Dynamic Application Reconfiguration on Heterogeneous Hardware

Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak
Maria Xekalaki, James Clarkson, Christos Kotselidis
The University of Manchester
{first.last}@manchester.ac.uk

Abstract

By utilizing diverse heterogeneous hardware resources, developers can significantly improve the performance of their applications. Currently, in order to determine which parts of an application suit a particular type of hardware accelerator better, an offline analysis that uses *a priori* knowledge of the target hardware configuration is necessary. To make matters worse, the above process has to be repeated every time the application or the hardware configuration changes.

This paper introduces TornadoVM, a virtual machine capable of reconfiguring applications, at run-time, for hardware acceleration based on the currently available hardware resources. Through TornadoVM, we introduce a new level of compilation in which applications can benefit from heterogeneous hardware. We showcase the capabilities of TornadoVM by executing a complex computer vision application and six benchmarks on a heterogeneous system that includes a CPU, an FPGA, and a GPU. Our evaluation shows that by using dynamic reconfiguration, we achieve an average of 7.7× speedup over the statically-configured accelerated code.

CCS Concepts • **Software and its engineering → Dynamic compilers**; **Runtime environments**;

Keywords Dynamic Reconfiguration, FPGAs, GPUs, JVM

ACM Reference Format:
Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 13–14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3313808.3313819

VEE '19, April 13–14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313819

1 Introduction

The advent of heterogeneous hardware acceleration as a means to combat the stall imposed by the Moore's law [39] created new challenges and research questions regarding programmability, deployment, and integration with current frameworks and runtime systems. The evolution from single-core to multi- or many- core systems was followed by the introduction of hardware accelerators into mainstream computing systems. General Purpose Graphics Processing Units (GPGPUs), Field-programmable Gate Arrays (FPGAs), Application Specific Integrated Circuits (ASICs), and integrated many-core accelerators (e.g., Xeon Phi) are some examples of hardware devices capable of achieving higher performance than CPUs when executing suitable workloads. Whether using a GPU or an FPGA for accelerating specific workloads, developers need to employ programming models such as CUDA [9] and OpenCL [17], or High Level Synthesis (HLS) tools [29] to program and accelerate their code.

The integration of these new programming models to mainstream computing has not been fully achieved in all aspects of programming or programming languages. For example, in the Java world, excluding IBM's J9 [21] GPU support and APARAPI [2], there are no other commercial solutions available for automatically and transparently executing Java programs on hardware accelerators. The situation is even more challenging for FPGA acceleration where the programming models are not only different than the typical ones, but also the tool-chains are in the majority of the cases separated from the programming languages [29]. Consequently, programming language developers need to create either new bindings to transition from one programming language to another [22], or specific (static or dynamic) compilers that compile a subset of an existing programming language to another one tailored for a specific device [1, 12, 13, 19, 33, 38]. Therefore, applications are becoming more heterogeneous in their code bases (i.e. mixing programming languages and paradigms), resulting in harder to maintain and debug code.

Ideally, developers should follow the programming norm of *"write-once-run-anywhere"* and allow the underlying runtime system to dynamically adjust the execution depending on the provisioned hardware. Achieving the unification or co-existence of the various programming models under a common runtime, would not only result in more efficient code development but also in portable applications, in terms

of performance as well, where the system software adapts the application to the underlying hardware configuration.

At the same time, the question of which parts of the code should execute on which accelerator remains open, further increasing the applications' complexity. Various techniques such as manual code inspection [36], offline machine learning based models [20], heuristics [6], analytical [3], and statistical models [16] have been proposed in order to identify which parts of an application are more suitable for acceleration by the available hardware devices. Such approaches, however, require either high expertise on the developer's side in order to reason about which part of their source code would be better accelerated, or *a priori* decision making regarding the type and characteristics of the devices upon which the offline analysis will be performed. Therefore, the majority of these approaches require developers' intervention and offline work to achieve the desired results.

Naturally, the research question that rises is: *"Is it possible for a system to find the best configuration and execution profile automatically and dynamically?"*.

In this paper we show that the answer to this question can be positive. We introduce a novel mechanism that tackles the aforementioned challenges by allowing the dynamic and transparent execution of code segments on diverse hardware devices. Our mechanism performs execution permutations, at run-time, in order to find the highest performing configuration of the application. To achieve this, we employ *nested application virtualization* for Java applications running on a standard Java Virtual Machine (JVM). At the first level of virtualization, standard Java bytecodes are executed either in interpreted or just in time (JIT) compiled mode on the CPU. At the second level of virtualization, the code compiled for heterogeneous hardware is executed via a secondary lightweight bytecode interpreter that allows code migration between devices, while handling automatically both execution and data management. This results in a system capable of dynamically adapting its execution until it discovers the highest performing configuration completely transparently to the developer and the application. In detail, this paper makes the following contributions:

- Presents TornadoVM: a virtualization layer enabling dynamic migration of tasks between different devices.
- Analyses how TornadoVM performs a best-effort execution in order to automatically and dynamically (i.e. at run-time) discover which device or combination of devices results in the best performing execution.
- Discusses how TornadoVM can be used, by existing JVMs with tiered compilation, as a new tier, breaking the CPU-only compilation boundaries.
- Showcases that by using TornadoVM we are able to achieve an average of 7.7× performance improvements over the statically-configured accelerated code for a representative set of benchmarks.

Listing 1. Example of the Tornado Task Parallel API.

```
1  public class Compute {
2    public void map(float[] in, float[] out) {
3      for (@Parallel int i = 0; i < n; i++) {
4          out[i] = in[i] * in[i];
5    }}
6    public void reduce(float[] in, float[] out) {
7      for (@Parallel int i = 0; i < n; i++) {
8          out[0] += in[i];
9    }}
10   public void run(float[] in, float[] out, float[] temp) {
11     new TaskSchedule("s0")
12       .task("t0", this::map, in, temp)
13       .task("t1", this::reduce, temp, out)
14       .streamOut(out).execute();
15   }}
```

2 Background and Motivation

This work builds upon Tornado [23], an open-source parallel programming framework that enables dynamic JIT compilation and execution of Java applications onto OpenCL-compatible devices, transparently to the user. This way it enables inexperienced –with hardware accelerators– users to accelerate their Java applications by introducing a minimal set of changes to their code and choosing an accelerator to target. Tornado consists of the following three main components: a parallel API, a runtime system, and a JIT compiler and driver.

Tornado API: Tornado provides a task-based parallel API for parallel programming within Java. By using the Tornado API, developers express parallelism in existing Java applications with minimal alterations of the sequential Java code. Each task comprises a Java method handle containing the pure Java code and the data it accesses. The Tornado API provides interfaces to create *task-schedules*; groups of tasks that will be automatically scheduled for execution by the runtime. In addition to defining tasks, Tornado allows developers to indicate that a loop is a candidate for parallel execution through the `@Parallel` annotation.

Listing 1 shows a parallel map/reduce computation using the Tornado API. The Java class `Compute` contains two methods, `map` in Line 2 and `reduce` in line 6. These two methods are written in Java augmented with the `@Parallel` annotation. The first method performs a vector multiplication while the second computes an addition of the elements.

Lines 11–14 create a task-schedule containing the two tasks of our example along with their input and output arguments. Both the task-schedule and the individual tasks receive string identifiers (`s0`, `t0` and `t1`) that enable programmers to reference them at runtime.

Furthermore, since our example performs a map-reduce operation, the intermediate results of `map` (`t0`) are passed to `reduce` (`t1`) through `temp`. Finally, we invoke `execute` (Line 14) to signal the execution of the task-schedule.

