异构计算项目报告

1. 题目：LeNet-5卷积神经网络计算优化

2. 目的和要求

主要内容：

基于已有的LeNet-5 CNN手写识别机器学习程序，分析程序的计算热点（kernal）；基于CPU+GPU的异构计算平台，采用OpenCL编程框架，编写异构执行代码，对程序的执行性能进行优化，在保持识别精度的前提下，最大限度地减少程序执行时间。

基本要求：

1. 下载运行LeNet-5卷积神经网络程序，统计训练时间、识别时间和识别精度关系，分析程序的计算热点；
2. 在保持识别精度条件下，基于通过CPU计算平台，采用循环展开、SIMD指令优化、多线程执行等方法对程序执行进行优化；
3. 在保持识别精度条件下，基于CPU+GPU的异构计算平台，编写OpenCL异构执行代码， 对程序的执行性能进行优化；
4. 基于3）编写的OpenCL异构执行代码，采用存储器优化、工作项并行优化等方法对程序执行时间进行进一步化；
5. 至少对LeNet-5卷积神经网络的前向传导网络进行优化，包括：卷积层和池外层计算；
6. 撰写报告，对2）、3）、4）中程序优化方法进行详细说明，对比分析不同方法获得的优化效果和原因。

3. 项目环境

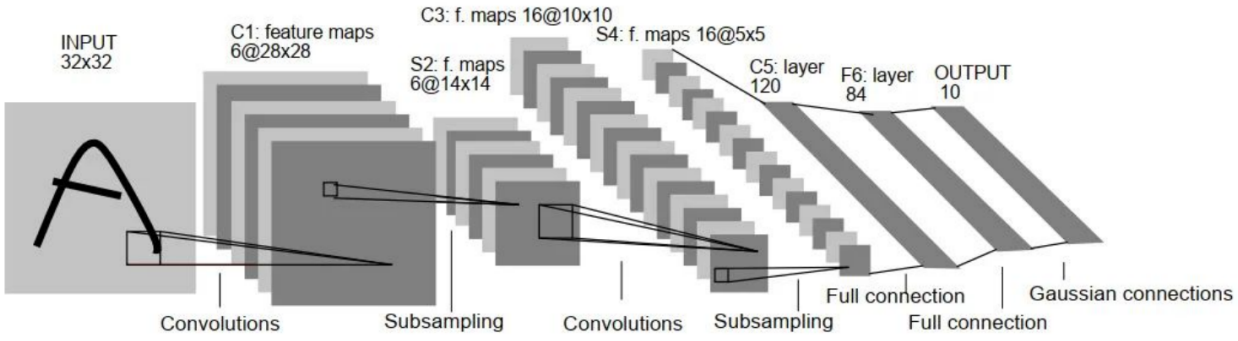
OS: Ubuntu 20.04
GPU: NVIDIA GeForce RTX 3060
OpenCL C 1.2

4. 项目实施步骤和过程

CNN基本原理

前向传播

前向传播基本原理如图所示。输入一幅32*32大小的图像，通过一层层卷积和池化进行特征的提取，最终在output中得到对应的预测。需要注意的是，与图片有所差异，在本项目中，没有F6层。



反向传播

$$\delta^{(k)} = \frac{\partial L(y,\hat{y})}{\partial z^{(k)}}$$

$$= \frac{\partial n^{(k)}}{\partial z^{(k)}} * \frac{\partial z^{(k+1)}}{\partial n^{(k)}} * \frac{\partial L(y,\hat{y})}{\partial z^{(k+1)}}$$

$$= \frac{\partial n^{(k)}}{\partial z^{(k)}} * \frac{\partial z^{(k+1)}}{\partial n^{(k)}} * \delta^{(k+1)}$$

$$= f_k{}'\left(z^{(k)}\right) * \left(\left(W^{(k+1)}\right)^T * \delta^{(k+1)}\right)$$

$$\frac{\partial L(y,\hat{y})}{\partial w^{(k)}} = \frac{\partial L(y,\hat{y})}{\partial z^{(k)}} * \frac{\partial z^{(k)}}{\partial w^{(k)}} = \delta^{(k)} * \left(n^{(k-1)}\right)^T$$

$$\frac{\partial L(y,\hat{y})}{\partial b^{(k)}} = \frac{\partial L(y,\hat{y})}{\partial z^{(k)}} * \frac{\partial z^{(k)}}{\partial b^{(k)}} = \delta^{(k)}$$

