The University of Manchester

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

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

Twitter: @snatverk

Joker<?> Conference 2019, Saint Petersburg, October 26th

MANCHESTER 1824
The University of Manchester

Agenda

- Motivation & Background
- TornadoVM
  - API examples
  - Runtime
  - JIT Compiler
  - Dynamic Reconfiguration
  - Data Management
- Performance Results
- Related Work
- Conclusions

About me

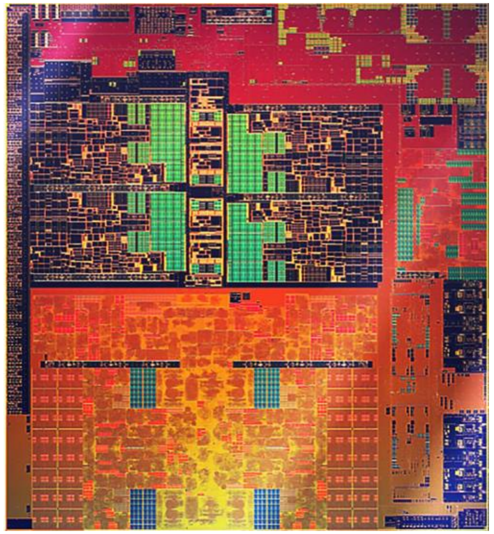
- Postdoc @ The University of Manchester (Since October 2017)
  - Currently technical lead of TornadoVM
- 2014-2017: PhD in Dynamic Compilation for GPUs using Graal & Truffle (Java, R, Ruby) @ The University of Edinburgh
- Oracle Labs alumni (worked on Truffle FastR + Flink for distributed computing)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

https://jjfumero.github.io/

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

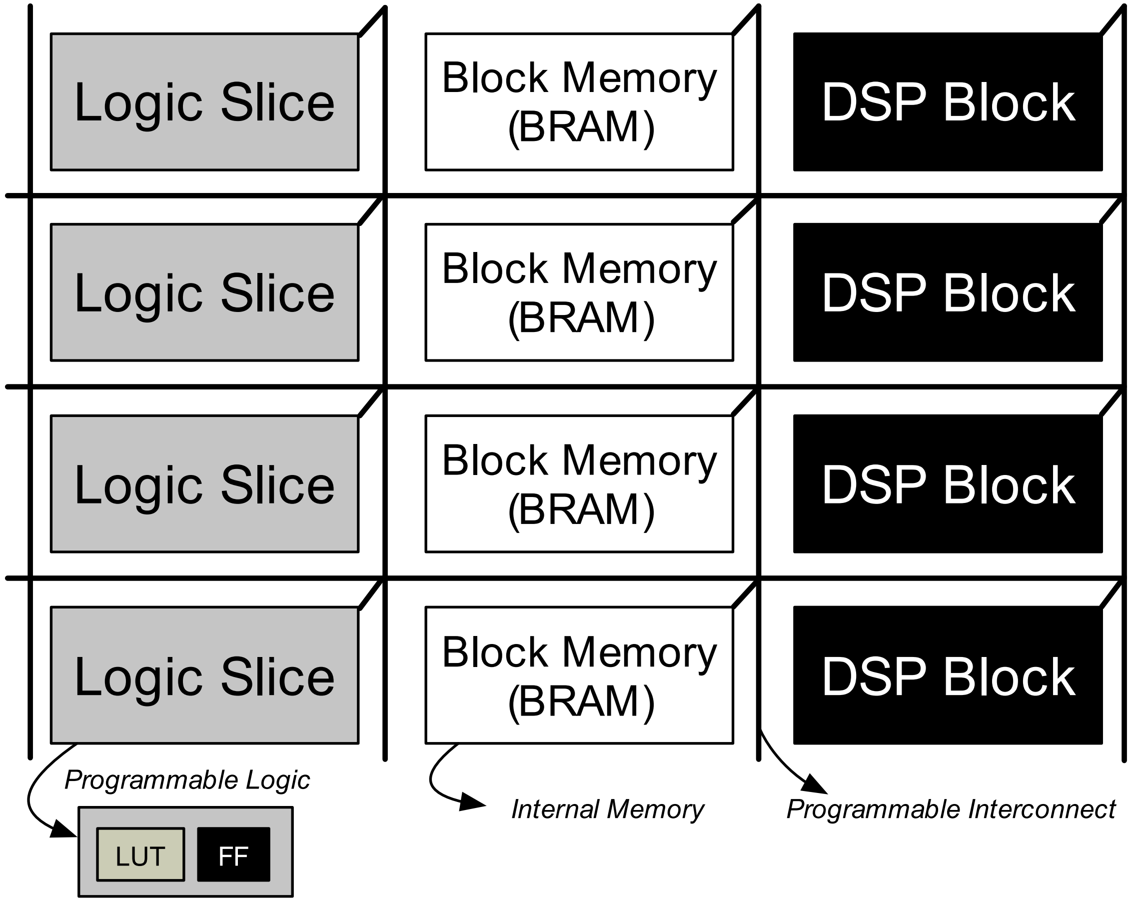
What is a GPU? Graphics Processing Unit

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.

What is an FPGA? Field Programmable Gate Array

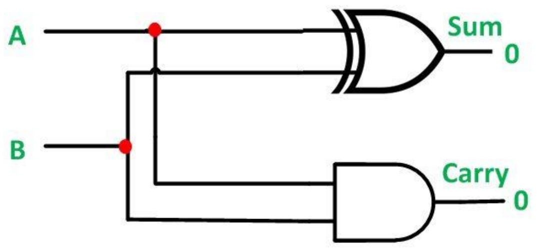


You can configure the design of your hardware after manufacturing

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

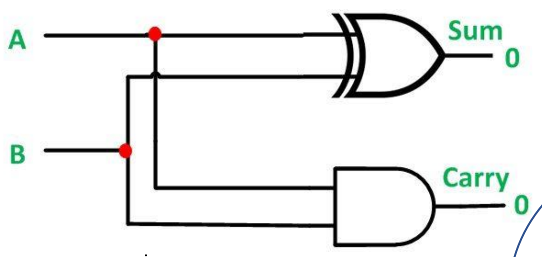
Example in VHDL (using structural modelling)

```
library ieee;
use ieee.std_logic_1164.all;

entity half_adder is                -- Entity
 port (a, b: in std_logic;
   sum, carry: out std_logic);
end half_adder;

architecture structure of half_adder is    -- Architecture
  component xor_gate                -- xor component
   port (i1, i2: in std_logic;
     o1: out std_logic);
  end component;

  component and_gate                -- and component
   port (i1, i2: in std_logic;
     o1: out std_logic);
  end component;

begin
   u1: xor_gate port map (i1 => a, i2 => b, o1 => sum);
   u2: and_gate port map (i1 => a, i2 => b, o1 => carry);
end structure;
```



Using OpenCL instead

```
library ieee;
use ieee.std_logic_1164.all;

entity half_adder is              -- Entity
 port (a, b: in std_logic;
   sum, carry: out std_logic);
end half_adder;

architecture structure of half_adder is    -- Architecture
 component xor_gate                -- xor component
  port (i1, i2: in std_logic;
   o1: out std_logic);
 end component;

 component and_gate                -- and component
  port (i1, i2: in std_logic;
   o1: out std_logic);
 end component;

begin
  u1: xor_gate port map (i1 => a, i2 => b, o1 => sum);
  u2: and_gate port map (i1 => a, i2 => b, o1 => carry);
end structure;
```

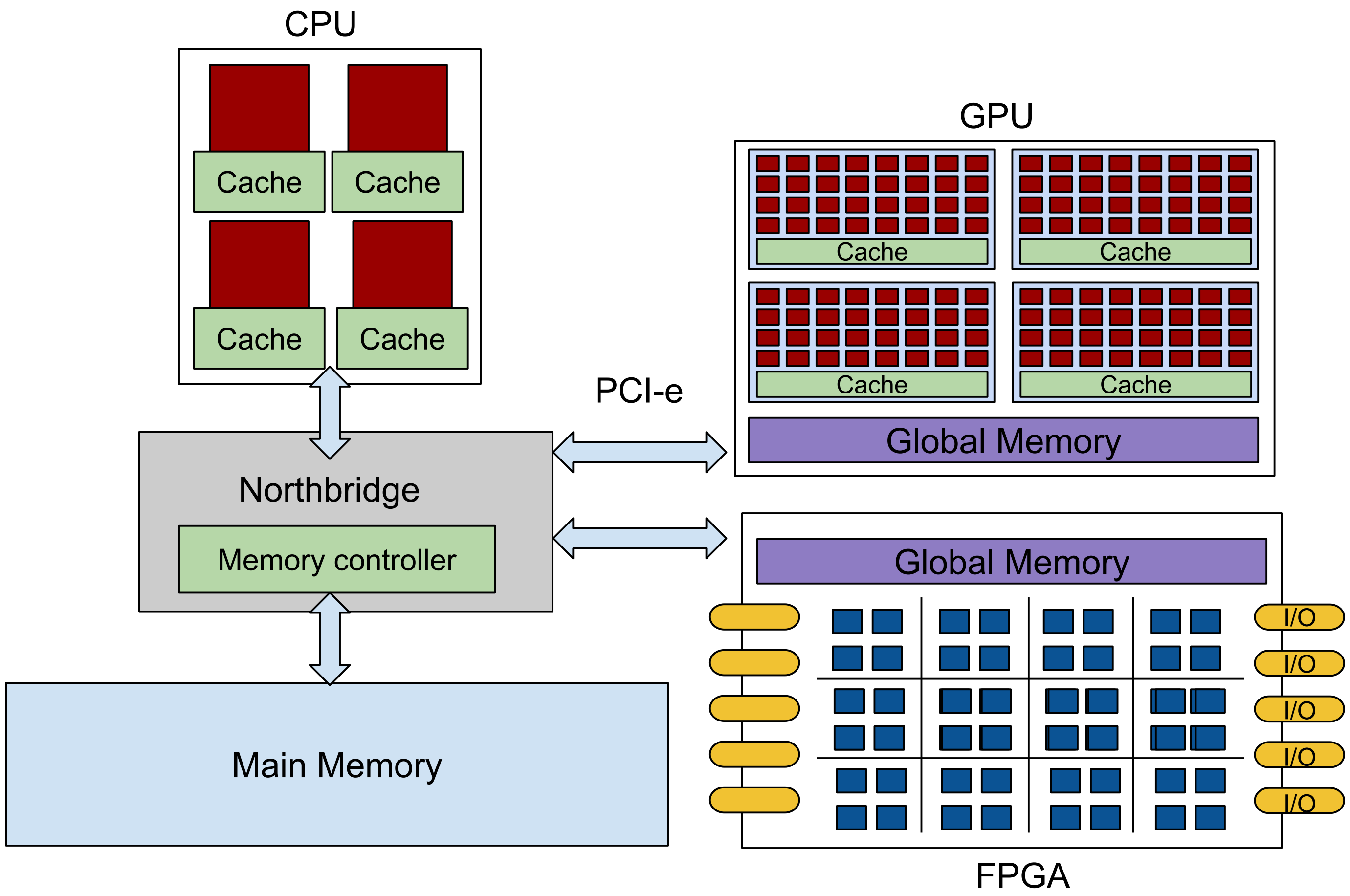



```
_kernel void sum
(float a,float b,__global
float*result)
{
    result[0] = a + b;
}
```

Industry is pushing for OpenCL on FPGAs!

OpenCL

We could potentially use ALL devices!

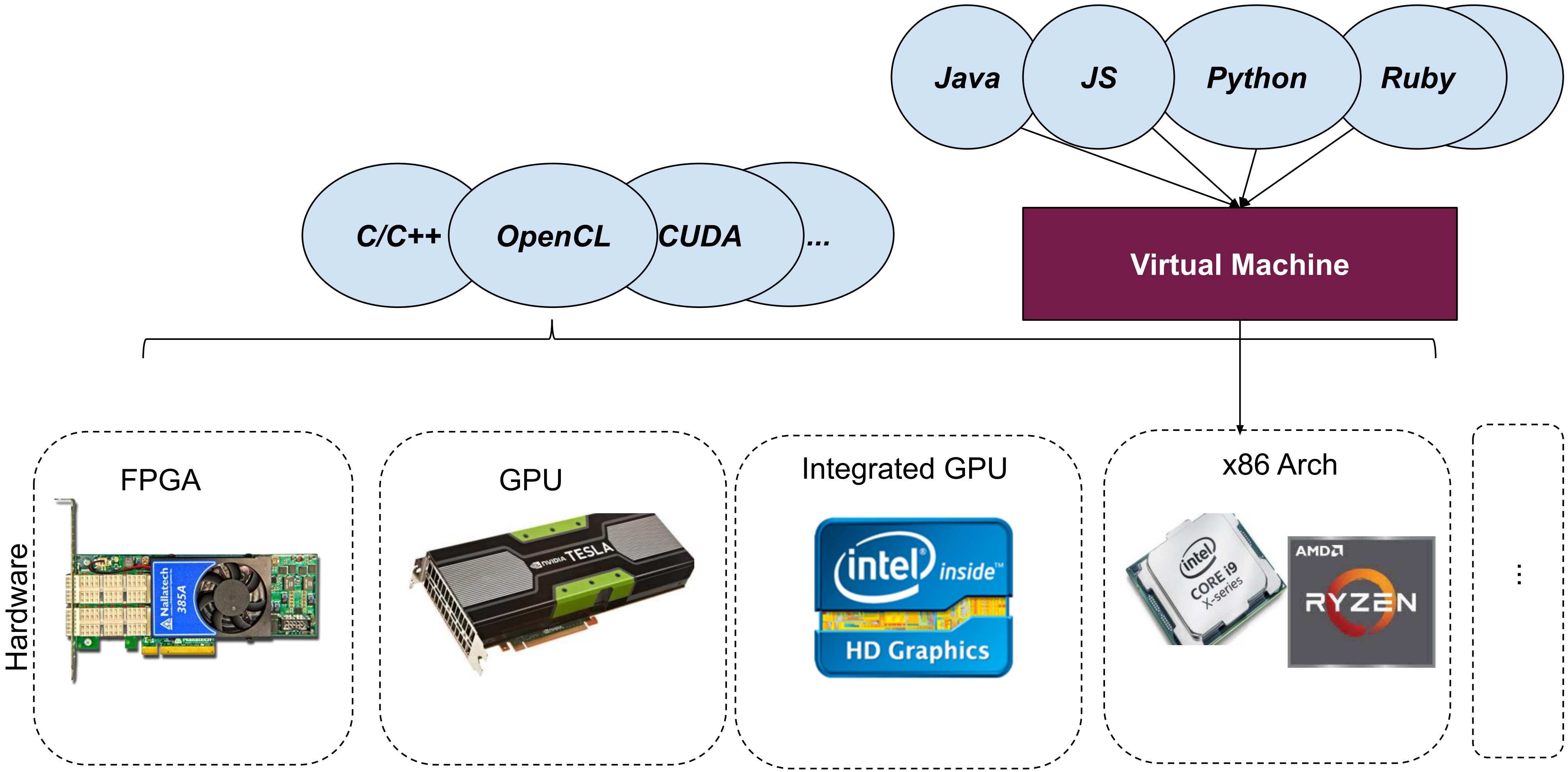


CPU Cores:
* 4-8 cores per CPU
* Local cache (L1-L3)

