南开大学

Nankai University

南　　开　　大　　学

计 算 机 学 院

并行程序设计实验报告

GPU 编程

许洋

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 7 月 2 日

摘要

研究了基于 GPU 并行计算的 PCFG（概率上下文无关文法）口令猜测算法优化方法。通过分析 Rockyou 等密码数据集的结构特征，针对算法中口令生成阶段的性能瓶颈，提出并实现了优化方案。

基础优化通过动态任务分流机制（阈值 MIN_GPU_SIZE=1000），将单个 PT 内的口令生成循环映射为 GPU 线程并行任务，设计扁平化内存布局降低传输开销。进一步探索时意外发现了批处理架构，引入批量 PT 处理策略（BATCH_SIZE=64），合并多个 PT 的 GPU 调用，减少优先队列操作和 P 传输次数。最后进阶操作，探索多 PT 协同生成模型，采用 Block-PT 映射机制实现核函数级并行，并通过前缀预计算优化多 segment 拼接效率。

关键字：GPU 并行计算，PCFG 口令猜测，CUDA 优化

目录

一、 问题重述

PCFG（概率上下文无关文法）口令猜测算法是一种融合统计学习与文法规则的密码攻击技术，相比传统暴力破解和字典攻击能更高效生成密码候选列表。该算法通过分析密码数据集（如 Rockyou）提取密码结构模式（如 L8D3S3 表示 8 字母 +3 数字 +3 符号），并建立概率模型指导密码生成。其实验流程包含四个关键阶段：训练集结构分析、概率模型构建、候选密码生成以及按概率排序的密码验证。在本次 GPU 实验中，重点优化后三个阶段的口令生成与验证性能。

随着密码规模增至 10^7 量级，串行实现面临显著性能瓶颈：MD5 哈希验证占据超过 70% 的计算时间；不同密码结构（PT）产生的候选密码量差异悬殊，如 L6S1 结构平均生成 52,000 个密码而 D4S2 仅 82 个；优先队列需要动态管理数千个 PT 结构的概率关系。这些特性使得 GPU 并行化成为提升效率的必然选择。

基础优化目标是将 PT 内部的口令生成循环通过 GPU 并行化，将串行生成逻辑（如字符组合、概率计算）映射为 GPU 线程并行任务，同时设计高效的 GPU 内存分配策略（如全局内存、共享内存），减少主机与设备间的数据拷贝开销。进阶优化目标是多 PT 批量 GPU 并行化，在传统方案中，每次仅向 GPU 传送单个 PT，未利用 GPU 多线程并发能力。需要设计数据结构一次性装载多个 PT 至 GPU，实现多 PT 并行生成。

二、 实现思路

(一) guessing.cu 的 GPU 实现思路

通过动态任务分流和批量处理实现性能优化。核心思路是根据数据规模动态选择计算路径，当待处理候选值数量超过预设阈值 (1000 条) 时，激活 GPU 加速通道，否则使用传统 CPU 处理。这种设计既利用了 GPU 的并行计算能力，又避免了小数据量时的设备调用开销。实现包含五个关键层面：首先设计了预分配的设备内存池，通过全局指针维护设备内存状态，消除重复分配开销。其次数据结构优化，主机端将离散的候选值连续打包存储，分离偏移量和长度信息，输出预分配 64 字节固定槽位。GPU 核函数采用两级拼接策略——每个线程处理一个候选值，先复制共享前缀 (所有线程访问相同的 d_prefix)，再追加专属段值 (通过 d_input 偏移量访问)，最后添加终止符。在运行控制层面，创建专用 CUDA 流，异步拷贝主机数据到设备 (输入值、偏移量、长度、前缀)，触发核函数 (基于线程块自动划分任务)，异步回传结果。整个过程与主机计算重叠进行，通过流同步确保数据一致性。

