Compilers 101

Outro

Previously…

- Preprocessing
- Lexical analysis
- Syntax analysis
- Semantic analysis
- IR Generation
- IR Optimization
- Code generation
- Optimization

Fronten
d

Middle-end

Backend

Today

- 3rdparty projects using LLVM (and other compiler infrastructures)
- AOT vs JIT
- Compilers integration into ML workflows
- Python compilation process

LLVM is not the only compiler infrastructure

Throughout this course, we were talking about LLVM compiler infrastructure

But this is not the one and only compiler infrastructure (if we focus on C/C++ only):

- https://github.com/llvm/llvm-project
- https://github.com/gcc-mirror/gcc

Some smaller alternatives (not ready for production):

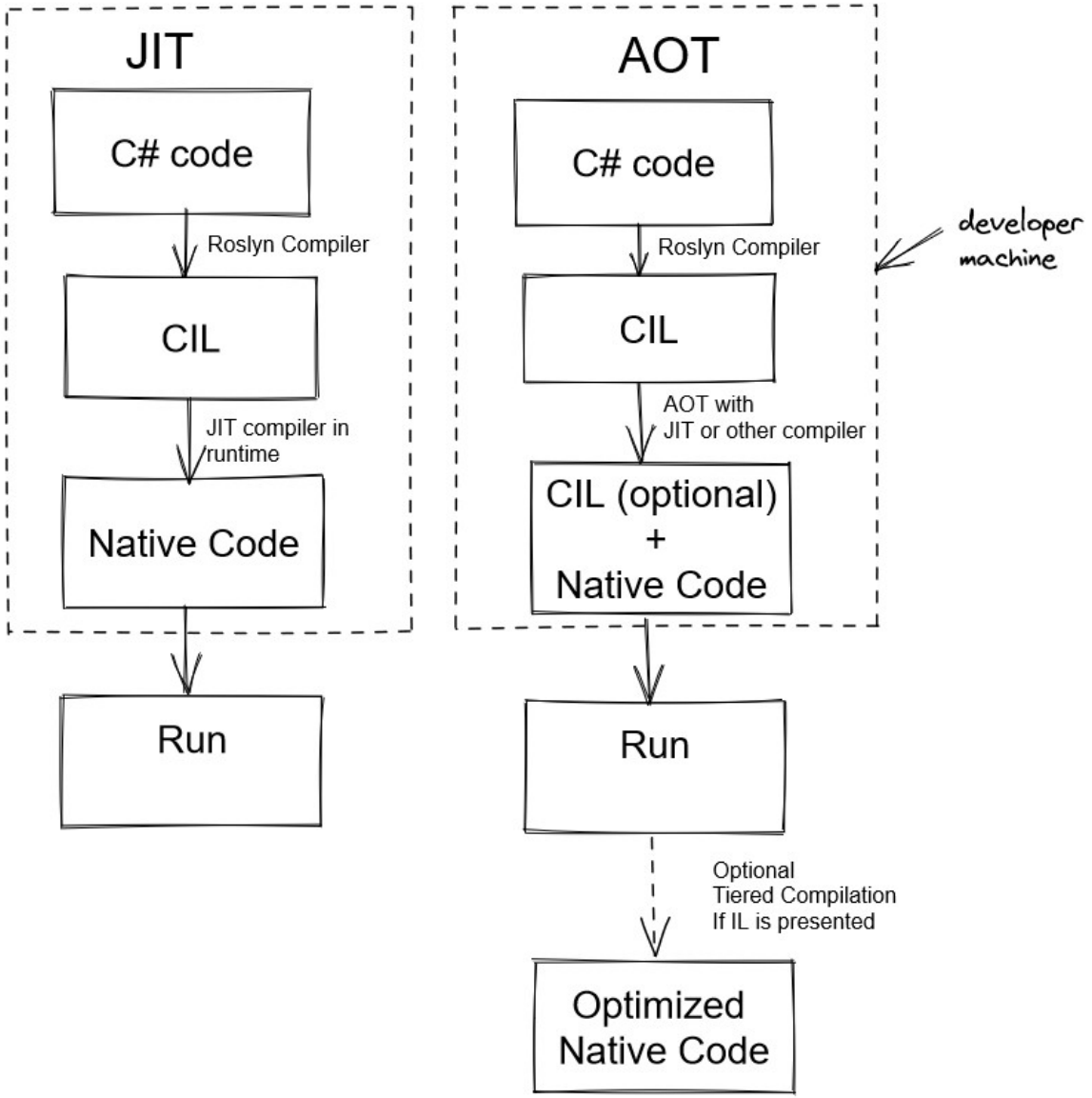
- https://bellard.org/tcc/
- https://github.com/dstogov/rcc
- https://github.com/rui314/chibicc

Compilation types: AOT vs JIT

A compiler translates a program from one representation to another.

But when the transformation happens?

- Before execution (ahead-of-time, AOT)
  - The program is compiled into machine code in advance
- During execution (just-in-time, JIT)
  - The program is compiled while it is running



