华东师范大学计算机科学技术系上机实践报告

课程名称：并行计算	**年级**：2020 级	**上机实践成绩**：
指导教师：钱莹	**姓名**：李锦浩	
上机实践名称：CNN 的并行实现	**学号**：10205102464	**上机实践日期**：
上机实践编号：	**组号**：	**上机实践时间**：

一、 使用方式

使用根目录下的.travis.yml 文件或 appveyor.yml 文件中的不同的 CMake 命令在 MacOS，Linux 或 Windows 上编译。

为了使用 MINST 数据集，需要将相应的压缩包解压到 data/目录下。

二、 串行实现方式

本实现基于 GitLab 中的一个项目，网址为 https://gitlab.com/ribtoks/yannpp。其中包含各种复杂的数据结构，现忽略底层实现，主要讲述 CNN 中各层的实现方式。

2.1 卷积层

本实验中卷积层的卷积方式主要包括循环卷积与 2D 卷积。

2.1.1 循环卷积

对于离散信号 $x(n), h(n)$，可以定义周期 N 的循环卷积 $(x \otimes h)[n]$ 为：

$$\begin{aligned}(x_N * h)[n] &= \sum_{m=-\infty}^{\infty} h[m] \cdot x_N[n-m] \\ &= \sum_{m=-\infty}^{\infty} \left(h[m] \cdot \sum_{k=-\infty}^{\infty} x[n-m-kN] \right)\end{aligned} \tag{1}$$

对于循环卷积层的输入 I 和卷积核 K，输出 S 可以定义为：

$$S(i,j) = (I * K)(i,j) = \text{sum}[I(m,n)K(i-m, j-n)] \tag{2}$$

核心部分的实现代码如下：

```
for (int fi = 0; fi < fsize; fi++)
{
    auto filter = this->filter_weights_[fi].slice();
    auto &bias = this->filter_biases_[fi](0);
    // 2D loop over the input and calculation convolution of input and
       current filter
    // convolution is S(i, j) = (I * K)(i, j) = Sum[ I(m, n)K(i - m, j -
       n) ]
    // which is commutative i.e. (I * K)(i, j) = Sum[ I(i - m, j -
       n)K(m, n) ]
    // where I is input and K is kernel (filter weights)
    for (int y = 0; y < output_shape.y(); y++)
    {
        int ys = y * this->stride_.y() - pad_y;
        for (int x = 0; x < output_shape.x(); x++)
        {
            int xs = x * this->stride_.x() - pad_x;
            // in this case cross-correlation (I(m, n)K(i + m, j + n)) is
               used
            // (kernel is not rot180() flipped for the convolution, not
               commutative)
            // previous formula (w*x + b) is used with convolution
               instead of product
            result(x, y, fi) =
                bias + dot<T>(
                    this->input_.slice(
                        index3d_t(xs, ys, 0),
                        index3d_t(xs + filter_shape.x() - 1,
                                  ys + filter_shape.y() - 1,
                                  input_shape.z() - 1)),
                    filter);
        }
    }
}
```

2.1.2 2D 卷积

在卷积神经网络中，二维卷积核并不是真正的二维，之所以称其为 2D 卷积，是因为卷积核在数据上沿 2 维滑动。如下图所示：

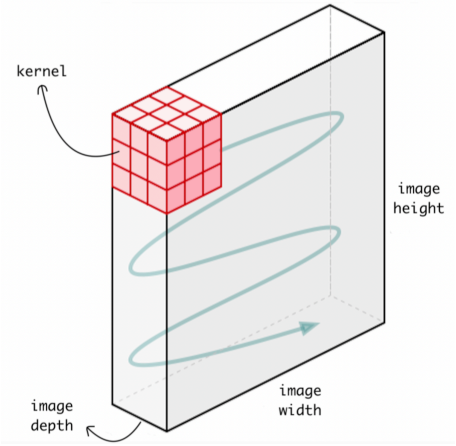


图 1　2D 卷积

本实验的具体实现方式是：将输入分成若干与卷积核形状相同的块，再跟卷积核作点乘运算，具体核心代码如下：

```
for (size_t i = 0; i < patches_size; i++)
{
    // result has size of [filters_number]
    auto conv = dot21(filters, patches[i]);
    assert(conv.shape() == shape3d_t(output_shape.z(), 1, 1));
    conv.add(biases);
    result.insert(result.end(), conv.data().begin(), conv.data().end());
}
```

