Tornado: A Practical And Efficient Heterogeneous Programming Framework For Managed Languages

JAMES CLARKSON, University of Manchester
CHRISTOS KOTSELIDIS, University of Manchester

Hardware accelerators offer developers the ability to improve the performance and energy efficiency of their applications. However, a key barrier that is preventing their widespread adoption are the shortcomings of existing tools for heterogeneous programming — they are not designed to handle the complexity of deploying an application into an environment where the availability and type of hardware accelerators are unknown or cater for developers that use managed languages to process “Big Data”.

This paper describes our experiences creating Tornado: a practical and efficient heterogeneous programming framework for managed languages. The novel aspect of Tornado is that it turns the programming of heterogeneous systems from an activity predominantly based on a priori knowledge into one based on a posteriori knowledge. Alternatively put, it simply means developers do not need to overcomplicate their code by catering for all possible eventualities. Instead, Tornado provides the ability to specialize each application for a specific system in situ which avoids the need for it to be pre-configured by the developer. To enable this, Tornado employs a sophisticated runtime system that can dynamically configure all aspects of the application — from selecting which parallelization scheme to apply to specifying which accelerators to use. By using this ability, the end-user, and not the developer, can transparently make use of any available multi- or many-core processor and accelerators.

To showcase the impact of Tornado, we implement a real-world computer vision application and deploy it across nine accelerators without having to modify the source code or even explicitly re-compile the application. Using dynamic configuration, we show that our implementation can achieve up to 124 frames per second (FPS) - up to 166× speedup over the reference implementation. Finally, our implementation is always within 21% of a hand-written OpenCL version but avoids much of the programming tedium.

Keywords: Java, GPGPU

1 INTRODUCTION

The appearance of hardware accelerators in 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 that developers might wish to utilize them, especially if this leads to improvements in performance and energy efficiency. However, to develop effective programming languages for this heterogeneous hardware, we need to invalidate one long-standing assumption — that all applications execute exclusively on homogeneous systems.

To this end, programming languages that enable the programming of systems containing hardware accelerators, like GPGPUs, have emerged. Hereafter, we refer to these as heterogeneous programming languages and heterogeneous systems respectively. The most popular heterogeneous programming languages like CUDA [20], OpenCL [16], and OpenACC [23] exist because of the necessity to efficiently program GPGPUs. Consequently, they all 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 GPU.

This work is supported by the AnyScale Apps and PAMELA projects funded by EPSRC EP/L000725/1 and EP/K008730/1. 2018. This is the author’s version of the work. It is posted here for your personal use.

Publication date: January 2016.
The problem with these languages is that they are geared towards the programming of systems that have both a known configuration and a low-degree of heterogeneity — they only contain a single type of hardware accelerator, for example, NVIDIA GPGPUs.

In this paper we describe the **Tornado** heterogeneous programming framework that can be used to program heterogeneous systems that have an unknown, or changing, configuration. Furthermore, Tornado is geared towards systems that exhibit a higher degree of heterogeneity; a common situation when executing applications in public clouds. In these systems, developers are offered more than a binary choice between accelerated or not, making it tougher to decide how to construct the application as they may need to manage three or more different sets of choices. Tornado solves this problem by eliminating the need to make assumptions about the system ahead-of-time. Instead, we leverage the dynamic compilation infrastructure of a managed language, Java, to allow the application to be dynamically configured at runtime\(^1\).

Tornado is an open-source project that, currently, allows developers to utilize any OpenCL compatible device — such as CPUs, GPGPUs, FPGAs, and other hardware accelerators like the Intel Xeon Phi. At its core is a sophisticated runtime system that provides developers with the necessary support to construct complex multi-stage processing pipelines that are portable across hardware accelerators. To showcase Tornado’s maturity and ability to execute complex real-world applications, we detail our experiences accelerating a Java version of the Kinect Fusion (KF) application: an application that is typically beyond the computational capability of non-hardware accelerated implementations. In detail, this paper makes the following contributions:

- It provides a small survey of state-of-the-art heterogeneous programming languages analyzing their main advantages and disadvantages.
- Introduces **Tornado**: a Java based state-of-the-art heterogeneous programming framework enabling the easy implementation of portable and high-performing heterogeneous applications.
- Showcases that with the proposed technology we can deploy a real-world Computer Vision application across nine OpenCL compatible devices and achieve a speedup of up to 166× over our reference implementation. Moreover, we demonstrate that our implementation obtains performance comparable to hand-written OpenCL — a maximum slowdown of 21% — but without the programming tedium.
- Demonstrates the value of dynamic configuration by applying a series of optimizations, without altering the application, achieving significant speedups of up to 14.8× over hand-written OpenCL.

2 HETEROGENEOUS PROGRAMMING

Heterogeneous programming languages provide control over where a computation is to execute — a feature that is distinct from traditional languages which assume the whole application is to execute on a single processor. Nearly all of the issues that emerge when trying to simplify these languages fall into two categories: enabling developers to code in a hardware agnostic manner and clearly separating the logic that defines the computation from the logic that coordinates execution — separating the **what** from the **where**. In the remainder of this section, we provide a taxonomy that can be used to categorize languages for programming heterogeneous systems and highlight their strengths and weaknesses.

\(^1\)Although we use Java as the programming language of choice, the techniques described in this paper are applicable to all managed languages supporting dynamic compilation.
2.1 Classifying Languages For Heterogeneous Programming

In our experience, the greatest difference between the various languages is the sophistication of their three principle components: the language or API, the runtime system, and the compilation infrastructure. To compare languages fairly we must consider the design and capability of all these three components collectively and not in isolation.

Language Design. Typically, language implementers prefer to augment existing languages such as C/C++ opposed to creating one from scratch. By doing this, they significantly reduce the implementation effort and enjoy the benefits of having a ready-made pool of developers. However, the downside is that existing languages come with baggage that can hinder code generation on some heterogeneous architectures. A good example of this is supporting polymorphic method calls in managed languages. For this reason, there is a real lack of support for programming hardware accelerators from within managed languages and most prior art opts to severely restrict the languages features that can be used on accelerator; sometimes to the point where very few language features are available.

