### OpenCL 2.0 异构计算 [第三版] （Heterogeneous Computing with OpenCL 2.0）

https://www.bookstack.cn/read/Heterogeneous-Computing-with-OpenCL-2.0/README.md

### Intel opencl sdk下载安装

https://software.intel.com/content/www/us/en/develop/tools/opencl-sdk/choose-download.html

### OpenCL较大的改变

1、**共享虚拟内存**

主机和设备端可以共享复杂数据结构指针，比如：树和链表；以减少花在数据结构转换上的精力。

2、**动态并行**

可以在不用主机代码参与的情况下，进行内核的加载，为的就是减小加载内核所产生的瓶颈。

3、**统一地址空间**

这样同样的函数就可以处理GPU和CPU上的数据，以减少编程的难度。

4、**C++原子操作**

工作项可以跨越工作组共享数据和设备，可以让更多的算法使用OpenCL实现。

# 第1章 异构计算简介

### 异构计算：串行处理、并行处理

### 加速的方法

1、**分而治之**

将一个问题递归的划分为数个子问题，直到可用的计算资源能够解决划分后的子问题。

2、**分散-聚合**

先发送一部分输入数据到每个并行资源中，然后将这些输入数据处理后的结果进行收集，再将这些结果合并到一个结果数据集中。

### CMOS

互补金属氧化物半导体，Complementary Metal Oxide Semiconductor

### 并发与并行

并行编程必须是并发的，不过并发编程不一定并行。虽然很多并发程序可以并行执行，但是互相有依赖的并发任务就不能并行了。比如：交错执行就符合并发的定义，而不能并行的执行。所以，并行是并发的一个子集，并发程序是所有程序集的一个子集。

### 线程

进程中的所有线程都会共享一些资源(比如：**内存、打开的文件、全局变量**)，不过他们也有属于自己的资源(比如：**堆栈、自动变量**)。

线程使用**全局共享地址空间分配出的变量**进行**通讯**。

通讯时需要有**同步机制**来保证**同一个内存区域的内容不会被多个线程更新**。

### 共享内存模型

关键的特性：**不需要编程者去管理数据的移动**

在这样的系统中，线程如何去更新全局变量，底层硬件和编程者要达成共识，并遵守**“内存一致性模型”**

### 瓶颈

内存一致性模型下，能够使用的处理器相对较少，这是因为**共享总线**和**相干性协议**将会成为性能瓶颈。

系统中多一些松散的共享内存，相对来说会好一些；当有成规模的内核在共享内存系统中时，其会让系统变得复杂，并且内核间的互相通讯也要付出很高的代价。

大多数多核CPU平台都支持某一种共享内存，OpenCL也可以在支持共享内存的设备上运行。

### 消息通讯机制

**消息传递接口(MPI)库**在当今环境下依旧是很受欢迎的**消息传递中间件**

### 并行计算的粒度

线程中计算量与通讯量(比如，需要在线程间进行同步的变量)

### 细粒度并行

使用细粒度并行时，需要考虑如下几点：

1、**计算量不要过大**，这样每个线程都有足够的工作可做。

2、**尽可能减少在数据同步上的开销**，这样每个线程能够独立的完成自己的任务。

3、**负载的划分**也很重要，因为有大量的独立任务需要并行执行，设计良好的任务调度器都能灵活控制负载，并保证在多任务运行的同时，让线程上的达到均衡。

### 粗粒度并行

使用粗粒度并行时，需要考虑如下几点：

1、**计算量**肯定要**高于**细粒度并行时的计算量，因为不会像细粒度并行那样，有很多线程同时执行。

2、编程者使用粗粒度编程时，需要**了解应用的整体结构**，让粗粒度中的每个线程作为任务，服务于应用。

### 粗细粒度选择

当**同步和通讯的开销大于计算**，那么将**并行粒度加大**，更有利于控制同步和通讯所需的过高开销。

细粒度并行能**减少负载不均**和**访存延迟在性能上的开销**(对于GPU来说更是如此，其为粒度之细可以切换线程时达到零开销，同时也能隐藏访存延迟)。

### 将数据视为向量

当一个算法需要对大量数据进行同一组操作时，就可将数据视为向量，执行同一操作时，多个数据作为输入，经过向量操作后输出多个数据。

这种执行方式就是利用**单指令多数据(SIMD)**的方式对数据进行处理，可并行硬件可以利用这种执行方式，并行的对不同的数据进行处理。这种粒度的并行与**向量的大小**有关，通过**SIMD执行单元的多数据处理**来获得应用性能的提升。

### 数据共享的用途

1、一个任务中的输入，**依赖**于另外一个任务的输出——例如：生产者-消费者模型、流水线执行模型。

2、需要将中间结果进行**综合**(比如：归约，还有图1.5里的例子)。

理想状态下，可以尝试**将应用能并行的部分剥离出来**，确保并行的部分没有数据依赖，不过这只是理想状态而已。**栅栏**和**锁**有时会用来做数据的同步。

### OpenCL2.0的三种共享虚拟内存

1、粗粒度共享缓存

2、细粒度共享缓存

3、系统级细粒度共享缓存

使用共享虚拟内存需要OpenCL实现**将系统中主机和设备端的地址链接起来**。这就允许在OpenCL内核中使用**数据结构的指针**(比如：链表)，之前版本的OpenCL标准是不支持内核中使用自定义数据结构指针。

