南開大學
Nankai University

计算机学院

并行程序设计实验报告

OpenMP Pthread 实验报告

姓名：张铭徐 张惠程

学号：2113615 2112241

专业：计算机科学与技术

2023 年 4 月 25 日

目录

1 实验目的及背景介绍

1.1 实验目的

多线程编程可以在深度学习的卷积神经网络中发挥重要作用，可以提高训练速度和效率，从而加快模型的训练和优化，提高在训练过程中的收敛速度。在卷积神经网络中，大多数计算任务都是矩阵运算，这些运算通常可以被有效地并行化。例如，在卷积层中，每个卷积核都可以被视为一个独立的计算任务，可以使用多个线程并行地计算每个卷积核的输出。类似地，在池化层中，每个池化操作也可以被看作是独立的计算任务，可以并行计算。此外，在深度学习中，通常需要处理大量的数据，因此使用多线程可以提高数据读取和预处理的速度，进而加快模型训练的速度。

OpenMP 和 Pthread 都是常用的并行编程库，可以在卷积神经网络中应用以加速模型的训练过程。OpenMP 是一种基于共享内存的并行编程模型，可以在 C、C++ 和 Fortran 等编程语言中使用。在卷积神经网络中，OpenMP 可以用于并行化循环操作，例如对于每个卷积核的循环计算。通过使用 `#pragma omp parallel for` 指令，可以将 for 循环并行化，将计算任务分配到多个线程中执行。

Pthread 是一种 POSIX 标准的线程库，支持 C 和 C++ 等编程语言，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，可以令编程人员自由的控制 CPU 的多个核心，自由设计并行计算。在卷积神经网络中，Pthread 可以用于实现线程级别的并行计算，例如在卷积层中对于每个卷积核的矩阵乘法操作，可以使用pthread_create()函数创建多个线程来执行卷积核的计算任务，并使用pthread_join()函数等待线程完成任务。在使用 OpenMP 和 Pthread 时，需要注意线程的调度和同步问题，例如避免线程间的竞争条件和死锁。此外，还需要考虑性能的平衡问题，例如多线程计算的负载均衡等。

在本次实验中，我们将聚焦矩阵乘法，卷积，反向传播这三个卷积神经网络中极为常见的操作进行 pthread 多线程策略优化和 OpenMP 多线程策略优化，并测试出程序运行时间；针对优化代码进行 profiling 分析，给出程序加速的原因。最后，我们将会比 OpenMP 和 Pthread 两种方法的优缺点。

1.2 实验背景知识介绍

1.2.1 矩阵乘法背景知识简介

在深度学习中，矩阵乘法，卷积运算，以及反向传播 (BP 算法) 是几个非常常见的操作，广泛存在在各种框架中，特别是反向传播，在每层神经网络之间，都需要进行反向传播以更新参数。下面分别介绍它们的基本概念和使用多线程优化的必要性。

矩阵乘法是深度学习中非常常见的操作之一。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，并且矩阵乘法可以多线程进行运算，因此使用多线程编程的技巧是很有必要的。

1.2.2 卷积知识简介

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。

1.2.3 反向传播知识简介

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。在实现反向传播算法时，可以使用 Pthread 或 OpenMP 等多线程技术来优化计算效率。具体来说，可以将前向传播和反向传播分配给多个线程并行计算，以加快计算速度。同时，为了避免数据竞争等问题，需要使用适当的同步机制来保证计算的正确性。

在 Pthread 中，可以使用 `pthread_create` 函数创建多个线程，并将任务分配给不同的线程处理。在每个线程的内部，可以使用 SIMD 指令来实现向量化计算，以进一步提高计算效率。同时，为了保证计算的正确性，可以使用 `pthread_mutex_lock` 和 `pthread_mutex_unlock` 等同步机制来避免数据竞争问题。

类似地，在 OpenMP 中，可以使用 #pragma omp parallel 指令来将任务分配给多个线程并行计算。在每个线程的内部，可以使用 #pragma omp simd 指令来实现向量化计算，以进一步提高计算效率。同时，可以使用 #pragma omp critical 等同步机制来避免数据竞争问题。

因此，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。

1.2.4 Pthread 知识简介

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。

1.2.5 OpenMP 知识简介

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。

1.3 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。

2 对矩阵乘法进行 Pthread 优化

2.1 朴素算法设计思想

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6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

2.2 矩阵乘法 Pthread 优化策略设计思想

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 2 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 $a, b, c, m, n, p, star$**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。

