
南開大學
Nankai University

计算机学院

并行程序设计报告

# 口令猜测中的 MD5 哈希算法 SIMD 并行化

姓名：王众

学号：2313211

专业：计算机科学与技术

2025 年 4 月 22 日

# 目录

**1 实验平台** **2**

**2 基础任务** **2**

2.1 任务内容分析 . . . 2
2.2 并行框架的实现 . . . 2
2.3 性能的优化可行方案总结 . . . 3
2.3.1 打开循环 . . . 5
2.3.2 内联函数改宏定义 . . . 6
2.3.3 优化内存读取方式 . . . 7
2.4 尝试过的但是未优化方案 . . . 8

**3 进阶要求** **9**

3.1 o1o2 由优化编译的比较 . . . 9
3.1.1 宏展开优化 . . . 9
3.1.2 向量指令调度 . . . 10
3.1.3 寄存器分配优化 . . . 10
3.1.4 循环优化 . . . 11
3.1.5 函数内联 . . . 11
3.1.6 为什么不编译优化并行速度不能超过串行 . . . 11
3.1.7 o1o2 编译优化的区别 . . . 12
3.1.8 非优化编译与 O2 优化编译的 perf 结果对比 . . . 12
3.2 SSE 指令集实现并行算法 . . . 13
3.2.1 指令的对应关系 . . . 13
3.2.2 运用指令改造相应的宏定义函数 . . . 13
3.3 探究并行度对效率的影响 . . . 14
3.3.1 不同路数之间的优化幅度 . . . 15
3.3.2 随着路数增加优化编译对于速度的影响 . . . 15
3.3.3 不同并行度的信息处理速度 . . . 16

**4 实验小结** **16**

**5 Github 链接** **16**

# 1 实验平台

- 鲲鹏 ARM 服务器平台
- Perf
- x86(win 本)

# 2 基础任务

## 2.1 任务内容分析

在给定的框架函数中，给定了三类宏定义：分别是单操作数的位运算、单操作数的循环左移、单操作数的核心轮函数。显然其中的变量只能使用单个操作数，这也是我们需要进行并行化的主要部分。

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

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

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

## 2.2 并行框架的实现

我们查阅资料找到了 NEON 的 SIMD 框架下的 32x4 操作数的各项操作对应的运算符：

| SIMD 操作符 | 普通运算符 |
| --- | --- |
| vandq_u32((x), (y)) | x&y |
| vorrq_u32((x), (y)) | x\|y |
| vmvnq_u32((x)) | ~x |
| vdupq_n_u32(xy) | 填充向量 |
| veorq_u32((x), (y)) | x^y |
| vshlq_n_u32((num), (n)) | (num) << (n) |
| vshrq_n_u32((num), (n)) | (num) >> (n) |

最开始的时候我们因为对于该函数会在我们新创建的 MD5-SIMD 版本中反复使用，便选择了内联函数的版本。得到了以下代码：

```
inline bit32x4_t F_SIMD(bit32x4_t x, bit32x4_t y, bit32x4_t z) {
    return vorrq_u32(vandq_u32(x, y), vandq_u32(vmvnq_u32(x), z));
```

```
}
inline bit32x4_t ROTATELEFT_SIMD(bit32x4_t x, int n) {
    return vorrq_u32(vshlq_n_u32(x, n), vshrq_n_u32(x, 32 - n));
}
inline bit32x4_t FF_SIMD(*a, b, c, d, x, s, ac) { \
    (*a) = vaddq_u32((a), vaddq_u32(vaddq_u32(F_SIMD((b), (c), (d)), (x)),  vdupq_n_u32(ac)));
    (*a) = ROTATELEFT_SIMD((a), (s)); \
    (*a) = vaddq_u32((a), (b)); \
}
```

实现了关键的内部调用函数之后，我们需要重写一个 MD5 的 SIMD 版本的调用函数以适应我们的一次性四位操作数。以下是代码的伪代码部分：

在这里我们在输入的时候因为已经限制了是一次性四个操作数，所以我们直接使 input[] 数组而不是 input* 这个指针变量，以免因为每次调用函数的时候需要给指针分配空间从而导致并行效率变低。stat 状态我们也使用 [4][4](不过使用数组可能会在找下标的阶段出现耗时巨大的问题，这个我们放在速度优化的地方再讨论)。基本上之前在函数里面的单个的数据现在都需要变成一个四位的数组，用来一次性储存多个变量。在最后也需要把所有的空间释放掉。

在编译没有报错之后，我们便可以开始进行运行编译文件了。首先我们先用 correctness.cpp 文件验证一下我们实现的正确性。

```
guess > ≡ test.o
1  bba46eb8b53cf65d50ca54b2f8afd9db
2  原始MD5Hash结果: bba46eb8b53cf65d50ca54b2f8afd9db
3  验证结果: 相同
4
5  Authorized users only. All activities may be monitored and reported.
6
```

图 2.1: 并行化后代码的正确性报告

但是在这之前我们先进行串行代码的运行，以串行代码作为我们优化时间的基准线。

```
112  Guesses generated: 9997458
113  Guesses generated: 10097691
114  Guess time:18.7103seconds
115  Hash time:9.05261seconds
116  Train time:0.856026seconds
117
118  Authorized users only. All activities may be monitored and reported.
119
```

(a) 串行时间

```
86  Guesses generated: 9853408
87  Guesses generated: 10106852
88  Guess time:8.23305seconds
89  Hash time:13.8043seconds
90  Train time:98.4926seconds
91
92  Authorized users only. All activities may be monitored and reported.
93
```

(b) 并行时间 (初始)

图 2.2: 最开始的串行并行时间对比

## 2.3 性能的优化可行方案总结

因为到最后我都没能在不编译优化的情况下是的并行速度快于串行的之前的基准速度，但是我们可以进一步地去逼近串行的时间，使得自己代码的运行速度更快。

需要提醒的是我们在优化阶段一直使用的是 perf 来检测文件运行的速度，但是 perf 上本地运行的时间与鲲鹏服务器上地运行时间略有差异，所以我们需要再在 perf 上跑一个基准时间，二者的初始时间如下：

