4 Python 多线程

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 理解并发程序和并行程序的区别
- ⋆⋆⋆ 理解 Python 多线程的基本执行模型和性能瓶颈
- ⋆⋆⋆ 理解基于本地代码进行并行优化的方法

4.1 并发和并行

4.1.1 基本概念

在现代计算系统中，并发（Concurrency）是提升程序响应性和资源利用率的重要手段。一个并发程序允许多个任务在逻辑上同时推进，而不要求它们在物理时间上真正同时执行。需要区分的是，并发并不等同于并行（Parallelism）。

- **并发**：多个任务在时间上交错执行，强调程序结构
- **并行**：多个任务在多个处理器或核心上同时执行，强调硬件能力

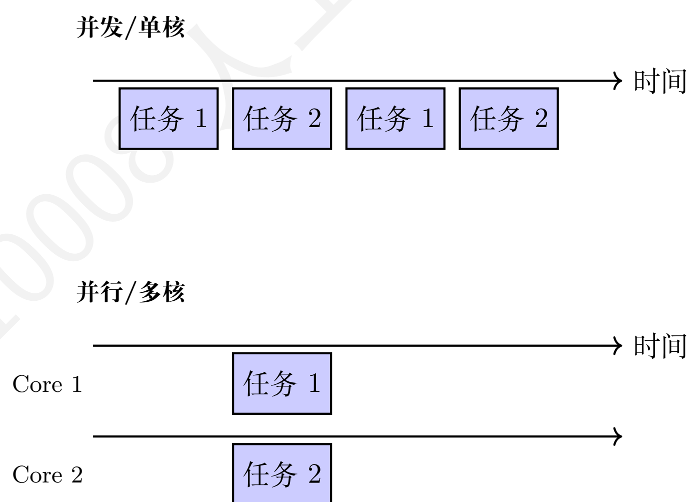

图 4.1: 并发与并行的区别示意图

如图 4.1所示，在单核处理器上，操作系统可以通过时间片轮转的方式调度多个线程，使它们在逻辑上并发执行；而在多核处理器上，多个线程可能真正同时运行。并行可以看做是并发的一种特殊形式。

4.1.2 数据依赖与并发安全

并发的关键在于保证任务在任意切换时程序语义仍然正确。换言之，当在不同任务之间进行调度切换时，程序的最终执行结果不应依赖于具体的调度顺序。当多个任务共享同一份数据，并且至少有一个任务

AIE310008 人工智能软件基础

对该数据进行写操作时，就产生了数据依赖。如果这种依赖关系未被正确处理，不同的任务切换顺序可能导致不同的执行结果，从而破坏程序的确定性。

```python
# 全局共享变量，多个并发任务都会访问和修改
def task():
    global counter
    for _ in range(N):
        # 第一步：从共享变量 counter 中读取当前值
        tmp = counter

        # 此处若发生任务切换，另一个任务可能也读取到相同的 counter

        # 第二步：基于旧值进行计算并写回
        counter = tmp + 1
        # 该“读-改-写”过程不是原子操作，可能导致更新丢失
```

代码 4.1: 累加操作

代码 4.1 给出了一个存在数据依赖的并发程序示例。假设图 4.1 中的线程任务 1 与任务 2 同时执行该函数，并由系统在运行过程中进行时间片轮转调度。在该示例中，变量 counter 是一个全局共享变量，任务 1 与任务 2 都会对其进行读写操作。一次看似简单的累加语句在语义上可被拆解为三个步骤：读取当前值、基于旧值计算新值、将结果写回共享变量。由于这一“读–算–写”过程并非原子操作，系统可能在任意两个步骤之间切换正在执行的任务。

例如，任务 1 在完成对 counter 的读取后尚未写回新值时被切换，任务 2 也可能读取到相同的旧值并完成写回操作。随后任务 1 恢复执行并写回其计算结果，从而覆盖任务 2 的更新。这种由于任务切换顺序不同而导致更新丢失的现象，正是并发程序中典型的数据竞争问题。因此，该示例清楚地表明：当并发任务之间存在未受控的共享可变状态时，程序的最终结果将依赖于具体的调度顺序，从而破坏程序行为的确定性。

