南開大學
Nankai University

计算机学院

并行程序设计实验报告

基于 Intel DevCloud 平台学习 SYCL 课程学习报告

姓名：蔡沅宏

学号：2213897

专业：计算机科学与技术

2024 年 6 月 15 日

目录

1 DevCloud 平台学习基本要求

1.1 oneAPI_Intro

1.1.1 学习目标

- 解释 oneAPI 编程模型如何解决异构环境中的编程挑战
- 使用 oneAPI 项目启用工作流
- 了解 SYCL 语言和编程模型
- 熟悉在整个课程中使用 Jupyter 笔记本进行培训

1.1.2 学习内容

目前，在数据中心领域，专用工作负载的增长显著。每种类型的数据中心硬件通常需要使用不同的语言和库进行编程，因为没有通用的编程语言或 API，这需要维护独立的代码库。开发者必须学习一整套不同的工具，因为各平台的工具支持不一致。

为解决以上问题，oneAPI 旨在提供统一的编程模型，以简化跨多种架构的开发，如图1.1所示。它包括统一且简化的语言和库，用于表达并行性。

oneAPI oneAPI 编程模型提供了一个全面而统一的开发人员工具组合，可以用于跨硬件目标的开发，它包括跨多个工作负载的一系列性能库。这些库包括为每个目标架构自定义编码的函数，因此相同函数调用可在支持的体系结构中提供优化的性能。DPC++ 基于行业标准和开放规范，以鼓励生态系统协作和创新。

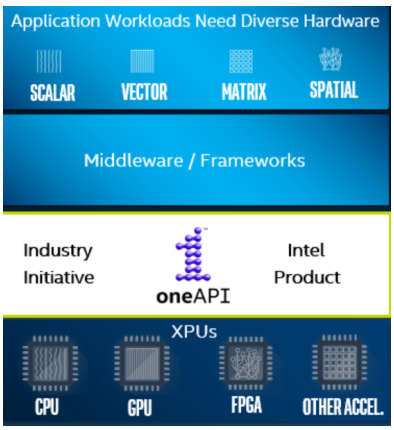


图 1.1: OneAPI 实现跨架构开发

接下来，平台用一个小程序为我们展示 SYCL 编程，并以此为基础帮助我们在 DevCloud 环境中入门，同时介绍了用于编辑和保存代码的 Jupyter 笔记本环境。这部分将在实践模块中细讲。

然后平台介绍了 SYCL，SYCL 构建在 OpenCL 之上，提供跨平台抽象层，使得异构处理器代码可以用 C++ “单源” 风格编写。而与 OpenCL 不同，SYCL 包含模板和 lambda 函数，使高层次应用软件可以干净地编码，同时优化内核代码的加速。

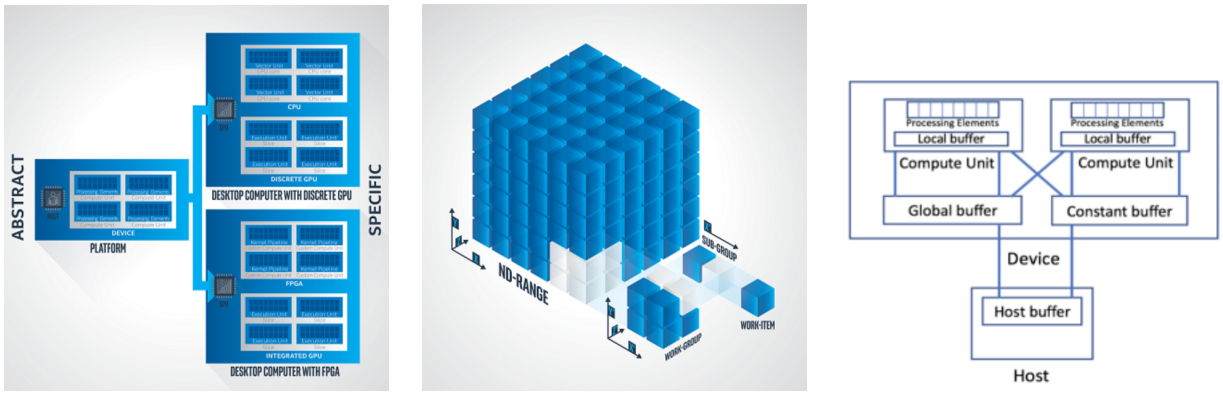
oneAPI 编程模型

1. 平台模型：oneAPI 的平台模型基于 SYCL 平台模型，主要涉及主机和设备之间的协调与控制，如图2(a)所示。主机通常是基于 CPU 的系统，负责控制一个或多个设备执行的计算工作。设备则是加速器，包含专用的计算资源，能够高效执行操作，通常比主机的 CPU 更有效。每个设备包含一个或多个计算单元，这些单元能够并行执行多个操作，每个计算单元内部又包含一个或多个处理元素，充当独立的计算引擎。这种关系可以通过图示直观地描述出来，其中一个主机可以与一台或多台设备通信，每个设备可以包含一个或多个计算单元，每个计算单元又可以包含一个或多个处理元素。

2. 执行模型：执行模型基于 SYCL 执行模型，主要定义了代码（内核）在设备上执行以及与主机交互的方式。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，其中命令组是内核调用和访问器等命令的分组，被提交到队列中执行。访问器是内存模型的一部分，同时传达执行的排序要求。程序使用队列来声明和实例化，并可以按照程序可控制的顺序或无序策略执行。设备执行模型指定了在加速器上完成计算的方式，将计算范围分布在 ND 范围、工作组、子组和工作项的层次结构中。实际内核代码表示为一个工作项执行的工作，其并行性由外部控制，工作的数量和分布由 ND 范围和工作组的大小规范控制，如图2(b)。