**Algorithm 1** MD5 SIMD 并行处理

**Input:** 输入字符串数组 $inputs[4]$
**Output:** 输出哈希值 $states[4][4]$

**function** MD5Hash_SIMD($inputs[4]$, $states[4][4]$)
  初始化 4 组哈希状态向量 $a_0, b_0, c_0, d_0$ 分别为 0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476
  创建填充后的消息数组 $paddedMessages[4]$
  创建消息长度数组 $messageLengths[4]$
  **for** $i \leftarrow 0$ **to** 3 **do**
    $paddedMessages[i] \leftarrow$ StringProcess($inputs[i]$, $\&messageLengths[i]$)
  **end for**
  $n\_blocks \leftarrow messageLengths[0]/64$
  **for** $block \leftarrow 0$ **to** $n\_blocks - 1$ **do**
    创建块向量数组 $M[16]$
    **for** $j \leftarrow 0$ **to** 15 **do**
      **for** $i \leftarrow 0$ **to** 3 **do**
        从 $paddedMessages[i]$ 的第 $block$ 块中提取第 $j$ 个 32 位字 $values[i]$
      **end for**
      $M[j] \leftarrow$ vld1q_u32($values$)
    **end for**
    $A \leftarrow a_0, B \leftarrow b_0, C \leftarrow c_0, D \leftarrow d_0$
    执行 64 步 MD5 轮函数 (FF, GG, HH, II)，使用向量操作同时处理 4 个消息
    $a_0 \leftarrow a_0 + A$
    $b_0 \leftarrow b_0 + B$
    $c_0 \leftarrow c_0 + C$
    $d_0 \leftarrow d_0 + D$
  **end for**
  提取最终哈希值 $a\_values[4], b\_values[4], c\_values[4], d\_values[4]$
  **for** $i \leftarrow 0$ **to** 3 **do**
    $states[i][0] \leftarrow$ ByteSwap($a\_values[i]$)
    $states[i][1] \leftarrow$ ByteSwap($b\_values[i]$)
    $states[i][2] \leftarrow$ ByteSwap($c\_values[i]$)
    $states[i][3] \leftarrow$ ByteSwap($d\_values[i]$)
  **end for**
  **for** $i \leftarrow 0$ **to** 3 **do**
    释放 $paddedMessages[i]$
  **end for**
  **return** $states$
**end function**


(a) 串行时间　　(b) 并行时间

图 2.3: 在 perf 上的串行并行时间对比

### 2.3.1 打开循环

我们在 perf 上面打开界面之后去看汇编代码并寻找耗时最多的汇编语句。

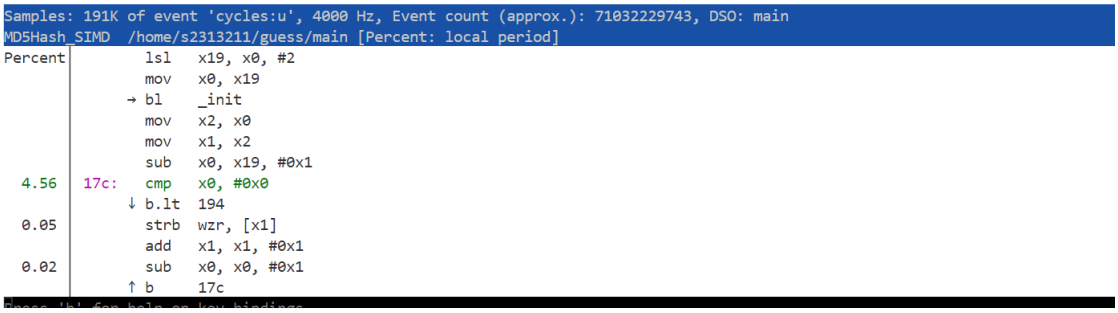



图 2.4: 打开 SIMD 函数后占时间最多的汇编语句

我们可以看到下面该比较语句是一个耗时非常的大语句，而这个比较行为通常是在进行 for 语句循环的时候产生的。我们将原串行代码与我们现在的代码进行比较，果然发现：由于一次性的操作数从一个变为了四个，所以对应的赋值处理语句也从之前的单变量变为了由 for 循环循环赋值的语句，而在反复的比较中，时间会加长，从而导致并行代码效率变低。

```
1  cmp        x0      #0x0
```

所以我们现在对我们原本的代码做如下修改：即讲所有的增加的 for 循环的括号取消（但是如果串行代码中本身就有的地方就不用优化了保持变量统一的正确性）。

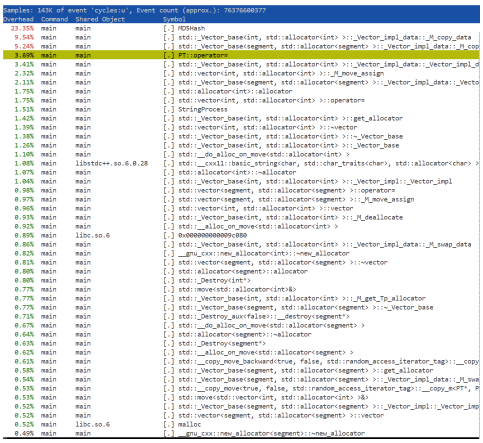
```
1  for (int i = 0; i < 4; i++) {
2      paddedMessages[i] = StringProcess(inputs[i], &messageLengths[i]);
3  }
4
5  paddedMessages[0] = StringProcess(inputs[0], &messageLengths[0]);
6  paddedMessages[1] = StringProcess(inputs[1], &messageLengths[1]);
7  paddedMessages[2] = StringProcess(inputs[2], &messageLengths[2]);
8  paddedMessages[3] = StringProcess(inputs[3], &messageLengths[3]);
```

在进行优化之后。我们可以看见运行时间稳定减少，并且查看汇编代码，耗时最大的已经不是 for 循环了。

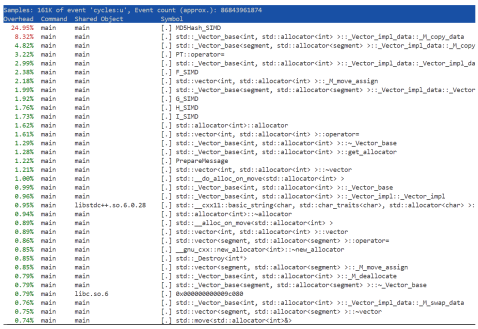
```
Guesses generated: 9595702
Guesses generated: 9696752
Guesses generated: 9796757
Guesses generated: 9897171
Guesses generated: 9997458
Guesses generated: 10097691
Guess time:30.9011seconds
Hash time:17.4475seconds
Train time:1.42324seconds
[ perf record: Woken up 56 times to write data ]
[ perf record: Captured and wrote 14.753 MB perf.data (318200 samples) ]
```

