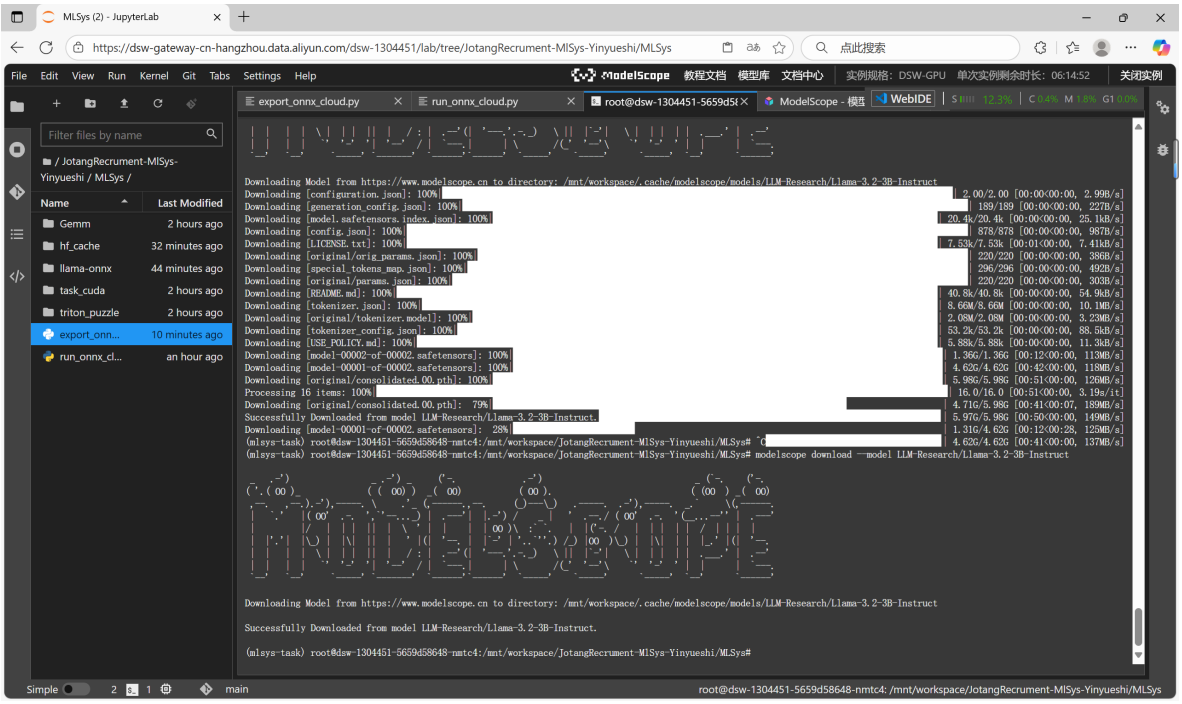
Part 1

问题

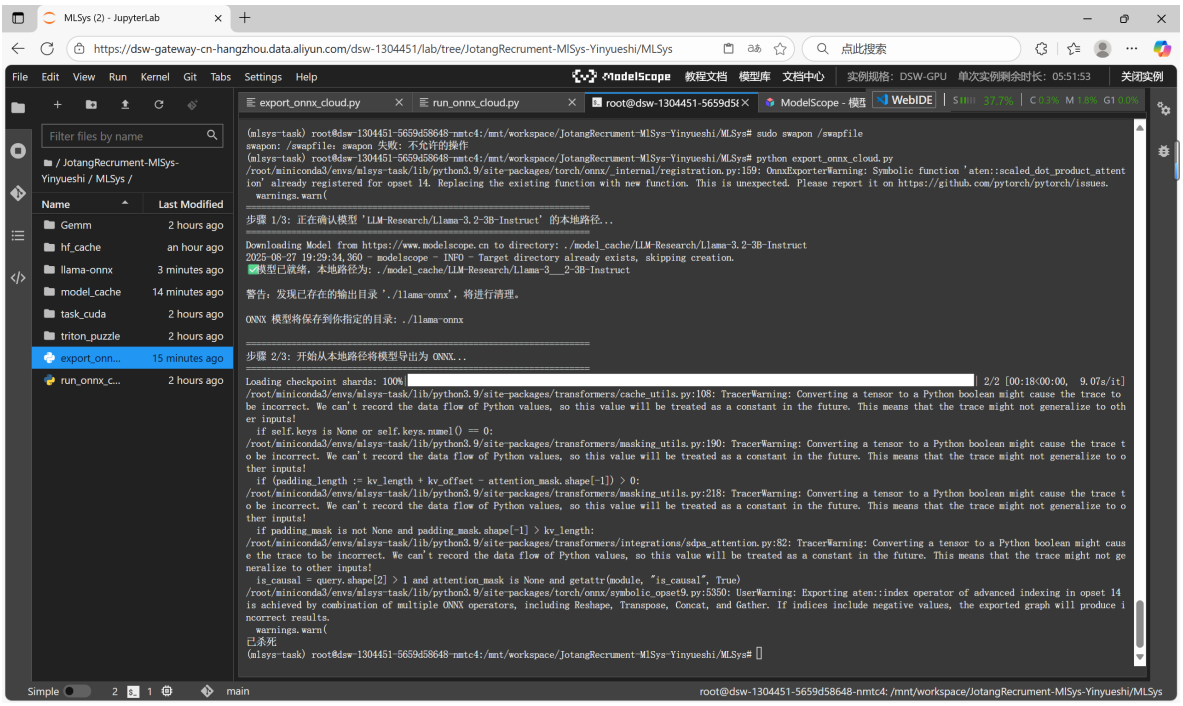
1. **将 llama 3b 模型导出为onnx格式并用 onnx runtime 进行部署**