3. 内存型号：oneAPI 的内存模型基于 SYCL 内存模型，定义了主机和设备之间的内存交互和管理，如图2(c)所示。内存可以驻留在主机或设备上，通过缓冲区和图像两种类型的内存对象进行指定和访问。访问器指定了访问位置和模式。例如，可以通过创建缓冲区对象使主机分配的内存能够与设备通信，访问器对象定义了对缓冲区的具体访问方式。

4. 内核编程模型: oneAPI 的内核编程模型基于 SYCL 内核编程模型，支持显式并行性，程序员可以明确决定在主机和设备上执行哪些代码。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代码，即主机代码和设备代码可以位于同一源文件中。不过，主机代码和设备代码在语言一致性和功能上有所不同，SYCL 规范详细定义了主机代码和设备代码所需的语言功能。



(a) 平台模型中的关系示意图 (b) ND 范围、工作组、子组和工作项之间的关系 (c) 内存模型示意图

图 1.2: oneAPI 编程模型相关示意图

1.1.3 实践

本部分课程的实践只有一个 simple.cpp，并且不要求进行修改或编写，代码主要是为我们展示 SYCL 编程的格式，让我们熟悉平台的使用。

simple.cpp

```cpp
#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;
}
```

我们观察这段代码，代码实际上是在设备上进行并行计算与内存管理操作。首先程序创建一个 SYCL 队列，选择默认计算设备。用统一共享内存（USM）为大小为 16 的整数数组分配内存，然后在设备上并行地将数组中的每个元素乘以 2。再输出计算后的数组元素并释放内存。

运行后可以得到如下图1.3结果：

2.1.1.构建和运行

选择下面的单元格，然后单击"运行"以编译并执行上面的代码：

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

## ua08be3491b6b993322287772d458427 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: simple.cpp 程序运行截图

1.2 SYCL_Program_Structure

1.2.1 学习目标

- 解释 SYCL 基础类
- 使用设备选择卸载内核工作负载
- 决定何时使用基本并行内核和 ND 范围内核
- 在 SYCL 程序中使用统一共享内存或缓冲区访问器内存模型
- 通过动手实验练习构建示例 SYCL 应用程序

1.2.2 学习内容

本章的课程内容主要介绍 SYCL 的相关编程结构知识。

SYCL 当中有一些重要的类，来方便我们在代码中进行调用使用，以下对其进行总结整理，并未展示出课程中所示所有类，总结整理如下：

1. Device Selector（设备选择器）：这些类允许根据用户提供的启发式方法来选择特定的设备来执行内核。以下是标准设备选择器的使用示例：

```
queue q(gpu_selector_v);      // GPU 选择器
// queue q(cpu_selector_v);   // CPU 选择器
// queue q(accelerator_selector_v);  // 加速器选择器
// queue q(default_selector_v);  // 默认选择器
// queue q;  // 默认选择器（默认选择器等同于 default_selector_v）
```

输出选择的设备名称：

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

2. Queue（队列）：队列用于提交命令组以在 SYCL 运行时执行。每个队列与一个设备相关联，可以在异构系统中使用多个队列。

```
q.submit([&](handler& h) {
    // 命令组代码
});
```

3. Kernel（内核）：内核类封装了在设备上执行代码所需的方法和数据。内核对象不需要用户显式构造，而是在调用诸如 parallel_for 的内核调度函数时构造。

```
q.submit([&](handler& h) {
    h.parallel_for(range<1>(N), [=](id<1> i) {
        A[i] = B[i] + C[i];
    });
});
```

4. 选择设备内核运行的方式: 可以通过选择不同的设备选择器来确定队列将提交到哪个设备，每个队列与一个设备关联, 如图1.4所示。

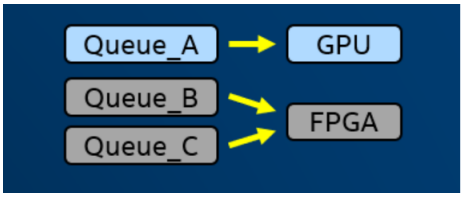


图 1.4: 每个队列与一个设备关联

这些类和方法使得在 SYCL 中编写并行程序可以方便地利用异构计算设备的能力。

此外，为了实现并行执行，并行内核允许多个操作实例同时执行，例如通过使用 parallel_for() 函数在加速器上执行简单的 for 循环。基本并行内核使用 range、id 和 item 类来描述执行空间和索引。ND-Range 内核则进一步优化性能，通过工作组的方式将任务分配到硬件上的计算单元。

基本并行内核使用演示

```
q. parallel_for (range<1>(1024) , [=](item<1> item){
  auto i = item.get_id() ;
  auto R = item.get_range() ;
}) ;
```

同时，一个 SYCL 应用程序可以使用以下两种内存模型之一编写:

- 统一共享内存模型（USM）缓冲区
- 缓冲区内存模型

统一共享内存模型是一种基于指针的内存模型方法，类似于 C/C++ 基于指针的内存分配。这使得将 C/C++/CUDA 应用程序迁移到 SYCL 变得更容易。

缓冲区内存模型允许使用一种名为缓冲区的新内存抽象，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。允许数据以 1、2 或 3 维表示，并使使用 2/3 维数据的内核编程更容易。在 USM 中多个内核之间的依赖关系是显式的，而在后者当中是隐式的。

