异构计算项目报告

## 一、题目

基于已有的LeNet-5 CNN手写识别机器学习程序，分析程序的计算热点（kernal）；基于CPU+GPU的异构计算平台，采用OpenCL编程框架，编写异构执行代码，对程序的执行性能进行优化，在保持识别精度的前提下，最大限度地减少程序执行时间。

## 二、目的和要求

1. 下载运行LeNet-5卷积神经网络程序，统计训练时间、识别时间和识别精度关系，分析程序的计算热点；
2. 在保持识别精度条件下，基于通过CPU计算平台，采用循环展开、SIMD指令优化、多线程执行等方法对程序执行进行优化；
3. 在保持识别精度条件下，基于CPU+GPU的异构计算平台，编写OpenCL异构执行代码， 对程序的执行性能进行优化；
4. 基于3编写的OpenCL异构执行代码，采用存储器优化、工作项并行优化等方法对程序执行时间进行进一步化；
5. 至少对LeNet-5卷积神经网络的前向传导网络进行优化，包括：卷积层和池外层计算；
6. 撰写报告，对2、3、4中程序优化方法进行详细说明，对比分析不同方法获得的优化效果和原因。

## 三、项目环境

本次实验全部在课程服务器上完成，提供的计算资源包括CPU和GPU，下面是相关硬件的环境

