基础MLSys知识

- MLSyser眼中的模型：只需要关注模型的计算即可，只需要关注的每个层都是矩阵乘法，然后输入进激活函数即可
- 计算图：将网络模式表示为计算图

  NETRON可视化工具

  - ONNX格式：一种常用的跨平台的网络模型表示格式

```
import torch
import torch.nn as nn
import torch.onnx

class MLP(nn.Module):
    """
    Multi-Layer Perceptron (MLP) model class.
    Defines a neural network with four linear layers and sigmoid activations.
    """

    def __init__(self) -> object:
        super().__init__()
        # Define model layers
        self.layer0 = nn.Linear(8, 8, bias=True)
        self.layer1 = nn.Linear(8, 4, bias=True)
        self.layer2 = nn.Linear(4, 2, bias=True)
        self.layer3 = nn.Linear(2, 2, bias=True)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Forward pass of the MLP model.

        Args:
            x (torch.Tensor): Input tensor.

        Returns:
            torch.Tensor: Output tensor after forward pass.
        """
        x = self.layer0(x)
        x = torch.sigmoid(x)
        x = self.layer1(x)
        x = torch.sigmoid(x)
        x = self.layer2(x)
        x = torch.sigmoid(x)
        x = self.layer3(x)
        return x

model = MLP()
example_x = torch.randn(97, 8, dtype=torch.float32)
torch.onnx.export(model, example_x, "mlp.onnx", opset_version=12)
```

写完代码后，在命令行运行（主要是得写上面最后两行代码才能导出）

```
python test.py
```

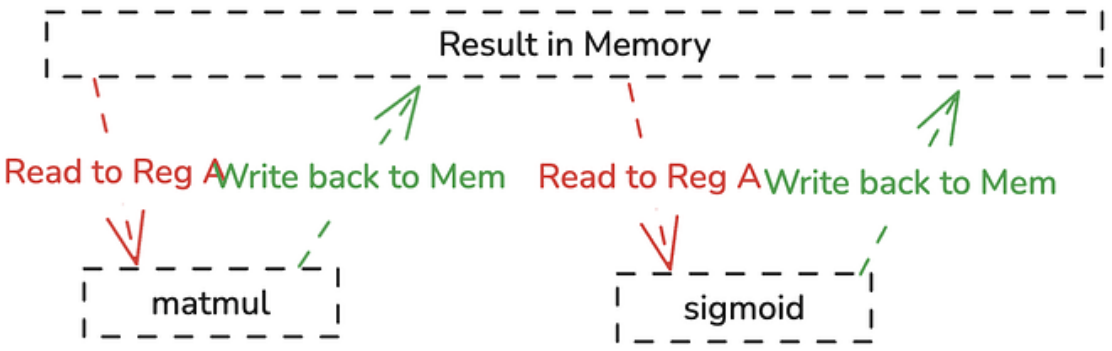
然后就会在当前目录生成onnx

然后再去NETRON可视化即可

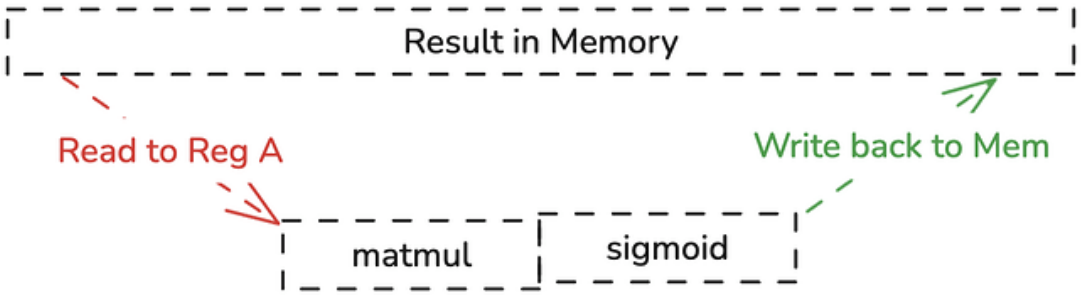
- 计算图优化
  1. 算子融合--图层面的优化
     - 例子:

       通过合并上面的两个算子，使其向内存读写的次数减少

```
# m = 97 , n = k =8
for i in range(m):
    for j in range(n):
        for a in range(k):
            result[i][j] += x[i][a] * weight[a][j]
        result[i][j] += bias[j] # A
for i in range(m):
    for j in range(n):
        result[i][j] = 1 / (1 + math.exp(-M[i][j])) #B
```




```
# m = 97 , n = k =8
for i in range(m):
    for j in range(n):
        for a in range(k):
            result[i][j] += x[i][a] * weight[a][j]
        result[i][j] += bias[j] # A
        result[i][j] = 1 / (1 + math.exp(-M[i][j])) #B
```



2. 自动算子优化--算子层面的优化（针对计算图中特定算子如何在目标架构中生成高性能的实现）

- 例子：循环向量化优化变换(clang -O3 -ffast-math)

```
float reduce_sum(float* array, int size) {
    float sum = 0.0f;
    for (int i = 0; i < size; i++) {
        sum += array[i];
    }
    return sum;
}
```

利用了SIMD（单指令多数据）思想

向量化：主循环每次循环加载8个float然后并行相加，然后再横向求和得到最终的结果

这么做，利用了CPU的并行计算的能力，并且减少了循环的开销，达到了优化的目的

```
// Assume that size is divisible by 8
float reduce_sum_avx(float* array, int size) {
    // 确保数组是 32 字节对齐（AVX 要求对齐加载以获得最佳性能）
    if ((uintptr_t)array % 32 != 0) {
        printf("Warning: Array is not 32-byte aligned. Performance may be
reduced.\n");
    }
    //声明并初始化一个AVX寄存器256位，每个float4字节（32位）
    __m256 sum_vec = _mm256_setzero_ps();  // 初始化 8 个 float = 0

    // 主循环：每次处理 8 个 float
    for (int i = 0; i < size; i += 8) {
        __m256 data = _mm256_load_ps(&array[i]);  // 对齐加载 8 个 float进AVX寄存
器
        sum_vec = _mm256_add_ps(sum_vec, data);  // 8 个 float 并行相加，这一步的目
的是得到一个sum向量，这个向量里面的8部分之和就是最终的结果sum
    }

    // 横向求和：8 个 float -> 1 个 sum，将AVX寄存器中8个浮点数横向求和（合并为一个标量
值）
    float sum = horizontal_sum_avx(sum_vec);
    return sum;
}

// 辅助函数：AVX 寄存器横向求和（8 个 float -> 1 个 float）
float horizontal_sum_avx(__m256 vec) {
    // 1. 加载到128位寄存器
    __m128 low = _mm256_extractf128_ps(vec, 0);  // 低 128 位（4 个 float），提取到
一个SSE寄存器，[a0, a1, a2, a3]
    __m128 high = _mm256_extractf128_ps(vec, 1); // 高 128 位（4 个 float），提取到
另一个SSE寄存器，[a4, a5, a6, a7]

    //2.横向求和
    /*模拟这个函数先对low里面的合成[a0,a1,a2,a3]->[a0+a1,a2+a3]，然后再把两个已经横向求
和的向量相加*/
```

```
    __m128 sum128 = _mm_add_ps(low, high);       // 4 + 4 = 4 个 float，将两个寄存器中的浮点数对应相加，[a0+a1,a2+a3,a4+a5,a6+a7]
    sum128 = _mm_hadd_ps(sum128, sum128);  // 相邻相加
    //[a0+a1+a2+a3,a4+a5+a6+a7,a0+a1+a2+a3,a4+a5+a6+a7]
    sum128 = _mm_hadd_ps(sum128, sum128);  // 再次相加

//[a0+a1+a2+a3+a4+a5+a6+a7,a0+a1+a2+a3+a4+a5+a6+a7,a0+a1+a2+a3+a4+a5+a6+a7,a0+a1+a2+a3+a4+a5+a6+a7]
    // 3. 提取最终结果
    float sum;
    _mm_store_ss(&sum, sum128);  // 存储最低 32 位（即总和）
    return sum;
}
```

