南開大學
Nankai University

计算机学院

并行程序设计实验报告

实验六：GPU 编程（口令猜测）

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 7 月 2 日

目录

1 问题分析

在密码猜测系统中，我们需要根据 PCFG 模型生成大量的密码候选。传统的 CPU 串行处理方式在处理大规模字符串生成时效率较低，主要瓶颈在于：

- **字符串拼接操作频繁，CPU 处理速度有限**
- **内存分配和释放开销大**
- **无法充分利用现代 GPU 的并行计算能力**

在本次实验中，我们需要**基于 GPU 实现口令猜测算法的并行化**。

2 并行算法设计

2.1 并行算法思路

1. 初始化 CUDA 环境，分配 GPU 内存资源
2. 主线程维护优先级队列，存储待处理的口令节点
3. 主线程根据阈值判断，将任务按批次分配给 GPU 或 CPU
4. GPU 工作线程接收任务后，并行执行字符串拼接和概率计算
5. GPU 计算完成后，异步传输结果回主线程，更新全局队列
6. 重复任务分发与结果合并过程，直至满足终止条件
7. 所有计算完成后，释放 GPU 资源，清理 CUDA 环境

通过以上步骤，我们通过有效利用 CUDA 计算资源，通过阈值优化、异步处理和内存池管理，从而实现了口令猜测的 GPU 加速。

2.2 并行算法具体实现

```
#define GPU_ACCELERATION_THRESHOLD 1000
#define CHECK_CUDA_ERROR(operation) do // CUDA运行时错误检测宏定义
```

首先我们进行了两个宏定义。GPU_ACCELERATION_THRESHOLD 定义了 GPU 加速处理的数据量阈值，当待处理数据量达到 1000 时启用 GPU 并行计算，否则就使用 CPU 串行处理。通过这个，当数据量比较小的时候，我们就会采取 CPU 计算，避免了 GPU 的大幅度开销。第二个宏定义用于监测程序是否出错。

```
__global__ void string_concatenation_kernel // CUDA核心函数：字符串拼接处理
```

在这里我们定义了一个函数，GPU 并行字符串拼接核函数。该函数的实现主要是，每个 CUDA 线程负责处理一个字符串的拼接操作，首先会获取线程索引，确定要处理的数据项，然后将前缀字符串与输入字符串合并生成完整的口令候选，并且要注意在最后添加字符串结束符。通过并行执行实现高效的批量字符串处理。

```
class CudaComputeManager {// GPU计算资源管理器
private:
    // 内存容量限制常量
    // CUDA设备内存指针
public:
    CudaComputeManager() // 分配GPU内存空间
    ~CudaComputeManager() // 释放GPU内存资源
    bool process_string_batch    // 批量处理字符串拼接任务
 };
```

然后在这里我们定义了一个 GPU 资源管理类，用于对 CUDA 进行内存分配、数据传输和核函数执行的统一管理。其中构造函数会通过类中变量分配 GPU 内存空间，输入空间为 60MB，输出空间为 120MB，通过 cudaStreamCreate 创建 CUDA 流用于异步操作管理，还会设置一些数组、变量等。析构函数会释放所有 GPU 内存资源并销毁 CUDA 流。

其中的 process_string_batch 函数是我们实现的核心。伪代码如下：

```
function process_string_batch(input_data, input_count, output_buffer):
    // 1. 输入验证
    if input_count > MAXIMUM_ITEM_COUNT:
        return error
    // 2. 计算内存需求
    total_input_size = 0
    for each item in input_data:
        total_input_size += item.length
    if total_input_size > MAXIMUM_INPUT_CAPACITY:
        return error
    // 3. 分配主机内存
    allocate host_input_buffer[total_input_size]
    allocate host_offset_array[input_count]
    allocate host_length_array[input_count]
    allocate host_output_buffer[MAXIMUM_OUTPUT_CAPACITY]
    // 4. 数据组织
    current_offset = 0
    for i = 0 to input_count-1:
        copy input_data[i] to host_input_buffer[current_offset]
        host_offset_array[i] = current_offset
        host_length_array[i] = input_data[i].length
        current_offset += input_data[i].length
    // 5. 异步数据传输到GPU
    async_copy host_input_buffer to device_input_buffer
    async_copy host_offset_array to device_offset_array
    async_copy host_length_array to device_length_array
    // 6. 启动GPU核函数
    launch string_concatenation_kernel(
        device_input_buffer,
        device_offset_array,
        device_length_array,
        device_output_buffer,
```

```
        input_count
    )
    // 7. 异步传输结果回主机
    async_copy device_output_buffer to host_output_buffer
    // 8. 同步等待完成
    wait_for_gpu_completion()
    // 9. 处理结果
    copy host_output_buffer to output_buffer
    // 10. 清理资源
    free host_input_buffer
    free host_offset_array
    free host_length_array
    free host_output_buffer
    return success
end function
```