In terms of productivity, we highlight ten key concepts in Table 1 that developers often need to grasp in order to write applications for heterogeneous systems. We find that the more of these concepts are needed, the complexity of coding heterogeneous applications increases. Therefore, we use the number of new concepts required to use a particular language a good indicator of productivity. In reality, it is undesirable to try to eliminate these concepts from a language as this leads to oversimplification and restricts the applicability of the language. Instead, our experience is that to improve productivity we should try to minimize the depth of knowledge required for a new developer to learn the system but also allow more experienced developers the ability to apply their domain-specific knowledge.

Runtime System. The runtime system is the part of the language implementation that helps to abstract away low-level implementation details from the developer. A well designed and integrated runtime system is key to supporting features that increase programmer productivity. A prime example of this is the Java Virtual Machine (JVM) where developers do not need to worry about memory management as this is done automatically. An example of the lack of runtime system is in OpenCL that places extra burden on developers to manage low-level interactions with devices, such as compiling code, initiating data transfers and launching kernels. However, even productive languages like OpenACC are hampered by limited runtime support as the application often needs to be recompiled each time a compiler directive needs to be modified.

| Key | Concept                | Description                                                                 |
|-----|------------------------|-----------------------------------------------------------------------------|
| DS  | Data Structures        | Learning how to use a new data structure and API.                           |
| DP  | Data Parallelism       | Learning a new parallel execution model, terminology and API.                |
| TP  | Task Parallelism       | Learning the terminology and API to manage asynchronous execution.           |
| DE  | Distributed Execution  | Learning how application state needs to be managed across devices.           |
| MM  | Memory Management      | Learning how to use the memory that is attached to a remote device.         |
| DM  | Device Management      | Learning how to find and transfer data between devices.                     |
| NL  | New Language           | Learning a new programming language to write code for a remote device.      |
| CG  | Code Generation        | Learning how to generate code for each device.                              |
| DA  | Device Architecture    | Learning the best way to write code for a particular device.                |
| RO  | Runtime Optimizations  | Learning how to write optimizations to minimize runtime overheads.          |

Table 1. Common concepts introduced by heterogeneous programming languages.
In the context of programming heterogeneous systems, there are two aspects of runtime systems we need to consider: how to construct a runtime system that provides the best support for programming heterogeneous systems, and how to support existing language features that require runtime support to implement. What we have found is that the former is more suitable for differentiating languages in the general case and the latter is the best for differentiating the state-of-the-art with respect to managed languages. Primarily, this is because without advanced language features the code ends up being written using a subset of the language that is akin to C or OpenCL C. This is self-defeating as in many cases it nullifies the reason why developers chose that particular language in the first place.

Compilation Infrastructure. A desirable goal of a language for programming heterogeneous systems is to have the application written in a single language and executed across a broad range of heterogeneous processor architectures. Implicitly, this requires the source code to be compiled into a diverse range of machine languages. The key difference over a traditional programming language is that a program now compiles into a series of smaller binaries — each expressed in a different machine language — opposed to a single binary. The two options for implementing the compilation infrastructure are: statically Ahead-Of-Time (AOT) or dynamically Just-In-Time (JIT).

2.2 Analysis of Existing Solutions

As shown in Table 2, there is a plethora of prior art involved with heterogeneous programming. Currently, most of the focus is on improving the four more established and community-driven languages: CUDA, OpenACC, OpenCL, and OpenMP. Unfortunately, this excludes the plethora of JVM-based languages which are being used in the Big Data arena. Whether developing desktop applications or Big Data cloud-deployed applications, JVM support for heterogeneous programming is imperative to increase performance and energy efficiency. Presently, there are no JVM-based solutions capable of either competing in performance with CUDA or OpenCL nor providing a seamless programming interface to Java developers. The reason behind that is the complexity that heterogeneous programming brings into JVM implementations and functionalities enabled by them — e.g. polymorphic calls, exception handling, memory management and dynamic deoptimization.

| Language   | Hosting Language | Compiler | Devices                        |
|------------|------------------|----------|--------------------------------|
| CUDA       | C/C++            | AOT      | NVIDIA GPGPUs                  |
| OpenACC    | C/C++/FORTRAN    | AOT      | Multi-core and Many-core       |
| OpenCL     | C/C++/FORTRAN    | AOT      | Multi-core, Many-core          |
| River Trail| Javascript       | JIT      | Multi-core, Many-core          |
| SkelCL     | C++              | AOT      | Multi-core, Many-core          |
| Lime       | Lime             | JIT      | CPU, GPU, FPGA                  |
| Firepile   | Scala            | JIT      | OpenCL GPGPUs                  |
| FastR-OCL  | R                | JIT      | OpenCL GPGPUs                  |
| JOCL       | Java             | JIT      | OpenCL Compatible              |
| RootBeer   | Java             | AOT      | NVIDIA GPGPUs                  |
| APARAPI    | Java             | JIT      | Multi-core, Many-core          |
| IBM J9     | Java             | JIT      | NVIDIA GPGPUs                  |
| Sumatra    | Java             | JIT      | NVIDIA GPGPUs, HSA Compatible  |
| Tornado    | Java             | JIT      | OpenCL Compatible              |

Table 2. Existing Languages For Programming Heterogeneous Systems.
Table 3. Heterogeneous Languages ranked by productivity (left) and VM-based languages ranked by language capability (right). Abbreviations: LP (limited purpose), T (template-style compilation), RH (runtime support for heterogeneous execution), RE (runtime support for existing language features), RO (runtime optimizations for multi-kernel applications).

