
南開大學
Nankai University

计算机学院

并行程序设计 **GPU** 编程报告

# **GPU** 并行加速的口令猜测算法

姓名：苏雨辰

学号：2313828

专业：计算机科学与技术

2025 年 7 月 2 日

# 目录

# 1 背景介绍

## 1.1 GPU 编程简介

GPU（Graphics Processing Unit，图形处理器）编程是指利用 GPU 的并行计算能力来加速各种计算任务的编程方式。

与 CPU 强调顺序执行和复杂控制逻辑不同，GPU 拥有数以千计的计算核心，其设计初衷是为了并行处理大量图形渲染任务。这种架构使得 GPU 在处理高度并行化的计算任务时，具备远超 CPU 的计算能力。为了支持大量计算核心同时访问数据，GPU 配备了高带宽的内存系统，能够快速地读取和写入数据，以满足并行计算对数据的需求。

在本实验中，就是利用 GPU 多核心并行处理大量简单任务的特点，将 guessing.cpp 中生成口令的两个循环使用 CUDA 进行 GPU 并行编程。同时针对 PT 层面进行并行编程，即一次性取出多个 PT 并且进行并行生成。考虑到 GPU 与 CPU 之间的通信开销，尝试针对不同 PT，选择 GPU 或 CPU 进行运算，避免 GPU 的额外开销。

## 1.2 CUDA 编程的基本模式

CUDA（Compute Unified Device Architecture）是由 NVIDIA 推出，是目前最广泛使用的 GPU 编程模型之一。它基于 C/C++ 语言进行扩展，提供了一系列的 API 和编程接口。在 CUDA 编程中，开发者编写在 GPU 上执行内核函数（kernel），并定义线程层次结构（线程、线程块、网格）来组织计算任务。在内存传输方面，可以在 CPU 分配好内存后显式调用数据传输的接口传给 GPU，或者可以通过统一虚拟内存（UVM）接口或者零拷贝内存（zero-copy）接口，将数据分配在 CPU 端，此时 GPU 端的线程将可以通过 PCIE 总线直接访问这块内存区域。[1]

## 1.3 实验平台及环境配置

本次实验使用并行智算云平台的云服务器进行，使用显卡为 1 个 RTX4090，10 核 24GB 显存，54GB 内存。编译采用 nvcc 进行编译，编译器优化模式开启 O1，使用 vscode 连接远程服务器进行编程。

# 2 CUDA 编程基础要求

在基础实验部分，实验目的依然是对于生成猜测口令的两个循环进行 CUDA 并行化编程，通过 GPU 多核心的特点，同时生成多个口令。本次代码分析以第二个循环（多个 segment）口令生成为例子，分析如何实现 CUDA 的并行加速。

## 2.1 guessing_cuda.cu 及 guessing_cuda.h

对于原来的两个循环，我们需要将它们分别包装成两个核函数，在原来的 guessing 文件中进行任务分配并调用。核函数通过单指令多线程（SIMT）模型实现并行，现成的层次结构为：网格（Grid 对应整个数据集）——> 线程块（Block 共享内存）——> 线程（Thread 执行相同代码，但处理不同数据）。GPU 编程需要 include “cuda_runtime.h” 的头文件。我们以第二次循环为例分析核函数的作用，代码如下。

```
__global__ void GenerateLastSegmentKernel_FlatOutput(char *prefix, char **values,
char *output_flat, int count, int prefix_len, int total_len_per_guess) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < count) {
        char *val = values[idx];
        int val_len = 1;
        // 复制前缀到输出缓冲区的正确位置
        for (int i = 0; i < prefix_len; ++i) {
            output_flat[idx * total_len_per_guess + i] = prefix[i];
        }
        output_flat[idx * total_len_per_guess + prefix_len] = val[0];
        output_flat[idx * total_len_per_guess + prefix_len + val_len] = '\0';
    }
}
```

___global___ 是一个特殊的函数声明说明符，其作用是告诉编译器：被修饰的函数是一个核函数，该函数会在 GPU 设备上执行，并且可以从主机端（CPU）调用。由于 GPU 无法直接返回值给 CPU，返回类型为 void，需通过全局内存间接传递结果。传入的参数从左到右依次是固定前缀字符串的指针、二位字符指针数组（每一个元素使一个指向字符的指针）、扁平化输出缓冲区数组、生成口令（后缀）数量、前缀长度、生成口令总长度。