### 硬件环境

CPU和GPU的硬件配置如下

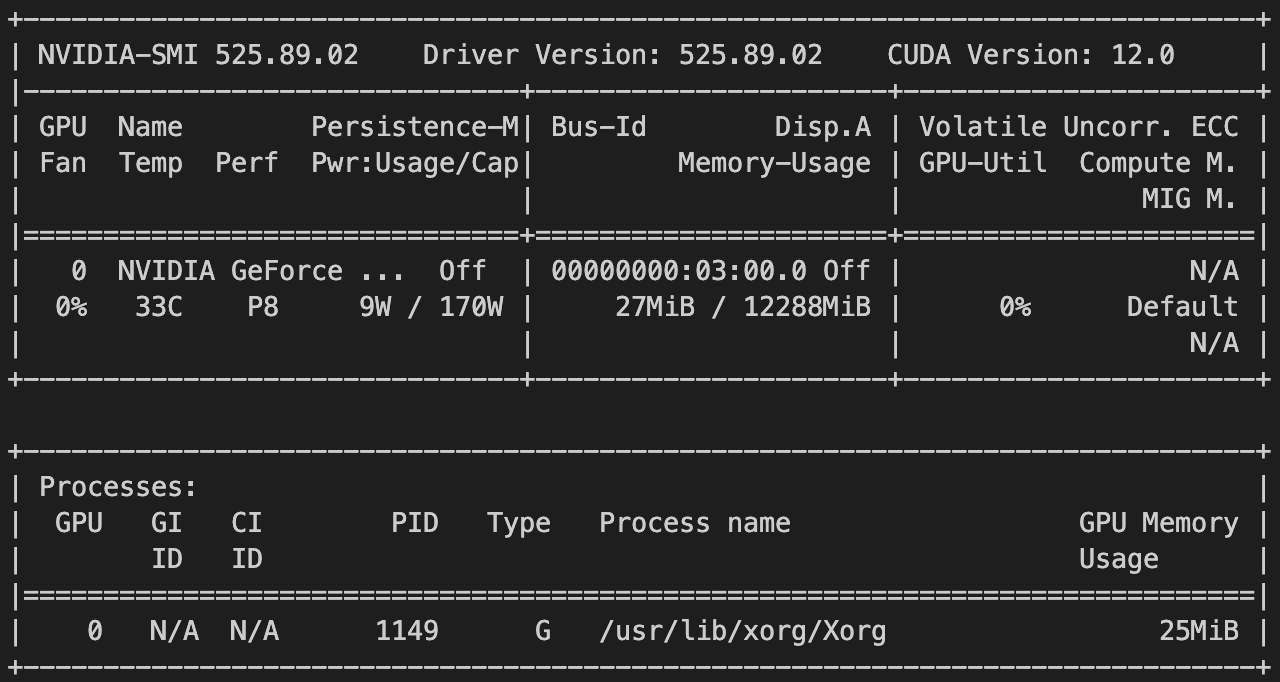
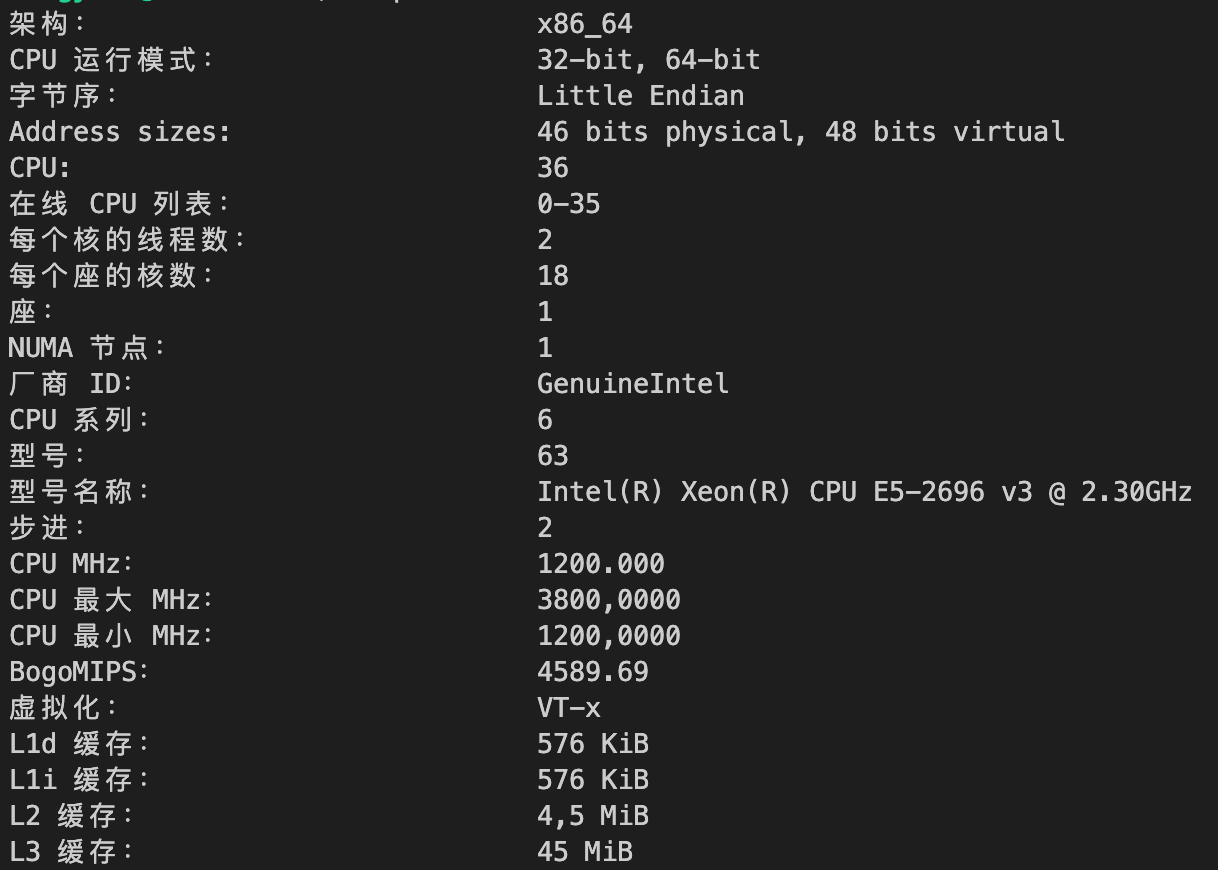


图1:GPU和CPU的硬件配置

### 软件环境

本次实验使用的GPU编程工具是CUDA，CUDA的版本为11.8，针对CPU编程使用的是c++，gcc的版本为9.4.0

## 四、项目实施步骤和过程

### CPU优化

#### AVX SIMD优化

AVX是指高级向量扩展（Advanced Vector Extensions），是Intel公司和AMD在x86架构中用于处理浮点数的一套指令集。AVX指令集引入了一些新的特性，最重要的是SIMD（单指令多数据）的扩展。使用AVX的优势是可以在一个指令中同时处理多个数据，这大大提高了处理器的计算能力。本次实验平台上的CPU支持AVX256的SIMD处理，这种宽度的向量能同时处理8个32位的浮点数或者4个64位的浮点数。利用AVX SIMD优化，可以同时处理多个数据，极大地加速这些运算。

