并行与分布式计算大作业

K 近邻算法的多种并行优化

中山大学计算机学院　计算机科学与技术

19335174　施天予

目录

一、概述

本次大作业我选择的自选题目是机器学习下的并行化 K 近邻算法，使用 Python 语言实现。总共完成了以下几个版本：

- 串行实现
- 基于 mpi4py（MPI）的进程级并行
- 基于 pymp（OpenMP）的线程级并行
- 基于 PyCUDA 的 GPU 并行
- sklearn 调库实现

并对所有方法进行实验结果的分析和对比总结。

二、实验原理

1. K 近邻算法

K 近邻 (k-Nearest Neighbor，KNN) 分类算法，是一个典型的机器学习算法。该方法的思路是：在特征空间中，如果一个样本附近的 k 个最近 (即特征空间中最邻近) 样本的大多数属于某一个类别，则该样本也属于这个类别。简单来说，即是给定一个训练数据集，对新的输入实例，在训练数据集中找到与该实例最邻近的 K 个实例，这 K 个实例的多数属于某个类，就把该输入实例分类到这个类中。常用的距离度量方式有曼哈顿距离、欧氏距离、余弦距离等。

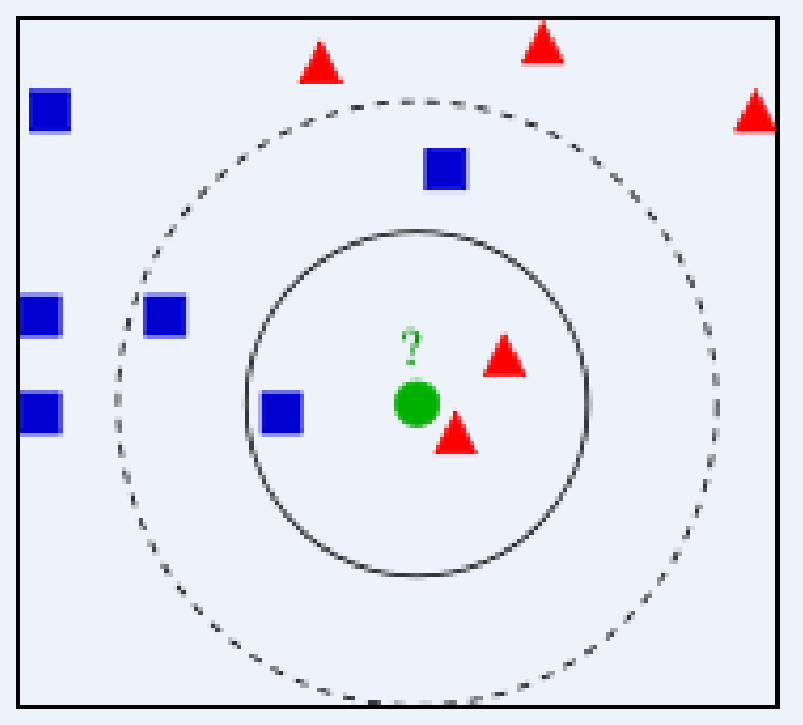

图 1: K 近邻算法

2. Python 并行优化

计算机编程语言很多，但是适合高性能数值计算的语言却并不多，在高性能计算的项目中通常会使用到的语言有 Fortran、C、C++ 等，他们是传统的高性能计算机语言，这主要得益于它们的静态编译特性，使得有它们生成的机器代码，在底层上做了很多优化，能够充分发挥硬件的性能，但是这一特性也限制了它们的灵活性和易用性。一些动态的计算机语言在灵活性和易用性方面有着明显的优势，但是由于性能等其他方面的原因却并不适合用来做大规模的数值计算。

Python 作为一种解释型的动态语言，不需要类型声明，简单易用，但是其运行性能却比 Fortran、C、C++ 等常规的高性能计算语言要差的多。然而近年来 Python 却在数值计算领域占据了越来越大的份额，甚至在高性能计算领域也看到越来越多 Python 的身影。这是因为 Python 已经不仅仅只是作为一个单独的计算机编程语言，而是变成了一个由庞大的库和工具组成的完整生态系统，包含着有 Numpy、Scipy、Pandas 等构成的科学数值计算栈。这些数值计算库和工具在底层一般封装和调用优秀的由 Fortran、C、C++ 等实现的高效算法库，因此在一定程度上弥补了 Python 本身性能上的不足。在 Python 中也有 mpi4py（MPI）、pymp（OpenMP）、PyCUDA（CUDA）等库，使我们在 Python 环境下也能完成高性能计算。

三、实验过程

1. 数据集说明

本次 K 近邻算法我使用的数据集是 MNIST 手写数字数据集，是一个十分类问题。数据集中每张图片的大小是 28*28 像素，将每张图片二值化后，我们可以根据灰度值将一个图片转为一个 784 维的向量，处理后的数据集如图2所示。训练集和验证集按 4：1 划分。