该函数主要负责实现字符串批量拼接的完整 GPU 加速流程。该方法首先进行严格的**输入验证**，检查输入字符串数量是否在预设的 MAXIMUM_ITEM_COUNT 范围内，并计算所有字符串的总字节数是否超过输入容量的限制，确保不会超出 GPU 内存容量。接着，该函数会在主机内存中准备三个很重要的**数据结构**：host_input_data 连续存储所有字符串的原始内容，并通过**扁平化处理**消除字符串间的间隙；host_offset_data 记录每个字符串在连续内存中的起始位置；host_length_data 存储各字符串的长度信息，这三个数组共同构成了 GPU 处理所需的完整输入描述。随后我们根据字符串数量和预设的最大字符串长度来**计算输出缓冲区大小**，并检查是否超出最大输出容量，以确保输出空间充足。

在这个函数的核心部分，我们采取了**异步流水线设计策略**：首先通过 cudaMemcpyAsync 将输入数据、偏移数组和长度数组并行拷贝到设备内存，同时处理前缀字符串的拷贝；然后配置 CUDA 内核的执行参数，并启动我们之前实现的 string_concatenation_kernel 核函数进行并行拼接；最后异步将结果拷贝回主机。

在整个过程中，所有操作都绑定到同一个 CUDA 流中从而实现操作序列化，并通过 cudaStreamSynchronize 确保所有异步操作完成。在最后的结果处理阶段，我们将设备返回的扁平化结果数据重新组织为 vector<string> 的格式，将每个字符串从结果缓冲区的固定偏移位置提取，并在最后清理临时内存，然后返回成功状态。

通过这个函数，我们不仅从内存角度最大化了内存访问效率，而且通过异步操作的流水线设计降低了等待延迟，从而加速了计算能力。

另外特别值得注意的是，我们通过固定长度输出缓冲区的设计从而避免了动态内存分配，以实现更好的性能。

```
void initialize_cuda_environment()
void cleanup_gpu_resources()
// 对外提供的GPU处理接口
bool processWithGPU(const vector<string>& values, const string& prefix,
    vector<string>& results)
```

然后这里我们定义了几个接口函数，其中 initialize_cuda_environment 是 CUDA 环境初始化函数，用于创建我们上面定义的全局 GPU 资源管理器实例，确保 GPU 资源的正确初始化。cleanup_gpu_resources 是资源清理函数，在 main 函数中被调用，用于释放 CudaComputeManager 实例，回收所有 GPU 相关资源。processWithGPU 是对外的 GPU 处理接口函数，封装 GPU 批量处理逻辑，提供统一的 GPU

加速服务。确保 GPU 环境已初始化后再调用批量处理函数。

```
// ========== PCFG算法实现区域 ==========
void PriorityQueue::CalProb(PT &pt)
void PriorityQueue::init()
void PriorityQueue::PopNext()
vector<PT> PT::NewPTs()
void PriorityQueue::Generate(PT pt)
========== for循环 ==========
        int num_values = pt.max_indices[0];
        if (num_values >= GPU_ACCELERATION_THRESHOLD) {
            vector<string> temp_results;
            if (processWithGPU(a->ordered_values, "", temp_results)) {
                guesses.insert(guesses.end(), temp_results.begin(),
                    temp_results.end());
                total_guesses += temp_results.size();
                return;}}
        // GPU处理失败或数据量不足，使用CPU处理
        for (int i = 0; i < num_values; i++){
            guesses.push_back(a->ordered_values[i]);
            total_guesses++;}
```

以上是我们 PCFG 算法的实现区域，我们在这里主要说吗 Generate 函数中的修改，主要修改也就是在**检查阈值**后，首先会去尝试进行 GPU 处理，当数据量很小的时候，便会回退到 CPU 处理，从而减少了 GPU 分配内存的开销，提高了性能。

2.3 并行算法的正确性验证

```
Guesses generated: 11654409
Guess time:6.42006seconds
Hash time:9.95916seconds
Train time:62.8189seconds
Cracked:356278
s2312631@VM-0-43-ubuntu:~$
```

