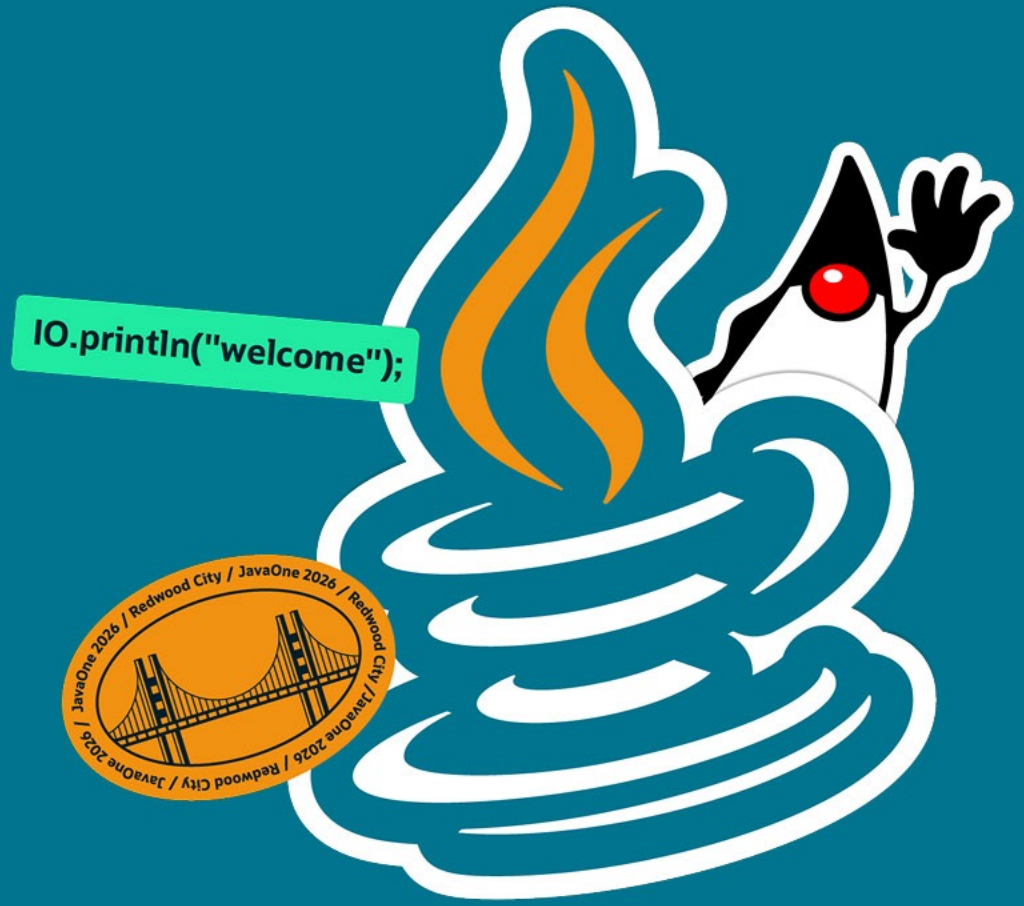
JavaOne™

Under the HAT

Empowering GPU Acceleration for Java



Juan Fumero
Consulting Member of Technical Staff
Java Core Libraries, Oracle
March 17th, 2026

Agenda title

1 Brief Introduction to Code Reflection

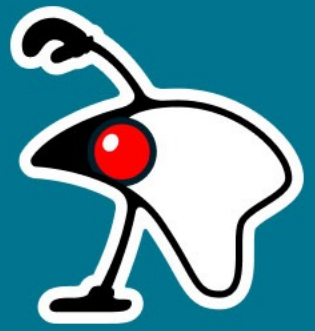
2 Heterogeneous Accelerator Toolkit (HAT)

3 HAT Programming Model

4 Study: Cost of Data Abstractions

5 Study: How Far Could we Go? A Matmul Test Case

6 Conclusions



Brief Introduction to Code Reflection

Project Babylon: Quick Overview

Project Babylon introduces an enhanced reflection API that facilitates Java programmers to create a symbolic representation of methods and lambdas (called code-models) to be able to manipulate, transform and generate code for foreign programming models.

Project Babylon: Quick Overview

Project Babylon introduces an enhanced reflection API that facilitates Java programmers to create a symbolic representation of methods and lambdas (called code-models) to be able to manipulate, transform and generate code for foreign programming models.

```
@Reflect
float compute(float a, float b, float c) {
    return a * b + c;
}
```

Project Babylon: Quick Overview

Project Babylon introduces an enhanced reflection API that facilitates Java programmers to create a symbolic representation of methods and lambdas (called code-models) to be able to manipulate, transform and generate code for foreign programming models.

```
@Reflect
float compute(float a, float b, float c) {
    return a * b + c;
}
```

```
func @"compute" (%0 : java.type:"float", %1 : java.type:"float",
%2 : java.type:"float")java.type:"float" -> {
    %3 : Var<java.type:"float"> = var %0 @"a"
    %4 : Var<java.type:"float"> = var %1 @"b"
    %5 : Var<java.type:"float"> = var %2 @"c"
    %6 : java.type:"float" = var.load %3
    %7 : java.type:"float" = var.load %4
    %8 : java.type:"float" = mul %6 %7
    %9 : java.type:"float" = var.load %5
    %10 : java.type:"float" = add %8 %9
    return %10
};
```

Project Babylon: Quick Overview

Project Babylon introduces an enhanced reflection API that facilitates Java programmers to create a symbolic representation of methods and lambdas (called code-models) to be able to manipulate, transform and generate code for foreign programming models.

```
@Reflect
float compute(float a, float b, float c) {
    return a * b + c;
}
```

```
func @"compute" (%0 : java.type:"float", %1 : java.type:"float",
%2 : java.type:"float")java.type:"float" -> {
    %3 : Var<java.type:"float"> = var %0 @"a"
    %4 : Var<java.type:"float"> = var %1 @"b"
    %5 : Var<java.type:"float"> = var %2 @"c"
    %6 : java.type:"float" = var.load %3
    %7 : java.type:"float" = var.load %4
    %8 : java.type:"float" = mul %6 %7
    %9 : java.type:"float" = var.load %5
    %10 : java.type:"float" = add %8 %9
    return %10
};
```

```
func @"compute" (%0 : java.type:"float", %1 : java.type:"float", %2 :
java.type:"float")java.type:"float" -> {
    %3 : Var<java.type:"float"> = var %0 @"a"
    %4 : Var<java.type:"float"> = var %1 @"b"
    %5 : Var<java.type:"float"> = var %2 @"c"
    %6 : java.type:"float" = var.load %3
    %7 : java.type:"float" = var.load %4
    %8 : java.type:"float" = var.load %5
    %9 : java.type:"float" = fma %6 %7 %8
    return %9
};
```

```
$ java --add-modules jdk.incubator.code \
    -cp target/crsamples-1.0-SNAPSHOT.jar \
    oracle.code.samples.DialectFMAOp
```

What can we do with Code Reflection?

Articles:

- **LINQ-like** language within Java: https://openjdk.org/projects/babylon/articles/linq
- **Automatic differentiation**: https://openjdk.org/projects/babylon/articles/auto-diff
- **ONNX** Runtime: https://github.com/openjdk/babylon/tree/code-reflection/cr-examples/onnx
- **GPU Frontend for CUDA/OpenCL: this presentation**
  - https://openjdk.org/projects/babylon/articles/hat-matmul/hat-matmul

Related Sessions at JavaOne'26:

Java and AI [LRN1423]

Tuesday, Mar 17 |
10:30 AM - 11:20 AM PDT
Auditorium

We'll discuss how Java is already a good fit for AI and how it can fit better with existing and future features of the Java platform. Such features vary across the whole Java platform, from the languag…

Reflecting on HAT: A Project Babylon Case Study…

Tuesday, Mar 17 |
11:30 AM - 12:20 PM PDT
Room 202

Project Babylon continues to simplify access to foreign programming models through the development of code reflection. Using code reflection, a Java…

Writing GPU-Ready AI Models in Pure Java with Babylon…

Tuesday, Mar 17 |
4:00 PM - 4:50 PM PDT
Auditorium

Imagine building AI models like LLMs and image classifiers directly in Java and running them efficiently on your GPU. Project Babylon introduces the…

Integrating ONNX for Generative AI LLMs in Java…

Tuesday, Mar 17 |
5:00 PM - 5:50 PM PDT
Auditorium

The Open Neural Network Exchange (ONNX) provides a standardized format for machine learning models, enabling seamless deployment across various…

Preparing for Incubation: Code Reflection is in Draft Mode

JEP draft: Code reflection (Incubator)

Owner	Paul Sandoz
Type	Feature
Scope	JDK
Status	Draft
Component	core-libs
Effort	L
Duration	L
Reviewed by	Adam Sotona, Gary Frost, Juan Fumero, Maurizio Cimadamore
Created	2025/06/30 19:54
Updated	2026/02/16 15:49
Issue	8361105

Summary

Enhance the core reflection API to model Java code, build and transform models of Java code, and access models of Java code in methods and lambda expressions. Libraries can use this enhancement to analyze Java code and extend its reach, such as executing it as code on GPUs. This is an incubating API.

Goals

1. Enable Java developers to interface with non-Java (foreign) programming models using familiar Java language constructs, such as lambda expressions and static typing.
2. Encourage libraries to expose novel programming models to Java developers without requiring developers to embed non-Java code inside Java code, or to write tedious Java code that builds data structures to model Java code or other (foreign) code.

https://openjdk.org/jeps/8361105

JEP (Java Enhancement Proposal) **8361105**

- No concrete date yet, but working on first incubation
- **Available on GitHub**: https://github.com/openjdk/babylon
- **Examples-Suite**: https://github.com/openjdk/babylon/tree/code-reflection/cr-examples

If you want, you can get involved. Feedback is welcome.
Use the **babylon-dev** mailing list:
https://mail.openjdk.org/pipermail/babylon-dev/

Enabling Acceleration on GPUs with Code Reflection

Heterogeneous Accelerator Toolkit (HAT)

What are we trying to answer?

Is code reflection good enough to support GPU programming models?

