TornadoVM

Release v1.0

The University of Manchester

Feb 29, 2024

CONTENTS

TornadoVM is a plug-in to OpenJDK and other JDK distributions (e.g., GraalVM, Red Hat Mandrel, Amazon Corretto, Microsoft OpenJDK, Azul Zulu) that allows developers to automatically run Java programs on heterogeneous hardware. TornadoVM targets devices compatible with OpenCL, PTX and Level-Zero, which include multi-core CPUs, dedicated GPUs (Intel, NVIDIA, AMD), integrated GPUs (Intel HD Graphics and ARM Mali), and FPGAs (Intel and Xilinx).

TornadoVM has three backends: OpenCL, NVIDIA CUDA PTX, and SPIR-V. Developers can choose which backends to install and run.

TornadoVM does not replace existing VMs, but it rather complements them with the capability of offloading Java code to OpenCL, PTX and SPIR-V, handling memory management between Java and hardware accelerators, and running/coordinating the compute-kernels.

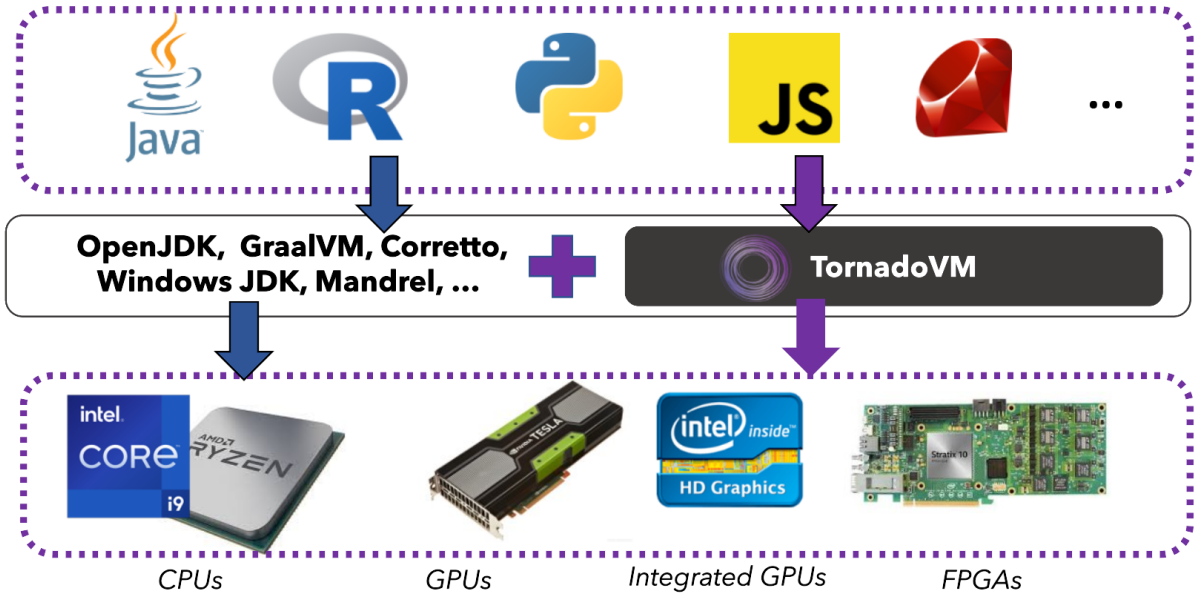
CHAPTER

ONE

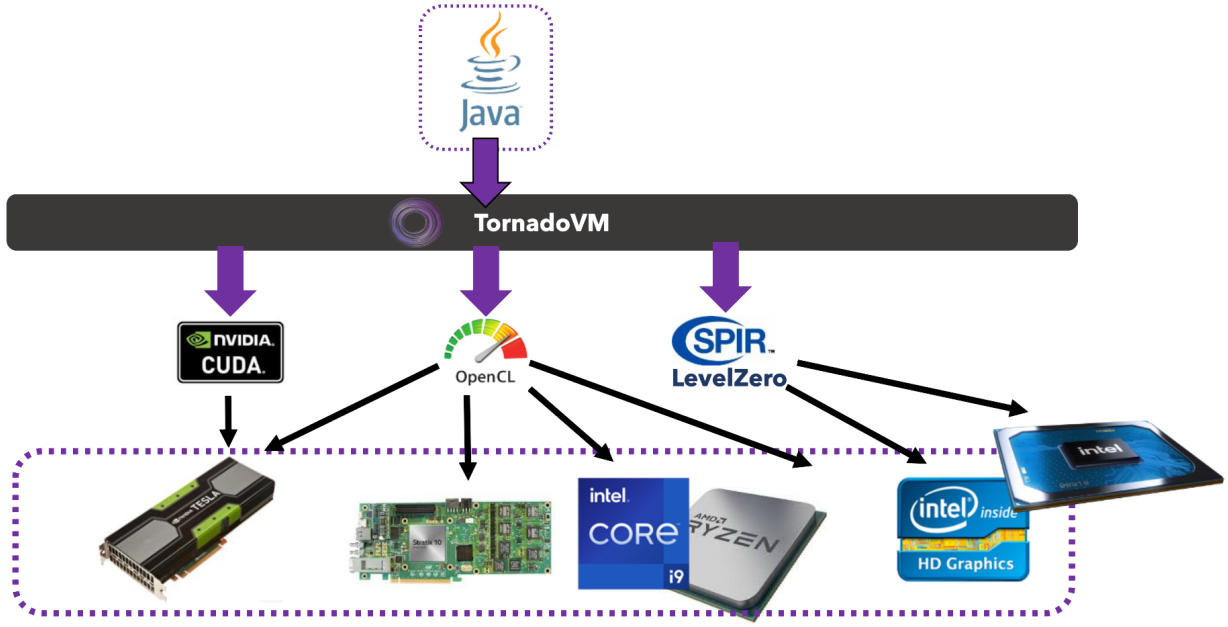
CONTENTS

1.1 Introduction to TornadoVM

TornadoVM is a plug-in to OpenJDK and other JDK distributions (e.g., GraalVM, Red Hat Mandrel, Amazon Corretto, Microsoft OpenJDK, Azul Zulu) that allows developers to automatically run Java programs on heterogeneous hardware. TornadoVM targets devices compatible with OpenCL, PTX and Level-Zero, which include multi-core CPUs, dedicated GPUs (Intel, NVIDIA, AMD), integrated GPUs (Intel HD Graphics and ARM Mali), and FPGAs (Intel and Xilinx).



TornadoVM currently has three backends: It compiles Java code, at runtime, from Java bytecode to OpenCL C, NVIDIA CUDA PTX, and SPIR-V binary. Developers can choose which backends to install and run.



TornadoVM does not replace existing VMs, but it rather complements them with the capability of offloading Java code to OpenCL, PTX and SPIR-V, handling memory management between Java and hardware accelerators, and running/coordinating the compute-kernels.

1.1.1 Main Features

TornadoVM includes the following features:

- **Hardware agnostic APIs:** The APIs are hardware agnostic, which means that, from the developers' view, the source code is exactly the same for CPUs, GPUs and FPGAs. It is the TornadoVM runtime and the TornadoVM JIT Compiler that specialises the code per architecture.
- **Optimising Just In Time (JIT) compiler per device architecture:** This means that the code is specialised per architecture (e.g., the GPU generated code is specialised differently compared to FPGAs and CPUs).
- **Live task migration:** this means that TornadoVM can migrate, at runtime, tasks from one accelerator to another (e.g., from CPU to the GPU).
- **Batch processing:** TornadoVM can partition datasets to process in batches on devices that has less physical memory than the main device (CPUs).
- **Deployable from edge to cloud:** TornadoVM can be deployed on low-power device (e.g., NVIDIA Jetson Nano), Desktop PCs, servers and data centers.
- **Containers:** TornadoVM can be used within Docker containers for running on NVIDIA GPUs, Intel CPUs and Intel GPUs.

TornadoVM is a research project developed at APT Group at The University of Manchester.

1.2 Installation & Configuration

1.2.1 Pre-requisites

- Maven Version >= 3.6.3
- CMake >= 3.6
- **At least one of following drivers:**
  - OpenCL drivers: GPUs and CPUs >= 2.1, FPGAs >= 1.0
  - NVIDIA drivers and CUDA SDK 10.0+
  - Intel drivers and Level-Zero >= 1.2
- GCC or clang/LLVM (GCC >= 9.0)
- Python (>= 3.0)

For Mac OS X users: the OpenCL support for your Apple model can be confirmed here.

1.2.2 Supported Platforms

The following table includes the platforms that TornadoVM can be executed.

OS	OpenCL Backend	PTX Backend	SPIR-V Backend
CentOS >= 7.3	OpenCL for GPUs and CPUs >= 2.1, OpenCL for FPGAs >= 1.0	CUDA 10.0+	Level-Zero >= 1.2
Fedora >= 21	OpenCL for GPUs and CPUs >= 2.1, OpenCL for FPGAs >= 1.0	CUDA 10.0+	Level-Zero >= 1.2
Ubuntu >= 16.04	OpenCL for GPUs and CPUs >= 2.1, OpenCL for FPGAs >= 1.0	CUDA 10.0+	Level-Zero >= 1.2
Pop!_OS >= 22.04	OpenCL for GPUs and CPUs >= 2.1, OpenCL for FPGAs >= 1.0	CUDA 10.0+	Level-Zero >= 1.2
OpenSuse Leap 15.4	OpenCL for GPUs and CPUs >= 2.1, OpenCL for FPGAs >= 1.0	CUDA 10.0+	Level-Zero >= 1.2
Mac OS X Mojave 10.14.6	OpenCL for GPUs and CPUs >= 1.2	Not supported	Not supported
Mac OS X Catalina 10.15.3	OpenCL for GPUs and CPUs >= 1.2	Not supported	Not supported
Mac OS X Big Sur 11.5.1	OpenCL for GPUs and CPUs >= 1.2	Not supported	Not supported
Apple M1	OpenCL for GPUs and CPUs >= 1.2	Not supported	Not supported
Windows 10/11	OpenCL for GPUs and CPUs >= 2.1, FPGAs not tested	CUDA 10.0+	Not supported/tested

Note: The SPIR-V backend of TornadoVM is only supported for Linux OS. Besides, the SPIR-V backend with Level Zero runs on Intel HD Graphics (integrated GPUs), and Intel ARC GPUs.

1.2.3 Installation

TornadoVM can be built with three compiler backends and is able to generate OpenCL, PTX and SPIR-V code. There are two ways to install TornadoVM:

A) Automatic Installation

The `tornadoVMInstaller.sh` script provided in this repository will compile/download OpenJDK, `cmake` and it will build TornadoVM. This installation script has been tested on Linux and OSx. Additionally, this installation type will automatically trigger all dependencies, therefore it is recommended if users only need to invoke TornadoVM as a library.

```
$ ./bin/tornadovm-installer
  usage: tornadovm-installer [-h] [--version] [--jdk JDK] [--backend BACKEND] [--
↪listJDKs] [--javaHome JAVAHOME]

  TornadoVM Installer Tool. It will install all software dependencies except the GPU/
↪FPGA drivers

  optional arguments:
   -h, --help            show this help message and exit
   --version             Print version of TornadoVM
   --jdk JDK             Select one of the supported JDKs. Use --listJDKs option to see␣
↪all supported ones.
   --backend BACKEND     Select the backend to install: { opencl, ptx, spirv }
   --listJDKs            List all JDK supported versions
   --javaHome JAVAHOME   Use a JDK from a user directory
```

NOTE Select the desired backend with the `--backend` option:

- `opencl`: Enables the OpenCL backend (it requires OpenCL drivers)
- `ptx`: Enables the PTX backend (it requires NVIDIA CUDA drivers)
- `spirv`: Enables the SPIRV backend (it requires Intel Level Zero drivers)

For example, to build TornadoVM with GraalVM and JDK 21:

```
## Install with Graal for JDK 21 using PTX, OpenCL and SPIRV backends
./bin/tornadovm-installer --jdk graalvm-jdk-21  --backend opencl,ptx,spirv
```

Another example: to build TornadoVM with OpenJDK 21 for the OpenCL and PTX backends:

```
./bin/tornadovm-installer --jdk jdk21 --backend opencl,ptx
```

After the installation, the scripts create a directory with the TornadoVM SDK. The directory also includes a source file with all variables needed to start using TornadoVM. After the script finished the installation, set the `env` variables needed by using:

```
$ source source.sh
```

B) Manual Installation

TornadoVM can be executed with the following configurations:

Note: For simplicity you can use SDKMAN for managing multiple JDK versions.

Linux

- TornadoVM with GraalVM for Linux and OSx (JDK 21): see the installation guide here: *Installation for GraalVM for JDK 21.0.1 on Linux and OSx*.
- TornadoVM with JDK21 (e.g. OpenJDK 21, Red Hat Mandrel, Amazon Corretto): see the installation guide: *TornadoVM for JDK 21 on Linux and OSx*.

Windows

To run TornadoVM on **Windows 10/11 OS**, install TornadoVM with GraalVM. More information here: *TornadoVM for Windows 10/11 using GraalVM*.

ARM Mali GPUs and Linux

To run TornadoVM on ARM Mali, install TornadoVM with GraalVM and JDK 21. More information here: *TornadoVM on ARM Mali GPUs*.

Compilation with Maven

This installation type requires users to manually install all the dependencies, therefore it is recommended for developing TornadoVM. At least one backend must be specified at build time to the `make` command:

```
## Choose the desired backend
$ make BACKENDS=opencl,ptx,spirv
```

Installation for GraalVM for JDK 21.0.1 on Linux and OSx

1. Download GraalVM JDK 21.0.1

GraalVM **Community Edition** builds are available to download at:

https://github.com/graalvm/graalvm-ce-builds/releases/tag/jdk-21.0.1.

The examples below show how to download and extract GraalVM for JDK 21.0.0

- Example for GraalVM for JDK 21 Community 21.0.1:

```
$ wget https://github.com/graalvm/graalvm-ce-builds/releases/tag/jdk-21.0.1/graalvm-
↪community-jdk-21.0.1_linux-x64_bin.tar.gz
$ tar -xf graalvm-community-jdk-21.0.1_linux-x64_bin.tar.gz
```

with SDKMAN:

```
$ sdk install java 21-graalce
$ sdk use java 21-graalce
```

The Java binary will be found in the *graalvm-jdk-{JDK_VERSION}-23.1.0`* directory. This directory is used as the JAVA_HOME (See step 2).

Note if installed with SDKMAN there is no need to manually set your JAVA_HOME.

For OSX:

- Example for GraalVM for JDK 21 Community 21.0.1:

```
$ wget https://github.com/graalvm/graalvm-ce-builds/releases/tag/jdk-21.0.1/graalvm-
↪community-jdk-21.0.1_macos-x64_bin.tar.gz
```

then `untar` it to the OSX standard JDK location `/Library/Java/JavaVirtualMachines/` or to a folder of your choice.

1. Download TornadoVM

```
$ cd ..
$ git clone https://github.com/beehive-lab/TornadoVM tornadovm
$ cd tornadovm
```

Create/edit your configuration file:

```
$ vim etc/sources.env
```

The first time you need to create the `etc/sources.env` file and add the following code in it **(after updating the paths to your correct ones)**:

```
#!/bin/bash
export JAVA_HOME=<path to GraalVM jdk> ## This path is produced in Step 1
export PATH=$PWD/bin/bin:$PATH    ## This directory will be automatically generated␣
↪during Tornado compilation
export TORNADO_SDK=$PWD/bin/sdk   ## This directory will be automatically generated␣
↪during Tornado compilation
export CMAKE_ROOT=/usr            ## or <path/to/cmake/cmake-3.10.2> (see step 4)
```

This file should be loaded once after opening the command prompt for the setup of the required paths:

```
$ source ./etc/sources.env
```

For OSX: the exports above may be added to `~/.profile`

3. Install CMAKE (if cmake < 3.6)

For Linux:

If the version of cmake is > 3.6 then skip the rest of this step and go to Step 4. Otherwise try to install cmake.

For simplicity it might be easier to install cmake in your home directory. * Redhat Enterprise Linux / CentOS use cmake v2.8 * We require a newer version so that OpenCL is configured properly.

```
$ cd ~/Downloads
$ wget https://cmake.org/files/v3.10/cmake-3.10.1-Linux-x86_64.tar.gz
$ cd ~/opt
$ tar -tvf ~/Downloads/cmake-3.10.1-Linux-x86_64.tar.gz
$ mv cmake-3.10.1-Linux-x86_64 cmake-3.10.1
$ export PATH=$HOME/opt/cmake-3.10.1/bin/:$PATH
$ cmake -version
cmake version 3.10.1
```

Then export `CMAKE_ROOT` variable to the cmake installation. You can add it to the `./etc/sources.env` file.

```
$ export CMAKE_ROOT=/opt/cmake-3.10.1
```

For OSX:

Install cmake:

```
$ brew install cmake
```

then

```
export CMAKE_ROOT=/usr/local
```

which can be added to `~/.profile`

4. Compile TornadoVM with GraalVM

```
$ cd ~/tornadovm
$ . etc/sources.env
```

To build with GraalVM and JDK 21:

```
$ make graalvm-jdk-21 BACKEND={ptx,opencl}
```

and done!!

TornadoVM for JDK 21 on Linux and OSx

DISCLAIMER:

TornadoVM is based on the Graal compiler that depends on JVMCI (Java Virtual Machine Compiler Interface). Different JDKs come with different versions of JVMCI. Therefore, the version of the Graal compiler that TornadoVM uses might not be compatible with the JVMCI version of some JDKs. Below are listed the Java 21 JDK distributions against which TornadoVM has been tested, but compatibility is not guaranteed.

```
./bin/tornadovm-installer --listJDKs
jdk21            : Install TornadoVM with OpenJDK 21 (Oracle OpenJDK)
graalvm-jdk-21   : Install TornadoVM with GraalVM and JDK 21 (GraalVM 23.1.0)
mandrel-jdk-21   : Install TornadoVM with Mandrel and JDK 21 (GraalVM 23.1.0)
corretto-jdk-21  : Install TornadoVM with Corretto JDK 21
zulu-jdk-jdk-21  : Install TornadoVM with Azul Zulu JDK 21
temurin-jdk-21   : Install TornadoVM with Eclipse Temurin JDK 21
```

1. Download a JDK 21 distribution

- OpenJDK distributions are available to download at https://adoptium.net/.
- Red Hat Mandrel releases are available at https://github.com/graalvm/mandrel/releases.
- Amazon Coretto releases are available at https://aws.amazon.com/corretto/.
- Microsoft OpenJDK releases are available at https://docs.microsoft.com/en-us/java/openjdk/download. Azul Zulu
- OpenJDK releases are available at https://www.azul.com/downloads.
- Eclipse Temurin releases are available at https://github.com/adoptium/temurin21-binaries/releases/tag/jdk-21.0.1%2B12.

1.1 Manage JDKs manually

After downloading and extracting the JDK distribution, point your `JAVA_HOME` variable to the JDK root.

Example using Amazon Corretto:

```
$ wget https://corretto.aws/downloads/latest/amazon-corretto-21-x64-linux-jdk.tar.gz
$ tar xf amazon-corretto-21-x64-linux-jdk.tar.gz
$ export JAVA_HOME=$PWD/amazon-corretto-21-x64-linux
```

1.2 Manage JDKs with SDKMAN

There is no need to change your `JAVA_HOME` as SDKMAN exports it every time you switch between distributions.

Example using Amazon Corretto:

```
$ sdk install java 21-amzn
$ sdk use java 21-amzn
```

A complete list of all available Java Versions for Linux 64bit can be obtained with:

```
$ sdk list java
```

2. Download TornadoVM

```
$ git clone https://github.com/beehive-lab/TornadoVM tornadovm
$ cd tornadovm
```

Create/edit your configuration file:

```
$ vim etc/sources.env
```

The first time you need to create the `etc/sources.env` file and add the following code in it **(after updating the paths to your correct ones)**:

```
#!/bin/bash
export JAVA_HOME=<path to JDK21> ## This path is produced in Step 1
export PATH=$PWD/bin/bin:$PATH    ## This directory will be automatically generated␣
↪during Tornado compilation
export TORNADO_SDK=$PWD/bin/sdk   ## This directory will be automatically generated␣
↪during Tornado compilation
export CMAKE_ROOT=/usr            ## or <path/to/cmake/cmake-3.10.2> (see step 4)
```

This file should be loaded once after opening the command prompt for the setup of the required paths:

```
$ source ./etc/sources.env
```

For OSX: the exports above may be added to `~/.profile`

3. Install CMAKE (if cmake < 3.6)

For Linux:

```
$ cmake -version
```

If the version of cmake is > 3.6 then skip the rest of this step and go to Step 4. Otherwise try to install cmake.

For simplicity it might be easier to install cmake in your home directory.

- Redhat Enterprise Linux / CentOS use cmake v2.8
- We require a newer version so that OpenCL is configured properly.

```
$ cd ~/Downloads
$ wget https://cmake.org/files/v3.10/cmake-3.10.1-Linux-x86_64.tar.gz
$ cd ~/opt
$ tar -tvf ~/Downloads/cmake-3.10.1-Linux-x86_64.tar.gz
$ mv cmake-3.10.1-Linux-x86_64 cmake-3.10.1
$ export PATH=$HOME/opt/cmake-3.10.1/bin/:$PATH
$ cmake -version
cmake version 3.10.1
```

Then export `CMAKE_ROOT` variable to the cmake installation. You can add it to the `./etc/sources.env` file.

```
$ export CMAKE_ROOT=/opt/cmake-3.10.1
```

For OSX:

Install cmake:

```
$ brew install cmake
```

then

```
export CMAKE_ROOT=/usr/local
```

which can be added to ~/.profile

4. Compile TornadoVM for JDK 21

```
$ cd ~/tornadovm
$ . etc/sources.env
```

To build with a distribution of JDK 21

```
$ make jdk21 BACKEND={ptx,opencl}
```

and done!!

Running with JDK 21

TornadoVM uses modules:

To run examples:

```
$ tornado -m tornado.examples/uk.ac.manchester.tornado.examples.compute.
↪MatrixMultiplication2D --params "512"
```

To run benchmarks:

```
$ tornado -m tornado.benchmarks/uk.ac.manchester.tornado.benchmarks.BenchmarkRunner --
↪params "dft"
```

To run individual tests:

```
tornado --jvm "-Dtornado.unittests.verbose=True -Xmx6g"  -m  tornado.unittests/uk.ac.
↪manchester.tornado.unittests.tools.TornadoTestRunner --params "uk.ac.manchester.
↪tornado.unittests.arrays.TestArrays"
```

TornadoVM for Windows 10/11 using GraalVM

[DISCLAIMER] Please, notice that, although TornadoVM can run on Windows10 via MSys2, it is still experimental.

1. Install prerequisites

Maven

Download Apache Maven (at least 3.9.0) from the official site, and extract it to any location on your computer. Below it's assumed that Maven's home is `C:/bin/`, but you can use any other directory.

MSys2

1. Download the MSys2 installer from the official website and run it. You may choose any installation directory, below it will be referred as <MSYS2>.

IMPORTANT: the only executable you should use as a terminal is <MSYS2>/mingw64.exe.

2. Update MSys2 **system** packages. Start <MSYS2>/mingw64.exe and run the following command in the terminal:

```
pacman -Syu
```

You might need to execute it several times until you see that no updates found.

3. Update MSys2 **default** packages. In the terminal window of <MSYS2>/mingw64.exe run:

```
pacman -Su
```

You might need to execute it several times until you see that no updates found.

4. Install necessary tools to MSys2. In the terminal window of <MSYS2>/mingw64.exe run:

```
pacman -S \
mingw-w64-x86_64-make           \
mingw-w64-x86_64-cmake          \
mingw-w64-x86_64-gcc            \
mingw-w64-x86_64-opencl-headers \
mingw-w64-x86_64-opencl-icd \
python python3-pip make git
```

5. Create default Maven repository for MSys2 user:

```
cd ~
mkdir .m2
```

6. Create default content for the file ~/.m2/settings.xml:

```
cat > ~/.m2/settings.xml << EOF
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/
↪2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.
↪org/xsd/settings-1.0.0.xsd">
    <localRepository/>
```

(continues on next page)

(continued from previous page)

```
    <interactiveMode/>
    <offline/>
    <pluginGroups/>
    <servers/>
    <mirrors/>
    <proxies/>
    <profiles/>
    <activeProfiles/>
</settings>
EOF
```

7. Create file `mvn` in `<MSYS2>/mingw64/bin` with any text editor (e.g., Visual Studio Code) with the following content:

```
#!/usr/bin/env bash
C:/<path-to-your-maven-install>/bin/mvn.cmd --settings ${HOME}/.m2/settings.xml "$@"
```

Example:

```
#!/usr/bin/env bash
C:/bin/apache-maven-3.9.1-bin/apache-maven-3.9.1/bin/mvn.cmd --settings ${HOME}/.m2/
↪settings.xml "$@"
```

You only need to change the path to your maven installation in Windows.

2. Download TornadoVM

Clone the latest TornadoVM source code from the GitHub repository using `<MSYS2>/mingw64.exe`:

```
cd D:/MyProjects
git clone https://github.com/beehive-lab/TornadoVM.git
cd TornadoVM
```

We will refer hereafter the directory with TornadoVM sources as `<TornadoVM>`.

3. Download GraalVM for JDK 21 Community 21.0.1

TornadoVM can run with JDK 21. Visit GraalVM for JDK21 and download the following build:

- Download for JDK 21

Extract the downloaded file to any directory.

4. Install the NVIDIA drivers and CUDA SDK

A) CUDA Driver

Most Windows systems come with the NVIDIA drivers pre-installed. You can check your installation and the latest drivers available by using NVIDIA GEFORCE Experience tool.

Alternatively, all NVIDIA drivers can be found here: NVIDIA Driver Downloads.

B) OpenCL and NVIDIA PTX

If you plan to only use the OpenCL backend from TornadoVM, then you only need the NVIDIA driver from the previous step.

If you want to also use the PTX backend, then you need to install the NVIDIA CUDA Toolkit.

- Complete CUDA Toolkit from CUDA Toolkit Downloads.

It is important to make sure that the GPU drivers are included with the CUDA Toolkit, so you may avoid downloading drivers separately. The only thing to note is that the GPU driver you are currently using should be of the same or higher version than the one shipped with CUDA Toolkit. Thus, if you have the driver already installed, make sure that the version required by the CUDA SDK is same or higher, otherwise update the GPU driver during toolkit installation. Note, that NSight, BLAST libs and Visual Studio integration are irrelevant for TornadoVM builds, you just need the CUDA SDK - so you may skip installing them.

5. Configure the TornadoVM build: setting ENV variables

Using any text editor create file `<TornadoVM>/etc/sources.env` with the following content:

```
#!/bin/bash

# UPDATE PATH TO ACTUAL LOCATION OF THE JDK OR GRAAL
export JAVA_HOME="C:\Users\jjfum\Documents\bin\jvms\graalvm-jdk-21_windows-x64_bin\
↪graalvm-jdk-21+35.1"

## NEXT TWO LINES NECESSARY TO BUILD PTX (NVIDIA CUDA) BACKEND
## COMMENT THEM OUT OR JUST IGNORE IF YOU ARE NOT INTERESTED IN PTX BUILD
## OTHERWISE UPDATE 'CUDA_PATH' WITH ACTUAL VALUE (REMEMBER OF UNIX_STYLE SLASHES AND␣
↪SPACES!!!)
export CUDA_PATH="C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v12.1"
export PTX_LDFLAGS=-L\"$CUDA_PATH/lib/x64\"

# LEAVE THE REST OF FILE 'AS IS'
# DON'T ALTER!
export PATH=$PWD/bin/bin:$PATH                ## This directory will be automatically␣
↪generated during Tornado compilation
export TORNADO_SDK=$PWD/bin/sdk               ## This directory will be automatically␣
↪generated during Tornado compilation
CMAKE_FILE=$(where cmake | head -n 1)
export CMAKE_ROOT=${CMAKE_FILE%\\*\\*}
```

There are only 2 places you should adjust:

1. `JAVA_HOME` path that points to your Graal installation

2. CUDA_PATH pointing to your NVIDIA GPU Computing Toolkit (CUDA) - this one is necessary only for builds with PTX backend.