如图所示，反向传播采用凸优化的思想。首先用设计好的损失函数对正向所包含的参数求偏导，如此得到的就是对应参数的梯度，也就是损失函数在该参数维度上下降最快的方向。

$$\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$$
$$\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$$

然后根据偏导值对原参数值进行更新（原值=学习率*偏导值），若学习率设置合理，有理由相信，更新后的值比原值更接近极值点。

CPU程序的实现和优化

由于CNN的每一层大同小异，不是池化就是卷积，所以以下以C3层为例，足够说明优化原理。

C3层的核心代码如下：

```
bool CNN::Forward_C3()
{
    for (int channel = 0; channel < num_map_C3_CNN; channel++) {
        for (int y = 0; y < height_image_C3_CNN; y++) {
            for (int x = 0; x < width_image_C3_CNN; x++) {
                int index = (channel*height_image_C3_CNN*width_image_C3_CNN) + y*width_image_C3_CNN + x;
                neuron_C3[index] = 0.0;
                //卷积运算
                for (int inc = 0; inc < num_map_S2_CNN; inc++) {
                    if (!tbl[inc][channel]) continue;
                    int addr1 = get_index(0, 0, num_map_S2_CNN * channel + inc, width_kernel_conv_CNN, he
                    int addr2 = get_index(0, 0, inc, width_image_S2_CNN, height_image_S2_CNN, num_map_S2_
                    const float* pw = &weight_C3[0] + addr1;   //卷积核
                    const float* pi = &neuron_S2[0] + addr2;   //输入图像
                    float sum = 0.0;
                    const float* ppw = pw;
                    const float* ppi = pi + y * width_image_S2_CNN + x;
                    for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                        for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                            sum += *ppw++ * ppi[wy * width_image_S2_CNN + wx];
                        }
                    }
                    neuron_C3[index] += sum;
                }
                neuron_C3[index] += bias_C3[channel];     //加偏置
                neuron_C3[index] = activation_function_tanh(neuron_C3[index]);  //激励函数
            }
        }
    }
    return true;
}
```

可以看到，C3层主要是对图片做卷积运算得到新的数组。

AVX指令集优化

在卷积操作中，对每一个像素点所做的操作都是相似的，这就启发我们，可以一次对多个像素做相同操作，于是就有了用AVX指令集进行优化的思路。

avx指令集是一种向量指令集，运用avx指令集，可以实现一次处理多个数据，而且对多个数据的处理效率比每个数据单独处理加起来会更有效率。这里由于数据个数不能完全整除向量指令大小，所以做了拆分处理。

```
bool CNN::Forward_C3()
{
    for (int channel = 0; channel < num_map_C3_CNN; channel++) {
        for (int y = 0; y < height_image_C3_CNN; y++) {
            for (int x = 0; x < width_image_C3_CNN; x++) {
                int index = (channel*height_image_C3_CNN*width_image_C3_CNN) + y*width_image_C3_CNN + x;
                neuron_C3[index] = 0.0;
                //卷积运算
                for (int inc = 0; inc < num_map_S2_CNN; inc++) {
                    if (!tbl[inc][channel]) continue;
                    int addr1 = get_index(0, 0, num_map_S2_CNN * channel + inc, width_kernel_conv_CNN, he
                    int addr2 = get_index(0, 0, inc, width_image_S2_CNN, height_image_S2_CNN, num_map_S2_
                    const float* pw = &weight_C3[0] + addr1;    //卷积核
                    const float* pi = &neuron_S2[0] + addr2;    //输入图像
                    float sum = 0.0;
                    const float* ppw = pw;
                    const float* ppi = pi + y * width_image_S2_CNN + x;
                    for (int wy = 0; wy < height_kernel_conv_CNN; wy++) {
                        for (int wx = 0; wx < width_kernel_conv_CNN; wx++) {
                            sum += *ppw++ * ppi[wy * width_image_S2_CNN + wx];
                        }
                    }
                    neuron_C3[index] += sum;
                }
                neuron_C3[index] += bias_C3[channel];     //加偏置
                neuron_C3[index] = activation_function_tanh(neuron_C3[index]);  //激励函数
            }
        }
    }
    return true;
}
```

