ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/327097904

Exploiting High-Performance Heterogeneous Hardware for Java Programs using Graal

Preprint · September 2018

DOI: 10.1145/3237009.3237016

CITATIONS
4

READS
1,615

7 authors, including:

Juan Fumero
The University of Manchester
30 PUBLICATIONS **153** CITATIONS
SEE PROFILE

Michalis Papadimitriou
The University of Manchester
7 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Foivos Zakkak
The University of Manchester
19 PUBLICATIONS **35** CITATIONS
SEE PROFILE

Maria Xekalaki
The University of Manchester
7 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

TERAFLUX View project

Transactional Memory View project

All content following this page was uploaded by Foivos Zakkak on 19 December 2018.

The user has requested enhancement of the downloaded file.

Exploiting High-Performance Heterogeneous Hardware for Java Programs using Graal

James Clarkson
The University of Manchester
Manchester, UK
james.clarkson@manchester.ac.uk

Juan Fumero
The University of Manchester
Manchester, UK
juan.fumero@manchester.ac.uk

Michail Papadimitriou
The University of Manchester
Manchester, UK
michail.papadimitriou@manchester.ac.uk

Foivos S. Zakkak
The University of Manchester
Manchester, UK
foivos.zakkak@manchester.ac.uk

Maria Xekalaki
The University of Manchester
Manchester, UK
maria.xekalaki@manchester.ac.uk

Christos Kotselidis
The University of Manchester
Manchester, UK
christos.kotselidis@manchester.ac.uk

Mikel Luján
The University of Manchester
Manchester, UK
mikel.lujan@manchester.ac.uk

ABSTRACT

The proliferation of heterogeneous hardware in recent years means that every system we program is likely to include a mix of compute elements; each with different characteristics. By utilizing these available hardware resources, developers can improve the performance and energy efficiency of their applications. However, existing tools for heterogeneous programming neglect developers who do not have the time or inclination to switch programming languages or learn the intricacies of a specific piece of hardware.

This paper presents a framework that enables Java applications to be deployed across a variety of heterogeneous systems while exploiting any available multi- or many-core processor. The novel aspect of our approach is that it does not require any *a priori* knowledge of the hardware, or for the developer to worry about managing disparate memory spaces. Java applications are transparently compiled and optimized for the hardware at run-time.

We also present a performance evaluation of our just-in-time (JIT) compiler using a framework to accelerate SLAM, a complex computer vision application entirely written in Java. We show that we can accelerate SLAM up to 150x compared to the Java reference implementation, rendering 107 frames per second (FPS).

CCS CONCEPTS

• **Software and its engineering → Virtual machines**; **Just-in-time compilers**;

KEYWORDS

Heterogeneous Hardware, Java, Virtual Machine, Graal, openCL

ManLang'18, September 12–14, 2018, Linz, Austria
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria*, https://doi.org/10.1145/3237009.3237016.

ACM Reference Format:
James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting High-Performance Heterogeneous Hardware for Java Programs using Graal. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3237009.3237016

1 INTRODUCTION

The majority of programming languages used by everyday developers make the fundamental assumption that the whole program will execute on a single processor. Moreover, the portability of these languages is due to the fact that the majority of systems used today run on the same type of processor: whether it be x86, POWER, ARM or MIPS. In this homogeneous world software development is simplified as computing systems only use a single type of processor: either in a single-core or multi-core configuration. As a result, porting languages in their entirety to use a different processor architecture was enough. Until recently, we did not have the need, or requirement, to use multiple processors of different architectures within a single application.

The pervasion of hardware accelerators into mainstream computing systems is rapidly changing the programming landscape. For example, we can find general purpose graphics accelerators or GPGPUs in mobile phones, tablets, laptops, PCs, and servers. Since these accelerators are programmable, it is natural to assume that developers wish to utilize them in order to achieve improvements in performance and/or energy-efficiency. However, in order to develop effective programming languages for this heterogeneous hardware we need to invalidate one long-standing assumption — that all applications execute exclusively on homogeneous hardware.

To this end, heterogeneous programming languages have emerged. These languages are specifically designed so that applications can utilize multiple types of devices in concert. The popular ones like CUDA [11], OpenCL [24], and OpenACC [38] are born out of the necessity to efficiently program GPGPUs. The consequence of this

is that they all adopt a position where work is *offloaded* from a host-device onto an accelerator; mirroring how the rendering of complex compute graphics is offloaded from a processor onto a GPU. For example, languages such as OpenACC are geared towards the creation of applications which offload computation onto one or more devices of the same type. A true heterogeneous language, however, needs to offer more to developers; such as the ability to construct complex processing pipelines across multiple devices, or the ability to map computation onto the device which is closest to the data it needs to process. These efforts led us to the design and implementation of Tornado, a framework that enables the execution of managed languages onto any OpenCL compatible device such as CPUs, GPUs, FPGAs, and Intel Xeon Phi. Tornado builds upon our previous work [9, 33] and is part of the Beehive Ecosystem [1, 44].

Tornado ultimately aims to assist developers to transparently execute their code onto any OpenCL compatible device. Furthermore, by exploiting the portability of the Java language, via the Java Virtual Machine (JVM), Tornado is able to execute across any JVM compatible architecture — extending its reach beyond that of existing approaches. Finally, we showcase Tornado's maturity and ability to execute real-world complex applications by accelerating a Java version of the Kinect Fusion (KF) application. An application that is typically beyond the computational capability of non-hardware-accelerated implementations.

More specifically, this paper makes the following contributions:

- It presents Tornado, a heterogeneous programming framework for Java that through JIT compilation transparently accelerates applications using hardware accelerators.
- It presents how Tornado can be used to accelerate a complex computer vision application, that is written entirely in Java.
- It evaluates the performance of a complex computer vision application written using Tornado and demonstrates throughput of up to 107 frames per second (FPS).

2 RELATED WORK

This section reviews the most relevant related works to our approach. We focus on Java JIT compilation for heterogeneous computing, and, in particular, to GPUs.

Parallel API. Aparapi [2] is one of the most well-known API and JIT compilers from Java byte-code to OpenCL. Aparapi programmers extend their classes from a common Aparapi base-class and override a `run` method. Data within the kernel can be accessed if it is declared in the same lexical scope of fields in the same class. However, Java exceptions, allocation of new arrays or creating objects are not allowed. Moreover, only programs that follow the parallel map semantics can be expressed with Aparapi.

Sumatra [39] uses the new Java Stream 8 API to generate parallel code for HSAIL architectures using Graal. Sumatra makes use of the `forEach` construct (map parallel semantics) and offloads, at run-time, the Java code passed to the construct to HSAIL (a new assembly-standard for heterogeneous devices).

Other frameworks such as JOCL [31], and JCUDA [43] use OpenCL wrappers for Java. Using these frameworks, programmers have to explicitly implement their kernels in OpenCL or CUDA. This is a handicap for many high-level users, because it requires knowledge about the new parallel architecture and programming model.

Fumero et. al [22, 23] provide a Java API for function composition to program heterogeneous hardware. The API follows a pure functional style within Java to easily identify and generate parallel code. That API, however, relies on Java lambdas and well-known parallel skeletons, such as map, reduce and pipeline, thus requiring the user to change the code from a non-functional style.

Tornado provides just a few annotations to add in the existing Java sequential code (in similar way to OpenMP or OpenACC) to inform the compiler that certain loops are potentially parallel and candidates to execute on parallel hardware (e.g., GPUs). It also provide a very light API to group methods (task-schedule) to highly optimize data transfers.

GPU Compilation for Java. Liquid Metal and the Lime Compiler [3, 15] are a runtime and a language implementation based on Java to execute on GPUs and FPGAs. The Lime compiler statically generates heterogeneous code. Habanero Java [26] is also a Java based language that generates OpenCL code at run-time. It combines compile-time and run-time code generation. In similar way, Rootbeer [40] statically generates CUDA for CUDA devices. With our approach in Tornado, we generate code at run-time. This allows us to specialize the generated code depending on the target device, as we show in Section 3.3.

JaBEE [45] is a compiler framework that generates, at run-time, CUDA from Java-bytecode. JaBEE supports many Java features such as virtual methods and exceptions. However, in our opinion, the reported performance is not good. Tornado makes use of existing compiler optimizations of the JVM, such as escape analysis, aggressive inlining and loop unrolling, that improve the generated code. Tornado also contains an optimizing runtime that improves data movement between Java and the accelerator.