3. 手工算子优化及算子库

gpu并行计算及其CPU算子库Eigen

- 例子：对外层循环进行分块变换

给定一个二维离散信号（图像）$I \in \mathbb{Z}^{X \times Y}$，该代码计算其垂直方向（Y轴）的 均值模糊（Box Blur），生成输出图像 $I_{\text{blur}} \in \mathbb{Z}^{X \times Y}$。
对每个像素位置 (i, j)（其中 $0 \le i < X$，$0 \le j < Y$），其模糊后的值为：

$$I_{\text{blur}}[i, j] = \frac{1}{2k + 1} \sum_{t=-k}^{k} I[i + t, j] \cdot 1_{\{0 \le i+t < X\}}$$

其中：

- $k = \text{BLUR\_SIZE}$ 是模糊半径。
- $1_{\{\cdot\}}$ 是指示函数，确保索引不越界（边界处理）。

未优化：

```
#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);
    }
  }
}
```

优化后：

为什么分块之后就快了：

- 因为数组是按行存储的，cache缓存的是按行缓存的，原代码在内循环执行的过程中是访问**同一列的各个行**，这样就造成了cache里面的复用率不高（即命中率不高）

  而优化后的代码是在原来的基础上，再加了一个内循环，即在**特定的一行上面，访问16个相邻的列，即访问同一行的不同列，并且16保证了满足现代CPU缓存行是64字节（16×4字节int）**，大大增加了cache的命中率

```
for (int j = 0; j < Y; j += 16) {     // 外层循环：按块遍历列
  for (int i = 0; i < X; i++) {       // 中层循环：遍历行
    for (int k = 0; k < 16; k++)
```

```
void blur_y_tile(int const image[X * Y], int image_blur[X * Y]) {
  for (int j = 0; j < Y; j += 16) {
    for (int i = 0; i < X; i++) {
      for (int k = 0; k < std::min(16, Y - j); k++) {
        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 + k)];
          }
        }
        image_blur[i * Y + (j + k)] = sum / (2 * BLUR_SIZE + 1);
      }
    }
  }
}
```

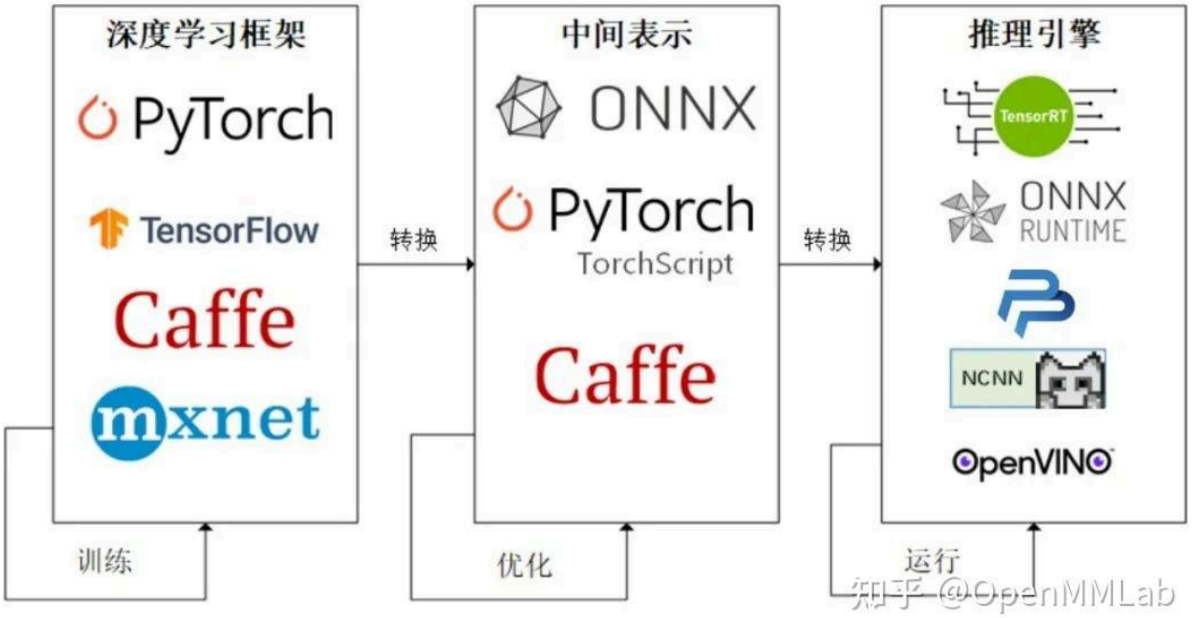
- 问题：

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

   这个题旨在让学者了解如何下载并部署模型

   流程：从Hugging Face获取模型，转换成ONNX等更加高效的模式并优化，模型部署（将优化后的模型集成到一个应用环境（环境稳定高效）中，使其能够对外进行服务）

   模型部署教程--知乎



为了让模型最终能够部署到某一环境上，开发者们可以使用任意一种**深度学习框架**来定义网络结构，并通过训练确定网络中的参数。之后，模型的结构和参数会被转换成一种只描述网络结构的**中间表示**，一些针对网络结构的优化会在中间表示上进行。最后，用面向硬件的高性能编程框架(如 CUDA，OpenCL）编写，能高效执行深度学习网络中算子的**推理引擎**会把中间表示转换成特定的文件格式，并在对应硬件平台上高效运行模型。

这一条流水线解决了模型部署中的两大问题：使用对接深度学习框架和推理引擎的中间表示，开发者不必担心如何在新环境中运行各个复杂的框架；通过中间表示的网络结构优化和推理引擎对运算的底层优化，模型的运算效率大幅提升。

- 了解llama 3b模型
  - transformer架构，用于自然语言处理任务
  - 3b：30亿参数（属于小参数规模）

1. 模型下载

   首先，先从https://www.llama.com/官网download models填写身份信息认证（梯子挂美国，国家写美国）

   本人qq邮箱认证的邮件，之后用这个来配置

下载

要下载模型权重和标记器:

1. 访问 Meta Llama 网站。
2. 阅读并接受许可证。
3. 请求获得批准后,您将通过电子邮件收到签名的URL。
4. 安装 Llama CLI : `pip install llama-stack` . .(**<--如果您已经收到电子邮件,请从这里开始。**
5. 运行 `llama model list` 显示最新的可用模型并确定您要下载的模型 ID。**注:** 如果您想要旧版本的模型,请运行 `llama model list --show-all` 展示所有可用的Llama模型。
6. 运行: `llama download --source meta --model-id CHOSEN_MODEL_ID`
7. 提示启动下载时传递提供的 URL。

请记住,这些链接在24小时后过期,下载量一定。如果您开始看到错误,例如 `403: Forbidden` . .

我尝试了这种方法，发现总是显示403,应该是邮件里面的url过期太快了，没过期的话也会因为网络问题终止下载我也尝试了huggingface下载，发现同样需要在官网认证，然后我也放弃了

然后我就转向了国内开源大模型平台ModelScope，然后通过配置成功下载了llama3b模型llama-3.3-3B-Instruct

2. 利用huggingface的transformer库加载一个预训练的模型（注意一定要用transformer架构的，因为我用的是huggingface的transformer来加载的模型）

   加载模型的模板（transformer版）

```
from transformers import AutoModelForCausalLM,AutoTokenizer

#使用transformer加载模型
device = "cuda"

model_dir = "/home/lyjy/model/llama_3b2/LLM-Research/Llama-3___2-3B-Instruct"

model =
AutoModelForCausalLM.from_pretrained(model_dir,torch_dtype="auto",device_map="a
uto")
#加载分词器
tokenizer = AutoTokenizer.from_pretrained(model_dir)
#提示词
prompt = "你好，世界"
message = [{"role":"system","content":"Hello world"},
{"role":"user","content":prompt}]

text =
tokenizer.apply_chat_template(message,tokenize=False,add_generation_prompt=True
)

model_inputs = tokenizer([text],return_tensors="pt").to(device)

generate_ids = model.generate(model_inputs.input_ids,max_new_tokens=512)
```

```
print(generate_ids)

responce = tokenizer.batch_decode(generate_ids,skip_special_tokens=True)
print(responce)
```

```
(nanobindPractice) lyjy@Lenovo:~/model/llama_3b2/LLM-Research$
/home/lyjy/miniconda3/envs/nanobindPractice/bin/python
/home/lyjy/MlSys_LYJY/Part2_2/llama3b/usellama.py
......
['system\n\nCutting Knowledge Date: December 2023\nToday Date: 03 Sep
2025\n\nHello worlduser\n\n你好，世界assistant\n\n你好！']
```

