嵌入式系统作业八

21307077 凌国明

1. 作业 1

1.1 **任务描述.** 使用 C#, Python 实现下述两段代码逻辑，测量二者的运行时间并分析结果。

```csharp
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    0 个引用
    internal class Program
    {
        0 个引用
        static void Main(string[] args)
        {
            DateTime time1 = DateTime.Now;
            double result;
            for (int i = 0; i < 100000; i++)
                result = Math.Sqrt(i) + Math.Pow(i, 2);
            DateTime time2 = DateTime.Now;
            TimeSpan ts = time2 - time1;
            Console.Write(ts.ToString());
            Console.ReadKey(true);
        }
    }
}
```

图 1. C# 代码

```python
import datetime
import math

beforeCal = datetime.datetime.now()
for number in range(100000):
    result = math.sqrt(number) + math.pow(number,2)
afterCal=datetime.datetime.now()
calCost = afterCal - beforeCal
print(calCost)
```

图 2. python 代码

```
for (int i = 0;i < 100000; i++)
    result = Math.Sqrt(i) + Math.Pow(i, 2) ;
```

Listing 1. 代码逻辑

两份代码都是在干这件事,做 100000 次的 Sqrt 和 Pow 计算。

1.2 C# **结果.** 以下是运行 4 次 C# 代码的结果

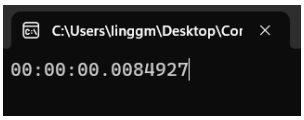

图 3. C# 结果 1

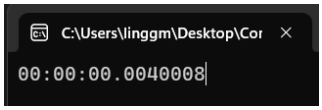

图 4. C# 结果 2

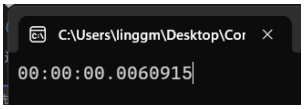

图 5. C# 结果 3

图 6. C# 结果 4

1.3 python **结果.** 以下是运行 4 次 python 代码的结果

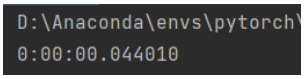

图 7. python 结果 1

图 8. python 结果 2

图 9. python 结果 3

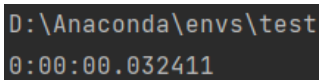

图 10. python 结果 4

1.4 **对比分析.** 同样的代码逻辑，C# 代码运行时间平均为 6.64653ms，python 代码平均运行时间为 38.5805 ms

1) 从语言的角度分析：C# 是一种静态类型、编译型语言，它在编译时进行了大量优化，生成的是机器码，这意味着它运行时的效率较高。Python 是一种动态类型、解释型语言。它在运行时进行编译，这增加了额外的开销。虽然 Python 有一些优化措施，如即时编译（JIT），但这些通常不足以匹配静态编译型语言的性能。

2) 从内存管理的角度分析：C# 使用高效的内存管理和垃圾回收机制，这有助于维持其性能。Python 也有垃圾回收，但其动态类型系统可能导致额外的内存开销，进而影响性能。
3) 从循环处理的角度分析：在 C# 中，类型是静态的，编译器可以更好地优化循环和变量的处理。在 Python 中，由于其动态类型的特性，解释器需要在每次迭代时做更多的类型检查和解析。

2. 作业 2

参考链接中的项目，对 python 代码进行优化，对各种方法进行效率分析。我参考了项目中的代码和优化方法，并自己编写了几种方法进行优化分析。

2.1 **原始方法.** 这种方法不涉及任何特殊优化或外部库，有助于展示 Python 在没有任何优化时的基本性能水平。

```
    def cal_sum_navie(n):
    beforeCal = datetime.datetime.now()
    for number in range(n):
        result = math.sqrt(number ) + math.pow(number,2)
    afterCal=datetime.datetime.now()
    calCost = afterCal - beforeCal
    return calCost
```