data.csv

```
ixel162,pixel163,pixel164,pixel165,pixel166,pixel167,pixel168,pixel169,pixel170,pixel171,pixel172,p
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,89,251,253,250,131,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
,0,0,0,0,0,0,0,0,0,0,0,0,60,212,254,254,254,194,48,48,34,41,48,209,254,254,254,171,0,0,0,0,0,0,0,0,
,0,0,0,0,185,254,254,184,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,63,254,254,62,0,0,0,0,0,0
,0,0,0,0,0,0,18,254,73,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5,206,106,0,0,0,0,0,0,
,0,0,33,152,253,253,253,107,1,0,0,0,0,0,0,0,206,253,253,253,253,140,0,0,0,0,0,30,139,234,253,253
,36,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,103,254,222,74,143,235,254,228,83,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,51,253,254,50,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,51,252,213,10,0,0,0,0,0,0,
,0,6,214,151,0,0,0,0,0,0,0,0,0,0,10,231,86,2,0,0,0,0,0,0,0,0,0,0,0,191,207,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,62,254,255,241,30,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,10,118,235,253,249,103
,0,0,0,26,231,253,253,191,183,223,253,253,253,253,172,216,112,0,0,0,0,0,0,0,0,0,0,0,0,36,215,25
,0,0,0,80,232,252,203,58,0,0,135,253,252,121,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,43,246,252,200,11,0,0,
,202,12,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,175,248,252,136,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,173,254,87,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,9,254,87,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,64,255,255,64,0,0,0,0,0,0,191,255,255,255,64,0,0,0,0,0,0,0,0,0,0,0,0,255,25
,215,31,0,0,0,0,0,0,0,0,0,0,101,225,175,0,0,0,0,0,10,85,147,225,231,252,252,168,33,0,0,0,0,0,0,0,0,
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,45,253,66,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
,42,42,18,0,0,0,0,0,0,0,0,0,0,0,107,253,253,253,253,253,178,131,131,131,143,253,253,253,253,182,8
,0,0,0,0,0,0,0,0,0,108,253,228,68,41,132,174,178,253,255,253,253,176,17,0,0,0,0,0,0,0,0,0,0,0,0,0,
,0,0,0,0,0,0,0,0,0,63,254,151,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,191,226,27,0,0,0,0
```

图 2: 处理后的数据集

2. 串行实现

串行的 K 近邻算法比较简单，将验证集中的单个样本与训练集中的所有样本计算距离（这里我使用了欧氏距离），选出与验证集中该样本最近的 K 个样本，最后进行多数投票决定该样本的 label。算法的主要的函数如下：

```
# 多数投票
def most_frequent(List):
    return max(set(List), key=List.count)
# 对验证集单个样本选出label的KNN算法
def KNN(valid_set, train_set, K):
    dist_arr = np.array([np.linalg.norm(valid_set[1:] - other[1:]) for other in
        ↪ train_set])
    index_arr = np.argpartition(dist_arr, K)[:K]
    return most_frequent([item[0] for item in [train_set[p] for p in index_arr]])
```

主函数先对数据集进行 shuffle 操作，再划分训练集和验证集，最后通过 KNN 算法预测 label，计算运行时间与准确率。

```
if __name__ == "__main__":
    data = np.genfromtxt("data.csv", delimiter=',', dtype=np.uint8)
    data = data[1:]
    total = int(len(data)*sample_ratio)
    data = data[:total]
    valid_num = int(split_ratio * total)
    train_num = total - valid_num

    indexes = np.array(range(total))
    random.Random(0).shuffle(indexes)
    train_set = [data[train] for train in indexes[:train_num]]
    valid_set = [data[valid] for valid in indexes[train_num:]]

    prediction = np.empty((len(valid_set)), dtype='uint8')
    start = time.time()
    for i, valid in enumerate(valid_set):
        prediction[i] = KNN(valid, train_set, K)
    end = time.time()

    print(end-start)
    print(accuracy_score([valid[0] for valid in valid_set], prediction))
```

3. mpi4py 进程级并行

MPI（Message-Passing Interface）消息传递接口是一个基于分布式内存系统使用消息传递的并行编程模型。不同的进程之间使用集合通信，以实现进程级的并行。MPI 的具体实现并没

有提供 Python 的编程接口，这就使得我们没法直接地使用 Python 调用 MPI 实现高性能的计算，不过幸运的是，我们有 mpi4py。mpi4py 是一个构建在 MPI 之上的 Python 库，主要使用 Cython 编写，它以一种面向对象的方式提供了在 Python 环境下调用 MPI 标准的编程接口，和 C++ 的 MPI 编程接口非常类似，因此我们可以轻松使用 mpi4py 编写基于 MPI 的高性能并行计算程序。

与 MPI 一样，mpi4py 首先需要声明通信子，并且获取进程数和进程 id，数据的读取只需由主进程完成。

```
if __name__ == "__main__":
    comm = MPI.COMM_WORLD   # 通信子
    size = comm.Get_size()  # 进程数
    rank = comm.Get_rank()  # 进程id

    # 主进程读取数据
    if rank == 0:
        data = np.genfromtxt("data.csv", delimiter=',', dtype=np.uint8)
        data = data[1:]
        total = int(len(data)*sample_ratio)
        data = data[:total]
        valid_num = int(split_ratio * total)
        train_num = total - valid_num

        indexes = np.array(range(total))
        random.Random(0).shuffle(indexes)
        train_set = [data[train] for train in indexes[:train_num]]
        valid_set = [data[valid] for valid in indexes[train_num:]]
        start_time = time.time()
    else:
        train_set = None
        valid_set = None
```

主进程将 train_set 和 valid_set 广播到各个进程后，每个进程分别计算 valid_set 自己部分的预测值。这里我的划分方法是块划分，即将 valid_set 连续的一部分划分给单个进程。每个进程都能得到大致相同的数据量，只有最后一个进程可能会稍少一些。在所有进程计算完毕后，将结果发送给主进程。

```
    # 所有进程分别计算valid_set自己部分的prediction
    train_set = comm.bcast(train_set, root=0)
    valid_set = comm.bcast(valid_set, root=0)
    local_start = rank * (len(valid_set) // size)
    local_end = len(valid_set)-1 if rank==size-1 else local_start+(len(valid_set)
        ↪ //size)-1
    local_n = local_end - local_start + 1
```

```
    local_pred = np.empty((local_n), dtype='uint8')
    for i in range(local_start, local_end+1):
        local_pred[i-local_start] = KNN(valid_set[i], train_set, K)
    # 结果发送给主进程
    if rank != 0:
        comm.send(local_pred, dest=0)
```

