P2：Matrix Multiplication Autotuner

杨浩然 1019550144
华东师范大学

2021 年 12 月

目录

1 摘要 **1**

2 算法设计 **2**
2.1 评价指标 . . . 2
2.1.1 程序性能评价指标 . . . 2
2.1.2 算法性能评价指标 . . . 2
2.2 算法实现 . . . 2
2.2.1 网格搜索算法 . . . 2
2.2.2 随机搜索算法 . . . 3

3 代码实现 **4**

4 结果分析 **5**

5 部署方式 **6**

A 代码附录 **7**
A.1 autotuner.py . . . 7
A.2 parser.py . . . 11

1 摘要

本项目实现了一个简单的程序自动调优器 Autotuner，可以自动测试 Matrix Multiplication 程序的可配置参数，并根据性能选择最优配置输出。程序是基于 Python 进行编写的，兼顾了程序的低耦合性和可读性。调优算法的实现上，选择了两种较为经典的算法，即网格搜索 (grid search) 和随机搜索 (random search)。性能指标的选择上，将程序的运行时间定为指标，以评估不同参数配置的优劣。根据实验，

2 算法设计

2.1 评价指标

这里的评价指标有两个含义，一是评价不同参数配置下，待调优程序的性能，二是评价不同调优算法的性能。

2.1.1 程序性能评价指标

在 Matrix Multiplication(以下称 matrix-multiplication.c) 中已经给出计算程序运行时间的模块。所以我们将不同参数配置下，程序的运行时间作为性能评价指标。考虑到单次实验的随机性，我们在同一参数配置下多次 (本文实验中为 10 次) 运行程序，取平均值作为性能评价指标。

2.1.2 算法性能评价指标

这里总共采用了两种参数搜索算法，即网格搜索算法和随机搜索算法。由于这两种算法的实现方法，考虑空间复杂度是几乎没有意义的，以算法运行时间作为评价指标，有一定的意义。

此外，对于运行时间的测量，不易达到高度的精准，所以这里的算法性能评价指标更多是宏观上 “定性”，而非精确地 “定量”。

2.2 算法实现

2.2.1 网格搜索算法

网格搜索算法 (grid search) 是指定参数配置的一种穷举搜索方法，在所有候选的参数配置选择中，通过循环遍历，尝试每一种可能性，将表现最好的参数配置作为最终的结果。

对于维度较低、规模较小的参数配置，网格搜索是有效的。但参数的组合较多时，网格搜索将受到自身时间复杂度的限制而无法使用了。

即使网格搜索的实现是轻松的，这里仍给出它的伪代码。

算法 1 网格搜索算法

输入: 参数集合 $X = \{x_1, x_2, \cdots, x_m\}$, $Y = \{y_1, y_2, \cdots, y_n\}$; 待调程序 f; 性能评价指标 η.

输出: 最优参数配置 $(\hat{x}, \hat{y})$; 最优性能 $\hat{\delta}$.

$\hat{\delta} = -\infty$
$(\hat{x}, \hat{y}) = (x_1, y_1)$
for i = 1 to m **do**
    for j = 1 to n **do**
        $\delta = \eta(f(x_i, y_j))$
        if $\hat{\delta} < \delta$ **then**
            $\delta = \eta(f(x_i, y_j))$
            $(\hat{x}, \hat{y}) = (x_i, y_j)$
        end if
    end for
end for
return $(\hat{x}, \hat{y}), \hat{\delta}$

2.2.2 随机搜索算法

粗略来说，可以认为随机搜索算法 (random search) 与网格搜索算法是相似的。网格搜索算法遍历了网格上的每一个点，而随机搜索算法只是尝试了部分点。显然，由网格搜索得出的参数配置一定是该参数空间内全局最优的，但对于随机搜索来说，并不能保证全局最优。随机搜索算法牺牲了一定的准确度，换取了算法时间上的减少。

针对本文描述的问题，找到一个概率分布来对参数进行刻画是不实际的，所以在对参数进行随机抽样时，最好考虑一个不放回的等概率抽样。

下面仍给出算法的伪代码。

算法 2 随机搜索算法

输入: 参数集合 $X = \{x_1, x_2, \cdots, x_m\}$, $Y = \{y_1, y_2, \cdots, y_n\}$; 待调程序 f; 性能评价指标 η; 迭代次数 k.