具体的实验中，在卷积的计算中，可以使用\_mm256\_load\_ps读取输入向量数据，使用\_mm256\_fmadd\_ps进行数据相乘后的累加。

#### 多线程优化

多线程是指在一个程序中有多个线程同时执行，各个线程间共享数据和代码。在有多个处理器核心的计算机上，多个线程可以同时在不同的核心上执行，以提高总体的运行速度。在神经网络训练中，可以利用多线程并行处理多个样本的训练，或者将一个样本的计算分解到多个线程中并行处理。这种方式可以有效地利用多核处理器的性能，加速训练过程。

本次实验中，使用支持多线程的库，如OpenMP，可以自动地利用多线程进行优化。同时也可以进行手动优化：可以在程序中显式地创建和管理线程，将工作分解到各个线程中进行。这需要对线程编程有深入的理解。最后，考虑线程的并发控制：由于多个线程可能需要访问和修改同一份数据，因此需要考虑线程同步和互斥等问题，以保证计算的正确性。

#### 循环展开优化

#pragma unroll是一种常见的编译器指令，用于循环展开优化。这种指令是在编译时处理的，它会提示编译器尝试将循环中的操作展开。在很多情况下，当循环的迭代次数是固定且较小的时候，循环展开（Loop Unrolling）能够显著提升程序性能。展开后，程序中的循环次数会减少，每次循环的计算量会增加，但每次循环的开销（比如循环条件判断和跳转）会减少。这可以减少程序的执行时间，但可能会增加程序的代码大小。

在本次的实验中，可以在进行计算卷积的时候，或者是计算池化的时候，将内层的循环进行展开，增加程序的并行性。

#### 编译器优化

本次实验可以使用gcc -O3的编译方式，实现编译器的自动优化

### GPU优化

#### LeNet5训练的CUDA实现

本次实验由于是使用cuda实现，所以首先需要构建cuda的代码结构，首先使用cuda实现网络的三个层次，conv.cu、dense.cu和pooling.cu。在每个cuda文件中分别构造卷积类、池化类、全连接类。每个类中包括如下的内容：

•构造函数：根据输入参数设置卷积核的相关参数

•初始化函数：分配device内存用来前反向计算

•析构函数：释放device内存

•前向函数、反向函数：调用并执行kernel

使用这种代码结构的优势在于，可以在训练的过程中都把前反向的网络权重都放在device的内存中，从而减少CPU到GPU之间的数据搬运，增加数据的利用率。同时这种结构也使得显存方便管理，实现谁申请谁释放的代码原则，避免出现内存泄漏的问题。在train.cu文件中，负责网络构建和训练的功能，具体包括•创建三种类的实例、分配输入数据的device内存、前向调用、计算loss、反向调用、统计训练用时等模块。具体的代码结构如下图。

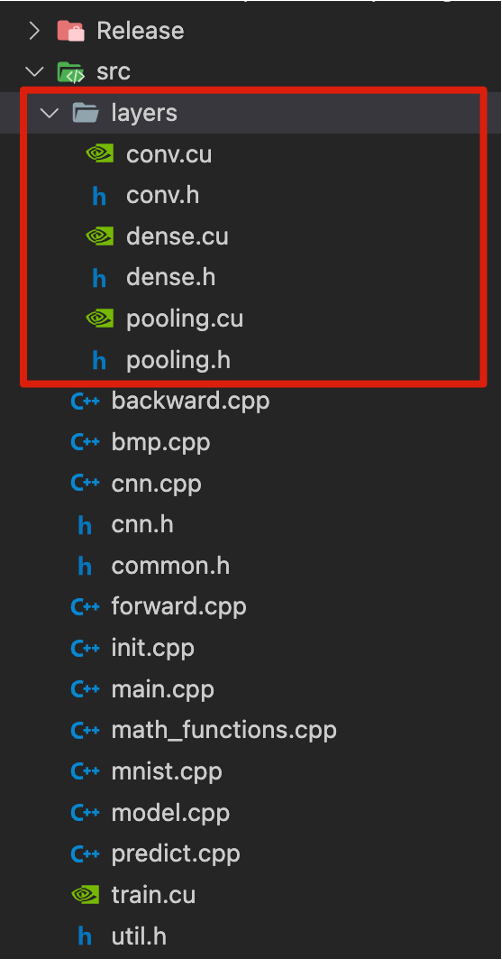
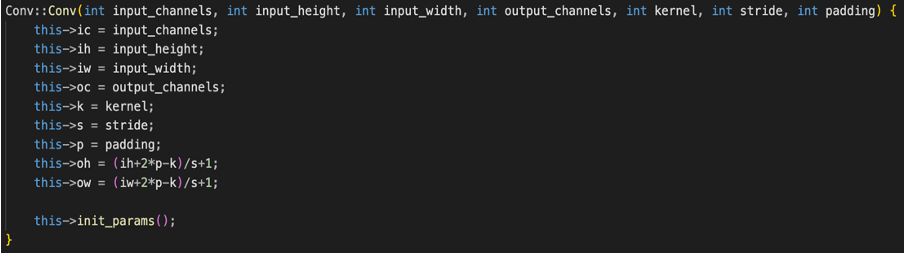


