南開大學
Nankai University

计算机学院

并行程序设计实验报告

GPU 编程

马浩祎

学号：2213559

专业：计算机科学与技术

2024 年 6 月 7 日

目录

1 摘要

Abstract

本次实验是 GPU 并行编程实验。首先学习了 **oneAPI 平台的三个基础部分**，第一部分主要介绍 oneAPI 在数据并行和异构编程中的作用，并初步介绍 SYCL 的基本知识，语法和编程模型。第二部分深入介绍 SYCL 基本类的概念和使用，缓冲区、访问器、命令组处理程序和内核编写 SYCL 程序的方法。第三部分在 USM 层面上介绍进行隐式和显式内存移动，解决内核任务之间的数据依赖问题的方法。然后在兴趣的驱使下进一步探索了**第 8 节使用并行内核执行归约操作**。在学习的过程中同时进行了详细的编程实践！实现了**平台提供的 4 个章节练习**，值得注意的是，我还进行了**自主选题的探索和编程实现**，给出了 oneAPI-SYCL 下的性能数据和解释分析。

关键词： GPU 编程，oneAPI，SYCL，列表求交

2 实验相关介绍

2.1 问题描述

2.1.1 倒排索引

早在 1958 年，IBM 就在一次会议上展示了一台“自动索引机器”。在当今的搜索引擎里也经常能看到它的身影，它就是——倒排索引。倒排索引又叫反向索引，它是一种逆向思维运算，是现代信息检索领域里面最有效的一种索引结构。

为满足用户需求，顺应信息时代快速获取信息的趋势，开发者们在进行搜索引擎开发时对这些信息数据进行了逆向运算，开发出“关键词——文档”形式的映射结构，实现了通过了物品属性信息对物品进行映射，可以帮助用户快速定位到目标信息，极大地降低了信息获取难度。

2.1.2 索引求交

有了倒排索引后，搜索引擎在查询处理过程中，需要对这些索引集合进行求交操作，这个也是运算时间占比非常大的部分。假定用户提交一个 k 个词的查询，查询词分别是 $t_1, t_2, \ldots, t_k$，这 k 个关键词对应的倒排列表为 $\ell_{t_1}, \ell_{t_2}, \ldots, \ell_{t_k}$ 求交算法返回 $\cap_{1 \le i \le k} l(t_i)$。

首先求交会按照倒排列表的长度对列表进行升序排序，使得：

$$|l(t_1)| \le |l(t_2)| \le \cdots \le |l(t_k)|$$

然后求交操作返回 k 个倒排列表的公共元素 $\cap_{1 \le i \le k} l(t_i)$。

2.2 GPU 简介

GPU（Graphics Processing Unit，图形处理单元）是一种专门用于处理图形计算任务的处理器，如图2.1。最初，GPU 主要用于加速图形渲染和图像处理，但随着技术的发展，GPU 的应用已经扩展到通用计算领域。

GPU 具有如下特点：

- 高并行计算能力：GPU 包含大量的计算单元（称为流处理器或 CUDA 核心），能够同时执行大量的简单计算任务。这使得 GPU 非常适合处理大规模并行计算任务，如图像处理、科学计算和机器学习等。

- 高速内存访问：GPU 通常具有高速显存（VRAM），提供高带宽的内存访问，适合处理大量数据。
- 可编程性：现代 GPU 支持可编程着色器，允许开发者使用图形 API（如 OpenGL、DirectX）或通用计算 API（如 CUDA、OpenCL、SYCL）编写复杂的计算程序。
- 专用硬件：GPU 包含专门的硬件单元，如纹理处理器、光栅化器、视频编码解码器等，能够高效地处理特定类型的计算任务。

