南開大學
Nankai University

计算机学院

并行程序设计期末报告

并行加速的经典机器学习框架

姓名：张铭徐 张惠程

学号：2113615 2112241

专业：计算机科学与技术

2023 年 6 月 30 日

目录

1 课程回顾与作业说明

1.1 课程回顾

本学期的并行程序设计我们主要学习了一些能够显著提高程序运行效率的并行化编程思想，具体而言，如下所示：

- 体系结构相关知识，例如循环展开策略，C++ 底层二维数据的存储格式，以及我们如何通过适当的更改枚举顺序提高访问效率，顺序读取的效率要明显高于需要寻址的访问方式。
- SIMD 的相关知识，即单指令多数据，可以在一条指令中对多条数据进行处理，SIMD 编程显著的特点是我们需要用到特殊的指令集和对应的寄存器，例如我们下面主要使用的 AVX 指令集，使用更大位数的寄存器可以存储更多的数据，加速读取效率。
- Pthread 和 OpenMP 的相关知识，Pthread 和 OpenMP 就是利用了计算机多线程的编程能力进行问题的分解和求解，将任务划分为若干的子任务分配给不同的线程，最后归约到一起，Pthread 和 OpenMP 也可以显著的提高程序运行的效率。
- MPI 的相关知识；消息传递接口，是一种并行编程模型，主要用于分布式内存系统，如集群和超级计算机。在 MPI 模型中，每个进程都有自己的内存空间，进程之间通过发送和接收消息进行通信，这种模型的主要优点是可扩展性强。
- GPU 的相关知识：由于 GPU 的结构特别适合并行编程和计算密集型的任务，我们可以考虑使用 GPU 进行编程，目前成熟的 GPU 编程框架有 OneAPI 套件和 SYCL 等，通过使用 GPU 编程，可以显著提高算法的执行效率，不过需要注意的是，极有可能 GPU 的速度慢于 CPU，因为要有通信时间，以及 CPU 和 GPU 之间的性能差异所致。

1.2 作业说明

本次作业为并行程序设计的期末大作业，我们的主要任务是将几种经典的机器学习算法：矩阵乘法，卷积，池化，反向传播，嵌入操作，PCA 主成分分析使用本学期讲授过的几种编程手段进行优化，测试不同优化方法的算法执行时间，并给予 Profiling 分析，针对 PCA，卷积我们将进行上述四种并行程序设计的思想进行算法设计，并测试程序运行时间，并与基础算法进行对比，尝试通过 profiling 等手段解释其差异的原因；对于余下的几种算法，我们将使用 SIMD(AVX 指令集)，Pthread，MPI 进行优化设计，同样通过相同的办法测定时间与分析实验数据。总结来说，本次大作业**并非是平时作业的简单串联**，至少有大约 50%-60% 的全新任务需要我们处理。

2 背景知识与实验平台介绍

2.1 并行背景知识介绍

2.1.1 SIMD 介绍

SIMD（Single Instruction Multiple Data）指令集是一种用于在单个指令中处理多个数据元素的计算机指令集。使用 SIMD 指令集可以在相同的时间内处理多个数据元素，从而提高计算机程序的运行效率。

在深度学习中，神经网络的训练通常需要大量的计算。使用 SIMD 指令集可以在较短的时间内对大量的数据进行计算，从而加快神经网络的训练速度。此外，许多深度学习框架（如 TensorFlow、PyTorch 和 MXNet）都已经优化了其底层的计算库，使其能够利用 SIMD 指令集进行高效的计算。

具体来说，使用 SIMD 指令集可以优化以下计算任务：

- 矩阵乘法：在神经网络中，矩阵乘法是一项非常常见的计算任务。使用 SIMD 指令集可以加速矩阵乘法的计算，从而加快神经网络的训练速度。
- 向量运算：在深度学习中，许多计算都涉及到对向量的操作，如向量加法、向量点积等。使用 SIMD 指令集可以同时处理多个向量，从而加快这些向量运算的速度。
- 卷积运算：卷积运算是深度学习中常见的计算任务之一，也是计算量最大的任务之一。使用 SIMD 指令集可以加速卷积运算的计算，从而加快神经网络的训练速度。

总的来说，使用 SIMD 指令集可以显著提高深度学习的计算效率，从而使神经网络的训练速度更快、更有效。SIMD 同样有两种，一种是 SSE，另一种是 AVX 指令集优化；通过先前的实验，我们可以知道：SSE 指令集相较于 AVX2 的效果较差一些，经过查阅资料和分析，我们找到了 SSE 和 AVX2 算法有以下异同之处：

相同之处：

- 都属于 SIMD 指令集：AVX2 和 SSE 都是 SIMD 指令集，它们通过在 CPU 中同时处理多个数据元素来实现并行性，从而提高程序的性能。
- 都是针对浮点数和整数操作的：AVX2 和 SSE 都支持浮点数和整数操作，包括加法、乘法、乘加等。
- 提高程序性能：通过使用 AVX2 或 SSE 指令集，您可以优化程序的性能，特别是在进行矩阵操作、图像处理、信号处理等高度数据并行的计算任务时。

不同之处：

- 寄存器大小：SSE 指令集使用 128 位的寄存器（XMM 寄存器），而 AVX2 指令集使用更大的 256 位寄存器（YMM 寄存器）。这意味着 AVX2 可以在单个指令中处理更多数据元素，从而实现更高的并行度。
- 指令集功能：AVX2 相比 SSE 提供了更多的功能和指令。例如，AVX2 支持整数的 Fused Multiply-Add（FMA）操作，而 SSE 不支持。此外，AVX2 还引入了许多针对整数操作的指令，包括排列、混合、位操作等，这在 SSE 中是不完整的或者完全缺失的。
- 性能差异：由于更大的寄存器大小和更丰富的指令集功能，AVX2 通常比 SSE 提供更好的性能。然而，实际性能差异可能因处理器架构、编译器优化选项以及输入数据的大小和特性而有所不同。
- 兼容性：AVX2 是在较新的处理器上引入的，而 SSE 是较早引入的指令集。因此，在较旧的处理器上，可能仅支持 SSE，而不支持 AVX2。在使用 AVX2 或 SSE 优化代码时，应检查处理器和编译器的兼容性。

总之，AVX2 和 SSE 都是针对数据并行计算的 SIMD 指令集，但 AVX2 提供了更大的寄存器、更丰富的指令集功能和更好的性能。然而，在选择使用 AVX2 或 SSE 时，应考虑到处理器和编译器的兼容性。既然 AVX2 使用了更大的寄存器，那么其有更好的性能表现也就不奇怪了，由于篇幅限制，在这里不展示 AVX2 与 SSE 指令集具体的性能表现的差异。

2.1.2 PthreadopenMP 介绍

pthread（POSIX threads）是一种基于 POSIX 标准的线程库，可用于实现多线程编程。pthread 提供了一组线程管理函数，例如创建线程、等待线程、设置线程属性等。我们在使用 pthread 编程时，需要注意以下问题：

- 线程安全：在多个线程中访问共享资源时，需要确保数据的一致性。可以使用互斥锁（mutex）或读写锁（rwlock）等同步原语来解决此问题。在需要访问共享资源的代码段前加锁，访问完成后解锁。
- 死锁：当两个或多个线程在资源上形成循环等待时，可能发生死锁。为避免死锁，可以采用以下方法：
  - 按顺序请求锁：确保所有线程以相同的顺序请求锁。
  - 限制同时请求的锁数量：确保一个线程在请求新锁之前释放已持有的锁。
- 错误处理：在使用 pthread 函数时，需要检查返回值以处理可能的错误。例如，pthread_create 可能会因为系统资源不足等原因失败。在这种情况下，应确保程序能够妥善处理错误。
- 资源泄露：线程退出时，应释放分配给线程的资源。如我们不及时释放资源，随着程序的运行，资源泄露可能导致性能下降，甚至耗尽系统资源，导致程序崩溃。因此，避免资源泄露是编写高质量、可靠的代码的重要原则。可以使用 pthread_cleanup_push 和 pthread_cleanup_pop 函数注册清理处理程序，以确保线程退出时资源得到释放。

并且，在使用 pthread 多线程操作时，我们还需要关注算法的正确性，同时，使用多线程编程技术相较于原本的单线程技术，也在时间方面具有较强的优越性，具体来说，如下所示：

- 正确性：pthread 的正确性取决于具体实现的算法。当在多线程环境中使用正确的同步原语（如互斥锁、条件变量等）时，可以确保算法的正确性。通常可以通过形式化方法、代码审查或者测试来证明算法的正确性。
- 优越性：使用 pthread 可以充分利用多核处理器的并行计算能力，从而提高程序的性能。在某些任务中，例如计算密集型任务或者需要并行处理大量数据的任务，多线程可以显著减少程序执行时间。然而，需要注意的是，并非所有任务都能受益于多线程。在 I/O 密集型任务中，线程同步和上下文切换的开销可能导致性能下降。因此，在实际应用中，需要根据具体任务特点选择合适的并发策略。

OpenMP（Open Multi-Processing）是一个用于 C/C++ 和 Fortran 编程语言的并行编程框架。它提供了一种简单、灵活的方式来编写多线程程序，利用多核处理器提高程序性能。OpenMP 通过使用编译器指令（pragma）和运行时库函数来实现并行操作，使得程序员可以在不修改底层代码结构的情况下，快速地将串行程序转换为并行程序。以下是一些基础性的知识和注意事项，我们后续的程序设计将围绕这些基础知识和注意事项所展开。

- 并行区域：并行区域是并行执行的代码块，可以通过 #pragma omp parallel 指令定义。当一个线程遇到这个指令时，会创建一个线程团队，其中包括主线程和若干个辅助线程。这些线程将并行地执行并行区域内的代码。

- 工作共享指令：OpenMP 提供了一系列工作共享指令，以便在并行区域内的线程之间划分任务。例如，#pragma omp for 指令用于将 for 循环的迭代划分给多个线程。还有其他工作共享指令，如 sections、single 和 task。
- 同步指令：为了避免数据竞争和不一致，OpenMP 提供了一系列同步指令。例如，#pragma omp critical 定义一个临界区，确保同一时刻只有一个线程可以执行该区域的代码。其他同步指令包括 barrier（障碍）、atomic（原子操作）和 flush（内存一致性）。
- 线程私有数据和共享数据：在并行区域内，数据可以被声明为线程私有（每个线程拥有一份独立的数据副本）或共享（所有线程共享同一份数据）。可以使用 private、shared、firstprivate 和 lastprivate 子句来指定数据的共享属性。
- 数据竞争：并行区域内的线程可能访问和修改共享数据，这可能导致数据竞争和不一致。要避免数据竞争，可以使用同步指令（如临界区和原子操作）或将数据声明为线程私有。
- 死锁：与 pthread 一样，在使用同步指令时，要注意避免死锁。死锁是指一组线程相互等待对方释放资源，导致程序无法继续执行。要避免死锁，可以确保以相同的顺序获取和释放资源，或使用嵌套锁（如 omp_set_nest_lock）。
- 负载均衡：为了实现最佳性能，需要确保并行区域内的任务在各个线程之间分配得尽量均匀。负载不均衡会导致部分线程闲置，从而影响性能。我们可以通过采用不同的调度策略，进行较为合理的任务划分，和利用嵌套并行等方法来实现负载均衡。

2.1.3 MPI 介绍

MPI，全称为 Message Passing Interface，即消息传递接口，是一种并行编程模型，主要用于分布式内存系统，如集群和超级计算机。在 MPI 模型中，每个进程都有自己的内存空间，进程之间通过发送和接收消息进行通信。这种模型的主要优点是可扩展性强，可以在各种硬件和网络配置上运行。MPI 的主要操作包括：

- 点对点通信：一个进程向另一个进程发送消息，或从另一个进程接收消息。这是最基本的通信模式，可以使用 MPI_Send 和 MPI_Recv 函数来实现。
- 集体通信：一个进程向所有其他进程发送消息，或从所有其他进程接收消息。这是一种更高级的通信模式，可以使用 MPI_Bcast（广播）、MPI_Scatter（分散）、MPI_Gather（收集）等函数来实现。
- 数据分发和收集：一个进程将数据分发给所有其他进程，或从所有其他进程收集数据。这是一种常用的数据处理模式，可以使用 MPI_Scatter 和 MPI_Gather 函数来实现。

MPI 还提供了许多其他的功能，例如：

- 通信域：MPI 允许用户创建自定义的通信域，即一组可以相互通信的进程。这可以让我们更灵活地组织你的并行程序。
- 派生数据类型：MPI 允许定义自己的数据类型，这可以让我们更方便地处理复杂的数据结构。
- 虚拟拓扑：MPI 允许创建虚拟的进程拓扑，例如网格或环形，这可以让我们更方便地编写需要特定拓扑结构的并行程序。

- 非阻塞通信：MPI 提供了非阻塞版本的发送和接收函数，例如 MPI_Isend 和 MPI_Irecv，这可以令通信和计算之间进行重叠，从而提高程序的效率。

深度学习中的许多操作，如卷积，池化和嵌入，都可以通过并行化来加速。MPI 可以在这些操作中发挥作用，因为它可以将任务分配给多个进程，每个进程在自己的数据集上工作，然后将结果合并。

对于一般 MPI 的编程范式，我们整理出了以下的可视化图表2.1：

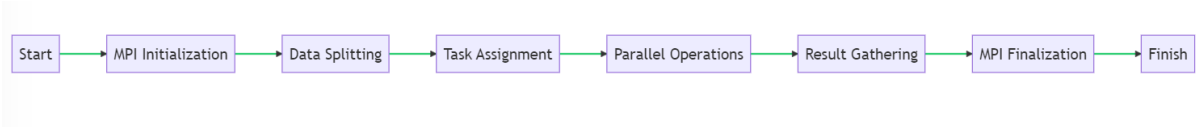

图 2.1: MPI_process_table

2.1.4 GPU 介绍

我们采用 Intel 的 OneAPI 套件进行实验。接下来我们总结一下 GPU 的相关内容：

GPU 编程是指利用图形处理器（Graphics Processing Unit，GPU）进行并行计算的编程方式。GPU 是一种专门设计用于处理图形和并行计算的硬件设备，具有大量的计算单元和内存，并且可以同时执行大量的线程。相比于中央处理器，GPU 具有更强大的并行计算能力，适用于处理需要大量数据并行计算即计算密集型的任务。GPU 具有大量的计算单元和内存，并且可以同时执行大量的线程。相比于中央处理器 CPU，GPU 具有以下特点和优势：

- 并行计算能力：GPU 拥有大量的计算单元（通常以流处理器或 CUDA 核心的形式存在），可以同时执行大量的线程。与 CPU 相比，GPU 能够并行处理更多的任务和数据，从而加速计算过程。这使得 GPU 在处理大规模数据集和计算密集型任务时表现出色。
- 大规模数据并行处理：GPU 的设计目标是处理图形渲染，而图形渲染涉及大量的并行计算，例如顶点变换、光照计算和纹理映射。这种设计使 GPU 在处理大规模数据并行任务时具有优势，例如科学计算、机器学习、深度学习等领域。
- 向量化计算：GPU 在设计上支持向量化计算，即同时对多个数据元素执行相同的操作。这种向量化计算可以提高数据的处理效率，并且可以通过优化指令和内存访问模式来提高计算性能。
- 大规模并行内存访问：GPU 具有较大的内存带宽和高效的内存访问模式，可以支持大规模并行的内存访问操作。这对于访问大规模数据集和高维数组非常重要，并且对于处理像素、图像和视频等数据密集型任务非常有利。
- 异构计算：现代计算平台越来越多地采用异构计算模型，其中 GPU 和 CPU 组合使用以实现更高的性能和能效。GPU 编程使得开发者可以充分利用 GPU 的并行计算能力，将计算任务分配到 GPU 和 CPU 之间，从而实现更高效的计算和加速。

