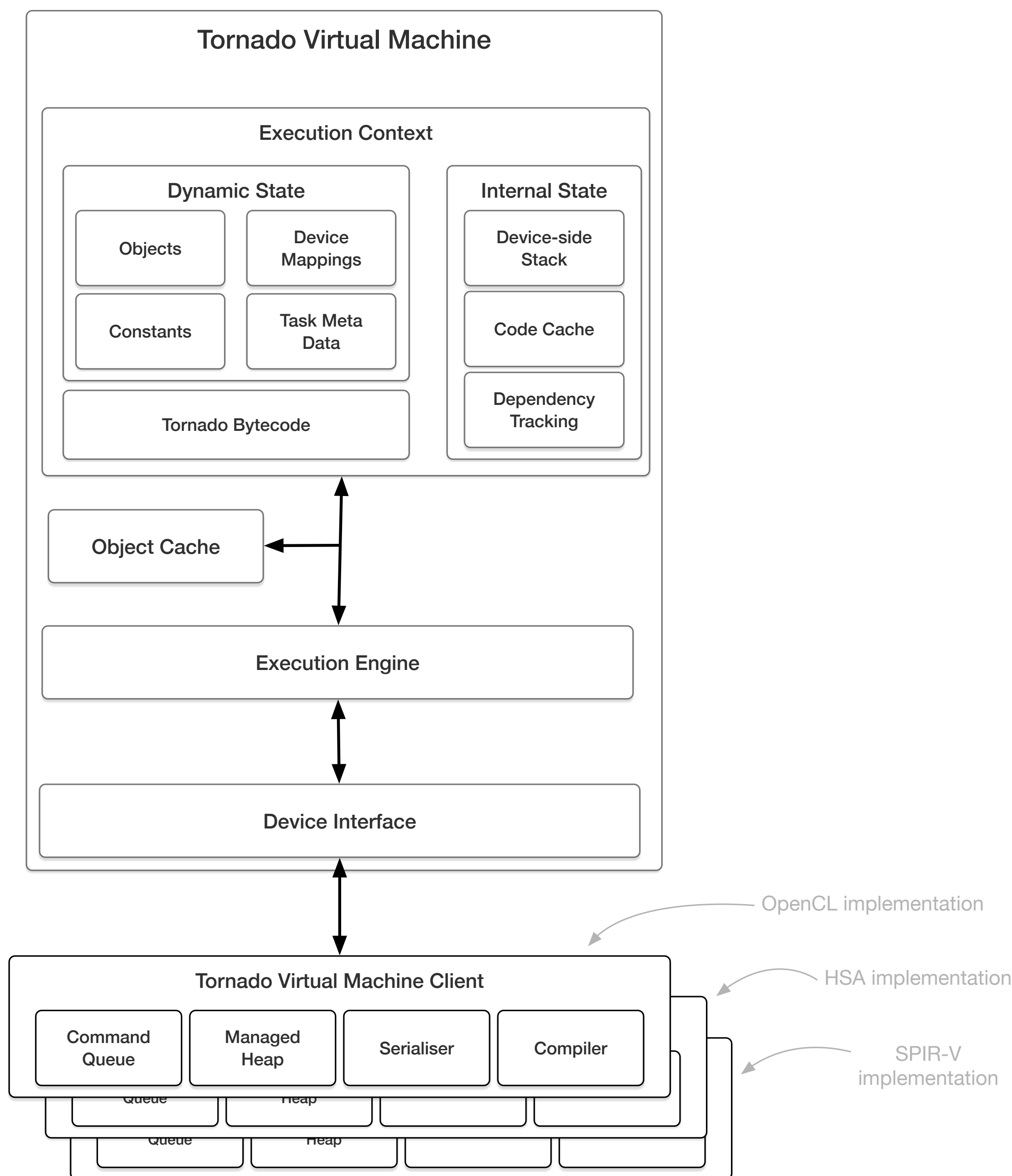


Figure 4.2: An architectural overview of the Tornado Virtual Machine. The TVM provides a virtualisation layer between the application and the underlying hardware. This is achieved by translating bytecodes into method calls that execute on a dynamically loadable TVM client. Where each client is responsible for implementing support for a specific device or low-level platform – such as OpenCL, HSAIL/HSA, and SPIR-V/Vulkan As there is no direct coupling between the execution engine and client then it is possible to simply swap out each client so that the bytecode running on the TVM will target a difference device.

until completion (or when an error occurs).

Initially, the execution context requires the underlying application to provide four lists: (1) a list that holds variables that are passed by-value; (2) a list that holds variables that need to be passed by-reference; (3) a list that holds the list of devices to use; and (4) a list that holds meta-data for each high-level task. Collectively, these lists capture the state of the coordination logic running on the host-device and make it available to the TVM. As all of this state is owned and controlled by the application and can be updated freely outside of the TVM, hence, it is referred to as *dynamic* state.

The TVM is designed to handle the complexities of moving both data and execution between devices. Presently, it operates using an object caching scheme where objects are always created on the host-device but can be duplicated and modified on the device-side. There are two components that make the scheme work: the object cache that provides centralised management of an objects state across all devices; and a TVM client that performs both serialises and transfers between devices.

Finally, the TVM does not provide code generation support directly. Instead, support for code generation is delegated to each TVM client – allowing each driver to provide either a specialised JIT compiler, pre-compiled binaries or even link execution to a native library. In the implementation of Tornado evaluated in Chapter 7 the GRAAL [39] dynamic compiler is augmented to support the generation of OpenCL C – allowing Tornado to execute on OpenCL compatible devices.

4.3.1 Execution Context

The execution context is designed for high-performance, dynamic compilation, and re-configuration. Each execution context holds all the code and data required to execute a single coordination program on the TVM. In it the code is provided as bytecode and is encoded according to the specification provided in Section 4.4. The data it contains encapsulates both the state of the application and the internal state of the TVM. The application state – such as the mapping of tasks to devices and variable lists – is owned by the application and is assumed to be dynamic in nature. This dynamic state enables applications running atop of the TVM to be dynamically configurable – even at runtime. However, the internal state – such as the device-side call-stacks and code caches – is owned by the TVM and can persist across invocations. This is not available to the application but allows the TVM to cache frequently used configuration to improve performance over multiple invocations of the TVM. All state, both dynamic and internal, is exposed to bytecodes as indexed lists allowing them to be indirectly

referenced. For instance, by changing an entry in the device list all bytecodes that reference this entry will now by applied to the updated device. This indirection allows changes like this to be made without the need to regenerate or re-optimise the bytecode. Besides improving configurability, the saving of internal state provides a mechanism for the TVM to preserve state across TVM invocations. This is essential if the bytecode is to be executed multiple times as it reduces costly overheads, like compilation, as it does not need to be performed every time the bytecode is executed.

4.3.2 Dynamic State

The dynamic state contains indexed lists of variables that are required by the device-side code. These lists are created by aggregating the parameters lists of all high-level tasks executed by the coordination logic together. The lists are split according to whether the variable needs to be passed by-value (constants) or by-reference (objects). Bytecodes uses the constant and object lists to specify host-side variables when transferring data or creating a device-side call-stack. Splitting the lists is necessary as both sets of variables need to be handled differently – as a large number of devices do not share physical memories with the host and therefore it is not possible to access the variable directly on the host. Instead, variables that are passed by-reference need to be translated from host-side variables into device-side variables. The process for doing this varies by device but, in general, requires the variable to be allocated space and then copied to the device-side memory. Similarly, a indexed list of devices is provided by the application that is used by bytecodes to specify which device they should be issued to. As Tornado is dealing with heterogeneous architectures, each device may have its own unique configuration settings. To accommodate this each high-level task has meta-data associated with it. The meta-data is a simple key-value store that is exposed to the TVM and subsequently TVM clients. It is through this meta-data that the application can communicate task and device specific configurations to and from TVM clients. For instance, the meta-data may contains information about what code to execute, what compiler optimisations to apply, how many threads to execute it with or telemetry about the device-side code.

4.3.3 Internal State

To improve the runtime performance of the TVM it is able to persist some of its internal state across invocations. This is important when coordination logic is executed

frequently – for instance to process a continual stream of data or as part of an iterative algorithm. One of the most costly operations performed inside the TVM is JIT compilation. Therefore, to reduce the cost a code cache exists into which TVM clients can insert any code that they have generated. By doing this, all future attempts to execute the task will use the code found in the cache and avoid the cost of multiple compilations.

Any device-side code that is executed needs to be provided with a list of parameters. These parameter lists needs to be generated for each device as any host-side pointers need to be translated into a device-side pointer. Typically, these pointers exist where variables are passed by-reference as part of the device-side codes formal parameter list or as embedded pointers within a composite data-type. All pointers are automatically translated inside the TVM via the object cache. The resulting device-specific call-stacks are also preserved so that they only need to be updated when an argument is changed.

4.3.4 Bytecode

The TVM is programmed via bytecode requires it to be provided as a `byte[]` – this allows applications to have flexibility on how the bytecode is provided. For example, the bytecode can be stored in a file or generated on-the-fly. The format of the bytecode is described in Section 4.4. Currently, the bytecode does not support control-flow operations like branching and can be considered to be a linear list of operations. However, there are no technical reasons why these features cannot be added at a later date. Each bytecode has well defined semantics and provide a easy target for automated code generation. The benefit of this is that the bytecode represents the coordination logic in a intermediate representation (IR) – like in a compiler – and by doing this it is possible to apply compiler-like optimisations to improve its performance. For instance, Sections 5.3.3 and 5.3.5 describe how redundancy elimination and strength reduction are used to minimise the costs of data movement in TVM bytecode.

4.3.5 Execution Engine

Once an execution context is available, the TVM is able to start interpreting the bytecode. The bytecode contains a pre-amble that is used to configure the internal state of the TVM and run any one-off initialisation that needs to occur. After the initialisation code has been executed the TVM can start executing the main body of the bytecode.

Each bytecode is processed sequentially until either an `end` bytecode is executed or an exception occurs.

The bytecode structure has be designed to enable dynamism: references to specific devices or variable are made indirectly. This means that the behaviour of the TVM is parameterised by the dynamic state. For example, changing the device mapping state means that the bytecode can execute on a different device. The benefit of the TVM is that there is no need to regenerate the bytecode to do this – just update the mapping.

TVM bytecodes can be categorised into control-flow – they alter the control-flow of the program inside the TVM – or device operations. Device operations are designed to represent abstract operations on a virtual device. They do not impose restrictions on how each operation should be implemented but allow the implementation to be delegated to the TVM client. A clear example that shows the importance of this are the data-movement instructions like `copy_in` and `copy_out`. These instructions have to make host-side variables available on a specific device. The semantics of these instructions is that the variable needs to be copied into the target memory. However, these bytecode do not have specific data-layout requires on either the host- or device-side. This allows each TVM client to implement their own device-specific data-layout and serialisation processes.

The most time consuming bytecodes – the ones that involve the transfer of data – are designed to be issued asynchronously to devices. In this situation, the TVM will issue these operation to the TVM client and return immediately. Asynchronous bytecodes return a handle that can be used by the TVM to track their progress and obtain profiling information.

4.3.6 Device Interface

The device interface is designed to abstract away the implementation details of the underlying heterogeneous devices. The TVM is structured so that all device specific functionality is implemented inside each TVM client and access via the device interface. Typically, the TVM client is responsible for implementing device specific functionality such as: a compiler to generate device-side code; a data serialiser to move data between host- and device-side memories; and a memory manager to manage device-side memory. This design decision means the TVM bytecode remains device-neutral – i.e. it can be applied to any implementation of the TVM client. A practical benefit of this is that it also simplifies the implementation of the TVM client. For instance, the TVM client is free to implement its own device-specific (or application specific)

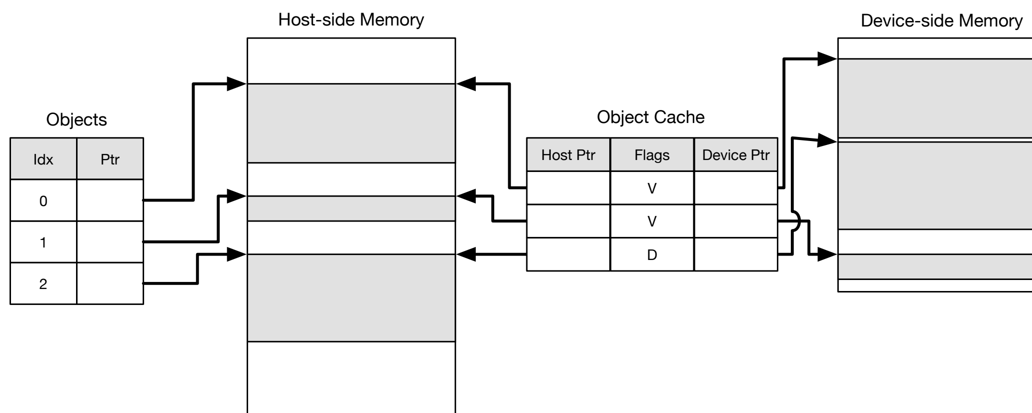


Figure 4.3: The role of the object cache is to allow objects that exist in the host JVM to be cached (or mirrored) in device-side memory. The Object Cache is able to translate the memory address of an object between the host-side and device-side via a lookup table. This table also tracks the state of device-side objects: they can be valid (V) when they have a valid allocation, and dirty (D) when they have been updated by device-side code but not synchronised with the host. Using this scheme allows objects to persist across invocations of task-schedules.

data-layouts to achieve higher performance. Normally, this is not possible in a shared-memory system as another threads may need to access the same data and might incur severe performance penalties if the layout is changed. The communication of data between devices is performed via the data serialiser inside the TVM client. Again this makes it easier to implement device-specific optimisations as it leaves the implementer of the TVM client in full control of: the compiler, serialiser, and memory manager.

Each TVM client is responsible for generating its own machine code for the devices it exposes to Tornado. Normally, this occurs the first time a task is launched – via the `launch` bytecode. At this point the TVM client is provided with the meta-data for the task that is to be launched. The TVM client can then query the meta-data to determine how it should execute the task. For instance, the meta-data might specify a file containing source code that can be compiled, provide a reference to reflected method that should be compiled or map the task onto a native library. If the TVM client uses JIT compilation then the TVM provides a code cache facility that enables the client to avoid repeated compilation.

4.3.7 Object Cache

A fundamental problem in heterogeneous applications is managing variables that exist in different memories – such as host-side and device-side. This problem only exists for composite data-types as they are either passed by-reference using a pointer or may contain pointers to other data. For device-side code to work correctly all host-side pointers need to be translated into device-side pointers. In the TVM this is done via the object cache.

Figure 4.3 shows the process that translates between host-side and device-side pointers. The process start by being provided with a host-side pointer. Typically, these are found by either looking one up in the object list (inside the execution context) or when the data serialiser encounters a new object whilst serialising a variable with a composite data-type. The object cache is indexed by the value of the host-side pointer so that its corresponding cache entry can be located easily. If no entry exists in the cache then the variable also does not exist in device-side memory – this could be due to a number of reasons such as the variable never being allocated on the device or that it previously existed but was removed by a garbage collector. In this situation a new device-side pointer can be requested via the TVM client. Otherwise, the return object cache entry contains the value of the device-side pointer and a set of flags indicating the device-side status of the pointer. Typical flags can be used to indicate whether a pointer is valid and whether the allocation is dirty – meaning that it has been modified by device-side code but has not been synchronised with the host. To understand how the object cache is used refer to Section 5.4 which explains how Tornado optimises data movement between different task-schedules and across multiple devices.

4.4 Bytecode Specification

The TVM bytecode is designed to support the execution of heterogeneous applications. Therefore, the main differences between it and any other bytecode format is that its bytecodes are specifically targeted at managing execution in heterogeneous systems. Table 4.1 provides a list of the key bytecodes supported by the TVM and their semantics. Section 5.4 also describes additional bytecodes that work with the object cache to improve the performance of task-schedules that execute across multiple devices. As these extra bytecodes are specialisms of ones described in Table 4.1 they are not discussed in this Section.

Name	Output	Parameters	Async	Description
setup		$\{c,t,q\}$		Instructs the TVM to create a new execution context that uses c devices, t tasks and q event queues.
begin				Indicates the start of the coordination logic.
end				Instructs the TVM to terminate.
allocate		$\{idx\}$		Allocates enough storage in the memory of the currently active device for the reference variable at position idx in the variable list.
read_host	$\{event\}$	$\{mode, ridx, didx, eidx\}$	✓	Performs a deep-copy of the reference variable at position $ridx$ in the variable list from the host-device into the memory of the device $didx$ in the device list. The copy is performed once all events in the event queue $eidx$ have completed.
launch	$\{event\}$	$\{mode, id, didx, tidx, n, eidx\}$	✓	Launches the task at index $tidx$ in the task list on the device at position $didx$ of the device.list after all events in the event list with index $eidx$ have completed. This tasks requires n parameters to be pushed onto the device-side stack – these are encoded directly after this instruction.
push_arg		$\{idx\}$		Pushes the variable at position idx of the parameter list onto the specified device-side stack.
push_event_queue		$\{idx\}$		Appends the event generated by the last executed bytecode to the back of the event queue at position idx.
write_host	$\{event\}$	$\{mode, ridx, didx, eidx\}$	✓	Performs a deep-copy of the reference variable at position $ridx$ in the variable list from the device at position $didx$ in the device list to the host-device. The copy is performed once all events in event queue $eidx$ have completed.
sync				Blocks until all outstanding operations are complete.

Table 4.1: Tornado Virtual Machine Bytecode Specification

Listing 4.4 provides an example of TVM bytecode that executes a method, `foo`, on a hardware accelerator. The bytecode starts by specifying an initialisation procedure that is executed the first time an execution context is loaded into the TVM. This procedure must start with the `setup` bytecode which tells the TVM how many device-side call-stacks, code cache entries, and event queues need to be created in this execution context. Afterwards, the bytecode is free to specify any other optional tasks that need to be performed. The initialisation procedure ends when the bytecode interpreter reaches the `begin` bytecode. At this point the execution engine will mark the current bytecode index and use it as the future entry point into the bytecode. Next, comes the bytecodes that represent the compiled coordination logic of the application. The TVM bytecode format requires this to be enclosed by the `begin` and `end` bytecodes.

The most complex bytecode within the specification is `launch` – which submits device-side code for execution on the TVM client. The complexity stems from this bytecode having to also specify the parameters that need to be passed into the device-side code. Although it is tempting to embed direct references to the parameters this needs to be avoided as doing so will tightly couple parameters to a specific task. However, if parameters are indirectly referenced via the execution context it becomes possible to re-use this context with different parameters – allowing the same bytecode to be re-used with different parameters. Hence, the `launch` bytecode needs to build a call-stack for the remote device. This call-stack is defined by the `launch` bytecode which specifies its size. Then it is populated by subsequent bytecodes that load variables into the call-stack. In the example this is what the `push_arg_X` bytecodes are doing and is illustrated in Figure 4.4.

4.4.1 Execution Model

The host-application is able to either execute the coordination logic synchronously – waiting for it to complete before continuing – or asynchronously – continuing its execution while the coordination logic executes in the background. Both of these modes have advantages. Synchronous execution provides well-defined synchronisation points where the application state can be relied on to be up-to-date. Whereas asynchronous execution allows application code to run in parallel with the TVM. However, in this mode there is a problem because the TVM does not have exclusive access to the application state exposed to the TVM. Therefore, the TVM requires the developer to avoid modifying application state whilst it is running.

Presently, the TVM is designed to execute device-side code in complete isolation.

```
      setup          1, 1 , 0            ; create new execution context:
                                         ;   - 1 device stack
                                         ;   - a 1 entry device list
                                         ;   - no event queues
foo_entry:
      begin                              ; start of executable bytecodes
      copy_in        0                   ; copy_in arg[0]
      copy_in        1                   ; copy_in arg[1]
      copy_in        2                   ; copy_in arg[2]
      launch         0, 0, 0, 3, 0       ;
      push_arg_ref   0                   ; push arg[0] onto device stack
      push_arg_ref   1                   ; push arg[1] onto device stack
      push_arg_ref   2                   ; push arg[2] onto device stack
      copy_out       0                   ; copy_out arg[0]
      copy_out       1                   ; copy_out arg[1]
      copy_out       2                   ; copy_out arg[2]
      sync                               ; block until all bytecodes in
                                         ; context are complete
      end                                ; terminate TVM
```

Listing 4.4: Tornado Bytecode Example

This means that device-side code is executed in a sandbox like environment and its updates are only made available on the host via explicit data transfers. There are many benefits to this approach such as data persistence, post mortem analysis, and efficient migrations between devices. Some of these benefits would be candidates for future research topics.

In terms of data, the TVM is able to transparently synchronise data between the host-side and device-side memory. However, it does not permit the creation of new objects: as doing so would require the TVM to be tightly integrated into the JVM – which it is not. Instead, the TVM uses a migratory memory model that is suited to Tornado's task-based programming model. In the memory model the application running on the host-device owns all application state and is responsible for managing all variables. The TVM is able to duplicate existing data from the host-side memory into device-side memory. The duplicated data is free to be modified on each device, however, for it to become available in the host-side memory it will need to be explicitly copied. The exact duplication process used is device-specific and is handled by the data serialiser inside each TVM client. Typically, duplication will perform a deep-copy as all nested variables also need to be made available on the device.

Finally, the TVM is able to support complex out-of-order asynchronous execution modes. These modes rely on the TVM supplying the TVM client with task dependency

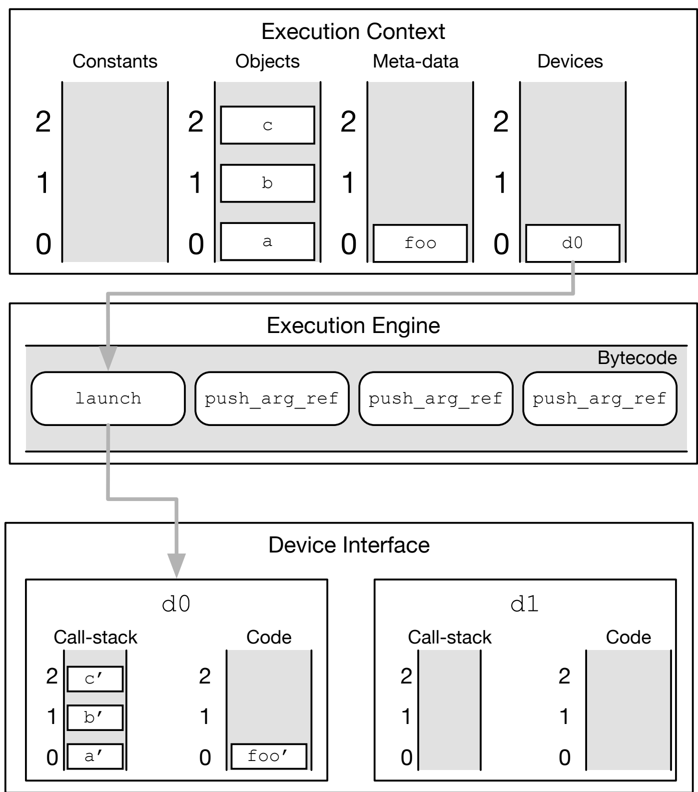


Figure 4.4: The inner workings of the TVM. The execution context (top) captures the state of the application. The execution engine executes the TVM bytecodes serially (left to right). Bytecodes operate using slot numbers in the execution context so that it can be dynamically updated. All interactions with the TVM client is made via the device interface (bottom).

information. Normally, this is a list of tasks that each task requires to be complete before it can execute. The TVM supports the notion of event queues – as each asynchronous task is submitted a handle, called an event, can optionally be appended to an event queue inside the TVM using the `add_dep` bytecode. These queues can then be specified by any asynchronously executing bytecode to provide a list of dependencies to the TVM client. The TVM does not automatically track dependency information, so this needs to be encoded into the bytecode when it is generated. Section 5.3 describes how this feature allows task-schedules with multiple tasks to be optimised.

4.5 Novel Features of The TVM

The TVM is designed to support the execution of compute logic across multiple devices. For this to happen it has three distinct features that support the design of the Tornado API. These features are: virtualisation – the ability of the TVM to abstract away the details of the underlying hardware accelerator (see Section 4.5.1); dynamic configuration – the ability to change mappings between task and device (see Section 4.5.3); and asynchronous execution – the ability to overlap the execution of long running bytecodes (see Section 4.5.2).

4.5.1 Virtualisation

The TVM is primarily designed to provide a virtualisation layer for programming heterogeneous systems. To achieve this it has been designed not to impose any restrictions on how code can be generated or how data is laid out. Instead, these decisions are delegated to each TVM client. All the TVM aims to do is present the TVM client with information about where to find the code it needs to execute and where to obtain the data that it should operate on. Moreover, this approach provides the TVM with a range of different code generation options that enable it to take advantage of new and existing code. For example, there are four ways that code can be provided by the TVM client.

- Each task is mapped onto a native library call.
- A task specifies a pre-compiled binary to execute.
- A task provides the source code of a kernel that should be compiled.
- The task is JIT compiled by the driver.

This flexibility also allows the TVM to work with a the widest range of device possible. For example, using this model it is possible to integrate FPGAs or fixed-function accelerators into Tornado by implementing a new TVM client.

This ability of Tornado is evaluated in Section 7.5 where it is demonstrated that – by using the TVM – it is possible to write a Tornado application once and execute in across thirteen different hardware accelerators. Moreover, for an idea of what the TVM enables Section 3.3.4 provides an example where a two stages of the same processing pipelines are randomly migrated between devices on each execution of the pipeline.

4.5.2 Asynchronous Operations

One of the most complex aspects of heterogeneous programming is handling asynchronous or concurrent execution. The TVM is designed for heterogeneous environments and has in-built support for managing asynchronous execution. It has three configurable operating modes:

- Synchronous (or blocking) where the TVM ensures that each bytecode has completed before executing the next. This mode is generally only used for debugging purposes.
- In-order asynchronous where the TVM allows all bytecodes to execute asynchronously on devices but assumes a first-in-first-out (FIFO) execution model on each device.
- Out-of-order asynchronous where the TVM executes bytecodes asynchronously and provides each device with dependency information to allow them to schedule tasks out-of-order.

The difference between each of these modes is the amount of work that can be overlapped. By overlapping work it is possible to reduce the total execution time of the coordination logic. This is a natural optimisation to perform in heterogeneous systems as there are multiple resources that can be utilised concurrently.

From the developers perspective it is very difficult to switch between these operating modes in low-level heterogeneous programming languages as changing between them sometimes requires a processing pipeline to be implemented multiple times. As a consequence, developers generally only support one of these operating modes. Later in Section 7.8.3 it will be shown how beneficial it is to have the ability to switch between these modes on a per-device basis.

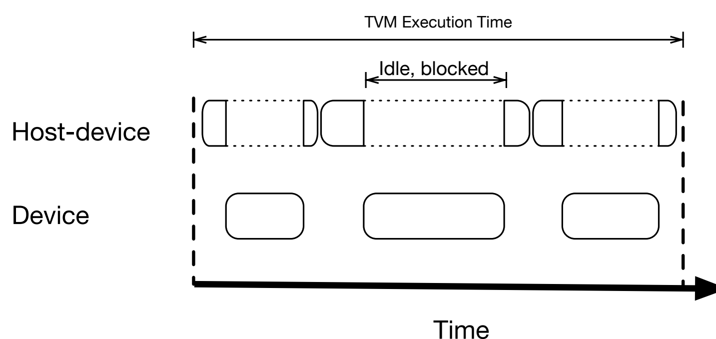


Figure 4.5: Example of synchronous execution. Here each time the TVM issues a bytecode to the device it will wait for the bytecode to complete before proceeding. As a result the TVM, and the rest of the application, is unable to make progress. This can result in a large amount of wasted time.

The TVM supports asynchronous execution by providing specialised bytecodes that help to track data-dependencies between tasks. Section 5.3 describes these bytecodes and how the Tornado Runtime System is able to automatically generate TVM bytecode that supports asynchronous execution.

The most basic mode, synchronous (shown in Figure 4.5) does not overlap any work and enforces a strict execution order: each bytecode must complete before the next is issued. The result is that the coordination logic will be executed in serial fashion. This mode is generally used for debugging purposes as the TVM is blocked whilst work is performed on the device. Thus, leading to a large amount of idle time in the TVM where no useful work can be performed.

By default the TVM will execute using an asynchronous in-order mode (shown in Figure 4.6). In this mode the TVM will try to issue bytecodes as fast as possible to the command queue on each device. The command queue will then be responsible for executing the queued operations. In this mode the TVM relies on the command queue to execute the queued operations in the order that they are added to the queue (in-order). By doing this the the TVM can avoid sending dependency information to the TVM client. This mode is useful because it allows the TVM and the application to progress whilst work is being performed on the device, i.e. the application code is overlapped with device code. This is the default mode for the TVM as legacy devices do not always support out-of-order execution.

Finally, some devices are able to support out-of-order execution of work (Figure 4.7). In this mode the TVM passes the command queue information about work

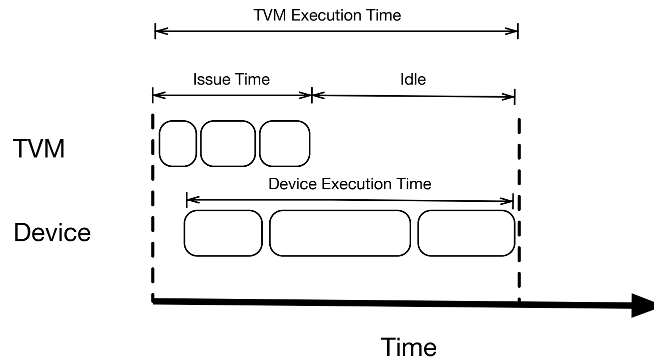


Figure 4.6: Example showing how the asynchronous in-order execution policy works. In this mode the TVM issues all bytecodes as early as possible to the TVM client where they are queued. This allows the TVM to avoid blocking and enables the application to make progress while code is executing on the device. Here the TVM client enforces a strict serial ordering of all the commands in its queue. Thus allowing all commands that are issued to the same queue to be executed in serial order.

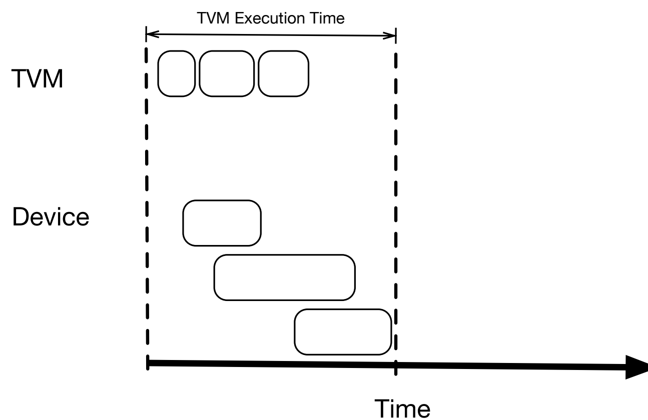


Figure 4.7: Using the asynchronous out-of-order execution policy the TVM will use the dependency information calculated by the Tornado Runtime System to allow tasks on the device to both: overlap in execution and execute out-of-order. The advantage of this policy is that this should provide the quickest way to execute each task schedule on the device.

```
void bar(float[] a, float[] b, float[] c) {
  schedule s0 {
    task t0 foo(a, b, c);
  }
  s0.execute();
}


bar(x,y,z);
bar(t,u,v);
```

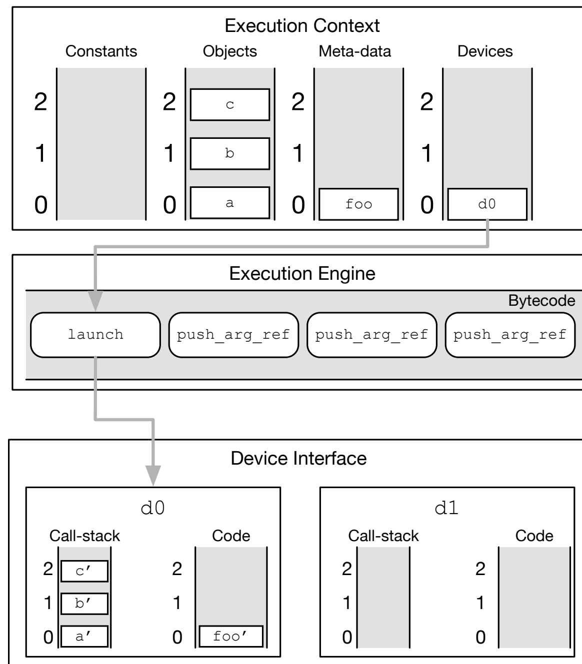
Listing 4.5: An example of a polymorphic task-schedule. One each invocation task-schedule `s0` is called with different arguments. In this situation the TVM is able to avoid continual re-compilation of the task by allowing the execution context to be updated.

and the dependency information that is calculated by the Tornado Runtime System (see Section 5.3). This allows the command queue to schedule work when all its dependencies have been satisfied. This allows some work to be executed earlier than it would be in the in-order mode. A good example is that data-transfers and computation can be overlapped in this mode – helping to amortise the cost of data-movement. The TVM itself does not track data-dependency information from the bytecode – this is done in the Tornado Runtime System. Instead it provides a special bytecode that can be used to append a handle to an asynchronously executing bytecode to a event queue. These queues are then passed to the TVM client when executing any bytecode that can utilise dependency information.

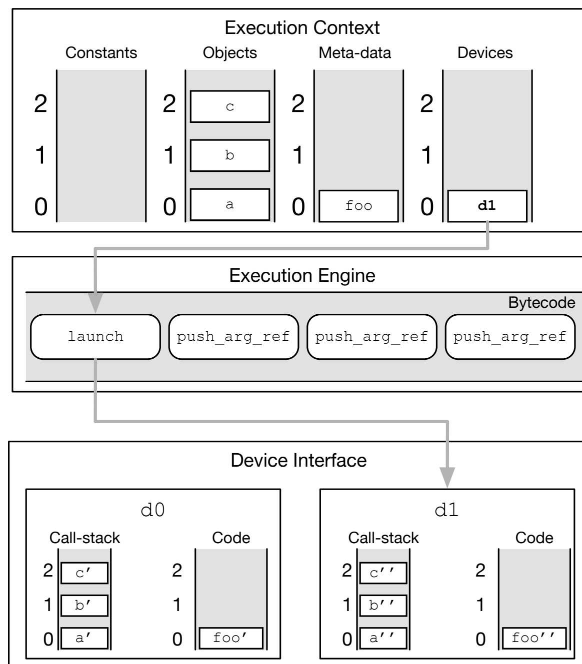
4.5.3 Dynamic Configuration

One of the most useful feature of Tornado is its ability to easily re-configure execution at runtime. As seen earlier in Section 3.3.4 and will be seen later in Section 7.8, dynamic configuration allows applications written in Tornado to be dynamically adapted to their execution environment.

To show the utility of dynamic configuration a complex example is given in Listing 4.5. In this example the parameters provided to task `t0` are defined by the invoker of `bar`. Hence, each time task `t0` is invoked `a`, `b`, or `c` may reference different variables. The TVM is designed to cope with this situation by accessing variables indirectly inside the execution context. Each time `t0` is invoked the application only needs to update the execution context before invoking the TVM. If variables have been seen before, the object cache will recognise them and reuse their existing allocations, otherwise, a new allocation will be created.



(a) Initial execution of TVM bytecode.



(b) Execution after slot 0 in the device list is updated.

Figure 4.8: The inner workings of the TVM showing how a task-schedule is migrated between devices. Notice that the `launch` bytecode references to value contained in slot 0 of the device list. Once this is changed, in (b), the TVM will automatically start issuing commands to device `d1`.

A more important example of dynamic configuration is provided in Figure 4.8 where the `foo` example is initially executed on device `d0` and then migrated in the next invocation of the TVM so that it executes on device `d1`. This example shows how all the examples in Section 3.3.4 work. Similar to changing parameter values, the application can also update the device list in order to migrate code and data to other devices. In this example, slot 0 of the device list has been changed to `d1`. The benefit to the developer in this situation is that all the complexities of the migration are handled automatically by the machinery inside the TVM. For instance, the object cache is used to ensure variables `a`, `b`, and `c` are translated into `a″`, `b″`, and `c″` on device `d1`. Similarly, a second version of `foo` is generated by the TVM client `foo″` for device `d1`. After the migration both the code and data persist on `d0` until it is invalidated by the TVM client. This normally occurs when a garbage collection occurs or the driver is reset. This feature means that if the execution is migrated back to device `d0` then there will be no need to copy all the data and code again.

4.6 Performance Model and Optimisation Criteria

Measuring performance in a heterogeneous environment is difficult. Most notably because there is a lot of work involved in executing code on a hardware accelerator and some of this work can execute concurrently. For instance, Section 4.5.2 has described the three operating modes of the TVM and how each mode is designed to vary the amount of work that is overlapped between the device and the application. The aim of this Section is to provide clarity on how performance inside the TVM is measured and define the optimisation strategy that Tornado will take.

4.6.1 Defining A Performance Model

In general the the execution time (ET) of a task-schedule running on the the TVM is modelled as:

$$ET = DET + \varepsilon \tag{4.1}$$

where the device execution time (DET) is the total amount of time taken to execute all tasks – both data-transfers and executing code – on all devices and ε is the overhead caused by the TVM. Figure 4.9 shows this visually. This metric can be measured directly by measuring the duration of the blocking call that executes the TVM.

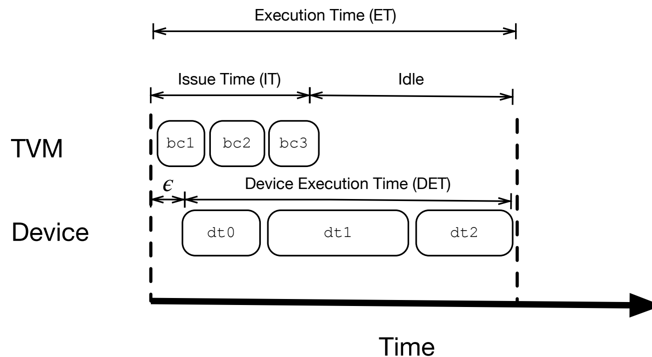


Figure 4.9: Performance Measurements in Asynchronous In-order Mode. The time to execute a task-schedule (ET) is modelled as the sum of the time taken to execute all tasks on the device (DET) – `dt0` to `dt2` – and some overhead (ε). The time for the TVM to process all bytecodes – `bc1` to `bc3` – is called the issue time (IT).

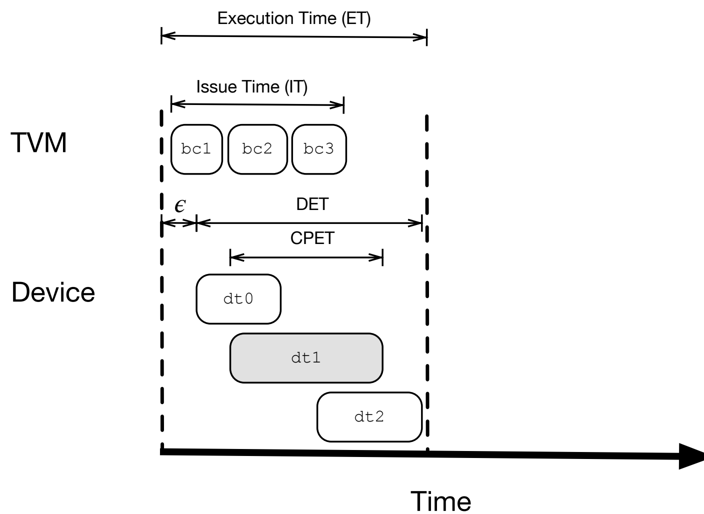


Figure 4.10: Performance Measurements in Asynchronous Out-of-order Mode. In the out-of-order execution mode the TVM is able to overlap the execution of device-side tasks – `dt0` to `dt2`. Compared to Figure 4.9 the ET is now much shorter as device-side tasks can overlap their execution. This also means that the DET is no-longer the aggregation of the execution times of each task. In this mode the key scheduling metric is the critical path execution time (CPET) in this example this is represented by the shaded `dt1` device-side task.

The Issue Time (IT) is the time taken by the TVM to process all bytecodes in a single execution context. As bytecode issue overlaps with the execution of device-side tasks it is not explicitly added into any performance model. However, it does provide a lower bound on performance of the TVM as the inequality in Equation (4.2) should always hold.

$$DET \geq IT \tag{4.2}$$

It is very difficult to measure or calculate the DET accurately in asynchronous execution modes because each device-side task may execute in an unspecified order and sometimes the execution time of a task may not contribute to the DET. Figure 4.10 shows the complexity of trying to measure the DET. As the DET cannot be measured or calculated directly an approximation is used instead.

In the asynchronous in-order execution mode the DET is estimated as the aggregate execution time of all device-side tasks. This is called the Sequential Device Execution Time (SDET) – as it assumes all tasks execute in a sequential manner:

$$SDET = \sum_{i=0}^{n} dtask_i \tag{4.3}$$

However, the SDET is inappropriate in the out-of-order execution mode as device-side tasks can overlap their execution. Therefore, a different approximation is used: the CPET – the Critical Path Execution Time. This measure assumes that device-side tasks are organised as a dependency graph and that the minimum possible execution time is the time it takes to execute the longest path (referred to as the critical path). The assumption made is that all tasks outside of this critical path have their execution overlapped by the tasks on the critical path and therefore their times do not contribute to the CPET. Conversely, the maximum execution time that can occur is when all nodes lie along the critical path – in this situation the CPET will be equal to the SDET.

The CPET is calculated by:

$$CPET = \sum_{i=0}^{n} x_i \cdot dtask_i \tag{4.4}$$

where x is a vector that represents each task on the critical path such that $x \in X | x = \{0, 1\}$. From this equation it is possible to know that the DET will always satisfy the following inequality:

$$CPET \leq DET \leq SDET \leq ET \tag{4.5}$$

The highest performance of the TVM occurs in asynchronous out-of-order execution mode when all the computation is overlapped perfectly with the critical path. If the CPET is substituted equation 4.2 it yields:

$$ET_{\lim \varepsilon \to 0} = CPET \tag{4.6}$$

Therefore, the scheduling efficiency of the TVM can be defined as:

$$E_{TVM} = \frac{CPET}{ET} \tag{4.7}$$

Similarly, the efficiency of the the TVM client can also be calculated as:

$$E_{DD} = \frac{CPET}{SDET} \tag{4.8}$$

4.6.2 Optimisations

In the TVM the DET term is always the largest contributor to the overall execution time – whether it is being estimated using either CPET or SDET. This means that to improve performance the DET should be minimised. There are two ways of doing this: (1) minimising the execution times of individual device-side tasks (the `dtask` term in Equation (4.4)), and (2) reducing the number of device-side tasks that contribute directly to the DET (the `x` term in Equation (4.4)). Generally, the former is achieved by improving the quality of generated code. To address the latter there are two options: (1) to maximise the number of device-side tasks that can execute concurrently, (2) or by eliminating redundant work. As (1) is generally fixed by the developer the only viable was to optimise execution is to eliminate redundant work.

