

I WriteUp

完整代码：[XyeaOvO/ZXHPC2025](#)

0. 环境

IDE：Cursor

链接方式（需要开北邮VPN[北京邮电大学ATrust VPN使用指南-北京邮电大学信息化技术中心](#)）：

```
ssh username@10.160.16.3 -p 20674
```

组织方式：一题一文件夹

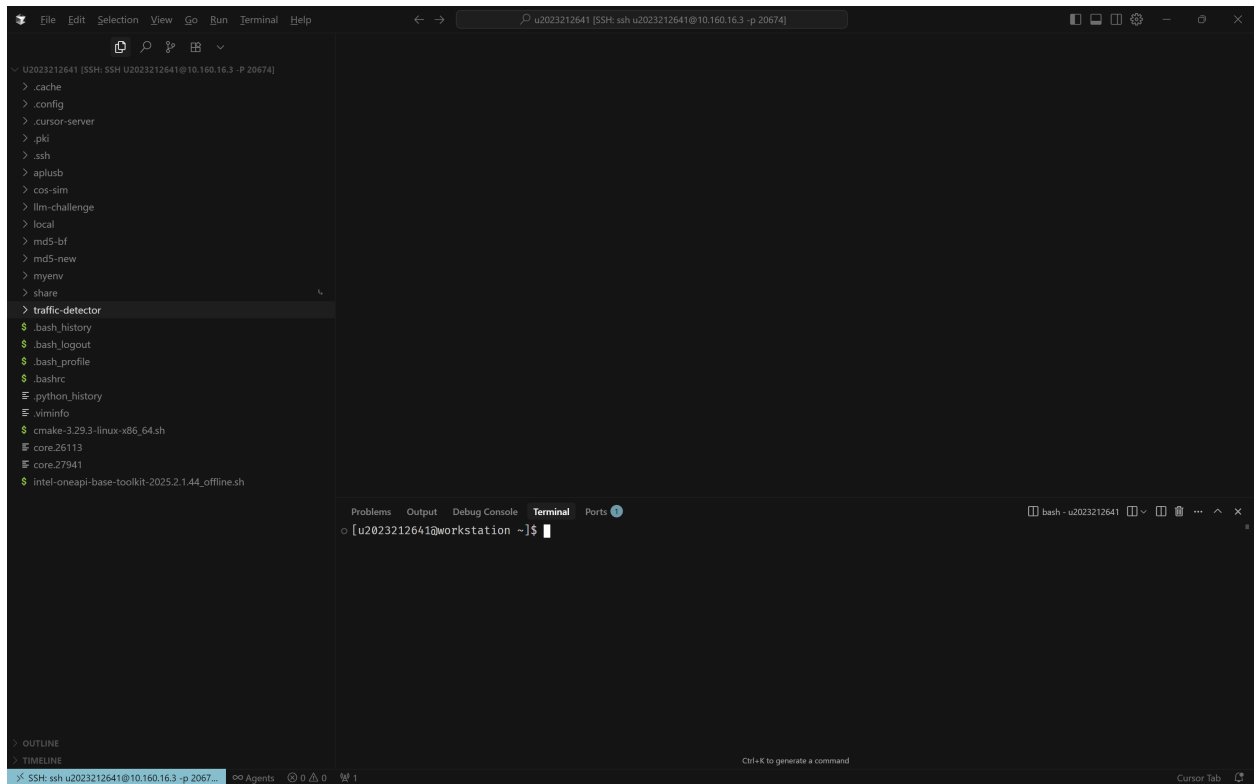


image.png

提交方式：

```
sbatch submit.sh
```

查看可用模块：

```
[u2023212641@workstation cos-sim]$ module avail
```

```
----- /opt/app/spack/share/spack/modules/linux-centos7-haswell -----  
-----
```

```
autoconf-2.69-gcc-4.8.5-6k6kik7          libxml2-2.9.10-gcc-4.8.5-3foymu4  
autoconf-archive-2019.01.06-gcc-4.8.5-7rxz2yv m4-1.4.18-gcc-4.8.5-7x2wh2t  
automake-1.16.2-gcc-4.8.5-ipyg4ha        mpc-1.1.0-gcc-4.8.5-g6zd7ob
```

```

binutils-2.34-gcc-4.8.5-2csi6vr      mpc-1.1.0-gcc-4.8.5-kv3zuys
bzip2-1.0.8-gcc-4.8.5-ersrl36        mpfr-3.1.6-gcc-4.8.5-nol4vkt
diffutils-3.7-gcc-4.8.5-jknorwe      mpfr-4.0.2-gcc-4.8.5-kluqbcj
gcc-10.1.0-gcc-4.8.5-2new4ox         ncurses-6.2-gcc-4.8.5-tbpd5z4
gcc-7.5.0-gcc-4.8.5-of6wn6o         perl-5.30.2-gcc-4.8.5-uay4u7v
gdbm-1.18.1-gcc-4.8.5-7xh2soi        pkgconf-1.6.3-gcc-4.8.5-2qrpqpd
gettext-0.20.2-gcc-4.8.5-kapb6qj    pkgconf-1.7.3-gcc-4.8.5-z3r4unw
gmp-6.1.2-gcc-4.8.5-zn55wh7         readline-8.0-gcc-4.8.5-3jeiguw
isl-0.18-gcc-4.8.5-igs522o          tar-1.32-gcc-4.8.5-v3iynan
isl-0.21-gcc-4.8.5-ikicpxe          texinfo-6.5-gcc-4.8.5-fjg3jyt
libiconv-1.16-gcc-4.8.5-qazxaa4     xz-5.2.5-gcc-4.8.5-rcyjfkv
libsigsegv-2.12-gcc-4.8.5-ymriiur   zlib-1.2.11-gcc-4.8.5-pkmj6e7
libtool-2.4.6-gcc-4.8.5-fzl2npj     zstd-1.4.5-gcc-4.8.5-3boiaus

```

```

----- /usr/share/Modules/modulefiles -----

```

```

-----

```

```

dot          module-git  module-info modules      null          use.own

```

```

----- /opt/app/modulefiles -----

```

```

-----

```

```

feko/2017          gcc/10.1.0      gcc/4.8.5      gcc/6.5.0      gcc/8.4.0
freesurfer/7.4.1  gcc/4.4.7      gcc/5.5.0      gcc/7.5.0      gcc/9.3.0

```

lscpu:

```

1  =====
2  | | | | Running lscpu
3  =====
4  Architecture:          x86_64
5  CPU op-mode(s):       32-bit, 64-bit
6  Byte Order:           Little Endian
7  CPU(s):                40
8  On-line CPU(s) list:  0-39
9  Thread(s) per core:   1
10 Core(s) per socket:   20
11 Socket(s):            2
12 NUMA node(s):         2
13 Vendor ID:             GenuineIntel
14 CPU family:            6
15 Model:                 85
16 Model name:            Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
17 Stepping:              4
18 CPU MHz:               1000.047
19 CPU max MHz:           2401.0000
20 CPU min MHz:           1000.0000
21 BogomIPS:              4800.00
22 Virtualization:        VT-x
23 L1d cache:             32K
24 L1i cache:             32K
25 L2 cache:              1024K
26 L3 cache:              28160K
27 NUMA node0 CPU(s):     0-19
28 NUMA node1 CPU(s):     20-39

```

a-plus-b

代码

aplusb.py:

```
import sys

def main():
    input_data = sys.stdin.read().strip()
    A, B = map(int, input_data.split())
    print(A + B)

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

submit.sh:

```
#!/bin/bash
#SBATCH --job-name=a_plus_b      # 任务名称
#SBATCH --nodes=1                # 申请 1 个节点
#SBATCH --ntasks=1               # 1 个任务
#SBATCH --cpus-per-task=1        # 每个任务使用1个核心
#SBATCH --time=00:05:00          # 时间限制 5 分钟
#SBATCH --mem=16G                #内存申请16G
#SBATCH --exclusive              #独占节点
#SBATCH --partition=c003t        # 提交到 c003t 分区
#SBATCH --output=aplusb_%j.out   # 标准输出文件（%j 会被替换为任务ID）
#SBATCH --error=aplusb_%j.err    # 标准错误文件

# 运行评分器
zxscorer "https://hpci.chouhsing.org/problems/a-plus-b/" --token="7c699a11-8c28-5dc7-b27d-67def56181af" -- python3 aplusb.py
```

llm-challenge

题目复述

请选择合适的 LLM 模型进行推理，完成给定的100道测试题。优化一个双目标函数：总运行时间 (T) 与正确答案数 (C)。

- **时间分 (S_1):**

$$S_1 = \begin{cases} 100 & , T \leq 0.5 \text{ 分钟} \\ 100 \left(\log \left(\frac{T}{30} \right) / \log \left(\frac{0.5}{30} \right) \right) & , 0.5 < T < 30 \text{ 分钟} \\ 0 & , T \geq 30 \text{ 分钟} \end{cases}$$

- **正确率分 (S_2):**

$$S_2 = \frac{100}{65} \max(C - 35, 0)$$

- **最终得分:** $S = \sqrt{S_1 S_2}$

推理框架

考虑到题目环境为 **Linux x86 CPU-only**，这里我选用 llama.cpp。

优势：

- **高性能:** 纯 C/C++ 实现，针对 CPU 进行了深度优化（如 AVX2 指令集）。
- **GGUF 格式:** 支持多种量化等级的 GGUF 模型，便于在模型大小、速度和精度之间做权衡。
- **功能强大:** 自带 server 功能，支持并行处理、流式生成，并支持 **GBNF (GGML BNF)** 语法来约束模型输出。

image-1.png

选用commit版本:

```
• [u2023212641@workstation llama.cpp]$ git log -1
commit 3007baf201e7ffcd17dbdb0335997fa50a6595b
Author: Georgi Gerganov <gggerganov@gmail.com>
Date: Mon Aug 18 18:11:44 2025 +0300

readme : update hot topics (#15397)
```

image-3.png

安装:

下载:

```
git clone https://github.com/ggml-org/llama.cpp.git
cd ~/llm-challenge/llama.cpp
git checkout 3007baf201e7ffcd17dbdb0335997fa50a6595b
```

编译:

错误示范:

在旧版本的llama.cpp中可以使用:

```
make -j$(nproc)
```

但是新版本会报错：

```
Makefile:2: *** The Makefile build is deprecated. Use the CMake build instead. For
more details, see [https://github.com/ggml-org/llama.cpp/blob/master/docs/build.md]
(https://www.google.com/url?sa=E&q=https%3A%2F%2Fgithub.com%2Fggml-
org%2Fllama.cpp%2Fblob%2Fmaster%2Fdocs%2Fbuild.md). Stop.
```

正确示范：

```
module load gcc/8.4.0
```

下载安装cmake：

```
# 回到您的主目录，或者任何您想存放下载文件的地方
cd ~

# 下载适用于 x86-64 Linux 的 CMake 二进制文件（这是一个常用版本）
# 注意：这个链接可能会变，但通常这个格式是稳定的
wget https://github.com/Kitware/CMake/releases/download/v3.29.3/cmake-3.29.3-linux-
x86_64.sh

# 运行安装脚本
bash cmake-3.29.3-linux-x86_64.sh --prefix=$HOME/local/cmake --skip-license

# 将 cmake 的 bin 目录添加到当前终端会话的 PATH 中
export PATH=$HOME/local/cmake/bin:$PATH
# 或者你可以 echo 'export PATH=$HOME/local/cmake/bin:$PATH' >> ~/.bashrc

#验证
cmake --version
```

修改CMakeLists.txt文件：

```
# 回到 llama.cpp 目录
cd /home/u2023212641/llm-challenge/llama.cpp

vim ggml/src/CMakeLists.txt
```

修改以下两行代码

```
226     if (CMAKE_SYSTEM_NAME MATCHES "Linux")
227     |         target_link_libraries(ggml PRIVATE dl stdc++fs)
228     endif()
229     |
```

image-4.png