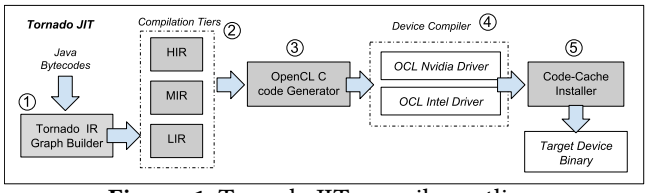

Figure 1. Tornado JIT compiler outline.

Tornado runtime: The role of the Tornado runtime system is to analyze data dependencies between tasks within a task-schedule, and use this information to minimize data transfers between a host (e.g., a CPU) and the devices (e.g., a GPU). In the example of Listing 1, the Tornado runtime will discover the read-after-write dependency on the `temp` array and instead of copying it back to the host, it will persist it on the device. Additionally, due to this dependency, it will ensure that task `t1` will not be scheduled before task `t0` completes.

Tornado JIT compiler and driver: At runtime, Tornado has a two-tier JIT compilation mode that allows it to first compile Java bytecode to OpenCL C, and then OpenCL C to machine code. Figure 1 provides a high-level overview of Tornado's compilation chain. As shown, Java bytecode is transformed to an Intermediate Representation (IR) graph (Step ①) which is then optimized and lowered incrementally from High-level IR (HIR), to Mid-level IR (MIR), and finally reaching the Low-level IR (LIR) state, which is close to the assembly level (Step ②). From that point, instead of generating assembly instructions, a special compiler phase is invoked which rewrites the LIR graph to OpenCL C code (OpenCL code generator) (Step ③). After the OpenCL C source code is created, depending on the target device that execution will take place, the respective OpenCL device compiler is invoked (Step ④). Finally, the generated binary code gets installed to the code cache (Step ⑤) and is ready for execution.

2.1 Motivation

Due to architectural differences, different hardware accelerators tend to favour different workloads. For instance, GPUs are well known for their high efficiency when all threads of the same *warp* perform the same operation, but they fail to efficiently accelerate parallel applications with complex control flows [26]. Experienced developers with in-depth understanding of their applications and the underlying hardware can potentially argue about which devices better suit their applications. However, even in such cases, choosing the best configuration or the best device from a family of devices is not trivial [5, 6, 11].

To better understand how different accelerators or data sizes affect the performance of an application we run the DFT (Discrete Fourier Transform) benchmark using Tornado on three different devices while varying the input size. Figure 2 depicts the obtained results with the X-axis showing the range of the input size and the Y-axis showing the speedup

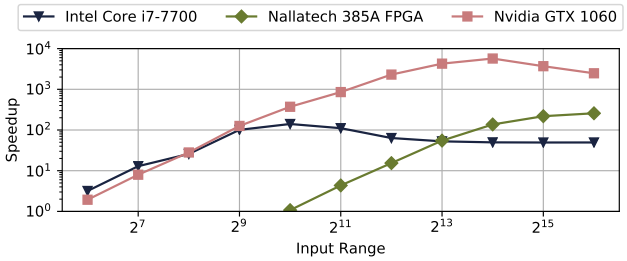

Figure 2. Tornado speedup over sequential Java on a range of different input sizes for DFT (Discrete Fourier Transform).

over the sequential Java implementation. Each line in the graph represents one of the three different devices we run our experiment on: an Intel i7 CPU, an NVIDIA GTX 1060 GPU, and an Intel Altera FPGA. Overall, DFT performs better when running on the NVIDIA GPU. However, when running with small sizes, the highest performing device varies. For example, for input sizes between 2^6-2^8, the parallel execution on the multi-core CPU is the highest performing one.

The importance of dynamic reconfiguration: As showcased by our experiment, to achieve the highest performance one needs to explore a large space of different executions before discovering the best possible configuration for an application. To make matters worse, this configuration is considered the "best possible" only for the given system setup and input data. Each time the code changes, a new device is introduced, or the input data changes we need to perform further exploration and potentially restart our application to apply the new configuration.

To address the challenge of discovering the highest performing configuration, we introduce TornadoVM: a system that is able to automatically and dynamically adapt execution to the best possible configuration, according to the user's requirements, for each application and input data size in a heterogeneous system, without the need of restarting the application.

3 TornadoVM

In order to enable dynamic application reconfiguration of the executed applications on heterogeneous hardware, we implement a **virtualization layer** that uses the Tornado parallel programming framework to run Java applications on heterogeneous hardware. The implemented virtualization layer is responsible for executing and performing code migration of the generated JIT compiled code through a lightweight bytecode-based mechanism, as well as managing the memory between the different devices. The combination of the aforementioned components results in a heterogeneous JVM, called TornadoVM, capable of dynamically reconfiguring the executed applications on the available hardware resources completely transparently to the users.

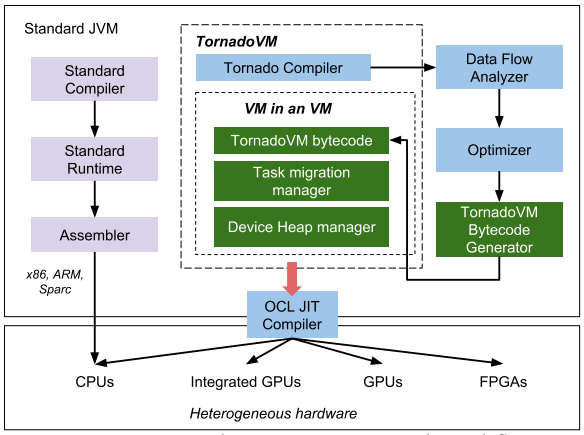

Figure 3. TornadoVM overview and workflow.

TornadoVM is implemented in Java and, as illustrated in Figure 3, runs inside a standard Java Virtual Machine (e.g., the HotSpot JVM [25, 32]); resulting in a VM that runs inside another VM—*VM in a VM*. The TornadoVM interprets TornadoVM bytecodes, manages the corresponding memory, and orchestrates the execution on the heterogeneous devices. The JVM executes Java bytecodes and the interpreter methods of TornadoVM.

To implement TornadoVM, we augment the original Tornado components (shown in light blue color in Figure 3) with the components shown in dark green color. Namely, we introduce *a)* a *bytecode generator* (Section 3.2) responsible for the generation of TornadoVM specific bytecodes (Section 3.1) that are used to execute code on heterogeneous devices, *b)* a *bytecode interpreter* (Section 3.1) that executes the generated bytecodes, *c)* a *device heap manager* (Section 3.3) responsible for managing data across the different devices ensuring a consistent memory view, and *d)* a *task migration manager* (Section 3.4) responsible for migrating tasks between devices. All the components of TornadoVM are device agnostic except for the final connection with the underlying OpenCL driver.

Initially, the application starts on the standard JVM (host) which can execute it on CPUs. When the execution reaches a Tornado API method invocation, the control flow of the execution is passed to the Tornado compiler in order to create and optimize the data flow graph for the task-schedule at hand. The data flow graph is then passed to the TornadoVM bytecode generator that generates an optimized compact sequence of TornadoVM bytecodes, describing the corresponding instructions of the task-schedule. In contrast to the original Tornado, at this point, TornadoVM does not JIT compile the tasks involved in the compiled task-schedule to binaries. The task compilation, from Java bytecode to binary, is performed *lazily* by the TornadoVM upon attempting to execute a task whose code has not been compiled yet for the corresponding target device. For each device there is a code cache maintaining the binaries corresponding to the tasks that have been already compiled for this device to avoid paying the compilation overhead multiple times.

3.1 TornadoVM Bytecodes

TornadoVM relies on a custom set of bytecodes that are specifically tailored to describe task-schedules, resulting in a more compact representation, which is also easier to parse and translate to heterogeneous hardware management actions. Table 1 enlists the bytecodes that are currently generated and supported by the TornadoVM. TornadoVM employs 11 bytecodes that allow the VM to prepare the execution, to perform data allocation and transferring (between the host and the devices), as well as to launch the kernels. All the bytecodes are hardware agnostic and are used to express a task-schedule regardless of the device(s) it will run on.

