



南開大學

Nankai University

计算机学院  
并行程序设计实验报告

GPU 实验

姓名：徐海滢

学号：2212180

专业：计算机科学与技术

2024 年 6 月 15 日

# 目录

<b>1</b>	<b>基本要求</b>	<b>2</b>
1.1	01_oneAPI_intro . . . . .	2
1.1.1	实验目的 . . . . .	2
1.1.2	oneAPI 编程模型概述 . . . . .	2
1.1.3	SYCL . . . . .	2
1.1.4	simple.cpp 的简单练习 . . . . .	3
1.1.5	oneAPI 编程模型 . . . . .	4
1.1.6	编译并运行 SYCL 程序 . . . . .	5
1.2	02_SYCL_Program_Structure . . . . .	6
1.2.1	实验目的 . . . . .	6
1.2.2	C++ SYCL 代码示例 . . . . .	6
1.2.3	SYCL 类: 设备类 . . . . .	6
1.2.4	队列与内核 . . . . .	7
1.2.5	示例: 使用 USM 和缓冲区实现向量加法 . . . . .	8
1.2.6	缓冲区内存模型 . . . . .	8
1.2.7	示例: 与主机访问器的同步 . . . . .	9
1.2.8	示例: 自定义设备选择器 . . . . .	10
1.2.9	实验: 向量加法 . . . . .	10
1.3	03_SYCL_Unified_Shared_Memory . . . . .	12
1.3.1	实验目的 . . . . .	12
1.3.2	统一共享内存 (USM) . . . . .	12
1.3.3	示例: USM 的隐式数据移动 . . . . .	12
1.3.4	USM 中的数据依赖关系 . . . . .	13
1.3.5	示例: USM 和数据依赖关系 1 . . . . .	14
1.3.6	示例: USM 和数据依赖关系 2 . . . . .	15
1.3.7	实验: 统一共享内存 . . . . .	16
<b>2</b>	<b>进阶要求</b>	<b>18</b>
2.1	04_SYCL_Sub_Groups . . . . .	18
2.1.1	实验目标 . . . . .	18
2.1.2	子组 (Subgroup) 的定义 . . . . .	18
2.1.3	示例: 打印子组信息 . . . . .	18
2.1.4	实验: 利用子组完成编码练习 . . . . .	19

# 1 基本要求

## 1.1 01\_oneAPI\_intro

### 1.1.1 实验目的

1. 解释 oneAPI 编程模型如何解决在异构世界中编程的挑战
2. 使用 oneAPI 项目来支持的工作流程
3. 理解 SYCL 语言和编程模型
4. 熟悉在整个课程中使用 Jupyter notebooks

### 1.1.2 oneAPI 编程模型概述

oneAPI 编程模型提供了一个全面且统一的开发工具组合，可以跨越各种硬件目标使用，包括覆盖多个工作负载领域的性能库。这些库包含为每个目标架构定制编码的函数，使得相同的函数调用在支持的架构上实现优化性能。DPC++ 基于行业标准和开放规范，以鼓励生态系统的协作与创新。



图 1.1

### 1.1.3 SYCL

SYCL 是一种高级编程模型，允许开发者使用现代 C++ 特性来编写并行代码，同时保留了对底层 OpenCL 的访问。

**SYCL 的特点:** SYCL 允许在单个源文件中编写主机代码和设备代码，这使得代码管理更加方便，并且易于维护和调试。SYCL 支持现代 C++ 的所有特性，包括模板、类和函数重载，这使得代码更加清晰和可重用。此外，SYCL 的设计使得同一代码可以在多种硬件平台上运行，包括 CPU、GPU、FPGA 等，这简化了多平台开发的复杂性。通过 SYCL，开发者可以轻松地编写并行计算程序，充分利用不同硬件平台的性能。

SYCL 编程模型基于以下几个关键概念：



1. **队列 (Queue)**: SYCL 使用队列来管理命令的执行顺序。每个队列关联一个计算设备, 命令通过队列提交到设备上执行。
2. **缓冲区 (Buffer)**: 缓冲区用于在主机和设备之间传递数据。SYCL 的缓冲区管理机制确保数据的一致性和高效传输。
3. **内核 (Kernel)**: 内核是实际执行的并行代码。开发者可以使用 lambda 表达式或函数对象来定义内核函数。
4. **访问器 (Accessor)**: 访问器提供了一种安全的方式来访问缓冲区中的数据。访问器在内核执行期间管理数据的读写权限。

数据并行 C++ (DPC++) 是 oneAPI 对 SYCL 编译器的实现。它利用现代 C++ 的生产力优势和熟悉的构造, 并结合了 SYCL\* 标准以实现数据并行和异构编程。SYCL 是一种单一源语言, 其中主机代码和异构加速器内核可以混合在同一个源文件中。SYCL 程序在主机上调用, 并将计算卸载到加速器。程序员使用熟悉的 C++ 和库构造, 并添加一些功能, 例如用于工作定位的队列、用于数据管理的缓冲区和用于并行的 `parallel_for`, 以指示应卸载计算和数据的哪些部分。

#### 1.1.4 simple.cpp 的简单练习

实验的内容是通过一小段简单的代码向开发人员介绍 SYCL。此外, 它还向开发人员介绍了 Jupyter 笔记本环境, 用于编辑和保存代码; 以及运行和将程序提交到英特尔® DevCloud。

