



南開大學
Nankai University

计算机学院

并行程序设计第三次实验报告

口令猜测 MD5 哈希算法并行化

姓名：葛明宇

学号：2312388

专业：计算机科学与技术

2025 年 4 月 29 日

目录

1 代码仓库	2
2 基础要求	2
2.1 算法设计	2
2.2 ARM 平台并行实现	2
2.2.1 md5_neon.h	2
2.2.2 md5_neon.cpp 文件	2
2.2.3 main_neon.cpp 文件	3
2.3 正确性验证	4
3 进阶要求	5
3.1 性能测试	5
3.1.1 串行版本	5
3.1.2 NEON 并行版本	6
3.1.3 编译选项对加速比的影响	6
3.1.4 delete 对加速比的影响	7
3.1.5 单次运算并行度对加速比影响	8
3.2 x86 设备并行实现	8
3.2.1 md5_sse.h	8
3.2.2 md5_sse.cpp 文件	9
3.2.3 main_sse.cpp 文件	10
3.2.4 正确性验证	11
3.2.5 性能测试	11
3.3 profiling	12
4 思考	13
4.1 超过 64 字节的长口令的处理	13
4.2 WSL2 上运行 NEON 版本	13
4.3 优化方向	13

1 代码仓库

并行度为 2、并行度为 4 的 NEON 版本代码和并行度为 4 的 SSE 版本代码已经上传至 [Gitee](#)

2 基础要求

2.1 算法设计

问题：对于口令猜测中 MD5 哈希算法实现 SIMD 并行化。

算法：MD5 哈希算法本身难以实现并行化，但是可以采用多条口令并行执行 MD5 哈希的操作。

2.2 ARM 平台并行实现

2.2.1 md5_neon.h

(1) 基于对 ARM 架构官方文档的系统梳理 [2]，实现了 MD5 核心逻辑函数（F、G、H、I）、算术移位及其组合操作（FF、GG、HH、II）的 SIMD 指令级优化重构，转为 NEON 指令集的函数，实现了基于 NEON 指令的向量化处理技术。以 F、FF 和 ROTATELEFT 为例，其 NEON 指令实现如下：

```

1  #define F_NEON(x, y, z) vorrq_u32(vandq_u32(x, y), vandq_u32(vmvnq_u32(x), z))
2
3
4  inline uint32x4_t ROTATELEFT_NEON(uint32x4_t num, int n) {
5      return vorrq_u32(vshlq_n_u32(num, n), vshrq_n_u32(num, 32 - n));
6  }
7
8  #define FF_NEON(a, b, c, d, x, s, ac) { \
9      a = vaddq_u32(a, vaddq_u32(F_NEON(b, c, d), vaddq_u32(x, vdupq_n_u32(ac)))); \
10     a = ROTATELEFT_NEON(a, s); \
11     a = vaddq_u32(a, b); \
12 }
```

类似的，我们可以得到 G、H、I 及 GG、HH、II 几个函数的 SIMD 并行重构。使用 #define 定义以及 inline 函数，编译时会在可执行文件中展开，减少函数调用栈的开销，减少不必要的时间浪费。

(2) 同时，在接口上 MD5Hash_NEON 的 inputs 参数类型由 string 改为 vector<string>, state 参数类型由 uint32_t* 修改为 uint32x4_t*:

```

1 void MD5Hash_NEON(vector<std::string>& inputs, uint32x4_t state[4]);
```

2.2.2 md5_neon.cpp 文件

(1) 将 4 个独立消息块的同位 32 位转化为 uint32x4_t 向量寄存器。

```

1 for (int block = 0; block < n_blocks; ++block) {
2     uint32x4_t x[16];
3     for (int k = 0; k < 16; ++k) {
4         uint32_t x_buf[batch] = {0};
5         for (int j = 0; j < batch; ++j) {
```

```

6         x_buf[j] = *reinterpret_cast<uint32_t*>(paddedMessages[j] + block *
7             64+k * 4);
8     }
9     x[k] = vld1q_u32(x_buf);
10 }
11 ...
12 }