https://flerka.github.io/personal-blog/2022-06-21-ahead-of-time-compilation/

AOT (ahead-of-time) compilation

Idea: compile the program before it is executed

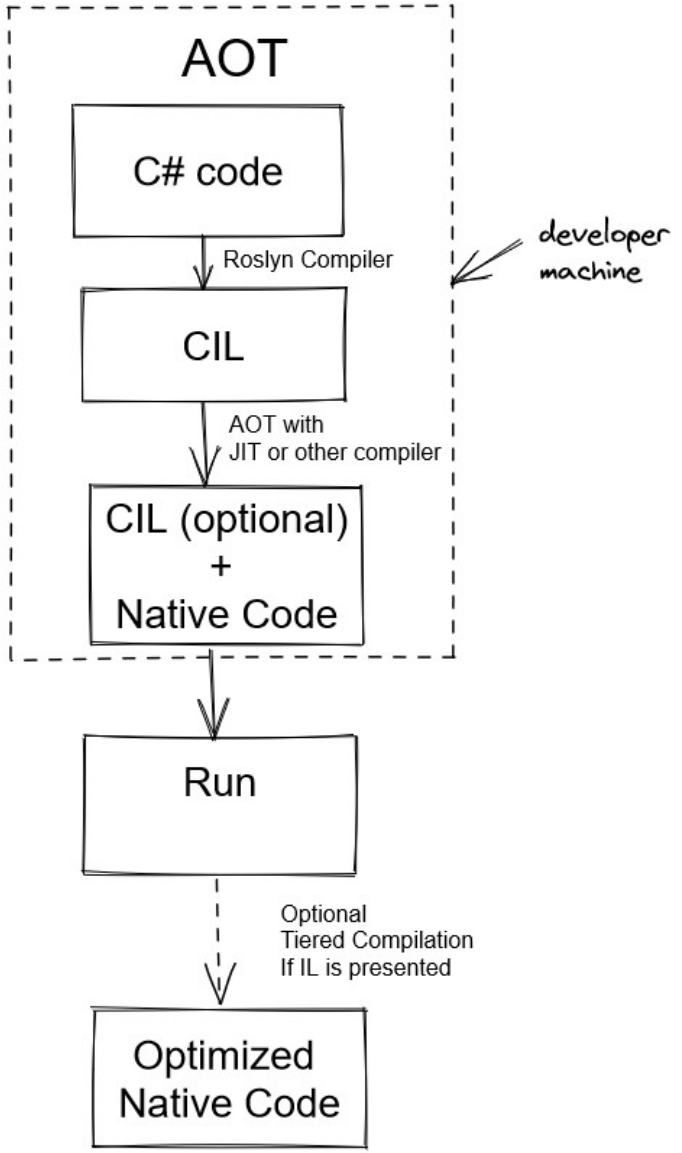
Examples: statically compiled native languages (C/C++, Rust, Go, …)

+ Fast startup: code is already compiled
+ Easier deployment as a native binary
+ Predictable performance
+ Many optimizations can be done before execution
+ No compiler has to run during program execution

- Less information about actual runtime behavior
- Binary is usually platform-specific
- Harder to optimize for real input data and real execution paths
- Compilation may take a long time

Typical use cases:

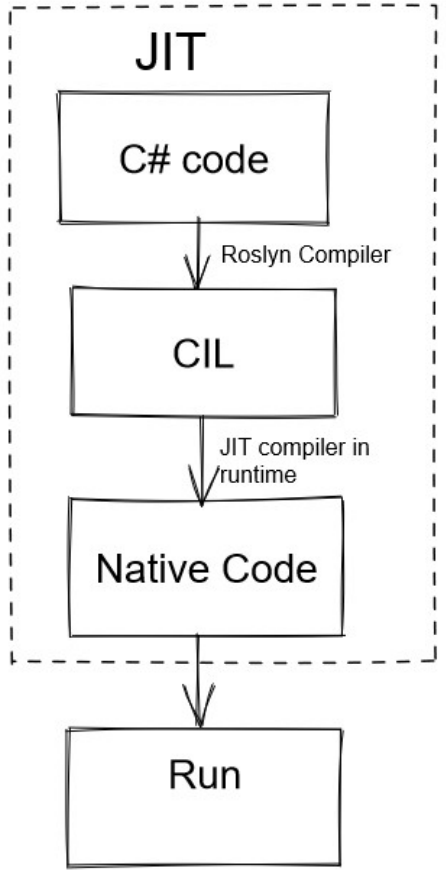
- Operating systems
- Embedded systems
- Performance-critical native applications



JIT (just-in-time) compilation

Idea: compile code while the program is running

Examples: dynamically compiled languages and their runtimes (Java JVM, .NET CLR, JavaScript engines such as V8, PyPy)



+ Can use runtime information for optimization
+ Can optimize hot paths: the parts of code executed most often
+ Can specialize code for actual types, values, CPU features, and branches
+ Portable bytecode can run on many platforms

- Slower startup due to interpretation and compilation overhead
- More complex runtime system
- Uses extra memory for profiling data and generated machine code
- Performance can change during execution due to recompilation and lack of optimizations

Typical use cases:

- Web browsers
- Virtual machines
- Managed languages
- Dynamic languages

AOT vs JIT

Feature	AOT	JIT
Compilation time	Before execution	During execution
Startup time	Usually faster	Usually slower
Runtime optimization	Limited	Strong
Runtime profiling	Usually unavailable	Available
Portability	Binary is platform-specific	Bytecode/IR can be portable
Runtime system complexity	Lower	Higher
Peak performance	High	Can be very high
Memory overhead	Lower	Higher

Integrating compilers to the workflow

- Programming language implementations
  - Apart from C/C++/Fortran: Rust, Swift, Julia, Haskell and many-many others
- Machine learning frameworks
  - TensorFlow, PyTorch, OpenXLA
- GPU and accelerator software
  - CUDA, ROCm, SYCL
- Databases and query engines
  - JIT query compilation in PostgreSQL
- Game engines and shader tools
  - shader compilation in Unreal Engine, Unity 3D
- Static analysis and developer tools

Compiler infrastructure inside products

- Frontend reuse:
  - IDEs, linters, static analyzers, refactoring tools
- Middle-end reuse:
  - Custom IR transformations, custom optimization pipelines, domain-specific lowering
- Backend reuse:
  - CPU/GPU code generation, JIT, cross-compilation

LLVM: in-tree and out-of-tree

In-tree projects:

- Projects living in LLVM monorepo (https://github.com/llvm/llvm-project)
- Examples: clang, clang-tools-extra (clang-tidy, …), flang, mlir, lld, lldb, …

Out-of-tree projects:

- Projects developed outside the LLVM monorepo but built against, extending, or reusing LLVM/Clang/MLIR libraries, APIs

LLVM-based compilers for other languages

- rustc compiler: https://github.com/rust-lang/rust
- Swift compiler: https://github.com/swiftlang/swift
- Julia compiler: https://github.com/JuliaLang/julia
- Emscripten (C++ to Wasm): https://github.com/emscripten-core/emscripten
- Zig: https://codeberg.org/ziglang/zig
- Haskell: https://github.com/ghc/ghc

