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

期末课程报告

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 6 日

目录

1 本学期并行实验总结

1.1 国产处理器调研

在学期初刚刚接触并行化程序设计时，我首先对我国处理器的发展做出了详细的调研。

1.1.1 处理器类型介绍

在研究计算机系统核心组件时，我深入了解了三种主要处理器类型：CPU、GPU和APU，每种处理器在功能和应用场景上各具特色。CPU（中央处理器）是我眼中计算机的“大脑”，负责处理数据和执行控制逻辑。它通常配备4到16个物理核心，能够高效应对复杂的计算任务和系统管理需求。我注意到，现代CPU通过多核心设计和超线程技术显著提升了并行处理能力，特别适合需要高单线程性能或多任务处理的应用，比如运行操作系统或执行复杂算法。这种通用计算能力让我对CPU在各种场景中的广泛适用性印象深刻。

GPU（图形处理器）则展现了完全不同的设计理念。它专为并行计算打造，最初用于图形渲染，如今在科学计算、机器学习和数据分析领域大放异彩。我了解到，GPU拥有数千个小型计算单元，可以同时处理大量线程，尤其擅长矩阵运算和图像处理等高度并行的任务。与CPU注重单线程性能不同，GPU的架构更强调吞吐量，这让我认识到它在特定高并行场景中的独特优势。

APU（加速处理器单元）则让我看到了CPU和GPU融合的潜力。它将通用计算和并行处理能力集成到单一芯片上，优化了性能与功耗的平衡。我发现，APU在能效比和集成度上表现优异，非常适合移动设备和嵌入式系统，比如笔记本电脑或游戏主机。这种集成设计让我感受到它在功耗受限场景中的灵活性和高效性。

1.1.2 我国GPU技术发展分析

在研究我国GPU技术发展时，我深刻体会到自主研发在国家科技战略中的重要性。近年来，我国在GPU领域的研发取得了一些进展，特别是在高性能计算和人工智能应用中，国产GPU开始崭露头角。例如，我了解到一些国产GPU已在服务器和嵌入式设备中得到应用，这让我对国内技术进步感到振奋。然而，与国际领先厂商如NVIDIA和AMD相比，我注意到国产GPU在性能、软件生态和技术成熟度上仍存在差距。

我分析了国内GPU研发面临的挑战，包括核心架构设计的复杂性、先进制造工艺的限制，以及软件生态的构建。GPU的指令集优化和驱动程序开发需要与硬件高度匹配，这对技术积累和产业协同提出了很高要求。此外，国际技术封锁和市场竞争进一步加大了自主创新的难度。尽管如此，我看到政策支持和产业投资为GPU研发注入了动力。未来，我认为需要在芯片设计、制造工艺和软件生态方面持续突破，才能逐步缩小与国际领先水平的差距。这种挑战与机遇并存的现状让我对国产GPU的未来充满期待。

1.1.3 国产典型处理器结构与性能

在研究国产典型处理器时，我聚焦于其结构设计和性能表现。以龙芯3A6000系列处理器为例（材料中提到的“英特尔3A6000”可能为笔误），我发现这些处理器采用多核心架构，支持复杂的指令集，能够高效处理多任务环境。它们在设计上强调并行计算能力，同时在功耗控制和指令集优化方面进行了针对性改进。这让我感受到国产处理器在高性能计算领域的潜力。

通过分析SPEC CPU 2017测试数据，我观察到这些处理器在单线程和多线程任务中表现均衡，尤其在整数运算和浮点运算方面表现出色。材料附录中的性能数据表明，在不同核心数和频率配置下，国产处理器的性能在特定场景下已接近国际主流水平。例如，我注意到在机器学

习推理任务中（如结合ResNet50模型），这些处理器展现了较强的计算能力。然而，我也认识到在通用性和软件生态支持方面，国产处理器仍有提升空间。这让我对未来优化的方向有了更清晰的认识，同时也为国产处理器的进步感到自豪。

1.2 体系结构编程

1.2.1 基础要求

体系结构编程实验任务核心是对于n*n矩阵的向量内积以及对n个数相加进行并行优化。

基础要求部分，我实现了多种算法，并对其性能进行了深入测试和分析。针对N*N矩阵与向量内积，我们将逐列访问的“平凡算法”优化为采用逐行访问的“Cache优化算法”，通过充分利用矩阵的cache操作，每次在计算中尽可能从cache中读取数据，大大减少了内存访问的次数，从而提高了程序运行的效率。并通过QueryPerformanceFrequency和QueryPerformanceCounter函数进行精确计时，结果显示Cache优化算法在不同规模下均显著优于平凡算法。之后，我又使用VTune工具的Profiling分析进一步证实，Cache优化算法显著降低了L1、L2、L3 Cache的未命中次数，从而验证了其高效性。最终分析表明，随着矩阵规模的增大，Cache优化算法的优化比例也越高，最高可达80.2% 。

对于N个数求和问题，除了一个一个数相加的“平凡算法”外，我实现了“多链路算法”、“递归算法”和“二重循环算法” 。性能测试结果显示，多链路算法通过利用CPU超标量架构，有效降低了CPI（每条指令执行的周期数），从而实现了比平凡算法更快的运行速度，优化比例最高达到42.4% 。虽然递归算法和双重循环算法的CPI较低，但因其执行指令数过多，在较小规模下性能反而不及平凡算法。

1.2.2 进阶要求

进阶要求部分我探讨了更深层次的优化技术及大规模数据处理的挑战。循环展开优化通过减少循环迭代次数来降低循环控制开销，以N*N矩阵与向量内积为例，该优化技术显著提升了程序运行效率和Cache命中率，特别是在Cache优化算法中表现更为突出，N=10000时优化比例达到82.6% 。

我还验证了浮点数运算顺序对结果的影响，指出由于浮点数的近似表示和四舍五入规则，即使表面输出相同，不同运算顺序也可能导致最终结果不一致。在大规模矩阵运算方面，报告测试了20000x20000到50000x50000的矩阵，发现当矩阵规模从30000x30000增加到40000x40000时，运行时间不再与规模成比例增长。报告分析这可能是由于内存不足（内存墙挑战）导致的问题，因为40000x40000矩阵所需的内存（约12.8GB）超出了实验机器可用内存（不到8GB），迫使程序进行分块或碎片化处理，从而大幅降低了运行效率。

1.2.3 实验总结

在矩阵与向量内积计算里，平凡算法按列访问的平凡算法 Cache 命中率低，而按行遍历的 Cache 优化算法能利用空间局部性原理，提升 Cache 命中率与算法效率，VTune 分析结果也印证了这点。这让我明白，设计算法需考虑体系结构特性。同时，在 n 个数求和实验中，多链路算法通过双累加器并行计算，利用 CPU 超标量架构降低 CPI，提升计算效率。这表明合理设计并行算法，可充分发挥多核处理器优势。

接下来就进入了口令猜测选题的并行化研究中————

1.3 simd 编程

首先是对 MD5Hash 函数实现 SIMD 编程，在本次实验中，我需要将原来多次重复执行相同操作的串行算法向量化，使用 NEON 指令集中的相关指令，实现 SIMD 向量化编程。

1.3.1 基础要求

在基础要求部分，我首先对 MD5 哈希函数进行了 SIMD 优化。我修改了 md5.h 文件中的F和FF函数，将其优化为F_SIMD和FF_SIMD，通过将F_SIMD和ROTATELEFT融入FF_SIMD，我减少了函数调用开销，并使用NEON指令集（如uint32x4_t和vdupq_n_u32）对常量和向量进行了操作，实现了向量化计算。

接着，我对MD5Hash函数进行了多项修改以支持SIMD。我改变了其传入参数的规模，使其能够一次性处理batch_size（默认为4）个输入字符串数组，并将状态寄存器从标量改为了NEON的uint32x4_t向量。在输入字符串处理和填充方面，我为每个输入调用了 String Process，并分配了paddedMessages和messageLength数组，同时利用n_blocks数组来记录不同长度字符串所需的块数，并计算出max_blocks以适应不同长度输入。在状态初始化阶段，我使用NEON的 vdupq_n_u32指令初始化了四个uint32x4_t向量寄存器 state_a、state_b、state_c和state_d 。为了处理不同长度输入导致的无效块问题，我生成了mask_pool数组，并用0xFFFFFFFF标记有效块，0标记无效块。消息块处理方面，由于轮函数需要同时加载四个输入的同一字，我将填充数据转换为SIMD友好的交错存储格式，先将字节数据转为32位字并存储到 messages 数组，再将四个输入的对应字交错存储到interleaved数组。消息块加载和轮函数初始化时，我从interleaved加载了16个uint32x4_t向量到x[0..15]，并初始化了轮函数向量a, b, c, d 。轮函数处理时，我加载了mask数组以标记需要处理的消息，并执行了四轮（64步）SIMD轮函数，这些轮函数在md5.h中均已优化。状态更新时，我使用vaddq_u32并行累加轮函数结果，并通过vbslq_u32指令根据mask选择性更新有效消息的状态。字节序调整是为了确保字节顺序正确，我使用了vreinterpretq_u8_u32、vrev32q_u8和vreinterpretq_u32_u8等NEON指令进行转换。最后，我将SIMD向量寄存器中的哈希结果重组并存储到目标数组中，完成了MD5哈希算法的SIMD实现。