图 2.5: 将新添加的 for 循环打开后的 hashtime

### 2.3.2 内联函数改宏定义


Samples: 143K of event 'cycles:u', Event count (approx.): 76376600377
Overhead Command Shared Object Symbol
23.35% main main [.] MD5Hash
9.54% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_impl_data::_M_copy_data
5.24% main main [.] std::_Vector_base<segment, std::allocator  >::_Vector_impl_data::_M_cop
3.69% main main [.] PT::operator=
3.41% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_impl_data::_Vector_impl_d
2.32% main main [.] std::vector<int, std::allocator<int> >::_M_move_assign
2.11% main main [.] std::_Vector_base<segment, std::allocator  >::_Vector_impl_data::_Vecto
1.75% main main [.] std::allocator<int>::allocator
1.75% main main [.] std::vector<int, std::allocator<int> >::operator=
1.51% main main [.] StringProcess
1.42% main main [.] std::_Vector_base<int, std::allocator<int> >::get_allocator
1.39% main main [.] std::vector<int, std::allocator<int> >::~vector
1.38% main main [.] std::_Vector_base<int, std::allocator<int> >::~_Vector_base
1.26% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_base
1.10% main main [.] std::__do_alloc_on_move<std::allocator<int> >
1.08% main libstdc++.so.6.0.28 [.] std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >
1.07% main main [.] std::allocator<int>::~allocator
1.04% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_impl::_Vector_impl
0.98% main main [.] std::vector<segment, std::allocator  >::operator=
0.97% main main [.] std::vector<segment, std::allocator  >::_M_move_assign
0.96% main main [.] std::vector<int, std::allocator<int> >::vector
0.91% main main [.] std::_Vector_base<int, std::allocator<int> >::_M_deallocate
0.92% main main [.] std::__alloc_on_move<std::allocator<int> >
0.89% main libc.so.6 [.] 0x000000000009c080
0.86% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_impl_data::_M_swap_data
0.82% main main [.] __gnu_cxx::new_allocator<int>::~new_allocator
0.81% main main [.] std::vector<segment, std::allocator  >::~vector
0.80% main main [.] std::allocator::allocator
0.80% main main [.] std::_Destroy<int*>
0.77% main main [.] std::move<std::allocator<int>&>
0.77% main main [.] std::_Vector_base<int, std::allocator<int> >::_M_get_Tp_allocator
0.77% main main [.] std::_Vector_base<segment, std::allocator  >::~_Vector_base
0.71% main main [.] std::_Destroy_aux<false>::__destroy<segment*>
0.67% main main [.] std::__do_alloc_on_move<std::allocator  >
0.64% main main [.] std::allocator::~allocator
0.61% main main [.] std::_Destroy<segment*>
0.62% main main [.] std::__alloc_on_move<std::allocator  >
0.61% main main [.] std::__copy_move_backward<true, false, std::random_access_iterator_tag>::__copy
0.58% main main [.] std::_Vector_base<segment, std::allocator  >::get_allocator
0.54% main main [.] std::_Vector_base<segment, std::allocator  >::_Vector_impl_data::_M_swa
0.53% main main [.] std::__copy_move<true, false, std::random_access_iterator_tag>::__copy_m<PT*, P
0.53% main main [.] std::move<std::vector<int, std::allocator<int> >&>
0.52% main main [.] std::_Vector_base<segment, std::allocator  >::_Vector_impl::_Vector_imp
0.52% main main [.] std::vector<segment, std::allocator  >::vector
0.52% main libc.so.6 [.] malloc
0.49% main main [.] __gnu_cxx::new_allocator::~new_allocator

(a) 串行 perf


Samples: 161K of event 'cycles:u', Event count (approx.): 86843961874
Overhead Command Shared Object Symbol
24.95% main main [.] MD5Hash_SIMD
8.32% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_impl_data::_M_copy_data
4.82% main main [.] std::_Vector_base<segment, std::allocator  >::_Vector_impl_data::_M_copy
3.22% main main [.] PT::operator=
2.99% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_impl_data::_Vector_impl_da
2.38% main main [.] F_SIMD
2.18% main main [.] std::vector<int, std::allocator<int> >::_M_move_assign
1.99% main main [.] std::_Vector_base<segment, std::allocator  >::_Vector_impl_data::_Vector
1.92% main main [.] G_SIMD
1.76% main main [.] H_SIMD
1.73% main main [.] I_SIMD
1.62% main main [.] std::allocator<int>::allocator
1.61% main main [.] std::vector<int, std::allocator<int> >::operator=
1.29% main main [.] std::_Vector_base<int, std::allocator<int> >::~_Vector_base
1.28% main main [.] std::_Vector_base<int, std::allocator<int> >::get_allocator
1.22% main main [.] PrepareMessage
1.21% main main [.] std::vector<int, std::allocator<int> >::~vector
1.06% main main [.] std::__do_alloc_on_move<std::allocator<int> >
0.99% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_base
0.96% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_impl::_Vector_impl
0.95% main libstdc++.so.6.0.28 [.] std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::
0.94% main main [.] std::allocator<int>::~allocator
0.89% main main [.] std::__alloc_on_move<std::allocator<int> >
0.89% main main [.] std::vector<int, std::allocator<int> >::vector
0.86% main main [.] std::vector<segment, std::allocator  >::operator=
0.85% main main [.] __gnu_cxx::new_allocator<int>::~new_allocator
0.85% main main [.] std::_Destroy<int*>
0.85% main main [.] std::vector<segment, std::allocator  >::_M_move_assign
0.79% main main [.] std::_Vector_base<int, std::allocator<int> >::_M_deallocate
0.79% main main [.] std::_Vector_base<segment, std::allocator  >::~_Vector_base
0.79% main libc.so.6 [.] 0x000000000009c080
0.76% main main [.] std::_Vector_base<int, std::allocator<int> >::_Vector_impl_data::_M_swap_data
0.75% main main [.] std::vector<segment, std::allocator  >::~vector
0.74% main main [.] std::move<std::allocator<int>&>

