Lab1 实验报告

人工智能学院 薛正海 181220063

1. 实现multimod

1.1. 代码实现

multimod的原始实现借鉴了高精度乘法的想法,即提取乘数的每一位数字,并手动进行乘法过程.在 `p1.c` 中,提取数字的函数是 `gen_bits()` ,返回指向数字数组的指针.最终计算的过程就是一个二重循环,其中有几个防止溢出的细节:

1. 对乘数1的第a位和乘数2的第b位时,需对1左移$a+b$位.连续移位可能导致溢出,故必须在移位后立刻对m取模.即使这样,当$m<x<2m$且m很大时,计算$2x$的移位过程本身会溢出,需计算$x-m+x$.
2. 累加结果sum在累加的过程中可能暂时溢出,但不影响最终结果,故可使用 `unsigned int64_t sum` , 并 `assert(sum>>63==0)`

1.2. 测试

测试采用10000组随机生成的64位数据,生成过程在函数 `gen_rand_64()` 中.测试文件为 `test_12.py` ,运行前需在对应c代码中定义TIMING宏.所有测试全部通过.

2. 优化multimod

2.1. 代码改进

multimod的原始实现的时间复杂度达到了$O(n^2)$,主要耗时在于乘法内部的双重循环,其中每一步只能对两乘数的各一位进行计算.一个改进的想法就是每一步对一个乘数 `b` 整体和另一个乘数 `a` 的一位进行运算,每计算 `a` 的左侧一位,只需将 `b` 左移一位即可.这样可将时间复杂度减至$O(n)$.同时对乘数每一位数字的提取消耗了大量的时间空间.注意到 `a` 的一位或0或1,可以通过模2取余得到,这样就免去了额外存储 `a` 的每一位.

2.2. 测试

首先正确性如原multimod通过测试.运行时间如下表所示:

	原multimod	**改进后的multimod**
O0优化	5995ms ± 16ms	**98** ms ± 8ms

	原multimod	改进后的multimod
O1优化	3880ms ± 28ms	**99** ms ± 12ms
O2优化	3641ms ± 21ms	**96** ms ± 8ms
O3优化	3653ms ± 40ms	**92** ms ± 15ms

其中我加入了对每组数据的输出,防止计算过程被编译优化.可以看出我的优化大大减少了运行时间.

3. 解析神秘代码

3.1. 表达式分析

若记 $x =$ (int64_t)((double)a * b / m)*m, 则

$$t \equiv a \times b - x \equiv a \times b \,(\mathrm{mod}\, m) \text{ iff } x \equiv 0 (\mathrm{mod}\, m) \;\; (*)$$

即数学上,只要x被m整除,表达式结果就是正确的.条件$(*)$容易验证.但若考虑溢出, x必须足够大,使 $a \times b - x \leq 2^{63} - 1$.

Theorm 1:若 $A = \{x | x \leq a \times b \ \wedge \ m | x\},\ x = \sup A (**)$,则$a \times b - x \leq 2^{63} - 1$.

证明: 假设$a \times b - x > 2^{63} - 1$,由 $m < 2^{63} - 1$, 有$a \times b - x > m$. 则存在 $x' = x + m$, s.t. $m | x'$ 且 $a \times b - x' = a \times b - x - m > 0,\ x' < a \times b$. 故 $x' \in A, x' > x$, 与 $x = \sup A$矛盾! 证毕.

而事实上,如果 `(double)a * b / m)` 计算精确, 条件$(**)$可由强制类型转换 `(int64_t)` 保证; 如果不精确且数值偏大, 可能导致 $x > a \times b$, 可由 `t < 0 ? t + m : t` 的判断弥补;但如果不精确且数值偏小,可能导致 $a \times b - x > 2^{63} - 1$,使结果溢出.

3.2. 实验

3.2.1. double保证精度

根据 `IEEE754` 浮点数标准, `double` 类型的尾数为52位,加上标准形式的1共计53位,所以当 $a \times b \leq 2^{53} - 1$ 时, `double` 能保证乘法的精度.根据理论推导,代码能产生正确结果.测试代码为 `test_3.py` 中的 `test1()` .经实验验证,代码确实能产生正确结果.

3.2.2. double损失精度

测试代码为 `test_3.py` 中的 `test2()` .其中为判断 `double(a*b)` 与 $a \times b$ 的大小关系, 调用了 `get_double` 程序,其返回 `(double)input` .下表为实验结果:

`double(a*b)` 与$a \times b$的关系	大于	小于	等于
总数量	5004	4996	0
结果正确的数量	135	124	0

其中 `double(a*b)` $> a \times b$ 的情形正确率较低,与理论预测相符. 但是在 `double(a*b)` $< a \times b$ 的情形中, `t < 0 ? t + m : t` 的弥补措施并未提高准确率.进一步调试后发现: `double(a*b)//m` 与 `(a*b)//m` 差距较大,基本维持在 10^2 的数量级,再乘m同样会有很大概率溢出.

启示:浮点数表示范围大,可一定程度上缓解乘法溢出的问题; 但损失精度后,其所能表示的大数过于稀疏,不能准确计算.