根据教程, 检查下面的代码单元, 然后单击运行 将代码保存到文件。运行 代码片段下方的“构建和运行”部分中的单元格, 以编译并执行保存的文件中的代码。代码和运行结果的截图如下:

1. 检查下面的代码单元，然后单击运行  将代码保存到文件
2. 运行  代码片段下方的“构建和运行”部分中的单元格，以编译并执行保存的文件中的代码

```
%%writefile lab/simple.cpp
//=====
// Copyright © Intel Corporation
//
// SPDX-License-Identifier: MIT
// =====
#include <sycl/sycl.hpp>
using namespace sycl;
static const int N = 16;
int main(){
    /// define queue which has default device associated for offload
    queue q;
    std::cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";

    /// Unified Shared Memory Allocation enables data access on host and device
    int *data = malloc_shared<int>(N, q);

    /// Initialization
    for(int i=0; i<N; i++) data[i] = i;

    /// Offload parallel computation to device
    q.parallel_for(range<1>(N), [=] (id<1> i){
        data[i] *= 2;
    }).wait();

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

    free(data, q);
    return 0;
}
```

Overwriting lab/simple.cpp

图 1.2: simple.cpp

```
! chmod 755 q; chmod 755 run_simple.sh;if [ -x "$(command -v qsub)" ]; then ./q run_simple.sh; else ./run_simple.sh; fi

## u4f00f5e464281c4ab12edf2f369d262 is compiling SYCL_Essentials Module1 -- oneAPI Intro sample - 1 of 1 simple.cpp
Device: Intel(R) Data Center GPU Max 1100
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
```

图 1.3: 编译结果

### 1.1.5 oneAPI 编程模型

(1) 平台模型：oneAPI 的平台模型基于 SYCL\* 平台模型。它指定了一个或多个设备由主机控制。主机通常是基于 CPU 的计算机系统，负责执行程序的主要部分，特别是应用程序范围和命令组范围的任务。主机的主要职责是协调和控制设备上执行的计算工作。设备通常是一种加速器，包含丰富的计算资源，能够高效地执行特定操作。设备的效率通常高于系统中的 CPU。

(2) 执行模型：执行模型基于 SYCL\* 执行模型。它定义并指定代码（称为内核）如何在设备上执行并与控制主机交互。主机执行模型通过命令组协调主机和设备之间的执行和数据管理。命令组是内核调用和访问器等命令的分组，提交到队列中执行。访问器是内存模型的正式组成部分，它还传达执

行的排序要求。采用执行模型的程序声明并实例化队列。队列可以按照程序可控制的按序或无序策略执行。按序执行是英特尔扩展。设备执行模型指定了如何在加速器上完成计算。从小型一维数据到大型多维数据集的计算都分配在 ND 范围、工作组、子组（英特尔扩展）和工作项的层次结构中，这些层次结构都是在将工作提交到命令队列时指定的。值得注意的是，实际的内核代码代表针对一个工作项执行的工作。内核之外的代码控制执行的并行度；工作量和分配由 ND 范围和工作组的大小规范控制。

(3) 内存模型：oneAPI 的内存模型基于 SYCL\* 内存模型。它定义了主机和设备如何与内存交互。它协调主机和设备之间的内存分配和管理。内存模型是一种抽象，旨在概括并适应不同的主机和设备配置。在此模型中，内存驻留在主机或设备上，由主机或设备拥有，并通过声明内存对象来指定。内存对象有两种类型：缓冲区和图像。主机和设备之间这些内存对象的交互是通过访问器完成的，访问器会传达所需的访问位置（例如主机或设备）以及特定的访问模式（例如读取或写入）。考虑通过传统的 malloc 调用在主机上分配内存的情况。在主机上分配内存后，将创建一个缓冲区对象，这使得主机分配的内存可以传送到设备。缓冲区类会传达要传送到设备进行计算的该类型的项目类型和数量。在主机上创建缓冲区后，将通过访问器对象传达设备上允许的访问类型，该访问器对象指定对缓冲区的访问类型。

(4) 内核编程模型：oneAPI 的内核编程模型基于 SYCL\* 内核编程模型。它支持主机和设备之间的显式并行性。并行性是显式的，因为程序员决定在主机和设备上执行哪些代码；它不是自动的。内核代码在加速器上执行。采用 oneAPI 编程模型的程序支持单源，这意味着主机代码和设备代码可以位于同一个源文件中。但是，主机代码和设备代码所接受的源代码在语言一致性和语言特性方面存在差异。SYCL 规范详细定义了主机代码和设备代码所需的语言特性。

