

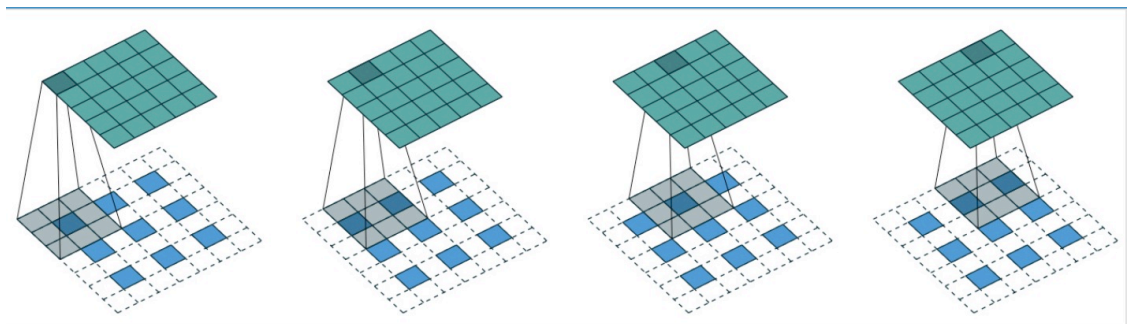
# 并行与可视化计算导论大作业报告

## 0. Transposed Convolution

### 0.1 背景介绍

转置卷积 (Transposed Convolution) 也叫反卷积/fractionally strided convolution。由于卷积层的前向操作可以表示为和矩阵相乘，相应地很容易得到卷积层的反向传播就是和矩阵的转置相乘。相对于卷积提取图片特征的降维作用，反卷积能够根据输入的图片特征还原为原图片。随着反卷积在神经网络可视化上的成功应用，其被越来越多的工作所采纳比如：场景分割、生成模型等。

### 0.2 实验步骤



如上图所示，反卷积实质上是二维卷积的逆运算，要实现反卷积包括以下步骤：（以串行算法步骤为例）

（1）根据输入的核矩阵 (filter) 维数与原矩阵内容，进行 padding，即在原矩阵的每个元素四周都加上 0，并且要适当地根据 filter 的大小对边界进行补 0。

（2）对图片矩阵的每一个 channel，将 filter 与 padding 过后的原矩阵根据给定的步长要求，进行卷积操作，得到针对每个 channel 的输出。

（3）对每个 channel 的输出进行累加，最后得到矩阵的转置卷积的结果。

由于题目给定输入为 32-channel 的  $1920 \times 1080$  矩阵，filter 为  $9 \times 9$ ，因此 padding 过后的矩阵大小为  $(2 \times 1920 + 8) \times (2 \times 1080 + 8) = 3848 \times 2168$ 。由于选定步长为  $1/2$ ，只需要对 padding 过后的矩阵进行步长为 1 的基本矩阵卷积操作即可。本文将介绍逆卷积的并行实现的各种细节，并将其与串行实现进行比较。

## 0.3 实验内容

### 0.3.1 数据生成

编写生成特定维数的数据矩阵以及特定的核矩阵(filter)，以便对代码进行多次检查与比较。包含在代码包中的 `put_data.cpp` 中，缺省随机生成  $32 \times 1920 \times 1080$  的数据矩阵与  $9 \times 9$  的核矩阵，考虑到计算成本开销，每个矩阵元素在 0-9 之间，生成的矩阵分别存在 `A.txt` 与 `kernel.txt` 中。

### 0.3.2 串行实现

串行执行步骤如上所述。实现过程中将 (2) (3) 合并，每次迭代将 (1) 与 (2) + (3) 严格串行化处理，共进行 32 次迭代。

## 0.4 并行实现

对于转置卷积，我们认为主要存在两个可以并行的地方：

(1) 每个 channel 之间的计算独立，可以并行（“粗粒度”）

(2) 对于同一个 channel 而言，不同位置的元素的 padding 操作可以并行，不同位置的元素的卷积操作可以并行。但我们认为对于一个 channel 而言，必须要所有位置的元素都经过 padding 操作，才能进行下一步的卷积操作（“细粒度”）