2.3 代码

根据上面的分析，我们在下面给出平凡算法和 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。

矩阵乘法平凡算法

```
std::vector<std::vector<double>> matrix_multiply(const
    std::vector<std::vector<double>>& A, const std::vector<std::vector<double>>& B)
    {
    int m = A.size();
    int n = A[0].size();
    int p = B[0].size();
    std::vector<std::vector<double>> C(m, std::vector<double>(p, 0.0));
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C;
}
```

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 对卷积操作进行 Pthread 优化

3.1 朴素算法设计思想

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 3 所示：

Algorithm 3 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 卷积操作 Pthread 优化策略设计思想

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。
- 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 4 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

3.3 代码

根据上述代码描述，我们在下面给出平凡卷积算法和 Pthread 优化卷积算法所对应的 C++ 代码，同样的，我们也仅仅给出对应的计算函数，全部代码将在 GitHub 的项目链接中给出。值得注意的是，我们在这里所采用的是朴素版卷积操作，同样的，我们可以实现 SIMD 优化的卷积操作，那么如果使用 SIMD 优化卷积操作，耗时会更短。SIMD 优化卷积操作的代码和设计思想在这里我们由于篇幅问题不再给出，可以参照上次实验的代码和实验思路进行考察。

卷积操作平凡代码

```
std::vector<float> convolution_avx2(const std::vector<float>& signal, const
    std::vector<float>& kernel) {
    int signal_length = signal.size();
    int kernel_length = kernel.size();
    int result_length = signal_length + kernel_length - 1;
    std::vector<float> result(result_length, 0);

    for (int i = 0; i < signal_length; ++i) {
        for (int j = 0; j < kernel_length; ++j) {
            float kernel_elem = kernel[j];
            float signal_elem = signal[i];
            int result_idx = i + j;
            result[result_idx] += kernel_elem * signal_elem;
        }
    }
    return result;
}
```

对于算法优化部分，我们采用以下代码进行多线程操作，我们在主函数中采用多线程操作，至于卷积操作部分，则可以直接调用写好的函数进行。值得注意的是，我们在后面计时部分会同时给出 pthread+ 普通卷积核 pthread+SIMD 优化卷积操作。

卷积操作 Pthread 优化代码

```
    vector<pthread_t> threads(num_threads);
    vector<ConvolutionData> thread_data(num_threads);
    int chunk_size = signal_length / num_threads;
    auto start = chrono::high_resolution_clock::now();
    // 创建并运行线程
    for (int i = 0; i < num_threads; ++i) {
        thread_data[i].signal = &signal;
        thread_data[i].kernel = &kernel;
        thread_data[i].result = &result;
        thread_data[i].start = i * chunk_size;
        thread_data[i].end = (i == num_threads - 1) ? signal_length : (i + 1) *
            chunk_size;
        pthread_create(&threads[i], nullptr, convolution_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);
        tim[i] += thread_data[i].elapsed_time;
    }
    for (int i = 0; i < num_threads; i++) {
        std::cout << "线程 " << i << "用时" << tim[i] << endl;
    }
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed = end - start;
    cout << "计算耗时: " << elapsed.count() << " 秒" << endl;
```

代码中，使用了 pthread 库来实现多线程优化。具体来说，优化的部分包括：

- 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。
- 等待所有线程完成计算。

4 对 BP 算法 (反向传播) 进行 Pthread 优化

4.1 朴素算法设计思想

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 5 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

4.2 反向传播 Pthread 优化策略设计思路

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 6 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

4.3 代码

鉴于神经网络部分代码特别长，在这里暂且不放入训练和反向传播的代码，仅放入创建线程的相关代码，完整代码与测试数据请参照后面附的 GitHub 项目链接。

BP 算法 Pthread 优化代码

```
vector<pthread_t> threads(num_threads);
 vector<BPData> thread_data(num_threads);
 int chunk_size = num_samples / num_threads;
 auto start = std::chrono::high_resolution_clock::now();
 for (int epoch = 0; epoch < epochs; ++epoch) {
     for (int i = 0; i < num_threads; ++i) {
         thread_data[i].inputs = &inputs;
         thread_data[i].targets = &targets;
         thread_data[i].output_weights = &output_weights;
         thread_data[i].hidden_weights = &hidden_weights;
         thread_data[i].output_bias = &output_bias;
         thread_data[i].hidden_bias = &hidden_bias;
         thread_data[i].start = i * chunk_size;
         thread_data[i].end = (i == num_threads - 1) ? num_samples : (i + 1) *
             chunk_size;
         pthread_create(&threads[i], nullptr, bp_thread, &thread_data[i]);
     }
     for (int i = 0; i < num_threads; ++i) {
         pthread_join(threads[i], nullptr);
         tim[i] += thread_data[i].elapsed_time;
     }
 }
 for (int i = 0; i < num_threads; i++) {
     std::cout<<"线程 "<<i<<"用时" << tim[i] << endl;
```

```
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "训练耗时: " << elapsed.count() << " 秒" << std::endl;
    return 0;
```

