软件系统优化 P1

10211900416 郭夏辉

实验目的

这个project我的目标是什么？这个问题要从实验文档中的代码开始（之后我将其命名为了MatrixMultiplication.c）：

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <assert.h>
#define n 4096
double A[n][n];
double B[n][n];
double C[n][n];
float tdiff(struct timeval *start, struct timeval *end) {
    return (end->tv_sec-start->tv_sec) + 1e-6*(end->tv_usec-start->tv_usec);
}
int main(int argc, const char *argv[]){
    assert(argc==2);
    int s = atoi(argv[1]);
    if (s < 1 || s > 4096) {
        printf("Invalid input values.\n"); return -1;
    }
    for (int i=0; i<n; ++i){
        for (int j = 0; j<n; ++j){
            A[i][j] = (double)rand() / (double)RAND_MAX; B[i][j] = (double)rand() /
(double)RAND_MAX; C[i][j] = 0;
        }
    }
    struct timeval start, end;
    gettimeofday(&start, NULL);
    for (int ih = 0; ih < n; ih += s)
        for (int jh = 0; jh < n; jh += s)
            for (int kh = 0; kh < n; kh += s)
                for (int il = 0; il < s; ++il)
                    for (int kl = 0; kl < s; ++kl)
                        for (int jl = 0; jl < s; ++jl)
                            C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jh+jl];
    gettimeofday(&end, NULL);
    printf("%0.6f\n", tdiff(&start, &end));
    return 0;
}
```

这个程序进行了什么样的工作呢？有两个大小为$n*n$的随机矩阵（里面的元素都是随机的）进行矩阵乘法。在这个实验中n被固定为了4096,然后程序以输入的参数作为分块的大小，分开来进行矩阵的乘法操作。这个程序最后会得到进行矩阵乘法的总时间。

学习过《CSAPP》的我们都知道不同的步长会对程序的运行效率产生显著的影响（这里埋下一个坑，后续会探究这个的原因），那么我们给上述程序不同的输入参数，就能够得到不同的运行时间；与此同时，不同的编译优化等级也会让程序的运行效率产生明显的差异。

这次实验，就是要求我们设计并实现一个程序性能的自动调优器AutoTuner,它能够在块大小和编译优化等级这两种参数的各个组合中找到性能最佳的。并且，这个调优器还有以下几个要求：

- 明确输入的目标程序（这个实验中当然我的是MatrixMultiplication.c）、要配置的参数组合、参数搜索算法
- 要配置的参数组合（实验要求）：分块大小：8, 16, 32, 64, 128五种情况；编译优化等级：O0, O1, O2, O3四种情况
- 参数搜索算法一种为Grid Search,另外的可以自己去设计
- 可以参考OpenTuner的实现。但是根据助教学长在2023.10.18实验课上的强调，最好还是自己来设计并实现。

设计和实现

程序框架

调优器我是用python实现的，这主要是因为它灵活的语法及丰富的库。在实验过程中，我从最开始的想法到最终实现过程中查阅了很多方法的用法，这里主要来选一些经典的地方讲一讲。

1. ArgumentParser()解析命令

AutoTuner的运行始于我们特定的输入,这也是实验的基本要求。为了方便调试，我们尽量还是希望程序的输入参数是在命令行中传入的，这里我利用了ArgumentParser()来解析命令，读入参数。

```
from argparse import ArgumentParser
    parser = ArgumentParser()
    parser.add_argument('-t', type=str, default='MatrixMultiplication.c', help='target')
    parser.add_argument('-b', type=str, default='8,16,32,64,128', help='block sizes')
    parser.add_argument('-o', type=str, default='O0,O1,O2,O3', help='optimization
levels')
    parser.add_argument('-m', type=str, default='grid', help='search method')
    parser.add_argument('-i', type=int, default=10, help='max iterations in random
search')
    parser.add_argument('-T', type=int, default=3, help='repeat times to avoid
deviation')
    args = parser.parse_args()
```

读入之后直接调用args的相关成员（比如args.t）就能获得相应的参数，十分方便。

2. 利用check_output()运行相应的shell命令

为了在不同的编译优化级别下编译MatrixMultiplication.c生成相应的可执行文件，gcc命令我一定是躲不过了。想要在程序中指定去执行相应的外部命令，OS课程中我学过C语言中的exec方法；但是在python中我利用的则是check_output方法。注意这里字符串的拼接。

