DPC++调度特征与内存管理

王顺洪

2020.02.22

硬件抽象

- **管理内存有三种抽象**
  - 统一共享内存（USM）
    - 基于指针
  - Buffers
    - 用于描述一到三维的数组
    - 不由程序直接访问，而是通过访问器对象使用
  - Images
    - 是一种特殊类型的缓冲区
    - 提供特定于图像处理的额外功能

调度特征

- **通过调用parallel_for或者single_task定义命令组**
  - 命令组可以执行两种类型的操作：
    - 内核操作，表示我们希望在设备上执行的计算
    - 显式内存操作，如USM的memcpy、memset，buffers的copy、fill等

- **通过三种不同的模式描述依赖关系**
  - 顺序队列
  - 基于事件的依赖关系
  - 通过缓冲区和访问器来表示命令组之间的数据依赖关系
    - 访问器指定如何使用它们读取或写入缓冲区对象中的数据，运行时使用这些信息来确定不同内核之间存在的数据相关性
    - 在声明accessor时还可以提供额外的访问信息，以帮助调度

调度特征

- **以一个Y型依赖图为例**
  - 顺序队列

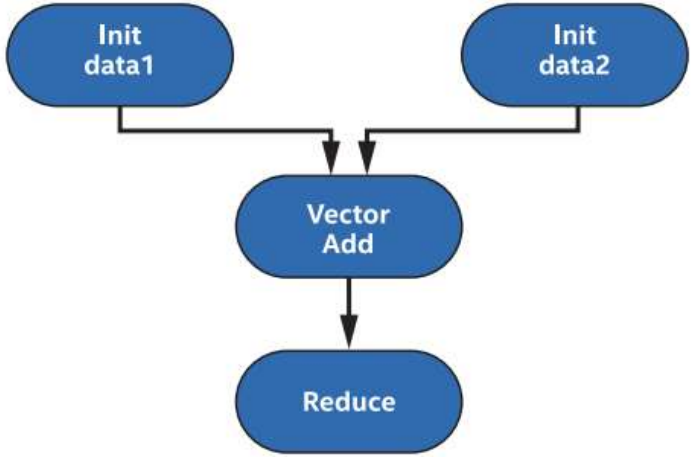

Figure 8-2. *"Y" pattern dependence graph*

```
constexpr int N = 42;

queue Q{property::queue::in_order()};

int *data1 = malloc_shared<int>(N, Q);
int *data2 = malloc_shared<int>(N, Q);

Q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });

Q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });

Q.parallel_for(N, [=](id<1> i) { data1[i] += data2[i]; });

Q.single_task([=]() {
  for (int i = 1; i < N; i++)
    data1[0] += data1[i];

  data1[0] /= 3;
});

Q.wait();
assert(data1[0] == N);
```

