



南開大學
Nankai University

计算机学院
并行程序设计报告

SIMD 编程实验

姓名：高景珩

学号：2310648

专业：计算机科学与技术

2025 年 4 月 22 日

目录

1 完整代码地址	2
2 MD5 原理分析	2
2.1 MD5 基础原理	2
2.1.1 填充 (Padding)	2
2.1.2 初始化 MD 缓冲区	2
2.1.3 处理数据块	2
2.1.4 四轮运算	2
2.1.5 更新 MD 缓冲区	3
2.1.6 输出哈希值	3
2.2 MD5 伪代码	3
2.3 MD5 串行代码思路简单分析	4
3 MD5 并行算法	4
3.1 MD5.h 文件的修改	4
3.2 MD5.cpp 文件的修改	5
3.3 main.cpp 文件的修改	5
4 性能分析	6
5 profiling	6
5.1 串行版本的 profiling	6
5.1.1 硬件事件	6
5.2 SIMD 并行版本的 profiling	7
5.2.1 硬件事件	7
5.2.2 性能事件	8
6 深入分析	9
6.1 define 的相关问题	9
6.2 编译选项的差异	10
7 总结	10
7.1 实验成果	10
7.1.1 SIMD 并行化实现	10
7.1.2 性能提升	11
7.2 性能分析	11
7.2.1 串行版本	11
7.2.2 SIMD 并行版本 (宏定义版)	11
7.2.3 编译选项影响	11
7.3 优化效果与意义	11
7.4 不足与缺点	12
7.5 结论	12

1 完整代码地址

以下是最终完整的代码仓库: <https://github.com/crystalsugarhawthorn/Parallel-computing/tree/main/2.SIMD%E5%B9%B6%E8%A1%8C%E7%BC%96%E7%A8%8B/%E4%BB%A3%E7%A0%81>

2 MD5 原理分析

在 SIMD 实验部分, 选择对在口令猜测中使用到的 MD5 进行并行化。虽然, 单轮次的 MD5 运算具有前后一致性与依赖性无法使用并行化加速运算, 但是 MD5 的运算在数据上是高度对齐的, 并且运算过程具有很高的确定性, 因此设计程序一次对多条口令产生哈希值。

2.1 MD5 基础原理

MD5 (Message-Digest Algorithm 5) 是一种常用的哈希函数, 用于将任意长度的输入数据转换为一个固定长度的 128 位 (16 字节) 哈希值。它的主要目的是验证数据的完整性, 通过生成并比较哈希值来检测数据是否被篡改。下面是 MD5 的工作原理分步讲解:

2.1.1 填充 (Padding)

MD5 首先对输入数据进行填充, 使其长度满足处理要求。

填充规则:

在数据末尾添加一个 ‘1’ 位。

接着添加若干个 ‘0’ 位, 直到数据长度对 512 取模等于 448 (即 $\text{长度} \% 512 = 448$)。

最后附加一个 64 位的二进制数, 表示原始数据的长度 (以位为单位)。

2.1.2 初始化 MD 缓冲区

MD5 使用四个 32 位寄存器 (A, B, C, D) 来存储中间和最终的哈希值。这些寄存器有固定的初始值:

- $A = 0x67452301$
- $B = 0xEFCDAB89$
- $C = 0x98BADCFE$
- $D = 0x10325476$

2.1.3 处理数据块

将填充后的数据分成 512 位 (64 字节) 的块。每个 512 位块再细分为 16 个 32 位字。对每个块执行四轮运算, 每轮包含 16 次操作。

2.1.4 四轮运算

每轮运算使用一个特定的非线性函数:

- 第一轮: $F(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$

- 第二轮: $G(B, C, D) = (B \wedge D) \vee (C \wedge \neg D)$
- 第三轮: $H(B, C, D) = B \oplus C \oplus D$
- 第四轮: $I(B, C, D) = C \oplus (B \vee \neg D)$