图2:总体代码结构



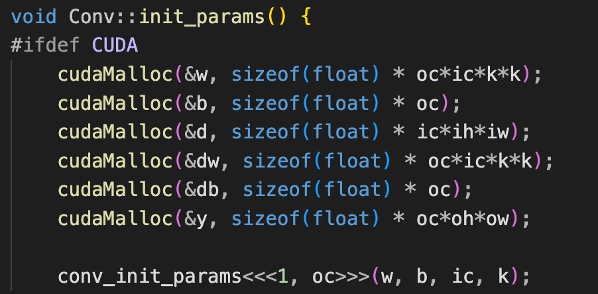
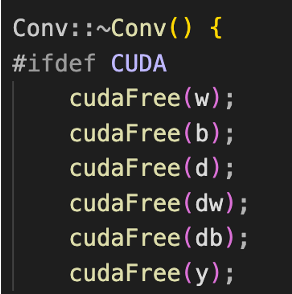


图3:每个类中的内容

#### 卷积层和池化层的优化

在我们对LeNet-5网络的训练过程中，卷积操作是计算量最大、最需要优化的部分。为了提高卷积操作的效率，我们采用了多种优化策略，包括共享内存优化、卷积核寄存器优化、循环展开以及tbl寄存器优化。

**共享内存优化**

首先，我们采用了共享内存优化技术。我们创建了一个线程块，其维度为 (TILE + k - 1, TILE + k - 1, TILEZ)。每个线程块中的所有线程都用于搬运数据。通过将数据移动到共享内存，我们可以避免多次从全局内存中读取同一数据，从而提高内存读取效率。

在搬运数据后，我们使用 \_\_syncthreads() 函数来同步所有线程，确保数据已经被完全搬运。然后，我们关闭一部分线程，只保留需要进行计算的线程。这样，我们可以在数据搬运和计算阶段之间灵活地调整线程的使用，以最大限度地提高计算效率。

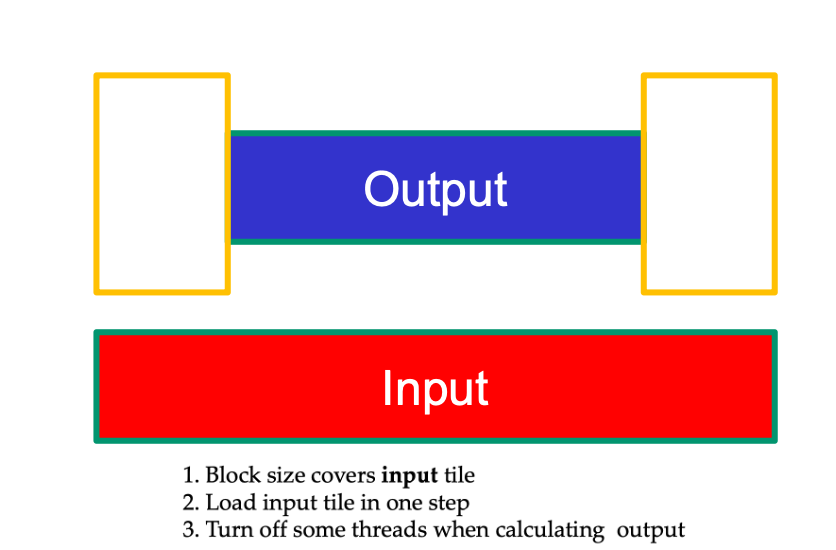


图4:卷积层中计算和加载数据的设计

**卷积核寄存器优化**

接下来，我们将卷积核加载到寄存器中，以减少卷积核的读取时间。由于寄存器是在硬件级别提供的存储空间，其读取速度比内存要快得多。这样，我们可以在计算卷积时减少读取卷积核的时间，从而提高卷积计算的速度。