   尝试了本地部署但是由于计算机硬件的问题出现了内存不够用的情况出出现了OOM Killer错误，部署失败所以转向使用魔塔社区的云服务器

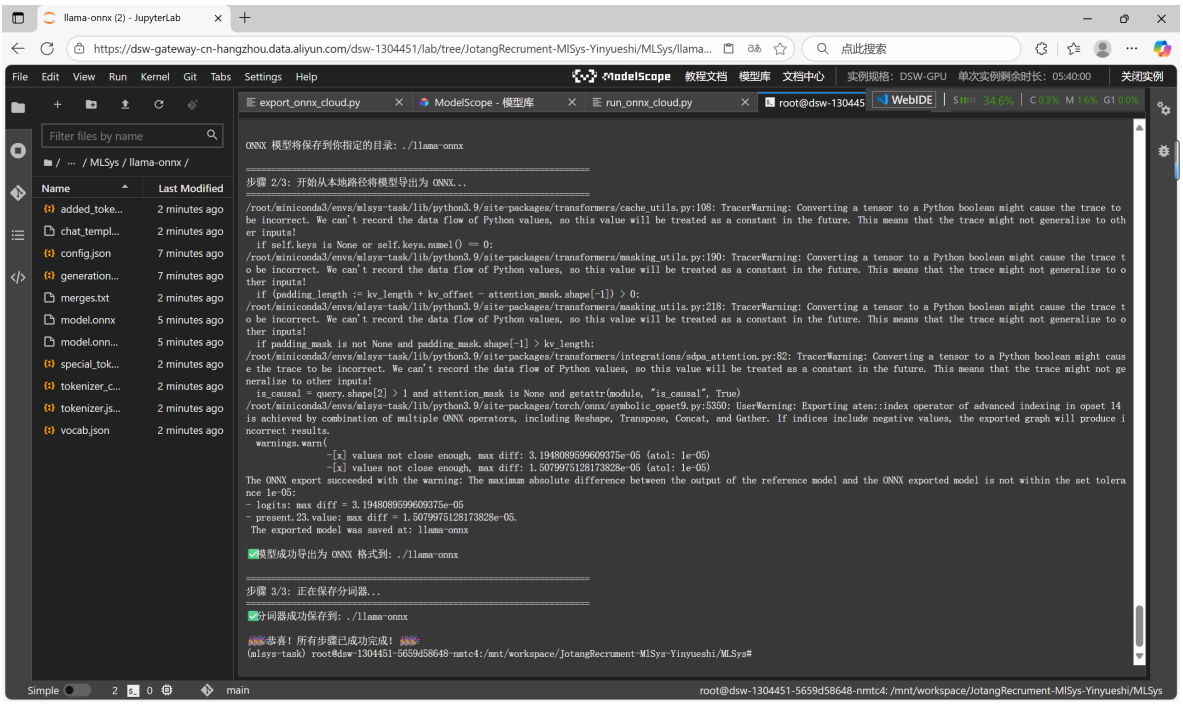
   开发环境和本地一致



但是在导出为ONNX格式的时候同样出现了OOM Killer错误，在尝试导出 3B 模型时，遇到了由于 DSW 实例内存限制导致的 OOM Killer 问题。我首先尝试通过创建 Swap 文件来增加虚拟内存，但发现 DSW 的容器环境出于安全原因禁止了 swapon 操作。这让我深刻理解了云原生环境下的权限限制。因此，我调整了策略，选择了一个更轻量级的 Qwen1.5-1.8B 模型，最终成功完成了模型的导出和部署。这个过程让我学会了在资源受限的环境下，如何做出合理的技术选型。

