南開大學
Nankai University

计算机学院

并行程序设计实验报告

期末研究报告——口令猜测并行化化选题

姓名：葛明宇

学号：2312388

专业：计算机科学与技术

2025 年 7 月 6 日

目录

1 引言

在现代密码学领域，密码猜测攻击作为一种基础且有效的攻击方式，其效率提升始终是研究热点。随着高性能计算技术的发展，并行计算在密码破解场景中的应用愈发重要。

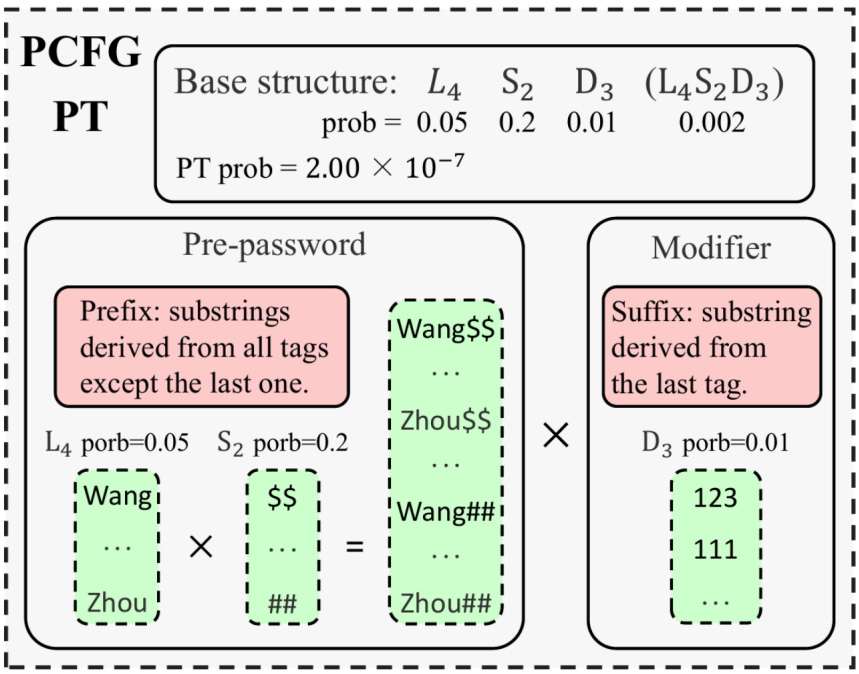

图 1.1: PCFG 并行化算法

本学期的并行程序设计实验中，基于论文 [3] 中的并行算法，如图1.1所示。利用并行体系结构的各种思想对于口令猜测任务进行并行化处理，其中主要是对于 MD5 哈希函数以及口令生成 Generate() 函数进行并行化处理。对于 md5 哈希函数的并行处理已经尝试过 SIMD 向量化，在期末实验中又尝试了使用 AVX2 实现的并行度为 2、4、8 的 SIMD 向量化以及 GPU 优化；对于 Generate() 函数的并行化处理已经尝试过多线程、多进程、GPU 等并行化方法。不过由于多线程实验中并未实现 pthread 实现的多线程优化，导致线程创建销毁的开销较大，在期末实验中也实现了线程池的改进。

2 md5 哈希优化

2.1 SIMD 向量化

2.1.1 并行设计

MD5 哈希算法本身难以实现并行化，但是可以采用多条口令并行执行 MD5 哈希的操作。

2.1.2 NEON 指令集实现

基于对 ARM 架构官方文档的系统梳理 [1]，实现了 MD5 核心逻辑函数（F、G、H、I）、算术移位及其组合操作（FF、GG、HH、II）的 SIMD 指令级优化重构，转为 NEON 指令集的函数，实现了基于 NEON 指令的向量化处理技术。

2.1.3 SSE 实现

基于对 x86 架构官方文档的系统梳理 [2]，实现了 MD5 核心逻辑函数（F、G、H、I）、算术移位及其组合操作（FF、GG、HH、II）的 SIMD 指令级优化重构，转为到 SSE 指令集的函数，实现了基于 SSE 指令的向量化处理技术。

2.1.4 AVX2 实现

实现了 MD5 核心逻辑函数（F、G、H、I）、算术移位及其组合操作（FF、GG、HH、II）的 SIMD 指令级优化重构，转为到 AVX2 指令集的函数，实现了基于 AVX2 指令的向量化处理技术。实现了 2、4、8 三种并行度的 AVX2 指令集下的 MD5 哈希操作，下面以并行度为 8 的 AVX2 指令集下的 MD5 哈希操作为例，因为 NEON 指令集下难以实现并行度 8 下的 SIMD 优化，所以先前没有实现。

以 F、FF 和 ROTATELEFT 为例，其 AVX2 指令实现如下：

```
//md5.h
#define F_AVX2_8(x, y, z) _mm256_or_si256(_mm256_and_si256(x,
    y),_mm256_andnot_si256(x, z))

inline __m256i ROTATELEFT_AVX2_8(__m256i x, int n) {
  return _mm256_or_si256(_mm256_slli_epi32(x, n), _mm256_srli_epi32(x, 32 - n));
}

#define FF_AVX2_8(a, b, c, d, x, s, ac) { \
  a = _mm256_add_epi32(a, _mm256_add_epi32(F_AVX2_8(b, c, d), _mm256_add_epi32(x,
      _mm256_set1_epi32(ac)))); \
  a = ROTATELEFT_AVX2_8(a, s); \
  a = _mm256_add_epi32(a, b); \
}
• • •
```

MD5Hash_AVX2_8way()

输入预处理与内存准备段。StringProcess 函数应是自定义的用于实现 MD5 消息填充规则（补位、添加长度等）的函数，会返回填充后消息的字节数组指针，同时通过输出参数 lengths[j] 带回填充后消息的总长度（字节数）。

```
    //md5.cpp
    Byte* paddedMessages[8];
    int lengths[8];
    for (int j = 0; j < 8; ++j)
  {
    paddedMessages[j] = StringProcess(inputs[j], &lengths[j]);
  }
```

初始化 AVX2 状态寄存器段。初始化 MD5 算法所需的 4 个状态变量（对应 MD5 标准中的 A、B、C、D 初始值），并且利用 AVX2 指令 _mm256_set1_epi32 将相同的初始值广播到 256 位（8 个 32 位整数）的寄存器中，为 8 路并行计算提供初始状态。

```
    // 初始化8路状态寄存器
    __m256i a = _mm256_set1_epi32(0x67452301);
```

```
    ...
```

消息块处理循环（外层块循环）段。按 MD5 算法中 512 位（64 字节）为一个消息块的规则，遍历每个输入消息的所有块，对每个块进行处理。因为 8 路消息是并行处理且假设它们的块数量相同（lengths[0]/64 作为统一块数，实际应保证 8 路消息填充后块数一致，否则需额外处理），所以用 lengths[0] 来控制循环次数。

```
    // 处理所有消息块
    for (int blk = 0; blk < lengths[0]/64; ++blk) {
        __m256i x[16];

        // 加载8个消息块的数据到AVX寄存器
        for (int k = 0; k < 16; ++k) {
            alignas(32) uint32_t words[8];
            for (int j = 0; j < 8; ++j) {
                memcpy(&words[j], paddedMessages[j] + blk*64 + k*4, 4);
                words[j] = htole32(words[j]);  // 转换为小端序
            }
            x[k] = _mm256_load_si256((__m256i*)words);
        }

        __m256i aa = a, bb = b, cc = c, dd = d;

        /* Round 1 */
        FF_AVX2_8(a, b, c, d, x[0], s11, 0xd76aa478);
        ...
        /* Round 2 */
        GG_AVX2_8(a, b, c, d, x[1], s21, 0xf61e2562);
        ...
        /* Round 3 */
        HH_AVX2_8(a, b, c, d, x[5], s31, 0xfffa3942);
        ...
        /* Round 4 */
        II_AVX2_8(a, b, c, d, x[0], s41, 0xf4292244);
        ...
        a = _mm256_add_epi32(a, aa);
        ...
    }
```

结果存储与字节序转换段。先将 AVX2 寄存器中存储的 8 路并行计算得到的最终状态，通过 _mm256_store_si256 指令存储到内存数组 result 中；然后对每个结果进行字节序转换（因为 MD5 结果通常以大端序展示，而计算过程中用的是小端序），并填充到输出参数 states 中，供调用者使用。

```
    // 存储结果并转换字节序
    alignas(32) uint32_t result[4][8];
    _mm256_store_si256((__m256i*)result[0], a);
    _mm256_store_si256((__m256i*)result[1], b);
    _mm256_store_si256((__m256i*)result[2], c);
    _mm256_store_si256((__m256i*)result[3], d);
```

```
7
8       for (int j = 0; j < 8; ++j) {
9           states[j][0] = __builtin_bswap32(result[0][j]);
10          states[j][1] = __builtin_bswap32(result[1][j]);
11          states[j][2] = __builtin_bswap32(result[2][j]);
12          states[j][3] = __builtin_bswap32(result[3][j]);
13      }
```

内存释放段。释放之前调用 StringProcess 函数为 8 路消息分配的填充后消息内存，避免内存泄漏。

```
1       // 释放内存
2       for (int j = 0; j < 8; ++j) {
3           delete[] paddedMessages[j];
4       }
```

2.1.5 编译选项对加速比的影响

性能测试

编译选项	串行 Hash 时间 (s)	并行 Hash 时间 (s)	加速比
不编译优化	9.53	12.07	0.79
O1 优化	3.11	1.70	1.83
O2 优化	2.94	1.65	1.78