Ishizaki et. al [30] present a GPU JIT compiler for generating CUDA code for Java collections. In a similar way to Sumatra [39], their compiler generates parallel code from Java lambda expressions for the `forEach` construct of the Java Stream API. To take advantage of the automatic GPU JIT compilation, programmers need to adapt their code to use these streams. Tornado takes a different approach in which changes to the source code are minimal (as we show in Section 3). Moreover, Tornado can compile any arbitrary Java code, not only lambda expressions representing the map parallel semantics.

GPU JIT compilation for other managed languages. There are some works that generate at run-time, GPU code from high-level and interpreted languages. Haskell [8, 28, 34], Python [4, 6, 32, 41], Scala [7, 37], MATLAB [4, 10], JavaScript [29], Lua [10], R [21] and Ruby [42]. As Fumero et. al [21] has demonstrated, the presented solution for generating Java code can be extensible to other high-level programming languages.

Summary. Tornado differs from prior work by: 1) not using a super-set of the Java language [3, 27], 2) not using ahead-of-time compilation [13, 40], 3) not requiring developers to write heterogeneous code in another language [14, 31], 4) not requiring manual parallelization of kernels [2], and 5) supporting native library calls. Furthermore, to the best of our knowledge, Tornado is the first framework that is able to automatically accelerate Java computer vision applications on GPGPUs, as we show in detail in Section 4.

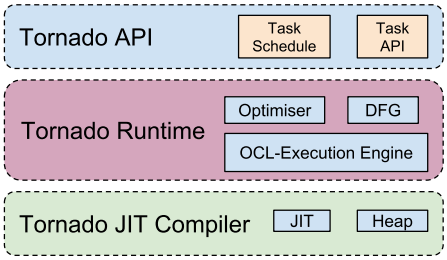


Figure 1: Tornado Overview

3 TORNADO

Tornado is a Java-based parallel programming framework that enables managed programming languages to take advantage of heterogeneous hardware platforms. Figure 1 shows an overview of Tornado, which comprises three main software-layers:

- **Tornado API:** a parallel API which enables developers to identify loops that can be executed in parallel. It also provides an API to compose and build a pipeline of multiple tasks, in which dependencies and optimizations between them are automatically managed in our runtime.
- **Tornado Runtime:** an optimizing runtime that performs data dependence analysis, optimizes data transfers, and orchestrates the parallel execution between the Java host and the target parallel devices.
- **Tornado JIT Compiler:** a JIT compiler that dynamically generates heterogeneous and optimized machine code for the target devices.

The following sections describe each component in more detail.

3.1 Tornado API

To efficiently and safely compile Java code for heterogeneous platforms, Tornado relies on a minimal API that works alongside existing code without requiring developers to re-write their code. We achieve that by providing support for expressing data-parallelism and allowing developers to markup the induction variable of any data-parallel loop with the `@Parallel` annotation. This signals the compiler that each iteration of the loop can be executed independently and that, consequently, it is safe to parallelize it. Note that the use of the annotation does not provide any guarantees that the loop will be parallelized or any information about how it can be parallelized — just that each loop iteration can be performed independently. If the code is executed on a machine without hardware accelerators, the JVM will simply ignore any Tornado annotations.

A key feature of Tornado is its portability across different hardware platforms as we show in Section 4. For this reason, Tornado prohibits developers from deliberately parallelizing code for a specific architecture by not providing a mechanism to explicitly map code onto individual threads. The parallelization is applied automatically by the compiler and is discussed further in Section 3.3. Furthermore, Tornado encourages developers not to specialize data-parallel code for a specific accelerator by using techniques such as loop-tiling — as these are device specific and therefore restrict portability.

Listing 1: A simple Tornado example of array addition.

```
1  public class Compute {
2     public void add(int[] a, int[] b, int[] c) {
3        for (@Parallel int i = 0; i < c.length; i++) {
4           c[i] = a[i] + b[i];
5        }
6     }
7     public void compute(int[] a, int[] b, int[] c) {
8        TaskSchedule s = new TaskSchedule ("s0")
9           .task("t0", this::add, a, b, c)
10          .streamOut(c).execute();
11    }
12 }
```

To manage the execution of Java code on parallel hardware, Tornado employs a task-based programming model. A task is a reference to an existing Java method that has the potential to compute data-parallel code. Each task encapsulates: a) the code to execute, the data it should operate on, and b) meta-data that contains both the compiler and runtime configurations for the task. As data-parallel code is always enclosed within a task, we use tasks as the basic unit of execution for heterogeneous code. Developers have the ability to map each task onto a different device in the following ways: in the application (either statically or dynamically), automatically by the Tornado runtime system, or as a tuning parameter on the command line.

3.1.1 Task-Schedules. A key feature of Tornado is *composability* — the ability to write applications with many tasks that have complex data-dependencies. To achieve that, tasks are not executed directly by the application. Instead, they are scheduled indirectly via a *task-schedule* which provides the Tornado runtime system greater scope for optimizing the execution of tasks. A task-schedule is simply a group of multiple tasks. Therefore, it is a group of multiple Java methods executing data parallel code. A task-schedule provides developers with an easy way to compose complex processing pipelines which might run multiple tasks across multiple accelerators. The task-schedule exists to shield developers from the complexities of scheduling data-movement in complex applications. The result is that Tornado is able to infer all data-movement from the task-schedule and automatically exploit any available *task-parallelism* within it. Moreover, Tornado enables task-schedules to be executed asynchronously and, hence, developers do not need to wait for their completion. This allows developers to automatically overlap code execution between the application running on the JVM and the code running on the accelerators.

Listing 1 illustrates a simple Tornado example of adding two arrays of integers. To execute the `add` method on a hardware accelerator, a task-schedule, `s0`, that contains a single task, `t0`, is created. Here the task-schedule can be thought of as a lexical closure: task `t0` will invoke the `add` method with parameters `a`, `b`, and `c`, but only when the task-schedule is executed. In Tornado, task-schedules are represented internally as data-flow graphs (also called task-graphs) that explicitly model the data-movement between tasks. The nodes in these graphs are individual tasks that could be executing code, allocating memory, or transferring data. The benefit of using this

(a) By default data are left *in-situ* so that it can be reused by subsequent tasks running on the same device.

(b) An explicit copy needs to be generated using the `streamOut(c)` operator to transfer the data back to the host.

Figure 2: Data Management

representation is that it is straightforward to generate an optimal execution schedule that exploits task-parallelism but satisfies all data-dependencies.

3.1.2 Data Transfers. As each task has the potential to execute on a different device, managing data-movement is critical to obtain high performance. Tornado manages all data-movement within the task-schedule automatically. By default, all reference types — objects and arrays — are copied onto the accelerator automatically the first time they are used but are never copied back to the host. This way the data will always remain on the last device on which it was created and an explicit request must be made to transfer it back to the host; hence the use of the `streamOut` operator. Figure 2 illustrates the semantics of the task-schedule defined in Listing 1 and why the `streamOut` operator is necessary to transfer `c` back to the host. Although this may seem counter-intuitive for developers who are used to shared memory programming, it provides a new dimension of optimization for heterogeneous programming — *locality*. On balance, the ability to exploit locality in this way helps to dramatically increase performance as opposed to enforcing coherency between disparate physical memories which severely degrades performance. Finally, since the task-schedule works as a closure it is also a synchronization point. That means that when control is returned from `execute`, the host is guaranteed to view all required memory updates.

3.1.3 Task Execution. By default, Tornado will execute all tasks on the first accelerator it will find in the system. However, the API allows assigning names to each task-schedule and task, `s0` and `t0` respectively from Listing 1. This way, task-schedules and tasks can be referenced by name and configured on the command line (or elsewhere in the application). Therefore, different properties such as the accelerator to execute a set of tasks, are not embedded in the source code. This immediately benefits the developer by enabling the configuration of the application without the need of modifications to the source code and a re-compilation of the whole application.