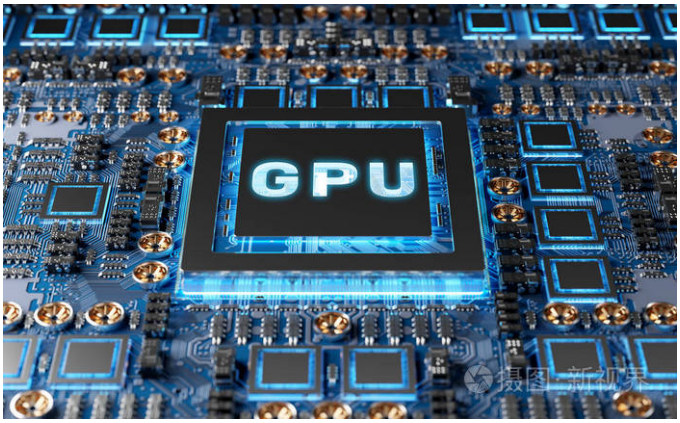

图 2.1: GPU

2.3 实验目的 & 实验环境

先前，我进行了许多并行化编程的实验，接触到了 SIMD，多线程，多进程等方法和很多编程工具。本次实验的实验目的简述如下：掌握 SYCL 基础知识和编程模型，熟悉 SYCL 基本类和设备选择，学习统一共享内存（USM）管理方法，在以上三者的基础上探索并行内核和归约操作并进行实际编程实践与分析，以最终提升我的 GPU 并行编程能力。

实验环境：本次实验统一在 Intel DevCloud 实验训练平台下进行。

2.4 学习证明 &GitHub 仓库

本次实验中数据部分不是报告重点，这里不再赘述，可参考仓库内先前的实验报告。仓库中还包含**本次实验涉及的练习程序，运行结果截图，以及自主选题的数据性能表**。

https://github.com/Mhy166/parallel-programming.git

这里篇幅所限只给出仓库 GPU-res 目录内的截图图2.2以作整体的证明，结果正确性详见仓库！

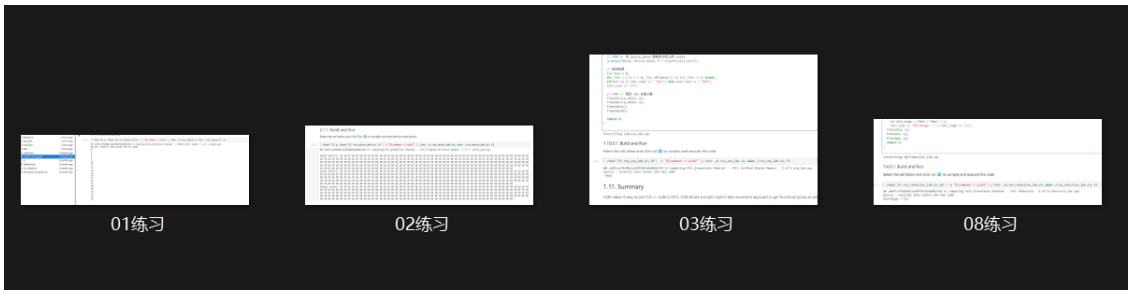

图 2.2: 整体证明

3 Essentials of SYCL 课程基础学习

本课程使用 oneAPI 和 SYCL c++ 来演示在英特尔云上提供的多个不同平台上实现高性能，可移植代码的方法。旨在介绍和教授 SYCL（异构计算领域的标准编程模型）编程的基础知识和核心概念。该课程的主要目标包括：

- 理解 SYCL 框架：帮助学生了解 SYCL 编程模型的基本概念和框架结构。
- 开发并行应用程序：指导学生如何使用 SYCL 编写并行计算应用程序，从而充分利用异构计算资源。
- 跨平台编程：教授如何在不同类型的硬件（如 CPU、GPU 和 FPGA）上编写和优化代码。
- 性能优化：提供有关如何利用 SYCL 实现性能优化的技巧和方法。

下面我们就来具体学习这门课程的基础部分！

3.1 01-oneAPI-Intro

本节课程是 oneAPI 的介绍引入部分，主要介绍了相关知识，并以一段简单的代码来帮助我理解 SYCL 程序。

3.1.1 oneAPI 编程模型简述及其挑战和解决方案

当前，数据中心的专用工作负载在增长，不同硬件需要不同的编程语言和库，导致开发人员必须维护多套代码库，并学习多种工具，增加了开发和维护的复杂性。

oneAPI 编程模型提供了一个跨多个硬件目标的统一开发工具组合，包括高性能库，支持 CPU、GPU 和 FPGA 等硬件，实现无损的高级语言性能。oneAPI 基于行业标准和开放规范，与现有高性能计算编程模型互操作，降低了多架构编程的复杂性和成本。

总的来说，oneAPI 通过统一的编程语言 DPC++ 和高性能库，使开发人员可以用相同的代码在不同硬件平台上运行，显著降低了开发和维护成本，提升了创新和生产力。oneAPI 的优势包括：统一编程模型、高性能库、行业标准和开放规范，以及互操作性。