```
392
393 | target_link_libraries(ggml-base PRIVATE Threads::Threads stdc++fs)
394
395 | find_library(MATH_LIBRARY m)
```

image-5.png

编译 llama.cpp:

```
# 创建 build 目录并进入
mkdir -p build
cd build

# 运行 CMake 配置
cmake .. -DLLAMA_CURL=OFF

# 运行编译
cmake --build . -- -j$(nproc)
```

如果中途出错:

```
cd /home/u2023212641/llm-challenge/llama.cpp
rm -rf build
mkdir build
cd build
cmake .. -DLLAMA_CURL=OFF
cmake --build . -- -j$(nproc)
```

验证:

确认 llama-server 的位置:

```
ls -l /home/u2023212641/llm-challenge/llama.cpp/build/bin/llama-server
```

模型选择

一个优秀、先进的模型是正确率和效率的基石。最后我选择的是微软的 **Phi-4-mini-instruct-Q4_K_M** GGUF 量化模型。

选择原因有:

1. 模型发布时间较新: [unsloth/Phi-4-mini-instruct-GGUF · Hugging Face](#)
2. 模型大小合适: 3.8B
3. instruct模型, 适合回答综合问题, Phi-4-mini-reasoning只针对数学问题
4. 全英文题目, Qwen等经过中文调优的并无意义
5. Q4_K_M相比Q5_K_M速度实测翻番, 正确率无显著下降

6. 实测所得（

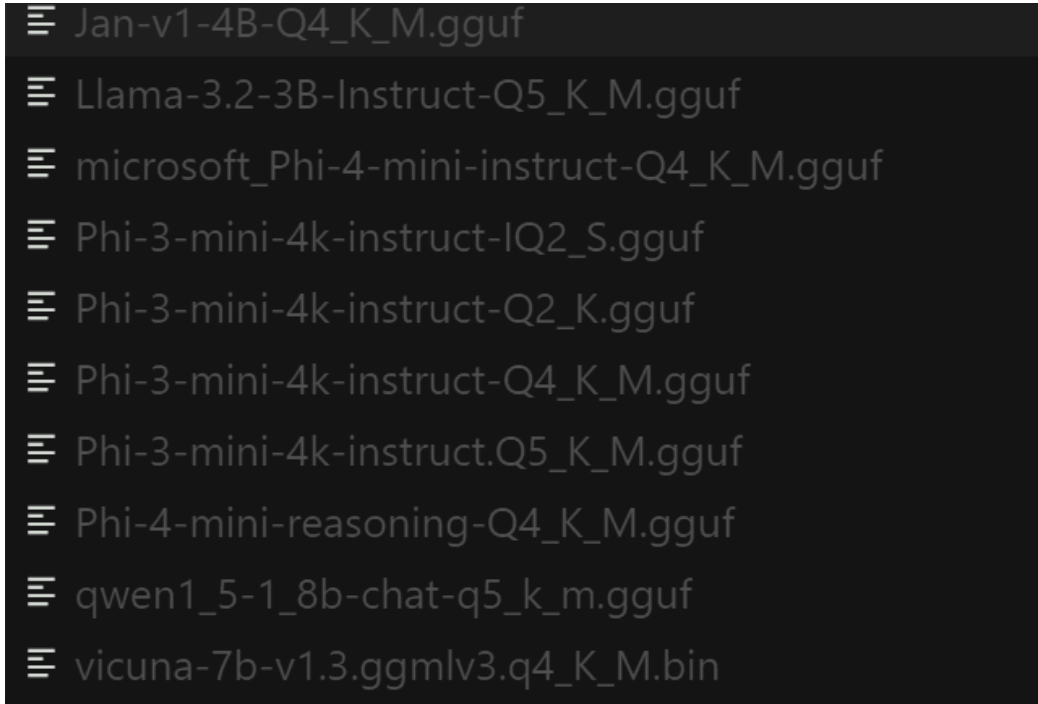


image-6.png

下载教程：

登录Huggingface[unsloth/Phi-4-mini-instruct-GGUF · Hugging Face](#)，选择模型下载到PC本地：

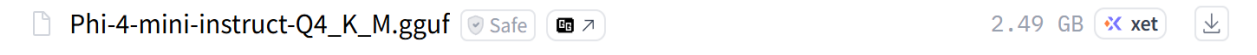


image-7.png

使用winSCP上传到服务器 `${LLAMA_CPP_DIR}/models/` 目录下。

性能优化

a. 客户端-服务器架构

`zxscorer` 计算的是从 `<command>` 开始执行到结束的时间。

本题的 `submit.sh` 脚本采用了客户端-服务器架构：

1. 在 `zxscorer` 运行之前，使用 `&` 将 `llama-server` 进程放到后台运行。
2. 通过 `curl` 循环检测服务器的 `/health` 端点，确保模型已完全加载并准备好接收请求。
3. 当服务器就绪后，才执行 `zxscorer`，`zxscorer` 内部调用我们的 `solver_client.py`。

b. Prompt

直接输出1token的答案，放弃思维链，否则时间成本难以接受。

同时，GBNF 强制模型在第一步生成时必须选择 A/B/C/D 四个 token 中的一个。

```
# ===== GBNF 和推理参数 =====  
GBNF_GRAMMAR = r'''  
root ::= ("A" | "B" | "C" | "D")
```

```
'''
INFERENCE_PARAMS = {
    "n_predict": 1,
    "temperature": 1,
    "top_p": 0.95,
    "stop": ["<|end|>"],
    "grammar": GBNF_GRAMMAR,
}
```

```
# ===== 2. Prompt=====
PROMPT_TEMPLATE = """<|user>
The following is a multiple-choice question with options A, B, C, D. Select the
correct answer and respond with ONLY the letter (A, B, C, or D). No ANY explanation.

Question:
{q}
<|end|>
<|assistant>
"""
```

c. 参数调优

submit_and_run.sh

- `-t 40 -tb 40` : 将线程数设置为 40
- `-c 1024` : 上下文长度
- `--mlock` : 将模型锁定在内存中
- `-ctk q8_0` & `-ctv q8_0` : 压缩 KV 缓存, 同时必须要求开启: `--flash-attn`
- `--numa distribute` : 在多 NUMA 节点的服务器上, 优化内存分配策略。

d. 并行优化

经过测试, 同步调整CONCURRENT_REQUESTS=5和CONTEXT_SIZE=4096, 似乎没啥大用。

若只是一味调高CONCURRENT_REQUESTS, 则每个问题能分到的上下文token数会急剧减少, 速度迅速提升, 准确率迅速下降。

代码

最高得分: 70.97pts

solver_client.py:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import sys
import requests
from concurrent.futures import ThreadPoolExecutor, as_completed
```



```

# ===== 1. 路径与参数（客户端配置） =====
# 假设 llama.cpp 服务器正在此地址和端口上运行
SERVER_HOST = "127.0.0.1"
SERVER_PORT = 8080
# 并发请求数，应与服务器启动时的 --parallel 参数保持一致
CONCURRENT_REQUESTS = 5

# ===== GBNF 和推理参数 =====
# GBNF (Guided Backus-Naur Form) 语法
# 这个语法强制模型只能生成 "A", "B", "C", "D" 四个字符中的一个。
GBNF_GRAMMAR = r'''
root ::= ("A" | "B" | "C" | "D")
'''

# 推理参数
# 这些参数会随每个请求发送给服务器
INFERENCE_PARAMS = {
    "n_predict": 1,          # 只需要预测1个 token（即一个字母）
    "temperature": 1,        # 温度设置为1，允许模型有一定的随机性
    "top_p": 0.95,           # top-p 采样
    "stop": ["<|end|>"],     # 停止符（虽然 n_predict=1 使其作用不大）
    "grammar": GBNF_GRAMMAR, # 使用上面定义的 GBNF 语法来约束输出
}

# ===== 2. Prompt 模板 =====
PROMPT_TEMPLATE = """<|user>
The following is a multiple-choice question with options A, B, C, D. Select the
correct answer and respond with ONLY the letter (A, B, C, or D). No ANY explanation.

Question:
{q}
<|end|>
<|assistant>
"""

# ===== 3. 服务器启动代码已被移除 =====
# 这个脚本是一个纯客户端，它假设服务器已经在后台独立运行。

# ===== 4. 并行处理函数 =====
def solve_question(question_tuple):
    """
    处理单个问题：构建 prompt，发送请求到服务器，并返回结果。
    """
    index, q = question_tuple
    try:
        # 如果问题行为空，直接返回默认答案，避免发送无效请求
        if not q.strip():
            return index, "B"
    
```

```

# 使用模板格式化 Prompt
prompt = PROMPT_TEMPLATE.format(q=q.strip())

# 准备请求体
data = {"prompt": prompt, **INFERENCE_PARAMS}
# 使用 "Connection: close" 避免在大量短连接时出现问题
headers = {"Connection": "close"}

# 向正在运行的 llama.cpp 服务器发送 POST 请求
response = requests.post(
    f"http://{SERVER_HOST}:{SERVER_PORT}/completion",
    json=data,
    timeout=600, # 10分钟超时
    headers=headers
)
# 如果服务器返回错误状态码 (如 4xx, 5xx), 则抛出异常
response.raise_for_status()

# 从 JSON 响应中获取模型生成的内容
full_output = response.json()['content']

# --- 答案提取 ---
answer = full_output.strip().upper()

# 添加一个简单的验证, 以防万一出现意外输出
if answer not in {"A", "B", "C", "D"}:
    print(f"WARN on question {index}: Unexpected output '{answer}'.
Defaulting to 'B'.", file=sys.stderr)
    return index, "B"

return index, answer

except Exception as e:
    # 捕获所有可能的异常 (网络问题、超时、JSON 解析错误等)
    print(f"ERROR on question {index}: {e}", file=sys.stderr)
    # 出错时返回一个默认答案 "B", 确保程序不会中断
    return index, "B"

# ===== 5. 读取输入并并行执行 =====
def main():
    """
    主函数: 从标准输入读取所有问题, 使用线程池并行处理, 并按原始顺序打印结果。
    """
    # 从 stdin 读取全部内容, 并按双换行符分割成问题列表
    questions = sys.stdin.read().strip().split("\n\n")
    results = {} # 使用字典来存储结果, 键为原始索引, 值为答案

    # 为每个非空问题创建一个带索引的元组

```

```

questions_with_indices = list(enumerate(q for q in questions if q.strip()))

# 使用线程池来并行发送请求
with ThreadPoolExecutor(max_workers=CONCURRENT_REQUESTS) as executor:
    print(f"Submitting {len(questions_with_indices)} questions to the running
server...", file=sys.stderr)

    # 提交所有任务到线程池
    future_to_question = {
        executor.submit(solve_question, q_tuple): q_tuple for q_tuple in
questions_with_indices
    }

    try:
        from tqdm import tqdm
        # as_completed 会在任务完成时立即返回 future 对象，实现乱序完成
        progress_iterator = tqdm(as_completed(future_to_question),
total=len(questions_with_indices), file=sys.stderr, desc="Processing questions")
    except ImportError:
        progress_iterator = as_completed(future_to_question)

    # 遍历已完成的任务
    for future in progress_iterator:
        index, answer = future.result()
        results[index] = answer

# ===== 6. 按顺序输出结果 =====
# 确保即使某些问题处理失败，也能为每个原始问题输出一个答案
for i in range(len(questions)):
    print(results.get(i, "B"))

if __name__ == "__main__":
    main()

```

`submit.sh` 脚本：

```

#!/bin/bash

#SBATCH -J llm_challenge_job
#SBATCH -p c003t
#SBATCH -N 1
#SBATCH --cpus-per-task=40
#SBATCH --mem=40G
#SBATCH --exclusive
#SBATCH --time=00:30:00
#SBATCH -o job_%j.out
#SBATCH -e job_%j.err

# --- 1. 设置环境 ---