3. Compile TornadoVM

Start <MSYS2>/mingw64.exe terminal, navigate to the <TornadoVM> directory, and build TornadoVM as follows:

```
cd D:/MyProjects/TornadoVM
source etc/sources.env
make graal-jdk-21 BACKEND=ptx,opencl
```

The BACKEND parameter has to be a comma-separated list of ptx and opencl options. You may build ptx only when NVIDIA GPU Computing Toolkit (CUDA) is installed.

7. Check the installation

Don't close <MSYS2>/mingw64.exe after the build. Run the following command to see that TornadoVM is working:

```
tornado --devices
```

You should see a list of OpenCL and/or CUDA devices available on your system.

Now try to run a simple test. To run examples with Graal JDK 21, TornadoVM uses modules:

```
tornado -m tornado.examples/uk.ac.manchester.tornado.examples.compute.
↪MatrixMultiplication2D --params="512"
```

To run individual tests:

```
tornado --jvm="-Dtornado.unittests.verbose=True -Xmx6g"  -m  tornado.unittests/uk.ac.
↪manchester.tornado.unittests.tools.TornadoTestRunner --params="uk.ac.manchester.
↪tornado.unittests.arrays.TestArrays"
```

To run all unit-tests:

```
make tests
```

TornadoVM on ARM Mali GPUs

Installation

The installation of TornadoVM to run on ARM Mali GPUs requires JDK21 with GraalVM. See the *Installation for GraalVM for JDK 21.0.1 on Linux and OSx* for details about the installation.

The OpenCL driver for Mali GPUs on Linux that has been tested is:

- OpenCL C 2.0 v1.r9p0-01rel0.37c12a13c46b4c2d9d736e0d5ace2e5e: link

Testing on ARM MALI GPUs

We have tested TornadoVM on the following ARM Mali GPUs:

- Mali-G71, which implements the Bifrost architecture: link

Some of the unittests in TornadoVM run with `double` data types. To enable double support, TornadoVM includes the following extension in the generated OpenCL code:

```
cl_khr_fp64
```

However, this extension is not available on Bifrost GPUs.

The rest of the unittests should pass.

Known issues on Linux

For Ubuntu >= 16.04, install the package `ocl-icd-opencl-dev`

In Ubuntu >= 16.04 CMake can cause the following error:

```
Could NOT find OpenCL (missing:  OpenCL_LIBRARY) (found version "2.2").
```

Then the following package should be installed:

```
$ apt-get install ocl-icd-opencl-dev
```

Known issues on Windows

1. If you already have MSys2 installed and heavily customized you may experience issues with build or tests. We are suggesting to start with fresh MSys2 installation in this case and follow the instructions above. Most notably, make sure that you have no `mingw-w64-x86_64-python` installed - it prevents Python scripts that execute tests from running. Also, make sure that you have updated all GCC / Make / CMake packages mentioned.
2. If you see no output from `tornado --devices` this may be either of 2 reasons: - OpenCL / CUDA is misconfigured. Download any third-party tool for OpenCL / CUDA capabilities viewing and check that you can see your devices there. Sometimes order of installation of different OpenCL drivers matters - Intel OpenCL SDK may shadow NVIDIA OpenCL and alike. - You build native code of the library using wrong compiler, most probably you ran `<MSYS2>/msys2.exe` terminal instead of `<MSYS2>/mingw64.exe` . Please re-try with correct terminal (and therefore GCC) version.
3. If you see JVM crashes or `UnsatisfiedLinkError` or some `Error initializing DLL` during `tornado --devices` execution than it's definitely due to wrong GCC (and hence terminal) version used during build.

IDE Code Formatter

Using Eclipse and Netbeans

The code formatter in Eclipse is automatically applied after generating the setting files.

```
$ mvn eclipse:eclipse
$ python scripts/eclipseSetup.py
```

For Netbeans, the Eclipse Formatter Plugin is needed.

Using IntelliJ

Install plugins: - Eclipse Code Formatter - Save Actions

Then : 1. Open File > Settings > Eclipse Code Formatter 2. Check the `Use the Eclipse code` formatter radio button 3. Set the Eclipse Java Formatter config file to the XML file stored in `/scripts/templates/eclise-settings/Tornado.xml`. 4. Set the Java formatter profile in Tornado

TornadoVM Maven Projects

To use the TornadoVM API in your projects, you can checkout our maven repository as follows:

```
<repositories>
  <repository>
    <id>universityOfManchester-graal</id>
    <url>https://raw.githubusercontent.com/beehive-lab/tornado/maven-tornadovm</url>
  </repository>
</repositories>

<dependencies>
   <dependency>
      <groupId>tornado</groupId>
      <artifactId>tornado-api</artifactId>
      <version>1.0.2</version>
   </dependency>

   <dependency>
      <groupId>tornado</groupId>
      <artifactId>tornado-matrices</artifactId>
      <version>1.0.2</version>
   </dependency>
</dependencies>
```

Notice that, for running with TornadoVM, you will need either the docker images or the full JVM with TornadoVM enabled.

Versions available

- 1.0.2
- 1.0.1
- 1.0
- 0.15.2
- 0.15.1
- 0.15
- 0.14.1
- 0.14
- 0.13
- 0.12

- 0.11
- 0.10
- 0.9
- 0.8
- 0.7
- 0.6
- 0.5
- 0.4
- 0.3
- 0.2
- 0.1.0

1.3 Running Examples and Benchmarks

1.3.1 Running TornadoVM Programs

TornadoVM includes a tool for launching applications from the command-line:

```
$ tornado --help
usage: tornado [-h] [--version] [-version] [--debug] [--threadInfo] [--igv] [--
↪igvLowTier] [--printKernel] [--printBytecodes] [--enableProfiler ENABLE_PROFILER] [--
↪dumpProfiler DUMP_PROFILER] [--printJavaFlags] [--devices] [--ea]
              [--module-path MODULE_PATH] [--classpath CLASSPATH] [--jvm JVM_OPTIONS] [-m␣
↪MODULE_APPLICATION] [-jar JAR_FILE] [--params APPLICATION_PARAMETERS]
              [application]

Tool for running TornadoVM Applications. This tool sets all Java options for enabling␣
↪TornadoVM.

positional arguments:
  application

optional arguments:
  -h, --help            show this help message and exit
  --version             Print version of TornadoVM
  -version              Print JVM Version
  --debug               Enable debug mode
  --threadInfo          Print thread deploy information per task on the accelerator
  --igv                 Debug Compilation Graphs using Ideal Graph Visualizer (IGV)
  --igvLowTier          Debug Low Tier Compilation Graphs using Ideal Graph Visualizer␣
↪(IGV)
  --printKernel, -pk    Print generated kernel (OpenCL, PTX or SPIR-V)
  --printBytecodes, -pc
                        Print the generated TornadoVM bytecodes
  --enableProfiler ENABLE_PROFILER
                        Enable the profiler {silent|console}
```

(continues on next page)

(continued from previous page)

```
  --dumpProfiler DUMP_PROFILER
                        Dump the profiler to a file
  --printJavaFlags      Print all the Java flags to enable the execution with TornadoVM
  --devices             Print information about the accelerators available
  --ea, -ea             Enable assertions
  --module-path MODULE_PATH
                        Module path option for the JVM
  --classpath CLASSPATH, -cp CLASSPATH, --cp CLASSPATH
                        Set class-path
  --jvm JVM_OPTIONS, -J JVM_OPTIONS
                        Pass Java options to the JVM. Use without spaces: e.g., --jvm="-
↪Xms10g" or -J"-Xms10g"
  -m MODULE_APPLICATION
                        Application using Java modules
  -jar JAR_FILE         Main Java application in a JAR File
  --params APPLICATION_PARAMETERS
                       Command-line parameters for the host-application. Example: --
↪params="param1 param2..."
```

Some examples:

```
$ tornado -m tornado.examples/uk.ac.manchester.tornado.examples.compute.
↪MatrixMultiplication1D
```

Use the following command to identify the `ids` of the devices that are being discovered by TornadoVM:

```
$ tornado --devices
```

Every device discovered by TornadoVM is associated with a pair of `id` numbers that correspond to the type of driver and the specific device, as follows:

```
Tornado device=<driverNumber>:<deviceNumber>
```

Example output:

```
Number of Tornado drivers: 2
Total number of PTX devices  : 1
Tornado device=0:0
  PTX -- GeForce GTX 1650
    Global Memory Size: 3.8 GB
    Local Memory Size: 48.0 KB
    Workgroup Dimensions: 3
    Max WorkGroup Configuration: [1024, 1024, 64]
    Device OpenCL C version: N/A

Total number of OpenCL devices  : 4
Tornado device=1:0
  NVIDIA CUDA -- GeForce GTX 1650
    Global Memory Size: 3.8 GB
    Local Memory Size: 48.0 KB
    Workgroup Dimensions: 3
    Max WorkGroup Configuration: [1024, 1024, 64]
    Device OpenCL C version: OpenCL C 1.2
```

(continues on next page)

(continued from previous page)

```
Tornado device=1:1
  Intel(R) OpenCL HD Graphics -- Intel(R) Gen9 HD Graphics NEO
    Global Memory Size: 24.8 GB
    Local Memory Size: 64.0 KB
    Workgroup Dimensions: 3
    Max WorkGroup Configuration: [256, 256, 256]
    Device OpenCL C version: OpenCL C 2.0

Tornado device=1:2
        Intel(R) OpenCL -- Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
              Global Memory Size: 31.0 GB
              Local Memory Size: 32.0 KB
              Workgroup Dimensions: 3
              Max WorkGroup Configuration: [8192, 8192, 8192]
              Device OpenCL C version: OpenCL C 1.2

Tornado device=1:3
        AMD Accelerated Parallel Processing -- Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
              Global Memory Size: 31.0 GB
              Local Memory Size: 32.0 KB
              Workgroup Dimensions: 3
              Max WorkGroup Configuration: [1024, 1024, 1024]
              Device OpenCL C version: OpenCL C 1.2
```

The output might vary depending on which backends you have included in the build process. To run TornadoVM, you should see at least one device.

To run on a specific device use the following option:

```
-D<g>.<t>.device=<driverNumber>:<deviceNumber>
```

Where `g` is the *TaskGraph name* and `t` is the *task name*.

For example running on `driver:device 1:1` (Intel HD Graphics in our example) will look like this:

```
$ tornado --jvm="-Ds0.t0.device=1:1" -m tornado.examples/uk.ac.manchester.tornado.
↪examples.compute.MatrixMultiplication1D
```

The command above will run the MatrixMultiplication1D example on the integrated GPU (Intel HD Graphics).

1.3.2 Benchmarking

Running all benchmarks with default values

```
$ tornado-benchmarks.py
Running TornadoVM Benchmarks
[INFO] This process takes between 30-60 minutes
List of benchmarks:
       *saxpy
       *addImage
       *stencil
```

(continues on next page)

(continued from previous page)

```
      *convolvearray
      *convolveimage
      *blackscholes
      *montecarlo
      *blurFilter
      *renderTrack
      *euler
      *nbody
      *sgemm
      *dgemm
      *mandelbrot
      *dft
[INFO] TornadoVM options: -Xms24G -Xmx24G -server
...
```

Running a specific benchmark

```
$ tornado -m tornado.benchmarks/uk.ac.manchester.tornado.benchmarks.BenchmarkRunner --
↪params="sgemm"
```

1.3.3 Unittests

To run all unittests in Tornado:

```
$ make tests
```

To run an individual unittest:

```
$  tornado-test uk.ac.manchester.tornado.unittests.TestHello
```

Also, it can be executed in verbose mode:

```
$ tornado-test --verbose uk.ac.manchester.tornado.unittests.TestHello
```

To test just a method of a unittest class:

```
$ tornado-test --verbose uk.ac.manchester.tornado.unittests.TestHello#testHello
```

To see the OpenCL/PTX generated kernel for a unittest:

```
$ tornado-test --verbose -pk uk.ac.manchester.tornado.unittests.TestHello#testHello
```

To execute in debug mode:

```
$ tornado-test --verbose --debug uk.ac.manchester.tornado.unittests.TestHello#testHello
task info: s0.t0
        platform          : NVIDIA CUDA
        device            : GeForce GTX 1050 CL_DEVICE_TYPE_GPU (available)
        dims              : 1
        global work offset: [0]
```

(continues on next page)

(continued from previous page)

```
        global work size  : [8]
        local  work size  : [8]
```

1.4 Core Programming

TornadoVM exposes to the programmer task-level, data-level and pipeline-level parallelism via a light Application Programming Interface (API). In addition, TornadoVM uses single-source property, in which the code to be accelerated and the host code live in the same Java program.

Programming in TornadoVM involves the development of four parts:

1. **Data Representation:** TornadoVM offers an API to efficiently allocate data off-heap. These data is automatically managed by the TornadoVM Runtime and the compiler.
2. **Expressing parallelism within Java methods:** TornadoVM offers two APIs: one for loop parallelization using Java annotations; and a second one for low-level programming using a Kernel API. Developers can choose which one to use. The loop API is recommended for non-expert GPU/FPGA programmers. The kernel API is recommended for experts GPU programmers than want more control (access to GPU's local memory, barriers, etc.).
3. **Selecting the methods to be accelerated using a Task-Graph API:** once Java methods have been identified for acceleration (either using the loop parallel API or kernel API), Java methods can be grouped together in a graph. TornadoVM offers an API to define the data as well as the Java methods to be accelerated.
4. Building an **Execution Plan**: From the task-graphs, developers can accelerate all methods that are indicate in that graph on an accelerator. Additionally, through an execution plan in TornadoVM, developers can change the way TornadoVM offloads and runs the code (e.g., by selecting a specific GPU, enabling the profiler, etc.).

1.4.1 1. Data Representation

TornadoVM offers a set of off-heap types that encapsulate a Memory Segment, a contiguous region of memory outside the Java heap. Below is a list of the native array types, with an arrow pointing from the on-heap primitive array types to their off-heap equivalent in TornadoVM.

- `int[]` -> `IntArray`
- `float[]` -> `FloatArray`
- `double[]` -> `DoubleArray`
- `long[]` -> `LongArray`
- `char[]` -> `CharArray`
- `short[]` -> `ShortArray`
- `byte[]` -> `ByteArray`

To allocate off-heap memory using the TornadoVM API, each type offers a constructor with one argument that indicates the number of elements that the Memory Segment will contain.

E.g.:

```
// allocate an off-heap memory segment that will contain 16 int values
IntArray intArray = new IntArray(16);
```

Additionally, developers can create an instance of a TornadoVM native array by invoking factory methods for different data representations. In the following examples we will demonstrate the API functions for the `FloatArray` type, but the same methods apply for all support native array types.

```
// from on-heap array to TornadoVM native array
public static FloatArray fromArray(float[] values);
// from individual items to TornadoVM native array
public static FloatArray fromElements(float... values);
// from Memory Segment to TornadoVM native array
public static FloatArray fromSegment(MemorySegment segment);
```

The main methods that the off-heap types expose to manage the Memory Segment of each type are presented in the list below.

```
public void set(int index, float value) // sets a value at a specific index
   E.g.:
       FloatArray floatArray = new FloatArray(16);
       floatArray.set(0, 10.0f); // at index 0 the value is 10.0f
public float get(int index) // returns the value of a specific index
   E.g.:
       FloatArray floatArray = FloatArray.fromArray(new float[] {2.0f, 1.0f, 2.0f, 5.0f}
↪);
       float floatValue = floatArray.get(3); // returns 5.0f
public void clear() // sets the values of the segment to 0
   E.g.:
       FloatArray floatArray = new FloatArray(1024);
       floatArray.clear(); // the floatArray contains 1024 zeros
public void init(float value) // initializes the segment with a specific value
   E.g.:
       FloatArray floatArray = new FloatArray(1024);
       floatArray.init(1.0f); // the floatArray contains 1024 ones
public int getSize() // returns the number of elements in the segment
   E.g.:
       FloatArray floatArray = new FloatArray(16);
       int size = floatArray.getSize(); // returns 16
public float[] toHeapArray(); // Converts the data from off-heap to on-heap
public long getNumBytesOfSegment(); // Returns the total number of bytes the underlying␣
↪Memory Segment occupies, including the header bytes
public long getNumBytesWithoutHeader(); // Returns the total number of bytes the␣
↪underlying Memory Segment occupies, excluding the header bytes
```

NOTE: The methods `init()` and `clear()` are essential because, contrary to their counterpart primitive arrays which are initialized by default with 0, the new types contain garbage values when first created.

1.4.2 2. Expressing Parallelism within Java Methods

TornadoVM offloads Java methods to heterogeneous hardware such as GPUs and FPGAs for parallel execution. Those Java methods usually represents the sequential (single thread) implementation of the work to perform on the accelerator. However, TornadoVM does not auto-parallelize Java methods.

Thus, TornadoVM needs a hint about how to parallelize the code. TornadoVM has two APIs to achieve this goal: one for loop parallelization using Java annotations; and a second one for low-level programming using a Kernel API. Developers can choose which one to use. The loop API is recommended for non-expert GPU/FPGA programmers.

Loop Parallel API

Compute kernels are written in a sequential form (tasks programmed for a single thread execution). To express parallelism, TornadoVM exposes two annotations that can be used in loops and parameters: a) `@Parallel` for annotating parallel loops; and b) `@Reduce` for annotating parameters used in reductions.

The following code snippet shows a full example to accelerate Matrix-Multiplication using TornadoVM and the loop-parallel API: The two outermost loops can be parallelizable because there are no data dependencies across different iterations. Therefore, we can annotate these two loops. Note that, since TornadoVM maps parallel loops to Parallel ND-Range for OpenCL, CUDA and SPIR-V, developers can benefit from 1D (annotating one parallel loop), 2D (annotating two consecutive parallel loops) and 3D (annotating 3 consecutive parallel loops) in their Java methods.

```
public class Compute {
    private static void mxmLoop(Matrix2DFloat A, Matrix2DFloat B, Matrix2DFloat C, final␣
↪int size) {
        for (@Parallel int i = 0; i < size; i++) {
            for (@Parallel int j = 0; j < size; j++) {
                float sum = 0.0f;
                for (int k = 0; k < size; k++) {
                    sum += A.get(i, k) * B.get(k, j);
                }
                C.set(i, j, sum);
            }
        }
    }

    public void run(Matrix2DFloat A, Matrix2DFloat B, Matrix2DFloat C, final int size) {
        TaskGraph taskGraph = new TaskGraph("s0")
                .transferToDevice(DataTransferMode.FIRST_EXECUTION, A, B) // Transfer␣
↪data from host to device and mark buffers as read-only,
                                                                          // since data␣
↪will be transferred only during the first execution.
                .task("t0", Compute::mxmLoop, A, B, C, size)              // Each task␣
↪points to an existing Java method
                .transferToHost(DataTransferMode.EVERY_EXECUTION, C);     // Transfer␣
↪data from device to host in every execution.

        // Create an immutable task-graph
        ImmutableTaskGraph immutableTaskGraph = taskGraph.snaphot();

        // Create an execution plan from an immutable task-graph
        TornadoExecutionPlan executionPlan = new␣
↪TornadoExecutionPlan(immutableTaskGraph);
```

(continues on next page)

(continued from previous page)

```
        // Execute the execution plan
        TorandoExecutionResult executionResult = executionPlan.execute();
    }
}
```

The code snippet shows a complete example, using the Loop Parallel annotations, the Task Graphs and the execution plan. This document explains each part.

Kernel API

Another way to express compute-kernels in TornadoVM is via the kernel API. To do so, TornadoVM exposes a `KernelContext` with which the application can directly access the thread-id, allocate memory in local memory (shared memory on NVIDIA devices), and insert barriers. This model is similar to programming compute-kernels in OpenCL and CUDA. Therefore, this API is more suitable for GPU/FPGA expert programmers that want more control or want to port existing CUDA/OpenCL compute kernels into TornadoVM.

The following code-snippet shows the Matrix Multiplication example using the kernel-parallel API:

```
public class Compute {
    private static void mxmKernel(KernelContext context, Matrix2DFloat A, Matrix2DFloat␣
↪B, Matrix2DFloat C, final int size) {
        int idx = context.threadIdx;
        int jdx = context.threadIdy;
        float sum = 0;
        for (int k = 0; k < size; k++) {
            sum += A.get(idx, k) * B.get(k, jdx);
        }
        C.set(idx, jdx, sum);
    }

    public void run(Matrix2DFloat A, Matrix2DFloat B, Matrix2DFloat C, final int size) {
        // When using the kernel-parallel API, we need to create a Grid and a Worker

        WorkerGrid workerGrid = new WorkerGrid2D(size, size);     // Create a 2D Worker
        GridScheduler gridScheduler = new GridScheduler("s0.t0", workerGrid);  // Attach␣
↪the worker to the Grid
        KernelContext context = new KernelContext();              // Create a context
        workerGrid.setLocalWork(32, 32, 1);                       // Set the local-group␣
↪size

        TaskGraph taskGraph = new TaskGraph("s0")
                .transferToDevice(DataTransferMode.FIRST_EXECUTION, A, B) // Transfer␣
↪data from host to device only during the first execution
                .task("t0", Compute::mxmKernel, context, A, B, C, size)   // Each task␣
↪points to an existing Java method
                .transferToHost(DataTransferMode.EVERY_EXECUTION, C);     // Transfer␣
↪data from device to host in every execution.
        // Create an immutable task-graph
        ImmutableTaskGraph immutableTaskGraph = taskGraph.snaphot();

        // Create an execution plan from an immutable task-graph
        TornadoExecutionPlan executionPlan = new␣
```

(continues on next page)

(continued from previous page)

```
↪TornadoExecutionPlan(immutableTaskGraph);

        // Execute the execution plan
        executionPlan.withGridScheduler(gridScheduler)
                     .execute();
    }
}
```

Kernel Context

`KernelContext` is a Java object exposed by the TornadoVM API to the developers in order to leverage Kernel Parallel Programming using the existing `TaskGraph` API. An instance of the `KernelContext` object is passed to each task that uses the kernel-parallel API.

Additionally, for all tasks using the `KernelContext` object, the user must provide a Grid of execution threads to run on the parallel device. This grid of threads is similar to the number of threads to be launched using CUDA or OpenCL (Number of threads per block and number of blocks). Examples can be found in the `Grid` unit-tests.

KernelContext Features

The following table presents the available features that TornadoVM exposes in Java along with the respective OpenCL and CUDA PTX terminology.

```
// Note:
kc = new KernelContext();
```

TornadoVM KernelContext	OpenCL	PTX
kc.globalIdx	get_global_id(0)	blockIdx * blockDim.x + threadIdx
kc.globalIdy	get_global_id(1)	blockIdy * blockDim.y + threadIdy
kc.globalIdz	get_global_id(2)	blockIdz * blockDim.z + threadIdz
kc.getLocalGroupSize()	get_local_size()	blockDim
kc.localBarrier()	barrier(CLK_LOCAL_MEM_FENCE)	barrier.sync
kc.globalBarrier()	barrier(CLK_GLOBAL_MEM_FENCE)	barrier.sync
int[] array = kc.allocateIntLocalArray(size)	__local int array[size]	.shared .s32 array[size]
float[] array = kc.allocateFloatLocalArray(size)	__local float array[size]	.shared .s32 array[size]
long[] array = kc.allocateLongLocalArray(size)	__local long array[size]	.shared .s64 array[size]
double[] array = kc.allocateDoubleLocalArray(size)	__local double array[size]	.shared .s64 array[size]

Example

The following example is the Matrix Multiplication implementation using the `KernelContext` object for indexing threads and access to local memory. The following example also makes use of loop tiling. There are three main steps to leverage the features of the `KernelContext`:

1. The `KernelContext` object is passed as an argument in the method that will be accelerated. This implementation follows the OpenCL implementation description provided in https://github.com/cnugteren/myGEMM.

```
public static void matrixMultiplication(KernelContext context,
                                        final FloatArray A, final FloatArray B,
                                        final FloatArray C, final int size) {

    // Index thread in the first dimension ( get_global_id(0) )
    int row = context.localIdx;

    // Index thread in the second dimension ( get_global_id(1) )
    int col = context.localIdy;

    int globalRow = TS * context.groupIdx + row;
    int globalCol = TS * context.groupIdy + col;

    // Create Local Memory via the context
    float[] aSub = context.allocateFloatLocalArray(TS * TS);
    float[] bSub = context.allocateFloatLocalArray(TS * TS);

    float sum = 0;

    // Loop over all tiles
    int numTiles = size/TS;
    for(int t = 0; t < numTiles; t++){

        // Load one tile of A and B into local memory
        int tiledRow = TS * t + row;
        int tiledCol = TS * t + col;
        aSub[col * TS + row] = A.get(tiledCol * size + globalRow);
        bSub[col * TS + row] = B.get(globalCol * size + tiledRow);

        // Synchronise to make sure the tile is loaded
        context.localBarrier();

        // Perform the computation for a single tile
        for(int k = 0; k < TS; k++) {
            sum += aSub[k* TS + row] * bSub[col * TS + k];
        }
        // Synchronise before loading the next tile
        context.globalBarrier();
    }

    // Store the final result in C
    C.set((globalCol * size) + globalRow, sum);
}
```

2. A TornadoVM program that uses the `KernelContext` must use the context with a `WorkerGrid` (1D/2D/3D). This is necessary in order to obtain awareness about the dimensions and the sizes of the threads that will be

deployed. Therefore, `KernelContext` can only work with tasks that are linked with a `GridScheduler`.

```
// Create 2D Grid of Threads with a 2D Worker
WorkerGrid workerGrid = new WorkerGrid2D(size, size);

// Create a GridScheduler that associates a task-ID with a worker grid
GridScheduler gridScheduler = new GridScheduler("s0.t0", workerGrid);

// Create the TornadoVM Context
KernelContext context = new KernelContext();

// [Optional] Set the local work size
workerGrid.setLocalWork(32, 32, 1);
```

3. Create a `TornadoExecutionPlan` and execute:

```
TaskGraph tg = new TaskGraph("s0")
      .transferToDevice(DataTransferMode.EVERY_EXECUTION, matrixA, matrixB)
      .task("t0",MxM::compute,context, matrixA, matrixB, matrixC, size)
      .transferToHost(DataTransferMode.EVERY_EXECUTION, matrixC);

ImmutableTaskGraph immutableTaskGraph = tg.snapshot();
TornadoExecutionPlan executionPlan = new TornadoExecutionPlan(immutableTaskGraph);

executionPlan.withGridScheduler(gridScheduler)  // Pass the GridScheduler in the execute␣
↪method
             .execute();
```

Running Multiple Tasks with the Kernel Context

The TornadoVM Task-Graph can be composed of multiple tasks which can either exploit the `KernelContext` features or adhere to the original TornadoVM annotations (`@Parallel`, `@Reduce`).

The following code snippet shows an example of how to combine tasks that require a `KernelContext` object with tasks that do not need a `KernelContext`.

```
WorkerGrid worker = new WorkerGrid1D(size);

GridScheduler gridScheduler = new GridScheduler();
gridScheduler.setWorkerGrid("s02.t0", worker);
gridScheduler.setWorkerGrid("s02.t1", worker);

// Create a Kernel Context object
KernelContext context = new KernelContext();

TaskGraph taskGraph = new TaskGraph("s02") //
   .transferToDevice(DataTransferMode.EVERY_EXECUTION, a, b) //
   .task("t0", TestCombinedTaskGraph::vectorAddV2, context, a, b, cTornado) //
   .task("t1", TestCombinedTaskGraph::vectorMulV2, context, cTornado, b, cTornado) //
   .task("t2", TestCombinedTaskGraph::vectorSubV2, context, cTornado, b) //
   .transferToHost(DataTransferMode.EVERY_EXECUTION, cTornado);

ImmutableTaskGraph immutableTaskGraph = taskGraph.snapshot();
```

(continues on next page)

(continued from previous page)

```
TornadoExecutionPlan executor = new TornadoExecutionPlan(immutableTaskGraph);

executor.withGridScheduler(gridScheduler) //
        .execute();
```

In this test case, each of the first two tasks uses a separate `WorkerGrid`. The third task does not use a `WorkerGrid`, and it relies on the TornadoVM Runtime for the scheduling of the threads.

You can see more examples on GitHub.