```
from subprocess import check_output
for opt in opts:
    compile_cmd = check_output(['gcc', '-' + opt, target, '-o', './matrix' +
opt],shell=False)
```

这里的compile_cmd还包括执行外部命令的返回结果呢（是以byte串形式，要转化成正式的文本，还要decode一下，这里自己出了点小小的bug）

3. 使用面向对象让代码可读性增强

代码的实验并没有花费我很多的时间，这主要得益于自己使用了面向对象形式来设计AutoTuner类，可读性提高了很多。这里只放一个大致的类框架：

```
class AutoTuner:
    def __init__(self,target,blocks,opts,method,iter,times):
        self.target = target
        self.blocks = blocks
        self.opts = opts
        self.method = method
        self.iter = iter
        self.times = times
    def run(self):
        # 1 编译目标c语言程序(各个优化级别)
        for opt in self.opts:
            compile_cmd = check_output(['gcc', '-' + opt, self.target, '-o', './matrix' +
opt],shell=False)
        # 2 选择特定的调优程序运行
        if self.method == 'grid':
            self.grid()
        elif self.method == 'random':
            self.random1()
        elif self.method == 'fire':
            self.fire()
    def grid(self):
        # grid search: 朴素搜索
        pass
    def random1(self):
        # random search: 随机搜索
        pass
    def fire(self):
        # 模拟退火
        pass
if __name__ == "__main__":
    # other code
    autotuner = AutoTuner(args.t,args.b.split(','),args.o.split(','),args.m,args.i)
    autotuner.run()
```

有了以上的框架，我只用具体地考虑并实现每种方法就行了。

Grid Search

网格搜索就像是sklearn的GridSearchCV一样，老老实实地遍历了所有可能的参数组合，然后找到各个组合中效率最高的那一个作为最终结果。

因为实在是乏善可陈，程序的实现如下所示：

```
    def grid(self):
        # grid search: 朴素搜索
```

```
        result = []
        time_st = time.time()
        for block in self.blocks:
            for opt in self.opts:
                sumtime = 0
                for i in range(self.times):
                    run_cmd = check_output(['./matrix' + opt, block], shell=False)
                    sumtime += float(run_cmd.decode().strip())
                sumtime = round((sumtime/self.times),3)
                resultdict={
                    'block':block,
                    'optimize':opt,
                    'Time':sumtime,
                }
                result.append(resultdict)
                print(f'blocksize={block} optimize={opt} time={sumtime}')

        time_ed = time.time()
        print(f'Average Sum Time={round(((time_ed - time_st)/self.times ) ,3)}s')
        result.sort(key=lambda x: x['Time'])
        print('best params:',result[0])
```

Random Search

首先我们要明确一个问题，就是我们要随机抽取的是一个参数的组合，而尽量不要一类参数随机抽一个、另一类参数再随机抽一个。构建两种参数的组合的方法是利用itertools.product构建并使用笛卡尔积，等概率随机抽取（因为我们是不放回抽样，每次概率的分母会发生变化，这里要注意）采用random.choice方法即可。

其次我们要深知，随机搜索是比较"碰运气"的，比如说原始的取值空间大小为N,我们抽取的样本数量是M,那么我们最终抽到的这M个解中的最优解是整体的最优解的概率是$\frac{1\cdot C_{N-1}^{M-1}}{C_N^M}=\frac{M}{N}$.如果$M$取太大，就退化到了朴素的搜索；如果$M$取太小，就很难搜索到真正正确的结果。

那么，如果我们从概率角度做一个小折中如何？我们最终抽到的这M个解中的最优解是整体的第K优解的概率是$\frac{1\cdot C_{N-K}^{M-1}}{C_N^M}=\frac{C_{N-K}^{M-1}}{C_N^M},N\geq K+M-1$

假设这是一个关于M的函数，根据高中数学组合数的知识，我们解不等式便可以得到组合型分式的极值点：

$f(M)=\frac{C_{N-K}^{M-1}}{C_N^M}$

$f(M)\geq f(M+1),f(M)\geq f(M-1)$

可得$M\geq\frac{N+1}{K}-1$且$M\leq\frac{N+1}{K}$

我们取不到最好的，取第二好的也是可以接受的，取$K=2$,这样M可以近似于$\frac{N}{2}$,在这个实验中我让$M=10$。

