南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

口令猜测SIMD编程实验

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 4 月 26 日

摘要

关键字：优化；SIMD；并行

目录

一、 实验选题与选题的简述

本次实验中，我选择的选题是口令猜测并行化，下面是有关MD5哈希算法的概述，以及对于该选题的理解

在本实验中，MD5哈希算法在md5.cpp中通过MD5Hash函数实现。MD5Hash 函数实现了对单个输入字符串的 MD5 哈希算法，生成 128 位哈希值，存储在 4 个 bit32 元素的 state 数组中。其过程首先调用 StringProcess 填充输入字符串，确保消息长度（以位为单位）模 512 等于 448，填充方式为添加 0x80 字节、若干 0 字节及 64 位原始长度（小端序）。填充后的消息被分为多个 512 位（64 字节）块，每块拆为 16 个 32 位字（x[0..15]，小端序）。函数初始化四个 32 位状态寄存器（A=0x67452301, B=0xefcdab89, C=0x98badcfe, D=0x10325476），对每个块执行四轮（共 64 步）变换，使用 FF、GG、HH、II 函数进行非线性运算、模加和左旋转，结合预定义的移位量（s11-s44）和规定好的常数。每块处理后，状态通过累加更新。最后，状态进行字节序转换（小端转大端），生成 MD5 哈希。

本实验需要在标量实现的串行MD5Hash算法的基础上，通过利用 ARM NEON 指令集实现 SIMD（单指令多数据）优化，每次传入四个字符串进行计算，并将串行算法使用的寄存器改为向量寄存器进行单指令多数据的运算，从而实现SIMD优化，提高程序运行的执行速度。

二、 基础要求

（一） 对md5.h的修改

对于md5.h我们需要对其中的9个函数进行SIMD优化，下面我们以对F和FF函数的优化为例进行说明。

为了提高函数运行的速度，将F_SIMD函数与ROTATELEFT融入FF_SIMD函数中,从而减少函数调用的时间开销。在代码实现中，我们使用uint32x4_t指令将常量进行向量化，然后将32位常量 ac 复制到128位寄存器的四个32位通道中，生成 [ac, ac, ac, ac]，确保每个消息块使用相同的常量值，之后使用一系列NEON指令完成对于向量的加减、移位的操作。具体代码实现比较简单，在此不做赘述。

（二） 对MD5Hash函数的修改

1. 传入参数规模的改变

MD5Hash_SIMD函数定义

```
void MD5Hash_SIMD(const string *inputs, uint32_t states[][4], int
    batch_size)
```

优化后的MD5Hash_SIMD函数每次接受 batch_size 个输入字符串数组（const string *inputs）、状态数组（uint32_t states[][4]）和每次处理的批大小（int batch_size），并行处理 batch_size 个输入，其中batch_size默认为4.

2. 输入字符串处理和填充

MD5Hash_SIMD函数字符串处理

```
Byte **paddedMessages = new Byte *[batch_size];
int *messageLength = new int[batch_size];
```

```
int n_blocks[batch_size];
int max_blocks = 0;
for (int k = 0; k < batch_size; ++k){
        paddedMessages[k] = StringProcess(inputs[k], &messageLength[k]);
        n_blocks[k] = messageLength[k] / 64;
        if (n_blocks[k] > max_blocks)
        max_blocks = n_blocks[k];
}
```

为 4 个输入分别调用 StringProcess，分配 paddedMessages（指针数组）和 messageLength（长度数组），同时由于输入的4个字符串的长度不同，其块数也很有可能不相同，创建数组n_blocks来记录每个字符串所需的块数，并利用n_blocks计算最大块数 max_blocks 以支持不同长度输入。

3. 状态初始化

MD5Hash_SIMD函数状态初始化

```
uint32x4_t state_a = vdupq_n_u32(0x67452301);
uint32x4_t state_b = vdupq_n_u32(0xefcdab89);
uint32x4_t state_c = vdupq_n_u32(0x98badcfe);
uint32x4_t state_d = vdupq_n_u32(0x10325476);
```