每轮结合数据块中的字和一个预定义的常量表进行位运算，并通过循环左移位更新寄存器。

2.1.5 更新 MD 缓冲区

处理完一个数据块后，将 A, B, C, D 与处理前的值相加，更新缓冲区。对所有数据块重复此过程。

2.1.6 输出哈希值

最终哈希值是 A, B, C, D 四个寄存器的组合，按小端字节序拼接成 128 位输出。值得注意的是，尽管 MD5 设计精巧，但由于已发现其存在碰撞漏洞，不再推荐用于安全性要求高的场景（如密码存储或数字签名）。

2.2 MD5 伪代码

Algorithm 1 MD5 哈希算法伪代码

```

1: 初始化常量:
2:  $a_0 \leftarrow 0x67452301$ 
3:  $b_0 \leftarrow 0xefcdab89$ 
4:  $c_0 \leftarrow 0x98badcfe$ 
5:  $d_0 \leftarrow 0x10325476$ 
6:
7: 填充消息:
8: 将消息  $m$  填充至长度  $\equiv 448 \pmod{512}$ 
9: 在消息末尾附加原始消息长度 (64 位)
10:
11: 处理每个 512 位块:
12: for 每个 512 位块  $M$  do
13:   初始化变量:  $A \leftarrow a_0, B \leftarrow b_0, C \leftarrow c_0, D \leftarrow d_0$ 
14:   for  $i = 0$  to 63 do
15:     if  $0 \leq i \leq 15$  then
16:        $F \leftarrow (B \wedge C) \vee (\neg B \wedge D)$ 
17:        $g \leftarrow i$ 
18:     else if  $16 \leq i \leq 31$  then
19:        $F \leftarrow (D \wedge B) \vee (\neg D \wedge C)$ 
20:        $g \leftarrow (5 \cdot i + 1) \bmod 16$ 
21:     else if  $32 \leq i \leq 47$  then
22:        $F \leftarrow B \oplus C \oplus D$ 
23:        $g \leftarrow (3 \cdot i + 5) \bmod 16$ 
24:     else
25:        $F \leftarrow C \oplus (B \vee \neg D)$ 

```

```

26:       $g \leftarrow (7 \cdot i) \bmod 16$ 
27:      end if
28:       $temp \leftarrow D$ 
29:       $D \leftarrow C$ 
30:       $C \leftarrow B$ 
31:       $B \leftarrow B + \text{rotate\_left}(A + F + K[i] + M[g], s[i])$ 
32:       $A \leftarrow temp$ 
33:  end for
34:  更新哈希值:  $a_0 \leftarrow a_0 + A, b_0 \leftarrow b_0 + B, c_0 \leftarrow c_0 + C, d_0 \leftarrow d_0 + D$ 
35: end for
36: 输出:  $a_0 \parallel b_0 \parallel c_0 \parallel d_0$ 

```

2.3 MD5 串行代码思路简单分析

在 main.cpp 文件中, 猜测口令数量达到一定程度时就对这些口令进行 MD5 哈希处理。

首先将口令内容传入函数中进行补齐操作。首先将输入字符串转换为字节数组, 并计算其原始比特长度。随后确定需要填充的比特数, 使得填充后的总长度模 512 余 448。“512”是因为哈希算法(如 SHA-1、SHA-256、MD5)通常以 512 位(64 字节)为基本处理单元。输入消息需要被分割为若干个完整的 512 位块, 每个块按顺序参与哈希计算。“448”是因为字节数组的最后 64 位用于存放原始消息的比特长度。

填充操作从原始数据末尾开始, 首先写入一个字节 0x80 (二进制 10000000), 作为填充起始标记。随后填充全零字节直至总长度满足模 512 余 448 的条件。若原始数据恰好对齐到 448 比特位, 则填充 512 比特的全零内容以保证存在填充段。最后, 将原始消息的比特长度转换为 64 位无符号整数, 按照小端字节序写入末尾 8 个字节。这一步骤确保处理后的数据总长度为 512 比特的整数倍, 同时完整记录原始信息长度, 为后续的分块处理提供标准化的输入格式。

