



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。

```

1  #define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
2
3  #define ROTATELEFT(num, n) (((num) << (n)) | ((num) >> (32-(n))))
4
5  #define FF(a, b, c, d, x, s, ac) { \
6      (a) += F ((b), (c), (d)) + (x) + ac; \
7      (a) = ROTATELEFT ((a), (s)); \
8      (a) += (b); \
9  }
```

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：

```

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

```

3 }
4 inline bit32x4_t ROTATELEFT_SIMD(bit32x4_t x, int n) {
5     return vorrq_u32(vshlq_n_u32(x, n), vshrq_n_u32(x, 32 - n));
6 }
7 inline bit32x4_t FF_SIMD(*a, b, c, d, x, s, ac) { \
8     (*a) = vaddq_u32((a), vaddq_u32(vaddq_u32(F_SIMD((b), (c), (d)), (x)), vdupq_n_u32(ac))); \
9     (*a) = ROTATELEFT_SIMD((a), (s)); \
10    (*a) = vaddq_u32((a), (b)); \
11 }

```

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。

<pre> 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 </pre>	<pre> 86 Guesses generated: 9853408 87 Guesses generated: 10106052 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 </pre>
--	---

(a) 串行时间
(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 $states[4][4]$

```

1: function MD5HASH_SIMD( $inputs[4]$ ,  $states[4][4]$ )
2:   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
3:   创建填充后的消息数组  $paddedMessages[4]$ 
4:   创建消息长度数组  $messageLengths[4]$ 
5:   for  $i \leftarrow 0$  to 3 do
6:      $paddedMessages[i] \leftarrow \text{StringProcess}(inputs[i], \&messageLengths[i])$ 
7:   end for
8:    $n\_blocks \leftarrow messageLengths[0]/64$ 
9:   for  $block \leftarrow 0$  to  $n\_blocks - 1$  do
10:    创建块向量数组  $M[16]$ 
11:    for  $j \leftarrow 0$  to 15 do
12:      for  $i \leftarrow 0$  to 3 do
13:        从  $paddedMessages[i]$  的第  $block$  块中提取第  $j$  个 32 位字  $values[i]$ 
14:      end for
15:       $M[j] \leftarrow \text{vld1q\_u32}(values)$ 
16:    end for
17:     $A \leftarrow a_0, B \leftarrow b_0, C \leftarrow c_0, D \leftarrow d_0$ 
18:    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
19:     $a_0 \leftarrow a_0 + A$ 
20:     $b_0 \leftarrow b_0 + B$ 
21:     $c_0 \leftarrow c_0 + C$ 
22:     $d_0 \leftarrow d_0 + D$ 
23:  end for
24:  提取最终哈希值  $a\_values[4], b\_values[4], c\_values[4], d\_values[4]$ 
25:  for  $i \leftarrow 0$  to 3 do
26:     $states[i][0] \leftarrow \text{ByteSwap}(a\_values[i])$ 
27:     $states[i][1] \leftarrow \text{ByteSwap}(b\_values[i])$ 
28:     $states[i][2] \leftarrow \text{ByteSwap}(c\_values[i])$ 
29:     $states[i][3] \leftarrow \text{ByteSwap}(d\_values[i])$ 
30:  end for
31:  for  $i \leftarrow 0$  to 3 do
32:    释放  $paddedMessages[i]$ 
33:  end for
34:  return  $states$ 
end function

```

