南開大學
Nankai University

计算机学院

并行程序设计实验报告

GPU 编程

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 25 日

目录

1 实验目的和实验背景知识介绍

1.1 实验背景知识介绍

1.1.1 卷积背景知识介绍

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1.1所示：

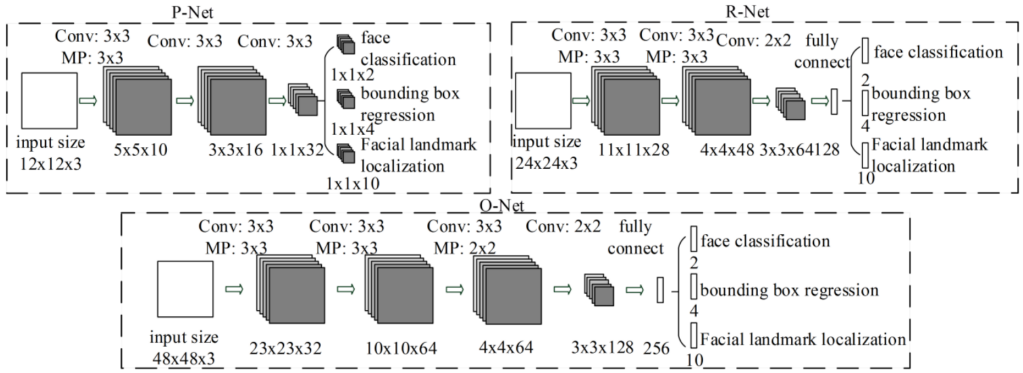

图 1.1: MTCNN's Framework

1.1.2 PCA 背景知识介绍

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。

1.1.3 GPU 背景知识介绍

我们在本次实验中，采用 Intel 的 OneAPI 套件进行实验。在进行本次实验之前，我们需要先了解一下什么是 GPU 编程。

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。

1.2 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 Core 12th，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 Core 11th, 操作系统和编译器同上。

2 SJTU-练习三

2.1 实验说明

我们在这里，首先完成 SJTU 培训中的练习三：练习三要求我们测试给定矩阵乘法不同的 tile_X 和 tile_Y 大小（tile_X 和 tile_Y 的大小可以不同）的矩阵计算性能，并分析结果并找出原因。

2.2 代码解读

给定的代码是在进行矩阵乘法运算的计算过程，并通过调整 tileX 和 tileY 参数来实现矩阵分块计算。

Algorithm 1 给定矩阵乘法算法流程图

输入: 矩阵 A, B, C, 矩阵维度 M, N, K, 工作组大小 $BLOCK$, SYCL 队列 q
输出: 计算结果矩阵 C，错误码 $errCode$

初始化矩阵 A, B, C 和 C_host
初始化计时器变量
for *run in [0, iterations + warmup)* **do**
    计算持续时间 ← GPU 内核 (A, B, C, M, N, K, $BLOCK$, q)
    if $run \geq warmup$ **then**
        累加 GPU 计算时间
    end
end
for *run in [0, iterations/2 + warmup)* **do**
    计算持续时间 ← CPU 内核 (A, B, C_host, M, N, K)
    if $run \geq warmup$ **then**
        累加 CPU 计算时间
    end
end
$errCode$ ← 验证 (C_host, C, $M \times N$)
输出性能 Flops 和 GPU/CPU 计算时间

在矩阵乘法中，将输入矩阵按照一定的大小划分为多个子矩阵，然后使用并行计算来加速矩阵乘法运算。tileX 和 tileY 表示子矩阵的大小，即每个子矩阵的行数和列数。这样做的目的是利用计算设备（如 GPU）的并行计算能力和局部内存来提高矩阵计算的性能。

