


# Designing Parallel Programming APIs for Heterogenous Hardware on top of Managed Runtime Environments

Juan Fumero

Research Fellow & Intel Innovator

@snatverk



The University of Manchester

La Laguna, Tenerife

13th March 2024



**TORNADOVM**

# Outline

1. Motivation
2. Overview of Modern Parallel Programming Models
3. TornadoVM API for GPU/FPGA programming
4. New Proposal for future TornadoVM API
5. Conclusions

# Disclaimer

This presentation shows a set of thoughts/ideas collected during my short experience in parallel computing. Summary of discussions with many people including (our team, Oracle, Intel oneAPI/Intel Level-Zero teams, Aparapi founder)

This is not intended to be a tutorial and/or a complete guide.

We show a proposal that may (or may not) be fully integrated into the TornadoVM parallel programming framework



# Who am I?

Dr. Juan Fumero



Research Fellow @ University of Manchester

Architect and Developer of TornadoVM  
oneAPI **Intel Innovator**

- oneAPI Lang SIG

- oneAPI Hardware SIG



oneAPI

[juan.fumero@manchester.ac.uk](mailto:juan.fumero@manchester.ac.uk)

@snatverk

Former Member of:



PhD: Java, JIT  
Compilers for GPUs

Oracle Labs *Truffle and FastR Team*



Intel CilkPlus  
Vectorization  
Techniques for Root  
and GeantV

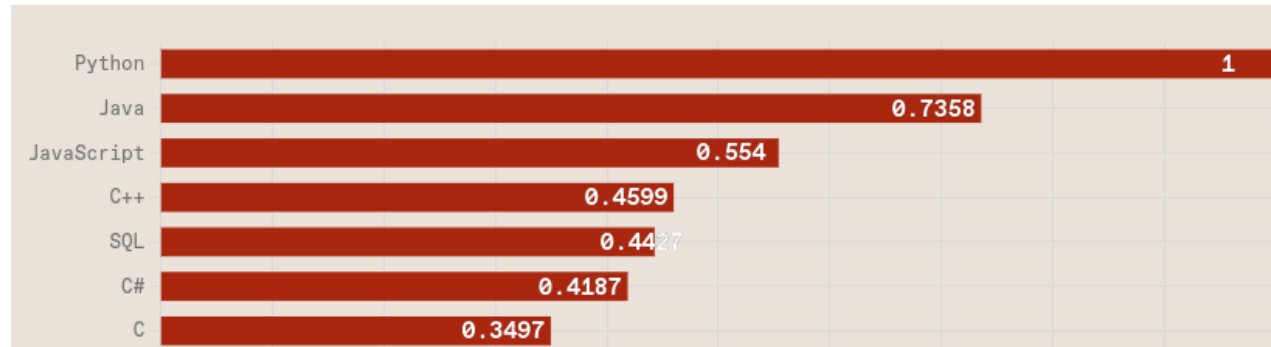


ULL Alumni

# Motivation

# Why Managed Runtime Systems (MRS)? Why Java?

- Java and MRS are one of the most popular [1]



- Java is the "king" in the Enterprise [2]
- Java is safe language and recommended by the NSA [3]

[1] <https://spectrum.ieee.org/the-top-programming-languages-2023>

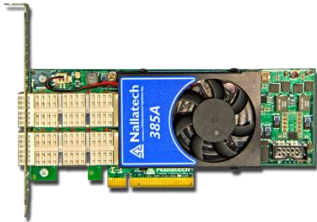
[2] <https://programmers.io/blog/why-java-is-king-in-enterprise-software/>

[3] [https://media.defense.gov/2023/Apr/27/2003210083/-1/-1/0/CSI\\_SOFTWARE\\_MEMORY\\_SAFETY\\_V1.1.PDF](https://media.defense.gov/2023/Apr/27/2003210083/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY_V1.1.PDF)

# Current Computer Systems

Hardware

FPGA



GPU



Integrated GPU

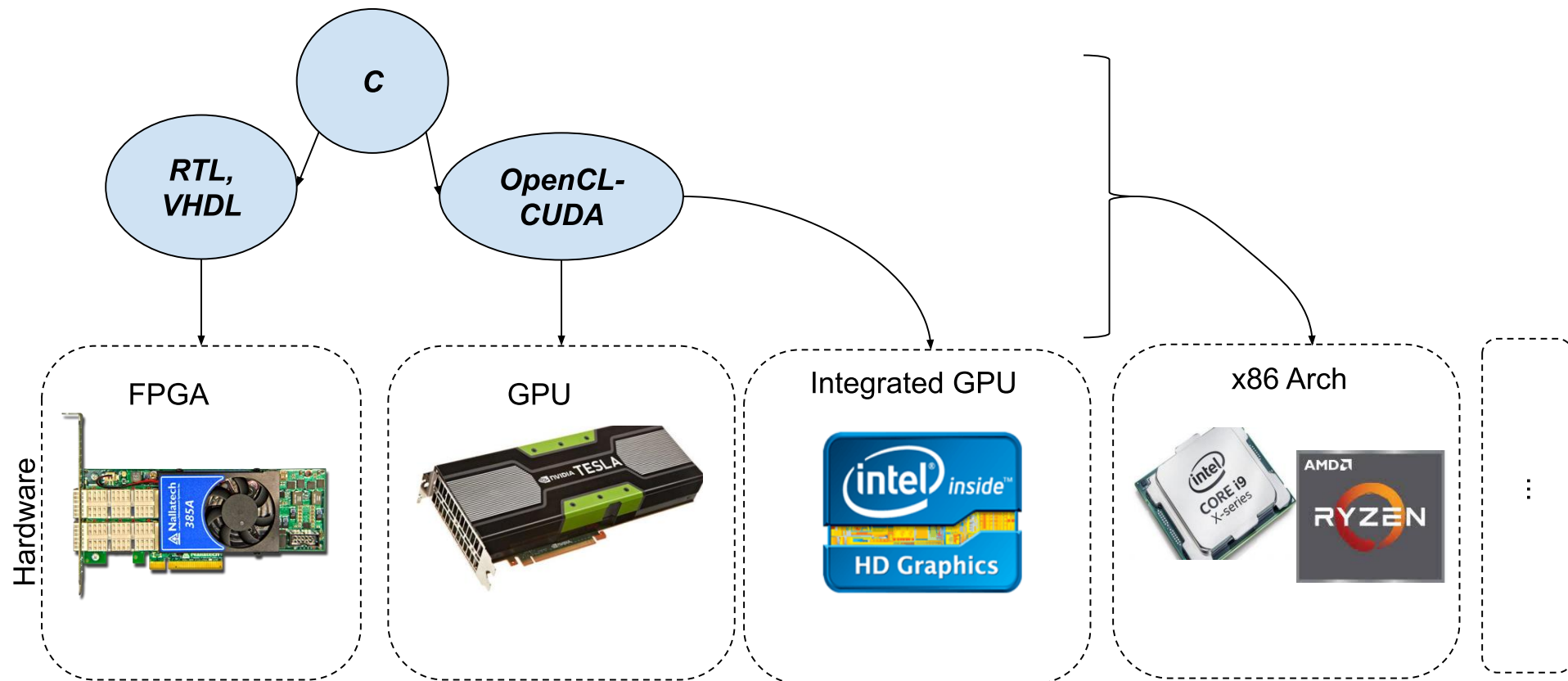


x86 Arch



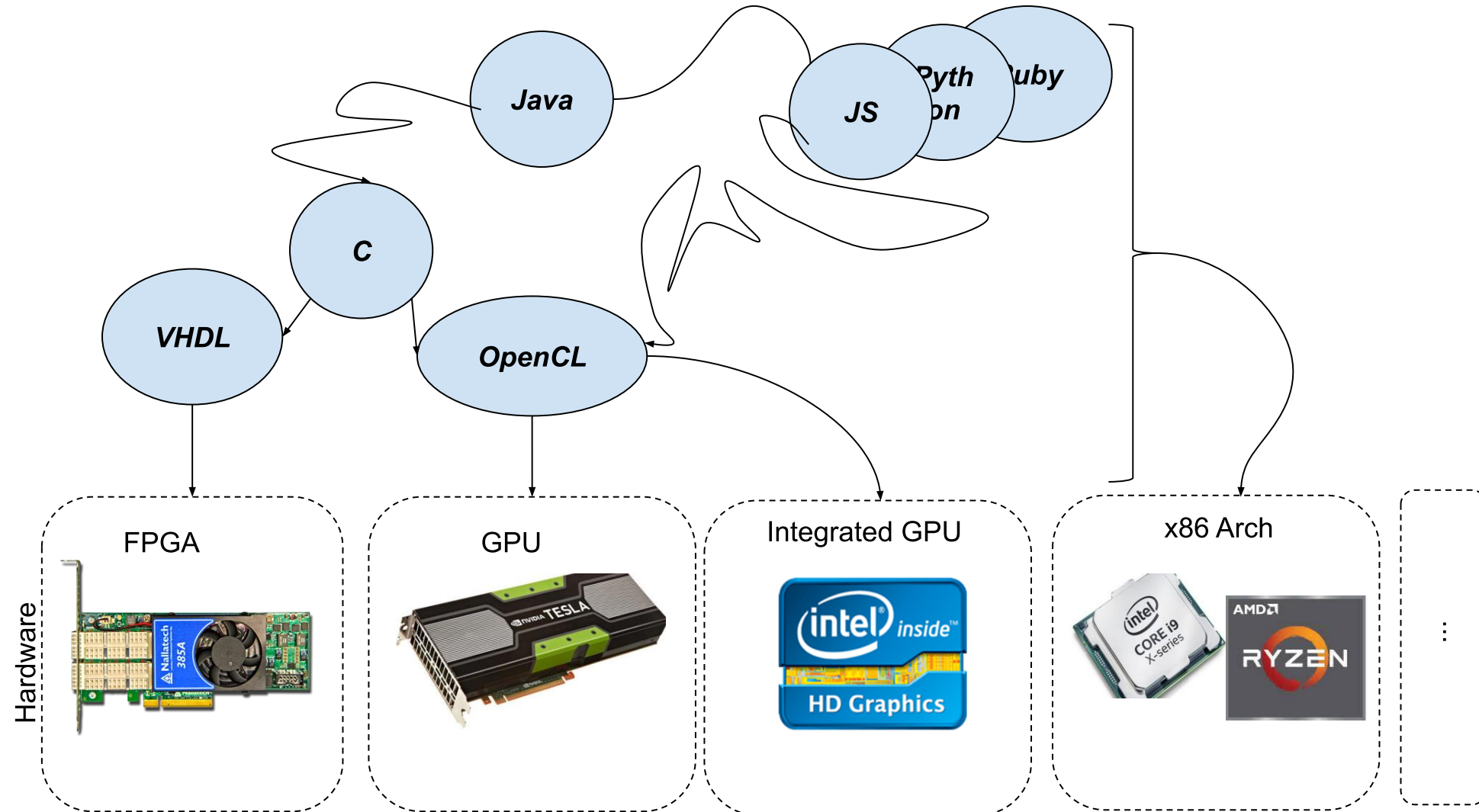
...

