预习作业1

2010139 张晏睿

一、 C语言编程优化

1.1实验结论

用指针遍历数组更优(LOOP2>LOOP1)

1.2设计思路

1.2.1实验环境

- OS：WSL2（Ubuntu20.04）
- editor：vim
- compiler：gcc、g++
- ISA：x86-64

1.2.2实验方案

a.不同数据类型

数组的类型有：int、float、double三种

b.不同数据规模

规模N从2^1~2^{20}按指数梯度依次递增

c.计时

对于较小规模，使用COUNT变量进行多次循环，取平均值

计时使用linux下gettimeofday，并使用微秒usec

d.初始化手段

三种方式：初始化为0，按下标初始化，随机数初始化

e.不同优化级别

分别进行编译器不优化、一级优化、二级优化、三级优化的测试

1.3实验结果

分三部分，分别对三种数据类型在不同**规模**、不同**优化级别**下进行测试

1.3.1初始化为0

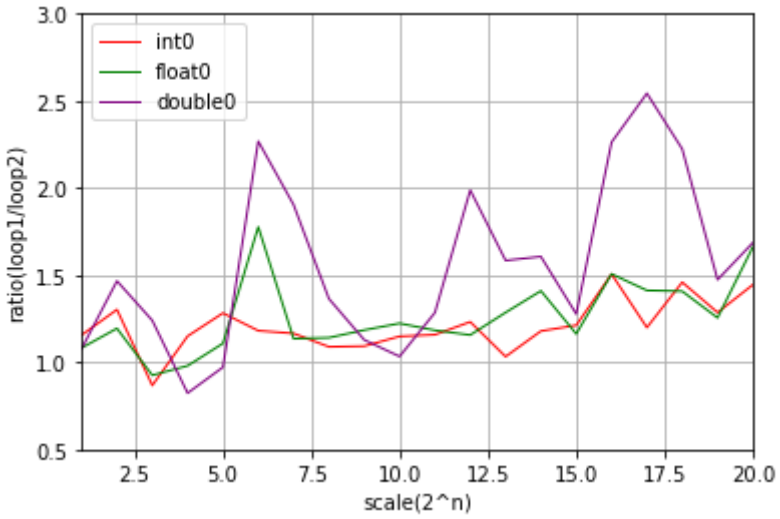
a.原始数据表

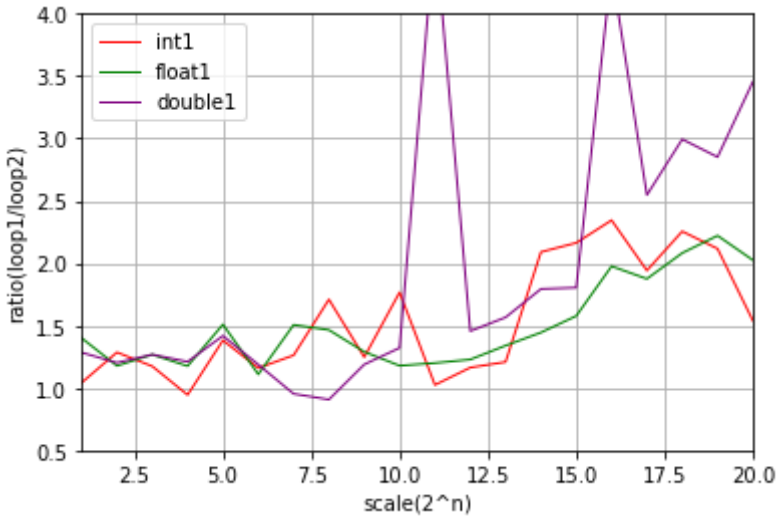
a[i]=0												
log2N	int0	float0	double0	int1	float1	double1	int2	float2	double2	int3	float3	double3
1\2	222/192	236/218	235/216	150/144	133/95	122/108	67/67	70/75	52/57	65/64	93/91	88/85
2\4	211/162	215/180	264/212	99/77	93/79	95/74	58/59	55/51	54/88	57/55	35/33	48/45
3\8	196/226	212/229	284/287	162/138	71/56	71/56	53/57	37/51	47/49	49/50	19/18	31/29
4\16	230/200	294/300	247/232	152/161	67/57	69/55	56/69	37/46	48/47	77/94	16/14	25/24
5\32	218/170	235/212	206/179	152/110	83/55	78/62	64/62	38/44	50/49	50/50	11/9	26/25
6\64	202/171	311/175	397/285	167/144	71/64	76/66	51/110	47/48	57/55	50/50	11/9	26/25
7\128	204/175	201/177	337/284	212/168	128/85	81/70	55/84	48/47	59/57	53/53	13/11	27/24
8\256	196/180	194/170	232/67	106/62	126/86	78/70	62/62	48/47	58/57	61/59	10/9	27/26
9\512	212/194	378/319	360/279	101/81	89/69	82/76	86/89	52/52	63/62	53/53	12/11	32/30
10\1024	193/168	219/179	185/193	106/60	66/56	74/69	48/59	47/47	57/115	58/47	13/11	82/27
11\2048	183/158	303/256	329/326	85/83	66/55	267/66	48/58	47/44	58/54	58/54	13/11	35/28
12\4096	207/168	192/166	330/244	121/104	70/57	83/70	51/59	62/47	66/58	51/48	17/11	266/33
13\8192	222/215	198/154	244/174	116/96	71/53	83/64	53/54	52/54	69/54	52/44	21/11	49/27
14\16384	250/212	258/183	294/229	163/78	91/63	113/76	71/65	68/52	96/64	69/53	35/14	72/33
15\32768	314/259	449/386	496/348	212/98	153/97	175/102	105/87	103/69	148/85	104/70	58/18	120/44
16\65536	360/239	556/369	835/403	204/87	164/83	361/158	142/88	133/82	421/87	137/71	156/24	198/43
17\131072	685/571	702/497	1264/1008	466/240	360/192	489/215	280/174	281/152	531/172	288/144	202/35	413/89
18\262144	1401/960	1395/991	2204/1064	830/368	687/330	988/418	576/348	711/337	892/355	952/551	441/73	1066/191
19\524288	3107/2416	3566/2841	4188/2378	1617/764	1466/660	1882/826	1303/721	1152/572	1899/1493	1238/586	768/154	1435/371
20\1048576	5636/3900	6893/4147	6992/4398	3239/2104	2705/1336	4616/2738	2279/1467	2187/1168	4616/2132	2224/1560	1353/287	3051/1264