不过需要注意的是，相比于 CPU，GPU 也存在一些限制和局限性：

- 控制流限制：GPU 通常适用于数据并行任务，而在控制流方面的处理相对较弱。由于 GPU 在执行时需要尽可能保持所有线程的同步，因此在涉及条件分支和循环等控制流操作较多的任务上，GPU 的效率可能相对较低。
- 内存限制：GPU 的内存容量通常较小，因此在处理大规模数据集时可能需要考虑内存限制。此外，GPU 的内存架构和访问模式也不同于 CPU，需要针对 GPU 的内存特性进行优化。

- 编程模型的学习曲线：相比于 CPU 编程，GPU 编程通常需要学习特定的编程模型和编程语言，例如 CUDA、OpenCL 或 SYCL。这要求开发者具备一定的并行计算和 GPU 架构的知识，并且需要进行适当的调优和优化才能实现最佳性能。

综上所述，GPU 编程是一种利用 GPU 的并行计算能力来加速计算的编程方式。通过充分利用 GPU 的并行计算、大规模数据并行处理和高带宽内存访问等特点，开发者可以实现更高效的计算和加速，在处理大规模数据集和计算密集型任务时具有明显的优势。然而，GPU 编程也需要考虑控制流限制、内存限制和学习曲线等因素，并进行适当的优化和调优才能实现最佳性能。

在 GPU 编程中，通常使用特定的编程模型和编程语言来利用 GPU 的并行计算能力。一种常见的 GPU 编程模型是通用并行计算，它允许开发者使用 GPU 进行通用计算而不仅仅是图形渲染。另外，许多厂商提供了针对各自 GPU 硬件的编程接口和开发工具，例如 NVIDIA 的 CUDA 和 AMD 的 OpenCL。

OneAPI 是由 Intel 提供的一个开发套件，旨在为异构计算提供统一的编程模型。OneAPI 套件包括多种编程语言、编译器和库，其中包括对 GPU 编程的支持。在 OneAPI 中，可以使用 SYCL（简单可扩展的异构编程）来进行 GPU 编程。SYCL 是一种基于标准 C++ 的编程模型，它提供了一种高级抽象的方式来利用 GPU 的并行计算能力。

使用 OneAPI 中的 GPU 编程功能，开发者可以使用 SYCL 编写基于 GPU 的并行计算代码，并利用 OneAPI 提供的编译器和运行时库将代码编译为针对特定 GPU 硬件的可执行文件。OneAPI 还提供了优化工具和调试工具，帮助开发者分析和优化 GPU 应用程序的性能。

总之，GPU 编程是利用 GPU 进行并行计算的编程方式，通过利用 GPU 的并行计算能力来加速计算密集型任务。Intel 的 OneAPI 套件提供了对 GPU 编程的支持，使用 SYCL 作为编程模型，开发者可以编写基于 GPU 的并行计算代码，并使用 OneAPI 提供的工具进行编译、优化和调试。

2.2 机器学习算法介绍

2.2.1 矩阵乘法

在深度学习中，矩阵乘法是非常常见的操作，因为目前联结主义的人工智能的代表作深度学习，就是以矩阵为单位进行数据的处理，分析和拟合。如果我们能够加速矩阵乘法，那么我们就可以大大提高深度学习的效率。假设有两个矩阵 A 和 B，它们的维度分别为 $m \times n$ 和 $n \times p$，则它们的矩阵乘法结果为 C，维度为 $m \times p$。矩阵乘法的公式如下所示：

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j} C$$

其中，$C_{i,j}$ 表示矩阵 C 中第 i 行第 j 列的元素，$A_{i,k}$ 表示矩阵 A 中第 i 行第 k 列的元素，$B_{k,j}$ 表示矩阵 B 中第 k 行第 j 列的元素。

在深度学习中，矩阵乘法通常用于计算两个矩阵的点积，以及神经网络中的前向传播和反向传播等操作。由于神经网络通常包含大量的矩阵计算，因此使用并行化的策略可以加速这些计算，提高神经网络的训练速度。

2.2.2 卷积操作

卷积运算是深度学习中非常常见的操作之一，用于提取图像、音频等数据中的特征。假设有两个矩阵 I 和 K，它们的维度分别为 $m \times n$ 和 $k \times k$，则它们的卷积运算结果为 S，维度为 $(m-k+1) \times (n-k+1)$。卷积运算的公式如下所示：

$$S_{i,j} = \sum_{u=1}^{k} \sum_{v=1}^{k} I_{i+u-1,j+v-1} K_{u,v} S$$

其中，$S_{i,j}$ 表示卷积运算结果中第 i 行第 j 列的元素，$I_{i+u-1,j+v-1}$ 表示输入矩阵中第 $(i+u-1)$ 行第 $(j+v-1)$ 列的元素，$K_{u,v}$ 表示卷积核矩阵中第 u 行第 v 列的元素。

在深度学习中，卷积运算通常用于卷积神经网络中的卷积层，用于提取图像、音频等数据中的特征。由于神经网络中通常需要进行大量的卷积运算，如果我们能够使用多线程策略对其进行加速，那么整体的训练和收敛速度将会大大提高，极大程度上减少算力的要求。

在计算机视觉的主流任务中，例如图像分类与目标检测等任务，往往都需要提取特征，在此时，卷积层便能起到很好的特征提取的效果，在目前效果比较好的框架中，例如人脸检测的 MTCNN，就使用了三层级联的网络用于进行特征提取与人脸检验，其网络主要应用的，就是卷积和池化的操作，网络的整体架构如下图2.2所示：

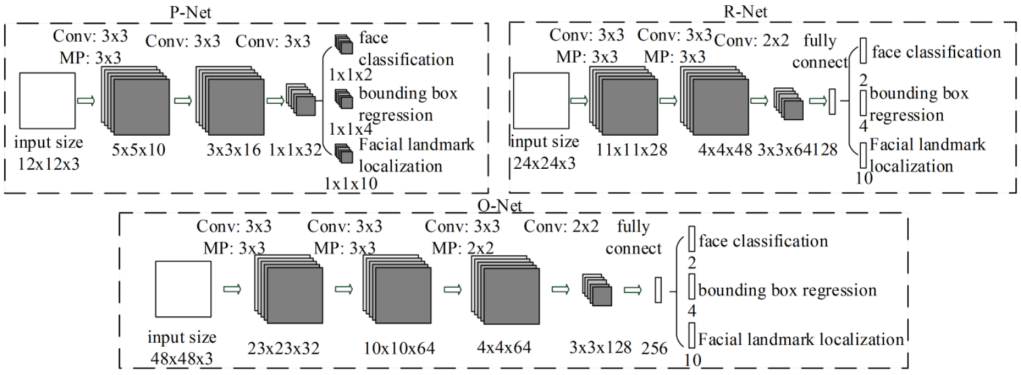

图 2.2: MTCNN's Framework

2.2.3 反向传播

在神经网络中，反向传播算法的训练过程可以分为两个主要步骤：前向计算和反向计算。在前向计算过程中，网络将输入数据通过一系列的线性变换和非线性变换，得到最终的输出结果。在反向计算过程中，网络通过反向传播误差信号，更新网络中的权重和偏置，以实现训练的目的。

具体来说，设输入层的数据为 $\mathbf{x}$，输出层的数据为 $\mathbf{y}$，网络的损失函数为 $L(\mathbf{y}, \mathbf{t})$，其中 $\mathbf{t}$ 是目标数据。BP 算法的目标是最小化损失函数，即 $\min L(\mathbf{y}, \mathbf{t})$。为了实现这一目标，BP 算法使用梯度下降法来更新网络中的权重和偏置。具体来说，BP 算法通过计算损失函数对权重和偏置的导数，来更新网络中的参数。

在反向计算过程中，首先需要计算输出层的误差信号。假设输出层的激活函数为 $\sigma(\cdot)$，则输出层的误差信号可以表示为：

$$\delta^L = \nabla_{\mathbf{a}^L} L \odot \sigma'(\mathbf{z}^L)$$

其中，$\mathbf{z}^L = \mathbf{w}^L \mathbf{a}^{L-1} + \mathbf{b}^L$，表示输出层的输入，$\odot$ 表示向量逐元素相乘，$\sigma'(\cdot)$ 表示激活函数的导数。接下来，反向传播误差信号，计算隐藏层的误差信号。假设第 l 层的激活函数为 $\sigma(\cdot)$，则第 l 层的误差信号可以表示为：

$$\delta^l = ((\mathbf{w}^{l+1})^T \delta^{l+1}) \odot \sigma'(\mathbf{z}^l)$$

其中，$\mathbf{z}^l = \mathbf{w}^l \mathbf{a}^{l-1} + \mathbf{b}^l$，表示第 l 层的输入。最后，可以使用误差信号来更新权重和偏置，以实现网络的训练。假设第 l 层和第 $l+1$ 层之间的权重为 $\mathbf{w}^{l+1}$，偏置为 $\mathbf{b}^{l+1}$，学习率为 η，则可以使用以下公式来更新网络参数：

$$\mathbf{w}^{l+1} \leftarrow \mathbf{w}^{l+1} - \eta \delta^{l+1} (\mathbf{a}^l)^T$$
$$\mathbf{b}^{l+1} \leftarrow \mathbf{b}^{l+1} - \eta \delta^{l+1}$$

其中，δ^{l+1} 表示第 $l+1$ 层的误差信号，$\mathbf{a}^l$ 表示第 l 层的输出。

我们同样可以使用多线程技术对反向传播算法进行优化，可以大大提高训练效率和准确性，使神经网络能够更快地收敛和达到更好的性能。

2.2.4 池化操作

池化操作（Pooling Operation）是深度学习中常用的一种操作，用于减小特征图的空间维度，同时保留重要的特征信息。池化操作在卷积神经网络（CNN）等模型中广泛应用，用于降低计算复杂度、减少过拟合，并增强模型的平移不变性。

池化操作主要有两种类型：最大池化（Max Pooling）和平均池化（Average Pooling）。这里我们将重点介绍最大池化操作。

最大池化：给定输入特征图（或称为池化输入）X，最大池化操作通过将特征图划分为不重叠的区域，并在每个区域中选取最大值作为池化输出。具体而言，对于输入特征图的每个区域，最大池化操作选取该区域内的最大值作为对应的输出值。类似的，我们有平均池化的定义：给定输入特征图（或称为池化输入）X，平均池化操作通过将特征图划分为不重叠的区域，并在每个区域中计算特征值的平均值作为池化输出。具体而言，对于输入特征图的每个区域，平均池化操作计算该区域内特征值的平均值作为对应的输出值。

数学上，最大池化和平均池化操作可以用以下公式表示：

$$\text{Max Pooling}(X)_{i,j,k} = \max_{m,n}(X_{(i\cdot\text{stride}+m),(j\cdot\text{stride}+n),k})$$

$$\text{Average Pooling}(X)_{i,j,k} = \frac{1}{\text{pooling\_size} \times \text{pooling\_size}} \sum_{m,n}(X_{(i\cdot\text{stride}+m),(j\cdot\text{stride}+n),k})$$

其中，$X_{i,j,k}$ 表示输入特征图 X 在第 i 行、第 j 列、第 k 个通道上的值。$\text{Max Pooling}(X)_{i,j,k}$ 表示最大池化操作的输出值，它是对应区域内的最大值；$\text{Average Pooling}(X)_{i,j,k}$ 表示平均池化操作的输出值，它是对应区域内特征值的平均值；stride 表示池化操作的步长（stride），用于控制池化窗口的移动步幅。通过调整步长，可以控制输出特征图的尺寸；pooling_size 表示池化窗口的大小，即池化操作在每个区域内考虑的特征值数量。

需要注意的是，池化操作在每个通道上是独立进行的，因此在应用池化操作时，针对输入特征图的每个通道分别进行池化。

最大池化操作的效果是通过保留每个区域内的最大值，来提取出特征图的显著特征。通过降低特征图的空间维度，最大池化操作可以减少参数数量，并提高模型的计算效率。平均池化操作通过计算特征图区域内的平均值来提取特征图的平滑特征，能够对输入特征图进行降维和平滑处理。它可以在

一定程度上减少噪声的影响，并且也有助于提取图像或特征图的整体统计特征

总结起来，池化操作通过划分输入特征图为不重叠的区域，并选取每个区域内的最大值或者平均值作为输出值，来减小特征图的空间维度。它是深度学习中常用的操作之一，用于增强模型的平移不变性和降低计算复杂度。

2.2.5 Embedding(嵌入操作背景知识介绍)

嵌入层（Embedding Layer）是深度学习中常用的一种层类型，用于将离散的输入特征映射到低维连续向量空间。嵌入层通过学习每个离散特征的低维度表示，可以捕捉特征之间的语义关系和相似性。

给定一个离散的输入特征 x，例如单词或类别的索引，嵌入层将其映射为一个低维的嵌入向量 e。嵌入向量的维度通常由设计者指定，例如 d 维。嵌入层的参数是一个嵌入矩阵 W，其大小为 $V \times d$，其中 V 是离散特征的总数目。

嵌入操作可以表示为：

$$e = \text{Embedding}(x) = Wx$$

其中，W 是嵌入矩阵，x 是输入特征的索引，e 是对应的嵌入向量。

嵌入层的参数 W 是通过训练过程学习得到的。在训练过程中，通过最小化某个损失函数，模型会更新嵌入矩阵 W 的值，使得嵌入向量能够更好地表示特征之间的语义关系和相似性。

嵌入层可以用于各种深度学习任务，如文本分类、推荐系统、序列建模等。它将离散的输入特征转换为连续的低维嵌入向量，提供了一种更有效地表示和处理离散特征的方法。

总结起来，嵌入层通过学习将离散的输入特征映射到低维连续向量空间，提供了一种有效的特征表示方法。通过嵌入操作，输入特征 x 被映射为对应的嵌入向量 e，并且嵌入矩阵 W 的参数通过训练过程学习得到。这样的嵌入向量可以用于后续的模型训练和推断。

2.2.6 PCA 主成分分析

主成分分析（Principal Component Analysis，PCA）是一种常用的数据降维和特征提取技术，用于发现数据集中的主要变化方向。

给定一个包含 n 个样本的数据集，每个样本有 m 个特征，可以构建一个 $n \times m$ 的数据矩阵 X，其中每一行表示一个样本，每一列表示一个特征。PCA 的目标是通过线性变换，将原始数据投影到一个新的坐标系中，使得投影后的数据具有最大的方差。

- 1. 数据中心化：首先，对数据进行中心化操作，即从每个特征维度中减去其均值，以确保数据的均值为零。通过计算每个特征的均值 μ，可以将数据矩阵 X 中的每个元素减去相应的均值，得到中心化后的数据矩阵 $\hat{X}$。

  $$\hat{X}_{ij} = X_{ij} - \mu_j$$

  其中，$\hat{X}_{ij}$ 表示中心化后的数据矩阵 $\hat{X}$ 的元素，X_{ij} 表示原始数据矩阵 X 的元素，μ_j 表示第 j 个特征的均值。

- 2. 协方差矩阵：接下来，计算中心化后的数据矩阵 $\hat{X}$ 的协方差矩阵 C。协方差矩阵用于描述数据特征之间的相关性和方差。

  $$C = \frac{1}{n-1}\hat{X}^T\hat{X}$$

其中，$\hat{X}^T$ 表示中心化后的数据矩阵 $\hat{X}$ 的转置。

- 3. 特征值分解：对协方差矩阵 C 进行特征值分解，得到特征值和对应的特征向量。特征值表示数据中的主要方差，特征向量表示对应于主要方差的方向。

$$CV = \lambda V$$

其中，V 是由特征向量组成的矩阵，λ 是特征值的对角矩阵。

- 4. 选择主成分：根据特征值的大小，选择前 k 个特征向量作为主成分，其中 k 是降维后的维度。这些特征向量对应于最大的特征值，表示数据中的主要变化方向。
- 5. 数据投影：将中心化后的数据矩阵 $\hat{X}$ 投影到选择的主成分上，得到降维后的数据矩阵 Y：

$$Y = \hat{X}V_k$$

其中，V_k 是选择的前 k 个特征向量组成的矩阵。

通过 PCA 降维，我们可以将原始数据投影到一个较低维度的子空间中，保留了数据中最重要的变化方向。这样可以减少特征维度，提高计算效率，并且在某些情况下，可以更好地可视化和解释数据。