随后，教程中给出了两向量对应相加的和的代码示例：USM 使用 malloc 方法，而缓冲区使用 buffer 方法申请内存；USM 的代码更具有 C++ 风格，依次进行了内存数据迁移到 GPU 设备（等待）、创建并行内核执行任务、数据回迁、释放内存这几个步骤，而缓冲区代码则将这几步省略，只留下了 submit 步，在设备的缓冲数据上创建缓冲处理器，再传入并行内核。可以看到两者，两种方法的共同点在于都需要申请内存、创建并行内核这两步骤。最后两种方法的输出应该都是一致的，即 1+2=3。输出结果如图1.5所示。

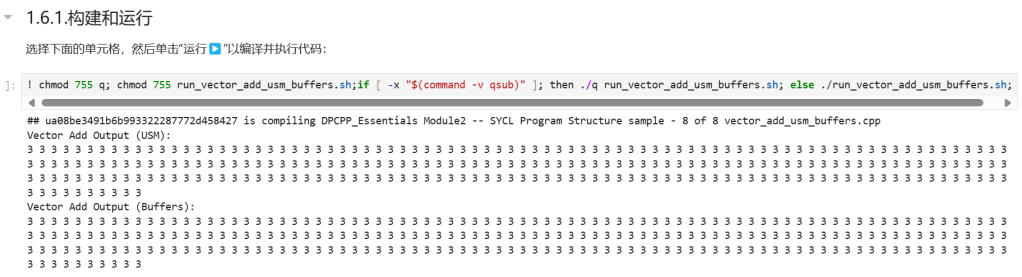


图 1.5: 使用 USM 和 Buffers 内存模型实现的向量加法计算结果

在以上所示的使用多种操作实现并行后，接下来自然会遇到同步的问题，教程中给出了使用主机访问器和缓冲区销毁进行同步的方法。以下是使用主机访问器的代码示例。

host_accessor_sample.cpp

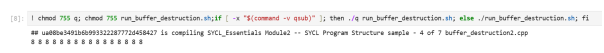
```
%%writefile lab/host_accessor_sample.cpp
#include <sycl/sycl.hpp>
using namespace sycl;

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;
}
```

在运用 HostAccessor 实现数据同步的代码中，首先初始化了一个包含 16 个元素的向量，每个元素的值为 10。然后创建了一个 SYCL 队列和一个与向量关联的 buffer 对象。在提交的命令组中，代码通过 accessor 对象访问 buffer，并在并行 for 循环中将每个元素的值减去 2。接着，代码创建了一个 hostaccessor 对象来同步数据回主机。值得注意的是，Host accessor 的创建是一个阻塞调用，只有在所有修改同一 buffer 的 SYCL 内核执行完毕后，数据才会通过 host accessor 可用。最后，代码通过 host accessor 读取并打印 buffer 中的每个元素，展示了数据同步后的结果。

随后教程介绍了用缓冲区销毁（BufferDestruction）来实现数据同步（相关数据与上述代码中的设置是相同的，只是实现数据同步的方式不同）。缓冲区的创建发生在一个单独的函数作用域内。当执行超出这个函数作用域时，缓冲区的析构函数被调用，释放数据的所有权并将数据复制回主机内存。

虽然以上实现数据同步的方式不同，但是两种方式产生的结果是相同的，如图1.7所示。

(a) 主机访问器



(b) 缓冲区销毁

图 1.6: 使用主机访问器和缓冲区销毁同步两种方式的结果

同时，SYCL 中还允许开发者使用自定义设备选择器来指定程序应该在哪种类型的设备上运行。自定义设备选择器的示例包括根据供应商名称选择设备、选择特定 GPU 设备名称以及基于设备类型设定优先级。

1. 根据供应商名称选择设备：可以编写自定义选择器函数，返回 1 以选择供应商名称为”Intel” 的设备，或返回 0 以允许其他设备作为备选选择。

2. 选择特定 GPU 设备名称：通过检查设备是否为 GPU 并且名称包含”Intel” 来选择特定的 GPU 设备。

3. 基于设备类型设置优先级：可以根据设备类型设定优先级，例如优先选择 Xeon 设备，其次是任何 GPU，再次是任何 CPU。

自定义设备选择器允许开发者根据应用程序的特定需求优化设备选择，从而实现更高效的计算资源利用。

1.2.3 实践

最后，在本章课程中给出了一个向量添加的实验练习，要求使用 SYCL Buffer 和 Accessor 概念完成以下编码练习：

- 该代码在主机上共初始化了三个向量
- 内核代码将向量 vector1 的每个元素增加 1
- 创建一个新的向量 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
- 将向量增量修改为向量相加，将 vector1 的值加到 vector2 上

以下是我补充完成的代码：

vector_add.cpp

```
%%writefile lab/vector_add.cpp
#include <sycl/sycl.hpp>
using namespace sycl;
int main() {
    const int N = 256;

    //# Initialize a vector and print values
    std::vector<int> vector1(N, 10);
    std::cout<<"\nInput Vector1: ";
    for (int i = 0; i < N; i++) std::cout << vector1[i] << " ";

    //# STEP 1 : Create second vector, initialize to 20 and print values
    //# YOUR CODE GOES HERE
    std::vector<int> vector2(N, 20);
    std::cout << "\nInput Vector2: ";
    for (int i = 0; i < N; i++) std::cout << vector2[i] << " ";

    //# Create Buffer
    buffer vector1_buffer(vector1);

    //# STEP 2 : Create buffer for second vector
```

```
    //# YOUR CODE GOES HERE
    buffer vector2_buffer(vector2);

    //# Submit task to add vector
    queue q;
    q.submit([&](handler &h) {
      //# Create accessor for vector1_buffer
      accessor vector1_accessor (vector1_buffer,h);

      //# STEP 3 - add second accessor for second buffer
      //# YOUR CODE GOES HERE
        accessor vector2_accessor(vector2_buffer, h, read_only);

      h.parallel_for(range<1>(N), [=](id<1> index) {

        //# STEP 4 : Modify the code below to add the second vector to first one
        vector1_accessor[index] += vector2_accessor[index];

      });
   });

  //# Create a host accessor to copy data from device to host
  host_accessor h_a(vector1_buffer,read_only);

  //# Print Output values
  std::cout<<"\nOutput Values: ";
  for (int i = 0; i < N; i++) std::cout<< vector1[i] << " ";
  std::cout<<"\n";

  return 0;
}
```

