MANCHESTER
1824
The University of Manchester

Tornado VM: *A Virtual Machine for Exploiting High-Performance Heterogeneous Hardware of Java Programs*

Juan Fumero
Postdoc @ The University of Manchester, UK
<juan.fumero@manchester.ac.uk>
Twitter: @snatverk
JVMLS 2019, July 30th

Agenda

- Motivation & Background
- TornadoVM
    - API examples
    - Runtime
    - JIT Compiler
    - Dynamic reconfiguration
    - Data Management
- Performance Results
- Related Work
- Notes for Discussion of JEP 8047074 (GPU Integration)
- Conclusions

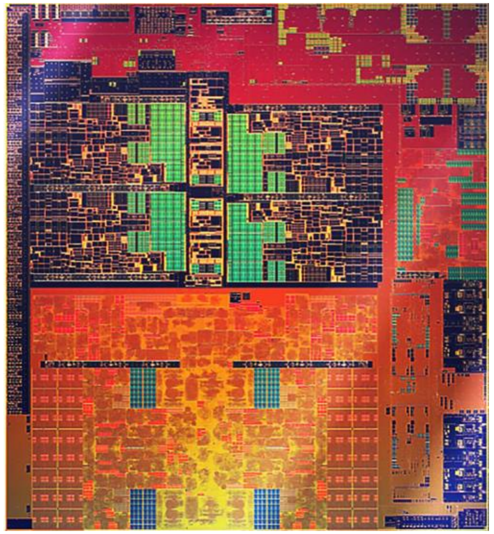
About me

- Postdoc @ The University of Manchester (Since October 2017)
  - Currently technical lead of TornadoVM
- 2014-2017: PhD in Dynamic Compilation for GPUs using Graal & Truffle (Java, R, Ruby) @ The University of Edinburgh
- Oracle Labs alumni (worked on Truffle FastR + Flink for distributed computing)
- CERN OpenLab on the evaluation of the CilkPlus compiler for the ROOT physics framework

Motivation

Why should we care about GPUs/FPGAs, etc.?

CPU



Intel Ice Lake (10nm)
8 cores HT, AVX(512 SIMD)
~1TFlops* (including the iGPU)
~ TDP 28W

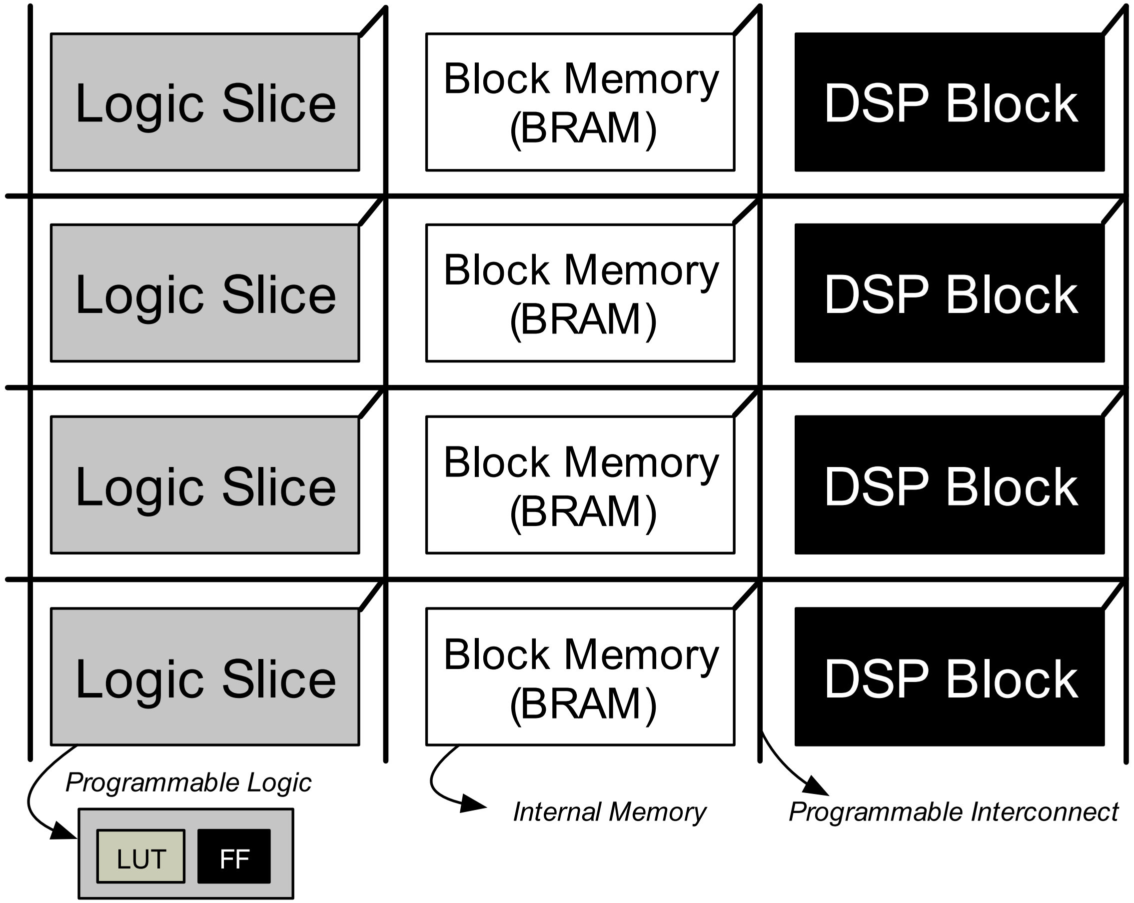
GPU

NVIDIA GP 100 – Pascal - 16nm
60 SMs, 64 cores each
3584 FP32 cores
10.6 TFlops (FP32)
TDP ~300 Watts
https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

FPGA

Intel FPGA Stratix 10 (14nm)
Reconfigurable Hardware
~ 10 TFlops
TDP ~225Watts

What is an FPGA? Field Programmable Gate Array



You can configure the design of your hardware after manufacturing

It is like having "*your algorithms directly wired on hardware*" with only the parts you need

Industry is pushing for OpenCL on FPGAs!

MANCHESTER 1824
The University of Manchester

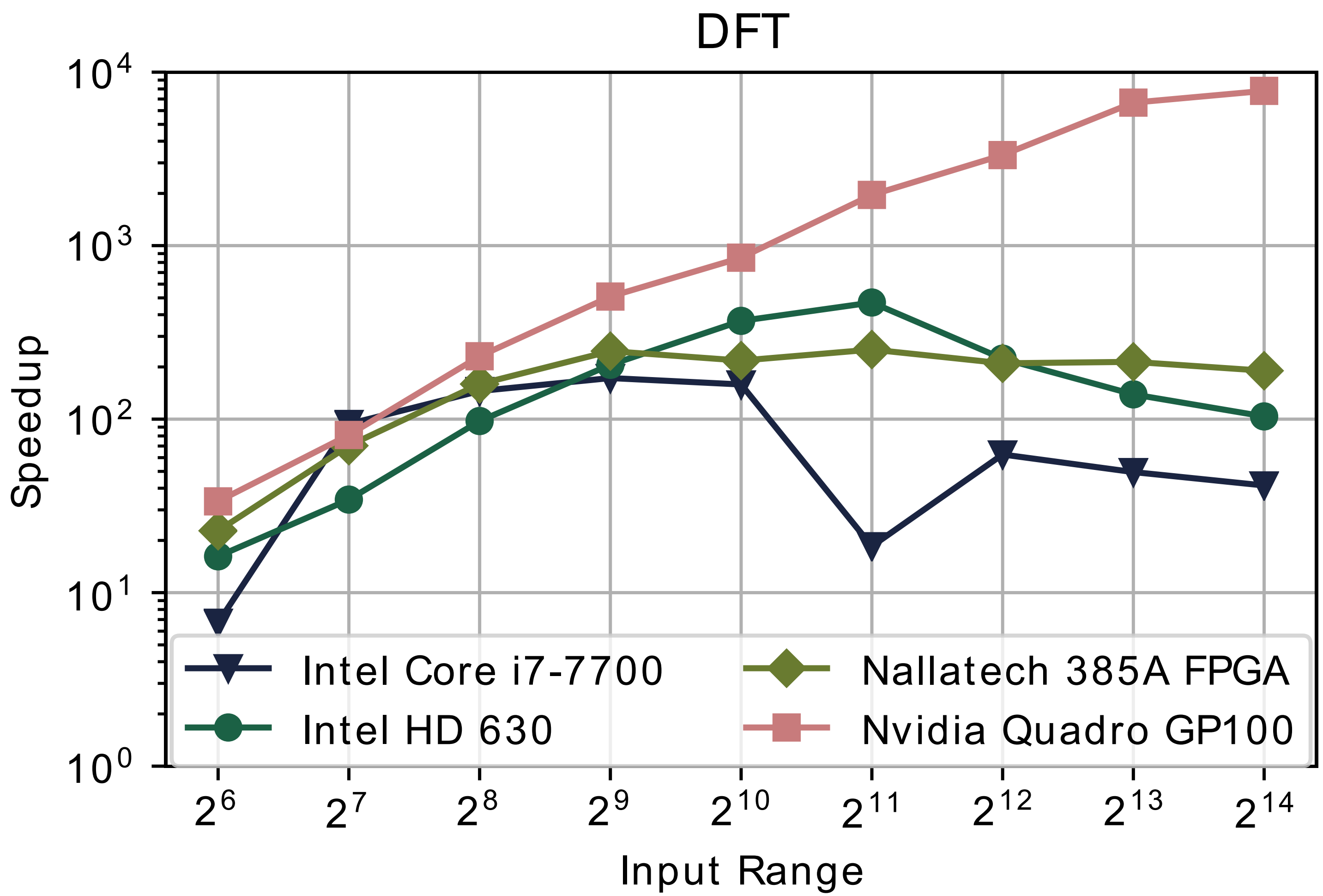
What is a GPU? Graphics Processing Unit

Source: NVIDIA docs

Contains a set of Stream Multiprocessor cores (SMx)
* Pascal arch. 60 SMx
* ~3500 CUDA cores

Users need to know:
A) Programming model (normally CUDA or OpenCL)
B) Details about the architecture are essential to achieve performance (e.g., memory tiers (local/shared memory, global memory, threads distribution).
  * Non sequential consistency, manual barriers, etc.

Still, why should we care about GPUs/FPGAs, etc?



Performance for each device against Java hotspot:
* Up to 4500x by using a GPU
* 240x by using an FPGA

