南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计课程报告

多进程编程实验

孙沐赟

年级：2023级

专业：计算机科学与技术

指导教师：王刚

2025 年 7 月 2 日

目录

1 使用GPU加速程序运行

1.1 GPU加速简述

首先在进行实验之前，我们来了解一下为什么GPU能够对程序进行加速？

CPU 由专为顺序串行处理而优化的几个核心组成，而 GPU 则拥有一个由数以千计的更小、更高效的核心（专为同时处理多重任务而设计）组成的大规模并行计算架构，这就使 GPU 拥有了计算并行计算的优势。虽然 GPU 每个核心都很弱，但众多的核心还是让GPU在并行计算上拥有了相当的优势。另外一点，GPU 有相当的价格优势，也就是说我们可以通过较小的代价来实现 GPU 的加速。

在简单了解 GPU 的用法与功能之后，我们来具体实现在口令猜测中Generate函数的 GPU 实现。

2 GPU 优化（第一版代码）

首先我们来看一下实现的思路：要进行 GPU 上的程序执行与代码实现，我们需要首先在 CPU 端将所有待生成的字符串内容及其偏移量进行集中打包，然后传输至 GPU 显存。然后，GPU 启动一个定制的 CUDA 内核，其中每个线程独立地负责从打包数据中提取并构建一个完整的猜测字符串。这里的猜测过程我们就能够利用 GPU 的并行计算的优势，将这种大规模并行计算使用 GPU 加速猜测的生成效率。最后，GPU 将所有生成的猜测回传至 CPU，并在后续处理前进行必要的同步，确保数据完整性。

按照以上思路，我进行了第一版代码的实现，代码具体实现方法如下：

2.1 GPU 数据处理：打包、传输与回传

为了充分利用 GPU 的并行计算能力，数据在主机（CPU）和设备（GPU）之间需要经过精心准备和高效传输。这个过程主要分为数据打包、主机到设备的内存传输，以及计算结果从设备回传到主机三个核心步骤。

2.1.1 数据打包 (CPU 端操作)

在将数据发送到 GPU 之前，我们首先在 CPU 端将所有独立的字符串内容有效地组织起来，形成一个连续的内存块，并记录每个字符串的精确位置。

```
std::vector<char> all_data; // 存储所有字符串内容的连续缓冲区
std::vector<int> offsets(num + 1);
int pos = 0; // 当前在 all_data 中的写入位置
for (int i = 0; i < num; ++i) {
        const std::string &s = /* (a->ordered_values[i] 或
            guess_prefix + a->ordered_values[i]) */;
        offsets[i] = pos;
        all_data.insert(all_data.end(), s.begin(), s.end());
        pos += s.size(); // 更新写入位置
}
offsets[num] = pos; // 最后一个偏移量记录了 all_data 的总大小
```

这段代码将多个独立的 std::string 对象转换成 GPU 更易于处理的连续字节数组 all_data。这是为了实现 GPU 的合并访问，即多个线程同时访问内存时能够以更宽的位宽高效读取相邻数据。同时，我们构建了一个 offsets 数组，它精确记录了 all_data 中每个原始字符串的起始位置。通过 offsets 数组，GPU 上的每个线程都能快速定位到它需要处理的特定字符串，从而避免了在 GPU 上进行复杂的字符串解析操作。

2.1.2 主机到设备内存传输 (CPU 端操作)

数据在 CPU 端准备好之后，就需要被高效地传输到 GPU 显存，以便 GPU 内核能够访问它们。

```
static char *d_all_data = nullptr;
static int *d_offsets = nullptr;
static char *d_output = nullptr;
static size_t all_data_capacity = 0, offsets_capacity = 0,
    output_capacity = 0;
static cudaStream_t stream = nullptr; // 用于管理异步操作的 CUDA 流
if (!stream) cudaStreamCreate(&stream); // 首次调用时创建流

if (all_data.size() > all_data_capacity) {
        if (d_all_data) cudaFree(d_all_data);
        cudaMalloc(&d_all_data, all_data.size() * sizeof(char));
        all_data_capacity = all_data.size();
}
cudaMemcpyAsync(d_all_data, all_data.data(), all_data.size() *
    sizeof(char), cudaMemcpyHostToDevice, stream);
cudaMemcpyAsync(d_offsets, offsets.data(), (num + 1) * sizeof(int)
    , cudaMemcpyHostToDevice, stream);
```