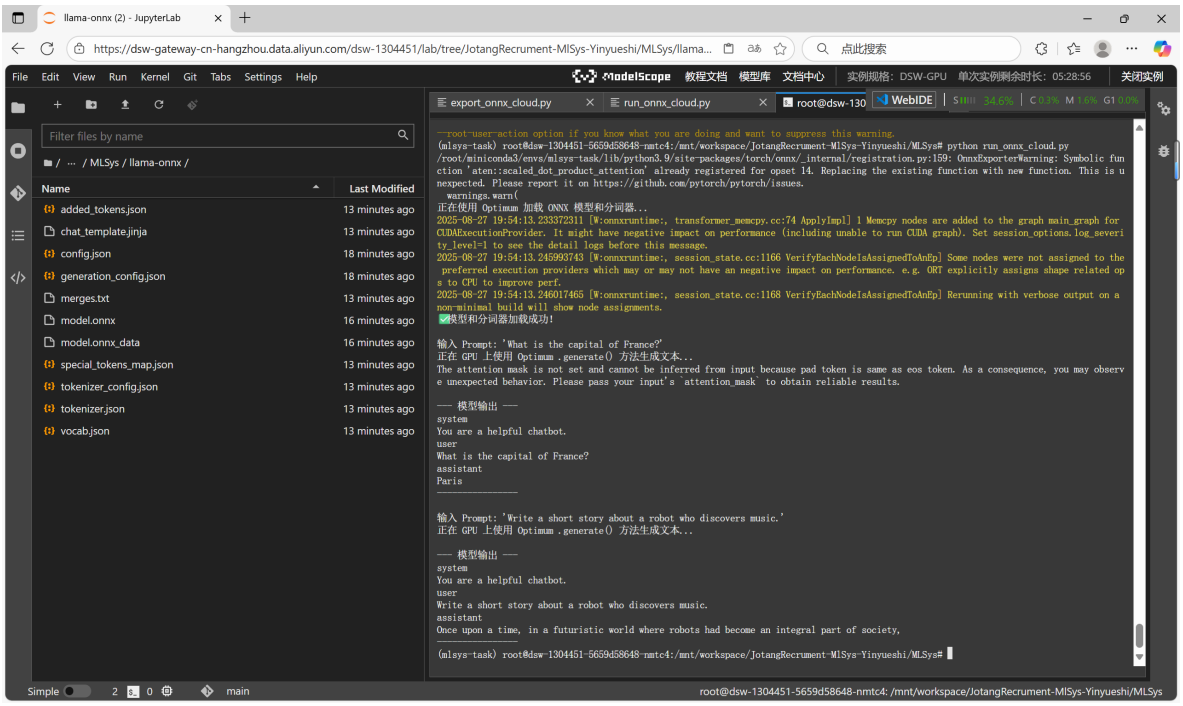


接下来转向部署**Qwen/Qwen1.5-1.8B-Chat**



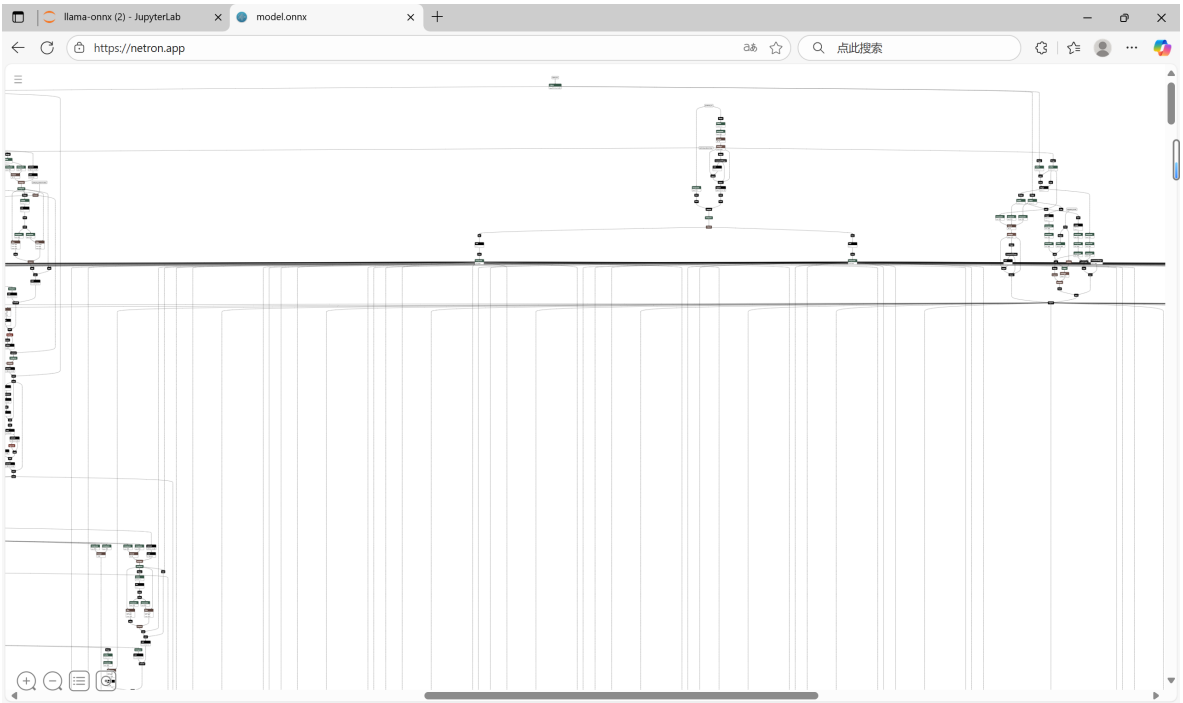
导出成功

接下来开始部署



部署成功

在线查看onnx文件，



2. **profiling算子融合所带来的性能受益**

   算子融合是一种常见的提高神经网络模型执行效率的方法。这种融合的基本思想与优化编译器所做的传统循环融合相同，它们会带来：1）消除不必要的中间结果实例化，2）减少不必要的输入扫描；3）实现其他优化机会。

   首先回顾一下计算图，计算图是对算子执行过程形象的表示，假设 C={N,E,I,O} 为一个计算的计算表达，那么可以有：

   - 计算图表示为由一个节点 N（Node），边集（Edge），输入边（Input），输出边（Output）组成的四元组

- 计算图是一个有向联通无环图，其中的节点也被称为算子(Operator)
- 算子必定有边相连，输入边，输出边不为空
- 计算图中可以有重边（两个算子之间可以由两条边相连）

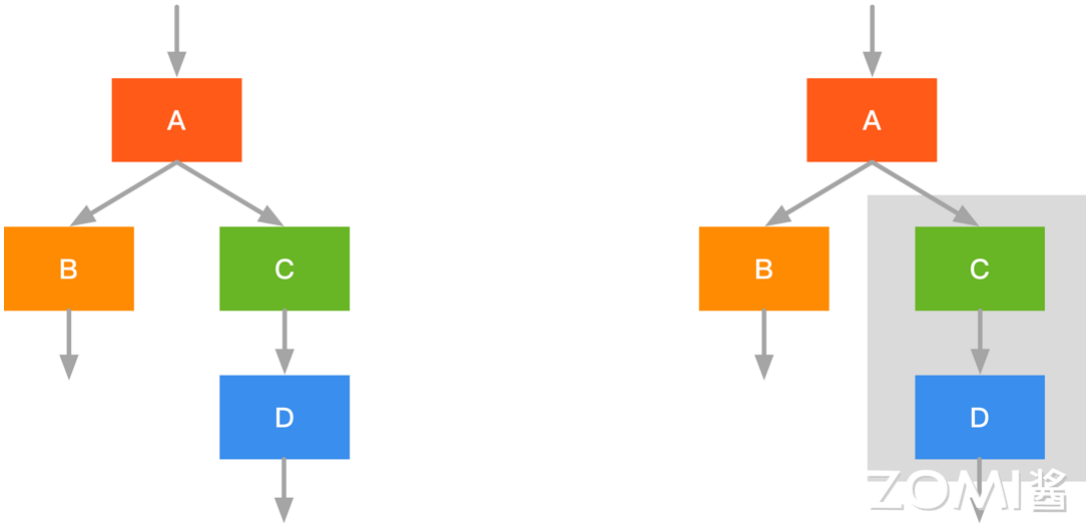
于是当我们遇到一个具体的算子实例时，我们可以在其对应的计算图上做等价的融合优化操作，这样的抽象使我们能够更加专心于逻辑上的处理而不用在意具体的细节。

为什么要算子融合呢？这样有什么好处呢？ 融合算子出现主要解决模型训练过程中的读入数据量，同时，减少中间结果的写回操作，降低访存操作。它主要想解决我们遇到的内存墙和并行强墙问题：

