ORACLE

Java Code Reflection for Targeting Foreign Programming Languages

Enabling GPU Acceleration from Java

Juan Fumero

Consulting Member of Technical Staff @ Java Platform Group

February 23rd, 2026

Venue: Seminar at The University of Salerno, Italy

Outline

1 Project Babylon and Code Reflection

2 Heterogeneous Accelerator Toolkit (HAT)

3 Compiling to GPUs via Dialects in Code Reflection

4 Cost of Data Abstractions

5 Optimizing Matrix Multiplication for GPUs from Java

6 Discussions

Project Babylon

Deep dive into Code Reflection

Example: Using Java Reflection

```
Class<?> testClass = Class.forName(myClassName);

List<Method> listOfMethods = new ArrayList<>();

for (Method m : testClass.getDeclaredMethods()) {
  Annotation[] markers = m.getDeclaredAnnotations();

  for (Annotation a : markers) {
    if (a.annotationType().equals(MyAnnotation.class)) {
      listOfMethods.add(m);
    }
  }
}
```

Find all Java annotations from a given class and compared with the desired annotation.

If successful, then add the method to a list.

Common operations (e.g., frameworks for testing).

But can we inspect the actual method at runtime?

Currently, NO

Example of current Java Reflection

```
Class<?> testClass = Class.forName(myClassName);

List<Method> listOfMethods = new ArrayList<>();

for (Method m : testClass.getDeclaredMethods()) {
  Annotation[] markers = m.getDeclaredAnnotations();

  for (Annotation a : markers) {
    if (a.annotationType().equals(MyAnnotation.class)) {
      listOfMethods.add(m);
    }
  }
}
```

Find all Java annotations from a given class and compared with the desired annotation.

If successful, then add the method to a list.

Common operations (e.g., frameworks for testing).

But can we inspect the actual method code at runtime?

Currently, NO

This is why **code-reflection** comes in!

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 @loc="67:5" @"a";
    %4 : Var<java.type:"float"> = var %1 @loc="67:5" @"b";
    %5 : Var<java.type:"float"> = var %2 @loc="67:5" @"c";
    %6 : java.type:"float" = var.load %3 @loc="69:16";
    %7 : java.type:"float" = var.load %4 @loc="69:20";
    %8 : java.type:"float" = mul %6 %7 @loc="69:16";
    %9 : java.type:"float" = var.load %5 @loc="69:24";
    %10 : java.type:"float" = add %8 %9 @loc="69:16";
    return %10 @loc="69:9";
};
```

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 @loc="67:5" @"a";
    %4 : Var<java.type:"float"> = var %1 @loc="67:5" @"b";
    %5 : Var<java.type:"float"> = var %2 @loc="67:5" @"c";
    %6 : java.type:"float" = var.load %3 @loc="69:16";
    %7 : java.type:"float" = var.load %4 @loc="69:20";
    %8 : java.type:"float" = mul %6 %7 @loc="69:16";
    %9 : java.type:"float" = var.load %5 @loc="69:24";
    %10 : java.type:"float" = add %8 %9 @loc="69:16";
    return %10 @loc="69:9";
};
```

```
func @"compute" (%0 : java.type:"float", %1 : java.type:"float", %2 :
java.type:"float")java.type:"float" -> {
    %3 : Var<java.type:"float"> = var %0 @loc="67:5" @"a";
    %4 : Var<java.type:"float"> = var %1 @loc="67:5" @"b";
    %5 : Var<java.type:"float"> = var %2 @loc="67:5" @"c";
    %6 : java.type:"float" = var.load %3 @loc="69:16";
    %7 : java.type:"float" = var.load %4 @loc="69:20";
    %8 : java.type:"float" = var.load %5 @loc="69:24";
    %9 : java.type:"float" = fma %6 %7 %8 @loc="69:16";
    return %9 @loc="69:9";
};
```

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

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
```

Code Reflection Jargon

Code Model: a symbolic representation of the Java method/Java lambda.

Code Element: an interface representing a single element from a code tree.

Operation (Op): nodes in the code tree. It implements code elements

Block/Body: a set of operations associated to a block (e.g., a for-loop, a function, etc.).

Dialect: a domain specific code model without modifying the core Java code models. It can be seen as an extention for a specific domain of applications.

Value: value associated with a code element. E.g., a parameter for an invoke op.

Type Element: code item to model the set of possible values of an element.

Transformation: modification of a code model. All code-models are immutable. Thus, transformations return a new code model.

Example 1: Hello Code Reflection

```
@Reflect
private double myFunction(int value) {
    return Math.pow(value, 2);
}
```

```
var myFunction = Stream.of(Hello.class.getDeclaredMethods())
  .filter(m -> m.getName().equals("myFunction"))
  .findFirst();
Method m = myFunction.get();

CoreOp.FuncOp codeModel = Op.ofMethod(m).get();
IO.println(codeModel.toText());
```

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

Example 1: Hello Code Reflection

```
@Reflect
private double myFunction(int value) {
    return Math.pow(value, 2);
}
```

codeModel.toText()

```
func @loc="71:5:file:…/oracle/code/samples/HelloCodeReflection.java" @"myFunction" (%0 :
java.type:"oracle.code.samples.HelloCodeReflection", %1 : java.type:"int")java.type:"double" -> {
    %2 : Var<java.type:"int"> = var %1 @loc="71:5" @"value";
    %3 : java.type:"int" = var.load %2 @loc="73:25";
    %4 : java.type:"double" = conv %3 @loc="73:16";
    %5 : java.type:"int" = constant @loc="73:32" @2;
    %6 : java.type:"double" = conv %5 @loc="73:16";
    %7 : java.type:"double" = invoke %4 %6 @loc="73:16" @java.ref:"java.lang.Math::pow(double,
double):double";
    return %7 @loc="73:9";
};
```

Example 1: Code Model to SSA Representation

```
@Reflect
private double myFunction(int value) {
    return Math.pow(value, 2);
}
```

```
CoreOp.FuncOp ssaCodeModel = SSA.transform(codeModel);
```

```
func @loc="71:5:file…/oracle/code/samples/HelloCodeReflection.java" @"myFunction" (%0 :
java.type:"oracle.code.samples.HelloCodeReflection", %1 : java.type:"int")java.type:"double" -> {
    %2 : java.type:"double" = conv %1 @loc="73:16";
    %3 : java.type:"int" = constant @loc="73:32" @2;
    %4 : java.type:"double" = conv %3 @loc="73:16";
    %5 : java.type:"double" = invoke %2 %4 @loc="73:16" @java.ref:"java.lang.Math::pow(double,
double):double";
    return %5 @loc="73:9";
};
```

Example 1: Running in the Java BC Interpreter

```
MethodHandle mhandle = BytecodeGenerator
                      .generate(MethodHandles.lookup(), ssaCodeModel);

var result = mhandle.invoke(obj, 10);
System.out.println("Result from bytecode generation: " + result);
```

Recap for the first example

1. Build the code-model for a Java instance method.
2. Lower to SSA representation.
3. Build a `methodHandle` to run on the Java bytecode interpreter.
4. Run the inspected & lowered (SSA) Java method in the bytecode interpreter.

BUT, we could run a specialised version in the Java BC interpreter?.
Let’s do that next.

How do we do transforms?

```
CoreOp.FuncOp b = a.transform((blockBuilder, node) -> {
    blockBuilder.op(node);
    return blockBuilder;
});
```

Transform Code Model A -> B (Just a copy)

```
CoreOp.FuncOp transform = codeModel.transform(
 (blockBuilder, node) -> {
    if (node instanceof JavaOp.InvokeOp invokeOp) {
        // replace/process node tree
    } else {
        // insert original node into the new tree
        blockBuilder.op(node);
    }
    return blockBuilder;
});
```

Find specifc nodes and process data/perform replacements

Example 2: “Math Optimizer” for the Interpreter

```
@Reflect
private static double myFunction(int value) {
    return Math.pow(2, value);
}
```

Let’s replace the Math.pow function for an optimize function using the code reflection API. The optimized function can be applied under some conditions:

- Replace Math.pow(x,y) when x==2 to (1 << y), if only if the “y” parameter is an integer.
- Replace Math.pow(x,y) when y==2 to x*x

Let’s jump to the IDE to follow the example:
https://github.com/openjdk/babylon/blob/code-reflection/cr-examples/samples/src/main/java/oracle/code/samples/MathOptimizer.java

Example 3 - Creating Own Dialect: an FMA Op

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

```
// Custom Node inherits from Op
private static class FMA extends Op {
  …
}
```

```
FMA myFMA = new FMA(…);
Op.Result resultFMA = builder.op(myFMA);
context.mapValue(addOp.result(), resultFMA);
```

- We can analyse the code models and extends with new nodes/custom nodes in the tree → Dialect
- This is similar to LLVM/MLIR dialects

Example to follow:
https://github.com/openjdk/babylon/blob/code-reflection/cr-examples/samples/src/main/java/oracle/code/samples/DialectFMAOp.java

Example 3 – Code Models Before/After the FMA Transform

```
func ”DialectFMAOp.java"
@"compute" (%0 : java.type:"float",
            %1 : java.type:"float",
            %2 : java.type:"float")java.type:"float" -> {
    %3 : Var<java.type:"float"> = var %0 @loc="67:5" @"a";
    %4 : Var<java.type:"float"> = var %1 @loc="67:5" @"b";
    %5 : Var<java.type:"float"> = var %2 @loc="67:5" @"c";
    %6 : java.type:"float" = var.load %3 @loc="69:16";
    %7 : java.type:"float" = var.load %4 @loc="69:20";
    %8 : java.type:"float" = mul %6 %7 @loc="69:16";
    %9 : java.type:"float" = var.load %5 @loc="69:24";
    %10 : java.type:"float" = add %8 %9 @loc="69:16";
    return %10 @loc="69:9";
};
```