3rdparty projects using LLVM

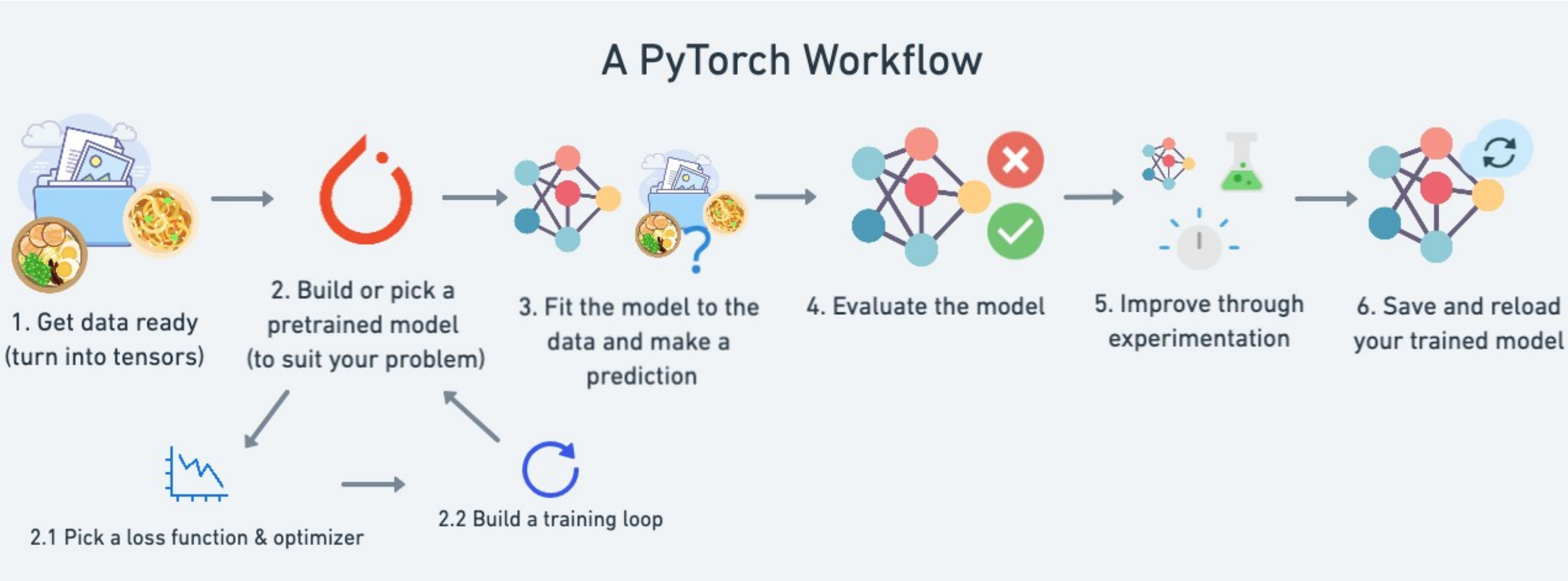
- PostgreSQL JIT: https://github.com/postgres/postgres/blob/master/src/backend/jit
- ClickHouse database: https://github.com/ClickHouse/ClickHouse
- Mesa llvmpipe - Software OpenGL rasterizer: https://docs.mesa3d.org/drivers/llvmpipe.html
- The Interactive C++ Interpreter: https://github.com/root-project/cling