```

vld1q_u32 会自动按小端序加载数据，单周期内完成 4 个 32 位的寄存器加载。

由于本次实验处理后的口令均是 64 字节，直接取 4 条口令中第一个口令的块数，这样处理不会出现问题。为了与 md5.cpp 性能对比公平，代码中还是写了 for (int block = 0; block < n_blocks; ++block)。思考模块会讨论如何处理 block 数量不同的情况。

(2) 状态初始化、更新与结果存储

```

1     state[0] = vdupq_n_u32(0x67452301);
2     state[1] = vdupq_n_u32(0xefcdab89);
3     state[2] = vdupq_n_u32(0x98badcfe);
4     state[3] = vdupq_n_u32(0x10325476);
5
6
7     uint32x4_t a = state[0], b = state[1], c = state[2], d = state[3];
8
9     state[0] = vaddq_u32(state[0], a);
10    state[1] = vaddq_u32(state[1], b);
11    state[2] = vaddq_u32(state[2], c);
12    state[3] = vaddq_u32(state[3], d);
13
14    for (int i = 0; i < 4; ++i) {
15        uint8x16_t val_u8 = vreinterpretq_u8_u32(state[i]);
16        val_u8 = vrev32q_u8(val_u8);
17        state[i] = vreinterpretq_u32_u8(val_u8);
18    }

```

使用 vrev32q_u8 在一个 128 位的向量中，按每 32 位为一组对无符号 8 位整数进行反转操作。vreinterpretq_u8_u32 则将 32 位的向量转换为 8 位的向量。

(3) 内存管理 (十分关键)

```

1     for (int j = 0; j < batch; ++j) {
2         delete [] paddedMessages[j];
3     }

```

2.2.3 main_neon.cpp 文件

在修改了 md5_neon.h 和 md5_neon.cpp 之后，main_neon.cpp 中也要相应的进行读取口令的修改。

(1) 每次读取到 4 条口令后进行 MD5Hash_NEON 处理。当最后剩余口令数量不足 4 条时，则直接调用 MD5Hash 进行处理。

main_neon.cpp

```

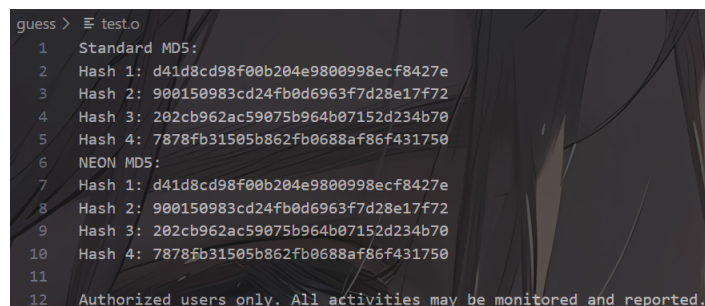
1  . . .
2  vector<string> batch_buffer;
3  uint32x4_t batch_states[4];
4  while (!q.priority.empty()) {
5      . . .
6      if (curr_num > 1000000) {
7          auto start_hash = system_clock::now();
8
9          size_t total = q.guesses.size();
10         size_t int_total=(total/4)*4;
11         size_t left=total-int_total;
12
13         for (size_t i = 0; i < int_total; i += 4)
14         {
15             batch_buffer.clear();
16             for (int j = 0; j < 4; ++j)
17             {
18                 batch_buffer.push_back(q.guesses[i + j]);
19             }
20             MD5Hash_NEON(batch_buffer, batch_states);
21         }
22
23         bit32 state[4];
24         for(int i=0;i<left;i++){
25             MD5Hash(q.guesses[int_total+i], state);
26         }
27         . . .
28     }
29     . . .

