南開大學
Nankai University

计算机学院

并行程序设计实验报告

口令猜测选题 SIMD 编程

姓名：周重天

学号：2311082

专业：计算机科学与技术

2025 年 4 月 29 日

目录

1 摘要

本实验报告基于并行程序设计课程的口令猜测选题，完成了 SIMD 基础要求的实验。在 ARM 服务器上利用 NEON 指令集实现了 MD5 哈希算法的向量化，能够同时处理多个口令并生成正确的哈希值。报告详细描述了问题背景、算法设计、实验实现、性能测试及分析，重点探讨了 SIMD 算法未实现性能加速的原因，并提出了改进建议。

2 引言

2.1 问题描述

口令猜测任务通过计算候选口令的 MD5 哈希值并与目标哈希值比较来验证口令的正确性。本实验旨在利用 SIMD 技术（基于 ARM 平台的 NEON 指令集）优化 MD5 哈希算法，实现一次性计算多个口令的哈希值。根据实验指导书要求，需验证 SIMD 算法的正确性，测试不同输入规模下的性能，并分析未实现相对于串行算法加速的原因。本实验聚焦于 MD5 算法中 FF、GG、HH、II 函数的向量化优化，探索 NEON 指令在位运算和数据并行中的应用。

3 实验环境

- **硬件**：ARM 服务器，支持 NEON 指令集。
- **软件**：C++ 编程语言，NEON Intrinsics 库，g++ 编译器。

4 SIMD 并行化的基本思路

4.1 基础工作：修改.h 头文件中的宏定义

在实验提供的原始头文件中，已经提供了九个宏定义，它们的工作是在 md5 算法迭代的过程中要经历的 64 轮结构固定的运算。在这里，我们的任务是对其使用 arm 平台的 neon 将其进行向量化，转化为适合并行计算的 simd 版本。

具体思路其实很简单，就是使用 neon 将其处理的元素向量化。原始宏定义（F, G, H, I, ROTATELEFT, FF, GG, HH, II）是为串行 MD5 算法设计的，处理单个 32 位数据。SIMD 版本的宏（$\mathrm{F}_S IMD, G_S IMD, H_S IMD, I_S IMD, ROTATELEFT_S IMD, FF_S IMD, GG_S IMD, HH_S IMD, II_S IMD$ NEO

F、G、H、I 四个函数：

这四个函数是 md5 迭代过程中最基础的 4 个函数，并且，事实上，在真正的迭代过程中，仅仅使用了 FF、GG、HH、II 四个宏定义，而这四个宏定义中才调用了 FGHI 四个函数。

向量化的过程取 F 和 G 两个宏定义为例：

原始的 F 宏的定义如下：

```
1 #define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
```

在并行化的过程中，我们最终将其修改为如下结果：

```
#define F_SIMD(x, y, z) \
    vorrq_u32(vandq_u32((x), (y)), vandq_u32(vmvnq_u32(x), (z)))
```

这里可以看到，原始定义是先将 x 和 y 相与，然后将 x 取反在和 z 相与，最终将两个结果相或。这里我们使用 neon 中的 vandq_u32 进行两个向量的与操作，使用 vmvnq_u32 进行单个向量的取反操作。最终使用 vorrq_u32 进行两个向量的或操作，如以上代码所示，这样就完成了 F 宏的 simd 修改。

同理，G 宏原定义如下：

```
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
```

其对应的向量化版本为：

```
#define G_SIMD(x, y, z) \
    vorrq_u32(vandq_u32((x), (z)), vandq_u32((y), vmvnq_u32(z)))
```

这里，我们使用 vandq_u32((x), (z)) 获取了 x 向量和 z 向量的与结果，使用 vandq_u32((y), vmvnq_u32(z)) 获取了 y 和 z 的与结果，然后使用 vorrq_u32 获得了原定义中的表达式 (((x) (z)) | ((y) (z)))。

另外的 H、I 两个宏定义其过程也同上，因此不做赘述。

ROTATELEFT：该宏的原定义：

```
#define ROTATELEFT(num, n) (((num) << (n)) | ((num) >> (32-(n))))
```

借助大模型，我知道了对于定义中的逻辑左移操作，对应的 neon 中的语法为 $vshlq_{nu}32$, 对于逻辑右移，neon 中的语法应该是 vshrq_n_u32。知道了这些，simd 版本也就不难修改了。

```
#define ROTATELEFT_SIMD(x, n) \
    vorrq_u32(vshlq_n_u32((x), (n)), vshrq_n_u32((x), 32 - (n)))
```