3 MD5 并行算法

因为 MD5 的计算都是从整顿的 512bit 平均分块进行计算, 因此, 同时传入多个口令进行计算是完全可行的。

3.1 MD5.h 文件的修改

首先修改 MD5 定义好的 F, G, H, I 宏定义为 SIMD 并行函数, 利用 uint32x4_t 数据类型, 同时运算 4 个 32 位的数据(即可同时处理 4 条口令)。然后同样修改 ROTATELEFT, FF, GG, HH, II 函数, 修改后少部分示例代码如下(F, ROTATELEFT, FF):

Algorithm 2 F_neon 函数

```

1: function F__NEON(x, y, z)
2:   term1  $\leftarrow$  VANDQ_U32(x, y)                                ▷ 计算 x & y
3:   temp  $\leftarrow$  VMVNQ_U32(x)                                     ▷ 计算 x
4:   term2  $\leftarrow$  VANDQ_U32(temp, z)                             ▷ 计算 x & z
5:   return VORRQ_U32(term1, term2)                             ▷ 返回 term1 | term2
6: end function

```

Algorithm 3 ROTATELEFT_NEON 函数

```

1: function ROTATELEFT_NEON(vec, n)
2:   left  $\leftarrow$  VSHLQ_N_U32(vec, n)                                ▷ 左移 n 位
3:   right  $\leftarrow$  VSHRQ_N_U32(vec, 32 - n)                          ▷ 右移补位
4:   return VORRQ_U32(left, right)                                ▷ 返回 left | right
5: end function

```

Algorithm 4 FF_NEON 函数

```

1: function FF_NEON(a, b, c, d, x, s, ac)
2:   F_val  $\leftarrow$  F_NEON(b, c, d)                                ▷ 计算 F(b, c, d)
3:   add_term  $\leftarrow$  VADDQ_U32(F_val, x)                        ▷ F + x
4:   temp_ac  $\leftarrow$  VDUPQ_N_U32(ac)                            ▷ 复制 ac
5:   add_term  $\leftarrow$  VADDQ_U32(add_term, temp_ac)              ▷ F + x + ac
6:   a  $\leftarrow$  VADDQ_U32(a, add_term)                            ▷ a += (F + x + ac)
7:   a  $\leftarrow$  ROTATELEFT_NEON(a, s)                            ▷ a 循环左移 s 位
8:   a  $\leftarrow$  VADDQ_U32(a, b)                                    ▷ a += b
9: end function

```

3.2 MD5.cpp 文件的修改

因为现在的头文件函数支持的是 4 组数据同时计算，也就是最大可以处理 4 个口令，因此将 cpp 文件内的 MD5Hash 函数的参数可以改为 4 个字符串用于传递 4 组代码，再额外传递一个 bit32 型的二维数组 states 同时记录 4 组数据的 state。之后的思路和串行代码大致一致，将 4 个口令传入 StringProcess 函数中得到对应的值即可。与串行不同的点是，因为我们前面将函数改为支持 SIMD 的数据类型 uint32x4_t，会和这里的 bit32 型不兼容，因此需要更改为对应的数据类型，例如 vdupq_n_u32，uint32x4_t。

此外如果四个口令长度不同，则在后面补 0 使得得到的每个口令对应的块数相同。

3.3 main.cpp 文件的修改

在 main 函数的 hash 部分，只需要一次传四个参数即可，具体思路与伪代码如下：

Algorithm 5 密码哈希批量处理

Input: 密码猜测列表 $q.guesses$ (长度为 n)

Output: 所有密码完成 MD5 哈希计算

```

1:  $i \leftarrow 0$ 
2: while  $i + 3 < n$  do                                            ▷ 处理 4 个一组的批次
3:   MD5HASHFOUR( $q.guesses[i]$ ,  $q.guesses[i + 1]$ ,  $q.guesses[i + 2]$ ,  $q.guesses[i + 3]$ )
4:    $i \leftarrow i + 4$ 
5: end while
6: while  $i < n$  do                                                ▷ 处理剩余单个密码
7:   MD5HASH( $q.guesses[i]$ )
8:    $i \leftarrow i + 1$ 
9: end while