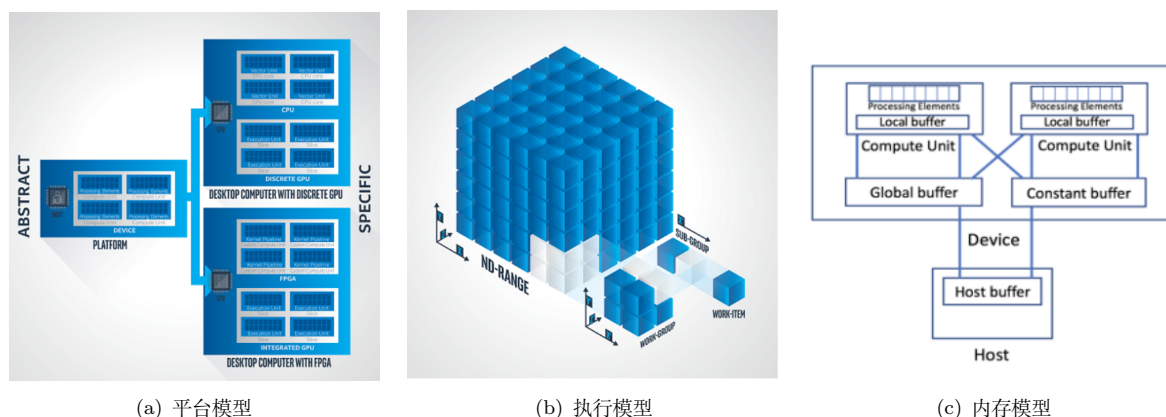


图 1.4: 模型的结构图

### 1.1.6 编译并运行 SYCL 程序

在前面的 simple.cpp 简单练习中，我们已经简单尝试过如何编译并运行 SYCL 程序。结合之前的实验思考，总结出编译并运行 SYCL 程序的步骤。

编译和运行 SYCL 程序的三个主要步骤是：

1. 初始化环境变量
2. 编译 SYCL 源代码
3. 运行应用程序

在 intel Devcloud 上可以通过命令 `./q run.sh` 运行脚本。此脚本将脚本提交 `run.sh` 到 DevCloud 上的 GPU 节点以供执行，等待作业完成并打印出输出/错误。上面的示例代码就是通过这种方式运行的。

## 1.2 02\_SYCL\_Program\_Structure

### 1.2.1 实验目的

1. 解释 SYCL 基本类
2. 学会使用设备选择来卸载内核工作负载
3. 学会决定何时使用基本并行内核和 ND 范围内核
4. 在 SYCL 程序中使用统一共享内存或缓冲区访问器内存模型
5. 通过实践实验室练习构建示例 SYCL 应用程序

### 1.2.2 C++ SYCL 代码示例

代码执行以下操作：选择 GPU 设备进行卸载，分配可在主机和 GPU 上访问的内存，在主机上初始化数据数组。将计算任务卸载到 GPU，在主机上打印输出。代码和详细注释如下：

---

```
1 #include <sycl/sycl.hpp>
2 static const int N = 16; // 定义常量 N，表示数组的大小
3 int main() {
4     // 创建 SYCL 队列，并选择 GPU 作为设备
5     sycl::queue q(sycl::gpu_device_selector_v);
6     // 在共享内存中分配一个大小为 N 的整数数组
7     int *data = sycl::malloc_shared<int>(N, q);
8     for (int i = 0; i < N; i++) data[i] = i;
9     // 使用并行计算在 GPU 上执行内核代码
10    q.parallel_for(N, [=](auto i) {
11        data[i] *= 2; // 内核代码：将数组中的每个元素乘以 2
12    }).wait(); // 等待所有任务完成
13    // 输出结果
14    for (int i = 0; i < N; i++) std::cout << data[i] << "\n";
15    // 释放分配的内存
16    sycl::free(data, q);
17    return 0;
18
```

---

### 1.2.3 SYCL 类：设备类

**设备：**SYCL 提供了一套抽象类来表示和管理计算设备。设备类 (`sycl::device`) 是这些抽象类中的一个，它表示一个计算设备，可以是 CPU、GPU 或其他加速器。设备类在 SYCL 中扮演着重要角色，负责提供关于设备的信息，并允许开发者选择和使用合适的设备来执行计算任务。设备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。函数 `get_info` 提供有关设备的信

息有：设备的名称、供应商和版本，本地和全局工作项 ID，内置类型的宽度、时钟频率、缓存宽度和大小、在线或离线。

**设备选择器实验：**这些类支持根据用户提供的启发式方法在运行时选择特定设备来执行内核。给出的代码示例展示了标准设备选择器的使用。根据实验指引保存编译运行代码，成功输出了系统中所选择的 GPU 设备名称。截图如下：

1. 检查下面的代码单元并单击运行  将代码保存到文件
2. 接下来运行  代码下方的Build and Run部分中的单元格以编译并执行代码。

```
%writefile lab/gpu_sample.cpp
//=====
// Copyright © Intel Corporation
//
// SPDX-License-Identifier: MIT
// =====
#include <sycl/sycl.hpp>

using namespace sycl;

int main() {
    /// Create a device queue with device selector

    queue q(gpu_selector_v);
    //queue q(cpu_selector_v);
    //queue q(accelerator_selector_v);
    //queue q(default_selector_v);
    //queue q;

    /// Print the device name
    std::cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";

    return 0;
}

Overwriting lab/gpu_sample.cpp
```

图 1.5: cpp

```
! chmod 755 q; chmod 755 run_gpu.sh; if [ -x "${command -v qsub}" ]; then ./q run_gpu.sh; else ./run_gpu.sh; fi

## u4f00f5e464281c4ab12edf2f369d262 is compiling SYCL_Essentials Module2 -- SYCL Program Structure sample - 1 of 7 gpu_sample.cpp
Device: Intel(R) Data Center GPU Max 1100
```

图 1.6: 编译运行结果

#### 1.2.4 队列与内核

**队列：**在 SYCL 中，队列（queue）是一个用于管理和控制设备上命令执行的机制。队列提供了一种将任务（如内核执行、内存拷贝等）提交到设备的方式。通过使用队列，开发者可以确保任务按照特定的顺序执行，并能够同步或异步地管理任务。队列提交要由 SYCL 运行时执行的命令组。队列是一种将工作提交给设备的机制。一个队列映射到一个设备，多个队列可以映射到同一个设备。

```
1 q.submit([&](handler& h) {
2     ///COMMAND GROUP CODE
3 ;
```