程序运行结果如下图1.7:

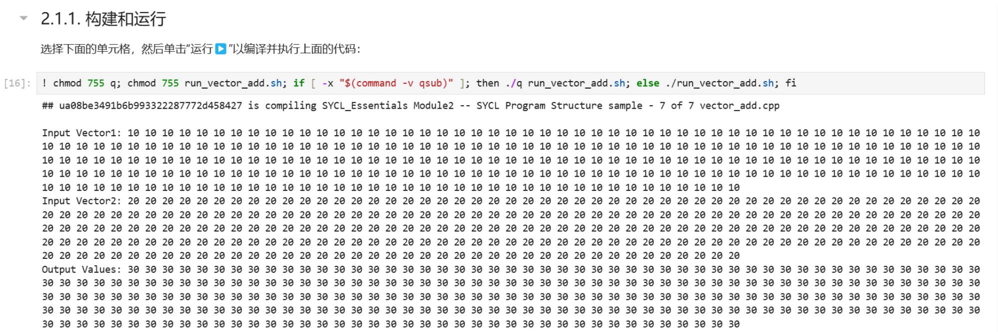


图 1.7: 向量添加实验练习运行结果

1.3 unified_Shared_Memory

1.3.1 学习目标

- 使用新的 SYCL2020 功能（如统一共享内存）来简化编程。
- 了解使用 USM 移动内存的隐式和显式方式。
- 以最优方式解决内核任务之间的数据依赖性。

1.3.2 学习内容

本小节主要介绍了有关 USM 模型的使用。Unified Shared Memory (USM) 是一种编程模型，允许在同一地址空间中共享内存，无论是在 CPU 还是 GPU 上。这简化了内存管理，使得数据在设备之间可以无需显式拷贝而共享。

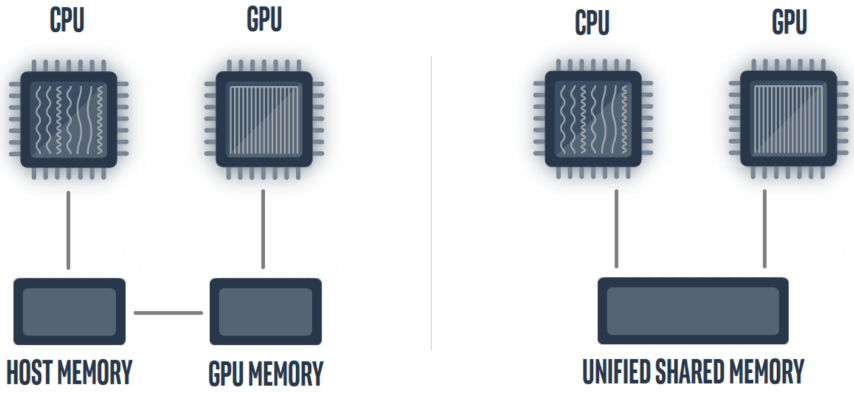


图 1.8: 不带 USM 和带 USM 的内存的开发人员视图

在 USM 编程时需要注意在 USM 中的数据依赖性。因为任务是异步执行的，多个任务可以同时执行，所以当使用 USM 时，任务之间的依赖关系必须通过事件来指定，因为任务是异步执行的，多个任务可以同时执行。

为解决数据依赖性，USM 提供了以下几种解决数据依赖性的不同选项：

1. wait() 方法：在任务之间使用 q.wait() 来等待依赖完成，但会阻塞主机执行。
2. in_order 队列属性：使用 property::queue::in_order() 属性创建的队列可以序列化所有内核任务，确保执行顺序不重叠。
3. depends_on 方法：在命令组中使用 h.depends_on(e) 方法来指定依赖事件 e，确保在任务开始之前等待事件完成。

此外，还可以使用简化的方式指定依赖性，通过在 parallel_for 中传递额外参数（事件对象）来简化指定依赖性，例如 $q.parallel_for(range<1>(N), e, [=](id<1> i) data[i] += 3;)$;。

这些方法和属性使得在使用 USM 时可以有效地管理和控制任务之间的数据依赖关系，以避免非期望的结果。

1.3.3 实践

本章课程的实践教程首先展示了单项依赖，使用 USM，并将三个内核提交到设备。每个内核都修改同一个数据数组。由于这三个队列提交之间存在数据依赖关系，因此需要修正代码以获得期望的输出 20。有三种解决方案：使用 in_order 队列属性，使用 wait() 事件，或使用 depends_on() 方法。在实践中，我使用了 wait() 事件，并得到了正确的输出结果，如下图1.9：

