3 Python 与 Native Code 交互

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 理解 Python 解释执行模型对程序性能的影响及其根本原因；
- ⋆⋆⋆ 掌握 Python 调用 native code 的基本方式及其底层执行过程；
- ⋆⋆⋆ 理解装饰器的工作原理及其在跨语言调用中的工程意义。

3.1 解释执行的性能问题

3.1.1 解释执行 vs 编译执行

程序语言的执行方式通常可以分为两大类：解释执行与编译执行。

- **解释执行**：程序在运行时由解释器逐条读取源代码（或中间表示），并立即执行相应语义。我们在上一章学习的 Python 采用的正是这种解释执行模型：源代码首先被翻译为字节码，随后由 PVM 逐条解释执行。
- **编译执行**：程序运行前，由编译器将源代码整体翻译为面向目标硬件架构的机器代码（即 native code 或本地代码），程序运行时由处理器直接执行这些指令，而无需解释器参与。C/C++、Rust 等语言通常采用这种执行模式。由于编译过程发生在程序运行之前，因此这种方式也称为 Ahead-of-Time（AOT）编译。

解释执行的主要优点是执行过程灵活，便于交互式编程和动态特性支持，开发效率高。但每条指令的执行都伴随着解释、动态类型检查，运行时开销较大。编译执行运行阶段无需解释和类型检查等操作，且在编译阶段可进行全局优化（如内联、常量传播、寄存器分配等），因此运行性能更好。

3.1.2 案例分析

为具象化说明解释执行机制对 Python 数值计算性能的影响，本节设计了一组一维向量点积实验，对比纯 Python 实现、基于 NumPy 的实现以及手写 C 函数实现之间的运行开销差异。三种实现均采用相同的数学计算公式，以确保性能差异主要来源于执行模型与运行时机制，而非算法复杂度或计算逻辑本身。

一维向量点积的定义如下：

$$\mathrm{dot}(a, b) = \sum_{i=0}^{N-1} a_i \cdot b_i,$$

其中 a 与 b 为长度为 N 的实数向量。

实验组 1：纯 Python 实现 该实现完全使用 Python 语言编写，其循环控制与数值运算均由 Python 解释器执行。在 Python 中，对象（object）是程序运行时创建的基本实体，每个对象都具有三个基本属性：标识、类型和值。因此，Python 程序中的变量并不直接存储数据本身，而是保存对对象的引用。

在这种模型下，每一次循环迭代都会触发解释器的字节码调度，并涉及动态类型检查以及对象级别的算术运算。因此，该实现的整体运行性能在很大程度上依赖于 Python 解释执行的效率。

```python
def dot_python(a, b):
    # 使用 Python 的浮点对象进行累加
    s = 0.0

    # Python for循环
    # 每一次迭代都会经历解释器调度和动态类型检查
    for i in range(len(a)):
        # a[i] 和 b[i] 均为 Python 对象，
        # 乘法与加法均为对象级别运算
        s += a[i] * b[i]

    return s
```

代码 3.1: 纯 Python 实现的一维向量点积函数

实验组 2：基于 NumPy 的实现 NumPy 是 Python 生态中最核心的科学计算库之一，提供了高效的多维数组对象以及大量底层优化的数值计算函数。该实现调用 NumPy 提供的 `dot` 接口。虽然函数调用发生在 Python 层，但实际的数值计算由底层库实现，从而避免了 Python 解释器在循环与算术运算中的参与。

```python
def dot_numpy(a, b):
    # np.dot 在 Python 层仅作为接口存在
    # 实际计算由底层库实现
    return np.dot(a, b)
```

代码 3.2: 基于 NumPy 的一维向量点积函数

实验组 3：基于手写 C 函数的实现 该实现直接使用 C 语言编写点积函数，所有循环与数值运算均在本地机器码层面执行，不存在解释执行、动态类型检查或对象封装开销。该实现可作为理想的本地执行性能基线，对比 Python 解释执行所引入的额外成本。

```c
double dot_c(const double *a, const double *b, size_t n) {
    // 使用原生 double 类型进行累加
    double s = 0.0;

    // 纯 C 语言 for 循环，直接编译为机器指令
    for (size_t i = 0; i < n; i++) {
        s += a[i] * b[i];
    }

    return s;
}
```

