南開大學
Nankai University

计算机学院

并行程序设计期末报告

实验总结

姓名：王泽舜

学号：2310655

专业：计算机科学与技术

2025 年 7 月 6 日

目录

1 SIMD 算法优化哈希部分

1.1 原算法原理

MD5 算法核心流程分为四个处理阶段（Round），每轮包含 16 次相似操作：

- **消息填充**：将输入补足为 512bit 的整数倍（具体由 StringProcess 完成）
- **初始化**：初始化 state[][],x[i],abcd
- **四轮处理**：共 64 步非线性运算
- **结果输出**：组合结果生成 128 位哈希值

1.2 改造点分析

原始代码存在以下可并行化部分：

文件	原始实现	改造内容
main.cpp	单次处理 1 个口令	改为 4 口令批量处理
md5.h	标量位运算	NEON 向量化指令
md5.cpp	串行四轮运算	SIMD 并行四轮

表 1: 代码改造对照表

1.3 关键技术实现

我们需要修改两部分，第一部分是核心的哈希函数的向量化，第二部分是中间变量修改为并行版本，这就需要回去找 neon 指令集的对应函数

1.3.1 核心运算向量化

采用 ARM NEON intrinsics 实现：

Listing 1: FF 函数的并行版本

```
#define FF_NEON(a, b, c, d, x, s, ac) { \
    a = vaddq_u32(a, vaddq_u32( \
        vaddq_u32(F_NEON(b, c, d), x), ac)); \
    a = ROTATELEFT_NEON(a, s); \
    a = vaddq_u32(a, b); \
}
```

1.3.2 消息块处理

并行加载 4 个消息块，将 ff 等函数的七个参数全部向量化：

Listing 2: 同步修改了 stringprocess 函数，用以填充 paddedMessage 数组

```
    paddedMessage = new Byte *[4];
```

```
    for (int i = 0; i < 4; i += 1) {
        paddedMessage[i] = StringProcess(inputs[i], &messageLength[i]);
    }
```

Listing 3: 加载一些中间变量

```
            // 为4个并行哈希初始化MD5常量
            uint32_t initial_state[4][4] = {
                {0x67452301, 0x67452301, 0x67452301, 0x67452301},
                ......
            };
            // 复制到输出状态数组
            for (int i = 0; i < 4; ++i) {for (int j = 0; j < 4; ++j) {
                    state_neon[i][j] = initial_state[i][j];}}
            // 加载到NEON寄存器
            uint32x4_t a = vld1q_u32(state_neon[0]);
            ......
            //并行加载原先的x[16]，算法和组合方式照搬
           uint32x4_t x_neon[16];
    for (int i1 = 0; i1 < 16; i1++) {
        // 计算4个消息块用来初始化的四字节的地址
        Byte* msg0 = &paddedMessage[0][i*64 + i1 * 4];
        ......
        // 并行加载
        uint32x4_t x0 = vmovq_n_u32(msg0[0] | (msg1[0] << 8) | (msg2[0] << 16) |
            (msg3[0] << 24));
        ......
        // 组合结果
        x_neon[i1] = vorrq_u32(
            vshlq_n_u32(x3, 24),
            vorrq_u32(vshlq_n_u32(x2, 16),
            vorrq_u32(vshlq_n_u32(x1, 8),x0)));}
```

Listing 4: hash 函数具体执行示例

```
    // sx和立即数也都向量化
    FF_NEON(a, b, c, d, x_neon[0], vdupq_n_u32(s11), vdupq_n_u32(0xd76aa478));
    ......
```

优化后的哈希时长在总指令量为 1000w 的情况下从 3.00787s 优化到了 2.36285s，加速比为 1.27x。

本次实验中，主要改进在于将原有的标量位运算改为向量化处理，利用 NEON 指令集实现了 FF、GG 等函数的并行化。通过对消息块的并行加载和处理，显著提升了 MD5 算法的执行效率。

