南开大学
Nankai University

南　开　大　学

计 算 机 学 院

并行程序设计期末实验报告

基于 CNN 的并行工具调研

周辰霏 1712991

年级：2017 级

专业：计算机科学与技术

2020 年 6 月 25 日

摘要

基于卷积神经网络(CNN)中核心计算的两部分——卷积层、池化层的实现问题，调研并行工具的特色及其使用方法，并选取 OpenMP 和 CUDA 并行工具进行并行加速优化。之后通过 Python 中两大著名框架 TensorFlow 和 PyTorch 实现对 minst 手写数字的 CNN 模型训练并对算法 GPU 并行化。通过调研资料以及实验结果对同一层次的并行工具进行其特色及优缺点对比分析。

关键字：CNN；并行工具；CUDA

目录

一、 概述

（一） 研究问题综述

卷积神经网络 (CNN) 如今在深度学习领域大放异彩，视觉图像处理应用、语音识别、自然语言处理等等都少不了它的存在。尤其是在图像处理方面，不同于传统线性训练方式，卷积神经网络不必细究每个样本内容，它通过卷积层的提取压缩，池化层的平均或放大特征得到特征向量，适合于解决此类高复杂度问题，同时整个过程的思路对于使用者而言也不算艰难晦涩。由于整个计算量相对较大，一般实际应用中的 CNN 都是基于并行工具在 GPU 上加速实现的。

卷积神经网络（Convolutional Neural Network, CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，尤其对于大型图像处理有出色表现。CNN 一般由一个或多个卷积层和顶端的全连通层（对应经典的神经网络）组成，同时也包括关联权重和池化层。这一结构使得卷积神经网络能够利用输入数据的二维结构。与其他深度学习结构相比，卷积神经网络在图像和语音识别方面能够给出更好的结果。这一模型也可以使用反向传播算法进行训练。相比较其他深度、前馈神经网络，卷积神经网络需要考量的参数更少，这让它成为一种颇具吸引力的深度学习结构。[1]

CNN 中的卷积层由二维卷积组成，和代数上的卷积稍有区别，可以理解为根据给定过滤器将图像特征矩阵进行特征浓缩提取；而最大池化层类似，是根据过滤器选取最大值进一步寻找更具代表性的特征。

CNN 算法可以较容易的在 CPU 上串行实现，但由于图像输入的二维矩阵一般都量级较大，故卷积层在计算量上的需求很大，一般的多图片数据集训练时间开销很大，不利于算法的推广。所以并行化是其中一种前人工作已被验证可行的方法。[2] [3] [4] 这些研究的并行化之所以成功都是依靠着并行工具实现的。

而当下有各式各样的并行计算辅助工具，包括课程学习中的 OpenMP、CUDA；以及深度学习常用的 TensorFlow 框架等，都是可以帮助我们解决卷积神经网络的并行化的并行化工具，不同的工具由于内在的方法不一样最终的并行优化效果也不同，本文集中讨论各种工具的使用及其各自的特色，并最终通过借助工具实现卷积层、池化层甚至整个 CNN 的并行。并对同一层次工具在使用和效果上进行对比分析。

（二） 卷积层、池化层计算过程

CNN 总的构成可大致分为五个层次，输入层、卷积层、激励层、池化层、全连接层。其中核心计算部分就是卷积层。

卷积层在通常应用尤其是最通用的图像处理中一般指的是二维卷积，即对图像矩阵和滤波矩阵 (卷积内核：一组固定权重) 做内积，如下图就可以很好的表示这一“卷积”过程。最左边是原始输入层，中间是滤波矩阵，通过蓝框内对应做内积——对应位置数字相乘后相加得到右侧“卷积”后的结果。这种计算方法实际是经过输入层处理过后抽象的数据，实际如果转化到图像应用上，则是根据不同的滤波矩阵提取出不同的图像代表信息：如颜色深浅或图形轮廓等等。简要的串行实现将在下文实现，实际上就是通过滤波矩阵滑窗进行矩阵内积计算，其中滤波矩阵可以有多个。

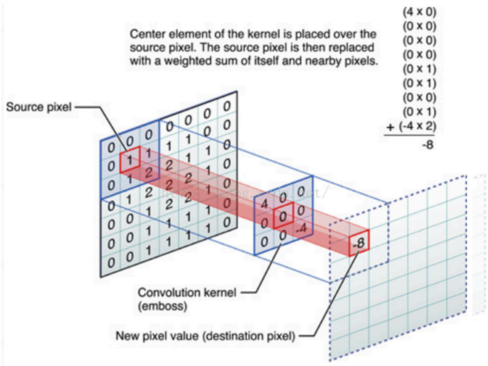

图 1: 卷积计算示意图

激励层的重点是使用 sigmoid 或 ReLU 激励函数模拟神经元反馈传递的过程，是对卷积层的输出通过激励函数的计算进行一个非线性映射。实际应用中使用 ReLU 更多，因为其具有收敛快和求梯度简单的优点。

池化层一般可分为平均池化层和最大池化层，作用是为了使得特征和参数减少并保持某种特征的不变性（旋转、平移），平均池化可以减少领域大小受限造成的估计值方差偏大，而最大池化可以减少卷积层参数误差造成的估计均值的偏移，并保存更多的纹理信息，故一般应用中更多采用最大池化。最大池化运算如下图示，根据滤波矩阵的滑窗对滤波矩阵内的数值取最大值最终合并成结果。

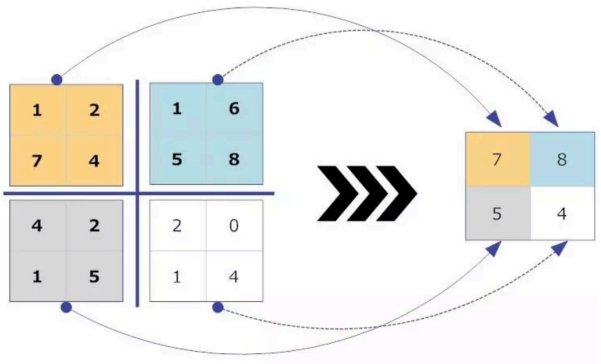

图 2: 最大池化计算示意图

（三） 串行卷积及最大池化实现

卷积层和池化层的计算实际上都是通过矩阵内积实现的，此串行实现内容仅根据原理实现而不考虑优化。

串行

```
//卷积
void conv2(int** filter, int** arr, int** res, int filtern, int arrn)
{
        int temp;
        for(int j=0;j<arrn-filtern+1;j++)
        {
            for(int k=0;k<arrn-filtern+1;k++)
            {
```

```
                temp=0;
                for(int i=0;i<filtern;i++)
                {
                    for(int o=0;o<filtern;o++)
                    {
                        temp+= arr[j+i][k+o]*filter[i][o];
                    }
                }
                res[j][k]=temp;
            }
        }
}
// 最大池化
void max_pooling(const int arrn, const int filtern, int** arr, int** filter,
    int** res) {
    for (int i = 0; i < arrn-filtern+1; ++i) {
        for (int j = 0; j < arrn-filtern+1; ++j) {
            int max_ = (int)-FLT_MAX;
            for (int k = 0; k < filtern; ++k) {
                for (int l = 0; l < filtern; ++l) {
                    if (arr[i + k][j + l] > max_) {
                        max_ = arr[i + k][j + l];
                    }
                }
            }
            res[i][j] = max_;
        }
    }
}
```