**内核：**内核（kernel）是实际在计算设备（如 GPU、CPU）上执行的并行代码片段。内核函数包含并行计算的具体实现，通常在设备上以并行的方式运行，以充分利用设备的计算能力。内核类封装了在实例化命令组时在设备上执行代码的方法和数据。内核对象不是由用户明确构造的，而是在调用内核调度函数（例如 parallel\_for）时构造的



```

1 q.submit([&](handler& h) {
2     // 使用 parallel_for 提交并行任务, 范围为 range<1>(N)
3     h.parallel_for(range<1>(N), [=](id<1> i) {
4         // 内核代码: 对每个工作项, 将 A[i] 设置为 B[i] 与 C[i] 之和
5         A[i] = B[i] + C[i];
6     });

```

### 1.2.5 示例：使用 USM 和缓冲区实现向量加法

练习内容：编译并运行 `vector_add_usm_buffers.cpp`，使用 USM 和缓冲区实现向量加法下面的 SYCL 代码展示了使用 USM 和缓冲区内存模型实现的向量加法计算。

```
// initialize c = 0 and offload computation using USM, print output
for (int i=0; i<N; i++) c[i] = 0;
std::cout << "Vector Add Output (Buffers): \n";
kernel_buffers(a, b, c, N);
for (int i=0; i<N; i++)std::cout << c[i] << " ";std::cout << "\n";
}
```

Overwriting lab/vector add usm buffers.cpp

图 1.7

### 1.6.1. 构建并运行

选择下面的单元格并单击运行  来编译并执行代码。

[illegible]

图 1.8: 编译运行结果

### 1.2.6 缓冲区内存模型

缓冲区在设备和主机之间封装 SYCL 应用程序中的数据。访问器是访问缓冲区数据的机制。缓冲内存模型 (Buffer Memory Model) 是一个重要的概念, 用于在主机 (Host) 和设备 (Device) 之间管理数据的传输和存储。SYCL 缓冲区 (Buffer) 提供了一种高效、安全和易于使用的方式来处理异构计算环境中的数据。缓冲区管理内存分配, 可以根据数据的大小自动分配所需的内存空间, 并在任务完成后自动释放内存。

SYCL 程序是标准 C++。该程序在主机上调用，并将计算卸载到加速器。程序员使用 SYCL 的队列、缓冲区、设备和内核抽象来指示应卸载计算和数据的哪些部分。

作为 SYCL 程序的第一步，我们创建一个队列。我们通过将任务提交到队列来将计算卸载到设备上。程序员可以通过选择器选择 CPU、GPU、FPGA 和其他设备。此程序在此处使用默认的 q，这意味着 SYCL 运行时使用默认选择器选择运行时可用的最强大的设备。我们将在接下来的模块中讨论设备、设备选择器以及缓冲区、访问器和内核的概念，但下面是一个简单的 SYCL 程序，供您开始了解上述概念。

设备和主机可以共享物理内存，也可以拥有不同的内存。当内存不同时，卸载计算需要在主机和设备之间复制数据。SYCL 不需要程序员管理数据副本。通过创建缓冲区和访问器，SYCL 确保主机和设备无需程序员付出任何努力即可获得数据。当需要实现最佳性能时，SYCL 还允许程序员明确控制数据移动。

### 1.2.7 示例：与主机访问器的同步

主机访问器是使用主机缓冲区访问目标的访问器。它是在命令组范围之外创建的，它允许访问的数据将在主机上可用。它们用于通过构造主机访问器对象将数据同步回主机。缓冲区销毁是将数据同步回主机的另一种方式。

缓冲区拥有存储在向量中的数据的所有权。创建主机访问器是一个阻塞调用，并且只有在修改任何队列中同一缓冲区的所有入队 SYCL 内核完成执行并且数据可通过此主机访问器供主机使用后会返回。下面的 SYCL 代码演示了与主机访问器的同步。代码和运行结果如下：

2. 接下来运行  代码下方的Build and Run部分中的单元格以编译并执行代码。

```
]: %%writefile lab/host_accessor_sample.cpp
//=====
// Copyright © Intel Corporation
//
// SPDX-License-Identifier: MIT
// =====

#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
    constexpr int N = 16;
    auto R = range<1>(N);
    std::vector<int> v(N, 10);
    queue q;
    // Buffer takes ownership of the data stored in vector.
    buffer buf(v);
    q.submit([&](handler& h) {
        accessor a(buf, h);
        h.parallel_for(R, [=](auto i) { a[i] -= 2; });
    });
    // Creating host accessor is a blocking call and will only return after all
    // enqueued SYCL kernels that modify the same buffer in any queue completes
    // execution and the data is available to the host via this host accessor.
    host_accessor b(buf, read_only);
    for (int i = 0; i < N; i++) std::cout << b[i] << " ";
    return 0;
}
```

Overwriting lab/host\_accessor\_sample.cpp

图 1.9

```
! chmod 755 q; chmod 755 run_host_accessor.sh;if [ -x "$(command -v qsub)" ]; then ./q run_host_accessor.sh; else ./run_host_accessor.sh; fi
## u4f00f5e464281c4ab12edf2f369d262 正在编译 SYCL_Essentials Module2 -- SYCL 程序结构示例 - 7 个中的 3 个 host_accessor_sample.cpp
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
```