（考虑到一边 padding 一边卷积涉及到时间开销较大的标志位操作以及复杂的矩阵计算，因此将二者串行化处理）

### 0.4.0 编程环境

本机：

CPU: 2.9 GHz Intel Core i5

Intel Iris Graphics 550 1536 MB

内存: 8 GB 2133 MHz LPDDR3

OS: MacOS 10.13.4

IDE(虚拟机) : Microsoft Visual Studio 2017 + OpenCL

服务器：

222.29.98.19 以及相应配置

222.29.98.21 以及相应配置

### 0.4.1 OpenCL

OpenCL 是一个为异构平台编写程序的框架，此异构平台可由 CPU，GPU 或其他类型的处理器组成。通过多个工作项（Work-item，即线程）执行同样的核函数，每个 Work-item 都有一个唯一固定的 ID 号，一般通过这个 ID 号来区分需要处理的数据。多个工作项组成一个工作组（Work-group），Work-group 内的这些 Work-item 之间可以通信和协作。

反卷积的并行实现如下：

- 1、设置 32 个 work-group 来处理不同 channel 的 padding 与卷积计算。每个 work-group 根据任务不同分为 8 个（padding）和 16 个（convolution）。padding 和 卷积操作串行，利用设置不同的核函数，然后分别调用 `clEnqueueNDRangeKernel` 来入队操作，循环 K 次。（K 为人工设定，初始值为 1，用来计算多次运行时间求平均）
- 2、padding 部分：每个 work-group 中含有 8 个 work-item，每个 work-item 负责连续的特定行数范围的子矩阵 padding，即负责  $1920/8 = 240$  行的 padding 工作。
- 3、卷积计算：每个 work-group 中含有 16 个 work-item，对 padding 后的矩阵（ $3848 \times 2168$ ）的连续的特定行数范围的子矩阵进行卷积操作。每次计算完乘积后累加在原数据项上。

work-group 和 work-item 的设置如下，其中要保证局部 item 数能够被全局 item 数整除，否则报错“Enqueueing kernel error”。

```
size_t globalThreads1[] = {8};
size_t localThreads1[] = {2};
size_t globalThreads2[] = {32*16};
size_t localThreads2[] = {16};
```

```

errNum = clEnqueueNDRangeKernel(commandQueue, kernel1,
                                1, NULL, globalThreads1,
                                localThreads1, 0,
                                NULL, NULL);
if (errNum != CL_SUCCESS) {
    printf("Error: Enqueuing kernel1\n");
}

```

```

errNum = clEnqueueNDRangeKernel(commandQueue, kernel2,
                                1, NULL, globalThreads2,
                                localThreads2, 0,
                                NULL, NULL);
if (errNum != CL_SUCCESS) {
    printf("Error: Enqueuing kernel2\n");
    return -1;
}

```

图 1: work-group/item 设置

与串行实现相比，使用 OpenCL 可以更充分利用 GPU，通过数据交换来尽可能提高并行度，但是也带来了额外的通信开销。

#### 0.4.2 增加 work-item 数

由于更多的 work-item 数可以将任务划分得更细，从而更好地发挥计算能力强的设备的作用，于是考虑将卷积部分的 work-item 数目增加为每组 32、64 个。

```

size_t globalThreads1[] = {32*8};
size_t localThreads1[] = {8};
size_t globalThreads2[] = {32*32};
size_t localThreads2[] = {32};

```

```

size_t globalThreads1[] = {32*8};
size_t localThreads1[] = {8};
size_t globalThreads2[] = {32*64};
size_t localThreads2[] = {64};

```

### 0.4.3 每个 work-item 负责行数的位置

不是将连续的行划分到一个 work-item 工作集，而是对于 work-item 的 id，让它负责行数 $\%(\text{work-group size})$ 的离散行数。

左边为离散行号，右边为连续行号。



```

group_num = 32;
local_num = 16;
int group_id = global_index/group_num;
int local_id = global_index%group_num;
unsigned int _size = 2*rsz/size/loc_num;
unsigned int _st = local_id * _size;
for(int i = _st; i < 2*rsz; i+=_size)
{
    for(int j = 0; j < 2*csz; ++j)
    {
        Y[group_id * (2*rsz) * (2*csz) + i * (2*csz) + j] = 0;
        for(int l = 0; l < ksz; ++l)
        {
            for(int k = 0; k < ksz; ++k)
            {
                Y[group_id * (2*rsz) * (2*csz) + i * (2*csz) + j] += B[group_id *
                    (2*rsz+ksz-1) * (2*csz+ksz-1) + (i+l)*(2*csz+ksz-1)+(j+k)]*X[l*ksz+k];
            }
        }
    }
}
}