One of most common redundant device-side tasks that can be eliminated are data transfers. For example, the bytecode used in Listing 4.6 contains some redundant data transfers. To automatically eliminate these transfers some extra information is required about the high-level task: a read-write set that specifies how parameters are used within the high-level task. For instance, a parameter can be either `read`, `write`, `read-write` or `unknown`. If the read-write set is known a task then the optimisation rules from Table 4.2 can be applied. These rules allow expensive bytecodes that are emitted before and after the `launch` bytecode to be replaced whilst also preserving

Access	Pre-launch	Post-launch
unknown	copy_in	copy_out
read	copy_in	-
write	allocate	copy_out
read_write	copy_in	copy_out

Table 4.2: Given a parameters access modifiers – `unknown`, `read`, `write`, or `read_write` – their pre- and post- launch data transfers can be replaced with the specified bytecodes to eliminate unnecessary data transfers.

```
      setup          1, 1 , 0          ; create new execution context:
                                       ;   - 1 device stack
                                       ;   - a 1 entry device list
                                       ;   - no event queues
foo_entry:
      begin                            ; start of executable bytecodes
      copy_in        0                 ; copy_in arg[0]
      copy_in        1                 ; copy_in arg[1]
      copy_in        2                 ; copy_in arg[2]
      launch         0, 0, 0, 3, 0     ;
      push_arg_ref   0                 ; push arg[0] onto device stack
      push_arg_ref   1                 ; push arg[1] onto device stack
      push_arg_ref   2                 ; push arg[2] onto device stack
      copy_out       0                 ; copy_out arg[0]
      copy_out       1                 ; copy_out arg[1]
      copy_out       2                 ; copy_out arg[2]
      sync                             ; block until all bytecodes in
                                       ; context are complete
      end                              ; terminate TVM
```

Listing 4.6: Example TVM Bytecode before the optimisation of data-movement.

correctness. By applying these optimisations to the example in Listing 4.6 produces Listing 4.7 that contains 50% fewer data transfers.

4.7 Summary

This Chapter presented the Tornado Virtual Machine (TVM). The TVM is implemented in a novel way – as a nested virtual machine (see Section 4.1) – that avoids the need for Tornado applications to be directly linked against the software driver for each hardware accelerator (this would be OpenCL in the context of this thesis). Thus, allowing Tornado applications to be distributed entirely as Java bytecode. This means

```
      setup          1, 1 , 0           ; create new execution context:
                                        ;   - 1 device stack
                                        ;   - a 1 entry device list
                                        ;   - no event queues
foo_entry:
      begin                             ; start of executable bytecodes
      copy_in        0                  ; copy_in arg[0]
      copy_in        1                  ; copy_in arg[1]
      allocate       2                  ; allocate arg[2]
      launch         0, 0, 0, 3, 0      ;
      push_arg_ref   0                  ; push arg[0] onto device stack
      push_arg_ref   1                  ; push arg[1] onto device stack
      push_arg_ref   2                  ; push arg[2] onto device stack
      copy_out       2                  ; copy_out arg[2]
      sync                              ; block until all bytecodes in
                                        ; context are complete
      end                               ; terminate TVM
```

Listing 4.7: Listing 4.6 after the rules from Table 4.2 have been applied.

that the TVM provides Tornado with a virtualisation layer that decouples the application from device specific software. This feature of the TVM is the primary reason why the Tornado application is able to be written once and executed across thirteen hardware accelerators in Section 7.5.

The TVM itself is an abstract machine that is designed to execute the coordination logic of heterogeneous applications. Executing this logic involves converting a stream of bytecodes into calls to the software driver of each hardware accelerator via a device interface (see Section 4.3.6). The specification of the bytecodes used is given in Section 4.4.

Next, the novel features of the TVM are described: virtualisation (see Section 4.5.1), the ability to support asynchronous operations (see Section 4.5.2), and the ability to dynamically configure an application (see Section 4.5.3). These features are the main reasons why Tornado is able to support code like that provided in Section 3.3.4 where tasks are being freely moved between hardware accelerators. Moreover, as will be seen in Section 7.8 dynamic configuration is a very powerful feature that allows an application to be tuned without having to be recompiled – in this Section we see that this relates to performance increases of between 14 and 17%.

Finally, Section 4.6 describes the performance model of the TVM. This is important as this sets out the optimisation criteria that is used by the Tornado Runtime System when optimising task-schedules in Section 5.3.

5 | Tornado Runtime System

One of the most noticeable issues with heterogeneous programming languages is the lack of an abstraction that allows a developer to express *where* code should execute. Depending on the type of language this can lead to different problems. For instance, in low-level languages, like CUDA or OpenCL, this leads to higher code verbosity as the developer has to manage more aspects of heterogeneous execution themselves. To a lesser extent, this is equally true for high-level languages, like OpenACC or OpenMP, which allow code to be executed on different devices but rely on the developer to optimise the data-flow between kernels.

To remedy this situation, Tornado employs a task-based programming model that provides the abilities to specify the locality of each task and combine multiple tasks to form a structured pipeline (which Tornado calls a task-schedule). Moreover, unlike other languages Tornado allows both of these to change at runtime – which is called *dynamic configuration*.

The Tornado Runtime System (TRS) sits between the user-facing API and the Tornado Virtual Machine. Its role is to map the high-level abstractions used by the API into TVM bytecode. For those familiar with programming language implementation the TRS can be considered to be a dynamic optimising compiler for coordination logic.

This Chapter describes both the TRS and how it is used to translate the Tornado API into TVM bytecode. The TRS is the component within Tornado that ties together the Tornado Virtual Machine and the Tornado Device-side Compilation Pipeline to implement the Tornado API. As such it is the component that is used to support much of the developer productivity features of Tornado – such as transparent data movement between devices and the out-of-order execution of tasks.

5.1 Architecture

The Tornado Runtime System (TRS) has five main components. Three are used to compile coordination logic into optimised TVM bytecode – the Tornado-IR Builder, the Optimiser and the TVM Bytecode Assembler – and two that aid the compilation of device-side code – the Sketcher and the Task Cache. Figure 4.5 illustrates the TRS architecture along with a typical execution flow.

1 The Tornado API is used to define a task-schedule – the code that coordinates the execution of tasks in a heterogeneous system. Internally, the API creates an Execution Context that captures the variables, meta-data and tasks referenced by the task-schedule.

2 The task-schedule is passed to the Tornado-IR Builder that converts the task-schedule into a high-level intermediate representation – called Tornado-IR – that represents the flow of control and data within a task-schedule.

3 Any computation tasks found by the Tornado-IR Builder are immediately sent to the device-side compilation pipeline.

4 The Sketcher is responsible for generating the high-level intermediate representation used by the device-side compiler inside the TVM client (in this case GRAAL-IR is used). The result is then inserted into the Task cache alongside the tasks meta-data.

5 Once the Tornado-IR is built, each task will have an entry in the Task Cache. By using the GRAAL-IR of each task the Optimiser can perform an interprocedural data-flow analysis to determine how data moves through the tasks contained within the task-schedule. Using this information the Optimiser can automatically insert intra-device data transfers and overlap the execution of independent tasks.

6 After optimisation, the Tornado-IR is converted into bytecode by the TVM Bytecode Assembler and stored in the Execution Context ready for execution by the TVM.

7 When either `execute` or `schedule` is called on a task-schedule the Execution Context is loaded into the TVM and executed.

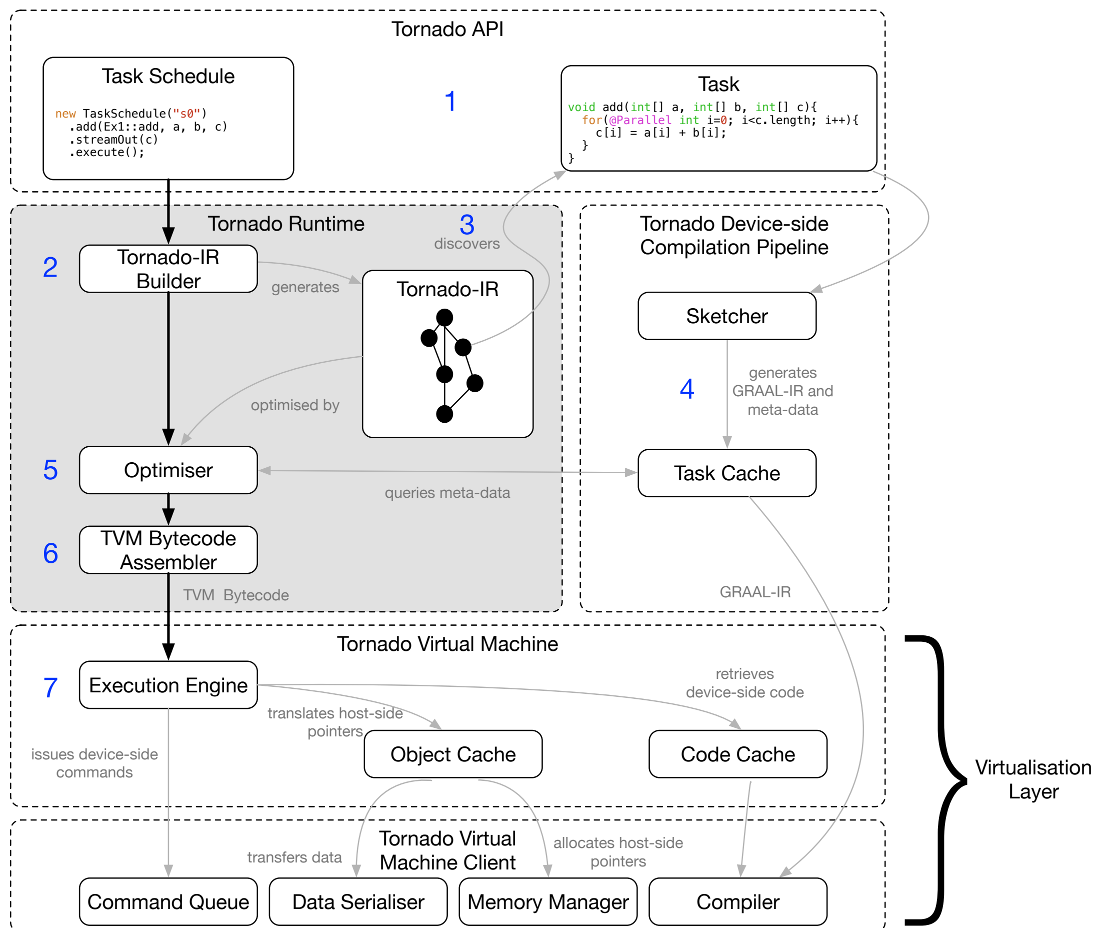


Figure 5.1: The Tornado work-flow: (1) a developer writes an application using the Tornado API; (2) when a task-schedule is executed the Tornado-IR Builder generates Tornado-IR; (3) each task that is contained in the Tornado-IR is sent to the Sketcher; (4) the Sketcher generates GRAAL-IR for the task and inserts it in to the Task Cache; (5) the Optimiser is run on the Tornado-IR to remove any redundant tasks; (6) the Tornado-IR is assembled into TVM cytecode; (7) the Execution engine executes the TVM bytecode.

Figure 5.2: Tornado-IR is a graph-based intermediate representation. It contains control-flow edges (black) and data-flow edges (grey) that connect a series of nodes. Nodes can be executable tasks (boxes) or references to variables (ovals).

5.1.1 Components For Compiling Coordination Logic

5.1.2 Tornado-IR Builder

The Tornado API (described in Section 3.1) allows developers to coordinate execution by modelling the interactions between tasks and variables. The Tornado-IR Builder takes the output of the Tornado API, a task-graph, and translates it into a high-level intermediate representation that represents the coordination logic as a series of low-level tasks with explicit control-flow and data-flow dependencies. The output of this process is a intermediate representation specific to Tornado which will be referred to as Tornado-IR and an example is given in Figure 5.2.

The builder starts building the Tornado-IR by inserting two nodes into an empty graph – one to signify the start and the end of the coordination logic. Next, each task from the task graph is inserted between these two nodes in program order. For each one, the control-flow edges are updated preserve this ordering. Once complete, the builder will add nodes to represent each parameter that is passed to a task. These nodes are connected to the tasks using data-flow edges. At this point, there is no need to worry about disambiguating variables – determining whether multiple tasks use the same variable – this will happen later. After all the nodes from the task-graph have

been inserted the Tornado-IR undergoes a lowering process so that all data-transfers are explicitly modelled as nodes in the graph. This means that by default all data transfers within a task-schedule are executed by default. There are two consequences to this approach: firstly, doing this ensures data is correctly moved between devices, and secondly, it introduces a large amount of redundant data transfers that are potentially expensive to perform continually.

5.1.3 Tornado-IR Optimiser

The Tornado-IR optimiser (or optimiser for short) provides Tornado with the ability to minimise the movement of data between devices. It takes the Tornado-IR created by the Tornado-IR Builder, removes any redundant work and produces an execution schedule (or ordered list of nodes). The optimiser aims to use the performance model set out in Section 4.6 to guide optimisation.

One of the problems that heterogeneous programming languages face is that they are unable to optimise the data flow between devices. Tornado can do this through the optimiser because – like a dynamic compiler – it can query the state of the running application. For instance, the optimiser can calculate exactly what variables are required by each task by inspecting the GRAAL-IR that is contained within the Task Cache. From this the optimiser can determine which variables are captured by the scope of the task-schedule. Next, a per task data-flow analysis can be performed on the GRAAL-IR to determine how these variables are accessed by each task: read-only, write-only, or read-write. From this analysis it is possible to eliminate redundant data transfers from the Tornado-IR. This optimisation is covered in more depth later in Section 5.3.

Moreover, Tornado is able to use the same technique to solve the problem of determining how many threads to execute. To do this the GRAAL-IR and task meta-data is retrieved by the optimiser from the task cache. The optimiser then locates the header of the parallel for-loop and uses a combination of constant propagation and partial evaluation to determine the iteration space of the loop. Note that the constants being propagated in this case are the parameters that will be passed in to the task. One of the benefits of this approach is that often the iteration-space is defined by the length of an array. By using this approach Tornado can use reflection to determine the iteration-space automatically opposed to forcing the developer to explicitly calculate the iteration-space. Once the iteration-space is known, then Tornado is able to automatically determine the number and dimensions of the thread-groups needed to execute each task.

The optimiser has been designed so that it is able to employ the same analyses as any other optimising compiler. This feature becomes useful when trying to minimise the cost of data-movement within the Tornado-IR. For instance, strength reduction is used to replace expensive data copies with simple data allocations and redundancy elimination is used to eliminate unnecessary data transfers.

The final responsibility of the optimiser is to determine an execution schedule for the nodes contained in the Tornado-IR. The scheduler is designed to try and overlap the execution of as many tasks as possible – according to the performance model in Section 4.6. A final schedule is generated by traversing the data-flow edges of the Tornado-IR to schedule asynchronous nodes as soon as all their data-dependencies are satisfied. Hence, they will be placed as early as possible in the final schedule and by doing this they are given the greatest possibility of overlapping their execution with other nodes.

5.1.4 TVM Bytecode Assembler

The TVM bytecode assembler takes the node schedule generated by the Tornado-IR optimiser and encodes it into TVM bytecode. Once complete, the bytecode is inserted into the execution context; ready for it to be executed by the TVM. The full bytecode specification can be found in Section 4.4 and an example can be found in Listing 5.1 later in Section 5.2.

5.1.5 Tornado Device-side Compilation Pipeline

Tornado employs a novel dynamic compiler infrastructure that is designed to work efficiently where a single input program needs to be compiled into multiple unknown machine languages. For this reason, it employs a split-compilation approach where the generation and optimisation of the device-side compiler intermediate representations (IR) are separated away from their code generation stage. By sharing the same compiler IR across multiple code generators Tornado is able to quickly parse and re-optimise the same task for each unique accelerator. At the moment Tornado uses GRAAL-IR [39] as its primary device-side compiler IR.

5.1.6 Sketcher

The sketcher has a single job, to turn Java bytecode into a device-side compiler IR. In the current implementation of Tornado The Sketcher is triggered asynchronously as

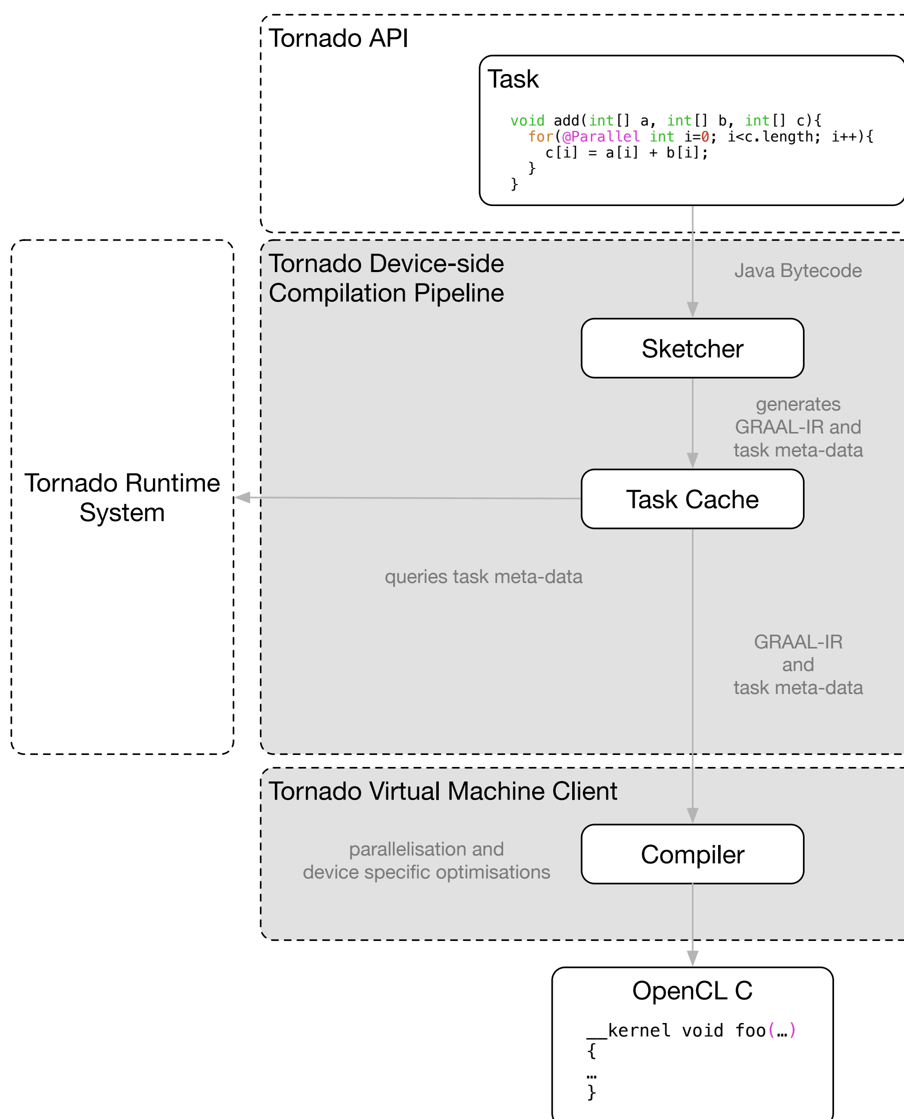


Figure 5.3: Each time the Tornado Runtime System discovers a task within a task-schedule that it wants to execute it sends the task to the device-side compilation pipeline. The Tornado device-side compilation pipeline is responsible for turning each task into machine-code for a target device. If a task has not been seen before (i.e. it does not have an entry in the task cache) it is sent to the sketcher. Here the Sketcher will generate GRAAL-IR from the Java bytecode of the task and insert the result into the task cache. Later when a `launch` bytecode is issued, the Tornado Virtual Machine client that receives the command will request the GRAAL-IR of the task that it needs to compile. The TVM client will receive a copy of the GRAAL-IR that it is free to specialise for each device. In the current implementation of Tornado this GRAAL-IR is used to generate OpenCL C. The advantage of this approach is that many TVM clients can share the GRAAL-IR stored within the task cache – making it easier to dynamically re-compile tasks for new devices.

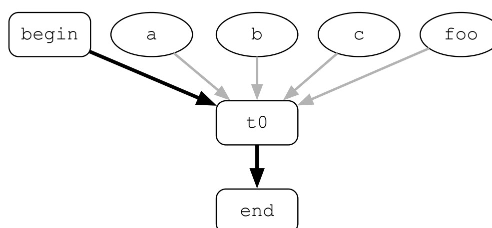
tasks are discovered by the Graph Builder and it generates the graph-based GRAAL-IR. It also has two important roles. Firstly, that it is responsible for handling any Tornado specific transformation on the GRAAL-IR. Typically, this is to implement any features defined by the Tornado API, like parallelisation or the Collections API – as described in Sections 3.2.1 and 3.2.2 respectively. Examples of this are the support for vectors and the support of built in functions. Secondly, that it needs to generate any information needed by the optimiser. Such as the read-write sets for each variable accessed by as task that are used as an input to strength reduction in Section 5.3.3 and calculating the bounds of its iteration space. The newly created GRAAL-IR is then inserted into the Task Cache along with a reference to task specific meta-data.

5.1.7 Task Cache

The Task Cache is designed to communicate the GRAAL-IR and task specific meta-data generated by the Sketcher between other components within the Tornado Runtime System. For example, the optimiser uses the Task Cache to obtain the read-write set that describes how each of the parameters are accessed by a task. More importantly, the Task Cache is accessible from the JIT compiler inside the TVM client. Allowing each TVM client to load the GRAAL-IR from the cache instead of having to build it from scratch each time it compiles a task. The advantages of this approach is that it makes it very quick to dynamically recompile a task. This becomes important to support dynamic configuration, where tasks can migrate between devices or a request is made to recompile a task to use a different parallelisation scheme.

5.1.8 TVM Client Compiler

Inside each Tornado Virtual Machine client is a finalising compiler that takes the GRAAL-IR that stored in the Task Cache and outputs machine code. This compiler is physically separated from the rest of the device-side compilation pipeline so that it can be specialised for a specific machine language or specific device. In the version of Tornado evaluated in Section 7.3 the client compiler is implemented using GRAAL and targets OpenCL C. A discussion on why this is the case can be found in Section 6.4.




```
task t0 foo(a, b, c)
```

Figure 5.4: Tornado-IR representing a single task `t0` that operates on parameters `a`, `b`, and `c`. Underneath is the task definition written using the informal specification of the Tornado API.

5.2 Coordinating a Single Task

The previous Chapter describes how an each high-level task is implemented using a series of low-level tasks running on the Tornado Virtual Machine. The focus of this Chapter is on describing how TVM bytecode is dynamically generated and optimised from the task-schedule abstraction introduced in Section 3.1. For instance, consider the example in Figure 5.4 that defines a single task using the Tornado informal specification.

Internally, the TRS will generate a Tornado-IR representation as shown at the top of Figure 5.4. The Tornado-IR represents the example as a graph is that is composed of different types of node; some of which represent tasks and others variables. These nodes connect via control-flow edges (black lines) and data-flow edges (grey lines). The execution order of the nodes in the Tornado-IR is found by traversing the control-flow edges – starting from the `start` node and traversing the control-flow (black) edges until the `end` node is reached.

Nodes in the Tornado-IR start at a high-level of abstraction – to model interactions between tasks and variables – and undergo an iterative lowering process. At each stage of this lowering process the Tornado-IR optimiser tries to replace high-level nodes with one or more nodes that have a lower-level of abstraction. Once lowering is complete, the Tornado-IR will model low-level interactions that take place within the Tornado Virtual Machine. In Figure 5.4, there is only one node that can be lowered: the one that represents the high-level task `t0` and during lowering it will be replaced by another that represents the launch of this task on a device. However, as the optimiser must preserve correctness it also has to insert six other nodes: a (`copy in`) and (`copy out`) for each variable. The output of the lowering stage is visible in Figure 5.5.

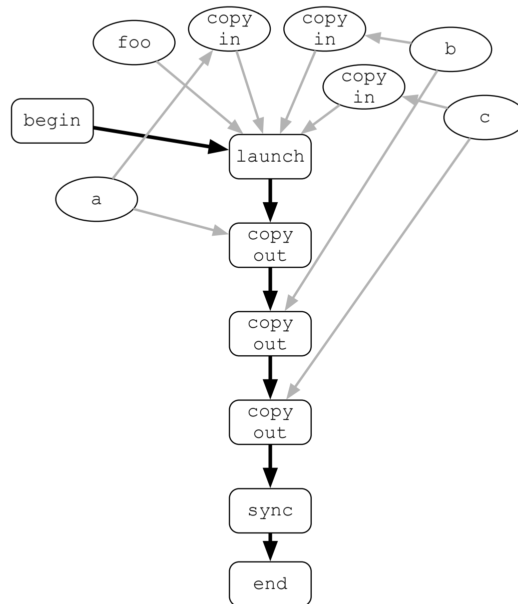


Figure 5.5: The only lowerable node in Figure 5.4 is `t0` that is replaced with a new node `launch` that represents code being executed on a target device. Notice how all data transfers – `copy in` and `copy out` – are now nodes in their own right and a `sync` node is added to ensure all data transfers have completed before the task can be considered complete.

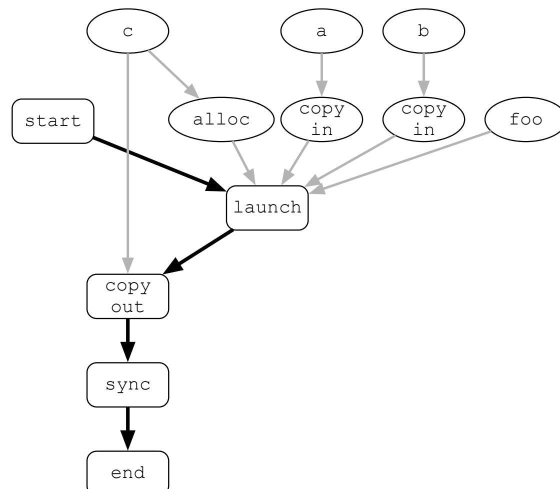


Figure 5.6: The Optimiser is able to apply the same rules as described in Section 4.6.2 to eliminate redundant data movement.

At this stage Tornado-IR is very conservative as it has been constructed to preserve the sequential consistency of each task-schedule and does not take into account the cost of data movement. As outlined in Section 4.6, there are two major factors which will influence how much time a task-schedule takes to execute. Firstly, the time attributed to moving data between devices and, secondly, the time taken to execute the tasks that sit on the critical path. To try to minimise the cost of moving data the next steps of the optimiser are to try and eliminate as many data transfers as possible from the Tornado-IR. One way to do this is to remove data transfers using variables that are either read-only or write-only. Read-only variables are never modified on the device-side and so don't need to be transferred back to the host after execution. Similarly, write-only variables are generated on the device-side and so do not need to be copied onto the device before execution. In fact this is the same optimisation that was discussed in Section 4.6.2 and by applying these optimisations the Tornado-IR shown in Figure 5.6 is produced.

After optimising data movement, the Tornado-IR now needs to be turned into TVM bytecode. The first step is to linearise the Tornado-IR into a schedule that satisfies all control and data dependencies. Typically, a schedule is generated by traversing the Tornado-IR via the control-flow (black) edges and at each node emitting all data-dependent nodes (linked with grey edges) before the control-dependent node. Figure 5.7 shows the result of the scheduling stage. At this point, the Tornado-IR can be sent to an assembler to produce TVM bytecode (Listing 5.1).

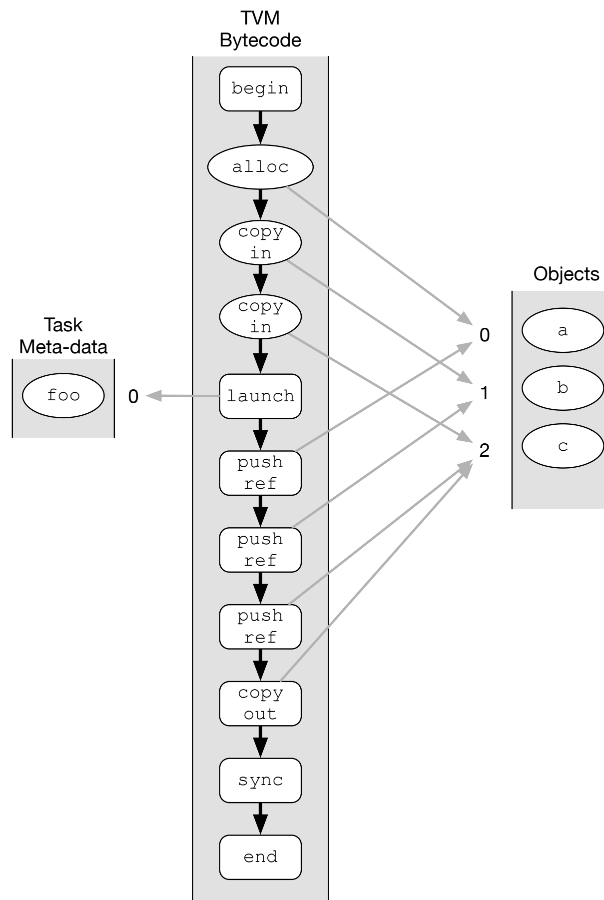


Figure 5.7: Linearisation and packing of the Tornado-IR.

```
      setup          1, 1 , 0          ; create new execution context:
                                       ;   - 1 device stack
                                       ;   - a 1 entry device list
                                       ;   - no event queues

foo_entry:
      begin                            ; start of executable bytecodes
      copy_in        0                 ; copy_in arg[0]
      copy_in        1                 ; copy_in arg[1]
      allocate       2                 ; allocate arg[2]
      launch         0, 0, 0, 3, 0     ;
      push_arg_ref   0                 ; push arg[0] onto device stack
      push_arg_ref   1                 ; push arg[1] onto device stack
      push_arg_ref   2                 ; push arg[2] onto device stack
      copy_out       2                 ; copy_out arg[2]
      sync                             ; block until all bytecodes in
                                       ; context are complete
      end                              ; terminate TVM
```

Listing 5.1: The TVM bytecode generated by the TVM Bytecode Assembler.

5.3 Coordinating Multiple Tasks

So far in this Chapter has only encountered examples of coordination logic that has a single task. Examples like this are common in synthetic benchmarks but rarely exist in real-world applications – like Kinect Fusion in Section 7.3. It is more likely that real-world examples will need to schedule multiple tasks and some of these may need to execute on different devices. By introducing this requirement – that execution of a task-schedule can span multiple devices – a problem is created because each task can now have a locality. This means that Tornado will have to ensure that data flows between these devices correctly. The role of this Section is to demonstrate how Tornado handles this situation and more importantly that by taking locality into account Tornado can produce much higher performing TVM bytecode.

In this Section Listing 5.2 will be used to explain how Tornado optimises to coordination of multiple tasks. Once the Tornado-IR builder finishes the translation, it will generate the Tornado-IR similar to that in top of Figure 5.8. Although this Tornado-IR is semantically correct, it does not provide many opportunities for optimisation. Therefore, to try and improve the situation the first step is to determine whether any data-dependencies exist between the parameter nodes – `p0` to – `p8`.

5.3.1 Variable Disambiguation

One of the most common optimisations is to avoid repeatedly copying the same variables onto the same device. To identify situations where this occurs the optimiser needs to be able to determine which variables are used by multiple tasks. As the task-schedule captures all parameters used by its constituent tasks the optimiser can check this list for duplicate entries – and where one occurs the corresponding parameter nodes are merged in the Tornado-IR. By applying variable disambiguation to the example the Tornado-IR in Figure 5.9 is produced. (Note that the parameter nodes have been renamed for additional clarity.)

At this stage, it is possible to use the Tornado-IR to identify data dependencies. In the example, three dependencies exist:

- A read-after-write (RAW) on variable `c` between tasks `t0` and `t1`.
- A read-after-read (RAR) on variable `a` between tasks `t0` and `t2`.
- A read-after-read (RAR) on variable `d` between tasks `t1` and `t2`.

```
schedule s0 {
  task t0 foo(a, b, c);
  task t1 bar(c, d, e);
  task t2 baz(a, d, f);
}
```

Listing 5.2: Example task-schedule that contains multiple tasks with data-dependencies.

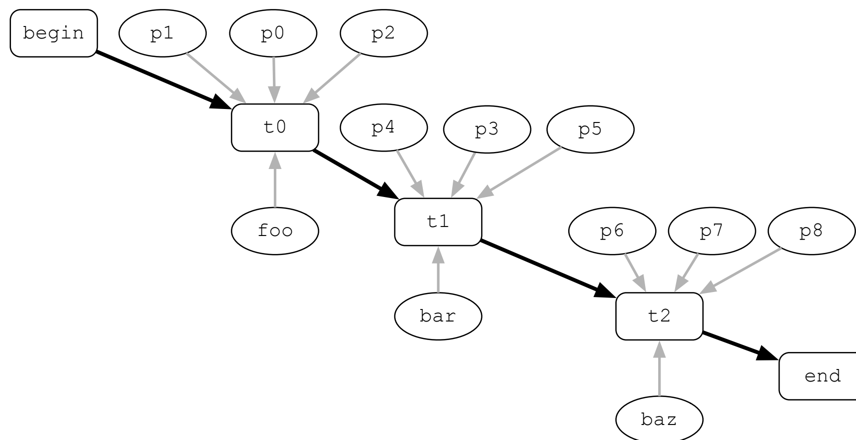


Figure 5.8: Tornado-IR representation of the task-schedule `s0`.

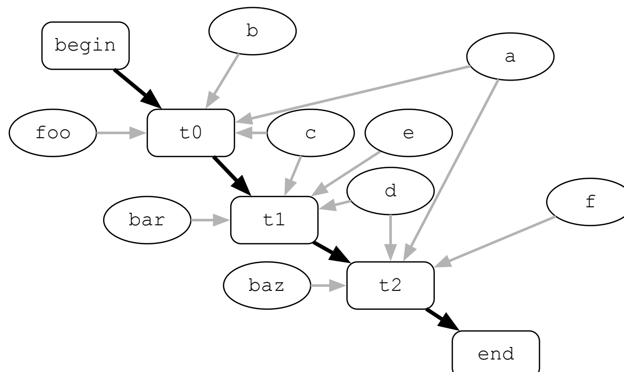


Figure 5.9: Figure 5.8 after variable disambiguation. Notice how the parameter nodes – `p0` to `p8` – are now associated with the parameters from Listing 5.2 – `a` to `f`.

The TRS needs to generate TVM bytecode that satisfies all these dependencies. However, it has the extra problem that each task can potentially execute on a different device. Therefore, the Tornado-IR needs to be updated to take this into account.

5.3.2 Handling Cross Device Data Flow

A typical limitation of heterogeneous programming languages is their inability to automatically optimise data movement across either a single or multiple devices. Tornado is unique as it is able to do this. It is possible in Tornado because the Tornado-IR can be used to model the flow of data between the host and device. This is done as follows.

Initially, the optimiser assumes that each task executes on a different device and so inserts a `copy in` node before each variable is used by a task and a `copy out` node after each task has executed. By doing this, the correctness of the coordination logic is preserved, allowing it to run correctly irrespective of where any of the tasks are executing. However, this is a very costly approach as it involves a lot of redundant data movement between devices. Therefore, the optimiser utilises two common analyses to minimise the amount of data movement required: strength reduction and redundancy elimination.

5.3.3 Strength Reduction

Strength reduction is an optimisation that aims to either replace the expensive bi-directional data transfers for each variable with less expensive uni-directional ones or eliminate them. For example, each variable has a `copy in` and `copy out` node associated with it. In principle, if it is known that one of these transfers is redundant then it can be eliminated. An obvious example of this is when a task does not modify a parameter – referred to as a read-only parameter. In this situation, the `copy out` node is redundant (as the parameter is unmodified) and can be removed. The other typical example is when a task overwrites the contents of a variable with new values (referred to as a write-only parameter). In this case, there is no value in copying data onto the device; hence, the `copy in` node can be replaced with an `alloc` node that only allocates space for the data.

For strength reduction to work, the optimiser needs to know how each parameter is modified when a task uses it. Typically, parameters are accessed either in a read-only, write-only or read-write mode. In Tornado this information is discovered by the sketcher and is made accessible to the optimiser via the task cache.

```
t0: { {a, b, c}, {READ, READ, WRITE} }
t1: { {c, d, e}, {READ, READ, WRITE} }
t2: { {a, d, f}, {READ, READ, WRITE} }
```

Listing 5.3: Read-write sets for Listing 5.2.

	a	b	c	d	e	f
t0	READ	READ	WRITE			
t1			READ	READ	WRITE	
t2	READ			READ		WRITE
Optimal Node To Use						
Pre-launch	copyin	copyin	allocate	copyin	allocate	allocate
Post-launch	-	-	copyout	-	copyout	copyout

Table 5.1: Data dependency matrix for Listing 5.3. Using the read-write sets from Listing 5.3 and the optimisation rules from Table 4.2 the it is possible to determine the most optimal nodes for the Tornado-IR in Figure 5.9 that minimise data movement.

This means that each task has a set of parameter accesses associated with it – referred to as read-write sets. Listing 5.3 shows the read-write sets generated by the sketcher for the example. What should be noticeable is that these read-write sets are represented using as two lists: a list of variables and a list specifying the access types. In Tornado, an object can either be:

- `READ` which means that it is left unmodified by the task
- `WRITE` which means it is overwritten by the task
- `READ_WRITE` which means that the task reads and writes the variables
- `NONE` when the variable is neither read or written by the task

Using these read-write sets, the optimiser can construct a dependency matrix to model the interaction between variables and tasks. The dependency matrix for Listing 5.3 is shown in Table 5.1. In this table, the rows of the matrix represent the number and types of access made to each variable, whereas the columns represent accesses to individual variables. As the rows are inserted in program order, by traversing each column, it is possible to both identify and categorise each data dependency. This table allows the read-after-read dependencies for variables `a` and `d` and the read-after-write dependency on variable `c` to be correctly identified.

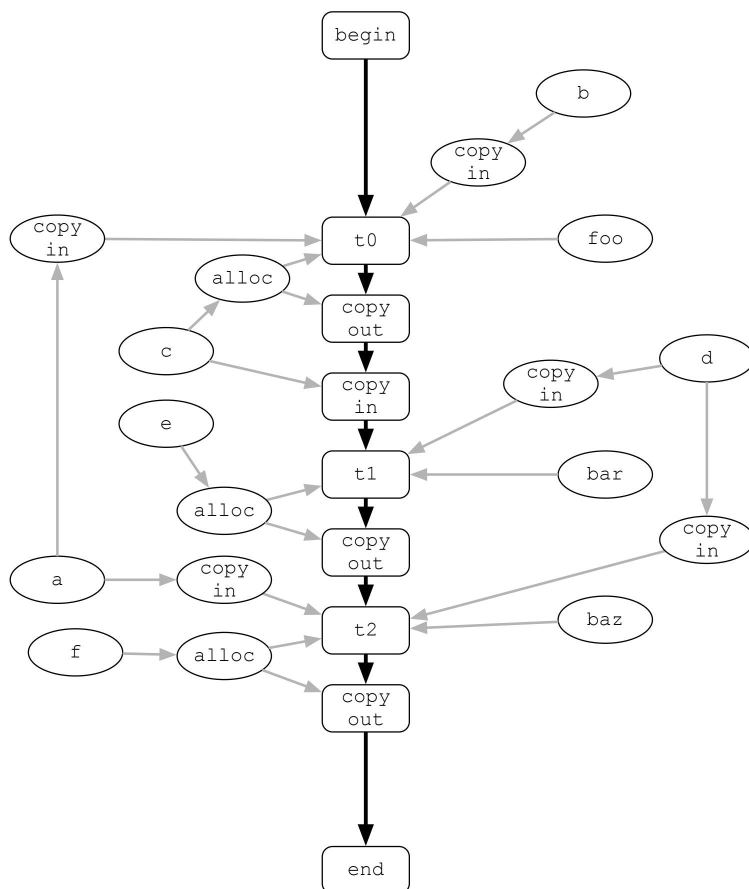


Figure 5.10: Figure 5.9 after data flow nodes are inserted and strength reduction is applied.

The resulting dependency matrix can be used to determine the optimal data transfers that should be performed for each variable. The bottom two rows of Table 5.1 shows the optimal nodes for the example and once strength reduction is applied it will produce the Tornado-IR shown in Figure 5.10.

5.3.4 Locality Disambiguation

At this point the Tornado-IR does not take into account the *locality* of any tasks. If all tasks are to execute on different devices then this Tornado-IR would be optimal. However, if all these tasks are to be scheduled on the same device there are more opportunities for optimisation as data does not need to be moved between tasks that execute on the same device. Therefore, in the same way that the optimiser is able to inspect the parameter list of the task-schedule it is also able to inspect the meta-data

Figure 5.11: Figure 5.10 after locality disambiguation is applied. Notice that the `copy in` node linking `c` to `t1` is eliminated.

of each task to identify where it is to execute. In the example, it is assumed that all tasks are co-scheduled on the same device, and by doing this it is possible to eliminate the `copy in` node that copies the modified version of `c` from task `t0` onto the device where `t1` is located. The resulting Tornado-IR is shown in Figure 5.11.

5.3.5 Redundancy Elimination

Once locality has been factored into the Tornado-IR, it is safe to start re-factoring the Tornado-IR to remove unnecessary nodes. Here the aim of redundancy elimination is to find any read-after-read (RAR) data dependencies that exist within the Tornado-IR and eliminate the unnecessary data transfer. There are two such dependencies in the example: variable `a` is read by both tasks `t0` and `t1`; and variable `d` is read by both tasks `t1` and `t2`. Currently, both of these variables have two `copy in` nodes that

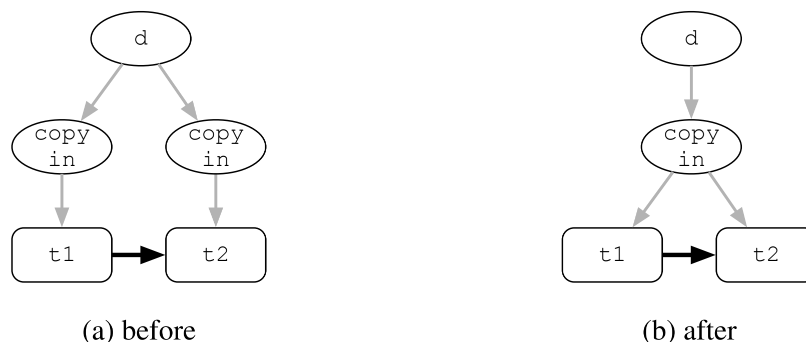


Figure 5.12: As both copy in nodes in (a) have the same input (or transfer the same variable to the device) they can be merged into a single node – shown in (b).

copy them onto the same device. Therefore, the tasks can share the output of a single copy in node. The transformation is shown in Figures 5.12a and 5.12b. The benefits of redundancy elimination is that all duplicate data transfers will be removed from the generated bytecode – as these are typically costly its performance should increase also.