代码 4.1中，for 循环体内的代码即使合并为一条 counter += 1，仍然存在数据竞争隐患。如代码 4.2 的字节码所示，该语句在底层会被分解为“读–算–写”三个步骤：其中，LOAD_GLOBAL counter 对应“读”操作，BINARY_ADD 完成“算”操作，而 STORE_GLOBAL counter 则对应“写”操作。由于这三个步骤并非原子执行，在并发环境下可能被其他线程打断，从而引发数据竞争问题。

```
    0 LOAD_GLOBAL              0 (range)       # 读取全局变量 range
    2 LOAD_CONST               1 (10000)       # 常量 10000 入栈
    4 CALL_FUNCTION            1               # 调用 range(10000)
    6 GET_ITER                                 # 获取迭代器
    8 FOR_ITER                 6 (to 22)       # for 循环迭代，取出下一个元素；退出循环则跳转到22
    10 STORE_FAST              0 (_)           # 将当前迭代值存入局部变量 _

    12 LOAD_GLOBAL             1 (counter)     # 【读】读取全局变量 counter 的当前值
    14 LOAD_CONST              2 (1)           # 常量 1 入栈
    16 BINARY_ADD                              # 【算】执行 counter + 1
    18 STORE_GLOBAL            1 (counter)     # 【写】将结果写回 counter

    20 JUMP_ABSOLUTE           4 (to 8)        # 跳回循环开始
    22 LOAD_CONST              0 (None)        # 加载 None
    24 RETURN_VALUE                            # 返回
```

代码 4.2: task 函数内 for 循环改为 counter += 1 后的字节码

为了安全地实现并发，程序必须对共享数据的访问进行显式控制，其中最常见的手段是使用锁（Lock）等同步机制。锁通过在临界区执行期间排他性地授予对共享资源的访问权限，保证任意时刻只有一个任务能够执行涉及共享可变状态的操作，从而避免数据竞争的发生。在前述代码示例中，若对变量 `counter` 的"读–算–写"过程加以锁保护，即可将其视为一个不可分割的原子操作。这样，无论操作系统如何在任务 1 与任务 2 之间切换，都不会出现更新丢失的问题，程序的最终结果也将不再依赖于具体的调度顺序。

4.1.3 数据依赖与并行性

数据依赖不仅影响并发程序的正确性，也直接决定了程序能够利用并行计算资源的程度。当一个计算过程中的某一步依赖于前一步的结果时，该过程在逻辑上就必须顺序执行，从而限制了其并行性；反之，若各个计算步骤之间相互独立，则可以安全地并行执行。

代码 4.3 所示的向量点积计算是典型的存在数据依赖的示例。在该实现中，每次循环迭代都会读取并更新变量 `result`，且其当前值依赖于此前所有迭代的累积结果。此类循环携带依赖使得循环迭代之间无法相互独立执行，因此难以直接并行化；否则将破坏计算结果的正确性。

```python
def dot_product(a, b):
    result = 0
    for i in range(len(a)):
        # result 的当前值依赖于前一次迭代的计算结果
        result += a[i] * b[i]
    return result
```

代码 4.3: 点积运算存在数据依赖关系

相比之下，代码 4.4 所示的逐元素相乘函数，其中每个元素的计算彼此独立。在该实现中，不同迭代之间不存在共享可变状态，也不存在跨迭代的数据依赖。因此，每一次循环迭代都可以被视为一个独立的任务，可以在多核处理器上并行执行。

```python
def vector_multiply(a, b):
    result = [0] * len(a)
    for i in range(len(a)):
        # 每个位置的计算仅依赖 a[i] 和 b[i]
        result[i] = a[i] * b[i]
    return result
```

代码 4.4: 逐元素相乘示例

上述对比表明，并行性的关键并不在于是否使用循环或计算量大小，而在于是否存在未消除的数据依赖。通过消除共享状态、重构计算结构等方法，许多原本顺序的算法都可以被改写为高效且安全的并行形式。