1.4.3 3. Selecting the methods to be accelerated using a Task-Graph API

A `TaskGraph` is an TornadoVM object that defines and identify which Java methods to be accelerated and the data involved. Task-graph defines data to be copied in, and out of the accelerator as well as all tasks (Java methods) to be accelerated. Note that a `TaskGraph` object does not compute/move data, but rather annotates what to do when the computation in launched. As we will see in Step 3, a task-graph is only executed through an execution plan.

The following code snippet shows how to instantiate a `TaskGraph` TornadoVM object.

```
TaskGraph taskGraph = new TaskGraph(""name");
```

A. Defining copies from the host (main CPU) to the device (accelerator).

The Task-Graph API also defines a method, named `transferToDevice` to set which arrays need to be copied to the target accelerator. This method receives two types of arguments:

1. Data Transfer Mode: a. `EVERY_EXECUTION`: Data is copied from host to device every time a task-graph is executed by an execution plan. b. `FIRST_EXECUTION`: Data is only copied the first time a task-graph is executed by an execution plan.
2. All input arrays needed to be copied from the host to the device.

The following code snippet sets two arrays (a, b) to be copied from the host to the device every time a task-graph is executed.

```
taskGraph.transferToDevice(DataTransferMode.EVERY_EXECUTION, a, b);
```

Note that this call is only used for the definition of the data flow across multiple tasks in a task-graph, and there are no data copies involved. The TornadoVM runtime stores which data are associated with each data transfer mode and the actual data transfers take place only during the execution by the execution plan.

B. Code definition

To identify which Java methods, from all existing Java methods in a Java program, to accelerate. This is performed using the `task` API call as follows:

```
taskGraph.task("sample", Class::method, param1, param2);
```

- The first parameter sets an ID to the task. This is useful if developers want to change device, or other runtime parameters, from the command line.
- The second parameter is a reference (or a Java lambda expression), to an existing Java method.
- The rest of the parameters correspond to the function call parameters, as if the method were invoked.

Developers can add as many tasks as needed. The maximum number of tasks depends on the amount of code that can be shipped to the accelerator. Usually, FPGAs are more limited than GPUs.

C. Copy out from the device (accelerator) to the host (main CPU).

Similar to `transferToDevice`, the `TaskGraph` API also offers a call to sync the data back from the device to the host. The API call is `transferToHost` with the following parameters:

1. Data Transfer Mode: a. `EVERY_EXECUTION`: Data is copied from the device to the host every time a task-graph is executed by an execution plan. b. `USER_DEFINED:` Data is only copied by an execution result under demand. This is an optimization if developers plan to execute the task-graph multiple times and do not want to copy the results every time the execution plan is launched.
2. All output arrays to be copied from the device to the host.

Example:

```
taskGraph.transferToHost(DataTransferMode.EVERY_EXECUTION, output1, output2);
```

1.4.4 4. Execution Plans

The last step is the creation of an execution plan. An execution plan receives a list of immutable task graphs ready to be executed, as follows:

```
TornadoExecutionPlan executionPlan = new TornadoExecutionPlan(itg);
```

What can we do with an execution plan?

We can execute an execution plan directly, and TornadoVM will apply a list of default optimisations (e.g., it will run on the default device, using the default thread scheduler).

```
executionPlan.execute();
```

How can we optimize an execution plan?

The execution plan offers a set of methods that developers can use to optimize different execution plans. Note that the execution plan operates over all immutable task graphs given in the constructor. Therefore, all immutable task graphs will be executed on the same device in order.

Example:

```
executionPlan.withProfiler(ProfilerMode.SILENT) // Enable Profiling
    .withWarmUp() //  Perform a warmup (compile and code and install it in a code-cache).
    .withDevice(device); Select a specific device
```

And then:

```
executionPlan.execute();
```

Obtain the result and the profiler

Every time an execution plan is executed, a new object of type `TornadoExecutionResult` is created.

```
TornadoExecutionResult executionResult = executionPlan.execute();
```

From the execution result, developers can obtain the result of the TornadoVM profiler:

```
executionResult.getProfilerResult();
```

And query the values of the profiling report. Note that the TornadoVM profiler works only if enabled in the execution plan (via the `withProfiler` method).

1.4.5 Parallel Reductions

TornadoVM now supports basic reductions for `int`, `long`,`float` and `double` data types for the operators + and *, `max` and `min`. Examples can be found in the `examples/src/main/java/uk/ac/manchester/tornado/unittests/reductions` directory on GitHub.

TornadoVM exposes the Java annotation `@Reduce` to represent parallel reductions. Similarly to the `@Parallel` annotation, the `@Reduce` annotation is used to identify parallel sections in Java sequential code. The annotations is used for method parameter in which reductions must be applied. This is similar to OpenMP and OpenACC.

Example:

```
public static void reductionAddFloats(FloatArray input, @Reduce FloatArray result) {
    for (@Parallel int i = 0; i < input.length; i++) {
        result.set(0, result.get(0) + input.get(i));
    }
 }
```

The code is very similar to a Java sequential reduction but with `@Reduce` and `@Parallel` annotations. The `@Reduce` annotation is associated with a variable, in this case, with the `result` float array. Then, we annotate the loop with `@Parallel`. The OpenCL/PTX JIT compilers generate OpenCL/PTX parallel version for this code that can run on GPU and CPU.

Creating reduction tasks

TornadoVM generates different OpenCL/SPIR-V code depending on the target device. Internally, if the target is a GPU, TornadoVM performs full and parallel reductions using the threads within the same OpenCL work-group. If the target is a CPU, TornadoVM performs full reductions within the same thread-id. Besides, TornadoVM automatically resizes the output variables according to the number of work-groups and threads selected.

For PTX code generation, TornadoVM will always perform full and parallel reductions using the threads within the same CUDA block.

```
FloatArray input = new FloatArray(SIZE);
FloatArray result = new FloatArray(1);

Random r = new Random();
IntStream.range(0, SIZE).sequential().forEach(i -> {
    input.set(i, r.nextFloat());
});
```

(continues on next page)

(continued from previous page)

```
TaskGraph taskGraph = new TaskGraph("s0") //
            .transferToDevice(DataTransferMode.EVERY_EXECUTION, input) //
            .task("t0", TestReductionsFloats::reductionAddFloats, input, result) //
            .transferToHost(DataTransferMode.EVERY_EXECUTION, result);

ImmutableTaskGraph immutableTaskGraph = taskGraph.snapshot();
TornadoExecutionPlan executionPlan = new TornadoExecutionPlan(immutableTaskGraph);
executionPlan.execute();
```

Map/Reduce

This section shows an example of how to perform map/reduce operations with TornadoVM. Each of the operations corresponds to a task as follows in the next example:

```
public class ReduceExample {
    public static void map01(IntArray a, IntArray b, IntArray c) {
        for (@Parallel int i = 0; i < a.length; i++) {
            c.set(i, a.get(i) + b.get(i));
        }
    }

    public static void reduce01(IntArray c, @Reduce IntArray result) {
        result.set(0, 0);
        for (@Parallel int i = 0; i < c.length; i++) {
            result.set(0, result.get(0) + c.get(i));
        }
    }

    public void testMapReduce() {
         IntArray a = new IntArray(BIG_SIZE);
         IntArray b = new IntArray(BIG_SIZE);
         IntArray c = new IntArray(BIG_SIZE);
         IntArray result = IntArray.fromElements(0);

         IntStream.range(0, BIG_SIZE).parallel().forEach(i -> {
            a[i] = 10;
            b[i] = 2;
         });

         TaskGraph taskGraph = new TaskGraph("s0") //
             .transferToDevice(DataTransferMode.EVERY_EXECUTION, a, b, c) //
             .task("t0", TestReductionsIntegers::map01, a, b, c) //
             .task("t1", TestReductionsIntegers::reduce01, c, result) //
             .transferToHost(DataTransferMode.EVERY_EXECUTION, result);
         ImmutableTaskGraph immutableTaskGraph = taskGraph.snapshot();
         TornadoExecutionPlan executionPlan = new␣
↪TornadoExecutionPlan(immutableTaskGraph);
         TornadoExecutionResult executionResult = executionPlan.execute();
     }
}
```

Reduction with dependencies

TornadoVM also supports reductions using data dependencies. The next example illustrates this case with the PI computation.

```
public static void computePi(FloatArray input, @Reduce FloatArray result) {
    for (@Parallel int i = 1; i < input.length; i++) {
        float value = (float) (Math.pow(-1, i + 1) / (2 * i - 1));
        result.set(0, result.get(0) + value + input.get(i));
    }
}
```

1.4.6 Dynamic Reconfiguration

The dynamic configuration in TornadoVM is the capability to migrate tasks at runtime from one device to another (e.g., from one GPU to another, or from one CPU to GPU, etc). The dynamic reconfiguration is not enabled by default, but it can be easily activated through the execution plan as follows:

```
executionPlan.withDynamicReconfiguration(Policy.PERFORMANCE, DRMode.Parallel)
             .execute();
```

The *withDynamicReconfiguration* call receives two arguments:

1. Policy: dynamically changes devices based on one of the following policies:
    - *PERFORMANCE*: after a warmup of all devices (JIT compilation is excluded). The TornadoVM runtime evaluates the execution for all devices before making a decision.
    - *END_2_END*: best performing device including the warm-up phase (JIT compilation and buffer allocations). The TornadoVM runtime evaluates the execution for all devices before making a decision.
    - *LATENCY*: fastest device to return. The TornadoVM runtime does not evaluate the execution for all devices before making a decision, but rather it switches context with the first device that finishes the execution.

1.4.7 Batch Computing Processing

TornadoVM supports batch processing through the following API call `withBatch` of the `TornadoExecutionPlan` API. This was mainly designed to run big data applications in which host data is much larger than the device memory capacity. In these cases, developers can enable batch processing by simply calling `withBatch` in their execution plans.

```
int size = 2110000000;
FloatArray arrayA = new FloatArray(size); // Array of 8440MB
FloatArray arrayB = new FloatArray(size); // Array of 8440MB

TaskGraph taskGraph = new TaskGraph("s0") //
      .transferToDevice(DataTransferMode.FIRST_EXECUTION, arrayA) //
      .task("t0", TestBatches::compute, arrayA, arrayB) //
      .transferToHost(DataTransferMode.EVERY_EXECUTION, arrayB);

ImmutableTaskGraph immutableTaskGraph = taskGraph.snapshot();
TornadoExecutionPlan executor = new TornadoExecutionPlan(immutableTaskGraph);
executor.withBatch("512MB") // Run in blocks of 512MB
        .execute();
```

The batch method-call receives a Java string representing the batch to be allocated, copied-in and copied-out to/from the heap of the target device.

Examples of allowed sizes are:

```
batch("XMB");   // Express in MB (X is an int number)
batch("ZGB");   // Express in GB (Z is an int number)
```

Current Limitations of Batch Processing

There is a set of limitations with the current implementation of batch processing.

1. All arrays passed to the input methods to be compiled to the target device have to have the total size and element size (e.g. combining FloatArray and IntArray is possible). 2. We only support arrays of primitives that are passed as arguments. This means that scope arrays in batches are not currently supported. 3. All bytecodes make use of the same OpenCL command queue / CUDA stream. 4. Matrix or non-regular batch distributions. (E.g., MxM would need to be split by rows in matrix-A and columns in matrix-B).

1.5 Migrating from v0.15.2 to v1.0

Starting from v1.0, TornadoVM is providing its custom off-heap data types. Below is a list of the new types, with an arrow pointing from the on-heap primitive array types to their off-heap equivalent.

- `int[]` -> `IntArray`
- `float[]` -> `FloatArray`
- `double[]` -> `DoubleArray`
- `long[]` -> `LongArray`
- `char[]` -> `CharArray`
- `short[]` -> `ShortArray`
- `byte[]` -> `ByteArray`

The existing Matrix and Vector collection types that TornadoVM offers (e.g., `VectorFloat`, `Matrix2DDouble`, etc.) have been refactored to use internally these off-heap data types instead of primitive arrays.

1.5.1 1. Off-heap types API

The new off-heap types encapsulate a Memory Segment, a contiguous region of memory outside the Java heap. To allocate off-heap memory using the TornadoVM API, each type offers a constructor with one argument that indicates the number of elements that the Memory Segment will contain.

E.g.:

```
// allocate an off-heap memory segment that will contain 16 int values
IntArray intArray = new IntArray(16);
```

Additionally, developers can create an instance of a TornadoVM native array by invoking factory methods for different data representations. In the following examples we will demonstrate the API functions for the `FloatArray` type, but the same methods apply for all support native array types.

```
// from on-heap array to TornadoVM native array
public static FloatArray fromArray(float[] values);
// from individual items to TornadoVM native array
public static FloatArray fromElements(float... values);
// from Memory Segment to TornadoVM native array
public static FloatArray fromSegment(MemorySegment segment);
```

The main methods that the off-heap types expose to manage the Memory Segment of each type are presented in the list below.

```
public void set(int index, float value) // sets a value at a specific index
   E.g.:
       FloatArray floatArray = new FloatArray(16);
       floatArray.set(0, 10.0f); // at index 0 the value is 10.0f
public float get(int index) // returns the value of a specific index
   E.g.:
       FloatArray floatArray = FloatArray.fromArray(new float[] {2.0f, 1.0f, 2.0f, 5.0f}
↪);
       float floatValue = floatArray.get(3); // returns 5.0f
public void clear() // sets the values of the segment to 0
   E.g.:
       FloatArray floatArray = new FloatArray(1024);
       floatArray.clear(); // the floatArray contains 1024 zeros
public void init(float value) // initializes the segment with a specific value
   E.g.:
       FloatArray floatArray = new FloatArray(1024);
       floatArray.init(1.0f); // the floatArray contains 1024 ones
public int getSize() // returns the number of elements in the segment
   E.g.:
       FloatArray floatArray = new FloatArray(16);
       int size = floatArray.getSize(); // returns 16
public float[] toHeapArray(); // Converts the data from off-heap to on-heap
public long getNumBytesOfSegment(); // Returns the total number of bytes the underlying␣
↪Memory Segment occupies, including the header bytes
public long getNumBytesWithoutHeader(); // Returns the total number of bytes the␣
↪underlying Memory Segment occupies, excluding the header bytes
```

NOTE: The methods `init()` and `clear()` are essential because, contrary to their counterpart primitive arrays which are initialized by default with 0, the new types contain garbage values when first created.

1.5.2 2. Example: Migrating TornadoVM applications from <= 0.15.2 to 1.0

Below is an example of a TornadoVM program that uses primitive arrays:

```
public static void main(String[] args) {
    int[] input = new int[numElements];

    Arrays.fill(input, 10);

    TaskGraph taskGraph = new TaskGraph("s0")
            .transferToDevice(DataTransferMode.FIRST_EXECUTION, input)
            .task("t", Example::add, input, 1)
```

(continues on next page)

(continued from previous page)

```
            .transferToHost(DataTransferMode.EVERY_EXECUTION, input);

    ImmutableTaskGraph immutableTaskGraph = taskGraph.snapshot();
    TornadoExecutionPlan executor = new TornadoExecutor(immutableTaskGraph);
    executor.execute();
}

public static void add(int[] input, int value) {
    for (@Parallel int i = 0; i < input.length; i++) {
        input[i] = input[i] + value;
    }
}
```

Here is how the code above would need to be transformed to use the new data types (the changes are highlighted):

```
 public static void main(String[] args) {
     IntArray input = new IntArray(numElements); // create a new off heap segment of int␣
↪values

     input.init(10); // initialize all the values of the input to be 10

     TaskGraph taskGraph = new TaskGraph("s0")
             .transferToDevice(DataTransferMode.FIRST_EXECUTION, input)
             .task("t", Example::add, input, 1)
             .transferToHost(DataTransferMode.EVERY_EXECUTION, input);

     ImmutableTaskGraph immutableTaskGraph = taskGraph.snapshot();
     TornadoExecutionPlan executor = new TornadoExecutor(immutableTaskGraph);
     executor.execute();
 }

 public static void acc(IntArray input, int value) { // Pass the IntArray as a parameter
     for (@Parallel int i = 0; i < input.getSize(); i++) {
         input.set(i, input.get(i) + value);  // Use the set and get functions access␣
↪data
     }
 }
```

1.6 Polyglot Programming

TornadoVM can be used with the GraalVM Truffle Polyglot API to invoke Task-Graphs from guest programming languages such as Python, Ruby, etc. This guide will describe how to execute TornadoVM programs through code written in Python, JavaScript, and Ruby.

1.6.1 1. Prerequisites

A) Configuration of the JAVA_HOME Variable

To enable polyglot support, the `JAVA_HOME` variable must be set to the GraalVM path. Instructions on how to install TornadoVM with GraalVM can be found here: *Installation for GraalVM for JDK 21.0.1 on Linux and OSx*.

```
$ export JAVA_HOME=<path to GraalVM jdk 21>
```

B) GraalVM Polyglot Dependencies

The implementations of the programming languages (e.g., Python, JavaScript, Ruby) that are supported by the GraalVM polyglot API are now shipped as standalone distributions. And they can either be used as any other Java library from *Maven Central*, or as *standalone toolkits*.

1.6.2 2. Using the GraalVM Polyglot Dependencies from Maven Central

A) Build TornadoVM with the GraalVM Polyglot Dependencies from Maven Central

To build TornadoVM and utilize the maven dependencies for GraalPy, GraalVM JavaScript and TruffleRuby, you can build TornadoVM and use the `graalvm-polyglot` profile.

```
$ make graal-jdk-21 polyglot
```

B) Run the Examples

Now, that TornadoVM is built with the polyglot dependencies, you can run the available examples that exhibit the interoperability of Python, JavaScript and Ruby code from Java.

```
$ tornado --debug -m tornado.examples/uk.ac.manchester.tornado.examples.polyglot.
↪HelloPython
$ tornado --debug -m tornado.examples/uk.ac.manchester.tornado.examples.polyglot.HelloJS
$ tornado --debug -m tornado.examples/uk.ac.manchester.tornado.examples.polyglot.
↪HelloRuby
```

1.6.3 3. Using the GraalVM Polyglot Dependencies as Standalone Toolkits

However, to interoperate from programs written in those programming languages and invoke a Java method, users would use the standalone distributions. However, an aftermath of the last change is that the dedicated builds of GraalVM implemented languages, such as GraalPy, GraalVM JavaScript and TruffleRuby, do not work out-of-the-box with TornadoVM. Instead, users must build those frameworks from source.

A) Build GraalVM Polyglot Dependencies from Source

GraalVM implementations of the programming languages that can interoperate with Java are provided as standalone distributions, e.g., GraalPy, GraalVM JavaScript, TruffleRuby. As detailed in the GraalVM Reference Manuals, the following dependencies must be downloaded for each of the programming languages supported:

To ease users, we outline beneath two steps in order to build each of the programming languages supported and to interoperate with TornadoVM.

Step 1: Build GraalPy, GraalVM JavaScript and TruffleRuby, from source.

- **Python**

```
$ git clone https://github.com/oracle/graalpython.git && cd graalpython && git checkout␣
↪graal-23.1.0
$ git clone https://github.com/graalvm/mx.git mx
$ export PATH=$PWD/mx:$PATH
$ mx fetch-jdk
$ export JAVA_HOME=~/.mx/jdks/labsjdk-ce-21.0.1-jvmci-23.1-b22
$ mx --dy /compiler python-gvm
```

- **JavaScript**

```
$ git clone https://github.com/oracle/graaljs.git && cd graaljs && git checkout graal-23.
↪1.0
$ git clone https://github.com/graalvm/mx.git mx
$ export PATH=$PWD/mx:$PATH
$ mx fetch-jdk
$ export JAVA_HOME=~/.mx/jdks/labsjdk-ce-21.0.1-jvmci-23.1-b22
$ mx --dynamicimports /compiler build
```

- **Ruby**

```
$ git clone https://github.com/oracle/truffleruby.git && cd truffleruby && git checkout␣
↪graal-23.1.0
$ git clone https://github.com/graalvm/mx.git mx
$ export PATH=$PWD/mx:$PATH
$ mx fetch-jdk
$ export JAVA_HOME=~/.mx/jdks/labsjdk-ce-21.0.1-jvmci-23.1-b22
$ mx sforceimports
$ mx --dynamicimports /compiler build
```

Step 2: Set up the suitable variable for each programming language.

Set the `JAVA_HOME` variable to the GraalVM JDK:

```
$ export JAVA_HOME=<path to GraalVM jdk 21>
```

To enable TornadoVM to employ the standalone built distribution of the GraalVM implementations, users must set the following variables.

Note: The following examples show tentantive paths for a Linux environment. If you are using Mac OS X, you should ensure that your path includes the `</Contents/Home>` suffix.

- For Python, set **GRAALPY_HOME**:

```
$ export GRAALPY_HOME=<path-to-graalpy>/../graal/sdk/mxbuild/linux-amd64/GRAALVM_
↪03DCD25EA1_JAVA21/graalvm-03dcd25ea1-java21-23.1.0-dev
```

- For JavaScript, set **GRAALJS_HOME**:

```
$ export GRAALJS_HOME=<path-to-graaljs>/../graal/sdk/mxbuild/linux-amd64/GRAALVM_
↪3AF13F6F38_JAVA21/graalvm-3af13f6f38-java21-23.1.0-dev
```

- For Ruby, set **TRUFFLERUBY_HOME**:

```
$ export TRUFFLERUBY_HOME=<path-to-truffleruby>/../graal/sdk/mxbuild/linux-amd64/GRAALVM_
↪AEA5C30A3B_JAVA21/graalvm-aea5c30a3b-java21-23.1.0-dev
```

B) Interoperate between a Polyglot Programming Language and TornadoVM through Graal's Polyglot API

In the following example, we will iterate over the necessary steps to invoke a TornadoVM computation from Python, JavaScript and Ruby programs, using the `MyCompute` class from the TornadoVM examples module. However, users can create their own Java classes with the code to be accelerated following the TornadoVM API guidelines *Core Programming*.

Step 1: Create a variable that is of the Java class type.

- **Python**

```
myclass = java.type('uk.ac.manchester.tornado.examples.polyglot.MyCompute')
```

- **JavaScript**

```
var myclass = Java.type('uk.ac.manchester.tornado.examples.polyglot.MyCompute')
```

- **Ruby**

```
myclass = Java.type('uk.ac.manchester.tornado.examples.polyglot.MyCompute')
```

Step 2: Use this variable to invoke the Java function that contains the Task-Graph.

In this example, the function is named `compute()` and it performs a matrix multiplication.

- **Python**

```
myclass.compute()
```

- **JavaScript**

```
myclass.compute()
```

- **Ruby**

```
myclass.compute()
```

Step 3: Execute the Ruby/JavaScript/Python program through TornadoVM.

The polyglot program can be executed using the `tornado` command, followed by the `--truffle` option and the language of the program, as follows:

```
$ tornado --truffle python|ruby|js|node <path/to/polyglot/program>
```

All of the existing TornadoVM options (e.g., `--printKernel`, etc.) can be used as always.

C) Run the Examples

The `tornado-assembly/src/examples/polyglotTruffle` directory contains three examples, one for each of the supported languages. These examples can be executed using the `polyglotTests.sh` script.

```
$ ./scripts/polyglotTests.sh
```

- **Python**

```
$ tornado --printKernel --truffle python $TORNADO_SDK/examples/polyglotTruffle/
↪mxmWithTornadoVM.py
```

- **JavaScript**

```
$ tornado --printKernel --truffle js $TORNADO_SDK/examples/polyglotTruffle/
↪mxmWithTornadoVM.js
```

- **Ruby**

```
$ tornado --printKernel --truffle ruby $TORNADO_SDK/examples/polyglotTruffle/
↪mxmWithTornadoVM.rb
```

1.7 TornadoVM Profiler

The TornadoVM profiler can be enabled either from the command line (via a flag from the `tornado` command), or via an `ExecutionPlan` in the source code.

1.7.1 1. Enable the Profiler from the Command Line

To enable the TornadoVM profiler, developers can use `--enableProfiler <silent|console>`.

- `console`: It prints a JSON entry for each task-graph executed via STDOUT.
- `silent` : It enables profiling information in silent mode. Use the profiler API to query the values.

Example:

```
$ tornado --enableProfiler console  -m tornado.examples/uk.ac.manchester.tornado.
↪examples.VectorAddInt --params="100000"
{
    "s0": {
        "TOTAL_BYTE_CODE_GENERATION": "10465146",
        "COPY_OUT_TIME": "24153",
        "COPY_IN_TIME": "72044",
        "TOTAL_DRIVER_COMPILE_TIME": "63298322",
        "TOTAL_DISPATCH_DATA_TRANSFERS_TIME": "0",
        "TOTAL_CODE_GENERATION_TIME": "16564279",
        "TOTAL_TASK_GRAPH_TIME": "285317518",
        "TOTAL_GRAAL_COMPILE_TIME": "109520628",
        "TOTAL_KERNEL_TIME": "47974",
        "TOTAL_COPY_OUT_SIZE_BYTES": "400024",
        "TOTAL_COPY_IN_SIZE_BYTES": "1600096",
        "s0.t0": {
```

(continues on next page)

(continued from previous page)

```
            "BACKEND": "SPIRV",
            "METHOD": "VectorAddInt.vectorAdd",
            "DEVICE_ID": "0:0",
            "DEVICE": "Intel(R) UHD Graphics [0x9bc4]",
            "TASK_COMPILE_GRAAL_TIME": "109520628",
            "TASK_CODE_GENERATION_TIME": "16564279",
            "TASK_COMPILE_DRIVER_TIME": "63298322",
            "TASK_KERNEL_TIME": "47974"
        }
    }
}
```

All timers are printed in nanoseconds.

1.7.2 1. Enabling/Disabling the Profiler using the TornadoExecutionPlan

The profiler can be enable/disable using the *TornadoExecutionPlan* API:

```
// Enable the profiler and print report in STDOUT
executionPlan.withProfiler(ProfilerMode.CONSOLE) //
    .withDevice(device) //
    .withDefaultScheduler()
    .execute();
```

It is also possible to enable the profiler without live reporting in STDOUT and query the profiler after the execution:

```
// Enable the profiler in silent mode
executionPlan.withProfiler(ProfilerMode.SILENT) //
    .withDevice(device) //
    .withDefaultScheduler();

TornadoExecutionResult executionResult = executorPlan.execute();
TornadoProfilerResult profilerResult = executionResult.getProfilerResult();

// Print Kernel Time
System.out.println(profilerResult.getDeviceKernelTime() + " (ns)");
```

1.7.3 Explanation of all values

- *COPY_IN_TIME*: OpenCL timers for copy in (host to device)
- *COPY_OUT_TIME*: OpenCL timers for copy out (device to host)
- *DISPATCH_TIME*: time spent for dispatching a submitted OpenCL command
- *TOTAL_KERNEL_TIME*: It is the sum of all OpenCL kernel timers. For example, if a task-graph contains 2 tasks, this timer reports the sum of execution of the two kernels.
- *TOTAL_BYTE_CODE_GENERATION*: time spent in the Tornado bytecode generation.
- *TOTAL_TASK_GRAPH_TIME*: Total execution time. It contains all timers.
- *TOTAL_GRAAL_COMPILE_TIME*: Total compilation with Graal (from Java. to OpenCL C / PTX)

- *TOTAL_DRIVER_COMPILE_TIME*: Total compilation with the driver (once the OpenCL C / PTX code is generated, the time that the driver takes to generate the final binary).
- *TOTAL_CODE_GENERATION_TIME*: Total code generation time. This value represents the elapsed time from the last Graal compilation phase in the LIR to the target backend code (e.g., OpenCL, PTX or SPIR-V).

Then, for each task within a task-graph, there are usually three timers, one device identifier and two data transfer metrics:

- *BACKEND*: TornadoVM backend selected for the method execution on the target device. It could be either `SPIRV`, `PTX` or `OpenCL`.
- *DEVICE_ID*: platform and device ID index.
- *DEVICE*: device name as provided by the OpenCL driver.
- *TASK_COPY_IN_SIZE_BYTES*: size in bytes of total bytes copied-in for a given task.
- *TASK_COPY_OUT_SIZE_BYTES*: size in bytes of total bytes copied-out for a given task.
- *TASK_COMPILE_GRAAL_TIME*: time that takes to compile a given task with Graal.
- *TASK_COMPILE_DRIVER_TIME*: time that takes to compile a given task with the OpenCL/CUDA driver.
- *TASK_KERNEL_TIME*: kernel execution for the given task (Java method).
- *TASK_CODE_GENERATION_TIME*: time that takes the code generation from the LIR to the target backend code (e.g., SPIR-V).