```
func “DialectFMAOp.java"
@"compute" (%0 : java.type:"float",
            %1 : java.type:"float",
            %2 : java.type:"float")java.type:"float" -> {
    %3 : Var<java.type:"float"> = var %0 @loc="67:5" @"a";
    %4 : Var<java.type:"float"> = var %1 @loc="67:5" @"b";
    %5 : Var<java.type:"float"> = var %2 @loc="67:5" @"c";
    %6 : java.type:"float" = var.load %3 @loc="69:16";
    %7 : java.type:"float" = var.load %4 @loc="69:20";
    %8 : java.type:"float" = var.load %5 @loc="69:24";
    %9 : java.type:"float" = fma %6 %7 %8 @loc="69:16";
    return %9 @loc="69:9";
};
```

Then, we can lower to something else, or generate code for a foreign programming model

Code Reflection is in Incubation Process (draft)

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) 8361106

- 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 **babylon-dev** mailing list from OpenJDK:
- https://mail.openjdk.org/pipermail/babylon-dev/

What else can we do with code-reflection?

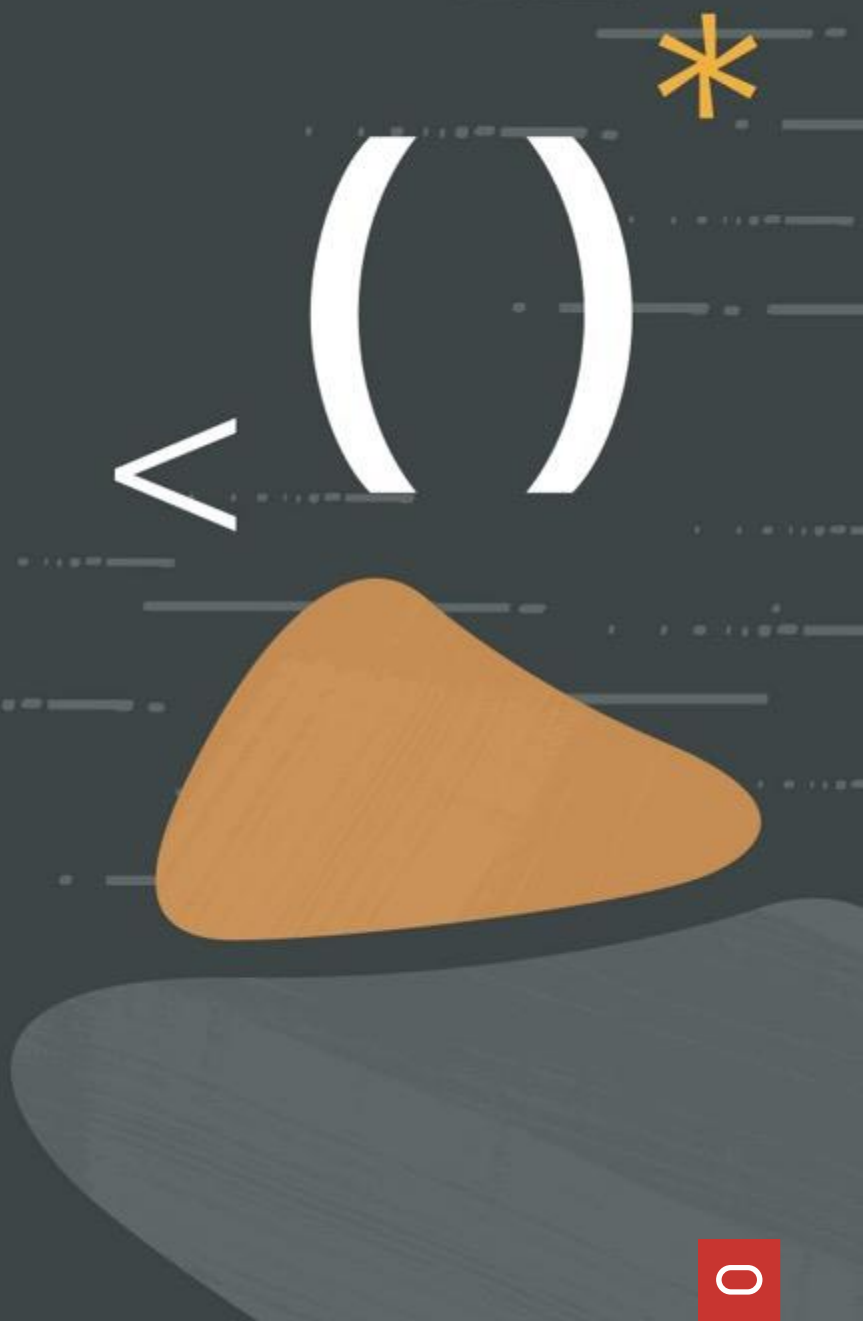
Articles:

- 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: the second part of this presentation**
  - https://openjdk.org/projects/babylon/articles/hat-matmul/hat-matmul

Videos:

Enabling Code Reflection for Acceleration on GPUs

Heterogeneous Accelerator Toolkit (HAT)



What Are We Trying to Answer?

Is code reflection good enough to support GPU programming models?

GPU support is just a use case for code reflection

+ eventually, it could be good target to specific workloads (e.g., AI) more efficiently from Java

Why do we want to run on hardware accelerators?

Higher Performance for a wide set of applications

Workloads:

- LLMs
- Deep Learning, Machine Learning
- Big Data
- Computer Vision
- HPC
- Simulations

Many types of accelerators:

- GPUs
- FPGAs
- RISC-V Accelerators
- Tensor Accelerators