4.2 Python 中的并发机制

在并发程序中，常见的执行单元包括进程、线程和协程，Python 也不例外。其主要关系如图 4.2所示。

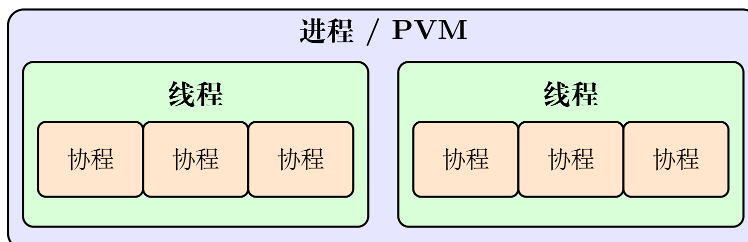

图 4.2: 进程、线程与协程的层次关系示意图

- **进程（Process）**是操作系统进行资源分配和调度的基本单位。不同进程之间拥有独立的地址空间，彼此隔离，通信需要借助进程间通信机制（IPC）。在常见实现中，一个 PVM 往往以单个操作系统进程作为承载单元。
- **线程（Thread）**是进程内的执行单元，是操作系统调度的基本执行实体。同一进程中的多个线程共享地址空间和大部分资源，但各自拥有独立的执行上下文。线程由操作系统负责调度，线程间通信开销较小，但需要显式处理同步与数据竞争问题。
- **协程（Coroutine）**是一种运行在用户态的轻量级执行单元，由程序显式控制调度与切换。协程通常不依赖操作系统线程切换，创建和切换开销极低，适合处理大量并发任务。

本节将简要介绍 Python 中常见的并发机制，并结合其运行时特性说明它们在实际应用中的适用场景。

4.2.1 多线程

多线程通过在同一进程内创建多个执行线程来实现并发。所有线程共享同一地址空间，因此数据共享较为方便，但也更容易引入数据竞争问题。

代码 4.5展示了一段使用两个线程的代码。假设每个线程执行的任务 `task` 是代码 4.1的累加运算，则结果一般等于或小于 2N。如果出现小于 2N 的情况，则是由于数据竞争导致。

```python
import threading

# task: 对共享变量 counter 执行累加操作
t1 = threading.Thread(target=task)  # 创建第一个线程
t2 = threading.Thread(target=task)  # 创建第二个线程

t1.start()  # 启动线程 t1
t2.start()  # 启动线程 t2

t1.join()   # 等待线程 t1 执行结束
t2.join()   # 等待线程 t2 执行结束

# 输出最终结果
print(counter)
```

代码 4.5: 多线程下的累加操作

Python 的多线程机制在 CPython 实现中无法实现真正意义上的并行计算，其主要原因在于全局解释器锁（GIL, Global Interpreter Lock）的存在。GIL 是 CPython 解释器内部采用的一种互斥机制，其核心语义是：在任意时刻，至多只有一个线程能够执行 Python 字节码。因此，Python 的多线程更适用于 I/O 密集型任务，如网络通信、文件读写等，在这些场景下线程可以在等待 I/O 时主动让出执行权，从而提高程序的整体响应性。

GIL 是一种解释器级别的执行互斥机制，而不是语言层面的并发控制原语。结合前面的示例可以看到，即便在存在 GIL 的情况下，多个线程仍然可能在共享变量的“读–算–写”过程中交错执行，从而导致更新丢失等并发错误。在 CPython 中，线程切换可能发生在以下情形：

- 执行一定数量的 Python 字节码指令（时间片耗尽）；
- 执行阻塞操作（如 I/O）；
- 显式调用会让出执行权的函数，如 `time.sleep()`。

```python
def task():
    global counter
    for _ in range(10000):
        tmp = counter
        time.sleep(0.00001)
        counter = tmp + 1
```

代码 4.6: 通过 sleep 触发线程切换

代码 4.6 在累加过程中引入了 `time.sleep()` 调用，该操作恰好位于对共享变量 `counter` 的读操作与写操作之间。在多线程环境下执行 `task` 函数时，`time.sleep()` 会主动释放 GIL，从而触发线程切换，使得其他线程有机会在当前线程完成写回之前介入执行，进而导致数据竞争的发生。

4.2.2 锁

如前所述，尽管 GIL 保证了同一时刻仅有一个线程执行字节码，但它并不能保证多条语句或由多条字节码构成的复合操作具有原子性。对于涉及共享变量读写的代码片段，这类可能被多个线程同时访问的区域称为临界区（critical section）。若临界区中的操作不是原子执行的，则在执行过程中可能发生线程切换，从而导致数据竞争问题。

为了解决这一问题，Python 提供了显式的同步机制，例如 `threading.Lock`，用于对临界区进行互斥保护，从而确保其中操作的原子性：

```python
import threading
lock = threading.Lock()
counter = 0

def task():
    global counter
    for _ in range(10000):
        with lock:  # 加锁，保证更新原子性
            counter += 1
```

代码 4.7: 使用 Lock 保护共享变量

在上述代码中：

- `lock = threading.Lock()` 创建一个互斥锁；

- with lock：在进入代码块时自动获取锁，在退出时释放锁；
- 任意时刻仅有一个线程可以持有该锁，从而保证临界区内操作的互斥执行；
- 因此，即使线程发生切换，counter += 1 也不会被中断，能够正确更新共享变量。