每个线程通过全局索引 idx 确定自己负责处理的后缀字符串在数组中的位置，在核函数内通过 BlockID 核 ThreadID 共同计算确定。这里使用 if 进行边界检查边界检查，确保只处理有效范围内的索引。用 for 循环将前缀依次拷贝，加上后缀，口令就生成了。这里注意，由于 CUDA 不支持 string 的一些操作如 strcpy，因此拷贝需要自己手动写。一次得到的口令是 out，我们根据线程代码 idx，将它放入最终输出结果 output 的对应位置。最后放入最后一个 segment 和串尾符。上述就是核函数的具体执行逻辑（单个 segment 类似），每一个核函数只进行一个口令的拼接，需要在原来循环的文件里进行任务、内存分配，数据传输等。

在 guessing_cuda.h 中，是头文件，定义了这两个函数的声明，方便在 guessing 中进行调用。

## 2.2 guessing.cu 修改

在此部分由于需要进行 CUDA 编程，将后缀“.cpp”改为“.cu”，方便后续编译核函数编写，如果不修改，会导致编译时不能识别 CUDA 函数，编译失败。下面主要从与 CUDA 相关的地方进行分析，其他地方改动前后逻辑几乎不变不做赘述。

```
char **h_values = new char*[count];
for (int i = 0; i < count; i++) {
    h_values[i] = const_cast<char*>(a->ordered_values[i].c_str());
}

 // 计算每个拼接后的总长度 (前缀长度 + 后缀长度 1 + 空终止符 1)
int total_len_per_guess = guess_prefix.length() + 1 + 1; // 后缀长度固定为 1
```

```

  char **d_values;
  char *d_prefix;
  char *d_output_flat; // 扁平化输出缓冲区

  cudaMalloc(&d_values, count * sizeof(char*));
  cudaMalloc(&d_prefix, guess_prefix.size() + 1); // +1 for null terminator
  cudaMalloc(&d_output_flat, count * total_len_per_guess * sizeof(char));

  cudaMemcpy(d_values, h_values, count * sizeof(char*), cudaMemcpyHostToDevice);
  cudaMemcpy(d_prefix, guess_prefix.c_str(), guess_prefix.size() + 1,
  cudaMemcpyHostToDevice);
```

此部分为分配 GPU 内存并传输数据的部分。创建主机端指针数组 h_values，用于存储所有最后一个 segment 的指针。接下来在 CPU 上声明变量：d_values 作为指针数组：存储最后一个片段的所有可能值；d_prefix：存储固定前 n-1 的字符串；d_output_flat 作为主机端扁平化输出缓冲区。这些值目前还不能被 GPU 访问，在 GPU 上也没有对应的空间，因此我们需要 CUDA 的 API 申请内存和传输数据。cudaMalloc 用于在 GPU 上分配内存。它需要两个参数，第一个参数是 CPU 中数组的指针，第二个参数是该部分的字节数，我们将所有需要用到的数组都进行分配。cudaMemcpy 用于 CPU，GPU 之间复制内存，其参数从左到右为目标指针、源指针、要复制的字节数、复制方向（CPU–>GPU）。复制只针对前面的 segments 和最后一个 segment，用于将参与运算的内容放入 GPU。

这里有一个细节是采用了扁平化缓冲区，其原理是对于多个不定长元素的存储，通常使用一个二维数组进行管理（如 char[][] 数组），但是此种方式不仅会浪费内存同时分散的内存块对 GPU 的访问也不是很友好。GPU 在访问连续内存时效率最高，因此我们对二维数组 arr[m][n] 进行扁平化处理。它会被存储为一个一维数组 flat_arr[m*n]。访问二维数组中的元素 arr[i][j] 对应于扁平化数组中的 flat_arr[i*n + j]。

```
  int blockSize = 256;
  int gridSize = (count + blockSize - 1) / blockSize;
  GenerateLastSegmentKernel_FlatOutput<<<gridSize, blockSize>>>(d_prefix, d_values,
  d_output_flat, count, guess_prefix.length(), total_len_per_guess);

```

此部分为线程配置并启动核函数。每个线程块包含 256 个线程，这是根据本次实验所用的显卡型号，经过查询设置的。根据 blocksize 确认 gridsize，确保覆盖所有可能的 segment。下面的部分就是核函数调用的主要部分，调用了 GenerateLastSegmentKernel 函数用 < < < > > >(三重尖括号) 传入变量 gridSize 和 blockSize，后面的括号内传入核函数参数。

此部分相当于，使用 GPU 并行处理多个 pattern，尝试采用此种方式提高运行效率，但实际运行效果不佳，这个在后续结果分析会详细分析。

```
  char *h_output_flat = new char[count * total_len_per_guess];
  cudaMemcpy(h_output_flat, d_output_flat, count * total_len_per_guess * sizeof(char),
```

```
3   cudaMemcpyDeviceToHost);
4
5   for (int i = 0; i < count; i++) {
6       guesses.emplace_back(std::string(&h_output_flat[i*total_len_per_guess]));
7       total_guesses += 1;
8   }
```