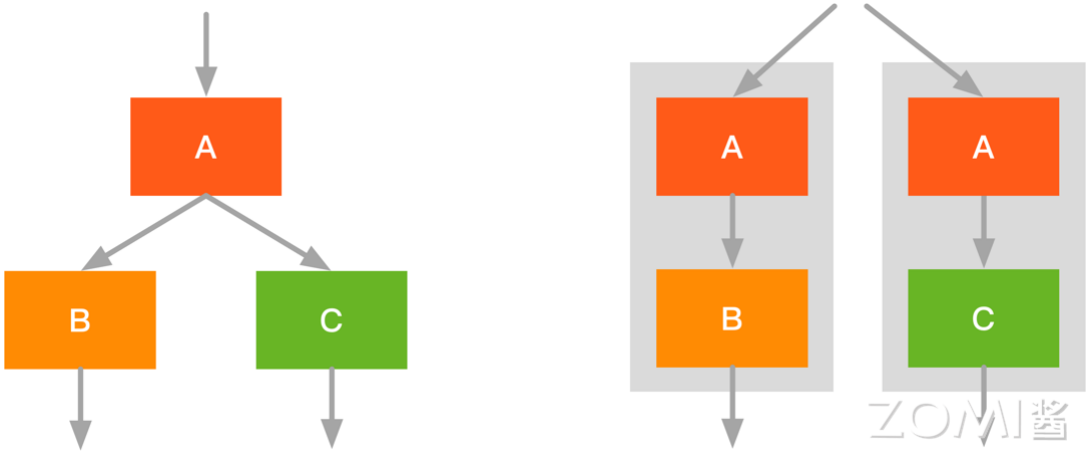
- **内存墙**：主要是访存瓶颈引起。算子融合主要通过对计算图上存在数据依赖的“生产者-消费者”算子进行融合，从而提升中间 Tensor 数据的访存局部性，以此来解决内存墙问题。这种融合技术也统称为“Buffer 融合”。在很长一段时间，Buffer 融合一直是算子融合的主流技术。早期的 AI 框架，主要通过手工方式实现固定 Pattern 的 Buffer 融合。
- **并行墙**：主要是由于芯片多核增加与单算子多核并行度不匹配引起。可以将计算图中的算子节点进行并行编排，从而提升整体计算并行度。特别是对于网络中存在可并行的分支节点，这种方式可以获得较好的并行加速效果。

算子的融合方式非常多，我们首先可以观察几个简单的例子，不同的算子融合有着不同的算子开销，也有着不同的内存访问效率提升。

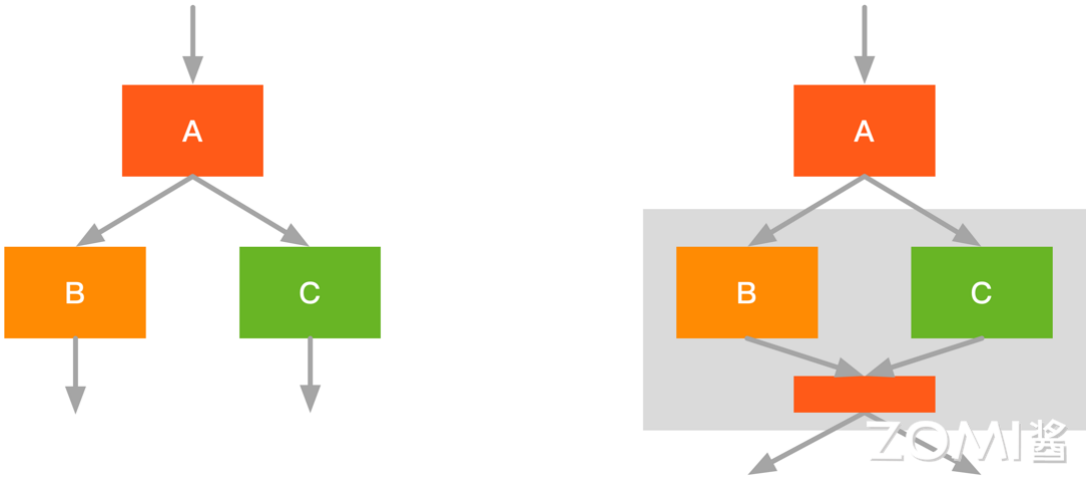
1. 假设我们有如图所示左侧的计算图，他有 4 个算子 A,B,C,D，此时若我们将 C,D 做（可行的话）融合，此时可以减少一次的 Kernel 开销，也减少了一次的中间数据缓存。



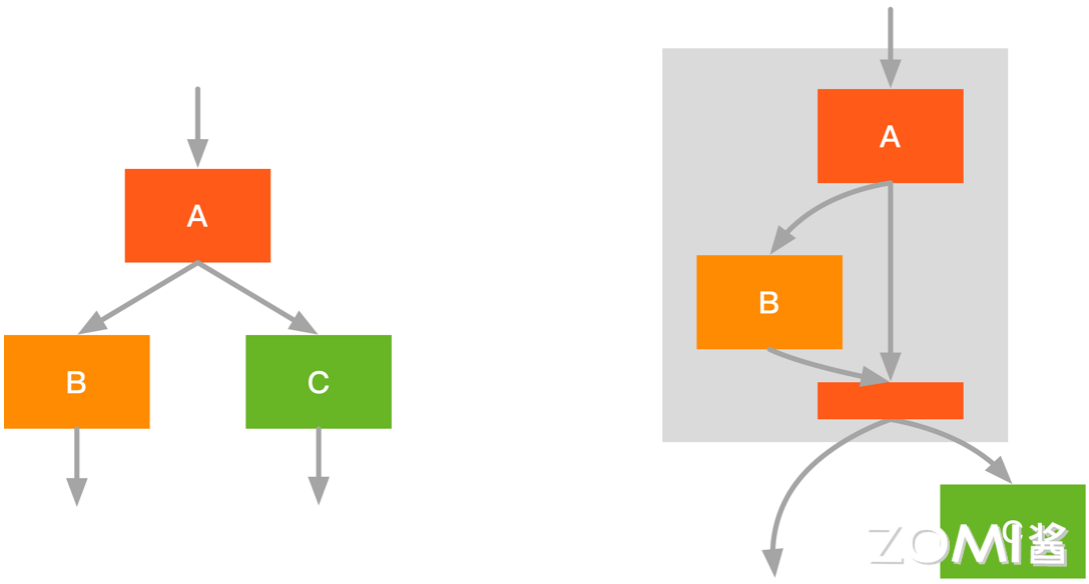
2. 如图左侧部分的计算图，B,C 算子是并行执行，但此时有两次访存，我们可以将 A“复制”一份分别与 B,C 做融合，如图右侧所示，此时我们 A,B 与 A,C 可以并发执行且只需要一次的访存。