(b) 并行 perf

图 2.6: 串行与并行的 perf 图

我们打开 perf 的总时间消耗表，可以看到我们使用的内联函数的耗时尤为突出，所以我们开始进行资料的查阅。接下来是可能的一些解释。首先是宏定义可以完全避免函数调用机制，不存在任何函数调用框架，而内联函数即使被内联，编译器仍可能会保留一些函数调用特性。在汇编层面，宏展开不会产生任何函数调用相关指令。我们从以下汇编代码来进行分析：由于每次调用内联函数的时候都会将当前的基址指针 (rbp 寄存器的值) 保存到栈，建立新的栈帧，而这个多出来的处理在我们写的代码中是作为主体存在的，便会造成大量的时间消耗。同时宏展开后的代码被视为单一代码块，编译器能进行更全局的寄存器分配优化，减少寄存器保存/恢复操作。这一点在 O1,O2 编译优化的时候我们再进行讨论。综上所述这些微小的差异在性能极度敏感的场景下（如加密算法中的位操作）会累积成显著的性能差距。

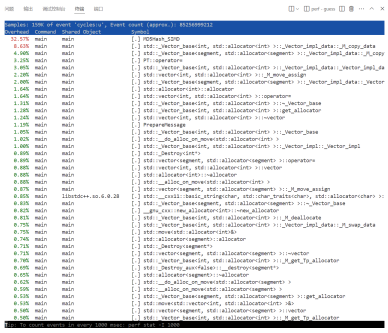
```
// 宏展开生成的汇编
mov eax, [var1]
add eax, [var2]  // 直接嵌入的操作

// 内联函数生成的汇编
push rbp          // 即使内联，仍有函数框架代码
mov rbp, rsp
mov eax, [var1]
add eax, [var2]
pop rbp
```

所以我们统一将 inline 函数改为 # 宏定义：

```
inline bit32x4_t F_SIMD(bit32x4_t x, bit32x4_t y, bit32x4_t z) {
    return vorrq_u32(vandq_u32(x, y), vandq_u32(vmvnq_u32(x), z));
```

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

在将内联函数改为宏定义之后，果然 hashtime 进一步减少。并且在 perf 报告中内联函数的消耗也没有了。



(a) 并行的 perf 报告

```
Guesses generated: 9490827
Guesses generated: 9595702
Guesses generated: 9696752
Guesses generated: 9796757
Guesses generated: 9897171
Guesses generated: 9997458
Guesses generated: 10097691
Guess time:28.6479seconds
Hash time:16.6006seconds
Train time:1.26677seconds
[ perf record: Woken up 46 times to write data ]
[ perf record: Captured and wrote 11.763 MB perf.data (252457 samples) ]
```

(b) hashtime 的截图

图 2.7: 内联函数改宏定义的效果

### 2.3.3 优化内存读取方式

由于并行程序与串行设计的一次性读取的数据量不同，并且如果每次循环结束之后将 4 个指针全部释放下一次调用的时候再重新 new4 个指针太消耗时间了，所以我们开始进行大改造：首先需要改造信息读取的方式，然后我们可以建造一个缓冲池，来储存即将处理的数据。最后我们将指针声明为全局变量，在函数使用完之后不进行 delete 而是储存起来，下一次调用的时候直接调用已经存在的指针就行。

(1) 指针改造

```
//在 md5.h 里面声明指针变量
static Byte zero_buffer[MAX_BUFFER_SIZE] = {0};
//在 md5cpp 里面定义释放指针的函数，并在 correctness 和 main 函数里面调用避免内存泄漏
void CleanupMD5Resources() {
    delete[] reusable_buffers;
    reusable_buffers = nullptr;
    total_buffer_capacity = 0;
}
```

(2) 缓存区

```
// 直接写入传入的 output 缓冲区
memcpy(output, input.c_str(), input_length);
//使用优化的储存方式，只有在大于的时候才使用效率低 memset 填充数据。
//其他时候直接使用赋值语句进行填充，增加效率。。
if (padding_bytes - 1 <= MAX_BUFFER_SIZE) {
```

```
    memcpy(output + input_length + 1, zero_buffer, padding_bytes - 1);
}
else {
memset(output + input_length + 1, 0, padding_bytes - 1);
}
```

(3) 静态数组预分配

```
// 使用静态预分配的数组替代栈上数组
bit32x4_t* M = M_static;
```

经过以上的优化之后我们在数据处理和内存使用方面已经很好地适配了一次性 4 个操作数的情况。在 hashtime 上也进一步地减少了。

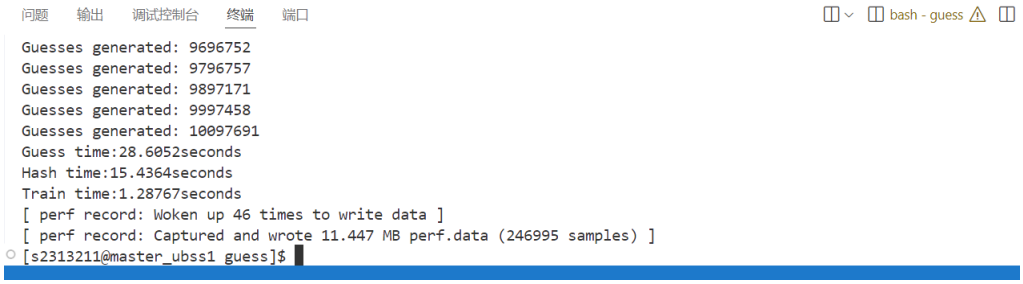



图 2.8: 优化内存的读取方式之后得到的时间

## 2.4 尝试过的但是未优化方案

在根据汇编语句进行优化的过程中，我们难免会出现进行了优化并且从理论分析上看应该会出现优化之后，出现稳定时间基本不变甚至负优化的情况。限于篇幅我们选取其中在不同的优化编译方式的汇编代码反复出现的却无法解决的高耗时点为例子:

```
str q0, [x4]
add w2, w2
add x0, x0
cmp w2
//对应的实际代码段是 memcpy 的行为
memcpy(&values[0], paddedMessages[0] + offset, 4);
```

我们发现，不管是不优化编译还是 o1o2 编译，这些内存复制和填充操作本身就是带宽受限的即使编译器将 memcpy 内联，仍然需要将数据从输入字符串移动到对齐缓冲区。而缓冲区这一操作是我们在进行内存读取优化之后新建立的在之前的直接读取的基础上优化来的。并且 memcpy 是编译器高度优化的内建函数，代编译器通常会将 memcpy 完全内联，优化后的 memcpy 可能直接映射到单条 ldr/str 指令，不会有实际函数调用。如果我们将它换成其他函数，很难说对于编译器来说会不会继续维持现在的内联程度。

