南開大學
Nankai University

计算机学院

并行程序设计第五次实验报告

CUDA 并行实现口令猜测算法

姓名：王泽舜

学号：2310655

专业：计算机科学与技术

2025 年 7 月 2 日

目录

1 实验平台配置

1.1 服务器实验环境

本实验在如下共享服务器环境下进行：

- **操作系统**: Ubuntu 22.04 LTS (内核版本 5.15.0-91-generic)
- **处理器**: Intel Xeon Platinum 8255C @ 2.50GHz
  - 架构: x86_64
  - 核心数: 4 核/8 线程
  - 支持指令集: SSE, SSE2, SSE4.1, SSE4.2, AVX, AVX2, AVX512F, AVX512DQ, AVX512CD, AVX512BW, AVX512VL, AVX512_VNNI 等
  - L1 缓存: 256 KiB (8 实例)
  - L2 缓存: 16 MiB (4 实例)
  - L3 缓存: 35.8 MiB (1 实例)
- **内存**: 30 GiB DDR4
  - 已用: 2.7 GiB
  - 可用: 27 GiB
  - 缓存: 8.4 GiB
- **虚拟化**: KVM
- **其他**: 支持超线程，NUMA 节点数 1

说明: 服务器支持多种高级指令集，内存资源充足，适合并行计算实验。

- 代码仓库: https://github.com/wzs267/Parallel-homework
- 开发环境: Visual Studio Code + Remote SSH
- 版本控制: Git

2 实验目的

用 cuda 方法实现 PCFG 口令猜测中 generate guesses 部分的并行化，熟悉 CUDA 编程的基本方法。

3 算法原理与代码分析

3.1 串行 PCFG 口令猜测框架回顾

原始串行实现（见 `main_origin.cpp`、`guessing_origin.cpp`）的主流程为，每次取一个 pt 来 generate guesses，直到存放 pt 的优先队列为空。串行代码大致如下：

Listing 1: 串行密码生成算法核心代码

```
while (!q.priority.empty()) {
    q.PopNext();
    // q.Generate(pt) 生成guesses
    // 达到阈值时进行哈希处理
}
```

3.2 CUDA 并行基础要求设计与实现

优化思路

考虑字符串写入 `guesses` 这个过程,是否可以先将 `guesses.resize(old_size2 + pt.max_indices[pt.conte - 1] or [0])`; 得到的结果传入 `cuda_generate` 函数，避免回收结果之后还得 `emplace_back`。然后发挥 GPU 对简单运算的优势，挨个把要赋值的值写入 `guesses` 中。

但是这个过程无法实现，其一是 GPU 无法操作 string 指针，其二是即使我们采用 char** 存放，共享显存 (一般为 48kb) 也放不下 guesses 那么大的块。因此我们退而求其次，

考虑将所有后缀前缀拼接的过程放在 GPU 上进行。并在优化过程中，我们发现用二维数组存放所有的 value 会导致访存的不连续，导致性能下降，因此我们将二维数组扁平化为一维 `char*` 数组，使用 `flat_values` 存放所有的 value。GPU 端每个线程负责将一个 value 拼接前缀 (前缀是一维的，不用修改) 之后写入计算好的区间。经测试，比原始实现：传二维 `char**` 数组（230s/50w guesses）优化到（2.1s/1000w guesses）。

Listing 2: CUDA 并行口令猜测核心流程

```
// 1. 主机端将所有value拼接为连续数组
int n = pt.max_indices[pt.content.size() - 1];
char **h_values = new char*[n];
        for (int i = 0; i < n; ++i) {
            h_values[i] = new char[value_len+1];
            strcpy(h_values[i], a->ordered_values[i].c_str());
        }
// 2. 分配device内存并拷贝数据
cudaMalloc(&d_flat_values, flat_size * sizeof(char));
cudaMemcpy(d_flat_values, flat_values, flat_size * sizeof(char),
    cudaMemcpyHostToDevice);
```

```
if (prefix_len > 0) {
    cudaMalloc(&d_guess_prefix, prefix_len * sizeof(char));
    cudaMemcpy(d_guess_prefix, h_guess_prefix, prefix_len * sizeof(char),
        cudaMemcpyHostToDevice);
}
cudaMalloc(&d_result_data, n * (prefix_len + value_len + 1) * sizeof(char));
// 3. 启动核函数并行拼接
int block = 256;
int grid = (n + block - 1) / block;
generate_guesses_kernel_flat<<<grid, block>>>(d_flat_values, value_len,
    d_guess_prefix, prefix_len, d_result_data, n);
cudaDeviceSynchronize();
//核函数:
__global__ void generate_guesses_kernel_flat(const char *flat_values, int value_len,
    const char *d_guess_prefix, int prefix_len, char *result_data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        char *dst = result_data + idx * (prefix_len + value_len + 1);
        if (prefix_len > 0) {//如果是第二种情况，则将前缀也写到结果中
            for (int i = 0; i < prefix_len; ++i) dst[i] = d_guess_prefix[i];
        }
        const char *src = flat_values + idx * (value_len + 1);
        for (int i = 0; i < value_len; ++i) dst[prefix_len + i] = src[i];
        dst[prefix_len + value_len] = '\0';
    }
}
// 4. 回传结果
cudaMemcpy(result_data, d_result_data, n * (prefix_len + value_len + 1) *
    sizeof(char), cudaMemcpyDeviceToHost);
for (int i = 0; i < n; ++i) {
    guesses[offset + i].assign(result_data + i * (prefix_len + value_len + 1));
}
// 5. 释放内存
// ...
```

