

# 简单卷积神经网络的并行加速优化

## 1. 背景介绍

### 1.1. 选题动机

自 1943 年神经元模型提出至今，神经网络已经经历了三个历史阶段。依托于大规模数据集和大量计算资源，深度学习技术如今在众多领域都表现出统治级的地位。我所在的计算机视觉方向便是其一，传统的视觉方法几乎销声匿迹，取而代之的是以卷积神经网络为代表的深度学习方法。

深度学习的兴起伴随着其编程语言的迭代，现阶段有很多的深度学习框架可以被研究者和从业者使用，比如 Tensorflow, Pytorch, Caffe, Keras 等，这些框架提供了很多的接口，使得深度学习的开发难度相对减小，效率大大提高。但当进入到深层次的科研阶段或者工业界的落地阶段时，算法往往需要研究员和从业者进行底层的编码以进行新方法的实现或代码的优化。

本项目基于此理念，复现了一个纯 C++ 实现的简单卷积神经网络，在理解的基础上进行了 CUDA 编程，引入 GPU 并行加速模块，加深了对多卷积神经网络的理解和底层实现。

### 1.2. 网络框架

卷积神经网络是深度学习网络中代表性的网络结构之一，其将深度神经网络的大量全连接层替换为具有局部性的卷积操作。如图 1-1 所示，本项目实现了具有卷积层、池化层、全连接层等网络层的简单卷积神经网络，其中卷积层、池化层、全连接层间可以相互嵌套，形成深层神经网络。

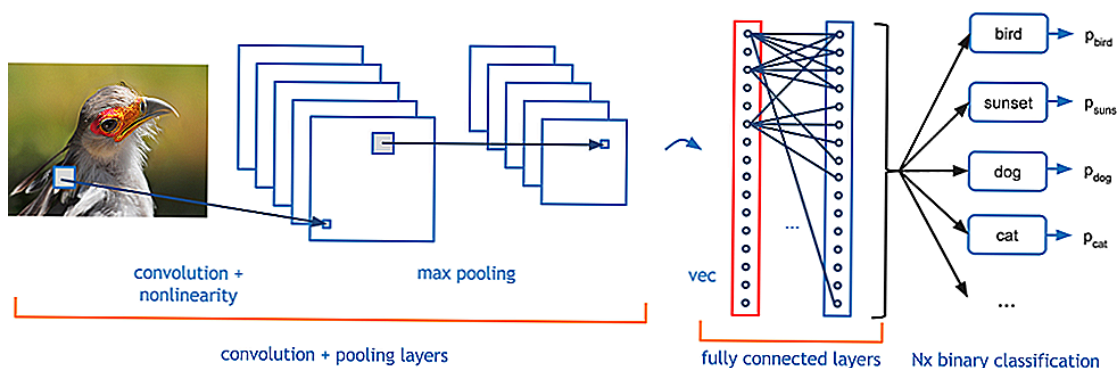


图 1-1 卷积神经网络

## 1.3. 数据集与实验设计

### 1.3.1. 数据集

本项目使用了 Mnist 手写数字数据集[1]，一般以该数据集作为视觉方向网络测试的典范数据集。训练集包含了 60,000 个样本，测试集包含了 10,000 个样本，图片大小为  $28 \times 28 \times 1$ 。

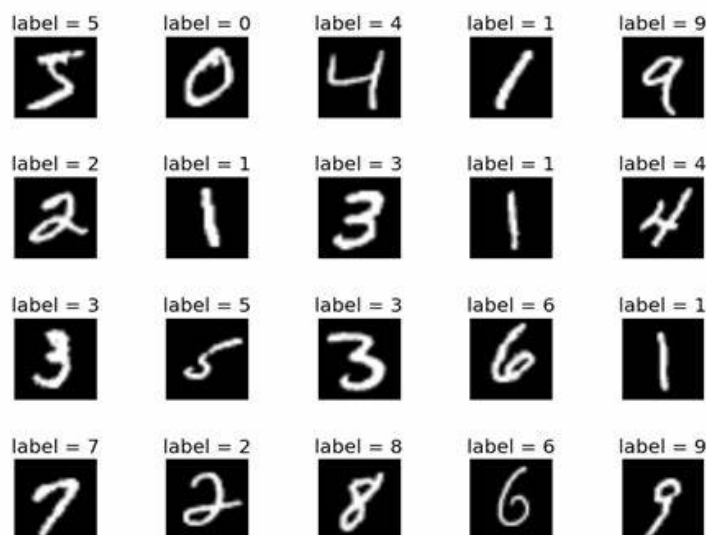


图 1-2 Mnist 手写数字数据集

### 1.3.2. 实验设计

本项目的实验环境为 Visual Studio 2019, CUDA11.2 和 GeForce GTX 1080 Ti。实验环节对单个训练周期的时间进行统计，网络包括 1 个卷积层 ( $5 \times 5 \times 32$ )、1 个池化层 ( $2 \times 2 \times 32$ )、1 个全连接层 (1024) 和 1 个 SoftMax 层 (10)。以 Batch Size=16, 5 次训练迭代 debug 模式下的运行平均值作为单次训练周期的时长，通过对不同模块采用 CPU/GPU 运算，对比不同配置下的代码执行效率，反映优化效率。

后经老师指点，增大训练数据，将 Batch Size 增大到 1024，同时以 50 次训练迭代的时长进行计算。由于增大数据量后程序运行时间较长，对每一单步并行的改进不再测试，对串、并及优化后的代码分别测试，实验结果处保留各次记录。

## 2. 项目框架及算法实现

本项目的串行代码实现参考了 lygyue 的 github 项目 SimpleDeep LearningFramework[2], 项目共包含 stdafx.h, targetver.h, MnistFileManager.h, Sdlf.h, SdlfModel.h, SdlfLayer.h, SdlfCalculator.h, SdlfCalculatorCPU.h, SdlfCalculatorCUDA.h, SdlfFunction.h, SdlfFunctionCUDA.h, KernelCPU.h, KernelCUDA.h 和 Common.h 共计 11 个头文件, 以及 stdafx.cpp, train.cpp, MnistFileManager.cpp, SdlfModel.cpp, SdlfLayer.cpp, SdlfCalculator.cpp, SdlfCalculatorCPU.cpp, SdlfCalculatorCUDA.cpp, SdlfFunction.cpp, SdlfFunctionCUDA.cpp 和 Kernel.cu 共计 10 个源码文件, 大部分头文件与源码文件相对应, 头文件主要负责相关结构体、类和函数等数据结构的定义, 源码文件对头文件中的结构体、类、函数等进行实现。其中带有 CUDA 字样的文件为并行过程的代码实现, 其余文件为串行过程代码, 下面给出主要框架及各文件的简单介绍。

### 2.1. 框架概述

Stdafx.h/cpp, targetver.h 是项目无关的 VS 文件; Common.h 定义了项目通用的内存管理、错误处理等宏和方法; MnistFileManager.h/cpp 定义实现了数据集的处理; Sdlf.h 定义了模型控制相关的基准结构体及其方法, 包括网络模型 (SdlfModel)、网络层 (SdlfLayer)、网络计算 (SdlfCalculator) 和函数计算 (SdlfFunction) 等, 用于后续的继承与实现, 如图 2-1 所示:

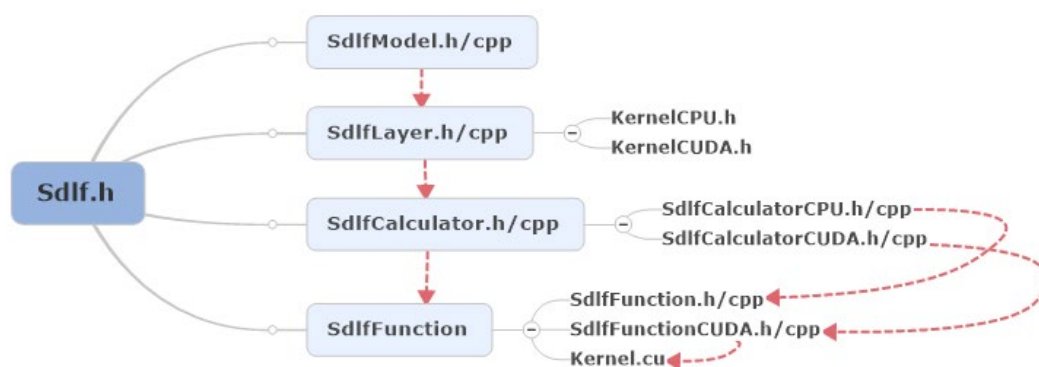


图 2-1 项目代码关系图

- 1) 网络模型在 SdlfModel.h 中被继承, 定义了整个网络模型中可能会用到的模型控制方法, 包括模型的建立, 训练与测试过程的进行, 各层间的调度等, 其方法在 SdlfModel.cpp 中被具体实现;