```

```
echo "Setting up environment..."
source ~/.llm-challenge/.venv/bin/activate
module load gcc/8.4.0

# --- 2. 定义服务器参数 ---
LLAMA_CPP_DIR="/home/u2023212641/llm-challenge/llama.cpp"
SERVER_EXE="${LLAMA_CPP_DIR}/build/bin/llama-server"
MODEL="${LLAMA_CPP_DIR}/models/microsoft_Phi-4-mini-instruct-Q4_K_M.gguf"

SERVER_HOST="127.0.0.1"
SERVER_PORT="8080"
# --- 关键改动：大幅提高并发度 ---
CONCURRENT_REQUESTS=5
CONTEXT_SIZE=4096
NUM_THREADS=${SLURM_CPUS_PER_TASK:-40}

SERVER_CMD=(\
    "$SERVER_EXE" \
    -m "$MODEL" \
    --host "$SERVER_HOST" \
    --port "$SERVER_PORT" \
    -t "$NUM_THREADS" \
    -tb "$NUM_THREADS" \
    -c "$CONTEXT_SIZE" \
    -b 4096 \
    -ctk q8_0 \
    -ctv q8_0 \
    --parallel "$CONCURRENT_REQUESTS" \
    # --kv-unified \
    --numa distribute \
    --timeout 600 \
    --mlock \
    --flash-attn \
)

# --- 3. 启动服务器 ---
echo "Starting llama.cpp server in the background..."
echo "Server command: ${SERVER_CMD[@]}"
# 直接后台运行，日志会进入 SLURM 的输出/错误文件
"${SERVER_CMD[@]}" &
SERVER_PID=$!
echo "Server started with PID: $SERVER_PID"

trap 'echo "Cleaning up server..."; kill -TERM $SERVER_PID; wait $SERVER_PID
2>/dev/null' EXIT

# --- 4. 等待服务器就绪 ---
echo "Waiting for server to become ready..."
MAX_WAIT=300
```

```

SECONDS=0
while true; do
    HTTP_STATUS=$(curl -s -o /dev/null -w "%{http_code}"
http://${SERVER_HOST}:${SERVER_PORT}/health)
    if [ "$HTTP_STATUS" -eq 200 ]; then
        echo "Server is ready!"
        break
    fi
    if [ $SECONDS -ge $MAX_WAIT ]; then
        echo "Server failed to start within $MAX_WAIT seconds. Exiting."
        exit 1
    fi
    sleep 2
    echo -n "."
done

# --- 5. 运行评分程序 ---
echo "Running zxscorer with solver_client.py..."
zxscorer "https://hpci.chouhsing.org/problems/llm-challenge/" \
    -- python ~/llm-challenge/solver_client.py

echo "Scoring finished."

```

cos-sim

题目复述

余弦相似度

- 数学公式：

$$\text{cosine_similarity}(A, B) = \frac{A \cdot B}{\|A\| \|B\|}$$

这个公式可以拆成三部分计算：

1. **向量 A 和 B 的点积 (Dot Product):** $A \cdot B = \sum_{i=1}^D A_i B_i$
2. **向量 A 的模长 (Norm):** $\|A\| = \sqrt{\sum_{i=1}^D A_i^2}$
3. **向量 B 的模长 (Norm):** $\|B\| = \sqrt{\sum_{i=1}^D B_i^2}$

- 计算量分析：

- 对于两个 D 维向量，计算点积需要 D 次乘法和 D-1 次加法。
- 计算一个向量的模长，需要 D 次乘法，D-1 次加法，和 1 次开方。
- 所以，计算一次余弦相似度，大致需要 $(D + D + D) = 3D$ 次乘法， $(D + D) = 2D$ 次加法，和 2 次开方。

任务

题目要求计算 **N 个向量两两之间** 的余弦相似度。

- 如果暴力计算，对于向量 i ，我们需要计算它和向量 $0, 1, \dots, N - 1$ 的相似度。
- 总共需要计算的相似度次数大约是 $N \times N$ 次。
- 总计算量大致是 $O(N^2 \cdot 2D)$ 。

输入输出格式：小端序二进制串

- **二进制串**：我们平时 `cin >> my_var;` 是在读取文本，计算机会把文本（比如 "123"）转换成数字（`int` 类型的 123）。而读取二进制串，是直接从输入流里拷贝内存块。这比文本解析快得多，因为不需要转换。
- **小端序 (Little-Endian)**：这是多字节数据（比如 `int32` 是4字节，`float32` 是4字节）在内存中的存储方式。x86 架构的 CPU（我们日常用的 Intel, AMD 处理器）都是小端序。
 - 举个例子：整数 `0x01020304`（十六进制）。
 - **大端序 (Big-Endian)**：高位字节存放在低地址（像我们读书写字一样）：`01 02 03 04`
 - **小端序 (Little-Endian)**：低位字节存放在低地址（反过来）：`04 03 02 01`
- **为什么题目要强调这个？** 因为如果你的代码运行在非 x86 的大端序机器上，直接读取二进制数据就会出错。但这题限定了 x86 环境，所以我们用 `std::cin.read()` 这样直接读内存块的方式是安全且高效的。

原始代码，得分：3.33pts

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>
#include <cstdint>

float cosine_similarity(const float *a, const float *b, int D) {
    float dot_product = 0.0f;
    float sum_a2 = 0.0f;
    float sum_b2 = 0.0f;
    for (int i = 0; i < D; ++i) {
        dot_product += a[i] * b[i];
        sum_a2 += a[i] * a[i];
        sum_b2 += b[i] * b[i];
    }
    return dot_product / (std::sqrt(sum_a2) * std::sqrt(sum_b2) + 1e-12);
}

int main() {
    uint32_t N, D;
```

```

std::cin.read(reinterpret_cast<char *>(&N), sizeof(N));
std::cin.read(reinterpret_cast<char *>(&D), sizeof(D));
std::vector<float> data(N * D);
std::cin.read(reinterpret_cast<char *>(data.data()), N * D * sizeof(float));

for (int i = 0; i < N; ++i) {
    std::vector<float> cosine_sim(N);
    for (int j = 0; j < N; ++j) {
        cosine_sim[j] = cosine_similarity(data.data() + i * D, data.data() + j *
D, D);
    }
    std::partial_sort(cosine_sim.begin(), cosine_sim.begin() + 5,
cosine_sim.begin() + N, std::greater<float>());
    std::cout.write(reinterpret_cast<char *>(cosine_sim.data() + 1), 4 *
sizeof(float));
}

return 0;
}

```

瓶颈

1. 向量模长被重复计算
2. 单线程执行，浪费多核CPU

50.63pts

预计算所有向量的模长

```

std::vector<float> norms(N);
for (uint32_t i = 0; i < N; ++i) {
    float sum_sq = 0.0f;
    const float *vec = data.data() + i * D;
    for (uint32_t j = 0; j < D; ++j) {
        sum_sq += vec[j] * vec[j];
    }
    norms[i] = std::sqrt(sum_sq);
}

```

使用 OpenMP 并行计算

对于不同的 `i`，计算它与其他向量的相似度这个任务是**完全独立**的，互相不影响。

```

#pragma omp parallel for schedule(dynamic)
for (uint32_t i = 0; i < N; ++i) {
}

```

注意：每个线程计算完自己的结果后，写入到一个大的、预先分配好的 `final_results` 数组的**不同位置** (`final_results.data() + i * 4`)。这避免了“竞争条件”(Race Condition)，即多个线程同时去写同一个内存地址导致数据错乱。

代码

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <stdint>
#include <omp.h> // 引入 OpenMP 头文件

// 计算点积的函数
float dot_product(const float *a, const float *b, int D) {
    float result = 0.0f;
    for (int i = 0; i < D; ++i) {
        result += a[i] * b[i];
    }
    return result;
}

int main() {
    // 关闭 C++ 标准流与 C 标准流的同步，可以提速 I/O
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(NULL);

    // 1. 读取 N 和 D
    uint32_t N, D;
    std::cin.read(reinterpret_cast<char *>(&N), sizeof(N));
    std::cin.read(reinterpret_cast<char *>(&D), sizeof(D));

    // 2. 读取所有向量数据
    std::vector<float> data(N * D);
    std::cin.read(reinterpret_cast<char *>(data.data()), N * D * sizeof(float));

    // 3. 【优化点一】预计算所有向量的模长
    std::vector<float> norms(N);
    for (uint32_t i = 0; i < N; ++i) {
        float sum_sq = 0.0f;
        const float *vec = data.data() + i * D;
        for (uint32_t j = 0; j < D; ++j) {
            sum_sq += vec[j] * vec[j];
        }
        norms[i] = std::sqrt(sum_sq);
    }

    // 准备一个线程安全的输出缓冲区
    std::vector<float> final_results(N * 4);
```



```

// 4. 【优化点二】使用 OpenMP 并行计算
#pragma omp parallel for schedule(dynamic)
for (uint32_t i = 0; i < N; ++i) {
    std::vector<float> cosine_sim(N);
    const float *vec_i = data.data() + i * D;

    for (uint32_t j = 0; j < N; ++j) {
        const float *vec_j = data.data() + j * D;
        float dot_prod = dot_product(vec_i, vec_j, D);
        // 从预计算的数组中直接取模长，避免重复计算
        cosine_sim[j] = dot_prod / (norms[i] * norms[j] + 1e-12f);
    }

    // 找出除自身外最大的4个相似度值
    // partial_sort 将[first, middle)区间的元素排好序，且这里的元素大于等于[middle,
    last)区间的元素
    // 我们需要最大的5个，因为第一个是与自身的相似度(值为1)
    std::partial_sort(cosine_sim.begin(), cosine_sim.begin() + 5,
        cosine_sim.end(), std::greater<float>());

    // 将结果写入输出缓冲区的对应位置
    // `cosine_sim.data() + 1` 跳过了与自身比较的结果
    float* result_ptr = final_results.data() + i * 4;
    std::copy(cosine_sim.data() + 1, cosine_sim.data() + 5, result_ptr);
}

// 5. 从主线程一次性写入所有结果
std::cout.write(reinterpret_cast<char *>(final_results.data()), N * 4 *
    sizeof(float));

return 0;
}

```

下一步前置知识：SIMD 向量化 (SIMD Vectorization)

• 前置知识：SIMD

- SIMD 的全称是 **S**ingle **I**nstruction, **M**ultiple **D**ata，单指令多数数据流。
- 想象一下，普通CPU计算 `a[i] * b[i]` 是一个一个地做乘法。
- 现代 CPU 拥有一组特殊的、更宽的寄存器（比如 128位, 256位, 512位），可以一次性对多个数据执行相同的操作。
- 例如，一个 256 位的 AVX 寄存器可以装下 8 个 `float` (32位) 数据。CPU 提供一条指令，就能同时完成这 8 对 `float` 的乘法，理论上性能提升 8 倍！

• 如何实现 SIMD?