3.1.2 代码初体验

首先我们介绍一下 SYCL：SYCL（Sick-el）是一种为跨不同硬件架构编写单一源代码的开源标准编程模型。它建立在标准的 C++ 基础上，旨在实现异构计算，它的定义是 “用于 OpenCL 的 C++ 单源异构编程”。与 OpenCL 一样，SYCL 标准由 Khronos Group 管理。与 OpenCL 不同，SYCL 包括模板和 lambda 函数，使高层应用软件能够以更清晰的方式编写，同时优化内核代码的加速。程序员可以在比 OpenCL 更高的抽象层次上进行编程，但始终可以通过与 OpenCL 以及 C/C++ 库的无缝集成访问底层代码。

与 SYCL 相关的，我们还需要了解数据并行 C++：这是 oneAPI 对 SYCL 编译器的实现。它利用现代 C++ 的高效性和熟悉的构造，并结合 SYCL 标准来实现数据并行和异构编程。SYCL 程序在主机计算机上调用，并将计算卸载到加速器。程序员使用熟悉的 C++ 和库构造，并增加了如队列用于工作目标管理、缓冲区用于数据管理以及 parallel_for 用于并行化等功能，以指示应卸载哪些部分的计算和数据。

最后在介绍代码前，我们需要知道 oneAPI 有着丰富的基于 SYCL 的规范的编程模型，这些编程模型详细定义了主机代码和设备代码所需的语言特性。以下是编程模型的内容摘要：

- 平台模型：主机控制多个设备，主机协调和控制设备上的计算工作。
- 执行模型：代码（内核）在设备上执行，主机通过命令组协调执行和数据管理。
- 内存模型：主机和设备之间的内存交互通过内存对象和访问器完成。
- 内核编程模型：显式并行性，程序员决定代码执行位置，支持单一源文件。

这些模型共同组成了 oneAPI 的编程框架，使开发人员能够高效地在多种异构硬件上开发高性能应用程序。我们的实验也都是基于 oneAPI 下的 SYCL 模型进行的，本小节课程给我们提供了第一个简单的 SYCL 程序，现在我们就来看一下代码，并结合先前所学的四个模型进行理解和解释（本节无练习）：

```
#include <sycl/sycl.hpp>  // 包含 SYCL 的头文件
using namespace sycl;  // 使用 sycl 命名空间
static const int N = 20;  // 定义一个常量 N，表示数组的大小
int main() {
  queue q;//# 定义一个队列，其关联了一个默认设备用于计算卸载
  std::cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";// \
  输出使用的设备名称
  //通过上面这两行，我们可以看到当前使用的设备名称，主机协调和控制设备上执行的计算任务。\
  这是平台模型的体现!
  //# 统一共享内存分配，允许在主机和设备上访问数据，这是内存模型的体现!
  int *data = malloc_shared<int>(N, q);
  //# 初始化数组
  for (int i = 0; i < N; i++) data[i] = i;
  //# 将并行计算卸载到设备上
  q.parallel_for(range<1>(N), [=] (id<1> i) {
    data[i] *= 2;  // 将每个元素乘以 2
  }).wait();  // 等待计算完成
  //这里的内核代码是一个 lambda 表达式，定义了如何处理数据，每个元素都乘以 2。\
  wait() 函数确保主机等待设备完成计算。这是执行模型和内核编程模型的体现!
  for(int i=0; i<N; i++) std::cout << data[i] << "\n"; // 打印数组的每个元素
  free(data, q); // 释放分配的内存
```

3.1.3 本章小结

对于本模块，我学习到了以下几点：

- 掌握 oneAPI 在多架构编程中的优势和应用。
- 了解如何在不同的硬件平台上实现高性能计算。
- 了解 SYCL 作为 oneAPI 的一部分，其在数据并行和异构编程中的作用，并掌握 SYCL 的基本知识，语法和编程模型，包括内核编程、内存管理和执行模型。
- 掌握如何在 Jupyter 笔记本中编辑、保存和运行 SYCL 代码，以开发自己的程序。

3.2 02-SYCL-Program-Structure

本节课程是 SYCL 编程结构的深入学习部分。非常值得介绍 SYCL 的相关结构知识!

