Examples in Tornado

In the Tornado SDK you can find a number of examples in the `examples` directory. This is a copy of the unittesting framework of Tornado for data types, arrays, and other utilities.

This document describes how to code and run a full example in Tornado.

1. Run a simple example within Tornado: Vector Addition

Below you can find a snapshot of the TestArrays example code in Tornado (full code listing can be found in the `examples` directory). In this example, we will run the `vectorAddDouble` method on a heterogeneous device. Tornado will dynamically compile and run the Java code (of the `vectorAddDouble` method) to an OpenCL device. During the execution process, the code will be compiled from Java bytecode to OpenCL C and afterwards it will run on the OpenCL-compatible device, transparently.

As you can see in the example below, the `accelerated vectorAddDouble` method performs a double vector addition. Furthermore, it does not differ at all from a vanilla sequential Java implementation of the method. The only difference is the addition of the `@Parallel` annotation that instructs Tornado that the loop has to be computed in parallel (eg. using the global identifier in OpenCL).

The `testVectorAdditionDouble` method prepares the input data and creates a Tornado `task`. Tornado `tasks` can not execute directly; instead they must be part of a `TaskSchedule`. This is a design choice allowing a number of optimizations, such as task pipelining and parallelism, to be performed. Furthermore, `TaskSchedules` define which parameters are copied in and out from the device.

Once the method `execute` is invoked, Tornado builds the data dependency graph, the Tornado bytecode, compiles the referenced Java method to OpenCL C, and executes the generated application on the available OpenCL device.

```
package examples;

import java.util.Random;
import java.util.stream.IntStream;

import uk.ac.manchester.tornado.api.Parallel;
import uk.ac.manchester.tornado.runtime.api.TaskSchedule;

public class TestTornado {

  public static void main(String[] args) {
    testVectorAdditionDouble(4096);
```

```
  }


  // Method to be executed on the parallel device (eg. GPU)
  public static void vectorAddDouble(double[] a, double[] b, double[] c) {
    for (@Parallel int i = 0; i < c.length; i++) {
        c[i] = a[i] + b[i];
    }
  }

  public static void testVectorAdditionDouble(int size) {

    double[] a = new double[size];
    double[] b = new double[size];
    double[] c = new double[size];

    Random r = new Random();
    IntStream.range(0, size).sequential().forEach(i -> {
        a[i] = r.nextDouble();
        b[i] = r.nextDouble();
    });

    // Tornado Task API
    new TaskSchedule("s0") // new group of Tasks
        .streamIn(a, b)        // copy in from the host to the device (a and b arrays)
        .task("t0", TestTornado::vectorAddDouble, a, b, c)   // task 0
        .streamOut(c)          // copy out from the device to host
        .execute();            // run the task (Tornado bytecode generation, Tornado
                               // tasks graph, OpenCL JIT compilation and execution)
  }
}
```

2. Compiling and Running with Tornado SDK

The example above is already provided in the `examples` directory. To compile with Tornado SDK, there is a utility command that sets all the `CLASSPATHs` to use Tornado. Alternatively, you can use the standard JDK 1.8 and define all jars in `share/java/tornado` into your `CLASSPATHs`.

```
$ javac.py examples/TestTornado.java
```

To run, just execute `tornado`. If you want to see the auto-generated OpenCL C code, you can run with the following option:

```
$ tornado --printKernel --debug examples/TestTornado
```

The `--debug` option will print in which device the kernel was executed (e.g. GPU or CPU).

3. Vector Addition using Vector Types

Tornado API exposes a set of data structures to developers to use specific vector operations such as addition, multiplication, etc. The simple algorithm of vector addition can be rewritten to use Tornado vector types. The Tornado JIT compiler will generate OpenCL vector types that match the Tornado vector types.

The following snippet shows the vector addition example using Tornado vector types.

```
public static void addVectorFloat4(VectorFloat4 a, VectorFloat4 b,
                                   VectorFloat4 results) {
  for (@Parallel int i = 0; i < a.getLength(); i++) {
    results.set(i, Float4.add(a.get(i), b.get(i)));
  }
}
```

The type `VectorFloat4` is collection, in Tornado, that contains a list of `Float4` element types. When Tornado compiles this code to OpenCL, it will use the OpenCL type `float4`. Note that `Float4` provides a static method called `add`. These are intrinsics to the compiler.

Tornado exposes `Float2`, `Float3`, `Float4`, `Float6` and `Float8` vector types. Vector operations are also exposed for int and double (e.g `Double8`, `Int4`). More examples using vector types are provided in the `examples` directory.