为了最大限度地提高效率，我在 GPU 内存的分配和数据传输部分都采用了优化策略。首先，我们使用静态变量来持有 GPU 内存指针及其容量信息。这意味着 GPU 内存块在函数多次调用之间可以被复用，只有当新的数据量超出当前已分配容量时，才会触发重新分配操作，从而显著减少了频繁 cudaMalloc 和 cudaFree 的性能开销。同时，数据传输通过 cudaMemcpyAsync 函数和 cudaStream_t 实现异步化。这允许 CPU 在将 all_data 和 offsets 复制到 GPU 的同时，能够继续执行其他任务，实现了计算与数据传输的重叠，进一步提升了程序的整体吞吐量。

2.1.3 结果回传 (GPU 端到 CPU 端)

在 GPU 完成所有并行计算并生成猜测字符串后，这些结果需要被复制回 CPU 内存，以便后续的主机端逻辑进行处理。

```
static std::vector<char> h_output;
h_output.resize(num * max_len);
cudaMemcpyAsync(h_output.data(), d_output, num * max_len * sizeof(
    char), cudaMemcpyDeviceToHost, stream);

cudaStreamSynchronize(stream);
```

```
6
7  guesses.reserve(guesses.size() + num);
8  for (int i = 0; i < num; ++i) {
9          guesses.emplace_back(&h_output[i * max_len]);
10         total_guesses += 1;
11 }
```

首先，GPU 上的计算结果存储在设备内存的 d_output 缓冲区中。为了让 CPU 能够使用这些结果，我们再次利用 cudaMemcpyAsync 将 d_output 中的数据异步复制回主机内存的 h_output 缓冲区。h_output 也被声明为静态，以实现与 GPU 内存类似的复用机制。在此之后，一个关键步骤是调用 cudaStreamSynchronize(stream)。这是一个阻塞式调用，它确保了当前 CUDA 流中所有排队的操作（包括先前的内存传输和 GPU 内核执行）都已完全完成，特别是从设备到主机的最终数据传输已经结束。只有在同步完成后，CPU 才能安全地从 h_output 中提取出每个固定长度的猜测字符串，将它们构造为 std::string 对象，并添加到最终的 guesses 结果列表中，同时更新猜测总数。

至此，GPU 数据处理：打包、传输与回传的完整过程我们已经实现，接下来我们来具体解释一下在数据传输过程中的内存复用是如何实现的。

2.2 GPU 内存管理：静态复用

在 GPU 编程中，内存管理与 CPU 端截然不同。不像 C++ 的标准库容器（ std::string 和 std::vector等）能够自动处理内存，GPU 内存需要显式控制。我在代码通过运用静态内存复用技术，大大减少了内存管理中的开销。

2.2.1 静态变量声明与流初始化

```
1          static char *d_all_data = nullptr;
2          static int *d_offsets = nullptr;    // 设备上字符串偏移量的指针
3          static char *d_output = nullptr;
4          static size_t all_data_capacity = 0, offsets_capacity = 0,
               output_capacity = 0;
5          static cudaStream_t stream = nullptr;
6          if (!stream) cudaStreamCreate(&stream);
```

这部分代码负责设置用于 GPU 内存的静态指针和容量追踪器，以及一个用于异步操作的 CUDA 流。这些 static 声明是实现跨函数调用内存复用的关键。通过将这些指针和容量变量声明为 static，它们在 Generate 函数的不同调用之间得以保留。这意味着程序可以“记住”是否已经分配了 GPU 内存以及当前有多少可用空间。cudaStream_t 也被静态初始化，确保所有后续的异步内存传输和内核启动都在特定的 CUDA 流中进行，从而实现 CPU 和 GPU 任务的重叠。这种设置通过为可能的内存复用做准备，为高效内存管理奠定了基础。