对其本身来说，SYCL 语言和运行时由一组 C++ 类、模板和库组成。这里有两个运行范围需要我们注意区分:

- 应用范围和命令组范围：在主机上执行的代码, 在应用范围和命令组范围内可以使用 C++ 的全部功能。
- 内核范围：在设备上执行的代码, 在内核范围内，接受的 C++ 功能存在限制。

3.2.1 SYCL Classes

我们在先前的并行课程学习中不难发现，不论是 pthread 还是 OpenMP，还是 MPI，都有一个其对应的库，里面对应实现了一些类和方法！同样的，SYCL 框架也提供了很多类，需要我们学习并能利用它们分析编写程序。具体如下:

- Device 类包含用于查询设备信息的成员函数，这对于创建多个设备的 SYCL 程序非常有用。
- Device Selector 设备选择器类允许在运行时根据用户提供的启发式方法选择特定设备来执行内核。
- Queue 类用于提交由 SYCL 运行时执行的命令组。队列是将工作提交给设备的机制。一个队列对应一个设备，多个队列也可以映射到同一个设备。
- Kernel 类封装了在设备上执行代码的方法和数据。当命令组被实例化时，内核对象自动构建。用户不需要显式地构建内核对象，而是在调用内核调度函数（如 parallel_for）时自动构建内核对象。
- 设备选择器的类型丰富多样，除了单个的固定设备，还可以设置用户自定义的选择器和多 GPU 选择器。

下面我们简要看看平台上具体的设备，关键代码如下:

```
//……
    queue q(gpu_selector_v);
    //queue q(cpu_selector_v);
    //queue q(accelerator_selector_v);
    //queue q(default_selector_v);
    //# Print the device name
//……
```

这里我为了学习实践，通过更改上述设备的选择，我看到了运行结果，了解到了云平台上的各种设备型号:

```
  gpu_selector_v: Intel(R) Data Center GPU Max 1100
  cpu_selector_v: Intel(R) Xeon(R) Platinum 8468V
```

```
default_selector_v: Intel(R) Data Center GPU Max 1100
accelerator_selector_v: 平台尚未有此设备类型
```

3.2.2 并行内核

并行内核允许操作的多个实例并行执行。这对于卸载基本 for 循环的并行执行非常有用，其中每次迭代是完全独立的，并且可以以任何顺序执行。并行内核使用 parallel_for 函数表达。这里提供了一个简单的更改示例：

```
//简单示例
for(int i=0; i < 1024; i++){a[i] = b[i] + c[i];});
//卸载到加速器的代码更改如下：
q.parallel_for(range<1>(1024), [=](id<1> i){A[i] =  B[i] + C[i];});
```

对于并行内核，其还有两种实现方式，这是值得我学习的：

- 基本并行内核：通过 range、id 和 item 类实现。range 类描述迭代空间，id 类索引单个内核实例，适用于仅需索引的情况；item 类适用于范围。上述代码使用 id 类是合适的，因为只用到了索引 i。
- ND-Range 内核：通过 nd_range 和 nd_item 类实现更低级的性能调优。nd_range 类表示分组的执行范围，nd_item 类表示内核函数的单个实例，并查询工作组范围和索引。这种方式允许更细粒度的性能调优和资源管理，提升硬件级别的并行计算效率。

3.2.3 内存模型

SYCL 的内存模型主要有两种：

- USM：统一共享内存模型是一种基于指针的内存模型，类似于 C/C++ 的基于指针的内存分配。多个内核之间的依赖关系使用事件显式处理。
- 缓冲区内存模型：缓冲区内存模型引入了一种新的内存抽象，称为缓冲区，并使用访问器进行访问。访问器允许设置读/写权限和其他内存属性。这种模型支持以 1、2 或 3 维表示数据，使得处理多维数据的内核编程更加容易。多个内核之间的依赖关系是隐式处理的。

平台还提供了一个对比代码，由于篇幅原因，这里不再介绍，有兴趣的读者可以查看仓库。关于同步机制，这里有必要介绍两个新部件：

- 主机访问器：它是一种使用主机缓冲区访问目标的访问器。它在命令组范围之外创建，允许主机访问数据。通过构建主机访问器对象，可以将数据同步回主机。它主要适用于缓冲区内存模型。
- 缓冲区销毁：缓冲区的创建发生在一个单独的函数作用域内。当执行超出此函数作用域时，将调用缓冲区析构函数，从而放弃数据所有权并将数据复制回主机内存。这是另一种将数据同步回主机的方法。