图 1.10: 编译结果

缓冲区销毁的同步练习和该练习结果相似，故不再省略。

### 1.2.8 示例：自定义设备选择器

自定义设备选择器 (Custom Device Selector) 是在 SYCL 中使用特定逻辑来选择计算设备的一种方法。开发者可以通过编写自定义函数，根据设备的属性（如供应商名称、设备类型、设备名称等）来评估设备的优先级并选择最合适的设备。在网站上提供了基于不同属性进行选择的代码。

```
};
int main() {
    //pass in the name of the vendor for which the device you want to query
    std::string vendor_name = "Intel";
    //std::string vendor_name = "AMD";
    //std::string vendor_name = "Nvidia";
    my_device_selector selector(vendor_name);
    queue q(selector);
    std::cout << "Device: "
    << q.get_device().get_info<info::device::name>() << "\n";
    return 0;
}
```

Overwriting lab/custom\_device\_sample.cpp

图 1.11

```
! chmod 755 q; chmod 755 run_custom_device.sh;if [ -x "$(command -v qsub)" ]; then ./q run_custom_device.sh; else ./run_custom_device.sh; fi
## u4f00f5e464281c4ab12edf2f369d262 is compiling SYCL_Essentials Module2 -- SYCL Program Structure sample - 5 of 7 custom_device_sample.cpp
Device: Intel(R) Data Center GPU Max 1100
```

图 1.12: 编译运行结果

### 1.2.9 实验：向量加法

实验内容：使用 SYCL Buffer 和 Accessor 概念完成下面的编码练习：该代码 vector1 在主机上初始化了三个向量，内核代码将其加 vector11。，创建一个新的秒 vector2 并初始化为值 20。为上述第二个向量创建 sycl 缓冲区，在内核代码中，为第二个向量缓冲区创建第二个访问器，并将向量增量修改为向量加法，方法是将其添加 vector2 到 vector1

在基础框架上进行代码的编写，我编写的代码如下：

```
1  %%writefile lab/vector_add.cpp
2
3  #include <sycl/sycl.hpp>
4  #include <iostream>
5  #include <vector>
6
7  using namespace sycl;
8
9  int main() {
10  const int N = 256;
11
12  /// Initialize a vector and print values
13  std::vector<int> vector1(N, 10);
```



于  $10 + 20 = 30$ ，符合程序逻辑。

### 1.3 03\_SYCL\_Unified\_Shared\_Memory

#### 1.3.1 实验目的

1. 使用统一共享内存等 SYCL2020 的新功能来简化编程。
2. 了解使用 USM 移动内存的隐式和显式方式。
3. 以最优方式解决内核任务之间的数据依赖关系。

#### 1.3.2 统一共享内存 (USM)

统一共享内存 (USM) 是 SYCL 中基于指针的内存管理。USM 是一种基于指针的方法，使用 `malloc` 或 `new` 分配数据的 C 和 C++ 程序员应该很熟悉。在将现有 C/C++ 代码移植到 SYCL 时，USM 简化了程序员的开发工作。

USM 的开发者视图如下，显示了开发人员对没有 USM 和有 USM 的内存的看法。利用 USM，开发人员可以在主机和设备代码中引用相同的内存对象。

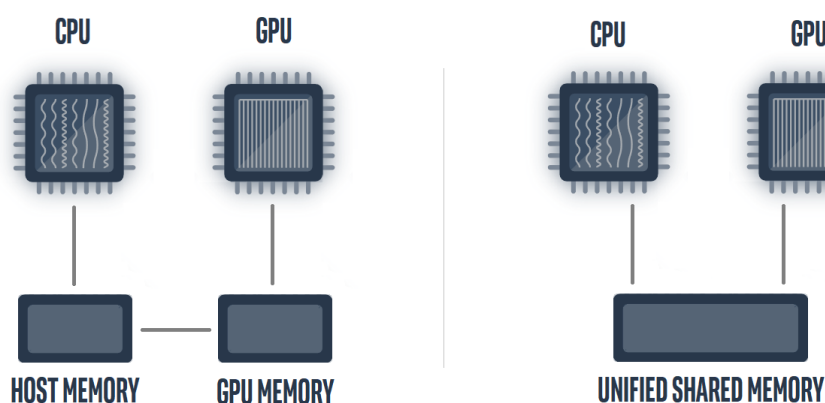


图 1.14: USM 的开发者视图

统一共享内存提供了用于管理内存的显式和隐式模型，具体的类型如下表格：

类型	函数调用	描述	可在主机上访问	可在设备上访问
设备	<code>malloc_device</code>	设备上的分配（显式）	否	是的
主持人	<code>malloc_host</code>	主机上的分配（隐式）	是的	是的
共享	<code>malloc_shared</code>	分配可以在主机和设备之间迁移（隐式）	是的	是的

表 1: 内存分配类型与访问权限

#### 1.3.3 示例：USM 的隐式数据移动

USM 可以使用共享分配的示例 `malloc_shared`，“q” 队列参数提供了有关可访问内存的设备的信息。并通过 `free` 进行释放。

---

```

1  int *data = malloc_shared<int>(N, q); //USM 的初始化
2
3  free(data, q); //释放 USM