- <Your-Future-Accelerator>

So, how do we program/accelerate from Java?

What is around?

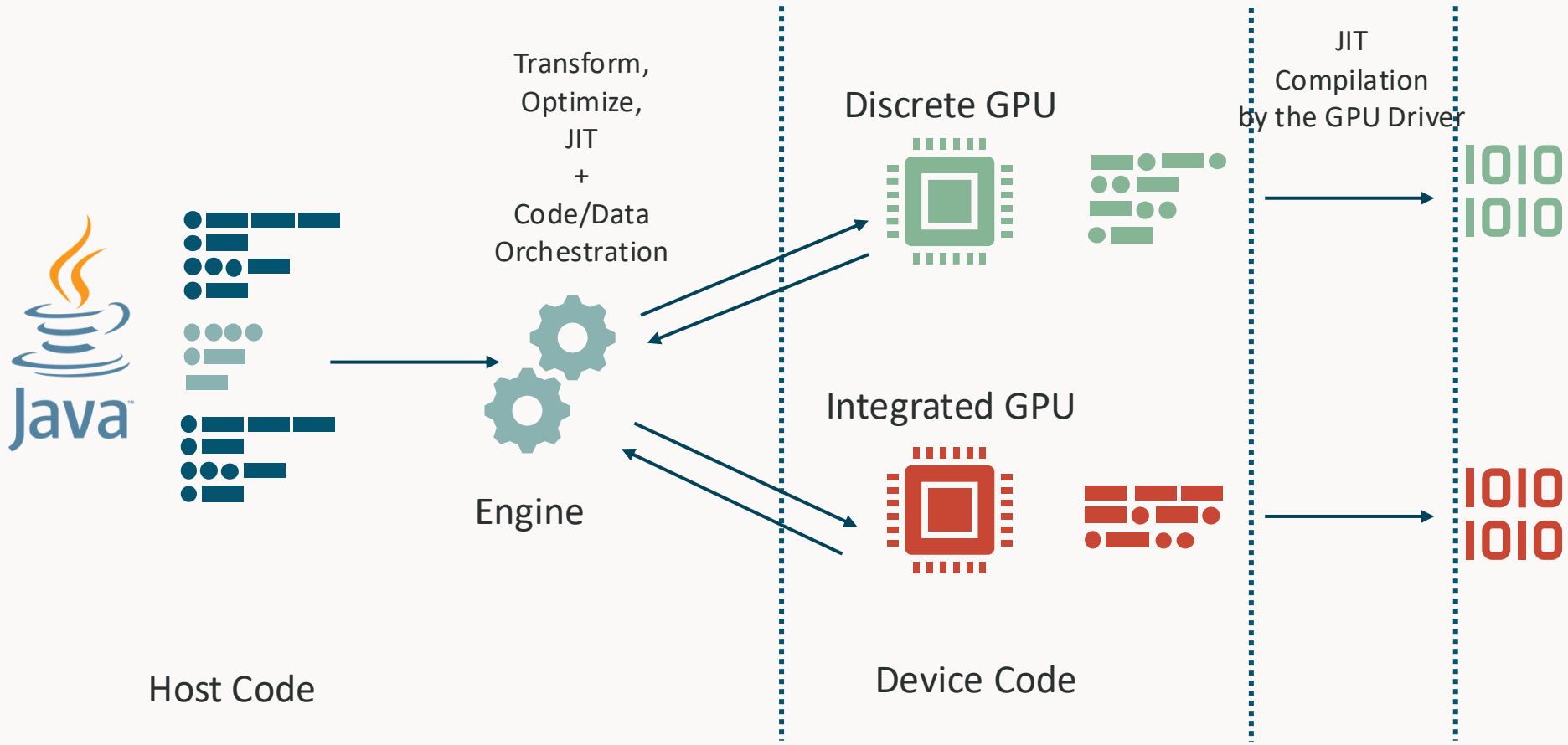
- **Sumatra: https://openjdk.org/projects/sumatra/**
- **Aparapi: https://github.com/Syncleus/aparapi**
- **IBM GPU J9: link**
- **RootBeer: https://github.com/bsletten/rootbeer1**
- **Marawacc: https://github.com/jjfumero/marawacc**
- **JaBEE: https://dl.acm.org/doi/10.1145/2159430.2159439**
- **TornadoVM: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

All of these projects focused on a **Top-Down approach** -> Creating abstractions at the cost of loosing some performance on the table.

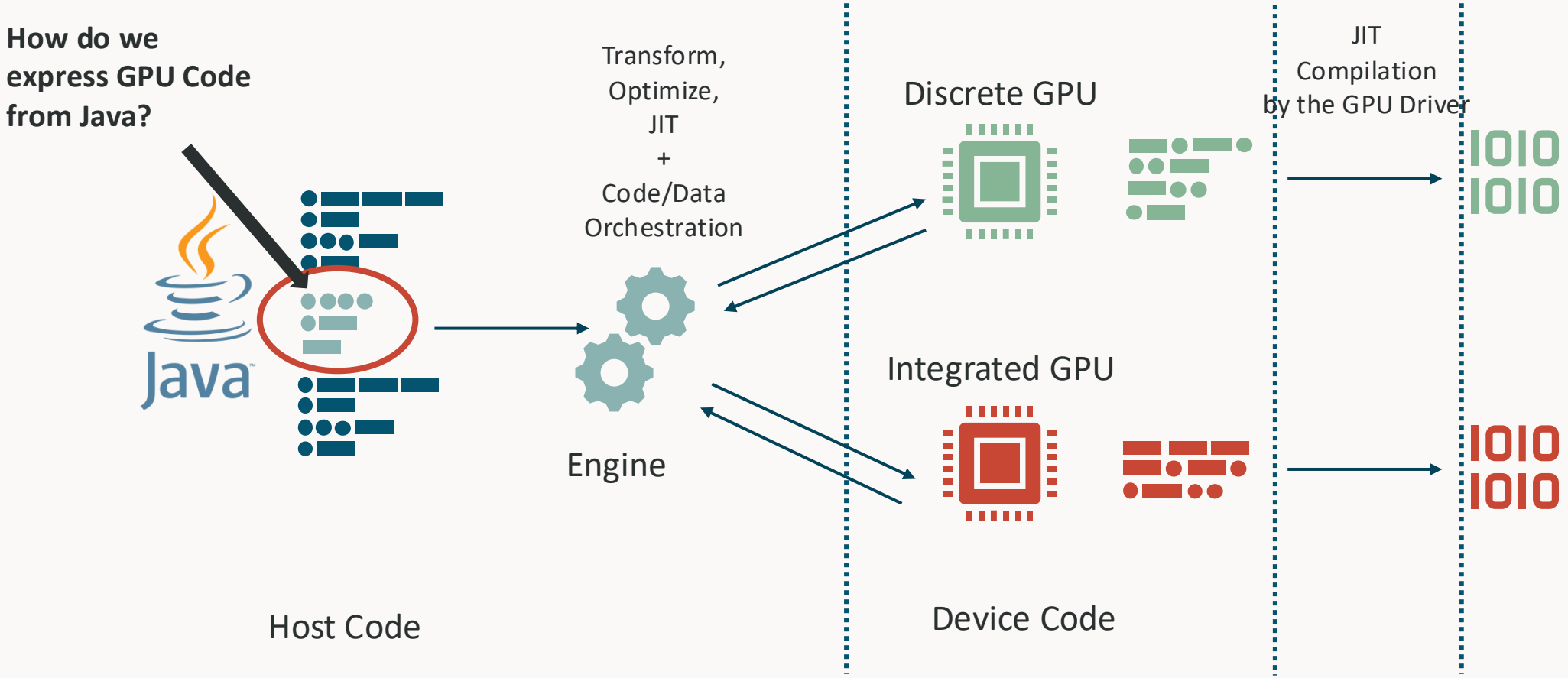
In the AI era, just running faster than Java might not be enough.

In HAT, we are focusing on a **Bottom-> UP approach**. From constructs that deliver high-performance, how can we create new abstractions without lossing performance?

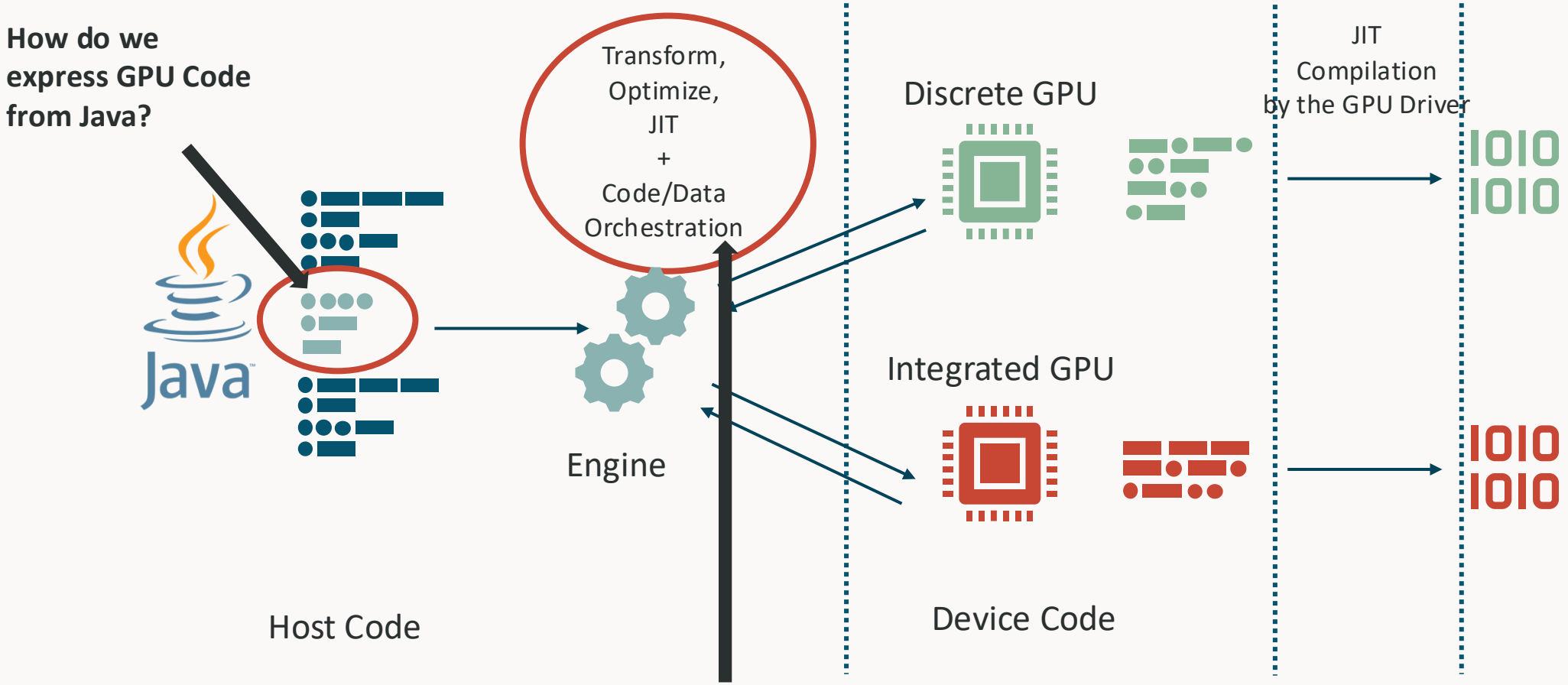
High-Level Overview: GPU Programming Workflow from Java



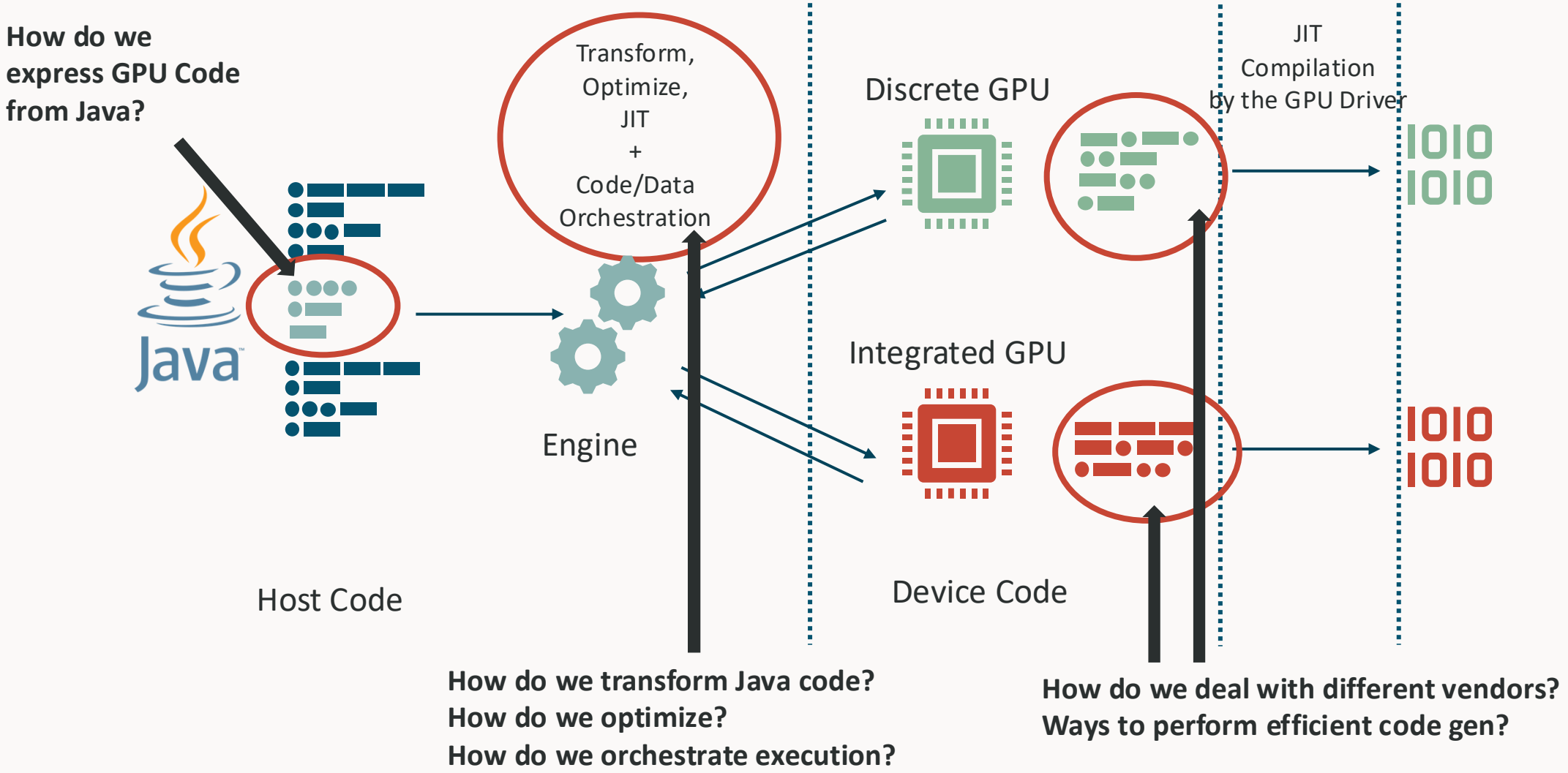
Identifying the Challenges: (1) APIs



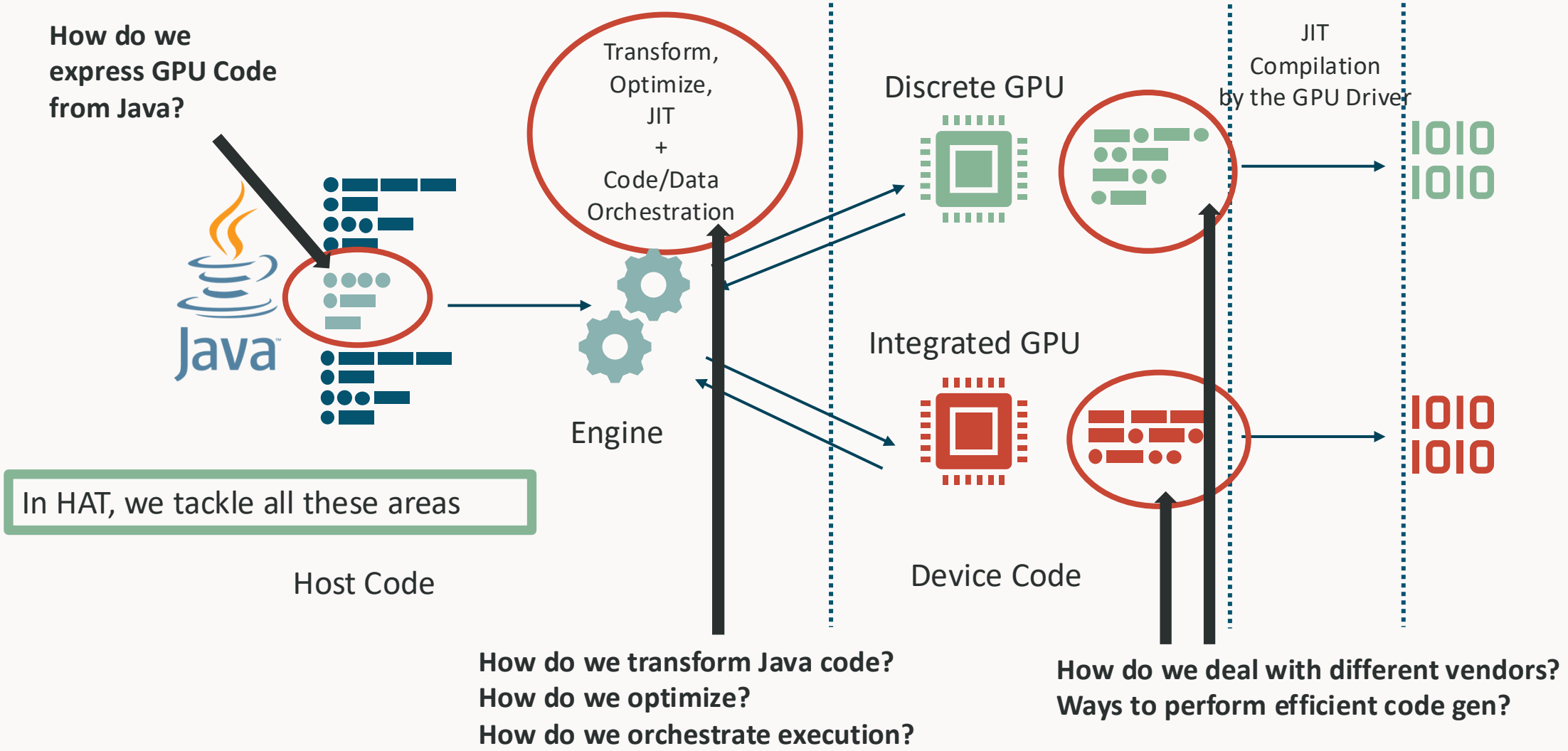
Identifying the Challenges: (2) Code/Data Orchestration



Identifying the Challenges: (3) Vendor Agnostic + Efficiency



Identifying the Challenges



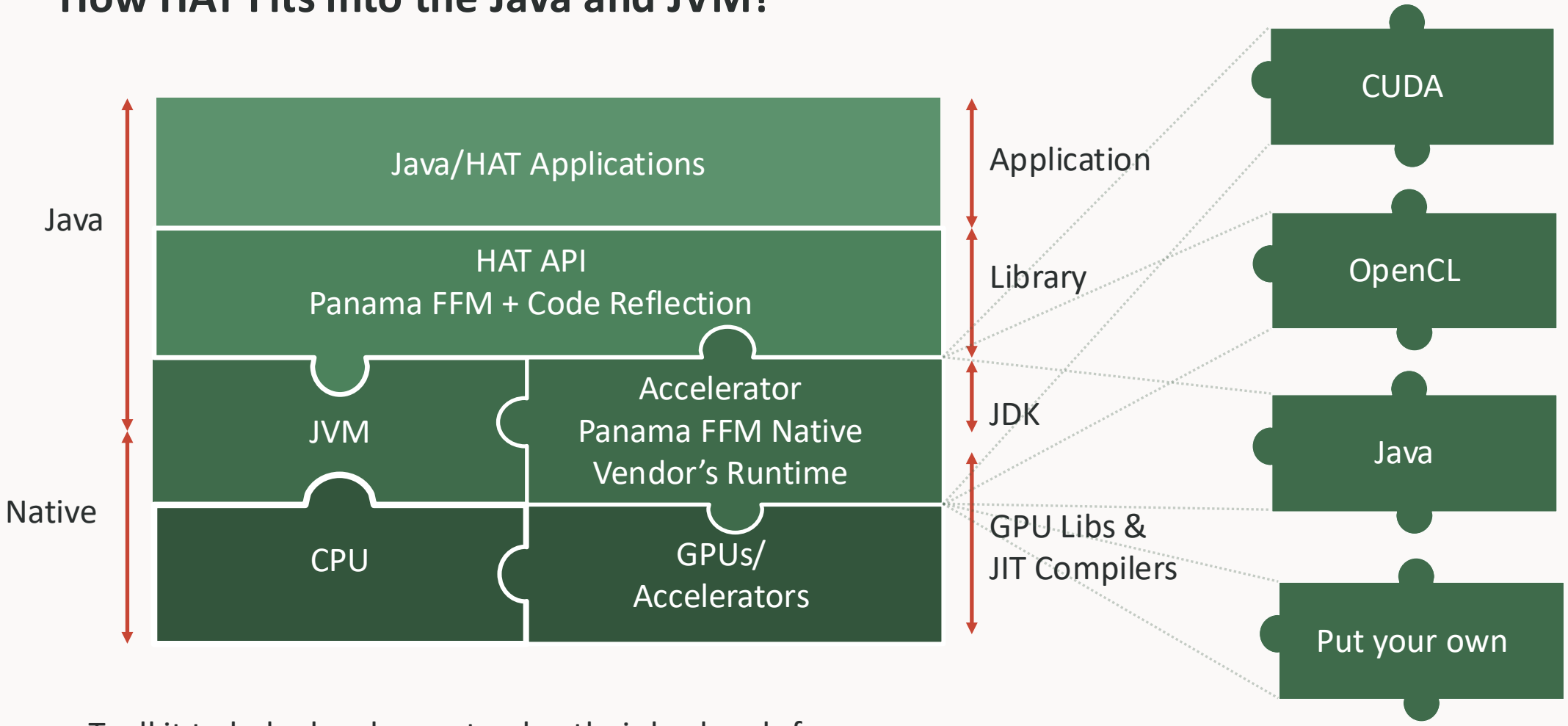
Heterogeneous Accelerator Toolkit (HAT)

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

- An API for Kernel Programming
- An API for Composing Compute Kernels (graphs)
- A plugable backend abstraction
  - CUDA
  - OpenCL
  - Java (in progress)
  - → Plug our own
- An Interface for Abstracting Data and Easy Mapping to Accelerators (Iface Mapper)

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

Programming GPUs from Java

Let’s look at an example: Vector Multiplication 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];
    }
}
```