具体而言，该代码中的 gpu_kernel 函数实现了使用 SYCL（异构编程框架）进行矩阵乘法的并行计算。在该函数中，通过将输入矩阵划分为大小为 tileX 和 tileY 的子矩阵，并使用局部内存存储子矩阵的部分数据，以利用数据的局部性和缓存优化。然后，使用并行计算单元（工作组）对划分的子矩阵进行计算，并将计算结果写回到输出矩阵中。

通过调整 tileX 和 tileY 的大小，可以影响矩阵分块计算的效果。较小的块大小可能会导致更多的数据传输和寄存器使用，从而增加通信和寄存器的开销。较大的块大小可能会限制并行性，并导致局部内存的利用不充分。因此，通过实验和分析不同的 tileX 和 tileY 大小，可以找到最佳的块大小，以实现最佳的性能。

总之，tileX 和 tileY 参数的调整可以优化矩阵乘法的性能，通过合理的矩阵分块计算和并行化，充分利用计算设备的并行计算能力，提高计算效率。具体的算法流程图如上图1所示。我们将在结果分析中给出具体的计算结果和对应的结果分析过程。

3 PCA(主成分分析)GPU 优化

3.1 朴素算法设计思想

Algorithm 2 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。

3.2 GPU 优化算法设计思想

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。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 3 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 ← 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 ← 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] ← 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] ← data[i * numFeatures + j] − result[groupId]
        end
    end
end
```

4 实验结果分析

4.1 SJTU 实验三结果分析

我们首先给出 SJTU 的实验结果，如1所示，接下来，我们根据测试的数据，从代码的角度进行详细分析，以了解不同的 tile 大小对性能的影响。

首先，我们来看一下代码中的关键部分，即 gpu_kernel 函数和 gemm 函数。在 gpu_kernel 函数中，通过将计算任务划分为网格和工作组，利用 GPU 的并行计算能力进行矩阵计算。具体来说，grid_rows 和 grid_cols 变量确定了网格的大小，local_ndrange 和 global_ndrange 则指定了工作组的大小。通过循环遍历网格中的每个元素，并在每个工作组中进行矩阵计算，从而实现了并行计算。

在 gemm 函数中，首先分配了共享内存用于存储输入矩阵 A、B 和输出矩阵 C 的数据。然后，随机初始化输入矩阵 A 和 B 的值，并将输出矩阵 C 初始化为零。接下来，通过循环执行 GPU 计算和 CPU 计算，并测量它们的执行时间。最后，通过比较 CPU 和 GPU 计算的结果来验证计算的正确性，并输出性能信息。