最后主进程接收来自所有进程的预测结果，计算运行时间和准确率。

```
    if rank == 0:
        prediction = np.empty((len(valid_set)), dtype='uint8')
        prediction[:local_n] = local_pred
        for i in range(1, size):
            data = comm.recv(source=i)
            start = i*(len(valid_set)//size)
            end = len(valid_set)-1 if i==size-1 else start+(len(valid_set)//size)-1
            prediction[start:end+1] = data
        end_time = time.time()
        print(end_time-start_time)
        print(accuracy_score([valid[0] for valid in valid_set], prediction))
```

4. pymp 线程级并行

与基于分布式内存系统的进程级并行 MPI 不同的是，OpenMP 是一种基于共享内存系统的线程级并行编程模型。OpenMP 提供了对并行算法的高层的抽象描述，编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信，具有强大的灵活性。

OpenMP 不支持 Python，但有一个开源的 pymp 库（https://github.com/classner/pymp）带来了类似 OpenMP 的功能。它拥有 OpenMP 的优点，例如最小的代码更改量和极高的效率，并将它们与代码清晰度和易用性的 Python Zen（一种 python 编程原理指导，使用 import this 可获得打印结果）相结合。

pymp 与 OpemMP 一样简单，我们只需将串行程序代码的一段修改一下即可。将原来的 numpy 数组 prediction 改为 pymp 的共享内存数组，再在 for 循环前设置线程数，即可完成 pymp 的线程级并行。

```
    prediction = pymp.shared.array((len(valid_set)), dtype='uint8')
    with pymp.Parallel(12) as p:
        for i in p.range(len(valid_set)):
            prediction[i] = KNN(valid_set[i], train_set, K)
```

5. PyCUDA 并行

然而 CPU 受限于线程数的影响，使计算不能得到充分的并行，因此我也使用了 PyCUDA 将 K 近邻算法在 GPU 上实现并行。CUDA（Compute Unified Device Architecture），是显卡

厂商 NVIDIA 推出的运算平台，是一种由 NVIDIA 推出的通用并行计算架构，该架构使 GPU 能够解决复杂的计算问题。通过使用 GPU 高性能并行多线程能力，解决大规模计算的模型已被广泛应用在图像处理、深度学习等各项领域。CUDA 线程数目通常能达到数千个甚至上万个，通过 CUDA 编程实现 GPU 上的并行计算，性能可以得到大幅度的提升。

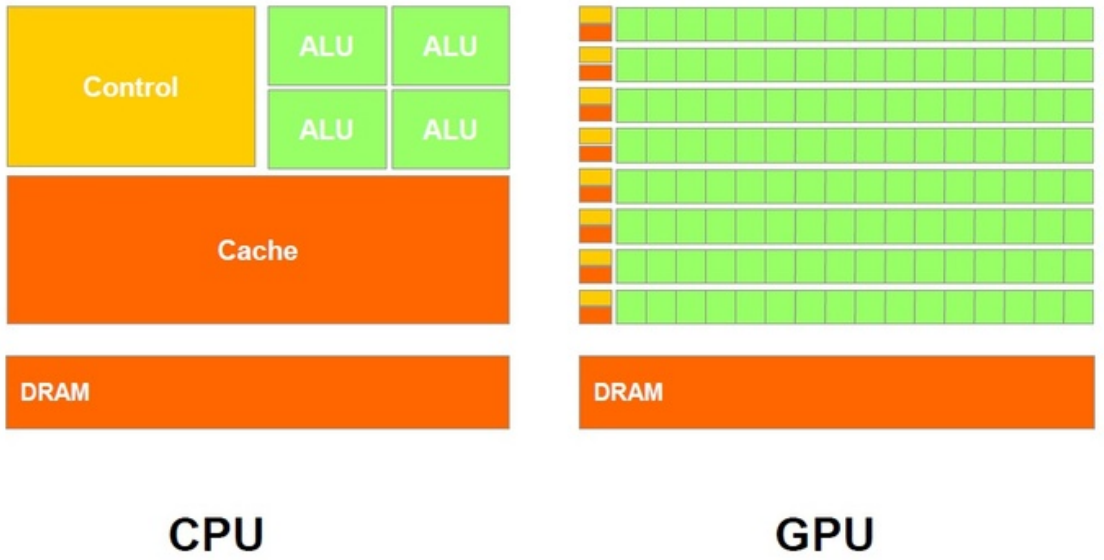

图 3: CPU 与 GPU 的结构对比

使用 PyCUDA 之前，必须先导入这三个库进行初始化

```
import pycuda.driver as drv
import pycuda.autoinit
from pycuda.compiler import SourceModule
```

CUDA 的并行计算需要依赖 CPU-GPU 的异构计算，即在 CPU 的 host 端进行串行处理，在 GPU 的 device 端进行并行计算。我们首先需要在 GPU 上为 device 端分配相应空间，将 train_set 和 valid_set 从 host 端拷贝到 device 端。另外还需要一个 dest_gpu 用于存储计算返回的结果。dest_gpu 是一个二维矩阵，行数是 valid_set 的样本数，列数是 train_set 的样本数，每一行存储的是 valid_set 中单个样本到 train_set 所有样本的距离。

```
def GPU_init_memory(train_set, valid_data, valid_num, train_num):
    train_gpu = drv.mem_alloc(train_set.nbytes)
    drv.memcpy_htod(train_gpu, train_set)
    valid_gpu = drv.mem_alloc(valid_data.nbytes)
    drv.memcpy_htod(valid_gpu, valid_data)
    dest_gpu = drv.mem_alloc(valid_num * train_num * 4) # uint32
    return train_gpu, valid_gpu, dest_gpu
```