表 1: SIMD 不同编译选项的加速比

不进行编译优化没能实现加速，-O1 优化时加速比最大，-O2 优化比-O1 优化加速比稍低。

结果分析

不进行编译优化时，加速比为 0.79：(1)NEON 的向量加载被拆分为多次单值加载，而串行代码直接使用单值操作，避免了额外开销。(2)NEON 中的向量会被存储在内存之中，频繁读取内存会导致额外开销。(3) 未优化流水线填充，无法充分发挥硬件并行性。

结果是 NEON 的并行计算仍在发生，但频繁的内存操作和低效调度导致整体性能低下。

-O1 优化时，加速比为 1.83：(1) 变量优先存入寄存器，减少内存访问。(2) 合并内存操作，例如 uint32x4_t state[0] = vdupq_n_u32(0x67452301) 同时加载四个 0x67452301。(3) 调整指令顺序以填充流水线，NEON 的并行性得以高效体现。

结果是满足了预期加速比，虽然由于标量转为向量的额外开销，使得无法实现 4 倍的加速比。但是提升 1 倍的性能也是很可观的。

-O2 优化时，加速比为 1.78：(1) 减少循环控制语句的开销。(2) 调整指令顺序以充分利用 CPU 流水线。

结果是比-O1 优化加速比稍低，neon 实现 simd 并行的主要思想都在-O1 优化时已经得到充分体现，在-O2 只是进一步提升性能，但这种提升与 neon 的并行性相关性不大，对串行和并行都会提升。可以认为这种时间带来的提升主要是来自于小循环的展开。

2.1.6 delete 对加速比的影响

初始编写代码时，没有对 paddedMessages 数组开辟的空间进行释放。在测试程序性能时，Hash time 相对于串行版本的 Hash time 有提升，但是没有达到预期的加速比。

性能测试

编译选项	串行 Hash 时间 (s)	并行 Hash 时间 (s)	加速比
不编译优化	9.53	12.56	0.76
O1 优化	3.11	2.14	1.45
O2 优化	2.94	2.00	1.47

表 2: 缺失 delete 加速比结果

这里的缺失 delete 是相对于 NEON 优化后的版本。

结果分析

从 4 条口令上的处理来看其实并没有什么影响。但是我们的实验是在 10000000 条口令的量级上进行的，不进行 delete 的处理会导致堆碎片化和分配延迟增加以及缓存性能劣化，内存占用过高影响缓存局部性，使得 cache 命中率下降。最终导致整体性能下降。

2.1.7 单次运算并行度对加速比影响

由上述对各编译模式下加速比的分析，我们可以使用-O1 和-O2 优化的 NEON 并行版本对单次运算并行度进行测试。同时，期末实验中基于 AVX2 实现了 2、4、8 三种并行度情况下的 SIMD 实现，并重新测试了加速比。在-O1 及以上级别的优化下，当单次运算并行度越高，加速比越高。因为吞吐量

编译选项	并行度为 2 的加速比	并行度为 4 的加速比	并行度为 8 的加速比
O1 优化	1.20	1.85	2.43
O2 优化	1.13	1.76	2.32

表 3: 单次运算并行度的加速比

带来的巨大性能提升会削弱标量变向量的影响。

2.2 GPU 优化

在 GPU 实验中，对于 Generate() 函数的优化主要在于降低拼接字符串的开销，但是当开启编译优化后，CPU 对于字符串的拼接性能很高，而对于 GPU 的提升不大。因此尝试寻找更适合 GPU 优化的部分。GPU 适合处理大规模的运算，而 md5 哈希函数中存在大量位运算、逻辑运算以及算术运算，自然地实现了 CUDA 版本的 md5 哈希函数。

2.2.1 CUDA 优化设计

采用批处理模型。每个 CUDA 线程独立处理一个密码的完整 MD5 哈希计算，同时 内存布局为扁平化存储：所有密码存储在连续内存块中和辅助数组：使用偏移量数组 (d_offsets) 和长度数组 (d_lengths) 定位密码。一次内核调用处理所有密码，避免多次内核启动开销。

2.2.2 CUDA 优化实现

md5_kernel_batch()

核函数由 GPU 上的多个线程并行执行，每个线程负责计算一个密码的 MD5 哈希值。

线程索引与数据准备。计算线程索引，确保不越界，并获取当前线程要处理的密码数据。

```

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= num_passwords) return;
    const char* password = d_password_data + d_offsets[idx];
    int length = d_lengths[idx];
```

消息填充。按照 MD5 算法要求对输入字符串进行填充。

```
    Byte paddedMessage[256]; // 足够大的缓冲区
    int paddedLength;
    StringProcess_gpu(password, length, paddedMessage, &paddedLength);
    int n_blocks = paddedLength / 64;
```

初始化 MD5 状态。初始化 MD5 算法的四个初始哈希值（A、B、C、D）。

```
    bit32 state[4];
    state[0] = 0x67452301;
    • • •
```

处理消息块（四轮循环）。对每个 512 位数据块进行四轮变换处理，每轮变换包含 16 次操作，使用不同的逻辑函数和位移量，四轮变换使用不同的逻辑函数：F、G、H、I，每轮结束后更新哈希状态。

```
    for (int i = 0; i < n_blocks; i++) {
        bit32 x[16];
        // 准备消息调度
        for (int j = 0; j < 16; j++) {
            x[j] = (paddedMessage[i * 64 + j * 4]) |
                   (paddedMessage[i * 64 + j * 4 + 1] << 8) |
                   (paddedMessage[i * 64 + j * 4 + 2] << 16) |
                   (paddedMessage[i * 64 + j * 4 + 3] << 24);
        }
        bit32 a = state[0], b = state[1], c = state[2], d = state[3];
        // Round 1
        for (int j = 0; j < 16; j++) {
            bit32 f = F_gpu(b, c, d);
            bit32 temp = d;
            d = c;
            c = b;
            b = b + rotateLeft_gpu(a + f + d_k[j] + x[j], d_s[j]);
            a = temp;
        }
        // Round 2
        • • •
        // Round 3
```

```
        ...
        // Round 4
        ...
        state[0] += a;
        ...
    }
```

结果处理。将计算结果从大端序转换为小端序（符合 MD5 标准），并存储到输出数组。

```
    for (int i = 0; i < 4; i++) {
        uint32_t value = state[i];
        d_results[idx * 4 + i] = ((value & 0xff) << 24) |
                                 ((value & 0xff00) << 8) |
                                 ((value & 0xff0000) >> 8) |
                                 ((value & 0xff000000) >> 24);
    }
```

MD5Hash_GPU_Batch()

主机端函数负责整个批量 MD5 计算的流程控制。

输入处理与内存准备。计算所有输入字符串的总长度，并分配连续内存存储所有字符串。

```

    void MD5Hash_GPU_Batch(const vector<string>& passwords, vector<bit32*>& results) {
    int num_passwords = passwords.size();
    if (num_passwords == 0) return;

    // 计算总的字符串长度
    int total_chars = 0;
    for (const string& pw : passwords) {
        total_chars += pw.length() + 1; // +1 for null terminator
    }

    // 准备主机端数据
    char* h_password_data = new char[total_chars];
    int* h_lengths = new int[num_passwords];
    int* h_offsets = new int[num_passwords];

    int offset = 0;
    for (int i = 0; i < num_passwords; i++) {
        h_offsets[i] = offset;
        h_lengths[i] = passwords[i].length();
        strcpy(h_password_data + offset, passwords[i].c_str());
        offset += passwords[i].length() + 1;
    }
```

GPU 内存分配与数据传输。在 GPU 上分配内存，并将输入数据从主机复制到 GPU。

```
    char* d_password_data;
    int* d_lengths;
```

```
    int* d_offsets;
    bit32* d_results;

    cudaMalloc(&d_password_data, total_chars);
    cudaMalloc(&d_lengths, num_passwords * sizeof(int));
    cudaMalloc(&d_offsets, num_passwords * sizeof(int));
    cudaMalloc(&d_results, num_passwords * 4 * sizeof(bit32));

    // 复制数据到GPU
    cudaMemcpy(d_password_data, h_password_data, total_chars, cudaMemcpyHostToDevice);
    cudaMemcpy(d_lengths, h_lengths, num_passwords * sizeof(int),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_offsets, h_offsets, num_passwords * sizeof(int),
        cudaMemcpyHostToDevice);
```

核函数调用与同步。配置线程块和网格，启动核函数进行并行计算，并处理可能的错误。

```
    int threadsPerBlock = 256;
    int blocksPerGrid = (num_passwords + threadsPerBlock - 1) / threadsPerBlock;
    md5_kernel_batch<<<blocksPerGrid, threadsPerBlock>>>(d_password_data, d_lengths,
        d_offsets, num_passwords, d_results);

    // 等待GPU完成
    cudaDeviceSynchronize();

    // 检查错误
    cudaError_t error = cudaGetLastError();
    if (error != cudaSuccess) {
        fprintf(stderr, "CUDA error: %s\n", cudaGetErrorString(error));
    }
```

结果处理与内存清理。将计算结果从 GPU 复制回主机，整理结果格式，并释放分配的内存。

```
    bit32* h_results = new bit32[num_passwords * 4];
    cudaMemcpy(h_results, d_results, num_passwords * 4 * sizeof(bit32),
        cudaMemcpyDeviceToHost);

    results.clear();
    for (int i = 0; i < num_passwords; i++) {
        bit32* result = new bit32[4];
        for (int j = 0; j < 4; j++) {
            result[j] = h_results[i * 4 + j];
        }
        results.push_back(result);
    }

    // 清理内存
    delete[] h_password_data;
    delete[] h_lengths;
```

```
    delete[] h_offsets;
    delete[] h_results;

    cudaFree(d_password_data);
    cudaFree(d_lengths);
    cudaFree(d_offsets);
    cudaFree(d_results);
}
```