2.2 全连接层

卷积层的输出展平后经由全连接层可以得到期望的分类结果，因此全连接层在卷积神经网络中是至关重要的。

2.2.1 前向传播

前向传播的过程可以简述为输出由权重线性组合再经由激活函数输出，本实验的激活函数一律采用 RuLu 激活函数，上述过成可以形式化为：

$$z^{l+1} = \text{ReLu}(\omega_l^{l+1} \cdot a^l + b) \tag{3}$$

其中 z 为输出，ω 为权重，a 为输入，b 为偏置项。实现代码如下：

```
virtual array3d_t<T> feedforward(array3d_t<T> &&input) override {
    input_shape_ = input.shape();
    input_ = std::move(input);
    input_.flatten();
    // z = w*a + b
    output_ = dot21(weights_, input_); output_.add(bias_);
    return activator_.activate(output_);
}
```

2.2.2 反向传播

反向传播通过链式法则实现，令 $\delta^l = \frac{\mathrm{d}a^{l+1}}{\mathrm{d}a^l}$，则

$$\begin{aligned}
\frac{\mathrm{d}a^{l+1}}{\mathrm{d}b^l} &= \frac{\mathrm{d}a^{l+1}}{\mathrm{d}a^l} \cdot \frac{\mathrm{d}a^l}{\mathrm{d}b^l} = \delta^l \\
\frac{\mathrm{d}a^{l+1}}{\mathrm{d}\omega^l} &= \frac{\mathrm{d}a^{l+1}}{\mathrm{d}a^l} \cdot \frac{\mathrm{d}a^l}{\mathrm{d}\omega^l} = \delta^l \cdot a^{l-1} \\
\delta^{l-1} &= \frac{\mathrm{d}a^{l+1}}{\mathrm{d}a^{l-1}} = \frac{\mathrm{d}a^{l+1}}{\mathrm{d}a^l} \cdot \frac{\mathrm{d}a^l}{\mathrm{d}a^{l-1}} = \delta^l \cdot \omega_{l-1}^l
\end{aligned} \tag{4}$$

可见 δ 反映了当前层的梯度，具体的实现代码如下：

```
virtual array3d_t<T> backpropagate(array3d_t<T> &&error) override {
    array3d_t<T> delta, delta_next, delta_nabla_w;
    // delta(l) = (w(l+1) * delta(l+1)) [X] derivative(z(l))
    // (w(l+1) * delta(l+1)) comes as the gradient (error) from the
       "previous" layer
    delta = activator_.derivative(output_); delta.element_mul(error);
    // dC/db = delta(l)
    nabla_b_.add(delta);
    // dC/dw = a(l-1) * delta(l)
    delta_nabla_w = outer_product(delta, input_);
    nabla_w_.add(delta_nabla_w);
    // w(l) * delta(l)
    delta_next = transpose_dot21(weights_, delta);
    delta_next.reshape(input_shape_);
    return delta_next;
}
```

卷积层反向传播的实现方式类似。

三、 串行测试

串行测试分为两部分，分别是各层的运行时间测试和神经网络的运行时间测试。测试所用的处理器为 Apple M1 Pro,CPU 的核心数为 8，内存大小为 16GB。

3.1 测试一

3.1.1 卷积层

卷积层的测试代码包含在 convolution——.cpp 文件中，修改参数得到实验结果为：

表 1　卷积层运行时间测试

Input Shape	Filter Shape	Output Shape	Padding Type	Run Time
5*5*5	7*3*3*5	5*5*7	Same	0.00147s
50*50*5	7*3*3*5	50*50*7	Same	0.137883s
500*500*5	7*3*3*5	500*500*7	Same	14.0277s
50*50*50	7*3*3*50	50*50*7	Same	1.28739s
50*50*5	7*30*30*5	50*50*7	Same	12.2755s
50*50*5	70*3*3*5	50*50*70	Same	1.37161s