```

---

练习使用 USM 实现 malloc\_shared, 其中数据移动在主机和设备之间隐式发生。有助于以最少的代码快速实现功能, 开发人员无需担心在主机和设备之间移动内存。代码编译与运行结果如下:

```

]: | chmod 755 q; chmod 755 run_usm.sh; if [ -x "$(command -v qsub)" ]; then ./q run_usm.sh; else ./run_usm.sh; fi
## u4f00f5e464281c4ab12edf2f369d262 is compiling SYCL_Essentials Module3 -- SYCL Unified Shared Memory - 1 of 5 usm.cpp
Device : Intel(R) Data Center GPU Max 1100
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30

```

图 1.15: 编译运行结果

代码在主机端初始化 data 数组, 设置其元素为 0 到 N-1 (即 0 到 15)。使用 SYCL 的 parallel\_for 方法在设备上并行地将 data 数组中的每个元素乘以 2。在主机端打印修改后的 data 数组的值, 结果正确。显式数据移动同理。

### 1.3.4 USM 中的数据依赖关系

在 SYCL 的统一共享内存 (USM) 模型中, 数据依赖关系是指计算任务之间的数据读取和写入依赖性。管理这些依赖关系对于确保程序的正确性和性能至关重要。

程序员可以明确地 wait 在事件对象上或使用 depends\_on 命令组内的方法来指定在任务开始之前必须完成的事件列表。在使用 USM 时管理数据依赖性有不同选项: 1. 内核任务上的 wait(): 在内核任务上使用 q.wait() 来等待下一个依赖任务开始, 但是它会阻止主机上的执行。

---

```

1  q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 2; });
2  q.wait(); // <--- wait() will make sure that task is complete before continuing
3  q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 3; });

```

---

2. 使用 in\_order 队列属性: 对队列使用 in\_order 队列属性, 这将序列化所有内核任务

---

```

1  queue q{property::queue::in_order()}; // <--- this will serialize all kernel tasks
2  q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 2; });
3  q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 3; });

```

---

3. 使用 depends\_on 方法: 使用命令组中的 h.depends\_on(e) 方法来指定任务开始之前必须完成的事件。

---

```

1  auto e = q.submit([&](handler &h) { // <--- e is event for kernel task
2      h.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 2; });
3  });
4
5  q.submit([&](handler &h) {
6      h.depends_on(e); // <--- waits until event e is complete
7      h.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 3; });
8  });

```

---

### 1.3.5 示例：USM 和数据依赖关系 1

使用 USM，有三个内核提交给设备。每个内核修改相同的数据数组。三个队列提交之间存在数据依赖性，因此需要修复代码以获得所需的输出 20。

有三种解决方案，使用 `in_order` 队列属性或使用 `wait()` 事件或使用 `depending_on()` 方法。示例中为每个队列提交添加 `wait()`，在第二和第三个内核任务中实现 `depends_on()` 方法，使用 `in_order` 队列属性代替常规队列：`queue q{property::queue::in_order()};`

```

%%writefile lab/usm_data.cpp
//=====
// Copyright © Intel Corporation
//
// SPDX-License-Identifier: MIT
// =====
#include <sycl/sycl.hpp>
using namespace sycl;

static const int N = 256;

int main() {
    queue q;
    std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";

    int *data = static_cast<int *>(malloc_shared(N * sizeof(int), q));
    for (int i = 0; i < N; i++) data[i] = 10;

    q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 2; });

    q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 3; });

    q.parallel_for(range<1>(N), [=](id<1> i) { data[i] += 5; });
    q.wait();

    for (int i = 0; i < N; i++) std::cout << data[i] << " ";
    std::cout << "\n";
    free(data, q);
    return 0;
}

```

Overwriting lab/usm\_data.cpp

图 1.16: Enter Caption







和上一个示例相比使用，`malloc_shared` 为 `data1` 和 `data2` 数组分配共享内存，使得主机和设备可以同时访问该内存。最终主机端打印修改后的 `data1` 数组的值 25，结果正确。

### 1.3.7 实验：统一共享内存

**实验要求：**用统一共享内存概念完成下面的编码练习：该代码有两个数组 `data1` 并 `data2` 在主机上初始化，为 USM 设备分配 `data1` 并将 `data2` 数据复制到设备。创建两个内核任务，一个用值的平方根进行更新，另一个用值的平方根 `data1` 进行更新 `data2`，创建第三个内核任务以添加 `data2` 到 `data1`，复制 `data1` 回主机并验证结果。根据以上要求和给出的代码框架，我编写的代码如下：

---

```

1
2  #include <sycl/sycl.hpp>
3  #include <cmath>
4  using namespace sycl;
5
6  static const int N = 1024;
7
8  int main() {
9  queue q;
10 std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";
11 // 初始化
12 int *data1 = static_cast<int *>(malloc(N * sizeof(int)));
13 int *data2 = static_cast<int *>(malloc(N * sizeof(int)));
14 for (int i = 0; i < N; i++) {
15     data1[i] = 25;
16     data2[i] = 49;
17 }
18 // STEP 1 : Create USM device allocation for data1 and data2
19 int *device_data1 = malloc_device<int>(N, q);
20 int *device_data2 = malloc_device<int>(N, q);
21 // STEP 2 : Copy data1 and data2 to USM device allocation
22 q.memcpy(device_data1, data1, N * sizeof(int)).wait();
23 q.memcpy(device_data2, data2, N * sizeof(int)).wait();
24 // STEP 3 : Write kernel code to update data1 on device with square root of its value
25 q.parallel_for(N, [=](id<1> i) {
26     device_data1[i] = std::sqrt(device_data1[i]);
27 }).wait();
28 // STEP 4 : Write kernel code to update data2 on device with square root of its value
29 q.parallel_for(N, [=](id<1> i) {
30     device_data2[i] = std::sqrt(device_data2[i]);
31 }).wait();
32 // STEP 5 : Write kernel code to add data2 on device to data1
33 q.parallel_for(N, [=](id<1> i) {