使用 GPU 并行的 K 近邻算法代码如下。我们在 GPU 的 device 端分配空间后，需要指定 blocksize 和 gridsize 启动核函数 get_neighbor_dist，一个 block 用于计算 train_set 中单个样本到所有 valid_set 的欧氏距离。在 GPU 计算结束后，我们将结果从 device 端拷贝回 host 端，得到最后的预测结果。

```
def parallel_KNN(train_set, valid_data, valid_num, train_num, K):
    train_labels = np.array([item[0] for item in train_set])
    train_data = np.array([item[1:] for item in train_set])
    train_gpu, valid_gpu, dest_gpu = GPU_init_memory(train_data, valid_data,
        ↪ valid_num, train_num)
    func = mod.get_function("get_neighbor_dist")
    for i in range(valid_num):
        func(train_gpu, valid_gpu, dest_gpu, np.int32(i), np.int32(train_num),
            ↪ block=(416, 1, 1), grid=(train_num, 1, 1))
    results = np.empty([valid_num, train_num], dtype=np.uint32)
    drv.memcpy_dtoh(results, dest_gpu)
    prediction = np.empty((len(valid_set)), dtype='uint8')
    for i, item in enumerate(results):
        index_arr = np.argpartition(item, K)[:K]
        prediction[i] = most_frequent([train_labels[index] for index in index_arr])
    return prediction
```

CUDA 编程中最重要的就是核函数了，这个核函数 get_neighbor_dist 需要由 C/C++ 进行编写。刚开始我的想法是一个 thread 计算一个维度的距离，再将所有维度的距离加起来计算欧式距离。代码如下：

```
mod = SourceModule ("""
#include <stdint.h>
#include <math.h>
__global__ void get_neighbor_dist(uint8_t* train, uint8_t* valid, uint32_t* dest,
    ↪ int offset, int train_num)
{
    int bid = blockIdx.x;
    int x = threadIdx.x;
    uint8_t *cur_item = valid + offset * 784;
    __shared__ float dist_arr [784];
    int difference = *(cur_item + x) - *(train + bid * 784 + x);
    dist_arr[x] = difference * difference;
    for (int stride = 392; stride > 0; stride /= 2) {
        __syncthreads();
        if (x < stride)
            dist_arr[x] += dist_arr[x + stride];
    }
    if (x == 0)
        dest[offset * train_num + bid] = (int)sqrt(dist_arr[0]);
```

```
    return;
}
""")
```

然而后来我发现一个 thread 计算一个维度距离的效果不是最优的，为每个 block 分配过多的线程会使 GPU 显存的创建开销以及数据拷贝开销更大。我发现使用更少的线程数，为部分 thread 分配计算两个维度距离的任务时，每个线程计算粒度更大，运行时间还能减小。通过不断调参，最后我设置单个 block 线程数即 blocksize=416 时效果达到最佳。

```
mod = SourceModule ("""
#include <stdint.h>
#include <math.h>
__global__ void get_neighbor_dist(uint8_t* train, uint8_t* valid, uint32_t* dest,
    ↪ int offset, int train_num)
{
    int bid = blockIdx.x;
    int x = threadIdx.x;
    uint8_t *cur_item = valid + offset * 784;
    __shared__ float dist_arr [832];
    if (x < 392) {
        int difference = *(cur_item + x) - *(train + bid * 784 + x);
        dist_arr[x] = difference * difference;
        difference = *(cur_item + 392 + x) - *(train + bid * 784 + 392 + x);
        dist_arr[x + 416] = difference * difference;
    }
    else {
        dist_arr[x] = 0;
        dist_arr[x + 416] = 0;
    }
    for (int stride = 416; stride > 0; stride /= 2) {
        __syncthreads();
        if (x < stride)
            dist_arr[x] += dist_arr[x + stride];
    }
    if (x == 0)
        dest[offset * train_num + bid] = (int)sqrt(dist_arr[0]);
    return;
}
""")
```

这里还有一个值得关注的点，那就是在计算所有维度距离求和时，多核计算全局总和与普通单核求和的不同。当核的数目比较多时，不再由单个核计算所有的累加结果，而是将各个核两两结对，用一半的核将每两个核的结果加起来保存。以此类推，示意图如图4 所示。圆圈表示当前核所得到的和，箭头表示一个核将自己的部分和发送给另一个核，加号表示一个核在收

到另一个核发送来的部分和后与自己本身的部分和相加。这样相比于用单核去一次次累加计算，效率可以得到极大的提升。这个方法在期末考试中也有考到，可惜我大作业的代码是在去年写的，有点生疏，而考试又时间紧张导致我并没有完全写对。