- GPU support is an ideal **use case for code reflection**
- Study could we **target to specific workloads** (e.g., AI, LLMs, HPC, Simulations, Big Data, etc) to be accelerated efficiently **from Java**

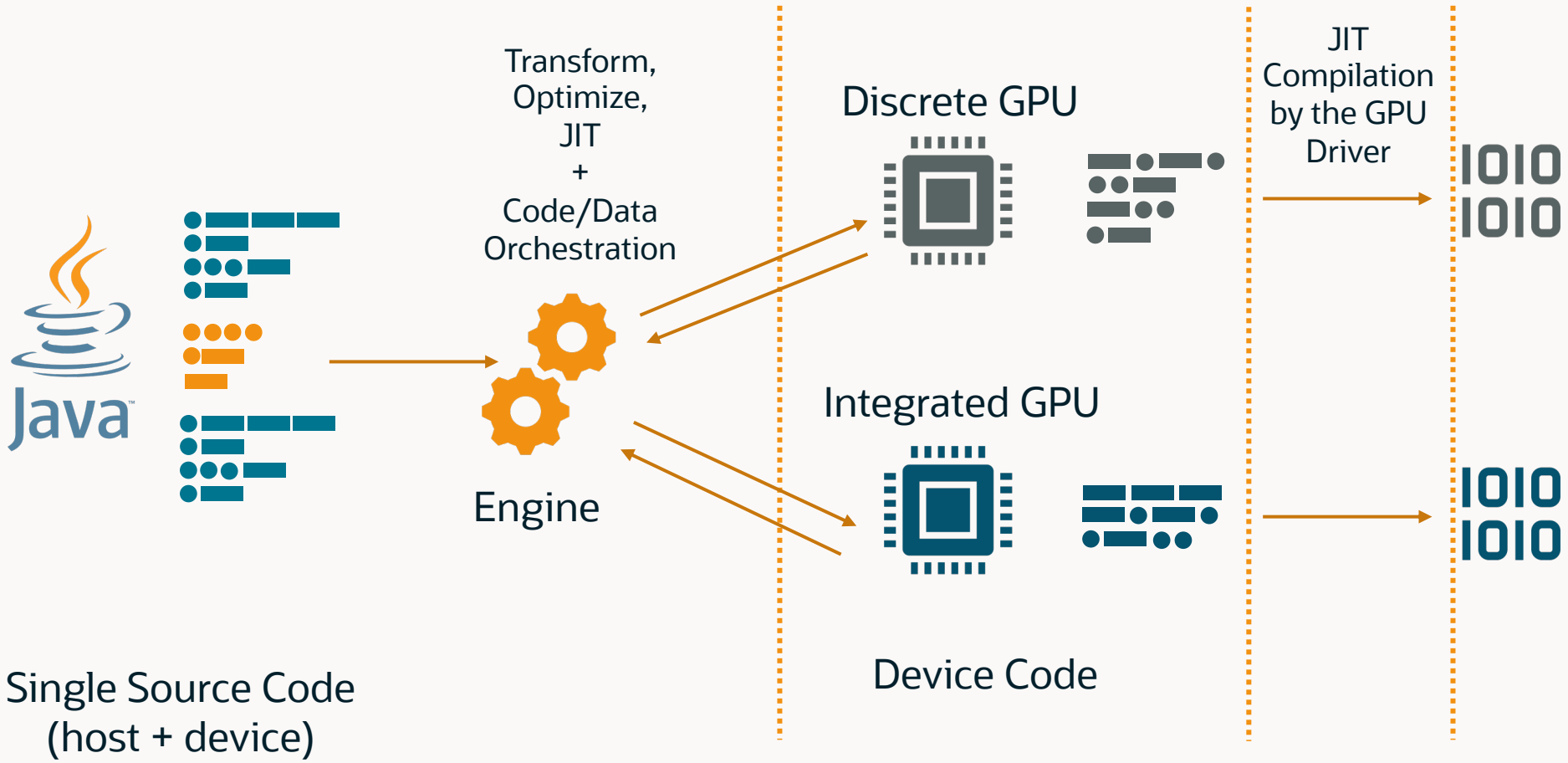
But, how do we program/accelerate from Java?

What is Around WRT Java & GPUs?

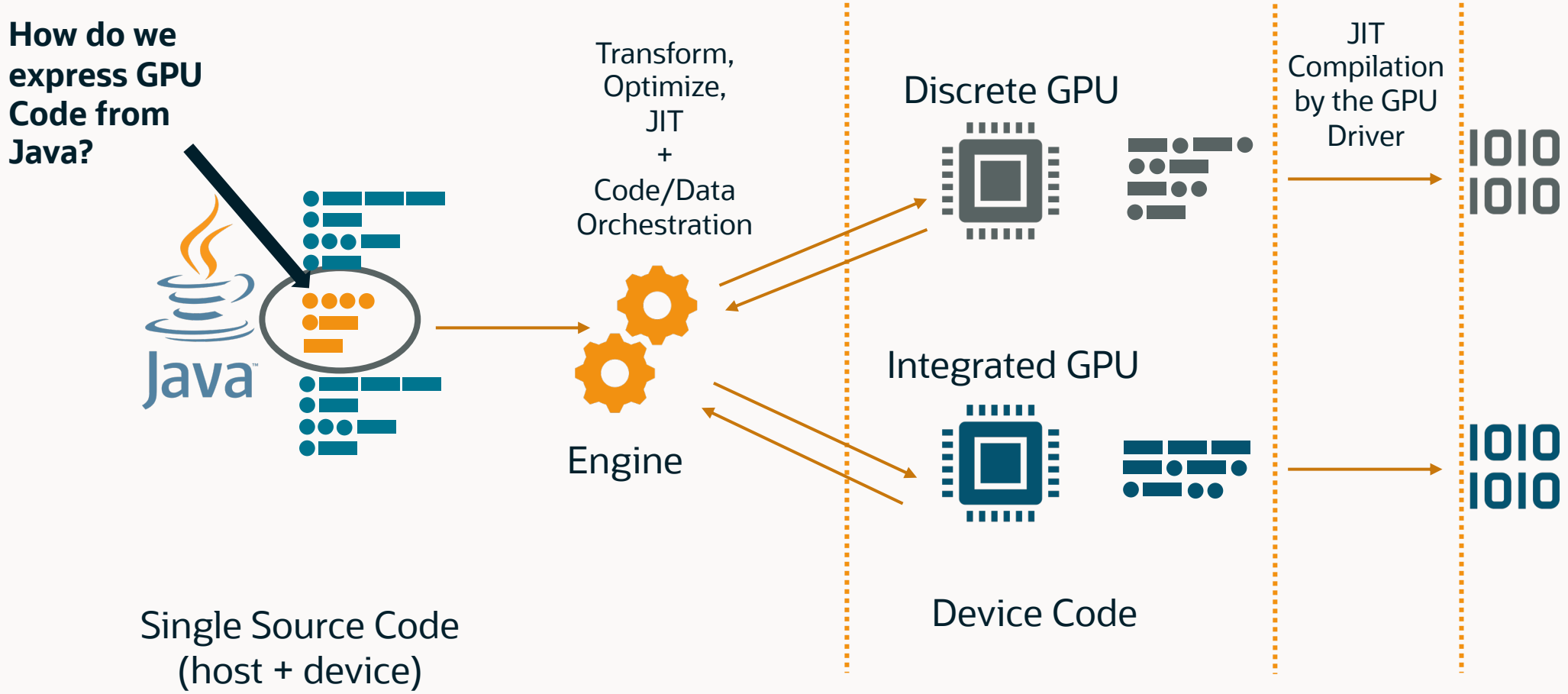
Sumatra: https://openjdk.org/projects/sumatra/
Aparapi: https://github.com/Syncleus/aparapi
IBM GPU J9: link
RootBeer: https://github.com/bsletten/rootbeer1
JaBEE: https://dl.acm.org/doi/10.1145/2159430.2159439
Marawacc: https://github.com/jjfumero/marawacc
TornadoVM: https://github.com/beehive-lab/TornadoVM

- Many of these projects **have focused compute kernels in isolation**
- **TornadoVM** introduced **analysis for multiple kernels for a single program**
- They provide very good programming abstractions at the cost of potentially **leaving some performance on the table**.
- However, **for some domains**, e.g., **AI**, just **running faster than Java might not be enough**.
- In HAT, we are focusing on **larger program analysis** and **ways to perform the extra mile to deliver high-performance**, and investigating how we create new abstractions without losing performance.

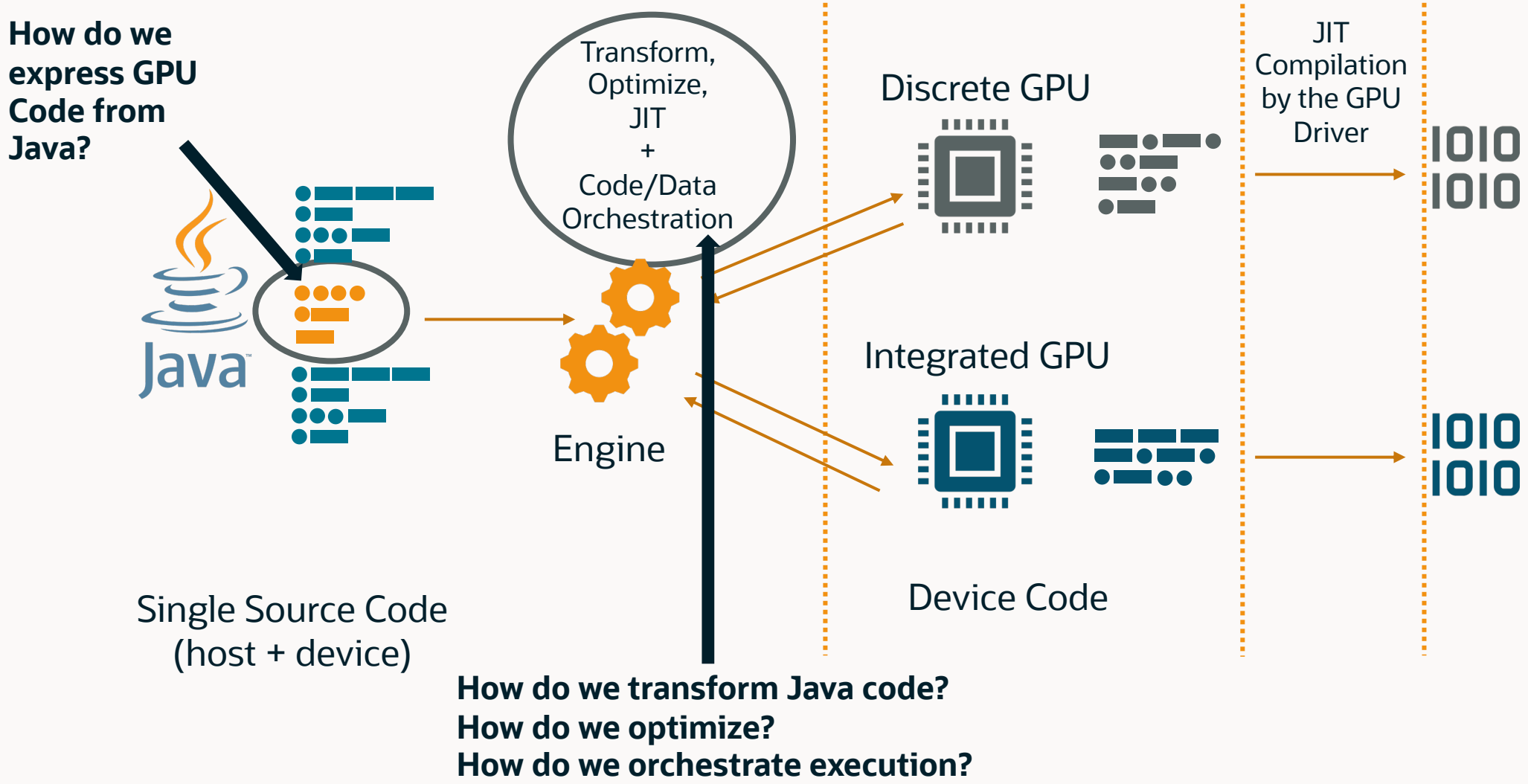
High-Level Overview: GPU Programming Workflow from Java