通过改变输入形状，卷积核形状和通道数，发现运行时间会对应线性变换，由此得串行代码实现正确。

3.1.2 池化层

本实验所用的池化操作为最大池化，其测试代码包含在 pooling.cpp 文件中，修改参数得到实验结果为：

表 2　池化层运行时间测试

Input Shape	Window Size	Stride	Output Shape	Run Time
5*5*10	2	2	2*2*10	0.000188s
50*50*10	2	2	25*25*10	0.011725s
500*500*10	2	2	250*250*10	0.569993s
5000*5000*10	2	2	2500*2500*10	59.3533s

观察实验结果发现，当输入形状较小时，改变输入形状，窗口大小和通道数，运行时间没有对应线性变换，因为程序运行时有其他指令的开销。当输入形状较大时，运行时间对应线性变换，由此得串行代码实现正确。

3.1.3 全连接层

卷积层的测试代码包含在 tf _dense.cpp 文件中，修改参数得到实验结果为:

表 3　全连接层运行时间测试

Input Shape	Output Shape	Parameter Amount	Run Time
30*1*1	20*1*1	600	4.3e-05s
300*1*1	200*1*1	60000	0.002517s
3000*1*1	2000*1*1	6000000	0.118662s
30000*1*1	20000*1*1	600000000	12.1124s

当使用全连接层是，一个重要的观察指标是它的参数量，因为他会产生巨大的开销。观察实验结果发现，运行时间与参数量的大小成正比，由此得串行代码实现正确。

3.2 测试二

测试二围绕分类 MNIST 数据集展开，测试代码包含在 test_mnist.cpp 文件中。

3.2.1 LearnMnistDenseTest

该网络架构如下，其中包含两个全连接层。

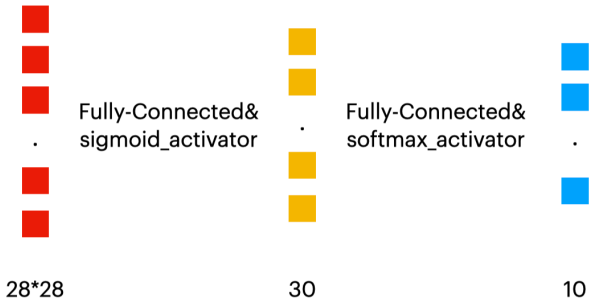


图 2　LearnMnistDenseTest

超参数设置为：epoch=2，mini_batch_size=10，learning_rate=0.01，decay_rate=20。当 1000 个样本中有 800 个样本预测正确时测试通过。

3.2.2 DeepLearningLoopMnistTest

该网络架构如下，其中包含一个循环卷积层和两个全连接层。

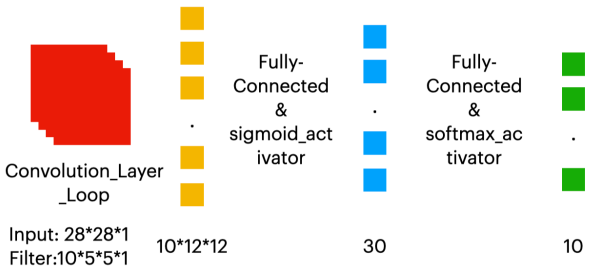


图 3　DeepLearningLoopMnistTest

超参数设置为：epoch=2，mini_batch_size=10，learning_rate=0.001，decay_rate=10。当有一半测试样本预测正确时测试通过。

3.2.3 DeepLearning2DMnistTest

该网络架构如下，其中包含一个 2D 卷积层和两个全连接层。

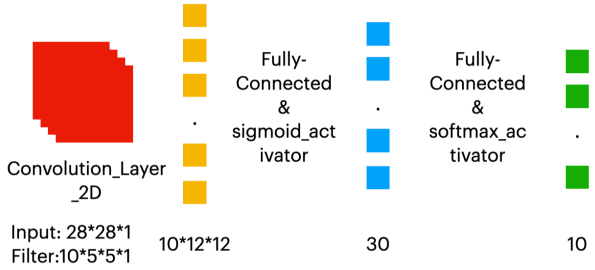


图 4　DeepLearning2DMnistTest

超参数设置为：epoch=2，mini_batch_size=10，learning_rate=0.001，decay_rate=10。当有一半测试样本预测正确时测试通过。