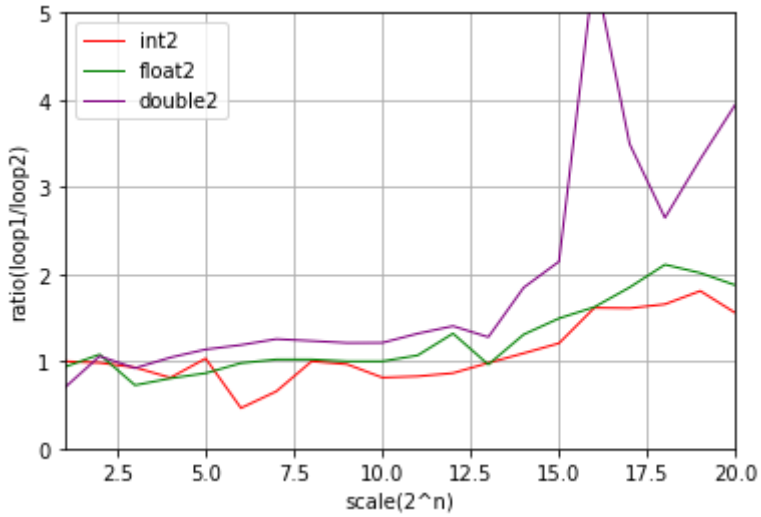
> 注：
>
> - 表格纵轴表示规模变化，横轴表示类型及优化等级的变化，如int0表示int类型数组，O0优化，double2表示double类型数组，O2优化
> - 对于小规模测试，都利用循环重复计数取均值(具体次数在1.3.2的表格中可以见到)
> - 时间单位是usec微秒