Current Computer Systems

Hardware

FPGA

GPU

Integrated GPU

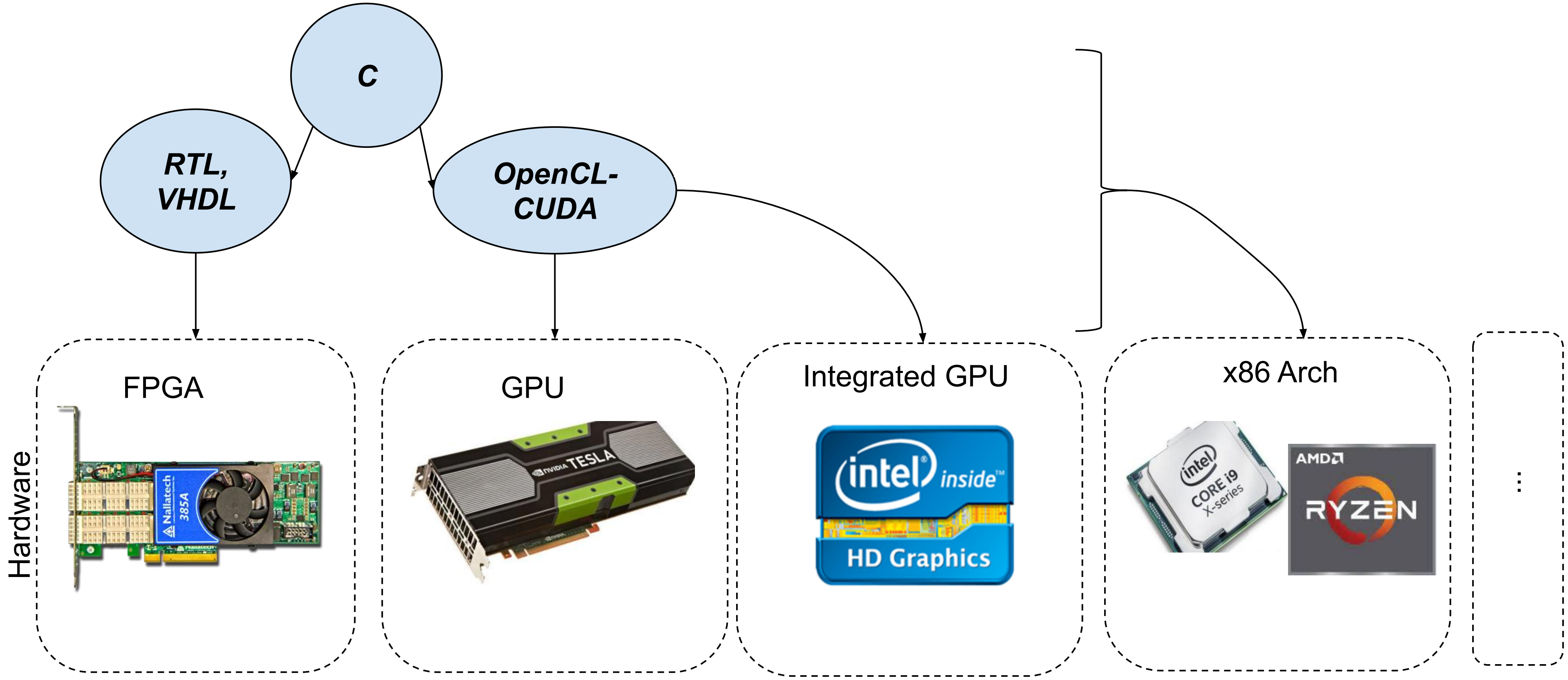
intel inside
HD Graphics

x86 Arch

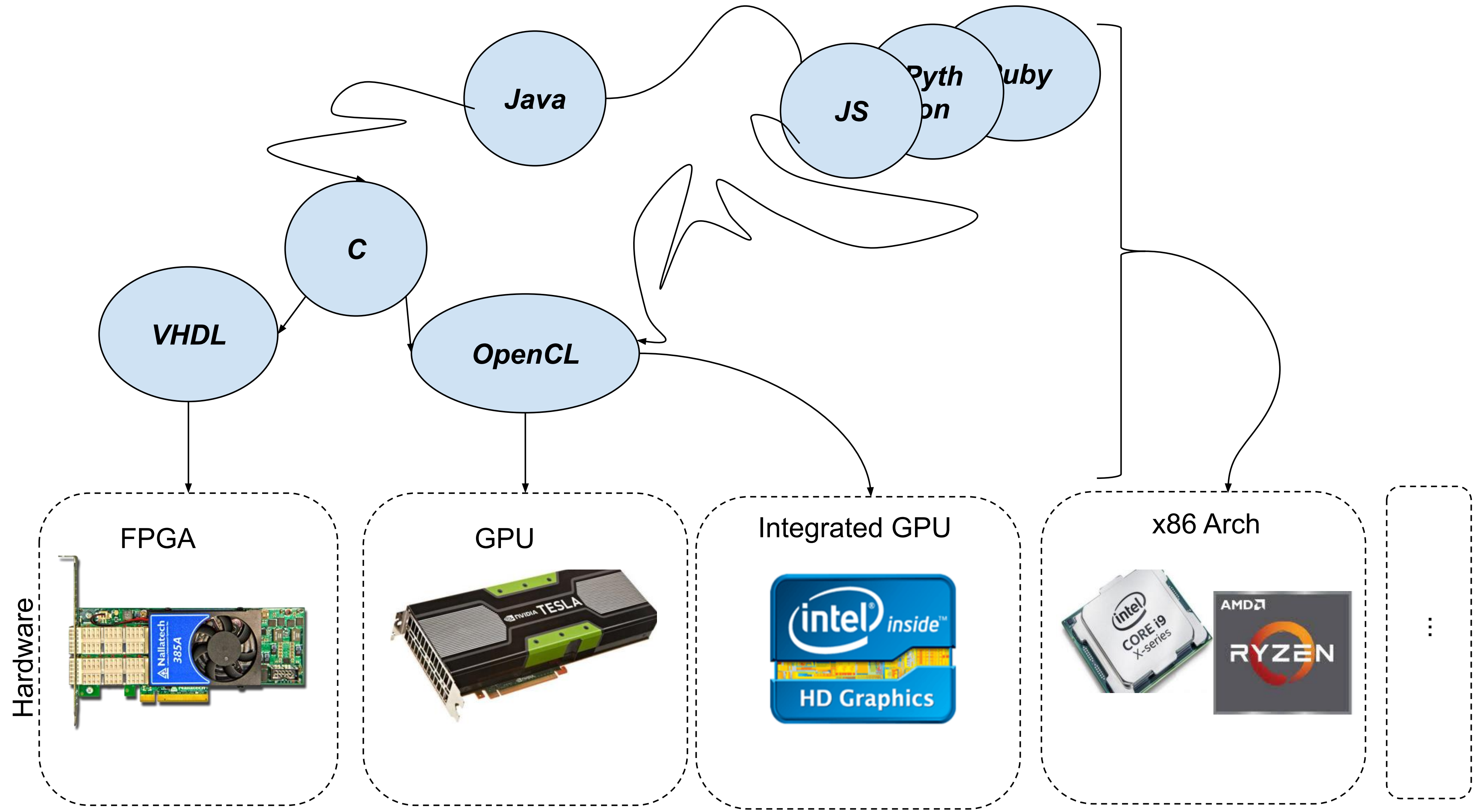
AMD
RYZEN

⋮

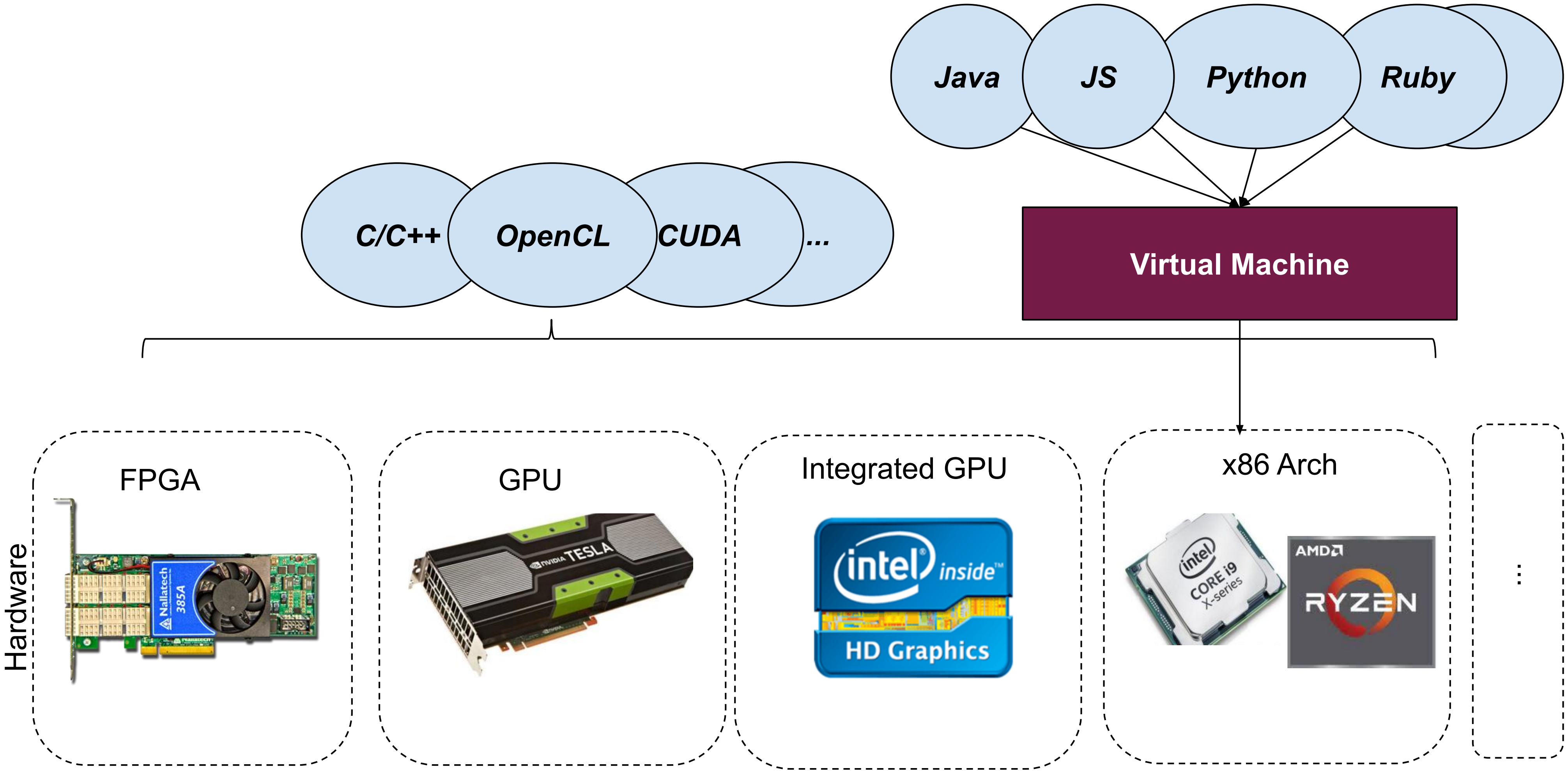
Current Computer Systems & Prog. Lang.



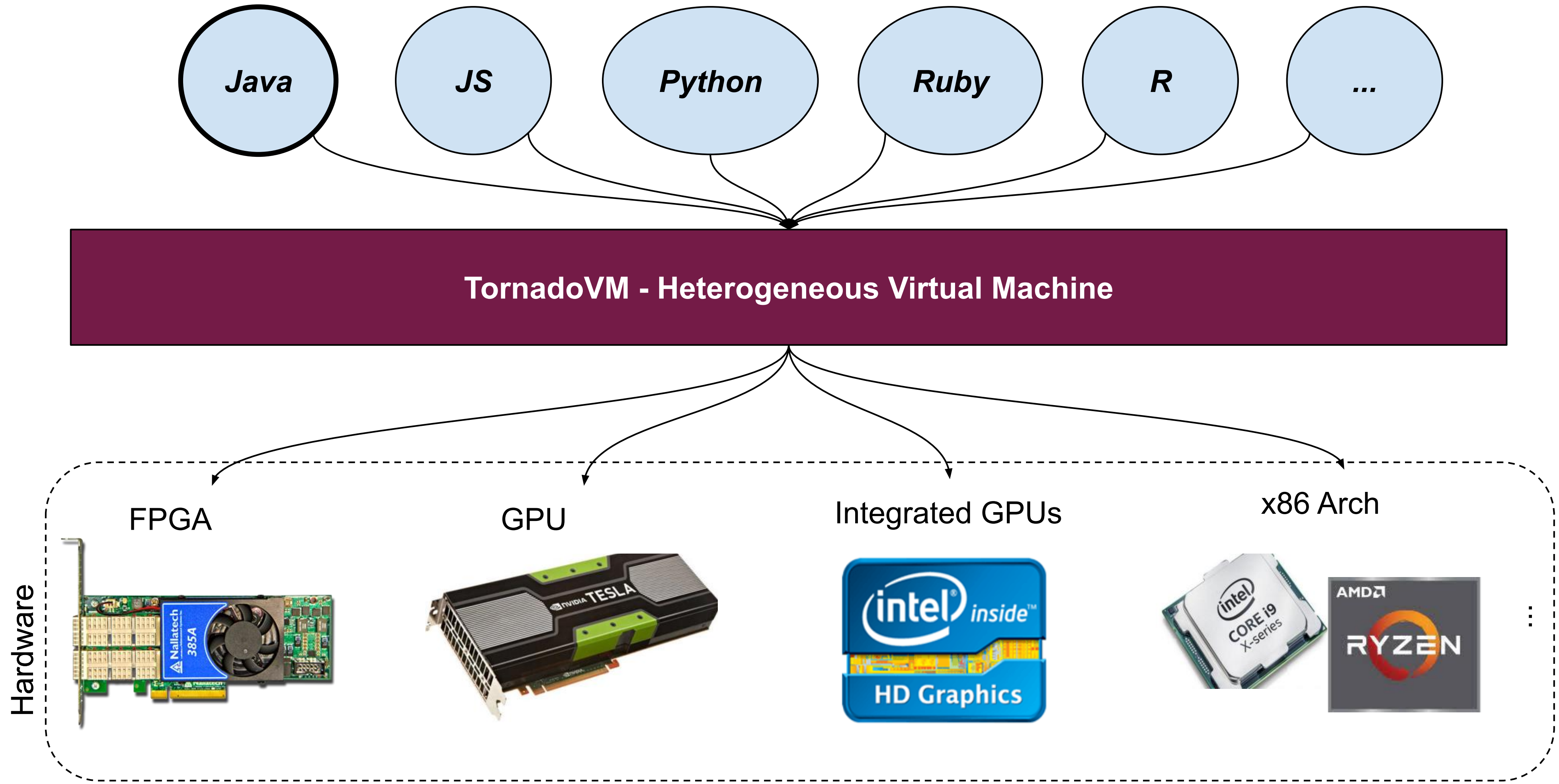
Current Computer Systems & Prog. Lang.



Current Computer Systems & Prog. Lang.



Ideal System for Managed Languages

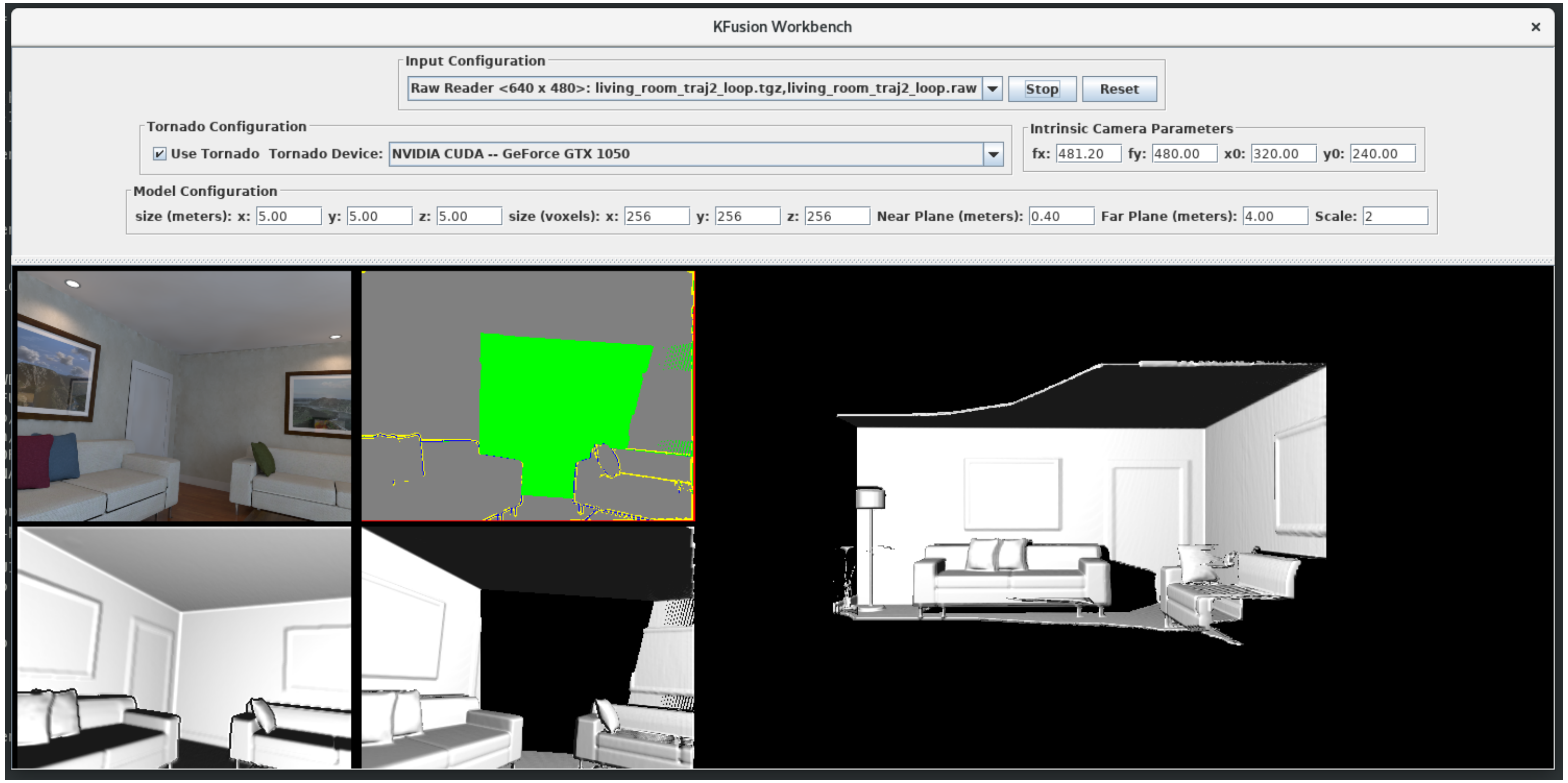


TornadoVM

The University of Manchester

MANCHESTER 1824
The University of Manchester

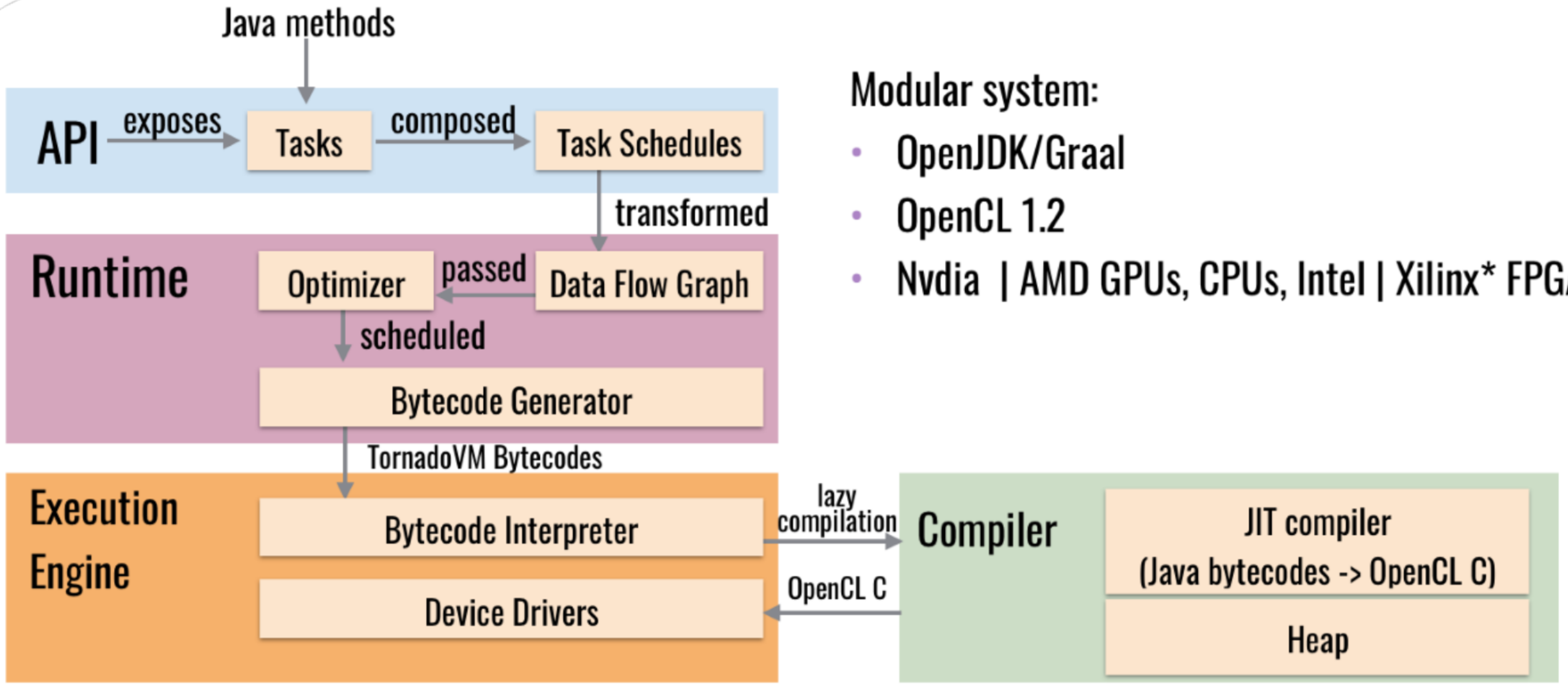
Demo: Kinect Fusion with TornadoVM



* Computer Vision Application
* ~7K LOC
* Thousands of OpenCL LOC generated.

https://github.com/beehive-lab/kfusion-tornadovm

TornadoVM Overview



Modular system:

- OpenJDK/Graal
- OpenCL 1.2
- Nvdia | AMD GPUs, CPUs, Intel | Xilinx* FPGAs

Tornado API – example

MANCHESTER 1824
The University of Manchester

```java
class Compute {
  public static void mxm(Matrix2DFloat A, Matrix2DFloat B,
                         Matrix2DFloat C, final int size) {
     for (int i = 0; i < size; i++) {
        for (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);
        }
     }
  }
}
```

Tornado API – example

```java
class Compute {
  public static void mxm(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);
         }
      }
  }
}
```

We add the parallel annotation as a hint for the compiler.