总结起来，PCA 通过数据中心化、计算协方差矩阵、特征值分解、选择主成分和数据投影等步骤，将高维的原始数据降低到较低维度的表示。通过选择主要方差所对应的特征向量，PCA 帮助我们发现数据集中的主要变化方向，并提供了一种数据降维和特征提取的方法。

2.3 实验平台介绍

本次实验平台采用 X86 平台，操作系统为 Windows 11，CPU 为 Intel Core12th，实验电脑型号为联想拯救者 Y9000P2022 版，GPU 为移动端 130W 的 RTX3060，同时，计算机的 RAM 为 16G，编译器采用 Visual Studio。对于 Profiling 部分，采用 VTune 进行剖析，电脑采用 X86 平台，CPU 为 Intel Core11th, 操作系统和编译器同上。

3 算法设计思路

3.1 矩阵乘法设计思路

3.1.1 朴素算法设计思路

朴素版的矩阵乘法算法使用三重循环进行计算，对于矩阵 $A \in \mathbb{R}^{m\times n}$ 和矩阵 $B \in \mathbb{R}^{n\times p}$，可以得到其乘积矩阵 $C \in \mathbb{R}^{m\times p}$：

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j} \quad (i = 1, \ldots, m;\ j = 1, \ldots, p)$$

算法的时间复杂度为 $O(n^3)$，在矩阵较大时，计算代价会比较高。根据上述的设计思想，我们可以写出算法流程图，如算法11所示：

Algorithm 1 矩阵乘法朴素版

Input: 矩阵 A 和矩阵 B
Output: 矩阵 $C = A \times B$
$m \leftarrow$ 矩阵 A 的行数
$n \leftarrow$ 矩阵 A 的列数
$p \leftarrow$ 矩阵 B 的列数
初始化矩阵 C 为 $m \times p$ 的零矩阵
for $i \leftarrow 1$ **to** m **do**
  for $j \leftarrow 1$ **to** p **do**
    $c_{i,j} \leftarrow 0$
    for $k \leftarrow 1$ **to** n **do**
      $c_{i,j} \leftarrow c_{i,j} + a_{i,k} \times b_{k,j}$
    end
  end
end

3.1.2 SIMD 设计思路

在 SIMD 优化的算法中，我们使用向量寄存器进行优化。将矩阵拆分为多个小矩阵，然后使用向量指令进行计算，可以一次性处理多个数据。具体步骤如下：

- 将矩阵 B 转置为 B^T。
- 初始化一个向量寄存器，用于存储 B^T 矩阵中的一列。
- 对于矩阵 A 的每一行，取出一列数据，与 B^T 中的一列进行向量乘法。
- 将所有向量乘积的结果求和，得到 C 矩阵的一个元素。
- 算法的时间复杂度为 $O(\frac{n^3}{w})$，其中 w 表示向量寄存器的长度。通过使用向量寄存器，可以显著提高计算效率。

对于上述的分析过程，我们可以写出对应的算法流程图, 如算法2所示：

Algorithm 2 矩阵乘法 SIMD 优化版 (使用 AVX2 指令集)

Input: 矩阵 A 和矩阵 B
Output: 矩阵 $C = A \times B$
$m \leftarrow$ 矩阵 A 的行数 $n \leftarrow$ 矩阵 A 的列数 $p \leftarrow$ 矩阵 B 的列数 初始化矩阵 C 为 $m \times p$ 的零矩阵 使用向量寄存器，将矩阵 B 展开为一维向量 B_{vec} **for** $j \leftarrow 1$ **to** p **do**
  for $k \leftarrow 1$ **to** n，以 w 为步长 **do**
    使用向量指令，将矩阵 A 的第 k 列展开为一维向量 $A_{k,vec}$ 对于矩阵 C 的第 j 列，使用向量指令，将其展开为一维向量 $C_{j,vec}$ 使用向量指令，计算 $A_{k,vec} \times B_{vec}$，得到一组乘积向量 对于矩阵 C 的第 j 列，使用向量指令，将乘积向量的所有元素相加，得到 $c_{i,j}$ 的值 将 $c_{i,j}$ 的值写入矩阵 C 中对应的位置
  end
end

3.1.3 Pthread&OpenMP 设计思路

原始代码只使用单个线程进行矩阵乘法计算。通过使用多线程，我们可以充分利用现代多核处理器的计算能力，从而加速计算过程。根据我们的背景知识，我们可以列出以下设计思想：值得注意的

是，在这里我们使用了 SIMD 优化策略进行优化：

- 为了实现多线程，我们首先定义一个结构体 MatrixData 来存储每个线程所需的数据。
- 接下来，我们创建一个 matrix_multiply_avx2_thread 函数，它会接受 MatrixData 类型的指针作为参数，并完成矩阵乘法的部分计算。
- 在 main 函数中，我们创建多个线程，并为每个线程分配一部分矩阵 A 的行进行计算。最后，我们使用 pthread_join 函数等待所有线程完成计算。

下面是使用多线程优化后的矩阵乘法算法流程图：

Algorithm 3 使用 AVX2 指令集的矩阵乘法并行计算算法

Input: 矩阵 $A \in \mathbb{R}^{m\times n}$, 矩阵 $B \in \mathbb{R}^{n\times p}$, 矩阵 $C \in \mathbb{R}^{m\times p}$
Output: 矩阵 $C = AB$ 的结果
创建 $num\_threads$ 个线程和线程数据结构
将矩阵 A 和 B 随机初始化
for $i \leftarrow 0$ *to* $num\_threads - 1$ **do**
  初始化线程数据结构中的 $a, b, c, m, n, p, star\_row$ 和 end_row
  创建并启动线程 i，每个线程执行 matrix_multiply_avx2_thread 函数
end
等待所有线程完成

值得注意的是，在这个例子中，我们并未展示在 matrix_multiply_avx2_thread 函数中具体使用 AVX2 指令集实现矩阵乘法的细节。这部分代码与上次 SIMD 实验的代码相同，可以参考那部分代码，或直接访问 GitHub 项目仓库进行查找相关文件。

根据上面的分析，我们在下面给出 Pthread 优化的矩阵乘法代码，值得注意的是，我们仅仅给出矩阵乘法对应的函数，输入数据和整体代码详见后文的 GitHub 项目链接。对于 pthread 优化部分，我们仅仅给出主函数部分有关多线程操作部分的关键代码，数据生成部分，以及具体的 matrix_multiply_avx2_thread 函数，由于篇幅限制，我们在此处暂未给出。

矩阵乘法 pthread 优化部分

```
    std::vector<pthread_t> threads(num_threads);
    std::vector<MatrixData> thread_data(num_threads);
    int chunk_size = m / num_threads;
    auto start = std::chrono::high_resolution_clock::now();
    // 创建并启动线程
    for (int i = 0; i < num_threads; ++i) {
        thread_data[i].a = &a;
        thread_data[i].b = &b;
        thread_data[i].c = &c;
        thread_data[i].m = m;
        thread_data[i].n = n;
        thread_data[i].p = p;
        thread_data[i].start_row = i * chunk_size;
        thread_data[i].end_row = (i == num_threads - 1) ? m : (i + 1) * chunk_size;
        pthread_create(&threads[i], nullptr, matrix_multiply_avx2_thread,
            &thread_data[i]);
```

```
    }
    // 等待所有线程完成
    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], nullptr);
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "矩阵乘法完成" << std::endl;
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "计算耗时: " << elapsed.count() << " 秒" << std::endl;
```

3.1.4 MPI 设计思路

我们针对该部分的整体设计思路是利用 MPI 的分布式计算能力，将矩阵乘法任务划分为多个进程进行并行计算，并使用 MPI 的通信函数进行数据的分发和收集，最终将各个进程的计算结果合并得到最终的矩阵乘积结果。这种设计思路能够充分利用集群或多核系统的计算资源，提高矩阵乘法的计算效率。我们整理出了如下所示的算法流程图，如4所示：

Algorithm 4 矩阵乘法

Input: 矩阵 A，矩阵 B
Output: 结果矩阵 C

function MatrixMultiply(A, B)
  $m \leftarrow$ 行数(A)
  $n \leftarrow$ 列数(A)
  $p \leftarrow$ 列数(B)
  $C \leftarrow$ 创建大小为$m \times p$ 的零矩阵 **for** $i \leftarrow 0$ *to* $m-1$ **do**
    end
    $j \leftarrow 0$ to $p-1$ **for** $k \leftarrow 0$ *to* $n-1$ **do**
    end
    $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
  return C
end function

3.2 卷积操作设计思路

3.2.1 朴素算法设计思路

朴素版的卷积运算算法使用四重循环进行计算，对于输入矩阵 $I \in \mathbb{R}^{m \times n}$ 和卷积核 $K \in \mathbb{R}^{k \times k}$，可以得到其输出矩阵 $S \in \mathbb{R}^{(m-k+1)\times(n-k+1)}$：

$$S_{i,j} = \sum_{u=1}^{k} \sum_{v=1}^{k} I_{i+u-1,j+v-1} K_{u,v} \quad (i = 1, \ldots, m-k+1;\ j = 1, \ldots, n-k+1)$$

算法的时间复杂度为 $O(n^4)$，在卷积核较大时，计算代价会比较高。对于朴素算法，其算法流程图如算法5所示：

Algorithm 5 naive-Convolution

Input: 输入矩阵 I 和卷积核 K
Output: 输出矩阵 S
$m \leftarrow$ 输入矩阵 I 的行数
$n \leftarrow$ 输入矩阵 I 的列数
$k \leftarrow$ 卷积核 K 的大小
初始化输出矩阵 S 为 $(m-k+1) \times (n-k+1)$ 的零矩阵
for $i \leftarrow 1$ **to** $m-k+1$ **do**
  for $j \leftarrow 1$ **to** $n-k+1$ **do**
    $S_{i,j} \leftarrow 0$ **for** $u \leftarrow 1$ **to** k **do**
      for $v \leftarrow 1$ **to** k **do**
        $S_{i,j} \leftarrow S_{i,j} + I_{i+u-1,j+v-1} \times K_{u,v}$
      end
    end
  end
end

3.2.2 SIMD 设计思路

在 SIMD 优化的算法中，我们同样使用向量寄存器进行优化。将输入矩阵和卷积核矩阵分别展开成一维向量，然后使用向量指令进行计算，可以一次性处理多个数据。具体步骤如下：

- 将卷积核矩阵展开成一维向量 K_{vec}。
- 对于输入矩阵 I 的每个子矩阵，将其展开成一维向量 I_{vec}。
- 使用向量指令对 I_{vec} 和 K_{vec} 进行向量乘法，得到一组乘积向量。
- 将乘积向量的所有元素相加，得到输出矩阵 S 的一个元素。
- 算法的时间复杂度为 $O(\frac{n^4}{w})$，其中 w 表示向量寄存器的长度。通过使用向量寄存器，可以显著提高计算效率。

综上所述，通过使用向量寄存器和向量指令进行 SIMD 优化，可以大大提高矩阵乘法和卷积运算的计算效率，从而加速深度学习模型的训练速度。在实际应用中，我们可以根据具体情况选择不同的优化方法，以达到最优的计算效率。基于上述的说明方法，我们可以写出对应的算法流程图，如算法6所示：

3.2.3 Pthread&OpenMP 设计思路

在大型数据集的计算任务中，如果仅使用单线程进行计算，可能会导致计算速度过慢。通过利用多线程技术，可以将计算任务分配到多个线程中并行执行，从而显著提高计算速度。

我们可以使用 C++ 标准库中的线程（pthread 库）来进行多线程优化，从而实现卷积计算的并行化。使用多线程可以加快计算速度，尤其是在处理大型数据集时更为明显。以下则是我们对算法优化的设计思想：

- 首先，定义一个名为 ConvolutionData 的结构体，用于存储每个线程的计算数据。
- 实现一个名为 convolution_thread 的函数，这个函数将在每个线程中运行。它从传递给它的结构体中读取数据，并计算卷积的一部分。在 main 函数中，生成随机的信号和卷积核，并分配线程和线程数据。

Algorithm 6 卷积运算 SIMD 优化版

Input: 输入矩阵 I 和卷积核 K
Output: 输出矩阵 S
$m \leftarrow$ 输入矩阵 I 的行数
$n \leftarrow$ 输入矩阵 I 的列数
$k \leftarrow$ 卷积核 K 的大小
初始化输出矩阵 S 为 $(m-k+1) \times (n-k+1)$ 的零矩阵
使用向量寄存器，将卷积核 K 展开为一维向量 K_{vec}
for $i \leftarrow 1$ **to** $m-k+1$ **do**
  for $j \leftarrow 1$ **to** $n-k+1$，以 w 为步长 **do**
    对于输入矩阵 I 中 (i,j) 位置的 $k \times k$ 的子矩阵，使用向量指令，将其展开为一维向量 I_{vec} 使用向量指令，计算 $I_{vec} \times K_{vec}$，得到一组乘积向量
    使用向量指令，将乘积向量的所有元素相加，得到 $S_{i,j}$ 的值
    将 $S_{i,j}$ 的值写入输出矩阵 S 中对应的位置
  end
end

- 根据线程数量将信号分割成等份，为每个线程分配一个数据片段。
- 创建并运行线程，然后等待所有线程完成计算。
- 最后，输出每个线程的用时，并计算总的计算耗时。

Algorithm 7 AVX2 Optimized Convolution with Multi-threading

Input: 输入信号 *signal*, 卷积核 *kernel*
Output: Result 向量 *result*
signal_length $\leftarrow$ length of *signal*
kernel_length $\leftarrow$ length of *kernel*
kernel_aligned_length $\leftarrow \lceil \frac{kernel\_length}{8} \rceil \times 8$
初始化 *result* 向量置 0
for *each thread* **do**
  for $i \leftarrow$ *start* **to** *end* **do**
    for $j \leftarrow 0$ **to** *kernel_aligned_length* ***step*** *8* **do**
      result_idx $\leftarrow i + j$
      if $j <$ *kernel_length* **then**
        signal_reg $\leftarrow$ _mm256_set1_ps(*signal*[i])
        kernel_reg $\leftarrow$ _mm256_loadu_ps(&*kernel*[j])
        mul_reg $\leftarrow$ _mm256_mul_ps(*signal_reg*, *kernel_reg*)
        add_reg $\leftarrow$ _mm256_loadu_ps(&*result*[*result_idx*])
        add_reg $\leftarrow$ _mm256_add_ps(*add_reg*, *mul_reg*)
        _mm256_storeu_ps(&*result*[*result_idx*], *add_reg*)
      end
    end
  end
end

3.2.4 MPI 设计思路

我们知道，矩阵的操作是可以并行化的，包括池化，卷积，矩阵乘法等，于是我们考察卷积操作的并行化，通过 MPI 的集合通信操作，可以将局部计算结果进行有效的汇总，确保最终的卷积结果的正

确性。他的正确性和设计思想是显然的，对于算法的正确性如下所示：

- 卷积操作是可并行的，因为每个输出元素可以独立计算，不需要依赖其他元素。
- 使用 MPI 进行优化时，可以将输入数据和卷积核分割成多个块，并分配给不同的 MPI 进程进行计算。
- 每个 MPI 进程只处理自己分配的块，计算出局部的卷积结果。
- 最后，使用 MPI 的集合通信操作将所有进程的局部卷积结果汇总到主进程，得到最终的卷积结果。