- 2) 网络层在 `SdlfLayer.h` 中被继承, 层类型包括卷积层、池化层、全连接层、SoftMax 层, 网络层中定义了各层的构建、参数初始化、执行和梯度传递等方法, 在 `SdlfLayer.cpp` 中被具体实现。其中卷积核等参数的结构体和其操作方法被定义实现在 `KernelCPU.h` 和 `KernelCUDA.h` 中。`SdlfLayer` 的相关方法会被 `SdlfModel` 所调用, 用于构建模型;
- 3) 网络计算类在 `SdlfCalculator.h` 中被定义, 一些属性被定义, 包括卷积、池化、全连接和 SoftMax 的计算方法被声明。在 `SdlfCalculatorCPU.h` 和 `SdlfCalculatorCUDA.h` 对 `SdlfCalculator.h` 进行了继承, 分别负责串行计算和并行计算过程, 其方法也在 `SdlfCalculatorCPU.cpp` 和 `SdlfCalculatorCUDA.cpp` 中进行了实现, `SdlfCalculatorCPU.cpp` 仅实现了基础的共用方法。`SdlfCalculator` 相关的方法会被 `SdlfLayer` 所调用, 用于各层的计算;
- 4) 函数计算类在 `SdlfFunction.h` 和 `SdlfFunctionCUDA.h` 中被分别声明, 同时在 `SdlfFunction.cpp` 和 `SdlfFunctionCUDA.cpp` 中被分别实现。`SdlfFunction` 主要集中了项目计算中可能会用到的数学计算的具体实现, 被 `SdlfModel`, `SdlfLayer` 和 `SdlfCalculator` 所调用, 即真正的计算代码是被写在 `SdlfFunction` 中, 其与 `SdlfCalculator` 共同完成了各个计算过程。此外, CUDA 的具体所有真正计算过程被写在 `Kernel.cu` 中。

## 2.2. 串行运算

卷积神经网络的实现除了上述到的数据结构、函数等服务于运算的过程, 最令人关注的是其具体网络层的前向传播和反向传播, 由于时间关系, 在实现时仅对前向传播过程进行了并行加速, 因此本文的串并行算法介绍主要集中在各层的前向计算, 反向传播过程仅有串行实现。

### 2.2.1. 卷积层

在进行卷积运算时, 原项目[2]采用先加载数据并匹配卷积核, 后卷积运算的两步卷积, 而非数据加载和运算同时进行的方式。这种两步方式理解起来较为简单, 但占用内存较大, 本项目在实现时沿用了这种传统。

对于输入数据 `InData[BatchSize*InChannel*ImgHeight*ImgWidth]` (`InData` 表示数据名称, `[]`内表示其组织形式及大小, 单位为 `sizeof(float)`字节), 卷积核的大小为 `W[OutChannel*InChannel*ConvHeight*ConvWidth]`, `B[OutChannel]`, 表示有 `OutChannel` 个长度为 `ConvLen=InChannel*ConvHeight*ConvWidth` 的卷积核。其中卷积核大小选择为 3, 即卷积前后图像大小不变。则输出数据为 `OutData[BatchSize*OutChannel*ImgHeight*ImgWidth]`。

如图 2-1 所示, 对于 `OutData` 中的每一个输出, 都是由 `OutChannel` 个卷积核

与  $\text{InData}$  中的  $\text{ConvLen}$  个数据进行点乘得到。两步卷积是先映射取出每个输出要使用的  $\text{ConvLen}$  长度的  $\text{InData}[\text{ConvLen}]$  数据，放在一个新的连续存储空间  $\text{DataBuffer}[\text{BatchSize} \times \text{ImgHeight} \times \text{ImgWidth} \times \text{ConvLen}]$  中，然后按顺序将  $\text{DataBuffer}$  与  $W$ 、 $B$  进行点积操作，得到  $\text{OutData}$ ；

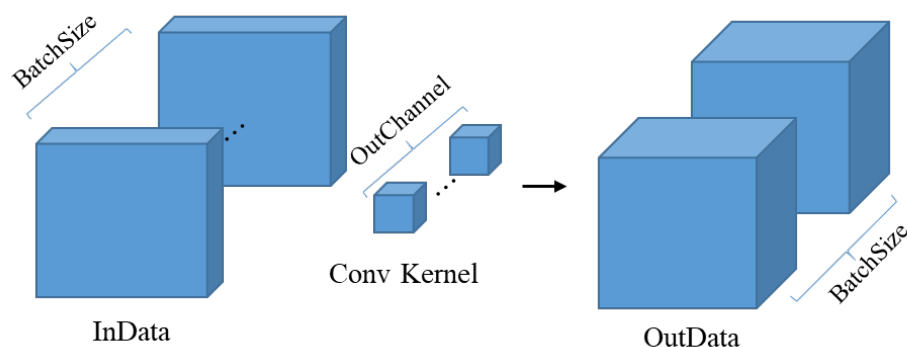


图 2-1 卷积运算

### 2.2.2. 池化层

池化层的计算相对较简单，本项目实现了  $2 \times 2$  的最大池化，需要注意的最大池化的梯度计算。当 4 个元素相等时，最大池化退化为平均池化，采用平均池化的梯度计算方法。

如图 2-2，对于输入数据  $\text{InData}[\text{BatchSize} \times \text{InChannel} \times \text{ImgHeight} \times \text{ImgWidth}]$ ， $2 \times 2$  最大池化得到  $\text{OutData}[\text{BatchSize} \times \text{InChannel} \times \text{ImgHeight}/2 \times \text{ImgWidth}/2]$ ；

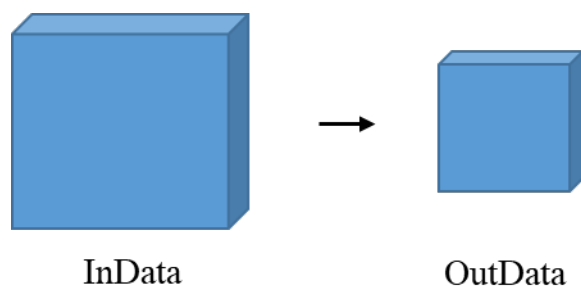


图 2-2 池化运算

### 2.2.3. 全连接层

全连接层直接将上一层的输出拉伸为单个向量，即将  $\text{ImgHeight}$  和  $\text{ImgWidth}$  视为 1， $\text{InChannel} \times \text{ImgHeight} \times \text{ImgWidth}$  视为  $\text{InChannel}$ ，然后使用  $1 \times 1$  的卷积运算进行计算，由于此时  $\text{ImgSize}$  为 1，两步卷积与单步卷积无异。

如图 2-3，对于输入数据  $\text{InData}[\text{BatchSize} \times \text{InChannel}]$  ( $\text{InChannel} = \text{InChannel} \times \text{ImgHeight} \times \text{ImgWidth}$ )，

$\text{ImgHeight} \times \text{ImgWidth}$ ), 输出数据  $\text{OutData}[\text{BatchSize} \times \text{OutChannel}]$ , 参数  $W[\text{InChannel} \times \text{OutChannel}], B[\text{OutChannel}]$ 。理论上标准的全连接层使用矩阵乘法更佳, 但鉴于原项目使用的  $1 \times 1$  卷积, 也保留了这个传统。