3. 将llama模型转化为onnx格式，并用ONNX Runtime部署

   [huggingface官方教程]

   导出指令

```
optimum-cli export onnx --model ./Llama-3___2-3B-Instruct --task text-
generation ./onnx_model
```

   666直接kill掉了，好像是内存不足

```
(nanobindPractice) lyjy@Lenovo:~/model/llama_3b2/LLM-Research$ optimum-cli
export onnx --model ./Llama-3___2-3B-Instruct --task text-generation
./onnx_model
.....
Killed
```

   寻找解决方法：

   这里我尝试租用一个阿里云实例来完成云端上转换为onnx格式

   1. 创建实例（我这里租用了一个Nvidia A10来完成）

      几个必要的选项：

      付费类型：按量计费

      地域：成都

      网络及可用区：专有网络（创建专有网络：填写转有网络名称选172.16.0.0/16IPv4网段，交换区填个名字可用区选和地域一样的，交换机网段要保持和刚才专有网络网段一样的（除了"/"后面的可以不一样，其他必须一样））

      选择你需要的实例：我这里选的是GPU->GPU计算型gn7i（Nvidia A10）选的内存尽可能大的

      镜像：Ubuntu2204，选上安装GPU驱动（用来预先创建和深度学习有关的东西），然后选择版本比12.9低的就行（因为本地电脑是12.9）

      系统盘：把容量改成100GiB

      数据盘：添加一个，容量改成200GiB

      网络和安全组：选上分配公网IPv4地址，如果对带宽有需求可以改

      管理设置：选用密钥对，root，密钥对选用创建好的，然后其他的不用管，点击创建即可

      然后启动实例，然后使用完实例别忘了关就行

2. 连接（使用密钥对连接）

   从前到后解读命令：

   /home/lyjy/Downloads/lyjy.pem： 你的本地电脑上ssh.pem（私钥位置）

   root@47.109.111.96：云端GPU的公网IP地址

```
lyjy@Lenovo:~$ ssh -i /home/lyjy/Downloads/lyjy.pem root@47.109.111.96
```

3. 从本地电脑传输数据到云端GPU

   注意这里需要再开一个终端，刚才的终端用于连接云端GPU并操作，这个新开的终端用于执行以下命令来传输文件到云端

   从前到后解读命令：

   /home/lyjy/Downloads/lyjy.pem：你的本地电脑上ssh.pem（私钥位置）

   /home/lyjy/model/llama_3b2/LLM-Research： 你要传输的文件的位置

   root@47.109.111.96:./model：云端GPU的公网地址和你要把文件传输到云端的那个目录里面

```
lyjy@Lenovo:~$ scp -i /home/lyjy/Downloads/lyjy.pem -r
/home/lyjy/model/llama_3b2/LLM-Research root@47.109.111.96:./model
```

4. 在云端GPU上下载相应的包，然后开始将模型转换成onnx格式

   只有这里是云端GPU的终端

```
(test) root@iZ2vc4hkiei7dj2rzsa8d7Z:~/model/LLM-Research# optimum-cli
export onnx --model ./Llama-3___2-3B-Instruct --task text-generation
./onnx_model
```

   成功了

```
The ONNX export succeeded with the warning: The maximum absolute difference
between the output of the reference model and the ONNX exported model is
not within the set tolerance 1e-05:
- logits: max diff = 0.0002262592315673828.
 The exported model was saved at: onnx_model
```

5. 在原电脑终端上下载回导出成功的onnx

```
scp -i [你的密钥文件路径] -r [用户名]@[服务器IP]:[服务器上的源路径] [本地目标路径]
```

   下载太慢，先在GPU终端打包

```
# 首先进入包含 onnx_model 的目录
cd ~/model/LLM-Research/

# 将 onnx_model 文件夹打包压缩成一个 .tar.gz 文件
tar -czvf onnx_model.tar.gz onnx_model/
```

然后再在主机上下载

```
scp -i /home/lyjy/Downloads/lyjy.pem root@47.109.111.96:~/model/LLM-Research/onnx_model.tar.gz /home/lyjy/model/llama_3b2/LLM-Research
```