1. 自动向量化 (Auto-vectorization):

- **是什么：** 寄希望于编译器。现代编译器（如 GCC, Clang）非常智能，当你开启优化选项（如 `-O2`, `-O3`）时，它会尝试分析你的循环，并自动将其转换为 SIMD 指令。
- **怎么做：** 在编译命令中加入 `-O3 -mavx2 -mfma`。
 - `-O3`：开启高级别优化。
 - `-mavx2`：告诉编译器可以使用 AVX2 指令集（256位寄存器）。
 - `-mfma`：告诉编译器可以使用 FMA (Fused Multiply-Add) 指令。FMA 可以把 `a*b+c` 这样的一次乘法和一次加法合并成一条指令，延迟更低，精度更高。我们的点积计算 `result += a[i] * b[i]` 正是 FMA 的完美应用场景。
- **优点：** 简单，只需改编译选项。
- **缺点：** 不保证成功。如果循环中有复杂的分支、函数调用或特定的内存访问模式，编译器可能会放弃向量化。

2. 手动向量化 (Intrinsics):

- **是什么：** 使用编译器提供的内建函数 (Intrinsics)，在 C++ 代码里直接调用 SIMD 指令。这给了你对硬件完全的控制。
- **怎么做：** 引入 `<immintrin.h>` 头文件，并用 `_mm256_...` 类型的函数重写 `dot_product`。

67.20pts

实现了 AVX2 版本的点积函数

1. `dot_product_avx`

```
#include <immintrin.h> // 引入AVX/AVX2指令集的头文件

float dot_product_avx(const float *a, const float *b, int D) {
    __m256 sum_vec = _mm256_setzero_ps();
    int i = 0;
    for (; i <= D - 8; i += 8) {
        __m256 a_vec = _mm256_loadu_ps(a + i);
        __m256 b_vec = _mm256_loadu_ps(b + i);
        sum_vec = _mm256_fmadd_ps(a_vec, b_vec, sum_vec);
    }
    // ...
}
```

• 水平求和:

```
__m128 sum128 = _mm_add_ps(_mm256_extractf128_ps(sum_vec, 1),
    _mm256_castps256_ps128(sum_vec));
__m128 sum64 = _mm_hadd_ps(sum128, sum128);
__m128 sum32 = _mm_hadd_ps(sum64, sum64);
float result = _mm_cvtss_f32(sum32);
```

- `_mm256_extractf128_ps`: 将 256 位的 `sum_vec` 拆成高 128 位和低 128 位。
- `_mm_add_ps`: 将高低 128 位相加, 结果是一个 128 位向量 `[s0+s4, s1+s5, s2+s6, s3+s7]`。
- `_mm_hadd_ps`: 水平相加指令。第一次 `hadd` 后, 结果是 `[s0+s4+s1+s5, s2+s6+s3+s7, ...]`。第二次 `hadd` 后, 所有 8 个 float 的和就集中在了结果向量的第一个元素中。
- `_mm_cvtss_f32`: 将结果向量的第一个 float 提取出来。

• 处理“尾巴”:

```
for (; i < D; ++i) {
    result += a[i] * b[i];
}
```

- 处理当 `D` 不是 8 的倍数时的情况

修改求模长部分

```
#pragma omp parallel for schedule(static) // 并行
for (uint32_t i = 0; i < N; ++i) {
    const float *vec = data.data() + i * D;
    // 使用新函数计算 vec 和自己的点积
    float dot_self = dot_product_avx(vec, vec, D);
    norms[i] = std::sqrt(dot_self);
}
```

82.55pts

缓存分块

• 前置知识: CPU 缓存 (Cache)

- CPU 访问内存的速度远慢于其计算速度。为了弥补这个差距, CPU 内置了多级高速缓存 (L1, L2, L3 Cache)。
- 当 CPU 需要数据时, 它会先把内存中包含该数据的一整块 (称为一个 Cache Line, 通常是 64 字节) 加载到缓存中。下次再访问这块数据或其附近的数据时, 就可以直接从飞快的缓存中读取, 这叫**缓存命中 (Cache Hit)**。如果数据不在缓存里, 就得去慢速的主内存读取, 叫**缓存未命中 (Cache Miss)**。
- HPC 的一个核心思想就是: **最大化缓存命中率**。

• 分析我们当前代码的内存访问模式:

```
for (uint32_t i = 0; i < N; ++i) { // 线程 T1 可能在处理 i=0
    const float *vec_i = data.data() + i * D; // vec_i 加载到缓存
    for (uint32_t j = 0; j < N; ++j) {
        const float *vec_j = data.data() + j * D; // 依次加载 vec_0, vec_1, ...,
        vec_{N-1}
        dot_product_avx(vec_i, vec_j, D);
    }
}
```

```

    }
}

```

- 在内层循环 `for j` 中，`vec_i` 的数据会一直被重复使用 `N` 次，它很可能会一直待在缓存里（这是好事）。
- 但是，`vec_j` 每次迭代都会换一个新的。当 `N` 很大时（比如 20000），所有 `N` 个向量的总大小 `N * D * sizeof(float) = 20000 * 4096 * 4 = 327MB`，这远远大于 CPU 的 L3 缓存（通常几十MB）。
- 这意味着，当 `i=0` 的循环跑完，开始跑 `i=1` 的循环时，之前为 `i=0` 加载进来的 `vec_0` 到 `vec_{N-1}` 的数据，大部分都已经被踢出缓存了。在 `i=1` 的循环里，它们需要被重新从主内存加载一遍。这导致了对 `data` 数组的反复、低效的扫描。

• 怎么优化？分块 (Blocking / Tiling)

- 思路：我们不要一次性计算一个 `i` 和所有 `j` 的关系，而是把整个 $N \times N$ 的计算矩阵分成很多个小方块。每次只计算一个小方块，这样可以保证这个小方块需要的数据都能装进缓存里，并被充分利用。

```

cpp // 【核心优化】缓存分块计算点积矩阵 std::vector<float> dot_products(N * N); const int
B = 64; // 定义块大小 // 使用 collapse(2) 让 OpenMP 将两层循环一起并行化 #pragma omp
parallel for schedule(dynamic) collapse(2) for (uint32_t i_block = 0; i_block < N;
i_block += B) { for (uint32_t j_block = 0; j_block < N; j_block += B) { // 计算一个
B x B 大小的点积块 uint32_t i_max = std::min(i_block + B, N); uint32_t j_max =
std::min(j_block + B, N); for (uint32_t i = i_block; i < i_max; ++i) { for
(uint32_t j = j_block; j < j_max; ++j) { dot_products[i * N + j] =
dot_product_avx(data.data() + i * D, data.data() + j * D, D); } } } }

```

其中，块大小 `B` (Block Size) 是一个需要调整的“超参数”。

代码

```

#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <cstdint>
#include <omp.h>
#include <immintrin.h>

float dot_product_avx(const float *a, const float *b, int D) {
    __m256 sum_vec = _mm256_setzero_ps();
    int i = 0;
    for (; i <= D - 8; i += 8) {
        __m256 a_vec = _mm256_loadu_ps(a + i);
        __m256 b_vec = _mm256_loadu_ps(b + i);
        sum_vec = _mm256_fmadd_ps(a_vec, b_vec, sum_vec);
    }
    __m128 sum128 = _mm_add_ps(_mm256_extractf128_ps(sum_vec, 1),
    _mm256_castps256_ps128(sum_vec));
}

```

```

__m128 sum64 = _mm_hadd_ps(sum128, sum128);
__m128 sum32 = _mm_hadd_ps(sum64, sum64);
float result = _mm_cvtss_f32(sum32);
for (; i < D; ++i) {
    result += a[i] * b[i];
}
return result;
}

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(NULL);

    uint32_t N, D;
    std::cin.read(reinterpret_cast<char *>(&N), sizeof(N));
    std::cin.read(reinterpret_cast<char *>(&D), sizeof(D));

    std::vector<float> data(N * D);
    std::cin.read(reinterpret_cast<char *>(data.data()), N * D * sizeof(float));

    std::vector<float> norms(N);
    #pragma omp parallel for schedule(static)
    for (uint32_t i = 0; i < N; ++i) {
        const float *vec = data.data() + i * D;
        norms[i] = std::sqrt(dot_product_avx(vec, vec, D));
    }

    // 【优化】缓存分块计算点积矩阵
    std::vector<float> dot_products(N * N);
    const int B = 64; // 定义块大小，这是一个可以调整的超参数

    // 使用 collapse(2) 让 OpenMP 将两层循环一起并行化，以获得更好的负载均衡
    #pragma omp parallel for schedule(dynamic) collapse(2)
    for (uint32_t i_block = 0; i_block < N; i_block += B) {
        for (uint32_t j_block = 0; j_block < N; j_block += B) {
            // 计算一个 B x B 大小的点积块
            uint32_t i_max = std::min(i_block + B, N);
            uint32_t j_max = std::min(j_block + B, N);
            for (uint32_t i = i_block; i < i_max; ++i) {
                for (uint32_t j = j_block; j < j_max; ++j) {
                    dot_products[i * N + j] = dot_product_avx(data.data() + i * D,
data.data() + j * D, D);
                }
            }
        }
    }

    // 最后一步：并行化计算相似度、排序并收集结果
    std::vector<float> final_results(N * 4);

```

```

#pragma omp parallel for schedule(dynamic)
for (uint32_t i = 0; i < N; ++i) {
    std::vector<float> cosine_sim(N);
    const float norm_i = norms[i];
    for (uint32_t j = 0; j < N; ++j) {
        cosine_sim[j] = dot_products[i * N + j] / (norm_i * norms[j] + 1e-12f);
    }

    std::partial_sort(cosine_sim.begin(), cosine_sim.begin() + 5,
cosine_sim.end(), std::greater<float>());

    float* result_ptr = final_results.data() + i * 4;
    std::copy(cosine_sim.data() + 1, cosine_sim.data() + 5, result_ptr);
}

std::cout.write(reinterpret_cast<char *>(final_results.data()), N * 4 *
sizeof(float));

return 0;
}

```

91.19pts

利用对称性，削减一半计算量

```

#pragma omp parallel for schedule(dynamic)
for (int i_block = 0; i_block < N; i_block += B) {
    for (int j_block = i_block; j_block < N; j_block += B) { // <-- j_block 从
i_block 开始
        // ...
        if (i_block == j_block) {
            // 对角块：内部也利用对称性
            for (int i = i_block; i < i_max; ++i) {
                for (int j = i; j < j_max; ++j) { // <-- j 从 i 开始
                    // ...
                    dot_products[(long)i * N + j] = dp;
                    dot_products[(long)j * N + i] = dp; // 同时填充对称位置
                }
            }
        } else {
            // 非对角块：计算整个 BxB 块
            for (int i = i_block; i < i_max; ++i) {
                for (int j = j_block; j < j_max; ++j) {
                    // ...
                    dot_products[(long)i * N + j] = dp;
                    dot_products[(long)j * N + i] = dp; // 同时填充对称位置
                }
            }
        }
    }
}

```

```
}  
}
```

预计算范数的倒数，将后续的除法变为乘法

```
std::vector<float> inv_norms(N);  
#pragma omp parallel for schedule(static)  
for (uint32_t i = 0; i < N; ++i) {  
    // ...  
    inv_norms[i] = 1.0f / (std::sqrt(norm_sq) + 1e-12f);  
}  
// ...  
cosine_sim_thread_buffer[j] = dot_products[(long)i * N + j] * inv_norm_i *  
inv_norms[j];
```

线程私有缓存

```
// 为每个线程创建一个私有缓存，避免在循环内反复分配内存  
#pragma omp parallel  
{  
    std::vector<float> cosine_sim_thread_buffer(N);  
    #pragma omp for schedule(static)  
    for (uint32_t i = 0; i < N; ++i) {  
        // ... 使用 cosine_sim_thread_buffer ...  
    }  
}
```

91.82pts

内存对齐

将 `dot_product_avx` 中使用的 `_mm256_loadu_ps` 改为 `_mm256_load_ps`。 `u` 代表 "unaligned"，意味着它可以从任意内存地址加载数据。这很方便，但 CPU 在处理跨越某些边界（如 cache line 边界）的非对齐地址时，可能需要额外的操作，带来微小的性能损失。`_mm256_load_ps` 是它的“对齐”版本。它要求内存地址必须是 32 字节（256位）的倍数。如果满足这个条件，数据加载会更直接、更高效。

```
// 使用 _mm_malloc 来保证数据起始地址是 32 字节对齐的。  
float *data = (float *)_mm_malloc((size_t)N * D * sizeof(float), 32);  
// ...  
_mm_free(data); // 使用 _mm_free 来释放
```

循环展开

手动进行循环展开。

96.71pts

从 AVX2 升级到 AVX-512，需要加入编译选项-mavx512f。

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <stdint>
#include <omp.h>
#include <immintrin.h>

// AVX-512 点积函数 + 对齐加载
// 前提是传入的指针 a 和 b 都是 64 字节对齐的
float dot_product_avx512_aligned(const float *a, const float *b, int D) {
    __m512 sum_vec = _mm512_setzero_ps();
    int i = 0;
    for (; i <= D - 16; i += 16) {
        // 使用 _mm512_load_ps, 要求 a+i 和 b+i 地址是 64 字节对齐的
        __m512 a_vec = _mm512_load_ps(a + i);
        __m512 b_vec = _mm512_load_ps(b + i);
        sum_vec = _mm512_fmadd_ps(a_vec, b_vec, sum_vec);
    }
    return _mm512_reduce_add_ps(sum_vec);
}

