南開大學
Nankai University

计算机学院

并行程序设计实验报告

口令猜测 SIMD 编程实验

姓名：熊诚义

学号：2313310

专业：计算机科学与技术

2025.4

目录

1 问题描述

1.1 期末研究报告的大问题

口令猜测算法的并行化优化是本次期末研究的核心问题。具体目标是通过并行化技术（如 SIMD、多线程、MPI、CUDA）加速以下两个关键流程：

1. **PCFG 口令生成：**利用概率上下文无关文法模型，按概率降序生成口令猜测列表。需解决优先队列的并行访问、preterminal 的分布式生成等问题。
2. **MD5 哈希计算：**对生成的口令进行快速哈希运算，以模拟攻击者的实际破解流程。需通过数据级并行提升哈希计算吞吐量。

研究需实现并行算法，分析加速比，并探索不同并行技术的适用性与优化空间。

1.2 SIMD 编程实验涉及的子问题

本次 SIMD 实验聚焦于 MD5 哈希算法的向量化优化，具体子问题包括：

1. **数据集并行设计：**
   - 如何利用 SIMD 指令同时处理多个口令的 MD5 哈希计算。例如，将多个输入字符串的哈希计算映射到向量寄存器中，实现单指令多数据（SIMD）并行。
   - 确保输入数据（口令字符串）的内存对齐与布局适合 SIMD 操作，减少冗余内存访问。
2. **算法适配与优化：**
   - 分析 MD5 的分块处理（512-bit 切片）与循环运算，识别可向量化的步骤（如位运算、模加操作）。
   - 消除分支判断对并行化的影响，例如通过掩码操作或条件预计算实现无分支代码。
3. **性能对比与验证：**
   - 对比 SIMD 并行化前后的计算吞吐量（如每秒哈希次数），验证加速效果。
   - 分析不同 SIMD 指令集（SSE/NEON）的性能差异，探索硬件资源利用率上限。

关键挑战：

- MD5 的运算存在前后依赖（如缓冲区更新），需保证单口令计算的顺序性，但不同口令的计算完全独立，适合 SIMD 并行化。
- 需处理输入口令长度不一致的问题，可能通过填充或批量分组解决。

通过上述优化，目标是在单核上显著提升 MD5 哈希计算的速度，为后续多线程或分布式并行提供基础支持。

2 设计思路

我设计的 SIMD 并行化 MD5 算法基于 ARM NEON 指令集，核心思想是**横向并行化**，即同时处理多个独立消息，而非加速单个消息的内部计算。利用 NEON 的 SIMD 向量指令，每个向量操作可同时处理 4 个 32 位值。

主要设计要点

1. **向量化基本运算：**将 MD5 核心函数 (F、G、H、I) 转换位 NEON 向量操作
2. **批处理多个消息：**一次处理 4 个消息以充分利用 SIMD 指令宽度

3 算法实现（伪代码）

基本向量函数实现

```
// NEON 版本的基本 MD5 函数
function F_SIMD(x, y, z):
    temp1 = vandq_u32(x, y)       // x & y
    temp2 = vbicq_u32(z, x)       // z & ~x
    return vorrq_u32(temp1, temp2) // 返回 (x & y) | (~x & z)

function G_SIMD(x, y, z):
    temp1 = vandq_u32(x, z)       // x & z
    temp2 = vbicq_u32(y, z)       // y & ~z
    return vorrq_u32(temp1, temp2) // 返回 (x & z) | (y & ~z)

function H_SIMD(x, y, z):
    return veorq_u32(veorq_u32(x, y), z) // 返回 x ^ y ^ z

function I_SIMD(x, y, z):
    temp = vorrq_u32(x, vmvnq_u32(z)) // x | (~z)
    return veorq_u32(y, temp)         // 返回 y ^ (x | (~z))

// 循环左移的 SIMD 实现
function ROTATELEFT_SIMD(num, n):
    shift_left = vdupq_n_s32(n)
    shift_right = vdupq_n_s32(-(32-n))
    left_part = vshlq_u32(num, shift_left)
    right_part = vshlq_u32(num, shift_right)
    return vorrq_u32(left_part, right_part)
```