4.2.3 多进程

多进程通过创建多个独立的操作系统进程来实现并发执行。每个进程都拥有独立的 Python 解释器和内存空间，因此不存在 GIL 的限制，能够在多核处理器上实现真正的并行计算。然而，由于进程之间默认不共享内存，进程间的数据交换需要借助管道、队列或共享内存等进程间通信（IPC）机制，其开销相对较高，编程复杂度也更大。因此，多进程通常适用于计算量较大、任务粒度较粗的 CPU 密集型任务。

代码 4.8 展示了一个简单的多进程示例，其中创建了两个子进程分别执行累加任务。需要注意的是，传入子进程的 counter 变量在进程间是以副本形式存在的，各个进程对其修改互不影响。因此，即使每个子进程内部均执行了 N 次累加操作，主进程中的 counter 仍保持初始值不变，最终输出结果为 0。

```python
from multiprocessing import Process  # 导入多进程模块

N = 1000
counter = 0  # 主进程中的全局变量

def task(counter):
    # 注意：这里的 counter 是从主进程拷贝来的副本（值传递）
    for _ in range(N):
        counter += 1  # 只修改当前子进程中的局部变量

if __name__ == "__main__":
    # 创建两个进程
    processes = [
        Process(target=task, args=(counter,))
        for _ in range(2)
    ]

    for p in processes:
        p.start()  # 启动子进程（各自拥有独立内存空间）

    for p in processes:
        p.join()   # 等待子进程执行结束

    # 主进程中的 counter 未被修改
    print("multiprocessing:", counter)  # 仍然为 0
```

代码 4.8: 多进程下的累加操作

4.2.4 协程

协程是一种运行在用户态的轻量级执行单元，通常由运行时系统（如 asyncio）负责调度。与线程不同，协程的切换是协作式的，只有在显式的挂起点（如 await）处才会发生任务切换。因此，协程在单线程中即可实现高并发执行，但无法利用多核实现真正的并行计算。

代码 4.9 展示了一个基于协程的累加示例，其中创建了两个协程任务并发执行。与多线程类似，协程之间共享同一地址空间中的变量 counter。在不引入挂起操作的情况下，各协程对 counter 的更新按顺

序执行，最终结果为 2N。然而，如果在累加过程中引入 `await` 操作，则当前协程会在执行过程中主动让出执行权，使得其他协程有机会在“读–改–写”序列尚未完成时介入执行，从而发生执行交错并引发数据竞争。在这种情况下，多个协程可能基于相同的旧值进行更新，导致更新丢失，最终结果可能退化为 N。

```python
import asyncio

N = 1000
counter = 0

async def task():
    global counter
    for _ in range(N):
        tmp = counter
        # await asyncio.sleep(0)  # 主动让出执行权
        counter = tmp + 1

async def runner():
    tasks = [asyncio.create_task(task()) for _ in range(2)]
    await asyncio.gather(*tasks)

asyncio.run(runner())  # 启动事件循环并执行协程
print("asyncio:", counter)
```

代码 4.9: 协程下的累加操作

AIE310008 人工智能的软件基础

4.3 在本地代码中并行

通过前面的讨论可以看到，在 CPython 实现中，多线程由于 GIL 的限制难以实现真正的计算并行。因此，通常需要借助多进程来实现并行计算。然而，进程间的数据交换需要依赖进程间通信机制，其实现相对复杂且开销较高。

本节将以矩阵乘法为例介绍如何在本地代码中实现并行计算。给定矩阵 $A \in \mathbb{R}^{M\times K}$ 和 $B \in \mathbb{R}^{K\times N}$，其乘积 $C = A \cdot B \in \mathbb{R}^{M\times N}$ 定义为：

$$C_{ij} = \sum_{k=0}^{K-1} A_{ik} \cdot B_{kj}.$$

在 Python 中，最直接的实现是三重循环：

```
def matmul(A, B, M, K, N):
    C = [[0.0]*N for _ in range(M)]
    for i in range(M):
        for j in range(N):
            s = 0.0
            for k in range(K):
                s += A[i][k] * B[k][j]
            C[i][j] = s
    return C
```

代码 4.10: Python 矩阵乘法

4.3.1 矩阵乘法分块思想

在矩阵乘法中，C_{ij} 的计算依赖于 $A_{i,:}$ 和 $B_{:,j}$，但不同的 C_{ij} 之间是相互独立的。利用这一特点，可以将矩阵划分为若干子块（block），并以块为单位进行计算，从而提高数据局部性并便于并行化。

