南開大學
Nankai University

计算机学院

并行程序设计实验报告

实验三：SIMD 编程（口令猜测）

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 4 月 28 日

目录

1 问题重述

本次 SIMD 实验需基于 MD5 算法计算字符串的 hash 值，来完成对其并行化的优化。

先来说一下 MD5 算法的原理, 大体分为 5 个步骤 (在并行算法设计部分已有部分说明):

1. 消息填充（调用 StringProcess 函数）
2. 分块处理
3. 初始化寄存器
4. 依次处理每个块
5. 合并结果

2 串行算法运行测试

编译程序并提交到服务器进行运行后，返回结果，可以看到程序正常运行并返回结果，在此基础上，我们开始之后的实验。

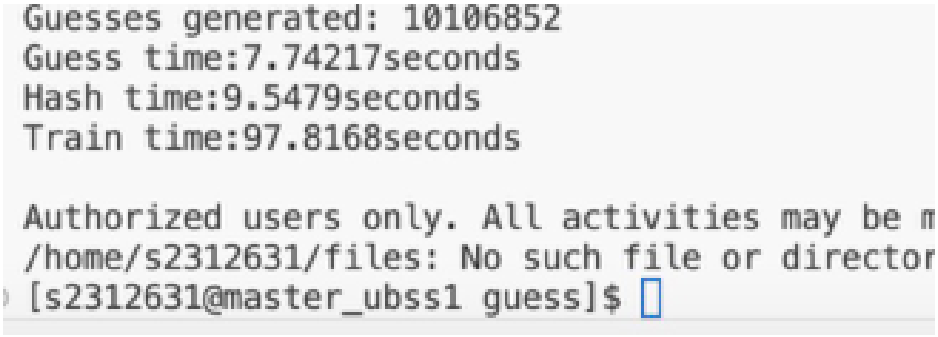

图 2.1: 串行算法测试

3 并行算法设计

3.1 并行算法思路

现给出 MD5 并行算法的思路：基于原来一次处理一个字符串，我们将其改为一次性处理 4 个字符串，具体实现是基于 NEON128 位指令集，其一次指令处理一个 128 位向量，也就是 4 个 32 位数据。

3.2 并行算法具体实现

既然要实现一次处理 4 个字符串的任务，也就是转化为一次对一个 128 位向量进行操作，我们需要更改相关的操作逻辑。主要需要并行化的就是 9 个逻辑函数, 这里只粘出三个:

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

#define ROTATELEFT_NEON(vec, n) \
```

```
    vorrq_u32(vshlq_n_u32((vec), (n)), vshrq_n_u32((vec), 32 - (n)))

#define FF_NEON(a, b, c, d, x, s, ac) do { \
  uint32x4_t _f_term = F_NEON((b), (c), (d));       \
  uint32x4_t _f_add = vaddq_u32(_f_term, (x));      \
  _f_add = vaddq_u32(_f_add, vdupq_n_u32(ac));      \
  (a) = vaddq_u32((a), _f_add);                     \
  (a) = ROTATELEFT_NEON((a), (s));                  \
  (a) = vaddq_u32((a), (b));                        \
} while(0)
```

F_NEON 函数的原型是计算

$$(X \wedge Y) \vee (\neg X \wedge Z)$$

, 并行版本即每次处理一个 128 位向量，对应的运算操作也应该适配于 128 位向量，vorrq_u32 对应的是或运算，vandq_u32 对应的是与运算；

ROTATELEFT_NEON 的目的是实现 128 位向量的循环左移操作，即 4 个 32 位数的循环左移操作，只需将每个数据左移 n 位，右移 32-n 位的结果合并起来 (取或) 即可。vshlq_n_u32 是左移操作，第一个参数为操作数，第二个参数为移动位数，vshrq_n_u32 对应了右移操作；

FF_NEON 是求取 hash 的一个步骤，用于对每个块进行操作，其参数 a、b、c、d 代表的是四个需要迭代的变量，每次函数处理只对第一个参数进行修改，x 代表的是传入四个字符串某个块的某 4 字节的 4 维向量，s，ac 都是一些常量，用于 hash 求取的过程，不再赘述。

然后比较重要的修改就是，我们需要修改 MD5hash 函数的 NEON 版本，给出代码如下：在 md5.cpp 中添加了 MD5Hash_NEON 函数

另外还需要在 main.cpp 中修改调用时的逻辑，比较简单，这里不再赘述。

在完成了并行化的处理之后，我发现对于 MD5Hash_NEON 函数中的分块处理，可以采用循环展开的逻辑，加快并行的效率，代码链接如下：在 md5_unroll.cpp 中进行了循环展开

