南開大學
Nankai University

计算机学院

并行程序设计实验报告

基于 CUDA 的口令猜测优化

姓名：葛明宇

学号：2312388

专业：计算机科学与技术

2025 年 7 月 2 日

目录

1 引言

本实验聚焦于运用 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。

2 基础部分

2.1 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。

2.2 并行实现

CUDA 核函数 generateGuessesKernel()

线程索引计算与边界检查：计算全局线程索引，并过滤超出任务范围的线程，确保每个线程只处理有效的值。

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx >= numValues) return;
```

值数据定位:通过累积长度数组 (d_valueLengths) 定位当前线程对应值的起始位置和长度。valueOffset 表示当前值在 d_values 数组中的起始偏移量；valueLen 表示当前值的长度（通过相邻累积长度的差值计算）。

```
int valueOffset = (idx == 0) ? 0 : d_valueLengths[idx - 1];
int valueLen = d_valueLengths[idx] - valueOffset;
```

输出缓冲区定位：定位当前线程输出结果的存储位置，确保每个线程的输出不会覆盖其他线程的结果。

```
// 获取输出缓冲区的偏移量
int outOffset = d_outputOffsets[idx];
char* outPtr = d_output + outOffset;
```

字符串拼接与结束符添加：将固定前缀 (d_prefix) 复制到输出缓冲区。将当前值 (d_values 中对应位置的数据) 追加到前缀之后。添加字符串结束符 (\0)，确保生成的是完整的 C 风格字符串。

```
    for (int i = 0; i < prefixLen; ++i) {
        outPtr[i] = d_prefix[i];
    }
    // 复制当前值
    for (int i = 0; i < valueLen; ++i) {
        outPtr[prefixLen + i] = d_values[valueOffset + i];
    }
    // 添加字符串结束符
    outPtr[prefixLen + valueLen] = '\0';
```

generate() 函数的改进

为了实现方便，对于单 segment 和多 segment 的情况均使用了统一的 generateGuessesKernel() 核函数。那么在实现单 segment 时需要将 prefix 赋值为“”，同时 prefixlen 赋值为 0。下面只展示多 segment 的情况。

数据准备与元信息计算：计算固定前缀长度和值集合。构建累积长度数组 (valueLengths)，用于定位每个值在扁平化数组中的偏移量。计算输出缓冲区的偏移量 (outputOffsets) 和总大小。

```
    ...
    for (int i = 0; i < numValues; ++i) {
        int len = values[i].length();
        valueLengths[i] = (i == 0) ? len : valueLengths[i - 1] + len;
        outputOffsets[i] = totalOutputLen;
        totalOutputLen += prefixLen + len + 1; // +1 for null terminator
    }
    totalValuesLen = valueLengths.empty() ? 0 : valueLengths.back();
```

GPU 内存分配: 在 GPU 设备上分配五类内存：固定前缀存储 (d_prefix)、扁平化值字符串存储 (d_values)、值长度数组 (d_valueLengths)、输出偏移量数组 (d_outputOffsets)、最终结果缓冲区 (d_output)。

```
    char *d_prefix, *d_values, *d_output;
    int *d_valueLengths, *d_outputOffsets;

    CUDA_CHECK(cudaMalloc(&d_prefix, prefixLen));
    ...
```

数据扁平化与主机到 GPU 传输：将值集合 (values) 扁平化到单个字符串 (flatValues)，减少内存碎片。将四类数据从主机内存传输到 GPU：前缀、扁平化值、长度数组、偏移量数组。

```
    string flatValues;
    for (const auto& val : values) {
        flatValues += val;
    }
    CUDA_CHECK(cudaMemcpy(d_prefix, guess.c_str(), prefixLen,
        cudaMemcpyHostToDevice));
    ...
```

CUDA 核函数调用: 配置线程块和网格大小（每个线程处理一个值）。启动核函数，并行执行前缀

与值的拼接操作。检查核函数启动是否成功。

```
    int blockSize = 256;
    int gridSize = (numValues + blockSize - 1) / blockSize;
    generateGuessesKernel<<<gridSize, blockSize>>>(d_prefix, prefixLen,
                                                   d_values, d_valueLengths,
                                                   d_output, d_outputOffsets,
                                                   numValues);
    CUDA_CHECK(cudaGetLastError());
```

GPU 到主机结果传输：在主机上分配内存存储最终结果。将 GPU 生成的拼接字符串复制回主机。

