南開大學
Nankai University

计算机学院

并行程序设计实验报告

PCFG 口令猜测并行化选题: GPU 编程实验

姓名：陈语童

学号：2311887

专业：计算机科学与技术

2025 年 7 月 2 日

目录

1 实验环境

SSH 远程：并行智算云平台 GPU 服务器

共享 GPU 服务器和之前使用的 OpenEuler ARM 服务器个人账号无区别，共享服务器的 CUDA-nvcc 环境已配置好，可以直接开始 CUDA 编程，共享 GPU 服务器不会关机 24 小时可使用。

GPU 具体配置未提供，但足够承担千数量级的多线程并行运行。

2 问题描述

2.1 选题：PCFG 口令猜测

2.1.1 口令猜测

口令，就是俗称的“密码”。对于一定长度范围的口令，通过穷举进行暴力破解的效率是极低的。而本选题要做的，就是通过用户口令的语义和偏好规律，制定一定的策略来破解口令，并用**并行化**的方法对整个过程的各个部分进行优化与加速。

本选题中，不考虑个人信息、口令重用这些额外的信息，只考虑非定向的口令猜测。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。那么问题就变成了：

- 如何有效生成用户可能选择的口令；
- 如何将生成的口令按照降序进行排列。

在本选题中，选择使用最经典的 **PCFG**（Probabilistic Context-Free Grammar，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。

2.1.2 PCFG 串行训练

PCFG 串行训练流程关键步骤如下：

1. **内容分类**。将口令内容分为三类不同字段（segments）——Letters（字母字段）、Digits（数字字段）、Symbols（特殊字符）
2. **解构口令**。然后将实际每条口令内容用这三种字段类型解构其组成，形成一系列的口令 preterminal（类似于 pattern）。
3. **统计频率**。对不同的 preterminal，以及每个 preterminal 中的每个 segment 的不同值均需统计出现频次。

经过上面三步，PCFG 模型生成。

2.1.3 PCFG 串行猜测生成

猜测生成的本质要求，是构建出一个出现概率降序排列的口令**优先队列**，对其动态更新和从中取猜测值。具体生成策略较为繁琐，此处只做概括：

1. **初始化**。所有 preterminal 中出现概率最高的口令组合先入队列，preterminal 本身的概率越高越优先入队列。同时，赋给一个初始 pivot 值为 0.

2. **出列**。只要队列有口令，就从先入队列的开始取，取到的即作为当前口令猜测值。

3. **变异**。取出的口令不立即放回，而是根据 pivot 值，依次单项更新大于 pivot 值的 segment 值，依然是概率降序入队列，并且新入列的要设置 pivot 值到发生变异的 segment.

4. **遍历**。接着向后遍历和动态更新即可。

通过这种策略生成的优先队列可尽可能保证口令是不重复地以概率降序排列的，从而提高遍历查找效率和命中率。

2.1.4 PCFG 猜测生成并行化 *

这是本选题的重点和最终目标。

PCFG 猜测生成过程似乎比较复杂，难以进行并行化。并行化的最大阻碍，是按照概率降序生成口令这一过程。但 PCFG 往往用于没有猜测次数限制的场景中（例如哈希破解），实际应用中并不需要严格按照降序生成口令。

