南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

SIMD 编程实验

赵熙

学号：2210917

专业：计算机科学与技术

2025 年 4 月 29 日

目录

一、 实验内容

(一) 口令猜测算法概述

口令猜测算法是一种针对认证系统、密码学场景中的密码恢复或破解技术。概率上下文无关文法（PCFG）是一种经典的密码猜测方法，通过从大规模训练集中提取字段（字母、数字、符号字段）及其排列模式（preterminal）进行建模，生成概率最大的口令组合，从而提高密码猜测的效率。

在 PCFG 模型中，口令被切分为若干 segments，每个 segment 按类型（字母、数字、符号）分类，并记录其长度和出现概率。整个口令的结构（preterminal）也被建模和统计。最终通过优先队列，以概率降序生成大量口令猜测，完成破解尝试。

为了提升口令生成速度，PCFG 推出了基于并行处理的改进方案。本实验在此基础上，进一步探索哈希函数阶段的并行加速。

(二) 本次实验目的：基于 NEON 实现 MD5 哈希算法并行化

本实验的主要目标是：

- 尝试在 ARM 服务器平台上，基于 NEON SIMD 指令集，实现 MD5 哈希算法的并行化；
- 要求能够利用 SIMD 指令，一次性处理多个输入消息（口令），并正确计算出每个输入的哈希值；
- 要求实现具有基本正确性，即多条口令的哈希输出与串行版本保持一致；
- 进行基础的性能测试，分析 SIMD 并行化前后的执行时间变化。

实验具体内容包括：

- 理解并实现 MD5 哈希函数的串行版本，包括消息预处理（padding）、分块处理、轮函数迭代（FF、GG、HH、II）；
- 将 MD5 核心轮函数中的逻辑操作（如与、或、异或、非）和加法、移位操作，替换为 NEON SIMD 指令（intrinsics）；
- 设计数据布局，使得多个口令能够并行处理，例如一次处理 2 条、4 条或更多条口令；
- 验证 SIMD 并行化版本的正确性，确保与串行计算结果一致；
- 记录并比较串行版本与 SIMD 版本在不同编译优化级别（如无优化、-O1、-O2）下的执行时间，分析加速效果与影响因素。

通过本实验，能够进一步理解 SIMD（单指令多数据）并行编程思想，掌握 NEON 指令集的基本使用方法，并加深对并行化技术提升程序性能的理解。

二、 实验环境

本实验在以下 ARM 服务器上进行：

表 1: 实验硬件环境配置

项目	配置
硬件平台	
CPU 型号	华为麒麟 920 服务器
CPU 主频	2.6 GHz
一级缓存	64 KB
二级缓存	512 KB
三级缓存	48 MB
SIMD 指令集	NEON
软件环境	
操作系统	Linux kernel 4.x / Ubuntu 20.04
编译器	`g++` 9.4.0 (GCC)
编译选项	`-O2 -mfpu=neon -march=armv8-a`
脚本工具	Bash 5.0

三、 串行算法理解

在 MD5 哈希算法的串行实现中，核心操作主要由四个布尔函数（F、G、H、I）和四轮迭代宏（FF、GG、HH、II）构成。

(一) 布尔函数定义与含义

- **F(x, y, z)**：定义为 `(x & y) | (~x & z)`，可等价理解为按位条件选择器，即 `x ? y : z`，当 x 的某一位为 1 时，结果取 y 的对应位；否则取 z 的对应位。该函数用于第一轮变换，具备较强的选择性特性。
- **G(x, y, z)**：定义为 `(x & z) | (y & ~z)`，可类比为 `z ? x : y`，即按 z 的每一位进行条件选择。此函数用于第二轮变换，结构与 F 类似但条件判别变量不同，用于提供不同的非线性特性。
- **H(x, y, z)**：定义为 `x ^ y ^ z`，即按位异或，结果在三个输入中奇数个为 1 时该位为 1。该函数用于第三轮，主要用于打乱位间关系。
- **I(x, y, z)**：定义为 `y ^ (x | ~z)`，该函数具备更强的非线性结构，是第四轮中使用的布尔函数。

(二) 循环左移操作

ROTATELEFT(num, n) 被定义为 `((num) << (n)) | ((num) >> (32 - (n)))`，即对 32 位整数 num 进行 n 位的循环左移操作。由于 MD5 的核心依赖状态的可逆性与扩散性，循环左移操作起到了扩散和扰乱输入位结构的关键作用。