主算法流程

```
function MD5Hash_SIMD(inputs, states):
    batchSize = 4  // NEON 可以同时处理 4 个 32 位整数

    // 处理每一批 4 个消息
```

```
    for batchStart = 0 to inputs.size() step batchSize:
        // 预处理消息
        paddedMessages = []
        messageLengths = []

        // 为每个消息准备填充后的数据
        for i = 0 to batchSize-1:
            if batchStart + i >= inputs.size(): break
            paddedMessage = StringProcess(inputs[batchStart + i], &length)
            paddedMessages.add(paddedMessage)
            messageLengths.add(length)

        // 检查所有消息长度是否相同
        if 任一消息长度不同:
            // 使用普通方法分别处理
            for i = 0 to batchSize-1:
                if batchStart + i >= inputs.size(): break
                MD5Hash(inputs[batchStart + i], states[batchStart + i])
            继续下一批

        // 初始化 MD5 状态
        for i = 0 to batchSize-1:
            if batchStart + i >= inputs.size(): break
            初始化 states[batchStart + i] 的四个 32 位值

        blocks = messageLengths[0] / 64  // 块数

        // 处理每个块
        for blockIdx = 0 to blocks-1:
            // 准备 SIMD 数据
            x[16] = {}  // 16 个 uint32x4_t 向量

            // 将 4 个消息的对应块加载到 SIMD 寄存器
            for i = 0 to 15:
                values[4] = {}
                for j = 0 to 3:
                    // 组合字节为 32 位字
                    msgIdx = batchStart + j
                    values[j] = 组合第 j 个消息中对应的 4 个字节

                // 加载到 SIMD 向量
                x[i] = 加载 values 到向量寄存器
```

```
47
48              // 加载当前状态
49              a, b, c, d = 加载当前状态到 SIMD 向量
50              original_a, original_b, original_c, original_d = a, b, c, d
51
52              // 执行 MD5 四轮变换
53              // Round 1 - 16 个操作
54              // Round 2 - 16 个操作
55              // Round 3 - 16 个操作
56              // Round 4 - 16 个操作
57
58              // 更新状态
59              a = a + original_a
60              b = b + original_b
61              c = c + original_c
62              d = d + original_d
63
64              // 存储结果
65              将 a, b, c, d 的结果存回 states 数组
66
67          // 字节序调整
68          for i = 0 to batchSize-1:
69              if batchStart + i >= inputs.size(): break
70              调整 states[batchStart + i] 的字节序
71
72          // 释放资源
73          释放 paddedMessages
```

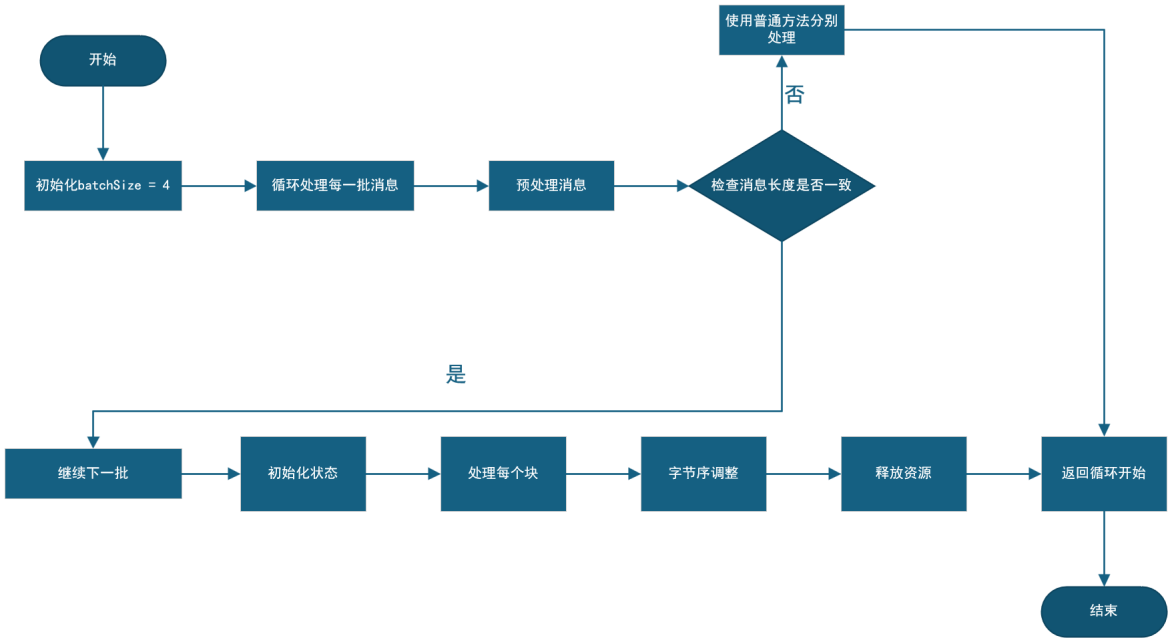

图 3.1: 算法流程图

4 复杂性分析

时间复杂性

- **串行版本：** O(N*L)，其中 N 是消息数量，L 是平均消息长度
- **SIMD 版本理论值：** O(N*L/4)，因为同时处理 4 个消息
- **SIMD 版本实际值：** 约 O(N*L/k)，其中 k<4，因为条件分支和长度检查 (O(N))