```

4 性能分析

将修改前后的代码都使用 O2 编译后运行各 5 次以此来分析性能提升情况，得到以下表格：

实验次数 n	串行 hash_time (s)	SIMD 并行 hash_time (s)
1	3.27492	2.14965
2	2.99845	1.81303
3	2.98392	2.09614
4	3.00465	2.04907
5	2.96811	1.96674

可以得出，串行的 hash_time 的平均值为 3.04601 秒，SIMD 并行 hash_time 的平均值为 2.014926 秒，提升了大约 34%。

5 profiling

本次实验在华为鲲鹏服务器上用 perf 进行性能的测试，选择对比改进前后的硬件事件与性能事件。

5.1 串行版本的 profiling

5.1.1 硬件事件

首先通过命令 `g++ main.cpp train.cpp guessing.cpp md5.cpp -o main -O2` 进行编译，之后在终端中运行 `perf stat ./main`，运行后得到的结果如下图所示：

```
Performance counter stats for './main':

      1.77 msec task-clock:u          #    0.706 CPUs utilized
           0      context-switches:u    #    0.000 K/sec
           0      cpu-migrations:u      #    0.000 K/sec
        105      page-faults:u          #    0.059 M/sec
    2,098,307      cycles:u              #    1.186 GHz
    1,777,435      instructions:u        #    0.85 insn per cycle
<not supported>      branches:u
        16,550      branch-misses:u

      0.002505325 seconds time elapsed

      0.002503000 seconds user
      0.000000000 seconds sys
```

图 5.1: 串行硬件事件总览

为了具体分析我们感兴趣的部分，我们调用以下指令 `perf stat -e L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses,cache-references,cache-misses ./main` 来测定 L1 与 LLC 的 miss 与 hit 情况，得到结果如下：

```

Performance counter stats for './main':

    648,936      L1-dcache-loads:u
    11,291      L1-dcache-load-misses:u    #    1.74% of all L1-dcache accesses
    11,888      LLC-loads:u
     4,379      LLC-load-misses:u          #   36.84% of all LL-cache accesses
    648,936      cache-references:u
    11,291      cache-misses:u            #    1.740 % of all cache refs

    0.002266124 seconds time elapsed

    0.002277000 seconds user
    0.000000000 seconds sys

```

图 5.2: 串行 L1 与 LLC 分析

从得到的性能数据来看：

- L1 缓存未命中率为 1.74% (理想 <5%) L1 缓存访问效率良好，数据局部性较高，无明显优化紧迫性。
- 36.84% (理想值 <10%) LLC 未命中率极高，表明程序频繁访问主存，内存访问延迟成为瓶颈。
- 总缓存未命中率为 1.74%，缓存未命中主要由 LLC 未命中贡献，需重点优化末级缓存效率。
- 用户态占比 100%)，程序为 CPU 密集型，无明显 I/O 或系统调用开销。

为了探寻 L1 的具体问题，可以用以下命令 `perf record -e L1-dcache-load-misses` 继续分析，定位高频 L1 未命中的代码段（平台的权限不足，不能分析 CPU 事件）。

5.2 SIMD 并行版本的 profiling

5.2.1 硬件事件

首先通过命令 `g++ main.cpp train.cpp guessing.cpp md5.cpp -o main -O2` 进行编译，之后在终端中运行 `perf stat ./main`，运行后得到的结果如下图所示：

```

Performance counter stats for './main':

    36,345.43 msec task-clock:u          #    1.000 CPUs utilized
           0      context-switches:u    #    0.000 K/sec
           0      cpu-migrations:u      #    0.000 K/sec
     92,851      page-faults:u          #    0.003 M/sec
  92,354,909,914 cycles:u                #    2.541 GHz
  49,688,875,812 instructions:u         #    0.54 insn per cycle
<not supported> branches:u
   699,366,198 branch-misses:u

    36.352012543 seconds time elapsed

    35.925181000 seconds user
     0.343509000 seconds sys