Language Design. As already mentioned, the productivity of a language is inversely proportional to how many programming concepts a developer has to learn and understand. In Table 3 we have ranked the languages, from Table 1, according to how many concepts are needed to operate them. We split the concepts into two groups: required — for the necessary concepts used to create a simple application, and optional — for the concepts that allow expert developers to apply their domain-specific knowledge. By ranking languages in this way, we highlight that the more productive a language is, the more restrictive it becomes. For example, the top-ranked languages only have the ability to apply a fixed number of functional operators to a specially designed data structure. Here developers are more productive because they only need to learn an API while every other aspect of heterogeneous programming is transparent to them. Such an approach suffers from oversimplification, since there is no way for the developer to choose which device to use, how many to use, or the how to parallelize the code. The challenge with heterogeneous programming is that these choices often need to be made for each instance of a task (or kernel) running on each device. This leads to a configuration complexity of \(O(n^2)\) — i.e. \(\text{tasks} \times \text{devices}\). For applications that are expected to run a few tasks on few devices this is not problematic, however, when the task count or the device count rises the inability to configure the application becomes restrictive.

Conversely, the least productive languages suffer from overcomplication, since there are far too many options for a developer to comprehend and little language support on offer. An example of this is OpenCL that suffers from high code verbosity because it forces developers to manually handle all aspects of heterogeneous programming. For instance, finding the device, allocating memory, transferring data and invoking a compiler all need to be completed in order to launch a kernel. A lot of this tedium can be designed away via both the API and improved runtime support.

We believe that a number of languages, such as OpenACC, OpenMP, and Tornado (Table 3), provide a sweet-spot where a lot of the tedium — like code generation, data movement, and parallelization — is taken care of automatically for the developer. As a result, the number of concepts needed to be understood by the developer is significantly reduced. For example, these languages do not require developers to perform parallelization manually; instead, their compilation infrastructure can apply different parallelization schemes to the application source. Moreover, these languages can be used to implement custom data structures and APIs to increase productivity.
**Runtime System.** One of the key differentiators between languages is the sophistication of their runtime system. Statically compiled languages like CUDA, OpenAcc, OpenMP, and OpenCL do not require complex runtime systems and, therefore, they do not provide significant runtime support. These languages perform optimization at compile time which means that the developer is required to specify information like the type of device ahead of time. This design is ideal for environments that demand ultimate performance in a stable environment (e.g. in HPC). However, this design also makes these languages unsuitable for computing in highly volatile environments. For example, if the application is running in a public cloud (e.g. in the context of Big Data) it may need to execute across a variety of different system configurations — some with and without accelerators. Alternatively, in the mobile market which has a short cadence between hardware refreshes an application is very likely to encounter new and unknown devices regularly. In such a situation it would be inviable to re-compile and re-tune the application as this would require the developers to additionally distribute their source code.

Table 3 ranks the runtime capabilities of the VM-based languages showing that the majority of them only try to automate the mechanics of heterogeneous programming — compilation, data movement and kernel launches. Their main differentiating factor is their support for their hosting language and the degree of runtime optimizations they provide. As shown, the majority of prior art only supports a subset of the hosting language. For example, IBM’s J9 JVM only supports operations on arrays of primitive types and it is quite uncommon for prior art to support the use of objects or even object creation. The result is that these languages become less useful for developers as significant parts of the language are unusable. The effect is observable as very few of these managed languages are evaluated using real-world applications that contain many tasks or use object-oriented language features.

A number of languages provide runtime optimizations, like memory alignment and high-performance data serialization, however, these optimizations are focused on improving the performance of single task applications. Tornado is the only framework to offer additional runtime optimizations to improve the composability of applications that require multi-stage processing pipelines. For example, it can eliminate unnecessary data movement between subsequent tasks automatically and exploit task parallelism between data transfers and compute kernels.

**Compilation Infrastructure.** Traditionally, AOT compilation is preferred as most languages are built on top of existing AOT compiled languages like C and C++. Unfortunately, AOT compiled languages do not provide a full coverage of the computing needs nowadays. Big Data software stacks or Android powered devices rely on managed languages. It is in these setups that heterogeneous computing can have a significant impact in both performance and energy efficiency. What makes AOT compilation and languages unsuitable for these environments is the diversity of the hardware resources involved. Typically, this means that developers need to decide ahead of time which devices will be available at runtime and if a new device is encountered or a different optimization needs applying, a full recompilation is required. These issues are alleviated using a JIT-based approach — deferring compilation until runtime when the specific hardware is known — thus eliminating the need to explicitly recompile applications.

### 2.3 Tornado Innovations

**Avoiding Oversimplification.** A well-designed language needs to avoid unnecessarily removing or restricting the developer’s ability to influence certain aspects of heterogeneous programming. In the prior art, we see that a lot of productive languages eliminate the developer’s ability to coordinate heterogeneous execution. For example, they do not support specifying where or when a computation should occur. To overcome this, Tornado has been designed so that developers can
easily co-schedule multiple tasks using task-schedules (Section 3.3). Moreover, we introduce the concept of dynamic configuration, in which every aspect of the task-schedule is configurable at runtime — from the device to use to the parallelization scheme to apply (Section 3.4).

Avoiding Overcomplication. Tornado has a sophisticated runtime system that can provide support for all aspects of heterogeneous programming. For example, it has a JIT compiler capable of dynamically generating data-parallel code (Section 3.6). The use of managed device memory allows transparent data allocation and movement between devices (Section 3.5). By using task-schedules, the runtime system can automatically eliminate unnecessary transfers and exploit task-parallelism between data movement and computation tasks (Section 3.3).

Increasing Language Support. A trend in the prior art is that managed languages try to avoid supporting complex features of programming languages. The result of this is that applications can only be written using a subset of the language, often akin to OpenCL C. Restricting the language in this way is self-defeating because it makes many productive features unavailable — like the use of object-orientation — which makes it harder to build complex applications. Tornado provides developers with the ability to use a larger subset of the Java language than the prior art (Section 3.7). It is differentiable from the prior art because it enables the construction of real-world applications written naturally in Java (Section 3.1).

Composing Processing Pipelines. Executing real-world applications requires the ability to execute multiple tasks on heterogeneous systems. Tornado is the only system that focuses on simplifying the construction of complex processing pipelines. It uses task-schedules to capture inter-task dependencies so that it can automatically eliminate unnecessary data transfers and overlap the execution of both data movement and computation tasks (Section 3.3). What distinguishes Tornado from other frameworks is that each element of a task-schedule can be dynamically configured on the command line to improve performance (Section 3.4).