每个工作组 tileX 和 tileY 是表示矩阵计算中的工作组尺寸的变量。在 GPU 编程中，工作组是并行计算的基本单位，它由多个处理单元（线程）组成。tileX 和 tileY 变量决定了每个工作组在二维矩阵中处理的数据块的大小。tileX 表示工作组在 X 方向（列方向）上的尺寸，而 tileY 表示工作组在 Y 方向（行方向）上的尺寸。

通过将矩阵任务划分为网格和工作组，每个工作组负责处理一个小的数据块，可以充分利用 GPU 的并行计算能力。在代码中，tileX 和 tileY 的取值决定了每个工作组的大小，影响了计算任务的划分和并行执行。

通过调整 tileX 和 tileY 的值，可以控制每个工作组处理的数据块的大小，从而影响 GPU 的计算负载和性能。不同的应用和硬件配置可能需要不同的 tileX 和 tileY 取值以实现最佳性能。

在这组数据中，我们可以观察到以下一些关键点：

- tile 大小的变化：根据给定的数据，我们可以看到 tile_X 和 tile_Y 在不同的情况下具有不同的取值，例如 2、4、8、16、32 和 64。这些值决定了每个工作组的大小，即每个工作组处理的数据块的大小。
- GPU 计算时间的变化：根据数据可以看出，在不同的 tile 大小下，GPU 计算时间有所不同。例如，在 tile_X=2 和 tile_Y=2 时，GPU 计算时间为 162.572946 ms；而在 tile_X=64 和 tile_Y=64 时，GPU 计算时间为 2321.789087 ms。从数据中可以看出，随着 tile 大小的增加，GPU 计算时间呈现出增加的趋势。
- CPU 计算时间的相对稳定性：与 GPU 计算时间相比，CPU 计算时间相对稳定，不受 tile 大小的显著影响。在给定的数据中，CPU 计算时间大致在 60-65 ms 之间波动，不论 tile 大小如何变化。这是因为我们没有改变输入运算矩阵的规模，仅改变了 GPU 运行的工作组数据块处理大小，故我们的 CPU 运行时间相对平稳。
- 随着 tile 大小的增加，GPU 计算时间相对增加。这是因为较大的 tile 大小可能导致更多的计算和内存访问，从而增加了 GPU 的计算负载。
- CPU 计算时间相对稳定，不受 tile 大小的影响。这是因为 CPU 采用了不同的计算策略，较小的 tile 大小可能对其计算性能没有明显的影响。

需要注意的是，我们得出的结论是基于上述的数据和代码的分析，不一定具有泛用性，不同的情况可能因硬件、算法和数据集的不同而有所不同。为了得到准确的结论和最佳性能，我们通常需要进行更多的性能测试、优化和调整。

tile_X	tile_Y	GPU Computation Time	CPU Computaiton Time
2	2	162.572946	63.465625
2	4	395.831476	61.992055
4	2	396.520572	61.516650
4	4	**1195.600964**	63.378084
8	8	294.300598	63.693671
16	16	134.053212	63.423185
32	32	347.93992	64.131882
64	64	2321.789087	62.008715

表 1: 不同 tileX 与 tileY 运行时间

我们可以看到当工作组为 4 的时候数据明显出现了异常，并且在 tile=16 的时候 GPU 运行时间最短，我们考虑最短的原因：较大的工作组大小有助于提高数据的局部性，减少了内存访问的延迟。这可能导致更高的并行计算效率和更短的运行时间。