Note

When the task-graph is executed multiple times (through an execution plan), timers related to compilation will not appear in the Json time-report. This is because the generated binary is cached and there is no compilation after the second iteration.

Print timers at the end of the execution

The options `--enableProfiler silent` print a full report only when the method `ts.getProfileLog()` is called.

Save profiler into a file

Use the option `--dumpProfiler <FILENAME>` to store the profiler output in a JSON file.

Parsing Json files

TornadoVM creates the `profiler-app.json` file with multiple entries for the application (one per task-graph invocation).

TornadoVM's distribution includes a set of utilities for parsing and obtaining statistics:

```
$ createJsonFile.py profiler-app.json output.json
$ readJsonFile.py output.json

['readJsonFile.py', 'output.json']
Processing file: output.json
Num entries = 10
Entry,0
```

(continues on next page)

(continued from previous page)

```
    TOTAL_BYTE_CODE_GENERATION,6783852
    TOTAL_KERNEL_TIME,26560
    TOTAL_TASK_GRAPH_TIME,59962224
    COPY_OUT_TIME,32768
    COPY_IN_TIME,81920
    TaskName, s0.t0
    TASK_KERNEL_TIME,26560
    TASK_COMPILE_DRIVER_TIME,952126
    TASK_COMPILE_GRAAL_TIME,46868099
    TOTAL_GRAAL_COMPILE_TIME,46868099
    TOTAL_DRIVER_COMPILE_TIME,952126
    DISPATCH_TIME,31008
    EndEntry,0

MEDIANS    ### Print median values for each timer
    TOTAL_KERNEL_TIME,25184.0
    TOTAL_TASK_GRAPH_TIME,955967.0
    s0.t0-TASK_KERNEL_TIME,25184.0
    COPY_IN_TIME,74016.0
    COPY_OUT_TIME,32816.0
    DISPATCH_TIME,31008.0
```

1.7.4 Code feature extraction for the OpenCL/PTX generated code

To enable TornadoVM's code feature extraction, use the following flag: `-Dtornado.feature.extraction=True`.

Example:

```
$ tornado --jvm="-Dtornado.feature.extraction=True" -m tornado.examples/uk.ac.manchester.
↪tornado.examples.compute.NBody --params "1024 1"
{
    "nBody": {
        "BACKEND" : "PTX",
        "DEVICE_ID": "0:2",
        "DEVICE": "GeForce GTX 1650",
        "Global Memory Loads":  "15",
        "Global Memory Stores":  "6",
        "Constant Memory Loads":  "0",
        "Constant Memory Stores":  "0",
        "Local Memory Loads":  "0",
        "Local Memory Stores":  "0",
        "Private Memory Loads":  "20",
        "Private Memory Stores":  "20",
        "Total Loops":  "2",
        "Parallel Loops":  "1",
        "If Statements":  "2",
        "Integer Comparison":  "2",
        "Float Comparison":  "0",
        "Switch Statements":  "0",
        "Switch Cases":  "0",
        "Vector Operations":  "0",
```

(continues on next page)

(continued from previous page)

```
        "Integer & Float Operations":  "57",
        "Boolean Operations":  "9",
        "Cast Operations":  "2",
        "Float Math Functions":  "1",
        "Integer Math Functions":  "0"
    }
}
```

Use the option `-Dtornado.feature.extraction=True -Dtornado.features.dump.dir=FILENAME`. `FILENAME` can contain the filename and the full path (e.g. features.json).

TornadoVM allows redirecting profiling and feature extraction logs to a specific port. This feature can be enabled with the option `-Dtornado.dump.to.ip=IP:PORT`.
The following example redirects the profiler output to the localhost (127.0.0.1) and to a specified open port (2000):

```
$ tornado --jvm="-Dtornado.profiler=True  -Dtornado.dump.to.ip=127.0.0.1:2000" -m␣
↪tornado.examples/uk.ac.manchester.tornado.examples.VectorAddInt --params "100000"
```

To test that the socket streams the logs correctly, open a local server in a different terminal with the following command:

```
$ ncat -k -l 2000
```

1.8 Benchmarking TornadoVM

1.8.1 Benchmarks

Currently the benchmark runner script can execute the following benchmarks:

```
*saxpy
*addImage
*stencil
*convolvearray
*convolveimage
*blackscholes
*montecarlo
*blurFilter
*euler
*renderTrack
*nbody
*sgemm
*dgemm
*mandelbrot
*dft
```

For each benchmark, a Java version exists in order to obtain timing measurements. All performance and time measurements are obtained through a number of iterations (e.g. 130). Also, each benchmark can be tested for various array sizes ranging from `256` to `16777216`.

1.8.2 How to run

Go to the directory <tornadovm path>/bin/sdk/bin. Then, the run options can be found with the following command:

```
usage: tornado-benchmarks.py [-h] [--validate] [--default] [--medium]
                             [--iterations ITERATIONS] [--full]
                             [--skipSequential] [--skipParallel]
                             [--skipDevices SKIP_DEVICES] [--verbose]
                             [--printBenchmarks]

Tool to execute benchmarks in TornadoVM. With no options, it runs all
benchmarks with the default size

optional arguments:
  -h, --help            show this help message and exit
  --validate            Enable result validation
  --default             Run default benchmark configuration
  --medium              Run benchmarks with medium sizes
  --iterations ITERATIONS
                        Set the number of iterations
  --full                Run for all sizes in all devices. Including big data
                        sizes
  --skipSequential      Skip java version
  --skipParallel        Skip parallel version
  --skipDevices SKIP_DEVICES
                        Skip devices. Provide a list of devices (e.g., 0,1)
  --verbose, -V         Enable verbose
  --printBenchmarks     Print the list of available benchmarks
  --jmh                 Run with JMH
```

Example

Example of running all benchmark for all devices available in your system with the default data size.

```
$ tornado-benchmarks.py
Running TornadoVM Benchmarks
[INFO] This process takes between 30-60 minutes
[INFO] TornadoVM options: -Xms24G -Xmx24G -server
bm=saxpy-101-16777216, id=java-reference      , average=7.604811e+06, median=7.
↪521843e+06, firstIteration=1.179550e+07, best=7.355636e+06
bm=saxpy-101-16777216, device=0:0  , average=1.852340e+07, median=1.708197e+07,␣
↪firstIteration=2.788138e+07, best=1.612269e+07, speedupAvg=0.4106, speedupMedian=0.
↪4403, speedupFirstIteration=0.4231, CV=10.5305%, deviceName=NVIDIA CUDA -- GeForce GTX␣
↪1050
bm=saxpy-101-16777216, device=0:1  , average=4.503467e+07, median=4.482944e+07,␣
↪firstIteration=6.696712e+07, best=4.236860e+07, speedupAvg=0.1689, speedupMedian=0.
↪1678, speedupFirstIteration=0.1761, CV=4.7203%, deviceName=Intel(R) OpenCL -- Intel(R)␣
↪Core(TM) i7-7700HQ CPU @ 2.80GHz
bm=saxpy-101-16777216, device=0:2  , average=2.212386e+07, median=2.129296e+07,␣
↪firstIteration=3.493844e+07, best=1.975243e+07, speedupAvg=0.3437, speedupMedian=0.
↪3533, speedupFirstIteration=0.3376, CV=7.5316%, deviceName=AMD Accelerated Parallel␣
```

(continues on next page)

(continued from previous page)

```
↪Processing -- Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
bm=saxpy-101-16777216, device=0:3   , average=1.835022e+07, median=1.830117e+07,␣
↪firstIteration=2.965289e+07, best=1.760201e+07, speedupAvg=0.4144, speedupMedian=0.
↪4110, speedupFirstIteration=0.3978, CV=3.2015%, deviceName=Intel(R) OpenCL HD Graphics␣
↪-- Intel(R) Gen9 HD Graphics NEO
bm=add-image-101-2048-2048, id=java-reference       , average=6.076920e+07, median=5.
↪912435e+07, firstIteration=9.159228e+07, best=5.539140e+07
bm=add-image-101-2048-2048, device=0:0   , average=2.587469e+07, median=2.560709e+07,␣
↪firstIteration=6.173938e+07, best=2.399116e+07, speedupAvg=2.3486, speedupMedian=2.
↪3089, speedupFirstIteration=1.4835, CV=5.1914%, deviceName=NVIDIA CUDA -- GeForce GTX␣
↪1050
bm=add-image-101-2048-2048, device=0:1   , average=3.250553e+07, median=3.089569e+07,␣
↪firstIteration=8.700214e+07, best=2.691534e+07, speedupAvg=1.8695, speedupMedian=1.
↪9137, speedupFirstIteration=1.0528, CV=11.3154%, deviceName=Intel(R) OpenCL --␣
↪Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
bm=add-image-101-2048-2048, device=0:2   , average=3.061671e+07, median=3.037699e+07,␣
↪firstIteration=7.024932e+07, best=2.742994e+07, speedupAvg=1.9848, speedupMedian=1.
↪9464, speedupFirstIteration=1.3038, CV=4.3990%, deviceName=AMD Accelerated Parallel␣
↪Processing -- Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
bm=add-image-101-2048-2048, device=0:3   , average=2.564357e+07, median=2.512443e+07,␣
↪firstIteration=6.052658e+07, best=2.316377e+07, speedupAvg=2.3698, speedupMedian=2.
↪3533, speedupFirstIteration=1.5133, CV=4.9465%, deviceName=Intel(R) OpenCL HD Graphics␣
↪-- Intel(R) Gen9 HD Graphics NEO
bm=stencil-101-1048576, id=java-reference       , average=1.841053e+05, median=1.
↪885090e+05, firstIteration=4.734246e+06, best=1.636910e+05
bm=stencil-101-1048576, device=0:0   , average=1.862818e+05, median=1.863900e+05,␣
↪firstIteration=8.547734e+06, best=1.672090e+05, speedupAvg=0.9883, speedupMedian=1.
↪0114, speedupFirstIteration=0.5539, CV=13.9480%, deviceName=NVIDIA CUDA -- GeForce GTX␣
↪1050
bm=stencil-101-1048576, device=0:1   , average=1.323170e+05, median=1.272060e+05,␣
↪firstIteration=7.506147e+06, best=1.057020e+05, speedupAvg=1.3914, speedupMedian=1.
↪4819, speedupFirstIteration=0.6307, CV=12.2388%, deviceName=Intel(R) OpenCL --␣
↪Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
bm=stencil-101-1048576, device=0:2   , average=1.238349e+05, median=1.095310e+05,␣
↪firstIteration=4.092201e+06, best=8.586900e+04, speedupAvg=1.4867, speedupMedian=1.
↪7211, speedupFirstIteration=1.1569, CV=47.6368%, deviceName=AMD Accelerated Parallel␣
↪Processing -- Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
bm=stencil-101-1048576, device=0:3   , average=2.464191e+05, median=2.296330e+05,␣
↪firstIteration=4.807327e+06, best=2.218090e+05, speedupAvg=0.7471, speedupMedian=0.
↪8209, speedupFirstIteration=0.9848, CV=12.3793%, deviceName=Intel(R) OpenCL HD␣
↪Graphics -- Intel(R) Gen9 HD Graphics NEO
bm=convolve-array-100-2048-2048-5, id=java-reference       , average=2.612301e+08,␣
↪median=2.609304e+08, firstIteration=4.006838e+08, best=2.544892e+08
bm=convolve-array-100-2048-2048-5, device=0:0   , average=8.143104e+06, median=8.
↪214443e+06, firstIteration=1.811648e+07, best=7.609697e+06, speedupAvg=32.0799,␣
↪speedupMedian=31.7648, speedupFirstIteration=22.1171, CV=4.6348%, deviceName=NVIDIA␣
↪CUDA -- GeForce GTX 1050
bm=convolve-array-100-2048-2048-5, device=0:1   , average=9.842007e+07, median=9.
↪631152e+07, firstIteration=1.018732e+08, best=9.032237e+07, speedupAvg=2.6542,␣
↪speedupMedian=2.7092, speedupFirstIteration=3.9332, CV=9.3753%, deviceName=Intel(R)␣
↪OpenCL -- Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
...
```

1.8.3 Using JMH

The `tornado-benchmarks.py` script is configured to use JMH.

```
$ tornado-benchmarks.py --jmh
```

The script runs all benchmarks using JMH. This process takes ~3.5h.

Additionally, each benchmark has a JMH configuration. Users can execute any benchmark from the list as follows:

```
$ tornado -m tornado.benchmarks/uk.ac.manchester.tornado.benchmarks.<benchmark>.JMH
↪<BENCHMARK>
```

This process takes ~10mins per benchmark.

For example:

```
$ tornado -m tornado.benchmarks/uk.ac.manchester.tornado.benchmarks.dft.JMHDFT
# JMH version: 1.23
...
Benchmark           Mode  Cnt   Score   Error  Units
JMHDFT.dftJava      avgt    5  19.736 ± 1.589   s/op
JMHDFT.dftTornado   avgt    5   0.155 ± 0.008   s/op
```

1.9 FPGA Programming in TornadoVM

TornadoVM supports execution and prototyping with OpenCL compatible Intel and Xilinx FPGAs. For debugging you can use common IDEs from Java ecosystem.

IMPORTANT NOTE: The minimum input size to run on the FPGA is 64 elements (which corresponds internally with the local work size in OpenCL).

This document (*Cloud Deployments*) shows a full guideline for running TornadoVM on Amazon AWS F1 with Xilinx FPGAs.

1.9.1 Pre-requisites

We have currently tested with an Intel Nallatech-A385 FPGA (Intel Arria 10 GT1150) and a Xilinx KCU1500 FPGA card. We have also tested it on the AWS EC2 F1 instance with `xilinx_aws-vu9p-f1-04261818_dynamic_5_0` device.

- HLS Versions: Intel Quartus 17.1.0 Build 240, Xilinx SDAccel 2018.2, Xilinx SDAccel 2018.3, Xilinx Vitis 2020.2
- TornadoVM Version: >= 0.9
- AWS AMI Version: 1.6.0

If the OpenCL ICD loaders are installed correctly, the output of the `clinfo` should be the following:

```
$ clinfo
  Number of platforms                               1
  Platform Name                                   Intel(R) FPGA SDK for OpenCL(TM)
  Platform Vendor                                 Intel(R) Corporation
  Platform Version                                OpenCL 1.0 Intel(R) FPGA SDK for␣
```

(continues on next page)

(continued from previous page)

```
↪OpenCL(TM), Version 17.1
  Platform Profile                                EMBEDDED_PROFILE
  Platform Extensions                             cl_khr_byte_addressable_store cles_
↪khr_int64 cl_intelfpga_live_object_tracking cl_intelfpga_compiler_mode cl_khr_icd cl_
↪khr_3d_image_writes
  Platform Extensions function suffix             IntelFPGA

  Platform Name                                   Intel(R) FPGA SDK for OpenCL(TM)
  Number of devices                                 1
  Device Name                                     p385a_sch_ax115 : nalla_pcie␣
↪(aclnalla_pcie0)
  Device Vendor                                   Nallatech ltd
  Device Vendor ID                                0x1172
  Device Version                                  OpenCL 1.0 Intel(R) FPGA SDK for␣
↪OpenCL(TM), Version 17.1
  Driver Version                                  17.1
  Device OpenCL C Version                         OpenCL C 1.0
  Device Type                                     Accelerator
```

1.9.2 Step 1: Update/Create the FPGA's configuration file

Update the "$TORNADO_SDK/etc/vendor-fpga.conf" file with the necessary information (i.e. fpga platform name (DEVICE_NAME), HLS compiler (COMPILER), HLS compiler flags (FLAGS), HLS directory (DIRECTORY_BITSTREAM). TornadoVM will automatically load the user-defined configurations according to the vendor of the underlying FPGA device. You can also run TornadoVM with your configuration file, by using the `-Dtornado.fpga.conf.file=FILE` flag.

Example of a configuration file for Intel FPGAs (Emulation mode) with the Intel oneAPI Base Tool Kit:

Edit/create the configuration file:

```
$ vim $TORNADO_SDK/etc/intel-oneapi-fpga.conf
```

```
# Configure the fields for FPGA compilation & execution
# [device]
DEVICE_NAME = fpga_fast_emu
# [compiler]
COMPILER = aocl-ioc64
# [options]
DIRECTORY_BITSTREAM = fpga-source-comp/ # Specify the directory
```

Example of a configuration file for an Intel Nallatech-A385 FPGA (Intel Arria 10 GT1150):

Edit/create the configuration file fo the FPGA:

```
$ vim $TORNADO_SDK/etc/intel-fpga.conf
```

```
# Configure the fields for FPGA compilation & execution
# [device]
DEVICE_NAME = p385a_sch_ax115
# [compiler]
COMPILER = aoc
# [options]
FLAGS = -v -report # Configure the compilation flags
DIRECTORY_BITSTREAM = fpga-source-comp/ # Specify the directory
```

Example of a configuration file for a Xilinx KCU1500 FPGA:

```
$ vim $TORNADO_SDK/etc/xilinx-fpga.conf
```

```
# Configure the fields for FPGA compilation & execution
# [device]
DEVICE_NAME = xilinx_kcu1500_dynamic_5_0
# [compiler]
COMPILER = xocc
# [options]
FLAGS = -O3 -j12 # Configure the compilation flags
DIRECTORY_BITSTREAM = fpga-source-comp/ # Specify the directory
```

In order to use the Xilinx Toolchain, it is required to initialize the env variables of the SDAccel toolchain as follows:

```
source /opt/Xilinx/SDx/2018.2/settings64.sh
```

Example of a configuration file for a Xilinx Alveo U50 FPGA:

```
$ vim etc/xilinx-fpga.conf
```

```
# Configure the fields for FPGA compilation & execution
# [device]
DEVICE_NAME = xilinx_u50_gen3x16_xdma_201920_3
# [compiler]
COMPILER = v++
# [options]
FLAGS = -O3 -j12 # Configure the compilation flags
DIRECTORY_BITSTREAM = fpga-source-comp/ # Specify the directory
```

In order to use the Xilinx Toolchain, it is required to initialize the env variables of the Vitis toolchain as follows:

```
source /opt/Xilinx/Vitis/2020.2/settings64.sh
source /opt/xilinx/xrt/setup.sh
```

Example of a configuration file for an AWS xilinx_aws-vu9p-f1-04261818_dynamic_5_0 FPGA:

```
$ vim $TORNADO_SDK/etc/xilinx-fpga.conf
```

```
# Configure the fields for FPGA compilation & execution
# [device]
DEVICE_NAME = /home/centos/src/project_data/aws-fpga/SDAccel/aws_platform/xilinx_aws-
↪vu9p-f1-04261818_dynamic_5_0/xilinx_aws-vu9p-f1-04261818_dynamic_5_0.xpfm
# [compiler]
COMPILER = xocc
# [options]
FLAGS = -O3 -j12 # Configure the compilation flags
DIRECTORY_BITSTREAM = fpga-source-comp/ # Specify the directory
```

1.9.3 Step 2: Select one of the three FPGA Execution Modes

1. Full JIT Mode

This mode allows the compilation and execution of a given task for the FPGA. As it provides full end-to-end execution, the compilation is expected to take up to 2 hours due HLS bistream generation process.

The log dumps from the HLS compilation are written in the `output.log` file, and potential emerging errors in the `error.log` file. The compilation dumps along with the generated FPGA bitstream and the generated OpenCL code can be found in the `fpga-source-comp/` directory which is defined in the FPGA configuration file (Step 1).

Example:

```
tornado --jvm="-Ds0.t0.device=0:1" -m tornado.examples/uk.ac.manchester.tornado.examples.
↪dynamic.DFTMT --params="1024 normal 1"
```

Note: The Full JIT mode on the Alveo U50 presents some constraints regarding the maximum allocated space on the device memory. Although the Xilinx driver reports 1GB as the maximum allocation space, the XRT layer throws an error (`[XRT] ERROR: std::bad_alloc`) when the heap size is larger than 64MB. This issue is reported to Xilinx, and it is anticipated to be fixed soon. For applications that do not require more than 64MB of heap size, the following flag can be used `-Dtornado.device.memory=64MB`.

```
tornado --jvm="-Ds0.t0.device=0:1 -Dtornado.device.memory=64MB" -m tornado.examples/uk.
↪ac.manchester.tornado.examples.dynamic.DFTMT --params="1024 normal 1"
```

2. Ahead of Time Execution Mode

Ahead of time execution mode allows the user to use a pre-generated bitstream of the Tornado tasks and then load it in a separated execution. The path of the FPGA bitstream file should be given via the `-Dtornado.precompiled.binary` flag, and the file should be named as `lookupBufferAddress`.

Example:

```
tornado --jvm="-Ds0.t0.device=0:1 -Ds0.t0.global.workgroup.size=1024 -Ds0.t0.local.
↪workgroup.size=64 \
    -Dtornado.precompiled.binary=/path/to/lookupBufferAddress,s0.t0.device=0:1 "
    -m tornado.examples/uk.ac.manchester.tornado.examples.dynamic.DFTMT \
    --params="1024 normal 10"
```

Note: The Ahead of Time mode on the Alveo U50 presents some constraints regarding the maximum allocated space on the device memory. Although the Xilinx driver reports 1GB as the maximum allocation space, the XRT layer throws an error (`[XRT] ERROR: std::bad_alloc`) when the heap size is larger than 64MB. This issue is reported to Xilinx, and it is anticipated to be fixed soon. For applications that do not require more than 64MB of heap size, the following flag can be used `-Dtornado.device.memory=64MB`.

```
tornado --jvm="-Ds0.t0.device=0:1 -Dtornado.device.memory=64MB "\
    -m tornado.examples/uk.ac.manchester.tornado.examples.dynamic.DFTMT \
    --params="1024 normal 1"
```

3. Emulation Mode

Emulation mode can be used for fast-prototyping and ensuring program functional correctness before going through the full JIT process (HLS).

Before executing the TornadoVM program, the following steps needs to be executed based on the FPGA vendors' toolchain:

A) Emulation of an Intel platform:

You can run in Emulation mode either by using a Docker container or locally. In the following examples, we assume that the FPGA device uses the identifier `1:0`.

- Dockerized execution:

If you use the TornadoVM Docker image, you can run the following example.

Example:

```
./run_intel_openjdk.sh tornado \
    --jvm="-Ds0.t0.device=1:0 "
    -m tornado.examples/uk.ac.manchester.tornado.examples.dynamic.DFTMT --params="1024␣
↪default 10"
```

- Local execution:

If you use the `aocl-ioc64` emulator compiler/linker provided by the Intel oneAPI Base Tool Kit, you can run:

```
tornado \
    --jvm="-Ds0.t0.device=1:0 "
    -m tornado.examples/uk.ac.manchester.tornado.examples.dynamic.DFTMT --params="1024␣
↪default 10"
```

Alternatively, if you use the `aoc` FPGA SDK compiler that requires you to have the Intel(R) Quartus(R) Prime software already installed, you can:

Set the `CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA` env variable to `1`, so as to enable the execution on the emulated device.

```
$ export CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=1
```

Example:

```
env CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=1 tornado \
    --jvm="-Ds0.t0.device=0:1" \
    -m tornado.examples/uk.ac.manchester.tornado.examples.dynamic.DFTMT \
    --params="1024 normal 10"
```

B) Emulation of a Xilinx platform (using Vitis):

- Configure the device characteristics (e.g. which platform, number of devices) with the Xilinx Emulation Configuration Utility (emconfigutil) . Then you can use the TornadoVM Makefile and pass the configuration parameters as variables (e.g. `make xilinx_emulation FPGA_PLATFORM=<platform_name> NUM_OF_FPGA_DEVICES=<number_of_devices>`). *Be aware that the platform name must be the same with the device name in Step 1.* The default options configure one `xilinx_u50_gen3x16_xdma_201920_3` device. For example:

```
make xilinx_emulation FPGA_PLATFORM=xilinx_u50_gen3x16_xdma_201920_3 NUM_OF_FPGA_
↪DEVICES=1
```

- Set the `XCL_EMULATION_MODE` env variable to `sw_emu`, so as to enable the execution on the emulated device.

```
$ export XCL_EMULATION_MODE=sw_emu
```

Example:

```
tornado \
    --jvm="-Ds0.t0.device=0:1" \
    -m tornado.examples/uk.ac.manchester.tornado.examples.dynamic.DFTMT \
    --params="1024 normal 10"
```

Note: The emulation mode through SDAccel results in wrong results. However, when we run in the Full JIT or the Ahead of Time modes the kernels return correct results.

1.10 Docker Containers

We have tested our docker images for CentOS >= 7.4 and Ubuntu >= 16.04. We currently have docker images for NVIDIA GPUs, and Intel Integrated GPUs, Intel CPUs and Intel FPGAs using OpenJDK 11, 17 and GraalVM 22.2.0:

- TornadoVM docker images for **NVIDIA GPUs**
- TornadoVM docker images for **Intel Integrated Graphics, Intel FPGAs, and Intel CPUs**

1.10.1 Nvidia GPUs

Prerequisites

The `tornadovm-nvidia-openjdk` docker image needs the docker `nvidia` daemon. More info here: https://github.com/NVIDIA/nvidia-docker.

How to run?

1) Pull the image

For the `tornadovm-nvidia-openjdk` image:

```
$ docker pull beehivelab/tornadovm-nvidia-openjdk:latest
```

This image uses the latest TornadoVM for NVIDIA GPUs and OpenJDK 17.

2) Run an experiment

We provide a runner script that compiles and run your Java programs with Tornado. Here's an example:

```
$ git clone https://github.com/beehive-lab/docker-tornado
$ cd docker-tornado

## Run Matrix Multiplication - provided in the docker-tornado repository
$ ./run_nvidia_openjdk.sh tornado -cp example/target/example-1.0-SNAPSHOT.jar example.
↪MatrixMultiplication

Computing MxM of 2048x2048
    CPU Execution: 0.36 GFlops, Total time = 48254 ms
    GPU Execution: 277.09 GFlops, Total Time = 62 ms
    Speedup: 778x
```

Using TornadoVM with GraalVM for NVIDIA GPUs

With JDK 17:

```
$ docker pull beehivelab/tornadovm-nvidia-graalvm:latest
```

Some options

```
# To see the generated OpenCL kernel
$ ./run_nvidia.sh tornado --printKernel example/MatrixMultiplication

# To check some runtime info about the kernel execution and device
$ ./run_nvidia.sh tornado --debug example/MatrixMultiplication
```

The `tornado` command is just an alias to the `java` command with all the parameters for TornadoVM execution. So you can pass any Java (OpenJDK or Hotspot) parameter.

```
$ ./run_nvidia.sh tornado --jvm="-Xmx16g -Xms16g" example/MatrixMultiplication
```

1.10.2 Intel Integrated Graphics

Prerequisites

The `beehivelab/tornadovm-intel-openjdk` docker image Intel OpenCL driver for the integrated GPU installed. More info here: https://github.com/intel/compute-runtime.

How to run?

1) Pull the image

For the `beehivelab/tornadovm-intel-openjdk` image:

```
$ docker pull beehivelab/tornadovm-intel-openjdk:latest
```

This image uses the latest TornadoVM for Intel integrated graphics and OpenJDK 17.

2) Run an experiment

We provide a runner script that compiles and run your Java programs with TornadoVM. Here's an example:

```
$ git clone https://github.com/beehive-lab/docker-tornado
$ cd docker-tornado

## Run Matrix Multiplication - provided in the docker-tornado repository
$ ./run_intel_openjdk.sh tornado -cp example/target/example-1.0-SNAPSHOT.jar example.
↪MatrixMultiplication --parms="256"

Computing MxM of 256x256
    CPU Execution: 1.53 GFlops, Total time = 22 ms
    GPU Execution: 8.39 GFlops, Total Time = 4 ms
    Speedup: 5x
```

Using TornadoVM with GraalVM for Intel Integrated Graphics

With JDK 17:

```
$ docker pull beehivelab/tornadovm-intel-graalvm:latest
```

1.11 Cloud Deployments

TornadoVM can be executed on the cloud. This document explains how to use TornadoVM for running on Amazons AWS instances that contain GPUs or FPGAs.