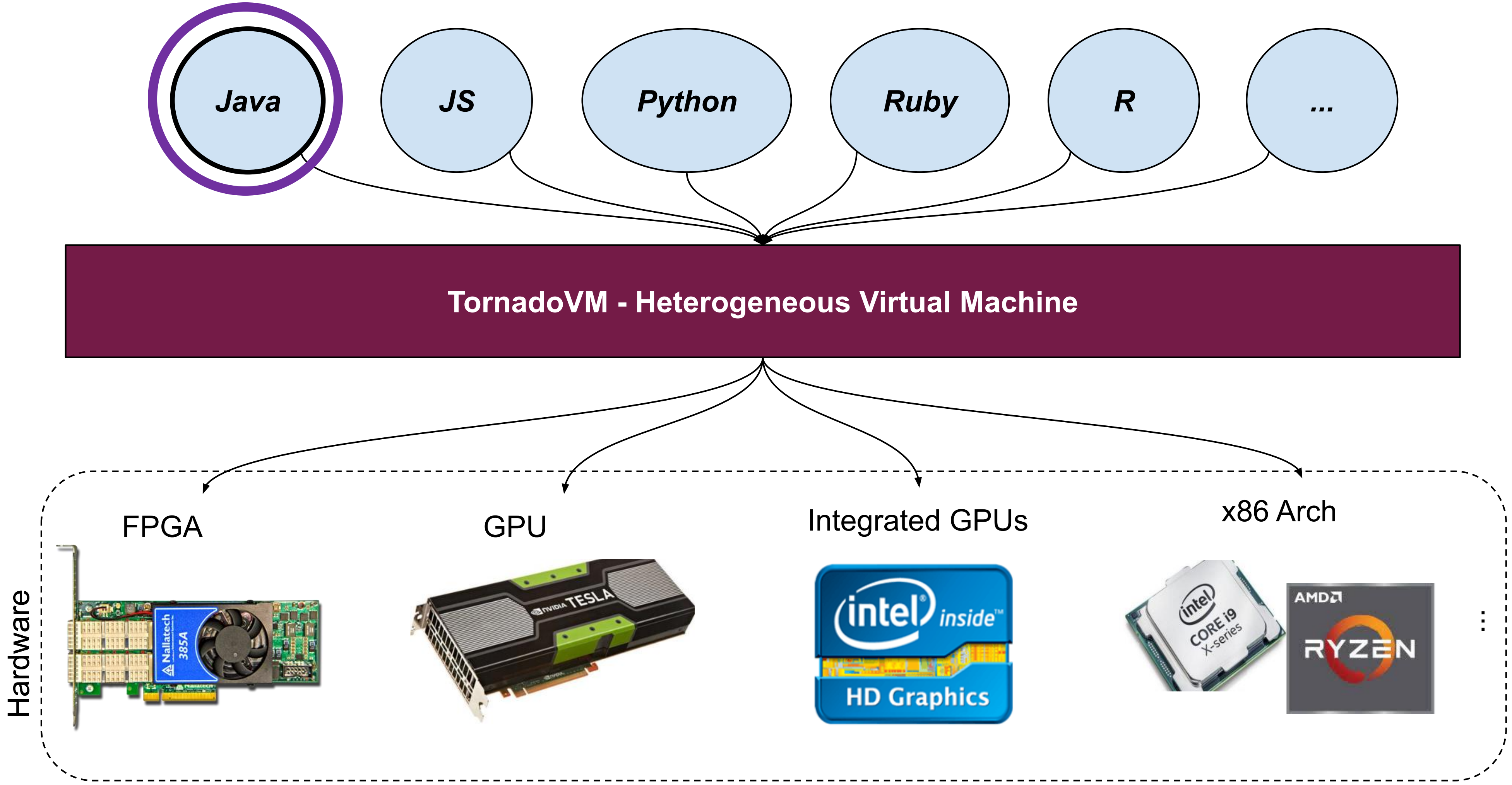
GPU cores:
* Thousands of cores per GPU card
* > 60 cores per SM
* Small caches per SM
* Global memory within the GPU
* Few thread/schedulers per SM

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

Current Computer Systems & Prog. Lang.

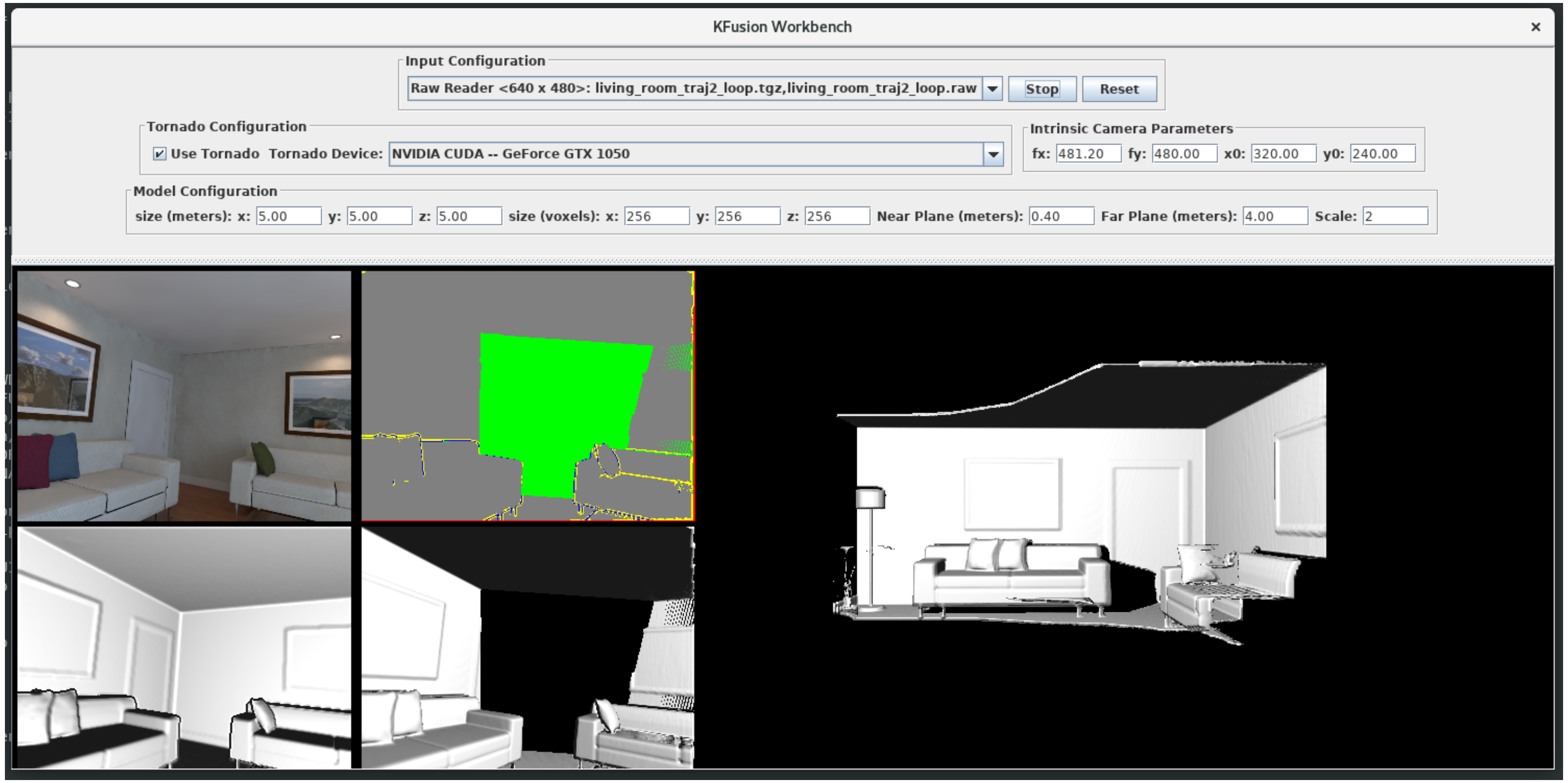


Ideal System for Managed Languages



TornadoVM

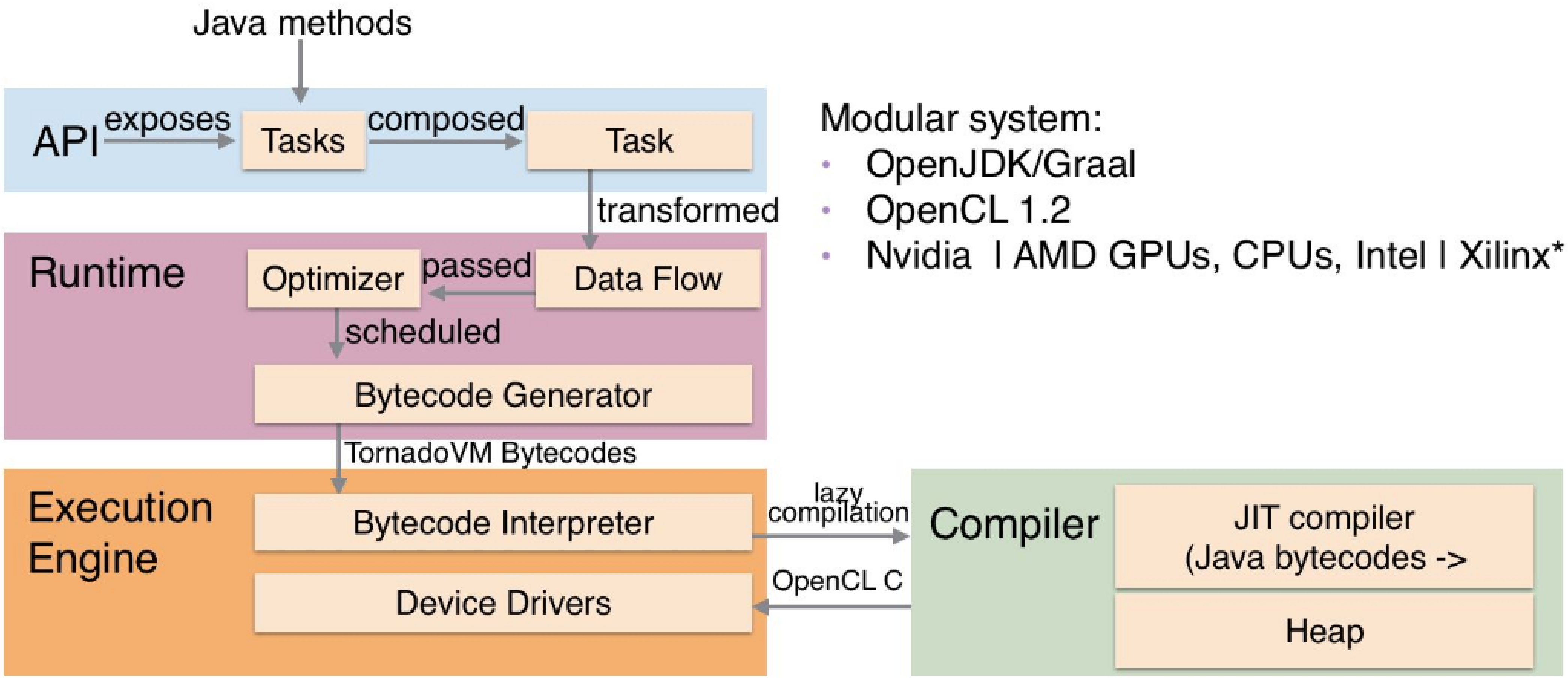
Demo: Kinect Fusion with TornadoVM



* 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
- Nvidia | AMD GPUs, CPUs, Intel | Xilinx* FPGAs

Tornado API – example

```
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

```
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

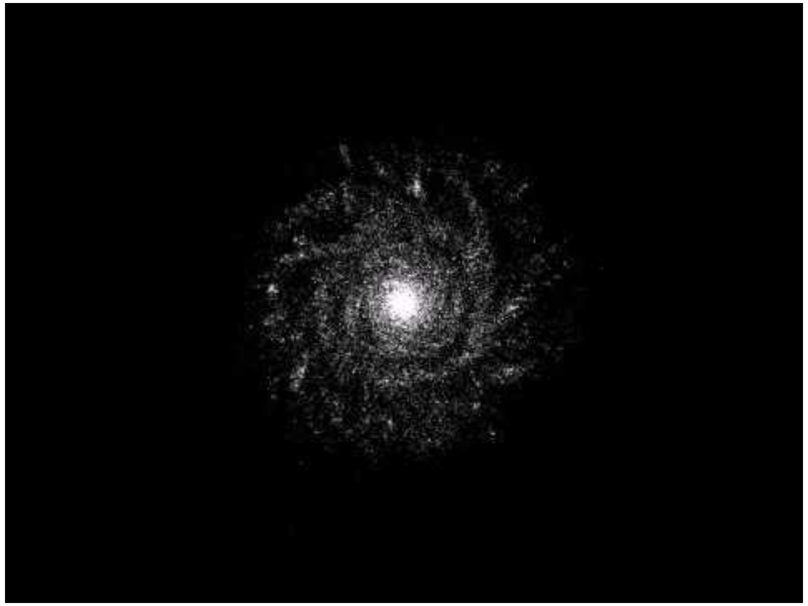
The University of Manchester

```
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

Demo: N-Body with TornadoVM

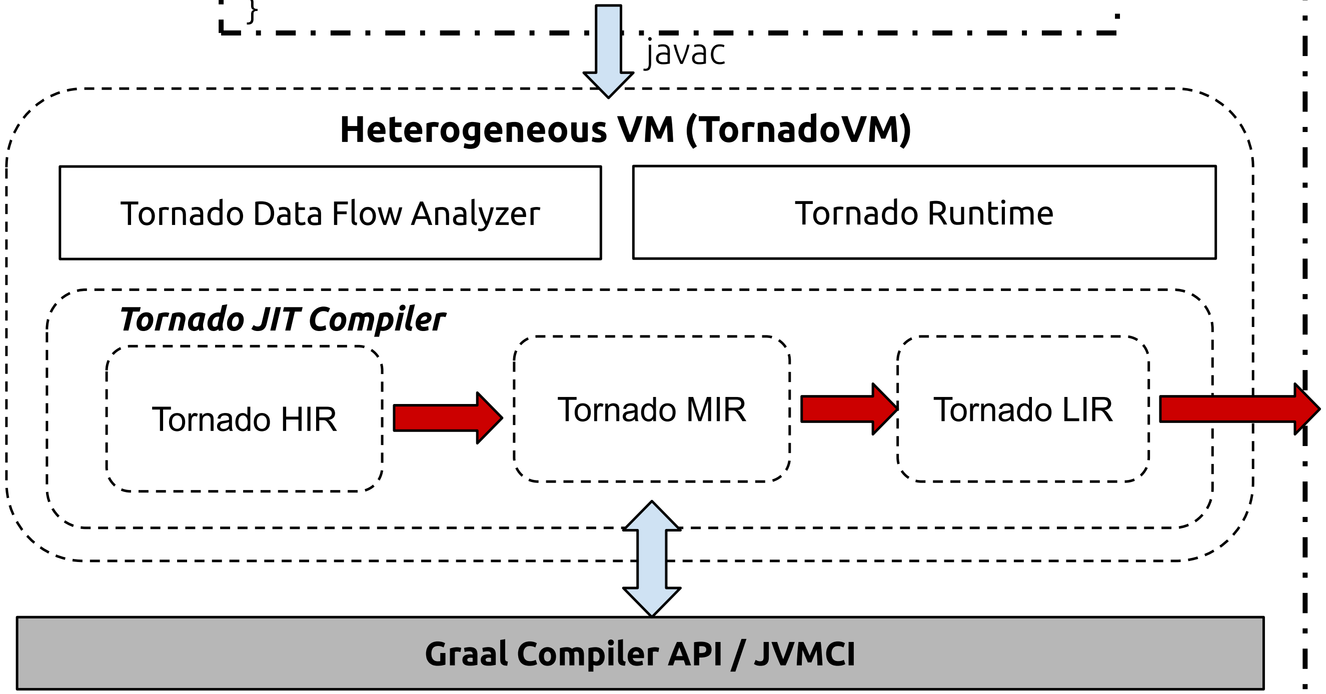


github.com/beehive-lab/TornadoVM/tree/master/examples

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

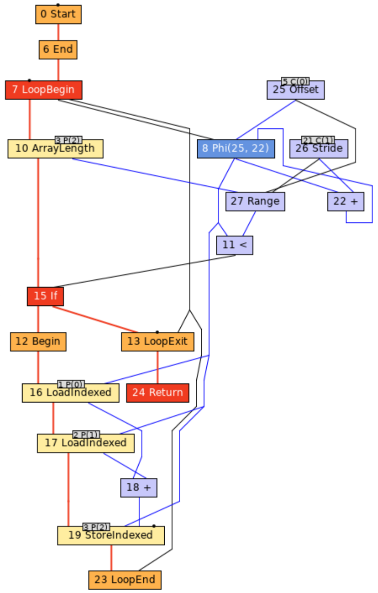
```c
kernel void compute(global uchar *frame, ...) {
  // data types declaration
  …
  ul_0  =  (ulong) frame[7];
  ul_1  =  (ulong) frame[8];
  i_2  =  get_global_id(0);
  i_3  =  i_2;

