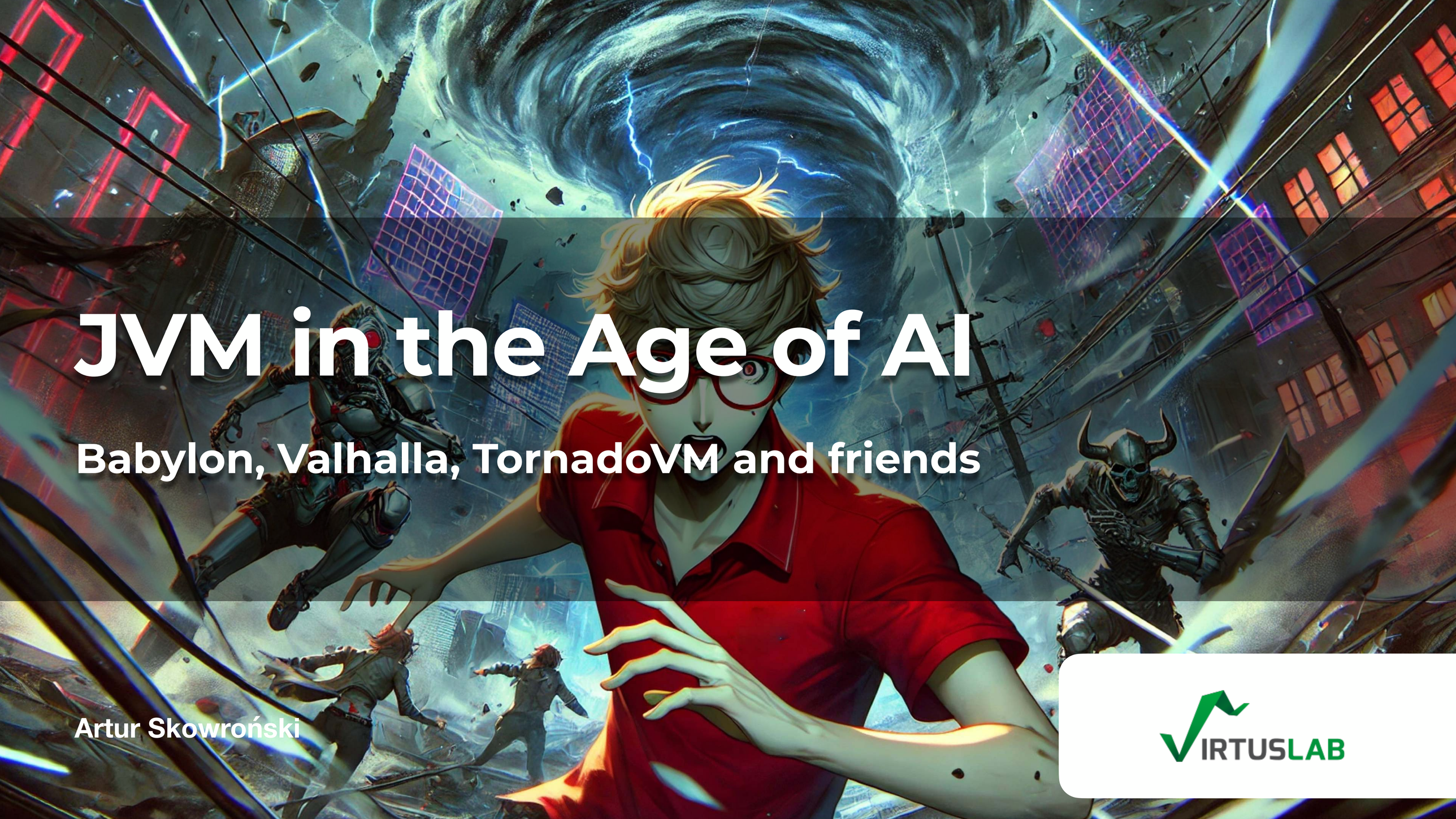


JVM in the Age of AI

Babylon, Valhalla, TornadoVM and friends

Artur Skowroński

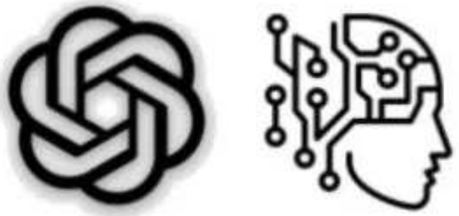
VIRTUSLAB

THE FASTEST THINGS ON EARTH

CHEETAH

AIRPLANE

SPEED OF LIGHT



PEOPLE BECOMING EXPERTS IN AI

Artur Skowronski

Head of Java / Kotlin Development

VIRTUSLAB

- **Using LLM through API** (Langchain4j, Semantic Kernel, Spring AI etc.)
- **Data Engineering** (data collection, data cleaning, etc.)
- **Model Development** (MLOps, observability etc.)

What we won't be talking about today...

- The **usage** and **inference** from existing models
- **Mainly about GPU programming** 😀

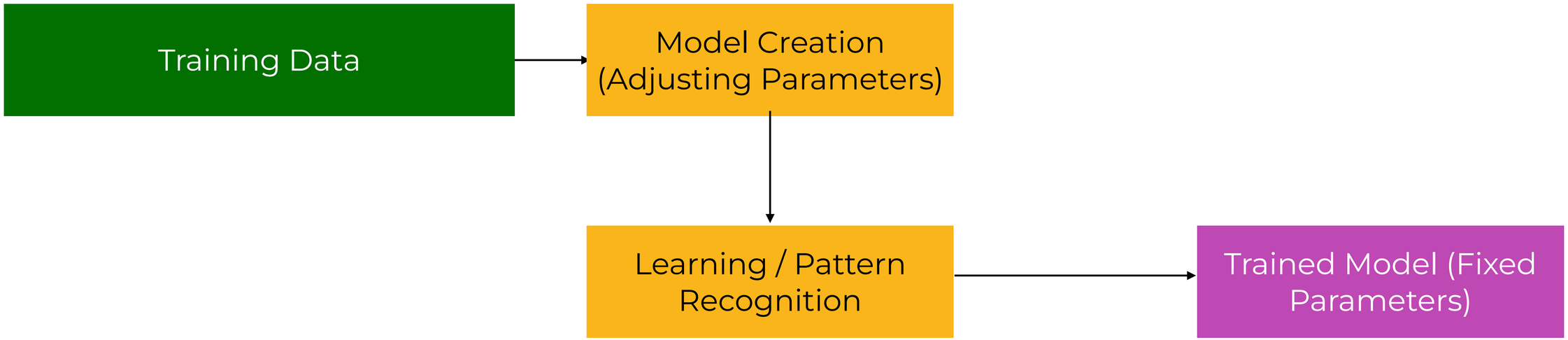
What we will be talking about today!

- The **usage** and **inference** from existing models
- **Mainly about GPU programming** 😃

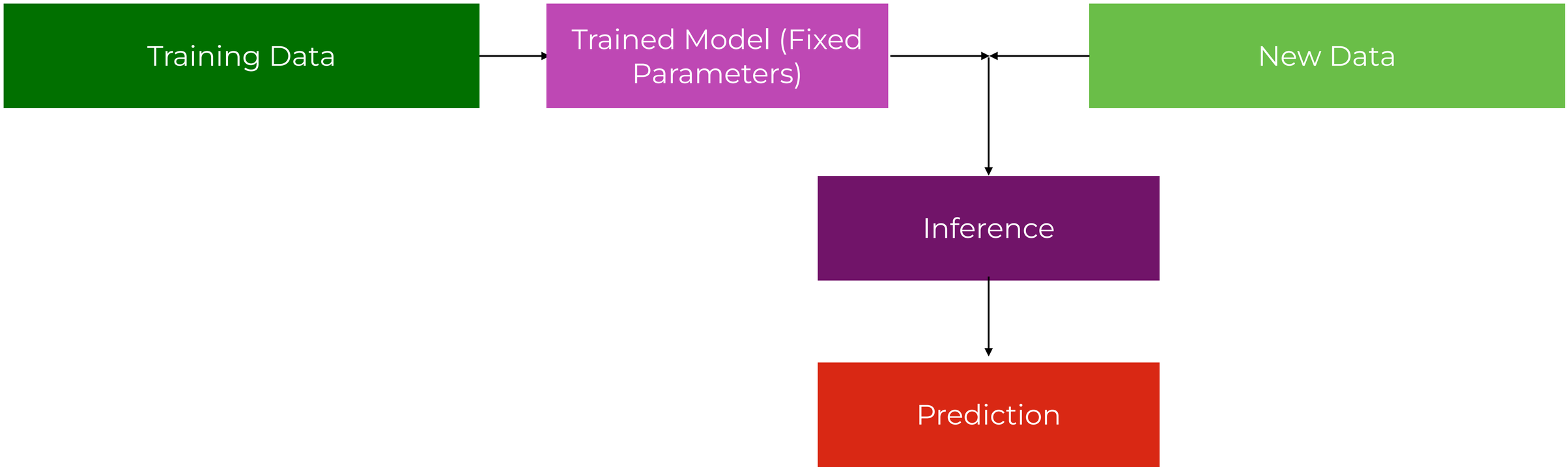
- ...but if I had told you about it,
  then probably no one would have shown up 😄

What we will be talking about today!

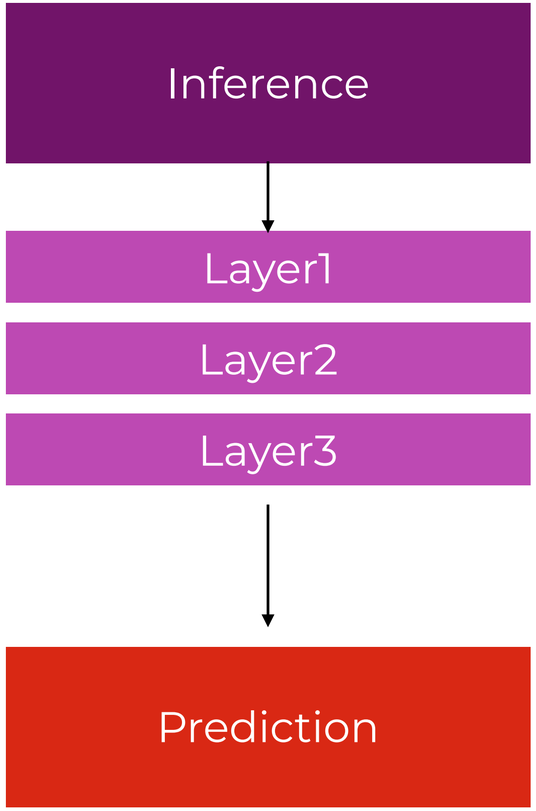
120 Slides / ? Minutes



Model Training



Model Inference



Next Token of Text
Classification of Image

Model Inference

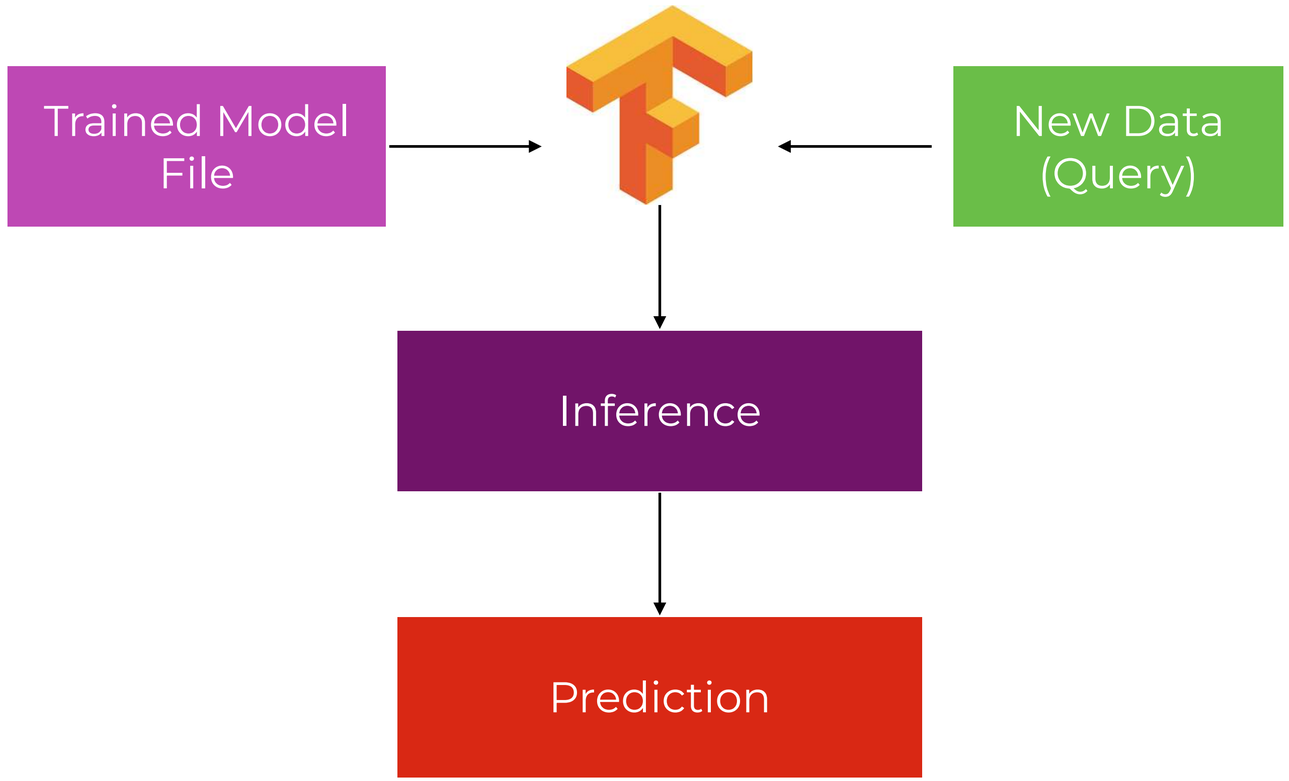
If you want to better visualize it, imagine that a transformer model might need to process billions of parameters (**weight values**) just to generate a response from a single sequence of text.

All that machinery works in the background to give you "simple" predictions or answers.

A lot of computation!

Inference in practice is often implemented using tools and frameworks such as **TensorFlow**, **PyTorch**, or **scikit-learn**.

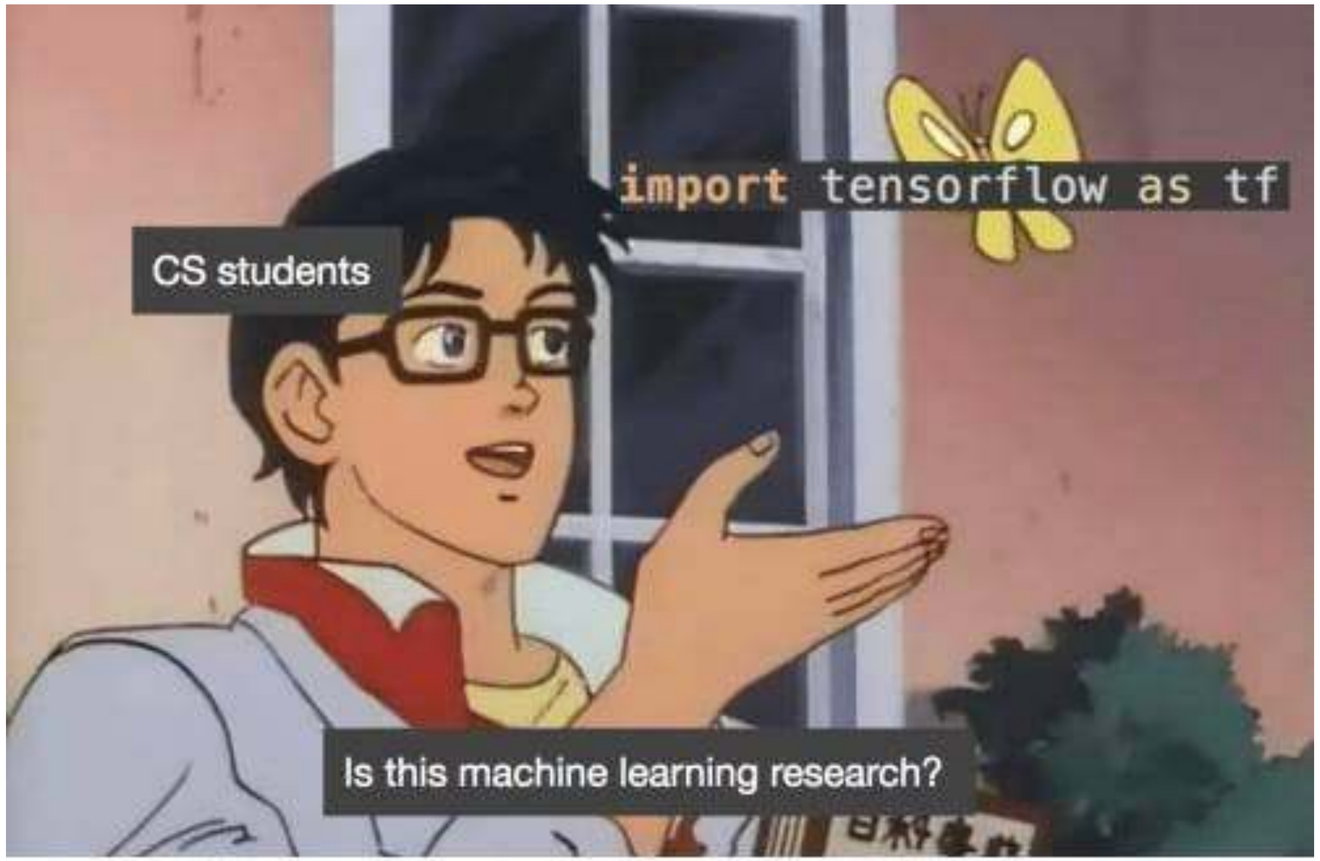
In a typical scenario, a developer loads the trained model (e.g., as a model file), provides it with input data (e.g., images, text, numbers), and the model returns predictions.



Model Inference

Inference in practice is often implemented using tools and frameworks such as **TensorFlow**, **PyTorch**, or **scikit-learn**.

In a typical scenario, a developer loads the trained model (e.g., as a model file), provides it with input data (e.g., images, text, numbers), and the model returns predictions.



Programmers Nowadays

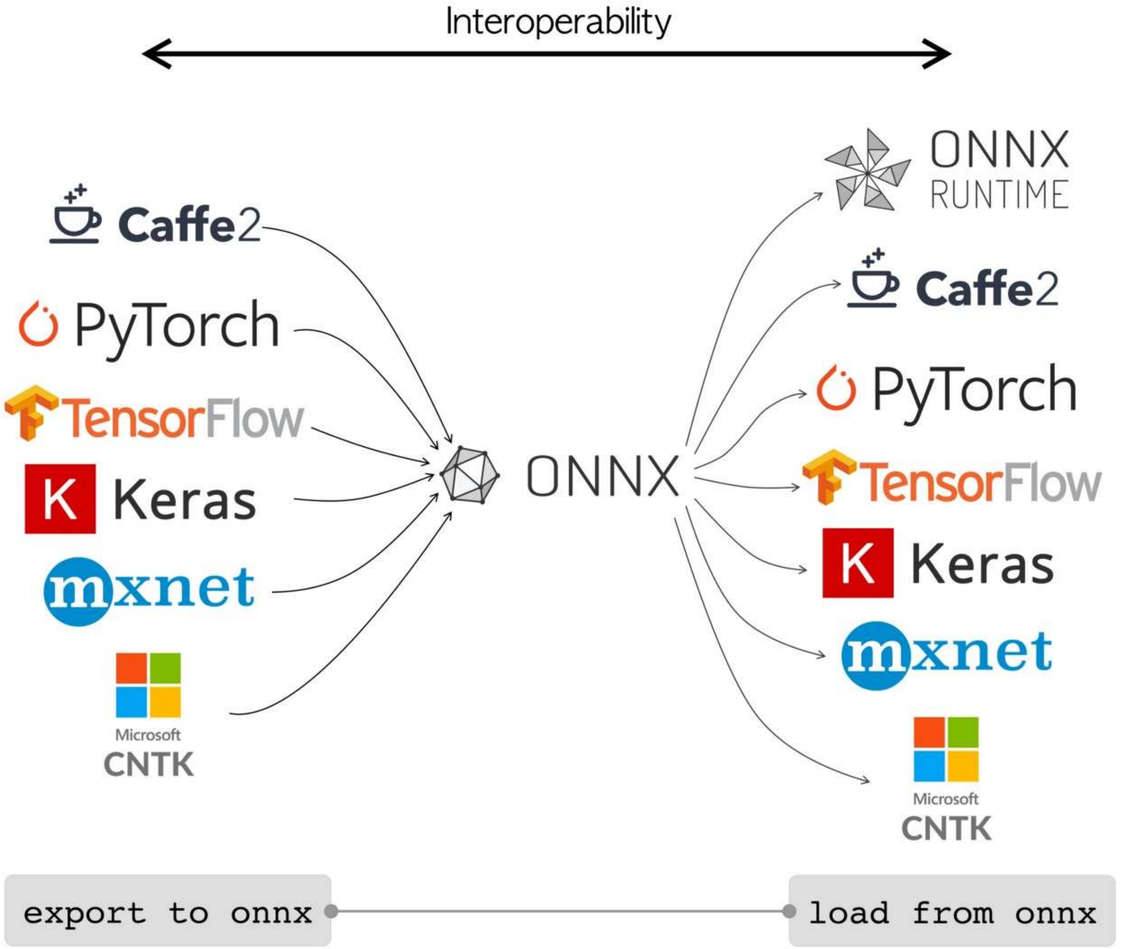
→ .h5 (weights & architecture)

→ .pkl (Pickle)

→ .pt

→ .pb

Different Model Formats



ONNX - SQL(?) for ML Models

Limited compatibility with frameworks and custom operations

ONNX doesn’t fully support all features and advanced layers of popular frameworks (e.g., PyTorch, TensorFlow), especially when dealing with custom operations or layers.