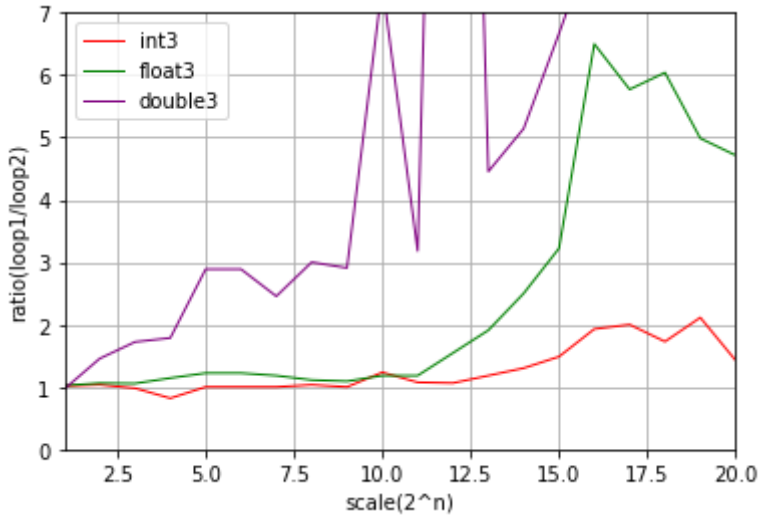
b.两种方式耗时比值曲线

纵轴是比率，是$\frac{T_{LOOP1}}{T_{LOOP2}}$，横轴是规模的对数，下面对**四种优化级别**分别进行比对









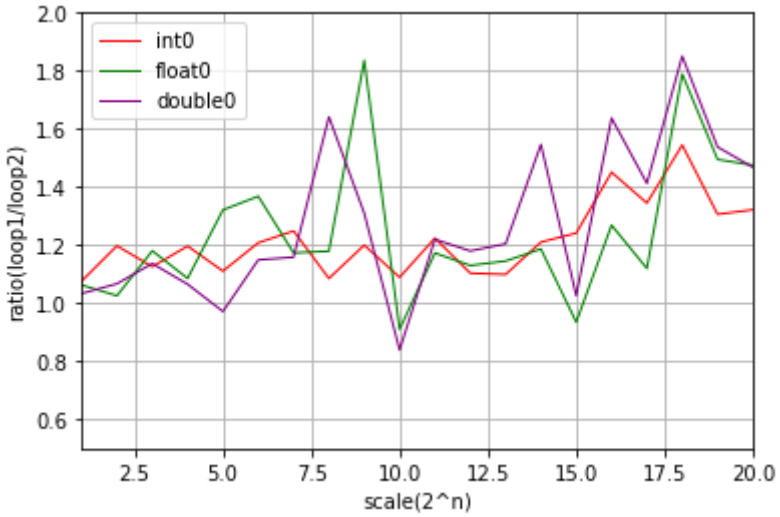
1.3.2初始化为下标

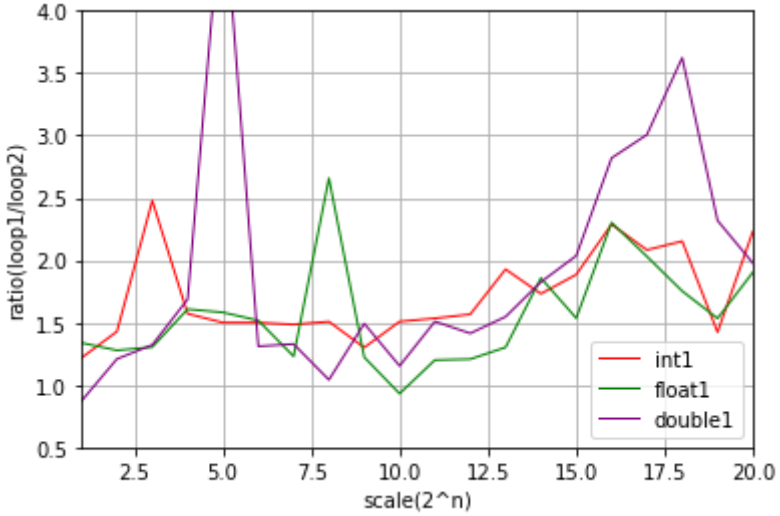
a.原始数据表

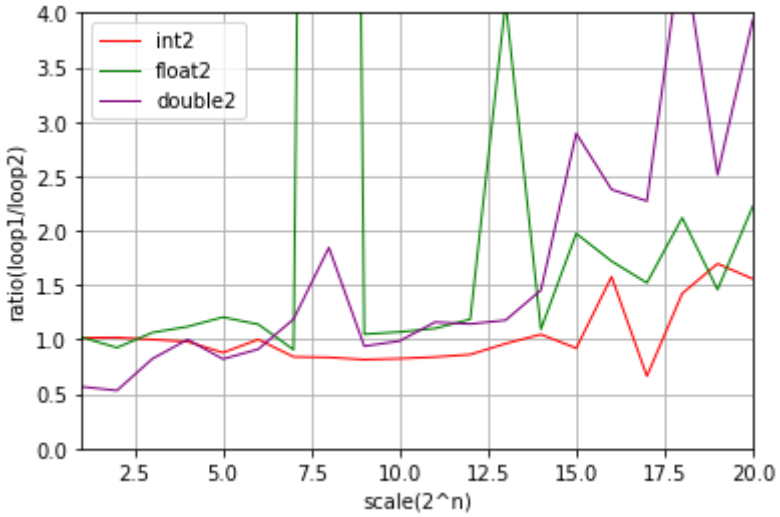
第一列(int0)第三部分表示循环的次数

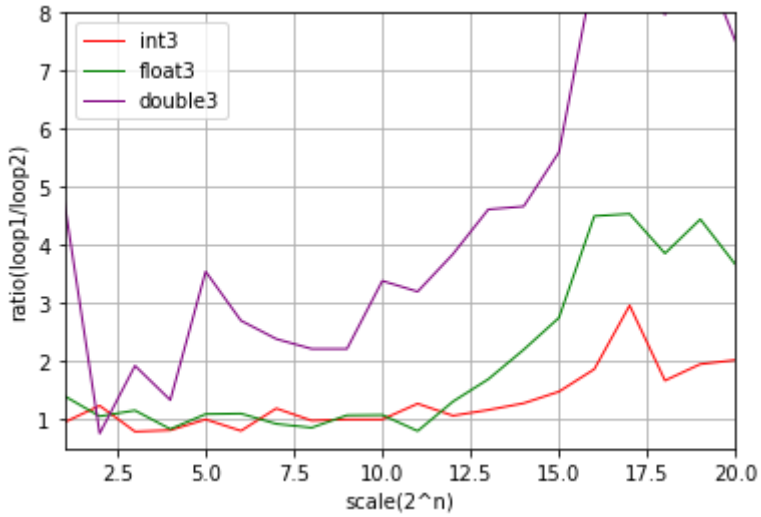
a[i]=i												
log2N	int0	float0	double0	int1	float1	double1	int2	float2	double2	int3	float3	double3
1\2	226/210/2523	255/240	248/253	134/110	210/157	137/89	65/64	99/97	55/63	92/97	73/52	247/235
2\4	206/172/11678	199/194	207/189	110/77	92/72	87/88	56/55	87/94	50/49	68/55	39/37	28/31
3\8	206/183/5699	216/183	208/187	166/67	73/56	74/68	50/50	84/79	65/47	49/62	15/13	25/23
4\16	213/178/2867	216/199	212/198	99/63	82/51	86/55	49/50	85/76	76/48	48/59	15/18	24/23
5\32	231/208/1501	325/246	239/200	96/64	87/55	275/77	51/58	94/78	64/71	50/50	12/11	39/25
6\64	238/197/739	275/201	231/204	93/62	88/58	76/75	59/59	90/79	72/67	49/61	11/10	27/26
7\128	246/197/358	237/202	234/218	98/66	75/61	81/71	53/63	87/96	114/86	63/53	12/13	31/30
8\256	229/211/177	237/201	330/203	95/63	194/73	76/71	51/61	2565/77	142/70	49/50	12/14	31/30
9\512	252/210/96	415/226	296/298	169/130	77/63	94/294	53/65	86/82	77/76	52/52	15/14	35/34
