南開大學
Nankai University

计算机学院

GPU 编程

姓名：钟坤原

学号：2212468

专业：计算机科学与技术

2024 年 6 月 16 日

目录

1 oneAPI 平台学习

本次实验是 GPU 并行编程实验。学习了解了 OneAPI 的基本知识，并进行了相关实验，在学习的过程中同时进行了详细的编程工作。

2 GPU 计算概述

GPU（Graphics Processing Unit，图形处理单元）最初被设计用于加速图形渲染任务，但其架构特别适合于执行更广泛的并行任务。因此，GPU 已经发展成为一种强大的通用计算设备，能够处理大量并行操作的数据密集型任务。

与传统的 CPU 相比，GPU 拥有更多的核心和更高的线程并行处理能力，使其在处理复杂的算法和大规模数据时能够显著提高性能。GPU 的计算能力特别适合执行可以分解为小部分并行处理的任务，如视频处理、科学模拟和深度学习等。

GPU 计算特点

- **高度并行化：** GPU 可以同时启动成千上万的线程，非常适合进行大规模并行计算。
- **数据吞吐量：**高带宽的内存接口使得 GPU 在处理大数据集时能够提供更好的性能。
- **成本效益：**相较于传统的多 CPU 系统，GPU 提供了一种成本效率更高的方式来提升计算性能。

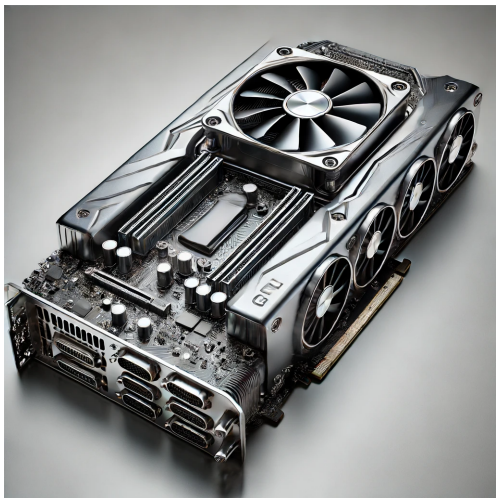

图 2.1: GPU 图片

3 学习目的和实验环境

3.1 学习目的

- 编写编译 SYCL 上的 C/C++ 程序
- 使用执行配置控制并行线程层次结构
- 重构串行循环以在 GPU 上并行执行迭代

3.2 实验环境

本次实验在 IntelDevCloud 实验训练平台下进行。

4 SYCL 课程基础学习概述

本课程旨在为学习者提供 SYCL 的基本知识和技能，SYCL 是一种基于 C++ 的高性能计算语言，用于开发异构计算系统。课程覆盖了从基础概念到高级编程技巧的广泛内容，使学习者能够在多种硬件平台上有效地开发并行和分布式计算程序。

通过本课程，学习者将能够：

- 理解 SYCL 的编程模型和架构。
- 掌握如何在 SYCL 中定义数据并行任务。
- 学习如何利用 SYCL 管理设备内存和执行。
- 熟悉 SYCL 在多种硬件上的应用，包括 CPU、GPU 和 FPGA。

本课程深入探讨如何在 SYCL 中构建数据并行程序，包括核函数的设计和调度任务。学生能够详细说明 SYCL 的内存模型，包括缓冲区和访问器的概念，以及如何优化数据传输和存取，并且可以解释如何在 SYCL 程序中选择和管理计算设备，以及如何创建和使用队列进行任务调度。

5 oneAPI 简介

5.1 oneAPI 编程简述

oneAPI 提供了一个跨多个硬件目标的统一开发工具组合，支持 CPU、GPU 和 FPGA 等硬件。该编程模型基于行业标准和开放规范，与现有的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。在数据中心，专用工作负载的增长导致开发人员需要维护多套代码库并学习多种工具，这增加了开发和维护的复杂性。

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。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。通过 oneAPI 的统一编程模型，开发人员能够更有效地利用不同硬件的计算资源，同时降低开发和维护的复杂性和成本。