5.3.6 Lowering

After applying redundancy elimination the Tornado-IR contains the absolute minimum number of data transfers that are required to preserve the sequential consistency of the task-schedule. At this point, the focus of the optimiser shifts from minimising data movement onto generating the most optimal TVM bytecode possible. Therefore, it needs to lower the level of abstraction of the Tornado-IR again through a lowering process. The results of applying redundancy elimination to the Tornado-IR is shown in Figure 5.13. Here the only difference to notice is that the task nodes have been replaced with launch nodes.

5.3.7 Out-of-order Execution

The final optimisation that is to be applied to the Tornado-IR is designed to enable the overlapping of task execution. As seen in Section 4.6 overlapping task execution can be one of the most profitable optimisations that can be performed on a task-schedule. The principle idea behind this is that the cost of executing expensive tasks can be amortised by overlapping their execution. The Tornado Virtual Machine has built-in functionality for handling asynchronous execution – providing the potential to overlap: multiple

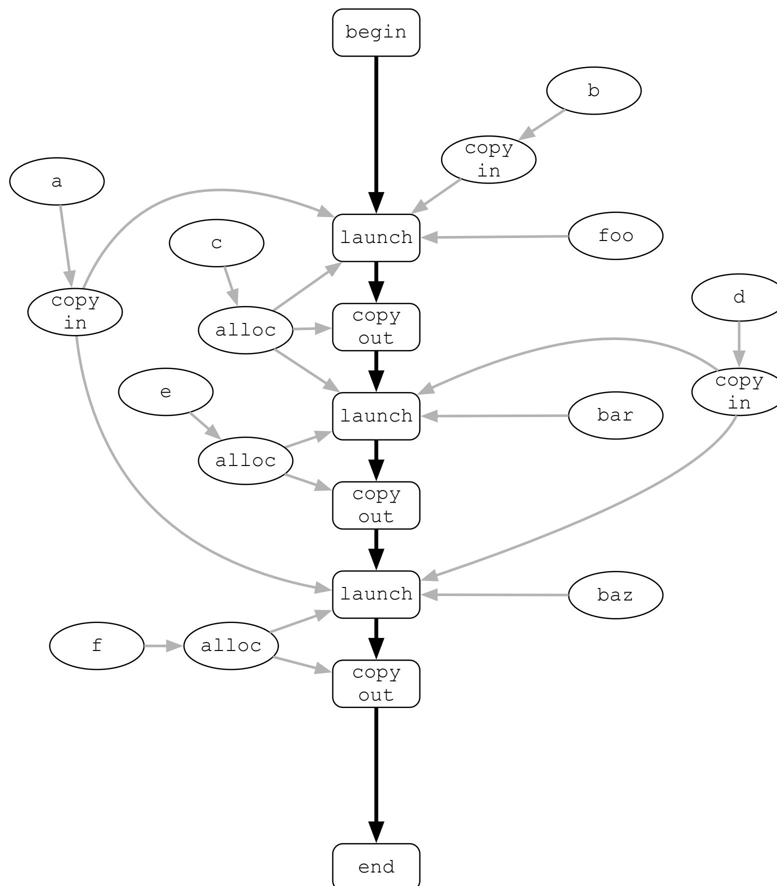


Figure 5.13: Figure 5.11 after applying redundancy elimination and strength reduction. Notice that there are fewer data transfers within the Tornado-IR.

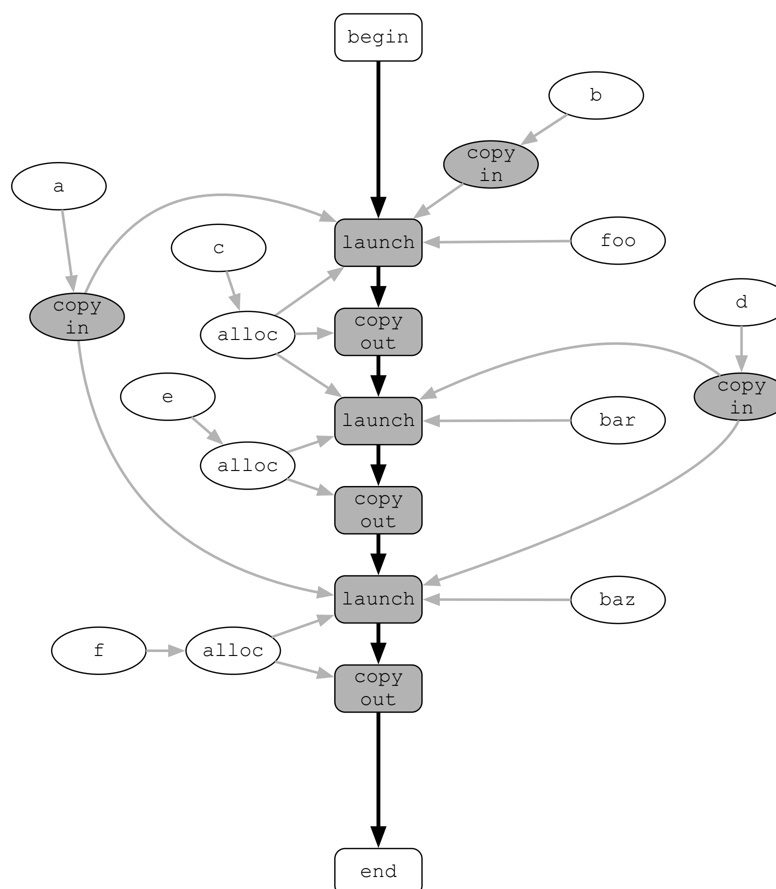


Figure 5.14: Tornado-IR nodes that are capable of asynchronous execution from Figure 5.13 are highlighted in grey.

data transfer tasks; multiple computation tasks; or both data movement and computation tasks. However, to make use of this functionality, the optimiser needs to embed dependency information into the generated TVM bytecodes to preserve correctness.

Figure 5.14 highlights the Tornado-IR nodes that are capable of being executed asynchronously. In this example, it is not possible to perform any computations out-of-order due to their control-flow dependencies. However, there is scope for overlapping the `copy in` and `copy out` nodes. To do this, a bytecode needs to be generated bytecode to utilises the event queue functionality in the TVM.

For this to happen the optimiser has to do two things: (1) create an event queue for each asynchronously executing task, and (2) to ensure that the correct dependencies are added to these queues. The Tornado-IR transformation used to achieve this is explained in Figure 5.15 that shows how task `t0` and its dependent data transfers are updated for asynchronous execution. The general process for this transformation is to

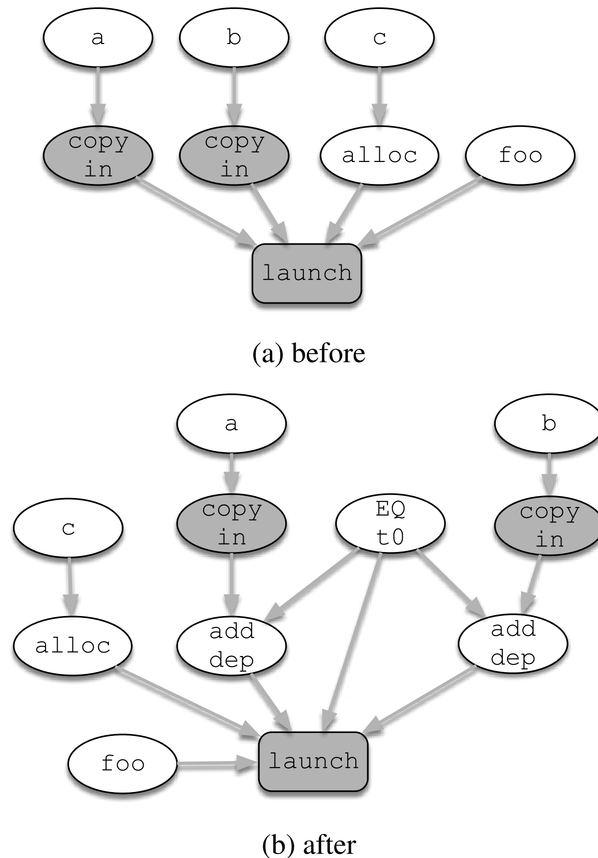


Figure 5.15: The transformation that needs to be applied to the Tornado-IR to allow nodes to execute asynchronously. Notice that an event queue – `EQ t0` – is created to hold the event handles of the two `copy in` nodes used to transfer `a` and `b`.

generate an event queue (represented as a `EQ` node) for each asynchronous (grey) node in the Tornado-IR. Once the queue exists, then an `add dep` node can be inserted on each asynchronously executing input edge (grey lines). The purpose of this node is to append the prerequisite task to the event queue.

5.3.8 Scheduling

Once all optimisations are applied the optimiser needs to convert the Tornado-IR into a form that is suitable for the graph assembler. To achieve this goal the scheduler needs to serialise the Tornado-IR – ensuring that it preserves all data and control-flow dependencies – into a list of nodes.

The algorithm for doing this traverses the Tornado-IR along the control-flow edges (black) from the `start` node until it arrives at the `end` node. It tries to schedule nodes

as early as possible to provide them with more opportunities to exploit asynchronous execution. Each time the algorithm arrives at a new fixed node (rectangle), the scheduler traverses all the floating nodes (ovals) and emits any that have their dependencies satisfied. Processing will continue until no more floating nodes can be scheduled and at this point, the fixed node is emitted, and the algorithm moves to the next fixed node along the control-flow edge.

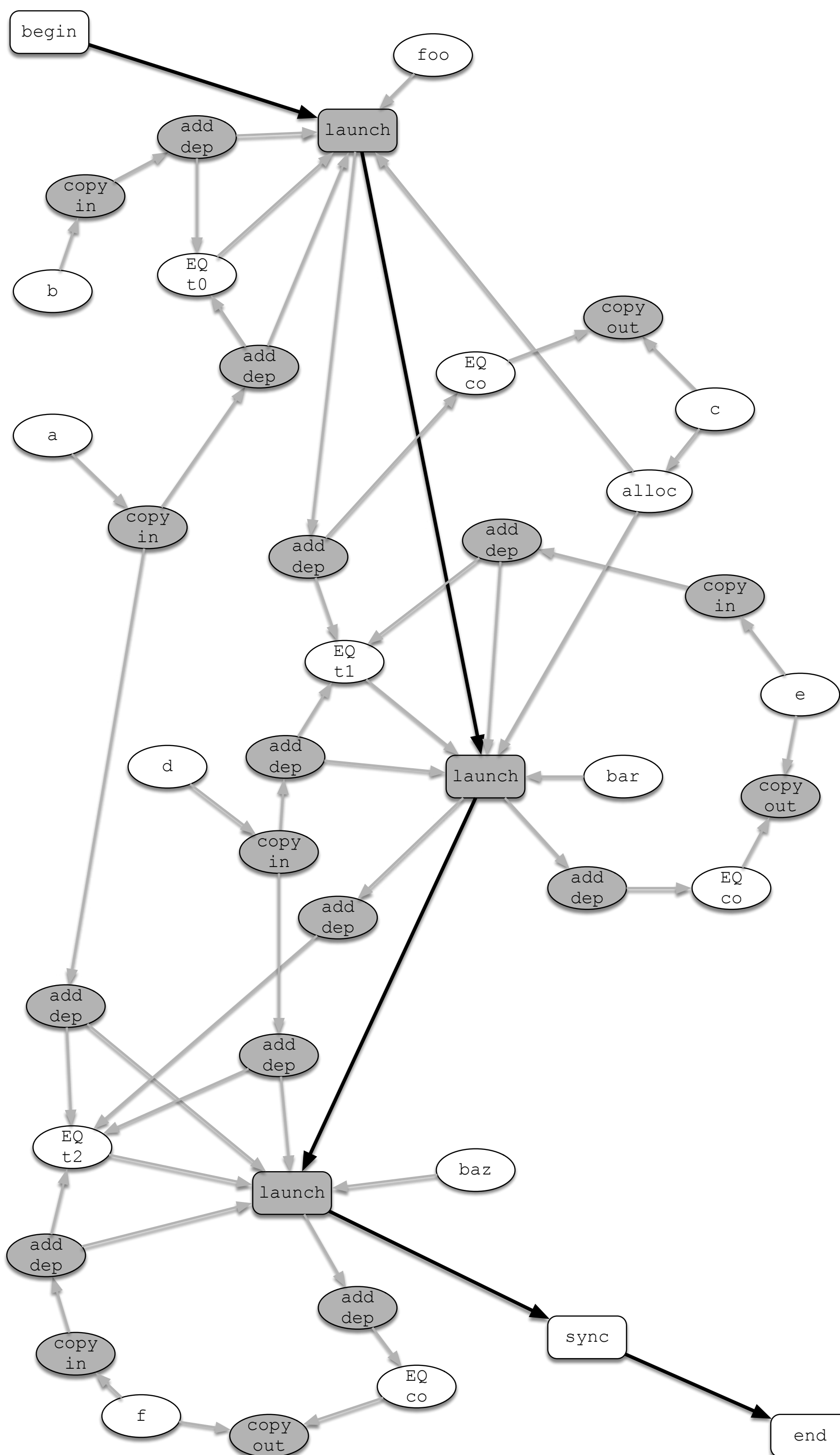


Figure 5.16: The GRAAL-IR from Figure 5.14 after being transformed for out-of-order execution. Notice the addition of event queues `EQ` and dependency tracking `add dep` nodes.

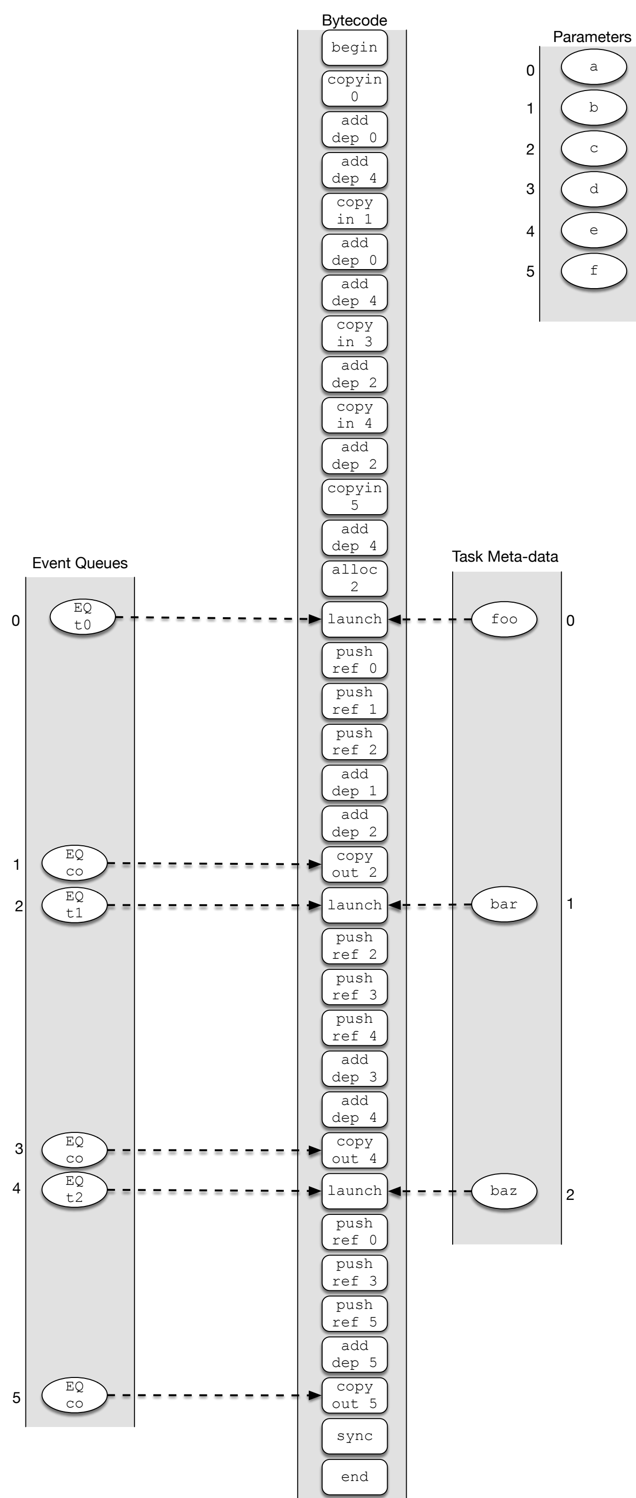


Figure 5.17: The GRAAL-IR from Figure 5.16 after being linearised by the scheduler. Notice how all the the values from the Tornado-IR are packed together to form the execution context that is to be sent to the TVM.

```
schedule s0 {
  task t0 foo(a, b, c);
  task t1 bar(c, d, e);
  task t2 baz(a, d, f);
}

schedule s1 {
  task t0 foo(a, g, h);
  task t1 bar(h, i, j);
  task t2 baz(a, i, k);
}
```

Listing 5.4: Two example task-schedules.

```
t0.execute();
t1.execute();
```

Listing 5.5: By executing `t0` and `t1` in a back-to-back fashion can be costly as they share a data-dependence – `a`. In this situation a conservative approach would be to always synchronise `a` with the host between the two calls. This can be costly in terms of data movement as it will generate two redundant data transfers.

5.4 Optimising Intra-task-schedule Data Movement

After discussing how complex multi-device task-schedules are optimised within the Tornado Runtime system the focus of this Section is to describe how Tornado is able to optimise data movement between different task-schedules. This is a optimisation that is relied on heavily in Kinect Fusion – see Section 7.3 – as it executes the same task-schedules multiple times. The key to this optimisation is eliminating the need to synchronise variables at the end of every task-schedule. Listings 5.5 to 5.7 show three situations where this occurs in practice.

Listing 5.5 shows two task-schedules that share a common variable `a`. In this example, the problem is that the TRS generates a TVM bytecode instruction to transfer `a` to the device in both task-schedules.

This becomes problematic when the second task-schedule is executed `t1` as variable `a` is already present on the device. Therefore, the extra data transfer is redundant. This problem becomes clearer in Listings 5.6 and 5.7 where each task-schedule executes multiple times. After the first iteration of the loop it becomes wasteful to transfer in any of the read-only variables.

This problem cannot be solved in its entirety through the TRS as the optimiser is not aware of what task-schedules execute before or after the one under compilation. The TVM has been designed to help remedy these situations by providing support for

```
for(int i=0;i<n;i++){
  t0.execute();
}
```

Listing 5.6: In a similar vein to Listing 5.5 this example can also produce redundant data transfers. Here the data transfers are generated as there is a potential data-dependency that spans loop iterations.

```
for(int i=0;i<n;i++){
  t0.execute();
  t1.execute();
}
```

Listing 5.7: To produce even more redundant transfers than Listing 5.6 it is possible to add data-dependencies between two task-schedules that execute in the loop body.

eliminating these extra data transfers.

The component that does this is the *object cache* (see Section 4.3.7) that is used to track both the state and location of variables that are used by task-schedules. To utilise the object cache, the TVM provides new bytecodes – such as `cond_copy_in` – that execute conditionally depending on whether the variable is already present on the target device or not. By using these bytecodes, it is possible to eliminate the redundant data transfers that the second invocation of the task-schedule or subsequent task-schedule initiates.

At the TVM object cache provides the ability to suppress these redundant transfers, the focus of this Section turns on to how these bytecodes are used. For instance, making all data transfers conditional is not an option as no new data will enter the device after the first execution of a task-schedule irrespective of whether the variable has been updated on the host-side or not. Hence, the greatest problem now is differentiating between variables that are modified by the host-side code in between executing task-schedules and those that are not. A problem that cannot be solved by the TRS, due it only having knowledge of the coordination logic and not the host-side application. Therefore, this issue must be addressed via the user facing Tornado API.

In the Tornado API, developers have the ability to mark variables as streaming. For incoming variables, it ensures that the cache entries are invalidated before the task-schedule begins executing. Therefore, each variable is copied to the device at the beginning of the task-schedule. This solves the problem of redundant transfers where a read-after-read data dependency exists between two task-schedules. However, these bytecodes do not help in the situation where a read-after-write data dependency exists

between two task-schedules.

Typically, to preserve the correctness of the application any variable that has been modified on the device should be transferred back to the host before a task-schedule completes. Another way to view this is that all 'dirty' variables contained within the object cache need to be written back to the host. Naturally, as this involves data movement, this is a costly operation to perform frequently.

However, to minimise its cost, it is sometimes possible to avoid synchronising certain variables. One typical case is where variables act as intermediate storage between two tasks – i.e. a read-after-write or write-after-write dependencies. If these variables only exist within the scope of the coordination logic, there is no requirement to synchronise them. Additionally, the same type of dependencies may exist at a higher-granularity. For example, a task may produce data for a subsequent task to consume. In this situation, it is beneficial to suppress any synchronisation of variables as they might be transferred immediately back.

At present Tornado cannot optimise all data movements that exist between task-schedules as it does not have enough information about the flow of data within the host-side application. The only way for this to happen would be to integrate the Tornado API directly into the host programming language (in this case Java). This integration would make it possible to check the scope of variables and understand what happens between task-schedule executions. Unfortunately, this is not the case in Tornado, and another option is required. Therefore, developers are also allowed to execute task-schedules using a more relaxed set of semantics that allows them to specify what variables to synchronise manually.

A final problem with the object cache is handling variables that are seldomly updated on the host-side. In this situation, it does not make sense to mark the object as streaming. Therefore, developers also have an explicit API function for invalidating cached variables.

5.4.1 Multi-device Task Schedules

A critical feature of Tornado is that it does not limit developers to using a single device to execute a task-schedule. Instead, each task can run on different devices. Although this enhances productivity for developers managing execution in such a distributed environment becomes complex, especially once caching has been introduced. Therefore, the object cache employs a caching protocol to ensure the consistency of variables that exist in multiple caches.

5.4.2 Object Caching Protocol

A caching protocol is necessary to provide a guarantee of consistency in cases where objects exist on multiple devices and to minimise the volume of data movement involved in multi-device applications. To do this, Tornado employs the MOESI cache coherence protocol to work between object caches on different devices. In order to utilise this feature, bytecodes that initiate a data transfer are augmented with two new operation modes: sharing and caching. The sharing mode determines whether the requesting device needs exclusive access to modify the data or not and the caching mode enables data transfers to be served from the cache or not. As the real-world application used in Section 7.3 does not execute multiple-tasks on multi-devices it is unnecessary to understand how object caching works.

5.5 Summary

This Chapter has described the Tornado Runtime System (TRS). The role of the TRS is to turn task-schedules – the coordination logic – of an application into bytecode that can be executed on the Tornado Virtual Machine (TVM). A breakdown of the key components and the work-flow of the TRS was discussed in Section 5.1.

The TRS acts like a dynamic compiler: taking a task-schedule as input, converting it into an intermediate representation – called Tornado-IR, optimising the Tornado-IR, and finally, emitting bytecode. An example of how this happens is provided in Section 5.2. One of the novel aspects of the TRS is that it has two compilation pipelines: one for the coordination logic (Section 5.1.1) and another for the computation logic (Section 5.1.5). These pipelines allow Tornado to both produce high performance device code and analyse the characteristics of the computation logic using a component called the Sketcher Section 5.1.6. The role of the Sketcher is to generate an intermediate representation of the computation logic early – in this case GRAAL-IR is used. From the GRAAL-IR the TRS can determine how the computation logic accesses its variables – either as read-only, write-only, or read-write. Information that can be used to optimise the movement of data within the coordination logic. Thus, providing the TRS with one of its most important features: the ability to support and optimise complex processing pipelines that span multiple devices. A step-by-step example of the way this is done is provided in Section 5.3. An example that illustrates how data movement within a complex task-schedule can be either reduced or eliminated entirely.

6 | Tornado Virtual Machine Client

The Tornado Virtual Machine Client (TVMC) is the component that implements all device specific functionality. It is responsible for: copying data to and from the device; managing the device-side memory; and generating machine-code for the device. Currently, Tornado has a single TVM client that is implemented using OpenCL. As a consequence most of the functionality required by Tornado is implemented by OpenCL. The only component that needs to be created is a Just-In-Time (JIT) compiler that turns Java bytecode into OpenCL C code.

One of the principal issues faced when implementing the OpenCL TVM client is that the JIT compiler needs to be designed to target OpenCL C. This is problematic as the programming language used to write tasks – Java – has a much higher level of abstraction than the ones traditionally used for this task. Remember that one of the main challenges in Section 1.1.5 is to understand how different abstraction affect the performance of the code that can be generated. Therefore, the goal of this Chapter is to describe how different features of the Java programming language are implementable on a hardware accelerator. Perhaps, the best place to start is to disambiguate what is meant by “compiling” or supporting Java on a hardware accelerator.

Java is an object-orientated programming language that has become iconic due to its write-once-run-everywhere philosophy. Java developers write their application once and can run it across a range of different machines irrespective of either the operating system or processor architecture used. Naturally, the last statement needs the caveat that developers can write platform specific code – especially when using operating specific functionality or calling native libraries – but this is a choice made freely by the developer and not one that is imposed by the language.

Portability of Java applications is made possible through the implementation of the language; primarily, that it is a virtual machine (VM) based language. This means that, by design, Java applications are not compiled into or distributed as binaries. Instead, Java applications are compiled into an intermediate form – called Java bytecode – that

is executed by a virtual machine. In the case of Java, this is the Java Virtual Machine (JVM).

It is the responsibility of the JVM to take the bytecode and translate it into low-level machine operations. Many misconceptions about the Java language stem from misunderstandings about how this process works. Historically, the Java Virtual Machine started out as a pure interpreter which led to Java becoming synonymous with low performance. This poor performance was simply because the JVM translated each bytecode sequentially into a series of high-level operations. Typically, this involves multiple operations to push and pop variables to and from the stack and another that represented the actual operation. However, modern implementations of the JVM, such as HotSpot[78], contain high-performance dynamic compilers [78, 96] that compile bytecode into machine code. Although, this approach incurs the extra overhead of the compilation process the generated machine code is very efficient and can be run multiple times – amortising the cost of compilation. Tornado just extends this prior art so that it can generate machine code for multiple devices within he same instantiation of the JVM.

Tornado is designed to extend the dynamic compilation framework that exists within the Java language so that it can compile for multiple devices. The key to Tornado's ability to compile for multiple devices is its capacity to compile bytecode dynamically. Although Java is not the first [36, 51] or the only programming language to use bytecode, it is arguably the most used [20, 80]. Bytecode is designed to be a form of intermediate representation (IR). Consequently, bytecode retains a lot of the high-level information from the source code to enable it to be input into a compiler. By using bytecode, some issues hindering the progress of other heterogeneous programming languages are instantly solved – like the need for a separate source file to be used to store device-side code and the ability to compile third party code for use in device-side code. Hence, enabling a Java application to run on an accelerator requires a compiler to work from Java bytecode. The upshot of this is that Tornado is easily compatible with any language that targets the JVM.

What is also important to note is that there is no need to support every single Java bytecode on hardware accelerators – as in some cases it is simply not necessary or sane to support all bytecodes on the hardware accelerator. Often the expectation is that all Java bytecodes need to be implemented for a framework to be useful – this is not the case. Here, it is good to be reminded of the bigger picture: Tornado accelerates applications by exploiting the *specialism* of *limited purpose* processors. Hence, it is not

```
public static void one(int[] a) {
  for (int i = 0; i < a.length; i++) {
    a[i] = 1;
  }
}
```

Listing 6.1: A simple counted loop in Java.

a goal (or requirement) to support all of the Java language on a hardware accelerator. In the rest of this Chapter a case study is provided to show how a dynamic compiler, like GRAAL, is able to generate high-performance code for heterogeneous hardware from idiomatic Java code.

6.1 Overview

To show how the Tornado compiler works, this Section will take some simple Java code, Listing 6.1, and describe the steps required to turn it into machine code. In Tornado, this process starts with Listing 6.2 – the Java bytecode generated when the developer compiles their application with the Java compiler (usually `javac`) – and finishes after it is transformed into OpenCL C code.

Currently, the Tornado dynamic compiler employs two key technologies: GRAAL and OpenCL. GRAAL [39] is an open-source dynamic compiler. It is used because of its close integration with the HotSpot JVM and as it is an industrially constructed compiler. OpenCL is an industry standard framework for heterogeneous programming and has been selected as the code generation target to allow Tornado to work with the widest range of devices possible. This will give Tornado the ability to execute code across the thirteen hardware accelerators listed in Table 7.1. To create the TVM client compiler the existing GRAAL compiler is augmented so that it is able to generate OpenCL C. Ultimately, this means that the OpenCL driver is responsible for the machine code that runs on the target device.

The first step of compilation is to parse the Java bytecode and generate a graph based IR – GRAAL-IR. Figure 6.1 shows the GRAAL-IR that has been constructed by GRAAL for Listing 6.2. Here the GRAAL-IR can be thought about simply as a set of nodes connected by control-flow (red) or data-flow (blue) edges. Notice that the GRAAL-IR has different types of node. For example, the dark orange nodes represent placeholders for the start and end of basic blocks; the red nodes represent control structures; and the light blue nodes are either arithmetic or relational operators. In

```
public static void one(int[]);
    descriptor: ([I)V
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
       3: aload_0
       4: arraylength
       5: if_icmpge     18
       8: aload_0
       9: iload_1
      10: iconst_1
      11: iastore
      12: iinc          1, 1
      15: goto          2
      18: return
    LineNumberTable:
      line 9: 0
      line 10: 8
      line 9: 12
      line 12: 18
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          2      16     1     i   I
          0      19     0     a   [I
```

Listing 6.2: Listing 6.1 compiled into Java bytecode.

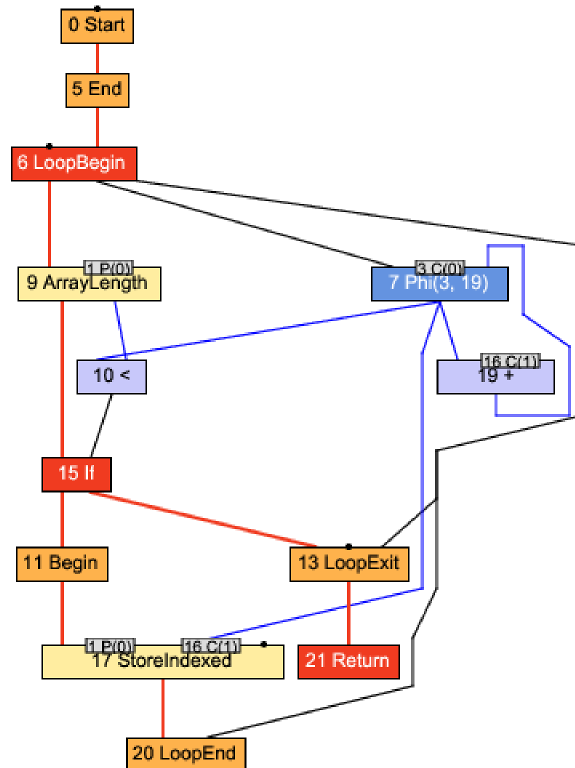


Figure 6.1: Listing 6.2 converted into GRAAL-IR.

the GRAAL-IR the control-flow edges (red lines) connect nodes to ensure they are executed in program order. The nodes connected by the control-flow edges are the ones which need to have their order preserved. Typically, this is because they represent operations that perform memory accesses – like the `StoreIndexed` node. However, there are also nodes that are connected via data-flow edges (blue) called floating nodes that represent a computed value.

Like other compilers, GRAAL starts with a high-level of abstraction and goes through a gradual lowering process that converts the nodes into increasingly lower-level representations. In this example, there is little scope for applying any high-level optimisations, and so the next step is to lower the GRAAL-IR – the result of which are shown in Figure 6.2. The clearest example of the effects of lowering is where nodes representing language-level operations – like indexed array accesses to an array or finding its length – are translated into memory reads or writes. Once lowered, all the remaining nodes in the GRAAL-IR are easily mappable onto low-level machine instructions.

By targeting OpenCL C, Tornado can avoid the need to implement multiple code generators – one for each architecture that it wants to support – but this comes with

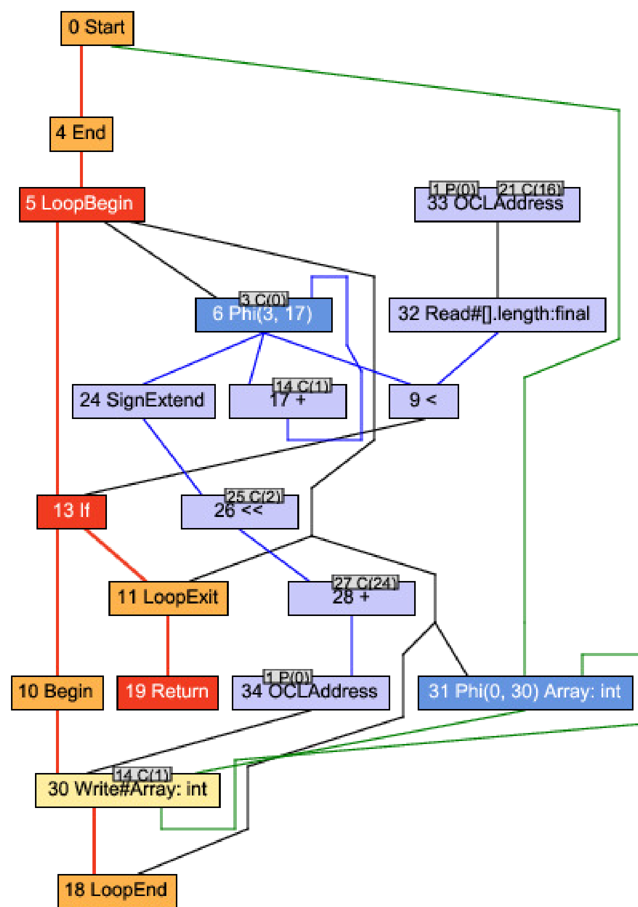


Figure 6.2: Figure 6.1 after being lowered.

Figure 6.3: The control-flow graph calculated by GRAAL is overlaid across Figure 6.2.

```
__kernel void one(__global uchar *_heap_base, ulong _stack_base)
{
  long l_5, l_4, l_6;
  int i_2, i_3, i_8;
  ulong ul_1, ul_0, ul_7;

  __global ulong *slots = (__global ulong *) &_heap_base[_stack_base];


  // BLOCK 0
  ul_0  =  (ulong) slots[6];
  ul_1  =  ul_0 + 16L;
  i_2  =  *((__global int *) ul_1);
  // BLOCK 1
  i_3  =  0;
  for(;i_3 < i_2;)  {
    // BLOCK 2
    l_4  =  (long) i_3;
    l_5  =  l_4 << 2;
    l_6  =  l_5 + 24L;
    ul_7  =  ul_0 + l_6;
    *((__global int *) ul_7)  =  1;
    i_8  =  i_3 + 1;
    i_3  =  i_8;
  }
  // BLOCK 3
  return;
}
```

Listing 6.3: The output of the Tornado JIT compiler for Listing 6.1.

the cost of translating GRAAL-IR into OpenCL C. By choosing to target OpenCL C, an extra level of complexity is imposed on the compiler to overcome the language restrictions. This complexity stems from the fact that OpenCL C does not support `goto` statements and so the compiler has to recover the original control-flow structures that were present in the source code. Fortunately, this is possible using structural-analysis [107]. However, if this becomes problematic it is possible to generate different low-level code like PTX [90], HSAIL [56] or SPIR-V [74].

Once the control-structures are recovered the compiler is then able to construct a control-flow graph (CFG) from the GRAAL-IR. The result is shown in Figure 6.3 where the CFG is overlaid on top of the GRAAL-IR. Finally, the code generator is able to traverse the basic-blocks along their control-flow edges of the CFG. Allowing it to visit each block in program order and emit OpenCL statements in a single pass. In the situation where control-flow diverges the appropriate control structure can be found by inspecting both the control-flow graph and the first and last instructions in each basic block. Typically, the GRAAL-IR contains control-flow specific nodes – such as `LoopBegin`, `LoopEnd` and `LoopExit` – that aid this process. The output of the compilation process is shown in Listing 6.3.

```java
public static Float3 mult(Float3 a, Float3 b) {
  return new Float3(
             a.getX() * b.getX(),
             a.getY() * b.getY(),
             a.getZ() * b.getZ());
}

public static float dot(Float3 a, Float3 b) {
  final Float3 m = mult(a, b);
  return m.getX() + m.getY() + m.getZ();
}
```

Listing 6.4: Implementing a dot-product operation using idiomatic Java.

6.2 Compiling Idiomatic Java

Java provides the developer with language features – such as object-orientation, automatic memory management and dynamic class loading – that aim to boost developer productivity. As a result of this Java code often has many different layers of indirection that need to be resolved during compilation. Typically, this is not a problem when running on the JVM as it has views of all the necessary internal state – from the class hierarchy, to live objects, through to profiling information about each method – it can even dynamically load classes to locate missing bytecode. Unlike the dynamic compiler in the JVM, Tornado has limited access to the internal state or features of the JVM. This limitation leads to some knock-on effects. For example, objects cannot be created on the device-side because there is no interface to migrate externally allocated objects into the managed heap within the JVM. Additionally, features like the dynamic resolution of virtual methods are difficult to implement on the device-side as the class meta-data needed to support this functionality is only available on the host-side. Hence, it is not viable for Tornado to support a full complement of Java's functionality. However, what is possible is to continue optimising unimplementable code and hope that it will eventually be eliminated by a compiler optimisation. An obvious example is supporting virtual method calls. If Tornado discovers a virtual method call, it does not immediately terminate compilation; instead, it continues until it is unable to infer the type of the base object statically. In the majority of cases, Tornado can resolve virtual method calls due to optimisations like partial evaluation, inlining and constant propagation eliminating unnecessary levels of indirection.

Nearly all Java developers write applications using an idiomatic form of Java. Typically, this involves making substantial use of object-orientated features like interfaces and inheritance that rely heavily on the ability of the JVM to resolve virtual method

calls. Due to its inability to resolve methods calls on the device-side, the Tornado dynamic compiler is unable to operate like HotSpot and compile each method in isolation. Instead, it must compile a maximal sub-graph of the original application which increases the cost of compilation.

Another related problem is that idiomatic Java code is likely to make use of object-orientation to hide away implementation details from the code. The consequence is that a lot of code will not operate directly on primitive types – such as integers and floating point values – but on composite types (objects) or structures of composite types. For instance, it is common practice for developers to store data in primitive arrays and then to wrap the array in an object to provide an interface for accessing the data. The problem here is that this adds a level of indirection into data accesses that can become costly on particular architectures, for example, SIMT.

Finally, as a consequence of object-orientation developers often make liberal use of the `new` keyword. There is an underlying issue with this because full support would require the ability to allocate objects inside the JVM – which is not currently possible.

The remainder of this Section will describe how Tornado compiles a simple example of idiomatic Java code to show that it is possible to support such code on hardware accelerators and that with a well-designed compiler infrastructure the use of object-orientation does not necessarily incur a disproportionate cost.

6.2.1 Example

The `dot` method in Listing 6.4 is a case of idiomatic Java code taken from the real-world application – Kinect Fusion – that is described in Section 7.3. Despite being relatively short, it utilises a broad range of Java language features – such as object instances, static methods, virtual methods, and objects allocations. Additionally this example uses the `Float3` short-vector type from the Tornado Collections API (see Section 3.2.2). From the performance perspective, some of these features are ill suited to the types of processor architectures that might be targeted. For instance, accelerators such as GPGPUs do not perform well for code that is dominated by control-flow. Therefore, a fundamental question is to what extent can the compiler eliminate these undesirable features? Hence, the upcoming sections will describe how GRAAL can compile this code and, more importantly, generate high-performance code compatible with a GPGPU or multi-core processor.

6.2.2 Inlining

Inlining is where a method call is replaced with a copy of the body of the target method. It is widely considered to be one of the most profitable optimisations and when it is applied the aggregated GRAAL-IR should have more scope for optimisation. In the context of Tornado, it is the primary vehicle for removing the layers of abstraction introduced by object-orientation. For instance, in Listing 6.4 there are three getter methods – `getX`, `getY` and `getZ` – that are eliminated entirely through the use of inlining.

The GRAAL-IR for the example is shown in Figure 6.5 where the body of the `dot` method is shown in the center. Note how the GRAAL-IR includes three virtual method calls and one static method calls – these calls will all be removed by inlining these methods. In Figure 6.5 the grey boxes indicate both the method bodies that are to be inserted and the callsite they replace. The `mult` method is an interesting example as it also contains virtual method calls to `getX`, `getY` and `getZ`. GRAAL is also able to inline these methods in `mult` before the body of `mult` is inserted into `dot`. The result of inlining is shown in Figure 6.4 where both the method calls and all the control-flow nodes have been eliminated. Each time GRAAL finishes an inlining optimisation it follows it up immediately with a combination of other optimisations – such as partial evaluation and constant propagation. It is these follow on optimisations that have eliminated the control-flow nodes in the GRAAL-IR.

6.2.3 Intrinsics

One of the interesting points about the compilation of Listing 6.4 is how Tornado is able feign support for the `new` keyword in this example. This is achieved because `Float3` is part of the Tornado Collections API and can be mapped directly onto a built-in type in OpenCL C. By doing this the compiler avoids the need to support object creation for small vectors – making them very efficient to use. The benefits of this are seen when code is operating on large amounts of them, as this is where the cost of object creation starts to manifest itself. The advantage of GRAAL-IR is that during its construction it is possible to replace the node that creates a new `Float3` object with a new `float3` value. Using this same technique, it is also possible to map method calls directly onto either OpenCL built-in functions or specific machine instructions. A good example is the ability to replace method calls to `java.lang.Math.sin` with the OpenCL built-in `sin` function.

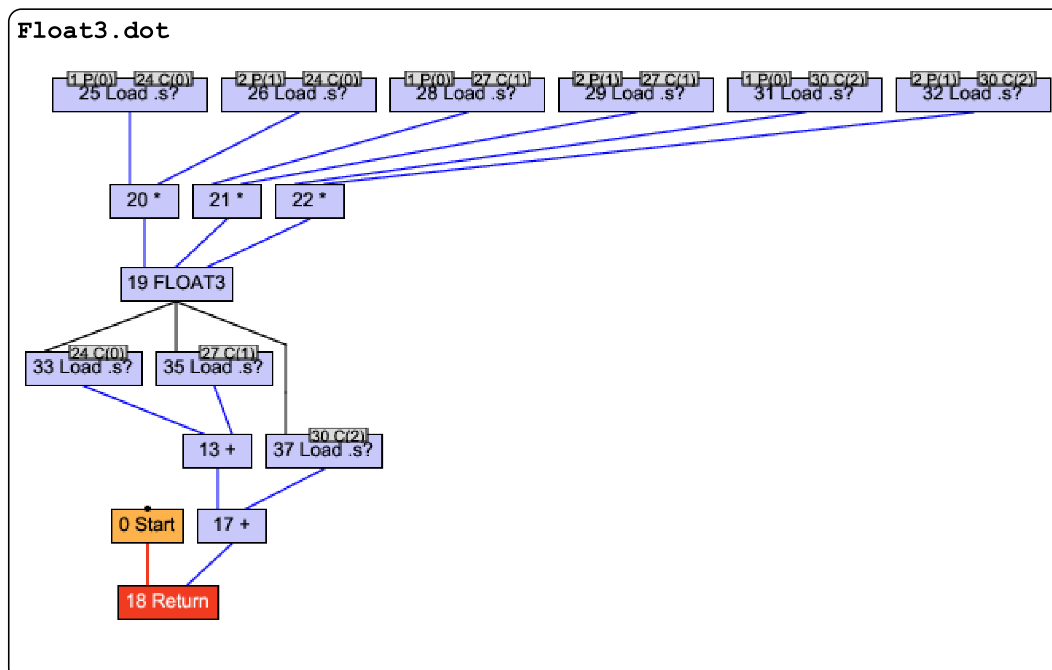


Figure 6.4: State of the GRAAL-IR after inlining the virtual method calls – `getX` textttgetY and `getZ` – from Figure 6.5.