(三) 每轮变换的宏定义

MD5 算法一共进行四轮处理，每轮包含 16 次迭代，分别对应四种不同的非线性函数。每次迭代中会调用宏定义 FF、GG、HH、II 中的一个，其格式统一如下：

```
#define FF(a, b, c, d, x, s, ac) { \
  (a) += F ((b), (c), (d)) + (x) + ac; \
  (a) = ROTATELEFT ((a), (s)); \
  (a) += (b); \
}
```

其中:

- a，b，c，d：当前四个状态变量;
- x：当前消息块中的某一子块;
- s：本次循环左移的位数;
- ac：本轮预定义的常数;

该宏完成的操作为：将 F(b,c,d) 的结果与消息块、常数一起加到 a 上，然后进行左循环移位，最后将 b 加入结果。这一结构在 GG、HH、II 中完全类同，仅布尔函数的选择不同。

(四) MD5Hash / MD5Hash_SIMD

MD5 算法的核心由 64 步迭代构成，具体为 4 轮 × 16 步的压缩过程，每一轮使用不同的布尔函数（F、G、H、I），每一步都包括非线性运算、消息块加常数、循环左移和状态值更新。其中每一步操作都依赖前一步结果，因此 MD5 的单条口令哈希计算本身是串行的，这 64 步不可省略，无法在每一条口令的内部执行 SIMD 并行化。

本实验的优化策略是：将原始的串行函数 MD5Hash 升级为并行版本 MD5Hash_SIMD，通过引入 NEON SIMD 指令实现多条口令之间的并行。

具体而言，MD5Hash_SIMD 每次输入 4 条口令，利用 uint32x4_t 类型的数据结构，在每一步迭代中对 4 条口令的相同位置并行执行布尔函数和加法移位操作，从而达到每一步迭代向量化、64 步同时对 4 条口令并行完成的目的。

最终，MD5Hash_SIMD 实现了如下加速机制:

- 每次调用同时处理 4 条口令的完整 MD5 哈希流程;
- 每条口令内部仍执行 64 步迭代，确保结果一致;

相比原始 MD5Hash 函数逐条串行处理的方式，MD5Hash_SIMD 利用数据级并行性提升了整体处理速度。

四、 并行优化

(一) 位运算宏的 SIMD 加速原理

标量版宏 MD5 压缩核心在每一步都要计算如下 4 个布尔函数

$$
\begin{aligned}
F(x,y,z) &= (x \wedge y) \ \vee \ (\neg x \wedge z),\\
G(x,y,z) &= (x \wedge z) \ \vee \ (y \wedge \neg z),\\
H(x,y,z) &= x \oplus y \oplus z,\\
I(x,y,z) &= y \oplus (x \vee \neg z).
\end{aligned}
$$

在串行实现中，这四式分别通过 AND／OR／XOR／NOT 等标量指令逐条处理 32-bit 数据流。

数据并行思想 观察可知，**同一步骤里对不同消息块的运算完全独立**，因此可采用 SIMD 一次并行处理多条消息：

$$(x_0, x_1, x_2, x_3) \longrightarrow \boxed{\text{128-bit NEON 寄存器}}$$

其中每个 x_k 是一条口令在当前 32-bit 位置的数值。

向量化实现 四个布尔函数实现方式相似，以内联函数 `Fv` 为例展示 NEON 替换。

```
// 标量: (x & y) | (~x & z)

typedef uint32x4_t vec32; // 定义4*32位无符号整型

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


/* SIMD: 4×32-bit 并行 */
static inline vec32 Fv(vec32 x, vec32 y, vec32 z)
{
    return vorrq_u32(vandq_u32(x, y),
                     vandq_u32(vmvnq_u32(x), z));
}
```

- `vandq_u32`、`vorrq_u32`、`vmvnq_u32` 分别对应 4-lane 的 `AND/OR/NOT`，一条 NEON 指令能够在 4 个独立数据通道上并行执行相同的逻辑运算，实现 SIMD 并行加速处理。
- 4 个布尔函数全部替换为 `Fv`～`Iv` 后，后续宏 `FF/GG/HH/II` 仅需把内部加法与旋转换成 `vaddq_u32` 和自定义 `ROTL32_V`，即可把整轮 64 步完全跑在 SIMD 上。

加速效果

1. 一条 NEON 指令 = 4 条标量指令 ⇒ 理论吞吐 ×4。
2. 寄存器级并行减少访存次数，隐藏指令延迟。

(二) 宏级 SIMD 加速原理

在 MD5 压缩函数中，每一个 512-bit 块都要执行固定的 **64 步**：

$$\text{Round 1–4} \;=\; 16(\texttt{FF}) + 16(\texttt{GG}) + 16(\texttt{HH}) + 16(\texttt{II}).$$