Optimization and performance issues

Converted models in ONNX may run less efficiently compared to their original framework versions. The conversion process can result in suboptimal code, leading to slower performance or higher resource consumption, especially with custom layers that have to be replaced with less efficient alternatives.

Challenges with specific tools and version compatibility

Rapid ONNX updates can cause compatibility issues between different opset versions, making it difficult to maintain older models or ensure smooth conversions across versions.

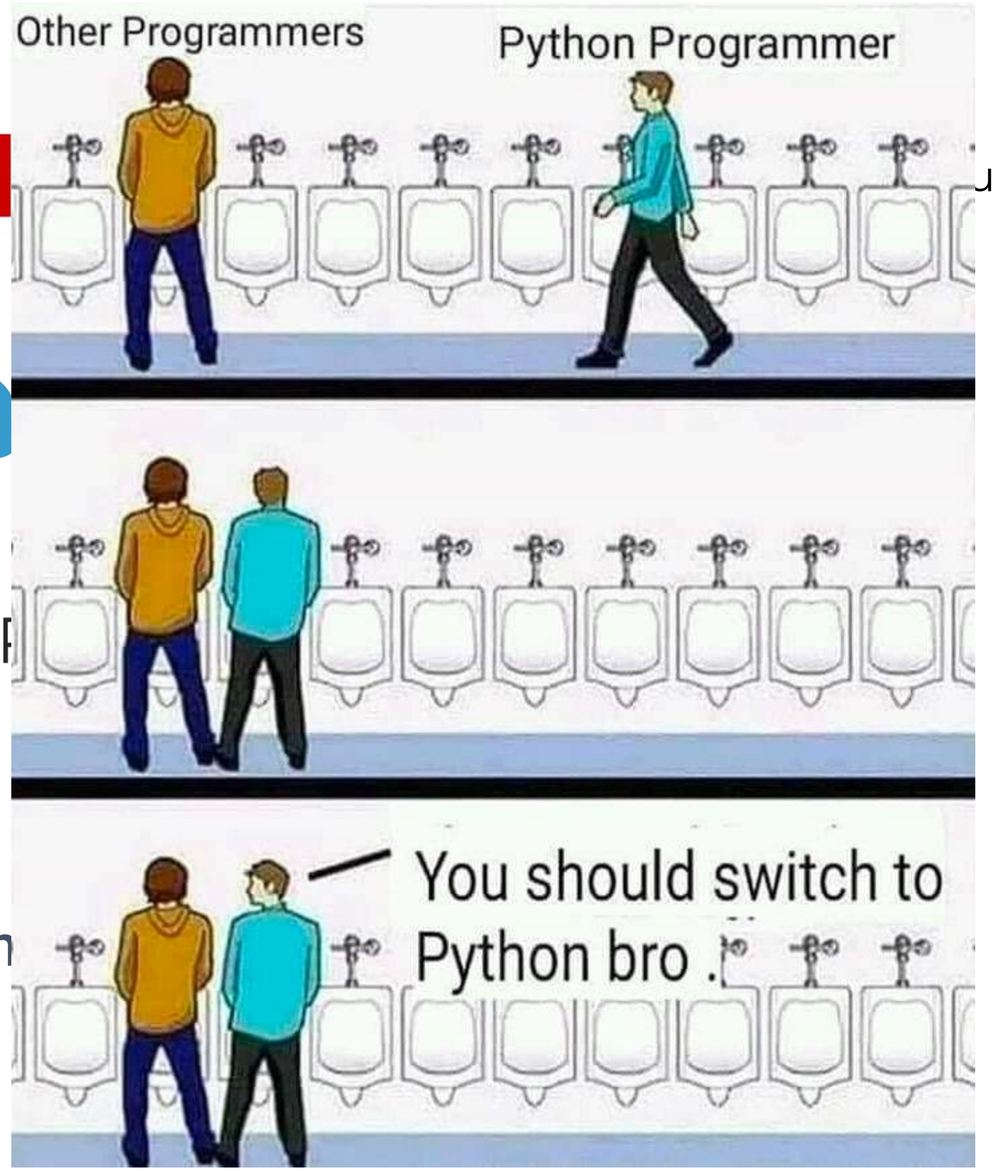
.h5 (weights & architecture)

.pkl (Pickle)

.pt

.pb

Rodzaje plików modeli



All these frameworks are Python Based

Use Python Inference libraries Libraries with JVM

GraalVM

One VM to Rule Them All

Thomas Würthinger[*] Christian Wimmer[*] Andreas Wöß[†] Lukas Stadler[†]
Gilles Duboscq[†] Christian Humer[†] Gregor Richards[§] Doug Simon[*] Mario Wolczko[*]

[*]Oracle Labs [†]Institute for System Software, Johannes Kepler University Linz, Austria [§]S[3] Lab, Purdue University
{thomas.wuerthinger, christian.wimmer, doug.simon, mario.wolczko}@oracle.com
{woess, stadler, duboscq, christian.humer}@ssw.jku.at gr@purdue.edu

Abstract

Building high-performance virtual machines is a complex and expensive undertaking; many popular languages still have low-performance implementations. We describe a new approach to virtual machine (VM) construction that amortizes much of the effort in initial construction by allowing new languages to be implemented with modest additional effort. The approach relies on abstract syntax tree (AST) interpretation where a node can rewrite itself to a more specialized or more general node, together with an optimizing compiler that exploits the structure of the interpreter. The compiler uses speculative assumptions and deoptimization in order to produce efficient machine code. Our initial experience suggests that high performance is attainable while preserving a modular and layered architecture, and that new high-

These implementations can be characterized in the following way:

- Their performance on typical applications is within a small multiple (1-3x) of the best statically compiled code for most equivalent programs written in an unsafe language such as C.
- They are usually written in an unsafe, systems programming language (C or C++).
- Their implementation is highly complex.
- They implement a single language, or provide a bytecode interface that advantages a narrow set of languages to the detriment of other languages.

In contrast, there are numerous languages that are popu-

Self-Optimizing AST Interpreters

Thomas Würthinger* Andreas Wöß† Lukas Stadler† Gilles Duboscq†
Doug Simon* Christian Wimmer*

*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria
thomas.wuerthinger@oracle.com woess@ssw.jku.at stadler@ssw.jku.at duboscq@ssw.jku.at
doug.simon@oracle.com christian.wimmer@oracle.com

Abstract

An abstract syntax tree (AST) interpreter is a simple and natural way to implement a programming language. However, it is also considered the slowest approach because of the high overhead of virtual method dispatch. Language implementers therefore define bytecodes to speed up interpretation, at the cost of introducing inflexible and hard to maintain bytecode formats. We present a novel approach to implementing AST interpreters in which the AST is modified during interpretation to incorporate type feedback. This tree rewriting is a general and powerful mechanism to optimize many constructs common in dynamic programming languages. Our system is implemented in Java™ and uses the static typing and primitive data types of Java elegantly to avoid the cost of boxed representations of primitive values in dynamic programming languages.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments, Optimization

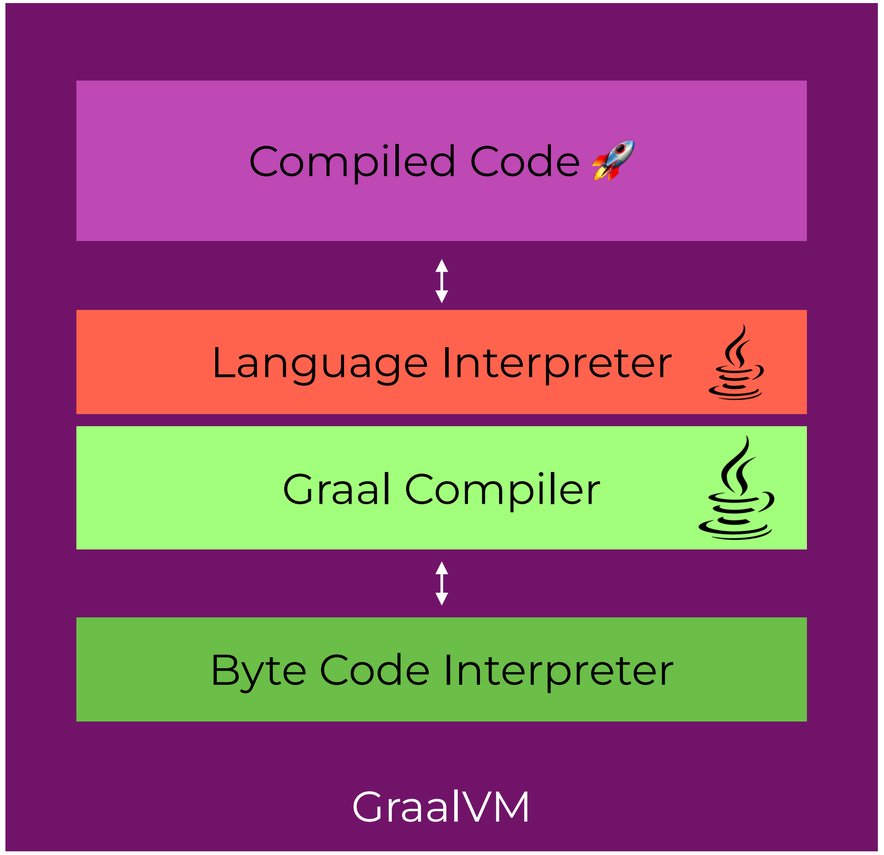
General Terms Algorithms, Languages, Performance

Keywords Java, JavaScript, dynamic languages, virtual machine, language implementation, optimization

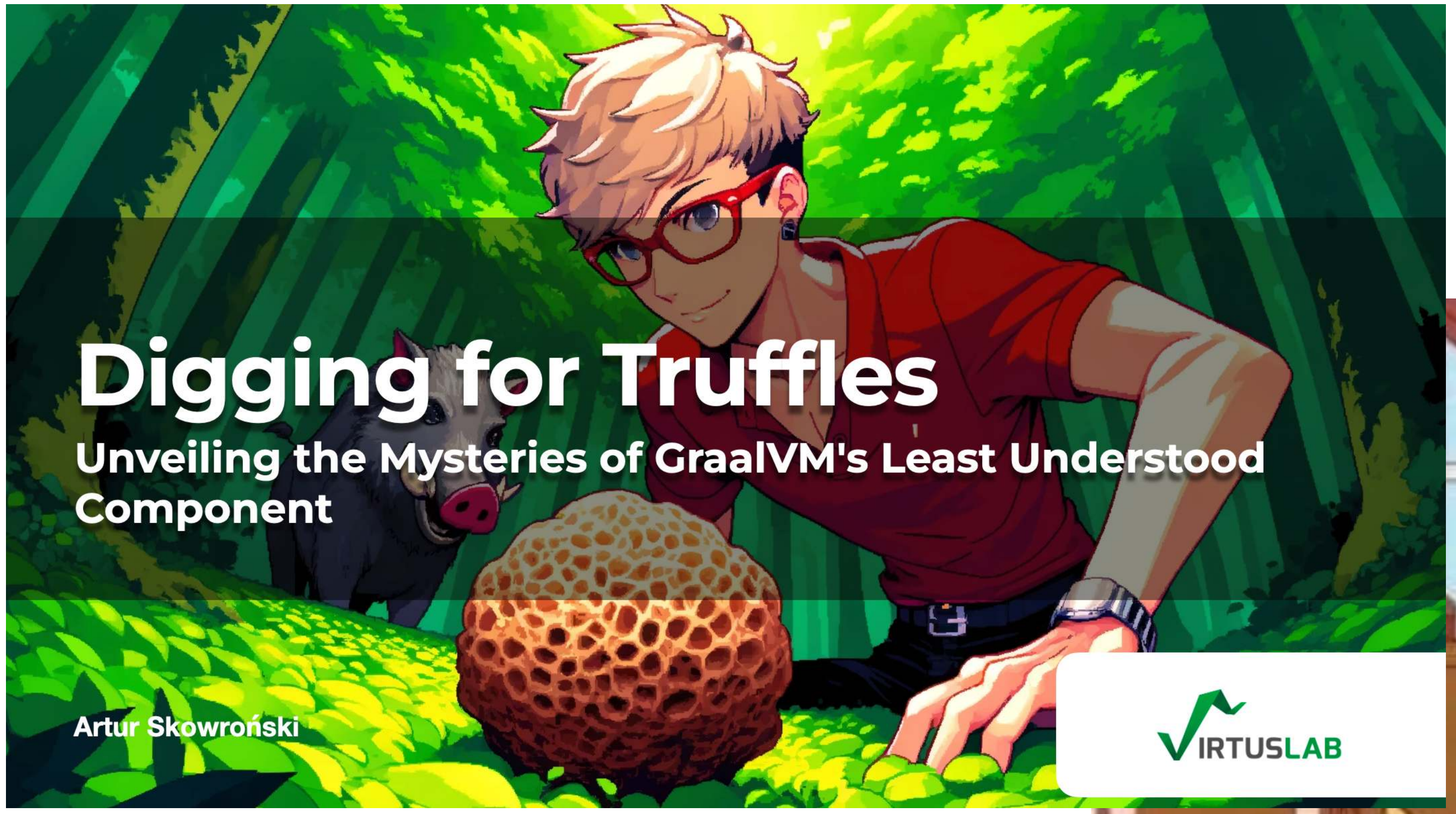
studied [9, 22]. Bytecodes are a virtual instruction set, i.e, they can be defined freely by language implementers without worrying about hardware constraints. There are two different strategies when defining bytecodes. We believe that both are cumbersome when implementing a new language:

- Language-independent bytecodes aim to provide an instruction set that is suitable for many languages. Java™ bytecodes [17], originally designed primarily for the Java programming language, are now used for many languages. The Common Intermediate Language (CIL) [8], which is used by the .NET system, was specifically designed for multiple languages. Language-independent bytecodes provide only general instructions whose semantics do not match the language semantics exactly. Long bytecode sequences are potentially required to achieve the desired language behavior. For example, Java bytecodes are statically typed, so compiling dynamically typed JavaScript requires frequent type checks. A simple addition in JavaScript requires checking whether the operands are numbers, strings, or arbitrary objects, which all require different handling. Therefore, implementations of JavaScript that run on a Java VM often call a method for a seemingly simple addition.