10\1024	220/202/43	243/267	224/197	89/59	67/72	83/68	47/57	77/72	71/68	46/46	14/13	44/30
11\2048	224/183/21	224/191	233/194	89/58	66/55	83/66	47/56	76/69	80/78	61/48	12/15	48/75
12\4096	224/203/11	226/200	236/202	94/60	70/58	82/69	50/58	83/70	80/86	50/47	17/13	50/37
13\8192	211/192/5	213/186	224/186	106/55	69/53	82/63	51/53	324/79	93/81	50/43	22/13	60/29
14\16384	259/214/3	267/225	348/218	116/67	117/63	115/76	67/64	116/106	154/130	69/54	33/15	70/35
15\32768	360/290/2	359/384	394/303	164/87	129/84	171/100	104/113	221/112	324/185	102/69	55/20	112/46
16\65536	428/295/1	392/309	506/299	199/87	191/83	234/99	134/85	196/114	271/102	138/74	90/20	175/46
17\13107	784/583/1	795/710	1003/593	404/194	335/165	496/199	272/410	342/225	511/211	465/157	177/39	344/92
18\26214	1784/1155/1	2181/1219	2256/181	805/374	680/388	1406/526	572/402	809/382	1773/703	549/329	351/91	725/188
19\52428	3217/2463/1	3624/2424	3728/2361	1626/1143	1509/984	2281/826	1182/697	1394/956	2403/1384	1165/597	729/164	1481/569
20\10485	6206/4965/1	7388/5008	7354/4835	3178/1424	3222/1693	3347/2007	2285/1468	3114/1400	5511/2464	2489/1231	1366/373	2798/1892

b.耗时比值曲线


int0
float0
double0
ratio(loop1/loop2)
scale(2^n)


int1
float1
double1
ratio(loop1/loop2)
scale(2^n)


int2
float2
double2
ratio(loop1/loop2)
scale(2^n)