2.2.3 性能测试

编译选项	串行算法 Hash 时间 (s)	CUDA 优化 Hash 时间 (s)	加速比
不编译优化	5.86791	1.57941	3.715
O1 优化	2.20474	1.13427	1.944
O2 优化	2.16091	1.09704	1.970

表 4: md5 哈希函数 CUDA 优化测试结果

2.2.4 结果分析

无论是不进行编译优化，还是 O1 和 O2 优化，CUDA 优化的版本都比串行算法的 Hash 时间快。这一结果表明，利用 GPU 的并行计算能力能够显著提升 MD5 哈希计算的效率。具体来看，不进行编译优化时，串行算法的 Hash 时间为 5.86791 秒，而 CUDA 优化版本的 Hash 时间为 1.57941 秒，加速比达到了 3.715。这说明在没有进行任何编译优化的情况下，仅通过将计算任务从 CPU 转移到 GPU，就能获得超过 3 倍的性能提升。这一显著的加速效果主要得益于 GPU 的高并行度，能够同时处理多个密码的哈希计算，从而大大减少了总的计算时间。

然而，当我们对比 O1 和 O2 优化版本时，发现其加速比仅约为不进行编译优化的 1/2。具体来说，O1 优化版本的串行算法 Hash 时间为 2.20474 秒，CUDA 优化版本的 Hash 时间为 1.13427 秒，加速比为 1.944；O2 优化版本的串行算法 Hash 时间为 2.16091 秒，CUDA 优化版本的 Hash 时间为 1.09704 秒，加速比为 1.970。这一现象充分体现了编译优化对于 GPU 优化力度不如对于 CPU 的优化。原因在于，编译优化主要针对代码的执行效率进行改进，例如通过循环展开、指令调度等方式减少 CPU 执行指令的数量和时间。这些优化措施在 CPU 上能够显著提升程序的运行速度，但对于 GPU 来说，其并行计算的优势已经在硬件层面得到了充分发挥，进一步的编译优化对 GPU 的性能提升相对有限。此外，GPU 的架构和执行模型与 CPU 存在较大差异，一些针对 CPU 的优化策略可能并不适用于 GPU，甚至可能会引入额外的开销，从而影响加速比的提升。

2.3 思考

2.3.1 超过 64 字节的长口令的处理

对于 SIMD 向量化优化时，存在很长的口令的情况，比如口令长度超过 56 字节，则经过 StringProcess() 之后会变成 64*2、64*3、64*4... 的长度。此时由于处理后口令长度的不确定。导致我们进行向量化时无法直接从标量转为向量。解决的想法：从上述实验的测试可以看到，当 block 数量为 1 时，

4 条口令并行的 SIMD 方式能提升接近一倍的性能，2 条口令并行的 SIMD 方式也能提升 8% 的性能。因此提出一种方法：

(1) 求出 4 条口令 block 的长度，取最大值 MAX 和最小值 MIN。

(2)MAX=MIN，直接进行向量化处理。

(3)MAX>MIN，从 MIN 处进行 4 条口令 block 的截断。

出现以下三种情况：

(3.1)MIN 与另一口令长度相等，MAX 与另一口令长度相等，则将 4 条剩余 block 分成两组进行向量化处理。

(3.2) 其中两条口令 block 相等，且非 (3.1) 情况，则将相等两条口令剩余 block 进行向量化处理，余下两条口令 block 串行处理。

(3.3)4 条口令剩余 block 串行处理。

2.3.2 WSL2 上运行 NEON 版本

WSL2 上运行 NEON 版本，测试出的加速比会与在 ARM 平台上测试的结果相差较多。查阅资料分析:

(1) 虚拟化环境中的内存访问效率低于物理机。

(2)SIMD 指令的硬件加速在虚拟化环境中可能被部分屏蔽。

(3)QEMU 模拟器性能低。

3 Generate() 口令猜测优化

3.1 多线程优化

3.1.1 并行思路

对 preterminal 中最后一个 segment 不进行初始化和改变。在 preterminal 从优先队列中弹出的时候，直接一次性将所有可能的 value 赋予这个 preterminal。在我们的代码框架中虽然使用了并行思路，但还是使用着串行的代码写法。本次实验目标即为对 Generate() 函数进行多线程并行化改造。原函数通过循环生成口令猜测，核心操作为 emplace_back() 向全局容器添加元素以及总猜测数自增操作。

3.1.2 并行分析与设计

在处理从优先队列弹出的“preterminal”时，Generate() 函数需将所有猜测口令的取值（value）赋予一个全局的 guesses 容器。这一过程在代码中体现为通过 emplace_back() 操作向容器添加元素，是性能优化的关键目标。为提升该操作的执行效率，我们尝试对其循环结构进行多线程改造，将 preterminal 生成的 value 均分为 num_thread 块，每一块单独使用一个线程执行。但初始测试结果未达预期。性能瓶颈主要源于两方面：其一，线程的频繁创建与销毁带来显著开销；其二，并行生成的局部结果需要进行全局整合，而这一过程存在线程安全隐患。

针对全局整合问题，我们评估了两种主要解决方案：(1) 同步整合策略。等待所有线程执行完毕后再统一合并局部结果。此方案虽能保证数据一致性，但会引入较长的线程等待时间。(2) 互斥锁机制。通过加锁保护共享资源，避免数据竞争。然而，锁的频繁获取与释放会消耗额外的系统资源，性能退化明显。

上述两种方案均会引入不可忽视的额外开销，因此我们提出以下优化策略：

(1) 线程资源池化：在 PriorityQueue 类中预先创建并维护固定数量（num_thread）的线程句柄（threadId）及对应的参数结构体（threadargs），避免重复创建销毁线程参数带来的开销。

(2) 局部结果隔离：为每个线程分配独立的局部结果容器（local_guesses），替代直接操作全局容器。同时，将统计计数操作从线程内部移至外部，通过直接累加区间长度（如 pt.max_indices[0]）的方式替代逐次递增（total_guesses += 1），彻底消除自增操作的线程同步需求。

3.1.3 阈值机制

为攻克性能瓶颈，需引入阈值机制以实现多线程与串行算法的切换。该阈值本质上是资源消耗与多线程理论收益的平衡点——当 PT 生成的口令猜测数量超过此阈值时，多线程并行处理的收益将覆盖其资源开销，此时启用多线程；反之，当猜测数量低于阈值时，串行算法因避免了线程管理开销而更具效率。

阈值影响因素

阈值的具体取值受多重因素影响：(1) 多线程实现方式：底层线程库（如 pthread）与高层并行框架（如 OpenMP）的线程创建、同步机制不同，导致资源开销差异显著。(2) 编程接口特性：不同 API 对线程生命周期管理、负载均衡的支持程度会直接影响阈值边界。(3) 硬件平台架构：CPU 核心数、缓存容量、内存带宽等硬件参数决定了并行任务的实际执行效率与资源消耗上限。

测试方法论

首先通过测试阈值为 10，100，1000 •••••• 测试得到阈值的数量级。后面采用采用二分法动态缩小阈值范围，通过持续迭代逼近最优解。由于线程数与资源开销呈正相关，不同线程数的阈值理论上呈线性比例关系，可利用已知线程数的阈值（如 8 线程）推导其他线程数（如 2、4 线程）的阈值范围，减少测试复杂度。