Handling Volatile Environments. One of the principle features of Tornado is that it is entirely dynamic: both compilation and parallelization happen at runtime. These features make Tornado unique by allowing applications to become portable across many different system configurations without requiring recompilation. This feature is essential in environments where it is not possible to guarantee a specific system configuration: something that can happen if running in a public cloud or on mobile devices. We demonstrate this, by deploying a real-world application across different systems without recompilation (Section 4).

3 TORNADO
Tornado is a heterogeneous programming framework designed to augment the Java programming language in a practical and high-performing manner. Utilizing the inherent portability of the JVM, we are able to address the majority of shortcomings found in state-of-the-art heterogeneous programming languages described in the previous Section. The dynamism of Tornado allows developers to make a posteriori decisions enabling high-performing optimizations to be applied at runtime. This is of particular interest in the embedded domain where power constraints may dictate a particular accelerator to be shut down for power saving purposes. Tornado, through its JIT compilation infrastructure and full knowledge of the system topology, is able to dynamically compile code and target the most desirable hardware resource without the need of ahead-of-time selection. The following subsections describe in detail the various components of the Tornado heterogeneous programming framework. Furthermore, throughout this section we will use code examples from the Kinect Fusion application (Section 4.1) used in our evaluation.
3.1 Tornado API

One of the key design principles of Tornado is simplicity. For this reason, it provides a minimal API that is designed to be easily incorporated into existing code with minimal refactoring. The simplicity of the API is enabled by the sophistication of the runtime system: firstly, Tornado works with Java’s functional interfaces which allows developers to create heterogeneous code from existing methods or lambda functions; and secondly, the runtime system and compiler are co-designed so that Tornado does not require developers to provide information that is easily derivable via the compiler.

To simplify the composition of complex heterogeneous codes, Tornado employs a task-based programming model. In this model a task is simply code that can be executed on a hardware accelerator and encapsulates: the code to execute, the data it should operate on, and some metadata that contains both the compiler and runtime configurations for the task. A task is an instance of data-parallel code that is analogous with an OpenCL kernel. Tornado uses tasks as the basic unit of execution and developers are free to configure tasks on an individual basis. This could be to specify where the task should execute or hints about how the code should be compiled. What makes Tornado stand out is that this configuration can be performed dynamically either by setting a system property or programmatically by the application. This feature means that the developer does not need to decide which device to use until runtime and hence, there is no need for this mapping to be specified within the application source code.

Another key feature of Tornado is composability; the ability to write applications with many tasks that have complex data-dependencies. To achieve this, tasks are grouped together to form a task-schedule providing the Tornado runtime system the ability to perform inter-task optimizations (something that is neglected by most of the prior art). Through the use of task-schedules developers are shielded from the complexities of scheduling data-movement in complex applications. Internally, the runtime system uses the data-dependencies between tasks to automatically infer data-movements and exploit any available task-parallelism between data-movement and computation tasks. Moreover, Tornado enables task-schedules to be executed asynchronously and, hence, developers do not need to wait for their completion. This allows them to overlap code execution between the application running on the JVM and the code running on the accelerators easily.

Finally, to extract high levels of performance from hardware accelerators Tornado supports data-parallelism. Tornado allows developers to markup any data-parallel loops with an @Parallel annotation. This annotation does not alter the semantics of the code running inside the JVM in any way. However, if the code is executed via a task-schedule, Tornado will try to execute the loop-iterations in parallel. In order to keep the source code architecture-neutral, Tornado prohibits embedding a specific parallelization scheme in the source. Instead, the parallelization scheme is automatically determined by the compiler or dynamically configured by the developer (Section 3.6). This feature allows the developer to quickly experiment with a range of different parallelization options without the overhead of having to re-compile the application — something that is not possible with any of the AOT compiled languages like OpenACC or OpenMP.

**Tornado Code Example**

Listing 1 is taken directly from our Kfusion application (Section 4.1) and illustrates how the first task of the pre-processing stage, mm2meter, is executed using Tornado. This aptly demonstrates one of the most important aspects of Tornado: the clear separation of the code that defines the computation from the code that co-ordinates its execution. Here the computation is defined by the mm2meter’s method and the co-ordination by the TaskSchedule s0. The real benefit of this separation is the ability to create tasks from existing code, enabling legacy code to be accelerated, while requiring
minimal code refactoring. Unlike all other prior art, Tornado provides more flexibility in the composition of heterogeneous code by allowing tasks to be defined from methods or lambda functions — using the Java 8 Functional Interfaces in line 16.

Listing 1. A Tornado example of the mm2meter kernel of Kinect Fusion.

```java
public static final ImageFloat mm2meters(ImageFloat dest, ImageFloat src, int scaleFactor) {
    for (@Parallel int y = 0; y < dest.Y(); y++) {
        for (@Parallel int x = 0; x < dest.X(); x++) {
            final int sx = scaleFactor * x;
            final int sy = scaleFactor * y;
            dest.set(x, y, src.get(sx, sy) * 1e-3f);
        }
    }
}

public static void main(String[] args) {
    ImageFloat inputImage = ...
    ImageFloat outputImage = ...
    int scaleFactor = ...
    new TaskSchedule("s0")
        .task("t0", ImagingOps::mm2meters, outputImage, inputImage, scaleFactor)
        .streamOut(outputImage);
    .execute();
}
```

In Tornado it is the responsibility of a task-schedule to coordinate which code needs to run on each hardware accelerator. Here a task-schedule, s0, is created in line 15 that contains a single task t0, from the mm2meter method, and copies out the final value of outputImage. The programmer should consider the task-schedule as a lexical closure — like a lambda function — that is not evaluated immediately. In this instance, the code is only executed when `execute` is called in line 18, here task t0 will invoke the mm2meter method with parameters outputImage, inputImage, and scaleFactor. Furthermore, then the streamOut operator will ensure the value of outputImage is synchronized with the host before control is returned to the application.