粗粒度缓存支持**通过API的调用**，来**更新整个缓存的内容**。

细粒度缓存和系统级细粒度的粒度为**字节级**，这就意味着无论是主机端，还是设备端，对数据的更新将会**立即同步**。

细粒度共享内存是按内存一致模型中定义好的**内存序**，在**同步点**对数据进行同步。

### OpenCL2.0新特性

1、嵌套并行化

2、共享虚拟内存

3、管道内存对象

4、C11原子操作

5、增强图像支持

### OpenCL多级别的并行化

能够将应用高效的映射到同构或异构、单独或多个CPU或GPU上，以及其他硬件供应商提供的硬件系统当中。

### OpenCL “设备端-主机端”语言

主机作为对其他设备的管理者。

设备端语言被设计用来**映射内存系统和执行模型**。

主机端语言用较低的开销，为复杂的并发程序**搭建任务管道**。

### OpenCL基于任务和基于数据的并行

OpenCL的内核类似于SPMD模型，内核就是并行单元(OpenCL称为**工作项**)中执行的实例，内核实例创建并入队后，会被映射到支持标量和矢量的硬件上运行。

### OpenCL2.0共享虚拟内存

共享虚拟内存是一项很重要的特性，其对能减轻编程者的负担，特别是在类似APU这样使用统一物理内存的系统上。

### OpenCL2.0内存一致模型

提供**“获取/释放”语义**以缓解编程者在不明确的锁上耗费不必要的精力

# 第2章 设备架构

### 超标量执行方式

VLIW和硬件管理：通过查询地址，并行执行同指令流中不相关的指令

SIMD与向量并行：让指令在数据上并行执行

### 向量操作

将向量化操作通用化，并且向量操作通常会用来处理较长连续的数据序列，通常会使用**流水线**的形式进行向量操作，而非同时对多个数据进行操作，并且向量操作对连续、密集的内存读写给予了更多的支持。

### 并行的三种常见的形式

指令并行，数据并行、线程并行。

并行的方式就是执行多个独立的指令流

### 两种方式实现硬件多线程

1、**并发多线程(SMT)**

多线程的指令在执行资源上交替执行(通过超标量扩展的调度逻辑和线程资源)

缺点：需要对更多的状态信息进行保存，并且会让指令间得依赖关系和调度逻辑变得更加复杂

2、**时域多线程**

每个线程都能通过轮询的方式连续执行

目的：两个线程共享一个ALU。

优势：

1、调度逻辑简单。

2、流水线的延迟可以隐藏对多个线程的切换(调度)，减少转发逻辑。

3、当有线程缓存未命中，或等待另一个分支计算的结果等之类事件，都能通过改变线程指令顺序进行掩盖，并且执行更多的线程能更好的掩盖流水线上的延迟。

原理：以延迟换取最大的吞吐量，通过时域多线程的方式，来你做吞吐量的计算：多个线程交替执行，以保证设备处于忙碌状态，不过每个独立线程的执行时间要多于其理论最小执行时间。

### 片上系统优点

1、将很多元素融合在一个设备上，这样在生产的时候一次就能搞定，避免在制造阶段耗费太多成本。

2、更少的功能将会降低设备上的面积占用率，可以节省功耗和降低设备体积，这对于移动通讯领域很重要。

3、更短的距离意味着数据交互和信息传递完全可以在一个内存系统中进行，并且降低了交互的功耗。

4、低通讯延迟可以增加处理器对负载分发的次数。

### 内存访问的局部方式

1、**空域**：两个以上(包括两个)的读或写操作，对内存上(一定程度上)附近的地址。

2、**时域**：在相对小的时间窗内对同一地址进行两个以上(包括两个)的读或写操作。

# 第3章 介绍OpenCL

### OpenCL标准

1、**平台模型**

指定一个host处理器，用于任务的调度。以及一个或多个device处理器，用于执行OpenCL任务(OpenCL C Kernel)。这里将硬件抽象成了对应的设备(host或device)。

2、**执行模型**

定义了OpenCL在host上运行的环境应该如何配置，以及host如何指定设备执行某项工作。这里就包括host运行的环境，host-device交互的机制，以及配置内核时使用到的并发模型。并发模型定义了如何将算法分解成OpenCL工作项和工作组。

3、**内核编程模型**

定义了并发模型如何映射到实际物理硬件。

4、**内存模型**

定义了内存对象的类型，并且抽象了内存层次，这样内核就不用了解其使用内存的实际架构。其也包括内存排序的要求，并且选择性支持host和device的共享虚拟内存。

### clGetPlatformIDs()

查找制定系统上的可用OpenCL平台的集合。

在具体的OpenCL程序中，这个API一般会调用两次，用来查询和获取到对应的平台信息：

第一次调用这个API需要**传入num\_platforms作为数量参数，传入NULL作为平台参数**。这样就能**获取在该系统上有多少个平台可供使用**。编程者可以开辟对应大小的空间(指针命名为platforms)，来存放对应的平台对象(类型为 cl\_platform\_id)。