1.11.1 1. Running on AWS for CPUs and GPUs

The installation and execution instructions for running on AWS CPUs and GPUs is identical to those for running locally. See the general installation steps here: *Installation*.

1.11.2 2. Running on AWS EC2 F1 Xilinx FPGAs

The following toolkit configuration comes with the AWS EC2 F1 instance:

- FPGA DEV AMI: 1.12.2
- Xilinx Vitis Tool: 2021.2

Pre-requisites:

- You need to install `python3` (tested with Python 3.9.6) and `OpenSSL 1.1.1`.
- You need to have an S3 storage bucket and permissions to access it (s3_bucket, s3_dcp_key and s3_logs_key) for Step 3.
- You need to clone the aws-fpga repository (this is the tested commit point), as follows:

```
$ cd /home/centos
$ git clone https://github.com/aws/aws-fpga.git $AWS_FPGA_REPO_DIR
```

1. Install TornadoVM as a CentOS user. The Xilinx FPGA is not exposed to simple users.

```
$ git clone https://github.com/beehive-lab/TornadoVM.git
$ cd TornadoVM
$ ./bin/tornadovm-installer --jdk jdk21 --backend opencl
$ source setvars.env
```

2. Follow these steps to get access to the Xilinx FPGA.

a. Enter a bash shell as root.

```
$ sudo -E /bin/bash
```

Note: If you face a failure regarding the generation of IP, try the patchhere**.**

b. Load the environment variables for Xilinx HLS and runtime.

```
$ source $AWS_FPGA_REPO_DIR/vitis_setup.sh
$ systemctl is-active --quiet mpd || sudo systemctl start mpd
```

c. Load the environment variables of TornadoVM for root.

```
$ cd /home/centos/TornadoVM
$ source setvars.env

$ tornado --devices
```

3. Update the the FPGA Conguration file

Update the `$TORNADO_SDK/etc/xilinx-fpga.conf` file or create your own (e.g. `$TORNADO_SDK/etc/aws-fpga.conf`), and append the necessary information (i.e. FPGA plarform name (DEVICE_NAME), HLS compiler flags (FLAGS), HLS directory (DIRECTORY_BITSTREAM), and AWS S3 configuration (s3_bucket, s3_dcp_key and s3_loogs_key)).

```
$ vim $TORNADO_SDK/etc/aws-fpga.conf
```

Example of configuration file:

```
[device]
DEVICE_NAME = /home/centos/src/project_data/aws-fpga/Vitis/aws_platform/xilinx_aws-vu9p-
↪f1_shell-v04261818_201920_3/xilinx_aws-vu9p-f1_shell-v04261818_201920_3.xpfm
[options]
COMPILER=v++
FLAGS = -O3 -j12 # Configure the compilation flags. You can also pass the HLS␣
↪configuration file (e.g. --config conf.cfg).
DIRECTORY_BITSTREAM = fpga-source-comp/
# If the FPGA is in AWS EC2 F1 Instance
AWS_ENV = yes
[AWS S3 configuration]
AWS_S3_BUCKET = tornadovm-fpga-bucket
AWS_S3_DCP_KEY = outputfolder
AWS_S3_LOGS_KEY = logfolder
```

You can run TornadoVM with your configuration file, by using the `-Dtornado.fpga.conf.file=FILE` flag. If this flag is not used, the default configuration file is the `$TORNADO_SDK/etc/xilinx-fpga.conf`.

4. Run a program that offloads a task on the FPGA.

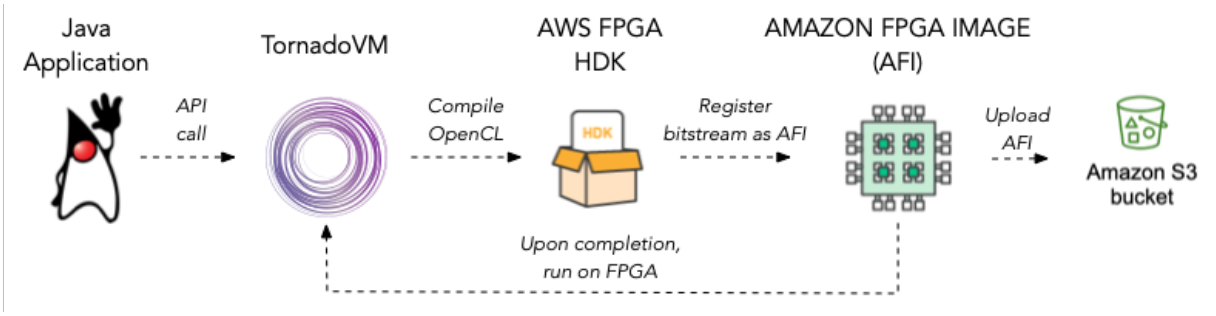


Fig. 1: image

The following example uses a custom configuration file (`aws-fpga.conf`) to execute the DFT on the AWS F1 FPGA:

```
$ tornado --jvm "-Ds0.t0.device=0:0 -Dtornado.fpga.conf.file=/home/centos/TornadoVM/bin/
↪sdk/etc/aws-fpga.conf -Xmx20g -Xms20g" --printKernel --threadInfo -m tornado.examples/
↪uk.ac.manchester.tornado.examples.dynamic.DFTMT --params="256 default 1" >> output.log
$ Ctrl-Z (^Z)
$ bg
$ disown
```

This command will trigger TornadoVM to automatically compile Java to OpenCL and use the AWS FPGA Hardware Development Kit (HDK) to generate a bitstream. You can also redirect the output from Standard OUT to a file (`output.log`) as the compilation may take a few hours and the connection may be terminated with a broken pipe (e.g. packet_write_wait: Connection to 174.129.48.160 port 22: Broken pipe).

Read the `output.log` file in order to monitor the outcome of the TornadoVM execution. To monitor the outcome of the HLS compilation, read the `outputFPGA.log` file, which is automatically generated in the `DIRECTORY_BITSTREAM` (e.g. `fpga-source-comp`). After the bitstream generation, TornadoVM will automatically invoke the creation of an Amazon FPGA Image (AFI) and upload a file related to the kernel to the Amazon S3 bucket (configured in the Step 3). The execution of the program will end up with an error as the bitstream is forwarded to be used, while the AFI image is not ready yet. E.g.:

```
[TornadoVM-OCL-JNI] ERROR : clCreateProgramWithBinary -> Returned: -44
```

5. You can monitor the status of your Amazon FPGA Image.

Instructions are given in `outputFPGA.log`. Ensure that you use the correct `FPGAImageId` (e.g. `afi-0c1bb6821ccc766fe`) .

```
$ cat fpga-source-comp/outputFPGA.log
$ aws ec2 describe-fpga-images --fpga-image-ids afi-0c1bb6821ccc766fe
```

This command will return the following message:


```
{
  "FpgaImages": [
    {
      "UpdateTime": "2021-05-27T23:55:15.000Z",
      "Name": "lookupBufferAddress",
      "Tags": [],
      "PciId": {
        "SubsystemVendorId": "0xfedd",
        "VendorId": "0x1d0f",
        "DeviceId": "0xf010",
        "SubsystemId": "0x1d51"
      },
      "FpgaImageGlobalId": "agfi-045c5d8825f920edc",
      "Public": false,
      "State": {
        "Code": "pending"
      },
      "ShellVersion": "0x04261818",
      "OwnerId": "813381863415",
      "FpgaImageId": "afi-0c1bb6821ccc766fe",
      "CreateTime": "2021-05-27T23:15:21.000Z",
      "Description": "lookupBufferAddress"
    }
  ]
}
```


When the state changes from `pending` to `available`, the `awsxlcbin` binary code can be executed via TornadoVM to the AWS FPGA.

6. Now that the AFI is available, you can execute the program and run the OpenCL kernel on the AWS FPGA.

If you have logged out, ensure that you run (Steps 2 and 4).

```
$ tornado --jvm="-Ds0.t0.device=0:0 -Dtornado.fpga.conf.file=/home/centos/TornadoVM/etc/
↪aws-fpga.conf -Xmx20g -Xms20g" --debug --printKernel -m tornado.examples/uk.ac.
↪manchester.tornado.examples.dynamic.DFTMT --params="256 default 1" >> output.log
```

The result is the following:

```
tornado --jvm="-Ds0.t0.device=0:0 -Dtornado.fpga.conf.file=/home/centos/TornadoVM-
↪Internal-feat-removeBufferCache/etc/aws-fpga.conf --threadInfo -Xmx20g -Xms20g" --
↪printKernel -m tornado.examples/uk.ac.manchester.tornado.examples.dynamic.DFTMT --
↪parms "256 default 1"
Initialization time:  705795966 ns
```

```
__attribute__((reqd_work_group_size(64, 1, 1)))
__kernel void computeDft(__global long *_kernel_context, __constant uchar *_constant_
↪region, __local uchar *_local_region, __global int *_atomics, __global uchar *inreal, _
↪_global uchar *inimag, __global uchar *outreal, __global uchar *outimag, __global␣
↪uchar *inputSize)
{
  int i_8, i_29, i_35, i_5, i_4, i_36;
  float f_6, f_7, f_24, f_25, f_26, f_27, f_28, f_16, f_17, f_18, f_19, f_20, f_21, f_22,
↪ f_23, f_13, f_15;
  ulong ul_12, ul_3, ul_2, ul_34, ul_14, ul_1, ul_33, ul_0;
  long l_9, l_10, l_11, l_30, l_31, l_32;

  // BLOCK 0
  ul_0  =  (ulong) inreal;
  ul_1  =  (ulong) inimag;
  ul_2  =  (ulong) outreal;
  ul_3  =  (ulong) outimag;
  i_4  =  get_global_id(0);
  // BLOCK 1 MERGES [0 5 ]
  i_5  =  i_4;
  // BLOCK 2
  // BLOCK 3 MERGES [2 4 ]
  f_6  =  0.0F;
  f_7  =  0.0F;
  i_8  =  0;
  __attribute__((xcl_pipeline_loop(1)))
  for(;i_8 < 256;)
  {
    // BLOCK 4
    l_9  =  (long) i_8;
    l_10  =  l_9 << 2;
    l_11  =  l_10 + 24L;
    ul_12  =  ul_0 + l_11;
    f_13  =  *((__global float *) ul_12);
    ul_14  =  ul_1 + l_11;
    f_15  =  *((__global float *) ul_14);
```

(continues on next page)

(continued from previous page)

```
    f_16  =  (float) i_8;
    f_17  =  f_16 * 6.2831855F;
    f_18  =  (float) i_5;
    f_19  =  f_17 * f_18;
    f_20  =  f_19 / 256.0F;
    f_21  =  native_sin(f_20);
    f_22  =  native_cos(f_20);
    f_23  =  f_22 * f_15;
    f_24  =  fma(f_21, f_13, f_23);
    f_25  =  f_7 - f_24;
    f_26  =  f_21 * f_15;
    f_27  =  fma(f_22, f_13, f_26);
    f_28  =  f_6 + f_27;
    i_29  =  i_8 + 1;
    f_6  =  f_28;
    f_7  =  f_25;
    i_8  =  i_29;
  }  // B4
  // BLOCK 5
  l_30  =  (long) i_5;
  l_31  =  l_30 << 2;
  l_32  =  l_31 + 24L;
  ul_33  =  ul_2 + l_32;
  *((__global float *) ul_33)  =  f_6;
  ul_34  =  ul_3 + l_32;
  *((__global float *) ul_34)  =  f_7;
  i_35  =  get_global_size(0);
  i_36  =  i_35 + i_5;
  i_5  =  i_36;
  // BLOCK 6
  return;
}  //  kernel

Task info: s0.t0
    Backend           : OPENCL
    Device            : xilinx_aws-vu9p-f1_shell-v04261818_201920_2 CL_DEVICE_TYPE_
↪ACCELERATOR (available)
    Dims              : 1
    Global work offset: [0]
    Global work size  : [256]
    Local  work size  : [64, 1, 1]
    Number of workgroups  : [4]

Total time:  4532676526 ns

Is valid?: true

Validation: SUCCESS
```

1.12 SPIR-V Devices

SPIR-V makes use of the Intel Level Zero API.

Disclaimer: The SPIR-V backend with the Intel Level-Zero dispatcher is a new project within TornadoVM. Currently, we offer a preview and an initial implementation.

1.12.1 Install Intel oneAPI Level Zero Compute Runtime

In order to use Intel Level Zero from oneAPI, you need to install the Intel driver for the Intel HD Graphics.

All drivers are available here: https://github.com/intel/compute-runtime/releases.

1.12.2 Install TornadoVM for SPIR-V

Install TornadoVM following the instructions in *Installation*.

To build the SPIR-V Backend, enable the backend as follows:

```
$ cd <tornadovm-directory>
$ ./scripts/tornadoVMInstaller.sh --jdk17 --spirv
$ . source.sh
```

1.12.3 Running examples with the SPIR-V backend

Running DFT from the unit-test suite

```
$ tornado-test -V --fast --threadInfo --debug uk.ac.manchester.tornado.unittests.compute.
↪ComputeTests#testDFT

SPIRV-File : /tmp/tornadoVM-spirv/8442884346950-s0.t0computeDFT.spv
Set entry point: computeDFT
Task info: s0.t0
    Backend           : SPIRV
    Device            : SPIRV LevelZero - Intel(R) UHD Graphics [0x9bc4] GPU
    Dims              : 1
    Global work offset: [0]
    Global work size  : [4096]
    Local  work size  : [256, 1, 1]
    Number of workgroups  : [16]

Test: class uk.ac.manchester.tornado.unittests.compute.ComputeTests#testDFT
    Running test: testDFT                    .............  [PASS]
```

In this execution, the SPIR-V Binary is stored in `/tmp/tornadoVM-spirv/8442884346950-s0.t0computeDFT.spv`. We can disassemble the binary with `spirv-dis` from Khronos

Note: Usually, `spirv-dis` can be installed from the common OS repositories (e.g., Fedora, Ubuntu repositories):

```
## Fedora OS
sudo dnf install spirv-tools

## Ubuntu OS:
sudo apt-get install spirv-tools
```

Disassemble the SPIR-V binary:

```
$ spirv-dis /tmp/tornadoVM-spirv/8442884346950-s0.t0computeDFT.spv
; SPIR-V
; Version: 1.2
; Generator: Khronos; 32
; Bound: 227
; Schema: 0
                OpCapability Addresses
                OpCapability Linkage
                OpCapability Kernel
                OpCapability Int64
                OpCapability Int8
                OpCapability Float64
          %1 = OpExtInstImport "OpenCL.std"
                OpMemoryModel Physical64 OpenCL
                OpEntryPoint Kernel %56 "computeDFT" %spirv_BuiltInGlobalInvocationId
↪%spirv_BuiltInGlobalSize
                OpExecutionMode %56 ContractionOff
                OpSource OpenCL_C 300000
                OpName %spirv_BuiltInGlobalInvocationId "spirv_BuiltInGlobalInvocationId"
                OpName %spirv_BuiltInGlobalSize "spirv_BuiltInGlobalSize"
                OpName %spirv_l_16F0 "spirv_l_16F0"
                OpName %spirv_l_12F0 "spirv_l_12F0"
                OpName %spirv_l_44F0 "spirv_l_44F0"
                OpName %spirv_l_13F0 "spirv_l_13F0"
                OpName %spirv_l_45F0 "spirv_l_45F0"
                OpName %spirv_l_14F0 "spirv_l_14F0"
                OpName %spirv_l_46F0 "spirv_l_46F0"
                OpName %spirv_l_42F0 "spirv_l_42F0"
                OpName %spirv_l_11F0 "spirv_l_11F0"
                OpName %spirv_l_43F0 "spirv_l_43F0"
                OpName %spirv_l_0F0 "spirv_l_0F0"
                OpName %spirv_l_1F0 "spirv_l_1F0"
                OpName %spirv_l_2F0 "spirv_l_2F0"
                OpName %spirv_l_3F0 "spirv_l_3F0"
                OpName %spirv_i_5F0 "spirv_i_5F0"
                OpName %spirv_i_4F0 "spirv_i_4F0"
                OpName %spirv_i_48F0 "spirv_i_48F0"
                OpName %spirv_i_47F0 "spirv_i_47F0"
                OpName %spirv_i_9F0 "spirv_i_9F0"
                OpName %spirv_i_41F0 "spirv_i_41F0"
                OpName %spirv_f_15F0 "spirv_f_15F0"
                OpName %spirv_f_17F0 "spirv_f_17F0"
                OpName %spirv_f_34F0 "spirv_f_34F0"
```

(continues on next page)

(continued from previous page)

```
               OpName %spirv_f_7F0 "spirv_f_7F0"
               OpName %spirv_f_23F0 "spirv_f_23F0"
               OpName %spirv_f_8F0 "spirv_f_8F0"
               OpName %spirv_f_40F0 "spirv_f_40F0"
               OpName %spirv_f_26F0 "spirv_f_26F0"
               OpName %spirv_d_20F0 "spirv_d_20F0"
               OpName %spirv_d_19F0 "spirv_d_19F0"
               OpName %spirv_d_22F0 "spirv_d_22F0"
               OpName %spirv_d_21F0 "spirv_d_21F0"
               OpName %spirv_d_24F0 "spirv_d_24F0"
               OpName %spirv_d_25F0 "spirv_d_25F0"
               OpName %spirv_d_28F0 "spirv_d_28F0"
               OpName %spirv_d_27F0 "spirv_d_27F0"
               OpName %spirv_d_30F0 "spirv_d_30F0"
               OpName %spirv_d_29F0 "spirv_d_29F0"
               OpName %spirv_d_32F0 "spirv_d_32F0"
               OpName %spirv_d_31F0 "spirv_d_31F0"
               OpName %spirv_d_33F0 "spirv_d_33F0"
               OpName %spirv_d_36F0 "spirv_d_36F0"
               OpName %spirv_d_35F0 "spirv_d_35F0"
               OpName %spirv_d_38F0 "spirv_d_38F0"
               OpName %spirv_d_37F0 "spirv_d_37F0"
               OpName %spirv_d_39F0 "spirv_d_39F0"
               OpName %spirv_d_18F0 "spirv_d_18F0"
               OpName %spirv_z_10F0 "spirv_z_10F0"
               OpName %spirv_z_6F0 "spirv_z_6F0"
               OpName %heapBaseAddr "heapBaseAddr"
               OpName %frameBaseAddr "frameBaseAddr"
               OpName %frame "frame"
               OpName %B0F0 "B0F0"
               OpName %B1F0 "B1F0"
               OpName %B2F0 "B2F0"
               OpName %B6F0 "B6F0"
               OpName %B3F0 "B3F0"
               OpName %B4F0 "B4F0"
               OpName %B5F0 "B5F0"
               OpName %returnF0 "returnF0"
               OpDecorate %spirv_BuiltInGlobalInvocationId BuiltIn GlobalInvocationId
               OpDecorate %spirv_BuiltInGlobalInvocationId Constant
               OpDecorate %spirv_BuiltInGlobalInvocationId LinkageAttributes "spirv_
↪BuiltInGlobalInvocationId" Import
               OpDecorate %spirv_BuiltInGlobalSize BuiltIn GlobalSize
               OpDecorate %spirv_BuiltInGlobalSize Constant
               OpDecorate %spirv_BuiltInGlobalSize LinkageAttributes "spirv_
↪BuiltInGlobalSize" Import
               OpDecorate %heapBaseAddr Alignment 8
               OpDecorate %frameBaseAddr Alignment 8
               OpDecorate %frame Alignment 8
               OpDecorate %spirv_l_16F0 Alignment 8
               OpDecorate %spirv_l_12F0 Alignment 8
               OpDecorate %spirv_l_44F0 Alignment 8
               OpDecorate %spirv_l_13F0 Alignment 8
```

(continues on next page)

(continued from previous page)

```
               OpDecorate %spirv_l_45F0 Alignment 8
               OpDecorate %spirv_l_14F0 Alignment 8
               OpDecorate %spirv_l_46F0 Alignment 8
               OpDecorate %spirv_l_42F0 Alignment 8
               OpDecorate %spirv_l_11F0 Alignment 8
               OpDecorate %spirv_l_43F0 Alignment 8
               OpDecorate %spirv_l_0F0 Alignment 8
               OpDecorate %spirv_l_1F0 Alignment 8
               OpDecorate %spirv_l_2F0 Alignment 8
               OpDecorate %spirv_l_3F0 Alignment 8
               OpDecorate %spirv_i_5F0 Alignment 4
               OpDecorate %spirv_i_4F0 Alignment 4
               OpDecorate %spirv_i_48F0 Alignment 4
               OpDecorate %spirv_i_47F0 Alignment 4
               OpDecorate %spirv_i_9F0 Alignment 4
               OpDecorate %spirv_i_41F0 Alignment 4
               OpDecorate %spirv_f_15F0 Alignment 4
               OpDecorate %spirv_f_17F0 Alignment 4
               OpDecorate %spirv_f_34F0 Alignment 4
               OpDecorate %spirv_f_7F0 Alignment 4
               OpDecorate %spirv_f_23F0 Alignment 4
               OpDecorate %spirv_f_8F0 Alignment 4
               OpDecorate %spirv_f_40F0 Alignment 4
               OpDecorate %spirv_f_26F0 Alignment 4
               OpDecorate %spirv_d_20F0 Alignment 8
               OpDecorate %spirv_d_19F0 Alignment 8
               OpDecorate %spirv_d_22F0 Alignment 8
               OpDecorate %spirv_d_21F0 Alignment 8
               OpDecorate %spirv_d_24F0 Alignment 8
               OpDecorate %spirv_d_25F0 Alignment 8
               OpDecorate %spirv_d_28F0 Alignment 8
               OpDecorate %spirv_d_27F0 Alignment 8
               OpDecorate %spirv_d_30F0 Alignment 8
               OpDecorate %spirv_d_29F0 Alignment 8
               OpDecorate %spirv_d_32F0 Alignment 8
               OpDecorate %spirv_d_31F0 Alignment 8
               OpDecorate %spirv_d_33F0 Alignment 8
               OpDecorate %spirv_d_36F0 Alignment 8
               OpDecorate %spirv_d_35F0 Alignment 8
               OpDecorate %spirv_d_38F0 Alignment 8
               OpDecorate %spirv_d_37F0 Alignment 8
               OpDecorate %spirv_d_39F0 Alignment 8
               OpDecorate %spirv_d_18F0 Alignment 8
               OpDecorate %spirv_z_10F0 Alignment 1
               OpDecorate %spirv_z_6F0 Alignment 1
     %uchar = OpTypeInt 8 0
     %ulong = OpTypeInt 64 0
      %uint = OpTypeInt 32 0
     %float = OpTypeFloat 32
    %double = OpTypeFloat 64
      %bool = OpTypeBool
    %uint_3 = OpConstant %uint 3
```

(continues on next page)

(continued from previous page)

```
   %ulong_24 = OpConstant %ulong 24
     %uint_2 = OpConstant %uint 2
%double_4096 = OpConstant %double 4096
  %uint_4096 = OpConstant %uint 4096
     %uint_1 = OpConstant %uint 1
%double_6_2831853071795862 = OpConstant %double 6.2831853071795862
    %float_0 = OpConstant %float 0
     %uint_0 = OpConstant %uint 0
       %void = OpTypeVoid
%_ptr_CrossWorkgroup_uchar = OpTypePointer CrossWorkgroup %uchar
         %74 = OpTypeFunction %void %_ptr_CrossWorkgroup_uchar %ulong
%_ptr_Function__ptr_CrossWorkgroup_uchar = OpTypePointer Function %_ptr_CrossWorkgroup_
↪uchar
%_ptr_CrossWorkgroup_ulong = OpTypePointer CrossWorkgroup %ulong
%_ptr_Function_ulong = OpTypePointer Function %ulong
%_ptr_Function__ptr_CrossWorkgroup_ulong = OpTypePointer Function %_ptr_CrossWorkgroup_
↪ulong
    %v3ulong = OpTypeVector %ulong 3
%_ptr_Input_v3ulong = OpTypePointer Input %v3ulong
%spirv_BuiltInGlobalSize = OpVariable %_ptr_Input_v3ulong Input
%spirv_BuiltInGlobalInvocationId = OpVariable %_ptr_Input_v3ulong Input
%_ptr_Function_uint = OpTypePointer Function %uint
%_ptr_Function_float = OpTypePointer Function %float
%_ptr_Function_double = OpTypePointer Function %double
%_ptr_Function_bool = OpTypePointer Function %bool
     %uint_4 = OpConstant %uint 4
     %uint_5 = OpConstant %uint 5
     %uint_6 = OpConstant %uint 6
    %ulong_2 = OpConstant %ulong 2
%_ptr_CrossWorkgroup_float = OpTypePointer CrossWorkgroup %float
         %56 = OpFunction %void DontInline %74
         %81 = OpFunctionParameter %_ptr_CrossWorkgroup_uchar
         %82 = OpFunctionParameter %ulong
       %B0F0 = OpLabel
%heapBaseAddr = OpVariable %_ptr_Function__ptr_CrossWorkgroup_uchar Function
%frameBaseAddr = OpVariable %_ptr_Function_ulong Function
%spirv_l_16F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_12F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_44F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_13F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_45F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_14F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_46F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_42F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_11F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_43F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_0F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_1F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_2F0 = OpVariable %_ptr_Function_ulong Function
%spirv_l_3F0 = OpVariable %_ptr_Function_ulong Function
%spirv_i_5F0 = OpVariable %_ptr_Function_uint Function
%spirv_i_4F0 = OpVariable %_ptr_Function_uint Function
```

(continues on next page)

(continued from previous page)

```
%spirv_i_48F0 = OpVariable %_ptr_Function_uint Function
%spirv_i_47F0 = OpVariable %_ptr_Function_uint Function
%spirv_i_9F0 = OpVariable %_ptr_Function_uint Function
%spirv_i_41F0 = OpVariable %_ptr_Function_uint Function
%spirv_f_15F0 = OpVariable %_ptr_Function_float Function
%spirv_f_17F0 = OpVariable %_ptr_Function_float Function
%spirv_f_34F0 = OpVariable %_ptr_Function_float Function
%spirv_f_7F0 = OpVariable %_ptr_Function_float Function
%spirv_f_23F0 = OpVariable %_ptr_Function_float Function
%spirv_f_8F0 = OpVariable %_ptr_Function_float Function
%spirv_f_40F0 = OpVariable %_ptr_Function_float Function
%spirv_f_26F0 = OpVariable %_ptr_Function_float Function
%spirv_d_20F0 = OpVariable %_ptr_Function_double Function
%spirv_d_19F0 = OpVariable %_ptr_Function_double Function
%spirv_d_22F0 = OpVariable %_ptr_Function_double Function
%spirv_d_21F0 = OpVariable %_ptr_Function_double Function
%spirv_d_24F0 = OpVariable %_ptr_Function_double Function
%spirv_d_25F0 = OpVariable %_ptr_Function_double Function
%spirv_d_28F0 = OpVariable %_ptr_Function_double Function
%spirv_d_27F0 = OpVariable %_ptr_Function_double Function
%spirv_d_30F0 = OpVariable %_ptr_Function_double Function
%spirv_d_29F0 = OpVariable %_ptr_Function_double Function
%spirv_d_32F0 = OpVariable %_ptr_Function_double Function
%spirv_d_31F0 = OpVariable %_ptr_Function_double Function
%spirv_d_33F0 = OpVariable %_ptr_Function_double Function
%spirv_d_36F0 = OpVariable %_ptr_Function_double Function
%spirv_d_35F0 = OpVariable %_ptr_Function_double Function
%spirv_d_38F0 = OpVariable %_ptr_Function_double Function
%spirv_d_37F0 = OpVariable %_ptr_Function_double Function
%spirv_d_39F0 = OpVariable %_ptr_Function_double Function
%spirv_d_18F0 = OpVariable %_ptr_Function_double Function
%spirv_z_10F0 = OpVariable %_ptr_Function_bool Function
%spirv_z_6F0 = OpVariable %_ptr_Function_bool Function
      %frame = OpVariable %_ptr_Function__ptr_CrossWorkgroup_ulong Function
               OpStore %heapBaseAddr %81 Aligned 8
               OpStore %frameBaseAddr %82 Aligned 8
         %88 = OpLoad %_ptr_CrossWorkgroup_uchar %heapBaseAddr Aligned 8
         %89 = OpLoad %ulong %frameBaseAddr Aligned 8
         %90 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_uchar %88 %89
         %91 = OpBitcast %_ptr_CrossWorkgroup_ulong %90
               OpStore %frame %91 Aligned 8
         %92 = OpLoad %_ptr_CrossWorkgroup_ulong %frame Aligned 8
         %93 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_ulong %92 %uint_3
         %94 = OpLoad %ulong %93 Aligned 8
               OpStore %spirv_l_0F0 %94 Aligned 8
         %95 = OpLoad %_ptr_CrossWorkgroup_ulong %frame Aligned 8
         %97 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_ulong %95 %uint_4
         %98 = OpLoad %ulong %97 Aligned 8
               OpStore %spirv_l_1F0 %98 Aligned 8
         %99 = OpLoad %_ptr_CrossWorkgroup_ulong %frame Aligned 8
        %101 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_ulong %99 %uint_5
        %102 = OpLoad %ulong %101 Aligned 8
```