还是太慢，我决定直接在云端GPU上部署！！，这样就不会出现吃计算资源的原因了，毕竟就算传输到本电脑的onnx格式，也可能由于吃计算资源而无法部署的情况

6. **直接在云端GPU上部署**

编写一个脚本

```
(test) root@iZ2vc4hkiei7dj2rzsa8d7Z:~/model/LLM-Research# nano onnx_deploy.py
```

```python
#!/usr/bin/env python3
"""
ONNX Runtime 部署脚本 for Llama 3B 模型
作者: [你的名字]
创建日期: 2024

此脚本用于加载 ONNX 格式的 Llama 3B 模型并使用 ONNX Runtime 进行文本生成。
支持命令行参数和交互式两种模式。
"""

import argparse
import time
import numpy as np
from transformers import AutoTokenizer
import onnxruntime as ort
import os
import sys

class ONNXModelDeployer:
    def __init__(self, model_path, use_gpu=True):
        """
        初始化模型部署器

        参数:
            model_path: ONNX 模型目录路径
            use_gpu: 是否使用 GPU 进行推理
        """
        self.model_path = model_path
        self.use_gpu = use_gpu
        self.session = None
        self.tokenizer = None
        self.input_names = None

        # 初始化模型和分词器
        self.load_model()

    def load_model(self):
        """加载模型和分词器"""
```

```
        print("正在加载分词器...")
        try:
            self.tokenizer = AutoTokenizer.from_pretrained(self.model_path)
            if self.tokenizer.pad_token is None:
                self.tokenizer.pad_token = self.tokenizer.eos_token
            print("✓ 分词器加载成功")
        except Exception as e:
            print(f"✗ 分词器加载失败: {e}")
            sys.exit(1)

        print("正在创建 ONNX Runtime 会话...")
        try:
            # 配置提供程序
            providers = ['CUDAExecutionProvider', 'CPUExecutionProvider']
if self.use_gpu else ['CPUExecutionProvider']

            # 会话选项
            sess_options = ort.SessionOptions()
            sess_options.graph_optimization_level =
ort.GraphOptimizationLevel.ORT_ENABLE_ALL

            # 创建会话
            model_file = os.path.join(self.model_path, "model.onnx")
            self.session = ort.InferenceSession(model_file,
sess_options=sess_options, providers=providers)

            # 获取输入名称
            self.input_names = [input.name for input in
self.session.get_inputs()]
            print(f"✓ ONNX Runtime 会话创建成功")
            print(f"  输入名称: {self.input_names}")
            print(f"  使用设备: {'GPU' if self.use_gpu else 'CPU'}")

        except Exception as e:
            print(f"✗ ONNX Runtime 会话创建失败: {e}")
            sys.exit(1)

    def prepare_inputs(self, input_text):
        """
        准备模型输入

        参数:
            input_text: 输入文本

        返回:
            包含模型输入的字典
        """
        # 使用分词器处理输入
        inputs = self.tokenizer(input_text, return_tensors="np")

        # 构建输入字典
        input_feed = {}
        for name in self.input_names:
```

```
            if "input_ids" in name:
                input_feed[name] = inputs["input_ids"].astype(np.int64)
            elif "attention_mask" in name:
                input_feed[name] =
inputs["attention_mask"].astype(np.int64)
            elif "position_ids" in name:
                # 某些模型需要 position_ids
                seq_length = inputs["input_ids"].shape[1]
                position_ids = np.arange(seq_length,
dtype=np.int64).reshape(1, -1)
                input_feed[name] = position_ids
            else:
                # 对于未知的输入，使用默认值
                input_feed[name] = inputs["input_ids"].astype(np.int64)

        return input_feed

    def generate_text(self, prompt, max_length=50, temperature=0.7,
top_k=50, top_p=0.9):
        """
        生成文本

        参数:
            prompt: 提示文本
            max_length: 最大生成长度
            temperature: 温度参数，控制随机性
            top_k: top-k 采样参数
            top_p: top-p (nucleus) 采样参数

        返回:
            生成的文本
        """
        start_time = time.time()

        # 编码输入文本
        inputs = self.tokenizer(prompt, return_tensors="np")
        input_ids = inputs["input_ids"]

        # 存储生成的 token
        generated_ids = input_ids.copy()

        print(f"\n开始生成 (最大长度: {max_length})...")
        print("=" * 60)

        # 自回归生成循环
        for step in range(max_length):
            # 准备输入
            input_feed =
self.prepare_inputs(self.tokenizer.decode(generated_ids[0]))

            # 运行模型推理
            outputs = self.session.run(None, input_feed)
```

```
            next_token_logits = outputs[0][0, -1, :]  # 获取最后一个位置的
logits

            # 应用温度缩放
            next_token_logits = next_token_logits / temperature

            # Top-k 采样
            if top_k > 0:
                indices_to_remove = next_token_logits <
np.partition(next_token_logits, -top_k)[-top_k]
                next_token_logits[indices_to_remove] = -float('inf')

            # Top-p (nucleus) 采样
            if top_p < 1.0:
                sorted_logits = np.sort(next_token_logits)[::-1]
                cumulative_probs = np.cumsum(np.exp(sorted_logits) /
np.sum(np.exp(sorted_logits)))

                # 移除累积概率高于 top_p 的 token
                remove_idx = cumulative_probs > top_p
                remove_idx[1:] = remove_idx[:-1].copy()
                remove_idx[0] = False

                sorted_indices = np.argsort(next_token_logits)[::-1]
                next_token_logits[sorted_indices[remove_idx]] = -
float('inf')

            # 应用 softmax 获取概率分布
            probs = np.exp(next_token_logits) /
np.sum(np.exp(next_token_logits))

            # 从分布中采样
            next_token_id = np.random.choice(len(probs), p=probs)

            # 将新 token 添加到序列中
            generated_ids = np.concatenate(
                [generated_ids, [[next_token_id]]], axis=-1
            )

            # 如果生成了结束符，停止生成
            if next_token_id == self.tokenizer.eos_token_id:
                break

        # 解码最终结果
        generated_text = self.tokenizer.decode(generated_ids[0],
skip_special_tokens=True)

        # 计算性能指标
        generation_time = time.time() - start_time
        tokens_generated = len(generated_ids[0]) - len(input_ids[0])
        speed = tokens_generated / generation_time if generation_time > 0
else 0
```

```
        print(f"生成完成!")
        print(f"耗时: {generation_time:.2f}秒")
        print(f"生成token数: {tokens_generated}")
        print(f"生成速度: {speed:.2f} token/秒")
        print("=" * 60)

        return generated_text

    def interactive_mode(self):
        """交互式模式"""
        print("\n进入交互式模式。输入 'quit' 或 'exit' 退出。")
        print("你可以使用以下命令调整参数:")
        print("  /max <数字>    - 设置最大生成长度")
        print("  /temp <数字>   - 设置温度 (默认: 0.7)")
        print("  /topk <数字>   - 设置 top-k (默认: 50)")
        print("  /topp <数字>   - 设置 top-p (默认: 0.9)")

        max_length = 50
        temperature = 0.7
        top_k = 50
        top_p = 0.9

        while True:
            try:
                user_input = input("\n请输入提示文本: ").strip()

                if user_input.lower() in ['quit', 'exit']:
                    break

                # 处理命令
                if user_input.startswith('/'):
                    parts = user_input.split()
                    cmd = parts[0].lower()

                    if cmd == '/max' and len(parts) > 1:
                        try:
                            max_length = int(parts[1])
                            print(f"最大生成长度设置为: {max_length}")
                        except ValueError:
                            print("错误: 请输入有效的数字")

                    elif cmd == '/temp' and len(parts) > 1:
                        try:
                            temperature = float(parts[1])
                            print(f"温度设置为: {temperature}")
                        except ValueError:
                            print("错误: 请输入有效的数字")

                    elif cmd == '/topk' and len(parts) > 1:
                        try:
                            top_k = int(parts[1])
                            print(f"Top-k 设置为: {top_k}")
                        except ValueError:
```

```
                            print("错误: 请输入有效的数字")

                    elif cmd == '/topp' and len(parts) > 1:
                        try:
                            top_p = float(parts[1])
                            print(f"Top-p 设置为: {top_p}")
                        except ValueError:
                            print("错误: 请输入有效的数字")

                    else:
                        print("未知命令")

                    continue

                # 生成文本
                if user_input:
                    result = self.generate_text(
                        user_input,
                        max_length=max_length,
                        temperature=temperature,
                        top_k=top_k,
                        top_p=top_p
                    )
                    print(f"\n生成结果:\n{result}")

            except KeyboardInterrupt:
                print("\n\n程序被用户中断")
                break
            except Exception as e:
                print(f"生成过程中出错: {e}")

def main():
    """主函数"""
    parser = argparse.ArgumentParser(description='ONNX Runtime Llama 3B 模型
部署')
    parser.add_argument('--model-path', type=str, default='./onnx_model',
                        help='ONNX 模型目录路径 (默认: ./onnx_model)')
    parser.add_argument('--prompt', type=str,
                        help='直接指定提示文本, 不指定则进入交互模式')
    parser.add_argument('--max-length', type=int, default=50,
                        help='最大生成长度 (默认: 50)')
    parser.add_argument('--temperature', type=float, default=0.7,
                        help='温度参数, 控制随机性 (默认: 0.7)')
    parser.add_argument('--top-k', type=int, default=50,
                        help='Top-k 采样参数 (默认: 50)')
    parser.add_argument('--top-p', type=float, default=0.9,
                        help='Top-p (nucleus) 采样参数 (默认: 0.9)')
    parser.add_argument('--cpu', action='store_true',
                        help='强制使用 CPU (默认使用 GPU 如果可用)')

    args = parser.parse_args()

    # 检查模型路径是否存在
```

```
    if not os.path.exists(args.model_path):
        print(f"错误: 模型路径 '{args.model_path}' 不存在")
        sys.exit(1)

    # 创建部署器实例
    print("初始化 ONNX 模型部署器...")
    deployer = ONNXModelDeployer(args.model_path, use_gpu=not args.cpu)

    # 运行模式选择
    if args.prompt:
        # 单次生成模式
        result = deployer.generate_text(
            args.prompt,
            max_length=args.max_length,
            temperature=args.temperature,
            top_k=args.top_k,
            top_p=args.top_p
        )
        print(f"\n生成结果:\n{result}")
    else:
        # 交互式模式
        deployer.interactive_mode()

    print("\n程序结束")

if __name__ == "__main__":
    main()
```

赋予执行权限

```
(test) root@iZ2vc4hkiei7dj2rzsa8d7Z:~/model/LLM-Research# chmod +x
onnx_deploy.py
```

开始测试

交互式测试（注意一定要用GPU跑）

2. profiling算子融合所带来的性能受益

对上面的两个循环编写一个对比脚本来发现优化的效果

可见在一百次循环中优化效果还是很明显的

```
性能对比 (运行100次的时间):
Version1 (原始版本): 0.2589 秒
Version2 (优化版本): 0.2264 秒
Version2 比 Version1 快 1.14 倍
```

但是在1000次运行和10000次运行下其实优化没有那么明显

通过查阅ai等我发现cpu和py解释器会对这种频繁执行的代码进行再度优化，比如cpu缓存频繁执行的指令，数据（cache 命中率增加）

```
1000次运行
Version1 (原始版本): 2.1150 秒
Version2 (优化版本): 2.0798 秒
Version2 比 Version1 快 1.02 倍

10000次运行
Version1 (原始版本): 20.6451 秒
Version2 (优化版本): 20.7452 秒
Version2 比 Version1 快 1.00 倍
```