```

#### 0.4.4 每个 work-item 负责不同列数

为了调查划分的任务集的局部性是否会影响并行计算效率，特定将按行号划分改为按照列号划分，一次调查局部性对性能的影响。还是每组 16 个 work-item，结果与预期相差较大，留作以后讨论。

## 0.5 性能评测

以下给出串行测试结果与初始并行测试结果。

（这里忽略 padding 的赋值，将乘法与加法看作一次 operation）

```

[s1500012946@ceca21 ~]$ ./test A.txt kernel.txt
NVIDIA Corporation
-----serial performance:-----
Total M ops = 133
Time in s: 103.502381
Test performance [G OP/s] : 0.001282

```

串行测试结果

```

[s1500012946@ceca21 ~]$ ./test A.txt kernel.txt
NVIDIA Corporation
-----parallel performance:-----
Total M ops = 133
Time in s: 57.059828
Test performance [G OP/s] : 0.002326
-----parallel performance end-----

```

## 并行测试结果

串行与 MPI 并行对比：（MPI 在进程数为 1 时相当于串行）：

串行算法	OpenCL	64-work-item+ 离 散行号	128-work-item+ 离 散行号
103.5s	57.1s	39.4s	46.4s

可见，采用 OpenCL 加速比达到了 50%左右，适当增加 work-item 数能进一步提升性能。离散行号的作用不明显，猜测是同一个 work-group 的 work-item 共享同一片地址空间导致通信开销极小，本身局部性就很好。

## 0.6 总结及感悟

1、充分利用局部性与 GPU 并行，可以看见虽然实验对 work-item 的分配做了不少优化的尝试，但是提升效果都不明显，而一开始实现的按行划分的 work-item 任务集，已经相对于并行的版本提高了不少，可见挖掘转置卷积的并行度是提升性能的关键。

2、合理划分 work-item 的任务集大小，控制并行粒度与数据传输开销的平衡，work-item 的数目是有上界的。并行的粒度并非是越细越好。随着粒度降低，加载数据、设置设备的开销可能增大，反而造成并行效率降低。

本次实验本来计划尝试脚本遍历最适合的 work-group 大小，但由于时间关系没有进行尝试，是本次实验的一大遗憾。

# 1. K-Nearest Neighbors

## 1.1 背景介绍

K 近邻算法是一个常用的机器学习算法，其分类思路是：给定一些已知类别的样本点，对于新添加的样本，如果它的 K 个“距离”最近的样本大多属于某一个类别，那么这个样本也属于这个类别。

K 近邻算法思路简单清晰，一个显著的不足在于计算量较大，对于每一个待分类的样本，我们都需要计算它到全体已知样本的距离，样本维度较高、采用欧式距离时需要大量的乘法运算。一个对于计算量的优化思路就是并行：计

