并行与分布式程序设计大作业报告

组号：Group14

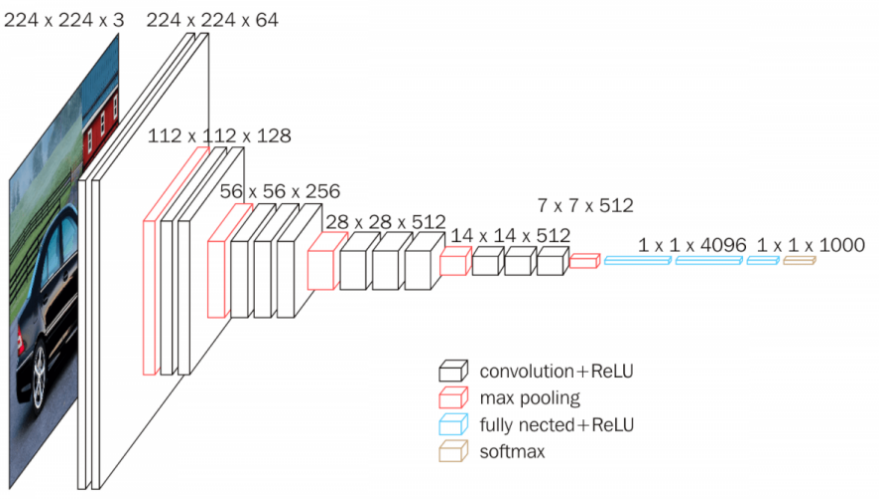
神经网络模型：VGG16

小组成员	组内分工	贡献度
林宇欣 - 220030910004	算法设计；代码实现；成果展示	1/3
王祎雯 - 220030910002	算法设计；代码实现；文档撰写	1/3
钟博子韬 - 220030910014	算法设计；代码实现；文档撰写	1/3

1 模型介绍及参数解析

1.1 VGG16模型简介

VGG16是一种得到了广泛运用的卷积神经网络 (CNN) 架构，以卷积层为其核心。一个典型的VGG16网络结构如下图所示：

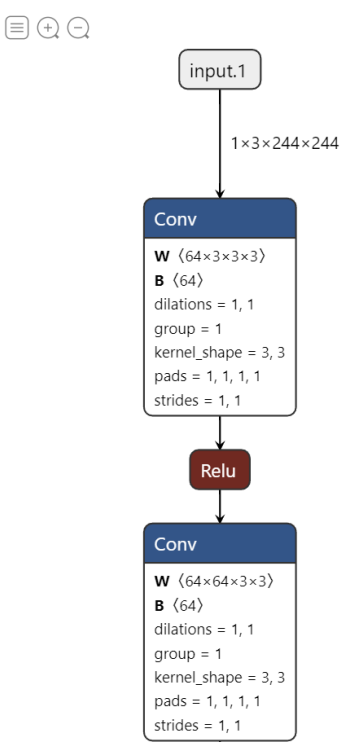


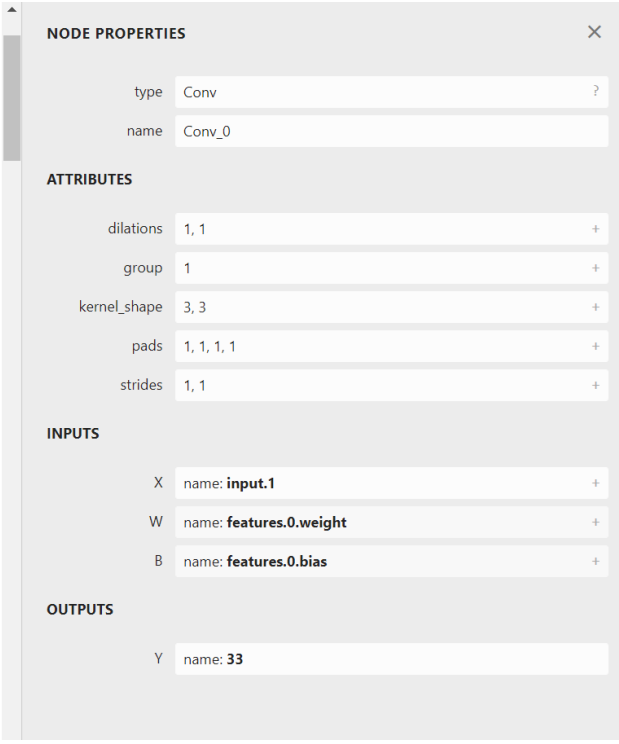
在本次大作业中，我们使用的VGG16模型包括13个卷积层和3个全连接层，采用ReLU作为激活函数，中间执行5次最大池化操作。与通常情况有所区别的地方是，我们的模型以3×244×244的三维矩阵作为输入，并输出1×1000的一维数组。

