**基于SIMD的卷积神经网络的并行化研究**

姓名：李林宇 学号：2120210452

**1. 问题分析**

我的期末研究报告的题目是对卷积操作进行相关的并行化研究。在卷积神经网络中，卷积层的功能是对输入数据进行特征提取，其内部包含多个卷积核，组成卷积核的每个元素都对应一个权重系数和一个偏差量，类似于一个前馈神经网络的神经元。卷积层内每个神经元都与前一层中位置接近的区域的多个神经元相连，区域的大小取决于卷积核的大小，在文献中被称为“感受野”，其含义可类比视觉皮层细胞的感受野。卷积核在工作时，会有规律地扫过输入特征，在感受野内对输入特征做矩阵元素乘法求和并叠加偏差量。在卷积神经网络中，卷积层负责提取局部空间特征，并负责完成大部分计算繁重的工作。因此，想要提升卷积神经网络的性能，一个很重要的方法是提升卷积操作的性能。在这次实验中，我做的主要工作是利用SIMD来对卷积操作进行并行化。

本次实验报告对应的项目Git地址为：[ParallelHomework/SIMD\_Parallel at main · youxiho1/ParallelHomework (github.com)](https://github.com/youxiho1/ParallelHomework/tree/main/SIMD_Parallel)

**2. SIMD算法设计以及代码实现**

**2.1 卷积操作的实现**

研究过程中需要针对卷积操作不断做优化，因此在研究的最开始阶段，本文作者将首先用C++语言自行实现一个卷积操作，以此作为后续所有实验的基础。一个简单的串行实现方式如下所示：

void conv() {

//Calculate

outputHeight = inputHeight - kernelHeight + 1;

outputWidth = inputWidth - kernelWidth + 1;

for(int i = 0; i < outputHeight; i++) {

for(int j = 0; j < outputWidth; j++) {

float sum = 0;

for(int c = 0; c < channels; c++) {

for(int k = 0; k < kernelHeight; k++) {

for(int t = 0; t < kernelWidth; t++) {

sum += input[c][i+k][j+t] \* kernel[c][k][t];

}

}

}

result[i][j] = sum;

}

}

}

**2.2 卷积操作的SIMD优化**

Intel CPU 中有一类被称为 SIMD 的指令集，包括 SSE、SSE2、AVX，可一次批量执行更多的计算，从而有效提升CNN推理的计算性能。这一次编程作业，我以SIMD作为主题，尝试优化卷积神经网络。在2.1节中所述的卷积操作中，我们的核心运算是输入图像和卷积核的点乘操作，该操作是可以向量化的。我们可以利用SIMD，一次送入输入图像中的四个像素点和卷积核中的四个像素点，对他们做点乘求和操作。

**2.3 卷积操作的SIMD优化的代码实现—x86架构**

我们用SSE来做x86架构下的SIMD优化，核心代码（采用SSE进行优化的部分已标红）如下所示：

void conv\_simd() {

\_\_m128 t1, t2, s;

//Calculate

outputHeight = inputHeight - kernelHeight + 1;

outputWidth = inputWidth - kernelWidth + 1;

for(int i = 0; i < outputHeight; i++) {

for(int j = 0; j < outputWidth; j++) {

float temp = 0;

s = \_mm\_setzero\_ps();

for(int c = 0; c < channels; c++) {

for(int k = 0; k < kernelHeight; k++) {

for(int t = kernelWidth - 4; t >= 0; t -= 4) {

t1 = \_mm\_loadu\_ps(input[c][i+k]+j+t);

t2 = \_mm\_loadu\_ps(kernel[c][k]+t);

t1 = \_mm\_mul\_ps(t1, t2);

s = \_mm\_add\_ps(s, t1);

}

for(int t = (kernelWidth % 4) - 1; t >= 0; t--) {

temp += input[c][i+k][j+t] \* kernel[c][k][t];

}

}

}

s = \_mm\_hadd\_ps(s, s);

s = \_mm\_hadd\_ps(s, s);

\_mm\_store\_ss(result[i]+j, s);

result[i][j] += temp;

