南開大學
Nankai University

计算机学院

并行程序设计报告

并行加速的口令猜测算法

姓名：潘涛

学号：2314033

专业：计算机科学与技术

2025 年 7 月 2 日

目录

1 问题描述

对于一个用户的口令，对其进行猜测的基本策略是：生成一个按照概率降序排列的口令猜测词典，这个词典包括一系列用户可能选择的口令。那么，现在问题就变成了：1. 如何有效生成用户可能选择的口令；2. 如何将生成的口令按照降序进行排列。在这个选题中，使用最经典的 PCFG（概率上下文无关文法）模型来进行口令的生成，并且尝试将其并行化，以提升猜测的时间效率。口令猜测选题的框架主要分为三个部分：模型训练、口令生成、MD5 哈希值生成。在本篇中，我们将着重讨论口令生成部分的 CPU 的并行实现，以提高口令猜测的时间。

2 cuda 编程

CUDA 是 NVIDIA 开发的并行计算平台和编程模型，它允许开发者利用 GPU 的强大并行处理能力来加速计算密集型应用程序。通过 CUDA，程序员可以将部分代码从 CPU 卸载到 GPU 上执行，从而显著提升程序性能，特别是在处理大规模数据并行任务时。

CUDA 编程的核心概念是将计算任务分解为大量可以并行执行的线程。GPU 包含数千个轻量级处理核心，这些核心被组织成流式多处理器（SM），可以同时执行成千上万个线程。CUDA 程序通常由主机代码（在 CPU 上运行）和设备代码（在 GPU 上运行的内核函数）组成。内核函数定义了每个线程要执行的操作，而线程被组织成网格和线程块的层次结构。

CUDA 编程使用 C/C++ 语言的扩展，添加了特殊的关键字和函数来管理 GPU 内存和启动内核。程序员需要显式管理 CPU 和 GPU 之间的数据传输，包括分配 GPU 内存、复制数据到 GPU、执行内核函数，然后将结果复制回 CPU。CUDA 还提供了丰富的库和工具，如 cuBLAS、cuFFT、Thrust 等，帮助开发者更容易地实现高性能计算应用。

使用 cuda 编程的一般流程如下：

1. 分配 GPU 内存
2. 将数据从 CPU 传输到 GPU
3. 启动 kernel 函数进行并行计算
4. 将结果从 GPU 传回 CPU
5. 释放 GPU 资源

3 算法设计

口令猜测部分是整个算法的瓶颈，其中 generate 函数存在两个串行循环，我们可以将需要大量重复计算的任务分配给 GPU 的多个线程，每个线程独立处理一小部分数据，实现大规模并行。因此，我们将使用 cuda 实现这一过程，完成口令猜测的并行化。

4 算法实现

4.1 cuda 内核函数实现

使用内核函数 generate_guesses_kernel 处理只包含单个 segment 的密码生成。具体实现方式是：每个 GPU 线程根据其全局索引 ($blockDim.x * blockIdx.x + threadIdx.x$) 从输入的字符数组 values

中提取对应的密码字符串片段，通过逐字符循环拷贝到输出缓冲区 out 的指定位置，并在末尾添加字符串结束符""，从而实现多个密码字符串的并行生成，其中每个线程独立处理一个完整的密码字符串。

```
__global__ void generate_guesses_kernel(const char* values, int value_len, int
    value_count, char* out) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < value_count) {
        // 拷贝value到输出
        for (int j = 0; j < value_len; ++j)
            out[idx * MAX_LEN + j] = values[idx * value_len + j];
        out[idx * MAX_LEN + value_len] = '\0';
    }
}
```

使用内核函数 generate_guesses_with_prefix_kernel 处理包含处理包含多个 segment 的复合密码生成。具体实现方式是：每个 GPU 线程根据其全局索引获取一个独立的任务，先将共享的前缀字符串 (prefix) 拷贝到输出缓冲区的对应位置，然后将该线程负责的特定值 (values[idx]) 紧接着拼接在前缀之后，最后添加字符串结束符，从而实现多个" 前缀 + 后缀" 组合密码的并行生成，用于多段密码结构的高效批量生成。

```
__global__ void generate_guesses_with_prefix_kernel(
    const char* prefix, int prefix_len,
    const char* values, int value_len,
    int value_count, char* out)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < value_count) {
        // 拼接prefix
        for (int i = 0; i < prefix_len; ++i)
            out[idx * MAX_LEN + i] = prefix[i];
        // 拼接value
        for (int j = 0; j < value_len; ++j)
            out[idx * MAX_LEN + prefix_len + j] = values[idx * value_len + j];
        // 结尾符
        out[idx * MAX_LEN + prefix_len + value_len] = '\0';
    }
}
```