**循环展开**

为了进一步优化卷积计算，我们采用了循环展开技术。通过使用 #pragma unroll 指令，我们将处理channel、ww、wh的循环展开，从而避免了每次迭代的开销，提高了计算效率。

**tbl寄存器优化**

在LeNet-5的结构中，S2和C3层之间存在着稀疏连接性。具体而言，C3层中的每一个特征图都只连接到S2层特征图的一个子集，这使得每一个特征图获取到的是不同的输入集，期望能提取到互补的特征。我们利用这种稀疏连接性进行优化。

我们将连接表存储在寄存器中，这样可以快速地读取每个C3特征图所连接的S2特征图的信息。这一步优化了S2到C3的卷积操作，减少了读取连接表的时间，进一步提高了卷积计算的效率。

#### 反向过程的实现

在神经网络训练中，反向传播算法是关键的一步，它负责计算每一层的梯度并更新权重和偏置。在我们优化LeNet-5网络的训练过程中，我们采用了多种优化策略来提高反向传播的效率。

**线程任务分配**

首先，我们明确规定了每个线程的职责。每个线程负责计算某一输出通道（oc）在某个位置的一部分梯度（dw）以及所有输入通道（ic）在同一位置的一部分梯度（d）。为了保证在并行计算过程中对共享数据的正确修改，我们使用了atomicAdd函数来进行梯度的计算和累加。这样，我们可以确保即使有多个线程同时修改同一数据，也不会出现数据不一致的问题。

每个线程块中的第一个线程被赋予了额外的任务，它负责在计算完所有梯度后更新所有的权重（w）和偏置（b）。这样，我们可以在梯度计算和权重更新之间进行有效的协调，提高反向传播的效率。

**同步计算梯度和更新梯度**

在我们的优化策略中，我们尽量将计算梯度和更新梯度的操作合并在一起进行。这样，我们可以避免频繁地在计算和更新之间切换，从而提高反向传播的速度。在本次的实验中，我把梯度的计算和更新放在一个kernel中完成，这样的实现方式在一定程度上减少了数据访问的开销。

为了保证数据一致性，我们在计算和更新梯度时都使用了atomicAdd函数。而且，我们在计算完dw和db后立即加上线程同步原语（如\_\_syncthreads()），以确保所有线程在进行下一步操作之前都已经完成了当前的计算和更新。

#### im2col优化

在完成前面的优化后，GPU占用大约可以到96%。最后，我们尝试把前向的卷积计算转换为矩阵乘法计算，从而使用矩阵乘法的优化手段进行优化。对于卷积运算，计算的公式是xxxxxxxxxx，是对多个位置的元素先做相乘，然后累加的操作，那么如果我们把输入的矩阵按照卷积核的顺序以列为单位排列，把卷积核按照行为单位进行排列，那么就可以把卷积计算转换为矩阵乘法计算。

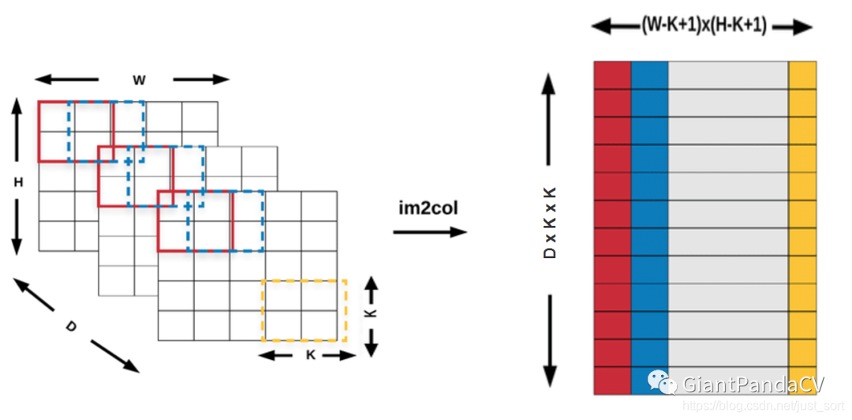
首先，我们可以输入的图像和卷积核都进行im2col，做完这一步的操作后，•假设输入的特征图维度是（1，D，H，W）其中表示Batch为1，通道数为D，高为H，宽为W，卷积核的维度是（D\_out，D，K，K）其中表示输出通道数为D\_out，卷积核大小是K\*K。那么，输入特征图的Im2col的维度（D\*K\*K,（W-K+1）\*（H-K+1 ））。卷积核Im2col的维度（D\_out，D\*K\*K），卷积的操作就变成两个二维矩阵的乘法的操作了。