算待分类样本到不同已知样本的距离彼此之间不存在依赖关系，明显存在并行改进的空间。

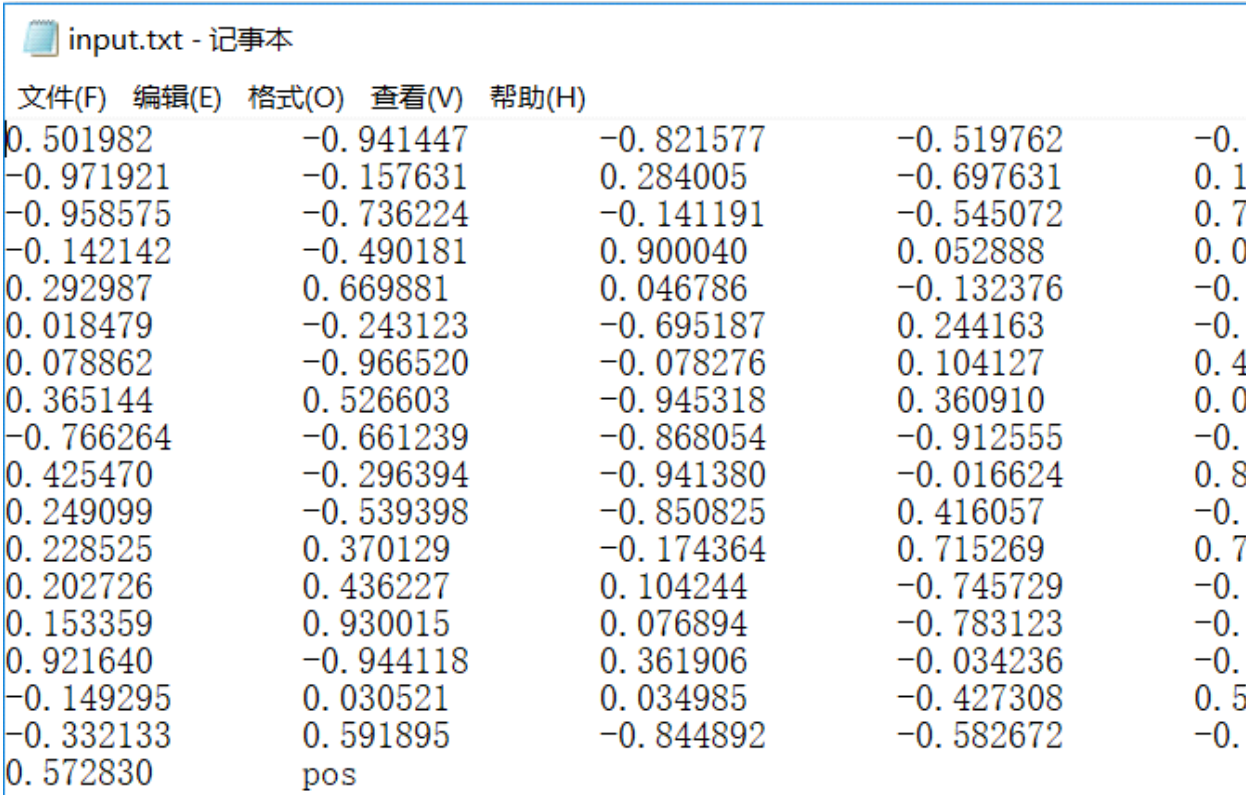
本文将介绍 KNN 算法的不同并行实现，并与串行实现进行比较。

## 1.2 实验准备

### 1.2.1 数据生成

实验所需数据采用 Python 进行生成，形式为：

其中  $x$  表示样本的不同特征，label 表示样本的类别。简单起见，若，则样本定为正类（记作 pos），若，则样本记作负类（记作 neg）。数据生成程序有两个参数，分别是数据数量和数据的特征维数。



0.501982	-0.941447	-0.821577	-0.519762	-0.
-0.971921	-0.157631	0.284005	-0.697631	0.1
-0.958575	-0.736224	-0.141191	-0.545072	0.7
-0.142142	-0.490181	0.900040	0.052888	0.0
0.292987	0.669881	0.046786	-0.132376	-0.
0.018479	-0.243123	-0.695187	0.244163	-0.
0.078862	-0.966520	-0.078276	0.104127	0.4
0.365144	0.526603	-0.945318	0.360910	0.0
-0.766264	-0.661239	-0.868054	-0.912555	-0.
0.425470	-0.296394	-0.941380	-0.016624	0.8
0.249099	-0.539398	-0.850825	0.416057	-0.
0.228525	0.370129	-0.174364	0.715269	0.7
0.202726	0.436227	0.104244	-0.745729	-0.
0.153359	0.930015	0.076894	-0.783123	-0.
0.921640	-0.944118	0.361906	-0.034236	-0.
-0.149295	0.030521	0.034985	-0.427308	0.5
-0.332133	0.591895	-0.844892	-0.582672	-0.
0.572830	pos			

图 1：一个 120 维的正类样本

### 1.2.2 串行实现

KNN 算法的串行算法如下：

- 1、读入数据（取总数据的前 10%作为测试数据，后 90%作为训练数据）
- 2、对数据进行归一化处理



- 3、对于每一个测试样本，计算它与所有训练样本的距离
- 4、对所有距离进行排序，统计最近的 K 个样本的类别
- 5、输出预测类别

## 1.3 并行实现

对于 KNN 算法，我们认为主要存在两个可以并行的地方：（1）测试样本与训练样本的距离计算；（“粗粒度”）（2）距离计算中每一维的差平方和（“细粒度”）。

### 1.3.0 编程环境

CPU: Intel Core i5 4210H 2.90 GHz, 2 核心, 支持超线程

内存: 8GB DDR3

GPU: Nvidia Geforce GTX 960M 4GB GDDR5 (OpenCL 限定为使用独立显卡)