  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;
  }
}
```

TornadoVM JIT Compiler Specializations

Input Java code

```java
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

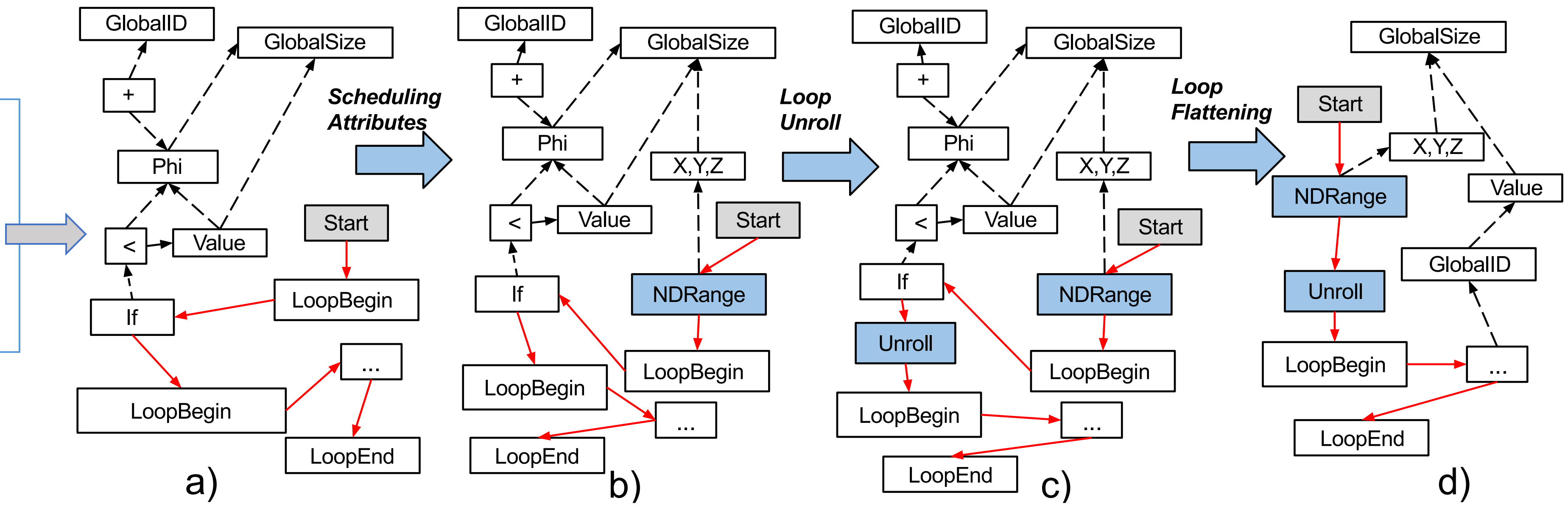

```c
int idx = get_global_id(0);
int size = get_global_size(0);
for (int i = idx; i < c.length; i += size) {
  c[i] = a[i] + b[i];
}
```

CPU Specialization


```c
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

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



FPGA Specializations

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

Non-specialized version

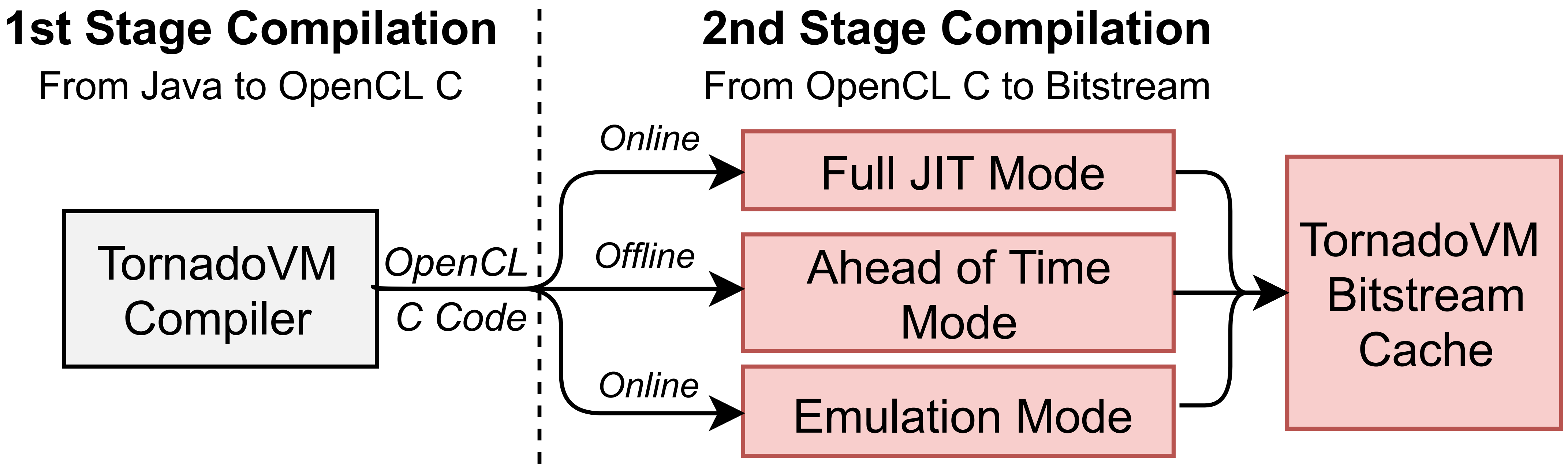
```
__kernel void dft(__global uchar *_heap_base,
                  ulong _frame_base, ... ) {
// variable declaration
...
__global ulong *_frame = (__global ulong *) &_heap_base[_frame_base];

base0 =  (ulong) _frame[6];
base1 =  (ulong) _frame[7];
base2 =  (ulong) _frame[7];
tid   =  get_global_id(0);
...
i8  =  *((__global int *) &_heap_base[base0]);
for(;tid < maxElements)  {
   ...
   f10  =  0.0F;
   i11  =  0;
   for(;i11 < i8;)    {
     ...
   }
   ul_38  =  base1 + index;
   *((__global float *) &_heap_base[ul_38])  =  result1;
   ul_37  =  base2 + index;
   *((__global float *) &_heap_base[ul_39])  =  result2;
   i_40  =  get_global_size(0);
   i_41  =  i_40 + tid;
   tid  =  i_41;

}
```

Specialized version

```
// Scheduling attributes
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void compute(__global uchar *_heap_base,
                      ulong _frame_base, ... ) {
// variable declaration
...
__global ulong *_frame = (__global ulong *) &_heap_base[_frame_base];

base0 =  (ulong) _frame[6];
base1 =  (ulong) _frame[7];
base2 =  (ulong) _frame[7];
tid   =  get_global_id(0);              // Loop flattening
...
i8  =  *((__global int *) &_heap_base[base0]);
...
f10  =  0.0F;
i11  =  0;
 #pragma unroll 2     // Loop unrolling with factor 2
 for(;i11 < i8;)    {
   ...
 }
 ul_38  =  base1 + index;
 *((__global float *) &_heap_base[ul_38])  =  result1;
 ul_37  =  base2 + index;
 *((__global float *) &_heap_base[ul_39])  =  result2;
}
```

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

More About FPGA Support



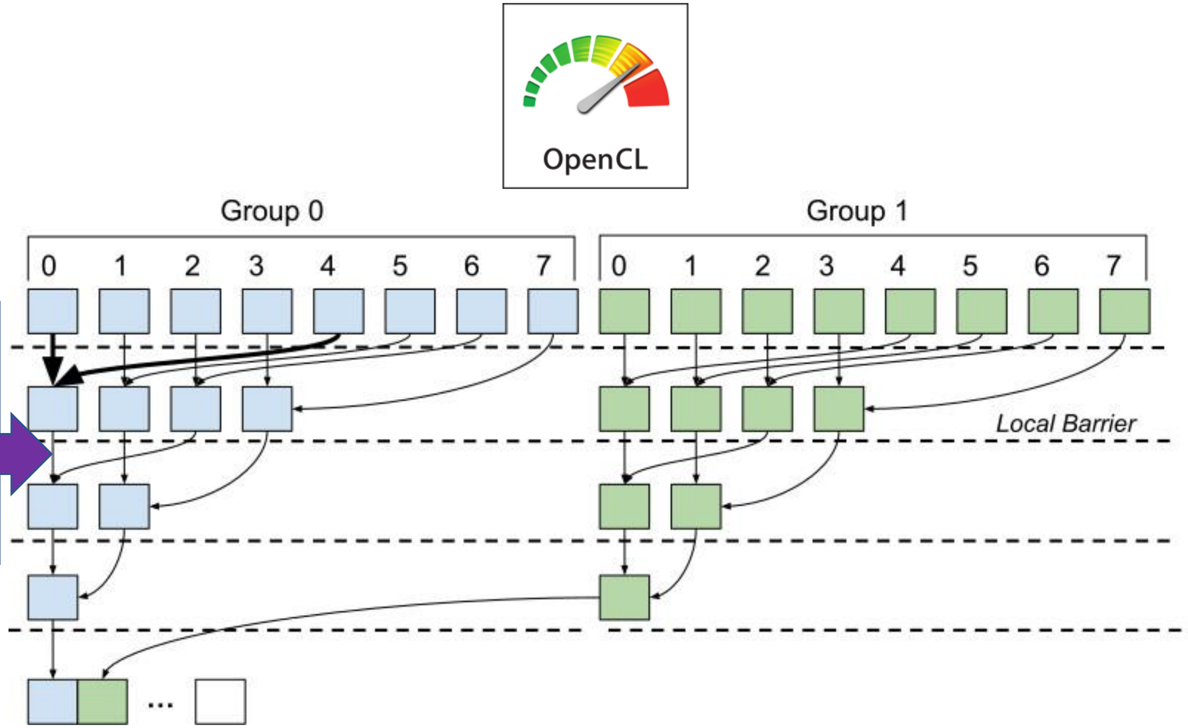
$ tornado YourProgram

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

$ tornado –Dtornado.fpga.emulation=True YouProgram

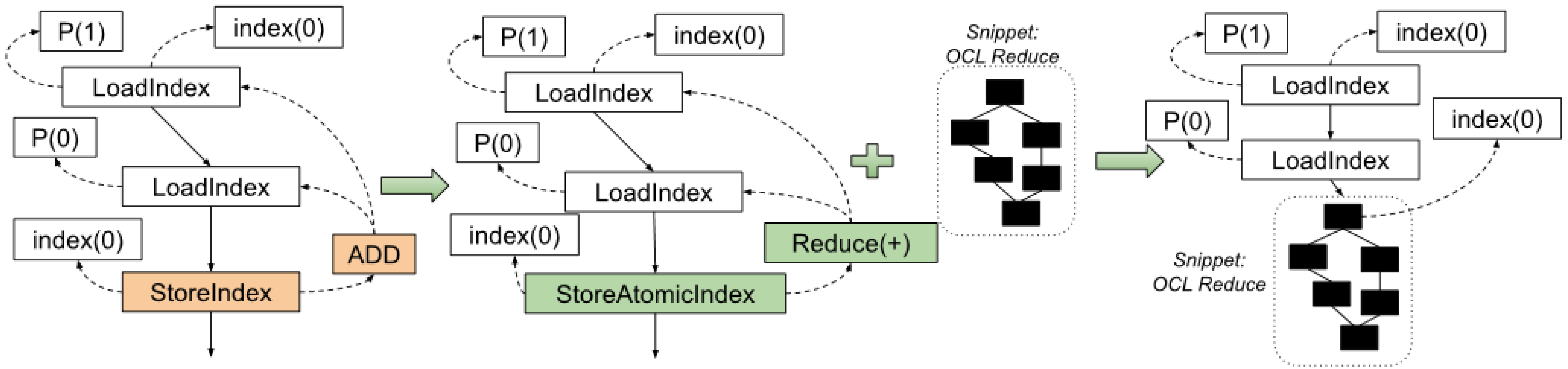
Specializations: reductions