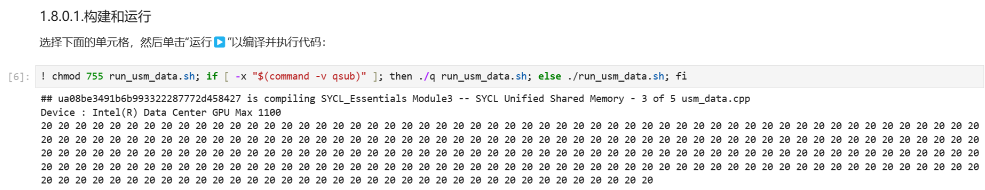


图 1.9: 单项依赖结果

接着教程介绍多项依赖，对每个被依赖的内核都命名，之后使用列表的方式以参数形式传入依赖其他内核的核当中，得到正确结果 25。实践中给出了相关提示，我根据提示，在代码中进行了如下修改：

```
auto e1 = q.parallel_for(range<1>(N), [=](id<1> i) { data1[i] += 2; });
auto e2 = q.parallel_for(range<1>(N), [=](id<1> i) { data2[i] += 3; });
q.parallel_for(range<1>(N), {e1, e2}, [=](id<1> i) { data1[i] += data2[i]; }).wait();
```

得到了正确的输出结果，如图1.10所示：

图 1.10: 多项依赖结果

在本章教程的实验练习部分，要求我们使用统一共享内存 (Unified Shared Memory（USM）) 概念完成以下编码练习：

- 在主机上初始化了两个数组 data1 和 data2
- 为 data1 和 data2 创建 USM 设备分配，并将数据复制到设备
- 创建两个内核任务，一个用于更新 data1 的值为其平方根，另一个用于更新 data2 的值为其平方根
- 创建第三个内核任务，将 data2 的值加到 data1 上

- 将 data1 复制回主机并验证结果

根据以上实验要求，我补充完成了以下代码：

usm_lab.cpp

```
%%writefile lab/usm_lab.cpp
#include <sycl/sycl.hpp>
using namespace sycl;
static const int N = 1024;
int main() {
  queue q;
  std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";

  //intialize 2 arrays on host
  int *data1 = static_cast<int *>(malloc(N * sizeof(int)));
  int *data2 = static_cast<int *>(malloc(N * sizeof(int)));
  for (int i = 0; i < N; i++) {
    data1[i] = 25;
    data2[i] = 49;
  }

  //# STEP 1 : Create USM device allocation for data1 and data2
  //# YOUR CODE GOES HERE
  int *d_data1 = malloc_device<int>(N, q);
  int *d_data2 = malloc_device<int>(N, q);

  //# STEP 2 : Copy data1 and data2 to USM device allocation
  //# YOUR CODE GOES HERE
  q.memcpy(d_data1, data1, N * sizeof(int)).wait();
  q.memcpy(d_data2, data2, N * sizeof(int)).wait();

  //# STEP 3 : Write kernel code to update data1 on device with square of its value
    auto e1 = q.parallel_for(N, [=](auto i) {

    //# YOUR CODE GOES HERE
    d_data1[i] = sqrt(d_data1[i]);
  });

  //# STEP 4 : Write kernel code to update data2 on device with square of its value
  auto e2 = q.parallel_for(N, [=](auto i) {

    //# YOUR CODE GOES HERE
      d_data2[i] = sqrt(d_data2[i]);
  });

  //# STEP 5 : Write kernel code to add data2 on device to data1
  auto e3 = q.parallel_for(N, {e1, e2}, [=](auto i) {

  //# YOUR CODE GOES HERE
```

```
45      d_data1[i] += d_data2[i];
46    });
47
48    //# STEP 6 : Copy data1 on device to host
49    //# YOUR CODE GOES HERE
50    q.memcpy(data1, d_data1, N * sizeof(int)).wait();
51
52    //# verify results
53    int fail = 0;
54    for (int i = 0; i < N; i++) if(data1[i] != 12) {fail = 1; break;}
55    if(fail == 1) std::cout << " FAIL"; else std::cout << " PASS";
56    std::cout << "\n";
57
58    //# STEP 7 : Free USM device allocations
59    //# YOUR CODE GOES HERE
60    free(d_data1, q);
61    free(d_data2, q);
62
63    //# STEP 8 : Add event based kernel dependency for the Steps 2 - 6
64    free(data1);
65    free(data2);
66
67    return 0;
68  }
```

从代码中我，我们可以看到，它创建了三个并行计算任务（内核）。第一个内核任务将 d_data1 中每个元素的值更新为其平方根，因为其本来值为 25，所以更新后为 5；第二个内核任务将 d_data2 中每个元素的值更新为其平方根，其值本来为 49，所以更新之后为 7；第三个内核任务将 d_data2 中每个元素的值加到 d_data1 中对应的元素上，则 d_data1 中所有元素都为 12，之后将 d_data1 从设备复制回主机后，那么 data1 数组中所有元素都为 12。那么最后，验证 data1 数组中的每个元素是否都等于 12 时是成立的，所以如果代码没有编写错误的话，输出结果应该为 PASS。

程序运行后得到如下图1.11结果：

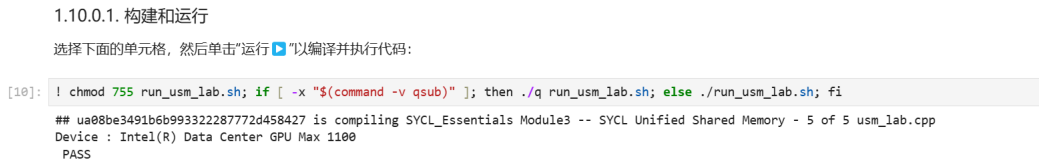


图 1.11: 统一共享内存实验练习程序运行结果

如上图所示，实验结果符合预期，程序编写正确。

2 DevCloud 平台学习进阶要求

2.1 Sub_Groups

2.1.1 学习目标

- 了解在 SYCL 中使用子组的优势
- 利用子组算法提高性能和生产力
- 使用子组随机播放操作可避免显式内存操作

2.1.2 学习内容

首先，在许多现代硬件平台上，工作组中工作项的子集是同时执行的，或者具有额外的计划保证。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。