```

```

34  device_data1[i] += device_data2[i];
35  }).wait();
36  // STEP 6 : Copy data1 on device to host
37  q.memcpy(data1, device_data1, N * sizeof(int)).wait();
38
39  int fail = 0;
40  for (int i = 0; i < N; i++) if(data1[i] != 12) {fail = 1; break;}
41  if(fail == 1) std::cout << " FAIL"; else std::cout << " PASS";
42  std::cout << "\n";
43  // STEP 7 : Free USM device allocations
44  free(device_data1, q);
45  free(device_data2, q);
46  free(data1);
47  free(data2);
48
49  return 0;
50

```

```

: ! chmod 755 run_usm_lab.sh; if [ -x "$(command -v qsub)" ]; then ./q run_usm_lab.sh; else ./run_usm_lab.sh; fi

## u4f00f5e464281c4ab12edf2f369d262 is compiling SYCL_Essentials Module3 -- SYCL Unified Shared Memory - 5 of 5 usm_lab.cpp
Device : Intel(R) Data Center GPU Max 1100
PASS

```

图 1.20: 代码编译运行结果

输出显示程序在 Intel(R) Data Center GPU Max 1100 设备上运行。这表明 SYCL 程序成功检测到并使用了该 GPU 设备。输出 PASS 表示程序在执行所有计算后验证了结果，并且所有计算的结果都符合预期，代码编写正确。

代码分析：首先在主机上初始化了两个数组 data1 和 data2，所有元素分别被设置为 25 和 49。接着，使用 SYCL 的统一共享内存 (USM) 模型，在设备上为这两个数组分配内存空间。然后，将 data1 和 data2 的数据从主机复制到设备的内存空间中。在设备上，代码依次执行了三个内核任务：第一个内核任务计算 device\_data1 数组中每个元素的平方根并更新其值；第二个内核任务计算 device\_data2 数组中每个元素的平方根并更新其值；第三个内核任务将 device\_data2 数组中每个元素的值加到 device\_data1 的对应元素上。所有内核任务完成后，程序将 device\_data1 数组的数据从设备复制回主机上的 data1 数组，并进行结果验证，检查每个元素是否等于 12（因为  $\sqrt{25} + \sqrt{49} = 5 + 7 = 12$ ）。所有计算结果都正确，程序就输出 PASS，表示验证通过。最后，程序释放了在设备和主机上分配的内存空间。

## 2 进阶要求

### 2.1 04\_SYCL\_Sub\_Groups

#### 2.1.1 实验目标

1. 了解在 SYCL 中使用 Subgroups 的优势
2. 利用 Subgroup 算法提高性能和生产力
3. 使用 Subgroup Shuffle 操作避免显式内存操作

#### 2.1.2 子组 (Subgroup) 的定义

许多现代硬件平台上，工作组中的部分工作项会同时执行或在额外调度保证的情况下执行。这些工作项子集称为子组。利用子组将有助于将执行映射到低级硬件，并可能有助于实现更高的性能。

使用 ND\_RANGE 内核的并行执行有助于对映射到硬件资源的工作项进行分组。这有助于调整应用程序的性能。

ND-range 内核的执行范围分为工作组、子组和工作项，如下图所示。

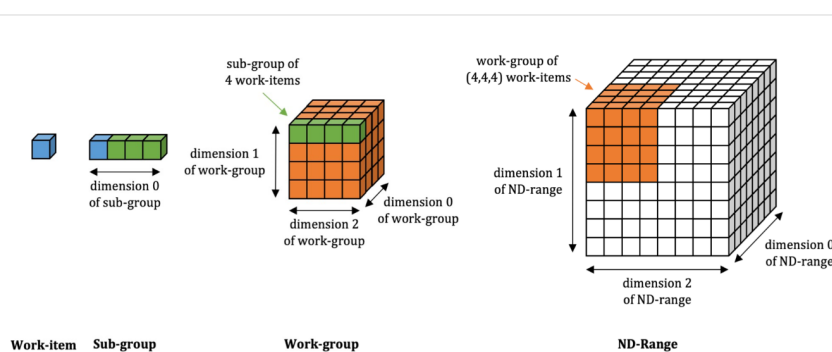


图 2.21

那么我们为什么要使用 Subgroup 呢？首先，子组中的工作项可以使用混洗操作直接进行通信，无需显式的内存操作。其次，子组中的工作项可以使用子组屏障进行同步，并使用子组内存围栏保证内存一致性。最后，子组中的工作项可以访问子组函数和算法，从而提供常见并行模式的快速实现。

#### 2.1.3 示例：打印子组信息

可以查询子组句柄以获取其他信息，例如子组中的工作项数量或工作组中的子组数量，这些信息将是开发人员使用子组实现内核代码所需要的：

---

```

1  get_local_id()//返回其子组中工作项的索引
2  get_local_range()//返回 sub_group 的大小
3  get_group_id()//返回子组的索引
4  get_group_range()//返回父工作组内的子组数量