阈值取值

经过实验测定，不同框架与线程数的阈值设定如下：

pthread：2 线程:2500，4 线程:4000，8 线程:10500

OpenMP：2 线程:1200，4 线程:1700，8 线程:2500

以 8 线程的 pthread 版本为例，以折线图展示阈值设定为 104 量级的过程。

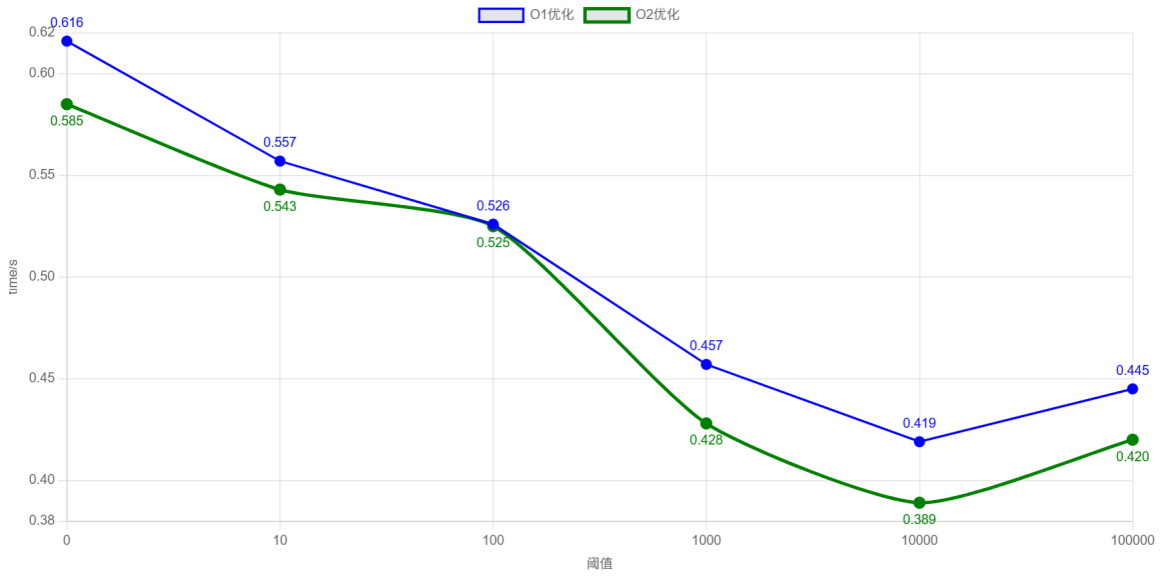

图 3.2: 总猜测口令数为 1000 万的阈值测试

结果局限性

环境波动性：服务器执行时间存在波动，难以获得绝对精确的阈值。样本偏差：PT 分布在理论阈值的分布稀疏时将难以确定阈值。

3.1.4 pthread 和 openmp 并行实现

pthread 和 openmp 版本的并行优化实现在多线程实验报告中有详细介绍，这里便不再赘述。

3.1.5 线程池实现

线程池类声明

核心组件：包含工作线程集合和任务队列。 任务提交接口：enqueue 支持任意可调用对象，返回 std::future 获取结果。同步机制：双条件变量实现任务通知 (condition) 和完成等待 (wait_condition)。 原子操作：stop 和 active_tasks 使用原子类型确保线程安全。 自动检测并发数：默认使用 std::thread::hardware_concurrency() 获取最优线程数。

```cpp
//ThreadPool.h
class ThreadPool {
public:
    ThreadPool(size_t num_threads = std::thread::hardware_concurrency());
    ~ThreadPool();

    // 提交任务到线程池
    template<typename F, typename... Args>
    auto enqueue(F&& f, Args&&... args)
        -> std::future<typename std::result_of<F(Args...)>::type>;

    // 等待所有任务完成
    void wait_for_all();

    // 获取线程数量
    size_t get_thread_count() const { return threads.size(); }

    // 检查线程池是否停止
    bool is_stopped() const { return stop; }

private:
    std::vector<std::thread> threads;
    std::queue<std::function<void()>> tasks;
    std::mutex queue_mutex;
    std::condition_variable condition;
    std::atomic<bool> stop;
    std::atomic<int> active_tasks;
    std::condition_variable wait_condition;
    std::mutex wait_mutex;
};
```

任务提交实现

完美转发：std::forward 保持参数值类别（左值/右值）。 任务封装：使用 std::packaged_task + std::shared_ptr 组合。packaged_task：封装可调用对象及其返回值、shared_ptr：延长任务生命周期避免悬空引用。 类型推导：std::result_of 自动推导返回类型。 异常安全：任务封装过程在锁作用域外执行，减少锁持有时间。

```cpp
//ThreadPool.h
template<typename F, typename... Args>
auto ThreadPool::enqueue(F&& f, Args&&... args)
    -> std::future<typename std::result_of<F(Args...)>::type> {

    using return_type = typename std::result_of<F(Args...)>::type;

    auto task = std::make_shared<std::packaged_task<return_type()>>(
        std::bind(std::forward<F>(f), std::forward<Args>(args)...)
    );

    std::future<return_type> res = task->get_future();

    {
        std::unique_lock<std::mutex> lock(queue_mutex);

        if (stop) {
            throw std::runtime_error("enqueue on stopped ThreadPool");
        }

        tasks.emplace([task]() {
            (*task)();
        });
    }

    condition.notify_one();
    return res;
}
```

工作线程核心逻辑

双重锁策略：queue_mutex：保护任务队列操作; wait_mutex：保护完成条件判断。 条件变量等待条件：condition.wait() 包含谓词检查。 任务执行分离：任务执行在无锁环境下进行，避免阻塞其他操作。 活跃任务追踪：active_tasks 精确计数当前运行任务数。

```cpp
//ThreadPool.cpp
ThreadPool::ThreadPool(size_t num_threads) : stop(false), active_tasks(0) {
    for (size_t i = 0; i < num_threads; ++i) {
        threads.emplace_back([this] {
            for (;;) {
                std::function<void()> task;

                {
                    std::unique_lock<std::mutex> lock(queue_mutex);
                    condition.wait(lock, [this] {
```

```
                    return stop || !tasks.empty();
                });

                if (stop && tasks.empty()) {
                    return;
                }

                task = std::move(tasks.front());
                tasks.pop();
                active_tasks++;
            }

            task();

            {
                std::unique_lock<std::mutex> lock(wait_mutex);
                active_tasks--;
                if (active_tasks == 0 && tasks.empty()) {
                    wait_condition.notify_all();
                }
            }
        }
    });
  }
}
```

线程池析构函数

原子停止标志：stop 确保所有线程可见性。 通知广播：notify_all() 唤醒所有休眠线程。 顺序终止：先设置标记再唤醒最后等待 join。 异常安全：即使任务队列非空也保证安全退出 。

```
ThreadPool::~ThreadPool() {
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        stop = true;
    }

    condition.notify_all();

    for (std::thread &worker : threads) {
        worker.join();
    }
}
```

任务完成等待实现

双条件检查：任务队列空 && 无活跃任务。 虚假唤醒防护：条件变量等待包含谓词检查。 无忙等待：条件变量实现高效阻塞。 独立锁机制：避免与任务提交锁冲突。

```
void ThreadPool::wait_for_all() {
    std::unique_lock<std::mutex> lock(wait_mutex);
```

```
    wait_condition.wait(lock, [this] {
        return tasks.empty() && active_tasks == 0;
    });
}
```

3.1.6 样本分布与阈值关联分析

数据分布

统计了猜测口令数量的分布，每 1000 条口令作为一个区间，统计此区间上的 PT 个数。

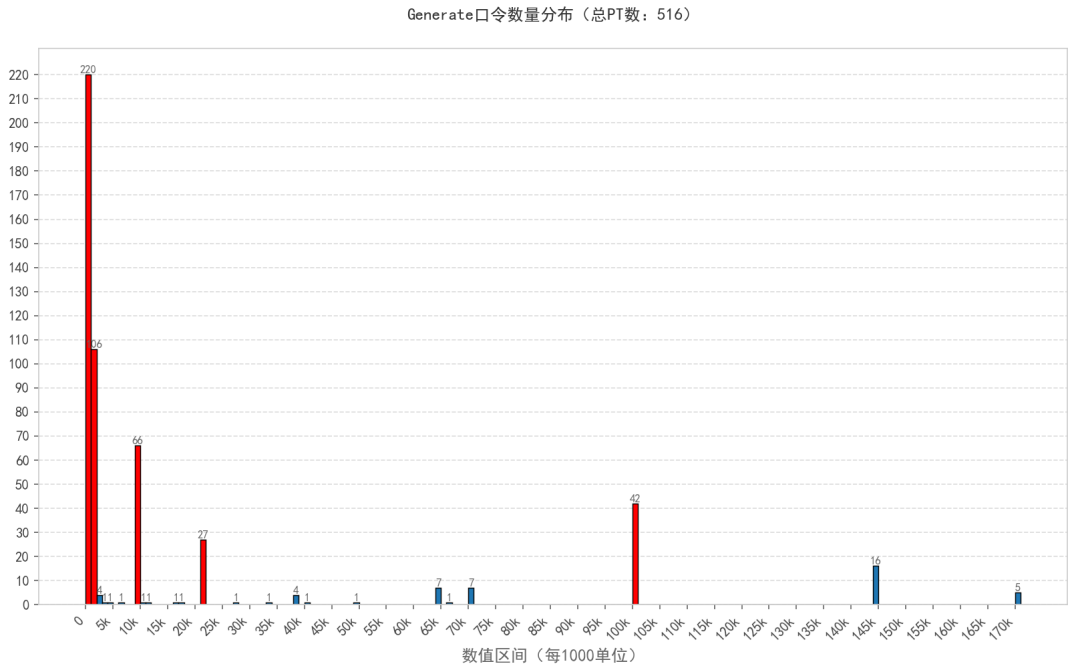

图 3.3: 总猜测口令数为 1000 万的 PT 分布

其中，红色的柱形是占总 PT 个数的比例超过 5% 的区间。

阈值设定的必要性

可以看到如果不进行阈值设置，对于多线程的资源开销由轻量级任务主导。[0,2000) 区间 PT 数占比 63%，多线程资源开销占比可估计为 63%，但总猜测数仅占 3%，串行处理耗时可忽略。花费了大量的多线程资源，加速占比极小的部分。而阈值的设定正好可以解决这一问题。

阈值设定的难度

数据分布具有部分区间稀疏性。[2000,10000) 和 [11000,21000) 区间 PT 数占比约 2%，异常的空旷，估计的 pthread 阈值又恰巧在这一区间，导致 pthread 阈值在该范围难以精确校准。OpenMP 因线程池等机制，阈值会更接近于 [0,2000) 的样本集中值，阈值也更好确定，而 pthread 需更严格的阈值控制。不过由于服务器测试时间波动，openMP 版本阈值也并不容易确定。想要检测目前设定的阈值的合理性需要对总猜测数更大的情况进行测试，以此减小服务器波动影响。这一过程在总猜测数对性能影响的分析中会有所体现。

3.1.7 编译优化选项对加速比的影响

性能测试

多线程版本	不编译优化	O1 优化	O2 优化
pthread	1.068	1.406	1.473
openmp	1.090	1.725	1.943

表 5: 多线程不同编译选项加速比结果

在数据表中，无论是 pthread 版本还是 openmp 版本，O2 的加速比最高，加速效果非常明显；O1 的加速比次之；不进行编译优化能够实现加速，但是效果不明显。

结果分析

对于串行代码，O1 优化时，编译器会将循环条件外移，对于 emplace_back() 的对象直接构造。O2 优化时，编译器会进行循环展开，可仍然是对 guesses 动态数组进行操作，存在着很大的数据依赖。

对于并行代码，O1 优化时，编译器会将循环条件外移，对于 emplace_back() 的对象直接构造。同时将各线程的地址缓存在寄存器中，避免重复计算线程位置的偏移。O2 优化时，编译器会进行循环展开，可这时每一个线程和参数都是独立的，能够更好的适配循环展开的优化。