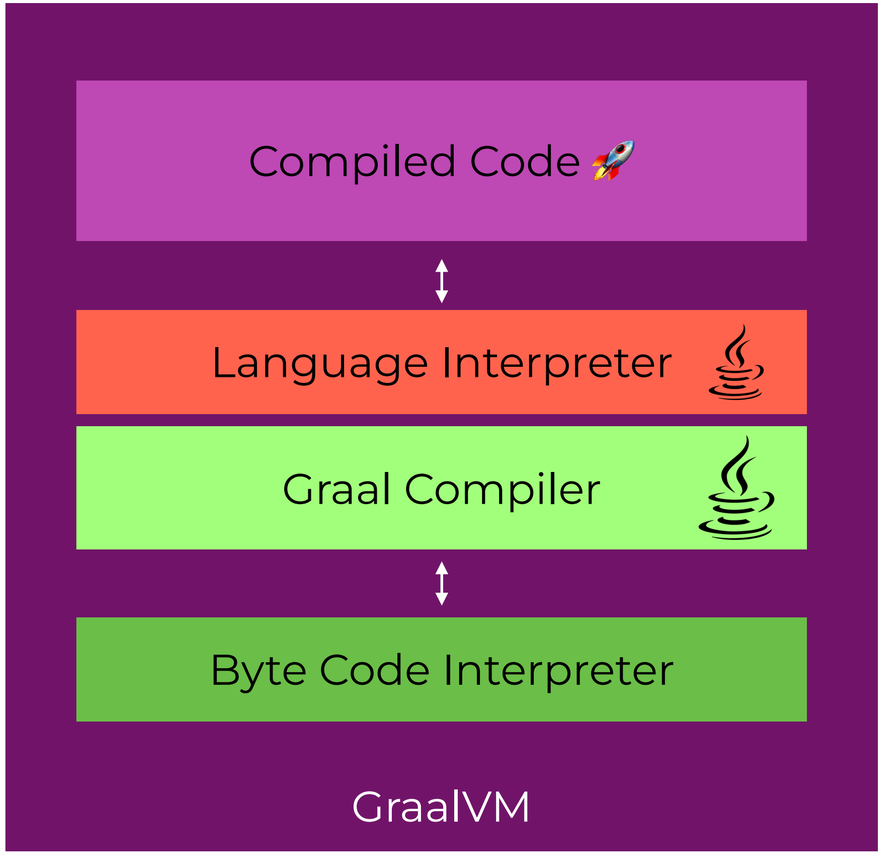
Truffle & GraalPy



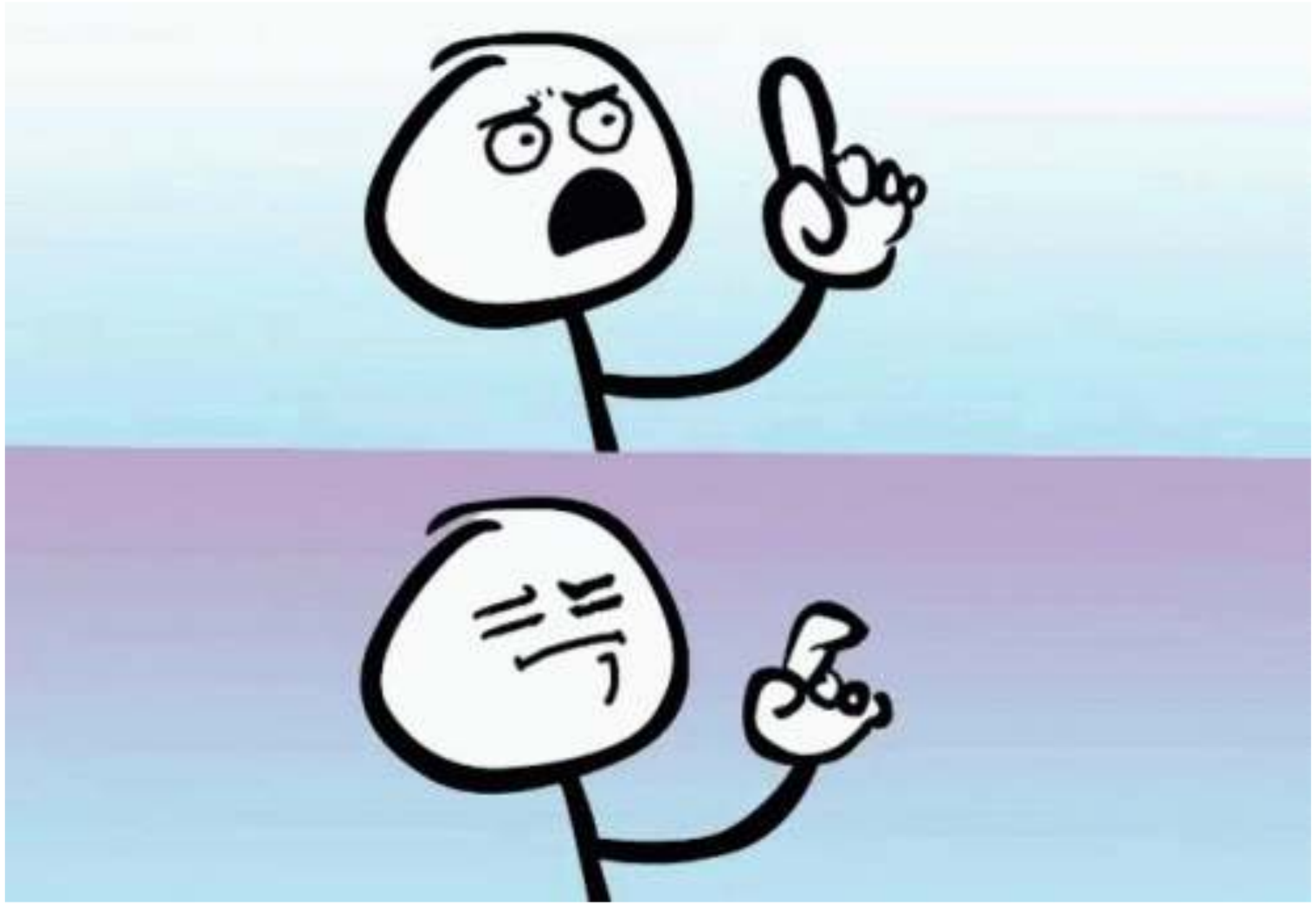
GraalVM in Truffle Mode



Truffle & GraalPy

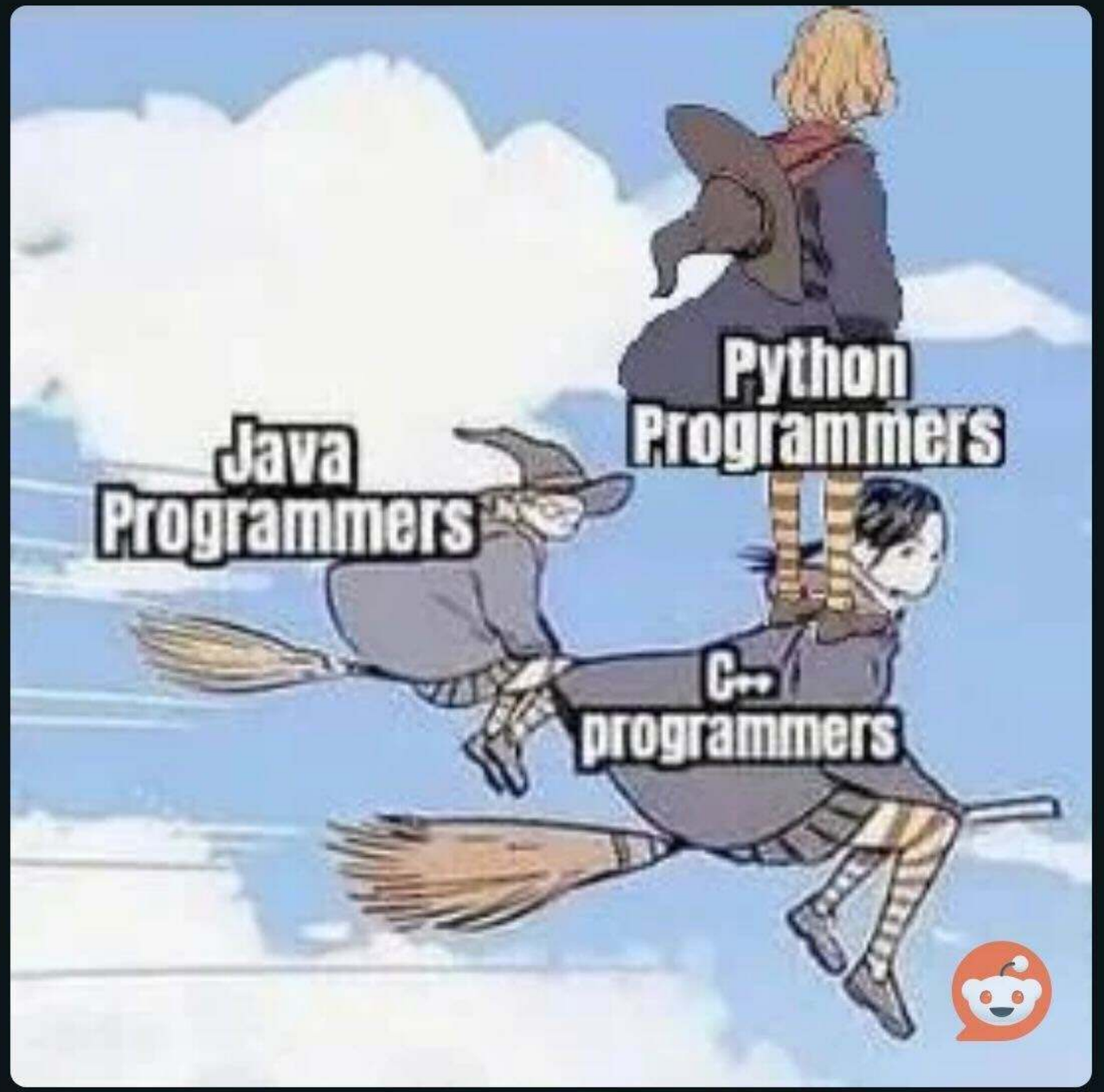


GraalVM in Truffle Mode



Truffle & GraalPy

From **ProgrammerHumor** community on **Reddit**



Experimental

Home › Dev › Reference Manual › Python ›

Native Extensions Support

CPython provides a native extensions API for writing Python extensions in C/C++. GraalPy provides experimental support for this API, which allows many packages like NumPy and PyTorch to work well for many use cases. The support extends only to the API, not the binary interface (ABI), so extensions built for CPython are not binary compatible with GraalPy. Packages that use the native API must be built and installed with GraalPy, and the prebuilt wheels for CPython from pypi.org cannot be used. For best results, it is crucial that you only use the `pip` command that comes preinstalled in GraalPy virtualenvs to install packages. The version of `pip` shipped with GraalPy applies additional patches to packages upon installation to fix known compatibility issues and it is preconfigured to use an additional repository from graalvm.org where we publish a selection of prebuilt wheels for GraalPy. Please do not update `pip` or use alternative tools such as `uv`.

Embedding limitations

Python native extensions run by default as native binaries, with full access to the underlying system. Native code is entirely unrestricted and can circumvent any security protections Truffle or the JVM may provide. Native data structures are not subject to the Java GC and the combination of them with Java data structures may lead to memory leaks. Native libraries generally cannot be loaded multiple times into the same process, and they may contain global state that cannot be safely reset. Thus, it is not possible to create multiple GraalPy contexts that access native modules within the same JVM. This includes the case when you create a context, close it, and then create another context. The second context will not be able to access native extensions.

Go a Bit More Complex: GraalPy

What's new in Truffle 24.0 and Graal Languages

Alina Yurenko · Following

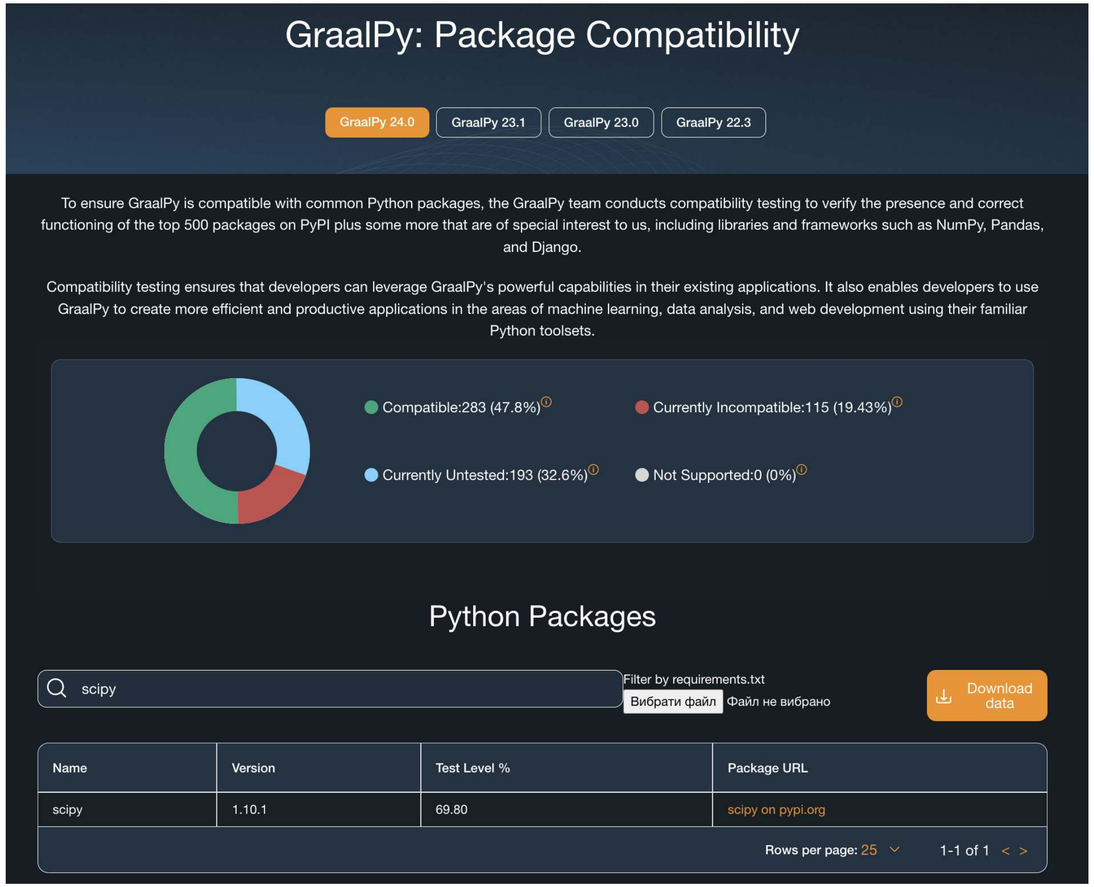
Published in graalvm · 5 min read · Mar 19, 2024

GraalPy 🐍

Compatibility ✅

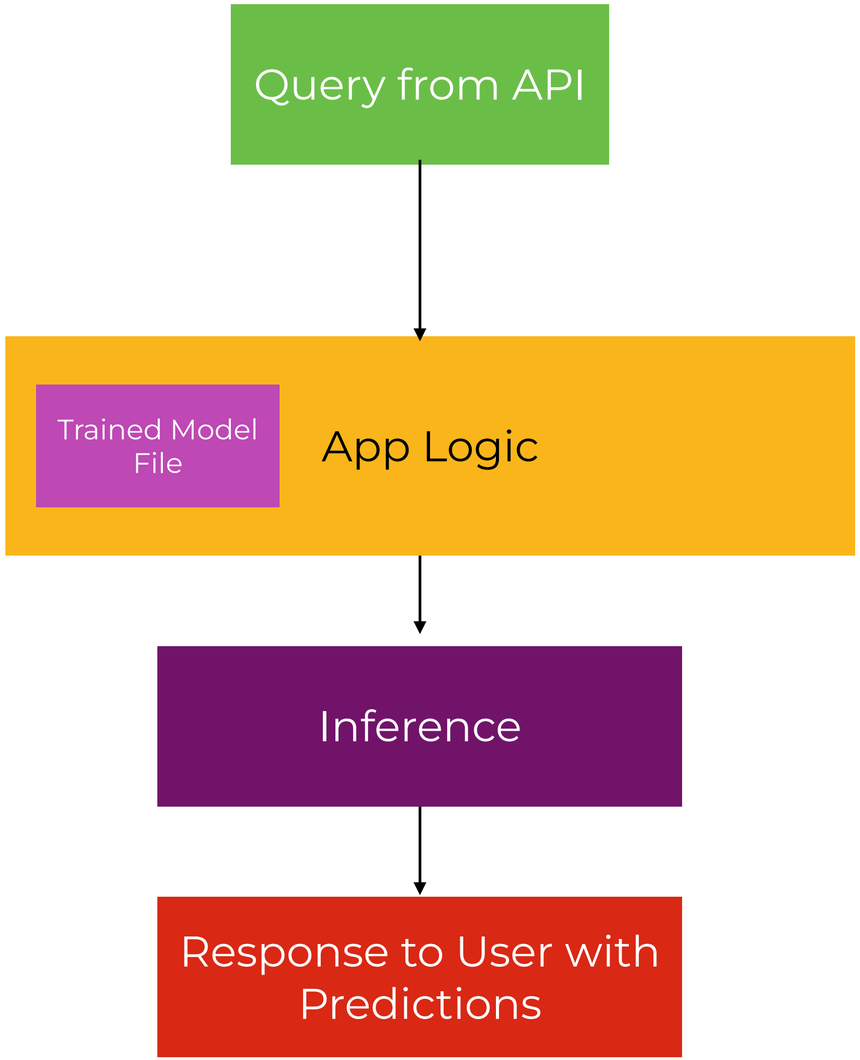
We are working hard to make more and more libraries available on GraalPy. In this release, we expanded support for the following: `tensorflow`, `tensorflow-io`, `protbuf`, `llvmlite`, `pydantic-core`, `catboost`, `ray`, `readme-renderer`, `safetensors`, `keras`, `pybind11`, `grpcio`, `PyO3`, `cryptography`, `bcrypt`, `cramjam`, `libcst`, `orjson`, `rpds_py`. Try them out, provide feedback, and let us know what other libraries you would like to run on GraalPy.

Go a Bit More Complex: GraalPy



Go a Bit More Complex: GraalPy

Inference of Model from Java



Model Inference

DL4J is a popular ML library in Java that allows for easy loading of models trained in other environments (e.g., Keras, TensorFlow).

```java
import org.deeplearning4j.nn.multilayer.MultiLayerNetwork;
import org.deeplearning4j.nn.api.Model;
import org.nd4j.linalg.dataset.api.preprocessor.DataNormalization;
import org.nd4j.linalg.dataset.api.preprocessor.NormalizerStandardize;
import org.nd4j.linalg.factory.Nd4j;
import org.nd4j.linalg.api.ndarray.INDArray;

import java.io.File;

public class ModelInferenceExample {

    public static void main(String[] args) throws Exception {

        File modelFile = new File("path/to/saved/model.zip");
        MultiLayerNetwork model = MultiLayerNetwork.load(modelFile, true);

        INDArray input = Nd4j.create(new double[]{0.5, 1.0, 0.8}, new int[]{1, 3});

        DataNormalization normalizer = new NormalizerStandardize();
        normalizer.transform(input);

        INDArray output = model.output(input);
        System.out.println("Wynik inferencji: " + output);
    }
}
```

Deeplearning4j (DL4J)

Tribuo

Tribuo is a relatively new machine learning library for Java, designed by Oracle Labs.

Its main goal is to simplify the creation, training, and deployment of machine learning models in Java applications.

```java
import org.tribuo.classification.Label;
import org.tribuo.data.csv.CSVLoader;
import org.tribuo.data.columnar.FieldResponseProcessor;
import org.tribuo.Dataset;
import org.tribuo.MutableDataset;
import org.tribuo.Example;
import org.tribuo.Prediction;

import java.io.File;

public class TribuoInferenceExample {

    public static void main(String[] args) throws Exception {
        // Załaduj wytrenowany model
        File modelFile = new File("path/to/saved-model.model");
        Model<Label> model = Model.load(modelFile);

        // Przygotuj nowe dane do przewidywania (przykład dla CSV)
        CSVLoader<Label> loader = new CSVLoader<>(new FieldResponseProcessor("label",
Label::parseLabel));
        Dataset<Label> dataset = new MutableDataset<>(loader.loadDataSource(new File("path/to/new-
data.csv")));

        // Wykonaj inferencję dla nowego przykładu
        Example<Label> newExample = dataset.getExample(0);
        Prediction<Label> prediction = model.predict(newExample);
        System.out.println("Przewidywana etykieta: " + prediction.getOutput());
    }
}
```

Tribuo is a relatively new machine learning library for Java, designed by Oracle Labs.

Its main goal is to simplify the creation, training, and deployment of machine learning models in Java applications.

JSR 381 specifically focuses on providing standardized APIs for deep learning frameworks in Java, enabling developers to work with deep learning models in a consistent and interoperable manner. This standardization helps foster innovation, promote compatibility between different deep learning libraries, and streamline the development of deep learning applications within the Java ecosystem.

Key objectives of the JSR 381

Key objectives of the standard Java API for machine learning include:

- Defining common APIs for tasks such as building, training, evaluating, and deploying deep learning models.
- Ensuring compatibility and interoperability between different deep learning frameworks and tools in Java.
- Providing a consistent programming interface for developers to work with deep learning models, regardless of the underlying framework being used.
- Promoting best practices and standard patterns for deep learning development in Java.
- Facilitating the integration of deep learning capabilities into existing Java applications and frameworks.

The creation of JSR 381 reflects the growing importance of deep learning in Java and the need for standardized APIs to support its adoption and development within the Java community. By establishing a common set of interfaces and specifications, JSR 381 aims to simplify the development and deployment of deep learning solutions in Java, ultimately benefiting developers, organizations, and the broader Java ecosystem.

DeepJavaLibrary

JSRs are formal requests for defining new standards or specifications that impact the entire Java ecosystem, including Java SE (Standard Edition), Java EE (Enterprise Edition), and Java ME (Micro Edition).

JSRs follow a rigorous standardization process under the Java Community Process (JCP), which involves community involvement, public review, and approval by a governing body. Once accepted, JSRs become part of the official Java standard.

JSRs are for defining industry-wide Java standards, while JEPs propose enhancements to the Java platform’s core development tools and libraries

JSR (Java Specification Request)

JSRs define standards that affect the entire Java ecosystem, while JEPs are enhancements or proposals focused on improving the JDK itself.

JSRs have broader, ecosystem-wide implications and can introduce new APIs, while JEPs are more narrowly focused on technical improvements within the core JDK.

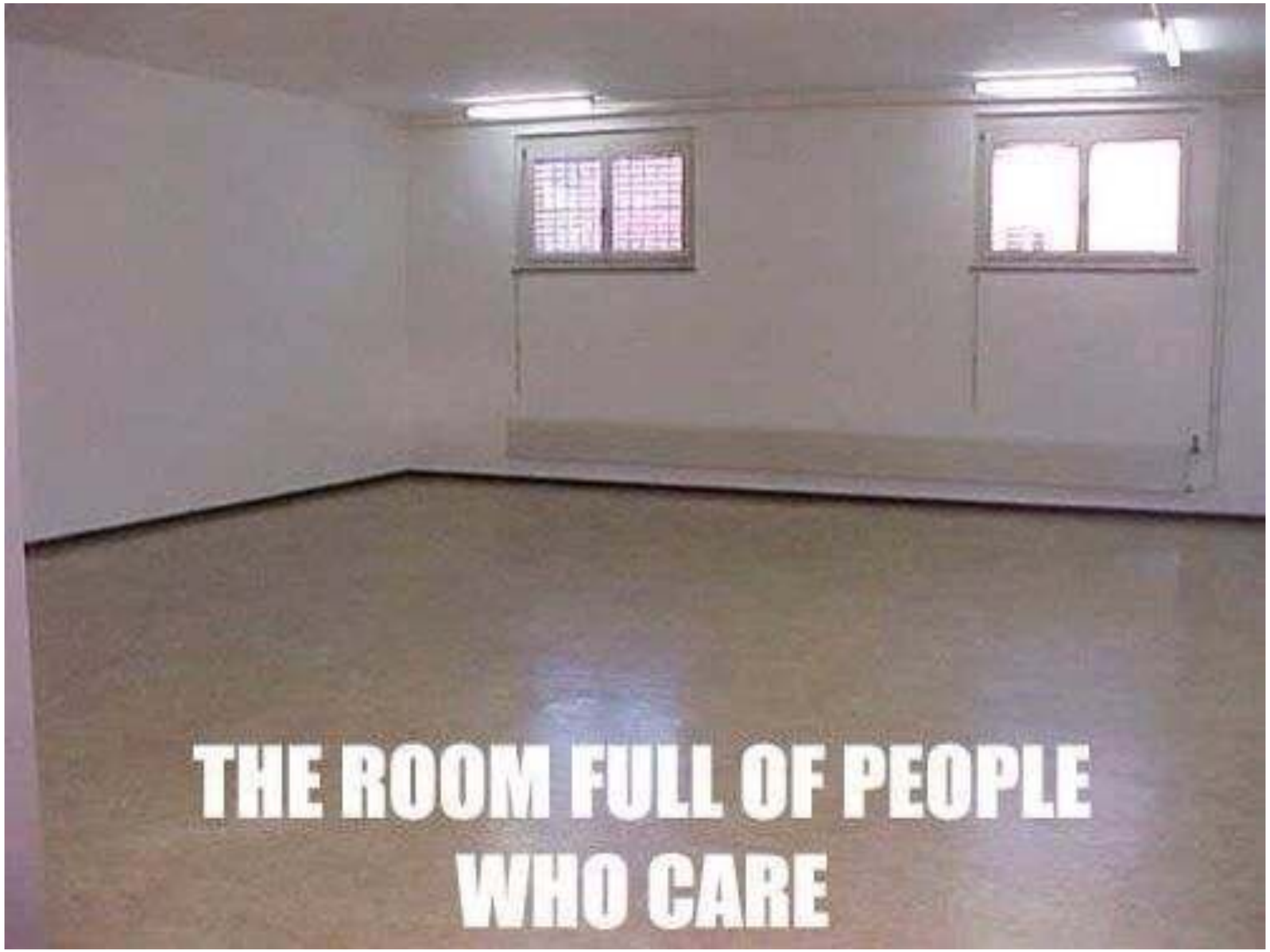
JSRs go through the Java Community Process (JCP) and involve formal approval, while JEPs are part of the more informal OpenJDK project governance.

