
南開大學
Nankai University

计算机学院

并行程序设计期末报告

# 并行加速的 SIMD 算法

姓名：刘迪乘

学号：2312810

专业：计算机科学与技术

2025 年 4 月 28 日

# 目录

**Abstract**

本文主要探究了 MD5 哈希算法在不同架构下的 SIMD 并行加速方法。在 ARM 平台上，基于 NEON 指令集实现了四路并行计算；在 x86 平台上，分别利用 SSE 与 AVX 指令集实现了 2、4、8、16 路的并行加速。

实验部分，通过实际测量不同版本程序的性能表现，验证了并行化带来的加速效果。同时，借助 Godbolt 平台分析了不同编译优化等级（O1、O2、O3）下的汇编代码，深入探究了编译器自动优化（如函数内联、循环展开、自动矢量化）在性能提升中的关键作用。本文不仅展示了 SIMD 技术在经典哈希算法加速中的应用潜力，也为后续在更多加密算法中引入并行优化提供了参考。

**关键词：** MD5, SIMD 加速, NEON, SSE, AVX, 编译器优化, 并行计算

# 1 实验内容

大问题：利用并行化手段，加速 PCFG 算法口令猜测的训练，猜测过程，同时对生成的猜测进行 MD5 算法加密。

本次问题：利用 SIMD 对 PCFG 算法生成的口令 md5 加密过程进行加速。

# 2 实验环境

本次实验选取的操作系统为 Windows11 下的 WSL(Linux 的版本为 24.04)，以及 windows 系统（用于 X86）。此外作业提交在 OpenEuler 服务器（ARM 架构）上

硬件参数如下：

| 硬件名称 \ 参数 | naive-conv |
|---|---|
| Architecture | x86_64 |
| CPU(s) | 16 |
| Thread(s) per core | 2 |
| Core(s) per socket | 8 |
| Socket(s) | 1 |
| CPU max MHz | 5000 |
| CPU min MHz | 3800 |
| L1 cache | 64K |
| L2 cache | 8MB |
| L3 cache | 32MB |

表 1: AMD R7 9700X 硬件参数

| 硬件名称 \ 参数 | naive-conv |
|---|---|
| Architecture | aarch64 |
| CPU(s) | 96 |
| Thread(s) per core | 1 |
| Core(s) per socket | 48 |
| Socket(s) | 2 |
| CPU max MHz | 2600 |
| CPU min MHz | 200 |
| L1 cache | 128K |
| L2 cache | 512K |
| L3 cache | 49152K |

表 2: OpenEuler 服务器硬件参数

编译器版本：

- gcc version 13.3.0 (Ubuntu 13.3.0-6ubuntu2 24.04)
- gcc version 14.2.0 (MinGW-W64 x86_64-ucrt-posix-seh, built by Brecht Sanders, r3)
- gcc (GCC) 9.3.1 20200408 (Red Hat 9.3.1-2)

代码开源地址:https://github.com/July-h5kf3/NKU-parallel-computing/tree/main

# 3 基础要求

尝试在 ARM 服务器上，基于 NEON 实现 MD5 哈希算法。

具体要求如下：

- 要求有基本的正确性，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。
- 不要求真正实现相对串行算法的加速，但是仍需要给出实验数据，并且分析为什么没有能够实现加速。

## 3.1 算法设计

并行的思路是将输入的消息预处理后涉及的 FF，GG，HH，II，四个函数进行并行。这些函数并不涉及条件判断，并且为较简单的运算 (移位，与运算等)。因此可以让这些函数同时处理多个口令。从而实现并行化。

简单来讲，首先将传入的多个口令预处理，然后在分块处理时将每个口 16 个 32bit 子块向量化 (uint32x4_t)，再将各个函数对应运算操作替换为 ARM NEON 中对应的向量操作指令。具体代码见代码实现部分。

具体流程图如下，并行加速的模块也有所体现:

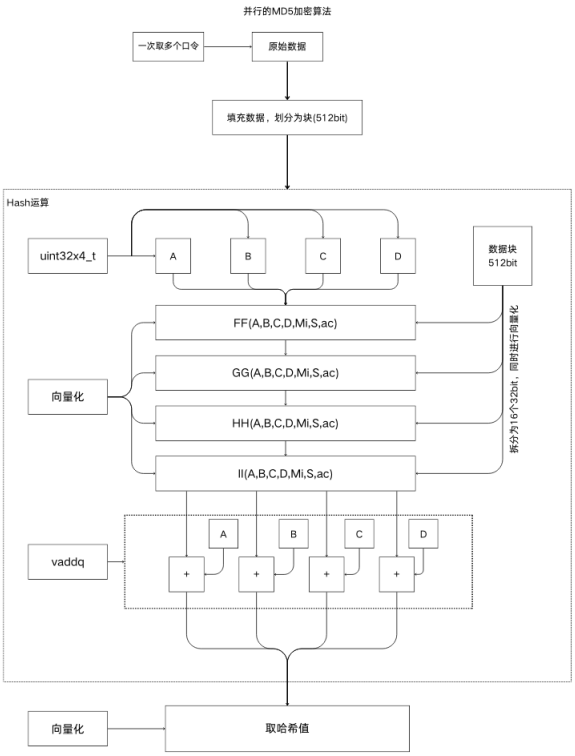



图 3.1: 流程图

## 3.2 代码实现

我从 main 函数接口部分着手，首先将传入的参数由单个口令改为 4 个口令，并适配相应的输出，也就是把 256bit 的缓冲区进行向量化。

```
1 uint32x4_t state[4];
```

```
/*
这部分在hash函数中执行
state[0] = vdupq_n_u32(0x67452301);
state[1] = vdupq_n_u32(0xefcdab89);
state[2] = vdupq_n_u32(0x98badcfe);
state[3] = vdupq_n_u32(0x10325476);
*/
int num = q.guesses.size();
for (int i = 0;i < num;i += 4)
{
    string pw1 = q.guesses[i];
    string pw2 = i+1 >= num ? "" : q.guesses[i + 1];
    string pw3 = i+2 >= num ? "" : q.guesses[i + 2];
    string pw4 = i+3 >= num ? "" : q.guesses[i + 3];
    string pw_batch[4] = {pw1,pw2,pw3,pw4};
    MD5Hash_SIMD(pw_batch,state);
}
```

然后，将各个口令进行初步的预处理，再将预处理后的口令逐 block 处理，由于口令长度的限制，各个口令的 paddedMessage 长度是相等的。对于每个 block，我首先将这个大小为 512bit 的 block 划分为 16 个 32bit 的子块。由于一次处理多个口令，因此要将多个口令得到的子块向量化:

```
uint32x4_t x[16];
bit32 y[4];
for(int i1 = 0;i1 < 16;i1++)
{
    for(int j = 0;j < 4;j++)
    {
        y[j] = (paddedMessage[j][4 * i1 + i * 64]) |
        (paddedMessage[j][4 * i1 + 1 + i * 64] << 8) |
        (paddedMessage[j][4 * i1 + 2 + i * 64] << 16) |
        (paddedMessage[j][4 * i1 + 3 + i * 64] << 24);
    }
    x[i1] = vld1q_u32(&y[0]);
}
```

之后就开始执行一系列的 FF，GG，HH，II 操作，下面展示并行化的相关函数:

md5 加速的核心部分

```
inline uint32x4_t ROTATELEFT_simd(uint32x4_t num,int n)
{
    return vorrq_u32(vshlq_n_u32(num,n),vshrq_n_u32(num,32-n));
}
inline uint32x4_t F_simd(uint32x4_t x,uint32x4_t y,uint32x4_t z)
{
    return vorrq_u32(vandq_u32(x,y),vandq_u32(vmvnq_u32(x),z));
}
inline uint32x4_t G_simd(uint32x4_t x,uint32x4_t y,uint32x4_t z)
{
```

```
    return vorrq_u32(vandq_u32(x,z),vandq_u32(y,vmvnq_u32(z)));
}
inline uint32x4_t H_simd(uint32x4_t x,uint32x4_t y,uint32x4_t z)
{
    return veorq_u32(veorq_u32(x,y),z);
}
inline uint32x4_t I_simd(uint32x4_t x,uint32x4_t y,uint32x4_t z)
{
    return veorq_u32(y,vorrq_u32(x,vmvnq_u32(z)));
}
//考虑到FF,GG,HH,II仅在调用F,G,H,I上存在差异，因此只展示FF
inline void FF_simd(uint32x4_t *a,uint32x4_t b,uint32x4_t c,uint32x4_t d,uint32x4_t
    x,int s,uint32_t ac)
{
    uint32x4_t ac_vec = vdupq_n_u32(ac);
    *a = vaddq_u32(*a,vaddq_u32(F_simd(b,c,d),vaddq_u32(x,ac_vec)));
    *a = ROTATELEFT_simd(*a,s);
    *a = vaddq_u32(*a,b);
}
```

经过 4 轮操作后，将缓冲区更新

```
state[0] = vaddq_u32(state[0],a);
state[1] = vaddq_u32(state[1],b);
state[2] = vaddq_u32(state[2],c);
state[3] = vaddq_u32(state[3],d);
```

最后对缓冲区进行移位操作

```
for(int i = 0;i < 4;i++)
{
    uint32x4_t value = state[i];
    uint32x4_t tmp1 = vdupq_n_u32(0xff);
    uint32x4_t tmp2 = vdupq_n_u32(0xff00);
    uint32x4_t tmp3 = vdupq_n_u32(0xff0000);
    uint32x4_t tmp4 = vdupq_n_u32(0xff000000);
    state[i] = vorrq_u32(vorrq_u32(vshlq_n_u32(vandq_u32(value,tmp1),24),
    vshlq_n_u32(vandq_u32(value,tmp2),8)),
    vorrq_u32(vshrq_n_u32(vandq_u32(value,tmp3),8)
    ,vshrq_n_u32(vandq_u32(value,tmp4),24)));
}
```

### 3.3 性能测试

首先展示我程序的正确性。

为验证程序的正确性，我自行编写了如下代码

```
int main()
{
```

```
    string pwd[4] = {"abc","def","hij","knb"};
    uint32x4_t state_simd[4];
    //auto start_hash = system_clock::now();
    MD5Hash_SIMD(pwd,state_simd);
    //auto end_hash = system_clock::now();
    //cout<<end_hash - start_hash<<endl;
    bit32 state_show[4][4];
    for (int i = 0; i < 4; i++) {
    uint32_t tmp0[4], tmp1[4], tmp2[4], tmp3[4];
    vst1q_u32(tmp0, state_simd[0]);
    vst1q_u32(tmp1, state_simd[1]);
    vst1q_u32(tmp2, state_simd[2]);
    vst1q_u32(tmp3, state_simd[3]);

    state_show[i][0] = tmp0[i];
    state_show[i][1] = tmp1[i];
    state_show[i][2] = tmp2[i];
    state_show[i][3] = tmp3[i];
}

    for(int i = 0;i < 4;i++)
    {
        bit32 state[4];
        MD5Hash(pwd[i],state);
        cout<<"correct:";
        for(int j = 0;j < 4;j++)
        {
            cout << std::setw(8) << std::setfill('0') << hex << state[j];
        }
        puts("");
        cout<<"me:";
        for(int j = 0;j < 4;j++)
        {
            cout << std::setw(8) << std::setfill('0') << hex << state_show[i][j];
        }
        puts("");
    }
    return 0;
}
```

经验证，结果正确

对于我 SIMD 的性能评估，我主要从时间开销以及 profiling 等角度展开。

对于时间开销，我分别统计在不同优化力度下的串行、并行完成 PCFG 产生的所有口令 MD5Hash 过程的时间开销，以及完成不同数据规模的时间平均开销。

| | SIMD | 串行 |
|---|---|---|
| 无编译优化 | 14.2729s | 11.477s |
| O1 | 2.57492s | 5.49482s |
| O2 | 2.35317s | 5.49505s |
| O3 | 2.14581s | 5.31903s |