6.2.4 Partial Escape Analysis And Allocation Removal

One of the most common issues with idiomatic Java code is the liberal usage of the `new` keyword. Creating a new object is time consuming as the JVM needs to register the object with the memory manager and execute the objects constructor. This can be problematic as the memory manager needs to be thread-safe – another thread or even the garbage collector might be interacting with the managed heap at the same time. Hence, there is a disproportionate overhead associated with allocating or de-allocating small objects. To avoid this problem the compiler should try and avoid allocating space on the managed heap. One way of doing this is to find all objects whose lifetimes exist exclusively within the current compilation scope and allocate them on the stack. However, to perform this optimisation the compiler needs to ensure that an object never escapes the current compilation scope – this is the role of partial escape analysis.

Consider what would happen if in Listing 6.4 `Float3` objects were not used and instead plain arrays where used (i.e. `float[]`). Listing 6.5 shows this alternate and less idiomatic implementation of the dot-product. Note that in this implementation a new array is created in the `mult3` method. If `mult3` is compiled in isolation then the new array cannot be eliminated as it escapes as a return parameter. However, if it is

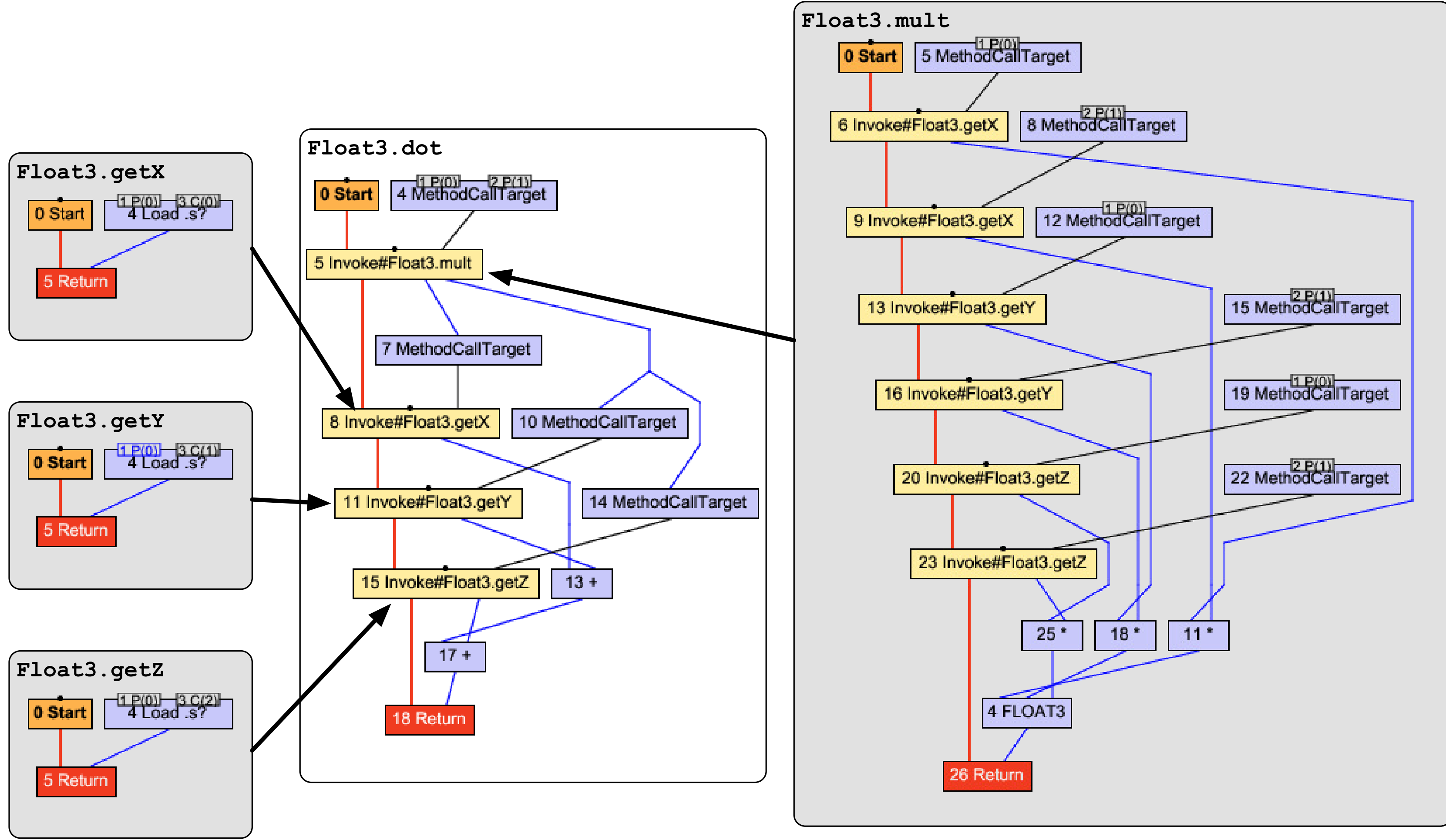


Figure 6.5: GRAAL will inspect the GRAAL-IR to find virtual method calls. If it is able to find one that it can resolve to a concrete method and the method is small enough then GRAAL will replace the node representing the method call with the GRAAL-IR of the method being called. This is shown above where `getX`, `getY`, `getZ`, and `mult` are being inlined into the GRAAL-IR for `Float3.dot`.

```java
public static float[] mult3(int n, float[] a, float[] b) {
  final float[] c = new float[n];
  for (int i = 0; i < n; i++) {
    c[i] = a[i] * b[i];
  }
  return c;
}

public static float dot3(int n, float[] a, float[] b) {
  float[] c = mult3(n, a, b);
  float sum = 0;
  for (int i = 0; i < n; i++) {
    sum += c[i];
  }
  return sum;
}
```

Listing 6.5: An alterative implementation of the dot product that does not use object-orientation.

`dot3` that is being compiled then `mult3` is a candidate for inlining. Once inlined then the array created within `mult3` will exist solely within the compilation scope of `dot3` – i.e. it no longer escapes. At this point GRAAL is able to allocate the variable on the stack and the GRAAL-IR is shown in Figure 6.6.

What is interesting about this example is that the entire array is written to in `mult3` and then read back in `dot3`. Here the compiler is able to recognise these read-after-write data dependencies and eliminate the intermediate writes and reads to the array. The result is shown in Figure 6.7 – note the similarity to Figure 6.4. In fact, if it is possible to coalesce the reads and writes into vector reads and writes then both examples will generate the same final code.

6.2.5 Partial Evaluation

One of the advantages of deferring the compilation process until runtime is that there is more information available to the compiler. For instance, it is possible for the compiler to know the exact specification of the hardware it needs to target. However, as Tornado is built upon Java – a language that provides reflection – the compiler has the ability to treat application data as information that is known at compile time. This means that the compiler is able to determine the exact values and types of task parameters and static variables. To exploit this information fully, a technique called partial evaluation is used [48, 71, 106, 117].

Partial evaluation is where the compiler evaluates IR using known information and

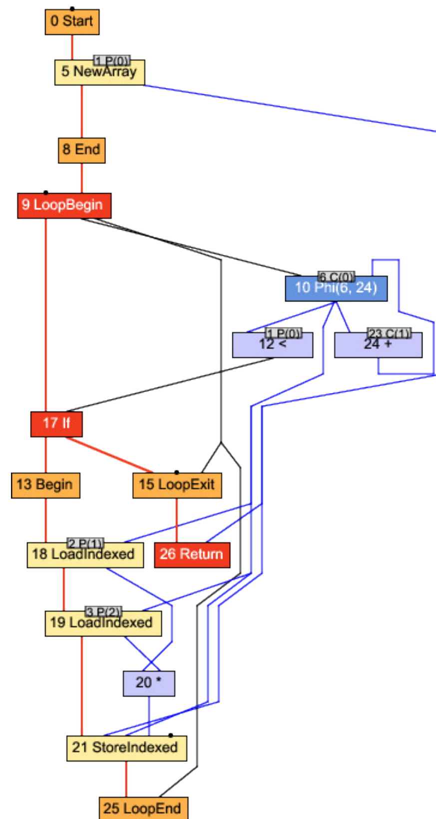


Figure 6.6: GRAAL-IR of Listing 6.5.

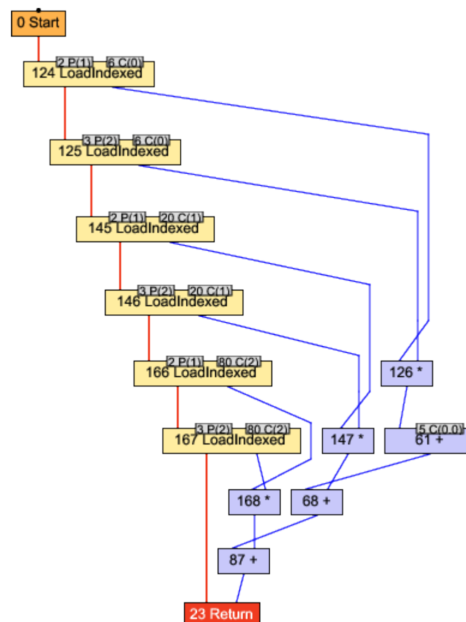


Figure 6.7: GRAAL-IR after performing inlining, partial escape analysis and partial evaluation on Figure 6.6. Notice how the `NewArray` node has been eliminated via partial escape analysis.

```
__kernel void dot(__global uchar *_heap_base, ulong _frame_base)
{
  float3 v3f_3, v3f_7, v3f_1;
  ulong ul_2, ul_0;
  float f_5, f_4, f_6, f_9, f_8;

  __global ulong *frame = (__global ulong *) &_heap_base[_frame_base];


  // BLOCK 0
  ul_0  =  (ulong) frame[6];
  v3f_1  =  vload3(0, (__global float *) ul_0);
  ul_2  =  (ulong) frame[7];
  v3f_3  =  vload3(0, (__global float *) ul_2);
  v3f_7  =  v3f_1 * v3f_3;
  f_8  =  v3f_7.s0 + v3f_7.s1;
  f_9  =  f_8 + v3f_7.s2;
  frame[0]  = (ulong) f_9;
}
```

Listing 6.6: The OpenCL C code generated by the Tornado compiler for Listing 6.5.

replaces the expression with the calculated value – usually a constant value. As Tornado knows the exact values of the parameters used for `dot3` then these can be propagated into the GRAAL-IR. This means that the size of the array in Figure 6.6 becomes constant. Once this happens partial evaluation is used to eliminate all the nodes that depend on this value. Hence, all the control-flow is removed from the example to produce Figure 6.7. The final result of compiling `dot3` is shown in Listing 6.6.

6.3 Bytecode Coverage

One of the challenges laid out in Section 1.1.5 is understanding what aspects of modern programming languages can be used on hardware accelerators. Therefore, the role of this Section is to describe what language features are available in Tornado. However, what might not be immediately obvious is that Tornado does not support the Java languages directly. Instead, it operates at the bytecode-level and so it more important to know what bytecodes are supported.

Table 6.1 summarises the range of bytecodes that Tornado supports. Here a category of bytecodes is either: fully-supported by Tornado, partially-supported by Tornado, supported by Jacc, or unsupported. The majority of bytecodes are fully supported by Tornado. However, it is the ones that are partially supported that are of interest. Typically, these are the bytecodes – such as `invokevirtual`, `invokeinterface`, and `new` – that are used to provide high-level language features in Java. As such, these bytecodes often require access to class and object meta-data that resides inside the JVM

to operate correctly. For example, `invokeinterface` needs the ability to resolve the concrete type of an object to locate the method that is to be invoked. Fully supporting these bytecodes on a device like a GPGPU is likely to become very complex and lead to slow code. However, what Tornado aims for is to support these bytecodes under certain circumstances. Typically, when the GRAAL compiler is able to remove the need to lookup class or object meta-data from the generated code. Normally, this happens when the compiler is able to eliminate the incompatible bytecode/ An example of this happening was shown in Section 6.2. If the compiler is unable to optimise away the unsupported bytecodes it is possible for Tornado to generate an error warning the user of the problem.

Some of the hardest bytecodes to support are the ones that create new objects or arrays. Tornado has no problem calling a constructor or allocating space for newly created objects on the device-side. The problem that Tornado encounters is that it is hard to push an object that is created outside the JVM into the JVM. The primary reason for this is that when an object or array is created inside the JVM it is registered with the memory manager. This is important as this is a requirement for garbage collection. However, as there is not a simple way for an object to be registered with the JVM then full support for the `new` bytecode will not be possible.

The reason for including Jacc (see Section 6.4) in the table of bytecodes is that it was possible to support more Java bytecodes in Java than in Tornado. This opens an interesting discussion surrounding the utility of OpenCL as an implementation target for languages. The problem with OpenCL is that targeting OpenCL C restricts the implementation of support for certain bytecodes. For instance, exception handling is difficult to implement as this requires the ability to branch to the exception handler. However, as the `goto` keyword is not allowed in OpenCL C there is no way to generate these branches in Tornado. In the case of Jacc, supporting exceptions was possible because it targeted PTX [90] – a virtual instruction set architecture for NVIDIA GPGPUs - that supports branching. An outcome of this is that future versions of Tornado should aim to use a lower-level compilation targets like SPIR-V [72] and HSAIL [56].

Finally, it should be made explicit that Tornado is not able to support any method calls to native libraries or support an operations that require operating system support. Unfortunately, this means that device-side code cannot access files or transfer data across the network.

Category	Example Bytecodes	FS	PS	Jacc	NS	Comments
Stack Manipulation	iload_0, fconst_0, dup, pop	✓		✓		Fully supported
Arithmetic	iadd, ldiv, fmul, fneg	✓		✓		Fully supported
Bitwise	lshl, lshr, ushr, iand	✓		✓		Fully supported
Casting	l2f, l2i, i2s, i2l	✓		✓		Fully supported
Control-flow 1	return, ret, ifne, goto	✓		✓		Fully supported
Control-flow 2	jsr, tableswitch					Not used but could be implemented
Arrays	arraylength, aaload, iaload, fastore	✓		✓		Fully supported
Object Creation	new, newarray, anewarray		✓			Only supported if the object does not escape from the compilation scope
Field Access	getfield, putfield, getstatic, putstatic	✓		✓		Fully supported
Method Calls 1	invokestatic, invokevirtual, invokespecial		✓			All supported if compiler can resolve method at JIT compile time
Method Calls 2	invokeinterface, invokedynamic		✓			All supported if compiler can resolve method at JIT compile time
Reflection	instanceof	✓		✓		Fully supported
Exceptions	athrow			✓		Unable to implement in OpenCL C
Other	monitorenter, monitorexit, breakpoint				✓	Not used or required.

Table 6.1: Summary of bytecode support in the Tornado JIT Compiler. Key: FS – Full Support in Tornado; PS – Partial Support in Tornado; Jacc – supported in Jacc; NP – either not possible to support or not needed.

6.4 Aside: Jacc

At this point an aside will be taken to help explain both the lineage of Tornado and provide some clarity as to the role OpenCL plays in the Tornado implementation. Moreover, it explains why there is confidence that the reductions required by the real-world application in Section 7.3 are only a temporary limitation to Tornado.

One of the most obvious questions about the current implementation of Tornado is: how much does it rely on OpenCL? In projects such as APARAPI [1], JOCL [127], JCUDAMP [37] OpenCL has been expose to the developer to create a heterogeneous programming environment to good effect. However, as discussed earlier in Section 2.4.7 this use of OpenCL means that these languages typically inherit the features of OpenCL to become a low-level heterogeneous programming language. Hence, to avoid implementing a low-level heterogeneous programming language in a modern dynamic programming language a different approach is needed.

Before Tornado was created, all the key technologies – the runtime system and JIT compiler – were prototyped in a system called Jacc [28]. Jacc was designed to solely target CUDA based GPGPUs and, as such, one of its novel features was that the compiler targeted PTX [90] – a virtual Instruction Set Architecture for NVIDIA GPGPUs. By targeting PTX, Jacc was able to implement support for: exception handling – a language feature that cannot be implemented in Tornado due to branching restrictions in OpenCL C; shared memory atomic operations – that cannot be implemented as they are not included in the OpenCL C specification; and hardware specific instructions – like `popc` that if used have a tangible impact on performance (see Section 6.4.5). In the remainder of this Section a brief overview of Jacc will be provided with some key features highlighted (for a fuller description please refer to [28]). The aim here is to demonstrate that the use of OpenCL is not a limitation of Tornado and that other implementations have been explored.

6.4.1 Jacc Architecture

Figure 6.8 provides a high-level overview of Jacc. Notice that it looks like a tightly coupled version of the Tornado Runtime System shown in Figure 5.1. The reason for this is that as Jacc solely targets CUDA based GPGPUs there no virtualisation layer is required. The other two points to note are that the Jacc compiler converts Java bytecode into PTX assembly and that the compiler is based on the SOOT [122]. The later became an bottleneck due when attempting to write the Kinect Fusion application

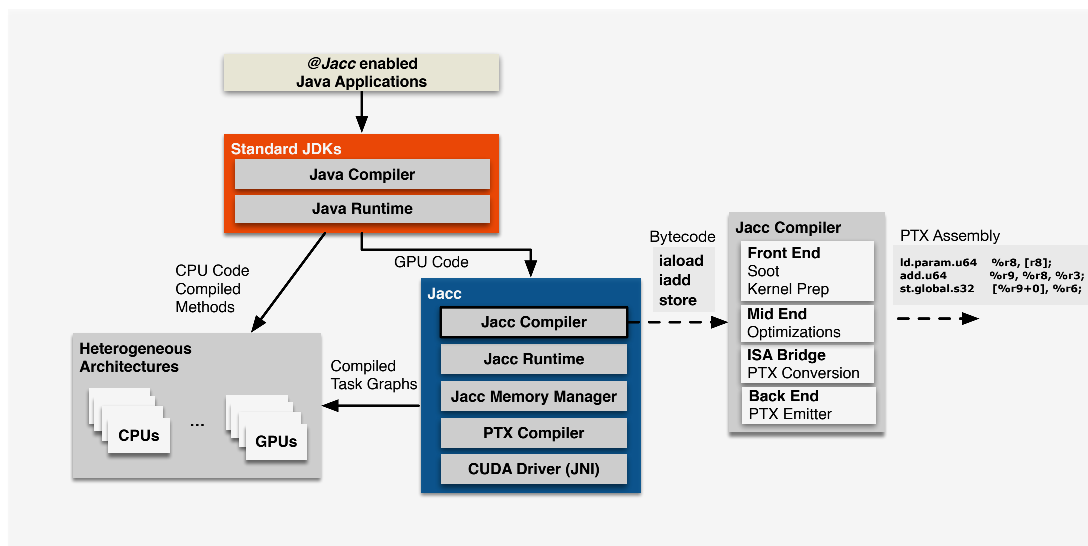


Figure 6.8: Jacc System Overview.

(see Section 7.3) in Jacc. This was primarily due to SOOT not being designed to function as a JIT compiler and when presented with complex application kernels – like the ones in Kinect Fusion – it struggled to achieve sub-second (or sometimes sub-minute) compilation times. Hence, a move to GRAAL was made and Tornado was born.

6.4.2 Shared Memory Atomics and Reductions

One of the strengths of Jacc is that it is possible to write high performance reductions. This is primarily achieved by exposing shared-memory atomic operations to the developer. Listing 6.7 shows the Jacc implementation of the reduction benchmark used later. Take note of the `@Atomic(op=ADD)` annotation in the second line. This instructs the compiler that all assignments to `result` should be combined together using the addition operation. The implication is that the operation `result = sum` is converted into a thread-safe `result += sum` operation by the compiler. By using this technique Jacc provides a way for the developer to implement the reduction operations using a single kernel. At the moment Tornado cannot use this approach with OpenCL 1.2 as the range of shared-memory atomic operations is limited – there are heavy restrictions on the use of floating-point types.

```
public class Reduction {
  @Atomic(op=ADD) float result;

  @Jacc(iterationSpace=ONE_DIMENSION)
  public void reduction(
      @Read float[] array) {
    float sum=0;
    for(int i=0;i<array.length;i++) {
      sum+=array[i];
    }
    result=sum;
  }
}
```

Listing 6.7: The implementation of a fully parallel reduction kernel in Jacc using shared-memory atomics. Presently, this is not currently possible in Tornado due to the lack of support for shared-memory atomics in OpenCL. This is a primary reason why an external OpenCL C kernel is needed in Section 7.6.

6.4.3 Benchmarking Jacc

To provide an idea of how well Jacc performs it has been compared against: serial Java, multi-threaded Java, OpenMP, CUDA and the more mature APARAPI [1] implementations.

The performance of each benchmark is calculated by measuring the time to perform the specified number of iterations of the performance critical section of the benchmark. Each quoted performance number is an average across a minimum of ten different experiments. The reported Jacc execution times are inclusive of a single data transfer to the device and a single transfer to the host but exclusive of JIT compilation times. This is done in order to demonstrate both the peak-performance of Jacc generated code and the low-overheads of the runtime system.

In terms of programmability, the stance is taken that code complexity is proportional to code size and that code can be accelerated, using a GPGPU, without requiring any significant increase in code complexity over a multi-threaded implementation. We assess this by measuring the number of source code lines required to express the data-parallel kernel(s).

The experimental hardware platform used was Server 1 (as described in Table 7.1 in Section 7.2.1). In terms of software: CUDA 6.5 and the Java SDK 1.7.0_25 were used on top of the CentOS 6.5 operating system. Note that all CUDA implementations that are evaluated are taken from the CUDA SDK except the matrix multiplications: SGEMM is taken from the `cuBLAS` library and SPMV from `cuSPARSE`.

6.4.3.1 Benchmarks

The benchmarks used for this performance evaluation are:

Vector Addition: adds two 16,777,216 element vectors (300 iterations).

Reduction: performs a summation over an array of 33,554,432 elements (500 iterations).

Histogram: produces frequency counts for 16,777,216 values placing the results into 256 distinct bins (400 iterations).

Dense Matrix Multiplication: of two 1024×1024 matrices (400 iterations). Note: the OpenMP implementation uses the OS supplied `libatlas` library.

Sparse Matrix Vector Multiplication: performs a sparse matrix-vector multiplication using a 44609×44609 matrix with 1029655 non-zeros (The bcsstk32 matrix from Matrix Market) (400 iterations).

2D Convolution: of a 2048×2048 image with a 5×5 filter (300 iterations).

Black Scholes: is an implementation of the Black Scholes option pricing model. The benchmark is executed to calculate 16,777,216 options over 300 iterations and is supplied as an example in the APARAPI source code.

Correlation Matrix is an implementation of the Lucene OpenBitSet "intersection count". The benchmark is executed using 1024 Terms and 16384 Documents and is supplied as an example in the APARAPI source code. Only a single iteration is performed.

6.4.4 Comparison with OpenMP and CUDA

Figure 6.9 shows the results of the benchmarking. As comparisons between GPGPU accelerated code normalised to that of a serial Java implementation lend themselves to producing speed-ups in the order of one to two orders of magnitude some sanity checks are introduced. Chiefly, the inclusion of results for the non-Java based OpenMP and CUDA implementations.

By comparing against the multi-threaded Java and OpenMP implementations, the observation is made that, with the exception of the sparse matrix vector multiplication benchmark, the GPGPU accelerated Jacc implementations tend to outperforms

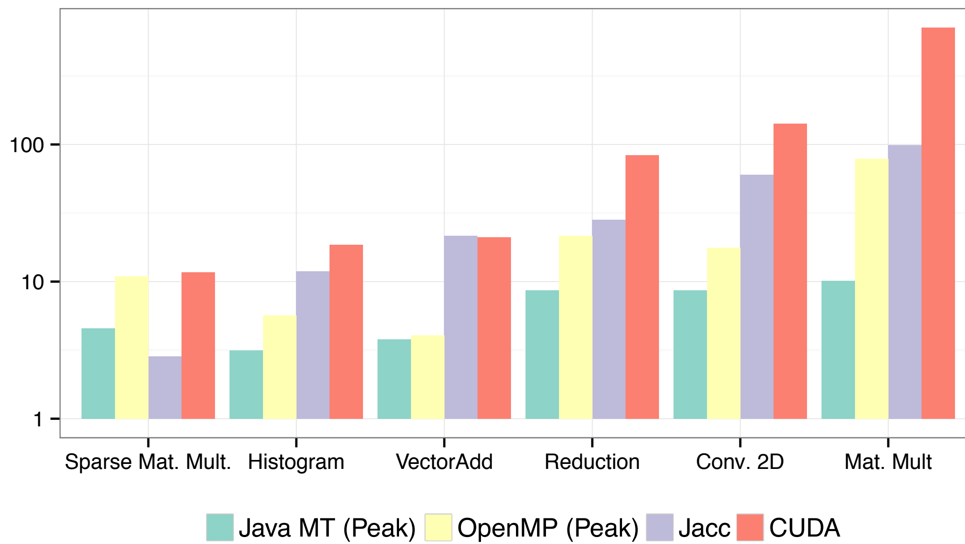


Figure 6.9: The performance of the Jacc (GPGPU) accelerated implementations of the benchmarks normalised to the performance of the serial Java implementation. Note: As multi-threaded implementations use between 1 and 24 threads (up to one per core) only the best performance has been recorded and is marked as peak.

the multi-threaded CPU only implementations. Furthermore, in order to provide a strong comparison point, the OpenMP version of SGEMM is provided by `libatlas`. Results indicate that even in this case Jacc is still able to outperform OpenMP, albeit by a reduced margin in comparison to Java multi-threaded implementations.

Table 6.2 summarises the speed-ups obtained by Jacc against the serial Java implementations. The speed-ups have been normalised using two different Java implementations: a serial Java implementation and the best performing multi-threaded Java implementation. These results indicate that Jacc, on average, outperforms the serial and best performing multi-threaded implementation of all Java implementations by $19\times$ and $5\times$ respectively. The pathological case is the sparse vector multiplication benchmark, where the irregular memory accesses pattern is not well suited to the parallelisation strategies that are employed by Jacc. Typically, this can be resolved either algorithmically or through better code generation – assigning loop iterations on a per warp basis and making use of the texture cache. Note that this is less of a problem in Tornado due to the use of dynamic configuration – described in Section 3.3.2 and evaluated in Section 7.8 – provides the developer with more control over how each kernel is parallelised and executed.

The effect on programmability is studied by comparing the lines of code required

	speed-up			Lines of Code		
Benchmark	Serial	Java MT Best		Java MT	Jacc	Reduction
Vector Add	21.52	6.00	(20)	40	6	6.67x
Matrix Mult.	98.56	13.08	(24)	46	16	2.88x
2D Conv.	60.31	10.18	(24)	66	33	2.00x
Reduction	28.31	4.21	(16)	43	11	3.91x
Histogram	11.86	7.53	(24)	61	8	7.62x
Sparse Mult.	2.85	0.63	(20)	51	14	3.64x
Black Scholes	5.93	-		-	-	-
Cor. Matrix	26.16	-		-	-	-
Geo. Mean	19.27	5.02		50	13	4.01x

Table 6.2: A comparison of Jacc against Java based implementations. Note: As multi-threaded implementations can use between 1 and 24 threads only the best performance has been recorded and the number of thread used recorded in brackets.

to implement data-parallel code in Jacc against that required to write multi-threaded Java code. The results show that using Jacc to create data-parallel code requires 4x fewer lines of codes than writing them using Java threads.

6.4.5 Comparison with APARAPI

Additionally, Jacc has been compared against APARAPI [1], an alternative Java based framework, using three of their benchmarks: Vector addition, Black Scholes, and Correlation Matrix. It should be noted that it was very hard to find benchmark code that was written in APARAPI and that no standard Java-based GPGPU benchmarks exist. A comparison of the results is shown in Figure 6.10. To understand the impact of JIT compilation on performance, experiments were conducted that were both inclusive and exclusive of compilation times. Comparing the geometric mean of these speed-ups, it is observed that both frameworks are very similar in terms of performance; APARAPI just incurs less overheads due to JIT compilation.

In contrast to Jacc, APARAPI is built upon OpenCL and uses source-to-source translation to generate OpenCL C from Java bytecode. This approach provides APARAPI with two advantages: consistently low-compilation times, around 400 milliseconds, and a high quality of generated code. As the compiler matures, the cost of JIT compilation will fall, so that it is comparable with APARAPI. Note that this last remark is now evident when comparing the compilation times of Tornado in Figure 7.14 in

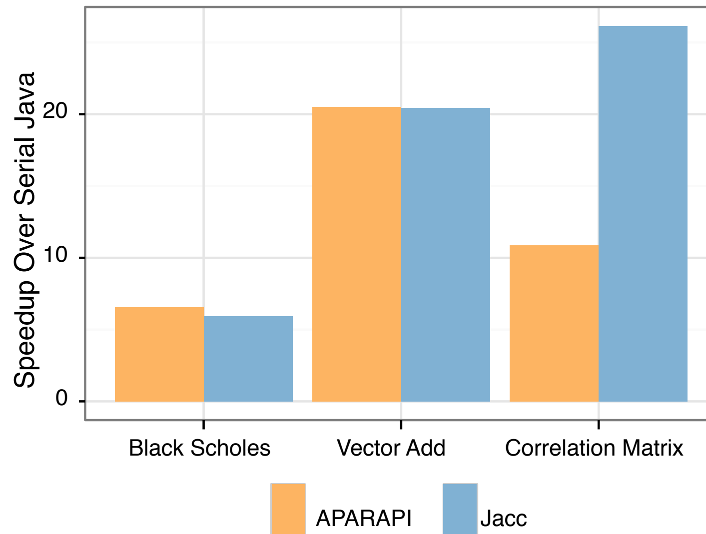


Figure 6.10: speed-up obtained by APARAPI and Jacc over serial Java implementations

Section 7.7.2 and justifies to move from SOOT to GRAAL.

In the Correlation Matrix benchmark, Jacc significantly outperforms APARAPI because of its ability to: (1) easily tune the number of threads in each work group, and (2) to utilise the `popc` instruction. On this benchmark it was identified that changing Jacc's work group size to match that of APARAPI, severely reduced performance but remained faster than APARAPI. The outcome of this was the inclusion of dynamic configuration in Tornado (discussed in Section 3.3.2). Note that the same technique is again used to improve the performance of Tornado in Section 7.8.

6.4.6 Why Is Tornado Based on OpenCL?

A key question remains as to why does Tornado not support CUDA/PTX? The difference between Jacc and Tornado is that Tornado *generalises* the technologies developed for Jacc so that they are less GPGPU centric. This can be seen by the fact when evaluating Tornado there is a selection of non-CUDA based GPGPUs, multi-core processors, and a discrete many-core accelerator (see Table 7.1). Hence, to try and demonstrate that building a framework like Tornado is possible the decision was made to target OpenCL C as it was the only available option that would realistically allow Tornado to be evaluated on both GPGPU and non-GPGPU accelerators. The benefit of this decision is that Tornado is able to target a wide range of devices from a single TVM client and more importantly that there is high confidence that adding PTX, HSAIL [56] or

even SPIR-V [72] support will only increase the capabilities of Tornado.

6.5 Summary

This Chapter has described some of the challenges faced implemented the Tornado Virtual Machine Client (TVMC). The role of the TVMC is to implement the client-side interface of the Tornado Virtual Machine. In the current implementation the TVMC is implemented to use OpenCL. As such the key challenge has been to develop a JIT compiler that is able to compiler Java bytecode into OpenCL C. To do this the industrial quality GRAAL[40] compiler is used. Section Section 6.1 describes the process of turning Java bytecode into OpenCL C.

One of the key challenges of heterogeneous programming languages, as described in Section 1.1.5, is understanding whether the language features that exist in modern programming languages are amenable to generating high-performance device-side code. Therefore, in Section 6.2 describes how some object-orientated Java code is handled in the Tornado JIT compiler. The result is also contrasted against a simpler implementation that uses no object-orientation in Section 6.2.4. The outcome is that the GRAAL compiler is able to eliminate nearly all of the cost of using object-orientation through the use of inlining, partial evaluation and partial escape analysis. Meaning that the performance of these two codes are almost identical.

Next, a list of language features that are supported by Tornado is provided in Section 6.3. This list describes what has been implemented and tested in Tornado and what could be implemented in the future. One of the outcomes is that supporting some of the more complex language features – like virtual method calls and the `invokedynamic` bytecode – is highly dependent on their compilation scope. Hence, these features can be supported but only if the compiler is able to optimise them away.

Finally, an aside is taken to discuss Jacc in Section 6.4 – the precursor to Tornado. This is done to highlight a number of key points: (1) that Tornado is not overly reliant on OpenCL, (2) to demonstrate that the real-world application used in Chapter 7 is complex enough to require an industrial quality compiler to execute, and (3) to highlight that using OpenCL actually limits some of the Java language features that Tornado is able to implement. Hence, Section 6.4.2 demonstrates how the OpenCL kernel that needed to be hand written in Section 7.6.1 could be written using Tornado in the future.

7 | Evaluation

Presently heterogeneous programming languages – like OpenACC, OpenCL, and CUDA – all require assumptions to be made about the number or type of components contained within a heterogeneous system. For instance, CUDA expects all systems to contain NVIDIA based GPGPUs. Whereas OpenACC expects that all potential hardware accelerators are disclosed to it at compile time. One of the consequences of treating the system configuration as a closed-world assumption like this is that any resulting application will not be robust to changes in the system configuration. Meaning that a small change, like replacing a broken GPGPU, could require an application to be re-compiled from source. Thus, the aim of this thesis is to demonstrate that there is no need for a heterogeneous programming language to make any assumptions about the number or type of hardware components that exist within a heterogeneous system. In the previous Chapters, a heterogeneous programming language called Tornado has been described that does just this. The salient feature of Tornado is that it enables an application to be written once and executed across a wide range of hardware accelerators. To support this thesis this Chapter evaluates Tornado's ability to create a real-world application, called Kinect Fusion, and execute it robustly across thirteen distinct hardware accelerators – a mix of integrated and discrete GPGPUs, multi-core processors and a discrete many-core accelerator. More specifically, the supporting evidence for this thesis is summarised as follows:

Device Coverage One of the most fundamental aspects of Tornado is that it should enable applications to be executed across a wide range of hardware accelerators. In Section 7.5 it is shown that Tornado is able to readily execute a real-world application across thirteen different devices. In contrast OpenCL – representing the state-of-the-art of heterogeneous programming languages – manages to execute on nine different devices (or 70% of the available devices). A discussion on the reasons behind this lack of portability is provided in Section 7.5. What is important to note is that Tornado achieves this portability without the developer

having to explicitly compile the application on a per-system basis – i.e. the application is freely distributable across all systems in a system-neutral bytecode format.

Application Performance Two scenarios are used to evaluate whether Tornado is able to increase application performance by utilising hardware accelerators. Firstly, the most portable Tornado implementation is evaluated in Section 7.5 where a maximum speed-up of 55× the Java reference implementation is observed. Secondly, a specialised implementation of Kinect Fusion is considered in order to determine the possible performance gains should a developer wish to sacrifice some portability. In this scenario a speed-ups of between 18× and 150× over the Java reference implementation are observed (see Figure 7.11).

To establish confidence in these results the performance capability of Tornado is compared against a non-Java reference implementation of Kinect Fusion. Here a comparison is made against a non-Java implementation written in OpenCL. This OpenCL implementation aims to allow a comparison to be made against the state-of-the-art heterogeneous programming language. In these experiments Tornado is able to achieve on average 59%of the performance of OpenCL with the most portable implementation of Kinect Fusion. However, this rises up to 77% when a specialised Tornado implementation is used (see Figure 7.15).

Code Quality In order to understand the performance gap between the Tornado and OpenCL implementations Kinect Fusion Section 7.7.1 compares the device-side performance of ten Kinect Fusion kernels. Here the observation is made that on average Tornado achieves on average 98% of the performance of OpenCL across these ten kernels (see Table 7.9).

Correctness To ensure that performance is only evaluated across implementations that produce a meaningful result a strict correctness criteria is applied: that implementations of Kinect Fusion must produce tracking errors that are either equal or less than the C++ implementation. Both the Tornado and non-Tornado Java implementations achieve absolute trajectory errors of 0.0119m (see Table 7.5). Note that this is considerably lower than all other implementations: C++ and OpenMP (0.0206m); and OpenCL (0.0207m).

Dynamic Configuration One of the key drivers for eliminating any closed-world assumptions made about the type or number of hardware accelerators available to

the application is that it allows the application to be configured (or optimised) in-situ on a device. For example, this may include configuring which hardware accelerator to use or tuning the distribution of work across parallel threads on a specific device. Section 7.8 outlines how dynamic configuration can be used to both improve portability and performance. For instance, using dynamic configuration it was possible to improve the performance of a GPGPU specialised Tornado implementation from 146× to 167× greater than the Java reference implementation (see Figure 7.16). More importantly, dynamic configuration makes it possible for the end-user to experiment with non-intuitive optimisations. In one example a performance increase of 91× was experienced by forcing the NVIDIA GTX550Ti GPGPU to use blocking calls to the OpenCL runtime system: an option that usually leads to lower performance (see Figure 7.17).

7.1 Limitations and Non-goals

One of the core aspects of this thesis is to demonstrate that it is possible to write a heterogeneous application once and execute it across a wide range of devices. To demonstrate this six different systems are used that contain thirteen unique hardware accelerators (see Section 7.2.1). It should be noted that this choice of hardware is intended to capture the widest range of devices that could be to assembled in a reasonable time frame and given unlimited time and resources this list could be further expanded.

Perhaps one of the most important, but possibly non-obvious, aspects of evaluating Tornado is the complexity. Normally, performance comparisons are conducted using small numbers of devices (one or two) which allows an investigator to drill down into low-lying performance issues: like inspecting assembly code, re-implementing the application using a different algorithm, or even re-working the existing program into a more amenable form for a given accelerator. However the problem with Tornado is that it makes it possible to quickly generate and optimise code for more devices than can be analysed by a single researcher in a reasonable time frame. For instance, consider the complexity of performing low-level analysis of ten kernels across thirteen different accelerators – multi-core processors, GPGPUs, a many-core accelerator and FPGAs – that can each use: three different parallelisation strategies – none, blocked and thread cyclic; a user configurable number of threads; a user configurable work group size; and the option to execute kernels asynchronously or synchronously. It should be clear that this complexity stems from the combinatorial explosion in tuning parameters available

when running each application. For this reason the evaluation of Tornado is very ambitious as it involves measuring the performance of a complex real-world application across six systems that contain thirteen different hardware accelerators. Below is a list of actions that have been taken to make this evaluation tractable within the given time constraints:

Compiler Backend To expedite the development of Tornado the compiler has been developed using Graal [39] to target OpenCL C (see Section 7.2.2). Consequently, two JIT compilers execute in a back-to-back fashion – one to compile Java bytecode into OpenCL C and the other to compile OpenCL C into native machine code. As a result, the JIT compilation times are longer than necessary, and the experimental data shows that compilation times could be halved if the OpenCL JIT compiler is removed (see Figure 7.14 in Section 7.7.2).

Moreover, to emphasise that Tornado does not overly rely on the OpenCL JIT compiler, it should be noted that an alternative compiler has been previously developed in [28] and is briefly discussed in Section 6.4. As this compiler targets NVIDIA GPGPUs specifically it is of limited use for evaluating the portability of application codes across non-GPGPU hardware accelerators and hence is not used.

Non-x86 based systems A limitation of this thesis is the inability to demonstrate that Tornado is also agnostic of both the operating system and the architecture of the host-processor. The main reasons for this is that at the time of writing it has not been possible to secure an ARM-based system for evaluation. Despite the availability of ARM-based systems the difficulty lies in obtaining a system that satisfies both of the prerequisite software dependencies: namely a GRAAL enabled JVM[1] and OpenCL drivers. It is very likely that such a system will become available in the near-future and when it does the expectation is that Tornado would behave in a similar fashion to the AMD APU based system.

Complete bytecode coverage Two of the most common misconceptions about executing Java applications on hardware accelerators are that either the entire JVM needs to be ported onto the hardware accelerator or that all Java bytecodes need to be supported for the language to be useful; Tornado does neither.

[1]Technically this requirement can be softened to requiring a Java Virtual Machine that implements Java's new Compiler Interface (JVMCI) rather than GRAAL (`http://openjdk.java.net/jeps/243`)

The aim is of Tornado is not to support general-purpose programming on hardware accelerators but to support the acceleration of computationally demanding applications. By definition hardware accelerators are limited-purpose devices and are specialised for a particular task. Therefore, the role of Tornado is not to allow any code to execute anywhere. Instead, it is designed to allow tasks that are amenable for hardware acceleration to be paired with the best-suited accelerator – as doing otherwise will degrade performance.

Programming Productivity Writing applications for heterogeneous systems is a complex task often requiring low-level knowledge of the target hardware. Tornado has been developed to expedite the creation of applications that can utilise hardware accelerators. However, this thesis refrains from evaluating Tornado's impact on programming productivity. Chiefly, this is because it is hard to quantify productivity: is it measured in lines of code, the time to write the application or the number of concepts a developer needs to learn? Moreover, productivity features are often a personal preference and what is perverse to one developer may not be in the eyes on another. Although, what can be shown is that developers are able to use the same abstractions (or productivity) features to write hardware accelerated code using Tornado.

Multiple devices at present this Chapter only evaluates Tornado in single accelerator mode. The reason for this is that the Kinect Fusion application transfers a large amount of data between pipeline stages and locating them on a physically seperate accelerator will ruin the chances of meeting the Quality-of-Service threshold of 30 frames per second. However, there is no reason why Tornado cannot use multiple accelerators and this is discussed further in Section 8.4.3.

7.2 Experimental Setup

7.2.1 System Configurations

Tornado is evaluated across five different heterogeneous systems. Each system has at least a single multi-core x86 processor and a GPGPU. However, across these systems there are thirteen unique accelerators that target different use-cases from low-power laptops to high-power servers. Specific details of each system and the types of accelerators used are provided in Table 7.1. As all device-side code is executed using

System	OS	Accelerator	Type
Laptop	OSX 10.11.6	Intel i7-4850HQ Intel Iris Pro 5200 NVIDIA GT 750M	mutli-core processor integrated GPGPU external GPGPU
Desktop 1	Fedora 21	AMD A10-7850K AMD Radeon R7	multi-core processor integrated GPGPU
Desktop 2	Fedora 25	Intel i7-2600K NVIDIA GTX 550 Ti	multi-core processor external GPGPU
Server 1	CentOS 6.8	Intel Xeon E5-2620 Intel Xeon Phi 5110P NVIDIA Tesla K20m	multi-core processors external many-core device external GPGPU
Server 2	CentOS 7	Intel Xeon E3-1285 Intel Iris Pro P6300 AMD Radeon HD 6970	multi-core processor integrated GPGPU external GPGPU

Table 7.1: System Configurations

OpenCL, Table 7.2 summarises the key OpenCL configuration for each accelerator.