我们的任务是基于已完成训练的模型参数，使用CUDA编程实现并加速相应的VGG16网络，使其能够完成神经网络推理的完整过程。此处我们的网络不需要进行迭代训练，因此只包括前向计算的实现，不考虑误差反向传播。

1.2 ONNX文件解析

已完成训练的模型参数由一个onnx类型的VGG16模型提供。我们利用https://netron.app/网站解析模型结构和模型参数，前者包括模型的整体构造和每层节点的属性数据，后者包括每层神经元的偏差与权重。以第一个卷积层为例，如下图所示：





在VGG16网络中，共有16层，也就是前文所述的13个卷积层和3个全连接层需要下载训练所得的模型参数。从上述网站中下载的偏差与权重以npy文件的形式存储，我们将其全部展平为了一维数组，并存储为txt文件形式，以便于C++语境下的读取。

1.2.1 卷积层重要参数

- kernel_shape = 3,3：声明卷积操作采用大小为3×3的卷积核。
- pads = 1,1,1,1：沿除去通道空间轴以外的两个空间轴，在矩阵的4个方向上填补宽度为1的0。
- features.weight：其形状为”输出通道数×输入通道数×卷积核高度×卷积核宽度“，集合了”输出通道数×输入通道数“个卷积核的权重。
- features.bias：其形状为”输出通道数“，集合了输出矩阵所有通道上的偏差值。

1.2.2 池化层重要参数

- kernel_shape = 2,2：声明最大池化操作采用大小为2×2的池化核。
- strides = 2,2：声明池化核在除去通道空间轴以外的两个空间轴上扫描的步长为2，使得采样空间不发生重叠。

1.2.3 全连接层重要参数

- classifier.weight：其形状为“输出向量大小×输入向量大小”，用于与输入向量进行矩阵相乘。
- classifier.bias：其形状为“输出向量大小”，用于对输出向量中的数值进行依次调整。

2 网络结构构建

2.1 数据读取

2.1.1 输入与输出数据

本次实验共有10个测试用例和10个对应的输出，以txt文件形式存储，使用统一规定的接口进行读取。此处为了摒除不必要的warning，我们将文件名类型改为了`string`。

```
float inputArr[TESTNUM][INPUTSHAPE];
float benchOutArr[TESTNUM][OUTPUTSHAPE];

void readInput(string filename);
void readOutput(string filename);
```

2.1.2 权重与偏差数据

本次实验需要读取13个卷积层的偏差数据，最大长度为512；13个卷积层的权重数据，最大长度为512×512×2×3；3个全连接层的偏差数据，最大长度为4096；3个全连接层的权重数据，最大长度为4096×25088。上述数据以txt文件形式存储，并在神经网络推理开始前使用`initModel()`函数一次完成读取。

```
float conv_bias[13][512];
float conv_weights[13][512*512*3*3];

float gemm_bias[3][4096];
float gemm_weights[3][4096*25088];

void read_para(string para_path, float *para);
void initModel();
```

2.2 函数封装

2.2.1 层函数封装

- 卷积层：函数包括padding操作和卷积计算两个过程，使用`layer_id`指定卷积层，以便于后续选取相应的偏差和权重参与计算。卷积计算可以选择调用CUDA函数和直接进行非并行计算两种方式。为了减少分支判断带来的延迟，此处仅将不被使用的代码放入注释。

```
void layer_conv(int layer_id, float *input, float *output, int input_channel, int output_channel, int out_height){
    float *padding_input;
    padding_input = (float *)malloc((input_channel*(out_height+2)*(out_height+2))*sizeof(float));
    padding(input, padding_input, input_channel, out_height);
    /********************** gpu ************************/
    cuda_conv_relu(layer_id, padding_input, output, input_channel, output_channel, out_height);
    /***************************************************/
    /********************** cpu ************************
    ...
    /***************************************************/
    free(padding_input);
}
```

