

省流：观点与建议

个人观点：Project 1 POW 运行时间异常，是 CPU 上的 x87 浮点数运算单元在处理浮点数的 `inf` 参与的运算时效率降低导致的。

个人建议：

- 做 Project 1 POW 的同学应注意浮点数溢出问题，在测试算法时，可选择更精确的计时器 + 一般的 a + 较小的 N ，使运算不会发生溢出，这样的结果最真实。
- 如果没有更精确的计时器，可选择 $a = 1.0$ 作为底数，同时取消编译优化（要不然就可能直接跳过 `pow` 运算直接返回 `1.0` 了）。
- 如果程序中确实发生了浮点数溢出，那么建议同学修改代码重新进行测试，因为此时的运算时间是不准确的，会干扰时间复杂度的分析。
- 建议使用更高版本的 Dev-C++，以及新版本的编译器。
- 建议在 Dev-C++ 的工具栏里，将编译器选项从 32 位 Release 位调整到 64 位 Release。

下面是具体的分析过程：

问题描述

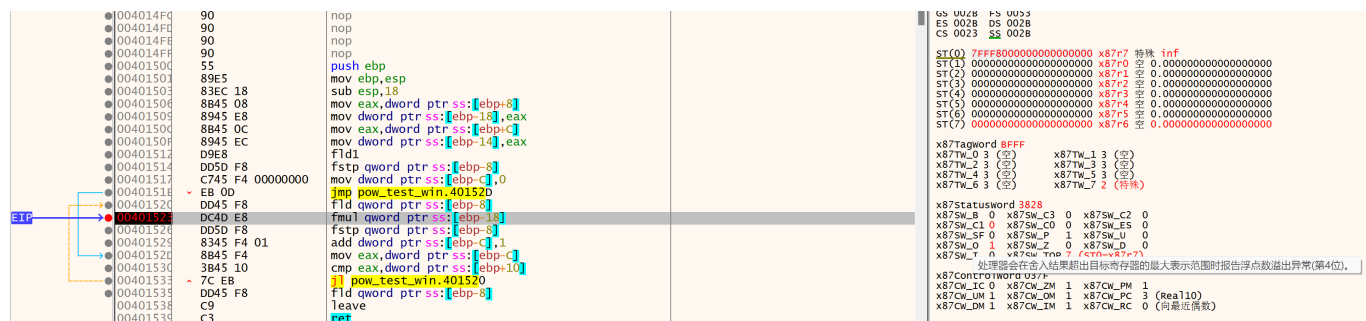
在进行幂运算 `pow(a, n)` 时，如果 n 值较大且 $a > 1.0$ ，那么 a^n 会在迭代过程中超过双精度浮点数 `double` 类型的表达范围上限，变为 `inf`。此时运算速度会发生骤降，比 $a = 1.0$ 要慢 100 倍以上。

问题追踪

这是一个朴素的幂运算函数：

```
double my_pow(double A, int N)
{
    double result = 1.0;
    for (int i = 0; i < N; i++)
        result *= A;
    return result;
}
```

使用 Dev-C++ 自带的 MinGW 4.9.2 编译器进行编译，配合 x32dbg 进行调试，进入 my_pow 函数体。



从汇编代码分析，其首先使用 `fld` 将上一次的运算结果压入浮点栈寄存器（右侧的 `st(0)` 至 `st(7)`），然后使用 `fmul` 进行乘法运算，最后由 `fstp` 将寄存器中的结果存入对应内存，并把结果从浮点栈寄存器弹出。

经过搜索，结合上面汇编代码的特征，发现编译器使用了 x87 指令集以及对应的寄存器 `st(0)` 至 `st(7)` 进行浮点运算。这是一个比较古老的指令集（名称源自于 Intel 8086 的 FPU 8087）。同时观察右侧标志位，出现了两个异常，`x87SW_O` 和 `x87SW_P`，分别表示浮点运算溢出和结果不精确。