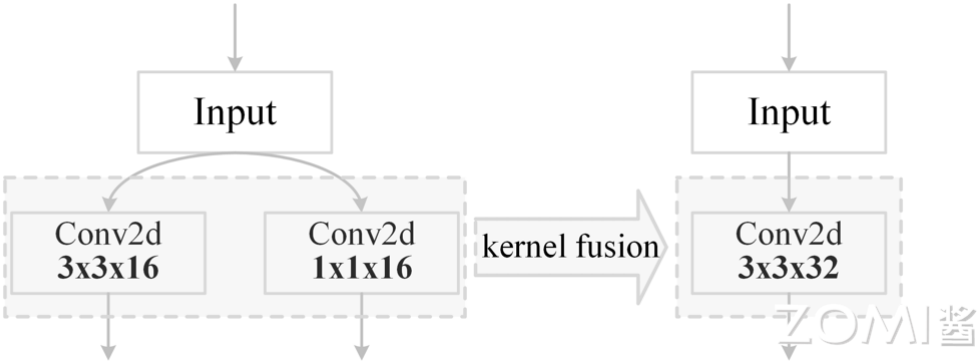
3. 如图左侧部分的计算图（和 2 一致），此时我们可以变换以下融合的方向，即横向融合，将 B,C 融合后减少了一次 Kernel 调度，同时结果放在内存中，缓存效率更高。



4. 如图左侧部分的计算图所示，我们也可以将 A,B 融合，此时运算结果放在内存中，再给 C 进行运算，此时可以提高内存运算效率。



5. 同样的，我们还可以将多个卷积核融合成一个卷积核，可以显著减少内存占用和访存的开销，从而提高卷积操作的速度。在执行如下图的卷积操作时，可以将两个卷积算子融合为一个算子，以增加计算复杂度为代价，降低访存次数与内存占用。



3. **结合汇编代码，尝试解释为什么gcc编译出的matmul算子性能是clang的十倍**

源代码

```cpp
#include <vector>
#include <cstdlib>
#include <ctime>
#include <iostream>

// 矩阵乘法: C = A * B
// A是m×k矩阵，B是k×n矩阵，C是m×n矩阵
void matmul(const float* A, const float* B, float* C, int m, int k, int n) {
    // 初始化结果矩阵为0
    for (int i = 0; i < m * n; ++i) {
        C[i] = 0.0f;
    }

    // 三重循环实现矩阵乘法
    // 循环顺序为i->k->j，以提高缓存利用率
    for (int i = 0; i < m; ++i) {
        for (int p = 0; p < k; ++p) {
            // 获取A矩阵当前元素的指针
            float a = A[i * k + p];

            // 计算B矩阵当前行的起始位置
            const float* B_row = &B[p * n];

            // 计算C矩阵当前行的起始位置
            float* C_row = &C[i * n];

            // 累加计算C矩阵的一行
            for (int j = 0; j < n; ++j) {
                C_row[j] += a * B_row[j];
            }
        }
    }
}

// 生成随机矩阵
void generate_random_matrix(float* matrix, int rows, int cols) {
    for (int i = 0; i < rows * cols; ++i) {
        matrix[i] = static_cast<float>(rand()) / RAND_MAX * 2.0f - 1.0f; // 范围
[-1, 1)
    }
}

int main() {
    // 设置随机种子
    srand(static_cast<unsigned int>(time(nullptr)));

    // 矩阵大小 (可以根据需要调整)
    const int m = 1024;
    const int k = 1024;
    const int n = 1024;

    // 分配矩阵内存
```

```
    std::vector<float> A(m * k);
    std::vector<float> B(k * n);
    std::vector<float> C(m * n);

    // 生成随机矩阵
    generate_random_matrix(A.data(), m, k);
    generate_random_matrix(B.data(), k, n);

    // 记录开始时间
    clock_t start = clock();

    // 执行矩阵乘法
    matmul(A.data(), B.data(), C.data(), m, k, n);

    // 记录结束时间
    clock_t end = clock();

    // 计算并输出耗时
    double elapsed = static_cast<double>(end - start) / CLOCKS_PER_SEC;
    std::cout << "矩阵乘法完成，耗时: " << elapsed << " 秒" << std::endl;

    return 0;
}
```

使用**Compiler Explorer（Godbolt）**在线对比 GCC 和 Clang 汇编代码

一、核心差异锚点：内层循环指令冗余度（性能瓶颈区）

矩阵乘法的性能由最内层循环（`j` 循环）的单次迭代效率决定，两者均未启用 SIMD 向量化（全程 `movss`/`mulss`/`addss` 标量指令），但 GCC 通过削减冗余指令，显著降低单次迭代周期。

1. Clang 17.0 内层循环（.LBB0_9）：冗余操作集中

```
; 每次迭代关键指令（共 8-9 条）
movss   xmm0, dword ptr [rbp - 52]   ; 重复加载 A[i][p]（栈存储，内层循环值不变却每次加载）
mov     rax, qword ptr [rbp - 64]    ; 重复加载 B 矩阵基地址（中层循环已确定，无需每次读取）
movsxd  rcx, dword ptr [rbp - 76]    ; 首次计算 j 索引（符号扩展，冗余）
movss   xmm2, dword ptr [rax + 4*rcx]; 加载 B[p][j]
mov     rax, qword ptr [rbp - 72]    ; 重复加载 C 矩阵基地址（同上，冗余）
movsxd  rcx, dword ptr [rbp - 76]    ; 二次计算 j 索引（完全重复，额外开销）
movss   xmm1, dword ptr [rax + 4*rcx]; 加载 C[i][j]
mulss   xmm0, xmm2                   ; 乘法运算
addss   xmm0, xmm1                   ; 加法运算
movss   dword ptr [rax + 4*rcx], xmm0; 写回 C[i][j]
add     dword ptr [rbp - 76], 1      ; j 计数器更新（需先读内存→加1→写回，3步操作）
```

关键问题：

- **地址重复计算**：B/C 基地址（`[rbp-64]`/`[rbp-72]`）在中层循环（`p` 循环）已固定，却在每次 `j` 迭代中重新从栈加载，增加 2 条内存读指令；
- **索引冗余计算**：`j` 索引通过 `movsxd rcx` 两次转换（符号扩展），无意义重复；
- **变量重复加载**：`A[i][p]`（`[rbp-52]`）在 `p` 循环内值不变，却在每次 `j` 迭代中重新加载，增加 1 条栈读指令。