Figure 8-6. *"Y" pattern with in-order queues*

调度特征

- **以一个Y型依赖图为例**
  - 基于事件

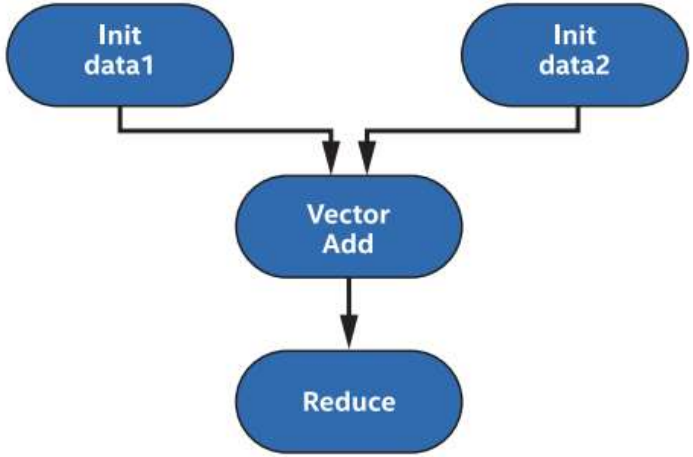

Figure 8-2. *"Y" pattern dependence graph*

```
constexpr int N = 42;
queue Q;

int *data1 = malloc_shared<int>(N, Q);
int *data2 = malloc_shared<int>(N, Q);

auto e1 = Q.parallel_for(N,
           [=](id<1> i) { data1[i] = 1; });

auto e2 = Q.parallel_for(N,
            [=](id<1> i) { data2[i] = 2; });

auto e3 = Q.parallel_for(range{N}, {e1, e2},
            [=](id<1> i) { data1[i] += data2[i]; });

Q.single_task(e3, [=]() {
  for (int i = 1; i < N; i++)
    data1[0] += data1[i];

  data1[0] /= 3;
});

Q.wait();
assert(data1[0] == N);
```

指定事件依赖关系

Figure 8-7. *"Y" pattern with events*

调度特征

- **以一个Y型依赖图为例**
  - 基于accessors

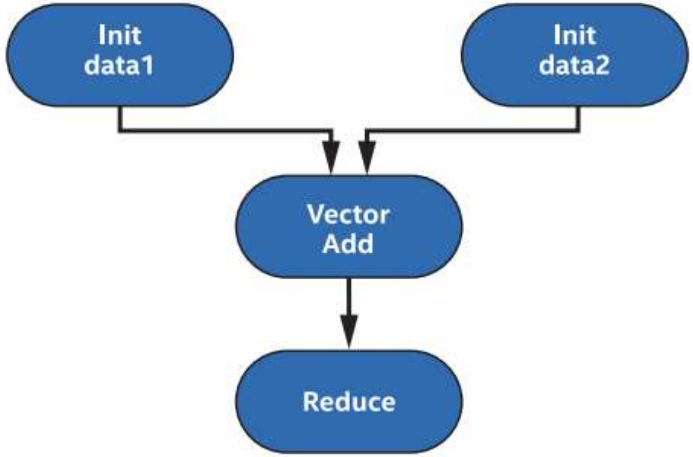

Figure 8-2. *"Y" pattern dependence graph*

```
constexpr int N = 42;
queue Q;

buffer<int> data1{range{N}};
buffer<int> data2{range{N}};

Q.submit([&](handler &h) {
  accessor a{data1, h};
  h.parallel_for(N, [=](id<1> i) { a[i] = 1; });
});

Q.submit([&](handler &h) {
  accessor b{data2, h};
  h.parallel_for(N, [=](id<1> i) { b[i] = 2; });
});

Q.submit([&](handler &h) {
  accessor a{data1, h};
  accessor b{data2, h, read_only};
  h.parallel_for(N, [=](id<1> i) { a[i] += b[i]; });
});

Q.submit([&](handler &h) {
  accessor a{data1, h};
  h.single_task([=]() {
    for (int i = 1; i < N; i++)
      a[0] += a[i];
    a[0] /= 3;
  });
});

host_accessor h_a{data1};
assert(h_a[0] == N);
```

由accessor确定
数据依赖关系

Figure 8-8. *"Y" pattern with accessors*

内存模型

❑ 标准C++与SYCL和DPC++中不同内存模型概念

Feature	Standard C++	SYCL / DPC++
Atomic Objects	`std::atomic`	Not available.
Atomic References	`std::atomic_ref` (C++20 onwards)	`sycl::atomic_ref`

Memory Orders	`relaxed`
	`consume`
	`acquire`
	`release`
	`scq_rel`
	`seq_cst`
		`relaxed`
		
		`acquire`
		`release`
		`scq_rel`
		`seq_cst`

Memory Scopes	Not available.
	