不过我们需要注意的是：卷积操作的 MPI 优化在实现时需要考虑数据的划分和通信的开销，确保任务的划分合理，并且在通信过程中进行数据的同步和汇总，以保证最终结果的正确性。同时，还需要针对具体的硬件和并行计算环境进行优化，以提高计算性能和效率。针对上述的讨论结果，我们可以给出算法流程图8, 如下所示：

Algorithm 8 卷积操作的 MPI 优化算法

Input: 输入数据和卷积核
Output: 卷积结果
将输入数据和卷积核广播给所有 MPI 进程
将输入数据和卷积核分割为多个块，并分配给各个 MPI 进程
for 每个 *MPI 进程* **do**
    在本地计算块的局部卷积结果
end
使用 MPI 的集合通信操作，将局部卷积结果汇总到主进程
主进程输出最终的卷积结果

3.2.5 GPU 设计思路

卷积操作是 CNN 中的重要组成部分，但由于大量的矩阵运算往往需要极高的算力要求，我们发现：卷积操作的本质是矩阵的操作，而矩阵的操作是可以并行化的，于是我们考察卷积操作的并行化，我们可以利用 SYCL 进行 GPU 编程实现卷积操作可以充分发挥 GPU 的并行计算能力，加速计算过程。以下是卷积操作利用 SYCL 的 GPU 编程的一般实现思路：

- 首先我们需要使用 SYCL 编写内核函数，该函数将在 GPU 上执行卷积操作。内核函数通常使用 parallel_for 来实现并行计算。
- 然后我们需要将输入数据和卷积核划分为合适的工作组和工作项。每个工作项负责计算输出数据的一个像素点，工作组则负责计算一部分输出数据。
- 在内核函数中，我们需要合理利用局部内存和私有内存来存储中间结果和临时数据，以减少对全局内存的访问。计算每个工作项的全局 ID 和局部 ID，以及像素位置的偏移量，来确定输入数据和卷积核的访问位置。
- 同时根据卷积操作的定义，对输入数据和卷积核进行逐元素相乘，并累加到输出数据中的相应位置。完成卷积操作后，将输出数据从 GPU 的全局内存中复制回主机内存，以便进一步处理或后续的计算。

Algorithm 9 GPU 优化卷积

输入: input, kernel, inputSize, kernelSize
输出: output

创建一个 GPU 队列
 定义本地工作组大小 localSize
 计算工作组数量 numGroups
 创建输入、卷积核和输出的缓冲区

 使用队列提交任务，其中：读取输入数据、卷积核到相应的访问器
  创建局部内存访问器 localMemory

  并行计算卷积操作，其中：获取本地 ID、组 ID 和全局 ID

  在局部内存中加载输入数据，包括边界处理

  同步工作项

  执行卷积计算，使用局部内存中的数据和卷积核

 将计算结果写入输出缓冲区，注意边界处理

 等待队列执行完毕

3.3 反向传播设计思路

3.3.1 朴素算法设计思路

反向传播（Backpropagation）算法的核心思想是使用链式法则计算损失函数相对于神经网络权重和偏置的梯度，然后使用梯度下降方法更新参数以减小损失函数的值。下面是反向传播算法的设计到的步骤：

- 前向传播：首先，将输入数据传递给神经网络，计算每一层神经元的输出（激活值）。在每一层，神经元的输入是上一层神经元输出与权重矩阵的乘积，加上偏置，然后通过激活函数计算输出。
- 计算损失：使用损失函数（例如均方误差）来衡量神经网络输出与目标值之间的差异。
- 反向传播误差：从输出层开始，计算每个神经元的误差。误差是损失函数对神经元输出的梯度。对于输出层，误差可以直接计算。对于其他隐藏层，需要根据后面一层的误差和权重计算当前层的误差。在计算误差时，需要使用链式法则和激活函数的导数。
- 更新权重和偏置：根据计算出的误差和梯度下降方法，更新神经网络的权重和偏置。更新的幅度由学习率控制。
- 迭代：重复以上步骤，直到达到预定的迭代次数或满足其他收敛条件。

事实上，反向传播的最终目的是不断的迭代每一层神经元的参数值，使其与我们的数据的误差达到最小。此外，通过链式法则，我们可以知道，如果是多个函数复合 (在此处为多层神经网络相连接)，基于链式法则，则我们最终结果相对于某一层的梯度，实际上是上游导数值与当前层导数值相乘得到

的，那么以此类推，我们便可以通过不断迭代来计算最终的误差相对于输入的导数。基于上述我们讨论的算法，我们可以设计出如下所示的算法流程图。

Algorithm 10 单层全连接神经网络训练算法

Input: 输入数据集 X, 目标输出 $\hat{Y}$, 学习率 η, 迭代次数 $epochs$
Output: 训练好的权重矩阵 W 和偏置向量 b
初始化权重矩阵 W 和偏置向量 b;
for $i \leftarrow 1$ *to* $epochs$ **do**
  for 每个输入样本 x_j 和目标输出 $\hat{y}j$ **do**
    计算中间层的输出 $z_j \leftarrow Wx_j + b$
    对中间层的输出 z_j 应用激活函数，得到 $a_j \leftarrow f(z_j)$
    计算输出层的输出 $y_j \leftarrow \text{softmax}(a_j)$
    计算损失函数 $L_j \leftarrow -\sum_k \hat{y}jk \log(y_{jk})$
    计算损失函数对中间层输出的梯度 $\delta_j \leftarrow (\hat{y}_j - y_j)W^T \odot f'(z_j)$
    计算损失函数对权重矩阵和偏置向量的梯度 $\nabla_W L_j \leftarrow x_j \delta_j^T, \nabla_b L_j \leftarrow \delta_j$
    使用梯度下降法更新权重矩阵和偏置向量 $W \leftarrow W + \eta \nabla_W L_j, b \leftarrow b + \eta \nabla_b L_j$
  end
end

3.3.2 SIMD 设计思路

我们知道，SIMD 可以通过使用向量寄存器来提高访问效率，我们在这里设计的算法和上面的朴素思路一样，都是只实现一个全连接神经网络，模拟一次训练过程，通过调整 epoch 和输入输出的维度来测试不同数据规模下算法的执行效率。

我们在这里做出了以下任务：

- 定义了使用 AVX 指令集进行 SIMD 优化的 sigmoid 激活函数和其导数函数。这些函数接受 256 位双精度浮点向量作为输入，并使用 AVX 指令进行向量化计算。
- 实现了 backwardPropagationAVX 函数，该函数使用 AVX 指令集进行 SIMD 优化的神经网络反向传播。在这个函数中，利用 AVX 指令进行向量化计算，加速了隐藏层和输出层的计算过程。
- 在主函数中，初始化了神经网络的权重和偏置，并生成了训练数据集。然后，通过多个训练轮次调用 backwardPropagationAVX 函数进行神经网络的训练。最后，输出训练耗时。

具体的算法流程图，可以参考上面原始算法流程图，完整的代码请参考文后附带的 Github 项目仓库中的文件。

3.3.3 Pthread&OpenMP 设计思路

由于神经网络算法较长，我们仅考虑实现一个简单的多层感知器（MLP）神经网络，并使用 pthread 去进行多线程训练。多线程训练可以充分利用多核处理器的计算能力，从而提高神经网络训练的速度。以下是代码的设计思想和算法流程图：

- 定义一个 BPData 结构体，用于在多线程之间传递必要的参数。结构体包括输入、目标、各层的权重和偏置，以及每个线程应处理的数据范围（start 和 end）。
- 实现一个 bp_thread 函数，该函数执行神经网络的一部分训练过程。bp_thread 接受一个指向 BPData 结构体的指针作为参数，并对结构体中指定范围的数据进行前向传播和反向传播。在反向传播过程中，神经网络的权重和偏置被更新。

- 在 main 函数中，首先生成随机训练数据（输入和目标），然后初始化权重和偏置。接下来，对于每个训练轮次（epoch），使用 pthread_create 创建多个线程，并将 bp_thread 函数作为线程入口。每个线程都会接收一个 BPData 结构体指针，指向一个包含其应处理的数据范围的结构体。线程创建后，使用 pthread_join 等待所有线程完成训练。
- 最后，在所有线程完成训练后，输出每个线程的用时和总用时。

Algorithm 11 Multi-threaded BP Neural Network Training

Input: Input samples *inputs*, target values *targets*
初始化权重 *hidden_weights* 和 *output_weights*
初始化偏差 *hidden_bias* 和 *output_bias*
epochs ← number of training epochs
num_threads ← number of threads
for *epoch* ← 1 **to** *epochs* **do**
  for *each thread* **do**
    for *i* ← *start* **to** *end* **do**
      Get input sample and target values
      Perform forward propagation
      Calculate output layer and hidden layer errors
      Update weights and biases
    end
  end
end

3.3.4 MPI 设计思路

我们考虑使用 MPI 进行优化，我们结合 MPI 的算法逻辑和 BP 算法特点，整理出了以下的操作步骤：

- 初始化 MPI 环境，获取总进程数和当前进程的标识符。
- 将训练数据和权重参数分发给各个进程，使每个进程都可以独立计算一部分数据的梯度更新。
- 在每个进程中执行反向传播算法，计算局部的权重和偏置的梯度。
- 使用 MPI 的全局归约操作，将各个进程的局部梯度进行汇总，得到全局的权重和偏置的梯度。
- 更新全局的权重和偏置参数。
- 重复步骤 3-5，直到达到预定的训练迭代次数。
- 结束 MPI 环境。

3.4 池化操作设计思路

3.4.1 朴素算法设计思路

池化操作是神经网络中常用的一种操作，用于减小特征图的空间尺寸并提取特征。最大池化没有明确的公式，但可以通过以下方式表示：给定输入数据 X 和池化区域大小 k，则最大池化的输出 Y 可以表示为：

$$Y_{i,j,c} = \max\left(X_{(i\cdot k):(i\cdot k+k),(j\cdot k):(j\cdot k+k),c}\right)$$

其中，i 和 j 表示输出的行和列索引，c 表示通道索引。$X_{(i\cdot k):(i\cdot k+k),(j\cdot k):(j\cdot k+k),c}$ 表示输入数据的池化区域。

针对上述的讨论过程，我们给出算法流程图12：

Algorithm 12 朴素算法的池化操作

输入: 输入数据
输出: 池化结果

```
for 每个输入通道 do
    for 每个池化窗口 do
        // 计算池化窗口内的最大值
        max_value ← 池化窗口内的最大值
            // 将最大值添加到池化结果
        将 max_value 添加到池化结果中
    end
end
```

3.4.2 SIMD 设计思路

我们考虑对池化进行 SIMD 的 AVX 指令集的优化：针对每个 SIMD 寄存器可以同时处理多个数据元素的特点，将数据划分为 SIMD 寄存器大小的块进行计算。我们可以使用 SIMD 指令集提供的函数和操作来执行向量化计算，例如 _mm256_loadu_pd、_mm256_storeu_pd 和 _mm256_max_pd 等。并且利用 SIMD 寄存器并行计算多个数据元素的最大值，以提高计算效率和性能。之后我们处理剩余的不足 SIMD 寄存器大小的数据元素，确保所有数据都被正确处理。通过使用 SIMD 优化，可以利用 SIMD 指令集的并行性，提高池化操作的计算速度和效率。

我们在下面给出 SIMD 优化的 Pooling 操作的部分代码，完整的代码可以参考 Github 项目链接：

池化操作 SIMD 优化部分

```
std::vector<double> performPoolingSIMD(const std::vector<double>& data) {
    int num_samples = data.size();
    int simd_size = num_samples / 4 * 4;
    std::vector<double> pooled_data(num_samples);
    __m256d max_values = _mm256_loadu_pd(&data[0]);
    for (int i = 4; i < simd_size; i += 4) {
        __m256d current_values = _mm256_loadu_pd(&data[i]);
        max_values = _mm256_max_pd(max_values, current_values);
    }
    // 将SIMD寄存器中的数据存储回数组
    _mm256_storeu_pd(&pooled_data[0], max_values);
    // 处理剩余的不足4个元素
    for (int i = simd_size; i < num_samples; ++i) {
        if (data[i] > pooled_data[i]) {
            pooled_data[i] = data[i];
        }
    }
    return pooled_data;
```

```
}
```

3.4.3 Pthread&OpenMP 设计思路

接下来我们考虑使用多线程技术对池化进行优化：我们可以将数据划分为多个子任务，并为每个子任务创建一个线程来执行池化操作。接下来在每个线程中独立计算部分数据的池化结果，通过参数传递数据的子集以及池化操作的结果。

用多线程并行计算不同数据子集的池化结果,提高计算效率和性能。我们主要的优化是使用 Pthread 库中的线程创建函数 pthread_create 和线程等待函数 pthread_join 来创建和管理线程。我们需要在主线程中等待所有子线程的执行完成，以保证所有池化操作都已完成，最后返回池化操作的结果。

对于上述讨论的思想，我们同样给出部分的代码，完整代码可以参考 Github 项目链接。

池化操作 Pthread 优化部分

```
std::vector<double> performPoolingPThread(const std::vector<double>& data, int
    num_threads) {
    int num_samples = data.size();
    std::vector<double> pooled_data(num_samples);
    std::vector<pthread_t> threads(num_threads);
    std::vector<ThreadArgs> thread_args(num_threads);
    int chunk_size = num_samples / num_threads;
    int start_index = 0;
    int end_index = chunk_size;
    for (int i = 0; i < num_threads; ++i) {
        thread_args[i].data = &data;
        thread_args[i].pooled_data = &pooled_data;
        thread_args[i].start_index = start_index;
        thread_args[i].end_index = end_index;
        pthread_create(&threads[i], nullptr, performPoolingPThread, &thread_args[i]);
        start_index = end_index;
        end_index = (i == num_threads - 2) ? num_samples : end_index + chunk_size;
    }
    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], nullptr);
    }
    return pooled_data;
}
```

3.4.4 MPI 设计思路

我们在考虑设计算法前，最重要的思路就是证明算法的正确性，没有正确性证明的算法，效率再高都无济于事，我们考虑证明 MPI 优化池化算法的可能性：首先我们假设：

- 假设输入特征图被正确地分布到各个 MPI 进程，确保每个进程上的特征图是完整且一致的。
- 各个 MPI 进程之间可以进行通信，并且 MPI 操作的结果符合预期。

接下来我们来证明算法的正确性：

- 特征提取的正确性：在每个 MPI 进程上，通过池化操作提取局部特征。由于每个进程上的特征图是完整的，因此在每个进程上提取的局部特征是准确的。
- 特征合并的正确性：通过 MPI 操作，在各个进程之间进行特征的合并，将局部特征合并得到全局特征。MPI 操作的正确性保证了合并操作的准确性。
- 特征传播的正确性：将全局特征传播到所有 MPI 进程，确保所有进程都使用相同的全局特征。

综上所述，通过 MPI 优化池化操作，在正确分布特征图、准确提取局部特征、正确合并特征和传播全局特征的前提下，可以保证最终得到准确的特征表示。这证明了 MPI 优化后的池化操作在神经网络训练过程中的正确性。

3.5 嵌入层设计思路

3.5.1 朴素算法设计思路

Algorithm 13 朴素版嵌入

输入: 数据, 嵌入向量
输出: 结果

$num\_data \leftarrow$ 数据大小
$embedding\_size \leftarrow$ 嵌入向量维度
$result\_size \leftarrow$ 结果维度

for $i \leftarrow 0$ **to** num_data **do**
  for $j \leftarrow 0$ **to** $embedding\_size$ **do**
    for $k \leftarrow 0$ **to** $result\_size$ **do**
      $result[i][k] \leftarrow result[i][k] + embeddings[j][k] \times data[i]$
    end
  end
end