使用 NEON 的 uint32x4_t 向量寄存器初始化 4 个状态（state_a, state_b, state_c, state_d），从标量 state[4] 变为向量state_a, state_b, state_c, state_d，每个寄存器存储 4 个输入的对应状态，使用vdupq_n_u32 将向量寄存器初始化为对应值。

4. 处理无效块

MD5Hash_SIMD函数处理无效块

```
alignas(16) static uint32_t mask_pool[1024 * 4];
uint32_t *mask_array = mask_pool;
for (int i = 0; i < max_blocks; ++i){
        for (int k = 0; k < batch_size; ++k){
                mask_array[i * batch_size + k] = (i < messageLength[k] /
                    64) ? 0xFFFFFFFF : 0;
        }
}
```

因为在一次处理多个字符串可能出现n_blocks不一致的情况，这时我们需要对于块进行判断，检查其是否为无效块，若为无效块则进行标记。生成数组 mask_pool2（16 KB），标记每个输入的块有效性（0xFFFFFFFF 表示有效，0 表示无效），以支持不同长度输入。

5. 消息块处理

MD5Hash_SIMD函数处理无效块

```
uint32_t *messages[4];
for (int k = 0; k < 4; k++){
        messages[k] = new uint32_t[n_blocks[k] * 16];
```

```
        for (int i = 0; i < n_blocks[k]; i++)
                for (int j = 0; j < 16; j++){
                        uint32_t word = 0;
                        for (int m = 0; m < 4; m++)
                                word |= (uint32_t)paddedMessages[k][i * 64
                                    + j * 4 + m] << (8 * m);
                        messages[k][i * 16 + j] = word;
                }
}
for (int i = 0; i < max_blocks; i++)
        for (int j = 0; j < 16; j++)
                for (int k = 0; k < 4; k++)
                        if (i < n_blocks[k])
                                interleaved[16 * 4 * i + 4 * j + k] =
                                    messages[k][i * 16 + j];
                        else
                                interleaved[16 * 4 * i + 4 * j + k] = 0;
```

由于轮函数（如FF_SIMD）需要同时加载4个输入的同一字（如 x[j] 含 4 个输入的第j个字），而现在四个消息的同一块中同一字的位置没有连续存储，那么如果在不连续的状态执行 SIMD 优化后的轮函数必然会导致错误，所以我们需要将四个输入消息的填充数据转换为适用于 SIMD 并行处理的交错存储格式。

在代码的实现中，先将 paddedMessages[k] 的字节数据转换为 32 位字，存储到messages[k]，为每个输入生成 [block0_word0, ..., block0_word15, block1_word0, ...] 的格式。目的是将字节流组织为 MD5 算法所需的 32 位字，方便后续处理。然后将 4 个输入的对应字交错存储到 interleaved，形成 [block_i_word_j_input0, block_i_word_j_input1, ...] 的能够进行SIMD的形式。

6. 消息块加载和轮函数初始化

MD5Hash_SIMD消息块加载和轮函数初始化

```
uint32x4_t x[16];
uint32_t *block_start = &interleaved[16 * batch_size * i];
for (int j = 0; j < 16; ++j)
        x[j] = vld1q_u32(block_start + 4 * j);
uint32x4_t a = state_a, b = state_b, c = state_c, d = state_d;
```

从 interleaved 加载 16 个 uint32x4_t 向量 x[0..15]，每个向量含 4 个输入的对应字。初始化轮函数向量 a, b, c, d。

7. 轮函数处理

MD5Hash轮函数处理

```
uint32x4_t mask = vld1q_u32(&mask_array[i * batch_size]);
FF_SIMD(a, b, c, d, x[0], 7, 0xd76aa478);
FF_SIMD(d, a, b, c, x[1], 12, 0xe8c7b756);//共步……64
```

加载数组 mask，用于标记哪些消息需要被处理，执行 4 轮（64 步）SIMD 轮函数（FF_SIMD, GG_SIMD, HH_SIMD, II_SIMD）。这里的几个轮函数在md5.h中定义，均使用SIMD进行优化。