输出: 最优参数配置 $(\hat{x}, \hat{y})$; 最优性能 $\hat{\delta}$.

$\hat{\delta} = -\infty$
$(\hat{x}, \hat{y}) = (x_1, y_1)$
for iter = 1 to k **do**
    依据均匀分布抽取 x', y'
    $X = X - \{x'\}$
    $Y = Y - \{y'\}$
    $\delta = \eta(f(x', y'))$
    if $\hat{\delta} <$ **then**
        $\hat{\delta} = \delta$
        $(\hat{x}, \hat{y}) = (x', y')$
    end if
end for
return $(\hat{x}, \hat{y}), \hat{\delta}$

3 代码实现

程序主要可以概括为两个功能，一是解析用户输入的命令行，二是按照解析出的参数配置执行算法。

在 parser.py 文件中基于 argparse 库实现了 Parser 类，该类用于解析用户输入的命令行。在 autotuner.py 文件中实现了网格搜索算法和随机搜索算法，将算法的结果输出到屏幕和文件中。

代码有详尽的注释，请参见附录 A 或代码文件。

4 结果分析

下面分别给出网格搜索算法的结果和随机搜索算法的结果：

opt level / block	O0	O1	O2	**O3**
8	334.6243	336.3172	341.7421	303.8557
16	262.7485	261.5145	261.5709	261.5868
32	245.7064	245.3797	245.4461	245.7257
64	241.1185	241.0958	241.5703	**241.0119**
128	243.3211	241.7159	242.1411	243.6011

表 1: 使用网格搜索算法，不同参数组合下 matrix-multiplication.c 的运行时间 (s)

opt level / block	O0	O1	**O2**	O3
8	301.1128	306.3723	null	null
16	null	261.4827	264.7079	262.6215
32	245.4708	null	null	null
64	null	null	**240.9468**	241.5422
128	243.3081	null	242.2343	null

表 2: 使用随机搜索算法，不同参数组合下 matrix-multiplication.c 的运行时间 (s)

对于每一种参数组合，程序运行 5 次后取运行时间的平均值，以减少噪声或随机性对结果测量的影响。根据实验结果可以得到，使用网格搜索算法时，最好的参数配置为 (O3,64)，即编译优化级别选择 O3，循环分块大小选择 64，此时 matrix-multiplication 的运行时间为 241.0119s；使用随机搜索算法时，最好的参数配置为 (O2,64)，即编译优化级别选择 O2，循环分块大小选择 64，此时 matrix-multiplication 的运行时间为 240.9468s。

注意到，(O0,8) 和 (O1,8) 的参数组合下，使用网格搜索算法和随机搜索算法得到的 matrix-multiplication 运行时间相差较大。这是因为运行网格搜索算法的同时，我在看电视剧，而运行随机搜索算法时我在床上睡觉。所以，在进行实验时，还是需要保持一个相对纯净的环境（不过在本实验中，“看电视剧”对实验结果没有太大影响，因为最优参数配置绝不可能是 O0,8 或 O1,8）。

对于算法的评价，本文使用运行时间作为性能指标。网格搜索算法的运行时间约为 26442.15s，随机搜索算法的运行时间约为 13065.57s。由于随机搜索算法只随机选择了 10 种参数配置，而网格搜索算法遍历了一共 20 种可能的参数配置，所以随机搜索算法的运行时间约为网格搜索算法的一半。