（四） 实验环境

- 系统环境：MacOS 10.15 (64 位),Windows 10 家庭版, CentOS 主机
- 编译环境-Python：Colab(Mac)
- 编译环境-C++/OpenMP：gcc 8.1.0 x86_64-posit-seh(Win), Codeblocks17.12(Win)
- 编译环境-CUDA：gcc 7.4.0(Cent),CUDA V10.0.130(Cent)

二、 API 及架构并行工具使用

比较常见且易用的并行 API 有 OpenMP 提供的库 API 部分以及 CUDA 平台提供的内容。本文主要对这两种工具进行调研和使用，并根据最终实验结果以及文献整理总结两种工具各自的优缺点。

（一） 并行接口 OpenMP 并行化

OpenMP（Open Multi-Processing）是一套支持跨平台共享内存方式的多线程并发的编程 API，使用 C,C++ 和 Fortran 语言，可以在大多数的处理器体系和操作系统中运行，包括 Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, 和 Microsoft Windows。包括一套编译器指令、库和一些能够影响运行行为的环境变量。OpenMP 是由 OpenMP Architecture Review Board 牵头提出的，并已被广泛接受的，用于共享内存并行系统的多线程程序设计的一套指导性注释（Compiler Directive）。OpenMP 提供了对并行算法的高层的抽象描述，程序员通过在原始码中加入专用的 pragma 来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。当选择忽略这些 pragma，或者编译器不支持 OpenMP 时，程序又可退化为通常的程序（一般为串行），程式码仍然可以正常运作，只是不能利用多线程来加速程序执行。[5]

OpenMP 使用可移植的可扩展模型，为程序员提供了一个简单灵活的界面，用于开发从标准台式计算机到超级计算机的平台的并行应用程序。OpenMP 是并行化方法多线程的实现，主线程（连续执行一系列指令）派生设定数量的子线程并且由系统为各个线程分配任务。线程并发运行，运行时环境将不同线程分配给不同进程。默认情况下，各个线程独立地执行并行区域的代码。运行时环境分配给每个处理器的线程数取决于使用方法、机器负载和其他因素。线程的数目可以通过环境变量或者代码中的函数来指定。在 C/C++ 中，OpenMP 的函数都声明在头文件 omp.h 中。

OpenMP Architecture Review Board (ARB) 于 1997 年 10 月发布了 OpenMP for Fortran 1.0。次年的 10 月，发布了 C/C++ 的标准。2000 年，发布了 Fortran 语言的 2.0 版本，并于 2002 年发布了 C/C++ 语言的 2.0 版本。2005 年发布了包含 Fortran 和 C/C++ 的 2.5 版本。2008 年 5 月发布的 3.0 版。3.0 中的新功能包括任务 (tasks) 和任务结构 (task construct) 的概念。这些新功能总结在 OpenMP3.0 规范的附录 F 中。OpenMP 规范的 3.1 版则于 2011 年 7 月 9 日发布。4.0 版本在 2013 年 7 月发布，它增加或改进了以下功能：支持加速器，原子性，错误处理，线程关联，任务扩展，减少用户定义的 SIMD 支持和 Fortran 2003 的支持。

OpenMP 基于拥有共享内存的多线程并行编程模式，其特点是简单易用，且灵活依存于共享内存系统，统一的代码使得 OpenMP 可以很好的适用于串并行程序，且可以应用多种加速器，如 GPU 和 FPGA 等。

作为一个简便易用多线程并行框架，虽然 OpenMP 在硬件上的制约要稍大，但在软件层面有很强的普适性，此外 OpenMP 的使用也很方便快捷，只需要在原有 C 代码上加入 OpenMP 的 Directives(本文主要使用 for) 及 clause(本文中的 num_threads) 即可。下述为卷积层和池化层的 OpenMP 实现。

在编写使用 OpenMP 的程序时，则需要先 include OpenMP 的头文件：omp.h，GCC 编译器是内置了 OpenMP 的，所以只需要在编译的时候进行参数设定“-fopenmp”即可使用 OpenMP，在本问题中由于耗时大的部分是 for 循环中内容，所以只需要使用 # pragma 语句对 for 循环并行化即可。经过试验，直接 for 并行化的效果比 schedule 略好，故代码采用直接运用 omp for 的部分。

OpenMP 并行

```
//其余部分维持串行朴素算法原样，只对卷积层和池化层函数部分进行omp处理
//对卷积函数和池化函数进行OpenMP并行化，选取相对结果最优的一种方式
void conv2(int** filter, int** arr, int** res, int filtern, int arrn)
{
        int temp;
        #pragma omp parallel for num_threads(NUM_THREADS)
        for(int j=0;j<arrn-filtern+1;j++)
        {
            for(int k=0;k<arrn-filtern+1;k++)
```

```
            {
                temp=0;
                for(int i=0;i<filtern;i++)
                {
                    for(int o=0;o<filtern;o++)
                    {
                        temp+= arr[j+i][k+o]*filter[i][o];
                    }
                }
                res[j][k]=temp;
            }
        }
}
void max_pooling(const int arrn, const int filtern, int** arr, int** filter,
    int** res) {
    #pragma omp parallel for num_threads(NUM_THREADS)
    for (int i = 0; i < arrn-filtern+1; ++i) {
        for (int j = 0; j < arrn-filtern+1; ++j) {
            int max_ = (int)-FLT_MAX;
            for (int k = 0; k < filtern; ++k) {
                for (int l = 0; l < filtern; ++l) {
                    if (arr[i + k][j + l] > max_) {
                        max_ = arr[i + k][j + l];
                    }
                }
            }
            res[i][j] = max_;
        }
    }
}
```