3.1.4 Summary of the API. Tornado provides a minimal and clean API for achieving heterogeneous execution of Java applications. It is mainly based on the notion of tasks and task-schedules that are compositions of calls to existing code — allowing code to be re-used extensively. Furthermore, the non-intrusive annotations can be used by developers to improve performance without sacrificing backwards-compatibility since they are ignored by the compiler in case Tornado is not activated.

3.2 Tornado Runtime

Figure 3 illustrates a more detailed overview of the Tornado components as well as their interaction along with a typical execution flow. In this section, we shortly discuss each step of the execution flow and the actions taken by Tornado. The execution is driven by task-schedules provided by the developer, which describe a data-flow graph of tasks (referred to as a *task-graph*).

Task Graph Optimizer. A task-graph is constructed the first time a task-schedule is executed via `execute` or `schedule` and is passed to the Graph Optimizer (1). At this stage the task-graph only contains tasks which execute code. Since Tornado handles data-transfers automatically, the next step is to be augment the task-graph with tasks (or nodes) that handle data-transfers.

Tornado Sketchers. To achieve this, the Graph Optimizer passes each node of the task-graph to the Sketcher which creates a *sketch* of the code that is executed by the node (2). Essentially, the sketcher constructs a High-level Intermediate Representation (HIR) of each task from Java byte-code and places it in a HIR cache so it can be retrieved by the code generator in the future. The Graph Optimizer is also able to query each sketch to aid in the optimization of the task-graph. For example, the sketcher determines the usage of every object accessed within a task — this can be read-only, write-only, read and write, or unknown. Knowing this information, the graph optimizer is able to fully populate the task-graph with the nodes to perform the required data-transfers. This means that data-movement is automatically inferred from the code, unlike OpenACC, OpenMP and OpenCL, where developers have to manually handle data-movement.

This "split-compilation" approach is highly-efficient since the HIR graphs of the tasks are constructed once and can be used multiple times — to compile tasks for different devices or to re-compile for the same device using a different set of optimizations. Additionally, the cache decouples the front-end and the back-end of the compiler, making it possible for multiple back-ends to share the same front-end. This is desirable because it makes it possible to use different heterogeneous frameworks and code generators like OpenCL [24], CUDA/PTX [11, 12] or HSA/HSAIL [19, 20].

Once all sketches are available, the optimizer tries to eliminate as many unnecessary nodes as possible from the task-graph and then generates an optimal execution schedule for the tasks. Here the optimizer aims to minimize the length of the critical path by overlapping the execution of data-transfers and execution where possible. The result is a serialized list of low-level tasks; each one describing an action that is to be applied to an abstract accelerator, e.g., data-transfers and code execution (4). To avoid having to repeatedly call the Graph Optimizer the serialized schedule is cached.

Execution Engine. The execution of the task schedule is performed by passing the serialized schedule to the Task Executor (5). The Task Executor reads the serialized tasks in order and translates them into calls to the driver API (6) — in our case the OpenCL

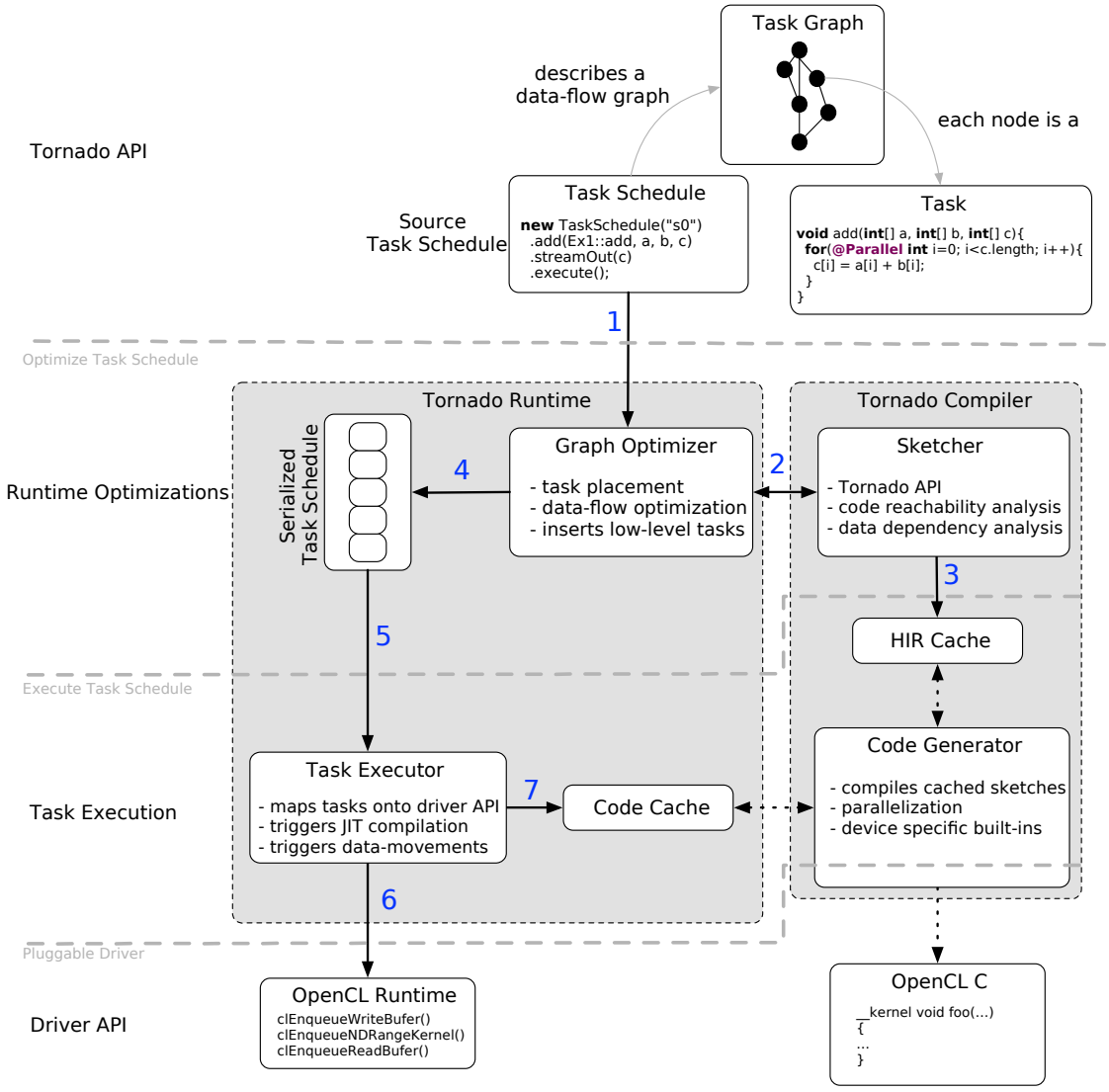


Figure 3: Tornado Architecture Outline.

Runtime API. If the current task executes code on the accelerator, the Task Executor retrieves the compiled code from a code cache (7). In the event that no compiled code exists in the cache, a HIR compilation will be triggered. Note that in this case the HIR will be retrieved from the HIR cache. Furthermore, any parallelization strategy or device specific built-ins are applied to the HIR at this stage. The output of the code generator is compatible with the driver API — in our case we generate OpenCL C code.

3.3 Tornado JIT Compiler

In Tornado, the JIT compiler is responsible for both parallelizing code and generating the Driver API code. The Tornado JIT compiler is built using Graal's API and compiler framework [16, 17]. Graal is an industrial strength JIT compiler with the ability to generate machine code directly from Java byte-code. Tornado augments Graal with the ability to parallelize code (discussed later) and generate the Driver API code.