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(NULL);

    uint32_t N, D;
    std::cin.read(reinterpret_cast<char *>(&N), sizeof(N));
    std::cin.read(reinterpret_cast<char *>(&D), sizeof(D));

    // 【优化1】使用对齐内存分配（64字节对齐以匹配AVX-512）
    // D=4096 是 16 的倍数，所以每个向量的起始地址也自然对齐了。
    float *data = (float *)_mm_malloc((size_t)N * D * sizeof(float), 64);
    std::cin.read(reinterpret_cast<char *>(data), (long)N * D * sizeof(float));

    std::vector<float> inv_norms(N);
    #pragma omp parallel for schedule(static)
    for (uint32_t i = 0; i < N; ++i) {
        float norm_sq = dot_product_avx512_aligned(data + (long)i * D, data +
(long)i * D, D);
        inv_norms[i] = 1.0f / (std::sqrt(norm_sq) + 1e-12f);
    }

    std::vector<float> dot_products(N * N);
    const int B = 64;
```



```

#pragma omp parallel for schedule(dynamic)
for (int i_block = 0; i_block < N; i_block += B) {
    for (int j_block = i_block; j_block < N; j_block += B) {
        int i_max = std::min(i_block + B, (int)N);
        int j_max = std::min(j_block + B, (int)N);

        if (i_block == j_block) {
            for (int i = i_block; i < i_max; ++i) {
                for (int j = i; j < j_max; ++j) {
                    float dp = dot_product_avx512_aligned(data + (long)i * D,
data + (long)j * D, D);

                    dot_products[(long)i * N + j] = dp;
                    dot_products[(long)j * N + i] = dp;
                }
            }
        } else {
            // 【优化2】循环展开 (2次)
            for (int i = i_block; i < i_max; ++i) {
                int j = j_block;
                for (; j <= j_max - 2; j += 2) {
                    const float* vec_i = data + (long)i * D;

                    const float* vec_j0 = data + (long)j * D;
                    float dp0 = dot_product_avx512_aligned(vec_i, vec_j0, D);
                    dot_products[(long)i * N + j] = dp0;
                    dot_products[(long)j * N + i] = dp0;

                    const float* vec_j1 = data + (long)(j + 1) * D;
                    float dp1 = dot_product_avx512_aligned(vec_i, vec_j1, D);
                    dot_products[(long)i * N + (j + 1)] = dp1;
                    dot_products[(long)(j + 1) * N + i] = dp1;
                }
                // 处理剩余的单次迭代
                if (j < j_max) {
                    float dp = dot_product_avx512_aligned(data + (long)i * D,
data + (long)j * D, D);

                    dot_products[(long)i * N + j] = dp;
                    dot_products[(long)j * N + i] = dp;
                }
            }
        }
    }
}

std::vector<float> final_results(N * 4);
#pragma omp parallel
{
    std::vector<float> cosine_sim_thread_buffer(N);

```

```

#pragma omp for schedule(static)
for (uint32_t i = 0; i < N; ++i) {
    const float inv_norm_i = inv_norms[i];
    for (uint32_t j = 0; j < N; ++j) {
        cosine_sim_thread_buffer[j] = dot_products[(long)i * N + j] *
inv_norm_i * inv_norms[j];
    }

    std::partial_sort(cosine_sim_thread_buffer.begin(),
        cosine_sim_thread_buffer.begin() + 5,
        cosine_sim_thread_buffer.end(),
        std::greater<float>());

    float* result_ptr = final_results.data() + (long)i * 4;
    std::copy(cosine_sim_thread_buffer.data() + 1,
        cosine_sim_thread_buffer.data() + 5, result_ptr);
}
}

std::cout.write(reinterpret_cast<char *>(final_results.data()), (long)N * 4 *
sizeof(float));

_mm_free(data); // 使用 _mm_free 来释放对齐内存

return 0;
}

```

100pts

虽然我们的点积函数 `dot_product_avx512_aligned` 内部计算很快，但在外层循环中，我们仍然在反复地从缓存中加载 `vec_i` 的数据。CPU 的计算单元仍然有大量时间在“饿着肚子”等数据从缓存送到寄存器。

寄存器分块和微内核

代码

关键在于：AVX-512 提供了 32 个 zmm 寄存器。这个微内核使用了 4+4=8 个寄存器加载数据，16 个寄存器做累加，充分利用了硬件资源。

这使得 `v_i0` 这个寄存器里的数据被加载一次，但被复用了 4 次（分别与 `v_j0` 到 `v_j3` 计算）。同理，`v_j0` 也被复用了 4 次。

```

inline void micro_kernel_4x4(int i_start, int j_start, int D, int N, const float*
data, float* dot_products) {
    const float* i_ptr0 = data + (long)(i_start + 0) * D;
    const float* i_ptr1 = data + (long)(i_start + 1) * D;
    const float* i_ptr2 = data + (long)(i_start + 2) * D;
    const float* i_ptr3 = data + (long)(i_start + 3) * D;

```

```

const float* j_ptr0 = data + (long)(j_start + 0) * D;
const float* j_ptr1 = data + (long)(j_start + 1) * D;
const float* j_ptr2 = data + (long)(j_start + 2) * D;
const float* j_ptr3 = data + (long)(j_start + 3) * D;

```

```

__m512 acc00 = _mm512_setzero_ps(), acc01 = _mm512_setzero_ps(), acc02 =
_mm512_setzero_ps(), acc03 = _mm512_setzero_ps();
__m512 acc10 = _mm512_setzero_ps(), acc11 = _mm512_setzero_ps(), acc12 =
_mm512_setzero_ps(), acc13 = _mm512_setzero_ps();
__m512 acc20 = _mm512_setzero_ps(), acc21 = _mm512_setzero_ps(), acc22 =
_mm512_setzero_ps(), acc23 = _mm512_setzero_ps();
__m512 acc30 = _mm512_setzero_ps(), acc31 = _mm512_setzero_ps(), acc32 =
_mm512_setzero_ps(), acc33 = _mm512_setzero_ps();

```

```

for (int k = 0; k < D; k += 16) {

```

```

    __m512 v_i0 = _mm512_load_ps(i_ptr0 + k);
    __m512 v_i1 = _mm512_load_ps(i_ptr1 + k);
    __m512 v_i2 = _mm512_load_ps(i_ptr2 + k);
    __m512 v_i3 = _mm512_load_ps(i_ptr3 + k);

```

```

    __m512 v_j0 = _mm512_load_ps(j_ptr0 + k);
    acc00 = _mm512_fmadd_ps(v_i0, v_j0, acc00);
    acc10 = _mm512_fmadd_ps(v_i1, v_j0, acc10);
    acc20 = _mm512_fmadd_ps(v_i2, v_j0, acc20);
    acc30 = _mm512_fmadd_ps(v_i3, v_j0, acc30);

```

```

    __m512 v_j1 = _mm512_load_ps(j_ptr1 + k);
    acc01 = _mm512_fmadd_ps(v_i0, v_j1, acc01);
    acc11 = _mm512_fmadd_ps(v_i1, v_j1, acc11);
    acc21 = _mm512_fmadd_ps(v_i2, v_j1, acc21);
    acc31 = _mm512_fmadd_ps(v_i3, v_j1, acc31);

```

```

    __m512 v_j2 = _mm512_load_ps(j_ptr2 + k);
    acc02 = _mm512_fmadd_ps(v_i0, v_j2, acc02);
    acc12 = _mm512_fmadd_ps(v_i1, v_j2, acc12);
    acc22 = _mm512_fmadd_ps(v_i2, v_j2, acc22);
    acc32 = _mm512_fmadd_ps(v_i3, v_j2, acc32);

```

```

    __m512 v_j3 = _mm512_load_ps(j_ptr3 + k);
    acc03 = _mm512_fmadd_ps(v_i0, v_j3, acc03);
    acc13 = _mm512_fmadd_ps(v_i1, v_j3, acc13);
    acc23 = _mm512_fmadd_ps(v_i2, v_j3, acc23);
    acc33 = _mm512_fmadd_ps(v_i3, v_j3, acc33);

```

```

}

```

```

float dps[4][4];

```

```

dps[0][0]=_mm512_reduce_add_ps(acc00); dps[0][1]=_mm512_reduce_add_ps(acc01);
dps[0][2]=_mm512_reduce_add_ps(acc02); dps[0][3]=_mm512_reduce_add_ps(acc03);

```

```

    dps[1][0]=_mm512_reduce_add_ps(acc10); dps[1][1]=_mm512_reduce_add_ps(acc11);
    dps[1][2]=_mm512_reduce_add_ps(acc12); dps[1][3]=_mm512_reduce_add_ps(acc13);
    dps[2][0]=_mm512_reduce_add_ps(acc20); dps[2][1]=_mm512_reduce_add_ps(acc21);
    dps[2][2]=_mm512_reduce_add_ps(acc22); dps[2][3]=_mm512_reduce_add_ps(acc23);
    dps[3][0]=_mm512_reduce_add_ps(acc30); dps[3][1]=_mm512_reduce_add_ps(acc31);
    dps[3][2]=_mm512_reduce_add_ps(acc32); dps[3][3]=_mm512_reduce_add_ps(acc33);

    for(int i=0; i<4; ++i) {
        for(int j=0; j<4; ++j) {
            dot_products[(long)(i_start + i) * N + (j_start + j)] = dps[i][j];
            dot_products[(long)(j_start + j) * N + (i_start + i)] = dps[i][j];
        }
    }
}

// 主循环中，对于非对角线块，使用 4x4 微内核：
int i = i_block;
for (; i <= i_max - IR; i += IR) { // IR=4
    int j = j_block;
    for (; j <= j_max - JR; j += JR) { // JR=4
        micro_kernel_4x4(i, j, D, N, data, dot_products.data());
    }
    // 清理 j 循环的剩余部分（如果 j_max-j_block 不是 4 的倍数）
    for (; j < j_max; ++j) {
        // ... 使用老的 dot_product 函数处理 ...
    }
}
// 清理 i 循环的剩余部分（如果 i_max-i_block 不是 4 的倍数）
for (; i < i_max; ++i) {
    // ... 使用老的 dot_product 函数处理 ...
}
}

```

完整代码

```

#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <cstdint>
#include <omp.h>
#include <immintrin.h>

// 点积函数
float dot_product_avx512_aligned(const float *a, const float *b, int D) {
    __m512 sum_vec = _mm512_setzero_ps();
    for (int i = 0; i <= D - 16; i += 16) {
        __m512 a_vec = _mm512_load_ps(a + i);
        __m512 b_vec = _mm512_load_ps(b + i);
        sum_vec = _mm512_fmadd_ps(a_vec, b_vec, sum_vec);
    }
}

```

```

    return _mm512_reduce_add_ps(sum_vec);
}

// 4x4 寄存器分块微内核
// 计算 i_ptr[0..3] 和 j_ptr[0..3] 之间的 16 个点积
// 并将结果存入 dot_products 矩阵及其对称位置
inline void micro_kernel_4x4(int i_start, int j_start, int D, int N, const float*
data, float* dot_products) {
    const float* i_ptr0 = data + (long)(i_start + 0) * D;
    const float* i_ptr1 = data + (long)(i_start + 1) * D;
    const float* i_ptr2 = data + (long)(i_start + 2) * D;
    const float* i_ptr3 = data + (long)(i_start + 3) * D;

    const float* j_ptr0 = data + (long)(j_start + 0) * D;
    const float* j_ptr1 = data + (long)(j_start + 1) * D;
    const float* j_ptr2 = data + (long)(j_start + 2) * D;
    const float* j_ptr3 = data + (long)(j_start + 3) * D;