3.2.4 测试结果

得到的测试结果为：

表 4　分类网络运行时间测试

	Number of Test Samples	Hit	Run Time
LearnMnistDenseTest	1000	855/875	22152ms
DeepLearningLoopMnistTest	334	204/273	351308s
DeepLearning2DMnistTest	334	211/281	126505s

其中 Hit 中的数据代表两轮测试中各自预测正确的样本数。

四、 并行实现方式

并行实现基于 OpenMP，主要思想是任务尽量平均分成若干大小相同子任务，然后分配给对应数量的线程，对于剩余的任务，按顺序分配给编号小的线程。代码框架如下：

```
#  pragma omp parallel num_threads(num_threads)
{
        size_t my_rank = omp_get_thread_num();
        size_t thread_count = omp_get_num_threads();
        size_t local_x = size_of_task / thread_count;
        size_t sub = size_of_task % thread_count;
        size_t start = my_rank * local_x + (sub > my_rank ? my_rank :
            sub);
        size_t end = start + local_x + (sub > my_rank ? 1 : 0);
        /* Serial Program */
}
```

4.1 循环卷积层

代码见 convolutionlayer.h 文件。

4.1.1 前向传播

观察循环卷积的前向传播代码，发现其主要在对各个卷积核跟输入做卷积运算，因此可以按照上述方法将卷积核平均分配给不同的线程。

4.1.2 反向传播

与全连接层的反向传播类似，循环卷积层的反向传播主要包括权重 ω、偏置项 b 和反应当前层梯度的变量 δ 的更新，他们都是基于各个卷积核进行对应的运算，因此可以按照与前向传播相同的方法进行并行化。

4.2 2D 卷积层

代码见 convolutionlayer.h 文件。

4.2.1 前向传播

与循环卷积层的前向传播不同，2D 卷积层的前向传播主要基于 patch 块实现。具体地，它先将输入分成若干与卷积核形状相同的 patch 块，在跟卷积核做点乘运算，最后加上偏置项后插入到一个 result 向量中，因此在并行时会导致 patch 块插入顺序的错误。

4.2.2 反向传播

相同地，在反向传播中所有基于 patch 块的操作并行时都会导致程序运行错误（Assertion Failed），而基于卷积核的操作可以按照之前相同的方法进行并行化。

4.3 池化层

代码见 poolinglayer.h 文件

4.3.1 前向传播

池化层的前向传播基于输入通道实现，对于每一个输入通道，计算当前窗口的最大值并记录到输出特征图的相应位置中，因此可以按照之前相同的方法进行并行化。

4.3.2 反向传播

池化层的前向传播基于输出通道实现，对于每一个输出通道，计算当前窗口的梯度并记录到值最大的神经元的相应位置中，因此可以按照之前相同的方法进行并行化。

4.4 全连接层

代码见 fullyconnectedlayer.h 文件

4.4.1 前向传播

全连接层的前向传播基于矩阵的点乘运算实现，因此可以按照之前相同的方法进行并行化。

4.4.2 反向传播

全连接层的反向传播基于矩阵的外积、转置和点乘等运算实现，因此可以按照之前相同的方法进行并行化。

五、 并行测试

并行测试分为两部分，分别是各层的并行效果测试和神经网络的并行效果测试。测试所用的处理器为 Apple M1 Pro，CPU 的核心数为 8，内存大小为 16GB。线程数可以通过 array3d_math.h 中的变量 num_thread 控制，测试中所设置的线程数均为 1，2，4，8，16，32 六种数量。

5.1 测试一

5.1.1 循环卷积层

循环卷积层测试分为三个问题规模，输入形状分别为 50×50×50（Half）、100×100×50（Origin）、200×200×50（Double），卷积核形状均为 7×3×3×50，测试结果如下：

表 5　循环卷积层并行时间

Input Shape	50×50×50	100×100×50	200×200×50
1 Threads	1.20656s	4.98343s	20.2446s
2 Threads	0.722776s	2.91948s	11.9909s
4 Threads	0.385299s	1.54318s	6.18871s
8 Threads	0.24097s	0.89841s	3.58686s
16 Threads	0.229838s	0.899553s	3.59355s
32 Threads	0.230617s	0.935639s	3.60868s

