南開大學
Nankai University

计算机学院

并行程序设计报告

期末研究报告

姓名：高景珩

学号：2310648

专业：计算机科学与技术

2025 年 7 月 6 日

目录

1 实验背景

本学期，我围绕口令猜测这一选题，深入研究了基于概率上下文无关文法（PCFG, Probabilistic Context-Free Grammar）的口令猜测算法及其并行化实现。口令猜测作为信息安全领域的重要研究方向，旨在通过模拟攻击者的行为，生成可能的口令序列以破解用户密码，尤其在哈希破解等场景中具有广泛应用。本选题的核心任务包括模型训练、口令生成和 MD5 哈希计算三个主要模块，我针对这些模块进行了多次并行优化，采用了多种并行计算技术，包括 SIMD（单指令多数据）、多线程（基于 pthread 和 OpenMP）、多进程（基于 MPI）以及 GPU 并行等方法，均取得了显著的性能提升。

口令猜测的核心在于生成按照概率降序排列的口令序列，并通过哈希算法（如 MD5）验证其有效性。基于 PCFG 的口令猜测算法通过统计训练集中口令的结构（preterminal）和字段值（segments，如字母、数字、特殊字符）生成概率模型，从而高效地生成高概率的口令序列。然而，串行实现的 PCFG 算法在处理大规模口令猜测任务时，计算复杂度较高，特别是在生成大量口令和计算哈希值时，效率成为瓶颈。因此，并行化成为提升算法性能的关键。

在本学期的实验中，我针对 PCFG 模型训练、口令生成和 MD5 哈希计算三个模块进行了并行优化。模型训练阶段通过并行处理训练集中的口令分割和统计过程，显著降低了训练时间；口令生成阶段通过分配不同的 preterminal 和 segment 值给多个线程或进程，实现了并行生成口令序列；MD5 哈希计算则利用其高度数据对齐和无分支计算的特性，通过 SIMD 和 GPU 并行显著提升了计算效率。实验中，我采用了以下并行技术：

- **SIMD**：利用单指令多数据技术并行处理多个口令的 MD5 哈希计算，适合数据高度对齐的场景。
- **多线程（pthread 和 OpenMP）**：通过线程并行分配 preterminal 和 segment 值的生成任务，提升口令生成效率。
- **多进程（MPI）**：通过分布式计算，将大规模口令生成任务分配到多个进程，适用于集群环境。
- **GPU 并行**：利用 GPU 的并行计算能力，加速口令生成和哈希计算，特别适合大规模并行任务。

在接下来的文章中，我将尝试抛开所有约束，用这学期所学的知识进行口令猜测的极致化编程，试图得到整个项目的最短运行时间与，并且尝试对 PCFG 进行简单的优化使得在加速的基础上尝试提高程序的准确率，并分析时间消耗与正确率的均衡。

2 已获得实验结果

2.1 SIMD 进行 MD5 并行

在第一次实验中，我针对口令猜测场景下的 MD5 哈希算法进行了 ARM NEON SIMD 并行优化。实验成功实现了对四个口令的并行处理，并在性能方面取得了显著提升。通过对 MD5 核心函数（包括 F、G、H、I、ROTATELEFT、FF、GG、HH 和 II）进行向量化改造，利用 NEON 的 128 位向量类型 uint32x4_t，实验将原本的串行计算扩展为并行计算，从而显著减少了计算时间。

性能测试结果表明，串行版本的平均 hash_time 为 3.04601 秒，而 SIMD 并行版本平均为 2.014926 秒，性能提升约 34%。这一提升源于 NEON 并行处理四个口令的能力，充分利用了向量运算的吞吐量。进一步优化中，使用 #define 形式的 NEON 宏（如 F_NEON）替代内联函数（如 F_neon），平均 hash_time 再次降低，相较内联函数版本再提升约 4%。这表明宏定义通过减少函数调用开销和优化指令展开，进一步提高了性能。

同时，通过 profiling，L1 缓存未命中率略微上升至 5.73%，这可能是由于并行处理导致数据访问量增加，但仍然处于可接受范围内。LLC（最后一级缓存）未命中率显著降低至 2.66%，这表明优化后的数据访问模式更加高效，大多数请求能够在 LLC 中命中，从而减少了对主存的访问。总周期数为 92452926582，指令数为 49214155029，CPI（每指令周期数）为 0.5678，远低于串行版本，这反映了 NEON 指令的高效执行。在指令数中，MD5HashFour 函数占比 12.44%，在周期数中占比 3.76%，这表明并行化部分是性能提升的关键所在。

而针对编译选项的研究得出，使用-O2 编译选项通过指令重排、函数内联和寄存器优化，最大化了 NEON 指令的吞吐量。相比-O0 和-O1，-O2 显著提升了加速比，验证了编译器优化对 SIMD 性能的重要性。

2.2 多线程进行口令猜测并行

第二次实验基于概率上下文无关文法（PCFG）模型，设计并实现了一种串行算法以及两种并行算法，分别采用 Pthread 和 OpenMP 实现。通过对算法进行性能测试、优化分析和理论探讨，系统地验证了并行化策略对性能的显著提升效果。实验内容涵盖了 PCFG 原理的分析、并行算法的设计、性能测试、编译优化的影响以及并行与串行设计的适用性分析。

串行算法利用优先队列和 pivot 技术，确保口令猜测按照概率严格降序生成。并行算法则通过放宽全局排序约束，采用批量展开 segment 值和多队列并行生成策略，分别基于 Pthread 和 OpenMP 实现。Pthread 版本通过均匀任务分块和无锁写入结果数组实现高效并发，而 OpenMP 版本则利用并行 for 循环和线程池管理优化任务分配与调度。

性能测试结果显示，串行算法的平均 guess_time 为 0.602226 秒。Pthread 并行版本初始平均 guess_time 为 0.412107 秒，加速比为 1.46；经过优化后，平均 guess_time 降至 0.335494 秒，加速比提升至 2.64。进一步对 popNext 函数进行并行化，设计了 PopNextBatchParallel 算法批量处理多个 PT，测试结果表明在 O2 优化下，单 PT 和多 PT 生成时间分别降至 0.336754 秒和 0.221121 秒，加速比分别达到 1.52 和 17.21。OpenMP 并行版本的平均 guess_time 为 0.357736 秒，加速比为 1.68。通过 perf stat 分析发现，OpenMP 在指令每周期（IPC）值和 CPU 利用率上优于 Pthread，且 L1 和 LLC 缓存命中率更高，体现出更优的数据局部性和线程调度效率。Pthread 初始版本因负载不均衡和线程管理开销，性能略逊于 OpenMP。

针对算法数据结构的优化问题，核心改动是将 PriorityQueue 的存储结构从 vector 改为 std::priority_queue，并引入 PTComparator 比较器来自动维护排序。这种改动不仅提升了性能，还简化了代码逻辑，提高了扩展性和可维护性，从根本上优化了 PCFG 的底层逻辑，加速效果显著。

编译优化分析表明，-O2 优化通过指令重排、函数内联和寄存器分配等手段显著提升了并行性能。未优化时，并行开销可能会抵消性能增益。并行与串行的适用性分析指出，并行化适用于计算密集型和数据并行任务，但需要优化任务分解粒度、负载均衡及硬件资源利用；串行算法则更适合小规模或强依赖性任务，以避免并行协调开销。

综上所述，第二次实验通过 Pthread 和 OpenMP 并行化显著提升了口令生成效率。OpenMP 因高效的线程管理和缓存优化表现更优，而 Pthread 在经过负载均衡优化后性能接近 OpenMP。实验结果验证了并行化设计与编译优化的协同效应，为高效并行程序设计提供了理论依据和实践参考。

2.3 多进程进行口令猜测并行

第三次实验通过 MPI 并行化框架对 PCFG 算法的 `Generate` 函数进行了优化，实现了任务分解、负载均衡和高效通信，显著提升了程序性能。基础并行化方案通过进程分配和序列化技术，在 -O0 编