朴素版的 embedding 层只需要利用我们上面说明的算法原理进行实现即可，在这里我们给出算法流程图，全部的代码和项目文件请参考后文附的 GitHub 项目链接。

3.5.2 SIMD 设计思路

我们将 Embedding 使用 SIMD 优化，在这里，我们将给出 SIMD 优化的 embedding 部分的代码，如下所示：

Embedding 操作 SIMD 优化部分

```
1        std::vector<std::vector<double>> forwardBatch(const
             std::vector<std::vector<int>>& batchInput) {
2        int batchSize = batchInput.size();
3        int inputSize = batchInput[0].size();
4        std::vector<std::vector<double>> output(batchSize,
             std::vector<double>(embeddingSize, 0.0));
5        for (int b = 0; b < batchSize; b += 4) {
6            for (int i = 0; i < inputSize; ++i) {
```

```
                __m256d sum_vec = _mm256_setzero_pd();
                for (int j = 0; j < embeddingSize; ++j) {
                    __m256d input_vec = _mm256_set_pd(
                        weights[batchInput[b + 3][i]][j],
                        weights[batchInput[b + 2][i]][j],
                        weights[batchInput[b + 1][i]][j],
                        weights[batchInput[b][i]][j]
                    );
                    sum_vec = _mm256_add_pd(sum_vec, input_vec);
                }
                _mm256_store_pd(&output[b][i], sum_vec);
            }
        }
        return output;
    }
```

在上述代码中，我们使用了 AVX 指令集对嵌入操作进行了优化。在 forward 函数中，我们使用 SIMD 指令集的 _mm256 函数来实现向量化计算，从而提高计算效率。具体而言，我们将输入数据分为 4 个整数一组，然后使用 AVX 指令集进行并行计算，将结果存储在输出向量中。这样可以加速嵌入操作的计算过程。

3.5.3 Pthread&OpenMP 设计思路

我们在这里给出大致的前向传播 Pthread 的优化代码，完整的代码可以参考 Github 项目仓库：

Embedding 操作 Pthread 优化部分

```
std::vector<std::vector<double>> forwardBatch(const
    std::vector<std::vector<int>>& batchInput, const
    std::vector<std::vector<double>>& weights, int embeddingSize, int numThreads)
    {
    int batchSize = batchInput.size();
    std::vector<std::vector<double>> output(batchSize,
        std::vector<double>(embeddingSize, 0.0));
    std::vector<pthread_t> threads(numThreads);
    std::vector<ThreadParams> threadParams(numThreads);
    for (int i = 0; i < numThreads; ++i) {
        threadParams[i] = {i * batchSize / numThreads, (i + 1) * batchSize /
            numThreads, &batchInput, &output, &weights, embeddingSize};
        pthread_create(&threads[i], nullptr, processBatch, &threadParams[i]);
    }
    for (int i = 0; i < numThreads; ++i) {
        pthread_join(threads[i], nullptr);
    }
    return output;
}
```

3.5.4 MPI 设计思路

在我们第一节中，介绍了嵌入操作的主要原理，我们发现嵌入操作的实质便是通过矩阵乘法将一系列输入数据嵌入为若干维度的向量，对于这种计算密集型的任务，我们显而易见的想要对其使用 MPI 优化，我们下面来考虑使用 MPI 优化的算法正确性，首先我们考虑嵌入结果的正确性：

- 每个进程根据其负责的工作范围执行嵌入操作，根据给定的输入数据和嵌入矩阵，更新局部结果矩阵的对应元素。
- 每个线程在自己的工作范围内并行执行嵌入操作，可以通过 SIMD 加速计算。
- 在进行结果的全局归约之后，每个进程都会通过 MPI_Allreduce 操作将局部结果与其他进程的结果进行求和，得到全局一致的结果矩阵。

接下来是结果的一致性：

- 每个进程在进行嵌入操作时，只更新自己负责工作范围内的结果矩阵，不会影响其他进程。
- 在进行结果的全局归约时，使用 MPI_Allreduce 操作将每个进程的局部结果与其他进程的结果进行求和，确保所有进程最终得到的结果矩阵是一致的。

综合以上两个方面的考虑，可以得出该嵌入操作算法使用 MPI 进行优化后的正确性。那么接下来我们可以通过算法的原理，给出算法的流程图14：

Algorithm 14 使用 MPI 的并行嵌入

输入: 数据, 嵌入向量
输出: 结果

```
for i ← start to end do
    for j ← 0 to embedding_size do
        for k ← 0 to result_size do
            result[i][k] ← result[i][k] + embeddings[j][k] × data[i]
        end
    end
end
for i ← 0 to num_data do
    for k ← 0 to result_size do
        // 全局归约
        MPI_Allreduce(result[i][k], MPI_SUM)
    end
end
```

3.6 PCA 设计思路

3.6.1 朴素算法设计思路

Algorithm 15 朴素 PCA 算法

输入: 数据, 主成分数
输出: 降维结果
for $i \leftarrow 0$ **to** num_data **do**
    // 计算数据的均值
    $mean \leftarrow$ 计算数据在每个维度上的均值
    将数据每个维度上的值减去对应维度的均值
    $cov\_matrix \leftarrow$ 计算数据的协方差矩阵
    $eigenvalues, eigenvectors \leftarrow$ 计算协方差矩阵的特征值和特征向量
    $selected\_eigenvectors \leftarrow$ 选择特征值最大的几个特征向量
    $result[i] \leftarrow$ 将数据点 i 投影到主成分上
end

朴素版的 PCA 只需要利用我们上面说明的算法原理进行实现即可，在这里我们给出算法流程图，全部的代码和项目文件请参考后文附的 GitHub 项目链接。

3.6.2 SIMD 设计思路

我们考虑 SIMD 优化 PCA 的具体过程：我们 PCA 主成分分析可以在数据中心化和协方差矩阵部分进行对应的优化，所以在这里，我们针对数据中心化和协方差矩阵计算进行了 SIMD 优化。通过使用 AVX 指令集的 256 位向量数据类型 ___m256d，我们可以在同一指令中处理多个数据元素，从而实现并行计算。使用 SIMD 指令集可以显著加速代码的执行，提高计算效率。我们将给出数据中心化和协方差矩阵部分的 SIMD 优化代码：

协方差矩阵 SIMD 优化

```
std::vector<std::vector<double>> computeCovarianceMatrixSIMD(const
    std::vector<std::vector<double>>& data) {
    int rows = data.size();
    int cols = data[0].size();
    std::vector<std::vector<double>> covariance(cols, std::vector<double>(cols, 0.0));
    for (int i = 0; i < cols; ++i) {
        for (int j = i; j < cols; ++j) {
            double sum = 0.0;
            for (int k = 0; k < rows; k += 4) {
                __m256d data_vec1 = _mm256_loadu_pd(&data[k][i]);
                __m256d data_vec2 = _mm256_loadu_pd(&data[k][j]);
                __m256d mul_result = _mm256_mul_pd(data_vec1, data_vec2);
                __m256d sum_result = _mm256_hadd_pd(mul_result, mul_result);
                __m128d sum_low = _mm256_extractf128_pd(sum_result, 0);
                __m128d sum_high = _mm256_extractf128_pd(sum_result, 1);
                __m128d sum_final = _mm_add_pd(sum_low, sum_high);
                double sum_temp;
                _mm_store_sd(&sum_temp, sum_final);
                sum += sum_temp;
```

```
            }
            covariance[i][j] = sum / (rows - 1);
            covariance[j][i] = sum / (rows - 1);
        }
    }
    return covariance;
}
```

数据中心化 SIMD 优化部分

```
void centerDataSIMD(std::vector<std::vector<double>>& data, const
    std::vector<double>& mean) {
    int rows = data.size();
    int cols = data[0].size();
    for (int j = 0; j < cols; ++j) {
        double mean_value = mean[j];
        __m256d mean_vec = _mm256_set1_pd(mean_value);
        for (int i = 0; i < rows; i += 4) {
            __m256d data_vec = _mm256_loadu_pd(&data[i][j]);
            __m256d centered_vec = _mm256_sub_pd(data_vec, mean_vec);
            _mm256_storeu_pd(&data[i][j], centered_vec);
        }
    }
}
```

3.6.3 Pthread&OpenMP 设计思路

在 Pthread 优化的代码中，我们使用多线程并行计算数据的均值。创建了多个线程，并将数据划分为若干部分进行并行计算，然后将计算结果合并。通过使用 Pthread 库提供的线程创建和等待函数，可以实现多线程计算并提高计算效率。不过注意的是，在多线程环境中，我们需要注意线程之间的数据访问和同步问题，确保线程安全性，在之前 Pthread 的实验中，我们就遇到了数据不一致和数据同步方面的问题，通过查阅了相应的资料才得以解决。

同样的，我们在这里给出优化部分的代码：

Pthread 优化 PCA

```
void* computeMeanThread(void* arg) {
    ThreadData* threadData = static_cast<ThreadData*>(arg);
    const std::vector<std::vector<double>>& data = threadData->data;
    std::vector<double>& mean = threadData->mean;
    int rows = data.size();
    int cols = data[0].size();
    for (int j = threadData->start; j < threadData->end; ++j) {
        double sum = 0.0;
        for (int i = 0; i < rows; ++i) {
            sum += data[i][j];
        }
        mean[j] = sum / rows;
```

```
    }
    pthread_exit(nullptr);
}
std::vector<double> computeMean(const std::vector<std::vector<double>>& data) {
    int cols = data[0].size();
    std::vector<double> mean(cols, 0.0);
    int numThreads = 4;  // 线程数
    std::vector<pthread_t> threads(numThreads);
    std::vector<ThreadData> threadData(numThreads);
    int step = cols / numThreads;
    for (int i = 0; i < numThreads; ++i) {
        int start = i * step;
        int end = (i == numThreads - 1) ? cols : (i + 1) * step;
        threadData[i] = {start, end, data, mean};
        pthread_create(&threads[i], nullptr, computeMeanThread, &threadData[i]);
    }
    for (int i = 0; i < numThreads; ++i) {
        pthread\_join(threads[i], nullptr);
    }
    return mean;
}
```

3.6.4 MPI 设计思路

正如我们在前面算法介绍部分介绍的那样：PCA 是一种常用的降维算法，用于将高维数据映射到低维空间。PCA 的目标是通过线性变换找到数据中的主成分，即方差最大的方向。PCA 通过计算数据的协方差矩阵的特征值和特征向量，选择最大的几个特征值对应的特征向量作为主成分，将数据投影到这些主成分上，实现降维。我们发现，事实上 PCA 的本质仍然是矩阵的运算，既然涉及到了矩阵的运算，我们便显而易见的想要通过并行化技术对计算过程进行加速优化。

Algorithm 16 使用 MPI 的并行 PCA

输入: 数据, 嵌入向量
输出: 结果

```
for i ← start to end do
    mean ← 计算数据在每个维度上的均值
    将数据每个维度上的值减去对应维度的均值
    cov_matrix ← 计算数据的协方差矩阵
    eigenvalues, eigenvectors ← 计算协方差矩阵的特征值和特征向量
    selected_eigenvectors ← 选择特征值最大的几个特征向量
    result[i] ← 将数据点 i 投影到主成分上
end
for i ← 0 to num_data do
    // 全局归约
    MPI_Allreduce(result[i], MPI_SUM)
end
```

上述算法中，我们首先计算数据在每个维度上的均值，并将数据减去均值，以实现数据的中心化。

然后计算中心化后数据的协方差矩阵。接着，我们计算协方差矩阵的特征值和特征向量，并选择最大的几个特征值对应的特征向量作为主成分。最后，将数据点投影到选定的主成分上，得到降维后的结果。在并行化过程中，每个进程根据其负责的工作范围执行计算，并通过 MPI_Allreduce 操作将局部结果与其他进程的结果进行求和，以得到全局一致的结果矩阵。

3.6.5 GPU 设计思路

我们发现，事实上 PCA 的本质仍然是矩阵的运算，既然涉及到了矩阵的运算，我们便显而易见的想要通过并行化技术对计算过程进行加速优化。上面我们使用了 MPI，Pthread，SIMD 等优化手段，在这里，我们采用的是 GPU 加速运算。

我们在这里利用 GPU 的部分主要体现在数据并行性的利用。正如我们所说的，SYCL 是一个基于 C++ 的异构编程模型，可以让我们在多种类型的处理器（如 CPU，GPU，FPGA 等）上编写并行程序。

在这段代码中，我们可以使用 SYCL 运行时库来将一些数据并行的计算任务分发给 GPU 来完成。我们可以在以下两个方面使用并行技术优化：对于我们所讨论的算法，我们给出如下的算法流程图：

- 特征平均值的并行计算：每个 GPU 线程都计算了部分样本的平均值，然后使用了一种叫做“归约”的技术在同一工作组的线程之间合并了这些部分平均值。这种归约技术充分利用了 GPU 的线程间通信能力，从而有效地减少了计算平均值所需的时间。
- 并行地从每个特征中减去平均值：计算出平均值后，每个 GPU 线程处理一部分样本，并从每个特征中减去对应的平均值。这一步操作是完全并行的，充分利用了 GPU 的并行计算能力，从而显著提高了运行速度。

Algorithm 17 基于 GPU 的主成分分析（PCA）

输入: 数据 data, 结果存储 result, 样本数量 numSamples, 特征数量 numFeatures
输出: 已减去均值的数据

并行计算每个特征的均值
for *GPU* 中的每个线程 **do**
  $sum \leftarrow 0$
  for 每个被分配到的样本 i **do**
    for 在 $numFeatures$ 中的每个特征 j **do**
      $sum \leftarrow sum + data[i * numFeatures + j]$
    end
  end
  执行工作组内的归约，计算所有部分和的总和
  对于每个工作组的第一个线程，$result[groupId] \leftarrow sum/numSamples$
end
并行从每个特征中减去均值
for *GPU* 中的每个线程 **do**
  for 每个被分配到的样本 i **do**
    for 在 $numFeatures$ 中的每个特征 j **do**
      $data[i * numFeatures + j] \leftarrow data[i * numFeatures + j] - result[groupId]$
    end
  end
end

4 实验结果分析

在这一节中，我们将分别对上述六种算法的不同实现的程序运行时间进行展示，并分析数据背后的原因，具体硬件方面的解释可以参考下一节的 Profiling 分析：

4.1 矩阵乘法结果分析

我们测定了不同数据规模下矩阵乘法的不同算法的程序运行时间，具体来说，如下表1所示：

表 1: 矩阵乘法性能比较

参数指标	运行时间（秒）			
	origin	AVX2 优化	Pthread	MPI
$n = m = p = 800$	35.72	1.77	9.42	8.93
$n = m = p = 700$	23.67	1.25	6.38	5.56
$n = m = p = 600$	14.64	0.74	3.99	3.98
$n = m = p = 500$	8.64	0.43	2.43	2.59
$n = m = p = 400$	4.43	0.21	1.29	1.47
$n = m = p = 300$	1.88	0.09	0.52	0.49
$n = m = p = 200$	0.56	0.026	0.16	0.26
$n = m = p = 100$	0.07	0.004	0.024	0.045