计算加速比和效率如下：

表 6　循环卷积层并行加速比和效率

Number of Threads		1	2	4	8	16	32
Half	Speedup	1.00	1.67	3.13	5.01	5.25	5.23
	Efficiency	1.00	0.84	0.78	0.63	0.66	0.65
Original	Speedup	1.00	1.71	3.23	5.55	5.54	5.33
	Efficiency	1.00	0.86	0.81	0.69	0.69	0.67
Double	Speedup	1.00	1.69	3.27	5.64	5.63	5.61
	Efficiency	1.00	0.85	0.82	0.71	0.70	0.70

将三种问题规模的加速比和效率汇总并画图得到结果如下：

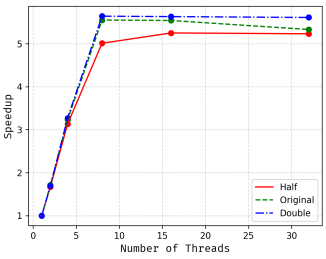


图 5　Loop Convolutional Layer Speedup

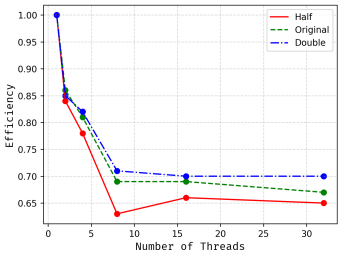


图 6　Loop Convolutional Layer Efficiency

同时可以发现实验结果大致满足效率随着线程数增大而减小，随着任务量增大而增大的规律。

5.1.2 2D 卷积层

2D 卷积层测试分为三个问题规模，输入形状分别为 50×50×50（Half）、100×100×50（Origin）、200×200×50（Double），卷积核形状均为 7×3×3×50，测试结果如下：

表 7　2D 卷积层并行时间

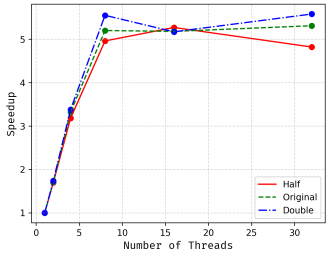
Input Shape	50×50×50	100×100×50	200×200×50
1 Threads	1.21706s	5.02127s	20.7476s
2 Threads	0.717032s	2.93259s	11.8976s
4 Threads	0.38289s	1.51233s	6.14212s
8 Threads	0.245322s	0.964955s	3.73766s
16 Threads	0.230747s	0.970218s	4.0128s
32 Threads	0.252517s	0.945606s	3.71788s

计算加速比和效率如下：

表 8　2D 卷积层并行加速比和效率

Number of Threads		1	2	4	8	16	32
Half	Speedup	1.00	1.70	3.18	4.96	5.27	4.82
	Efficiency	1.00	0.85	0.80	0.62	0.66	0.60
Original	Speedup	1.00	1.71	3.32	5.20	5.18	5.31
	Efficiency	1.00	0.86	0.83	0.65	0.65	0.66
Double	Speedup	1.00	1.74	3.38	5.55	5.17	5.58
	Efficiency	1.00	0.87	0.85	0.69	0.65	0.70

将三种问题规模的加速比和效率汇总并画图得到结果如下：



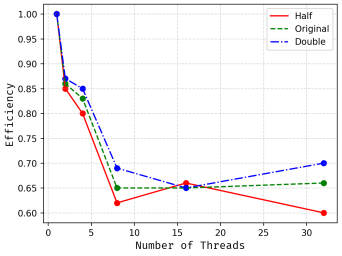


图 7　2D Convolutional Layer Speedup　**图 8　2D Convolutional Layer Efficiency**

5.1.3 池化层

池化层测试分为三个问题规模，输入形状分别为500×500×20（Half）、1000×1000×20（Origin）、2000×2000×20（Double），窗口大小和步长均为2，测试结果如下：