(continues on next page)

(continued from previous page)

```
                OpStore %spirv_l_2F0 %102 Aligned 8
         %103 = OpLoad %_ptr_CrossWorkgroup_ulong %frame Aligned 8
         %105 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_ulong %103 %uint_6
         %106 = OpLoad %ulong %105 Aligned 8
                OpStore %spirv_l_3F0 %106 Aligned 8
         %107 = OpLoad %v3ulong %spirv_BuiltInGlobalInvocationId Aligned 32
         %108 = OpCompositeExtract %ulong %107 0
         %109 = OpUConvert %uint %108
                OpStore %spirv_i_4F0 %109 Aligned 4
         %110 = OpLoad %uint %spirv_i_4F0 Aligned 4
                OpStore %spirv_i_5F0 %110 Aligned 4
                OpBranch %B1F0
        %B1F0 = OpLabel
         %112 = OpLoad %uint %spirv_i_5F0 Aligned 4
         %113 = OpSLessThan %bool %112 %uint_4096
                OpBranchConditional %113 %B2F0 %B6F0
        %B2F0 = OpLabel
                OpStore %spirv_f_7F0 %float_0 Aligned 4
                OpStore %spirv_f_8F0 %float_0 Aligned 4
                OpStore %spirv_i_9F0 %uint_0 Aligned 4
                OpBranch %B3F0
        %B3F0 = OpLabel
         %117 = OpLoad %uint %spirv_i_9F0 Aligned 4
         %118 = OpSLessThan %bool %117 %uint_4096
                OpBranchConditional %118 %B4F0 %B5F0
        %B4F0 = OpLabel
         %121 = OpLoad %uint %spirv_i_9F0 Aligned 4
         %122 = OpSConvert %ulong %121
                OpStore %spirv_l_11F0 %122 Aligned 8
         %123 = OpLoad %ulong %spirv_l_11F0 Aligned 8
         %125 = OpShiftLeftLogical %ulong %123 %ulong_2
                OpStore %spirv_l_12F0 %125 Aligned 8
         %126 = OpLoad %ulong %spirv_l_12F0 Aligned 8
         %127 = OpIAdd %ulong %126 %ulong_24
                OpStore %spirv_l_13F0 %127 Aligned 8
         %128 = OpLoad %ulong %spirv_l_0F0 Aligned 8
         %129 = OpLoad %ulong %spirv_l_13F0 Aligned 8
         %130 = OpIAdd %ulong %128 %129
                OpStore %spirv_l_14F0 %130 Aligned 8
         %131 = OpLoad %ulong %spirv_l_14F0 Aligned 8
         %133 = OpConvertUToPtr %_ptr_CrossWorkgroup_float %131
         %134 = OpLoad %float %133 Aligned 4
                OpStore %spirv_f_15F0 %134 Aligned 4
         %135 = OpLoad %ulong %spirv_l_1F0 Aligned 8
         %136 = OpLoad %ulong %spirv_l_13F0 Aligned 8
         %137 = OpIAdd %ulong %135 %136
                OpStore %spirv_l_16F0 %137 Aligned 8
         %138 = OpLoad %ulong %spirv_l_16F0 Aligned 8
         %139 = OpConvertUToPtr %_ptr_CrossWorkgroup_float %138
         %140 = OpLoad %float %139 Aligned 4
                OpStore %spirv_f_17F0 %140 Aligned 4
         %141 = OpLoad %uint %spirv_i_9F0 Aligned 8
```

(continues on next page)

(continued from previous page)

```
        %142 = OpConvertSToF %double %141
               OpStore %spirv_d_18F0 %142 Aligned 8
        %143 = OpLoad %double %spirv_d_18F0 Aligned 8
        %144 = OpFMul %double %143 %double_6_2831853071795862
               OpStore %spirv_d_19F0 %144 Aligned 8
        %145 = OpLoad %uint %spirv_i_5F0 Aligned 8
        %146 = OpConvertSToF %double %145
               OpStore %spirv_d_20F0 %146 Aligned 8
        %147 = OpLoad %double %spirv_d_19F0 Aligned 8
        %148 = OpLoad %double %spirv_d_20F0 Aligned 8
        %149 = OpFMul %double %147 %148
               OpStore %spirv_d_21F0 %149 Aligned 8
        %150 = OpLoad %double %spirv_d_21F0 Aligned 8
        %151 = OpFDiv %double %150 %double_4096
               OpStore %spirv_d_22F0 %151 Aligned 8
        %152 = OpLoad %double %spirv_d_22F0 Aligned 4
        %153 = OpFConvert %float %152
               OpStore %spirv_f_23F0 %153 Aligned 4
        %154 = OpLoad %float %spirv_f_23F0 Aligned 8
        %155 = OpFConvert %double %154
               OpStore %spirv_d_24F0 %155 Aligned 8
        %156 = OpLoad %double %spirv_d_24F0 Aligned 8
        %157 = OpExtInst %double %1 sin %156
               OpStore %spirv_d_25F0 %157 Aligned 8
        %158 = OpLoad %float %spirv_f_15F0 Aligned 4
        %159 = OpFNegate %float %158
               OpStore %spirv_f_26F0 %159 Aligned 4
        %160 = OpLoad %float %spirv_f_26F0 Aligned 8
        %161 = OpFConvert %double %160
               OpStore %spirv_d_27F0 %161 Aligned 8
        %162 = OpLoad %double %spirv_d_24F0 Aligned 8
        %163 = OpExtInst %double %1 native_cos %162
               OpStore %spirv_d_28F0 %163 Aligned 8
        %164 = OpLoad %float %spirv_f_17F0 Aligned 8
        %165 = OpFConvert %double %164
               OpStore %spirv_d_29F0 %165 Aligned 8
        %166 = OpLoad %double %spirv_d_28F0 Aligned 8
        %167 = OpLoad %double %spirv_d_29F0 Aligned 8
        %168 = OpFMul %double %166 %167
               OpStore %spirv_d_30F0 %168 Aligned 8
        %169 = OpLoad %double %spirv_d_25F0 Aligned 8
        %170 = OpLoad %double %spirv_d_27F0 Aligned 8
        %171 = OpLoad %double %spirv_d_30F0 Aligned 8
        %172 = OpExtInst %double %1 fma %169 %170 %171
               OpStore %spirv_d_31F0 %172 Aligned 8
        %173 = OpLoad %float %spirv_f_8F0 Aligned 8
        %174 = OpFConvert %double %173
               OpStore %spirv_d_32F0 %174 Aligned 8
        %175 = OpLoad %double %spirv_d_31F0 Aligned 8
        %176 = OpLoad %double %spirv_d_32F0 Aligned 8
        %177 = OpFAdd %double %175 %176
               OpStore %spirv_d_33F0 %177 Aligned 8
```

(continues on next page)

(continued from previous page)

```
        %178 = OpLoad %double %spirv_d_33F0 Aligned 4
        %179 = OpFConvert %float %178
               OpStore %spirv_f_34F0 %179 Aligned 4
        %180 = OpLoad %float %spirv_f_15F0 Aligned 8
        %181 = OpFConvert %double %180
               OpStore %spirv_d_35F0 %181 Aligned 8
        %182 = OpLoad %double %spirv_d_25F0 Aligned 8
        %183 = OpLoad %double %spirv_d_29F0 Aligned 8
        %184 = OpFMul %double %182 %183
               OpStore %spirv_d_36F0 %184 Aligned 8
        %185 = OpLoad %double %spirv_d_28F0 Aligned 8
        %186 = OpLoad %double %spirv_d_35F0 Aligned 8
        %187 = OpLoad %double %spirv_d_36F0 Aligned 8
        %188 = OpExtInst %double %1 fma %185 %186 %187
               OpStore %spirv_d_37F0 %188 Aligned 8
        %189 = OpLoad %float %spirv_f_7F0 Aligned 8
        %190 = OpFConvert %double %189
               OpStore %spirv_d_38F0 %190 Aligned 8
        %191 = OpLoad %double %spirv_d_37F0 Aligned 8
        %192 = OpLoad %double %spirv_d_38F0 Aligned 8
        %193 = OpFAdd %double %191 %192
               OpStore %spirv_d_39F0 %193 Aligned 8
        %194 = OpLoad %double %spirv_d_39F0 Aligned 4
        %195 = OpFConvert %float %194
               OpStore %spirv_f_40F0 %195 Aligned 4
        %196 = OpLoad %uint %spirv_i_9F0 Aligned 4
        %197 = OpIAdd %uint %196 %uint_1
               OpStore %spirv_i_41F0 %197 Aligned 4
        %198 = OpLoad %float %spirv_f_40F0 Aligned 4
               OpStore %spirv_f_7F0 %198 Aligned 4
        %199 = OpLoad %float %spirv_f_34F0 Aligned 4
               OpStore %spirv_f_8F0 %199 Aligned 4
        %200 = OpLoad %uint %spirv_i_41F0 Aligned 4
               OpStore %spirv_i_9F0 %200 Aligned 4
               OpBranch %B3F0
       %B5F0 = OpLabel
        %201 = OpLoad %uint %spirv_i_5F0 Aligned 4
        %202 = OpSConvert %ulong %201
               OpStore %spirv_l_42F0 %202 Aligned 8
        %203 = OpLoad %ulong %spirv_l_42F0 Aligned 8
        %204 = OpShiftLeftLogical %ulong %203 %ulong_2
               OpStore %spirv_l_43F0 %204 Aligned 8
        %205 = OpLoad %ulong %spirv_l_43F0 Aligned 8
        %206 = OpIAdd %ulong %205 %ulong_24
               OpStore %spirv_l_44F0 %206 Aligned 8
        %207 = OpLoad %ulong %spirv_l_2F0 Aligned 8
        %208 = OpLoad %ulong %spirv_l_44F0 Aligned 8
        %209 = OpIAdd %ulong %207 %208
               OpStore %spirv_l_45F0 %209 Aligned 8
        %210 = OpLoad %ulong %spirv_l_45F0 Aligned 8
        %211 = OpConvertUToPtr %_ptr_CrossWorkgroup_float %210
        %212 = OpLoad %float %spirv_f_7F0 Aligned 4
```

(continues on next page)

(continued from previous page)

```
               OpStore %211 %212 Aligned 4
       %213 = OpLoad %ulong %spirv_l_3F0 Aligned 8
       %214 = OpLoad %ulong %spirv_l_44F0 Aligned 8
       %215 = OpIAdd %ulong %213 %214
               OpStore %spirv_l_46F0 %215 Aligned 8
       %216 = OpLoad %ulong %spirv_l_46F0 Aligned 8
       %217 = OpConvertUToPtr %_ptr_CrossWorkgroup_float %216
       %218 = OpLoad %float %spirv_f_8F0 Aligned 4
               OpStore %217 %218 Aligned 4
       %219 = OpLoad %v3ulong %spirv_BuiltInGlobalSize Aligned 32
       %220 = OpCompositeExtract %ulong %219 0
       %221 = OpUConvert %uint %220
               OpStore %spirv_i_47F0 %221 Aligned 4
       %222 = OpLoad %uint %spirv_i_47F0 Aligned 4
       %223 = OpLoad %uint %spirv_i_5F0 Aligned 4
       %224 = OpIAdd %uint %222 %223
               OpStore %spirv_i_48F0 %224 Aligned 4
       %225 = OpLoad %uint %spirv_i_48F0 Aligned 4
               OpStore %spirv_i_5F0 %225 Aligned 4
               OpBranch %B1F0
     %B6F0 = OpLabel
               OpBranch %returnF0
  %returnF0 = OpLabel
               OpReturn
               OpFunctionEnd
```

1.13 CUDA Devices

1.13.1 Prerequisites

In order to use the PTX backend of TornadoVM, you will need a CUDA compatible device (NVIDIA GPUs with CUDA support).

Driver Installation

Step 1:
You will need to setup the `CUDA Toolkit`. If you don't have it installed already, you can follow this guide.

Step 2:
Make sure you follow the environment setup to add the required environment variables.
Depending on the installation, you might also have to expand your `C_INCLUDE_PATH` and `LD_LIBRARY_PATH` variables to include the CUDA headers.

```
$ export C_INCLUDE_PATH=/usr/local/cuda/include:${C_INCLUDE_PATH}
$ export LD_LIBRARY_PATH=/usr/local/cuda/lib64:${LD_LIBRARY_PATH}
```

To ensure that the installation has been successful, you can run the following commands: `nvidia-smi` and `nvcc --version`.

The output of `nvidia-smi` should be similar to:

```
+-----------------------------------------------------------------------------+
 | NVIDIA-SMI 440.100      Driver Version: 440.100      CUDA Version: 10.2     |
 |-------------------------------+----------------------+----------------------+
 | GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
 | Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
 |===============================+======================+======================|
 |   0  GeForce GTX 1650    Off  | 00000000:01:00.0 Off |                  N/A |
 | N/A   51C    P8     1W /  N/A |     73MiB /  3914MiB |      0%      Default |
 +-------------------------------+----------------------+----------------------+

 +-----------------------------------------------------------------------------+
 | Processes:                                                       GPU Memory |
 |  GPU       PID   Type   Process name                             Usage      |
 |=============================================================================|
 |    0      1095      G   /usr/lib/xorg/Xorg                            36MiB |
 |    0      1707      G   /usr/lib/xorg/Xorg                            36MiB |
 +-----------------------------------------------------------------------------+
```

The output of `nvcc --version` should be similar to:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Wed_Oct_23_19:24:38_PDT_2019
Cuda compilation tools, release 10.2, V10.2.89
```

1.13.2 TornadoVM Installation

Step 3:
Install TornadoVM as described here: *Installation*.
Build TornadoVM with the `PTX` backend selected and run `tornado --devices`.

The output of the TornadoVM build containing both backends (PTX and OpenCL) should look like this:

```
Number of Tornado drivers: 2
Total number of devices  : 1
Tornado device=0:0
 CUDA-PTX -- GeForce GTX 1650
     Global Memory Size: 3.8 GB
     Local Memory Size: 48.0 KB
     Workgroup Dimensions: 3
     Max WorkGroup Configuration: [1024, 1024, 64]
     Device OpenCL C version: N/A

Total number of devices  : 2
Tornado device=1:0
 NVIDIA CUDA -- GeForce GTX 1650
     Global Memory Size: 3.8 GB
```

(continues on next page)

(continued from previous page)

```
    Local Memory Size: 48.0 KB
    Workgroup Dimensions: 3
    Max WorkGroup Configuration: [1024, 1024, 64]
    Device OpenCL C version: OpenCL C 1.2

Tornado device=1:1
 Intel(R) OpenCL HD Graphics -- Intel(R) Gen9 HD Graphics NEO
    Global Memory Size: 24.8 GB
    Local Memory Size: 64.0 KB
    Workgroup Dimensions: 3
    Max WorkGroup Configuration: [256, 256, 256]
    Device OpenCL C version: OpenCL C 2.0
```

Note that the first Tornado driver will always correspond to the CUDA device detected by the PTX backend.

1.13.3 Addressing Possible issues

In some cases, running `nvidia-smi` might show the error `NVIDIA-SMI has failed because it couldn't communicate with the NVIDIA driver`. This can happen because the driver module is not loaded due to a black-list file.

You can remove this by running: `sudo rm /etc/modprobe.d/blacklist-nvidia.conf`

On Ubuntu, the driver can also fail to load if it is not selected in `prime-select`. In order to select it, you can run `prime-select nvidia` or `prime-select on-demand`.

For older versions of the driver, you might have to point your `LIBRARY_PATH` variable to the `libcuda` library in order to build TornadoVM.
Example: `export LIBRARY_PATH=$LIBRARY_PATH:/usr/local/cuda/lib64/stubs`

After these changes, a reboot might be required for the driver module to be loaded.

1.13.4 Testing the CUDA Backend of TornadoVM

We have tested the PTX backend of TornadoVM on the following configurations:

GPU	Arch	PTX ISA Version	Target	Driver version	CUDA version	Status
RTX 3070	Ampere	8.6	sm_86	510.54	11.8	OK
RTX 2060	Turing	7.5	sm_75	510.54	11.6	OK
Quadro GP100	Pascal	6.0	sm_60	384.111	9.0	Fun ctional
GeForce GTX 1650	Turing	6.5	sm_75	440.100	10.2	OK
GeForce 930MX	Maxwell	6.4	sm_50	418.56	10.1	OK
GeForce 930MX	Maxwell	6.5	sm_50	450.36	11.0	OK

DISCLAIMER:

The PTX backend might fail with the `Quadro GP100`, driver `384.111`, with segmentation faults for some of the unit test due to driver issues.

1.14 Multi-Device Execution

TornadoVM supports multi-device execution for task-graphs that contain multiple tasks without data dependencies between them. This feature allows users to better utilize the available hardware and potentially improve the overall execution time of their applications.

TornadoVM executes on multiple devices in two modes:

1) **Sequential:**
   It will launch each independent task on a different accelerator *sequentially* (i.e., one after the other). This mode is mainly used for debugging.

2) **Concurrent:**
   It will launch each independent task on a different accelerator *concurrently*. In this mode, a separate Java thread per accelerator is spawned.

1.14.1 Prerequisites

Before using TornadoVM's multi-device execution, make sure that you have one of the supported backends (e.g., OpenCL, PTX, and SPIRV) with at least 2 available devices.

Ensuring that multiple devices are available

By running the following command you can obtain the list of the available devices that will be required later on in this tutorial, as well as their unique Tornado device ids.

```
$ tornado --devices

 Number of Tornado drivers: 1
 Driver: OpenCL
   Total number of OpenCL devices  : 3
   Tornado device=0:0  (DEFAULT)
     OPENCL --  [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070
         Global Memory Size: 7.8 GB
         Local Memory Size: 48.0 KB
         Workgroup Dimensions: 3
         Total Number of Block Threads: [1024]
         Max WorkGroup Configuration: [1024, 1024, 64]
         Device OpenCL C version: OpenCL C 1.2

   Tornado device=0:1
     OPENCL --  [Intel(R) OpenCL] -- 13th Gen Intel(R) Core(TM) i7-13700
         Global Memory Size: 62.5 GB
         Local Memory Size: 32.0 KB
         Workgroup Dimensions: 3
         Total Number of Block Threads: [8192]
         Max WorkGroup Configuration: [8192, 8192, 8192]
         Device OpenCL C version: OpenCL C 3.0

   Tornado device=0:2
     OPENCL --  [Intel(R) FPGA Emulation Platform for OpenCL(TM)] -- Intel(R) FPGA␣
↪Emulation Device
         Global Memory Size: 62.5 GB
         Local Memory Size: 256.0 KB
         Workgroup Dimensions: 3
         Total Number of Block Threads: [67108864]
         Max WorkGroup Configuration: [67108864, 67108864, 67108864]
         Device OpenCL C version: OpenCL C 1.2
```

1.14.2 Sequential Execution on Multiple Devices

In the following example we are going to use the blur filter application that operates on three different color channels (e.g., red, blue, and green). Each colour channel is processed by a separate TornadoVM task, resulting in a TaskGraph (named *blur*) composed of three tasks (named *red*, *green* and *blue*). Also, from the devices shown above we going to use devices 0:0 (NVIDIA GeForce RTX 3070) and 0:1 (13th Gen Intel(R) Core(TM) i7-13700).

```
$ tornado --threadInfo  \
        --jvm=" -Dblur.red.device=0:0 -Dblur.green.device=0:1 -Dblur.blue.device=0:0" \
        -m  tornado.examples/uk.ac.manchester.tornado.examples.compute.BlurFilter
```

TornadoVM requires the user to explicitly define on which device each task will run; otherwise, TornadoVM will use its default device for the selected backend. In the given example, we have specified the device assignments for each task of the TaskGraph blur.

- *-Dblur.red.device=0:0* This specifies that the red task of the BlurFilter will run on device 0:0 (NVIDIA GeForce RTX 3070).
- *-Dblur.green.device=0:1* This specifies that the green task of the BlurFilter will run on device 0:1 (Intel Core i7-13700).
- *-Dblur.blue.device=0:0* This specifies that the blue task of the BlurFilter will also run on device 0:0 (NVIDIA GeForce RTX 3070).

The expected output after execution is:

```
Task info: blur.red
    Backend           : OPENCL
    Device            : NVIDIA GeForce RTX 3070 CL_DEVICE_TYPE_GPU (available)
    Dims              : 2
    Global work offset: [0, 0]
    Global work size  : [4000, 6000]
    Local  work size  : [32, 30, 1]
    Number of workgroups  : [125, 200]

Task info: blur.blue
    Backend           : OPENCL
    Device            : NVIDIA GeForce RTX 3070 CL_DEVICE_TYPE_GPU (available)
    Dims              : 2
    Global work offset: [0, 0]
    Global work size  : [4000, 6000]
    Local  work size  : [32, 30, 1]
    Number of workgroups  : [125, 200]

Task info: blur.green
    Backend           : OPENCL
    Device            : 13th Gen Intel(R) Core(TM) i7-13700 CL_DEVICE_TYPE_CPU␣
↪(available)
    Dims              : 2
    Global work offset: [0, 0]
    Global work size  : [24, 1]
    Local  work size  : null
    Number of workgroups  : [0, 0]
```

1.14.3 Concurrent Execution on Multiple Devices

In the previous example, although the tasks did not share dependencies, they still ran sequentially. Thus, one device has been idle, while the tasks were executed one after the other. To improve performance and run tasks concurrently on multiple devices, use the `--enableConcurrentDevices` flag:

```
$ tornado --threadInfo \
    --enableConcurrentDevices \
    --jvm=" -Dblur.red.device=0:0 -Dblur.green.device=0:1 -Dblur.blue.device=0:0" \
    -m  tornado.examples/uk.ac.manchester.tornado.examples.compute.BlurFilter
```

By adding the –enableConcurrentDevices flag, one instance of the TornadoVM Interpreter per device will be spawned through a Java thread-pool, allowing all devices to run concurrently.

The expected output after execution is:

```
    Task info: blur.red
    Backend           : OPENCL
    Device            : NVIDIA GeForce RTX 3070 CL_DEVICE_TYPE_GPU (available)
    Dims              : 2
    Global work offset: [0, 0]
    Global work size  : [4000, 6000]
    Local  work size  : [32, 30, 1]
    Number of workgroups  : [125, 200]

Task info: blur.green
    Backend           : OPENCL
    Device            : 13th Gen Intel(R) Core(TM) i7-13700 CL_DEVICE_TYPE_CPU␣
↪(available)
    Dims              : 2
    Global work offset: [0, 0]
    Global work size  : [24, 1]
    Local  work size  : null
    Number of workgroups  : [0, 0]

Task info: blur.blue
    Backend           : OPENCL
    Device            : NVIDIA GeForce RTX 3070 CL_DEVICE_TYPE_GPU (available)
    Dims              : 2
    Global work offset: [0, 0]
    Global work size  : [4000, 6000]
    Local  work size  : [32, 30, 1]
    Number of workgroups  : [125, 200]
```

1.14.4 How to debug

Previously, we enabled debug information solely to display the thread and device configuration for each task. TornadoVM can dump additional information to help developers to trace where the code is executed.

To access this information, you need to include the `--printBytecodes` flag in the above example from Section (*Concurrent Execution on Multiple Devices*). By adding this flag, the following output will be displayed in conjunction with the thread information:

```
$ tornado --threadInfo \
    --printBytecodes \
    --enableConcurrentDevices \
    --jvm=" -Dblur.red.device=0:0 -Dblur.green.device=0:1 -Dblur.blue.device=0:0" \
    -m  tornado.examples/uk.ac.manchester.tornado.examples.compute.BlurFilter
```

The expected output after execution is:

```
Interpreter instance running bytecodes for:   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070␣
↪Running in thread:  pool-1-thread-1
bc:  ALLOC [I@2ffe106e on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070 , size=0
bc:  ALLOC [I@705e1b5b on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070 , size=0
bc:  ALLOC [F@63f945a3 on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070 , size=0
bc:  TRANSFER_HOST_TO_DEVICE_ONCE  [Object Hash Code=0x2ffe106e] [I@2ffe106e on  ␣
↪[NVIDIA CUDA] -- NVIDIA GeForce RTX 3070 , size=0, offset=0 [event list=-1]
bc:  TRANSFER_HOST_TO_DEVICE_ONCE  [Object Hash Code=0x63f945a3] [F@63f945a3 on  ␣
↪[NVIDIA CUDA] -- NVIDIA GeForce RTX 3070 , size=0, offset=0 [event list=-1]
bc:  LAUNCH  task blur.red - compute on  [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070,␣
↪size=0, offset=0 [event list=0]
bc:  ALLOC [I@738395e4 on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070 , size=0
bc:  ALLOC [I@1d78beeb on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070 , size=0
bc:  TRANSFER_HOST_TO_DEVICE_ONCE  [Object Hash Code=0x738395e4] [I@738395e4 on  ␣
↪[NVIDIA CUDA] -- NVIDIA GeForce RTX 3070 , size=0, offset=0 [event list=-1]
bc:  LAUNCH  task blur.blue - compute on  [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070,␣
↪size=0, offset=0 [event list=2]
bc:  TRANSFER_DEVICE_TO_HOST_ALWAYS [0x705e1b5b] [I@705e1b5b on   [NVIDIA CUDA] --␣
↪NVIDIA GeForce RTX 3070 , size=0, offset=0 [event list=3]
bc:  TRANSFER_DEVICE_TO_HOST_ALWAYS [0x1d78beeb] [I@1d78beeb on   [NVIDIA CUDA] --␣
↪NVIDIA GeForce RTX 3070 , size=0, offset=0 [event list=5]
bc:  DEALLOC [0x2ffe106e] [I@2ffe106e on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070
bc:  DEALLOC [0x705e1b5b] [I@705e1b5b on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070
bc:  DEALLOC [0x63f945a3] [F@63f945a3 on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070
bc:  DEALLOC [0x738395e4] [I@738395e4 on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070
bc:  DEALLOC [0x1d78beeb] [I@1d78beeb on   [NVIDIA CUDA] -- NVIDIA GeForce RTX 3070
bc:  BARRIER  event-list 17
bc:  END

Interpreter instance running bytecodes for:   [Intel(R) OpenCL] -- 13th Gen Intel(R)␣
↪Core(TM) i7-13700 Running in thread:  pool-1-thread-2
bc:  ALLOC [I@41ac3343 on   [Intel(R) OpenCL] -- 13th Gen Intel(R) Core(TM) i7-13700 ,␣
↪size=0
bc:  ALLOC [I@16c36388 on   [Intel(R) OpenCL] -- 13th Gen Intel(R) Core(TM) i7-13700 ,␣
↪size=0
bc:  ALLOC [F@63f945a3 on   [Intel(R) OpenCL] -- 13th Gen Intel(R) Core(TM) i7-13700 ,␣
↪size=0
```

(continues on next page)

(continued from previous page)