All the TornadoVM bytecodes take at least one argument, the context identifier, which is a unique number used to identify a task-schedule. TornadoVM generates a new context identifier for each task-schedule in the program. The context identifier is used at run-time to obtain a context object which, among others, contains references to the data accessed by the task-schedule, and information about the device, on which, the tasks will be executed. Additionally, all bytecodes except `BEGIN`, `END`, and `BARRIER`, take at least a bytecode index as an argument. The bytecode indices are a way to uniquely identify bytecodes so that we can then reference them from other bytecodes. In addition, they are used for synchronization and ordering purposes since 6 out of the 11 bytecodes are non-blocking in order to increase performance by overlapping data transfers and execution of kernels. The TornadoVM bytecodes can be conceptually grouped in the following categories:

Initialization and Termination: Bytecode sections in the TornadoVM are encapsulated in regions that start with the `BEGIN` bytecode and conclude with the `END` bytecode. These bytecodes essentially signal the activation and deactivation of a TornadoVM context.

Memory allocation and data transferring: In order to execute code on a heterogeneous device, memory has to be allocated and data need to be transferred from the host to the heterogeneous device. The `ALLOC` bytecode allocates sufficient memory on the device heap (see Section 3.3), to accommodate the objects passed to it as an argument. The `COPY_IN` bytecode both allocates memory and transfers the object to the device, while the `STREAM_IN` bytecode only copies the object (assuming a previous allocation). Note that the `COPY_IN` bytecode is used for read-only data and implements a caching mechanism that allows it to skip data transfers if the corresponding data are already on the target device. On the other hand, the `STREAM_IN` bytecode is used for data streaming on the heterogeneous device, in which the kernel is executed multiple times with an open channel for receiving new data. The corresponding bytecodes for copying the data back from the device to the host are `COPY_OUT` and `STREAM_OUT`.

Table 1. TornadoVM bytecodes.

Bytecode	Operands	Blocking	Description
BEGIN	<context>	Yes	Creates a new parallel execution context.
ALLOC	<context, BytecodeIndex, object>	No	Allocates a buffer on the target device.
STREAM_IN	<context, BytecodeIndex, object>	No	Performs a copy of an object from host to device.
COPY_IN	<context, BytecodeIndex, object>	No	Allocates and copies an object from host to device.
STREAM_OUT	<context, BytecodeIndex, object>	No	Performs a copy of an object from device to host.
COPY_OUT	<context, BytecodeIndex, object>	No	Allocates and copies an object from device to host.
COPY_OUT_BLOCK	<context, BytecodeIndex, object>	Yes	A blocking COPY_OUT operation.
LAUNCH	<context, BytecodeIndex, task, Args>	No	Executes a task, compiling it if needed.
ADD_DEP	<context, BytecodeIndices>	Yes	Adds a dependency between labels.
BARRIER	<context>	Yes	Waits for all previous bytecodes to be finished.
END	<context>	Yes	Ends the parallel execution context.

Synchronization: TornadoVM bytecodes can be ordered and synchronized through the BARRIER, ADD_DEP, and END bytecodes. BARRIER and END wait for all previous bytecodes to reach completion, while ADD_DEP waits only for those corresponding to the indices passed to it as parameters.

Computation: The LAUNCH bytecode is used to execute a kernel. To execute the code on the target device, the TornadoVM first checks if a binary that targets the corresponding device (according to the context) has been generated for the task at hand. Upon success, it directly executes the binary on the heterogeneous device. Otherwise, TornadoVM compiles the input task, through Tornado, and installs into the code cache from where it is then retrieved for execution.

3.2 TornadoVM Bytecode generator

TornadoVM relies on Tornado to obtain a data flow graph for each task-schedule. The data flow graph is essentially a data structure that describes data dependencies between tasks. During the compilation of task-schedules, Tornado builds this graph and optimizes it to reduce data transfers. This optimized data dependency graph is then used to generate the final TornadoVM bytecode. The TornadoVM bytecode generation is a simple process of traversing a graph and generating, for each input node in the data flow graph, a set of TornadoVM bytecodes. Listing 2 demonstrates the generated TornadoVM bytecode that corresponds to the code from Listing 1.

TornadoVM Bytecode Interpreter The TornadoVM implements a bytecode interpreter for running the TornadoVM bytecodes. Since TornadoVM uses only a limited set of 11 bytecodes, we implement the interpreter as a simple switch statement in Java. TornadoVM bytecodes are not JIT compiled, but the interpreter itself can be JIT compiled by the underlying JVM (e.g., Oracle HotSpot) to improve performance. Note that the TornadoVM bytecodes only orchestrates the execution between the accelerators and the host machine; they do not perform the actual computation. The latter is JIT compiled by Tornado.

Listing 2. Generated TornadoVM bytecodes for Listing 1.

```
1  BEGIN   <0>            // Starts a new context
2  COPY_IN <0, bi1, in>   // Allocates and copies <in>
3  ALLOC   <0, bi2, temp> // Allocates <temp> on device
4  ADD_DEP <0, bi1, bi2>  // Waits for copy and alloc
5  LAUNCH  <0, bi3, @map, in, temp> // Runs map
6  ALLOC   <0, bi4, out>  // Allocates <out> on device
7  ADD_DEP <0, bi3, bi4>  // Waits for alloc and map
8  LAUNCH  <0, bi5, @reduce, temp, out> // Runs reduce
9  ADD_DEP <0, bi5>             // Wait for reduce
10 COPY_OUT_BLOCK <0, bi6, out> // Copies <out> back
11 END     <0>                  // Ends context
```

As shown in Listing 2, a new TornadoVM context starts by running the BEGIN bytecode with context-id 0 (line 1). Note that the context-id maps to a context object that contains initial information regarding the device on which execution will take place. Then, the TornadoVM performs an allocation and a data transfer through the COPY_IN bytecode (line 2). In line 3, TornadoVM performs an allocation for the temp Java array on the target device, and in line 4 it blocks to wait for the copy and the allocation to be completed. Note that the ADD_DEP in line 4 receives the bytecode indices of the COPY_IN and the ALLOC bytecodes. Then, in line 5 it launches the map task. At this stage, the TornadoVM compiles the map task by invoking the Tornado JIT compiler and launches the generated binary on the target device. Line 6 allocates the output variable of the reduce task. In addition, since the input for the reduce task is the output of the previous task, a dependency is added (line 7) and execution waits for the finalization of the LAUNCH bytecode at line 5, as well as for the allocation at line 6. Line 8 launches the reduce kernel, at line 9 it waits for the kernel to complete, and then the result is copied back from the device to the host in line 10. Finally, the current TornadoVM context ends by END at line 11. Each context of the TornadoVM manages one device meaning that all tasks that are launched from the same context are executed on the same device.

3.3 TornadoVM Memory Manager

Since heterogeneous systems typically comprise a number of distinct memories that are not always shared nor coherent, TornadoVM implements a memory manager, which is responsible for keeping the data consistent across the different devices, as well as for allocating and de-allocating memory on them. To minimize the overhead of accelerating code on heterogeneous devices with distinct non-shared memories, TornadoVM pre-allocates a memory region on each accelerator. This region can be seen as a heap extension on the target device and is managed by the TornadoVM memory manager. The initial device heap size is by default configured to be equal to the maximum capacity of global memory on each target device. However, this value can be tuned depending on the needs of each application. By pre-allocating the device heaps, TornadoVM's memory manager becomes solely responsible for transferring data between the host and the target heaps to ensure memory consistency at run-time.

In the common case, TornadoVM just copies the input data from the host to the target device and copies back the results from the target device to the host, according to the corresponding TornadoVM bytecode for each task-schedule. The most interesting cases where the TornadoVM memory manager acts are the cases of: a) migrating task-schedules to a different device (Section 3.4); and b) the case of dynamic reconfiguration (Section 4). In the case of task migration, TornadoVM allocates a new memory area on the new target device and performs the necessary data transfers from the previous target device to the new target device.