第二次调用该API时，就可**将platforms传入来获取对应数量的平台对象**。平台查找完成后，使用clGetPlatformInfo()API可以**查询对应供应商所提供的平台**，然后决定使用哪个平台进行运行OpenCL程序。

clGetPlatformIDs()这个API**需要在其他API之前调用**。

格式：

cl\_int

clGetPlatformIDs(

cl\_uint num\_entries,

cl\_platform\_id \***platforms**,

cl\_uint \***num\_platforms**)

### clGetDeviceIDs()

查询平台上可用的设备，多了平台对象和设备类型作为入参。

需要三步创建device：

第一，**查询设备的数量**；

第二，**分配对应数量的空间来存放设备对象**；

第三，**选择期望使用的设备(确定设备对象)**。

device\_type参数可以将设备限定为GPU(CL\_DEVICE\_TYPE\_GPU)，限定为CPU(CL\_DEVICE\_TYPE\_CPU)，或所有设备(CL\_DEVICE\_TYPE\_ALL)，当然还有其他选项。这些参数都必须传递给clGetDeviceIDs()。相较于平台的查询API，clGetDeviceInfo()API可用来查询每个设备的名称、类型和供应商。

格式：

cl\_int

clGetDeviceIDs(

cl\_platform\_id platform,

cl\_device\_type **device\_type**,

cl\_uint num\_entries,

cl\_device\_id \*devices,

cl\_uint \*num\_devices)

### clCreateContext()

创建上下文对象。

properties参数用于限制上下文作用的范围。这个参数可由特定的平台提供，其能够使能与图像的互用性，或使能其他能力。

通过限制给定平台的上下文，允许编程者使用多个平台创建的不同的上下文，并且能在一个平台中混用多个供应商提供的设备。

另外，创建上下文时**必须要使用设备对象**，并且编程者可以**设置一个用户回调函数**，还可以**额外传递一个错误码**(需要在错误码对象的生命周期内)用于**获取API运行的状态**。

格式：

cl\_context

clCreateContext(

const cl\_context\_properties \***properties**,

cl\_uint num\_devices,

const cl\_device\_id \*devices,

void (CL\_CALL\_BACK \*pfn\_notify)(

const char \*errinfo,

const void \*private\_info,

size\_t cb,

void \*user\_data),

void \*user\_data,

cl\_int \*errcode\_ret)

### clCreateContextFromType()

可以使用所有的设备类型(CPU、GPU和ALL)创建上下文。

### clGetContextInfo()

创建上下文之后，可以用来查询上下文中设备的数量，以及具体的设备对象。

### 命令指定的行为

执行内核、进行数据传递、执行同步

### clCreateCommandQueueWithProperties()

创建命令队列，且将命令队列与一个device进行关联。

peoperties参数是由一个位域值组成，其可**使能命令性能分析功能**(CL\_QUEUE\_PROFILING\_ENABLE)，以及/或**允许命令乱序执行**(CL\_QUEUE\_OUT\_OF\_DRDER\_EXEC\_MODE\_ENABLE)。

格式：

cl\_command\_queue

clCreateCommandQueueWithProperties(

cl\_context context,

cl\_device\_id device,

cl\_command\_queue\_properties peoperties,

cl\_int \*errcode\_ret)

### clEnqueue

任何以clEnqueue开头的OpenCL API都能**向命令队列提交一个命令**，并且这些API都需要一个命令队列对象作为输入参数。例如，**clEnqueueReadBuffer()**将device上的数据传递到host，**clEnqueueNDRangeKernel()**申请一个内核在对应device执行。

所有的clEnqueue开头的API，均有三个共同的参数：

1、**事件链表的指针**，其指定了当前命令依赖的事件列表

2、**等待列表的长度**

3、**表示当前命令执行的事件指针**，这个指针用于依赖该命令的其他命令。

### clFlush()

对命令队列进行**栅栏操作**，需要一个**命令队列**作为参数。

clFlush()将会阻塞host上的执行线程，直到命令队列上的命令**都从队列上移出**。移出命令队列后的命令，就已经提交到device端，不过不一定完全执行完成。

格式：

cl\_int clFlush(cl\_command\_queue command\_queue);

### clFinish()

对命令队列进行**栅栏操作**，需要一个**命令队列**作为参数。

clFinish()的调用将会阻塞host上的执行线程，直到命令队列上的**所有命令执行完毕**，其功能就是和同步栅栏操作一样。

格式：

cl\_int clFinish(cl\_command\_queue command\_queue);

### 事件

OpenCL API中，用来**指定命令之间依赖关系的对象**称为事件(event)