为方便观测，我们将上述的结果整理为可视化的图：图4.3展示了不同算法的程序运行时间，图4.4展示了不同算法相较于朴素算法的加速比，我们可以得到以下的结论：

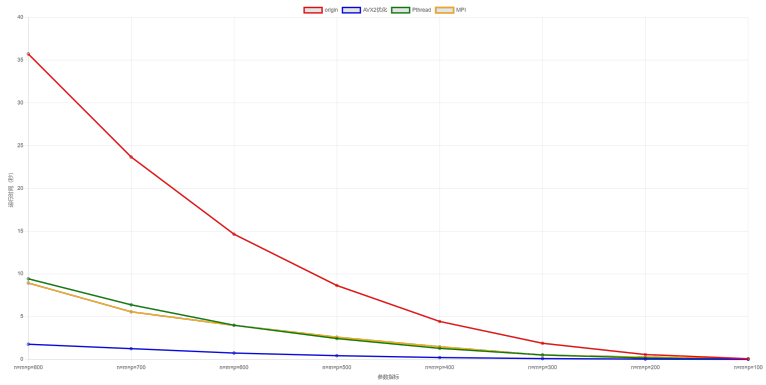

图 4.3: 不同算法的程序运行时间

- 原始算法（origin）：原始算法的运行时间随着参数指标 $n = m = p$ 的增大而增加，呈现出近似线性增长的趋势。这是因为矩阵乘法的时间复杂度为 $O(n^3)$，当参数指标增大时，计算量增加，导致运行时间增长。
- AVX2 优化：AVX2 优化相对于原始算法，在所有参数指标下都实现了显著的加速。例如，在 n=m=p=800 的情况下，AVX2 优化的运行时间仅为原始算法的约 5%。这是由于 AVX2 指令集的使用，能够并行处理多个数据，从而加快矩阵乘法的计算速度。
- Pthread 和 MPI：Pthread 和 MPI 在较大的参数指标下实现了一定程度的加速。例如，在 n=m=p=800 的情况下，Pthread 和 MPI 的运行时间分别为原始算法的约 26% 和 25%。这是由于并行化的方

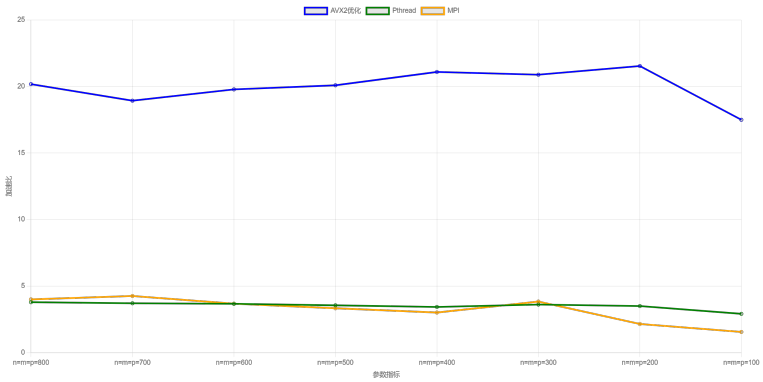

图 4.4: 不同算法加速比

式，使用多线程（Pthread）或分布式计算（MPI）进行矩阵乘法运算，将计算任务分配给多个计算单元进行并行计算，从而提高了计算效率。

总的来看，AVX2 优化在所有参数指标下都实现了较大的加速比提升，而 Pthread 和 MPI 在较大的参数指标下也能够获得一定的加速效果。然而我们需要注意的是在参数指标较小时，由于线程切换和通信开销等因素的影响，加速比的提升效果有所降低。因此，在选择优化策略时，需要综合考虑问题规模、硬件平台和算法复杂度等因素，选择适合的优化方法以获得最佳性能。

4.2 卷积操作结果分析

我们同样测定了不同输入规模下卷积任务的运行时间，如2所示：值得注意的是，为使表格的可读性增强，我们仅仅给出了 Signal_length 的大小，kernel_length 的长度是 8% 的 signal_length 的长度，例如对于输入 Signal_length 为 100000 时，其 kernel_length 为 8000。我们整理了可视化的图表如下图4.5以及图4.6所示：

表 2: 卷积任务的运行时间（单位：秒）

参数指标		算法				
		Origin	MPI	Pthread	AVX	GPU
signal_length	100000	0.772419	0.194989	0.115987	0.105485	1.97562
	200000	3.1691	0.806252	0.574984	0.474485	6.87412
	300000	7.12401	1.8042	1.04989	1.06939	14.85643
	400000	12.6769	3.18735	1.98786	1.89599	25.56969
	500000	19.8055	4.96612	3.00127	2.96107	40.04005
	600000	28.7963	7.12824	4.26987	4.24801	57.85461
	700000	38.8059	9.70485	6.24531	5.86649	77.69089
	800000	50.7671	12.6991	7.67589	7.76565	101.6363
	900000	66.2713	17.5021	9.97546	9.99211	133.5507
	1000000	91.2357	23.1332	11.99654	12.3811	180.2365

接下来我们观察不同算法在不同信号长度和内核长度下的运行时间：可以看到，随着信号长度和内核长度的增加，所有算法的运行时间也在增加。原始算法（Origin）的运行时间是理论上最长的，其他算法（MPI、Pthread、AVX、GPU）应该相对较短，但是我们可以看到，GPU 优化后的卷积反而要比原始算法的运行成本高，我们考虑原因：

- 性能限制：由于我们 OneAPI 所使用的的 GPU 编程是利用的 Intel 的集成显卡，性能相对较弱，

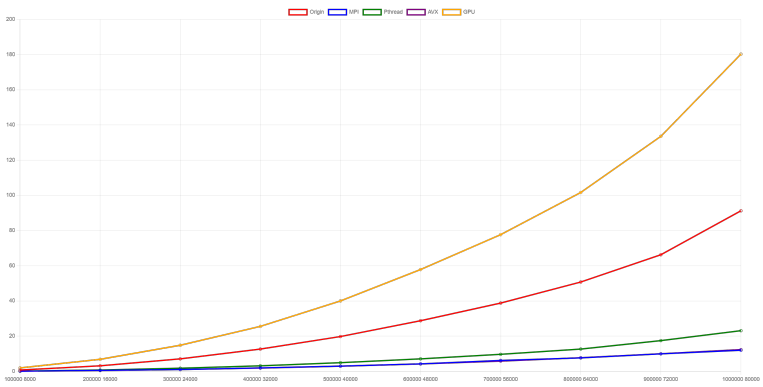

图 4.5: 不同算法的程序运行时间

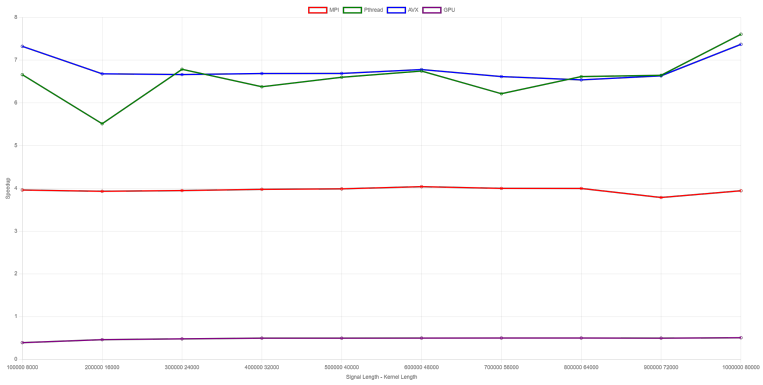

图 4.6: 不同算法加速比

相比于 CPU 而言，频率低，唯一的优势在于功耗较低，所以在我们的任务上就显得不占优势了。

- 通信传输开销：GPU 就和 MPI 一样，同样需要传输开销，在性能低，并且有额外的通信损耗下，GPU 计算比 CPU 慢也是情理之中的事情。我们上面的讨论过程也佐证了这一点。

4.3 反向传播结果分析

在这里，我们实现了单层全连接网络，配有 Sigmoid 激活函数，模拟训练神经网络的过程，在这里我们默认取 epoch=10，即训练 10 轮，input size 为输入数据规模，hidden size 为隐藏层神经元个数 num sample 为输入数据的个数，我们为训练方便起见，仅仅改变 num sample 的大小，测定不同的 num sample 下程序的运行时间，如表3和图4.7和图4.8所示：

我们对于输入大小为 30000、隐藏层大小为 2000 以及不同样本数的情况进行结果分析：

- 我们所有的并行优化算法都能够起到较好的优化作用，其加速比都比较高，其中 Pthread 的加速比较低，仅为 2。
- SIMD 算法在所有样本数下均表现较好，加速比都较高，说明其在向量化计算上具有优势。
- MPI 算法的加速比相对较低，随着样本数的增加略有提升，但仍低于 SIMD 算法算法。

事实上，我们 BP 算法并非计算密集型任务，对于 MPI 的通信开销相对较大，所以我们 SIMD 使用 AVX 指令集的优化效果强于 MPI 也是情理之中的事情；对于 Pthread，我们使用了 thread=4，也就是 4 个线程，理论加速比应该为 4，但是有线程间切换等开销导致了其加速比不可能为理想状态，

表 3: BP 算法性能比较

参数设置			Pthread	Origin	SIMD	MPI
input size	**hidden size**	**num_sample**	**时间**	**时间**	**时间**	**时间**
30000	2000	25	10.9	24.55	2.9	3.6
30000	2000	50	21.26	49.12	4.7	5.2
30000	2000	75	35.06	73.48	8.3	8.9
30000	2000	100	46.67	97.9	10.6	11.3
30000	2000	200	90.86	196.3	21.7	23.5
30000	2000	300	133.42	295.05	30.6	35.3
30000	2000	400	181.9	393.91	41.2	45.6
30000	2000	500	221.98	492.9	45.3	50.2
30000	2000	1000	454.13	979.73	90.5	116.7

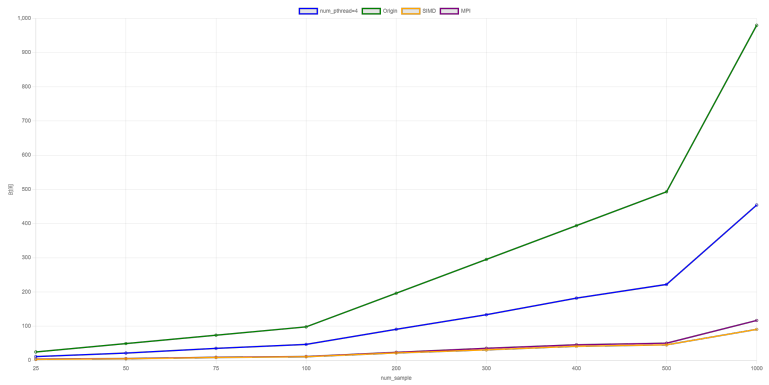

图 4.7: 不同算法的程序运行时间

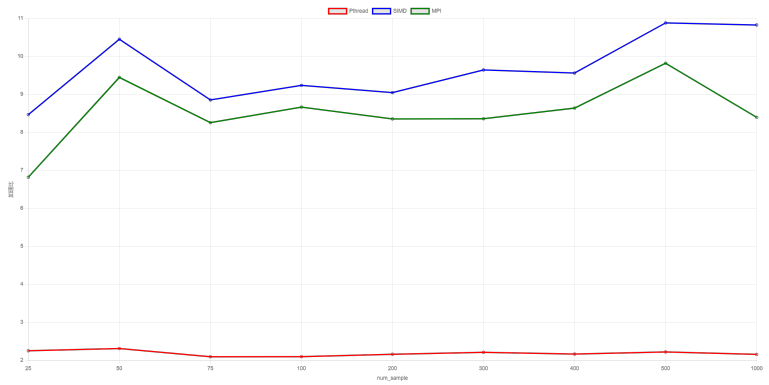

图 4.8: 不同算法加速比

所以加速比为 2 也相对比较合理。此外，对于 Pthread 不同线程数对程序运行时间的影响，我们也做了相应的分析与探索：在数据规模相同的情况下，不断增大 num_pthread，即不断增大线程数，从单线程开始，到 17 个线程。之所以选取 17 个线程是因为实验平台是 I7 Core 11th，具有 8 个核心和 16 个线程数。以探究在数据规模相同时，加速比随核心的变化趋势，并给出了损耗的百分比。如表4和图4.9所示。

表 4: 运行时间、加速比和损耗

num_pthread	时间（s）	加速比	损耗（1 − 实际加速比/理论加速比）
1	108.84	1.00	0.00
2	65.82	1.65	0.175
3	52.8	2.06	0.313
4	45.71	2.38	0.405
5	42.02	2.59	0.482
6	38.64	2.81	0.531
7	34.04	3.19	0.544
8	32.14	3.38	0.578
9	31.57	3.44	0.618
10	30.04	3.62	0.638
11	27.31	3.98	0.638
12	29.74	3.65	0.696
13	30.25	3.59	0.724
14	24.32	4.47	0.681
15	28.36	3.83	0.745
16	22.9	4.75	0.703
17	35.8	3.04	0.821

这些数据中出现的现象的原因是线程的数量和处理器的物理核数之间的关系。当使用更多的线程时，可以实现更高的并行度，从而加速程序的执行。然而，当线程数量大于处理器的物理核心数量时，会发生上下文切换的开销，导致程序执行时间的增加。这个上下文切换的开销可能会抵消通过并行化获得的性能提升，导致加速比达到峰值之后开始下降。

从这组数据中可以看出，在使用 1-8 个线程时，加速比和损耗都有着很好的表现，这是因为这些线程数量不会导致过多的上下文切换；然而，当使用 9 个或更多的线程时，加速比和损耗开始下降，这是因为线程数量超过了处理器的物理核心数量，导致上下文切换的开销开始占据较大的比例。并且，由于实验的环境是 8 核心 16 线程的 CPU，故当并行线程数为 17 时，速度和加速比出现了显著下降的现象，这便是由于超过 16 线程的部分无法及时的并行处理，只能等部分线程空闲再做相应的操作。并且，随着线程数的增加，在数据规模相同的情况下，损耗越来越大，都没有办法达到理论加速比。

此外，由于并行程序的性能通常受到数据访问模式和负载平衡的影响，因此对于不同的数据集和程序实现，可能会出现不同的最优线程数。因此，在进行并行化优化时，需要进行仔细的测试和评估，以找到最佳的线程数量和其他优化策略。

4.4 池化操作结果分析

在这部分中，我们并不进行单独的优化，而是多种优化策略并行处理，例如，我们会在 MPI 的基础上逐渐增加优化策略，探究使用多种优化策略对程序运行时间的影响，我们测定了不同数据规模下程序的运行时间，为了便于分析，我们所有的参数都是成倍增加的，下面我们给出不同参数的含义：

- n, m：输入数据的尺寸。

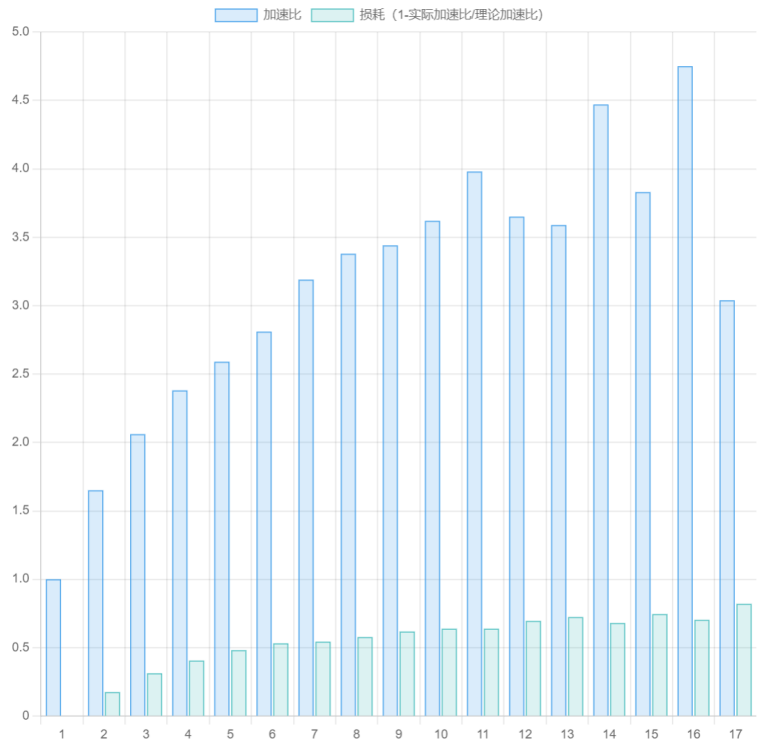

图 4.9: 加速比损耗与线程数关系图

- *pooling*：池化层的大小。

对于程序运行时间，我们整理了以下的可视化的图标，如表5和图4.10和图4.11所示：

表 5: Pooling 算法运行时间比较

n	m	pooling	Origin	MPI	MPI_OpenMP	AVX_OpenMP_MPI
1000	1000	2	0.00500842	0.00443384	0.000726752	0.00201667
2000	2000	4	0.0125016	0.0158015	0.00597195	0.00341615
4000	4000	8	0.0844329	0.00666671	0.0119086	0.0137254
8000	8000	16	0.447501	0.0295291	0.0724601	0.0634052
16000	16000	32	2.25992	0.965222	0.302678	0.224352
32000	32000	64	5.74283	1.65425	0.698356	0.507988