In the case of dynamic reconfiguration in which a single task may be running on more than one device, the process has more steps. Figure 4 sketches how TornadoVM manages memory on such cases. On the top left part of the Figure is a Tornado `task-schedule` that creates a task with two parameters, *a* and *b*. Parameter *a* represents an array of floats and is given as input to the task to be executed. Parameter *b* represents an array of floats where the user expects to obtain the output. These two variables are maintained in the Java heap on the host side, as in any other Java program. However, to enable code acceleration such variables need to get copied to the target device when the latter does not have access to the host memory. As a result, TornadoVM categorizes variables in two groups: *host variables* and *device variables*.

Host variables: Following the data-flow programming model, TornadoVM splits data in input and output. Data that are used solely as input are considered *read-only* and thus safe to be accessed by more than one device at a time. Output data on the other hand, contain the results of some computation and are mandatory for the correctness of the algorithm. When running the same task on different devices concurrently, despite expecting to obtain the same result at the end, we cannot use the same memory for storing that result. Different

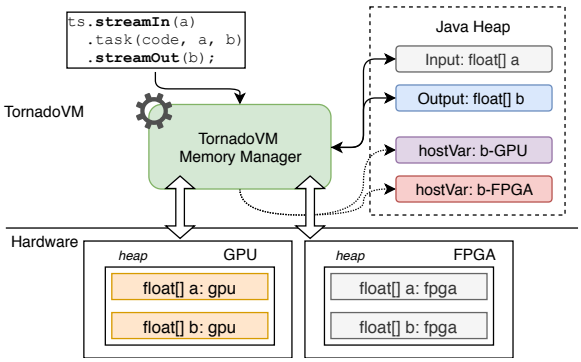

Figure 4. Overview of the TornadoVM Memory Manager

devices require different time to perform the computation and thus one device may overwrite the data of the other in an unpredictable order, possibly resulting in stale data. For this reason, the TornadoVM duplicates output variables in the host-side. This way, each device writes back the output data to a different memory segment avoiding the above issue. The code running on the host accesses this data through a proxy. When the execution for all devices finishes and the TornadoVM chooses the best device depending on the input policies (as we will present in Section 4), the TornadoVM sets the proxy to redirect accesses to the corresponding memory segment. For example, in Figure 4, if the selected device is the FPGA, the proxy of *b* will redirect accesses to the *b-FPGA* buffer.

Device variables: On the device side, different devices have different characteristics. For instance, integrated GPUs have direct coherent access to the host memory. Other devices may be able to directly access the host memory through their driver, but they still require proper synchronization to ensure coherence, e.g., external GPUs. Finally, there are devices that require explicit memory copies to and from the device. To maximize data throughput, TornadoVM dynamically queries devices for their capabilities and adapts its memory management accordingly.

3.4 Task Migration Manager

The TornadoVM task-migration manager is a component within the VM that handles code and data migration from one device to another. By employing the bytecodes and the new virtualization layer, TornadoVM is capable of migrating the executing task-schedules to different devices at runtime. Task migration is signalled by changing the target device of a task-schedule. To safely migrate task-schedules without losing data, task migrations are only allowed after *task-schedules* finish execution and become effective on the next execution.

Whenever a task-schedule completes its execution, TornadoVM checks whether the target device has been changed. If the target device has changed, TornadoVM performs two main actions: a) transfer all the necessary data from the current device to the new target device through its memory manager, and b) invoke the Tornado JIT compiler to compile

all the tasks in the `task-schedule` for the new target device, if not already compiled. After the transfers reach completion and the code gets compiled, TornadoVM can safely launch the corresponding binary on the target device and continue execution. Section 4 presents how task-migration enables TornadoVM to dynamically detect and use the best, according to some policy, configuration for the running application.

4 Dynamic Reconfiguration

With task migration, TornadoVM can dynamically reconfigure the running applications in order to discover the most efficient mapping of task-schedules on devices. TornadoVM starts executing task-schedules on the CPU, and in parallel it explores different devices on the system (e.g., GPU) and collects profiling data. Then, according to a *reconfiguration policy* it assigns scores to each device and selects the best candidate to execute each task-schedule.

4.1 Reconfiguration Policies

A *reconfiguration policy* is essentially the definition of an execution-plan, and a function that given a set of metrics (e.g., total runtime), obtained by executing a task-schedule on a device according to the execution-plan, returns an efficiency score. The higher the score, the more efficient the execution of the task-schedule on the corresponding device according to that policy. TornadoVM currently features three such policies, *end-to-end*, *latency* and *peak performance*:

- **End-to-end:** Measures the total execution time of the task-schedule by performing a single cold run on each device. The total execution time includes the time spent on JIT compilation, data transfers, and computation. The device that yields the shortest total execution time is considered the most efficient.
- **Latency:** Same as *end-to-end*, but does not wait for the profiling of all the devices to complete. By the time that the fastest device reaches completion, TornadoVM chooses that device and continues execution discarding the execution of the rest devices.
- **Peak performance:** Measures the time required to transfer data, that are not already cached on the device, and the computation time. JIT compilation and initial data transfers are not included in the measurements. To obtain these measurements the task-schedule is executed multiple times on the target device to warm it up before obtaining them.

The end-to-end policy is suitable for debugging and optimizing code. Getting access to the end-to-end measurements for each device gives users the power to tweak their programs to favour specific devices, or to identify bottlenecks and fix them to improve performance. The latency policy is more suitable for short running applications that are not expected to live long enough in order to offset the overhead of JIT compilation and warming up. The peak performance policy, on the other hand, is more suitable for long running applications that run the same task-schedules multiple times and thus diminish the initial overhead.

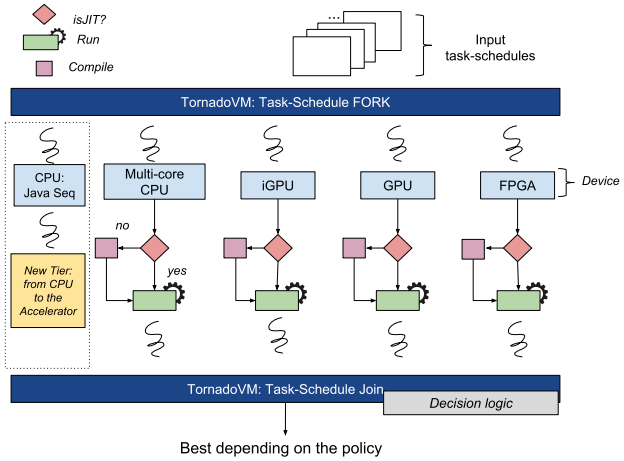

Figure 5. Overview of device selection in TornadoVM.

A policy is by default set for all the task-schedules in the whole application and can be altered through a parameter when starting the application. However, to allow users to have more control, we extend the task-based parallel API in Tornado to allow users to specify different policies per task-schedule execution. To avoid complicating the API, we overload the `execute` method with an optional parameter that defines the reconfiguration policy for the task-schedule at hand. If no parameters are passed, then TornadoVM uses the reconfiguration policy set for the whole application. For instance, to execute a task-schedule using the *performance* policy we use `taskSchedule.execute(Policy.PERFORMANCE)`.

Note that in addition to the aforementioned policies, TornadoVM allows the implementation of custom reconfiguration policies, giving its users the flexibility to set the metric on which they want their application to become more efficient, e.g., *energy* instead of *performance*.

4.2 Device Exploration

TornadoVM automatically starts an exhaustive exploration by running each task-schedule on all available devices and profiles their performance in accordance to the selected reconfiguration policy. This way, TornadoVM is able to select the best device for each task-schedule. In addition, TornadoVM does not require application restart or any prior knowledge from the user's perspective to execute and adapt their code to a target device.