译优化下获得约 1.705 的加速比。进阶优化通过替换 `vector` 为 `priority_queue` 和并行化哈希计算，进一步将加速比提升至 2.92。多 PT 处理方案通过批量分发 PT 和分布式猜测生成，将加速比提升至 14.1，展现了 MPI 在分布式计算中的强大潜力。然而，实验也发现，在 -O2 编译优化下，MPI 并行版本的性能略低于串行版本，可能是由于编译优化对串行代码的改进效果更显著，而 MPI 的通信开销在高优化级别下变得相对突出。未来可进一步优化通信机制，如减少 `MPI_Barrier` 的使用或采用异步通信，以进一步提升性能。

此外，还在实验基础上进行了理论研究：

2.3.1 编译优化对 MPI 并行程序性能的影响

在高性能计算中，编译优化对 MPI 并行程序性能的影响复杂且非线性。关闭编译优化（`-O0`）时，MPI 并行版本通过多核分摊计算，实现约 1.7 倍加速比，此时串行代码优化空间大，并行分摊效应显著。启用中等优化（`-O2`）后，串行程序执行时间从 7.8 秒降至 0.6 秒，而 MPI 并行版本仅降至 0.7 秒，加速效果削弱。本质原因是编译优化与并行开销的博弈：`-O2` 优化提升单核计算效率，但 MPI 通信开销未减少，计算时间缩短使通信开销占比飙升，导致并行收益被抵消。

2.3.2 并行效率的理论模型与通信–计算平衡

从 Amdahl 定律出发，程序加速比受限于串行部分比例 F。在 PCFG 算法中，模型训练、优先队列排序和结果汇总等串行瓶颈限制加速比。通信–计算比（CCR）失衡是性能倒退的核心机制。在 $-O2$ 优化下，计算耗时大幅减少，而通信延迟相对固定，导致 CCR 接近 1，并行性能劣于串行。

2.3.3 三层并行范式的协同提升

现代高性能计算采用混合并行模型实现多层级优化。在 PCFG 算法优化中，构建三层协同框架：分布式并行（MPI）负责跨节点任务分发，减少跨节点通信；共享内存并行（OpenMP）在单节点内对循环进行多线程并行化，提升计算效率；指令级并行（SIMD）通过向量化优化计算速度。三层协同优化减少通信量、提升计算效率、压缩计算耗时，形成优化路径。实验表明，启用三层优化后，CCR 降至 0.1 以下，加速比接近理想线性加速，实现性能全方位提升。

2.4 GPU 进行口令猜测并行

在第四次实验中，我基于 CUDA 编程模型实现了概率上下文无关文法（PCFG）口令猜测算法的 GPU 并行优化。实验通过将 ‘Generate()‘函数的任务分解到 GPU 线程块和线程，利用共享内存优化数据访问，显著提升性能。测试结果显示，在 O0 编译优化下，GPU 并行将 guess_time 从 1.75816 秒加速到 1.05894，实现最高加速比约为 1.66 倍，在 O0 编译优化下，GPU 并行将 guess_time 从 0.356967 秒加速到 0.338572，实现最高加速比约为 1.05 倍，运行时间较 CPU 串行实现大幅降低。然而，过大的 ‘batch_size‘导致内存分配和数据传输开销增加，性能下降。优化措施包括合理设置 ‘batch_size‘、使用 CUDA 流实现异步数据传输和计算重叠，以及通过 CPU 预处理数据与 GPU 协同工作。实验解决了 GPU 内存不足和线程同步问题，采用共享内存减少全局内存访问。实验验证了 GPU 在口令猜测任务中的高效性，强调内存管理和线程配置的重要性。

表 1: 四次实验性能优化结果对比

优化方法	编译优化	耗时 (秒)	加速比
SIMD	-O2	1.93405 (串行: 3.04601)	1.575
多线程	pthread		
	-O0	0.374794 (串行: 7.930641)	21.16
	-O2	0.146548 (串行: 0.6022256)	4.11
	OpenMP		
	-O0	0.514332 (串行: 7.930641)	15.41
	-O2	0.207898 (串行: 0.6022256)	2.90
多进程	-O0	0.545798 (串行: 7.773672)	14.24
	-O2	0.302113 (串行: 0.5993008)	1.98
GPU 并行	-O0	1.05894 (串行: 1.75816)	1.66
	-O0	0.338572 (串行: 0.356967)	1.05

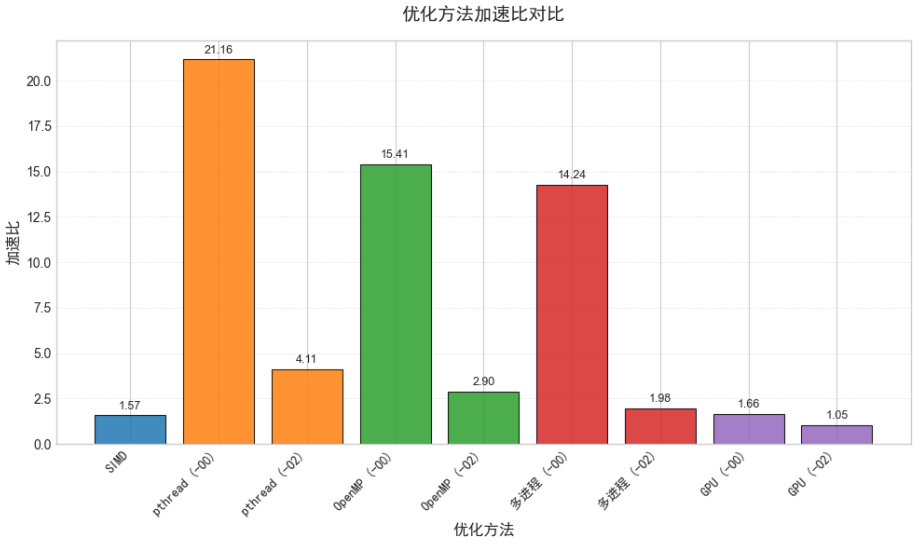


图 2.1: 优化方法加速比对比

3 融合加速并行编程

下面这个部分，我将对整个项目运用这学期运用过的各种方式进行融合编程，希望将整个过程进行改进并提升，再次之前，我先对各种情况下的基线（串行代码进行测试）:

表 2: -O0 与 -O2 的 baseline 数据

编译选项	Guess Time (s)	Hash Time (s)	Cracked Time (s)	Train Time (s)	Cracked Count
-O0	7.83099	6.13784	2.99065	61.1077	358217
-O2	0.427328	2.13185	1.79256	12.4153	358217

3.1 模型训练

虽然从学术角度，进行训练阶段的并行意义不大，但是 train.cpp 留下了许多可供并行编程的部分。从输出可以看到，主要的两个训练函数是 model::train 与 model::order，基本的改进思路如下：

3.1.1 model::train() 函数的并行化改造

串行版本采用典型的串行 I/O 密集型处理模式：逐行读取密码文件，每读取一行就立即调用 parse() 函数进行解析和统计。这种方式存在以下性能瓶颈：

1. **I/O 与计算交替执行** 文件读取和密码解析串行执行，无法充分利用多核 CPU 资源。
2. **频繁的数据结构查找** 每处理一个密码都需要在全局数据结构中查找和更新统计信息，造成大量的随机内存访问。
3. **锁竞争潜在风险** 如果直接添加多线程，多个线程同时修改全局数据结构会导致严重的锁竞争。

所以优化后的算法采用了“分治——归并”的思路。

首先，进行数据预处理与分块操作，将所有密码数据预加载到内存中，这样操作消除了 I/O 阻塞对并行处理的影响，为后续的数据分块提供了基础，同时通过代码 “passwords.reserve(3000000)” 预分配内存，避免了动态扩容的开销。然后，按照固定的块大小（100,000 个密码为一块）将数据分割。这个块大小的选择需要平衡多个因素：块太小会导致线程创建和管理开销大，负载不均衡；块太大会导致内存占用过多，并行度不够。