The following code shows a snippet of the generated OpenCL C code using the vector types. First, it loads the data from global memory to local memory for the two input arrays. Then it performs the addition and finally it stores the result in the new position in global memory.

```
v4f_18 = vload4(0, (__global float *) ul_17);  // <- float4 load
ul_19  =  ul_1 + 24L;
ul_20  =  *((__global ulong *) ul_19);
ul_21  =  ul_20 + l_16;
v4f_22  =  vload4(0, (__global float *) ul_21);
ul_23  =  ul_2 + 24L;
ul_24  =  *((__global ulong *) ul_23);
ul_25  =  ul_24 + l_16;
f_26  =  v4f_18.s0 + v4f_22.s0;                // <- float4 computation
f_27  =  v4f_18.s1 + v4f_22.s1;
f_28  =  v4f_18.s2 + v4f_22.s2;
f_29  =  v4f_18.s3 + v4f_22.s3;
v4f_30  =  (float4)(f_26, f_27, f_28, f_29);
vstore4(v4f_30, 0, (__global float *) ul_25);  // <- float4 store
```

4. Mandelbrot

Tornado allows nested `@Parallel` loops as follows:

```
private static void mandelbrotTornado(int size, short[] output) {
    final int iterations = 10000;
    float space = 2.0f / size;

    // This will be mapped to 2D kernel in OpenCL
    for (@Parallel int i = 0; i < size; i++) {
        for (@Parallel int j = 0; j < size; j++) {
            float Zr = 0.0f;
            float Zi = 0.0f;
            float Cr = (1 * j * space - 1.5f);
            float Ci = (1 * i * space - 1.0f);

            float ZrN = 0;
            float ZiN = 0;
            int y = 0;

            for (y = 0; y < iterations; y++) {
                float s = ZiN + ZrN;
                if (s > 4.0f) {
                    break;
                } else {
                Zi = 2.0f * Zr * Zi + Ci;
                Zr = 1 * ZrN - ZiN + Cr;
                ZiN = Zi * Zi;
                ZrN = Zr * Zr;

                }

            }
            short r = (short) ((y * 255) / iterations);
            output[i * size + j] = r;
        }
    }
}
```

5. Parallel Breadth-First Search (BFS) within Tornado

The following code shows the core method for the parallel BFS using Tornado. Note that the only two annotations needed are in the loops to indicate 2D kernel on the GPU.

This algorithm receives an input adjacency matrix and an array with the current

depth (depth per level in a graph) and updates the depth of the current node. This is also an iterative algorithm that will keep computing till the variable `h_true` does not change.

```
private static void runBFS(int[] vertices, int[] adjacencyMatrix, int numNodes,
                           int[] h_true, int[] currentDepth) {

  for (@Parallel int from = 0; from < numNodes; from++) {

      for (@Parallel int to = 0; to < numNodes; to++) {
          int elementAccess = from * numNodes + to;

          if (adjacencyMatrix[elementAccess] == 1) {
              int dfirst = vertices[from];
              int dsecond = vertices[to];
              if ((currentDepth[0] == dfirst) && (dsecond == -1)) {
                  vertices[to] = dfirst + 1;
                  h_true[0] = 0;
              }

              if (BIDIRECTIONAL) {
                if ((currentDepth[0] == dsecond) && (dfirst == -1)) {
                    vertices[from] = dsecond + 1;
                    h_true[0] = 0;
                }
              }
          }
      }
  }
}
```

The following Java snippet shows the data preparation, task definition, and invocation in Tornado.

```
   public void tornadoBFS(int rootNode, int numNodes) throws IOException {

        vertices = new int[numNodes];
        adjacencyMatrix = new int[numNodes * numNodes];

        if (SAMPLE) {
            initilizeAdjacencyMatrixSimpleGraph(adjacencyMatrix, numNodes);
        } else {
            generateRandomGraph(adjacencyMatrix, numNodes, rootNode);
        }

        // Step 1: vertices initialisation
        initializeVertices(numNodes, vertices, rootNode);
```

```
        TaskSchedule s0 = new TaskSchedule("s0");
        s0.task("t0", BFS::initializeVertices, numNodes, vertices, rootNode);
        s0.streamOut(vertices).execute();

        modify = new int[] { 1 };
        Arrays.fill(modify, 1);

        currentDepth = new int[] { 0 };

        TornadoDevice device = TornadoRuntime.getTornadoRuntime()
                           .getDefaultDevice();
        TaskSchedule s1 = new TaskSchedule("s1");
        s1.streamIn(vertices, adjacencyMatrix, modify,currentDepth)
                    .mapAllTo(device);
        s1.task("t1", BFS::runBFS, vertices, adjacencyMatrix,
            numNodes, modify, currentDepth);
        s1.streamOut(vertices, modify);

        boolean done = false;

        while (!done) {
            // 2. Parallel BFS
            boolean allDone = true;
            System.out.println("Current Depth: " + currentDepth[0]);
            //runBFS(vertices, adjacencyMatrix, numNodes, modify, currentDepth);
            s1.execute();
            currentDepth[0]++;
            for(int i = 0; i < modify.length; i++) {
                if (modify[i] == 0) {
                    allDone &= false;
                    break;
                }
            }

            if (allDone) {
                done = true;
            }
            Arrays.fill(modify, 1);
        }

        if (PRINT_SOLUTION) {
            System.out.println("Solution: " + Arrays.toString(vertices));
        }
    }
```