Figure 5 illustrates how device selection is performed within the TornadoVM. When execution is invoked with a policy the TornadoVM spawns a set of Java threads. Each thread executes a copy of the input *task-schedules* for a particular device. Therefore, TornadoVM spawns one thread per heterogeneous device on the system. In parallel with the dynamic exploration, a Java thread is also running the task-schedule on the CPU. This is done to ensure that the

application makes progress while we explore alternatives, and to obtain measurements that will allow us to compare the heterogeneous execution with the sequential Java. Once the execution is finished, TornadoVM collects the profiling information and selects the most suitable, according to the reconfiguration policy, device for the *task-schedule* at hand.

From this point on, TornadoVM remembers the target device for each input task-schedule and policy. Note that the same task-schedule may run multiple times, potentially with different reconfiguration policies, through the overloaded `execute` method. In this case, as any new policy is encountered, the TornadoVM starts the exploration and it performs a new decision that better adapts to the given policy. In conclusion, dynamic reconfiguration enables programmers to effortlessly accelerate their applications on any system equipped with heterogeneous hardware. Furthermore, it enables the applications to dynamically adapt to changes to the underlying hardware, e.g., in cases of dynamic resource provisioning.

4.3 A new High-Performance Tier Compilation

Contemporary managed runtime systems employ tiered compilation to achieve better performance (e.g., tier compiler within JVM). Tiered compilation enables the runtime system to use faster compilers that produce less optimized code for code that is not invoked as frequently. As the number of invocations increases for a code segment the runtime re-compiles it with higher-tier compilers, which might be slower, but produce more optimized code. The main idea behind tiered compilation is that it is only worth investing time to optimize code that will be invoked multiple times. Currently, after a JIT compiler reaches the maximum tier compilation (e.g., C2 compilation in JVM), there are no further optimizations.

Following the tier-compilation concept for JVM, we add dynamic reconfiguration as a new compilation tier, improving the state-of-the-art by enabling it to further optimize code and take advantage of hardware accelerators. The HotSpot JVM employs an interpreter and two compilers in its tiered compilation (C1, and C2). When a method is optimized with the highest tier compiler (C2), TornadoVM takes action and explores more efficient alternatives, possibly utilizing heterogeneous devices. This integration allows TornadoVM to pay the exploration overhead only for methods that are invoked multiple times or contain loops with multiple iterations. A current limitation of the TornadoVM is that it can only optimize code that is written using the Tornado API, thus it is not a generally applicable compilation tier.

5 Evaluation

This section presents the experimental evaluation of TornadoVM. We first describe the experimental setup and methodology, then the benchmarks we use, and finally we present and discuss the results.

Table 2. Experimental Platform

Hardware	
Processor	Intel Core i7-7700 @ 4.2GHz
Cores	4 (8 HyperThreads)
RAM	64GB
GPU	NVIDIA GTX 1060 (Pascal), up to 1.7GHz, 6GB GDDR5, 1280 CUDA Cores
FPGA	Nallatech 385A, Intel Arria 10 FPGA, Two banks of 4GB DDR3 SDRAM each
Software	
OS	CentOS 7.4 (Linux Kernel 3.10.0-693)
OpenCL (CPU)	2.0 (Intel)
OpenCL (GPU)	1.2 (Nvidia CUDA 9.0.282)
OpenCL (FPGA)	1.0 (Intel), Intel FPGA SDK 17.1, HPC Board Support Package (BSP) by Nallatech
JVM	Java SE 1.8.0_131 64-Bit JVMCI VM

5.1 Evaluation Setup and Methodology

For the evaluation of TornadoVM we use a heterogeneous system comprising three different processing units: an Intel CPU, an external NVIDIA GPU and an Intel Altera FPGA. This configuration essentially covers all the currently supported types of target devices of Tornado, which TornadoVM relies on for heterogeneous JIT compilation. Table 2 details the hardware and software configurations of our system.

TornadoVM, being a *VM running in another VM*, falls into the performance methodology traits of managed runtime systems [14]. VMs comprise a number of complex subsystems, like the JIT compiler and the Garbage Collector, that add a level of non-determinism in the obtained results. Adhering to standard techniques of evaluating VMs, we first perform a warm-up phase for every benchmark to stabilize the performance of the JVM. After the warm-up phase finishes, we perform one more run from which we obtain the measurements that we report. Note that in our measurements the TornadoVM instance itself is not warmed up.

The results we report depend on the reconfiguration policy that we evaluate each time. For example, for the *peak performance* policy, we only include execution and data transferring times, excluding JIT compilation and device initialization times. On the other hand, for the *end-to-end* and *latency* policies, we include all times related to JIT compilation (except for FPGA compilation), device initialization, data transferring, and execution. Anticipating that in the future FPGA synthesis times will decrease, we chose to exclude JIT compilation times from the FPGA measurements when using the *end-to-end* and *latency* policies. The current state-of-the-art FPGA synthesis tools take between 60 and 90 minutes to compile our benchmarks. Therefore, including the FPGA compilation time was resulting in non-comparable measurements. However, FPGA initialization and kernel loading are still included in all our measurements. We discuss in more details all JIT compilation times in Section 5.4.

Table 3. Input and data sizes for the given set of benchmarks.

Benchmark	Data Range		Input (Mb)	Output (Mb)
	min	max	max	max
Saxpy	256	33554432	270	135
MonteCarlo	256	33554432	<0.1	268
RenderTrack	64	4096	70	50
N-Body	256	131072	0.5	0.5
BlackScholes	256	4194304	270	135
DFT	64	65536	4	1

5.2 Benchmarks

To evaluate TornadoVM, we employ six different benchmarks and a complex computer vision application. Namely, the six benchmarks we use are *Saxpy*, *Montercarlo*, *RenderTrack*, *BlackScholes*, *NBody*, and *DFT* (Discrete Fourier Transform), and the computer vision application is the *Kinect Fusion* (KFusion) [31] implementation provided by SLAMBench [30]. The domain of the chosen benchmarks ranges from mathematical and financial applications to physics and linear-algebra kernels, while KFusion creates a 3D representation from a stream of depth images produced by an RGB-D camera such as the Microsoft Kinect.

We ported each benchmark as well as KFusion from C++ and OpenCL to pure Java using the Tornado API. Porting existing applications to the Tornado API requires: a) the creation of a task-schedule to pass the references of existing Java methods, and b) the addition of the `@Parallel` annotations to the existing loops. This results to 4–8 extra lines of code per task-schedule regardless its size. The porting of the six benchmarks resulted in six Tornado-based applications with a single-task task-schedule each, while the porting of KFusion resulted in an application with multiple task-schedules, both single- and multi-task. When TornadoVM runs the sequential version, it ignores the `@Parallel` annotation and the code is compiled and executed by the standard JVM (e.g., HotSpot).

Workload Size In addition to exploring different workloads, we also explore the impact of the workload size. Thus, for each benchmark, we vary the input data sizes in powers of two. Table 3 summarizes the input sizes used for each benchmark. Please note that for `MonteCarlo`, we copy only a single element (the number of iterations) and obtain a new array with the MonteCarlo computation, while for KFusion we use the input scenes from the ICL-NUIM data-set [18].

5.3 Dynamic Reconfiguration Evaluation

Figure 6 shows the performance evaluation of the six benchmarks on all three types of supported hardware devices (CPU, GPU, and FPGA). X-axis shows the range of input sizes while y-axis shows the speedup over the sequential Java code optimized by the HotSpot JVM. Each point represents the performance achieved by a particular Tornado device, while the solid black line highlights the speedup of TornadoVM through dynamic reconfiguration. Note that in the cases where the line does not overlap with a point, it means that the code is executed by the HotSpot JVM, since it happens to outperform the Tornado devices. Using the *end-to-end* and *peak-performance* policies we are able to observe the performance of each device on the system. The top of the Figure shows the performance results when the policy *end-to-end* is used. The bottom of the Figure shows the results when the policy *peak performance* is used.