简言之，多线程算法下，O1 优化不仅进行了串行同样的优化，还减小了并行部分线程需要频繁访问线程局部变量的开销；O2 优化进行循环展开，且并行部分是无数据依赖的，而串行的循环展开是有数据依赖的。所以不进行编译优化，O1 优化和 O2 优化的加速比随着优化程度的增大而增大。

3.1.8 pthread 与 openMP 的性能对比

性能测试

编译选项	pthread 加速比	openMP 加速比
不编译优化	1.068	1.090
O1 优化	1.406	1.725
O2 优化	1.473	1.943

表 6: pthread 和 openMP 加速比对比结果

多线程版本	L1 命中率	L2 命中率	L3 命中率
pthread	97.93%	85.63%	47.46%
openMP	98.17%	83.14%	57.22%

表 7: pthread 和 openMP 缓存命中率

结果分析

OpenMP 是一种基于指令的高层并行编程模型，通过编译器指令（如 #pragma omp parallel）自动管理线程的创建、销毁和任务分配。pthread 是底层的线程库，需要开发者显式管理线程生命周期、同步机制（如互斥锁、条件变量）和负载均衡。先前的多线程实验中主要是以下方面体现出 openMP 的效果更优：

(1) 线程池复用

从 vTune 结果中可以看到以下信息：OpenMP 默认使用线程池，在多次并行区域中复用线程，避

免重复创建和销毁线程的开销。并且如图 4.7 所示，每一个线程的负载比较均衡。Pthread 每次执行需显式调用 pthread_create，最终导致创建和销毁了大量的线程，线程创建和销毁的系统调用开销较高。

(2) 循环展开

OpenMP：编译器能直接识别 #pragma omp for 的并行语义，对循环进行激进优化。Pthread：线程函数（guess_thread）内的循环是通过手动分块的方式实现的，难以被编译器优化。

当然，初始时认为 openMP 的缓存优化会优秀很多，但是 profiling 中的数据显示，pthread 和 openMP 缓存命中率差距不大。

OpenMP 中 schedule(static) 采用块划分（Block Partitioning），将连续迭代块分配给线程，保证内存访问连续性，提高缓存命中率。Pthread 中手动划分的连续块（如 start 到 end）。二者的操作对缓存都很友好，此部分对性能影响不明显。

此外，其实在 pthread 的代码中的 PCFG.h 文件定义了全局的 pthread_t 类型的 8 个 threadId。但其实在我们的 pthread 写法中，使用这 8 个 threadId 的线程执行一次任务后就被销毁了，无法真正意义上的复用线程。只有通过线程池的方式才能真正复用线程。

性能测试

在期末实验中，实现了 pthread 版本的线程池，并进行了测试。vTune 与测试结果如下:

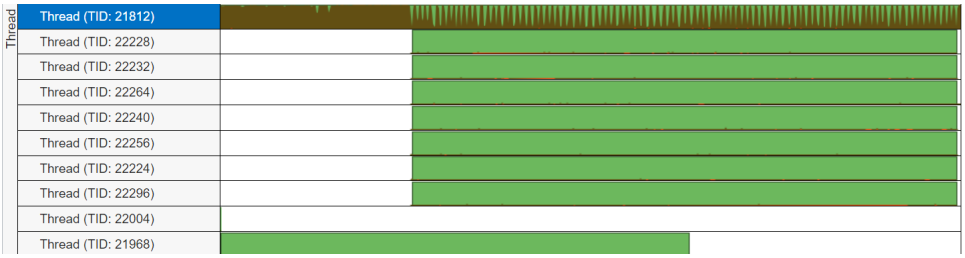

图 3.4: 线程池 pthread 多线程 vTune 结果

多线程版本	不编译优化	O1 优化	O2 优化
pthread	1.068	1.406	1.473
ThreadPool	1.082	1.620	1.772

表 8: 多线程不同编译选项加速比结果

可以清晰地看到线程池版本相较原 pthread 版本的实现有一定的加速，不过仍然低于 openmp 版本的实现。

结果分析

线程池版本相较于原始的 pthread 版本在性能上有所提升，主要得益于线程池减少了线程的创建和销毁开销。然而，线程池的实现仍然存在一些限制，例如任务分配和负载均衡的灵活性不足、硬件资源利用效率较低等。相比之下，OpenMP 作为一种高级并行编程模型，提供了更丰富的并行化指令和优化机制，能够更好地利用硬件资源，实现更高效的并行计算。因此，尽管线程池版本在性能上有所提升，但其加速效果仍然不如 OpenMP 版本。

3.1.9 Guess 和 Hash 同时加速

性能测试与结果分析

编译优化选项	串行 Hash 时间	并行 Hash 时间	Hash 时间加速比	Guess 时间加速比
不编译优化	9.454	12.091	0.782	1.068
O1 优化	3.077	1.745	1.763	1.406
O2 优化	2.917	1.660	1.757	1.473

表 9: pthread 多线程结果

编译优化选项	串行 Hash 时间	并行 Hash 时间	Hash 时间加速比	Guess 时间加速比
不编译优化	9.454	12.028	0.786	1.090
O1 优化	3.077	1.757	1.751	1.725
O2 优化	2.917	1.689	1.727	1.943

表 10: openMP 多线程结果

Md5 Hash 部分使用的是 4 条口令并发处理的 NEON 版本，Guess 部分使用的是 8 条口令并发处理的 pthread 版本。理论上来说这两个版本并不存在操作上的冲突，所以可以同时进行优化。

如表 9 和表 10 所示，最终结果为将 Hash 和 Guess 的加速部分合并后，Hash 时间得到的优化和 Guess 时间得到的优化与未合并时的结果相同，也符合理论上的这两个过程的独立性。只是在实现时，我们并未按照概率降序进行 Md5 Hash 处理，如果加上这一步骤，则 Guess 时间和 Hash 时间的优化均会略有降低。

3.1.10 总线程数对加速比的影响

性能测试

pthread 多线程结果			
总线程数	不编译优化	O1 优化	O2 优化
2	1.030	1.238	1.300
4	1.054	1.297	1.326
8	1.057	1.457	1.517
openMP 多线程结果			
总线程数	不编译优化	O1 优化	O2 优化
2	1.050	1.388	1.402
4	1.089	1.630	1.809
8	1.106	1.858	1.872

表 11: 不同线程数加速比结果

pthread 多线程和 openMP 多线程下，无论是不编译优化，O1 优化还是 O2 优化，总线程数越大，加速比越大。

结果分析

在阈值取值合理的情况下，理论上线程数越高，加速比越大。这是由于同时进行的任务越多，并

行度越高。和我们的测试结果也是一致的。但是在这个问题上，不一定会出现线程数越高，加速比越大的现象。虽然阈值不随样本数据的改变而改变，但可能会出现样本全部在 8 线程阈值内等情况。此时 4 线程必然是比 8 线程快的。所以对于不同分布的数据应该采取不同的线程数的优化。

3.1.11 总猜测数对性能的影响

对于总猜测数为 10 亿的情况下，执行时间会很长。尤其对于串行算法来说，10 亿条口令的结果无法在 10 分钟内完成，而由于服务器无法测试超过 10 分钟的代码。所以这里就不进行 10 亿条口令的加速比的计算。但是会记录并行算法在 10 亿条口令时的执行时间。

性能测试

pthread 多线程加速比结果				
总线程数	总猜测数	不编译优化	O1 优化	O2 优化
2	1000 万	1.054	1.158	1.289
	1 亿	1.030	1.238	1.300
4	1000 万	1.041	1.384	1.455
	1 亿	1.054	1.297	1.326
8	1000 万	1.068	1.406	1.473
	1 亿	1.057	1.457	1.517
openMP 多线程加速比结果				
总线程数	总猜测数	不编译优化	O1 优化	O2 优化
2	1000 万	1.038	1.502	1.539
	1 亿	1.050	1.388	1.402
4	1000 万	1.040	1.625	1.630
	1 亿	1.089	1.630	1.809
8	1000 万	1.090	1.725	1.943
	1 亿	1.106	1.858	1.872

表 12: 不同总猜测数加速比结果

调整了总猜测数后，O2 优化下，pthread 版本中，8 线程数的加速比在 1000 万和 1 亿的情况下均是最大的，不过 2，8 线程数版本在总猜测数 1 亿时的加速比均比 1000 万时的加速比大，而 4 线程版本的加速比在总猜测数 1 亿时比 1000 万时更小；openMP 版本中，8 线程数的加速比在 1000 万和 1 亿的情况下均是最大的，不过 2，8 线程数版本在总猜测数 1 亿时的加速比均比 1000 万时的加速比小，但是 4 线程版本的加速比在总猜测数 1 亿时比 1000 万时更大。

当猜测总数达到了 10 亿，不进行编译优化是无法在服务器上得到结果的，表中便用“/”代替。这里只关注 Guess 的时间，我们可以看到 pthread 版本中，各线程版本的时间是差不多的，openMP 版本中线程数愈大，Guess 时间越短。

结果分析

绘制 1 亿和 10 亿时的样本分布情况图。

pthread 多线程 Guess 时间结果			
总线程数	不编译优化	O1 优化	O2 优化
2	/	133.828	123.348
4	/	133.035	120.366
8	/	133.846	123.357
openMP 多线程 Guess 时间 (s) 结果			
总线程数	不编译优化	O1 优化	O2 优化
2	/	143.255	121.996
4	/	122.205	120.384
8	/	120.025	112.441