```
void MD5Hash_NEON(string inputs[4], bit32 ** state)
{
  Byte *paddedMessages[4];
  int *messageLengths = new int[4];

  for (int i = 0; i < 4; i += 1)    //###########一次处理四个输入
  {
    paddedMessages[i] = StringProcess(inputs[i], &messageLengths[i]);
    assert(messageLengths[i] == messageLengths[0]);
  }
  int n_blocks = messageLengths[0] / 64;

  uint32x4_t state_temp[4];
  // bit32* state= new bit32[4];
  state_temp[0]=vdupq_n_u32(0x67452301);
  state_temp[1]=vdupq_n_u32(0xefcdab89);
  state_temp[2]=vdupq_n_u32(0x98badcfe);
  state_temp[3]=vdupq_n_u32(0x10325476);
```

以上操作是一些预处理的步骤，传入参数为一个包含 4 个字符串的数组和一个二维数组 state 用于存储 hash 结果。我们需要一次性对这 4 个字符串进行操作。通过调用 StringProcess 函数，进行预处理，填充消息使其长度符合 MD5 算法要求（512 位的倍数）。assert 用于保证处理后的消息长度相同，另外我们定义了 n_blocks 这个变量用于记录一个字符串被分为了多少个块 (由于输入的字符串长度都差不多，一个块即可搞定，所以可以并行化，每个块 64 字节)。然后定义了 4 个 128 位的向量寄存器 state_temp，利用 vdupq_n_u32 指令，将初始常量值复制到每个向量的所有维度上，即我们在循环中需要使用的 a、b、c、d。

```
// 逐block地更新state
for (int i = 0; i <n_blocks ; i += 1) //每个块64字节
{
  uint32x4_t x[16];
  for (int i1 = 0; i1 < 16; i1+=4) //将1个块分成16部分 每部分4字节
  {
    uint32_t data1[4];
    uint32_t data2[4];
    uint32_t data3[4];
    uint32_t data4[4];
    data1[0] = (paddedMessages[0][4 * i1 + i * 64]) |    //第0字节
        (paddedMessages[0][4 * i1 + 1 + i * 64] << 8) |//第1字节
        (paddedMessages[0][4 * i1 + 2 + i * 64] << 16) |//第2字节
        (paddedMessages[0][4 * i1 + 3 + i * 64] << 24);//第3字节
    data1[1] =paddedMessages[1]...
    data1[2] =paddedMessages[2]...
    data1[3] =paddedMessages[3]...
    x[i1] = vld1q_u32(data1);
              ..............
  对data2、data3、data4数组作相同处理，并传入x[i1+1]、x[i1+2]、x[i1+3]之中，此处省略
  }
```

外层循环遍历每个块，内层循环将每个块分成 16 个部分，每个部分 4 字节。在循环内部，当 i1=0 时，data1 取出的就是四个块的第一部分，即前 4 个字节，并存储到 x[i1] 中，之后遍历依次取 4 个块的某一部分，即对应的 4 个字节。值得一提的是，我在这里使用了循环展开，使得一次循环可以处理每个块的前 16 个字节，这样提高了效率。另外，我需要对这里的逻辑说明一下，这里的操作是将 4 个连续的字节组合成一个 32 位整数，并且通过移位操作取出字节，或操作拼接字节，按照小端序组合字节，vld1q_u32 指令可以把数组元素存储在向量寄存器中。

```
  uint32x4_t a = state_temp[0], b = state_temp[1], c = state_temp[2], d =
      state_temp[3];

  auto start = system_clock::now();
  /* Round 1 */
  FF_NEON(a, b, c, d, x[0], s11, 0xd76aa478);
      /* Round 2 */
      /* Round 3 */
      /* Round 4 */
                ..........
```

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

这里是算法的核心部分，首先将当前哈希状态复制到临时变量 a、b、c、d 中，然后去执行 MD5 的四轮计算，依次去调用我们实现的 FF_NEON, GG_NEON, HH_NEON, II_NEON 函数，每轮进行 16 次操作，另外需要将计算后的临时状态 a、b、c、d 与原始状态相加，更新 hash 状态，这里相加使用了 vaddq_u32 指令。这些操作通过向量寄存器同时对 4 个输入进行并行处理，大大提高了计算效率。