```
bc:  TRANSFER_HOST_TO_DEVICE_ONCE  [Object Hash Code=0x41ac3343] [I@41ac3343 on  ␣
↪[Intel(R) OpenCL] -- 13th Gen Intel(R) Core(TM) i7-13700 , size=0, offset=0 [event␣
↪list=-1]
bc:  TRANSFER_HOST_TO_DEVICE_ONCE  [Object Hash Code=0x63f945a3] [F@63f945a3 on  ␣
↪[Intel(R) OpenCL] -- 13th Gen Intel(R) Core(TM) i7-13700 , size=0, offset=0 [event␣
↪list=-1]
bc:  LAUNCH  task blur.green - compute on  [Intel(R) OpenCL] -- 13th Gen Intel(R)␣
↪Core(TM) i7-13700, size=0, offset=0 [event list=1]
bc:  TRANSFER_DEVICE_TO_HOST_ALWAYS [0x16c36388] [I@16c36388 on   [Intel(R) OpenCL] --␣
↪13th Gen Intel(R) Core(TM) i7-13700 , size=0, offset=0 [event list=4]
bc:  DEALLOC [0x41ac3343] [I@41ac3343 on   [Intel(R) OpenCL] -- 13th Gen Intel(R)␣
↪Core(TM) i7-13700
bc:  DEALLOC [0x16c36388] [I@16c36388 on   [Intel(R) OpenCL] -- 13th Gen Intel(R)␣
↪Core(TM) i7-13700
bc:  DEALLOC [0x63f945a3] [F@63f945a3 on   [Intel(R) OpenCL] -- 13th Gen Intel(R)␣
↪Core(TM) i7-13700
bc:  BARRIER  event-list 17
bc:  END
```

Let's take a closer look at the first line: Interpreter instance running bytecodes for: [NVIDIA CUDA] – NVIDIA GeForce RTX 3070 Running in thread: pool-1-thread-1.

This line reveals details about the TornadoVM interpreter's operation. We observe that we have two separate instances of the TornadoVM interpreter, each running independently within distinct Java threads. One instance operates within pool-1-thread-1, while the other operates in pool-1-thread-2. In the sequential execution scenario showcased earlier in this tutorial (*Sequential Execution on Multiple Devices*), we would expect all instances of the TornadoVM interpreter to run from the main Java thread.

This distinction is essential as it helps us understand how TornadoVM's bytecodes are executed in parallel, while also efficiently utilizing available hardware resources, such as the NVIDIA GeForce RTX 3070 GPU and the 13th Gen Intel(R) Core(TM) i7-13700 CPU (based on the earlier debug output).

By comprehending these details, developers gain valuable information on how TornadoVM efficiently harnesses multi-threading capabilities. The feature of running multiple tasks on multiple devices results in enhanced performance and overall system efficiency.

1.14.5 Not Supported

- Tasks that share data dependencies can run only on a single device.
- Batch processing can run only on a single device.
- Dynamic reconfiguration only explores single device execution.

1.15 TornadoVM Flags

There is a number of runtime flags and compiler flags to enable experimental features, as well as fine and coarse grain profiling in the context of TornadoVM.

Note: for the following examples `s0` represents an arbitrary task-graph, as well as `t0` represents a given task's name.

All flags needs Java prefix of `-D`. An example of tornado using a flag is the following:

```
$ tornado --jvm "-Dtornado.fullDebug=true" -m tornado.examples/uk.ac.manchester.examples.
compute.Montecarlo 1024
```

1.15.1 List of TornadoVM Flags:

- `-Dtornado.fullDebug=true`:
  Enables full debugging log to be output in the. command line.
- `--printKernel`:
  Print the generated OpenCL/PTX kernel in the command line.
- `--threadInfo`:
  Print the information about the number of parallel threads used.
- `--debug`:
  Print minor debug information such as compilation status, generated code and device information.
- `--fullDebug`:
  In addition to the information dumped by the basic debug, the full debug mode also dumps information about the TornadoVM bytecode, and internal runtime status. This option is mainly used for development of TornadoVM.
- `--devices`:
  Output a list of all available devices on the current system.
- `-Dtornado.ns.time=true`:
  Converts the time to units to nanoseconds instead of milliseconds.
- `-Dtornado.{ptx,opencl}.priority=X`: Allows to define a driver priority. The drivers are sorted in descending order based on their priority. By default, the `PTX driver` has priority `1` and the `OpenCL driver` has priority `0`.
- `-Ds0.t0.global.workgroup.size=XXX,XXX,XXX`:
  Allows to define global worksizes (problem sizes).
- ``-Ds0.t0.local.workgroup.size=XXX,XXX,`:
  Allows to define custom local workgroup configuration and overwrite the default values provided by the TornadoScheduler.
- `-Dtornado.profiling.enable=true`:
  Enable profilling for OpenCL/CUDA events such as kernel times and data tranfers.
- `-Dtornado.opencl.userelative=true`:
  Enable usage of relative addresses which is a prerequisite for using DMA tranfers on Altera/Intel FPGAs. Nonetheless, this flag can be used for any OpenCL device.
- `-Dtornado.precompiled.binary=PATH`: Provides the location of the bistream or pre-generated OpenCL (.cl) kernel.
- `-Dtornado.fpga.conf.file=FILE`: Provides the absolute path of the FPGA configuation file.

- `-Dtornado.fpgaDumpLog=true`: Dumps the log information from the HLS compilation to the command prompt.
- `-Dtornado.opencl.blocking=true`:
  Allows to force OpenCL API blocking calls.
- `--enableProfiler console`:
  It enables profiler information such as `COPY_IN`, `COPY_OUT`, compilation time, total time, etc. This flag is disabled by default. TornadoVM will print by STDOUT a JSON string containing all profiler metrics related to the execution of each task-schedule.
- `--enableProfiler silent`: It enables profiler information such as `COPY_IN`, `COPY_OUT`, compilation time, total time, etc. This flag is disabled by default. The profiler information is stored internally and it can be queried using the TornadoVM Profiler API.
- `--dumpProfiler FILENAME`:
  It enables profiler information such as `COPY_IN`, `COPY_OUT`, compilation time, total time, etc. This flag is disabled by default. TornadoVM will save the profiler information in the `FILENAME` after the execution of each task-schedule.
- `-Dtornado.opencl.compiler.options=LIST_OF_OPTIONS`:
  It allows to pass the compile options specified by the OpenCL `CLBuildProgram` specification to TornadoVM at runtime. By default it doesn't enable any.
- `-Dtornado.concurrent.devices=true`:
  Allows to run a TaskGraph in multiple devices concurrently. The user needs explicitly to define the device for each task, otherwise all tasks will run on the default device. For instance, `-Ds0.t0.device=0:0 -Ds0.t1.device=0:1`

Optimizations

- `-Dtornado.enable.fma=True`:
  It enables Fused-Multiply-Add optimizations. This option is enabled by default. However, for some platforms, such as the Xilinx FPGA using SDAccel 2018.2 and OpenCL 1.0, this option must be disabled as it causes runtime errors. See issue on Github.
- `-Dtornado.enable.mathOptimizations`: It enables math optimizations. For instance, `1/sqrt(x)` is transformed into `rsqrt` instruction for the corresponding backend (OpenCL, SPIRV and PTX). It is enabled by default.
- `-Dtornado.experimental.partial.unroll=True`: It enables the compiler to force partial unroll on counted loops with a factor of 2. The unroll factor can be configured with the `tornado.partial.unroll.factor=FACTOR` that the FACTOR value can take integer values up to 32.
- `-Dtornado.enable.nativeFunctions=False`: It enables the utilization of native mathematical functions, in case that the selected backend (OpenCL, PTX, SPIR-V) supports native functions. This option is disabled by default.

Level Zero

- `-Dtornado.spirv.levelzero.alignment=64`: Memory alignment (in bytes) for Level Zero buffers. It is set to 64 by default.
- `-Dtornado.spirv.levelzero.thread.dispatcher=True`: If it is enabled, it uses the Level Zero suggested thread block for the thread dispatcher. True by default.
- `-Dtornado.spirv.loadstore=False`: It optimizes Loads/Stores and simplifies the generated SPIR-V binary. This option is still experimental. It is set to `False` by default.
- `-Dtornado.spirv.levelzero.memoryAlloc.shared=False`: If it is enabled, then it uses shared memory buffers between the accelerator and the host. It is set to false by default.

1.16 IDE Integration

1.16.1 IntelliJ

Download and install the latest IntelliJ IDEA Community Edition: https://www.jetbrains.com/idea/download/

Change the IntelliJ maximum memory to 2 GB or more (instructions).

For Intellij to pickup the required Tornado dependencies from the poms, go to **View > Tool Windows > Maven** and select **jdk-8, ptx-backend, opencl-backend** under profiles.

Required Plugins:

Open IntelliJ and go to **Preferences > Plugins > Browse Repositories**. Install the following plugins:

1. Eclipse Code Formatter: Formats source code according to eclipse standards (.xml). After the installation of the plugin, go to: **File > Settings > Other Settings > Ecliple Code Formatter**, and select **"Use the Eclipse code formatter"**. Finally, load the TornadoVM code formatter (**./scripts/templates/eclipse-settings/Tornado.xml**) as the Eclipse Java Formatter config file, and click **Apply**.
2. Save Actions: Allows post-save actions (e.g. code formating on save).
    - To enable the auto-formatter with save-actions, go to **Settings -> Other Settings -> Save Actions**, and mark the following:
        - Activate save actions on save
        - Activate save actions in shortcut
        - Reformat file
3. Python Plugin (Optional): Allows Python scripting.
4. **CheckStyle-IDEA Plugin (Optional):**
   Checks project for compliance with custom checkstyle rules.

Run and Debug TornadoVM with Intellij

Normal maven lifecycle goals like *package* and *install* will not result a succefull build for TornadoVM.

Two different configurations are needed for **Build** and **Debug**.

Build/Run Configuration

1. Go to **File > Project Structure** and ensure that:

- **In the Project Tab:**
  - The Project SDK uses the same java version as the project (e.g. Java 17).
  - The Project language level is using the same java version (e.g. Java 17 with Lambdas, type annotations etc.).
- **In the Modules Tab:**
  - The module SDK of every module uses the same java version (e.g. 1.8.0_131).

1. In the right vertical bar in Intellij: **Maven Projects > tornadovm (root) > Lifecycle > package > right click > create tornadovm [package]**
2. Then: **Run > Edit Configurations > Maven > tornadovm Package**. You need to manually add and check the following information:

- In the **Parameters** tab :
  - In **Command line**, add the following:
    - `-Dcmake.root.dir=/home/michalis/opt/cmake-3.10.2-Linux-x86_64/ clean package`
      - In case that you need to reduce the amount of maven warnings add also on the above line the command **–quiet**, which constraints maven verbose to only errors.
  - In **Profiles (separated with space)**, add the following:
    - `jdk-8`
    - at least one backend depending on what you want to build `{opencl-backend, ptx-backend}`
- In the **Runner** tab: Ensure that the selected **JRE** corresponds to `Use Project JDK` (e.g.1.8.0_131).

Finally, one the top right corner drop-down menu select the adove custome `tornadovm [package]` configuration. To build either press the **play button** on the top right corner or **Shift+F10**.

Debug/Run Configuration

In order to Run and Debug Java code running through TornadoVM another custom configuration is needed. For this configuration the TornadoVM `JAVA_FLAGS` and `CLASSPATHS` are needed.

Firstly, you need to obtain the ``JAVA_FLAGS`` used by TornadoVM. Use the following commands to print the flags:

```
$ make BACKENDS={opencl,ptx,spirv}
$ tornado --printJavaFlags
```

Output should be something similar to this:

```
/PATH_TO_JDK/jdk1.8.0_131/bin/java
-server -XX:-UseJVMCIClassLoader -XX:-UseCompressedOops -Djava.ext.dirs=/home/michalis/
↪Tornado/tornado/bin/sdk/share/java/tornado -Djava.library.path=/home/michalis/Tornado/
↪tornado/bin/sdk/lib -Dtornado.load.api.implementation=uk.ac.manchester.tornado.runtime.
↪tasks.TornadoTaskGraph -Dtornado.load.runtime.implementation=uk.ac.manchester.tornado.
↪runtime.TornadoCoreRuntime -Dtornado.load.tornado.implementation=uk.ac.manchester.
↪tornado.runtime.common.Tornado -Dtornado.load.device.implementation.opencl=uk.ac.
↪manchester.tornado.drivers.opencl.runtime.OCLDeviceFactory -Dtornado.load.device.
↪implementation.ptx=uk.ac.manchester.tornado.drivers.ptx.runtime.PTXDeviceFactory
```

You need to copy from `-server` to end.

Now, introduce a new Run Configuration

Again, **Run > Edit Configurations > Application > Add new (e.g. plus sign)**

Then, add your own parameters similar to the following:

- **Main Class:** uk.ac.manchester.tornado.examples.compute.MatrixMultiplication1D
- **VM Options:** What you copied from `-server` and on
- **Working Directory:** `/home/tornadovm`
- **JRE:** Default (Should point to the 1.8.0_131)
- **Use classpath of module** Select from drop-down menu e.g `tornado-examples`

Finally, you can select the new custom configuration by selecting the configuration from the right top drop-down menu. Now, you can run it by pressing the **play button** on the top right corner or **Shift+F10**.

CheckStyle-IDEA Configuration

First, add the custom checkstyle file to enable its rules go to **IntelliJ > Settings > Tools > CheckStyle** then, under configuration file click plus then add the configuration file which is under *tornado-assembly/src/etc/checkstyle.xml*.

Then, on the side on enabled plugins click on checkstyle and then in *rules* topdown menu click the custom rules file.

1.17 Developer Guidelines

This guide shows the configuration for the code formatting and code check styles for TornadoVM and IntelliJ.

1.17.1 IntelliJ Configurations

1. Enable Eclipse Code Formatter for IntelliJ

- Install the Eclipse Code Formatter plugin from the JetBrains plugin repository:

  Eclipse Code Formatter Plugin
- After installation, navigate to **File > Settings > Adapter for Eclipse Code Formatter**.
- Select the option **"Use the Eclipse code formatter"**.
- Load the TornadoVM code formatter from this path `scripts/templates/eclipse-settings/Tornadovm_eclipse_formatter.xml` using the selector **Eclipse Formatter Config > Eclipse workspace/project folder**.

- Click **Apply** to save the settings.

2. Enable IntelliJ Code Formatter

- Go to Menu Settings → Editor → Code Style.
- Import the code style scheme by following these steps: - Click on the cog icon in the top right corner of the Code Style settings ("Schema" field). - Choose "Import Schema" - Import the file located at: `scripts/templates/intellij-settings/Tornadovm_intellij_formatter.xml` - Click **Apply**.

3. Checkstyle-IDEA Plugin

This plugin provides both real-time and on-demand scanning of Java files with Checkstyle from within IDEA.

- Install the Checkstyle-IDEA plugin by going to **File > Settings** (Windows/Linux) or **IntelliJ IDEA > Preferences...** (macOS).
- Select **Plugins**, press **Browse Repository**, and find the plugin CheckStyle-IDEA
- Restart the IDE to complete the installation.
- Click **File > Settings > Tools > Checkstyle**.
- Set the **Scan Scope** to "Only Java sources (including tests)" to run Checkstyle for test source codes as well.
- Click the plus sign under **Configuration File**.
- Enter a description (e.g., "TornadoVM Checkstyle").
- Select **Use a local Checkstyle file**.
- Use the Checkstyle configuration file found at `tornado-assembly/src/etc/checkstyle.xml`.
- Click **Next > Finish**.
- Mark the newly imported check configuration as **Active** and click **Apply**.

4. EditorConfig

We use JetBrains' EditorConfig. This allows us to import and export code style settings easily.

- Copy the EditorConfig file to the root of your project:

```
cd $TORNADO_ROOT
cp scripts/templates/intellij-settings/.editorconfig .
```

- In IntelliJ IDEA, navigate to Menu Settings → Editor → Code Style.
- At the bottom of the settings window, check the box "Use EditorConfig".

1. Save Actions

Install the **Save Actions** Plugin. This allows you to define post-save actions, including code formatting.

- To enable the auto-formatter with save-actions, follow these steps: - Go to **Settings > Other Settings > Save Actions**. - Mark the following options: Activate save actions on save, Activate save actions in shortcut and Reformat file.

1.17.2 Pre-commit hooks

Install pre-commit hooks

Pre-commit docs: <*https://pre-commit.com/*>_

```
pip install pre-commit
pre-commit install
```

Every time there is a commit in the TornadoVM repo, the pre-commit will pass some checks (including code check style and code formatter). If all checks are correct, then the commit will be done.

To guarantee the commit, pass the check style before:

```
make checkstyle     ### If there are errors regarding the code formatting, fix it at␣
↪this stage.
```

1.18 Frequently Asked Questions

1.18.1 1. What can TornadoVM do?

TornadoVM accelerates parts of your Java applications on heterogeneous hardware devices such as multicore CPUs, GPUs, and FPGAs.

TornadoVM is currently being used to accelerate machine learning and deep learning applications, computer vision, physics simulations, financial applications, computational photography, natural language processing and signal processing.

1.18.2 2. Can I use TornadoVM in a commercial application?

Absolutely yes! TornadoVM employs many licenses as shown here, but its API is under **Apache 2**, and hence it can be freely used in any application.

1.18.3 3. How can I use it?

In Linux and Mac OSx, TornadoVM can be installed by the installer. Alternatively, TornadoVM can be configured either manually (*Installation*) or by using docker images (*Docker Containers*).

List of compatible JDKs

TornadoVM can be currently executed with the following configurations:

- TornadoVM with GraalVM (JDK 21): see the installation guide: *Installation for GraalVM for JDK 21.0.1 on Linux and OSx*.
- TornadoVM with JDK21 (e.g. OpenJDK 21, Red Hat Mandrel 21, Amazon Corretto 21, Azul Zulu JDK 21): see the installation guide: *TornadoVM for JDK 21 on Linux and OSx*.

Windows

To run TornadoVM on **Windows 10/11 OS**, install TornadoVM with GraalVM. More information here: *TornadoVM for Windows 10/11 using GraalVM*.

ARM Mali GPUs and Linux

To run TornadoVM on ARM Mali, install TornadoVM with GraalVM and JDK 11. More information here: *TornadoVM on ARM Mali GPUs*.

Usage

- Examples of how to use TornadoVM: *Running Examples and Benchmarks*.
- Code examples

1.18.4 4. Which programming languages does TornadoVM support?

TornadoVM primarily supports Java. However, with the integration with GraalVM you can call your TornadoVM-compatible Java code through other programming languages supported by GraalVM's polyglot runtime (e.g., Python, R, Ruby, Javascript, Node.js, etc).

Here you can find examples of how to use TornadoVM with GraalVM Polyglot.

1.18.5 5. Is TornadoVM a Domain Specific Language (DSL)?

No, TornadoVM is not a DSL. It compiles a subset of Java code to OpenCL C, NVIDIA PTX, and SPIR-V binary.

The TornadoVM API only provides two Java annotations (`@Parallel` and `@Reduce`) plus an APIs to: a) Create and define task-graphs (groups of Java methods to be accelerated by TornadoVM), and the data needed to execute those task-graphs. b) Define execution plans.

1.18.6 6. Does it support the whole Java Language?

No, TornadoVM supports a subset of the Java programming language. A list of unsupported features along with the reasoning behind it can be found here: *Unsupported Java features*.

1.18.7 7. Can TornadoVM degrade the performance of my application?

No, TornadoVM can only increase the performance of your application because it can dynamically change the execution of a program at runtime onto another device. If a particular code segment cannot be accelerated, then execution falls back to the host JVM which will execute your code on the CPU as it would normally do.

Also with the **Dynamic Reconfiguration**, TornadoVM discovers the fastest possible device for a particular code segment completely transparently to the user.

1.18.8 8. Dynamic Reconfiguration? What is this?

It is a novel feature of TornadoVM, in which the user selects a metric on which the system decides how to map a specific computation on a particular device. Further details and instructions on how to enable this feature can be found here:

- Dynamic reconfiguration: https://dl.acm.org/doi/10.1145/3313808.3313819.

1.18.9 9. Does TornadoVM support only OpenCL devices?

No. Currently, TornadoVM supports three compiler backends and therefore, it is able to generate OpenCL, PTX, and SPIR-V code depending on the hardware configuration.

1.18.10 10. Why is it called a VM?

The VM name is used because TornadoVM implements its own set of bytecodes for handling heterogeneous execution. These bytecodes are used for handling JIT compilation, device exploration, data management and live task-migration for heterogeneous devices (multi-core CPUs, GPUs, and FPGAs). We sometimes refer to a VM inside a VM (nested VM). The main VM is the Java Virtual Machine, and TornadoVM sits on top of that.

You can find more information here: https://dl.acm.org/doi/10.1145/3313808.3313819.

1.18.11 11. How does it interact with OpenJDK?

TornadoVM makes use of the Java Virtual Machine Common Interface (JVMCI) that is included from Java 9 to compile Java bytecode to OpenCL C/PTX/SPIR-V at runtime. As a JVMCI implementation, TornadoVM uses Graal (it extends the Graal IR and includes new backends for OpenCL C, PTX and SPIR-V code generation).

1.18.12 12. How do I know which parts of my application are suitable for acceleration?

Workloads with for-loops that do not have dependencies between iterations are very good candidates to offload on accelerators. Examples of this pattern are NBody computation, Black-scholes, DFT, KMeans, etc.

Besides, matrix-type applications are good candidates, such as matrix-multiplication widely used in machine and deep learning.

1.18.13 13. How can I contribute to TornadoVM?

TornadoVM is an open-source project, and, as such, we welcome contributions from all levels.

- **Solve** issues reported on the GitHub page.
- **Work on New Proposals**: We welcome new proposals and ideas. To work on a new proposal, use the discussion page on GitHub. Alternatively, you can open a shared document (e.g., a shared Google doc) where we can discuss and analyse your proposal.

Here you can find more information about how to contribute, code conventions, and tasks.

1.18.14 14. Does TornadoVM support calls to standard Java libraries?

Partially yes. TornadoVM currently supports calls to the Math library. However, invocations that imply I/O are not supported. Note that this restriction also applies to low-level parallel programming models such as OpenCL, SYCL, oneAPI and CUDA.

1.18.15 15. Do I need a GPU to run TornadoVM?

No. TornadoVM can also run on multi-core CPUs and/or FPGAs. What TornadoVM needs is a compatible driver/runtime installed in the machine. For example, to enable TornadoVM getting access to an Intel CPU, developers can use the Intel OpenCL runtime (e.g., from the Intel oneAPI base Toolkit).

To enable TornadoVM accessing FPGAs, developers can use the Intel and AMD OpenCL implementations for the Intel and Xilinx FPGAs, respectively.

1.19 Unsupported Java features

TornadoVM currently supports a subset of Java. Most limitations are due to the underlying programming model (OpenCL) that TornadoVM uses. This document summarizes each of the limitations and their explanations.

1.19.1 1. No Recursion

TornadoVM does not support recursion. This is also a current limitation of OpenCL, CUDA and SPIR-V.

1.19.2 2. No Object Support (*)

TornadoVM currently supports off-heap memory allocation via custom data types (e.g., `IntArray`, `FloatArray`, `DoubleArray`, `LongArray`, `CharArray`, `ShortArray`, and `ByteArray`) and arrays of primitive Java types (e.g., `int`, `float`, `double`, `short`, `char`, and `long`). Furthermore, TornadoVM offers support for some object types, such as `VectorFloat`, `VectorFloat4` and all variations with types as well as matrices types.

The full list of data structures supported is in this link.

Those are the objects that TornadoVM knows their memory layout. TornadoVM generates specialized OpenCL code for those data structures. For example, `VectorFloat4` generates accesses using OpenCL vector types (`float4`). This might speed-up user code if the target device contains explicit vector units, such as AVX on Intel CPUs or vector register on AMD GPUs.

1.19.3 3. No Dynamic Memory Allocation (*)

In general, TornadoVM cannot allocate memory on demand since it must know the size of the buffers in advanced. However, TornadoVM performs a sort of partial evaluation (PE), in which the JIT compiler evaluates expressions and "materializes" values at runtime. Therefore, if the JIT compiler can obtain, for example, the size of an array, it will generate code based on the values seen at runtime.

Note that other alternatives, such as Aparapi, cannot perform these type of operations due to lack of support for runtime optimizations before generating the OpenCL C code.

1.19.4 4. No Support for Traps/Exceptions (*)

On GPUs there is little support for exceptions. For example, on a division by 0 scenario, the CPU sets a flag in one of the special registers. Then the Operating System can query those special registers and pass that value to the application runtime (in this case, the Java runtime). Then Java runtime handles the exception.

However, there is no such mechanisms on GPUs (link), which means that TornadoVM must insert extra control-flow to guarantee those exceptions never happen. Currently, since TornadoVM compiles at runtime, many of those checks can be assured at runtime. However, we plan to integrate exception support for TornadoVM in the future.

1.19.5 5. No Support for static TaskGraphs and Tasks

TornadoVM currently does not support static `TaskGraph` and Tasks. For example, the code below is not considered valid:

```
public static void testMethod(IntArray in) {
    // ... some code ...
}

static IntArray inTor = IntArray.fromElements(0);
static TaskGraph taskGraph = new TaskGraph("g0");
static {
    taskGraph.task("t0", Main::testMethod, inTor);
}
```

The reason for not supporting this is that a deadlock might occur between the user thread running class initialization and the Tornado compiler thread performing JIT compilation of the Task method. Note that detecting such a deadlock is not trivial and therefore currently, TornadoVM will not issue any error or warning.

Note

Note that if a particular code segment cannot be accelerated with TornadoVM due to unsupported features, then execution falls back to the host JVM which will execute your code on the CPU as it would normally do.

1.20 Resources

1.20.1 Videos

- A. Stratikopoulos, Write Once, Run Anywhere… Well, What About Heterogeneous Hardware?, FOSDEM 2023
- J. Fumero, TornadoVM: Transparent Hardware Acceleration for Java and Beyond!, Devoxx Ukraine 2021.
- J. Fumero, Level up Your Java Performance with TornadoVM, QCon Plus 2021.
- A. Stratikopoulos, TornadoVM: A virtual machine for exploiting high performance heterogeneous hardware of Java programs, FOSDEM, 01/02/2020.
- J. Fumero, TornadoVM: A virtual machine for exploiting high performance heterogeneous hardware, Joker<?> Conference, 26/10/2019.
- J. Fumero, Overview of TornadoVM, JVMLS, 31/07/2019.

1.20.2 Presentations

- TornadoVM Slides @ QConLondon 2020: link
- TornadoVM Slides @ JokerConf<?> 2019: link
- TornadoVM Slides @ JVMLS 2019: link

1.20.3 Podcast

- Foojay Podcast #17: Execute Java Code with TornadoVM on CPUs, GPUs, and FPGAs (March 2023): link
- TornadoVM at Software Engineering Daily (2019): link

1.20.4 Articles

- Article by Juan Fumero, Level up Your Java Performance with TornadoVM, InfoQ, 10/03/2022.
- Article by Juan Fumero, TornadoVM: Accelerating Java with GPUs and FPGAs, InfoQ, 13/06/2020.
- Article by C. Swan, TornadoVM: Running Java on GPUs and FPGAs, InfoQ, 08/03/2020.
- F. Blanaru, J. Fumero, C.Kotselidis, Towards a unified VM for hardware acceleration of managed languages, E2DATA, 20/02/2020.
- J. Fumero, A. Stratikopoulos, C.Kotselidis, TornadoVM: Running your Java programs on heterogeneous hardware, JAXEnter, 17/10/2019.
- C. Kotselidis, TornadoVM: Dynamic Reconfiguration on Heterogeneous Hardware, E2Data, 28/02/2019.

1.20.5 Demos & Artefacts

- TornadoVM Examples: link.
- Examples used for QConLondon: link.
- TornadoVM running with KFusion Microsoft Kinect: link.
- TornadoVM Ray-Tracer: link.
- TornadoVM running with Docker images on NVIDIA GPUs: link.

1.21 Publications

If you are using TornadoVM >= 0.2 (which includes the Dynamic Reconfiguration, the initial FPGA support and CPU/GPU reductions), please use the following citation:

```
@inproceedings{Fumero:DARHH:VEE:2019,
  author = {Fumero, Juan and Papadimitriou, Michail and Zakkak, Foivos S. and Xekalaki,␣
↪Maria and Clarkson, James and Kotselidis, Christos},
  title = {{Dynamic Application Reconfiguration on Heterogeneous Hardware.}},
  booktitle = {Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on␣
↪Virtual Execution Environments},
  series = {VEE '19},
  year = {2019},
  doi = {10.1145/3313808.3313819},
  publisher = {Association for Computing Machinery}
}
```

If you are using Tornado 0.1 (Initial release), please use the following citation in your work.