# Current Computer Systems

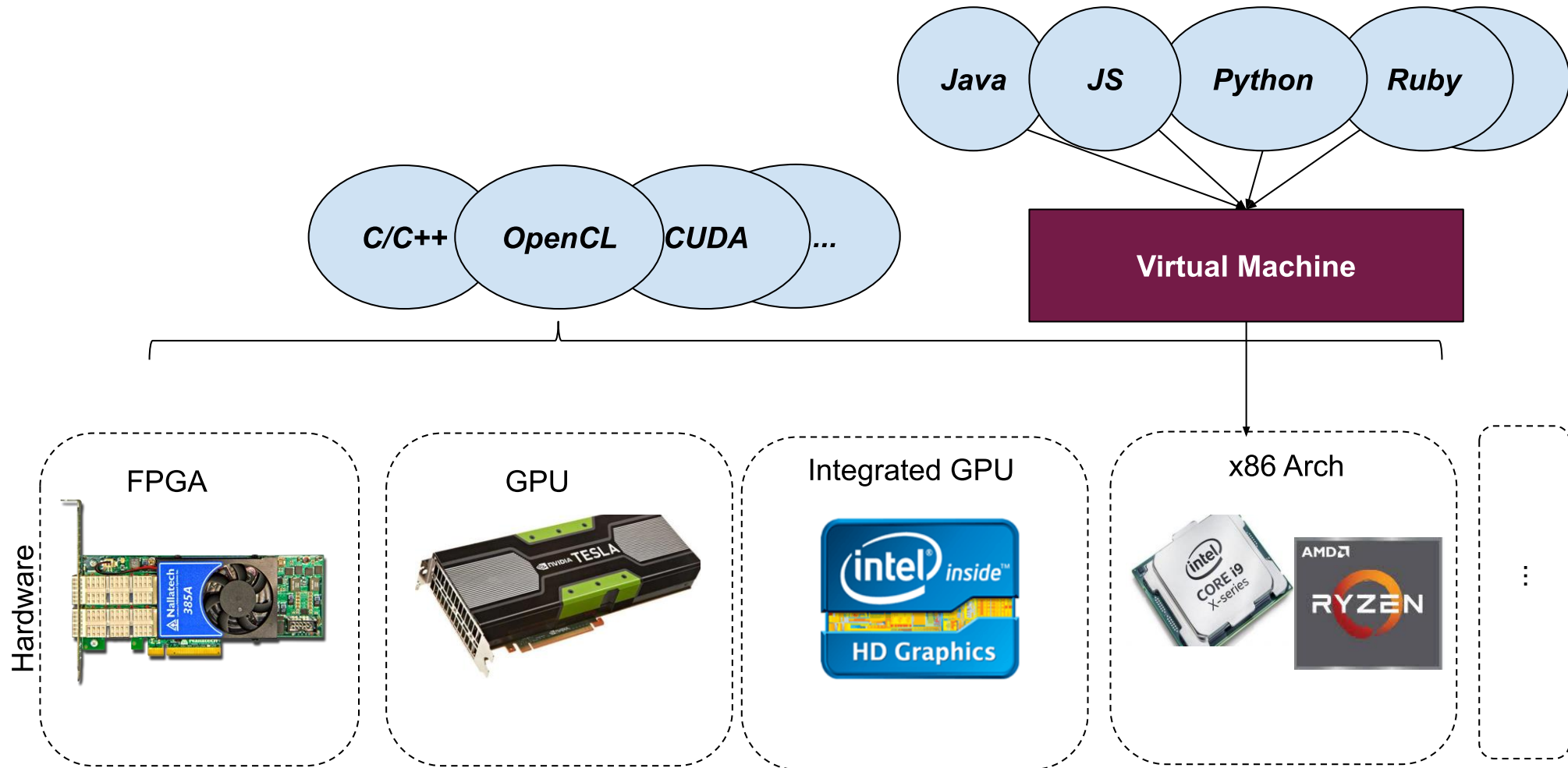




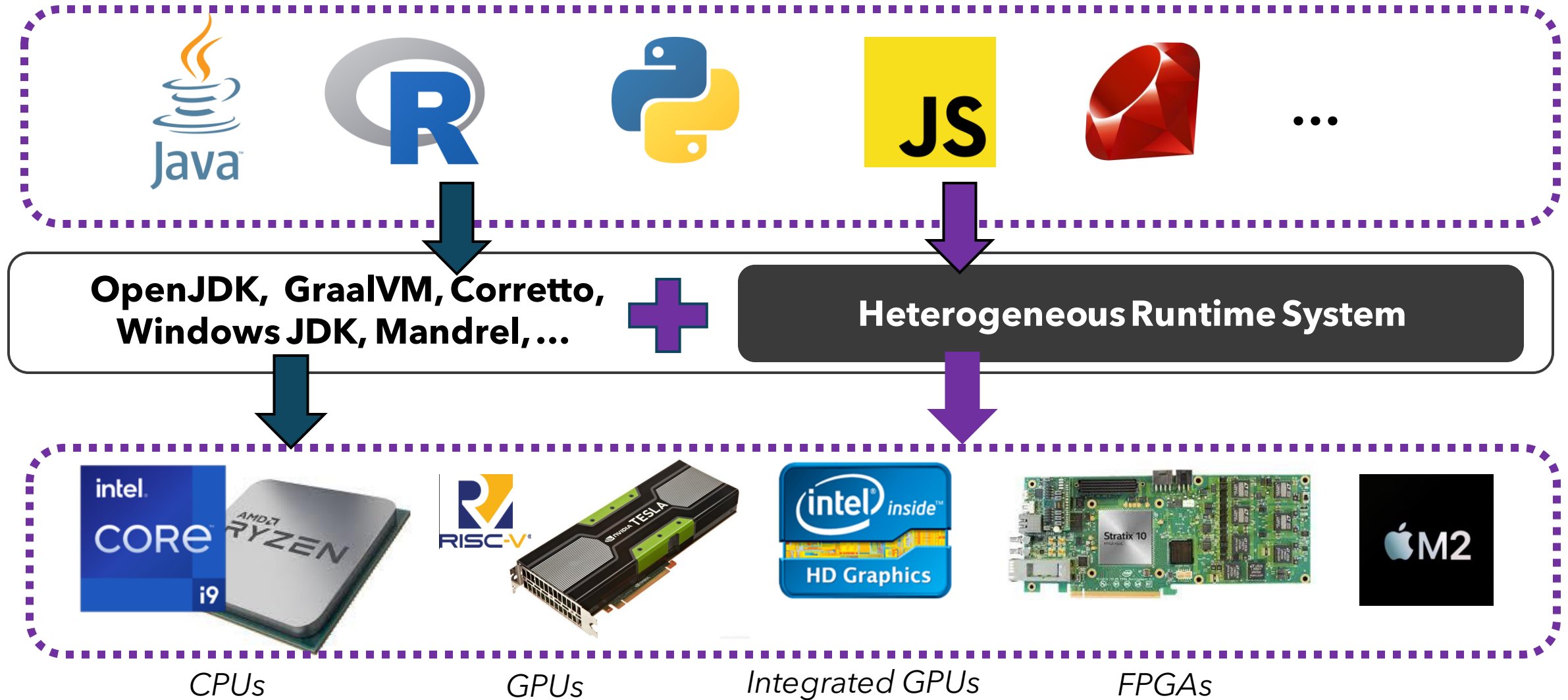
# Current Computer Systems



# Current Computer Systems



# But, what if?



# Hardware Characteristics and Parallelism

CPU



Optimized for  
**Latency**

- Task Parallelism
- Data Parallelism
- Pipeline Parallelism

GPU



Optimized for  
**throughput**

FPGAs



Optimized for **Latency**  
**and throughput**

# Hardware Characteristics and Parallelism

CPUs



Optimized for  
**Latency**

- Task Parallelism
- Data Parallelism
- Pipeline Parallelism

GPUs

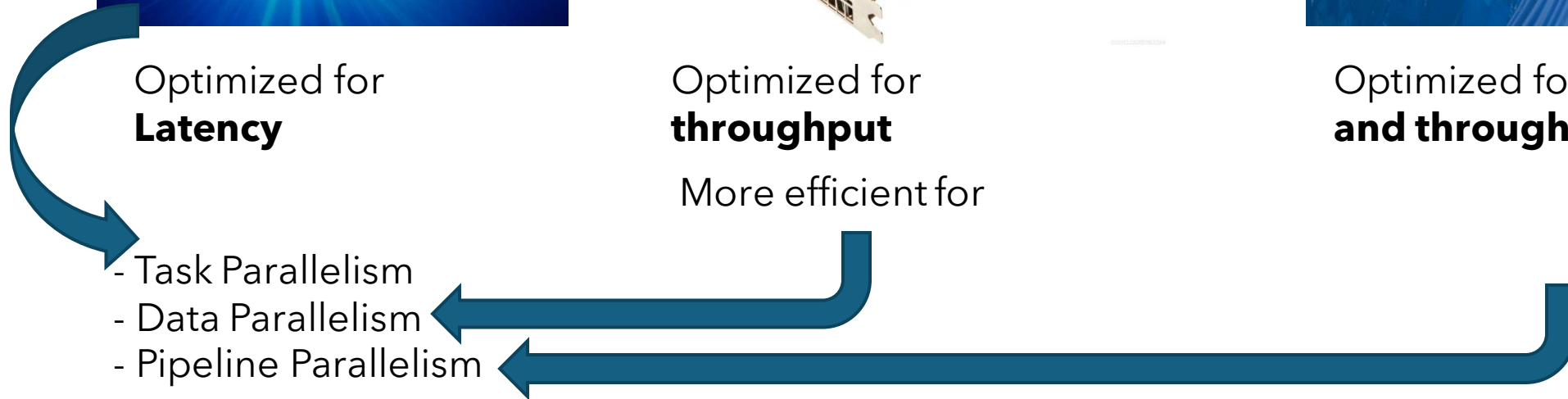


Optimized for  
**throughput**  
More efficient for

FPGAs



Optimized for **Latency**  
**and throughput**



What are we  
looking for?

# Looking for ...

How to express  
parallelism from  
high-level PL?

How to exploit  
different types of  
parallelism?

How to use  
memory  
efficiently?

How to  
dynamically  
launch new code?

How to express  
common patterns?

Opportunities for  
optimisation at the  
high-level. Should  
they be exposed?