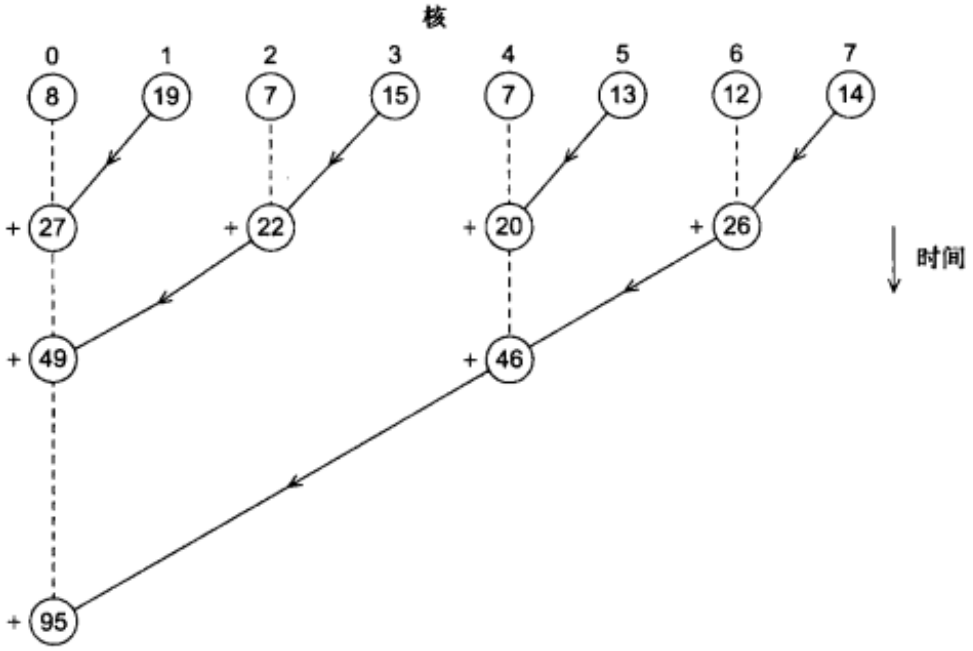

图 4: 多核共同计算形成全局总和

6. sklearn 实现

为了比较我的 K 近邻并行算法究竟性能如何，我也用 sklearn 实现了一遍。sklearn 是一个 Python 第三方提供的非常强力的机器学习库，它包含了从数据预处理到训练模型的各个方面，其代码编写十分简单，且计算效率也是极高的。

```
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=10)
neigh.fit(train_data, train_labels)
prediction = neigh.predict(valid_data)
```

四、实验结果与分析

1. 实验环境

软件环境

- Ubuntu Linux 18.04
- Python 3.8.8
- Cuda compilation tools, release 9.1, V9.1.85

硬件环境

- Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz
- GeForce GTX TITAN X/PCIe/SSE2

2. 正确性检验

首先我们检验所有版本的 K 近邻算法是否是正确的。设置最近邻个数 K=10，选取相同的数据集大小，各版本的运行结果如下：

```
(base) me@home:~/Desktop/STY/parallel/KNN$ python knn_serial.py
17.960474967956543
0.7273809523809524
(base) me@home:~/Desktop/STY/parallel/KNN$ mpiexec -n 4 python knn_mpi.py
4.9727795124053955
0.7273809523809524
(base) me@home:~/Desktop/STY/parallel/KNN$ python knn_pymp.py
3.9608113765716553
0.7273809523809524
(base) me@home:~/Desktop/STY/parallel/KNN$ python knn_pycuda.py
3360
0.1528182029724121
0.8988095238095238
(base) me@home:~/Desktop/STY/parallel/KNN$ python knn_sklearn.py
0.1470625400543213
0.9
```

图 5: 正确性检验

每次运行最后输出的一行是准确率。可以发现不同版本的 K 近邻算法准确率都是比较高的，说明实现是正确的。但是可能由于算法的一些实现细节不同和随机性的影响，不同方法在准确率上也有所不同。

3. 实验结果

(i) 串行实现

在老师的作业要求中提到可以比较不同操作系统运行程序的性能，但是由于并行的环境很难相同，所以我尝试比较了一下串行程序在 Ubuntu 和 Windows10 的性能。图6是两者在同一台电脑不同大小数据集时的性能对比。

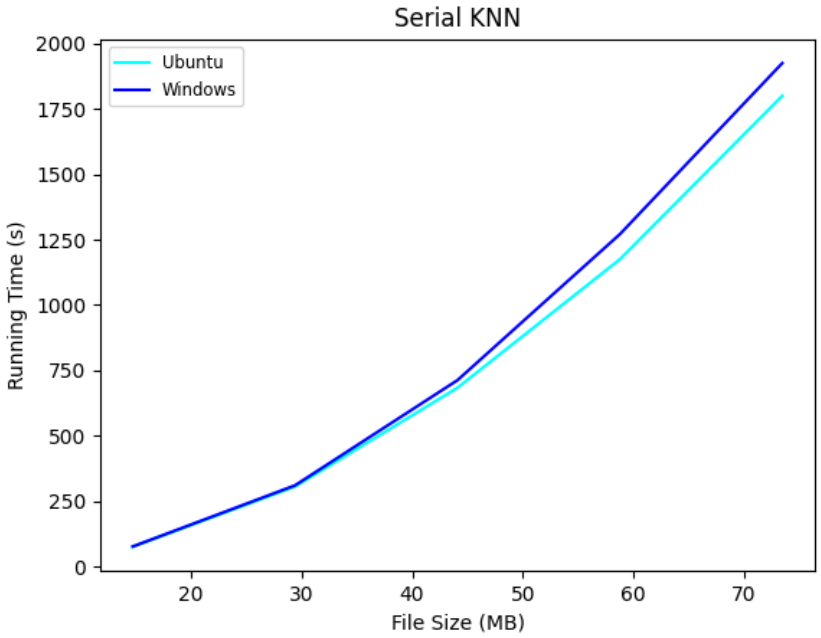

图 6: 串行算法在不同操作系统的运行时间

可以发现两者的运行速度相差不大，但在任何大小的数据集上 Ubuntu 都要稍快一些。于是我查阅相关资料，发现了以下几点原因：

1. **内存机制：** Ubuntu 完全发挥内存性能，且在程序关闭后不会立即释放内存，便于下次程序加速打开；而 Windows 无论内存是否足够都要使用虚拟内存，且在程序关闭后立即释放内存。
2. **软件差异：** 同种软件 Ubuntu 一般更快，因为 Ubuntu 下的软件比较轻量。
3. **文件系统：** Ubuntu 的文件系统是 ext4，相比使用 ntfs 的 Windows 更不易产生文件碎片。
4. **资源使用：** Windows 在使用时比 Ubuntu 会启动更多的进程，也就是会占用更多的资源。
5. **防病毒：** Windows 上的杀毒软件，会使程序的运行速度减慢。

(ii) mpi4py

接下来比较使用 mpi4py 并行化 K 近邻算法的性能。设置进程数分别为 2、4、6，比较程序在不同大小的数据集的运行速度。由于测试的次数不是特别多，受随机性的影响部分结果会有一定差异，但是仍然可以看出使用 mpi4py 并行化后程序运行时间大大减小，性能有显著提升，并且随着数据集规模的增大提升越来越明显。因为我的 CPU 共有 6 个核，所以当进程数为 6 时，程序性能达到最佳。具体结果如图7所示。