5 实验结果分析

5.1 矩阵乘法结果分析

我们通过 C++ 库中的 chrono 库计算程序的耗时，我们在下面给出了不同数据规模下，不同算法的运行时间对比的数据表1和可视化的图5.1，如下所示：

表 1: 矩阵乘法不同算法运行时间对比

$n = m = p$	origin	AVX2 优化	num_pthread=4	AVX2&pthread
800	35.72	1.77	9.42	0.5
700	23.67	1.25	6.38	0.37
600	14.64	0.74	3.99	0.22
500	8.64	0.43	2.43	0.12
400	4.43	0.21	1.29	0.067
300	1.88	0.09	0.52	0.03
200	0.56	0.026	0.16	0.01
100	0.07	0.004	0.024	0.004

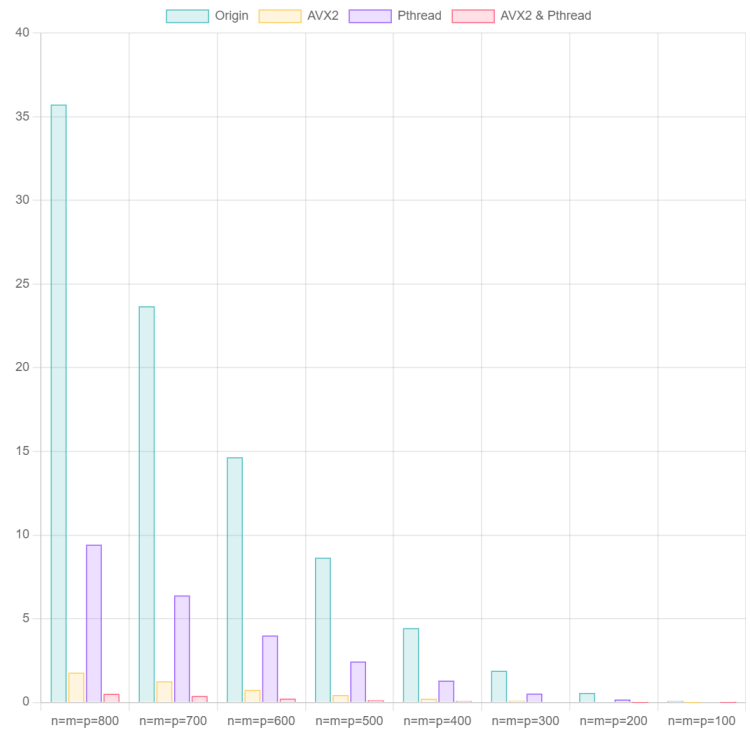

图 5.1: 矩阵乘法不同算法运行时间对比

我们还计算出了不同数据规模和不同的算法 (平凡算法和 SIMD 优化算法) 下，在线程数相同的情况下，他们的加速比的变化趋势，数据如下表2所示:

表 2: 矩阵乘法不同算法加速比

$n = m = p$	加速比 (AVX)	加速比 (origin)
800	3.54	3.791932059
700	3.378378378	3.710031348
600	3.363636364	3.669172932
500	3.583333333	3.555555556
400	3.134328358	3.434108527
300	3	3.615384615
200	2.6	3.5
100	1	2.916666667

我们考虑上述数据：当只使用 AVX2 优化时，我们给出的代码在各种数据规模下的加速比都很稳定。在 n=m=p=100 时，加速比稍低，但这可能是由于较小的数据规模使得并行计算效果不明显。相对的，在其他情况下，AVX2 优化提供了约 3 倍的加速。具体而言，我们的结论如下所示：

- 当使用 4 个 pthread 线程进行优化时，加速比在不同数据规模下略有波动。总体来看，pthread 优化的加速比在大多数情况下约为 3 到 4 倍。并且随着数据规模的增长，我们的加速比也在不断提高。
- 当同时使用 AVX2 和 pthread 优化时，加速比相对于原始代码 (origin) 有所提高，但相对于只使用 AVX2 优化的情况并没有显著提高。这可能是由于多线程和向量化之间的资源竞争导致的。在这种情况下，多线程和向量化可能无法充分利用硬件资源，从而限制了性能提升。
- 结论：从我们测得的数据来看，单独使用 AVX2 或 pthread 优化都能为矩阵乘法带来较好的加速效果。然而，同时使用 AVX2 和 pthread 优化时，性能提升并不显著。这可能是由于向量化和多线程之间的资源竞争或内存访问模式等原因导致的。为了获得最佳性能，我们下一步可以尝试调整代码和参数，以找到适合实验平台硬件的最佳性能平衡。