代码 3.3: C 语言实现的一维向量点积函数

实验过程和结果 代码 3.4为实验的具体过程。为确保计算规模足够大，提升结果的显著性，实验中向量长度设置为 $N = 5 \times 10^6$。所有向量元素均为双精度浮点数，并通过固定随机种子生成，以保证实验结果的可重复性。

```
# Data preparation
N = 5_000_000       # 向量长度
SEED = 42
random.seed(SEED)

# Python list: 元素为 Python float 对象（双精度浮点数）
a_list = [random.random() for _ in range(N)]
b_list = [random.random() for _ in range(N)]

# NumPy array: 连续内存中的 double 类型（双精度浮点数）
a_np = np.array(a_list, dtype=np.float64)
b_np = np.array(b_list, dtype=np.float64)

# Benchmark helper
def benchmark(func, *args):
    times = []
    start = time.perf_counter()
    func(*args)
    end = time.perf_counter()
    return end - start

# Benchmark
# 纯 Python: 解释器逐元素执行
t_py = benchmark(dot_python, a_list, b_list)

# NumPy: Python 层函数调用，底层 C 实现
t_np = benchmark(dot_numpy, a_np, b_np)

# C 实现：全部在本地机器码中执行
# 此处省略了加载C语言库函数的代码
t_c = benchmark(dot_c, a_np, b_np)

print(f"Dot via Python: {t_py:.4f} s")
print(f"Dot via NumPy : {t_np:.4f} s")
print(f"Dot via C     : {t_c:.4f} s")
```

代码 3.4: 点积运算性能对比实验

实验结果如图 3.1所示。

```
#: python bench-dot.py
Dot via Python : 0.3283 s
Dot via NumPy  : 0.0048 s
Dot via C      : 0.0048 s
```

图 3.1: 点积运算性能测试结果

从结果可以观察到，不同实现方式之间存在数量级上的性能差异。纯 Python 实现的运行时间约为 0.3

秒，明显慢于另外两种实现。相比之下，NumPy 和原生 C 实现的运行时间都约为 0.005 秒，性能提升接近两个数量级，表明 NumPy 在该算例中和原生 C 代码的性能类似。该实验清晰地表明：在数值密集型计算中，使用编译执行语言而非解释执行语言是提升性能的关键手段。

3.1.3 更多探讨

可以把 Python 实现为编译执行吗？从理论和工程实践角度看，答案是“可以，但需要付出代价”。Python 之所以难以像 C 或 Rust 那样采用纯粹的提前编译执行，根本原因在于其高度动态的语言特性，例如动态类型、动态属性或结构体字段、反射机制以及运行时生成代码等（如代码 3.5所示）。这些特性使得编译器在编译期难以确定变量类型、对象布局和调用目标，从而难以稳定地产生高效的本地机器码。

```python
class Person:
    def __init__(self, name):
        self.name = name

p = Person("Alice")

# 动态属性绑定：运行时添加新的属性
p.age = 20
print("Age:", p.age)

# 反射：通过字符串访问和修改属性
print("Name:", getattr(p, "name"))
setattr(p, "name", "Bob")
print("Updated name:", p.name)

# 运行时代码生成：动态创建并执行代码
code = """
def greet(person):
    print("Hello,", person.name)
"""
exec(code)

greet(p)
```

代码 3.5: Python 动态特性示例：动态属性、反射与运行时代码生成

在实际系统中，AOT 编译 Python 的主要思路是通过引入静态类型信息或限制动态行为，将 Python 约化为可静态分析的子集，从而在编译期生成高效的本地机器码。Cython 正是这一思路的典型代表：当代码中提供了足够的类型注解时，Cython 可以在编译期生成接近 C 语言性能的本地代码，但其代价是牺牲部分 Python 的动态性。

```python
def dot_python(double[:] a, double[:] b):
    cdef Py_ssize_t i, n = a.shape[0]
    cdef double s = 0.0
    for i in range(n):
        s += a[i] * b[i]
    return s
```