PyTorch

PyTorch

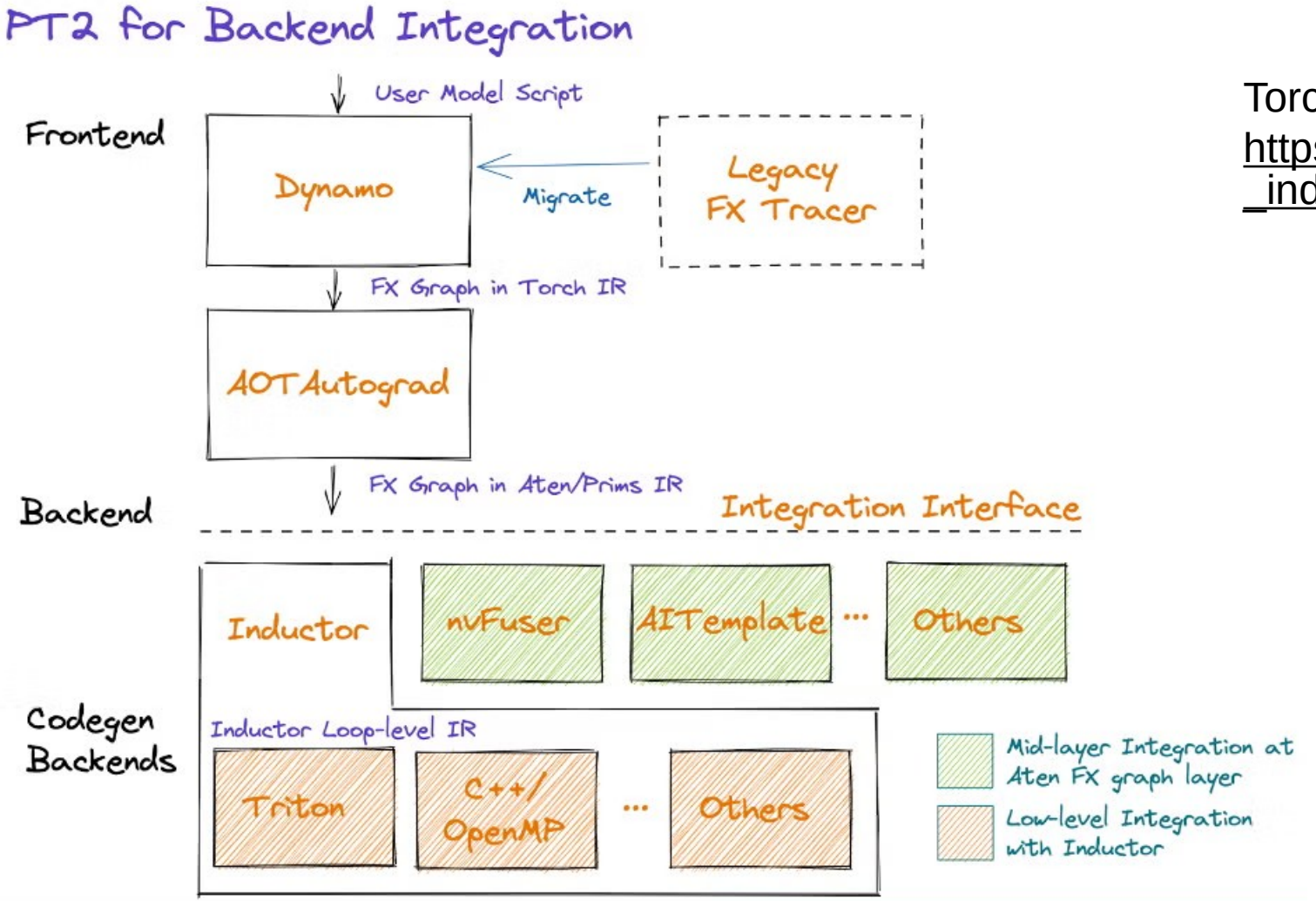
https://github.com/pytorch/pytorch

PyTorch is an open-source, Python-based machine learning library



https://www.learnpytorch.io/01_pytorch_workflow/

PyTorch compilation flow



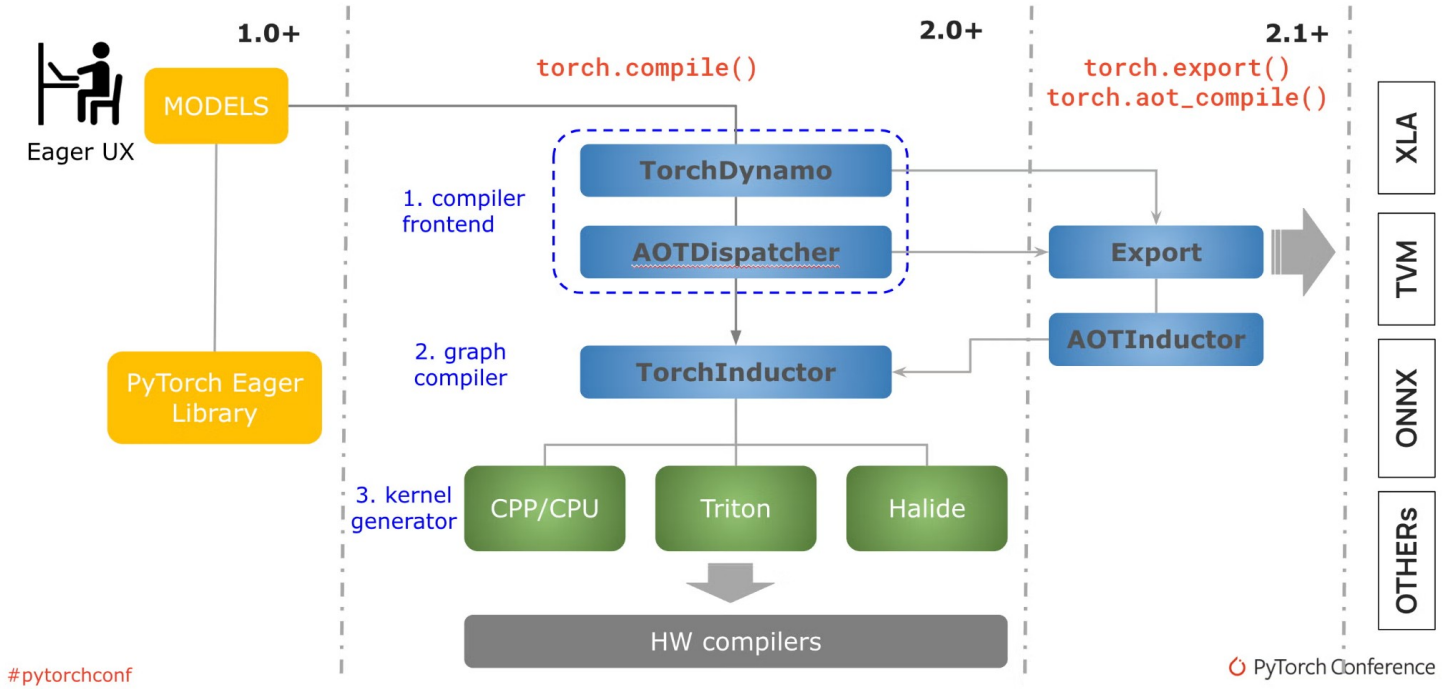
Torch provides a set of compilers by itself: https://docs.pytorch.org/tutorials/compilers_index.html

With PyTorch 2.0, we want to simplify the backend (compiler) integration experience. To do this, we have focused on reducing the number of operators and simplifying the semantics of the operator set necessary to bring up a PyTorch backend.

https://pytorch.org/get-started/pytorch-2-x/

PyTorch compilation modes

- inductor: the default production backend in torch.compile
- eager: not an optimizing compiler. It runs Dynamo graph capture but executes with regular PyTorch eager mode. Useful for debugging whether graph itself works or not



https://pytorch.org/blog/accelerated-pytorch-inference/

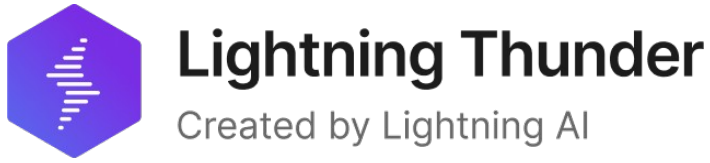
PyTorch vendor (3rdparty) backends

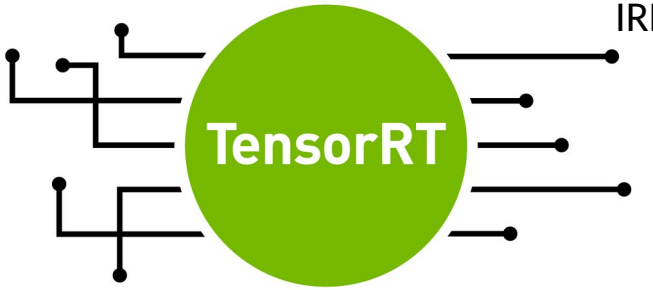
PyTorch provides an extendable interface that allows to bind different compilers for the actual model compilation

https://medium.com/@Modexa/eight-torchscript-alternatives-for-the-pytorch-2-x-era-34dcb68f2940