由此引出了两个推测：

推测1: 与浮点数溢出异常的中断处理有关

这个推测意思是说，每次遇到浮点数溢出，CPU 就要立刻停下手中的活，保存当前的工作状态，跳转到一个处理浮点数溢出的函数里，处理完，再把原先的工作状态恢复，继续干活，由此引发的效率低下。

在 `fmul` 这条指令上打断点，修改 `st(0)` 内部存储的数字，让它大一点（要不然增长得太慢了），然后按 `F9` 运行，观察运行情况。

发现此时 `st(0)` 寄存器的数字不断变大，当变为 `inf` 时，下面的 `x87SW_O` 变为了 1。而 `x87SW_O` 是表示浮点数溢出异常的标志位，x32dbg 提示“处理器会在舍入结果超出目标寄存器的最大表示范围时报告浮点数溢出异常”。

那么这个异常会不会被处理呢？经过查询得知，由于触发处理器中断来处理异常的开销比较大，对性能有影响，所以默认是不处理的。可以看到，下面的 `x87CW_OM` 标志位是 1，表示不处理浮点数溢出异常。

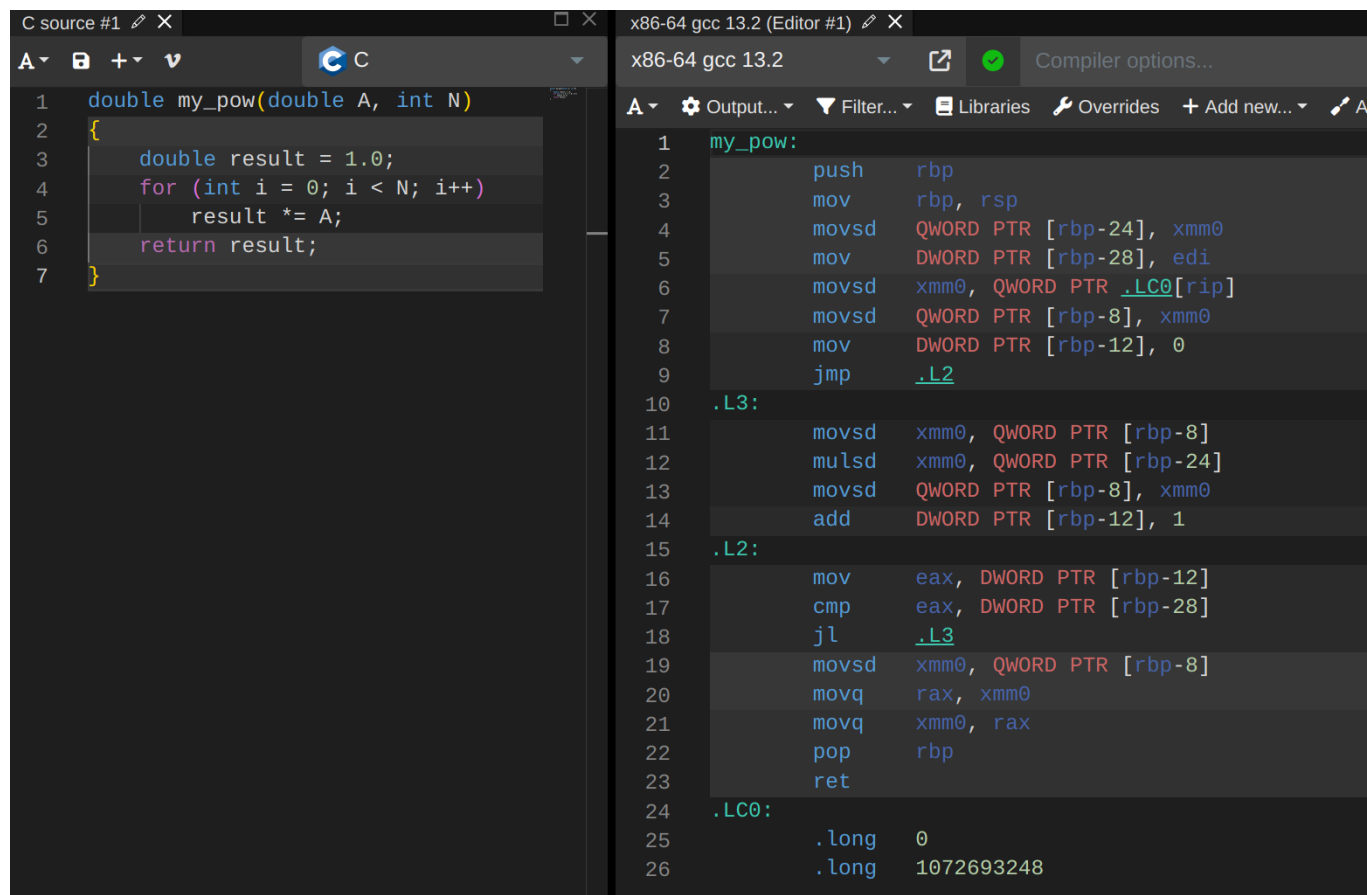
所以这个推测是错的。

（其实我还进行了一些测试，比如 $a = 1$ 时强行把 `x87SW_O` 变为 1，或者 $a = 1.2$ 时候强行把 `x87SW_O` 变成 0，都不影响运行的时间，后来发现这个异常并不会触发中断处理）

推测2: 与 x87 浮点数运算单元有关

这个推测来自于我最开始的尝试，一开始写好测试代码后，我用最新版 gcc 进行编译，发现当 $a = 1$ 和 $a = 1.2$ 时，二者的运行时间并无明显差异，即使加大运行次数 K 和指数 N 都一样，**只有在 Dev-C++ 里编译时，才会复现这个问题**。那么二者有什么区别呢？

使用 godbolt 来查看汇编：



```
C source #1 X
1 double my_pow(double A, int N)
2 {
3     double result = 1.0;
4     for (int i = 0; i < N; i++)
5         result *= A;
6     return result;
7 }

x86-64 gcc 13.2 (Editor #1) X
x86-64 gcc 13.2
Compiler options...
1 my_pow:
2     push    rbp
3     mov     rbp, rsp
4     movsd   QWORD PTR [rbp-24], xmm0
5     mov     DWORD PTR [rbp-28], edi
6     movsd   xmm0, QWORD PTR .LC0[rip]
7     movsd   QWORD PTR [rbp-8], xmm0
8     mov     DWORD PTR [rbp-12], 0
9     jmp     .L2
10 .L3:
11     movsd   xmm0, QWORD PTR [rbp-8]
12     mulsd   xmm0, QWORD PTR [rbp-24]
13     movsd   QWORD PTR [rbp-8], xmm0
14     add     DWORD PTR [rbp-12], 1
15 .L2:
16     mov     eax, DWORD PTR [rbp-12]
17     cmp     eax, DWORD PTR [rbp-28]
18     jl      .L3
19     movsd   xmm0, QWORD PTR [rbp-8]
20     movq    rax, xmm0
21     movq    xmm0, rax
22     pop     rbp
23     ret
24 .LC0:
25     .long   0
26     .long   1072693248
```