```

图 5.3: 硬件事件总览

为了具体分析我们感兴趣的部分，我们调用以下指令 `perf stat -e L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses,cache-references,cache-misses ./main` 来测定 L1 与 LLC 的 miss 与 hit 情况，得到结果如下：


```

Performance counter stats for './main':

    16,321,950,675      L1-dcache-loads:u
      950,366,066      L1-dcache-load-misses:u    #    5.82% of all L1-dcache accesses
    1,696,105,516      LLC-loads:u
      132,679,812      LLC-load-misses:u          #    7.82% of all LL-cache accesses
    16,321,950,675      cache-references:u
      950,366,066      cache-misses:u            #    5.823 % of all cache refs

    35.611722077 seconds time elapsed

    35.188769000 seconds user
     0.327643000 seconds sys

```

图 5.4: SIMD_L1_LLC

从得到的性能数据来看：

- L1 缓存未命中率为 5.82%，接近临界值（理想 <5%）。
- LLC（L3）缓存未命中率为 7.82%，相对合理，但若进一步降低可减少主存访问延迟，相比串行代码已经大幅提升。
- 总缓存未命中率 5.82%，主要由 L1 未命中贡献，说明瓶颈集中在 L1 层级。
- 用户态时间占比 98.8%，程序为 CPU 密集型，无明显 I/O 或系统调用开销。

为了探寻 L1 的具体问题，可以用以下命令 `perf record -e L1-dcache-load-misses` 继续分析，定位高频 L1 未命中的代码段（平台的权限不足，不能分析 CPU 事件）。