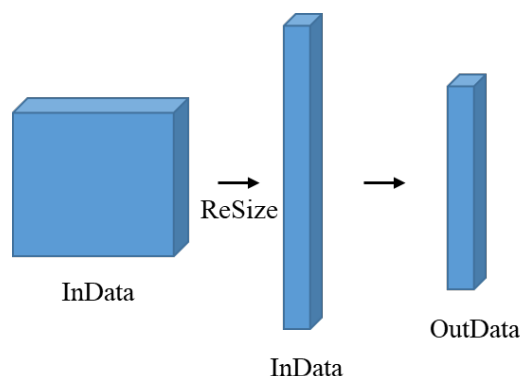


图 2-3 池化运算

#### 2.2.4. SoftMax 层

SoftMax 层本来不应该被独立划分, 但原项目中利用 SoftMax 层进行了 Channel 维度的变换, 将隐藏层的 1024 维度变为 10 类 (对应 0~9 个数字), 同时计算了预测结果的 SoftMax 值, 因此本项目也使用该层命名。实际上, SoftMax 层是具有 SoftMax 函数计算的标准全连接层, 本质上是矩阵乘法。

对于输入数据  $\text{InData}[\text{BatchSize} \times \text{InChannel}]$ , 输出数据为  $\text{OutData}[\text{BatchSize} \times \text{OutChannel}]$ , 参数  $W[\text{InChannel} \times \text{OutChannel}], B[\text{OutChannel}]$ , 此处的运算为矩阵乘法。

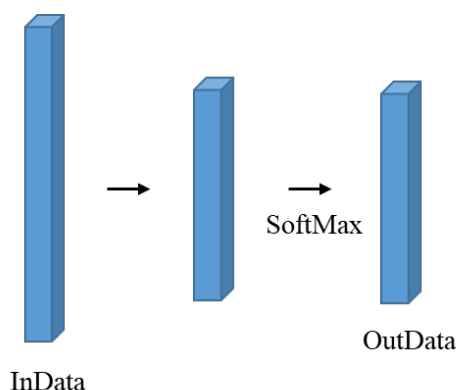


图 2-2 池化运算

### 2.3. 串行实验结果

如 1.3.2 节所提, 我们对实现了串行算法的整个训练过程进行运行速度检测, BatchSize=16 时, 如图附-1 所示, 串行算法的训练时间为 4.69180 s/Iter。

### 3. 并行算法及性能优化

在第2节中已经对涉及并行算法的文件进行了介绍，其中核心的计算单元在 Kernel.cu 文件中，由 Kernel.cu 为其他方法提供接口。下面对各个层的前向计算过程进行 CUDA 加速计算和性能优化，同时实验测试。

#### 3.1. 并行算法简述

##### 3.1.1. 卷积层

对于输入数据  $\text{InData}[\text{BatchSize} * \text{InChannel} * \text{ImgHeight} * \text{ImgWidth}]$ ，依照传统采用先取数据再计算的方式，仅对取数据后的卷积过程采用并行执行。此时卷积核参数  $W[\text{OutChannel} * \text{InChannel} * \text{ConvHeight} * \text{ConvWidth}]$ ， $B[\text{OutChannel}]$ ，卷积核长度  $\text{ConvLen} = \text{InChannel} * \text{ConvHeight} * \text{ConvWidth}$ 。DataBuffer[BatchSize \*  $\text{ImgHeight} * \text{ImgWidth} * \text{ConvLen}$ ]中与卷积核进行计算的各段数据是没有交叉的，因此无须优化，得到  $\text{OutData}[\text{BatchSize} * \text{OutChannel} * \text{ImgHeight} * \text{ImgWidth}]$ 。

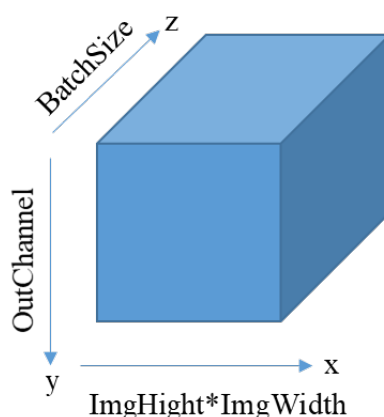


图 3-1 卷积并行化

如图 3-1，建立  $\text{BatchSize} * \text{OutChannel} * (\text{ImgHeight} * \text{ImgWidth})$  线程数的 Kernel 函数，对应  $z * y * x$  个线程，每个线程负责一个输出 OutData 的一个值。

---

```
dim3 Block(BLOCK_SIZE, BLOCK_SIZE, 1);
dim3 Grid((ImageSize2D - 1 + BLOCK_SIZE) / BLOCK_SIZE, (OutChannel - 1 +
BLOCK_SIZE) / BLOCK_SIZE, BatchSize);

Conv2D <<< Grid, Block >>> (d_CBC, d_CK, d_OutData, d_ReluData, BatchSize, InChannel,
OutChannel, ImageSize2D);
```

---

如图附-2 所示，其最终的 5 次平均训练时间为 4.69000 s/Iter。

### 3.1.2. 池化层

建立  $(BatchSize * OutChannel) * (ImgHeight / 2) * (ImgWidth / 2)$  线程数的 Kernel 函数，对应  $z * y * x$  个线程，每个线程负责一个输出 OutData 的一个值。

---

```
dim3 Block(BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE);
dim3 Grid((ImgWidth / 2 - 1 + BLOCK_SIZE) / BLOCK_SIZE, (ImgHeight / 2 - 1 +
BLOCK_SIZE) / BLOCK_SIZE, (BatchSize * Depth - 1 + BLOCK_SIZE) / BLOCK_SIZE);
_MAX_POOL_2_2 <<< Grid, Block >>> (CKC, BatchSize, Depth, ImgWidth, ImgHeight,
d_InData, d_OutData, d_GradData);
```

---

如图附-3 所示，其最终的平均训练时间为 4.71880 s/Iter。

### 3.1.3. 全连接层

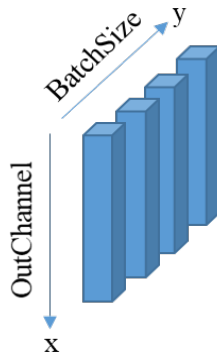


图 3-2 全连接并行化

如图 3-2，建立  $BatchSize * OutChannel$  线程数的 Kernel 函数，对应  $y * x$  个线程，每个线程负责一个输出 OutData 的一个值。

---

```
dim3 Block(BLOCK_SIZE, BLOCK_SIZE);
dim3 Grid((OutChannel - 1 + BLOCK_SIZE) / BLOCK_SIZE, (BatchSize - 1 + BLOCK_SIZE) /
BLOCK_SIZE);
FullyConnected <<< Grid, Block >>> (d_CBC, d_CKC, d_imgOut, d_reluOut, BatchSize);
```

---

如图附-4 所示，其最终的平均训练时间为 3.84620 s/Iter。

### 3.1.4. SoftMax 层

建立  $BatchSize * OutChannel$  线程数的 Kernel 函数，对应  $y * x$  个线程，每个线程负责一个输出 OutData 的一个值。

---

```
d dim3 Block(BLOCK_SIZE, BLOCK_SIZE);
dim3 Grid((SMKC->Column - 1 + BLOCK_SIZE) / BLOCK_SIZE, (mBatchSize - 1 +
BLOCK_SIZE) / BLOCK_SIZE);
SoftMax <<< Grid, Block >>> (d_SMKC, d_InData, d_OutData, mBatchSize, SMKC->Row,
```

---



---



---

SMKC->Column);

---



---

如图附-5 所示，其最终的平均训练时间为 475220 s/Iter。

### 3.1.5. 并行运算

所有层都使用并行计算后，如图附-6 所示，最终平均训练时间为 3.63560 s/Iter。

## 3.2. 性能优化

性能优化阶段在前面并行算法的基础上进行了改进，利用分块处理、内存共享、利用通用高速缓存等方式进行代码优化。

### 3.2.1. 常规三维卷积

在初步实现卷积时，沿用了先取数再计算的方式，而后对卷积过程进行了重写，将卷积运算和数据加载同时进行，此时的数据加载在卷积计算过程中进行，不同线程的数据读取之间出现了重叠。