此部分我们构建主机端的结果数组 h_output_flat，从 GPU 获取结果并处理，通过 cudaMencpy 从 GPU 向 CPU 拷贝所有得到的结果，将所有结果通过循环，使用 emplace_back 将生成的口令放入猜测优先队列。最后结束记得回收内存，完成所有的口令生成。[2]

# 3 实验结果与优化

## 3.1 实验结果

在本次实验中，我们依旧沿用上次 MPI 的数据信息，开启编译器 O1 优化，由于 CUDA 编译器与之前 SIMD 使用的 neon 指令不兼容，因此哈希部分采用串行算法，未实现并行加速。最终得到的实验结果如下：串行时 guess time：0.418593，crack：358217；CUDA：0.466957，crack 值不稳定，大约稳定在 155000 左右，但是有极好情况 crack 达到 349692。我们发现使用 CUDA 试图并行加速效果却并不好，准确率虽然与串行相比在同一数量级但仍然是大幅下降的。

## 3.2 优化尝试

考虑到可能是 CPU、GPU 之间的通信影响效率，我对 guessing 函数做出优化，新增 CUDA_THRESHOLD 常数，设置为 8192，动态检测最后一个 pt 的个数，如果大于该常数，使用 CUDA 编程，否则使用 CPU 编程。但是最终实验结果提升不明显，crack 也没有大的增长。为了考虑是否是编译器优化的结果，我又尝试不开启编译器优化，但结果仍然提升有限。因此只针对这两个循环并不能有效提升并行效率。[3]

# 4 进阶尝试

在这一部分，我尝试在 GPU 上并行计算大量口令猜测的 MD5 哈希值，而不是在 CPU 上逐个计算。由于各个口令之间并不相关，且口令的哈希计算相比循环中的字符串拼接更复杂一些，这正好可以充分发挥 GPU 的能力。通过引入 CUDA 核函数实现，大大提高了哈希计算的效率。在 q.guesses 达到一定数量（大于 1000 个）时，将这些口令猜测批量发送到 GPU 进行处理，减少了 CPU 和 GPU 之间的数据传输开销。这里我将概括地讲述实现的大致思路。

## 4.1 md5.cu

这部分重写了 md5.cpp 并改为可以支持 CUDA 的 md5.cu。此部分主要为两个函数，与原来的代码框架一样，分别为预处理函数和哈希运算函数，具体代码参考 [4]GitHub 链接。预处理函数包装在下面核函数中 ___device___ Byte* StringProcess_device(const char* input_str, int input_length, int* n_byte, char* buffer) ，此函数是在 GPU 端运行的，传入的参数未输入字符串、长度、转换后的字节长度、缓冲区等。第二个 md5 哈希函数包装在下面核函数中。

```
__global__ void MD5Hash_kernel(const char* d_inputs, int* d_input_lengths,
bit32* d_results, int num_inputs) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_inputs) {
        int input_length = d_input_lengths[idx];
        const char* input_str=d_inputs+(idx > 0 ? (d_input_lengths[idx-1] + 1) : 0);

        // Calculate offset for current string in the d_inputs buffer
        size_t current_offset = 0;
        for (int i = 0; i < idx; ++i) {
            current_offset += d_input_lengths[i] + 1;
        }
        input_str = d_inputs + current_offset;
        int messageLength;
        Byte* paddedMessage = StringProcess_device(input_str, input_length,
        &messageLength, padded_message_buffer);

        int n_blocks = messageLength / 64;
        //下面为哈希计算部分，与 GPU 变成无关故省略

```

它的参数分别为 d_inputs：设备端输入字符串缓冲区（所有字符串连续存储）;d_input_lengths：各输入字符串的长度数组;d_results：存储 MD5 结果的缓冲区;num_inputs：输入字符串的总数。在这里分配线程块，针对每一个口令有一个独立的 idx，同样通过 block、thread 计算得到。再进行一系列填充、处理，包括调用 StringProcess_device 函数之后，最后对他们分组处理。