This Figure shows the dynamic reconfiguration in action by deploying a benchmark and altering the input data that we pass to it. This way we can observe how TornadoVM dynamically reconfigures it to run on the *best* device, according to the reconfiguration policy. Additionally, thanks to the profiling data that both *end-to-end* and *peak-performance* policies gather, we can observe the differences between the different devices for each benchmark and input size.

Evaluation of End-to-End Policy When using the *end-to-end* policy we observe that for small input sizes the HotSpot optimized sequential code outperforms the other configurations. This is attributed to the relatively small workload that no matter how fast it will execute it cannot hide the overhead of the JIT compilation and device initialization. As the input data size increases, we observe that TornadoVM manages to find better suited devices for all the benchmarks, except for Saxpy. This indicates that the performance gain is not enough to hide the compilation, device initialization, and data transfer overheads for this benchmark. The fact that, as the input size increases the less slowdown is observed, indicates that JIT compilation time dominates, and as we increase the workload, it starts taking a smaller portion of the end-to-end execution time.

The rest of the benchmarks are split into two groups. MonteCarlo, BlackScholes, and DFT, after a certain input size seem to stabilize and perform better on the GPU. RenderTrack and NBody, on the other hand, yield comparable performance when ran on the GPU or in parallel on the CPU. As a result, depending on the input size, TornadoVM might prefer one over the other. These two benchmarks showcase why dynamic reconfiguration is important. Imagine moving these workloads to a system with a slower GPU, and to yet another system with a faster GPU. TornadoVM will be able to use the CPU in the first case and the GPU on the second case, getting the best possible performance out of each system without the need for the user to change anything in their code. In average, TornadoVM is 7.7x faster compared to the execution on the best parallel device for this policy.

Evaluation of the Peak Performance policy When using the *peak performance* policy we observe a slightly different behaviour, as shown at the lower part of Figure 6. In this case, TornadoVM takes into account only the execution time and the data transfers, excluding JIT compilation and device

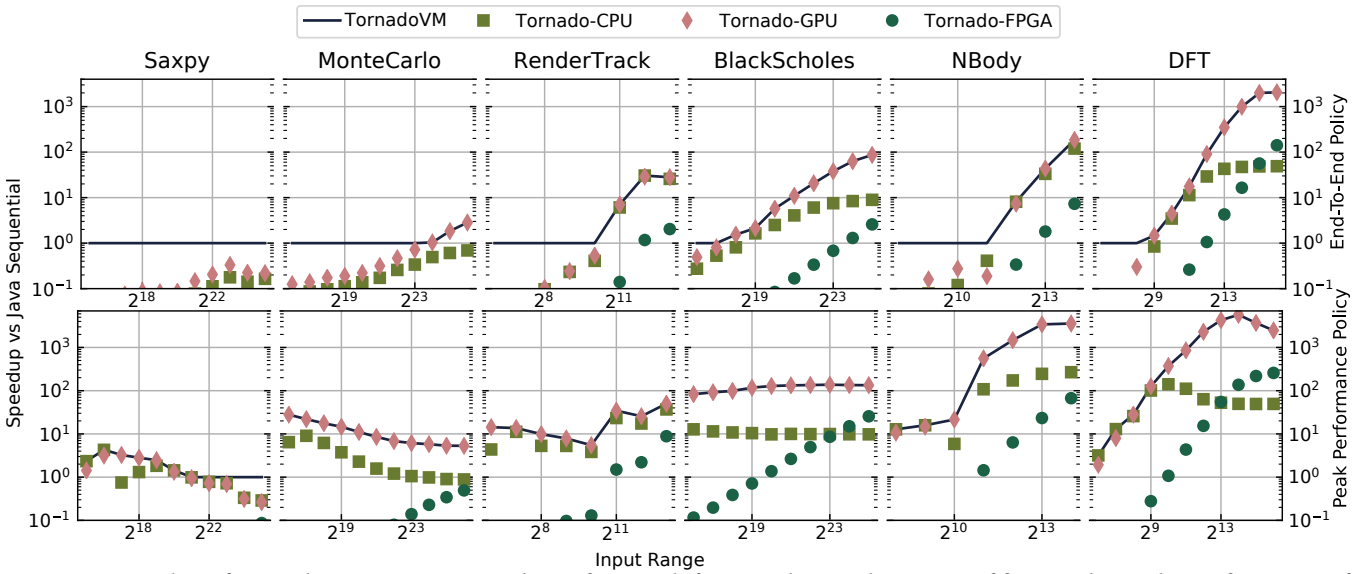

Figure 6. Speedup of TornadoVM over sequential Java for two different policies. The top set of figures shows the performance of the TornadoVM over Java sequential for the *end-to-end* policy. The bottom set of figures shows the performance of TornadoVM using the *peak performance* policy.

initialization times. In contrast to the *end-to-end* policy, we observe that when using the *peak performance* policy no matter the input size in all benchmarks, except from `saxpy`, Tornado outperforms the HotSpot sequential execution. This is expected, given that the *peak performance* policy does not take in account device initialization and JIT compilations that are the main overheads of cold runs. What is more interesting is that for Saxpy when using the *end-to-end* policy, larger input sizes were reducing the gap between sequential Java and the execution on the Tornado devices. On the contrary, when using the *peak performance* policy, we observe the opposite. This is an indication, that in Saxpy the first dominating part is JIT compilation and device initialization, and the second one is data transfers.

For the rest of the benchmarks we observe again that they can be split into two groups, however this time the groups are different. MonteCarlo, RenderTrack, and BlackScholes perform better on the GPU, no matter the input size. This indicates that these benchmarks feature highly parallel computations which also take a significant part of the total execution time. The second group includes NBody and DFT, which for smaller sizes perform better when ran in parallel on the multi-core CPU, than on the GPU.

Note that, although the FPGA is never selected in our system, it gives high-performance for BlackScholes and DFT (25× and 260× respectively compared to Java sequential, and 2.5× and 5× compared to the parallel execution on the CPU). Therefore, users that do not have a powerful GPU can benefit from executing on FPGAs, with the additional advantage that they consume less energy than running on the GPU or CPU [34].

TornadoVM vs Tornado using the Latency policy Figure 7 shows the speedups of TornadoVM, using the latency reconfiguration policy, over the corresponding Tornado executions using the best, on average, device — in our case the GPU. Recall that the *latency* policy starts running task-schedules on all devices and selects the first to finish, ignoring the rest. Then it stores these data to avoid running again on the less optimal devices. We see that for applications such as Saxpy, TornadoVM does not switch to an accelerator since sequential Java optimized by HotSpot performs better. This gives programmers speedups of up to 45× compared to the static Tornado [8]. More interesting are benchmarks such as MonteCarlo, BlackScholes and DFT. For these benchmarks, TornadoVM can run sequential Java for the smaller input sizes and migrate to the GPU for bigger input sizes, dynamically getting the best performance.

For RenderTrack and NBody, TornadoVM ends up using all devices, except for the FPGA, depending on the input size. For example in the case of RenderTrack, TornadoVM starts running sequential Java, then it switches execution to the GPU and finally to parallel execution on the CPU, giving up to 30× speedup over Tornado. Note that the speedup over Tornado goes down as the input size increases. This is due to the fact that for large input sizes the GPU manages to hide the overhead of the data transfers by processing the data significantly faster than the other devices. As a result, for large input sizes, the GPU ends up dominating on both the TornadoVM and Tornado, thus resulting in the same performance, which can be up to three orders of magnitude better than running on a standard JVM.