FF、GG、HH、II 四个函数：

修改到这里，已经对查阅 neon 对应的语法的流程很熟悉了，具体的过程也没怎么变，因此直接给出 FF 宏修改的结果作为举例：

```
// MD5核心操作的SIMD宏版本
#define FF_SIMD(a, b, c, d, x, s, ac) { \
    uint32x4_t ac_vector = vdupq_n_u32(ac); \
    (a) = vaddq_u32((a), vaddq_u32(vaddq_u32(F_SIMD((b), (c), (d)), (x)),
        ac_vector)); \
    (a) = ROTATELEFT_SIMD((a), (s)); \
    (a) = vaddq_u32((a), (b)); \
}
```

在以上过程中，涉及到的新语法不多，只有：

1. 使用 vaddq_u32 进行 4 个 32 位向量的加法。

2. 使用 vdupq_n_u32 将 32 位的标量常量加载为向量常量。

至于函数调用乃至于宏调用，与 c++ 基础语法一致。

至此，九个宏定义全部修改为 simd 的并行版本。

4.2 整合：md5hash 函数的修改

为了实现口令猜测任务的 SIMD 并行化，本实验基于 ARM NEON 指令集对 MD5Hash 函数进行了修改，设计了 MD5Hash_SIMD 函数，以一次性处理多个输入字符串的 MD5 哈希计算。以下是 MD5Hash_SIMD 的伪代码及并行化实现分析。

4.2.1 伪代码

```
函数 MD5Hash_SIMD(inputs, input_count, states):

    设置 simd_width ← 4  // 每批最多处理4个字符串

    遍历所有输入，按simd_width划分为多个批次：
    for batch 从 0 到 input_count 以步长 simd_width:

        current_batch_size ← min(simd_width, input_count - batch)

        // ---- 初始化每个输入的预处理数据 ----
        for i 从 0 到 current_batch_size - 1:
            paddedMessages[i] ← StringProcess(inputs[batch + i])
            messageLengths[i] ← 得到消息长度
            n_blocks[i] ← messageLengths[i] / 64

        // ---- 初始化SIMD状态 ----
        state0 ← [0x67452301, 0x67452301, 0x67452301, 0x67452301]
        state1 ← [0xefcdab89, 0xefcdab89, 0xefcdab89, 0xefcdab89]
        state2 ← [0x98badcfe, 0x98badcfe, 0x98badcfe, 0x98badcfe]
        state3 ← [0x10325476, 0x10325476, 0x10325476, 0x10325476]

        max_blocks ← 当前批次中最大的 n_blocks[i]

        // ---- 块级别处理：对每个block进行MD5核心迭代 ----
        for block 从 0 到 max_blocks - 1:

            // 构造当前block的输入向量x[0..15]
            for j 从 0 到 15:
                for i 从 0 到 current_batch_size - 1:
```

```
            if block < n_blocks[i]:
                x[j][i] ← 从paddedMessages[i]中提取第j项（转换为uint32小端格式）

    // 复制状态向量以供计算
    a ← state0
    b ← state1
    c ← state2
    d ← state3

    // ---- Round 1 到 Round 4 的16步操作 ----
    执行 FF_SIMD 宏 16次 (Round 1)
    执行 GG_SIMD 宏 16次 (Round 2)
    执行 HH_SIMD 宏 16次 (Round 3)
    执行 II_SIMD 宏 16次 (Round 4)

    // 状态更新
    state0 ← state0 + a
    state1 ← state1 + b
    state2 ← state2 + c
    state3 ← state3 + d

// ---- 字节序转换（小端 → 大端） ----
state0 ← ByteSwapSIMD(state0)
state1 ← ByteSwapSIMD(state1)
state2 ← ByteSwapSIMD(state2)
state3 ← ByteSwapSIMD(state3)

// ---- 存储结果 ----
for i 从 0 到 current_batch_size - 1:
    states[batch + i][0] ← state0[i]
    states[batch + i][1] ← state1[i]
    states[batch + i][2] ← state2[i]
    states[batch + i][3] ← state3[i]

// ---- 内存释放 ----
for i 从 0 到 current_batch_size - 1:
    释放 paddedMessages[i]
```

4.2.2 并行化实现分析

MD5Hash_SIMD 函数通过利用 ARM NEON SIMD 指令集并行化 MD5 哈希计算，一次性处理四个输入字符串，从而显著提高口令猜测任务的吞吐量。以下是对并行化实现方式的分析：

1. **批次处理与 SIMD 宽度**：函数以批次方式处理输入，每次批次最多处理 4 个输入（SIMD 宽度为 128 位，每个输入使用 32 位寄存器）。通过 `current_batch_size` 动态调整批次大小，确保处理剩余输入时不会越界。

2. **SIMD 状态向量初始化**：使用 `uint32x4_t` 类型（NEON 128 位向量）初始化 MD5 的四个状态寄存器（`state0` 到 `state3`），每个寄存器存储 4 个输入的对应状态值（如 `0x67452301`）。这允许同时对 4 个输入执行相同的初始化操作。