Figure 4 shows the main workflow of Tornado, emphasizing the JIT compilation part. On the top left is a Java method that performs a vector addition. Note that the loop is annotated with `@Parallel`. As usual, the method is compiled by `javac` into Java byte-code. Then, at run-time, Tornado analyzes the data dependencies, performs optimizations, generates the Driver API code (right side of Figure 4), and orchestrates the parallel execution. As indicated by the gray box on the lower left side of Figure 4, Tornado utilizes the Graal Compiler API and JVM compiler interface (JVMCI) to interact with the JVM. This interaction, enables one of the key features of the Tornado JIT compiler; the integration of the Tornado optimization pipeline with the JVM optimization pipeline. This integration

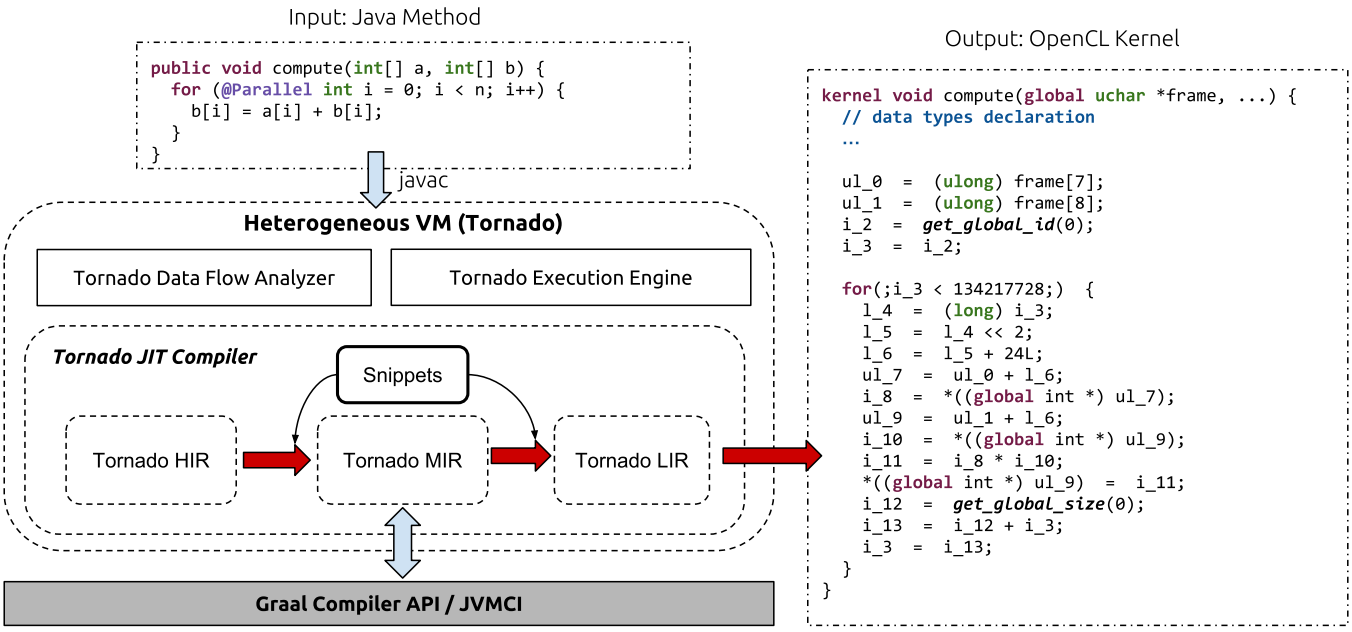


Figure 4: The Tornado workflow.

allows for better optimization by combining typical compiler optimizations, such as inlining and loop unrolling, with automatic parallelization. Moreover, it ensures proper Java semantics for both the code executed on the JVM host as well as for the code executed on the different modules of the underlying hardware.

Apart from the analyses and optimization passes which are already present in Graal, Tornado also applies the following optimization phases to the IR-graph:

- **Tornado Data Flow Analysis:** Analyzes data dependencies between tasks.
- **Tornado Reduce Replacements:** Detects reductions and performs node replacement with special Tornado reduction nodes.
- **Tornado Task Specialization:** Specializes the IR-graph by inlining fields and objects.
- **Tornado Driver API Intrinsics:** Sets nodes for Driver API intrinsics such as barriers and debug information.
- **Tornado Snippet Post-Processing:** Processes non-lowerable IR nodes that are introduced from snippets during the lowering phases.
- **Tornado Shape Analysis:** Analyzes the loop index space and determines the correct indices when using strides in loops.
- **Tornado Parallel Scheduler:** Optimizes data-parallel loops for the target device.

Presently, the Tornado JIT compiler supports two parallelization schemes: 1) the assignment of a thread to a block of iterations (block mapping), and 2) the assignment of one thread to each iteration in the loop. By default, the choice of scheme is governed by the type of the target device, but it is also possible to be dynamically configured. The first scheme provides a coarser thread granularity which suits latency oriented devices, such as x86 cores, whereas the second provides a finer thread granularity which is preferred by throughput oriented devices, like GPGPUs. Listings 2 and 3 demonstrate the generated Driver API code after the parallel scheduler phase transforms the `add` function from the example provided in Listing 1. Listing 2 shows the result of assigning a thread to a block of iterations, suitable for latency oriented devices. On the contrary, Listing 3 shows the result of assigning one thread to each iteration in the loop, better suited for throughput oriented devices.

Listing 2: Loop Re-Written For CPUs.

```
1 int id = get_global_id(0);
2 int size = get_global_size(0);
3 int block_size = (size + inputSize - 1) / size;
4 int start = id * block_size;
5 int end = min(start + bs, c.length);
6 for (int i = start; i < end; i++) {
7   c[i] = a[i] + b[i];
8 }
```

Listing 3: Loop Re-Written For GPGPUs.

```
1 int idx = get_global_id(0);
2 int size = get_global_size(0);
3 for (int i = idx; i < c.length; i += size) {
4   c[i] = a[i] + b[i];
5 }
```

Figure 5 illustrates these compiler transformations at the IR level when optimizing for throughput oriented devices. The left side of the figure corresponds to the IR before the optimization and the right to the IR after it. Each rectangle represents a node in the IR-graph, solid arrows indicate the control flow, and dashed arrows indicate the data flow. The right side of the figure shows the result

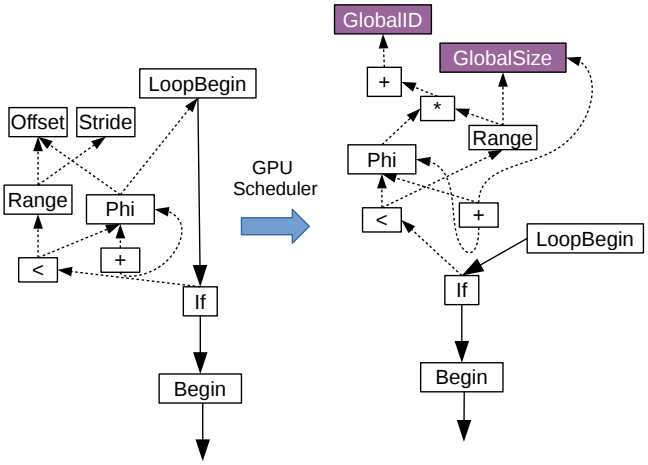


Figure 5: GPU Parallel Scheduler transformation.

of applying this transformation to execute on a GPU (Listing 3). Note that two new nodes appear (`GlobalID` and `GlobalSize`). These nodes are later translated to OpenCL intrinsics to obtain the correct indexes from the loop iteration space on the parallel hardware.

OpenCL C as the Driver API. As is evident by Figures 3 and 4, as well as by Listings 2 and 3, the current implementation of Tornado uses OpenCL C as the Driver API. The decision to use OpenCL C as the Driver API was mainly based on the fact that OpenCL supports a plethora of devices, allowing us to accelerate Java applications on all these devices with a single back-end. Despite the fact that OpenCL allows us to target a wide and diverse range of accelerators, it prohibits us from implementing some vital features of the Java language (e.g., exception handling) while it introduces difficulties when implementing others (e.g., objects). Furthermore, OpenCL comes at the cost of invoking a second JIT compiler. We first need to compile Java byte-code to OpenCL C using the Tornado JIT compiler and then compile the OpenCL C to machine code using an OpenCL-compatible compiler. These issues can be resolved, albeit at the cost of developing multiple back-ends, by generating PTX [12] or HSAIL [20] code for the Driver API.