```

1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319 1320 1321 1322 1323 1324 1325 1326 1327 1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341 1342 1343 1344 1345 1346 1347 1348 1349 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359 1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376 1377 1378 1379 1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399 1400 1401 1402 1403 1404 1405 1406 1407 1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427 1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451 1452 1453 1454 1455 1456 1457 1458 1459 1460 1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487 1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527 1528 1529 1530 1531 1532 1533 1534 1535 1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549 1550 1551 1552 1553 1554 1555 1556 1557 1558 1559 1560 1561 1562 1563 1564 1565 1566 1567 1568 1569 1570 1571 1572 1573 1574 1575 1576 1577 1578 1579 1580 1581 1582 1583 1584 1585 1586 1587 1588 1589 1590 1591 1592 1593 1594 1595 1596 1597 1598 1599 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611 1612 1613 1614 1615 1616 1617 1618 1619 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631 1632 1633 1634 1635 1636 1637 1638 1639 1640 1641 1642 1643 1644 1645 1646 1647 1648 1649 1650 1651 1652 1653 1654 1655 1656 1657 1658 1659 1660 1661 1662 1663 1664 1665 1666 1667 1668 1669 1670 1671 1672 1673 1674 1675 1676 1677 1678 1679 1680 1681 1682 1683 1684 1685 1686 1687 1688 1689 1690 1691 1692 1693 1694 1695 1696 1697 1698 1699 1700 1701 1702 1703 1704 1705 1706 1707 1708 1709 1710 1711 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721 1722 1723 1724 1725 1726 1727 1728 1729 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741 1742 1743 1744 1745 1746 1747 1748 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761 1762 1763 1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775 1776 1777 1778 1779 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791 1792 1793 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 1810 1811 1812 1813 1814 1815 1816 1817 1818 1819 1820 1821 1822 1823 1824 1825 1826 1827 1828 1829 1830 1831 1832 1833 1834 1835 1836 1837 1838 1839 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851 1852 1853 1854 1855 1856 1857 1858 1859 1860 1861 1862 1863 1864 1865 1866 1867 1868 1869 1870 1871 1872 1873 1874 1875 1876 1877 1878 1879 1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 1890 1891 1892 1893 1894 1895 1896 1897 1898 1899 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031 2032 2033 2034 2035 2036 2037 2038 2039 2040 2041 2042 2043 2044 2045 2046 2047 2048 2049 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091 2092 2093 2094 2095 2096 2097 2098 2099 2100 2101 2102 2103 2104 2105 2106 2107 2108 2109 2110 2111 2112 2113 2114 2115 2116 2117 2118 2119 2120 2121 2122 2123 2124 2125 2126 2127 2128 2129 2130 2131 2132 2133 2134 2135 2136 2137 2138 2139 2140 2141 2142 2143 2144 2145 2146 2147 2148 2149 2150 2151 2152 2153 2154 2155 2156 2157 2158 2159 2160 2161 2162 2163 2164 2165 2166 2167 2168 2169 2170 2171 2172 2173 2174 2175 2176 2177 2178 2179 2180 2181 2182 2183 2184 2185 2186 2187 2188 2189 2190 2191 2192 2193 2194 2195 2196 2197 2198 2199 2200 2201 2202 2203 2204 2205 2206 2207 2208 2209 2210 2211 2212 2213 2214 2215 2216 2217 2218 2219 2220 2221 2222 2223 2224 2225 2226 2227 2228 2229 2230 2231 2232 2233 2234 2235 2236 2237 2238 2239 2240 2241 2242 2243 2244 2245 2246 2247 2248 2249 2250 2251 2252 2253 2254 2255 2256 2257 2258 2259 2260 2261 2262 2263 2264 2265 2266 2267 2268 2269 2270 2271 2272 2273 2274 2275 2276 2277 2278 2279 2280 2281 2282 2283 2284 2285 2286 2287 2288 2289 2290 2291 2292 2293 2294 2295 2296 2297 2298 2299 2300 2301 2302 2303 2304 2305 2306 2307 2308 2309 2310 2311 2312 2313 2314 2315 2316 2317 2318 2319 2320 2321 2322 2323 2324 2325 2326 2327 2328 2329 2330 2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343 2344 2345 2346 2347 2348 2349 2350 2351 2352 2353 2354 2355 2356 2357 2358 2359 2360 2361 2362 2363 2364 2365 2366 2367 2368 2369 2370 2371 2372 2373 2374 2375 2376 2377 2378 2379 2380 2381 2382 2383 2384 2385 2386 2387 2388 2389 2390 2391 2392 2393 2394 2395 2396 2397 2398 2399 2400 2401 2402 2403 2404 2405 2406 2407 2408 2409 2410 2411 2412 2413 2414 2415 2416 2417 2418 2419 2420 2421 2422 2423 2424 2425 2426 2427 2428 2429 2430 2431 2432 2433 2434 2435 2436 2437 2438 2439 2440 2441 2442 2443 2444 2445 2446 2447 2448 2449 2450 2451 2452 2453 2454 2455 2456 2457 2458 2459 2460 2461 2462 2463 2464 2465 2466 2467 2468 2469 2470 2471 2472 2473 2474 2475 2476 2477 2478 2479 