```
  for (int k = 0; k < 4; ++k) {
    state[0][k] = vgetq_lane_u32(state_temp[k], 0); // 第一个输入的a/b/c/d
    state[1][k] = vgetq_lane_u32(state_temp[k], 1); // 第二个输入的a/b/c/d
    state[2][k] = vgetq_lane_u32(state_temp[k], 2); // 第三个输入的a/b/c/d
    state[3][k] = vgetq_lane_u32(state_temp[k], 3); // 第四个输入的a/b/c/d
  }
  for (int i = 0; i < 4; i++)
  {
    uint32_t value = state[i][0];
    state[i][0] = ((value & 0xff) << 24) |      // 将最低字节移到最高位
          ((value & 0xff00) << 8) |    // 将次低字节左移
          ((value & 0xff0000) >> 8) |    // 将次高字节右移
          ((value & 0xff000000) >> 24); // 将最高字节移到最低位
    value = state[i][1];
    state[i][1]=.........
    value = state[i][2];
    state[i][2]=.........
    value = state[i][3];
    state[i][3]=.........
  }
```

在这部分，首先从 4 个向量寄存器中取出结果，存储在二维数组 state 中，使用了 vgetq_lane_u32 指令。然后对得到的每个 hash 值进行字节序变换，从小端序转变为大端序。拿一个来解释逻辑，(value & 0xff) 将某四个字节与 0xff 相与，即保留最低位，再将其左移 24 位，即保留到了最高位上，另外三个逻辑也类似。