There are two distinct features of a task-schedule that a developer needs to be aware. Firstly, the assignment of code to run on devices can be done dynamically and so does not need to be made explicit in the source code — this is an example of dynamic configuration and is discussed in Section 3.4. Secondly, as each task has the potential to execute on a different device, managing data-movement is critical to obtain high performance. To avoid unnecessary transfers, Tornado will optimize all data-movement within the task-schedule automatically. The rule for data flowing across task-schedule boundaries is that Tornado will transfer the absolute minimum amount of data possible to enable the task-schedule to execute correctly. This means that by default, all reference types — objects and arrays — are copied onto the accelerator automatically the first time they are used, however, nothing is automatically (or implicitly) copied back to the host. This behaviour means that data will always remain on the last device on which it was created and forces the developer to make explicit requests to transfer data back to the host; hence the purpose of the streamOut operator (line 18) on the outputImage variable.

As each task has the potential to execute on a different device, managing data-movement is critical to obtain high performance. To avoid unnecessary transfers, Tornado will optimize all data-movement within the task-schedule automatically. The rule for data flowing across task-schedule boundaries is that Tornado will transfer the absolute minimum amount of data possible to enable
the task-schedule to execute correctly. This means that by default, all reference types — objects and arrays — are copied onto the accelerator automatically the first time they are used, however, nothing is automatically (or implicitly) copied back to the host. This behaviour means that data will always remain on the last device on which it was created and forces the developer to make explicit requests to transfer data back to the host; hence the purpose of the `streamOut` operator (line 18) on the `outputImage` variable.

Regarding the exploitation of data-parallelism, Tornado offers an optional annotation that can be inserted in the source code. The `@Parallel` annotation (lines 2 and 3) is used to inform the compiler that it is safe to execute each iteration of the annotated loop independently or out-of-order. It should be obvious that this annotation does not imply that the code will always be parallelized or how — as the device that it is to execute can change dynamically — and is only a strong hint. The decision on how a parallelization scheme is chosen is described in Section 3.6.

### 3.2 System Architecture

Tornado consists of three main components: an API, a runtime system, and a JIT compiler. Figure 1 illustrates Tornado’s system architecture along with a typical execution flow. As shown, the execution is driven by task-schedules which describe a data-flow graph of tasks (referred to as a `task-graph`).

A task-graph is constructed the first time a task-schedule is executed via `execute` and is passed to the Graph Optimizer (1). At this stage the task-graph only contains tasks which execute code and, therefore, it has to be augmented with tasks (or nodes) that handle data-transfers. To achieve this, the Graph Optimizer passes each node 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 bytecode 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 is of paramount importance since it 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 HIR 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 enables the usage of other heterogeneous frameworks and code generators like CUDA/PTX [20, 21] or HSA/HSAIL [11, 12].

Once all sketches are available, the optimizer (2) tries to eliminate as many unnecessary nodes as possible from the task-graph and then generates an 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 also cached.

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 this is the OpenCL 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 (retrieved from the HIR cache) compilation will be triggered. Furthermore, any parallelization strategy or device specific builtins are applied to the HIR at this stage. The output of the code generator is currently OpenCL C code. Finally, since Tornado uses a pluggable driver interface, it is designed to sit atop of low-level heterogeneous programming APIs, such as CUDA, HSA and OpenCL. This allows us to easily port Tornado on other low-level heterogeneous programming languages with minimum effort.

3.3 Task Schedules

Instead of scheduling individual tasks, Tornado allows developers to compose multiple tasks together into task-schedules. A task-schedule is essentially a lexical closure around a set of tasks and from the developer’s perspective it serves as a synchronization point. Task-schedules are modelled as data-flow graphs (or task-graphs) and form a larger schedulable unit of work than single tasks.
Tornado uses this approach for a number of reasons: firstly, it provides a clean separation between the code which coordinates task execution and the code which performs the actual computation; and secondly, it allows the Tornado runtime to employ a wider range of runtime optimizations. For instance, the task-graphs provide information about inter-task data-dependencies that the runtime can use to exploit any available task-parallelism; thereby automatically overlapping the execution of non-data-dependent tasks. Additionally, it also allows unnecessary data transfers, such as read-after-write data-dependencies between tasks, to be eliminated.

The motivation behind task-schedules is exemplified by the task-graphs shown in Figure 2 that are constructed from Listings 2 and 3. Again, in this example we focus on the two kernels that constitute the pre-processing stage of the Kinect Fusion application (Section 4.1). In this stage, the two kernels (mm2meters and bilateralFilter) form a pipeline in which the output of the first kernel is consumed by the second. As shown in Listings 2, the outputImage of the mm2meters kernel is passed as input parameter to the bilateralFilter kernel. If Tornado did not employ task-schedules, the runtime system would have to copy the outputImage variable from the host device to the JVM and backwards as shown in Figure 2a. By allowing the construction of task-schedules, as shown in Listing 3, the Graph Optimizer can automatically infer this producer-consumer dependency and persist the data on the host device eliminating the redundant copy (Figure 2b). In addition, by using task-schedules, the application does not need to hand control
back to the JVM between executing the tasks improving performance even further. Another benefit of using task-schedules is that it allows Tornado to work with any composite variable: both objects and arrays. Developers are not required to access variables through a custom data-structure and, hence, less modifications to the source code are required and more code can be re-used.

### 3.4 Dynamic Configuration

A common issue with languages such as OpenMP and OpenACC is that their APIs embed hardware specific optimizations into the source code of applications. Subtle options like specifying a parallelization strategy or loop tile size actually introduce bias to a particular device. During the development of Tornado, it became clear that these types of configuration options need to be specified for each data-parallel region of code (task) on a per device basis. Although this is possible to some degree in OpenACC, the API makes it difficult to specify parameters for different classes of the same accelerator, for example it is difficult to distinguish between NVIDIA GT750M and K20 GPGPUs. Moreover, we found three main problems: 1) all tuning parameters need to be specified ahead-of-time, 2) for every device we wish to support an extra clause is required, and 3) 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. While these issues do not seem limiting, they lead to a dramatic decrease in productivity in the situation where an application needs to run across multiple different system configurations. In this situation, the combinatorial effect of tuning these systems becomes chaotic since the developer has to develop, run, and re-optimize the application for every system before it is released to the end-user. These issues are resolved in Tornado through the use of JIT compilation since configurations can be changed dynamically and, if necessary, the code for the accelerator can be re-compiled.\(^2\)

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 or to compile ahead-of-time. Instead, the tuning parameters can be specified on the command line (or in a configuration file) on a per system basis. The application can, then, be tuned without having to modify the source code and/or re-compile the application. Moreover, the design space exploration of the parameter space can be automated without requiring access to the source code — making it possible for the end-user, and not the developer, to optimize performance for each system.