Probably, Java developers will not write code like this. Instead they will use for-each, and a lambda.
BUT, we can improve this in the near future. For now, we need to generate efficient GPU code from these expressions.

- Functions return void
- I/O are passed as arguments to the function. We can also insert them via lexical scope, but we need to guarantee immutability (think of 1000s threads running this method).

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];
   }
}
```

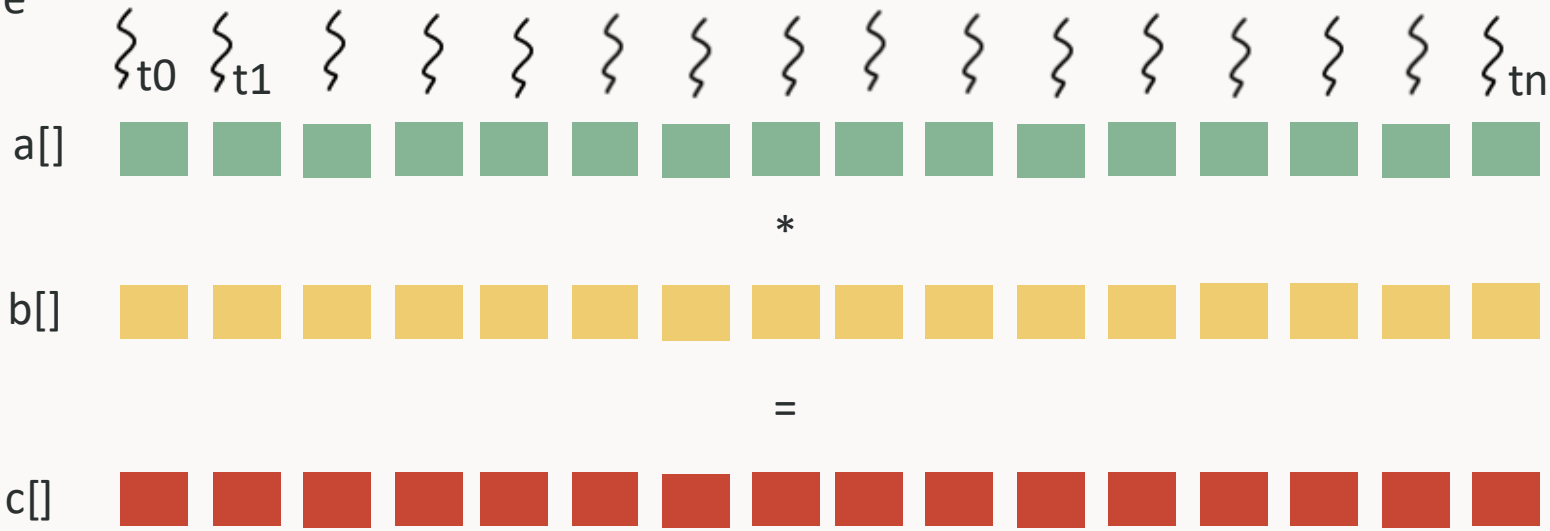
Work to be done per iteration

```
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

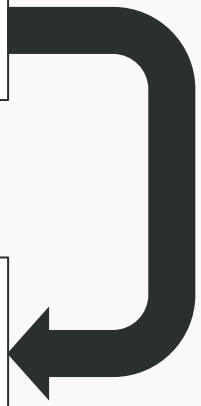
```
public void compute(int i, float[] a, float[] b, float[] c) {
    c[i] = a[i] * b[i];
}
```

Work to be done per thread



“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) {

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

Work to be done per thread

“HATify” Vector Multiplication: Step 1

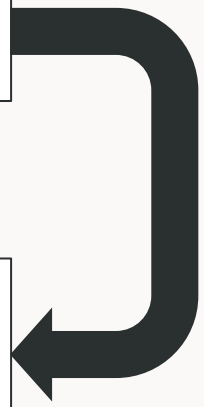
```
public void compute(int i, float[] a, float[] b, float[] c) {
    c[i] = a[i] * b[i];
}
```

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

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

KernelContext offers a set of builints to access GPU primitives (e.g., global thread index, thread-block, barriers, etc)

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

“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
    NDRange ndRange = NDRange.of(Global1D.of(arrayA.length()));

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

}
```

“HATify” Vector Multiplication: Step 3 ->Create a Compute Graph

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

    // Define how many threads to run
    NDRange ndRange = NDRange.of(Global1D.of(arrayA.length()));

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

}
```

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

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

Demo

Running Flash Attention v1 on Apple Silicon GPUs with Java

Demo Running Flash-Attention v1 on Apple Silicon M4

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 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


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
```

Compiling Java to GPUs with Dialects

How do we use code reflection APIs?

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
- Data access data from/to different levels of the memory hierarchy
- Operations in Special Types (e.g., bfloat16, float16, explicit vector types)
- Vector Operations
- Math Intrinsics

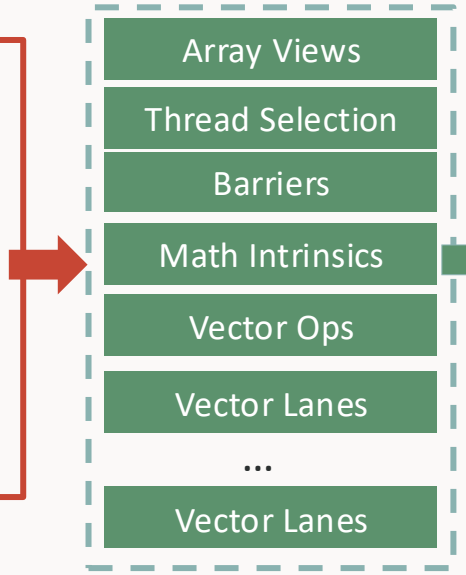
This is similar to MLIR and its GPU dialects: https://mlir.llvm.org/docs/Dialects/GPU/

HAT dialect is a Work in Progress

We are working on bringing a **minimal set of Ops that allows developers to create C99-based backends for hardware accelerators**.

- We are trying different approaches
- Expect this to change frequently

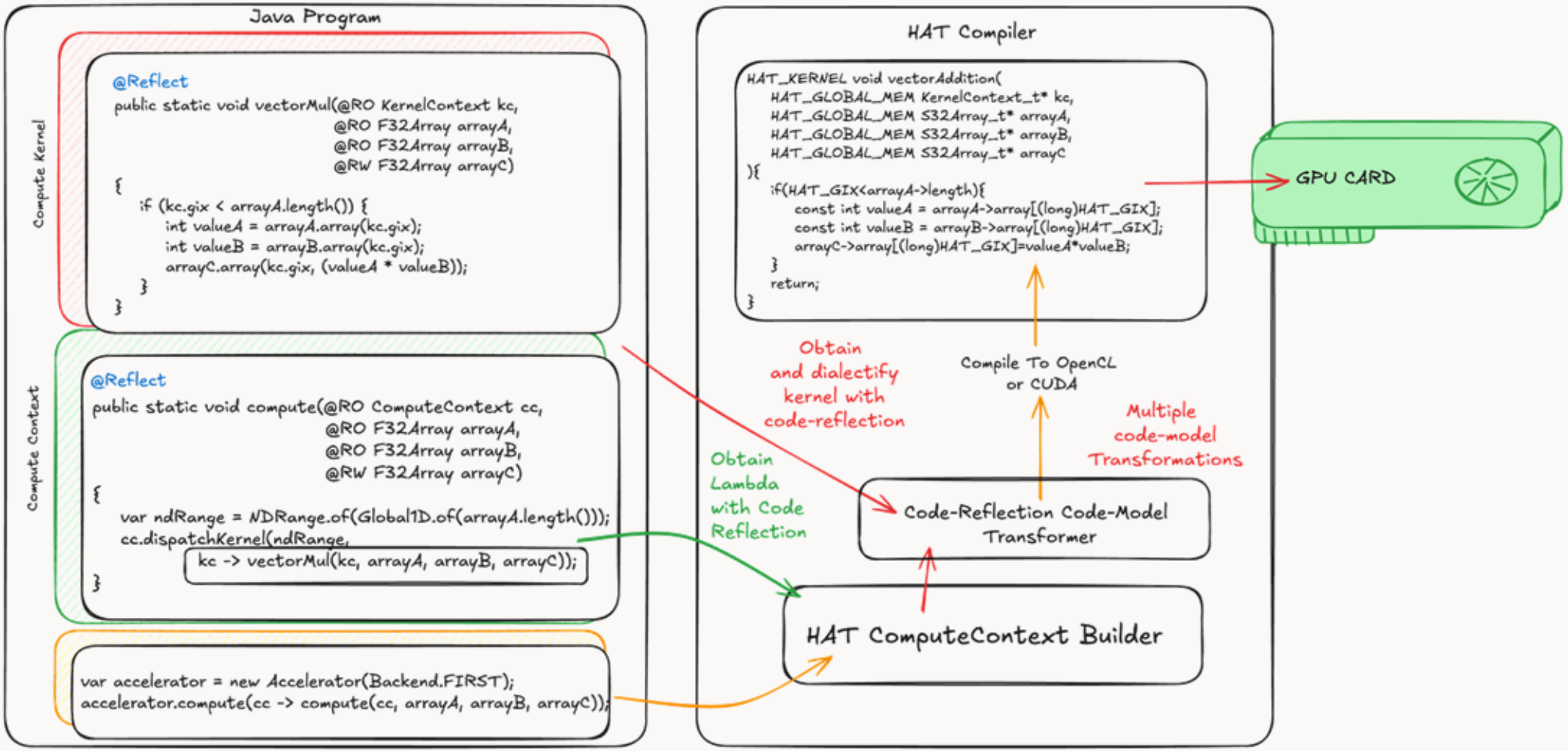
```
%37 : java.type:"int" = field.load %36 @java.ref:"hat.KernelContext::liy:int";
%38 : java.type:"hat.KernelContext" = var.load %5
%39 : java.type:"int" = field.load %38 @java.ref:"hat.KernelContext::lsx:int";
%40 : java.type:"int" = mul %37 %39
%41 : java.type:"hat.KernelContext" = var.load %5
%42 : java.type:"int" = field.load %41 @java.ref:"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 @java.ref:"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 @java.ref:"hat.KernelContext::liy:int";
%50 : Var<java.type:"int"> = var %49 @"threadRow";
%51 : java.type:"matmul.Main$SharedMemory" = invoke SharedMemory::createLocal";
%52 : Var<java.type:"matmul.Main$SharedMemory"> = var %51 @"tileA";
%53 : java.type:"matmul.Main$SharedMemory" = invoke SharedMemory::createLocal";
```




```
%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<java.type:"SharedMemory"> = hat.dialect.HATMemoryVarOp$HATLocalVarOp
%45 : Var<java.type:"SharedMemory"> = hat.dialect.HATMemoryVarOp$HATLocalVarOp
```

HAT Dialectify

HAT Compilation Workflow: Connecting all Pieces



Studying the cost of User Data Structures (UDT) for GPUs

Allowing User Data Structures (UDT)

HAT allows developers to define their own User Data Structures, UDT, by extending the Buffer Types.

UDT can be also mapped to Shared/Private memory of the GPU! More on this later.

By default, buffer types are mapped to off-heap memory segments using the Panama FFM API.

HAT also offers a set of common types:
- F32Array
- S32Array
- F16Array/ BF16Array
- F16/BF16 values
- Vector Types: Float2, Float4, etc.

Probably more soon

Modeling Data Abstractions

Java

```
class Complex {
  float real;
  float imag;
  public Complex(float real, float imag){
    this.real = real;
    this.imag = imag;
  }
  public void real(float real) {
    this.real = real;
  }
  public void imag(float imag) {
    this.imag = imag;
  }

  public float real() {
    return this.real;
  }
  public float imag() {
    return this.imag;
  }
}
Complex[] c = new Complex[size];
```

```
Complex c1 = Complex[i];
c1.setReal(2.1f);
```

On-Head Array

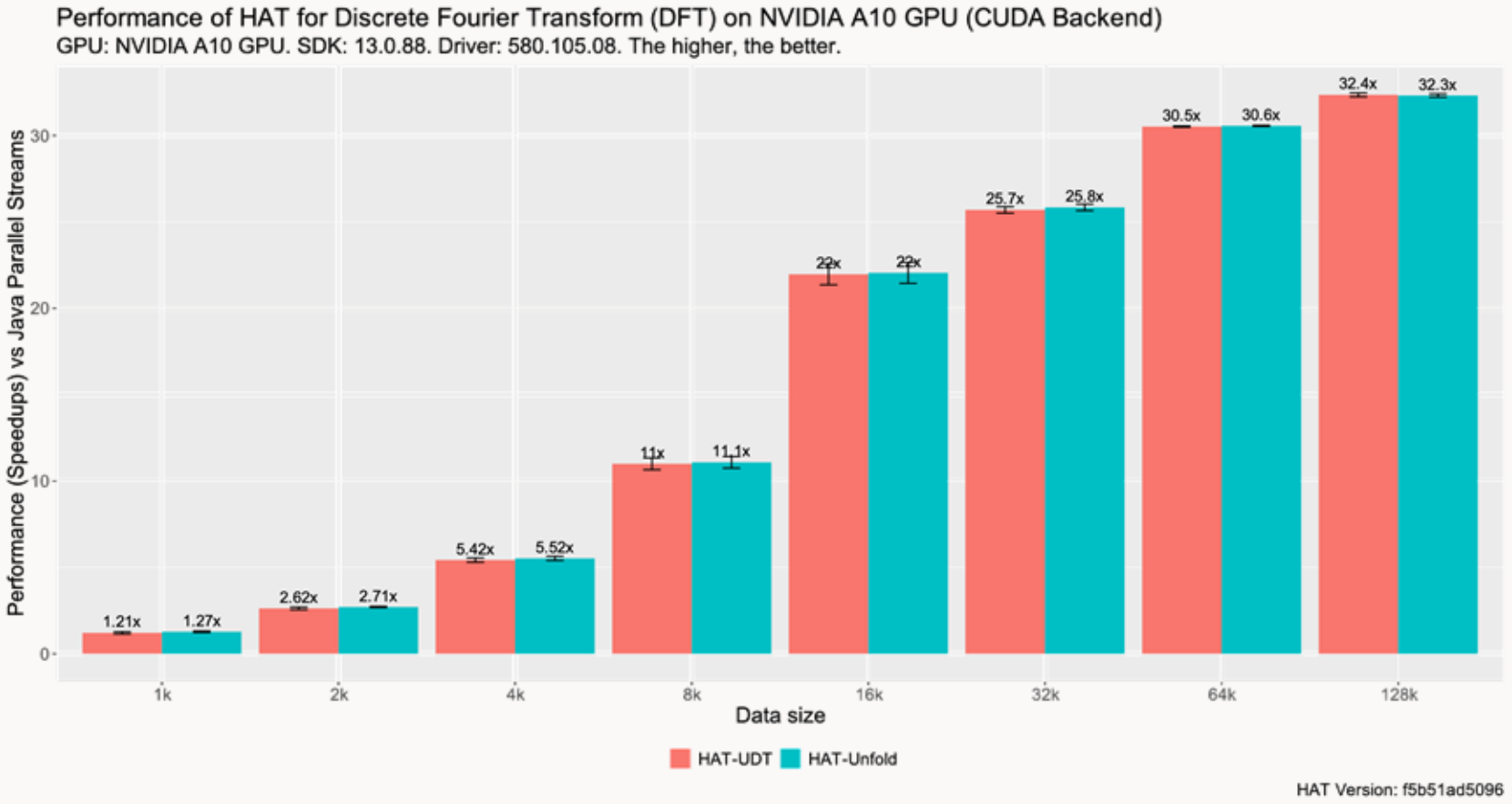
HAT UDT

```
public interface CArray extends Buffer {
    int length();   // size of the array