### 命令状态

1、**Queued**：命令处于命令队列中。

2、**Submitted**：命令从命令队列中移除，已经提交到设备端执行。

3、**Ready**：命令已经准备好在设备上执行。

4、**Running**：命令正在设备上执行。

5、**Ended**：命令已经在设备上执行完成。

6、**Complete**：所有命令以及其子命令都执行完成。

### CL\_COMPLETE

当命令成功的执行完成，事件的状态将会被设置为CL\_COMPLETE。

如果命令非正常终止，事件的状态将会为一个负数值。这种情况下，有非正常终止的命令队列，以及其他在同一上下文上创建的命令队列，都将不能正常使用或运行。

### clGetEventInfo()

查询事件

### APIclWaitForEvents()

用于和host同步，该API阻塞host的执行线程，等待指定事件队列上的所有命令执行完毕。

格式：

cl\_int

clWaitForEvents(

cl\_uint num\_events,

const cl\_event \*event\_list)

### 设备端入队

执行中的内核现在可以让另外一个内核进入命令队列中。这种情况下，正在执行的内核可以称为“**父内核**”，刚入队的内核称为“**子内核**”。虽然，父子内核是以异步的方式执行，但是**父内核需要在子内核全部结束后才能结束**。我们可**通过与父内核关联的事件对象来对执行状态进行查询，当事件对象的状态为CL\_COMPLETE时，就代表父内核结束执行**。

设备端的命令队列是**无序**命令队列，其具有无序命令队列的所有特性。设备端命令会进入到设备端产生的命令队列中，并且使用事件的方式来存储各个命令间的依赖关系。这些事件对象只有执行在设备端的父内核可见。

### 向量加法

1、**串行加法实现**

void vecadd(int C, int A, int \*B, int N){

for (int i = 0; i < N; ++i){

C[i] = A[i] + B[i];

}

}

2、**分块处理向量加法**，使用**粗粒度多线程**(例如，使用POSIX CPU线程)。输入向量上的不同元素被分配到不同的核上

void vecadd(int C, int A, int \*B, int N, int NP, int tid){

int ept = N / NP; // 每个线程所要处理的元素个数

for (int i = tid ept; i < (tid + 1) ept; ++i){

C[i] = A[i] + B[i];

}

}

3、**OpenCL版向量相加内核**

kernel

void vecadd(global int C, \_\_global int A, \_\_global int \*B){

int tid = get\_global\_id(0); // OpenCL intrinsic函数

c[tid] = A[tid] + B[tid];

}

### 条带处理

对于一个多核设备，我们要不就使用底层粗粒度线程API(比如，Win32或POSIX线程)，要不就使用数据并行方式(比如，OpenMP)。

粗粒度多线程需要对任务进行**划分(循环次数)**。因为循环的迭代次数特别多，并且每次迭代的任务量很少，这时我们就需要**增大循环迭代的粒度**，这种技术叫做“条带处理”

### 工作项

OpenCL C上的**并发执行单元**称为工作项(work-item)。

每一个工作项都会执行内核函数体。OpenCL运行时可以创建很多工作项，其个数可以和输入输出数组的长度相对应，并且工作项在运行时，以一种默认合适的方式映射到底层硬件上(CPU或GPU核)。

当OpenCL设备开始执行内核，OpenCL C中提供的内置函数可以让工作项知道自己的编号。

### get\_global\_id(0)

获取**当前工作项的位置**，以及**访问到的数据位于数组中的位置**。get\_global\_id()的参数用于获取指定维度上的工作项编号，其中“0”这个参数，可获取当前第一维上工作项的ID信息。

### 每个维度上工作项的数量(NDRange)

当要执行一个内核时，编程者需要指定每个维度上工作项的数量(NDRange)。一个NDRange可以是一维、二维、三维的，其不同维度上的工作项ID映射的是相应的输入或输出数据。

NDRange的每个维度的工作项数量由**size\_t类型**指定。

### 工作组

为了增加NDRange的可扩展性，需要将工作项继续的划分成**更细粒度的线程**，那么需要再被划分的工作项就称为工作组(work-group)。

与工作项类似，工作组也需要从三个维度上进行指定，每个维度上的工作项有多少个。为了保证硬件工作效率，工作组的大小通常都是**固定**的。

在同一个工作组中的工作项具有一些特殊的关系：一个工作组中的工作项**可以进行同步**，并且他们**可以访问同一块共享内存**。

工作组的大小在每次分配前就已经固定，所以对于更大规模的任务分配时，**同一工作组中的工作项间交互不会增加性能开销**。实际上，**工作项之间的交互开销与分发的工作组的大小并没有什么关系**，这样就能保证在更大的任务分发时，OpenCL程序依旧能保证良好的扩展性。

允许各维度上的工作项和工作组数量**不成倍数关系**，工作数量被分成两部分：第一部分是编程者指定给工作组的工作项，另一部分是剩余的工作组，这些工作可以没有工作项。