而并不涉及修改算法，主要工作量在于理清和维持中间量的传递关系，还有去查 neon 指令集的官方文档找替换函数

2 多线程并行化实验

本实验要改进的核心算法本质是在密码猜测过程中，对 PT(猜测模板) 的最后一个 segment 进行具体的填充。以 L_8D_2 为例，经过前序优先队列的初始化和不断迭代，L_8 部分已被实例化，剩余的 D_2 需要在此处补全。假设模型统计得到 12、11、23 这三个具体的值，则本循环会依次将这三个值填充到 D_2，从而生成三个新的口令猜测。

该过程在代码中表现为如下循环：

Listing 5: 核心循环伪代码

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

考虑并行化上述循环，当我们将上述任务划分成多线程时，基本地，需要考虑以下几点：

1. 如果是静态的分配任务，分到最后剩余的部分很可能少于线程数，这时需要考虑如何处理剩余任务。我们在代码中 `int num_threads = std::min(4, n);` 来避免上述情况

2. 多个线程需要访问共享数据（如 guesses），如何确保数据的一致性和线程安全：我们考虑提前将 guesses 数组扩容到目标大小，每个线程负责写入自己负责的区间，这样在保证安全的同时也避免了更多的分配内存操作。。

3. 如果只考虑将这个循环并行化，由于任务粒度太细，可能创建销毁线程的开销大于优化的收益，因此可以尝试 PT 层面的并行

2.1 for 循环并行优化代码分析

2.1.1 基于 pthread 的并行优化实现

Listing 6: pthread 并行实现关键代码片段

```
struct ThreadArg2 {
    segment* a;
    std::string guess_prefix;
    int start, end;
    std::vector<std::string>* guesses_ptr;
    int offset;
};
//我们将前缀和指针、偏移量传入线程函数中，在内部计算要写入结果的位置
// 线程函数
void* thread_func2(void* arg) {
    ThreadArg2* t_arg = (ThreadArg2*)arg;
    for(int i = t_arg->start; i < t_arg->end; ++i) {
        (*t_arg->guesses_ptr)[t_arg->offset + (i - t_arg->start)] =
            t_arg->guess_prefix + t_arg->a->ordered_values[i];
    }
    return nullptr;
}
```

```
18
19 // 主线程分配任务
20 int n = pt.max_indices[pt.content.size() - 1];
21 int num_threads = std::min(4, n);//处理剩余量
22 size_t old_size = guesses.size();
23 guesses.resize(old_size + n);//结果扩容
24 ......
25 for(int t = 0; t < num_threads; ++t) {
26     .....//创建线程
27 }
28 for(int t = 0; t < num_threads; ++t) {
29     if(args[t].start < args[t].end)
30         pthread_join(threads[t], nullptr);}
31 total_guesses += n;
```

2.1.2 总结

通过 pthread 并行优化，PCFG 口令生成算法在多核环境下能显著提升生成速度，尤其适合大规模口令空间的遍历任务。优化后的实现既保证了线程安全，又减少了内存分配和拷贝的开销，是高效并行编程的典型范例。

基础实验的主要工作量是熟悉 pthread 的代码格式，同时在前期改代码但难以实现加速的过程中也让我对计算、分配内存、线程开销等成本有一定意识，最后通过优化减少分配内存次数实现了加速。

2.2 OpenMP 并行化设计思路与代码实现对比分析

OpenMP 并行化设计思路：为消除多线程竞争和提升效率，采用如下并行化策略：

- **空间预分配：**先由主线程 (或 #pragma omp single) 将 guesses 扩展到最终所需大小，记录插入起始下标。
- **分段并行写入：**用 #pragma omp parallel for，每个线程写自己负责的区间，避免 push_back 带来的锁竞争。
- **总数累加优化：**total_guesses 只需在并行循环后由主线程统一加上本轮生成的数量。

Listing 7: OpenMP 并行实现关键代码

```
1  int n = pt.max_indices[pt.content.size() - 1];
2  int start_idx;//计算插入起始下标
3  #pragma omp single
4  {
5      start_idx = guesses.size();
6      guesses.resize(start_idx + n);
7  }
8  #pragma omp parallel for
9  for(int i = 0; i < n; ++i) {
10     guesses[start_idx + i] = guess + a->ordered_values[i];
11 }
```

```
12 #pragma omp single
13 {
14     total_guesses += n;
15 }
```