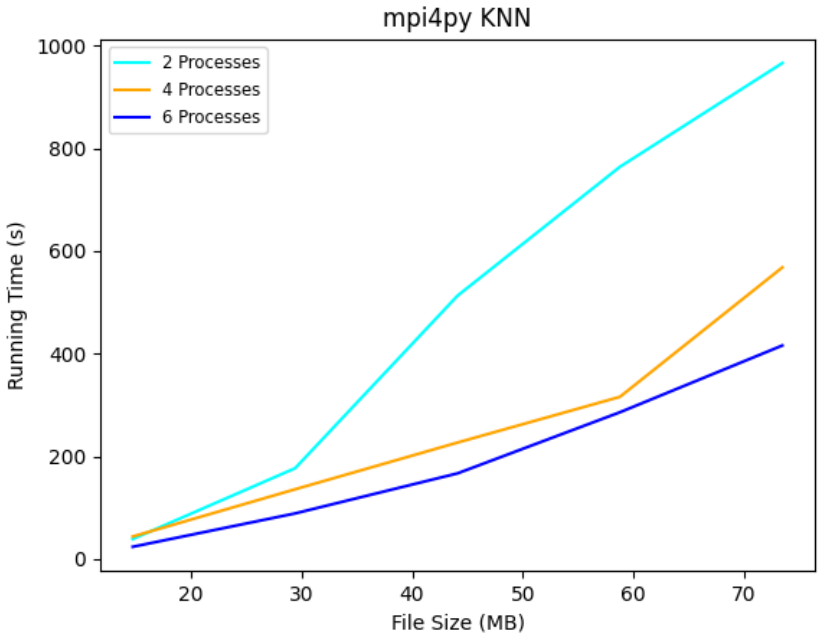

图 7: mpi4py 不同进程数的运行时间

(iii) pymp

设置 pymp 的线程数分别为 2、4、6、8、12，可以发现 pymp 在 12 线程时加速效果达到最佳，这也是由于我的 CPU 是 6 核 12 线程的原因。具体结果如图8所示。

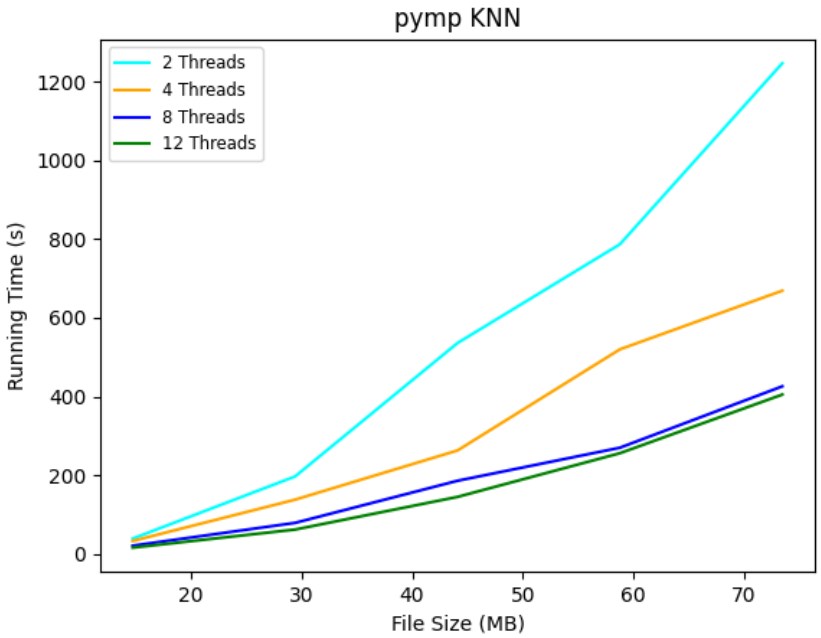

图 8: pymp 不同线程数的运行时间

mpi4py 和 pymp 都是 CPU 的并行，前者是进程级的并行，后者是线程级的并行，那么它们的并行效果哪个更好呢？我将它们的运行结果合并到图9，方便我们进行对比分析。

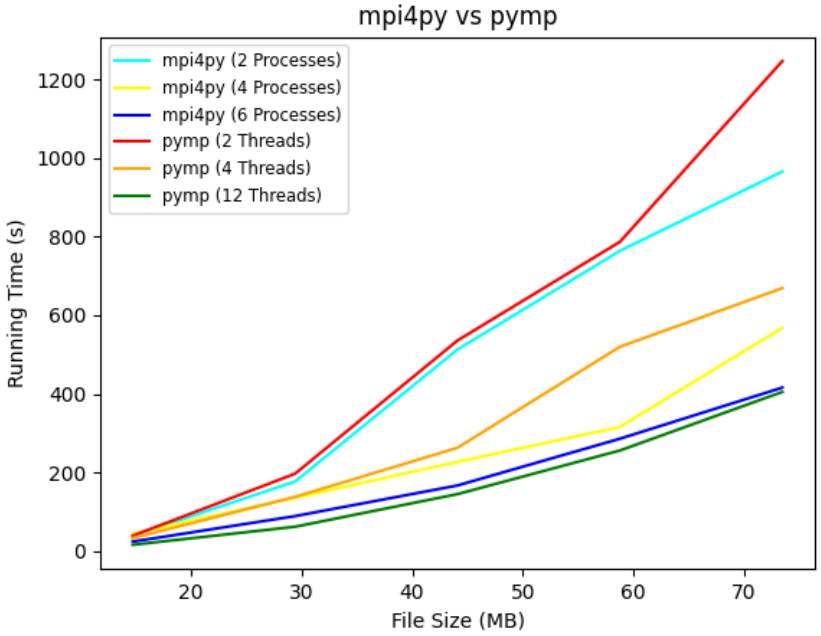

图 9: mpi4py 与 pymp 的运行时间

可以发现，当我们用 mpi4py 的进程数等于 pymp 的线程数进行比较时，mpi4py 的并行效果是优于 pymp 的。这也不难理解，mpi4py 的并行计算由每一个核完成，性能相比使用每个线程完成并行化的 pymp 肯定是更好的，并行粒度也更大。