允许编程者不去分配工作组的尺寸，其会在实现中进行**自动的划分**；这样的话，开发者就可以传递NULL作为工作组数组的尺寸。

### 使用源码创建内核的步骤

1、将OpenCL C源码存放在一个**字符数组**中。如果源码以文件的形式存放在硬盘上，那么需要将其读入内存中，并且存储到一个字符数组中。

2、调用**clCreateProgramWithSource()**通过源码可以创建一个**cl\_program类型对象**。

3、创建好的程序对象需要进行编译，编译之后的内核才能在一个或多个OpenCL设备上运行。调用**clBuildProgram()**完成对内核的**编译**，如果编译有问题，该API会将输出错误信息。

4、之后，需要创建**cl\_kernel类型**的内核对象。调用**clCreateKernel()**，并**指定对应的程序对象和内核函数名**，从而创建内核对象。

### 创建内核对象

将内核函数名和程序对象传入clCreateKernel()，如果程序对象是合法的，并且这个函数名是存在的，那么久会返回一个内核对象。

**一个程序对象**上可以提取**多个内核对象**。每个OpenCL上下文上可有**多个OpenCL程序对象**，这些程序对象可由OpenCL源码创建。

格式：

cl\_kerenl

clCreateKernel(

cl\_program program,

const char \*kernel\_name,

cl\_int \*errcode\_ret)

### clBuildProgram()创建NVIDIA GPU的中间码

NVIDIA这种中间码为**并行线程执行(PTX，parallel thread execute)**。

**好处**：如果一个设备上的GPU指令集有所变化，其产生的代码功能依旧不会受到影响。这种产生中间代码的方式，能给编译器的开发带来更大的发展空间。

### clGetProgramInfo()

**返回一些关于程序对象的信息**。这个函数中其中有一个参数可以传**CL\_PROGRAM\_BINARIES**，这个参数可以返回**供应商指定的二进制程序对象**(通过clBuildProgram()编译之后)。

OpenCL程序对象另一个特性是能在构建之后，产生**二进制格式的文件**，并写入到硬盘。

### clCreateProgramWithBinary()

这个函数可以通过一系列编译好的二进制文件直接创建程序对象，对应的二进制文件可以通过**clGetProgramInfo()**得到。

使用二进制表示的OpenCL内核，因为**不需要以源码的方式存储，以及安装编译器**，这种方式更容易部署OpenCL程序。

### clSetKernelArg()

**对内核的参数进行设置**。这个API需要传入一个内核对象，指定参数的索引值，参数类型的大小，以及对应参数的指针。内核参数列表中的类型信息，需要一个个的传入内核中。

与调用C函数不同，我们不能直接将参数赋予内核函数的参数列表中。执行一个内核需要通过一个**入队函数**进行发布。由于核内的语法为C，且内核参数具有持续性(如果我们只改变参数里面的值，就没有必要再重新进行赋值)。

格式：

cl\_int

clSetKernelArg(

cl\_kernel kernel,

cl\_uint arg\_index,

size\_t arg\_size,

const void \*arg\_value)

### clEnqueueNDRangeKernel()

**入队一个命令道命令队列**。

内核执行时，有四个地方与工作项创建有关：

1、**work\_dim**参数指定了创建工作项的维度(一维，二维，三维)。

2、**global\_work\_size**参数指定NDRange在每个维度上有多少个工作项，

3、**local\_work\_size**参数指定NDRange在每个维度上有多少个工作组。

4、**global\_work\_offset**参数可以指定全局工作组中的ID是否从0开始计算。

和所有clRnqueueAPI相同，需要提供一个**event\_wait\_list**，当这个参数不为NULL时，当前的内核要等到等待列表中的所有任务都完成才能执行。

该API也是**异步**的：命令入队之后，函数会立即返回(通常会在内核执行之前就返回)。OpenCL中**clWaitForEvents()**和**clFinish()**可以在host端阻塞等待，直到对应的内核对象执行完成。

格式：

cl\_int

clEnqueueNDRangeKernel(

cl\_command\_queue command\_queue,

cl\_kernel kernel,

cl\_uint work\_dim,

const size\_t \*global\_work\_offset,

const size\_t \*global\_work\_size,

const size\_t \*local\_work\_size,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event \*event\_wait\_list,

cl\_event \*event)

### OpenCL三种内存类型

数组、图像、管道

### clCreateBuffer()

**为Buffer类型分配内存，并返回一个内存对象**。这种类型中数据在内存上是连续的。理论上，这种类型可以在设备端以指针的方式使用。

格式：

cl\_mem

clCreateBuffer(

cl\_context context,

cl\_mem\_flags flags,

size\_t size,

void \*host\_ptr,

cl\_int \*errcode\_ret)

创建一个数组需要知道其**长度**和**创建在哪一个上下文对象上**；创建之后，与该上下文对象相关联的设备就能看到这个内存对象。

对于第二个标识参数，是用来指定设备端可对内存进行的操作，可以是**“只读”、“只写”或“读写”**。

其他标识需要在数组创建的时候指定。

### clCreateImage()

图像数组数据不能直接访问，因为相邻的数据并不保证在内存上连续存储。图像没有数据类型或维度，图像对象的创建需要通过描述符，让硬件了解这段内存数据的具体信息。

格式：

cl\_mem

clCreateImage(

cl\_context context,

cl\_mem\_flags flags,

const **cl\_image\_format** \*image\_format,

const cl\_image\_desc \*image\_desc,

void \*host\_ptr,

cl\_int \*errcode\_ret)