    // nested interface to obtain <real,imag> values
    interface Complex extends Struct {
        float real();          // getter
        float imag();          // getter
        void real(float real); // setter
        void imag(float imag); // setter
    }

    // Getter
    Complex complex(long index);

    // Schema descriptor
    Schema<CArray> schema = Schema.of(CArray.class, complex ->
            complex.arrayLen("length")
                    .array("complex", array ->
                         array.fields("real", "imag")));
}
```

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

Buffers are stored in a Memory Segment (Panama FFM)

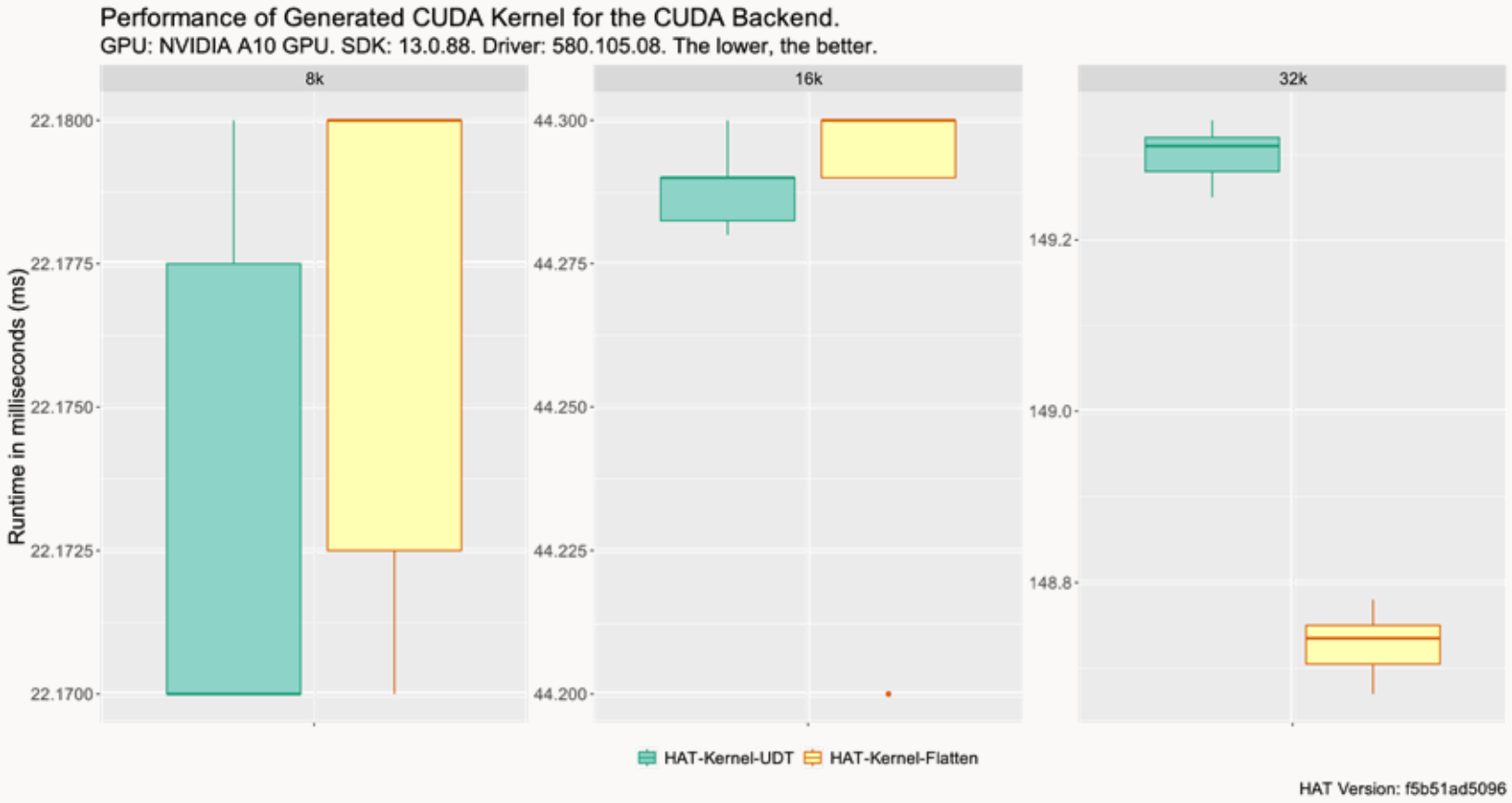
Cost of Data Types Abstractions



As end-2-end time, there is no difference in performance.

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

Cost of Data Types Abstractions



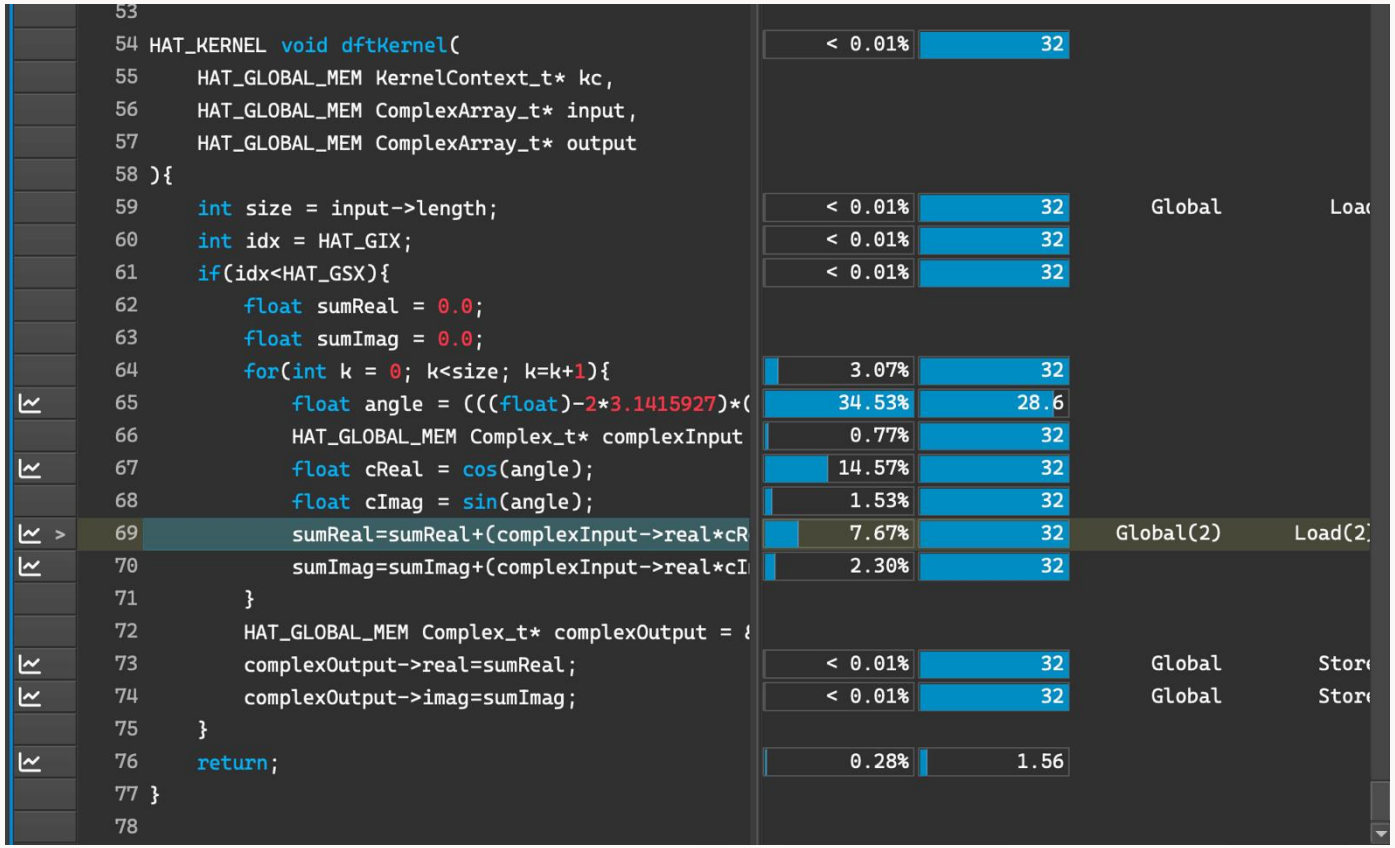
Kernel Time Measured with NVIDIA NCU Profiler

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.

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

Cost of Data Types Abstractions



```
54 HAT_KERNEL void dftKernel(                                  < 0.01%   32
55     HAT_GLOBAL_MEM KernelContext_t* kc,
56     HAT_GLOBAL_MEM ComplexArray_t* input,
57     HAT_GLOBAL_MEM ComplexArray_t* output
58 ){
59     int size = input->length;                               < 0.01%   32   Global   Loa
60     int idx = HAT_GIX;                                      < 0.01%   32
61     if(idx<HAT_GSX){                                        < 0.01%   32
62         float sumReal = 0.0;
63         float sumImag = 0.0;
64         for(int k = 0; k<size; k=k+1){                       3.07%    32
65             float angle = (((float)-2*3.1415927)*(          34.53%    28.6
66             HAT_GLOBAL_MEM Complex_t* complexInput           0.77%    32
67             float cReal = cos(angle);                       14.57%    32
68             float cImag = sin(angle);                        1.53%    32
69             sumReal=sumReal+(complexInput->real*cR           7.67%    32   Global(2)   Load(2
70             sumImag=sumImag+(complexInput->real*cI           2.30%    32
71         }
72         HAT_GLOBAL_MEM Complex_t* complexOutput = &
73         complexOutput->real=sumReal;                        < 0.01%   32   Global   Stor
74         complexOutput->imag=sumImag;                        < 0.01%   32   Global   Stor
75     }
76     return;                                                  0.28%    1.56
77 }
78
```

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.

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

But, how far can we go with code reflection?

Matrix Multiplication Case Study

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

Matmul on CPU with Parallel Streams with 15 CPU cores (reference)

```
$ java -cp hat/job.jar hat.java run matmul  MT
...
Elapsed Time: 295976770 ns
Elapsed Time: 297536710 ns
Elapsed Time: 295931303 ns
Result is correct!
```

Running Java on CPU
with Parallel Streams

~299 ms per iteration. This runs for 100 times.

This translates to ~7.18 GFLOPS

Baseline for GPUs – HAT Initial Version – 1D Kernel

```
@Reflect
public static void matrixMultiplyKernel1D(@RO KernelContext kc,   // GPU context
                                          @RO F32Array matrixA,   // first matrix
                                          @RO F32Array matrixB,   // second matrix
                                          @RW F32Array matrixC,   // result matrix
                                          int size) {
    if (kc.gix < size) {
        for (int j = 0; j < size; j++) {
            float acc = 0.0f;
            for (int k = 0; k < size; k++) {
                acc += (matrixA.array(kc.gix * size + k) * matrixB.array(k * size + j));
            }
            matrixC.array(kc.gix * size + j, acc);
        }
    }
}
…

var ndRange = NDRange.of(Global1D.of(1024), Local1D.of(16));
cc.dispatchKernel(ndRange,
    kc -> matrixMultiplyKernel1D(kc, matrixA, matrixB, matrixC, 1024));
```

Profiling with NVIDIA Tooling (NVIDIA Nsight Compute)

```
matrixMultiplyKernel1D (64, 1, 1)x(16, 1, 1), Context 1, Stream 13, Device 0, CC 8.6
    Section: GPU Speed Of Light Throughput
    ----------------------- ----------- ------------
    Metric Name             Metric Unit Metric Value
    ----------------------- ----------- ------------
    DRAM Frequency                  Ghz         6.24  // Main Memory Frequency
    SM Frequency                    Mhz       885.00  // Frequency of the Stream Multiprocessor
    Elapsed Cycles                cycle     84774133  // GPU cycles consumed
    Memory Throughput                 %        18.71  // % of the peak memory throughput consumed
    Duration                         ms        95.79  // Kernel duration in ms
    ...
    Compute (SM) Throughput           %         4.40  // % of the peak compute throughput consumed
    ----------------------- ----------- ------------
```

Kernel takes 95ms, 3x faster than Java Parallel Streams on CPU

Can we do better?

https://developer.nvidia.com/nsight-compute

Profiling with NVIDIA Tooling (NVIDIA Nsight Compute)

https://developer.nvidia.com/nsight-compute

This **kernel grid is too small** to fill the available resources on this device, resulting in only 0.1 full waves across all SMs.

We need to increase the number of threads.

2D Matmul Kernel in HAT:

```
@Reflect
public static void matrixMultiplyKernel2D(@RO KernelContext kc, @RO F32Array matrixA, @RO F32Array matrixB,
@RW F32Array matrixC, int size) {
    if (kc.gix < kc.gsx) {       // control for 1D range  ( thread id 1D -> kc.gix )
        if (kc.giy < kc.gsy) {   // conntrol for 2D range ( thread id 2D -> kc.giy )
            float acc = 0.0f;
            for (int k = 0; k < size; k++) {
                acc += (matrixA.array(kc.gix * size + k) * matrixB.array(k * size + kc.giy));
            }
            matrixC.array(kc.gix * size + kc.giy, acc);
        }
    }
}

…

final int globalSize = 1024;
var ndRange = NDRange.of(Global2D.of(globalSize, globalSize), Local2D.of(16, 16));
cc.dispatchKernel(ndRange,
    kc -> matrixMultiplyKernel2D(kc, matrixA, matrixB, matrixC, globalSize)
);
```

Profiling with NVIDIA Tooling (NVIDIA Nsight Compute)