5.2.2 性能事件

首先调用命令 `perf record -g ./main` 生成 `perf_data` 文件，然后用命令 `perf report` 分析记录的性能事件。

Samples: 9K of event 'cycles:u', Event count (approx.): 9067753581

Children	Self	Command	Shared Object	Symbol
+ 100.00%	0.00%	main	main	[.] _start
+ 100.00%	0.00%	main	libc.so.6	[.] __libc_start_main
+ 100.00%	0.00%	main	libc.so.6	[.] 0x0000ffffa46b0000
+ 100.00%	0.15%	main	main	[.] main
+ 53.02%	53.02%	main	main	[.] model::FindPT
+ 52.91%	0.01%	main	main	[.] model::train
+ 18.40%	0.00%	main	main	[.] model::order
+ 19.07%	18.07%	main	main	[.] std::_Hashtable<std::_cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, std::pair<std::_cxx11::
+ 17.47%	0.59%	main	main	[.] segment::order
+ 16.47%	1.10%	main	main	[.] model::parse
+ 14.26%	0.86%	main	main	[.] std::_detail::_Map_base<std::_cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, std::pair<std::
+ 12.65%	1.76%	main	main	[.] std::_IntrusiveList<gnu_cxx::__normal_iterator<std::_cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>
+ 10.29%	0.04%	main	main	[.] segment::insert
+ 5.47%	3.72%	main	main	[.] MD5HashFour
+ 3.09%	2.47%	main	main	[.] std::_detail::_Map_base<int, std::pair<int const, int>, std::allocator<std::pair<int const, int>>, std::_detail::_Sele
+ 2.89%	2.89%	main	main	[.] std::_detail::_Map_base<int, std::pair<int const, int>, std::allocator<std::pair<int const, int>>, std::_detail::_Sele
+ 2.88%	0.70%	main	libc.so.6	[.] cfree
+ 2.59%	0.00%	main	main	[.] PriorityQueue::~PriorityQueue
+ 2.18%	0.54%	main	main	[.] std::vector<segment, std::allocator<segment>>::~vector
+ 1.65%	0.35%	main	main	[.] PriorityQueue::PopNext
+ 1.44%	0.00%	main	libc.so.6	[.] 0x0000ffffa4720000
+ 1.23%	1.23%	main	main	[.] segment::segment
+ 1.23%	0.29%	main	libstdc++.so.6.0.28	[.] std::operator<><char, std::char_traits<char>, std::allocator<char>>
+ 1.22%	0.03%	main	main	[.] std::_unguarded_linear_insert<gnu_cxx::__normal_iterator<std::_cxx11::basic_string<char, std::char_traits<char>, std::
+ 1.16%	0.19%	main	libstdc++.so.6.0.28	[.] operator new
+ 1.02%	0.17%	main	main	[.] PriorityQueue::Generate
+ 1.01%	0.17%	main	main	[.] PT::PT
+ 0.90%	0.00%	main	main	[.] std::_Hashtable<std::_cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, std::pair<std::_cxx11::
+ 0.79%	0.23%	main	main	[.] std::vector<segment, std::allocator<segment>>::_M_realloc_insert<segment&>
+ 0.73%	0.73%	main	libc.so.6	[.] malloc
+ 0.64%	0.00%	main	libc.so.6	[.] 0x0000ffffa471ebac
+ 0.56%	0.31%	main	main	[.] StringProcess
+ 0.52%	0.18%	main	main	[.] std::_Hashtable<int, std::pair<int const, int>, std::allocator<std::pair<int const, int>>, std::_detail::_Selectlist, st
+ 0.51%	0.45%	main	main	[.] PT::~PT
+ 0.48%	0.48%	main	libstdc++.so.6.0.28	[.] std::_Hash_bytes
+ 0.47%	0.47%	main	libc.so.6	[.] 0x0000000000000000
+ 0.47%	0.00%	main	libc.so.6	[.] 0x0000ffffa471e074
+ 0.47%	0.00%	main	libc.so.6	[.] 0x0000000000000000
+ 0.47%	0.00%	main	libc.so.6	[.] 0x0000ffffa471e008
+ 0.45%	0.00%	main	libc.so.6	[.] 0x0000ffffa471e074
+ 0.45%	0.00%	main	main	[.] std::vector<std::_cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, std::allocator<std::_cxx11::
+ 0.43%	0.42%	main	main	[.] std::_cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::_M_construct<char>
+ 0.42%	0.04%	main	libstdc++.so.6.0.28	[.] std::use_facet<std::ctypechar>
+ 0.41%	0.27%	main	main	[.] segment::~segment
+ 0.41%	0.41%	main	libstdc++.so.6.0.28	[.] std::istream::sentry::sentry
+ 0.40%	0.40%	main	libc.so.6	[.] 0x0000000000000000

Tip: Show current config key-value pairs: perf config --list

图 5.5: 性能分析总览

进一步分析 MD5 的 CPI, 运用代码 `perf record -e cycles,instructions -g ./main` 和 `perf report` 进行计算, cycles 总数为 92452926582, MD5HashFour 占据 3.76%, instruments 总数为 49214155029, MD5HashFour 占据 12.44%, 即 CPI 总数为 0.5678, 并行化程度很高, 效率优。

6 深入分析

6.1 define 的相关问题

在 MD5.h 的串行代码中, F、G、H、I、ROTATELEFT、FF、GG、HH、II 这九个过程都使用宏定义, 例如:

Listing 1: F 宏定义

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

而转化后的并行代码为 inline 函数形式, 例如:

Listing 2: F 并行函数

```
1 inline uint32x4_t F_neon(uint32x4_t x, uint32x4_t y, uint32x4_t z) {
2     uint32x4_t term1 = vandq_u32(x, y);           // x & y
3     uint32x4_t term2 = vandq_u32(vmvnq_u32(x), z); // ~x & z
4     return vorrq_u32(term1, term2);                // 按位或
5 }
```

为了解释 define 没有返回值的问题, 需要从以下角度回答这个问题:

宏通过简单的文本替换展开到调用位置, 当调用 `F(x, y, z)` 时预处理器会直接将其替换为宏定义的代码块。由于宏没有函数栈帧, 修改的变量 (如 `x`) 直接作用于调用处的上下文。这样的好处是可以避免频繁的返回值传递, 提升效率。