2.2.2 容量检查与条件式内存分配

```
1          if (all_data.size() > all_data_capacity) {
2                  if (d_all_data) cudaFree(d_all_data);
```

```
            cudaMalloc(&d_all_data, all_data.size() * sizeof(
                char));
            all_data_capacity = all_data.size(); // 更新容量
        }
        if ((num + 1) > offsets_capacity) {
            if (d_offsets) cudaFree(d_offsets);
            cudaMalloc(&d_offsets, (num + 1) * sizeof(int));
            offsets_capacity = num + 1;
        }
        if ((num * max_len) > output_capacity) {
            if (d_output) cudaFree(d_output);
            cudaMalloc(&d_output, num * max_len * sizeof(char)
                );
            output_capacity = num * max_len;
        }
```

这部分代码是静态内存复用机制的关键操作。它不是在每次调用 Generate 时都盲目地分配新内存，而是首先检查现有已分配的内存是否足以满足当前任务的需求。对于每个 GPU 内存缓冲区（d_all_data、d_offsets、d_output），代码会比较当前操作所需的内存大小与静态 _capacity 变量记录的已分配容量。如果当前数据大小大于先前分配的容量，则会首先使用 cudaFree() 释放现有 GPU 内存，然后使用 cudaMalloc() 分配新的、更大的内存。这种条件式重新分配是一种强大的优化手段。它避免了在 GPU 上频繁进行内存分配和释放操作所带来的性能开销，因为这些操作通常相对昂贵。通过尽可能地保留和复用内存，应用程序的整体效率得到了显著提升。

2.3 GPU 的 CUDA 内核

GPU 编程的核心在于 CUDA 内核，它们是专门设计用于在 GPU 上大规模并行执行的函数。与传统的 CPU 函数不同，CUDA 内核利用了 GPU 的独特架构，让成千上万个线程能够同时处理数据，从而真正的实现程序在 GPU 上的执行。

2.3.1 内核启动与线程配置

在主机（CPU）端，我们启动 CUDA 内核。这不仅调用了内核函数，还定义了它在 GPU 上执行的并行结构。

```
int block = 256; // 每个线程块中的线程数量
int grid = (num + block - 1) / block;
generate_guesses_kernel<<<grid, block, 0, stream>>>(d_all_data,
    d_offsets, num, d_output, max_len);
```

这段代码首先计算了内核启动所需的线程块（block）和网格（grid）维度。block 定义了每个线程块包含多少个线程，而 grid 则根据需要处理的任务总数 (num) 计算出应该有多少个线程块。generate_guesses_kernel¡¡¡grid, block, 0, stream¿¿¿ 语法是 CUDA 内核的启动方式。它指示 GPU 以 grid 个线程块，每个块包含 block 个线程的方式来执行 generate_guesses_kernel 函数。第三个参数 0 表示不使用动态共享内存，而 stream 参数则将内核的执行安排在特定的 CUDA 流中，确保其与之前的数据传输操作保持异步且有序。

2.3.2 GPU 内核：并行执行与数据定位

generate_guesses_kernel 是在 GPU 上实际执行并行计算的函数。它的逻辑是为单个线程设计的，但通过 CUDA 运行时环境，该逻辑会被成千上万个线程同时执行。

```
__global__ void generate_guesses_kernel(const char *all_data,
    const int *offsets, int num, char *d_output, int max_len) {
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if (idx < num) {
                int start = offsets[idx];
                int len = offsets[idx + 1] - start;
                const char *src = all_data + start;
                char *dst = d_output + idx * max_len;

                if (len > 0) {
                        memcpy(dst, src, len);
                }
                dst[len] = '\0'; // 添加空终止符，使其成为有效的 C 字符串
        }
}
```

这个 __global__ 函数是 GPU 并行的核心。进入内核后，每个 GPU 线程会通过访问 CUDA 内置变量（如 blockIdx.x、blockDim.x 和 threadIdx.x）来计算出一个唯一的全局索引 idx。这个索引直接对应了它需要处理的特定密码猜测。if (idx ¡ num) 语句是一个重要的边界检查，以防启动的线程数量多于实际需要处理的任务数量。