可以看到新版本的编译器会默认使用 SSE 指令集与 xmm 寄存器进行浮点运算。

经过搜索，发现 GCC 有一个编译选项 `-mfpmath` 表示浮点数运算所使用的运算单元，可以是 sse 或者 387(x87)。

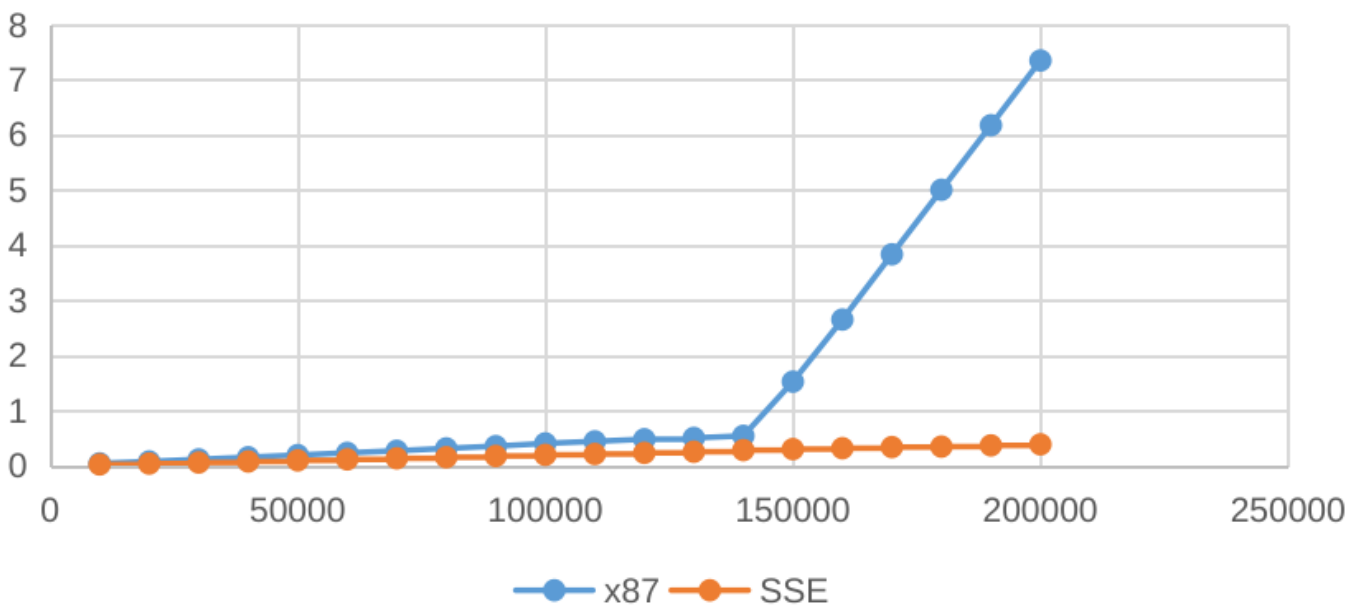
添加 `-mfpmath=387`，就会看到熟悉的汇编结果（注意到 `fld`, `fmul` 和 `fstp`）：

```
C source #1 X x86-64 gcc 13.2 (Editor #1) X
A+ + v C x86-64 gcc 13.2 -mfpmath=387
1 double my_pow(double A, int N)
2 {
3     double result = 1.0;
4     for (int i = 0; i < N; i++)
5         result *= A;
6     return result;
7 }

1 my_pow:
2     push    rbp
3     mov     rbp, rsp
4     movsd   QWORD PTR [rbp-24], xmm0
5     mov     DWORD PTR [rbp-28], edi
6     fld1
7     fstp    QWORD PTR [rbp-8]
8     mov     DWORD PTR [rbp-12], 0
9     jmp     .L2
10 .L3:
11     fld     QWORD PTR [rbp-8]
12     fmul    QWORD PTR [rbp-24]
13     fstp    QWORD PTR [rbp-8]
14     add     DWORD PTR [rbp-12], 1
15 .L2:
16     mov     eax, DWORD PTR [rbp-12]
17     cmp     eax, DWORD PTR [rbp-28]
18     jl      .L3
19     fld     QWORD PTR [rbp-8]
20     fstp    QWORD PTR [rbp-40]
21     mov     rax, QWORD PTR [rbp-40]
22     movq    xmm0, rax
23     pop     rbp
24     ret
```

接下来就可以进行测试了，在同一台电脑，同一个编译器，同一段代码的前提下，只改变浮点数运算所使用的指令集，结果如下：

Total run time of running
pow(1.005, N) 1000 times (sec)



在 N=150000 时，结果由 double 可以表示的数值转变为 inf（把 a 调小就是为可以观察到这个转变），此时发现 x87 指令集运算速率骤降，而 SSE 依旧保持线性增长。