如下表为处理结果，实验机器为四核故并行线程数选择 4 线程。N 是待处理矩阵大小，n 是 filter 滤波矩阵大小，OpenMP 并行化和串行算法处理方式相同，都为随机生成大小为 N×N 的矩阵，经过卷积层函数处理后将结果传入进行池化函数处理，最后得出结果，进行结果比对成功后再进行时间测算及结果输出。这一过程的耗时与 N 和 n 的大小正相关，对普通 for 并行、dynamic 等都进行测试后发现直接进行线程数设置的 for 并行加速效果最佳，可以看出除了 N 较小的个别测试数据外，OpenMP 的测试结果都很不错。池化层的效果在 1000-10000 区段明显强于卷积层可能是因为卷积计算复杂，而池化只是进行 if 的分支比较并赋值，OpenMP 的并行加速更有利所致。

将以上测试结果绘制成如下图，可以看出 OpenMP 对朴素算法加速效果显著，加速比整体上都处于良好及以上的水准。原算法复杂度由 $O(n^3)$ 变为了 $O(\frac{n^3}{4})$，最终结果与预期一致加速比大致在 2-4 之间。最后再对不同大小的滤波矩阵进行测算，跟 n=4 时的 filter 相比，运行时间几乎和 n 的大小成正比。

N/n\Algo	naive-conv	naive-pool	omp-conv	omp-pool
64/2	0.0167	0.01255	0.04142	0.03799
64/4	0.03599	0.0394	0.0458	0.0421
128/4	0.1497	0.1715	0.0945	0.0888
256/4	0.6535	0.6972	0.3290	0.2736
512/4	2.538	2.9656	1.3760	1.1148
1024/4	10.4311	12.0539	4.675	3.047
1024/16	141.25	160.68	50.641	45.813
2048/4	40.9796	48.0939	18.1401	12.2016
2048/16	571.437	660.566	196.279	189.004
4096/4	162.656	194.651	74.5763	51.9689
4096/16	2432.07	2702.85	932.882	1045.86
8192/4	655.638	775.054	292.893	206.928
8192/32	44470.5	48486.2	19594	16661.9
16384/4	2682.6	3111.3	1370.55	1461.9
16384/64	706344	637576	325458	290727

表 1: 性能测试结果 (4 线程)(单位:ms)

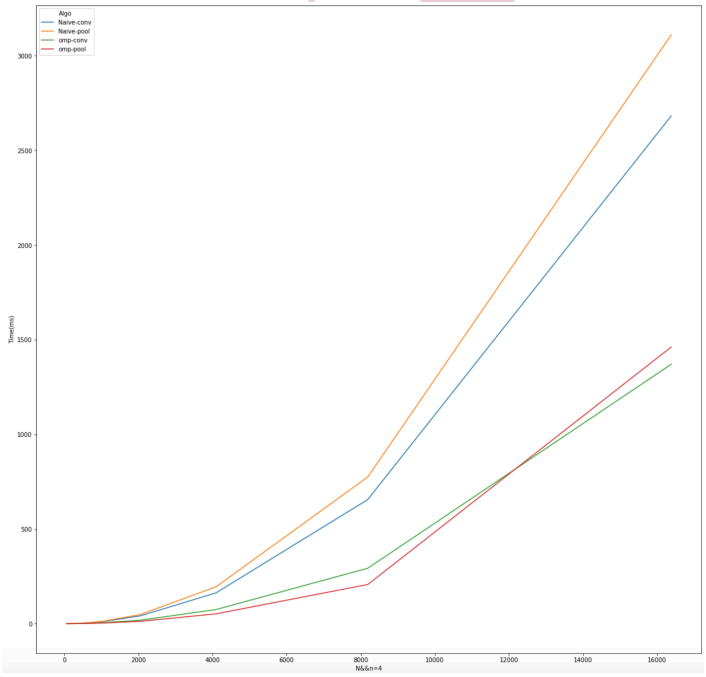

图 3: OpenMP 并行结果

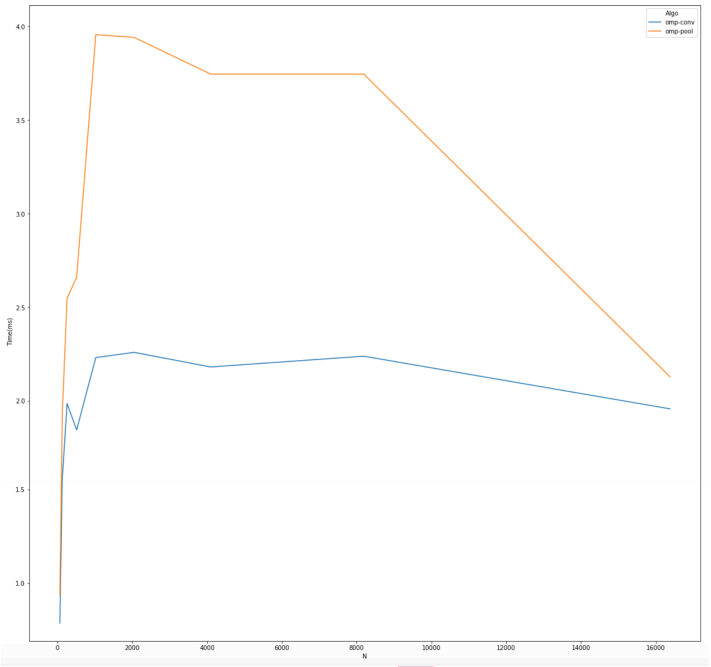

图 4: OpenMP 加速比

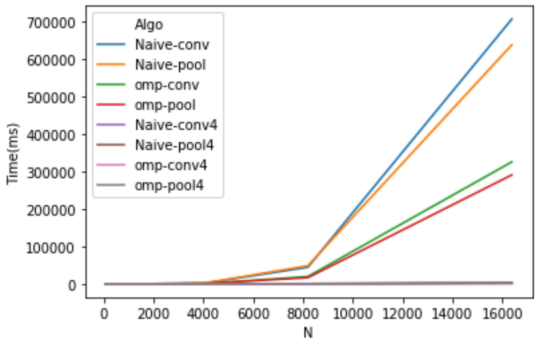

图 5: OpenMP 滤波矩阵大小对比

由上述可以得出，OpenMP 本身就可以当做一个好用的并行线程库，所以在使用上非常的简单，并且相对灵活，只需要在需要加速并行的函数主体部分 (关注和确定好可以并行化的代码块且注意避免数据依赖) 加上 # pragma 语句即可，由语句对执行进行设置再分配给线程完成任务即可。由于 OpenMP 提供的这种对于并行描述的高层抽象降低了并行编程的难度和复杂度，程序员就可以把更多的精力投入到并行算法本身，而非其具体实现细节。对基于数据分集的多线程程序设计，OpenMP 是一个很好的选择，此外 OpenMP 也具有着更强的灵活性，可以较容易的适应不同的并行系统配置，OpenMP 帮助程序员解决线程粒度和负载平衡这两个传统多线程编程难题。其缺点和不足也是显而易见的，OpenMP 是用于共享内存并行系统 (单机) 的多处理器程序，所以它不能在非共享内存系统 (如计算机集群) 上使用，相对的，一般在集群上使用 MPI。