表 3: 不同优化力度下的时间开销

加速比为:**2.48**. 对于不同数据规模的情况，由于服务器限制，选择在 X86 平台下对其他指令集的 SIMD 进行测试，测试结果见下面部分。

### 3.4 结果分析

## 4 进阶要求

### 4.1 在 x86 架构下利用 SSE 指令集实现 SIMD 并行算法

SSE 指令集下实现 SIMD 算法大体与 NEON 相同，只需要更改一些指令的名称，以及数据类型。具体而言，其 128bit 寄存器名称为 ___int128。涉及的运算变更如下:

```
1 x[i1] = vld1q_u32(&y[0]); -> x[i1] =
      _mm_loadu_si128((___m128i*)y);//从内存中加载数
2 state[0] = vdupq_n_u32(0x67452301); -> state[0] =
      _mm_set1_epi32(0x67452301);//向量广播操作
3 vshlq_n_u32(num,n); -> _mm_slli_epi32(num,n);//左移操作
4 vshrq_n_u32(num,n); -> _mm_srli_epi32(num,n);//右移操作
5 vorrq_u32(x,y); -> _mm_or_si128(x,y);//按位或操作
6 vandq_u32(x,y); -> _mm_and_si128(x,y);//按位与操作
7 veorq_u32(x,y); -> _mm_xor_si128(x,y);//按位异或操作
8 vaddq_u32(x,y); -> _mm_add_epi32(x,y);//加法
9 vmvnq_u32(x) -> _mm_xor_si128(x,_mm_set1_epi32(0xFFFFFFFF);
      //按位取反操作，由于在SSE指令集中没有需要通过异或实现。
```

最后的时间开销对比，由于在 X86 平台和 ARM 平台下二者硬件参数存在差异，因此比较意义不大，只对串并行算法做对比。

| | SIMD | 串行 |
|---|---|---|
| 无编译优化 | 3.40103s | 2.64341s |
| O1 | 0.859473s | 1.1589s |
| O2 | 0.869919s | 1.09493s |
| O3 | 0.630818s | 1.07665s |

表 4: 不同优化力度下的时间开销

由此可见在 X86 架构下，运行速度显著快于 ARM 架构，一方面是硬件相关参数上的差异，另一方面是二者设计的差异前者为了高性能，而后者则为了低能耗牺牲了部分性能。

在 SSE 下的加速比大约为 **1.70**. 低于 ARM 架构.

## 4.2 尝试不同并行度

在原版 SIMD 的基础上，我还尝试了同时处理 2,4,8,16 个口令的 SIMD。出于保证架构一致的考虑，均在 X86 架构下使用 AVX 指令集（这个指令集包含了 SSE）。由于 64 位寄存器现代编译器已经不对其操作进行优化，因此对于两个口令的同时处理，考虑采用 128 位寄存器，只利用其两位的方式实现。对于 8 个以及 16 个口令的 SIMD 则分别使用 AVX 的 256bit 寄存器和 AVX-512 的 512bit 指令集。

二路并行指令集与四路并行的指令集相同，因此不做赘述；

八路并行的 256bit 寄存器的名称为 ___m256i, 十六路并行的 512bit 寄存器名称为 ___m512i.