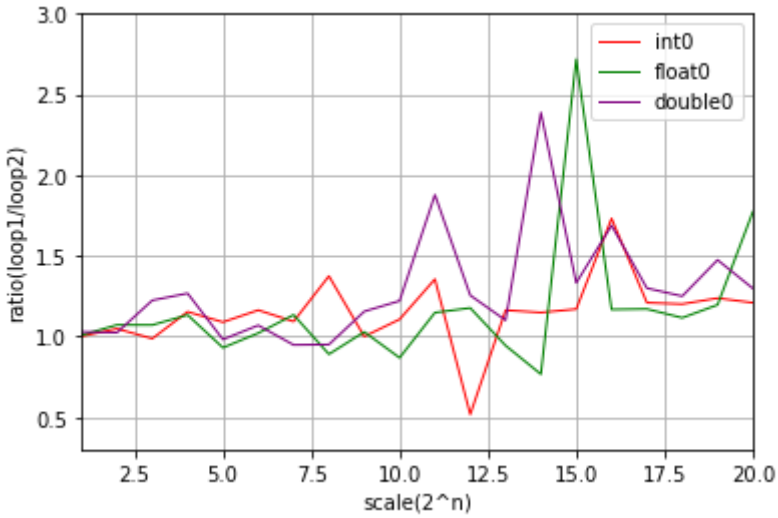
1.3.3初始化为随机数

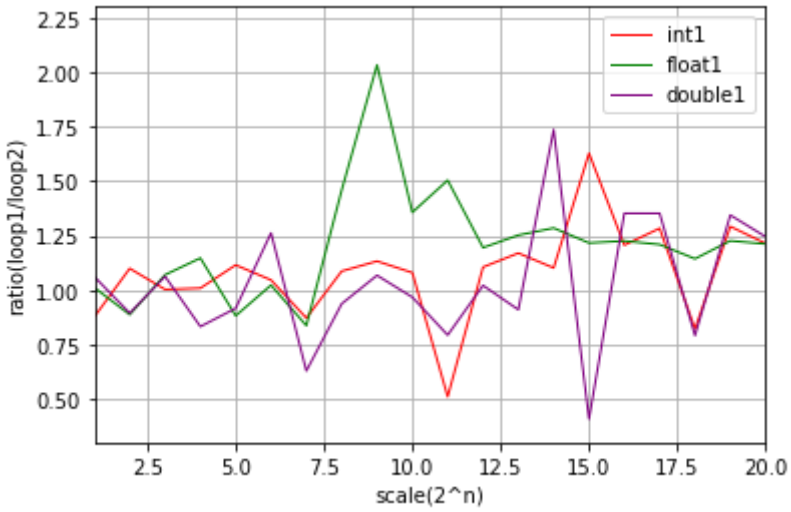
a.原始数据表

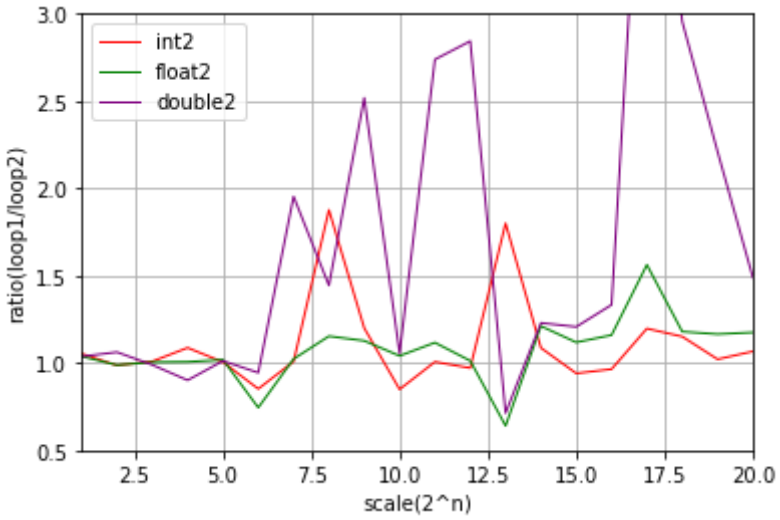
a[i]=rand()%97

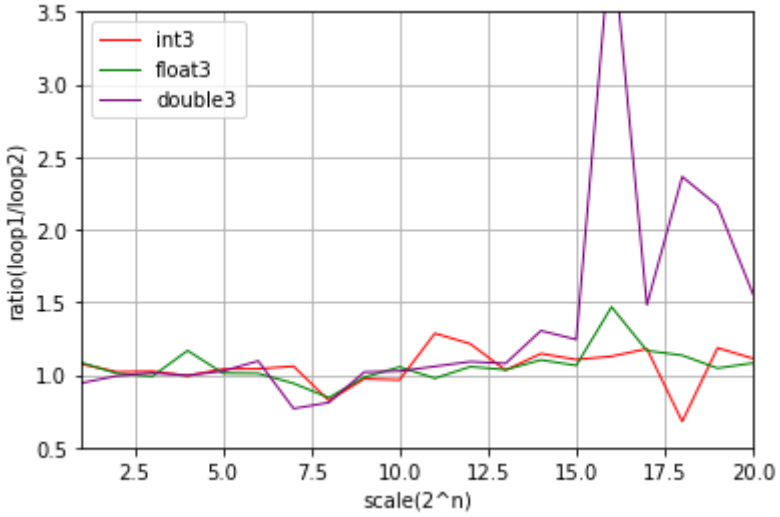
log2N	int0	float0	double0	int1	float1	double1	int2	float2	double2	int3	float3	double3
1\2	28720/28676	28159/27924	28729/27872	27729/31616	28043/27749	29428/28066	34143/32406	29047/27989	29036/29894	33544/31261	32759/30242	28544/28051
2\4	13837/13203	14155/13209	13525/13569	14881/13545	13208/14896	13311/13146	13008/13187	13048/13200	14021/13765	13672/13398	13194/13073	12975/14243
3\8	6664/6752	7056/6588	8064/9628	6573/6558	6842/6398	6795/6965	6602/6542	6543/6512	6443/6606	6686/6528	6475/6544	6632/6733
4\16	5234/4539	3917/3456	4380/3748	3483/3455	3902/3404	2832/3436	3749/3448	3846/3821	3444/3453	3424/3444	3980/3411	3397/3335
5\32	2446/2246	2048/2201	2161/235	2212/1985	1926/2185	2001/1947	2002/1991	1974/1937	1960/1964	2036/1958	1945/1924	1973/1886
6\64	1418/1219	1226/1201	1284/1292	1215/1162	1117/1093	1379/1156	1127/1323	1168/1569	1482/1708	1161/1115	1109/1099	1204/1075
7\128	893/817	1041/918	870/1095	743/853	954/1141	717/713	701/695	707/691	1350/1410	736/695	862/916	703/657
8\256	1369/996	608/683	649/644	542/499	872/598	560/512	942/502	623/540	780/512	547/665	503/597	483/483
9\512	566/566	543/528	610/788	480/424	941/463	494/462	558/465	487/432	1088/1098	422/434	407/414	422/409
10\1024	479/433	441/508	620/460	381/353	529/390	377/353	332/391	355/341	360/351	330/342	332/315	323/331
11\2048	557/411	442/385	723/536	330/647	624/415	329/332	302/300	363/325	890/827	387/301	287/294	311/291
12\4096	514/994	466/396	497/448	348/315	406/340	347/335	306/315	317/313	890/771	369/304	303/287	313/294
13\8192	407/350	385/408	449/372	325/278	428/342	311/497	486/270	288/450	321/289	291/282	264/255	275/262
14\16384	479/417	459/599	1430/1327	370/336	506/394	684/621	351/323	401/331	407/372	369/322	334/303	395/450
15\32768	648/554	1514/557	742/585	721/443	1628/1339	546/455	460/489	491/439	530/479	473/428	578/543	675/1199
16\65536	973/562	684/586	989/686	539/447	550/449	607/455	494/512	522/450	600/490	496/440	587/400	1623/1113
17\131072	1346/1113	1368/1169	1519/1154	1111/866	1072/887	1199/936	1098/917	1411/903	4280/3283	1019/864	960/823	1219/929
18\262144	2736/2278	2737/2453	3066/2320	2449/2983	3571/124	2470/1825	2205/1915	2074/1758	5197/1929	2013/2966	2116/1866	4412/751
19\524288	5601/4526	5564/4652	6859/4738	4444/3441	4389/3581	4811/3737	4178/4090	4179/3586	7928/6878	4095/3455	3804/3638	7877/7222
20\1048576	10871/8990	16713/9426	12233/9416	8886/7324	8970/7405	9213/7439	8408/7883	11842/10076	15008/7528	7687/6906	7250/6707	10449/10580