（二） 并行运算架构 CUDA 并行化

CUDA(统一计算架构) 是有 NVIDIA 推出的一个并行运算平台及 API 模型。它借助支持 CUDA 的 GPU 实现并行加速。CUDA 是一个直接联通到 GPU 虚拟指令集的和软件层。GPU 定位于密集数据以及并行数据计算，所以 CUDA 非常适合含有大量并行计算的工作任务。CUDA

广泛应用于游戏、图像仿真、科学计算等诸多领域。结合 GPU 的基本特性已经 CUDA 架构提供的便利的开发环境，CUDA 在深度学习、数据挖掘上也是很有前景的，因为这些工作都是数据密集型的，这和 CUDA 的优势不谋而合。CUDA 的特点在于不需要数据排布整齐的存取、统一虚存、共享内存以及对 GPU 的控制。[6]

CUDA 这个架构实际上是在 CPU 和 GPU 的运算中充当一个翻译的角色，它负责把 CPU 的计算指令翻译成 GPU 的计算指令，同时还负责显存和计算机系统内存中数据的交换操作，而 CUDA 程序本身也还是要靠 CPU 才能顺利执行。开发人员使用一种全新的编程模式将并行数据映射、安排到 GPU 中。CUDA 程序则把要处理的数据细分成更小的区块，然后并行的执行它们。这种编程模式允许开发人员只需对 GPU 编程一次，无论是包含多处理器的 GPU 产品或是低成本、处理器数量较少的产品。当 GPU 计算程序运行的时候，开发者只是需要在主 CPU 上运行程序，CUDA 驱动会自动在 GPU 上载入和执行程序。host 程序可以通过高速的 PCI Express 总线与 GPU 进行信息交互。数据的传输、GPU 运算功能的启动以及其它一些 CPU 和 GPU 交互都可以通过调用专门的运行时驱动中的专门操作来完成。这些高级操作把程序员从手动管理 GPU 运算资源中解放出来。

CUDA 编程模型需要 CPU 和 GPU 协同工作。在 CUDA 中，用 host 指代 CPU 及其内存、device 指代 GPU 及其内存。CUDA 程序中既包含 host 程序，又包含 device 程序，它们分别在 CPU 和 GPU 上运行。同时，host 与 device 之间可以进行通信，这样它们之间可以进行数据拷贝。典型的 CUDA 程序的执行流程如下：

1. 分配 host 内存，并进行数据初始化；
2. 分配 device 内存，并从 host 将数据拷贝到 device 上；
3. 调用 CUDA 的核函数在 device 上完成指定的运算；
4. 将 device 上的运算结果拷贝到 host 上；
5. 释放 device 和 host 上分配的内存。

其中最重要的一个过程是调用 CUDA 的核函数来执行并行计算，kernel 是在 device 上线程中并行执行的函数，核函数用 __global__ 符号声明，在调用时需要用 «<grid, block»> 来指定 kernel 要执行的线程数量，在 CUDA 中，每一个线程都要执行核函数，并且每个线程会分配一个唯一的线程号 thread ID，这个 ID 值可以通过核函数的内置变量 threadIdx 来获得。由于 CUDA 是异构模型，所以需要通过函数类型限定词开区别 host 和 device 上的函数，主要有 __global__、__device__、__host__ 三种，__global__ 是在 device 上执行，从 host 中调用（一些特定的 GPU 也可以从 device 上调用），返回类型必须是 void，不支持可变参数参数，不能成为类成员函数，用 __global__ 定义的 kernel 是异步的，这意味着 host 不会等待 kernel 执行完就执行下一步。kernel 在 device 上执行时实际上是启动很多线程，一个 kernel 所启动的所有线程称为一个网格（grid），同一个网格上的线程共享相同的全局内存空间，grid 是线程结构的第一层次，而网格又可以分为很多线程块（block），一个线程块里面包含很多线程，这是第二个层次。grid 和 block 都是定义为 dim3 类型的变量，dim3 可以看成是包含三个无符号整数（x，y，z）成员的结构体变量，在定义时，缺省值初始化为 1。因此 grid 和 block 可以灵活地定义为 1-dim，2-dim 以及 3-dim 结构，kernel 在调用时也必须通过执行配置 «<grid, block»> 来指定 kernel 所使用的线程数及结构。grid 和 block 的结构如下图所示。这样来看一个线程需要两个内置的坐标变量（blockIdx，threadIdx）来进行唯一标识，它们都是 dim3 类型变量，其中 blockIdx 指明线程所在 grid 中的位置，而 threaIdx 指明线程所在 block 中的位置。kernel 的这种线程组织结构天然适合 vector,matrix 等运算。

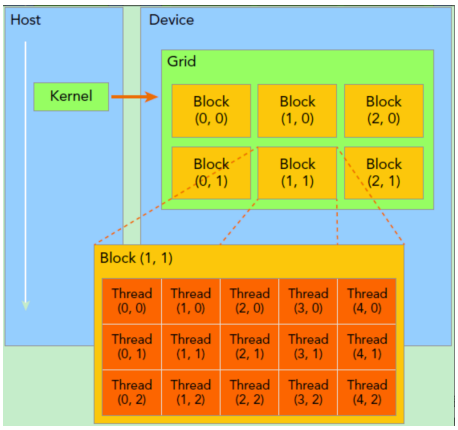

图 6: Kernel 上的两层线程组织结构（2-dim）

CUDA 内存模型如下图所示，可以看到每个线程有自己的私有本地内存（Local Memory），且每个线程块有包含共享内存（Shared Memory）可以被线程块中所有线程共享，其生命周期与线程块一致。此外，所有的线程都可以访问全局内存（Global Memory）。还可以访问一些只读内存块：常量内存（Constant Memory）和纹理内存（Texture Memory）。