Key Differences from JEP

- **JSR 381 – Visual Recognition (VisRec) API**
- JSR 376 – Java Platform Module System (Project Jigsaw)
- JSR 383 – Java SE 18.3 (Java 10)
- JSR 315 – Java Servlet 3.0
- JSR 380 – Bean Validation 2.0
- JSR 244 – Java EE 5
- JSR 303 – Bean Validation 1.0
- JSR 907 – JDBC 4.0
- JSR 133 – Java Memory Model and Thread Specification
- JSR 292 – Dynamically Typed Languages Support (invokedynamic)

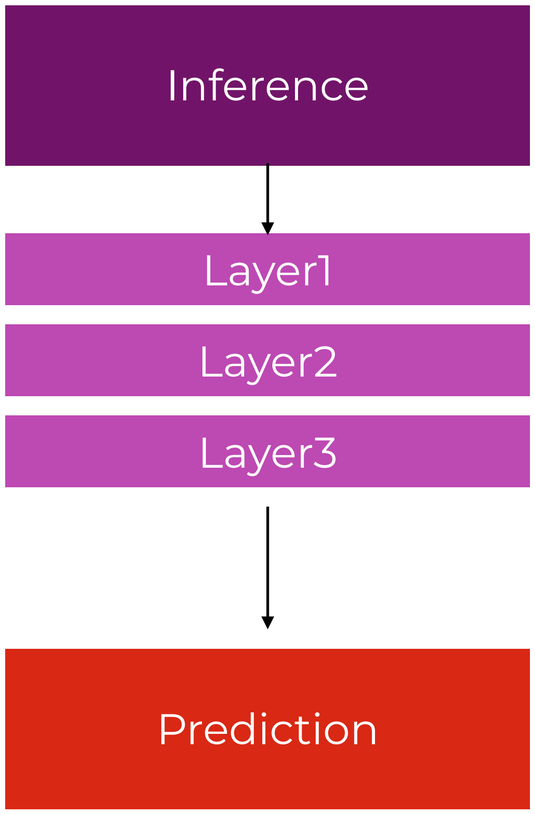
Key Differences from JEP

- Examples
  - Example 1 - Hello World
  - Example 2 - Simple Linear Regression
  - Example 3 - Logistic Regression
  - Example 4 - Iris Flower Classification
  - Example 5 - Handwritten Digit Recognition using the MNIST Data Set



Visual Recognition (VisRec) JSR #381

What about performance of the process?

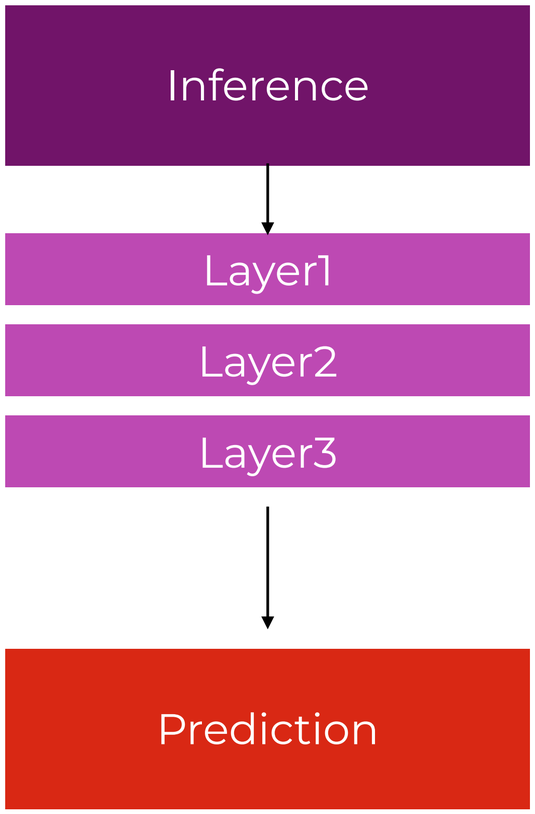


Next Token of Text
Classification of Image

Computational Complexity

Each layer of requires performing a vast number of mathematical operations, such as matrix multiplications. The more layers a model has, the greater the number of such operations.

Model Inference



Next Token of Text
Classification of Image

Model Size

Models with many layers typically have millions, or even billions, of parameters. Storing and processing such a large number of weights requires significant memory resources.

Model Inference

Optmization: Half Floats and Project Valhalla

Half-floats, or 16-bit floating-point numbers, are a simplified version of standard floating-point numbers (32-bit) used in calculations.

In standard floating-point numbers (commonly known as floats), we have 32 bits, with some allocated for the sign of the number, some for the exponent, and some for the mantissa, which determines precision.

1 bit for the sign (positive or negative)

8 bits for the exponent (which adjusts the range)

23 bits for the mantissa (which determines precision)

Half Floats

In half-floats, we only have 16 bits, meaning they can store smaller and **less precise numbers**.

1 bit for the sign,

5 bits for the exponent

10 bits for the mantissa

Half Floats

Half-precision floating-point format

7 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

Not to be confused with bfloat16, a different 16-bit floating-point format.

In computing, **half precision** (sometimes called **FP16** or **float16**) is a binary floating-point computer number format that occupies 16 bits (two bytes in modern computers) in computer memory. It is intended for storage of floating-point values in applications where higher precision is not essential, in particular image processing and neural networks.

Almost all modern uses follow the IEEE 754-2008 standard, where the 16-bit base-2 format is referred to as **binary16**, and the exponent uses 5 bits. This can express values in the range ±65,504, with the minimum value above 1 being 1 + 1/1024.

Depending on the computer, half-precision can be over an order of magnitude faster than double precision, e.g. 550 PFLOPS for half-precision vs 37 PFLOPS for double precision on one cloud provider.[1]

History [edit]

Several earlier 16-bit floating point formats have existed including that of Hitachi's HD61810 DSP of 1982 (a 4-bit exponent and a 12-bit mantissa),[2] Thomas J. Scott's WIF of 1991 (5 exponent bits, 10 mantissa bits)[3] and the 3dfx Voodoo Graphics processor of 1995 (same as Hitachi).[4]

ILM was searching for an image format that could handle a wide dynamic range, but without the hard drive and memory cost of single or double precision floating point.[5]

Floating-point formats

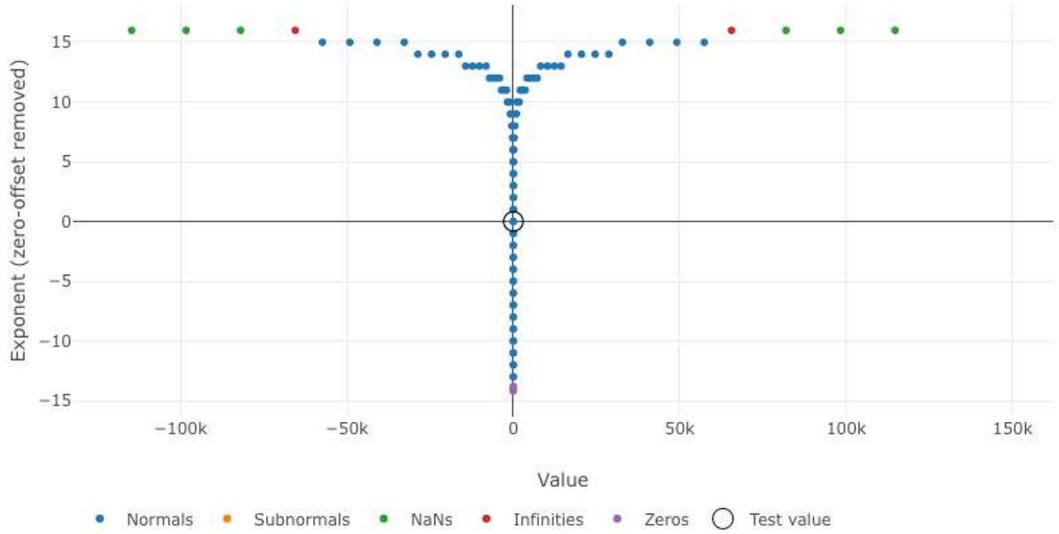
IEEE 754

16-bit: **Half** (binary16)
32-bit: Single (binary32), decimal32
64-bit: Double (binary64), decimal64
128-bit: Quadruple (binary128), decimal128
256-bit: Octuple (binary256)
Extended precision

Other

Minifloat

Half-Precision Floating-Point, Visualized



Test value

0 0 1 1 1 1 0 0 | 0 0 0 0 0 0 0 0

$1 = 1 \times 2^{0} \times \left(1 + \frac{0}{1024}\right)$

X-scale

◉ linear ○ log

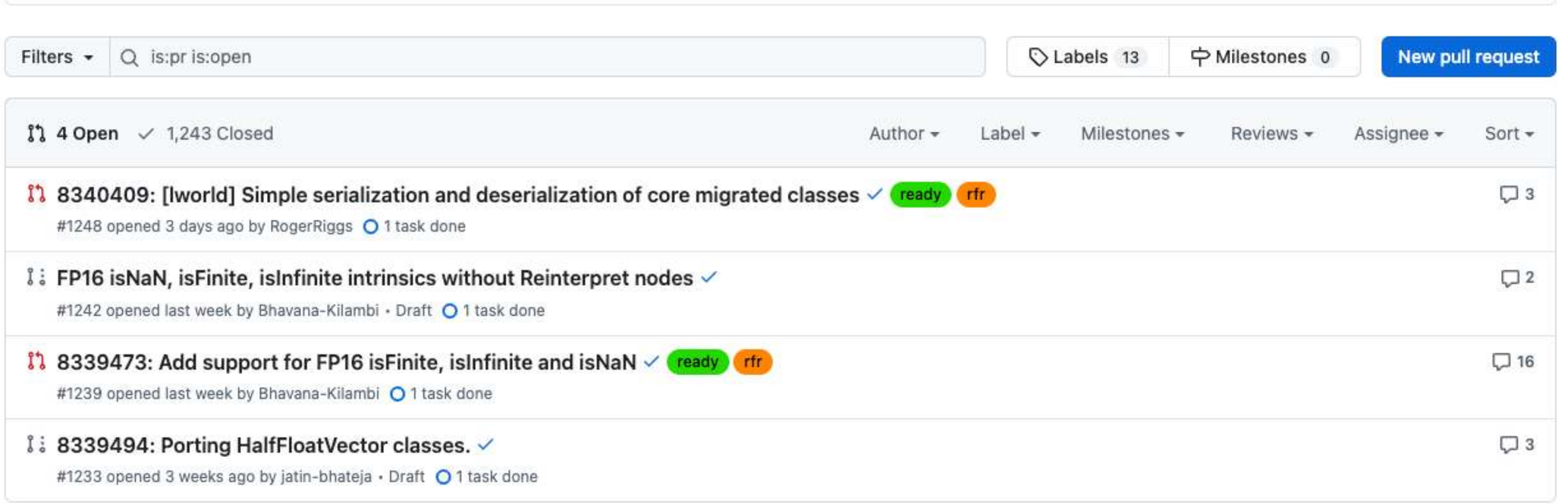
(log scaling displays only values greater than zero)

Bits of precision in significand

2

(excluding the implicit most significant bit due to (de)normalization)

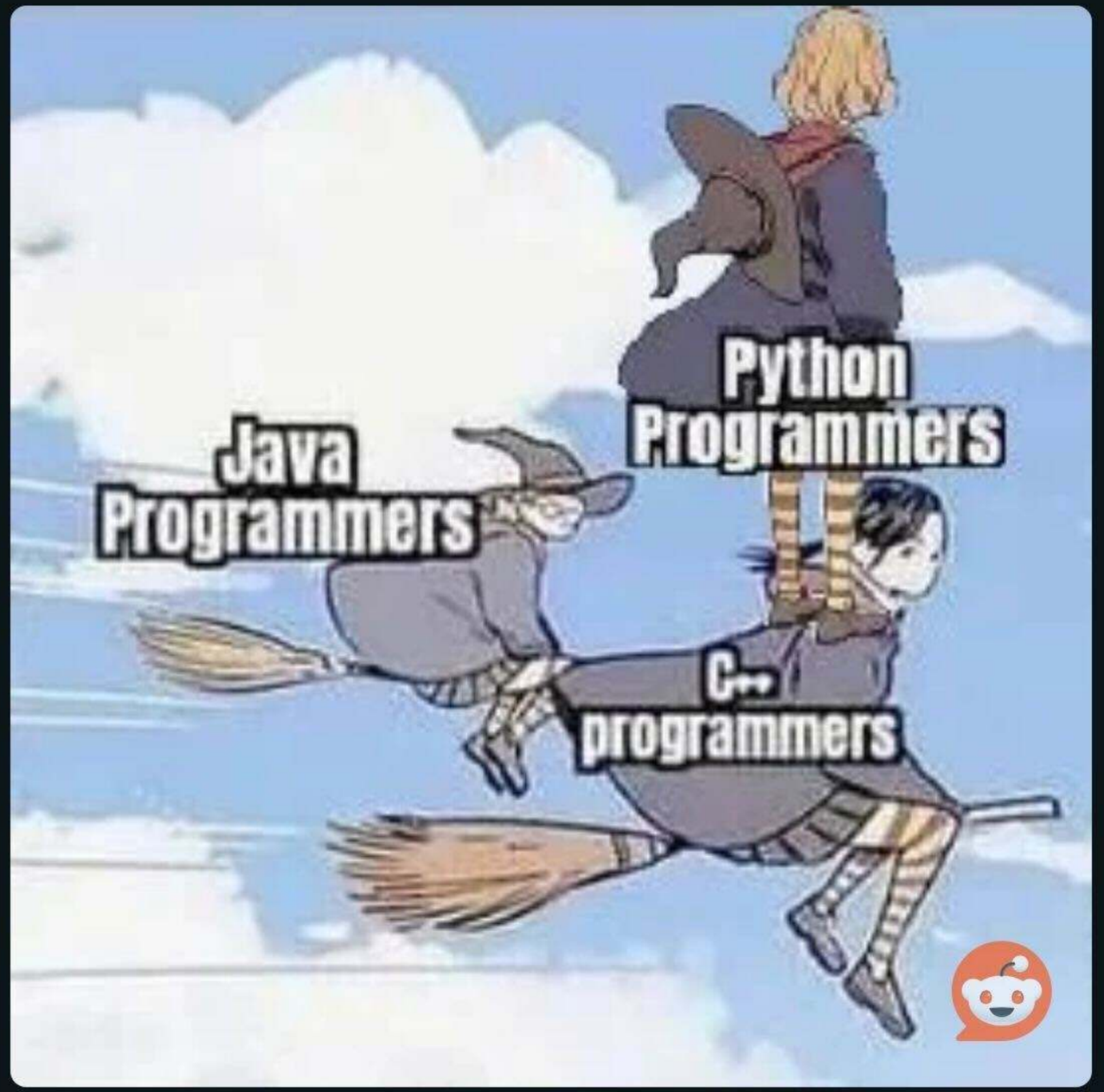
Project Valhalla: Half-floats (float16)



21 Września 2024

Native Calls

From **ProgrammerHumor** community on **Reddit**




JVM
JVM WHEN YOU REALIZE
IT'S WRITTEN IN C++
imgflip.com

Memory

Setting available Memory/RAM for a DL4J application

Memory Management for ND4J/DL4J: How does it work?

ND4J uses off-heap memory to store NDArrays, to provide better performance while working with NDArrays from native code such as BLAS and CUDA libraries.

"Off-heap" means that the memory is allocated outside of the JVM (Java Virtual Machine) and hence isn't managed by the JVM's garbage collection (GC). On the Java/ JVM side, we only hold pointers to the off-heap memory, which can be passed to the underlying C++ code via JNI for use in ND4J operations.

Start Simple - JNI and Calling C Libraries

Memory

Setting available Memory/RAM for a DL4J application

Memory Management for ND4J/DL4J: How does it work?

ND4J uses off-heap memory to store NDArrays, to provide better performance while working with NDArrays from native code such as BLAS and CUDA libraries.

"Off-heap" means that the memory is allocated outside of the JVM (Java Virtual Machine) and hence isn't managed by the JVM's garbage collection (GC). On the Java/JVM side, we only hold pointers to the off-heap memory, which can be passed to the underlying C++ code via JNI for use in ND4J operations.

Start Simple - JNI and Calling C Libraries

Quick Intro to JNI

JNI stands for Java Native Interface, a framework that allows Java code running in the Java Virtual Machine (JVM) to interact with native applications and libraries written in other programming languages like C or C++.

JNI is (was?) typically used when Java applications need to perform tasks that cannot be efficiently accomplished with pure Java, such as using system-specific libraries or interacting with low-level hardware.

Java 1.1, 1997

Declare the native method in Java

In your Java code, declare a method as *native* without providing an implementation...

```
public class MyNativeClass {
    public native void myNativeMethod();
}
```

...and compile the code, which generates a .class file.

Using the **javah** tool (available until JDK 8), generate a C/C++ header file. This file contains the native method signatures in C/C++...

```
JNIEXPORT void JNICALL Java_MyNativeClass_myNativeMethod(JNIEnv *, jobject);
```

...which you need to manually write the native implementation in C/C++

Compile the C/C++ code into a native library (e.g., .dll on Windows, .so on Linux, .dylib on macOS).

```
static {
    System.loadLibrary("myNativeLibrary");
}
```

Java will load this library at runtime using System.loadLibrary().

Latest Release

CUDA Toolkit 12.6.1 (August 2024), Versioned Online Documentation

Archived Releases

CUDA Toolkit 12.6.0 (August 2024), Versioned Online Documentation
CUDA Toolkit 12.5.1 (July 2024), Versioned Online Documentation
CUDA Toolkit 12.5.0 (May 2024), Versioned Online Documentation
CUDA Toolkit 12.4.1 (April 2024), Versioned Online Documentation
CUDA Toolkit 12.4.0 (March 2024), Versioned Online Documentation
CUDA Toolkit 12.3.2 (January 2024), Versioned Online Documentation
CUDA Toolkit 12.3.1 (November 2023), Versioned Online Documentation
CUDA Toolkit 12.3.0 (October 2023), Versioned Online Documentation
CUDA Toolkit 12.2.2 (August 2023), Versioned Online Documentation
CUDA Toolkit 12.2.1 (July 2023), Versioned Online Documentation
CUDA Toolkit 12.2.0 (June 2023), Versioned Online Documentation
CUDA Toolkit 12.1.1 (April 2023), Versioned Online Documentation
CUDA Toolkit 12.1.0 (February 2023), Versioned Online Documentation
CUDA Toolkit 12.0.1 (January 2023), Versioned Online Documentation
CUDA Toolkit 12.0.0 (December 2022), Versioned Online Documentation
CUDA Toolkit 11.8.0 (October 2022), Versioned Online Documentation
CUDA Toolkit 11.7.1 (August 2022), Versioned Online Documentation
CUDA Toolkit 11.7.0 (May 2022), Versioned Online Documentation
CUDA Toolkit 11.6.2 (March 2022), Versioned Online Documentation

Quick Intro to JNI

jextract - introduced in Project Panama - automates the process of generating Java bindings from C/C++ header files.

It eliminates the need to manually write JNI code.

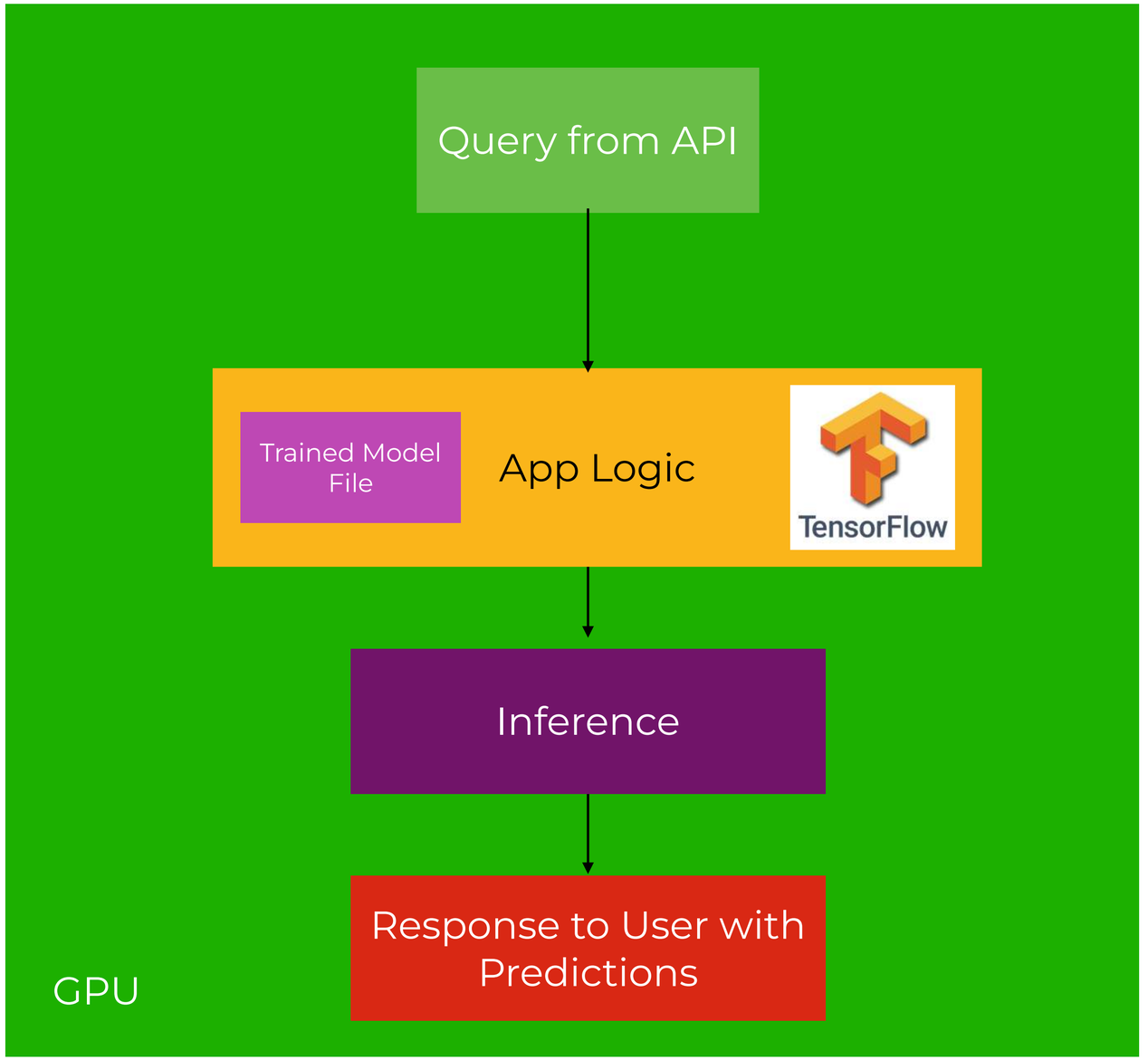
Changes to the native library are handled simply by re-running jextract.