```
void reduce(float[] input,  @Reduce float[] output) {
 for (@Parallel int i = 0; I < N; I++) {
     output[0] += input[I];
 }
}
```



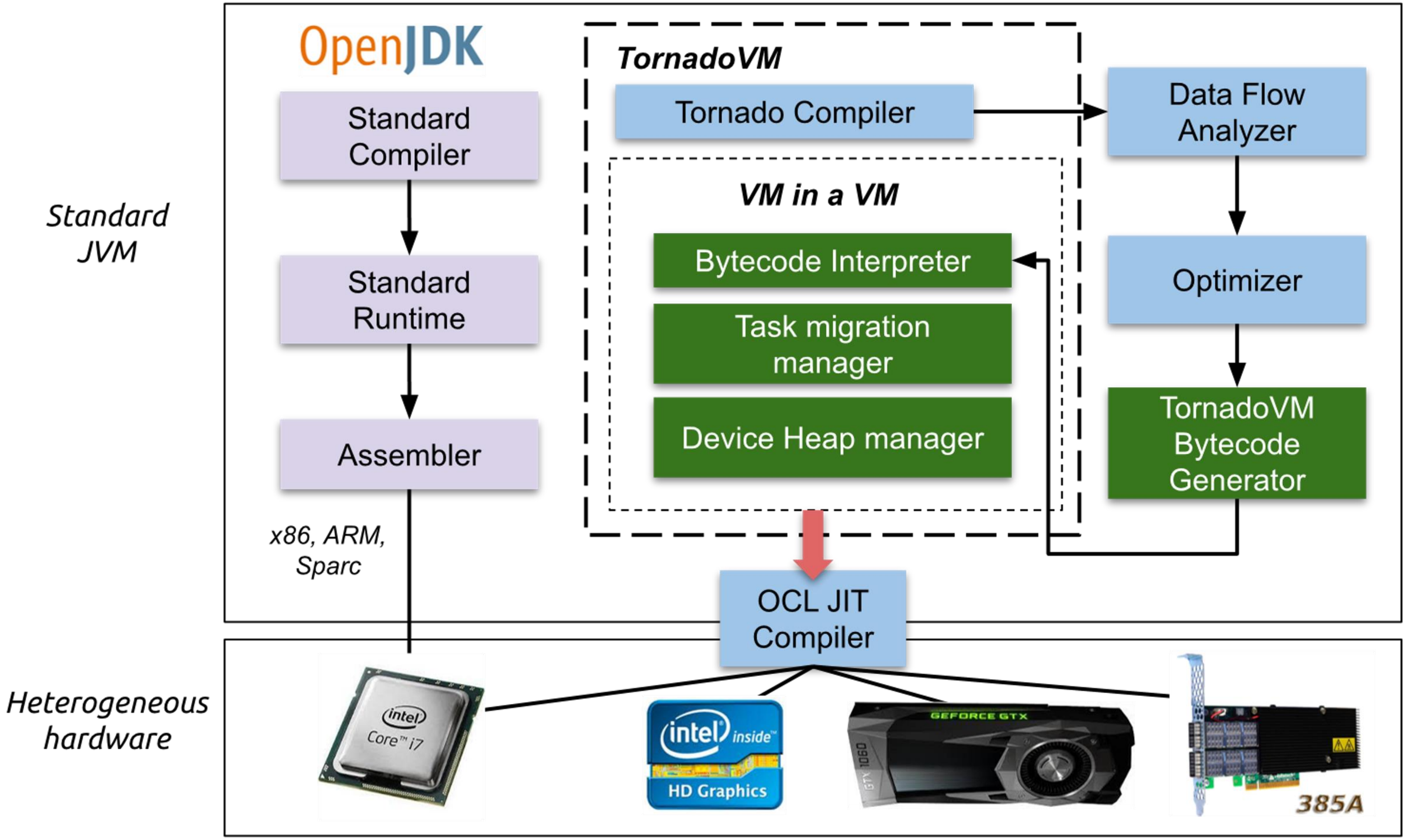
... but how?

Reduction Specializations via Snippets



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

TornadoVM: VM in a VM

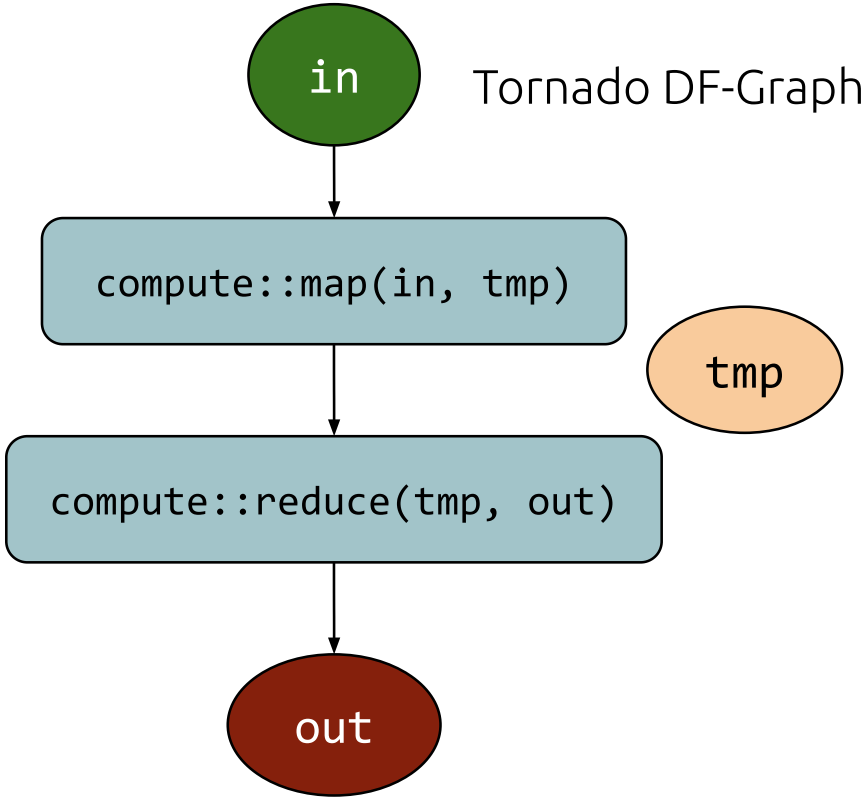


TornadoVM Bytecodes - Example

Input Java code

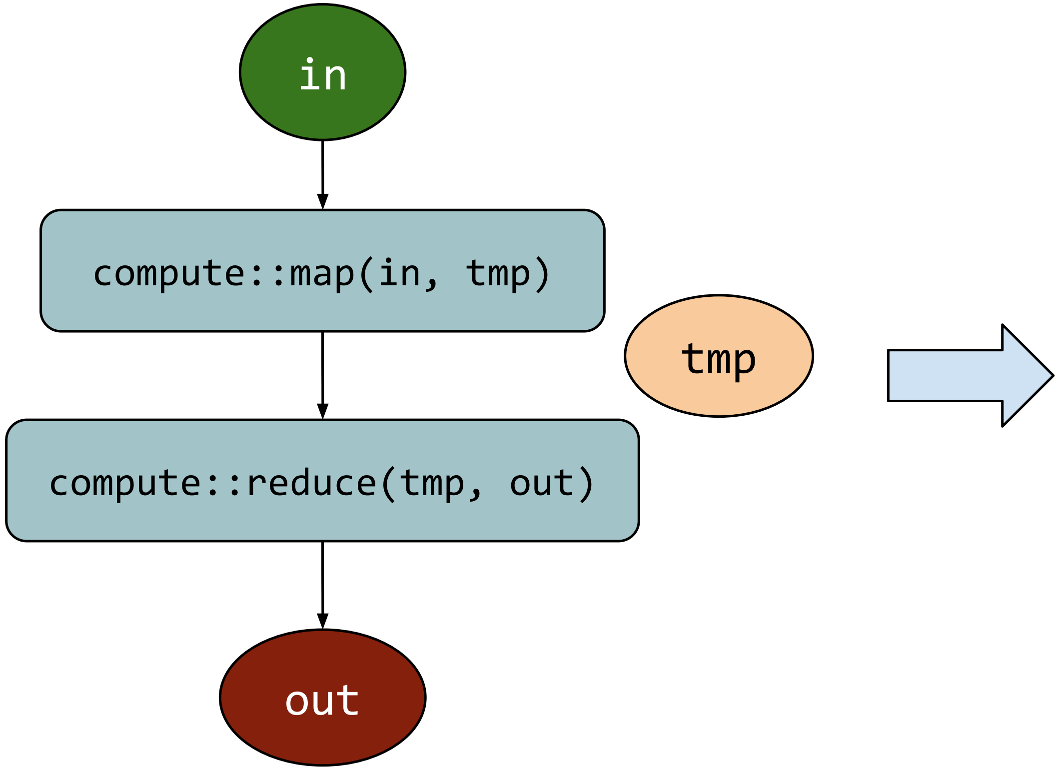
```
public class Compute {
  public void map(float[] in, float[] out) {
    for (@Parallel int i = 0; i < n; i++) {
      out[i] = in[i] * in[i];
  }}
  public void reduce(float[]in,@Reduce float[]out) {
    for (@Parallel int i = 0; i < n; i++) {
      out[0] += in[i];
  }}
  public static void compute(float[] 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



TornadoVM Bytecodes - Example

Tornado DF-Graph

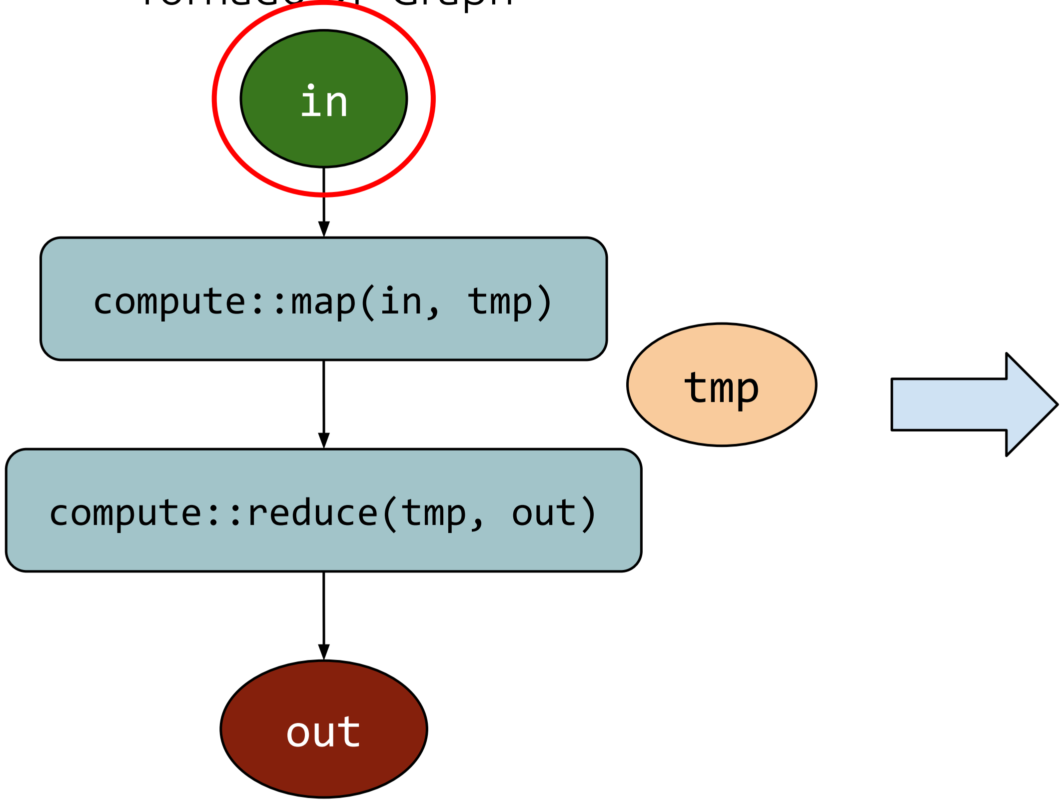


TornadoVM Bytecodes

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



END     <0>             // Ends context
```

TornadoVM Bytecodes - Example

Tornado DF-Graph

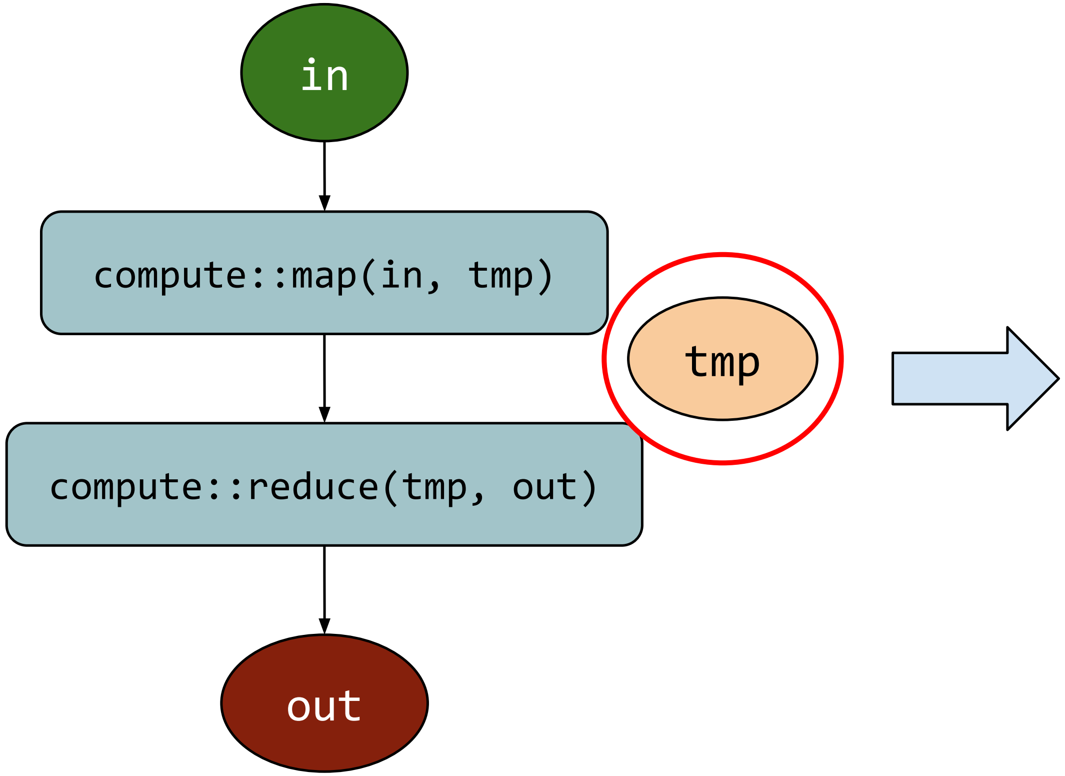


TornadoVM Bytecodes

```
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

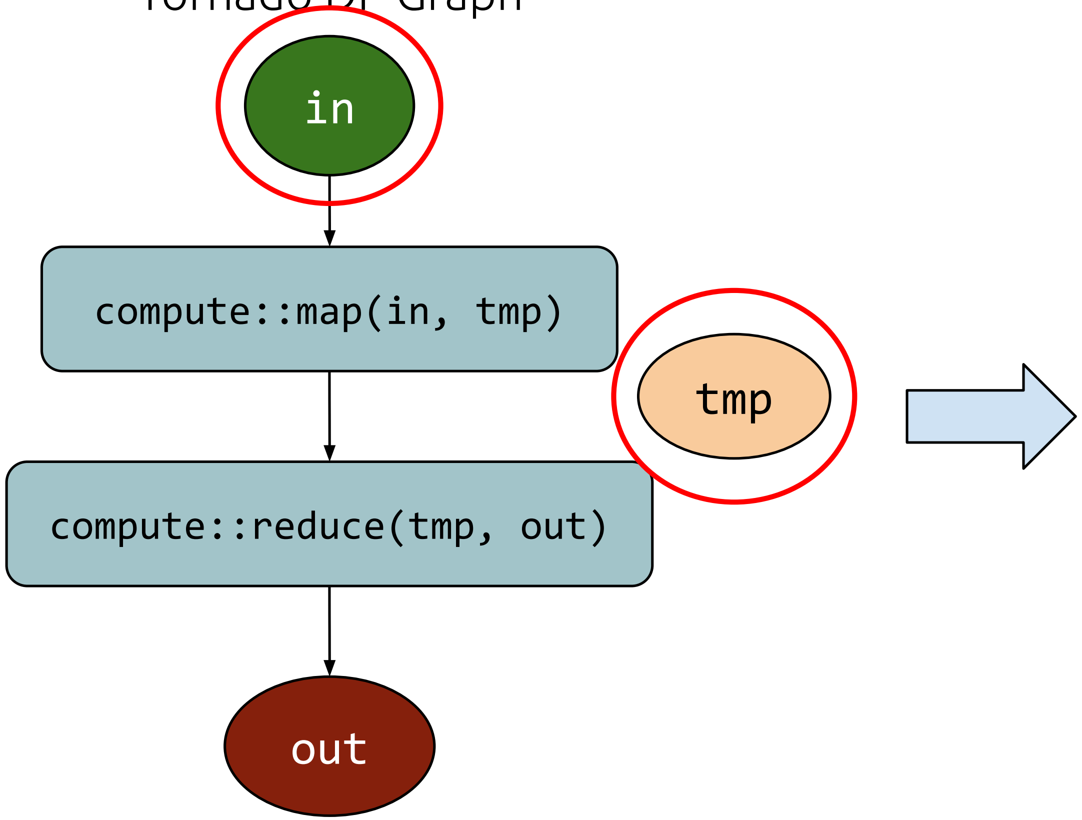


TornadoVM Bytecodes

```
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

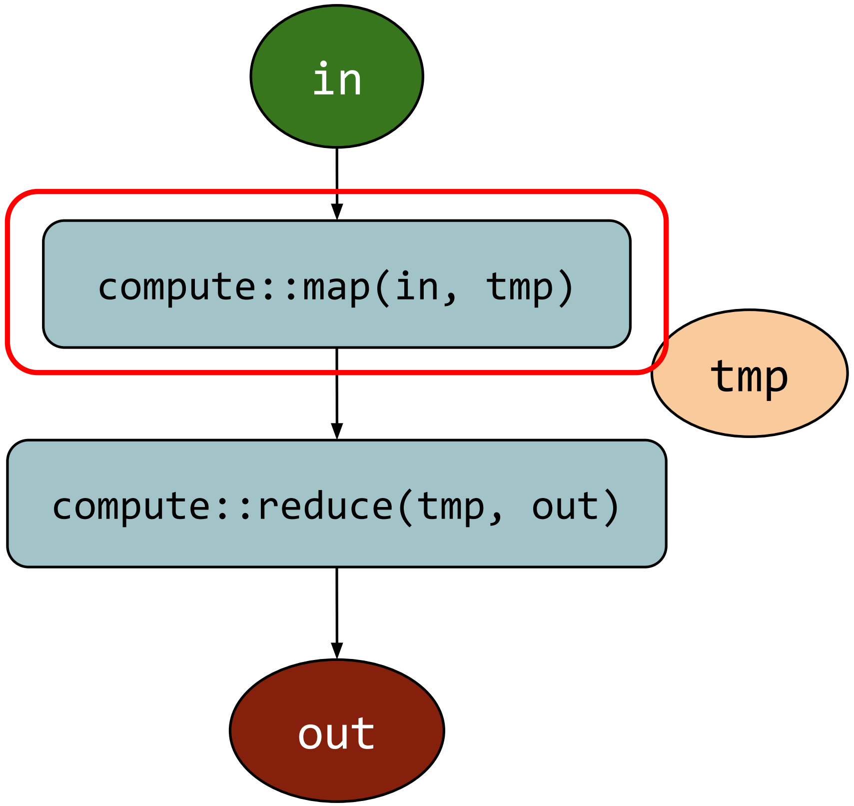


TornadoVM Bytecodes