b.耗时比值曲线









1.3.4结果总结

通过横纵向比对，发现如下规律：

- 无论何种数据类型、何种初始化方法、何种优化级别，均是**LOOP2性能优于LOOP1**(用指针操作数组更快)
- 曲线都不是恒**稳定**的，对于初始化为0和下标的方法，LOOP2的性能有时会远大于LOOP1，表现出异常；而如果采取随机数初始化，这种异常出现的会少，曲线也更加稳定，即两个方法性能的差异比较稳定
- 随着**数据规模**增大，LOOP2和LOOP1的性能整体上没有过大变化，只是更加容易上下浮动

- 随着编译器**优化等级**的提升，LOOP2和LOOP1的性能会越来越靠近，即差距减小
- 整体来看，int型的曲线更为稳定

1.4实验分析

主要从**汇编代码**角度分析

1.4.1输出.s文件

```
g++ -E src.cpp -o src.i
g++ -S src.i -o src.s
```

1.4.2LOOP1与LOOP2对比

看看编译器分别做了什么(O0优化)(注：arr是我定义的数组名)

a.LOOP1运算过程

```
LOOP1：    cmpl    $20479, -4(%rbp)
    jg  .L9
    movq    arr(%rip), %rax      //基址放入rax
    movl    -4(%rbp), %edx       //-4存着i
    movslq  %edx, %rdx           //把i拿出来到rdx
    salq    $2, %rdx             //i*4
    addq    %rdx, %rax           //rax是arr基址，加上i*4
    movl    (%rax), %eax         //把rax地址中的给eax，eax就是arr[i]
    movq    arr(%rip), %rdx      //再拿一份arr旧地址
    movl    -4(%rbp), %ecx       //再拿一份i到ecx
    movslq  %ecx, %rcx           //ecx换到64bit的寄存器，用来存结果
    salq    $2, %rcx             //i*4
    addq    %rcx, %rdx           //再算一个arr[i]
    imull   $2000, %eax, %eax    //arr[i]*2000,这一份是第一次拿的放到eax的
    movl    %eax, (%rdx)         //rdx是第二份arr[i]，把结果eax存到里面

    movq    arr(%rip), %rax      //(下面就是除法了)拿一份基址
    movl    -4(%rbp), %edx       //拿一份i
    movslq  %edx, %rdx           //i扩展
    salq    $2, %rdx             //i*4
    addq    %rdx, %rax           //算arr[i]
    movl    (%rax), %eax         //放到eax
    movq    arr(%rip), %rdx      //拿一份基址
    movl    -4(%rbp), %ecx       //拿一份i
    movslq  %ecx, %rcx           //i扩展
    salq    $2, %rcx             //i*4
    addq    %rcx, %rdx           //算arr[i]放到rdx
    movslq  %eax, %rcx           //第三份arr[i]放到rcx
    imulq   $1759218605, %rcx, %rcx
    shrq    $32, %rcx            //取arr[i]，rcx的高32bit
```