}

}

}

**2.4 卷积操作的SIMD优化的代码实现—arm架构**

我们用Neon来做arm架构下的SIMD优化，核心代码（采用Neon进行优化的部分已标红）如下所示：

void conv\_simd() {

float32x4\_t t1, t2, s;

float32x2\_t s1, s2;

//Calculate

outputHeight = inputHeight - kernelHeight + 1;

outputWidth = inputWidth - kernelWidth + 1;

for(int i = 0; i < outputHeight; i++) {

for(int j = 0; j < outputWidth; j++) {

float temp = 0;

s = vdupq\_n\_f32(0.0);

for(int c = 0; c < channels; c++) {

for(int k = 0; k < kernelHeight; k++) {

for(int t = kernelWidth - 4; t >= 0; t -= 4) {

t1 = vld1q\_f32(input[c][i+k]+j+t);

t2 = vld1q\_f32(kernel[c][k]+t);

t1 = vmulq\_f32(t1, t2);

s = vaddq\_f32(s, t1);

}

for(int t = (kernelWidth % 4) - 1; t >= 0; t--) {

temp += input[c][i+k][j+t] \* kernel[c][k][t];

}

}

}

s1 = vget\_low\_f32(s);

s2 = vget\_high\_f32(s);

s1 = vpadd\_f32(s1, s2);

s1 = vpadd\_f32(s1, s1);

vst1\_lane\_f32(result[i]+j, s1, 0);

result[i][j] += temp;

}

}

}

**3. 实验和结果分析**

**3.1 实验环境**

本次实验中涉及到的x86环境的CPU：Intel Core i7-10700K

本次实验中涉及到的Arm环境：鲲鹏服务器

由于条件并不严格相同，因此虽然在本次实验中，我分别基于不同架构做了实验，但是我们仍然不将不同架构的数据做对比分析。（因为我们无法说明性能差异是CPU本身性能所带来的，还是由于架构所带来的）但从后续的实验数据分析的过程中，我们仍然可以很清晰地看到不同架构对于加速比的影响。

**3.2 在x86架构上探究SSE并行优化的加速比**

我们首先在x86架构上来探究SSE优化的效果。在这样的实验中，我们首先将卷积核的大小固定为4\*4，输入图片均选择灰度图像（通道数固定为1），然后调整输入图片的大小，比对使用SSE和不使用SSE的情况下的运行时间，每组实验重复100次，然后计算总时间，以此来减小波动带来的误差。实验结果如表1所示。从表1中我们可以看出，在卷积核的大小为4\*4的时候，加速比基本上在1.1左右，加速效果并不是特别明显。造成这种现象的原因主要是卷积核大小较小，卷积操作数量并不够多。因此，我们控制其他条件不变，将卷积核大小扩大为64\*64，再来进行一组实验，实验结果如表2所示。从表2中我们可以看到，加速效果相对来说更为显著，加速比在2左右。可以想见，随着问题规模的逐渐增大，随着卷积操作的次数逐渐增多，SIMD的优化效果会非常显著。

表1 x86架构上的测试结果，卷积核大小为4\*4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **输入图片大小** | **重复次数** | **串行执行总时间** | **并行执行总时间** | **加速比** |
| 10\*10 | 100 | 0.221ms | 0.200ms | 1.105 |
| 20\*20 | 100 | 1.279ms | 1.179ms | 1.084817642 |
| 30\*30 | 100 | 3.284ms | 2.837ms | 1.157560804 |
| 40\*40 | 100 | 6.226ms | 5.316ms | 1.171181339 |
| 50\*50 | 100 | 9.698ms | 8.546ms | 1.134799906 |
| 60\*60 | 100 | 14.278ms | 12.745ms | 1.120282464 |
| 70\*70 | 100 | 19.769ms | 20.881ms | 0.946745846 |
| 80\*80 | 100 | 26.032ms | 23.076ms | 1.128098457 |
| 90\*90 | 100 | 33.613ms | 35.080ms | 0.9581813 |
| 100\*100 | 100 | 42.061ms | 37.008ms | 1.136538046 |
| 200\*200 | 100 | 170.633ms | 153.180ms | 1.113937851 |
| 300\*300 | 100 | 391.179ms | 350.447ms | 1.116228702 |
| 400\*400 | 100 | 711.585ms | 620.097ms | 1.147538208 |
| 500\*500 | 100 | 1.111s | 974.427ms | 1.140157241 |
| 600\*600 | 100 | 1.604s | 1.411s | 1.136782424 |
| 700\*700 | 100 | 2.169s | 1.920s | 1.1296875 |
| 800\*800 | 100 | 3.263s | 2.521s | 1.294327648 |
| 900\*900 | 100 | 4.149s | 3.191s | 1.300219367 |
| 1000\*1000 | 100 | 5.248s | 3.953s | 1.327599292 |
| 5000\*5000 | 100 | 128.947s | 99.226s | 1.299528349 |
| 10000\*10000 | 100 | 546.241s | 396.311s | 1.378314001 |