可以看出，随机搜索算法以精度的牺牲换取了运行效率的提高，并且这种精度的缺失是完全可以接受的，它与网格搜索得到的全局最优解的差异低于 0.1s。

此外，注意到实验结果表现出一定的误差，这个误差与理性是相悖的。即，已知网格搜索算法得出的最佳参数配置（全局最优）为 (O3,64)，那么，如果随机搜索算法抽取了 (O3,64)，随机搜索算法得出的解应一定也为 (O3,64)。但实验结果表明，随机搜索算法选择了更低的优化等级，即 (O2,64)。本文认为，这个现象是可解释的。两种算法中，O2 和 O3 在优化上的差异是微小的，所以 (O2,64) 和 (O3,64) 的优劣评估容易受到外部条件的扰动。如果要获得相当精确的结果，还可以在降低噪声和时间测量上进行优化。当然，在这里，本文认为这个误差是可以接受的，对于实验整体的可信度没有造成影响。

5 部署方式

1. 安装需要的 Python 包

```
pip install -r requirements.txt
```

2. 用下面的命令来运行 AutoTuner

```
python autotuner.py --file matrix-multiplication.c --opt o0,o1,o2,o3 --blk 8,16,32,64,128 --alg
    grid,random
```

3. 结果会在终端中打印出来，也会输出到当前文件夹中的 result.txt 中

A 代码附录

A.1 autotuner.py

```
'''
autotuner.py

功能：主程序，也是和用户交互的 interface
通过 Paser 类对用户的 command 进行解析，然后调用对应的参数搜索算法进行调优

优化过程包含两个超参数，一个是运行时的循环分块大小(blocksize)，一个是编译时的优化等级(optimization level)
对于这个问题来讲，可以在不同的优化等级下直接先将文件编译出来，每次只需调整循环分块大小，而不用多次编译文件

这里实现了两种参数搜索算法，一种是 grid search，另一种是 random search。
其实 random search 在这里有点不合时宜，因为参数的维度并不高，取值也不多，引入随机化算法得到最优解的概率相对较低。
'''

import itertools
import random
import time
from parser import Parser
from subprocess import PIPE, Popen, call


class AutoTuner:
    # 解析 command
    def __init__(self):
        argparser = Parser()
        self.paras = argparser.parse()
        self.paras["blk"] = self.paras["blk"].split(",")
        self.paras["opt"] = self.paras["opt"].split(\newpage
\section{部署方式}",")
        self.paras["alg"] = self.paras["alg"].split(",")

        self.grid_search_time = None
        self.grid_best = None
        self.grid_run_time = None
        self.random_search_time = None
```

```
        self.random_run_time = None
        self.random_best = None

    # 在不同优化等级下编译
    def compile(self):
        for opt in self.paras["opt"]:
            compile_cmd = ["gcc", "-"+opt,
                           self.paras["file"], "-o", "test"+opt]  # 编译命令
            if call(compile_cmd):   # 编译
                raise Exception("{} compile error!".\newpage
\section{部署方式}format(compile_cmd))
            # self.grid_search_size[opt] = os.path.getsize("test")   # 获取文件大小

    # grid search
    def grid_search(self):
        self.grid_search_time = {}  # 记录不同参数组合下, 程序的运行时间
        # self.grid_search_size = {}  # 记录不同参数组合下, 程序的大小
        grid_begin_time = time.time()  # grid search 算法开始时间
        repeat = 1  # 重复次数
        for blk in self.paras["blk"]:
            for opt in self.paras["opt"]:
                run_time = 0
                for _ in range(repeat):
                    p = Popen(["./test"+opt, str(blk)], stdout=PIPE)  # 带参数运行
                    run_time += float(str(p.communicate()[0])[2:-3])  # 获取运行时
                        间
                self.grid_search_time[blk+"," +
                                      opt] = round(run_time / repeat, 4)  # 记录平
                                         均用时

        self.grid_best = min(self.grid_search_time.items(),
                             key=lambda x: x[1])  # 最优参数组合
        grid_end_time = time.time()  # grid search 算法结束时间
        self.grid_run_time = round(
            grid_end_time - grid_begin_time, 4)   # grid search 算法运行时间

        print("(grid search)execution time with different parameter combinations:\n
            ",
              self.grid_search_time)
        print("(grid search)best parameter combination: ",  self.grid_best[0],
              " ("+str(self.grid_best[1]), "s)")
```

```
        print("(grid search)algorithm run time:", self.grid_run_time, "s\n")

    # random search
    def random_search(self):
        self.random_search_time = {}  # 记录不同参数组合下, 程序的运行时间
        iterations = 10  # 随机次数
        repeat = 1  # 重复次数
\newpage
\section{部署方式}
        # 计算所有参数组合
        paras_combinations = []
        for i in itertools.product(self.paras["opt"], self.paras["blk"]):
            paras_combinations.append(i)
        random_begin_time = time.time()  # random search 算法开始时间