图像对象中的每个元素通过格式描述符来表示(cl\_image\_format)。格式描述符用于描述图像元素在内存上是如何存储，以及使用通道的信息。

### 通道序

通道序(channel order)指的是由多少个通道元素组成一个图像元素(例如，RGBA就是由四个通道值组成一个像素，其通道序为4)，并且**通道类型(channel type)**指定了每个元素的大小。大小可以设置为1到4字节中的任意值，这样就能表现多种不同的格式(从整型到浮点)。其他数据元通过**图像描述符(cl\_image\_desc)**提供，其包括了图像的**类型**和**维度**。

### OpenCL读写图像数据的内置函数

图像读写函数需要额外的参数，并且这些函数根据图像的具体数据类型进行使用。

**read\_imagef()**函数就适用于读取**浮点型**的数值；

**read\_imageui()**函数就使用与读取**无符号整型**的数值。

这些函数在使用的数据类型上有些不同，但在读取方面至少需要有**一组访问坐标**和**一个采样器对象**。采样器可以指定，设备访问到图像外部时，这些不存在的数据应该**如何获取**，**是否使用差值**，以及**是否对坐标进行归一化**。

写入图像需要**手动**将数据转换成对应的存储数据格式(例如，对应的通道和对应的数据大小)，目的坐标也需要手动的进行转换。

### clCreatePipe()

创建一个管道对象，需要提供包的大小和管道中可容纳包的最大数量(例如，创建时固定了管道中可容纳包的最大值)。

管道内存对象就是一个数据元素(被称为packets)**队列**，其和其他队列一样，遵循**FIFO(先进先出)**的方式。一个管道对象具有一个**写入末尾点**，用于表示**元素由这里插入**；并且，有一个**读取末尾点**，用于表示**元素由这里移除**。

格式：

cl\_mem

clCreatePipe(

cl\_context context,

cl\_mem\_flags flags,

cl\_uint pipe\_packet\_size,

cl\_uint pipe\_max\_packets,

const cl\_pipe\_properties \*properties,

cl\_int \*errcode\_ret)

### clGetPipeInfo()

返回管道中包的大小和整体大小(也就是可容纳包的最大值)。

属性参数是一个保留参数，在OpenCL 2.0阶段，这个值只能传NULL。

### 内核对管道的读写

任意时间点，只能有一个内核向管道中存入包，并且只有一个内核从管道中读取包。

为了支持“生产者-消费者”设计模式，一个内核与写入末尾点连接(生产者)，同时另一个内核与读取末尾点连接(消费者)。

同一个内核不能同时对一个管道进行读取和存入。

### OpenCL访问管道

图像和管道都是不透明的数据结构，其只能通过OpenCL C的内置函数进行访问(比如，read\_pipe()和write\_pipe())。OpenCL C也提供相应的函数，可以保留管道的读取点和写入点。

内置函数允许管道在**工作组级别**上进行访问，而不需要单独访问每个工作项，并且能在工作组级别上执行同步。

### clEnqueueWriteBuffer()和clEnqueueReadBuffer()

内存对象是一个数组时，用于**主机端和设备端的内存互传**。

显式的数据传输也需要检索主机端的内存空间。因此，通常情况下使用**显式数据传输命令**，在内存对象被内核调用之前，为其写入相应的数据到设备端；以及，在内存对象最后一次使用之后，读取其数据到主机端。

设备端使用的内存与主机内存通常是**离散**的，当命令执行时，数据可能就已经传输到设备端了(比如，使用PCIe总线)。

clEnqueueWriteBuffer()格式：

cl\_int

clEnqueueWriteBuffer(

cl\_command\_queue command\_queue,

cl\_mem buffer,

cl\_bool **blocking\_write**,

size\_t offset,

size\_t cb,

const void \*ptr,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event \*event\_wait\_list,

cl\_event \*event)

除了命令队列，该函数还需要**数组型的内存对象**，**所要传输的数据大小**，以及**数组的偏移**。偏移量和传输数据可以将原始数据的一个子集进行传输。

如果需要**数据传输完成再返回**，**blocking\_write**参数可以设置成CL\_TRUE；如果设置成CL\_FALSE，则以**更加高效的异步方式**进行传输，且函数会**立即返回**，无需等到数据完全传输完再返回。

### OpenCL内存

1、**主机内存**

主机内存可在主机上使用，其并不在OpenCL的定义范围内。使用对应的OpenCL API可以进行主机和设备的数据传输，或者通过共享虚拟内存接口进行内存共享。

2、**设备内存**

设备内存，指定是能在执行内核中使用的内存空间。

### OpenCL设备内存分类

1、**全局内存**

全局内存对于**执行内核中的每个工作项**都是**可见**的(类似于CPU上的内存)。当数据**从主机端传输到设备端**，数据就存储在全局内存中。有数据需要**从设备端传回到主机端**，那么对应的数据需要存储在全局内存中。