5.2 卷积操作结果分析

与矩阵乘法类似的，我们采取相同的方式测定在不同数据规模下不同算法对于卷积操作的加速程度，如下表和下图所示：

表 3: 卷积运行时间对比（运行时间，单位：秒）

input_dim	kernel_dim	origin	AVX2 优化	num_pthread=4	AVX2&pthread
300000	24000	236.82	40.78	62.50	11.00
200000	16000	105.2	18.23	27.61	4.91
100000	8000	26.41	4.69	7.08	1.33
50000	4000	6.64	1.12	1.86	0.347
40000	3200	4.30	0.794	1.22	0.231
30000	2400	2.45	0.458	0.65	0.136
20000	1600	1.11	0.193	0.2933	0.054
10000	800	0.279	0.051	0.084	0.015

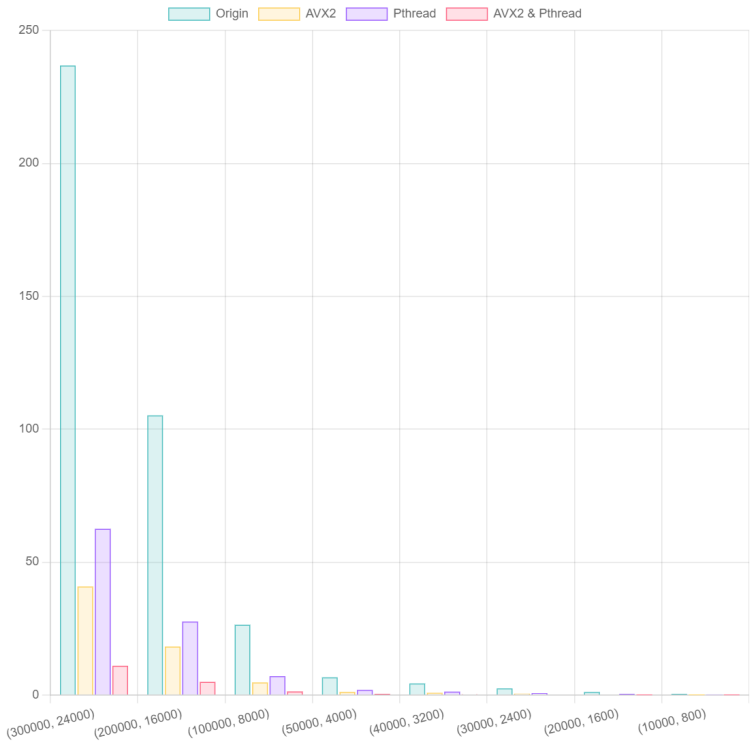

图 5.2: 卷积操作不同算法运行时间对比

同时，我们也针对不同算法和不同数据规模的加速比做了相应的测试与计算，结果如下表所示：

表 4: 卷积加速比对比

input_dim	kernel_dim	加速比 _AVX	加速比 _origin
300000	24000	3.707272727	3.78912
200000	16000	3.712830957	3.810213691
100000	8000	3.526315789	3.730225989
50000	4000	3.227665706	3.569892473
40000	3200	3.437229437	3.524590164
30000	2400	3.367647059	3.769230769
20000	1600	3.574074074	3.784520968
10000	800	3.4	3.321428571

从我们测得的数据中，我们可以得出以下结论：

- 在所有情况下，使用 AVX2 优化和 pthread (4 个线程) 优化相结合的情况下，计算时间都得到了最大的减少，即性能最佳。
- 使用 AVX2 优化相对于原始算法，在大部分情况下的加速比在 3.3 到 3.7 之间，相对稳定。
- 使用 4 个 pthread 线程相对于原始算法，在不同的数据规模下，加速比有所波动，但整体上性能优于单纯使用 AVX2 优化。
- 数据规模越大，不同优化方法之间的性能差异越明显。在较小的数据规模下，各优化方法之间的差异相对较小。

通过对这些数据的分析，我们可以得出结论，将 AVX2 优化与 pthread 结合使用，可以在不同数据规模下获得较好的性能提升。不过，值得注意的是，具体的性能提升可能会因硬件配置和实际应用场景而有所不同。我们应该根据场景和 CPU 核心数目的不同灵活选取合适的算法和线程数。

5.3 反向传播结果分析

对于反向传播部分，为了测量方便起见，我们直接建立了一个完整的单层全连接神经网络，采用的参数如下表所示；同时，为了探究 CPU 线程核心数目对于 pthread 编程加速比的影响，我们做了两种实验：