图 2.1: 并行算法正确性验证

我们实现的并行算法的猜测正确数与串行算法完全相同，可以验证并行算法逻辑上的正确性。

2.4 并行算法的改进

在我实现了以上的并行算法之后，我发现并没有得到很好的效果，guess time 仍和串行差不多甚至更慢一些，为此我尝试了进阶选题，在尝试批量处理 pt 时，错写了一版代码，但没想到收获到了很好的效果。修改如下：

```
void PriorityQueue::PopNext()
{
    const int BATCH_SIZE = 64;// 批处理大小
    vector<PT> batch_pts;
    for (int i = 0; i < BATCH_SIZE && !priority.empty(); i++) {
        batch_pts.push_back(priority.front());
```

```
        priority.erase(priority.begin());
    }
    if (!batch_pts.empty()) {
        BatchGenerate(batch_pts);
    }
```

可以看到我们对 PopNext 函数进行了修改，每次批量处理 64 个 pt，调用 BatchGenerate 函数。

```
void PriorityQueue::BatchGenerate(const vector<PT>& batch_pts) {
    for (const PT& pt : batch_pts) {
        Generate(const_cast<PT&>(pt));
    }
}
```

这里对 BatchGenerate 函数进行了实现，其会调用 Generate 函数进行处理。但是，很明显可以发现，我们通过这两个函数，其实并没有实现批处理，本质上还是对于每一个 pt 通过 for 循环调用 generate 函数每次处理一个 pt，这是一个伪批处理的串行处理过程，但是在我进行测试后，在正确性得到保证的前提下，其性能明显提高，优于串行算法。为此，我进行了**一些思考：为什么同样是串行处理，后者却比前者效果好得多？**

首先，可以看到，这样处理之后，我们调用 PopNext 的次数要减少很多，那么就有可能是我们在 PopNext 函数中花费了很多时间。原来的代码中每处理一个 PT 就执行一次 priority.erase(priority.begin())，这是 O(n) 复杂度的操作。而 guessing_gpu.cu 批量删除 64 个元素，虽然单次成本更高，但**总体删除次数减少 64 倍，显著降低了队列维护开销**。

其次，也有可能是因为**批量数据的缓存友好性**，batch_pts 向量虽然逻辑上仍是串行处理，但 64 个 PT 对象在内存中连续存储，提高了 CPU 缓存的命中率。相比 example 文件每次只访问 priority.front()，批量访问能更好地利用缓存行，减少内存访问延迟。

另外，通过我们在计算机组成原理课上学到的**分支预测**也可以解释说明这一点，在修改后的代码中，统一的循环结构减少了复杂的条件分支，使 CPU 的分支预测器能更准确预测执行路径，减少流水线停顿。**从指令的角度**来看，BatchGenerate 函数中的简单 for 循环使得 Generate 函数的指令序列被重复执行 64 次，这些指令能长时间驻留在 CPU 指令缓存中，提高指令缓存命中率。**从调度的连续性角度**来看，批量处理的连续模式减少了操作系统上下文切换，从而收到了较好的效果。

从内存管理的角度来看，64 个 PT 在短时间内连续调用 processWithGPU，使 CUDA 运行时能更有效管理 GPU 内存分配释放，减少内存碎片化。并且**连续的 GPU 调用**能保持 GPU 活跃状态，避免频繁的空闲激活状态间的切换，从而提高整体吞吐量，减少了内存碎片和分配的开销。

3 性能对比

需要说明的是，由于我们在程序中设置了一系列参数，比如批处理大小、gpu 内存大小等，所以在性能对比时为了方便以及合理的比对，我们统一选择固有的参数大小，但可能会由于参数大小不合适导致性能差异较大，可能会有意想不到的情况出现。

对于 CPU 串行程序，采用编译命令为

```
g++ -std=c++11 -o pcfg_cpu main.cpp guessing.cpp md5.cpp train.cpp
```

并且运行串行时，要把 main 中的 cleanup_gpu_resources 部分注释掉。

对于 GPU 并行程序，采用编译命令为

```
nvcc -std=c++11 -o pcfg_gpu guessing_gpu.cu main.cpp md5.cpp train.cpp -lcudart
```