```
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



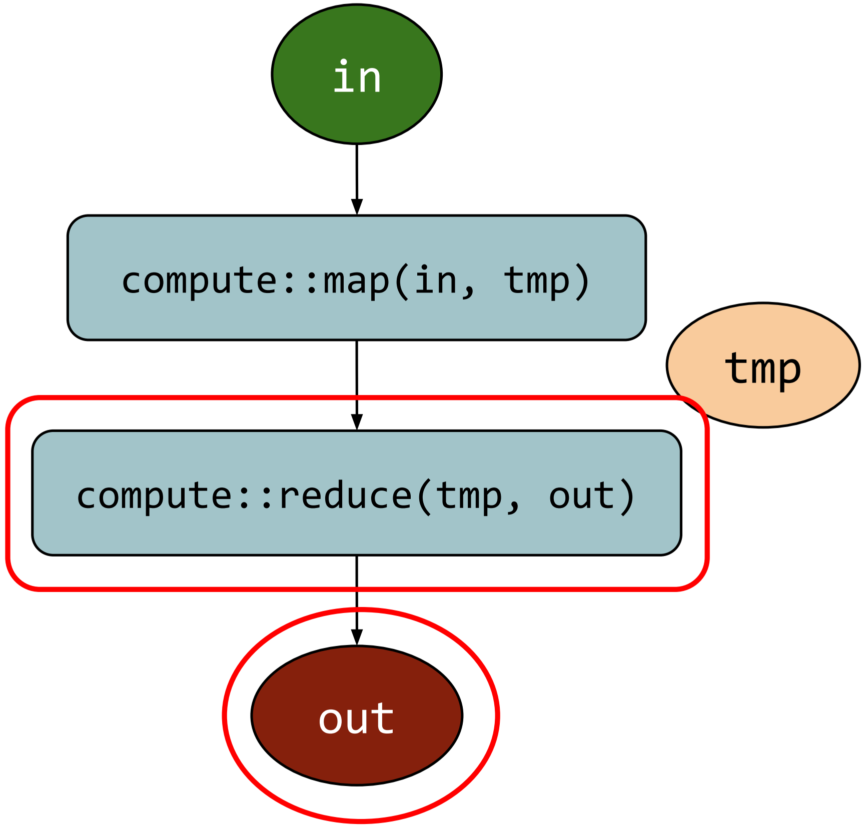
TornadoVM Bytecodes

```
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
```

MANCHESTER 1824
The University of Manchester

TornadoVM Bytecodes - Example

Tornado DF-Graph

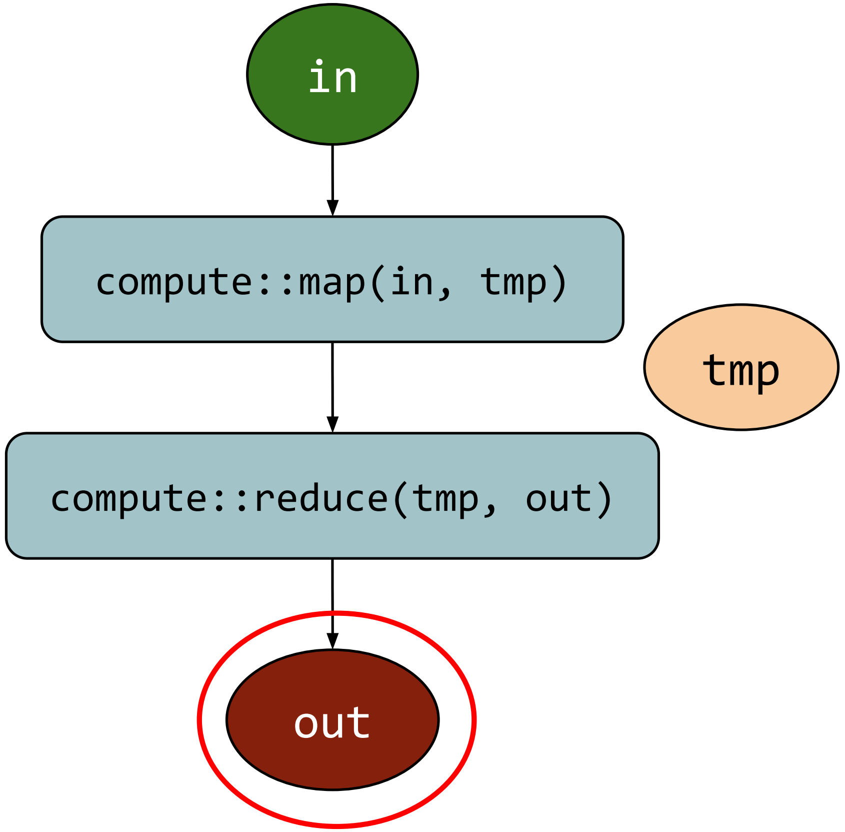


TornadoVM Bytecodes

```
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

The University of Manchester

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];
   }
  }
}
```

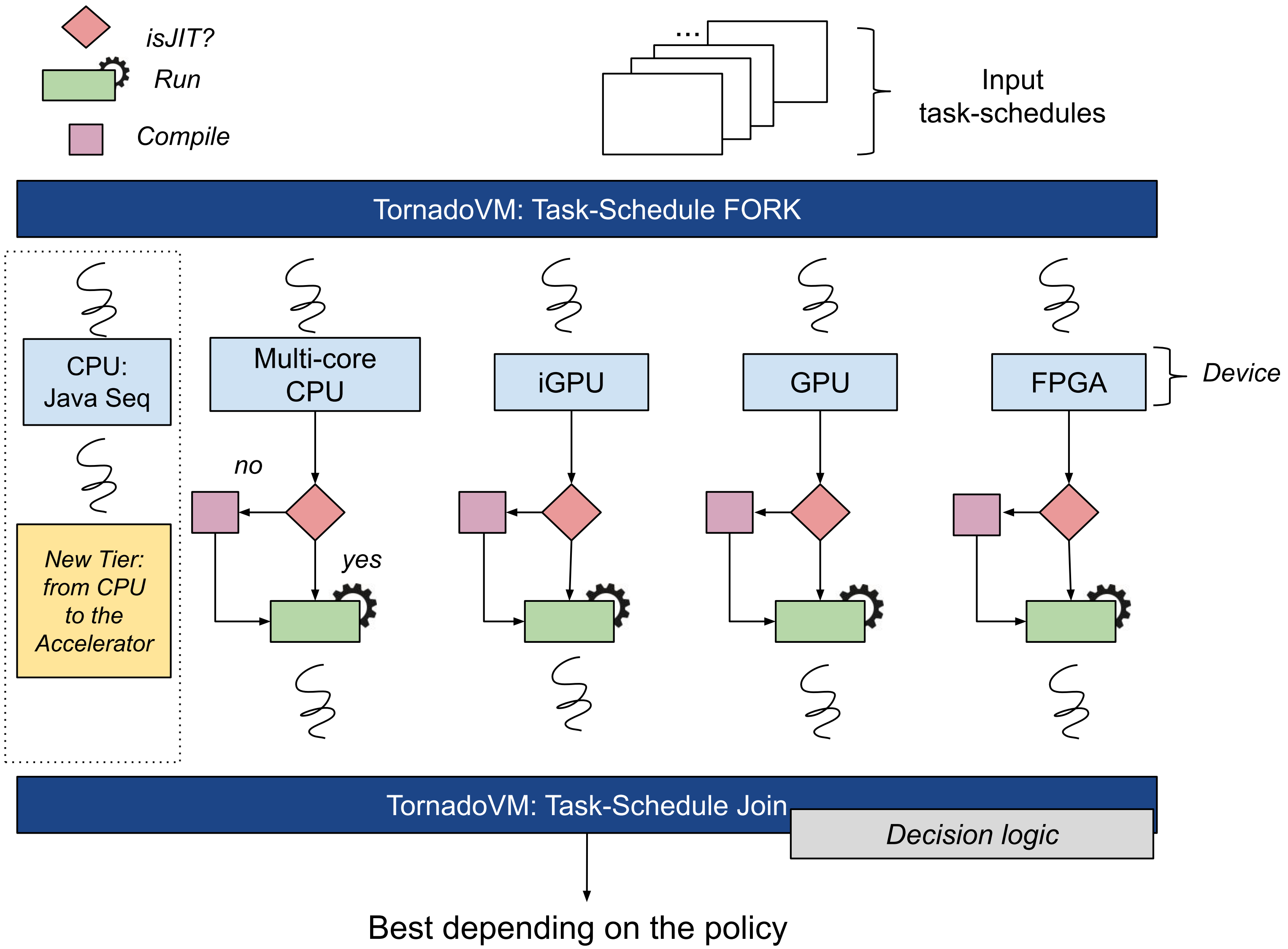
```
// 16GB data
double[] a = new double[2000000000];
double[] b = new double[2000000000];
double[] c = new double[2000000000];
TaskSchedule ts = new TaskSchedule("s0");

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

Tornado VM

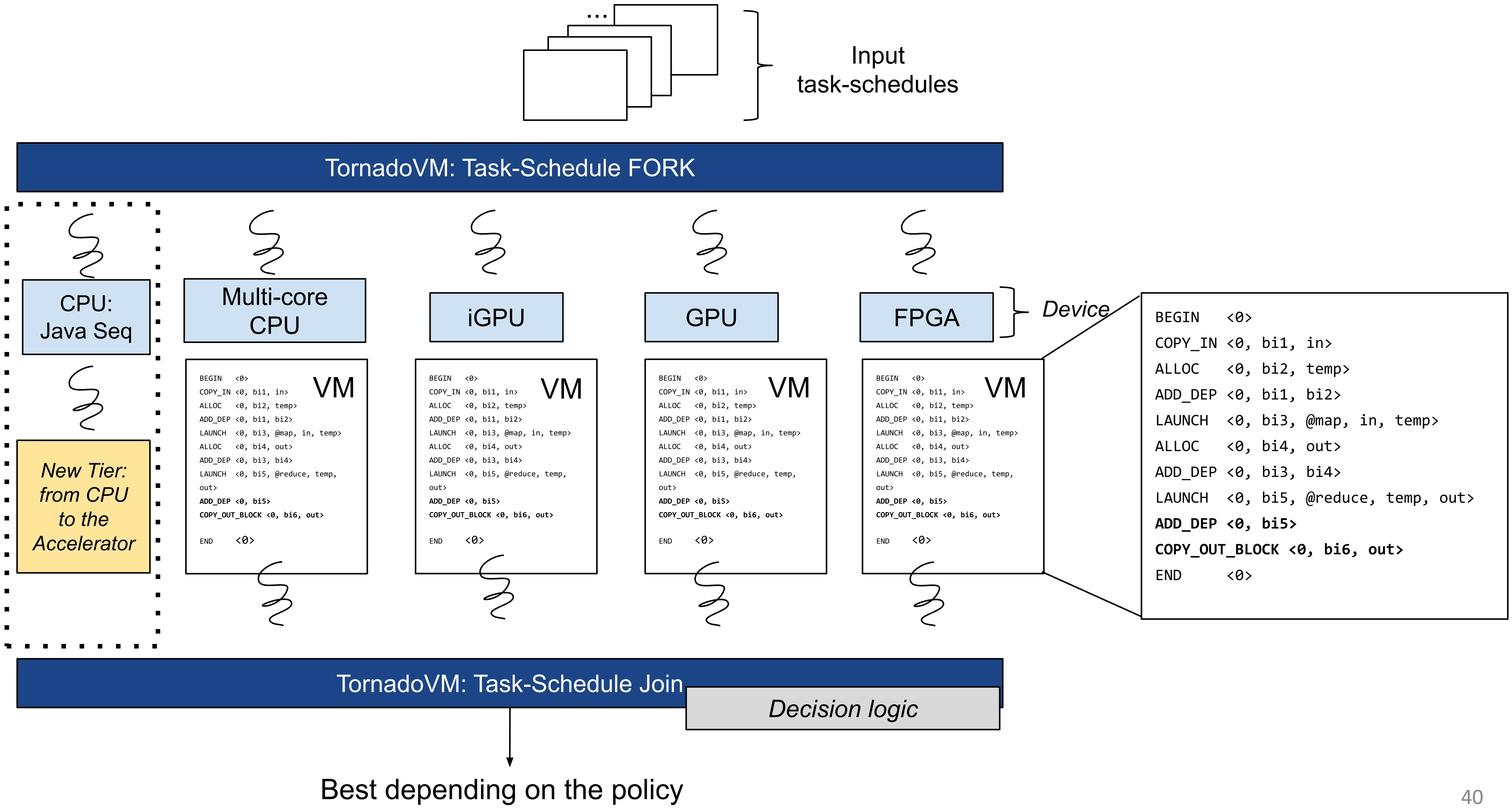
```
vm: BEGIN
vm: COPY_IN bytes=300000000, offset=0
vm: COPY_IN bytes=300000000, offset=0
vm: ALLOCATE bytes=300000000
vm: LAUNCH s0.t0 threads=37500000, offset=0
vm: STREAM_OUT bytes=300000000, offset=0
vm: COPY_IN bytes=300000000, offset=300000000
vm: COPY_IN bytes=300000000, offset=300000000
vm: ALLOCATE 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

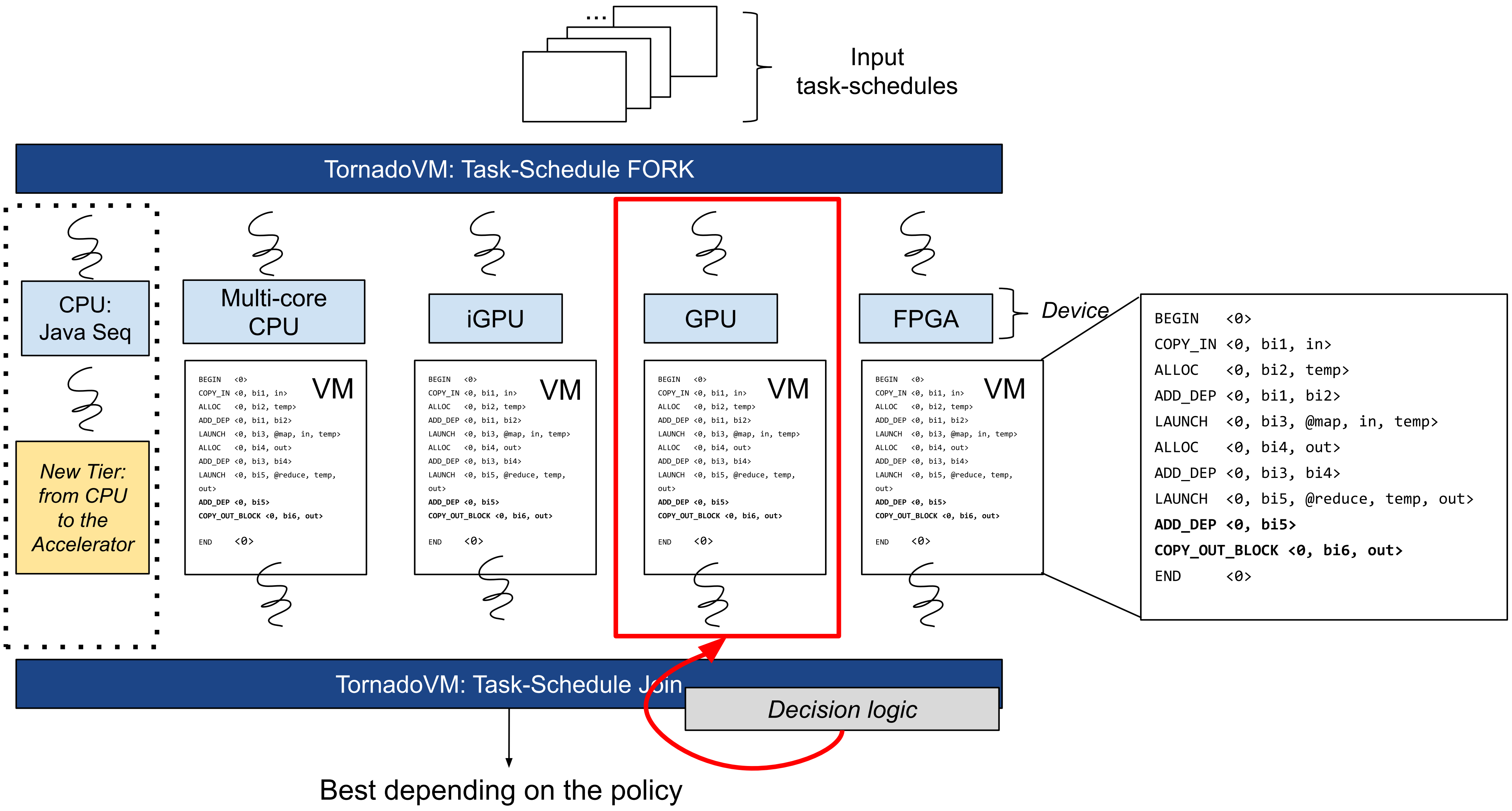
Dynamic Reconfiguration



Dynamic Reconfiguration



Dynamic Reconfiguration



How is the decision made?