图5:im2col转换示意图

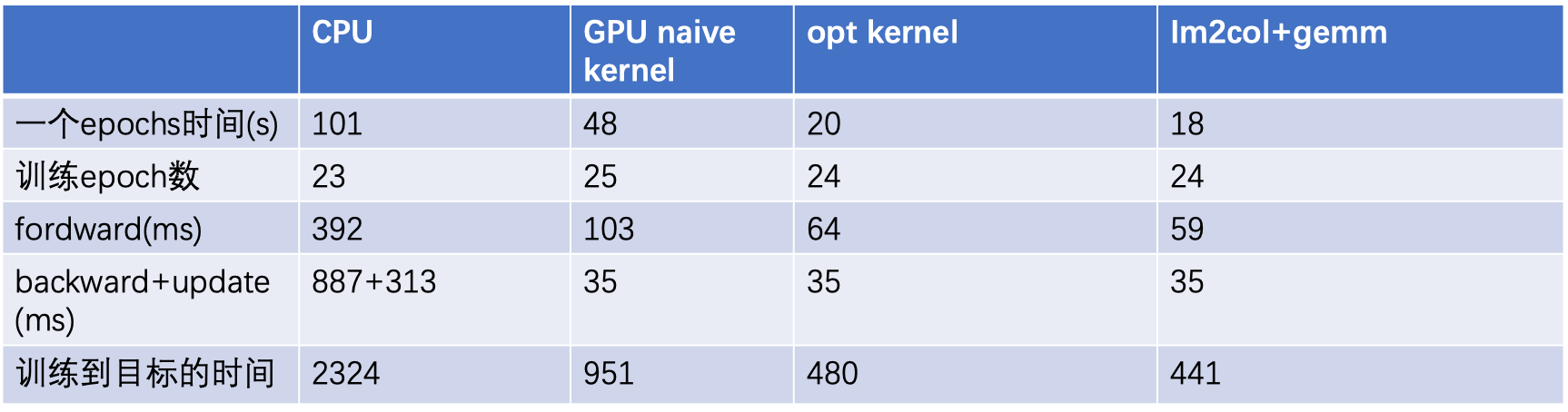
随后我们就可以使用矩阵乘法的优化方式来优化卷积操作。之前我们针对矩阵乘法有很多优化方式，例如从global memory到shared memory分块、从shared memory到register file分块、float4取数、double buffer(数据预取)等等方式，这里不一一展开。关于矩阵乘法的优化，我之前完成了一版达到91%的cuBLAS性能的矩阵乘法，本次实验按照这个作为基础。

## 五、实验数据记录

本次的实验数据记录分为几个部分，使用CPU -O3编译优化作为基准，比较GPU在不同优化级别下的性能。下图是一个epoch时间，训练epoch数，前向时间，反向+更新时间，训练到目标精度的总时间的对比。其中CPU表示CPU -O3编译优化的版本，GPU naive kernel表示不使用任何优化的GPU实现版本，Opt kernel表示对核函数采取多种优化技术后的版本，im2col+gemm表示使用im2col转化后的版本。