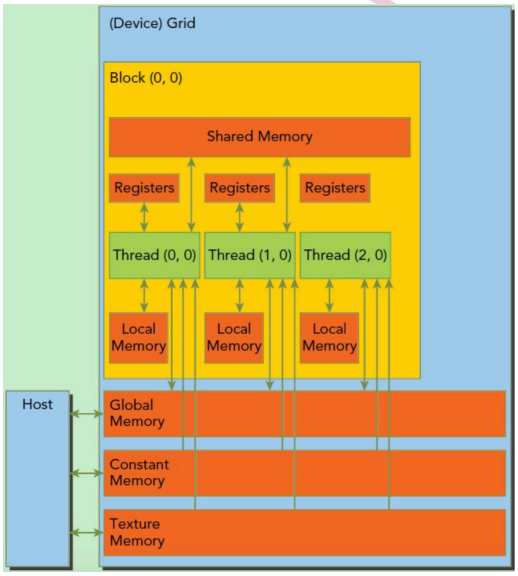

图 7: CUDA 内存模型

在学习完 NVIDIA 提供的入门课程后，使用最简单的步骤实现 CUDA 的并行，即先将数据转化成便于 CUDA 工作的形式，然后再对主要函数进行核函数处理，即找出 index 根据 block 和 thread 定位并计算该位置的数值，之后再对结果进行还原处理即可，同时主函数中使用 «<grid, block»> 来调用函数指定要执行的线程数，最后进行辅助内存上的管理即可。

CUDA 并行

```
1 // 卷积
2 __global__ void conv2d(float *arr, float *res, float *fil, int filn, int n) {
3         int Col = blockIdx.x*blockDim.x + threadIdx.x;
4         int Row = blockIdx.y*blockDim.y + threadIdx.y;
5         if (Row < n&&Col < n) {// 根据block和线程确定位置
```

```
                float val = 0;
                int startCol = Col - filn / 2;
                int startRow = Row - filn / 2;
                for (int i = 0; i < filn; i++)//卷积
                {
                        for (int j = 0; j < filn; j++)
                        {
                                int curRow = startRow + i;
                                int curCol = startCol + j;
                                if (curRow > -1 && curRow<n&&curCol>-1 &&
                                    curCol < n)
                                {
                                        val += fil[i*filn + j] * arr[curRow*n
                                             + curCol];
                                }
                        }
                }
                res[Row*n + Col] = val;
        }
}
//最大池化
__global__ void pooling(float *arr, float *res, float *fil, int filn, int n)
    {
        int Col = blockIdx.x*blockDim.x + threadIdx.x;
        int Row = blockIdx.y*blockDim.y + threadIdx.y;
        if (Row < n&&Col < n) {//根据block和线程确定位置
        float max_ = -FLT_MAX;
                int startCol = Col - filn / 2;
                int startRow = Row - filn / 2;
                for (int i = 0; i < filn; i++)
                {
                        for (int j = 0; j < filn; j++)
                        {
                                int curRow = startRow + i;
                                int curCol = startCol + j;
                                if (curRow > -1 && curRow<n&&curCol>-1 &&
                                    curCol < n)
                                {
                                        if(arr[curRow*n + curCol]>max_) max_=
                                            arr[curRow*n + curCol];
                                }
                        }
                }
                res[Row*n + Col] = max_;
        }
}
//主函数部分
//将矩阵转化为一维数列，便于CUDA进行index划分定位和操作
```

```
int arr_size_1D = arr_size*arr_size;
int filter_size_1D = filter_size*filter_size;
float *arr_1D = (float*)malloc(arr_size_1D * sizeof(float));
float *arr_1D_Cpu = (float*)malloc(arr_size_1D * sizeof(float));
float *res_1D = (float*)malloc(arr_size_1D * sizeof(float));
float *filter1D = (float*)malloc(filter_size_1D * sizeof(float));
for (int i = 0; i<arr_size; i++) {
        for (int j = 0; j < arr_size; j++) {
                arr_1D[i*arr_size + j] = arr[i][j] * 1.0;
                arr_1D_Cpu[i*arr_size + j] = res[i][j] * 1.0;
        }
}
for (int i = 0; i<filter_size; i++) {
        for (int j = 0; j < filter_size; j++) {
                filter1D[i*filter_size + j] = filter[i][j] * 1.0;
        }
}
//分配CUDA内存
float *inD, *outD, *filD;
cudaMalloc((void**)&inD, sizeof(float)*arr_size_1D);
cudaMalloc((void**)&outD, sizeof(float)*arr_size_1D);
cudaMalloc((void**)&filD, sizeof(float*)*filter_size_1D);
//copy
cudaMemcpy(inD, arr_1D, sizeof(float)*arr_size_1D, cudaMemcpyHostToDevice);
cudaMemcpy(outD, arr_1D, sizeof(float)*arr_size_1D, cudaMemcpyHostToDevice);
cudaMemcpy(filD, filter1D, sizeof(float)*filter_size_1D,
    cudaMemcpyHostToDevice);
//运行及计时
int threadPerBlockX = 16;
int threadPerBlockY = 16;
//根据输入数据调整grid和block范围
dim3 grid((arr_size - 1) / threadPerBlockX + 1,(arr_size - 1) /
    threadPerBlockY + 1,1);
dim3 block(threadPerBlockX, threadPerBlockY);
gettimeofday(&starttime, NULL);//精确到微秒
conv2d <<<grid, block >>>(inD, outD, filD, filter_size, arr_size);
gettimeofday(&endtime, NULL);
timersub(&endtime, &starttime, &diff);
timecost= diff.tv_sec + (1.0 * diff.tv_usec)/1000;
cout<<"conv Cost: "<<timecost<<"ms."<<endl;

gettimeofday(&starttime, NULL);//精确到微秒
pooling <<<grid, block >>>(inD, outD, filD, filter_size, arr_size);
gettimeofday(&endtime, NULL);
timersub(&endtime, &starttime, &diff);
timecost= diff.tv_sec + (1.0 * diff.tv_usec)/1000;
cout<<"pool Cost: "<<timecost<<"ms."<<endl;

```

```
95 //释放内存，拷贝结果
96 cudaMemcpy(res_1D, outD, sizeof(float)*arr_size_1D, cudaMemcpyDeviceToHost);
97 cudaFree(inD);
98 cudaFree(outD);
99 cudaFree(filD);
```

最终经过测试，花费十几分钟的 16384/64 卷积层经过 cuda 加速后运行时间可以得到将近几十到一百倍的加速, 而 1024/16 的卷积层的加速比仅为 4.28, 可以看出运行时间有了很大的改进，且改进效果与 grid 和 bock 的设置有关。但 CUDA 较大测试数据与偏小测试数据的时间性能差距不大，推测原因是测试数据还不够大，即使 N=16384 仍属于小数据范围，但由于串行算法再设更大的输入值需要耗费几何倍数的时间计算，所以没有选取更大的测试数据。