具体而言，将矩阵按一定的块大小划分后，矩阵乘法可以表示为块级别的运算：

$$C_{ij}^{(block)} = \sum_{k} A_{ik}^{(block)} \cdot B_{kj}^{(block)}.$$

其中，每个 C 的子块由对应行块与列块之间的多个子块乘积累加得到，而非单次块乘即可完成。

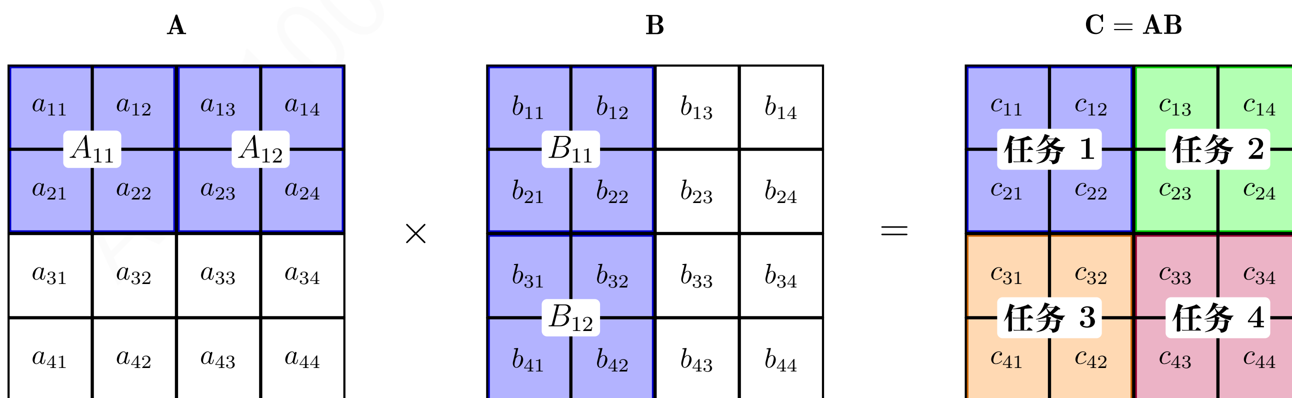

图 4.3: 分块矩阵乘法

如图 4.3所示，矩阵 C 被分为四个子块，每个子块可以独立计算，从而实现并行计算。其中每个子块又可以分解为多个部分进行累加，例如任务 1 对应的 C_{11} 是由两部分计算组成：$A_{11}B_{11}$ 和 $A_{12}B_{21}$，在内层循环中依次累加得到最终结果。这种分块加累加的方式既保证了计算正确性，又有利于充分利用缓存和并行化。

4.3.2 分块矩阵乘法 C 实现

下面给出一个最小可运行的 C 语言分块矩阵乘法实现，利用 OpenMP 对外层循环进行多线程并行计算，从而自动加速矩阵运算。OpenMP（Open Multi-Processing）是一种在 C/C++ 中实现多线程并行计算的标准，它通过简单的编译指令和库函数，就可以让循环或代码块在多个线程上同时执行，提高 CPU 多核利用率。

```c
#include <stddef.h>
#include <omp.h>

// 分块矩阵乘法函数
// A: M×K 矩阵
// B: K×N 矩阵
// C: M×N 矩阵
// 按行优先存储
// M, K, N: 矩阵维度
// BS: 块大小（Block Size）
void matmul_blocked(const double* A, const double* B, double* C,
                    int M, int K, int N, int BS) {

    omp_set_num_threads(4); // 手动设置并发线程数

    #pragma omp parallel for collapse(2) schedule(static)
    // 遍历 C 的块（输出块）
    for (int ii = 0; ii < M; ii += BS) {      // 行块
        for (int jj = 0; jj < N; jj += BS) {  // 列块
            for (int kk = 0; kk < K; kk += BS) {  // 累加维度块

                // 处理边界（防止越界）
                int i_max = (ii + BS < M) ? ii + BS : M;
                int j_max = (jj + BS < N) ? jj + BS : N;
                int k_max = (kk + BS < K) ? kk + BS : K;

                for (int i = ii; i < i_max; i++) {
                    for (int j = jj; j < j_max; j++) {
                        double sum = C[i*N + j];
                        for (int k = kk; k < k_max; k++) {
                            sum += A[i*K + k] * B[k*N + j];
                        }
                        C[i*N + j] = sum;
                    }
                }

            }
        }
    }
}
```