b.LOOP2运算过程

```
LOOP2 :  cmpl    $20479, -12(%rbp)    //这是规模
    jg  .L13
    movq    -8(%rbp), %rax           //-8是指针指向的arr地址
    movl    (%rax), %eax             //取出arr[i]到eax
    imull   $2000, %eax, %edx        //乘2000, 存到edx
    movq    -8(%rbp), %rax           //把栈中基址存到rax
    movl    %edx, (%rax)             //运算结果存到rax指定的地址

    movq    -8(%rbp), %rax           //(下面就是除法了)栈中基址放到rax
    movl    (%rax), %eax             //rax中arr[i]放到eax
    movslq  %eax, %rdx               //eax扩展到rdx
    imulq   $1759218605, %rdx, %rdx
    shrq    $32, %rdx
    sarl    $12, %edx
    sarl    $31, %eax
    subl    %eax, %edx
    movq    -8(%rbp), %rax
    movl    %edx, (%rax)
    addq    $4, -8(%rbp)
    addl    $1, -12(%rbp)
    jmp .L12
```

c.从代码量上对比

- 运算过程(乘法或除法)，二者没有明显差别
- 差别体现在元素的存取访问中

d.进一步解释

- LOOP1需要从栈中取出i，乘4后再加到arr基址上，计算出arr[i]地址；这样的计算要**重复2次!**（一次是从arr[i]地址中取出数，进行乘法运算；第二次又计算了一遍arr[i]地址，用来把计算结果放进去）
- LOOP2不用从栈中取出 i 计算，而是直接用指针定位arr[i]，直接取出数计算即可，然后再根据指针把结果放回，不用计算基址的偏移量!

所以LOOP2性能优于LOOP1

e.浮点运算

运算过程是类似的，因此**结论相同**

1.4.3编译器优化

考虑不同优化等级下，**运算部分**的汇编代码(其他部分暂不考虑)

a.O1级别

LOOP1如下图：

```
.L12:
    leaq    0(,%rsi,4), %rdx            //leaq是mov的变种，计算了地址存到rdx
                                        //这里计算了i*4
    movq    %rdx, %rax                  //i*4移动到rax
    addq    arr(%rip), %rax             //基址加i*4，rax更新后就是arr[i]地址
    imull   $2000, (%rax), %ecx         //乘
    movl    %ecx, (%rax)                //结果存入rax表示地址里
    addq    arr(%rip), %rdx
    movl    (%rdx), %ecx
    movslq  %ecx, %rax
    imulq   $1759218605, %rax, %rax
    sarq    $44, %rax
    sarl    $31, %ecx
    subl    %ecx, %eax
    movl    %eax, (%rdx)
    addq    $1, %rsi
    cmpl    %esi, N(%rip)
    jg  .L12
```

LOOP2如下图：

```
    movq    arr(%rip), %rsi
    cmpl    $0, N(%rip)
    jle .L14
    movl    $0, %edx
.L16:
    imull   $2000, (%rsi,%rdx,4), %ecx
    movslq  %ecx, %rax
    imulq   $1759218605, %rax, %rax
    sarq    $44, %rax
    sarl    $31, %ecx
    subl    %ecx, %eax
    movl    %eax, (%rsi,%rdx,4)
    addq    $1, %rdx
    cmpl    %edx, N(%rip)
    jg  .L16
```

主要变动如下：

- 单独使用rsi寄存器存放i，使用leaq计算i*4存放到rdx，这样就跳过了从栈中取i、再计算的繁琐过程
- arr[i]不会被当作两个分别的操作数而被寻址两次，优化后只进行了一次arr[i]寻址，取出并计算结果后直接写回了原地址
- LOOP1中的乘法指令imull没有改变，但在LOOP2中编译器找到了更简洁的方式：直接使用mul的变种"imull $2000, (%rsi, %rdx, 4), %eax"，取i乘4寻址arr[i]后乘2000，一气呵成