| **AVX 256-bit (YMM)** | **AVX-512 512-bit (ZMM)** | **注释** |
|---|---|---|
| `_mm256_loadu_si256((__m256i*)y);` | `_mm512_loadu_si512((__m512i*)y);` | 从内存中加载数据 |
| `_mm256_set1_epi32(0x67452301);` | `_mm512_set1_epi32(0x67452301);` | 向量广播操作 |
| `_mm256_slli_epi32(num,n);` | `_mm512_slli_epi32(num,n);` | 左移操作 |
| `_mm256_srli_epi32(num,n);` | `_mm512_srli_epi32(num,n);` | 右移操作 |
| `_mm256_or_si256(x,y);` | `_mm512_or_si512(x,y);` | 按位或操作 |
| `_mm256_and_si256(x,y);` | `_mm512_and_si512(x,y);` | 按位与操作 |
| `_mm256_xor_si256(x,y);` | `_mm512_xor_si512(x,y);` | 按位异或操作 |
| `_mm256_add_epi32(x,y);` | `_mm512_add_epi32(x,y);` | 加法 |
| `_mm256_xor_si256(x,_mm256_set1_epi32(0xFFFFFFFF));` | `_mm512_xor_si512(x,_mm512_set1_epi32(0xFFFFFFFF));` | 按位取反（通过异或实现） |

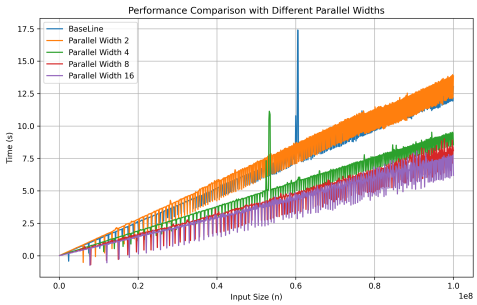
表 5: AVX/AVX-512 指令对照表（针对 `uint32_t` 数据类型）

实验均在 O3 优化下进行，时间开销以及加速比结果如下：

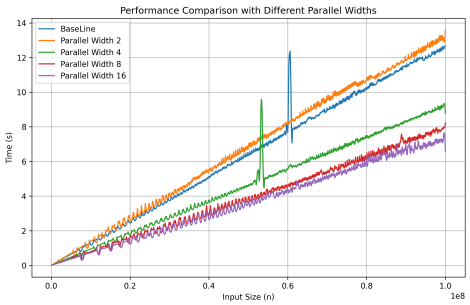
| | 串行 | 二路并行 | 四路并行 | 八路并行 | 十六路并行 |
|---|---|---|---|---|---|
| 时间 | 1.07665s | 0.989887s | 0.630818s | 0.595189s | 0.607883s |
| 加速比 | | 1.02 | 1.70 | **1.81** | 1.78 |

表 6: 不同并行程度下的开销

此外，测试了在不同问题规模下，不同并行路数的时间开销，结果如下：




((a)) 原始不同规模并行度时间开销




((b)) 平滑化后的

图 4.2: 不同规模不同并行度 SIMD 时间开销

我是以 100000 作为初始大小，以步长为 100000 逐步增加到 100000000 从而测试不同并行度算法在不同问题规模下的时间开销。由于受系统波动影响，原始图像呈现出震荡，不便于观察趋势，因此对原始数据的图像做了一定的平滑化操作。

可以发现总体上来说，并行度越高，其性能就越好。

虽然从平滑化后的数据可以发现二路版本的时间开销较串行版本有所增加，但实际上二路并行的版本是优于串行版本的，只是加速比很小。这很大程度上是因为目前支持 64bit(2*32bit) 的寄存器已

经不再被编译优化，因此采用 128bit 的寄存器空出 64bit 的方法实现，但是这样可能会导致内存访问不连续的问题，从而导致其优化性能不佳。

## 4.3 Profiling + 加速成功分析

根据之前的实验结果，在未启用编译器优化时，SIMD 版本的执行时间反而高于串行版本；而在启用编译器优化后，SIMD 版本则成功实现了对串行版本的性能超越。

在进一步分析这一现象背后的机制之前，我们先通过下表了解编译器不同优化等级（O1、O2、O3）所对应的特性。

| 优化等级 | 特性说明 |
|---|---|
| **O1** | 启用基础优化，如删除死代码、简化表达式、减少跳转。在保证编译速度较快的同时，能够提升程序基本性能，且代码稳定性较高。 |
| **O2** | 启用更高级优化，如函数内联、循环优化、跳转预测，更激进的死代码消除。性能与稳定性平衡良好。 |
| **O3** | 启用最激进的优化，包括大规模循环展开、深度内联、自动矢量化（SIMD）、预取指令等，最大化性能但编译时间更长。 |