后面的四张图分别是训练精度收敛曲线、epoch的时间的比较、前反向kernel时间的比较、不同线程块大小对kernel时间的影响。

表1: 不同优化级别下的性能比较



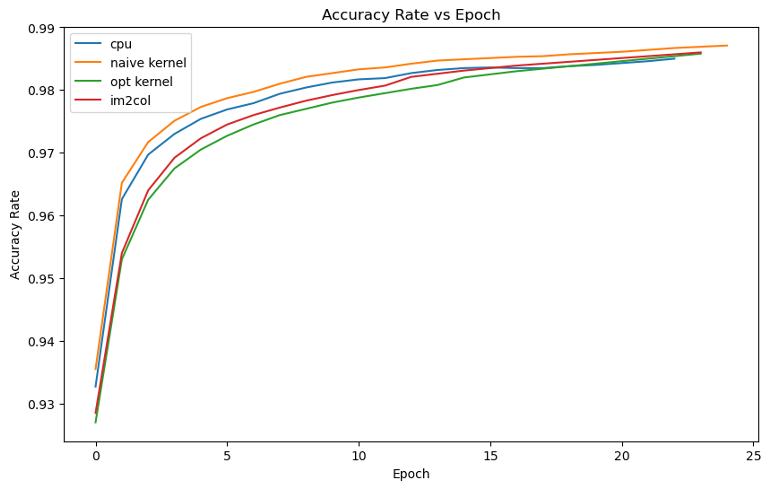


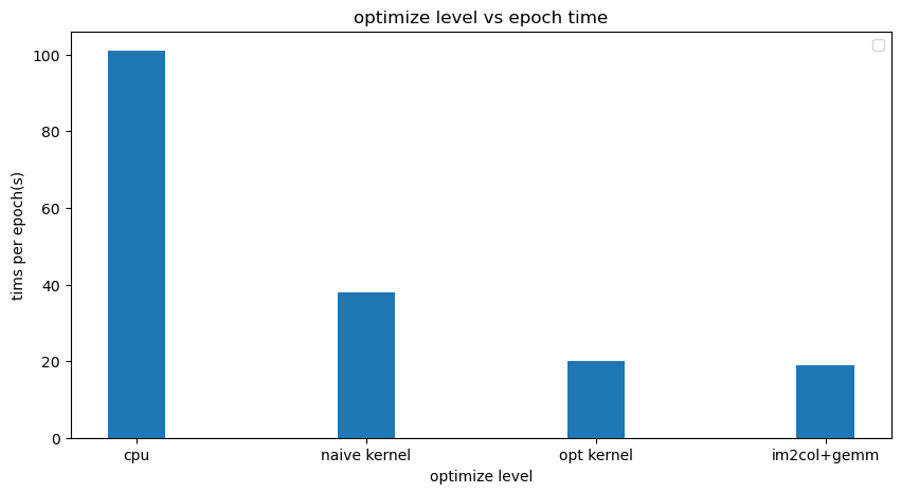
图6: 训练精度收敛曲线

图7: epoch的时间的比较

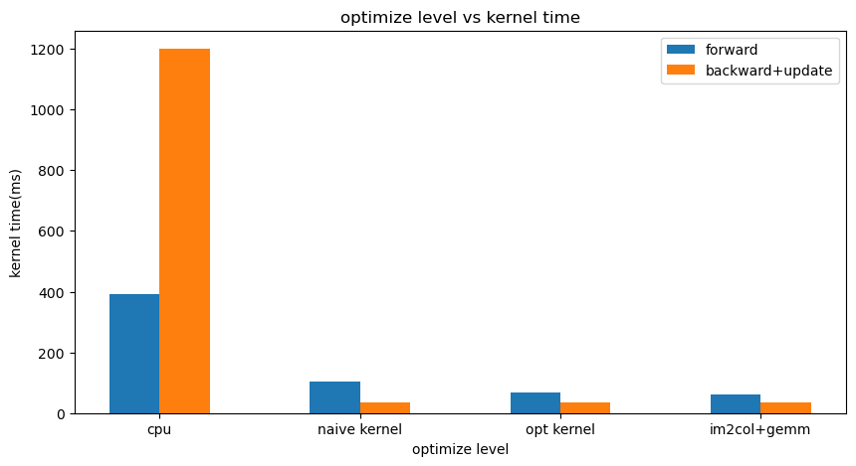


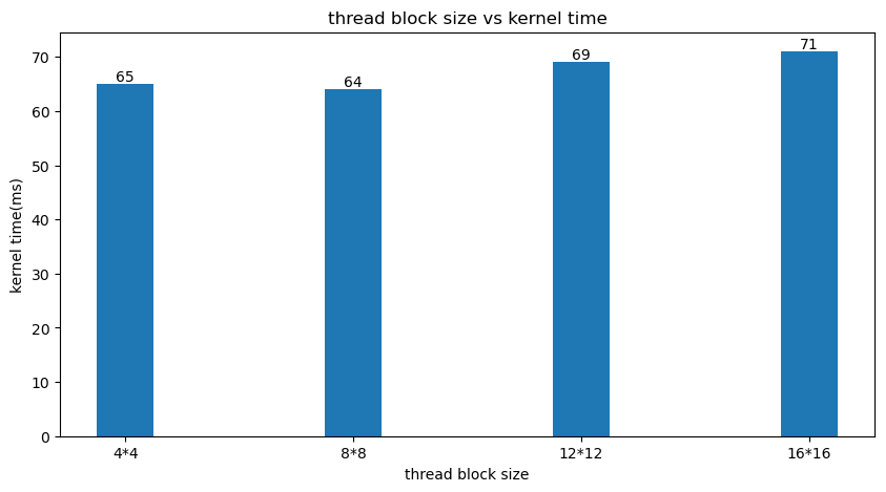
图8:前反向kernel时间的比较

图9: 不同线程块大小对kernel时间的影响。

## 六、分析及结论

根据第五部分的性能数据，我们可以对各种优化方法的效果进行分析：

1. **CPU vs GPU**：在原生态的条件下，使用GPU进行训练比使用CPU快了大约一倍以上。这主要归功于GPU的并行计算能力，使得它在处理神经网络训练这类需要大量并行计算的任务上有显著优势。
2. **naive vs kernel vs Opt kernel vs Im2col+gemm**：我们可以看到，随着优化技术的逐步引入，训练的性能有了显著的提高。首先，通过引入基本的卷积核(kernel)优化，一个epoch的时间从101秒减少到了48秒，这是一个显著的提升。进一步引入优化后的卷积核(Opt kernel)，一个epoch训练时间进一步降低到20秒。最后，通过使用Im2col+gemm技术，一个epoch训练时间进一步降低到18-19秒，前向传播时间(fordward)从64毫秒降低到了59毫秒，这表明Im2col+gemm技术在卷积操作的计算效率上有一定的优势，但是由于卷积核不够大，所以提升暂时有限，如果说数据量更大，卷积核的层数更多，卷积核更大，带来的性能提升会更加明显。
3. **backward+update**：在使用GPU及以上述各种优化技术的条件下，反向传播及权重更新的时间在所有的方法中都表现为相同，这是因为这部分操作在各个方法中使用的优化技术基本相同。对于这个部分的优化，主要是来源于cuda对于浮点原子操作的支持，以及我在实现过程中将计算和更新放在了一起，减少了kernel时间。
4. **训练到目标的时间**：这是一个综合的指标，反映了整个训练过程的时间效率。我们可以看到，随着优化技术的逐步引入，训练到目标的时间有了显著的减少。特别是在引入了Opt kernel和Im2col+gemm技术后，训练时间比原生态的GPU训练快了大约一倍。但是这个指标也收到训练数据初始化和网络随机初始化的影响，并不是一个稳定的数值。

