

# Parallel Computing - PB20000178李笑

---

电脑配置

在实验 lab1 中我们已经将 MPI,OpenMP,CUDA 配置完成, 这里仅简略列出电脑的主要参数。

TechPowerUp GPU-Z 2.52.0


Graphics Card | Sensors | Advanced | Validation

Name: Intel(R) UHD Graphics [Lookup](#)

GPU: Comet Lake GT2 Revision: V0

Technology: 14 nm Die Size: Unknown

Release Date: Aug 21, 2019 Transistors: Unknown

BIOS Version: Unknown  ☒ UEFI

Subvendor: Huaqin Device ID: 8086 9B41 - 1E83 3E1B

ROPs/TMUs: 8 / 16 Bus Interface: N/A ?

Shaders: 24 Unified DirectX Support: 12 (12\_1)

Pixel Fillrate: 9.2 GPixel/s Texture Fillrate: 18.4 GTexel/s

Memory Type: LPDDR3 Bus Width: 128 bit

Memory Size: N/A Bandwidth: 34.0 GB/s

Driver Version: 27.20.100.8984 DCH / Win10 64

Driver Date: Nov 19, 2020 Digital Signature: WHQL


GPU Clock: 299 MHz Memory: 1064 MHz Boost: 1147 MHz

Default Clock: 300 MHz Memory: 1067 MHz Boost: 1150 MHz

Multi-GPU: Disabled Resizable BAR: Disabled

Computing ☒ OpenCL ☐ CUDA ☒ DirectCompute ☒ DirectML

Technologies ☒ Vulkan ☐ Ray Tracing ☒ PhysX ☒ OpenGL 4.6

Intel(R) UHD Graphics  [Close](#)

TechPowerUp GPU-Z 2.52.0


Graphics Card | Sensors | Advanced | Validation

Name: NVIDIA GeForce MX350 [Lookup](#)

GPU: GP107 Revision: A1

Technology: 14 nm Die Size: 132 mm<sup>2</sup>

Release Date: Feb 10, 2020 Transistors: 3300M

BIOS Version: 86.07.92.00.7A  ☐ UEFI

Subvendor: Huaqin Device ID: 10DE 1C94 - 1E83 3E1B

ROPs/TMUs: 16 / 40 Bus Interface: PCIe x16 3.0 @ x4 1.1 ?

Shaders: 640 Unified DirectX Support: 12 (12\_1)

Pixel Fillrate: 23.5 GPixel/s Texture Fillrate: 58.7 GTexel/s

Memory Type: GDDR5 (Hynix) Bus Width: 64 bit

Memory Size: 2048 MB Bandwidth: 56.1 GB/s

Driver Version: 26.21.14.4250 (NVIDIA 442.50) DCH / Win10 64

Driver Date: Feb 24, 2020 Digital Signature: WHQL


GPU Clock: 1354 MHz Memory: 1752 MHz Boost: 1468 MHz

Default Clock: 1354 MHz Memory: 1752 MHz Boost: 1468 MHz

NVIDIA SLI: Disabled Resizable BAR: Disabled

Computing: ☒ OpenCL ☒ CUDA ☒ DirectCompute ☒ DirectML

Technologies: ☒ Vulkan ☐ Ray Tracing ☒ PhysX ☒ OpenGL 4.6

NVIDIA GeForce MX350  [Close](#)

```

+ xiaoli@xiaoli-KLVC-WXX9: ~
xiaoli@xiaoli-KLVC-WXX9:~$ cat /proc/cpuinfo | grep "physical id" | sort | uniq | wc -l
1
xiaoli@xiaoli-KLVC-WXX9:~$ cat /proc/cpuinfo | grep "cpu cores" | uniq
cpu cores          : 4
xiaoli@xiaoli-KLVC-WXX9:~$ cat /proc/cpuinfo | grep "processor" | wc -l
8
xiaoli@xiaoli-KLVC-WXX9:~$ cat /proc/cpuinfo | grep name | cut -f2 -d: | uniq -c
8 Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz

```

# 基于 KNN 的购物反馈预测的OpenMP优化

## Introduction:

选择这个题目是受到上学期《机器学习》课程大作业的启发，课程主页[https://miralab.ai/course/ml\\_2022fall/](https://miralab.ai/course/ml_2022fall/)中有项目要求和数据集描述，由于提供的原始数据集数据量巨大（87766\*15，数据条数为87766条，特征空间为15维），而knn算法必须计算每个点与其他点之间的距离以求得最近k个点的label来预测自己的label，这就导致运行完整个程序需要花费很多时间。因为距离计算之间相对独立，所以在这里我可以利用OpenMP并行很多循环处理和矩阵乘法操作，有效的降低了程序运行所需时间。

原始数据集train\_data\_all.json是从购物网站直接抓来，并没有进行数据清洗工作，所以有很多空白项和杂乱的数据，因此我们需要先对数据进行处理。我将数据处理部分放在data\_proc.py文件中，knn预测实现放在kNN.py文件中，整个处理和预测过程可以通过运行pb20000178.py文件来进行，并输出预测准确率。

为了测试的快捷性，我们只选取了预测的中间数据支持向量集（Store\_data.npy大小为28904\*10，Store\_label.npy大小为28904\*1，通过transfer.py将其拼接为一个大小为28904\*11的support\_data.txt，最后一维是数据的label）来进行我们的实验，但是需要知道这个方法是可泛化的，而且随着数据量或近邻点数量k的增加，程序运行时间是多项式增加的。

## Code Description:

lab6\_shopping.cpp中部分注释代码没有删去，其作用是输出中间信息，确保算法在并行前后都具有正确性。

```
#define NUM 28904 //总数据的数量
#define NUM1 5780 //测试数据的数量 28904*0.2 5780
#define NUM2 23124 //训练数据的数量 28904*0.8 23124
#define N 10 //特征数据的数量（维数）
#define KN 15//K的最大取值

typedef struct {
    double data;//距离
    char trainlabel;//用于链接训练标签
}Distance;

typedef struct {
    int data[N];
    int label;
}TestAndTrain; // 数据存储结构

TestAndTrain test[NUM1]; //测试数据结构体数组
TestAndTrain train[NUM2]; //训练数据结构体数组
TestAndTrain temp[NUM]; //临时存放数据结构体数组
Distance distance[NUM2]; //存放距离结构体数组

void makerand(TestAndTrain a[],int n){ //函数功能：打乱存放标签后的结构体数组
    TestAndTrain t;
    int i=0,n1,n2;
```

```

    srand((unsigned int)time(NULL));
    for(i=0;i<n;i++){
        n1 = (rand() % n); //产生n以内的随机数  n是数组元素个数
        n2 = (rand() % n);
        if(n1 != n2){ //若两随机数不相等 则下标为这两随机数的数组进行交换
            t = a[n1];
            a[n1] = a[n2];
            a[n2] = t;
        }
    }
}

void tempdata(char filename[]){ //临时存放所有数据然后打乱
    FILE* fp = NULL;
    fp = fopen(filename, "r");
    int i=0,j=0;
    for(i=0;i<NUM;i++){
        for(j=0;j<N;j++){
            fscanf(fp, "%d ", &temp[i].data[j]);
            fgetc(fp);
        }
        fscanf(fp, "%d", &temp[i].label);
    }

    makerand(temp, NUM); //打乱所有数据

    fclose(fp);
    fp = NULL;
}

void loaddata() { //加载数据          分割：测试NUM1组    训练NUM2组
    int i, j, n = 0, m = 0;
    for (i = 0; i < NUM; i++) {
        if (i < NUM1) { //存入测试集
            for (j = 0; j < N; j++) {
                // printf("i=%d  j=%d\n", i, j);
                test[n].data[j] = temp[i].data[j]; //存入花的四个特征数据
            }
            test[n].label = temp[i].label; //存入花的标签
            n++;
        }
        else { //剩下的行数存入训练集
            for (j = 0; j < N; j++) {
                train[m].data[j] = temp[i].data[j]; //存入花的四个特征数据
            }
            train[m].label = temp[i].label; //存入花的标签
            m++;
        }
    }
}

double computedistance(int n1, int n2) { //计算距离
    double sum = 0.0;