OS: Windows 10 64bit

IDE: Microsoft Visual Studio 2017 + MS-MPI + OpenCL

### 1.3.1 MPI

MPI（消息传递接口）是一种常用的并行编程技术，基本模式为不同节点并行运算，采用 Send/Receive 相互通信。

KNN 算法的 MPI 并行实现如下：

- 1、0 号进程读入数据，并对数据进行归一化处理
- 2、0 号进程划分数据，调用 Send 接口将训练样本分发给其他进程
- 3、for i=0 to testSize

各进程分别计算自己的样本点到测试样本的距离

将前 k 近的点发送给 0 号进程

0 号进程对所有进程的 k 近邻进行排序，得到总的 k 近邻，输出结果

为了降低进程之间的通信开销，我们利用 MPI 的自定义数据类型将样本的 ID 和距离捆绑发送：

```

103 //自定义主从进程传送数据类型
104 MPI_Datatype oldTypes[2] = { MPI_INT,
105 MPI_Datatype mpi_result;
106 int blockLength[2] = { 1, 1 };
107 MPI_Aint offset[2] = { 0, sizeof(double)
108 MPI_Type_create_struct(2, blockLength,
109 MPI_Type_commit(&mpi_result);

```

图 2: MPI 自定义数据类型

与串行实现相比，使用 MPI 可以更充分利用多核 CPU，并且降低了每一结点的排序工作量，带来的额外开销主要是通信开销。

### 1.3.2 MPI+OpenCL

如前文所述，KNN 算法另一个并行优化点是两样本之间的欧式距离计算，但样本维数较高，如果一个节点一维的话显然是不现实的，通信开销也较大。所以我们考虑利用 GPU。OpenCL 是一个用于异构平台的并行计算框架，通过 OpenCL 我们可以利用 GPU。

KNN 算法的 MPI+OpenCL 并行实现如下：

- 1、0 号进程读入数据，并对数据进行归一化处理
- 2、0 号进程划分数据，调用 Send 接口将训练样本分发给其他进程
- 3、for i=0 to testSize

对于每一个训练样本到测试样本的距离，通过 OpenCL 接口调用 GPU 计算将前 k 近的点发送给 0 号进程

0 号进程对所有进程的 k 近邻进行排序，得到总的 k 近邻，输出结果

与 MPI 实现相比，MPI+OpenCL 主要是在计算距离这一步引入了 GPU，但由此引入了大量 OpenCL 的相关代码，反复的 EnqueueWriteBuffer 可能带来了不小开销。

```

20 //openCL kernel
21 const char *source =
22 "__kernel void clGetDistance(int dimension, __global double *p1, __global double *p2, __global double *distSquare) {\n"
23 "{\n"
24 "  int tid = get_local_id(0);\n"
25 "  int tsize = get_local_size(0);\n"
26 "\n"
27 "  double sumSquare = 0;\n"
28 "  for(int i = tid; i < dimension; i += tsize) {\n"
29 "    sumSquare += (p1[i] - p2[i]) * (p1[i] - p2[i]);\n"
30 "  }\n"
31 "  __local double tmp[128];\n"
32 "  tmp[tid] = sumSquare;\n"
33 "  barrier(CLK_LOCAL_MEM_FENCE);\n"
34 "  if (tid == 0){\n"
35 "    for (int i = 1; i < 128; i++){
36 "      tmp[0] += tmp[i];
37 "    }\n"
38 "    *distSquare = tmp[0];\n"
39 "  }\n"