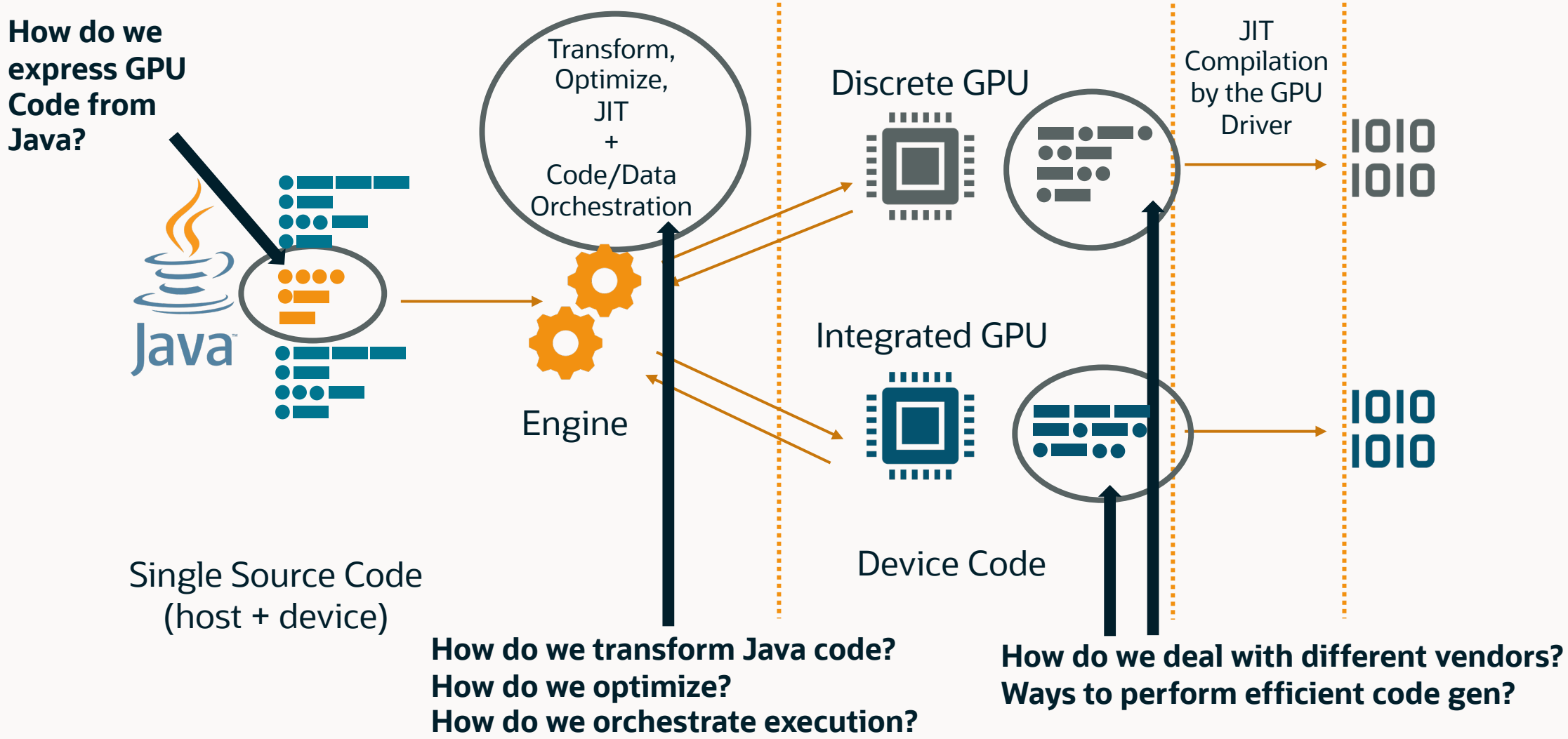
Identifying the Challenges: (1) APIs



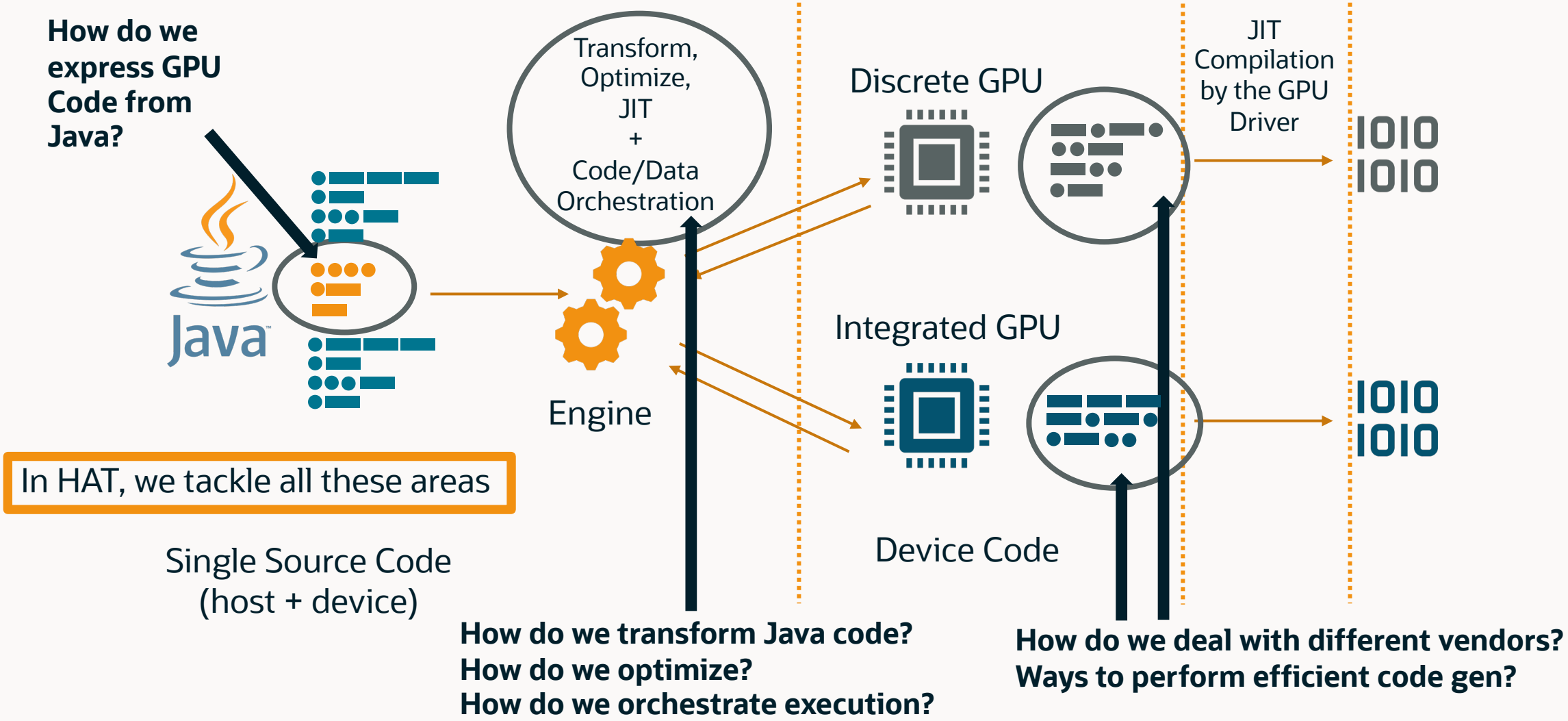
Identifying the Challenges: (2) Code/Data Orchestration



Identifying the Challenges: (3) Vendor Agnostic + Efficiency



Identifying the Challenges



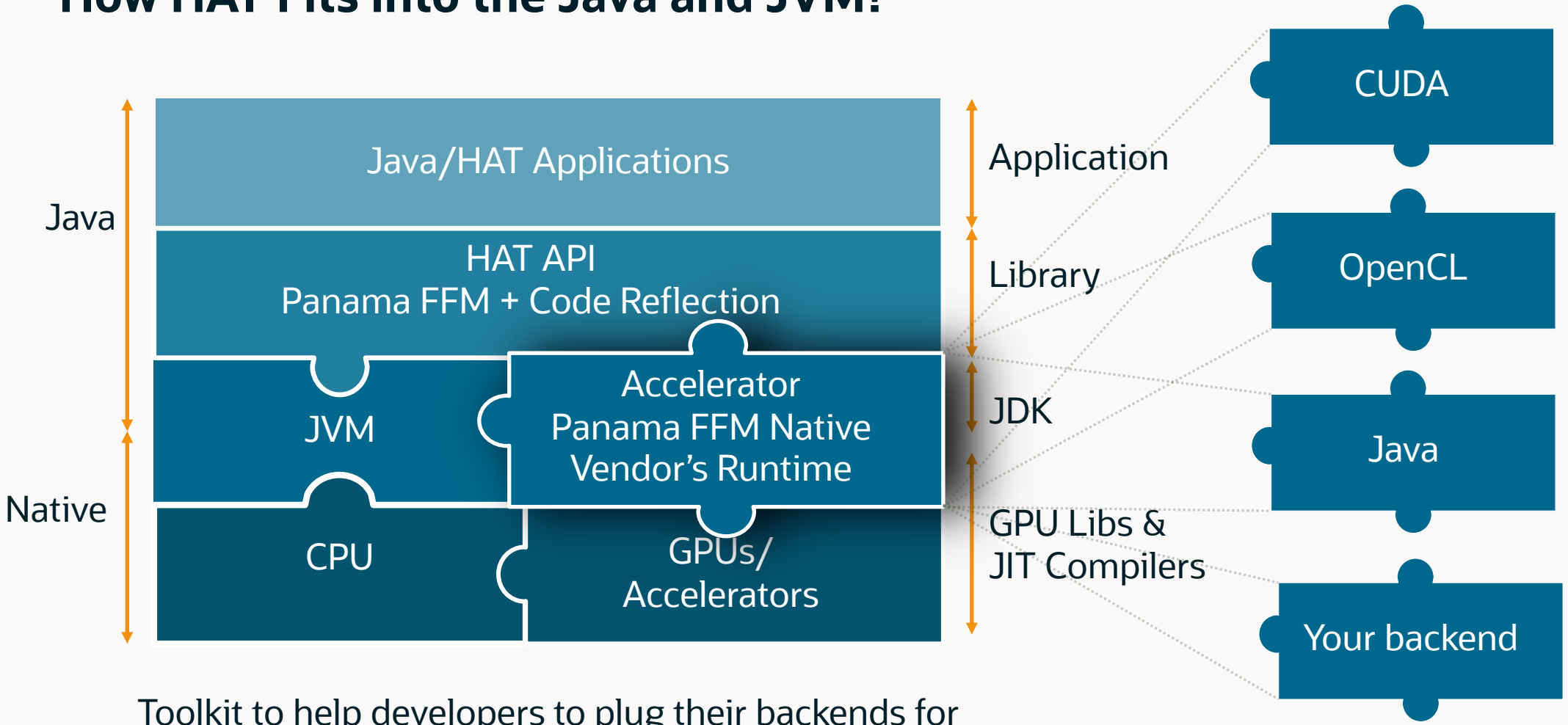
Heterogeneous Accelerator Toolkit (HAT)

HAT is a **Java Toolkit** to target **Hardware Accelerators**. It currently offers:

- An **API for Kernel Programming**
- An **API for Composing** Compute Kernels (**compute-graphs**)
- An **Interface for Abstracting Data** and Easy Mapping to Accelerators (`Iface`)
- A **plugable backend** abstraction
  - CUDA
  - OpenCL
  - Java
  - **Plugin your own**

During the presentation, we will expand on each of these components with examples and some performance numbers.