表 13: 总猜测数 10 亿时 Guess 时间结果

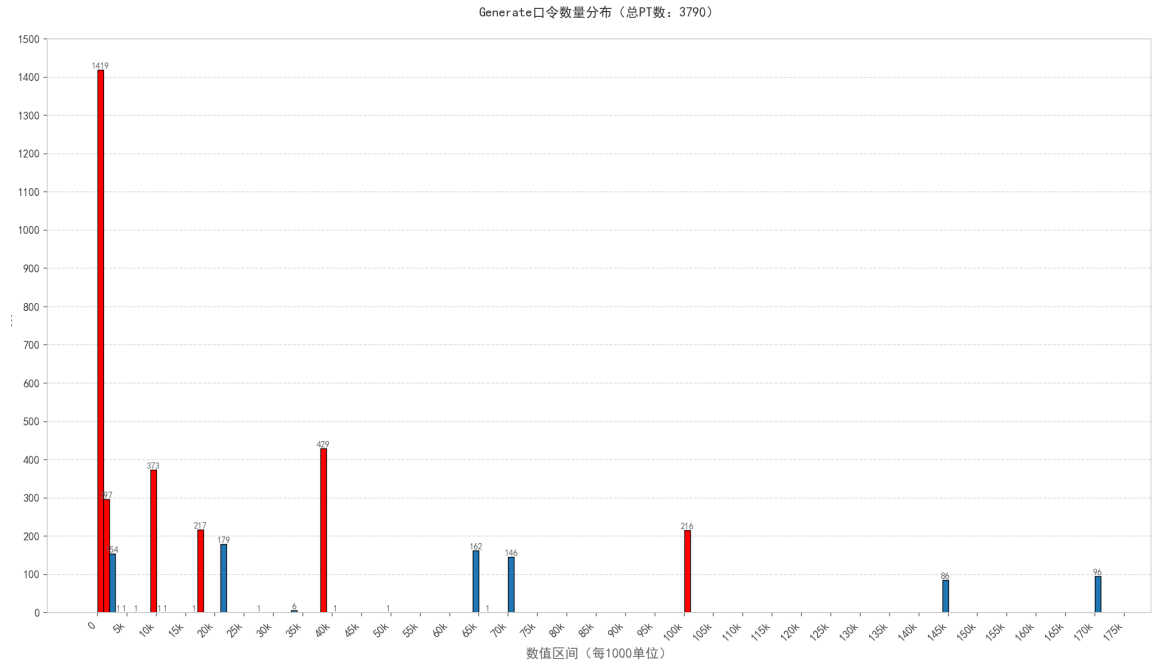

图 3.5: 总猜测数 1 亿时的 PT 分布

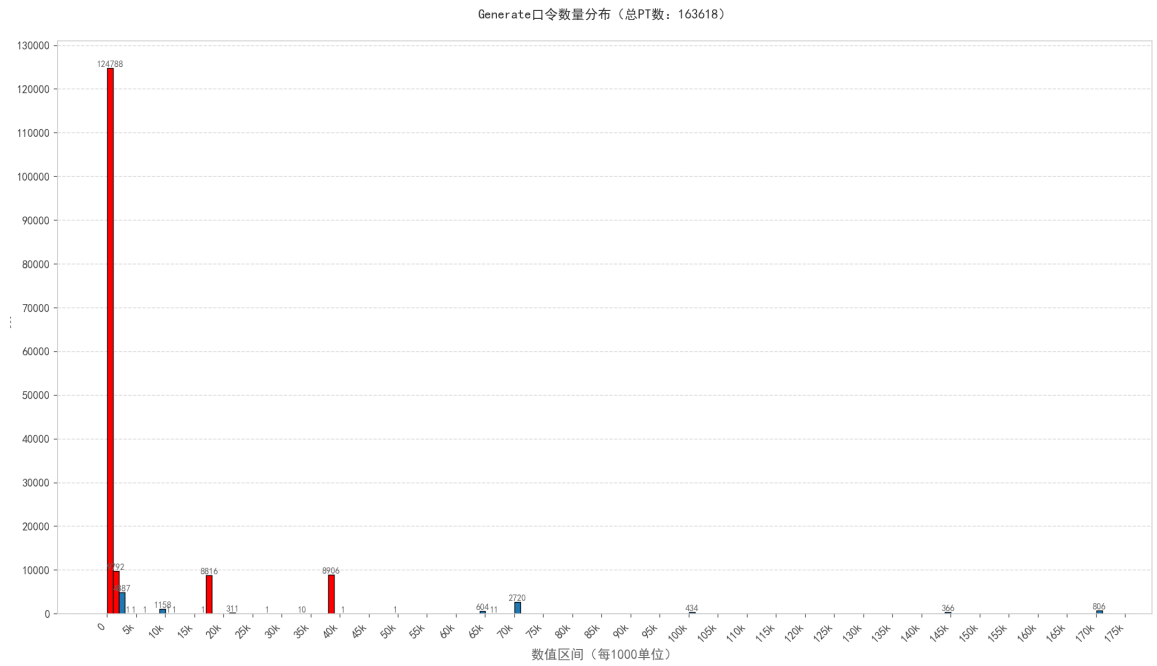

图 3.6: 总猜测数 10 亿的 PT 分布

随着总猜测数量的增加，[0,1000) 的 PT 数占比一直在增大，到了总猜测数 10 亿时，占比已经高达 76.3%。但是 Guess 时间占比仍然不超过 10%。所以只要阈值设定比 1000 大，实现方式不太糟糕，多线程版本几乎就是可以实现加速的。但是想要达到最优的性能是需要更精细的筛选的。

3.2 mpi 多进程优化

多线程实验中，我们已通过 pthread 和 OpenMP 实现了单个 Preterminal（PT）生成密码猜测时的内部并行化，通过阈值机制动态切换串行与并行算法。本次实验在此基础上，基于 MPI（Message Passing Interface）多进程模型，进一步对密码猜测过程进行优化，主要包括单个 PT 生成过程的并行化、多 PT 并发处理以及流水线式的口令生成与哈希计算协同优化，旨在探索并行程序设计在密码学中的应用潜力并提升攻击效率。

3.2.1 单 PT 多进程设计

单 PT 多进程设计围绕 Generate() 函数展开，该函数负责将优先队列中弹出的”preterminal”生成具体口令并存储至全局 guesses 容器。核心优化点在于将生成过程的循环结构改造为多进程并行模式：通过将 preterminal 生成的口令值均分为 size 个数据块，每个进程独立处理一块数据，最终汇总结果。这种设计的关键优势在于：

(1) 负载均衡。通过动态计算每个进程的任务区间，确保前 remainder 个进程多分配 1 个任务，后续进程分配标准任务量，避免进程间负载失衡。

(2) 减少全局访问。各进程维护本地猜测列表，仅在最终阶段合并至全局容器，降低并发访问冲突。

3.2.2 进程数对性能的影响

选择对优化效果最好的 O1 进行进程数对性能影响的实验。

性能测试

进程数	2 进程	4 进程	8 进程	16 进程
加速比	1.705	2.133	2.040	0.019

表 14: 单 PT 不同进程数加速比

结果分析

(1) 并行开销：当进程数增加时，并行开销（如进程间通信、同步等）也随之增加。在进程数较少时，这些开销相对较小，因此加速比随着进程数的增加而增加。但是当进程数过多时，这些并行开销可能会超过并行带来的性能提升，导致加速下降比。

(2) 资源竞争：多个进程同时运行时，系统资源（如 CPU、内存等）可能会出现竞争。在进程数较少时，资源竞争相对较小，进程可以充分利用系统资源。但进程数过多时，资源竞争加剧，可能导致部分进程无法及时获取所需资源，从而影响整体性能。从 8 进程比 4 进程慢看出 4 进程更适合完成这项任务, 对于并行带来的收益以及并行开销的平衡更优。

不过更值得分析的是 16 进程下的性能急速下降，加速比仅为 0.019。搜索资料后，我认为是以下的原因:

(1) 任务划分逻辑中，当进程数 $p = 16$ 时：若口令段总长度 total 较小，会出现任务碎片化（单个进程分配到的任务量接近 0）进程频繁处于空闲等待状态，而通信同步仍持续消耗资源，导致整体效率暴跌。

(2) 多进程共享计算节点内存带宽时,16 进程同时访问 guesses 容器、共享数据段（如 a->ordered_values），引发内存带宽争用。尤其在 emplace_back 操作时，大量进程并发写内存可能触发缓存一致性风暴，大幅降低内存访问效率。

3.2.3 多 PT 多进程设计

处理多个 PT 时采用层次化的并行策略，将计算任务分解为三个主要阶段：批量获取、并行处理和结果聚合。

任务分发

主进程从优先队列中批量弹出多个 PT，并采用固定批量大小控制每次处理的任务量。同时，动态调整实际处理的 PT 数量，避免队列过小时的无效操作。

并行处理

将 PT 批次按进程数动态分配，确保负载均衡，同时，每个进程独立处理分配的 PT，生成局部猜测列表，并维护独立的局部猜测列表，避免频繁全局访问。

结果聚合

对于猜测聚合,每个进程将本地生成的猜测直接追加到全局容器,对于计数聚合,使用 MPI_Allreduce 聚合各进程的猜测计数，更新全局总数。最后所有进程还需要同步生成新 PT，计算概率后插入优先队列。采用有序插入策略，确保队列按概率降序排列。

3.2.4 批次数对于多 PT 并行优化的影响

性能测试

这里的批次数 batch_size 是指每次 PopNextBatch_mpi() 函数需要处理的 PT 数，那么每个进程也就要处理 batch_size/size 个 PT，size 是进程数。在探索的过程中，最终发现，是在主函数中的

PopNextBatch_mpi() 函数调用时，批次数为进程数，那么在 PopNextBatch_mpi() 的任务分配中，每个进程每次只处理一个 PT，导致了多进程资源开销过大，于是便无法实现加速。