我们考虑对数据进行分析：对于输入大小为 1000x1000、池化大小为 2 的情况，MPI 算法的运行时间为 0.00443384，加速比为 0.00500842 / 0.00443384 = 1.13。MPI_OpenMP 算法的运行时间为 0.000726752，加速比为 0.00500842 / 0.000726752 = 6.89。AVX_OpenMP_MPI 算法的运行时间为 0.00201667，加速比为 0.00500842 / 0.00201667 = 2.48。可以看出，MPI_OpenMP 算法在这种情况下获得了最高的加速比，其次是 AVX_OpenMP_MPI 算法，MPI 算法的加速比相对较低。

随着输入大小和池化大小的增加，MPI 算法的加速比逐渐降低，而 MPI_OpenMP 算法和 AVX 优化算法的加速比相对稳定。这可能是由于 MPI 算法在处理更大规模的数据时，通信开销和负载不均衡的问题逐渐显现，导致加速比下降。

在最大的输入大小为 32000x32000、池化大小为 64 的情况下，MPI 算法的加速比为 5.74283 / 1.65425 = 3.47，MPI_OpenMP 算法的加速比为 5.74283 / 0.698356 = 8.22，AVX_OpenMP_MPI 算法的加速比为 5.74283 / 0.507988 = 11.31。可以看出，随着输入大小增加，MPI_OpenMP 算法和

AVX_OpenMP_MPI 算法相对于 Origin 算法的加速比增加更快，表现出更好的可扩展性。

总的来说，根据给定的数据，MPI_OpenMP 算法和 AVX_OpenMP_MPI 算法在大部分情况下都表现出较高的加速比，尤其是在处理较大规模的数据时。这表明多线程和向量化优化对于提高算法性能具有显著的影响。而 MPI 算法在小规模数据上表现较好，但随着数据规模增加，其性能优势逐渐减弱，可能受到通信开销和负载均衡等因素的影响。

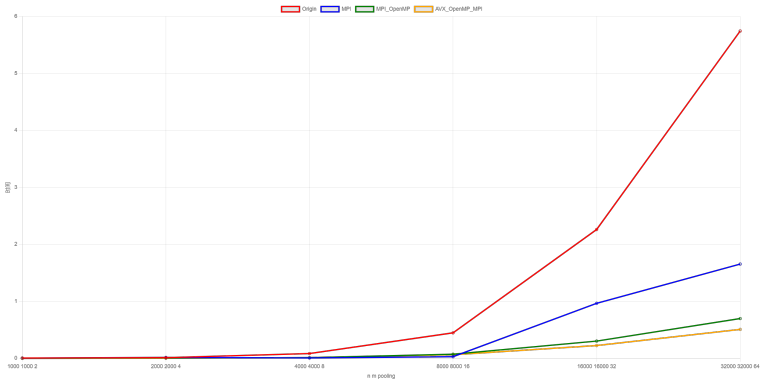

图 4.10: 不同算法的程序运行时间

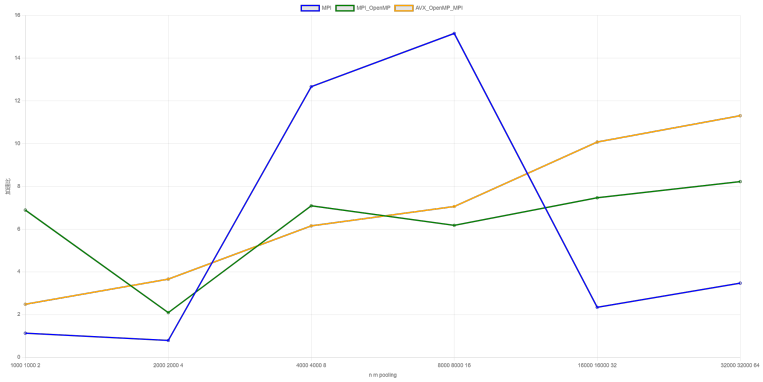

图 4.11: 不同算法加速比

4.5 嵌入操作结果分析

接下来我们考虑嵌入操作，与上面相同，在这部分中，我们并不进行单独的优化，而是多种优化策略并行处理，例如，我们会在 MPI 的基础上逐渐增加优化策略，探究使用多种优化策略对程序运行时间的影响。我们测得了不同数据规模下程序的运行时间，如下表6所示，可视化的图如4.12和图4.13所示：

我们考虑对结果分析：对于输入大小为 600000 和 1800 维的情况，MPI 算法的运行时间为 179.6515，加速比为 729.235 / 179.6515 = 4.06。MPI_OpenMP 算法的运行时间为 140.89841，加速比为 729.235 / 140.89841 = 5.17。AVX_OpenMP_MPI 算法的运行时间为 135.8489，加速比为 729.235 / 135.8489 = 5.36。可以看出，MPI_OpenMP 和 AVX_OpenMP_MPI 相比于 MPI 算法都有更好的性能，其中 AVX_OpenMP_MPI 的性能最佳。也就是说，多种并行化优化策略是可以在其基础上进一步提高性能的。

但是我们仍然发现：对于多种优化策略并行，虽然仍然可以减少程序运行时间，但是与该优化策略单独进行相比，优化效果要弱很多，我们推测这可能是由于我们的处理器的性能有限，在单一并行

下已经发挥了比较好的并行性能，多种优化策略并行很可能造成通信等开销进一步增大，影响程序的运行效果。

随着输入大小和嵌入维度的减小，各个算法的运行时间和加速比也相应减小。这表明随着问题规模的减小，算法的并行化和优化效果可能会有所降低。我们推测这是由于在数据规模减少的情况下，进程之间通信的开销变得格外突出，所以加速比和优化效果可能相应降低。

表 6: Embedding 算法运行时间比较

参数设置		Origin	MPI	MPI_OpenMP	AVX_OpenMP_MPI
input_size	embedding_dim	时间	时间	时间	时间
600000	1800	729.235	179.6515	140.89841	135.8489
500000	1500	179.798	64.2789	43.2365	45.14654
400000	1200	45.656	19.7265	13.1215	12.79865
300000	900	13.689	5.05952	3.5478	3.72877
200000	600	5.789	1.61169	1.4988	1.49291
100000	300	0.999	0.317183	0.35987	0.329065
50000	150	0.16584	0.0734297	0.078964	0.0693496
25000	75	0.019545	0.020658	0.0149876	0.0139778

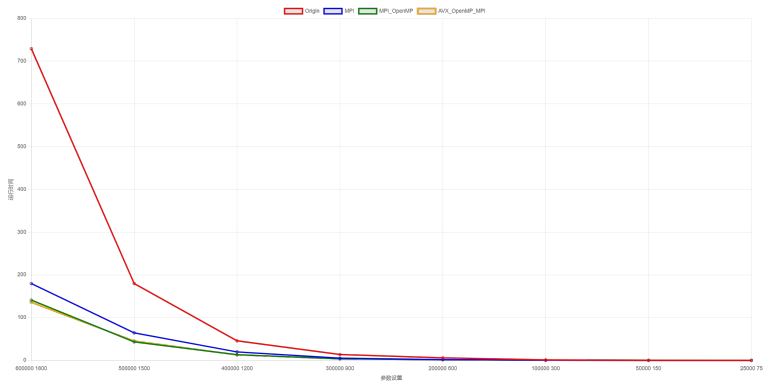

图 4.12: 不同算法的程序运行时间

4.6 PCA 主成分分析结果分析

与前面一样，我们在这里同样针对多种优化策略并行叠加测试性能，我们额外添加了 GPU 优化的选项供参考，下面是我们整理的数据，如7和图4.14和图4.15所示：

表 7: PCA 算法测试时间

n	m	k	Origin	MPI	MPI_OpenMP	AVX_MPI_OpenMP	GPU
2000	200	10	0.725664	0.0595061	0.0649667	0.0736112	1.98456
4000	400	10	1.54789	0.327868	0.366709	0.414305	3.25649
6000	600	10	5.26589	0.957157	0.863804	0.84937	11.6598
8000	800	10	12.59848	3.56623	2.021	1.98605	29.5894
10000	1000	10	46.7366	5.24536	3.57661	3.66901	98.3595
15000	1500	10	203.324	38.24	16.092	15.053	456.1259
20000	2000	10	806.897	120.34	60.5729	69.4669	1608.959
30000	3000	10	1980.26	743.685	147.069	148.581	3789.598

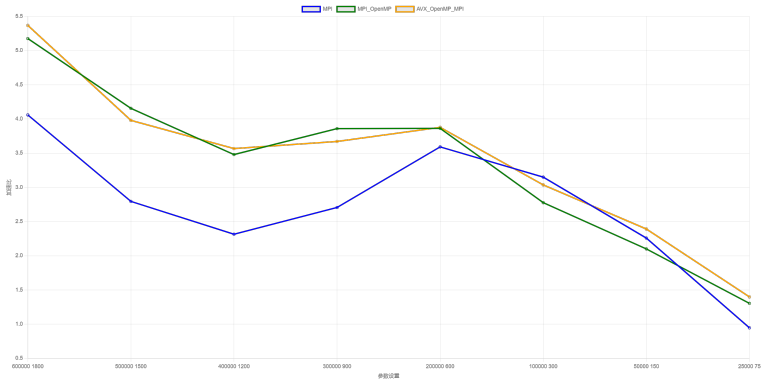

图 4.13: 不同算法加速比

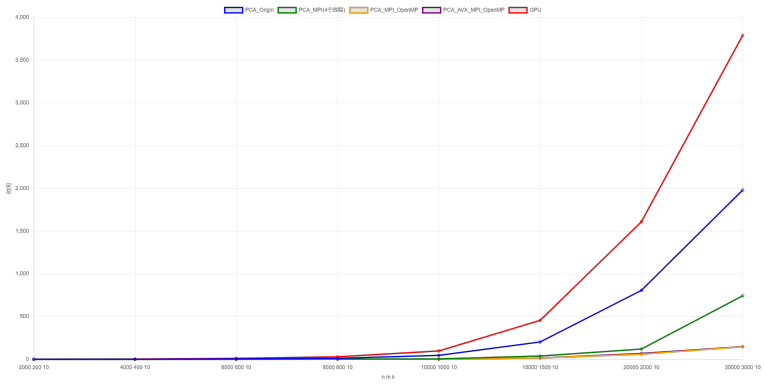

图 4.14: 不同算法的程序运行时间

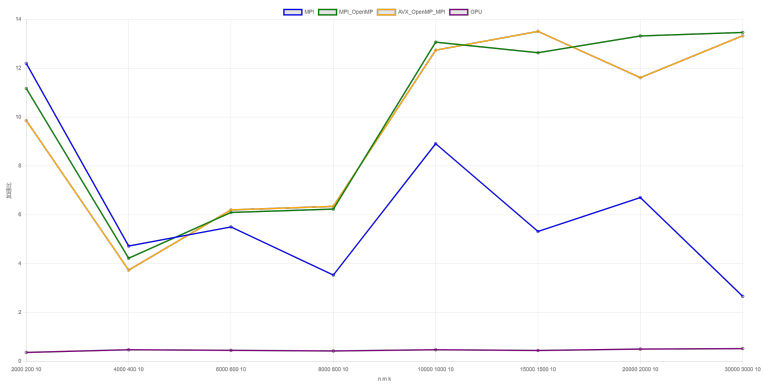

图 4.15: 不同算法加速比

根据我们所测得的 PCA 算法测试时间数据，可以观察到以下情况：

- 对于相同的输入规模（n，m，k），MPI 算法相对于 Origin 算法具有更短的运行时间，表明 MPI 算法在并行计算中有效地利用了多个处理器或计算节点。
- 在 MPI 算法的基础上，结合 OpenMP 并行化技术，MPI_OpenMP 算法在大部分情况下进一步减少了运行时间，表明 MPI 和 OpenMP 的组合对于提高计算效率起到了积极作用。
- 在 AVX_OpenMP_MPI 算法中，除了使用 MPI 和 OpenMP 并行化之外，还使用了 AVX 指令集优化。可以看到，AVX_OpenMP_MPI 算法在大部分情况下相对于 MPI_OpenMP 算法进一步减少了运行时间，这是由于 AVX 指令集的向量化计算能够更高效地处理大规模数据。
- GPU 算法在所有情况下都显示出最高的运行时间,这与我们上面讨论的结论是一致的。虽然 GPU 的并行效果极好，特别适合做计算密集型的任务，但是由于 PCA 计算密集程度不高，并且需要大量的通信，就导致了在通信时间上花费的时间过于长了。此外，由于 SYCL 的 GPU 编程架构只能使用集成显卡，即 Intel 中与 CPU 一起的集成显卡，无法使用英伟达的独立显卡，这也就导致了一种“不公平”的现象出现。CPU 的运算性能远远好于 GPU，无论是主频，位宽，缓存等，都要优于集成显卡，这也是导致 GPU 运行速度慢于 CPU 的原因之一。

总体而言，MPI 和 OpenMP 的并行化技术以及 AVX 指令集优化对于加速 PCA 算法具有显著的效果。而 GPU 作为一种强大的并行计算设备，能够进一步提升计算性能，但是由于集成显卡和任务的不适配性，就导致了在我们的实验中出现了反常，这也告诉我们：我们需要根据任务的不同选择合适的并行化策略和优化技术，这样才可以显著改善算法的运行时间。

5 Profiling 分析

Profiling 是一种软件性能分析方法，用于确定程序的性能瓶颈和优化机会。通过对程序运行过程进行详细的性能测量和分析，可以帮助我们了解程序的运行情况、资源利用情况和性能瓶颈，并针对性地进行性能优化。通常而言，Profiling 通常包括以下几个步骤：

- 收集性能数据：使用专门的性能分析工具，如性能分析器或代码调试器，对程序进行运行时性能数据的收集。这些工具可以记录函数调用、内存使用、CPU 利用率、I/O 操作等关键性能指标。
- 分析性能数据：通过对收集的性能数据进行分析和统计，可以获得关于程序性能的详细信息。例如，可以确定哪些函数或代码段消耗了最多的时间、哪些内存操作较为频繁、是否存在线程竞争等。
- 发现性能瓶颈：通过分析性能数据，可以确定程序中的性能瓶颈所在。性能瓶颈可能包括计算密集型任务、内存访问延迟、磁盘或网络 I/O 等。
- 优化性能：根据性能分析的结果，针对性地进行性能优化。可以通过改进算法、优化关键代码段、减少内存访问次数、使用并行化技术等方法来提高程序的性能。