3.1 不开启编译优化的情况下

表 1: 串行 CPU 与并行 GPU

问题规模	串行 guess	并行 guess	guess 加速
500w	5.594s	3.267s	1.712
1000w	7.987s	6.424s	1.243
2000w	15.808s	11.733s	1.347
3000w	21.212s	18.126s	1.170

通过我们的优化，可以看到，在不开始编译优化的情况下，我们基于 GPU 的并行算法在时间性能上已经优于串行算法，这说明我们的 CUDA 函数确实发挥了优秀的作用。其中在数据规模为 500w 的情况下，加速比甚至达到了 1.712。

但是我们可以发现，按照常识来说，随着数据规模的增大，GPU 处理计算的能力应该越来越强，也就是加速比应该越来越高，但是根据实验数据可以看到事实并不是这样。我觉得出现这个情况有很多种原因，一个原因就是，我们在之前已经提到过，在程序中有很多**参数**，比如批处理的大小，分配的 GPU 内存大小等等，有可能是批处理太小，那么批次就会变多，会增加内存开销，频繁的内存分配和释放可能导致 GPU 内存碎片化，碎片化严重时，即使总内存足够，也可能无法分配连续的大块内存，GPU 内存太小，而导致在处理大数据时，原先分配的内存并不足够，可能会回退到 CPU 处理，所以优化效果并不如小规模数据。还有一个原因可能是数据规模增大时，**串行部分的比重增加**，从而导致并行加速比会下降。

3.2 O1 优化情况下

表 2: 串行 CPU 与并行 GPU

问题规模	串行 guess	并行 guess	guess 加速
500w	0.275s	0.394s	0.698
1000w	0.431s	0.648s	0.665
2000w	0.810s	1.103s	0.734
3000w	1.144s	1.601s	0.715

在 O1 的情况下，可以看到加速比小于 1，为负优化，说明在这种情况下我们的并行算法不如串行算法。那么也就说明了，O1 编译条件对于串行算法的优化效果明显优于并行算法。我认为是有以下几个原因引起的：首先，我们都知道，CPU 之所以强大，有很大一部分归功于其所具有的**分支预测功能**，在没开编译优化的情况下，分支预测失效，缓存利用率差，而在 O1 编译优化的条件下，串行算法的分支预测进一步增强，减少分支预测失败的开销，从而使得其优化效果更好一些。其次，对于串行算法来说，其循环次数更多，那么通过**循环展开或向量化**等，就可以加速对字符串的处理，通过函数内联优化也可以对频繁的函数调用进行优化。并且 O1 优化还会进行**指令重排**，会提高指令级并行度和流水线效率，从而使得串行算法优化效果很好。

而对于 CUDA 而言，**CUDA 编译器 nvcc 本身已经高度优化**，O1 编译优化对其提升较小（相对于 g++ 来说），并且我们详细分析可以发现，O1 优化主要在指令级进行优化，即减少指令条数，而**GPU 的性能瓶颈主要在内存带宽和并行度，而非指令级优化**，另外 CPU 和 GPU 之间的数据传输开销也不会因编译优化而减少。

3.3 O2 优化情况下

表 3: 串行 CPU 与并行 GPU

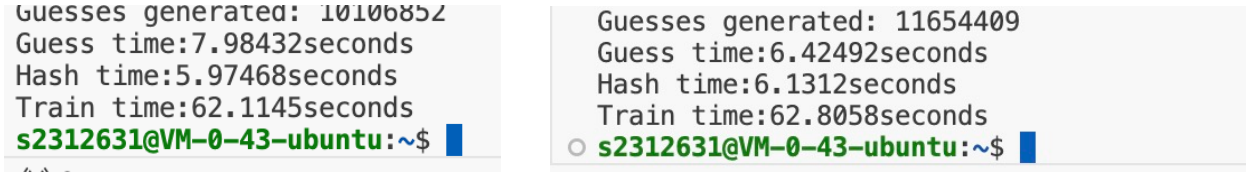
问题规模	串行 guess	并行 guess	guess 加速
500w	0.275s	0.394s	0.698
1000w	0.428s	0.651s	0.657
2000w	0.806s	1.078s	0.748
3000w	1.138s	1.576s	0.722

而在 O2 编译优化的条件下，可以看到与 O1 的情况差不多，GPU 并行算法不如 CPU 串行算法，具体原因就是我们上面分析到的那些。

基于以上的测试与观察，我们可以发现，GPU 不是一定优于 CPU，GPU 加速不是绝对的，具体要从内存开销、算法设计、系统资源、问题描述等多个角度进行合理的使用。