```
    char* h_output = new char[totalOutputLen];
    CUDA_CHECK(cudaMemcpy(h_output, d_output, totalOutputLen,
        cudaMemcpyDeviceToHost));
```

结果解析与收集：将拼接后的字符串解析为最终结果，并将结果添加到 guesses 列表中。

```
    for (int i = 0; i < numValues; ++i) {
        guesses.emplace_back(h_output + outputOffsets[i]);
        total_guesses++;
    }
```

资源释放: 释放主机端分配的内存 (h_output)。释放 GPU 端分配的所有内存，防止内存泄漏。

```
    delete[] h_output;
    CUDA_CHECK(cudaFree(d_prefix));
    • • •
```

3 阈值优化

3.1 优化设计

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 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。

3.2 优化实现

```
    int numValues = pt.max_indices[pt.content.size() - 1];
```

```
    if (numValues > threshold) {
        // 值的数量超过阈值，使用GPU加速
        ...
    } else {
        // 小规模数据使用CPU
        for (int i = 0; i < numValues; i += 1) {
            string temp = guess + a->ordered_values[i];
            guesses.emplace_back(temp);
            total_guesses += 1;
        }
    }
```

3.3 性能测试

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 1: 阈值优化测试结果

3.4 结果分析

从表 1 的测试结果来看，阈值优化的效果与编译选项密切相关，整体加速比均小于 1，说明当前阈值策略未达到预期收益，具体原因分析如下：

1. **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。

2. **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。

4 多 PT 优化

4.1 优化设计

基础并行设计仅处理单个 PT 的字符串拼接，而实际场景中存在大量独立的 PT 任务，频繁启动核函数会导致 GPU 空闲时间增加。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。

4.2 优化实现

受篇幅限制这里只给出 CUDA 核函数代码。

CUDA 核函数 generate_batch_passwords_kernel()

线程索引计算与边界检查：计算全局线程索引 global_idx，通过循环映射到具体的 PT（Production Template）索引 pt_idx，并进行边界检查，超出范围的线程直接返回。

```
int global_idx = blockIdx.x * blockDim.x + threadIdx.x;
int pt_idx = 0;
int local_password_idx = global_idx;
while (pt_idx < num_pts && local_password_idx >= d_pt_counts[pt_idx]) {
    local_password_idx -= d_pt_counts[pt_idx];
    pt_idx++;
}
if (pt_idx >= num_pts) return;
```

数据索引计算: 根据 PT 索引和局部密码索引，计算在值数组中的实际索引位置。

```
int value_idx = d_pt_starts[pt_idx] + local_password_idx;
```

前缀数据获取：通过偏移量数组定位当前 PT 对应的前缀字符串，并获取前缀长度信息。

```
char* prefix_ptr = d_all_prefixes + d_prefix_offsets[pt_idx];
int prefix_len = d_prefix_lengths[pt_idx];
```

值数据获取：通过偏移量数组定位当前值字符串，同时获取值的长度信息。

```
char* value_ptr = d_all_values + d_value_offsets[value_idx];
int value_len = d_value_lengths[value_idx];
```

字符串拼接与输出：定位输出缓冲区位置，将前缀和值按顺序复制到输出位置，并添加字符串结束符\0。

```
char* output_ptr = d_output + d_output_offsets[global_idx];
for (int i = 0; i < prefix_len; i++) {
    output_ptr[i] = prefix_ptr[i];
}
for (int i = 0; i < value_len; i++) {
    output_ptr[prefix_len + i] = value_ptr[i];
}
```

```
8     output_ptr[prefix_len + value_len] = '\0';
```

4.3 性能测试

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 2: CUDA 多 PT 优化测试结果

4.4 结果分析

表 2 的测试结果显示，多 PT 优化在无编译优化时表现出一定优势（加速比 1.926），但在 O1/O2 优化后加速比下降（0.350-0.351），原因分析如下：

1. **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。

2. **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。

5 混合并行流水线优化

5.1 优化设计

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。

5.2 优化实现

通过“任务分配 → 异步 GPU 处理 →CPU 并行维护队列 → 结果同步”的流水线，实现了 CPU 与 GPU 的高效协同：PopNextBatchCuda 负责全局调度，区分 CPU/GPU 任务并合并结果；GenerateBatchOnGPU_Async 专注于 GPU 数据准备和异步启动，最大化设备利用率；ProcessGPUResults 确保 GPU 结果正确同步并整合到最终列表。代码中的 AsyncGPUData 是自定义的异步 GPU 任务数据结构。

PopNextBatchCuda() 是核心调度逻辑，负责协调 CPU 与 GPU 的任务分配、结果合并和队列维护。

初始化与处理遗留 GPU 任务: 计算实际可处理的批次大小（不超过优先队列长度）；初始化静态变量跟踪 GPU 任务状态；若存在未完成的 GPU 任务，先调用 ProcessGPUResults 处理结果，确保任务不堆积。