代码 3.6: Cython 点积函数

介于纯解释执行与提前编译执行之间，还存在一种基于即时编译（Just-In-Time, JIT）的执行方式。JIT 编译器在程序运行过程中对热点代码进行分析，并将频繁执行的代码片段即时编译为本地机器码，从

而在保留语言动态特性的同时降低解释执行带来的运行时开销。常见的相关工具包括 PyPy 和 Numba。其中，PyPy 是一种替代性的 Python 解释器实现，通过内置的追踪式 JIT 在运行时优化热点代码；而 Numba 是面向数值计算的 JIT 编译库，通常通过函数装饰器将 Python 函数即时编译为机器码，特别适合加速基于 NumPy 的计算任务。代码 3.7 和 3.8 展示了一组使用 Numba 的性能测试及其结果。

```python
from numba import njit

N = 5_000_000

def dot_python(a, b):
    s = 0.0
    for i in range(len(a)):
        s += a[i] * b[i]
    return s

@njit
def dot_numba(a, b):
    s = 0.0
    for i in range(len(a)):
        s += a[i] * b[i]
    return s

# 注意：这里如果用list几乎没有优化效果。
a_np = np.array([random.random() for _ in range(N)])
b_np = np.array([random.random() for _ in range(N)])

# Python benchmark
t0 = time.perf_counter()
dot_python(a_list, b_list)
t1 = time.perf_counter()

# Numba 第一次调用（包含 JIT 编译）
t2 = time.perf_counter()
dot_numba(a_np, b_np)
t3 = time.perf_counter()

# Numba 第二次调用（已编译）
t4 = time.perf_counter()
dot_numba(a_np, b_np)
t5 = time.perf_counter()

print("Python time:", t1 - t0)
print("Numba first call (compile + run):", t3 - t2)
print("Numba second call (run only):", t5 - t4)
```

代码 3.7: Numba JIT 实现及其基准测试

```
#: python bench-numba.py
Python time: 1.0850062351673841
Numba first call (compile + run): 0.46062095928937197
Numba second call (run only): 0.006434974260628223
```

代码 3.8: Numba JIT 基准测试结果

3.2 本地代码

所谓本地代码，是指针对具体硬件体系结构（如 x86_64、ARM）生成的机器指令序列。程序运行时，这些指令由处理器直接执行，而无需解释器或虚拟机在运行过程中参与指令调度。本地代码通常由编译器在程序运行前生成，其执行效率主要取决于处理器微架构以及编译器的优化策略。