表9　池化层并行时间

Input Shape	500×500×20	1000×1000×20	2000×2000×20
1 Threads	0.810491s	3.26202s	13.7975s
2 Threads	0.425852s	1.71660s	7.25889s
4 Threads	0.241000s	0.958903s	4.50202s
8 Threads	0.208261s	0.781608s	3.67811s
16 Threads	0.192426s	0.832559s	3.71140s
32 Threads	0.181896s	0.735488s	3.47381s

计算加速比和效率如下：

表10　池化层并行加速比和效率

Number of Threads		1	2	4	8	16	32
Half	Speedup	1.00	1.90	3.36	3.89	4.21	4.46
	Efficiency	1.00	0.95	0.84	0.49	0.53	0.56
Original	Speedup	1.00	1.90	3.40	4.17	3.92	4.44
	Efficiency	1.00	0.95	0.85	0.52	0.49	0.56
Double	Speedup	1.00	1.90	3.06	3.75	3.72	3.79
	Efficiency	1.00	0.95	0.77	0.47	0.47	0.47

此时发现当线程数从4变为8时，程序的加速效果不是很明显。因为任务的分配方式类似于静态调度，前四个线程的任务数为4，后四个线程的任务数为3，所以程序的加速效果受到了一定限制。将三种问题规模的加速比和效率汇总并画图得到结果如下：

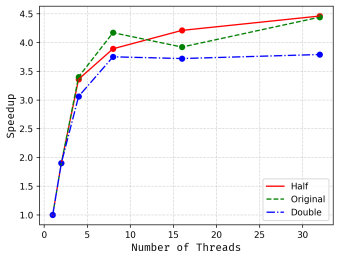


图 9　Pooling Layer Speedup

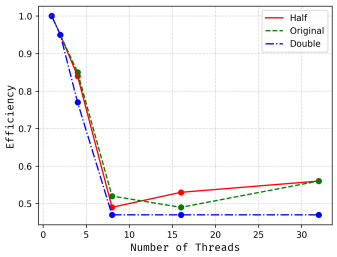


图 10　Pooling Layer Efficiency

5.1.4 全连接层

全连接层测试分为三个问题规模，他们的输入和输出的形状都分别为 10000×1×1（Half）、20000×1×1（Origin）、40000×1×1（Double），由此得他们的参数量分别为 100000000，400000000，1600000000，测试结果如下：

表 11　全连接层并行时间

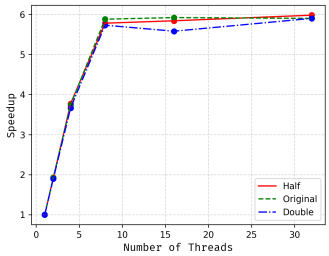
Parameter Amount	100000000	400000000	1600000000
1 Threads	1.93702s	7.87483s	32.1381s
2 Threads	1.01064s	4.07649s	16.8942s
4 Threads	0.513632s	2.10713s	8.78854s
8 Threads	0.335037s	1.33935s	5.61266s
16 Threads	0.331559s	1.32996s	5.76109s
32 Threads	0.323704s	1.33456s	5.44774s

计算加速比和效率如下：

表 12　全连接层并行加速比和效率

Number of Threads		1	2	4	8	16	32
Half	Speedup	1.00	1.92	3.77	5.78	5.84	5.98
	Efficiency	1.00	0.96	0.94	0.72	0.73	0.75
Original	Speedup	1.00	1.93	3.74	5.88	5.92	5.90
	Efficiency	1.00	0.97	0.94	0.74	0.74	0.74
Double	Speedup	1.00	1.90	3.66	5.73	5.58	5.90
	Efficiency	1.00	0.95	0.92	0.72	0.70	0.74

将三种问题规模的加速比和效率汇总并画图得到结果如下：



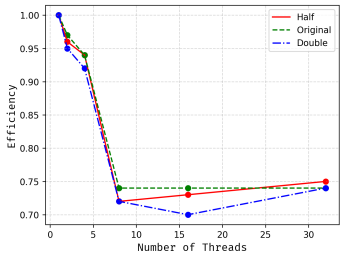


图 11　Fully-Connected Layer Speedup　**图 12　Fully-Connected Layer Efficiency**

5.2 测试二

测试二围绕分类 MNIST 数据集展开，测试代码包含在 test_mnist.cpp 文件中。

5.2.1 LearnMnistDenseTest

LearnMnistDenseTest 网络架构中仅包含全连接层，测试结果为：

表 13　LearnMnistDenseTest 并行结果

Number of Threads	1	2	4	8	16	32
Run Time	21670ms	14298ms	11201ms	11664ms	12814ms	16155ms
Speedup	1.00	1.52	1.93	1.86	1.69	1.34
Efficiency	1.00	0.76	0.48	0.23	0.21	0.17