空间复杂性

- **串行版本：** O(L)，仅需存储当前处理的消息
- **SIMD 版本：** O(4L)，需同时存储 4 个消息的数据

5 实验及结果分析

5.1 正确性验证

在进行串行版本与 SIMD 版本的性能分析之前，首先需要对基本的正确性进行验证，即能够利用 SIMD 指令，做到一次性产生多个消息（口令）的哈希值，并且哈希值正确。

于是我修改了 correctness.cpp 中的代码，将我所实现的 SIMD 的 MD5 哈希函数添加进去，与标准的 MD5 哈希函数进行对比，验证哈希值是否一致。代码如下：

```
// 测试字符串
string testStr = "bvaisdbjasdkafkasdfnavkjnakdjfejfanjsdnfkajdfkajdfjkwanfdjaknsvjkanbjbjadfaj
// 标准 MD5 计算
bit32 state[4];
MD5Hash(testStr, state);

cout << " 标准 MD5 结果: ";
for (int i1 = 0; i1 < 4; i1 += 1)
{
    cout << std::setw(8) << std::setfill('0') << hex << state[i1];
}
cout << endl;

// SIMD 版本 MD5 计算
vector<string> inputs(4, testStr); // 4 个相同的输入
vector<bit32*> simdStates;
for (int i = 0; i < 4; i++) {
    simdStates.push_back(new bit32[4]);
}

MD5Hash_SIMD(inputs, simdStates);

cout << "SIMD MD5 结果: ";
for (int i1 = 0; i1 < 4; i1 += 1)
{
    cout << std::setw(8) << std::setfill('0') << hex << simdStates[0][i1];
}
cout << endl;

// 检查结果是否一致
bool match = true;
for (int i = 0; i < 4; i++) {
    if (state[i] != simdStates[0][i]) {
        match = false;
        break;
    }
}

cout << " 结果比较: " << (match ? " 一致 " : " 不一致 ") << endl;
```

使用编译指令 g++ correctness.cpp train.cpp guessing.cpp md5.cpp -o main 进行编译后使用 qsub qsub.sh 提交查看 test.o 文件中的结果。可见实现的 SIMD 的哈希函数功能正确。

```
guess > ≡ test.o
1   标准MD5结果: bba46eb8b53cf65d50ca54b2f8afd9db
2   SIMD MD5结果: bba46eb8b53cf65d50ca54b2f8afd9db
3   结果比较: 一致 ✓
4
5   Authorized users only. All activities may be monitored and reported.
```

图 5.2: 正确性验证

5.2 实验数据

不进行编译优化

```
Guess time:7.56233seconds
Hash time:9.39982seconds
Train time:94.4916seconds
```

```
Guess time:8.01063seconds
Hash time:28.2443seconds
Train time:91.1408seconds
```

图 5.3: 串行（左）与 SIMD（右）运行时间对比

O1 编译优化

```
Guess time:0.65779seconds
Hash time:3.19223seconds
Train time:29.0497seconds
```

```
Guess time:0.568879seconds
Hash time:3.76851seconds
Train time:26.0942seconds
```

图 5.4: 串行（左）与 SIMD（右）运行时间对比

O2 编译优化

```
Guess time:0.58115seconds
Hash time:2.98403seconds
Train time:29.5507seconds
```

```
Guess time:0.564371seconds
Hash time:3.65993seconds
Train time:27.8783seconds
```

图 5.5: 串行（左）与 SIMD（右）运行时间对比

5.3 结果分析

由所得到的实验数据发现，相较于串行版本的 MD5 哈希算法，经过我实现的 SIMD 并行优化后的版本并未实现加速，反而发生了减速。

对此，我思考为什么会出现这样的情况，我猜测可能的原因是算法本身的顺序特性与 SIMD 并行模型不匹配，再加上实现中的额外开销。

1. **MD5 算法的顺序依赖性限制**

MD5 算法本质上是一种高度顺序的算法，其设计特点使它不适合 SIMD 并行化：

- 每一轮运算都依赖于前一轮运算的结果
- 算法中的四轮运算必须严格按顺序执行
- 数据流动路径是单向的，无法重组为并行格式

2. **数据排列和管理开销过大**

SIMD 实现中引入了大量额外开销：

- **数据重排：**需要将散列的数据整合成 SIMD 友好的格式
- **额外内存拷贝：**由于兼容性问题，使用 memcpy 替代直接向量加载以及存储指令
- **内存管理：**频繁分配和释放临时数组（如 batchStates）