由此产生的并行化思路很显然：一次取出多个 preterminal，或是一次为某一 segment 分配多个 value 等。

选题代码框架采用的并行化方案如图示：

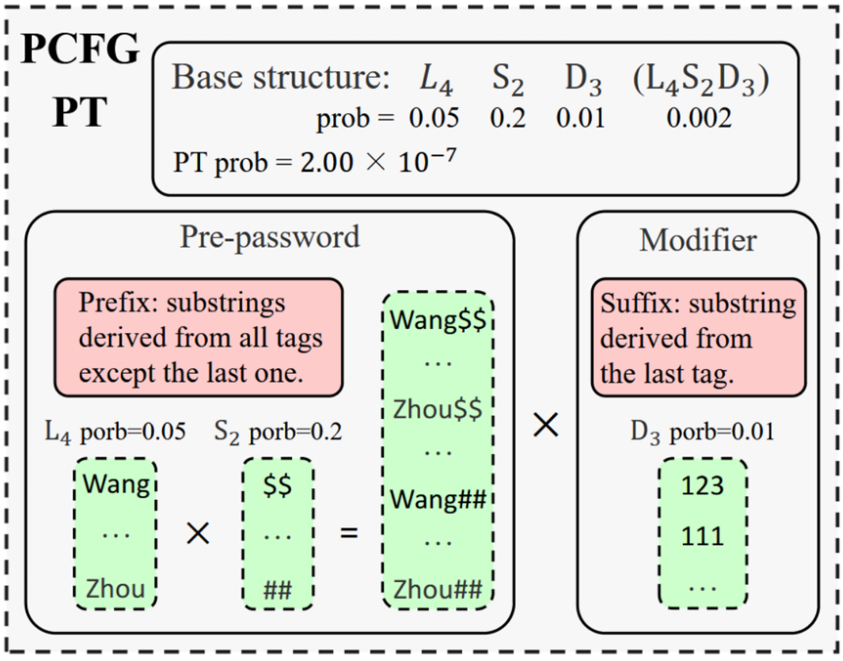

图 2.1: 本代码框架采用的并行化方案

这种方案和原始 PCFG 的区别在于，其对 preterminal 中最后一个 segment 不进行初始化和改变。在 preterminal 从优先队列中弹出的时候，直接一次性将所有可能的 value 赋予这个 preterminal. 不难看出，这个过程是可以并行进行的。

本选题的代码框架，已经将这个并行算法用串行的方式进行了实现，实际实验只需将其用并行编程的方式，变成一个并行程序，即可完成 PCFG 的并行化。

2.2 本次实验：GPU-CUDA 编程实验

本次实验在 GPU 服务器上进行，使用到 CUDA（Compute Unified Device Architecture，统一计算设备架构）——NVIDIA 公司推出的并行计算平台和编程模型，允许使用 GPU（图形处理器）进行高性能通用计算。

CUDA 编程的基本模式围绕 “主机（Host）-设备（Device）协作” 和 “大规模并行线程” 展开，其核心思想是将计算任务分配到 GPU 的数千个核心上并行执行。主机，一般由 CPU 实现，负责控制逻辑、内存分配、启动 GPU 核函数（Kernel）、数据拷贝（CPU <-> GPU）；而设备，一般由 GPU 实现，负责执行并行计算任务（由核函数定义），每个线程处理数据的一部分。

基本的 CUDA 编程模式：

- 分配 GPU 内存。使用 `cudaMalloc` 在 GPU 上分配显存；使用 `cudaMemcpy` 将数据从 CPU 内存拷贝到 GPU 显存。
- 定义核函数。核函数是运行在 GPU 上的并行计算函数，用 `__global__` 修饰，一般每个线程执行相同的核函数代码，但处理不同的数据。
- 启动核函数。使用 `<<<gridDim, blockDim>>>` 指定线程布局，其中前一参数表示启动多少个 Block（线程块），后一参数表示每个 Block 中有多少线程。
- 拷贝结果回 CPU。再使用 `cudaMemcpy` 将 GPU 计算结果拷贝回 CPU，并用相关指令释放 GPU 内存。

注意 GPU 中需要将`.cpp` 文件改为`.cu` 文件才能够进行 CUDA 的编译和运行，以及要使用 nvcc 编译器相关指令进行编译。

2.2.1 PCFG 猜测循环的并行化

PCFG 算法中生成猜测部分的串行算法通过循环来实现：

```
// ... ... ... ...
// Multi-thread TODO:
// 这个 for 循环就是你需要进行并行化的主要部分了，特别是在多线程 &GPU 编程任务中
// 可以看到，这个循环本质上就是把模型中一个 segment 的所有 value，赋值到 PT 中，形成一系列新的猜测
// 这个过程是可以高度并行化的
for (int i = 0; i < pt.max_indices[0]; i += 1)
{
    string guess = a->ordered_values[i];
    guesses.emplace_back(guess);
    total_guesses += 1;
}
// ... ... ... ...
// Multi-thread TODO:
// 这个 for 循环就是你需要进行并行化的主要部分了，特别是在多线程 &GPU 编程任务中
// 可以看到，这个循环本质上就是把模型中一个 segment 的所有 value，赋值到 PT 中，形成一系列新的猜测
// 这个过程是可以高度并行化的
for (int i = 0; i < pt.max_indices[pt.content.size() - 1]; i += 1)
{
    string temp = guess + a->ordered_values[i];
```

```
    guesses.emplace_back(temp);
    total_guesses += 1;
}
// ... ... ... ...
```