3. **并行消息预处理**：对每个输入调用 `StringProcess` 函数进行消息填充，生成符合 MD5 要求的 512 位块。填充后的消息存储在 `paddedMessages` 数组中，并记录每个输入的块数（`n_blocks`）。

4. **SIMD 数据加载与内存优化**：在处理每个 512 位块时，将 4 个输入的 16 个 32 位字（`x[0..15]`）加载到 SIMD 向量中。代码通过检查内存对齐（`addr & 0x3`）选择高效的 32 位加载或逐字节组合，减少内存访问开销。此外，使用 `__builtin_prefetch` 预取下一组数据，优化缓存命中率。

5. **状态更新与字节序调整**：每轮运算后，使用 `vaddq_u32` 并行更新状态向量。最终结果通过 `ByteSwapSIMD`（基于 `vrev32q_u8`）调整字节序，转换为大端序，符合 MD5 标准输出格式。

6. **结果存储与内存管理**：使用 `vst1q_u32` 将 SIMD 向量中的结果提取到 `states` 数组，确保每个输入的哈希值正确存储。动态分配的 `paddedMessages` 数组在批次处理完成后释放，避免内存泄漏。

通过上述方式，`MD5Hash_SIMD` 利用 NEON 指令集实现了 MD5 算法的并行化，理论上可将 4 个输入的哈希计算合并为一次运算。然而，由于内存访问、数据预处理和指令流水线等开销，实际加速比可能受限，需进一步通过实验数据分析瓶颈。

4.3 正确性验证

为了验证 `MD5Hash_SIMD` 函数的正确性，设计了测试程序，通过比较串行 `MD5Hash` 函数和并行 `MD5Hash_SIMD` 函数的输出，确认 SIMD 实现是否能正确计算多个输入的 MD5 哈希值。以下是正确性验证的伪代码、实现思路及测试结果。

4.3.1 伪代码

```
// 测试串行版本
定义 testString 为长输入字符串
定义 state 为 4x32位数组
调用 MD5Hash(testString, state)
输出 "串行MD5结果: "
对于 i 从 0 到 3
    输出 state[i] (16进制, 8位填充0)
结束循环
输出换行
```

```
// 测试SIMD并行版本
输出 "测试SIMD并行版本:"
定义 test_count = 4
定义 inputs 为 test_count 个字符串数组
对于 i 从 0 到 test_count-1
    inputs[i] = testString + "_" + i
结束循环

// 计算串行参考结果
定义 serial_states 为 test_count x 4 x 32位数组
对于 i 从 0 到 test_count-1
    调用 MD5Hash(inputs[i], serial_states[i])
结束循环

// 准备SIMD输出存储
定义 simd_states 为 test_count 个 4x32位数组指针
对于 i 从 0 到 test_count-1
    simd_states[i] = 新分配 4x32位数组
结束循环

// 调用SIMD版本
调用 MD5Hash_SIMD(inputs, test_count, simd_states)

// 输出并比较结果
对于 i 从 0 到 test_count-1
    输出 "SIMD结果 " + (i+1) + ": "
    对于 j 从 0 到 3
        输出 simd_states[i][j] (16进制, 8位填充0)
    结束循环函数 main():

// 准备测试字符串
定义 testString ← "一段较长的字符串"

// 使用串行版本计算MD5并输出结果
调用 MD5Hash(testString, state)
打印 "串行MD5结果: " + state[0..3] (十六进制)

// 构造多个不同输入用于SIMD测试
定义 test_count ← 4
for i 从 0 到 test_count - 1:
    inputs[i] ← testString + "_" + i
```

```
// 串行计算每个输入的参考结果
for i 从 0 到 test_count - 1:
    调用 MD5Hash(inputs[i], serial_states[i])

// 初始化输出数组用于存放SIMD结果
for i 从 0 到 test_count - 1:
    simd_states[i] ← 新建 bit32[4]

// 调用SIMD版本进行并行计算
调用 MD5Hash_SIMD(inputs, test_count, simd_states)

// 输出和比较每组SIMD结果与串行参考
for i 从 0 到 test_count - 1:
    打印 "SIMD结果 i: " + simd_states[i][0..3]
    打印 "串行结果: " + serial_states[i][0..3]

    match ← true
    for j 从 0 到 3:
        if simd_states[i][j]  serial_states[i][j]:
            match ← false
    if match:
        打印 "(匹配)"
    else:
        打印 "(不匹配!)"

// 释放内存
for i 从 0 到 test_count - 1:
    释放 simd_states[i]

返回 0
```