其关键字为**global或\_\_global**，关键字加在指针类型描述符的前面，用来表示该指针指向的数据存储在全局内存中。

2、**常量内存**

常量内存并非为只读数据设计，但其能让**所有**工作项**同时**对该数据进行访问。这里存储的值通常不会变化。OpenCL的内存模型中，常量内存为全局内存的子集，所以**内存对象传输到全局内存的数据**可以指定为“常量”。

使用关键字**constant或\_\_constant**将相应的数据映射到常量内存。

3、**局部内存**

局部内存中的数据，只有在**同一工作组内的工作项**可以共享。通常情况下，局部内存会映射到片上的物理内存，例如：软件管理的暂存式存储器。

比起全局内存，局部内存具有**更短的访问延迟**，以及**更高的传输带宽**。

调用**clSetKernelArg()**设置局部内存时，只需要**传递大小**，而无需传递对应的指针，相应的局部内存会由运行时进行开辟。

使用**local或\_\_local**关键字来描述**指针**，从而来定义局部内存(例如，local int \*sharedData)。不过，**数据**也可以通过关键字**local**，静态申明成局部内存变量(例如，local int[64])。

4、**私有内存**

私有内存只能由**工作项自己**进行访问。局部变量和非指针内核参数通常都在私有内存上开辟。

实践中，私有变量通常都**与寄存器对应**。不过，当寄存器不够私有数组使用是，这些溢出的数据通常会**存储到非片上内存(高延迟的内存空间)**上。

### 通用地址空间

通用地址空间支持**指向私有、局部和全局地址指针的互相转换**，这样编程者只需要写一个简单的函数即可，其指针参数可以接受这指向这三种内存区域的指针。

### 创建并执行OpenCL应用的步骤

1、**查询平台和设备信息**

cl\_int status; // 用于错误检查

// 检索平台的数量

cl\_uint numPlatforms = 0;

status = clGetPlatformIDs(0, NULL, &numPlatforms);

// 为每个平台对象分配足够的空间

cl\_platform\_id platforms = NULL;

platforms = (cl\_platform\_id )malloc(numPlatforms \* sizeof(cl\_platform\_id));

// 将具体的平台对象填充其中

status = clGetPlatformIDs(numPlatforms, platforms, NULL);

// 检索设备的数量

cl\_uint numDevices = 0;

status = clGetDeviceIDs(platforms[0], CL\_DEVICE\_TYPE\_ALL, 0, NULL, &numDevices);

// 为每个设备对象分配足够的空间

cl\_device\_id devices;

devices = (cl\_device\_id )malloc(numDevices \* sizeof(cl\_device\_id));

// 将具体的设备对象填充其中

status = clGetDeviceIDs(platforms[0], CL\_DEVICE\_TYPE\_ALL, numDevices, devices, NULL);

2、**创建一个上下文**

// 创建的上下文包含所有找到的设备

cl\_context context = clCreateContext(NULL, numDevices, devices, NULL, NULL, &status);

3、**为每个设备创建一个命令队列**

// 为第一个发现的设备创建命令队列

cl\_command\_queue cmdQueue = clCreateCommandQUeueWithProperties(context, devices[0], 0, &status);

4、**创建一个内存对象(数组)用于存储数据**

// 向量加法的三个向量，2个输入数组和1个输出数组

cl\_mem bufA = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, datasize, NULL, &status);

cl\_mem bufB = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, datasize, NULL, &status);

cl\_mem bufC = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, datasize, NULL, &status);

5、**拷贝输入数据到设备端**

// 将输入数据填充到数组中

status = clEnqueueWriteBuffer(cmdQueue, bufA, CL\_TRUE, 0, datasize, A, 0, NULL, NULL);

status = clEnqueueWriteBuffer(cmdQueue, bufB, CL\_TRUE, 0, datasize, B, 0, NULL, NULL);

6、**使用OpenCL C代码创建并编译出一个程序**

// 使用源码创建程序

cl\_program program = clCreateProgramWithSource(context, 1, (const char \*\*)&programSource, NULL, &status);

// 为设备构建(编译)程序

status = clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);

7、**从编译好的OpenCL程序中提取内核**

// 创建向量相加内核

cl\_kernel kernel = clCreateKernel(program, “vecadd”, &status);

8、**执行内核**

// 设置内核参数

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), &bufA);

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), &bufB);

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), &bufC);

// 定义工作项的空间维度和空间大小

// 虽然工作组的设置不是必须的，不过可以设置一下

size\_t indexSpaceSize[1],workGroupSize[1];

indexSpaceSize[0] = datasize / sizeof(int);

workGroupSize[0] = 256;

// 通过执行API执行内核

status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, indexSpaceSize, workGroupSize, 0, NULL, NULL);

9、**拷贝输出数据到主机端**

// 将输出数组拷贝到主机端内存中

status = clEnqueueReadBuffer(cmdQueue, bufC, CL\_TRUE, 0, datasize, C, 0, NULL, NULL);