How HAT Fits into the Java and JVM?



Toolkit to help developers to plug their backends for hardware accelerators

HAT Programming Model

Main components to accelerate Java programs on GPUs

HAT is evolving

Disclaimer:

HAT is continiously evolving and we are improving its interfaces and studying new abstractions.

This presentation shows the current state and highlights what’s possible to achieve by using Code Reflection and Panama FFM

Let’s look at an example: Vector Multiplication v1 in Java

```
public void vectorMultiplication(float[] a, float[] b, float[] c) {
    for (int i = 0; i < a.length; i++) {
        c[i] = a[i] * b[i];
    }
}
```

Let’s look at an example: Vector Multiplication v2 in Java

```
public void vectorMultiplication(float[] a, float[] b, float[] c) {
    for (int i = 0; i < a.length; i++) {
        c[i] = a[i] * b[i];
    }
}
```

Work to be done per iteration

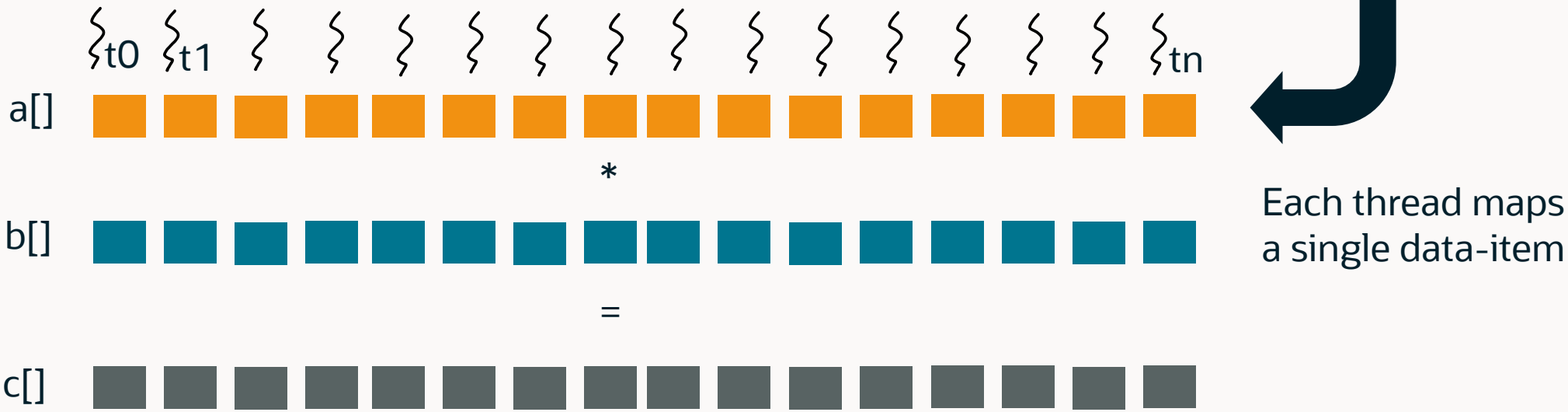
```
public void compute(int i, float[] a, float[] b, float[] c) {
     c[i] = a[i] * b[i];
}

public void vectorMultiplication(float[] a, float[] b, float[] c) {
    for (int i = 0; i < a.length; i++) {
        compute(i, a, b, c);
    }
}
```

“HATify” Vector Multiplication: Step 1: representing a kernel

Isolate a method with the work to be done per thread

```
public void compute(int i, float[] a, float[] b, float[] c) {
     c[i] = a[i] * b[i];
}
```



Each thread maps a single data-item

“HATify” Vector Multiplication: Step 1: representing a kernel

```
public void compute(int i, float[] a, float[] b, float[] c) {
     c[i] = a[i] * b[i];
}
```

```
@Reflect
public void compute(KernelContext kc, F32Array a, F32Array b, F32Array c) {

     float valueA = arrayA.array(kc.gix);
     float valueB = arrayB.array(kc.gix);
     arrayC.array(kc.gix, (valueA * valueB));
}
```

Work to be done per thread

“HATify” Vector Multiplication: Step 1

```
public void compute(int i, float[] a, float[] b, float[] c) {
     c[i] = a[i] * b[i];
}
```

```
@Reflect
public void compute(KernelContext kc, F32Array a, F32Array b, F32Array c) {
    if (kc.gix < arrayA.length()) {
        float valueA = arrayA.array(kc.gix);
        float valueB = arrayB.array(kc.gix);
        arrayC.array(kc.gix, (valueA * valueB));
    }
}
```

We add a condition to check for out of bounds accesses

Note that, at run-time, we can launch any number of threads, indendently of the actual array sizes. Thus, we need to protect for illegal accesses.

“HATify” Vector Multiplication: Step 2- Which set of methods to offload?

```
@Reflect
public static void computeContext(ComputeContext cc,
                                  F32Array arrayA,
                                  F32Array arrayB,
                                  F32Array arrayC) {

    // Define how many threads to run
    // In this case, we launch as many threads as data items in arrayA
    NDRange ndRange = NDRange.of(arrayA.length());

    // Launch the kernel (JIT compile + dispatch)
    cc.dispatchKernel(ndRange, kc -> compute(kc, arrayA, arrayB, arrayC));

}
```

With **NDRange** objects we can express the **number of threads to deploy** and the **block partitions** (optional).

- **NDRange** for **1D**, **2D** and **3D** kernels

How do we invoke the Compute Layer?

```
var accelerator = new Accelerator(Backend.CUDA);



accelerator.compute(cc -> MyClass.computeContext(cc, arrayA, arrayB, arrayC, size));
```

How do we manage data?

```
final int size = 1024;
var accelerator = new Accelerator(Backend.CUDA);
var arrayA = F32Array.create(accelerator, size);



accelerator.compute(cc -> MyClass.computeContext(cc, arrayA, arrayB, arrayC, size));
```

Data is bounded to a specitic accelerator, e.g., a GPU.

HAT Data Objects makes use of Panama FFM. Thus, **data is stored off-heap**

How do we manage data?

```
final int size = 1024;
var accelerator = new Accelerator(Backend.CUDA);
var arrayA = F32Array.create(accelerator, size);
var arrayB = F32Array.create(accelerator, size);
var arrayC = F32Array.create(accelerator, size);

accelerator.compute((@Reflect Compute) cc ->
                    MyClass.computeContext(cc, arrayA, arrayB, arrayC, size));
```

Data is bounded to a specific accelerator, e.g., a GPU.

Whole View

```
public class ExampleHAT {
  @Reflect
  public static void vectorMul(KernelContext kc, F32Array a, F32Array b, F32Array c) {
    if (kc.gix < arrayA.length()) {
         int valueA = arrayA.array(kc.gix);
         int valueB = arrayB.array(kc.gix);
         arrayC.array(kc.gix, (valueA * valueB));
    }
  }

  @Reflect
  public static void compute(ComputeContext cc, F32Array arrayA, F32Array arrayB, F32Array arrayC){
    NDRange ndRange = NDRange.of(Global1D.of(arrayA.length()));
    cc.dispatchKernel(ndRange, kc -> vectorMul(kc, arrayA, arrayB, arrayC));
  }

  public void setupAndLaunch(int size) {
    var accelerator = new Accelerator(lookup(), Backend.FIRST);
    var arrayA = F32Array.create(accelerator, size);
    var arrayB = F32Array.create(accelerator, size);
    var arrayC = F32Array.create(accelerator, size);
    accelerator.compute((@Reflect Compute)
                        cc -> ExampleHAT.compute(cc, arrayA, arrayB, arrayC, size));
  }
}
```

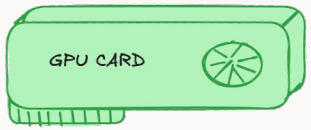
Where is the Code Executed?

```
public class ExampleHAT {
  @Reflect
  public static void vectorMul(KernelContext kc, F32Array a, F32Array b, F32Array c) {
    if (kc.gix < arrayA.length()) {
         int valueA = arrayA.array(kc.gix);
         int valueB = arrayB.array(kc.gix);
         arrayC.array(kc.gix, (valueA * valueB));
    }
  }

  @Reflect
  public static void compute(ComputeContext cc, F32Array a, F32Array b, F32Array b){
    NDRange ndRange = NDRange.of(arrayA.length(), 256);
    cc.dispatchKernel(ndRange, kc -> vectorMul(kc, a, b, c));
  }

  public void setupAndLaunch(int size) {
    var accelerator = new Accelerator(lookup(), Backend.FIRST);
    var arrayA = F32Array.create(accelerator, size);
    var arrayB = F32Array.create(accelerator, size);
    var arrayC = F32Array.create(accelerator, size);
    accelerator.compute((@Reflect Compute)
                        cc -> ExampleHAT.compute(cc, arrayA, arrayB, arrayC));
  }
}
```

Kernel Layer



Compute Layer

Restricted Java

Some other language is also possible

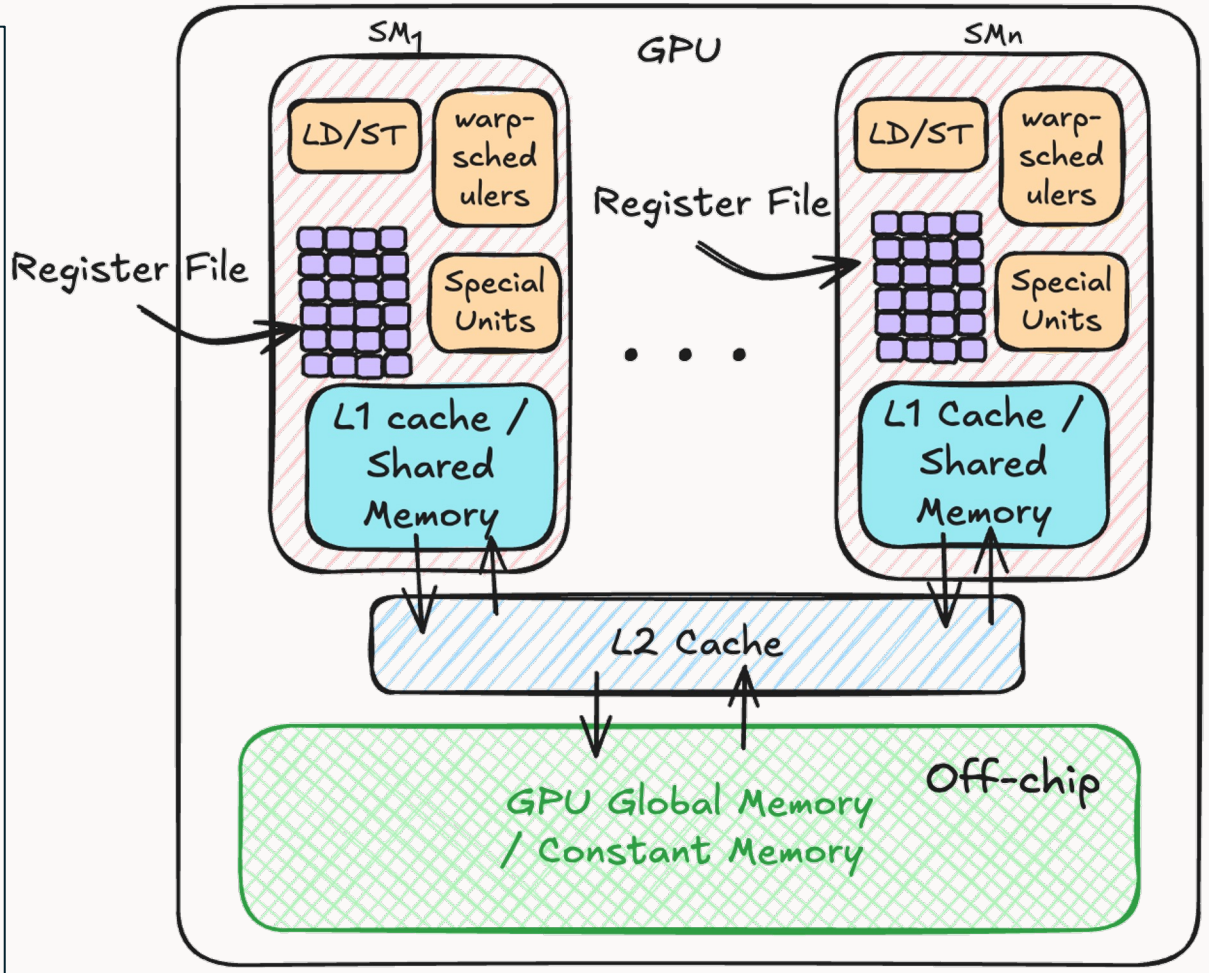
Java application

Let's talk about GPU Memory Hierarchy (Very High-Level Overview)