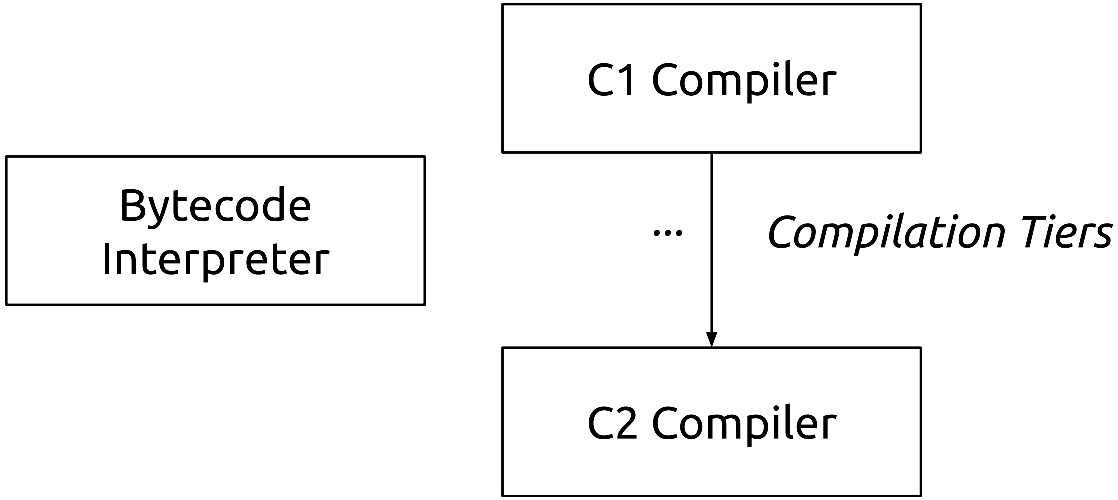
- End-to-end: including JIT compilation time
- Peak Performance: without JIT and after warming-up
- Latency: does not wait for all threads to finish

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

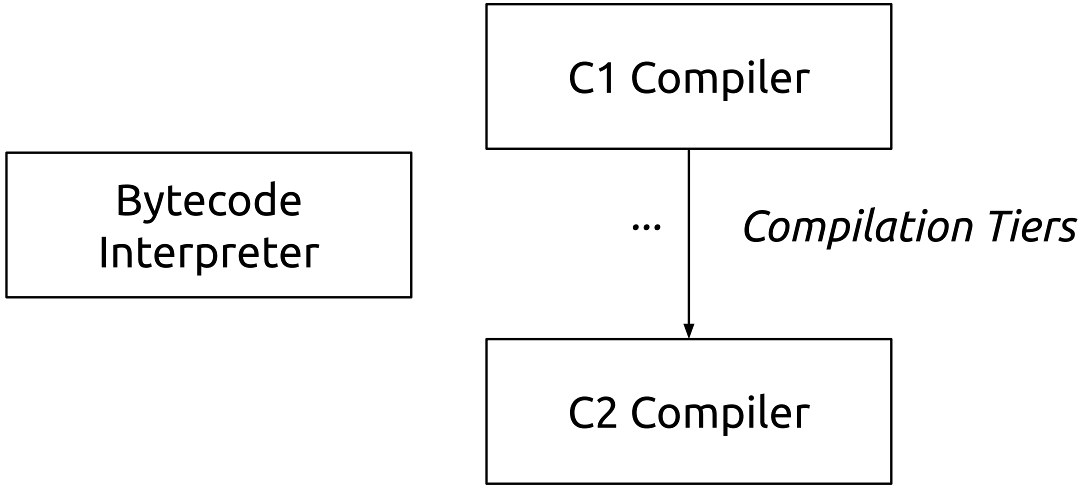
```
// PEAK PERFORMANCE
ts.task(Compute::add, a, b, c)
  .streamOut(c)
  .execute(Profiler.PERFORMANCE);
```

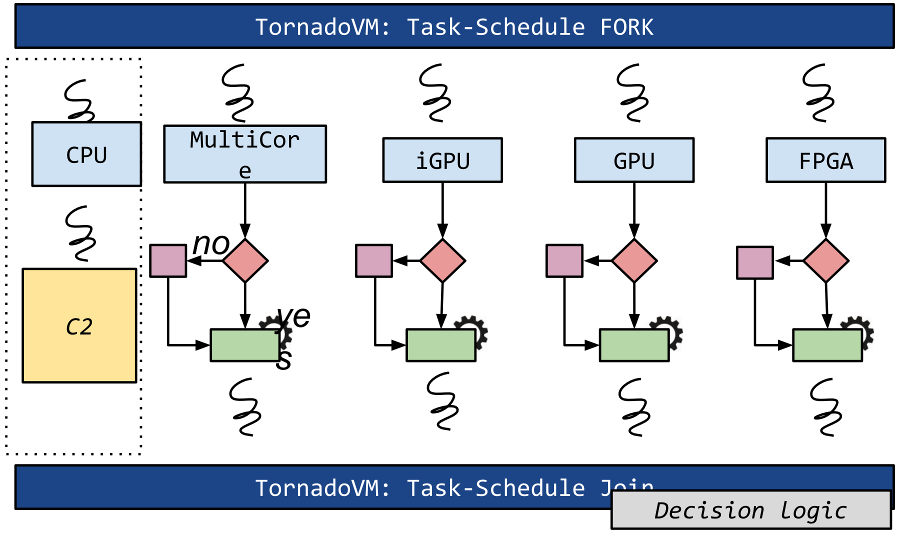
```
// Latency
ts.task(Compute::add, a, b, c)
  .streamOut(c)
  .execute(Profiler.LATENCY);
```

New compilation tier for Heterogeneous Systems



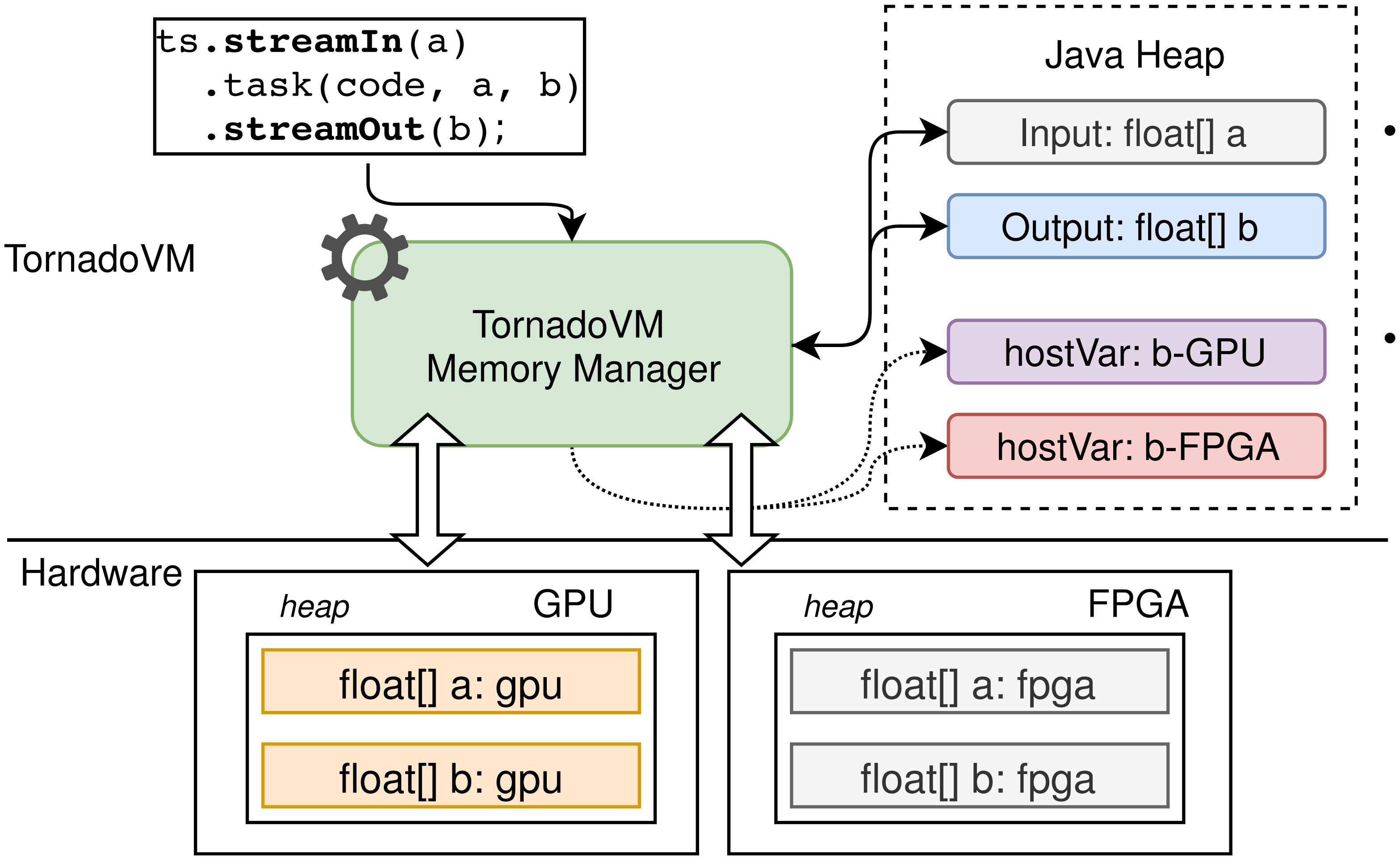
New compilation tier for Heterogeneous Systems





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

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

Related Work

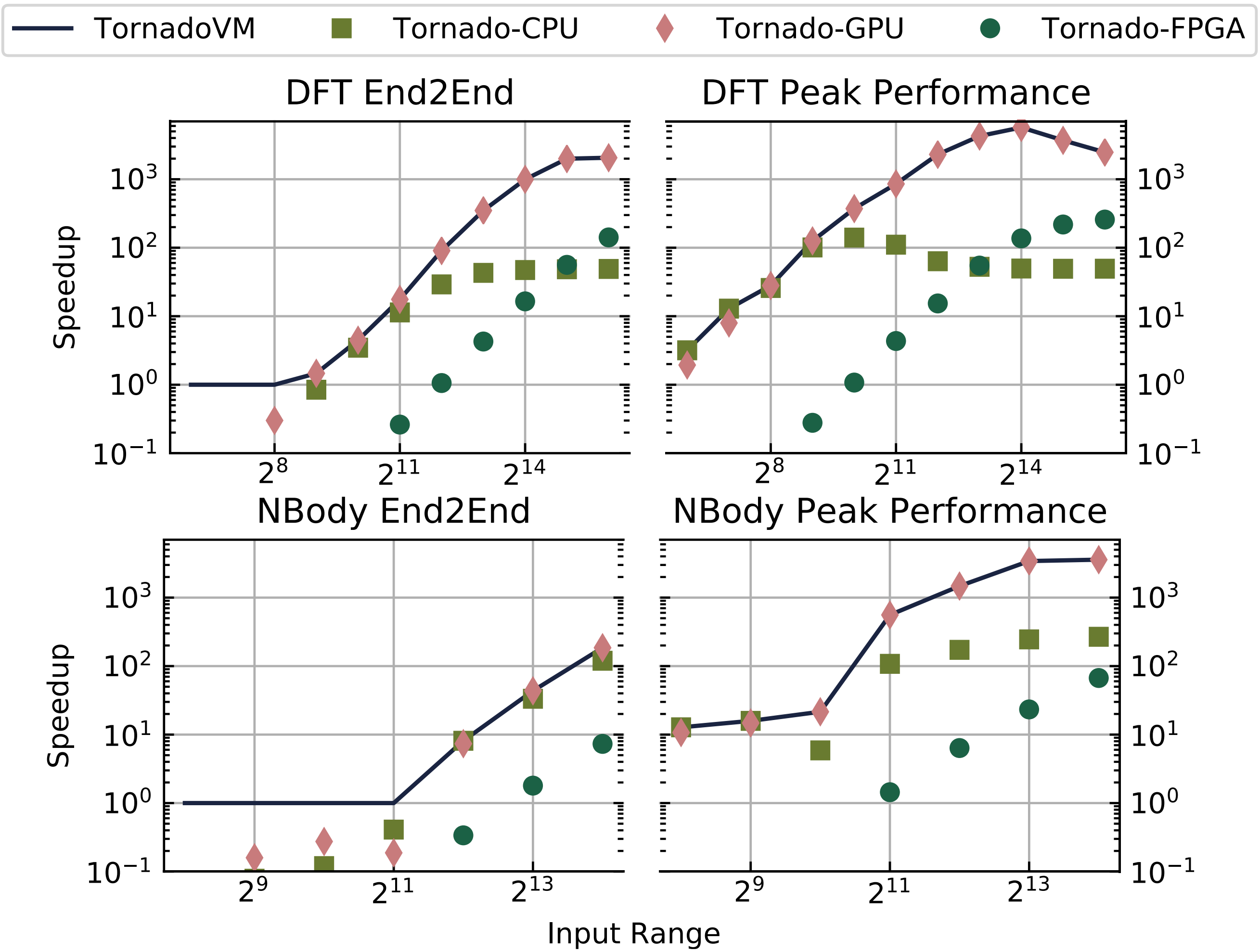
Related Work (in the Java context)

- 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
- Marawacc
  - It targets only GPUs/CPUs
  - Only map-style operation
  - It also targets R and Ruby!
- IBM GPU J9
  - Similar to Sumatra accelerating parallel Streams -> Targets only NVIDIA GPUs
  - No Dynamic Application Reconfiguration

TornadoVM supports more type of hardware & offloading only when it offers better performance

Ok, cool! What about performance?