```

---

下面的 SYCL 代码演示了子组查询方法来打印子组信息：

```

: ! chmod 755 q; chmod 755 run_sub_group_info.sh; if [ -x "$(command -v qsub)" ]; then ./q run_sub_group_info.sh; else ./run_sub_group_info.sh; fi
## u4f00f5e464281c4ab12edf2f369d262 is compiling SYCL_Essentials Module4 -- SYCL Sub Groups - 1 of 7 sub_group_info.cpp
Device : Intel(R) Data Center GPU Max 1100
sub_group id: 1 of 2, size=32
sub_group id: 0 of 2, size=32

```

图 2.22: 编译结果

### 2.1.4 实验：利用子组完成编码练习

**实验要求：**使用子组概念完成下面的编码练习：代码初始化了一个 data 大小  $N=1024$  元素的数组，我们将卸载内核任务来计算每个子组中所有项目的总和并保存在新数组中 sg\_data。我们将子组大小设置为  $S=32$ ，这将使 sg\_data 大小数组  $N/S$ ，创建 USM 共享 data 分配 sg\_data。创建一个 nd 范围内核任务，其子组大小固定为  $S$ ，reduce\_over\_group 在核任务中，使用函数计算子组和。在内核任务中，将每个 sub\_group 的总和保存到 sg\_data 数组中。在主机上，将所有元素相加，sg\_data 得到最终的总和。

根据实验要求和原先的代码框架编写代码如下：

```

1
2  #include <sycl/sycl.hpp>
3  using namespace sycl;
4
5  static constexpr size_t N = 1024; // global size
6  static constexpr size_t B = 256;  // work-group size
7  static constexpr size_t S = 32;   // sub-group size
8
9  int main() {
10   queue q;
11   std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";
12
13   // Allocate USM shared allocation for input data array and sg_data array
14   int *data = malloc_shared<int>(N, q);
15   int *sg_data = malloc_shared<int>(N / S, q);
16
17   // Initialize input data array
18   for (int i = 0; i < N; i++) data[i] = i;
19   for (int i = 0; i < N; i++) std::cout << data[i] << " ";
20   std::cout << "\n\n";
21
22   // Kernel task to compute sub-group sum and save to sg_data array
23
24   // Set fixed sub_group size of value S in the kernel below
25   q.parallel_for(nd_range<1>(N, B), [=](nd_item<1> item) {
26     auto sg = item.get_sub_group();

```

```

27  auto i = item.get_global_id(0);
28
29  // Add all elements in sub_group using sub_group reduce
30  int sub_group_sum = reduce_over_group(sg, data[i], plus<>());
31
32  // Save each sub-group sum to sg_data array
33  if (sg.leader()) {
34      sg_data[item.get_group(0) * sg.get_group_range().size() + sg.get_group_id()] = sub_group_sum;
35  }
36 }.wait();
37
38 // Print sg_data array
39 for (int i = 0; i < N / S; i++) std::cout << sg_data[i] << " ";
40 std::cout << "\n";
41
42 // Compute sum of all elements in sg_data array
43 int sum = 0;
44 for (int i = 0; i < N / S; i++) sum += sg_data[i];
45
46 std::cout << "\nSum = " << sum << "\n";
47
48 // Free USM allocations
49 free(data, q);
50 free(sg_data, q);
51
52 return 0;
53
54

```

```

! chmod 755 run_sub_group_lab.sh; if [ -x "$(command -v qsub)" ]; then ./q run_sub_group_lab.sh; else ./run_sub_group_lab.sh; fi
## u4f00f5e464281c4ab12edf2f369d262 is compiling SYCL_Essentials Module4 -- SYCL Sub Groups - 7 of 7 sub_group_lab.cpp
Device : Intel(R) Data Center GPU Max 1100
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 7
0 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 1
78 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 22
8 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278
279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 3
29 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 37
9 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429
430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 4
80 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 53
0 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580
581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 6
31 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 68
1 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731
732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 7
82 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 83
2 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882
883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 9
33 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 98
3 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023

496 1520 2544 3568 4592 5616 6640 7664 8688 9712 10736 11760 12784 13808 14832 15856 16880 17904 18928 19952 20976 22000 23024 24048 25072 26096 27120 28144 29168 30192 31216 32240

Sum = 523776

```

图 2.23: 编译运行结果

该代码首先在主机上创建并初始化了大小为  $N=1024$  的数组 `data`，然后使用 SYCL 框架在设备

上进行并行计算。它为 `data` 和 `sg_data` 分配了统一共享内存 (USM)，其中 `sg_data` 的大小为  $N/S$ ，即 32。代码中的内核任务使用 `nd_range` 设置工作组大小  $B=256$  和子组大小  $S=32$ ，在每个子组内计算总和，并使用 `reduce_over_group` 函数进行子组内求和。每个子组的总和保存到 `sg_data` 数组中。最后，程序在主机上计算并输出 `sg_data` 数组的总和，并释放分配的内存。

输出首先显示了使用的设备 Intel(R) Data Center GPU Max 1100，接着打印了初始化后的 `data` 数组内容，即从 0 到 1023 的连续整数。然后，输出了 `sg_data` 数组的内容，显示了每个子组的总和，例如第一个子组的和为 496，第二个子组的和为 1520，依此类推。最后，程序在主机上计算 `sg_data` 数组中所有元素的总和，并打印 `Sum = 523776`。这表明所有步骤都正确执行，子组求和和最终总和计算结果都符合预期。

部分实验代码已提交 Github 仓库 [git](#)