2480 2481 2482 2483 2484 2485 2486 2487 2488 2489 2490 2491 2492 2493 2494 2495 2496 2497 2498 2499 2500 2501 2502 2503 2504 2505 2506 2507 2508 2509 2510 2511 2512 2513 2514 2515 2516 2517 2518 2519 2520 2521 2522 2523 2524 2525 2526 2527 2528 2529 2530 2531 2532 2533 2534 2535 2536 2537 2538 2539 2540 2541 2542 2543 2544 2545 2546 2547 2548 2549 2550 2551 2552 2553 2554 2555 2556 2557 2558 2559 2560 2561 2562 2563 2564 2565 2566 2567 2568 2569 2570 2571 2572 2573 2574 2575 2576 2577 2578 2579 2580 2581 2582 2583 2584 2585 2586 2587 2588 2589 2590 2591 2592 2593 2594 2595 2596 2597 2598 2599 2600 2601 2602 2603 2604 2605 2606 2607 2608 2609 2610 2611 2612 2613 2614 2615 2616 2617 2618 2619 2620 2621 2622 2623 2624 2625 2626 2627 2628 2629 2630 2631 2632 2633 2634 2635 2636 2637 2638 2639 2640 2641 2642 2643 2644 2645 2646 2647 2648 2649 2650 2651 2652 2653 2654 2655 2656 2657 2658 2659 2660 2661 2662 2663 2664 2665 2666 2667 2668 2669 2670 2671 2672 2673 2674 2675 2676 2677 2678 2679 2680 2681 2682 2683 2684 2685 2686 2687 2688 2689 2690 2691 2692 2693 2694 2695 2696 2697 2698 2699 2700 2701 2702 2703 2704 2705 2706 2707 2708 2709 2710 2711 2712 2713 2714 2715 2716 2717 2718 2719 2720 2721 2722 2723 2724 2725 2726 2727 2728 2729 2730 2731 2732 2733 2734 2735 2736 2737 2738 2739 2740 2741 2742 2743 2744 2745 2746 2747 2748 2749 2750 2751 2752 2753 2754 2755 2756 2757 2758 2759 2760 2761 2762 2763 2764 2765 2766 2767 2768 2769 2770 2771 2772 2773 2774 2775 2776 2777 2778 2779 2780 2781 2782 2783 2784 2785 2786 2787 2788 2789 2790 2791 2792 2793 2794 2795 2796 2797 2798 2799 2800 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845 2846 2847 2848 2849 2850 2851 2852 2853 2854 2855 2856 2857 2858 2859 2860 2861 2862 2863 2864 2865 2866 2867 2868 2869 2870 2871 2872 2873 2874 2875 2876 2877 2878 2879 2880 2881 2882 2883 2884 2885 2886 2887 2888 2889 2890 2891 2892 2893 2894 2895 2896 2897 2898 2899 2900 2901 2902 2903 2904 2905 2906 2907 2908 2909 2910 2911 2912 2913 2914 2915 2916 2917 2918 2919 2920 2921 2922 2923 2924 2925 2926 2927 2928 2929 2930 2931 2932 2933 2934 2935 2936 2937 2938 2939 2940 2941 2942 2943 2944 2945 2946 2947 2948 2949 2950 2951 2952 2953 2954 2955 2956 2957 2958 2959 2960 2961 2962 2963 2964 2965 2966 2967 2968 2969 2970 2971 2972 2973 2974 2975 2976 2977 2978 2979 2980 2981 2982 2983 2984 2985 2986 2987 2988 2989 2990 2991 2992 2993 2994 2995 2996 2997 2998 2999 3000 3001 3002 3003 3004 3005 3006 3007 3008 3009 3010 3011 3012 3013 3014 3015 3016 3017 3018 3019 3020 3021 3022 3023 3024 3025 3026 3027 3028 3029 3030 3031 3032 3033 3034 3035 3036 3037 3038 3039 3040 3041 3042 3043 3044 3045 3046 3047 3048 3049 3050 3051 3052 3053 3054 3055 3056 3057 3058 3059 3060 3061 3062 3063 3064 3065 3066 3067 3068 3069 3070 3071 3072 3073 3074 3075 3076 3077 3078 3079 3080 3081 3082 3083 3084 3085 3086 3087 3088 3089 3090 3091 3092 3093 3094 3095 3096 3097 3098 3099 3100 3101 3102 3103 3104 3105 3106 3107 3108 3109 3110 3111 3112 3113 3114 3115 3116 3117 3118 3119 3120 3121 3122 3123 3124 3125 3126 3127 3128 3129 3130 3131 3132 3133 3134 3135 3136 3137 3138 3139 3140 3141 3142 3143 3144 3145 3146 3147 3148 3149 3150 3151 3152 3153 3154 3155 3156 3157 3158 3159 3160 3161 3162 3163 3164 3165 3166 3167 3168 3169 3170 3171 3172 3173 3174 3175 3176 3177 3178 3179 3180 3181 3182 3183 3184 3185 3186 3187 3188 3189 3190 3191 3192 3193 3194 3195 3196 3197 3198 3199 3200 3201 3202 3203 3204 3205 3206 3207 3208 3209 3210 3211 3212 3213 3214 3215 3216 3217 3218 3219 3220 3221 3222 3223 3224 3225 3226 3227 3228 3229 3230 3231 3232 3233 3234 3235 3236 3237 3238 3239 3240 3241 3242 3243 3244 3245 3246 3247 3248 3249 3250 3251 3252 3253 3254 3255 3256 3257 3258 3259 3260 3261 3262 3263 3264 3265 3266 3267 3268 3269 3270 3271 3272 3273 3274 3275 3276 3277 3278 3279 3280 3281 3282 3283 3284 3285 3286 3287 3288 3289 3290 3291 3292 3293 3294 3295 3296 3297 3298 3299 3300 3301 3302 3303 3304 3305 3306 3307 3308 3309 3310 3311 3312 3313 3314 3315 3316 3317 3318 3319 3320 3321 3322 3323 3324 3325 3326 3327 3328 3329 3330 3331 3332 3333 3334 3335 3336 3337 3338 3339 3340 3341 3342 3343 3344 3345 3346 3347 3348 3349
```

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。

Samples: 191K of event 'cycles:u', 4000 Hz, Event count (approx.): 71032229743, DSO: main	
MD5Hash SIMD /home/s2313211/guess/main [Percent: local period]	
Percent	
	lsl x19, x0, #2
	mov x0, x19
	→ bl _init
	mov x2, x0
	mov x1, x2
	sub x0, x19, #0x1
4.56	17c: cmp x0, #0x0
	↓ b.lt 194
0.05	strb wzr, [x1]
	add x1, x1, #0x1
0.02	sub x0, x0, #0x1
	↑ b 17c

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
```