Tornado API – example

```java
class Compute {
  public static void mxm(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);
        }
     }
  }
}
```

```java
TaskSchedule ts = new TaskSchedule("s0");
ts.task("t0", Compute::mxm, matrixA, matrixB, matrixC, size)
   .streamOut(matrixC);
   .execute();
```

```
$ tornado Compute
```

Tornado API – Map-Reduce

The University of Manchester

```
class Compute {
  public static void map(float[] input, float[] output) {
     for (@Parallel int i = 0; i < size; i++) {
       … // map computation
     }
  }
  public static void reduce(@Reduce float[] data) {
     for (@Parallel int i = 0; i < size; i++) {
        data[0] += …
     }
  }
}
```

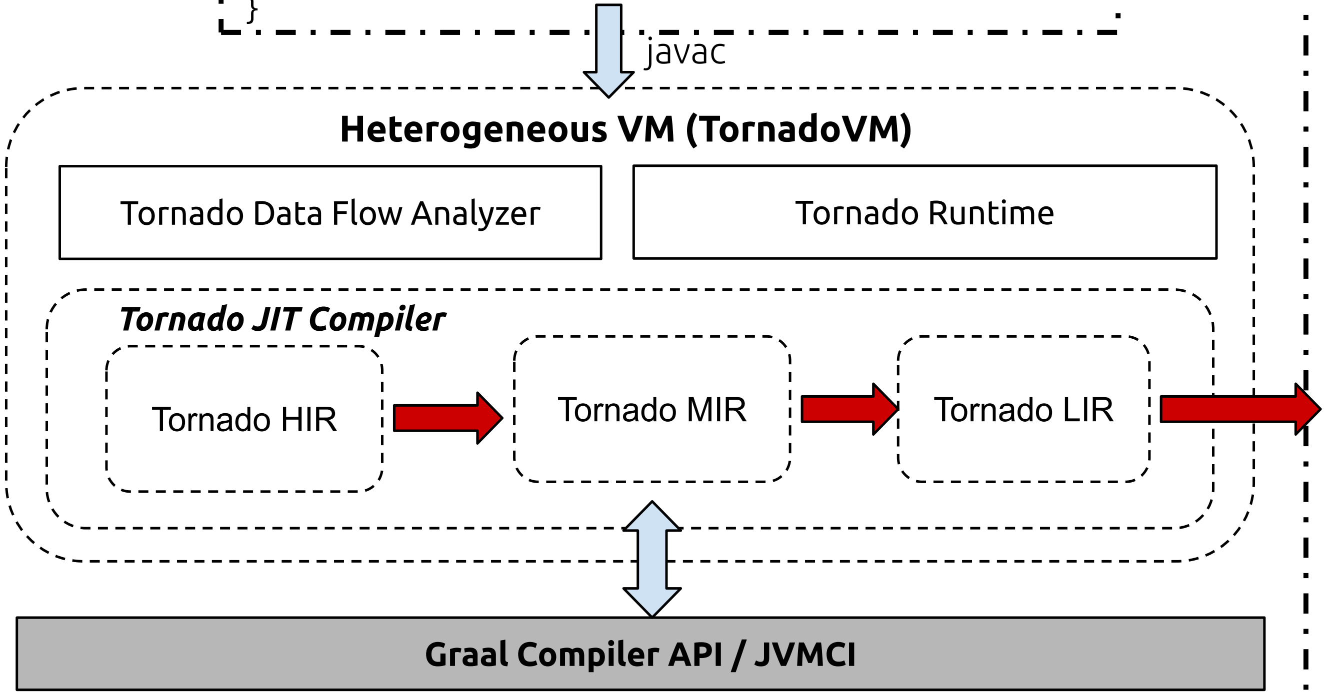
```
TaskSchedule ts = new TaskSchedule("MapReduce");
ts.streamIn(input)
    .task("map", Compute::map, input, output)
    .task("reduce", Compute::reduce, output)
  .streamOut(output);
  .execute();
```

github.com/beehive-lab/TornadoVM/tree/master/examples

TornadoVM Compiler & Runtime Overview

Input: Java Method

```java
public void compute(int[] a, int[] b) {
  for (@Parallel int i = 0; i < n; i++) {
    b[i] = a[i] + b[i];
  }
}
```



Output: OpenCL Kernel

```c
kernel void compute(global uchar *frame, ...) {
  // data types declaration
  ...
  ul_0  =  (ulong) frame[7];
  ul_1  =  (ulong) frame[8];
  i_2  =  get_global_id(0);
  i_3  =  i_2;

  for(;i_3 < 134217728;)  {
    l_4  =  (long) i_3;
    l_5  =  l_4 << 2;
    l_6  =  l_5 + 24L;
    ul_7  =  ul_0 + l_6;
    i_8  =  *((global int *) ul_7);
    ul_9  =  ul_1 + l_6;
    i_10  =  *((global int *) ul_9);
    i_11  =  i_8 * i_10;
    *((global int *) ul_9)  =  i_11;
    i_12  =  get_global_size(0);
    i_13  =  i_12 + i_3;
    i_3  =  i_13;
  }
}
```

OpenCL

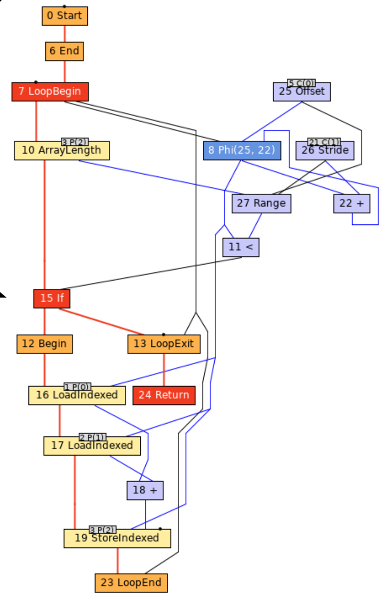
TornadoVM JIT Compiler Specializations

The University of Manchester

Input Java code

```
public static void add(int[] a, int[] b, int[] c)
   for (@Parallel int i = 0; i < c.length; i++)
      c[i] = a[i] + b[i];
   }
}
```

Compiler IR



GPU Specialization

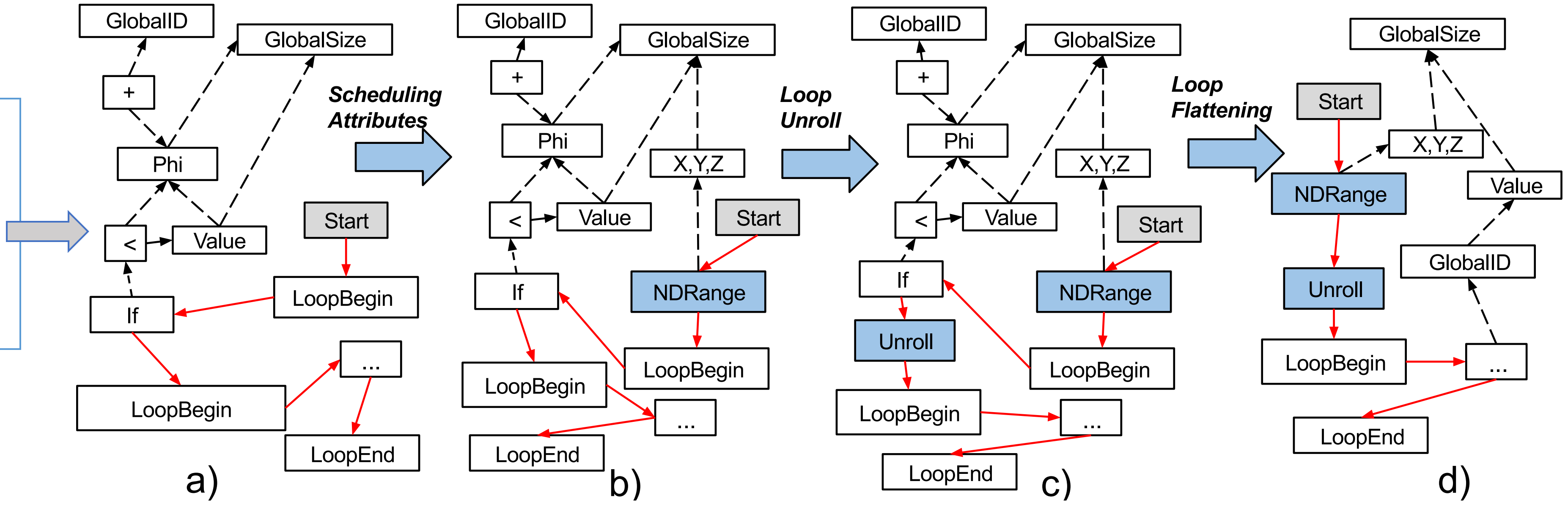

```
int idx = get_global_id(0);
int size = get_global_size(0);
for (int i = idx; i < c.length; i += size) {
  c[i] = a[i] + b[i];
}
```

CPU Specialization


```
int id = get_global_id(0);
int size = get_global_size(0);
int block_size = (size + inputSize - 1) / size;
int start = id * block_size;
int end = min(start + block_size, inputSize);
for (int i = start; i < end; i++) {
  c[i] = a[i] + b[i];
}
```

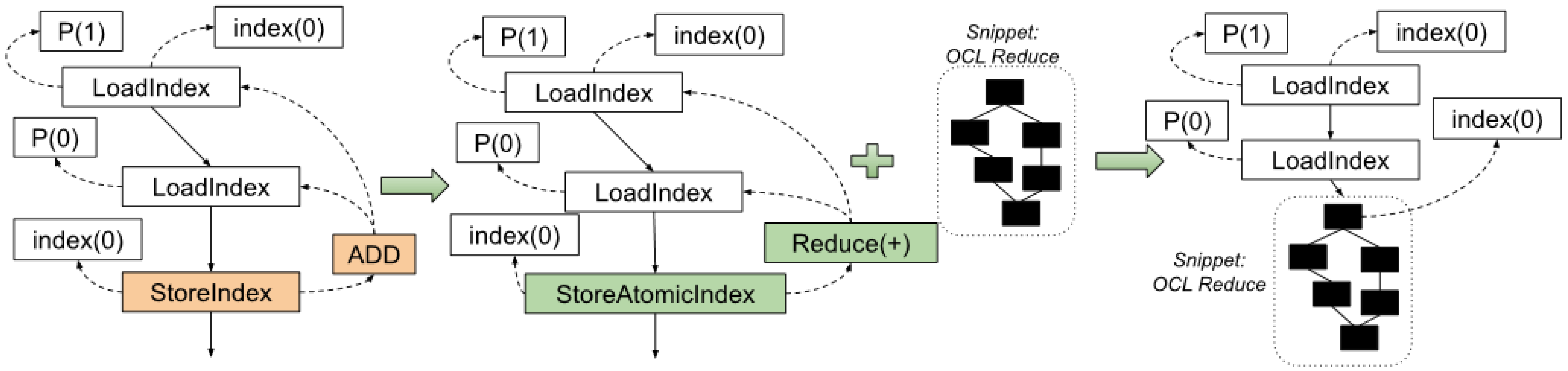
FPGA Specializations

```java
void compute(float[] input,
             float[] output) {
 for (@Parallel int i = 0; …) }
   for (@Parallel int j = 0; ...) {
     // Computation
   }
 }
}
```



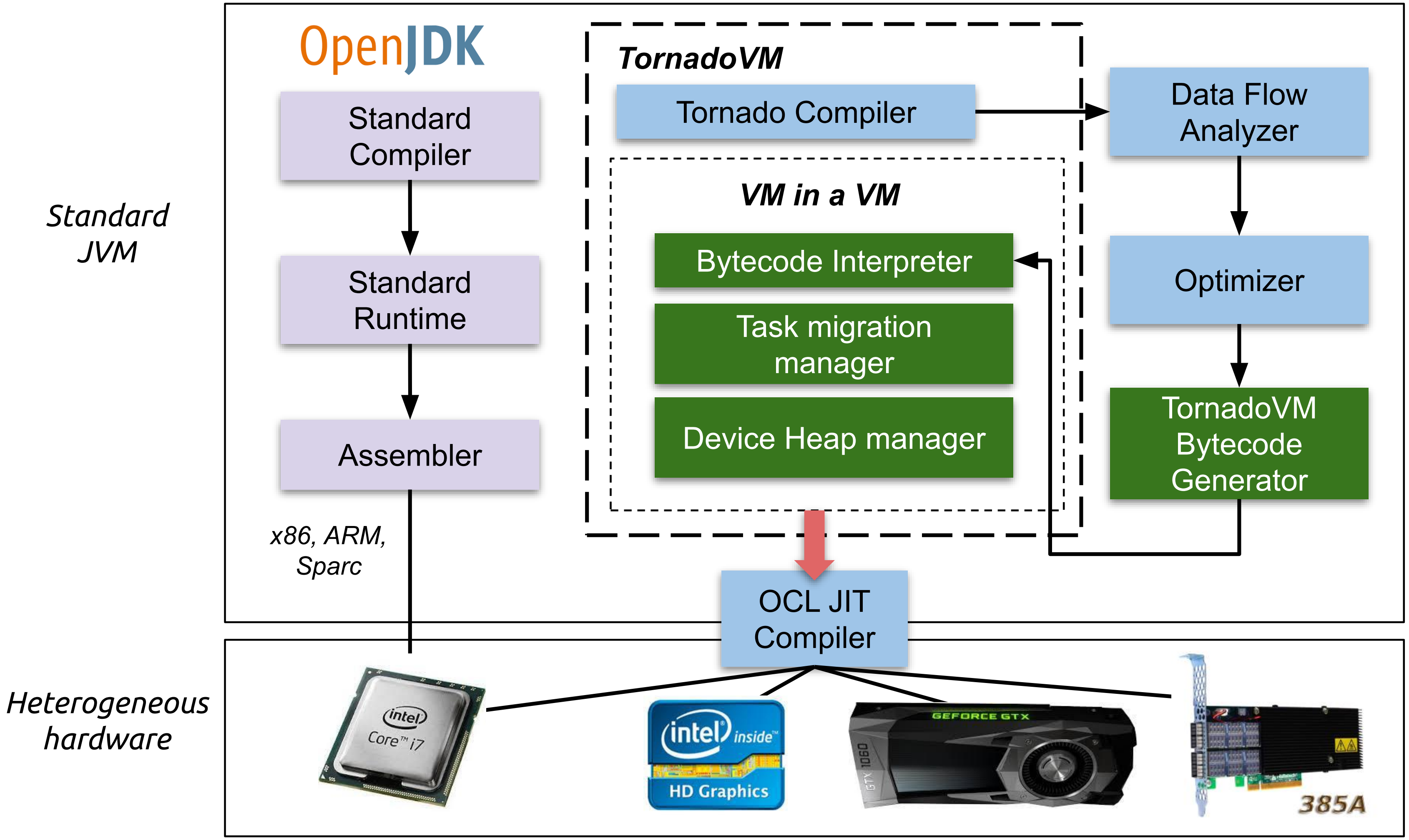
With Compiler specializations, TornadoVM performs from 5x to 240x against Java Hostpot for DFT!!!