也就是说，通过子组，我们可以将任务划分为多个子集将其映射到不同硬件上以获得更快的效率。

同时，课程中提到：使用 ND_RANGE 内核进行并行执行有助于将工作项分组映射到硬件资源。这有助于为性能进行应用调优。

ND-range 内核的执行范围如图2.12所示，分为工作组、子组和工作项。

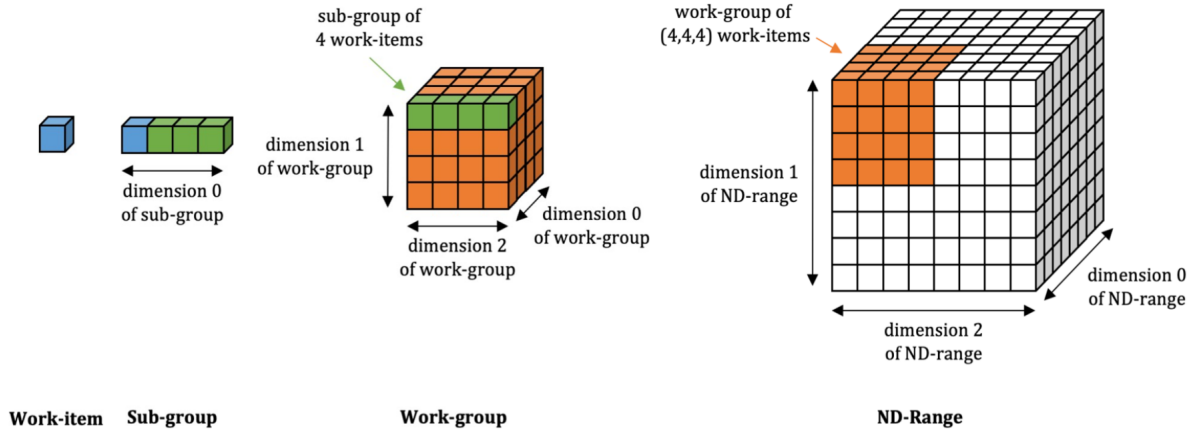


图 2.12: ND-range 内核的执行范围

分工如下所示：

工作项 (Work-item)	表示核函数的各个实例。
工作组 (Work-group)	整个迭代空间被划分为更小的组，称为工作组， 工作组内的工作项在硬件上的单个计算单元上调度执行。
子组 (Subgroup)	工作组内的工作项的一个子集，同时执行，可能映射到向量硬件上。

接下来，教程介绍使用子组时可以用到的相关函数和成员变量，总结如下：

1. 子组信息:
   - 可以通过查询子组句柄获取详细信息，如子组中工作项的索引 (get_local_id())、子组的大小 (get_local_range())、子组在父工作组中的索引 (get_group_id()) 以及父工作组内子组的数量 (get_group_range())。

2. 子组大小:

   - 为了优化性能，可以设置特定的子组大小，如 8、16 或 32。编译器通常会选择最佳的大小，但也可以使用 reqd_sub_group_size(S) 属性显式设置。

3. 子组函数和算法:

   - 这些函数和算法作用于子组内的工作项,例如 select_by_group、shift_group_left、shift_group_right、permute_group_by_xor 等。它们可以执行洗牌、减少、扫描和投票等操作，用于优化并行计算。

4. 子组洗牌 (Shuffle):

   - 允许子组内的工作项直接通信，而无需显式内存操作。例如，使用 permute_group_by_xor 可以交换工作项的值，从而提高内核执行效率，减少对全局内存的访问。

为了让我们更好地理解 Shuffle 的内涵，教程中给出了如下的代码示例：

子组随机操作示例

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
  auto sg = item.get_sub_group();
  auto i = item.get_global_id(0);
  /* Shuffles */
  //data[i] = select_by_group(sg, data[i], 2);
  //data[i] = shift_group_left(sg, data[i], 1);
  //data[i] = shift_group_right(sg, data[i], 1);
  data[i] = permute_group_by_xor(sg, data[i], 1);
});
```

以上代码中利用 permute_group_by_xor 交换两个工作项的值, 如图2.13所示：

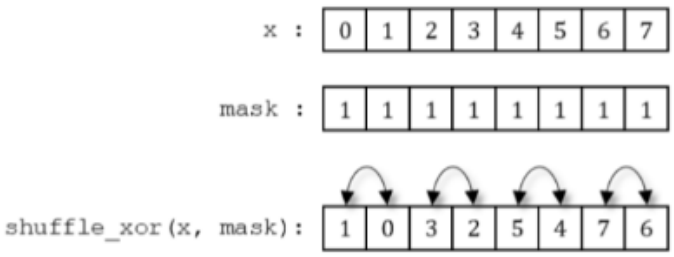


图 2.13: 两个工作项值的交换示意图

2.1.3 实践

本章课程实践部分需要我们通过使用子组中的 reduce 方法实现从 1 到 1024 的求和。

题目要求我们将把内核任务卸载到计算每个子组中所有项的总和，并保存在新数组 sg_data 中。子组大小要设置为 S=32，这也就使得 sg_data 数组的大小为 N/32。然后我们需要运用第三章的内容创建 USM 共享分配，用于 data 和 sg_data。同时创建一个 nd-range 内核任务，固定子组大小为 32。

在内核任务中，我需要按要求使用 reduce_over_group 函数计算子组总和。并将每个子组的总和保存到 sg_data 数组中。最后将 sg_data 的所有元素相加以获得最终总和。

实现代码如下：

sub_group_lab.cpp