代码 4.11: C 分块矩阵乘法

上述代码中，OpenMP 语句的含义如下：

- `omp_set_num_threads(n)`：在代码中手动指定线程数。也可以通过环境变量 `OMP_NUM_THREADS` 设

置，例如 `export OMP_NUM_THREADS=4`。如果不设置并行度，OpenMP 会自动使用系统默认的线程数，这通常是 CPU 的物理或逻辑核心数。

- `#pragma omp parallel for`：表示紧随其后的循环将被多线程并行执行，每个线程负责循环的一部分迭代。
- `collapse(2)`：将外层两层循环（例如 ii 和 jj 控制的行/列子块循环）合并成一个大循环，以便线程可以均匀分配任务，提高负载均衡。
- `schedule(static)`：表示迭代按照固定块平均分配给各线程，适合每次迭代耗时相近的情况，例如矩阵分块乘法。

编译上述代码时，需要打开 `-fopenmp` 选项：

```
clang -O3 -shared -fPIC -fopenmp $1 -o $2
```

4.3.3 性能对比实验

下面展示如何在 Python 中调用之前 C 语言实现的本地分块矩阵乘法函数 `matmul_blocked`，将 NumPy 矩阵传入本地函数进行高性能计算，并将结果返回到 Python。实验中，我们对比了三种实现的性能：

- 纯 Python 实现：直接使用嵌套循环计算矩阵乘法，作为基准。
- 本地 C 分块矩阵乘法 + OpenMP 并行：利用我们之前实现的 `matmul_blocked` 函数，通过 OpenMP 并行化外层循环，实现多线程加速。
- NumPy 矩阵乘法（BLAS 库）：调用 NumPy 的矩阵乘法操作（底层使用高性能 BLAS），作为参考。

通过该实验可以直观地观察不同实现的性能差异，从而验证本地分块矩阵乘法和多线程加速的效果。

```python
M, K, N = 256, 256, 256
BS = 64

# ------------------------------------------------------
# Prepare data
# ------------------------------------------------------
A_np = np.random.rand(M, K).astype(np.float64)
B_np = np.random.rand(K, N).astype(np.float64)
C_np = np.zeros((M, N), dtype=np.float64)

# Python list
A_py = A_np.tolist()
B_py = B_np.tolist()

# ------------------------------------------------------
# 1. Pure Python
# ------------------------------------------------------
def matmul(A, B, M, K, N):
    C = [[0.0] * N for _ in range(M)]
    for i in range(M):
        for j in range(N):
            s = 0.0
```

```python
            for k in range(K):
                s += A[i][k] * B[k][j]
            C[i][j] = s
    return C
start = time.time()
C_py = matmul(A_py, B_py, M, K, N)
t_naive = time.time() - start
print(f"Python time:     {t_naive:.3f}s")

# ------------------------------------------------------
# 2. native blocked (C + OpenMP)
# ------------------------------------------------------
lib = ctypes.CDLL("./libmatmul.so")

lib.matmul_blocked.argtypes = [
    ctypes.POINTER(ctypes.c_double),  # A
    ctypes.POINTER(ctypes.c_double),  # B
    ctypes.POINTER(ctypes.c_double),  # C
    ctypes.c_int,  # M
    ctypes.c_int,  # K
    ctypes.c_int,  # N
    ctypes.c_int,  # BS
]

start = time.time()
lib.matmul_blocked(
    A_np.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
    B_np.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
    C_np.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),
    M, K, N, BS,
)
t_native = time.time() - start
print(f"Blocked C (OpenMP):    {t_native:.3f}s")

# ------------------------------------------------------
# 3. NumPy (BLAS)
# ------------------------------------------------------
start = time.time()
C_ref = A_np @ B_np
t_numpy = time.time() - start
print(f"NumPy (BLAS):          {t_numpy:.3f}s")
```

代码 4.12: Python 调用本地分块矩阵乘法

实验结果参考如下：

```
#: python bench-matmul.py
Python time:     1.331s
Blocked C (OpenMP):    0.005s
NumPy (BLAS):          0.008s
```

代码 4.13: Python 调用本地分块矩阵乘法

练习

1) 针对矩阵乘法实验，分析不同参数设置下的优化效果，并总结性能规律。

   - 线程数
   - 矩阵规模
   - 分块大小

2) 针对点积运算，实现基于 OpenMP 并行化版本，验证计算正确性，并评估性能提升效果。

AIE310008 人工智能的软件基础