	Behavior of atomics and fences matches DPC++ `system` scope.
		`work_item`
		`sub_group`
		`work_group`
		`device`
		`system`

Fences	`std::atomic_thread_fence`	`sycl::atomic_fence`
Barriers	`std::barrier` (C++20 onwards)	`nd_item::barrier` `sub_group::barrier`
Address Spaces	All memory is in a single (host) address space.	Host Device (Global) Device (Local) Device (Private) Shared (USM)

Figure 19-9. *Comparing standard C++ and SYCL/DPC++ memory models*

内存模型

❑ Barriers

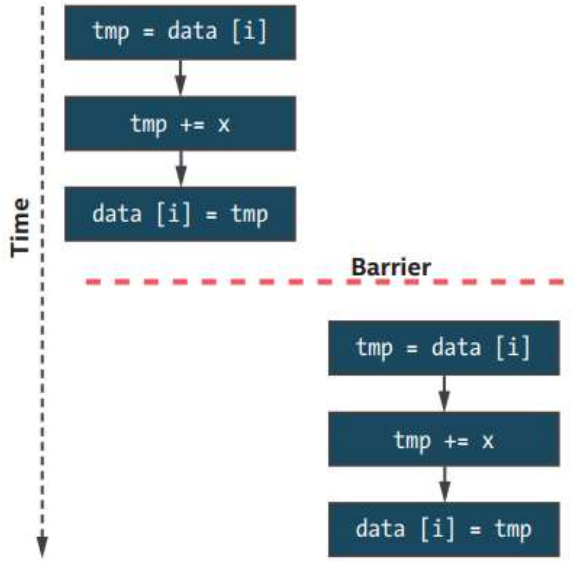

Figure 19-5. *Two instances of* `data[i] += x` *separated by a barrier*

```
int* data = malloc_shared<int>(N, Q);
std::fill(data, data + N, 0);

// Launch exactly one work-group
// Number of work-groups = global / local
range<1> global{N};
range<1> local{N};

Q.parallel_for(nd_range<1>{global, local}, [=](nd_item<1> it) {
   int i = it.get_global_id(0);
   int j = i % M;
   for (int round = 0; round < N; ++round) {
     // Allow exactly one work-item update per round
     if (i == round) {
       data[j] += 1;
     }
     it.barrier();
   }
 }).wait();

for (int i = 0; i < N; ++i) {
  std::cout << "data [" << i << "] = " << data[i] << "\n";
}
```

Figure 19-6. *Avoiding a data race using a barrier*

内存模型

❑ 一个原子操作的示例

```
int* data = malloc_shared<int>(N, Q);
std::fill(data, data + N, 0);

Q.parallel_for(N, [=](id<1> i) {
   int j = i % M;
   atomic_ref<int, memory_order::relaxed, memory_scope::system,
              access::address_space::global_space> atomic_data(data[j]);
   atomic_data += 1;
 }).wait();

for (int i = 0; i < N; ++i) {
  std::cout << "data [" << i << "] = " << data[i] << "\n";
}
```

Figure 19-7. *Avoiding a data race using atomic operations*

内存模型

memory_order枚举类

可为fences和atomics操作提供内存顺序参数

- relaxed。读写操作可以在操作完成之前或者之后重新排序，没有任何限制。没有顺序保证。
- acquire。程序中在某一操作之后出现的读写操作必须在该操作之后进行（即在操作之前不能重新排序）
- release。在程序中某一操作之前出现的读写操作必须在该操作之前发生（即他们不能再该操作之后重新排序），并且保证先前的写操作对于已通过相应获取同步的其他程序实例是可见的
- acq_rel。这个操作既有acquire又有release。读写操作无法围绕该操作重新排序，必须使先前的写入可见
- seq_cst。这个操作充当acquire、release或者两者都有。取决于是读、写还是read-modify-write操作。以该存储器顺序进行的所有操作均按顺序一致的顺序进行观察。

内存模型

memory_scope枚举类

标准C++内存模型假定应用程序在具有单个地址空间的单个设备上执行。这些假设都不适合于DPC++应用程序：应用程序的不同部分在不同的设备（即主机设备和一个或多个加速器设备）上执行；每个设备都有多个地址空间（即私有、本地和全局）；每个设备的全局地址空间可以不相交（取决于USM的支持）

- work_item
- sub_group/work_group
- device
- system