3.2.4 本节练习

对于本节课程，在平台的 2.1 节有一个编程练习，我认为这是检验我学习成效的好办法，我也顺利完成了并得到了输出，下面看一下本练习的关键代码，(所有 STEP 下的都是我实现补全的)：

```
STEP 1: 创建第二个向量 vector2 并初始化为 20，然后打印其值。
  std::vector<int> vector2(N, 20);
  std::cout << "\nInput Vector2: ";
  for (int i = 0; i < N; i++) std::cout << vector2[i] << " ";
STEP 2: 为第二个向量创建缓冲区 vector2_buffer。
  buffer vector2_buffer(vector2);
STEP 3: 在内核代码中为第二个缓冲区创建访问器 vector2_accessor。
  accessor vector2_accessor(vector2_buffer, h, read_only);
STEP 4: 修改内核代码，使其将 vector2 加到 vector1。
  vector1_accessor[index] += vector2_accessor[index];
```

可以看到，这些补全的代码部分都是本节讲述过的关键内容，它们共同构成了 SYCL 编程框架。

3.2.5 本章小结

对于本模块，我学习到了以下几点：

- 掌握 SYCL 基本类的概念和使用。
- 掌握选择各种用于卸载内核工作负载的设备的方法。
- 掌握使用缓冲区、访问器、命令组处理程序和内核编写 SYCL 程序。
- 掌握如何使用主机访问器和缓冲区销毁进行同步。

3.3 03-SYCL-Unified-Shared-Memory

本节课程是 SYCL 内存结构的更深入学习部分。主要介绍统一共享内存。本节的知识内容较少，重点在代码方法的理解和撰写上，因此我的侧重点也有所倾向于代码介绍了。

3.3.1 USM 相关知识

统一共享内存（USM）是 SYCL 中的一种基于指针的内存管理方式。USM 是一种基于指针的方法，对于使用 malloc 或 new 来分配数据的 C 和 C++ 程序员来说，这种方法应该是熟悉的。USM 在将现有的 C/C++ 代码移植到 SYCL 时，简化了程序员的开发工作。其形象表示为图3.3

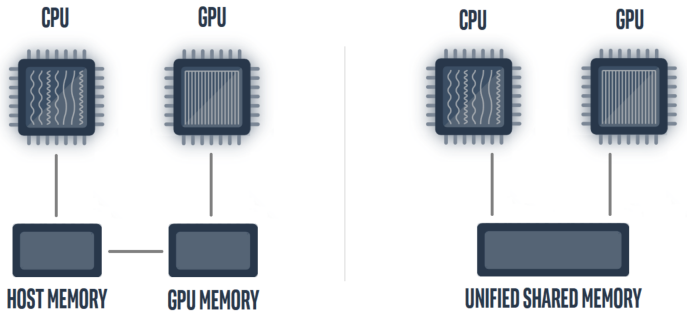

图 3.3: 有无 USM 的直观视角

课程介绍了其语法，我将在后面的代码部分介绍，这里暂且略过。这里介绍一下课程提到的 USM 的两种数据移动方式：

- USM 隐式数据移动。数据在主机和设备之间的移动是隐式进行的。这种方法有助于以最少的代码快速实现功能，开发者不必担心在主机和设备之间移动内存。其特点有点像栈。
- USM 显式数据移动。数据在主机和设备之间的移动需要由开发者使用 memcpy 显式完成。这种方法允许开发者更精确地控制主机和设备之间的数据移动。其特点有点像堆。

在使用统一共享内存时，由于任务是异步执行的，并且多个任务可以同时执行，因此必须使用事件来指定任务之间的依赖关系。程序员可以显式地等待事件对象完成，或者在命令组内部使用 depends_on 方法来指定必须完成的事件列表，然后任务才能开始。这里给出一些常用的方法:(1) 使用 wait() 等待内核任务 (2) 使用 in_order 队列属性 (3) 使用 depends_on 方法。