3.4 Java Coverage

Throughout the development of Tornado, we have found that the only real constraint is the support of features which require calls, either internally to the JVM or externally to a native library or the OS. Typically, this means that features like Java reflection, I/O, or the threading API are unable to be used inside Tornado tasks. The reason behind these restrictions is that tasks need to be able to execute on devices other than the one hosting the OS. Theoretically, Tornado can support the majority of the Java language. However, its ability to do so depends on the type of the generated low-level code. For example, a major issue we discovered using OpenCL C is the lack of support for direct branches, which inhibits our ability to adequately support exceptions. Currently, Tornado is unable to create objects on accelerators and move them under the control of the memory manager inside the JVM or remove objects from under the control of the memory manager. To handle such cases, Tornado

Listing 4: Handling Library Calls in Tornado.

```
1 public static void testCos(double[] a) {
2   for (@Parallel int i = 0; i < a.length; i++) {
3     a[i] = Math.cos(a[i]);
4   }
5 }
```

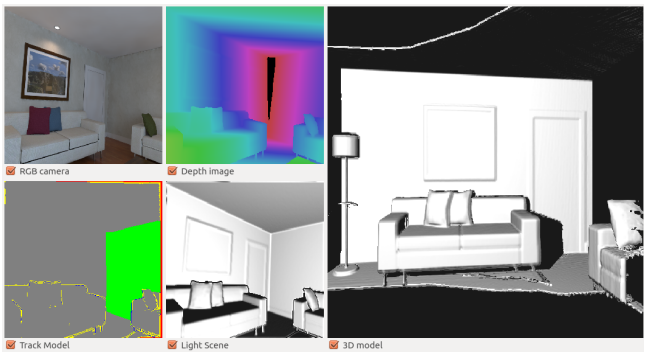


Figure 6: RGB-D camera combines RGB with Depth information to form a 3D reconstruction of a scene (right).

takes a practical approach where an attempt is made to compile all code and if it is not possible, execution will revert back to running the sequential Java inside the JVM.

3.4.1 Library Calls. Tornado also supports the invocation of library calls in Tornado tasks. Java libraries are usually implemented in Java itself. However, some libraries rely on native implementations, e.g., `java.util.concurrent`. For libraries implemented in Java, Tornado automatically inlines the library call using Graal (see Section 3.3). However, Tornado has no way to automatically handle invocations to library calls with native implementations. To support such library calls, the Tornado compiler relies on intrinsic substitution during compilation. These instrinsics need to be manually defined in the Tornado compiler. Currently, Tornado ships with pre-defined intrinsics for the `java.lang.Math` library. Other library calls to native code are currently unsupported. Listing 4 gives an example of a Tornado task invoking a method from the `java.lang.Math` library.

4 ACCELERATING THE KINECT FUSION COMPUTER VISION APPLICATION

To further stress Tornado and demonstrate its capability of running real applications, we accelerate end-to-end a complex Computer Vision (CV) application; namely, Kinect Fusion (KF) [36]. Another goal of this demonstration is to show that using Tornado we can: *1)* execute across as many devices as possible without requiring code modifications, and *2)* achieve high performance.

4.1 Kinect Fusion

Kinect Fusion [36] processes a stream of depth images, obtained by a RGB-D camera, and reconstructs a three-dimensional representation of the space (Figure 6). In order to achieve its quality

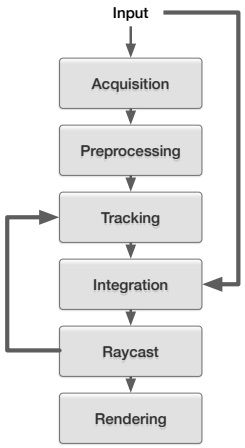


Figure 7: Kinect Fusion Pipeline stages

of service (QoS) target (i.e. real-time reconstruction of the environment) Kinect Fusion needs to operate at the frame-rate of the camera, which is 30 frames per second (FPS). Implementation-wise, some of Kinect Fusion's kernels are very large — about 250 lines of code — and utilize a much wider range of Java language features than other benchmarks that are typically used to evaluate the performance of heterogeneous programming frameworks.

Kinect Fusion comprises a six-stage processing pipeline (depicted in Figure 7) to process the input stream of depth images:

- **acquisition** obtains the next RGB-D frame - either from a camera or from a file.
- **pre-processing** applies a bilateral filter to remove anomalous values, re-scales the input data to represent distances in millimeters and builds a pyramid of vertex and normal maps using three different image resolutions.
- **tracking** estimates the difference in camera pose between frames. This is achieved by matching the incoming data to an internal model of the scene using a technique called Iterative Closest Point (ICP) [5, 46].
- **integrate** fuses the current frame into the internal model, if the tracking error is less than a predetermined threshold.
- **raycast** constructs a new reference point cloud from the internal representation of the scene.
- **rendering** uses the same ray-casting technique of the previous stage to produce a visualization of the 3D scene.

In SLAMBench each stage of the Kinect Fusion pipeline is composed from a series of kernels. Typically, a single frame will require the execution of 18 to 54 kernels. Therefore, to achieve the target frame-rate of 30 FPS, the application must sustain the execution of 540 to 1620 kernels per second.

4.2 Java & Tornado Implementation

A common characteristic of CV applications, regardless of the scenario in which they are used, is their extreme computational demands. Typically, they are written in programming languages such as C++ and OpenMP with binding extensions for OpenCL or CUDA execution. A common drawback of such implementations is the lack of portability since the application has to be recompiled and optimized for each underlying hardware platform. Building and optimizing CV applications on top of a managed runtime system such as the Java Virtual Machine (JVM) would enable single implementations to run across multiple devices such as desktop machines or low-power devices. To demonstrate this, we evaluate Kinect Fusion against a diverse set of heterogeneous hardware resources in Section 4.3.

Our Java reference implementation is derived from the open-source C++ version provided by SLAMBench [35]. During porting, we ensured that the Java implementation produces bit-exact results when compared to the C++ one.[1] This is highly important, and challenging, since Java does not support unsigned integers. Therefore, we had to modify the code to use signed representations and maintain correctness. Although all kernels produce near identical results during unit-testing, each implementation can produce slightly different results when combined together due to the nature of floating-point arithmetic.

We have developed the Java implementation with minimal dependencies on third-party code and we do not use any form of Foreign Function Interface (FFI) or native libraries. Our only dependency is on the EJML library [18] for its implementation of Singular Value Decomposition (SVD).

During preliminary performance analysis, we discovered that the C++ implementation is 3.4× faster than Java. Despite outperforming Java, the C++ implementation barely manages to achieve 4 FPS, which is much lower than the expected QoS target of 30 FPS. After the initial validation and performance analysis of our serial Java Kinect Fusion implementation, we ported Kinect Fusion to Tornado. To enable our baseline Java Kinect Fusion implementation to take advantage of Tornado's capabilities we: *1)* used the Tornado API to describe the processing pipeline, *2)* annotated loops that are safe to be executed in parallel with `@Parallel`, and *3)* executed the task-graph at an appropriate point in the application. The Tornado Kinect Fusion implementation has eight separate task graphs in total: one for each of the pre-processing, integrate, raycast and rendering stages, and four for the tracking stage — one to create the image pyramid and one to process each of three levels of the pyramid.

4.3 Kinect Fusion Evaluation

To evaluate our Tornado Kinect Fusion implementation, we use four distinct classes of heterogeneous systems (shown in Table 1). Each system has a multi-core processor, along with a minimum of one GPGPU that can be used for acceleration. To provide fair comparisons, all experiments use the same application configuration and scene from the ICL-NUIM data-set [25].

Table 2 provides the frame-rates we achieved during our experiments for all tested implementations. To better understand the results we provide three different Tornado implementations: Tornado-NR, Tornado-JR, and Tornado-OR. Tornado-NR (No Reduce) does not support reduction operations. Tornado-JR (Java Reduce) has limited support for reduction operations, which are written in pure Java. Tornado-OR (OpenCL Reduce) has fully supports reduction operations, but relies on hand-crafted OpenCL

[1] Even if this failed, it came within 5 Units of Last Place (ULP).

Machine Name	OS (kernel)	CPU	Cores	OpenCL	GPGPU	CU	OpenCL
Laptop	OSX 10.11.6 (14.5.0)	Intel i7-4850HQ @ 2.3 GHz	4 (8)	1.2 (Apple)	Intel Iris Pro 5200	40	1.2 (Apple)
					NVIDIA GT 750M @ 925 MHz	2	1.2 (Apple)
Desktop	Fedora 21 (4.1.10)	AMD A10-7850K @ 1.7 GHz	4	1.2 (AMD)	AMD Radeon R7 @ 720 MHz	8	2.0 (AMD)
Enterprise	CentOS 6.8 (2.6.32)	Intel Xeon E5-2620 @ 1.2 GHz	12 (24)	1.2 (Intel)	NVIDIA Tesla K20m @ 705 MHz	13	1.2 (NVIDIA)

Table 1: Hardware Configurations, CU: Number of OpenCL Compute Units.