3 MPI 多进程并行化设计与实现

为了将这个串行的过程在可接受的非严格 pt 概率序下并行化，采用 MPI 实现多进程并行。主要思路如下：

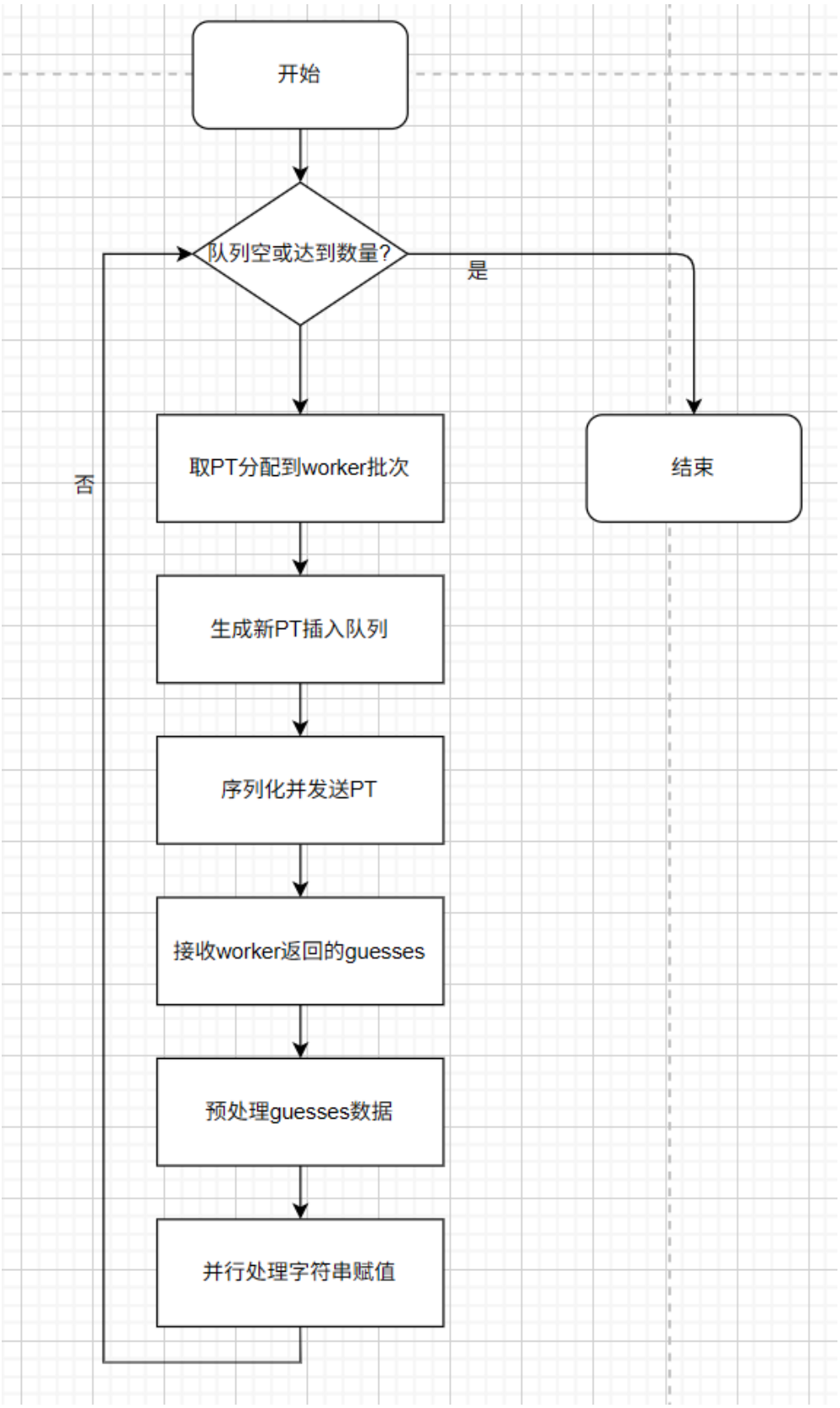

图 3.1: MPI 并行密码生成流程示意图

如图3.1所示，该流程主要包含以下步骤：

1. **主进程（rank 0）**负责 PT 任务分发和结果收集, 以及原来的优先队列维护。
2. **Worker 进程（rank > 0）**每次接收一批 PT，批量 generate guesses 并返回主进程（原先的 popnext 的事主进程做，worker 只 generate PT）。
3. **任务分发：**主进程每轮从优先队列顶部取出 k*batch 个 pt，分发给各 worker。
4. **结果统计：**每个 PT 生成的 guesses 会发回主进程，主进程将结果重新生成成字符串再赋值给 guesses 的对应位置。

更细致的解释见代码及注释：**主流程核心代码与具体思路分析**如下：

Listing 8: MPI 并行密码生成算法核心代码

```
while (!q.priority.empty() && pt_idx < k * batch_size) {
    // 1. 从优先队列中取出batch_size个PT
    worker_batches[pt_idx % k].push_back(q.priority.front());

    // 2. 按照popNext()的逻辑生成PT并插入到优先队列
    vector<PT> new_pts = q.priority.front().NewPTs();
    for (PT pt : new_pts) {
        // ... 照抄popNext()代码，除了generate guesses的部分
    }

    // 3. 序列化PT并发送给worker
    for (int j = 0; j < batch_cnt; ++j) {
        std::string pt_str = serialize_PT(worker_batches[i][j]);
        int len = pt_str.size();
        MPI_Send(&len, 1, MPI_INT, i + 1, 0, MPI_COMM_WORLD);
        MPI_Send(pt_str.data(), len, MPI_CHAR, i + 1, 0, MPI_COMM_WORLD);
    }

    // 4.接收worker返回的guesses，生成字符串并插入到guesses
    // 4.1 接收数据