4.3.2 验证结果

运行正确性验证程序，输出如下：

串行 MD5 结果：bba46eb8b53cf65d50ca54b2f8afd9db

SIMD 并行版本结果：
SIMD 结果 1:6c386c04aed94575f5cfa489174a1742 串行结果:6c386c04aed94575f5cfa489174a1742 (匹配)
SIMD 结果 2:678d8f671ede972fabeefeb62b00c8e4 串行结果:678d8f671ede972fabeefeb62b00c8e4 (匹配)

SIMD 结果 3:61470546a002628e9b6361082c34f448 串行结果:61470546a002628e9b6361082c34f448 (匹配)
SIMD 结果 4:dab8f6ba03099d67858dda3ac75e0227 串行结果:dab8f6ba03099d67858dda3ac75e0227 (匹配)

从验证结果可以看出，我们实现的 SIMD 并行版本 MD5 哈希函数能够正确处理输入，所有四个通道的输出结果均与串行版本完全一致。这证明了我们的并行化实现在功能上是正确的，可以用于实际应用场景。

值得注意的是，测试使用的是一个较长的字符串（超过 512 位，需要处理多个块），这确保了我们的算法能够正确处理多块输入的情况。同时，通过对所有四个并行通道的结果进行比较，也验证了 SIMD 指令在向量化处理过程中的正确性和一致性。

这个验证过程是确保 SIMD 并行化 MD5 实现正确性的重要步骤，为后续的性能优化和实际应用奠定了基础。

4.4 进阶要求

4.4.1 为实现加速做的措施

为了提升 `MD5Hash_SIMD` 函数的性能，本实验在 ARM NEON SIMD 实现中引入了针对数据加载阶段的优化措施，旨在减少内存访问开销、提升指令流水线效率，并结合编译优化实现相对于串行 `MD5Hash` 函数的加速。以下为具体优化措施及其实现细节。

1. **4 路循环展开**：在加载输入块到 SIMD 向量 `x[0..15]` 的过程中，原始代码通过单次循环（`j` 从 0 到 15，步长为 1）逐个生成 32 位字。本实验将其优化为 4 路循环展开（`j` 步长改为 4），每次同时处理 4 个字（`values0`, `values1`, `values2`, `values3`），并存储到 `x[j..j+3]`。此优化通过减少循环迭代次数（从 16 次降至 4 次），显著降低了分支预测和循环控制开销，提升了指令流水线利用率。同时，展开后每次循环加载更多数据，增加了数据局部性，有助于缓存命中率的提高。

2. **数据预取与内存对齐优化**：在 4 路循环展开的基础上，进一步引入数据预取和内存对齐优化，以减少内存访问延迟。

   - **数据预取**：通过 `__builtin_prefetch` 指令，在每次循环中提前预取后续 4 个字的数据（`j+4` 至 `j+7`，对应 `paddedMessages[i][4*(j+4)..4*(j+7) + block*64]`）。预取操作将数据提前加载到缓存，减少了后续内存访问的等待时间，特别适用于大输入字符串的连续块处理。
   - **内存对齐优化**：为每个输入检查内存地址是否 32 位对齐（通过 `addr & 0x3` 判断）。若对齐，则使用高效的 32 位直接加载（`*(uint32_t*)addr`），避免逐字节移位和组合操作；若未对齐，则回退到逐字节加载（`paddedMessages[i][..] | ... << 8 | ...`）。此优化显著降低了未对齐访问的性能开销，尤其在处理非对齐输入数据时效果明显。
   - **内存对齐分配与分支消除的进一步优化**：
     在原先基础上，我继续进行了一些优化，`StringProcess` 函数采用 `posix_memalign` 进行 16 字节对齐的内存分配，确保所有 `paddedMessages` 均为对齐地址。这样，在 `MD5Hash_SIMD` 阶段将消息块加载到 `values` 数组时，无需再判断对齐情况，也不再需

要通过 uintptr_t 检查地址或做逐字节拼装。原有的对齐判断和分支代码已被精简为直接的 32 位对齐加载：

```
values0[i] = *(uint32_t*)&paddedMessages[i][4*j + block*64];
values1[i] = *(uint32_t*)&paddedMessages[i][4*(j+1) + block*64];
values2[i] = *(uint32_t*)&paddedMessages[i][4*(j+2) + block*64];
values3[i] = *(uint32_t*)&paddedMessages[i][4*(j+3) + block*64];
```

该优化彻底消除了相关分支和多余的内存访问，使 SIMD 加载更高效，进一步提升了整体哈希计算性能。

通过上述优化，MD5Hash_SIMD 在数据加载阶段的性能得到显著提升。在这些优化之后，在 O1、O2 级别的编译优化的情况下即可实现相对于串行算法的加速，具体的加速结果会放在针对编译优化的讨论一节中进行叙述。