```

2.3 正确性验证

使用 NEON 版本的 correctness.cpp 进行 MD5 哈希算法的正确性验证。输入“”、“abc”、“123”、“!q\$%^”四个口令，串行和并行算法得到的 Hash 结果相同。

测试结果如图2.1所示



```

guess > E test.o
1 Standard MD5:
2 Hash 1: d41d8cd98f00b204e9800998ecf8427e
3 Hash 2: 900150983cd24fb0d6963f7d28e17f72
4 Hash 3: 202cb962ac59075b964b07152d234b70
5 Hash 4: 7878fb31505b862fb0688af86f431750
6 NEON MD5:
7 Hash 1: d41d8cd98f00b204e9800998ecf8427e
8 Hash 2: 900150983cd24fb0d6963f7d28e17f72
9 Hash 3: 202cb962ac59075b964b07152d234b70
10 Hash 4: 7878fb31505b862fb0688af86f431750
11
12 Authorized users only. All activities may be monitored and reported.

```

图 2.1: NEON 正确性验证

3 进阶要求

3.1 性能测试

在 ARM 服务器上编译运行串行版本以及 NEON 并行版本，并对比性能。采用单次测试 (一次只运行一次 MD5Hash_NEON)，目的是为了去除噪声的影响。因为只保留 10 次的结果，任一次测试的偏差过大都会导致平均值计算的不准确。所以没有循环测试 10 次取平均值，而是单独测试 10+ 次，保留最集中的 10 个结果，再取平均值。后续的测试方法与上述相同。

3.1.1 串行版本

不进行编译优化 平均 Hash 时间 (s):9.53			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	9.41039	6	9.54412
2	9.34859	7	9.61734
3	9.60302	8	9.66548
4	9.37704	9	9.70196
5	9.40104	10	9.68016
-O1 优化 平均 Hash 时间 (s):3.11			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	2.94983	6	2.99351
2	3.16808	7	3.18598
3	2.97332	8	2.94843
4	3.3091	9	3.27506
5	3.07015	10	3.18166
-O2 优化 平均 Hash 时间 (s):2.94			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	2.92918	6	2.79832
2	3.0233	7	2.95102
3	2.95172	8	3.05432
4	2.93111	9	2.96322
5	2.86973	10	2.97357

表 1: 串行版本测试结果

3.1.2 NEON 并行版本

不进行编译优化 平均 Hash 时间 (s):12.07			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	12.0696	6	12.0162
2	12.1451	7	12.004
3	12.012	8	12.022
4	12.2024	9	12.1435
5	12.0252	10	12.0441
-O1 优化 平均 Hash 时间 (s):1.70			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	1.6981	6	1.70209
2	1.72746	7	1.65551
3	1.70172	8	1.76503
4	1.66454	9	1.73013
5	1.70901	10	1.64058
-O2 优化 平均 Hash 时间 (s):1.65			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	1.66984	6	1.59678
2	1.66039	7	1.59709
3	1.67717	8	1.66778
4	1.66593	9	1.67953
5	1.64698	10	1.66945

表 2: 并行版本测试结果

从上面的测试结果我们可以看到，在不进行编译优化的情况下，NEON 并行版本的 Hash 时间比串行版本的 Hash 时间要长。在-O1 优化和-O2 优化的情况下，NEON 并行版本的 Hash 时间明显比串行版本的 Hash 时间要短，实现了加速，加速比为 1.8 左右。

3.1.3 编译选项对加速比的影响

由表 1、表 2 的数据可得各编译选项下的加速比。

编译选项	串行 Hash 时间 (s)	并行 Hash 时间 (s)	加速比
不进行编译优化	9.53	12.07	0.79
-O1 优化	3.11	1.70	1.83
-O2 优化	2.94	1.65	1.78

表 3: 各编译选项下的加速比

从表 3 数据可以看到，不进行编译优化没能实现加速，-O1 优化时加速比最大，-O2 优化比-O1 优化加速比稍低。

不进行编译优化时，加速比为 0.79:

- (1)NEON 的向量加载被拆分为多次单值加载，而串行代码直接使用单值操作，避免了额外开销。
- (2)NEON 中的向量会被存储在内存之中，频繁读取内存会导致额外开销。

(3) 未优化流水线填充, 无法充分发挥硬件并行性。

结果就是 NEON 的并行计算仍在发生, 但频繁的内存操作和低效调度导致整体性能低下。

-O1 优化时, 加速比为 1.83:

(1) 变量优先存入寄存器, 减少内存访问。

(2) 合并内存操作, 例如 `uint32x4_t state[0] = vdupq_n_u32(0x67452301)` 同时加载四个 `0x67452301`。

(3) 调整指令顺序以填充流水线, NEON 的并行性得以高效体现。

结果是满足了预期加速比, 虽然由于标量转为向量的额外开销, 使得无法实现 4 倍的加速比。但是提升 1 倍的性能也是很可观的。

-O2 优化时, 加速比为 1.78:

(1) 减少循环控制语句的开销。

(2) 调整指令顺序以充分利用 CPU 流水线。

结果是比-O1 优化加速比稍低, neon 实现 simd 并行的主要思想都在-O1 优化时已经得到充分体现, 在-O2 只是进一步提升性能, 但这种提升与 neon 的并行性相关性不大, 对串行和并行都会提升。可以认为这种时间带来的提升主要是来自于小循环的展开。

3.1.4 delete 对加速比的影响

初始编写代码时, 没有对 `paddedMessages` 数组开辟的空间进行释放。在测试程序性能时, Hash time 相对于串行版本的 Hash time 有提升, 但是没有达到预期的加速比。以下是测试结果:

不进行编译优化 平均 Hash 时间 (s):12.56			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	12.4554	6	12.5476
2	12.5595	7	12.8261
3	12.4676	8	12.4384
4	12.4703	9	12.6577
5	12.5263	10	12.6233
-O1 优化 平均 Hash 时间 (s):2.14			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	2.25705	6	2.05112
2	2.28208	7	2.3009
3	2.20695	8	2.01464
4	2.10944	9	1.98965
5	2.20636	10	2.0297
-O2 优化 平均 Hash 时间 (s):2.00			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	1.99259	6	2.04406
2	1.90618	7	2.02123
3	1.98819	8	2.09134
4	1.97304	9	1.93593
5	1.98581	10	2.07033

表 4: 缺失 delete 版本测试结果

从 4 条口令上的处理来看其实并没有什么影响。但是我们的实验是在 10000000 条口令的量级上进行的，不进行 delete 的处理会导致堆碎片化和分配延迟增加以及缓存性能劣化，内存占用过高影响缓存局部性，使得 cache 命中率下降。最终导致整体性能下降。

3.1.5 单次运算并行度对加速比影响

由上述对各编译模式下加速比的分析，我们可以使用 -O1 和 -O2 优化的 NEON 并行版本对单次运算并行度进行测试。

-O1 优化 平均 Hash 时间 (s):2.79			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	2.77157	6	2.88523
2	2.78835	7	2.75786
3	2.78958	8	2.75556
4	2.74106	9	2.76999
5	2.85306	10	2.82282

-O2 优化 平均 Hash 时间 (s):2.71			
次数	Hash 时间 (s)	次数	Hash 时间 (s)
1	2.73565	6	2.69964
2	2.67791	7	2.76622
3	2.71642	8	2.71828
4	2.63492	9	2.73289
5	2.73419	10	2.7019