8. 状态更新

MD5Hash_SIMD状态更新

```
state_a = vaddq_u32(state_a, vbslq_u32(mask, a, vdupq_n_u32(0)));
state_b = vaddq_u32(state_b, vbslq_u32(mask, b, vdupq_n_u32(0)));
state_c = vaddq_u32(state_c, vbslq_u32(mask, c, vdupq_n_u32(0)));
state_d = vaddq_u32(state_d, vbslq_u32(mask, d, vdupq_n_u32(0)));
```

使用 vaddq_u32 并行累加 4 个输入的轮函数结果到 state_a 等，通过 vbslq_u32(mask, a, 0) 等指令，使用数组mask中的信息判断块是否有效，选择性更新有效消息的状态，避免无效数据干扰。

9. 字节序调整

MD5Hash_SIMD字节序调整

```
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)));
```

首先使用vreinterpretq_u8_u32(state_a)等操作将 uint32x4_t 向量重新转换为 uint8x16_t 类型，以便按字节操作，然后使用vrev32q_u8 对每个 32位块（4字节） 内的字节顺序进行反转，最低字节到最高位，最高字节到最低位，最后，再使用vreinterpretq_u32_u8 将结果重新解释为 uint32x4_t 类型。

10. 结果存储

MD5Hash_SIMD结果存储

```
alignas(16) uint32_t temp[16];
vst1q_u32(temp, state_a);
vst1q_u32(temp + 4, state_b);
vst1q_u32(temp + 8, state_c);
vst1q_u32(temp + 12, state_d);
for (int k = 0; k < batch_size; ++k){
        states[k][0] = temp[k];
        states[k][1] = temp[k + 4];
        states[k][2] = temp[k + 8];
        states[k][3] = temp[k + 12];
}
```

将SIMD向量寄存器中的多个批次的哈希结果重组并存储到目标数组中 ，把交错存储的结果保存到temp数组中，再利用temp数组将结果按原来的顺序存储到states中，最后进行内存清理操作后，完成MD5哈希算法的SIMD实现。

（三） 对main.cpp的修改

在实现MD5Hash_SIMD的算法之后，接下来就是要解决如何调用它实现最终的SIMD优化后程序的执行。首先，我们定义BATCH_SIZE作为每次处理的批大小，在这里我们设置为4，然

后，我们定义aligned_size确定所需进行MD5哈希算法的字符串数量，利用aligned_size计算能完整进行SIMD运算的字符串的最大数量（在不小于aligned_size的4的倍数中的最大值），对于这部分字符串我们调用MD5Hash_SIMD进行运算，对于除4后余下的部分进行不使用SIMD优化的原始操作。代码实现同样比较简单，不赘述。

（四） 验证修改后代码的正确性

在完成上述操作后，我们验证一下SIMD优化后代码的正确性。

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SIMD优化后的代码正确!

三、 进阶要求

（一） 实现相对串行算法的加速

在基础要求中，我们只是实现了SIMD算法的编写，但是在实际执行程序时，使用SIMD优化后的算法运行时间相比于原算法运行时间反而更长，这可能是因为在SIMD算法中的内存开销更大，预处理与数组之间进行的操作更多（额外实现交替存储、临时数组重组结果等），但是，我们仍然可以对于SIMD优化后的算法进行优化，下面是优化的实现以及优化的结果。

1. 交错存储模块优化

在上述实现交错存储的模块中，我们先创建了 messages 数组对字符串进行了预处理，但是我们也可以跳过 messages 数组，从 paddedMessages 直接生成interleaved数组，实现方法如下：通过interleaved[16 * batch_size * i + 4 * j + k] 直接定位目标地址，无需中间数组messages。

2. 循环展开

在上述代码中多处都使用到了循环操作，那么对于这些循环我们可以使用循环展开来减少循环的次数加速程序的执行。

3. 减少非必要数组的定义

我们可以取消某些中间数组如messages、n_blocks等，使用其他方式来替代这些数组起到的作用。

messages数组的例子上面说明过，我们以n_blocks数组为例进行解释。在上面的代码中，n_blocks数组用于存储每个字符串的块数，然后利用这个数组找出最大的块，但是，我们可以定义一个整数n_blocks_k来表示每个字符串所对应的块数，将当前最大的块数max_blocks与之相比较，若n_blocks_k大于max_blocks则更新max_blocks，从而实现最大块数的寻找。