以代码 3.3 中的 C 语言点积函数为例，编译器会将高级语言中的循环与算术表达式翻译为一段连续的汇编指令。代码 3.9 展示了该函数在 ARM 架构（AArch64）下生成的汇编代码片段。可以看到，函数首先在栈上分配局部变量空间并保存参数，随后通过加载指令（如 `ldr`）读取数组元素，通过浮点运算指令 `fmadd` 完成乘加操作，并利用分支指令（如 `b`、`b.hs`）实现循环控制。整个点积计算过程被映射为一系列显式的内存访问、算术运算和控制流指令，每条指令均由 CPU 直接执行，不再涉及动态类型检查或解释执行带来的额外开销。

```
        sub     sp, sp, #48         ; 为函数栈帧分配 48 字节空间
        str     x0, [sp, #40]       ; 保存参数 a 指针
        str     x1, [sp, #32]       ; 保存参数 b 指针
        str     x2, [sp, #24]       ; 保存参数 n (数组长度)
        movi    d0, #0000000000000000
        str     d0, [sp, #16]       ; s = 0.0 (累加器)
        str     xzr, [sp, #8]       ; i = 0 (循环计数器)
        b       .LBB0_1             ; 跳转到循环判断
.LBB0_1:                            ; 循环入口 (Loop Header)
        ldr     x8, [sp, #8]        ; x8 = i
        ldr     x9, [sp, #24]       ; x9 = n
        subs    x8, x8, x9          ; 比较 i 和 n
        b.hs    .LBB0_4             ; 如果 i >= n 跳出循环
        b       .LBB0_2             ; 否则进入循环体

.LBB0_2:                            ; 循环体: 计算 s += a[i] * b[i]
        ldr     x8, [sp, #40]       ; x8 = a 指针
        ldr     x9, [sp, #8]        ; x9 = i
        ldr     d0, [x8, x9, lsl #3] ; d0 = a[i] (double, 8 字节)

        ldr     x8, [sp, #32]       ; x8 = b 指针
        ldr     x9, [sp, #8]        ; x9 = i
        ldr     d1, [x8, x9, lsl #3] ; d1 = b[i]

        ldr     d2, [sp, #16]       ; d2 = 当前累加值 s
        fmadd   d0, d0, d1, d2      ; d0 = a[i] * b[i] + s (FMA 指令)
        str     d0, [sp, #16]       ; 更新 s
        b       .LBB0_3

.LBB0_3:                            ; 循环尾: i++
        ldr     x8, [sp, #8]        ; x8 = i
        add     x8, x8, #1          ; i = i + 1
        str     x8, [sp, #8]        ; 保存更新后的 i
        b       .LBB0_1             ; 回到循环判断

.LBB0_4:                            ; 循环结束
        ldr     d0, [sp, #16]       ; 返回值 s
        add     sp, sp, #48         ; 恢复栈指针
        ret                         ; 函数返回
```

代码 3.9: 代码 3.3 对应的汇编代码（ARM）

3.3 Python 与本地代码交互

为减少解释器和动态类型系统带来的运行时开销，Python 生态通常将核心计算下沉至本地代码，从而在保持语言易用性的同时获得接近底层语言的执行效率。Python 调用本地代码通常通过外部函数接口（Foreign Function Interface, FFI）实现。FFI 是一种允许程序调用由其它语言实现函数的机制，使高层语言能够在运行过程中执行本地代码，并在调用完成后继续自身的执行流程。

3.3.1 Python 调用本地代码

在 Python 中，调用已经编译好的本地代码库（以 C 为例）通常包括两步：

1) **编译 C 代码为动态库：**使用编译器（如 GCC 或 Clang）将 C 源文件编译为共享库（动态链接库）。

```
clang -O3 -shared -fPIC dot.c -o libdot.so
```

其中：

- `-O3` 开启最高优化等级，提高运行速度；
- `-shared` 生成动态库而非可执行文件；
- `-fPIC` 保证生成位置无关代码，可被动态链接。

2) **在 Python 中加载并调用动态库：** Python 可通过 `ctypes` 模块在运行时加载并调用已编译的本地代码，其基本过程如下：

- 使用 `LoadLibrary` 加载动态库，完成运行时链接；
- 通过函数名获取库中导出的 C 函数符号；
- 显式指定函数的返回类型（`restype`）与参数类型（`argtypes`）；
- 将 Python 对象中可暴露的连续内存区域地址转换为 C 指针并传递给本地函数；
- 本地代码直接在原生机器指令上执行并返回结果。

```
import ctypes

# 加载动态库
lib = ctypes.cdll.LoadLibrary("./libdot.so")

# 设置函数返回类型和参数类型
lib.dot_c.restype = ctypes.c_double
lib.dot_c.argtypes = [ ctypes.POINTER(ctypes.c_double), ctypes.POINTER(ctypes.c_double),
    ctypes.c_int,
]

# 封装为Python函数，或直接在Python代码中调用
def dot(a, b):
    return lib.dot_c(
        a.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
        b.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
        a.size
    )
```