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

   编写一个测试文件

```
# 编译可执行文件
g++ -O3 -march=native -o gemm_gcc gemm_test.cpp

# 生成汇编代码
g++ -O3 -march=native -S -fverbose-asm gemm_test.cpp -o gemm_gcc.s

# 编译可执行文件
clang++ -O3 -march=native -o gemm_clang gemm_test.cpp

# 生成汇编代码
clang++ -O3 -march=native -S -fverbose-asm gemm_test.cpp -o gemm_clang.s

# 运行GCC编译的程序
./gemm_gcc

# 运行Clang编译的程序
./gemm_clang
```

   性能对比：可见gcc编译确实比clang编译要快一点

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_4$ ./gemm_gcc
Matrix multiplication took 12 ms
Result verification passed!
lyjy@Lenovo:~/MlSys_LYJY/Part2_4$ ./gemm_clang
Matrix multiplication took 17 ms
Result verification passed!
```

   汇编文件对比：

   - Clang：使用 `vmovss` / `vfmadd231ss`（标量单精度 SSE 指令的 FMA 形式），即每次处理一个浮点

     此FMA指令，一条指令实现了乘和加的逻辑

     内循环

```
.LBB0_7:
    vmovss  (%rbx), %xmm1                   #从内存地址(%rbx)中读取32bits的
float，把它放到%xmm1的低32位
    addq    $4, %rbx                        #把%rbx向前移动4字节，指向下一个元素
```

```
    #下面这个是最重要的一行，用来告诉你Clang的优化思想

    vfmadd231ss  (%rbp), %xmm1, %xmm0            #执行FMA：先从地址%rbp取一个float（B当前元素），把刚取到的 A 元素（在 %xmm1）和 B 指向的元素相乘，然后累加到当前的 sum（保存在 %xmm0）



    addq    %rax, %rbp                           #只需要知道这里把B的指针指向了下一个元素
    addq    $-1, %rcx                            #每做完一次就把次数减1
    jne     .LBB0_7
# after loop:
    vmovss  %xmm0, (%rdx,%rcx,4)
```

- GCC：

  向量化（SIMD）+水平归约

  载入重排/聚合：先把没有在一起的float拼接成一个4float的向量（因为B是按列遍历，在内存中不在一起）

  向量化SIMD：并行对4个float做乘法

  水平归约：将4float的向量逐步累加到累加器（一个寄存器用来放sum的结果）

```
.L5:
    vmovss  (%rdx,%rax,2), %xmm0                 #从地址 rdx + rax*2 读一个 32-bit float，放入 %xmm0 的低 32 位（其余位不关心）。
    vinsertps $0x10, (%rdx,%r8), %xmm0, %xmm2    #内存 rdx + r8 取一个 float，按 vinsertps 的立即数 $0x10 指定的位置插入到源 %xmm0 中，结果写入 %xmm2。
    #利用 vinsertps 把第二个 B 元素插入到一个临时向量寄存器（%xmm2）。这表示 B 的元素在内存中不是连续 4 个 float 的形式，编译器通过 vinsertps 做“gather-like”操作把分散的标量组装为向量。
    addq    $16, %rcx                            #推进A指针，以便读取A连续的4个float
    vmovss  (%rdx), %xmm0                        #再从B的另一个地址读取一个标量
    vinsertps $0x10, (%rdx,%rax), %xmm0, %xmm0   #把第四个（或第三个，取决于插入顺序）B 标量放入 %xmm0 的另一个 slot。到现在为止，%xmm0 和 %xmm2 分别持有两个标量的低 64-bit，各自保存两组标量片段。
    addq    %r9, %rdx                            #把 rdx 移到下一行的起始（为了下一次迭代抓取后续 B 的不同位置），或者总体推进 B 的读出基址以便处理下一个向量组。



    vmovlhps %xmm2, %xmm0, %xmm0                 #把 %xmm2 的低 64-bit 拷贝到 %xmm0 的高 64-bit，形成一个完整的 128-bit 向量在 %xmm0。
    #即组成了一个4个float的向量
    #下面这条指令是最关键的，即实现并行乘法
    vmulps  -16(%rcx), %xmm0, %xmm0              #对 %xmm0 中的 4 个 float 与内存位置 -16(%rcx) 的 4 个 float 做并行乘法（vmulps 是 packed single-precision multiply）。结果仍然存在 %xmm0。
```

```
    vaddss  %xmm0, %xmm1, %xmm1                    #vaddss 做 scalar single-
precision add，只作用于每个寄存器的低 32 位。这里把 %xmm0 的低 32 位（即第 0 个乘积）加到
%xmm1（标量累加器）上。
    vshufps $85, %xmm0, %xmm0, %xmm2               #把向量中的第 1 个乘积提取到
%xmm2 的低 32 位，为下一步加到累加器做准备。
    vaddss  %xmm2, %xmm1, %xmm1                    #把第 1 个乘积累加进 sum。
    vunpckhps %xmm0, %xmm0, %xmm2                  #把原向量的第 2、3 个乘积取出来
放到 %xmm2 的低位，方便接下来单独提取并加到累加器
    vshufps $255, %xmm0, %xmm0, %xmm0              #提取出第 3 个乘积到 %xmm0 的低
位，准备加到累加器
    vaddss  %xmm2, %xmm1, %xmm1                    #把第 2 个乘积累加进 sum
    vaddss  %xmm0, %xmm1, %xmm1                    #把第 3 个乘积累加进 sum
```

- 为什么比FMA更快：

  向量化的使用减少了循环开支（每次处理4个float，迭代次数变成原来的1/4），并且提高了硬件的吞吐量（因为每次处理4个float）

  减少了访存的时间：一次取4个肯定比4次取一个时间短

4. 解释为什么对循环进行分块可以带来性能的巨大提升

   上面解释了--这里粘贴过来

   因为数组是按行存储的，cache缓存的是按行缓存的，原代码在内循环执行的过程中是访问**同一列的各个行**，这样就造成了cache里面的复用率不高（即命中率不高）

   而优化后的代码是在原来的基础上，再加了一个内循环，即在**特定的一行上面，访问16个相邻的列，即访问同一行的不同列，并且16保证了满足现代CPU缓存行是64字节（16×4字节int）**，大大增加了cache的命中率

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

   - blur算子：

     给定一个二维离散信号（图像）$I \in \mathbb{Z}^{X\times Y}$，该代码计算其垂直方向（Y轴）的 均值模糊（Box Blur），生成输出图像 $I_{\text{blur}} \in \mathbb{Z}^{X\times Y}$。
     对每个像素位置 (i,j)（其中 $0 \le i < X$，$0 \le j < Y$），其模糊后的值为：

     $$I_{\text{blur}}[i,j] = \frac{1}{2k+1} \sum_{t=-k}^{k} I[i+t,j] \cdot 1_{\{0\le i+t<X\}}$$

     其中：

     - $k = \text{BLUR\_SIZE}$ 是模糊半径。
     - $1_{\{\cdot\}}$ 是指示函数，确保索引不越界（边界处理）。

```
#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);
    }
  }
}
```

- 优化：

  1. 循环分块：上面提到了
  2. SIMD向量化：

     使用AVX2指令集：一个支持向量化手段的C++库

     代码实现循环分块+SIMD向量化（个人实现，由于测试用例10240能被16整除，所以没考虑边界情况）：实现向量化的同时也实现了循环分块的思想，并且一次处理一行16个数值也完美利用了cache的大小

```
//SIMD向量化+循环分块优化函数
void blur_y_simd(int const image[X * Y], int image_blur[X * Y]) {
  const __m512 scale_vec = _mm512_set1_ps(1.0f/Two_BLUR_SIZE_PLUS_ONE);
  for (int j = 0; j < Y; j += 16) {
    for (int i = 0; i < X; i++) {//一次处理水平8个的垂直模糊
        //初始化AVX寄存器
        __m512i acc = _mm512_setzero_si512();
        for(int t = -BLUR_SIZE;t<=BLUR_SIZE;t++){
            if(i+t >= 0 && i+t<X){
            __m512i data = _mm512_loadu_si512(
                        (__m512i const*)&image[(i + t) * Y + j]);
            acc = _mm512_add_epi32(acc, data);
            }
        }
        //转换成float进行乘法
        __m512 acc_float = _mm512_cvtepi32_ps(acc);
        acc_float = _mm512_mul_ps(acc_float, scale_vec);
        __m512i result = _mm512_cvttps_epi32(acc_float);
        _mm512_storeu_si512((__m512i*)&image_blur[i * Y + j], result);
      }
    }
  }
```

     666但是最后报错发现我的电脑用不了AVX512，还是老老实实用AVX256吧