三、 GPU 实现

(一) guessing.cu

GPU 核函数设计与实现

核函数

```
__global__ void fastGenerateKernel(char* d_input, int* d_offsets, int*
    d_lengths,
                                   char* d_prefix, int prefix_len,
                                   char* d_output, int max_output_len, int
                                       num_items) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  // 计算线程全局索引
    if (idx >= num_items) return;  // 边界检查: 跳过无效线程
```

```
    // 计算当前线程输出位置: 每个结果占固定槽位
    char* output_start = d_output + idx * max_output_len;
    int write_pos = 0;  // 当前写入位置指针
    // 阶段1: 复制共享前缀 (所有线程相同)
    if (d_prefix && prefix_len > 0) {  // 存在有效前缀
        // 逐字符复制, 同时检查输出缓冲区边界
        for (int i = 0; i < prefix_len && write_pos < max_output_len - 1; i
            ++) {
            output_start[write_pos++] = d_prefix[i];  // 合并内存访问
        }
    }
    // 阶段2: 复制线程专属段值
    char* input_start = d_input + d_offsets[idx];  // 值在输入缓冲区位置
    int input_len = d_lengths[idx];  // 当前值长度
    // 逐字符复制, 注意缓冲区边界保护
    for (int i = 0; i < input_len && write_pos < max_output_len - 1; i++) {
        output_start[write_pos++] = input_start[i];  // 连续内存访问
    }
    // 阶段3: 添加字符串终止符
    output_start[write_pos] = '\0';  // 保证有效C字符串
}
```

线程映射机制采用了一维网格布局模型，通过 blockIdx.x * blockDim.x + threadIdx.x 计算公式为每个候选值分配唯一全局索引，实现了大规模并行处理能力。这种设计确保了数万个密码段可以被同时处理，每个线程专注于单个密码段的高效生成。边界检查机制 idx >= num_items 智能过滤无效线程请求，从根本上防止了内存越界风险，保证了在非整数倍线程块配置时的稳定运行。

架构将密码生成过程划分为三阶段，在共享前缀复制阶段，所有线程同步访问只读前缀数据，充分利用 GPU 的常量缓存特性，显著减少全局内存访问延迟；专属段值复制阶段采用局部性优化设计，每个线程通过预计算的偏移量 d_offsets[idx] 和长度 d_lengths[idx] 精准定位私有数据区，实现零冲突访问；终止符添加阶段则通过单指令写入终止符，不仅保证输出合法性，还巧妙规避了字符串拼接常见的长度计算开销。

内存访问也进行优化，在空间维度上，输出缓冲区确保相邻线程访问连续内存地址；时间维度上，通过合并访问机制将线程束内 32 个线程的分散请求合并为单个高速缓存事务；安全维度上，双重边界保护（循环条件 write_pos < max_output_len - 1 和终止符写入位置控制）。

资源使用也进行优化，核函数内无内存分配操作，同时局部变量最大限度使用寄存器。

GPU 资源管理系统

GPU 资源管理

```
// 全局设备内存池
static char* g_d_input, g_d_prefix, g_d_output, g_d_offsets, g_d_lengths;
// 候选值输入缓冲区, 共享前缀缓冲区, 结果输出缓冲区, 值偏移量数组, 值长度数组
// 资源配置参数
static size_t g_max_items = 100000;     // 最大处理项数(10万)
static size_t g_max_input_size = 50 * 1024 * 1024;    // 输入缓冲50MB
static size_t g_max_output_size = 100 * 1024 * 1024; // 输出缓冲100MB
```

```
static bool g_gpu_initialized = false; // 初始化状态标志
// 初始化函数: 设备内存分配
bool initGPU() {
    if (g_gpu_initialized) return true;  // 避免重复初始化
    // 原子性分配五类设备内存
    cudaError_t err[5] = {cudaMalloc(&g_d_input, g_max_input_size) // 分配其
        他资源};
    // 成功判定: 所有分配均成功
    if (err[0,1,2,3,4]==cudaSuccess) {
        g_gpu_initialized = true;
        return true;
    }
    // 失败处理: 逆向释放已分配资源
    if (g_d_input) cudaFree(g_d_input);
    // 其他资源
    return false; // 传递错误状态
}
```