我对main.cpp也进行了修改，定义了BATCH_SIZE为4，并计算了能完整进行SIMD运算的字符串最大数量，对这部分字符串调用了MD5Hash_SIMD，而剩余部分则使用原始的串行操作。为了验证代码的正确性，我使用了correctness.cpp，初始化了四个不同长度的字符串，分别调用了四次串行MD5Hash和一次MD5Hash_SIMD函数，通过对比输出结果，确认了SIMD优化后的代码能够得到相同且正确的结果。

1.3.2 进阶要求

在进阶要求部分，我致力于实现相对串行算法的加速。我发现最初的SIMD算法运行时间反而更长，这主要是因为SIMD算法存在内存开销大、预处理和数组操作多等额外开销。为了解决这个问题，我进行了多项优化：首先，我优化了交错存储模块，跳过了中间的messages数组，直接从paddedMessages生成interleaved数组，从而减少了不必要的内存操作。其次，我多处使用了循环展开技术，减少了循环次数以加速程序执行。此外，我还减少了非必要数组的定义，例如取消了messages和n_blocks数组，通过更直接的方式实现了其功能。经过这些优化，特别是在使用O1、O2编译器优化后，我的SIMD算法运行速度显著超过了原串行算法，优化后的SIMD算法在使用O2优化时加速比达到1.66，这证明了编译优化能有效降低SIMD操作的额外开销，使其并行计算优势得以充分发挥。

我进一步探索了控制“单指令多数据”中数据的总数，尝试将Batch_Size从4改为2和8，并观察了其对加速效果的影响。对于Batch_Size=2，我主要修改了主函数中BATCH_SIZE的值和

MD5Hash_SIMD中数组的大小及循环参数。而对于Batch_Size=8，我使用了两组uint32x4_t状态向量来处理输入0-3和4-7，并且扩展了mask和x数组，增加了循环次数为2的循环来处理这两组数据，以保持与NEON指令集数据处理规模的一致性。验证结果表明，在未优化时，Batch_Size=2的执行时间最长，而Batch_Size=4和Batch_Size=8的执行时间相对较短，但仍高于串行算法。然而，在使用O1、O2优化后，所有Batch_Size的执行时间均大幅下降，特别是Batch_Size=8在O2优化下仅为1.78751秒。这让我认识到，随着Batch_Size增大，SIMD算法的并行处理能力得以充分发挥，有效抵消了额外开销，并且并行化程度越高，执行效率也会提升，但同时也会增加内存开销。

我还尝试不使用NEON指令集实现MD5算法。在了解SIMD算法实现方法后，我没有直接使用NEON向量类型（如uint32x4_t），而是尝试自定义MD5Vector类来封装四个标量值，模拟向量化操作。我还将NEON指令集中的一些指令手动编写成函数进行处理，最终成功实现了不依赖特定指令集的SIMD算法。

1.3.3 实验总结

总的来说，通过SIMD并行化编程实验，我对SIMD编程实践有了深入理解，掌握了NEON指令集的向量化操作，以及通过数据交错存储和掩码控制实现并行计算的完整流程，并对不使用NEON指令集进行SIMD优化有了初步了解。在实现SIMD算法并解决其加速问题过程中，我对性能优化方法，例如交错存储模块优化、循环展开和减少非必要数组定义，有了更深刻的认识，也提升了从现有算法中找出优化点并进行优化的能力。最后，在控制“单指令多数据”总数方面，我体会到了批量规模（batch_size）对吞吐量和内存开销的影响，这让我认识到在实际应用中需要综合考虑运行速度与内存开销等多种因素进行算法选择，从而加深了对并行算法分析和权衡的理解。

1.4 口令猜测过程的简述

在之后的几次实验中，主要的任务都是对口令猜测过程进行并行优化，那么我们有必要重温以下口令猜测任务的流程，来更方便我们理解不同方法并行化的实现思路。

首先，我们先来简单了解一下口令猜测算法，口令猜测算法是一种通过分析用户口令的常见规律和结构，通过对于猜测口令的降序排序，从高概率到低概率依次进行口令猜测，从而猜测出用户的口令。我们既然是猜测口令，肯定不能去盲目猜测，而是利用用户不会使用毫无规律的字符串作为密码，而是更倾向于使用有语义、可预测的密码模式（如”123456”、姓名缩写+数字等）的情况去进行猜测。

下面我们来分析一下口令猜测算法的执行过程，首先我们需要将口令拆解为字母（Letters）、数字（Digits）、符号（Symbols）三类字段，统计训练集中各字段的长度、具体值及其组合关系（preterminal），这个preterminal简言之就是类似于L6 D3 S1等这类模式的口令，我们需要计算不同preterminal 的出现概率，构建概率模型。之后，就是我们需要使用算法实现的部分了，也是在本是实验中需要优化的guessing.cpp中的内容。我们通过优先队列初始化最高概率的字段组合，每次取出队列顶端的，也就是概率最高的preterminal，按”pivot”标记的位置替换后续字段的下一个概率次高值（将字母字段从”thomas”替换为”nankai”），生成新口令并重新计算概率插入队列，从而按概率降序无重复地遍历所有可能的组合，最终输出猜测口令列表。

那么让我们对应到guessing.cpp的代码中：

- `CalProb()`函数： 用于计算PT(preterminal)的概率
- `init()`函数：用于初始化优先队列，按概率从高到低存储不同的猜测PT

- `PopNext()`函数： 调用Generate()生成当前PT对应的所有密码组合，再调用NewPTs()生成新的PT，并将其放到优先队列的对应位置
- `Generate()`函数：根据PT生成具体密码猜测，，支持单Segment和多Segment模式
- `NewPTs()`函数： 生成当前PT的衍生PT

具体函数指令流程的示意图如下：

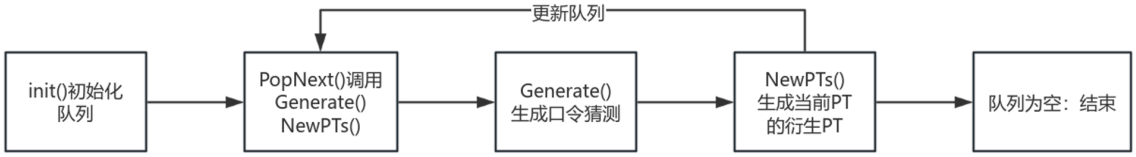

图 1: guessing.cpp执行流程图

1.5 多线程编程

本次实验的目标主要聚焦于口令猜测算法的优化，重点优化部分是Generate函数中的两个循环。我使用 `pthread` 和 `OpenMP` 两种库实现了多线程，并对不同优化方法进行了性能分析和总结。

1.5.1 方法1：动态多线程

我初步尝试使用动态多线程来并行化 `Generate` 函数。为了方便任务分配和数据处理，我定义了一个 `ThreadData` 结构体来封装每个线程所需的参数，包括共享的猜测字符串向量 `guesses`、前缀字符串 `base_guess`、最后一个 `segment` 的统计数据指针 `a`、任务的起始和结束索引 `start_idx` 和 `end_idx`，以及用于线程安全的互斥锁 `mutex` 和猜测总数计数器 `total_guesses`。在线程创建和任务分配时，我计算了每个线程应处理的索引范围，并为每个线程创建了独立的 `ThreadData` 实例。由于 `guesses` 向量和 `total_guesses` 计数器是共享资源，我使用 `pthread_mutex_lock` 和 `pthread_mutex_unlock` 对其更新操作进行保护，确保线程安全。然而，经过测试，我发现这种动态多线程的实现不仅没有加速，反而导致 Guess Time 翻倍。我分析认为，这是因为频繁地创建和销毁线程以及加锁解锁操作带来了过高的额外开销。

