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：

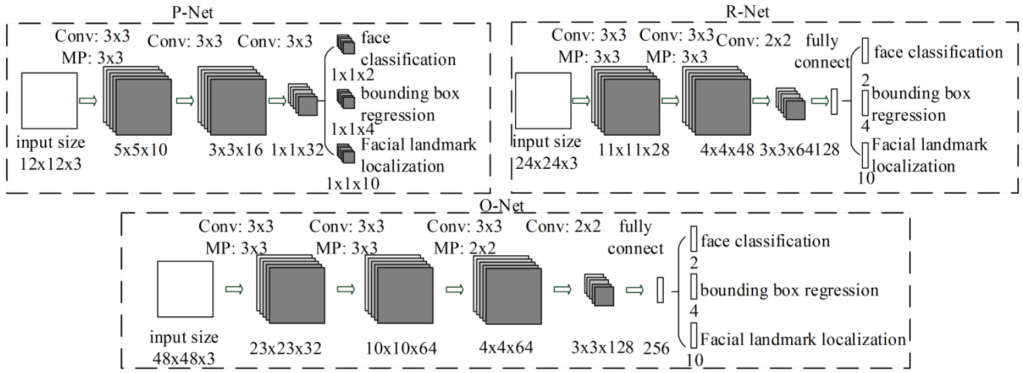

图 1.1: MTCNN's Framework

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-练习四

2.1 实验说明

我们在这里，首先完成 SJTU 培训中的练习四：练习四要求我们增加输入寄存器的大小以一次读取更多的行和列，并测试和分析性能。

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。我们知道：程序中使用的计算机内部寄存器，用于存储临时数据和中间计算结果。而在这段代码中，寄存器被用来存储子矩阵数据和计算结果。具体而言，有三个寄存器数组，其作用如下：

- float sum[tileY][tileX]：用于存储子矩阵相乘的累加和。
- float subA[tileY]：用于存储子矩阵 A 的一行数据。
- float subB[tileX]：用于存储子矩阵 B 的一列数据。

使用这些寄存器数组的目的是在 GPU 内核函数中实现数据的局部性和重用，以提高计算效率。

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

我们在实验中就是要增加或减少输入寄存器的大小来测定 CPU 时间和 GPU 时间。我们考虑增加寄存器的大小可以带来以下几个潜在的作用和好处：

- 提高数据局部性：通过增加寄存器的大小，可以存储更多的数据，减少对全局内存的访问。这可以提高数据的局部性，使得访问模式更加连续，减少内存访问延迟。
- 减少全局内存访问次数：寄存器中存储了更多的数据，可以减少对全局内存的频繁读写操作。由于全局内存访问通常是相对较慢的，减少访问次数可以显著提高程序的性能。
- 提高计算效率：增加寄存器的大小可以在计算过程中存储更多的中间结果，避免重复计算。这可以减少计算量，提高计算效率。
- 并行计算效率：在并行计算中，增大输入寄存器的大小可以让每个线程块或工作组处理更多的数据，从而充分利用并行计算资源，提高并行计算的效率。

不过需要我们注意的是，并不是寄存器数量越多越好，增加寄存器的大小也会占用更多的资源，特别是在 GPU 编程中。因此，在实际应用中，我们还是需要进行综合考虑，根据程序的特点和硬件的资源限制来确定合适的寄存器大小，以达到最佳的性能和资源利用率。

3 卷积操作实现 GPU 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2所示：

Algorithm 2 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 GPU 优化设计思想

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 3 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

4 实验结果分析

4.1 SJTU 实验四结果分析

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，以了解不同的寄存器大小对性能的影响。

在这段代码中，suba_reg 和 subb_reg 代表了输入寄存器的大小。这些寄存器用于存储子矩阵的数据，在计算过程中减少了对全局内存的访问。根据我们测试到的数据，我们可以观察到寄存器大小对程序执行时间的影响。首先我们给出输入寄存器大小的含义：

- suba_reg：表示存储子矩阵 A 的输入寄存器的大小。
- subb_reg：表示存储子矩阵 B 的输入寄存器的大小。

寄存器大小增加时，增加输入寄存器的大小可以存储更多的数据，从而减少了对全局内存的访问。这通常可以提高计算效率。同时，我们从计算量的角度考虑：矩阵乘法涉及大量的乘法和累加操作，较大的输入寄存器可以存储更多的中间结果，减少了重复计算的次数。接下来我们考虑对比不同寄存器大小的执行时间：

- 当 suba_reg 和 subb_reg 的大小较小时（如 2x2），GPU 计算时间较短（例如 162.817511 ms）。这可能是因为较小的寄存器大小可以在更快的速度加载和处理数据。
- 随着 suba_reg 和 subb_reg 的大小增加，GPU 计算时间也随之增加（例如 8x8 的情况下为 537.976575 ms）。这可能是因为较大的寄存器大小导致了更多的计算和数据加载操作，从而增加了计算时间。
- 我们考虑上述数据的异常情况：事实上，寄存器大小的选择需要综合考虑其他因素，例如硬件架构和资源限制。寄存器大小增加后，可能需要更多的时间来从全局内存加载数据到寄存器中；同时较大的寄存器可能会导致寄存器溢出，从而增加了存储器访问的延迟。可能存在其他因素或限制，例如数据依赖关系、内存带宽限制等，这些因素可能会影响程序的整体性能。

那么我们的结论是：寄存器大小的增加可以提高局部性和计算效率，但过大的寄存器大小可能会导致额外的计算和数据加载操作，从而影响程序执行时间。因此，在选择寄存器大小时，需要根据具体的硬件和算法特性进行优化和权衡。