```

图 3: OpenCL 实现对应的 kernel, 不同维度的差平方分给不同 workitem 计算

```

274 //try openCL
275 cl_command_queue mQueue;
276 mQueue = clCreateCommandQueue(mContext, device, 0, &err);
277 cl_mem p1 = clCreateBuffer(mContext, CL_MEM_READ_ONLY, dimension
278 cl_mem p2 = clCreateBuffer(mContext, CL_MEM_READ_ONLY, dimension
279 cl_mem res = clCreateBuffer(mContext, CL_MEM_WRITE_ONLY, sizeof(dou
280 err = clEnqueueWriteBuffer(mQueue, p1, CL_TRUE, 0, sizeof(double) * dim
281 err = clEnqueueWriteBuffer(mQueue, p2, CL_TRUE, 0, sizeof(double) * dim
282 double square;
283 clSetKernelArg(mykernel, 0, sizeof(int), &dimension);
284 clSetKernelArg(mykernel, 1, sizeof(cl_mem), &p1);
285 clSetKernelArg(mykernel, 2, sizeof(cl_mem), &p2);
286 clSetKernelArg(mykernel, 3, sizeof(cl_mem), &res);
287 size_t globalWorkSize[1];
288 size_t localWorkSize[1];
289 globalWorkSize[0] = 64;
290 localWorkSize[0] = 64;
291 err = clEnqueueNDRangeKernel(mQueue, mykernel, 1, NULL, globalWorkS
292     localWorkSize, 0, NULL, NULL);
293 clFinish(mQueue);
294 clEnqueueReadBuffer(mQueue, res, CL_TRUE, 0, sizeof(double), &square,

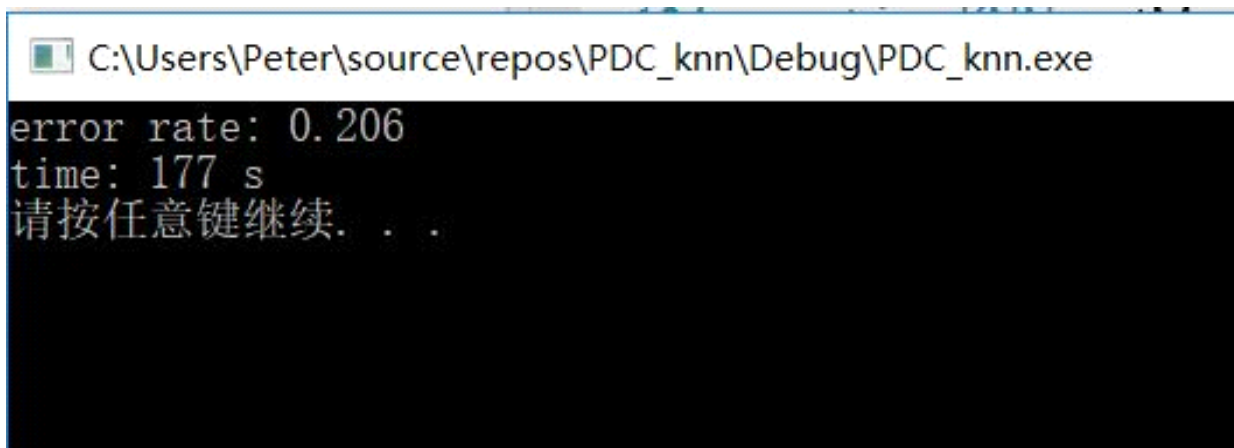
```

图 4：将数据写到 device，计算后读出结果

另外一种使用 MPI+OpenCL 的并行思路则是按行划分，即使用 OpenCL 并行计算一个进程分配到的各个样本到测试样本的距离。与前一种思路相比，这种思路带来的额外开销相对较小。

## 1.4 性能评测

按照 ppt 要求，数据采用 10000 个样本，K=18，每个样本为 120 维，10%的样本为测试数据。为了减少 IO 干扰，程序只统计 error rate，如果并行算法得到的 error rate 与串行算法一致，则说明并行算法是正确的。



```

C:\Users\Peter\source\repos\PDC_knn\Debug\PDC_knn.exe
error rate: 0.206
time: 177 s
请按任意键继续. . .