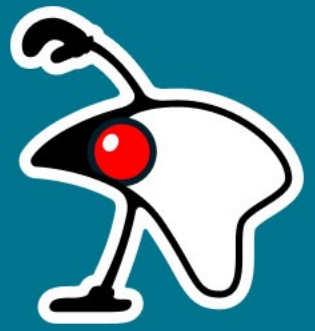
Explicit Memory usages in OpenCL/CUDA
- We can copy data to global memory
- We can copy data to L1 Shared Memory

This is also usable in HAT. It also allows User Defined Types in:
- GPU's global memory with Mem Segments
- L1 Shared Memory
- Constant memory (in progress)

- Map to Memory Segment and **GPU's global Memory** → **MappableIface**
- Map to GPU's **private and shared Memory** → **NonMappableIface**



For more details about GPU architectures:
https://www.aleksagordic.com/blog/matmul



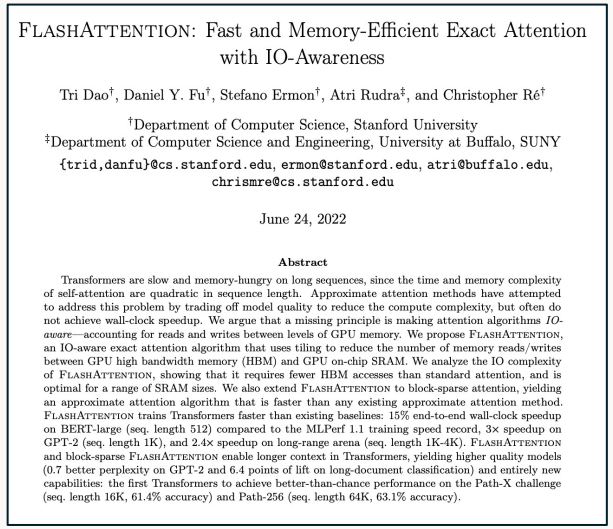
Demo

Flash Attention v1

Demo Running Flash-Attention v1 on Apple Silicon M4

Technique that optimises classical attention with:

- **Tiling computation** (partition inputs to fit in shared memory)
- **Fusing operations** (matmul and softmax)



FLASHATTENTION: Fast and Memory-Efficient Exact Attention with IO-Awareness

Tri Dao†, Daniel Y. Fu†, Stefano Ermon†, Atri Rudra‡, and Christopher Ré†

†Department of Computer Science, Stanford University
‡Department of Computer Science and Engineering, University at Buffalo, SUNY
{trid,danfu}@cs.stanford.edu, ermon@stanford.edu, atri@buffalo.edu, chrismre@cs.stanford.edu

June 24, 2022

Abstract

Transformers are slow and memory-hungry on long sequences, since the time and memory complexity of self-attention are quadratic in sequence length. Approximate attention methods have attempted to address this problem by trading off model quality to reduce the compute complexity, but often do not achieve wall-clock speedup. We argue that a missing principle is making attention algorithms *IO-aware*—accounting for reads and writes between levels of GPU memory. We propose FLASHATTENTION, an IO-aware exact attention algorithm that uses tiling to reduce the number of memory reads/writes between GPU high bandwidth memory (HBM) and GPU on-chip SRAM. We analyze the IO complexity of FLASHATTENTION, showing that it requires fewer HBM accesses than standard attention, and is optimal for a range of SRAM sizes. We also extend FLASHATTENTION to block-sparse attention, yielding an approximate attention algorithm that is faster than any existing approximate attention method. FLASHATTENTION trains Transformers faster than existing baselines: 15% end-to-end wall-clock speedup on BERT-large (seq. length 512) compared to the MLPerf 1.1 training speed record, 3× speedup on GPT-2 (seq. length 1K), and 2.4× speedup on long-range arena (seq. length 1K-4K). FLASHATTENTION and block-sparse FLASHATTENTION enable longer context in Transformers, yielding higher quality models (0.7 better perplexity on GPT-2 and 6.4 points of lift on long-document classification) and entirely new capabilities: the first Transformers to achieve better-than-chance performance on the Path-X challenge (seq. length 16K, 61.4% accuracy) and Path-256 (seq. length 64K, 63.1% accuracy).

https://arxiv.org/pdf/2205.14135

```
$ java -cp hat/job.jar hat.java \
        run ffi-opencl \
        flashattention --size=2048

Speedups vs Streams:
Java Streams / HAT-Self-Attention          = 1.22x
Java Streams / HAT-Flash-Attention         = 4.56x
Java Streams / HAT-Flash-Attention (FP16) = 5.89x


Speedups vs Java:
Java / Java Parallel Stream        = 10.6x
Java / HAT-Self-Attention          = 12.9x
Java / HAT-Flash-Attention         = 48.31x
Java / HAT-Flash-Attention (FP16) = 62.35x
```

Running on Apple Silicon M4 Laptop

Looking Under the HAT

How do we use code reflection for targeting foreign programming models?

Heterogeneous Accelerator Toolkit (HAT)

HAT exploits the availity to specialize (dialectify) a code-model specifically designed for GPU computing.

HAT introduces a **dialect** in code-reflection **for hardware accelerators** that includes:

- GPU Thread access
- GPU barriers
- Allocations in different memory hiearchies of the accelerator
- Data access data from/to different levels of the memory hierarchy
- Operations in Special Types (e.g., bfloat16, float16, explicit vector types)
- Vector Operations, etc

Inspired by MLIR and its GPU dialects: https://mlir.llvm.org/docs/Dialects/GPU/

We are working on bringing a **minimal set of Ops that allows developers to create backends for hardware accelerators**

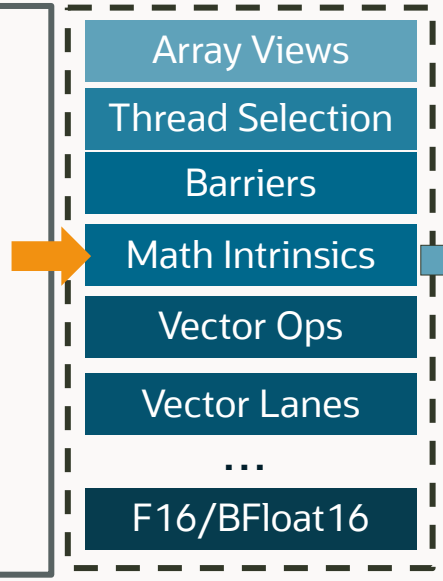
Modeling Java Programs for GPUs with Dialects

Code reflected

```
…
final int linearLocalId = kc.liy * kc.lsx + kc.lix;
final int threadCol = kc.lix;
final int threadRow = kc.liy;

SharedMemory tileA = SharedMemory.createLocal();
SharedMemory tileB = SharedMemory.createLocal();
…
```


```
%37 : java.type:"int" = field.load %36 hat.KernelContext::liy:int
%38 : java.type:"hat.KernelContext" = var.load %5
%39 : java.type:"int" = field.load %38 hat.KernelContext::lsx:int
%40 : java.type:"int" = mul %37 %39
%41 : java.type:"hat.KernelContext" = var.load %5
%42 : java.type:"int" = field.load %41 hat.KernelContext::lix:int
%43 : java.type:"int" = add %40 %42
%44 : Var<java.type:"int"> = var %43 @"linearLocalId";
%45 : java.type:"hat.KernelContext" = var.load %5
%46 : java.type:"int" = field.load %45 hat.KernelContext::lix:int
%47 : Var<java.type:"int"> = var %46 @"threadCol";
%48 : java.type:"hat.KernelContext" = var.load %5
%49 : java.type:"int" = field.load %48 hat.KernelContext::liy:int
%50 : Var<java.type:"int"> = var %49 @"threadRow";
%51 : matmul.Main$SharedMemory = invoke SharedMemory::createLocal
%52 : matmul.Main$SharedMemory = var %51 @"tileA";
%53 : matmul.Main$SharedMemory = invoke SharedMemory::createLocal
```



HAT Dialect

```
%34 : java.type:"int" = hat.dialect.HATThreadOp$HAT_LI$HAT_LIY
%35 : java.type:"int" = hat.dialect.HATThreadOp$HAT_LS$HAT_LSX
%36 : java.type:"int" = mul %34 %35
%37 : java.type:"int" = hat.dialect.HATThreadOp$HAT_LI$HAT_LIX
%38 : java.type:"int" = add %36 %37
%39 : Var<java.type:"int"> = var %38 @"linearLocalId";
%40 : java.type:"int" = hat.dialect.HATThreadOp$HAT_LI$HAT_LIX
%41 : Var<java.type:"int"> = var %40 @"threadCol";
%42 : java.type:"int" = hat.dialect.HATThreadOp$HAT_LI$HAT_LIY
%43 : Var<java.type:"int"> = var %42 @"threadRow";
%44 : Var hat.dialect.HATMemoryVarOp$HATLocalVarOp
%45 : Var hat.dialect.HATMemoryVarOp$HATLocalVarOp
```

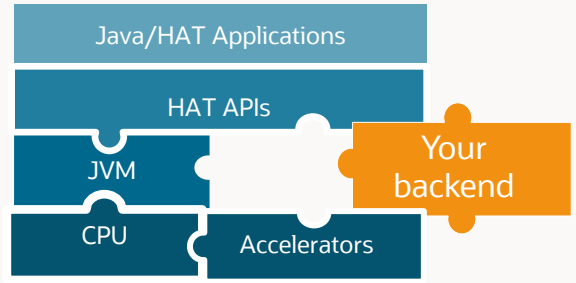
HAT Dialectify Process

What's the Value of HAT?

What this Toolkit really means for developers

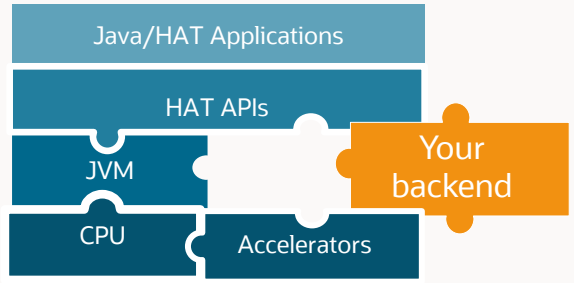
What is the Value of HAT for Java Developers?