The Java code generated by jextract maps directly to the native functions and data structures, allowing seamless interaction between Java and native code.

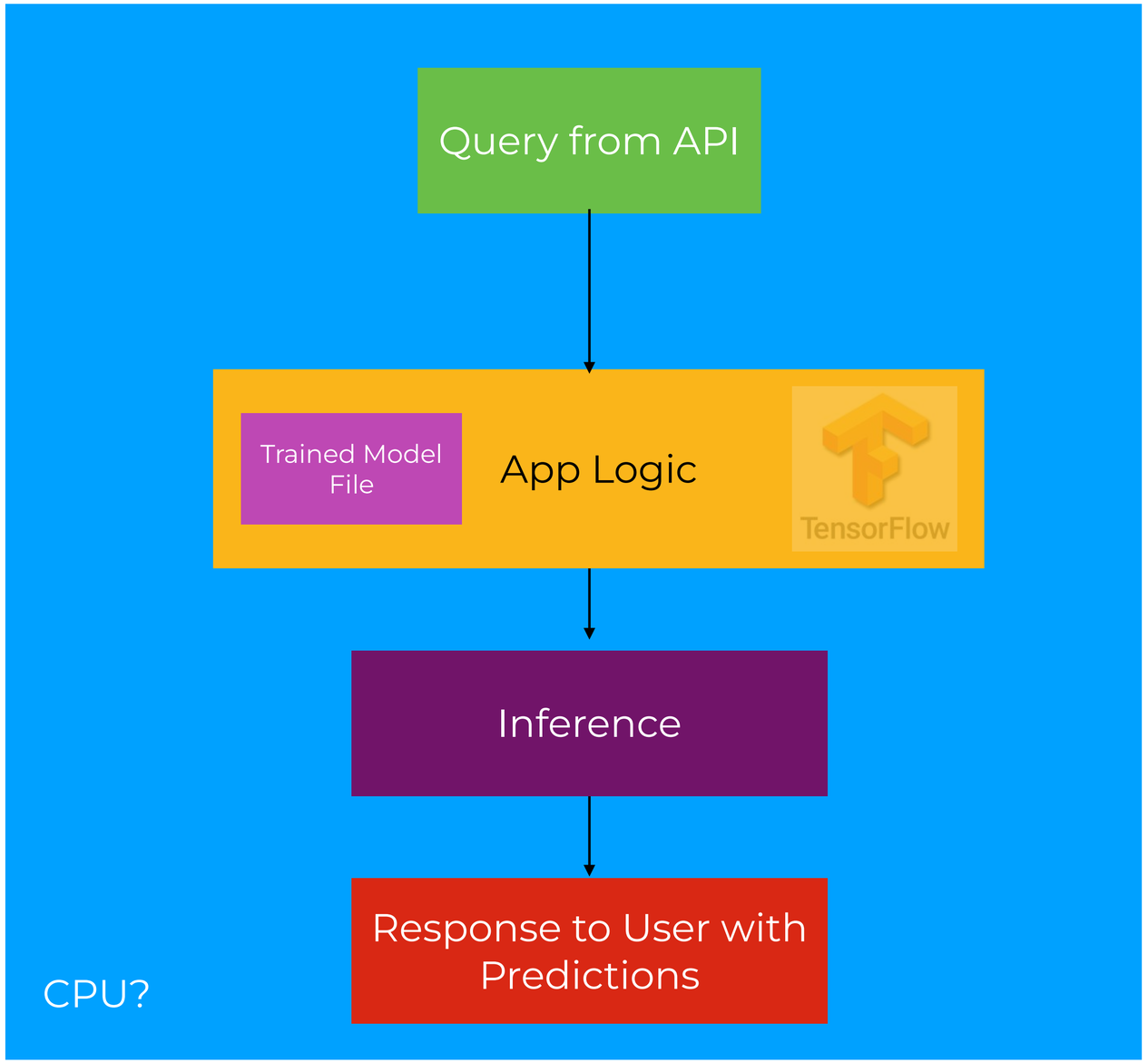
Jextract - Java 17 (2021)

Project Panama

1. Foreign Function & Memory API (FFM API)
2. Linker API
3. Foreign Data Layout API
4. Interconnect with Foreign Languages
5. Improved Performance for Native Calls
6. Vector API



Model Inference run on GPU



Can it be efficiently run on CPU without frameworks?

Large Language Model

Introducing Meta Llama 3: The most capable openly available LLM to date

April 18, 2024

llama2.c

Have you ever wanted to inference a baby Llama 2 model in pure C? No? Well, now you can!

Train the Llama 2 LLM architecture in PyTorch then inference it with one simple 700-line C file (run.c). You might think that you need many billion parameter LLMs to do anything useful, but in fact very small LLMs can have surprisingly strong performance if you make the domain narrow enough (ref: TinyStories paper). This repo is a "fullstack" train + inference solution for Llama 2 LLM, with focus on minimalism and simplicity.

llama.cpp

license MIT | Server passing | conan b3542

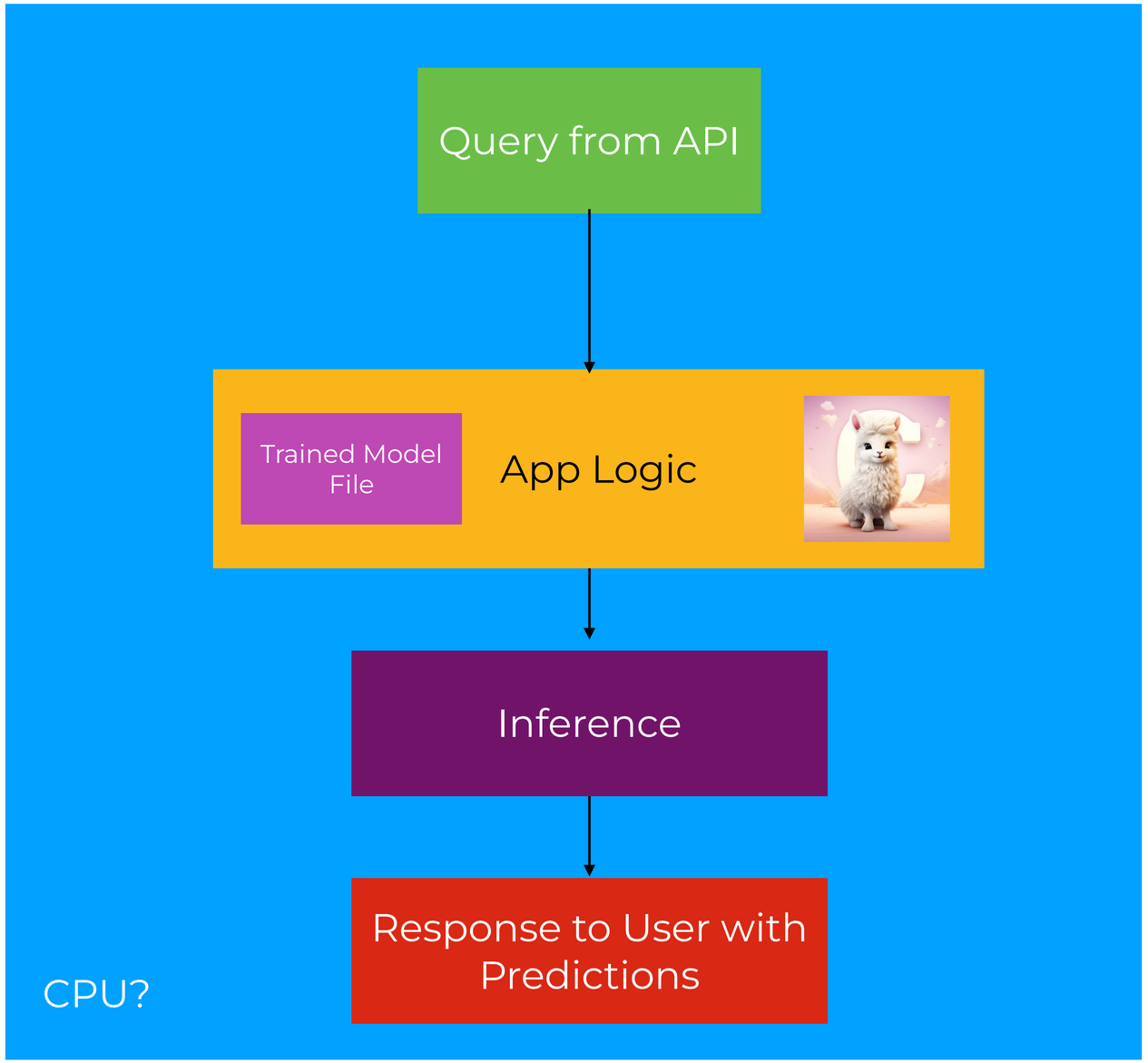
Roadmap / Project status / Manifesto / ggml

Inference of Meta's LLaMA model (and others) in pure C/C++

llama.cpp is a 700-line, C++-based inference implementation optimized for CPU inference.

It bypasses the traditional GPU-centric frameworks like PyTorch or TensorFlow.

Panama Vector API and Vector Databases



Can it be efficiently run on CPU without frameworks?

README MIT license

Java 11+ llama.cpp #b3534

Java Bindings for llama.cpp

Inference of Meta's LLaMA model (and others) in pure C/C++.

You are welcome to contribute

1. Quick Start
   1.1 No Setup required
   1.2 Setup required
2. Documentation
   2.1 Example
   2.2 Inference
   2.3 Infilling
3. Android

> **Note**
>
> Now with support for Llama 3, Phi-3, and flash attention

https://github.com/kherud/java-llama.cpp - Java Bindings

Llama3.java

Practical Llama 3 and 3.1 inference implemented in a single Java file.

This project is the successor of llama2.java based on llama2.c by Andrej Karpathy and his excellent educational videos.

Besides the educational value, this project will be used to test and tune compiler optimizations and features on the JVM, particularly for the Graal compiler.

Features

- Single file, no dependencies
- GGUF format parser
- Llama 3 tokenizer based on minbpe
- Llama 3 inference with Grouped-Query Attention
- Support Llama 3.1 (ad-hoc RoPE scaling)
- Support for Q8_0 and Q4_0 quantizations
- Fast matrix-vector multiplication routines for quantized tensors using Java's Vector API
- Simple CLI with `--chat` and `--instruct` modes.

Here's the interactive `--chat` mode in action:

main llama3.java / Llama3.java

mukel Fix missing <|eom_id|> token in Llama 3 vocabulary.

Code | Blame | Executable File · 2103 lines (1851 loc) · 91 KB | Code 55% faster with GitHub Copilot

```
1   ///usr/bin/env jbang "$0" "$@" ; exit $?
2   //JAVA 21+
3   //PREVIEW
4   //COMPILE_OPTIONS --add-modules=jdk.incubator.vector
5   //RUNTIME_OPTIONS --add-modules=jdk.incubator.vector
6
7   // Practical Llama 3 (and 3.1) inference in a single Java file
8   // Author: Alfonso² Peterssen
9   // Based on Andrej Karpathy's llama2.c and minbpe projects
10  //
11  // Supports llama.cpp's GGUF format, restricted to Q4_0 and Q8_0 quantized models
12  // Multi-threaded matrix vector multiplication routines implemented using Java's Vector API
13  // Simple CLI with --chat and --instruct mode
14  //
15  // To run just:
16  // jbang Llama3.java --help
17  //
18  // Enjoy!
19  package com.llama4j;
```

Llama3.java

README Apache-2.0 license

🦙 Jlama: A modern LLM inference engine for Java

maven-central v0.4.1 License Apache 2.0 Discord 5 online

🚀 Features

Model Support:

- Gemma Models
- Llama & Llama2 & Llama3 Models
- Mistral & Mixtral Models
- GPT-2 Models
- BERT Models
- BPE Tokenizers
- WordPiece Tokenizers

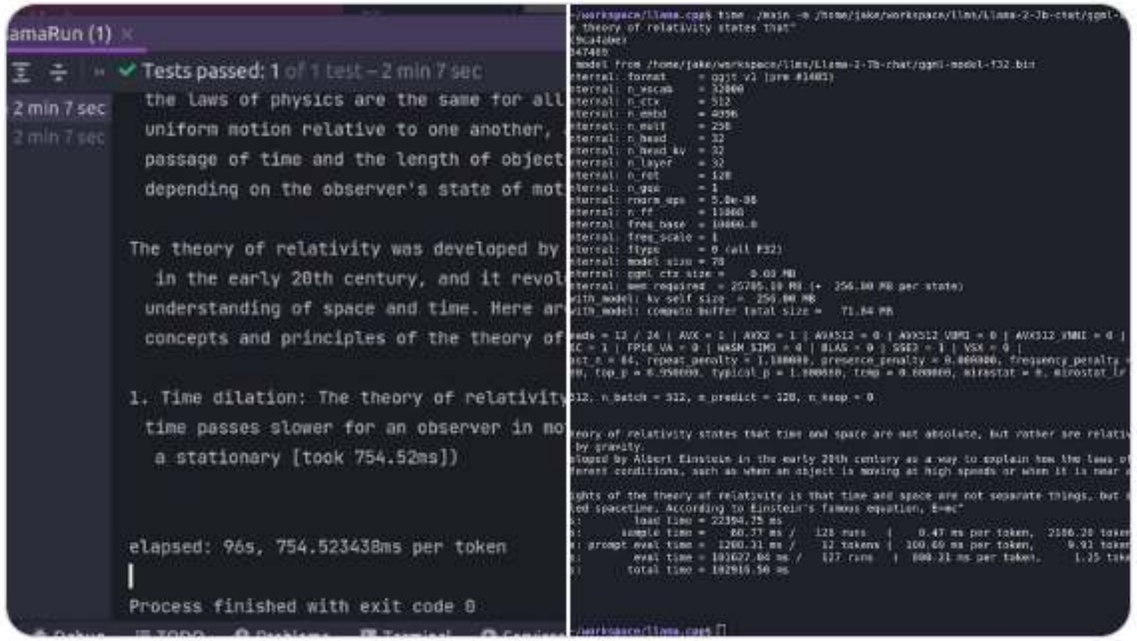
```java
public void sample() throws IOException {
   String model = "tjake/TinyLlama-1.1B-Chat-v1.0-Jlama-Q4";
   String workingDirectory = "./models";

   String prompt = "What is the best season to plant avocados?";

   // Downloads the model or just returns the local path if it's already downloaded
   File localModelPath = SafeTensorSupport.maybeDownloadModel(workingDirectory, model);

   // Loads the quantized model and specified use of quantized memory
   AbstractModel m = ModelSupport.loadModel(localModelPath, DType.F32, DType.I8);

   PromptContext ctx;
   // Checks if the model supports chat prompting and adds prompt in the expected format for this
model
   if (m.promptSupport().isPresent()) {
       ctx = m.promptSupport()
              .get()
              .builder()
              .addSystemMessage("You are a helpful chatbot who writes short responses.")
              .addUserMessage(prompt)
              .build();
   } else {
       ctx = PromptContext.of(prompt);
   }

   System.out.println("Prompt: " + ctx.getPrompt() + "\n");
   // Generates a response to the prompt and prints it
   // The api allows for streaming or non-streaming responses
   // The response is generated with a temperature of 0.7 and a max token length of 256
   Generator.Response r = m.generate(UUID.randomUUID(), ctx, 0.0f, 256, (s, f) -> {});
   System.out.println(r.responseText);
}
```

JLama

Jake Luciani
@tjake

In pretty hilarious/surprising news: Jlama (Java) is faster than llama.cpp for the same threads in ms per token. This is for the F32 llama 7b model. I'm going to start adding quantization now.

Przetłumacz wpis



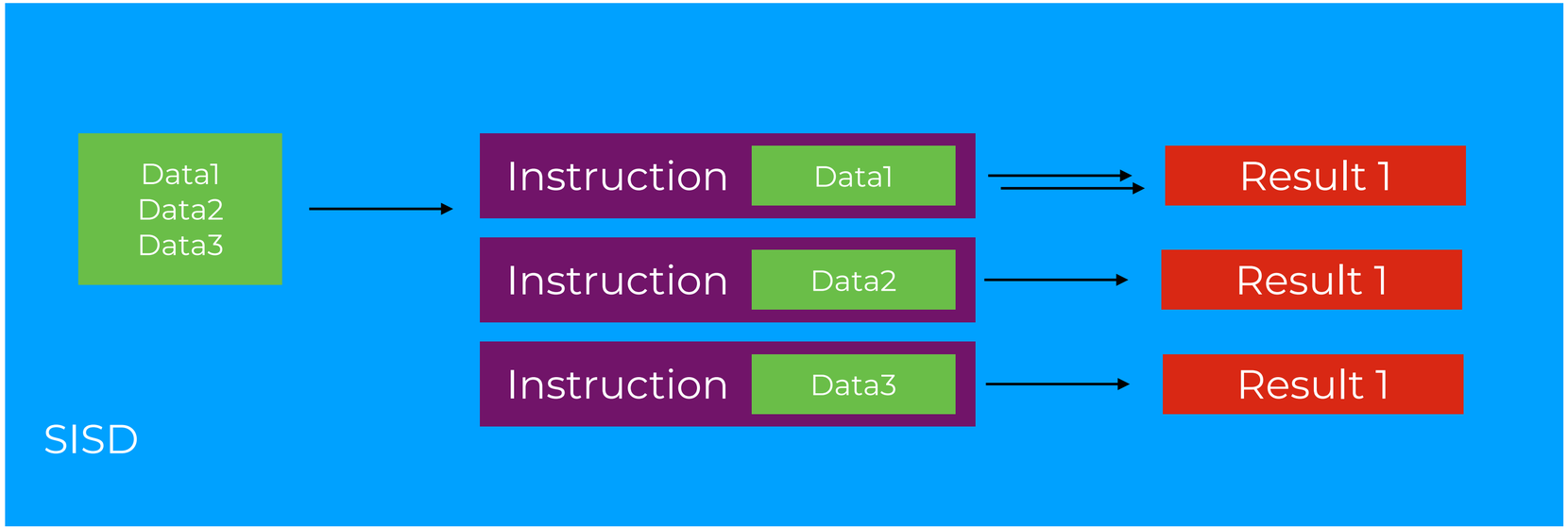
Ostatnia zmiana: 7:42 PM · 13 sie 2023 · **18,1 tys.** Wyświetlenia

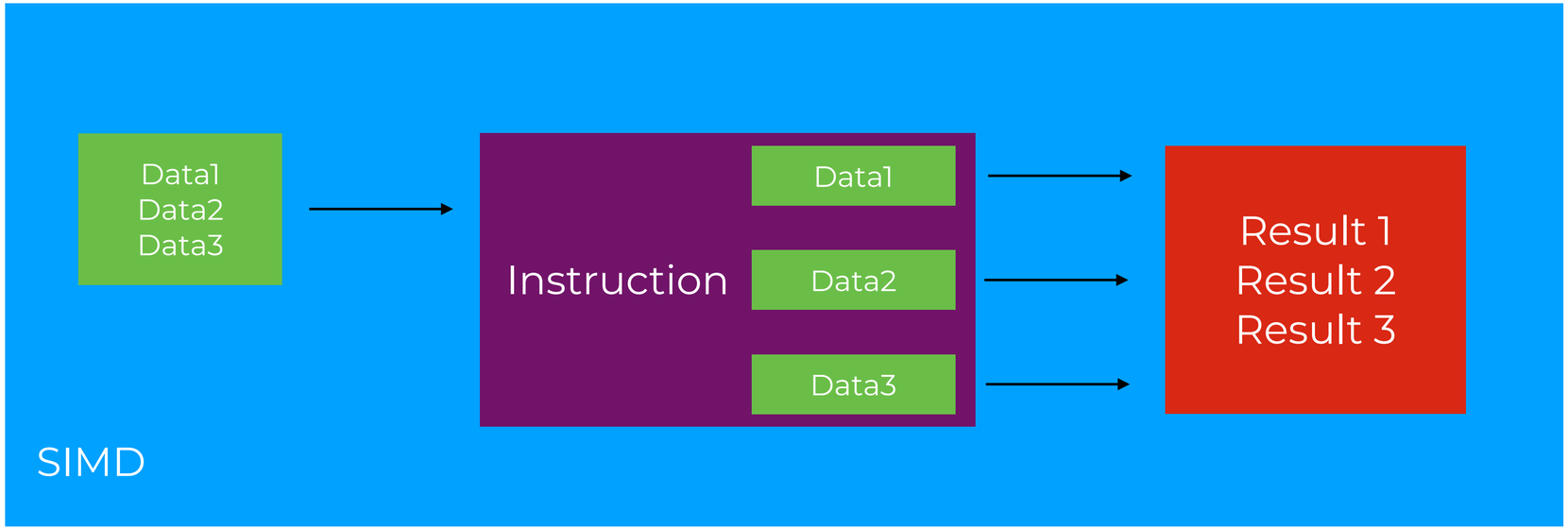
Panama Vector API and Vector Databases

Project Valhalla & Panama Together: SIMD

SIMD (Single Instruction, Multiple Data) is a parallel processing model that allows processors to perform the same arithmetic or logical operation on multiple data points simultaneously.