Machine	Java	C++	OMP	OpenCL			Tornado-NR			Tornado-JR			Tornado-OR	
				CPU	GPU 1	GPU 2	CPU	GPU 1	GPU 2	CPU	GPU 1	GPU 2	GPU 1	GPU 2
Laptop	0.87	3.69	-	e	57.93	e	15.01	24.84	20.15	15.39	45.66	21.00	48.36	24.61
Desktop	0.40	3.14	7.87	e	e	-	5.28	16.80	-	7.91	15.51	-	21.78	-
Enterprise	0.71	2.40	19.63	29.02	138.10	-	21.60	31.25	-	30.65	52.26	-	107.78	-

Table 2: SLAMBench performance in FPS for each implementation (e: failed to produce a valid result)

kernels (for the reductions only). The three implementations are discussed in more detail later in the evaluation.

4.3.1 Measuring Performance and Accuracy. A challenge when comparing different implementations of Kinect Fusion, and CV algorithms in general, is that performance and accuracy measures are subjective. Normally, this is due to the real measure of the algorithmic quality being the user experience: does the user notice slow performance and is it accurate enough for their needs? Nevertheless, we must ensure that each implementation of Kinect Fusion does the same work and produces the same answer. Therefore, out of a number of Kinect Fusion implementations we have selected the ones provided by SLAMBench since they provide ready-made infrastructure to measure the performance and accuracy, enabling reliable comparisons between different implementations.

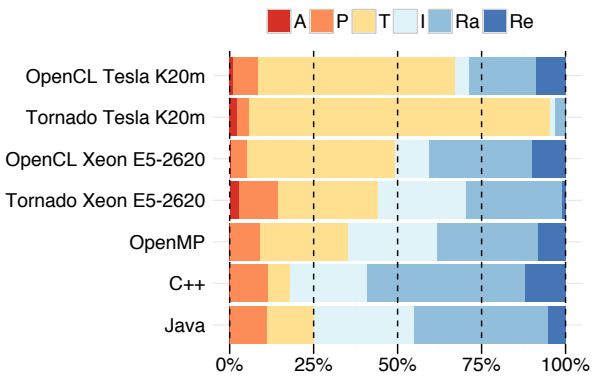
The accuracy of each reconstruction is determined by comparing the estimated trajectory of the camera against a provided ground truth, and is reported as an absolute trajectory error (ATE). The ground truths are provided by the synthetically generated ICL-NUIM data-set [25]. Finally, the performance is measured as the average frame-rate achieved when processing the entire data-set.

For a result to be considered valid, an implementation needs to return a mean Absolute Trajectory Error (ATE) of under 5 cm; as per the criteria set out by SLAMBench.

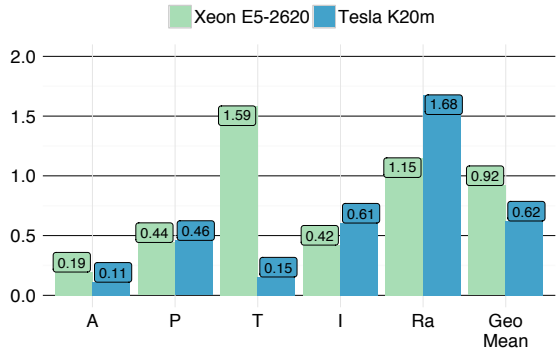
4.3.2 Portability. The first notable outcome of our experiments is that the OpenCL implementation produced valid results on only six devices, 60% of all devices, whereas Tornado produced valid results on all devices. This result strengthens our argument that Tornado with its dynamic JIT compilation is able to provide high performing heterogeneous implementations that are portable on a wide number or devices. On the contrary, the OpenCL implementation could not execute on all devices due to the assumptions made by the developers when they initially implemented SLAMBench on their device of choice. These assumptions regard work-group dimensions, and the amount of local memory available. If either of these assumptions are invalid on the target device, the reduce kernel fails to execute correctly. These problems are avoided in Tornado as resource usage is determined automatically by the runtime system and is based on the preferences of the target device. Additionally, Tornado provides developers with a number of runtime configuration options to influence how resources are allocated; meaning that a number of these issues can be corrected without re-compiling the application.

4.3.3 Performance Study. Inspecting the results in more detail, we observe that the baseline Tornado implementation (Tornado-NR) of SLAMBench achieves a speedup of 12-43× over the reference Java implementation and in one case it exceeds our minimum level of QoS at 31.25 FPS. Nevertheless, it produces 0.36× the performance of the OpenCL implementation. To understand where the performance loss occurs we ran a number of additional experiments with finer grained measurements. Figures 8a and 8b present the results. Figure 8a compares the amount of time spent in each pipeline stage, while Figure 8b compares the mean execution times of each Tornado pipeline stage against the OpenCL implementation on the Enterprise system. From these figures we see that: *1)* the total execution times of the parallel implementations in OpenMP, OpenCL, and Tornado, are dominated by the tracking stage, and *2)* Tornado achieves less than 0.15× the performance of the OpenCL implementation in the tracking stage. These observations indicate that the tracking stage is the performance bottleneck in our implementation.

Studying the tracking stage of our implementation we detected the issue to be related to data transfers between the device and the host. According to our measurements, the Tornado-NR implementation requires over 14MB of data to be moved between the device and host per frame. In the OpenCL version, this problem is addressed by using a hand-crafted reduction operation which compresses the size of the tracking result before transferring it back to the host. In our initial Tornado implementation (Tornado-NR), we chose not to port this kernel because it cannot be expressed in serial Java. The main issue preventing the reduction operation being written in Java, is the ability to express the movement of data between different threads and work-groups since such notions do not exist in the language. Note also that providing a hand-crafted

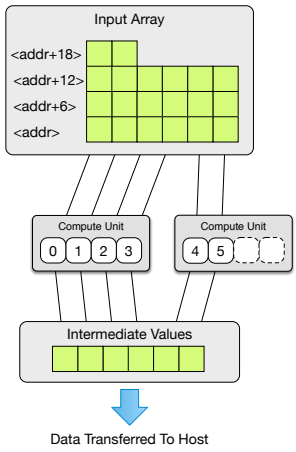


(a) Breakdown of time spent in each stage of the pipeline.

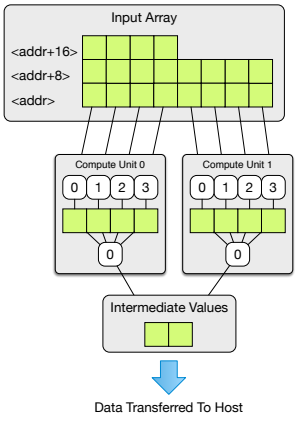


(b) Tornado execution times normalized to the OpenCL equivalent (lower is better).

Figure 8: Performance breakdown on the Enterprise system. (A: acquisition, P: pre-processing, T: tracking, I: integration, Ra: raycast, Re: rendering)



(a) Java Based Reduction



(b) OpenCL Based Reduction

Figure 9: Illustration of the different reduction algorithms.

implementation would negatively impact the portability of our implementation.

Nevertheless, since the tracking stage has become a performance bottleneck we revised two solutions. The first solution was to implement a reduction function, in Java, that does not use inter-thread communication (Tornado-JR), while the second was to provide Tornado with a hand-crafted OpenCL C kernel that does (Tornado-OR). The advantage of the first approach is that the code remains portable across all devices by sacrificing performance, whereas the second approach yields better performance but sacrifices portability.

Figures 9a and 9b illustrate the reduction operations implemented in Java and OpenCL respectively (Tornado-JR and Tornado-OR). The Java reduction uses a fixed number of threads to combine results in a thread-cyclic manner creating one partial result per used thread. However, the inability to communicate data between threads means that we cannot fully utilize the hardware (dashed boxes). The OpenCL implementation, similarly to the Java one, is a multi-stage reduction and it is able to utilize more threads by exploiting inter-thread/intra-work group communication. The most important difference, to the Java one, is that an extra reduction is performed inside each work-group which results in a single value being produced per work-group. We have also added the ability to change the number of work-groups used in the reduction varying the utilization of the GPGPUs compute units. It is important to note Tornado's ability to allow developers to add their user-defined reduction kernels while hand-optimizing their implementations if performance becomes an issue.