获取到自己的索引后，每个线程利用之前从主机传入的 offsets 数组，能够迅速定位到 all_data 连续缓冲区中属于自己的那部分源字符串的起始位置和长度。然后，它计算出在 d_output 缓冲区中应该写入结果的目标地址，max_len 确保了每个猜测都有固定的、互不重叠的内存空间。

最后，memcpy(dst, src, len) 执行了实际的字符串复制操作。在 GPU 上，memcpy 是一个高度优化的函数，能够充分利用硬件的内存带宽，实现比手动循环逐字节复制更高的效率。在复制完成后，dst[len] = '\0' 确保了每个生成的字符串都以空字符结尾，使其在被复制回 CPU 后仍然是有效字符串。通过这种设计，每个线程独立且高效地完成了一小部分工作，共同构成了大规模并行猜测生成。

2.4 GPU 同步机制：流与同步点

GPU 编程中的一个关键挑战是协调 CPU 和 GPU 之间的操作，以及管理 GPU 内部的异步任务。由于 CPU 和 GPU 通常并行工作，并行的操作需要一个同步机制来确保数据的正确性和程序的预期行为。CUDA 提供了“流”和“同步点”的概念来高效地实现这一点。

2.4.1 CUDA 流的创建与使用

CUDA 流是 GPU 上一系列操作（例如内存传输和内核执行）的有序队列。它们允许你在同一个流内保持操作顺序，同时在不同流之间实现并发。

```
static cudaStream_t stream = nullptr;
if (!stream) {
        cudaStreamCreate(&stream);
```

```
}
```

这段代码实现了创建和管理一个 CUDA 流。static cudaStream_t stream = nullptr; 将流句柄声明为静态变量，这意味着它在 Generate 函数的不同调用之间会保持其状态，从而复用同一个流。if (!stream) cudaStreamCreate(&stream); 确保了流只在程序首次运行时创建一次。通过将所有相关的内存传输 (cudaMemcpyAsync) 和内核启动 (¡¡¡¿¿¿ 语法) 都关联到这个 stream，你可以确保这些操作在 GPU 上按照它们在代码中出现的顺序依次执行。这对于维护数据依赖性至关重要，例如，数据必须先传输到 GPU，然后内核才能对其进行操作。

2.4.2 异步操作与 CPU/GPU 并发

CUDA 流的一个主要优势是它支持异步操作，这意味着 CPU 和 GPU 可以同时执行任务，而无需互相等待。

```
cudaMemcpyAsync(d_all_data, all_data.data(), all_data.size() *
    sizeof(char), cudaMemcpyHostToDevice, stream);
cudaMemcpyAsync(d_offsets, offsets.data(), (num + 1) * sizeof(int)
    , cudaMemcpyHostToDevice, stream);

int block = 256;
int grid = (num + block - 1) / block;
generate_guesses_kernel<<<grid, block, 0, stream>>>(d_all_data,
    d_offsets, num, d_output, max_len);
static std::vector<char> h_output;
h_output.resize(num * max_len);
cudaMemcpyAsync(h_output.data(), d_output, num * max_len * sizeof(
    char), cudaMemcpyDeviceToHost, stream);
```

在这里，cudaMemcpyAsync 函数是实现异步传输的关键。当调用它时，数据传输任务被提交到指定的 stream 中，但 CPU 不会等待传输完成，而是立即返回并执行后续代码。同样，内核的启动也是异步的，它被添加到流中并在数据准备好后开始执行。从 GPU 到 CPU 的结果回传也使用 cudaMemcpyAsync 实现异步。这种机制允许 CPU 准备数据、启动内核，甚至开始处理其他任务，而 GPU 则在后台并行执行这些操作。