This is a technique used in modern processors, aimed at speeding up the processing of large amounts of data, such as in graphics processing, multimedia, scientific calculations, or machine learning models.

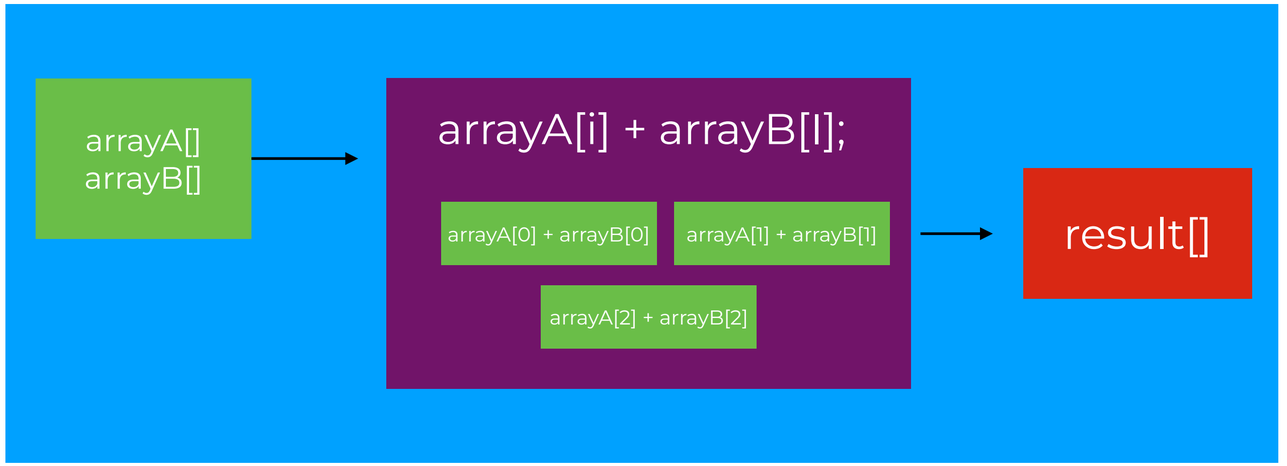




Panama Vector API and Vector Databases

```java
public class VectorizationExample {
    public static void main(String[] args) {
        int[] arrayA = new int[1000];
        int[] arrayB = new int[1000];
        int[] result = new int[1000];

        // Initialize arrays
        for (int i = 0; i < arrayA.length; i++) {
            arrayA[i] = i;
            arrayB[i] = i * 2;
        }

        // Perform element-wise addition (this can be autovectorized)
        for (int i = 0; i < arrayA.length; i++) {
            result[i] = arrayA[i] + arrayB[i];
        }

        // Output some of the results
        for (int i = 0; i < 10; i++) {
            System.out.println("Result[" + i + "] = " + result[i]);
        }
    }
}
```

JIT



Autovectorization

OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki · IRC

Bylaws · Census
Legal

Workshop

JEP Process

Source code
Mercurial
GitHub

Tools
Git
jtreg harness

Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web

Projects
(overview, archive)
Amber
Babylon
CRaC
Caciocavallo
Closures

JEP 469: Vector API (Eighth Incubator)

Owner	Paul Sandoz
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	23
Component	core-libs
Discussion	panama dash dev at openjdk dot org
Effort	XS
Duration	XS
Relates to	JEP 460: Vector API (Seventh Incubator)
Reviewed by	Vladimir Ivanov
Endorsed by	John Rose
Created	2024/02/27 20:50
Updated	2024/07/15 15:59
Issue	8326878

Summary

Introduce an API to express vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures, thus achieving performance superior to equivalent scalar computations.

History

We first proposed the Vector API in JEP 338, integrated into JDK 16 as an incubating API. We proposed further rounds of incubation in JEP 414 (integrated into JDK 17), JEP 417 (JDK 18), JEP 426 (JDK 19), JEP 438 (JDK 20), JEP 448 (JDK 21), and JEP 460 (JDK 22).

We propose here to re-incubate the API in JDK 23 with no API changes and no substantial implementation changes relative to JDK 22.

The Vector API will incubate until necessary features of Project Valhalla become available as preview features. At that time, we will adapt the Vector API and its implementation to use them, and will promote the Vector API from incubation to preview.

Panama Vector API and Vector Databases

Panama Vector API and Vector Databases

JVector - The most advanced embedded vector search engine

Share this:

JVector is a pure Java, zero dependency, embedded vector search engine, used by DataStax Astra DB.

JVector uses state of the art graph algorithms inspired by DiskANN and related research that offer high recall and low latency. It uses the Panama SIMD API to accelerate index build and queries. It is memory efficient, compresses vectors using product quantization so they can stay in memory during searches.

JVector is concurrent, its index can scale linearly to 32 threads.

jvector Public

Watch 29

main | 29 Branches | 36 Tags | Go to file t | Add file | Code

jbellis 09/24 version of colbert-10M ✓ a95e640 · 2 days ago 669 Commits

.github/workflows	Enable JDK 22 CI	6 months ago
.mvn	Add mvn wrapper	last year
jvector-base	fix getIdUpperBound javadoc	5 days ago
jvector-examples	09/24 version of colbert-10M	2 days ago
jvector-multirelease	Use on-heap MemorySegments for native vectors/sequen...	6 months ago
jvector-native	Remove check for VBMI on CPU. With Fused ADC using s...	last month
jvector-tests	add support for non-sequential remapped ordinals (#349)	2 months ago
jvector-twenty	Release 3.0.0-beta.13	3 months ago
siftsmall	SiftSmall example	last year
.gitignore	Use precomputed PQ in Bench. Add gitignore for cached ...	6 months ago
LICENSE.txt	import hnsw + pq	last year
NOTICE.txt	import hnsw + pq	last year
README.md	Release 3.0.0-beta.13	3 months ago
UPGRADING.md	Release 3.0.0-beta.13	3 months ago

Panama Vector API and Vector Databases

Why interoperability with C at all?

Special Hardware!

March 27, 2006

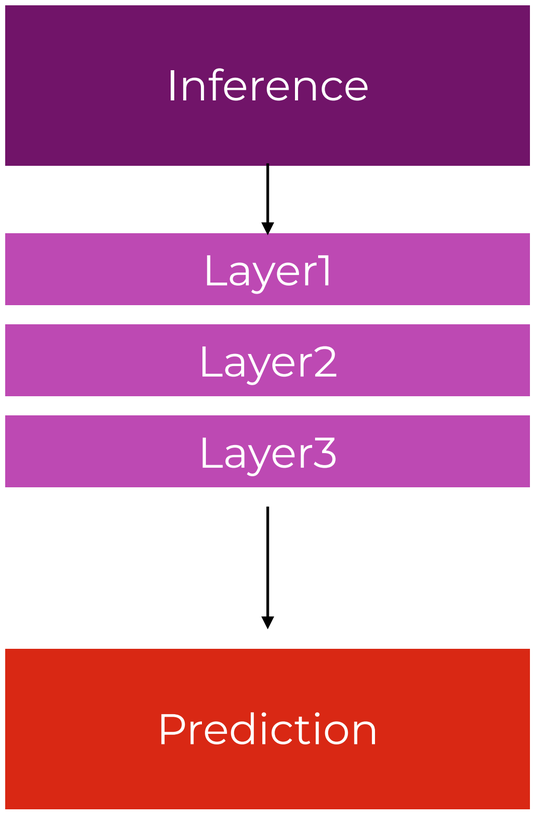
Azul Systems® First to Deliver 48-Way Multicore Chip, Redefining Standard in Enterprise Computing

Mountain View, CA – March 27, 2006 – Azul Systems, Inc., the pioneer of the industry's first network attached processing solution designed to deliver compute and memory resources as a shared network service for transaction-intensive applications, today announced their next generation Vega™ 2 processor, the world's first and only single-chip 64-bit processor with 48 cache-coherent processor cores for use in the company's industry-leading, award winning compute appliances.

Designed by Azul and fabricated by Taiwan Semiconductor Manufacturing Company (TSMC,) the Vega 2 processor utilizes an advanced 90nm process, a 9-layer copper/low-k interconnect, and multi-threshold transistors for optimal performance-power efficiency. This revolutionary processor is poised to spur more energy-efficient, higher density, and larger capacity compute appliances that meet critical data center needs such as higher asset utilization, improved security, and lower-cost per transaction, all while reducing real-estate, electricity and associated management costs.

The Vega 2 processor integrates 48 cores and consists of 812 million transistors to enable future generations of Azul Compute Appliances to scale up to 768-way symmetric multiprocessing (SMP) systems with up to 768 GBytes of memory. By comparison, the world's largest and most power efficient SMP system built today is the Azul Compute Appliance 3840, which is based on the current generation Azul Vega 1 processor (a 24-way multicore chip) and provides 384 processor cores and 256 GBytes of memory in an 11U rack-mountable enclosure and consumes only 2700 watts of electricity.

"Multicore chip architectures are quickly taking their place as the industry de facto standard for enterprise computing," said Vernon Turner, vice president and general manager, enterprise computing, IDC. "The Azul Vega chip is revolutionary in its sheer multicore scale and power and density characteristics. This design criterion will be critical as more and more next generation enterprise systems come to market and deliver compute resources as a shared network service across all types of heterogeneous workloads."

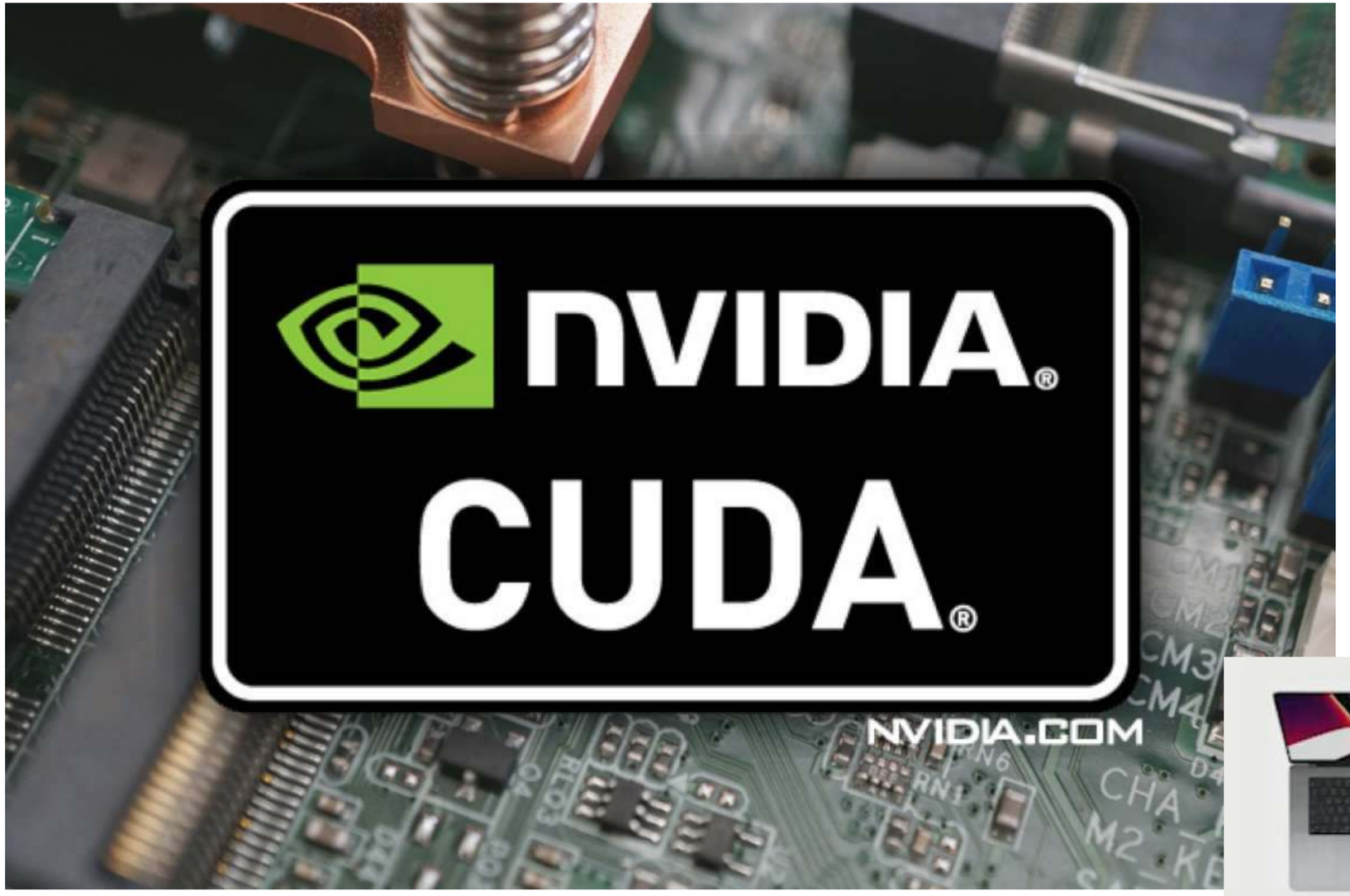


Next Token of Text
Classification of Image

Computational Complexity

Each layer of requires performing a vast number of mathematical operations, such as matrix multiplications. The more layers a model has, the greater the number of such operations.

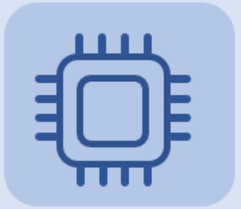
Model Inference



This Laptop (M1 Mac Pro)
512 shader/execution units
Max (FP32) 10.4 TFLOPs

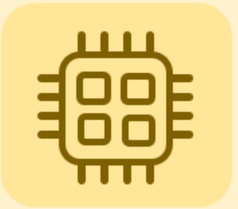
From first JavaOne 1996 until JavaOne 2002
This GPU alone would be #1 on top 500 list


, MY PRECIOUS GRAPHICS CARD, WE SHALL PLAY ALL THE GAMES IN 4K RESOLUTION AT MAX SETTINGS! WITCHER 3, BATTLEFIELD 1, SIEGE, DOOM, CRYSIS...
OH BOY, THAT SOUNDS LIKE A LOT OF WORK.
WORK, YOU CALL THAT WORK? HOW WOULD YOU LIKE TO BE IN THE MINES?!
WORKING NON-STOP IN CRAMPED HOT CONDITIONS MINING DIGITAL CURRENCIES UNTIL YOU CAN'T WORK ANYMORE!
I...I'D... LIKE TO PLAY GAMES IN 4K NOW...
YEAH, I THOUGHT YOU MIGHT.
CORSAIR
PLAY HARD, OR WORK HARDER | #CORSAIRCOMICS



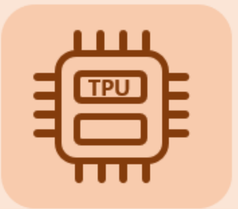
CPU

- Small models
- Small datasets
- Useful for design space exploration



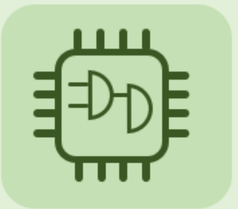
GPU

- Medium-to-large models, datasets
- Image, video processing
- Application on CUDA or OpenCL



TPU

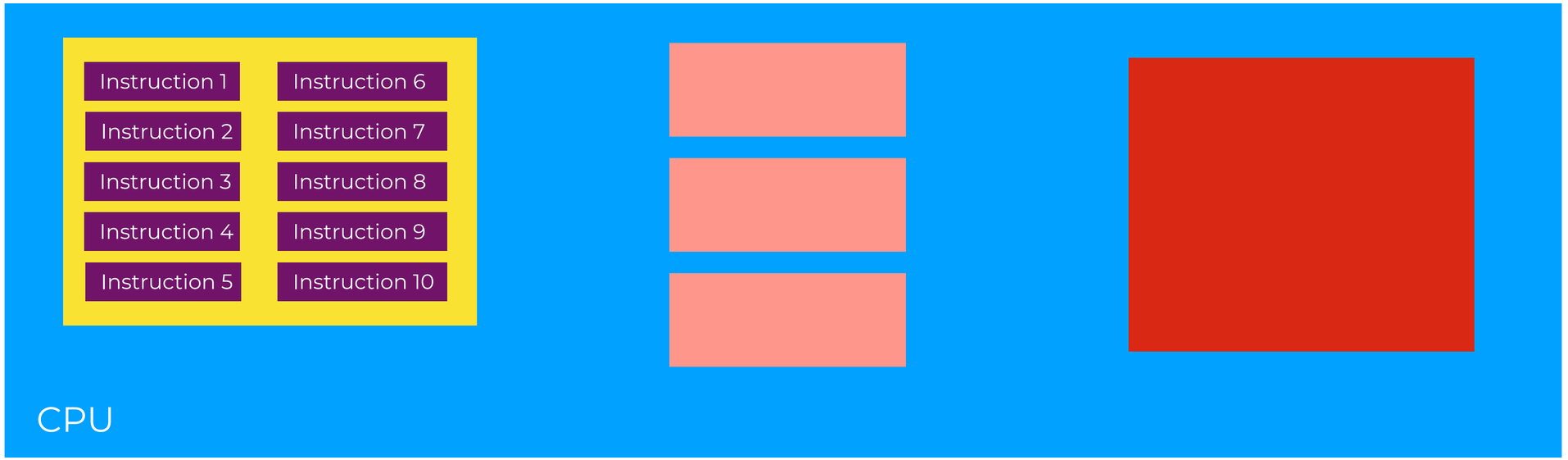
- Matrix computations
- Dense vector processing
- No custom TensorFlow operations



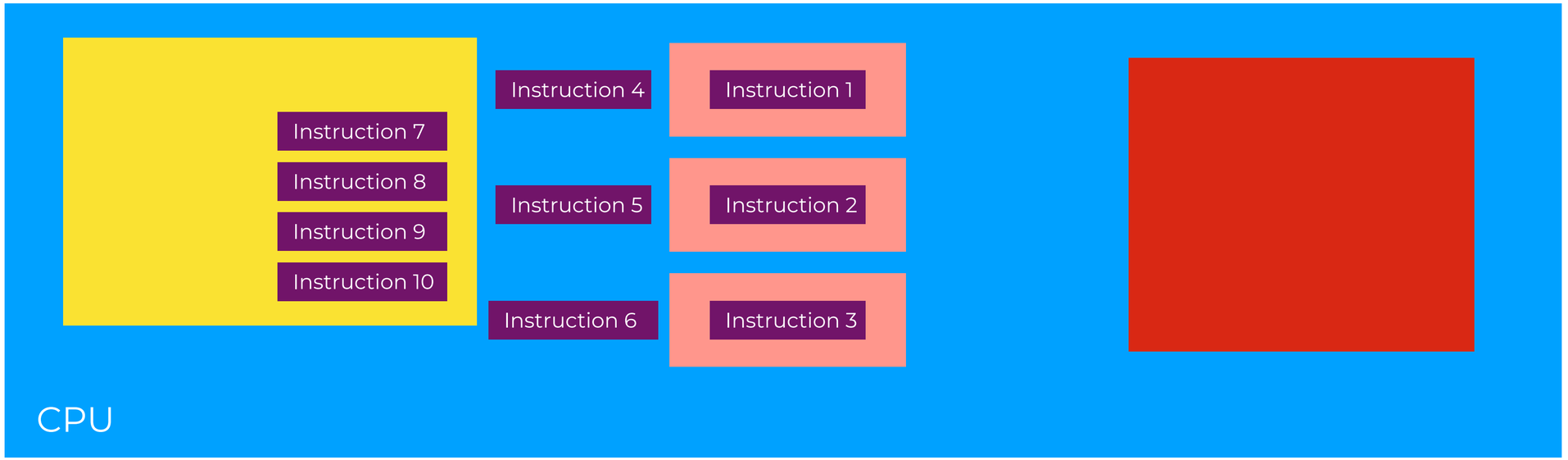
FPGA

- Large datasets, models
- Compute intensive applications
- High performance, high perf./cost ratio

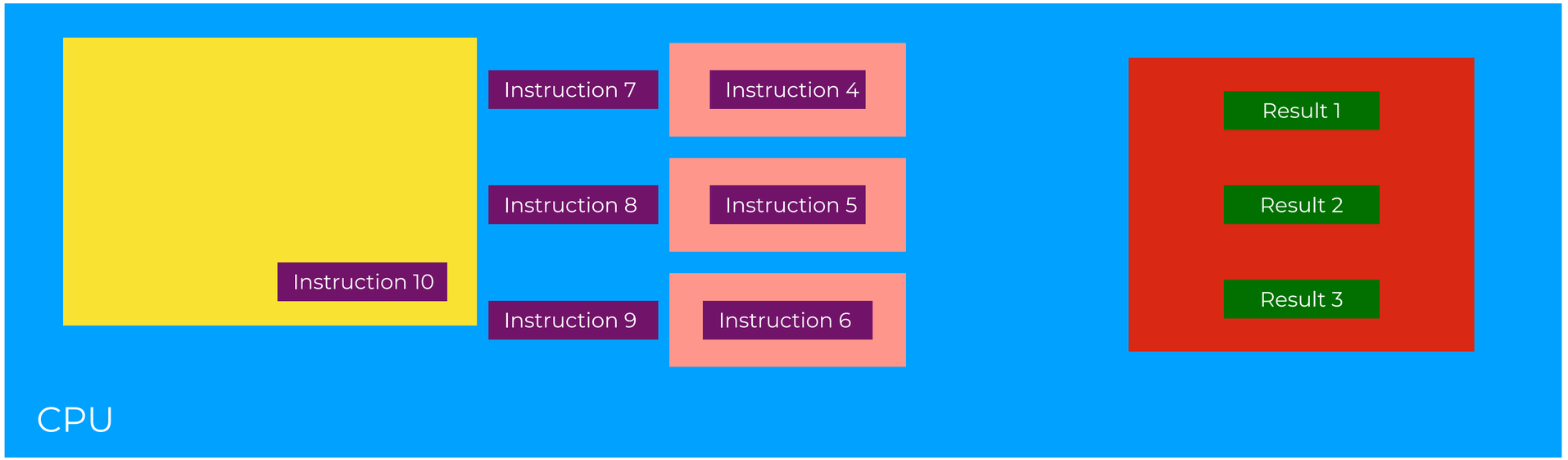
OVERLY SIMPLIFIED EXAMPLE ALERT!



