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 编程实验

赵熙

年级：2022 级

专业：计算机科学与技术

2025 年 7 月 1 日

目录

一、 实验要求

(一) 基础要求

在本次实验中，我的目标是将之前多线程实验中的两个循环（即 PT 内部的口令生成）通过 GPU 进行并行化，以加速口令生成过程。具体要求包括：

- **PT 内部口令生成的并行化：**将原本在 CPU 上串行执行的两个循环（即生成每个 PT 的多个口令）移至 GPU 进行并行计算，从而显著提升计算速度。

(二) 进阶要求

1. **MPI 实验中的并行化：**如果在之前的 MPI 实验中尝试了 PT 层面的并行（例如一次性取出多个 PT 并进行并行生成），那么在这次实验中，可以将多个 PT 一起装载到 GPU 上进行并行生成。这样可以充分利用 GPU 的并行计算能力，同时减少因多次上传数据到 GPU 产生的额外开销。

2. **避免 CPU 等待时间的浪费：**在将任务从 CPU 传输到 GPU 进行计算时，CPU 可能会进入“忙等待”状态，浪费计算资源。为了解决这个问题，可以考虑在 GPU 计算期间利用 CPU 的计算能力进行其他并行任务。这种方法可以有效“压榨”CPU 的计算资源，提高整体计算效率。

3. **根据不同 PT 的计算量调整计算资源分配：**不同的 PT 所能生成的口令数量差异很大，这意味着每个 PT 的计算量存在差异。可以根据不同 PT 的计算量大小，在 CPU 和 GPU 之间进行资源调整。例如，对于计算量较小的 PT，可以在 CPU 上直接处理，而对于计算量较大的 PT，可以将其上传到 GPU 进行并行计算，以避免 GPU 上传的开销过大导致效率下降。

4. **优化 GPU 任务分配：**当一次性将多个 PT 传送到 GPU 时，如何根据不同 PT 的计算需求对任务进行合理划分是一个关键点。可以尝试根据每个 PT 的计算量和 GPU 的负载情况，动态调整每次上传到 GPU 的 PT 数量，避免固定数量的 PT 传送导致不必要的性能损失。

二、 实验环境

在本次实验中，使用的实验环境配置如下表所示：

项	配置
操作系统	Ubuntu 5.15.0-91-generic
内核版本	5.15.0-91-generic
CPU 型号	Intel Xeon Platinum 8255C @ 2.50GHz
CPU 核心数	4 核心 (每核心支持 2 线程，共 8 逻辑 CPU)
内存总量	30GB
已用内存	2.5GB
可用内存	27GB
显卡 1	Cirrus Logic GD 5446 (集成显卡)
显卡 2	NVIDIA Tesla T4 (专用显卡)
虚拟化技术	KVM

表 1: 实验环境配置

三、 算法概述

在本次实验中，算法的核心是通过 CUDA 并行化加速口令生成过程。原始的口令生成算法是串行的，生成每个口令需要遍历所有可能的 segment 值，并对这些值进行组合生成最终的口令。随着口令段数和每个段的可能值数目的增加，计算量急剧增大，因此该过程的效率较低。

为了解决这个问题，我使用 CUDA 将口令生成的关键循环并行化，使得每个线程负责生成一个口令，从而大大加快了口令生成过程。通过利用 GPU 的并行计算能力，我能够在短时间内生成大量口令候选，并按概率排序。

(一) CUDA 加速的设计思路

在 CUDA 编程中，我的设计思路是将每个口令的生成过程分配到多个线程进行并行处理，每个线程处理一个口令生成任务。具体来说，CUDA 的加速部分主要包括以下几个方面：