- 实验一：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。
- 实验二：在线程数相同的情况下，我们不断增大数据规模，在 num_pthread 为 4 和单线程的时候，测定程序运行时间，并求出对应的加速比，以探究在理论加速比一定的情况下，实际加速比随着数据规模增大的变化趋势。
- 对于实验一的数据规模，如表5所示，结论如表6；对于实验二的结论，我们在表7中给出。

表 5: 采用的参数

input size	hidden size	epochs	num sample
30000	2000	10	100

表 6: 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

表 7: Performance comparison with different num_sample

input size	hidden size	num_sample	num_pthread=4	num_pthread=1	加速比
30000	2000	25	10.9	24.55	2.25
30000	2000	50	21.26	49.12	2.31
30000	2000	75	35.06	73.48	2.09
30000	2000	100	46.67	97.9	2.1
30000	2000	200	90.86	196.3	2.16
30000	2000	300	133.42	295.05	2.21
30000	2000	400	181.9	393.91	2.17
30000	2000	500	221.98	492.9	2.22
30000	2000	1000	454.13	979.73	2.16

为了更直观的看到趋势等因素，我们做了图5.3以展示在数据规模一定时，运行时间与线程数的关系；图5.4以展示加速比和损耗随着线程数增大的变化趋势；图5.5以展示在线程数一定时，程序运行时间与数据规模的关系。