在后续的探究中我们可以发现，memcpy 这个部分的耗时比较大并不是 memcpy 本身所造成的，而是数据布局导致的缓存不友好访问造成的。如果我们真的想要进一步优化，可能需要考虑预先将四个消息按交错方式组织，使相应位置的数据在内存中相邻。但是有个矛盾就是这个已经到了编译优化的范畴了，并不是我们纯粹并行化所带来的加速收益，所以并没有进行相应的代码优化。

# 3 进阶要求

## 3.1 o1o2 由优化编译的比较

虽然我们没能在非优化编译语句进行编译时是的并行速度快于串行速度，但是框架给了我们 o1,o2 的优化选项，所以我们先尝试在哦 o1,o2 上进行优化编译，同时尝试串行和并行的运行，并进行比较。
首先我们需要分析为什么 o1o2 编译方式会使得运行时间本身减少。我们将从以下几点来进行说明：


(a) 串行 (b) 并行

图 3.9: o1 的串行与并行的比较


(a) 串行 (b) 并行

图 3.10: o2 的串行与并行的比较

| 编译方式 | 串行运行时间 (s) | 并行运行时间 (s) |
|---|---|---|
| o1 | 2.987 | 1.334 |
| o2 | 2.770 | 1.285 |
| 不优化编译 | 9.78 | 10.42 |

表 1: 编译优化之后的串行和并行的 hash 速度

### 3.1.1 宏展开优化

关于宏展开优化，其中包含了智能代码合并、常量传播折叠、公共子表达式消除和内存访问减少。这里我们选用比较简单的公共子表达式消除来解释。

```
1 //在进行数据处理时这个宏定义函数被反复使用，但是在不同的调用中参数并不是完全不同的
2 #define F_SIMD(x, y, z) (vorrq_u32(vandq_u32((x), (y)), vandq_u32(vmvnq_u32((x)), (z))))
3 //例如下面两个调用语句中的 b 是相同的
4 result1 = F_SIMD(b, c, d);      result2 = F_SIMD(b, e, f);
```

```
//所以我们可以将 vmvnq_u32(b) 单独提出来变为一个将要反复调用的临时变量，可以避免重复计算。
temp_not_b = vmvnq_u32(b);
//即变为一下两个表达式
result1 = vorrq_u32(vandq_u32(b, c), vandq_u32(temp_not_b, d));
result2 = vorrq_u32(vandq_u32(b, e), vandq_u32(temp_not_b, f));
```

### 3.1.2 向量指令调度

向量指令调度分为了：依赖关系分析、流水线填充优化、指令延迟隐藏、SIMD 指令融合一共四种。我们接下来用具体的例子说明：

```
//在进行运算处理的主要步骤时
FF_SIMD(A, B, C, D, M[0], s11, 0xd76aa478);
FF_SIMD(D, A, B, C, M[1], s12, 0xe8c7b756);
FF_SIMD(C, D, A, B, M[2], s13, 0x242070db);
.......
```

在编译优化之前，指令执行的顺序是：编译器按照 MD5 算法的顺序先计算每一步，完成上一个操作后才开始下一个操作。这样的话临时结果可能反复存入内存再取出。在进行编译优化之后：编译器会分析连续的多个 FF_SIMD 调用间的依赖关系。在处理第一个 FF_SIMD(A, B, C, D, M[0], s11, 0xd76aa478) 时，同时开始预加载下一个操作需要的 M[1] 数据。当执行完第一个宏中的 vaddq_u32 计算还在流水线中时，编译器可能已经开始计算下一个 FF_SIMD 调用中的 F_SIMD(D, A, B, C)。

特别地，在进行连续操作时，编译优化模式会进一步地对代码进行优化：优化编译器会识别这种计算模式，将操作交错执行，确保处理器的每个部件都在工作，而不是等待前一步完成。PrepareMessage 这个函数在优化编译器处理下，会与主要哈希计算交错执行。例如，在等待 NEON 指令完成时，CPU 可以同时准备下一批消息数据，提高整体效率。

### 3.1.3 寄存器分配优化

在 MD5 SIMD 代码中，编译器会对多个函数块的变量进行全局分析：编译器识别 A, B, C, D 是核心变量，将它们分配到固定的 NEON 寄存器（如 q0-q3）这些变量在 64 次转换操作中持续使用，优化编译器不会频繁地将它们移入/移出寄存器即使在函数边界如从 FF_SIMD 到 GG_SIMD 的转换中，也保持同样的寄存器分配。

（1）首先他会对代码中变量的变量存活期进行分析

```
for (int j = 0; j < 16; j++) {
// 临时变量 values 只在循环内有效，编译器会为其分配可重用的寄存器
uint32_t values[4] __attribute__((aligned(16)));
memcpy(&values[0], paddedMessages[0] + offset, 4);
// ...
M[j] = vld1q_u32(values);
}//编译器分析后发现每次循环结束时 values 不再使用，因此相同的寄存器可在下一迭代中复用
```

(2) 然后编译器会进行优化以此来减少内存访问，以 FF_SIMD 为例：未优化版本中每次 (a) = 赋值后可能将结果写入内存再读出。但是在优化版本中编译器识别三个连续操作都使用和修改 (a)，保持在同一寄存器中完成全部运算。并将 F_SIMD((b), (c), (d)) 的结果直接存入临时寄存器，无需写入内存。

```
#define FF_SIMD(a, b, c, d, x, s, ac) { \
(a) = vaddq_u32((a), vaddq_u32(vaddq_u32(F_SIMD((b), (c), (d)), (x)), vdupq_n_u32(ac))); \
(a) = ROTATELEFT_SIMD((a), (s)); \
(a) = vaddq_u32((a), (b)); \
}
```

### 3.1.4 循环优化

MD5 哈希算法中存在两个主要的循环：消息块处理循环和轮转换循环。首先与宏定义展开相似，它会将常量向量提升，即将反复利用的变量变为常量储存起来，到在需要利用的时候再直接利用不需要直接计算，但是在这里它直接将变量地地位进行了提升变为了一个常量。例如 v 编译器会将 vdupq__n__u32(0xd76aa478) 等常量向量提升到循环外。实际生成的汇编代码会预先加载所有常量向量值。

```
// 在每个 FF_SIMD 调用中
FF_SIMD(A, B, C, D, M[0], s11, 0xd76aa478);
// 展开为
(A) = vaddq_u32((A),
      vaddq_u32(vaddq_u32(F_SIMD((B), (C), (D))(M[0])), vdupq_n_u32(0xd76aa478)));
```