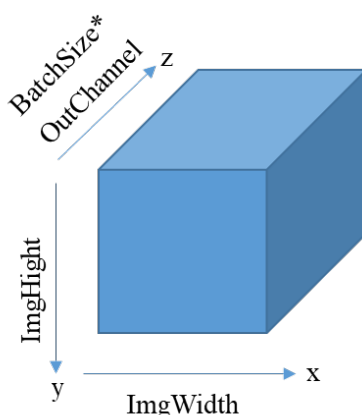


图 3-3 常规卷积并行化

如图 3-3，建立  $(BatchSize * OutChannel) * ImgHeight * ImgWidth$  线程数的 Kernel 函数，对应  $z * y * x$  个线程，每个线程负责一个输出 OutData 的一个值。

---



---

```
dim3 Block(BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE);
dim3 Grid((ImageSize2D - 1 + BLOCK_SIZE) / BLOCK_SIZE, (OutChannel - 1 +
BLOCK_SIZE) / BLOCK_SIZE, (BatchSize * OutChannel - 1 + BLOCK_SIZE) / BLOCK_SIZE);
_Conv2D_Both <<< Grid, Block >>> (d_Indata, d_CKC, d_OutData, d_ReLuData, BatchSize,
InChannel, OutChannel, ImageWidth, ImageHeight);
```

---



---

如图附-7 所示，其最终的平均训练时间为 3.65900 s/Iter。

### 3.2.2. 利用共享存储器的卷积运算

在将数据读取和卷积计算同时进行，这种分块卷积的情况下会出现数据重

叠的情况。即每个线程都会读取输入点周围的卷积核大小的数据，造成重复的访问全局存储器。于是可以采用借助于共享存储器的方式进行优化，使用光环元素的分块卷积方式。考虑到本项目中的卷积核大小仅为 5，1080Ti 的高速缓存空间大，因此利用高速寄存器进行光环元素的读取，无须加载光环元素至共享存储器。

由于每个点都需要对一个 3D 的数据和卷积核进行卷积，而各个通道间的在进行 2D 卷积时是相互独立的，数据并不重叠，利用循环对每个 2D 卷积间进行共享存储器载入即可。对于 Block\_Size 大小的分块，每个分块的共享存储器大小为 Data\_Share[Block\_Size\*Block\_Size]，位置映射不同于一维卷积，需对每一输出 OutChannel 进行，变得更加复杂，核心具体代码如下：

---

```

__global__ void _Conv2D_Opt(float* InData, ConvKernelCUDA* CKC, float* OutData, float*
ReluData, int BatchSize, int InChannel, int OutChannel, int ImageWidth, int ImageHeight)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int dep = blockIdx.z * blockDim.z + threadIdx.z;

    if (col < ImageWidth && row < ImageHeight && dep < OutChannel * BatchSize) {
        __shared__ float Data_Share[BLOCK_SIZE * BLOCK_SIZE];
        int ChannelIdx = dep % OutChannel, BatchIdx = dep / OutChannel;
        int ImgSize2D = ImageWidth * ImageHeight;
        int ConvWidth = CKC->ConvKernelWidth, ConvHeight = CKC->ConvKernelHeight;
        int ConvLen = CKC->ConvKernelWidth * CKC->ConvKernelHeight *
CKC->ConvKernelChannel;
        int HalfConv = ConvWidth >> 1;
        float sum = 0;
        for (int k = 0; k < InChannel; k++) {
            int InPos = BatchIdx * ImgSize2D * InChannel + k * ImgSize2D + row * ImageWidth +
col;

            Data_Share[threadIdx.y * blockDim.x + threadIdx.x] = InData[InPos];
            SYNC();
            int This_tile_start_point_x = blockIdx.x * blockDim.x;
            int Next_tile_start_point_x = (blockIdx.x + 1) * blockDim.x;
            int This_tile_start_point_y = blockIdx.y * blockDim.y;
            int Next_tile_start_point_y = (blockIdx.y + 1) * blockDim.y;
            int N_start_point_x = col - HalfConv;
            int N_start_point_y = row - HalfConv;
            for (int m = 0; m < ConvHeight; m++) {
                for (int n = 0; n < ConvWidth; n++) {
                    int x_index = N_start_point_x + n;

```

---

---

```

        int y_index = N_start_point_y + m;
        int ConvPos = k * ConvHeight * ConvWidth + m * ConvWidth + n;
        int ConvKernelPos = ChannelIdx * ConvLen + ConvPos;
        if (x_index >= 0 && x_index < ImageWidth && y_index >= 0 && y_index <
ImageHeight) {
            if (x_index >= This_tile_start_point_x && x_index <=
Next_tile_start_point_x &&
                y_index >= This_tile_start_point_y && y_index <=
Next_tile_start_point_y) {
                sum += Data_Share[(row + m - HalfConv) * ConvWidth + (col + n -
HalfConv)] * CKC->W[ConvKernelPos];
            }
            else {
                int Pos = BatchIdx * ImgSize2D * InChannel + k * ImgSize2D +
y_index * ImageWidth + x_index;
                sum += InData[Pos] * CKC->W[ConvKernelPos];
            }
        }
    }
    sum += CKC->B[ChannelIdx];
    SYNC();
    int OutPos = BatchIdx * ImgSize2D * OutChannel + ChannelIdx * ImgSize2D
+ row * ImageWidth + col;
    if (sum > 0.0f) {
        OutData[OutPos] = sum;
        ReluData[OutPos] = 1.0f;
    }
}
}
}

```

---

如图附-8 所示，其最终的平均训练时间为 3.6475 s/Iter，速度再次得到提升。

### 3.2.3. 性能比较

对于块大小为 BlockSize\*BlockSize，卷积核大小为 ConvHeight\*ConvWidth\*InChannel 的卷积操作，其中 ConvHeight=ConvWidth=2n+1，不考虑对卷积核的访存，不处理幽灵元素的线程块单块对全局存储器总共访问 InChannel\*BlockSize\*BlockSize\*ConvHeight\*ConvWidth，处理光环元素的线程块总访存次数为 InChannel\*(BlockSize\*BlockSize\*(2n+1)<sup>2</sup> - (ImgWidth+2n+1)\*(ImgHeight+2n+1)+ ImgWidth\*ImgHeight) = InChannel\*(BlockSize<sup>2</sup>\*(2n+1)<sup>2</sup>-(2n+1)

$(\text{ImgHeight} * \text{ImgWidth}) - (2n+1)^2$ 。

使用了共享存储器的方法，线程块访存次数为  $\text{InChannel} * \text{BlockSize}^2$ 。由此可见，使用了共享存储器后的卷积效率优于未使用，同时图片越大时，效率提升越小。卷积核尺寸越大时，效率提升越大。

### 3.2.4. 实验结果

对 16 Batch Size，5 个迭代次数结果进行汇总，如表 3-1 所示，可以看出，大部分的并行都对速度有了提升，但池化和 SoftMax 的并行却未有提升，同时优化了共享内存后的网络速度有了提升，但却慢于双步卷积，考虑到可能与数据量的大小有关，于是进行了补充实验。

表 3-1 5 次平均对比实验

方法	Batch Size	总次数	平均时间 (s/iter)
串行	16	5	4.69180
两步卷积并行	16	5	4.69000
池化并行	16	5	4.71880
全连接并行	16	5	3.84620
SoftMax 并行	16	5	4.75220
全部并行（两步卷积）	16	5	3.63560
全部并行（单步卷积）	16	5	3.65900
卷积优化后并行	16	5	3.64750

如图附-9~11 和表 3-1 所示，大数据量下的并行和卷积优化效果明显，由于总训练时长包括卷积、全连接等前向传播和梯度下降的反向传播，同时 1080Ti 的高速存储器数量完全满足需求，因此卷积优化的时间值相对较少。