3.3.2 本节练习

这部分，我也顺利完成了并得到了输出，下面看一下本次练习的关键代码，(所有 STEP 内的都是我实现补全的)：

```
1
2 atic const int N = 1024;
3 t main() {
4 queue q;
5 std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";
6
7 // 初始化主机上的两个数组，不再赘述
8 // STEP 1: 创建 USM 设备分配
9 int *device_data1 = malloc_device<int>(N, q);
10 int *device_data2 = malloc_device<int>(N, q);
11 // STEP 2: 将 data1 和 data2 复制到 USM 设备分配中
12 q.memcpy(device_data1, data1, N * sizeof(int)).wait();
13 q.memcpy(device_data2, data2, N * sizeof(int)).wait();
14 // STEP 3: 编写内核代码，将 device_data1 的每个元素更新为其平方根
```

```
15 q.parallel_for(N, [=](auto i) {
16   device_data1[i] = std::sqrt(device_data1[i]);
17 });
18 // STEP 4: 编写内核代码，将 device_data2 的每个元素更新为其平方根
19 q.parallel_for(N, [=](auto i) {
20   device_data2[i] = std::sqrt(device_data2[i]);
21 });
22 // STEP 5: 编写内核代码，将 device_data2 的每个元素加到 device_data1 中
23 q.parallel_for(N, [=](auto i) {
24   device_data1[i] += device_data2[i];
25 }).wait();
26 // STEP 6: 将 device_data1 复制回主机上的 data1
27 q.memcpy(data1, device_data1, N * sizeof(int)).wait();
28 // 验证结果
29 int fail = 0;
30 for (int i = 0; i < N; i++) if(data1[i] != 12) {fail = 1; break;}
31 if(fail == 1) std::cout << " FAIL"; else std::cout << " PASS";
32 std::cout << "\n";
33 // STEP 7: 释放 USM 设备分配
34 free(device_data1, q);
35 free(device_data2, q);
36 free(data1);
37 free(data2);
38 return 0;
39
```

下面我针对本次练习讲一下反思：练习通过使用 malloc_device 和 malloc_shared 函数来分配设备内存或共享内存，让我熟悉 USM 的基本概念及其用法。练习中还使用了 memcpy 将数据从主机复制到设备内存，这是 USM 显式数据移动的一个例子。练习中也使用了 wait() 方法来确保内核任务按顺序执行，这是管理数据依赖的一种方式。最后，练习使用了 malloc_device 和 free 来分配和释放设备内存。对于执行过程，练习通过 parallel_for 函数提交并行任务，让我熟悉 SYCL 中如何编写并行代码，如何在设备上执行计算。

3.3.3 本章小结

对于本 USM 模块，我学习到了以下几点：

- 理解使用 USM 进行隐式和显式内存移动的方法。
- 掌握解决内核任务之间的数据依赖问题的方法。
- 进一步熟悉了内存分配与释放以及并行编程的基本概念和操作。

4 GPU 进阶学习

4.1 08-SYCL-Reduction

可以说我观察目录，对 Reduction 较感兴趣，而且可以正好可以与我们先前学过的并行方法进行对比来强化我的知识理解。下面我们就来介绍一下本节的主要内容：

4.1.1 Reduction

归约通过使用一个既结合又交换的操作符，以不确定的顺序将多个值组合成一个单一的值。对于并行程序员来说，只关心归约最终产生的值。并行化归约可能会比较棘手，因为它涉及到计算的性质和加速器硬件。

那么 SYCL 怎么进行的归约操作呢？在 SYCL 中，parallel_for 提供了多种使用 sycl::reduction 对象进行归约的方式，以实现高效的并行计算。以下是几种不同的归约方法：

- 使用 USM 的 parallel_for 归约: 该方法使用统一共享内存(USM)进行内存管理，通过 sycl::reduction 对象在 parallel_for 中执行归约，只需一个内核即可完成。
- 使用缓冲区的 parallel_for 归约: 方法使用 SYCL 缓冲区和访问器进行内存管理。
- 在一个内核中进行多个归约: 该方法支持在一个内核中使用多个 sycl::reduction 对象进行归约。
- 使用自定义操作符进行归约: 该方法使用 sycl::reduction 对象和自定义操作符在 parallel_for 中执行归约，用于计算最小值和索引等操作。

```
h.parallel_for(nd_range<1>{N, B}, reduction1, reduction2, ..., \
[=](nd_item<1> it, auto& temp1, auto& temp2, ...) {
// 内核代码
;
```