4 进阶要求的实现

4.1 实现相对串行算法的加速



(a) 串行 1000w 不开编译优化 (b) 并行 1000w 不开编译优化

图 4.2: 串行并行性能对比

在数据规模为 1000w，不开启编译优化的情况下，在共享服务器上得到以上数据，串行 guess time 为 7.98432s，并行 guess time 为 6.42492s，加速比为 1.2427，我们成功基于 GPU 完成了对 PCFG 的并行优化。

4.2 一次性从优先队列中取出多个 PT

在这一部分我们实现真正的多 pt 批量处理。伪代码如下：（源代码链接在最后）

```
function process_pt_batch_parallel(batch_pts, results):
    // 步骤1: 计算输出偏移量
    total_outputs = 0
    output_offsets = []
    for each pt in batch_pts:
        output_offsets.add(total_outputs)
```

```
        total_outputs += pt.last_segment.values_count
    // 步骤2: 收集所有segment数据
    all_segment_data = collect_all_segments(batch_pts)
    // 步骤3: 传输数据到GPU
    GPU_transfer_data(all_segment_data, output_offsets)
    // 步骤4: 启动GPU核函数
    launch_parallel_kernel(batch_pts.size, total_outputs)
    // 步骤5: 获取结果
    GPU_get_results(results)
    return success
```

process_pt_batch_parallel() 函数是 GPU 批量处理的核心控制器，承担着从 CPU 到 GPU 的数据组织和内存管理的职责。该函数首先通过遍历批量 PT 集合，精确计算每个 PT 最后一个 segment 能够产生的输出数量，然后建立一个偏移量数组来记录每个 PT 的输出应该放在哪里。在我们的设计中这一步骤至关重要，因为它解决了传统估算方法导致的内存越界问题（我在实验过程中遇到过很多次这个问题。接下来，函数会把所有的 segment 数据打包成一个连续的内存块，同时记录下每个数据块的位置、长度和类型等信息，通过把复杂的数据结构扁平化的操作极大地提升了 GPU 内存访问效率。在内存传输阶段，我们采用异步拷贝的策略，将主机端的复杂数据结构转换为 GPU 更好处理的线性数组，并且会根据实际需要动态分配输出缓冲区的大小，这样既保证了性能又避免了内存浪费。最终，函数启动 GPU 核函数并负责结果的回收和格式转换。

```
__global__ function parallel_pt_generation_kernel():
    pt_id = get_thread_id()
    if pt_id >= total_pts: return
    // 构建前缀
    prefix = build_prefix_from_segments(pt_id)
    // 获取输出偏移量
    output_offset = pt_output_offsets[pt_id]
    // 生成所有可能的字符串
    last_segment = get_last_segment(pt_id)
    for val_idx = 0 to last_segment.max_values:
        output_pos = output_offset + val_idx
        output_buffer[output_pos] = prefix + last_segment.values[val_idx]
```

parallel_pt_generation_kernel 这个核函数是真正在 GPU 上工作的部分，我们让每个 GPU 线程都有合适的工作量，同时避免内存访问冲突。具体来说，每个 GPU 线程会分配到一个 PT 来处理，线程编号直接对应 PT 的编号，这样分工就很明确了。这个核函数的工作分成两步：第一步是把前面几个 segment 的内容拼接起来形成前缀字符串，第二步是处理最后一个 segment 的所有可能取值。在第二步中，我们利用之前计算好的偏移量来确保每个线程写数据时不会互相干扰。这种做法的好处是把原来需要多层嵌套循环的复杂操作简化成了单层的并行处理，每个线程都可以独立工作，不需要等待其他线程，这样就大大提高了效率。另外，核函数还用了一些小技巧来优化性能，比如用局部数组来缓存前缀内容，减少对全局内存的访问次数，同时加入了边界检查来保证程序的稳定性。总的来说，通过这样的设计，我们在保证正确性的前提下，充分发挥了 GPU 并行计算的优势。

4.2.1 算法分析

多 pt 批量处理的算法核心逻辑分为三个层次：首先，通过 PopNext() 函数收集 64 个 PT 实例形成批次，利用批量处理减少 GPU 调用开销；其次，BatchGenerate() 函数根据传入批量大小智能选择处理策略，当批量达到 8 个以上时启用 GPU 加速，否则回退至 CPU 串行处理；最后，process_pt_batch_parallel() 方法实现在 GPU 端的并行计算，通过预计算每个 PT 的输出偏移量解决从而解决了内存访问冲突，并使用 parallel_pt_generation_kernel 核函数对每个 PT 的最后一个 segment 进行并行展开。