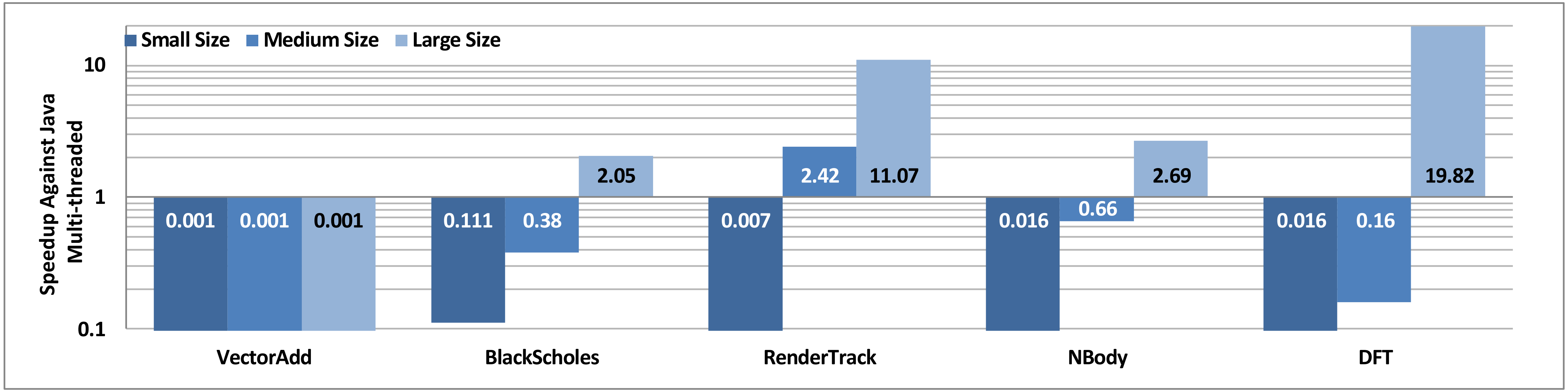
Performance



* 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: FPGA vs Multi-threading Java



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

* Slowdown for small sizes

MANCHESTER 1824
The University of Manchester

More details in our papers!

Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware

A Java Reduction Case Study

Juan Fumero
Advanced Processor Technologies Group
The University of Manchester
Manchester, M13 9PL, United Kingdom
juan.fumero@manchester.ac.uk

Christos Kotselidis
Advanced Processor Technologies Group
The University of Manchester
Manchester, M13 9PL, United Kingdom
christos.kotselidis@manchester.ac.uk

Abstract

Parallel skeletons are essential structured design patterns for efficient heterogeneous and parallel programming. They allow programmers to express common algorithms in such a way that it is much easier to read, maintain, debug and implement for different parallel programming models and parallel architectures. Reductions are one of the most common parallel skeletons. Many programming frameworks have been proposed for accelerating reduction operations on heterogeneous hardware. However, for the Java programming language, little work has been done for automatically compiling and exploiting reductions in Java applications on GPUs.

In this paper we present our work in progress in utilizing compiler snippets to express parallelism on heterogeneous hardware. In detail, we demonstrate the usage of Graal's snippets, in the context of the Tornado compiler, to express a set of Java reduction operations for GPU acceleration. The snippets are expressed in pure Java with OpenCL semantics, simplifying the JIT compiler optimizations and code generation. We showcase that with our technique we are able to execute a predefined set of reductions on GPUs within 85% of the performance of the native code and reach up to 20x over the Java sequential execution.

Reduction Case Study. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '18), November 4, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3281287.3281292

1 Introduction

Parallel programming skeletons such as map-reduce [8] and fork-join [17] have become essential tools for programmers to achieve higher performance of their applications, with ease in programmability. In particular, the map-reduce paradigm, since its conception, has been adopted by many applications that span from Big Data frameworks to desktop computing in various programming languages [21, 28, 32]. In addition, a number of such parallel skeletons have been combined to enable new usages as in the case of MR4J [3] that enables map-reduce operations in Java by employing the fork-join framework to achieve parallelism.

The introduction of heterogeneous hardware resources, such as GPUs and FPGAs into mainstream computing, creates new opportunities to increase the performance of such parallel skeletons. In the context of programming languages that have been designed specifically for heterogeneous programming like OpenCL [19], significant work has been done to implement high-performance reductions on GPUs lever-

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,

https://github.com/beehive-lab/TornadoVM/blob/master/assembly/src/docs/Publications.md

The University of Manchester

Limitations
&
Future Work

Limitations

We inherit limitations from the underlaying Prog. Model:

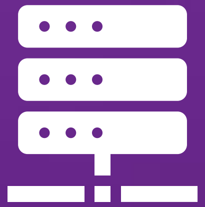
- No object support (except for a few cases)
- No recursion
- No dynamic memory allocation (*)
- No support for exceptions (*)

Future Work

The University of Manchester

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

Current Applicability of TornadoVM



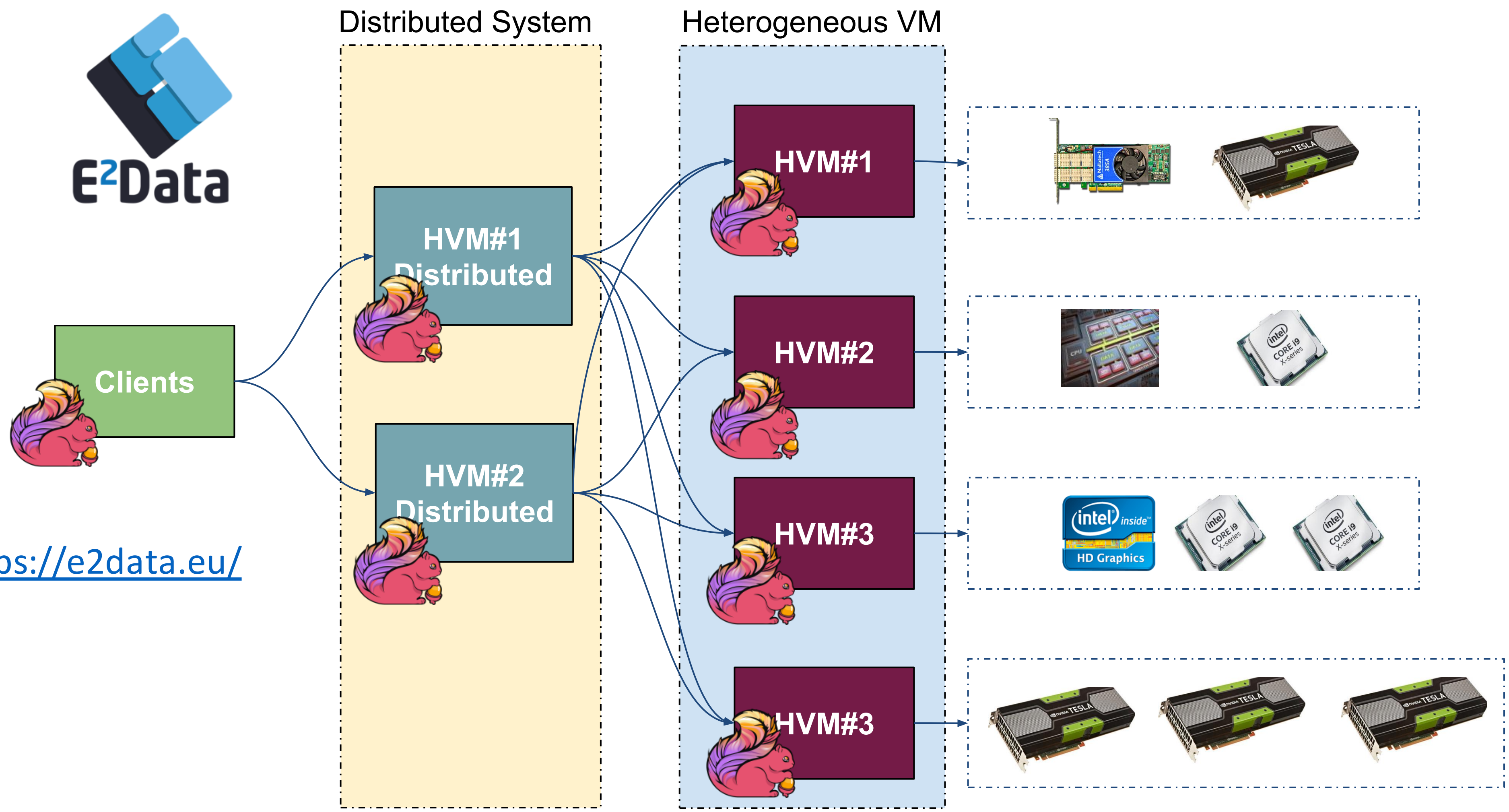
EU H2020 E2Data Project

https://e2data.eu/

"End-to-end solutions for 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/

How TornadoVM is currently being used in Industry?

Problem:
Many patents who had been discharged from a hospital are admitted again within a specific time interval.

Goal:
Improve the predictive capability of a hospital readmission by considering some features like the patent profile, characteristics, medical condition, etc.

Input
A data set that represents 10 years of clinical care at 130 US hospitals and integrated delivery networks.
- It includes over 50 features such as patent number, gender, age, admission type, …

Output:
- Predict if a patent will be readmitted or not after the hospitalization.

How TornadoVM is currently being used in Industry?

Using TornadoVM for the training phase (2M patients):
*** ~2615s --> 185s ! (14x)**

Problem:
Many patents who had been discharged from a hospital are admitted again within a specific time interval.

Goal:
Improve the predictive capability of a hospital readmission by considering some features like the patent profile, characteristics, medical condition, etc.

Input
A data set that represents 10 years of clinical care at 130 US hospitals and integrated delivery networks.

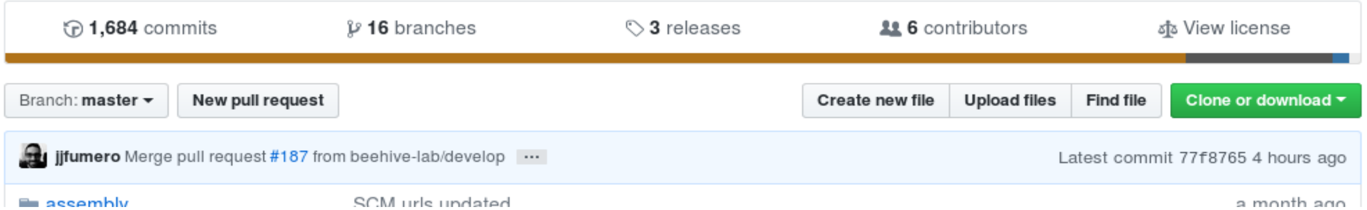
- It includes over 50 features such as patent number, gender, age, admission type, …

Output:
- Predict if a patent will be readmitted or not after the hospitalization.