2.4.3 同步点：确保数据一致性

尽管异步操作能够提高效率，但在某些时刻，CPU 必须确保 GPU 上的所有操作都已完成，尤其是在它需要访问 GPU 生成的结果时。

```
cudaStreamSynchronize(stream);
guesses.reserve(guesses.size() + num);
for (int i = 0; i < num; ++i) {
        guesses.emplace_back(&h_output[i * max_len]);
        total_guesses += 1;
}
```

cudaStreamSynchronize(stream) 是一个阻塞调用。当 CPU 执行到这一行时，它会暂停并等待指定 stream 中的所有先前操作（包括所有异步内存传输和 generate_guesses_kernel 内核的

执行）全部完成。这个函数充当了 CPU 和 GPU 之间的同步点。它的存在至关重要，因为它保证了当 CPU 开始从 h_output 中读取数据时，这些数据已经完全从 GPU 传输回来并且是最终、正确的计算结果。没有这个同步点，CPU 可能会尝试读取尚未完成传输或计算的数据，导致程序出错或结果不正确。

2.5 第一版代码的结果分析

下面，我们对程序进行运行结果的分析。

将使用 GPU 优化之后的函数与原串行算法在公共服务器进行执行，并在不使用编译优化与使用O2编译优化的情况下进行对比，结果如下：

表 1: GPU 优化算法（版本1）与串行算法Guessing Time的比较

	不使用编译优化/单位(s)	编译优化/单位(s)
串行算法	8.87	0.45
多进程算法	7.71（加速比：1.15）	1.19 （加速比：0.38）

通过上述表格我们可以知道，在不使用编译优化的情况下第一版代码实现了加速但是加速比不算很高，在使用 O2 编译优化时，使用 GPU 的程序的运行速度反而更慢了，为了进一步优化我们的程序，我们来对代码进行进一步分析。

2.6 进一步优化的方向

为什么我们上述代码的加速的效果一般呢？

通过进行代码的分析发现，每次调用 Generate 都完整执行数据打包、CPU-GPU 传输和 GPU-CPU 传输，即使猜测的任务量很小，其固定开销也会抵消 GPU 的计算优势，所以在有许多小任务执行的情况下，最终的运行时间可能反而会更长，所以我们需要对口令猜测的任务量进行判断。

并且在上述代码中，PopNext() 函数是单次处理的，即每次从优先级队列中取出一个 PT 对象，然后调用 Generate() 来生成其对应的猜测。即使 Generate() 内部有 GPU 加速逻辑，每次 GPU 任务的启动、数据传输和同步都会带来不小的开销。所以我们可以通过依次从队列中取出多个 PT 进行对于 PopNext() 函数的并行化。

3 进阶要求（第二版代码）

通过对于第一版代码不足之处的分析，我们可以找出代码的进一步优化方向：通过判断任务量的大小来判断是否需要进行并行化，对 PopNext() 函数进行并行化处理。在代码实现中，我们把任务量阈值判断优化最为 PopNext() 优化的一部分来进行代码实现，下面是对于 PopNext() 进行优化后的 PopNextParallel() 代码实现的具体分析。

3.1 函数声明和初始检查

```
1 void PriorityQueue::PopNextParallel()
2 {
3         int batch_size = 100;
```

```
        if (priority.empty()) return;
        int n = std::min(batch_size, (int)priority.size());
        std::vector<PT> batch_pts(priority.begin(), priority.begin
            () + n);
```

我们定义了 PopNextParallel 函数并处理初始设置，通过声明一个批处理大小为 100 的变量 batch_size 来确定单次迭代中处理的最大 PT 对象数量，随后使用 if (priority.empty()) return 检查优先队列是否为空，以避免无效操作，然后通过 std::min(batch_size, (int)priority.size()) 计算实际批处理数量 n，取批处理大小和当前队列大小中的较小值以限制处理项数量，最后使用范围构造函数从 priority.begin() 到 priority.begin() + n 将前 n 个元素提取到 std::vector¡PT¿ batch_pts 中，为高效的批处理奠定基础。

3.2 数据准备和字符串生成

这部分代码实现与使用 CPU 的串行代码实现方法几乎一致，在此不赘述具体代码的实现，只讲解实现思路。通过循环遍历批次中的每个处理对象，统一处理单段和多段情况下的字符串生成，初始时准备一个存储所有猜测结果的集合和一个偏移量记录机制，用于后续内存优化。处理过程中，先对每个对象进行概率计算以确保一致性，然后根据段数区分处理逻辑：单段直接从预定义值中生成字符串，多段则先构建前缀，再与最后一个段的值拼接生成完整字符串。