5.2.2 DeepLearningLoopMnistTest

DeepLearningLoopMnistTest 网络架构中包含循环卷积层、池化层和全连接层，测试结果为：

表 14　DeepLearningLoopMnistTest 并行结果

Number of Threads	1	2	4	8	16	32
Run Time	348425ms	183804ms	114492ms	79916ms	80181ms	82240ms
Speedup	1.00	1.90	3.04	4.36	4.35	4.24
Efficiency	1.00	0.95	0.76	0.55	0.54	0.53

5.2.3 DeepLearning2DMnistTest

DeepLearning2DMnistTest 网络架构中包含 2D 卷积层、池化层和全连接层，测试结果为：

表 15　DeepLearning2DMnistTest 并行结果

Number of Threads	1	2	4	8	16	32
Run Time	124066ms	120510ms	139088ms	163423ms	208693ms	405737ms
Speedup	1.00	1.03	0.89	0.76	0.59	0.31
Efficiency	1.00	0.52	0.22	0.09	0.07	0.04

此时发现随着线程数的增大，加速比非但没增大，甚至减小到了 1 以下，因此设计以下的消融实验：

表 16　DeepLearning2DMnistTest 消融实验

Number of Threads	1	2	4	8	16	32
Run Time	124066ms	120510ms	139088ms	163423ms	208693ms	405737ms
-Convolutional Layer	123909ms	121678ms	140624ms	178091ms	224185ms	415937ms
-Fully-Connected Layer	123938ms	114692ms	113523ms	111728ms	111936ms	111868ms

由此发现当去除全连接层的并行结构后，程序正常运行。但是，全连接层结构在之前的测试结果是正常的。进一步分析发现，全连接层的并行结构是基于外积、转置和点

乘等运算实现的，而在 2D 卷积层中同样引用了这些方法。更重要的是，在测试网络中，2D 卷积层的核大小为 5×1×1，这导致调用多线程的开销大于并行所节省的开销，所以线程数增大时（大于 2），程序的效率反而减小。因此，为了得到最好的并行效果，设置全连接层的线程数为 2（array3d_math.h 文件中），得到的实现结果为：

表 17　DeepLearning2DMnistTest 并行结果（改进后）

Number of Threads	1	2	4	8	16	32
Run Time	123938ms	114692ms	113523ms	111728ms	111936ms	111868ms
Speedup	1.00	1.08	1.09	1.11	1.11	1.11
Efficiency	1.00	0.54	0.27	0.14	0.14	0.14

最后合并以上三个实验，得到的实验结果为：

表 18　Mnist Test 并行结果

Number of Threads	1	2	4	8	16	32
Run Time	495197ms	317473ms	249700ms	213326ms	214559ms	218980ms
Speedup	1.00	1.56	1.98	2.32	2.31	2.26
Efficiency	1.00	0.78	0.50	0.29	0.29	0.28

根据各个实验的加速比和效率画图得到的结果如下：

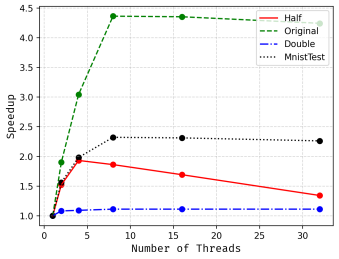


图 13　Mnist Test Speedup

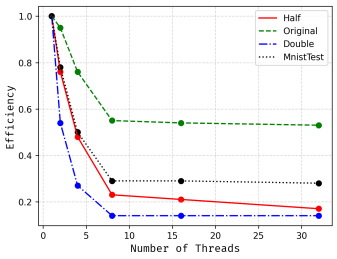


图 14　Mnist Test Efficiency