Reduction Specializations via Snippets



With reduction-specializations we execute the code within 80% of the native (manual written code)

TornadoVM: VM in a VM

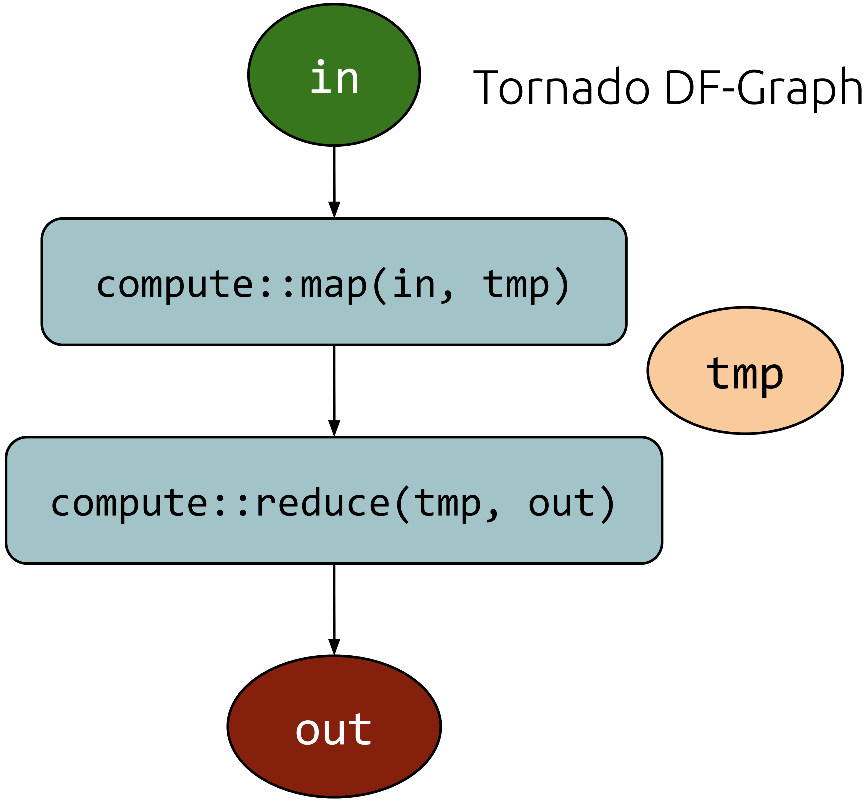


TornadoVM Bytecodes - Example

Input Java code

```java
public class Compute {
  public void map(float[] in, float[] out) {
    for (@Parallel int i = 0; i < n; i++) {
      out[i] = in[i] * in[i];
  }}
  public void reduce(float[]in,@Reduce float[]out) {
    for (@Parallel int i = 0; i < n; i++) {
      out[0] += in[i];
  }}
  public static void compute(float[] in, float[] out,
                             float[] tmp, Compute obj){
    TaskSchedule t0 = new TaskSchedule("s0")
      .task("t0", obj::map, in, tmp)
      .task("t1", obj::reduce, tmp, out)
      .streamOut(out)
      .execute();
  }}
```

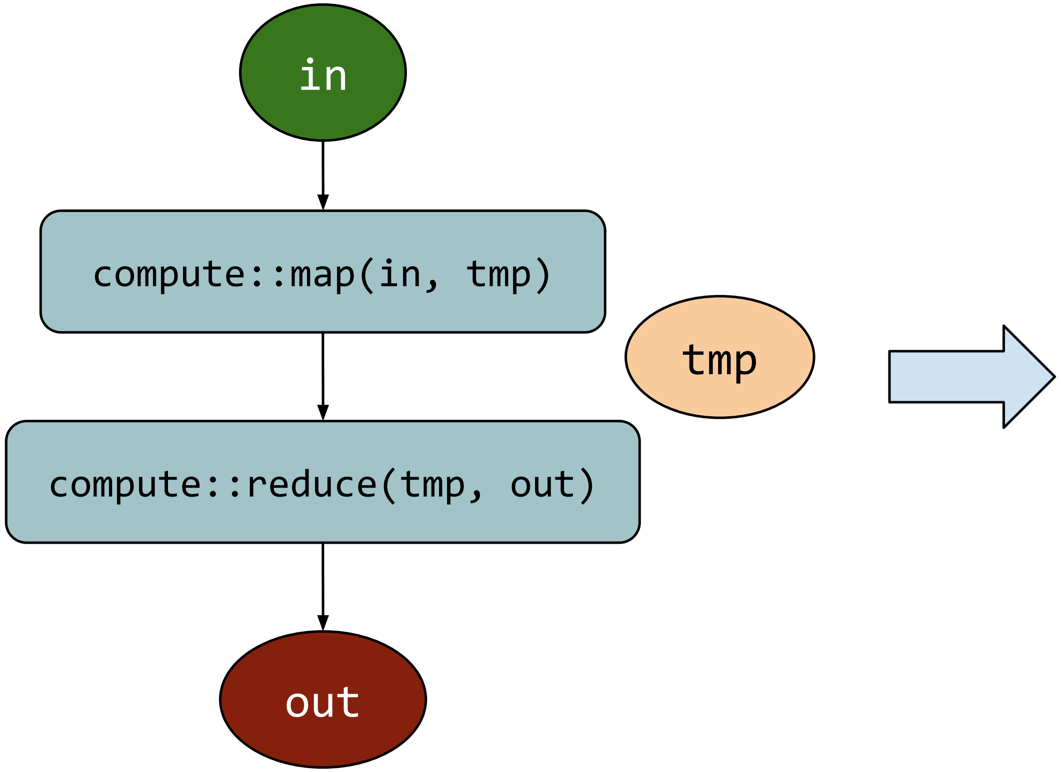
Tornado builds



Tornado DF-Graph

TornadoVM Bytecodes - Example

Tornado DF-Graph

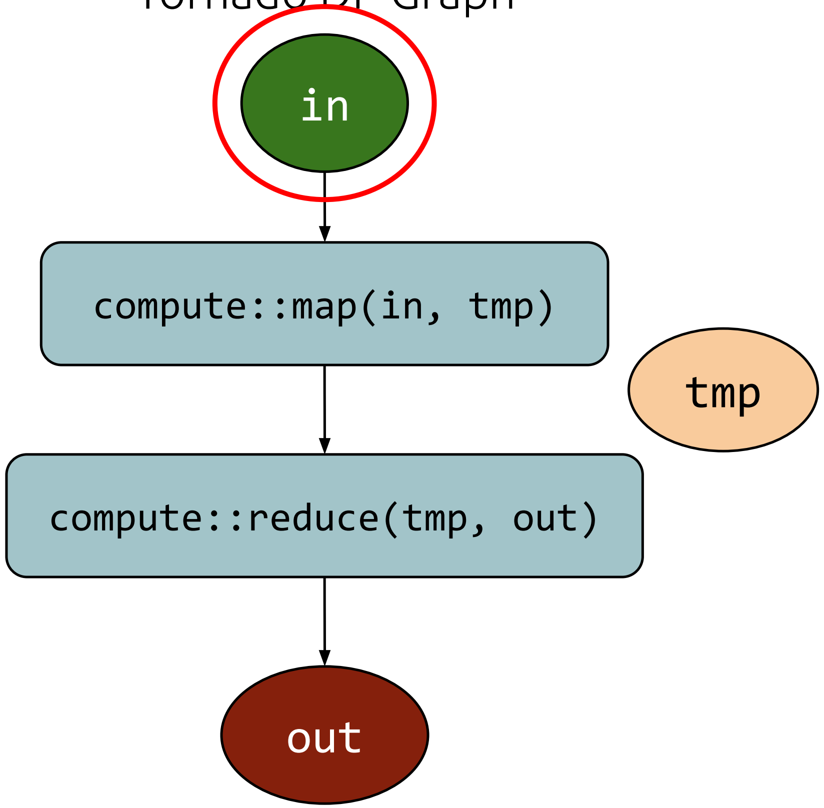


TornadoVM Bytecodes

```
BEGIN   <0>          // Starts a new context



END     <0>          // Ends context
```

TornadoVM Bytecodes - Example

Tornado DF-Graph

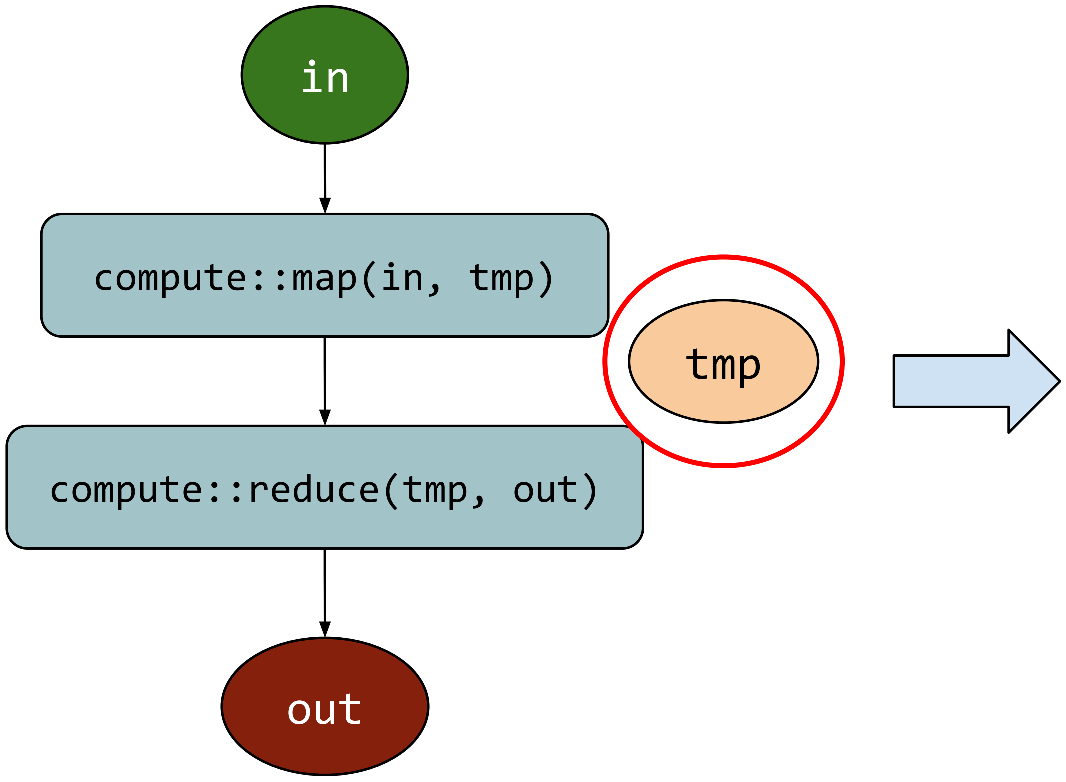


TornadoVM Bytecodes

```
BEGIN   <0>                 // Starts a new context
COPY_IN <0, bi1, in>        // Allocates and copies <in>



END     <0>                 // Ends context
```

TornadoVM Bytecodes - Example

Tornado DF-Graph

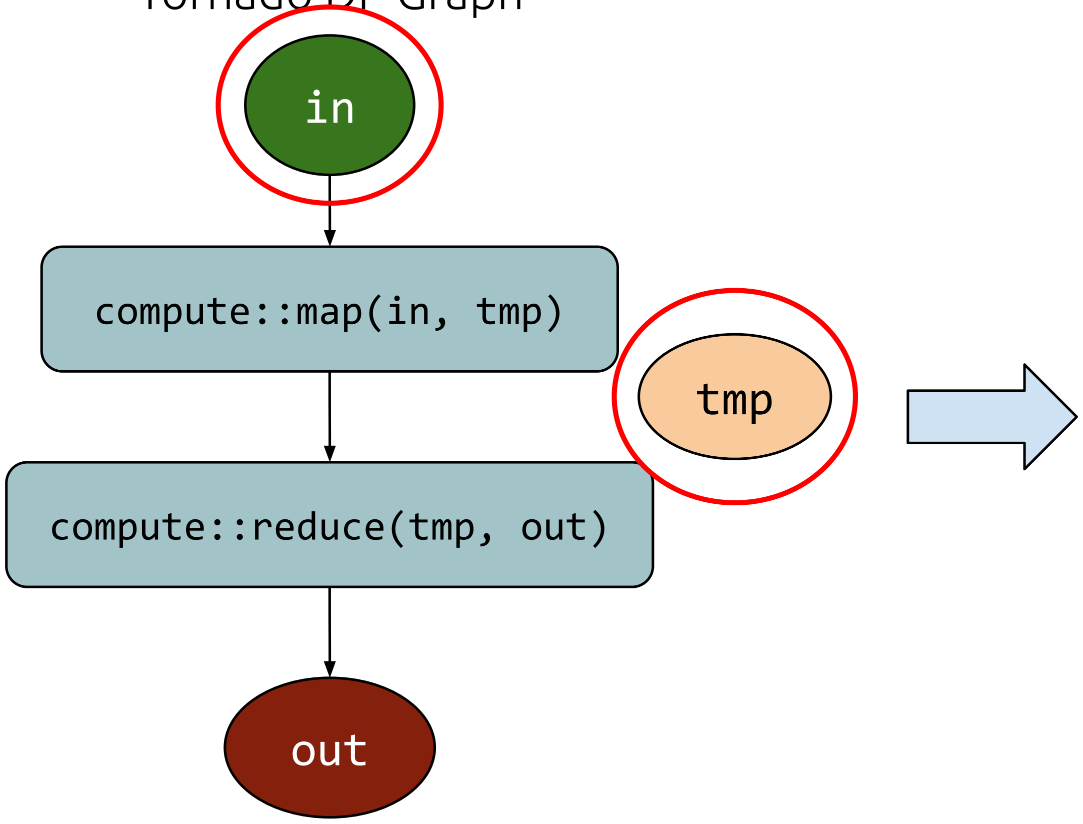


TornadoVM Bytecodes

```
BEGIN   <0>              // Starts a new context
COPY_IN <0, bi1, in>     // Allocates and copies <in>
ALLOC   <0, bi2, tmp>    // Allocates <tmp> on device

END     <0>              // Ends context
```

TornadoVM Bytecodes - Example

Tornado DF-Graph

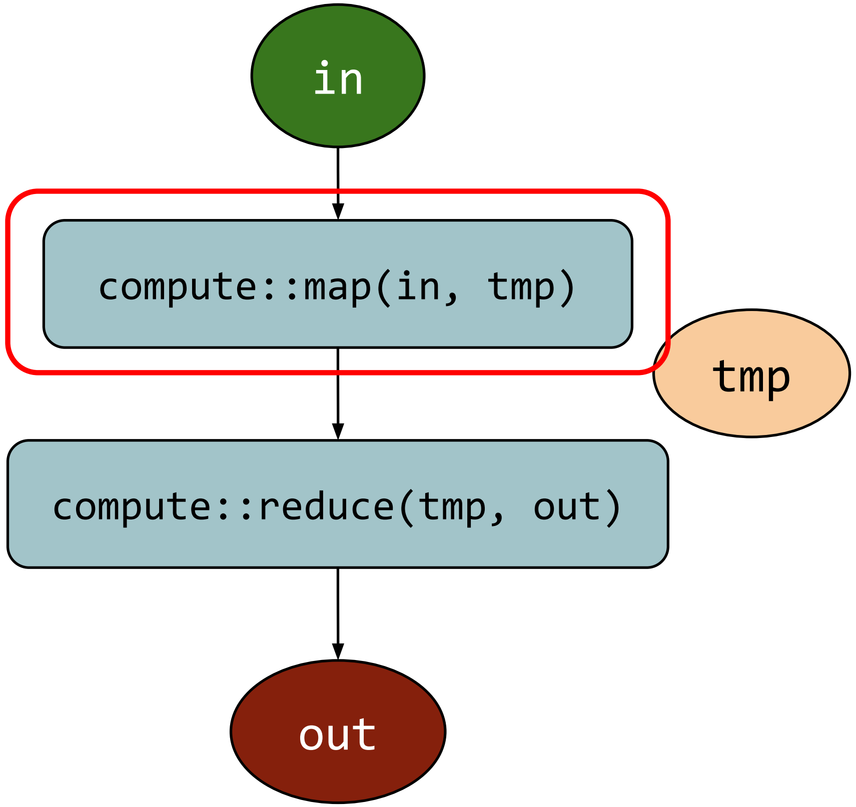


TornadoVM Bytecodes

```
BEGIN   <0>              // Starts a new context
COPY_IN <0, bi1, in>     // Allocates and copies <in>
ALLOC   <0, bi2, tmp>    // Allocates <tmp> on device
ADD_DEP <0, bi1, bi2>    // Waits for copy and alloc




END     <0>              // Ends context
```

TornadoVM Bytecodes - Example

Tornado DF-Graph

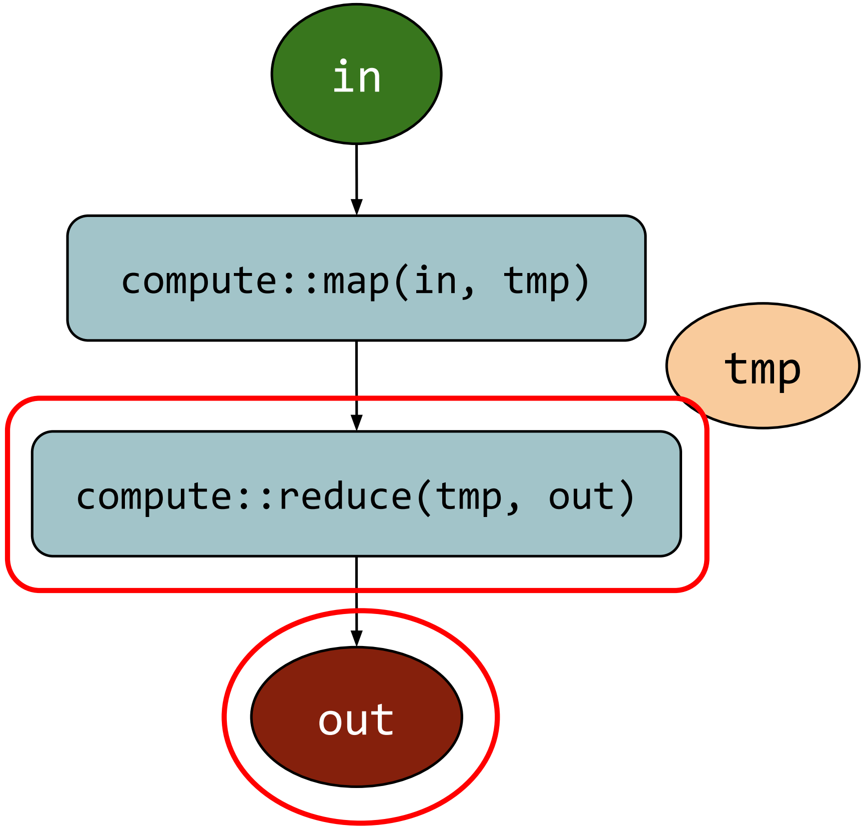


TornadoVM Bytecodes

```
BEGIN    <0>                  // Starts a new context
COPY_IN  <0, bi1, in>         // Allocates and copies <in>
ALLOC    <0, bi2, tmp>        // Allocates <tmp> on device
ADD_DEP  <0, bi1, bi2>        // Waits for copy and alloc
LAUNCH   <0, bi3, @map, in, tmp> // Runs map



END      <0>                  // Ends context
```