    __m512 acc00 = _mm512_setzero_ps(), acc01 = _mm512_setzero_ps(), acc02 =
_mm512_setzero_ps(), acc03 = _mm512_setzero_ps();
    __m512 acc10 = _mm512_setzero_ps(), acc11 = _mm512_setzero_ps(), acc12 =
_mm512_setzero_ps(), acc13 = _mm512_setzero_ps();
    __m512 acc20 = _mm512_setzero_ps(), acc21 = _mm512_setzero_ps(), acc22 =
_mm512_setzero_ps(), acc23 = _mm512_setzero_ps();
    __m512 acc30 = _mm512_setzero_ps(), acc31 = _mm512_setzero_ps(), acc32 =
_mm512_setzero_ps(), acc33 = _mm512_setzero_ps();

    for (int k = 0; k < D; k += 16) {
        __m512 v_i0 = _mm512_load_ps(i_ptr0 + k);
        __m512 v_i1 = _mm512_load_ps(i_ptr1 + k);
        __m512 v_i2 = _mm512_load_ps(i_ptr2 + k);
        __m512 v_i3 = _mm512_load_ps(i_ptr3 + k);

        __m512 v_j0 = _mm512_load_ps(j_ptr0 + k);
        acc00 = _mm512_fmadd_ps(v_i0, v_j0, acc00);
        acc10 = _mm512_fmadd_ps(v_i1, v_j0, acc10);
        acc20 = _mm512_fmadd_ps(v_i2, v_j0, acc20);
        acc30 = _mm512_fmadd_ps(v_i3, v_j0, acc30);

        __m512 v_j1 = _mm512_load_ps(j_ptr1 + k);
        acc01 = _mm512_fmadd_ps(v_i0, v_j1, acc01);
        acc11 = _mm512_fmadd_ps(v_i1, v_j1, acc11);
        acc21 = _mm512_fmadd_ps(v_i2, v_j1, acc21);
        acc31 = _mm512_fmadd_ps(v_i3, v_j1, acc31);

        __m512 v_j2 = _mm512_load_ps(j_ptr2 + k);
        acc02 = _mm512_fmadd_ps(v_i0, v_j2, acc02);
        acc12 = _mm512_fmadd_ps(v_i1, v_j2, acc12);
        acc22 = _mm512_fmadd_ps(v_i2, v_j2, acc22);
    }
}

```

```

        acc32 = _mm512_fmadd_ps(v_i3, v_j2, acc32);

        _mm512 v_j3 = _mm512_load_ps(j_ptr3 + k);
        acc03 = _mm512_fmadd_ps(v_i0, v_j3, acc03);
        acc13 = _mm512_fmadd_ps(v_i1, v_j3, acc13);
        acc23 = _mm512_fmadd_ps(v_i2, v_j3, acc23);
        acc33 = _mm512_fmadd_ps(v_i3, v_j3, acc33);
    }

    float dps[4][4];
    dps[0][0]=_mm512_reduce_add_ps(acc00); dps[0][1]=_mm512_reduce_add_ps(acc01);
    dps[0][2]=_mm512_reduce_add_ps(acc02); dps[0][3]=_mm512_reduce_add_ps(acc03);
    dps[1][0]=_mm512_reduce_add_ps(acc10); dps[1][1]=_mm512_reduce_add_ps(acc11);
    dps[1][2]=_mm512_reduce_add_ps(acc12); dps[1][3]=_mm512_reduce_add_ps(acc13);
    dps[2][0]=_mm512_reduce_add_ps(acc20); dps[2][1]=_mm512_reduce_add_ps(acc21);
    dps[2][2]=_mm512_reduce_add_ps(acc22); dps[2][3]=_mm512_reduce_add_ps(acc23);
    dps[3][0]=_mm512_reduce_add_ps(acc30); dps[3][1]=_mm512_reduce_add_ps(acc31);
    dps[3][2]=_mm512_reduce_add_ps(acc32); dps[3][3]=_mm512_reduce_add_ps(acc33);

    for(int i=0; i<4; ++i) {
        for(int j=0; j<4; ++j) {
            dot_products[(long)(i_start + i) * N + (j_start + j)] = dps[i][j];
            dot_products[(long)(j_start + j) * N + (i_start + i)] = dps[i][j];
        }
    }
}

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(NULL);

    uint32_t N, D;
    std::cin.read(reinterpret_cast<char *>(&N), sizeof(N));
    std::cin.read(reinterpret_cast<char *>(&D), sizeof(D));

    float *data = (float *)_mm_malloc((size_t)N * D * sizeof(float), 64);
    std::cin.read(reinterpret_cast<char *>(data), (long)N * D * sizeof(float));

    std::vector<float> inv_norms(N);
    #pragma omp parallel for schedule(static)
    for (uint32_t i = 0; i < N; ++i) {
        float norm_sq = dot_product_avx512_aligned(data + (long)i * D, data +
(long)i * D, D);
        inv_norms[i] = 1.0f / (std::sqrt(norm_sq) + 1e-12f);
    }

    std::vector<float> dot_products(N * N);

```

```

// 块大小，微内核步长
const int B = 64;
const int IR = 4;
const int JR = 4;

#pragma omp parallel for schedule(dynamic)
for (int i_block = 0; i_block < N; i_block += B) {
    for (int j_block = i_block; j_block < N; j_block += B) {
        int i_max = std::min(i_block + B, (int)N);
        int j_max = std::min(j_block + B, (int)N);

        if (i_block == j_block) {
            // 对角线块，情况复杂，继续使用简单循环保证正确性
            for (int i = i_block; i < i_max; ++i) {
                for (int j = i; j < j_max; ++j) {
                    float dp = dot_product_avx512_aligned(data + (long)i * D,
data + (long)j * D, D);

                    dot_products[(long)i * N + j] = dp;
                    dot_products[(long)j * N + i] = dp;
                }
            }
        } else {
            // 非对角线块，使用 4x4 微内核
            int i = i_block;
            for (; i <= i_max - IR; i += IR) {
                int j = j_block;
                for (; j <= j_max - JR; j += JR) {
                    micro_kernel_4x4(i, j, D, N, data, dot_products.data());
                }
                // 清理 j 循环的剩余部分
                for (; j < j_max; ++j) {
                    for(int ii=0; ii<IR; ++ii) {
                        float dp = dot_product_avx512_aligned(data + (long)(i +
ii) * D, data + (long)j * D, D);

                        dot_products[(long)(i + ii) * N + j] = dp;
                        dot_products[(long)j * N + (i + ii)] = dp;
                    }
                }
            }
            // 清理 i 循环的剩余部分
            for (; i < i_max; ++i) {
                for (int j = j_block; j < j_max; ++j) {
                    float dp = dot_product_avx512_aligned(data + (long)i * D,
data + (long)j * D, D);

                    dot_products[(long)i * N + j] = dp;
                    dot_products[(long)j * N + i] = dp;
                }
            }
        }
    }
}

```

```

    }
}

std::vector<float> final_results(N * 4);
#pragma omp parallel
{
    std::vector<float> cosine_sim_thread_buffer(N);
    #pragma omp for schedule(static)
    for (uint32_t i = 0; i < N; ++i) {
        const float inv_norm_i = inv_norms[i];
        for (uint32_t j = 0; j < N; ++j) {
            cosine_sim_thread_buffer[j] = dot_products[(long)i * N + j] *
inv_norm_i * inv_norms[j];
        }

        std::partial_sort(cosine_sim_thread_buffer.begin(),
                        cosine_sim_thread_buffer.begin() + 5,
                        cosine_sim_thread_buffer.end(),
                        std::greater<float>());

        float* result_ptr = final_results.data() + (long)i * 4;
        std::copy(cosine_sim_thread_buffer.data() + 1,
cosine_sim_thread_buffer.data() + 5, result_ptr);
    }
}

std::cout.write(reinterpret_cast<char *>(final_results.data()), (long)N * 4 *
sizeof(float));

_mm_free(data);

return 0;
}

```

traffic-detector

题目复述

1. **输入**：海量的网络流量日志，按时间戳排序。
2. **任务**：识别并统计两种恶意行为。
 - **端口扫描 (portscan)**：某个源IP向某个目的IP+端口，只发送了一个SYN包，之后再无下文。我们要统计每个源IP发起了多少次这样的行为。
 - **DNS隧道 (tunnelling)**：某个源IP发起的DNS查询，其域名前缀（第一个 . 之前的部分）长度大于等于30。我们要统计每个源IP发起的这类查询的**前缀长度之和**。
3. **输出**：

- 先输出所有 `portscan` 结果，再输出所有 `tunnelling` 结果。
- 每个类别内部，按IP地址的字典序升序排列。

17.71pts

原始实现

```
#include <iostream>
#include <sstream>
#include <map>
#include <vector>
#include <string>

struct Packet {
    double timestamp;
    std::string protocol;
    std::string src_ip, dst_ip;
    int src_port = -1, dst_port = -1;
    std::string flags;
    int data_len = 0;
    std::string data;
};

Packet parse_line(const std::string& line) {
    Packet pkt;
    std::istringstream iss(line);
    iss >> pkt.timestamp >> pkt.protocol >> pkt.src_ip >> pkt.dst_ip;
    if (pkt.protocol == "TCP" || pkt.protocol == "DNS") {
        iss >> pkt.src_port >> pkt.dst_port;
        if (pkt.protocol == "TCP") {
            iss >> pkt.flags;
        }
        iss >> pkt.data_len;
        if (iss.peek() == ' ' || iss.peek() == '\\t') iss.get();
        std::getline(iss, pkt.data);
        if (!pkt.data.empty() && pkt.data[0] == ' ') pkt.data.erase(0, 1);
    }
    return pkt;
}

std::string get_dns_prefix(const std::string& domain) {
    size_t dot = domain.find('.');
    if (dot != std::string::npos) return domain.substr(0, dot);
    return "";
}

int main() {
    std::ios_base::sync_with_stdio(false);
```

```

std::cin.tie(NULL);

struct FiveTuple {
    std::string src_ip, dst_ip;
    int src_port, dst_port;
};

auto tuple_str = [](const FiveTuple& t) {
    return t.src_ip + "|" + t.dst_ip + "|" + std::to_string(t.src_port) + "|" +
std::to_string(t.dst_port);
};

std::map<std::string, std::vector<Packet>> syn_flows;

std::map<std::string, int> dnstunnel_count; //自动处理字典序

std::string line;
while (std::getline(std::cin, line)) {
    if (line.empty()) continue;
    Packet pkt = parse_line(line);

    if (pkt.protocol == "TCP") {
        FiveTuple key{pkt.src_ip, pkt.dst_ip, pkt.src_port, pkt.dst_port};
        syn_flows[tuple_str(key)].push_back(pkt);
    }
    else if (pkt.protocol == "DNS" && !pkt.data.empty()) {
        std::string prefix = get_dns_prefix(pkt.data);
        if (prefix.length() >= 30) {
            dnstunnel_count[pkt.src_ip] += prefix.length();
        }
    }
}

std::map<std::string, int> portscan_ip_count;
for (const auto& kv : syn_flows) {
    const std::vector<Packet>& pkts = kv.second;
    if (pkts.size() == 1 && pkts[0].flags == "SYN") {
        portscan_ip_count[pkts[0].src_ip]++;
    }
}

for (const auto& kv : portscan_ip_count) {
    std::cout << kv.first << " portscan " << kv.second << std::endl;
}

for (const auto& kv : dnstunnel_count) {
    std::cout << kv.first << " tunnelling " << kv.second << std::endl;
}

```

```
    return 0;
}
```

75.77pts

数据结构

原始版本中 `std::map<string, std::vector<Packet>> syn_flows;` 会存储**所有**TCP数据包的**完整信息**。如果有一亿条TCP日志，内存中就会有一亿个 `Packet` 对象，这是不可接受的。而题目对“端口扫描”的定义：“源IP只发送一个SYN包便没有后续流量”。实际上，对于一条TCP流，我们根本不需要记录它的所有数据包，只需要一个char知道它的**状态**即可。

```
// 0 = 未见, 1 = 仅见一次且为 SYN, 2 = 多包或非 SYN (不可计为 portscan)
std::unordered_map<std::string, char> flow_map;
```

状态转移