资源池化设计使得系统在初始化阶段即完成五类关键设备内存的单次预分配（输入/输出/前缀缓冲区、偏移量/长度数组），形成永久性资源池。减少运行时分配开销，同时资源池具备动态调优能力，通过全局参数 g_max_input_size/g_max_output_size 实现按需扩容，充分发挥硬件潜能。全局指针设计消除跨模块指针传递复杂性，使得任务调度模块可通过简单指针解引用访问设备资源，大幅降低系统耦合度。

原子性分配策略建立了健壮的错误处理体系。在初始化阶段采用全或无原则：五类设备内存必须全部成功分配才能启用 GPU 加速，任一分配失败即触发逆向资源释放流程，确保系统始终处于确定状态。这种原子化事务管理机制实现保障功能，资源状态一致性保证消除了资源泄漏风险；错误回滚机制在 CUDA 错误时自动清理半初始化状态；双重验证逻辑（g_gpu_initialized 标志位 +cudaError 双重检查）彻底杜绝重复初始化可能。

进行容量规划采用数据驱动的资源配置模型。输入缓冲区配置 50MB 容量，其设计基于百万量级密码段的统计分析，匹配主流 GPU 的并行处理能力。输出缓冲区 100MB 规划遵循密码安全规范。前缀缓冲区采用最小化设计（1KB）。三类缓冲区形成金字塔式容量结构，减少内存浪费。

高效 GPU 数据处理

GPU 数据处理（伪）

```
function processWithGPU(values: list[str], prefix: str) -> (bool, results:
    list[str]):
    # 阶段1: 初始化与验证
    if not initGPU(): return False, None
    if values is empty or too large: return False, None
    # 阶段2: 内存计算
    total_input_size = 总字符串长度 + values数量  # 每个值后加分隔符
    output_size = values数量 * 64  # 固定输出长度
    # 阶段3: 主机内存分配
    h_input = 分配(total_input_size)          # 扁平化字符串存储
    h_offsets = 分配(values数量 * int大小)    # 各字符串起始位置
```

```
    h_lengths = 分配(values数量 * int大小)    # 各字符串长度
    h_output = 分配(output_size)              # 输出缓冲区
    # 阶段4: 数据打包
    current_offset = 0
    for i, val in enumerate(values):
        h_offsets[i] = current_offset
        h_lengths[i] = len(val)
        复制val内容到h_input[current_offset]
        current_offset += len(val)
        h_input[current_offset] = 分隔符  # 添加分隔符
        current_offset += 1
    # 阶段5: GPU数据传输
    stream = 创建CUDA流()
    异步复制h_input -> 设备内存
    异步复制h_offsets -> 设备内存
    异步复制h_lengths -> 设备内存
    if prefix不为空:
        异步复制prefix -> 设备内存
    # 阶段6: GPU核函数执行
    blocks = 计算所需线程块数量()
    启动核函数(blocks, 512, stream):
        参数: 输入数据, 偏移量, 长度, 前缀, 输出缓冲区
    # 阶段7: 取回结果
    异步复制GPU输出 -> h_output
    等待所有操作完成(stream)
    # 阶段8: 结果处理
    results = []
    for i in range(values数量):
        start = i * 64
        results.append(从h_output提取字符串(start, 64))
    # 阶段9: 清理资源
    释放所有主机内存()
    销毁CUDA流()
    return True, results
```

从资源验证阶段开始，通过 GPU 初始化状态检查/任务规模校验/显存容量预判确保处理环境可靠性。内存计算阶段采用动态预取技术，对百万量级密码段实施实时长度采样。数据打包阶段应用内存压缩算法，采用扁平化存储方式，加快传输，同时通过零拷贝内存映射技术消除打包过程中的两次冗余复制（传统 memcpy 操作）。