b.O2/O3级别

到了O2/O3级别，LOOP1和LOOP2汇编代码就**相同**了，而且O3与O2也**相同**

相比于O1的变化是：LOOP1也是用了imull $2000, (%rsi, %rdx, 4), %eax, 至此把LOOP1优化成为了LOOP2的样式

1.4.4其他细节

a.除法的运算处理

值得一提的是，在int类型中的÷10000操作，并没有用除法指令，而是用乘法和减法实现的，LOOP1和LOOP2在这方面差别不大

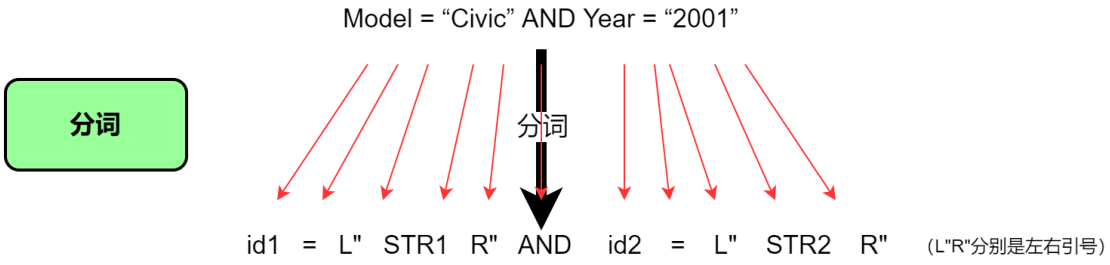
在float和double类型的汇编代码中，使用了div指令

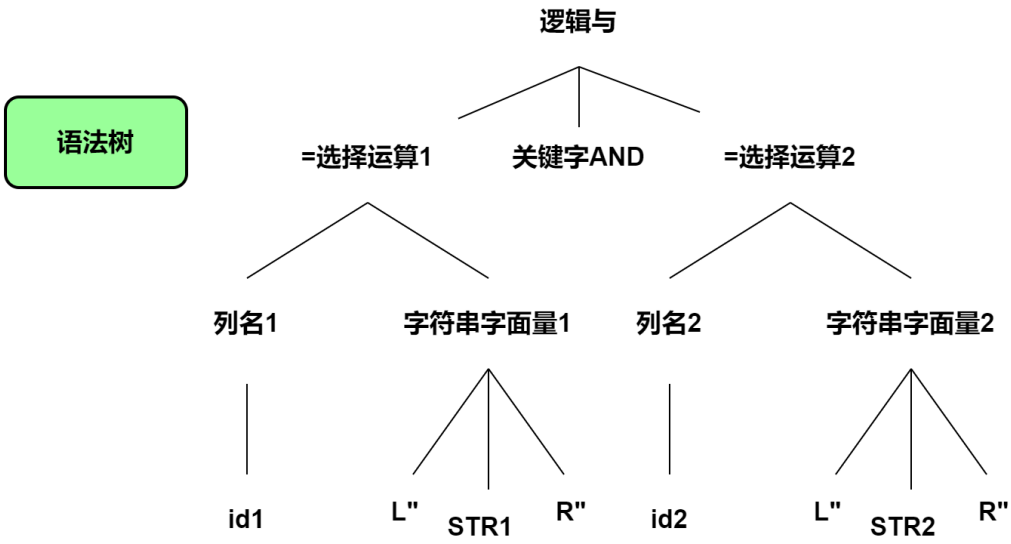
b.寄存器使用不同

int型，无论是arr[i]元素还是运算结果，都存放在eax、ecx等寄存器中

而浮点型，浮点数都存放在浮点寄存器xmm中

二、分词&语法树





三、静态检查

使用splint工具，在ubuntu20.04环境下测试该文件，发现了7处警告：

```
splint.c: (in function f)
splint.c:7:9: Stack-allocated storage &loc reachable from return value: &loc
  A stack reference is pointed to by an external reference when the function
  returns. The stack-allocated storage is destroyed after the call, leaving a
  dangling reference. (Use -stackref to inhibit warning)
```

```
splint.c:7:9: Immediate address &loc returned as implicitly only: &loc
  An immediate address (result of & operator) is transferred inconsistently.
  (Use -immediatetrans to inhibit warning)
splint.c:7:14: Stack-allocated storage *x reachable from parameter x
   splint.c:6:2: Storage *x becomes stack-allocated storage
splint.c:7:14: Function returns with global glob referencing released storage
  A global variable does not satisfy its annotations when control is
  transferred. (Use -globstate to inhibit warning)
   splint.c:7:9: Storage glob released
splint.c: (in function h)
splint.c:11:5: Comparison of unsigned value involving zero: i >= 0
  An unsigned value is used in a comparison with zero in a way that is either a
  bug or confusing. (Use -unsignedcompare to inhibit warning)
splint.c:11:5: Variable i used before definition
  An rvalue is used that may not be initialized to a value on some execution
  path. (Use -usedef to inhibit warning)
splint.c:1:6: Variable exported but not used outside splint: glob
  A declaration is exported, but not used outside this module. Declaration can use static qualifier. (Use
-exportlocal to inhibit warning)
```

问题归结 如下：

1. **全局指针**glob指向了**局部变量**loc，局部变量在栈上，函数f调用结束后会销毁，所以glob在执行完f后会变为**野指针**，指向不安全的未知地址
2. 返回&loc是**立即寻址**的方式，它不能保证地址对齐、规整，可能返回没有意义的结果
3. 使用了栈内存x，x是参数，调用f前入栈，虽然参数值是相同的但是参数的地址是不同的(这里的x有一个栈地址)，因此对于x的解引用不能得到程序外实参的地址值
4. 返回地址是全局的glob指向的**释放的**内存，不能完成标注的控制权转移
5. **无符号**变量是**一定**大于等于0的，此时或者出现bug，或者是迷惑行为...
6. i在定义前被使用了
7. 全局指针glob声明但未使用

四、标识符列表递归定义

对于 标识符列表 ：

```
1    int a,b,...,x;
```

递归定义如下：

def.

1.基本规则

若type是变量类型，a是一个 标识符 ，则在声明语句

```
1    type a;
```

中，a构成一个**标识符列表**，且a是**最基本**的标识符列表

2.递归定义

若b是一个 标识符列表 ，c是一个 标识符 ，在语句

```
type b,c;
```

中，二者用**逗号连接**

则"b，c"也组成一个**标识符列表**