TornadoVM Bytecodes - Example

Tornado DF-Graph

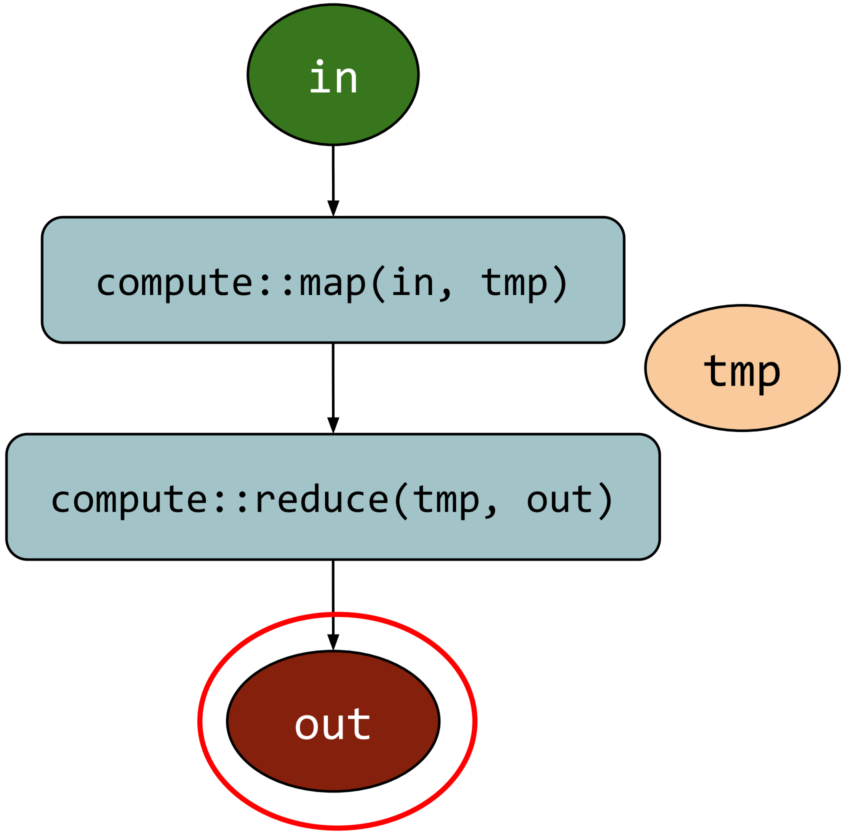


TornadoVM Bytecodes

```
BEGIN   <0>                 // Starts a new context
COPY_IN <0, bi1, in>        // Allocates and copies <in>
ALLOC   <0, bi2, tmp>       // Allocates <tmp> on device
ADD_DEP <0, bi1, bi2>       // Waits for copy and alloc
LAUNCH  <0, bi3, @map, in, tmp> // Runs map
ALLOC   <0, bi4, out>       // Allocates <out> on device
ADD_DEP <0, bi3, bi4>       // Waits for alloc and map
LAUNCH  <0, bi5, @reduce, tmp, out> // Runs reduce


END     <0>                 // Ends context
```

TornadoVM Bytecodes - Example

Tornado DF-Graph



TornadoVM Bytecodes

```
BEGIN   <0>                 // Starts a new context
COPY_IN <0, bi1, in>        // Allocates and copies <in>
ALLOC   <0, bi2, tmp>       // Allocates <tmp> on device
ADD_DEP <0, bi1, bi2>       // Waits for copy and alloc
LAUNCH  <0, bi3, @map, in, tmp> // Runs map
ALLOC   <0, bi4, out>       // Allocates <out> on device
ADD_DEP <0, bi3, bi4>       // Waits for alloc and map
LAUNCH  <0, bi5, @reduce, tmp, out> // Runs reduce
ADD_DEP <0, bi5>            // Wait for reduce
COPY_OUT_BLOCK <0, bi6, out> // Copies <out> back
END     <0>                 // Ends context
```

Batch Processing: 16GB into 1GB GPU

Input Java user-code

```java
class Compute {
  public static void add(double[] a, double[] b,
  double[] c) {
   for (@Parallel int i = 0; i < c.length; i++)
      c[i] = a[i] + b[i];
   }
  }
}
```

```java
// 16GB data
double[] a = new double[2000000000];
double[] b = new double[2000000000];
double[] c = new double[2000000000];
TaskSchedule ts = new TaskSchedule("s0");

ts.batch("300MB")
  .task(Compute::add, a, b, c)
  .streamOut(c)
  .execute();
```

Tornado VM

```
vm: BEGIN
vm: COPY_IN bytes=300000000, offset=0
vm: COPY_IN bytes=300000000, offset=0
vm: ALLOCATE bytes=300000000
vm: LAUNCH s0.t0 threads=37500000, offset=0
vm: STREAM_OUT bytes=300000000, offset=0
vm: COPY_IN bytes=300000000, offset=300000000
vm: COPY_IN bytes=300000000, offset=300000000
vm: ALLOCATE bytes=300000000
vm: LAUNCH task s0.t0 threads=37500000, offset=300000000
vm: STREAM_OUT bytes=300000000, offset=300000000
vm: ...
vm: ...
vm: STREAM_OUT_BLOCKING bytes=100000000, offset=1500000000
vm: END
```

Easy to orchestrate heterogeneous execution

Batch Processing: 16GB into 1GB GPU

MANCHESTER 1824
The University of Manchester

Input Java user-code

```
class Compute {
  public static void add(double[] a, double[] b,
  double[] c) {
   for (@Parallel int i = 0; i < c.length; i++)
      c[i] = a[i] + b[i];
   }
  }
}
```

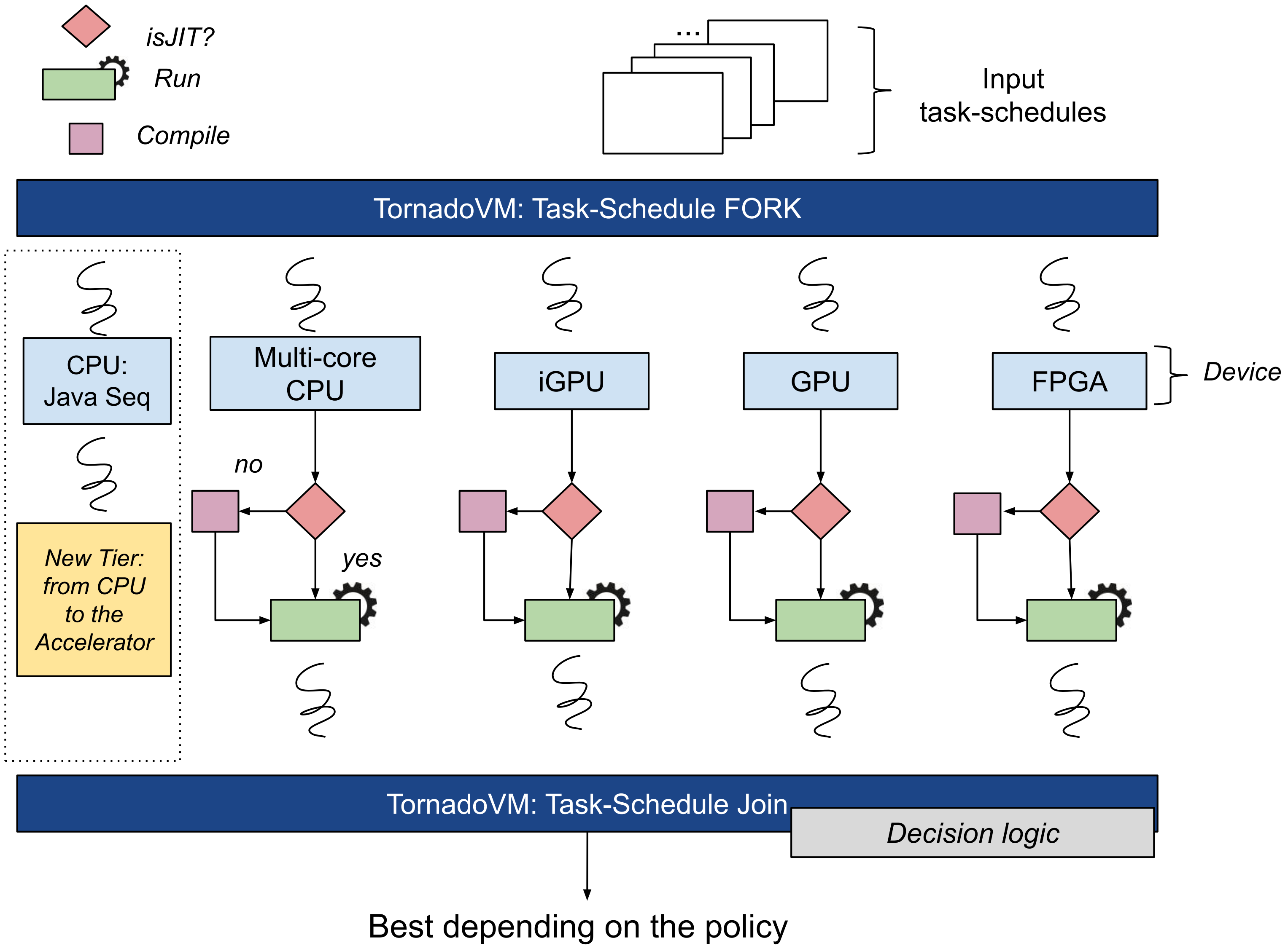
```
// 16GB data
double[] a = new double[2000000000];
double[] b = new double[2000000000];
double[] c = new double[2000000000];
TaskSchedule ts = new TaskSchedule(“s0”);

ts.batch(“300MB”)
  .task(Compute::add, a, b, c)
  .streamOut(c)
  .execute();
```

Tornado VM

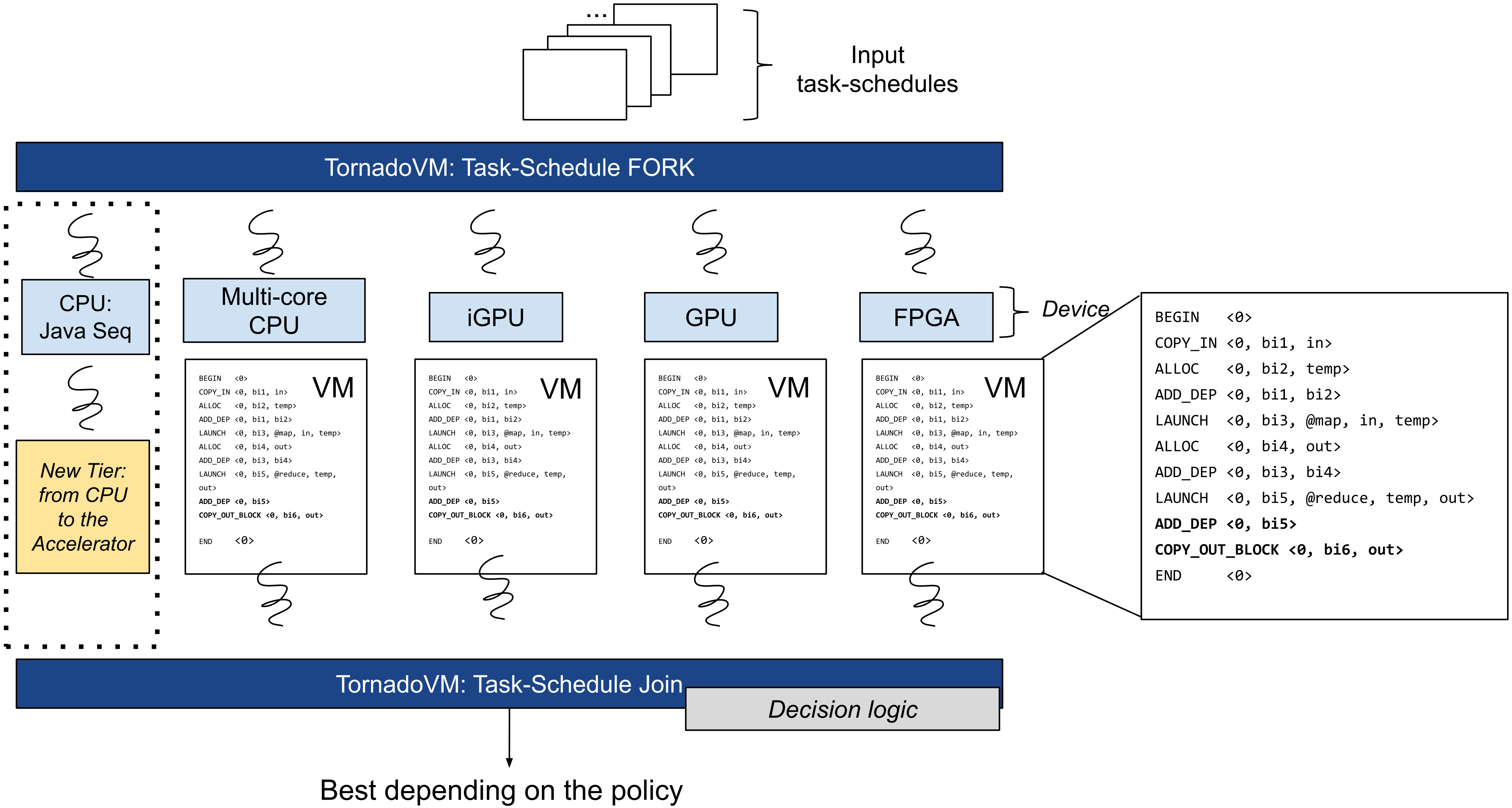
```
vm: BEGIN
vm: COPY_IN bytes=300000000, offset=0
vm: COPY_IN bytes=300000000, offset=0
vm: ALLOCATE bytes=300000000
vm: LAUNCH s0.t0 threads=37500000, offset=0
vm: STREAM_OUT bytes=300000000, offset=0
vm: COPY_IN bytes=300000000, offset=300000000
vm: COPY_IN bytes=300000000, offset=300000000
vm: ALLOCATE bytes=300000000
vm: LAUNCH task s0.t0 threads=37500000, offset=300000000
vm: STREAM_OUT bytes=300000000, offset=300000000
vm: …
vm: …
vm: STREAM_OUT_BLOCKING bytes=100000000, offset=1500000000
vm: END
```

Easy to orchestrate heterogeneous execution

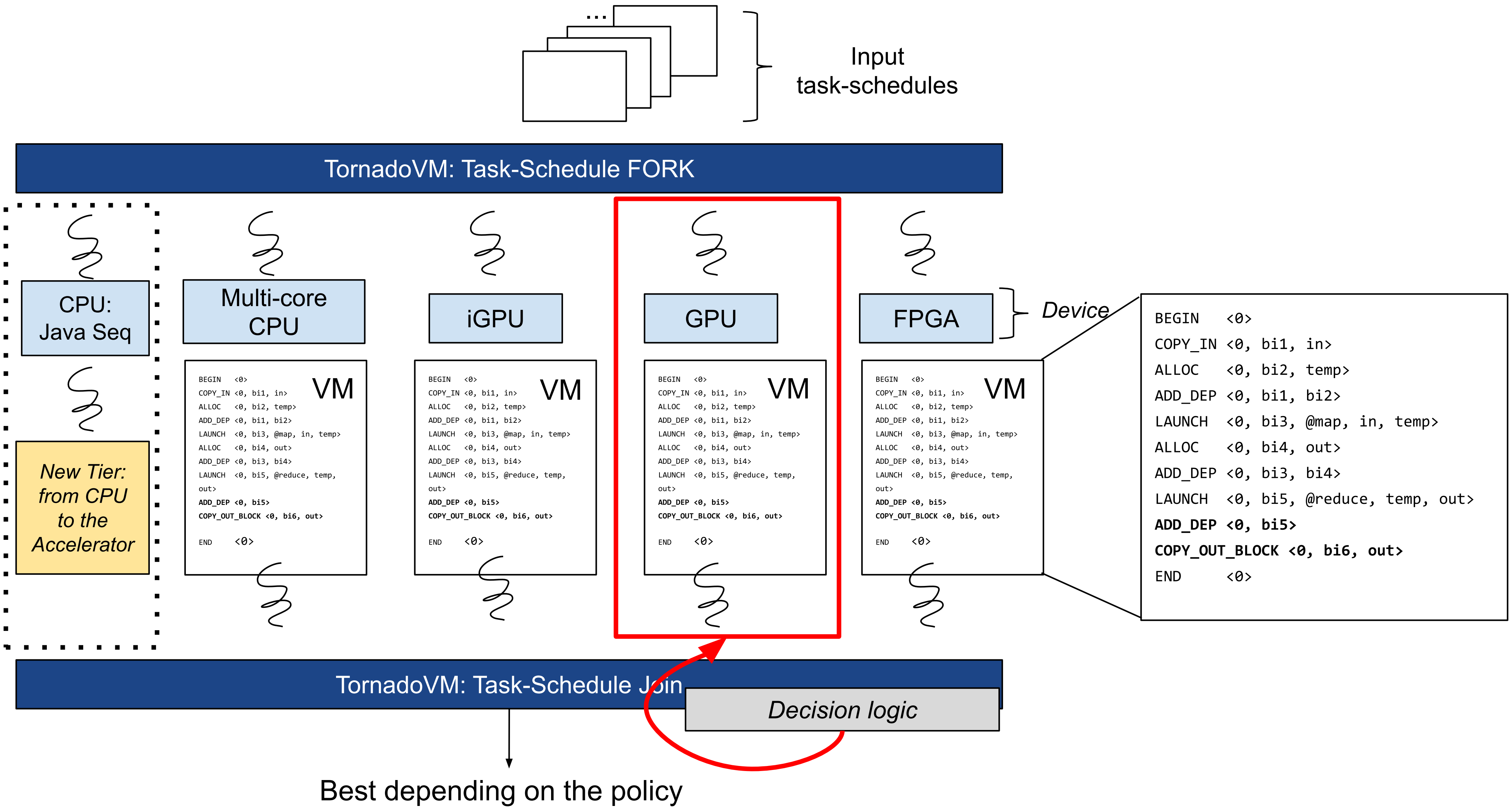
Dynamic Reconfiguration



Dynamic Reconfiguration



Dynamic Reconfiguration



How is the decision made?

