MLSYS前置知识

一、CPU 结构

CPU（中央处理器）是计算机的核心，主要由以下部分构成：

- **控制单元（CU）**：负责取指令、解码指令，协调 CPU 各部件工作（如控制指令执行顺序、数据流向）。
- **算术逻辑单元（ALU）**：执行算术运算（加 / 减 / 乘 / 除）和逻辑运算（与 / 或 / 非 / 比较）。
- **寄存器组（Registers）**：CPU 内部的高速存储单元，用于临时存放指令、数据和地址（如程序计数器 PC、累加器 ACC），容量极小（通常几十到几百个，每个几十字节）。
- **缓存（Cache）**：位于 CPU 与主存之间的高速存储，分为 L1、L2、L3 三级（详见下节）。
- **总线接口单元（BIU）**：负责 CPU 与内存、外设之间的数据交换（通过地址总线、数据总线、控制总线）。

现代 CPU 还包含**超标量执行单元**（多个 ALU 并行处理）、**流水线控制器**（指令分阶段重叠执行）、**分支预测器**（优化条件跳转效率）等模块，以提升性能。

二、不同内存层级的特点

内存层级从 CPU 核心向外延伸，遵循 “速度越快→容量越小→成本越高” 的规律，核心是利用**局部性原理**（时间局部性：近期访问的数据可能再次访问；空间局部性：相邻数据可能被连续访问）提升效率。

层级	位置	速度（访问延迟）	容量典型值	作用
寄存器（Register）	CPU 核心内部	~1ns	几十 KB	存放当前指令的操作数、临时结果，直接供 ALU 使用，是 CPU 能直接访问的最快存储。
L1 Cache	CPU 核心内部	~2-4ns	几十到几百 KB	分为数据缓存（L1d）和指令缓存（L1i），缓存最近使用的数据和指令。
L2 Cache	CPU 核心内部（或附近）	~10-20ns	几百 KB 到几 MB	缓存 L1 未命中的数据，速度比 L1 慢，但容量更大。
L3 Cache	多个 CPU 核心共享	~30-100ns	几 MB 到几十 MB	所有核心共享的缓存，缓存 L2 未命中的数据，平衡容量和速度。

层级	位置	速度（访问延迟）	容量典型值	作用
主存（Memory）	主板上（DRAM）	~100-300ns	几 GB 到几十 GB	长期存放程序和数据，CPU 需通过缓存间接访问（直接访问速度太慢）。
外存（硬盘 / SSD）	外部存储设备	~10ms（机械盘）/ ~100μs（SSD）	几百 GB 到几 TB	永久存储数据，速度最慢，用于长期保存不活跃数据。

三、计算 `a[0:31] = b[0:31] + d[0:31]` 的过程

假设数组 `a, b, d` 均为 32 元素的整数数组，CPU 执行该操作的步骤如下（简化版）：

1. **指令加载**：CPU 从内存读取 “循环计算数组和” 的指令，经 L1 指令缓存加载到指令寄存器，由控制单元解码。
2. 数据预取：因空间局部性，CPU 通过**预取器**将 `b[0:31]` 和 `d[0:31]` 从内存加载到 L3→L2→L1 数据缓存。
3. 循环计算

   （以单元素为例，实际可能用 SIMD 指令并行处理）：
   - 从 L1 缓存读取 `b[i]` 和 `d[i]`，加载到通用寄存器（如 `reg1` 和 `reg2`）。
   - ALU 对 `reg1` 和 `reg2` 执行加法，结果存入目标寄存器 `reg3`。
   - 将 `reg3` 的值写回 L1 缓存的 `a[i]` 位置。
4. **写回内存**：循环结束后，L1 缓存中的 `a[0:31]` 数据逐步写回主存（通过缓存一致性协议确保数据同步）。

优化：现代 CPU 会通过**SIMD 指令**（如 x86 的 AVX2）一次性处理 16/32 字节数据（例如一次计算 8 个 int32 元素），大幅提升效率。