4.2 generate 函数的实现

在 Generate 函数中，CUDA 实现的具体流程如下：

数据准备阶段：首先将所有待处理的 value 打包成连续内存 (h_values)，便于 GPU 高效访问；然后进行内存复用检查，只在需要时分配或扩容 GPU 内存 (d_values, d_out, d_prefix)。

```
//单段
        std::vector<char> h_values(value_count * value_len);
        for (int i = 0; i < value_count; ++i)
            memcpy(&h_values[i * value_len], a->ordered_values[i].c_str(), value_len);
```

```
        // 内存复用：只在需要时分配或扩容
        size_t values_size = value_count * value_len;
        size_t out_size = value_count * MAX_LEN;
        if (!d_values || last_values_size < values_size) {
            if (d_values) cudaFree(d_values);
            cudaMalloc(&d_values, values_size);
            last_values_size = values_size;
        }
        if (!d_out || last_out_size < out_size) {
            if (d_out) cudaFree(d_out);
            cudaMalloc(&d_out, out_size);
            last_out_size = out_size;
        }

//多段
    std::vector<char> h_values(value_count * value_len);
    for (int i = 0; i < value_count; ++i)
        memcpy(&h_values[i * value_len], a->ordered_values[i].c_str(), value_len);

    size_t prefix_size = guess.size();
    size_t values_size = value_count * value_len;
    size_t out_size = value_count * MAX_LEN;
    if (!d_prefix || last_prefix_size < prefix_size) {
        if (d_prefix) cudaFree(d_prefix);
        cudaMalloc(&d_prefix, prefix_size);
        last_prefix_size = prefix_size;
    }
    if (!d_values || last_values_size < values_size) {
        if (d_values) cudaFree(d_values);
        cudaMalloc(&d_values, values_size);
        last_values_size = values_size;
    }
    if (!d_out || last_out_size < out_size) {
        if (d_out) cudaFree(d_out);
        cudaMalloc(&d_out, out_size);
        last_out_size = out_size;
    }
```

异步执行阶段：使用 cudaMemcpyAsync 异步将数据传输到 GPU，配置合适的线程块大小和网格大小，通过 CUDA Stream 异步启动对应的内核函数 (generate_guesses_kernel 或 generate_guesses_with_prefix_kernel) 进行并行密码生成。

```
//单段
    cudaMemcpyAsync(d_values, h_values.data(), values_size,
        cudaMemcpyHostToDevice,stream);

    // 启动kernel
    int threads = 256;
```

```
    int blocks = (value_count + threads - 1) / threads;

    generate_guesses_kernel<<<blocks, threads, 0, stream>>>(d_values, value_len,
        value_count, d_out);

//多段
    cudaMemcpyAsync(d_values, h_values.data(), values_size, cudaMemcpyHostToDevice,
        stream);
    cudaMemcpyAsync(d_prefix, guess.data(), prefix_size, cudaMemcpyHostToDevice,
        stream);

    // 启动kernel
    int threads = 256;
    int blocks = (value_count + threads - 1) / threads;

    generate_guesses_with_prefix_kernel<<<blocks, threads, 0, stream>>>(
        d_prefix, prefix_size, d_values, value_len, value_count, d_out);
```

结果回收阶段：异步将生成结果从 GPU 传回主机内存，通过 cudaStreamSynchronize 等待所有操作完成，最后将结果逐一添加到 guesses 容器中，实现了从串行字符串拼接到并行批量生成的性能飞跃。

整个过程通过流水线式的异步执行最大化了 GPU 利用率，避免了 CPU 等待 GPU 计算的空闲时间。

```
    //  拷贝结果回主机
    std::vector<char> h_out(out_size);
    cudaMemcpyAsync(h_out.data(), d_out, out_size, cudaMemcpyDeviceToHost, stream);
    cudaStreamSynchronize(stream);

    //  插入guesses
    for (int i = 0; i < value_count; ++i)
        guesses.emplace_back(&h_out[i * MAX_LEN]);
    total_guesses += value_count;
```