# Desired Properties of a Parallel API

Performance



Portability



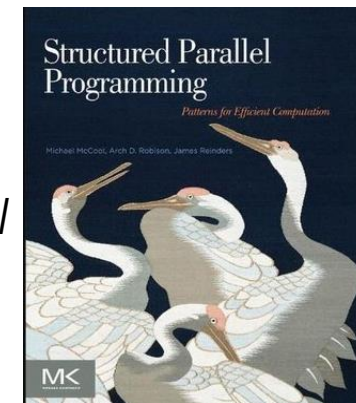
Productivity



Easy to adopt



Extensions of properties described in: "*Structured Parallel Programming*. Michael McCool, Arch Robison, James Reinders"





# Approaches - Classification

Explicit Parallel & Low-level Programming Models:

- E.g., CUDA, OpenCL

Directive-based **low**-level Heterogeneous Programming

- E.g., OpenACC, OpenMP > 4, OmpSs, PGI, HMPP, hiCUDA

Directive-based & **high**-level Heterogeneous Programming

- E.g., Copperhead, Numba JIT

Explicit Parallel PL:

- E.g., IBM Lime, Halide

Functional Parallel Programming Languages

- E.g., NVIDIA Nova, Futhark, Lift, etc

Java based:

- E.g., Sumatra, Aparapi, IBM J9, Marawacc, TornadoVM

# OpenCL & CUDA

Device  
Code

```
__kernel void saxpy(__global double *a, __global double *b,
                  __global double *c, const double alpha,
                  int iNumElements) {
    int idx = get_global_id(0);
    if (idx >= iNumElements) {
        return;
    }
    c[idx] = a[idx] * alpha + b[idx];
}
```

Host Code

```
int main() {
    openclInitialization();
    hostDataInitialization(elements);
    allocateBuffersOnGPU();
    clEnqueueWriteBuffer(queue, data, ...);
    clEnqueueNDRangeKernel(queue, kernel, ... )
    clEnqueueReadBuffer(queue, data, ...);
}
```

## Work-Flow in OpenCL

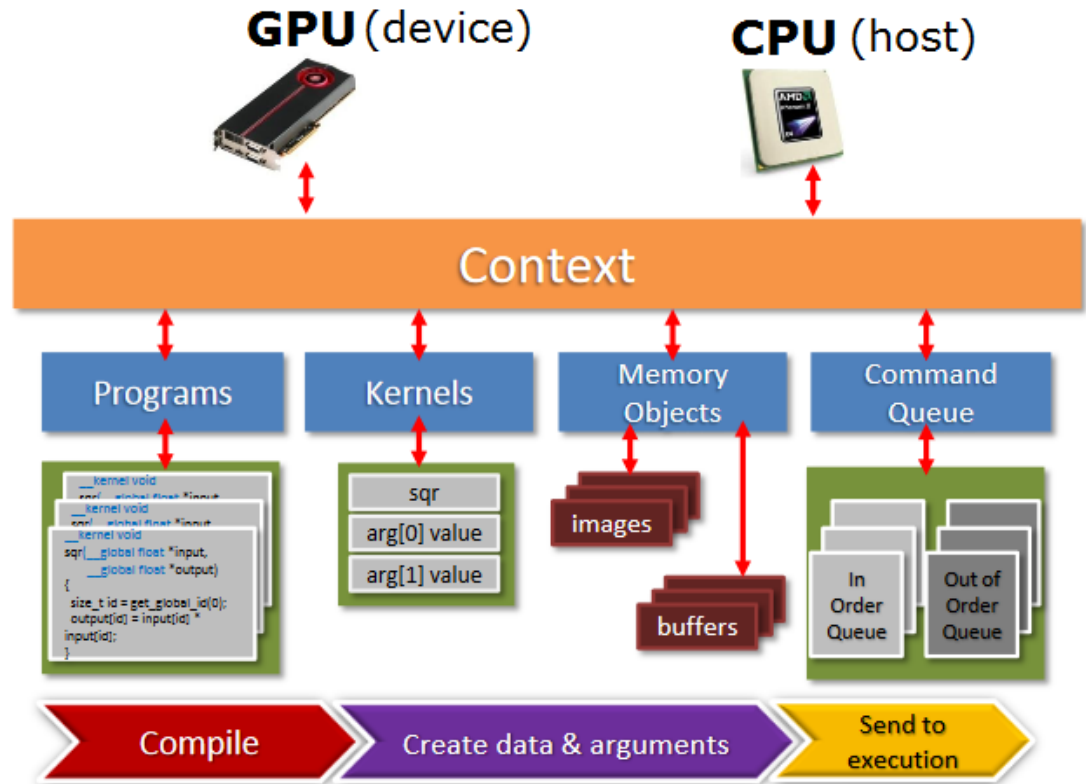


Image from "Introduction to OpenCL Programming– AMD 2010

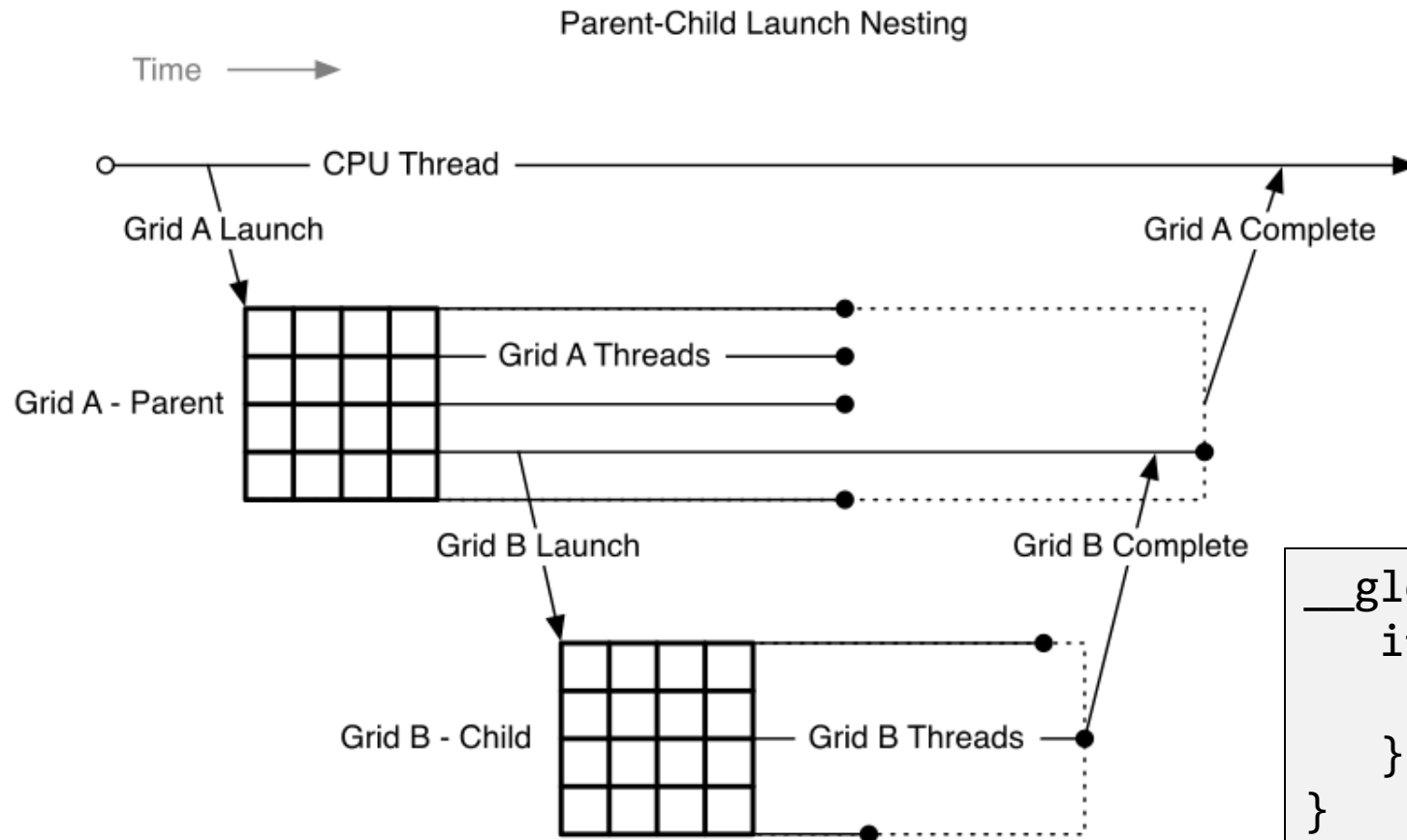
# OpenCL & CUDA

- Compute Kernels are expressed using Kernel Parallelism - It scales really well
- Very Low-Level Control - In fact: from the OpenCL Standard

"The target of OpenCL is expert programmers wanting to write portable yet efficient code. [...] Therefore, OpenCL provides a low-level hardware abstraction plus a framework to support programming, and many details of the underlying hardware are exposed."

- High-Performance
- Difficult to adopt
- Error-prone (separation of kernel and host code)
- Not very productive
- Code is portable, but performance is hard to make it portable

# OpenCL & CUDA: Dynamic Parallelism



This is a great feature to be exploited by higher level abstractions, and hard it to get right

Example in CUDA

```
__global__ void compute(float *data) {
    if (threadIdx.x == 0) {
        childKernel <<<1, 32>>> (data);
    }
}
```

*Image from NVIDIA docs*

# Annotation based (OpenACC, OpenMP)

## OpenACC

```
#pragma acc kernels copyin(a[0:n], b[0:n]) copyout(c[0:n])
for (int i = 0; i < N; i++) {
    c[i] = a[i] * b[i];
}
```

- **Loop Parallelism**
- Easy To Adopt
- Easy to learn
- Not performance portable & Hard to tune
- Productive



R. Reyes, Iván López, Juan Fumero and Francisco de Sande. ***A Preliminary Evaluation of OpenACC Implementations***. Journal of Supercomputing. September 2013,

## OpenMP

```
#pragma omp target data map(to:a[0:n], b[0:n]) map(from:c[0:n])
{
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < N; i++) {
        c[i] = a[i] * b[i];
    }
}
```

- **Loop Parallelism**
- Easy To Adopt
- NOT Easy to learn
- Not performance portable
- Productive

# Explosion of Annotations

OmpSs

```
#pragma omp target device(openc1) implements(matrix_multiply)
#pragma omp task in([BS]a,[BS]b) inout([BS]c)
__kernel void matrix_multiply_openc1(float a[BS][BS], float b[BS][BS],float c[BS][BS]);

#pragma omp target device(fpga,smp) copy_deps num_instances(3)
#pragma omp task in([BS]a,[BS]b) inout([BS]c)
void matrix_multiply(float a[BS][BS], float b[BS][BS], float c[BS][BS]) {
#pragma HLS inline
#pragma HLS array_partition variable=a block factor=BS/2 dim=2
#pragma HLS array_partition variable=b block factor=BS/2 dim=1
    for (int ia = 0; ia < BS; ++ia)
        for (int ib = 0; ib < BS; ++ib) {
            #pragma HLS PIPELINE II=1
            float sum = 0;
            for (int id = 0; id < BS; ++id)
                sum += a[ia][id] * b[id][ib];
            c[ia][ib] += sum;
        }
    }
    ...
}
```



## Heterogeneous Computing Architectures Challenges and Vision

Edited by: [Olivier Terzo](#) , [Karim Djemame](#) , [Alberto Scionti](#) , [Clara Pezuela](#)

# Explosion of Annotations

OmpSs

```
#pragma omp target device(openc1) implements(matrix_multiply)
#pragma omp task in([BS]a,[BS]b) inout([BS]c)
__kernel void matrix_multiply_openc1(float a[BS][BS], float b[BS][BS],float c[BS][BS]);

#pragma omp target device(fpga,smp) copy_deps num_instances(3)
#pragma omp task in([BS]a,[BS]b) inout([BS]c)
void matrix_multiply(float a[BS][BS], float b[BS][BS], float c[BS][BS]) {
#pragma HLS inline
#pragma HLS array_partition variable=a block factor=BS/2 dim=2
#pragma HLS array_partition variable=b block factor=BS/2 dim=1
    for (int ia = 0; ia < BS; ++ia)
        for (int ib = 0; ib < BS; ++ib) {
            #pragma HLS PIPELINE II=1
            float sum = 0;
            for (int id = 0; id < BS; ++id)
                sum += a[ia][id] * b[id][ib];
            c[ia][ib] += sum;
        }
    }
    ...
}
```

**We want to avoid many annotations**



**Heterogeneous Computing Architectures  
Challenges and Vision**

Edited by: [Olivier Terzo](#) , [Karim Djemame](#) , [Alberto Scionti](#) , [Clara Pezuela](#)

# Single Source Property - E.g., SYCL and Intel oneAPI

```
#include <CL/sycl.hpp>
#include <array>
#include <iostream>
using namespace sycl;

int main() {
    constexpr int size=16;
    std::array<int, size> data;


    // Create queue on implementation-chosen default device
    queue Q;

    // Create buffer using host allocated "data" array
    buffer B { data };

    Q.submit([& (handler& h) {
        accessor A{B, h};
        h.parallel_for(size, [=](auto& idx) {
            A[idx] = idx;
        });
    });

    // Obtain access to buffer on the host
    // Will wait for device kernel to execute to generate data
    host_accessor A{B};
    for (int i = 0; i < size; i++)
        std::cout << "data[" << i << "] = " << A[i] << "\n";

    return 0;
}
```



Code from "Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL. Reinders", J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., Tian, X.