Tornado’s JIT compiler has the ability to parallelize tasks, allowing developers to adjust the strategy used dynamically without re-writing any code. The parallelization scheme is similar to OpenACC and OpenMP but is performed dynamically. This allows us to quickly adjust application performance for individual accelerators. If required, Tornado also allows tasks to be compiled using hand-crafted OpenCL. Listing 4 shows the dynamic configuration file used to obtain 124 frames per second on the NVIDIA K20 in Figure 4.

Listing 4. Dynamic configuration example.

```plaintext
kfusion.tornado.platform=1
kfusion.tornado.device=0
kfusion.reduce.custom=True
kfusion.reduce.fraction=1.0
tornado.opencl.schedule=False
tornado.opencl.gpu.block.x=1024
tornado.opencl.gpu.block2d.x=32
tornado.opencl.gpu.block2d.y=4
```

\(^2\)Although OpenCL has a JIT compiler, it does not provide support for parallelizing code and, therefore, developers must provide multiple implementations of the same kernel — one for each device.
3.5 Managed Device Memory
Tornado is designed to make both data-movement and memory management as transparent as possible. This is achieved by creating a managed heap to store objects on each device. The heap persists for the duration of the application, allowing the runtime system to keep large objects in-situ on the devices; allowing the developer to exploit data locality. Using the information gathered by the Graph Optimizer, the runtime system is able to track the state of objects across multiple and disparate memories. This is critical to avoid unnecessary data-movement between these disparate memories and enable garbage collection to be performed. Typically, objects are only collected if the on-device heap becomes full and the object has either been collected inside the JVM or is not required by next task.

3.6 Just-In-Time Compiler
The Tornado compiler is constructed using Graal [6, 7]. This gives it the advantage of using an industrial strength JIT compiler and the ability to generate code directly from Java bytecode. To create our final compiler, we augmented Graal with the ability to parallelize code (discussed later) and to generate OpenCL C. For the current implementation of Tornado, we made a practical decision to target OpenCL C since it provides the largest accelerator coverage.

To target highly-parallel devices, Tornado is able to parallelize any canonical for loop contained within a Java method. To enable this feature, developers need to annotate the induction variable of a 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. The use of the annotation does not provide any guarantees about if or how the loop will 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.

Presently, the Tornado 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 orientated processors, such as x86 cores, whereas the second provides a finer thread granularity which is preferred by throughput orientated devices, like GPGPUs.

Listing 5. Loop Re-Written for CPUs.
```java
int id = get_global_id(0);
int block_size_x = (dest.X() + get_thread_size - 1) / get_thread_size(0);
int block_size_y = (dest.Y() + get_thread_size - 1) / get_thread_size(0);
for (int y = id * block_size; y < min(start_y + block_size_y, dest.Y()); y++) {
    for (int x = id * block_size; x < min(start_x + block_size_x, dest.X()); x++) {
        final int sx = scaleFactor * x;
        final int sy = scaleFactor * y;
        final float value = src.get(sx, sy) * 1e-3f;
        dest.set(x, y, value);
    }
}
```

Listing 6. Loop Re-Written for GPGPUs.
```java
for (int y = get_global_id(0); y < dest.Y(); y+= get_global_size(0)) {
    for (int x = get_global_id(1); x < dest.X(); x+= get_global_size(1)) {
        final int sx = scaleFactor * x;
        final int sy = scaleFactor * y;
        final float value = src.get(sx, sy) * 1e-3f;
        dest.set(x, y, value);
    }
}
```

Publication date: January 2016.
3.7 Java Coverage

Since Tornado aims to maximize code re-use, its API has been designed to avoid introducing unnecessary changes into existing application code. To enable this, Tornado imposes fewer restrictions on the code contained within tasks by supporting a larger subset of the Java language than the majority of the prior art. This is demonstrated in Listing 1 where the code still remains valid sequential Java code and freely uses language features such as virtual method calls and objects. Throughout the development of Tornado, we have found that the only real constrain 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 (Figure 3b) which inhibits our ability to adequately support exceptions.

4 EVALUATION

The evaluation of Tornado differs from previous related published work in terms of complexity and completeness. Instead of focusing on micro-benchmarks or individual kernels, we accelerated end-to-end a complex Computer Vision (CV) application that showcases the general applicability and performance of Tornado. In addition, the evaluation was performed on a wide and diverse set of heterogeneous hardware resources presented in Table 5. The following subsections describe in detail the anatomy of the evaluated CV application along with the performance results and optimizations employed.

4.1 Kinect Fusion

Kinect Fusion (KF) is a Computer Vision application which reconstructs a three-dimensional scene from a stream of depth images produced by an RGB-D camera, such as the Microsoft Kinect. SLAMBench [18] provides many open-source implementations of KF [19], from which we derive our Java reference implementation. Using SLAMBench also provides the benefit of ready-made infrastructure to measure both performance and accuracy which enables us to compare different KF implementations reliably. The performance measure of KF is the number of incoming RGB-D frames processed each second and is measured in frames per second (FPS). For an implementation to work in real-time a minimum processing rate (or QoS level) of 30 FPS or more is required. Moreover, to ensure accuracy, we use the ICL-NUIM dataset [9] for experiments as this provides us with ground truths for the camera trajectory. For a result to be considered valid it must have a mean absolute trajectory error (ATE) of under 5cm; as per the criteria set out by SLAMBench.

KF is also interesting from an implementation perspective since there is an abundance of parallelism which can be exploited to improve its performance. Implementation-wise, some of the kernels are large containing over 250 lines of code (LOC) and utilize a much wider range of Java language features than the benchmarks typically used to evaluate the performance of heterogeneous programming frameworks.