Listing 2. 朴素方法

2.2 numpy **方法.** Numpy 通过利用底层的数值计算优化和硬件加速来提高性能，它的向量化操作可以比纯 Python 循环快得多，因为这些操作是在底层和优化过的 C 语言代码上执行的。

```
import numpy as np
def cal_sum_numpy(n):
    beforeCal = datetime.datetime.now()
    numbers = np.arange(n)
    results = np.sqrt(numbers) + np.power(numbers, 2)
    afterCal = datetime.datetime.now()
    calCost = afterCal - beforeCal
    return calCost
```

Listing 3. numpy 方法

2.3 list **方法.** 列表推导是 Python 的一种简洁高效的构造列表的方法。list 是在底层优化过的，因此它有可能比传统的循环更快。

```
def cal_sum_list(n):
    before_cal = datetime.datetime.now()
    tmp = [math.sqrt(number) + number ** 2 for number in
        range(n)]
    after_cal = datetime.datetime.now()
    return after_cal - before_cal
```

Listing 4. list 方法

2.4 numba **方法.** 通过即时编译 Python 代码到机器码，Numba 能够显著提高 Python 代码的执行速度，特别适用于循环和数值密集型的任务。这种方法克服了 Python 动态类型和解释型语言的一些性能限制。

```
from numba import jit
import math

# 使用Numba的装饰器来编译这个函数
@jit
def calculate_sums(n):
    total_sum = 0
    for number in range(n):
        total_sum += math.sqrt(number) + math.pow(number,
            2)
    return total_sum


def cal_sum_numba(n):
    beforeCal = datetime.datetime.now()
    # 计算总和
    result = calculate_sums(n)
    afterCal = datetime.datetime.now()
    calCost = afterCal - beforeCal
    return calCost
```

Listing 5. numba 方法

2.5 multiprocess **方法.** 多进程并行计算可以有效利用多核处理器的能力，同时执行多个操作，从而大幅度提高计算速度。

```
from multiprocessing import Pool

def calculate(number):
    return math.sqrt(number) + number ** 2

def parallel_calculate(numbers):
    with Pool() as pool:
        results = pool.map(calculate, numbers)
    return results

def cal_sum_parallel(n):
    beforeCal = datetime.datetime.now()
    results = parallel_calculate(range(n))
    afterCal = datetime.datetime.now()
    calCost = afterCal - beforeCal
    return calCost
```

Listing 6. multiprocess 方法

2.6 **速度测试.** 在主程序中测试五种不同方法在不同输入规模下的性能表现，将结果存储在一个 5x5 的矩阵中。

```
def measure_execution_time(function, n):
    before_cal = datetime.datetime.now()
    _ = function(n)
    after_cal = datetime.datetime.now()
    return (after_cal - before_cal).total_seconds()

if __name__ == '__main__':
    # 定义不同的输入规模
    input_scales = [1000, 10000, 100000, 1000000, 10000000]

    # 定义不同的方法
    methods = [cal_sum_navie, cal_sum_numpy, cal_sum_numba,
        cal_sum_list, cal_sum_parallel]

    # 创建5x5矩阵来存储结果
    results = np.zeros((5, 5))

    # 测试每种方法在每种输入规模下的运行时间
    for i, n in enumerate(input_scales):
        for j, method in enumerate(methods):
            results[i, j] = measure_execution_time(method,
                n)
```

Listing 7. 速度测试

2.7 **系统信息.** 采用以下代码打印系统信息

```python
def get_system_info():
    cpu_details = cpuinfo.get_cpu_info()
    info = {
        "操作系统": platform.system(),
        "系统版本": platform.version(),
        "主机名称": platform.node(),
        "机器类型": platform.machine(),
        "CPU类型": platform.processor(),
        "CPU品牌": cpu_details['brand_raw'],
        "核心数量": cpu_details['count'],
        "核心速度": cpu_details['hz_actual_friendly']
    }
    return info

for key, value in system_info.items():
    print(f"{key}: {value}")
```