```

图 5：串行测试结果

```
Windows PowerShell
PS C:\Users\Peter\source\repos\PDC_knn_MPI\x64\Debug> mpiexec
error rate: 0.206000
time: 80 s
PS C:\Users\Peter\source\repos\PDC_knn_MPI\x64\Debug> mpiexec
error rate: 0.206000
time: 107 s
PS C:\Users\Peter\source\repos\PDC_knn_MPI\x64\Debug> mpiexec
error rate: 0.206000
time: 85 s
PS C:\Users\Peter\source\repos\PDC_knn_MPI\x64\Debug>
```

图 6：MPI 测试结果

MPI

串行与 MPI 并行对比：（MPI 在进程数为 1 时相当于串行）：

串行算法	MPI（2 进程）	MPI（4 进程）	MPI（8 进程）
177s	107s	80s	85s

程序运行时 CPU 占用保持在 100%，说明确实充分利用了 CPU。可以观察到 MPI 的加速比还是比较可观的，2 进程时为  $177/107=1.65$ ，4 进程时为  $177/80=2.21$ ，8 进程时运行时间相比 4 进程反而上升，分析是因为本机 CPU 算上超线程也只有 4 个并发运行的线程，8 进程意义不大，反而可能引入进程切换开销。

MPI+OpenCL

进程数为 4，workgroup 为 1，workitem 为 64 时，1000 个样本运行时间：28s

注：10000 个样本时间过长，放弃测试

增加 OpenCL 并行计算欧式距离后，理论上运算时间应该减少，但在测试中却出现了时间增加的情况。经过调试检查和比对最后输出的 error rate，OpenCL 的运算是正确的。推测是因为粒度过细，运算不够密集，大量的时间消

耗在数据传输上。观察任务管理器的“性能”界面，可以发现 GPU 虽然用上了，但占用的主要是“Copy”资源，可以作为上述猜测的佐证。

MPI+OpenCL 改进

推测粒度过细导致的通信开销可能是时间上升的原因，于是我们又尝试将 OpenCL 并行的粒度增大，即每个 workitem 计算两点之间的距离，一个进程有 64 个 workitem。

```
20 //openCL kernel
21 const char *source =
22     "__kernel void clGetDistance(int dimension, int size, __global double *p1,
23     \"{\\n\"
24     \"    int tid = get_local_id(0);\\n\"
25     \"    int tsize = get_local_size(0);\\n\"
26     \"    double sumSquare = 0;\\n\"
27     \"\\n\"
28     \"    for (int row = tid; row < size; row += tsize){\\n\"
29     \"        for (int j = 0; j < dimension; j++){\\n\"
30     \"            sumSquare += (p1[j] - p2[j + row * dimension]) * (p1[j] - p2[j + r
31     \"        }\\n\"
32     \"        distSquare[row] = sumSquare;\\n\"
33     \"        sumSquare = 0;\"
34     \"    }\\n\"
35     \"}\\n\";
```

图 7：增大并行粒度后的 OpenCL kernel

测试结果：（10000 个测试样本）

进程数为 1	进程数为 4
157s	63s

可以观察到改进的 MPI+OpenCL 实现相比串行和单独使用 MPI 都有一定的性能提升，说明此时 OpenCL 减少的运算时间>访存开销，改进有效。

另外我们在实验中注意到了一个问題：只有 1 个进程时，error rate 始终与串行一致，说明此时 OpenCL 的同步互斥不存在问题；但当进程数>1 时，并行程序有一定概率出现与串程序 error rate 不一致的情况，推测是不同进程并行调用 OpenCL 可能还存在同步互斥的问题。查询网上资料，每个进程拥有各

自的 command queue 时确实可能出现问题，共用 command queue 则不会，但在 MPI 模型下感觉共用 command queue 比较困难。由于此问题不影响性能分析，我们将其留待后续解决。

## 1.5 总结及感悟

对于 KNN 算法，本文给出了串行实现和 MPI、MPI+OpenCL 两种并行，并分别取得了一定的加速效果。我们在实验中主要有两点感悟：

1、并行的粒度并非是越细越好。随着粒度降低，加载数据、设置设备的开销可能增大，反而造成并行效率降低。

2、MPI 模型由于采用消息的收发进行抽象，在编程时较为容易。但如果需要全局变量，则不如共享内存的并行模型实现简单。因此 MPI 可能更适合在集群上使用。

本次实验本来还计划尝试 Nvidia 的 CUDA，但由于环境配置较为繁琐，没有进行尝试。CUDA 对于 KNN 算法的作用与 OpenCL 类似，但可能会因为 Nvidia 的优化而更加高效一些。