表 7: 编译器不同优化等级（O1、O2、O3）特性总结

从实验现象来看，只要开启编译器优化（即使仅为 O1 等级），SIMD 实现的性能即明显优于串行版本。为了深入理解其原因，我在 Godbolt 平台上生成 SSE 下的 SIMD 算法在无编译优化以及 O1 编译优化下的汇编代码，发现开启 O1 编译优化的代码，相较于没有开启编译优化的，增删改了部分代码。

例如，大量原本显式调用的 FF、GG、HH、II 等 Hash 运算函数被删除，相应地，这些运算逻辑被直接内联进了主处理流程中（图中红色部分为删除内容，黄色部分为修改或新增内容）：

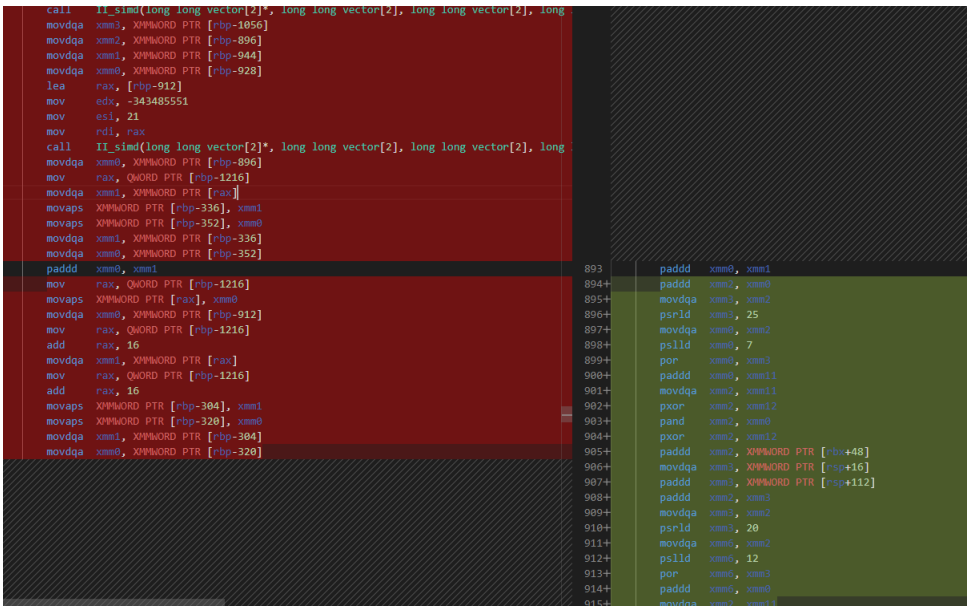



图 4.3: 汇编代码对比

通过函数的内联，减少了调用函数时参数的频繁的出栈入栈，从而大大加速了程序的运行速度，从

而实现了加速。

进一步将 O1 与 O3 优化后的汇编代码进行对比，可以观察到 O3 引入了更加激进的优化策略：不仅将主要处理函数（如 StringProcess）完整地内联到 MD5_Hash 函数内部:

```
        call    std::__cxx11::basic_string<char, std::char_traits<char>, st
        mov     qword ptr [rsp + 16], rax
        mov     rcx, qword ptr [rsp + 400]
        mov     qword ptr [rsp + 32], rcx
.LBB2_3:
        test    rbp, rbp
        je      .LBB2_22
        cmp     rbp, 1
        jne     .LBB2_21
        movzx   ecx, byte ptr [r14]
        mov     byte ptr [rax], cl
        jmp     .LBB2_22
.LBB2_21:
        mov     rdi, rax
        mov     rsi, r14
        mov     rdx, rbp
        call    memcpy@PLT
.LBB2_22:
        mov     qword ptr [rsp + 80], rbx
        mov     rax, qword ptr [rsp + 400]
        mov     qword ptr [rsp + 24], rax
        mov     rcx, qword ptr [rsp + 16]
        mov     byte ptr [rcx + rax], 0
        mov     rax, qword ptr [rsp + 16]
        mov     qword ptr [rsp], rax
        mov     r13, qword ptr [rsp + 24]
        lea     eax, [8*r13]
        lea     ecx, [8*r13 + 511]
        test    eax, eax
        cmovns  ecx, eax
        and     ecx, -512
        sub     eax, ecx
        mov     ecx, 448
        sub     ecx, eax
        cmp     eax, 448
        mov     edx, 512
```