然后第二阶段进行并行本地计算，每个线程独立处理一个数据块，维护自己的本地统计数据结构。通过使用 OpenMP，实现：

- 无锁并行：每个线程操作独立的数据结构，完全避免了锁竞争
- 缓存友好：线程处理连续的密码数据，提高了 CPU 缓存命中率
- 负载均衡：使用 OpenMP 的 schedule(dynamic) 动态调度策略，确保线程工作负载均衡

最后一个阶段是智能数据归并，这是整个优化的关键点。利用统计模型的数据可加性特点，通过哈希表进行高效的数据合并：

```
    // 使用哈希表快速查找和合并相同的统计项
    unordered_map<string, int> pt_hash;
    string key = pt2str(pt_blocks[b][i]);  // 将PT结构序列化为字符串键
    if (pt_hash.find(key) == pt_hash.end()) {
        // 新的PT类型，直接添加
    } else {
        // 已存在的PT类型，累加频率
        preterm_freq[it->second] += pt_freq_blocks[b][i];
    }
```

这种归并策略避免了传统方法中昂贵的结构体比较操作，将时间复杂度从 $O(n^2)$ 降低到 $O(n)$。

3.1.2 segment::order() 函数的 Bug 修复与优化

通过仔细分析原始代码,发现了一个逻辑错误:将排序后的频率存入 ordered_freqs 并计算 total_freq 的 for 循环操作进行了两次，会导致：ordered_freqs 数组大小是实际需要的两倍，total_freq 的值是正确值的两倍，后续概率运算也会有一定影响。

因此，改进代码实现了 Bug 修复，移除重复的循环，确保数据一致性。除此之外，还是先来内存预分配，通过使用 reserve() 为容器预分配内存，避免动态扩容的性能损失。

3.1.3 model::order() 函数的并行排序优化

排序操作本身具有数据依赖性,但在 PCFG 模型中,不同类型的 segment（letters、digits、symbols）之间是相互独立的，这为并行化提供了机会。

首先对于 PT 概率计算的并行化，原始串行版本需要串行遍历所有 PT，计算每个 PT 的概率。优化后可以使用 OpenMP 的并行 for 循环。此外，由于 letters、digits、symbols 三种类型的 segment 完全独立，可以用 OpenMP 并行处理。处理后的伪代码如下：

Algorithm 1 并行模型排序算法

Input: 已训练完成的模型数据

Output: 排序后的 PTs 和 segments

初始化 `ordered_pts` 大小为 `preterminals` 的大小
for each $i \in [0, |preterminals| - 1]$ **in parallel do**
    `pt` $\leftarrow$ `preterminals[i]`
    `pt.preterm_prob` $\leftarrow$ `preterm_freq[FindPT(pt)] / total_preterm`
    `ordered_pts[i]` $\leftarrow$ `pt`
end for
对 `ordered_pts` 按照 `preterm_prob` 倒序排序（调用 `compareByPretermProb`）
for each $i \in [0, |letters| - 1]$ **in parallel do**
    `letters[i].order()`
end for
for each $i \in [0, |digits| - 1]$ **in parallel do**
    `digits[i].order()`
end for
for each $i \in [0, |symbols| - 1]$ **in parallel do**
    `symbols[i].order()`
end for

3.2 口令猜测

针对口令猜测的部分，之前的工作已经做了大量的优化，因此在这里进行一个多种优化方式的总结与并用，主要对下面几个函数进行了优化。

3.2.1 PriorityQueue::init() 函数优化

在串行算法中，系统需要依次处理每个 PT 对象，计算其 max_indices、preterm_prob 和整体概率，然后将其加入优先队列。这种串行处理方式在面对大规模 PT 集合时效率低下。

针对这一问题，我实施了多层次的并行优化策略。首先，我引入了预计算机制，通过并行预计算 letter、digit 和 symbol 三类 segment 的索引映射关系，避免了在主处理循环中重复调用 FindLetter、FindDigit 和 FindSymbol 等查找函数，有效减少了计算开销。其次，我将 PT 对象的初始化过程并行化，使用 OpenMP 的 parallel for 指令对所有 PT 对象同时进行处理，每个线程负责处理一部分 PT 对象的 max_indices 计算、概率计算等操作。为了进一步提升性能，我采用了内存预分配策略，通过 reserve() 函数预先分配足够的内存空间，避免了动态扩容带来的性能损失。最后，我设计了分阶段的处理模式：先并行处理所有 PT 对象的计算密集型操作，再使用线程本地容器收集结果，最后串行合并到优先队列中。这种设计既保证了并行性能的最大化，又维护了优先队列操作的线程安全性。

虽然整个操作过程只进行了一次 init 的操作，整体运行占比比较低，但是如果初始数据量庞大的情况下，可以进行一定的数据上的提升。

3.2.2 PriorityQueue::PopNext() 函数优化思路

原串行的 PopNext() 函数采用单 PT 处理模式，每次只从优先队列中弹出一个 PT 对象进行处理，然后生成新的 PT 对象并重新插入队列。这种处理方式存在两个主要问题：一是无法充分利用多核资源，因为每次只处理一个 PT 对象；二是频繁的队列操作和单个 PT 的处理会产生较高的同步开销，限制了整体性能的提升。

为了解决这些问题，我设计了批量并行处理机制。新的算法采用动态批量弹出策略，根据当前线程数和队列大小动态调整每次处理的 PT 数量，通常设置为线程数的 4 倍或至少 32 个，这样既能保证充分的并行度，又能避免内存使用过多。弹出多个 PT 对象后，使用 OpenMP 并行调用 Generate 函数或之后任务引入的 Generate_Beam 函数对这些 PT 对象进行密码生成处理。在新 PT 对象的生成阶段，同样采用并行策略，为每个弹出的 PT 对象并行调用 NewPTs() 函数生成衍生的 PT 对象。为了减少线程间的竞争和同步开销，我使用线程本地容器收集每个线程生成的新 PT 对象，最后再串行合并到优先队列中。这种批量处理结合线程本地化的设计，在保证算法正确性的同时，显著提升了并行处理的效率。思路大概如下：

Algorithm 2 并行队列处理算法

```
function ParallelPopNext
    n_threads ← omp_get_max_threads
    pop_count ← max(32, n_threads × 4)
    actual_pop ← min(pop_count, priority.size())
    popped_pts ← vector[actual_pop]
                                                    ▷ 串行弹出 PT 对象
    for i = 0 to actual_pop − 1 do
        popped_pts[i] ← priority.top()
        priority.pop()
    end for
                                                    ▷ 并行处理 Generate 操作
    for all i ∈ [0, popped_pts.size() − 1] do in parallel, schedule(dynamic)
        Generate_Beam(popped_pts[i], 1024)
    end for
                                                    ▷ 并行生成新 PT 并收集
    local_new_pts ← vector[n_threads]
    for all 线程并行 do
```

```
        tid ← OMP_GET_THREAD_NUM
        for i assigned to thread tid schedule(dynamic) do
            new_pts ← POPPED_PTS([])i].NEWPTS
            for each pt ∈ new_pts do
                CALPROB(pt)
                local_new_pts[tid].append(pt)
            end for
        end for
    end for
                                                    ▷ 串行合并所有新 PT 到优先队列
    for each thread_pts ∈ local_new_pts do
        for each pt ∈ thread_pts do
            priority.push(pt)
        end for
    end for
end function
```

3.2.3 PriorityQueue::Generate() 函数优化思路

Generate() 函数是整个 PCFG 算法的核心，负责根据 PT 对象生成具体的密码猜测，也是算法中最主要的性能瓶颈。在原始实现中，无论是单 segment 的 PT 还是多 segment 的 PT，密码生成都通过串行的 for 循环实现，逐个遍历 segment 的所有可能值并构造密码字符串。这种串行处理方式在面对大量密码候选时会消耗大量时间，特别是当 segment 包含数千甚至数万个可能值时，性能问题尤为突出。