四、进程和线程，及多进程 / 多线程的区别

1. 基本概念

- **进程**：操作系统资源分配的基本单位（拥有独立的内存空间、文件描述符等），是程序的一次执行实例。
- **线程**：进程内的执行单元（共享进程的内存空间），是 CPU 调度的基本单位，一个进程可包含多个线程。

2. 多进程 vs 多线程（单核 / 多核）

维度	多进程	多线程（单核）	多线程（多核）
执行方式	多个进程独立运行（地址空间隔离）	线程通过时间片切换并发执行	线程在不同核心并行执行
资源共享	需通过 IPC（管道 / 共享内存 / 消息队列）	直接共享进程内存（变量、堆等）	直接共享进程内存
开销	大（创建 / 销毁需分配 / 释放资源）	小（仅需保存线程上下文）	小
安全性	高（隔离性强，一个崩溃不影响其他）	低（共享资源易引发竞态条件）	低
通信效率	低（依赖 IPC 机制）	高（直接访问共享内存）	高

3. 优缺点

- **多进程**：
  - 优点：稳定性高（隔离性）、可利用多核并行、适合计算密集型任务。
  - 缺点：资源开销大、通信复杂、不适合频繁创建销毁。
- **多线程**：
  - 优点：开销小、通信高效、适合 I/O 密集型任务（如网络请求）。
  - 缺点：需处理线程同步（锁 / 信号量）、一个线程崩溃可能导致整个进程崩溃。

五、用 C++ 多线程 / 多进程加速数组计算

以 `a[i] = b[i] + d[i]`（32 元素）为例，可将数组分片并行处理。

1. 多线程实现（用`std::thread`）

```
#include <thread>
#include <vector>

void add_chunk(const std::vector<int>& b, const std::vector<int>& d,
               std::vector<int>& a, int start, int end) {
    for (int i = start; i < end; ++i) {
        a[i] = b[i] + d[i];
    }
}

int main() {
    const int n = 32;
    std::vector<int> b(n, 1), d(n, 2), a(n);

    // 分成2个线程处理（0-15和16-31）
```

```
    std::thread t1(add_chunk, std::ref(b), std::ref(d), std::ref(a), 0, 16);
    std::thread t2(add_chunk, std::ref(b), std::ref(d), std::ref(a), 16, 32);

    t1.join();
    t2.join();
    return 0;
}
```

2. 多进程实现（用`std::async`，本质是进程池）

```
#include <future>
#include <vector>

void add_chunk(const std::vector<int>& b, const std::vector<int>& d,
               std::vector<int>& a, int start, int end) {
    for (int i = start; i < end; ++i) {
        a[i] = b[i] + d[i];
    }
}

int main() {
    const int n = 32;
    std::vector<int> b(n, 1), d(n, 2), a(n);

    // 用async创建异步任务（默认可能用新进程）
    auto f1 = std::async(std::launch::async, add_chunk,
                         std::ref(b), std::ref(d), std::ref(a), 0, 16);
    auto f2 = std::async(std::launch::async, add_chunk,
                         std::ref(b), std::ref(d), std::ref(a), 16, 32);

    f1.wait();
    f2.wait();
    return 0;
}
```

六、并行与并发

- **并发（Concurrency）**：多个任务在**同一时间段内交替执行**（宏观上同时，微观上串行），如单核 CPU 通过时间片切换处理多个线程。
- **并行（Parallelism）**：多个任务在**同一时刻真正同时执行**，需多核 CPU 支持（每个核心处理一个任务）。

举例：并发是“一个厨师同时处理多个订单（交替切菜、炒菜）”，并行是“多个厨师同时处理不同订单”。

七、pybind11/nanobind 混合编程（简单示例）

pybind11/nanobind 用于将 C++ 函数 / 类暴露给 Python，实现跨语言调用。

1. 安装依赖

```
pip install pybind11  # 或 nanobind
```

2. C++ 代码

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>  // 支持vector与Python列表转换

std::vector<int> array_add(const std::vector<int>& b, const std::vector<int>& d) {
    std::vector<int> a(b.size());
    for (size_t i = 0; i < b.size(); ++i) {
        a[i] = b[i] + d[i];
    }
    return a;
}