- **每个线程处理一个口令生成：** 传统的算法是通过一个循环逐个处理每个口令，而 CUDA 允许我将每个口令的生成过程分配到独立的线程进行并行计算。这种方式能够有效提升计算效率，特别是在生成多个口令时。
- **双循环优化：** 生成口令的过程中，通常有两个关键循环：一个是处理单个 segment 的情况，另一个是处理多个 segment 组合的情况。通过 CUDA 的并行化设计，我将这两个循环的计算任务拆分到不同的线程进行处理，避免了串行计算的瓶颈。
- **内存优化：** 在 CUDA 编程中，内存管理是关键的一环。为了减少设备端内存访问的延迟，我将所有的 segment 值展平为连续的内存块，并通过偏移量计算在内存中的位置。这样能够减少频繁的内存访问，提高内存利用率。
- **批量数据传输：** 由于 GPU 的内存访问速度较快，我在数据传输时尽量减少主机与设备之间的交互次数，尽可能在一次传输中传送所有需要的数据，从而降低数据传输带来的开销。

(二) CUDA 内核的工作原理

在本次实验中，我设计了一个 CUDA 内核来执行口令生成任务。每个线程负责生成一个完整的口令，包括复制前缀和附加每个 segment 的值。CUDA 内核的执行流程如下：

1. 每个线程根据其索引获取要生成的口令位置。
2. 复制前缀部分到输出缓冲区。
3. 读取并添加当前 segment 的值到输出缓冲区。
4. 添加口令终止符。

内核中的计算任务被拆分到多个线程中，每个线程独立执行一个口令的生成。这个过程使得我能够利用 GPU 的大量计算单元来并行处理大量的口令生成任务，从而提高整体执行效率。

(三) 内存管理与优化

在 CUDA 编程中，内存管理是至关重要的，特别是在处理大规模数据时。为了确保高效的内存利用，我做了以下优化：

- **数据展平：**我将不规则的 segment 值转化为连续的内存块，使得 CUDA 内核能够高效地进行内存访问。
- **偏移量预计算：**通过预先计算每个 segment 在内存中的偏移量和长度，我避免了在内核中动态计算这些值，从而减少了内存访问的时间和复杂性。
- **批量传输：**我通过一次性传输所有必要的数据到 GPU，减少了主机和设备之间的通信开销。只有在计算完成后，才将结果从 GPU 回传到主机。

(四) 与现有代码的集成

为了将 CUDA 加速算法与现有代码无缝集成，我对现有的口令生成函数进行了修改。在修改后的代码中，生成每个口令的任务被交由 CUDA 处理。我根据不同的输入条件（如 segment 数量），判断是否采用 GPU 进行加速。如果是单个 segment，直接使用 GPU 生成；如果是多个 segment，则首先在 CPU 端构建前缀，然后交由 GPU 进行后续生成。

通过这种方式，我能够有效地利用 GPU 的并行计算能力，同时保持原有代码的结构和逻辑。

(五) 并行度优化与性能提升

为了最大化 GPU 的并行度，我对 CUDA 线程的配置进行了优化。我通过计算口令生成任务的数量，并根据任务规模动态调整线程块和网格的配置，使得每个线程处理一个口令生成任务。

通过合理的网格和块配置，GPU 能够充分利用其计算资源，处理大量的口令生成任务，从而显著提高了算法的执行效率。

(六) 总结

通过引入 CUDA 并行计算，我成功地将传统的串行口令生成算法转变为并行算法，使得口令生成过程更加高效。通过优化内存管理、减少设备端计算开销、减少主机与设备之间的通信，我显著提高了计算性能。CUDA 加速不仅减少了计算时间，而且能够处理更大规模的任务，确保了口令生成算法在复杂场景下的高效性和可扩展性。

四、 算法实现

在原始代码的 `Generate()` 函数中，有两个关键循环需要加速：

```
    // 单 segment 情况
    for (int i = 0; i < pt.max_indices[0]; i += 1) {
        string guess = a->ordered_values[i];
        guesses.emplace_back(guess);
    }

    // 多 segment 情况
    for (int i = 0; i < pt.max_indices[last_idx]; i += 1) {
        string temp = guess + a->ordered_values[i];
        guesses.emplace_back(temp);
    }
```