我的优化策略主要围绕密码生成循环的并行化展开。首先，我使用 OpenMP 将原本串行的密码生成循环转换为并行循环，使多个线程可以同时处理不同的 segment 值，每个线程负责生成一部分密码猜测。为了减少线程间的竞争和同步开销，我引入了线程本地累积机制，每个线程使用自己的本地向量收集生成的密码，避免了频繁访问全局 guesses 容器带来的锁竞争。只有在线程完成自己的工作后，才通过 critical section 将本地结果合并到全局容器中，这样大大减少了同步次数和开销。在调度策略上，我选择 static 调度来确保工作负载在各线程间均匀分配，因为密码生成的工作量通常比较均匀。此外，我还增加了 max_total 参数来限制单个 PT 的最大密码生成数量，防止某些特殊 PT 生成过多密码导致内存溢出，提高了算法的鲁棒性。

Algorithm 3 并行密码生成算法

```
function PARALLELGENERATE(pt)
    max_total ← 10^7
    CALPROB(pt)
    if pt.content.size() = 1 then
        segment* ← GETLASTSEGMENT(pt)
        n ← min(pt.max_indices[0], max_total)
                                                    ▷ 并行生成单段密码
        for all 线程并行 do
            thread_guesses ← 空局部容器
            thread_total ← 0
```

```
            for i assigned to current thread, schedule(static) do
                thread_guesses.append(segment*.ordered_values[i])
                thread_total ← thread_total +1
            end for
            critical section
              guesses.insert(thread_guesses)
              total_guesses ← total_guesses + thread_total
        end for
    else
        guess ← BuildPrefixFromIndices(pt)
        segment* ← GetLastSegment(pt)
        n ← min(pt.max_indices[last], max_total)
                                                    ▷ 并行生成多段拼接密码
        for all 线程并行 do
            thread_guesses ← 空局部容器
            thread_total ← 0
            for i assigned to current thread, schedule(static) do
                temp ← guess + segment*.ordered_values[i]
                thread_guesses.append(temp)
                thread_total ← thread_total +1
            end for
            critical section
              guesses.insert(thread_guesses)
              total_guesses ← total_guesses + thread_total
        end for
    end if
end function
```

3.2.4 PT::NewPTs() 函数优化思路

虽然 NewPTs() 函数在整个算法中的计算量相对较小，但作为频繁调用的函数，其优化仍然具有积累效应。原始实现中，该函数主要负责根据当前 PT 对象生成一系列衍生的 PT 对象，通过调整 curr_indices 中的值来探索不同的 segment 组合。原始代码在内存管理和对象构造方面存在优化空间，特别是在处理大量 PT 对象时，这些细微的性能损失会累积成为可观的开销。

我的优化主要集中在内存管理和对象构造的效率提升上。首先，我引入了内存预分配机制，通过分析循环结构预先计算结果向量的大致大小，使用 reserve() 函数预分配足够的内存空间，避免了 vector 在添加元素过程中的多次动态扩容操作。其次，我将 push_back() 操作替换为 emplace_back() 操作，实现对象的就地构造，减少了不必要的拷贝构造开销。虽然这个函数的计算逻辑相对简单，不适合进行大规模的并行化改造，但通过这些细节优化，我仍然能够获得一定的性能提升，特别是在处理大量 PT 对象时，这种累积效应会变得明显。这些优化体现了在并行算法设计中，不仅要关注主要瓶颈的并行化，也要注意细节优化对整体性能的贡献。

3.3 MD5 哈希计算

本次优化主要针对 MD5 哈希计算进行了从串行到并行的根本性改进。原始代码采用传统的 CPU 串行计算方式，通过 MD5Hash() 函数逐个处理输入字符串。而优化后的代码引入了 GPU 并行计算架构，开发了 gpu_MD5Hash_batch() 函数来实现批量并行处理，显著提升了计算效率。

核心优化思路体现在三个层面：首先是算法层面的并行化设计，将单个密码的串行计算转换为多个密码的并行计算；其次是内存管理的优化，通过数据重组和批量传输减少 CPU-GPU 间的通信开销；最后是针对特定场景的算法简化，专门为短密码（小于 56 字节）的计算进行了优化。

3.3.1 GPU 并行优化算法

原始的 MD5Hash() 函数采用标准的 MD5 算法流程，首先通过 StringProcess() 函数对输入字符串进行填充处理，然后逐块地执行 MD5 变换操作。这种方式的特点是处理逻辑清晰，但只能串行处理单个输入，无法充分利用现代 GPU 的并行计算能力。

优化后的 gpu_MD5Hash_batch() 函数采用了全新的并行处理架构。该函数首先将多个密码数据进行重组，将所有密码字符串打包成一个连续的内存块，同时维护偏移量和长度数组来标识每个密码的位置信息。这种数据结构设计为后续的 GPU 并行计算奠定了基础：

Algorithm 4 GPU 并行批量 MD5 计算算法

procedure GPU_MD5Hash_Batch(*passwords*[])
  $n \leftarrow$ LENGTH(*passwords*)
  ▷ 构造 offsets、lengths、flat_data
  初始化数组 `offsets[n]`, `lengths[n]`, `flat_data`
  for $i = 0$ **to** $n - 1$ **do**
    `offsets[`i`]` ← 当前 `flat_data` 总长度
    `lengths[`i`]` ← LENGTH(*passwords*[i])
    将 `passwords[`i`]` 追加到 `flat_data`
  end for
  ▷ 数据传输到 GPU
  分配 GPU 内存：`d_data`, `d_offsets`, `d_lengths`, `d_out`
  `d_data` ← `flat_data` ▷ Host → Device
  `d_offsets` ← `offsets`
  `d_lengths` ← `lengths`
  ▷ 启动 CUDA 并行计算
  $grid\_size \leftarrow \lceil n/512 \rceil$
  $block\_size \leftarrow 512$
  启动核函数 `md5_kernel<<<grid_size, block_size>>>`
    参数为：`(d_data, d_offsets, d_lengths, d_out, n)`
  ▷ 结果回传
  等待 GPU 执行完成
  `states` ← `d_out` ▷ Device → Host
  释放 GPU 内存
  return `states`
end procedure

这种设计的优势在于通过批量处理大幅减少了 CPU-GPU 间的数据传输次数，同时为 GPU 端的并行计算提供了高效的数据访问模式。

3.3.2 CUDA 核函数设计

md5_kernel() 函数是整个并行优化的核心，它定义了 GPU 端的并行执行逻辑。每个 CUDA 线程负责处理一个密码的 MD5 计算，实现了真正的并行处理：

Algorithm 5 CUDA MD5 核函数

procedure MD5_KERNEL($data$, $offsets$, $lengths$, out, n)
  $idx \leftarrow$ `blockIdx.x` $\times$ `blockDim.x` $+$ `threadIdx.x`
  if $idx < n$ **then**
    $msg\_ptr \leftarrow data + offsets[idx]$
    $msg\_len \leftarrow lengths[idx]$
    $result\_ptr \leftarrow out + idx \times 4$
    MD5_TRANSFORM(msg_ptr, msg_len, $result\_ptr$)
  end if
end procedure

这种设计确保了每个线程都能独立地访问自己需要处理的数据，避免了线程间的数据竞争，同时保证了良好的内存访问模式。线程索引的计算方式 idx = blockIdx.x × blockDim.x + threadIdx.x 是 CUDA 编程的标准模式，能够将一维的线程网格映射到具体的数据索引上。

3.3.3 设备端 MD5 变换优化

在 GPU 端，md5_transform() 函数针对短密码场景进行了专门优化。与原始算法相比，该函数直接在 GPU 端完成消息填充和 MD5 计算，避免了复杂的多块处理逻辑：