总的来说，这些数据表明，通过引入并行计算、卷积核优化、Im2col+gemm技术等手段，我们可以显著提高神经网络训练的效率。这些优化技术在实际的神经网络训练过程中有着广泛的应用价值。

最后实验也对卷积计算中线程块大小的选择做出了指导，可以看出当计算线程块选择大小为8\*8的时候能获得最好的前向计算的性能提升。

## 七、感受及建议

通过本次实验，我深刻认识到异构计算在神经网络训练中的重要性。以下是我的一些感悟和建议：

首先，我理解到，硬件平台和计算模型的匹配度对于算法性能的影响不可忽视。例如，GPU具有强大的并行处理能力，如果能够充分利用这一特点，例如通过使用多线程技术，那么计算效率将大大提高。同样，在CPU平台上，使用SIMD指令集和多线程优化可以充分提升计算效能。

其次，我了解到，为特定计算任务设计和优化计算模型至关重要。例如，通过卷积核优化、Im2col+gemm等技术，我们可以显著提高神经网络训练的效率。这种优化需要对计算任务有深入的理解，以及对计算模型有精确的把握。

此外，我认识到，优化是一个持续的过程，需要不断尝试新的优化方法，并结合实验结果进行反馈和调整。这种不断迭代的过程可以让我们更深入地理解计算任务，更有效地使用硬件资源。

在未来，我建议我们的实验可以进一步探索新的优化技术，例如使用更先进的并行计算模型，或者设计更针对性的优化算法。同时，我们也可以进一步关注硬件技术的发展，以便能更好地利用新的硬件资源。

最后，特别感谢刘老师的指导，让我对异构计算有了更深入的理解，并期待在未来的实验中继续学习和探索。

**参考文献**

[1]Sanders J, Kandrot E. CUDA by example: an introduction to general-purpose GPU programming[M]. Addison-Wesley Professional, 2010.

[2]Kågström B, Ling P, Van Loan C. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark[J]. ACM Transactions on Mathematical Software (TOMS), 1998, 24(3): 268-302.

报告撰写人：王一迪

撰写时间：2023年6月7日