代码 3.10: 静态绑定：使用 ctypes 调用 C 实现的点积函数

代码 3.10 展示了调用本地函数 dot_c 的方法。dot_c 与 C 代码中定义的函数名一一对应，其函数接口及类型信息在构建阶段即已确定。这种调用方式属于静态绑定。在该用例中，本地函数调用被封装为普通的 Python 函数，并在封装层中完成参数转换与类型检查。

在 Python 中，对象的具体类型通常只有在运行时才能确定，因此在调用本地代码时，需要根据数据类型选择合适的函数实现。与静态绑定相对的是动态绑定（Dynamic Binding）。如代码 3.11 所示，在动态绑定方式下，程序可以在运行时根据对象的类型选择不同的点积函数实现，例如 dot_float 或 dot_double。

```python
lib = ctypes.cdll.LoadLibrary("./libdot.so")
def get_dot_func(a: np.ndarray):
    """
    根据数组类型动态选择调用的 C 函数
    """
    if a.dtype == np.int32:
        func_name = "dot_int"
        dot_func.restype = ctypes.c_int
        ptr_type = ctypes.POINTER(ctypes.c_int)
    elif a.dtype == np.float64:
        func_name = "dot_double"
        dot_func.restype = ctypes.c_double
        ptr_type = ctypes.POINTER(ctypes.c_double)
    else:
        raise TypeError("Unsupported dtype")

    dot_func = getattr(lib, func_name)
    dot_func.argtypes = [ptr_type, ptr_type, ctypes.c_size_t]
    return dot_func, ptr_type

def dot(a: np.ndarray, b: np.ndarray):
    """
    Python 封装函数，内部调用对应的 C 实现
    """
    dot_func, ptr_type = get_dot_func(a)

    return dot_func(
        a.ctypes.data_as(ptr_type),
        b.ctypes.data_as(ptr_type),
        a.size
    )
```

代码 3.11: 动态绑定：使用 ctypes 调用 C 实现的点积函数

3.3.2 通过 Python 装饰器交互

虽然用户可以直接通过接口调用本地代码，但这种方式通常需要显式处理函数签名、参数转换以及调用细节，对使用者而言存在一定的工程负担。基于 Python 装饰器的方式提供了另一种更高层次的交互抽象。对于添加了装饰器的函数，Python 拦截并替换原始 Python 函数对象，并按照装饰器的逻辑对该函数的内容进行编译和运行。

代码 3.12 定义了三个 Python 函数，并通过 @kernel 装饰器标记。每个函数描述一个简单的向量计算操作，分别对应逐元素向量乘法、向量加法以及乘加运算。在这种模式下，函数体主要用于表达计算逻辑，而具体的执行方式由装饰器及其背后的 DSL 运行时系统负责处理。

```python
@kernel
def vec_elem_mul(a, b):
    # element-wise multiplication
    return a * b

@kernel
def vec_elem_add(a, b):
    # element-wise addition
    return a + b

@kernel
def vec_elem_fma(a, b, c):
    # element-wise fused multiply-add: a * b + c
    return (a * b) + c
```