Listing 8. 系统信息

3. 结果展示与分析

```
D:\Anaconda\envs\test\python.exe D:/hw8-2/speedTest.py
操作系统: Windows
系统版本: 10.0.22621
主机名称: LAPTOP-A2R8SDHV
机器类型: AMD64
CPU类型: AMD64 Family 25 Model 80 Stepping 0, AuthenticAMD
CPU品牌: AMD Ryzen 5 5600H with Radeon Graphics
核心数量: 12
核心速度: 3.3010 GHz
```

图 11. 系统信息

```
运行时间矩阵 (行: 规模, 列: 方法):
输入规模: [1000, 10000, 100000, 1000000, 10000000, 100000000]
0.000000 0.000000 0.214319 0.000000 1.514359
0.002000 0.000000 0.000000 0.004063 1.143000
0.020089 0.000000 0.004004 0.036173 1.207634
0.213048 0.008023 0.002001 0.398022 1.537103
2.270540 0.072215 0.020005 4.003547 2.861121
23.488198 0.730643 0.213348 40.207268 15.097120
```

图 12. 速度测试 1

```
运行时间矩阵 (行: 规模, 列: 方法):
输入规模: [1000, 10000, 100000, 1000000, 10000000, 100000000]
0.000000 0.000000 0.212056 0.000000 1.461486
0.004008 0.000000 0.000000 0.004110 1.154223
0.024235 0.000000 0.000000 0.040310 1.241292
0.216849 0.008002 0.000000 0.380602 1.569227
2.184628 0.068008 0.020015 4.007053 2.705493
22.796608 0.730062 0.220966 44.590977 16.741107
```

图 13. 速度测试 2

表 1. 不同方法在不同输入规模下的运行时间（毫秒）

规模	Naive	NumPy	Numba	List	Parallel
10^3	0.0	0.0	212.1	0.0	1461.5
10^4	4.0	0.0	0.0	4.1	1154.2
10^5	24.2	0.0	0.0	40.3	1241.3
10^6	216.8	8.0	0.0	380.6	1569.2
10^7	2184.6	68.0	20.0	4007.1	2705.5
10^8	22796.6	730.1	221.0	44591.0	16741.1

朴素方法 (cal_sum_naive)： 在较小的输入规模下表现尚可，但随着输入规模的增加，计算时间急剧增长。由于使用了基本的 Python 循环，它受到 Python 解释器性能的限制，在处理大规模运算时效率低下。

Numpy 方法 (cal_sum_numpy)： 对于所有输入规模，Numpy 方法的表现都优于朴素方法。Numpy 通过优化的底层 C 语言实现进行计算，在处理大规模数据时，其向量化操作大大提高了效率。对于需要大量数值运算的场景，这是一个非常好的选择。

List 方法 (cal_sum_list)： 在各种输入规模下，都比朴素方法差。

Numba 方法 (cal_sum_numba)： 在不同输入规模下均表现出色，在大输入规模下的优势最为明显。Numba 通过即时编译 Python 代码到机器码，克服了 Python 动态类型和解释型语言的性能限制，特别适合循环和数值密集型的任务。对于追求高性能计算的应用，这是一个非常好的选择。

多进程方法 (cal_sum_parallel)： 在小输入规模下，由于进程创建和管理的开销，其性能不佳。在大输入规模下，多进程并行计算能有效利用多核处理器，显著提高了计算速度。适合于可以并行处理的计算密集型任务，但需要权衡进程管理的开销。

综合分析： 对于小输入规模，简单的方法（如朴素方法和列表推导）通常已经足够快，但随着输入规模增大，更先进的方法（如 Numpy 和 Numba）的性能优势变得明显。对于并行计算，多进程方法在大输入规模上效果显著，但在小输入规模上可能不是最佳选择。在选择最适合的方法时，应根据具体的输入规模、计算需求和可用资源（例如 CPU 核心数）来决定。