我们考虑对结果进行分析：对于实验一的数据：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。

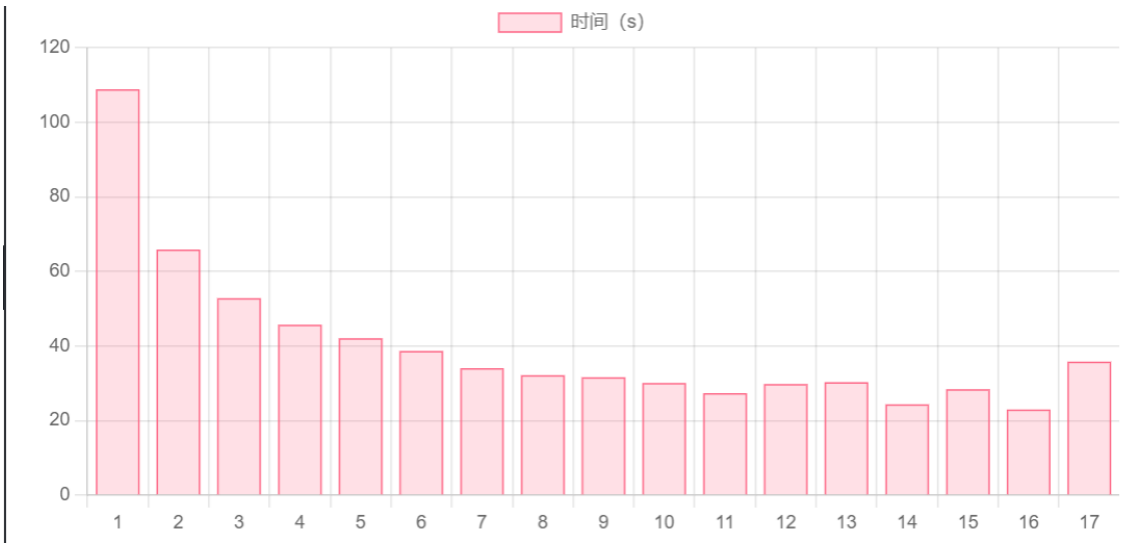

图 5.3: 运行时间与线程数关系图

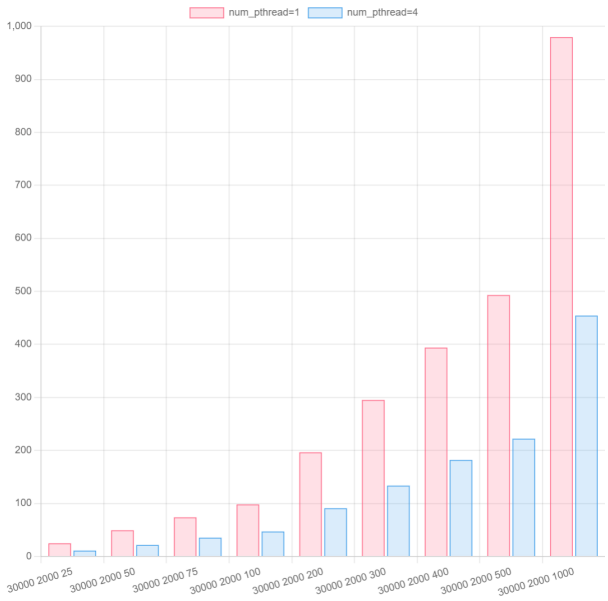

图 5.4: 加速比损耗与线程数关系图

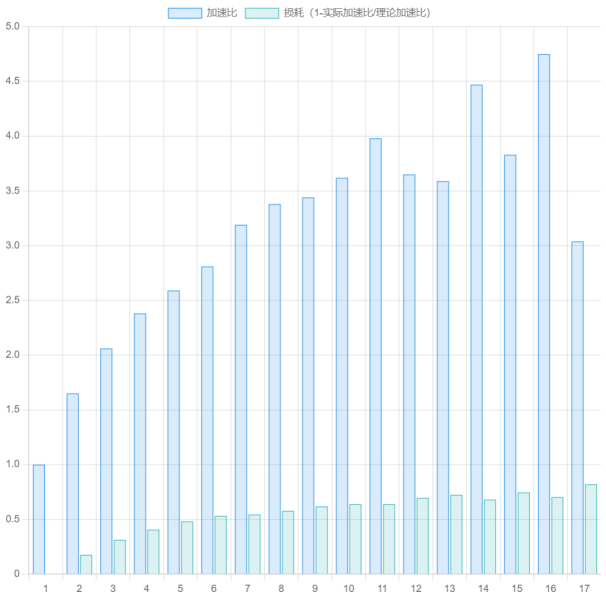

图 5.5: 线程数一定时，程序运行时间与数据规模关系图

对于实验二的数据，如图5.5所示，根据给出的数据，我们可以观察到：

- 随着数据量的增加，加速比整体上呈现略微下降的趋势，但差别不是很大，波动也较大。这可能是因为随着数据量增加，计算量也增加，但是由于任务的细粒度不同，所以并行效率并没有明显的提升。
- 对于同样的数据量，在样本数量逐渐增加的情况下，加速比也逐渐增加。这是因为任务的细粒度更小，线程之间的负载更均衡，可以更好地利用并行计算资源。
- 当样本数量大于等于 200 时，加速比整体上趋于稳定，变化不大。这可能是因为此时任务已经足够细粒度，可以较好地利用多线程计算资源，进一步增加样本数量不会带来明显的效果提升。

- 在样本数量较小的情况下，加速比的增长比较明显，但是当样本数量增加到一定程度时，加速比的增长开始趋于平缓。这说明在计算任务较小的情况下，多线程计算的效果更加显著。

综上所述，如果数据集较大，使用多线程并行化处理是很有必要的。但是，由于线程切换等开销，当数据集不断增大时，多线程并不一定能够带来线性加速比的提升。因此，在使用多线程并行化处理时，需要根据具体情况选择合适的线程数量，以最大化性能的提升。

6 Profiling 分析

我们针对上述设计的代码进行 profiling 分析，试图找到并行编程优化算法的底层原因，对于不同算法的参数对比如表8和表9所示：

表 8: 不同卷积算法 Profiling 参数对比

算法	CPI	指令数	LLC 占用	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%
pthread 优化	0.721	168424400000	0%	11.072	2.212	15.40%
pthread&AVX 优化	0.705	29890800000	0%	3.785	4.736	7.80%
openmp 优化	0.261	134918000000	0%	2.054	7.92	24.00%
openmp&AVX 优化	0.700	7555500000	0%	1.188	1.165	6.10%

我们对上述数据进行说明分析，分析结果如下所示：

- CPI 是 CPU 周期数和指令数的比率，反映了 CPU 执行指令的效率。从数据中可以看出，朴素卷积算法的 CPI 最高，为 0.713，说明执行每条指令需要的 CPU 周期数较多；而经过 pthread 优化后的算法 CPI 略有提高，为 0.721，说明优化后的算法执行每条指令所需的 CPU 周期数相对减少。而使用 AVX 指令集后的算法 CPI 有所下降，说明 AVX 指令集对算法性能有一定的优化作用。

- 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。可以看出，不同优化方式下的算法指令数基本保持不变，都在 1.3 亿左右；只有使用 SIMD 优化策略的指令数相对较少，一般而言要少一个数量级左右，这就是为什么 SIMD 编程的速度最快。

- LLC 是指 CPU 中最后一级缓存，它是 CPU 和主存之间的缓存。从数据中可以看出，不同优化方式下的算法 LLC 均为 0%，说明算法在执行过程中没有 L3 缓存的命中。说明 L1 和 L2 缓存中已经完成算法了。