代码 3.12: 通过装饰器使用 DSL

装饰器 `@kernel` 的实现如代码 3.13 所示。该装饰器并不直接执行被修饰函数的 Python 代码，而是基于抽象语法树对函数体进行解析与分析，将其视为一段领域专用语言（DSL：Domain-Specific Language）。在函数定义阶段，装饰器首先获取并解析函数源码，提取函数参数与核心计算表达式；随后对表达式对应的 AST 进行遍历与降级（lowering），将高层的算术运算映射为预先实现的本地向量算子。在运行时，装饰器生成的包装函数负责完成参数准备与调度，并将实际计算委托给本地代码执行。

```python
def kernel(func):
    # ------------------------------------------------
    # 第一阶段：解析 Python 抽象语法树（AST）
    # ------------------------------------------------
    src = inspect.getsource(func)   # 获取被修饰函数的源码字符串
    tree = ast.parse(src)   # 将源码解析为 Python 抽象语法树
    func_def = tree.body[0] # 取出函数定义节点
    return_stmt = func_def.body[0]    # 函数体内部只有一条return语句
    expr = return_stmt.value    # 提取赋值语句右侧的表达式 AST
    arg_names = [arg.arg for arg in func_def.args.args] # 提取函数参数名列表

    # ------------------------------------------------
    # 第二阶段：AST 降级（Lowering）；将高层表达式映射为本地算子调用
    # ------------------------------------------------
    def lower(node, env, n):
        """
        递归遍历表达式 AST，将其转换为对本地向量算子的调用。
        参数：
          node : 当前 AST 节点
          env  : 变量名到数组的映射环境
          n    : 向量长度
        """

        # 如果节点是变量名，直接从环境中取出对应数组
        if isinstance(node, ast.Name):
            return env[node.id]

        # 如果节点是二元运算（如加法、乘法）
        if isinstance(node, ast.BinOp):
```

```
            left = lower(node.left, env, n)  # 递归处理左右子表达式
            right = lower(node.right, env, n)
            out = np.empty_like(left) # 为结果分配输出数组

            # 根据运算类型选择对应的本地算子
            if isinstance(node.op, ast.Mult):
                lib.vec_elem_mul(
                    left.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
                    right.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
                    out.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
                    n,
                )
            elif isinstance(node.op, ast.Add):
                lib.vec_elem_add(
                    left.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
                    right.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
                    out.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
                    n,
                )
            else:
                # 当前示例不支持的运算符
                raise NotImplementedError("Unsupported operator")

            return out

        # 不支持的 AST 节点类型
        raise NotImplementedError("Unsupported AST node")

    # ------------------------------------------
    # 第三阶段：运行时包装函数
    # ------------------------------------------
    def wrapper(*args):
        """
        wrapper 是最终返回给用户调用的函数。
        """

        # 将输入参数统一转换为 float64 的 NumPy 数组
        arrays = [np.asarray(a, dtype=np.float64) for a in args]
        n = arrays[0].size  # 假定所有向量长度相同
        env = dict(zip(arg_names, arrays)) # 构造变量名到数组的映射环境

        # 对表达式 AST 进行降级并执行本地计算
        result = lower(expr, env, n)
        return result

    # 返回包装后的函数，替代原始 Python 函数
    return wrapper
```

代码 3.13: 定义装饰器：调用 Native Code

装饰器 `@kernel` 的执行流程可以概括为以下几个步骤：

1) **解析 Python 抽象语法树**：使用 `inspect.getsource(func)` 获取函数源码，并通过 `ast.parse` 将

其转换为抽象语法树。由于 DSL 函数仅包含一条返回表达式（如 `return a * b`），因此只需提取该返回语句对应的表达式节点即可。

2) **定义 AST 节点处理规则**：通过递归函数 `lower` 遍历 AST 节点。对于变量节点（`ast.Name`），直接从运行时环境中获取对应的 NumPy 数组；对于二元运算节点（`ast.BinOp`），递归计算左右子表达式，并调用本地算子完成对应的逐元素运算，例如 `lib.vec_elem_mul` 表示逐元素乘法，`lib.vec_elem_add` 表示逐元素加法。

3) **构造运行时包装函数并返回**：定义 `wrapper` 函数，将传入实参统一转换为 NumPy 数组，并构建变量名到数组的映射环境 `env`。随后调用 `lower` 对表达式进行降级并执行对应的本地计算，最终返回计算结果。

练习

1) 设计针对向量进行 ReLU 的本地代码，并在 Python 中调用。（难度 ⋆⋆）

2) 调研与思考：if 和 for 循环怎么通过装饰器 DSL 实现？（难度 ⋆⋆⋆）

3) 进阶：在第一节课中设计的机器学习程序中，哪些计算密集部分适合使用本地代码实现？尝试对其进行改造，并对比改造前后的性能差异。（难度 ⋆⋆⋆⋆）