SYCL: Standard for heterogeneous compute on top of OpenCL (and CUDA).

- \* Uses a single source property (We want this)
- \* Use of lambdas
- \* Good performance (kernel parallelism)
- \* Hard to tune (ranges and local work groups)
- \* Increase productivity
- \* Not easy to adopt\* (requires extensive knowledge)
  - Explicit use of queues



# Skeletons and Functional

## NOVA – Data Parallel Language

```
(let (inc (lambda (x:int):int(+ x 1)))  
  in (inc(+ a b)))
```



Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. 2014. NOVA: A Functional Language for Data Parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*.

- Skeletons are easy to understand, but it might be difficult to adopt if the code requires too many changes, or it is a new language
- High-performance from high-level lang. Is hard to achieve
- More productive approaches

## JPAI – Java Parallel Array Interface

```
ArrayFunction<Tuple2<Integer, Integer>, Double>  
computation = new Map<>  
  (vectors -> {  
    Return 2.5 * vectors._1()  
      + vectors._2() );
```



Juan Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. Runtime Code Generation and Data Management for Heterogeneous Computing in Java (PPPJ15)

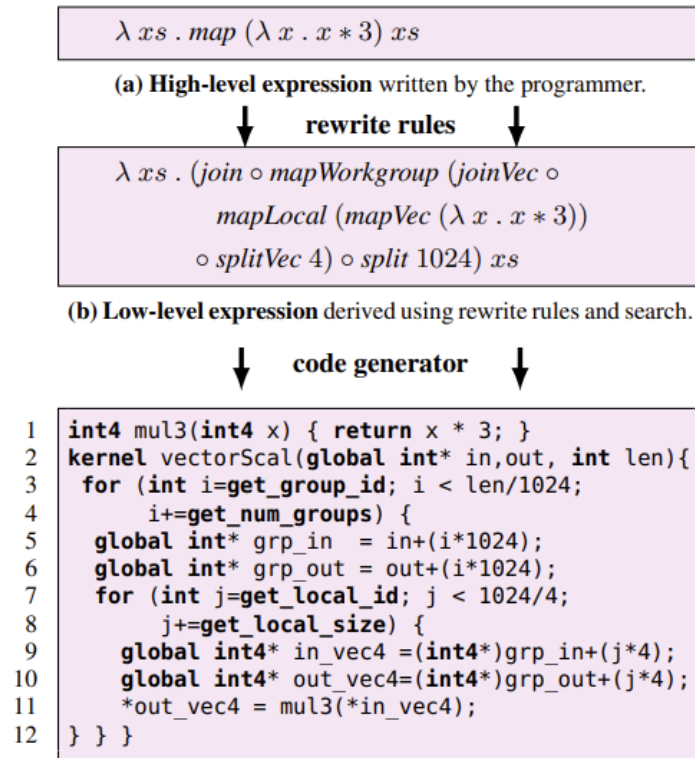
## Delite



Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.*

## GPU Functional Programming on top of Scala

# Expression Rewriting – E.g., Lift



- Ahead of time compiler that rewrites the input expressions using its own API rules as an intermediate language
- High-Performance
- Productive & Portable
- New Language, hard to adopt
- Google FASTEST Matrix Multiplication Alg. uses a variation of this technique

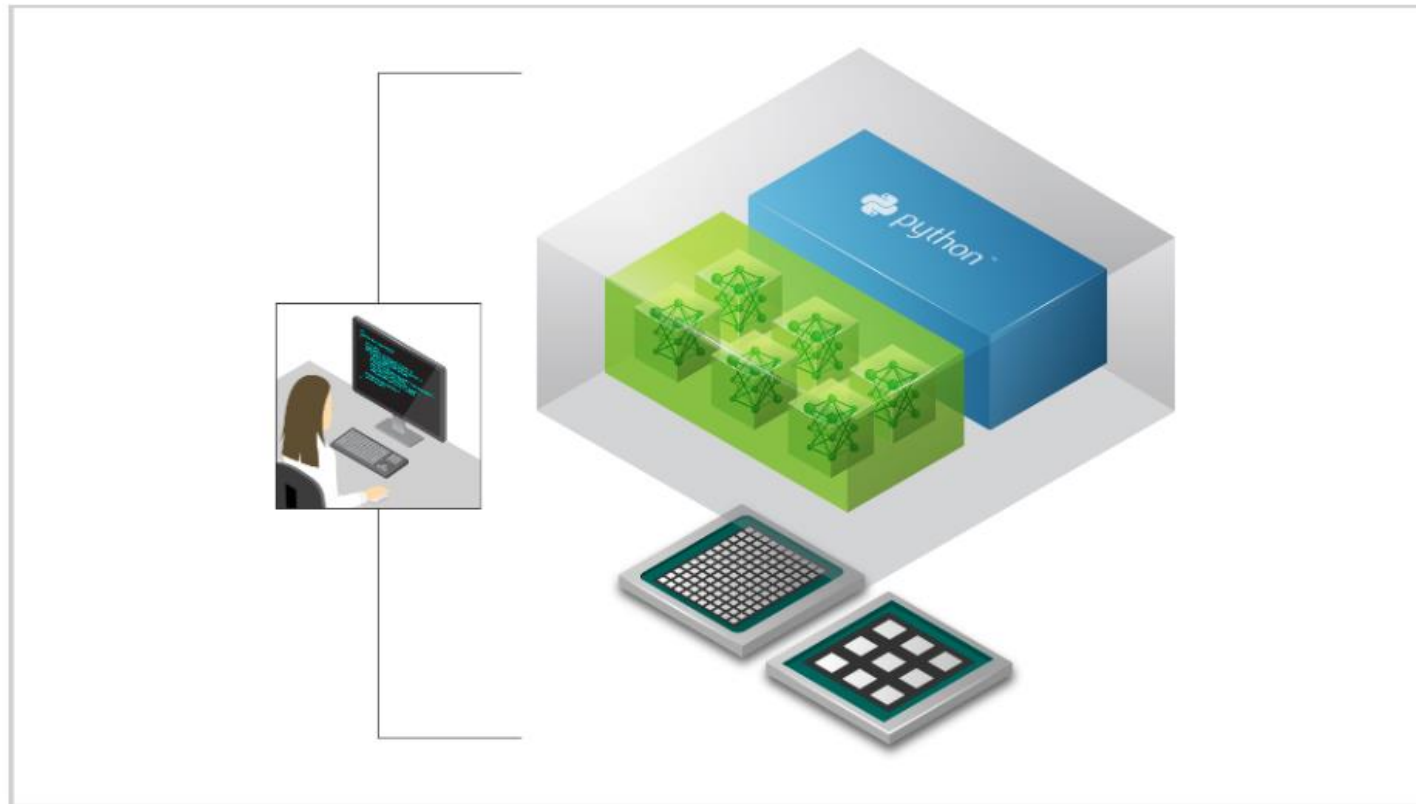


Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. ICFP 2015.



**Discovering faster matrix multiplication algorithms with reinforcement learning**  
<https://doi.org/10.1038/s41586-022-05172-4>

# Libraries



*PyTorch,  
Tensorflow,  
Etc.*

*Just libraries -> high  
performance at the cost  
of vendor lock-in*

# And, in the Java world...

## Sumatra & IBM J9:

- Using the Java Stream API for offloading code into the GPU
- Not high-portability
- Not very productive (due to hard adoption)
- Very limited

## Aparapi

- Not that easy to adopt from the Java community
- High-performance
- Very verbose
- Not composable at the method-level
- Low-tune available from Java

## Marawacc:

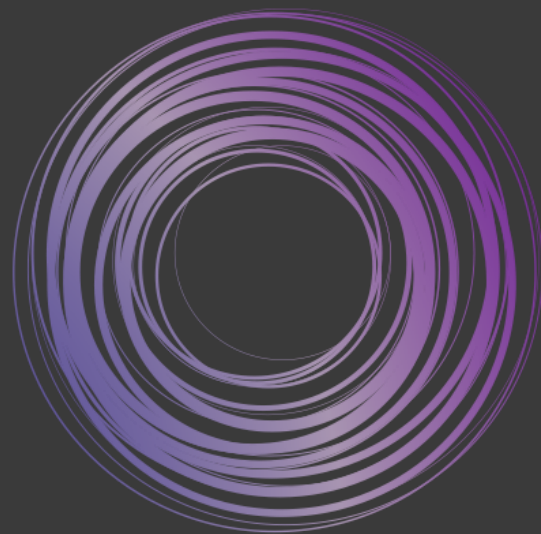
- Hard to adopt (many code changes)
- High Performance
- Very limited – **but function first, data last!**



Juan Fumero, Toomas Rimmelg, Michel Steuwer, and  
Christophe Dubach. 2015. **Runtime Code Generation and  
Data Management for Heterogeneous Computing in  
Java**

# Parallel API Design Summary of Nice-to-have features

Property	Present?
Single Source Property	
Three types of Parallelism	
Loop Parallelism and Kernel Parallelism	
Easy Access for Experts and Non-Experts	
Use of Lambdas	
Expression Rewriting	
Support for Skeletons	
Dynamic Parallelism	
Reusability of Libraries	



# TornadoVM

<https://www.tornadovm.org/>

# Different components of the TornadoVM's User API

## a) How to represent parallelism within functions/methods?

- A.1: Java annotations for expressing parallelism (**@Parallel**, **@Reduce**) for Non-Experts
- A.2: Kernel API for GPU experts (use of **kernel context** object)

## b) How to define which methods to accelerate?

Build a **Task-Graph API** to define **data In/Out** and the **code to be accelerated**