系统引入 CUDA 流引擎实现硬件级并行调度，构建三大并行通道：PCIe 传输通道（DMA 控制器直连主机内存）、核函数执行通道（SM 多处理器并发）、结果回传通道（H2D 复制引擎）。这种架构形成重叠执行窗口，当核函数处理第 N 批数据时，结果回传通道正处理第 N-1 批结果，同时主机向设备传输第 N+1 批数据。系统设置智能同步屏障，采用事件驱动机制（cudaEventRecord/cudaEventSynchronize）替代传统全流阻塞。

数据布局优化采用扁平化存储，将离散的字符串对象转化为连续内存块。将元数据分离，偏移量和长度数组独立存储，同时固定槽位输出，64 字节对齐内存设计直接映射 L2 缓存行。

CPU-GPU 协同调度系统

Generate 函数

```
void PriorityQueue::Generate(PT pt) {
    // ... 概率计算等预处理 ...
    const int MIN_GPU_SIZE = 1000; // GPU激活阈值
    if (pt.content.size() == 1) { // 单segment情况
        segment* a = /* 类型相关定位逻辑 */;
        int num_values = pt.max_indices[0];
        // GPU路径决策
        if (num_values >= MIN_GPU_SIZE) {
            vector<string> temp_results;
            if (processWithGPU(a->ordered_values, "", temp_results)) {
                // 批量添加结果
                for (const auto& result : temp_results) {
                    guesses.push_back(result);
                    total_guesses++;
                }
                return; // 成功则提前退出
            }// GPU失败时自动回退CPU
        }
        // CPU回退路径
        for (int i = 0; i < num_values; i++) {
            guesses.push_back(a->ordered_values[i]);
            total_guesses++;
        }
    } else { // 进行类似操作
    }
}
```

动态路径决策引擎实现智能计算，决定由 GPU 还是 CPU 来进行计算。系统设置可调阈值 MIN_GPU_SIZE=1000 作为分水岭：当候选密码段数量 1000 时，激活高性能 GPU 通道，通过并行核函数生成密码，小规模任务则自动保留给 CPU 处理，规避 GPU 启动开销，设计自动回退功能，当 GPU 初始化失败 (cudaError_t 检测) 或内存超限时无缝切换至 CPU 路径，同时针对海量数据进行批处理优化，系统检测到 num_values>50,000 时自动分割为多个批次，防止内核超时。

在单 segment 处理时，直接调用 processWithGPU(values, "", results)，空前缀参数激活高效模式。在多 segment 场景时采用前缀预计算技术。通过

拼接

```
string prefix;
for (int idx : pt.curr_indices) {
    // 遍历非结尾segment构建前缀
    if (seg.type==1) prefix += letters[seg].values[idx];
    // 类型判断与拼接
}
```

算法保持 O(n) 时间复杂度，支持大量密码空间。同时设计统一接口 processWithGPU，对外提供一致的调用参数 (values, prefix)，大幅降低系统复杂性。

在资源优化整合方面实现零开销结果管理采取直接合并技术,GPU 生成的结果通过 temp_results 向量直接 merge 到全局队列。

资源合并

```
for (const auto& result : temp_results) {
    guesses.push_back(result); // 无拷贝移动语义
}
```

(二) 结果

没有实现显著的加速，guess time 与串行相比相差不大，同时 cracked 数量没有发生变化。

分析没有实现显著加速的可能原因:

1.GPU 启动开销占比过大，每次调用 GPU 需要创建 CUDA 流、多次内存拷贝、内核启动、同步等待等固定开销，GPU 启动开销是实际计算的多倍。

2. 数据传输瓶颈，数据传输过程中占用大量时间，降低了整体的运算效率。

3. 内存分配开销过大，每次任务要进行 4 次 malloc 和 4 次 free

四、 意外收获

这里是在探究 PT 并行时候意外改出来的。震惊!!! 实现了加速，效果还不错。

(一) 代码实现

优化的 GPU 字符串复制函数

GPU 字符串复制

```
void copy_strings_to_gpu_optimized(const vector<string>& strings, char*
    d_buffer, int* d_offsets, int* d_lengths) {
    if (strings.empty()) return;  // 空检查
    vector<int> offsets(strings.size());  // 主机偏移数组
    vector<int> lengths(strings.size());   // 主机长度数组
    // 计算总内存需求
    int total_size = 0;
    for (size_t i = 0; i < strings.size(); i++) {
        offsets[i] = total_size;  // 当前字符串偏移
        lengths[i] = strings[i].length();  // 当前字符串长度
        total_size += lengths[i] + 1;  // 累加总大小（含终止符）
    }
    // 创建连续的主机缓冲区
    char* host_buffer = new char[total_size];
    // 复制所有字符串到连续内存
    for (size_t i = 0; i < strings.size(); i++) {
        memcpy(host_buffer + offsets[i], strings[i].c_str(), lengths[i] + 1);
    }
    // 一次性批量传输到GPU
```

```
    CUDA_CHECK(cudaMemcpy(d_buffer, host_buffer, total_size * sizeof(char),
        cudaMemcpyHostToDevice));
    //其他资源
    delete[] host_buffer;  // 释放临时主机内存
}
```

copy_strings_to_gpu_optimized 函数实现了高效的主机到设备字符串传输机制。首先在主机端计算字符串数据, 为每个字符串计算偏移位置和长度信息，然后将所有字符串集中复制到连续的临时缓冲区中。通过三次批量的 cudaMemcpy 操作（字符串内容、偏移数组和长度数组），一次性完成所有数据到 GPU 的传输。这种优化方法相较于逐字符串传输减少了大量小的数据传输，大幅提高传输效率。函数还包含完善的错误检查和内存管理，确保临时缓冲区的正确释放，解决了 GPU 数据处理中的关键瓶颈问题。

密码生成主循环

PopNext 函数

```
void PriorityQueue::PopNext() {
    const int BATCH_SIZE = 64;  // 批量处理大小
    vector<PT> batch_pts;
    // 1. 批量获取PT
    for (int i = 0; i < BATCH_SIZE && !priority.empty(); i++) {
        batch_pts.push_back(priority.front());  // 取出最前面的PT
        priority.erase(priority.begin());  // 从队列移除
    }
    // 2. 批量生成密码
    if (!batch_pts.empty()) {
        BatchGenerate(batch_pts);
    }
    // 3. 生成新PT并插入队列
    for (PT& processed_pt : batch_pts) {
        vector<PT> new_pts = processed_pt.NewPTs();  // 创建变体
        for (PT pt : new_pts) {
            CalProb(pt);  // 计算新PT概率
            // 按概率插入到正确位置（降序）
            bool inserted = false;
            for (auto iter = priority.begin(); iter != priority.end(); iter
                ++) {
                if (pt.prob > iter->prob) {
                    priority.emplace(iter, pt);
                    inserted = true;
                    break;
                }
            }
            if (!inserted) {
                priority.emplace_back(pt);  // 插入队尾
            } } } }
```

PopNext 实现密码生成的核心循环逻辑。函数以批量方式高效处理密码模板：一次取出 64

个模板（BatchGenerate），生成当前密码组合；然后为每个模板创建新变体（通过 NewPTs 函数递增终结符索引）；重新计算变体概率后按序插回优先队列。批量处理机制大幅减少优先队列操作次数，平衡了并行加速和顺序处理的需求。概率排序确保系统始终优先探索最有希望的密码组合，显著提高密码猜测效率。