```
    def random1(self):
        # random search: 随机搜索
        time_st = time.time()
        result = []
        iters = []
        for i in itertools.product(self.blocks, self.opts):
```

```
            iters.append(i)

        for i in range(self.iter):
            randselect = random.choice(iters)
            iters.remove(randselect)
            opt = randselect[1]
            block = randselect[0]
            sumtime = 0
            for j in range(self.times):
                run_cmd = check_output(['./matrix' + opt, block], shell=False)
                sumtime += float(run_cmd.decode().strip())
            sumtime = round((sumtime/self.times),3)
            resultdict={
                'block':block,
                'optimize':opt,
                'Time':sumtime,
            }
            result.append(resultdict)
            print(f'blocksize={block} optimize={opt} time={sumtime}')

        time_ed = time.time()
        print(f'Average Sum Time={round(((time_ed - time_st)/self.times ) ,3)}s')
        result.sort(key=lambda x: x['Time'])
        print('best params:',result[0])
```

模拟退火

这个问题本质是在一个多维的解空间中搜索的最优化问题。结合曾经在数学建模中的经验，我按捺不住心中的想法把模拟退火算法实现了一遍。我采用的是常用的Metropolis准则，即如果新解更好（$\Delta = newtime - nowtime \leq 0$），接受；否则，以概率$exp\{-\frac{\Delta}{T}\}$来接受这个新解。针对本次实验，我特别地设置了初始温度、终止温度和降温率，为的就是不让迭代的次数过分庞大以至于退化到朴素搜索。

同时，我最初设计的模拟退火算法的效率甚至还不如随机搜索算法，这就十分具有戏剧性。经过深入的检查，我发现自己对于下一个可能用来迭代的点设置错了——理论上来说应该是当前点的邻接点，而不能是取值空间中的任意点，只有这样才能避免"反复横跳"情况的发生，才能使得优化算子朝着最优解不断逼近。计算邻接结点，需要注意边界情况。并且为了索引的使用之效率，我对其进行了预处理操作。有一点需要注意，模拟退火迭代过程中的每一点之间，可能存在很多探测了但是并没有选择的中间点，这些中间点的探索也花费了一些时间。

最后就是对于已经算出来的时间，就没有必要再花费一些时间等待结果了。利用"记忆化搜索"的思想，我直接将结果存入了一个列表中方便访问。

```
    def neighbor(self):
        # 这个函数可以很方便地获取(x,y)的紧邻结点(不包括自身且考虑了边界效应)
        # 以这个实验的默认数据为例，第一行 8,0 8,1 8,2 8,3 第二行16,0 16,1 16,2 16,3
        # (x,y) 索引表示是 4x+y x从0开始，y也是. 每行的block是一样的，每列的优化级别是一样的。
        # 注意这里的4即len(self.opts)要根据实际情况变化
        a = len(self.blocks)
        b = len(self.opts)
        self.nei = [[] for _ in range(a*b)]
```

```
        for x in range(a):
            for y in range(b):
                tmp = b*x + y
                if x==0:
                    self.nei[tmp].append(tmp+4)
                elif x + 1 == a:
                    self.nei[tmp].append(tmp-4)
                else:
                    self.nei[tmp].append(tmp+4)
                    self.nei[tmp].append(tmp-4)

                if y==0:
                    self.nei[tmp].append(tmp+1)
                elif y + 1 == b:
                    self.nei[tmp].append(tmp-1)
                else:
                    self.nei[tmp].append(tmp+1)
                    self.nei[tmp].append(tmp-1)

    def fire(self):
        # 模拟退火
        self.neighbor()
        iters = []
        for i in itertools.product(self.blocks, self.opts):
            iters.append(i)
        iterbook = [[] for _ in range(len(self.blocks)*len(self.opts))]

        for i in range(len(self.blocks)*len(self.opts)):
            for j in self.nei[i]:
                iterbook[i].append(iters[j])