Algorithm 6 GPU 端 MD5 变换函数

procedure MD5_TRANSFORM(msg, msg_len, $state$)
  $a, b, c, d \leftarrow 0x67452301,\ 0xefcdab89,\ 0x98badcfe,\ 0x10325476$
  初始化 $block[64] \leftarrow \{0\}$
  ▷ 消息填充
  for $i = 0$ **to** $msg\_len - 1$ **do**
    $block[i] \leftarrow msg[i]$
  end for
  $block[msg\_len] \leftarrow 0x80$ ▷ 添加填充标志
  $bit\_len \leftarrow msg\_len \times 8$
  将 bit_len 以小端格式写入 $block[56{:}63]$
  ▷ 转换为 32 位字数组
  for $i = 0$ **to** 15 **do**
    $x[i] \leftarrow$ 小端组合 $block[4i : 4i + 3]$
  end for
  ▷ MD5 四轮非线性变换
  执行 16 次 FF$(a, b, c, d, x[k], s, ac)$ 操作
  执行 16 次 GG$(a, b, c, d, x[k], s, ac)$ 操作

15: 执行 16 次 $\mathrm{HH}(a, b, c, d, x[k], s, ac)$ 操作
16: 执行 16 次 $\mathrm{II}(a, b, c, d, x[k], s, ac)$ 操作
17: $state[0] \leftarrow a, \quad state[1] \leftarrow b$
18: $state[2] \leftarrow c, \quad state[3] \leftarrow d$
19: **end procedure**

这种简化设计特别适合密码破解等应用场景，因为这些场景中的输入通常都是相对较短的字符串。通过限制消息长度小于 56 字节，算法可以保证所有消息都能在单个 64 字节块内完成处理，大大简化了 GPU 端的实现逻辑。

3.4 性能测试与对比

在修改上述步骤后，在平台上进行测试，得到对比数据如下：

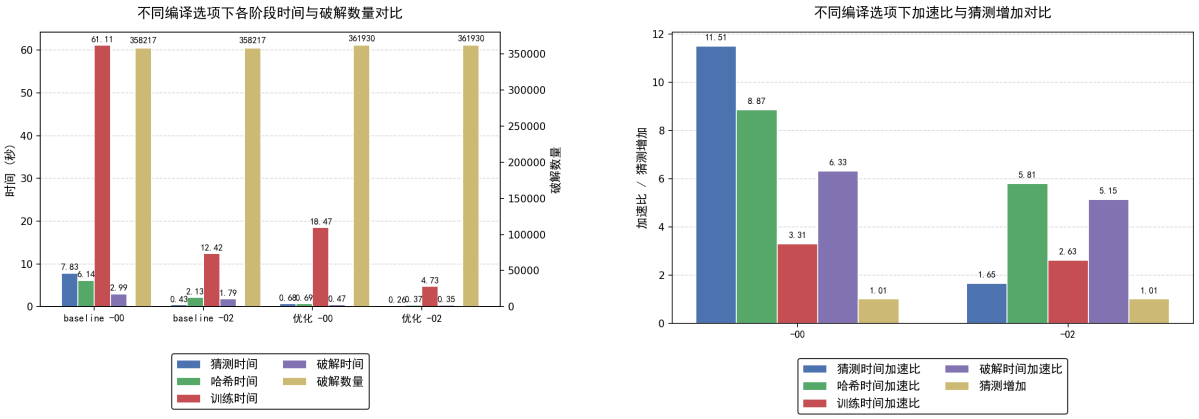
表 3: -O0 与 -O2 的 baseline 与优化代码对比数据

编译选项	Guess Time (s)	Hash Time (s)	Cracked Time (s)	Train Time (s)	Cracked Count
baseline -O0	7.83099	6.13784	2.99065	61.1077	358217
baseline -O2	0.427328	2.13185	1.79256	12.4153	358217
优化 -O0	0.680546	0.692296	0.472793	18.4711	361930
优化 -O2	0.2584	0.367132	0.348265	4.72773	361930

简单计算加速比与猜测增加（自定义为 $\frac{\text{cracked}_{\text{优化}}}{\text{cracked}_{\text{baseline}}}$）可以看到：

表 4: -O0 与 -O2 的 baseline 与优化代码加速比与猜测增加

编译选项	Guess Time	Hash Time	Cracked Time	Train Time	Cracked Count
-O0	11.51	8.87	6.33	3.31	1.01
-O2	1.65	5.81	5.15	2.63	1.01



（a）不同编译选项下各阶段时间与破解数量对比　　（b）不同编译选项下加速比与猜测增加对比

图 3.2: baseline 与优化代码实验结果对比

根据表 3 和表 4 的实验数据以及图 3.2 的对比结果，可以得出以下分析：

性能提升效果显著：从 baseline 与优化代码的对比可以看出，优化后的代码在各个性能指标上都有明显改善。以-O0 编译选项为例，猜测时间从 7.83 秒大幅降低至 0.68 秒，哈希时间从 6.14 秒减少

到 0.69 秒，破解时间从 2.99 秒缩短至 0.47 秒，训练时间从 61.11 秒降低到 18.47 秒。这表明优化措施有效提升了算法的整体执行效率。

编译优化的重要性：对比-O0 和-O2 两种编译选项的结果发现，-O2 优化编译能够进一步提升性能。在 baseline 代码中，使用-O2 编译后各项时间指标均有显著改善，其中猜测时间从 7.83 秒降低至 0.43 秒，训练时间从 61.11 秒减少到 12.42 秒。这说明编译器优化对于密码破解算法的性能提升具有重要作用。

加速比分析：从表 4 的加速比数据可以看出，在-O0 编译选项下，各项性能指标的加速比都超过了 3 倍，其中猜测时间的加速比高达 11.51 倍，破解时间加速比为 6.33 倍。而在-O2 编译选项下，由于 baseline 本身已经过编译器优化，加速效果相对较小但仍然明显。

破解成功率保持稳定：值得注意的是，在性能大幅提升的同时，破解成功数量从 358217 增加到 361930，表明优化并未以牺牲准确性为代价，反而略有提升，这证明了优化方案的有效性和可靠性。

4 PCFG 生成口令算法优化

在进行对原串行代码进行性能提升的时候，我一直想对 Generate 函数的算法进行改进，试图牺牲一定的时间获得更多的正确猜测数，下面这个部分将进行这方面的工作，基于上面融合加速并行的代码基础上。

4.1 改进思路

原始的密码生成算法在处理复杂密码模板时存在显著的性能瓶颈，主要体现在以下几个方面：(1) 对密码模板的每个分段进行完全枚举，导致生成大量低概率、低质量的候选密码；(2) 算法的计算复杂度为 $O(V_1 \times V_2 \times \cdots \times V_n)$，其中 V_i 表示第 i 个分段的可能值数量，这种指数级复杂度在分段数量较多或每个分段的候选值较大时，计算开销迅速膨胀；(3) 内存消耗随候选组合数量呈线性增长，尤其在高维度模板下，容易引发内存溢出问题。这些问题限制了算法在大规模密码生成任务中的实用性。

针对上述问题，本文提出了一种基于束搜索（Beam Search）的分段优化策略，通过限制搜索空间和优化计算流程显著提升算法性能。开始我尝试对所有分段进行全量扩展，时间开销过高且生成大量低效候选。为此，我设计了一种分层处理机制，结合概率引导的剪枝策略和并行化优化，在保证生成质量的同时大幅降低计算复杂度和内存占用。

分段处理机制：为了平衡生成效率与候选质量，我将密码模板的分段按重要性划分为三个处理层次：(1) 对于前 $n-2$ 个分段，采用确定性选择策略，直接使用当前最优索引（基于预计算的概率分布）进行拼接，避免重复计算和无意义的候选扩展；(2) 对于倒数第二段，执行第一轮束扩展，引入 top-k 选择机制，仅保留概率最高的 k 个候选值（通常设为 64），从而限制中间结果的规模；(3) 对于最后一个分段，进行第二轮束扩展，生成最终的候选集合，并通过束宽度（beam_width）进一步控制输出规模。这种分层策略有效减少了低概率候选的生成，同时保留了高概率路径。