5 结果分析

使用下面两个编译指令分别编译串行和 cuda 并行程序：

- 串行实现：g++ main.cpp train.cpp guessing.cpp md5.cpp -o test.exe -O2
- 并行实现：nvcc -O2 main.cpp train.cpp guessing.cu md5.cpp guessing_cuda.cu -o test.exe

进行测试后，我们得到如下表的结果，我们可以看到使用 cuda 优化的程序的 guess time 在 O2 优化下为 0.638s，明显大于串行的 0.435s。这表明在不同的优化条件下，使用了 cuda 进行 gpu 编程的程序，在 guess time 方面明显差于串行实现，出现了负优化。

针对出现负优化的现象，我们使用 nvprof 对当前处于 O2 优化的 cuda 程序进行性能测试，结果如下图。根据这个 nvprof 性能分析报告，可以看出 GPU 执行时间主要分布在：CUDA 内存拷贝

guess time	串行实现	cuda 优化
无优化	7.528	8.569
O1 优化	0.432	0.757
O2 优化	0.435	0.638

表 1: 性能测试结果 (单位:s)

从设备到主机 (DtoH) 耗时 95.382ms 占 84.52%，而单纯的密码生成内核仅耗时 683.48 s 占 0.61%；API 调用方面，cudaMemcpyAsync 占用 65.35% 的时间达 345.90ms，cudaStreamCreate 占 32.95% 达 174.38ms。这个性能剖析显示了负优化的一个重要原因：实际 GPU 计算时间 (内核执行) 仅占总时间的约 10%，而数据传输时间占据了近 90%，表明当前的密码生成任务，被频繁的 CPU-GPU 数据传输拖累，加上 516 次高频的小批次内存拷贝调用，使得传输开销完全压倒了并行计算的收益。进一步的优化：1. 应该在于降低数据传输的开销，减少内存传输频率，尽量在 GPU 上保留数据，减少 CPU GPU 传输，可以使用 GPU 内存池，避免频繁分配/释放；2. 增大批处理规模，不要每个 PT 都调用 GPU，而是在一定数量再处理。

```
==1451334== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   84.52%  95.302ms       516  184.69us     928ns  1.8825ms  [CUDA memcpy DtoH]
                    9.43%  10.635ms       494  21.527us  2.4000us  2.3321ms  generate_guesses_with_prefix_
kernel(char const *, int, char const *, int, int, char*)
                    5.45%  6.1405ms      1010  6.0790us     736ns  99.647us  [CUDA memcpy HtoD]
                    0.61%  683.48us        22  31.067us  6.2400us  73.215us  generate_guesses_kernel(char
const *, int, int, char*)
      API calls:   65.35%  345.90ms      1526  226.67us  2.3700us  3.2973ms  cudaMemcpyAsync
                   32.95%  174.38ms         1  174.38ms  174.38ms  174.38ms  cudaStreamCreate
                    0.91%  4.8119ms       516  9.3250us  6.2750us  32.629us  cudaLaunchKernel
                    0.47%  2.4769ms       516  4.8000us  3.9670us  8.6130us  cudaStreamSynchronize
                    0.17%  916.52us         8  114.57us  7.4470us  329.52us  cudaMalloc
                    0.09%  489.75us         5  97.949us  18.278us  404.43us  cudaFree
                    0.06%  295.36us       101  2.9240us     131ns  119.44us  cuDeviceGetAttribute
                    0.00%  5.8140us         1  5.8140us  5.8140us  5.8140us  cuDeviceGetName
                    0.00%  2.0240us         3     674ns     272ns  1.4720us  cuDeviceGetCount
                    0.00%     500ns         2     250ns     164ns     336ns  cuDeviceGet
                    0.00%     319ns         1     319ns     319ns     319ns  cuDeviceTotalMem
                    0.00%     250ns         1     250ns     250ns     250ns  cuDeviceGetUuid
```