其他辅助函数

批量处理

```
// 批量生成密码
void PriorityQueue::BatchGenerate(const vector<PT>& batch_pts) {
    for (const PT& pt : batch_pts) {
        Generate(const_cast<PT&>(pt));
    }   }
// 清理GPU资源
void cleanup_gpu_resources() {
    if (g_d_input) { cudaFree(g_d_input); g_d_input = nullptr; }
    // 其他资源
    g_gpu_initialized = false;
}
```

辅助功能包括批量处理函数 BatchGenerate，它简单迭代调用 Generate 处理密码模板集合。全局资源清理函数 cleanup_gpu_resources 确保程序退出时释放所有 GPU 资源，防止内存泄漏。这些补充功能使系统更完整健壮，提供更好的资源管理特性，支持长时间运行的稳定性。

其他部分 CUDA 核函数、GPU 资源管理与初始化以及 GPU 处理主函数等都和基础部分相同。

(二) 结果

这个本质上还是串行的，但是实现了加速，达到了 1.25 倍的加速比。

分析实现加速的原因：

1. 批处理架构重构，引入批量 PT 处理机制（BatchGenerate 函数），将原始的单 PT 处理模式改为每次处理 64 个 PT 单元的批量操作，将 GPU 调用次数从每 PT 一次降低至每批一次，有效分摊了 GPU 启动开销。

2. 内存传输进行优化，实现 copy_strings_to_gpu_optimized 高效传输函数，通过主机端连续缓冲区整合将传输次数从每个字符串单独传输优化为单批次传输，大幅减少 PCIe 调用。

3. 资源管理强化，实现 cleanup_gpu_resources 统一释放机制防止内存泄漏。

五、 进阶实现 1

(一) 实现目的

尝试使用多进程编程，在 PT 层面实现并行计算。先前的并行算法是对于单个 PT 而言，使用多进程进行并行的口令生成，现在一次性从优先队列中取出多个 PT，并同时进行口令生成。

在实现过程中需要注意每个 PT 生成之后均需要将产生的新 PT 放回优先队列，如果一次性取出多个 PT，那么等待各 PT 生成完成后，再将一系列新的 PT 挨个放回优先队列。

(二) 代码实现

GPU 核函数

核函数参数

```
__global__ void gpu_batch_generate_guesses(// 各种参数)
```

核函数有 11 个参数，主要分为三大类：

输入数据：所有 segment 值的扁平化存储，每个 PT 的元数据 (偏移量、值数量等)，每个 PT 的前缀数据

输出数据：生成的密码猜测，每个猜测的实际长度

控制参数：PT 数量，最大密码长度，核函数主体逻辑

并行策略

```
{
    // 确定当前处理的PT索引和线程ID
    int pt_id = blockIdx.x;
    int thread_id = threadIdx.x;
    // 检查索引是否在有效范围内
    if (pt_id >= num_pts) return;
    // 获取当前PT的相关信息
    int last_seg_count = d_last_seg_counts[pt_id]; // 最后一个segment的值数量
    int prefix_len = d_prefix_lengths[pt_id];      // 前缀的实际长度
```

每个 PT 分配一个 CUDA block(blockIdx.x)，每个 block 内使用多个线程 (threadIdx.x) 并行处理 PT 的值。last_seg_count 是值数量，prefix_len 是前缀实际长度

并行循环

```
    // 并行处理当前PT的所有值
    for (int value_idx = thread_id; value_idx < last_seg_count; value_idx +=
        blockDim.x) {
        // 计算当前值对应的输出位置(在d_output_guesses中的索引)
        int output_offset = d_pt_offsets[pt_id] + value_idx;
```

循环变量 value_idx 从线程 ID 开始，以 blockDim.x(线程数) 为步长，实现线程间的任务分配，output_offset 计算当前值的输出位置