（三） 优缺点对比

如果仅凭我自己浅薄的实验内容和结果来看，OpenMP 在易用性上显然超过大多数工具，其一是 OpenMP 被 GCC 编译器内置，不需要额外配置环境；其二是只要包含头文件及编译参数并在函数框架中使用 Directives 和 Clause 即可。但由于我的 OpenMP 只是在本机 CPU 上运行所以加速比受到 CPU 核数的限制，整体加速效果达到预期但没有显著的提升，如果可以利用 GPU 加速的话效果也许会更好，但是由于从数据划分上不如 CUDA 做的更优所以可能整体效果仍不如 CUDA 表现好。而 CUDA 在加速效果上确实由于其优越的数据划分更适合矩阵一类的计算，所以加速效果显而易见的非常好，且由于是在 GPU 上运行的，所以更是超出了想象。但是 CUDA 本身的环境配置需要一定时间，上手学习也不如 OpenMP 那么轻松。包括代码并行化的编写都要更加复杂，而且数据要重新进行排布分配才可以更好的适用于 CUDA 环境，不过在这些部分上艰难的前提下得到了优秀的并行结果。

此外，根据收集到的资料，OpenMP 的主要优点是更易使用、gcc 等普遍支持、对原串行代码改动较小可以保持代码原貌并易于维护；缺点就如上面所提及的所有线程共享内存空间，对硬件的约束比较大，而且主要针对的是循环上的并行化，不够广泛。CUDA 是既有平台支持也有 API 可使用、对代码改动较大、GPU 加速效果好（尤其是矩阵相关运算）；缺点直观明显且无法克服，只支持 NVIDIA 显卡。本质上的区别来看，OpenMP 是 CPU 上线程的并行，而 CUDA 是 GPU 线程上的并行，所以不能直接对二者进行比较，如果按照 CPU 普遍四八核配置以及中端显卡计算能力 2.0 来看，CUDA 在运行效率上可能更有优势。二者的效率不止和其本身构成相关还和 CPU、GPU 的水平以及程序员的水平挂钩。从目前来看，GPU 加速会更多倾向于 CUDA，许多深度学习框架也都支持 CUDA 加速，泛用性很好；OpenMP 多为单机使用，加速效果也很好，且编程简单方便易用。

三、 Python 框架工具应用

Python 在机器学习领域可谓独领风骚，说起机器学习，绝大多数人都因为 Python 的便于使用以及丰富的包和框架支持而选择了它，加之 CPython 解释器可以大幅增加运行效率。

在此主要以 TensorFlow 和 PyTorch 为例进行 Python 框架工具在并行计算开发上的调研和特色介绍。

（一） Tensorflow 并行工具

TensorFlow 是一个免费的开源软件库，用于跨各种任务的数据流和可区分编程。它是一个符号数学库，也用于机器学习应用程序，例如神经网络。TensorFlow 主要包括三种不同的并行策略：

数据并行，模型并行和模型计算流水线。TensorFlow 可以在多个 CPU 和 GPU 上运行（具有可选的 CUDA 和 SYCL 扩展，用于在图形处理单元上进行通用计算）。[7]

TensorFlow 全系统适用，且其灵活的架构可以在 CPU、GPU、TPU 等多种底层平台上运行。Tensorflow 的一个特色就是分布式计算，分布式 Tensorflow 是由高性能的 gRPC 框架作为底层技术来支持的。这是一个通信框架 gRPC(google remote procedure call)，是一个高性能、跨平台的 RPC 框架。RPC 协议，即远程过程调用协议，是指通过网络从远程计算机程序上请求服务。TensorFlow 的并行分为操作间并行和操作内并行，二者同时发生，并且都是通过线程池技术实现的。操作间并行是指 node 与 node 之间可以并行执行，操作内并行是指每个 node 内的运算可以并行计算。对于 CNN，每一层的卷积和 pooling 操作都可以完全并行。

本文主要调研 TensorFlow 的多 GPU 并行计算，TensorFlow 程序通过 tf.device 函数来指定运行每一个操作的设备，这个设备可以是 CPU 或 GPU，也可以是某一台远程的服务器，设备名称：CPU:/cup:0,GPU: /gpu:n。在配置好 GPU 的环境中 TF 会优先选择 GPU。并不是所有的操作都可以放在 GPU 上，如果将无法放在 GPU 上的操作指定到 GPU 上，程序将会报错，并且不同版本的 TF 对 GPU 的支持也是不同的。TensorFlow 在并行化上的使用方法

而在我们主要讨论的卷积层和最大池化层问题是属于深度学习范畴的，TensorFlow 深度学习方法的并行主要有同步 (数据并行) 和异步 (模型并行) 两种模式，同步模式可以避免更新不同步的问题，但是同步模式低于异步模式，如果设备的运行速率不一致，那么每一轮训练都需要等待最慢的设备结束才能开始更新参数。同步模式几乎适用于所有深度学习模型。使用数据并行时最好 GPU 的型号，速度最好一致，这样效率最高，而异步的数据并行，则不等待所有 GPU 都完成一次训练，而是哪个 GPU 完成了训练，就立即将梯度更新到共享的模型参数中，通常来说数据并行比异步并行的模式收敛速度更快，模型的精度更高。CNN 一般是同步并行。

TensorFlow 的深度学习并行化其实都已经内置在框架中了，所以只要在调用相应深度学习方法的同时将并行化配置加入 session 中让客户端与 TF 系统交互并实现不同的并行效果。

卷积层在 TF 中对应的 API 为 tf.layers.conv2d(), 池化层在 TF 中对应的 API 为 tf.layers.max_pooling2d(), 在对 API 需要的参数进行设定后再通过 tf.ConfigProto() 来配置 session 的运行参数，其中参数 device_count，使用字典分配可用的 GPU 设备号和 CPU 设备号，用以限制 CPU 或 GPU 的数目，同时保证各个 device 之间的变量共享。

首先实现一个最基本的 CNN 模型，全部使用 CPU 运行，在 Colab 上对 minst 进行训练，最终耗时 103min，得到了 0.9916 的准确率 (迭代 5 次结束得到 98% 的准确率串行和并行耗时分别为 627s 和 121s)。然后加入如下内容对 CNN 进行 GPU 并行化处理，其他设置保持原样可以得到如下结果：准确率 0.9923，耗时 45min。为了得到更明显的对比，总迭代次数为 20000 次，故而两者时间都较长，但并行 GPU 加速比为 2.29 则可以看出明显的性能提升。