最终，整个项目的加速比为串行时长/并行时长=1.463:1。

表 3-1 50 次平均对比实验

方法	Batch Size	总次数	平均时间	总时长
串行	1024	50	23.36866	1168.43300
并行（单步卷积）	1024	50	15.96346	798.17300
卷积优化后并行	1024	50	15.90028	795.01400

## 4. 总结与归纳

本项目算是对我自己的一个挑战，第一次实现了从底层算子写起的纯 C 代码神经网络框架，同时将上课所学的 CUDA 编程与之结合，达到了自我锻炼的效果。纵观此次项目，有几个很大的收获，令我印象深刻：

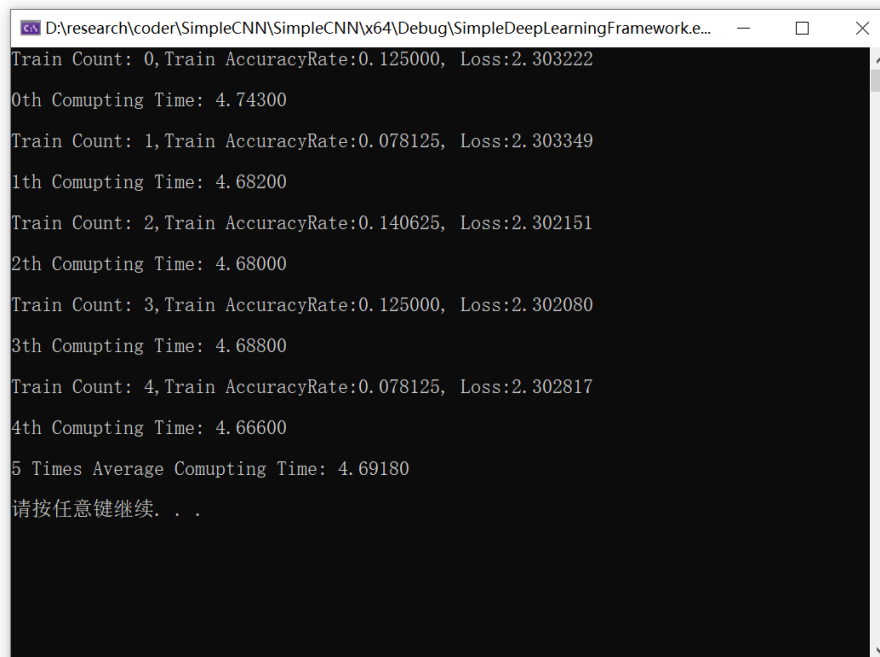
1. 对 C++ 项目、类的继承等语言属性的掌握更加熟练。以前都是写一个 `cpp` 文件的小算法，也有写过几千行代码的 C 语言项目，但此次使用了 C++ 类的继承属性，对同一过程，利用 CPU、GPU 不同子类进行继承，使我对 C++ 语言属性、项目编写的理解大大加深；
2. 对 CUDA 并行加速的思想更加透彻。从对 Kernel 函数架构的马马虎虎，到最后能完成 `Batch*Channel*Height*Width` 四个维度、3D 卷积的共享存储器的映射，我对并行思想更加的熟悉；
3. 对 CUDA 编程实战的理解更加深刻。倘若我实现一个简单的算法，就像写一个 `cpp` 文件一样写一个 `cu`，那我就无法知道 C++ 与 CUDA 混合编程中需要注意的问题，比如 C++ 项目无法直接编译 `cu` 文件，需要将 `cu` 中函数以 `extern "C"` 的形式引入其中进行调用；又比如在编写时需要进行错误处理，不然就会像我一样调试一天找 BUG 在哪，最后发现是函数调用了未在显存上的程序...刚开始我还以为是环境的问题；又比如要做好内存管理，不然在大数据训练时会出现 `Out Of Memory` 的情况。

总之，此次的大作业使我收获颇多，完成了自己 C++ 手撸神经网络的部分愿望，实现了 GPU 编程的任务，迈出了算法工程师底层优化小小的一步~

## 参考文献

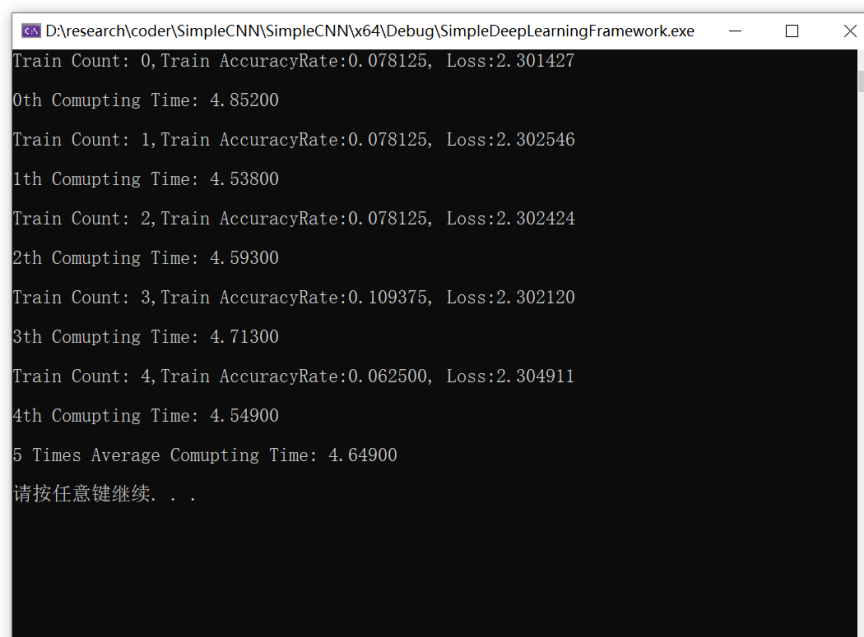
- [1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.
- [2] 简单深度学习框架实现 Simple Deep Learning Framework:  
<https://github.com/lygyue/SimpleDeepLearningFramework>
- [3] CUDA Kernel 核函数内分配存储空间 :  
[https://blog.csdn.net/u010454261/article/details/74972953?utm\\_medium=distribute.pc\\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-6.control&dist\\_request\\_id=1332036.10371.16191712279570405&depth\\_1-utm\\_source=distribute.pc\\_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-6.control](https://blog.csdn.net/u010454261/article/details/74972953?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-6.control&dist_request_id=1332036.10371.16191712279570405&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7Edefault-6.control)
- [4] 赵开勇、汪朝辉、程亦超[译]B. Kirk, Wen-mei W. Hwu, David. (2013 年 11 月). 大规模并行处理器编程实战(第 2 版). 清华大学出版社.

## 附录



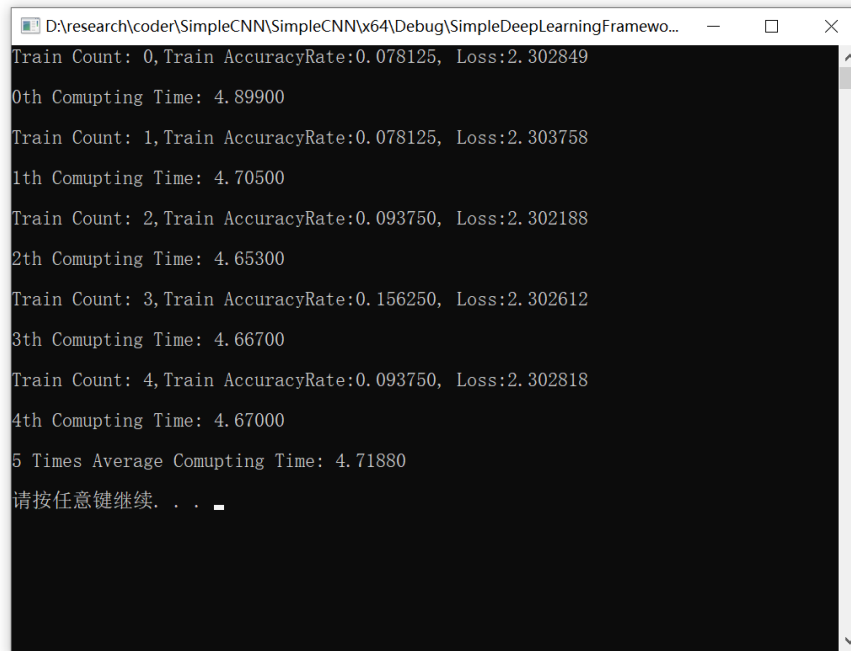
```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Debug\SimpleDeepLearningFramework.e...  
Train Count: 0,Train AccuracyRate:0.125000, Loss:2.303222  
0th Comupting Time: 4.74300  
Train Count: 1,Train AccuracyRate:0.078125, Loss:2.303349  
1th Comupting Time: 4.68200  
Train Count: 2,Train AccuracyRate:0.140625, Loss:2.302151  
2th Comupting Time: 4.68000  
Train Count: 3,Train AccuracyRate:0.125000, Loss:2.302080  
3th Comupting Time: 4.68800  
Train Count: 4,Train AccuracyRate:0.078125, Loss:2.302817  
4th Comupting Time: 4.66600  
5 Times Average Comupting Time: 4.69180  
请按任意键继续. . .
```