### 3.1.5 函数内联

编译器会把所有 64 次 FF/GG/HH/II 都写在一个函数 MD5Hash_SIMD 里，不会再去调用额外的小函数，编译器可以看到整个 “4 轮共 64 步” 计算是一长串指令流。它能跨越 “假想的函数边界” 做死码剔除（常量没有用到就可以删）、公共子表达式消除（对相同参数的 vmvnq_u32(B) 只做一次）、指令调度（交错执行）等。若用很多小函数，分散跳转会浪费 I-cache 和分支预测资源。在代码中，几乎所有核心运算都已经是 “内联” 状态——FF_SIMD/GG_SIMD/…全部是宏，NEON intrinsic 也是 header-only inline，所以编译器在把它们都展开到 MD5Hash_SIMD 后，才能施展前面讲的各种全局优化，使 SIMD 代码异步并行、流水线不空闲，达到极致性能。

### 3.1.6 为什么不编译优化并行速度不能超过串行

首先是 intrinsic 的问题：无优化时每个 intrinsic 被视为普通 C 函数，产生完整的函数调用栈帧、参数传递、返回值处理流程；优化后编译器直接将 intrinsic 映射为对应的单条 NEON 指令。消除全部调用开销。比如在 3.1.2 里讲解使用的代码中：无优化时每次 intrinsic 调用后可能将结果写回内存，然后再读取。优化后 A、B、C、D 始终保持在向量寄存器中，减少 90% 以上的内存访问。

其次在寄存器使用的方面，普通的串行运算会使用通用寄存器（r0-r15），计算得到的值值在无优化时仍可能留在寄存器中；NEON intrinsic 会使用专用向量寄存器（q0-q15），所以无优化时可能频繁写回内存再读取，使得每个操作可能涉及更大数据量的移动。

然后一个原因就是我们设计的流水线在非优化编译的情况下并没有实现满载运行。下面这个宏定义展开代码在无优化时严格按顺序执行，导致流水线停顿和气泡。在优化之后编译器分析指令延迟，交错调度不同操作，保持流水线满载。

```
// 在这个宏展开中有多种不同延迟的指令，这几行代码用来展示不同语句之间存在周期差异
vandq_u32(B, C)            // 逻辑运算 (1-2 周期)
vmvnq_u32(B)               // 位运算 (1 周期)
vaddq_u32(...)             // 加法 (3-4 周期)
vdupq_n_u32(0xd76aa478)    // 常量加载 (1-2 周期)
//这两行代码用来解释优化编译如何使得流水线满载
FF_SIMD(A, B, C, D, M[0], s11, 0xd76aa478);
FF_SIMD(D, A, B, C, M[1], s12, 0xe8c7b756);
```

我们观察代码发现虽然在顺序上第二行代码的执行依赖于第一行代码，但第二次调用的常量加载、M[1]读取、B/C 操作是可以提前进行的。所以编译器会把两次调用的指令交错排列。下面我们用一段伪指令进行解释。

```
// 伪指令序列，展示如何交错执行
vdupq_n_u32(0xd76aa478)                    // 第一个 FF_SIMD 的常量
vdupq_n_u32(0xe8c7b756)                    // 同时，第二个 FF_SIMD 的常量也可提前准备
F_SIMD 的计算 (B,C,D)                       // 第一个 FF_SIMD 的 F 函数
加载 M[1]                                  // 同时预取第二个 FF_SIMD 的消息块
vaddq_u32 操作完成 A 值更新                 // 完成第一个 FF_SIMD
F_SIMD 的计算 (A,B,C)                      // 开始第二个 FF_SIMD 的 F 函数 (使用更新后的 A)
//同时优化编译器会在一条高延迟指令之后立即安排一些不依赖其结果的低延迟指令
vaddq_u32(A, ...)              // 需要 3-4 个周期
vdupq_n_u32(下一个常量)         // 同时开始，不需要等待 vaddq 结果
vmvnq_u32(下一个操作的输入)     // 继续用这些指令填充流水线
```

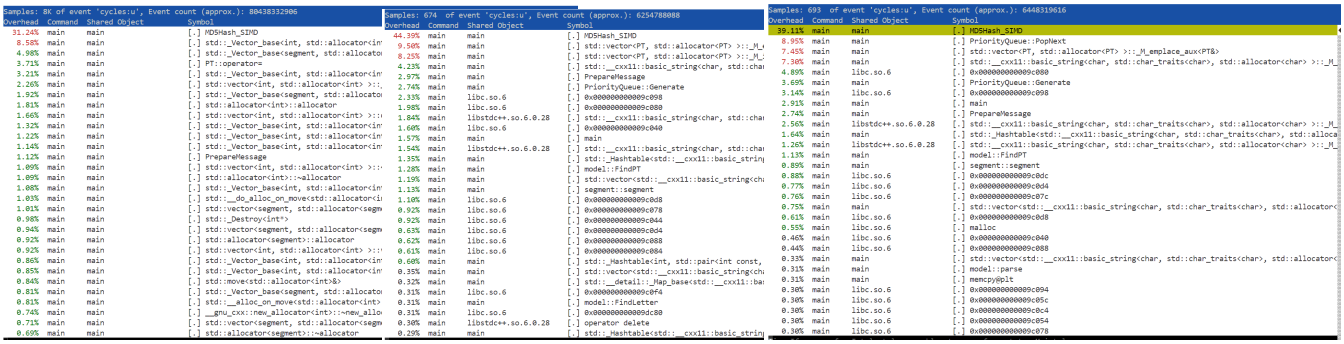
综上所述，这些都是在非编译阶段并行的执行时间不能优于串行时间的可能原因。

### 3.1.7 o1o2 编译优化的区别

在 o1o2 的优化编译中，二者的时间基本上是一样的，但是他们的优化程度还是有略微的不同。o2 一般采用更加激进的优化方式，有可能会把内部的逻辑几乎进行完全的重排以达到流水线的饱和。但是这样的话会使得调试性能的下降，比较适合 release 版本时用到。