TensorFlow 并行 GPU

```
# 基本CNN实现与课程干系不大直接使用API生成
import input_data   #21: 13-
import time
mnist = input_data.read_data_sets('./MNIST_data', one_hot=True)
sess=tf.compat.v1.InteractiveSession()
# 初始化CNN的权重
def weight_variable(shape):
    initial = tf.truncated_normal(shape=shape, stddev=0.1)
    return tf.Variable(initial)
# 初始化CNN的偏置
def bias_variable(shape):
```

```
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
# 卷积层定义
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
# 池化层定义
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
        padding='SAME')
# 图1*784的形式转换成原始28*28  -1代表样本数量不固定，1代表颜色通道数量
x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])
x_image = tf.reshape(x, [-1, 28, 28, 1])
# conv1
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
# conv2
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
#fc
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
# dropout
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
# softmax层概率输出
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv),
    reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
# 评定指标
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
init = tf.global_variables_initializer()
sess.run(init)
start = time.time()
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i % 100 == 0:
        train_accuracy = sess.run(accuracy, feed_dict={x: batch[0], y_: batch
```

```
            [1], keep_prob: 0.1})
        print('step %d, training accuracy %g' % (i, train_accuracy))
    sess.run(train_step, feed_dict={x: batch[0], y_: batch[1], keep_prob:
        0.5})
end = time.time()
print('run time',end - start)
print('test accuracy %g' % sess.run(accuracy, feed_dict={x: mnist.test.images
    , y_: mnist.test.labels, keep_prob: 1.0}))
# 以上为tensorflow框架下最简单的CNN实现

# 并行化使用GPU加速，在session初始化之前使用device对运行设备进行定义并配置给
    sesseion之后运行
with tf.device("/gpu:0"): # 设置device使用gpu0
    i = tf.Variable(3)
config = tf.ConfigProto()
config.allow_soft_placement = True
sess = tf.Session(config=config) # 用config配置
sess.run(i.initializer)
```

(二) PyTorch 并行工具

PyTorch 是基于 Torch 库的开源机器学习库，使用 CPU 和 GPU 优化的深度学习张量库，能在强大 GPU 加速基础上实现张量和动态神经网络，多用于计算机视觉和自然语言处理。PyTorch 和 TensorFlow 一样都是非常底层的框架。它是一个能提供最大的灵活性和速度的深度学习研究平台。[9]PyTorch 主要由 Facebook 于 2017 年 1 月成立的 AI 开发研究实验室（FAIR）开发。Facebook 同时运营 PyTorch 和 Caffe2，但由于根本模型定义二者并不兼容。2018 年 3 月 Caffe2 被合并入 Pytorch 框架中。PyTorch 主要有两大特征：类似于 NumPy 的张量计算，可使用 GPU 加速；基于带自动微分系统的深度神经网络。PyTorch 就和 TensorFlow 一样可以帮助程序员轻松的解决大部分 ML 场景问题，只有将 PyTorch 应用于工业生产中时，才需要将其格式转换为 Caffe 以满足工业生产的需求。

PyTorch 的使用和 TensorFlow 一样安装相应包即可，并通过 API 使用即可。

同样首先实现借助 PyTorch 的 CNN 识别 minst 再对其进行并行化。由于 GPU 版本的 PyTorch 自带 CUDA，所以可以省去配置步骤，直接调用即可。并行化的步骤是数据并行——将数据设置为 CUDA 可识别的格式、模型并行——将模型设置为 CUDA 格式、最后将输出结果转化为 Python 可识别的格式即可。

PyTorch 并行 CNN

```
# 同样先实现一个基本的CNN，主要内容还是调包
import torch
import torchvision
import torch.utils.data as Data
import torch.nn as nn
from matplotlib import pyplot as plt
import time
import numpy as np

BATCH_SIZE = 50
```

```
LEARNING_RATE = 0.001
EPOCH_NUM = 3
# pytorch就可以直接调包使用minst了
train_data = torchvision.datasets.MNIST(root='./mnist', train=True,
    transform=torchvision.transforms.ToTensor(), download=True )

test_data = torchvision.datasets.MNIST(root='./mnist', train=False)

train_loader = Data.DataLoader(
    dataset=train_data,
    batch_size=BATCH_SIZE,
    shuffle=True  # 设置是否随机顺序
)

test_x = torch.unsqueeze(test_data.test_data, dim=1).type(torch.FloatTensor)
    [:2000] / 255.
test_y = test_data.test_labels[:2000]
# CNN
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            # 卷积层
            nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5,
                stride=1, padding=2 ),
            # 激活函数
            nn.ReLU(),
            # 池化层
            nn.MaxPool2d(kernel_size=2)
        )
        # 卷积2
        self.conv2 = nn.Sequential(
            nn.Conv2d(
                in_channels=16, # 上一层输出channel是16
                out_channels=32,kernel_size=5,stride=1,padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        # 线性输出层，原图大小28，经两次卷积核为2的池化，最后图片大小为7*7，
            且有32个通道，所以共有32 * 7 * 7个输出，线性映射到10个分类上
        self.out = nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        conv1_res = self.conv1(x)
        conv2_res = self.conv2(conv1_res)
        out = conv2_res.view(conv2_res.size(0), -1)
        output = self.out(out)
        return output
```

```
55
56
57  cnn = CNN()
58  print(cnn)
59
60  # adam optimizer
61  optimizer = torch.optim.Adam(cnn.parameters(), lr=LEARNING_RATE)
62  loss_func = nn.CrossEntropyLoss()
63
64  step_list = []
65  loss_list = []
66  accuracy_list = []
67  counter = 0
68
69  t1 = time.time()
70  for epoch in range(EPOCH_NUM):
71      for step, (b_x, b_y) in enumerate(train_loader):
72          predict_y = cnn(b_x)
73          loss = loss_func(predict_y, b_y)
74          optimizer.zero_grad()
75          loss.backward()
76          optimizer.step()
77          counter += 1
78
79          # 画图呈现loss和精确率
80          if step % 25 == 0:
81              test_output = cnn(test_x)
82              pred_y = torch.max(test_output, 1)[1].data.squeeze().numpy()
83              accuracy = float(np.sum((pred_y==test_y))) / float(test_y.size(0)
                    )
84              print('epoch:', epoch, '|step:%4d' % step, '|loss:%6f' % loss.
                    data.numpy(), '|accuracy:%4f' % accuracy)
85
86              step_list.append(counter)
87              loss_list.append(loss.data.numpy())
88              accuracy_list.append(accuracy)
89
90              plt.cla()
91              plt.plot(step_list, loss_list, c='red', label='loss')
92              plt.plot(step_list, accuracy_list, c='blue', label='accuracy')
93              plt.legend(loc='best')
94              plt.pause(0.1)
95  t2 = time.time()
96  print(t2 - t1)
97
98  # 修改1 将数据改成CUDA可识别的格式
99  test_x = torch.unsqueeze(test_data.test_data, dim=1).type(torch.FloatTensor)
        [:2000].cuda() / 255.
```

```
test_y = test_data.test_labels[:2000].cuda()
# 修改2 给网络设置cuda
cnn.cuda()
# 修改3 将数据转换成CUDA可识别的格式
predict_y = cnn(b_x.cuda())
loss = loss_func(predict_y, b_y.cuda())
# 修改4 将数据转换成CUDA可识别的格式
pred_y = torch.max(test_output, 1)[1].data.squeeze()
accuracy = float(torch.sum(pred_y == test_y)) / float(test_y.size(0))
# 修改5 将数据转换成Python可识别的格式
print('epoch:', epoch, '|step:%4d' % step, '|loss:%6f' % loss.data.cpu(), '|
    accuracy:%4f' % accuracy)
```