图 附-1 串行算法的训练时间



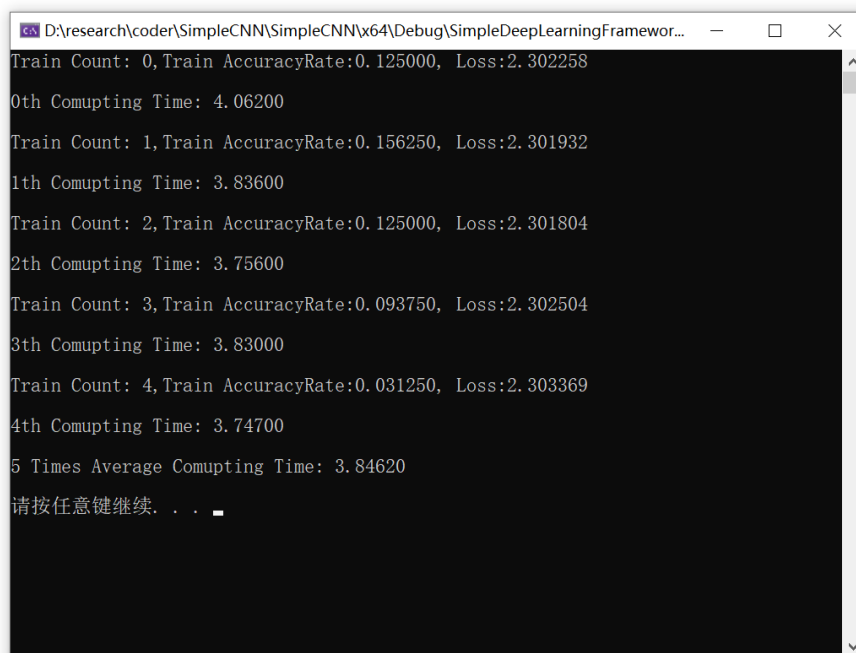
```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Debug\SimpleDeepLearningFramework.exe  
Train Count: 0,Train AccuracyRate:0.078125, Loss:2.301427  
0th Comupting Time: 4.85200  
Train Count: 1,Train AccuracyRate:0.078125, Loss:2.302546  
1th Comupting Time: 4.53800  
Train Count: 2,Train AccuracyRate:0.078125, Loss:2.302424  
2th Comupting Time: 4.59300  
Train Count: 3,Train AccuracyRate:0.109375, Loss:2.302120  
3th Comupting Time: 4.71300  
Train Count: 4,Train AccuracyRate:0.062500, Loss:2.304911  
4th Comupting Time: 4.54900  
5 Times Average Comupting Time: 4.64900  
请按任意键继续. . .
```

图 附-2 卷积层并行后的训练时间



```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Debug\SimpleDeepLearningFrameworko...
Train Count: 0,Train AccuracyRate:0.078125, Loss:2.302849
0th Comupting Time: 4.89900
Train Count: 1,Train AccuracyRate:0.078125, Loss:2.303758
1th Comupting Time: 4.70500
Train Count: 2,Train AccuracyRate:0.093750, Loss:2.302188
2th Comupting Time: 4.65300
Train Count: 3,Train AccuracyRate:0.156250, Loss:2.302612
3th Comupting Time: 4.66700
Train Count: 4,Train AccuracyRate:0.093750, Loss:2.302818
4th Comupting Time: 4.67000
5 Times Average Comupting Time: 4.71880
请按任意键继续. . .
```

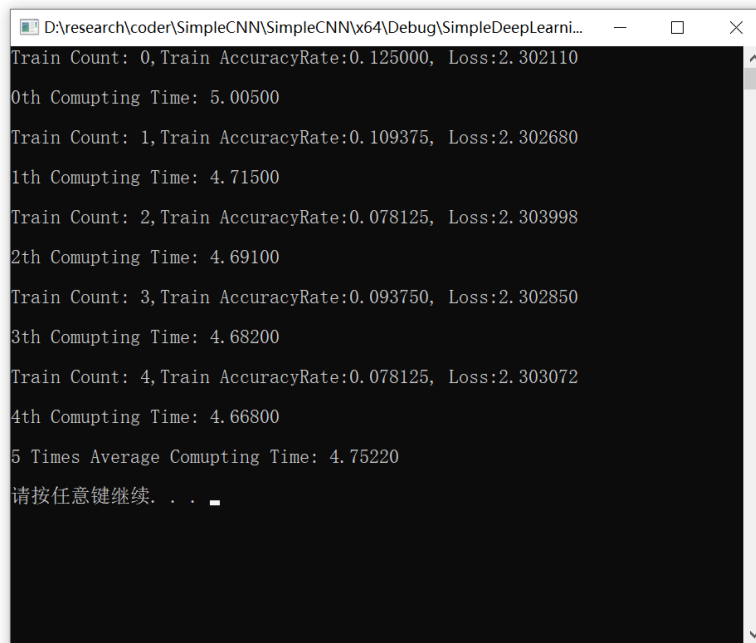
图 附-3 池化层并行后的训练时间



```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Debug\SimpleDeepLearningFrameworkor...
Train Count: 0,Train AccuracyRate:0.125000, Loss:2.302258
0th Comupting Time: 4.06200
Train Count: 1,Train AccuracyRate:0.156250, Loss:2.301932
1th Comupting Time: 3.83600
Train Count: 2,Train AccuracyRate:0.125000, Loss:2.301804
2th Comupting Time: 3.75600
Train Count: 3,Train AccuracyRate:0.093750, Loss:2.302504
3th Comupting Time: 3.83000
Train Count: 4,Train AccuracyRate:0.031250, Loss:2.303369
4th Comupting Time: 3.74700
5 Times Average Comupting Time: 3.84620
请按任意键继续. . .
```