这里的多 PT 执行时也是分配了 8 个进程。

编译选项	串行 Guess 时间 (s)	多 PT 并行 Guess 时间 (s)	加速比
不编译优化	12.979	104.297	0.124
O1 优化	0.561	2.628	0.213
O2 优化	0.581	2.849	0.204

表 15: 多 PT 并行优化加速比

结果分析

(1) 批次数过小的影响

当批次数过小时，每个进程每次处理的 PT 数量较少。在进程数为 8 的情况下，如果批次数仅为进程数，即每个进程每次只处理一个 PT，这会导致多进程资源开销过大，无法实现加速。具体而言，频繁地启动和切换进程会消耗大量的系统资源，包括 CPU 时间和内存。每个进程在处理一个 PT 后，需要进行上下文切换，将控制权交给其他进程，这个过程会产生额外的开销。此外，进程之间的同步和通信也会变得更加频繁，进一步增加了系统的负担。因此，批次数过小使得进程的优势没有得到充分发挥，系统性能反而受到影响。

(2) 批次数适中的优势

从测试结果来看，当批次倍数为 25 时，加速效果最好，加速比能够达到 1.565。这是因为在这个批次数下，每个进程能够处理相对合理数量的 PT，使得进程的并行计算能力得到了较好的发挥。进程在处理多个 PT 的过程中，可以减少上下文切换的次数，提高 CPU 的利用率。同时，适中的批次数也能够在一定程度上平衡进程之间的负载，避免某些进程空闲而其他进程繁忙的情况。这样，系统能够充分利用多进程的并行计算能力，从而实现较好的加速效果。

(3) 批次数过大的问题

随着批次倍数的继续增大，加速比会越来越低。当批次数过大时，虽然每个进程能够充分利用自身的资源，处理大量的 PT，但在结果的汇总过程中会产生大量的通信开销和维护队列的开销。具体来说，在多进程并行计算中，每个进程处理完自己分配的 PT 后，需要将结果汇总到主进程或其他进程进行统一处理。批次数越大，每个进程处理的结果就越多，进程之间的通信量也会相应增加。频繁的通信会导致网络延迟和带宽占用，降低系统的整体性能。此外，维护优先队列也会变得更加复杂和耗时。在 PopNextBatch_mpi() 函数中，处理完一批 PT 后，需要将新生成的 PT 插入到优先队列中。批次数越大，新生成的 PT 数量就越多，插入操作的时间复杂度也会相应增加，这会进一步影响系统的性能。当然，批次数过大也会带来其他问题，比如内存占用过大，降低缓存命中率。

3.2.5 流水线设计

这里在原有 Generate() 函数的基础上，增加了 Pipeline 优化。将口令破解任务分解为两个主要阶段并分配给不同进程：

(1) 进程 0 作为口令生成器。基于 PCFG (概率上下文无关文法) 模型生成可能的口令猜测，按照概率优先级队列组织和生成口令，并将生成的口令批量发送给哈希计算进程。

核心逻辑：当前进程作为口令生成器, 使用 Rockyou 数据集训练 PCFG 模型，计算口令概率分布。口令生成完成之后，需要先发送批量大小，再逐个发送口令长度和内容 (这种实现最简单，但是非常耗

费时间)，记录训练时间和总执行时间，输出生成的口令总数。并发送终止信号 (-1) 通知哈希计算进程结束任务。

(2) 进程 1 作为哈希计算器。接收来自口令生成进程的口令，对每个口令计算 MD5 哈希值，最后统计处理过的口令数量。

核心逻辑：当前进程作为哈希计算器，持续接收来自进程 0 的口令批量。先接收批量大小，再逐个接收口令长度和内容，并动态分配缓冲区存储口令，处理后立即释放。使用 MD5Hash 函数计算每个口令的哈希值后，当接收到终止信号 (-1) 时退出循环。

性能测试

在这一部分我们关注的是除了训练时间外，口令生成和哈希计算的总时间，在串行源码和我的类似 pipeline 的优化版本上都进行了这一部分的统计。最终结果如下：

编译选项	串行 Guess+Hash 时间 (s)	MPI 并行 Guess+Hash 时间 (s)	加速比
不编译优化	23.6004	30.9829	0.762
O1 优化	3.51221	7.4282	0.473
O2 优化	3.52399	7.4332	0.474

表 16: pipeline 优化加速比

无论是哪种编译选项下，类似 pipeline 的口令生成与哈希优化均无法实现加速。

结果分析

尽管我们的实现方式采用了阻塞式通信机制，实现了类似流水线的操作，但根据实验结果，哈希操作在整体任务中占比较大，这在 Guess 时间和 Guess+Hash 时间的测试结果中得到了体现。

在哈希计算过程中，口令生成器可以继续执行下一轮口令生成任务，但由于通信机制的限制，当新一轮口令生成完成后，必须等待当前哈希操作结束才能将新生成的口令传输给哈希进程。这意味着，只有在哈希进程成功接收口令后，口令生成器才能开展新一轮的执行工作。正是由于两个进程之间的通信消耗远远超过了口令生成所需的时间，导致了流水线的效率低下，失去了流水线优化的初衷。

此外，在哈希操作执行期间，由于无法进行进程间的信息传输，流水线会被阻塞，使得口令生成进程需要长时间等待哈希进程开始接收新的口令，从而进一步加剧了流水线的效率问题。

综上所述，阻塞式通信机制的应用在流水线并行优化过程中存在明显的性能瓶颈，主要表现为通信时间过长以及流水线阻塞，这两方面问题共同导致了流水线效率的低下，使得流水线优化的实际效果未能达到预期。

3.3 GPU 优化

GPU 实验聚焦于运用 CUDA 并行计算技术对 PCFG 密码猜测算法开展深度优化工作。其主要研究目标涵盖以下方面：精心设计出真正的批量 GPU 处理机制；对内存管理策略进行优化升级；有效降低 GPU 与 CPU 之间的数据传输开销；并且达成智能的 CPU 与 GPU 负载均衡。

3.3.1 CUDA 并行设计

并行设计的核心目标是通过 GPU 的大规模并行计算能力，加速前缀与值的字符串拼接过程。其核心思路基于数据并行范式，将独立的字符串拼接任务分配给不同线程并行执行。具体设计如下：

1. 任务划分策略: 以“值”为最小任务单元，每个线程负责处理一个“前缀 + 值”的拼接任务。由于每个值的拼接过程相互独立（无数据依赖），天然适合并行化。通过线程索引与值的一一映射，确

保每个线程仅处理一个值，避免线程间的同步开销。

2. 内存布局优化: 采用扁平化数据存储。将所有值字符串存储在一个连续数组（d_values）中, 通过累积长度数组（d_valueLengths）记录每个值的起始偏移量和长度。这种设计减少了内存碎片，提高了 GPU 全局内存的访问效率（连续内存访问更易利用内存带宽）。输出缓冲区通过偏移量数组（d_outputOffsets）预分配，每个线程的输出结果存储在独立区域，避免线程间的内存冲突。

3.3.2 阈值优化

并行设计

阈值优化的核心是动态选择计算单元（CPU 或 GPU），以平衡不同数据规模下的计算效率。设计思路基于以下观察：GPU 虽擅长大规模并行计算，但存在核函数启动开销、数据传输开销；而 CPU 在小规模数据处理时，由于无需跨设备传输数据且单线程效率高，可能表现更优。具体设计如下：

1. 阈值划分逻辑: 设定一个阈值（threshold），当值的数量（numValues）超过阈值时，使用 GPU 加速；否则由 CPU 直接处理。阈值的选择需通过测试不同数据规模下 CPU 与 GPU 的性能差异确定，目标是让“GPU 加速收益”覆盖“启动与传输开销”。

2. 任务分配策略: 小规模任务（numValues ff threshold）：CPU 通过循环直接拼接字符串（guess + a->ordered_values [i]），避免 GPU 的额外开销。大规模任务（numValues > threshold）：复用基础并行设计中的 GPU 核函数，通过数据并行加速拼接过程。

实验结果与分析

编译选项	串行算法 Guess 时间 (s)	阈值优化 Guess 时间 (s)	加速比
不编译优化	8.36013	8.83531	0.947
O1 优化	0.425231	0.793109	0.536
O2 优化	0.423975	0.7526	0.563

表 17: 阈值优化测试结果

无编译优化时: 串行算法耗时 8.36s，阈值优化后耗时 8.84s，加速比 0.947。此时 CPU 串行性能较差，但 GPU 的启动开销（核函数初始化、上下文切换）和数据传输开销（主机到 GPU 的数据复制）抵消了并行计算的收益，导致整体效率略低于串行。

O1/O2 编译优化时: 编译优化显著提升了 CPU 性能（串行耗时从 8.36s 降至 0.42s 左右），而 GPU 由于硬件加速的优势未被充分发挥，导致阈值优化后的耗时（0.75-0.79s）显著高于串行。这说明：CPU 在编译优化后，单线程字符串拼接效率极高，小规模任务无需依赖 GPU；当前阈值设置可能不合理（如阈值过低，导致本应 CPU 处理的小任务被分配给 GPU），或 GPU 处理逻辑存在优化空间（如减少数据传输量）。但是后续发现当不断调高 threshold 的值时，加速比只会不断接近于 1，也就是说在目前的测试数据上 gpu 版本并不能很好的加速。可能是由于单 PT 产生的口令数仍然不够大，也可能是由于目前的实现方式带来的。

3.3.3 多 PT 优化

并行设计

多 PT（Production Template）优化的目标是通过批量处理多个 PT 任务，提高 GPU 的利用率。具体设计如下：

1. 批量任务映射: 将多个 PT 的任务整合为一个批次，通过线程索引映射到具体的 PT（pt_idx）和该 PT 内的值（local_password_idx）。线程通过循环遍历 PT 计数数组（d_pt_counts），确定自身负责的 PT 及值索引，实现多 PT 任务的并行处理。