7.2.2 Tornado Software stack

Java Virtual Machine All Java-based experiments, on both Linux x86_64 and OSX, are performed using OpenJDK version 1.8.0_131. The JDK has been built from source to allow early access to the Java Virtual Machine Compiler Interface (JVMCI) that is due for release with Java 9. (For completeness build 25.71-b01-internal-jvmci-0.27-dev is used.) The JDK has been modified only allows Tornado to use runtime type annotations. Experiments are performed using the server compiler and an initial heap-size of 8GB using the `-server` and `-Xms8G` options respectively.

Graal Tornado uses a modified version of Graal version 0.22. It has only been modified to allow a new Graal backend to be created outside of the Graal project.

7.3 Real-world Application: Kinect Fusion

Kinect Fusion is a computer vision application that constructs a three-dimensional model of a scene from a single hand-held RGB-D camera such as the Microsoft Kinect.

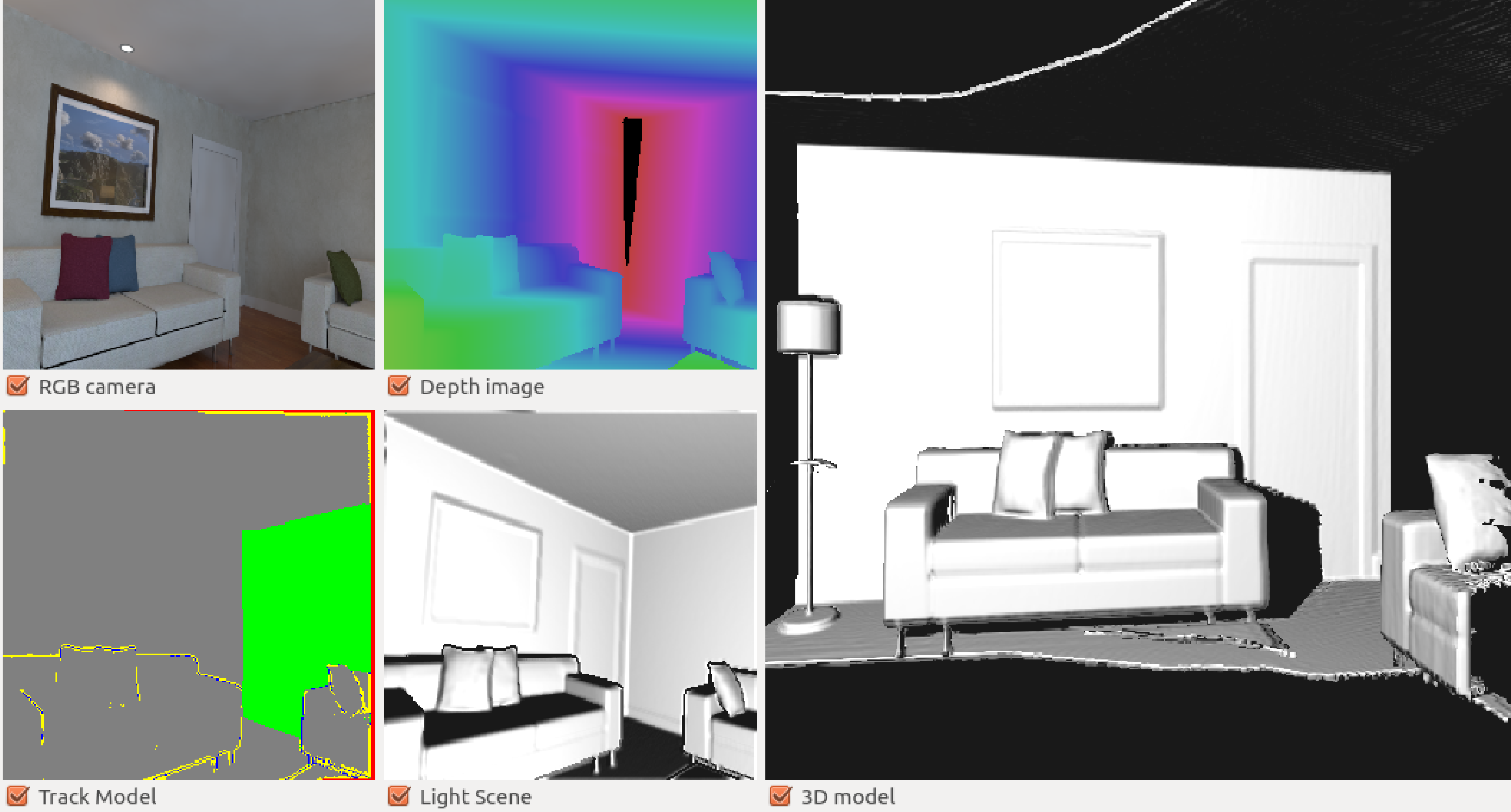


Figure 7.1: SLAMBench [91] implementation of Kinect Fusion [91]. The input from s RGB-D camera is used to reconstruct a 3D scene in real-time. Top-left: the video (RGB) and depth (D) inputs from the camera. Bottom-middle: the point cloud constructed from the camera inputs. Bottom-left: the results of tracking the incoming point cloud against point cloud generated from the last known camera pose. Right: the reconstructed 3D models of the scene.

Accelerator	CUs	Frequency	OpenCL Version (Vendor)
Intel i7-4850HQ	8	2.3 GHz	1.2 (Apple)
Intel Iris Pro 5200	40	1.2 GHz	1.2 (Apple)
NVIDIA GT 750M	2	925 MHz	1.2 (Apple)
AMD A10-7850K	4	1.7 GHz	2.0 (AMD)
AMD Radeon R7	8	720 MHz	2.0 (AMD)
Intel i7-2600K	8	3.4 GHz	1.2 (Intel)
NVIDIA GTX 550 Ti	4	1.9 GHz	1.1 (NVIDIA)
Intel Xeon E5-2620	24	1.2 GHz	1.2 (Intel)
Intel Xeon Phi 5110P	236	1.0 GHz	1.2 (Intel)
NVIDIA Tesla K20m	13	705 MHz	1.2 (NVIDIA)
Intel Xeon E3-1285	8	3.5 GHz	1.2 (AMD)
AMD Radeon HD 6970	24	880 MHz	1.2 (AMD)
Intel Xeon E3-1285	8	3.5 GHz	1.2 (Intel)
Intel Iris Pro P6300	47	1.2 GHz	1.2 (Intel)

Table 7.2: OpenCL Hardware Configuration: Key: CU – Number of OpenCL Compute Units.

In many computer vision related fields, KF is categorised as a Simultaneous Localisation and Mapping (SLAM) application. The inherent complexity in KF is in the calculation of the six degree-of-freedom pose from a moving camera using only the stream of depth images being published by the camera. If the calculate pose is more than a couple of centimeters off then the reconstruction will fail.

Figure 7.1 shows Kinect Fusion in action. Firstly, the inputs of the camera – the RGB and depth images – are seen in the two frames in the top-left of the Figure. These inputs are turned into a point cloud that represents the scene from the current view point of the camera. Next this point cloud is matched (or tracked) against a point cloud that estimates the view from the last known position of the camera. The results of the tracking stage are visualised in the bottom-left frame in Figure 7.1 where grey represents successfully tracked pixels and other colours indicating failure. If the point clouds are tracked successfully the current pose of the camera is obtained and the point cloud can be fused into a persistent model of the scene. It is this persistent model that the user can freely move around and explore and is shown in the right frame of Figure 7.1. More detailed information about KF can be found in [92] and a open-source implementation is provided by SLAMBench [91].

Kinect Fusion is both a challenging and interesting application in the context of this

thesis. Primarily, it provides an application that has real requirements for hardware acceleration to operate. The reason behind this is that KF needs to process the incoming RGB-D data at the frame rate of the camera to minimise the change in camera pose between subsequent frames. Hence, KF has a Quality-of-Service (QoS) target equal to the frame rate of the camera, which is 30 frames per second (FPS) on a Microsoft Kinect camera. Dropping below this frame rate means that pose changes, in both the camera and the subject, have the potential to become greater and, subsequently, finding correspondences between frames becomes increasingly difficult. Finally, Kinect Fusion is also interesting from an implementation perspective as: (1) there is an abundance of parallelism which can be exploited to improve its performance, and (2) that the performance critical path of the application requires sustaining the execution of a large number of different kernels over a prolonged period of time. Hence, KF will extensively stress all components of the Tornado framework: from the programming API, to the compiler, through to the runtime system.

7.3.1 Processing Pipeline

As an application Kinect Fusion comprises of two parts – a visualisation component and a processing pipeline – it is the latter that is the performance critical part of the application and as such is the focus for implementation. The key to understanding the working of the KF pipeline is knowing that as incoming frames are successfully tracked they are fused together to form a persistent model of the scene. Over many pipeline executions a large number of discrete point clouds (from different camera positions) will be fused together to form a very accurate model of the scene. What makes the pipeline non-intuitive to understand is that there is a feedback loop (or data dependency) between the tracking and raycasting stages that spans successive invocations of the pipeline. This can be seen in Figure 7.2. More formally, the KF processing pipeline is comprised of the following six stages:

1. **Acquisition** obtains the next RGB-D frame: either from a camera or from a file.

2. **Pre-processing** is responsible for cleaning and interpreting the raw data by: applying a bilateral filter to remove anomalous values, rescaling the input data to represent distances in millimeters and, finally, building a pyramid of vertex and normal maps using three different image resolutions.

3. **Tracking** estimates the difference in camera pose between two point clouds. This is achieved by matching (or tracking) the point cloud derived from the

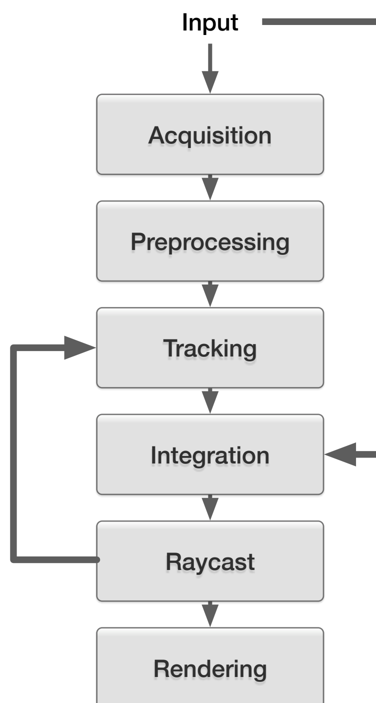


Figure 7.2: The six stages of the Kinect Fusion processing pipeline. Arrows indicate the flow of data between stages.

incoming depth image against the point cloud estimate generated in step (5). Tracking is performed using a technique called Iterative Closest Point (ICP) [13, 126].

4 **Integrate** fuses any successfully tracked point cloud into the persistent model of the scene.

5 **Raycast** uses the persistent model to construct the point cloud estimate that is used in step (3).

6 **Rendering** renders the persistent model of the scene by using the same raycasting technique of the previous stage.

As Kinect Fusion was initially implemented to target GPGPUs (via CUDA C++) each stage of the processing pipeline is further decomposed into as a series of kernels. These kernels are often executed multiple times: sometimes with different parameters (e .g. when forming an image pyramid) or as part of an iterative algorithm (as is the case for ICP). A full breakdown of each pipeline stage is given in Table 7.3 showing the total number of kernel invocation required in each stage. The highlighted variation in kernel

Kernel	Stage	Invocations
mm2meters	Preprocessing	1
bilateral filter	Preprocessing	1
half sample	Tracking	3
depth to vertex	Tracking	3
vertex to normal	Tracking	3
track	Tracking	1 - 19
reduce	Tracking	1 - 19
integrate	Integration	0 - 1
raycast	Raycast	0 - 1
render depth	Rendering	0 - 1
render track	Rendering	0 - 1
render volume	Rendering	0 - 1
Total	-	18 - 54

Table 7.3: Breakdown of a single execution of the Kinect Fusion pipeline into device-side kernels. Note that the entire pipeline has a non-deterministic number of kernel invocations that is tied to the performance of the tracking algorithm.

invocations is due to the non-deterministic nature of the tracking (ICP) algorithm: the quicker a solution converges the less iterations are require and therefore less kernels are executed. Consequently, processing a single frame of RGB-D data will require a minimum of 18 distinct kernel executions; rising to a maximum of 54 in the worst case scenario. What is important to note is that to achieve a frame rate of 30 FPS the application must sustain the execution of between 540 and 1620 kernels each second. This means that the runtime performance of Kinect Fusion is dependent on both: the performance of individual kernels and sustaining a high throughput of kernels over extended periods of time.

7.3.2 Tracking Algorithm

The most complex part of the Kinect Fusion pipeline is the tracking stage. The reason for this is the use of the Iterative Closest Point (ICP) algorithm [13, 126] to estimate the difference in camera pose between two point clouds. The ICP algorithm has two stages: (1) it tries to find correspondences between two point clouds – returning the error associated with each correspondence and, (2) it uses a least-squares approach to identify a new camera pose which minimises this error. Presently, Single Value Decomposition (SVD) is used during error minimisation. Next, the algorithm iterates

until the correspondence error is below a pre-configured threshold. However, to try to improve the performance of the ICP algorithm (allow it to converge faster) Kinect Fusion uses a multi-scale implementation. This is achieve by sub-sampling depth images into half and quarter size images and point clouds to form an image pyramid. To use this image pyramid the ICP algorithm starts running on the smallest size of point clouds to provide a crude approximation of the camera pose. Once the camera pose is within a desired error bound the algorithm then moves to the next level of the pyramid. This repeats until either no pose can be found or there are no more levels of the pyramid left. Importantly this means that a crude approximate to the camera pose can be estimated quickly from smaller point clouds and then refined to provide a more accurate estimate using the larger point clouds. Consequently, this improves execution times by reducing the number of times the algorithm has to iterate over the largest point clouds.

The simplest form of the ICP algorithm is easily implementable on hardware accelerators, such as a GPGPU, by splitting it up so that the correspondences are found on the accelerator and error minimization is performed on the host. The first step is a highly parallel per-pixel map operation that is amenable to hardware acceleration and the second step is a reduction step that is more complex to accelerate. In this form of the ICP algorithm a single, but manageable, one-way data transfer exists on the critical path of the application between the device to the host. However, the use of a multi-scale version of ICP compounds this problem by inserting multiple bi-direction data transfers onto the performance critical path of the application. For instance, the `track` kernel maps each pixel in the camera image onto an eight-wide vector of type `Float8`. Consequently, this means that the tracking stage of the pipeline will be performing regular data transfers of 2.34 MB (320×240), 600 KB (160×120), and 150 KB (80×60) to the host. Additionally, in the opposite direction the host needs to transfer the newly calculated pose back to the device at the end of each iteration and as each pose is represented by a 4×4 matrix these transfers are only 64 bytes in size. This problem is shown in Figure 7.3 where computation is represented by rectangles and data transfers as diamonds.

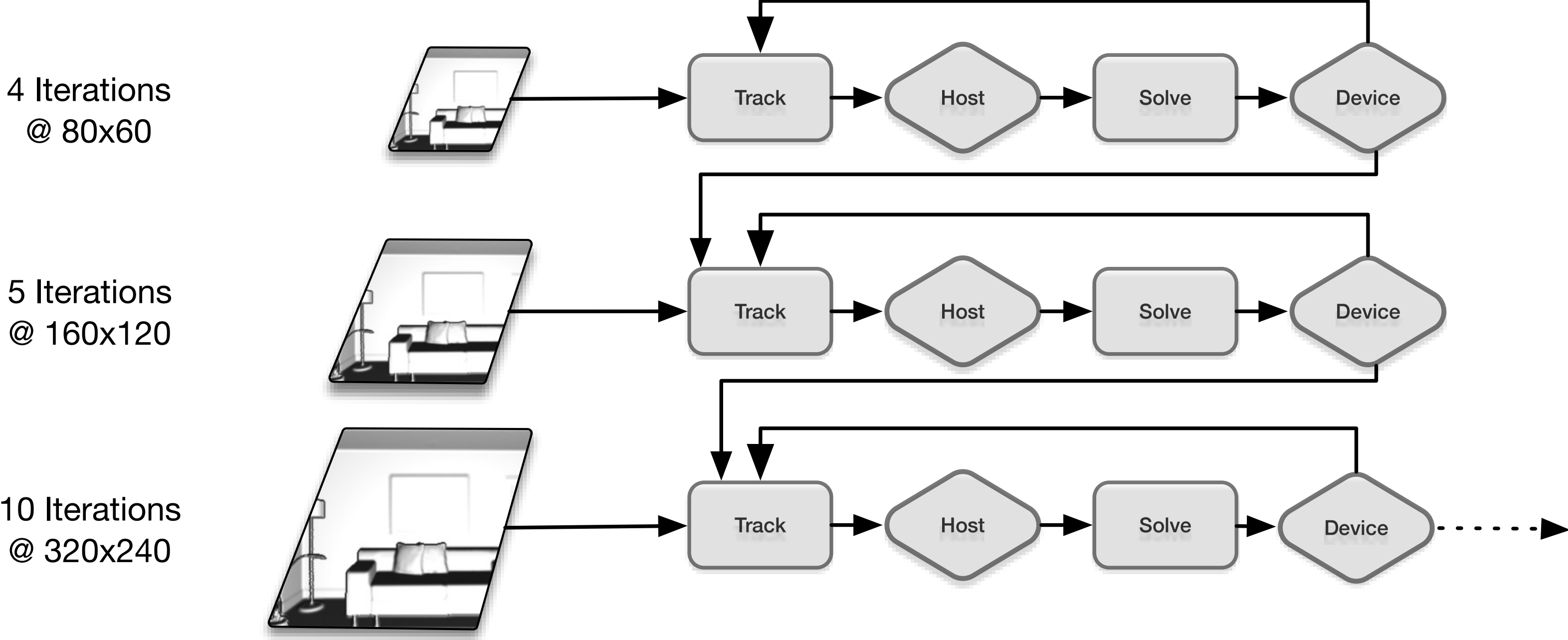


Figure 7.3: The data-flow of the hardware accelerated multi-scale ICP algorithm used in Kinect Fusion. Left: the sub-sampled depth images and point clouds along with their dimensions. Right: data-flow between computations (rectangles) with explicit data transfers between device and host.

7.3.3 Measuring Performance and Accuracy

One of the complexities of trying to empirically compare computer vision applications is that performance is often subjective. Normally, this is due to the fact that algorithmic quality is determined through the user experience: does the user notice slow performance and is it accurate enough for their needs? Nevertheless, to allow a quantitative evaluation across different implementations of Kinect Fusion the evaluation metrics as set out by SLAMBench [91] will be used. SLAMBench provides multiple reference implementations of Kinect Fusion and has ready-made infrastructure available for measuring the performance and accuracy of each implementation. Thus enabling meaningful comparisons to be made across different implementations of Kinect Fusion. SLAMBench achieves this by using synthetically generated datasets where each data set comprises a stream of RGB-D frames that have associated camera poses. Hence, in SLAMBench the quality of each implementation is determined by evaluating how well the estimated camera trajectory compares against the provided ground truth. This measure of quality is referred to as the absolute trajectory error (ATE). The datasets used by SLAMBench are taken from the ICL-NUIM dataset [57]. For all experiments in this evaluation the Living Room (trajectory two) was used and has 882 frames that should take approximately one minute to process at 30 FPS. The measured ATE for both the serial and hardware accelerated implementations of Kinect Fusion are shown in Tables 7.4 and 7.5 respectively. The lower the ATE the better, however, it is unlikely to achieve an ATE of zero due to the discretisation of the persistent model into voxels. Therefore, a good ATE is one that is not more that a couple of voxel widths out. In the experiments the voxel size is fixed at 0.01875 m^3 so any ATE that is less than 0.05 m is acceptable. Measuring the quality (or accuracy) of a reconstruction is only half of the problem. A good implementation should be able to reconstruct the scene to the desired accuracy in the shortest amount of time possible. Therefore, to evaluate the performance of an implementation the average frame rate – measured in frames per second (FPS) – achieved when processing the entire dataset is also.

Type	Accelerator	C++	Java
Multi-core	AMD 10K-7850K	0.0206	0.0119
	Intel i7-2600K	0.0206	0.0119
	Intel i7-4850HQ	0.0206	0.0117
	Intel Xeon E3-1285	0.0206	0.0119
	Intel Xeon E5-2620	0.0206	0.0118

Table 7.4: Observed Absolute Trajectory Errors from all serial implementations of Kinect Fusion (smaller is better). All measurements are in meters and each voxel is a cube with dimension 0.01875 m.

Type	Accelerator	OpenMP	OpenCL	Tornado
Multi-core	AMD 10K-7850K	0.0206	Fail	0.0119
	Intel i7-2600K	0.0207	0.0207	0.0118
	Intel i7-4850HQ	Comp.	Error	0.0118
	Intel Xeon E3-1285	0.0206	0.0207	0.0118
	Intel Xeon E5-2620	0.0206	0.0208	0.0118
Many-core	Intel Xeon Phi 5110P		0.0207	0.0119
Embedded GPGPU	AMD Radeon R7		Fail	0.0119
	Intel Iris Pro 5300		0.0207	0.0119
	Intel Iris Pro P6300		0.0207	0.0119
External GPGPU	NVIDIA GT 750M		Fail	0.0119
	NVIDIA GTX 550Ti		0.0207	0.0119
	AMD Radeon HD 6970		0.0206	0.0119
	NVIDIA Tesla K20m		0.0207	0.0119

Table 7.5: Observed Absolute Trajectory Errors from all hardware accelerated implementations of Kinect Fusion (smaller is better). Key: Fail - ATE exceeded accuracy requirements, Error - fatal runtime error, Comp. - unable to compile. All measurements are in meters and each voxel is a cube with dimension 0.01875 m.

7.4 Implementation

To support this thesis it needs to demonstrate that Tornado can be used to implement Kinect Fusion and achieve a high-level of performance. By doing this it will demonstrate that the barriers preventing mainstream heterogeneous application development can be overcome. Hence, the challenge of producing a high-performing version of Kinect Fusion in Java that can achieve the desired QoS targets has been undertaken. The remainder of this Section will discuss how this has been done.

7.4.1 Serial Java

As Tornado is a Java-based programming framework it is necessary to have a reference implementation written in Java that does not use any form of hardware acceleration. This implementation is derived from the serial C++ implementation provided by SLAMBench. During the porting from C++ to Java particular care was taken to ensure that the Java implementation produces identical results to the C++. In the majority to cases it was possible to obtain bit-exact result, however, in a few cases this was not possible due to differences in how floating-point numbers are handled in both Java and by different hardware architectures – like NVIDIA GPGPUs. Hence, in these cases it was satisfactory to come within five units of last place (ULP) of the C++. It should be noted that despite individual kernels producing near identical results during isolated unit-testing, each implementation can produce slightly differing results when combined together to form a larger system.

One of the problems from porting across different programming languages is that it is not always possible to find equivalent language features or third-party libraries. For example, Java does not support unsigned integers and so a lot of changes were required in the acquisition stage to decode the raw depth images correctly. To ensure that a pure Java implementation is created no form of Foreign Function Interface (FFI) or calls native libraries are allowed. Additionally, only a single dependency on third-party code exists: a call to the SVD method in the the EJML linear algebra library [3].[2]

During a preliminary performance analysis, shown in Table 7.8, it was observed that serial C++ implementation is between 3.4-7.9$\times$ faster than the Java. Despite the C++ implementation outperforming Java, it barely manages to achieve 4 FPS which is significantly lower than the desired QoS target of 30 FPS. Hence, it is unlikely that

[2] In the future this dependency could be removed.

System	Device	C++	Java	Slowdown
Laptop	Intel i7-4850HQ	3.69	0.87	4.24×
Desktop 1	AMD 10K-7850K	3.14	0.40	7.85×
Desktop 2	Intel i7-2600K	.	1.21	.
Server 1	Intel Xeon E5-2620	2.40	0.71	3.38×
Server 2	Intel Xeon E3-1285		1.30	

Table 7.6: Performance of C++ and Java serial implementations measured in Frames Per Second (FPS).

either implementation could achieve 30 FPS without some form of hardware acceleration would be required. It is also important to note that the Java implementation has to overcome a significantly larger gap in performance than the C++.

```java
final long[] timings = new long[7];
timings[0] = System.nanoTime();

// acquisition stage
boolean haveDepthImage = depthCamera.pollDepth(depthImageInput);
videoCamera.skipVideoFrame();
while (haveDepthImage) {
    timings[1] = System.nanoTime();

    // preprocessing stage
    preprocessing();
    timings[2] = System.nanoTime();

    // tracking stage
    boolean hasTracked = track();
    timings[3] = System.nanoTime();

    // integration stage
    final boolean doIntegrate =
        (hasTracked && frames % integrationRate == 0) || frames <= 3;
    if (doIntegrate)
        integrate();
    timings[4] = System.nanoTime();

    // raycasting stage
    final boolean doUpdate = frames > 2;
    if (doUpdate)
        updateReferenceView();
    timings[5] = System.nanoTime();

    // rendering stage
    if (frames % renderingRate == 0) {
        renderTrack(renderedTrackingImage, trackingResult.getResultImage());
        renderDepth(renderedDepthImage, filteredDepthImage, nearPlane, farPlane);
        final Matrix4x4Float scenePose = sceneView.getPose();
        renderVolume(renderedScene, volume, volumeDims,
                     scenePose, nearPlane, farPlane * 2f,
                     smallStep, largeStep, light, ambient);
    }
    timings[6] = System.nanoTime();

    // visualisation code elided
    ...

    // next frame
    timings[0] = System.nanoTime();

    // acquisition stage
    haveDepthImage = depthCamera.pollDepth(depthImageInput);
    videoCamera.skipVideoFrame();
}
```

Listing 7.1: The Java version of the Kinect Fusion pipeline.

```
// populate first row of the pyramid with incoming depth image
pDepth[0] = filteredDepthImage;
pVertex[0] = currentView.getVerticies();
pNormal[0] = currentView.getNormals();

// sub-sample depth images
for (int i = 1; i < iterations; i++)
     ImagingOps::resize(
        pDepth[i],
        pDepth[i - 1],
        2,
        eDelta * 3,
        1);

// build vertex and normal maps for each pyramid level
final Matrix4x4Float scaledInvK = new Matrix4x4Float();
for (int i = 0; i < iterations; i++) {
    final Float4 camera = mult(scaledCamera, 1f / (float) (1 << i));
    getInverseCameraMatrix(camera, scaledInvK);
    GraphicsMath::depthToVertex(pVertex[i], pDepth[i], scaledInvK);
    GraphicsMath::vertexToNormal(pNormal[i], pVertex[i]);
}
```

Listing 7.2: Exert from the `track` method of the Kinect Fusion pipeline (see Listing 7.1) that builds an image pyramid from the incoming depth image.

7.4.2 Tornado

The primary goal of the Tornado implementation is to demonstrate portability: that it can execute across all of the hardware specified in Table 1.1 irrespective of whether the underlying device is a multi-core processor, a GPGPU or other many-core accelerator. The Tornado implementation is created by adapting the serial Java implementation using the Tornado API that was described in Chapter 3. There are two things that need to be changed: (1) the processing pipeline needs to be updated to make use of task-schedules, and (2) any task-parallel code that is executed by a task-schedule needs to be annotated with the Tornado API.

7.4.2.1 Building The Image Pyramid

To illustrate the steps needs to port the Java implementation into Tornado, this Section will walkthrough the code changes required to execute code on a hardware accelerator. The example used is code taken from the first step of the tracking stage where the incoming depth image is sub-sampled at both half and quater sizes to create a image pyramid. It is this pyramid that is later used by the multi-scale ICP algorithm to determine the new pose of the camera. The code that builds the image pyramid from the serial implementation of Kinect Fusion is shown in Listing 7.2. For this code to execute on a hardware accelerator it needs to be converted into a task-schedule.

A task-schedule captures the coordination of control through a number of tasks. Therefore, converting this code to build a task schedule is a matter of creating a task-schedule and replacing each method call with code that instead inserts it into the task-schedule. These changes are shown in Listing 7.3 along with a number of other minor modifications. Apart from the first line that is needed to define the task-schedule the last three lines are required to ensure that the `projectReference` variable is not cached on the remote device and all tasks within the task-schedule are executed on the same hardware accelerator (`oclDevice`). The final change that is required is that a copy of the matrix containing the scaled camera intrinsics is kept for each level of the pyramid to allow all the matricies to be cached on the device. Once the task-schedule has been created it can be replaced in the original code with a single line, shown in Listing 7.4, that triggers the execution of the task-schedule. Note that task-schedules are designed to be constructed once and executed many times – which is why the definition on a task-schedule is logically seprated from its invocation. In order to visualise what is happening within the task-schedule Figure 7.4 is provided. As in Chapter 3 any data transfer enclosed within the black box can be optimised away by Tornado. Hence, as all data movement happens within Tornado's optimisation scope no explicit data transfers are required in the execution of this task-schedule. At this stage the task-schedule will only be able to execute serial versions of tasks on the hardware accelerator and in most cases this is likely to introduce a slowdown in performance. In order to improve device-side performance the developer must now annotate the task-parallel code contained within each task using the Tornado API – this will be discussed next in Section 7.4.2.2.

Two key features of Tornado is demonstrated by the `buildPyramid` example: (1) code re-use and (2) the ability to execute task-schedules with complex data or control dependencies. A key design goal of Tornado is to avoid introducing changes to the computational logic and the effect this has is seen here because each task in the task-schedule is created from the existing code and uses exactly the same parameters as before. This means that the computational logic of the application can be re-used, it is just the coordination logic of the application that is updated. Next it should also be clear that Tornado is able to execute a complex task-schedule transparently. This means that no effort is required on the developers part to ensure that memory updates that happen in the task-schedule appear in program order on the host.

```
buildPyramidSchedule = new TaskSchedule("buildPyramid");

// Resize depth image for each level of the pyramid
for (int i = 1; i < iterations; i++) {
    // execute resize(...) on accelerator
    buildPyramidSchedule
        .task("resize" + i,
              ImagingOps::resize,
              pDepth[i],
              pDepth[i - 1],
              2,
              eDelta * 3,
              2);
}

// Calculate verticies and normals for each level of the pyramid
for (int i = 0; i < iterations; i++) {
    // generate scaled inverse camera matrices for all levels
    final Float4 camera = mult(scaledCamera, 1f / (float) (1 << i));
    getInverseCameraMatrix(camera, scaledInvK[i]);

    // execute depthToVertex(...) and then vertexToNormal(...) on accelerator
    buildPyramidSchedule
        .task("d2v" + i,
             GraphicsMath::depthToVertex,
             pVertex[i],
             pDepth[i],
             scaledInvK[i])
        .task("v2n" + i,
              GraphicsMath::vertexToNormal,
              pNormal[i],
              pVertex[i]);
}

// ensure projectReference is not cached on the accelerator
buildPyramidSchedule
    .volatile(projectReference)
    .mapAllTo(oclDevice);
```

Listing 7.3: The definition of a task-schedule that is equivalent to Listing 7.2 in the Tornado implementation of Kinect Fusion. Now each method call to `resize`, `depthToVertex` and `vertexToNormal` is executed on a hardware accelerator.

```
buildPyramidSchedule.execute();
```

Listing 7.4: Once the `buildPyramid` task-schedule has been defined (as in Listing 7.3) it can be replaced in the original code (Listing 7.2) with this single line.

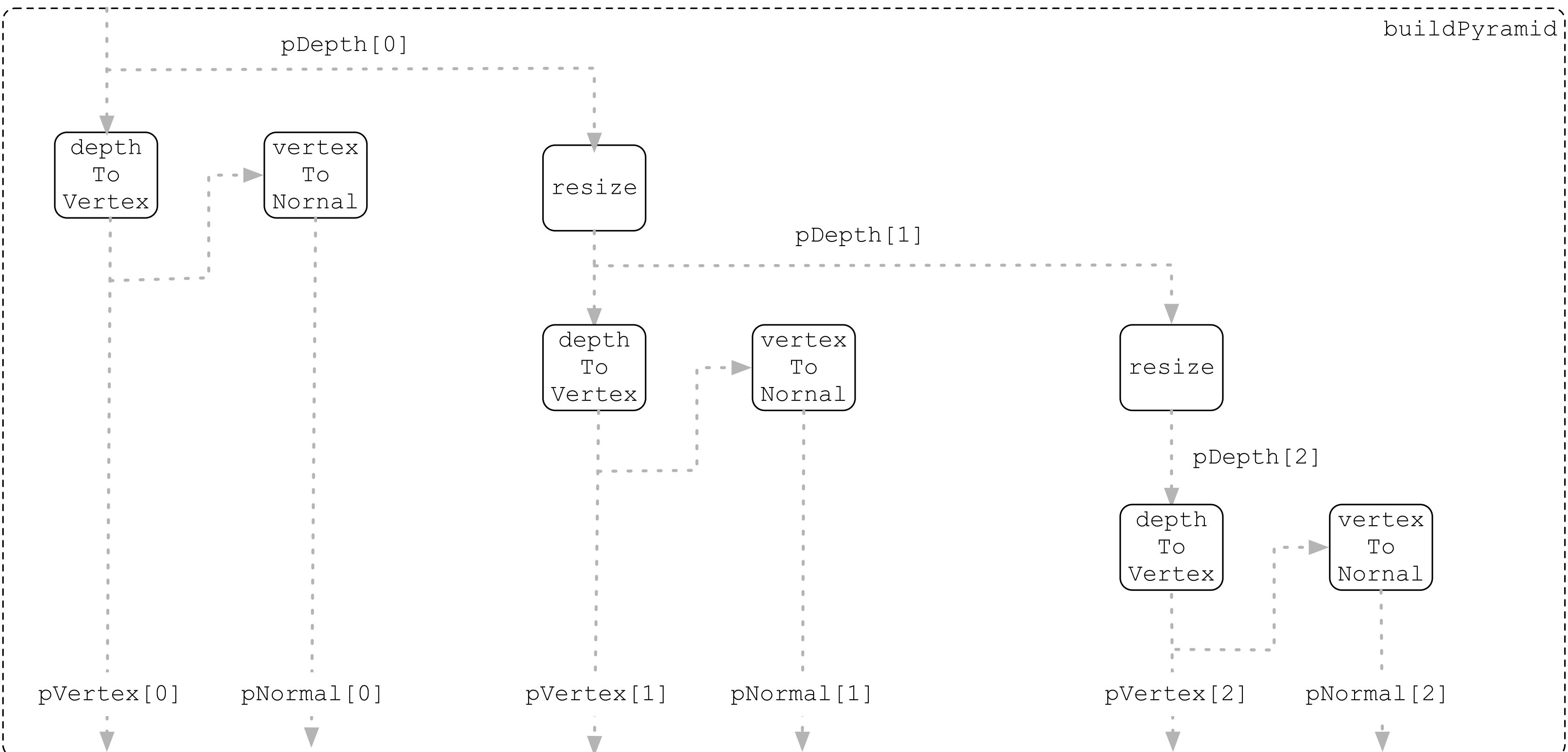


Figure 7.4: A visual representation of the data-flow between components contained within the `buildPyramid` task-schedule as defined in Listing 7.3. Notice the distinct lack of data transfers this task-schedule operates entirely on data cached on the device (as represented by the dashed-lines). This is because the input (`pDepth[0]`) is created on the device during the previous pipeline stage.

```
public void depthTovertex(
                ImageFloat3 verticies,
                ImageFloat depths,
                Matrix4x4Float invK) {

    for (@Parallel int y = 0; y < depths.Y(); y++) {
        for (@Parallel int x = 0; x < depths.X(); x++) {

            final float depth = depths.get(x, y);
            final Float3 pix = new Float3(x, y, 1f);

            final Float3 vertex = (depth > 0) ?
                  mult(rotate(invK, pix),depth) :
                    new Float3();

            verticies.set(x, y, vertex);
        }
    }
}
```

Listing 7.5: The final version of the `depthToVertex` method after being annotated with the Tornado API. The Tornado API is designed to create backwards compatable code through the use of Java annotations. In this example the `@Parallel` annotation is used to instruct Tornado that where possible execute each iteration of a loop in parallel and that it is safe for it to do so.

7.4.2.2 Parallelising The Image Pyramid

The final part of porting an application into Tornado is annotating any code to exploit task-parallelism. As described earlier in Section 3.2.1 task-parallel loops can be parallelised by the JIT compiler inside each Tornado Virtual Machine Client. The choice of how the loop is parallelised is delegated to the JIT compiler as it should have full knowledge of the preferences of the target hardware. All the developer is required to do is annotate the appropriate induction variables inside the task-parallel loops. This is done through use of the `@Parallel` annotation as shown in Listing 7.5. At this stage developers need to be careful not to use the Tornado API incorrectly as this will result in incorrect behaviour in Tornado. For example, the current implementation of Tornado will assume that it is always possible to parallelise loops that are `@Parallel` annotated. In the future, Tornado should be able to either verify that parallelisation is safe or alternatively determine if a loop is parallelisable (this way it would be possible to avoid annotating loops completely). For reference, all code that is parallelise by Tornado has been validated against the results obtained by both the serial Java and C++ implementations and as before are required to be within 5 ULP of the serial versions.

```java
final long[] timings = new long[7];
timings[0] = System.nanoTime();

// acquisition stage
boolean haveDepthImage = depthCamera.pollDepth(depthImageInput);
videoCamera.skipVideoFrame();
while (haveDepthImage) {
    timings[1] = System.nanoTime();

    // preprocessing stage
    preprocessing();
    timings[2] = System.nanoTime();

    // tracking stage
    boolean hasTracked = track();
    timings[3] = System.nanoTime();

    // integration stage
    final boolean doIntegrate =
        (hasTracked && frames % integrationRate == 0) || frames <= 3;
    if (doIntegrate)
        integrate();
    timings[4] = System.nanoTime();

    // raycasting stage
    final boolean doUpdate = frames > 2;
    if (doUpdate)
        updateReferenceView();
    timings[5] = System.nanoTime();

    // rendering stage
    if (frames % renderingRate == 0)
        renderSchedule.execute();
    timings[6] = System.nanoTime();

    // visualisation code elided
    ...

    // next frame
    timings[0] = System.nanoTime();

    // acquisition stage
    haveDepthImage = depthCamera.pollDepth(depthImageInput);
    videoCamera.skipVideoFrame();
}
```

Listing 7.6: The final Tornado version of the Kinect Fusion pipeline. Notice how the pipeline structure does not need to be altered to support hardware acceleration via Tornado.

7.5 Evaluating Portability

One of the core objectives of this thesis is to demonstrate that a hardware accelerated application can be written once and executed across a variety of different hardware accelerators. Therefore, it is necessary to compare how well Tornado can produce portable applications relative to state-of-the art heterogeneous programming languages. For this section of the evaluation, three different implementations of Kinect Fusion are used: OpenCL, OpenMP, and Tornado. These three implementations are evaluated across five different systems that between them contain four distinct classes of hardware accelerator (as described earlier in Table 7.1). Each system comprises of a multi-core processor, along with a minimum of one GPGPU that can be used for acceleration. The aim of the experiments are to understand: (1) how many hardware accelerators can be used by each implementation without requiring code modifications, and (2) what levels of performance are being achieved. At this point each implementation is tested without any performance tuning or specialisation and should therefore represent out-of-the-box performance. As a result this set of experiments is designed to provide a clear understanding of the tradeoff between *portability* and *performance*.

All the experimental data for this set of portability experiments is provided in Table 7.8. The stand out observation is that there was only one implementation that was able to both execute and generate a valid result across all thirteen accelerators: Tornado. There are a few reasons for this the first was that the OpenMP implementation was only capable of running on multi-core processors and so could not target any of the available GPGPUs. However, when running on multi-core processors the OpenMP implementation was able to produce valid results in four out of the five supported platforms. Out of these four valid implementations the OpenMP results were closest in performance to Tornado and both were outperformed by OpenCL. Comparing Tornado with OpenMP saw a worse-case performance loss of 30% – on the AMD 10K-7850K – and a best-case performance improvement of 9% – on the Intel Xeon E5-2620. In the case where OpenMP failed to produce a result the problem was due to the operating system (OSX) not supporting OpenMP.

The OpenCL implementation failed to produce valid results on all devices: failing in in four out of thirteen cases. The core problem with the OpenCL implementation of Kinect Fusion is that it makes two assumptions about: (1) work-group dimensions, and (2) the amount of local memory available on each device. If either of these assumptions prove incorrect for a target device the application fails. This failure could manifest in

a number of different ways but the two encountered during the experiments were: a working implementation that fails silently and produces an invalid result; or a segmentation fault that prematurely kills the application. Whether an implementation falls into one category or the other is determined by both the characteristics of the device and the specific implemention of OpenCL that is being used. The actual code that creates this problem is discussed in a brief aside in Section 7.5.1 where the outcode is that this problem is completely avoidable. However, it is important to understand that to make this code avoidable would requires extra effort from the developer to write code that first checks what resources are available of the device before trying to use them. In contrast, developers using Tornado avoid this issue and the need to write extra code because resource usage – like the number of compute units, the maximum work-group sizes, and the amount of local memory – is automatically determined by the runtime system based on the actual characteristics of the target device. What makes Tornado really flexible in situations where the application is being run on a unknown configuration is *dynamic configuration*. Later Section 7.8 will discuss how dynamic configuration allows developers to change an applications resources usage without having to re-compiling the application. Thus, an issue like this one experienced with OpenCL is easily resolvable by someone other than a developer.