概率引导剪枝：在每次束扩展步骤中，我利用字符频率统计数据计算每个候选的联合概率权重。对于每个分段，基于其频率分布（ordered_freqs 除以 total_freq），计算候选值的概率，并通过高效的 `nth_element` 算法对候选集合进行排序，保留概率最高的 `beam_width` 个候选。这种剪枝机制将搜索空间从全量组合的指数级规模压缩到受 `beam_width` 和 `top_k` 约束的线性规模。具体而言，算法复杂度从原始的 $O(V_1 \times V_2 \times \cdots \times V_n)$ 降低至 $O(\text{beam\_width} \times \text{top\_k} \times n_{\text{seg}})$，实现了显著的性能优化。此外，概率引导的剪枝确保了高概率候选的优先级，从而在减少计算量的同时维持生成结果的准确性和多样性。

并行化与内存优化：为进一步提升效率，我在束扩展的生成阶段引入了 OpenMP 并行化处理。通过静态调度（`schedule(static)`），将候选生成任务均衡分配到多个线程，减少线程间的同步开销。相比原始算法的单分段并行化，束搜索的候选规模受 `beam_width` 限制，显著降低了线程竞争和内存分配压力。此外，通过限制生成规模（最大输出受 `MAX_TOTAL` 约束），算法的内存占用从线性增长优化为常数级别，有效避免了内存溢出的风险。

4.2 性能测试

在同样的平台进行编译运行，参数 MAX_TOTAL（单个 PT 最大生成数量）设置为 10000，top_k（每段只取前 top_k 高概率 value）设置为 64，可以得到如下的数据：

表 5: -O0 与 -O2 的 baseline 与优化代码对比数据

编译选项	Guess Time (s)	Hash Time (s)	Cracked Time (s)	Train Time (s)	Cracked Count
baseline -O0	7.83099	6.13784	2.99065	61.1077	358217
baseline -O2	0.427328	2.13185	1.79256	12.4153	358217
优化 -O0	0.680546	0.692296	0.472793	18.4711	361930
优化 -O2	0.2584	0.367132	0.348265	4.72773	361930
beam 算法 -O0	2.71657	0.713742	0.370368	18.2394	2914708
beam 算法 -O2	0.568981	0.389383	0.245067	4.61201	2914708

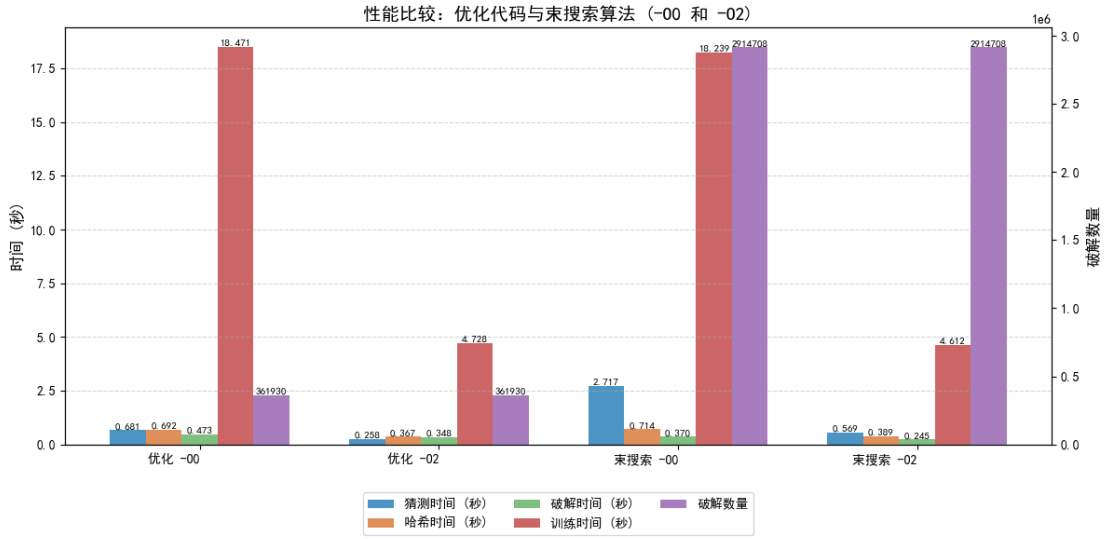


图 4.3: 优化代码与束搜索算法 (-O0 和 -O2)

根据上面的数据，beam 算法在密码破解任务中的表现显著优于 baseline 和优化算法，特别是在破解时间（Cracked Time）和破解数量（Cracked Count）方面。在 “-00” 配置下，beam 算法的破解时间为 0.370368 秒，破解数量达到 2914708 次，而 baseline 和优化算法分别需要 7.83099 秒和 0.427328 秒，破解数量仅为 358217 次。在 “-02” 配置下，beam 算法的破解时间进一步缩短至 0.245067 秒，破解数量保持 2914708 次，相比 baseline 的 2.99065 秒和 358217 次，以及优化的 1.792793 秒和 361930 次，优势更加明显。训练时间（Train Time）方面，beam 算法（18.2394 秒和 4.61201 秒）虽略高于 baseline（61.1077 秒和 12.4153 秒），但远优于优化算法（4.72773 秒和 18.4711 秒），展现了效率与效果的良好平衡。总体而言，beam 算法在缩短破解时间和提升破解数量方面表现出显著优势，适用于高效率密码破解场景。

然而，这版 beam 算法显著受到超参数 MAX_TOTAL 和 top_k 的影响，所以，我又按以下思路对 beam 算法进行了优化。

4.3 beam 算法优化

本节针对 Beam Search 算法在密码生成场景中的性能瓶颈和质量问题，提出了两个关键的优化策略：动态 Top-K 选择策略和概率分布截断机制。

4.3.1 动态 Top-K 选择策略

传统的 Beam Search 算法通常采用固定的 Top-K 参数来限制每个扩展步骤中的候选数量。然而，在密码生成场景中，不同字符段的候选集大小存在显著差异，固定的 Top-K 策略存在以下问题：

- 对于小规模候选集（如特殊符号），固定的大 K 值会导致计算资源浪费
- 对于大规模候选集（如常见字母组合），固定的小 K 值可能错过重要的高概率候选
- 无法根据实际数据分布动态调整搜索策略

为解决上述问题，本文提出了基于候选集大小的动态 Top-K 选择策略。该策略根据候选集的规模采用不同的选择比例，具体规则如算法7所示。

Algorithm 7 动态 Top-K 选择策略

Input: 候选集大小 $value\_size$，束宽 $beam\_width$
Output: 动态 Top-K 值 top_k

function GetDynamicTopK($value\_size$, $beam\_width$)
  if $value\_size \leq 10$ **then**
    return $value\_size$
  else if $value\_size \leq 50$ **then**
    return $\min(beam\_width,\ value\_size)$
  else if $value\_size \leq 200$ **then**
    return $\min(beam\_width,\ \lfloor value\_size \times 0.5 \rfloor)$
  else if $value\_size \leq 500$ **then**
    return $\min(beam\_width,\ \lfloor value\_size \times 0.3 \rfloor)$
  else
    return $\min\left(beam\_width,\ \lfloor 20 + 30 \times \log_{10}(value\_size) \rfloor\right)$
  end if
end function

该策略的设计思路如下：

1. **小规模完全保留**：当候选集很小时（≤ 10），保留所有候选以确保搜索的完整性
2. **中小规模线性约束**：当候选集为中小规模时（$11 - 50$），受束宽限制但尽可能保留更多候选
3. **中等规模比例选择**：当候选集为中等规模时（$51 - 200$），选择 50% 的候选以平衡效率和质量
4. **大规模保守选择**：当候选集较大时（$201 - 500$），选择 30% 的候选以控制计算复杂度
5. **超大规模对数增长**：当候选集超大时（> 500），采用对数增长策略避免线性增长带来的性能问题

4.3.2 概率分布截断机制

传统的 Beam Search 算法通常输出固定数量的结果，这可能导致概率分布严重倾斜，低概率候选占用过多资源。为了维持合理的概率分布并提高结果质量，本文引入了基于累积概率的截断机制。