```
  for (int i = 0; i < 4; i++) {
    delete[] paddedMessages[i];
  }
  delete[] messageLengths;
}
```

最后记得释放之前动态分配的内存。

3.3 并行算法的正确性验证

这里利用给的 correctness.cpp(已作相关修改) 对并行算法的正确性进行验证，可以看到我们实现的并行算法是没有问题的。

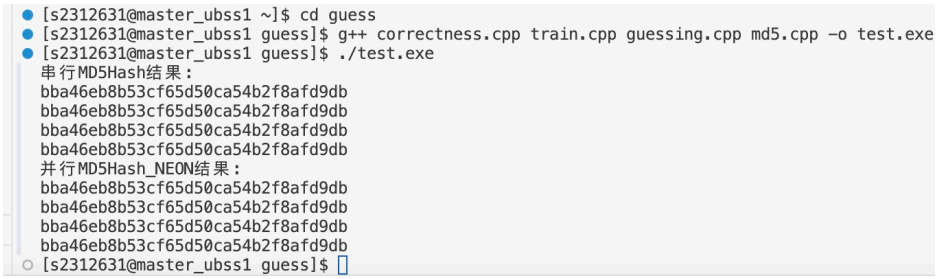

图 3.2: 验证并行算法的正确性

另外，为了验证其正确性，我在自己的 arm 电脑上分别运行并保存了并行和串行的 hash 结果，由于 github 上传文件的大小限制，故只提供了两张截图，事实证明并行算法和串行算法的结果完全一致。

4 性能对比

4.1 不开启编译优化的情况下

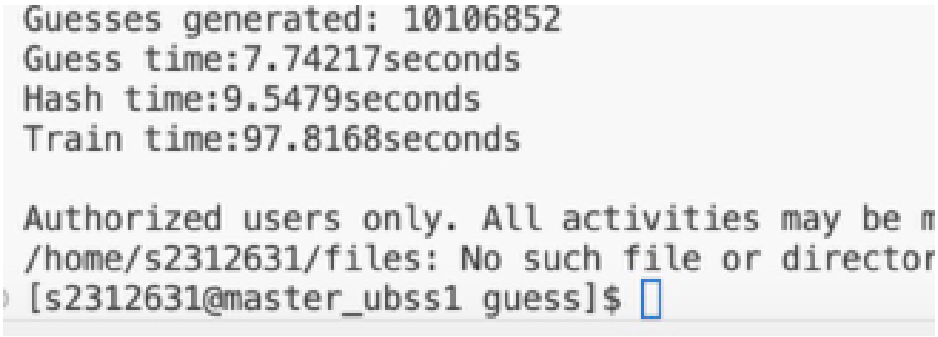

图 4.3: 串行算法

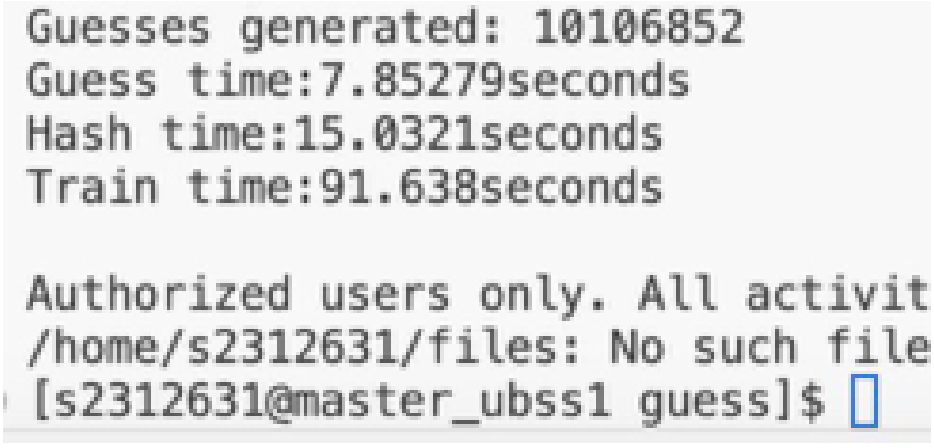

图 4.4: 并行算法

在不开启编译优化的情况下，串行算法 hash time 为 9.5479seconds，并行算法为 15.0321seconds，加速比为 0.6351。慢于串行的可能原因是，并行算法可能将数据分割到不同线程，导致内存访问模式不连续（如跨步访问），降低缓存利用率，而串行算法通常按顺序访问内存，更易被缓存预取优化。在之后编译优化部分也有这部分的分析，可能是因为 NEON 函数大量调用，没有进行内联展开，导致并行程序的执行时间更长。通过修改问题规模，得到了不同问题规模 n 下的串行时间和并行时间，加速比大致相同。在未开启编译优化的情况下，相同问题规模下，并行算法的执行时间均长于串行算法，加速比小于 1，猜测是由于上面提到的情况。

表 1: 不开启编译优化的情况下

问题规模 n	串行时间	并行时间	加速比
250w	2.06162s	3.23499s	0.6373
500w	4.18681s	6.38415s	0.6558
1000w	9.5479s	15.0321s	0.6351
2000w	19.118s	28.8331s	0.6631
3000w	28.3434s	44.2445s	0.6406

4.2 O1 优化情况下

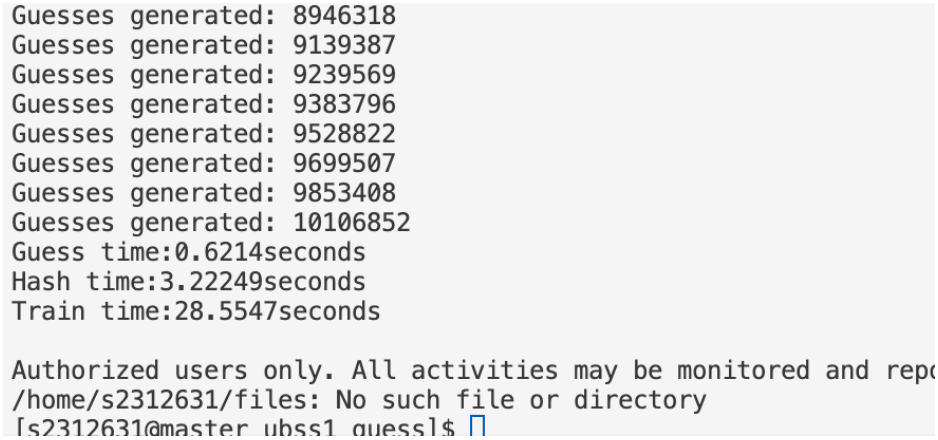

图 4.5: 串行算法

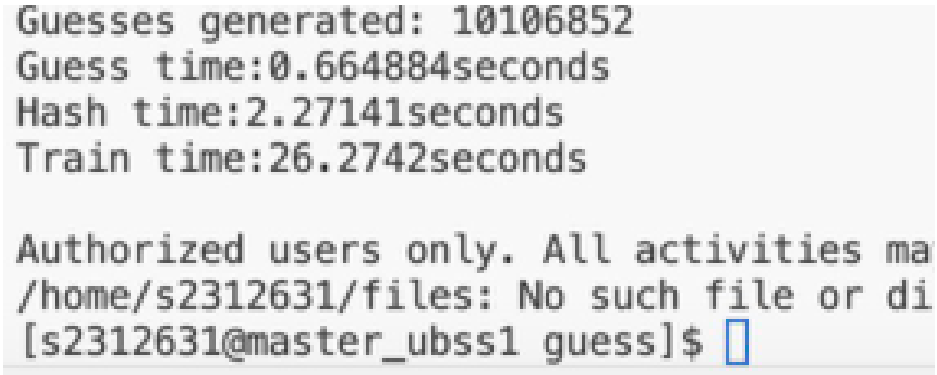

图 4.6: 并行算法

在 O1 优化的情况下，串行算法 hash time 为 3.22249seconds，并行算法为 2.27141seconds，加速比为 1.4187。可以看到，在 O1 编译优化的情况下，不同规模下的并行算法都优于串行算法。

表 2: O1 编译优化的情况下

问题规模 n	串行时间	并行时间	加速比
250w	0.680395s	0.410322s	1.658
500w	1.3972s	0.820188s	1.7035
1000w	3.22249s	2.27141s	1.4187
2000w	6.24842s	3.74257s	1.6696
3000w	9.3177s	5.58716s	1.6677