```
if (gpu_task_in_progress) {
    ProcessGPUResults(gpu_data);
    gpu_task_in_progress = false;
}
```

预处理批次数据: 遍历批次内的每个 PT，计算概率并构建其前缀（前 n-1 个 segment 的 value 拼接）；按 threshold 阈值区分任务：小批量 value 由 CPU 直接生成密码，大批量由 GPU 处理；收集每个 PT 生成的新 PT（通过 NewPTs），用于后续更新优先队列。

```
for (int i = 0; i < actual_batch_size; i++) {
    CalProb(priority[i]);
    PTData pt_data = BuildPTData(priority[i]);

    if (pt_data.values.size() < threshold)
        HandleCPUWork(pt_data);  // 小任务CPU处理
    else
        pt_data_batch.push_back(pt_data);  // 大任务GPU处理

    all_new_pts.insert(all_new_pts.end(),
                       priority[i].NewPTs().begin(),
                       priority[i].NewPTs().end());
}
```

处理 CPU 任务: 将 CPU 生成的密码数量累加到总计数，确保本地任务先完成。

```
total_guesses += cpu_work_passwords;
```

启动 GPU 异步批量处理：若存在 GPU 任务，调用 GenerateBatchOnGPU_Async 异步启动 GPU 处理；标记 GPU 任务为“进行中”，避免重复处理。

```
if (!pt_data_batch.empty() && total_passwords > 0) {
```

```
    GenerateBatchOnGPU_Async(pt_data_batch, total_passwords, gpu_data);
    gpu_task_in_progress = true;
}
```

移除已处理的 PT：从优先队列中删除当前批次已处理的 PT，释放资源。

```
priority.erase(priority.begin(), priority.begin() + actual_batch_size);
```

CPU 并行合并新 PT 到优先队列：在 GPU 处理密码生成时，CPU 同步对新 PT 进行排序（按概率降序）；将排序后的新 PT 与剩余的旧 PT 合并，维持优先队列的有序性（高概率 PT 在前）；利用 GPU 计算的时间窗口，充分利用 CPU 资源更新队列，实现流水线并行。

```
if (!all_new_pts.empty()) {
    ...
    while (i < priority.size() && j < all_new_pts.size()) {
        if (priority[i].prob >= all_new_pts[j].prob) {
            new_priority.push_back(priority[i++]);
        } else {
            new_priority.push_back(all_new_pts[j++]);
        }
    }
    ...
}
```

处理当前 GPU 任务结果: 等待 GPU 异步任务完成后，调用 ProcessGPUResults 处理结果；标记 GPU 任务为“已完成”，确保资源正确释放。

```
if (gpu_task_in_progress) {
    ProcessGPUResults(gpu_data);
    gpu_task_in_progress = false;
}
```

GenerateBatchOnGPU_Async() 负责为 GPU 准备数据并异步启动核函数。

计算内存需求与偏移量: 遍历 GPU 任务批次的每个 PT，计算前缀、value、输出结果的总内存需求；记录各类偏移量（如 prefix_offsets 标记每个前缀在合并数组中的位置）和长度，用于 GPU 核函数快速定位数据；扁平化数据存储（合并为连续数组），减少 GPU 内存碎片，提高访问效率。

```
for (const auto& pt_data : pt_data_batch) {
    prefix_offsets.push_back(current_prefix_offset);
    current_prefix_offset += pt_data.prefix.length();

    for (const string& value : pt_data.values) {
        value_offsets.push_back(current_value_offset);
        output_offsets.push_back(current_output_offset);
        current_value_offset += value.length();
        current_output_offset += pt_data.prefix.length() + value.length() + 1;
    }
}
```

分配主机内存并填充数据: 在主机上分配内存，存储合并后的前缀、value 和输出结果；将每个 PT 的前缀和 value 复制到合并数组（h_all_prefixes、h_all_values），通过 memcpy 高效填充数据。