## c) How to explore different optimizations?

Build an **Execution Plan** to define different optimizations

# Tornado API - example Java sequential code for MxM

```
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 using the **Loop Parallel API**

```
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);  
            }  
        }  
    }  
}
```

Device  
Code

We add the parallel annotation as a hint for the compiler

We only have 2 annotations:

**@Parallel**  
**@Reduce**

# Tornado API - example using the **Kernel API**

```
class Compute {  
    public static void mxm(Matrix2DFloat A, Matrix2DFloat B,  
                           Matrix2DFloat C, final int size,  
                           KernelContext context) {  
        int idx = context.globalIdx;  
        int jdx = context.globalIdy;  
        float sum = 0.0f;  
        for (int k = 0; k < size; k++)  
            sum += A.get(idx, k) * B.get(k, jdx);  
        C.set(idx, jdx, sum);  
    }  
}
```

Device  
Code

Kernel-Context accesses  
thread ids, local memory  
and barriers

It needs a **Grid of Threads** to  
be passed during the kernel  
launch

# Tornado API - example

## How to identify which methods to accelerate? --> TaskGraph

```
TaskGraph taskGraph = new TaskGraph("s0")

    .transferToDevice(DataTransferMode.EVERY_EXECUTION , matrixA, matrixB)

    .task("t0", Compute::mxm, matrixA, matrixB, matrixC, size)

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



Host Code  
(Runs on CPU)

Task-Graph is a new Tornado object exposed to developers to define :

- a) **The code to be accelerated** (which Java methods?)
- b) **The data (Input/Output)** and how data should be streamed

## How to explore different optimizations? --> ExecutionPlan

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

TornadoExecutionPlan executionPlan = new TornadoExecutionPlan(itg);

executionPlan.withWarmUp()
              .withProfiler(ProfilerMode.CONSOLE)
              .withDynamicReconfiguration(PERFORMANCE, PARALLEL);

TornadoExecutionResult result = executionPlan.execute();