代码流程 cuda 函数设计:`cuda_generate_guesses(char **h_values, int value_len, char *h_guess_prefix, int prefix_len, int n, std::vector<std::string> &guesses, size_t offset)`

- `h_values`：主机端的值数组
- `value_len`：值长度
- `h_guess_prefix`：主机端的前缀，如果是第二个 for 循环就需要传入
- `prefix_len`：前缀长度
- `n`：需要拼接的猜测数量
- `guesses`：拼接结果，直接用引用把 guesses 传进去

- `offset`：结果偏移量，因为传入的 `guesses` 有之前的结果，所以需要偏移到本次 generate 开始时候的位置，故用 `size_t old_size = guesses.size();` 赋值

4 性能测试

Listing 3: 核函数代码

```
__global__ void generate_guesses_kernel_flat(const char *flat_values, int value_len,
    const char *d_guess_prefix, int prefix_len, char *result_data, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        char *dst = result_data + idx * (prefix_len + value_len + 1);
        if (prefix_len > 0) {
            for (int i = 0; i < prefix_len; ++i) dst[i] = d_guess_prefix[i];
        }
        const char *src = flat_values + idx * (value_len + 1);
        for (int i = 0; i < value_len; ++i) dst[prefix_len + i] = src[i];
        dst[prefix_len + value_len] = '\0';
    }
}
```

核函数代码流程：根据是否有前缀，计算出每个线程需要处理的值的索引，和要写入结果数组的位置，然后将对应的值和前缀起来，写入结果数组中。

4.1 测试环境配置

环境配置见 1.1 节

4.2 测试结果

测试除了原有的计时外，还测量了串行实现的两个 for 循环所用的时间，和并行实现的主机端准备时间 (包括分配内存和 memcpy)、核函数运行时间 (拼接时间)、结果收集时间 (包括将拼好的字符串写入到 guesses)。

表 1: 并行实现各阶段耗时（单位：秒）

序号	Guess time	Prepare	Kernel	Collect
1	5.02851	1.75292	0.63605	0.64209
2	5.26718	1.82522	0.78890	0.64341
3	5.46996	1.86242	0.82392	0.64446
4	4.99559	1.70134	0.69945	0.64082
5	4.99582	1.73317	0.63752	0.64270
6	4.93739	1.72491	0.64972	0.64166
7	4.99378	1.71390	0.69786	0.64068
8	5.19900	1.87338	0.67086	0.64103
9	5.22694	1.94104	0.66292	0.64073
10	4.86642	1.66311	0.66302	0.64017
均值	5.07886	1.77954	0.69383	0.64187

表 2: 串行实现各阶段耗时（单位：秒）

序号	Guess time	Serial Gen
1	1.78181	0.94192
2	1.77719	0.93763
3	1.77995	0.94056
4	1.76992	0.93223
5	1.77576	0.93904
6	1.77857	0.93766
7	1.77732	0.93877
8	1.77589	0.93726
9	1.77195	0.93622
10	1.77564	0.93602
均值	1.77590	0.93713

可见性能瓶颈大部分在于主机端分配和拷贝内存阶段，核函数运行时间占比很小。串行实现的生成时间大约为并行实现的 1/3 左右。具体到拼接和收集这两个原先也有的阶段，并行的实现 (0.69383s+0.64187s) 相比于串行的实现 (0.93713s) 也并没有明显的提升。

4.3 正确性验证

为了验证并行实现的正确性，我们用 correctness.cpp 替代 main.cpp 编译进行测试，得到的 cracked 值为 358217，和串行实现的结果大致相同，说明并行实现的结果是正确的。

5 后续优化方向分析

从实验结果来看，当前 CUDA 并行实现的性能瓶颈主要集中在主机端的内存分配与数据拷贝阶段。核函数（Kernel）本身的运行时间占比极小，说明 GPU 端的并行计算效率较高，但整体加速效果被主机端的数据准备和结果收集过程所限制。具体分析与优化方向如下：

- **减少主机端内存分配与拷贝开销**：目前每次生成都需要在主机端分配大块内存并进行多次 cudaMemcpy 操作。可以考虑：
  - 复用内存：将 flat_values、result_data 等缓冲区在多次调用中复用，避免频繁分配和释放。
  - 异步拷贝与流（cudaStream）：利用 CUDA 流和异步拷贝（cudaMemcpyAsync），实现主机端和设备端的数据传输与计算重叠，提升整体吞吐量。
- **提升核函数利用率**：虽然核函数耗时很短，但可以进一步优化：
  - 一次传入多个 PT，增大单次批量处理规模，减少核函数启动次数，提升 GPU 利用率。
- **优化数据收集阶段**：结果收集（Collect）阶段仍有一定开销，但出于如设计思路所说的，guesses 无法传入显存，目前也想不到除了直接扩容写入之外的优化方法。
- **算法层面优化**：串行实现在生成阶段的效率反而高于并行，说明部分串行逻辑本身已较为高效。可进一步分析串行与并行在数据结构、内存布局等方面的差异，寻找更适合 GPU 的算法重构方案。

综上，后续优化应重点关注主机端与 GPU 之间的数据流转效率，充分发挥 GPU 的并行计算能力，并结合异步机制、内存复用等手段，进一步缩短整体运行时间。