4.2.2 正确性验证

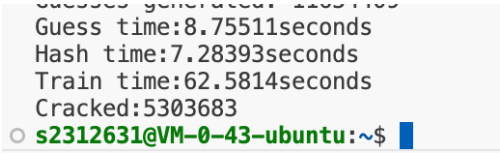


图 4.3: 多 pt 批量处理算法正确性验证

4.2.3 结果分析

但是，经过我们实验测试之后，发现这种算法的效率并不是很好，不如之前的伪 pt 处理。我又看了眼实验指导书，发现助教在指导书里写的第 3 个进阶选题应该就可以很好地解释这一点。

不同 PT 生成的输出数量差异很大，有些 PT 可能只生成几个密码，而有些可能生成几千个，这种不均匀的工作负载让 GPU 的并行优势发挥不出来，因为 GPU 最怕的就是线程间工作量不平衡。其次，我们为了解决内存越界问题引入的动态偏移量计算和精确内存分配，虽然保证了正确性，但也带来了额外的计算开销和复杂的内存管理逻辑。再加上每个 PT 的 segment 数据需要大量的元数据来描述，比如偏移量、长度、类型等，这些元数据的传输和处理也消耗了不少时间。

相比之下，之前的伪批量处理虽然看起来简单粗暴，但它避免了复杂的数据重组和内存管理，直接利用了简单的字符串拼接逻辑，反而在实际运行中更加高效。这也给我启示，不是一定越高级越复杂的算法就会有很好的效果，而是要看具体的问题、环境等，有时候简单的解决方案可能比复杂的优化更实用，特别是当数据特征不适合并行处理的时候。

4.3 修改参数

在 1000w 的数据规模下，不开编译优化的情况下，我们在 GPU_ACCELERATION_THRESHOLD 为 2000，输入大小为 30MB，输出大小为 60MB 得到最优结果，guess time 大概为 6.42s，而在其他任何数据规模组合下，我们测试后大概都在 6.67s 左右。

表 4: 修改参数

GPU 阈值	输入内存	输出内存	guess time
1000	20MB	40MB	6.693
2000	30MB	60MB	6.424
3000	40MB	80MB	6.647

另外值得一提的是，我们通过修改 PopNext 函数中的 BATCH_SIZE = 64 参数，也就是修改批处理大小，也会使得收到不同的效果，经过测试，我们选取 64 为最佳参数。

5 心得体会

5.1 一些收获

在这次实验中，我深刻体会到了 GPU 并行计算与传统 CPU 串行处理的本质区别。刚开始实现 CUDA 核函数时，我犯了一个典型错误——直接把 CPU 的串行逻辑搬到 GPU 上，结果性能反而更差了。通过反复调试才发现，**GPU 编程的核心在于重构算法**，要把问题分解成可以并行执行的独立任务。比如在字符串拼接核函数中，我最终设计成每个线程处理一个完整的字符串拼接，这样既避免了线程同步问题，又充分利用了 GPU 的并行能力。另外我也发现，**GPU 编程中内存操作的代价可能比计算本身更高**。

5.2 意外发现

最让我意外的是” 伪批量处理” 带来的性能提升。本来我想实现真正的并行批处理，结果代码写错了反而获得了更好的效果。经过分析发现是由以下原因造成的：

- **数据结构连续性带来的缓存优化**：连续访问 64 个 PT 对象显著提高了 CPU 缓存命中率
- **系统调用开销的减少**：批量处理减少了 CUDA 运行时环境的初始化和销毁次数
- **内存碎片的降低**：集中分配释放大块内存比频繁处理小块内存更高效

这个发现也让我意识到，有时候简单的架构调整可能比复杂的并行算法更有效。特别是在处理不规则工作负载时，过度追求并行化反而可能会适得其反。

另外我还了解到在 O1，O2 编译的条件下，分支预测和指令级并行优化使串行 CPU 代码性能大幅度提升，所以采取编译优化后，CPU 处理问题的速度比 GPU 快，而 GPU 的性能瓶颈不在于指令级优化，而主要在内存带宽和并行度。GPU 不是一定总是优于 CPU，GPU 加速不是绝对的，具体要从内存开销、算法设计、系统资源、问题描述等多个角度进行合理的使用。

6 代码链接

点击访问 GitHub 主页