4.3.4 Performance Improvements. To evaluate the impact of our different reduction kernels we repeated our experiments using a number of configurations. The Tornado-JR kernel can be configured at run-time to use different numbers of threads - we used values of: 512, 1024, 2048, and 4096 on the GPGPUs; and 0.5, 1, 2, and 4× the number of available compute units on the CPUs. The Tornado-OR kernel is implemented to dynamically adjust the work-group size and number of work-groups it uses to help us vary the utilization of the compute-units. By default, we assign a single work-group with the largest possible dimensions onto each compute unit. As shown in Table 2 Tornado's performance has improved in both Tornado-JR and Tornado-OR configurations compared to the baseline one (Tornado-NR).

As shown in Figure 10, the Tornado implementations are obtaining significantly higher levels of performance. Regarding the JR experiments, we observe performance improvements across all devices with a maximum speedup of 74× over the reference implementation and a maximum frame-rate of 52 FPS. More importantly, we see that three devices are now able to exceed our QoS threshold of 30 FPS. This means that we have been able to exceed the QoS threshold on the same devices as OpenCL by using an implementation written entirely in Java. Moreover, if we compare these results to OpenMP we see that although we started from a performance point of 3-7× lower than C++, our Tornado implementation is able to achieve higher performance on all CPU implementations. By

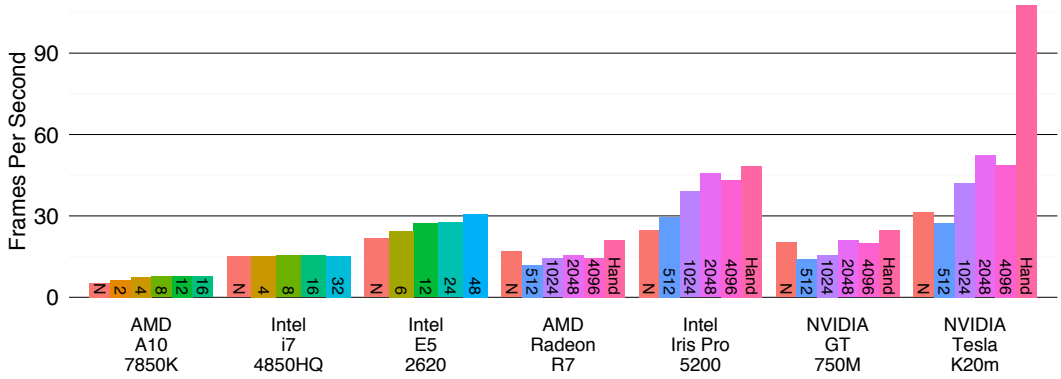


Figure 10: Performance in FPS after implementing the reduce kernel (N: Tornado-NR, Hand: Tornado-OR, Rest: Tornado-JR with variable number of threads).

using the OR version, we have managed to obtain the highest performance with a maximum speedup of 150× over the reference implementation and a maximum frame-rate of 107 FPS on the Tesla K20m.

Finally, regarding the comparison with the OpenCL implementation, Tornado achieves 0.59× the performance of the OpenCL implementation by using only Java (Tornado-JR), and if a developer wishes to sacrifice a little portability by using a single hand-crafted OpenCL kernel this rises to 0.77×.

4.3.5 Host Side Performance. Figure 11a shows the average compilation times of all tasks across all devices. In general, we observe that compilation takes between 100-200 milliseconds and is split, almost evenly, between the Graal and the OpenCL compiler. Tornado provides the ability to manually trigger the optimization of task-graphs and the JIT compilation of tasks before a task-schedule is executed; this way the corresponding overheads can be removed from time-sensitive task-schedules — such as the ones in Kinect Fusion. Using this option mirrors how the compilation overheads are handled in the OpenCL of Kinect Fusion.

Figure 11b shows the time spent to process each frame during execution for the Tornado JR (2048), OR (Hand), and OpenCL implementation on the Enterprise system. During the early stages of the benchmark, all Tornado implementations experience extra overheads from JIT compilation and garbage collection. However, performance stabilizes after approximately 100 frames and continues to improve. After profiling, we discovered that the memory usage of our Tornado implementations stabilizes at around 400 MB resulting in minimal GC interference after the warmup period.

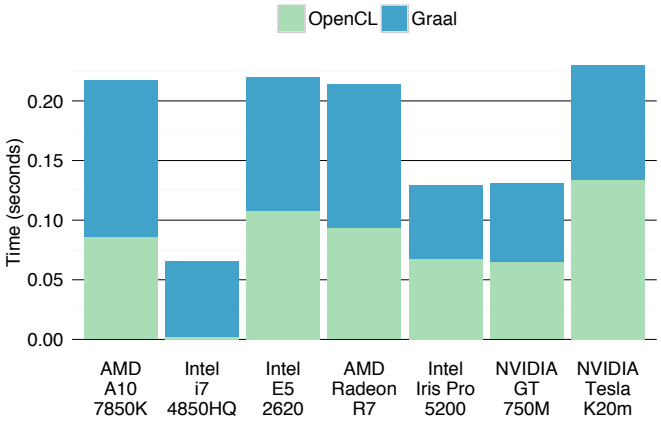
5 CONCLUSIONS AND FUTURE WORK

We demonstrate that through holistic design it is possible to develop a practical heterogeneous programming framework. The distinguishing feature of Tornado is that it enables developers to write portable heterogeneous code in pure Java. This allows them to write applications that can be quickly deployed across different hardware accelerators and operating systems. Moreover, our dynamic design allows them to avoid making a priori decisions — instead applications can be dynamically configured at run-time.

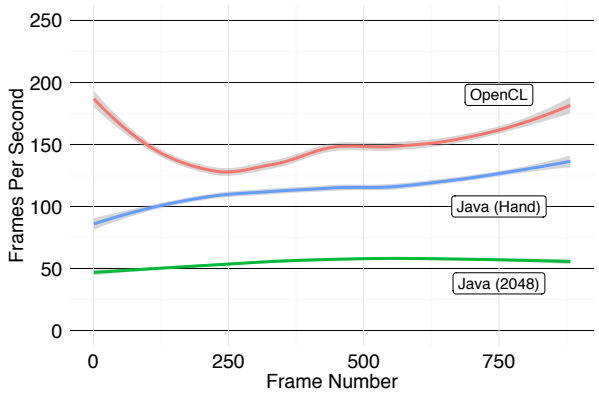
We demonstrate that by using Tornado it is possible to write a single implementation of a complex Computer Vision application and deploy it across a variety of heterogeneous systems, while maintaining comparable performance to hand-crafted optimized equivalents. What makes Tornado unique is that it has been developed to provide heterogeneous programming support to the general purpose Java programming language, a language that would not normally be associated with writing either high-performance or heterogeneous code. By utilizing the introduced Tornado framework, we managed to obtain speedups of 18-150× over the reference Java implementation. The results show that not only can we obtain levels of performance that meet our QoS target of 30 FPS, but we can also exceed our target by up to 3× (at 107 FPS). Moreover, we demonstrate that Tornado is able to utilize 2.3× more devices than the hand-written OpenCL.

Experiences with OpenCL. Although Tornado is able to execute complex Java applications across a wide range of accelerators successfully, a number of key issues remain. Tornado is currently based on OpenCL. More desirable options such as CUDA/PTX or even HSA/HSAIL would have severely restricted the diversity and number of accelerators that we could use. Aside from issues regarding Java/OpenCL compatibility, we struggled to find an OpenCL-compatible way to create on-device managed heaps and subsequently objects. The problems arise from the fact that we are being forced to access device memory via OpenCL buffers while having no standard way of resolving device-side addresses: either relative or absolute. Advanced OpenCL features, like Shared Virtual Memory, could not help solving our problems as this is not, yet, supported on the majority of devices we used. The main reason is that vendors' support for OpenCL is variable across operating systems and devices — sometimes getting access to an SDK is near impossible.