    MPI_Recv(&total_guess_count, 1, MPI_INT, i + 1, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    std::vector<char> buf(total_guess_count * 64);
    MPI_Recv(buf.data(), total_guess_count * 64, MPI_CHAR, i + 1, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

    // 4.2 处理guesses
    // 预计算所有字符串长度和总字符数，以便接下来算每个guess字符串的位置
    std::vector<size_t> lengths(total_guess_count);
    for (int j = 0; j < total_guess_count; ++j) {
        lengths[j] = strnlen(buf.data() + j * 64, 64);
    }

    // 优化push_back：直接resize为原先长度+worker返回长度后
    //将缓冲区中计算好位置的字符串赋值给结果
    //基本mpi框架搭起来之后这部分一直是性能瓶颈
    //虽然改到现在这种方法但也没有做到加速
```

```
    size_t old_size = q.guesses.size();
    q.guesses.resize(old_size + total_guess_count);

    // 并行处理字符串赋值
    #pragma omp parallel for
    for (int j = 0; j < total_guess_count; ++j) {
        q.guesses[old_size + j].assign(buf.data() + j * 64, lengths[j]);
    }

    pt_idx++;
}
```

Worker 进程：基本上就是收批次长度、一批 pt，反序列化，generate，然后返回 guesses 给主进程。

3.1 并行算法性能分析与瓶颈诊断

表 2: 不同算法生成 3000wguesses 的时间性能对比（单位：秒）

版本类型	测试次数	平均猜测时间	猜测时间标准差	平均哈希时间	平均训练时间	MPI 通信占比
带分块计时 MPI	10	3.58	2.87	11.35	18.47	68.7%
无分块计时 MPI	10	3.34	0.42	9.31	14.95	-
原始串行	10	1.28	0.30	8.15	15.59	-