表 1: 优化后的SIMD算法与未优化的SIMD算法、原算法对比

	未使用O1、O2优化	使用O1优化	使用O2优化
原算法	9.40692s	3.09324s	3.01565s
未优化的SIMD算法	19.7488s	3.39635s	3.33243s
优化后的SIMD算法	16.4269s	1.95606s	1.81424s

表中结果均采用对Hash time多次测试取平均值的方法

表 2: 优化后的SIMD算法与未优化的SIMD算法对比原算法的加速比

	未使用O1、O2优化	使用O1优化	使用O2优化
未优化的SIMD算法	0.48	0.91	0.90
优化后的SIMD算法	0.57	1.58	1.66

加速比的计算按照老师上课将的使用串行算法的时间除以并行算法的时间得到

4. 结果分析

通过以上结果我们可以发现，未优化的SIMD算法不管是否使用O1、O2优化，其运行速度都比原算法的运行速度慢，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。

另外，未使用 O1、O2 优化时，优化后的 SIMD 算法虽进行了代码层面的改进（如交错存储、循环展开等），但 SIMD 操作本身存在固有额外开销。此时，编译器未对这些 SIMD 相关操作进行深度优化，使得这些额外开销无法被有效抵消，甚至超过了 SIMD 并行计算带来的潜在收益。而原算法逻辑简单直接，无需处理这些复杂的 SIMD 额外操作，因此在未优化时，优化后的 SIMD 算法运行速度更慢。但是使用 O1、O2 优化后，编译器会对代码进行全面优化，如减少冗余指令、优化寄存器分配、改进指令流水线利用、增强向量化程度等。这些优化措施能有效降低 SIMD 操作的额外开销，使 SIMD 的并行计算优势得以充分发挥。此时，SIMD 算法可同时利用代码层面的优化（如交错存储提升数据访问效率、循环展开减少循环控制开销）和编译优化（如更高效的指令生成），并行处理大量数据，大幅提升执行效率，最终使得运行速度超过原算法。

（二） 控制“单指令多数据”中数据的总数

在实现Batch_Size批处理数量位4的SIMD后，我们尝试更改Batch_Size的值，将其改为2、8，改变单次运算的并行度，并探索其加速效果会发生怎样的改变。

1. 代码修改

实现Batch_Size等于2需要对主函数调用MD5Hash_SIMD中的BATCH_SIZE的值进行修改和对MD5Hash_SIMD中的各个数组的大小分配和循环参数进行修改，但是代码逻辑基本没有变化。实现Batch_Size等于8可以将传入的数组使用两组uint32x4_t状态向量，分别处理输入 0-3 和 4-7，每组向量处理四个输入。同样，对于mask数组，x数组也可以扩充为两组进行处理，这里的扩冲为两组并不是分批次进行前4个字符串与后四个字符串分别进行处理，而是因为与NEON指令集的数据处理规模保持一致，所以需要将8个字符串分为两组，分为两组之后仍然按照Batch_Size等于4的SIMD算法的思路进行实现，只不过在处理数据时需要多增加循环次数为2的循环来处理这两组数据。

将更改完成后的代码使用correctness.cpp进行验证，同样能得到正确的结果。

MD5Hash_2NEON 通过减少状态向量、缩小交错步幅和简化掩码生成，适配小批量场景，降低资源占用，但并行度有限。MD5Hash_8NEON 通过引入双状态向量、增大步幅、完全展开掩码生成和调整块数计算，适配大批量场景，修复正确性问题并提升性能，但内存占用和复杂性增加。

2. 结果分析

表 3: 优化后的SIMD算法与未优化的SIMD算法、原算法对比

	未使用O1、O2优化	使用O1优化	使用O2优化
原算法	9.40692s	3.09324s	3.01565s
Batch_Size=2	29.5636s	3.08359s	2.92355s
Batch_Size=4	16.4269s	1.95606s	1.81424s
Batch_Size=8	16.4804s	1.88265s	1.78751s