long elapsedKernelTime = result.getProfiler().getDeviceKernelTime(); // in ns
```

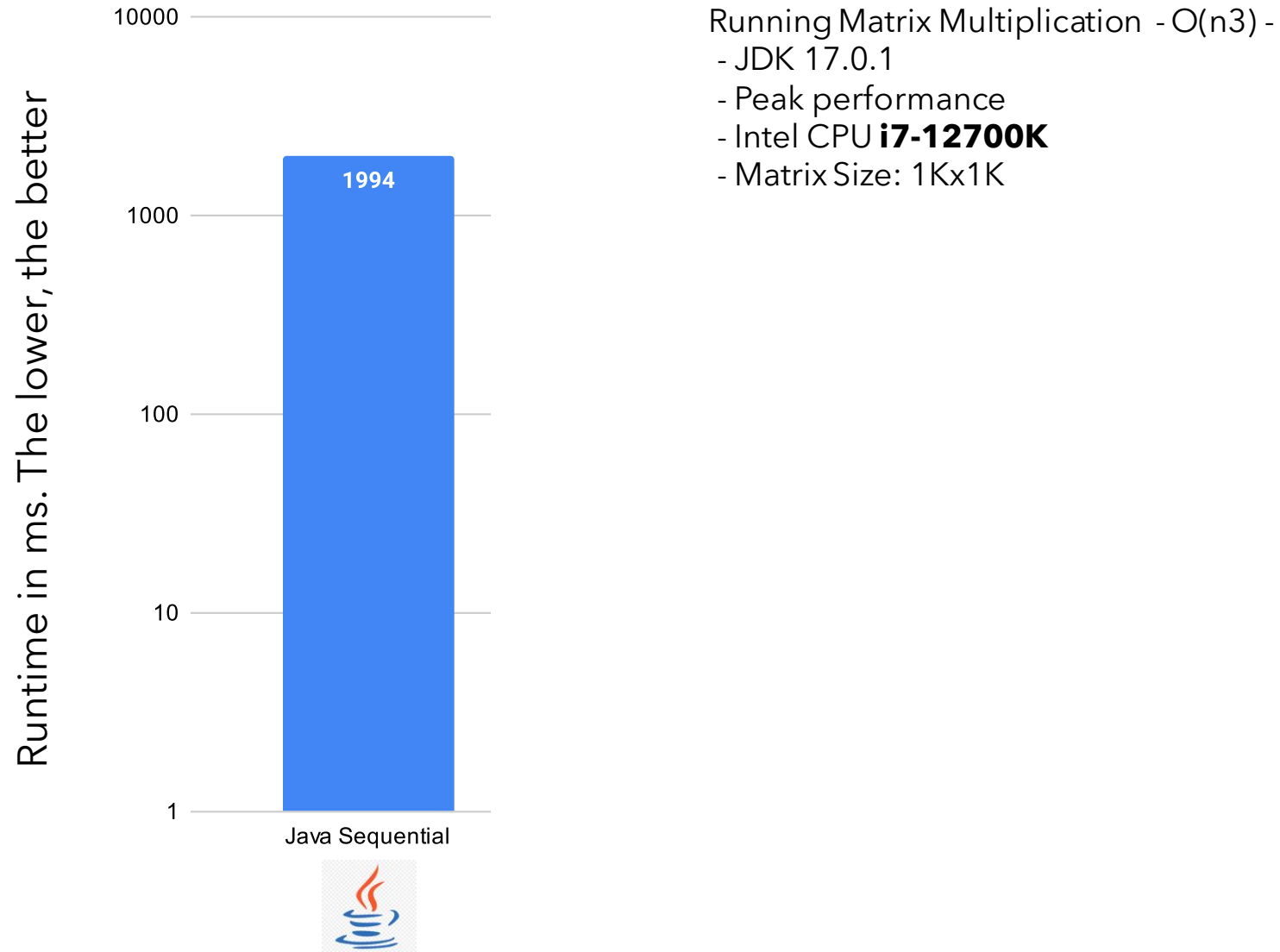


Host Code

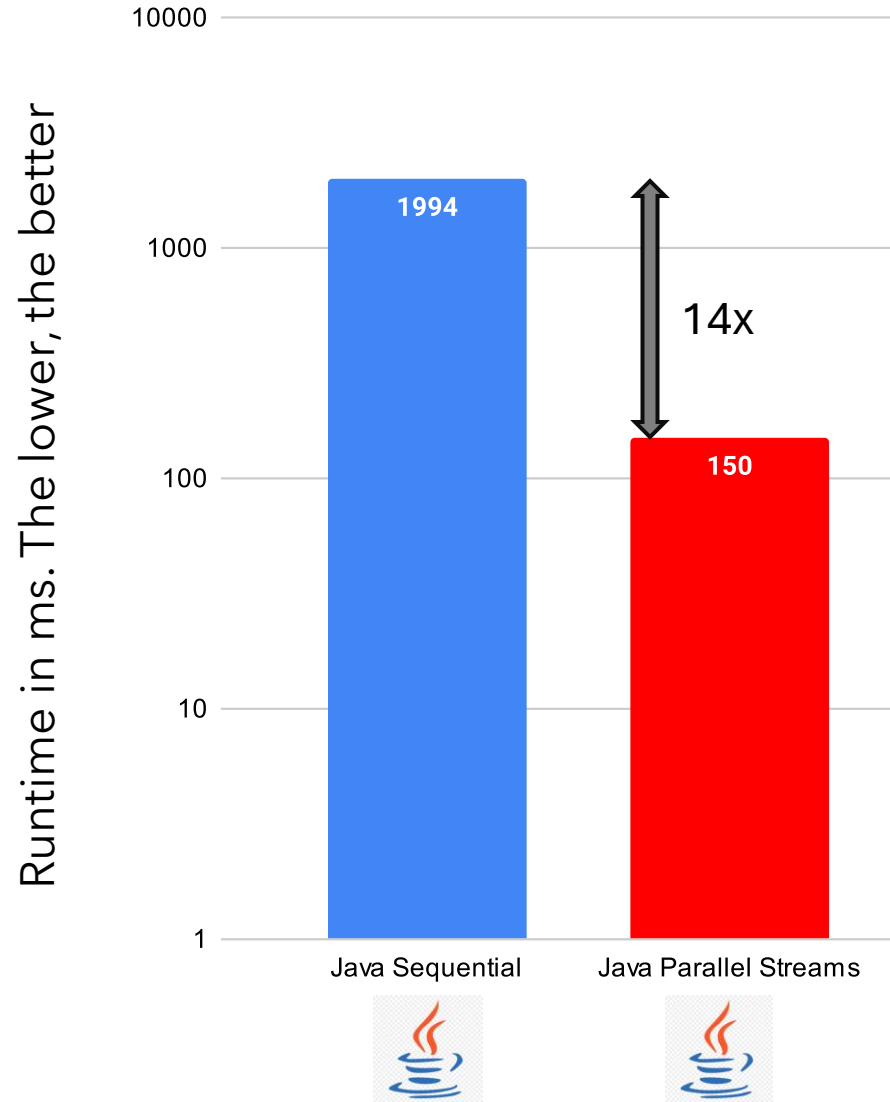
### Optional High-Level Optimization Pipelines:

- Enable/Disable Profiler
- Enable Warmup
- Enable Dynamic Reconfiguration
- Enable Batch Processing
- Enable Thread Scheduler (no need for recompilation for different grids schedulers)

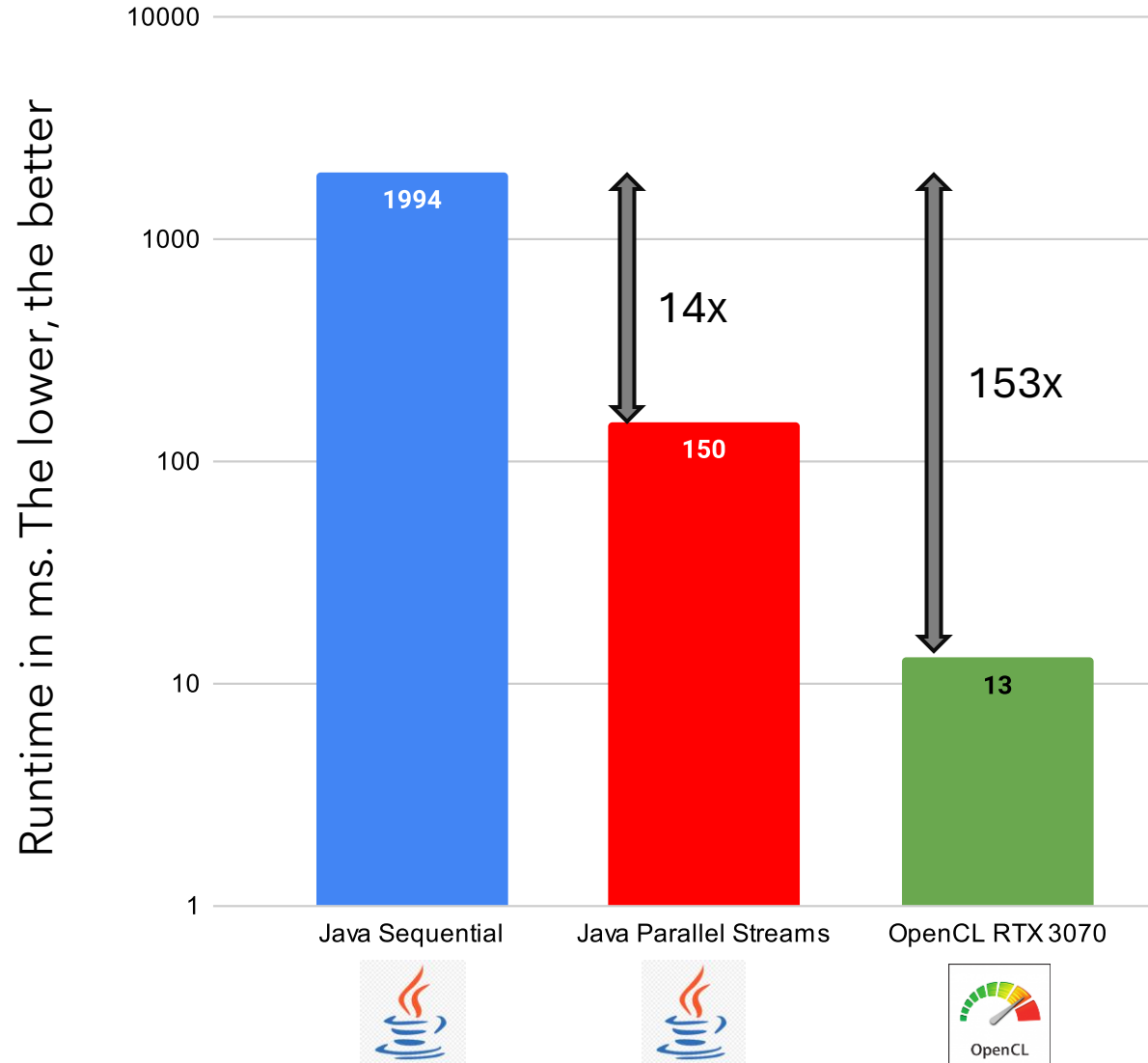
# But, why not just using the Kernel API?



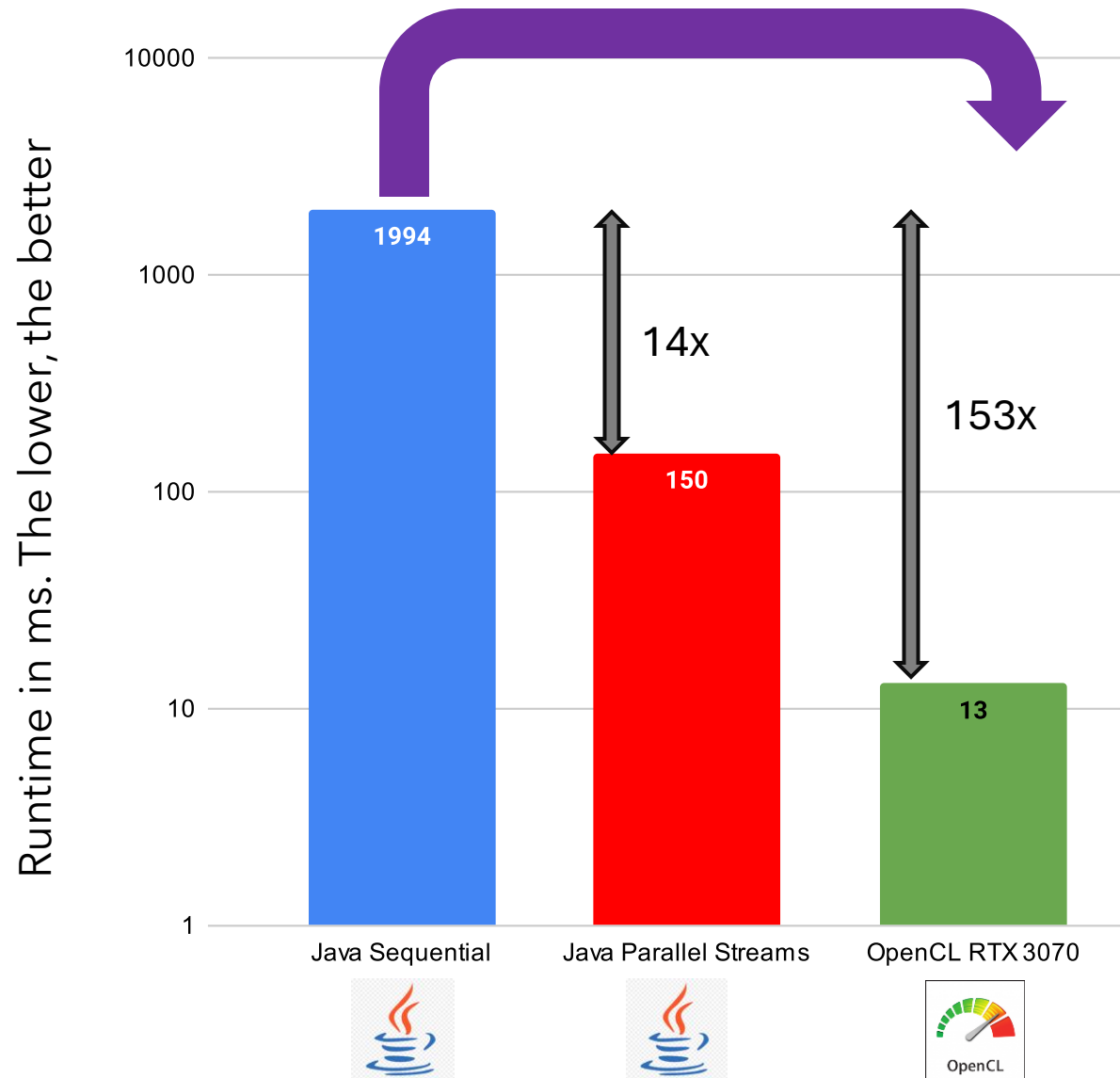
# Why not just using the Kernel API?



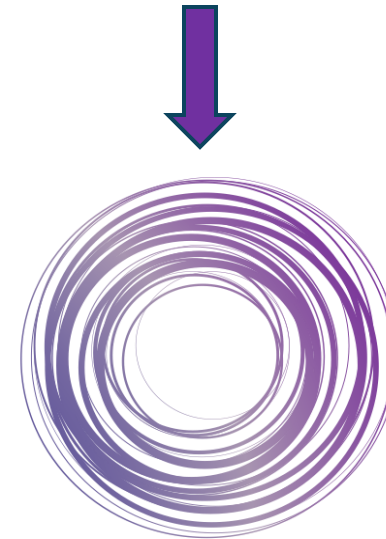
# Why not just using the Kernel API?



# Why not just using the Kernel API?

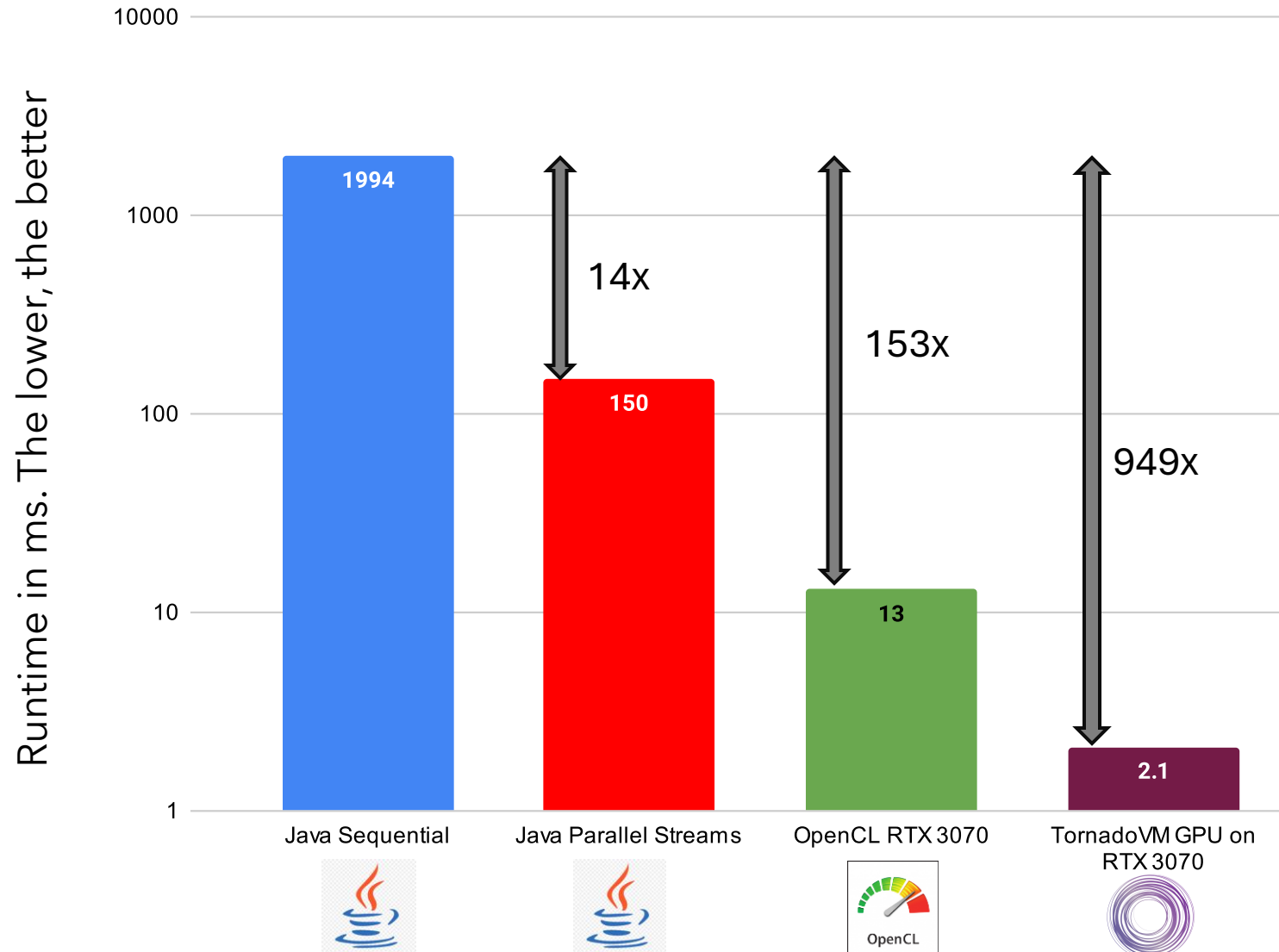


Loop Parallel API transform the code to sequential to parallel.

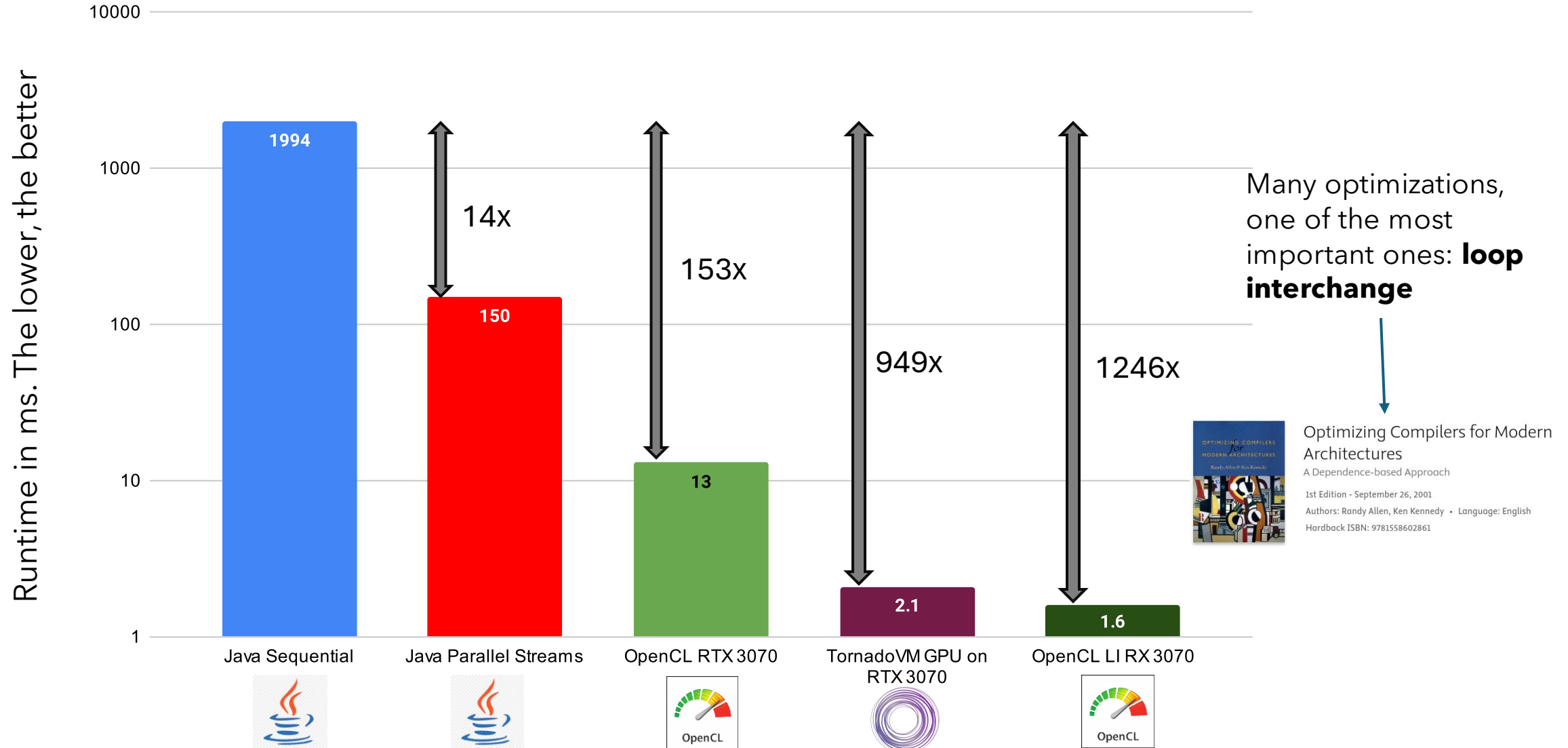




# Running with TornadoVM on NVIDIA RTX 3070



# TornadoVM Performance for MxM



## Key messages:

- 1) Unless you are an expert on the parallel architecture and the parallel programming model, a smart compiler can do a better job sometimes
- 2) Low-level programming might be faster, but that's not the way Java developers program. Thus, we need to high-level APIs

# Parallel API Design

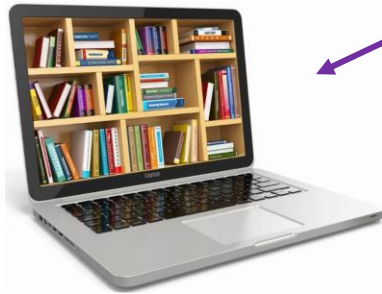
Property	Present?	
Single Source Property	✓	
Three types of Parallelism	✓	→ We haven't discussed it in detail, but it is supported
Loop Parallelism and Kernel Parallelism	✓	
Easy Access for Experts and Non-Experts	?	→ Hard to Measure
Use of Lambdas	✓	
Expression Rewriting	✓	→ Yes, automatic reductions
Support for Skeletons		
Dynamic Parallelism		
Reusability of Libraries		



VMIL'18: "Using compiler snippets to exploit parallelism on heterogeneous hardware: a Java reduction case study"

# Cool, but what about libraries?

## Options for Parallel Compute on GPUs from Managed Runtime Systems



Pre-built-Libraries (e.g, oneMKL, cuDNN,  
etc)

Use **vendor** optimized libraries  
Not easily portable  
Usually very fast and high performance

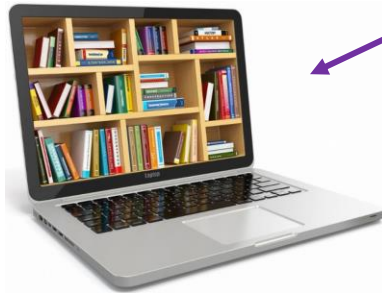


Full JIT Compiler (e.g., current TornadoVM)

Flexible, Portable, Reusable  
Lower Performance

# Proposal: Hybrid API

Options for Parallel Compute on GPUs from  
Managed Runtime Systems

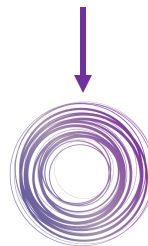


Pre-built-Libraries (e.g, oneMKL, cuDNN,  
etc)



Full JIT Compiler (e.g., current TornadoVM)

But, what if we combine  
both?



# Proposal: Hybrid API

Extension of the TornadoVM APIs for allowing JIT + Library  
Calls within the same Engine

# Hybrid API

Extension of the TornadoVM APIs for allowing JIT + Library Calls within the same Engine