```
@inproceedings{Clarkson:2018:EHH:3237009.3237016,
  author = {Clarkson, James and Fumero, Juan and Papadimitriou, Michail and Zakkak,␣
↪Foivos S. and Xekalaki, Maria and Kotselidis, Christos and Luj\'{a}n, Mikel},
  title = {{Exploiting High-performance Heterogeneous Hardware for Java Programs Using␣
↪Graal}},
  booktitle = {Proceedings of the 15th International Conference on Managed Languages \&␣
↪Runtimes},
  series = {ManLang '18},
  year = {2018},
  isbn = {978-1-4503-6424-9},
  location = {Linz, Austria},
  pages = {4:1--4:13},
  articleno = {4},
  numpages = {13},
  url = {http://doi.acm.org/10.1145/3237009.3237016},
  doi = {10.1145/3237009.3237016},
  acmid = {3237016},
  publisher = {ACM},
  address = {New York, NY, USA},
  keywords = {Java, graal, heterogeneous hardware, openCL, virtual machine},
}
```

1.21.1 Full list of publications

- Juan Fumero, György Rethy, Athanasios Stratikopoulos, Nikos Foutris, Christos Kotselidis. **Beehive SPIR-V Toolkit: A Composable and Functional API for Runtime SPIR-V Code Generation** VMIL'23, Collocated with SPLASH'#23. October 23, Cascais, Portugal.
- Athanasios Stratikopoulos, Florin Blanaru, Juan Fumero, Maria Xekalaki, Orion Papadakis, Christos Kotselidis. **Cross-Language Interoperability of Heterogeneous Code** MoreVMs'23. Collocated with Programming 2023. Preprint.
- Maria Xekalaki, Juan Fumero, Athanasios Stratikopoulos, Katerina Doka, Christos, Katsakioris, Constantinos Bitsakos, Nectarios Koziris, Christos Kotselidis. **Enabling Transparent Acceleration of Big Data Frameworks Using Heterogeneous Hardware**. 49th International Conference on Very Large Data Bases (VLDB 2023). Preprint.
- Florin Blanaru, Athanasios Stratikopoulos, Juan Fumero, Christos Kotselidis. Enabling Pipeline Parallelism in Heterogeneous Managed Runtime Environments via Batch Processing. Virtual Execution Environments (VEE) 2022.
- Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, Christos Kotselidis. Automatically Exploiting the Memory Hierarchy of GPUs through Just-in-Time Compilation. Virtual Execution Environments (VEE) 2021.
- Michail Papadimitriou, Eleni Markou, Juan Funero, Athanasios Stratikopoulos, Florin Blanaru and Christos Kotselidis. Multiple-Tasks on Multiple-Devices (MTMD): Exploiting Concurrency in Heterogeneous Managed Runtimes. Virtual Execution Environments (VEE) 2021.
- Michail Papadimitriou, Juan Fumero, Athanasios Stratikopoulos, Foivos S. Zakkak, Christos Kotselidis Transparent Compiler and Runtime Specializations for Accelerating Managed Languages on FPGAs. The Art, Science, and Engineering of Programming 2021.
- Athanasios Stratikopoulos, Mihai-Cristian Olteanu, Ian Vaughan, Zoran Sevarac, Nikos Foutris, Juan Fumero, Christos Kotselidis. Transparent Acceleration of Java-based Deep Learning Engines.. International Conference on Managed Programming Languages & Runtimes (MPLR'20).
- Juan Fumero, Michail Papadimitriou, Foivos Zakkak, Maria Xekalaki, James Clarkson, Christos Kotselidis. Dynamic Application Reconfiguration on Heterogeneous Hardware.. In Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19). Preprint.
- M. Papadimitriou, J. Fumero, A. Stratikopoulos and C. Kotselidis. Towards Prototyping and Acceleration of Java Programs onto Intel FPGAs. IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'19). Preprint.
- M. Xekalaki, J. Fumero and C. Kotselidis. Challenges and Proposals for Enabling Dynamic Heterogeneous Execution of Big Data Frameworks. 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). DOI.
- Juan Fumero, Christos Kotselidis. Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware: A Java Reduction Case Study. In Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'18).
- James Clarkson, Juan Fumero, Michalis Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, Mikel Luján (The University of Manchester). **Exploiting High-Performance Heterogeneous Hardware for Java Programs using Graal**. *Proceedings of the 15th International Conference on Managed Languages & Runtime (ManLang'18).* Preprint. DOI.
- Sajad Saeedi, Bruno Bodin, Harry Wagstaff, Andy Nisbet, Luigi Nardi, John Mawer, Nicolas Melot, Oscar Palomar, Emanuele Vespa, Tom Spink, Cosmin Gorgovan, Andrew Webb, James Clarkson, Erik Tomusk, Thomas Debrunner, Kuba Kaszyk, Pablo Gonzalez-de-Aledo, Andrey Rodchenko, Graham Riley, Christos Kotselidis, Björn Franke, Michael FP O'Boyle, Andrew J Davison, Paul HJ Kelly, Mikel Luján, Steve Furber. **Navigating**

the Landscape for Real-Time Localization and Mapping for Robotics and Virtual and Augmented Reality. In Proceedings of the IEEE, 2018.

- C. Kotselidis, J. Clarkson, A. Rodchenko, A. Nisbet, J. Mawer, and M. Luján. 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-DL.

1.22 TornadoVM Changelog

This file summarizes the new features and major changes for each *TornadoVM* version.

1.22.1 TornadoVM 1.0.2

29/02/2024

Improvements

- #323: Set Accelerator Memory Limit per Execution Plan at the API level
- #328: Javadoc API to run with concurrent devices and memory limits
- #340: New API calls to enable `threadInfo` and `printKernel` from the Execution Plan API.
- #334: Dynamically enable/disable profiler after first run

Compatibility

- #337 : Initial support for Graal and JDK 21.0.2

Bug Fixes

- #322: Fix duplicate thread-info debug message when the debug option is also enabled.
- #325: Set/Get accesses for the `MatrixVectorFloat4` type fixed
- #326: Fix installation script for running with Python >= 3.12
- #327: Fix Memory Limits for all supported Panama off-heap types.
- #329: Fix timers for the dynamic reconfiguration policies
- #330: Fix the profiler logs when silent mode is enabled
- #332: Fix Batch processing when having multiple task-graphs in a single execution plan.

1.22.2 TornadoVM 1.0.1

30/01/2024

Improvements

- #305: Under-demand data transfer for custom data ranges.
- #313: Initial support for Half-Precision (FP16) data types.
- #311: Enable Multi-Task Multiple Device (MTMD) model from the `TornadoExecutionPlan` API:
- #315: Math `Ceil` function added

Compatibility/Integration

- #294: Separation of the OpenCL Headers from the code base.
- #297: Separation of the LevelZero JNI API in a separate repository.
- #301: Temurin configuration supported.
- #304: Refactor of the common phases for the JIT compiler.
- #316: Beehive SPIR-V Toolkit version updated.

Bug Fixes

- #298: OpenCL Codegen fixed open-close brackets.
- #300: Python Dependencies fixed for AWS
- #308: Runtime check for Grid-Scheduler names
- #309: Fix check-style to support STR templates
- #314: emit Vector16 Capability for 16-width vectors

1.22.3 TornadoVM 1.0

05/12/2023

Improvements

- Brand-new API for allocating off-heap objects and array collections using the Panama Memory Segment API. - New Arrays, Matrix and Vector type objects are allocated using the Panama API. - Migration of existing applications to use the new Panama-based types: https://tornadovm.readthedocs.io/en/latest/offheap-types.html
- Handling of the TornadoVM's internal bytecode improved to avoid write-only copies from host to device.
- `cospi` and `sinpi` math operations supported for OpenCL, PTX and SPIR-V.
- Vector 16 data types supported for `float`, `double` and `int`.
- Support for Mesa's `rusticl`.
- Device default ordering improved based on maximum thread size.
- Move all the installation and configuration scripts from Bash to Python.

- The installation process has been improved for Linux and OSx with M1/M2 chips.
- Documentation improved.
- Add profiling information for the testing scripts.

Compatibility/Integration

- Integration with the Graal 23.1.0 JIT Compiler.
- Integration with OpenJDK 21.
- Integration with Truffle Languages (Python, Ruby and Javascript) using Graal 23.1.0.
- TornadoVM API Refactored.
- Backport bug-fixes for branch using OpenJDK 17: `master-jdk17`

Bug fixes:

- Multiple SPIR-V Devices fixed.
- Runtime Exception when no SPIR-V devices are present.
- Issue with the kernel context API when invoking multiple kernels fixed.
- MTMD mode is fixed when running multiple backends on the same device.
- `long` type as a constant parameter for a kernel fixed.
- FPGA Compilation and Execution fixed for AWS and Xilinx devices.
- Batch processing fixed for different data types of the same size.

1.22.4 TornadoVM 0.15.2

26/07/2023

Improvements

- Initial Support for Multi-Tasks on Multiple Devices (MTMD): This mode enables the execution of multiple independent tasks on more than one hardware accelerators. Documentation in link: https://tornadovm.readthedocs.io/en/latest/multi-device.html
- Support for trigonometric `radian`, `cospi` and `sinpi` functions for the OpenCL/PTX and SPIR-V backends.
- Clean-up Java modules not being used and TornadoVM core classes refactored.

Compatibility/Integration

- Initial integration with ComputeAorta (part of the Codeplay's oneAPI Construction Kit for RISC-V) to run on RISC-V with Vector Instructions (OpenCL backend) in emulation mode.
- Beehive SPIR-V Toolkit dependency updated.
- Tests for prebuilt SPIR-V kernels fixed to dispatch SPIR-V binaries through the Level Zero and OpenCL runtimes.
- Deprecated `javac.py` script removed.

Bug fixes:

- TornadoVM OpenCL Runtime throws an exception when the detected hardware does not support FP64.
- Fix the installer for the older Apple with the x86 architecture using AMD GPUs.
- Installer for ARM based systems fixed.
- Installer fixed for Microsoft WSL and NVIDIA GPUs.
- OpenCL code generator fixed to avoid using the reserved OpenCL keywords from Java function parameters.
- Dump profiler option fixed.

1.22.5 TornadoVM 0.15.1

15/05/2023

Improvements

- Introduction of a device selection heuristic based on the computing capabilities of devices. TornadoVM selects, as the default device, the fastest device based on its computing capability.
- Optimisation of removing redundant data copies for Read-Only and Write-Only buffers from between the host (CPU) and the device (GPU) based on the Tornado Data Flow Graph.
- New installation script for TornadoVM.
- Option to dump the TornadoVM bytecodes for the unit tests.
- Full debug option improved. Use `--fullDebug`.

Compatibility/Integration

- Integration and compatibility with the Graal 22.3.2 JIT Compiler.
- Improved compatibility with Apple M1 and Apple M2 through the OpenCL Backend.
- GraalVM/Truffle programs integration improved. Use `--truffle` in the `tornado` script to run guest programs with Truffle. Example: `tornado --truffle python myProgram.py` Full documentation: https://tornadovm.readthedocs.io/en/latest/truffle-languages.html

Bug fixes:

- Documentation that resets the device's memory: https://github.com/beehive-lab/TornadoVM/blob/master/tornado-api/src/main/java/uk/ac/manchester/tornado/api/TornadoExecutionPlan.java#L282
- Append the Java `CLASSPATH` to the `cp` option from the `tornado` script.
- Dependency fixed for the `cmake-maven` plugin fixed for ARM-64 arch.
- Fixed the automatic installation for Apple M1/M2 and ARM-64 and NVIDIA Jetson nano computing systems.
- Integration with IGV fixed. Use the `--igv` option for the `tornado` and `tornado-test` scripts.

1.22.6 TornadoVM 0.15

27/01/2023

Improvements

- New TornadoVM API:
  - API refactoring (`TaskSchedule` has been renamed to `TaskGraph`)
  - Introduction of the Immutable `TaskGraphs`
  - Introduction of the TornadoVM Execution Plans: (`TornadoExecutionPlan`)
  - The documentation of migration of existing TornadoVM applications to the new API can be found here: https://tornadovm.readthedocs.io/en/latest/programming.html#migration-to-tornadovm-v0-15
- Launch a new website https://tornadovm.readthedocs.io/en/latest/ for the documentation
- Improved documentation
- Initial support for Intel ARC discrete GPUs.
- Improved TornadoVM installer for Linux
- ImprovedTornadoVM launch script with optional parameters
- Support of large buffer allocations with Intel Level Zero. Use: `tornado.spirv.levelzero.extended.memory=True`

Bug fixes:

- Vector and Matrix types
- TornadoVM Floating Replacement compiler phase fixed
- Fix `CMAKE` for Intel ARC GPUs
- Device query tool fixed for the PTX backend
- Documentation for Windows 11 fixed

1.22.7 TornadoVM 0.14.1

29/09/2022

Improvements

- The tornado command is replaced from a Bash to a Python script.
  - Use `tornado --help` to check the new options and examples.
- Support of native tests for the SPIR-V backend.
- Improvement of the OpenCL and PTX tests of the internal APIs.

Compatibility/Integration

- Integration and compatibility with the Graal 22.2.0 JIT Compiler.
- Compatibility with JDK 18 and JDK 19.
- Compatibility with Apple M1 Pro using the OpenCL backend.

Bug Fixes

- CUDA PTX generated header fixed to target NVIDIA 30xx GPUs and CUDA 11.7.
- The signature of generated PTX kernels fixed for NVIDIA driver >= 510 and 30XX GPUs when using the TornadoVM Kernel API.
- Tests of virtual OpenCL devices fixed.
- Thread deployment information for the OpenCL backend is fixed.
- `TornadoVMRuntimeCI` moved to `TornadoVMRutimeInterface`.

1.22.8 TornadoVM 0.14

15/06/2022

New Features

- New device memory management for addressing the memory allocation limitations of OpenCL and enabling pinned memory of device buffers.
  - The execution of task-schedules will still automatically allocate/deallocate memory every time a task-schedule is executed, unless lock/unlock functions are invoked explicitly at the task-schedule level.
  - One heap per device has been replaced with a device buffer per input variable.
  - A new API call has been added for releasing memory: `unlockObjectFromMemory`
  - A new API call has been added for locking objects to the device: `lockObjectInMemory` This requires the user to release memory by invoking `unlockObjectFromMemory` at the task-schedule level.
- Enhanced Live Task migration by supporting multi-backend execution (PTX <-> OpenCL <-> SPIR-V).

Compatibility/Integration

- Integration with the Graal 22.1.0 JIT Compiler
- JDK 8 deprecated
- Azul Zulu JDK supported
- OpenCL 2.1 as a default target for the OpenCL Backend
- Single Docker Image for Intel XPU platforms, including the SPIR-V backend (using the Intel Integrated Graphics), and OpenCL (using the Intel Integrated Graphics, Intel CPU and Intel FPGA in emulation mode). Image: https://github.com/beehive-lab/docker-tornado#intel-integrated-graphics

Improvements/Bug Fixes

- `SIGNUM` Math Function included for all three backends.
- SPIR-V optimizer enabled by default (3x reduce in binary size).
- Extended Memory Mode enabled for the SPIR-V Backend via Level Zero.
- Phi instructions fixed for the SPIR-V Backend.
- SPIR-V Vector Select instructions fixed.
- Duplicated IDs for Non-Inlined SPIR-V Functions fixed.
- Refactoring of the TornadoVM Math Library.
- FPGA Configuration files fixed.
- Bitwise operations for OpenCL fixed.
- Code Generation Times and Backend information are included in the profiling info.

1.22.9 TornadoVM 0.13

21/03/2022

- Integration with JDK 17 and Graal 21.3.0
  - JDK 11 is the default version and the support for the JDK 8 has been deprecated
- Support for extended intrinsics regarding math operations
- Native functions are enabled by default
- Support for 2D arrays for PTX and SPIR-V backends:
  - https://github.com/beehive-lab/TornadoVM/commit/2ef32ca97941410672720f9dfa15f0151ae2a1a1
- Integer Test Move operation supported:
  - https://github.com/beehive-lab/TornadoVM/pull/177
- Improvements in the SPIR-V Backend:
  - Experimental SPIR-V optimizer. Binary size reduction of up to 3x
    - https://github.com/beehive-lab/TornadoVM/commit/394ca94dcdc3cb58d15a17046e1d22c6389b55b7
  - Fix malloc functions for Level-Zero
  - Support for pre-built SPIR-V binary modules using the TornadoVM runtime for OpenCL

  - Performance increase due to cached buffers on GPUs by default
  - Disassembler option for SPIR-V binary modules. Use `--printKernel`
- Improved Installation:
  - Full automatic installer script integrated
- Documentation about the installation for Windows 11
- Refactoring and several bug fixes
  - https://github.com/beehive-lab/TornadoVM/commit/57694186b42ec28b16066fb549ab8fcf9bff9753
  - Vector types fixed:
    * https://github.com/beehive-lab/TornadoVM/pull/181/files
    * https://github.com/beehive-lab/TornadoVM/commit/004d61d6d26945b45ebff66641b60f90f00486be
  - Fix AtomicInteger get for OpenCL:
    * https://github.com/beehive-lab/TornadoVM/pull/177
- Dependencies for Math3 and Lang3 updated

1.22.10 TornadoVM 0.12

17/11/2021

- New backend: initial support for SPIR-V and Intel Level Zero
  - Level-Zero dispatcher for SPIR-V integrated
  - SPIR-V Code generator framework for Java
- Benchmarking framework improved to accommodate all three backends
- Driver metrics, such as kernel time and data transfers included in the benchmarking framework
- TornadoVM profiler improved:
  - Command line options added: `--enableProfiler <silent|console>` and `--dumpProfiler <jsonFile>`
  - Logging improve for debugging purposes. JIT Compiler, JNI calls and code generation
- New math intrinsincs operations supported
- Several bug fixes:
  - Duplicated barriers removed. TornadoVM BARRIER bytecode fixed when running multi-context
  - Copy in when having multiple reductions fixed
  - TornadoVM profiler fixed for multiple context switching (device switching)
- Pretty printer for device information

1.22.11 TornadoVM 0.11

29/09/2021

- TornadoVM JIT Compiler upgrade to work with Graal 21.2.0 and JDK 8 with JVMCI 21.2.0
- Refactoring of the Kernel Parallel API for Heterogeneous Programming:
  - Methods `getLocalGroupSize(index)` and `getGlobalGroupSize` moved to public fields to keep consistency with the rest of the thread properties within the `KernelContext` class.
    - Changeset: https://github.com/beehive-lab/TornadoVM/commit/e1ebd66035d0722ca90eb0121c55dbc744840a74
- Compiler update to register the global number of threads: https://github.com/beehive-lab/TornadoVM/pull/133/files
- Simplification of the TornadoVM events handler: https://github.com/beehive-lab/TornadoVM/pull/135/files
- Renaming the Profiler API method from `event.getExecutionTime` to `event.getElapsedTime`: https://github.com/beehive-lab/TornadoVM/pull/134
- Deprecating `OCLWriteNode` and `PTXWriteNode` and fixing stores for bytes: https://github.com/beehive-lab/TornadoVM/pull/131
- Refactoring of the FPGA IR extensions, from the high-tier to the low-tier of the JIT compiler
  - Utilizing the FPGA Thread-Attributes compiler phase for the FPGA execution
  - Using the `GridScheduler` object (if present) or use a default value (e.g., 64, 1, 1) for defining the FPGA OpenCL local workgroup
- Several bugs fixed:
  - Codegen for sequential kernels fixed
  - Function parameters with non-inlined method calls fixed

1.22.12 TornadoVM 0.10

29/06/2021

- TornadoVM JIT Compiler sync with Graal 21.1.0
- Experimental support for OpenJDK 16
- Tracing the TornadoVM thread distribution and device information with a new option `--threadInfo` instead of `--debug`
- Refactoring of the new API:
  - `TornadoVMExecutionContext` renamed to `KernelContext`
  - `GridTask` renamed to `GridScheduler`
- AWS F1 AMI version upgraded to 1.10.0 and automated the generation of AFI image
- Xilinx OpenCL backend expanded with:
  - a) **Initial integration of Xilinx OpenCL attributes for loop** pipelining in the TornadoVM compiler
  - b) Support for multiple compute units
- Logging FPGA compilation option added to dump FPGA HLS compilation to a file
- TornadoVM profiler enhanced for including data transfers for the stack-frame and kernel dispatch time

- Initial support for 2D Arrays added
- Several bug fixes and stability support for the OpenCL and PTX backends

1.22.13 TornadoVM 0.9

15/04/2021

- Expanded API for expressing kernel parallelism within Java. It can work with the existing loop parallelism in TornadoVM.
  - Direct access to thread-ids, OpenCL local memory (PTX shared memory), and barriers
  - `TornadoVMContext` added:

    See https://github.com/beehive-lab/TornadoVM/blob/5bcd3d6dfa2506032322c32d72b7bbd750623a95/tornado-api/src/main/java/uk/ac/manchester/tornado/api/TornadoVMContext.java
  - Code examples:
    * https://github.com/beehive-lab/TornadoVM/tree/master/examples/src/main/java/uk/ac/manchester/tornado/examples/tornadovmcontext
  - Documentation:
    * https://github.com/beehive-lab/TornadoVM/blob/master/assembly/src/docs/21_TORNADOVM_CONTEXT.md
- Profiler integrated with Chrome debug:
  - Use flags: `-Dtornado.chrome.event.tracer.enabled=True -Dtornado.chrome.event.tracer.filename=userFile.json`
  - See https://github.com/beehive-lab/TornadoVM/pull/41
- Added support for Windows 10:
  - See https://github.com/beehive-lab/TornadoVM/blob/develop/assembly/src/docs/20_INSTALL_WINDOWS_WITH_GRAALVM.md
- TornadoVM running with Windows JDK 11 supported (Linux & Windows)
- Xilinx FPGAs workflow supported for Vitis 2020.2
- Pre-compiled tasks for Xilinx/Intel FPGAs fixed
- Slambench fixed when compiling for PTX and OpenCL backends
- Several bug fixes for the runtime, JIT compiler and data management.

1.22.14 TornadoVM 0.8

19/11/2020

- Added PTX backend for NVIDIA GPUs
  - Build TornadoVM using `make BACKEND=ptx,opencl` to obtain the two supported backends.
- TornadoVM JIT Compiler aligned with Graal 20.2.0
- Support for other JDKs:

  - Red Hat Mandrel 11.0.9
  - Amazon Coretto 11.0.9
  - GraalVM LabsJDK 11.0.8
  - OpenJDK 11.0.8
  - OpenJDK 12.0.2
  - OpenJDK 13.0.2
  - OpenJDK 14.0.2
- Support for hybrid (CPU-GPU) parallel reductions
- New API for generic kernel dispatch. It introduces the concept of `WorkerGrid` and `GridTask`
  - A `WorkerGrid` is an object that stores how threads are organized on an OpenCL device: `java WorkerGrid1D worker1D = new WorkerGrid1D(4096);`
  - A `GridTask` is a map that relates a task-name with a worker-grid. `java GridTask gridTask = new GridTask(); gridTask.set("s0.t0", worker1D);`
  - A TornadoVM Task-Schedule can be executed using a `GridTask`: `java ts.execute(gridTask);`
  - More info: link
- TornadoVM profiler improved
  - Profiler metrics added
  - Code features per task-graph
- Lazy device initialisation moved to early initialisation of PTX and OpenCL devices
- Initial support for Atomics (OpenCL backend)
  - Link to examples
- Task Schedules with 11-14 parameters supported
- Documentation improved
- Bug fixes for code generation, numeric promotion, basic block traversal, Xilinx FPGA compilation.

1.22.15 TornadoVM 0.7

22/06/2020

- Support for ARM Mali GPUs.
- Support parallel reductions on FPGAs
- Agnostic FPGA vendor compilation via configuration files (Intel & Xilinx)
- Support for AWS on Xilinx FPGAs
- Recompilation for different input data sizes supported
- New TornadoVM API calls:
  a) Update references for re-compilation: `taskSchedule.updateReferences(oldRef, newRef);`
  b) Use the default OpenCL scheduler: `taskSchedule.useDefaultThreadScheduler(true);`

- Use of JMH for benchmarking
- Support for Fused Multiply-Add (FMA) instructions
- Easy-selection of different devices for unit-tests `tornado-test.py -V --debug -J"-Dtornado.unittests.device=0:1"`
- Bailout mechanism improved from parallel to sequential
- Improve thread scheduling
- Support for private memory allocation
- Assertion mode included
- Documentation improved
- Several bug fixes

1.22.16 TornadoVM 0.6

21/02/2020

- TornadoVM compatible with GraalVM 19.3.0 using JDK 8 and JDK 11
- TornadoVM compiler update for using Graal 19.3.0 compiler API
- Support for dynamic languages on top of Truffle
  - examples
- Support for multiple tasks per task-schedule on FPGAs (Intel and Xilinx)
- Support for OSX Mojave and Catalina
- Task-schedule name handling for FPGAs improved
- Exception handling improved
- Reductions for `long` type supported
- Bug fixes for ternary conditions, reductions and code generator
- Documentation improved

1.22.17 TornadoVM 0.5

16/12/2019

- Initial support for Xilinx FPGAs
- TornadoVM API classes are now `Serializable`
- Initial support for local memory for reductions
- JVMCI built with local annotation patch removed. Now TornadoVM requires unmodified JDK8 with JVMCI support
- Support of multiple reductions within the same `task-schedules`
- Emulation mode on Intel FPGAs is fixed
- Fix reductions on Intel Integrated Graphics
- TornadoVM driver OpenCL initialization and OpenCL code cache improved

- Refactoring of the FPGA execution modes (full JIT and emulation modes improved).

1.22.18 TornadoVM 0.4

14/10/2019

- Profiler supported
  - Use `-Dtornado.profiler=True` to enable profiler
  - Use `-Dtornado.profiler=True -Dtornado.profiler.save=True` to dump the profiler logs
- Feature extraction added
  - Use `-Dtornado.feature.extraction=True` to enable code extraction features
- Mac OSx support
- Automatic reductions composition (map-reduce) within the same task-schedule
- Bug related to a memory leak when running on GPUs solved
- Bug fixes and stability improvements

1.22.19 TornadoVM 0.3

22/07/2019

- New Matrix 2D and Matrix 3D classes with type specializations.
- New API-call `TaskSchedule#batch` for batch processing. It allows programmers to run with more data than the maximum capacity of the accelerator by creating batches of executions.
- FPGA full automatic compilation pipeline.
- FPGA options simplified:
  - `-Dtornado.precompiled.binary=<binary>` for loading the bitstream.
  - `-Dtornado.opencl.userelative=True` for using relative addresses.
  - `-Dtornado.opencl.codecache.loadbin=True` *removed*.
- Reductions support enhanced and fully automated on GPUs and CPUs.
- Initial support for reductions on FPGAs.
- Initial API for profiling tasks integrated.

1.22.20 TornadoVM 0.2

25/02/2019

- Rename to TornadoVM
- Device selection for better performance (CPU, multi-core, GPU, FPGA) via an API for Dynamic Reconfiguration
  - Added methods `executeWithProfiler` and `executeWithProfilerSequential` with an input policy.
  - Policies: `Policy.PERFORMANCE`, `Policy.END_2_END`, and `Policy.LATENCY` implemented.
- Basic heuristic for predicting the highest performing target device with Dynamic Reconfiguration
- Initial FPGA integration for Altera FPGAs:

  - Full JIT compilation mode
  - Ahead of time compilation mode
  - Emulation/debug mode
- FPGA JIT compiler specializations
- Added support for Java reductions:
  - Compiler specializations for CPU and GPU reductions
- Performance and stability fixes

1.22.21 Tornado 0.1.0

07/09/2018

- Initial Implementation of the Tornado compiler
- Initial GPU/CPU code generation for OpenCL
- Initial support in the runtime to execute OpenCL programs generated by the Tornado JIT compiler
- Initial Tornado-API release (`@Parallel` Java annotation and `TaskSchedule` API)
- Multi-GPU enabled through multiple tasks-schedules