所以现在我们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


```

3 }
4 #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。

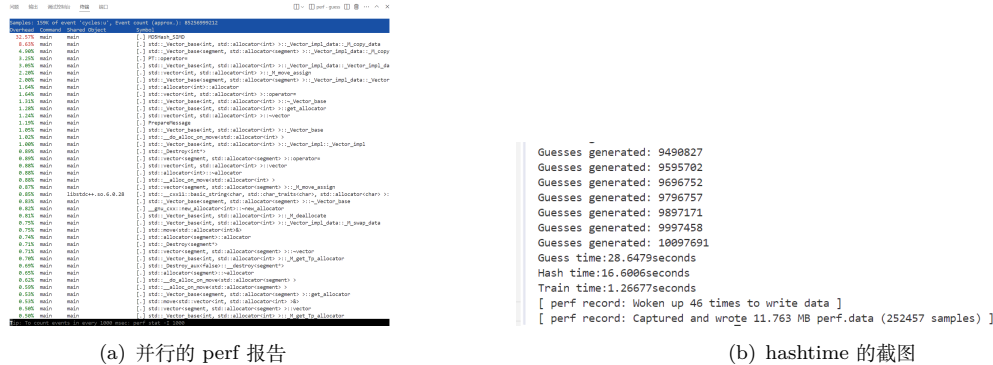


图 2.7: 内联函数改宏定义的效果

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

```

1 //在 md5.h 里面声明指针变量
2 static Byte zero_buffer[MAX_BUFFER_SIZE] = {0};
3 //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
4 void CleanupMD5Resources() {
5     delete[] reusable_buffers;
6     reusable_buffers = nullptr;
7     total_buffer_capacity = 0;
8 }

```

(2) 缓存区