```
gpu_data.h_all_prefixes = new char[total_prefix_size];
gpu_data.h_all_values = new char[total_values_size];
// ... 填充数据 (memcpy)
```

异步分配设备内存: 使用 cudaMallocAsync 在 GPU 上异步分配内存，避免 CPU 等待内存分配完成；为合并后的前缀、value、输出结果及各类偏移量数组分配设备内存，与主机内存结构对应。

```
// 异步分配设备内存
cudaMallocAsync(&gpu_data.d_all_prefixes, total_prefix_size, gpu_data.stream);
• • •
```

异步传输数据到 GPU: 使用 cudaMemcpyAsync 将主机数据（合并的前缀、value、偏移量数组）异步传输到 GPU；所有传输操作绑定到 gpu_data.stream，与 CPU 后续操作并行执行，减少等待时间。

```
cudaMemcpyAsync(gpu_data.d_all_prefixes, gpu_data.h_all_prefixes, total_prefix_size,
                cudaMemcpyHostToDevice, gpu_data.stream);
• • •
```

异步启动 GPU 核函数：配置线程块大小（256）和网格大小（根据总密码数动态计算）；在指定流（gpu_data.stream）中异步启动核函数 generate_batch_passwords_kernel，GPU 开始并行生成密码；核函数参数为设备端数据地址，确保 GPU 可直接访问前缀、value 和偏移量数组。

```
generate_batch_passwords_kernel<<<gridSize, 256, 0, gpu_data.stream>>>(
    gpu_data.d_all_prefixes, ..., gpu_data.d_output_offsets);
```

ProcessGPUResults() 负责同步 GPU 任务并处理结果。

同步 CUDA 流确保操作完成: 等待流中所有 GPU 操作（数据传输、核函数执行）完成，确保结果就绪。

```
cudaStreamSynchronize(gpu_data.stream);
```

复制结果从设备到主机: 异步将 GPU 生成的密码结果（d_output）复制回主机内存（h_output）；再次同步流，确保复制完成后再处理数据。

```
cudaMemcpyAsync(gpu_data.h_output, gpu_data.d_output, gpu_data.total_passwords,
                cudaMemcpyDeviceToHost, gpu_data.stream);
cudaStreamSynchronize(gpu_data.stream);
```

解析结果并添加到猜测列表: 根据预计算的输出偏移量（output_offsets），从合并的输出数组（h_output）中提取每个密码字符串；将 GPU 生成的密码添加到 guesses 列表，并累加到总计数。

```
for (int i = 0; i < gpu_data.total_passwords; i++) {
    guesses.push_back(string(gpu_data.h_output + gpu_data.output_offsets[i]));
}
total_guesses += gpu_data.total_passwords;
```

5.3 性能测试

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 3: 混合并行流水线优化测试结果

5.4 结果分析

表 3 的测试结果显示，混合并行流水线优化在无编译优化时加速比为 1.950（优于多 PT 优化的 1.926），但在 O1/O2 优化后仍不理想（加速比 0.405），具体分析如下：

1. **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%。这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 计算与 CPU 队列维护的重叠”。

2. **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。

6 总结

本文围绕字符串拼接任务的并行优化展开研究，通过基础并行设计、阈值优化、多 PT 优化及混合并行流水线优化四个阶段，逐步探索 CPU 与 GPU 协同计算的高效模式，主要工作总结如下：

基础并行设计：基于 GPU 数据并行范式，将独立的“前缀 + 值”拼接任务分配给单个线程，通过扁平化数据存储和偏移量数组实现高效内存访问，验证了 GPU 在大规模字符串拼接任务中的并行潜力。

阈值优化：尝试通过动态选择计算单元（CPU/GPU）平衡不同数据规模的效率，结果显示在无编译优化时 GPU 优势初步显现，但编译优化后 CPU 单线程性能显著提升，阈值策略需进一步调整以适应硬件特性。

多 PT 优化：通过批量处理多个 PT 任务减少核函数启动开销，在无编译优化场景下实现 1.926 倍加速，证明批量调度能有效提升 GPU 利用率，但多 PT 数据的分散访问和线程映射开销在编译优化后成为瓶颈。

混合并行流水线优化：通过异步操作与任务拆分实现 CPU 与 GPU 协同，在无编译优化时加速比提升至 1.950，进一步挖掘了设备重叠工作的潜力，但小规模任务下异步管理开销抵消了并行收益。

7 代码仓库

仓库链接 Gitee