```
matrixMultiplyKernel2D (64, 64, 1)x(16, 16, 1), Context 1, Stream 13, Device 0, CC 8.6
    Section: GPU Speed Of Light Throughput
    ----------------------- ----------- ------------
    Metric Name             Metric Unit Metric Value
    ----------------------- ----------- ------------
    DRAM Frequency                  Ghz         6.24
    SM Frequency                    Mhz       884.96
    Elapsed Cycles                cycle      8010119
    Memory Throughput                 %        99.37
    Duration                         ms         9.05
    ...
    Compute (SM) Throughput           %        23.31
    ----------------------- ----------- ------------
```

Kernel takes 9ms, 10x faster than first GPU version

But now, how can we improve it?

Profiling second version with NVIDIA Nsight Compute

https://developer.nvidia.com/nsight-compute

The memory access pattern for global loads from L1 might not be optimal. On average, only 4.2 of the 32 bytes transmitted per sector are utilized by each thread. This could possibly be caused by a stride between threads.

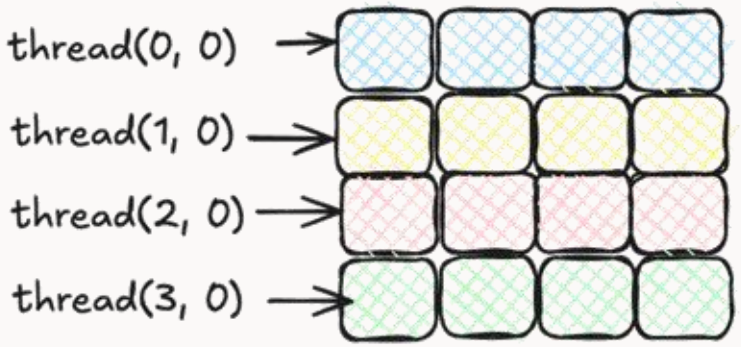
We need to improve **global coalesced memory accesses**

Let’s talk about Global Coalesced Memory Accesses

GPU coalesced memory accesses occur when consecutive threads within a **warp** access consecutive memory locations, minimizing cache misses.

In our Java code, we are accessing global memory as follows:

```
acc += (matrixA.array(kc.gix * size + k) * matrixB.array(k * size + kc.giy));
```

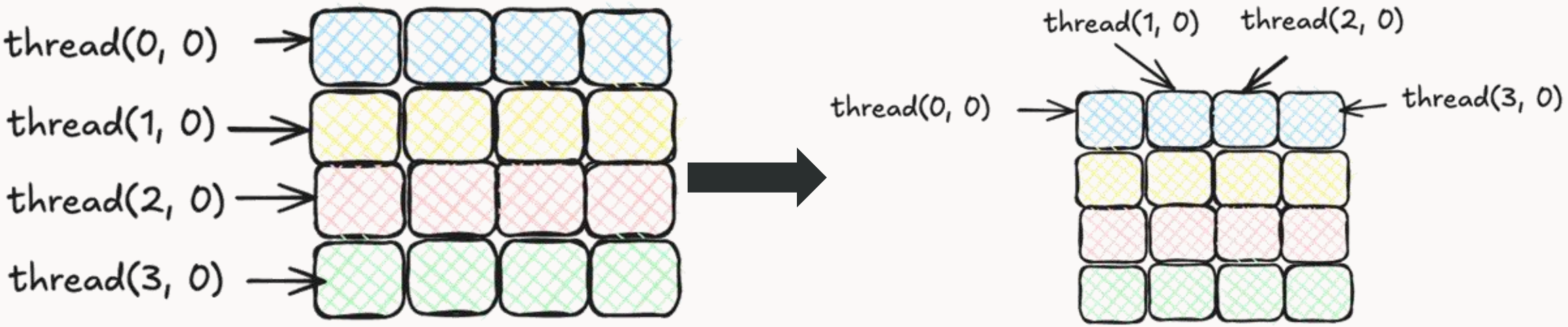


”When using 2 or 3-dimensional thread blocks in a CUDA kernel, the threads are laid out linearly with the X index, or threadIdx.x, moving the fastest.”

https://developer.nvidia.com/blog/unlock-gpu-performance-global-memory-access-in-cuda/

(0, 0), (1, 0), (2, 0), (3, 0), (1, 0), (1, 1), (2, 1), (3, 1), ...

Instead, what we want:




```
acc += (matrixA.array(kc.giy * size + k) * matrixB.array(k * size + kc.gix));
```

Profiling 2D-LI with NVIDIA Tooling (NVIDIA Nsight Compute)

```
matrixMultiplyKernel2DLI (64, 64, 1)x(16, 16, 1), Context 1, Stream 13, Device 0, CC 8.6
    Section: GPU Speed Of Light Throughput
    ----------------------- ----------- ------------
    Metric Name             Metric Unit Metric Value
    ----------------------- ----------- ------------
    DRAM Frequency                  Ghz         6.24
    SM Frequency                    Mhz       884.50
    Elapsed Cycles                cycle      1939911
    Memory Throughput                 %        96.35
    Duration                         ms         2.19   >> Kernel Time
    ...
    Compute (SM) Throughput           %        96.35
```

Kernel takes 2.1ms: 4.1 x faster than non-coalesced

Profiling third version with NVIDIA Nsight Compute

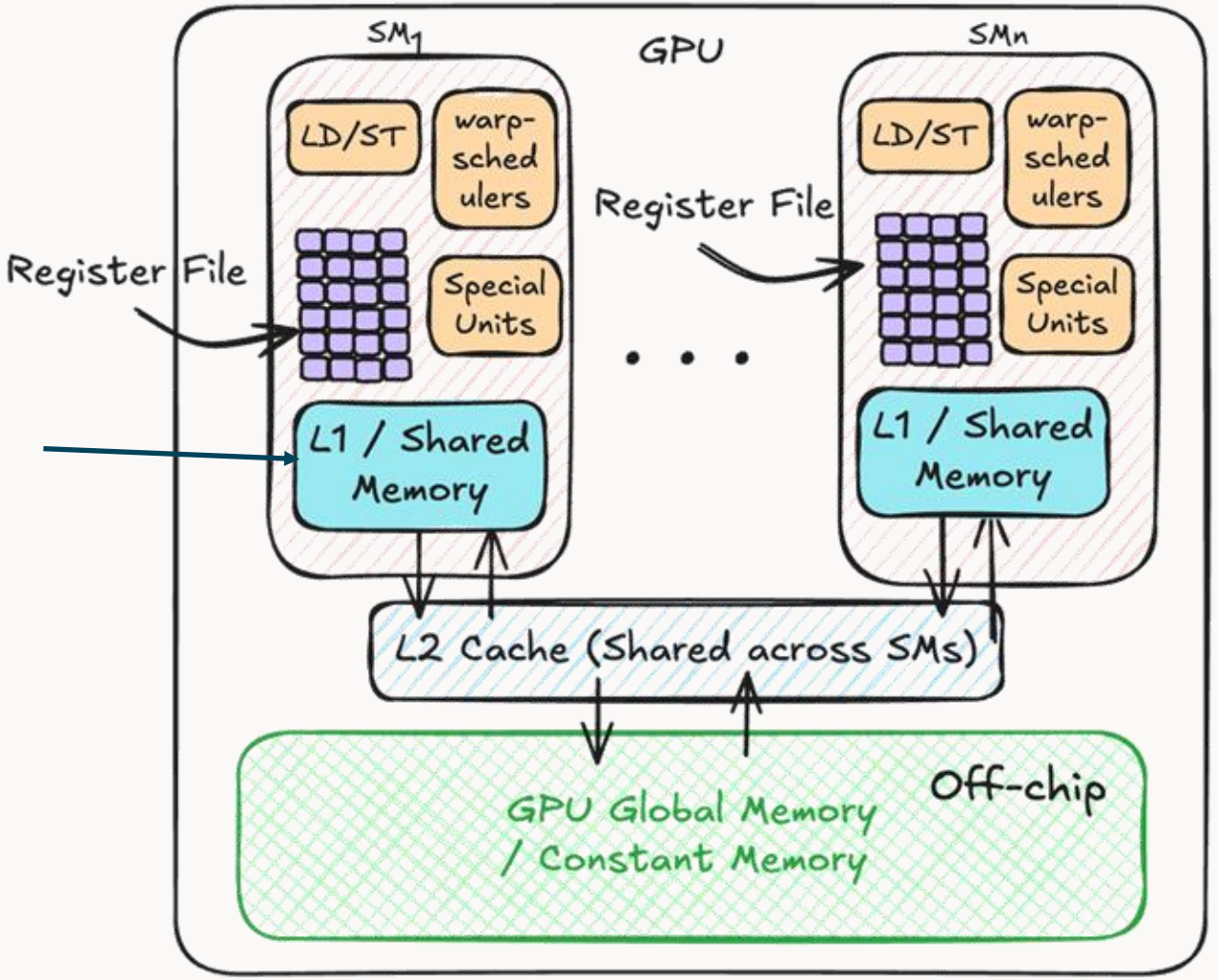
The memory access pattern for global loads from L1TEX might not be optimal. On average, only 14.4 of the 32 bytes transmitted per sector are utilized by each thread.

Since both compute and memory throughput are in equal percentage, the suggestion could be to reduce both by increasing the work per thread, and storing reusable data in **shared memory of the GPU**

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

Programmable Memory in OpenCL/CUDA

Also programmable in HAT. It also allows User Defined Types in Shared Programmable Memory



Using Scratchpad Memory of GPUs with HAT

```
private interface MyLocalArray extends DeviceType {

    // Getter
    void array(long index, float value);

    // Setter
    float array(long index);


    // Device-Schema: static object that composes an intermediate
    // representation of this Data Structure to be consumed by the
    // HAT code generator.
    DeviceSchema<MyLocalArray> schema = DeviceSchema.of(MyLocalArray.class,
            arr -> arr.withArray("array", 256));  // Fixed size

    // data structure to allocate this data structure in shared memory
    static MyLocalArray createLocal() { return null;}

    // Marker method to allocate this data structure in private memory
    static MyLocalArray createPrivate() { return null;}
}
```

CUDA/OpenCL

```
typedef struct MyLocalArray_s{
    float array[256];
} MyLocalArray_t;

kernel (... ) {
    ...
    HAT_LOCAL_MEM MyLocalArray_t tileA;
}
```

Forth Optimization: Using Shared Memory

```
@Reflect
public static void matrixMultiplyKernel2DTiling(@RO KernelContext kc, @RO F32Array matrixA, @RO F32Array matrixB, @RW F32Array matrixC, int size) {

    final int tileSize = 16;
    MyLocalArray tileA = MyLocalArray.createLocal();
    MyLocalArray tileB = MyLocalArray.createLocal();

    int groupIndexX = kc.bix;   // access to the thread-block (1D)
    int groupIndexY = kc.biy;   // access to the thread-block (2D)
    int localIdx = kc.lix;      // thread indexing to access local-block (1D)
    int localIdy = kc.liy;      // thread indexing to access local-block (2D)

    // we identify the row and column
    int row = groupIndexY * tileSize + localIdy;
    int col = groupIndexX * tileSize + localIdx;

    // Compute matrix-vector and accumulate the result over the tiles
    float sum = 0.0f;
    for (int tile = 0; tile < (size / tileSize); tile++) {
        // Copy from global to shared memory
        tileA.array((long) localIdy * tileSize + localIdx, matrixA.array((long) row * size + tile * tileSize + localIdx));
        tileB.array((long) localIdy * tileSize + localIdx, matrixB.array((tile * tileSize + localIdy) * size + col));

        // Apply a barrier for the local group: we need to guarantee that all threads that belong
        // to the same group reach this point before doing the partial reduction
        kc.barrier();

        // compute partial reductions over the tile
        for (int k = 0; k < tileSize; k++) {
            sum += (tileA.array((long) localIdy * tileSize + k) * tileB.array(k * tileSize + localIdx));
        }

        // A new local barrier for all threads that belong to the same group before loading a new tile into
        // share memory. With the following barrier, we can ensure that all threads within the same workgroup
        // finished the compute for the partial reduction
        kc.barrier();
    }

    // copy result from shared memory to global memory
    matrixC.array((long) row * size + col, sum);
}
```

This is getting more complex, but more optimized

Profiling fourth version with NVIDIA Nsight Compute


```
matrixMultiplyKernel2DTiling (64, 64, 1)x(16, 16, 1), Context 1, Stream 13, Device 0, CC 8.6
    Section: GPU Speed Of Light Throughput
    ----------------------- ----------- ------------
    Metric Name             Metric Unit Metric Value
    ----------------------- ----------- ------------
    DRAM Frequency                  Ghz         6.24
    SM Frequency                    Mhz       884.87
    Memory Throughput                 %        95.81
    DRAM Throughput                   %         1.95
    Duration                         ms         1.65
    ...
    Compute (SM) Throughput           %        95.81
    ----------------------- ----------- ------------
```

Kernel takes 1.65ms, 1.33x faster than the prev. version.

Fifth Optimization: Register Tiling

Inspired by Simon's blog: https://siboehm.com/articles/22/CUDA-MMM

The idea is to exploit register memory as another memory tier to compute smaller matrices. Thus, we move a block of data from global to shared memory, then we move a sub-block from shared to private, and compute a smaller tile in registers.

In HAT, we can also program User Defined Types to be stored and load to/from private memory as follows:

```
private interface PrivateArray extends DeviceType {
    void array(long index, float value);
    float array(long index);