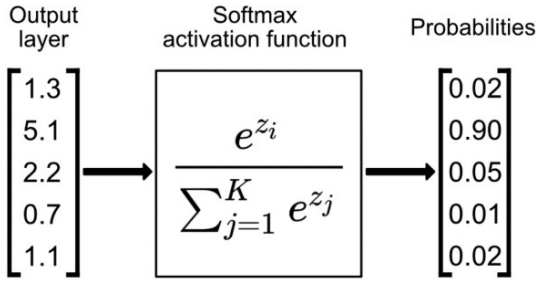
PyTorch backends

- https://pytorch.org/docs/stable/torch.compiler.html
  - Targets CPUs, GPUs
- https://github.com/Lightning-AI/lightning-thunder
  - Performs transformations on Pythonic IR (for NVIDIA GPUs)
- https://github.com/iree-org/iree-turbine
  - MLIR-based compiler/runtime path for PyTorch models using Torch-MLIR-style lowering
- https://github.com/pytorch/TensorRT
  - compiler for NVIDIA GPUs
- https://docs.openvino.ai/torchcompile
  - for Intel CPU/GPU/NPU deployment
- https://github.com/pytorch/xla
  - XLA, mainly for Google TPU and XLA devices





Triton

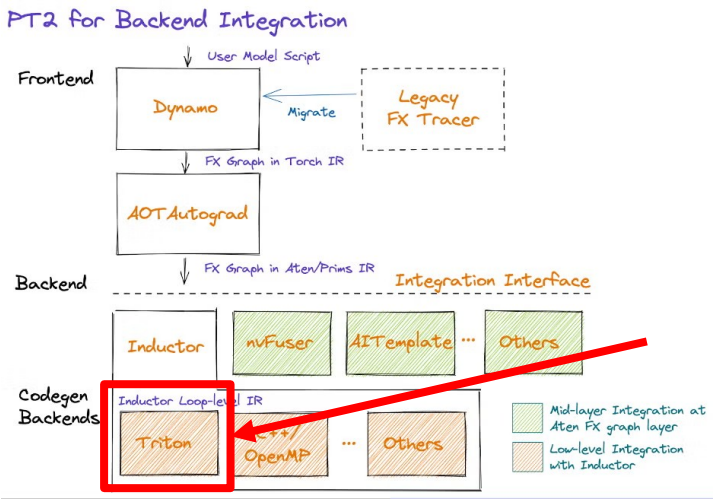


Triton is a Python-based language and compiler for writing high-performance GPU kernels without writing CUDA (or other GPU code) directly

It serves DSL and compiler for custom GPU kernels

Goal: Make high-performance GPU kernel programming more accessible

LLVM and MLIR are under the hood



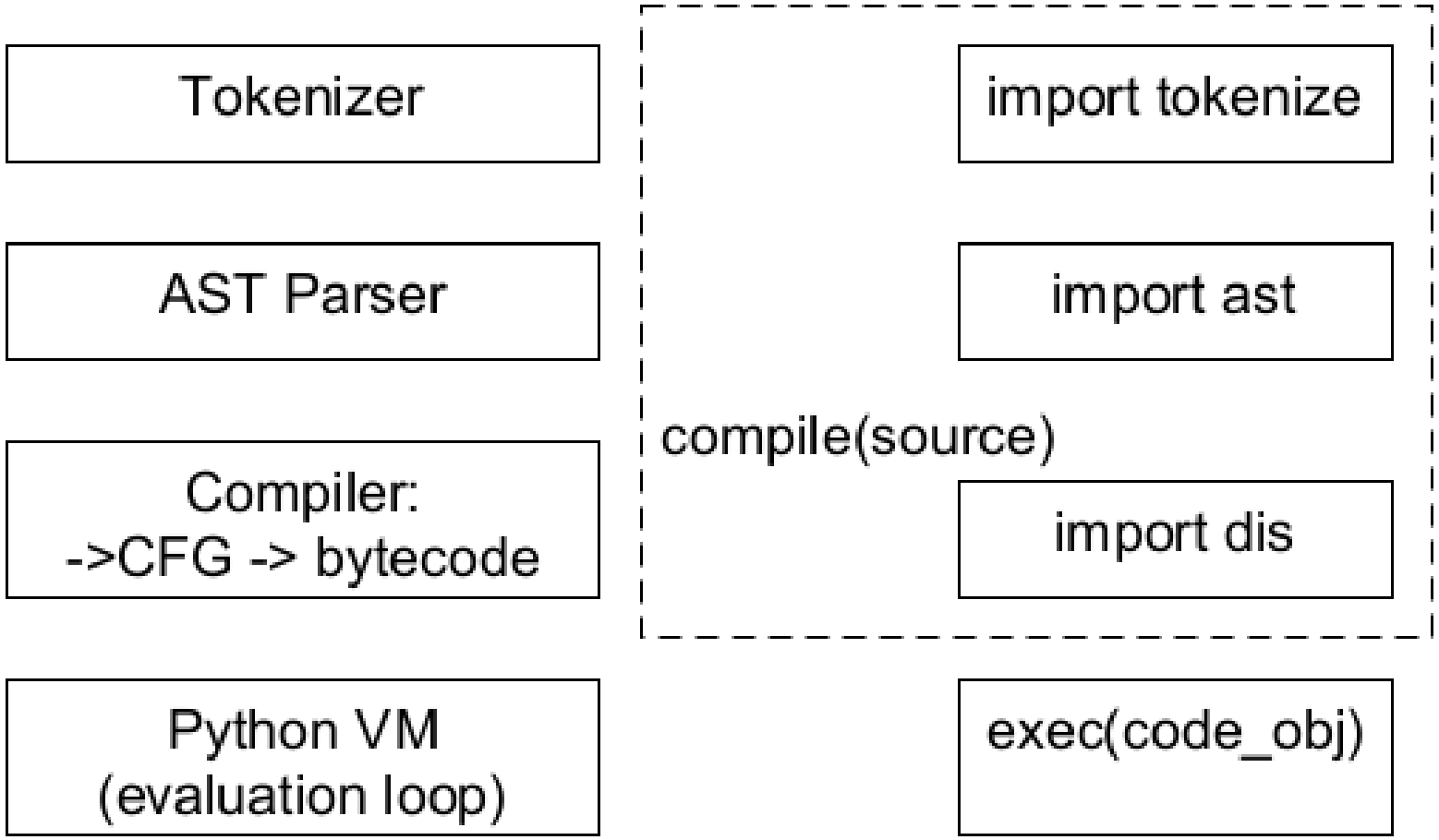
https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/