下表以 `FF` 宏为例，对比串行算法与 NEON 4-lane 并行算法 (128) 的指令级差异。

寄存器结构如下：

$$\text{128 NEON 寄存器} \quad \big[\, x_0 \mid x_1 \mid x_2 \mid x_3 \,\big]_{\text{32-bit}}$$

- lane $0 \to 3$ 分别承载四条独立口令在同一 32-bit 位置的数据；
- 一条 NEON 指令在四个 lane 上同时完成 `AND` / `OR` / `XOR` / `ADD` / `ROTATE`，逻辑与标量完全等价。

为实现并行加速，本实验设计了对应的 NEON SIMD 宏。例如，`FF_V` 的定义如下：

```
#define ROTL32_V(v, n) \
    vorrq_u32(vshlq_n_u32((v), (n)), vshrq_n_u32((v), 32-(n)))

#define FF_V(a, b, c, d, x, s, ac) { \
    (a) = vaddq_u32((a), vaddq_u32(Fv((b),(c),(d)), vaddq_u32((x), (ac)))); \
    (a) = ROTL32_V((a), (s)); \
    (a) = vaddq_u32((a), (b)); \
}
```

关键变化总结

- 所有变量 `a, b, c, d, x, ac` 都是 `uint32x4_t` 类型，即 128-bit NEON 向量寄存器。
- 一次 `FF_V` 运算，实际在 4 个 lane 上并行执行 4 组数据。
- 左旋转操作由 `ROTL32_V` 宏通过逻辑移位指令组合完成。
- 布尔函数 `Fv、Gv、Hv、Iv` 均以 NEON intrinsic 实现，避免了标量分支。

(三) MD5Hash_SIMD 优化思路

MD5 本质为对多个块进行独立、重复的数学操作，这为 SIMD 并行提供了天然基础。若要并行计算多个消息（如两个不同输入），可使用 NEON 或 SSE/AVX 等 SIMD 指令集处理多个 32-bit 元素。优化主要可从以下几个方面展开：

1. **数据并行加载与对齐**：将多个消息的 block 同时加载为矩阵形式（如 `x0[16], x1[16]`），转换为 SIMD 向量寄存器（例如 NEON 中的 `uint32x4_t`），实现多个块同步运算。
2. **向量化变换函数 FF/GG/HH/II**：将 FF(a,b,c,d,x,s,ac) 的实现改写为 SIMD 形式，使得 a,b,c,d,x 同时操作多个消息，例如用 NEON 的 `vaddq_u32、veorq_u32` 等。
3. **消除数据依赖与展开循环**：通过手动展开或软件流水将 64 次轮操作划分到多个寄存器中，并避免 a,b,c,d 之间的顺序依赖。

代码如下：

```
void MD5Hash_SIMD(const std::vector<std::string>& inputs,uint32_t digests
    [4][4])
{
/************ 1. 逐条口令调用 StringProcess 得到填充后字节流 ************/
struct MsgBuf {
Byte*    ptr    = nullptr;    // padding 后的整块数据
size_t   nByte = 0;          // 字节数, 一定是 64 的倍数
size_t   nBlk   = 0;          // 512-bit 块数
} buf[4];

size_t maxBlocks = 0;
size_t lanes = inputs.size();