信息复制

```
        // 步骤1: 复制前缀到输出缓冲区
        for (int i = 0; i < prefix_len && i < max_guess_length; i++) {
            d_output_guesses[output_offset * max_guess_length + i] =
                d_prefixes[pt_id * max_guess_length + i];
        }
        // 步骤2: 获取当前值的信息
        int value_data_offset = d_value_offsets[output_offset];
        int value_len = d_value_lengths[output_offset];
        // 步骤3: 添加当前值到输出缓冲区
        for (int i = 0; i < value_len && (prefix_len + i) < max_guess_length;
             i++) {
```

```
            d_output_guesses[output_offset * max_guess_length + prefix_len +
                i] =
                d_all_segment_values[value_data_offset + i];
        }
        // 步骤4: 记录实际密码长度
        d_output_counts[output_offset] = min(prefix_len + value_len,
            max_guess_length);
    }
}
```

将 PT 的前缀复制到输出缓冲区的前部，使用 pt_id * max_guess_length 定位前缀起始位置，确保复制长度不超过最大密码长度

value_data_offset 储存当前值在字符数组中的位置，value_len 储存当前值的实际长度复制过程将当前值添加到前缀后面，使用 prefix_len + i 定位输出位置, 同时确保总长度不超过最大密码长度

最后记录最终密码的实际长度 (前缀长度 + 值长度)，使用 min 确保不超过缓冲区大小，将长度存入 d_output_counts 供后续处理使用

多 PT 处理

处理前准备

```
void PriorityQueue::BatchProcessPTs(int batch_size)
{
    if (priority.empty()) return;
    vector<PT> batch_pts;
    vector<string> batch_prefixes;
    for (int i = 0; i < batch_size && !priority.empty(); i++) {
        PT current_pt = priority.front();
        priority.erase(priority.begin());
```

批处理准备,初始化两个向量,batch_pts 用于存储当前批次要处理的 PT 对象,batch_prefixes 用于存储每个 PT 的前缀字符串（即除最后一个 segment 外的所有值组成的字符串)。然后从优先队列头部取出最多 batch_size 个 PT 进行后续处理。

单 segement

```
        if (current_pt.content.size() == 1) {
            segment *seg;
            if (current_pt.content[0].type == 1) {
                // 串行方法的变式    }
        }
```

对于只有一个 segment 的 PT，直接在 CPU 上处理，根据 segment 类型定位到相应的模型数据，将模型中所有值直接作为密码猜测添加到全局列表，不再生成新的 PT（单 segment PT 不产生子 PT)。

多 segement

```
        batch_pts.push_back(current_pt);
        string prefix = "";
```

```
        int seg_idx = 0;
        for (int idx : current_pt.curr_indices) {
            // 串行方法的变式
            seg_idx++;
        }
        batch_prefixes.push_back(prefix);
    }
```

对于多 segment 的 PT，将其加入批次列表，遍历除最后一个 segment 外的所有 segment，根据当前索引值从对应模型中取出具体值并拼接

数据回传准备

```
    const int MAX_GUESS_LENGTH = 64;
    vector<char> all_segment_values;
    vector<int> pt_offsets;
    vector<int> value_offsets;
    vector<int> value_lengths;
    vector<int> last_seg_counts;
    vector<char> prefixes(batch_pts.size() * MAX_GUESS_LENGTH, '\0');
    vector<int> prefix_lengths;
    int total_values = 0;
    int data_offset = 0;
```

准备将数据传输到 GPU 所需的各种数据结构:

all_segment_values–所有值的字符数据扁平化存储，pt_offsets–每个 PT 在值列表中的起始索引，value_offsets–每个值在字符数组中的偏移量，value_lengths–每个值的长度，prefixes–固定长度的前缀数组（不足部分空字符填充）。