1.5.2 方法2：静态线程池

为了解决方法1中线程频繁创建销毁和资源浪费的问题，我引入了静态线程池。我设计了一个 `ThreadPool` 结构体，包含 `pthread_t` 线程句柄数组、任务队列 `tasks`（存储 `ThreadData` 任务）、互斥锁 `task_mutex`、条件变量 `task_cond` 和 `completion_cond`、停止标志 `stop`、线程数量 `num_threads` 以及原子变量 `active_tasks`。

- `thread_pool_init` 函数负责初始化线程池，它会一次性创建指定数量的线程并使其保持活跃，从而消除了重复创建线程的开销。每个线程进入循环，从任务队列中获取任务，若无任务则休眠等待，任务完成后更新 `active_tasks` 并通知主线程。
- `thread_pool_destroy` 函数用于安全销毁线程池，释放所有资源，避免内存泄漏。
- `thread_pool_enqueue` 函数负责将新任务添加到任务队列中，并唤醒等待的线程。

- `thread_pool_wait_all` 函数让主线程等待线程池中所有任务完成，通过检查 `active_tasks` 和任务队列状态，避免了忙等待。

通过这些优化，方法2的运行时间显著降低，已经与原串行算法的运行时间相近，但在某些情况下仍略慢。

1.5.3 方法3：优化后的线程池

尽管方法2有所改进，但我继续分析其性能瓶颈，发现仍存在以下问题：频繁的锁操作、对所有任务（包括小规模任务）都进行并行化以及主线程空闲。因此，我提出了进一步的优化：

- **优化掉锁操作**：为了避免每次生成猜测字符串时对共享资源进行频繁的加锁解锁操作，我通过预先分配 `guesses` 向量的空间并直接索引写入，同时由主线程统一更新 `total_guesses`，从而完全消除了锁竞争。
- **任务分配优化**：我意识到对于小规模任务，多线程的并行化开销可能超过串行算法的运行时间。因此，我引入了任务规模判断，对于小任务使用串行方式处理，只有大任务才使用多线程并行实现。
- **主线程参与运算**：在之前的实现中，主线程在分配任务后会等待子线程完成，处于空闲状态。我修改了代码，让主线程也参与到计算之中，充分利用了CPU资源。

这些优化使得最终的 `pthread` 多线程实现取得了更好的性能。

1.5.4 OpenMP 实现多线程化

除了 `pthread`，我还尝试了使用 `OpenMP` 实现多线程化。 `OpenMP` 是一个高级的并行编程API，我参考 `pthread` 的实现方法，利用 `OpenMP` 指令实现了相同功能的并行化。相较于 `pthread`， `OpenMP` 在编程上更为简洁，通过编译器指令即可实现并行区域和任务划分，大大简化了多线程代码的编写。

1.5.5 对于算法的继续优化

首先，我研究了在“问题规模与线程数量的思考”，并发现了我需要对不同线程使用不同任务阈值判断是否使用多进程优化，并基于研究成果，针对算法本身进行了进一步的思考和优化，最终我深入探讨了如何根据问题规模和硬件特性选择最优的线程数量，以达到最佳的并行性能。

1.5.6 实验总结与思考

通过本次实验，我对SIMD编程实践有了深入理解，掌握了NEON和SSE指令集的向量化操作，以及通过数据交错存储和掩码控制实现并行计算的完整流程，并对不使用NEON指令集进行SIMD优化有了初步了解。在实现SIMD算法并解决其加速问题过程中，我对性能优化方法，例如交错存储模块优化、循环展开和减少非必要数组定义，有了更深刻的认识，也提升了从现有算法中找出优化点并进行优化的能力。最后，在控制“单指令多数据”总数方面，我体会到了批量规模 (`batch_size`) 对吞吐量和内存开销的影响，这让我认识到在实际应用中需要综合考虑运行速度与内存开销等多种因素进行算法选择，从而加深了对并行算法分析和权衡的理解。

1.6 MPI多进程编程

在 MPI（Message Passing Interface）多进程编程实验中，主要的优化对象依然是对口令猜测算法中的Generate函数进行多进程优化，虽然 MPI 算法的实现思路与多进程实验的思路大致相同，但是具体的实现方法依然有区别。

1.6.1 版本1：基本 MPI 实现

在多进程实现部分，我深入探索了两种版本。首先我们来回顾以下版本1的 MPI 多进程编程的实现方法。

在数据分发和初始化阶段，我设计让根进程（rank 0）负责训练数据的读取 (train_P_T 和 train_P_C)，完成训练模型，然后通过 MPI_Bcast 将训练模型广播给所有工作进程。类似地，测试数据（即需要猜测的口令范围）也被根进程广播给所有工作进程。

接着，我详细实现了并行猜测循环。主循环开始时，根进程会生成 Preterminal 数据 (PTs) 并使用 MPI_Bcast 广播给所有工作进程。每个工作进程接收到分配的 guesses 范围后，会生成具体口令，并进行本地的 MD5 哈希验证，记录所有正确的匹配。在结果归约和终止条件检查阶段，每个工作进程将其本地找到的匹配结果和剩余的 total_guesses 发送回根进程 (MPI_Reduce)。根进程聚合这些结果，并根据是否达到预设的匹配数量或猜测上限来决定是否终止循环。一旦进程终止，将进行最终的时间统计并输出结果。对于 generate 函数的多进程实现，我首先在函数定义中加入了 MPI 的初始化 (MPI_Init, MPI_Comm_rank, MPI_Comm_size)。然后，我针对只包含一个段的 PTs，设计了它们在进程间的分布和处理方式。对于包含多个段的 PTs，我则采用了更复杂的策略，涉及在进程内重复计算或更精细的负载均衡。

1.6.2 版本2：优化后的 MPI 实现

在版本1的基础上，我发现了一些性能瓶颈，主要来源于不必要的通信开销和潜在的冗余处理，这促使我进行进一步优化。关键优化点包括：

- **取消训练数据广播：** 在版本2中，我修改了程序，使得每个进程在本地读取训练数据，从而消除了启动阶段的 `MPI_Bcast` 操作，减少了初始化时的通信负担。
- **并行化口令段猜测：** 我将 `Generate()` 函数拆分为 `Generate_main`（供根进程使用）和 `Generate_worker`（供工作进程使用）。这种分离使得口令的生成任务能够更细粒度地分布到各个进程中，实现了更彻底的并行猜测。
- **测试数据预分发：** 我不再广播测试数据，而是根据进程的 rank，预先将 `test_PT_C` 和 `test_PT_T` 等测试数据分配到每个进程，进一步减少了运行时的通信量。
- **猜测生成与哈希检查（工作进程逻辑）：** 工作进程现在负责生成其分配到的 `PT` 范围内的口令，计算其 MD5 哈希值，并与目标哈希进行比对。
- **结果报告（工作进程逻辑）：** 工作进程将其找到的匹配口令及其数量发送回根进程。
- **根进程聚合：** 根进程负责收集所有工作进程的结果，打印已找到的口令，更新总计数，并判断是否满足终止条件。

经过性能对比，版本2相对于版本1和串行算法都展现出显著的性能提升。在口令猜测上限为10,000,000时，版本2的加速比约为11，这表明我的多进程优化取得了较好的效果。但是，我也观察到随着进程数的增加，Guessing Time 的减少速度逐渐变慢，甚至在某些点呈现上升趋势。我推测这可能是因为尽管增加了并行处理单元，但进程间的通信时间也随之增加，最终抵消了部分并行带来的收益。

1.6.3 实验总结

经过 MPI 多进程编程实验，我对多进程并行化编程有了深刻的理解。我体会到其在分布式计算中的强大优势，能够有效利用多台机器或多核处理器，避免了共享内存的复杂性。然而，我也认识到多进程编程的挑战，包括通信开销、负载均衡的复杂性以及同步机制的设计。我特别思考了问题规模与进程数量之间的关系，意识到并非进程数越多越好，而是需要根据具体的任务特性和硬件资源来确定最优的并行粒度。总而言之，这次实验让我对多进程并行有了更全面和实践性的认识，也提升了我解决分布式计算问题的能力。

1.7 GPU 编程

GPU 编程实验使用 NVidia GPU 的任务分配算法，实现 CUDA 程序，主要进行优化的对象还是Generate函数。

1.7.1 版本1：基本 CUDA 实现

我首先简述了 GPU 加速的概念，它通过利用 GPU 的大规模并行处理能力来显著提升计算密集型任务的性能，这对于口令哈希计算这类高度并行化的任务尤其适用。

在**数据处理（打包、传输与回传）**方面，我进行了以下操作：

- **数据打包（CPU 端操作）：** 为了优化数据传输效率，我将输入数据（如口令段和 preterminal 概率）在 CPU 端打包成连续的内存块。
- **主机到设备内存传输（CPU 端操作）：** 我使用 `cudaMemcpy` 函数将打包好的数据从 CPU（主机）内存传输到 GPU（设备）内存。
- **结果回传（GPU 端到 CPU 端）：** 计算完成后，我再次使用 `cudaMemcpy` 将结果（如找到的口令及其数量）从 GPU 内存传输回 CPU 内存。