- 池化层：函数包括最大池化采样的过程，可以选择调用CUDA函数和直接进行非并行计算两种方式。

```
void layer_maxpool(float *input, float *output, int channel, int in_height, int
out_height){
    /********************** gpu **************************/
    cuda_maxpool(input, output, channel, in_height, out_height);
    /*****************************************************/
    /********************** cpu **************************
    ...
    /*****************************************************/
}
```

- 全连接层：函数包括全连接层进行矩阵乘法的过程，使用`layer_id`指定全连接层，使用`bool`类型的`relu`指定是否使用激活函数。可以选择调用CUDA函数和直接进行非并行计算两种方式。

```
void layer_gemm(int layer_id, float *input, float *output, int input_size, int
output_size, bool relu){
    /********************** gpu **************************/
    cuda_gemm(layer_id, input, output, input_size, output_size, relu);
    /*****************************************************/
    /********************** cpu **************************
    ...
    /*****************************************************/
}
```

2.2.2 推理过程函数封装

- 结构函数：以3×244×244的矩阵作为输入，以1×1000的数组作为输出。调用13次卷积层封装函数，5次池化层封装函数，3次全连接层封装函数。以第二次卷积操作为例，完成一层神经网络推理的过程包括：
  - 定义输出向量，分配指定大小的内存空间；
  - 调用相应的层函数，以上一层的输出作为输入，将输出放入方才定义的向量中；
  - 释放上一层的输出向量所占据的内存空间。

```
void inference(float *image, float *prob){
    ...
    float *layer2_output = (float *)malloc((64*244*244)*sizeof(float));
    layer_conv(2, layer1_output, layer2_output, 64, 64, 244);
    free(layer1_output);
    ...
}
```

- 主函数：使用统一规定的主函数。由于cc文件调用cu文件较为复杂，此处将主函数也存储为cu文件，其他基本不作改动。

3 网络功能实现

3.1 卷积层

3.1.1 初步并行

- 线程配置：在Grid中，每个Block负责完成一个输出通道内的卷积计算；在Block中，每个Thread负责完成一个输入通道内的卷积计算。理想情况下，卷积计算中的每个二维卷积核扫描每个相应的二维矩阵的操作都可以并行完成。

```
dim3 dimGrid(output_channel);
dim3 dimBlock(input_channel);
```

- 卷积计算：在核函数内，使用指定卷积核对指定输入通道的二维矩阵进行扫描，并将卷积结果存入形状为“输出大小×输入通道数”的中间输出中。在Host端处理中间输出，将对应同一输出通道的不同输入通道内的卷积结果进行累加，存入输出，并加上相应的偏差值。相应实现函数如下：

```
__global__ static void conv_back(float *input, float *output, float *weights, int
*input_channel, int *out_height);
void cuda_conv_relu_back(int layer_id, float *input, float *output, int input_channel,
int output_channel, int out_height);
```

3.1.2 共享内存优化

由于在Host端进行卷积结果累加的操作效率太低，此处引入shared memory的概念，也就是利用同一个Block中不同Thread间的共享内存来实现计算结果的汇总。其中，卷积核的形状为3×3，输入通道数最多为512，即一个三维卷积核进行的卷积计算最多需要存储512×3×3个中间数据用于后续累加。

```
__shared__ float shared[512][3][3];
```