        for iter in range(iterations):
            # 没有一个分布可以刻画参数, 所以用均匀分布进行不放回的抽样
            paras_selected = random.choice(paras_combinations)   # 选一组参数
            paras_combinations.remove(paras_selected)    # 不放回
            opt_level = paras_selected[0]
            blocksize = paras_selected[1]

            run_time = 0
            for _ in range(repeat):  # 运行 5 次, 取平均时间
                p = Popen(["./test"+opt_level, blocksize],
                          stdout=PIPE)  # 带参数运行\newpage
\section{部署方式}
                run_time += float(str(p.communicate()[0])[2:-3])  # 获取运行时间
            self.random_search_time[blocksize+"," +
                                    opt_level] = round(run_time / repeat, 4)  # 记
                                       录平均用时

        self.random_best = min(self.random_search_time.items(),
                               key=lambda x: x[1])  # 最优参数组合
        random_end_time = time.time()  # grid search 算法结束时间
        self.random_run_time = round(
            random_end_time - random_begin_time, 4)  # random search 算法运行时间
        print("(random search)execution time with different parameter combinations
            :\n",
              self.random_search_time)
        print("(random search)best parameter combination: ", self.random_best[0],
              " ("+str(self.random_best[1]), "s)")
```

```
112         print("(random search)algorithm run time:", self.random_run_time, "s\n")
113 
114     # 程序入口\newpage
115 \section{部署方式}
116     def run(self):
117         self.compile()
118         for alg in self.paras["alg"]:
119             if alg == "grid":
120                 self.grid_search()
121             elif alg == "random":
122                 random_begin_time = time.time()
123                 self.random_search()
124                 random_end_time = time.time()
125                 random_run_time = round(random_end_time - random_begin_time, 4)
126         # 将结果写入到文件
127         with open('result.txt', 'w', encoding='utf-8') as f:
128             grid_result = "grid search result\n"+"all combinations: " + \
129                 str(self.grid_search_time)+"\nbest parameter combination: " + \
130                 str(self.grid_best[0])+"\nbest time: " + \
131                 str(self.grid_best[1])+"\n"
132             random_result = "random search result\n"+"all combinations: " + \
133                 str(self.random_search_time)+"\nbest parameter combination: " + \
134                 str(self.random_best[0])+"\nbest time: " + \
135                 str(self.random_best[1])
136             f.write(grid_result+"\n"+random_result)
137 
138 
139 if __name__ == "__main__":
140     autotuner = AutoTuner()
141     autotuner.__init__()
142     autotuner.run()
```

A.2 parser.py

```
'''
parser.py

功能：解析 command

由作业要求可知，command 包含三个信息：
（1）输入的目标程序；
（2）输入的配置参数值组合；
（3）输入的参数值搜索算法。

parser.py 将 command 解析为不同的部分，并且返回给调用程序。

允许用户输入如下的参数：
--file: filename - 输入的目标程序
--blk: parameter - 输入的参数值组合
--opt optimization - 输入的优化等级组合
--alg: algorithm - 输入的参数值搜索算法
'''

import argparse


class Parser:
    def __init__(self):
        self.parser = argparse.ArgumentParser(
            description="Input your filename, blocksize combinations, optimazation
                level combinations and search algorithm.")
        self.parser.add_argument("--file", type=str, help="specify your file")
        self.parser.add_argument("--blk", type=str, default="32",
                                 help="specify blocksize combinations")
        self.parser.add_argument("--opt", type=str, default="00",
                                 help="specify optimazation level combinations")
        self.parser.add_argument("--alg", type=str, default="grid",
                                 help="specify your search algorithm")

    def parse(self):
        return vars(self.parser.parse_args())
```