除此之外，我还尝试了在加在数据和存入结果阶段手动进行内存对齐、增加内存缓冲池从而避免多次内存分配和释放、修改 StringProcess 函数称为并行版本等优化，但经过测试，加速比均不如这一版本高，表明其优化效果可能由于我的操作不当并不理想，因此在这里并未加入。

4.4.2 编译优化对加速比的影响

在本实验中，我们对 MD5Hash_SIMD 和串行 MD5Hash 函数进行了性能测试，分别在无优化、-O1 和 -O2 优化级别下记录哈希计算时间，并计算加速比。

profiling 同时，我通过 perf 工具记录下了指令数、cpu cycle、缓存未命中、前后端停顿等性能指标进行分析，以探究编译优化对加速比的影响。以下表格总结了各优化级别下的性能指标：

表 1: 不同优化级别下的性能指标比较

指标	无优化	-O1	-O2
指令数 (亿条)	3863.42	739.66	691.29
周期数 (亿)	3738.49	1074.79	1027.27
IPC	1.03	0.69	0.67
缓存未命中 (百万)	1018.12	999.87	936.89
分支预测失败 (百万)	991.77	637.03	671.06
前端停顿周期 (亿)	794.20	61.20	59.05
前端停顿占比 (%)	21.24	5.69	5.75
后端停顿周期 (亿)	1446.94	686.77	647.82
后端停顿占比 (%)	38.70	63.90	63.06
执行时间 (秒)	146.53	42.23	40.58

此外, 还有三种编译情况下 md5hash 的串并行算法的执行时间与加速比对比:

- **无优化**: 串行版本哈希时间为 9.59138 秒, SIMD 版本为 13.34590 秒, 加速比仅为 0.71867x。perf 数据显示，程序执行约 3863 亿条指令，指令每周期（IPC）为 1.03，表明指令吞吐量较低。前端流水线停顿占比 21.24%，后端停顿占比 38.70%，反映出指令调度和数据依赖的瓶颈。未优化的 MD5Hash_SIMD 中，paddedMessages 的加载未充分利用 SIMD 指令特性，导致内存访问效率低下。此外，循环未展开增加了控制指令开销，限制了并行性能。

表 2: 不同优化级别下串行与 SIMD 哈希性能对比

优化级别	串行哈希时间 (秒)	SIMD 哈希时间 (秒)	加速比	训练时间 (秒)
无优化	9.59138	13.34590	0.71867	97.5477
-O1	3.02053	2.45335	1.23119	27.0872
-O2	3.20098	2.36249	1.35492	27.6180

- **-O1 优化**：串行版本哈希时间降至 3.02053 秒，SIMD 版本降至 2.45335 秒，加速比提升至 1.23119x。`perf` 数据表明，指令数减少至 739 亿条，IPC 降至 0.69，总周期数减至 1075 亿。-O1 优化通过常量传播、死代码消除和简单循环优化，减少了冗余内存访问，并提取了循环内常量表达式，降低了迭代开销。对 `StringProcess` 等小型函数的内联展开进一步减少了函数调用开销。然而，后端流水线停顿占比仍高达 63.90%，表明数据依赖问题尚未完全解决。
- **-O2 优化**：串行版本哈希时间为 3.20098 秒，SIMD 版本为 2.36249 秒，加速比进一步提升至 1.35492x。`perf` 数据显示，指令数降至 691 亿条，IPC 为 0.67，总周期数为 1027 亿。-O2 优化在 `-O1` 基础上引入自动向量化、循环展开和指令调度，显著提升了 `MD5Hash_SIMD` 的效率。例如，`paddedMessages` 的加载被优化为对齐的向量加载，提高了内存访问效率。部分循环展开减少了控制指令开销，指令调度优化了流水线利用率。尽管如此，后端流水线停顿占比仍为 63.06%，表明数据依赖和内存访问延迟仍是性能瓶颈。

通过比较不同优化级别的测试结果和 `perf` 数据可知，编译优化显著提升了 `MD5Hash_SIMD` 的加速比。无优化时，SIMD 版本性能低于串行版本，反映出未优化代码无法充分发挥 SIMD 指令集潜力；而在 `-O1` 和 `-O2` 优化下，SIMD 版本性能超越串行版本，加速比分别达 1.23119x 和 1.35492x。`-O2` 优化通过更激进的向量化与调度略优于 `-O1`，但编译时间较长。

另外，我编译优化后的程序的指令数显著低于无优化版本，达到了仅有不到五分之一的惊人效果。另一方面，周期数也得到显著改善，两种编译优化的情况下的周期数仅有无优化版本的三分之一不到。相比起来，优化后的缓存未命中情况虽然确有改善，但却无明显差距，应该不是性能提升的主要原因。