```

1 // 直接写入传入的 output 缓冲区
2 memcpy(output, input.c_str(), input_length);
3 //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。
4 //其他时候直接使用赋值语句进行填充，增加效率。
5 if (padding_bytes - 1 <= MAX_BUFFER_SIZE) {

```



```

6     memcpy(output + input_length + 1, zero_buffer, padding_bytes - 1);
7 }
8 else {
9     memset(output + input_length + 1, 0, padding_bytes - 1);
10 }

```

(3) 静态数组预分配

```

1 // 使用静态预分配的数组替代栈上数组
2 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。

```

问题  输出  调试控制台  终端  端口
Guesses generated: 9696752
Guesses generated: 9796757
Guesses generated: 9897171
Guesses generated: 9997458
Guesses generated: 10097691
Guess time: 28.6052seconds
Hash time: 15.4364seconds
Train time: 1.28767seconds
[ perf record: Woken up 46 times to write data ]
[ perf record: Captured and wrote 11.447 MB perf.data (246995 samples) ]
[s2313211@master_ubss1 guess]$

```

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：

```

1 str q0, [x4]
2 add w2, w2
3 add x0, x0
4 cmp w2
5 //对应的实际代码段是 memcpy 的行为
6 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：



图 3.9: o1 的串行与并行的比较



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);

```

```

5 //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。
6 temp_not_b = vmvnq_u32(b);
7 //即变为一下两个表达式
8 result1 = vorrq_u32(vandq_u32(b, c), vandq_u32(temp_not_b, d));
9 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:

```

1 //在进行运算处理的主要步骤时
2 FF_SIMD(A, B, C, D, M[0], s11, 0xd76aa478);
3 FF_SIMD(D, A, B, C, M[1], s12, 0xe8c7b756);
4 FF_SIMD(C, D, A, B, M[2], s13, 0x242070db);
5 .....

```

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

```

1 for (int j = 0; j < 16; j++) {
2 // 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
3 uint32_t values[4] __attribute__((aligned(16)));
4 memcpy(&values[0], paddedMessages[0] + offset, 4);
5 // ...
6 M[j] = vld1q_u32(values);
7 } //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。

```

1  #define FF_SIMD(a, b, c, d, x, s, ac) { \
2  (a) = vaddq_u32((a), vaddq_u32(vaddq_u32(F_SIMD((b), (c), (d)), (x)), vdupq_n_u32(ac))); \
3  (a) = ROTATELEFT_SIMD((a), (s)); \
4  (a) = vaddq_u32((a), (b)); \
5  }

```

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。

```

1  // 在每个 FF_SIMD 调用中
2  FF_SIMD(A, B, C, D, M[0], s11, 0xd76aa478);
3  // 展开为
4  (A) = vaddq_u32((A),
5  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。

```

1 // 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
2 vandq_u32(B, C)           // 逻辑运算 (1-2 周期)
3 vmvnq_u32(B)              // 位运算 (1 周期)
4 vaddq_u32(...)            // 加法 (3-4 周期)
5 vdupq_n_u32(0xd76aa478)   // 常量加载 (1-2 周期)
6 //这两行代码用来解释优化编译如何使得流水线满载
7 FF_SIMD(A, B, C, D, M[0], s11, 0xd76aa478);
8 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。

```

1 // 伪指令序列，展示如何交错执行
2 vdupq_n_u32(0xd76aa478)           // 第一个 FF_SIMD 的常量
3 vdupq_n_u32(0xe8c7b756)           // 同时，第二个 FF_SIMD 的常量也可提前准备
4 F_SIMD 的计算 (B,C,D)              // 第一个 FF_SIMD 的 F 函数
5 加载 M[1]                          // 同时预取第二个 FF_SIMD 的消息块
6 vaddq_u32 操作完成 A 值更新         // 完成第一个 FF_SIMD
7 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)
8 //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
9 vaddq_u32(A, ...)                  // 需要 3-4 个周期
10 vdupq_n_u32(下一个常量)           // 同时开始，不需要等待 vaddq 结果
11 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