数据结构填充

```
    for (int pt_idx = 0; pt_idx < batch_pts.size(); pt_idx++) {
        PT& pt = batch_pts[pt_idx];
        pt_offsets.push_back(total_values);
        // 获取最后一个segment...
        last_seg_counts.push_back(last_seg->ordered_values.size());
        // 复制前缀...
        for (const string& value : last_seg->ordered_values) {
            value_offsets.push_back(data_offset);
            value_lengths.push_back(value.length());
            for (char c : value) {
                all_segment_values.push_back(c);
                data_offset++;
            }
            total_values++;
        }
    }
```

遍历批次中的每个 PT，记录 PT 在值列表中的起始位置，获取最后一个 segment 并存储其值数量，然后将前缀字符串复制到固定长度的数组中。之后处理最后一个 segment 的所有值，记录偏移量和长度，并将字符数据添加到扁平化数组。

GPU 调用

```
    char *d_all_segment_values, *d_prefixes, *d_output_guesses;
    int *d_pt_offsets, *d_value_offsets, ...;
    cudaMalloc(&d_all_segment_values, all_segment_values.size() * sizeof(char
        ));
    // ...其他分配...
    cudaMemcpy(d_all_segment_values, all_segment_values.data(), ...);
    // ...其他数据复制...
    int num_blocks = batch_pts.size();
    int threads_per_block = 256;
    gpu_batch_generate_guesses<<<num_blocks, threads_per_block>>>(...);
    cudaDeviceSynchronize();
```

分配 GPU 设备内存并复制主机数据到设备，同时配置并启动 GPU 核函数，每个 PT 对应一个 CUDA block，每个 block 使用 256 个线程，传入所有必需的设备指针参数。

GPU 回传

```
    vector<char> output_guesses(total_values * MAX_GUESS_LENGTH);
    vector<int> output_counts(total_values);
    cudaMemcpy(output_guesses.data(), ...);
    for (int i = 0; i < total_values; i++) {
        string guess(...);
        guesses.push_back(guess);
    }
```

将 GPU 生成的猜测复制回主机内存，创建接收缓冲区，执行内存复制，并将字符数组转换为字符串并添加到全局猜测列表。

新 PT 生成

```
    for (PT& pt : batch_pts) {
        vector<PT> new_pts = pt.NewPTs();
        for (PT& new_pt : new_pts) {
            CalProb(new_pt);
            // 按概率插入优先队列...
        }
    }
```

为每个处理过的 PT 生成新的 PT 对象，通过调用 NewPTs() 方法生成可能的子结构，之后计算新 PT 的概率，按概率将新 PT 插入优先队列。

资源处理

```
    cudaFree(d_all_segment_values);
    // ...其他释放...
    total_guesses = guesses.size();
}
```

释放所有 GPU 资源并更新全局猜测计数。

(三) 结果

这里未能实现对 guessing 的加速，Cracked 数量没有很大的变化。

可能原因分析：

1. 内存传输瓶颈：每次批处理需要 7 次独立的内存传输操作，传输数据结构复杂（标量、数组、字符数据混合）。

2. 负载不均衡：不同 PT 的 last_seg_counts 差异巨大（如 L6 可能有 50,000 值，D4 仅 100 值），然而每个 PT 分配相同数量线程 (256)，小 PT 浪费资源，大 PT 处理不足。

3. 并行度不足：主机在 GPU 运行时完全空闲，无流水线处理。

六、 总结

本研究通过 GPU 并行化技术实现了 PCFG 口令猜测算法的系统性优化，基础设计线程级并行核函数；批处理机制创新，将 64 个 PT 单元合并处理以减少 GPU 调用与优先队列维护开销，配合全局内存资源池化消除重复分配成本；进阶实现了多 PT 并行化处理。整个过程进行多方面探索，资源管理方面，建立原子化资源分配体系确保设备状态一致性，实现零泄漏资源释放保障长时间任务稳定性。内存方面，进行共享内存优化，在核函数内缓存高频访问数据（如前缀），减少全局内存访问延迟。

代码仓库：https://github.com/X-u-Y-a-n-g/parallel