通过观察上述结果我们可以发现，对于不同 $Batch\_Size$ 的情况，未优化时 $Batch\_Size = 2$ 执行时间高达 $29.5636s$，远超原算法，$Batch\_Size = 4$ 和 $Batch\_Size = 8$ 未优化时执行时间为 16 秒多，相对 $Batch\_Size = 2$ 更好，但仍高于串行算法。而使用 O1、O2 优化后，各 $Batch\_Size$ 的执行时间大幅下降，尤其是 O2 优化， $Batch\_Size = 8$ 时仅为 $1.78751s$。SIMD 算法在O1、O2优化后，随着 $Batch\_Size$ 增大，其并行处理能力得以充分发挥，有效抵消了原本的额外开销，使得并行优势凸显，执行时间大幅缩短，并且 $Batch_Size$越大，代表程序的并行化程度更高，所以执行效率也会提升，但是也会使用更多的内存开销。

（三） 使用x86指令集在x86环境中实现SIMD并进行profiling

使用SSE指令集实现SIMD的思路不做过多介绍，只是将NEON指令集换为对应的SSE指令集，下面我们使用VTune对串行与并行的两个SSE程序进行分析。

表 4: SSE指令集优化后的SIMD算法与串行算法对比

	Clockticks	Instructions Retired	CPI Rate
串行算法	3683000000	1075900000	3.423
SIMD算法	2871000000	980200000	2.929

使用SIMD优化后的MD5哈希算法在Clockticks（时钟周期数）、Instructions Retired（执行指令数）、CPI Rate（每指令周期数），下面让我们对这一现象进行分析。首先，SIMD通过减少循环次数和分支开销（如循环计数器更新和条件判断），进一步降低了Clockticks，也就是缩短了运行时间。其次，SIMD通过利用128位或更宽的向量寄存器（如SSE中的__m128i或NEON的uint32x4_t），在单条指令中同时处理多个数据元素。在MD5算法中，轮函数（FF、GG、HH、II）涉及大量32位整数的加法、位运算和循环移位操作，SIMD可以将这些操作并行应用于4个输入的相同步骤，从而减少指令数量。SIMD优化的算法在单个周期内完成更多工作，所以其CPI Rate也会降低。

（四） 尝试不使用NEON指令集实现MD5算法

在了解SIMD算法的实现方法之后，我尝试不使用NEON指令集实现MD5算法，NEON指令集中直接使用NEON向量类型（uint32x4_t）实现标量的向量化处理，而如果我想要不使用NEON指令集，我需要自定义 MD5Vector 类（封装4个标量值）模拟向量化。

其实现思路大致相同，除定义一个结构体来表示向量之外，还需要将NEON指令集包含的一些指令编写成函数进行处理，逻辑思路大致与使用NEON指令集来实现相同，在此不赘述，最终实现了不使用NEON指令集的SIMD算法，具体实现代码在github中。

四、 实验总结与思考

本次实验通过ARM NEON指令集与SSE指令集实现了MD5哈希算法的SIMD优化，成功将串行算法改造为支持批量处理的并行版本。并实现了实现相对串行算法的加速、控制“单指令多数据”中数据的总数、尝试不使用NEON指令集实现MD5算法等进阶要求。

1. SIMD编程实践

深入理解了NEON指令集与SSE指令集的向量化操作，掌握了通过数据交错存储、掩码控制实现并行计算的完整流程。并且对于不使用NEON指令集将标量向量化实现SIMD优化也有了一定的了解。

2. 性能优化方法

在实现SIMD算法并不能加速串行时间后，我通过交错存储模块的优化、循环展开、减少非必要数组的定义等优化方法实现了SIMD算法相对于串行算法的加速，在探索实现SIMD算法加速的过程中，我对理解MD5算法与从现有算法找出优化的部分并对其进行优化有了更深刻的认识。

3. 并行化权衡

在实现进阶要求中的控制“单指令多数据”中数据的总数实现中，我体会到批量规模（batch_size）对吞吐量和内存开销的影响，在实际应用中可能需要综合考虑运行速度与内存开销等多种因素进行算法的选择，这使我对于并行算法的分析与认识进一步提高。