### 3.1.8 非优化编译与 O2 优化编译的 perf 结果对比

首先我们可以明显地看到随着编译优化方案的不断激进，底层分配器操作的操作越来越多，而非优化编译的 perf 图中则更多出现的是大量标准库操作。O2 优化成功内联了大量 STL 容器操作，使 STL 相关开销从约 25% 降至不足 10% 程序整体运行时间大幅缩短（样本数从 800 亿 cycles 降至 64 亿 cycles）。但同时 MD5_SIMD 的开销矛盾也变得更加地明显，因对于非编译优化的 stl 开销，o2 明显能够



(a) 非编译优化 perf 图 (b) o1 优化 perf 图 (c) o2 优化 perf 图

图 3.11: 优化编译前后的 perf 对比

更加激进地去打破并减少时间消耗，原本被 STL 操作掩盖的 PriorityQueue 操作 (PopNext/Generate) 成为次要热点。

## 3.2 SSE 指令集实现并行算法

我们之前使用的是一次计算四个 hash 值的方法，接下来我们想探究一次性多一点和少一点操作数对于 hashtime 的影响。因为 NEON 并没有提供一次性 8 个操作数的运算向量，并且如果我们在 arm 平台上使用连续运用两次 4 路运算似乎得到的时间结果是一样的，所以我们选择在 x86 平台上实现 MD5 的二路四路和八路的并行算法。

### 3.2.1 指令的对应关系

首先因为 x86 平台使用的是 SSE 指令集，所以相应的并行算法的指令页相应地要换成 SSE 的指令。如下所示：

```
_mm_and_si128((x), (y))  _mm256_and_si256((x), (y))     x&y
_mm_or_si128((x), (y))   _mm256_or_si256((x), (y))      x|y
_mm_xor_si128((x), _mm_set1_epi32(-1))                  ~x
_mm_set1_epi32(ac)         _mm256_set1_epi32(ac)        填充向量
_mm_xor_si128((x), (y))  _mm256_xor_si256((x), (y))     x^y
_mm_slli_epi32((x), (y)) _mm256_slli_epi32((x), (y))    x<<y
_mm_srli_epi32((x), (y)) _mm256_srli_epi32((x), (y))    x>>y
```

### 3.2.2 运用指令改造相应的宏定义函数

在找到了相应的运算符之后，我们便可以开始改造相应的 F 和 FF 和 ROTATE 函数了。

```
#define F_simd(x, y, z) \
_mm_or_si128( \
    _mm_and_si128((x), (y)), \
```

```
    _mm_and_si128(_mm_xor_si128((x), _mm_set1_epi32(-1)), (z)) \
)
#define ROTATELEFT_simd(num, n) \
_mm_or_si128( \
    _mm_slli_epi32((num), (n)), \
    _mm_srli_epi32((num), 32 - (n)) \
)
#define FF_simd(a, b, c, d, x, s, ac) { \
a = _mm_add_epi32(a, F_simd(b, c, d)); \
a = _mm_add_epi32(a, x); \
a = _mm_add_epi32(a, _mm_set1_epi32(ac)); \
a = ROTATELEFT_simd(a, s); \
a = _mm_add_epi32(a, b); \
}
```

在这里以四路的运算符为例，当我们在实现二路的时候，我们只需要使用四路指令的后 64 位即可。在进行一次 8 个操作数的运算的时候，一般将 128 改为 256 即可。对于主函数的改造直接仿照之前我们在 arm 架构中改造的方式一样更改即可。

### 3.3 探究并行度对效率的影响

在操作数方面我们将串行与二路四路和八路相比较，通过不断扩大每次输入的数据和一次性处理的向量的长度进行计算。同时，因为在非优化编译的情况下我们的 SIMD 算法并不能进行很好地加速，所以我们将非优化编译与 o1o2 编译一起来进行对比，探究在实现加速和未实现加速的情况下，一次处理不同的操作数会得到什么样的结果。下面是我们得到的不同条件下的热力图：

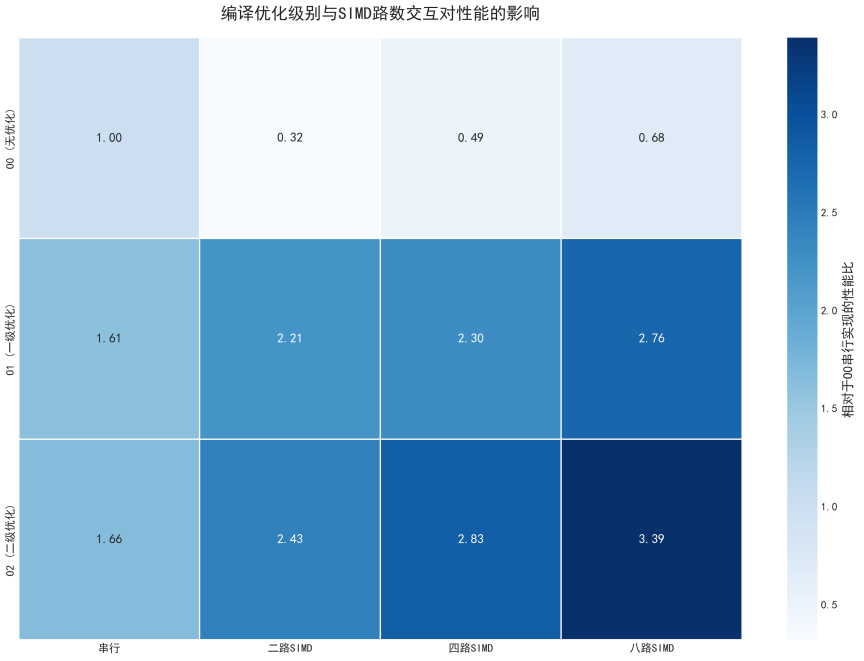



图 3.12: 不同并行度的加速比热力图

### 3.3.1 不同路数之间的优化幅度

我们首先可以直观地从图中看出，随着路数的不断增大，不管是优化编译之前还是优化编译之后的值都是在不断地增大的。之所以没有进行十六路以及更大的路数探究，是因为现代 CPU 支持的 SIMD 寄存器宽度（如 128-bit SSE、256-bit AVX、512-bit AVX-512）直接限制了路数的最大值。而 intel 13 将 AVX-512 的功能去掉了，所以我们仅限于八路及以下的进行探究。

我们首先看总体趋势：随着不优化编译、o1 优化到 o2 优化，路数之间的差距以两倍的速度在扩大，编译优化带来的性能提升也更加显著。但是唯独从串行到二路并行的时候出现了一个非常大的性能跃升，这是从四路到八路也无法企及的。我们来分析其原因。首先阿姆达尔定律可以在一定程度上解释这个现象 Amdahl's Law:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Where:

- $S$: Speedup (加速比)
- $P$: The proportion of the program that can be parallelized (可以并行化的部分比例)
- $N$: The number of parallel processors or SIMD lanes (并行处理器或 SIMD 路数)

当 $N$ 增加时，加速比 $S$ 逐渐逼近一个上限值 $\frac{1}{1-P}$。即

$$\lim_{N\to\infty} S = \frac{1}{1-P}$$

在从串行到二路的阶段中，加速比 $S$ 从串行的 1 提升到接近 2，收益最大。同时从串行到二路，SIMD 指令开始利用硬件的矢量寄存器、流水线和并行计算单元，因此性能提升幅度远超后续路数的扩展。$N$ 提升到 4 或 8 时，尽管并行能力进一步增强，但程序中不可并行化的串行部分开始限制整体性能的提升。因此，从四路到八路的性能提升幅度明显减小，收益递减。

同时还有另一个问题：硬件资源的初始利用最大化。串行计算完全依赖单一处理单元一次处理一个数据块，执行效率非常低。二路并行开启了 SIMD（单指令多数据）的基本能力，使一个指令同时处理两个数据块，大幅度提升了并行计算的吞吐量。因此，从完全串行转变为部分并行的过程中，性能提升的幅度是最大的。SIMD 的矢量寄存器（如 128 位的 SSE 或 AVX）在二路并行时被完全激活。此时，寄存器和流水线的利用率大幅提高，硬件资源的效率接近理想状态。从四路到八路的时候，资源利用开始进入瓶颈。比如：四路并行利用了更宽的寄存器，而八路并行依赖更高端的指令集（如 AVX-512），但寄存器压力、内存对齐、缓存带宽等问题开始限制性能进一步提升。限制了效率的提升。

### 3.3.2 随着路数增加优化编译对于速度的影响

我们继续对热力图进行观察：对于串行来说，o1o2 之间对于程序加速的影响并没有从不优化到优化的那么大的跨度，反而随着路数的增多，o1o2 之间因为编译优化而产生的性能提升越来越大。我们来分析一下原因：

在串行情况下，O1 仅执行基础优化（如寄存器分配优化和简单循环矢量化），性能提升有限。随着路数增加（如从二路到四路），O1 的简单矢量化开始利用 SIMD 的并行能力，但由于其缺乏对复杂循环和内存访问的优化，性能提升有限。但是 O2 的深度矢量化能力与路数增加的趋势相匹配，比如说：从二路到四路：O2 能更好地分解循环、优化数据布局，使得更宽的 SIMD 路数能够被充分利用。从四路到八路：O2 的高级优化（如掩码矢量化和高级指令集支持）进一步放大了路数增加的性能提升。

### 3.3.3 不同并行度的信息处理速度

看完了相对的加速比，我们现在来分析绝对的不同路数之间的信息处理速度。

从折线图可以清晰地看到，随着优化级别的提升（从 O1 到 O2），不同路数的性能差异逐渐扩大：对于二路来说，O1 和 O2 的性能差距较小（4.41M vs. 5.17M）。说明二路 SIMD 的性能提升主要来自基础的矢量化优化（O1 即可实现大部分性能提升）；对于四路，O1 和 O2 的性能差距显著增加（4.89M vs. 6.02M）。说明随着路数增加，O2 的高级优化（如循环展开和内存预取）对性能的贡献更加显著；对于八路，O1 和 O2 的性能差距最大（5.86M vs. 7.21M）。说明在高路数下，O2 的复杂优化（如掩码矢量化和指令调度）被充分发挥，显著提升了性能。综上所述：O1 的优化效果随着路数增加趋于稳定，但 O2 的优化效果随着路数增加逐渐放大。这表明 O2 优化在高路数 SIMD 下的性能潜力远大于 O1。

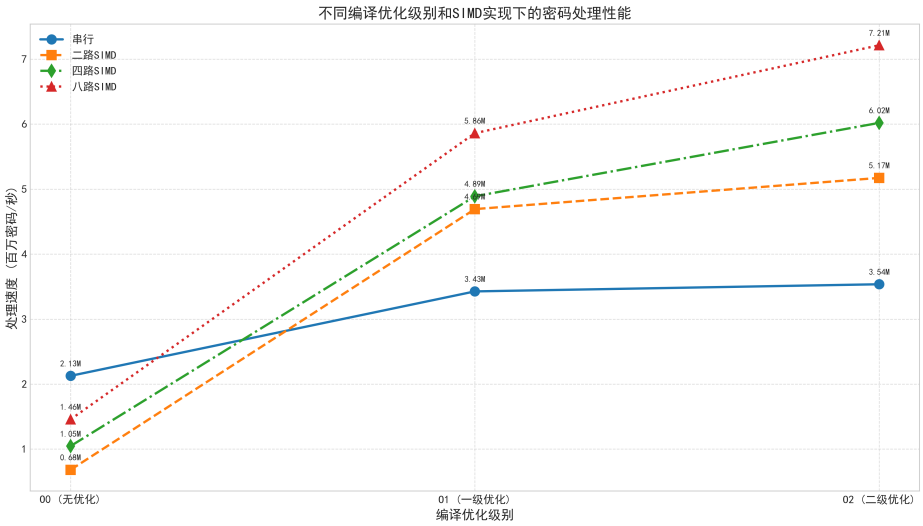



图 3.13: 不同并行度的处理速度折线图

# 4 实验小结

在实验开始之前，我们对实验可能达到的性能做过了理论上的分析。理想情况下，SIMD 版本可以达到近 4 倍的吞吐量。但是实际上再不优化编译的情况下并没有出现我们预想的加速比甚至还出现了变慢的情况，我们在经过反复地尝试加速之后尝试去分析我们失败的原因，并在这其中学习了优化编译的好处和区别，深度理解了 SIMD 运算符在编译器内的处理和内存额外开销随着并行化的变化而变化的现象。

虽然因为继续深入优化已经不属于并行加速的部分而是属于代码重排的部分了便没有继续深入修改，但是我们对并行运行时间变慢的原因有了非常深刻的理解。在日后自己设计并行程序时会注意流水线的满载问题等等，努力避免进行了并行化但是并没有对时间进行加速的情况。

# 5 Github 链接

https://github.com/Mercycoffee12138/guess