多线程优化

在卷积操作中，新矩阵不同位置的计算不存在数据相关。这就启发了我们，可以采用多线程对CPU程序进行优化。OpenMP用于共享存储节点并行编程，通过简单的编译指令就可以让编译器生成多线程的代码。使用OpenMP优化后代码如下：

```
#pragma omp parallel for num_threads (NUM_THREADS)
for (int channel = 0; channel < num_map_C3_CNN; channe
    for (int y = 0; y < height_image_C3_CNN; y++) {
        int x = 0;
```

GPU的实现与优化

基础OpenCL实现

由于卷积/池化操作对每个像素点不相关，那么很容易可以想到，可以把它们放到GPU上，让每个计算单元计算一个节点。基础OpenCL核心代码如下，每个节点执行一个卷积操作：

```
for (inc=0; inc<in_num; inc++) {
    int addr1 = (in_num * channel + inc) * kernel_height * kernel_width;
    int addr2 = (inc)*in_width*in_height;
    __global const float* pw = weight + addr1;   //卷积核
    __global const float* pi = in + input_index + addr2;       //输入图像
    sum = 0.0;
    __global const float* ppw = pw;
    __global const float* ppi = pi + y * in_width + x;
    for(wy = 0; wy < kernel_height; wy++)  {
        for(wx = 0; wx < kernel_width; wx++) {
            sum += *ppw++ * ppi[wy * in_width + wx];
        }
    }
    out[index] += sum;
}
out[index] += bias[channel];
out[index] = tanh((float)(out[index]));
```

OpenCL常量优化

可以看到，对于每一个计算单元，卷积核都是不会变的，那么就可以把它声明为常量放在常量存储区，在常量存储区取拥有比在全局存储区取数更快的速度。

```
for (inc=0; inc<in_num; inc++) {
    int addr1 = (in_num * channel + inc) * kernel_height * kernel_width;
    int addr2 = (inc)*in_width*in_height;
    __constant const float* pw = weight + addr1;    //卷积核
    __global const float* pi = in + input_index + addr2;        //输入图像
    sum = 0.0;
    __constant const float* ppw = pw;
    __global const float* ppi = pi + y * in_width + x;
    for(wy = 0; wy < kernel_height; wy++)  {
        for(wx = 0; wx < kernel_width; wx++) {
            sum += *ppw++ * ppi[wy * in_width + wx];
        }
    }
    out[index] += sum;
}
out[index] += bias[channel];
out[index] = tanh((float)(out[index]));
//out[index] = out_val;
```

OpenCL局部存储优化

对于每一个局部工作组的每个工作项，有大量重复使用的数据，对于这些数据，可以先取到局部工作组共享的局部存储（local memory）中来，让工作项在局部存储中存取数据，拥有比全局存储更快的速度。

```
constant int num_map_input_CNN = 1;
float local pixel[num_map_input_CNN][BS1+filtersize-1][BS1+filtersize-1];


for (int i=0; i<in_num; i++)
{
    int addr2 = i*in_width*in_height;
    pixel[i][tidy][tidx]=in[addr2 + y*in_width + x];
    if(tidx < filtersize -1 ) pixel[i][tidy][tidx+BS1] = in[addr2+y*in_width+x+BS1];
   if(tidy < filtersize -1 ) pixel[i][tidy+BS1][tidx] = in[addr2+(y+BS1)*in_width+x];
    if(tidx < filtersize -1 &&tidy < filtersize-1)
                pixel[i][tidy+BS1][tidx+BS1] = in[addr2+(y+BS1)*in_width+x+BS1];
}
barrier(CLK_LOCAL_MEM_FENCE);
```

后向传播实现

如果只是简单的做前向传播，实际上并不能有比较好的效果。因为每次迭代都要对前向传播的参数进行更新，同时前向传播的参数也需要传给后向传播。这就意味着每次迭代都要

把数据在GPU和CPU之间来回搬移，浪费大量的时间。事实上，迭代一个epoch的时间达到了两个小时之久。解决方案就是将数据一次性送到gpu上，等gpu将整个过程跑完，再把数据取回。

后向传播在实现时，由于有对某个内存单元进行+=的操作，可能由于多个计算单元的同时读写造成数据一致性出现问题。但是对于浮点变量，OpenCL并没有提供原子操作，而栅栏之类的同步操作也不能起作用。我的解决方案是，让每个计算单元处理不同地址的数据，这样就可以保证不同计算单元之间不相关，解决数据一致性的问题。