而 tile 为 4 的时候数据出现了异常，我们推测可能是因为在代码的核心计算部分，可能存在数据依赖性，使得并行计算时需要进行同步或等待操作。当 tileX 和 tileY 的大小为 4 时，可能导致数据依赖性的问题变得更加突出，从而影响了并行计算的效率。

我们将对应的数据整理为可视化图表，如下所示：

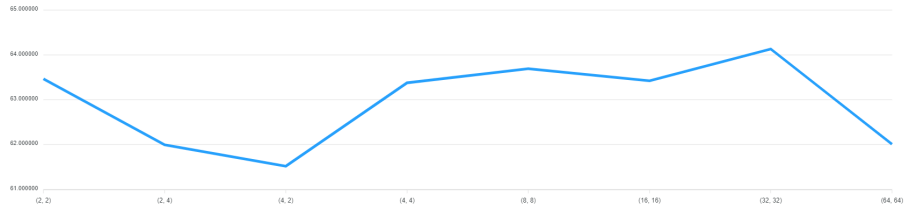

图 4.2: CPU 执行时间图

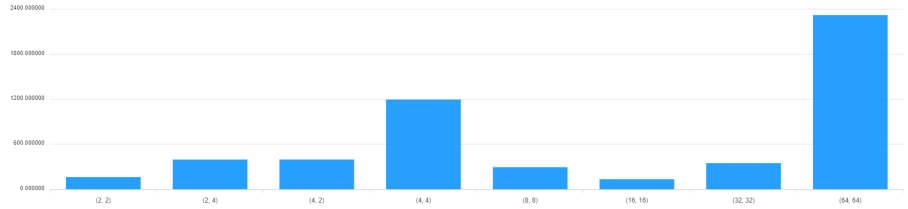

图 4.3: GPU 执行时间图

4.2 PCA 结果分析

我们对于 PCA 算法，执行了不同工作组大小的算法，并且使用 chrono 库记录了程序运行时间，以毫秒为单位，如下表2所示；我们还针对此，整理出了对应的可视化图表，如图4.4和图4.5所示：

表 2: PCA 算法执行时间

n	m	k	CPU	GPU (64)	GPU (128)
1000	1024	128	248	3107	3096
2000	1024	128	469	3204	3215
3000	1024	128	698	3486	3489
4000	1024	128	953	3878	3878
5000	1024	128	1217	4446	4447
6000	1024	128	1420	8513	8632
7000	1024	128	1653	8626	8596
8192	1024	128	1955	8839	8932

我们考虑对结果进行分析：首先我们给出具体代表的含义：

- 样本数（n）：从 1000 增加到 8192，以增加数据量和计算复杂性。
- 特征数（m）：固定为 1024，表示每个样本的特征维度。
- 组件数（k）：固定为 128，表示 PCA 保留的主要成分数量。
- CPU 时间：程序的 CPU 时间。
- GPU (64) 时间：使用 64 个工作组，GPU 执行程序花费的时间。
- GPU (128) 时间：使用 128 个工作组，GPU 执行程序花费的时间。

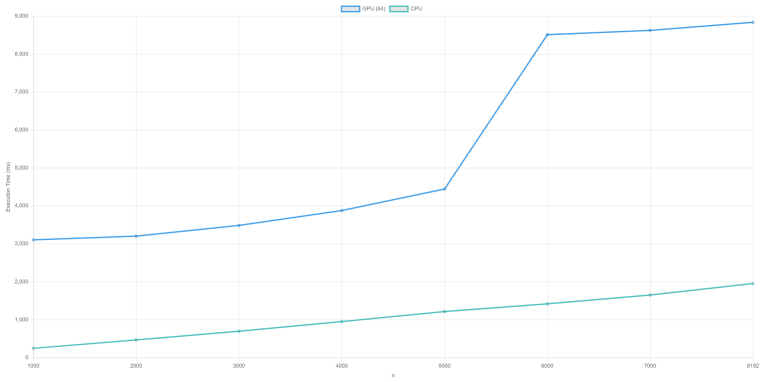

图 4.4: PCA 算法 GPU 和 CPU 运行时间对比图

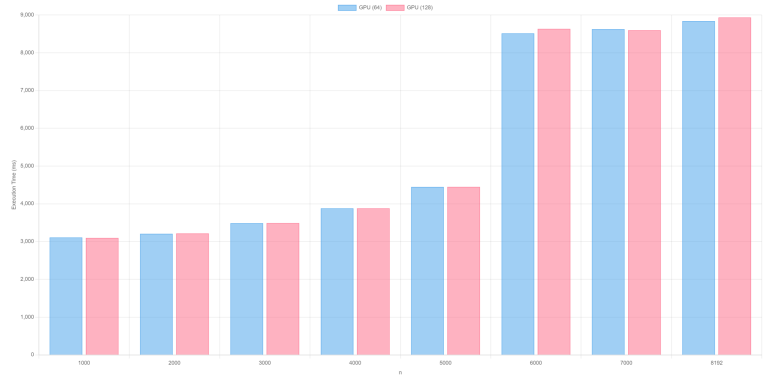

图 4.5: PCA 算法 GPU 运行时间对比图

我们考虑对数据进行分析：我们可以看到，随着数据规模的扩大，程序运行时间显然是要增长的，从图4.4我们可以看出，CPU 时间是稳定线性增长的，而 GPU 的执行时间缺出现了一些分歧：具体来说，在数据规模小于 6000 时，算法的运行时间相对较为合理，但是在 6000 时却发成了突变，在图中对应了一段斜率极大的直线；此外，我们采用工作组为 64 和 128 相比，绝大部分情况下，都是工作组大小为 64 时，运行时间要快于工作组为 128。下面我们将对上述的讨论结果进行详细的结果分析：

- 数据规模较小：在小规模数据上，GPU 的并行计算能力可能没有完全发挥出来，因此执行时间相对稳定且接近 CPU 的执行时间。
- 数据量增加导致的问题：当数据量增加到一定程度时，可能超出了 GPU 的处理能力范围，导致性能下降。这可能是因为 GPU 的资源有限，无法有效地并行处理大规模数据，从而导致执行时间的增加。
