基于 SIMD 和 pThread 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 PThread 和 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 • youxihol/ParallelHomework (github.com)

2. PThread & 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，在上一次实验中，作者已经用C++语言自行实现了一个卷积操作，以此作为该实验的基础。卷积操作的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 PThread 优化

基于多线程编程技术，我们可以设计卷积神经网络在多处理器上的并行计

算
方案。在每一个线程内部，我们使用 SIMD 的指令集，包括 SSE、SSE2、AVX，可一次批量执行更多的计算，从而再次提升计算性能。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 Pthread 将待处理的数据分块，然后每一个线程内采用 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 pThread 优化的代码实现—x86 架构

我们用 pThread 来做并行优化，核心代码（采用 pThread 进行优化的部分已标红）如下所示：

```
void conv_pthread() {
    //Calculate
    outputHeight = inputHeight - kernelHeight + 1;
    outputWidth = inputWidth - kernelWidth + 1;
    QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
    seg = outputHeight / THREAD_NUM;
    mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_t thread[THREAD_NUM];
    threadParam_t threadParm[THREAD_NUM];
    QueryPerformanceCounter((LARGE_INTEGER *)&head);
    for (int i = 0; i < THREAD_NUM; i++) {
        threadParm[i].threadId = i;
        pthread_create(&thread[i], nullptr, pthread_simd_calc_conv, (void
*)&threadParm[i]);
    }
    for (int i = 0; i < THREAD_NUM; i++) {
        pthread_join(thread[i], nullptr);
    }
    pthread_mutex_destroy(&mutex);
    putchar('\n');
    putchar('\n');
}
void *pthread_calc_conv(void *parm) {
    threadParam_t *p = (threadParam_t *) parm;
    int r = p->threadId;
    long long tail;
    for(int i = r * seg; i < (r + 1) * seg; 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++) {        //kernelWidth
                    for(int t = 0; t < kernelWidth; t++) {
                        sum += input[c][i+k][j+t] * kernel[c][k][t];
                    }
```

```
            }
        }
        result[i][j] = sum;
    }
  }
  pthread_mutex_lock(&mutex);
  QueryPerformanceCounter((LARGE_INTEGER *)&tail);
  printf("Thread %d: %lfms.\n", r, (tail - head) * 1000.0 / freq);
  pthread_mutex_unlock(&mutex);
  pthread_exit(nullptr);
}
```

2.4 卷积操作的 pThread & SIMD 优化的代码实现—x86 架构

在 2.3 的基础上，我们在每个线程内部加入 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 *pthread_simd_calc_conv(void *parm) {
  __m128 t1, t2, s;
  threadParam_t *p = (threadParam_t *) parm;
  int r = p->threadId;
  long long tail;
  for(int i = r * seg; i < (r + 1) * seg; 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++) {        //kernelWidth
          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;
    }
  }
  pthread_mutex_lock(&mutex);
```

```
    QueryPerformanceCounter((LARGE_INTEGER *)&tail);
    printf("Thread %d: %lfms.\n", r, (tail - head) * 1000.0 / freq);
    pthread_mutex_unlock(&mutex);
    pthread_exit(nullptr);
}
```

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 探究 SSE 和 PThread 并行优化的加速比

由于 PThread 在 x86 和 Arm 架构上实现方式一致，因此本实验中我们主要在 x86 架构上来探究 SSE 和 PThread 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 64*64，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，比对不使用 SSE、仅使用 SSE、仅使用 Pthread 和使用 PThread+SIMD 的情况下的运行时间。实验结果如表 1 所示。

从表 1 中我们可以看出，在数据规模较小时，采用 pThread 进行计算会明显慢于不使用 pThread 的版本，这主要是由于线程创建开销相较于较小数据规模的计算过大。而随着问题规模的逐渐增大，线程创建的开销所占的运行时间的比例就不断缩小，最终可以接近忽略不计。从表中我们可以看出在卷积核的大小为64*64的时候，仅使用SIMD和仅使用pThread的加速比基本上在4左右，而同时使用 SIMD 和 pThread 的加速比能够接近 16，加速效果非常显著。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，并行优化会变得越发重要。

表 1 测试结果，卷积核大小为 64*64

输入图片大小	Original	SIMD	加速比	pThread	加速比	pThread&SIMD	加速比
70*70	0.243ms	0.064ms	3.80	4.438ms	0.05	3.764ms	0.06
80*80	1.182ms	0.311ms	3.80	4.519ms	0.26	3.054ms	0.39
90*90	2.937ms	0.745ms	3.94	4.480ms	0.66	4.453ms	0.66
100*100	5.497ms	1.390ms	3.95	4.606ms	1.19	4.188ms	1.31
200*200	75.550ms	18.938ms	3.99	23.438ms	3.22	8.805ms	8.58
300*300	225.493ms	55.914ms	4.03	60.309ms	3.74	17.683ms	12.75
400*400	460.827ms	113.975ms	4.04	118.735ms	3.88	32.723ms	14.08
500*500	767.807ms	191.068ms	4.02	196.090ms	3.92	52.363ms	14.66
600*600	1.159s	288.071ms	4.02	293.610ms	3.95	76.730ms	15.10
700*700	1.631s	405.537ms	4.02	412.472ms	3.95	105.815ms	15.41
800*800	2.202s	541.658ms	4.07	555.024ms	3.97	141.439ms	15.57
900*900	2.818s	701.767ms	4.02	706.361ms	3.99	182.736ms	15.42
1000*1000	3.548s	875.473ms	4.05	885.645ms	4.01	222.524ms	15.94
5000*5000	98.556s	24.371s	4.04	24.555s	4.01	6.238s	15.80

10000*10000	397.049s	99.185s	4.00	99.399s	3.99	25.395s	15.63

4. 总结

在本次实验中，我们采用了 SIMD 和 PThread 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。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。