A **pluggable backend system** for targeting hardware accelerators, with a default Java impl.



What is the Value of HAT for Java Developers?

A **pluggable backend system** for targeting hardware accelerators, with a default Java impl.

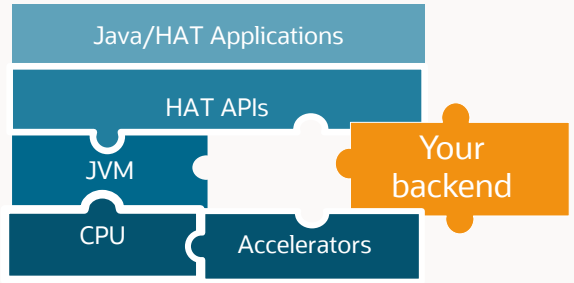


An **extensible type system** for your foreign programming models

```
interface MyType extends MappableIface/NonMappableIface
```

What is the Value of HAT for Java Developers?

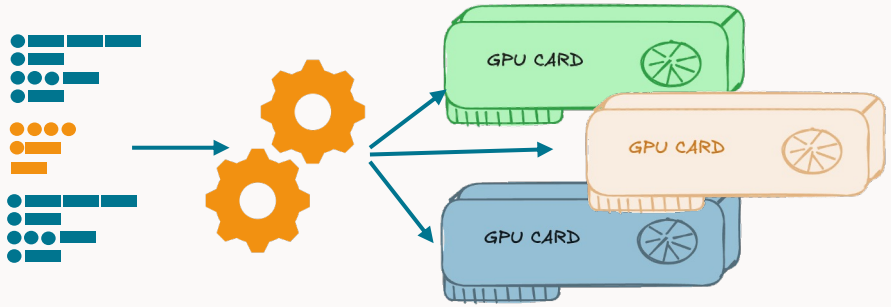
A **pluggable backend system** for targeting hardware accelerators, with a default Java impl.



An **extensible type system** for your foreign programming models

```
interface MyType extends MappableIface/NonMappableIface
```

A **extensible Java Code Model to target** foreign programming environments



Cost of User Data Structures on GPUs

Case Study

Understanding Cost of Data Abstractions

```
static void dftKernel(
   KernelContext kc,
   ComplexArray input,
   ComplexArray output) {
 …
 Complex c = input.complex(k);
 sumReal+=(c.real()*cReal)-(c.imag()*cImag);
 sumImag+=(c.real()*cImag)+(c.imag()*cReal);
…
}
```

HAT User Defined Types (UDT)

vs

```
static void dftPlain(
     KernelContext kc,
     F32Array inReal,
     F32Array inImag,
     F32Array outReal,
     F32Array outImag) {
 …
  sumReal += (inReal.array(k) * cReal)-(inImag.array(k) * cImag);
  sumImag += (inReal.array(k) * cImag)+(inImag.array(k) * cReal);
 …
}
```

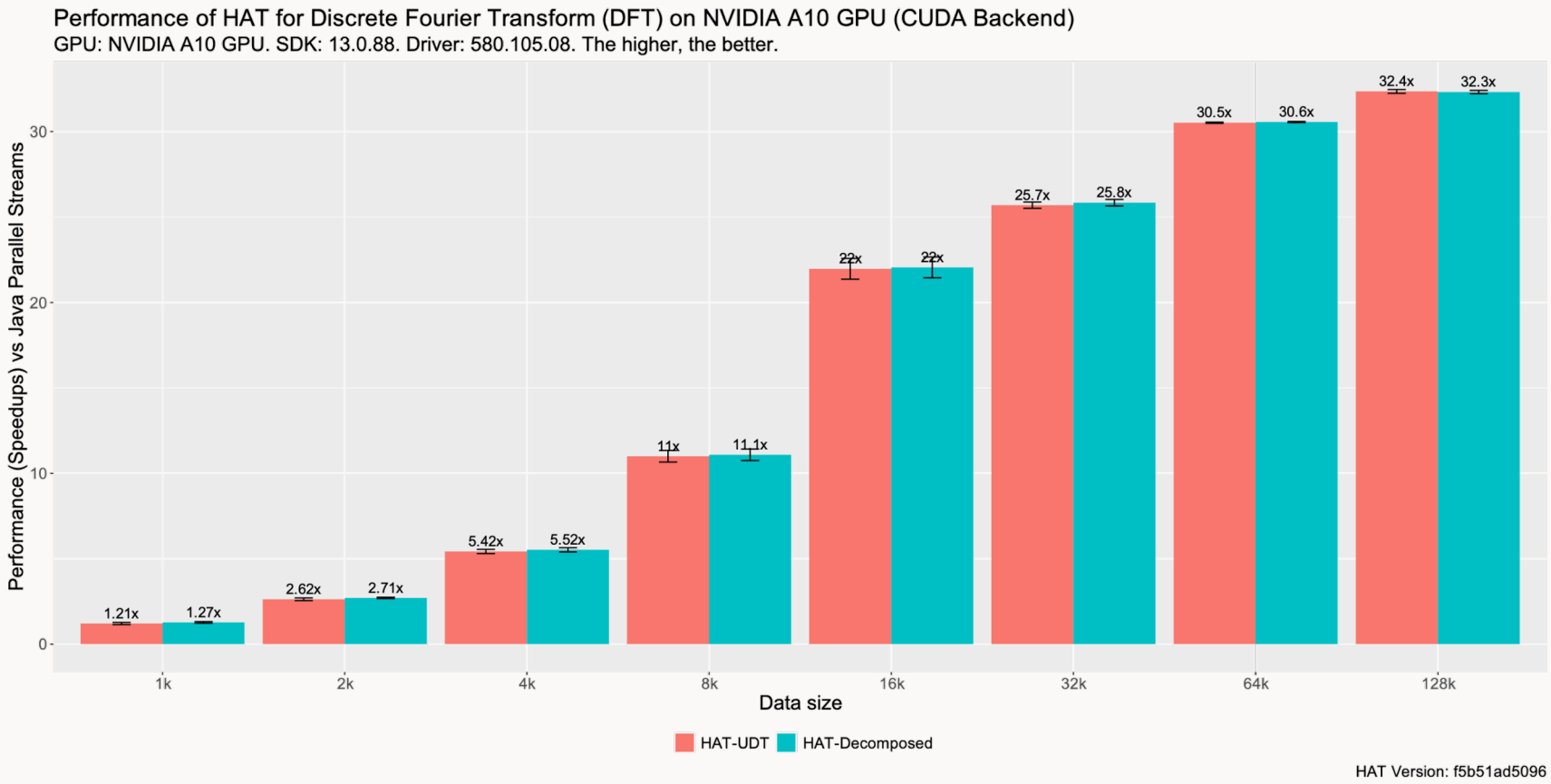
HAT Separated Arrays (decomposition)

```
Complex c1 = array.complex(i);
c1.real(2.1f);
```

ComplexArray is a User Defined Type **vs** Making use of explicit array types in HAT

This case is simple, but Java programs usually have more complex that ideally we want to port to GPUs

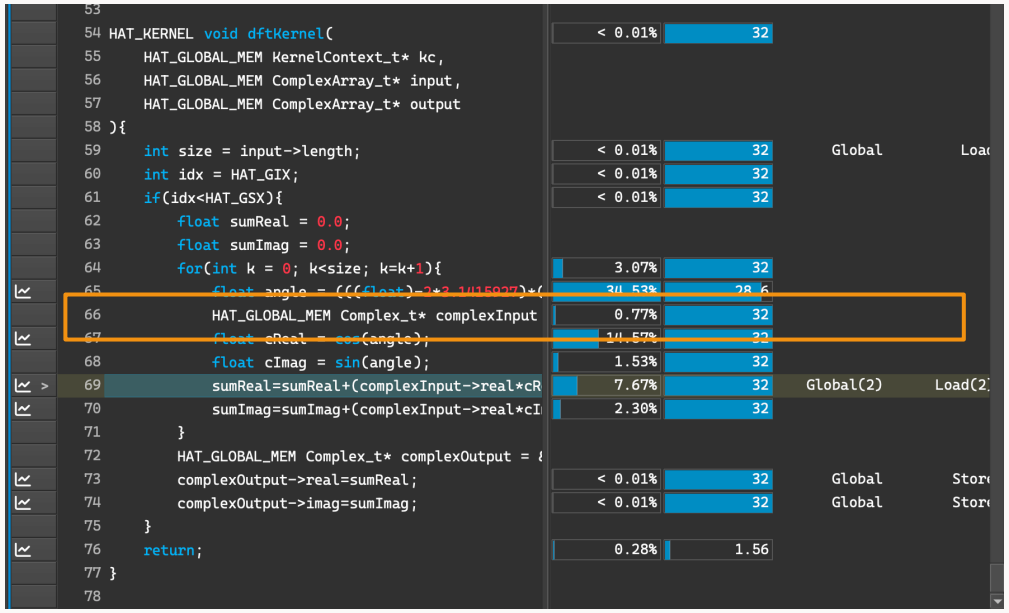
Cost of Data Types Abstractions



As end-2-end time, which includes copies, kernel and runtime overheads, there is no difference in performance.

Running DFT on a A10 NVIDIA GPU.
UDT→ User Data Types
Decomposed → Separate HAT Arrays (no abtractions)

A Closer Look with the NVIDIA Kernel Profiler



Profiler using NVIDIA Nsight Compute:
https://developer.nvidia.com/nsight-compute

From the CUDA Profiler, we see that ~1% is spent in loading the struct for large data sets

```
HAT_GLOBAL_MEM Complex_t* complexInput = &input->complex[(long)k];
```

For small/medium input sizes, this is negligible.

Kernel Time Measured with NVIDIA NCU Profiler

There is no penalty for small and medium data sizes.

For large datasets, since we need to perform an object load, if we are not careful, there is a small penalty (0.5 ms) in this case. But, with algorithm redesign, we can avoid it.

How far can we go with code reflection?

Case Study: Matrix Multiplication for GPUs

Optimizing Matrix Multiplication for GPUs from Java

Why matmul?

