Contents

[位操作优化 1](#_Toc35879988)

[英文字母大小写转换 1](#_Toc35879989)

[二进制乘法 1](#_Toc35879990)

[通用代码优化经验（C++） 1](#_Toc35879991)

[基本思想 1](#_Toc35879992)

[优化对象构造与复制 1](#_Toc35879993)

[优化数据选型和避免类型转换data type 1](#_Toc35879994)

[优化循环loop 1](#_Toc35879995)

[优化函数设计 1](#_Toc35879996)

[优化函数调用跳转 2](#_Toc35879997)

[优化分支跳转 2](#_Toc35879998)

[优化缓存和内存访问cache 2](#_Toc35879999)

[优化计算过程开销 2](#_Toc35880000)

[优化栈和堆的使用 2](#_Toc35880001)

# 通用代码优化经验（C++）

## 基本思想

·阿姆达尔定律Ahmdal’s Law，让经常执行的路径运行更加高效，而运行稀少的路径正确运行。

·代码先保证正确，然后再考虑优化。

·尽量多使用（标准）库内容和最新标准的功能，一般情况下，性能会更好，除非你十分了解库内容的不足和自己代码的优势。

·逻辑上可以提前计算化简的内容，不要放在程序中尤其是循环中执行。

## 优化对象构造与复制

·用初始化initialization代替赋值assignment，即使用构造函数的初始化列表代替构造函数内的成员赋值。

·上一条的另外一层含义是，使用带参数的构造函数直接初始化对象，而不是使用复制、赋值或拷贝等。

·减少复杂类型对象拷贝，使用移动语义解决或其它方式。

·避免不必要的数据初始化，如果你需要初始化一大段的内存，考虑使用memset。

·对于常用的简单类型，由于经常被复制，应使用足够简单的构造函数。

·必要时用自定义构造函数覆盖掉默认构造函数，以提升性能。

·推迟定义本地变量，定义一个对象变量通常需要调用一次函数（构造函数）。如果一个变量只在某些情况下需要（例如在一个if声明语句内），仅在其需要的时候定义，这样，构造函数仅在其被使用的时候调用。

·前置自增自减优化，用++i代替i++，若功能不变。使用后缀操作符需要执行一次对象拷贝（这也导致了额外的构造和析构函数调用），而前缀的构造函数不需要一个临时的拷贝。

·对大多数自定义类型，优先使用+= 、 -= 、 \*= 和 /=，而不是使用+ 、 -、 \* 、 和/。一连串的算术运算需要创建大量的匿名临时中间变量。例如：Vector v = Vector(1,0,0) + Vector(0,1,0) + Vector(0,0,1);?创建了五个匿名临时Vector: Vector(1,0,0), Vector(0,1,0), Vector(0,0,1), Vector(1,0,0) + Vector(0,1,0), 和 Vector(1,0,0) + Vector(0,1,0) + Vector(0,0,1)。对上述代码进行简单转换：Vector v(1,0,0); v+= Vector(0,1,0); v+= Vector(0,0,1);仅仅创建了两个临时Vector: Vector(0,1,0) 和 Vector(0,0,1)。这节约了6次函数调用（3次构造函数和3次析构函数）。

·上一条对于基本数据类型不适用，应尽量使用+、-、\*、/。

## 优化数据选型和避免类型转换data type

·整数和浮点数指令通常在不同的寄存器中进行操作，所以它们之间的类型转换需要进行一次拷贝操作。

·短整型（char和short）虽然只有半字长，但仍然使用较大的整个寄存器进行计算（如单字长或双字长），它们需要先被填充为32/64位，运算完毕后在存储回内存时需要再次转换为小字节。不过，考虑这个开销时需要权衡使用一个更大的数据类型的内存空间开销。

·在现代CPU，浮点数运算和整数运算差不多拥有同样的效率。在计算密集型应用，这意味这可以忽略整数和浮点数计算的开销差异。这也就是说，你不必要对算数进行整数处理优化。

双精度浮点数运算也不比单精度浮点数运算更慢，尤其是在64位机器上。同一台机器测试某算法全部使用double比全部使用floats运行有时候更快，当然有时也会更慢。

·复杂数据结构选型，选择最最合适的数据结构，避免选择功能相较过强的，浪费性能。

## 优化循环loop

·循环展开，对于n个循环的k级循环展开（一个循环里重复k次），可以令循环因子i < n – k + 1，再对剩余的循环进行单独操作即可。循环展开增加代码总量，可以降低循环处理开销，可以便于指令流水线化，但综合效果必须依赖实际情况。