In terms of performance the highest performing implementation is OpenCL, with it achieving up to 138 FPS on the NVIDIA Tesla K20m GPGPU. As expected the Tornado implementation, albeit more portable, is much lower performaning when compared with the OpenCL. For instance, on the NVIDIA Tesla K20m it achieves just under 39.71 FPS – approximately 28% of the performance of OpenCL. Although, this seems quite poor the real value of the Tornado implementation can be seen when considering the performance improvement over the original Java implementation. Here a maximum speed-up of 55 $\times$ is measured obtained on the NVIDIA Tesla K20m by accelerating the Java implementation from 0.71 – Server 1 in Table 7.6 – to 39.71 FPS. More generally, it should also be noted that in three cases – the Intel Iris Pro P6300, NVIDIA GTX550Ti and the NVIDIA Tesla K20m – Tornado implementations either met or exceeded the 30 FPS QoS target. This last point is significant because it means a computationally demanding application like Kinect Fusion can be written without requiring in-depth knowledge of hardware accelerators – only a basic understanding of writing parallel code was needed. The question that naturally stems from this is whether the 3-4 $\times$ performance gap with OpenCL can be bridged? To answer this question Section 7.6 will evaluate how the Tornado implementation can be specialised

Type	Accelerator	OpenMP	OpenCL	Tornado
Multi-core	AMD 10K-7850K	7.87	Fail	5.28
	Intel i7-2600K	19.40	30.11	17.44
	Intel i7-4850HQ	Unsup.	Fail	15.01
	Intel Xeon E3-1285	22.32	36.54	20.88
	Intel Xeon E5-2620	19.63	29.02	21.60
Many-core	Intel Xeon Phi 5110P		24.27	3.41
Embedded GPGPU	AMD Radeon R7		Inv.	16.80
	Intel Iris Pro 5300		57.95	24.84
	Intel Iris Pro P6300		94.23	52.76
External GPGPU	NVIDIA GT 750M		Inv.	20.15
	NVIDIA GTX 550Ti		4.43	40.77
	AMD Radeon HD 6970		135.34	11.05
	NVIDIA Tesla K20m		138.10	39.71
Totals	Valid Results	4	9	13
	Invalid Results	0	2	0
	Failed To Execute	0	2	0
	Unsupported Platform	1	0	0
	Success	80%	69%	100 %

Table 7.7: Observed performance of Kinect Fusion across all available hardware. Performance numbers are in terms of Frames Per Second (FPS). Key: Unsupp. – means that the programming language was not supported on that platform; Fail – means the application experienced a failure that prevented it from completing; Inv. – means that the application completed but the result was invalid (i.e. the Absolute Trajectory Error was above 5cm).

for a specific device.

```
1 __kernel void reduceKernel (
2      __global float * out,
3     __global const TrackData * J,
4     const uint2 JSize,
5     const uint2 size,
6     __local float * S  // local memory is allocated by the host API
7 )
```

Listing 7.7: OpenCL Bug: Kernel Code

7.5.1 Aside: OpenCL Portability Bug

In Section 7.5 highlighted that the OpenCL implementation of Kinect Fusion failed to execute properly on four devices. The reason for this is that the code contains hard-coded assumptions about the resources available on a hardware accelerator. The offending code is highlighted in Listing 7.7 that shows the OpenCL kernel and Listing 7.8 that shows the host-side code. Here the problems are caused in the host-side code where the resources of the device are being assumed in lines two and three. In this example the host-side code allocates some local memory for the kernel in line 26. However, there is no check to see whether the amount of local memory requested is available on the device. It is this assumption about local memory that prevented the OpenCL version of Kinect Fusion running on the AMD 10K-7850K and AMD Radeon R7 – as this was a low-end device that did not have very many resources. The second problem is found in line 19 where the same assumption about work-group sizes is made when launching the kernel via `clEnqueueNDRangeKernel`. In this situation a check normally has to be made to ensure that the requested work-group size is supported by the device. This was not the case on the Intel i7-4850HQ and as a consequence the reduction kernel could not be launched.

In comparison to OpenCL, Tornado does not suffer these problems because the Tornado Virtual Machine is designed to check both of these properties before launching a kernel. This is why the Tornado implementation of Kinect Fusion is able to execute across all thirteen devices in Section 7.5. The difference between these two approaches is that OpenCL is designed as a low-level heterogeneous programming language that expects developers to make these checks – opposed to Tornado that is a high-level heterogeneous programming language that performs these checks on the users behalf.

```
 1 // reduction parameters
 2 static const size_t size_of_group = 64;
 3 static const size_t number_of_groups = 8;
 4
 5 ...
 6
 7 clError = clSetKernelArg(reduce_ocl_kernel,
 8     arg++,
 9     size_of_group * 32 * sizeof(float), // specifies amount of local memory
10     NULL);
11 checkErr(clError, "clSetKernelArg");
12
13 // Below is the hard-coding of the workgroup dimensions of the reduction kernel
14 size_t RglobalWorksize[1] = { size_of_group * number_of_groups };
15 size_t RlocalWorksize[1] = { size_of_group };
16
17 // launch the kernel with hardcoded workgroup sizes
18 clError = clEnqueueNDRangeKernel(commandQueue, reduce_ocl_kernel, 1,
19     NULL, RglobalWorksize, RlocalWorksize, 0, NULL, NULL);
20 checkErr(clError, "clEnqueueNDRangeKernel");
21
22 clError = clEnqueueReadBuffer(commandQueue,
23     ocl_reduce_output_buffer,
24     CL_TRUE,
25     0,
26     32 * number_of_groups * sizeof(float),
27     reduceOutputBuffer,
28     0,
29     NULL,
30     NULL);
31 checkErr(clError, "clEnqueueReadBuffer");
32
33 // hardcoded parameter perculates through both host and device code
34 TooN::Matrix<TooN::Dynamic, TooN::Dynamic, float,TooN::Reference::RowMajor>
35         values(reduceOutputBuffer, number_of_groups, 32);
36
37 for (int j = 1; j < number_of_groups; ++j) {
38     values[0] += values[j];
39 }
```

Listing 7.8: This code launches a reduction kernel with a hard-coded work-group size. The top two lines are the problem – these two values should be calculated using the actual properties of the device. This bug leads to two failures: (1) where the amount of local memory allocated in line 26 exceeds the amount available on the device; and (2) where the work-group dimensions used in line 19 to launch the kernel exceeds the maximum supported by the device.

7.5.2 Aside: Issues Effecting Portability

One of the key problems associated with writing heterogeneous code is portability: the ability to build and execute the same code on different machines. For instance, out of a total of 18 possible OpenMP and OpenCL experiments only 13 (72%) produced a valid result and this gets worse when languages like CUDA and OpenACC are also considered. The three main reasons for this were: (1) the inability to compile the benchmark on the target operating system or for a specific device; (2) the code containing incorrect assumptions about the target hardware; and (3) encountering some OpenCL implementation specific behaviour that prevented Kinect Fusion from functioning correctly. Below is a brief summary of the typical issues that hampered the evaluation of Tornado against other heterogeneous programming languages:

Compilation Problems Many heterogeneous programming languages exist but implementations often behave differently on different operating systems. The clearest example of this is the lack of a readily available OpenMP runtime on OSX. However, there are a number of other more subtle issues that were experienced. For example, it was not possible to compile the CUDA implementation of Kinect Fusion on OSX because of an incompatability between CUDA and clang in OSX. Unfortunately, as it was not possible to run CUDA on a single non-linux machine it was dropped from the evaluation. Another problem was that Kinect Fusion assumes particular C++ standard without making any explicit checks to see what versions of C++ the compiler supports. Consequently, the behaviour of Kinect Fusion on the linux platforms was dictated by the version of `gcc` being used – as different versions assume different C++ standards by default. As a result compilation would often fail when compiling C++ templates. Fortunately, in the majority of the cases compilation problems were resolved but doing so did require detailed knowledge of the compiler toolchain and the operating system. Although these problems can sometimes be dismissed as being insignificant it took a disproportional amount of time to compile Kinect Fusion on new platforms before any experimentation could be perfomed – this was especially true when learning a new build system, like CMake, is also involved.

Incorrect Assumptions The largest problem inhibiting portability is that each device has its own characteristics. For example, GPGPUs may have different physical characteristics, such as amount of local memory or maximum number of work

items in a work group. Another problem may be that the device we are programming is not even a GPGPU. In OpenCL, it is the developer's responsibility to handle the differences between devices. This approach requires extra boilerplate code to check whether the assumptions made hold for the target device, and to optionally take corrective action them if they do not. Experience has shown that once developers have a working implementation they often neglect adding this extra code which leads to a portability gap between applications that: work on a single device and others that work universally. Typically, this manifests as hard-coded work group sizes and shared memory allocations which make the implementation biased to a particular device.

Implementation Defined Behaviours Not all OpenCL implementations are created equal: each different vendor seems to interpret the OpenCL specification differently (or sometimes ignore it completely). A good example is the handling of denorms: the specification defines a compiler flag `-denorms-are-zero` but whether this flag works is implementation defined.

7.6 Specialisation

In Section 7.5 the focus was on evaluating the portability of Tornado – i.e. the ability for an application to execute over a range of different devices. What has been demonstrated is that Tornado applications are *portable* – with Kinect Fusion running across all thirteen hardware accelerators – but typically have a *lower performance* than the state-of-the-art heterogeneous programming languages – about 28% of OpenCL on the NVIDIA Tesla K20m. The aim of this Section is to determine whether this performance gap between Tornado and OpenCL can be overcome.To try and improve performance developers are free to implement more *specialised* versions of their applications; however, by doing this they will sacrifice portability. This Section will evaluate two *specialised* implementations of Kinect Fusion that target GPGPUs.

The first step of this optimisation process starts by profiling the Kinect Fusion processing pipeline to identify the most expensive operations. Hence, a breakdown of the time spent in each of the six stages of the Kinect Fusion processing pipeline on Server 1 is given in Figure 7.5. The two configurations that are of interest are the top two – both for the NVIDIA Tesla K20m GPGPU – the others are provided for information only. In these two GPGPU implementations it is clear that the Tornado implementation spends nearly all of its time in the Tracking stages of the pipeline. If the execution times of each of these pipeline stages are compared – as in Figure 7.6 – it is clear that the Tornado implementation only achieves 0.15 $\times$ the performance of the OpenCL implementation. This Figure compares the relative performance of Tornado against OpenCL on both a multi-core processor – the Intel Xeon E5-2620 – and a GPGPU – the NVIDIA Tesla K20m. In the case of the multi-core processor Tornado achieves 0.92$\times$ of the performance of the OpenCL implementation averaged across all pipeline stages and is evenly matched. However, in the case of the GPGPU Tornado achieves a significantly lower performance level of 0.62$\times$ the OpenCL. The main reason fo this is its relatively poor performance in the tracking stage of the pipeline where it only achieves 0.15 $\times$ the performance of the OpenCL. Therefore, in order to improve the performance of the Tornado implementation on the NVIDIA Tesla K20m the tracking stage needs to be optimised.

One of the features of the tracking stage (as described in Section 7.3.2) is the use of the Iterative Closest Point algorithm across an image pyramid. In the experiments, the tracking algorithm performs an average of 13 iterations per frame. This means that each time the tracking stage is executed it will perform on average 3.75, 4.43,

and 4.8 iterations of ICP algorithm on the 80×60, 160×120 and 320×240 sized depth images respectively. Cumulatively, this equates to over 14MB of data that needs to be transferred from the device to the host for each frame that is processed by the pipeline. To hit a performance level of 30 FPS this would require 420 MB of data to be transferred each second. Conversely, only 64 bytes needs to be transferred from the host to the device per frame which means that the bottleneck is limited to data transfers off the device. From the performance model in Section 4.6 there are two ways in which performance can be optimised: (1) reducing the time taken to execute each device side kernel, and (2) reducing the time taken transferring data. Since the tracking stage is moving a large volume of data between device and host on each frame it makes sense to try and optimise according to (2) and minimise the volume of data being transferred. In the OpenCL version of Kinect Fusion, this problem is addressed by using a hand-crafted reduction operation which compresses the tracking result on the device first before transferring it back to the host. However, in the Tornado implementation this kernel was omitted as it limited the portability of Kinect Fusion and this is highlighted in Section 7.5 which explains why the OpenCL implementation struggled with portability. Nevertheless, as the tracking stage is a performance bottleneck two solutions are investigated: (1) a reduction function implemented purely in Java that does not use inter-thread communication (see Section 7.6.2), and (2) using Tornado to call a hand-crafted OpenCL C kernel (see Section 7.6.1). The advantage of the first approach is that the code will remain portable across all devices by sacrificing the ability to achieve maximum performance, whereas the second approach aims to provide the missing performance but sacrifice portability.

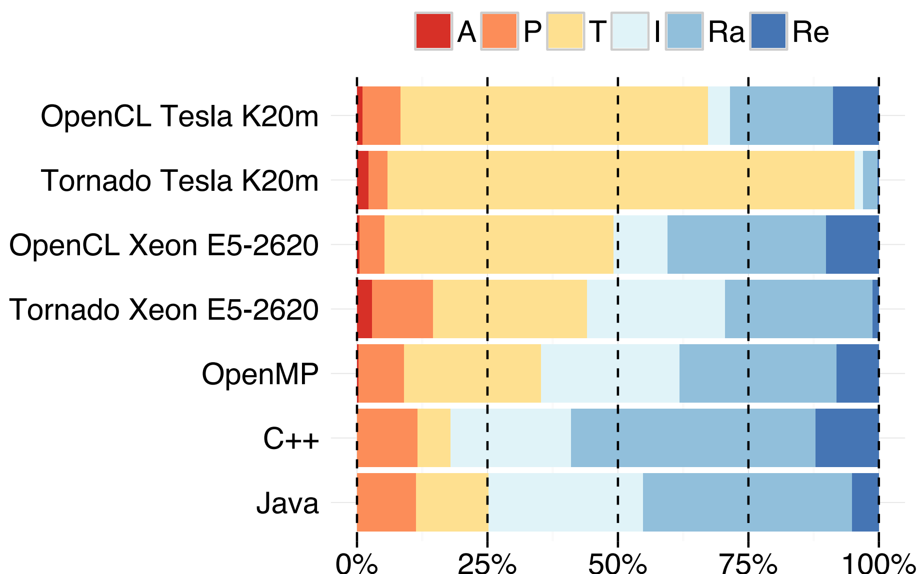


Figure 7.5: A breakdown of time spent in each stage of the Kinect Fusion pipeline. The important comparision is between the OpenCL and Tornado implementations as they execute on the same device. The other implementations serial C++, OpenMP, and serial Java are provided for reference only.

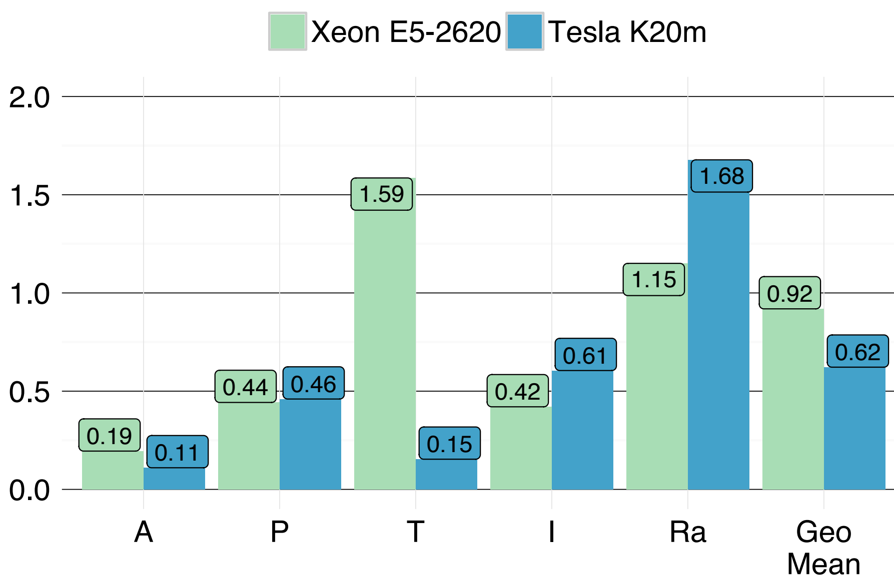


Figure 7.6: A breakdown of performance of the portable Tornado implementation of Kinect Fusion running on Server 1. The execution time of each of the five stages is normalised to the OpenCL implementation. Key: (A) Acquisition, (P) Pre-processing, (T) Tracking, (I) Integration and (Ra) Raycast

7.6.1 Implementing the Reduction Kernel in OpenCL

Section 7.6 has highlighted that on GPGPU systems the Tornado Kinect Fusion implementation spends the majority of its time in the tracking stage. In this stage, it is transferring 420 MB of data between the device and host each second. Therefore, to improve the performance of Kinect Fusion a reduction kernel is to be added to minimises the amount of data moved between the device and host. Typically, reduction kernels are implemented in two stages: (1) a global reduction phase and (2) a local phase. In such an implementation the first phase divides up the iteration space (or incoming data) equally among a fixed number of threads. This phase typically exploits as many threads as possible to process the input data into a smaller intermediate result that can be stored in local memory. Once complete, the second phase of the reduction takes the intermediate result and applies the same reduction function using successively fewer threads. Normally, this second phase is structured as a binary tree where the thread count is halved on each iteration until either a single value is obtained or a specific thread count is reached. The reduction kernel implemented in OpenCL follows this typical multi-phase implementation and its implementation is provided in Listing 7.9. Additionally, it is visualised in Figure 7.7 to show how work is distributed across compute units and threads. Notice that the number of threads used is not a function of the length of the input array. Instead, it is determined by the number of compute units that are to be used to process the array. This is due to reductions being memory bound operations that mean their highest performance is achieved by saturating the memory bandwidth of the device. Therefore, to improve performance dynamic reconfiguration will be used to adjust the number of compute units used in the reduction. The host side code to execute the reduction kernel is given in Listing 7.10. In this example, take note of how Tornado is being used to schedule an externally written OpenCL kernel. The expectation is that because the developer is able to utilise all OpenCL language features – such as `barrier` and `__local` memory – this kernel will be highly performant.

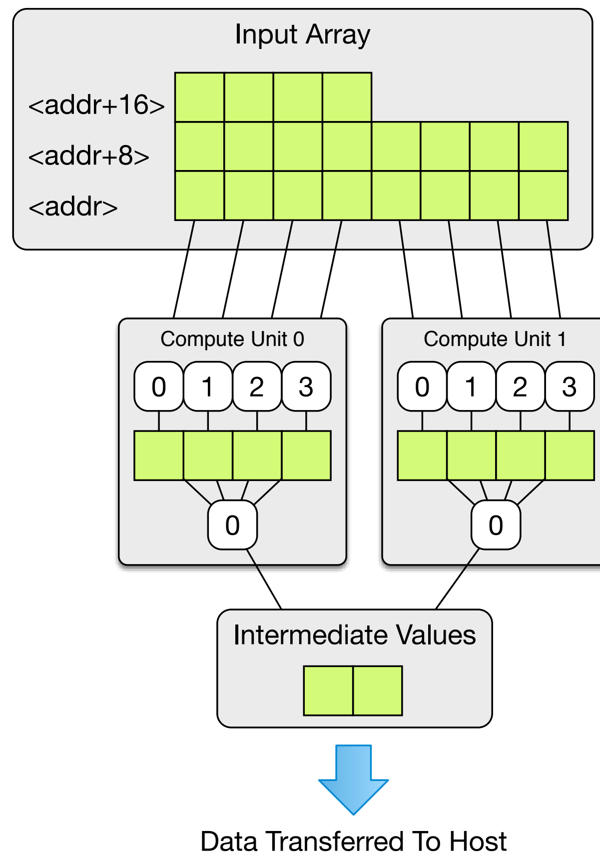


Figure 7.7: An input array is processed in parallel by multiple compute units to produce a partial result. Here each compute unit has four threads that each process at least two elements of the input array and produce a single result. A partial result of four values is stored inside each compute unit (one value per thread). Internally, each partial result is further reduced by the threads inside each compute unit to produce a single value. This value is written into global memory by the first thread – typically thread zero – to produce a final result of two values. It is these values that are transferred back to the host where they can be further reduced into a single value.

```
__kernel void reduce(__global void *_heap_base, ulong _stack_base) {
    // obtain arguments from the stack
    __global ulong *slots = ((__global ulong *)
            (((ulong) _heap_base) + _stack_base));
    ulong4 args = vload4(0,&slots[6]);

    const int sizeX = (int) args.s2;
    const int sizeY = (int) args.s3;

    // field access trackingResult.storage
    ulong a3 = args.y + 32;
    ulong a4 =  *((__global ulong *) a3);

    // define access to raw storage of data
    __global float* trackingResults = (__global float *)  ( a4 + 24);
    __global float* output = (__global float *) (args.x + 24) ;

    const int gtid = get_global_id(0);
    const int gdim = get_global_size(0);
    const int ltid = get_local_id(0);
    const int ldim =get_local_size(0);
    const int wgid = get_group_id(0);
    const int wgdim = get_num_groups(0);

    const int index = ltid * 32;

    __local float localOutput[WGS * 32];
    __local float *privateOutput = &localOutput[index];

    int i;
    const int numElements = sizeX * sizeY;

    // zero private output
    for(i=0;i<32;i+=4)
        vstore4((float4)(0),0,&privateOutput[i]);

    // reduce into private output
    for(int x=gtid;x<numElements;x+=gdim){
        const int index = x * 8;
        float8 result = vload8(0,&trackingResults[index]);
        reduceValue(result,privateOutput);
    }

    // copy into local output
    barrier(CLK_LOCAL_MEM_FENCE);
    const int wgindex = wgid * 32;
    if(ltid < 8){
        const int loIndex = ltid * 4;
        float4 sum = (float4)(0);
        for(i=0;i<ldim;i++){
            const int index = i * 32;
            sum += vload4(0,&localOutput[index + loIndex]);
        }

        // write to global memory
        vstore4(sum,0,&output[wgindex + loIndex]);
    }
}
```

Listing 7.9: OpenCL Reduction Kernel

```
final ImageFloat8 result = pyramidTrackingResults[i];
final int numElements = result.X() * result.Y();
final int numWgs = Math.min(roundToWgs(numElements / cus, 128), maxwgs);

final PrebuiltTask oclReduce = TaskUtils.createTask(
    "reduce",
    "./opencl/reduce.cl",
    new Object[]{icpResultIntermediate, result, result.X(), result.Y()},
    new Access[]{Access.WRITE, Access.READ, Access.READ, Access.READ},
    deviceMapping,
    new int[]{numWgs});

final OCLKernelConfig kernelConfig = OCLKernelConfig.create(oclReduce.meta());
kernelConfig.getGlobalWork()[0] = maxwgs;
kernelConfig.getLocalWork()[0] = maxBinsPerCU;

trackingPyramid[i].add(oclReduce)
    .streamOut(icpResultIntermediate);
```

Listing 7.10: The first three lines calculates how many work groups should be used for the reduction kernel. Next a task `oclReduce` is defined that uses a specified OpenCL kernel (from `./opencl/reduce.cl`). As the Tornado Runtime System is unable to inspect the OpenCL code the developer must define which parameters to pass to the kernel and how they are used. The latter allows the data movement between the host and kernel to be automatically scheduled and optimised by the the Tornado Runtime System. Notice how the tasks meta-data is used to create an `OCLKernelConfig` object. This is how developers can manually specify low-level OpenCL parameters to Tornado. Finally, the task is added into a task-schedule for execution.

7.6.2 Implementing the Reduction Kernel in Java

One of the limitations of the Tornado API is that it does not support all features of the OpenCL language. Good examples of missing functionality is the inability to perform inter-thread communication or access to low-level synchronisation primitives such as barriers. Therefore, implementing a reduction kernel in the same way as Listing 7.9 in Tornado is not currently possible. Instead, an attempt is made to write a parallel reduction kernel only using the language features provided by the Tornado API. The purpose of this implementation is to: (1) demonstrate that both a portable and parallel reduction kernel can be written using the Tornado API; and (2) to understand the performance gap introduced because the Tornado API does not support the necessary language features.

To create the Tornado reduction kernel a different approach is taken where the size of the intermediate result produced on the device is traded off against the number of threads that can be used to compute it. Typically, the smallest tracking result produced by the ICP algorithm is when the image resolution is of $80\times$ 60 at the top of the image pyramid (as seen earlier in Figure 7.3). At this level of the pyramid the ICP result has 4800 values, hence, the reduction kernel should aim to produce an intermediate result with less values. Like the OpenCL reduction kernel, the Java reduction kernel is implemented in two phases: one to perform a partial reduction on the device, and a second one on the host that reduces the intermediate result into a single value. However, the difference is that only a single reduction stage is run on the device. As a result the intermediate results will be larger than the OpenCL implementation but by keeping the result to less than 4800 values it should be possible to obtain a modest speed-up. The device side reduction is shown in Figure 7.8 and when run uses a fixed number of threads to reduce the size of the input array. This means that the size of the intermediate result is equal to the number of threads used. Note that by producing a value per thread means that no device-side intra-thread communication is required. However, it also means that the hardware cannot be fully utilised as represented by the dashed boxes. Although, this implementation is not expected to yield the highest level of performance it should perform better than doing no reduction at all and should remain portable across all devices.

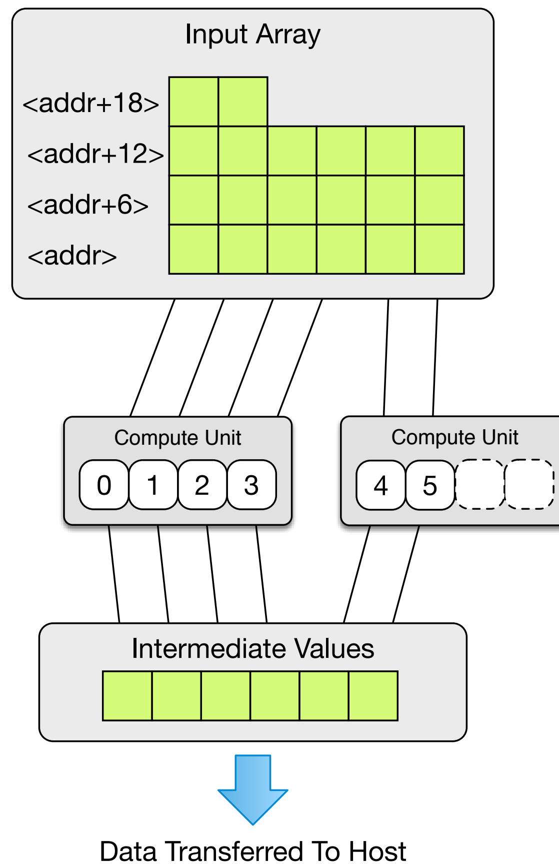


Figure 7.8: An input array is processed in parallel by a fixed number of threads across multiple compute units. Here each threads processes the input array in a thread-cyclic manner to produce a single result. This value is written into global memory by each thread and the intermediate result is transferred back to the host. Note: as a fixed number of threads are used the hardware will be under-utilised (as depicted by the dashed boxes).

```
public static void reduce(
    @Write final float[] output,
    @Read final ImageFloat8 input) {
    // calculate number of threads to use by the amount of
    // memory allocated for the intermediate result
    final int numThreads = output.length / 32;
    final int numElements = input.X() * input.Y();

    // start numThread threads
    for (@Parallel int i = 0; i < numThreads; i++) {

        // index into intermediate result
        final int startIndex = i * 32;
        for (int j = 0; j < 32; j++) {
            output[startIndex + j] = 0f;
        }

        // iterate over input array in a thread-cyclic manner
        for (int j = i; j < numElements; j += numThreads) {
            reduceValue(output, startIndex, input, j);
        }
    }
}
```

Listing 7.11: Tornado Reduction Kernel

```java
    // host side reduction of the intermediate result returned by the device
    public static void reduceIntermediate(
        final float[] output,
        final float[] input) {

      final int elementSize = 32;
      final int numDestElements = output.length / elementSize;
      final int numSrcElements = input.length / elementSize;

      // potentially parallelisable loop in the future
      for (@Parallel int i = 0; i < numDestElements; i++) {
          final int startIndex = i * elementSize;
          final float[] result = new float[elementSize];

          // copy first block of values
          for (int j = 0; j < elementSize; j++)
              result[j] = (i < numSrcElements) ? input[startIndex + j] : 0;

          // reduce the remainder
          for (int j = i + numDestElements;
                  j < numSrcElements;
                  j += numDestElements) {
              final int startElement = j * elementSize;
              for (int k = 0; k < elementSize; k++)
                  result[k] += input[startElement + k];
          }

          // copy out to main memory
          for (int j = 0; j < elementSize; j++)
              output[startIndex + j] = result[j];
          }
      }

   // allocates space for the intermediate result
   // (it is this that determines the number of threads used on the device)
   icpResultIntermediate = new float[config.getReductionSize() * 32];

   // adds the first stage reduction to a task-schedule and immediately streams
   // the result back to the host
   trackingPyramid[i]
       .add(IterativeClosestPoint::reduce, icpResultIntermediate, pTrackResult[i])
       .sync(icpResultIntermediate);
```

Listing 7.12: First, the method `reduceIntermediate` is required to perform the second phase of the reduction on the host. In the future, this method may also be accelerated by the Tornado API but for now it is executed serially on the JVM. Apart from the code the perform the second phase of reduction the developer only needs to allocate space for the intermediate result and insert the reduce kernel into a task-schedule.

7.6.3 Performance Improvements

To evaluate the impact of using specialised implementations of the tracking stage, all Kinect Fusion experiments were repeated under three different conditions: (1) no reduce kernel was used (NR), (2) the reduction kernel implemented using Tornado was used (TR), and (3) the reduction kernel written in OpenCL was used (OR). Despite the initial focus of the reduction optimisation being to improve performance on the NVIDIA Tesla K20m each implementation is evaluated on both multi-core processors and GPGPUs. In the experiments where the Tornado reduction was used the kernel was dynamically configured at runtime to use different numbers of threads: 512, 1024, 2048, and 4096 on the GPGPUs; and 0.5, 1, 2, and 4 $\times$ the number of available compute units on the CPUs. Whereas the experiments using the OpenCL reduction kernel, dynamic configuration is used to adjust the number and sizes of OpenCL work-groups. By default, a single work-group with the largest possible dimensions is assigned onto each compute unit. However, during experimentation it was found that this default setting suited the NVIDIA GPGPUs but not the Intel Iris Pro GPGPUs. Thus, after some experimentation it was discovered that the best performance was achieved on the Intel Iris Pro was when eight out of the 40 available (20%) compute units were utilised in the reduction.

As shown in Table 7.8 and fig. 7.9, by introducing a reduction stage the majority of Tornado implementations obtain significantly higher levels of performance. Exceptions to this are seen on the multi-core processors where a more modest improvement is seen – apart from on the Intel i7-4850HQ where the performance is flat. As the reduction optimisation is not targeted at these types of hardware accelerator then this is expected behaviour. Regarding the TR experiments, performance improved in 64% of cases (seven out of eleven), dropped in 27% of cases (three out of eleven) and remained the same in one. Out of the improvements a maximum speed-up of 74$\times$ the serial Java implementation was observed on the Intel Iris Pro P6300 with a frame rate of 52.76 FPS. More importantly, in three cases – on the Intel Iris Pro P6300, NVIDIA GTX550Ti and the NVIDIA Tesla K20m – the Kinect Fusion QoS threshold of 30 FPS was exceeded using this more portable reduction.

By using the OpenCL reduction a maximum speed-up of 150$\times$ the serial Java implementation was observed on the Tesla K20m with a frame rate of 125.30 FPS. However, in all cases where the OpenCL reduction produced a valid result the device-side performance was improved by between 1.16 and 2.85$\times$ over using the Tornado based-reduction. Note that as the OpenCL reduction was specialised for GPGPUs it

Type	Accelerator	OpenCL	Tornado		
			NR	TR	OR
Multi-core	AMD 10K-7850K	.	5.28	7.91	.
	Intel i7-2600K	30.11	17.44	20.13	23.47
	Intel i7-4850HQ	.	15.01	15.01	.
	Intel Xeon E3-1285	36.54	20.88		.
	Intel Xeon E5-2620	29.02	21.60	30.65	.
Many-core	Intel Xeon Phi 5110P	24.27	3.41	5.25	.
Embedded GPGPU	AMD Radeon R7	.	16.80	15.51	22.40
	Intel Iris Pro 5300	57.95	24.84	45.66	53.34
	Intel Iris Pro P6300	94.23	52.76	20.48	.
External GPGPU	NVIDIA GT 750M	.	20.15	21.00	25.64
	NVIDIA GTX 550Ti	4.43	40.77	37.66	65.40
	AMD Radeon HD 6970	135.34	11.05	.	.
	NVIDIA Tesla K20m	138.10	39.71	43.89	125.30

Table 7.8: Table shows measured performance of each implementation measured in Frames Per Second (FPS). Key: NR - no reduction kernel used, TR - Tornado reduction kernel used, OR - OpenCL reduction kernel used.

was less portable and so only ran on 46% of devices (six out of thirteen).

Comparing the specialised OpenCL reduction against the pure OpenCL implementation speedups between 0.78 and 14.76× are observed. In three out of four cases performance was between 0.78 and 0.92× the OpenCL. The final case, it was 14.77× which flatters Tornado, however, this result is treated as an anomaly and discussed further in Section 7.8.3. If these results are compared with OpenMP (Table 7.7) the observation is that although we started from a performance point of 3-7× lower than C++, the Tornado implementation is able to achieve higher performance on all CPU implementations. To obtain these results dynamic configuration was used to tune the OpenCL work-group sizes on each device. The results of these different experiments are shown in Figure 7.9. Moreover, the performance of the OpenCL reduction is compared against the performance of OpenCL in Figure 7.10 and the performance of the serial Java implementation in Figure 7.11. These two Figures show that: (1) using specialised implementations it is possible for Tornado to produce performance comparible with OpenCL, and (2) that using these specialised implementations a speed-up of between 17.63× and 149.55× over serial Java is possible.

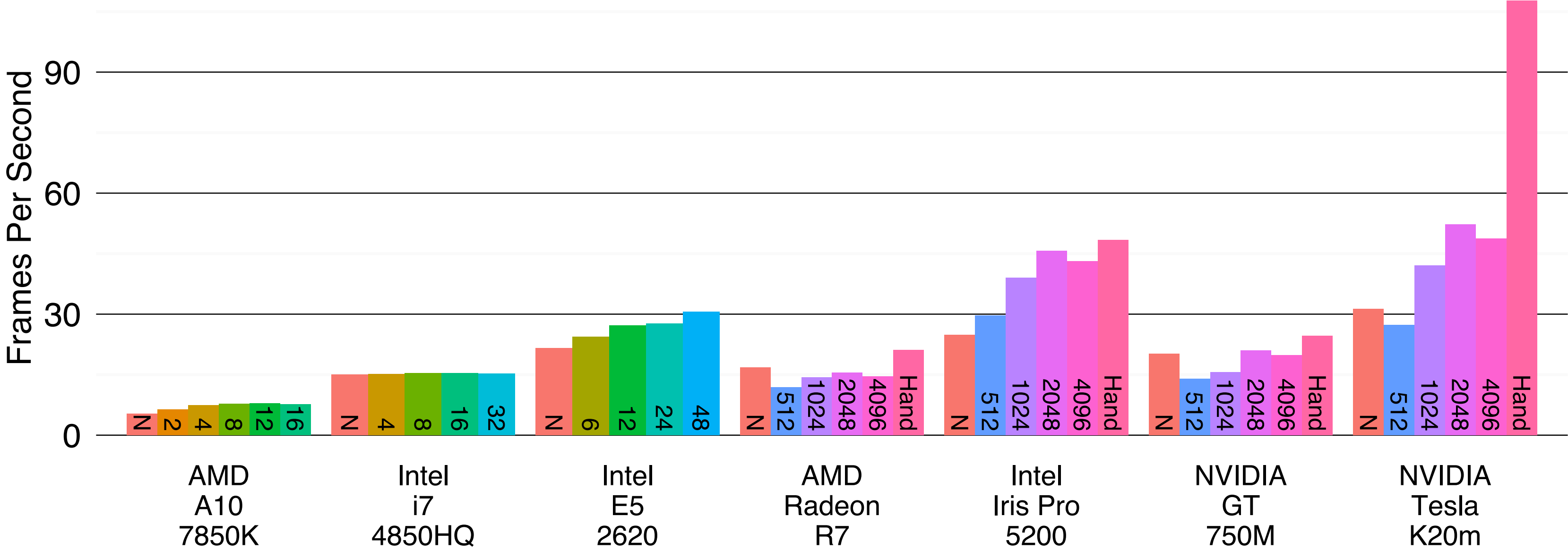


Figure 7.9: The above Figure shows the performance impact from using the two different reduction kernels: Tornado and OpenCL. To show how the Tornado implementation responds to changes in thread count multiple experiments are included. For brevity, only the best performing OpenCL experiment is included. Key: the reduction implemented using Tornado show the numbers of threads used in the experiment; and the OpenCL reduction kernel is marked as Hand (meaning that this was running a hand-written OpenCL kernel).

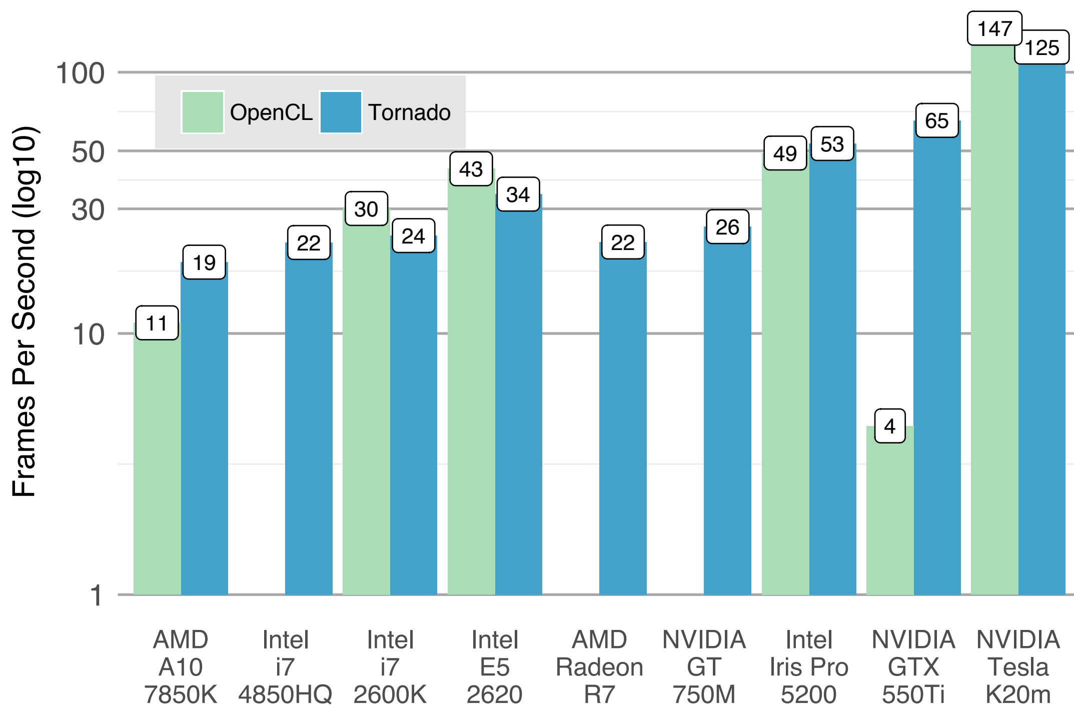


Figure 7.10: Above is a summary showing the performance in Frames Per Second (FPS) of the highest performing Tornado implementation of Kinect Fusion alongside the performance of the pure OpenCL implementation.

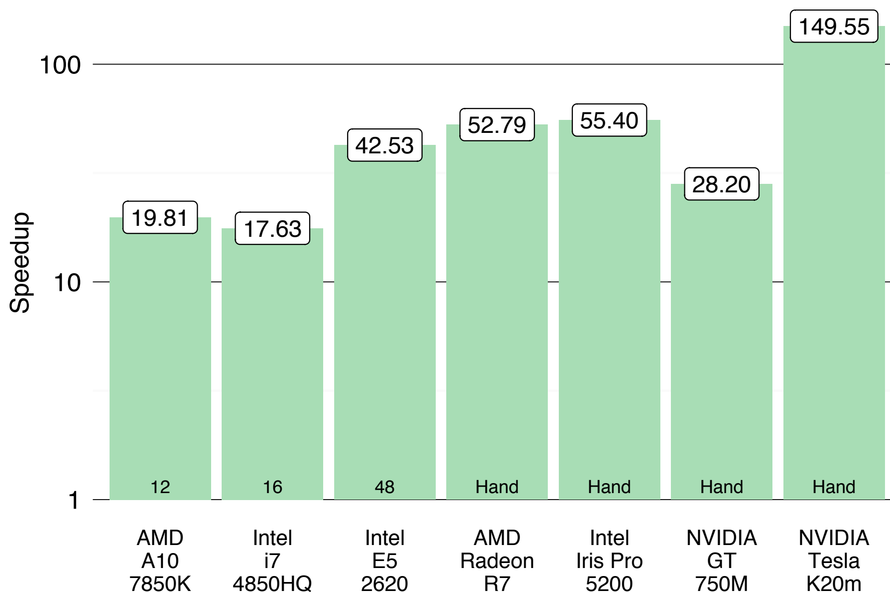


Figure 7.11: A summary showing the speed-ups Tornado obtained over the serial Java implementation. Note that the Tornado implementations reported are the ones that are using an optimised reduction kernel.

7.7 System Performance

The previous Section has highlighted that a specialised Tornado implementation can achieve similar performance levels similar to a pure OpenCL implementation of Kinect Fusion. Now as Tornado is built on top of the OpenCL software stack, any performance difference between the two is either as a result of: (1) differences in the compilation infrastructure – such as missing optimisations or semantic differences between C++ and Java; or (2) overhead introduced by the Tornado Runtime System – such as the additional cost of JIT compilation or the overheads of managing the execution of task-schedules. Hence, the role of this Section is to better understand the performance gap between the OpenCL and Tornado implementations of Kinect Fusion. Therefore, both the Tornado and OpenCL implementations of Kinect Fusion have been profiled across three different devices: the Intel Xeon E5-2620, the Intel Iris Pro 5200, and the NVDIA Tesla K20m.

The first comparison is to evaluate the end-to-end performance of Kinect Fusion and not just the average frame rate. To measure this the total execution time taken to process all 882 frames of the Kinect Fusion benchmark can be compared. Hence, the total execution time is inclusive of: (1) the time spent to execute kernels on the device, (2) data movement between devices, (3) the overhead of managing kernel execution, and (4) any host-side work that needs to be performed. And comparing these total execution times ensures that the different implementation are not being solely evaluated on the performance of individual kernels but rather as a whole system (i.e. compiler and runtime system). The results are shown in Figure 7.12 which compares the total execution times of the benchmark normalised to the pure OpenCL implementation. For clarity, these results are organised into three categories for comparison: (1) the minimal Tornado implementation that uses no reduction kernels – referred to as *base Java*; (2) the best performing Tornado implementation that is implemented entirely in Java but can include the use of the Tornado reduction kernel – referred to as *best Java only*; and (3) the highest performing Tornado implementation but can include the use of the OpenCL reduction kernel – referred to as *best*. From Figure 7.12 it is observable that across all three devices Tornado achieves a geometric mean of $0.59\times$ the performance of the OpenCL implementation using the most portable Java implementations, and if a specialised but less portable OpenCL reduction kernel is used this speed-up rises to $0.77\times$.