本此实验的主要目标是，通过 CUDA 编程 的方式将以上两个循环并行优化（与上次 MPI 编程方法优化和上上次 pthread/OpenMP 进行的优化本质是一样的）。

3 基础实验

基础要求： 将猜测生成过程用 GPU-CUDA 编程 进行多线程并行化。具体要求如下：

- 保证并行化后的猜测过程“相对正确”。口令猜测并行化会在一定程度上牺牲口令概率的“颗粒度”，但总体上口令的生成结果基本不会有差别。检查正确性的方法是：在主函数中新增给定代码检查口令破解数，需要保证并行算法破解数量在 数量级 上和串行算法保持一致。
- 不要求真正实现相对串行算法的加速，但是仍需要给出实验数据，并且分析为什么没有能够实现加速。

3.1 GPU-CUDA 的口令猜测并行化实现

在 [PCFG.h] 中添加相应的优先队列成员函数 `CUDAGenerate` 后，在 [guessing.cu] 中进行实现。

以下展示的是进行一定优化后的最终实现。在最初写成最简逻辑版本的时候进行测试，发现效率收益非常低（运行时间超过 2s），分析后发现最主要的两个瓶颈在于：

1. 每次处理一个 PT 都需要调用 `cudaMemcpy` 从 CPU 上拷贝数据到 GPU，这是一个非常消耗时间的过程（相较于 PT 本身的口令猜测生成过程）；
2. 每次处理一个 PT 也都需要调用 `cudaMalloc` 来分配相应的显存，当 PT 非常多的时候，这也是一个非常大的开销。

因此，在展示 `CUDAGenerate` 这一主要函数逻辑实现之前，对于处理上面两个效率瓶颈各自相应的策略进行陈述。

对于频繁调用 `cudaMemcpy` 的需求，解决方案是一次性将所有 PT 相关信息从 CPU 传递到 GPU。在 [PCFG.h] 添加优先新的队列成员函数以及相应的指针或参数：

```
// 将所有 ordered_values（字母、数字、符号）加载到 GPU，并建立偏移索引
void LoadAllOrderedValuesToGPU();
// 释放 GPU 上用于 ordered_values 与偏移索引的缓冲区
void FreeGlobalBuffers();
// 存储所有字母类段的 ordered_values 拼接结果（位于 GPU 显存中）
char *d_letters_all = nullptr;
// 存储所有数字类段的 ordered_values 拼接结果（位于 GPU 显存中）
char *d_digits_all = nullptr;
// 存储所有符号类段的 ordered_values 拼接结果（位于 GPU 显存中）
char *d_symbols_all = nullptr;
// 每个字母段在 d_letters_all 中的起始偏移（GPU 端缓冲区）
int *d_letter_offsets = nullptr;
// 每个数字段在 d_digits_all 中的起始偏移（GPU 端缓冲区）
```

```
int *d_digit_offsets = nullptr;
// 每个符号段在 d_symbols_all 中的起始偏移（GPU 端缓冲区）
int *d_symbol_offsets = nullptr;
// 所有字母段中候选值（ordered_values）的总数量，用于分配 d_letters_all
int total_letter_count = 0;
// 所有数字段中候选值的总数量
int total_digit_count = 0;
// 所有符号段中候选值的总数量
int total_symbol_count = 0;
// 每个字母段的起始偏移（Host 端缓存，便于 CPU 访问）
std::vector<int> h_letter_offsets_gpu;
// 每个数字段的起始偏移（Host 端缓存）
std::vector<int> h_digit_offsets_gpu;
// 每个符号段的起始偏移（Host 端缓存）
std::vector<int> h_symbol_offsets_gpu;
```