- 资源竞争：在较大规模数据上，可能会出现资源竞争的情况，例如工作组大小、寄存器分配等。这可能导致 GPU 的执行效率下降，从而影响了执行时间。
- 寄存器和共享内存的使用：较大的工作组可能需要更多的寄存器和共享内存来存储计算中间结果。当资源不足时，GPU 可能需要频繁地将数据从寄存器和共享内存中移出，导致额外的延迟和开销。较小的工作组可以更有效地利用有限的资源，从而获得更快的执行时间。
- 资源利用率：64 大小的工作组可能更好地利用了 GPU 的计算资源。较小的工作组可以更好地适应 GPU 的多处理器结构，并充分利用每个多处理器中的计算单元。相比之下，128 大小的工作组可能会导致资源利用率的下降，从而影响了执行时间。

5 实验总结与反思

5.1 实验总结

对于 PCA，卷积，SJTU 的练习 3，我们发现：CPU 的运行时间反而比 GPU 的运行时间快一些，而理论上讲，GPU 应当更加适合进行并行化操作，而我们实际操作来看，却和我们的结论截然相反，通过请教老师，我们得到了一些解释：

- 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，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。

此外，对于 GPU 编程，我们之所以用分布式计算的思路提高整个系统的并行计算的能力，主要原因是 CPU 的发展已面临物理定律的限制，无论是考虑核心数还是主频，人们无法在有限的成本下创造出核心数更多的 CPU；此外，由于功耗墙的限制，我们也没办法制造频率更高的 CPU，这样散热就得不到保证了。就在此时，原本用来做图形渲染的专用硬件，GPU 非常适合用于并行计算。

目前在人工智能的各大研究分支，例如深度学习，强化学习等，都在使用 GPU 进行优化，虽然 GPU 编程具有一些性能上的优势，但是编写代码的过程却极为复杂：例如 GEMM，原本的 GEMM 可能只需要 20 余行代码，但是在 GPU 并行的背景之下，却需要 100 余行。幸运的是，目前来说，GPU 对程序的优化已经下沉到了编译器的级别，使得我们无需手动编写一些复杂的代码就可以实现 GPU 加速。

例如 VSCode，下载 CUDA 驱动后，配置完毕环境变量，下载对应的支持 GPU 加速的库后，就可以实现使用 GPU 加速深度学习的训练，以个人的经验来看，在上学期的 Python 课程上，有一项任务是虚假新闻检测，如果使用 CPU 进行训练，需要将近 2 分半训练一轮，而使用了 GPU 加速后，仅仅需要 30s 即可训练一轮！

人工智能的发展是对算力的大挑战，我们在实现算法的同时，为提高训练效率，通常也需要静下心来思考，能否对算法有底层方面的提升。相信随着越来越多的算法和设计的加持，加之对底层体系结构的不断改良，人工智能的训练速度一定会越来越快，而不会让人望而却步！

5.2 实验分工

本次实验 GPU 部分的相关知识由张铭徐学习，然后与张惠程讨论得到 SJTU 两道题目的解决思路，对于 SJTU 部分，张铭徐负责第三题，张惠程负责第四题；对于自主选题部分，张铭徐完成了 PCA 相关部分的编写任务，并测定了程序运行时间，张惠程完成了卷积部分的代码编写任务，并测定了程序运行时间。最后实验结果的讨论部分由两人共同完成。

在本次实验中涉及到的代码，数据，运行时间图都可以在并行程序设计仓库中找到。