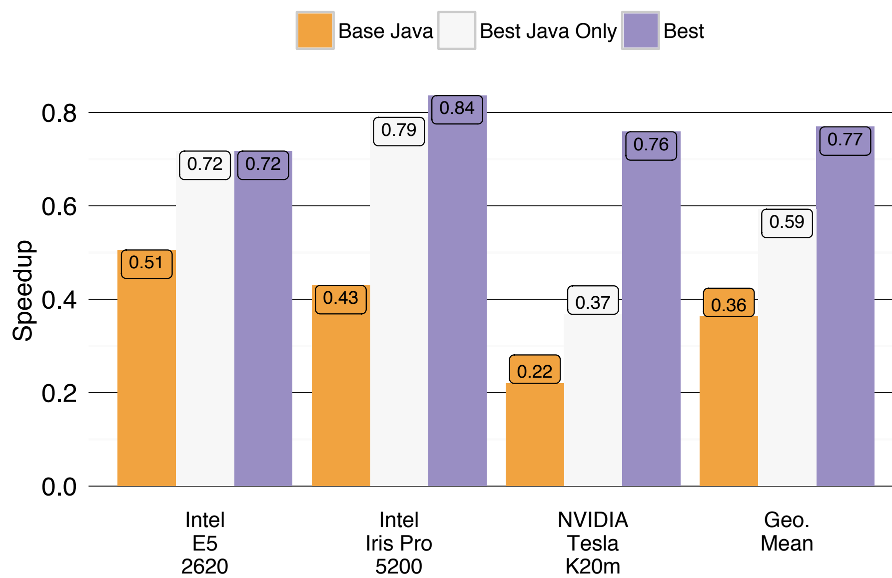


Figure 7.12: A summary of the performance obtained by Tornado normalised to the performance of the pure OpenCL implementation on the same accelerator. Key: *Best Java* – the minimal Tornado implementation that uses no reduction kernels; *Best Java Only* – the best performing Tornado implementation that is implemented entirely in Java but can include the use of the Tornado reduction kernel; and *Best* – the highest performing Tornado implementation but can include the use of the OpenCL reduction kernel.

7.7.1 Tornado Compiler Performance

One of the anomalies from Figure 7.12 is the low performance of Tornado relative to OpenCL on the NVIDIA Tesla K20m GPGPU. In the lowest performing implementation Tornado achieves 22% of the OpenCL and this only increases to 37% is the Tornado reduction kernel is used – a performance improvement that is significantly higher on the other two devices. What makes this interesting is that it is on this same device where the highest absolute performance is achieved for both Tornado and OpenCL – 125 and 147 FPS respectively. To try and understand why there is such a big difference in performance the performance of each individual device-side kernel has been profiled and is shown in Table 7.9. Here the raw execution times of each of the automatically generated Tornado kernels are compared against their OpenCL counterparts. Note that although these kernels perform the same work, they can differ in two ways: (1) Tornado kernels are derived from serial Java code that does not support all the language features of OpenCL – for example synchronisation primitives or built in vector intrinsics; and (2) both kernels may execute using different numbers of threads and work-group sizes.

Overall, the observation is that on average Tornado kernels achieve 98.8% of the performance of the hand-written OpenCL kernels. In 80% of these cases (eight out of ten), the hand-written OpenCL kernels perform better than the ones generated by Tornado with a speed-up between 0.55 and 0.98×. There are three contributing factors to these slowdowns: (1) the indirection added by object orientation – the Java implementations perform a number of field lookups to locate the starting point of arrays, (2) extra control-flow code is added by the Tornado compiler to handle irregular shaped arrays, and (3) the use of `constant` memory in the OpenCL kernels. The first two factors dramatically reduce the performance of kernels which have a low-arithmetic density and short execution times, e.g. `mm to meters` and `depth to vertex`. The third factor – the use of `constant` memory – is a simple optimisation that could be added into Tornado in the future to improve performance. In the cases where Tornado generated kernels execute faster than the hand-crafted OpenCL – `render volume` and `track` – the performance difference is down to optimisations in the GRAAL compiler. Typically, GRAAL performs aggressive in-lining and constant folding that allow it to remove a significant amount of redundant computation from these kernels. The reason why GRAAL is better in these situations than the OpenCL JIT compiler is because Tornado is designed to allow the propagation of constants from the host-side code into the device-side code (see Section 6.2).

Kernel	Invocations	Time (ms)	speed-up
raycast	879	1.635	0.978
track	12168	0.058	0.628
integrate	443	1.050	0.621
bilateral filter	882	0.144	1.459
vertex to normal	2646	0.024	0.870
depth to vertex	2646	0.020	0.554
render volume	221	0.192	9.180
half sample	1764	0.022	0.740
mm2meters	882	0.017	0.573
render track	221	0.036	0.847
Geo Mean	-	-	0.988

Table 7.9: Summarised invocation counts and average times for each Tornado compiled kernel running on the NVIDIA Tesla K20m. The speed-up is normalised to the performance of the same kernel running in the pure OpenCL implementation.

7.7.2 Tornado Runtime System Performance

Kinect Fusion is good application for comparing the ability of heterogeneous programming languages as it both tests the compiler and the runtime system. The compiler is tested because each device-side kernel needs to be performant but most importantly it stretches the runtime system because it requires the execution of between 540 and 1620 device-side kernels per second to be sustained over 60 seconds. To see how well the Tornado and OpenCL implementations achieve this Figure 7.13 shows the time taken to process each of the 882 frames. These results are taken from the NVIDIA Tesla K20m and show the differences between: (1) the Tornado implementation using the Tornado reduction kernel – referred to as *Java 2048* as it is a Java implementation that uses 2048 threads for the reduction; (2) the Tornado implementation that uses the OpenCL reduction – referred to as *Java Hand*; and (3) the pure OpenCL implementation. One of the key differences between the Tornado implementations and the OpenCL implementation is they experience extra overheads caused by the underlying Java Virtual Machine during the first 100 frames. These overheads are caused by two factors the JIT compilation of the application by the HotSpot compiler [96] and the garbage collector. After the first 100 frames, the performance of Tornado starts to stabilise and in the case of the implementation using the OpenCL reduction continues improving so that the performance starts to mirror that of the OpenCL implementation (albeit at a slightly lower level of performance). By profiling the memory usage of the JVM it was observed that the memory usage of the Tornado implementations stabilises around 400 MB which results in minimal interference by the garbage collector after the warmup period.

Generally, the Tornado implementations of Kinect Fusion avoid invoking the Tornado JIT compiler during the benchmark by explicitly triggering a compilation request before the benchmark begins. By triggering the JIT compiler early the cost of JIT compilation is removed from critical path of the benchmark and allows the Tornado implementations to provide a consistent level of performance. In Tornado the cost of compilation is between 100-200 milliseconds per kernel and this is shown in Figure 7.14 that compares the average compilation times for each kernel across all devices. In the case of Kinect Fusion the preference is not to compile each kernel on demand as the first frame to be processed will incur a penalty of one to two seconds while all ten kernels are compiled. Now as the Kinect Fusion pipeline needs to run in real-time this penalty could result in catastrophic failure of the application by dropping frames

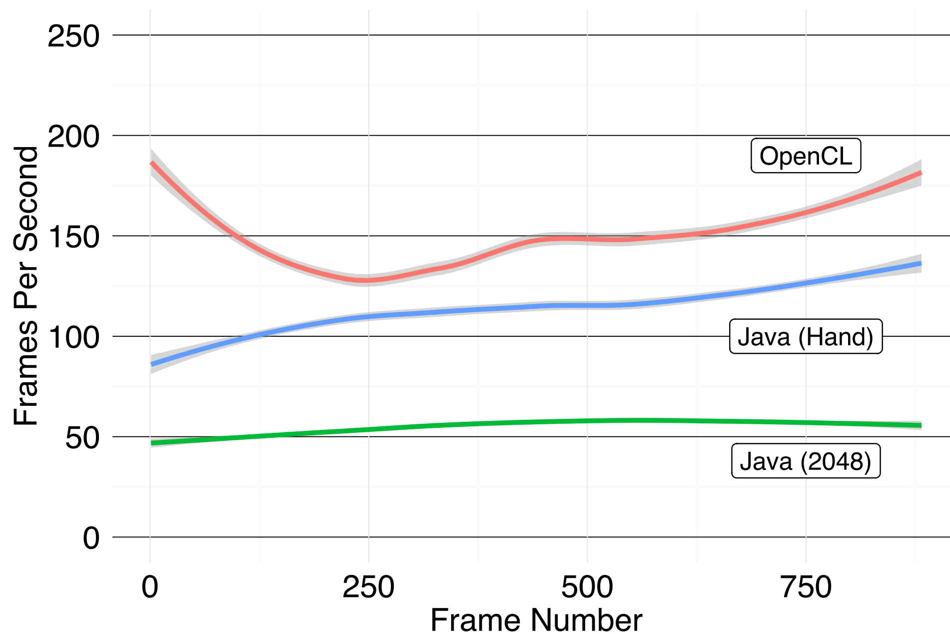


Figure 7.13: A plot showing how the frame rate varies on the NVIDIA Tesla K20m over the 882 frames of the ICL-NUIM trajectory 2 experiment. The red line is the pure OpenCL implementation. The blue and green lines represents the Tornado implementations. Green shows the reduction that is implemented in Tornado using 2048 threads and blue shows the OpenCL reduction.

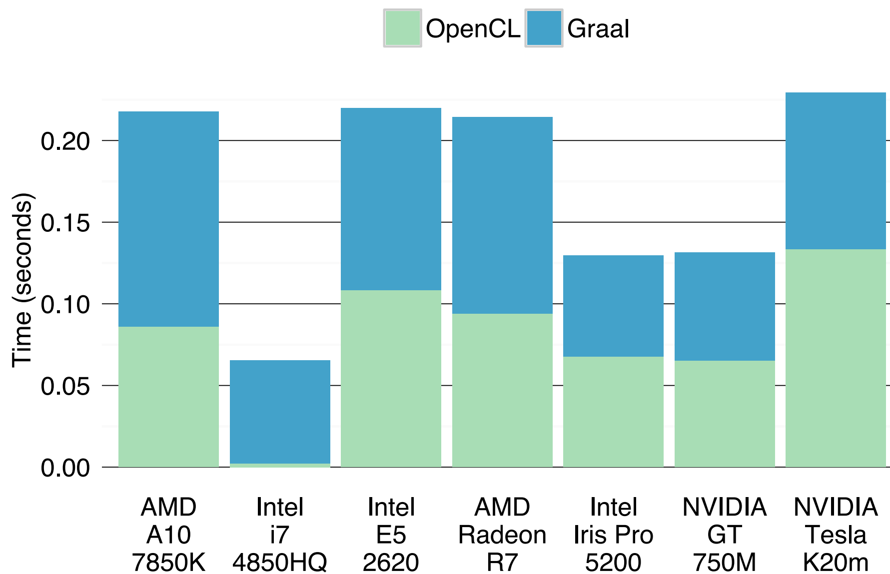


Figure 7.14: A breakdown of the compilation times for Kinect Fusion kernels inside the Tornado JIT compiler. Time is divided between the Tornado GRAAL compiler turning Java bytecode into OpenCL C and the OpenCL JIT compiler turning the OpenCL C into machine code.

or losing tracking. Hence, it is preferred in Kinect Fusion to always trigger compilation before starting the pipeline. In terms of the compilation times, they are made up from two almost even parts: (1) the compilation of Java bytecode into OpenCL C that is performed by the GRAAL compiler; and (2) the compilation of OpenCL C into machine code by the OpenCL JIT compilers. In the future it may be feasible to merge these two compilation tasks by having the compiler generate machine code directly – similar to how the compilation is performed in Jacc (described in Section 6.4.1) – and by doing this compilation times could be approximately halved.

7.8 Dynamic Configuration and Optimization

One of the most novel features of Tornado is its ability to configure and optimise an application dynamically. Section 3.3 describes how the Tornado programming API is designed to help the developer manage the uncertainty related to the execution environment of the application. It is this ability to adapt an application to its execution environment that differentiates Tornado from other heterogeneous programming frameworks. Since Tornado allows the user to quickly experiment with many different

configuration options it allows the use to experiment with a range of both intuitive and non-intuitive optimisations without the need to recompile the application. It is for this reason that users are able to extract a small amount of extra performance from Tornado application by tuning them to their execution environment. The remainder of this Section will be used to evaluate the impact of optimising Kinect Fusion through dynamic configuration and then go on to discuss some of the anomalies that are discovered.

7.8.1 Typical Optimization Process

When writing a Tornado application the process is to first aim to write a portable implementation that works across the majority of accelerators and then if extra performance is required to write a specialised implementation – as has been done with Kinect Fusion so far. However, once the application is running on the final system it can be further optimised using dynamic configuration. These final optimisations are aimed at further specialising the application to the exact hardware it is to execute on. Typically, it is desirable to be able to select which hardware accelerators the application should use and have the ability to specify some driver or device specific parameters. Good examples of these are the number of threads to use or to swap between using a synchronous or asynchronous API calls. In Tornado, an application can be configured in many different ways: programmatically by the application itself (as shown by the examples in Section 3.3.4) or on the command line (as shown in Listing 3.9 in Section 3.3). However, the Tornado version of Kinect Fusion has been modified to also allow the configuration to be loaded from external configuration files. Examples of these are shown in Listings 7.13 and 7.14 that contain the final configurations for a multi-core processor – the Intel i7-4850HQ – and a GPGPU – the Intel Iris Pro.

One of the key aspects of dynamic configuration is that this final stage of optimisation becomes a heuristic process. Currently, Tornado does not provide support for automatically exploring the optimisation space for each kernel. Instead this exploration is left to the user and their own intuition to find the best configuration for a specific system. What should be noted is that Tornado supports all the necessary features – profiling information and the dynamic re-compilation – for this heuristic search to be automated in the future; perhaps using some form of deep-learning!?!

In terms of the Kinect Fusion application the optimisation was limited to optimising the application to use a single hardware accelerator. Although Tornado make it possible to execute Kinect Fusion across disparate hardware accelerators the amount of data that needs to be moved between each device means that there is no realistic

chance of achieving the real-time constrains of the application. Thus, nearly all of the optimisations for Kinect Fusion are found by specialising the launch parameters of each OpenCL kernel. Typically an OpenCL kernel is launched using a global and local work-group size. In OpenCL the global work-group size determines how many threads are launched in total and the local work-group size determines how these threads are decomposed into small batches. The key to optimisation is being able to alter the number and dimensions of each local work-group as doing this alters how the kernel utilises the underlying processor architecture of the hardware accelerator. For example, in certain types of kernels the local work-group size correlates strongly to memory locality and by configuring it correctly it is possible to increase cache usage. Alternatively, in GPGPU systems the size of the local work-group determines the maximum number of work-groups that can be executing concurrently and on which processor the work-groups get assigned to. As a rule of thumb the dimensionality of both work-groups is determined by the dimensionality of the data being operated on by a kernel. For instance, a one-dimensional array is typically processed using a one-dimensional work-group and multi-dimensional arrays processed using multi-dimensional work-groups. In the case of Kinect Fusion nearly all of the code that runs on the hardware accelerator is processing an image (or two-dimensional array). This means that nearly all of the kernels use two-dimensional work-groups and often optimising the work-group sizes can give a small performance boost.

If the best performing Tornado implementations of Kinect Fusion are compared against OpenCL a few unexpected results occur and are shown in Figure 7.15. In three out of six cases, Tornado is between 0.79× and 0.85× slower than OpenCL – a level of performance degradation that is both expected and desirable. However, in the remaining three cases Tornado achieves a speed-up of between 1.09× on the Intel Iris Pro 5200 and 14.77× on the NVIDIA GTX 550Ti. Due to the variation across these results it is hard to quantify how close the performance gap between Tornado and OpenCL actually is. What is important is why this variation occurs and it is down to the fact Tornado applications can be dynamically configured. For instance, the 14.77× speed-up on the NVIDIA GTX 550Ti is not due to differences in the JIT compilers but because Tornado was able to configure the OpenCL driver differently. This anomalous result will be explained later in Section 7.8.3. The remainder of this Section will discuss the steps taken to specialise Tornado for three different accelerators: the NVIDIA Tesla K20m to produce the highest performing implementation at 120.30 frames per second; the result that achieves the 14.77× speed-up over OpenCL on the NVIDIA

```
# Intel i7 4850HQ
kfusion.tornado.platform=0
kfusion.tornado.device=1

# Use the Tornado reduction stage with 32 threads
kfusion.reduce.tornado_reduce=True
kfusion.reduce.num_threads=32

# Let OpenCL select the best work-group dimensions
tornado.opencl.schedule=True
```

Listing 7.13: Final configuration for the Intel i7-4850HQ multi-core processor. It is configured to use the Tornado reduction stage with and allows the OpenCL driver to determine the best work-group configurations.

```
# Iris Pro
kfusion.tornado.platform=0
kfusion.tornado.device=1

# Select the OpenCL reduction stage
kfusion.reduce.opencl_reduce=True

# Use 20% of available compute units
kfusion.reduce.fraction=0.2

# Manually specifiy work-group dimensions
tornado.opencl.schedule=False
tornado.opencl.gpu.block.x=128
tornado.opencl.gpu.block2d.x=16
tornado.opencl.gpu.block2d.y=4
```

Listing 7.14: Final configuration for the Intel Iris Pre embedded GPGPU. It has been configured to use the specialised OpenCL reduction stage and manually specified work-group sizes. Note that the OpenCL kernel used in the reduction stage is also dynamically configurable. However, in this case the default version was used and is specified to Tornado as part of the task defintion (see Listing 7.10).

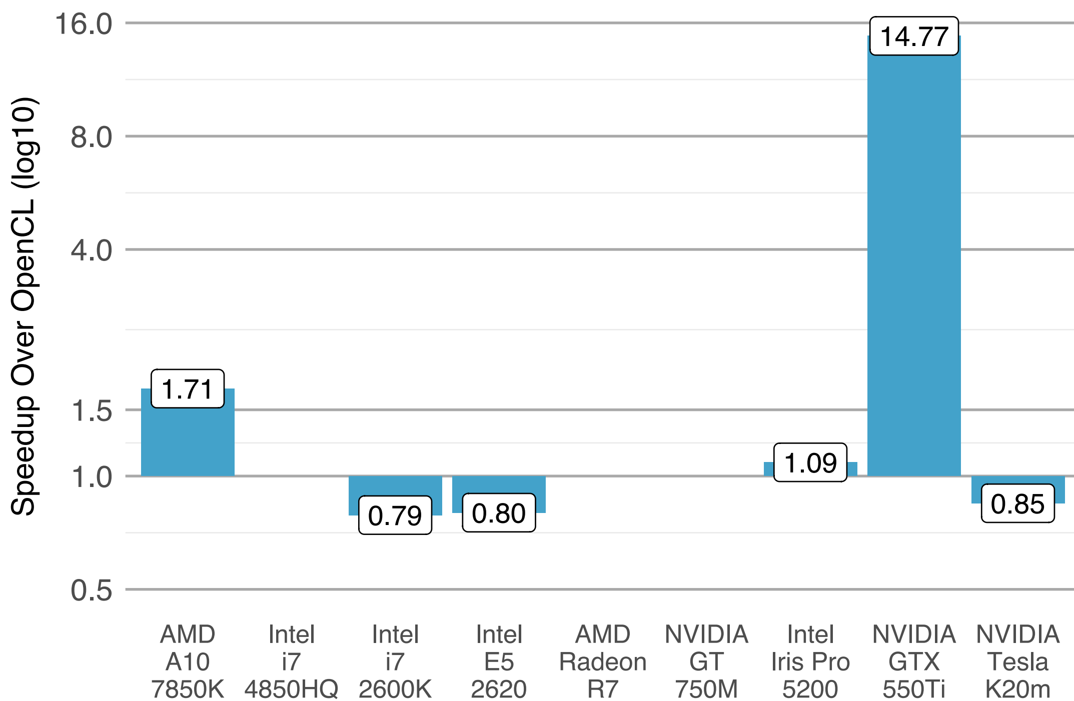


Figure 7.15: Final performance of the highest performing Tornado Kinect Fusion implementations on each device. Performance is normalised to the OpenCL implementation. Note the three cases where Tornado exceeds the performance of OpenCL. These results are often helped by using dynamic configuration to specialise the Tornado implementation for a specific system. Note: the missing results are where OpenCL was unable to produce a result on that accelerator.

GTX 550Ti; and the result that finds a 15% performance improvement on a multi-core processor – the Intel i7-2600K – by pretending that it is a GPGPU.

7.8.2 Obtaining The Highest Kinect Fusion Performance

Figure 7.16 shows the four steps that were taken to increase the performance of the Tornado Kinect Fusion implementation running on the NVIDIA Tesla K20m. Initially, the performance is improved by 52.88× simply by running Kinect Fusion on the GPGPU. At this stage, it was known though the profiling performed in Section 7.6 that data movement within the tracking stage was a bottle-neck; so the Tornado based reduction was tried. Using this reduction provided a performance improvement but did not significant close the performance gap with the OpenCL version. Next the OpenCL reduction stage was tried and this time the performance gap was closed significantly with the Tornado implementation achieving 109.49 FPS. However, as it was clear

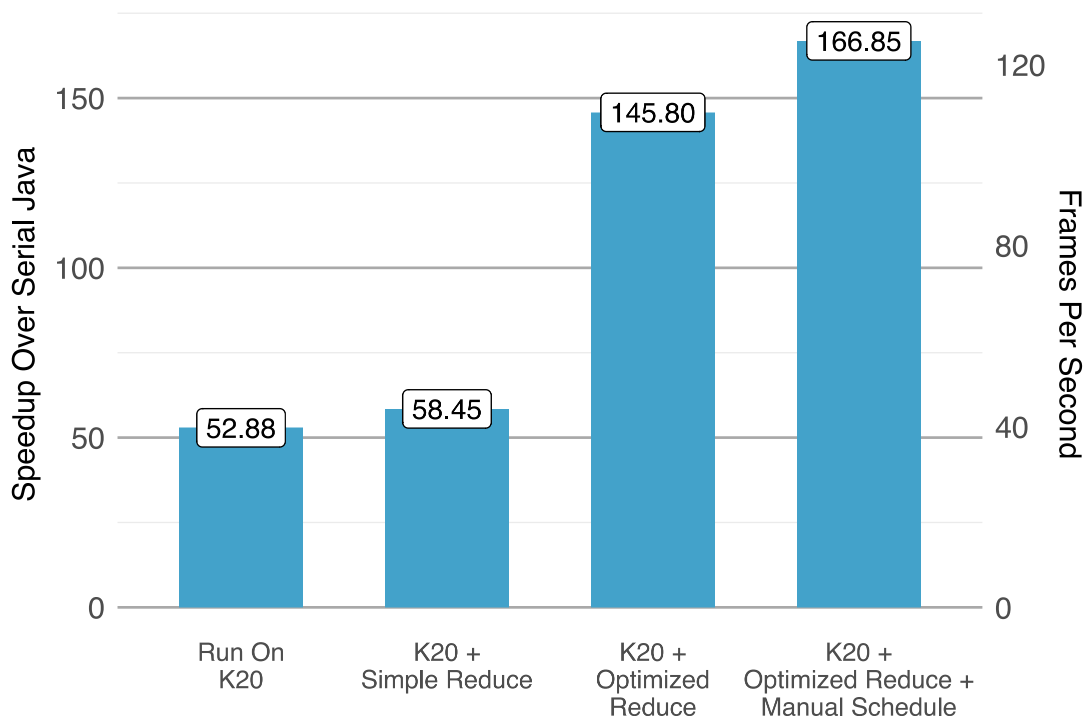


Figure 7.16: A summary of the steps taken to optimise Kinect Fusion on the NVIDIA Tesla K20m. One Kinect Fusion is running on the GPGPU it is a case of first selecting the highest performing tracking stage and then using dynamic configuration to tune the local-work group sizes. Key: Simple Reduce – uses the Tornado reduction kernel; Optimised Reduce – uses the OpenCL reduction kernel.

there was still some performance left to find a number of different local-work group sizes were tried using dynamic configuration. Through trial-and-error it was possible to find a local-work group configuration that provides a performance of 125.30 FPS or a 166.85$\times$ speeup over the serial Java implementation of Kinect Fusion. In this scenario the ability to dynamically configure the application provided a 14% increase in performance.

7.8.3 Blocking Versus Non-blocking OpenCL API Calls

One of the most obvious anomalies in Figure 7.15 is the result for the NVIDIA GTX 550Ti. This is an interest case because the steps taken to specialise Tornado on this GPGPU are very similar to how the NVIDIA Tesla K20 was optimised but with one exception: Tornado was instructed to use blocking OpenCL API calls instead of the non-blocking calls which are used by default. The differences are shown in Figure 7.17 which applies the same optimizations to the NVIDIA GTX 550Ti as was applied previously to the NVIDIA Tesla K20m but with Tornado using blocking and non-blocking OpenCL API calls. Ultimately, a speed-up of 48.21× the serial Java was obtained by using the blocking API calls and this becomes a prime example of why dynamic configuration is useful.

By default the Tornado Virtual Machine (TVM) uses non-blocking OpenCL API calls as it has been designed to exploit as much concurrency between the host and device as possible. Normally, this is beneficial as the TVM is able to queue up future kernel launches and data transfers ahead-of-time. This makes the use of blocking OpenCL API calls undesirable as it effectively serialises all operations within the TVM. Hence, this optimisation is very counter-intuitive and would never normally be considered in another heterogeneous programming language. However, as it is very easy to use dynamic configuration to explore all configuration options this optimisation was both discovered and used. The impact is that the Tornado implementation is able to out perform the OpenCL implementation by over an order of magnitude on this specific device. What should also be noted is that this GPGPU was one of the oldest that was used and this behaviour is likely to be due to a bug that has presumably been resolved in later versions of the CUDA driver. Hence, this result should be treated as an outlier when comparing Tornado with OpenCL.

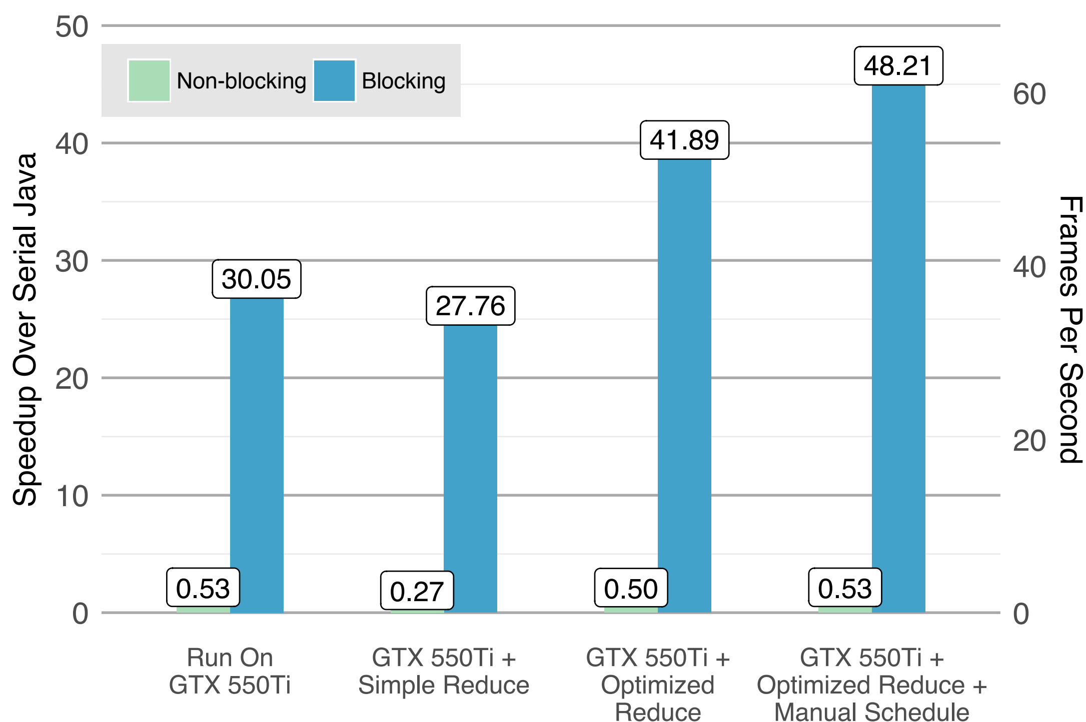


Figure 7.17: A summary of the steps taken to optimise Kinect Fusion on the NVIDIA GTX 550Ti. The optimisation steps and parameter values are the same as on the NVIDIA Tesla K20m except that the Tornado Virtual Machine is instructed to use blocking calls to the OpenCL API. Key: Simple Reduce – uses the Tornado reduction kernel; Optimised Reduce – uses the OpenCL reduction kernel.

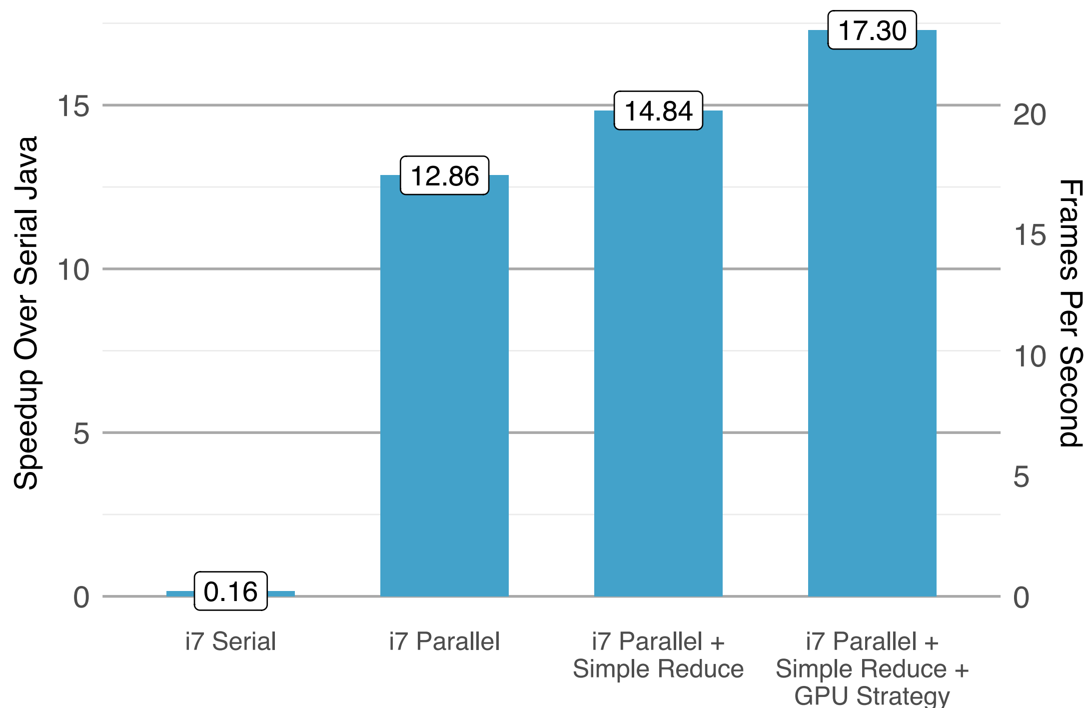


Figure 7.18: A summary of the steps taken to optimise Kinect Fusion on the Intel i7-2600K multi-core processor. The optimisation steps are the same as for the NVIDIA GPGPUs except that in the final step Tornado is instructed to treat the device as a GPGPU. Key: Simple Reduce – uses the Tornado reduction kernel; Optimised Reduce – uses the OpenCL reduction kernel; GPU Strategy – uses the thread-cyclic parallelisation scheme that is preferred for GPGPUs and is outlined in Listing 7.16.

7.8.4 Changing Parallelisation Schemes

As a final example of how dynamic configuration can be used to improve performance this Section will outline the steps taken to specialise Kinect Fusion for the Intel i7-2600K multi-core processor. The steps to specialise the application for the Intel i7-2600K are exactly the same as for the GPGPUs: (1) run on the accelerator, (2) select the best reduction strategy for the accelerator, and (3) fine-tune performance via dynamic reconfiguration. Figure 7.18 shows the changes in performance over each of these stages. For illustrative purposes only the performance of running Kinect Fusion serially in the accelerator is also shown. The difference this time is that dynamic configuration was used in the final step to improve performance by 17%.

By default Tornado parallelises each task according to the device that it is to execute on. In the case of multi-core processors a block-cyclic parallelisation scheme is used. An example of such a scheme is shown in Listing 7.15. However, in on this hardware

```
int id = get_global_id(0);
int block_size = (c.length + get_thread_size - 1) / get_thread_size(0);
int start = id * block_size;
int end = min(start + bs, c.length);
for (int i = start; i < end; i++) {
  c[i] = a[i] + b[i];
}
```

Listing 7.15: An example of a block-cyclic parallelisation scheme – this is typically used on multi-core accelerators that have fewer high performance processor cores. This code splits the iteration space into `block_size` continuous items that are processed on a single core. Note that if the number of elements being processes is not divisible by `block_size` the last core will be under-utilised.

```
for (int i = get_global_id(0); i < c.length; i += get_global_size(0)) {
  c[i] = a[i] + b[i];
}
```

Listing 7.16: An example of a thread-cyclic parallelisation scheme – this is typically used on many-core accelerators like GPGPUs that are capable of executing thousands of theads simultaneously. This code assigns one thread to process each element in the arrays `a`, `b` and `c`. If there are more elements than threads then each thread will process multiple elements with a stride size equal to the total number of threads being used.

accelerator it was discovered that using a thread-cyclic parallelisation scheme – as shown in Listing 7.16 – yielded better results. In the case of the Intel i7-2600K this optimisation lead to a 17% increase in performance taking Kinect Fusion from 14.84 to 17.30 frames per second. Again this optimisation was found by trial-and-error and is another example of a counter-intuitive optimisation that is easy to apply via dynamic configuration.

7.9 Summary

During this Chapter it has been demonstrated that a complex application – Kinect Fusion – written using Tornado can be seamlessly executed across thirteen unique hardware accelerators: five multi-core processors, a discrete many-core accelerator, three embedded GPGPUs and four discrete GPGPUs. What is more important is that to do so only required the application to be compiled once, unlike the OpenCL implementation of Kinect Fusion that often required re-compilation to handle variations due to differences in the operating system (Section 7.5.2) and source code modifications to handle variation between devices (Section 7.5.1). During the evaluation it has been shown that a serial Java implementation of Kinect Fusion running at 0.75 frames per second can be accelerated using a NVIDIA Tesla K20m GPGPU to achieve 125.30 frames per second – representing a 166.85× improvement in performance (see Figure 7.16). This result is important as it demonstrates that a Java application, via Tornado, can achieve the necessary levels of performance that enables new classes of computationally demanding applications to be written in Java. Here it is shown that a complex computer vision application can be written in Tornado and achieves a level of performance that is 4× greater than the minimum Quality-of-Service threshold of Kinect Fusion (30 FPS). However, to ensure that this result is not taken out of context this Chapter has also examined the performance gap between Tornado and the state-of-the-art heterogeneous programming language OpenCL. Consequently, these analyses shown that on the NVIDIA Tesla K20m a specialised Tornado implementation of Kinect Fusion is only 15% slower than OpenCL (see Figure 7.15).

To evaluate the quality of the Tornado system this Chapter draws a number of comparisons between the Tornado and other implementations of Kinect Fusion. In terms of accuracy Table 7.5 in Section 7.3.3 compares the Absolute Trajectory Error (ATE) of each of the different Kinect Fusion implementations and shows that both the Java and Tornado implementations of Kinect Fusion achieve significantly lower ATEs (0.0119 m) than the other implementations – C++ (0.0206 m), OpenMP (0.0206 m) and OpenCL (0.0207 m). In terms of code quality that the Tornado JIT produces Table 7.9 in Section 7.7.1 shows that Tornado achieves 98% of the performance (geometric mean) of OpenCL across ten kernels on the NVIDIA Tesla K20m. Moreover, Section 7.7 shows that the Tornado JIT compiler is able to compile these kernels in 100-200 milliseconds (see Figure 7.14) and that Tornado is able to sustain its performance over the full duration – 882 frames – of the SLAMBench benchmark (see

Figure 7.13).

One of the interesting tradeoffs that is examined in the Chapter is the difference between portable and specialised implementations. Initially, it starts by showing that it is possible to write a single implementation of Kinect Fusion using the Tornado API and have it execute across thirteen hardware accelerators. In contrast a state-of-the-art heterogeneous programming language, OpenCL, only manages to execute on nine different hardware accelerators (or 70% of the available hardware accelerators). A discussion on the reasons behind this lack of portability is provided in Section 7.5. Using this portable implementation a maximum speed-up of 55× the serial Java implementation was observed on the NVIDIA Tesla K20m. Now if the developer wishes to sacrifice portability, Tornado can be used to create a specialised implementation of Kinect Fusion. Section 7.6 describes how an extra reduction stage was added to the Kinect Fusion pipeline to improve performance. This stage consisted of two specialised tasks: (1) a simple reduction written entirely in Tornado that works well on multi-core processors, and (2) a hand-written OpenCL kernel specialised for GPGPUs. By using these specialised implementations of Kinect Fusion the maximum performance of Kinect Fusion increased to 166.85× over the serial Java implementation on the NVIDIA Tesla K20m GPGPU. Moreover, it was also observed in Section 7.6.3 that for these specialised implementations Tornado achieved at worst a 21% slower than OpenCL and at best 14.77× faster than OpenCL (see Figure 7.15). If this performance is compared against the original Java implementation of Kinect Fusion then Tornado achieves a speed-up between 17.64× and 149.55× (see Figure 7.11).

Finally, the ability to use dynamic configuration as a final optimisation step was evaluated. In Section 7.8 three use cases where dynamic configuration allowed a Tornado application to be further specialised for a specific device but without the developer having to make an code modifications. In the first case (Section 7.8.2), it was demonstrated that having the ability to tune the OpenCL work-group dimensions yielded a speed-up of 1.14×; a seemingly intuitive optimisation. However, in the next two cases counter-intuitive configurations were used to provide speed-ups of 1.17× and 14.77×. The first was obtained by instructing Tornado to treat a multi-core processor as a GPGPU (see Section 7.8.4), and the second when Tornado was instructed to use blocking calls to the OpenCL API (see Section 7.8.3).

8 | Conclusion

The aim of this thesis is to ascertain whether it is possible to create a heterogeneous programming language without the need to make any closed-world assumptions about either the number or type of components contained within a heterogeneous system.

This aim has been met by first designing and implementing a novel heterogeneous programming framework called Tornado – described in Chapters 3 to 6. After implementing Tornado, Chapter 7 describes how a complex real-world application, called Kinect Fusion, was written once using Tornado and executed across thirteen different hardware accelerators: five multi-core processors, a discrete many-core accelerator, three embedded GPGPUs and four discrete GPGPUs. In the next three Sections these claims will be broken down further and discussed. Next the limitations of this thesis will be discussed in Section 8.4. Finally, this Chapter will conclude with some final remarks and ideas for future work in Section 8.5.

8.1 Tornado Design

Tornado has three key components that have been co-designed that enable the creation of code that is agnostic to both the number and type of hardware accelerators contained within a heterogeneous system. The first component is the Tornado API that is designed to decouple the application code that decides where code should execute – the co-ordination logic of the application – away from the code that defines the computation – the computation logic of the application. It is described in Chapter 3 along with a description of its task based programming model. In Tornado the key abstraction is a task – an abstraction that is similar to a continuation in that it represents the invocation of a particular method with specific parameters. The second component, the Tornado Virtual Machine is introduced in Chapter 4 that provides a virtualisation layer between the Tornado API and the underlying architecture of the heterogeneous system. Like the Java Virtual Machine, the Tornado Virtual Machine is designed to provide a target

against which an application can be programmed. The difference between the Java Virtual Machine and the Tornado Virtual Machine is that it is only a subset of an application – an applications coordination logic – that targets the Tornado Virtual Machine. In the Tornado API the task-based programming model is used to generate Tornado bytecode that is fed into the Tornado Virtual Machine where it is used to dispatch kernels on hardware accelerators. The missing link between the Tornado API and the Tornado Virtual Machine, and the third and final component, is the Tornado Runtime System (described in Chapter 5). The Tornado Runtime System can be thought of as a dynamic optimising compiler that converts the Tornado API into the Tornado bytecode consumed by the Tornado Virtual Machine. Together these three components – the Tornado API, the Tornado Virtual Machine, and the Tornado Runtime System – have allowed a complex real-world application to be created and specialised for a range of different hardware accelerators. Chapter 7 describes how a Tornado application can be compiled once and executed across thirteen different accelerators: five multi-core processors, a discrete many-core accelerator, three embedded GPGPUs and four discrete GPGPUs.

8.2 Distinguishing Features of Tornado

As a direct result of combining these three three key components together Tornado has a number of distinguishing features. One of these features is the transparent optimisation of the co-ordination logic by the Tornado Runtime System. By providing a programming abstraction, the task-schedule, developers are able to express the relationships between tasks and data during the computationally critical parts of an application. Internally, this task-schedule is converted into a graph that tracks both control and data flow between kernels. The Tornado Runtime System is then able to optimise this graph to eliminate redundant data movement and re-order the execution of tasks to exploit any concurrency between them. A clear example of the ability of Tornado to optimise complex task-schedules is given in Section 5.3. Additionally, Section 3.3.4 provides examples of how Tornado applications can be written so that both tasks and task-schedules can migrate across different hardware accelerators. Moreover, Listing 3.12 demonstrates a situation where each stage of a multi-stage processing pipeline can be executed on a randomly selected hardware accelerator each time the pipeline is invoked. Unlike languages such as OpenACC and OpenCL, this kind of code is possible in of Tornado because parallelisation happens in the JIT compiler. Hence,

switching between parallelisation schemes is a matter of re-running the JIT compiler with a different option and it is this feature of Tornado that enables its applications to become portable across many devices. This functionality has become so important to Tornado that it is also exposed to the developer in the form of dynamic configuration. Thereby, allowing the user to dynamically change how either the Tornado Runtime System or the Tornado JIT compiler behaves. This can be done either programmatically as shown in the migrating task examples mentioned earlier or on the command line which requires no code modifications. Having this ability allows developers to easily generate and specialise applications for different hardware accelerators and in doing so has an impact on the typical development cycle. Section 7.8.1 discusses the typical optimisation process of a Tornado application: (1) a portable application is written, (2) any poorly performing code is specialised for a specific type of device (using hand-written OpenCL if necessary), and (3) the application is specialised for a specific device using dynamic configuration. Here the first two steps require a developer to write code and are normally responsible for the bulk of the performance improvements. However, Tornado enables the third step to be performed without modifying any code. In one scenario it was possible to obtain a $14.77\times$ speed-up over OpenCL by changing a Tornado configuration flag (see Section 7.8.3). Although the this scenario is a little extreme, in the other two scenarios performance was improved by between 14 and 17% (see Sections 7.8.2 and 7.8.4 respectively).

8.3 Evaluation of Tornado

One of the key questions that needs to be answered is whether it is feasible to implement a heterogeneous programming language that avoids making any form of closed-world assumption about the system architecture it is to execute on. To answer this question Chapter 7 evaluates the ability of Tornado to accelerate a complex real-world application call Kinect Fusion. The evaluation starts by examining the portability of the Tornado implementation of Kinect Fusion against the state-of-the-art heterogeneous programming language OpenCL. In this comparison the Tornado implementation could be compiled once and executed across each of the thirteen devices that were tested, in contrast OpenCL only managed to execute on nine out of thirteen devices (see Section 7.5). The reason why OpenCL struggled with portability is that the application was over specialised and so struggled to execute of lower-end devices (see Section 7.5.1). Tornado avoids this problem by created a dynamic configuration based

on the properties of each device; a clear advantage of using a fully dynamic compilation process. In this first set of comparisons Tornado is able to achieve a maximum speed-up of 55× over the serial Java implementation on the NVIDIA Tesla K20m. However, it struggles in comparison to OpenCL where it only achieves approximately 28% of its performance.