- Core kernel for many AI applications (LLMs)
- Used of advanced constructs that are useful for other applications

```
static void matmul(F32Array matrixA, F32Array matrixB, F32Array matrixC, final int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            float sum = 0;
            for (int k = 0; k < size; k++) {
                float a = matrixA.array(i * size + k);
                float b = matrixB.array(k * size + j);
                sum += a * b;
            }
            matrixC.array(i * size + j, sum);
        }
    }
}
```

Machine Setup: OCI instance with an NVIDIA A10 GPU

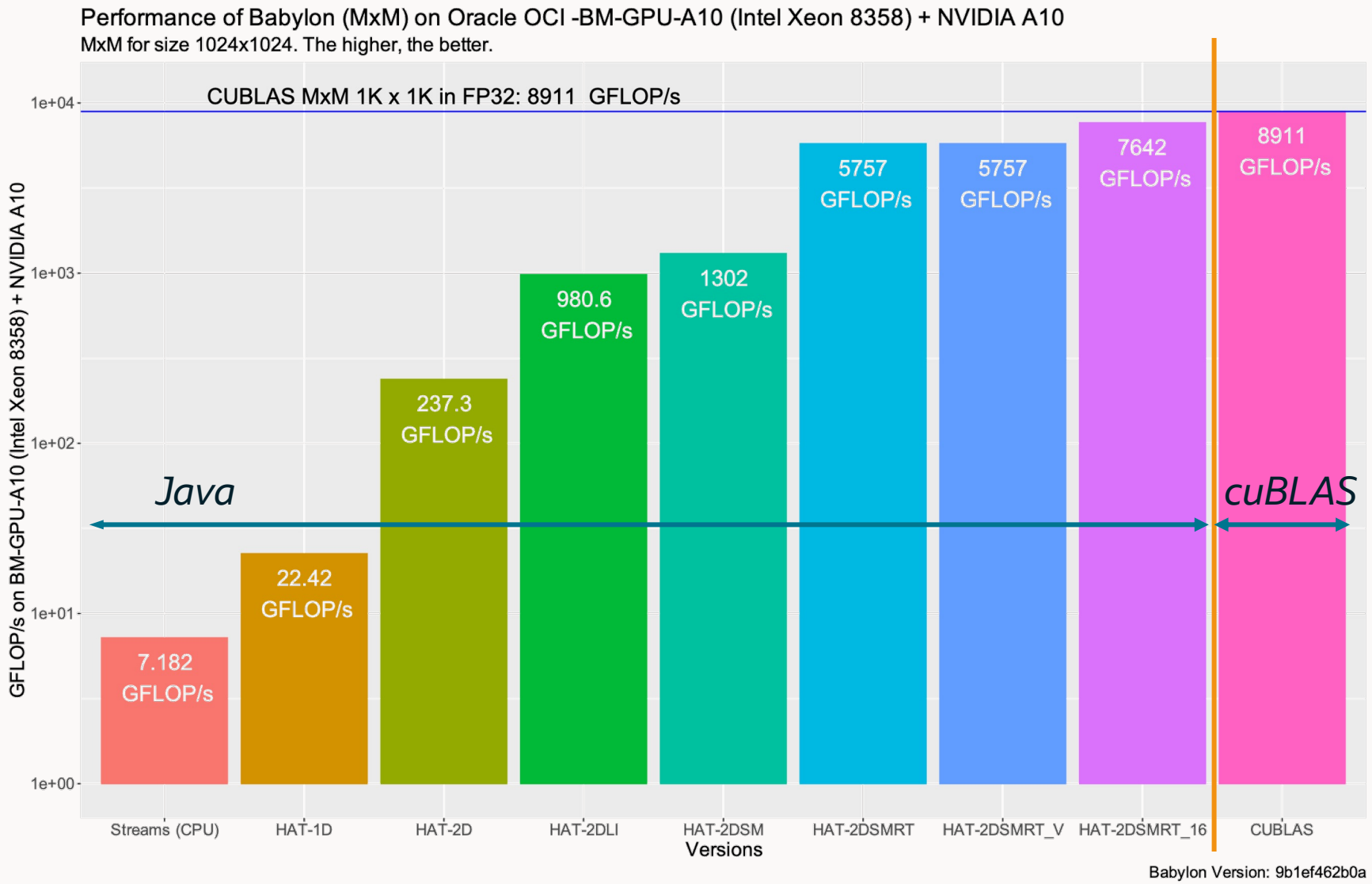
System Component	Version
OCI Instance	BM.GPU.A10
CPU	Intel Xeon Platinum 8358 CPU @ 2.60GHz
System RAM	236 GB
GPU	NVIDIA A10
OS	Ubuntu 22.04.5 LTS
Linux Kernel	6.8.0-1039-oracle
NVIDIA Driver	580.105.08
CUDA SDK	13.0.88
Java Version (Babylon)	9b1ef462b0a
JDK Base Version	26.ea.10-open (downloaded from sdkman)
Application	**Matrix-Multiplication**
Matrix Sizes	1024x1024

```
java @hat/run ffi-cuda \
 matmul \
  --kernel=<VERSION> \
  --iterations=100 \
  --size=<SIZE>
```

Detailed article about this analysis:

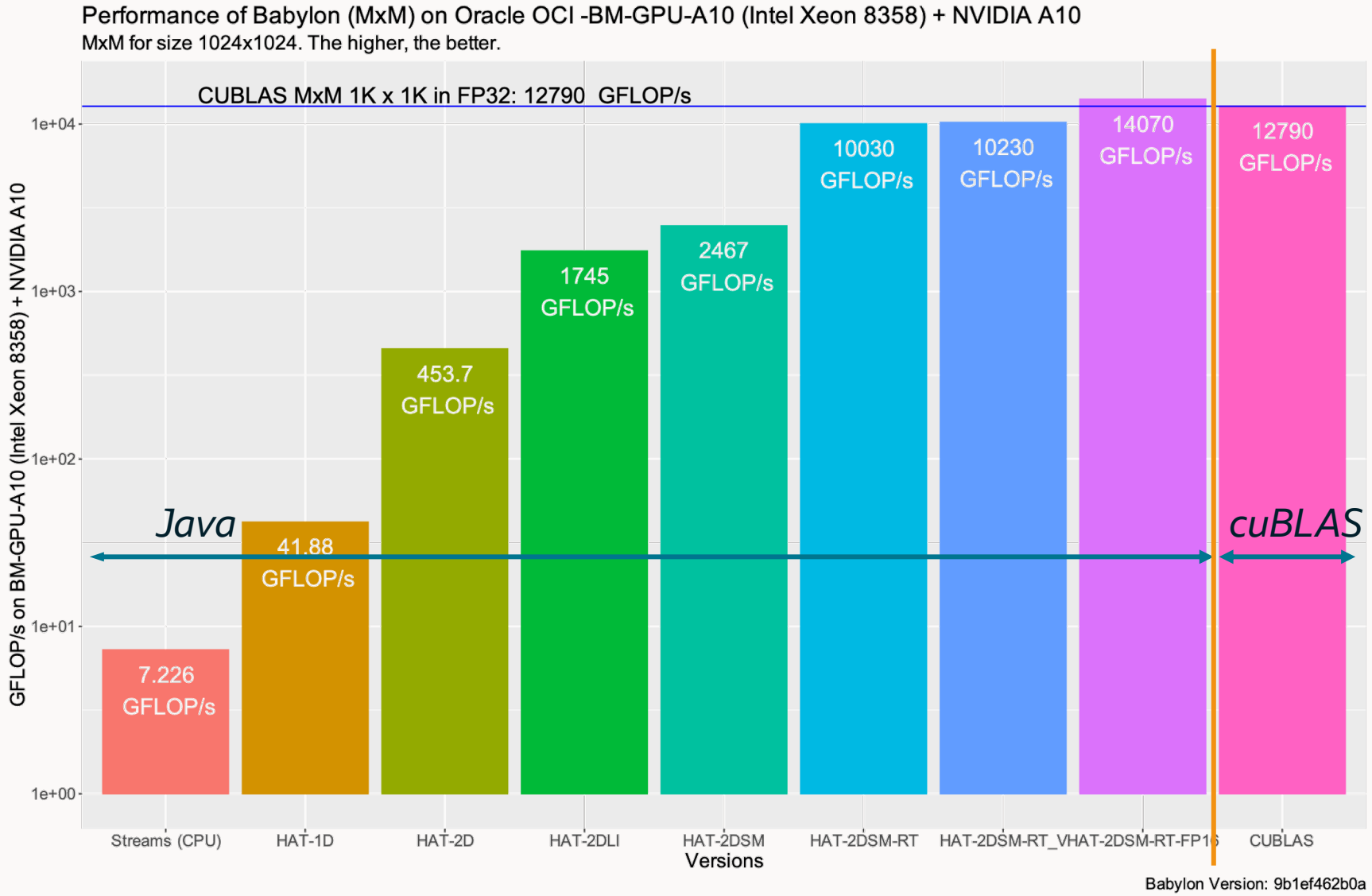
https://openjdk.org/projects/babylon/articles/hat-matmul/hat-matmul

Performance with Nvidia Nsight Compute (NCU)



Java:	299ms
1D:	95ms (+3x)
2D:	9ms (+10x)
2DLI:	2.1ms (+4x)
2DSM:	1.6ms (+1.33x)
2DSMRT:	373 us (+5.8x)
2DSMRT_v:	373 us (+0x)
2DSMRT_f16:	281 us (+1.32x)
cuBLAS:	244 us

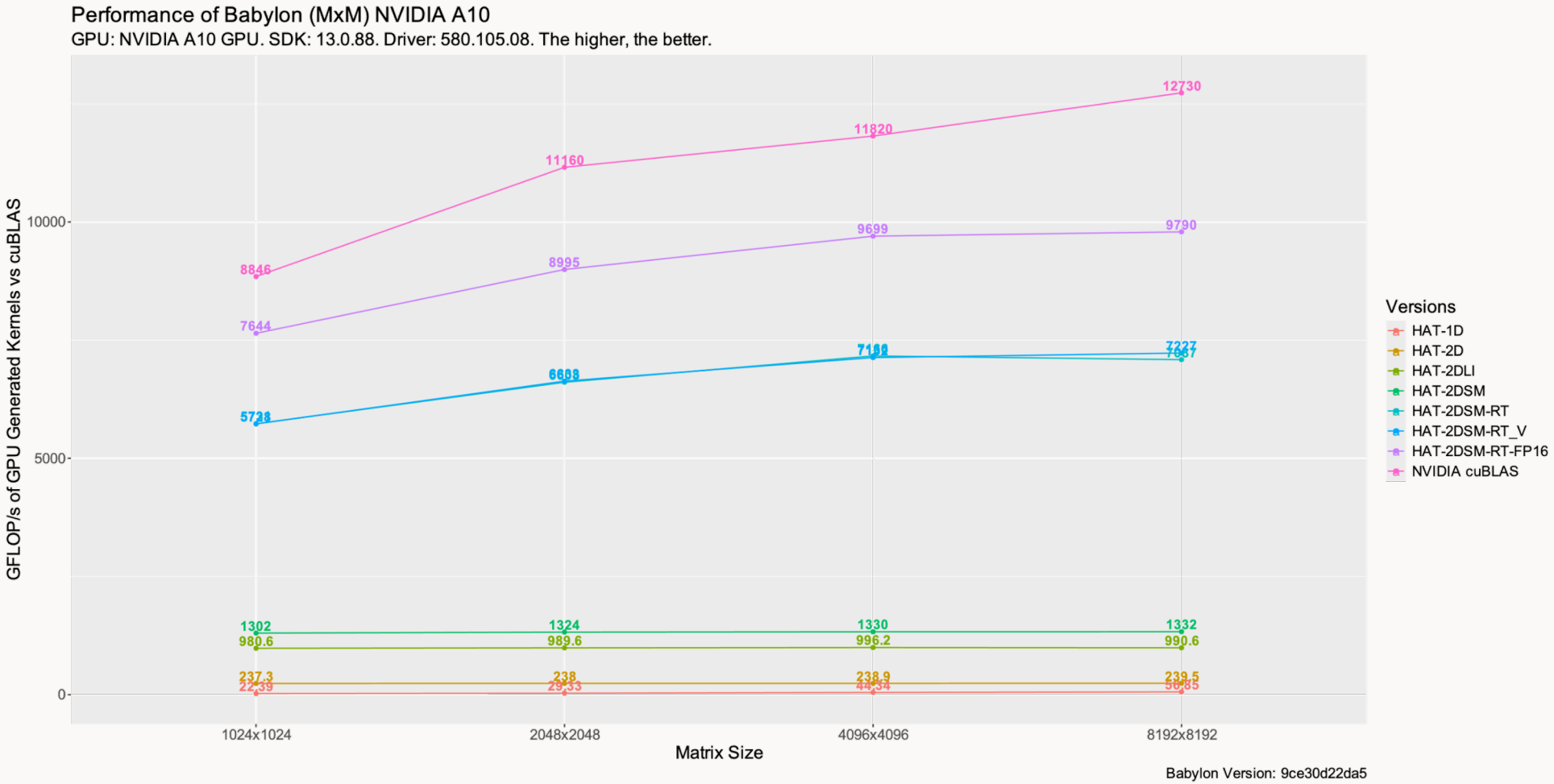
Performance with CUDA Events



With CUDA Events, we don't fix the frequency or flash the caches.
This looks like closer to real-world executions.