另一方面，可以看到编译优化之后的前端停顿比无优化版本的前端停顿少了很多，这也符合了我们找到的资料，即编译器在优化过程中可能通过内联函数、减少分支、优化循环（例如循环展开）等手段减少了分支预测失败的概率，从而降低了取指停顿。

但是，后端停顿虽然在绝对数值上也实现了相对于无优化版本的减少，但其在占比方面却比无优化版本增加了将近一倍。这说明 cpu 大多数时间都处于在等待的时间，导致计算和写回的过程性能受到限制。

显然，虽然编译优化后前端停顿得到了优化，但很可能将性能瓶颈转移到了计算、写回等后端进程上，导致了虽然 simd 并行效果改善，但仍然远远不如理论效率。这可能需要优化计算过程的内存排布、内存对齐，还有增加数值预取等操作。但我经过多次探索，进行了这些修改之后的加速比反而降低，因此只能怀着疑问结束实验。

4.4.3 x86 平台上的 SSE 实现

在本实验的拓展阶段，我们将 MD5 哈希算法从普通实现扩展到 x86 平台的 SIMD 向量化指令集实现。通过深入研究 SSE 指令集特性，我们成功设计并实现了多种并行宽度的 MD5 哈希算法，以充分利用现代处理器的向量计算能力。

SSE 指令集简介与特性 SSE (Streaming SIMD Extensions) 是 Intel 为 x86 架构开发的 SIMD 指令集族，自 Pentium III 处理器开始引入。本实验使用的主要是 SSE2 指令集，它包含 128 位宽的 XMM 寄存器，能同时操作：

- 4 个 32 位整数
- 8 个 16 位整数
- 16 个 8 位整数
- 2 个 64 位双精度浮点数
- 4 个 32 位单精度浮点数

在 MD5 算法中，我们主要使用 SSE 处理 32 位整数运算，每个 SSE 寄存器可同时进行 4 个 MD5 哈希计算，从而实现 4 路并行。

SSE 指令在 MD5 中的应用 MD5 算法的核心操作主要包括位运算 (与、或、异或)、整数加法和位移操作。我们使用以下 SSE 指令映射这些操作：

```
_mm_and_si128()      // 按位与
_mm_or_si128()       // 按位或
_mm_xor_si128()      // 按位异或
_mm_andnot_si128()   // 按位与非 (~a & b)
_mm_add_epi32()      // 4个32位整数并行加法
_mm_slli_epi32()     // 4个32位整数并行左移
_mm_srli_epi32()     // 4个32位整数并行右移
_mm_set1_epi32()     // 用相同值填充4个整数位置
_mm_set_epi32()      // 设置4个不同的整数值
_mm_load_si128()     // 从对齐内存加载到SSE寄存器
_mm_store_si128()    // 从SSE寄存器存储到对齐内存
```

MD5 核心函数的 SSE 实现 基于上述指令，我们重新实现了 MD5 的四个核心变换函数 F、G、H、I 以及轮转操作：

```
// 原始MD5位运算函数的SSE向量化版本
#define F_SSE(x, y, z) _mm_or_si128(_mm_and_si128((x), (y)), _mm_andnot_si128((x),
    (z)))

// 向左轮转操作的SSE实现
#define ROTATELEFT_SSE(x, n) \
    _mm_or_si128(_mm_slli_epi32((x), (n)), _mm_srli_epi32((x), (32-(n))))
```

```

// MD5每轮变换的SSE向量化版本
#define FF_SSE(a, b, c, d, x, s, ac) { \
    __m128i tmp = F_SSE((b), (c), (d)); \
    tmp = _mm_add_epi32(tmp, (x)); \
    tmp = _mm_add_epi32(tmp, _mm_set1_epi32(ac)); \
    (a) = _mm_add_epi32((a), tmp); \
    (a) = ROTATELEFT_SSE((a), (s)); \
    (a) = _mm_add_epi32((a), (b)); \
}
```