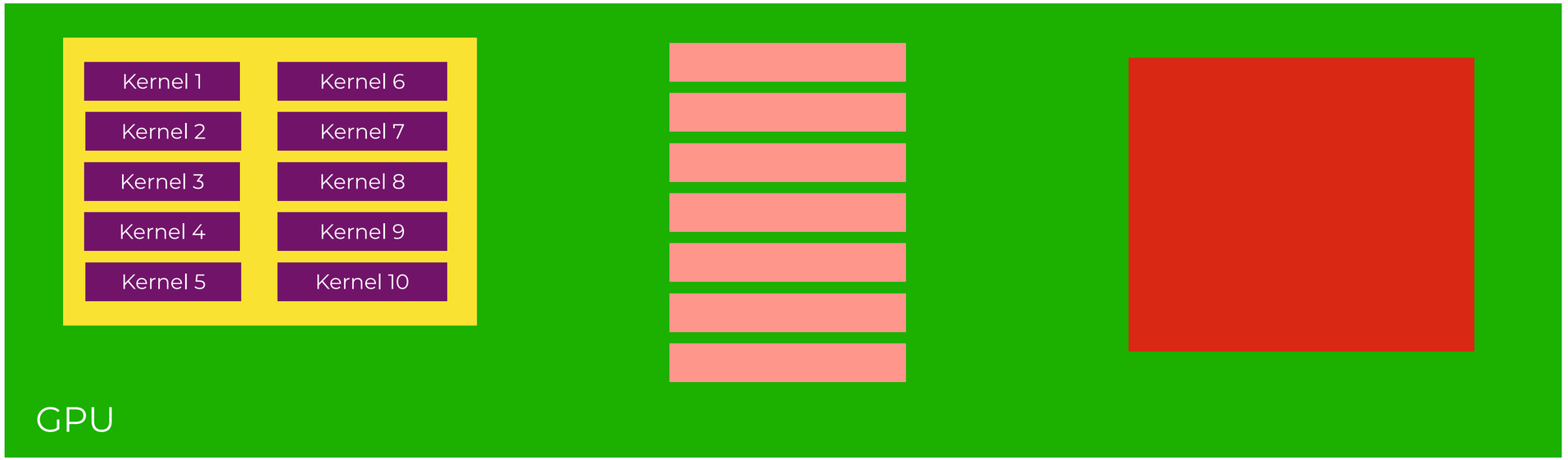
Model Inference run on GPU



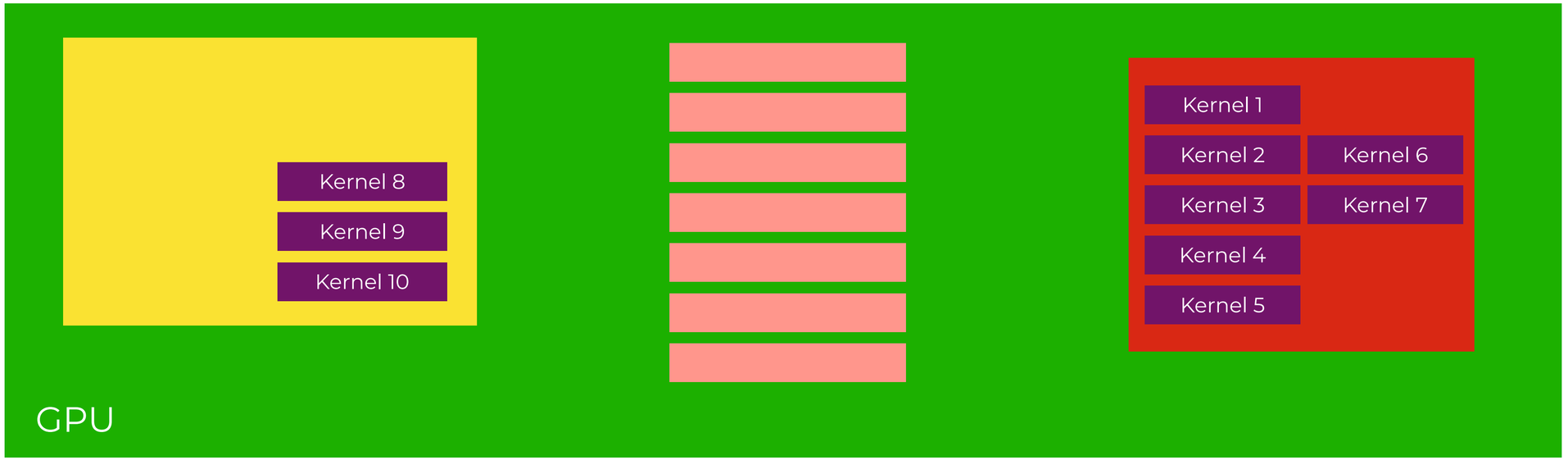
Model Inference run on GPU



Model Inference run on GPU



Model Inference run on GPU



Model Inference run on GPU

Parallelism: The programmer must divide tasks into very small parts that can be processed simultaneously, which requires advanced thinking about task division and synchronization.

Memory management: The GPU has its own separate memory, so data must be appropriately transferred between the CPU and GPU, which requires additional resource management.

API and low-level control: Programming on the GPU often requires using low-level APIs (such as CUDA or OpenCL), which are more complex and less user-friendly than higher-level programming languages.

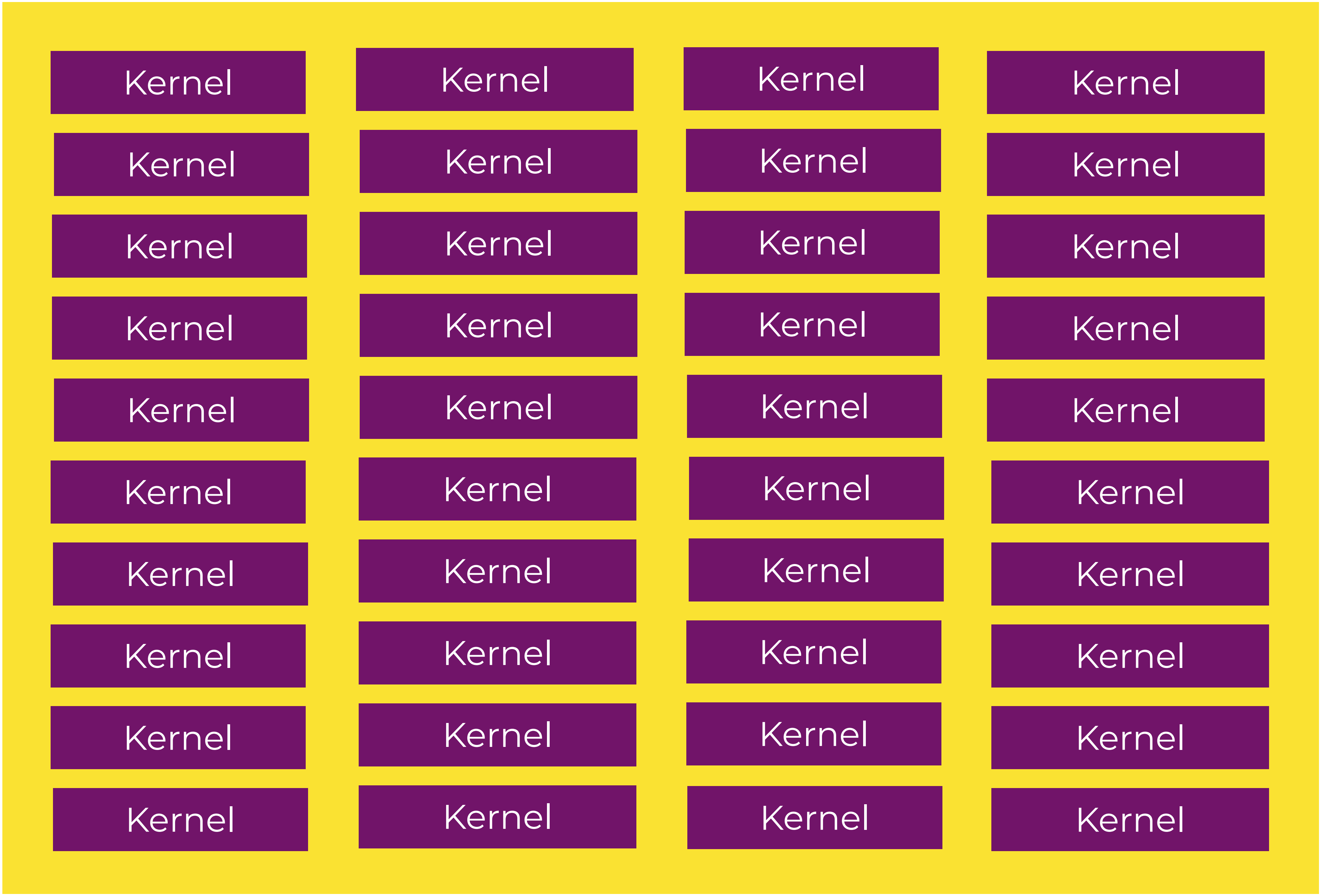
The main difficulties

Kernel

Kernel is a piece of code that is executed by hundreds or thousands of threads simultaneously.

Each thread runs the same code but operates on different data, enabling the processing of large amounts of data in a highly parallel manner.

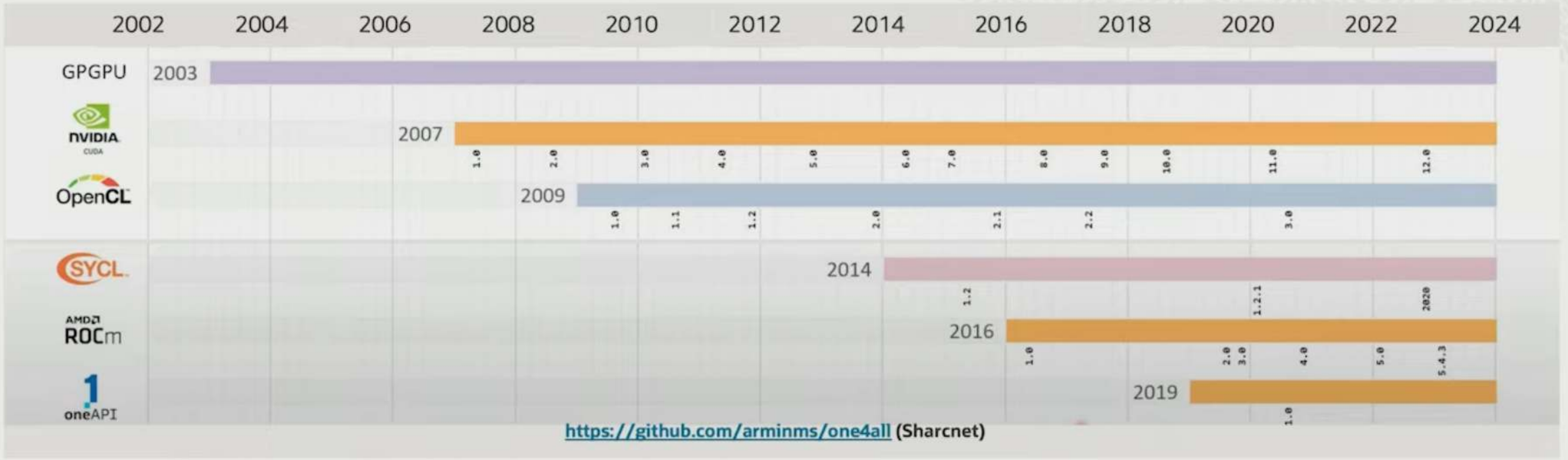
What is Kernel and NDGrid?



NDGrid (n-dimensional grid) refers to a multi-dimensional grid of threads used for parallel computation on a GPU.

The GPU runs threads in grids with multiple dimensions, allowing kernel processing in different dimensions (e.g., 1D, 2D, 3D) simultaneously.

GPGPU Timeline



Fairly consistent Kernel + NDRange/Grid based programming models across vendors..

Hardware Matters

The first steps in GPU programming in Java began with projects that exposed CUDA and OpenCL APIs through **JNI**.

Projects such as JCuda, and JOpenCL allowes the use of native GPU libraries from within Java, but require the programmer to manage GPU tasks in a rather low-level manner.

Calculating the square of each element in an array

```
cudaMalloc(deviceInput, n * Sizeof.FLOAT);
cudaMalloc(deviceOutput, n * Sizeof.FLOAT);

// Copyin data from CPU to GPU (CPU -> GPU)
cudaMemcpy(deviceInput, Pointer.to(hostInput), n * Sizeof.FLOAT, cudaMemcpyHostToDevice);

// Defining CUDA kernel as String
String cudaKernel =
    "extern \"C\"" + "\n" +
    "__global__ void squareKernel(float *input, float *output) {" + "\n" +
    "    int idx = threadIdx.x;" + "\n" +
    "    output[idx] = input[idx] * input[idx];" + "\n" +
    "}";

// Kernel Compiling
CUmodule module = new CUmodule();
CUfunction function = new CUfunction();
JCudaDriver.cuModuleLoadData(module, cudaKernel);
JCudaDriver.cuModuleGetFunction(function, module, "squareKernel");
```

Kernels are embedded as strings in the code, and programmers had to manually manage data movement and task execution.

Aparapi and Rootbeer are projects that allowed the expression of kernels **directly in Java code**, eliminating the need to manually write CUDA or OpenCL code.

They don’t provide a direct API for defining **ndgrid** in the way CUDA or OpenCL does

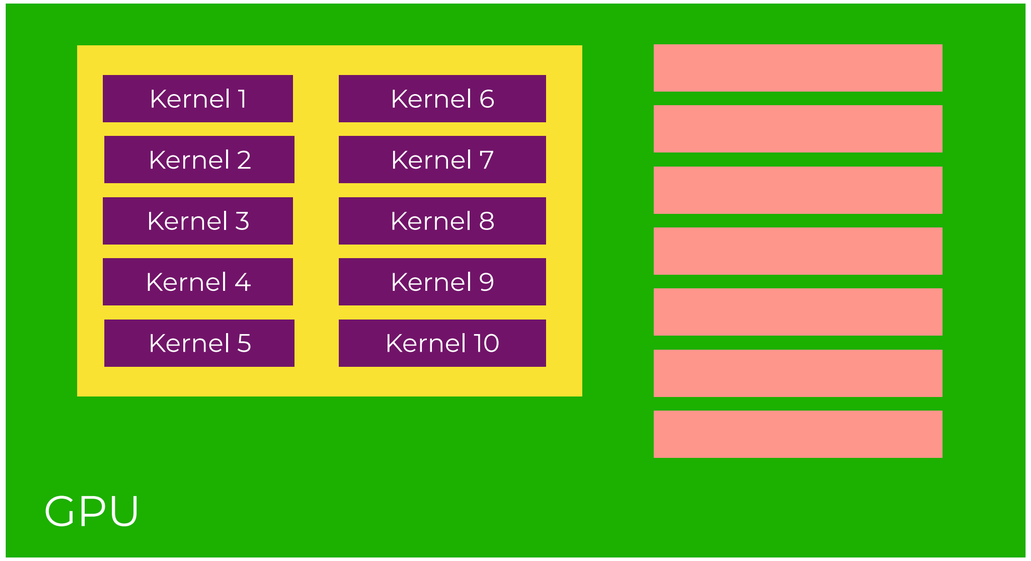
```java
import com.aparapi.Kernel;
import com.aparapi.Range;

public class AparapiExample {

    public static void main(String[] args) {
        // Array size
        final int size = 10;
        final float[] input = new float[size];
        final float[] output = new float[size];

        // Fill the input array with sample data
        for (int i = 0; i < size; i++) {
            input[i] = i;
        }

        // Define the kernel in Java, which will be processed on the GPU
        Kernel kernel = new Kernel() {
            @Override
            public void run() {
                int id = getGlobalId();
                output[id] = input[id] * input[id];
            }
        };

        // Define the range (number of threads)
        Range range = Range.create(size);

        kernel.execute(range);

        System.out.println("Results:");
        for (int i = 0; i < size; i++) {
            System.out.println(input[i] + " squared = " + output[i]);
        }

        // Clean up the kernel resources
        kernel.dispose();
    }
}
```

Kernels expressed in Java

1. Lack of support for proper data types
2. Lack of support for diverse hardware
3. In-efficient transpiled code…
4. …or manual hacks

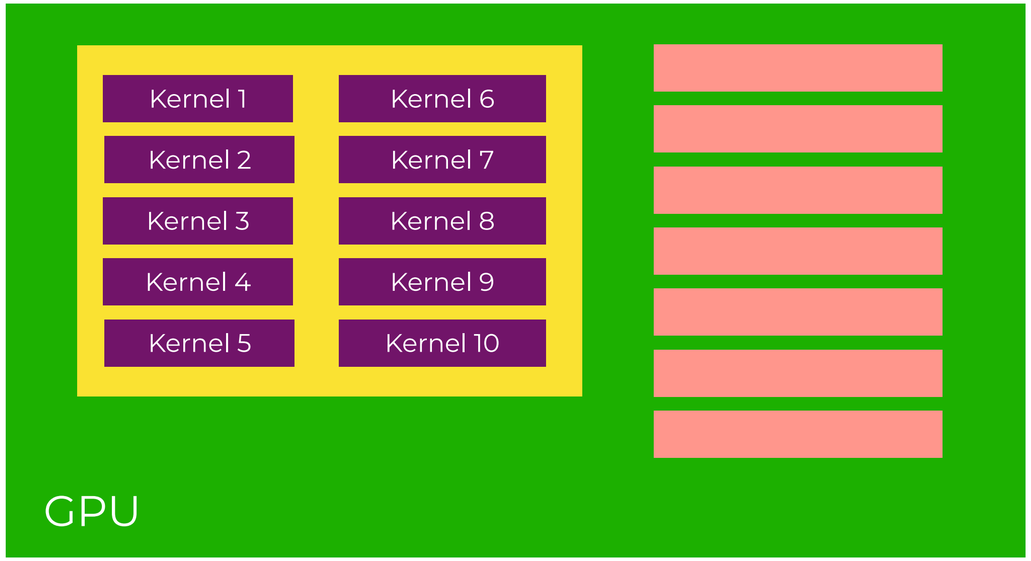
```java
import com.aparapi.Kernel;
import com.aparapi.Range;

public class AparapiExample {

    public static void main(String[] args) {
        // Array size
        final int size = 10;
        final float[] input = new float[size];
        final float[] output = new float[size];

        // Fill the input array with sample data
        for (int i = 0; i < size; i++) {
            input[i] = i;
        }

        // Define the kernel in Java, which will be processed on the GPU
        Kernel kernel = new Kernel() {
            @Override
            public void run() {
                int id = getGlobalId();
                output[id] = input[id] * input[id];
            }
        };

        // Define the range (number of threads)
        Range range = Range.create(size);

        kernel.execute(range);

        System.out.println("Results:");
        for (int i = 0; i < size; i++) {
            System.out.println(input[i] + " squared = " + output[i]);
        }

        // Clean up the kernel resources
        kernel.dispose();
    }
}
```

Kernels expressed in Java



Parallel processing

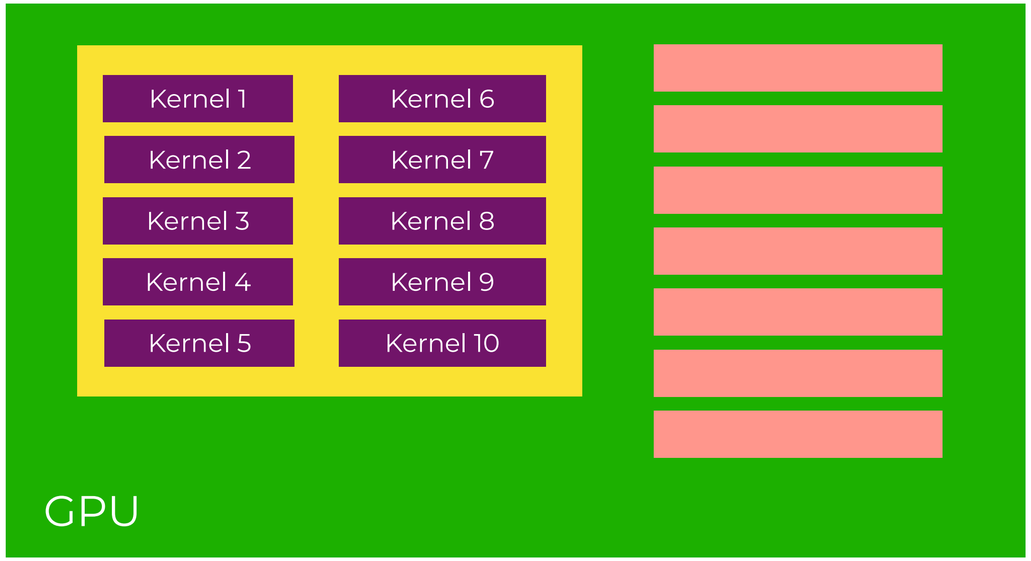
?