PYBIND11_MODULE(my_module, m) {  // 模块名my_module
    m.def("array_add", &array_add, "Add two arrays element-wise");
}
```

3. 编译（`setup.py`）

```
from setuptools import setup
from pybind11.setup_helpers import Pybind11Extension

ext_modules = [
    Pybind11Extension("my_module", ["array_add.cpp"]),
]

setup(name="my_module", ext_modules=ext_modules)
```

编译后生成`.so`文件，Python 中调用：

```
import my_module
b = [1, 2, 3]
d = [4, 5, 6]
print(my_module.array_add(b, d))  # 输出 [5,7,9]
```

八、Torch 矩阵与 C++ 数组的转换

PyTorch 的`torch.Tensor`在 C++ 中对应`at::Tensor`（需依赖 LibTorch），可通过指针直接访问底层数据。

步骤：

1. **Python 端**：将 Tensor 转为 C++ 可访问的格式（确保在 CPU 上）：

```python
import torch
x = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)  # 假设是float32矩阵
```

2. **C++ 端**（用 LibTorch）：

```cpp
#include <torch/torch.h>
#include <vector>

void process_tensor(const at::Tensor& x) {
    // 检查设备（必须在CPU）和数据类型
    AT_ASSERT(x.device().is_cpu());
    AT_ASSERT(x.scalar_type() == at::kFloat);

    // 获取维度信息
    int rows = x.size(0);
    int cols = x.size(1);

    // 获取原始数据指针（转为float*）
    float* data_ptr = x.data_ptr<float>();

    // 转为C++二维数组（或vector）
    std::vector<std::vector<float>> cpp_array(rows, std::vector<float>(cols));
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cpp_array[i][j] = data_ptr[i * cols + j];  // 按行优先访问
        }
    }
}
```

通过 pybind11 绑定后，即可在 Python 中传递 Tensor 给 C++ 函数。

九、Python GIL（全局解释器锁）

- **定义**：CPython 解释器中的一把互斥锁，确保同一时间只有一个线程执行 Python 字节码。
- **原因**：CPython 的内存管理（如引用计数）不是线程安全的，GIL 通过序列化线程执行避免竞态条件。
- **影响**：多核 CPU 上，Python 多线程无法真正并行执行 CPU 密集型任务（仍受限于单线程），但对 I/O 密集型任务影响较小（I/O 时线程会释放 GIL）。

缓解方法：

1. 用**多进程**（`multiprocessing`）：每个进程有独立 GIL，可利用多核。
2. 用**C 扩展**：在 C/C++ 代码中释放 GIL（通过`Py_BEGIN_ALLOW_THREADS`），执行 CPU 密集型操作。
3. 用**异步编程**（`asyncio`）：单线程内通过事件循环处理并发 I/O。

十、计算机的三种并行模式

1. **流水线（Pipelining）**：
   将指令执行分解为多个阶段（如取指、解码、执行、访存、写回），不同阶段可并行处理不同指令（类似工厂流水线）。例如，CPU 执行第 1 条指令的 “执行” 阶段时，可同时对第 2 条指令 “解码”，第 3 条指令 “取指”。
2. **超标量（Superscalar）**：
   CPU 内置多个独立的执行单元（如多个 ALU），可在一个时钟周期内发射多条不相关的指令并行执行（如同时执行加法和乘法）。需通过**指令调度器**识别无依赖的指令。
3. **多核（Multi-core）**：
   一个 CPU 芯片集成多个独立核心（每个核心类似一个小 CPU），每个核心可独立执行线程，实现真正的并行计算。例如，4 核 CPU 可同时执行 4 个线程。