在 **GPU 内存管理（静态复用）**方面，我采取了以下策略：

- **静态变量声明与流初始化：** 我利用静态设备内存（通过 `__device__` 关键字）来高效管理内存，并创建 `CUDA` 流进行异步操作。
- **容量检查与条件式内存分配：** 在 GPU 上分配内存时，我动态检查现有已分配内存是否足够，如果不足，则进行重新分配，以适应不同规模的任务。

我设计了 **CUDA 核函数（`__global__` 函数）**，它是实际在 GPU 上并行执行的代码：

- **核函数启动与线程配置：** 我通过指定网格（block）和线程（thread）的数量来启动核函数，并利用 `blockIdx.x` 和 `threadIdx.x` 将每个线程映射到特定的数据元素（即口令段），从而实现并行处理。
- **GPU 核函数：并行执行与数据定位：** 每个 GPU 线程负责处理一个特定的口令段，执行 MD5 哈希计算，并与目标哈希进行比对。线程能够高效地从全局内存块中访问各自所需的数据。

我建立了**同步机制（流与同步点）**来协调 CPU 和 GPU 的操作：

- **CUDA 流的创建与使用：** 我利用 `CUDA` 流来管理多个异步操作，使其能够并发执行，有效避免了 CPU 等待 GPU 完成任务的停滞。

- **异步操作与 CPU/GPU 并发：** 通过流，我实现了数据传输和核函数执行的重叠，使得 CPU 能够在 GPU 忙碌时执行其他任务，从而提高了整体系统的并发性。

我还实现了 **CPU 回退路径**，以应对 GPU 执行失败或不可用的情况，确保程序的鲁棒性。版本1的测试结果显示，与串行算法相比，GPU 加速已能带来显著的性能提升。

1.7.2 版本2：PopNext() 并行化

尽管版本1有所改进，但我分析发现 `PopNext()` 函数（负责提取口令猜测任务）仍是串行执行的瓶颈。为了进一步优化性能，我着手对 `PopNext()` **进行并行化**，使其能够一次性在 GPU 上处理一批任务 (`Batch`)。我引入了**任务量判断机制**，即设定一个任务规模阈值。当任务量小于此阈值时，选择在 CPU 上串行处理，以避免 GPU 调用的额外开销；当任务量较大时，则将任务卸载到 GPU 上进行并行计算。结果传输与 CPU 回退路径的实现与版本1类似。版本2的测试结果显示，经过对 `PopNext()` 函数的并行化处理并结合任务量判断，加速比变得非常可观，相比串行算法和多进程算法都有了显著的性能提升。

1.7.3 问题规模阈值的思考

在问题规模阈值的思考中，我进行了详细的测试来寻找最优的任务量阈值和 `BatchSize`。首先，在不使用编译优化的情况下，我测试了 `BatchSize` 分别为10和100时，不同任务量阈值对 Guess Time 的影响，并绘制折线图，最终确定最佳任务量阈值在10000左右。接着，我将任务量阈值设定为10000，进一步测试了不同的 `BatchSize`，从而找到了最优的 `BatchSize`。

1.7.4 实验总结

通过本次 GPU 加速实验，我对 GPU 编程有了更深入的理解，掌握了 `CUDA` 内存管理、核函数设计和异步流操作等关键技术。我认识到 GPU 加速对于高度并行化任务的巨大潜力，同时也学会了识别和优化并行瓶颈，如数据传输开销和串行部分的限制。

2 新增内容

以下全部是本次期末报告中的新增内容。

3 在x86环境中使用SSE指令进行MD5Hash的SIMD向量化编程

由于在 GPU 的服务器中无法使用 NEON 指令集实现 SIMD 的编程，所以我将原有的代码进行修改，使 SIMD 的编程适配与 x86 架构的处理器，但保持其中的逻辑不发生变化，具体实现方法不做展开（因为代码逻辑大致相同），在此简要列举两处的代码的修改示例。

3.1 数据加载和交错方式

交错存储是为了在 SIMD 向量中并行处理来自不同输入的相同位置的消息块。由于 NEON 提供了丰富的指令来处理不同数据粒度的数据加载和重排，但 SSE/AVX 在加载未对齐或需要特定字节重排的数据时，不如 NEON 灵活，所以我们需要对数据加载与交互方式进行调整。

首先我们来看原有的 NEON 指令的实现方式：

```
// 使用 SIMD 交错消息块，展开 k 循环以使用常量通道索引
uint32_t *interleaved = interleaved_pool4;
for (int i = 0; i < max_blocks; ++i)
{
        for (int j = 0; j < 16; ++j) // j 是 MD5 消息块中的 word 索引 (0-15)
        {
                // k = 0 处理第一个输入的消息块()
                if (i < messageLength[0] / 64)
                {
                        // 从原始字节流加载 16 字节
                        uint8x16_t bytes = vld1q_u8(&paddedMessages[0][i * 64 + j * 4]);
                        // 位重解释为 uint32x4_t 向量（包含个 4 位字）32
                        uint32x4_t word = vreinterpretq_u32_u8(bytes);
                        // 将 word 向量的第一个 32 位通道索引 ( 0) 存储到交错数组的相应位置
                        vst1q_lane_u32(&interleaved[16 * batch_size * i + 4 * j + 0], word, 0);
                }
                else
                {
                        interleaved[16 * batch_size * i + 4 * j + 0] = 0;
                }
                // ... 对 k = 1, 2, 3 重复类似逻辑 ...
                // 例如 k = 1:
                if (i < messageLength[1] / 64)
                {
                        uint8x16_t bytes = vld1q_u8(&paddedMessages[1][i * 64 + j * 4]);
                        uint32x4_t word = vreinterpretq_u32_u8(bytes);
                        vst1q_lane_u32(&interleaved[16 * batch_size * i + 4 * j + 1], word, 0);
                }
                else
                {
                        interleaved[16 * batch_size * i + 4 * j + 1] = 0;
                }
                // ... 此处省略( k=2, k=3 的类似代码，但原理相同)
        }
```

```
}
```

上述代码首先将 16 字节的数据加载为 uint8x16_t，然后直接位重解释为 uint32x4_t，最后通过 vst1q_lane_u32 精确地将 uint32x4_t 向量中的第一个 32 位字存储到交错数组的正确位置。

下面我们来看修改之后的使用 SSE 指令集的实现方法：

```
// 交错装载每个的每个blockword
uint32_t *interleaved = interleaved_pool4;
for (int i = 0; i < max_blocks; ++i)
{
        for (int j = 0; j < 16; ++j) // j 是 MD5 消息块中的 word 索引 (0-15)
        {
                for (int k = 0; k < batch_size; ++k) // k 是批处理中的输入索引
                {
                        if (i < messageLength[k] / 64)
                        {
                                int base = i * 64 + j * 4; // 计算当前消息块和 word 的起始字节偏移
                                // 手动将 4 个字节组合成一个 32 位整数 (小端序)
                                interleaved[16 * batch_size * i + 4 * j + k] =
                                (paddedMessages[k][base]) |
                                (paddedMessages[k][base + 1] << 8) |
                                (paddedMessages[k][base + 2] << 16) |
                                (paddedMessages[k][base + 3] << 24);
                        }
                        else
                        {
                                interleaved[16 * batch_size * i + 4 * j + k] = 0;
                        }
                }
        }
}

// 在处理块时加载交错数据到 SIMD 寄存器
for (int i = 0; i < max_blocks; ++i)
{
        // ...
        __m128i x[16];
        uint32_t *block_start = &interleaved[16 * batch_size * i];
```

```
        for (int j = 0; j < 16; ++j)
        {
                // 从预处理的 32 位整数数组中组装 __m128i 向量
                x[j] = _mm_setr_epi32( // _mm_setr_epi32 按参数顺序反向填充向量
                block_start[4 * j + 0],
                block_start[4 * j + 1],
                block_start[4 * j + 2],
                block_start[4 * j + 3]
                );
        }
        // ...
}
```

在这段 SSE 代码中，首先通过 C 语言的位移和按位或操作，将 4 个 Byte（uint8_t）手动组合成一个 uint32_t。这些 32 位整数然后直接存储在 interleaved_pool4 数组中。在后续的消息块处理循环中，再使用 _mm_setr_epi32 从这个预处理好的 interleaved 数组中加载 4 个 32 位整数，将它们组装成一个 __m128i 向量。这样，我们就复现了使用 NEON 的交错存储。