```
auto it = flow_map.find(key);
if (it == flow_map.end()) {
    // 首次见到
    if (flags == "SYN") {
        // 仅保存状态，源IP在最后汇总时通过解析key得到
        flow_map.emplace(std::move(key), 1);
    } else {
        flow_map.emplace(std::move(key), 2);
    }
} else {
    // 已存在，若之前是 1 则变为 2（多包）
    if (it->second == 1) {
        it->second = 2;
    }
}
```

同时，我们将 `std::map` -> `std::unordered_map`

- `std::map`：内部实现是**红黑树**（一种自平衡二叉搜索树）。它的优点是键（key）**总是有序的**。缺点是插入和查找的时间复杂度都是 $O(\log N)$ ，其中N是map中的元素数量。当N很大时，每次操作都会变慢。
- `std::unordered_map`：内部实现是**哈希表**。它的优点是，在没有哈希冲突的理想情况下，插入和查找的平均时间复杂度是 $O(1)$ ，即与元素数量无关，速度**非常快**。缺点是元素是无序的。

```
flow_map.reserve(1 << 22);
```

而哈希表为了保持 $O(1)$ 的高效性能，需要维持一个合适的“装载因子”（元素数量 / 桶的数量）。当元素不断增多，超过某个阈值时，哈希表就需要进行**扩容**，创建一个更大的新表，并把所有旧元素重新计算哈希

值再放进去。这个过程称为“重哈希”，它是一次非常耗时的操作。`reserve(n)` 的作用是告诉 `unordered_map`：“我准备要放大约 `n` 个元素，你提前把空间准备好”。这样，在插入元素的过程中，就可以**避免或大大减少**耗时的重哈希操作。

当然，这样做带来的后果是最后需要统一进行一次重排

```
// 汇总 portscan
std::unordered_map<std::string, long long> portscan_count;
portscan_count.reserve(1 << 20);
for (const auto &[key, state] : flow_map) {
    if (state == 1) {
        // 从 key 中解析出 src_ip
        size_t pos = key.find('|');
        portscan_count[key.substr(0, pos)] += 1;
    }
}

// 输出
std::vector<std::pair<std::string, long long>> portvec;
portvec.reserve(portscan_count.size());
for (const auto &kv : portscan_count) portvec.emplace_back(kv.first, kv.second);
std::sort(portvec.begin(), portvec.end());
for (const auto &p : portvec) {
    std::cout << p.first << " portscan " << p.second << '\n';
}

std::vector<std::pair<std::string, long long>> dnsvec;
dnsvec.reserve(dnstunnel_count.size());
for (const auto &kv : dnstunnel_count) dnsvec.emplace_back(kv.first, kv.second);
std::sort(dnsvec.begin(), dnsvec.end());
for (const auto &p : dnsvec) {
    std::cout << p.first << " tunnelling " << p.second << '\n';
}
```

I/O

使用 C++17 的 `std::string_view` 替换 `istreamstream`。

```
// 跳过前导空格
inline void skip_spaces(std::string_view s, size_t &i) {
    while (i < s.size() && s[i] == ' ') ++i;
}

// 读取下一个 token, 返回 string_view
inline std::string_view next_token_sv(std::string_view s, size_t &i) {
    skip_spaces(s, i);
    if (i >= s.size()) return {};
    size_t j = i;
```

```

    while (j < s.size() && s[j] != ' ') ++j;
    std::string_view tok = s.substr(i, j - i);
    i = j;
    return tok;
}
//读取...
// 使用 string_view 进行解析
std::string_view line_sv(line);
size_t i = 0;

(void) next_token_sv(line_sv, i); // timestamp (skip)

std::string_view proto = next_token_sv(line_sv, i);
if (proto.empty()) continue;

std::string_view src_ip = next_token_sv(line_sv, i);
if (src_ip.empty()) continue;
std::string_view dst_ip = next_token_sv(line_sv, i);
if (dst_ip.empty()) continue;
std::string_view src_port = next_token_sv(line_sv, i);
if (src_port.empty()) continue;
std::string_view dst_port = next_token_sv(line_sv, i);
if (dst_port.empty()) continue;

```

85.32pts

用整数/结构体代替字符串键。

IPV4

一个IPv4地址，如 `a.b.c.d`，本质上就是4个字节的数据。用一个32位的无符号整数 (`uint32_t`) 来表示。

```

// "192.168.1.10" -> 0xC0A8010A (3232235786)
inline uint32_t parse_ipv4(std::string_view sv);

// 0xC0A8010A -> "192.168.1.10"
std::string format_ipv4(uint32_t ip);

```

TCP五元组

对于TCP流的五元组，用结构体 `TCPKey` 来表示。

```

struct TCPKey {
    uint32_t src_ip;    // 4 bytes
    uint32_t dst_ip;    // 4 bytes
    uint16_t src_port;  // 2 bytes

```

```
uint16_t dst_port; // 2 bytes
}; // Total: 12 bytes
```

哈希：

```
// 为 TCPKey 提供自定义哈希函数
namespace std {
template <>
struct hash<TCPKey> {
    size_t operator()(const TCPKey& k) const {
        // 使用一个简单的组合哈希函数
        size_t h1 = hash<uint32_t>{}(k.src_ip);
        size_t h2 = hash<uint32_t>{}(k.dst_ip);
        size_t h3 = hash<uint16_t>{}(k.src_port);
        size_t h4 = hash<uint16_t>{}(k.dst_port);
        // 将多个哈希值组合起来
        size_t seed = h1;
        seed ^= h2 + 0x9e3779b9 + (seed << 6) + (seed >> 2);
        seed ^= h3 + 0x9e3779b9 + (seed << 6) + (seed >> 2);
        seed ^= h4 + 0x9e3779b9 + (seed << 6) + (seed >> 2);
        return seed;
    }
};
}
```

118.25pts

MMAP

用 `mmap` (Memory-mapped I/O) 替换 `std::getline` 读取。

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>

// ... in main()
int fd = STDIN_FILENO;
struct stat sb;
fstat(fd, &sb);
size_t file_size = sb.st_size;
const char* buffer = (const char*)mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, fd,
0);
```

排序

原始代码在输出前，将 `uint32_t` 类型的 IP 地址转换为 `std::string`，然后再对字符串向量进行排序。问题在于，`std::string` 对象在堆上分配内存，访问时可能导致缓存未命中。

在这里，我们直接对包含 `uint32_t` IP 和计数的 `std::vector<std::pair<uint32_t, long long>>` 进行排序。

在需要按字典序输出 IP 时，自定义一个比较函数，该函数在比较时才将整数 IP 转换为字符串。

由于是在 `char buf_a[16], buf_b[16];` 栈上分配内存，速度更快。

```
std::sort(sorted_counts.begin(), sorted_counts.end(), [](const auto& a, const auto& b) {
    char buf_a[16], buf_b[16];
    sprintf(buf_a, "%u.%u.%u.%u", (a.first >> 24) & 0xFF, (a.first >> 16) & 0xFF,
            (a.first >> 8) & 0xFF, a.first & 0xFF);
    sprintf(buf_b, "%u.%u.%u.%u", (b.first >> 24) & 0xFF, (b.first >> 16) & 0xFF,
            (b.first >> 8) & 0xFF, b.first & 0xFF);
    return strcmp(buf_a, buf_b) < 0;
});
```

fwrite

```
std::string out_buffer;
out_buffer.reserve(sorted_counts.size() * 50);
char line_buf[128];
for (const auto& [ip_int, count] : sorted_counts) {
    int len = sprintf(line_buf, "%u.%u.%u.%u %s %lld\n",
                      (ip_int >> 24) & 0xFF, (ip_int >> 16) & 0xFF, (ip_int >> 8) & 0xFF, ip_int &
0xFF,
                      type.c_str(), count);
    out_buffer.append(line_buf, len);
}
fwrite(out_buffer.data(), 1, out_buffer.size(), stdout);
```

OpenMP并行化!

我们不能所有线程共享一个全局的 `portscan_count` 和 `dnstunnel_count` 哈希表。然后：

```
// 伪代码 - 低效的方式
std::unordered_map<uint32_t, long long> global_portscan_count;

#pragma omp parallel for
for (long i = 0; i < total_lines; ++i) {
    // ... 解析日志得到 src_ip ...

    #pragma omp critical
    {
        global_portscan_count[src_ip]++;
    }
}
```

这样会导致高锁竞争。

数据分线程

我们应该先整理分发数据，确保**所有与同一个 `src_ip` 相关的数据最终都交给同一个线程处理**。

使用 `mmap` 将整个文件映射到内存后，我们将这个巨大的内存块在逻辑上切分成 N 份（ $N = \text{线程数}$ ），每个线程分配一份。

每个线程 `tid` 都会创建一个**线程本地的二维 `vector`**：

```
std::vector<std::vector<std::string_view>> local_buckets(num_threads);
```

桶 `j` 的作用是：临时存放所有未来应该由线程 `j` 来处理的日志行。

对于每一行日志解析出 `src_ip_int`

然后，它根据这个 IP 计算出一个目的线程：

```
int destination_thread_id = src_ip_int % num_threads;
```

最后，它将这行日志（以 `string_view` 的形式）放入自己的本地桶中：

```
local_buckets[destination_thread_id].push_back(line_sv);
```

这个过程保证了：**无论一个 `src_ip` 出现在文件的哪个角落，只要它的 `src_ip_int % num_threads` 的结果是 `k`，那么处理它的任务最终一定会被送到线程 `k` 的手中。**

最后，合并所有线程的结果：

```
// Merge local buckets into global task lists
#pragma omp critical
{
    for(int i = 0; i < num_threads; ++i) {
        if(!local_buckets[i].empty()) {
            thread_lines[i].insert(thread_lines[i].end(), local_buckets[i].begin(),
local_buckets[i].end());
        }
    }
}

// Wait for all threads to finish distribution
#pragma omp barrier
```

随后，每个线程只用处理 `thread_lines[tid]` 这个任务列表，将其放入 `thread_data[tid]` 中，这部分可以并行。

```
worker_func(thread_lines[tid], thread_data[tid]);
```


122.95pts

并行

引入三维 `vector`： `all_local_buckets[tid][dest_tid]`。

这样，合并时候也可以并行化了：

```
#pragma omp parallel for num_threads(num_threads)
for (int i = 0; i < num_threads; ++i) {
    // ...
    thread_lines[i].reserve(total_size);
    for (int j = 0; j < num_threads; ++j) {
        thread_lines[i].insert(thread_lines[i].end(), all_local_buckets[j]
[i].begin(), all_local_buckets[j][i].end());
    }
}
```

解析日志

用新实现的 `fast_parse_line` 代替一系列 `next_token_sv` 调用来分割字符串

125.71pts

在分发数据的阶段，我们不需要解析整行日志。我们只需要知道这行日志应该被哪个线程处理。所以用

`get_key_for_hash` 去获取最少需求信息

```
优化上一版本中的 all_local_buckets[tid][dest_tid].push_back(line_sv)
thread_lines[i].insert(thread_lines[i].end(), all_local_buckets[j][i].begin(),
all_local_buckets[j][i].end());
```

通过算出hash后只统计counts[i][j]和write_offsets[i][j](前缀和)，在需要插入时候直接索引赋值。

185.85pts

用手写的 `FastMap` 替换了标准库的 `std::unordered_map`

`std::unordered_map` 通常采用“开链法”来解决哈希冲突。它的内部结构大致是一个 `std::vector`（桶数组），每个桶里存放一个指针，指向一个在堆上分配的节点链表（或红黑树）。

其问题在于每次插入新元素，都可能需要在堆上进行一次小内存分配。节点在内存中的位置是随机的、不连续的，当遍历一个冲突链表时，CPU需要进行多次指针跳转，极易导致缓存未命中。

`FastMap` 采用的是线性探测。

整个哈希表就是一个巨大的、连续的 `std::vector<Entry>`。

当发生哈希冲突时，它不会创建链表，而是简单地检查 `index + 1`，直到找到一个空槽。

这使得当访问一个元素时，CPU的缓存预取机制会自动将它后面的多个元素也加载到高速缓存中。这样，当发生冲突并进行线性探测时，需要检查的下一个、再下一个元素极有可能已经在缓存里了，访问速度飞快。

194.66pts

在前一个版本中，存在一个逻辑上的冗余：

1. **PASS 1 (计数)**: 调用 `get_key_for_hash` 或 `ultimate_parser` 来解析出Key，进行计数。
2. **PASS 2 (分散)**: 再次调用 `get_key_for_hash` 或 `ultimate_parser` 来解析出Key，确定目标线程，然后移动 `string_view`。
3. **PASS 3 (处理)**: 在 `worker_func` 中，第三次调用 `ultimate_parser` 或 `fast_parse_line`，对 `string_view` 进行**完整的最终解析**以执行业务逻辑。
这使得同一行日志字符串被CPU访问了三次。

引入 `ParsedData` 结构体：

```
struct ParsedData {
    bool is_tcp;
    uint32_t src_ip;
    union {
        struct { // TCP specific info
            uint32_t dst_ip;
            uint16_t src_port;
            uint16_t dst_port;
            bool is_syn;
        } tcp;
        struct { // DNS specific info
            uint16_t prefix_len;
        } dns;
    };
};
```

现在只需要每行日志只需要进行一次解析即可。

md5-bf

本题的核心任务是在一个由伪随机数生成器（PRNG）产生的巨大序列中，通过暴力搜索找到一个特定MD5哈希值的原像。得分 = $15/t - 1$ 。

43.33pts

并行化

为了确保每个线程负载均衡，我们采用了交错分配的策略。假设有 `T` 个线程，那么：

- 线程 0 负责计算第 1, 1+T, 1+2T, ... 个输入。
- 线程 1 负责计算第 2, 2+T, 2+2T, ... 个输入。
- ...