图 5.1: 测试结果

6 进阶实现

基础的 gpu 并行实现的是对于一个 PT 的猜测生成的不同部分进行并行实现，在进阶要求中我们实现可以尝试往 gpu 上一次装载多个 PT 进行生成。基本的实现思路为在 CPU 端维护按概率排序的 PT 优先队列，每次从队列头部动态提取多个 PT 组成处理批次，通过计算每个 PT 得到可能组合数，并累加控制总组合数不超过预设阈值以避免内存溢出；将对应的 PT 批量传输至 GPU 后，由 CUDA 内核实施两级并行化处理——每个 PT 分配一个线程块，块内各线程并行生成该 PT 的不同组合变体，通过预计算的字符段偏移量表实现高效字符串拼接，最终将生成的猜测口令写回统一输出缓冲区。

6.1 具体实现

首先在 gpu_structs.h 中定义了 GPU 需要使用的数据结构，这些结构体是主机 (CPU) 和设备 (GPU) 共用的，用于在两者之间传递数据。

```cpp
struct PT_GPU {
    int seg_num;
    int curr_indices[MAX_SEG];
    int max_indices[MAX_SEG];
    int types[MAX_SEG];
    int type_indices[MAX_SEG];
};

struct Segment_GPU {
    int value_num;
    int value_offset;
};
```

然后是 CUDA 内核函数 batch_generate_kernel 的实现。它实现了高效的并行密码猜测生成，它通过将密码模板 (PT) 分解为多个字符段，利用 GPU 的并行计算能力批量生成所有可能的组合。每个 CUDA 线程处理一个独特的密码组合：首先根据全局索引确定所属的密码模板，然后解码出各字符段的索引位置，接着拼接对应的字符值形成完整密码，最后将结果写入输出缓冲区。

```cpp
__global__ void batch_generate_kernel(
    PT_GPU* pts, Segment_GPU* letters, Segment_GPU* digits, Segment_GPU* symbols,
    char* all_values,
    char* output, int* output_offsets, int* pt_offsets, int batch_size, int
        total_combos)
{
    // 计算全局索引
    int global_idx = blockIdx.x * blockDim.x + threadIdx.x;

    // 确定属于哪个PT
    int pt_idx = 0;
    while (pt_idx < batch_size && global_idx >= pt_offsets[pt_idx + 1]) ++pt_idx;

    // 解码组合索引
    int combo_idx = global_idx - pt_offsets[pt_idx];
    PT_GPU pt = pts[pt_idx];

    // 解码当前索引
    int curr_indices[MAX_SEG];
    int div = combo_idx;
    for (int i = pt.seg_num - 1; i >= 0; --i) {
        curr_indices[i] = div % pt.max_indices[i];
        div /= pt.max_indices[i];
    }

    // 拼接猜测字符串
    char guess[128] = {0};
    int guess_len = 0;
    for (int i = 0; i < pt.seg_num; ++i) {
        // 根据类型获取对应的字符串段
```

```
        const char* seg_val = nullptr;
        if (type == 1) seg_val = get_value_ptr(all_values,
            letters[type_idx].value_offset, value_idx);
        else if (type == 2) seg_val = get_value_ptr(all_values,
            digits[type_idx].value_offset, value_idx);
        else if (type == 3) seg_val = get_value_ptr(all_values,
            symbols[type_idx].value_offset, value_idx);

        // 拼接字符串
        if (seg_val) {
            int l = 0;
            while (seg_val[l] && guess_len < 63) {
                guess[guess_len++] = seg_val[l++];
            }
        }
    }

    // 写入输出
    int offset = output_offsets[global_idx];
    for (int i = 0; i < guess_len; ++i) {
        output[offset + i] = guess[i];
    }
    output[offset + guess_len] = '\0';
}
```