3.2 字节序转换

MD5 算法通常要求处理小端序数据，但在某些场景下（如果原始输入是大端序或需要在不同字节序系统之间交换数据），可能需要进行字节序转换。在这一处理过程中，涉及到 32 位字节反转的操作，其中 NEON 提供了 vrev32q_u8 这样的专用指令，可以直接对 128 位向量中的 8 位数据执行 32 位反转，但是，SSE 中没有这一功能，因此，在使用 SSE 进行编写时需要通过其他方式实现这一功能。

NEON 字节序转换代码如下：

```
// 字节序转换
state_a = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(state_a)));
state_b = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(state_b)));
state_c = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(state_c)));
state_d = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(state_d)));
```

上述代码直接使用 vrev32q_u8 将每四个 8 位字节视为一个 32 位字，并将其字节序反转（例如，[b0, b1, b2, b3] 变为 [b3, b2, b1, b0]）。

但是在 SSE 中，实现方式发生了变化：

```
// 字节序转换（小端转大端）
auto bswap32 = [](__m128i x) {
        // 提取最低 8 位，左移 24 位 (b0 -> b3)
        __m128i t1 = _mm_slli_epi32(_mm_and_si128(x,
            _mm_set1_epi32(0x000000ff)), 24);
```

```
        // 提取次低 8 位，左移 8 位 (b1 -> b2)
        __m128i t2 = _mm_slli_epi32(_mm_and_si128(x,
            _mm_set1_epi32(0x0000ff00)), 8);
        // 提取次高 8 位，右移 8 位 (b2 -> b1)
        __m128i t3 = _mm_srli_epi32(_mm_and_si128(x,
            _mm_set1_epi32(0x00ff0000)), 8);
        // 提取最高 8 位，右移 24 位 (b3 -> b0)
        __m128i t4 = _mm_srli_epi32(_mm_and_si128(x,
            _mm_set1_epi32(0xff000000)), 24);
        // 将所有部分按位或组合起来
        return _mm_or_si128(_mm_or_si128(t1, t2), _mm_or_si128(t3,
            t4));
};

state_a = bswap32(state_a);
state_b = bswap32(state_b);
state_c = bswap32(state_c);
state_d = bswap32(state_d);
```

SSE 代码需要通过一系列的位移和按位逻辑操作来模拟这个功能。我在代码中定义了一个 lambda 函数 bswap32 来封装这个逻辑，它将一个 32 位字拆分成 4 个字节，然后重新组合，但字节顺序是反过来的，由此实现了 NEON 中的 vrev32q_u8 字节序反转功能。

3.3 结果验证与分析

在完成上述操作后，我们验证一下使用 SSE 进行 SIMD 向量化编程后代码的正确性。

同样使用correctness.cpp，初始化4个不同长度的字符串，分别调用4次MD5Hash、调用一次 MD5Hash SIMD函数，将MD5Hash SIMD函数输出的结果与MD5Hash函数的输出结果进行对比。

将上述验证方法重复多次，发现输出的结果均相同，代表我们使用 SSE 进行 SIMD 向量化编程后的代码正确!

对应加速效果以及加速比如下所示：

表 1: 使用 SSE 进行 SIMD 向量化 Hash Time 加速比

	不使用编译优化加速比	O2编译优化加速比
使用 NEON 进行 SIMD 优化	0.57	1.66
使用 SSE 进行 SIMD 优化	0.80	1.68

通过上述测试结果我们可以发现，使用 NEON 进行优化的加速比与使用 SSE 进行优化的加速比几乎一致，都是在不使用编译优化时为负优化，在使用编译优化时候的加速比约为1.67。

我们来深入探究一下测试结果，为什么不使用编译优化时反而是负优化呢？这主要是因为 SIMD 算法在执行过程中存在额外开销，且这些开销在未充分优化时无法被有效抵消。SIMD 操作需要进行数据对齐、特殊指令加载与存储等额外操作，这些操作本身具有一定开销，因此导致其并行计算带来的效率提升不足以弥补这些额外开销。串行算法虽未利用 SIMD 并行性，但

逻辑简单直接，无需处理这些复杂额外操作，避免了相关开销，因此即便未使用 O1、O2 优化，未优化的 SIMD 算法运行速度仍比原算法慢。

4 对于 train.cpp 的优化

在之前的实验中，我们始终没有对 train.cpp进行优化，下面，我将对 train.cpp 使用多线程的方法对 train.cpp 进行优化。

4.1 train.cpp 的训练过程

在进行优化之前，我们先来分析一下整个 train.cpp 的实现过程，有助于我们更好的理解如何优化代码。

train.cpp 通过对大量口令的统计分析，构建一个概率上下文无关文法（PCFG）模型。其核心实现理念在于“解剖”口令，将其分解为结构模式和内容组件，并量化它们的出现频率，最终形成对口令特征的概率性描述。

整个训练过程首先从指定文件中读取训练口令。程序逐一处理每个口令，将其中的连续字符序列识别为不同类型的“片段”（segment），例如字母片段、数字片段或符号片段。当字符类型发生变化时，当前的片段即被确定，其类型（如字母）、长度（如3位）和具体内容（如”abc”）都会被记录。这些按顺序排列的片段共同构成了一个“预终结符”（Pre-Terminal，简称PT），它抽象地表示了口令的结构模式（如“L3D2S1”代表一个口令由一个3位字母串、一个2位数字串和一个1位符号串组成）。在解析过程中，程序会实时统计每种独有片段类型及其具体内容、以及每种独有PT模式的出现次数。

完成所有口令的解析后，程序会进入一个数据整理阶段。在此阶段，系统会根据前面收集到的统计数据，计算每种PT模式的相对频率，并据此对PT模式进行排序，通常是按出现概率从高到低排列。同时，对于每种类型的片段（如所有长度为3的字母片段），其内部的具体值（如“abc”、“xyz”）也会根据它们的出现频率进行排序。这些排序后的频率数据和模式信息构成了PCFG模型的核心，使得模型能够学习并反映出实际口令数据中最常见的结构和内容，从而为后续的口令生成或猜测任务提供概率依据。

4.2 train.cpp 的优化思路

train.cpp 的优化思路主要集中在通过多线程并行化加速PCFG模型的训练过程。其核心思想是分治合并：将大规模的口令数据集拆分成若干个独立的子集，然后为每个子集启动一个独立的线程来并行地训练一个“局部”的PCFG模型。每个局部模型负责统计其分配到的口令分片中的预终结符（PT）模式和片段内容的频率。由于这些统计计数是可加的，即对各个子集进行统计并将结果累加，与直接对整个数据集进行串行统计得到的结果是等价的，因此这种方法在保证模型正确性的同时实现了并行处理。

具体实现步骤包括：首先将所有训练口令从文件中一次性读取到内存中，并将其均等地分割成多份。随后，启动多个线程，每个线程处理一个数据分片并独立更新其对应的模型统计数据。在所有线程完成各自的训练任务后，通过 model::merge 函数将所有局部模型的统计数据合并到主模型中。最后，在合并完成的总统计数据上进行最终的概率计算和排序，从而得到完整的PCFG模型。这种优化方法有效利用了现代多核处理器的计算能力，显著缩短了处理大规模口令数据集所需的训练时间。

下面我们具体开看对于 train.cpp 的代码实现方法。

1.数据读取与分片

```
// 1. 读取所有口令到内存
vector<string> all_pwds;
string pw;
ifstream train_set(path);
int lines = 0;
cout << "Training..." << endl;
cout << "Training phase 1: reading and parsing passwords..." << endl;
while (train_set >> pw)
{
        lines += 1;
        if (lines % 10000 == 0)
        {
                cout << "Lines processed: " << lines << endl;
                if (lines > 3000000)
                {
                        break;
                }
        }
        all_pwds.push_back(pw);
}
cout << "[Main] Total passwords loaded: " << all_pwds.size() << endl;
// 2. 分成份4
size_t total = all_pwds.size();
size_t part = (total + 3) / 4;
vector<string> pwds1(all_pwds.begin(), all_pwds.begin() + min(part, total));
vector<string> pwds2(all_pwds.begin() + min(part, total), all_pwds.begin() + min(2 * part, total));
vector<string> pwds3(all_pwds.begin() + min(2 * part, total), all_pwds.begin() + min(3 * part, total));
vector<string> pwds4(all_pwds.begin() + min(3 * part, total), all_pwds.end());
```