- 线程 `i` 负责计算第 `i+1`, `i+1+T`, `i+1+2T`, ... 个输入。

并行搜索需要一个机制来同步结果并及时终止。一旦某个线程找到了答案，其他线程应尽快停止工作，以避免浪费计算资源。

这里我们使用 `std::atomic<uint64_t> min_found_n`。

同时，服务器上没有 OpenSSL，于是在代码中包含了一个完整的 MD5 实现。

154.29pts

优化 generator 跳转

在每个线程的主循环中，为了跳到下一个任务批次，我们需要通过循环调用 `generator.generate()` 数万甚至数十万次。

```
for (int i = 0; i < skip_amount; ++i) {  
    generator.generate(dummy_out);  
}
```

这个循环是纯粹串行的，它的执行时间与线程数和批处理大小成正比。

为了解决这个问题，我们需要一种方法，能够直接计算出 PRNG 在 `N` 步之后的状态。

`xorshift64` 算法中的所有操作（异或和移位）在数学上都是 GF(2) 上的线性变换。这意味着整个 `xorshift64` 的状态更新过程可以被一个 64x64 的转移矩阵 `M` 所描述：

$$\text{State_new} = M * \text{State_old}$$

因此，要将状态向前推进 `N` 步，我们只需要计算：

$$\text{State_N} = M^N * \text{State_old}$$

这个矩阵的幂 `M^N` 可以通过快速幂算法在 `O(log N)` 的时间内高效计算得出。这使得我们能够将原来 `O(N)` 的串行跳转操作，优化成几乎瞬时的对数级操作。

SIMD AVX-512 进行 MD5 计算

之前代码有几个问题：

1. 每一次只能计算一个 48 字节输入的 MD5 哈希。
2. 之前的 MD5 实现是通用的，但题目给定的输入长度（48 字节）是固定的，并且小于一个 MD5 块（64 字节）。
解决方法是：
3. 引入 AVX-512，可以一条指令同时对多个数据（例如，16 个 32 位整数）执行相同的操作。
4. 编写了一个全新的函数 `md5_16x_48_byte`，它专门用于并行计算 16 个 48 字节输入的 MD5 哈希。
于是，线程的任务分配粒度从“一个输入”变为了“一组（16 个）输入”。

- 线程 `i` 的起始点快进 `thread_id * SIMD_WIDTH` 次。
- 主循环中，每个线程一次性生成 16 个输入存入 `input_buffer[SIMD_WIDTH][6]`。
- 调用 `md5_16x_48_byte` 批量处理。
- 检查 16 个输出结果。
- 跳过 `(num_threads - 1) * SIMD_WIDTH` 个输入，到达下一个任务批次。

md5-new

题面从破解一个MD5到破解5个MD5。

21.18pts

在原先的版本上加一个 `for (int i = 0; i < 5; ++i)` 串行处理即可。

118.22pts

使用查找表 (LUT) 加速 PRNG 状态跳转

我们发现，在每个线程的主循环中，`XorshiftJump::transform` 函数：

```
uint64_t transform(const matrix& mat, uint64_t state) {
    uint64_t new_state = 0;
    // 这个循环需要执行 64 次
    for (int i=0; i<64; ++i) {
        if ((state>>i)&1) {
            new_state ^= mat[i];
        }
    }
    return new_state;
}
```

这个函数虽然在算法复杂度上是常数 $O(1)$ （因为位数是固定的64），但它内部的循环和条件分支对于 CPU 的流水线来说并不友好。

利用空间换时间的思想，预先计算出所有可能的部分结果，并将它们存储在一个查找表 中。之后，我们可以通过几次简单的查表和异或操作来替代原来复杂的循环计算。

由于 `transform` 是一个线性变换，我们可以利用其叠加性：

`Transform(A XOR B) == Transform(A) XOR Transform(B)`

我们将一个 64 位的状态 `state` 拆分成 8 个 8 位的字节 (byte)：

`state = byte_7 | byte_6 | ... | byte_0`

那么，根据线性性质：

`Transform(state) = Transform(byte_7) XOR Transform(byte_6) XOR ... XOR Transform(byte_0)`

于是：

- 我们创建了一个全局的查找表 `g_jump_luts`，它是一个 `8 x 256` 的二维数组。
- `g_jump_luts[i][j]` 存储的含义是：一个值为 `j` 的字节，如果它位于 64 位整数的第 `i` 个字节位置上，经过 `stride_jump_mat` 变换后的结果是多少。
- `precompute_jump_luts` 函数负责填充这个表。它在 `#pragma omp single` 块中被调用一次，因为这个查找表对于所有线程和所有循环迭代都是固定不变的（因为它只依赖于 `stride_jump_mat`）。
- 之后，我们用新的 `transform_lut` 函数，它利用预计算好的查找表来完成状态变换：

```
inline uint64_t transform_lut(uint64_t state) {  
    return g_jump_luts[0][(state >> 0) & 0xFF] ^ // 查第0个字节  
           g_jump_luts[1][(state >> 8) & 0xFF] ^ // 查第1个字节  
           ...  
           g_jump_luts[7][(state >> 56) & 0xFF]; // 查第7个字节  
}
```

这个新函数只包含 8 次查表和 7 次异或操作，没有循环和分支，CPU 执行效率高。

136.84pts

之前的 `md5_16x_48_byte` 中需要将输入的 `inputs[SIMD_WIDTH][6]` 【“结构数组”(Array of Structures, AoS)】转置成“数组结构”(Structure of Arrays, SoA) 布局，以便 SIMD 指令能够高效处理。

```
// 之前的低效转置操作，使用 _mm512_set_epi32  
for (int j = 0; j < 12; ++j) {  
    x[j] = _mm512_set_epi32(  
        input_ptr[15*12+j], input_ptr[14*12+j], ..., input_ptr[0*12+j]  
    );  
}
```

我们编写了一个新函数 `generate_16x_soa`，它负责这个转换。

- 它内部循环 16 次，每次调用 `generator.generate()` 得到一个 AoS 格式的输入。
- 然后，它立即将这个输入的数据“拆开”，并直接写入 `soa_buffer` 中正确的位置，从而在生成数据的同时就完成了转置。
- 然后用 `x[j] = _mm512_load_si512((const __m512i*)soa_inputs[j]);` 加载数据，过程相比原先 `_mm512_set_epi32` 快很多。

148.42pts

之前的在 SIMD 寄存器中完成计算后，将全部 16 个哈希结果写回主内存，再通过循环逐一进行比较。可以将比较逻辑下沉，在数据仍在 AVX-512 寄存器中时，利用 `_mm512_cmpeq_epi32_mask` 指令并行完成所有 16 个结果与目标哈希的比较。

165.59pts

之前的实现是在每次调用 `md5_16x_48_byte_compare` 函数时，其内部都会通过 `_mm512_set1_epi32` 指令重复地广播 MD5 算法所需的各种常量（如初始状态 IV 和轮常数 K）。同时，数据的生成逻辑被封装在 `generate_16x_soa` 函数中，每次调用都存在固有的函数调用开销。

我们创建了一个 `MD5_Constants` 结构体，在处理每个测试点的主循环开始前，仅执行一次。

```
int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(NULL);

    MD5_Constants constants;
    constants.A_init = _mm512_set1_epi32(0x67452301);
    constants.B_init = _mm512_set1_epi32(0xefcdab89);
    constants.C_init = _mm512_set1_epi32(0x98badcfe);
    constants.D_init = _mm512_set1_epi32(0x10325476);
    constants.X12 = _mm512_set1_epi32(0x80);
    constants.X13 = _mm512_setzero_si512();
    constants.X14 = _mm512_set1_epi32(384);
    constants.X15 = _mm512_setzero_si512();

    static const uint32_t K_scalar[64] = {
        0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501,
        0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be, 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821,
        0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa, 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,
        0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed, 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a,
        0xffffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c, 0xa4bbee44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,
        0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05, 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
        0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039, 0x655b59c3, 0x8f0ccc92, 0xffefff47d, 0x85845dd1,
        0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1, 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391
    };
    for(int j=0; j<64; ++j) { constants.K[j] = _mm512_set1_epi32(K_scalar[j]); }

    for (int i = 0; i < 5; ++i) {
```

image-11.png

删除了 `generate_16x_soa` 函数和 `RndGen` 类，将其数据生成逻辑展开到每个线程的 `while` 循环内部。

230pts

MD5 算法的每一轮计算都存在数据依赖。例如，计算 `d` 的新值依赖于 `a` 的旧值，计算 `c` 的新值依赖于 `d` 的新值。

这种串行的依赖链会限制 CPU 的指令级并行 (ILP) 能力。现代 CPU 拥有多个执行单元，可以同时执行多条没有相互依赖的指令。但由于 MD5 算法的依赖性，CPU 的超标量 (Superscalar) 特性无法被充分利用，可能会导致流水线停顿 (Pipeline Stall)，等待前一条指令的结果。

软件流水线/双路计算

我们不再一次处理一批（16个）哈希，而是同时处理两批独立的哈希。这两批计算（我们称之为 `path0` 和 `path1`）之间没有任何数据依赖关系。

这使得 CPU 在执行 `path0` 的一条指令时，如果需要等待结果，它可以立刻切换去执行 `path1` 的一条完全不相关的指令，从而保持其执行单元始终繁忙，掩盖了指令的延迟。

好的，这是为你续写的 Writeup 内容，它在前一版的基础上进一步扩展，并加入了四路计算和最终的向量化数据生成优化。

248pts

我们将软件流水线的宽度从两路扩展到了**四路**。现在，每个线程在一次循环迭代中同时处理四批（ $4 * 16 = 64$ 个）独立的哈希。我们为这四路计算分别设置了独立的状态寄存器，并将其指令流完全交错。

同时，之前逐个生成随机数并写入 SoA 缓冲区的内联代码，其串行特性已无法匹配四路计算的需求。

为此，我们现在逐个生成随机数时分两步：

1. 首先，在一个临时缓冲区中，通过循环串行地生成当前迭代所需的所有 PRNG 随机数。
2. 随后，我们利用 AVX-512 指令，以 SIMD 方式从这个临时缓冲区加载数据，并并行完成从 64 位整数到两个 32 位整数的拆分、以及到四套 SoA 缓冲区的转置和存储。

其实第一步应该也能优化，但是我没有实现....