        #print(iterbook)
        T = 1
        alpha = 0.5
        T1 = 0.001

        time_st = time.time()
        result = [0 for _ in range(len(iters))]
        nowselect = random.choice(iters)
        tmpopt = nowselect[1]
        tmpblock = nowselect[0]
        tmptime = 0

        for j in range(self.times):
            run_cmd = check_output(['./matrix' + tmpopt, tmpblock], shell=False)
            tmptime += float(run_cmd.decode().strip())
        nowtime = round((tmptime/self.times),3)
        result[iters.index(nowselect)] = nowtime
        print(f'blocksize={tmpblock} optimize={tmpopt} time={nowtime}')

        while T > T1:
            newselect = random.choice(iterbook[iters.index(nowselect)])
            if(result[iters.index(newselect)] == 0):
```

```
                newopt = newselect[1]
                newblock = newselect[0]
                sumtime = 0
                for j in range(self.times):
                    run_cmd = check_output(['./matrix' + newopt, newblock], shell=False)
                    sumtime += float(run_cmd.decode().strip())
                newtime = round((sumtime/self.times),3)
                result[iters.index(newselect)] = newtime

            else:
                newtime = result[iters.index(newselect)]
                newopt = newselect[1]
                newblock = newselect[0]

            diff = newtime - nowtime
            if diff < 0 or random.random() < math.exp(-diff / T):
                print(f'blocksize={newblock} optimize={newopt} time={newtime}')
                nowselect = newselect
                nowtime = newtime