但是 mpi4py 的最佳并行效果却不如 pymp，这是因为 pymp 的线程数可以设置得比 mpi4py 的进程数更多，在分配给每个进程/线程的计算任务不那么大的情况下，使用更多数目的线程效果也会更优。另外，基于共享内存系统的 pymp 使用线程间共享内存的方式协调并行计算，在单台主机的多核结构上效率更高、内存开销更小；而基于分布式内存系统的 mpi4py 需要进程间的通信，开销也会更大一些，不过如果在多台主机上 mpi4py 的优势就体现出来了，而 pymp 却无法完成多台主机间的并行计算。

(iv) PyCUDA

使用 PyCUDA 将 K 近邻算法用 GPU 并行化后，性能可以得到更加显著的提升。之前的实验过程已经提到，在设置 416 线程时我的 PyCUDA 并行效果达到最佳。因此这里我也仅比较了 784 线程和 416 线程的性能提升效果。具体结果如图10所示。

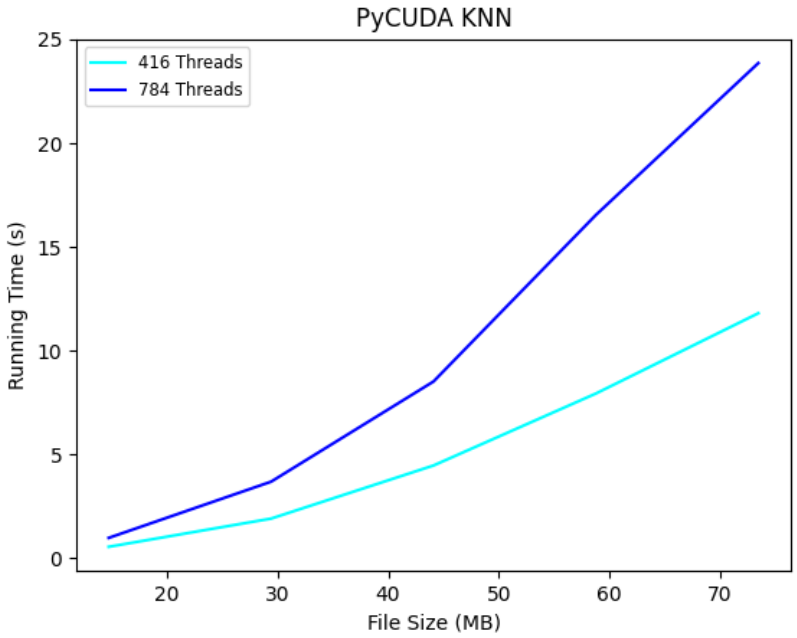

图 10: Pycuda 不同线程数的运行时间

可以发现使用 GPU 并行计算后，运行时间大大减小，性能可以得到几十倍甚至上百倍的提升。PyCUDA 在 784 线程和 416 线程的主要不同之处在于，784 线程用一个 thread 计算一个维度的距离，而 416 线程的部分 thread 计算两个维度的距离，每个线程计算粒度更大。而且 784 线程在 GPU 显存的创建开销以及数据拷贝开销会更大，当使用过多线程时，这一部分的拷贝调度的延迟会增大，而并行化的可加速部分占比较小，从而导致 784 线程比 416 线程慢了一倍。

因此我们也不能盲目地为每个 block 分配过多的线程，在每个线程计算任务不大时，如果使用过多线程会产生过多不必要的开销。合理分析不同的程序任务，根据计算任务的不同为 block 分配合理的线程数，才能得到最佳的性能。

4. 对比分析

(i) 并行实现对比

我们将所有并行 K 近邻算法的最优性能结果与串行算法的性能进行比较，不同并行算法的运行时间和加速比如图11和图12所示。

由实验数据可以看出，当数据集规模不断增大时，串行算法的运行速度几乎不变，故处理时间也随着问题规模同比例增大；使用 mpi4py 和 pymp 的 CPU 并行实现随问题规模增大加速比不断增加，运行时间也大大减小；而 PyCUDA 的 GPU 并行实现的运行速度更随着问题规模增加更是有着显著的提升，并在当前实验规模的情况下未发生加速饱和现象，相对于串行算

法的加速比也在不断增加，加速效果比 CPU 并行也要出色许多。

当数据集为 74MB 时，mpi4py 加速比达到 4.32，pymp 加速比达到 5.25，PyCUDA 加速比更是达到了 183！虽然 mpi4py、pymp 和 PyCUDA 的并行实现对 K 近邻算法的串行程序都有加速效果，但显然使用 GPU 的并行计算加速效果更加显著。随着问题规模的扩大，GPU 并行计算可以大幅度减少程序的运行时间，从而极大程度上地提升了 K 近邻算法的应用性能。