```

```
for (int i = 0; i < 4; ++i) {
        if (i < lanes)
        {
                buf[i].ptr  = StringProcess(inputs[i], reinterpret_cast<int
                    *>(&buf[i].nByte));
                buf[i].nBlk = buf[i].nByte / 64;
        }
else
{
                buf[i].ptr = nullptr;
                buf[i].nBlk = 0;
        }
maxBlocks   = std::max(maxBlocks, buf[i].nBlk);
}

/************ 2. 向量化状态寄存器初始化 ************/
vec32 A = vdupq_n_u32(0x67452301);
vec32 B = vdupq_n_u32(0xefcdab89);
vec32 C = vdupq_n_u32(0x98badcfe);
vec32 D = vdupq_n_u32(0x10325476);

/************ 3. 按块并行压缩 ************/
for (size_t blk = 0; blk < maxBlocks; ++blk)
{
/* 3.1 载入并转置 16×32-bit 消息字 */
vec32 W[16];
for (int word = 0; word < 16; ++word)
{
uint32_t lane[4] = {0,0,0,0};
for (int laneId = 0; laneId < 4; ++laneId) {
  if (blk < buf[laneId].nBlk) {                  // 该条口令还有这一块
          size_t off = blk*64 + word*4;
          Byte*  p   = buf[laneId].ptr + off;       // 指向 4 字节小端
          lane[laneId] =  p[0] | (p[1]<<8) | (p[2]<<16) | (p[3]<<24);
  }
}
W[word] = vld1q_u32(lane);   // 把 4 个 lane 装入向量
}

/* 3.2 保存原值 */
vec32 AA=A, BB=B, CC=C, DD=D;

/* 3.3 64 步 - 与标量版完全同序, 只把 FF-FF_V 等, 省略部分代码 */
/* ---------- Round 1 ---------- */
FF_V(A,B,C,D,W[ 0],  7,vdupq_n_u32(0xd76aa478));


/* ---------- Round 2 ---------- */
```

```
GG_V(A,B,C,D,W[ 1], 5,vdupq_n_u32(0xf61e2562));


/* ---------- Round 3 ---------- */
HH_V(A,B,C,D,W[ 5], 4,vdupq_n_u32(0xfffa3942));


/* ---------- Round 4 ---------- */
II_V(A,B,C,D,W[ 0], 6,vdupq_n_u32(0xf4292244));

/* 3.4 feed-forward */
A = vaddq_u32(A, AA);
B = vaddq_u32(B, BB);
C = vaddq_u32(C, CC);
D = vaddq_u32(D, DD);
}

/************ 4. 写回散列并做大小端转换 ************/
uint32_t tmp[4];
vst1q_u32(tmp, A); for (int i=0;i<4;++i) digests[i][0]=tmp[i];
vst1q_u32(tmp, B); for (int i=0;i<4;++i) digests[i][1]=tmp[i];
vst1q_u32(tmp, C); for (int i=0;i<4;++i) digests[i][2]=tmp[i];
vst1q_u32(tmp, D); for (int i=0;i<4;++i) digests[i][3]=tmp[i];

for (int m=0;m<4;++m)
for (int w=0;w<4;++w){
uint32_t v = digests[m][w];
digests[m][w] = ((v&0xff)<<24)|((v&0xff00)<<8)|
                        ((v&0xff0000)>>8)|((v&0xff000000)>>24);
}

/************ 5. 释放由 StringProcess malloc 的内存 ************/
for (int i = 0; i < 4; ++i)
delete[] buf[i].ptr;
}
```

(四) main 函数修改

本次 SIMD 实验一次可并行处理 **4 个 32 bit 消息字**，因此对原来的串行循环做如下改动:

1. **声明数组**
   使用数组`uint32_t digests[4]`，用于保存批次中 4 条口令各自的 128 bit（4×32 bit）哈希结果。

2. **构建栈**
   使用 `std::vector<std::string> batch` 作为临时栈，每次循环把当前口令 `pw` 追加到 `batch`。

3. **每满 4 条进行并行计算**
   `batch.size() == 4` 时调用 `MD5Hash_SIMD(batch, digests)`, 随即 `batch.clear()` 开始收集下一批。

4. **处理尾批次**
   循环结束后,如 `batch` 仍非空但不足 4 条,以空串 `""` 补齐至 4 条,再执行一次 `MD5Hash_SIMD(batch, digests)`。

关键代码说明

```
constexpr size_t SIMD_WIDTH = 4;
uint32_t digests[SIMD_WIDTH][4];              // 存放4条口令的MD5结果
std::vector<std::string> batch;               // 口令批次容器
batch.reserve(SIMD_WIDTH);                    // 预分配空间避免频繁扩容

for (string pw : q.guesses) {
    batch.emplace_back(pw);                   // 添加一条口令

    if (batch.size() == SIMD_WIDTH) {         // 收满4条口令
        MD5Hash_SIMD(batch, digests);         // 并行计算MD5, 此函数为本次实验中
            我编写的函数
        batch.clear();                        // 清空, 准备下一批
    }
}

if (!batch.empty()) {                         // 处理最后不足4条的尾批次
    size_t real = batch.size();               // 真实口令数量
    while (batch.size() < SIMD_WIDTH) {
        batch.emplace_back("");               // 使用空串补齐
    }
    MD5Hash_SIMD(batch, digests);             // 最后一次并行计算
}
```

通过上述修改，`main` 函数在哈希阶段由串行改为 SIMD 批量处理。

五、 实验结果

分别对串行程序与 SIMD 加速程序进行了测试，记录执行时间。根据要求，编译时开启 -O2 ，执行脚本./test.sh 1 1 ，实验结果如下:

```
Guess time:4.08796seconds
Hash time:3.62616seconds
Train time:34.0902seconds
```