综上，两个核函数的基本逻辑与原来的代码逻辑几乎一致，只是在参数、空间分配上略有不同。

## 4.2 main.cu

由于涉及调用核函数,这里改后缀为.cu。当待处理的猜测密码数量达到 1000 个及以上时启用 GPU 加速运行，以提高处理效率。首先，程序计算所有猜测密码的总长度，并为设备端分配三块内存：连续存储所有口令的输入缓冲区、记录各密码长度的整数数组，以及存储哈希结果的输出缓冲区（每个 MD5 结果为 4 个 32 位整数）。随后，主机端将所有密码及其长度信息整理到扁平化缓冲区，并同步传输至设备端。CUDA 核函数配置为每个线程处理一个密码，通过块大小和网格大小动态调整并行度。核函数内部会对输入字符串进行 MD5 哈希，并将结果写入输出缓冲区。主机端等待 GPU 完成计算后，将结果拷贝回 CPU，并验证攻破率，同时将所有哈希结果格式化输出到文件。

```
if (q.guesses.size() >= 1000) // 达到一定数量才进行 GPU 处理 {
        auto start_hash = system_clock::now();
        int num_guesses = q.guesses.size();
        // 为 GPU 准备输入数据
        char* d_inputs;
```

```
    int* d_input_lengths;
    bit32* d_results;

    size_t total_input_length = 0;
    for (const string& pw_str : q.guesses) {
        total_input_length += pw_str.length() + 1; // +1 for null terminator
    }

    cudaMalloc(&d_inputs, total_input_length * sizeof(char));
    cudaMalloc(&d_input_lengths, num_guesses * sizeof(int));
    cudaMalloc(&d_results, num_guesses * 4 * sizeof(bit32));

    vector<int> input_lengths(num_guesses);
    vector<char> inputs_buffer(total_input_length);
    size_t current_offset = 0;

    for (int i = 0; i < num_guesses; ++i) {
        const string& pw_str = q.guesses[i];
        input_lengths[i] = pw_str.length();
        memcpy(inputs_buffer.data() + current_offset, pw_str.c_str()
        , pw_str.length() + 1);
        current_offset += pw_str.length() + 1;
    }

    cudaMemcpy(d_inputs, inputs_buffer.data(), total_input_length*sizeof(char),
    cudaMemcpyHostToDevice);
    cudaMemcpy(d_input_lengths, input_lengths.data(), num_guesses*sizeof(int),
    cudaMemcpyHostToDevice);

    // 设置 CUDA 核函数的启动配置
    int blockSize = 256;
    int gridSize = (num_guesses + blockSize - 1) / blockSize;

    // 调用 CUDA 核函数
    MD5Hash_kernel<<<gridSize, blockSize>>>(d_inputs, d_input_lengths,
    d_results, num_guesses);
    cudaDeviceSynchronize(); // 等待 GPU 完成计算

    // 将结果从 GPU 拷贝回 CPU
    vector<bit32> h_results(num_guesses * 4);
    cudaMemcpy(h_results.data(), d_results, num_guesses * 4 * sizeof(bit32),
    cudaMemcpyDeviceToHost);
```

```
48
49          // 处理结果并写入文件
50          for (int i = 0; i < num_guesses; ++i) {
51              if (test_set.count(q.guesses[i])) {
52                  cracked += 1;
53              }
54              for (int i1 = 0; i1 < 4; i1 += 1) {
55                  outfile << std::setw(8) << std::setfill('0') << std::hex
56                  << h_results[i * 4 + i1];
57              }
58              outfile << std::endl;
59          }
60
61          // 释放 GPU 内存
62          cudaFree(d_inputs);
63          cudaFree(d_input_lengths);
64          cudaFree(d_results);
65
66          // 在这里对哈希所需的总时长进行计算
67          auto end_hash = system_clock::now();
68          auto duration = duration_cast<microseconds>(end_hash - start_hash);
69          time_hash += double(duration.count()) * microseconds::period::num /
70          microseconds::period::den;
71
72          // 记录已经生成的口令总数
73          history += curr_num;
74          curr_num = 0;
75          q.guesses.clear();
76      }
```

### 4.3 进阶实验结果

在进阶实验中，并没有再尝试对两个循环进行并行处理，仅仅针对哈希做 CUDA 并行优化，实验仍然采用 O1 编译器优化，我们在这部分主要比较 hash time 和 crack 的值，guess time 做次要参考。最终结果如下：在串行实验中，hash time 为 24.7519s，crack 为 358217，guess time 为 0.418593s；CUDA 并行结果，hash time 为 5.50553s，crack 为 316369，guess time 为 0.798591s。我们发现 hash 的性能大幅度提升，提升约 4.496 倍，同时 crack 没有大幅度下降，在可接受范围内，猜测时间相比之前增加了一点，但影响不是很大。综上，根据实验数据，对 md5 哈希的 CUDA 并行加速效果很好，说明 GPU 的多核心模式完全可以加速多条口令哈希生成的任务。

# 参考文献

[1] CUDA 编程
[2] 基础要求原代码
[3] 基础要求优化版原代码
[4] 进阶要求原代码