表 5: NEON2 版本测试结果

编译选项	并行度为 2 加速比	并行度为 4 加速比
-O1 优化	1.14	1.83
-O2 优化	1.08	1.78

表 6: 单次运算并行度的加速比

由于单次处理 8 条口令在 NEON 指令集中很难实现。这是由于一个向量寄存器为 128 位，但是 md5 哈希函数单次要对 32 位进行操作。无法使用 32×8 向量寄存器。而分为两组 32×4 向量寄存器，每组 4 个向量寄存器分别进行运算，也无法实现真正的 8 条口令的并行。所以这里就只进行了 2、4 条口令的实验。从这两个版本的算法也能看出，在 -O1 及以上级别的优化下，当单次运算并行度越高，加速比越高。因为吞吐量带来的巨大性能提升会削弱标量变向量的影响。

3.2 x86 设备并行实现

3.2.1 md5_sse.h

(1) 基于对 x86 架构官方文档的系统梳理 [1]，实现了 MD5 核心逻辑函数（F、G、H、I）、算术移位及其组合操作（FF、GG、HH、II）的 SIMD 指令级优化重构，转为到 SSE 指令集的函数，实现了基于 NEON 指令的向量化处理技术。以 F、FF 和 ROTATELEFT 为例，其 SSE 指令实现如下：

```

1 #define F_SSE(x, y, z) _mm_or_si128( \
2   _mm_and_si128(x, y), \
3   _mm_andnot_si128(x, z) \
4 )
5
6 inline __m128i ROTATELEFT_SSE(__m128i num, int n) {
7     return _mm_or_si128(
8         _mm_slli_epi32(num, n),
9         _mm_srli_epi32(num, 32 - n)
10    );
11 }
12
13 #define FF_SSE(a, b, c, d, x, s, ac) { \
14     a = _mm_add_epi32(a, _mm_add_epi32( \
15         F_SSE(b, c, d), \
16         _mm_add_epi32(x, _mm_set1_epi32(ac)) \
17     )); \
18     a = ROTATELEFT_SSE(a, s); \
19     a = _mm_add_epi32(a, b); \
20 }

```

类似的，我们可以得到 G、H、I 及 GG、HH、II 几个函数的 SIMD 并行重构。使用 #define 定义以及 inline 函数，编译时会在可执行文件中展开，减少函数调用栈的开销，减少不必要的时间浪费。

(2) 同时，在接口上 MD5Hash_SSE 的 inputs 参数类型由 string 改为 vector<string>, state 参数类型由 uint32_t* 修改为 __m128i*:

```

1 void MD5Hash_SSE(const vector<std::string>& inputs, __m128i state[4]);

```

3.2.2 md5_sse.cpp 文件

(1) 将 4 个独立消息块的同位 32 位转化为 uint32x4_t 向量寄存器。

```

1 for (int block = 0; block < n_blocks; ++block) {
2     __m128i x[16];
3     for (int k = 0; k < 16; ++k) {
4         uint32_t x_buf[batch] = {0};
5         for (int j = 0; j < batch; ++j) {
6             x_buf[j] = *reinterpret_cast<uint32_t*>(paddedMessages[j] + block * 64 +
7                 k * 4);
8         }
9         x[k] = _mm_loadu_si128(reinterpret_cast<__m128i*>(x_buf));
10    }
11    ...
12 }

```

_mm_loadu_si128 会自动按小端序加载数据，无需额外操作对数据进行处理，单周期内完成 4 个 32 位的寄存器加载。

由于本次实验处理后的口令均是 64 字节，直接取 4 条口令中第一个口令的块数，这样处理不会出现问题。为了与 md5.cpp 性能对比公平，代码中还是写了 for (int block = 0; block < n_blocks; ++block)。思考模块会讨论如何处理 block 数量不同的情况。