5.2

本节介绍了如何使用 SYCL 进行简单的数据并行计算，以及如何在多种硬件架构上实现高效代码的方法。SYCL 是一种基于 C++ 的高级编程模型，专为异构计算设计。通过利用现代 C++ 特性，SYCL 允许开发者编写在不同硬件平台上都能高效运行的代码，包括 CPU、GPU 和 FPGA。以下是一个简单的 SYCL 程序示例，演示了如何使用队列和内核来进行基本的向量加法运算。

```
#include <sycl/sycl.hpp> // 引入 SYCL 头文件
using namespace sycl; // 使用 sycl 命名空间

static const int N = 20; // 定义常量 N 为数组大小
```

```
int main() {
    queue q; // 创建队列，关联到默认设备
    std::cout << "Running on " << q.get_device().get_info<info::device::name>() << "\n";

    // 使用统一共享内存为数组分配内存
    int *data = malloc_shared<int>(N, q);

    // 初始化数组
    for (int i = 0; i < N; i++) data[i] = i;

    // 提交并行任务：每个元素乘以 2
    q.parallel_for(range<1>(N), [=] (id<1> i) {
        data[i] *= 2;
    }).wait(); // 等待所有并行任务完成

    // 输出结果
    for(int i = 0; i < N; i++)
        std::cout << data[i] << " ";
    std::cout << "\n";

    free(data, q); // 释放内存
    return 0;
}
```

5.3 小节总结

在本节中，通过一个简单的示例介绍了如何利用 SYCL 实现数据并行计算。示例展示了如何设置数据并行任务、管理内存以及如何在设备上执行计算任务。这个简单的代码示例主要是为了帮助理解如何在 SYCL 环境中设置并行计算任务，并展示了基本的内存管理和队列操作。

- **oneAPI 编程模型：**提供跨多种硬件（如 CPU、GPU 和 FPGA）的统一编程解决方案，减少开发和维护的复杂性。
- **SYCL 作为 oneAPI 的一部分：**利用 C++ 为基础，实现了数据并行和异构编程，使得代码能够在不同的计算平台上高效运行。

通过本节的学习，我们可以观察到 SYCL 提供的多个重要概念，包括队列、缓冲区、访问器以及并行 for 循环的使用。这些工具和技术是构建复杂的异构并行程序的基础，对于实现高性能计算应用至关重要。这次的代码初体验让我了解了如何使用 SYCL 框架进行基本的数据并行计算，并且理解了将计算任务卸载到支持的硬件设备上的过程。

6 SYCL 编程结构

6.1 SYCL 类

SYCL 框架通过一系列的 C++ 类、模板和库，为开发者提供了强大的工具来编写在不同计算设备上运行的程序。这些类包括设备选择器、队列、缓冲区和访问器等，它们共同协作，使得数据并行和异构编程变得更加直观和高效。

- **Device 类：** 用于查询和选择计算设备。开发者可以通过 Device 类获取关于设备的详细信息，以决定哪些计算任务最适合在该设备上执行。
- **Device Selector 类：** 允许在运行时根据特定标准自动选择最合适的设备。这是实现最优性能的关键，因为它确保任务在最适合的硬件上执行。
- **Queue 类：** 管理和调度设备上的执行命令。每个队列与特定的设备相关联，并负责向该设备提交执行任务。
- **Buffer 类和 Accessor 类：** 管理数据存储和访问。Buffer 类在主机和设备之间封装数据存储，而 Accessor 类则提供了访问这些数据的方式，确保在数据使用时的正确性和效率。

6.2 并行内核

并行内核是 SYCL 中用于表达并行操作的基本构建块。内核代表在设备上执行的并行计算任务，可以同时处理多个数据元素。

SYCL 中，使用 parallel_for 来调度并行内核，它允许开发者定义一个执行范围，SYCL 运行时负责将这些任务分配到设备上的不同计算单元上执行。这种模式极大地简化了复杂并行任务的实现，并能有效利用硬件资源来提升程序性能。

这部分介绍了如何使用 SYCL 的 parallel_for 来实现向量加法的并行计算。通过定义一个内核，该内核在每个元素上执行加法操作，我们演示了如何利用并行内核加速简单的数据处理任务。

```
q.parallel_for(range<1>(dataSize), [=](id<1> i) {
    result[i] = input1[i] + input2[i];
});
```