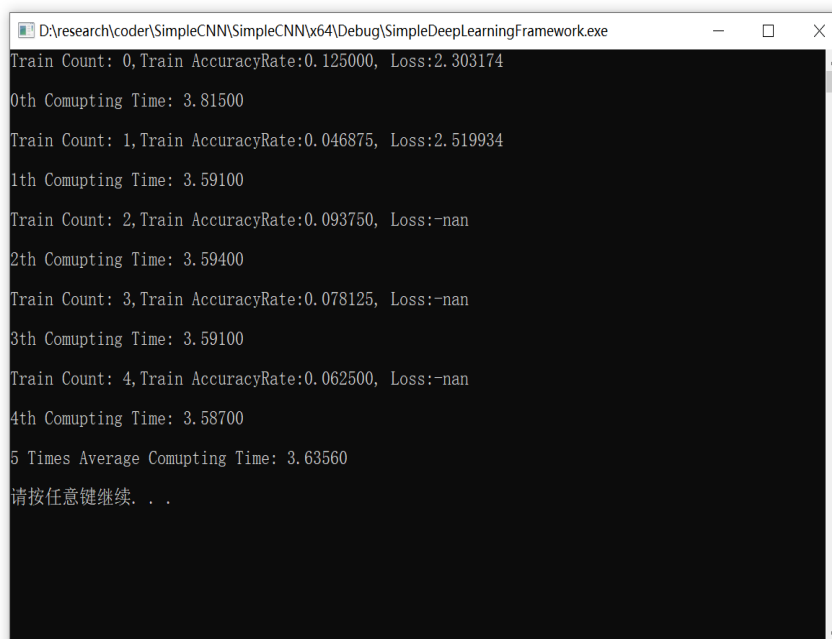
图 附-4 全连接层并行化的训练时间





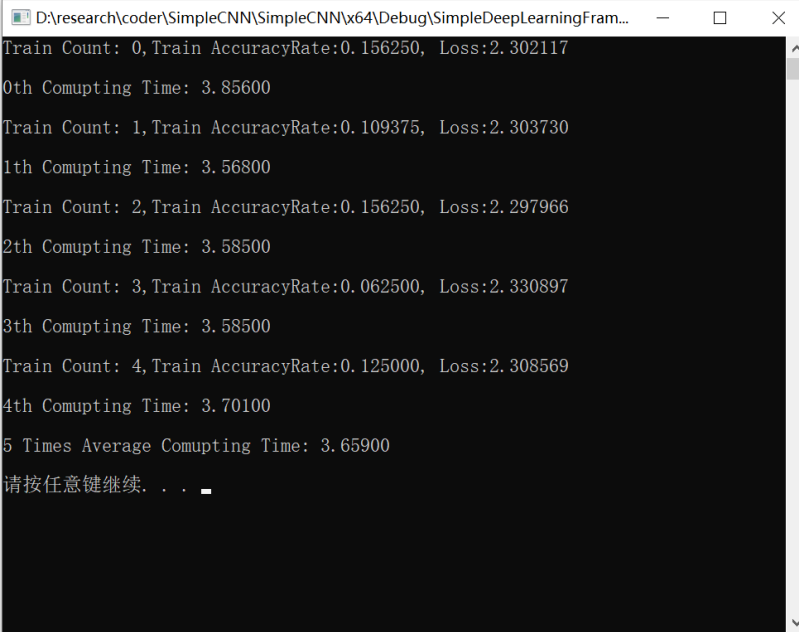
```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Debug\SimpleDeepLearn...
Train Count: 0,Train AccuracyRate:0.125000, Loss:2.302110
0th Comupting Time: 5.00500
Train Count: 1,Train AccuracyRate:0.109375, Loss:2.302680
1th Comupting Time: 4.71500
Train Count: 2,Train AccuracyRate:0.078125, Loss:2.303998
2th Comupting Time: 4.69100
Train Count: 3,Train AccuracyRate:0.093750, Loss:2.302850
3th Comupting Time: 4.68200
Train Count: 4,Train AccuracyRate:0.078125, Loss:2.303072
4th Comupting Time: 4.66800
5 Times Average Comupting Time: 4.75220
请按任意键继续. . .
```

图 附-5 SoftMax 层并行后的训练时间



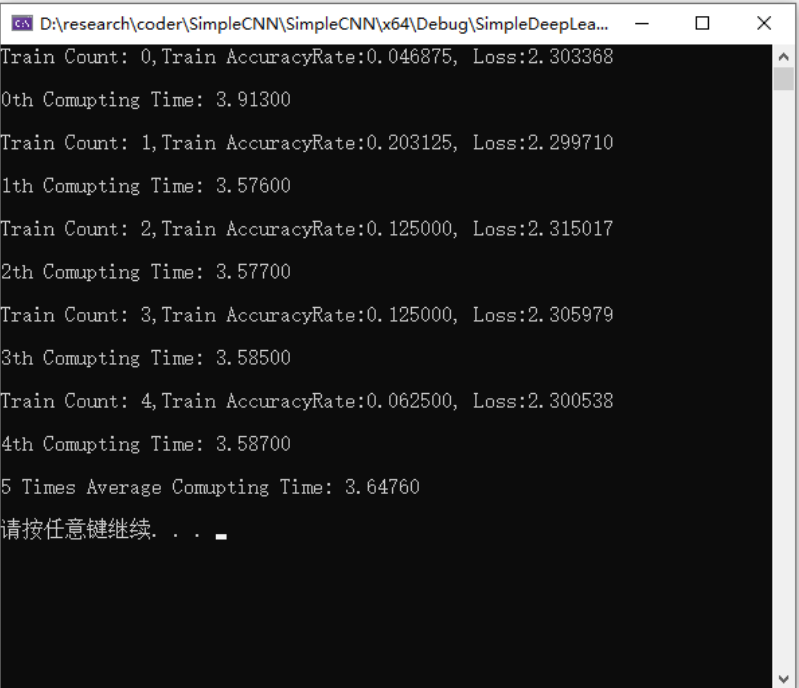
```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Debug\SimpleDeepLearningFramework.exe
Train Count: 0,Train AccuracyRate:0.125000, Loss:2.303174
0th Comupting Time: 3.81500
Train Count: 1,Train AccuracyRate:0.046875, Loss:2.519934
1th Comupting Time: 3.59100
Train Count: 2,Train AccuracyRate:0.093750, Loss:-nan
2th Comupting Time: 3.59400
Train Count: 3,Train AccuracyRate:0.078125, Loss:-nan
3th Comupting Time: 3.59100
Train Count: 4,Train AccuracyRate:0.062500, Loss:-nan
4th Comupting Time: 3.58700
5 Times Average Comupting Time: 3.63560
请按任意键继续. . .
```

图 附-6 并行算法的训练时间



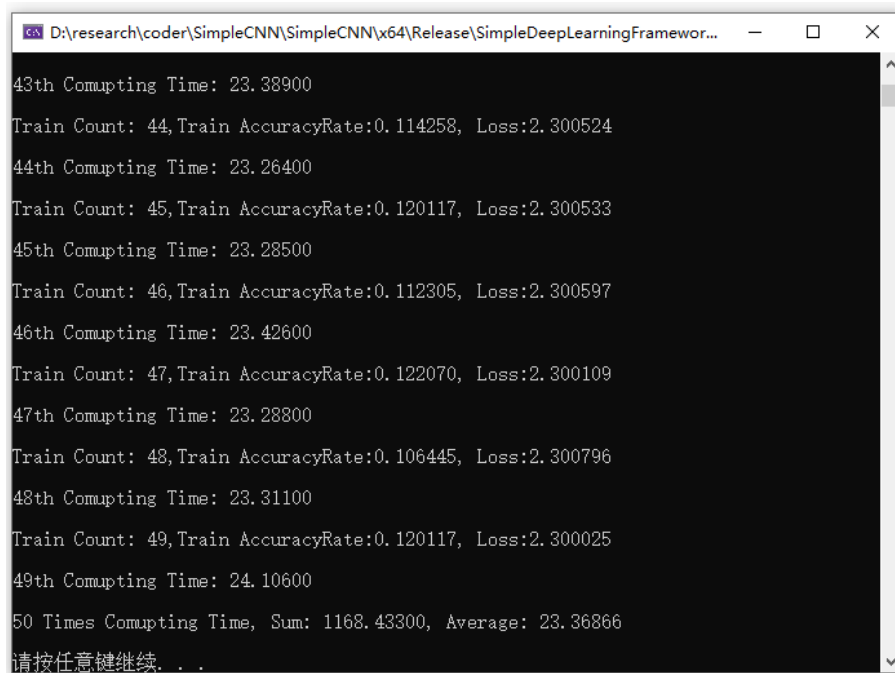
```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Debug\SimpleDeepLearningFram...
Train Count: 0,Train AccuracyRate:0.156250, Loss:2.302117
0th Comupting Time: 3.85600
Train Count: 1,Train AccuracyRate:0.109375, Loss:2.303730
1th Comupting Time: 3.56800
Train Count: 2,Train AccuracyRate:0.156250, Loss:2.297966
2th Comupting Time: 3.58500
Train Count: 3,Train AccuracyRate:0.062500, Loss:2.330897
3th Comupting Time: 3.58500
Train Count: 4,Train AccuracyRate:0.125000, Loss:2.308569
4th Comupting Time: 3.70100
5 Times Average Comupting Time: 3.65900
请按任意键继续. . .
```