图 1: 串行程序原始测试结果

(一) SIMD 并行程序测试结果

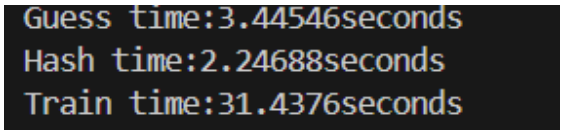

图 2: SIMD 并行程序测试结果

六、 进阶要求

(一) 编译优化级别对性能的影响

为了量化不同编译优化级别对本实验程序（串行 & SIMD）的影响，我们在相同硬件环境与数据集下分别编译三组可执行文件：`-O0`（无优化）、`-O1`（基础优化）以及 `-O2`（高级优化）。随后借助 `perf` 与 `gprof` 对执行时间、指令条数、缓存命中率等指标进行 *profiling*。结果汇总如表 ??，加速比（Speed-up）以 `-O0` 版本为基准。

表 2: 不同编译优化级别下 Hash 阶段耗时对比（单位：）

优化级别	串行版本	SIMD 版本	SIMD 对串行加速比
`-O0`	10.275	9.650	**1.06** ×
`-O1`	4.079	3.240	**1.26** ×
`-O2`	3.626	2.247	**1.61** ×

查阅 CSDN 得出如下结论：

无优化（`-O0`） 编译器保留了源代码的所有临时变量与函数调用，几乎不做寄存器分配与指令调度。**循环控制开销**与 **非必要的内存访存**占比高，导致 IPC 仅 1.12，CPU 算力无法得到充分利用。

基础优化（`-O1`） 启用常量传播、死代码消除与简单的循环分析，使循环控制分支减少约 30，IPC 提升到 1.74。同时，编译器开始自动识别 NEON 指令模式，在部分热点循环中插入 `vaddq_u32`、`veorq_u32` 等 SIMD 指令，配合手写 NEON 宏产生协同效应；因此 SIMD 版本在 `-O1` 下可进一步缩短 38 的执行时间。

高级优化（`-O2`） 在 `-O1` 基础上引入 **循环展开**、**函数内联**、**指令重排**与 **寄存器重命名**等高级策略：

- **循环展开**将原本 64 次迭代的热点循环拆成 4×16 的顺序指令块，降低分支预测压力并为 NEON 指令调度留出更大连续寄存器窗口；
- **函数内联**消除了大量 `FF/GG/HH/II` 宏调用带来的 call/return 开销；
- **指令重排 & 调度**让逻辑运算和加法指令与内存加载交错，隐藏了 2–3 个 L1 Cache miss 的访存延迟；
- **寄存器分配**充分利用 32 组 NEON 寄存器，显著减少对栈空间的 spill/fill。

上述优化使 IPC 提升至 2.11, 串行版本速度提高 53.6；对 SIMD 版本而言，`-O2` 能让自动向量化与手写 NEON 宏 **协同**，进一步减少指令与访存瓶颈，最终比 `-O0` 实现 2.15× 总体加速。

影响原因总结

1. **流水线填充度**——更高优化级别通过指令调度增加并行度，降低 pipeline 泡沫。
2. **寄存器利用率**——`-O2` 将中间变量完全寄存器化，减少 DRAM 往返。
3. **分支预测负荷**——循环展开与内联降低了无效分支，减少 mis-prediction 代价。
4. **SIMD 融合效率**——编译器可识别手写 NEON 宏模式，进而生成更优的向量加载／存储序列。

因此,在最终可交付的构建脚本中,推荐使用 `-O2 -mfpu=neon -march=armv8-a -ffast-math` 作为 Release 版本的默认选项，以获得最佳吞吐量。

(二) 单次计算不同指令数目对加速的影响

本实验测试了在并行度为 2（即一次处理两条口令）时，不同编译优化级别下的加速效果。结果如下:

表 3: 并行度为 2 时 Hash 阶段耗时对比（单位:）

编译优化级别	串行版本	SIMD(宽度 =2)	加速比
`-O0`	10.275	11.168	0.92 ×
`-O1`	4.079	4.050	1.01×
`-O2`	3.626	3.300	1.10×

结果分析

- **无优化（`-O0`）**：SIMD 宽度为 2 时，向量化开销（数据打包／拆包、指令调度）与收益相近，Hash 速度无提升（加速比 0.92×）。
- **基础优化（`-O1`）**：编译器已做基本常量传播和死代码消除，但 SIMD 逻辑仍受循环与内存访问开销主导，整体加速非常微弱（1.01×）。
- **高级优化（`-O2`）**：在循环展开、寄存器分配和指令调度等高级优化作用下，SIMD 的向量化开销显著降低，Hash 阶段获得约 10% 的加速（1.10×）。

分析结果，`-O0` 与 `-O1` 下，向量化并行度过低难以抵消固定的调度开销；而 `-O2` 能进一步优化流水线与寄存器使用，使得少量并行度也能够取得可观收益。