Up to 10TFlops vs 12.7 in cuBLAS for 1k x 1k matrix sizes.

Running with Large Matrix Sizes v2

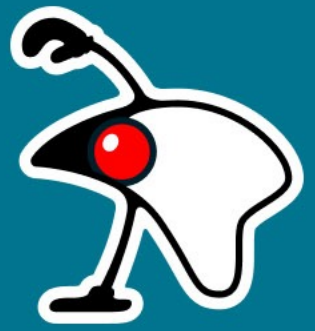


HAT Generated CUDA Kernels perform **60%** of the CUDA cuBLAS library.
HAT does not currently support tensors or warp operations.

What’s next?

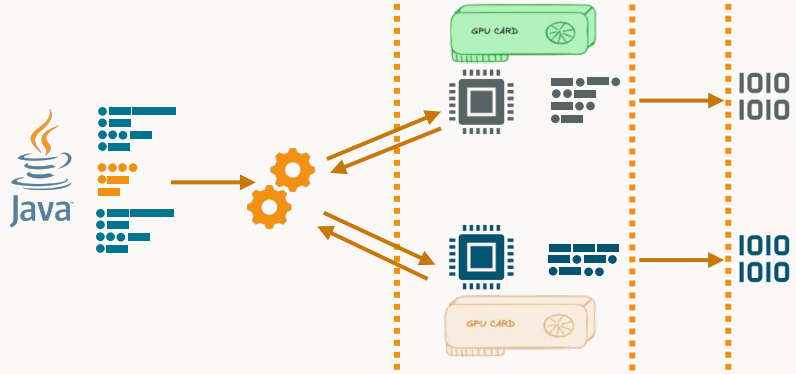
What next?

- Code Reflection is under incubation process (without the GPU port)
- Working on Dialects for Hardware Accelerators and Code Reflection
- Working on lifting low-level programming and providing programming abstractions for leveraging GPU programming from Java



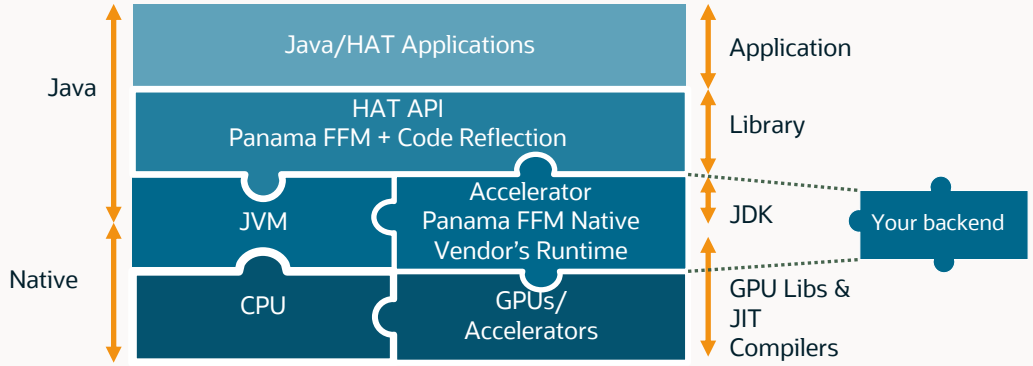
Conclusions

Takeaways

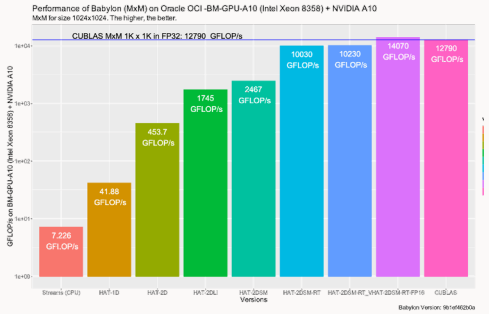


HAT tackles all SW-Stack: From APIs to CodeGen

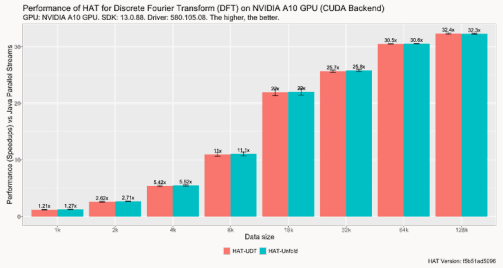
Write once, run everywhere



Extensible System for Hardware Acceleration in Java



Case Study:
From 7GLOP/s
To 14TFLOP/s



Minimal Cost for
User Defined Data
Abstractions

JEP draft: Code reflection (Incubator)

Owner	Paul Sandoz
Type	Feature
Scope	JDK
Status	Draft
Component	core-libs
Effort	L
Duration	L
Reviewed by	Adam Sotona, Gary Frost, Juan Fumero, Maurizio Cimadamore
Created	2025/06/30 19:54
Updated	2026/02/16 15:49
Issue	8361105

Summary

Enhance the core reflection API to model Java code, build and transform models of Java code, and access models of Java code in methods and lambda expressions. Libraries can use this enhancement to analyze Java code and extend its reach, such as executing it as code on GPUs. This is an incubating API.

Goals

1. Enable Java developers to interface with non-Java (foreign) programming models using familiar Java language constructs, such as lambda expressions and static typing.
2. Encourage libraries to expose novel programming models to Java developers without requiring developers to embed non-Java code inside Java code, or to write tedious Java code that builds data structures to model Java code or other (foreign) code.

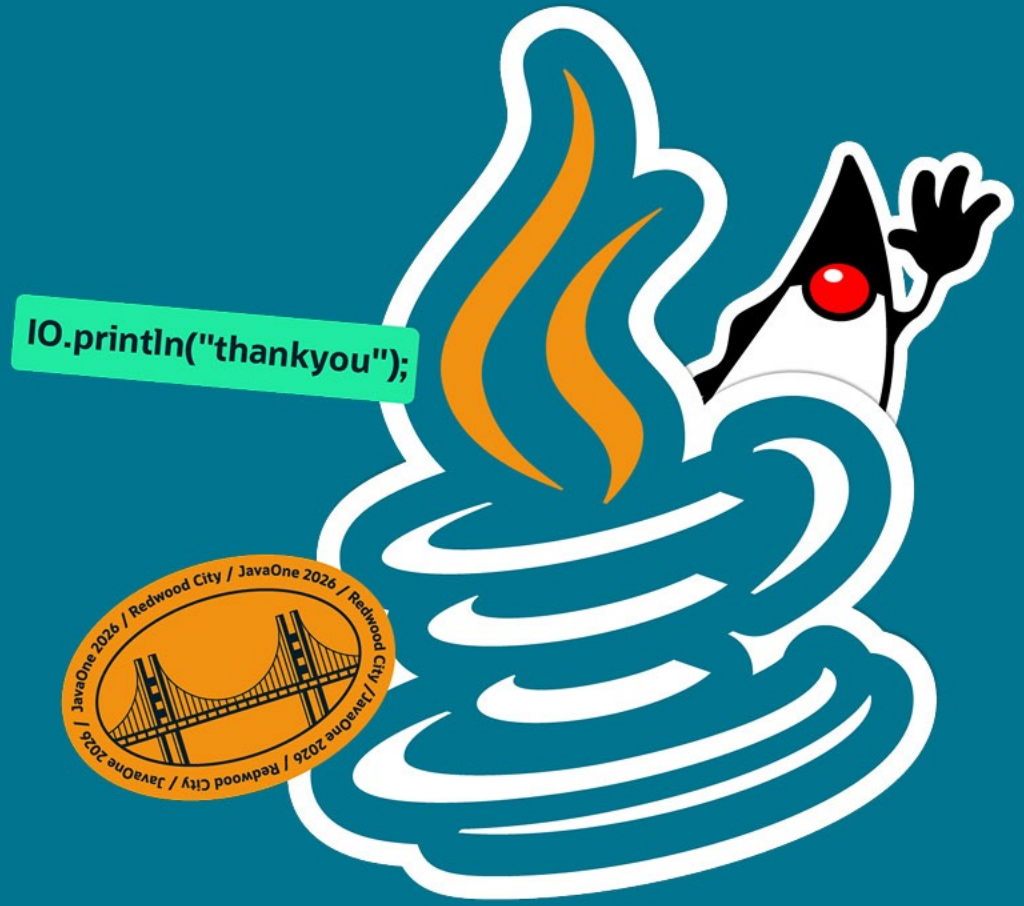
Working Towards
Incubation

(Just Code Reflection)

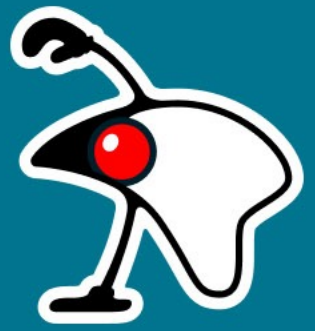
Thank you

Juan Fumero

juan.fumero@oracle.com




ORACLE



Backup Slides

How to Build OpenJDK/Babylon?

Build JDK

```
$ git clone https://github.com/openjdk/babylon
$ cd babylon
$ bash configure --with-boot-jdk=${JAVA_HOME}
$ make clean
$ make images
```

Update JAVA_HOME and PATH

```
$ export JAVA_HOME=<BABYLON-DIR>/build/macosx-aarch64-server-release/jdk/
$ export PATH=$JAVA_HOME/bin:$PATH
```

Enable Code-Reflection (for Incubation)

```
$ java --add-modules jdk.incubator.code MyClass
```

How to Build Babylon and HAT?

Build JDK

```
$ git clone https://github.com/openjdk/babylon
$ cd babylon
$ bash configure --with-boot-jdk=${JAVA_HOME}
$ make clean
$ make images
```

Update JAVA_HOME and PATH

```
$ export JAVA_HOME=<BABYLON-DIR>/build/macosx-aarch64-server-release/jdk/
$ export PATH=$JAVA_HOME/bin:$PATH
```

Build HAT

```
$ sdk install jextract  # if needed
$ cd hat
$ java @.bld
```

Run Examples (e.g., matmul)

```
$ java @.run ffi-opencl matmul
```

Run Tests (OpenCL and CUDA Backends)

```
$ java @.test-suite ffi-opencl
$ java @.test-suite ffi-cuda
```

End-To-End Performance (including copies)