3.3 任务量判断和 GPU/CPU 选择

3.3.1 任务量计算和初始化

```
int total_guesses = all_guesses.size();
max_len += 1;
```

在进行选择初始化任务量判断的关键变量，首先通过 total_guesses = all_guesses.size() 计算批次中所有 PT 对象生成的总字符串数量，作为后续阈值判断的依据，然后 max_len += 1 为每个字符串的结束符预留空间，确保后续内存分配和字符串处理的安全性，为 GPU 或 CPU 路径的选择奠定基础。

3.3.2 任务量阈值判断和 GPU 路径入口

如果 total_guesses 超过阈值，它会执行一次性的大规模 GPU 操作：将所有字符串数据拼接成一块连续内存，连同偏移量一起传输到 GPU，启动一个 CUDA Kernel 来并行生成所有猜测，然后一次性将结果传回主机。(代码实现与原实现方式相似，但是我们将所有的 PT 分配了更大的内存，并将其对应的偏移量一同传入 GPU)

如果 total_guesses 较小，则退回到 CPU 进行串行处理，避免了 GPU 启动的额外开销。

因为在使用 GPU 并行执行的代码实现大致相同，我们在此展示如何对一批 PT 分配内存空间

```
std::vector<char> all_data(offsets.back());
int pos = 0;
for (const auto& s : all_guesses) {
        memcpy(&all_data[pos], s.data(), s.size());
        pos += s.size();
```

```
}
```

通过一个for循环：for (const auto& s : all_guesses)遍历all_guesses中每一个独立的std::string猜测。在循环内部，memcpy(&all_data[pos], s.data(), s.size());是将当前字符串s的原始数据（s.data()指向的内存内容）精确地复制s.size()个字节到all_data向量中从pos位置开始的地方，实现了将所有离散的字符串数据“扁平化”到一块连续的内存中。最后，pos += s.size();更新了pos的值，使其指向下一个字符串应该开始写入的位置，确保了所有字符串紧密无间地拼接在一起。

3.3.3 GPU 数据传输和内核执行

```
cudaMemcpyAsync(d_all_data, all_data.data(), all_data.size() *
    sizeof(char), cudaMemcpyHostToDevice, stream);
cudaMemcpyAsync(d_offsets, offsets.data(), offsets.size() * sizeof
    (int), cudaMemcpyHostToDevice, stream);
int block = 256;
int grid = (total_guesses + block - 1) / block;
generate_guesses_kernel<<<grid, block, 0, stream>>>(d_all_data,
    d_offsets, total_guesses, d_output, max_len);
```

这部分代码执行 GPU 数据传输和内核调用，首先通过 cudaMemcpyAsync 异步将 all_data 和 offsets 传输到设备端，利用 stream 提高并发性，然后定义块大小 block = 256 并计算网格大小 grid 基于 total_guesses，启动 generate_guesses_kernel 并行处理，传递 total_guesses 作为任务总数。

后续的 GPU 结果回传与 CPU 回退路径的实现与版本1类似，在此不赘述。

3.4 优化后代码的测试结果

表 2: GPU 优化算法（版本1）与串行算法Guessing Time的比较

	不使用编译优化/单位(s)	编译优化/单位(s)
串行算法	8.87	0.45
多进程算法	3.08（加速比：2.88）	0.34 （加速比：1.32）

可以看到经过任务量判断与对 PopNext() 函数并行化处理之后，加速比已经变得非常可观了。

4 对于问题规模阈值的思考

在上述的版本二中有这样一个问题：当我们的 PopNext() 函数并行的 BatchSize 越大，每次要进行猜测的任务量就会越多，那么我们需要在任务阈值与 BatchSize 之间找到一个平衡，使我们的程序性能最优化。

首先，我们在 BatchSize 规模分别为10和100的情况下来筛选出最佳的任务量阈值，为了使测试结果更加准确，我不使用编译优化，然后将测试结果的 Guessing Time 通过折线图的方式来进行展示，进而找到最佳任务阈值。