表 3: Performance Metrics Across Runs

Run	Guess Time (s)	Deserialization Time (s)			MPI Send Time (s)	MPI Receive Time (s)		
		Worker 1	Worker 2	Worker 3		Total	Pure Receive	Data Processing
1	2.84876	0.0109539	0.0008191	0.0104925	0	2.35637	1.66293	0.639313
2	8.95342	0.0157132	0.0278985	0.0238563	0.015377	7.38819	4.73557	2.35069
3	11.5527	0.0186231	0.0079665	0.0317148	0.039071	9.88056	7.18948	2.40118
4	8.47527	0.0209938	0.0056679	0.0165048	0.016278	7.45988	5.65764	1.56891
5	4.07106	0	0.0116256	0	0.028064	3.19623	2.14647	0.927773
6	3.61226	0.0005115	0.0044082	0.0007496	0.012729	2.88957	1.98154	0.822191
7	3.65355	0.003308	0.0047705	0.0048067	0.014032	2.99156	2.03329	0.833825
8	3.46606	0.0040221	0.0108211	0.0056186	0.009239	2.83929	1.94573	0.796273
9	3.51646	0.0033345	0.0134644	0.0021544	0.011614	2.87988	1.92751	0.820323
10	3.34775	0.0112826	0.0164895	0.0165029	0.015583	2.69302	1.83096	0.739929
Avg.	5.3497	0.0089	0.0110	0.0112	0.0161	4.4575	3.1114	1.1842

其中主要性能瓶颈为主进程接收字符串，更细分之后其中接收和处理基本七三开，接收阶段的瓶颈可能是子进程导致的，但数据显示子进程的反序列化并不费时间，其他操作和串行相同。而字符串处理见第三节中，已经尝试在算法层面进行优化，但仍然开销很大。

后续做期末报告时候想到可以通过增加进程数来判断是不是任务分配不均导致完成时间方差太大，因此接收时间长，如果是这个原因的话，进程数增加会显著导致同一批 workers 个任务的接收卡壳时间进一步增长。

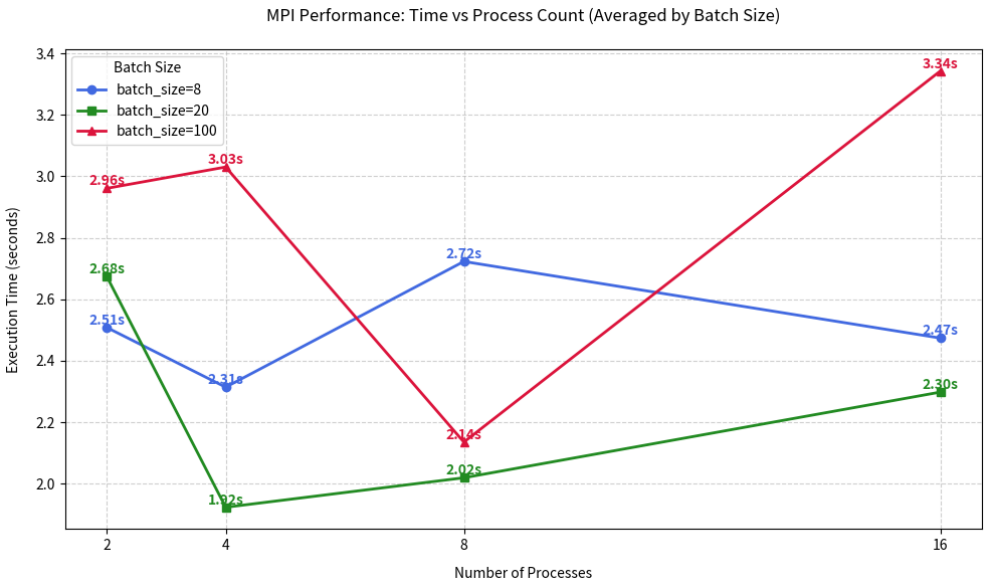

图 3.2: MPI 并行密码生成流程示意图

如图3.2所示, 选取的每批 pt 数在 20 左右最优，进程数为 4 最优。同时结果显示进程数增加确实会导致结果更不稳定，时间也略微增长，于是我们考虑在发送 PT 之前，根据 PT 要操作的那个 segment 的 indice 大小来当做任务量，均衡每个 worker 的任务量，来避免一个 worker 任务很重卡住所有 worker 被主进程接收这种情况。**关键修改如下：**