4.3 O2 优化情况下

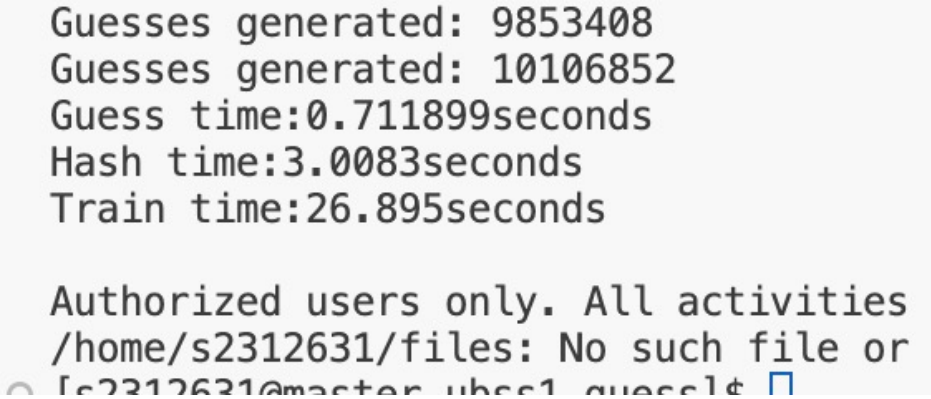

图 4.7: 串行算法

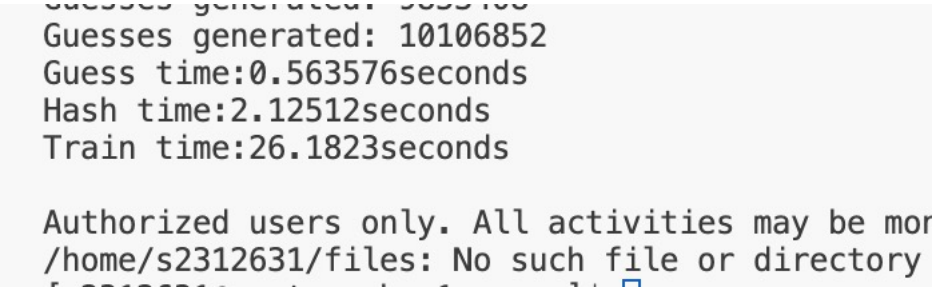

图 4.8: 并行算法

在 O2 优化的情况下，串行算法 hash time 为 3.0083seconds，并行算法为 2.12512seconds，加速比为 1.4155。

由于运行并行代码时，需要一次传入 4 个字符串，所以我使用了 vector 容器去接收，但这样会频繁的创建和释放 vector 容器，导致时间开销大大增加。所以我换了一种思路，采用一维数组去接收，这样就避免了不必要的开销，减少了运行时间。main_optimize.cpp

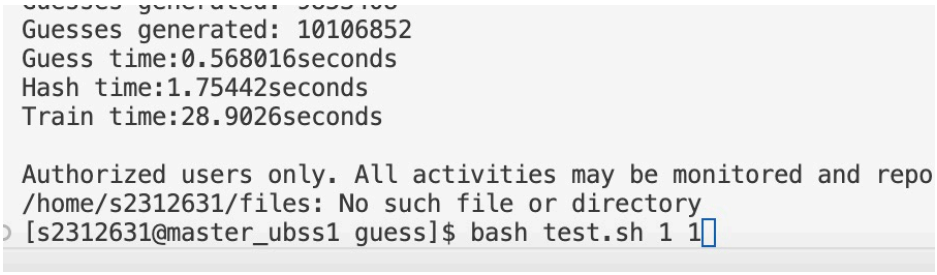

图 4.9: 并行算法优化

可以看到优化后的并行算法的 hash time 为 1.75442seconds，加速比为 1.7147。

在 O2 编译优化的情况下，也类似于 O1，不同规模下的并行算法都优于串行算法。但是可以发现的是，在多种情况下加速比都不如 O1 编译优化的情况下，原因可能是 O2 优化对串行的优化更多，使得加速比变得略小了一些。

表 3: O2 编译优化的情况下

问题规模 n	串行时间	并行时间	加速比
250w	0.654739s	0.468687s	1.397
500w	1.28342s	0.816379s	1.5721
1000w	3.0083s	2.12512s	1.7147
2000w	5.91223s	3.95275s	1.4957
3000w	8.47584s	5.34177s	1.5867

5 进阶要求的实现

5.1 实现相对串行算法的加速

已在上面分析，实现了并行算法相对于串行算法的优化，优化后的加速比达到了 1.7147。

5.2 编译时的选项会对加速比产生怎样的影响

5.2.1 基于 g++ 编译器编译时的优化选项进行调研

-O1 优化选项

-O1 是 GCC/G++ 的默认优化级别，它提供了一组合理的优化选项，旨在不显著增加编译时间的前提下提高程序的性能。-O1 包括的一些优化技术有：

函数内联：对一些小型函数进行内联展开，减少函数调用的开销。

常量传播：将代码中的常量表达式在编译时计算，减少运行时的计算量。

死代码消除：删除程序中永远不会执行的代码，减少程序的大小。

-O2 优化选项

-O2 优化级别在-O1 的基础上，进一步增加了更多的优化技术，以获得更好的性能提升。这些技术包括：

循环优化：包括循环展开、循环交换等，以减少循环的开销并提高缓存的利用率。

分支预测：优化分支指令，减少分支错误预测的可能性。

指令调度：重新排列指令的执行顺序，以提高流水线的利用率。

-O3 优化选项

-O3 优化级别包括-O2 的所有优化，并进一步应用了一些更激进的优化技术，这些技术可能会增加编译时间，但能获得更多的性能提升。例如：

进一步的指令调度和循环优化。

更激进的函数内联：对更大的函数进行内联展开。

浮点运算优化：对浮点运算进行特定的优化，以提高性能。

使用编译器标志

编译器标志是指导编译器应用特定优化技术的重要手段。通过合理使用这些标志，开发者可以显著提高程序的性能，减少资源消耗，并缩短开发周期。