Listing 3: F 宏定义并行函数

```
1 #define F_NEON(x, y, z) \
2     vorrq_u32(vandq_u32((x), (y)), vandq_u32(vmvnq_u32((x)), (z)))
```

因此, 在分析后可以继续改进 SIMD 代码用 define 宏定义来完成, 将 SIMD 代码与 SIMD (宏定义版) 测试 5 次, 进行 `hash_time` 的性能分析:

实验次数 n	SIMD 并行 hash_time	SIMD 并行 (宏定义版) hash_time
1	1.97033	1.95974
2	2.07477	2.15718
3	2.0784	1.95833
4	2.01473	1.9412
5	2.24994	1.9538

进过简单计算可以得出, 宏定义在 SIMD 并行的基础上又节省了约 4% 的时间。使用 `perf stat -e L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses,cache-references,cache-misses ./main` 命令后也得到了以下结果:

```

Performance counter stats for './main':

    16,239,104,603      L1-dcache-loads:u
    929,953,268        L1-dcache-load-misses:u    #    5.73% of all L1-dcache accesses
    1,664,933,309      LLC-loads:u
    44,283,550         LLC-load-misses:u          #    2.66% of all LL-cache accesses
    16,239,104,603      cache-references:u
    929,953,268        cache-misses:u            #    5.727 % of all cache refs

    33.851360756 seconds time elapsed

    33.320428000 seconds user
    0.456872000 seconds sys

```

图 6.6: SIMD(define)_L1_LLC

结果数据说明：

- L1 数据缓存：5.73% 未命中（访问 16.2 亿次，未命中 9.29 亿次）
- 末级缓存（LLC/L3）：2.66% 未命中（访问 16.6 亿次，未命中 0.44 亿次）

L1 访问量与 LLC 访问量相近（16.2 亿 vs 16.6 亿），说明大量请求穿透 L1 直达 LLC，但 LLC 未命中率较低，表明大部分数据最终仍在 LLC 中命中。即 define 的改进降低了 LLC 的 miss 值。

6.2 编译选项的差异

在编译选项中，常用且安全的编译等级一般为-O0（不优化）、-O1（基本优化）、-O2（推荐优化），他们之间的区别大致如下：

- -O0 关闭所有优化，保留调试信息，代码按原始逻辑逐行翻译。即使手动编写了 NEON 指令，编译器可能不会优化指令调度或寄存器分配，导致 SIMD 指令的吞吐量未完全发挥。
- -O1 启用轻量级优化，如常量传播、简单循环优化、冗余代码消除。编译器可能优化指令顺序，但不会修改手动编写的 SIMD 逻辑。
- -O2 启用绝大多数安全优化，包括循环展开、函数内联、指令调度等。通过指令重排、循环展开、寄存器优化，最大化 SIMD 指令吞吐量。

因此分别用-O0、-O1、-O2 编译代码，加速比从小到大逐级递增。

7 总结

本次实验针对口令猜测场景下的 MD5 哈希算法进行了 ARM NEON SIMD 并行优化，成功实现了对四个口令的同时处理，并在性能上取得了显著提升。通过对 MD5 核心函数（F、G、H、I、ROTATELEFT、FF、GG、HH、II）的向量化改造，利用 NEON 的 128 位向量类型 uint32x4_t，实验将串行计算扩展为并行计算，显著降低了计算时间。以下是对实验成果、性能分析及优化效果的总结。

7.1 实验成果

7.1.1 SIMD 并行化实现

通过将串行宏定义（如 F、ROTATELEFT）改写为 NEON 指令支持的宏（如 F_NEON、ROTATELEFT_NEON）和对应的内联函数，实现了同时对四个口令的并行哈希计算。