表 1: 不同寄存器大小的程序运行时间

suba_reg	subb_reg	GPU Computation	CPU Computaiton
2	2	162.817511	63.465625
4	4	225.732675	62.965845
8	8	537.976575	64.124355
16	16	159.992853	63.321244
32	32	163.614352	63.78445
64	64	202.542703	63.441981

我们将对应的数据整理为可视化图表，如下所示：

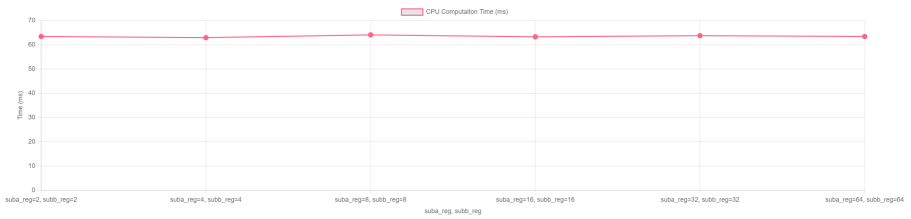

图 4.2: CPU 执行时间图

我们还测定了在 tileX=tileY=16 时，不同的寄存器大小程序的运行时间，如下表2所示：

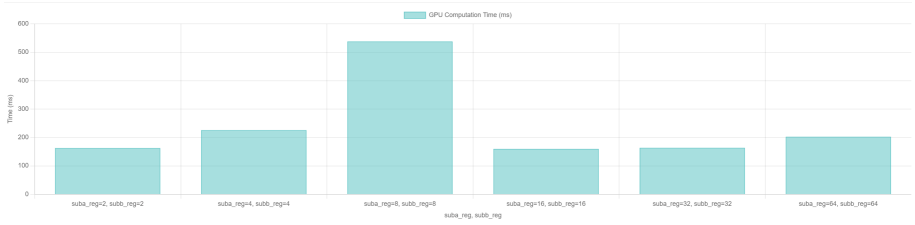

图 4.3: GPU 执行时间图

表 2: 不同寄存器大小的程序运行时间

suba_reg	subb_reg	GPU Computation	CPU Computaiton
16	16	136.631291	65.026806
32	32	132.943687	63.61852
64	64	132.825258	64.041306

我们可以看到，相比于 tileX=tileY=2 时，相同寄存器大小时，程序运行时间明显要更快一些，对应的原因可以参考张铭徐同学的实验报告 (本次实验的实验三就是在探究这件事情)。

4.2 卷积结果分析

我们对于卷积算法，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3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 3: 不同工作组大小和输入数据规模下的程序执行时间

signal_length	kernel_length	CPU Time(s)	GPU Time 128(s)	GPU Time 256(s)
100000	8000	1.41583	3.94428	4.06374
200000	16000	5.73004	11.7509	11.9157
210000	16800	6.24536	11.8978	10.213
220000	17600	6.85904	13.0322	11.857
230000	18400	7.57233	14.1802	12.7879
240000	19200	8.2076	15.3765	13.773
250000	20000	8.94796	16.5554	14.9997

我们考虑对结果进行分析：首先我们给出具体代表的含义：

- signal_length：输入数据规模。
- kernel_length：卷积核尺寸。
- CPU 时间：程序的 CPU 时间。
- GPU (128) 时间：使用 128 个工作组，GPU 执行程序花费的时间。
- GPU (256) 时间：使用 256 个工作组，GPU 执行程序花费的时间。

根据测试的数据，我们可以观察到以下情况：

- 随着输入数据规模的增加，CPU 和 GPU 的执行时间都呈现增长的趋势。这是因为处理更多数据需要更多的计算时间。在相同输入数据规模下，GPU 的执行时间通常比 CPU 的执行时间长，这是由于 GPU 的并行计算能力使其能够处理更多数据，但相应地也需要更多的时间。

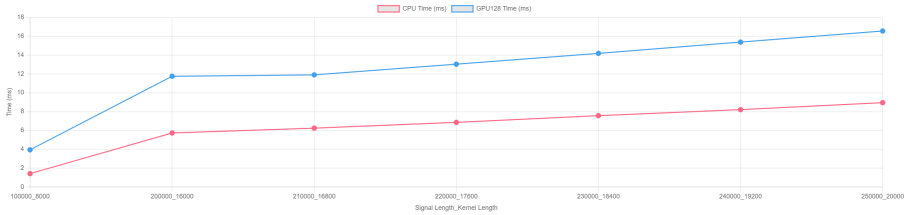

图 4.4: 卷积算法 GPU 和 CPU 运行时间对比图

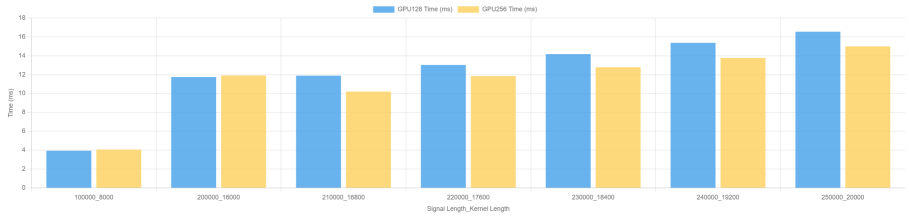

图 4.5: 卷积 GPU 运行时间对比图

- 工作组大小对 GPU 执行时间的影响：在给定输入数据规模下，不同工作组大小的 GPU 执行时间可能有所不同。在这组数据中，观察到当工作组大小为 128 时，GPU 的执行时间比工作组大小为 256 时短。这可能是因为具体的 GPU 设备在 128 工作组大小下能够更好地利用并行计算资源，从而提高了执行效率。

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。