Tornado implements KFusion’s processing pipeline in eight separate task schedules: one for each of the preprocessing, integrate, raycast, and rendering stages, and four for the tracking stage.

| Stage     | Kernel(s)                        | Invoc | Description                                                                                                                                 |
|-----------|----------------------------------|-------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Acquisition | 1: acquisition                   | 1     | Obtains the next RGB-D frame (from a camera or a file).                                                                                  |
| Pre-processing | 2: mm2meters, bilateral filter | 2     | Cleans and interprets the raw data by applying a bilateral filter to remove anomalous values and rescaling the input data to represent distances in millimeters. Builds a pyramid of vertex and normal maps. |
| Tracking   | 5: half sample, depth2vertex, vertex2normal, track, reduce | 11-44 | 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) [3]. |
| Integrate  | 1: integrate                     | 0-1   | Fuses the current frame into the internal model.                                                                                         |
| Raycast    | 1: raycast                       | 0-1   | Constructs a new reference point cloud.                                                                                                 |
| Rendering  | 3: render depth, render track, render volume | 0-3   | Uses the same raycasting technique of the previous stage to produce a visualization of the 3D scene.                                      |

Table 4. List of KF stages and kernels.

Publication date: January 2016.
Table 4 provides the details of the various stages of the KF application along with the numbers of the kernels implemented per stage. Typically, to process a single frame of RGB-D data, 18 to 54 kernels are required to be executed. Therefore, to achieve a frame rate of 30 FPS, the application must sustain the execution of between 540 and 1620 kernels per second — meaning that KF will test both the performance of individual kernels and the throughput of the runtime system.

4.2 Results

To evaluate our Tornado KF implementation, we target nine different multi- and many-core devices that are available across four classes of systems (shown in Table 5). Each system has a multi-core processor, along with a minimum of one GPGPU. The choice of systems is deliberate to show how subtle differences make heterogeneous programming difficult. For example, we have three classes of NVIDIA GPGPUs which, despite being from the same vendor, have vastly different performance characteristics and therefore require different code optimizations.

Figure 4 provides the best frame rates that we achieved during our experiments across all nine accelerators. Each Tornado implementation has been optimized using dynamic configuration and our approach is discussed in Section 4.2.1. The first notable outcome of our experiments is that the SLAMBench OpenCL implementation produced valid results on only six devices; opposed to Tornado that produced valid results on every device. The issue is that the OpenCL implementation makes some a priori assumptions about the target hardware accelerator such as the size of local memory or the maximum dimensions of a work-group. Consequently, if either of these assumptions

![Fig. 4. Performance in FPS of KF across all nine devices.](image_url)
are incorrect the application fails. More importantly, these results show that by using Tornado we have been able to exceed our minimum level of QoS on four devices and that the three devices that comfortably exceed this level are all GPGPUs. In terms of raw performance, we were able to achieve a maximum of 124 FPS on the NVIDIA K20 that translates into a speedup of $166\times$ over the reference serial Java implementation (Figure 5b).

One of the most positive aspects of the results, shown in Figure 5a, is that Tornado performs at worst 21% less than the hand-written OpenCL and in three cases it was even better by between 8 to 70% (if we exclude any outliers). The negative performance deltas are chiefly down to the extra runtime overheads of Tornado: data serialization, data movement and real-world effects caused by the JVM — for instance when it performs garbage collection or data structures are resized.

4.2.1 Dynamic Configuration and Optimization. The strength of Tornado is its ability to configure and optimize an application dynamically. This fact differentiates Tornado from other heterogeneous programming frameworks since it allows us to experiment easily with many configuration options and optimizations — that are both intuitive and non-intuitive — without having to recompile or augment the source code of our applications. Our general approach to tuning KF, and other applications more generally, is to write an application to be portable across all devices and then use dynamic configuration to specialize it in situ.

Figure 5a compares the performance between Tornado and OpenCL. As shown, in most cases, Tornado is on par with the hand-crafted OpenCL implementation of SLAMBench. A notable exception is the NVIDIA GTX 550Ti where Tornado significantly outperforms OpenCL (x14.77). The reason behind that the device performs significantly better when blocking OpenCL are used. This is explained, in detail, later in Section 4.2.1.2. Below we describe some of the configurations parameters and optimizations we dynamically applied to Tornado boosting its performance.

4.2.1.1 Typical Optimization Process. During our initial evaluation, we noticed that in some cases Tornado was not performing comparably to the hand-written OpenCL. After analyzing in detail the various stages of the KFusion algorithm, we discovered that the tracking stage was our performance bottleneck. The tracking stage was transferring 14 MB of data between the device and host per frame dominating the execution profile by between 25% and 75%. On the contrary, the OpenCL version avoids this heavy data transferring by always compressing the data on the device before transferring it to the host by using a hand-crafted reduction kernel. However, although this hand-crafted kernel works well on GPGPUs, it fails when the target device does not have the expected number of resources. To eliminate this performance difference and avoid this problem, the Tornado implementation has two available reduction schemes: 1) Simple Reduce which is a simple two-stage reduction which partially compresses the data on the device and then finishes the reduction on the host, and 2) Optimized Reduce which is a topologically optimized reduction that is able to efficiently reduce partial results on the device generating the smallest result — at the cost of being usable on GPGPU style devices only. Furthermore, each of the reduction options are dynamically configurable so that we can vary the amount of resources they utilize.

Figure 5b shows the various levels of performance achieved from the initial Tornado configuration of KFusion (52.88 FPS) up to the most optimized one (166.85 FPS). We notice that by dynamically selecting which reduce function to use (via the command line) we are able to boost KF’s performance on the NVIDIA K20 up to 145.80 FPS. The final optimization we performed was to finely tune the local work-group dimensions used to execute kernels on the K20. By tuning these parameters, we

\[\text{Currently, our topologically optimized reduction uses a hand-crafted OpenCL kernel to simulate support that is yet to be added into the Tornado JIT compiler.}\]
pushed the performance even higher to 166.85 FPS. Our dynamic configuration for this system is depicted in Listing 4.