图 附-7 数据读取与卷积同时的并行训练时间



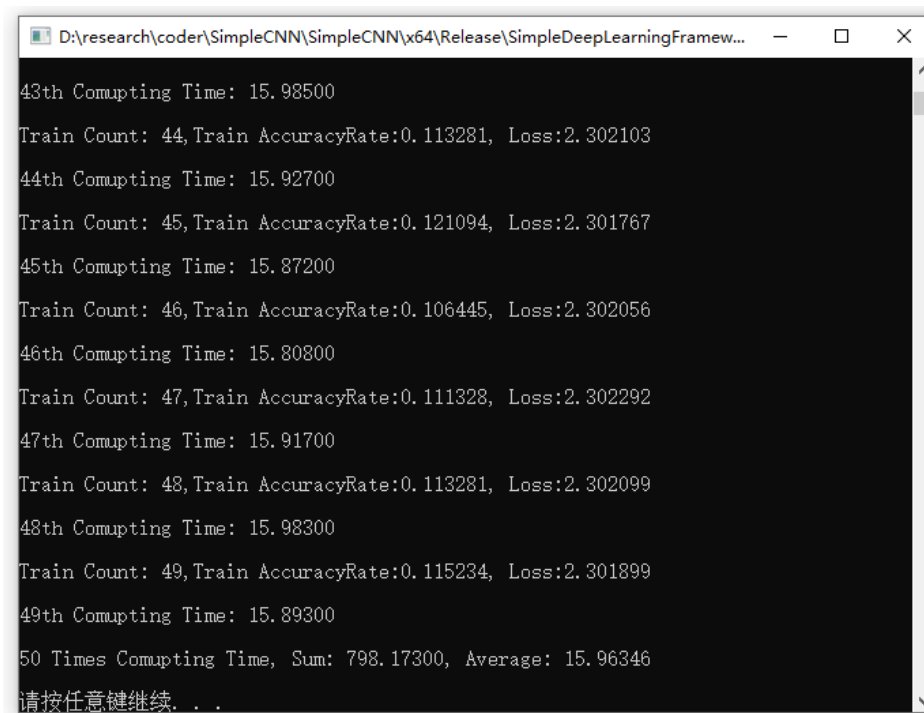
```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Debug\SimpleDeepLea...
Train Count: 0,Train AccuracyRate:0.046875, Loss:2.303368
0th Comupting Time: 3.91300
Train Count: 1,Train AccuracyRate:0.203125, Loss:2.299710
1th Comupting Time: 3.57600
Train Count: 2,Train AccuracyRate:0.125000, Loss:2.315017
2th Comupting Time: 3.57700
Train Count: 3,Train AccuracyRate:0.125000, Loss:2.305979
3th Comupting Time: 3.58500
Train Count: 4,Train AccuracyRate:0.062500, Loss:2.300538
4th Comupting Time: 3.58700
5 Times Average Comupting Time: 3.64760
请按任意键继续. . .
```

图 附-8 共享内存后的并行训练时间



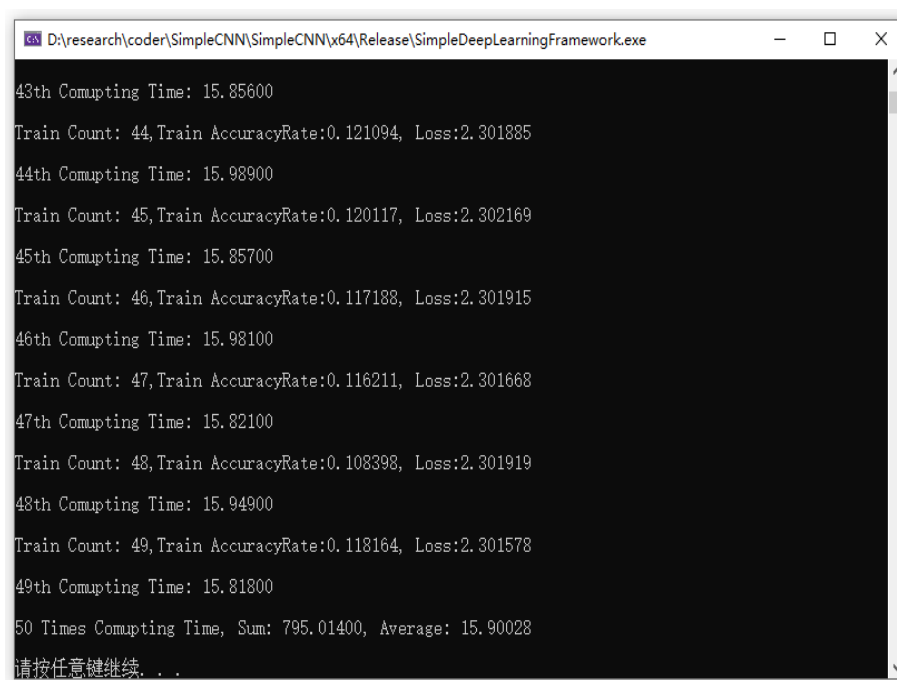
```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Release\SimpleDeepLearningFrameworkor...
43th Computing Time: 23.38900
Train Count: 44, Train AccuracyRate:0.114258, Loss:2.300524
44th Computing Time: 23.26400
Train Count: 45, Train AccuracyRate:0.120117, Loss:2.300533
45th Computing Time: 23.28500
Train Count: 46, Train AccuracyRate:0.112305, Loss:2.300597
46th Computing Time: 23.42600
Train Count: 47, Train AccuracyRate:0.122070, Loss:2.300109
47th Computing Time: 23.28800
Train Count: 48, Train AccuracyRate:0.106445, Loss:2.300796
48th Computing Time: 23.31100
Train Count: 49, Train AccuracyRate:0.120117, Loss:2.300025
49th Computing Time: 24.10600
50 Times Computing Time, Sum: 1168.43300, Average: 23.36866
请按任意键继续. . .
```

图 附-9 50 次串行训练时间



```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Release\SimpleDeepLearningFrameworkor...
43th Computing Time: 15.98500
Train Count: 44, Train AccuracyRate:0.113281, Loss:2.302103
44th Computing Time: 15.92700
Train Count: 45, Train AccuracyRate:0.121094, Loss:2.301767
45th Computing Time: 15.87200
Train Count: 46, Train AccuracyRate:0.106445, Loss:2.302056
46th Computing Time: 15.80800
Train Count: 47, Train AccuracyRate:0.111328, Loss:2.302292
47th Computing Time: 15.91700
Train Count: 48, Train AccuracyRate:0.113281, Loss:2.302099
48th Computing Time: 15.98300
Train Count: 49, Train AccuracyRate:0.115234, Loss:2.301899
49th Computing Time: 15.89300
50 Times Computing Time, Sum: 798.17300, Average: 15.96346
请按任意键继续. . .
```

图 附-10 50 次全部并行（单步卷积）的训练时间



```
D:\research\coder\SimpleCNN\SimpleCNN\x64\Release\SimpleDeepLearningFramework.exe

43th Comupting Time: 15.85600
Train Count: 44,Train AccuracyRate:0.121094, Loss:2.301885
44th Comupting Time: 15.98900
Train Count: 45,Train AccuracyRate:0.120117, Loss:2.302169
45th Comupting Time: 15.85700
Train Count: 46,Train AccuracyRate:0.117188, Loss:2.301915
46th Comupting Time: 15.98100
Train Count: 47,Train AccuracyRate:0.116211, Loss:2.301668
47th Comupting Time: 15.82100
Train Count: 48,Train AccuracyRate:0.108398, Loss:2.301919
48th Comupting Time: 15.94900
Train Count: 49,Train AccuracyRate:0.118164, Loss:2.301578
49th Comupting Time: 15.81800
50 Times Comupting Time, Sum: 795.01400, Average: 15.90028
请按任意键继续. . .
```

图 附-11 50 次并行优化后的训练时间