Profiling 分析对于软件开发和优化非常重要。它可以帮助我们了解程序的性能特征，找出性能瓶颈并采取相应的优化措施。通过准确的性能分析和优化，可以改善软件的响应时间、吞吐量、资源利用率等关键性能指标，提升用户体验和系统的整体效率。在这里，我们采用 Windows 系统下的 Intel VTune 进行性能分析，我们主要关注以下指标：

- CPI 是 CPU 周期数和指令数的比率，反映了 CPU 执行指令的效率。CPI 越低，表示 CPU 在执行每条指令时所需的周期数较少，执行效率较高。高 CPI 可能意味着指令级别的并行度较低、缓存命中率较低或者流水线发生了中断等。
- 指令数反映了 CPU 执行算法所需的指令数量，是算法复杂度的一个重要指标。指令数越多，表示算法执行过程中需要执行的指令越多，对 CPU 的负载也越高。
- LLC 是指 CPU 中最后一级缓存，它是 CPU 和主存之间的缓存。LLC 的大小和命中率直接影响程序的缓存性能。高命中率意味着程序能够充分利用缓存来提高数据访问效率，而低命中率可能导致频繁的缓存失效，增加了访存延迟。我们在这里，所展示的是 LLC 的缺失率。
- Elapsed Time 是算法执行的总时间，包括 CPU 执行时间、I/O 操作时间和其他等待时间。它是衡量算法整体执行效率的指标，表示从开始执行到完成执行所经过的时间。
- CPU 时间是指 CPU 执行算法的总时间，不包括其他操作的时间开销。它是算法性能的一个重要指标，用于衡量 CPU 在执行算法上的效率。
- 逻辑核心利用率反映了 CPU 在执行算法时的利用率。它表示逻辑核心实际执行任务的时间与总时间的比率。较高的逻辑核心利用率表示 CPU 在执行算法时能够充分利用多个逻辑核心的计算能力，提高并行计算效率。

在这里，我们仅对上述六个算法中的两个算法进行 Profiling 分析，通过 Profilling 的结果来说明上述实验结果出现的原因。选取矩阵乘法和卷积操作进行分析，是因为矩阵乘法是计算密集型任务，可以较好的体现并行策略的优化效果；此外，卷积操作是添加了 GPU 进行优化的，并且计算相对较密集，也具有一定的代表性。

5.1 对矩阵乘法进行 Profiling 分析

我们使用 VTune 进行性能分析，使用 $n = m = p = 500$ 的数据规模进行测试，结果如下表8所示：

表 8: 性能指标比较

算法	CPI	指令数	LLC	Elapsed Time	CPU 时间	逻辑核心利用率
朴素矩阵乘法算法	0.767	50429800000	550231	8.651	8.525	6.20%
pthread 优化	0.768	50406800000	0	2.287	2.198	23.70%
AVX 优化	0.778	3300500000	0	2.2	2.056	1.70%
MPI 优化	0.723	71298600000	0	2.59	2.489	25.80%

接下来我们根据该 Profiling 结果进行分析：

- CPI (CPU 周期数和指令数的比率)：朴素矩阵乘法算法具有较低的 CPI 值，说明 CPU 在执行指令时的效率较高，每个指令所需的周期数相对较少。其他优化算法（pthread 优化、AVX 优化、MPI 优化）的 CPI 值稍高，可能是由于额外的优化指令或处理开销引起的。
- 指令数：朴素矩阵乘法算法的指令数较高，说明它执行算法所需的指令数量较多，具有较高的算法复杂度。除去 MPI 外，其他优化算法在指令数方面有所改进，可能通过优化算法逻辑、并行化或硬件指令集的使用来减少指令数量；我们推测是由于 MPI 需要大量进程间通信，就导致了计算机需要执行的指令数变多，而由于其逻辑核心利用率更高一些，所以运行时间也会变得更短。

- LLC (最后一级缓存)：在测定的数据中，除去朴素算法外所有算法的 LLC 值均为 0，表示在执行过程中没有发生最后一级缓存的访问。对于朴素矩阵乘法算法，LLC 值为非零可能是因为在执行过程中发生了大量的内存访问操作。朴素矩阵乘法算法通常涉及多重嵌套循环，其中每个循环迭代都需要对矩阵元素进行读取和写入操作。这会导致大量的内存访问，而 LLC 作为 CPU 和主存之间的最后一级缓存，它的作用是存储经常使用的数据以减少对主存的访问。相比之下，优化算法（如 pthread 优化、AVX 优化、MPI 优化）可能采用了更高效的内存访问方式或数据分布策略，减少了对主存的直接访问需求。这可能涉及到数据的局部性、缓存友好性和数据重用等优化技术，以提高算法的性能和效率。因此，在优化算法中，LLC 值为 0 可能表示它们能够更好地利用 CPU 的缓存层次结构，减少了对 LLC 的访问需求，并通过其他方式优化了内存访问模式。
- Elapsed Time (算法执行的总时间)：优化算法（pthread 优化、AVX 优化、MPI 优化）的 Elapsed Time 较朴素矩阵乘法算法更短，说明这些优化算法在执行过程中能够更快地完成计算任务。
- CPU 时间：CPU 时间反映了 CPU 执行算法的总时间，也是算法性能的一个重要指标。优化算法（pthread 优化、AVX 优化、MPI 优化）的 CPU 时间较朴素矩阵乘法算法更短，说明这些优化算法能够更有效地利用 CPU 资源来执行计算任务。
- 逻辑核心利用率：逻辑核心利用率反映了 CPU 在执行算法时的利用率，即逻辑核心的利用程度。根据提供的数据，优化算法（pthread 优化、MPI 优化）的逻辑核心利用率较高，说明它们能够更充分地利用 CPU 的逻辑核心来并行执行计算任务。

根据我们上面的讨论，对于矩阵乘法这种计算密集型任务而言，并行策略能够有效的提高算法的效率。AVX 这种 SIMD 指令集的效果较好，是通过减少指令数提高 CPI 的角度提高程序运行的效率；MPI 和 Pthread 是通过提高逻辑核心的利用率来减少程序运行时间的。

5.2 对卷积操作进行 Profiling 分析

接下来我们对卷积操作进行 Profiling 分析，我们选取 (600000,48000) 这组数据进行 Profiling 分析，这样程序运行时间不会太长，同时数据规模也相对较大，可以有效抵消流水线启动等的开销，具体来说，性能体现如下表9所示：

表 9: 卷积任务的性能指标

算法	CPI	指令数	LLC	Elapsed Time	CPU 时间	逻辑核心利用率
朴素卷积算法	0.713	168534800000	0%	28.735	26.336	5.70%
MPI 优化	0.707	2085698000000	0%	7.12824	6.98756	15.40%
pthread 优化	0.693	168424400000	0%	4.26987	4.22699	15.40%
AVX 优化	0.682	29890800000	0%	4.24801	4.13266	7.80%
GPU 优化	0.896	258534800000	256989	57.85461	53.56894	2.30%

接下来我们根据该 Profiling 结果进行分析：

- 从 CPI 的角度：我们可以观察到，除去 GPU 优化外，其余的优化算法 CPI 都相对较低，说明在单位的时钟周期内能够执行更多的指令。所以其执行时间比朴素算法快一些；GPU 由于 CPI 低于朴素算法，所以其执行速度较为缓慢。

- 从指令数的角度来看：较少的指令数表示算法执行所需的指令数量较少，是算法复杂度的一个重要指标。在优化算法中，指令数较朴素算法有所减少，表明算法经过优化后减少了不必要的指令操作；而 GPU 指令数要多于所有的算法，且其 CPI 低，所以 GPU 速度慢也是应有之义。
- 从 LLC 的角度来看，只有 GPU 的 LLC 不为 0，其余算法皆为 0，我们将在后面进一步考虑原因，在这里先不讨论。
- 从逻辑核心利用率的角度考虑：该参数代表程序使用可用逻辑核心的比例。这个指标越高，意味着程序更有效地使用了多核处理器的并行处理能力。我们可以看到，GPU 的逻辑核心利用率极低，这似乎与我们的理论并不相符，而较低的逻辑核心利用率也是导致其性能较差的原因，我们将在后面的实验总结部分详细讨论 GPU 性能较差的原因。

6 总结与反思

6.1 课程收获

实验报告写到这里，标志着这门课暂且告一段落。通过本次课程，我们学到了一些经典的并行化的技巧与手段，以及如何通过并行化策略提高程序运行的效率，与计算机组成原理相结合，可以从底层了解到 SIMD 等并行化手段能够加速程序运行时间的底层原因；此外，最重要的收获是能够熟练的掌握 LaTeX 撰写实验报告，符合科技论文的格式规范，每次实验报告都是心血的结晶，当看到自己写完的实验报告时，内心只有欣喜。

在这门课程中，感谢王刚老师的指导，感谢各位学长学姐的指导，没有老师和学长学姐的答疑解惑，可能实验过程还要艰难百倍，包括本地环境配置，鲲鹏服务器的配置，都离不开学长学姐们的付出！在这里感谢老师和学长学姐的付出！

6.2 实验总结

6.2.1 GPU 处理速度缓慢的原因

首先我们先来解决前面隐藏的问题：GPU 为什么相较于 CPU 运行时间较长，并且对应的性能指标很差：我们使用的实验环境是 Intel Core12th 的 CPU 和 Intel(R) lris(R) Xe Graphics 的集成显卡，我们先对该 CPU 和 GPU 作相应的介绍：

首先我们给出 CPU 的参数：

- CPU 名称：CPU - Intel® Core™ i9-13900K
- 总核心数：14 核心，20 线程
- 高性能核心：6 核心，12 线程，基础频率 2.3 GHz，最高加速频率 4.7 GHz
- 高效能核心：8 核心，8 线程，基础频率 1.7 GHz，最高加速频率 3.5 GHz
- 缓存大小：L1 缓存 1248 KB，L2 缓存 11.5 MB，L3 缓存 24 MB
- 内存支持：最大内存容量为 64 GB（支持 DDR5 4800 MT/s、DDR4 3200 MT/s）
- 工业制程：7nm
- 提升功耗模式：115 W

接下来是 GPU 的参数：

- GPU 名称：GPU - Intel Iris Xe:
- 流处理单元：640 个
- 核心频率：300 MHz(Turbo 频率：1100 MHz)
- 工业制程：10nm
- 内存类型：系统共享内存
- 内存位宽：系统共享内存
- 最大显存：系统共享内存
- 二级缓存：1024 KB
- 三级缓存：3.75 MB
- TDP 功耗：28W

我们从参数对比中不难发现：CPU 的主频要远远低于 CPU 的主频，虽然流处理单元有 640 个，每一个单元都可以进行并行运算，但是由于其频率过于低下，就导致了时钟周期相较于 CPU 长了很多，这是导致 CPI 弱于 CPU 运行的第一个原因。

此外我们考虑缓存大小：对于 CPU 而言，cache 的缓存大小特别大，接近 36MB，这意味着我们可以存储更多的数据在 cache 中，通过 LRU 算法等来访问和动态维护一个极大的 cache 表，可以有效的提高访问效率，对于一个数据而言，如果在 cache 命中的情况下，可能仅需要几个时钟周期就可以实现读写操作；而 GPU 的 cache 一共才仅有 4MB 左右，且读取速率要远远弱于 CPU 的 cache，这就导致了 GPU 优化的算法 LLC 必然会出现缺失的现象。

第三个原因是内存的大小和访问速度：我们考虑 cache 未命中的情况：缺失数据需要在内存中寻找，内存同样没有 DDR 等方式进行优化，读取的速率仍然要远远弱于 CPU，这就是为什么 GPU 的速度要慢于 CPU 那么多的第三个原因。

此外，从功耗角度分析，CPU 的功耗最大为 110W，而该 GPU 仅仅有 28W，这就导致了其性能必然要弱于 CPU 很多，所以我们使用 SYCL 优化的卷积算法的加速比一直很低，甚至超过朴素算法的两倍也是情有可原的。

而如果我们使用集成显卡，由于集成显卡有独立的显存，并且各级的 cache 也要远远大于集成显卡，读取速度也相对较快，在这些指标与 CPU 持平的情况下，有更多的流处理单元就可以更好的利用 GPU 的并行性能了！事实上，目前在深度学习和机器学习中，我们训练神经网络中必然需要使用 GPU 进行加速训练，就个人经验来讲，使用独立显卡 (GTX3060 130W) 的训练速度就已经超过了 CPU，并且能够有效的提高近 20 倍的训练速度！这便是因为我们上述讨论的参数与 CPU 几乎一致而其拥有更多的处理单元供并行运算。我们给出使用 GPU 和 CPU 前后的训练速度对比：我们训练同一个深度学习模型，程序运行时间如图6.16和图6.17所示：

6.2.2 整体总结

解决了遗留问题后,我们来对本次大作业进行总结:本次大作业我们通过使用 MPI,SIMD,Pthread,GPU 等并行优化策略对目前用处比较广泛的深度学习方法进行了优化，例如卷积，嵌入，反向传播等。

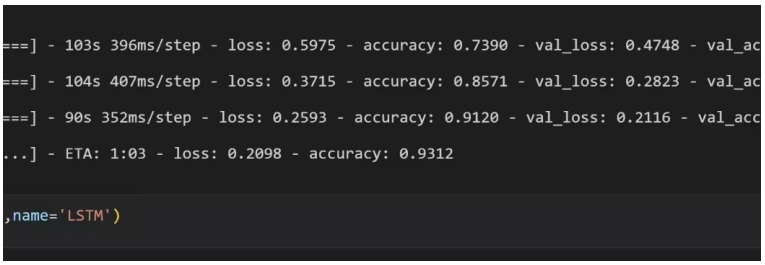

图 6.16: 使用 CPU 训练

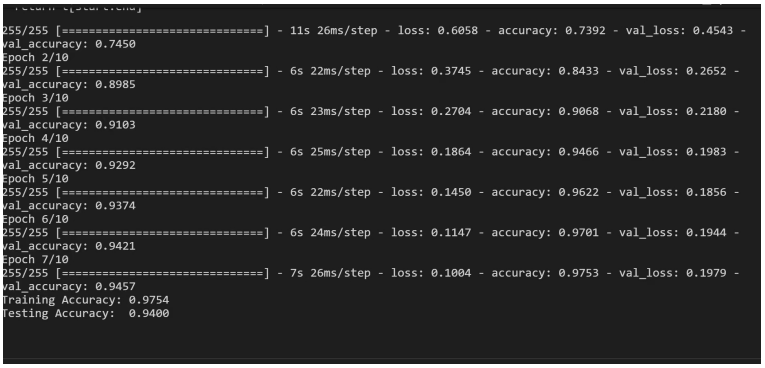

图 6.17: 使用 GPU 训练

其中，我们的全新工作量大约在 50% 以上，我们对于上述六钟算法，仅仅在前面实现了几种优化策略，其余的优化策略都是在本次实验中体现的。

在结果分析部分，我们给出了不同优化策略对于程序运行时间的影响，同时在 Profiling 部分以卷积和矩阵乘法为例，分析了 GPU 优化效率较差和不同优化算法为什么可以优化程序运行时间的原因。

在本小节中，我们通过对比本次实验的硬件指标，对 Profiling 指标出现的异常进行了又一次解答和分析，并且给出了自己对于 GPU 加速算法的理解，总体来看，本次实验工作量较大，需要编写 20 余种代码，并且查阅很多文献资料，对于所有的代码和原始数据，绘图代码和原始图表，都可以在并行程序设计仓库中给出，具体项目链接将在实验分工中给出。

6.3 实验分工

在本次期末大作业中，张铭徐负责 PCA，嵌入层，池化的所有部分的总结与代码编写任务，张惠程负责矩阵乘法，卷积，反向传播部分的代码编写和数据整理工作，最后的实验结果由两人共同讨论得出，本次实验以及本学期是所有的实验数据，原始代码，实验报告等都可以在并行程序设计仓库中找到。