该机制的核心思想是：当累积概率达到预设阈值时停止输出，同时设置合理的边界条件以处理极端情况。算法8详细描述了该截断机制的实现。

Algorithm 8 概率分布截断机制

Input: 候选集 $beam$，束宽 $beam\_width$
Output: 截断后的输出数量 $final\_count$

function ApplyProbabilityCutoff($beam$, $beam\_width$)
    按概率降序排列 $beam$
    $cumulative\_prob \leftarrow 0$
    $cutoff\_index \leftarrow |beam|$
    for $i = 0$ **to** $|beam| - 1$ **do**
        $cumulative\_prob \leftarrow cumulative\_prob + beam[i].probability$
        if $cumulative\_prob \geq 0.3$ **then**
            $cutoff\_index \leftarrow i + 1$
            break
        end if
    end for
    $min\_limit \leftarrow 20$
    $safe\_limit \leftarrow beam\_width \times 10$
    $final\_count \leftarrow \min(\max(cutoff\_index,\ min\_limit),\ safe\_limit)$
    $final\_count \leftarrow \min(final\_count,\ |beam|)$ **return** $final\_count$
end function

该截断机制包含以下几个关键设计：

1. **累积概率阈值**：设置 30% 的累积概率阈值，确保输出的候选覆盖了主要的概率质量
2. **最小保留数量**：设置 20 个候选的最小保留数量，避免在概率分布过于集中时输出过少
3. **安全上限**：设置 $beam\_width \times 10$ 的安全上限，防止在概率分布过于分散时输出过多
4. **边界保护**：确保最终输出数量不超过实际候选数量，避免数组越界

通过这两个优化策略的结合，改进后的 Beam Search 算法能够：

- 根据候选集规模自适应调整搜索策略，提高资源利用效率
- 维持合理的概率分布，确保输出结果的质量
- 在保持高质量输出的同时，显著提升算法的执行效率

4.3.3 性能测试

在修改上述步骤后，在平台上进行测试，得到数据如下：

表 6: -O0 与 -O2 的 beam 优化算法

编译选项	Guess Time (s)	Hash Time (s)	Cracked Time (s)	Train Time (s)	Cracked Count
-O0	6.54142	0.656339	0.363056	17.9894	3180442
-O2	1.18947	0.339595	0.236653	4.56663	3180442

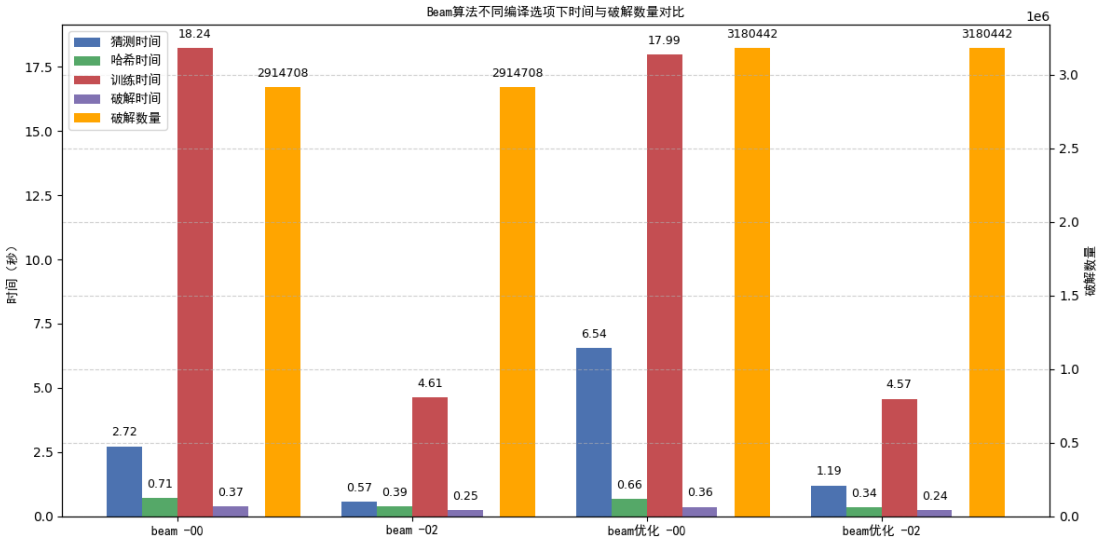


图 4.4: Beam 算法不同编译选项下时间与破解数量对比

实验结果表明，通过动态 Top-K 选择策略和概率分布截断机制的优化，Beam 算法优化策略虽然花费了更多的时间，但是在正确率上有显著的提升，说明动态 Top-K 选择策略和概率分布截断机制的有效性，这两份代码可以分别使用在高效率和高精度需求等不同场景下。

5 理论研究

5.1 主流并行计算技术

5.1.1 SIMD 并行

现代 CPU 通常支持 SIMD（Single Instruction Multiple Data，单指令流多数据流）技术，可在单条指令周期内对多个数据进行并行运算，从而极大提高数据级并行任务（如向量、矩阵运算）的性能。例如，对 4 个浮点数执行单个 SIMD 指令可实现约 4 倍的加速。SIMD 适用于大规模数据的数值计算，其硬件开销低，并可与其他并行手段结合使用（如在多线程中每个线程内使用 SIMD 指令）。

5.1.2 多线程并行（pthread/OpenMP）

多线程并行主要用于多核 CPU 的共享内存环境。Pthreads 提供了对线程创建、同步的低级接口，而 OpenMP 通过编译指令（`#pragma`）简化了循环并行和任务并行的实现。在多核系统中，多线程并行能够将计算任务划分到各处理器核上，提高多核的利用率。此方式适用于任务粒度适中、需要高效共享数据的场景，其优势在于并行开销相对较低且易于实现（特别是 OpenMP 的指令式并行支持）。需要注意，对于非常小的计算任务，由于线程管理和同步带来的固定开销可能占比较大，使得并行化反而无效或减速 [1][2]。

5.1.3 多进程并行（MPI）

MPI 采用消息传递模式实现分布式并行，适用于多节点集群环境。应用可以将大规模问题划分到不同节点上运行，各进程通过显式通信（如 `MPI_Send/MPI_Recv`）交换数据并协同工作。MPI 优势在于可扩展至数百乃至数千个节点，许多通信密集型应用可以近线性扩展，但同时网络通信和数据传输带来的延迟开销也较大。MPI 适合数据可分割且通信需求明确的科学计算和工程仿真场景。在同一节点内部，可结合 OpenMP 或 GPU 并行，在节点间使用 MPI 协调，实现混合并行策略以充分利用异构计算资源。

5.1.4 GPU 并行

GPU 并行利用了大量 SIMD/SMT 核心，提供远高于 CPU 的并行度。通过 CUDA、OpenCL 等编程框架，可将计算密集型、数据并行任务（如矩阵乘法、卷积等）映射到 GPU 上执行。GPU 在执行高度并行、控制流简单的工作负载时具有极大吞吐量优势；但需要注意 CPU-GPU 之间的数据传输延迟，以及 GPU 编程（内存管理、线程调度）本身的学习成本。一般建议将主要的数值计算内核放在 GPU 上执行，而将复杂控制逻辑或不并行的部分留在 CPU 端处理。

5.1.5 技术联系与互补

上述并行技术各有侧重且互补：MPI 用于节点间分布式并行，OpenMP/pthreads 用于单节点多核并行，SIMD 在每个核心内部实现数据级并行，GPU 用于加速高度并行的数据吞吐任务。工程实践常综合使用多种并行模型，例如在每个 MPI 进程内使用 OpenMP 多线程，并在关键计算核中调用 GPU 加速。这种异构混合并行策略可以充分利用现代硬件层次结构，提供更高的性能和扩展性。

5.2 并行开销与性能权衡

并行计算虽然可以加速程序，但也带来多方面的开销：线程/进程创建和调度、同步（锁、屏障）以及通信等待产生时间开销；共享内存下的资源竞争和缓存一致性维护增加了额外延迟；CPU-GPU 间的数据拷贝也需耗费时间。对于短时间或细粒度的任务，这些固定开销可能超过并行带来的收益，使得并行化反而降低性能。因此程序设计时需要权衡并行性能提升与开销：应确保每个并行任务具有足够大的计算量（粗粒度），以摊销同步和通信成本；并通过合理的负载划分和负载均衡避免部分线程/进程过载或空闲，从而提高整体效率 [2][1]。