例如-flto（链接时优化）和-march=native（为目标 CPU 优化）。

性能关键路径：可以使用-funroll-loops（循环展开）和-ftree-vectorize（循环向量化）等标志进行针对性优化。

内存敏感应用：可以使用-fdata-sections 和-ffunction-sections 标志来启用数据和函数级别的链接优化，从而减少程序的总内存占用。

代码大小优化：可以使用-Os 标志来优化代码大小，同时尽量保持性能。

另外，使用-fopt-info-优化级别标志可以生成优化信息报告。

5.2.2 对实验结果的分析

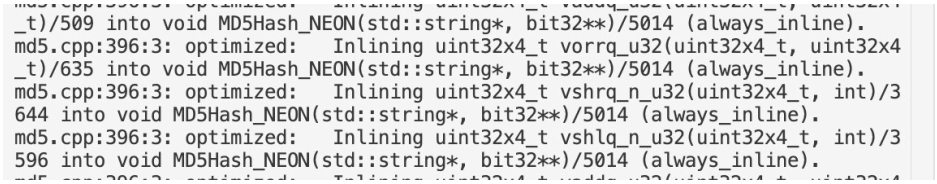

图 5.10: 优化报告

我首先生成了一下并行 O1 的优化报告，有 4200 行，其中很长一段内容都出现了 inline 的标志，通过查看，我发现是编译器对很多 NEON 指令进行了内联展开，减少函数调用的开销，这也就说明了为什么并行算法运行时间明显缩短了。另外这也在某种程度上解释了为什么未开编译优化的情况下，并行算法比串行算法慢，原因是我们使用了很多次 NEON 指令函数，调用了很多次，所以运行时间会比较长。另外，还有 std::string 的字样，其实是字符串的析构函数和拷贝构造函数会被内联，另外还进行了一些循环的展开，这些优化都会减少程序的运行时间。

接着我生成了串行 O1 的优化报告，除了无对于 NEON 指令的内联展开外，其他的部分都大致相同，在 O1 优化情况下，消除了 NEON 函数调用的开销，使得并行算法优于串行算法。

接着我生成了 O2 的优化报告，基本上大多数都是函数内联展开，并且增强了循环展开的力度，优化报告有 4300 行，优于 O1，所以运行速度也稍微快了些。

5.2.3 对加速比产生怎样的影响

基于实验结果及资料查询，以不开编译优化的情况下开始，并行算法由于大量调用函数所以差于串行算法，加速比小于 1；当开了 O1 编译优化时，并行算法优于串行算法，加速比大于 1；而当使用更高级别的编译优化时，如果并行算法存在可优化逻辑，那么有可能会使得加速比提高，否则如果无明显优化，加速比就会因串行算法运行时间大幅降低而减小，比如我们的实验结果中，基于 NEON 架构，simd 算法在 O2 编译情况下的加速比明显不如 O1。

5.3 X86 实现

5.3.1 实验环境

表 4: X86 环境

CPU 型号	i9-13900HX
CPU 主频	2.2GHz
L1 缓存	2.1MB
L2 缓存	32.0
L3 缓存	36.0MB

5.3.2 O2 编译下性能对比

代码修改与 NEON 并行修改类似，只需等价地修改一些变量及函数为 SSE 指令集的即可。代码链接如下:SSE 并行优化

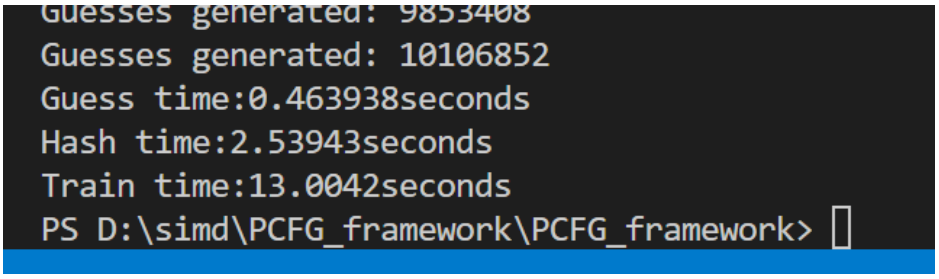

图 5.11: x86 串行

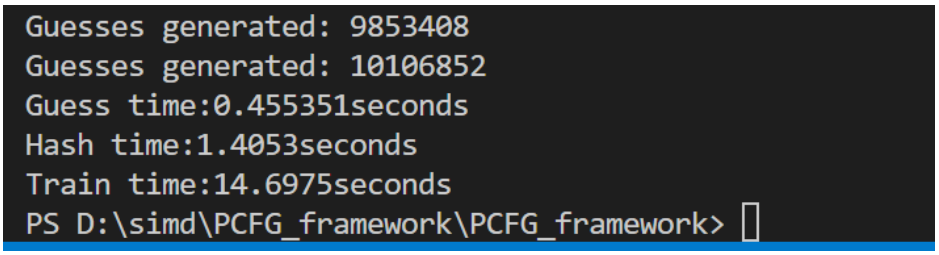

图 5.12: x86 并行

在我的 x86 电脑上，我实现了基于 SSE 指令集完成的并行算法优化。以数据规模为 1000w 为例，可以看到，串行算法的运行时间为 2.53943s，而并行算法的运行时间为 1.4053s，实现了加速，加速比为 1.807。我同样在 x86 电脑上运行并保存了结果，串并行结果完全一致。