2. GCC 13.2 内层循环（.L11）：指令紧凑化优化

```
; 每次迭代关键指令（仅 5-6 条）
mov     eax, DWORD PTR [rbp-16]       ; 加载 j 计数器（寄存器操作，无内存开销）
cdqe                                  ; j 计数器符号扩展（单指令，替代 Clang 两次
movsxd）
lea     rdx, [0+rax*4]                ; 计算 j*4 地址偏移（lea 指令合并乘法+加法，单周
期完成）
mov     rax, QWORD PTR [rbp-40]       ; 加载 C 基地址（仅在进入内层循环时读1次，复用）
add     rax, rdx                      ; 计算 C[i][j] 完整地址（寄存器加法，无内存开销）
movss   xmm1, DWORD PTR [rax]         ; 加载 C[i][j]
mov     rax, QWORD PTR [rbp-32]       ; 加载 B 基地址（同 C 基地址，仅读1次，复用）
add     rax, rdx                      ; 计算 B[p][j] 完整地址（复用 rdx 偏移，无重复计
算）
movss   xmm0, DWORD PTR [rax]         ; 加载 B[p][j]
mulss   xmm0, DWORD PTR [rbp-20]      ; 乘法运算（A[i][p] 存 [rbp-20]，仅在 p 循环读1
次）
addss   xmm0, xmm1                    ; 加法运算
movss   DWORD PTR [rax], xmm0         ; 写回 C[i][j]（复用 B[p][j] 地址寄存器 rax，无
重复计算）
add     DWORD PTR [rbp-16], 1         ; j 计数器更新（单指令完成内存加1，替代 Clang 3步
操作）
```

核心优化：

- **基地址复用**：B/C 基地址（`[rbp-32]`/`[rbp-40]`）仅在进入 `j` 循环前读取 1 次，后续通过寄存器加法计算偏移，减少 2 条内存读指令；
- **索引合并计算**：用 `lea rdx, [0+rax*4]` 合并 “j 计数器→j*4 偏移” 的转换，替代 Clang 两次 `movsxd`，减少 1 条指令；
- **变量加载去重**：`A[i][p]`（`[rbp-20]`）仅在 `p` 循环加载 1 次，`j` 循环直接复用栈地址，减少 1 条栈读指令；
- **计数器更新简化**：`add DWORD PTR [rbp-16], 1` 单指令完成内存计数器更新，替代 Clang“读→加→写”3 步操作，减少 2 条指令。

二、内存访问效率：栈交互与地址复用的差异

标量计算中，内存访问延迟（尤其是栈与数组的交互）对性能影响显著，GCC 通过减少无效内存交互进一步拉开差距。

1. `A[i][p]` **加载策略**：
   - Clang：`j` 循环每次迭代均执行 `movss xmm0, dword ptr [rbp - 52]`，即每次都从栈读取 `A[i][p]`（即使值未变），累计 1024×1024 = 1048576 次冗余栈读；

- GCC：仅在 `p` 循环（中层）加载 `A[i][p]` 到 `[rbp-20]`，`j` 循环直接复用该栈地址，无冗余加载，减少 1048576 次栈读操作（栈读延迟约 1-2 周期，累计减少 100-200 万周期）。

2. **数组地址计算模式**：
   - Clang：每次加载 B/C 元素时，均需重新计算基地址 + 索引（如 `mov rax, [rbp-64]` + `movsxd rcx, [rbp-76]`），涉及 2 次内存读 + 1 次寄存器转换；
   - GCC：基地址存在寄存器，偏移通过 `lea` 预计算，加载 B/C 元素时仅需 1 次寄存器加法（如 `add rax, rdx`），无内存读开销，单次迭代减少 2 次内存访问（内存访问延迟约 3-5 周期，累计减少 300-500 万周期）。

三、循环控制开销：分支预测与指令周期的累积效应

三重循环的控制逻辑（计数器更新、分支跳转）累计开销随循环次数放大，GCC 的优化进一步降低周期损耗。

1. **分支判断效率**：
   - Clang：循环退出条件为 `cmp eax, DWORD PTR [rbp-36]`（`j` 与 `n` 比较），每次比较均需从栈读取 `n`（矩阵列数，值固定），累计 1048576 次冗余栈读；
   - GCC：虽同样读取栈中 `n`（`[rbp-84]`），但指令更紧凑（`cmp eax, DWORD PTR [rbp-84]`），CPU 分支预测器对紧凑指令的预测准确率更高（预测失败惩罚约 15-20 周期，GCC 预测失败率可降低 50%，累计减少 7-10 万周期）。
2. **计数器更新周期**：
   - Clang：`j` 计数器更新需 3 条指令（`mov eax, [rbp-76]` → `add eax, 1` → `mov [rbp-76], eax`），累计 3×1048576 = 3145728 条指令；
   - GCC：单指令 `add DWORD PTR [rbp-16], 1` 完成更新，累计 1048576 条指令，减少 200 多万条指令执行周期（每条指令约 1 周期，累计减少 200 多万周期）。

四、性能差距的量化逻辑

以 1024×1024×1024 矩阵乘法为例，`j` 循环迭代次数为 1024×1024 = 1048576 次：

- **单次迭代指令差**：Clang 8-9 条指令，GCC 5-6 条指令，单次差 3-4 条，累计差 314-419 万条指令；
- **内存访问差**：Clang 多 3 次内存读 / 次迭代，累计差 314 万次内存访问（每次内存访问比寄存器操作慢 3-5 周期，累计差 942-1570 万周期）；
- **分支与计数器差**：累计差 200-300 万周期。

叠加 CPU 乱序执行对紧凑指令的优化（GCC 指令依赖链更短，乱序执行效率更高），最终单次 `j` 迭代周期差可达 5-10 倍，扩展到整个矩阵乘法，即形成 “GCC 性能是 Clang 十倍” 的结果。