            T*=alpha
        time_ed = time.time()
        print(f'Best: blocksize={nowselect[0]} optimize={nowselect[1]} time={nowtime}')
        print(f'Average Sum Time={round(((time_ed - time_st)/self.times ) ,3)}s')
```

结果与分析

1.运行结果

Grid Search的运行结果如下所示：

优化\块大小	8	16	32	64	128
O0	300.126	228.423	218.383	210.745	209.817
O1	85.342	59.066	55.732	48.028	49.616
O2	84.147	56.719	54.485	44.757	51.459
O3	75.99	49.367	42.145	37.393	35.797

最佳参数及运行时间：O3,128,35.797s

总的运行时间：1997.537s

Random Search的运行结果如下所示(括号中为每步所在位置)：

优化\块大小	8	16	32	64	128
O0	null	null	217.096(5)	null	null

优化\块大小	8	16	32	64	128
O1	84.875(2)	null	null	49.23(7)	51.634(6)
O2	84.074(4)	null	null	null	null
O3	76.402(3)	50.042(9)	41.162(1)	37.665(8)	36.521(10)

最佳参数及运行时间：O3,128,36.521s

总的运行时间：734.107s，耗时仅为Grid Search的36.75%

模拟退火的运行结果如下所示(括号中为每步所在位置)：

优化\块大小	8	16	32	64	128
O0	null	null	null	null	null
O1	null	null	null	null	null
O2	83.115(1)	56.594(2)	55.495(3)	null	null
O3	null	null	42.398(4)	37.187(5)	36.029(6)

最佳参数及运行时间：O3,128,36.029s

总的运行时间：501.004s，耗时仅为Grid Search的25.08%

综上所述，最佳参数是O3,128。但是有一个值得注意的现象是第一名和第二名差距并不是很大，这再次加深了我random search中"期望上取不了第一取第二也行"的想法。

2.性能可变性与解释

在理论课程中，老师提到了"可变性"这个概念。在性能测量时，可变性具体表现在多次性能测量的结果存在波动。面对自己系统中实打实存在着的干扰，我也采取了一些措施：

1. 从系统的角度出发，我在对应的搜索算法运行过程中，关闭了其他正在运行着的工作。虽然这并不能完全地达到"系统静默"状态（因为运行着的Windows操作系统还有数不胜数的后台服务），但是尽量得到了稳定的测量结果(经过计算，后两种算法对应的各个参数之运行时间与第一种之间的**余弦相似度**分别为0.999940,0.999937,十分接近于1，可以采信)
2. 从测量的角度出发，我在选定了每个参数时都进行了多次测量（这里我默认是3次测量）取平均值的方法消除了干扰，使得实验得到的数据更加精确。当然这里如果重复的次数多，实验消耗的时间就会增长。（最终数据来源于3次测量，最初实现程序时我是1次测量方便调试）但是多次测量真的就有很大的意义吗？经过实际测试，单次测量和多次测量最终的结果会有一定的差异，但是在5%的幅度以内，十分微小。（这里有一个小小的遗憾是我没有采用数理统计中的假设检验方法，只是通过简单的数值运算来验证这一假设）而且严谨地说，自己最终求得的"平均总时间"并不是严格意义上"重复1次，运行完一轮算法"的时间，因为中间会有一些其他的过程造成时间上的消耗。但是，整个运行过程中占绝对主导地位的时间消耗是在运行矩阵乘法环节的，直接去作平均，和真实的结果相差不大，可以采信。

3.理论性的分析

3.1 对块大小的探究

针对三种算法的结果，我来探究一下块大小对结果的影响。这让我想起了《CSAPP》中经典的memory mountain，那里也是在探讨缓存、步长对性能的影响。

MatrixMultiplication.c中一个数组大小就是$4096 * 4096 * 8bytes = 128MB$，根本无法全部放到cache中，大部分还是被放在了DRAM中等待读写。频繁的cache读写，就涉及到了命中问题，而设置的步长对命中率起到了至关重要的影响。在这个问题中，块大小可以认为是一种步长的等价表示。

1. 块小，内部循环（加载矩阵B）的局部性很差，cache不命中率高，内存中的元素在cache中频繁地换进换出，类似于"抖动"现象。
2. 块一般大，内部循环的局部性得以改善，cache不命中率下降，效率提升；
3. 块大，外部循环（加载矩阵A）的局部性变差，cache不命中率高，效率回落。如果块大到超过了L1-cache、L2-cache等的大小，会出现memory mountain中"断崖式下跌"的情况。