这代码展示了如何定义一个简单的并行内核，它对两个输入向量的对应元素进行加法操作，并将结果存储在结果向量中。

6.3 本节练习

本练习的目的是展示如何使用 SYCL 进行数组的规约操作，即计算数组中所有元素的总和。以下是使用 SYCL 实现数组规约的核心代码：

```
const int dataSize = 1024;
std::vector<int> data(dataSize, 20);  // 创建大小为N的向量，所有元素初始化为20

// 创建SYCL缓冲区
buffer<int> dataBuffer(data.data(), range<1>(dataSize));
```

```
// 使用规约来计算数组元素的总和
int totalSum = 0;
{
    // 使用accessor来访问缓冲区数据
    buffer<int> sumBuffer(&totalSum, range<1>(1));

    q.submit([&](handler& h) {
        auto dataAcc = dataBuffer.get_access<access::mode::read>(h);
        auto sumAcc = sumBuffer.get_access<access::mode::read_write>(h);
        h.parallel_for(range<1>(dataSize), reduction(sumAcc, std::plus<int>()),
                       [=](id<1> idx, auto& sum) {
            sum += dataAcc[idx];
        });
    }).wait();  // 等待队列完成所有任务
}
```

6.4 小节总结

在本小节中，我深入探讨了 SYCL 的程序结构，包括其核心类如设备 (Device)、队列 (Queue)、缓冲区 (Buffer) 和访问器 (Accessor) 等。通过这些学习，我不仅了解了这些类的功能和使用方法，还掌握了它们在实际并行编程中的应用。

- **设备选择和管理：** 学习如何通过 Device 和 Device Selector 类选择和管理计算设备。
- **任务调度和执行：** 掌握如何使用 Queue 类来管理和调度在设备上执行的任务。
- **数据管理：** 了解如何通过 Buffer 和 Accessor 类高效地管理和访问数据。
- **并行内核的实现：** 探索了如何使用 parallel_for 来定义并执行并行内核，实现数据并行处理。

7 SYCL 统一内存共享

7.1 USM 简介

统一共享内存（USM）是 SYCL 提供的一种内存管理机制，允许开发者以熟悉的 C++ 指针的形式来管理设备和主机的内存。USM 简化了数据的迁移和访问，使开发者可以更加专注于算法的实现，而不是底层的数据传输细节。

主要优势有：

- **简化的内存模型：** 开发者可以使用传统的 C++ 指针操作来访问内存，无需担心设备与主机间的显式数据传输。
- **高效的数据访问：** USM 允许内核直接访问主机内存或设备内存，降低了内存访问的延迟，并提高了数据处理速度。
- **灵活的内存控制：** USM 提供了设备内存、共享内存和主机内存三种模式，开发者可以根据需要选择最适合的内存类型。

7.2 本节练习

在本练习中，我通过 SYCL 编程模型使用 USM 分配内存，对两个大数组进行初始化，并计算它们的元素和。以下是使用 SYCL 和 USM 完成的练习代码：

```
#include <CL/sycl.hpp>
#include <iostream>
#include <vector>

using namespace sycl;

int main() {
    const int N = 1024;  // 数组大小
    queue q;  // 创建一个队列

    // STEP 1: 分配USM设备内存
    int* arrayA = malloc_device<int>(N, q);
    int* arrayB = malloc_device<int>(N, q);
    int* arrayC = malloc_device<int>(N, q);

    // STEP 2: 初始化数组A和B
    q.parallel_for(range<1>(N), [=](id<1> i) {
        arrayA[i] = 1;  // 将所有元素设为1
        arrayB[i] = 2;  // 将所有元素设为2
    }).wait();  // 确保初始化完成

    // STEP 3: 计算数组A和B的和，存储到数组C
    q.parallel_for(range<1>(N), [=](id<1> i) {
        arrayC[i] = arrayA[i] + arrayB[i];
    }).wait();  // 确保计算完成

    // STEP 4: 读取并打印结果
    auto result = malloc_shared<int>(N, q);
    q.memcpy(result, arrayC, sizeof(int)*N).wait();  // 将结果从设备内存复制到共享内存
    for (int i = 0; i < N; i++) {
        std::cout << "C[" << i << "] = " << result[i] << std::endl;
    }

    // 释放内存
    free(arrayA, q);
    free(arrayB, q);
    free(arrayC, q);
    free(result, q);
```

```
    return 0;
}
```