接着是在主机端实现的部分。在 guessing_new.cpp 中的 BatchGenerateGPU 函数处理主机与设备之间的数据传输和内核调用，主要的代码如下：

```

void PriorityQueue::BatchGenerateGPU(const vector<PT>& batch){

    ......

    \\打包PT数据
    PT_GPU* h_pts = new PT_GPU[batch_size];
    for (int i = 0; i < batch_size; ++i) {
        const PT& pt = batch[i];
        // 填充PT_GPU结构体
        h_pts[i].seg_num = pt.content.size();
        // 填充其他字段...
    }

    \\打包segment数据
    vector<char> all_values;
    vector<Segment_GPU> h_letters(letters_num);
    vector<Segment_GPU> h_digits(digits_num);
    vector<Segment_GPU> h_symbols(symbols_num);

    // 将字符串数据连续存储到all_values中
```

```
    for (const auto& val : m.letters[i].ordered_values) {
        all_values.insert(all_values.end(), val.begin(), val.end());
        all_values.push_back('\0');
    }

    \\计算组合偏移
    std::vector<int> pt_offsets(batch_size + 1, 0);
    int total_combos = 0;
    for (int i = 0; i < batch_size; ++i) {
        int combos = 1;
        for (int j = 0; j < h_pts[i].seg_num; ++j) {
            combos *= h_pts[i].max_indices[j];
        }
        pt_offsets[i] = total_combos;
        total_combos += combos;
    }
    pt_offsets[batch_size] = total_combos;

    \\分配设备内存并拷贝数据
    cudaMalloc(&d_pts, batch_size * sizeof(PT_GPU));
    cudaMemcpy(d_pts, h_pts, batch_size * sizeof(PT_GPU), cudaMemcpyHostToDevice);
    ......

    \\准备输出空间
    cudaMalloc(&d_output, total_combos * max_guess_len);
    cudaMalloc(&d_output_offsets, total_combos * sizeof(int));

    \\启动内核
    launch_batch_generate_kernel(d_pts, d_letters, d_digits, d_symbols,
        d_all_values, d_output, d_output_offsets, d_pt_offsets,
        batch_size, total_combos, threads, blocks);

    \\拷贝
    char* h_output = new char[total_combos * max_guess_len];
    cudaMemcpy(h_output, d_output, total_combos * max_guess_len,
        cudaMemcpyDeviceToHost);

    // 将结果添加到猜测列表中
    for (int i = 0; i < total_combos; ++i) {
        guesses.emplace_back(h_output + i * max_guess_len);
    }
    ...
}
```

然后是 PopBatch 的实现，能够高效地从优先队列中提取并处理多个密码模板，进行后续的处理。它从优先队列 priority 中一次性取出最多 batch_size 个密码模板 (PT)，同时计算每个 PT 可能生成的密码组合数（各段最大索引的乘积），并累加总组合数 total_combos。设置上限 MAX_COMBOS=100000 防止单次处理量过大导致内存问题，最后将收集好的批次通过 BatchGenerateGPU() 提交给 GPU 并

行处理，充分利用 GPU 的并行计算能力批量生成密码。我们可以在主程序中将 popnext() 函数替换为 PopBatch，实现一次性多个 PT 的并行。

```
void PriorityQueue::PopBatch(int batch_size) {
    std::vector<PT> batch;
    int total_combos = 0;
    const int MAX_COMBOS = 100000;

    for (int i = 0; i < batch_size && !priority.empty(); ++i) {
        PT& pt = priority.front();
        int pt_combos = 1;
        for (int j = 0; j < pt.content.size(); ++j) {
            pt_combos *= pt.max_indices[j];
        }

        if (total_combos + pt_combos > MAX_COMBOS) {
            break;  // 停止添加，避免溢出
        }

        batch.push_back(pt);
        total_combos += pt_combos;
        priority.erase(priority.begin());
    }

    if (!batch.empty()) {
        BatchGenerateGPU(batch);
    }
}
```

6.2 程序分析

在程序编写过程中，我主要遇到了两个问题。其中一个问题是同时进行多个 PT 的并行，容易出现内存溢出导致程序提前结束。修改后又出现了猜测过程卡住的问题，需要进一步的修改完善。

7 总结与反思

本次实验是通过 cuda 进行口令猜测并行化的实现，其核心思想是将原本 CPU 串行执行的、可并行的循环迁移到 GPU 上，由成百上千的线程同时处理，大幅提升处理速度。在这个过程中，CPU 主要负责数据准备和结果收集，而 GPU 负责高强度的并行计算。在使用 gpu 进行并行化的过程中，我们主要有两种思路，一种是对于每一个 pt 在 gpu 中并行处理猜测部分，另一种是通过将多个 pt 载入到 gpu 中进行并行处理。