As there is such a big performance gap between Tornado and the state-of-the-art, represented by OpenCL in the evaluation the follow on question is to ask whether this performance gap can be bridged? Section 7.6 describes two different ways in which Tornado applications can be specialised: (1) to write task specialised for a specific type of device using the Tornado API, or (2) to have a task launch a hand-written OpenCL C kernel. Using these specialisations improved the performance of the Tornado implementation of Kinect Fusion to provide a worst case performance loss of 21% and a best case performance increase of 14.77× (see Figure 7.15). The most important result is obtained on the NVIDIA Tesla K20m where both the Tornado and OpenCL implementations recorded their highest performances of 125.30 and 138.10 frames per second respectively. On this device the evaluation shows that the specialised Tornado implementation of Kinect Fusion suffers an overall performance loss of 15%. By comparing kernel-to-kernel execution times between Tornado and OpenCL it is observed that the Tornado device-side kernels are on average 1.2% slower than their OpenCL counterparts (see Table 7.9). Hence, the main source of performance loss in the Tornado application is not due to differences in kernel execution times – it is likely that much of this performance loss is an accumulation of differences introduced by having a complex runtime system.

In Chapter 7 the evidence shows that it was possible to write Kinect Fusion using Tornado and achieve a computationally demanding Qos threshold. In fact, it was possible on three GPGPUs – the Intel Iris Pro P6400, the NVIDIA GTX 550Ti, and NVIDIA Tesla K20m – to achieve the 30 frames per second threshold without requiring the application to be specialised (see Table 7.7). However, if specialisation is used this number rises to five where the maximum performance exceeds the QoS threshold by over 4× – 125.30 FPS on the NVIDIA Tesla K20m (see Table 7.8). The ultimate outcome is that Tornado is able to accelerate the performance of an application written in serial Java running at 0.71 frames per second to over 125.30 – representing a 166.80× speed-up. This is important for as it shows that a heterogeneous programming language like Tornado can be integrated with a popular programming language like Java to make it possible for developers to write new classes of application that

have much higher computation requirements than before.

8.4 Limitations

8.4.1 Evaluation

One of the obvious limitations of this thesis is that Tornado is only evaluated on a single real-world application. The reason for using such an application was two fold: (1) no heterogeneous benchmarks yet exist for the Java language, and (2) using a complex real-world application like Kinect Fusion would lead to a more robust evaluation as it would stress every part of the Tornado system – from the API to the compiler. It is important to note that to meet the Quality-of-Service requirements of Kinect Fusion application Tornado needs to sustain the execution of between 540 and 1620 tasks per second over 882 frames. To enable Tornado to achieve this time was spent optimising the internals of the Tornado Runtime System to avoid introducing unnecessary overheads – both in terms of memory and compute. Section 7.7 evaluates the overheads introduced by Tornado. Another aspect to this is that the complexity of the Kinect Fusion application required the use of an industrial compiler; this was the primary reason of abandoning Jacc in favour of Tornado (see the discussion in Section 6.4.6). Naturally, given time it would be possible to extend the evaluation to include other applications and frameworks. However, to really allay fears Tornado has been open-sourced (`https://github.com/beehive-lab/Tornado`) to make it is possible both verify these results and write your own applications.

8.4.2 Operating System and Hardware Diversity

A big issue during the evaluation of Tornado is the relative lack of diversity both among the operating systems and the systems used. Consequently, all experiments were executed on an Intel based system running either linux or OSX. One of the reasons for this is a lack of available hardware: throughout the development of Tornado it was envisaged that the final evaluation would include an ARM based system with an embedded GPGPU. However, this did not come to pass as at the time of writing, it was not possible to procure a system that is both capable of running an OpenJDK compatible with GRAAL or that contains an OpenCL programmable accelerator. It is very likely that such a system will become commercially available in the very near future and there is no fundamental reason why Tornado would not work in this context. For completeness,

it would also have been nice to have run Tornado on the Windows operating system. However, both of these issues – hardware platform and operating system – were not deemed vital to show as the key principle that has to be proven is that Tornado is able handle variation among the different types of hardware accelerator it encounters.

One of my most late-breaking experiments with Tornado has been to target OpenCL programmable FPGA systems. It has already been possible for Tornado target OpenCL programmable FPGAs with some single kernel benchmarks – such as SGEMM. However, there are some restrictions surrounding the usage of FPGAs that prevent Tornado running the Kinect Fusion application. The primary problem is that FPGAs require you to compile all device-side kernels ahead-of-time into a single monolithic kernel. A constraint that directly goes against the dynamic principles of Tornado and would require a little thought to work around.

8.4.3 Use of Multiple Accelerators

An obvious question that is not truly answered in this thesis is whether Tornado is able to utilise multiple hardware accelerators? The answer to this is that Tornado has been designed with this in mind and there are examples in Section 3.3.4 of tested code that migrates both tasks and task-schedules across different hardware accelerators. Moreover, Listing 3.12 in the same Section provides an example of how each stage of a multi-stage processing pipeline can be executed on a randomly selected hardware accelerator each time the pipeline is invoked. This latter example is code that would be difficult or impossible to write in languages such as OpenACC and OpenCL. As Kinect Fusion transfers a large amount of data between host and device it did not make sense to perform experiments where the each stage of the pipeline was mapped onto a different accelerator as the performance would be very poor. If a suitable application was found then this feature of Tornado could be easily evaluated.

There is a second way in which Tornado could use multiple accelerators and that would be to split the processing of a single task across multiple accelerators. This mode of computation is deliberately missing from Tornado as this requires the support of more complex parallelisation schemes inside the Tornado JIT compiler. However, the intention to solve this problem within Tornado would be to add extra functionality inside the Tornado Runtime System to tackle this issue at the task-schedule level. For example, it should be possible to add extra features, such as a loop, to task-schedules that make it possible to create co-operative tasks that span multiple hardware accelerators.

8.4.4 Limited Support For The Java Language

A criticism that is quite natural about this thesis is that Tornado supports only a subset of the Java language (and it cannot be useful until it supports all of it). There are a few views on this subject. First, that in this thesis only a subset of the Java language has been used (or needed to be used) to accelerate an application by 166.80×. Thus, demonstrating that supporting the whole of the Java language is not required to make use of a hardware accelerator. In this thesis the point has been to demonstrate that a programming language like Tornado is feasible and not to demonstrate that the whole of an existing language can be supported. Moreover, it should also be noted that the reason that Tornado is able to accelerate applications is because it uses specialised hardware, which by definition is unlikely to work well with every single application. Hence, the key with Tornado (and every other programming language) is in matching the right application to the right device. This being said it would make an interesting study in the future to understand the true extent of how much of Java Tornado can support. Section 6.3 provides a discussion on the current limitations of Tornado and suggests how they might be resolved.

8.5 Final Remarks

This thesis has demonstrated that it is possible to avoid making closed-world assumptions about the number or types of hardware contained within a heterogeneous system. And in doing so have created what is believed to be the first truly dynamic programming framework for modern heterogeneous architectures – Tornado. Moreover, by using Tornado it has been demonstrated, in Section 7.6.3, that by using hardware acceleration: (1) it is possible improve the performance of Java applications by between 17.64× and 149.55×, and (2) that this performance comes within 0.78 to 0.92× of a pure OpenCL implementation. Despite being a very good headline result the contents of this thesis need to be interpreted in the right context: that Tornado is an academic prototype. The key result is that the specialised Tornado implementation of Kinect Fusion is in the worst case 22% slower than the OpenCL equivalent. This is the important result because it demonstrates the viability of an approach like Tornado and that it may be possible to achieve performance parity with OpenCL in the future. Moreover, it should also be highlighted that as Tornado and to some extent GRAAL are still in their infancy their performance will continue to improve over time. Hence, it is more likely

that their performance will improve faster than than OpenCL at this point. Finally, this thesis shows that in the future developers do not need to be locked into using low-level languages to exploit hardware acceleration. And in doing so making a positive contribution to addressing the challenges set out in Section 1.1.

8.5.1 Future Work

To continue this line of research there are two lines of questioning that might be followed. Either extending Tornado to add more features or to develop a new heterogeneous programming language.

8.5.2 Extending Tornado

A natural outcome of this research is: "Can Tornado be applied to other dynamic languages?" Tornado has been designed from the ground up to work at the Java Bytecode level and not the Java language. Consequently, it is possible to utilise Tornado from other JVM based languages. For instance, a initial attempt at targeting Scala has been made. Furthermore, GRAAL has a sister project – Truffle – that enables many languages – such as C, JavaScript, Python, R, and Ruby – to execute on top of the Java Virtual Machine. Hence, it would be interesting to integrate Tornado with Truffle to allow all these languages to benefit from hardware acceleration.

Integrating Tornado with these other dynamic languages would be interesting. Typically, most complex language features get stripped away during compilation and do not cause dramatic performance loss. Even object-orientated features, like inheritance and polymorphism, can be used in Tornado. Remember that when a task is JIT compiled the Tornado compiler is able to use reflection to lookup the exact type of each object. This means that a lot of indirection caused by these features is optimised away through a combination of inlining and constant propagation inside the GRAAL compiler. As Tornado operates on Java bytecode and not the languages itself, there are very few bytecodes that Tornado cannot support. For example, it should be possible to support reflection and making system calls on a hardware accelerator; however, the real question here is why? (As many of these features should not be used inside the critical path of the application.) Perhaps, two of the weaknesses of Tornado are that it relies on explicitly knowing the sizes of iteration space and plays strongly on having mutable data. Whereas in Scala it is common to represent data structures recursively and as immutable objects. Hence, it would be interesting to see how Tornado could be

adapted to solve these issues.

Naturally, there are always going to be ways to write code that is bad for every architecture and it should be unsurprising to find that not all code written in Tornado will achieve a two orders-of-magnitude speed-up. A good example of code that would work in Tornado but perform poorly is operating on a linked list of objects on a GPGPU. The best way to handle this kind of situation would be to improve the diagnostics produced by Tornado to output warnings to the developer. For instance, Tornado can easily determine if a loop has been wrongly identified as parallel by the developer and vice versa. Additionally, it is also easy to generate messages when undesirable languages features exist with a task – such as memory allocations, try/catch blocks or irregular control-flow.

8.5.3 Development of a Programming Language

Throughout the development of Tornado, there has been a balance between what is expressible in Java and what is not. This inability to express the intent of Tornado is most apparent in the design of the Tornado API where two notations for expressing task-schedules has been used. It would be beneficial if task-schedules became first-class citizens of a programming language. This way a task-schedule would not be artificially restricted to being a sequential list of API calls; instead, developers would be free to make method calls and use control-flow like loops and if statements. Allowing these features inside task-schedules would make it easier to construct, and optimise even more complex processing pipelines. Moreover, as Tornado is implemented using Java it has had to cope with the idiosyncrasies that this entails: Java generics, garbage collection, and missing primitive types. Although, these features can be supported in Tornado they cause problems for the application developer and sometimes deter them from using a particular language. An example of this that the porting of Kinect Fusion was sometimes complicated because of of the lack of an unsigned integer type in Java. An interesting experiment in the future might be to investigate how a new programming language could be constructed to take better advantage of the features afforded by Tornado.

Tornado Terminology

Coordination Logic The code in an application that describes where and when a computation should be performed.

Host-side Refers to any code or data that is present or executed on the host-device – normally an x86 processor.

Device-side Refers to any code or data that is present or executed on a device like a GPGPU.

Task A unit of work that has to be scheduled by the coordination logic.

High-level Task An API abstraction that encapsulates all aspects of executing a computation on a device. i .e. a method call that executes on a device.

Low-level Task Are concrete instances of work that are scheduled on devices. e .g. data transfers and kernel execution. A high-level task is implemented as a series of low-level tasks.

TVM Bytecode Is a series of instructions that are executed by the TVM.

Event Is a handle that can be used to refer to any asynchronously executing operation on a device. Any asynchronously executing TVM bytecode returns an event. The event can be used for profiling, to determine how long the task took to execute, or scheduling, by placing it on an event queue.

Event Queues Hold a list of references to events.

TVM Client Is the bridge between a TVM bytecode and the code that executes the command.

Command Queue The TVM does not execute code directly but issues commands to the device driver. Internally, the device driver has a queue that records all the commands that are awaiting execution on the device.

References

[1] AMD. *APARAPI*. 2012. URL: http://aparapi.github.io.

[2] *ARM: big.LITTLE Processing*. ARM Holdings Ltd. 2014. URL: http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[3] Peter Abeles. *Efficient Java Matrix Library*. 2016. URL: http://ejml.org.

[4] K. Aingaran et al. "M7: Oracle's Next-Generation Sparc Processor". In: *IEEE Micro* 35.2 (2015), pp. 36–45. ISSN: 0272-1732. DOI: 10.1109/MM.2015.35.

[5] George Almási and David Padua. "MaJIC: Compiling MATLAB for Speed and Responsiveness". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI '02. Berlin, Germany: ACM, 2002, pp. 294–303. ISBN: 1-58113-463-0. DOI: 10.1145/512529.512564. URL: http://doi.acm.org/10.1145/512529.512564.

[6] T. S. Anantharaman and R. Bisiani. "A Hardware Accelerator for Speech Recognition Algorithms". In: *Proceedings of the 13th Annual International Symposium on Computer Architecture*. ISCA '86. Tokyo, Japan: IEEE Computer Society Press, 1986, pp. 216–223. ISBN: 0-8186-0719-X. URL: http://dl.acm.org/citation.cfm?id=17407.17382.

[7] Joshua Auerbach et al. "Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 89–108. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869469. URL: http://doi.acm.org/10.1145/1869459.1869469.

[8] TIOBE Software BV. *TIOBE Index*. 2017. URL: https://www.tiobe.com/tiobe-index/.

[9] J. W. Backus et al. “The FORTRAN Automatic Coding System”. In: *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*. IRE-AIEE-ACM ’57 (Western). Los Angeles, California: ACM, 1957, pp. 188–198. DOI: 10.1145/1455567.1455599. URL: http://doi.acm.org/10.1145/1455567.1455599.

[10] Ram Banin. “Hardware Accelerators in the Design Automation Environment”. In: *Proceedings of the 21st Design Automation Conference*. DAC ’84. Albuquerque, New Mexico, USA: IEEE Press, 1984, pp. 648–. ISBN: 0-8186-0542-1. URL: http://dl.acm.org/citation.cfm?id=800033.800868.

[11] G. H. Barnes et al. “The ILLIAC IV Computer”. In: *IEEE Transactions on Computers* C-17.8 (1968), pp. 746–757. ISSN: 0018-9340. DOI: 10.1109/TC.1968.229158.

[12] James Bergstra et al. “Theano: a CPU and GPU Math Expression Compiler”. In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation. Austin, TX, June 2010.

[13] P. J. Besl and H. D. McKay. “A method for registration of 3-D shapes”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (1992), pp. 239–256. ISSN: 0162-8828. DOI: 10.1109/34.121791.

[14] Guy E. Blelloch et al. “Implementation of a Portable Nested Data-parallel Language”. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’93. San Diego, California, USA: ACM, 1993, pp. 102–111. ISBN: 0-89791-589-5. DOI: 10.1145/155332.155343. URL: http://doi.acm.org/10.1145/155332.155343.

[15] OpenMP Architecture Review Board. *OpenMP Application Program Interface (Version 4.0)*. 2013. URL: http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[16] OpenMP Architecture Review Board. *OpenMP Application Program Interface (Version 4.5)*. 2015. URL: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[17] Kevin J. Brown et al. “A Heterogeneous Parallel Framework for Domain-Specific Languages”. In: *Proceedings of the 2011 International Conference*

on Parallel Architectures and Compilation Techniques. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 89–100. ISBN: 978-0-7695-4566-0. DOI: 10.1109/PACT.2011.15. URL: http://dx.doi.org/10.1109/PACT.2011.15.

[18] Ian Buck et al. "Brook for GPUs: Stream Computing on Graphics Hardware". In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. Los Angeles, California: ACM, 2004, pp. 777–786. DOI: 10.1145/1186562.1015800. URL: http://doi.acm.org/10.1145/1186562.1015800.

[19] Michael Budde, Martin Dybdal, and Martin Elsman. "Compiling APL to Accelerate Through a Typed Array Intermediate Language". In: *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY 2015. Portland, OR, USA: ACM, 2015, pp. 13–18. ISBN: 978-1-4503-3584-3. DOI: 10.1145/2774959.2774966. URL: http://doi.acm.org/10.1145/2774959.2774966.

[20] Stephen Cass. *The 2017 Top Programming Languages*. IEEE Spectrum. 2017. URL: http://spectrum.ieee.org/computing/software/the-2017-top-programming-languages.

[21] Stephen Cass. *The 2018 Top Programming Languages*. IEEE Spectrum. 2018. URL: https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages.

[22] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. "Copperhead: Compiling an Embedded Data Parallel Language". In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPoPP '11. San Antonio, TX, USA: ACM, 2011, pp. 47–56. ISBN: 978-1-4503-0119-0. DOI: 10.1145/1941553.1941562. URL: http://doi.acm.org/10.1145/1941553.1941562.

[23] Olivier Chafik. *ScalaCL: Faster Scala: Optimizing Compiler Plugin + GPU-based Collections (OpenCL)*. 2011. URL: https://github.com/nativelibs4java/ScalaCL.

[24] Manuel M. T. Chakravarty et al. "Data Parallel Haskell: A Status Report". In: *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*. DAMP '07. Nice, France: ACM, 2007, pp. 10–18. ISBN: 978-1-59593-690-5. DOI: 10.1145/1248648.1248652. URL: http://doi.acm.org/10.1145/1248648.1248652.

[25] Manuel M.T. Chakravarty et al. "Accelerating Haskell Array Codes with Multicore GPUs". In: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. DAMP '11. Austin, Texas, USA: ACM, 2011, pp. 3–14. ISBN: 978-1-4503-0486-3. DOI: 10.1145/1926354.1926358. URL: http://doi.acm.org/10.1145/1926354.1926358.

[26] C. Chambers, D. Ungar, and E. Lee. "An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes". In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. OOPSLA '89. New Orleans, Louisiana, USA: ACM, 1989, pp. 49–70. ISBN: 0-89791-333-7. DOI: 10.1145/74877.74884. URL: http://doi.acm.org/10.1145/74877.74884.

[27] Linchuan Chen, Xin Huo, and Gagan Agrawal. "Accelerating MapReduce on a coupled CPU-GPU architecture". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 25:1–25:11. ISBN: 978-1-4673-0804-5. URL: http://dl.acm.org/citation.cfm?id=2388996.2389030.

[28] James Clarkson et al. "Boosting Java Performance Using GPGPUs". In: *Architecture of Computing Systems - ARCS 2017*. Ed. by Jens Knoop et al. Cham: Springer International Publishing, 2017, pp. 59–70. ISBN: 978-3-319-54999-6.

[29] James Clarkson et al. "Exploiting High-performance Heterogeneous Hardware for Java Programs Using Graal". In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. ManLang '18. Linz, Austria: ACM, 2018, 4:1–4:13. ISBN: 978-1-4503-6424-9. DOI: 10.1145/3237009.3237016. URL: http://doi.acm.org/10.1145/3237009.3237016.

[30] James Clarkson et al. "Towards Practical Heterogeneous Virtual Machines". In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. Programming 2018. Nice, France: ACM, 2018, pp. 46–48. ISBN: 978-1-4503-5513-1. DOI: 10.1145/3191697.3191730. URL: http://doi.acm.org/10.1145/3191697.3191730.

[31] P. Colangelo et al. "Fine-Grained Acceleration of Binary Neural Networks Using Intel Xeon Processor with Integrated FPGA". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017, pp. 135–135. DOI: 10.1109/FCCM.2017.46.

[32] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0-262-53086-4.

[33] Alexander Collins et al. “NOVA: A Functional Language for Data Parallelism”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: ACM, 2014, 8:8–8:13. ISBN: 978-1-4503-2937-8. DOI: 10.1145/2627373.2627375. URL: http://doi.acm.org/10.1145/2627373.2627375.

[34] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. “Torch7: A Matlab-like Environment for Machine Learning”. In: *BigLearn, NIPS Workshop*. 2011.

[35] J. Davies. “The Bifrost GPU architecture and the ARM Mali-G71 GPU”. In: *2016 IEEE Hot Chips 28 Symposium (HCS)*. 2016, pp. 1–31. DOI: 10.1109/HOTCHIPS.2016.7936201.

[36] L. Peter Deutsch and Allan M. Schiffman. “Efficient Implementation of the Smalltalk-80 System”. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: ACM, 1984, pp. 297–302. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800542. URL: http://doi.acm.org/10.1145/800017.800542.

[37] Georg Dotzler, Ronald Veldema, and Michael Klemm. “JCudaMP”. In: *Proceedings of the 3rd International Workshop on Multicore Software Engineering*. 2010. DOI: 10.1145/1808954.1808959. URL: http://portal.acm.org/citation.cfm?doid=1808954.1808959.

[38] Christophe Dubach et al. “Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers)”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 1–12. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254066. URL: http://doi.acm.org/10.1145/2254064.2254066.

[39] G. Duboscq et al. “Graal IR: An extensible declarative intermediate representation”. In: *Asia-Pacific Programming Languages and Compilers*. 2013.

[40] Gilles Duboscq et al. "An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler". In: *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*. VMIL '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 1–10. ISBN: 978-1-4503-2601-8. DOI: 10.1145/2542142.2542143. URL: http://doi.acm.org/10.1145/2542142.2542143.

[41] M. Duranton et al. *The HiPEAC Vision 2017*. Tech. rep. HiPEAC, 2017. URL: https://www.hipeac.net/v17.

[42] Ismail El-Helw, Rutger Hofman, and Henri E. Bal. "Glasswing: Accelerating Mapreduce on Multi-core and Many-core Clusters". In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: ACM, 2014, pp. 295–298. ISBN: 978-1-4503-2749-7. DOI: 10.1145/2600212.2600706. URL: http://doi.acm.org/10.1145/2600212.2600706.

[43] Joseph A. Fisher. "Very Long Instruction Word Architectures and the ELI-512". In: *Proceedings of the 10th Annual International Symposium on Computer Architecture*. ISCA '83. Stockholm, Sweden: ACM, 1983, pp. 140–150. ISBN: 0-89791-101-6. DOI: 10.1145/800046.801649. URL: http://doi.acm.org/10.1145/800046.801649.

[44] M. J. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. ISSN: 0018-9219. DOI: 10.1109/PROC.1966.5273. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1447203.

[45] Henry Fuchs and John Poulton. "Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine". In: 2 (Jan. 1982). URL: http://ai.eecs.umich.edu/people/conway/VLSI/VLSIDesMag/Articles/Pixel-Planes.V3Q81.pdf.

[46] Juan José Fumero, Michel Steuwer, and Christophe Dubach. "A Composable Array Function Interface for Heterogeneous Computing in Java". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: ACM, 2014, 44:44–44:49. ISBN: 978-1-4503-2937-8. DOI: 10.1145/2627373.2627381. URL: http://doi.acm.org/10.1145/2627373.2627381.

[47] Juan Fumero et al. “Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation”. In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’17. Xi’an, China: ACM, 2017, pp. 60–73. ISBN: 978-1-4503-4948-2. DOI: 10.1145/3050748.3050761. URL: http://doi.acm.org/10.1145/3050748.3050761.

[48] Yoshihiko Futamura. “Partial Evaluation of Computation Process — An Approach to a Compiler-Compiler”. In: *Higher Order Symbol. Comput.* 12.4 (Dec. 1999), pp. 381–391. ISSN: 1388-3690. DOI: 10.1023/A:1010095604496. URL: http://dx.doi.org/10.1023/A:1010095604496.

[49] Rahul Garg and Laurie Hendren. “Just-in-time Shape Inference for Array-based Languages”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY’14. Edinburgh, United Kingdom: ACM, 2014, 50:50–50:55. ISBN: 978-1-4503-2937-8. DOI: 10.1145/2627373.2627382. URL: http://doi.acm.org/10.1145/2627373.2627382.

[50] Rahul Garg and Laurie Hendren. “Velociraptor: An Embedded Compiler Toolkit for Numerical Programs Targeting CPUs and GPUs”. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT ’14. Edmonton, AB, Canada: ACM, 2014, pp. 317–330. ISBN: 978-1-4503-2809-8. DOI: 10.1145/2628071.2628097. URL: http://doi.acm.org/10.1145/2628071.2628097.

[51] Adele Goldberg and David Robson. *Smalltalk-80: The Language And Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-11371-6.

[52] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN: 0201634511.

[53] Tobias Grosser and Torsten Hoefler. “Polly-ACC Transparent Compilation to Heterogeneous Hardware”. In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS ’16. Istanbul, Turkey: ACM, 2016, 1:1–1:13. ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926286. URL: http://doi.acm.org/10.1145/2925426.2926286.

[54] Max Grossman, Shams Imam, and Vivek Sarkar. "HJ-OpenCL: Reducing the Gap Between the JVM and Accelerators". In: *Proceedings of the Principles and Practices of Programming on The Java Platform*. PPPJ '15. Melbourne, FL, USA: ACM, 2015, pp. 2–15. ISBN: 978-1-4503-3712-0. DOI: 10.1145/2807426.2807427. URL: http://doi.acm.org/10.1145/2807426.2807427.

[55] J.R. Gurd. "The Manchester dataflow machine". In: *Future Generation Computer Systems* 1.4 (1985), pp. 201 –212. ISSN: 0167-739X. DOI: http://dx.doi.org/10.1016/0167-739X(85)90009-3. URL: http://www.sciencedirect.com/science/article/pii/0167739X85900093.

[56] HSA Foundation. *HSA Programmer Reference Manual Specification 1.1*. 2016. URL: http://www.hsafoundation.com/?ddownload=5115.

[57] A. Handa et al. "A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM". In: *IEEE Intl. Conf. on Robotics and Automation, ICRA*. Hong Kong, China, 2014.

[58] Akihiro Hayashi et al. "Accelerating Habanero-Java Programs with OpenCL Generation". In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. Stuttgart, Germany, 2013. ISBN: 978-1-4503-2111-2. DOI: 10.1145/2500828.2500840. URL: http://doi.acm.org/10.1145/2500828.2500840.

[59] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. "Continuations and Coroutines". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: ACM, 1984, pp. 293–298. ISBN: 0-89791-142-3. DOI: 10.1145/800055.802046. URL: http://doi.acm.org/10.1145/800055.802046.

[60] Patrick M. Hefferan et al. "The STE-264 Accelerated Electronic CAD System". In: *Proceedings of the 22Nd ACM/IEEE Design Automation Conference*. DAC '85. Las Vegas, Nevada, USA: IEEE Press, 1985, pp. 352–358. ISBN: 0-8186-0635-5. URL: http://dl.acm.org/citation.cfm?id=317825.317912.

[61] John Hennessy. *Computer architecture : a quantitative approach*. 4th ed. Amsterdam ;;Boston: Morgan Kaufmann, 2007. ISBN: 9780123704900.

[62] Sylvain Henry. “ViperVM: A Runtime System for Parallel Functional High-performance Computing on Heterogeneous Architectures”. In: *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 3–12. ISBN: 978-1-4503-2381-9. DOI: 10.1145/2502323.2502329. URL: http://doi.acm.org/10.1145/2502323.2502329.

[63] Stephan Herhut et al. “River Trail: A Path to Parallelism in JavaScript”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 729–744. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509516. URL: http://doi.acm.org/10.1145/2509136.2509516.

[64] Urs Hölzle, Craig Chambers, and David Ungar. “Debugging Optimized Code with Dynamic Deoptimization”. In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. PLDI ’92. San Francisco, California, USA: ACM, 1992, pp. 32–43. ISBN: 0-89791-475-9. DOI: 10.1145/143095.143114. URL: http://doi.acm.org/10.1145/143095.143114.

[65] Urs Hölzle and David Ungar. “A Third-generation SELF Implementation: Reconciling Responsiveness with Performance”. In: *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*. OOPSLA ’94. Portland, Oregon, USA: ACM, 1994, pp. 229–243. ISBN: 0-89791-688-3. DOI: 10.1145/191080.191116. URL: http://doi.acm.org/10.1145/191080.191116.

[66] IBM Corporation. *J9 JVM*. 2016. URL: https://www.ibm.com/developerworks/java/jdk/.

[67] *Intel AVX - Intel Software Network*. Intel. 2017. URL: http://software.intel.com/en-us/avx/.

[68] K. Ishizaki et al. “Compiling and Optimizing Java 8 Programs for GPU Execution”. In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 2015, pp. 419–431. DOI: 10.1109/PACT.2015.46.

[69] *JavaScript*. Mozilla. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[70] Julien Langou Jim Demmel Jack Dongarra. *LAPACK - Linear Algebra PACKage*. 1992. URL: http://www.netlib.org/lapack/.

[71] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. “An Experiment in Partial Evaluation: The Generation of a Compiler Generator”. In: *SIGPLAN Not.* 20.8 (Aug. 1985), pp. 82–87. ISSN: 0362-1340. DOI: 10.1145/988346.988358. URL: http://doi.acm.org/10.1145/988346.988358.

[72] J. Kessenich, B. Ouriel, and R Krisch. *The SPIR-V Specification*. Khronos Group. 2018. URL: https://www.khronos.org/registry/spir-v/.

[73] Khronos Group. *OpenCL*. 2017. URL: https://www.khronos.org/opencl/.

[74] Khronos OpenCL Working Group. *The OpenCL Specification Version 2.1*. 2015. URL: https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf.

[75] David B. Kirk and Wen-mei W. Hwu. “Programming Massively Parallel Processors (Third Edition)”. In: ed. by David B. Kirk and Wen-mei W. Hwu. Third Edition. Morgan Kaufmann, 2017, pp. 1 –18. ISBN: 978-0-12-811986-0. DOI: https://doi.org/10.1016/B978-0-12-811986-0.00001-7. URL: http://www.sciencedirect.com/science/article/pii/B9780128119860000017.

[76] Andreas Klöckner et al. “PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation”. In: *Parallel Comput.* 38.3 (Mar. 2012), pp. 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001. URL: http://dx.doi.org/10.1016/j.parco.2011.09.001.

[77] Christos Kotselidis et al. “Heterogeneous Managed Runtime Systems: A Computer Vision Case Study”. In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’17. Xi’an, China: ACM, 2017, pp. 74–82. ISBN: 978-1-4503-4948-2. DOI: 10.1145/3050748.3050764. URL: http://doi.acm.org/10.1145/3050748.3050764.

[78] Thomas Kotzmann et al. “Design of the Java HotSpot client compiler for Java 6”. In: *ACM Transactions on Architecture and Code Optimization* 5.1 (2008), pp. 1–32. ISSN: 15443566. DOI: 10.1145/1369396.1370017. URL: http://portal.acm.org/citation.cfm?doid=1369396.1370017.

[79] A. Krishna et al. “Hardware acceleration in the IBM PowerEN processor: architecture and performance”. In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 389–399.

[80] LangPop.com. *Programmming Language Popularity*. 2017. URL: http://langpop.com/.

[81] Doug Lea. “A Java Fork/Join Framework”. In: *Proceedings of the ACM 2000 Conference on Java Grande*. JAVA ’00. San Francisco, California, USA: ACM, 2000, pp. 36–43. ISBN: 1-58113-288-3. DOI: 10.1145/337449.337465. URL: http://doi.acm.org/10.1145/337449.337465.

[82] Calle Lejdfors and Lennart Ohlsson. “Implementing an Embedded GPU Language by Combining Translation and Generation”. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. SAC ’06. Dijon, France: ACM, 2006, pp. 1610–1614. ISBN: 1-59593-108-2. DOI: 10.1145/1141277.1141654. URL: http://doi.acm.org/10.1145/1141277.1141654.

[83] Adam Levinthal and Thomas Porter. “Chap - a SIMD Graphics Processor”. In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’84. New York, NY, USA: ACM, 1984, pp. 77–82. ISBN: 0-89791-138-5. DOI: 10.1145/800031.808581. URL: http://doi.acm.org/10.1145/800031.808581.

[84] Erik Lindholm et al. “NVIDIA Tesla: A Unified Graphics and Computing Architecture”. In: *IEEE Micro* 28.2 (2008), pp. 39–55. ISSN: 0272-1732. DOI: 10.1109/MM.2008.31. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4523358.

[85] J. Macri. “AMD’s next generation GPU and high bandwidth memory architecture: FURY”. In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. 2015, pp. 1–26. DOI: 10.1109/HOTCHIPS.2015.7477461.

[86] Geoffrey Mainland and Greg Morrisett. “Nikola: Embedding Compiled GPU Functions in Haskell”. In: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell ’10. Baltimore, Maryland, USA: ACM, 2010, pp. 67–78. ISBN: 978-1-4503-0252-4. DOI: 10.1145/1863523.1863533. URL: http://doi.acm.org/10.1145/1863523.1863533.

[87] Frank Mueller and Yongpeng Zhang. "Hidp: A Hierarchical Data Parallel Language". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11. ISBN: 978-1-4673-5524-7. DOI: 10.1109/CGO.2013.6494994. URL: http://dx.doi.org/10.1109/CGO.2013.6494994.

[88] *NEON architecture overview*. ARM Holdings. 2017. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/CACDJDDF.html.

[89] NVIDIA Corporation. *CUDA*. 2017. URL: http://developer.nvidia.com/cuda-zone.

[90] NVIDIA Corporation. *Parallel Thread Execution ISA Version 4.0*. 2017. URL: http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[91] Luigi Nardi et al. "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM". In: *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. arXiv:1410.2167. 2015.

[92] Richard A. Newcombe et al. "KinectFusion: Real-time Dense Surface Mapping and Tracking". In: *Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality*. ISMAR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 127–136. ISBN: 978-1-4577-2183-0. DOI: 10.1109/ISMAR.2011.6092378. URL: http://dx.doi.org/10.1109/ISMAR.2011.6092378.

[93] Nathaniel Nystrom, Derek White, and Kishen Das. "Firepile: Run-time Compilation for GPUs in Scala". In: *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*. GPCE '11. Portland, Oregon, USA: ACM, 2011, pp. 107–116. ISBN: 978-1-4503-0689-8. DOI: 10.1145/2047862.2047883. URL: http://doi.acm.org/10.1145/2047862.2047883.

[94] OpenACC-Standard.org. *The OpenACC Application Programming Interface (Version 2.5)*. 2015. URL: http://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf.

[95] OpenJDK. *Project Sumatra*. 2014. URL: http://openjdk.java.net/projects/sumatra/.

[96] Michael Paleczny, Christopher Vick, and Cliff Click. "The Java hotspot™ Server Compiler". In: *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*. JVM'01. Monterey, California: USENIX Association, 2001, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1267847.1267848.

[97] John Palmer. "The Intel® 8087 Numeric Data Processor". In: *Proceedings of the 7th Annual Symposium on Computer Architecture*. ISCA '80. La Baule, USA: ACM, 1980, pp. 174–181. DOI: 10.1145/800053.801923. URL: http://doi.acm.org/10.1145/800053.801923.

[98] P.C. Pratt-Szeliga, J.W. Fawcett, and R.D. Welch. "Rootbeer: Seamlessly Using GPUs from Java". In: *Proceedings of 14th International IEEE High Performance Computing and Communication Conference on Embedded Software and Systems*. 2012. DOI: 10.1109/HPCC.2012.57.

[99] Nishkam Ravi et al. "Apricot: An Optimizing Compiler and Productivity Tool for x86-compatible Many-core Coprocessors". In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: ACM, 2012, pp. 47–58. ISBN: 978-1-4503-1316-2. DOI: 10.1145/2304576.2304585. URL: http://doi.acm.org/10.1145/2304576.2304585.

[100] Dennis M. Ritchie. "The Development of the C Language". In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: ACM, 1993, pp. 201–208. ISBN: 0-89791-570-4. DOI: 10.1145/154766.155580. URL: http://doi.acm.org/10.1145/154766.155580.

[101] Christopher J. Rossbach et al. "Dandelion: A Compiler and Runtime for Heterogeneous Systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 49–68. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522715. URL: http://doi.acm.org/10.1145/2517349.2522715.

[102] Alex Rubinsteyn et al. “Parakeet: A Just-in-time Parallel Accelerator for Python”. In: *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*. HotPar’12. Berkeley, CA: USENIX Association, 2012, pp. 14–14. URL: http://dl.acm.org/citation.cfm?id=2342788.2342802.

[103] *Ruby: A Programmers Best Friend*. The Ruby Community. 2018. URL: http://www.ruby-lang.org/.

[104] S. Saeedi et al. “Navigating the Landscape for Real-Time Localization and Mapping for Robotics and Virtual and Augmented Reality”. In: *Proceedings of the IEEE* (2018), pp. 1–20. ISSN: 0018-9219. DOI: 10.1109/JPROC.2018.2856739.

[105] Paul B. Schneck. “The CDC STAR-100”. In: *Supercomputer Architecture*. Boston, MA: Springer US, 1987, pp. 99–117. ISBN: 978-1-4615-7957-1. DOI: 10.1007/978-1-4615-7957-1_5. URL: https://doi.org/10.1007/978-1-4615-7957-1_5.

[106] Amin Shali and William R. Cook. “Hybrid Partial Evaluation”. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’11. Portland, Oregon, USA: ACM, 2011, pp. 375–390. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048098. URL: http://doi.acm.org/10.1145/2048066.2048098.

[107] M. Sharir. “Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers”. In: *Comput. Lang.* 5.3-4 (Jan. 1980), pp. 141–153. ISSN: 0096-0551. DOI: 10.1016/0096-0551(80)90007-7. URL: http://dx.doi.org/10.1016/0096-0551(80)90007-7.

[108] Brian Smith. “Procedural Reflection in Programmable Languages”. PhD thesis. Massachusetts Institute of Technology, 1982. DOI: http://hdl.handle.net/1721.1/15961.

[109] James E. Smith. “Decoupled Access/Execute Computer Architectures”. In: *Proceedings of the 9th Annual Symposium on Computer Architecture*. ISCA ’82. Austin, Texas, USA: IEEE Computer Society Press, 1982, pp. 112–119. URL: http://dl.acm.org/citation.cfm?id=800048.801719.

[110] A. Snavely et al. “Multi-processor Performance on the Tera MTA”. In: *Supercomputing, 1998.SC98. IEEE/ACM Conference on*. 1998, pp. 4–4. DOI: 10.1109/SC.1998.10049.

[111] A. Sodani et al. “Knights Landing: Second-Generation Intel Xeon Phi Product”. In: *IEEE Micro* 36.2 (2016), pp. 34–46. ISSN: 0272-1732. DOI: 10.1109/MM.2016.25.

[112] Philip M. Spira and Carl Hage. “Hardware Acceleration of Gate Array Layout”. In: *Proceedings of the 22Nd ACM/IEEE Design Automation Conference*. DAC ’85. Las Vegas, Nevada, USA: IEEE Press, 1985, pp. 359–366. ISBN: 0-8186-0635-5. URL: http://dl.acm.org/citation.cfm?id=317825.317913.

[113] Nigel Stephens et al. “The ARM Scalable Vector Extension”. In: *IEEE Micro* 37.2 (Mar. 2017), pp. 26–39. ISSN: 0272-1732. DOI: 10.1109/MM.2017.35. URL: https://doi.org/10.1109/MM.2017.35.

[114] Michel Steuwer and Sergei Gorlatch. “SkelCL: A High-level Extension of OpenCL for multi-GPU Systems”. In: *J. Supercomput.* 69.1 (July 2014), pp. 25–33. ISSN: 0920-8542. DOI: 10.1007/s11227-014-1213-y. URL: http://dx.doi.org/10.1007/s11227-014-1213-y.

[115] Andrew Stromme, Ryan Carlson, and Tia Newhall. “Chestnut: A GPU Programming Language for Non-experts”. In: *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM ’12. New Orleans, Louisiana: ACM, 2012, pp. 156–167. ISBN: 978-1-4503-1211-0. DOI: 10.1145/2141702.2141720. URL: http://doi.acm.org/10.1145/2141702.2141720.

[116] Bjarne Stroustrup. “An Overview of C++”. In: *Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming*. OOPWORK ’86. Yorktown Heights, New York, USA: ACM, 1986, pp. 7–18. ISBN: 0-89791-205-5. DOI: 10.1145/323779.323736. URL: http://doi.acm.org/10.1145/323779.323736.

[117] Gregory T. Sullivan. “Dynamic Partial Evaluation”. In: *Programs as Data Objects: Second Symposium, PADO2001 Aarhus, Denmark, May 21–23, 2001 Proceedings*. Ed. by Olivier Danvy and Andrzej Filinski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 238–256. ISBN: 978-3-540-44978-2.

DOI: 10.1007/3-540-44978-7_14. URL: https://doi.org/10.1007/3-540-44978-7_14.

[118] *The R Project for Statistical Computing*. The R Foundation. 2018. URL: http://www.r-project.org/.

[119] R. M. Tomasulo. “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”. In: *IBM Journal of Research and Development* 11.1 (1967), pp. 25–33. ISSN: 0018-8646. DOI: 10.1147/rd.111.0025.

[120] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. “Towards dense linear algebra for hybrid GPU accelerated manycore systems”. In: *Parallel Computing* 36.5-6 (June 2010), pp. 232–240. ISSN: 0167-8191. DOI: 10.1016/j.parco.2009.12.005.

[121] John G. Torborg. “A Parallel Processor Architecture for Graphics Arithmetic Operations”. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’87. New York, NY, USA: ACM, 1987, pp. 197–204. ISBN: 0-89791-227-6. DOI: 10.1145/37401.37426. URL: http://doi.acm.org/10.1145/37401.37426.

[122] Raja Vallèe-Rai et al. “Soot - a Java Optimization Framework”. In: *Proceedings of CASCON 1999*. 1999. URL: www.sable.mcgill.ca/publications.

[123] W. J. Watson. “The TI ASC: A Highly Modular and Flexible Super Computer Architecture”. In: *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*. AFIPS ’72 (Fall, part I). Anaheim, California: ACM, 1972, pp. 221–228. DOI: 10.1145/1479992.1480022. URL: http://doi.acm.org/10.1145/1479992.1480022.

[124] Yonghong Yan, Max Grossman, and Vivek Sarkar. “JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA”. In: *Euro-Par 2009 Parallel Processing*. Ed. by Henk Sips, Dick Epema, and Hai-Xiang Lin. Vol. 5704. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-03868-6. URL: http://www.springerlink.com/index/10.1007/978-3-642-03869-3_82.

[125] Wojciech Zaremba, Yuan Lin, and Vinod Grover. “JaBEE: Framework for Object-oriented Java Bytecode Compilation and Execution on Graphics Processor Units”. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. GPGPU-5. London, United

Kingdom: ACM, 2012, pp. 74–83. ISBN: 978-1-4503-1233-2. DOI: 10.1145/2159430.2159439. URL: http://doi.acm.org/10.1145/2159430.2159439.

[126] Zhengyou Zhang. “Iterative Point Matching for Registration of Free-form Curves and Surfaces”. In: *Int. J. Comput. Vision* 13.2 (Oct. 1994), pp. 119–152. ISSN: 0920-5691. DOI: 10.1007/BF01427149. URL: http://dx.doi.org/10.1007/BF01427149.

[127] jocl.org. *Java Bindings For OpenCL*. 2016. URL: http://www.jocl.org/.