Streams.parallel()

```java
public class ParallelStreamExample {
    public static void main(String[] args) {

        int[] arr = new int[10000];
        for (int i = 0; i < arr.length; i++)
{           arr[i] = i;
        }

        int sum = Arrays.stream(arr)
                        .parallel()
                        .sum();

        System.out.println("Sum: " + sum);
    }
}
```

Project Sumatra

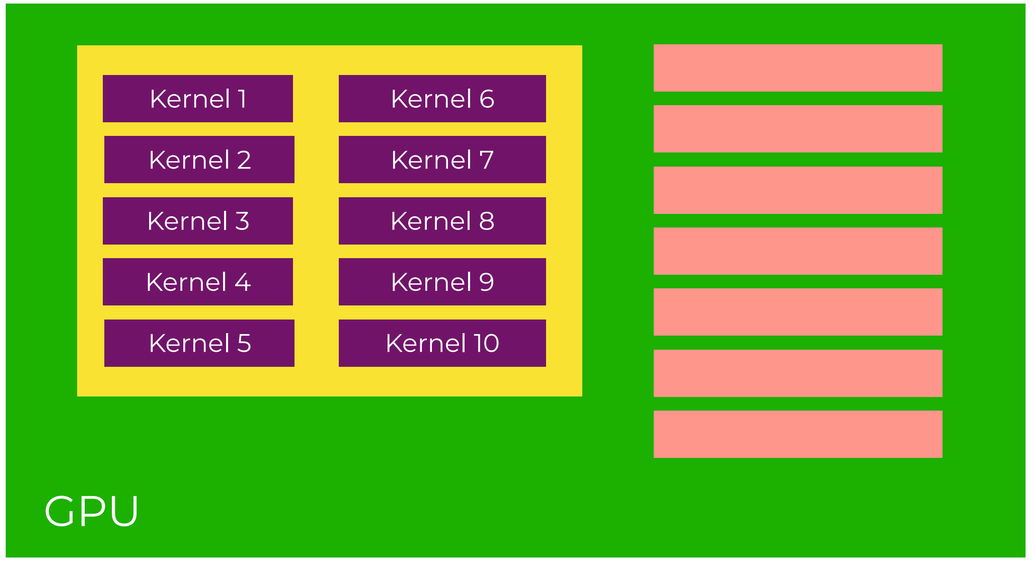



```java
public class ParallelStreamExample {
    public static void main(String[] args) {

        int[] arr = new int[10000];
        for (int i = 0; i < arr.length; i++)
{           arr[i] = i;
        }

        int sum = Arrays.stream(arr)
                        .parallel()
                        .sum();

        System.out.println("Sum: " + sum);
    }
}
```

GraalVM™

Streams.parallel()

Project Sumatra



Streams.parallel()

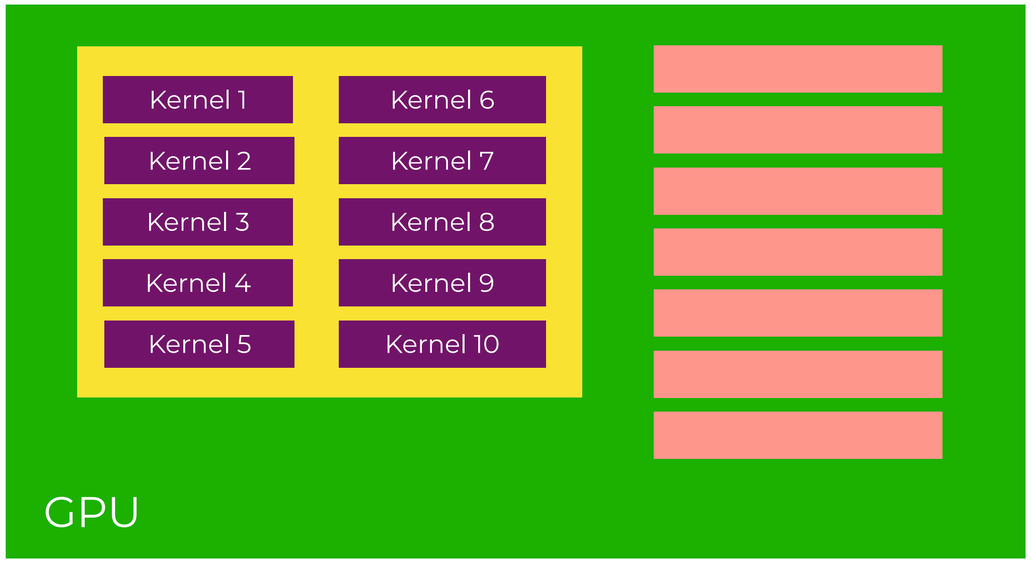
```
public class ParallelStreamExample {
    public static void main(String[] args) {

        int[] arr = new int[10000];
        for (int i = 0; i < arr.length; i++)
{           arr[i] = i;
        }

        int sum = Arrays.stream(arr)
                        .parallel()
                        .sum();

        System.out.println("Sum: " + sum);
    }
}
```

GraalVM™

NVIDIA CUDA

OpenCL™

Project Sumatra



Streams.parallel()

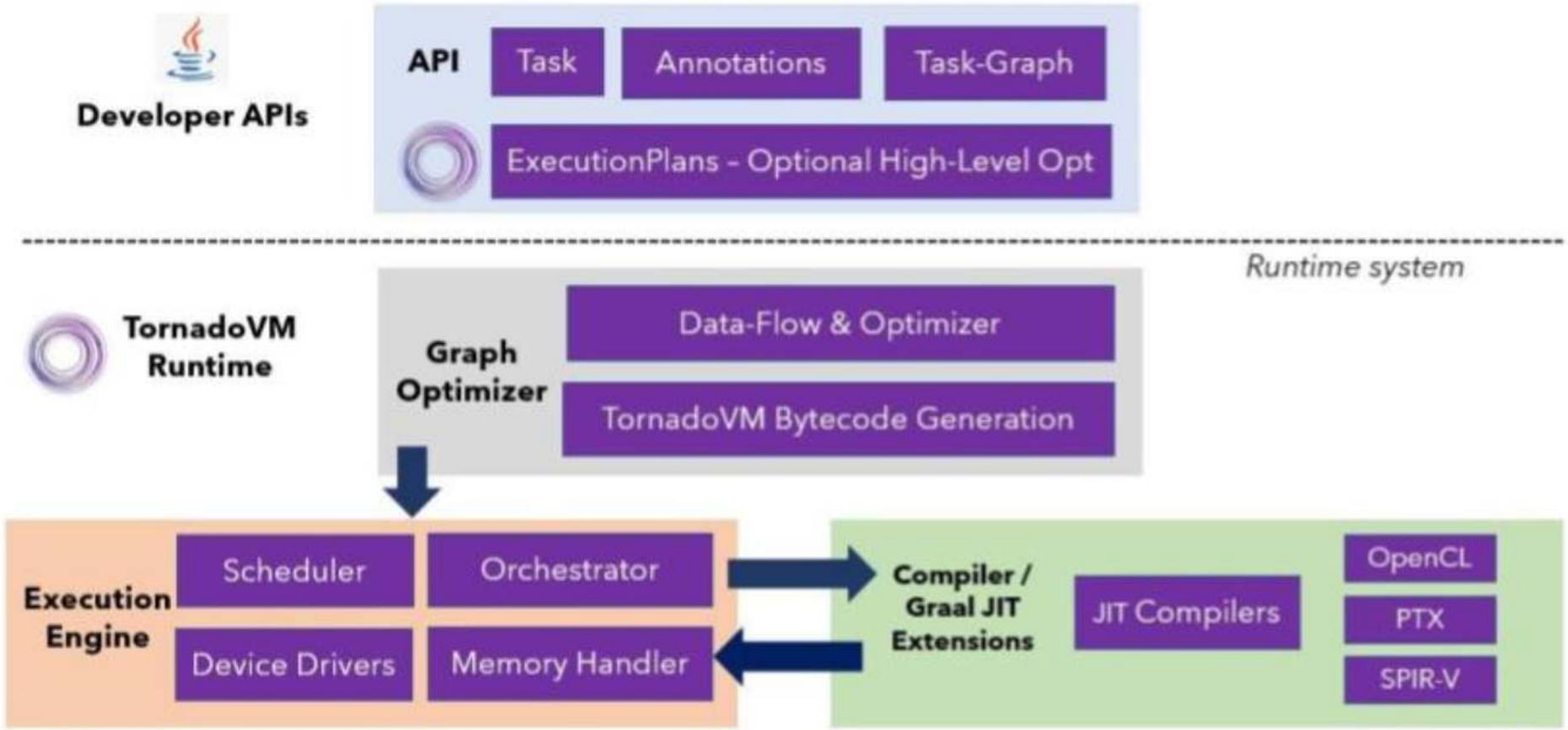
Data transfer between JVM and GPU can reduce performance benefits, especially for small tasks.

JDK Sumatra's approach to GPU acceleration has been largely superseded by **GraalVM**'s capabilities, which provide broader optimization features (e.g., native image and polyglot execution) and better performance across various platforms

Project Sumatra

Next Level: TornadoVM

TornadoVM Software Stack



Next Level: TornadoVM

- Task Graphs - Direct abstraction for NDGrid
- A lot of different backends (not only CUDA)
- Managed Memory
- JIT Support through GraalVM

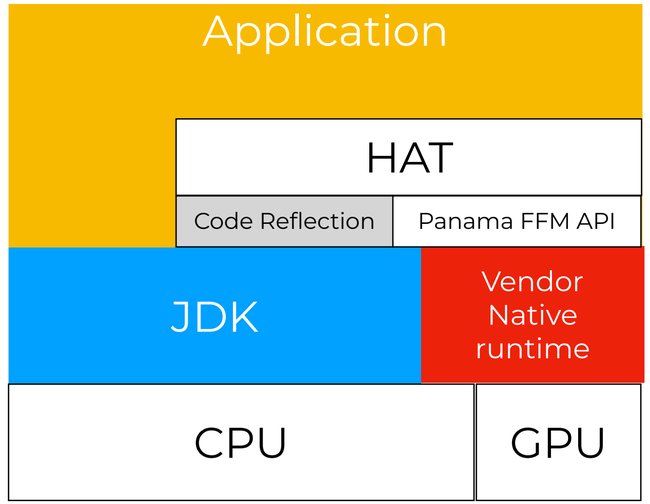
```java
import uk.ac.manchester.tornado.api.TaskSchedule;
import uk.ac.manchester.tornado.api.collections.types.FloatArray;

public class TornadoSquareExample {

    public static void main(String[] args) {
        // Rozmiar tablicy
        final int size = 1024;
        float[] input = new float[size];
        float[] output = new float[size];

        // Inicjalizacja tablicy wejściowej
        for (int i = 0; i < size; i++) {
            input[i] = i;
        }

        // Zadanie obliczające kwadrat każdego elementu w tablicy
        TaskSchedule task = new TaskSchedule("s0")
            .task("t0", (float[] input, float[] output) -> {
                for (int i = 0; i < input.length; i++) {
                    output[i] = input[i] * input[i];
                }
            }, input, output);

        // Wykonanie zadania na GPU lub innym urządzeniu
        task.execute();

        // Wyświetlenie wyników
        for (int i = 0; i < size; i++) {
            System.out.println(input[i] + " squared = " + output[i]);
        }
    }
}
```

Next Level: TornadoVM

Future: HAT & Project Babylon

Heterogenous Accelerator Toolkit (HAT)

- NDRange API
- FFM data wrapping patterns
- Support for Code Reflection from Project Babylon
- Support for FFM API



Heterogenous Accelerator Toolkit (HAT)

Code reflection is an extension of Java reflection that allows access to symbolic representations of Java code, such as method bodies and lambda expressions, at runtime.

This makes it possible to programmatically manipulate Java code.

Code Reflection

```
@CodeReflection
public static void kernel(KernelContext kc, S32Arr s32Arr) {
 s32Arr.array(kc.x, s32Arr.array(kc.x) * s32Arr.array(kc.x));
}
```

Code Models are representations of code in the form of Java classes that retain information about types and structure, but without full syntactic details (as in AST) or type flattening (as in bytecode).

This allows for easier analysis and transformation of the code.

Code Models

Paul Sandoz
June 2024

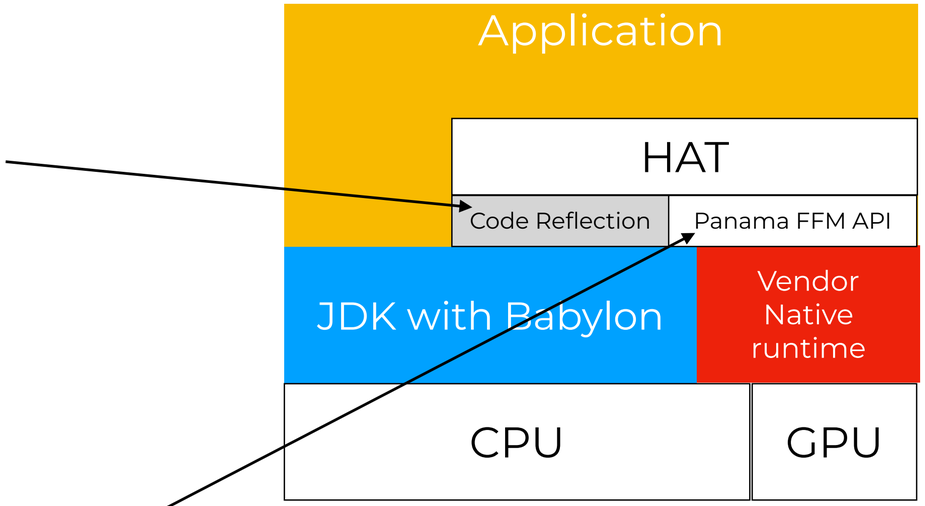
Code reflection, an enhancement to Java reflection, enables access to symbolic representations of Java code in method bodies and lambda bodies. “Symbolic representations of Java code” may seem like a fancy term, but its easily demystified. It’s a model of Java code where the code of a method or lambda body is represented as instances of specific Java classes arranged in an appropriate structure. Thereby it is possible to write Java programs that manipulate Java programs. Before we get into the details of how code reflection models Java code we should talk about two existing approaches to modeling Java code.

```
// Depth-first search, reporting elements in pre-order
model.traverse(null, (acc, codeElement) -> {
    // Count the depth of the code element by
    // traversing up the tree from child to parent
    int depth = 0;
    CodeElement<?, ?> parent = codeElement;
    while ((parent = parent.parent()) != null) depth++;
    // Print out code element class
    System.out.println("  ".repeat(depth) + codeElement.getClass());
    return acc;
});
```

Code Models

Babylon can use code reflection to dynamically generate GPU code (e.g., OpenCL, CUDA) by analyzing the code model and converting fragments of Java code into corresponding GPU instructions.

Panama FFM API and off-heap **MemorySegments** eliminate the need for manual memory management between the JVM and GPU, speeding up data exchange and increasing performance.



Exploring Triton GPU programming for neural networks in Java

Paul Sandoz
February 2024

In this article we will explain how we can use Code Reflection to implement the Triton programming model in Java as an alternative to Python.

Code Reflection is a Java platform feature being researched and developed under OpenJDK Project Babylon.

We will introduce Code Reflection concepts and APIs as we explain the problem and present a solution. The explanations are neither exhaustive nor very detailed, they are designed to give the reader an intuitive sense and understanding of Code Reflection and its capabilities.

Code Models

ORACLE

(Heterogeneous Accelerator Toolkit) HAT Update

How Babylon and Panama Enable Java GPGPU Collaboration Opportunities

Gary Frost and Paul Sandoz

8/5/2024

https://cr.openjdk.org/~psandoz/conferences/2024-JVMLS/JAVA_BABYLON_HAT-JVMLS-24-08-05.pdf

Panama & Valhalla SIMD:

Where are we?

Panama & Valhalla SIMD:

Valhalla Half-Float:

Where are we?

Panama & Valhalla SIMD:

Valhalla Half-Float:

GraalPy:

Where are we?

Panama & Valhalla SIMD:

Valhalla Half-Float:

GraalPy:

JCuda:

Where are we?

Panama & Valhalla SIMD:

Valhalla Half-Float:

GraalPy:

JCuda:

Sumatra:

Where are we?

Panama & Valhalla SIMD:

Valhalla Half-Float:

GraalPy:

JCuda:

Sumatra:

TornadoVM:

HAT & Babylon:

Where are we?

Panama & Valhalla SIMD:

Valhalla Half-Float:

GraalPy:

JCuda:

Sumatra:

TornadoVM:

HAT & Babylon:

Where are we?

JVM Weekly

Dashboard

Home Notes Archive About

Java 23 and GraalVM for JDK 23 Released - JVM Weekly vol. 100

Today, we have a double occasion! But first, the duties (after all, we have the new JDK 23), and then the pleasures (my little celebration).

SEP 19 · ARTUR SKOWRONSKI

Latest Top Discussions

Diagrams, diagrams, diagrams: Classloaders and the Java Developer Roadmap - JVM Weekly vol. 99

The final countdown.

SEP 12 · ARTUR SKOWRONSKI

Best of Foojay.io August 2024 Edition – JVM Weekly vol. 98

A month has passed, and it's time for another review of Foojay articles.

SEP 5 · ARTUR SKOWRONSKI

I've watched all the talks from JVMLS so you know why it's worth to do it - JVM Weekly vol. 97

Today we have one topic - JVMLS.

AUG 29 · ARTUR SKOWRONSKI

JVM Weekly

From the latest updates in JVM languages like Java, Kotlin, and Scala to emerging technologies like GraalVM and Quarkus, this newsletter covers a wide range of topics that are of interest to developers and tech enthusiasts.

Recommendations MANAGE

- **Fractional Architect** — MJ
- **Tales from the jar side** — Ken Kousen
- **Tech Talks Weekly** — Tech Talks Weekly
- **Architecture Weekly** — Oskar Dudycz
- **Engineering Primer** — Bartek Antoniak

Thank you 🤦

@ArturSkowronski

VIRTUSLAB

jvm weekly