上面的代码行就是一个内核多个归约的方法的具体实现。这些方法，可以有效地利用 SYCL 的并行计算能力，实现高效的归约操作。

4.1.2 本节练习

本节的代码练习其实是内核归约，我也比较有兴趣去尝试，作为进阶部分的内容，也有必要去挑战一下。因此这里我重点介绍一下练习的实现！

```
    // 设置默认选择器的队列，02 节的老生常谈
    // 使用 USM 初始化数据数组，隐式 USM 写入最小值和最大值，前面 03 节也提过了，不赘述
    // 下面是归约的重点实现：
    // 第一步：创建用于计算最小值和最大值的归约对象
    auto min_reduction = reduction(min, sycl::ext::oneapi::minimum<>());
    auto max_reduction = reduction(max, sycl::ext::oneapi::maximum<>());
    // 归约内核获取最小值和最大值
    q.submit([&](handler& h) {
```

```
    // 第二步：添加带有最小值和最大值归约对象的 parallel_for
    h.parallel_for(nd_range<1>{N, B}, min_reduction, max_reduction, \
    [=](nd_item<1> item, auto& min, auto& max) {
      int idx = item.get_global_id(0);
      min.combine(data[idx]);
      max.combine(data[idx]);
    });
  }).wait();
  // 第三步：从最小值和最大值计算中值范围
  int mid_range = (*min + *max) / 2;
  std::cout << "Mid-Range = " << mid_range << "\n";
  //释放内存
```

4.1.3 本章小结

对于归约模块的学习，结合上文代码练习的实践，我有以下收获：

- 理解如何使用并行内核执行归约操作。尤其是在代码练习中，我使用 parallel_for 并行执行内核操作，通过分配工作组和全局范围来实现并行计算,nd_range<1>N, B 用于定义并行范围。
- 使用 reduction 对象简化并行内核中的归约操作。我在代码中通过 sycl::reduction 对象来简化归约操作，使用了 min_reduction 和 max_reduction 对象，分别计算最小值和最大值。
- 在单个内核中使用多个归约对象。代码练习展示了如何在单个内核中同时使用多个归约对象。通过在 parallel_for 中传递多个 reduction 对象，我们可以在一次内核调用中计算多个归约结果。

4.2 自主选题尝试

4.2.1 oneAPI 程序设计

然后我尝试进行了本选题下的 GPU 程序设计，在云平台上以 NoteBook 的形式进行编写编译。SYCL 程序的设计思想如下（普通串行算法见仓库内的先前实验，这里不再赘述）：

- 数据并行处理：程序利用 SYCL 框架的并行处理能力，将查询的处理任务分配到多个计算单元上并行执行。这种设计大大提高了程序的执行效率，特别是面对大量数据和复杂计算时。
- 任务分解与独立执行：程序将查询处理任务分解成独立的子任务，每个查询的处理都是独立的。这种设计思想使得每个子任务可以在不同的计算单元上并行执行，充分利用计算资源，避免了任务之间的相互依赖，提高了计算效率。
- 缓冲区和访问器的使用：程序使用 SYCL 的 buffer 和 accessor 机制管理数据。这种设计保证了数据在主机和设备之间的高效传递和访问，并且能够在不同计算单元之间共享数据。此外，buffer 的生命周期管理和 accessor 的访问控制机制也提高了程序的健壮性和安全性。

下面我们看一下本问题的关键 SYCL 部分代码：

```
    //在处理逻辑部分，首先将 queries 向量中的字符串转换为 C 风格的字符串数组。
    //然后创建 SYCL 的 buffer，分别用于存储 arrays、queries 和 results。
```

```
    results.resize(queries.size());
    queue myQueue(default_selector_v);
    vector<const char*> query_cstrs(queries.size());
    for (size_t i = 0; i < queries.size(); ++i) {
        query_cstrs[i] = queries[i].c_str();
    }
    buffer arrays_buffer(arrays.data(), range<1>(arrays.size()));
    buffer queries_buffer(query_cstrs.data(), range<1>(query_cstrs.size()));
    buffer results_buffer(results.data(), range<1>(results.size()));
    //通过 myQueue.submit 提交并行任务。任务处理函数中创建了三个 accessor,
    //分别用于访问 arrays_buffer、queries_buffer 和 results_buffer。
    //然后使用 parallel_for 函数并行处理每个查询。
    myQueue.submit([&](handler &h) {
        accessor arrays_accessor(arrays_buffer, h, read_only);
        accessor queries_accessor(queries_buffer, h, read_only);
        accessor results_accessor(results_buffer, h, write_only, no_init);
        h.parallel_for(range<1>(queries.size()), [=](id<1> i) {
            set<uint32_t> res;
            process_query(queries_accessor[i], res);
            results_accessor[i] = res;
        });
    }).wait();