https://openai.com/index/triton/
https://triton-lang.org/main/index.html

```python
import triton
import triton.language as tl

@triton.jit
def softmax(Y, stride_ym, stride_yn, X, stride_xm, stride_xn, M,
N, BLOCK_SIZE: tl.constexpr):
  m = tl.program_id(0)
  n = tl.arange(0, BLOCK_SIZE)
  X = X + m * stride_xm + n * stride_xn
  x = tl.load(X, mask=n < N, other=-float('inf'))
  z = x - tl.max(x, axis=0)
  num = tl.exp(z)
  denom = tl.sum(num, axis=0)
  y = num / denom
  Y = Y + m * stride_ym + n * stride_yn
  tl.store(Y, y, mask=n < N)

import torch
# Allocate input/output tensors
X = torch.normal(0, 1, size=(583, 931), device='cuda')
Y = torch.empty_like(X)
# SPMD launch grid
grid = (X.shape[0], )
# enqueue GPU kernel
softmax[grid](Y, Y.stride(0), Y.stride(1),
              X, X.stride(0), X.stride(1),
              X.shape[0]    , X.shape[1],  1024)
```

https://godbolt.org/z/fc37Edob6

Python interpreter <-> built-in modules & functions



Python AST Parser

```
python3 -m ast --help
```

```
ast.parse("def sum(a, b): return a + b")
```

Classes and methods:

- `ast.NodeVisitor` - base class to walk AST in read-only mode
- `ast.NodeTransformer` - base class to walk AST to replace/remove nodes
- `ast.walk()` - walk without Visitor pattern
- `ast.literal_eval() / eval()` - evaluate AST
- `ast.unparse()` – useful for decompiler or obfuscator

```
$ python3
Python 3.14.3 (main, Feb  3 2026, 15:32:20)
[Clang 17.0.0 (clang-1700.6.3.2)] on darwin
Type "help", "copyright", "credits" or
"license" for more information.
>>> import ast
>>> a = ast.parse("def sum(a, b): return a +
b")
>>> a
Module(body=[FunctionDef(name='sum',
args=arguments(posonlyargs=[],
args=[arg(...), arg(...)], vararg=None,
kwonlyargs=[], kw_defaults=[], kwarg=None,
defaults=[]),
body=[Return(value=BinOp(...))],
decorator_list=[], returns=None,
type_comment=None, type_params=[])],
type_ignores=[])
>>> ast.unparse(a)
'def sum(a, b):\n    return a + b'
```

Python Virtual Machine

- Python source code is first compiled into bytecode
- Bytecode is a low-level, platform-independent instruction format
- The Python Virtual Machine executes this bytecode
- In CPython, the VM is an interpreter written mostly in C
- Execution is stack-based (like JVM): bytecode instructions push and pop values from a virtual stack

```
>>> o = compile("def sum(a, b): return a + b", "<string>", "exec")
>>> o
<code object <module> at 0x00000216339D3730, file "<string>", line 1>
>>> dis.dis(o)
  1           0 LOAD_CONST               0 (<code object sum at 0x000
              2 LOAD_CONST               1 ('sum')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (sum)
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE

Disassembly of <code object sum at 0x00000216339D
  1           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 RETURN_VALUE
```

Python bytecode (IR)