普通的 CNN 迭代结果如下，训练时间约为 353s，最终准确率可达，如下图所示；而进行并行优化的 CNN 在速度上确实快了一些，GPU 运行结果仅需要 47s，准确率同样可达到 0.9845。

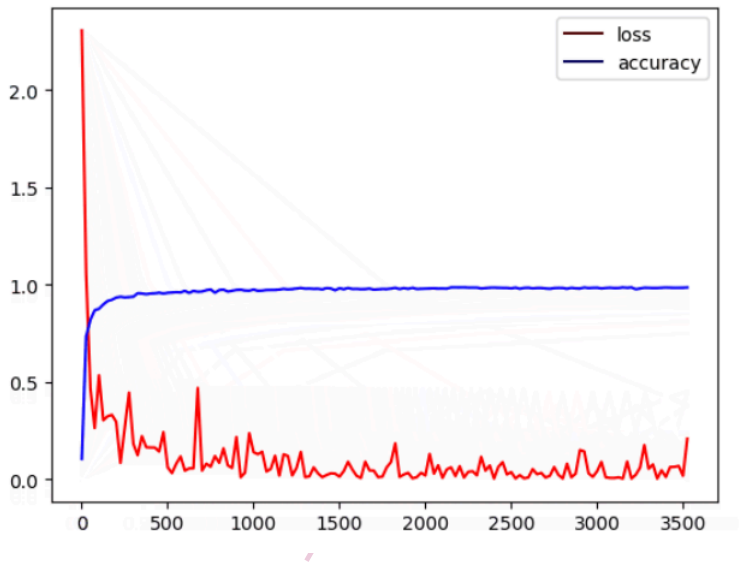

图 8: pytorch 结果

(三) 对比

从我以上浅薄的实验来看，串行实现的调包基本没有什么大的差别，差距主要在并行化上，PyTorch 对 CUDA 的支持和使用明显要更好一些，最终效果二者相差不大，PyTorch 光从运行时间上来看稍胜一筹，且代码也更加简洁。Pytorch 的特色在解决我的问题中并没能体现出来，因为 PyTorch 本来是可以直接调用 DataParallel 进行模型并行的，但数据之间的转换不可以直接进行，需要手动转换，所以直接使用了 PyTorch 中对 CUDA 的支持，也算是从另一方面上体现出了 PyTorch 比 TensorFlow 更简便易用的特点。

根据查阅的资料来看，二者都是在张量上运行，但两个框架中的定义是不同的。在 TensorFlow 中，在跑模型之前会静态的定义图形。和外界的所有联系都是通过 tf.Session 对象和 tf.Placeholder，它们都是会在模型运行被外部数据取代的张量。而 PyTorch 就是更动态的定义，写 PyTorch 就和写 Python 一样差别不大。TensorFlow 对动态输入的支持有限，这在 RNN 这种序列会有变化的网络上就会有诸多不便。在并行方面上，PyTorch 不同于 TensorFlow 的最大特性之一就是声明式数

据并行：可以通过 torch.nn.DataParellel 封装任何模型，而且模型能在批处理维度上实现并行。这样就可以毫不费力的使用多个 GPU。TensorFlow 也能实现类似的数据并行，但是从实现代码来看 TensorFlow 要繁琐的多。整体上 Pytorch 似乎更像是一个框架，而 TensorFlow 像是一个巨大的库。但是如果大规模分布式模型训练，整体上还是 TensorFlow 支持更好一些，Pytorch 在应用上的支持不是很好。而 TensorFlow 在调试方面不是很友好，同样 Pytorch 的可视化要借助第三方库。[8]

四、 总结

本次实验对基本常用的 CPU、GPU 并行工具中的典型——OpenMP 和 CUDA 进行了调研和实践，也得到了不错的效果，最终根据实验过程以及资料查阅对工具在并行的便捷、效果以及本质实现上进行对比。最后借助两个深度学习框架对 CNN 的全部内容进行实现和并行化。再一次比较两个框架的利弊。总的来说，以我目前的上不了工业和学术层面的水平来看，OpenMP 和 PyTorch 更适合我入并行的门，但是如果还需要更复杂的配置、更优的性能提升以及更自定义的运行配置，还需要更多的应用 CUDA 以及 TensorFlow。

整体而言，当下热门的深度学习问题更适合直接使用 Python 框架去直接实现和并行优化，毕竟省去了造轮子的功夫，但是实验过程中了解到 CUDA 局部优化可以达到非常好的提速，所以不仅仅是依靠前人已有框架代码实现模型的并行化，还可以借助 OpenMP、CUDA 这种相对底层的工具去怼核心运算部分进行并行优化，也许会得到更好的效果，毕竟框架封装代码就仿佛黑盒，非常庞大，直接模型的并行化很可能只是普适的一个策略，并不能完全达到最好的结果。

参考文献

[1] 卷积神经网络 [EB/OL].https://en.wikipedia.org/wiki/CNN

[2] 凡保磊, 范明. 卷积神经网络的并行化研究 [D]. 河南, 郑州大学.2013

[3] Yu Gao.Parallel Convolutional Neural Networks power by GPU acceleration[EB/OL].http://cacs.usc.edu/education/cs653/result01/YuGao1.pdf

[4] Xiaqing Li, Guangyan Zhang, H. Howie Huang, Zhufan Wang, Weimin Zheng.Performance Analysis of GPU-based Convolutional Neural Networks[C].2016.International Conference on parallel processing

[5] OpenMP.[EB/OL].https://en.wikipedia.org/wiki/OpenMP

[6] What is CUDA? Parallel programming for GPUs.[EB/OL].https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html

[7] TensorFlow[EB/OL].https://en.wikipedia.org/wiki/TensorFlow

[8] PyTorch vs TensorFlow —spotting the difference[EB/OL].https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b

[9] PyTorch[EB/OL].https://en.wikipedia.org/wiki/PyTorch

[10] https://www.openmp.org

[11] https://www.nvidia.com/en-us/deep-learning-ai/education/

[12] 课件 algo1-2,openmp4-12,cuda1-8