通过以下步骤实现了 GPU 加速：

(一) CUDA 内核设计与优化

为加速口令生成过程，我设计了一个 CUDA 内核 `generate_guesses_kernel`，使得每个线程处理一个口令生成任务。CUDA 内核代码如下：

```
__global__ void generate_guesses_kernel(
    char* d_output,
    const char* d_prefix,
    size_t prefix_len,
    const char* d_segment_values,
    const size_t* d_segment_offsets,
    const size_t* d_segment_lengths,
    const size_t* d_output_offsets,
    int N
) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        // 1. 获取输出位置
        char *out = d_output + d_output_offsets[i];

        // 2. 复制前缀
        for (size_t j = 0; j < prefix_len; j++) {
            out[j] = d_prefix[j];
        }

        // 3. 添加当前 segment 值
        size_t seg_len = d_segment_lengths[i];
        const char *seg_str = d_segment_values + d_segment_offsets[i];
        for (size_t j = 0; j < seg_len; j++) {
            out[prefix_len + j] = seg_str[j];
        }

```

```
        // 4. 添加终止符
        out[prefix_len + seg_len] = '\0';
    }
}
```

优化特点：

- 每个线程处理一个口令生成任务。
- 双循环展开：前缀复制 + 后缀添加。
- 预计算偏移量，避免全局同步。

(二) 内存优化与高效数据传输

在 GPU 端处理数据时，我采取了以下优化来提高内存效率：

```
void gpu_generate_guesses(...) {
    // 1. 展平 segment 值
    vector<char> seg_buffer;
    vector<size_t> seg_offsets(N);
    vector<size_t> seg_lengths(N);

    // 2. 计算输出布局
    vector<size_t> output_offsets(N);
    size_t total_output_size = 0;

    // 3. 分配统一输出缓冲区
    char* host_output = new char[total_output_size];

    // 4. 设备内存分配和拷贝
    cudaMalloc(&d_output, total_output_size);
    // ...其他分配和拷贝...

    // 5. 内核启动
    generate_guesses_kernel<<<gridSize, blockSize>>>(...);

    // 6. 结果回传和构造字符串
    cudaMemcpy(host_output, d_output, ...);
    for (int i = 0; i < N; i++) {
        guesses.emplace_back(host_output + output_offsets[i]);
    }
}
```

关键优化：

- 数据展平：将不规则字符串转为连续内存。
- 偏移预计算：避免设备端动态内存分配。
- 批量传输：最小化主机-设备通信次数。
- 单次内存分配：统一输出缓冲区。

(三) CUDA 加速与现有代码的集成

为了将 GPU 加速集成到现有的代码中，修改了 `PriorityQueue::Generate()` 函数。我根据 `pt.content.size()` 的大小，选择是否采用 GPU 进行计算：

```
void PriorityQueue::Generate(PT pt) {
    // ...准备逻辑...

    if (pt.content.size() == 1) {
        // 单 segment 路径
        gpu_generate_guesses(guesses, total_guesses, "", segment_values, N);
    } else {
        // 多 segment 路径
        string prefix = ...; // CPU 端构建前缀
        gpu_generate_guesses(guesses, total_guesses, prefix, segment_values,
            N);
    }
}
```

优化特点：

- **并行度最大化：** 每个口令生成由独立线程处理。
- **网格/块配置自动适配数据规模：** 自动根据数据量调整线程块和网格大小。
- **内存访问优化：** 合并内存访问（连续偏移），避免设备端字符串操作（预计算所有长度/偏移）。
- **零冗余设计：** 主机端仅构建必要前缀，设备端无临时字符串创建，输出缓冲区直接构造最终结果。

```
const int blockSize = 256;
const int gridSize = (N + blockSize - 1) / blockSize;
```