相应地，在 [guessing.cu] 中实现两个成员函数。受篇幅限制，此处代码略。

对于频繁调用 cudaMalloc 的需求，解决方案是使用初始化的方法一次性分配足够的显存缓冲区，而后使用相应指针访问，并在程序最后清空。在 [PCFG.h] 添加优先新的队列成员函数以及相应的指针或参数：

```
// 初始化 CUDA 所需的显存缓冲区（在使用前调用）
void InitCudaBuffers();
// 释放 CUDA 显存缓冲区（程序结束或不再使用时调用）
void FreeCudaBuffers();
// GPU 端用于存储所有 ordered_values 拼接后的大段值池（如所有 L6、D1 等的字符串拼接）
char *d_values = nullptr;
// GPU 端输出缓冲区，每个线程生成的结果字符串写入这里，后续从 GPU 拷贝回 host
char *d_output = nullptr;
// GPU 端用于存储 prefix 前缀字符串（仅多段 PT 生成时使用）
char *d_prefix = nullptr;
// 当前分配的 GPU 输出缓冲区容量（单位：条数），用于判断是否需要分批处理
int d_capacity = 0;
```

相应地，在 [guessing.cu] 中实现两个成员函数。受篇幅限制，此处代码略。

上面额外添加的几个成员函数都需要在 [main.cu] 中进行调用，才能达到优化的效果，具体代码此处略。

接下来实现主要的 CUDA 方法生成口令猜测逻辑。首先，对 segment 的处理部分代码不需要作改动，保留即可。此后代码需要作修改，编写成适合于 GPU 运行的模式。CUDAGenerate 主要逻辑部分伪代码如下所示，关键处作注释解析：

Algorithm CUDAGenerate

```
function CUDAGenerate(pt)
    CalProb(pt)
    if pt.content.size = 1 then
        /* 此前代码不变 */
        /* 根据 segment 类型，在统一传输的 GPU 缓存池中找到属于对应类型的那一部分 */
```

base_offset $\leftarrow \begin{cases} \textbf{h\_letter\_offsets\_gpu[seg\_idx]} & \text{if } seg.type = 1 \\ \textbf{h\_digit\_offsets\_gpu[seg\_idx]} & \text{if } seg.type = 2 \\ \textbf{h\_symbol\_offsets\_gpu[seg\_idx]} & \text{otherwise} \end{cases}$

if N $<$ 100000 **then** ▷ 任务量足够小，不用 CUDA 的 GPU 处理，串行算法就足够
  for $i = 0$ to $N - 1$ **do**
    guesses.append(a.`ordered_values`[i])
  end for
  total_guesses += N
  return
end if

/* 设置块数和每个块的线程数，并启动核函数 */
blockSize $\leftarrow$ 256
numBlocks $\leftarrow \lceil N/256 \rceil$
Launch GENERATE_KERNEL_INDEXED(d_pool, base_offset, N, d_output, MAX_STR_LEN) ▷ 启动核函数
CUDADEVICESYNCHRONIZE ▷ 线程同步阻塞

/* 将结果从 GPU 拷贝回 Host 并存储猜测 */
h_output $\leftarrow$ new char[N][MAX_STR_LEN]
CUDAMEMCPY(h_output $\leftarrow$ d_output)
for $i = 0$ **to** N - 1 **do**
  guesses.push_back(h_output[i])
end for
total_guesses += N
delete **h_output**
else
  /* 多 segment 处理类似，此处略 */
end if
end function

其中核函数是各个线程处理 PT 生成口令猜测的核心，需要单独实现：

```
__global__ void generate_kernel_indexed(char *d_values_all, int base_offset, int value_count,
↪  char *d_output, int max_len) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= value_count) return;
    char *src = d_values_all + (base_offset + idx) * max_len;
    char *dst = d_output + idx * max_len;
    for (int i = 0; i < max_len; i += 16) {
        int4 data = *((int4 *)(src + i));
        *((int4 *)(dst + i)) = data;
    }
}
```

多 segment 的核函数，还需要传入 prefix 的相关信息，其他逻辑基本相同，代码略。

CUDA 方法实现完整代码见 GitHub 仓库：[guessing.cu]
main 函数的完整修改见 GitHub 仓库：[main.cu]