并行处理流程设计 MD5_SSE 函数的整体设计考虑多个输入的并行处理：

```
void MD5Hash_SSE(const string* inputs, int count, bit32* states) {
    // 确保输入计数是4的倍数（SSE自然宽度）
    assert(count % 4 == 0);
    // 预处理所有输入字符串
    Byte** paddedMessages = new Byte*[count];
    int* messageLengths = new int[count];
    for (int i = 0; i < count; i++) {
        paddedMessages[i] = StringProcess(inputs[i], &messageLengths[i]);
    }
    // 按批次处理，每批4个
    for (int batch = 0; batch < count; batch += 4) {
        // 初始化SSE寄存器，加载MD5初始状态
        __m128i a = _mm_set1_epi32(0x67452301);
        __m128i b = _mm_set1_epi32(0xefcdab89);
        __m128i c = _mm_set1_epi32(0x98badcfe);
        __m128i d = _mm_set1_epi32(0x10325476);
        // 处理每个64字节块
        int n_blocks = messageLengths[batch] / 64;
        for (int i = 0; i < n_blocks; i++) {
            // 加载4个消息的对应块到SSE寄存器
            __m128i x[16];
            // ...（详细加载数据代码）
            // 保存当前状态
            __m128i aa = a, bb = b, cc = c, dd = d;

            // 执行MD5四轮变换，每个指令同时处理4个哈希

            // 累加到状态变量
            a = _mm_add_epi32(a, aa);
            b = _mm_add_epi32(b, bb);
            c = _mm_add_epi32(c, cc);
            d = _mm_add_epi32(d, dd);
        }
        // 存储结果并进行字节序调整
        // ...（结果存储代码）
```

```
36      }
37      // 释放临时内存
38  }
```

由于整体流程除了变量名和少数语法之外与 neon 基本一样，故不多赘述。

此外, 因为我本地的 c++ 环境除了一些问题, 导致拷贝下来代码文件夹并在调用 FCPG 代码文件的时候出现报错, 因此对于 x86 架构下的代码测试运行速度的时候, 我并没有使用口令猜测的框架, 而是重新生成自定义大小的字符串来进行测试 (性能测试代码是 md5compare.cpp).

4.4.4 并行宽度对加速比的影响

为探究 SIMD 并行宽度对 MD5 哈希算法性能的影响，本实验实现了三种不同并行度的版本：2 路 SSE2、4 路 SSE 和 8 路 AVX2，并在 AMD Ryzen 7 7840H 处理器上进行了性能测试。其中, 2 路和 4 路的实现代码在 guess_x86 文件夹下的 md5.cpp 和 md5.h 里实现, 使用 avx 完成的 8 路实现于 md5_avx2.h 和 md5_avx2.cpp 里. 测试结果如下表所示：

表 3: 不同并行宽度对 MD5 哈希性能的影响

数据量	密码长度	2 路加速比	4 路加速比	8 路加速比
10000	16	1.05	1.66	1.95
10000	32	1.38	1.46	2.00
10000	64	1.44	1.79	2.12
10000	128	1.34	1.87	2.36
10000	256	1.31	1.90	2.55
50000	16	1.21	1.68	1.97
50000	32	1.16	1.67	1.82
50000	64	1.25	1.79	2.23
50000	128	1.23	1.79	2.27
50000	256	1.27	1.77	2.30
100000	16	1.18	1.63	1.95
100000	32	1.20	1.70	1.95
100000	64	1.20	1.67	2.02
100000	128	1.24	1.67	2.04
100000	256	1.25	1.72	2.20
250000	16	1.22	1.70	1.93
250000	32	1.21	1.63	1.82
250000	64	1.22	1.67	1.96
250000	128	1.23	1.68	2.02
250000	256	1.24	1.73	2.22
500000	16	1.20	1.64	1.85
500000	32	1.19	1.61	1.83
500000	64	1.24	1.68	1.96
500000	128	1.22	1.67	2.03
500000	256	1.24	1.71	2.19

数据分析表明，SIMD 并行宽度的增加显著提升了 MD5 哈希算法的性能，但加速比远低于理论线性增长。

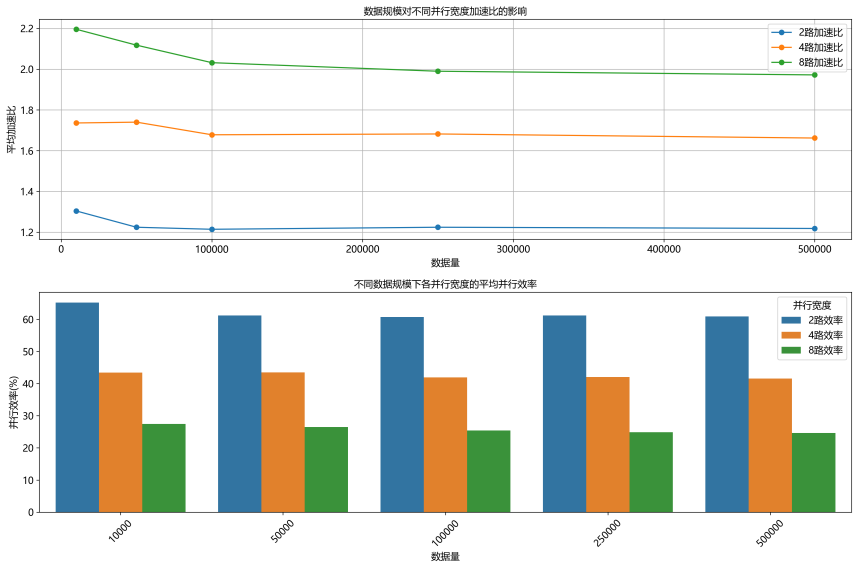

图 4.1: 加速比曲线图及平均并行效率可视化:

从可视化图表中，同样可以明显看到，加速比在并行宽度变大时，显著呈现 2 路 <4 路 <8 路的关系；与此同时，并行效率（实际加速比/理论加速比）随着并行宽度的增大，呈现了不断下降的趋势。

二路并行的特例 值得注意的是，sse 里的寄存器最小的大小即 128 位，在我的二路并行的实现中，采用的是只使用两个寄存器，另外两个寄存器全补 0 的方式。但这种方式严格来说增加了二路并行的开销，或许我认为这属于指令缺陷吧?

```
bit32 values[2] = {0}; // 初始化为0
// 只加载两个输入
for (int k = 0; k < 2; k++) {
    int msgIdx = batch + k;
    values[k] = (paddedMessages[msgIdx][4*j + i*64]) |
                (paddedMessages[msgIdx][4*j + 1 + i*64] << 8) |
                (paddedMessages[msgIdx][4*j + 2 + i*64] << 16) |
                (paddedMessages[msgIdx][4*j + 3 + i*64] << 24);
}
// 加载到SSE寄存器
x[j] = _mm_set_epi32(0,0, values[1], values[0]);
```

八路并行的特例 此外，8 路并行需使用 AVX2 而非 SSE。SSE 指令集使用 128 位 XMM 寄存器，最多支持 4 个 32 位整数的并行运算（4 路并行）。而 AVX2 引入 256 位 YMM 寄存器，可同时处理 8 个 32 位整数，适合 8 路并行。AVX2 实现（如 `md5_avx2.cpp`）通过 `__m256i` 类型和指令（如 `_mm256_and_si256`、`_mm256_add_epi32`）扩展了 SSE 的功能。例如，核心变换函数 `F_AVX2` 定义为:

```
#define F_AVX2(x, y, z) _mm256_or_si256(_mm256_and_si256((x), (y)),
    _mm256_andnot_si256((x), (z)))
```

`MD5Hash_AVX2` 函数将 8 个输入分批处理，每次加载 8 个 32 位字到 `__m256i x[16]`，并执行向量化 MD5 轮变换（如 `FF_AVX2`），最后通过 `_mm256_store_si256` 存储结果。

以上结果中，为了快速得到加速比与并行宽度之间的关系，三种并行宽度的 simd 算法均在 o2 级别的编译优化下完成，这也为我的数据采集节省了时间成本。

一些不足 由于时间原因和一些其他原因，我并没有对 sse 和 avx 版本的 md5 哈希算法进行其他的优化，而是仅仅使用 x86 指令集上的相关框架对齐完成了最基本的并行化。这之中，预处理的 StringProcess 函数依然是直接使用的串行版本，同时，也没有对其完成我在 neon 平台上做的循环展开和数据预处理等操作，如果完成以上优化工作，预计各阶段的加速比还能得到提升。只是当前的处理已经足够得到数据令我们来就并行宽度对加速比的影响作出分析。

综合分析 综合分析表明，MD5 哈希算法的 SIMD 并行优化虽然确实能提供显著加速，但受限于算法内在特性和硬件架构制约，实际加速比远低于理论值。并行宽度的选择应根据具体应用场景（特别是数据规模和密码长度）灵活调整，而非简单地追求更宽的 SIMD 指令。未来的优化可考虑将预处理步骤并行化，优化内存访问模式，或结合线程级并行和 SIMD 指令，进一步提升 MD5 哈希计算的性能。

5 结论

本次实验中，我学习并尝试使用 arm 架构下的 neon 指令集对 md5hash 算法进行了并行化，并尝试性地对算法过程进行了一些优化，在编译器编译优化的帮助下，使并行 simd 算法实现了相对于串行算法的加速。

另一方面，我在 x86 架构下使用了 sse、avx 等指令集也实现了一遍 simd 编程，这让我加深了对 simd 编程的了解，同时也对 sse、avx 指令集有了一些浅薄的认识。

同时，对并行宽度与加速比关系的探索让我意识到了在 simd 编程背景下大并行宽度的优势，尽管被打包解包开销、内存分配和循环等的开销所限制，使得随着并行宽度的增大，并行效率不断降低，但在一定程度上来说，并行宽度的增加能实现效率的改善。

本作业全部代码 (arm 上的 neon 版本,x86 上的 sse、sve 版本) 已打包上传至 git 仓库, 其中 arm 版本存在 guess 文件夹里,x86 版本存在 guess_x86 文件夹里：

https://github.com/1308176303/simd-.git