- Elapsed Time 是算法执行的总时间。从数据中可以看出，朴素卷积算法的 Elapsed Time 最长，为 28.735 秒；而经过 pthread 和 openmp 优化的算法的 Elapsed Time 均有较大的下降，分别为 11.072 秒和 2.054 秒，说明多线程优化对算法性能的提升非常显著。而使用 AVX 指令集后的算法 Elapsed Time 也有所下降，说明 AVX 指令集对算法性能的优化也是有效的。

- CPU 时间是 CPU 执行算法的总时间，是算法性能的一个重要指标。从数据中可以看出，经过 pthread 和 openmp 优化的算法的 CPU 时间都有较大的下降，分别为 2.212 秒和 7.92 秒，说明多线程优化对算法性能的提升非常明显；而使用 AVX 指令集后的算法 CPU 时间也有所下降，说明 AVX 指令集对算法性能的优化也是有效的。

- 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，是一个重要的性能指标。从数据中可以看出，pthread 优化后的算法逻辑核心利用率最高，为 15.4%，说明多线程优化对 CPU 的利用率有非常显著的提升；而 openmp 优化后的算法逻辑核心利用率也比朴素卷积算法有明显的提高，达到了惊人的 24%，说明在卷积这个基础的操作上，OpenMP 多线程编程要略优于 pthread 编程，具体原因请见后面的讨论。

表 9: 不同矩乘算法 Profiling 参数对比

算法名称	CPI	指令数	LLC	Elapsed Time (s)	CPU 时间 (s)	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	8.676	23.70
pthread&AVX 优化	0.778	3300500000	0	2.2	0.588	1.70
openmp 优化	0.759	50457400000	0	2.258	8.588	23.70
openmp&AVX 优化	0.819	10856000000	0	2.429	1.945	5.00

同样的，我们也将针对上述参数做相应的说明：

上组涉及不同优化方案对于矩阵乘法算法的性能影响。从 CPI 的角度看，所有的优化方案都成功地将 CPI 值降低到了朴素算法的水平以下，表明了优化方案都能够有效地减少指令的数目或是提高指令的并行度。

从指令数和 LLC 的角度来看，朴素算法和 OpenMP 优化方案在指令数和 LLC 的使用上表现得比较一致，而 Pthread 优化和 OpenMP&AVX 优化方案使用的 LLC 都是 0，这表明这些优化方案都通过增加缓存命中率来提高效率 (将缓存命中到了 L1 和 L2 上面)。而 Pthread&AVX 优化方案的指令数远远少于其他优化方案，表明其使用了 AVX 指令集来减少指令的数目，提高效率。

从时间和 CPU 时间的角度来看,所有的优化方案都明显地提高了程序运行的效率,而 Pthread&AVX 优化方案的时间和 CPU 时间则最小，进一步证明了使用 AVX 指令集可以有效地提高程序的效率。

从逻辑核心利用率的角度来看，所有的优化方案都明显地提高了逻辑核心的利用率，表明多线程的并行处理能够充分地利用 CPU 的性能。

因此，综合以上数据分析，多线程编程可以通过并行处理和优化方案的选择来有效地提高程序的运行效率。

7 实验总结与反思

7.1 OpenMP 和 Pthread 的异同

OpenMP 和 Pthread 是两种并行编程的技术，它们的主要区别在于编程模型和实现方式上。

OpenMP 采用了一种基于指令集的编程模型，这种编程模型是面向共享内存的，并且它采用了一种基于指令插入的方式来实现并行化，这种方式可以在编译时自动插入并行代码，从而简化了并行编程的复杂性，减少了开发人员的工作量。OpenMP 通常适用于循环计算、矩阵运算等密集计算型任务，并且可以很好地适应多核处理器的计算能力，从而提高了程序的执行效率。

Pthread 则采用了一种基于线程的编程模型，这种编程模型是面向多线程的，并且它需要开发人员手动控制线程的创建、销毁和同步等操作，从而使得并行编程的复杂度相对较高。Pthread 通常适用于一些需要精细控制和灵活度较高的任务，比如网络编程和操作系统等。

当需要进行大规模的并行计算时，OpenMP 通常更为适合，因为 OpenMP 能够简化并行编程的复杂性，提高程序的可读性和可维护性，并且具有较高的执行效率。而当需要进行一些精细控制和灵

活度较高的任务时，Pthread 通常更为适合，因为 Pthread 能够提供更多的线程管理和同步操作，从而更好地满足任务的需求。

在本次实验中，我们发现，对于卷积和矩阵乘法操作，OpenMP 编程的速度显著好于在同等数据规模下的 pthread 编程，矩阵乘法和卷积操作是涉及到矩阵乘法或卷积操作的计算密集型任务。OpenMP 和 Pthread 都是用于并行化程序的库，但是在处理计算密集型任务时，OpenMP 往往比 Pthread 更适合；除此之外，我们分析了可能的原因，如下所示：