Thanks to Gerald Mema from Exus for sharing the numbers and the use case

To sum up …

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
```

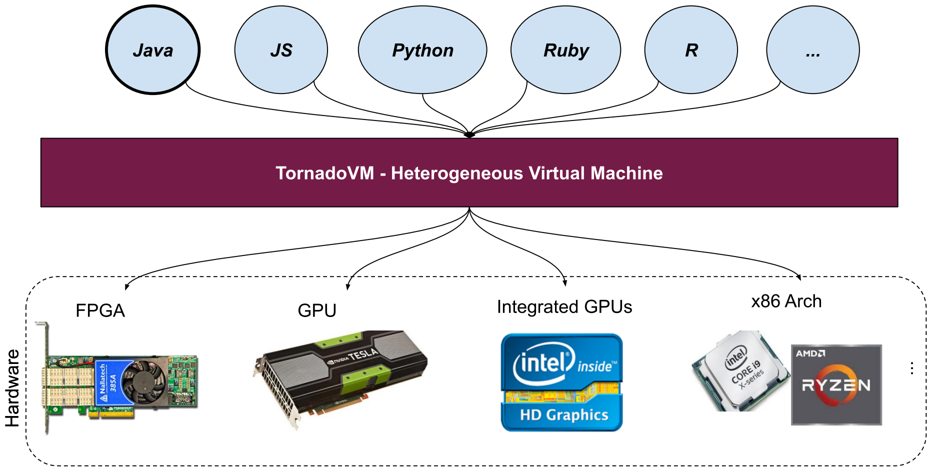
https://github.com/beehive-lab/docker-tornado

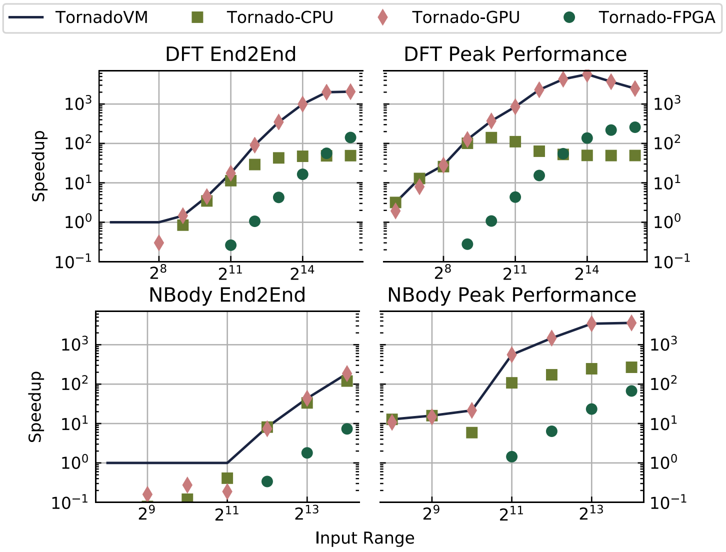
Team

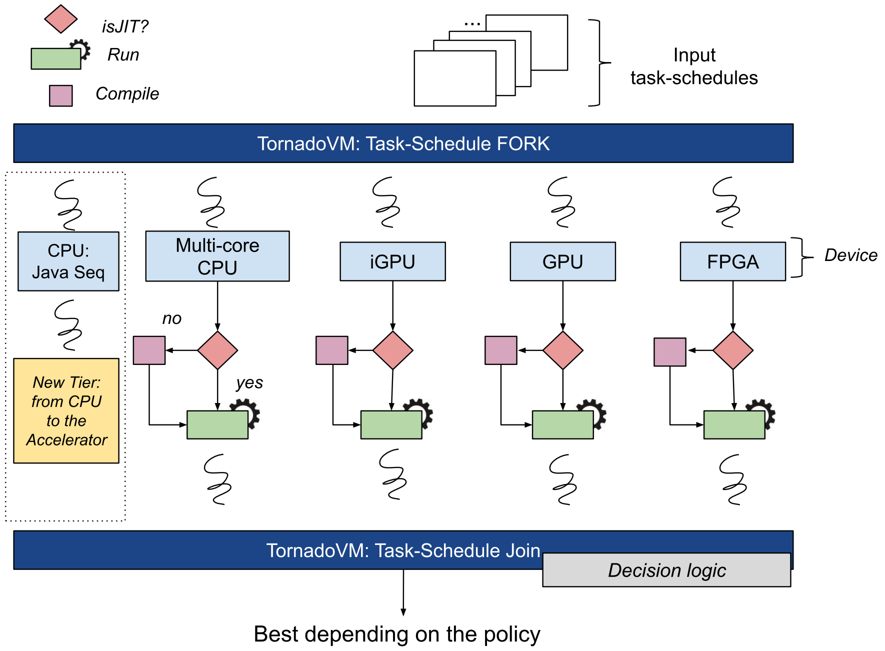
- Academic staff:
  - Christos Kotselidis
- Research staff:
  - Juan Fumero
  - Athanasios Stratikopoulos
  - Foivos Zakkak
  - Florin Blanaru
- Alumni:
  - James Clarkson
  - Benjamin Bell
  - Amad Aslam
- PhD Students:
  - Michail Papadimitriou
  - Maria Xekalaki
- Interns:
  - Undergraduates:
  - Gyorgy Rethy
  - Mihai-Christian Olteanu
  - Ian Vaughan

We are looking for collaborations (industrial & academics) -> Talk to us!

Takeaways







E²Data

https://e2data.eu

Thank you so much for your attention

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

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

Q&A

E2Data

MANCHESTER 1824

The University of Manchester

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

@snatverk

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

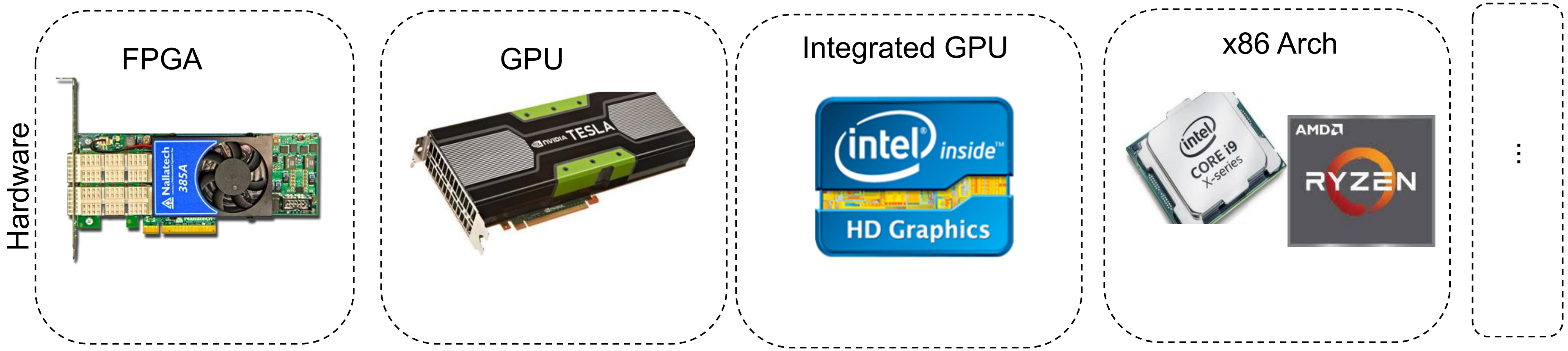
<juan.fumero@manchester.ac.uk>

Twitter: @snatverk

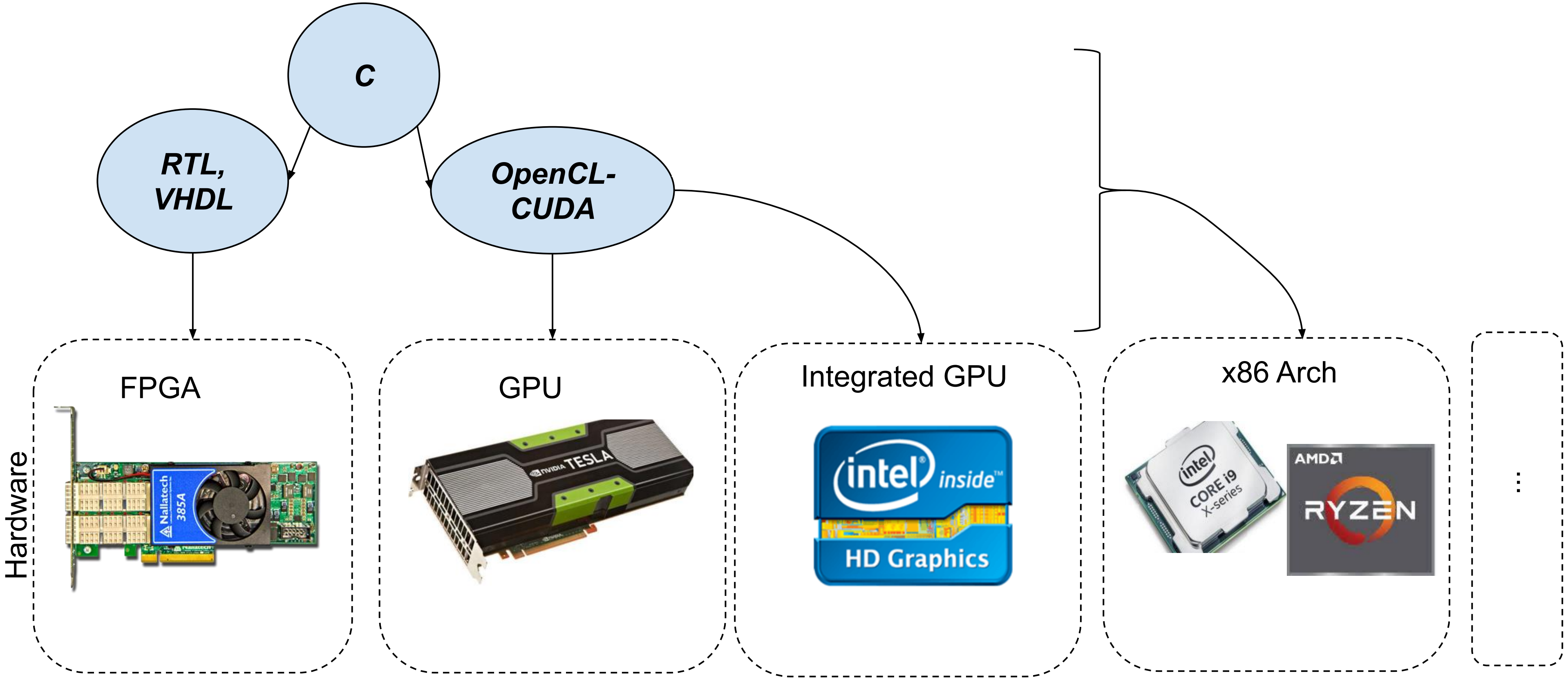
Joker<?> Conference 2019, Saint Petersburg October 26th

Back up slides

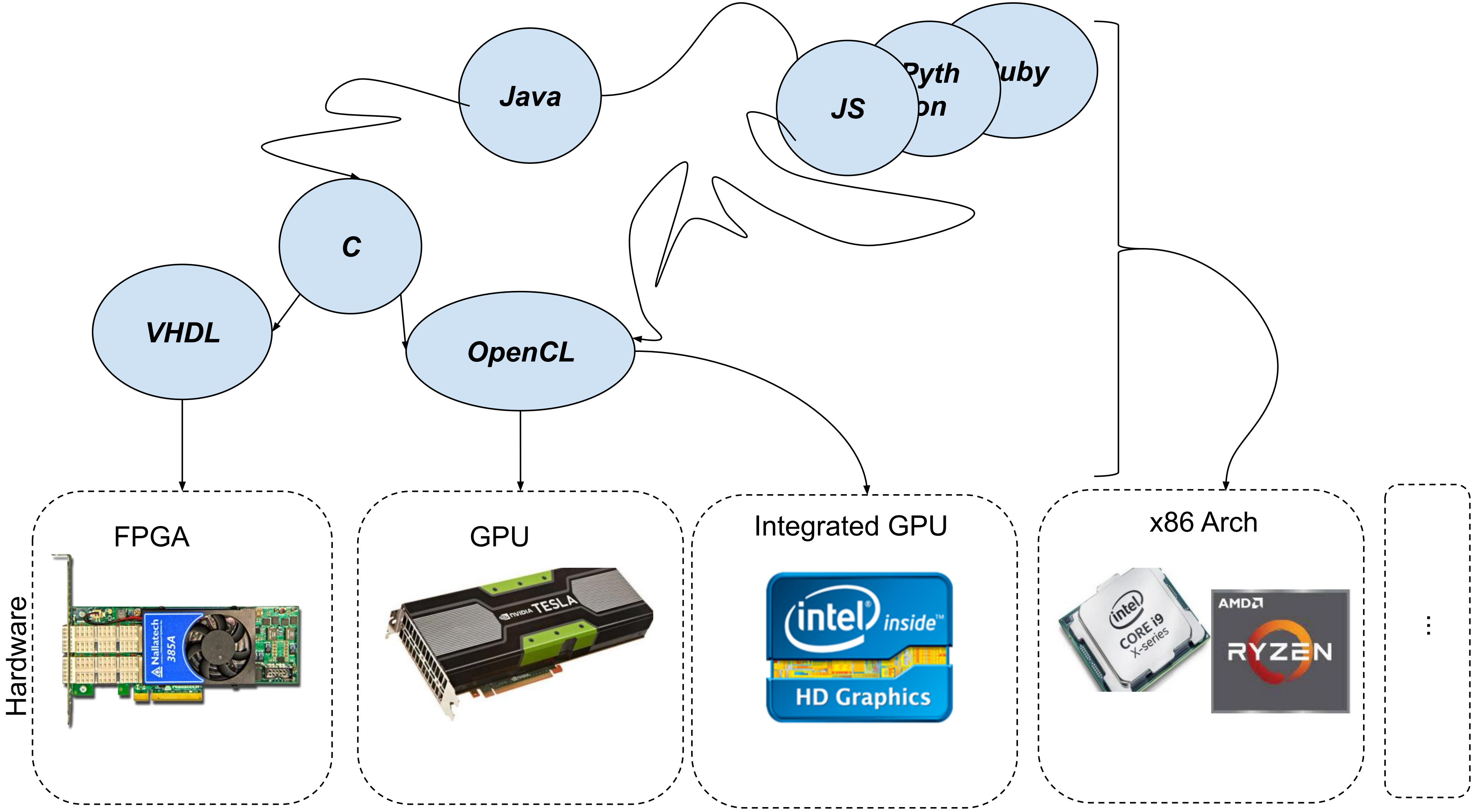
Current Computer Systems



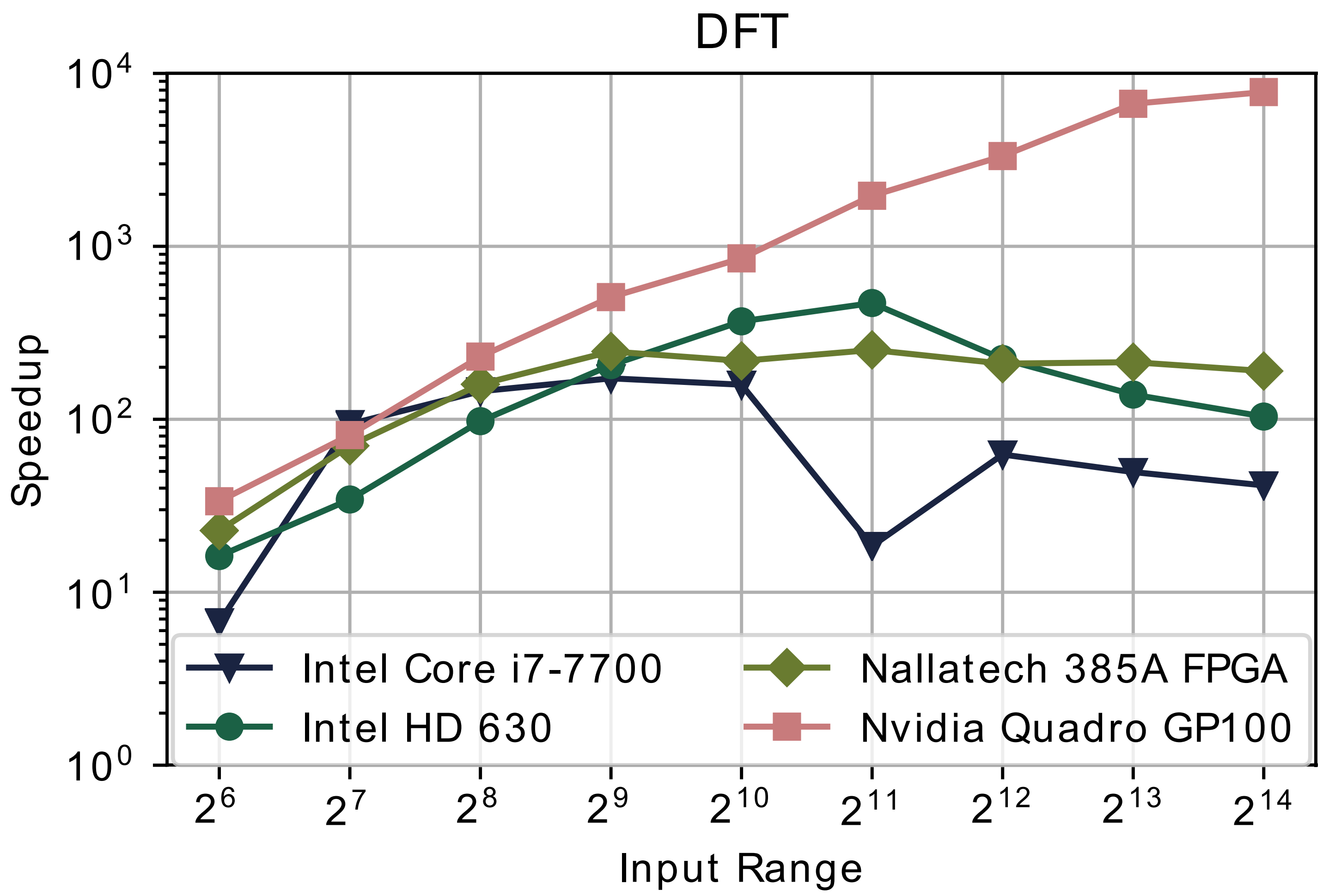
Current Computer Systems & Prog. Lang.



Current Computer Systems & Prog. Lang.



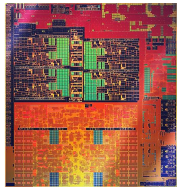
Still, why should we care about GPUs/FPGAs, etc?

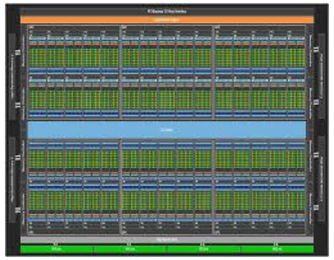


Performance for each device against Java hotspot:
* Up to 4500x by using a GPU
* 240x by using an FPGA

MANCHESTER 1824
The University of Manchester

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

How the OpenCL Generated Kernel looks like?

```
#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_bas

  // 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
    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

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

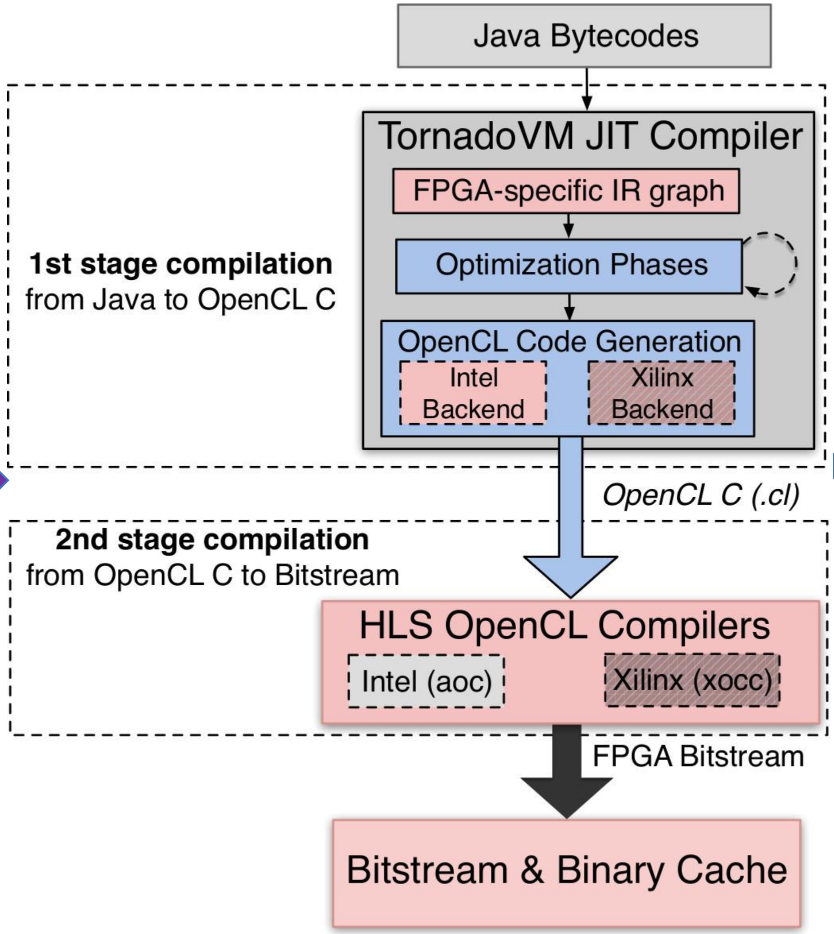
FPGA Support

Java

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

Program FPGAs within your favourite IDE: Eclipse, IntelliJ, …

TornadoVM



Physical hardware

MANCHESTER 1824
The University of Manchester

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

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 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>