通过上述代码，我们能够实现 PCFG 模型训练前的预处理阶段：它首先通过 ifstream train_set(path) 打开指定路径的训练文件，然后在一个 while 循环中逐行读取口令并将其存储到 std::vector¡string¿ all_pwds 中，为了防止加载过大的文件，设置了最多处理300万行口令的限制。一旦所有（或达到限制数量的）口令被载入内存，程序便计算总口令数 total，并依据此数量将 all_pwds 大致均等地分割成四份。分割逻辑通过计算 part = (total + 3) / 4 确定每份的大小（加3是为了向上取整，确保所有口令都能被分配），然后利用 std::vector 的构造函数和迭代器范围，创建了四个新的 vector¡string¿ 实例：pwds1、pwds2、pwds3、pwds4，每个实例分别持有原始数据的一个子集。这种数据分片是实现后续多线程并行训练的基础，它确保了每个线程都能在独立的数据

集上进行操作，从而避免了数据竞争和复杂的同步问题，为训练过程的加速奠定了结构上的基础。

2.并行训练

```
// 多线程训练辅助函数
void train_worker(model *mdl, const vector<string> &pwds)
{
        cout << "[Thread␣" << this_thread::get_id() << "]␣Start␣
            training,␣size=" << pwds.size() << endl;
        for (const string &pw : pwds)
        {
                mdl->parse(pw);
        }
        cout << "[Thread␣" << this_thread::get_id() << "]␣Finish␣
            training." << endl;
}

// 3. 启动个线程分别训练个模型，主线程自己处理33pwds1
model m1, m2, m3, m4;
cout << "[Main]␣Launching␣threads..." << endl;
thread t2(train_worker, &m2, cref(pwds2));
thread t3(train_worker, &m3, cref(pwds3));
thread t4(train_worker, &m4, cref(pwds4));
// 主线程自己处理pwds1
train_worker(&m1, pwds1);
t2.join();
t3.join();
t4.join();
```

上述代码是 train.cpp 实现并行训练的核心。我们首先定义一个辅助函数 train_worker，该函数接收一个指向 model 对象的指针和一个口令字符串向量的常量引用。train_worker 的职责是遍历其接收到的口令分片，并对每个口令调用 model::parse() 方法，从而独立地统计该分片内的口令特征频率，更新其专属 model 对象的内部状态。在 model::train 函数中，程序创建了四个独立的 model 对象 (m1、m2、m3、m4)，然后利用 std::thread 启动了三个新的线程 (t2、t3、t4)，分别将 train_worker 函数与 m2/pwds2、m3/pwds3、m4/pwds4 进行绑定，使其并行执行。同时，主线程也积极参与到计算中，直接调用 train_worker 处理 m1 和 pwds1。这种安排确保了数据被有效地分散到多个处理单元上。最后，通过 t2.join()、t3.join() 和 t4.join()，主线程会等待所有子线程完成各自的训练任务，保证在进行后续的数据合并之前所有并行计算都已结束。通过多线程的并行训练，我们将耗时的口令解析和频率统计工作并行化，从而大幅提升了训练效率。

3.线程汇合与模型合并

我们定义 merge 函数来实现线程汇合与模型合并操作，可以分为 **合并预终结符、合并字母片段、合并数字片段、合并符号片段** 四个主要功能块，每个块负责合并一种特定类型的数据，merge函数每块的操作过程大致如下：

1. **遍历**: 每个过程都遍历来自“其他”模型 (other) 的对应类型的数据集合。
2. **查找**: 对于遍历到的每个条目（无论是预终结符模式还是某种片段类型，以及片段内部的具体值），都会在当前模型 (this) 中查找是否已经存在相同的条目。
3. **累加/添加**:
   - 如果当前模型中**存在**该条目，则将“其他”模型中对应的频率累加到当前模型的该条目频率上。
   - 如果当前模型中**不存在**该条目，则将“其他”模型中的新条目（包括其初始频率）完整地添加到当前模型中。

我们以**合并预终结符、合并字母片段**为例来说明一下如何对进行线程汇合与模型合并：

（1）合并预终结符

```
// 合并preterminals
for (int i = 0; i < other.preterminals.size(); ++i)
{
        int idx = FindPT(other.preterminals[i]);
        if (idx == -1)
        {
                preterminals.push_back(other.preterminals[i]);
                preterm_freq[preterminals.size() - 1] = other.
                    preterm_freq.at(i);
        }
        else
        {
                preterm_freq[idx] += other.preterm_freq.at(i);
        }
}
total_preterm += other.total_preterm;
```

这段代码专门负责将一个“其他”模型 (other) 中统计到的预终结符（PT，即口令结构模式）数据合并到当前模型 (this) 中。它通过一个循环遍历 other 模型的所有 preterminals 向量。在循环内部，FindPT(other.preterminals[i]) 函数用于查找当前模型中是否已存在与 other 模型中当前PT模式相同的条目。如果 idx 返回 -1，表示当前模型中没有这个PT模式，那么该新模式会被添加到当前模型的 preterminals 向量的末尾，并且其对应的频率（other.preterm_freq.at(i)）会被记录在 preterm_freq 映射中。反之，如果 idx 不为 -1，说明当前模型已经包含了这个PT模式，程序便直接将 other 模型中该模式的频率累加到当前模型对应PT模式的频率 (preterm_freq[idx]) 上。最后，无论是否添加了新的PT模式，other 模型中的总预终结符计数 (other.total_preterm) 都会累加到当前模型 (this) 的 total_preterm 中。这个过程确保了来自并行训练的各个局部模型中的所有口令结构模式及其出现频率，都能被准确、无遗漏地整合到最终的总模型中。

（2）合并字母片段

```
// 合并letters
for (int i = 0; i < other.letters.size(); ++i)
{
        int idx = FindLetter(other.letters[i]);
        if (idx == -1)
        {
                letters.push_back(other.letters[i]);
                letters_freq[letters.size() - 1] = other.
                    letters_freq.at(i);
        }
        else
        {
                letters_freq[idx] += other.letters_freq.at(i);
                // 合并segment values
                for (const auto &kv : other.letters[i].values)
                {
                        string val = kv.first;
                        int val_idx = kv.second;
                        if (letters[idx].values.find(val) ==
                            letters[idx].values.end())
                        {
                                int new_idx = letters[idx].values.
                                    size();
                                letters[idx].values[val] = new_idx
                                    ;
                                letters[idx].freqs[new_idx] =
                                    other.letters[i].freqs.at(
                                    val_idx);
                        }
                        else
                        {
                                int exist_idx = letters[idx].
                                    values[val];
                                letters[idx].freqs[exist_idx] +=
                                    other.letters[i].freqs.at(
                                    val_idx);
                        }
                }
        }
}
```

上述代码负责将一个“其他”模型 (other) 中统计到的字母片段（如长度为3的字母序列、或纯小写字母序列等）数据合并到当前模型 (this) 中。它通过循环遍历 other 模型的所有 letters

向量。对于 other.letters 中的每一个字母片段类型，FindLetter(other.letters[i]) 函数会尝试在当前模型中查找是否存在相同类型的字母片段。如果 idx 返回 -1，表示当前模型没有这个特定类型的字母片段，那么这个新的字母片段类型会被整体添加到当前模型的 letters 向量中，并将其对应的总频率 other.letters_freq.at(i) 记录下来，同时，该片段类型内部的所有具体值（例如，"abc", "xyz"）及其频率也会被一并复制过来。如果 idx 不为 -1，说明当前模型已经包含了这个字母片段类型，程序首先将 other 模型中该类型片段的总频率累加到当前模型对应的总频率上。接着，它会进一步遍历 other 模型中该字母片段类型下的所有具体值（kv.first）及其频率（kv.second），将这些具体值的频率累加到当前模型对应具体值的频率上，或者如果当前模型没有这个具体值，则将其作为一个新的具体值添加到当前片段类型中。这个细致的合并过程确保了来自不同并行训练任务的所有字母片段的结构、具体值以及它们的出现频率都能够正确地汇总到最终的模型中。

（3）合并数字片段

与**合并字母片段**相似，只需要将 letters、letters_freq 向量替换为 digits、digits_freq 即可，不做展开。

（4）合并符号片段

与**合并字母片段**相似，只需要将 letters、letters_freq 向量替换为 symbols、symbols_freq 即可不做展开。

在解释完 merge 函数的实现方法之后，我们需要在训练中调用 merge 函数进行线程汇合与模型合并，具体代码如下：

```
// 4. 合并个模型4
cout << "[Main] Merging model 1..." << endl;
this->merge(m1);
cout << "[Main] Merging model 2..." << endl;
this->merge(m2);
cout << "[Main] Merging model 3..." << endl;
this->merge(m3);
cout << "[Main] Merging model 4..." << endl;
this->merge(m4);
cout << "[Main] Merge finished." << endl;
```