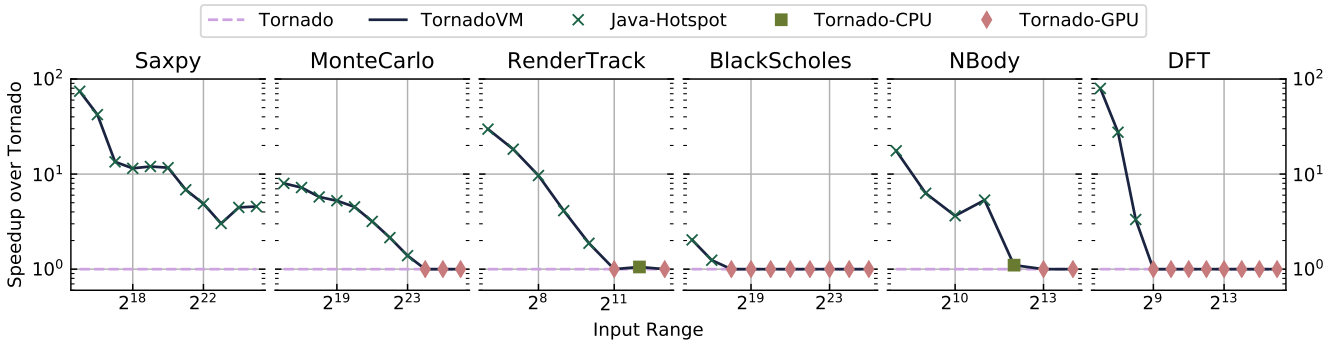

Figure 7. Speedup of TornadoVM over Tornado running on the best (on average) device.

Table 4. Breakdown analysis of execution times (in milliseconds, unless otherwise noted) per benchmark.

Benchmark	Compilation Time				Host to Device			Execution			Device To Host			Rest		
	CPU	FPGA	Load	GPU	CPU	FPGA	GPU	CPU	FPGA	GPU	CPU	FPGA	GPU	CPU	FPGA	GPU
Saxpy	7.44	53 mins	1314	99.64	19.78	19.85	59.01	57.04	248.64	2.72	10.06	13.54	20.57	1.15	20.14	1.50
MonteCarlo	85.85	54 mins	1368	87.60	1 ns	1 ns	1 ns	240.88	456.96	2.75	21.61	59.71	41.14	0.70	0.57	0.70
RenderTrack	111.10	51 mins	1380	105.03	18.59	40.10	58.35	24.50	242.15	1.96	3.86	6.84	7.70	0.69	3.03	2.61
BlackScholes	178.61	114 mins	1420	243.02	16.84	10.30	30.97	1036.12	400.31	4.43	21.07	20.45	41.14	1.09	2.36	0.93
NBody	144.68	51 mins	1387	151.25	0.05	0.04	0.08	101.81	441.48	7.47	0.04	0.10	0.08	1.31	1.21	1.04
DFT	83.80	68 mins	1398	161.96	0.09	0.10	0.16	31674.15	4424.13	460.68	0.05	0.10	0.08	1.03	1.85	1.24

KFusion To assess the correctness of TornadoVM and demonstrate its maturity, we opted to use it to accelerate KFusion as well. KFusion is composed of five distinct task-schedules, both single- and multi-task. KFusion is a streaming application that takes as input pictures captured by an RGB-D camera and processes them. After the first few frames have been processed, the system stabilizes and runs the optimized code from the code cache. As a result, we use the *peak performance policy* to accelerate it. Our evaluation results show that TornadoVM can successfully accelerate KFusion on the evaluation platform, yielding 135.27 FPS (frames-per-second) compared to the 1.69 FPS achieved by the HotSpot JVM by automatically selecting the GPU.

5.4 End-to-end times breakdown

Table 4 shows the breakdown of the total execution time, in milliseconds, for the largest input size of each benchmark divided into five categories: compilation, host to device data transfers, kernel execution, device to host data transfers, and the *rest*. The *rest* is the time spent to execute the parts of the applications that can not be accelerated, and it is computed as the total time to execute the benchmarks minus all the other categories ($Total_T = Comp_T + H2D_T + Exec_T + D2H_T + Rest_T$).

The compilation time ($Comp_T$) includes the time to build and optimize the Tornado data flow graph, and the time to compile the input tasks to the corresponding binary. For FPGA compilation we break the time down in two columns, the first one (FPGA) shows the compilation time including the FPGA synthesis in minutes, and the second (Load) shows the time needed to load the compiled bitstream on the FPGA in milliseconds. On average, CPU and GPU compilation is in the range of hundreds of milliseconds and is up to four orders of magnitude faster than FPGA compilation. From TornadoVM's perspective this is a limitation of the available FPGA synthesis tools and is the reason why it supports both just-in-time and ahead-of-time compilation for FPGAs.

Note that although we use the same configuration for all devices, data transfers between the host and the device ($H2D_T$ and $D2H_T$) are faster for FPGAs [4]. This is because of the use of pinned memory (unlocked memory) that enables fast DMA transfers between the host and the accelerators, and appears to have a greater impact on the FPGA. Since Tornado pre-allocates a buffer region for the heap, we cannot have zero copies because it does not have the pointers to the user data at the moment of calling `malloc` for the heap.

Regarding the kernel execution $Exec_T$, we observe that the GPU performs better than the rest of the devices. We also notice that the FPGA kernel execution performs much better than the multi-core CPU for `Blackscholes` and `DFT`, with speedups of 2.5× and 7.1× respectively over the multi-core execution. Finally, regarding the *rest* $Rest_T$, our measurements highlight that TornadoVM practically does not introduce any overhead, except for the FPGA execution, in which we connect our framework with the Altera tools and drivers. In the case of the memory intensive application, `saxpy`, the overheads comes from the delay between enqueuing the data into the command queue and starting the computation.

6 Related Work

This section presents the most relevant related works that allow dynamic application reconfiguration.

Dynamic Reconfiguration gMig [28] is a solution for live migration of GPU processes to other GPUs, through virtualization. When compared with TornadoVM, gMig poses the

limitation that it only targets GPUs. Aira [27] is a compiler and a runtime system that allows developers to automatically instrument existing parallel and heterogeneous code to select the best mapping device. Aira makes use of performance prediction models and resource allocation policies to perform CPU/GPU selection at runtime. TornadoVM, in contrast, does not predict upfront the best device to run a set of tasks. Instead, it adapts the execution, during runtime, to the heterogeneous device that best suits the input policy. TornadoVM can thus adapt the execution to any system configuration and achieve the best possible performance utilizing the available resources.

VirtCL [40] is a framework for programming multi-GPUs in a transparent manner by using data from previous runs to develop regression models. These models predict the total time of a task on each device. TornadoVM focuses on a single device, moving execution from CPU to the best device. Rethinagiri et al. [36] proposed a new heterogeneous system architecture and a set of applications that can fully exploit all available hardware. However, their solutions are highly customized for this new platform mixing programming languages and, thus, increasing software complexity. Che et al. [6] studied the performance of a set of applications on FPGAs and GPUs and presented applications' characteristics to decide where to place the code. However, the categorization is very generic and limited to the studied applications.

HPVM [24] is a compiler, a runtime system and a virtual instruction set for targeting different heterogeneous systems. The concept of the virtual instruction set in HPVM is similar to the TornadoVM bytecodes. However, all bytecodes described in TornadoVM are totally hardware agnostic, allowing to easily ship those bytecodes between different machines and different hardware. Besides, HPVM does not support task reconfiguration like TornadoVM. Hayashi et al. [20] and Grewe et al. [16] employed machine learning techniques to address the challenge of device selection. In contrast, TornadoVM adapts execution with no prior knowledge and models about the input programs.