```
//SIMD向量化+循环分块优化函数
void blur_y_simd(int const image[X * Y], int image_blur[X * Y]) {
  const __m256 scale_vec = _mm256_set1_ps(1.0f/Two_BLUR_SIZE_PLUS_ONE);
  for (int j = 0; j < Y; j += 8) {
    for (int i = 0; i < X; i++) {
        __m256i acc = _mm256_setzero_si256();
        for(int t = -BLUR_SIZE;t<=BLUR_SIZE;t++){
            if(i+t >= 0 && i+t<X){
            __m256i data = _mm256_loadu_si256(
                        (__m256i const*)&image[(i + t) * Y + j]);
            acc = _mm256_add_epi32(acc, data);
            }
        }
        __m256 acc_float = _mm256_cvtepi32_ps(acc);
        acc_float = _mm256_mul_ps(acc_float, scale_vec);
        __m256i result = _mm256_cvttps_epi32(acc_float);
        _mm256_storeu_si256((__m256i*)&image_blur[i * Y + j], result);
      }
    }
}
```

结果：达到了7的加速比

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ g++ -O0 -mavx2 -o opt_blur
opt_blur.cpp
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ ./opt_blur
Original: 7031ms
SIMD: 896ms
Original/SIMD加速比:7
```

3. 多线程并行化：

思路就是列循环并行化，每个CPU核心处理一个块，块内使用了SIMD并行化处理8个元素，并且使用了循环分块，减少了cache miss的情况

SIMD并行化+循环分块+OpenMP多线程：

```
#include <omp.h>

//SIMD向量化+循环分块优化+OpenMP多线程
void blur_y_simd(int const image[X * Y], int image_blur[X * Y]) {
  const __m256 scale_vec = _mm256_set1_ps(1.0f/Two_BLUR_SIZE_PLUS_ONE);
  //使用openmp并行化列循环
  #pragma omp parallel for
  for (int j = 0; j < Y; j += 8) {
    for (int i = 0; i < X; i++) {
        __m256i acc = _mm256_setzero_si256();
        for(int t = -BLUR_SIZE;t<=BLUR_SIZE;t++){
            if(i+t >= 0 && i+t<X){
            __m256i data = _mm256_loadu_si256(
                        (__m256i const*)&image[(i + t) * Y + j]);
            acc = _mm256_add_epi32(acc, data);
            }
        }
```

```
            __m256 acc_float = _mm256_cvtepi32_ps(acc);
            acc_float = _mm256_mul_ps(acc_float, scale_vec);
            __m256i result = _mm256_cvttps_epi32(acc_float);
            _mm256_storeu_si256((__m256i*)&image_blur[i * Y + j], result);
        }
    }
}
```

加速比达到了65:

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ g++ -O0 -mavx2 -fopenmp -o opt_blur
opt_blur.cpp
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ ./opt_blur
Original: 6649ms
SIMD: 101ms
Original/SIMD加速比:65
```

SIMD并行化+循环分块+手写多线程（列循环并行化）:

```
//SIMD向量化+循环分块优化+手写多线程（列循环并行化）
void blur_y_simd(int const image[X * Y], int image_blur[X * Y]) {
  const __m256 scale_vec = _mm256_set1_ps(1.0f/Two_BLUR_SIZE_PLUS_ONE);

  //获得可用的CPU核心数
  unsigned int num_threads = std::thread::hardware_concurrency();
  if (num_threads == 0) num_threads = 4; // 默认使用4个线程

  int total_blocks = Y/8;
  //因为这个例子是10240/8,所以一共1280个块
  //下面是每个线程需要处理的块
  int block_per_thread = total_blocks / num_threads;
  //下面是平分不完的blocks
  int remainer_blocks = total_blocks % num_threads;

  //创建线程向量
  std::vector<std::thread> threads;

  //启动线程
  int start_block = 0;
  //p为当前线程的索引
  for(unsigned int p =0;p<num_threads;p++){
    //计算当前thread需要处理的列块范围
    int end_block = start_block + block_per_thread;
    //分配余数
    /*比如: 1280/6余2,将余出来的两个block分配给前两个线程 */
    if(p < remainer_blocks) end_block ++;
    //确保不越界
    end_block = std::min(end_block,total_blocks);
    //启动线程处理指定的列块范围
      threads.emplace_back([=, &image, &image_blur]() {
            for(int block = start_block;block<end_block;block++){
```

```
                //实际索引
                int j = block*8;
                for (int i = 0; i < X; i++) {
                    __m256i acc = _mm256_setzero_si256();
                for(int t = -BLUR_SIZE;t<=BLUR_SIZE;t++){
                    if(i+t >= 0 && i+t<X){
                    __m256i data = _mm256_loadu_si256(
                        (__m256i const*)&image[(i + t) * Y + j]);
                                acc = _mm256_add_epi32(acc, data);
                    }
                }
                __m256 acc_float = _mm256_cvtepi32_ps(acc);
                acc_float = _mm256_mul_ps(acc_float, scale_vec);
                __m256i result = _mm256_cvttps_epi32(acc_float);
                _mm256_storeu_si256((__m256i*)&image_blur[i * Y + j],
result);
                }
            }
      });
      //这一行的作用是更新下一行的起始位置
      start_block = end_block;
    }

    //等待所有线程完成
     for (auto& thread : threads) {
        thread.join();
    }

}
```

结果：比调用OpenMP还要快一点，达到了67，感觉两种方法差不多

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ g++ -O0 -mavx2  -o opt_blur
opt_blur.cpp
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ ./opt_blur
Original: 6628ms
SIMD: 98ms
Original/SIMD加速比:67
```

目前多线程并行化到这里就可以了，因为1280个块已经远远超出了CPU的核心数，无需再用行分块分得更细致了，如果在GPU中，有足够多的SM就可以尝试分的更细致一点

这里我还得出了一个结论，就是线程级/进程级的并行化优化空间其实比指令级的优化空间大多了，一个多线程就可以提升这么多，比刚才的向量化效果多多了

4. prefetch：预取数据（由于我做了点下面的题回来做的这里，所以prefetch的具体逻辑不再展开）

   即在前一轮数据计算的基础上再取出一点数据为下一轮计算做准备，隐藏了延迟

```
//SIMD向量化+循环分块优化+手写多线程（列循环并行化）+prefetch
void blur_y_simd(int const image[X * Y], int image_blur[X * Y]) {
  const __m256 scale_vec = _mm256_set1_ps(1.0f/Two_BLUR_SIZE_PLUS_ONE);
```

```
  //获得可用的CPU核心数
  unsigned int num_threads = std::thread::hardware_concurrency();
  if (num_threads == 0) num_threads = 4; // 默认使用4个线程

  int total_blocks = Y/8;
  //因为这个例子是10240/8,所以一共1280个块
  //下面是每个线程需要处理的块
  int block_per_thread = total_blocks / num_threads;
  //下面是平分不完的blocks
  int remainer_blocks = total_blocks % num_threads;

  //创建线程向量
  std::vector<std::thread> threads;

  //启动线程
  int start_block = 0;
  //p为当前线程的索引
  for(unsigned int p =0;p<num_threads;p++){
    //计算当前thread需要处理的列块范围
    int end_block = start_block + block_per_thread;
    //分配余数
    /*比如: 1280/6余2,将余出来的两个block分配给前两个线程 */
    if(p < remainer_blocks) end_block ++;
    //确保不越界
    end_block = std::min(end_block,total_blocks);
    //启动线程处理指定的列块范围
      threads.emplace_back([=, &image, &image_blur]() {
            for(int block = start_block;block<end_block;block++){
                //实际索引
                int j = block*8;
                for (int i = 0; i < X; i++) {
                    __m256i acc = _mm256_setzero_si256();
                for(int t = -BLUR_SIZE;t<=BLUR_SIZE;t++){
                    if(i+t >= 0 && i+t<X){
                        //666加4说是经验值
                    _mm_prefetch((const char*)&image[(i + t + 4) * Y +
j], _MM_HINT_T0);

                    __m256i data = _mm256_loadu_si256(
                        (__m256i const*)&image[(i + t) * Y + j]);
                    acc = _mm256_add_epi32(acc, data);
                    }
                }
                __m256 acc_float = _mm256_cvtepi32_ps(acc);
                acc_float = _mm256_mul_ps(acc_float, scale_vec);
                __m256i result = _mm256_cvttps_epi32(acc_float);
                _mm256_storeu_si256((__m256i*)&image_blur[i * Y + j],
result);
                }
            }
      });
      //这一行的作用是更新下一行的起始位置
      start_block = end_block;
```

```
    }

    //等待所有线程完成
     for (auto& thread : threads) {
        thread.join();
    }

}
```