表2 x86架构上的测试结果，卷积核大小为64\*64

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **输入图片大小** | **重复次数** | **串行执行总时间** | **并行执行总时间** | **加速比** |
| 100\*100 | 10 | 150.901ms | 78.663ms | 1.918322464 |
| 200\*200 | 10 | 2.052s | 1.037s | 1.978784957 |
| 300\*300 | 10 | 6.139s | 3.114s | 1.971419396 |
| 400\*400 | 10 | 12.376s | 6.300s | 1.964444444 |
| 500\*500 | 10 | 20.837s | 10.617s | 1.962607139 |
| 600\*600 | 10 | 31.527s | 15.941s | 1.977730381 |
| 700\*700 | 10 | 44.444s | 22.462s | 1.978630576 |
| 800\*800 | 10 | 59.386s | 30.019s | 1.978280422 |
| 900\*900 | 10 | 76.535s | 38.791s | 1.973009203 |
| 1000\*1000 | 10 | 95.920s | 48.648s | 1.971715178 |

**3.3 在Arm架构上探究Neon并行优化的加速比**

接下来，我们借助鲲鹏服务器，来研究在Arm架构上使用Neon进行并行优化的效果。在这样的实验中，我们将卷积核的大小固定为64\*64，输入图片同样选择灰度图像（通道数固定为1），调整输入图片的大小，比对使用Neon优化和不使用Neon优化的运行时间。实验结果如表3所示。从表3中我们可以看到，arm架构下采用Neon进行并行优化的加速比接近4.

表3 arm架构上的测试结果，卷积核大小为64\*64

|  |  |  |  |
| --- | --- | --- | --- |
| **输入图片大小** | **串行执行时间** | **并行执行时间** | **加速比** |
| 100\*100 | 8ms | 2ms | 4 |
| 200\*200 | 121ms | 30ms | 4.033333 |
| 300\*300 | 354ms | 92ms | 3.847826 |
| 400\*400 | 716ms | 187ms | 3.828877 |
| 500\*500 | 1207ms | 312ms | 3.86859 |
| 600\*600 | 1817ms | 472ms | 3.849576 |
| 700\*700 | 2560ms | 663ms | 3.861237 |
| 800\*800 | 3425ms | 893ms | 3.835386 |
| 900\*900 | 4412ms | 1146ms | 3.849913 |
| 1000\*1000 | 5533ms | 1437ms | 3.850383 |
| 5000\*5000 | 154.248s | 40.119s | 3.844762 |
| 10000\*10000 | 626.160s | 162.419s | 3.855214 |

**4. 总结**

在本次实验中，我们分别针对x86架构和Arm架构采用了SSE和Neon来对其上运行的卷积操作进行并行优化，加速效果非常明显。此外，从实验数据中我们可以看到在卷积核大小为64\*64的灰度图像的假设下，基于SSE做优化的并行加速比大致为2左右，而基于Neon做优化的并行加速比大致为4左右，可以很清晰地看出不同架构、指令集对于并行优化效果的影响。本次实验也为期末报告中的卷积操作的并行优化的相关综合性研究做好了铺垫。