    DeviceSchema<PrivateArray> schema = DeviceSchema.of(PrivateArray.class,
        arr -> arr.withArray("array", 16));

    static PrivateArray createPrivate() {return null;}
}
```

Profiling 2D-Register Tiling with NVIDIA Tooling

```
matrixMultiplyKernel2DRegisterTiling (16, 16, 1)x(16, 16, 1), Context 1, Stream 13, Device 0, CC 8.6
    Section: GPU Speed Of Light Throughput
    ----------------------- ----------- ------------
    Metric Name             Metric Unit Metric Value
    ----------------------- ----------- ------------
    DRAM Frequency                  Ghz         6.24
    SM Frequency                    Mhz       881.14
    Elapsed Cycles                cycle       329296
    Memory Throughput                 %        66.39
    Duration                         us       373.47
    ...
    Compute (SM) Throughput           %        54.46
    ----------------------- ----------- ------------
```

Kernel takes **373 microseconds!!!.** 5.8 x faster than the prev. kernel

2D Register Tiling with NVIDIA Nsight Compute

Uncoalesced Global Accesses: 33% of this line's global accesses

Coming from this line in Java:

```
matrixA.array(((innerRowA + loadOffset) * size + innerColA) + aFrom);
```

What we can do is to vectorize accesses.
We can do this in HAT by using explicit vector types:

- **Float4**: 4 floats in fp32
- Load floats from a global array: **`matrixA.float4View(index);`**

2D Register Tiling With Vector Types (snippet)

```
Float4 loadA = matrixA.float4View((innerRowA * K + innerColA * 4) + aFrom);
tileA.array((innerColA * 4 + 0) * BM + innerRowA, loadA.x());
tileA.array((innerColA * 4 + 1) * BM + innerRowA, loadA.y());
tileA.array((innerColA * 4 + 2) * BM + innerRowA, loadA.z());
tileA.array((innerColA * 4 + 3) * BM + innerRowA, loadA.w());

Float4 loadB = matrixB.float4View((innerRowB * N + innerColB * 4) + bFrom);
tileB.array(innerRowB * (BN + extraCols) + innerColB * 4 + 0, loadB.x());
tileB.array(innerRowB * (BN + extraCols) + innerColB * 4 + 1, loadB.y());
tileB.array(innerRowB * (BN + extraCols) + innerColB * 4 + 2, loadB.z());
tileB.array(innerRowB * (BN + extraCols) + innerColB * 4 + 3, loadB.w());
```

matrixMultiplyKernel2DRegisterTilingVectorized (16, 16, 1)x(16, 16, 1), Context 1, Stream 13, Device 0, CC 8.6

Section: GPU Speed Of Light Throughput

Metric Name	Metric Unit	Metric Value
DRAM Frequency	Ghz	6.24
SM Frequency	Mhz	881.35
Memory Throughput	%	73.13
Duration	**us**	**373.73**
...		
Compute (SM) Throughput	%	43.33

Identical time to non-vectorized.

One last kernel: using narrow types (FP16: 16-bit floats)

Disclaimer: This is not really an optimization, unless we know we can compute with less precision (common case in LLMs)

```
private interface SharedMemoryHalf extends DeviceType {
    F16 array(int index);

    DeviceSchema<SharedMemoryHalf> schema = DeviceSchema.of(SharedMemoryHalf.class,
        arr -> arr.withArray("array", 1024)
                .withDeps(F16.class, half -> half.withField("value")));

    static SharedMemoryHalf createLocal() {return null;}
}
```

One last kernel: using narrow types (FP16: 16-bit floats)

Disclaimer: This is not really an optimization, unless we know we can compute with less precision (common case in LLMs)

```
private interface SharedMemoryHalf extends DeviceType {
    F16 array(int index);

    DeviceSchema<SharedMemoryHalf> schema = DeviceSchema.of(SharedMemoryHalf.class,
        arr -> arr.withArray("array", 1024)
                .withDeps(F16.class, half -> half.withField("value")));