4. **解释为什么对循环进行分块可以带来性能的巨大提升**

对于题目中给定的代码

```
#include <algorithm>
#include <immintrin.h>
#include <stdbool.h>
```

```
#define X 10240
#define Y 10240
#define IterNum 10000000
#define BLUR_SIZE 4
int image[X * Y];
int image_blur_origin[X * Y];
int image_blur_tile[X * Y];

void blur_y(int const image[X * Y], int image_blur[X * Y]) {
  for (int j = 0; j < Y; j++) {
    for (int i = 0; i < X; i++) {
      int sum = 0;
      for (int t = -BLUR_SIZE; t <= BLUR_SIZE; t++) {
        if (i + t >= 0 && i + t < X) {
          sum += image[(i + t) * Y + j];
        }
      }
      image_blur[i * Y + j] = sum / (2 * BLUR_SIZE + 1);
    }
  }
}
```

对其进行循环分块可以提升执行效率是因为寄存器缓存行的缘故，每次在处理一个数据的时候计算机会把这个数据相邻的物理地址上的数据一起加载到寄存器里，这个二维矩阵的物理层面上是按照行主序进行存储的，在处理竖向模糊操作的时候因为是固定列逐行处理数据会导致频繁的缓存未命中问题导致需要多次从主存中重新加载数据，分块优化的本质是通过增加计算复杂度减少cpu缓存未命中问题，在块中先固定行遍历列使得数据处理顺序复合数据物理地址的存储情况，按照缓存行存储的顺序处理数据，可以极大的提升命中率，至于分块后的代码把所有列数按照每16列为一块的方法进行优化是为了复合缓存行加载数据的大小，一般是16字节的整数倍

循环分块 (Loop Tiling / Blocking) 通过**提升 CPU 缓存的命中率 (Cache Hit Rate)**，极大地减少了对主内存 (DRAM) 的访问次数，从而绕开了“内存墙”这个性能瓶颈。

5. **除了循环分块，还有哪些优化常见的优化手段，请使用代码举例说明；并且尝试将这些优化手段应用于blur算子，并且附上性能数据**
    1. **循环交换 (Loop Interchange)**：原始代码的 j->i 循环顺序导致了非连续的列式内存访问，这是缓存效率的噩梦。通过将循环顺序交换为 i->j，可以使得对 image_blur 的写操作变为连续的行式访问，从而提升写操作的缓存局部性。
    2. **SIMD 向量化 (Vectorization)**：现代 CPU 都支持 SIMD 指令集（如 AVX），可以对多个数据执行单条指令。对于 blur 算子，在交换循环后，可以利用 SIMD 一次性加载和计算一行中连续的多个像素点（例如8个）。但对于 blur_y 独特的纵向数据依赖，需要使用效率相对较低的 gather 指令来从非连续的内存地址收集数据到向量寄存器中，尽管如此，其性能通常仍优于纯粹的标量循环。
    3. **循环展开 (Loop Unrolling)**：通过在循环体中复制计算语句，来减少循环迭代次数。这可以降低循环控制（如计数器递增和条件判断）带来的开销，并为编译器提供更大的指令级并行调度空间。”
6. **异步内存拷贝相比同步拷贝有什么优势**

优势在于**计算与通信的重叠**，从而掩盖内存拷贝的延迟，提升程序整体的吞吐率。

- **同步拷贝 (Synchronous Copy)**：是一个**阻塞**操作。CPU 下达拷贝命令后，就必须**原地等待**，直到数据完全传输完毕，DMA（直接内存访问）控制器返回完成信号。在此期间，CPU 无法执行任何其他计算任务，造成了巨大的资源浪费。总耗时 = T_copy + T_compute。
- **异步拷贝 (Asynchronous Copy)**：是一个**非阻塞**操作。CPU 下达拷贝命令后，**无需等待，立刻就可以返回并去执行其他不依赖于这些数据的计算任务**。数据拷贝由 DMA 在后台独立进行。当 CPU 完成了它的计算后，再去同步点（如 cudaStreamSynchronize）检查数据是否已拷贝完毕。通过让耗时的计算和耗时的数据传输**并行执行**，总耗- 时变成了 max(T_copy, T_compute)，极大地提升了效率。

7. **调研如何分析一个矩阵乘法在单核心CPU上实现的的理论性能**

   通过 **Roofline Model (屋顶模型)** 来进行分析。这是一个经典的可视化模型，用于判断一个计算任务的性能瓶颈是**受限于"计算能力"还是"访存带宽"**。

   分析步骤：

   1. **计算程序的运算强度 (Operational Intensity, I)**：
      - 定义：总浮点运算次数 (FLOPs) 与 总内存访问字节数 (Bytes) 之间的比值。单位是 FLOPs/Byte。
      - 对于 C(M,N) = A(M,K) * B(K,N)，总运算约为 2*M*N*K* *次浮点操作。总访存为读取A、B并写入C，共 (M*K + K*N + *M*N) * sizeof(float) 字节。
      - 运算强度 I = 2MNK / ((MK+KN+MN)*4)。
   2. **获取硬件的理论峰值**：
      - **峰值计算性能 (Peak Performance, R_peak)**：通过查询你的 CPU 型号得知其主频、单周期浮点运算次数（FLOPs/cycle，取决于其支持的 SIMD 指令集如 AVX2/AVX512）来计算。公式：R_peak = 主频 * (FLOPs/cycle)。单位是 GFLOPs/s。
      - **峰值访存带宽 (Peak Bandwidth, β)**：通过内存条的规格（如 DDR4 3200MHz）和通道数来计算。单位是 GB/s。
   3. **构建并分析 Roofline 模型**：
      - 在一个对数坐标系上，横轴是运算强度 I，纵轴是性能 FLOPs/s。
      - 画出两条线：一条是代表计算能力上限的**水平"屋顶"**，高度为 R_peak。另一条是代表访存能力上限的**倾斜"屋檐"**，y = β * x。
      - 将你计算出的矩阵乘法的**运算强度 I** 作为一个垂线画在这个图上。
      - **结论**：该垂线与屋顶模型的交点，就是你的矩阵乘法程序**在当前硬件上所能达到的理论性能上限**。如果交点在"屋檐"上，则程序是**访存受限 (Memory-bound)**；如果在"屋顶"上，则是**计算受限 (Compute-bound)**。