·循环内可以移到循环外的语句（比如循环不变量）一定要移走，比如声明变量。

·考虑看看某些数值是否可以在循环中通过循环变量进行增量修改得到，而不每次都重新开始计算。

·for循环嵌套优化，在不影响功能的前提下，将循环次数越少的循环放在越靠外部的位置，因为这种布局下各循环的循环变量的实例化、赋值、比较、增量次数可以达到最低。若在for语句外提前先声明循环变量，能提高效率，但在循环次数少的时候并不明显，有时反倒会影响效率。

·寻找可以尽早结束循环的情况加以利用。

## 优化函数设计

·避免或减少使用本地变量以及函数参数，如果数量不多，处理器可以充分发挥寄存器传参数的性能优势（取决于调用惯例calling convention）。

·但也要避免和控制静态变量、全局变量的数量。

·参数传递尽量选择引用传递，而非值传递。

·尽量避免函数需要返回值的情况。

·寻找尽早返回函数调用的情况加以利用，避免执行额外的逻辑。

·使用表查找代替复杂函数，许多人都鼓励将复杂的函数（比如：三角函数）转化为使用预编译的查找表。但有时这会导致不必要的内存查找，这个代价很昂贵（并不断增长），并且这和直接计算一个三角函数一样快（尤其考虑到内存查找打乱了cpu的cache存取）。在其他情况下，查找表会很有用。对于GPU编程通常优先使用表查找而不是复杂函数。

## 优化函数调用跳转

·减少函数调用，但是必须平衡地保证可读性。

·用迭代代替递归。

·使用inline和define替换少量但多次出现的代码，避免调用函数的开销，同时也有利于指令并行（原因类似于循环展开），虽然会稍许增加代码总量。

## 优化分支跳转

·通过调整程序逻辑，减少（条件）分支，分支会影响指令流水线。

·逻辑条件优化，即&&运算符的短路功能方面的优化。

·对于大规模的if else分支链，要执行到最后的分支需要很多的跳转。如果可能，将其转换为一个switch声明语句，编译器有时候会将其转换为一个表查询单次跳转。如果switch声明不可行，将最常见的场景放在if分支链的最前面（具体应根据平台对分支预测功能的实现来处理）。

·适当的使用goto，很多书上基本都写过"禁止使用goto"这样的字眼，不过在某些特定情况下，适当使用goto能漂亮的解决问题。在kernel开发中，goto被用于“跳出多重循环”和“错误处理跳转”，使得逻辑更加清晰，只要满足"goto不跨函数"、"goto不向上跳转"的情况下goto还是有用武之地的。

·使用三元运算符?:替代简单结构的if else。（待汇编求证）

## 优化缓存和内存访问cache

·根据所用平台上的数据存储规则（如多维数组的存储顺序），进行数据的集中存储和处理，避免出现cache不命中。

·字节对齐，或cache行对齐。这需要根据具体平台而定。

## 优化计算过程开销

·平方根计算尽量避免，尤其是可以用平方结果进行比较时。

·如果重复地需要处理 除x 操作，考虑先计算1/x的值，乘以它。这在向量单位化（3维条件下需3次除法）运算中有很大的改进，不过有时也并非效果明显。如果要进行三次或更多除法运算，这还是可以利用的。

·更多数学计算优化需要参考位操作优化。

## 优化栈和堆的使用

·动态内存对于存储场景和运行期间其他数据都很有用。

·但是，在许多（大多数）的系统动态内存分配需要获取控制访问分配器的锁。对于多线程应用程序，现实中使用动态内存由于额外的处理器导致了性能下降，因为需要等待分配器锁和释放内存。

·即便对于单线程应用，在堆上分配内存也比在栈上分配内存开销大得多。操作系统还需要执行一些操作来计算并找到适合尺寸的内存块。

# 位操作优化

## 英文字母大小写转换

ASCII码大小写转换只需对第5位进行OR 1操作（+32）即可。

## 二进制乘法

一个无符号数左移 n 位就是将其乘以 2^n。其他任何乘数都可以表示为 2 的幂之和，实际上就是将乘数转化为二进制形式。若乘数的二进制形式中位k为1（位0为最低位），则将被乘数左移k位，最后将结果相加即为乘积。

线程调优：首先看自身线程模型的调用方式，避免多线程加锁，可以的话可以做线程本地化处理

线程本地化处理，优化

gprof和日志，profiling工具