((a)) O3 优化下函数内联

```
        sub     rsp, 424
        lea     r14, [rsp+160]
        lea     r15, [rsp+176]
.L50:
        mov     rsi, QWORD PTR [r13+0+r12*8]
        mov     rdx, QWORD PTR [r13+8+r12*8]
        mov     rdi, r14
        mov     QWORD PTR [rsp+160], r15
        call    void std::__cxx11::basic_string<char, std::char_traits<char>,
        lea     rsi, [rsp+96+r12]
        mov     rdi, r14
        call    StringProcess(std::__cxx11::basic_string<char, std::char_traits
        mov     rdi, QWORD PTR [rsp+160]
        lea     rbx, [rsp+128]
        mov     QWORD PTR [rbx+r12*2], rax
        cmp     rdi, r15
        je      .L48
        mov     rax, QWORD PTR [rsp+176]
        lea     rsi, [rax+1]
        call    operator delete(void*, unsigned long)
```

((b)) O1 优化下部分函数未内联

图 4.4: 不同优化下汇编代码差异分析

同时也能看到，在 O3 优化的版本下拥有更多的向量寄存器指令，这表明 O3 优化，会进一步进行循环展开以及尝试 SIMD。这样一方面减少了对内存的访问，同时也提升了运算的效率。

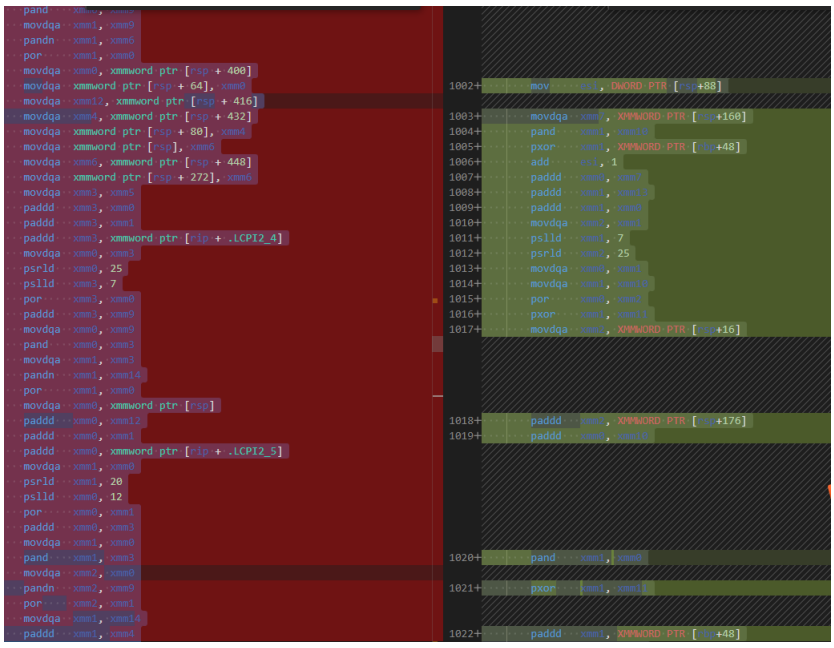


图 4.5: 更多的向量指令

通过自动矢量化与循环展开，程序显著减少了内存访问次数，并提升了指令级并行度，从而最大

限度地发挥了 SIMD 的性能潜力。

总体而言，从 O1 到 O3 的逐步优化，不仅解决了初期 SIMD 指令调度效率不高的问题，也充分体现了编译器优化在实际加速过程中所扮演的重要角色。

# 参考文献