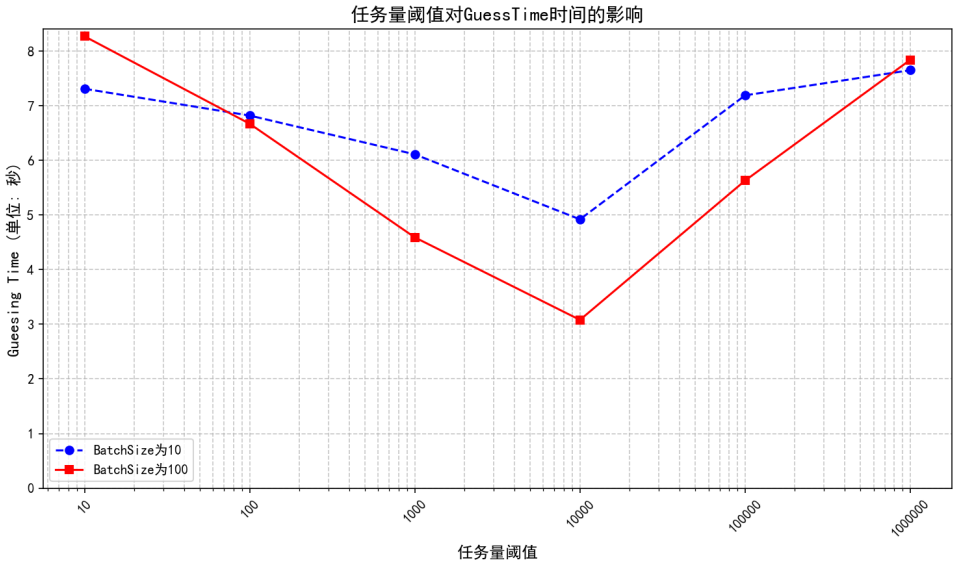

图 1: 任务量阈值对Guess Time的影响

从上面的分析中我们可以看出，最佳的任务量阈值在10000左右，接下来，我们将10000设置为任务量阈值来判断 BatchSize 的最优值为多少。

表 3: 任务阈值10000左右不同BatchSize的Guessing Time比较

串行	5	10	50	100	200	500
8.87	7.92	7.39	5.11	3.08	3.65	4.72

经过如上测试，我们可以确定问题规模与 BatchSize 之间的最佳平衡是任务阈值为10000左右，BatchSize在100左右。

5 实验总结

本次实验利用GPU并行计算能力对口令猜测程序进行优化。在初始版本中，我们实现了完整的GPU加速流程：通过数据打包、异步传输和CUDA内核并行执行，将字符串生成任务迁移至GPU处理。然而测试显示，当任务规模较小时频繁的GPU调用反而降低性能（串行0.45s vs GPU 1.19s）。基于此，第二版代码引入动态任务调度机制——在PopNextParallel()函数中设置批处理大小（BatchSize），通过total_guesses统计待处理任务量，仅当超过阈值（约10000）时启用GPU加速。经多轮测试，最终确定BatchSize=100时性能最优，在编译优化下实现0.34s的Guessing Time，较串行算法提升1.32倍加速比。实验同时验证了任务量阈值与BatchSize的强关联性：阈值过低导致GPU调用频繁，过高则降低并行度。

通过本实验，我对GPU加速的思想有了一定的了解：从数据打包传输（cudaMemcpyAsync）、内存静态复用（条件式cudaMalloc）到内核设计（generate_guesses_kernel的全局索引计算与内存定位）。同时，我也认识到GPU并非万能解——小规模任务因固定开销（流创建、传输延迟）可能劣于CPU，需通过阈值机制动态切换路径。最后，在实现对于程序的最终优化之后，我也发现了任务量规模与并行规模对于程序的影响也是巨大的。这次试验中的经历与收获对于以后我编写程序也一定会有很大的帮助!

我的本次实验的github仓库链接：https://github.com/fan-tuaner614/NKU-