```
TaskGraph tg = new TaskGraph("compute")
    .transferToDevice(. . .)
    .task("gemm", MyJavaCompute:sgemm, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
    .transferToHost(. . .);
```

Points to Existing  
Java Code  
annotated with  
**@Parallel**  
**@Reduce**

```
public static void sgemm(...) {
    for (@Parallel) {
        for (@Parallel) {
            ...
        }
    }
}
```



# Hybrid API

Example: Invoking SGEMM for Intel oneMKL (oneAPI toolkit)

```
TaskGraph tg = new TaskGraph("compute")
    .transferToDevice( . . . )
    .libraryTask("gemm", OneMKL:sgemm, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
    .transferToHost( . . . );
```



Points to a Proxy Class that gives you access to Intel oneMKL

# Hybrid API: So, what is the deal?

Going Hybrid: JIT + Library Tasks

**Uses: Deep Learning, AI, Math Library, Data Bases, etc.**

```
TaskGraph tg = new TaskGraph("compute")
    .taskGraph.transferToDevice(DataTransferMode.FIRST_EXECUTION, data)

    .task("prep", MyJavaPrepClass::dataInitOnGPU, data)    // FULL JIT (Java->Accelerator)

    .libraryTask("gemm", CuDNN::cudnnActivationForward, alpha, data, beta, output) //call to native
CuDNN

    .task("postProcessing", MyOtherJavaClass::post, output)    // FULL JIT (Java->Accelerator)

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

We have prototypes for oneMKL and cuDNN

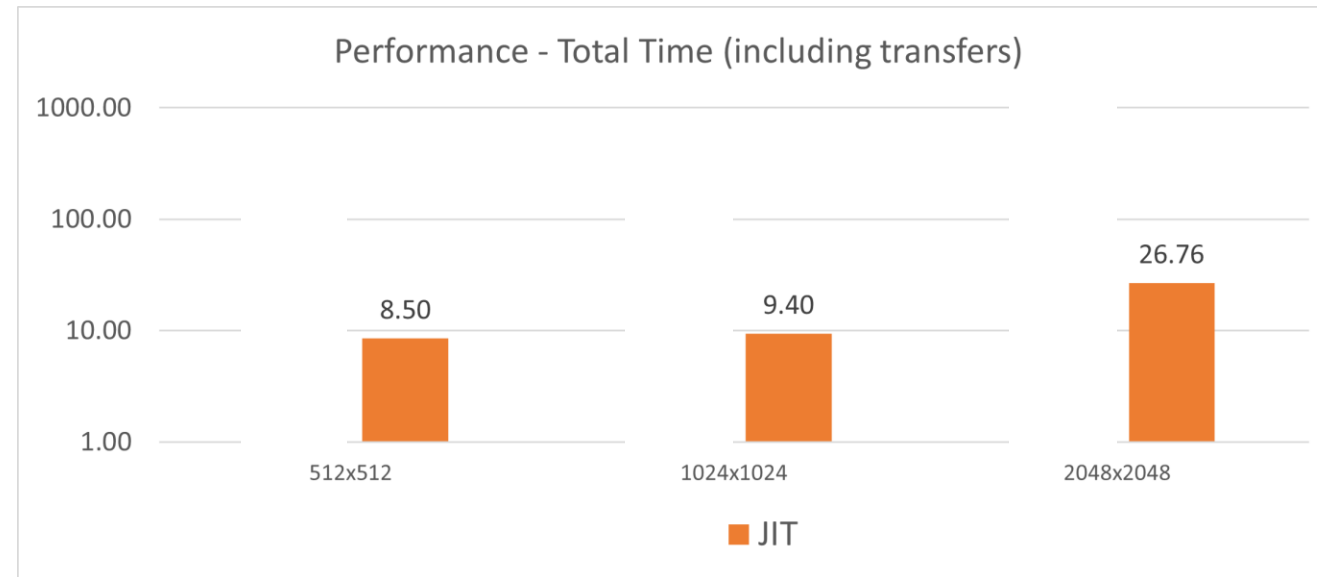
# What about performance? Running SGEMM from oneMKL

Running on Intel i9-10885H  
Processors:

- Intel UHD Graphics 630

TornadoVM 0.15.2-dev

Intel Runtime: 21.38.21026



The Higher, the better. Performance vs single-threaded Java

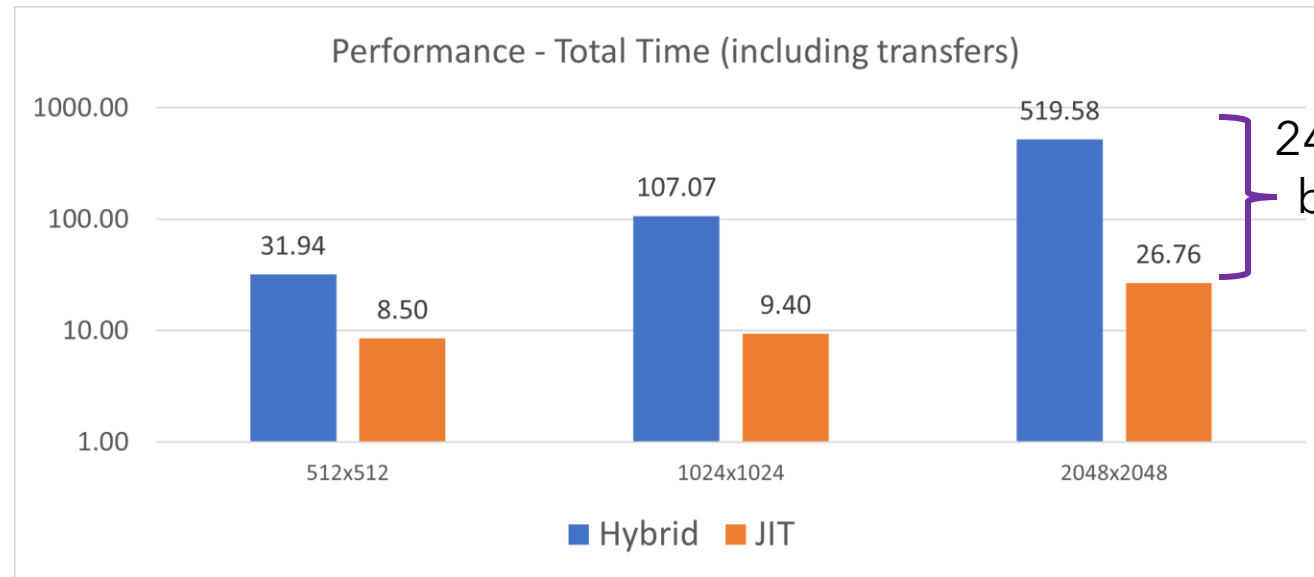
# What about performance? Running SGEMM from oneMKL

Running on Intel i9-10885H  
Processors:

- Intel UHD Graphics 630

TornadoVM 0.15.2-dev

Intel Runtime: 21.38.21026



24.7x on top of the  
base TornadoVM  
Performance

The Higher, the better. Performance vs single-threaded Java

# Do-While Pattern

```
TaskGraph graph = new TaskGraph("compute-graph")
    .transferToDevice(DataTransferMode.EVERY_EXECUTION, a, b)
    .task("l1", MyClass::computeKernelParallelism, context, a, b, c)
    .task("h1", MyClass::computeLoopParallelism, c, d)
    .transferToHost(DataTransferMode.EVERY_EXECUTION, c,
d);
while(condition) {
    ts.execute(grid);
}
```

- \* This code is valid but it transfers data back and forth until condition is reached
- \* No possible way to iterate in one task, rather than the whole task-schedule

# Do-While Pattern at the task-level

```
TaskGraph graph = new TaskGraph("compute-graph")
    .transferToDevice(DataTransferMode.EVERY_EXECUTION, a, b)
    .task("l1", MyClass::computeKernelParallelism, context, a, b, c)
    .task("h1", MyClass::computeLoopParallelism, c, d)
    .transferToHost(DataTransferMode.EVERY_EXECUTION, c,
d);
while(condition) {
    ts.execute(grid);
}
```

Two possible solutions:

Variables using U-Shared Memory

```
// JNI Code
// Save back and forth for JNI
while (condition) {
    clEnqueueNDRange(...)
}
```

Warning: Not tested yet

```
// Generated Kernel
__kernel generated (...) {
    compute(...)
    syncDevice()
    if (!condition)
        child_kernel- generated(N-Threads);
}
```

# Do-While Pattern at the Execution Plan

```
executionPlan.executeUntil(condition);
```

# Supporting Dynamic Parallelism within TornadoVM

```
public void dynamicParallelism(double[] input, float[] output,  
                               TornadoVMContext c) {  
  
    ...  
    compute ...  
    ...  
    if (context.threadIdx == 0) {  
        context.launch( (a, b) -> {  
            child_kernel_code ...  
        }  
    }  
}
```

← Lambda Expression to  
represent child  
kernels.



# Supporting Dynamic Parallelism within TornadoVM

```
public void dynamicParallelism(double[] input, float[] output,
                               TornadoVMContext c) {

    ...
    compute ...
    ...
    if (context.threadIdx == 0) {
        context.launch( (a, b) -> {
            child_kernel_code ...
        })
    }
}
```

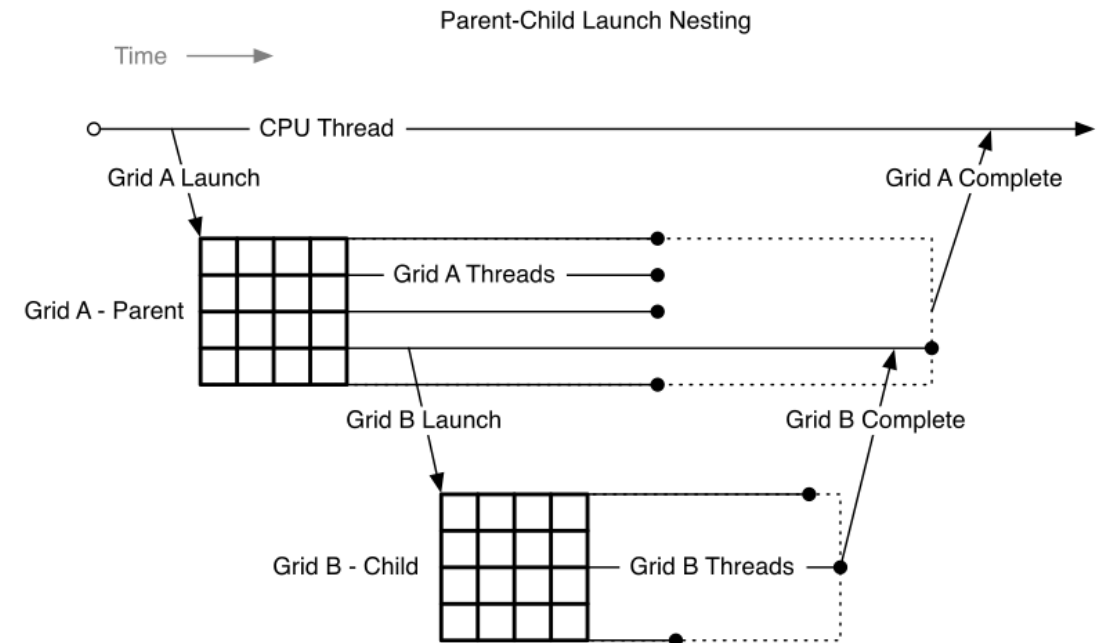


Image from NVIDIA docs

# Pushing for a more functional style

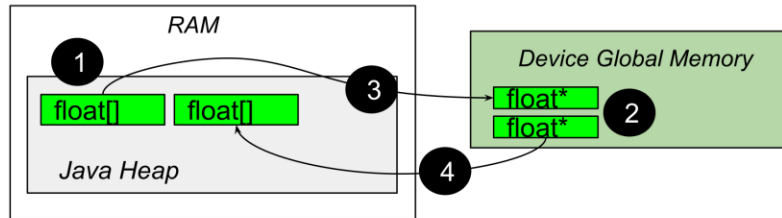
```
TaskGraph graph = new TaskGraph("functional")  
  
    .trasferToDevice(a, b)  
  
    .map("map01", (x, y, z) -> { code }, a, b, c, context)  
  
    .filter("filter1", (c) -> { idx = ... }, c, context)  
  
    .reduce("reduce1", (x, y) -> x + y, a, c, context)  
  
    .transferToHost(c);
```

Let's talk about memory:

In Java, and MRS, we need  
to design memory  
management VERY carefully

# What about memory?

On-Heap Data Structures (TornadoVM's current approach)



1. Memory reserved in the Java Heap
2. Device Buffer Malloc (e.g., GPU)
3. Data Transfer (host->device)
4. **Kernel Execution**
5. Data Transfer (device -> host)

*Good Luck with the GC to not move pointers*

Besides, we have experimented with other ideas such as :  
**Unified Shared Memory Java Heap**



**Unified Shared Memory: Friend or Foe?** Juan Fumero, Florin Blanu, Athanasios Stratikopoulos, Steve Dohrmann, Sandhya Viswanathan, Christos Kotselidis. **MPLR23**

SpringerBriefs in Computer Science  
Juan Fumero · Athanasios Stratikopoulos ·  
Christos Kotselidis



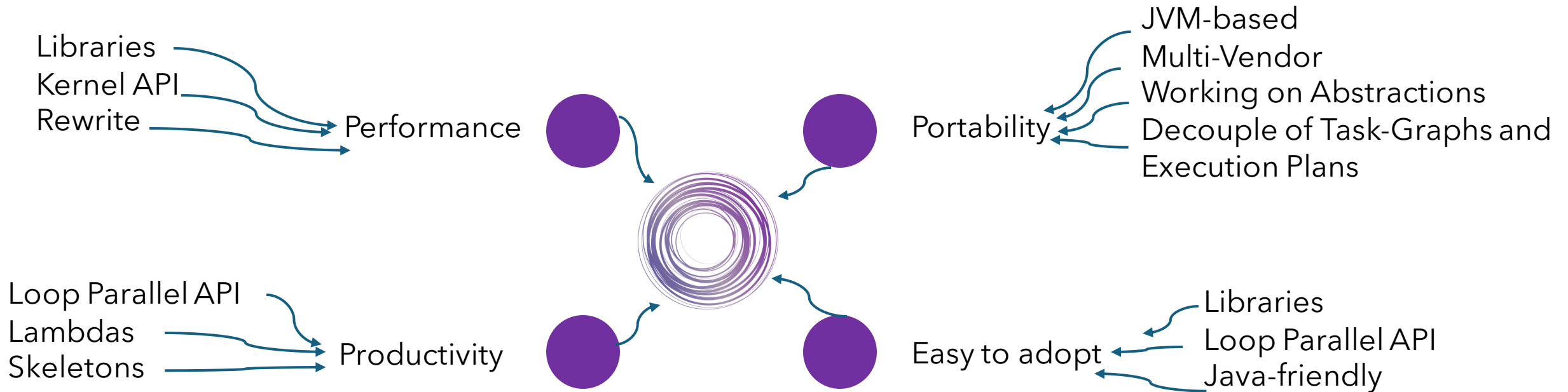
<https://link.springer.com/book/9783031495588>

**LOCK GC and blocking operations**



**Off-heap Data Structures** (e.g., Using Direct Memory or **Panama APIs**)

# Conclusions

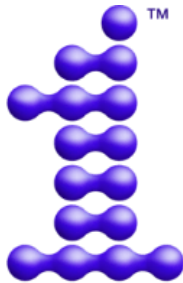


Designing parallel APIs is not a trivial task, and it is all about trade-offs:

- What are we gaining?
- What are we losing?
- What can we do better?

MANCHESTER  
1824

The University of Manchester



**oneAPI**



# Thank you so much for your attention

- Partially supported by the Horizon Europe AERO 101092850
- Partially supported by Intel Grant

Special thanks to:

- \* Christos Kotselidis (Manchester - TornadoVM)
- \* Thanos Stratikopoulos (Manchester - TornadoVM)
- \* Gary Frost (Aparapi co-founder, Senior Java Architect, Oracle)
- \* Peng Tu (Intel oneAPI)
- \* Steve Dohrmann (Intel Senior OpenJDK Architect)
- \* Sandhya Viswanathan (Intel Senior OpenJDK Architect)

Juan Fumero

<juan.fumero@manchester.ac.uk>