然后我改变问题规模分别测了不同数据下的 hash time（已多次运行尽量确保结果的稳定性）。可以看到在多种问题规模下都实现了并行优化。

表 5: x86 O2 编译优化

问题规模 n	串行时间	并行时间	加速比
250w	0.395239s	0.203417s	1.943
500w	0.72977s	0.461166s	1.582
1000w	2.53943s	1.4053s	1.807
2000w	3.99582s	2.13347s	1.873
3000w	5.16803s	3.20985s	1.61

5.3.3 VTune 分析

由于 main 函数是把所有操作捏合在一起的，所以使用 vtune 分析性能时，对单独的 MD5hash 部分的分析效果不是很好，我在这里只看差别而不分析具体数据。

测试结果如下：

表 6: vtune 性能分析

指标	串行	并行
运行时间	9.820s	9.552s
总指令数	71.6B	63.9B
CPI	0.632	0.684
内存带宽利用率	32.6%	34.2%
分支预测失误率	18.6%	15.7%
Memory Bound	30.4%	29.0%

我们依次来分析这些指标，首先是运行时间，并行时间少于串行时间，这是我们预料之中的（这里看起来差距很小是因为这是整个 main 函数的性能分析）。

接着来看总指令数，串行指令数为 63,912,000,000，并行指令数为 71,558,400,000，可以看到，并行指令数明显少于串行指令数，说明了我们的 simd 编程的优越性，在并行算法中，我们采用了分块技术，向量化操作（SSE）合并多个标量操作，减少了冗余计算，进而减少了指令数。另外可能的是，编译器对并行算法的优化效能更好，使得指令数大量减少，使得原来需要 4 倍指令完成的任务，只需更少的指令数就可以完成了。

然后是 CPI，也就是每条指令执行了多少个周期，并行版本的 CPI 是 0.684，串行版本的 CPI 是 0.632，说明并行算法每条指令的平均执行周期更长。这里可能的原因是，由 Memory Bound 这个指标可知，并行版本的 Memory Bound 是 30.4%，高于串行版本的 29.0%，表明更多时间浪费在等待内存访问，这也是我们所能预料到的，原因是我们调用了更多的线程去作为函数接口，但可能仍不足大量数据的读入，导致访问内存的时间比较多。另外，多线程同时访问主存，线程管理和同步指令抵消了部分优化的收益，导致总线的延迟增加。

相较于串行算法，并行算法的指令数虽少但 CPI 高，使得总的运行时间差别不是很多。

另外还有内存带宽利用率，并行带宽略高，但未显著提升性能，表明内存访问模式不佳，原因可能是等待时间过长未能充分利用内存带宽。还有一个指标是分支预测失误率，串行算法的分支预测失误率为 18.6%，并行算法为 15.7%，并行分支预测更优，可能通过减少条件分支或优化预测逻辑实现，但优化效果仍不明显，仍有优化的空间。

总的来说，并行算法相较于串行算法来说，还是有一定的优化，但是由于更多的关于内存、线程、分支预测等方面的优化效果不是很好，或者说是受到其限制，可能有更好的并行优化存在，仍有优化的空间。

5.4 ARM（NEON）与 x86（SSE）平台加速比差异分析