在这里加了一行

```
#include <xmmintrin.h>
for(int t = -BLUR_SIZE;t<=BLUR_SIZE;t++){
                    if(i+t >= 0 && i+t<X){
                        //666加4说是经验值
                    _mm_prefetch((const char*)&image[(i + t + 4) * Y +
j], _MM_HINT_T0);
```

加速比来到了84

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ g++ -O0 -mavx2  -o opt_blur
opt_blur.cpp
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ ./opt_blur
Original: 6695ms
SIMD: 79ms
Original/SIMD加速比:84
```

5. 算法优化：滑动窗口

滑动窗口：详情可以去看代码随想录，思想就是每轮循环只更新不同的，因为其他的都保存在sum里面不用动

```
//SIMD向量化+循环分块优化+手写多线程（列循环并行化）+prefetch+滑动窗口
void blur_y_simd(int const image[X * Y], int image_blur[X * Y]) {
  const __m256 scale_vec = _mm256_set1_ps(1.0f/Two_BLUR_SIZE_PLUS_ONE);

  //获得可用的CPU核心数
  unsigned int num_threads = std::thread::hardware_concurrency();
  if (num_threads == 0) num_threads = 4; // 默认使用4个线程

  int total_blocks = Y/8;
  //因为这个例子是10240/8,所以一共1280个块
  //下面是每个线程需要处理的块
  int block_per_thread = total_blocks / num_threads;
  //下面是平分不完的blocks
  int remainer_blocks = total_blocks % num_threads;

  //创建线程向量
  std::vector<std::thread> threads;

  //启动线程
  int start_block = 0;
  //p为当前线程的索引
```

```
  for(unsigned int p =0;p<num_threads;p++){
    //计算当前thread需要处理的列块范围
    int end_block = start_block + block_per_thread;
    //分配余数
    /*比如: 1280/6余2,将余出来的两个block分配给前两个线程 */
    if(p < remainer_blocks) end_block ++;
    //确保不越界
    end_block = std::min(end_block,total_blocks);

    // 复制起止值, 避免 lambda 捕获问题
    int thread_start = start_block;
    int thread_end = end_block;

    //启动线程处理指定的列块范围
      threads.emplace_back([thread_start, thread_end, &image,
&image_blur,scale_vec]() {
            for(int block = thread_start;block<thread_end;block++){
                //实际索引
                int j = block*8;

                // --- 初始化 i = 0 的滑动窗口 sum ---
                __m256i acc = _mm256_setzero_si256();
                for (int t = -BLUR_SIZE; t <= BLUR_SIZE; ++t) {
                    int idx = 0 + t;
                    if (idx >= 0 && idx < X) {
                        // 经验性 prefetch（预取未来几个元素）
                        int pf_idx = idx + 4;
                        if (pf_idx < X) {
                            _mm_prefetch((const char*)&image[pf_idx * Y
+ j], _MM_HINT_T0);
                        }
                        __m256i data = _mm256_loadu_si256((__m256i
const*)&image[idx * Y + j]);
                        acc = _mm256_add_epi32(acc, data);
                    }
                }

                // 存储 i=0 的结果
                {
                    __m256 acc_f = _mm256_cvtepi32_ps(acc);
                    acc_f = _mm256_mul_ps(acc_f, scale_vec);
                    __m256i res = _mm256_cvttps_epi32(acc_f);
                    _mm256_storeu_si256((__m256i*)&image_blur[0 * Y +
j], res);
                }

                // --- 对 i = 1 .. X-1 使用滑动窗口更新:  acc = acc -
remove + add ---
                for (int i = 1; i < X; ++i) {
                    int remove_idx = i - BLUR_SIZE - 1; // 要移除的旧元素
索引
                    int add_idx = i + BLUR_SIZE;        // 要加入的新元素
索引
```

```
                    __m256i sub = _mm256_setzero_si256();
                    __m256i add = _mm256_setzero_si256();

                    if (remove_idx >= 0) {
                        sub = _mm256_loadu_si256((__m256i
const*)&image[remove_idx * Y + j]);
                    }
                    if (add_idx < X) {
                        // 预取将来可能用到的元素（经验值 +4）
                        int pf_idx = add_idx + 4;
                        if (pf_idx < X) {
                            _mm_prefetch((const char*)&image[pf_idx * Y
+ j], _MM_HINT_T0);
                        }
                        add = _mm256_loadu_si256((__m256i
const*)&image[add_idx * Y + j]);
                    }

                    // acc = acc - sub + add
                    acc = _mm256_add_epi32(_mm256_sub_epi32(acc, sub),
add);

                    // 转浮点乘 scale，并存回
                    __m256 acc_f = _mm256_cvtepi32_ps(acc);
                    acc_f = _mm256_mul_ps(acc_f, scale_vec);
                    __m256i res = _mm256_cvttps_epi32(acc_f);
                    _mm256_storeu_si256((__m256i*)&image_blur[i * Y +
j], res);
                }


            }
        });
        //这一行的作用是更新下一行的起始位置
        start_block = end_block;
    }

    //等待所有线程完成
     for (auto& thread : threads) {
        thread.join();
    }

}
```

加上滑动窗口后，加速比来到了96

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ g++ -O0 -mavx2  -o opt_blur
opt_blur.cpp
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ ./opt_blur
Original: 6763ms
SIMD: 70ms
Original/SIMD加速比:96
```

用O3跑

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ g++ -O3 -mavx2  -o opt_blur
opt_blur.cpp
lyjy@Lenovo:~/MlSys_LYJY/Part2_5$ ./opt_blur
Original: 2755ms
SIMD: 70ms
Original/SIMD加速比:39
```

通过O0和O3优化版本的时间差不多的对比，说明我自己写的版本已经优化程度很高了

6. 异步内存拷贝相比同步拷贝有什么优势

- 同步内存拷贝：

  CPU亲自发出拷贝指令，并等待整个拷贝过程完成，在此期间，CPU核心会被占用，不能执行该线程上的任何其他指令

- 异步内存拷贝：发出拷贝指令后，立即返回并继续执行后续的，不依赖这个拷贝结果的指令
  - 优势：吞吐量很大，提高CPU利用率，提高相应速度，可以更好的实现加速（在此基础上）

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

- 理论性能：在你的硬件上，**完美优化**的代码所能达到的**最高浮点运算速率**，单位**FLOPS** (Floating-Point Operations per Second)。

  目标就是计算出这个数字

  Roofline Model模型：计算屋顶和内存屋顶

  1. 计算复杂度--计算得到目标矩阵需要多少次浮点运算：

     对于C[M,N] = A[M,K] * B[K,N]

     得到C中一个元素需要K次乘法，K次加法

     得到C矩阵需要**2×M×N×K次乘法，这就是总浮点运算次数FLOPs**

  2. 单核心CPU硬件峰值性能分析--单核心CPU在理论上每秒能完成多少次浮点运算

     结果=CPU核心频率×每个时钟周期的浮点运算次数

     - CPU核心频率（cycles/second）：每秒有几个时钟周期
     - 每个时钟周期的浮点运算次数（FLOPs/cycle）：明确两个量，向量位宽，有几个FMA单元

       比如AVX2的向量位宽为256,即可以容纳8个float

       现代CPU通常有两个FMA单元，一个FMA能在一个周期内完成2次浮点运算

       所以一个FMA单元一个周期可以完成8×2=16 FLOPs/cycles（因为一条FMA的执行就是并行对这8个数据元素每个元素都执行一次乘加操作）

       两个就是32 FLOPs/cycles

3. 内存带宽分析--决定数据理论供给能力：

   理论峰值带宽（GB/s） = 传输速率（MT/s）×总线宽度（bit）×通道数（/8）

   除以8是为了转换成字节，如果已经是字节就没有这一步

4. 计算访存比(计算目标矩阵所有的计算量/三个矩阵占用的字节数)（这里拿float做的例子，所以×4）

   访存比 = total FLOPs / total Bytes

   访存比 = （2MNK）/4×（M×K+K×N+M×N）

5. 建立性能模型并确定瓶颈

   比较峰值计算性能与（访存比×峰值带宽）

   如果访存比×峰值带宽小，即性能被内存带宽限制

   如果峰值计算性能小，即性能被计算性能限制

   结论：

   - 对于 `N < 32` 的小矩阵，我们的基础实现是 **内存瓶颈 (Memory-Bound)**。理论性能上限为 `Perf = (N/6) * 30 GB/s`。
   - 对于 `N >= 32` 的大矩阵，我们的基础实现从理论上讲应该是 **计算瓶颈 (Compute-Bound)**，理论性能上限应接近 `160 GFLOPS`。

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

   SIMD并行化（使用了FMA）+横向求和（用了比招新题上更靠谱的方法）+循环分块+手写多线程+转置

   转置矩阵：只用了循环分块

   性能数据：

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_6$ g++ -O3 -march=native -mavx2 -mfma -pthread -o
gemm_in_cpu gemm_in_cpu.cpp
lyjy@Lenovo:~/MlSys_LYJY/Part2_6$ ./gemm_in_cpu
矩阵尺寸: 1024 x 1024 x 1024
朴素版本时间: 2138 ms
优化版本时间: 10 ms
加速比: 213
```