(2) 状态初始化、更新与结果存储

```

1  state[0] = __mm_set1_epi32(0x67452301);
2  state[1] = __mm_set1_epi32(0xefcdab89);
3  state[2] = __mm_set1_epi32(0x98badcfe);
4  state[3] = __mm_set1_epi32(0x10325476);
5
6
7
8  __m128i a = state[0];
9  __m128i b = state[1];
10 __m128i c = state[2];
11 __m128i d = state[3];
12
13 state[0] = __mm_add_epi32(state[0], a);
14 state[1] = __mm_add_epi32(state[1], b);
15 state[2] = __mm_add_epi32(state[2], c);
16 state[3] = __mm_add_epi32(state[3], d);
17
18 for (int i = 0; i < 4; ++i) {
19     __m128i shuffle_mask = __mm_set_epi8(12,13,14,15, 8,9,10,11, 4,5,6,7, 0,1,2,3);
20     state[i] = __mm_shuffle_epi8(state[i], shuffle_mask);
21 }

```

使用 __mm_set_epi8(12,13,14,15, 8,9,10,11, 4,5,6,7, 0,1,2,3) 在一个 128 位的向量中，按每 32 位为一组对无符号 8 位整数进行反转操作。

(3) 内存管理

```

1  for (int j = 0; j < batch; ++j) {
2      delete [] paddedMessages[j];
3  }

```

3.2.3 main_sse.cpp 文件

在修改了 md5_sse.h 和 md5_sse.cpp 之后，main_sse.cpp 中也要相应的进行读取口令的修改。

(1) 每次读取到 4 条口令后进行 MD5Hash_SSE 处理。当最后剩余口令数量不足 4 条时，则直接调用 MD5Hash 进行处理。

main_sse.cpp

```

1  . . .
2  vector<string> batch_buffer;
3  __m128i batch_states[4];
4
5  while (!q.priority.empty()) {
6      . . .

```

```

7   if (curr_num > 1000000) {
8       auto start_hash = system_clock::now();
9
10      size_t total = q.guesses.size();
11      size_t int_total=(total/4)*4;
12      size_t left=total-int_total;
13      // 每次处理 4 个口令
14      for (size_t i = 0; i < int_total; i += 4) {
15          batch_buffer.clear();
16          // 收集 4 个口令
17          for (int j = 0; j < 4; ++j) {
18              batch_buffer.push_back(q.guesses[i + j]);
19          }
20          MD5Hash_SSE(batch_buffer, batch_states);
21      }
22
23      bit32 state[4];
24      for(int i=0;i<left;i++){
25          MD5Hash(q.guesses[int_total+i], state);
26      }
27      . . .
28  }
29  . . .

```

3.2.4 正确性验证

使用 SSE 版本的 correctness.cpp，代码与 NEON 版本的 correctness.cpp 类似。
测试结果如图3.2所示

```

PS D:\PCFG_framework> g++ correctness.cpp md5.cpp md5_sse.cpp -o correctness -msse4 -O2
PS D:\PCFG_framework> ./correctness
Standard MD5:
Hash 1: d41d8cd98f00b204e9800998ecf8427e
Hash 2: 900150983cd24fb0d6963f7d28e17f72
Hash 3: 202cb962ac59075b964b07152d234b70
Hash 4: 7878fb31505b862fb0688af86f431750
SSE MD5:
SSE Hash 1: d41d8cd98f00b204e9800998ecf8427e
SSE Hash 2: 900150983cd24fb0d6963f7d28e17f72
SSE Hash 3: 202cb962ac59075b964b07152d234b70
SSE Hash 4: 7878fb31505b862fb0688af86f431750

```

图 3.2: SSE 正确性测试结果

3.2.5 性能测试

编译选项	串行平均 Hash 时间 (s)	SSE 并行平均 Hash 时间 (s)	加速比
不进行编译优化	4.02	4.43	0.91
-O1 优化	2.17	1.20	1.80
-O2 优化	2.04	1.15	1.77