2. 数据结构优化: 统一存储所有 PT 的前缀（d_all_prefixes）、值（d_all_values），通过偏移量数组（d_prefix_offsets、d_value_offsets）定位每个 PT 的数据，减少内存碎片。输出结果通过全局偏移量（d_output_offsets）分配，确保每个 PT 的拼接结果独立存储。

实验结果与分析

编译选项	串行算法 Guess 时间 (s)	CUDA 多 PT 优化 Guess 时间 (s)	加速比
不编译优化	8.36013	4.34102	1.926
O1 优化	0.425231	1.21496	0.350
O2 优化	0.423975	1.20877	0.351

表 18: CUDA 多 PT 优化测试结果

无编译优化时: 串行算法耗时 8.36s，多 PT 优化后耗时 4.34s，加速比 1.926。此时 CPU 串行处理多个 PT 的效率极低（需逐个处理每个 PT 的所有值），而 GPU 通过批量处理多个 PT，减少了核函数启动次数（一次启动处理多个 PT），并行计算的收益覆盖了数据传输开销，因此效率提升明显。

O1/O2 编译优化时: CPU 串行性能大幅提升（耗时 0.42s），而 GPU 的批量处理优势被以下因素削弱：数据传输开销：多 PT 任务需要传输更多数据（所有 PT 的前缀和值），主机到 GPU 的复制时间增加；线程映射开销：线程需通过循环 s 定位 PT 索引，增加了线程内的计算开销；内存访问效率：多 PT 的数据分散存储（虽通过偏移量整合，但全局内存访问模式不如单 PT 连续），导致内存带宽利用率降低。

3.3.4 混合并行流水线优化

并行设计

混合并行流水线优化的核心是实现 CPU 与 GPU 的协同计算，通过流水线并行掩盖彼此的等待时间。基础并行和多 PT 优化仅关注单一计算单元（GPU），而混合优化通过任务拆分与异步操作，让 CPU 和 GPU 在时间上重叠工作，具体设计如下：

1. 流水线阶段划分: CPU 预处理：计算 PT 概率、构建前缀、区分 CPU/GPU 任务（按阈值）；GPU 异步处理：批量处理大任务（数据传输、核函数执行异步进行）；CPU 维护队列：在 GPU 处理期间，CPU 并行合并新生成的 PT、更新优先队列，避免空闲。

2. 异步操作机制: 采用 CUDA 流（stream）实现 GPU 操作的异步化：数据传输（cudaMemcpyAsync）和核函数执行（«<..., stream»）无需等待 CPU，CPU 可同时进行队列维护或小规模任务处理，减少设备间的等待时间。

3. 数据一致性保障：通过偏移量数组和全局内存屏障，确保 CPU 与 GPU 访问的数据互不干扰；结果处理阶段（ProcessGPUResults）通过流同步（cudaStreamSynchronize）保证数据就绪后再解析，避免读取无效数据。

实验结果与分析

无编译优化: 相比多 PT 优化，混合流水线将耗时从 4.34s 降至 4.29s，加速比提升约 1.5 是因为流水线掩盖了 GPU 处理期间的 CPU 空闲时间：GPU 执行核函数时，CPU 同步合并新 PT、更新优先队列，减少了整体流程的串行等待时间。此时 CPU 性能较差，流水线的并行收益主要体现在“GPU

编译选项	串行算法 Guess 时间 (s)	CUDA 优化 Guess 时间 (s)	加速比
不编译优化	8.36013	4.28735	1.950
O1 优化	0.425231	1.05055	0.405
O2 优化	0.423975	1.04791	0.405

表 19: 混合并行流水线优化测试结果

计算与 CPU 队列维护的重叠”。

O1/O2 编译优化: 与多 PT 优化类似，CPU 性能的大幅提升（串行耗时 0.42s）使得 GPU 的开销（异步传输、流同步）成为瓶颈。具体原因包括：异步操作的管理开销（流创建、状态跟踪）抵消了并行收益；小规模任务场景下，CPU 单独处理已足够高效，流水线的协同优势难以体现；数据预处理（计算偏移量、扁平化存储）的复杂度增加，导致 CPU 预处理时间变长。

4 总结与展望

4.1 总结

在本学期的并行程序设计实验中，我们围绕 MD5 哈希函数和口令生成函数的并行化优化展开了一系列研究。通过 SIMD 向量化、多线程、多进程、GPU 加速等多种并行计算技术，我们逐步探索了如何高效地提升密码猜测任务的性能。

4.1.1 阈值机制的重要性

在本学期实验中，阈值机制被证明是多线程和多进程优化的关键。通过合理设置阈值，能够有效区分适合串行处理和并行处理的任务规模，从而在不同数据规模下实现性能优化。实验结果表明，阈值的设定需要根据具体的并行实现方式（如 pthread 或 OpenMP）、硬件平台架构以及任务特性进行调整。例如，在 pthread 版本中，8 线程的阈值设置为 10500 时，在总猜测数为 1000 万时能够实现较好的加速效果；而在 OpenMP 版本中，8 线程的阈值设置为 2500 时表现更优。此外，随着总猜测数的增加，数据分布发生变化，阈值也需要相应调整以适应新的任务规模分布。

4.1.2 并行度的选取

并行度的提升是实现加速的重要手段。在 MD5 哈希优化中，通过 SIMD 向量化技术实现了不同并行度（如 2、4、8）的优化，结果表明并行度越高，加速比越高。这是因为更高的并行度能够充分利用硬件资源，减少单位任务的处理时间。然而，并行度的提升也会带来额外的开销，如内存管理、线程同步等。在实际应用中，需要根据硬件资源和任务特性选择合适的并行度，以实现性能与资源开销的平衡。

4.1.3 规模/批次数的确定

对于多 PT 并行优化和流水线优化，批次数的合理设置对性能有着显著影响。在多 PT 并行优化中，当批次数设置为进程数的 25 倍时，加速比达到最佳（1.565）。过小的批次数会导致进程频繁切换和通信开销过大，而过大的批次数则会增加结果汇总和队列维护的开销。在流水线优化中，合理的批

次数能够有效掩盖 CPU 与 GPU 之间的等待时间，提高整体效率。因此，在设计并行算法时，需要根据任务规模和硬件资源合理设置批次数，以实现最优的性能。

4.1.4 GPU 加速条件

GPU 加速在处理大规模并行任务时表现出显著的优势，但在某些情况下也可能无法充分发挥其性能。实验结果表明，当任务规模较小时，GPU 的启动开销（如核函数初始化、数据传输）可能会抵消并行计算的收益。此外，GPU 的内存带宽和全局内存访问模式也会对性能产生影响。在多 PT 优化和混合并行流水线优化中，由于数据传输量增加和内存访问效率降低，GPU 加速的优势被削弱。因此，GPU 加速适用于大规模、计算密集型的任务，且需要优化数据传输和内存访问模式，以充分发挥其并行计算能力。

4.1.5 流水线

流水线优化通过任务拆分和异步操作，实现了 CPU 与 GPU 的协同计算，掩盖了彼此的等待时间。在无编译优化的情况下，混合并行流水线优化能够实现较好的加速效果，但随着编译优化程度的提高，CPU 性能显著提升，GPU 的开销成为瓶颈。这表明，在小规模任务场景下，CPU 单独处理可能更为高效，而在大规模任务场景下，流水线优化能够有效提升性能。因此，流水线优化适用于大规模、复杂的并行任务，且需要根据任务规模和硬件资源合理设计流水线阶段和异步操作机制。

4.2 展望

4.2.1 阈值动态调整机制

目前的阈值设置是静态的，依赖于预先测试和经验。未来可以研究实现自适应阈值调整算法，根据实时任务规模和硬件资源动态调整阈值。例如，可以利用机器学习算法根据历史执行时间和任务特征预测最优阈值，从而实现更灵活、高效的并行优化。

4.2.2 混合并行架构

结合 MPI、OpenMP 和线程池等技术，实现多级并行架构。例如，可以利用 MPI 在多个节点之间进行任务分配，每个节点内部使用 OpenMP 进行线程级并行处理，同时结合线程池减少线程创建和销毁的开销。这种混合并行架构能够充分利用不同层次的并行资源，提高整体性能。

4.2.3 通信优化策略

在 MPI 多进程优化和流水线优化中，通信开销是一个重要的性能瓶颈。未来可以研究采用 MPI 非阻塞通信（如 Isend/Irecv）实现计算与通信的重叠，减少通信等待时间。此外，可以设计高效的数据压缩算法，减少口令传输量，从而降低通信开销对整体性能的影响。

4.2.4 GPU 优化改进

尽管 GPU 在大规模并行任务中表现出显著优势，但在某些情况下仍存在性能瓶颈。未来可以进一步优化 GPU 的内存管理策略，减少数据传输开销，提高内存访问效率。例如，可以研究使用共享内存和纹理内存等优化技术，减少全局内存访问次数。此外，可以探索更高效的 GPU 任务分配策略，减少线程映射开销，充分发挥 GPU 的并行计算能力。

5 代码仓库

仓库链接 Gitee

参考文献

[1] Intrinsics - arm developer. https://developer.arm.com/architectures/instruction-sets/intrinsics.

[2] Prob-hashcat: Accelerating proba-bilistic password guessing with hashcat by hundreds of times. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=SSE_ALL.

[3] Ziyi Huang, Ding Wang, and Yunkai Zou. Prob-hashcat: Accelerating proba-bilistic password guessing with hashcat by hundreds of times. In Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses, 2024.