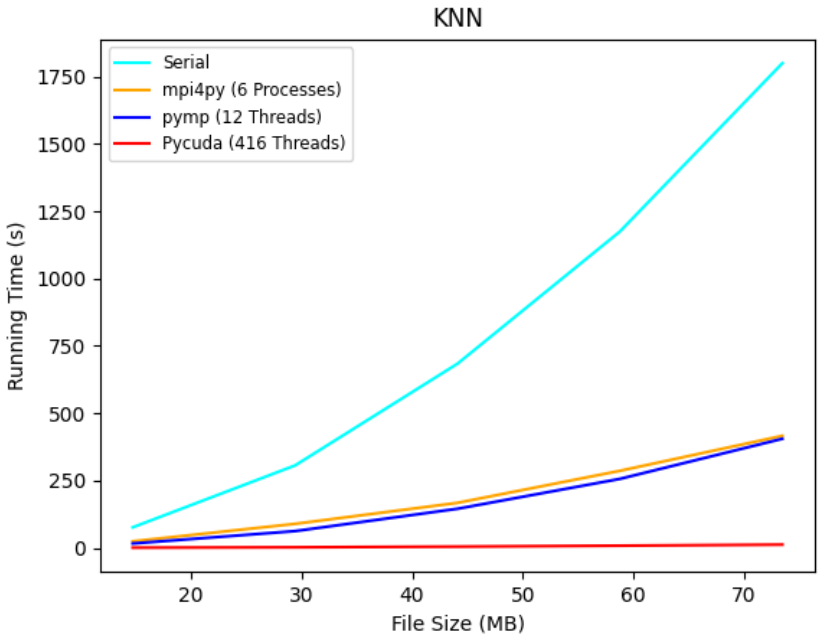

图 11: K 近邻算法的运行时间对比

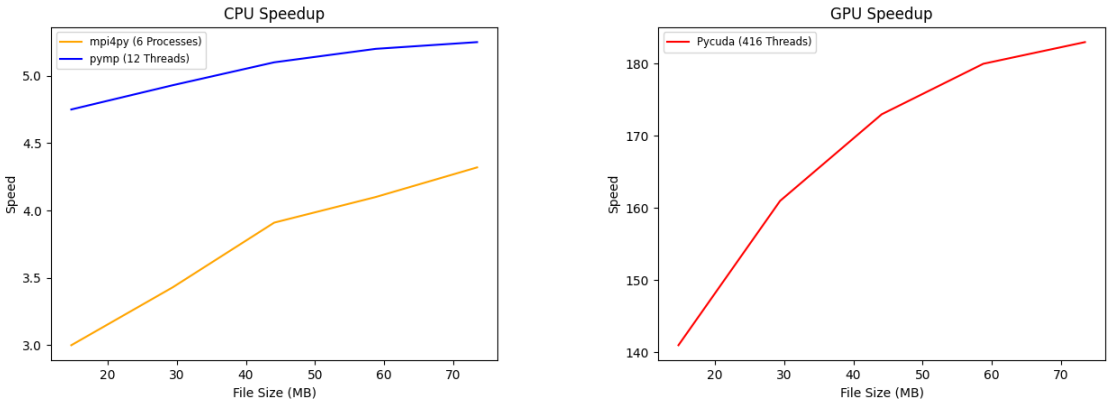

图 12: mpipy4、pymp 和 PyCUDA 的加速比

(ii) 与 sklearn 的对比

最后我又将 K 近邻算法的 PyCUDA 实现与使用机器学习库 sklearn 做了对比，为探究并行化后的 K 近邻算法究竟性能如何。然而结果出乎我的所料，使用 GPU 并行计算的运行时间比使用 CPU 计算的 sklearn 还慢了一倍左右。

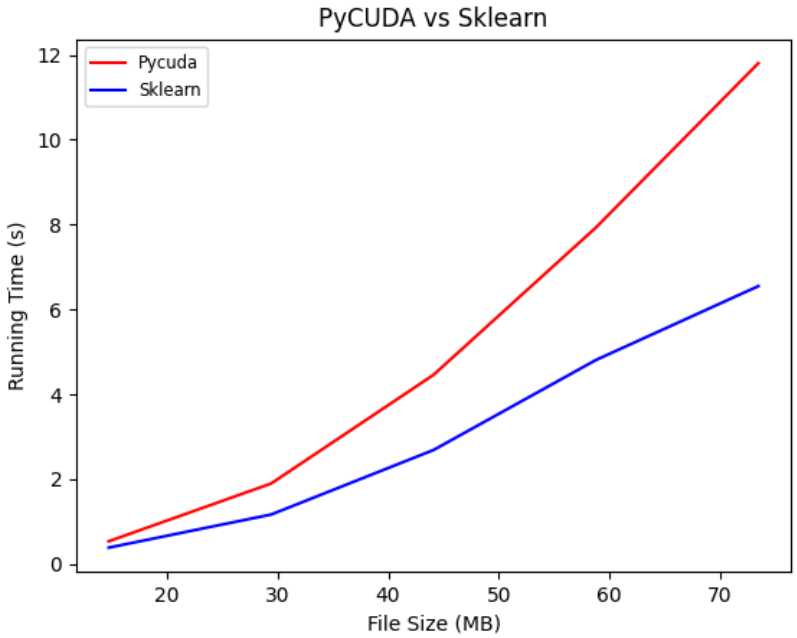

图 13: Pycuda 与 Sklearn 的运行时间

这让我十分困惑，通过查阅相关文档发现，因为常规暴力搜索是十分耗时的，sklearn 使用了一些不同的算法来绕开所有距离的计算，比如 20 维以下使用 KD Tree 算法，超过 20 维使用 Ball Tree 算法。这些算法在空间二叉树或是超球体中进行搜索，相比于暴力的距离计算性能可以得到极大的提升。

五、总结

本次实验我首先实现了 K 近邻算法的串行版本，然后分别实现了 mpi4py 和 pymp 的 CPU 并行版本，以及 PyCUDA 的 GPU 并行版本。通过实验发现，不论是 mpi4py、pymp 还是 PyCUDA，其对 K 近邻算法的实现都有不错的加速效果，但 PyCUDA 的 GPU 并行效果更好。随着问题规模的增大，其加速效果可以得到显著的提升。

其实在期末考完后我也想尝试用 Ray 并行化 K 近邻算法，但是我简单尝试后发现速度比串行还慢，又由于有其它大作业就没有深入探究了，有点可惜。《并行与分布式计算》的课程还是十分有趣的，我也常常在上课时被老师的博学所震撼，受益匪浅。希望在未来的日子我能有机会继续探索关于并行与分布式的的知识宝库。