因此我们需要不能盲目地取一个"越大越好"的分块大小，具体的情况还要结合实验、电脑的配置，综合地选择一个最佳分块大小，使程序地运行效率最大化。在这个实验中，由于给的最大块大小仅$128 * 8bytes = 1KB$且我的电脑中cache还是比较大的，所以并没有出现"块大反而效率下降"的情况，自己并没有条件来复现这一理论，还是有点遗憾的。

3.2 对编译优化等级的探究

对于编译优化等级，可以看到"等级越高，时间越短"的趋势。通过查阅资料，我来梳理一下各个等级的编译优化到底是在干什么：

1. O0:干脆不做任何优化；
2. O1:在不是特别影响编译时间的情况下，尽量缩短代码的执行时间。就这个程序而言，矩阵乘法过程中的乘法和加法运算会被优化以减少运算量；
3. O2:可以让编译时间提升，其他和O1比较类似；
4. O3:最高程度的优化等级，有全部的优化选项。比如以编译时间、二进制可执行文件的大小换取最佳的性能。因为我对编译原理相关的知识不甚了解，这里我只能猜想它可能是通过了一些"空间换时间"的操作来提升效率。

从O0到O1的优化程度是十分显著的——从无到有，优化了的和"傻傻的"程序效率一定会出现很明显的效果。之后的优化就展现出了"边际效应递减"的特点:O1到O2几乎没什么改进，甚至还有退步迹象（出力在编译上提升，但效率上并不讨好）.但是最后从O2到O3,毕竟是开启了最高程度的优化，所有的优化选项都能为编译器所用了，在效率上的提升还是肉眼可见的（当然也并没有那么显著）

3.3 算法的优劣分析

从整体上来说，我想谈一下自己设计和实现AutoTuner调优器的优劣所在。

优点：1. 结构简单，可拓展性相对较好。在一个较为明确的框架下，如果想要添加新的算法，只需实现对应的类中方法即可；2.考虑了性能可变性对于测量的影响，采用多次测量的方式理论上来说消除了一些误差，并且算法运行时停下了手头的操作，尽可能避免了外来的干扰。

缺点：1.opentuner采用了多线程，我的算法都是单线程。这种设计导致自己的算法并没有很好地利用计算机资源，效率普遍存在很大提升空间； 2.可供搜索的参数有限。我实现的调优器只供搜索两种参数（即块大小和编译优化等级），不像opentuner那样能实现多类参数的查找。

然后再来具体地对比三种算法的优劣：

朴素搜索 Grid Search:

优点：因为是参数取值空间全都搜索一遍，排除误差干扰，最终一定是能找到最优解；设计的是真的简单，直接的多层循环嵌套便是该算法的框架。

缺点：时间复杂度十分夸张。因为一点都不tricky，我们不得不把每一个可能的参数都搜索一遍。

随机搜索 Random Search:

优点：相对朴素搜索来说tricky了一些，跳过了很多取值空间点的运算，极大地节省了运行时间。

缺点：1.在算法运行过程中，两点之间的迭代是完全的随机，并不遵循一个科学的准则，时间上的开销主要取决于随机选到的点；2.根据我前面在理论上的分析，随机搜索只能说**有概率**得到最优解。运气不好的话，可能迭代了很多很多次都没有到全局最优解。

模拟退火:

优点：相对随机搜索算法，迭代具有明确的方向性，并不是完完全全的随机，最终取得全局最优解的概率大大提高。（可以认为是Random Search的提升版）

缺点：1.需要设置好初始温度T，终止温度T_1,降温率α等超参数，否则模拟退火算法会因为迭代次数太多退化到朴素搜索；2.算法的运行很看重第一个点的选取情况，还是比较"看运气"。如果初始点抽取的并不是很好，之后的迭代过程依然很多，退化到了随机搜索；3.其实模拟退火算法的执行更适合在一个连续的取值空间上，这里的取值空间是离散的，表现上可能没想象中那么好。

3.4 可能的优化思路杂谈

其实在做实验时我天马行空地想了一些思路，这些思路我并没有实现，但是放在这里以供探讨：

1. 分块搜索。可以看到，就是调优器主要的时间开销是矩阵乘法的运算过程，面对$O(n^3)$的矩阵乘法，如果真的像题目中给出的$n = 4096$还是十分耗时的。因此我们不妨让n小一点，比如$n = 2048$情况下得到一个规律，由于原理是类似的，以小见大，在n更大的情况下规律应该也是适用的；
2. 贪心算法。虽然取值空间是离散的，不能照搬连续曲面的梯度下降法，但是以此为参考，从当前点出发，找一个相邻的、下降幅度最大的点作为下一个取值点也是可行的。但是这样每次要把周边的点的运行效率都算一遍，可能得不偿失。
3. 定一移一法。这个主要来源于理论课老师讲到的"domain knowledge"(领域知识)，我们在实验前就能通过别人的实验得出一个经验——无论是什么优化等级，效率都是比没有优化的O0要高的。在此基础上，我们只需要"定一移一"——固定某一个特征A，在此基础上去求得一个表现最好的特征B;接下来固定此特征B，在此基础上求得一个表现最好的特征A。但我觉得这个思路先验地认定了极值点与两种特征都具有单调性，并不能保证最终求得比较正确的结果，这显然是不合适的。

总结

整体上来说，这次实验我有很大的收获。自我设计并实现AutoTuner地过程中有很多的思考，也面对了来自性能测量、程序设计的诸多问题。虽然最终成功跑出了结果，并且与理论知识相呼应，但是碍于实验条件的限制，很多更进一步的措施并没有落实，比如多线程提高运算效率、彻底系统静默（绑定某个进程到某处理器）、验证小cache对最终结果的差异。