    static SharedMemoryHalf createLocal() {return null;}
}
```

```
matrixMultiplyKernel2DRegisterTilingHalf (16, 16, 1)x(16, 16, 1), Context 1, Stream 13, Device 0, CC 8.6
    Section: GPU Speed Of Light Throughput
    ----------------------- ----------- ------------
    Metric Name             Metric Unit Metric Value
    ----------------------- ----------- ------------
    DRAM Frequency                  Ghz         6.24
    SM Frequency                    Mhz       881.43
    Memory Throughput                 %        70.82
    DRAM Throughput                   %         5.76
    Duration                         us       281.50
    ...
    Compute (SM) Throughput           %        70.82
    ----------------------- ----------- ------------
```

281 microseconds!!

What about native GPU code? How far are we compared to cuBLAS?

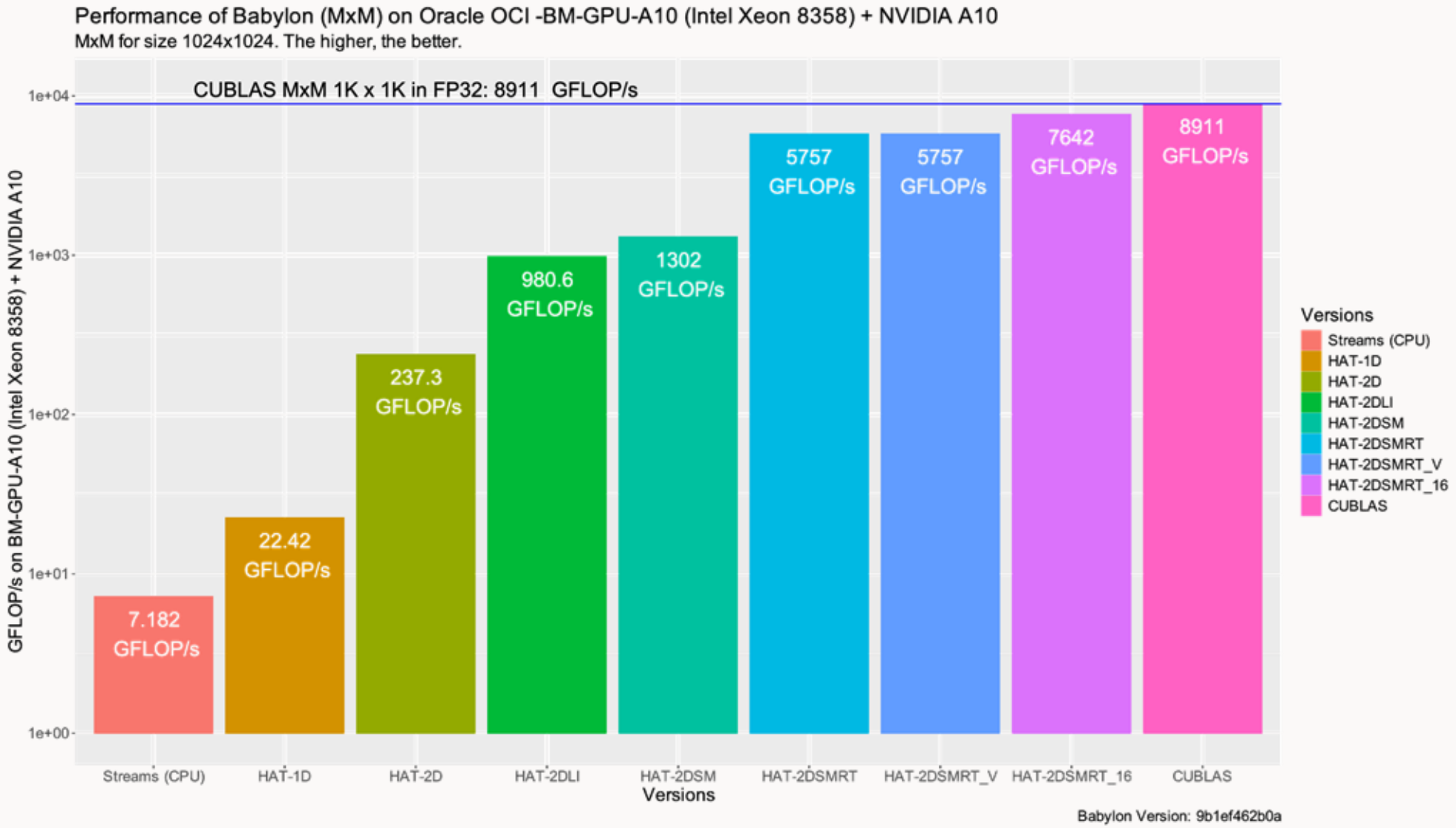
ampere_sgemm_128x64_nn (8, 16, 5)x(128, 1, 1), Context 1, Stream 7, Device 0, CC 8.6

Section: GPU Speed Of Light Throughput

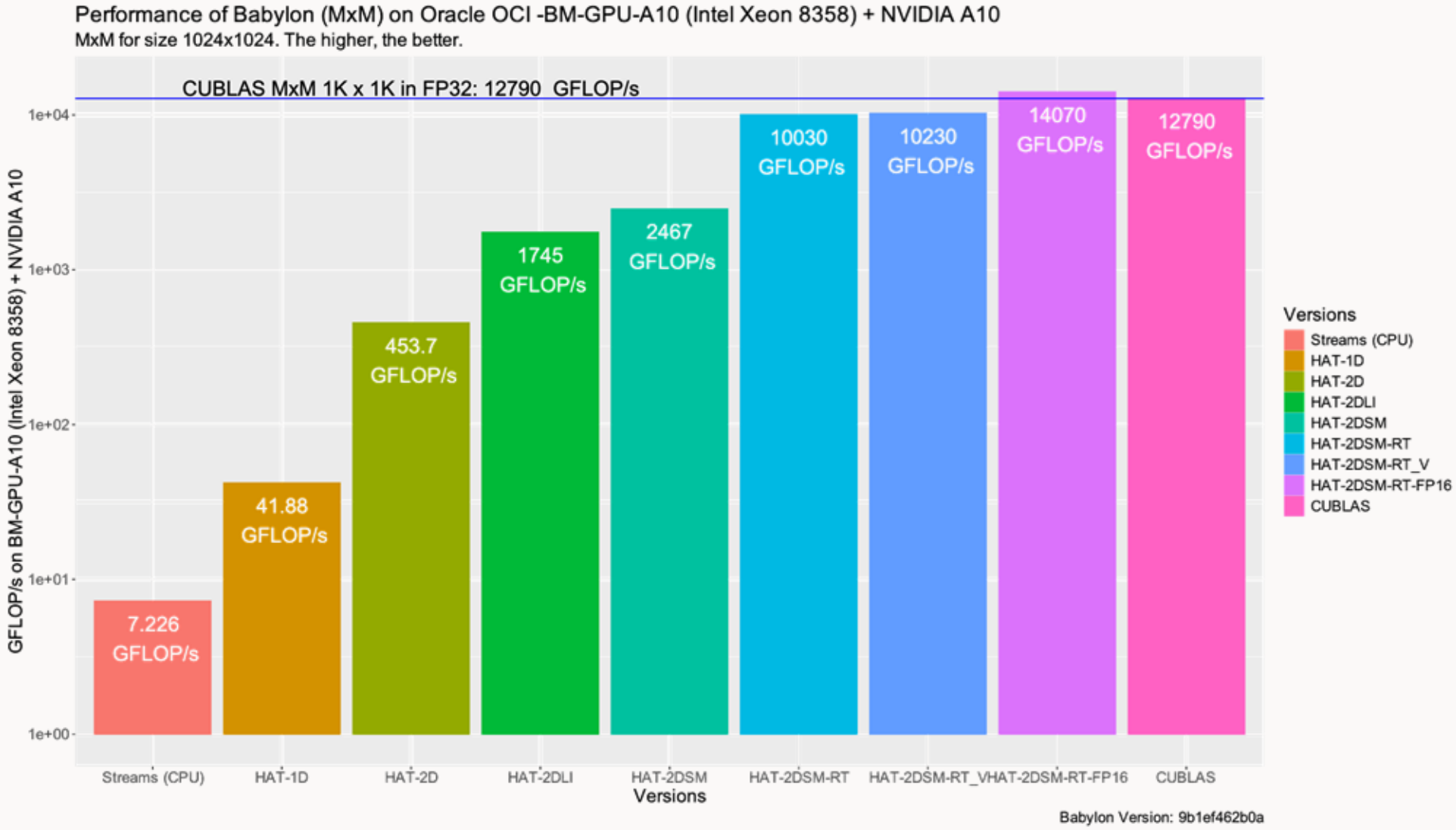
Metric Name	Metric Unit	Metric Value
DRAM Frequency	Ghz	6.24
SM Frequency	Mhz	879.61
Memory Throughput	%	58.28
Duration	**us**	**241.98**
...		
Compute (SM) Throughput	%	69.02

241 microseconds to run the same size matmul with cuBLAS on the same GPU.

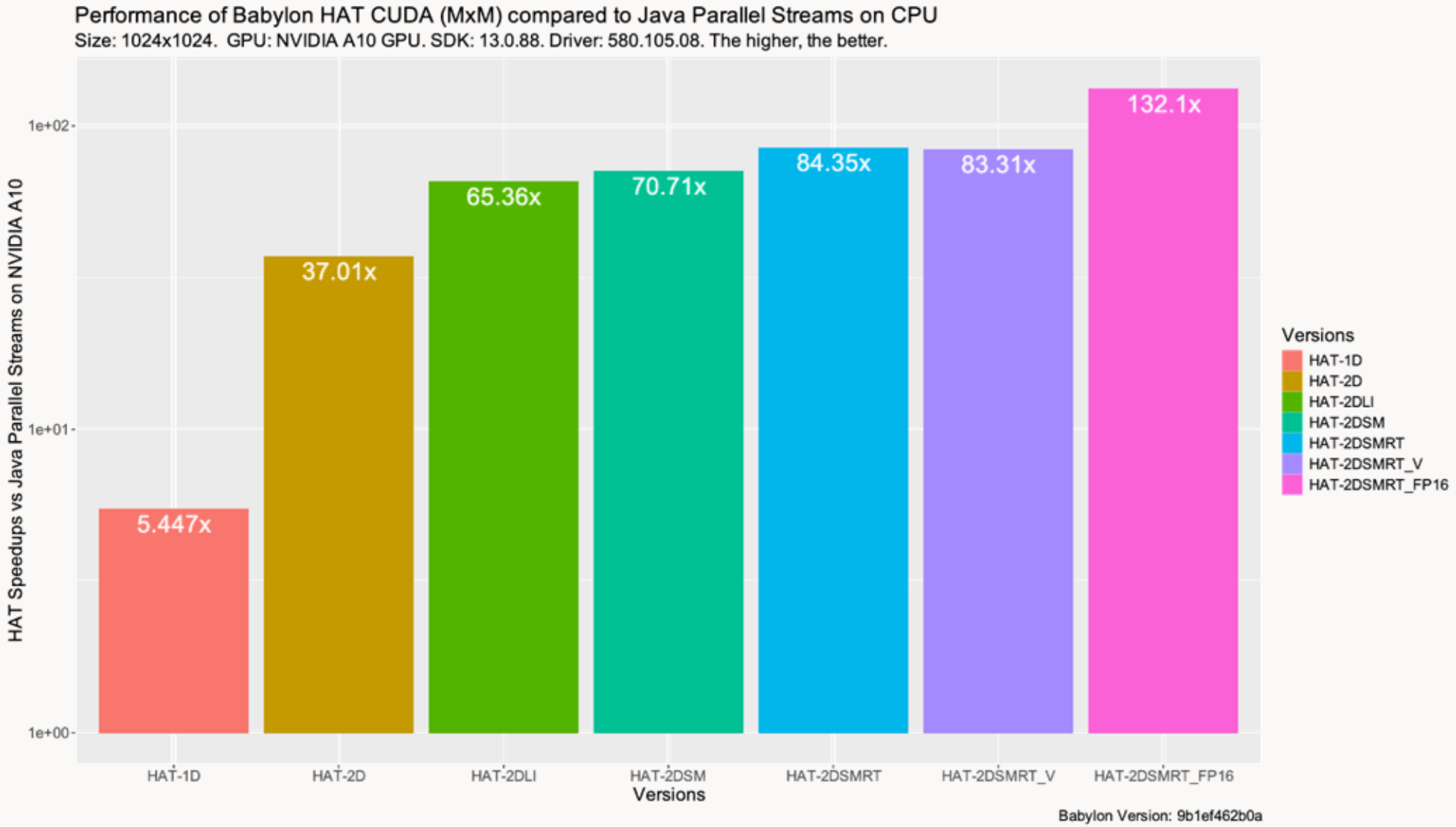
Performance with Nvidia Nsight Compute (NCU)



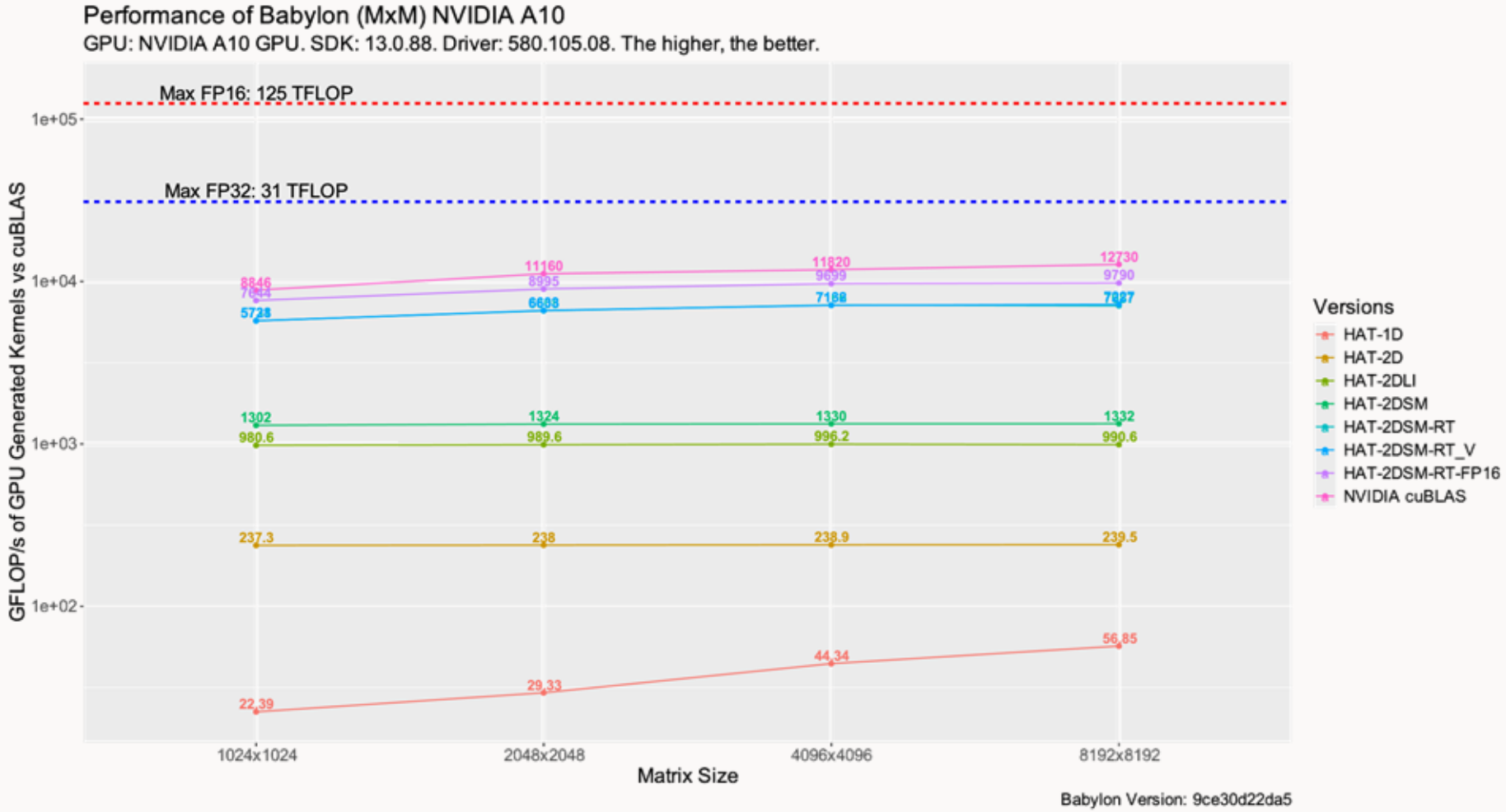
Performance with CUDA Events



End-To-End Performance (including copies)

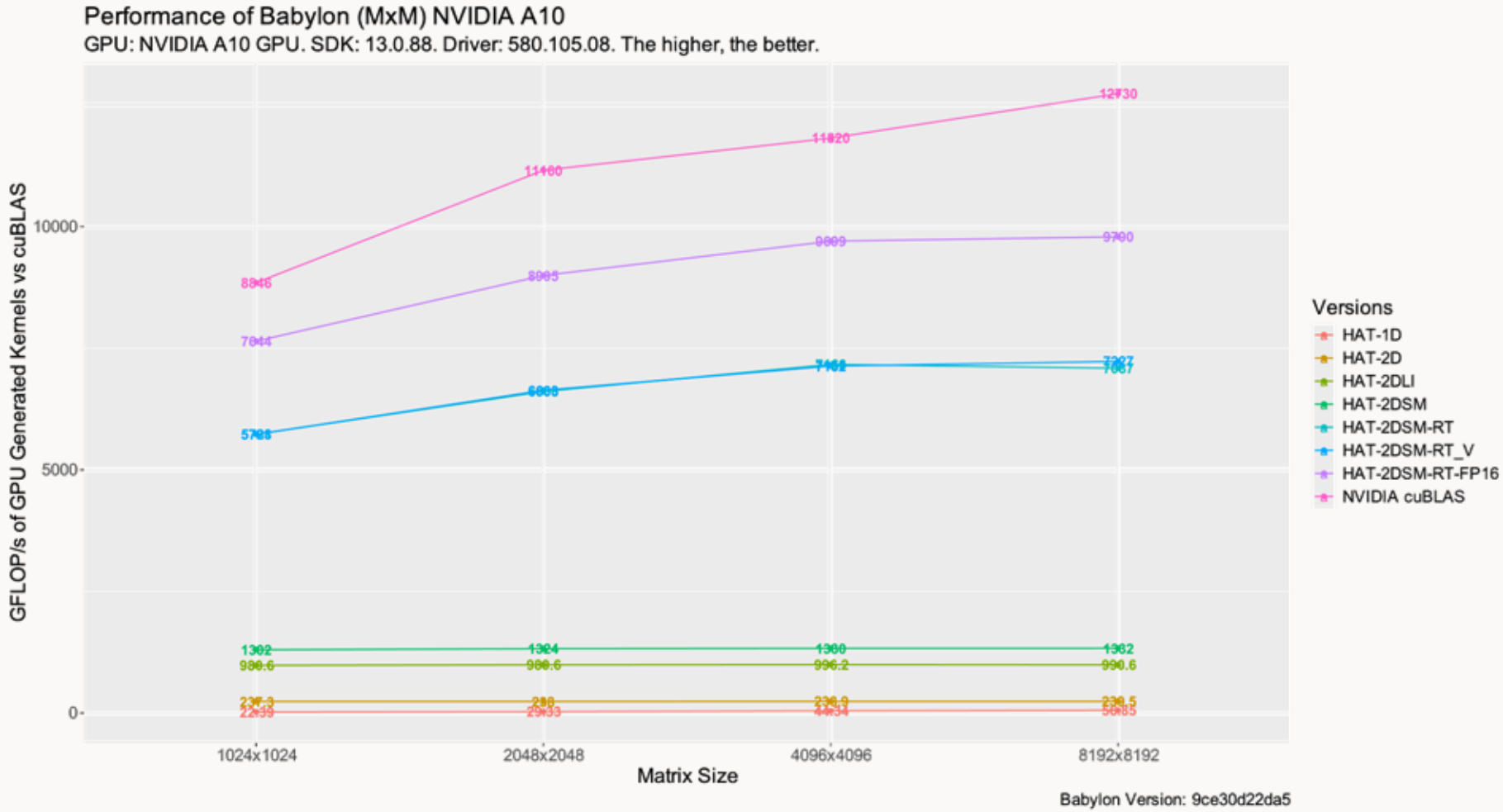


Running with Large Matrix Sizes v1



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

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.

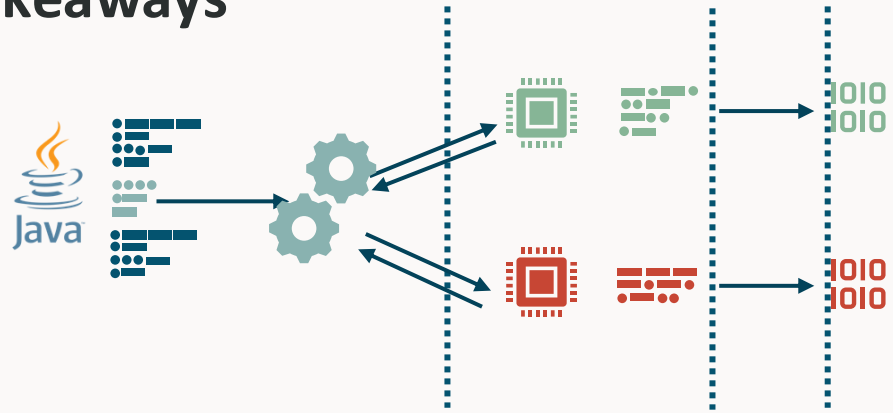
What’s next?

What next?

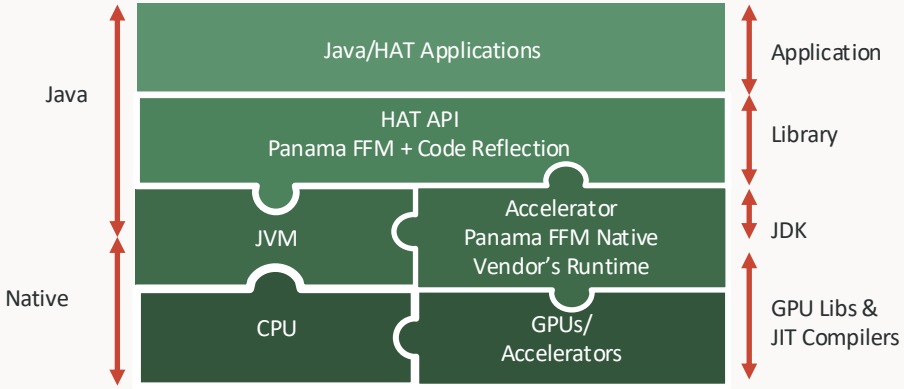
- Code Reflection is under incubation process (without the GPU).
- Working on Dialects for Hardware Accelerators and Code Reflection
- Working on providing programming abstractions for facilitating GPU programming within Java

Conclusions

Takeaways



HAT tackles all SW-Stack: From API to CodeGen



Extensible System for Hardware Acceleration in Java

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

Minimal Cost for UDT
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