同时我在我所实现的代码中也发现相关部分显示了显著的开销：

```
    // 创建临时数组
    vector<bit32*> batchStates;
    for (int j = 0; j < currentBatchSize; j++) {
        batchStates.push_back(new bit32[4]);
    }
    // ... 处理...
    // 释放内存
    for (int j = 0; j < currentBatchSize; j++) {
        delete[] batchStates[j];
    }
```

6 SIMD 进阶

报告以上部分是本次 SIMD 实验的基础要求，即基本的正确性验证和给出实验数据并分析为什么没有实现相对串行算法的加速。

在进阶要求部分我将**尝试实现相对串行算法的加速**并通过查阅资料等方式**说明编译时的选项会对加速比产生什么影响以及为什么会产生影响**。

一个“失败”的尝试

我尝试将 batchSize 从 4 增加至 8，由于 NEON 指令集实际上并不支持这么大的寄存器，于是我所做的操作是**每次将每个函数执行两轮 SIMD 的哈希**。

通过修改 correctness.cpp 对现在的 MD5 哈希函数与串行的输出是否一致。

```
测试不同输入组合:
输入 1 (password1): 一致 ✓
输入 2 (password2): 一致 ✓
输入 3 (password3): 一致 ✓
输入 4 (password4): 一致 ✓
输入 5 (password5): 一致 ✓
输入 6 (password6): 一致 ✓
输入 7 (password7): 一致 ✓
输入 8 (password8): 一致 ✓
总体比较: 所有结果一致 ✓

```

图 6.6: 正确性验证

可见通过测试不同输入组合，结果都一致，即实现正确，部分核心代码如下：

```
    for (int blockIdx = 0; blockIdx < n_blocks; blockIdx++) {
        // 准备SIMD数据
        // 准备SIMD数据 - 分两批
        uint32x4_t x1[16], x2[16]; // 前4个和后4个消息的数据

        // 优化数据准备 - 循环展开版本
int base_offset = blockIdx * 64;
for (int i = 0; i < 16; i += 4) {
    // 批次1: 处理前4个消息
    for (int j = 0; j < 4; j++) {
        // 处理x[i]到x[i+3]的4个块，展开循环
        values[i][j] = (paddedMessages[j][4 * i + base_offset]) |
                       (paddedMessages[j][4 * i + 1 + base_offset] << 8) |
                       (paddedMessages[j][4 * i + 2 + base_offset] << 16) |
                       (paddedMessages[j][4 * i + 3 + base_offset] << 24);

        values[i+1][j] = (paddedMessages[j][4 * (i+1) + base_offset]) |
                         (paddedMessages[j][4 * (i+1) + 1 + base_offset] << 8) |
                         (paddedMessages[j][4 * (i+1) + 2 + base_offset] << 16) |
                         (paddedMessages[j][4 * (i+1) + 3 + base_offset] << 24);

        values[i+2][j] = (paddedMessages[j][4 * (i+2) + base_offset]) |
                         (paddedMessages[j][4 * (i+2) + 1 + base_offset] << 8) |
                         (paddedMessages[j][4 * (i+2) + 2 + base_offset] << 16) |
                         (paddedMessages[j][4 * (i+2) + 3 + base_offset] << 24);

        values[i+3][j] = (paddedMessages[j][4 * (i+3) + base_offset]) |
                         (paddedMessages[j][4 * (i+3) + 1 + base_offset] << 8) |
                         (paddedMessages[j][4 * (i+3) + 2 + base_offset] << 16) |
                         (paddedMessages[j][4 * (i+3) + 3 + base_offset] << 24);
    }

    // 批次2: 处理后4个消息
```

图 6.7: 数据处理

```
// 保存原始状态
// 保存原始状态 - 批次1
uint32x4_t original_a1 = a1;
uint32x4_t original_b1 = b1;
uint32x4_t original_c1 = c1;
uint32x4_t original_d1 = d1;

// 保存原始状态 - 批次2
uint32x4_t original_a2 = a2;
uint32x4_t original_b2 = b2;
uint32x4_t original_c2 = c2;
uint32x4_t original_d2 = d2;
```

图 6.8: 保存原始状态

虽然正确性得到了验证，但在输出 Hashtime 时遇到了点问题，不能输出各阶段的时间。

```
309   here
310   Guesses generated: 170685
311   Guesses generated: 315647
312   Guesses generated: 459774
313   Guesses generated: 559924
314   Guesses generated: 660105
315   Guesses generated: 797589
316   Guesses generated: 927462
317   Guesses generated: 1051391
318
319   Authorized users only. All activities may be monitored and reported.
```