Listing 9: 基于工作量的 PT 负载均衡分配

```
std::vector<std::vector<PT>> worker_batches(k);
std::vector<size_t> worker_loads(k, 0); // 每个worker当前总工作量
int pt_idx = 0;
while (!q.priority.empty() && pt_idx < k * batch_size) {
    PT curr_pt = q.priority.front();
    // 计算PT的工作量
    size_t weight = 1;
    if (curr_pt.content.size() == 1) {
        weight = curr_pt.max_indices[0];
    } else {
        weight = curr_pt.max_indices[curr_pt.content.size() - 1];
    }
    // 找到当前负载最小的worker
    int min_worker = 0;
    for (int w = 1; w < k; ++w) {
        if (worker_loads[w] < worker_loads[min_worker]) min_worker = w;
    }
    worker_batches[min_worker].push_back(curr_pt);
    worker_loads[min_worker] += weight;

    // 生成新PT并插入优先队列（略）
    // ...
    q.priority.erase(q.priority.begin());
    pt_idx++;
```

```
}
```

表 4: 均衡负载后生成 3000 万 guesses 的 Guess/Hash/Train 时间（单位：秒）

Guess Time	Hash Time	Train Time
2.06185	4.93675	9.77798
2.29269	5.54244	9.95582
2.09226	4.59964	9.71931
1.99932	4.56900	9.69441
2.12546	5.03927	10.14700

结果显示，在任务分配均衡的情况下，接收时间和处理时间的方差都显著降低，不再出现之前偶尔会跑到 11s 的情况。平均接收时间也有所下降，但仍然是整体性能瓶颈。

3.2 总结

本实验通过 MPI 实现了 PCFG 口令猜测框架的高效多进程并行，主进程批量分发 PT 任务，worker 并行生成 guesses 并返回，支持每个 PT 生成量的统计与分析。针对 PT 任务“重量”极不均衡的问题，采用动态、细粒度 (具体到每个 PT) 的任务分配策略，以充分发挥并行计算资源，提高整体效率。

4 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（1.85s/1000w guesses）。

在做期末报告时，我们进行了更细致的分段计时，注意到预处理中分配内存的时间占用了约 0.4s，检查代码发现我们在主机端为存放 ordered_values 时候分配数组是在 for 循环中对每个 value 进行的，改为计算好总长度后一次性分配可以节省 0.3s 的时间，在改进之后我们提升到了 (1.26s/1000w 猜测)。

以下是具体的实现代码，分为预处理及其优化，分配和拷贝内存，调用核函数，收集结果

Listing 10: 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());
        }
//在期末报告中改进为：
char *flat_values = new char[n * (value_len + 1)];
        for (int i = 0; i < n; ++i) {
            memcpy(flat_values + i * (value_len + 1), a->ordered_values[i].c_str(),
                value_len + 1);
        }
// 2. 分配device内存并拷贝数据
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：（主要工作量是计算下标）
__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
// ...
```

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

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 cuda 并行化实验总结

本次实验中，我们认识了如何用 cuda 编程对简单计算密集型的任务进行并行处理，相比多线程，cuda 能够更细致地控制内存分配和数据传输，充分利用 GPU 的并行计算能力。

同时我们也对内存分配，访存优化等更底层的优化空间有了一定了解。

5 总结

本学期的并行实验中，我们从 simd 编程通过指令替代来优化和 openMP 练手开始，在 pthread 和 MPI 中通过亲自操作，学到了线程安全性如何保证，线程任务负载如何均衡。在 cuda 中更深入地体会到内存访问机制的巨大优化空间。在学习这些并行化代码结构的同时，也对程序从编译到运行整个过程中开销所在有了一定的认识。知道了有什么办法去探测、优化这些开销。

期末所做的新内容为，mpi 中测试 batch 值和进程数对性能的影响，均衡任务负载减小方差，cuda 编程中优化内存分配