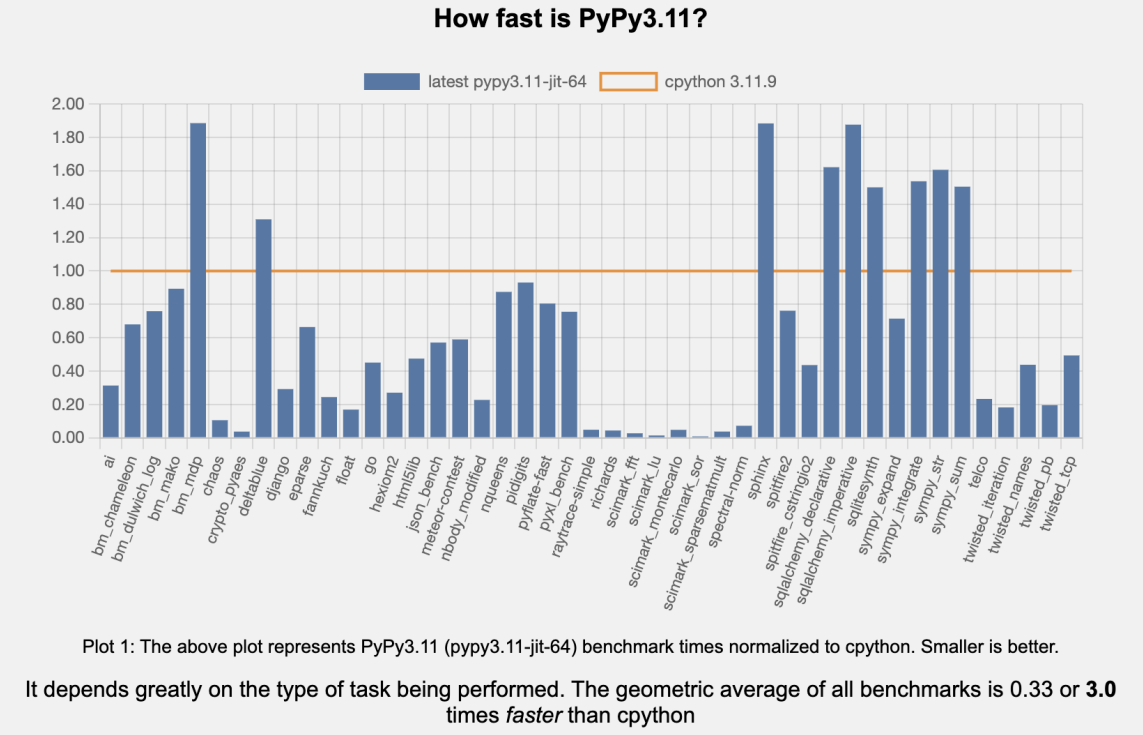
- As any IR it has 3 forms:
  - Byte-code itself: ***.pyc** file content or code_object**.co_code**
  - In-memory representation: **code_object = compile(...)**
  - String representation: **import dis; dis.dis(code_object)**

```
>>> o = compile("def sum(a, b): return a + b", "<string>", "exec")
>>> o
<code object <module> at 0x00000216339D3730, file "<string>", line 1>
>>> dis.dis(o)
  1           0 LOAD_CONST               0 (<code object sum at 0x000
              2 LOAD_CONST               1 ('sum')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (sum)
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE

Disassembly of <code object sum at 0x00000216339D
  1           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 RETURN_VALUE
```

JIT compiler for Python

PyPy is a Python implemented in RPython as a JIT compiler

- Provides good compatibility for pure Python, but C-extension-heavy workloads remain problematic
- CPython 3.14 is stable (as of May 2026), while current PyPy targets only Python 3.11
- 3 times faster on average comparing with CPython 3.11*



Plot 1: The above plot represents PyPy3.11 (pypy3.11-jit-64) benchmark times normalized to cpython. Smaller is better.

It depends greatly on the type of task being performed. The geometric average of all benchmarks is 0.33 or **3.0** times *faster* than cpython

https://speed.pypy.org/

* numbers are workload dependent, may vary in different scenarios

https://github.com/pypy/pypy

JIT compilers for Python functions

Numba

- JIT-compiles Python functions to machine code via LLVM
- Best for NumPy-heavy numerical code
- Supports CPU and some GPU targets, especially CUDA
- Minimal code changes: often just add `@numba.jit`

```python
from numba import njit
import random

@njit
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

https://numba.pydata.org/
https://github.com/numba/numba

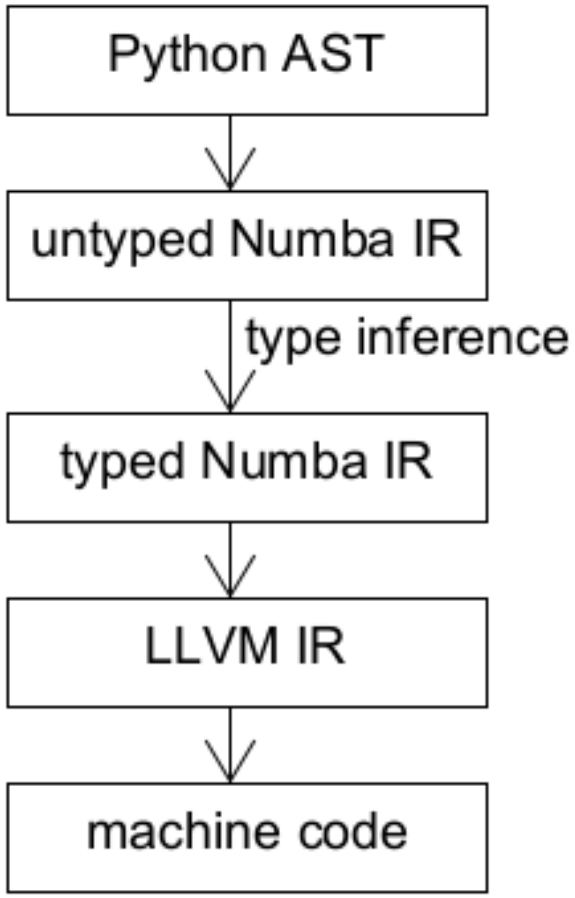
Taichi

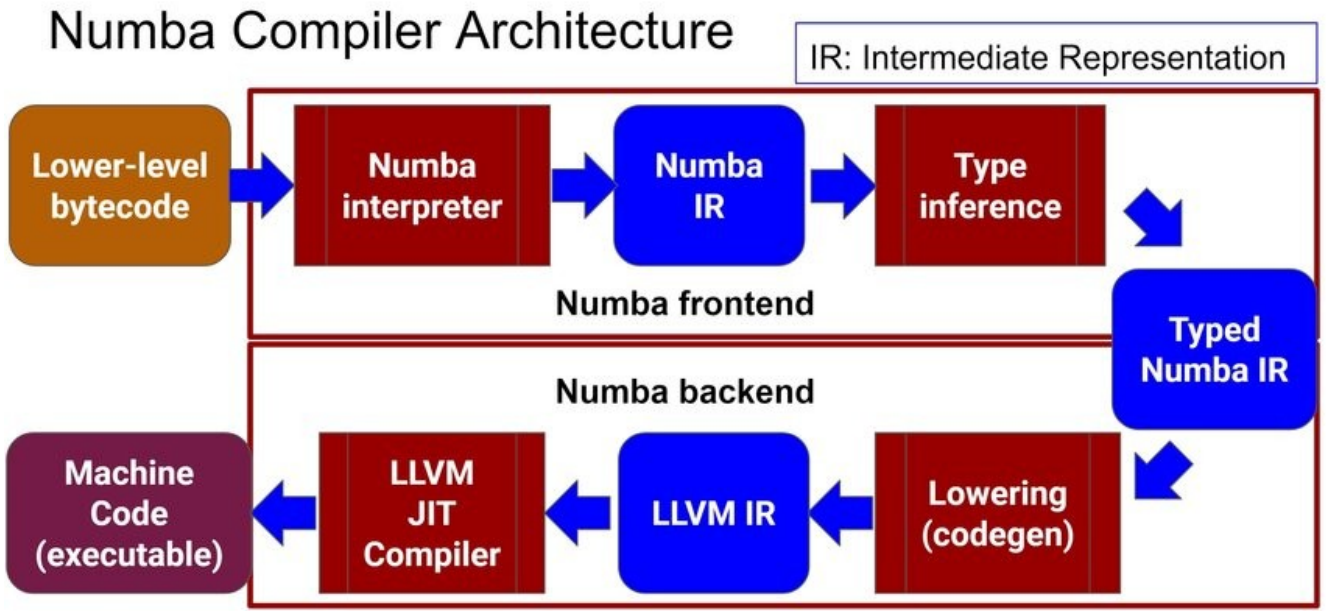
- Python-embedded DSL for high-performance kernels
- Designed for parallel simulation, graphics, and numerical computing
- Targets CPU/GPU backends

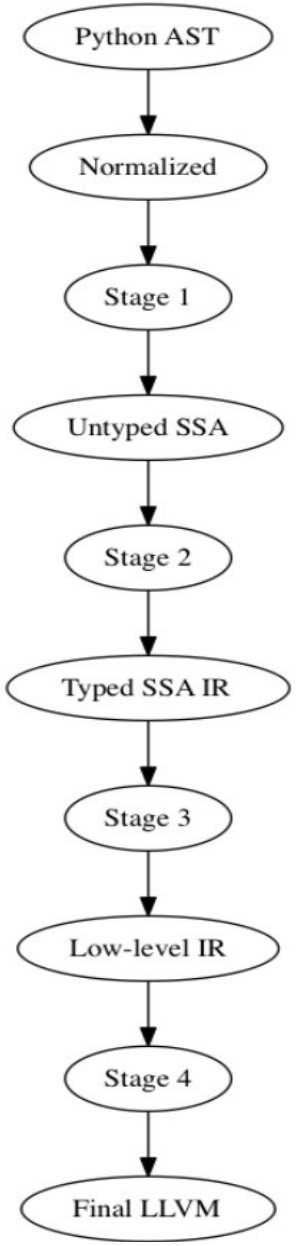
```python
import taichi as ti
ti.init()