表 7: x86 系统各编译选项下的加速比

SSE 版本的 md5 并行的性能测试结果与 NEON 版本的 md5 并行的性能测试结果相似。也能进一步验证上述的分析。

3.3 profiling

在本地的 WSL2 上，使用 perf 命令进行性能分析。但是出现以下提示。

```
<not supported>      cycles:u
<not supported>      instructions:u
<not supported>      cache-references:u
<not supported>      cache-misses:u
<not supported>      branch-misses:u

10.879976391 seconds time elapsed

9.923987000 seconds user
0.318076000 seconds sys
```

图 3.3: perf not support

不过我们可以预测一下 perf 的性能分析结果。

指标	md5_neon	md5
Cache 命中率	高	低
指令数	$\frac{n}{2} \sim n$	n
CPI	低 (SIMD 高吞吐)	高 (标量指令依赖)
实际吞吐量	高 (并行化 + 低 CPI)	低 (串行执行)

表 8: 预测 perf 分析结果

通过 Annotate MD5Hash_SSE 查看 MD5Hash_SSE 中最影响性能的部分。

Percent	
	nop
	uint32_t x_buf[batch] = {0};
168:	pxor %xmm0,%xmm0
	xor %eax,%eax
	movaps %xmm0,0x40(%rsp)
	x_buf[j] = *reinterpret_cast<uin
0.26	173: mov (%r14,%rax,8),%rcx
2.51	mov (%rcx,%rdx,1),%ecx
0.58	mov %ecx,(%rsi,%rax,4)
	for (int j = 0; j < batch; ++j)
2.25	add \$0x1,%rax
1.93	cmp \$0x4,%rax
	↑ jne 173
	x[k] = _mm_loadu_si128(reinterpr
0.13	movdqa 0x40(%rsp),%xmm1
29.84	movaps %xmm1,(%r9,%rdx,4)
	for (int k = 0; k < 16; ++k) {
6.37	add \$0x4,%rdx
	cmp %rdx,%rdi
	↑ jne 168

图 3.4: Annotate

分析得到限制 MD5Hash_SSE 性能的操作:

- (1)SIMD 数据加载与存储。
- (2) 外层循环控制指令。

4 思考

4.1 超过 64 字节的长指令的处理

对于存在很长的指令的情况,比如指令长度超过 56 字节,则经过 StringProcess 之后会变成 64*2、64*3、64*4... 的长度。此时由于处理后指令长度的不确定。导致我们进行向量化时无法直接从标量转为向量。

解决的想法:从上述实验的测试可以看到,当 block 数量为 1 时,4 条指令并行的 SIMD 方式能提升接近一倍的性能,2 条指令并行的 SIMD 方式也能提升 8% 的性能。

因此提出一种方法:

- (1) 求出 4 条指令 block 的长度,取最大值 MAX 和最小值 MIN。
- (2)MAX=MIN, 直接进行向量化处理。
- (3)MAX>MIN, 从 MIN 处进行 4 条指令 block 的截断。

出现以下三种情况:

(3.1)MIN 与另一指令长度相等,MAX 与另一指令长度相等,则将 4 条剩余 block 分成两组进行向量化处理。

(3.2) 其中两条指令 block 相等,且非 (3.1) 情况,则将相等两条指令剩余 block 进行向量化处理,余下两条指令 block 串行处理。

(3.3)4 条指令剩余 block 串行处理。

4.2 WSL2 上运行 NEON 版本

WSL2 上运行 NEON 版本,测试出的加速比会与在 ARM 平台上测试的结果相差较多。

查阅资料分析:

- (1) 虚拟化环境中的内存访问效率低于物理机。
- (2)SIMD 指令的硬件加速在虚拟化环境中可能被部分屏蔽。
- (3)QEMU 模拟器性能低。

4.3 优化方向

- (1) 优化 SIMD 数据加载与存储操作,减少标量和向量转换的操作。
- (2) 使用循环展开技术,减少外层循环控制指令。

参考文献

- [1] Intel intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=SSE_ALL.
- [2] Intrinsics - arm developer. <https://developer.arm.com/architectures/instruction-sets/intrinsics>.