在此阶段，通过顺序调用 this-¿merge() 方法，将各个局部训练出的模型 m1、m2、m3 和 m4 的统计数据（包括各种片段和预终结符的频率）累加到当前（主）模型中。由于PCFG模型中的频率统计具有可加性，这种简单的累加操作即可得到与串行处理整个数据集完全一致的最终统计结果。通过这种先并行分散计算再集中合并统计的设计，程序有效地利用了多核处理器的性能，显著缩短了整体训练时间。

4.最终排序

```
bool compareByPretermProb(const PT &a, const PT &b)
{
        return a.preterm_prob > b.preterm_prob; // 降序排序
}
```

```
5
6  void model::order()
7  {
8          cout << "Training phase 2: Ordering segment values and PTs
               ..." << endl;
9          for (PT pt : preterminals)
10         {
11                 pt.preterm_prob = float(preterm_freq[FindPT(pt)])
                       / total_preterm;
12                 ordered_pts.emplace_back(pt);
13         }
14         // cout << "total pts" << ordered_pts.size() << endl;
15         std::sort(ordered_pts.begin(), ordered_pts.end(),
               compareByPretermProb);
16         cout << "Ordering letters" << endl;
17         for (int i = 0; i < letters.size(); i += 1)
18         {
19                 letters[i].order();
20         }
21         cout << "Ordering digits" << endl;
22         for (int i = 0; i < digits.size(); i += 1)
23         {
24                 digits[i].order();
25         }
26         cout << "ordering symbols" << endl;
27         for (int i = 0; i < symbols.size(); i += 1)
28         {
29                 symbols[i].order();
30         }
31 }
```

这段代码负责在所有口令数据被解析和合并统计完成后，对PCFG模型的关键组成部分进行概率计算和排序。在 model::order() 函数中，程序首先遍历所有已收集的预终结符（PT）模式，计算每个PT模式在其总出现次数中的概率（preterm_prob = float(preterm_freq[FindPT(pt)]) / total_preterm），并将这些带有概率信息的PT模式收集到 ordered_pts 向量中。随后，使用 std::sort 结合自定义的比较函数 compareByPretermProb（按概率降序排列），对 ordered_pts 进行排序，确保最常见的口令结构模式排在前面。类似地，对于不同类型的片段（字母、数字、符号），代码也迭代遍历它们的统计数据，并调用各自的 order() 方法。例如，segment::order() 函数会计算每个具体片段值（如"abc"在所有字母片段中的出现频率），并对其进行排序。

这一阶段是模型构建的最后一步，它将原始的频率统计数据转化为具有概率意义的有序列表，为后续的口令生成或猜测算法提供了高效且有意义的查找依据。此过程是在所有数据合并完成后进行的，因此本身是串行操作，但它是并行化训练流程中不可或缺的最终整理步骤。

表 2: 进行优化后的 train.cpp 与未优化的基础算法的 Train Time比较

	不使用编译优化/单位(s)	使用O2编译优化/单位(s)
基础算法	96.62	28.52
多线程优化算法	51.86（加速比：1.86）	14.55 （加速比：1.96）

4.3 测试结果

通过上述表格我们可以明显地看出，使用多进程对 train.cpp 进行优化之后，加速比大约为1.9左右，说明我们实现的多线程并行化还是非常有效的。

4.4 进一步优化

当然，我们还可以对上述多线程 train.cpp 进行进一步优化，优化方法为：采用二进制格式存储和读取训练集，减少I/O时间，当然，这个和并行相关性不大，不过确实能够实现加速，我来简要解释一下其实现方式：

首先我们通过 python 代码将训练集转换为二进制格式（每行一个口令，先写长度再写内容），然后在C++中用二进制方式读取，使用二进制读写的方式远快于普通的I/O读写方式。

之后，我们再对训练过程进行修改：

```
void model::train(string path)
{
        vector<string> all_pwds;
        ifstream train_set(path, ios::binary);
        if (!train_set) {
                cerr << "Failed to open " << path << endl;
                return;
        }
        // 预分配容量（假设最大万条）300
        all_pwds.reserve(3000000);

        cout << "Training..." << endl;
        cout << "Training phase 1: reading and parsing passwords
            ..." << endl;
        int lines = 0;
        while (train_set.peek() != EOF)
        {
                uint32_t len = 0;
                train_set.read(reinterpret_cast<char*>(&len),
                    sizeof(len));
                if (!train_set) break;
                string pw(len, '\0');
                train_set.read(&pw[0], len);
                if (!train_set) break;
                all_pwds.push_back(pw);
```

```
                lines++;
                if (lines % 10000 == 0)
                {
                        cout << "Lines processed: " << lines <<
                            endl;
                        if (lines > 3000000)
                        break;
                }
        }
        cout << "[Main] Total passwords loaded: " << all_pwds.size
            () << endl;
        // 后续代码不变.....
}
```

1. **文件打开模式首先**:使用 ifstream train_set(path, ios::binary); 以二进制模式打开文件
2. **数据读取方式**: 使用二进制读取方式：首先读取一个 uint32_t len (train_set.read(reinterpret_cast <char*> (&len), sizeof(len));这表示它期望文件中每个口令前面都有一个4字节的长度指示器。然后，它根据这个读取到的 len 值预先分配字符串的内存 (string pw(len, '\0');)。最后，它精确地读取 len 个字节到字符串中 (train_set.read(&pw[0], len);)。
3. **性能优化**:使用all_pwds.reserve(3000000);预先为 vector 分配了足够的内存空间，可以减少在读取大量口令时因频繁重新分配内存而带来的性能开销。

最终优化结果如下：

表 3: 最终进行优化后的 train.cpp 与未优化的基础算法的 Train Time比较

	不使用编译优化/单位(s)	使用O2编译优化/单位(s)
基础算法	96.62	28.52
最终优化算法	45.21（加速比：2.14）	10.36 （加速比：2.75）

由表中数据可以看出，我们对 Train Time 又进行了优化，加速比超过了2。

5 对于 Guess Time 的最终优化

上面两个优化方案一个是对 Hash Time 的优化，一个是对 Train Time 的优化，接下来，我们来看我们是否能对 Guess Time 进行最终优化!

首先我们来回顾一下，我们之前都对 Guess Time 做了哪些优化操作：多线程优化、多进程优化、GPU 优化、对于 PopNext() 函数进行批处理的并行化。在这次实验中，我想尝试能否将上述优化的方案与经验组合起来，实现最终的优化，其中，比较遗憾的是我们的 GPU 服务器貌似不支持 MPI 的多进程实现，所以我最终放弃了将 GPU 编程也加入最终优化之中，那么，我们需要将在 MPI 编程中（我在 MPI 编程中已经实现 PopNext() 函数的批处理）融合多线程的方法，来实现最终的优化。

5.1 在 MPI 版本1中增加多线程

首先，因为我在 MPI 实验中在 Generate函数执行时做了分块处理，我们只需要对每块尽行并行化即可。代码如下：

5.1.1 任务分配

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

通过 MPI_Comm_rank(MPI_COMM_WORLD, &rank); 获取当前进程的唯一标识 rank。

通过 MPI_Comm_size(MPI_COMM_WORLD, &size); 获取总的进程数量 size。

最核心的并行分配体现在循环 for (int i = rank; i ¡ total_vals; i += size) 中。这意味着每个 MPI 进程 (rank) 会负责生成所有可能值中的一部分，具体是索引为 rank, rank + size, rank + 2*size, ... 的那些值。这种方式确保了所有进程能够均匀地分摊生成任务，避免重复，也降低了跨进程通信的复杂性。

之后，计算当前 PT（预终结符）需要生成的口令数量 problem_size。如果 problem_size 小于一个设定的阈值（这里是 10000），函数会选择串行方式生成口令。这是为了避免并行化的启动开销大于串行执行的收益；如果 problem_size 达到或超过阈值，则切换到OpenMP 多线程并行处理。

如果需要进行多线程并行化，则执行以下代码（以多 segment 为例）：

```

        int total_iterations = problem_size;
        size_t old_size = guesses.size();
        guesses.resize(old_size + total_iterations); // 预分配存储空
            间

        #pragma omp parallel num_threads(num_threads) // 创建固定数
            量的并行线程
        {
                int thread_id = omp_get_thread_num();
                int chunk_size = (total_iterations + num_threads -
                     1) / num_threads;
                int start = thread_id * chunk_size;
                int end = min((thread_id + 1) * chunk_size,
                    total_iterations);

                if (is_single_segment)
                {
                        for (int i = start; i < end; ++i)
                        {
                                guesses[old_size + i] = a->
                                    ordered_values[i];
                        }
                }
```

```
20                  else
21                  {
22                          for (int i = start; i < end; ++i)
23                          {
24                                  guesses[old_size + i] = base_guess
                                        + a->ordered_values[i];
25                          }
26                  }
27          }
28
29          total_guesses += total_iterations; // 并行结束后统一更新总数
```