```cpp
#include <CL/sycl.hpp>
using namespace cl::sycl;

static constexpr size_t N = 1024; // global size
static constexpr size_t B = 256;  // work-group size
static constexpr size_t S = 32;   // sub-group size

int main() {
    queue q;

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

    int* data = malloc_shared<int>(N, q);
    int* sg_data = malloc_shared<int>(N/S, q);

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

    for (int i = 0; i < N; i++)
        std::cout << data[i] << " ";

    std::cout << "\n\n";

    q.parallel_for(nd_range<1>(N, B), [=](nd_item<1> item)
        [[intel::reqd_sub_group_size(S)]] {
        auto sg = item.get_sub_group();
        auto i = item.get_global_id(0);
        int result = reduce_over_group(sg, data[i], plus<>());
        if (sg.leader()) {
            sg_data[item.get_group(0) * (B / S) + sg.get_group_id()[0]] = result;
         }
    }).wait();

    for (int i = 0; i < N/S; i++)
        std::cout << sg_data[i] << " ";

    std::cout << "\n";

    int sum = 0;
    for (int i = 0; i < N/S; i++) {
        sum += sg_data[i];
    }

```

```
44      std::cout << "\nSum = " << sum << "\n";
45
46      free(data, q);
47      free(sg_data, q);
48
49      return 0;
50  }
```

根据以上代码和预期完成功能，我们可以预测程序运行结果。如果程序逻辑正确，那么应该能正常分出 32 个子组，并下放至硬件进行计算，保存至 sg_data[] 中，那么执行代码后，应该会输出 sg_data 数组的内容，即每个子组中元素的总和。最后输出计算得到的所有子组和的总和 Sum。

运行结果如下图2.14所示：

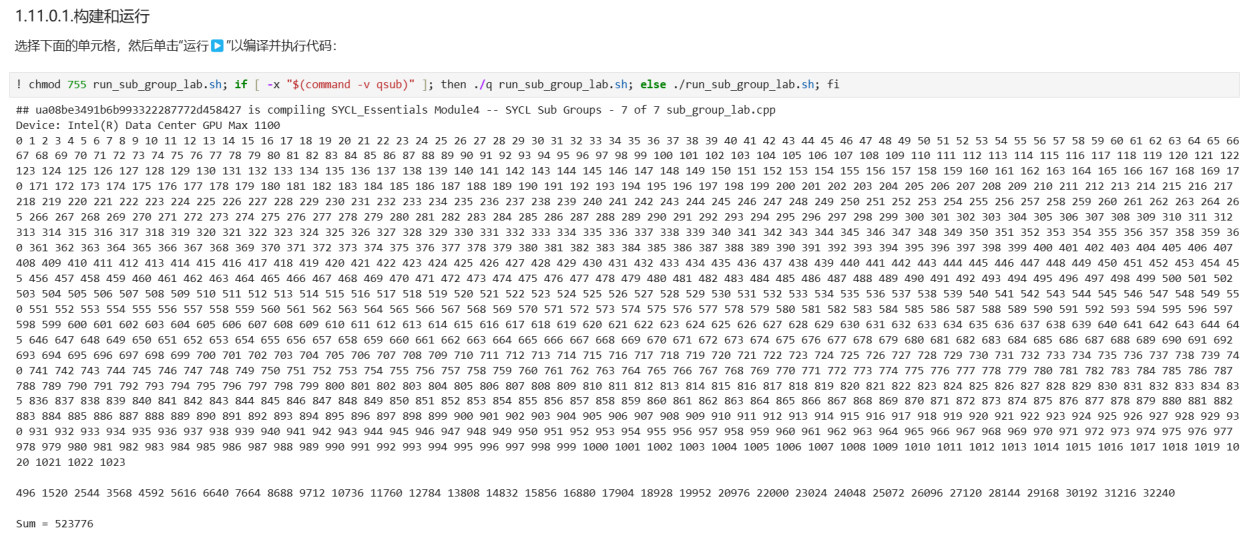


图 2.14: 程序运行截图

我们知道，从 1 加到 1023 的结果确实为 523776，且每一个子组的和与前后相邻子组的和应该相差 1024，这些数据在图中均可以对应上，说明程序编写是正确的。

2.2 Intel_Advisor