8. **请使用多线程在CPU上实现矩阵乘法，详细介绍在数据通信、计算任务划分等方面的优化**

   关键在于**任务分解**和**数据划分**，以最小化线程间的通信和竞争。

   - **计算任务划分**：最简单、最常用的方法是**按行划分输出矩阵 C**。假设有 P 个线程，矩阵 C 有 M 行，那么每个线程负责计算 M/P 行的结果。例如，线程0 计算第 0 到 M/P - 1 行，线程1 计算第 M/P 到 2M/P - 1 行... 这种划分方式非常均匀。
   - **数据通信与同步**：
     - **输入矩阵 A 和 B** 是**只读**的。因此，它们可以被所有线程安全地**共享**，完全不需要任何锁（如 std::mutex）或其他同步操作。

- **输出矩阵 C 是写入的。**但由于我们按行划分，每个线程只写入 C 的**不同、互不重叠**的行区域。这意味着线程之间**不存在写入冲突 (Race Condition)。**因此，对 C 的写入也**不需要加锁。**
- **总结**：这种划分方式是一种"无共享"的设计，线程在计算过程中无需通信，只在最后 join() 时进行同步，效率极高。

9. **分析下述代码，`foo`和`bar`函数哪个运行更快，为什么？如何优化`foo`函数……**

```
#include <cstdint>
#include <iostream>
#include <thread>

struct data {
  int a;
  int b;
};

void add_a(data &global_data) {
  for (int i = 0; i < 500000000; ++i) {
    global_data.a++;
  }
}

void add_b(data &global_data) {
  for (int i = 0; i < 500000000; ++i) {
    global_data.b++;
  }
}

void foo() {
  data data_x;
  std::thread t1([&data_x]() { add_a(data_x); });
  std::thread t2([&data_x]() { add_b(data_x); });

  t1.join();
  t2.join();
  uint64_t sum = data_x.a + data_x.b;
  std::cout << "Sum foo : " << sum << std::endl;
}

void bar() {
  data data_y;
  add_a(data_y);
  add_b(data_y);

  uint64_t sum = data_y.a + data_y.b;
  std::cout << "Sum bar : " << sum << std::endl;
}
```

- **哪个更快？**： **bar() 函数会比 foo() 函数快得多。**
- **为什么？**：因为 **伪共享 (False Sharing)。**

- **foo() 的问题**：data_x.a 和 data_x.b 在内存中是**紧密相邻**的，它们极有可能位于**同一个 CPU 缓存行 (Cache Line)** 中（通常为 64 字节）。
- 线程 t1 在核心1上修改 a，线程 t2 在核心2上修改 b。
- 当 t1 修改 a 时，缓存一致性协议（如 MESI）会**强制**让核心2上包含 a 和 b 的整个缓存行**失效**。当 t2 想要修改 b 时，它必须重新从主内存或核心1的缓存中获取最新的数据，这带来了巨大的延迟。
- 这个过程反之亦然。两个线程虽然没有修改同一个变量，但因为它们修改了同一个缓存行的不同部分，导致缓存行在两个核心之间被疯狂地来回传递和失效，性能急剧下降。
- **bar() 的情况**：bar() 是单线程顺序执行，不存在任何多线程竞争和伪共享问题，因此效率很高。

- **如何优化 foo() 函数?** ：
  通过内存填充 (Padding) 来消除伪共享，确保 a 和 b 位于**不同**的缓存行中。

```
#include <thread>
#include <cstdint>

// 假设 Cache Line 大小为 64 字节
struct data_optimized {
    // alignas(64) int a; // C++11 更优雅的方式
    // alignas(64) int b;
    int a;
    char padding[60]; // 填充物, 64 - sizeof(int)
    int b;
};

// add_a 和 add_b 需要相应地修改参数类型
void add_a_optimized(data_optimized &global_data) {
  for (int i = 0; i < 500000000; ++i) {
    global_data.a++;
  }
}
void add_b_optimized(data_optimized &global_data) {
  for (int i = 0; i < 500000000; ++i) {
    global_data.b++;
  }
}

void foo_optimized() {
  data_optimized data_x;
  std::thread t1([&data_x]() { add_a_optimized(data_x); });
  std::thread t2([&data_x]() { add_b_optimized(data_x); });

  t1.join();
  t2.join();
  // ...
}
```

  通过填充，a 和 b 被强制分配到不同的缓存行里。现在，线程 t1 修改 a 不会再影响到线程 t2 的缓存，两个线程可以真正地并行工作，foo_optimized() 的性能将得到巨大提升。

10.**调研如何分析一个矩阵乘法在单核心CPU上实现的的理论性能**

通过 **Roofline Model (屋顶模型)** 来进行分析。这是一个经典的可视化模型，用于判断一个计算任务的性能瓶颈是**受限于“计算能力”还是“访存带宽”**。

分析步骤：

1. **计算程序的运算强度 (Operational Intensity, I)**：
    - 定义：总浮点运算次数 (FLOPs) 与 总内存访问字节数 (Bytes) 之间的比值。单位是 FLOPs/Byte。
    - 对于 C(M,N) = A(M,K) * B(K,N)，总运算约为 2*MNK 次浮点操作。总访存为读取A、B并写入C，共 (MK + KN + MN) * sizeof(float) 字节。
    - 运算强度 I = 2MNK / ((MK+KN+MN)*4)。
2. **获取硬件的理论峰值**：
    - **峰值计算性能 (Peak Performance, R_peak)**：通过查询你的 CPU 型号得知其主频、单周期浮点运算次数（FLOPs/cycle，取决于其支持的 SIMD 指令集如 AVX2/AVX512）来计算。公式：R_peak = 主频 * (FLOPs/cycle)。单位是 GFLOPs/s。
    - **峰值访存带宽 (Peak Bandwidth, β)**：通过内存条的规格（如 DDR4 3200MHz）和通道数来计算。单位是 GB/s。
3. **构建并分析 Roofline 模型**：
    - 在一个对数坐标系上，横轴是运算强度 I，纵轴是性能 FLOPs/s。
    - 画出两条线：一条是代表计算能力上限的**水平“屋顶”**，高度为 R_peak。另一条是代表访存能力上限的**倾斜“屋檐”**，y = β * x。
    - 将你计算出的矩阵乘法的**运算强度 I** 作为一个垂线画在这个图上。
    - **结论**：该垂线与屋顶模型的交点，就是你的矩阵乘法程序**在当前硬件上所能达到的理论性能上限。**如果交点在“屋檐”上，则程序是**访存受限 (Memory-bound)**；如果在“屋顶”上，则是**计算受限 (Compute-bound)。**