整体的实现思路与多线程的实现方法类似，首先，利用 #pragma omp parallel num_threads(num_threads) 创建一个固定大小的线程池，随后通过 guesses.resize() 对结果向量进行预先扩容，为每个线程预留了独立的存储区域。在并行区域内部，每个线程根据其唯一 ID 和总线程数手动计算并负责其专属的迭代范围，这种块划分（chunking）策略实现了任务的均匀分发。由于每个线程操作的是向量中互不重叠的区域，从而实现了无锁写入，极大地提升了并行效率并保证了数据完整性。最后，在并行计算结束后，共享变量 total_guesses 进行一次性更新，有效避免了并行执行过程中对共享资源的频繁竞争，从而最大化了性能收益。

5.2 在 MPI 版本2中增加多线程

在 MPI 编程中，在版本2代码中，我将任务分配操作放在了 Generate 函数之外，所以要实现对于 Generate 函数的多线程优化，省去了在 Generate 函数中的任务划分与进程共同，实现多线程优化反而比 MPI 中的第一版代码更简单了，按照对版本1中的多线程优化思路，我们对 Generate 函数进行优化，具体代码内容不赘述。

5.3 MPI 版本1与版本2使用多线程优化后的结果

表 4: 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	8.00	1.26
原版本1	0.27（加速比：29.63）	0.11 （加速比：11.45）
多线程优化后版本1	0.16（加速比：53.33）	0.08 （加速比：15.75）
原版本2	0.53（加速比：15.09）	0.17 （加速比：7.41）
多线程优化后版本2	0.48（加速比：16.67）	0.17 （加速比：7.41）

根据表中（表4）的结果我们可以发现，在对 MPI 中的代码进行多线程优化之后，整体的测试结果均有优化，这说明我们最终优化的结果还是比较成功的，但是在使用编译优化的情况下的加速比反而低于使用编译优化的情况下的加速比，这是为什么呢？

经过对程序进行性能分析之后我发现，当开启编译优化之后，相比于不使用编译优化时 cache 率会下降，这会导致在程序运行时由于对于下一步的指令预测错误，这就使其虽然运行时间更短（可能对于执行的指令进行了其他的优化），但是 cache 率下降导致加速比反而有所下

降。最终导致经过编译优化后的 cache 预测准确率下降，加速比相比于不使用编译优化的情况较低。

6 最终结果展示

我们将上述优化后的代码融合在一起，进行最终的结果展示：

表 5: 最终优化结果（不使用编译优化）

	串行时间/单位(s)	最终并行/单位(s)	加速比
Hash Time	2.01	2.95	0.68
Guess Time	8.00	0.15	53.33
Train Time	88.38	54.24	1.63

表 6: 最终优化结果（使用编译优化）

	串行时间/单位(s)	最终并行/单位(s)	加速比
Hash Time	2.02	1.01	2.00
Guess Time	1.26	0.08	15.75
Train Time	25.75	16.18	1.60

至此，本学期的并行程序设计到这里就结束了！

7 本人对于并行优化的思考

经过这几次对于不同类型的串行算法并行化的实验，结合王刚老师与计组课上张金老师上课讲述的内容，我对并行方法有了一定的了解，在我看来，并行确实可以提高程序的运行速度，但是我们需要平衡好其收益与更多资源的开销，在实验中我们对于工程的整体流程可能了解还不够深入，但是经过老师上课提到的对于企业等需要进行大规模数据处理的情景，可能真正的并行化处理会更加的复杂，我也确实认识到并行优化的重要性及其可能会导致的潜在问题。

7.1 对于并行优化优点的思考

首先我认为单单是对于算法逻辑的优化，如循环展开、分治算法等这些并行化的操作对于额外硬件开销其实是相对较小的，可以说是在代价很小的条件下就能够实现对于之前串行算法的加速，这样的加速是对我们十分有利的。

另外的并行化的方式就是使用额外的运算资源，比如使用多线程、多进程、GPU 等，通过将任务分配给多个运算单元进行运行，从而大大加快程序的运行时间，并且随着硬件计算单元的快速发展，单个运算单元的处理速度会越来越快，程序运行的时间随着运算资源的增多，在理论上其运行时间也会显著降低。

7.2 对于并行优化缺点的思考

并行优化串行算法的缺点我觉得更值得我们关注，首先是实现并行化的难度自然是比实现之前的串行算法要大许多的，通过这几次实验就可以体会到对于一个小型的程序进行并行化优

化没有想象中的那么容易。就像老师上课举的例子，对于冒泡排序这类在每次循环或每一轮任务处理时，其串行方法中的前后相关性过大，从而导致并行化实现的难度增加。尤其是在多线程以及多进程实验中，涉及到多线程的任务分配、加锁解锁、线程池、通信、统一管理等问题，每一个问题都需要在对于代码以及并行思路的深入了解之后才能很好的解决，那么对于大规模的问题，需要进行多个计算机、成百上千个处理器共同运算时，解决它们之间的统一管理与运行监测更是难上加难。另外，就算真的实现了并行化，在实际应用中，我们也需要考虑到实现并行化的代价我们是否能够接受，如果实现并行化需要的代价过大，可能我们需要重新考虑去优化并行方式或重新思考如何有效的实现并行化。

最后是我自己在本学期学习中的一些心得体会：我们在实现并行优化的过程中可能需要放弃已有的结果，去不断思考新的优化方法，不断对并行方法进行优化与迭代，在反复试错与尝试中找出相对最好的结果。同时，对于已经实现的代码的性能分析也格外重要，它能够帮助你更容易地找出现在程序存在的性能瓶颈，从而让你知道后续需要重点解决的问题是什么。

8 期末总结与心得

本学期的并行程序设计课程让我对并行计算领域有了全面而深入的理解。课程内容从国产处理器的调研开始，使我认识到CPU、GPU和APU在功能与应用场景上的差异，并对我国GPU技术的发展现状、挑战与机遇有了清晰的认知，尤其对龙芯3A6000系列等国产处理器的结构与性能有了初步了解。

在体系结构编程实验中，我通过对N*N矩阵与向量内积的Cache优化和N个数求和的多链路算法实践，深入理解了如何利用Cache空间局部性原理和CPU超标量架构来提升程序效率，并学会了使用VTune等工具进行性能分析，这些实践让我认识到算法设计必须充分考虑体系结构特性。

SIMD编程实验让我掌握了NEON指令集的向量化操作，并通过数据交错存储和掩码控制实现了并行计算，同时在优化过程中体会到批量规模对吞吐量和内存开销的影响，这强调了在实际应用中权衡多种因素的重要性。

后续的口令猜测并行优化实验是课程的重点，我先后探索了多线程（pthread和OpenMP）、多进程（MPI）和GPU（CUDA）等多种并行化方法。在多线程编程中，我从动态多线程的低效到静态线程池的改进，再到通过消除锁操作、任务分配优化和主线程参与运算等方式进一步提升性能，深刻理解了多线程编程中线程管理、同步开销和任务粒度对性能的影响。MPI多进程编程则让我学习了进程间通信与协作，并通过不同版本的实现对比，认识到进程数量与通信开销对加速比的制约。GPU编程的实践使我体会到利用CUDA实现大规模并行计算的强大能力，特别是通过PopNext()函数的并行化优化，显著提升了Guess Time的性能。

在最后的期末汇总实验中，我首先在 GPU 服务器中使用 SSE 进行 SIMD 的优化。之后，我融合了前面几次实验的实验结果，又继续对 train.cpp 与已经实现的 MPI 多进程编程进行了多线程编程，并将整个的优化内容汇总到一起，实现了对 Hash Time、Guess Time、Train Time 的最终优化，并且取得了较好的优化结果!

这一个学期的并行程序设计的学习，不仅让我掌握了多种并行编程技术，更重要的是培养了我从分析程序性能与解决问题的能力，深刻认识到并行化在提升程序运行速度方面的巨大潜力，以及在实现过程中需要平衡收益与资源开销，解决如数据依赖、通信开销、负载均衡等复杂问题。这些经验让我对未来大规模数据处理和高性能计算的挑战与解决策略有了更全面的认识.

我的全部实验的github仓库链接：https://github.com/fan-tuaner614/NKU-

诚挚感谢王刚老师与助教学长们在这一学期的付出与教导!!!