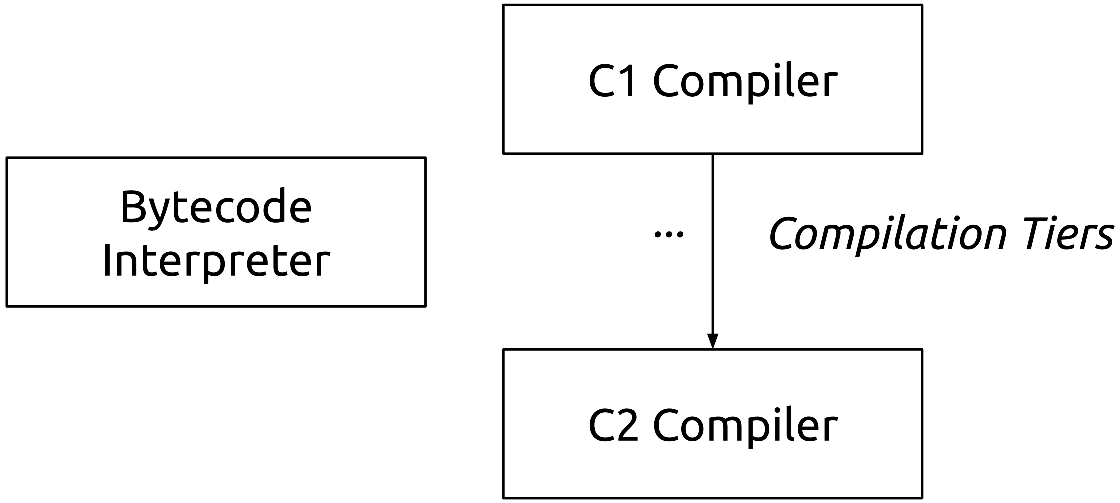
- End-to-end: including JIT compilation time
- Peak Performance: without JIT and after warming-up
- Latency: does not wait for all threads to finish

```
// END TO END PERFORMANCE
ts.task(Compute::add, a, b, c)
  .streamOut(c)
  .execute(Profiler.END2END);
```

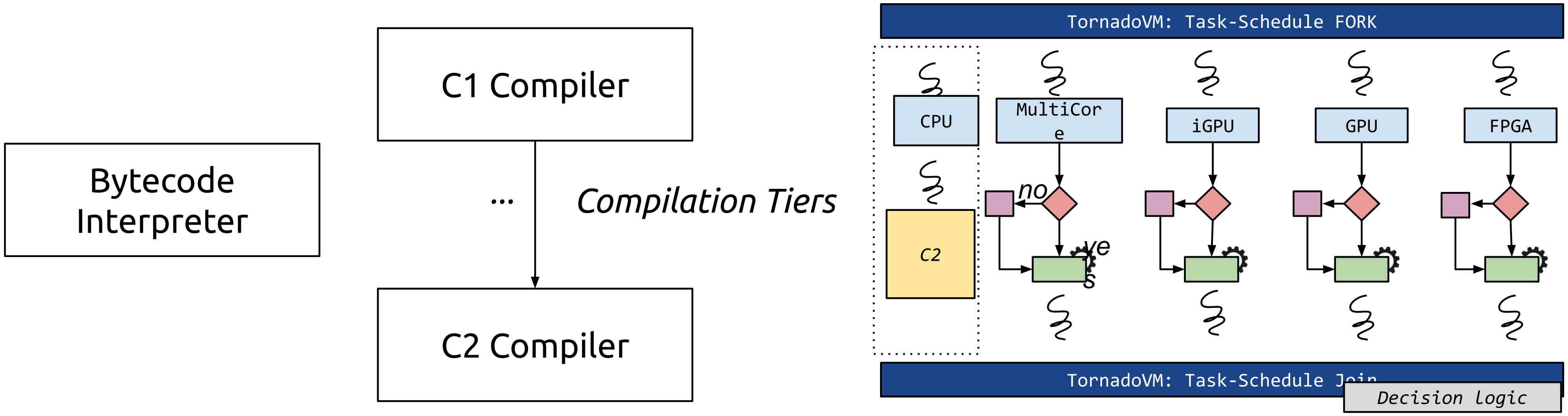
```
// PEAK PERFORMANCE
ts.task(Compute::add, a, b, c)
  .streamOut(c)
  .execute(Profiler.PERFORMANCE);
```

```
// Latency
ts.task(Compute::add, a, b, c)
  .streamOut(c)
  .execute(Profiler.LATENCY);
```

New compilation tier for Heterogeneous Systems



New compilation tier for Heterogeneous Systems



E.g., From C2 -> Multi-core -> GPU

Related Work

Related Work (in the Java context)

- Sumatra
  - Java Stream 8 API to target HSAIL
  - No FPGA Support
  - No Dynamic Application Reconfiguration

- Aparapi
  - Kernels follow OpenCL semantics but in Java (e.g., thread global-id is exposed)
  - AFAIK, target only GPUs/CPUs
  - No Dynamic Application Reconfiguration

- IBM GPU J9
  - Similar to Sumatra accelerating parallel Streams -> Targets only NVIDIA GPUs
  - No Dynamic Application Reconfiguration

JEP - 8047074 http://openjdk.java.net/jeps/8047074

MANCHESTER 1824
The University of Manchester

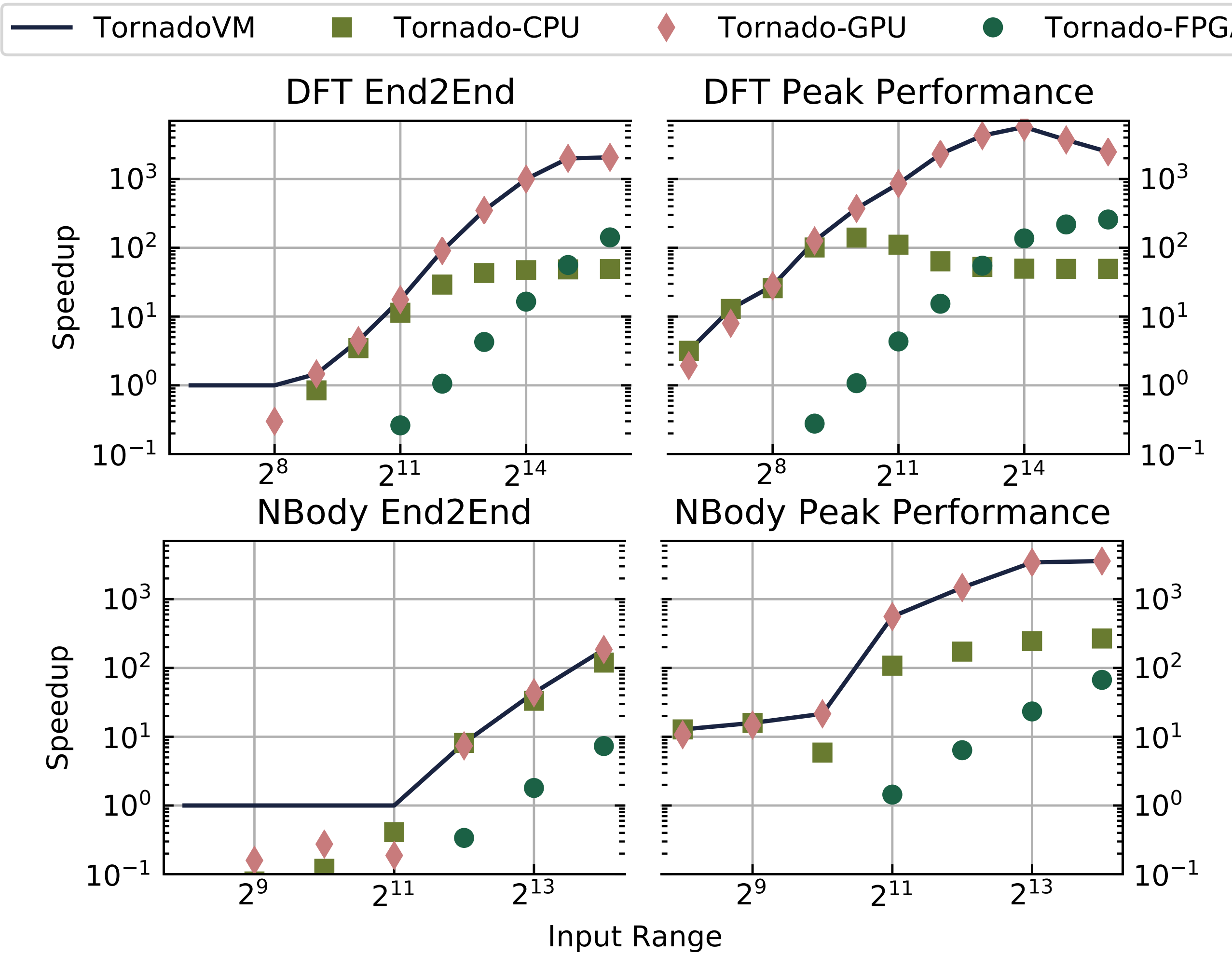
GOALS	Implemented in Tornado?
No syntactic changes to Java 8 parallel stream API	(Own API)
Autodetection of hardware and software stack	✓
Heuristic to decide when to offload to GPU gives perf gains	✓
Performance improvement for embarrassingly parallel workloads	✓
Code accuracy has the same (non-) guarantees you can get with multi core parallelism	✓
Code will always run with graceful fallback to normal CPU execution if offload fails	In progress!
Will not expose any additional security risks	Under research
Offloaded code will maintain Java memory model correctness (find JSR)	Under formal specification (several trade-offs have to be considered)
Where possible enable JVM languages to be offloaded	Plan to integrate with Truffle. E.g., FastR-GPU: https://bitbucket.org/juanfumero/fastr-gpu/src/default/

Additional features

Additional Features (not included JEP 8047074)	Implemented in Tornado?
Include GPUs, integrated GPU, FPGAs, multi-cores CPUs	✓
Live-task migration between devices	✓
Code specialization for each accelerator	✓
Potentially accelerate existing Java libraries (Lucene)	✓
Automatic use of tier-memory on the device (e.g., local memory)	< In progress>
Virtual Shared Memory (OpenCL 2.0)	< In progress>

Ok, cool! What about performance?

Performance

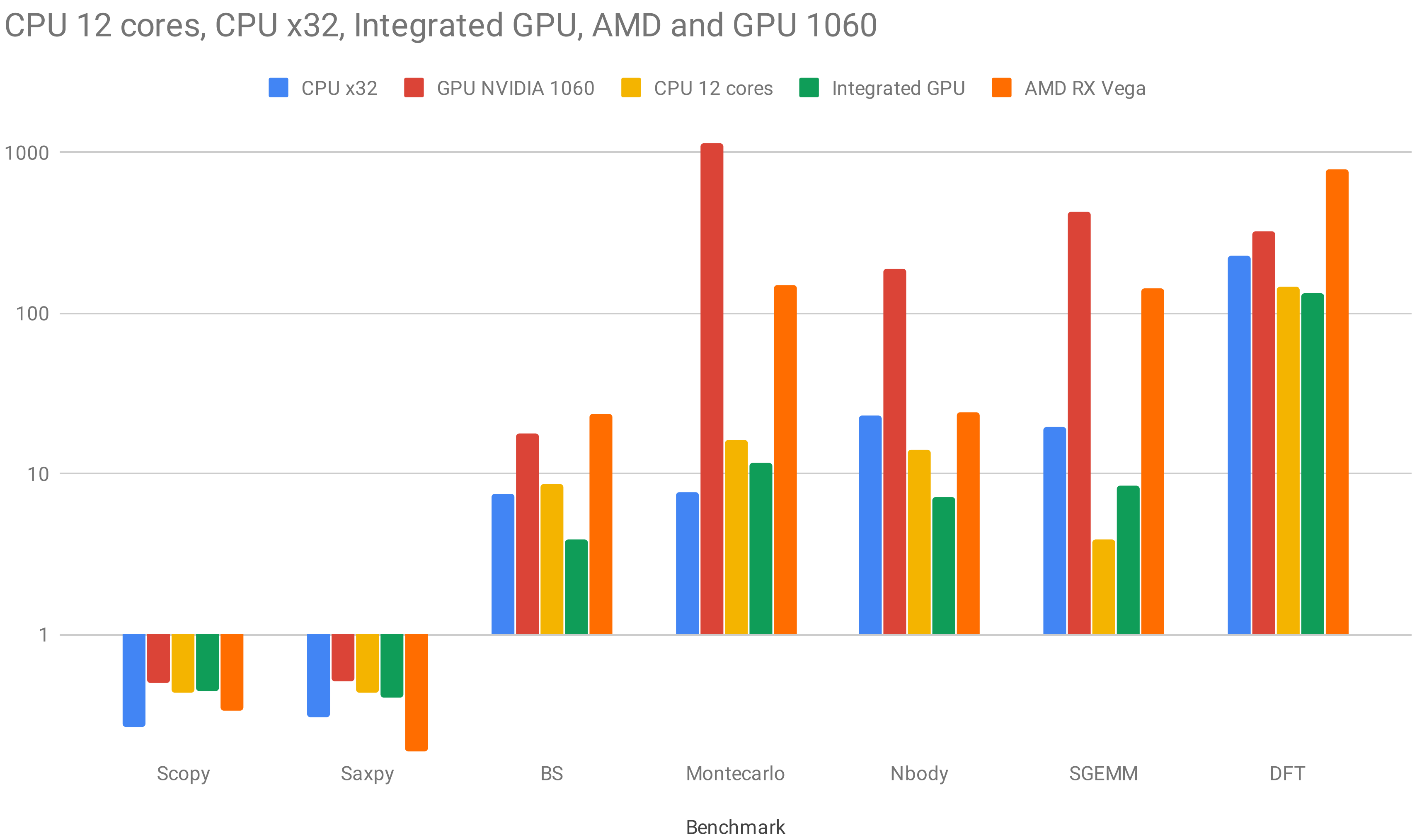


* TornadoVM performs up to 7.7x over the best device (statically).
* Up to >4500x over Java sequential

- *NVIDIA GTX 1060*
- *Intel FPGA Nallatech 385a*
- *Intel Core i7-7700K*

Performance on GPUs, iGPUs, and CPUs

The University of Manchester



Limitations
&
Future Work

Limitations

- No object support (except for a few cases)
- No recursion
- No dynamic memory allocation
- No support for exceptions (*)