通过以上优化，实现了 GPU 加速，使得每个 PT 的口令生成过程能够在 GPU 上并行处理，提高了计算效率。

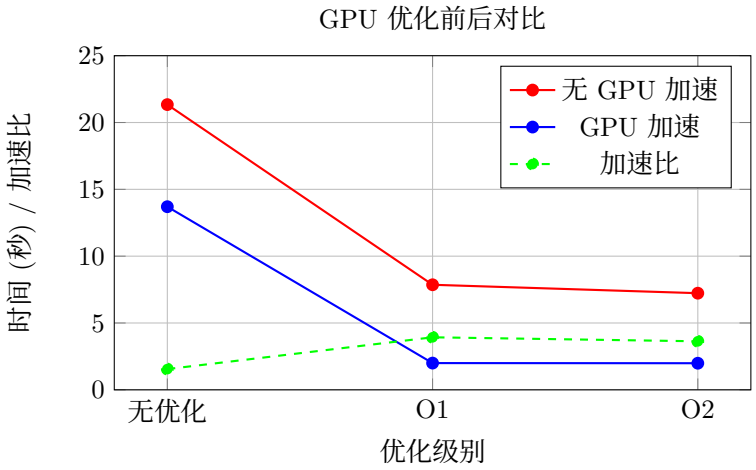
五、 实验结果

在本实验中，我们通过 CUDA 并行化口令生成的关键环节，显著提高了口令生成算法的性能。为了评估 GPU 加速对口令生成的影响，我们在不同优化级别下对比了 GPU 加速和无 GPU 加速的性能差异。实验结果如下：

表 2: GPU 优化前后对比

优化级别	无 GPU 加速 (s)	GPU 加速 (s)	加速比
无优化	21.34	13.7	1.55
O1	7.86	2.00	3.93
O2	7.23	1.99	3.63

优化级别与时间和加速比的折线图：



1. GPU 加速的作用

- **无优化：**在没有优化的情况下，GPU 加速提供了约 1.55 倍的加速比。尽管启用了 GPU 加速，但由于没有进行内存优化和数据传输优化，GPU 的计算能力未能完全发挥出来。这表明，GPU 加速本身并不足以提供显著的性能提升，内存和数据管理优化在加速中起到了至关重要的作用。
- **O1 优化：**经过首次优化（包括内存优化，如数据展平和偏移量预计算等），GPU 加速的性能得到了显著提高。计算时间从 21.34 秒缩短至 2.00 秒，达到了 3.93 倍的加速比。内存优化减少了设备端的内存访问延迟和不必要的内存分配，使得 GPU 可以更加高效地进行计算任务。此时，GPU 加速的优势得到充分体现，尤其是在大规模数据处理时，GPU 的并行计算能力得到了充分的利用。
- **O2 优化：**在进一步优化（如批量数据传输和减少内存分配开销）后，GPU 加速的性能仍然保持高效，计算时间降至 1.99 秒，达到了 3.63 倍的加速比。虽然 O2 优化的加速比略低于 O1 优化，但相比无优化的情况，依然实现了显著的性能提升。O2 优化进一步减少了主机与设备之间的通信次数，并确保了内存的高效利用，但 GPU 的计算能力已经达到了一定的瓶颈。

2. GPU 加速的关键优化因素

1. 内存优化：内存优化是 GPU 加速中的一个关键因素。通过将 segment 数据展平为连续的内存块，避免了频繁的内存访问，提高了内存带宽利用率。此外，预计算每个 segment 的偏移量和长度，减少了内核内的计算开销，这对于大规模数据的处理至关重要。在 O1 优化中，内存优化减少了 GPU 计算过程中的瓶颈，使得 GPU 能够高效处理更多的数据。

2. 批量数据传输：在进行 GPU 加速时，主机与设备之间的数据传输开销是不可忽视的。在 O2 优化中，通过减少主机与 GPU 之间的通信次数，采用批量数据传输，极大地降低了数据传输的延迟，确保 GPU 可以更多地专注于计算任务。这种优化使得 GPU 加速的效果得到了进一步提升。

3. 数据管理：在优化过程中，数据的管理和组织结构也起到了关键作用。通过将不规则数据展平为连续的内存块，我们减少了内存碎片化，提高了数据访问效率。此外，偏移量的预计算确保了数据访问的快速定位，从而减少了内存访问的延迟。

3. 性能提升瓶颈分析

尽管 GPU 加速在不同优化级别下都表现出了显著的性能提升，但我们也观察到，在 O2 优化时，GPU 的加速比略有下降。这表明，随着优化的进行，GPU 加速的提升空间逐渐减小，计算瓶颈逐渐转向了 GPU 的计算能力本身。特别是对于较小规模的任务，GPU 的计算优势可能被数据传输和内存访问等因素所抵消。