Future work. As future work, we aim to further improve Tornado's performance by implementing more compiler and runtime optimizations, such as control-flow minimization, predication, use of constant memory, and compressed object layouts for objects residing on the hardware accelerators. We also plan to enable new



(a) Average per-task compilation times for each device.



(b) Per-frame performance on NVIDIA Tesla K20m.

Figure 11: Compilation times and per-frame performance. Key: Hand - OR implementation, 2048 - JR with 2048 threads.

object allocation from tasks running on devices other than that running the JVM host. Furthermore, we plan to extend Tornado's reach to more devices and diverse accelerators such as Intel's Xeon Phi and FPGAs. We also plan to open-source and release the complete Tornado JIT compiler and the task-based API in Github, under the beehive-lab github-organization of the University of Manchester (https://github.com/beehive-lab/).

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback and the effort they put to review this manuscript. This work is partially supported by the EPSRC grants PAMELA EP/K008730/1 and AnyScale Apps EP/L000725/1, and the EU Horizon 2020 E2Data 780245.

REFERENCES

[1] 2015. Project Beehive: A Hardware/Software Co-designed Stack for Runtime and Architectural Research. *CoRR* abs/1509.04085 (2015). arXiv:1509.04085 http://arxiv.org/abs/1509.04085 Withdrawn.

[2] AMD. 2016. Aparapi. Retrieved December 19, 2018 from http://developer.amd.com/tools-and-sdks/heterogeneous-computing/aparapi

[3] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. 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)*. ACM, New York, NY, USA, 89–108. https://doi.org/10.1145/1869459.1869469

[4] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. 3–10.

[5] P. J. Besl and H. D. McKay. 1992. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14, 2 (Feb 1992), 239–256.

[6] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. 2011. Copperhead: Compiling an Embedded Data Parallel Language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 47–56. https://doi.org/10.1145/1941553.1941562

[7] Olivier Chafik. 2015. ScalaCL: Faster Scala: optimizing compiler plugin + GPU-based collections (OpenCL). Retrieved December 19, 2018 from https://github.com/nativelibs4java/ScalaCL

[8] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming (DAMP '11)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

[9] James Clarkson, Christos Kotselidis, Gavin Brown, and Mikel Luján. 2017. Boosting Java Performance Using GPGPUs. In *Architecture of Computing Systems - ARCS 2017*, Jens Knoop, Wolfgang Karl, Martin Schulz, Koji Inoue, and Thilo Pionteck (Eds.). Springer International Publishing, Cham, 59–70.

[10] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

[11] NVIDIA Corporation. 2017. CUDA. Retrieved December 19, 2018 from http://developer.nvidia.com/cuda-zone

[12] NVIDIA Corporation. 2017. Parallel Thread Execution ISA Version 4.0. Retrieved December 19, 2018 from https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[13] Georg Dotzler, Ronald Veldema, and Michael Klemm. 2010. JCudaMP. In *Proceedings of the 3rd International Workshop on Multicore Software Engineering*. 10–17. https://doi.org/10.1145/1808954.1808959

[14] Christophe Dubach, Perry Cheng, Rodric Rabbah, David Bacon, and Stephen Fink. 2012. Compiling a High-Level Language for GPUs (via Language Support for Architectures and Compilers). In *Proceedings of the 33rd ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*. 1–12. papers/dubach12pldi.pdf

[15] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. 2012. 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)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2254064.2254066

[16] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. 2013. Graal IR: An extensible declarative intermediate representation. In *Asia-Pacific Programming Languages and Compilers*.

[17] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. 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)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[18] EJML. 2017. EJML. Retrieved December 19, 2018 from http://ejml.org

[19] HSA Foundation. 2016. HSA Foundation. Retrieved December 19, 2018 from http://www.hsafoundation.com

[20] HSA Foundation. 2017. HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG) .95. Retrieved December 19, 2018 from https://hsafoundation.box.com/s/m6mrsjv8b7r50kqeyyalg

[21] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. 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)*. ACM, New York, NY, USA, 60–73. https://doi.org/10.1145/3050748.3050761

[22] Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. 2015. Runtime Code Generation and Data Management for Heterogeneous Computing in Java. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. ACM, New York, NY, USA, 16–26. https://doi.org/10.1145/2807426.2807428

[23] Juan José Fumero, Michel Steuwer, and Christophe Dubach. 2014. 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)*. ACM, New York, NY, USA, 44:44–44:49. https://doi.org/10.1145/2627373.2627381

[24] Khronos Group. 2017. OpenCL. Retrieved December 19, 2018 from https://www.khronos.org/opencl

[25] A. Handa, T. Whelan, J.B. McDonald, and A.J. Davison. 2014. A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM. In *IEEE Intl. Conf. on Robotics and Automation, ICRA*. Hong Kong, China, 1524–1531.

[26] Akihiro Hayashi, Max Grossman, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2013. 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 (PPPJ '13)*. ACM, New York, NY, USA, 124–134. https://doi.org/10.1145/2500828.2500840

[27] Akihiro Hayashi, Max Grossman, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2013. 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*. 124–134. https://doi.org/10.1145/2500828.2500840

[28] Sylvain Henry. 2013. 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)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/2502323.2502329

[29] Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. 2013. 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)*. ACM, New York, NY, USA, 729–744. https://doi.org/10.1145/2509136.2509516

[30] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. 2015. Compiling and Optimizing Java 8 Programs for GPU Execution. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 419–431. https://doi.org/10.1109/PACT.2015.46

[31] JOCL. 2017. Java bindings for OpenCL. Retrieved December 19, 2018 from http://www.jocl.org/

[32] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation. *Parallel Comput.* 38, 3 (March 2012), 157–174.

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

[34] Geoffrey Mainland and Greg Morrisett. 2010. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/1863523.1863533

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

[36] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. 2011. KinectFusion: Real-time Dense Surface Mapping and Tracking. In *Proceedings of the 2011 10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR '11)*. IEEE Computer Society, Washington, DC, USA, 127–136. https://doi.org/10.1109/ISMAR.2011.6092378

[37] Nathaniel Nystrom, Derek White, and Kishen Das. 2011. Firepile: Run-time Compilation for GPUs in Scala. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE '11)*. ACM, New York, NY, USA, 107–116. https://doi.org/10.1145/2047862.2047883

[38] OpenACC.org. 2017. OpenAcc: Directives for Accelerators. Retrieved December 19, 2018 from http://www.openacc-standard.org

[39] OpenJDK. 2017. OpenMP. Retrieved December 19, 2018 from http://openjdk.java.net/projects/sumatra

[40] P.C. Pratt-Szeliga, J.W. Fawcett, and R.D. Welch. 2012. Rootbeer: Seamlessly Using GPUs from Java. In *Proceedings of 14th International IEEE High Performance Computing and Communication Conference on Embedded Software and Systems*. https://doi.org/10.1109/HPCC.2012.57

[41] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. 2012. Parakeet: A Just-in-time Parallel Accelerator for Python. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12)*. USENIX Association, Berkeley, CA, USA, 14–14.

[42] Matthias Springer, Peter Wauligmann, and Hidehiko Masuhara. 2017. Modular Array-based GPU Computing in a Dynamically-typed Language. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2017)*. ACM, New York, NY, USA, 48–55. https://doi.org/10.1145/3091966.3091974

[43] Yonghong Yan, Max Grossman, and Vivek Sarkar. 2009. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Euro-Par 2009 Parallel Processing*, Henk Sips, Dick Epema, and Hai-Xiang Lin (Eds.), Vol. 5704. Springer Berlin Heidelberg.

[44] Foivos S. Zakkak, Andy Nisbet, John Mawer, Tim Hartley, Nikos Foutris, Orion Papadakis, Andreas Andronikakis, Iain Apreotesei, and Christos Kotselidis. 2018. On the Future of Research VMs: A Hardware/Software Perspective. In *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming (Programming'18 Companion)*. ACM, New York, NY, USA, 51–53. https://doi.org/10.1145/3191697.3191729

[45] Wojciech Zaremba, Yuan Lin, and Vinod Grover. 2012. 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)*. ACM, New York, NY, USA, 74–83. https://doi.org/10.1145/2159430.2159439

[46] Zhengyou Zhang. 1994. Iterative Point Matching for Registration of Free-form Curves and Surfaces. *Int. J. Comput. Vision* 13, 2 (Oct. 1994), 119–152.

View publication stats