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...

   foo用了2个线程，bar就是普通的加法

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_7$ g++ -O0 -o compare compare.cpp
lyjy@Lenovo:~/MlSys_LYJY/Part2_7$ ./compare
Sum foo : 1000000000
foo执行时间: 716ms
Sum bar : 1000000000
bar执行时间: 377ms
```

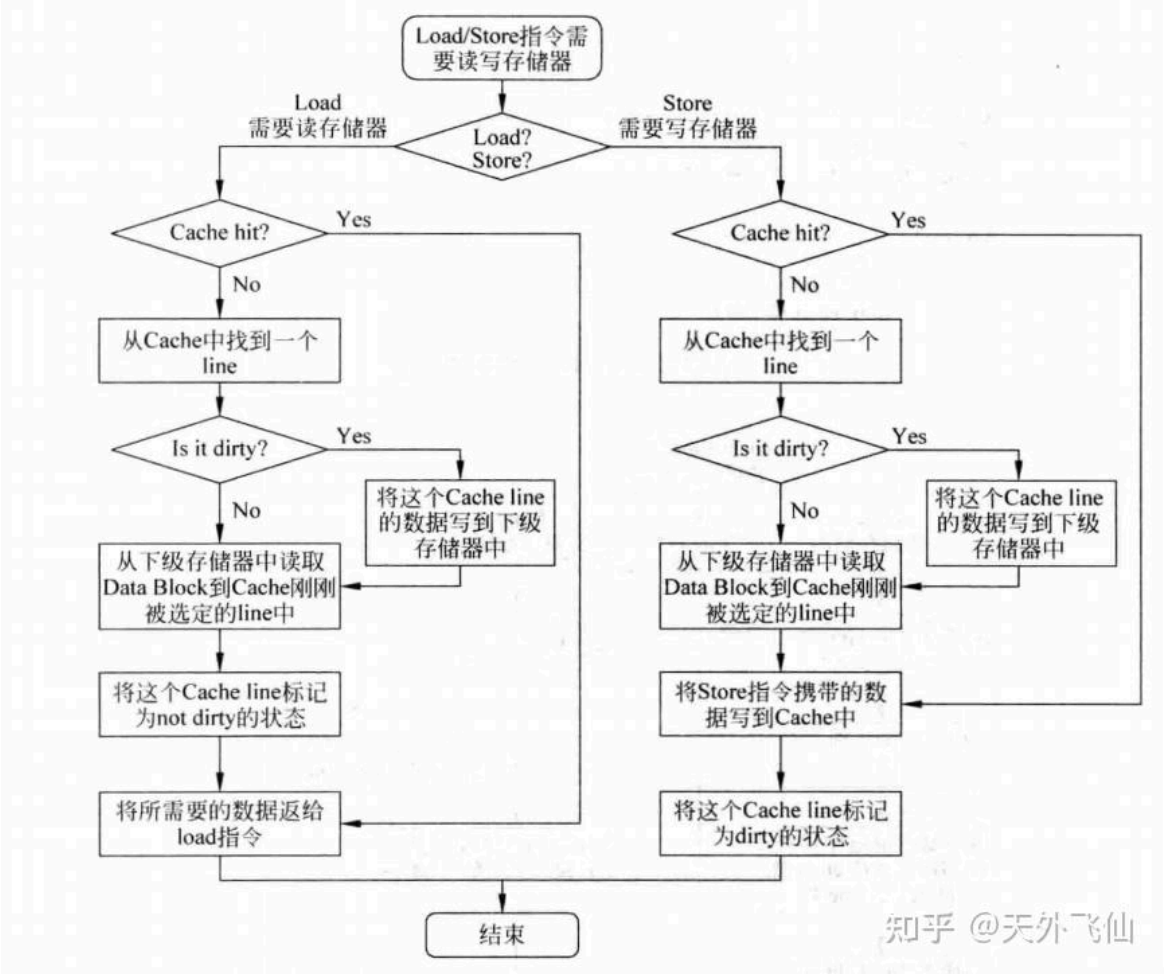
   竟然bar更快一点

   原因：多线程中存在这样一个问题--**伪共享（False Sharing）**

- 结构体的存储：首先需要明白结构体的成员是在内存中连续存储的，而现代CPU一个cache的缓存行一般是64字节，远远超过两个int的8字节，所以**很可能a，b位于同一缓存行（cache line）根本原因在这里**
- 涉及的cache的写入方法：写回+写分配

  简单复习：就是当CPU修改cache中的数据的时候，只**标记缓存行（cache line）**为Dirty，而不立即写回主存，只有在再次读取这个数据的时候，才会把这个数据再次写回主存中

  这里针对的下图的流程是store->cache hit->修改后标注为dirty->load->cache miss->dirty->写入主存->再读取标注为not dirty->返还



- 原因：所以这里的原因就明晰了，就是两个线程并行，交错的修改缓存行的数据，而又交错的读取缓存行，使缓存行不断cache miss，导致一直从主存和cache写写写，从而增加延迟

  > 在伪共享场景中，当线程1修改`a`时，它会使整个缓存行（包含`a`和`b`）变为脏数据，并通知其他核心使它们的缓存副本无效。线程2修改`b`时，同样会触发无效化操作。这样，两个线程频繁地使对方的缓存无效，导致缓存行在核心之间不断同步，从而增加延迟和降低性能。

  反而是bar函数串行不会发生这种问题，因为先线程1一直写，后线程2一直写，最后sum的时候读取在写到主存，这样就没这么多延迟

解决方法：缓存行对齐（Cache Line Alignment），来确保a，b位于不同的缓存行之中

填充剩余的缓冲行即可解决这个问题

```cpp
// 假设缓存行大小为64字节
constexpr size_t CACHE_LINE_SIZE = 64;

struct data {
  alignas(CACHE_LINE_SIZE) int a = 0;  // 对齐到缓存行
  char padding[CACHE_LINE_SIZE - sizeof(int)]; // 填充剩余空间
  alignas(CACHE_LINE_SIZE) int b = 0;  // 对齐到下一个缓存行
};
```

发现确实快了，说明确实是这个原因

```
lyjy@Lenovo:~/MlSys_LYJY/Part2_7$ g++ -O0 -o compare compare.cpp
lyjy@Lenovo:~/MlSys_LYJY/Part2_7$ ./compare
Sum foo : 1000000000
foo执行时间: 193ms
Sum bar : 1000000000
bar执行时间: 363ms
```