- OpenMP 中的线程创建和销毁比 Pthread 更快，这是因为 OpenMP 库使用的是线程池模型，而 Pthread 需要每次创建和销毁线程。这意味着在 OpenMP 中，线程创建和销毁的开销更小。
- OpenMP 提供了易于使用的并行化指令（例如 #pragma omp parallel for），可以方便地将代码块并行化。相比之下，Pthread 需要显式地创建和管理线程，这需要更多的代码和复杂的调试。
- OpenMP 的任务调度器（scheduler）具有更好的性能。在多线程并行计算中，任务分配的负载均衡对于并行性能至关重要。OpenMP 的任务调度器能够更好地负载均衡，从而更有效地利用多个线程。
- OpenMP 提供了更多的编译器优化支持。由于 OpenMP 使用更少的代码和易于并行化的指令，编译器可以更容易地进行优化。

在实际应用中，OpenMP 和 Pthread 都有其优缺点，具体选择哪一种并行编程技术，需要根据任务的具体需求和环境条件来确定。如果任务需要较高的并行效率，并且硬件条件允许，那么可以优先考虑使用 OpenMP；如果任务需要精细控制和灵活度较高，并且需要兼容多个操作系统和平台，那么可以优先考虑使用 Pthread。同时，还需要考虑开发人员的技能水平、开发周期、维护成本等方面的因素。

根据我们测出的实验数据，我们可以看到多线程编程在加速程序运行方面是非常有效的。首先，针对朴素卷积算法和矩阵乘法算法的优化实验，可以看到使用线程编程（pthread 和 openmp）的加速比明显高于朴素算法。同时，使用 AVX2 指令集优化后，可以进一步提高程序的效率。对于卷积算法而言，相比较于线程编程，使用 AVX2 指令集的加速比更高，而对于矩阵乘法算法而言，AVX2 指令集和线程编程的加速比差不多。

同时，可以发现在所有实验中，CPU 时间都远远小于 Elapsed Time，这表明程序并没有完全利用逻辑核心的资源。逻辑核心利用率最高的实验是朴素卷积算法，但即使在这个实验中，逻辑核心利用率也只有 5.7%。这提示我们在优化程序的同时，应该着重考虑如何更好地利用计算机的资源，例如通过进一步优化算法或者调整程序设计等方式。

另外，我们还可以观察到程序的运行时间随着数据规模的增大而变化。在所有实验中，随着数据规模的增大，程序的运行时间也呈现出明显的增长趋势。因此，在实际应用中，需要根据具体的情况，选择适当的数据规模和算法优化方式，以达到最优的性能表现。

最后，我们可以总结出以下结论：

- 多线程编程可以显著提高程序的运行效率，尤其是针对计算密集型任务。
- 使用 AVX2 指令集可以进一步提高程序的效率。
- 程序并没有充分利用计算机资源，需要在优化程序的同时，着重考虑如何更好地利用计算机的资源。
- 程序的运行时间随着数据规模的增大而变化，需要根据具体情况选择适当的数据规模和算法优化方式。

- 在矩阵乘法和卷积两个实验中，openmp 和 pthread 的表现有些不同。在矩阵乘法实验中，openmp 优化的速度相对更快，加速比也更高，特别是当矩阵规模较大时。而在卷积实验中，pthread 优化的速度更快，加速比也更高，特别是当卷积核规模较大时。可能的原因是因为在不同的实验中，数据读取和计算的比例不同，因此多线程优化的效果也不同。此外，openmp 和 pthread 之间的性能差异也可能受到特定的系统架构和配置影响。
- 需要注意的是，虽然在卷积实验中，pthread&AVX 优化的 CPI 最低，但是其逻辑核心利用率较低，表明线程的并行度不够高，不能充分利用系统资源。因此，在进行多线程优化时，需要综合考虑性能指标和资源利用率。

本次实验的相关的代码都可以在**超链接** 并行 GitHub 项目链接中找到，同时，对于原始的实验数据和 profiling 的项目文件，将一并上传到该项目链接中。

7.2 实验分工

本次实验的前期搜集资料，阅读相关文献的过程由张铭徐完成，针对后期的 profiling 数据分析由张惠程完成；此外，张铭徐负责了 pthread 优化算法，张惠程使用 OpenMP 优化算法，对于神经网络部分代码一同完成，并一起讨论出了上述的实验结论。