2.2.1 学习目标

1. 卸载顾问（Offload Advisor）
   - 运行卸载顾问并生成 HTML 报告
   - 阅读和了解报告中的指标
   - 在目标硬件上获取应用程序的性能估计
   - 确定哪些循环适合卸载
2. 车顶线分析（Roofline Analysis）
   - 说明英特尔顾问如何执行 GPU 屋顶线分析。
   - 使用命令行语法运行 GPU Roofline Analysis。
   - 使用 GPU 屋顶线分析确定有效的优化策略。

2.2.2 学习内容

Intel Offload Advisor Intel Offload Advisor 可以用分析应用程序，特别是循环，确定哪些部分适合加速或迁移到特定硬件上。它可以进行各种分析，包括时间、浮点运算、内存传输和数据依赖，以评估循环的性能。

课程中提到了可以使用这个 Advisor 的方法，即通过命令行脚本 collect.py 和 analyze.py 运行，当然，两个脚本各有不同的任务：collect.py 负责收集时间、浮点运算、循环次数等数据；analyze.py 则基于收集的数据生成详细报告。

详细使用步骤如下：

1. 从 GitHub 克隆 Intel oneAPI 示例存储库。
2. 使用 CMake 和 Make 构建和编译应用程序。
3. 运行 advixe-python 使用 collect.py 和 analyze.py 收集数据并生成报告。

那么生成的报告中有什么相关信息，课程中提到：生成 HTML 报告中会包括，比如在特定硬件上的预期加速（例如 Gen9），或者是发现潜在的瓶颈，以及对于建议适合迁移和不适合迁移的循环部分的分析。同时，还会有一些附加输出，比如包括 CSV 文件（report.csv，whole_app_metric.csv）、图形表示（program_tree.dot，program_tree.pdf）以及用于调试和详细分析的 JSON/LOG 文件。

那么这些信息都能帮助开发人员优化代码，可以针对特定硬件加速器通过提供性能瓶颈的洞察和适合迁移的区域。

Offload Advisor 报告示例如下图2.15所示：

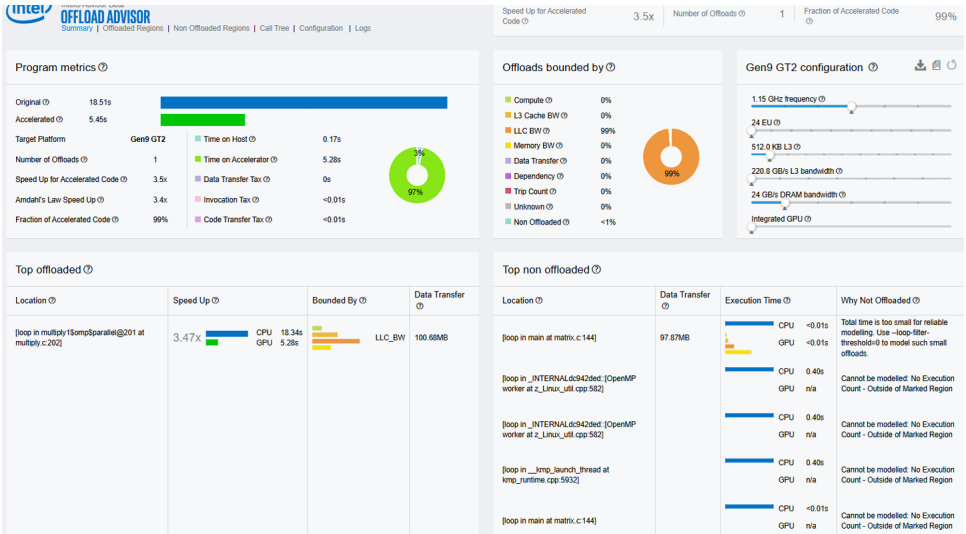


图 2.15: Offload Advisor 报告示例

根据以上总结分析，我们可以看到，Intel Offload Advisor 能够通过分析和报告应用程序的适合迁移到特定硬件的部分，帮助开发人员优化代码性能和效率。这无疑能帮助开发人员通过详细的分析和报告做出基于信息的决策，以实现最佳性能。

Intel Roofline Analysis 车顶线模型与 Offload Adivisor 不同的是，它是一种图形化分析工具，用于评估和优化计算机程序的性能。它帮助确定程序在特定硬件上的最大理论性能极限。但是我们可

以注意到，两者针对的是计算机程序运行在硬件上能获得的最大收益的不同方面，所以两者实际上可以结合分析一个程序，以获得多方面的信息对程序进行优化和改进。

因为车顶线模型是一种图形化工具，所以它可以通过图表的形式清晰地展示程序在计算性能和内存带宽之间的关系。下图2.16是在平台上生成的一个车顶线分析图表：

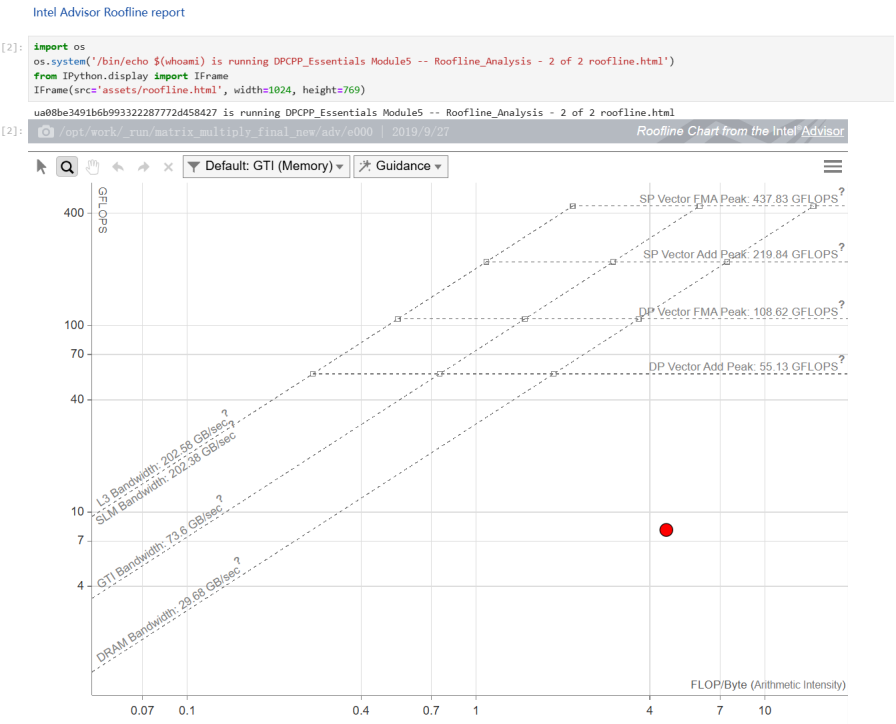


图 2.16: Roofline Analysis 分析图表示例

那么，这样的信息报告就能为开发人员提供指导，帮助确定优化策略，开发人员就能够针对硬件特性最大化程序性能，特别是在选择合适的硬件加速器或优化内存使用方面进行优化改进，从而达到最佳的加速效果。