@ti.kernel
def monte_carlo_pi(n: int) -> float:
    total = 0
    for i in range(n):
        x = ti.random()
        y = ti.random()
        if (x ** 2 + y ** 2) < 1.0:
            total += 1
    return 4.0 * total / n
```

https://www.taichi-lang.org/
https://github.com/taichi-dev/taichi

Typical compilation pipeline (Numba)







Saedi, Omid. "Towards Eco-Conscious Python: A Comparative Analysis of Performance, Energy Efficiency and Carbon Emissions Between CPython and Alternative Implementations." (2024).

https://numba.pydata.org/numba-doc/dev/developer/architecture.html

Type inference for Python

- mypy + mypyC (transpiler to C)
- pytype (type hints are not necessary; by Google)
- pyright (type checker for VSCode; by Microsoft; written in TypeScript)

Python type hints are Turing-complete that means that full static checking/inference is undecidable

https://arxiv.org/pdf/2208.14755.pdf

mypy, pyright, pytype implement useful decidable approximations

Compilers evolution

Compiler are constantly evolving…

Understanding compilers means understanding how modern software is built, optimized and adapted to run on the hardware

Course feedback form

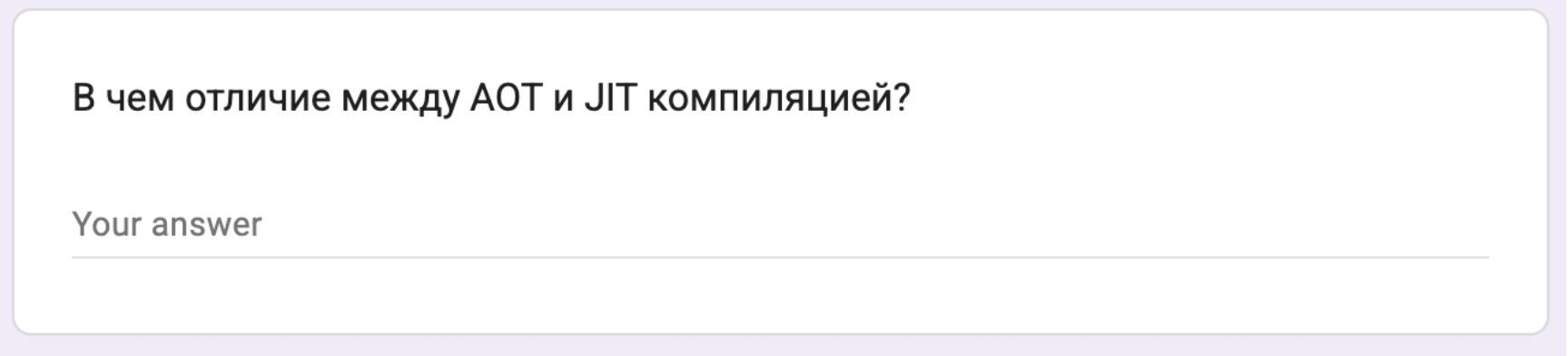
https://forms.gle/taqUuCfNCq4u5Xn9A

Backup: me@gooddoog.ru

Test

https://forms.gle/kQENCeeS6anmf6Lp8

Submission time: **10 minutes**



Backup: me@gooddoog.ru

Extra materials

- Triton Kernel Compilation Stages: https://pytorch.org/blog/triton-kernel-compilation-stages/
- CPython's internals: https://devguide.python.org/internals/