如图所示：

```
int outc = get_global_id(0);
int block = width_image_C3_CNN * height_image_C3_CNN * outc; //C3
int y = get_global_id(1);
int x = get_global_id(2);
float scale_factor = 1.0 /(height_kernel_pooling_CNN * width_kernel_pooling_CNN);
int rows = y * width_kernel_pooling_CNN;
int cols = x * height_kernel_pooling_CNN;
int index = (outc*height_image_S4_CNN*width_image_S4_CNN) + y*width_image_S4_CNN + x;
for (int m = 0; m < height_kernel_pooling_CNN; m++) {
    for (int n = 0; n < width_kernel_pooling_CNN; n++) {
        int addr1 = outc;  // 权重
        int addr2 = block + (rows + m) * width_image_C3_CNN + cols + n; //C3 神经元 k
        // int addr3 = outc;
        float temp = 1.0 - neuron_C3[addr2] * neuron_C3[addr2];
        delta_neuron_C3[addr2] = delta_neuron_S4[index] * weight_S4[addr1]
                                 * temp * scale_factor;
        // delta_weight_S4[addr1] += delta_neuron_S4[index] * neuron_C3[addr2] * scale_factor;
    }
}
```

只需要对示例代码进行简单修改，就可以使在每个计算单元中，处理的地址单元都是唯一且与其它单元不同。

5. 实验数据记录

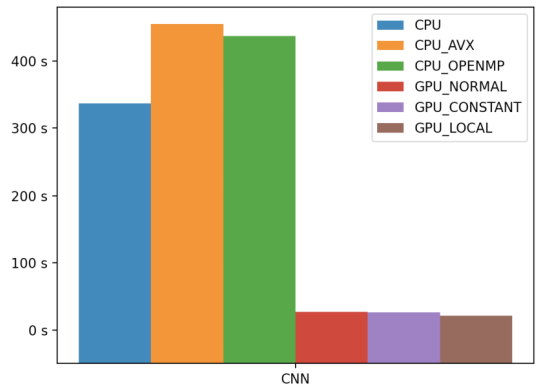
其中CPU程序优化级别为默认，应该是o2。

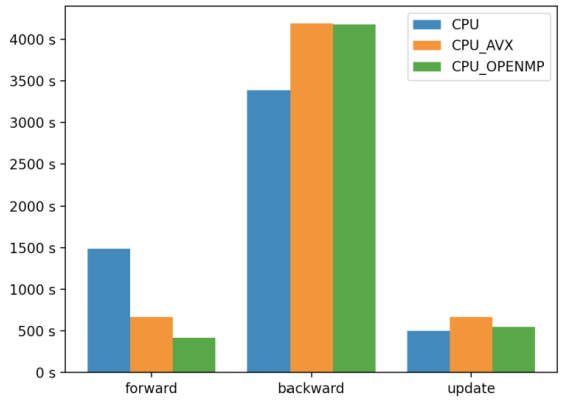
	forward(ms)	backward(ms)	update(ms)	count(s)
CPU	1490	3385	500	334
CPU_AVX	667	4188	669	467
CPU_OPENMP	417	4179	552	443
GPU				27

GPU_CONSTANT				26
GPU_LOCAL				21

6. 分析与结论

对执行时间用柱状图进行量化：





由于课程服务器无法使用，所以无法测试开o3优化之后的数据了。但是可以看到，对于CPU程序，在采用avx优化后，前向传播速度有了显著提升。在此之上采用OpenMP之后，前向传播速度更加快了。但于此同时，后向传播和数据更新时间也有一定延长。如果开了编译器优化，应该可以消除掉这些延长时间。

对于GPU程序，常量优化的效果并不明显，程序最终执行时间在小范围内波动，可以认为优化效果不好。而local优化则取得了明显的效果。

7. 感受及建议

整个异构计算课程下来，我对于GPU体系结构以及如何编写GPU程序有了新的理解，也惊异于程序优化后能取得的巨大性能提升。在此之前，我写程序都是以实现为最终目标，上完这个课之后，我发现写程序更加应该精益求精。

关于课程，感觉不管是平时作业还是最后的大作业，都要做很多重复的init工作，如果课程能提供代码框架，而让学生去修改核心代码，我觉得会更有效率。