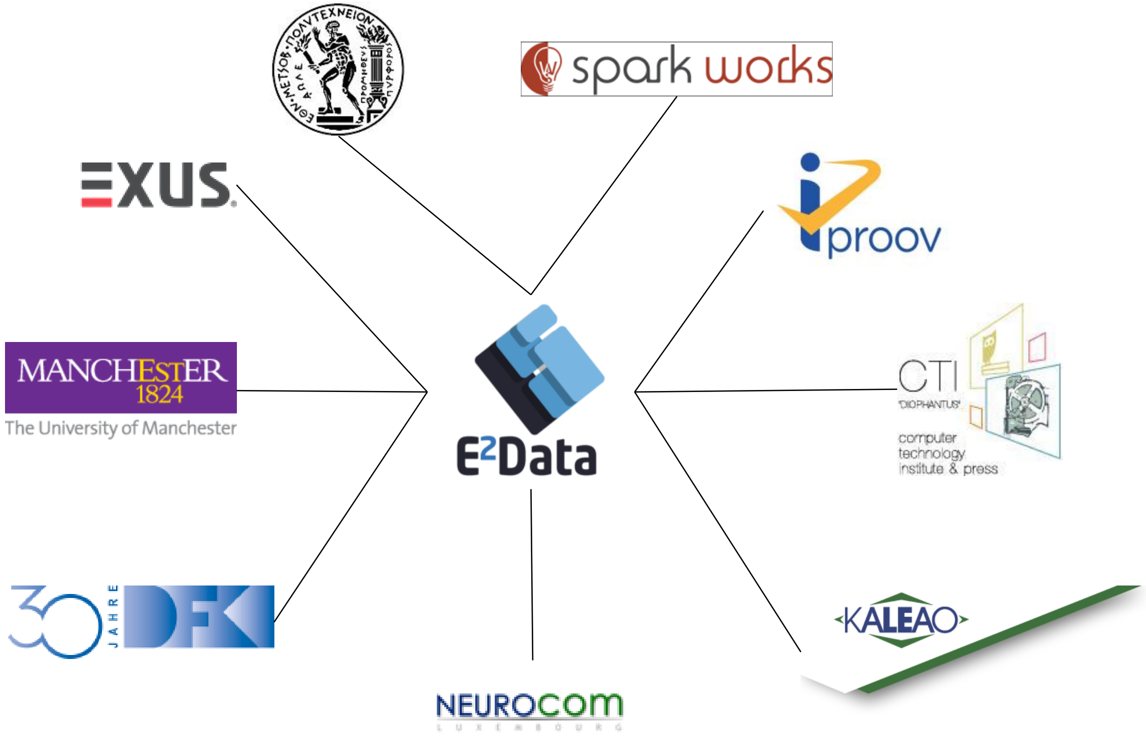
Future Work

- GPU/FPGA full capabilities
  - Exploitation of Tier-memories such as local memory
- Policies for energy efficiency
- Multi-device within a task-schedule
- More parallel skeletons (scan, filter, …)

The University of Manchester

Current Applicability of TornadoVM

EU H2020 E2Data Project

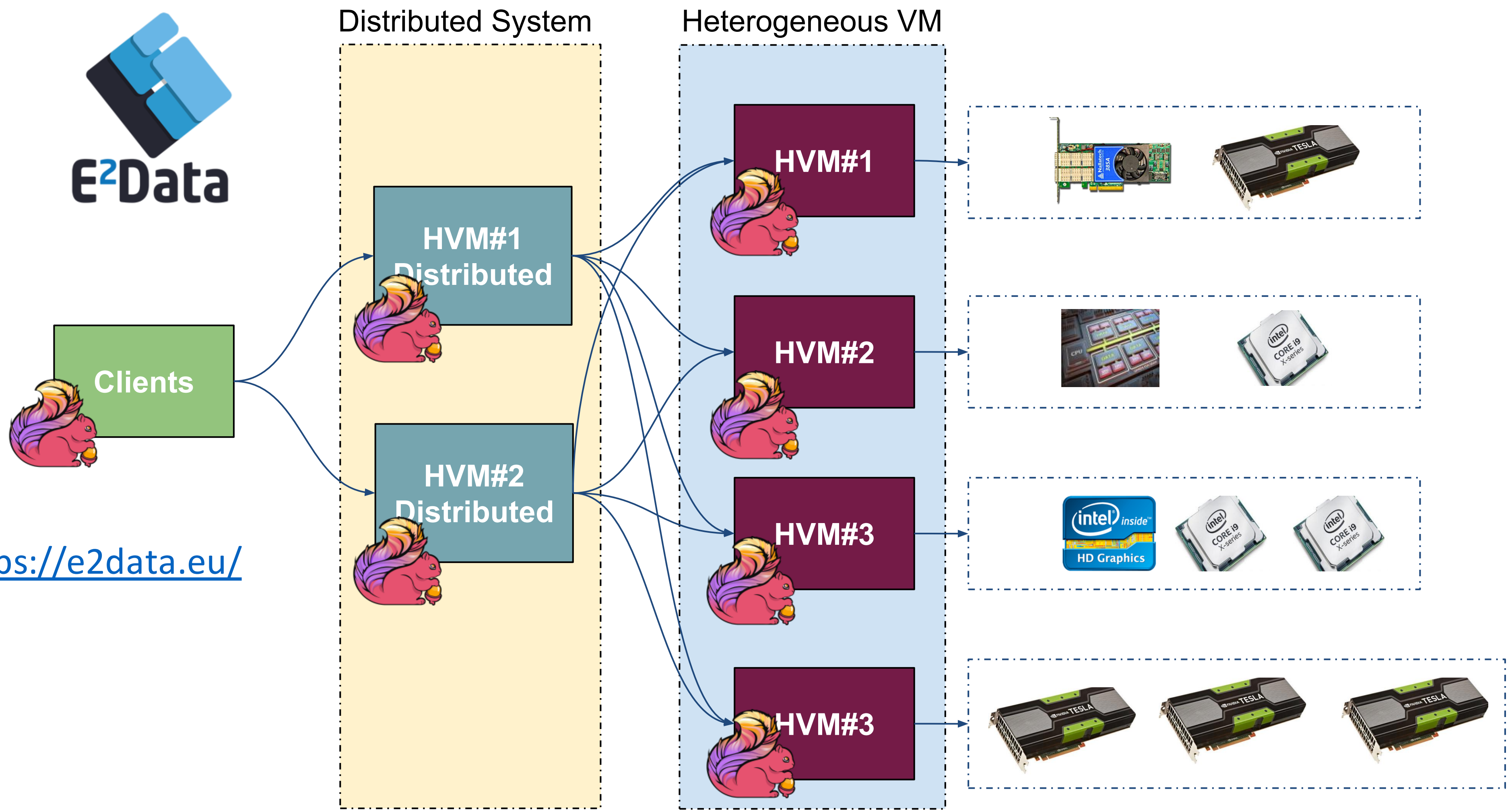


https://e2data.eu/

"End-to-end solutions for heterogeneous Big Data deployments that fully exploit heterogeneous hardware"

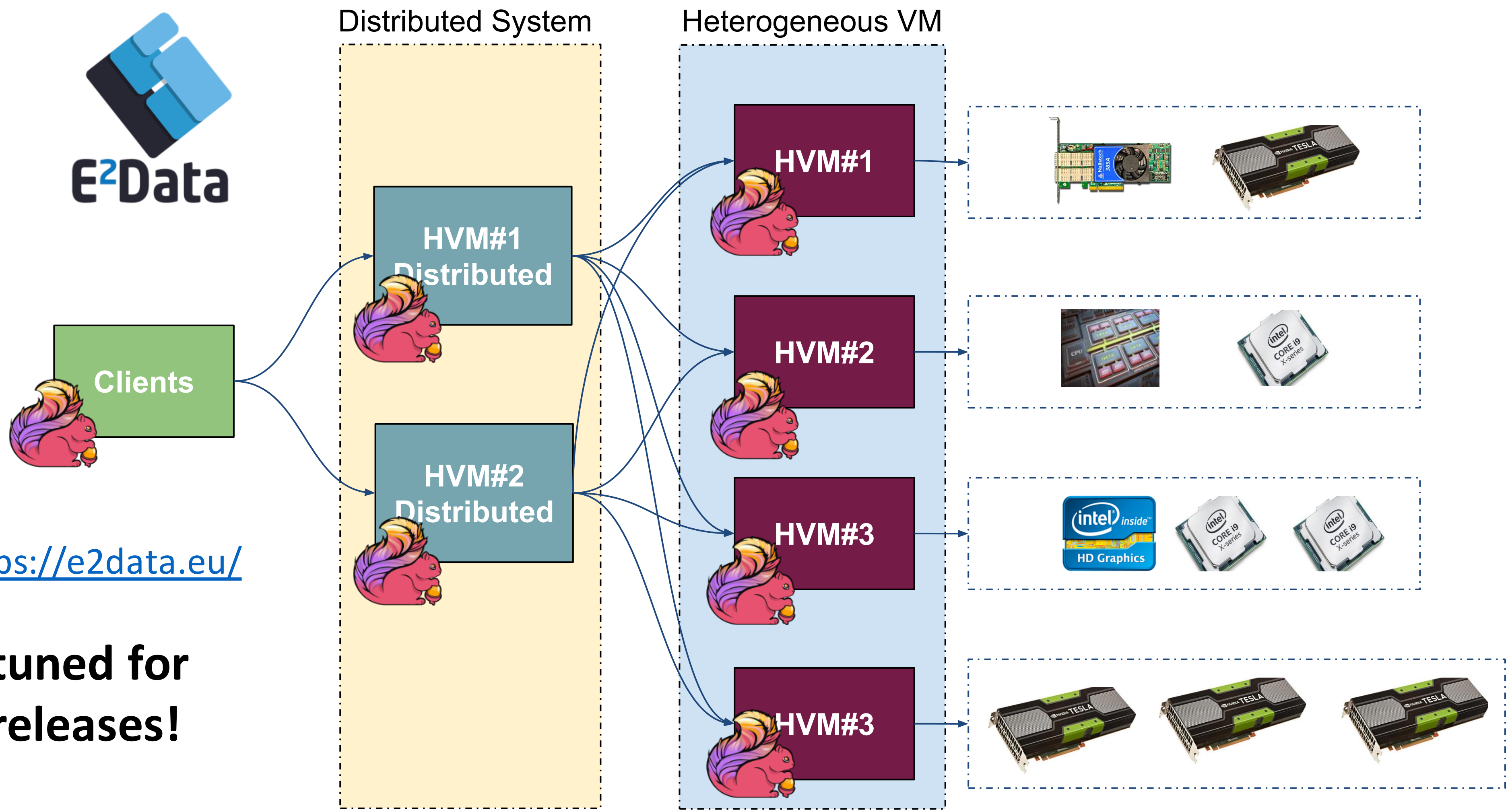
European Union's Horizon H2020 research and innovation programme under grant agreement No 780245

E2Data Project – Distributed H. System with Apache Flink & TornadoVM



https://e2data.eu/

E2Data Project – Distributed H. System with Apache Flink & TornadoVM



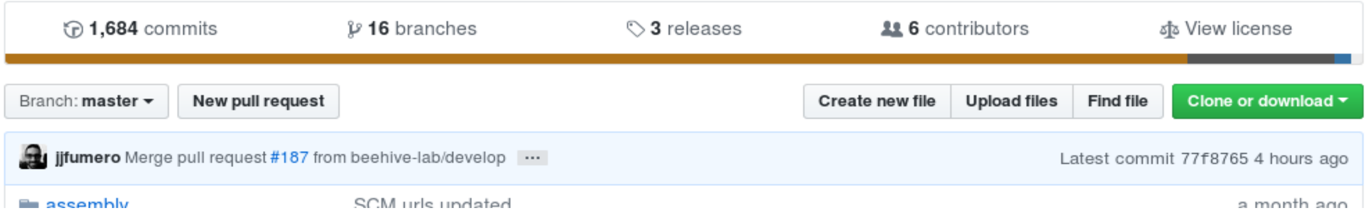
https://e2data.eu/

Stay tuned for new releases!

To sum up …

MANCHESTER 1824
The University of Manchester

TornadoVM available on Github and DockerHub



https://github.com/beehive-lab/TornadoVM

```
$ docker pull beehivelab/tornado-gpu

# And RUN !
$ ./run_nvidia.sh javac.py YouApp.java
$ ./run_nvidia.sh tornado YourApp
```

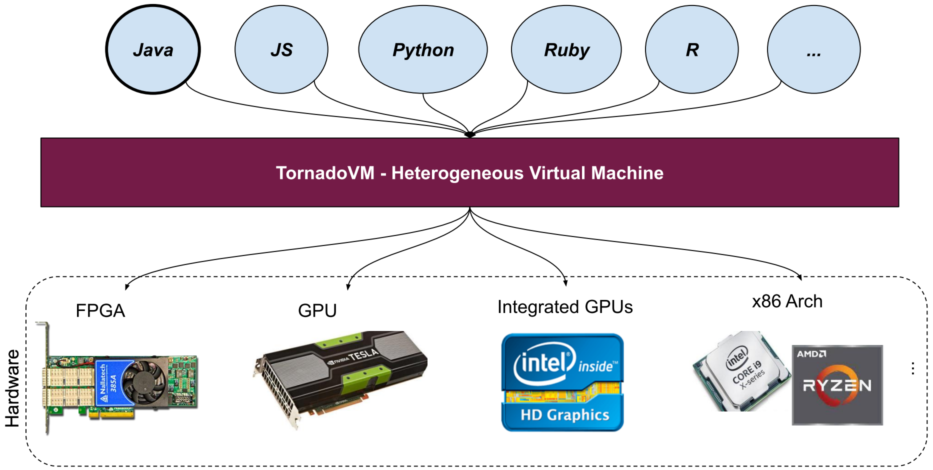
https://github.com/beehive-lab/docker-tornado

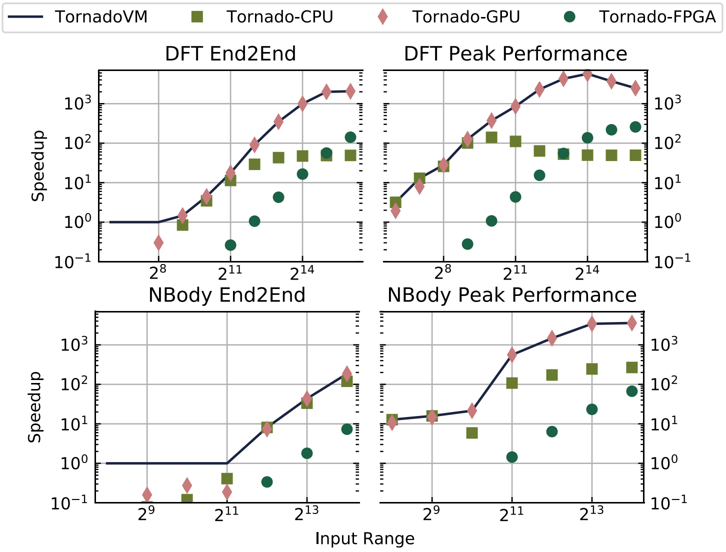
Team

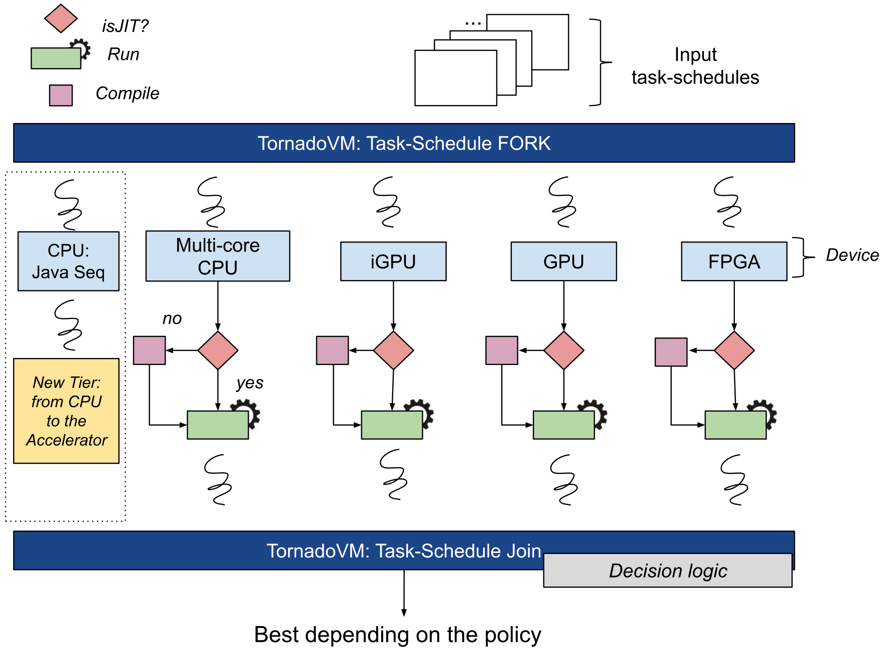
- Academic staff:
  Christos Kotselidis
- Research staff:
  Juan Fumero
  Athanasios Stratikopoulos
  Foivos Zakkak
- Alumni:
  James Clarkson
- PhD Students:
  Michail Papadimitriou
  Maria Xekalaki
- Interns:
  Benjamin Bell
- Undergraduates:
  Amad Aslam