在每一个Thread中，卷积核需要执行“输出高度×输出宽度”次卷积操作。利用`__syncthreads()`函数，我们可以实现线程的同步，保证每一个Block中的所有Thread都在对不同通道中同一位置下的一个3×3大小的掩膜进行卷积操作。在第0个Thread中，我们将所有的数据进行累加。

```
for(int m=0; m<oheight; m++){ // 输出高度
    for(int n=0; n<oheight; n++){ // 输出宽度
        __syncthreads();
        ... // 将输入数据与卷积核权重依次相乘，并将结果存入共享内存
        __syncthreads();
        if(tid == 0){ // 避免重复累加导致结果错误
            ... // 将共享内存中的所有数据进行累加
        }
    }
}
```

完成优化后的函数如下：

```
__global__ static void conv(float *input, float *output, float *weights, int
*input_channel, int *out_height);
void cuda_conv_relu(int layer_id, float *input, float *output, int input_channel, int
output_channel, int out_height);
```

3.2 池化层

- 线程配置：在Grid中，每个Block负责完成一个通道内的最大池化操作；在Block中，每个Thread负责完成一列四宫格的最大池化操作。理想情况下，在一个Thread完成一列数据采样的时间内，所有的采样工作都将完成。

```
dim3 dimGrid(channel);
dim3 dimBlock(out_height);
```

- 采样操作：需要注意的是，由于测试数据的形状为3×244×244，池化操作存在输入高度和宽度为奇数的情况。此处我们对输入数据进行上采样，也就是说若输入高度和宽度为奇数，直接忽略最下一行和最右一列的数据。在核函数内，每个Thread只执行“输出高度”次数的采样，并将四宫格内的最大数据存至输出的相应位置中。

```
    for(int i=0; i<oheight; i++){ // 输出高度
        float max_temp = 0; // 存储最大值（由于激活函数为ReLU，最大值不会小于0）
        if(input[bid*iheight*iheight + (2*i)*iheight + 2*tid] > max_temp)
            max_temp = input[bid*iheight*iheight + (2*i)*iheight + 2*tid];
        if(input[bid*iheight*iheight + (2*i+1)*iheight + 2*tid] > max_temp)
            max_temp = input[bid*iheight*iheight + (2*i+1)*iheight + 2*tid];
        if(input[bid*iheight*iheight + (2*i)*iheight + 2*tid+1] > max_temp)
            max_temp = input[bid*iheight*iheight + (2*i)*iheight + 2*tid+1];
        if(input[bid*iheight*iheight + (2*i+1)*iheight + 2*tid+1] > max_temp)
            max_temp = input[bid*iheight*iheight + (2*i+1)*iheight + 2*tid+1];
        output[bid*oheight*oheight + i*oheight + tid] = max_temp;
    }
```

3.3 全连接层

3.3.1 *初步并行*

- 线程配置：在Grid中，每个Block负责完成一行输入和一列权重的矩阵乘法；在Block中，每个Thread负责完成“输入向量大小/512”次乘法和“(输入向量大小/512)-1”次累加。

```
dim3 dimGrid(output_size);
dim3 dimBlock(512);
```

- 矩阵乘法：在核函数内，每个Thread分别执行各自负责的相乘和累加操作，并将结果存入中间输出中。在Host端对中间输出执行累加操作，存入输出，并加上相应的偏差值。相应实现函数如下：

```
__global__ static void gemm_back(float *input, float *output, float *weights, int
*input_size);
void cuda_gemm_back(int layer_id, float *input, float *output, int input_size, int
output_size, bool relu);
```

3.3.2 共享内存优化

与卷积层的计算类似，此处我们引入shared memory来提高计算效率。由于每个Block中都分配了512个Thread，共享内存的大小可以直接设定为512。同样的，我们使用`__syncthreads()`函数进行同步，并在第0个Thread中对共享内存进行初始化和完成计算后的累加。

```
__shared__ float shared[512];
if(tid == 0){
        ... // 初始化shared
    }
    __syncthreads();
    for(int i=tid; i<(*input_size); i+=512)
        shared[tid] += input[i] * weights[bid*(*input_size) + i];
    __syncthreads();
    if(tid == 0){
        ... // 将共享内存中的所有数据进行累加
    }
```

完成优化后的函数如下：

```
__global__ static void gemm(float *input, float *output, float *weights, int
*input_size);
void cuda_gemm(int layer_id, float *input, float *output, int input_size, int
output_size, bool relu);
```