```

4.2.2 实验结果

下面为节省篇幅，我们拿前五条查询和总体折线图简单看一下实验结果和总体性能，其他数据可见仓库！这里的 GPU 数量由设备选择器指定，这里采取的是 default_selector_v, 这通常是一个设备，我们可以看到 GPU 的高速运算能力，多设备的拟在期末研究中探索。

查询 ID	列表平均长度	串行平凡	oneAPI
1	20089	0.0008565	0.0003457
2	25584	0.0041958	0.0010457
3	30000	0.0038296	0.0010737
4	30000	0.0056320	0.0014437
5	22645	0.0019556	0.0005965
Total		6.031420	**3.521860**

图 4.4: oneAPI 优化数据图表

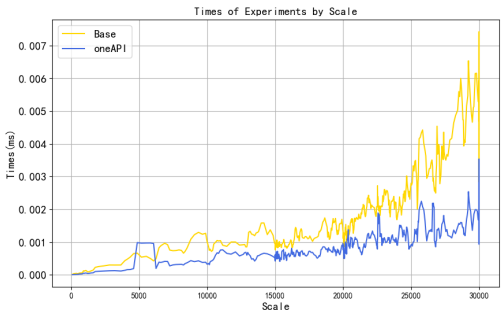

图 4.5: oneAPI 优化性能折线图

根据图表4.4和图4.5, 我可以得出以下结论和分析:

- 不难发现，oneAPI 下的实验性能普遍高于我们的串行算法，这可能是因为其下的 SYCL 并行化

框架通过优化数据传输、同步和通信、硬件利用率，再加上 **Intel(R) Data Center GPU Max 1100** 的高速运算能力，提高了程序的性能，达到更好的加速效果。这不仅体现在响应时间上，还体现在 Total 加速比上。

- 但是我们容易发现，oneAPI 下的程序性能响应时间的加速比显著高于 Total 加速比。对此值得我思考分析：响应时间对应单节点下的，而 Total 会涉及主机和设备间的通信，这就解释了为什么 Total 比会慢。同时我总结了几点 SYCL 内的一些功能的开销：
  - 数据传输开销：在主机和设备之间的数据传输是并行计算中常见的瓶颈。虽然计算部分加速了，但数据传输可能会占用较多时间。
  - 缓冲区传输：每次将数据从主机传输到设备（如 GPU）或从设备传回主机，都会引入额外的时间开销。尽量减少数据传输的次数和数据量可以提高整体性能。
  - 数据布局和对齐：不合理的数据布局和对齐会增加传输和访问开销，优化数据结构可以减少这种开销。比如位存储优化，这拟在期末研究中进一步探索。
- 最后我们来看一下随规模这两个算法的时间比也将不断增大，这意味着 SYCL 程序有着更好的规模适应性，在大小规模的性能差异不会很大，这可能是因为它能更好的利用硬件资源和缓存管理。而平凡串行算法受规模影响较严重！

5 总结

本次实验是倒排索引问题的 GPU 优化实验，也作为期末研究的最后一个子实验，主要有以下收获：

- 了解 SYCL 作为 oneAPI 的一部分，其在数据并行和异构编程中的作用，并掌握 SYCL 的基本知识，语法和编程模型，包括内核编程、内存管理和执行模型。
- 掌握 SYCL 基本类的概念和使用，选择各种用于卸载内核工作负载的设备的方法，以及增强使用缓冲区、访问器、命令组处理程序和内核编写 SYCL 程序的能力。
- 理解使用 USM 进行隐式和显式内存移动的方法，并掌握解决内核任务之间的数据依赖问题的方法。
- 探索并理解了如何使用并行内核执行归约操作，如何使用 reduction 对象简化并行内核中的归约操作，以及如何在单个内核中使用多个归约对象。
- 进行了详细的编程实践！这包括每节课程的课堂练习总共四次，以及自主选题的 oneAPI 编程实践，并针对实验结果做出了合理的解释和分析，这增强了我的 SYCL 编程能力和自主分析能力！

总之，这次实验帮我打好了期末研究的基础。让我掌握了 oneAPI 云平台的编程细节，理解了 SYCL 程序的结构和语法，增强了我的 GPU 并行编程能力。