4.2.1.2 Blocking vs Non-Blocking OpenCL calls. A prime example of Tornado’s strength is when counter-intuitive optimizations have to employed in order to boost performance. For instance, consider Figure 5c which applies the same optimizations to the NVIDIA GTX 550Ti as we applied previously to the NVIDIA K20. Here, both the hardware and OpenCL drivers are from the same vendor but behave completely differently depending on whether we use blocking or non-blocking OpenCL calls. As Tornado is able to quickly toggle between these modes, it was able to outperform OpenCL on this device by a factor of x14.77.

This example demonstrates why using a posteriori knowledge is beneficial when programming heterogeneous systems. The SLAMBench OpenCL version, that is programmed using a priori knowledge, uses only non-blocking calls assuming that this would yield the best performance in all cases — as it happens in the NVIDIA K20. However, in the context of the GTX550i this is not the case. This is hard to rectify in the hand-crafted OpenCL version as the developer would have to firstly speculate that this counter-intuitive optimization is the missing part of the performance bottleneck, and secondly to implement it in the source code. On the contrary, in Tornado this is just a matter of experimenting (on the command line) with different configuration parameters and hence performing a design-space exploration easily.

4.2.1.3 Parallelization scheme. As described in Section 3.6 and showcased in Listings 5 and 6, depending on the target device, Tornado can dynamically compile for different parallelization schemes (CPU: block cyclic, GPU: thread cyclic). Naturally, the developer would choose block cyclic if the target is a CPU or the more fine-grained thread-cyclic parallelization scheme if the target is a GPU (as happens by default in Tornado). Unfortunately, this intuitive choice does not always hold true as showcased in Figure 5d. In the case of the Intel i7-2600K multicore CPU, if we use the GPU-style parallelization scheme, we manage to boost performance from 14.84 FPS to 17.30 FPS (x1.17). Again, Tornado allows this counter-intuitive experimentation via dynamic configuration without requiring any code rewriting. We just simply force Tornado to treat the CPU as a GPU.

5 CONCLUSIONS AND FUTURE WORK

In this paper 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. 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 runtime. Optimization, through Tornado, no longer needs to involve costly re-compilation cycles and source modifications.

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 with a worst case performance loss of 21% when compared against hand-written OpenCL. 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 up to $166 \times$ over the reference Java implementation. The results show that not only, we can obtain levels of performance that meet our QoS target of 30 FPS, but also to exceed our target by $4 \times$ (at 124 FPS).
Regarding future work, we aim to extend Tornado to allow user-defined reductions and improve performance of both the runtime and compiler. Furthermore, we plan to extend Tornado’s reach to more devices and diverse accelerators such as Intel’s Xeon Phi and FPGAs. Finally, Tornado and the Kinect Fusion application are available as an open-source project at http://tornado.software.

REFERENCES

[1] AMD. 2012. APARAPI. (2012). Retrieved 20th April 2017 from http://aparapi.github.io

[2] 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. DOI: http://dx.doi.org/10.1145/1869459.1869469

[3] 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. DOI: http://dx.doi.org/10.1109/34.121791

[4] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface (Version 4.0). (2013). Retrieved 20th April 2017 from http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[5] 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. DOI: http://dx.doi.org/10.1145/2254064.2254066

[6] 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.
[7] 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. DOI: http://dx.doi.org/10.1145/2542142.2542143

[8] 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. DOI: http://dx.doi.org/10.1145/3050748.3050761

[9] 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.

[10] 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. DOI: http://dx.doi.org/10.1145/2509136.2509516

[11] HSA Foundation. 2016. HSA Programmer Reference Manual Specification 1.1. (2016). Retrieved 20th April 2017 from http://www.hsafoundation.com/?ddownload=5115

[12] HSA Foundation. 2017. HSA Foundation. (2017). Retrieved 20th April 2017 from http://www.hsafoundation.com

[13] IBM Corporation. 2016. J9 JVM. (2016). Retrieved 20th April 2017 from https://www.ibm.com/developerworks/java/jdk/

[14] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. 2015. Compiling and Optimizing Java 8 Programs for GPU Execution. In 2015 International Conference on Parallel Architecture and Compilation (PACT). 419–431. DOI: http://dx.doi.org/10.1109/PACT.2015.46

[15] HSA Foundation. 2016. HSA Programmer Reference Manual Specification 1.1. (2016). Retrieved 20th April 2017 from http://www.hsafoundation.com/?ddownload=5115

[16] IBM Corporation. 2017. HSA Foundation. (2017). Retrieved 20th April 2017 from http://www.hsafoundation.com

[17] 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. DOI: http://dx.doi.org/10.1145/3050748.3050764

[18] 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). arXiv:1410.2167.

[19] 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. DOI: http://dx.doi.org/10.1109/ISMAR.2011.6092378

[20] NVIDIA Corporation. 2017. CUDA. (2017). Retrieved 20th April 2017 from http://developer.nvidia.com/cuda-zone

[21] NVIDIA Corporation. 2017. Parallel Thread Execution ISA Version 4.0. (2017). Retrieved 20th April 2017 from http://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[22] Nathaniel Nystrom, Derek White, and Kishen Das. 2011. Firepille: 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. DOI: http://dx.doi.org/10.1145/2047862.2047883

[23] OpenACC-Standard.org. 2015. The OpenACC Application Programming Interface (Version 2.5). (2015). Retrieved 20th April 2017 from http://www.openacc.org/sites/default/files/inline-files/OpenACC_pt5.pdf

[24] Project Sumatra. (2014). Retrieved 20th April 2017 from http://openjdk.java.net/projects/sumatra/

[25] 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. DOI: http://dx.doi.org/10.1109/HPCCC.2012.57

[26] Michel Steuwer and Sergei Gorlatch. 2014. SkelCL: A High-level Extension of OpenCL for multi-GPU Systems. J. Supercomput. 69, 1 (July 2014), 25–33. DOI: http://dx.doi.org/10.1007/s11227-014-1213-y