通过之前的测试结果，我们发现，在 x86（SSE）平台的加速比能达到 1.8，优于在 ARM（NEON）平台，可以从这一点进行相关分析：

5.4.1 指令集特性对比

NEON（ARM）

寄存器与指令：

32 个 128 位向量寄存器（Q0-Q31），支持单指令多数据（如 4 个 32 位整数并行）。

指令集精简，部分复杂操作需组合多条指令实现（如无直接位旋转指令）。

编译器支持：GCC/Clang 对 NEON 的自动向量化能力较弱，需显式调用内联函数。

SSE（x86）

寄存器与指令：

16 个 128 位向量寄存器（XMM0-XMM15），支持更丰富的单指令操作（如 _mm_rot_epi32 直接循环移位）。

指令集包含更多专用指令（如内存对齐加载 _mm_load_si128），减少额外操作。

编译器支持：GCC/ICC 对 SSE 的自动向量化优化更成熟，代码生成效率更高。

x86 编译器（如 GCC）对 SSE 的优化更激进（如自动循环展开、函数内联），而 ARM 编译器对 NEON 的优化较为保守。

以上是我查阅到的相关资料，然后我发现这在程序中确确实实有验证，比如说在逻辑计算

$$\neg X \wedge Z$$

这种时，SSE 有专门的指令 **__mm__andnot__si128**，但是在 arm 中，需要使用两个指令，**vmvnq__u32** 按位取反，**vandq__u32** 按位取与，NEON 需通过组合指令实现相同功能，增加了指令流水线的压力，这就使得在 x86 平台上基于 SSE 的操作指令数更少，运行效率更高。

5.4.2 内存带宽与缓存差异

在 MD5 算法的向量化实现中，需频繁加载多个字符串的块数据到向量寄存器，那么对于程序从内存中读取数据的能力的要求就很高了，如果平台的内存带宽较低、缓存较少，就会使得并行效率降下来，因为不知道服务器的相关参数，可以猜测在内存带宽与缓存差异上不如我的 x86 实验环境。x86 平台更高的内存带宽和缓存命中率，支持更智能的 cache 预取机制，可更快完成数据加载，减少等待时间。

5.4.3 实验数据对比

表 7: x86 O2 编译优化

问题规模 n	串行时间	并行时间	加速比
250w	0.395239s	0.203417s	1.943
500w	0.72977s	0.461166s	1.582
1000w	2.53943s	1.4053s	1.807
2000w	3.99582s	2.13347s	1.873
3000w	5.16803s	3.20985s	1.61

表 8: O2 编译优化的情况下

问题规模 n	串行时间	并行时间	加速比
250w	0.654739s	0.468687s	1.397
500w	1.28342s	0.816379s	1.5721
1000w	3.0083s	2.12512s	1.7147
2000w	5.91223s	3.95275s	1.4957
3000w	8.47584s	5.34177s	1.5867

基于实验数据，可以发现 x86 加速比更高，无论是串行时间还是并行时间都有所减少，得益于 SSE 的指令效率与内存带宽优势，使得并行算法在 x86 上性能提升更显著,ARM 平台的内存延迟可能成为主要限制因素，这验证了我们关于内存带宽与缓存差异的猜想。

6 代码链接

点击访问 GitHub 主页