图 6.9: test.o 输出

由于不知道如何真正去解决这个问题，于是我更改了优化方案，仍然在 batchSize = 4 时进行优化。

6.1 加速尝试

以下是我在原 SIMD 并行化基础上进而进行的优化

1. **修改状态数组加载和存储**

```
    // 原：只加载第一个元素的状态
uint32x4_t a = vld1q_u32(&states[0][0]);

// 修改为：
alignas(16) uint32_t a_values[4], b_values[4], c_values[4], d_values[4];
for (int j = 0; j < 4; j++) {
    a_values[j] = states[j][0];
    b_values[j] = states[j][1];
    c_values[j] = states[j][2];
```

```
        d_values[j] = states[j][3];
    }
    uint32x4_t a = vld1q_u32(a_values);
    uint32x4_t b = vld1q_u32(b_values);
    uint32x4_t c = vld1q_u32(c_values);
    uint32x4_t d = vld1q_u32(d_values);
```

2. **修改状态存储方式**

```
    // 原：只更新第一个元素
    vst1q_u32(&states[0][0], a);

    // 修改为：
    vst1q_u32(a_values, a);
    vst1q_u32(b_values, b);
    vst1q_u32(c_values, c);
    vst1q_u32(d_values, d);

    for (int j = 0; j < 4; j++) {
        states[j][0] = a_values[j];
        states[j][1] = b_values[j];
        states[j][2] = c_values[j];
        states[j][3] = d_values[j];
    }
```

6.2 加速结果

以下分别是在不进行编译优化、O1 编译优化、O2 编译优化下所得到的测试结果：

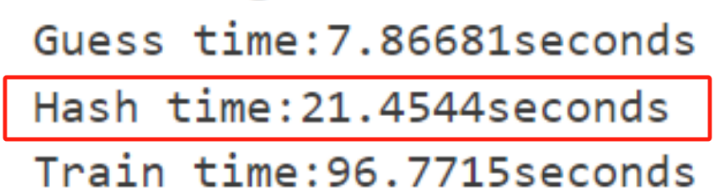

图 6.10: 不进行编译优化

图 6.11: O1 优化（左）与 O2 优化（右）运行时间

对比分析可见，在不进行编译优化时，相比于原 SIMD 并行优化得到了加速，但相比于串行算法仍然未实现加速。

而 O1、O2 优化相较于串行算法均得到了加速，加速比分别为 **1.05454** 和 **1.04428**

6.3 分析编译选项对加速比的影响

1. **无优化**

   SIMD 指令会按原样执行，没有任何指令级优化，函数调用不会内联，导致更多的函数调用开销，寄存器分配效率低，可能导致频繁的寄存器溢出，内存访问模式未优化，对 SIMD 性能影响很大。

2. **-O1 优化**

   基本的函数内联，减少 FF_SIMD、GG_SIMD 等函数调用开销，简单的死代码消除，减少不必要的 SIMD 操作，改进的寄存器分配，减少内存访问，基本的指令调度优化

3. **-O2 优化**

   积极的函数内联，几乎所有 SIMD 函数都会内联，循环展开，对 MD5 四轮变换循环特别有效，高级指令调度，充分利用 CPU 流水线，内存访问模式优化，改善缓存效率，更智能的寄存器分配，减少 NEON 寄存器溢出

6.4 为什么编译优化会影响 SIMD 加速

1. 内存访问模式优化

```
alignas(16) uint32_t values[16][4]; // 对齐的内存分配
```

   高级优化能更好地处理对齐内存访问，优化编译器能减少内存访问次数，合并读写操作，缓存预取优化对 SIMD 批处理特别有效。

2. 指令级并行度提升

```
FF_SIMD(a, b, c, d, x[0], s11, 0xd76aa478);
FF_SIMD(d, a, b, c, x[1], s12, 0xe8c7b756);
```

   高级优化能更好地调度这些 SIMD 指令，使其并行执行，减少了指令间依赖等待时间。

3. 循环优化

```
for (int blockIdx = 0; blockIdx < n_blocks; blockIdx++) {
    for (int i = 0; i < 16; i++) {
        for (int j = 0; j < batchSize; j++) {
            // ...
        }
    }
}
```

-O2 可以自动展开内部循环，减少循环控制开销，提高 SIMD 指令效率。

4. 函数内联

```
1  inline void FF_SIMD(uint32x4_t &a, uint32x4_t b, uint32x4_t c, uint32x4_t d, uint32x4_t x,
```

更高优化级别会积极内联这些函数，消除函数调用开销，允许跨函数优化

7 Git 项目链接

Parallel