3.2 正确性检验

测试正确性的代码和原理与前两次实验是完全一致的，此处不再一一赘述。多次提交测试，结果稳定如下所示：

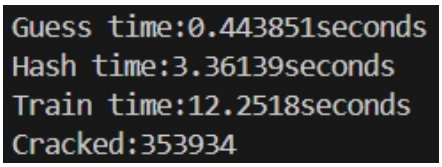

图 3.2: CUDA 方法下的破解结果

CUDA 方法：Cracked = 353934.
参考 上上次实验（Lab3-pthread&OpenMP）报告 ，串行算法的 Cracked = 358217.
可以看到 CUDA 实现的算法破解数量很接近原先的串行算法，说明原理是正确的。

3.3 加速效果测试

进行多次实验，测试 CUDA 方法的加速效果。口令猜测生成用时结果如下表所示（多组实验，剔除最高和最低，CUDA 最快结果如图 3.2所示；CUDA 服务器性能较之前更佳，串行方法也重新测了，为了做到严谨的对照）：

no.	1	2	3	4	5	**avg.**
serial	0.413984	0.420253	0.412033	0.418992	0.41605	**0.41626**
CUDA	0.45512	0.449825	0.443851	0.451643	0.456792	**0.45144**

表 1: 串行和 MPI 优化下的猜测生成用时（单位：秒）

根据以上数据依次求得 CUDA 在猜测生成上的**加速比**：

$$\text{Speedup(CUDA)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{CUDA})} = \frac{0.41626}{0.45144} \approx 0.92 < 1$$

可见在目前的已经作了比较关键的优化后的 CUDA 方法，依然没有实现相对于串行的加速。分析其可能的原因：

- 仍存在需要每次 PT 都调用的 `cudaMemcpy`，这仍然是极大的额外开销。
- `cudaDeviceSynchronize()` 目前必须使用，而强制阻塞会剥夺并发机会，如果其中某一线程进行时间较长，拖延了整个进程的进展。
- 本质上，GPU/CUDA 应该用于处理本身计算量很大的任务，而实际上，这里 GPU 各个线程中只是处理了简单的字符串连接、甚至只是拷贝，这种简单工作用 GPU 来处理，启动等环节消耗的额外时间是会导致得不偿失的。

4 进阶实验

4.1 尝试 PT 层面的并行

尝试将多个 PT 传给 GPU 同时处理。大体的思路是：

1. 一次取出多个 PT；
2. 统计总猜测数；
3. 构造 Host 端辅助数据，这是遇到的一大困难，GPU 无法批量地直接接受多个 PT，因为 CPU 支持的 `std::vector` 等数据结构，GPU 是没有的，无法直接调用处理；
4. 将这些数据复制传给 GPU；
5. 在 GPU 上调用核函数生成猜测串，这里重新写了一个专门用于多 PT 生成的核函数；
6. 将生成的猜测从 GPU 拷贝回 CPU 并保存；
7. 释放资源、移除此次处理的 PT 等后续善后操作。

分别编写了批量处理 PT 函数 `CUDAPopNext` 和专用核函数 `generate_kernel_multi_pt`，受篇幅限制不展示代码，见相应的 GitHub 仓库文件。

显然，这样朴素思路的实现，得到的结果依然是无法加速，甚至逊于基础实验中实现的 `CUDAGenerate`，因为这其中对于 PT 各种数据的 GPU 版本构造和传输是又一大额外开销，势必导致加速效果被掩盖。

然而，运行后还是能够证明此实现在原理上的正确性：

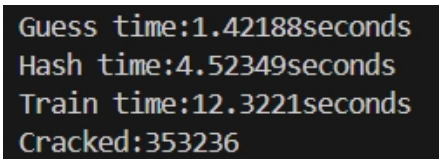

图 4.3: PT 层面 CUDA 并行化的破解结果

CUDA 方法的 PT 层面并行：Cracked = 353236.
已知串行算法的 Cracked = 358217.
新实现的方法的口令破解数接近原串行算法，说明原理上是没有问题的。
CUDA 方法实现 PT 层面并行的完整代码见 GitHub 仓库：[guessing.cu]

5 总结思考

GPU-CUDA 算法的理解、设计和编写相对 CPU 较为复杂，想要实现加速非常困难，后期有时间或思路将尝试继续优化加速本次实验实现的一系列算法。

本次实验所有代码均已上传至 GitHub：{并行智算云平台 GPU 服务器中的 CUDA 实现}