修改了 MD5HashFour 函数，支持同时处理四个输入字符串，并通过 uint32x4_t 类型兼容 NEON 向量作，确保数据类型一致性。

在 main.cpp 中调整了批量处理逻辑，通过 MD5HashFour 一次性处理四条口令，剩余口令由 MD5Hash 串行处理，兼顾效率与灵活性。

7.1.2 性能提升

性能测试结果表明，串行版本的平均 hash_time 为 3.04601 秒，而 SIMD 并行版本平均为 2.014926 秒，性能提升约 34%。这一提升源于 NEON 并行处理四个口令的能力，充分利用了向量运算的吞吐量。

进一步优化中，使用 #define 形式的 NEON 宏（如 F_NEON）替代内联函数（如 F_neon），平均 hash_time 再次降低，相较内联函数版本再提升约 4%。这表明宏定义通过减少函数调用开销和优化指令展开，进一步提高了性能。

7.2 性能分析

通过 perf 工具对串行版本和 SIMD 并行版本（宏定义版）的性能进行了深入分析，揭示了优化的具体效果。

7.2.1 串行版本

L1 缓存未命中率为 1.74%（理想值 <5%），显示良好的数据局部性，但 LLC 未命中率高达 36.84%（理想值 <10%），表明频繁访问主存是主要瓶颈。

指令数为 1777435，周期数为 2098307，CPI（每指令周期数）较高，反映串行计算的低效。

7.2.2 SIMD 并行版本（宏定义版）

L1 缓存未命中率略升至 5.73%，可能因并行处理增加了数据访问量，但仍在可接受范围内。LLC 未命中率显著降低至 2.66%，表明优化后的数据访问模式更高效，大部分请求在 LLC 中命中，减少了主存访问。

总周期数为 92452926582，指令数为 49214155029，CPI 为 0.5678，远低于串行版本，反映了 NEON 指令的高效执行。

MD5HashFour 函数在指令数中占比 12.44%，周期数占比 3.76%，表明并行化部分是性能提升的核心。

7.2.3 编译选项影响

使用 -O2 编译选项通过指令重排、函数内联和寄存器优化，最大化了 NEON 指令的吞吐量。相比 -O0 和 -O1，-O2 显著提升了加速比，验证了编译器优化对 SIMD 性能的重要性。

7.3 优化效果与意义

性能提升：SIMD 并行化将 MD5 哈希计算的平均耗时从 3.04601 秒降低到 1.93405 秒（宏定义版），总性能提升约 36.5%。这对于口令猜测等高吞吐量场景具有重要意义。

缓存效率：通过 NEON 向量化和宏定义优化，LLC 未命中率从 36.84% 降至 2.66%，显著减少了内存访问开销，提升了数据局部性。

代码优化：宏定义相较内联函数进一步减少了约 4% 的计算时间，验证了在 SIMD 场景下宏展开对指令效率的积极影响。

7.4 不足与缺点

尽管取得了显著优化效果，实验仍存在不足，最明显的是 L1 缓存未命中率略高。SIMD 版本的 L1 未命中率（5.73%）略高于串行版本，可能因并行处理增加了数据访问复杂性。

因此，未来可以从以下角度继续分析：

- 指令调度优化：分析轮次间数据依赖，调整 NEON 指令顺序，减少流水线停顿。例如修改的完全版代码在 FF、GG、HH、II 函数中简单修改了指令顺序。
- 内存访问优化：实现交错数据布局，结合预加载指令（如 PLD）提升缓存命中率。
- 批处理扩展：结合多线程或更多向量寄存器，探索处理更多口令的可能性。
- 性能分析：使用更细粒度的 perf 分析（如热点函数定位），进一步定位瓶颈。

7.5 结论

本次实验通过 ARM NEON SIMD 技术成功优化了 MD5 哈希算法，实现了对口令猜测场景的高效并行处理。性能测试和 perf 分析表明，SIMD 并行化结合宏定义优化显著提升了计算效率，降低了缓存未命中率，验证了 NEON 在密码学算法优化中的潜力。实验成果为后续 SIMD 优化提供了宝贵经验，同时也指明了进一步优化的方向，为高性能计算研究奠定了基础。