```

4     _mm_and_si128(_mm_xor_si128((x), _mm_set1_epi32(-1)), (z)) \
5 )
6 #define ROTATELEFT_simd(num, n) \
7 _mm_or_si128( \
8     _mm_slli_epi32((num), (n)), \
9     _mm_srli_epi32((num), 32 - (n)) \
10 )
11 #define FF_simd(a, b, c, d, x, s, ac) { \
12     a = _mm_add_epi32(a, F_simd(b, c, d)); \
13     a = _mm_add_epi32(a, x); \
14     a = _mm_add_epi32(a, _mm_set1_epi32(ac)); \
15     a = ROTATELEFT_simd(a, s); \
16     a = _mm_add_epi32(a, b); \
17 }

```

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：

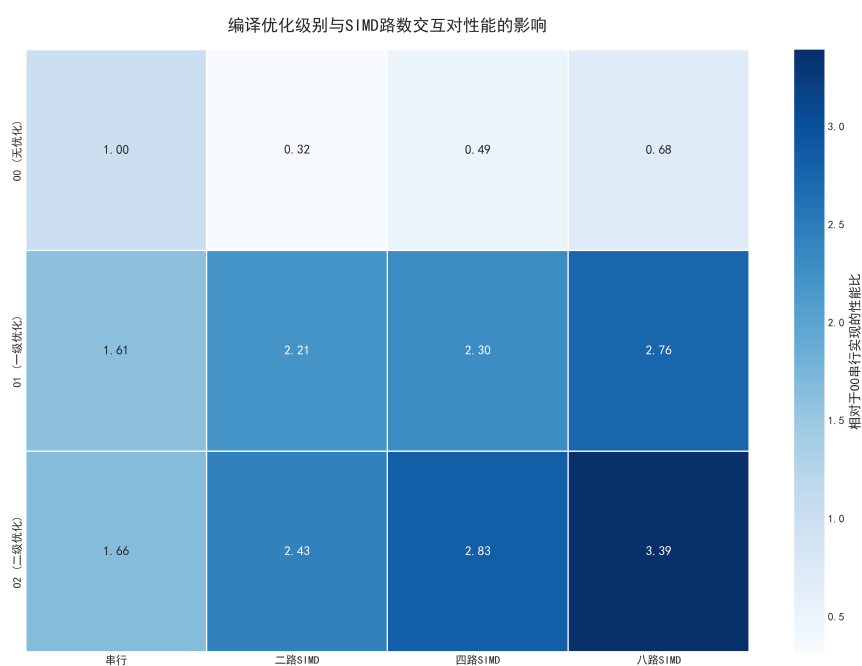


图 3.12: 不同并行度的加速比热力图

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

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

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。

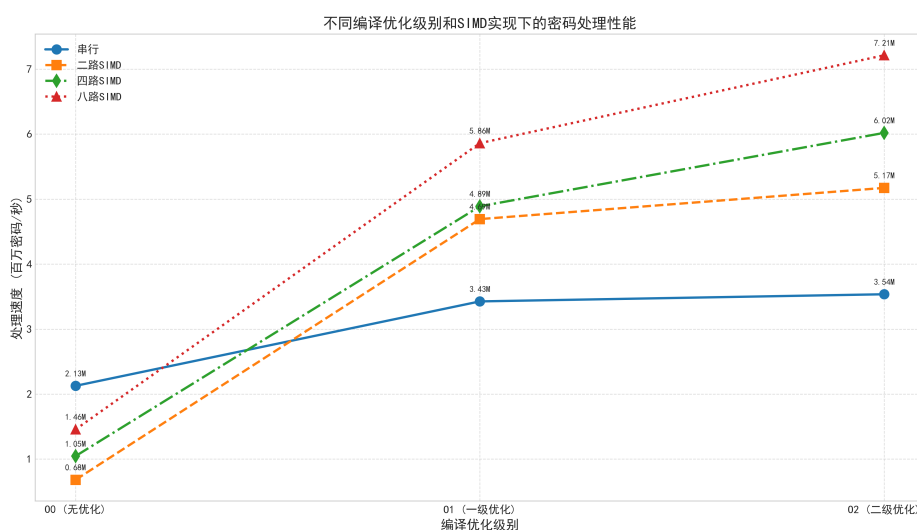


图 3.13: 不同并行度的处理速度折线图

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>