```

```

    int i; int tid;
    int temp[10];
    for (i = 0; i < N; i++) {
        sum += pow(test[n1].data[i] - train[n2].data[i], 2.0);
    }
    return sqrt(sum); //返回距离
}

int max(int a, int b, int c) { //找出频数最高的 测试数据就属于出现次数最高的
    if(a>b && a>c) return 1;
    if(b>a && b>c) return 2;
    if(c>a && c>b) return 3;
    return 0;
}

void countlabel(int* sum ,int k, int n) { //统计距离最邻近的k个标签出现的频数
    int i;
    int sum1 = 0, sum2 = 0, sum3 = 0;
    for (i = 0; i < k; i++) {
        switch (distance[i].trainlabel) { //用Distance结构体指针p来取K个距
        离最近的标签来进行判断
            case 1:sum1++; break;
            case 2:sum2++; break;
            case 3:sum3++; break;
        }
    }
    if (max(sum1, sum2, sum3) == test[n].label) { //检测距离最近的k个标签与
    原测试标签是否符合 并统计
        (*sum)++; //统计符合的数量
    }
}

int cmp(const void* a, const void* b) { //快速排序qsort函数的cmp函数(判断函
数)
    Distance A = *(Distance*)a;
    Distance B = *(Distance*)b;
    return A.data > B.data ? 1 : -1;
}

int main()
{
    omp_set_num_threads(8);
    double start_time, end_time;
    char filename[20]={"support_data.txt"};
    tempdata(filename); //加载临时数据->打乱数据
    loaddata(); //加载打乱后的数据并分割
    int i, j;
    int k=KN; //k值
    int sum = 0; //用于统计距离最近的k个标签与原测试标签符合的数量

    start_time = omp_get_wtime();

    // 对每条测试数据与训练数据计算距离的过程进行并行
    #pragma omp parallel for

```

```

    for (i = 0; i < NUM1; i++) {
        #pragma omp parallel for
        for (j = 0; j < NUM2; j++) {
            // printf("i = %d    j = %d\n", i, j);
            distance[j].data = computedistance(i,j); //把计算好的距离依次存
入distance结构体数组中
            distance[j].trainlabel = train[j].label; //以上距离存入的同时也
把训练集标签一起存入distance结构体数组中
        }
        qsort(distance, NUM2, sizeof(distance[0]), cmp); //用qsort函数从
小到大数据排序测试数据与每组训练数据的距离
        countlabel(&sum, k, i); //统计距离测试集标签最近的标签出现频数
    }
    end_time = omp_get_wtime();
    // printf("Sequential Time: %f\n", end_time-start_time);
    printf("Parallel Time: %f\n", end_time-start_time);
    // printf("K = %d    P = %.1lf%%\n", k, 100.0*(sum)/NUM1);
    sum = 0; //每次统计完后都赋值0 便于下一个测试数据统计

    return 0;
}

```

## Results and Analysis:

优化前串行时间：

```

● xiaoli@xiaoli-KLVC-WXX9:~/Project/Parallel-computing$ g++ -fopenmp lab6_shopping.cpp -o output
● xiaoli@xiaoli-KLVC-WXX9:~/Project/Parallel-computing$ ./output
Sequential Time: 53.554447

```

优化后并行时间：

```

● xiaoli@xiaoli-KLVC-WXX9:~/Project/Parallel-computing$ g++ -fopenmp lab6_shopping.cpp -o output
● xiaoli@xiaoli-KLVC-WXX9:~/Project/Parallel-computing$ ./output
Parallel Time: 14.080254

```

计算得加速比为3.8。

## Conclusion:

从最后的输出结果我们可以看出，openmp 并行显著降低了 knn 的计算时间，验证了我们最开始的猜想。