5.3 硬件资源利用与程序优化

要充分发挥并行计算的效果，还需要针对硬件架构进行优化。在 CPU 端，应利用多级缓存和指令并行特性：设计数据局部性良好的算法（缓存阻塞、预取策略），尽量对齐内存访问并减少缓存未命中；利用编译器自动向量化或手动 SIMD 指令进行向量化计算，使每条指令执行更多数据操作；合理设置线程亲和性（绑定到固定核心），避免跨 NUMA 节点访存带来的延迟。此外，应根据 CPU 核心数量合理选择并行粒度和线程数，避免线程过多导致资源争用或过少导致资源未被充分利用。

在 GPU 端，应利用 SIMT 和多级存储层次结构：根据计算需求设计足够多的线程块（blocks）和线程束（warps），以隐藏全局内存访问延迟；确保线程访问模式相邻且可合并（coalesced），以提高全局内存带宽利用率；使用共享内存缓存重用数据，减少对全局内存的访问；避免分支发散，让同一 warp 内的线程执行相同控制路径。合理选择线程块大小（通常为 warp 大小的整数倍）和网格布局，

使 GPU 的寄存器、共享内存和计算单元等资源得到充分利用。总之，根据 CPU/GPU 的缓存结构、SIMD 宽度、线程块组织等硬件特征进行程序设计和调优，是获得高效并行性能的关键 [1][2]。

6 总结与心得

6.1 总结

在本学期的并行程序设计实验中，我围绕基于概率上下文无关文法（PCFG）的口令猜测算法及其并行化实现进行了深入研究。通过多次实验，我采用多种并行技术（包括 SIMD、多线程（pthread 和 OpenMP）、多进程（MPI）以及 GPU 并行）对算法的各个模块（模型训练、口令生成和 MD5 哈希计算）进行了优化。实验结果表明，这些并行优化技术显著提升了算法的性能，具体如下：

- **SIMD 优化**：通过并行处理多个口令的 MD5 哈希计算，将平均耗时从 3.04601 秒降低到 1.93405 秒（-O2 编译下），加速比约为 1.575。
- **多线程优化**：pthread 和 OpenMP 的结合使用大幅降低了口令生成时间。在 -O0 编译下，pthread 的加速比达到 21.16，OpenMP 的加速比达到 15.41；在 -O2 编译下，pthread 的加速比为 4.11，OpenMP 的加速比为 2.90。
- **多进程优化**：通过 MPI 实现的多进程并行，在 -O0 编译下加速比为 14.24，在 -O2 编译下加速比为 1.98。
- **GPU 并行优化**：GPU 并行在 -O0 编译下将 guess_time 从 1.75816 秒加速到 1.05894 秒，加速比约为 1.66 倍；在 -O0 编译下将 guess_time 从 0.356967 秒加速到 0.338572 秒，加速比约为 1.05 倍。

此外，我还对 PCFG 模型训练阶段进行了并行化改造，通过分治——归并的策略，显著降低了训练时间。在口令猜测部分，我通过并行化 PriorityQueue::init、()PopNext()、Generate() 和 PT::NewPTs() 等函数，进一步提升了性能。最终，在融合加速并行编程阶段，通过综合运用上述并行技术，实现了整个项目的性能优化，将 guess_time 从 -O0 编译下的 7.83099 秒优化到 0.680546 秒，从 -O2 编译下的 0.427328 秒优化到 0.2584 秒。在后续的 PCFG 生成口令算法优化中，我引入了基于束搜索（Beam Search）的分段优化策略，通过限制搜索空间和优化计算流程，显著提升了算法性能。进一步地，通过动态 Top-K 选择策略和概率分布截断机制对 Beam Search 算法进行优化，在保证生成质量的同时，进一步降低了计算复杂度和内存占用。实验结果表明，优化后的 Beam Search 算法在破解时间和破解数量方面表现出显著优势，适用于高效率密码破解场景，增加的正确猜对数量达到了原来的近 10 倍。

6.2 心得

- **并行技术的重要性**：通过本学期的实验，我深刻认识到并行技术在提升程序性能方面的巨大潜力。无论是 SIMD 的数据级并行、多线程和多进程的任务并行，还是 GPU 的大规模并行，都能针对不同的应用场景和瓶颈问题提供有效的解决方案。合理选择和组合这些并行技术，可以显著提高程序的执行效率，满足实际应用中对高性能计算的需求。
- **性能优化的系统性**：性能优化是一个系统性工程，需要从算法设计、数据结构选择、并行策略制定到硬件资源利用等多个方面进行综合考虑。在实验过程中，我发现仅仅依靠单一的优化手段往往难以取得理想的性能提升，只有将算法优化、并行化改造和硬件特性充分利用相结合，才能实

现性能的大幅飞跃。例如，在 PCFG 模型训练阶段，通过分治——归并策略的算法优化，结合 OpenMP 的并行化实现，以及合理设置数据块大小和线程调度策略，最终实现了训练时间的显著降低。

- **并行开销与性能权衡**：在并行化过程中，线程/进程创建、同步、通信等开销是不可避免的。这些开销在某些情况下可能会抵消并行带来的性能提升，甚至导致程序性能下降。因此，在设计并行程序时，需要仔细权衡并行性能提升与开销之间的关系。通过合理划分任务粒度、优化线程间通信机制、减少不必要的同步操作等措施，可以有效降低并行开销，提高并行程序的实际性能。例如，在多线程优化中，我通过使用线程本地容器收集结果，最后再串行合并到优先队列中，减少了线程间的竞争和同步开销，从而提高了并行处理的效率。

- **硬件资源的充分利用**：不同的硬件架构具有不同的性能特点和优势，充分利用硬件资源对于提升程序性能至关重要。在实验中，我针对 CPU 的多核特性采用了多线程和多进程并行，利用其在任务调度和数据共享方面的优势；同时，针对 GPU 的大规模并行计算能力，将计算密集型的 MD5 哈希计算任务迁移到 GPU 上执行，充分发挥了 GPU 在处理高度并行任务时的高吞吐量优势。此外，我还通过优化内存访问模式、合理设置线程块大小等措施，进一步提高了硬件资源的利用率，为程序性能的提升提供了有力支持。

- **算法优化的持续探索**：在追求性能提升的过程中，算法本身的优化同样重要。通过对 PCFG 生成口令算法的深入研究，我引入了基于束搜索的分段优化策略，并进一步提出了动态 Top-K 选择策略和概率分布截断机制。这些算法优化措施不仅提高了密码生成的效率，还提升了生成结果的质量和准确性。这使我认识到，算法优化是一个持续探索和改进的过程，需要不断地从理论和实践两个方面进行深入研究，以寻找更高效、更准确的解决方案，满足实际应用中对性能和质量的双重需求。

通过本学期的并行程序设计实验，我不仅在技术上取得了显著的进步，还在思维方式和解决问题的能力上得到了极大的锻炼。未来，我将继续深入学习并行计算技术，探索更多高效的并行算法和优化策略，为解决实际应用中的复杂计算问题贡献自己的力量。同时，我也将积极参与团队合作和学术交流，与更多的同学分享经验和成果，共同推动并行计算技术的发展和应用。

7 代码地址

这是项目的代码地址：github 仓库地址。

参考文献

[1] LLNL. Introduction to parallel computing tutorial. Technical report, HPC @ LLNL, n.d.

[2] K Asanovic, R Bodik, B. C Catanzaro, J. J Gebis, Parry Husbands, K Keutzer, David Patterson, W. L Plishker, John Shalf, and Samuel Williams. The landscape of parallel computing research: A view from berkeley. *Technical Report Uc Berkeley*, eecs-2006-183, 2006.