4 正确性测试及性能优化分析

4.1 神经网络推理测试

在本次实验中，我们一共进行了以下3种神经网络推理正确性测试：

- CPU非并行运算（迭代次数：1）

```
group14@acalab-W760-G30:~/code/cuda_vgg16$ ./cpu
Begin reading ...
Complete reading.
Inference input[0]
Inference input[1]
Inference input[2]
Inference input[3]
Inference input[4]
Inference input[5]
Inference input[6]
Inference input[7]
Inference input[8]
Inference input[9]
Average Time is: 194512.843750 ms
```

- CUDA初步并行运算（迭代次数：5）

```
group14@acalab-W760-G30:~/code/cuda_vgg16$ ./gpu_back
Begin reading ...
Complete reading.
Inference input[0]
Inference input[1]
Inference input[2]
Inference input[3]
Inference input[4]
Inference input[5]
Inference input[6]
Inference input[7]
Inference input[8]
Inference input[9]
Average Time is: 20803.308594 ms
```

- CUDA共享内存优化并行运算（迭代次数：500）

```
group14@acalab-W760-G30:~/code/cuda_vgg16$ ./gpu
Begin reading ...
Complete reading.
Inference input[0]
Inference input[1]
Inference input[2]
Inference input[3]
Inference input[4]
Inference input[5]
Inference input[6]
Inference input[7]
Inference input[8]
Inference input[9]
Average Time is: 1197.866699 ms
group14@acalab-W760-G30:~/code/cuda_vgg16$
```

由于CPU和CUDA初步并行运算速度过慢，此处只进行迭代1次和5次的测试。完成最终优化后，我们分别进行了迭代1次、5次和500次的测试，此处只放出迭代500次的截图。显然，算法的正确性得到了保证。

4.2 CUDA加速效果分析

	iteration: 1	iteration: 5	iteration: 500
CPU	194512.84 ms	N/A	N/A
CUDA without shared memory	22363.98 ms	20803.31 ms	N/A
CUDA with shared memory	1239.10 ms	1194.58 ms	1197.87 ms

由上述表格可以看出，即使是最基础的并行，也可以将推理速度加快接近10倍；利用共享内存规避在CPU中执行循环累加的操作后，推理速度加快了约160倍。优化手段在网络功能实现部分已做详细描述，此处不予赘述。

根据助教发布的`generateTestbench.py`，我们在个人主机上部署python环境，测试了在CPU环境下基于`onnxtime`的神经网络推理效率，如下所示：

```
PS C:\Users\林宇欣\Desktop\计科三\分布式\作业\大作业 vgg16>
python .\generateTestbench.py
Iteration: 500
Avg time cost is:  0.2286069483757019 s
PS C:\Users\林宇欣\Desktop\计科三\分布式\作业\大作业 vgg16>
```

可以看到，其完成一张测试图片的平均推理时间为228.61 ms，速度是我们最终的优化结果的5倍左右。较为遗憾的是，我们在GPU上进行并行运算的速度并没有超过`onnxruntime`在CPU上运算的速度，这可能是出于以下3种原因：

1. Grid的分区和Block的分配都是一维的，设计较为粗糙；而尤其是在卷积前期输入通道数较少时，Block中的Thread数目太少，没能充分发挥并行运算的优越性。
2. Thread完成并行运算后只在第0个Thread中进行累加操作；出于同步的要求，其他Thread都需要等待其完成累加才能继续运行。
3. padding、relu和bias等操作都在在CPU上执行的，没有进行并行优化，一定程度上拖慢了神经网络推理的速度。

5 总结与反思

本次大作业的难点在于理解VGG16网络进行卷积计算的过程以及并行化计算的设计。因为一些理解差错以及设计缺陷，我们将代码推翻重写了数次，最终才完成了一个让我们初步满意的版本。为了调整代码，我们多次寻求了助教们的帮助，非常感谢助教们的耐心。

在本次大作业的完成过程中，我们充分体会到了合作与沟通的重要性，对于CUDA编程和神经网络推理的理解程度也上了一层台阶。相信在日后的学习和工作中，这段经历会成为一份宝贵的财富。