所以这个推测是正确的。

于是，要想找到问题的成因，就可以着重分析 x87 和 sse 浮点运算的区别，更精确地，是 x87 和 sse 浮点运算单元对于 inf 参与的运算的区别。

我又设计了一个简单的自乘测试，一个初值为 1.0，另一个初值为 `__DBLMAX__`（double 类型能表示的最大数值，一乘就变 inf），分别进行足够多次，以测出单次运算的耗时。

以下是测试结果：

```
● → pow_test gcc -o fpu_test fpu_test.c -mfpmath=387
● → pow_test ./fpu_test
1.000000
Iteration: 5000000000
Total time: 1.087511062622s
Single multiplication duration: 2.175e-10s
inf
Iteration: 1000000000
Total time: 9.183956146240s
Single multiplication duration: 9.184e-08s
422.25x slower!
● → pow_test gcc -o fpu_test fpu_test.c -mfpmath=sse
● → pow_test ./fpu_test
1.000000
Iteration: 5000000000
Total time: 1.416250228882s
Single multiplication duration: 2.833e-10s
inf
Iteration: 2000000000
Total time: 3.884930849075s
Single multiplication duration: 1.942e-09s
6.86x slower!
```

这说明无论是 x87 还是 SSE，都在处理 inf（IEEE 754 中的非规格数 / subnormal number）上有明显的性能下降，而且 x87 慢了四百多倍！（但是 SSE 在绝对时长上相差得没那么多，所以在先前的测试中没有体现出来）

经过搜索，这篇博客：[That's Not Normal—the Performance of Odd Floats](#) 中对这个问题有如下的描述，和我的测试相符合：

Performance implications on the x87 FPU

The performance of Intel's x87 units on these NaNs and infinities is pretty bad. Doing floating-point math with the x87 FPU on NaNs or infinities numbers caused a 900 times slowdown on Pentium 4 processors. Yes, the same code would run 900 times slower if passed these special numbers. That's impressive, and it makes many legitimate uses of NaNs and infinities problematic.

Even today, on a SandyBridge processor, the x87 FPU causes a slowdown of about 370 to one on NaNs and infinities. I've been told that this is because Intel really doesn't care about x87 and would like you to not use it. I'm not sure if they realize that the Windows 32-bit ABI actually mandates use of the x87 FPU (for returning values from functions).

The x87 FPU also has some slowdowns related to denormals, typically when loading and storing them.

Historically AMD has handled these special numbers much faster on their x87 FPUs, often with no penalty. However I have not tested this recently.

对 x87 FPU 的性能影响（机翻）

英特尔的x87单元在这些NaN和无穷上的性能非常糟糕。在 NaN 或无穷大数字上使用 x87 FPU 进行浮点数学运算会导致奔腾 4 处理器的速度减慢 900 倍。是的，如果传递这些特殊数字，相同的代码运行速度会慢 900 倍。这令人印象深刻，它使NaN和无穷大的许多合法使用成为问题。

即使在今天，在 SandyBridge 处理器上，x87 FPU 在 NaN 和无穷大上也会导致大约 370 比 1 的速度变慢。有人告诉我，这是因为英特尔真的不关心x87，并希望你不要使用它。我不确定他们是否意识到 Windows 32 位 ABI 实际上强制要求使用 x87 FPU（用于从函数返回值）。

x87 FPU 也有一些与异常相关的减速，通常在加载和存储它们时。

从历史上看，AMD 在其 x87 FPU 上处理这些特殊数字的速度要快得多，通常不会受到任何损耗。但是我最近没有对此进行测试。

至于在硬件层面，inf 及 NaN 导致的性能下降的根本原因，我还不太清楚，估计也搞不明白（还没学，也没有相关的资料）。

根据老师们的讨论，在硬件层面上，bit 0 和 bit 1 的个数确实会影响运算速度。这个提示很有用，在测试中也的确如此：经过测试多轮 `inf * a` 的时间，发现除 $a = 1.0$ 以外，一些其他接近的取值都会导致性能下降，且性能下降的倍数和 a 的取值有一定关联。而且，浮点数与 1.0 相

乘的速度也比一般的浮点数相乘要快（所以之前的“自乘测试”不是很严谨，但是可以感受到差距）