Reconfiguration for Interpreted Languages and DSLs In the dynamic programming language domain, Qunaibit et al. [35] presented MegaGuard, a compiler framework for compiling and running Python programs on GPUs. MegaGuard is able to choose the fastest device to offload the computation. Although this approach is similar to ours, the analysis was performed on a single GPU instead of multiple devices such as GPUs, FPGAs and CPUs. Dandelion [7, 37] combines a runtime system and a set of compilers for running Language Integrated Queries (LINQs) on heterogeneous hardware. Dandelion compiles .NET bytecodes to heterogeneous code, and generates data-flow-graphs for the orchestration of the execution. In contrast, TornadoVM compiles Java bytecodes to heterogeneous code, and data-flow-graphs to custom bytecode which it then interprets for the orchestration of the execution. Leo [10] builds on top of Dandelion and provides dynamic profiling and optimization for heterogeneous execution on GPUs. TornadoVM provides a more generic framework in which tasks can be profiled and re-scheduled between different types of devices at runtime (e.g., from an FPGA to a GPU).

7 Conclusions and Future Work

In this paper we present TornadoVM, a virtualization layer that works in cooperation with standard JVMs and is able to automatically compile, and execute code on heterogeneous hardware. In addition, TornadoVM is capable of automatically discovering, at runtime, the best combination of heterogeneous devices for running input tasks to increase the performance of running applications, completely transparently to the users. We also present TornadoVM as a new level of tier-compilation and execution to make transparent use of heterogeneous hardware. To the best of our knowledge, there is no prior work that can dynamically compile and reconfigure the running applications on heterogeneous hardware, including FPGAs, without requiring any *a priori* knowledge of the underlying hardware and the applications. Finally, we demonstrate that TornadoVM can achieve on average 7.7× speedup over statically-configured parallel executions for a set of six benchmarks.

Future Work We plan to extend TornadoVM by implementing a batch execution mechanism that will allow users to run big data applications that do not fit on the memory of a single device. We also plan to introduce new policies, such as power draw and energy consumption of each device. An interesting policy to implement is also the ability to adapt the code based on the real cost (in dollars) of running the code for each device, and try to minimize those costs at reasonable performance. Furthermore, we plan to extend the dynamic reconfiguration capabilities for tasks within a task-schedule as we currently explore migration at the task-schedule level. We also plan to integrate this solution for multiple VMs running on different nodes in distributed scenarios, namely cloud-based VMs in which code can adapt to different heterogeneous hardware. Moreover, we would like to extend TornadoVM with the ability to share resources, such as in [15], that allows user to run multiple task-schedules on GPUs and FPGAs concurrently. Lastly, we project to introduce a fault-tolerant mechanism that allows users to automatically reconfigure running applications in case of failures.

8 Acknowledgments

This work is partially supported by the European Union's Horizon 2020 E2Data 780245 and ACTiCLOUD 732366 grants. Authors would also like to thank Athanasios Stratikopoulos and the anonymous reviewers for their valuable feedback.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.

[2] AMD. 2016. Aparapi. (2016). http://aparapi.github.io/.

[3] Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, and Albert Cohen. 2017. Optimization Space Pruning Without Regrets. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. ACM, New York, NY, USA, 34–44. https://doi.org/10.1145/3033019.3033023

[4] Ray Bittner, Erik Ruf, and Alessandro Forin. 2014. Direct GPU/FPGA communication Via PCI express. *Cluster Computing* 17 (01 Jun 2014). https://doi.org/10.1007/s10586-013-0280-9

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[6] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. 2008. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In *2008 Symposium on Application Specific Processors*. 101–107. https://doi.org/10.1109/SASP.2008.4570793

[7] Eric Chung, John Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *40th International Symposium on Computer Architecture* (40th international symposium on computer architecture ed.). ACM.

[8] 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 *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*. ACM, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3237009.3237016

[9] NVIDIA Corporation. 2019. CUDA. (2019). Retrieved February 25, 2019 from http://developer.nvidia.com/cuda-zone

[10] Naila Farooqui, Christopher J. Rossbach, Yuan Yu, and Karsten Schwan. 2014. Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*. USENIX Association, Broomfield, CO. https://www.usenix.org/conference/trios14/technical-sessions/presentation/farooqui

[11] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 2012. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12)*. ACM, New York, NY, USA, 47–56. https://doi.org/10.1145/2145694.2145704

[12] 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

[13] 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

[14] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[15] A. Goswami, J. Young, K. Schwan, N. Farooqui, A. Gavrilovska, M. Wolf, and G. Eisenhauer. 2016. GPUShare: Fair-Sharing Middleware for GPU Clouds. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1769–1776. https://doi.org/10.1109/IPDPSW.2016.94

[16] Dominik Grewe and Michael F. P. O'Boyle. 2011. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *Compiler Construction*, Jens Knoop (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 286–305.

[17] Khronos Group. 2017. OpenCL. (2017). Retrieved February 25, 2019 from https://www.khronos.org/opencl

[18] 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.

[19] 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

[20] Akihiro Hayashi, Kazuaki Ishizaki, Gita Koblents, and Vivek Sarkar. 2015. Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. ACM, New York, NY, USA, 27–36. https://doi.org/10.1145/2807426.2807429

[21] IBM. 2018. IBM J9 Virtual Machine. (2018). https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.win.70.doc/user/java_jvm.html.

[22] JOCL 2017. Java bindings for OpenCL. (2017). http://www.jocl.org/.

[23] 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

[24] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous Parallel Virtual Machine. In *PPoPP '18 Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 68–80. https://doi.org/10.1145/3178487.3178493

[25] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot^{TM} Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[26] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 451–460. https://doi.org/10.1145/1815961.1816021

[27] Robert Lyerly, Alastair Murray, Antonio Barbalace, and Binoy Ravindran. 2018. AIRA: A Framework for Flexible Compute Kernel Execution in Heterogeneous Platforms. In *IEEE Transactions on Parallel and Distributed Systems*. https://doi.org/10.1109/TPDS.2017.2761748

[28] Jiacheng Ma, Xiao Zheng, Yaozu Dong, Wentai Li, Zhengwei Qi, Bingsheng He, and Haibing Guan. 2018. gMig: Efficient GPU Live Migration Optimized by Software Dirty Page for Full Virtualization. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*. ACM, New York, NY, USA, 31–44. https://doi.org/10.1145/3186411.3186414

[29] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions*

on Computer-Aided Design of Integrated Circuits and Systems 35, 10 (Oct 2016), 1591–1604. https://doi.org/10.1109/TCAD.2015.2513673

[30] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. LujÃąn, M. F. P. O'Boyle, G. Riley, N. Topham, and S. Furber. 2015. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 5783–5790. https://doi.org/10.1109/ICRA.2015.7140009

[31] 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

[32] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java hotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1267847.1267848

[33] 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

[34] Berten Digital Processing. 2016. *White paper: GPU vs FPGA Performance Comparison*. Technical Report. http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf

[35] Mohaned Qunaibit, Stefan Brunthaler, Yeoul Na, Stijn Volckaert, and Michael Franz. 2018. Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 16:1–16:29. https://doi.org/10.4230/LIPIcs.ECOOP.2018.16

[36] S. K. Rethinagiri, O. Palomar, J. A. Moreno, O. Unsal, and A. Cristal. 2015. Trigeneous Platforms for Energy Efficient Computing of HPC Applications. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. 264–274. https://doi.org/10.1109/HiPC.2015.19

[37] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 49–68. https://doi.org/10.1145/2517349.2522715

[38] Sumatra. 2015. Sumatra OpenJDK. (2015). http://openjdk.java.net/projects/sumatra/.

[39] Thomas N. Theis and H. S. Philip Wong. 2017. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science and Engg.* 19, 2 (March 2017), 41–50. https://doi.org/10.1109/MCSE.2017.29

[40] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. 2015. VirtCL: A Framework for OpenCL Device Abstraction and Management. In *PPoPP 2015 Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 161–172. https://doi.org/10.1145/2688500.2688505