We are looking for collaborations

Takeaways







https://e2data.eu

Thank you so much for your attention

This work is partially supported by the EU Horizon 2020 E2Data 780245

European Commission

E2Data

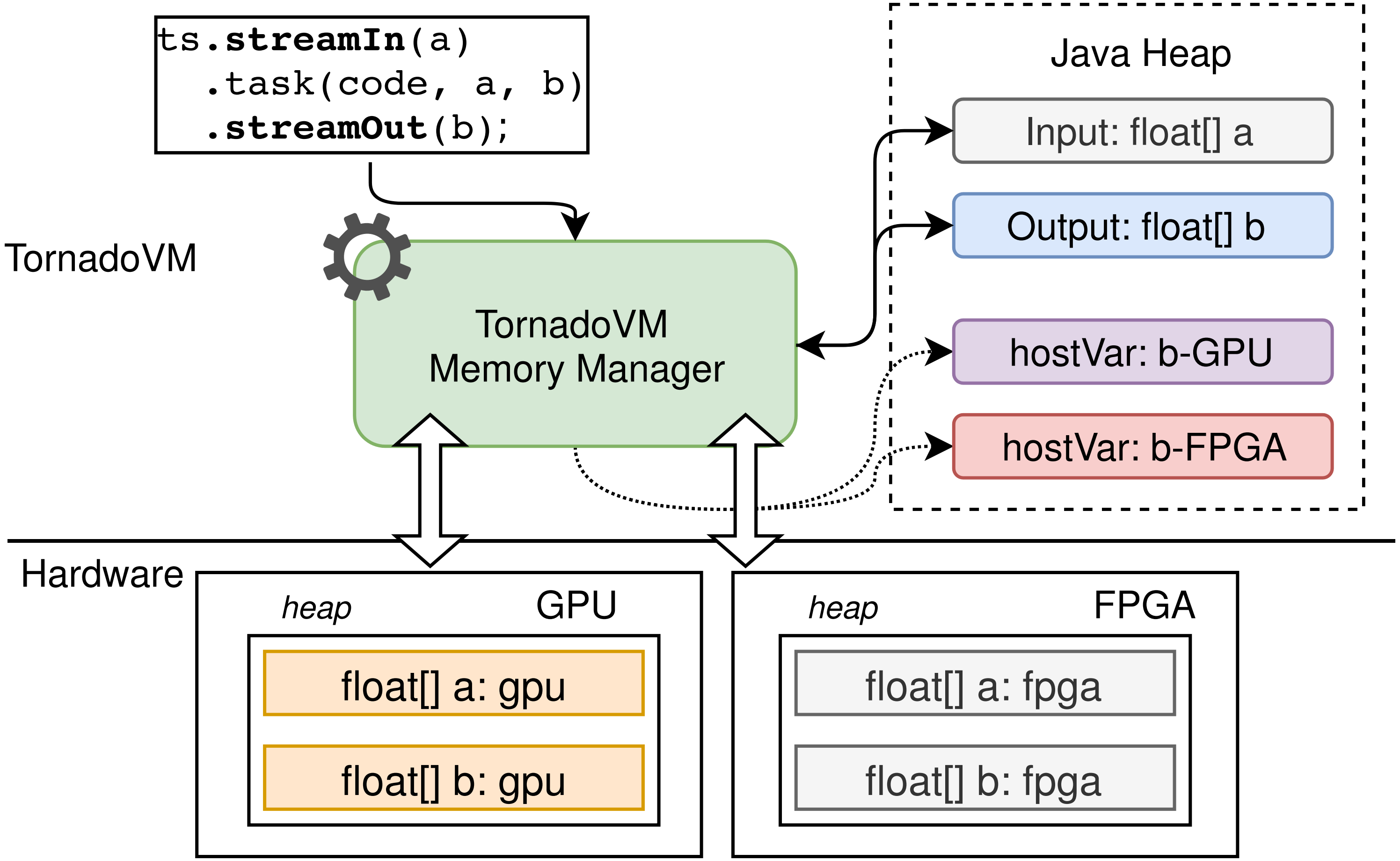
MANCHESTER 1824

The University of Manchester

Contact: Juan Fumero <juan.fumero@manchester.ac.uk>

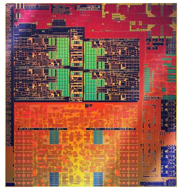
Back up slides

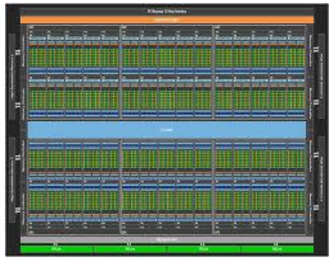
Memory Management in a Nutshell



- Host Variables: read-only in the JVM heap, R/W or W then we perform a new copy.
- Device Variables: a new copy unless OpenCL zero copy, e.g., iGPU

How to Program? E.g., OpenCL





1. Query OpenCL Platforms
2. Query devices available
3. Create device objects
4. Create an execution context
5. Create a command queue
6. Create and compile the GPU Kernels
7. Create <GPU> buffers
8. Create buffers and send data (Host -> Device)

9. Run <GPU> Kernel

10. Send data back (Device -> Host)
11. Free Memory

More details in our VEE#2019 Paper

MANCHESTER 1824
The University of Manchester

Dynamic Application Reconfiguration on Heterogeneous Hardware

Juan Fumero
The University of Manchester
United Kingdom
juan.fumero@manchester.ac.uk

Michail Papadimitriou
The University of Manchester
United Kingdom
mpapadimitriou@cs.man.ac.uk

Foivos S. Zakkak
The University of Manchester
United Kingdom
foivos.zakkak@manchester.ac.uk

Maria Xekalaki
The University of Manchester
United Kingdom
maria.xekalaki@manchester.ac.uk

James Clarkson
The University of Manchester
United Kingdom
james.clarkson@manchester.ac.uk

Christos Kotselidis
The University of Manchester
United Kingdom
ckotselidis@cs.man.ac.uk

Abstract

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

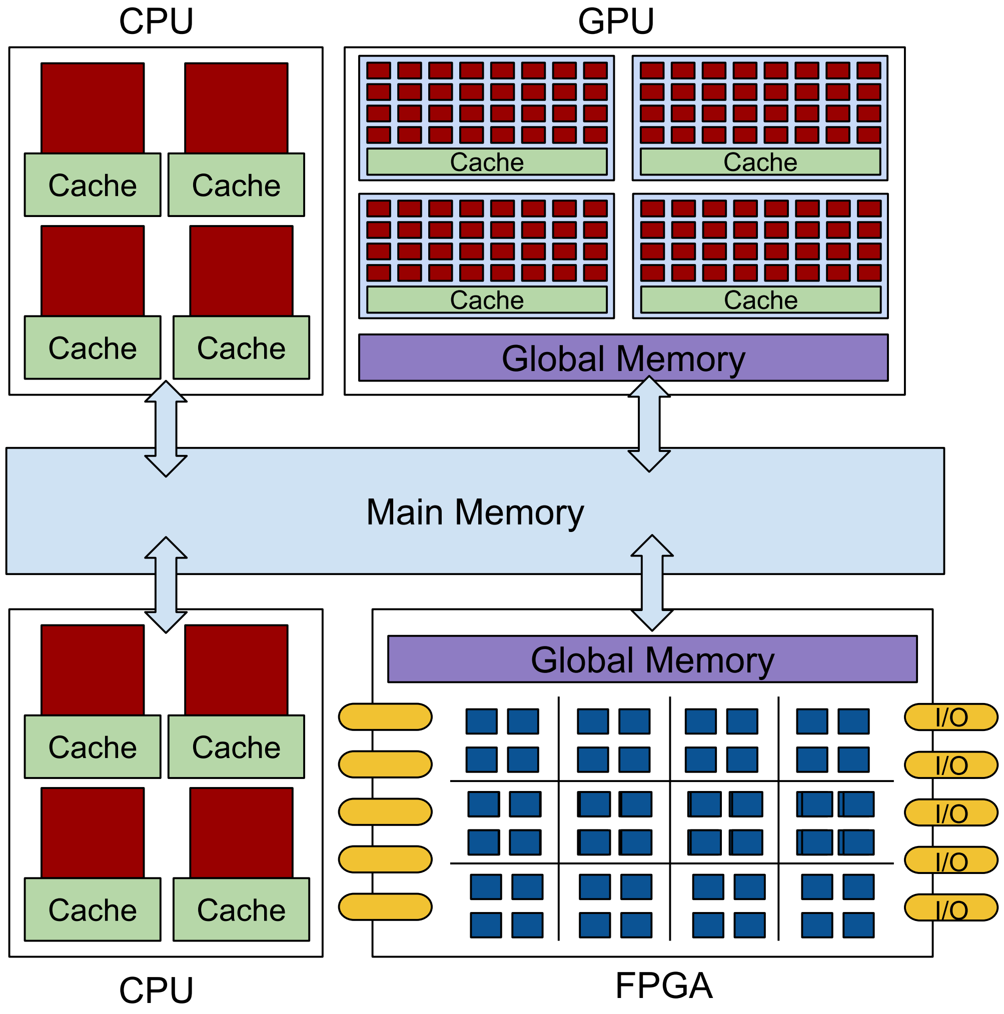
This paper introduces TornadoVM, a virtual machine capable of reconfiguring applications, at run-time, for hardware acceleration based on the currently available hardware resources. Through TornadoVM, we introduce a new level of compilation in which applications can benefit from heterogeneous hardware.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

Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3313808.3313819

1 Introduction

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

CPU, GPU, FPGA interconnection



CPU Cores:
* 4-8 cores per CPU
* Local cache (L1-L3)

GPU cores:
* Thousands of cores per GPU card
* > 60 cores per SM
* Small caches per SM
* Global memory within the GPU
* Few thread/schedulers per SM

FPGAs:
* Chip with LUTs, BRAMs, and wires to
* Normally global memory within the chip

OpenCL Generated Kernel

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void vectorAdd(__global uchar *_heap_base, ulong _frame_base, .. )
{
  int i_9, i_11, i_4, i_3, i_13, i_14, i_15;
  long l_7, l_5, l_6;
  ulong ul_0, ul_1, ul_2, ul_12, ul_8, ul_10;

  __global ulong *_frame = (__global ulong *) &_heap_base[_frame_base];

  // BLOCK 0
  ul_0  =  (ulong) _frame[6];
  ul_1  =  (ulong) _frame[7];
  ul_2  =  (ulong) _frame[8];
  i_3  =  get_global_id(0);
  // BLOCK 1 MERGES [0 2 ]
  i_4  =  i_3;
  for(;i_4 < 256;)  {
    // BLOCK 2
    l_5  =  (long) i_4;
    l_6  =  l_5 << 2;
    l_7  =  l_6 + 24L;
    ul_8  =  ul_0 + l_7;
    i_9  =  *((__global int *) ul_8);
    ul_10  =  ul_1 + l_7;
    i_11  =  *((__global int *) ul_10);
    ul_12  =  ul_2 + l_7;
    i_13  =  i_9 + i_11;
    *((__global int *) ul_12)  =  i_13;
    i_14  =  get_global_size(0);
    i_15  =  i_14 + i_4;
    i_4  =  i_15;
  }
  // BLOCK 3
  return;
}
```

Access to the Java frame

Access the data within the frame

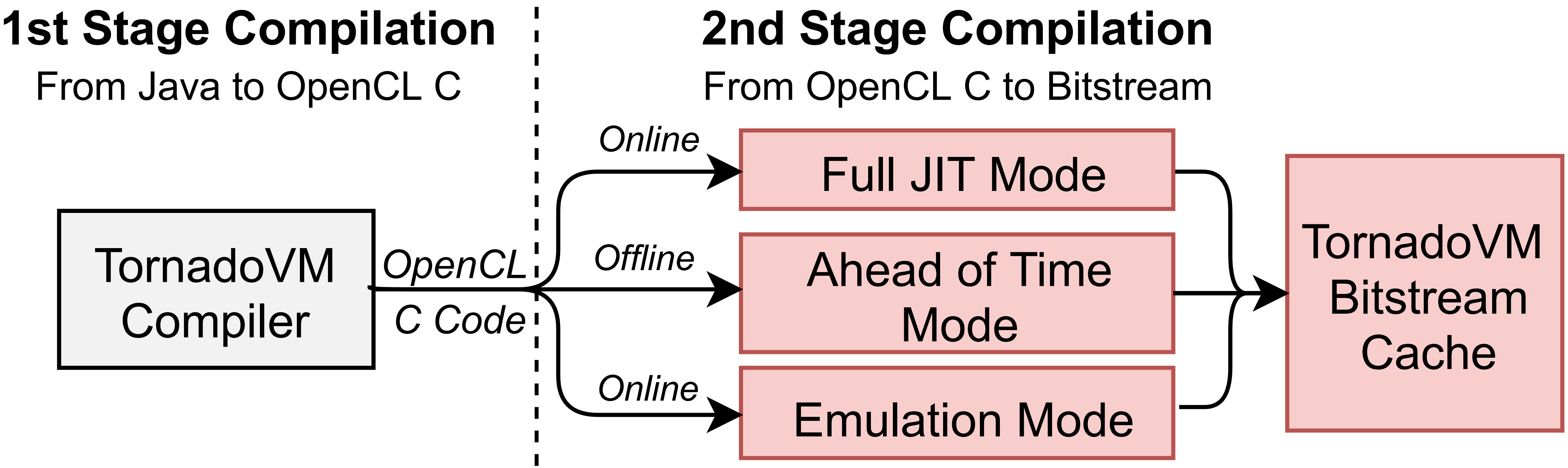
Access the arrays (skip object header)

Operation

Final Store

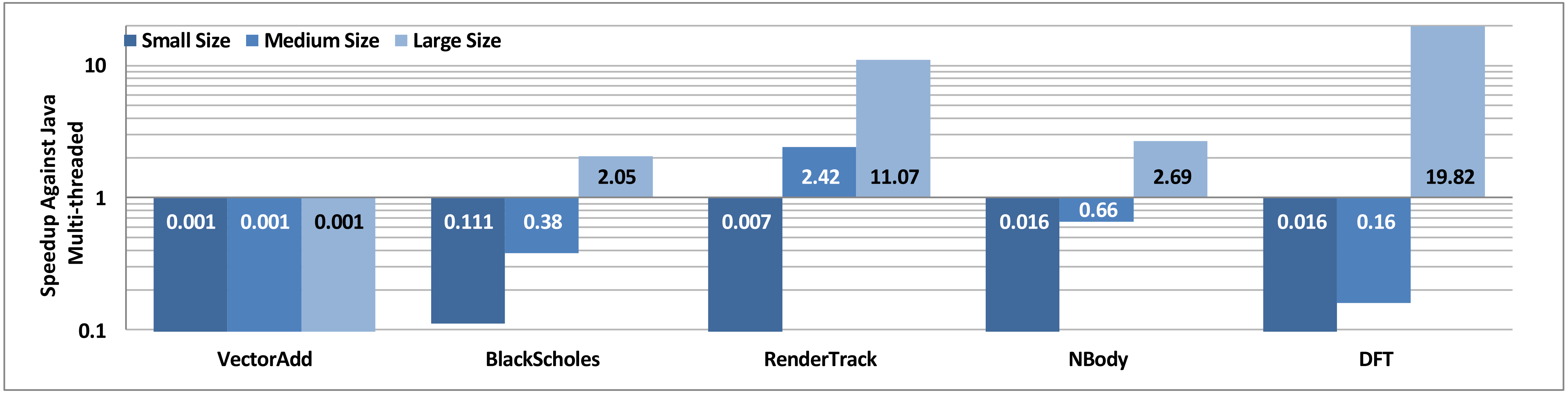
```java
private void vectorAdd(int[] a, int[] b, int[] c) {
  for (@Parallel int i = 0; i < c.length; i++) {
        c[i] = a[i] + b[i];
   }
}
```

FPGA Support




```
$ tornado  YourProgram

$ tornado –Dtornado.fpga.aot.bitstream=<path> YourProgram

$ tornado –Dtornado.fpga.emulation=True YouProgram
```

Performance: FPGA vs Multi-thread Java



* TornadoVM on FPGA is up to 19x over Java multi-threads (8 cores)
* Slowdown for small sizes