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。

7.3 小节总结

本小节探讨了统一共享内存(USM)的高级应用,特别是如何在更复杂的计算场景中有效利用 USM 来优化性能和简化数据管理。USM 不仅简化了内存管理，还为复杂的数据交互和异步任务执行提供了高效的解决方案。通过使用 USM，开发者可以减少对显式内存复制命令的依赖，实现更加流畅和高效的数据处理流程。总结为以下几点：

- **内存异步传输：** 利用 USM 的异步数据传输功能，可以在不阻塞主线程的情况下执行内存复制，从而提升程序的整体运行效率。
- **内存访问模式优化：** 根据具体的应用需求调整数据访问模式，例如选择最适合的内存类型（设备内存、共享内存或主机内存）以最小化访问延迟。
- **并行数据处理与内存操作：** 在执行数据处理的同时进行内存操作，利用设备的并行计算能力，减少总体执行时间。

8 GPU 进阶

8.1 基于 CUDA 加速 KMeans

本节讨论了 KMeans 算法在 CUDA 平台上的加速实现。KMeans 是一种广泛应用于聚类分析的算法，通过并行处理可以显著提高其执行速度。

8.2 算法概述

KMeans 算法通过迭代方式优化聚类中心，直至满足终止条件。每次迭代包括两个主要步骤：

- **聚类赋值步骤**：为每个数据点分配最近的聚类中心。
- **中心更新步骤**：更新每个聚类的中心点，通常为所属点的均值。

8.3 CUDA 加速尝试

在 CUDA 加速的实现中，利用 GPU 的并行计算能力同时处理多个数据点的聚类赋值和中心更新，从而减少算法运行时间。。其中的核心函数包含以下两个，一个是寻找最近的质心 assign_clusters()，一个是计算和质心之间的距离 update_centroids() 以下是 KMeans 算法的 CUDA 加速实现的核心代码部分：

```
__global__ void assign_clusters(float *data, int *clusters, float *centroids,
                                int n_points, int n_clusters, int dim) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if (index < n_points) {
        int best_cluster = 0;
        float min_distance = FLT_MAX;
        for (int cluster = 0; cluster < n_clusters; cluster++) {
            float distance = 0;
            for (int d = 0; d < dim; d++) {
                float diff = data[index * dim + d] - centroids[cluster * dim + d];
                distance += diff * diff;
            }
            if (distance < min_distance) {
                min_distance = distance;
                best_cluster = cluster;
            }
        }
        clusters[index] = best_cluster;
    }
}

__global__ void update_centroids(float *data, int *clusters, float *centroids,
                                 int n_points, int n_clusters, int dim) {
    __shared__ float local_centroids[1024]; // Adjust size based on dimensions and number of clu
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n_points) {
        int cluster_id = clusters[index];
        for (int d = 0; d < dim; d++) {
            atomicAdd(&local_centroids[cluster_id * dim + d], data[index * dim + d]);
        }
        __syncthreads();
        if (threadIdx.x == 0) { // Only one thread per block does the final update
            for (int d = 0; d < dim; d++) {
                atomicAdd(&centroids[cluster_id * dim + d], local_centroids[cluster_id * dim + d
            }
        }
    }
}
```

实验表明，将 KMeans 算法迁移到 CUDA 上可以显著减少算法的运行时间。特别是在处理大数据集时，GPU 的并行处理能力能够有效提高数据处理效率。

9 总结

通过这些 SYCL 实验，我们不仅加深了对异构编程的理解，还掌握了一系列优化并行程序性能的技术。实验展示了 SYCL 作为一个强大的异构编程模型，能够有效地提升程序在多种计算平台上的执

行效率。学习成果包括：

- 对 SYCL 程序结构和执行模型有了全面的理解，能够根据不同硬件特性编写和优化代码。
- 成功应用了 SYCL 进行异构编程，有效利用了 GPU 等加速器的强大计算资源。
- 掌握了利用统一共享内存（USM）和数据并行技术优化程序性能的方法。

代码与图片文件已上传 GitHub **Github 地址：** Github