10、**释放资源**

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseCommandQueue(cmdQueue);

clReleaseMemObject(bufA);

clReleaseMemObject(bufB);

clReleaseMemObject(bufC);

clReleaseContext(context);

### 使用C API实现的OpenCL向量相加 完整代码

// This program implements a vector addition using OpenCL

// System includes

// OpenCL includes

// OpenCL kernel to perform an element-wise addition

const char programSouce =

“kernel \n”

“void vecadd(global int A, \n”

“ global int \*B, \n”

“ global int \*C) \n”

“{ \n”

“ // Get the work-item’s unique ID \n”

“ int idx = get\_global\_id(0); \n”

“ \n”

“ // Add the corresponding locations of \n”

“ // ‘A’ and ‘B’, and store the reasult in ‘C’ \n”

“ C[idx] = A[idx] + B[idx]; \n”

“} \n”

;

int main(){

// This code executes on the OpenCL host

// Elements in each array

const int elements = 2048;

// Compute the size of the data

size\_t datasize = sizeof(int) \* elements;

// Allocate space for input/output host data

int A = (int )malloc(datasize); // Input array

int B = (int )malloc(datasize); // Input array

int C = (int )malloc(datasize); // Output array

// Initialize the input data

int i;

for (i = 0; i < elements; i++){

A[i] = i;

B[i] = i;

}

// Use this to check the output of each API call

cl\_int status;

// Get the first platforms

cl\_platform\_id platform;

status = clGetPlatformIDs(1, &perform, NULL);

// Get the first devices

cl\_device\_id device;

status = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_ALL, 1, &device, NULL);

// Create a context and associate it with the device

cl\_context context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);

// Create a command-queue and associate it with device

cl\_command\_queue cmdQueue = clCreateCommandQUeueWithProperties(context, device, 0, &status);

// Allocate two input buffers and one output buffer for the three vectors in the vector addition

cl\_mem bufA = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, datasize, NULL, &status);

cl\_mem bufB = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, datasize, NULL, &status);

cl\_mem bufC = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, datasize, NULL, &status);

// Write data from the input arrays to the buffers

status = clEnqueueWriteBuffer(cmdQueue, bufA, CL\_FALSE, 0, datasize, A, 0, NULL, NULL);

status = clEnqueueWriteBuffer(cmdQueue, bufB, CL\_FALSE, 0, datasize, B, 0, NULL, NULL);

// Create a program with source code

cl\_program program = clCreateProgramWithSource(context, 1, (const char\*\*)&programSource, NULL, &status);

// Build(compile) the program for the device

status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);

// Create the vector addition kernel

cl\_kernel kernel = clCreateKernel(program, “vecadd”, &status);

// Set the kernel arguments

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), &bufA);

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), &bufB);

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), &bufC);

// Define an incde space of work-items for execution

// A work-group size is not required, but can be used.

size\_t indexSpaceSize[1], workGroupSize[1];

// There are ‘elements’ work-items

indexSpaceSize[0] = elements;

workGroupSize[0] = 256;

// Execute the kernel

status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, indexSpaceSize, workGroupSize, 0, NULL, NULL);

// Read the device output buffer to the host output array

status = clEnqueueReadBuffer(cmdQueue, bufC, CL\_TRUE, 0, datasize, C, 0, NULL, NULL);

// Free OpenCL resouces

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseCommandQueue(cmdQueue);

clReleaseMemObject(bufA);

clReleaseMemObject(bufB);

clReleaseMemObject(bufC);

clReleaseContext(context);

// free host resouces

free(A);

free(B);

free(C);

return 0;

}

### 使用C++ Wapper实现的OpenCL向量相加

int main(){

const int elements = 2048;

size\_t datasize = sizeof(int) \* elements;

int A = new int[elements];

int B = new int[elements];

int \*C = new int[elements];

for (int i = 0; i < elements; i++){

A[i] = i;

B[i] = i;

}

try{

// Query for platforms

std::vector platforms;

cl::Platform::get(&platforms);

// Get a list of devices on this platform

std::vector<cl::Device> devices;

platforms[0].getDevices(CL\_DEVICE\_TYPE\_ALL, &devices);

// Create a context for the devices

cl::Context context(devices);

// Create a command-queue for the first device

cl::CommandQueue queue = cl::CommandQueue(context, devices[0]);

// Create the memory buffers

cl::Buffer bufferA = cl::Buffer(context, CL\_MEM\_READ\_ONLY, datasize);

cl::Buffer bufferB = cl::Buffer(context, CL\_MEM\_READ\_ONLY, datasize);

cl::Buffer bufferC = cl::Buffer(context, CL\_MEM\_WRITE\_ONLY, datasize);

// Copy the input data to the input buffers using the

// command-queue for the first device

queue.enqueueWriteBuffer(bufferA, CL\_TRUE, 0, datasize, A);

queue.enqueueWriteBuffer(bufferB, CL\_TRUE, 0, datasize, B);

// Read the program source

std::ifstream sourceFile("vector\_add\_kernel.cl");

std::string sourceCode(std::istreambuf\_iterator<char>(sourceFile), (std::istreambuf\_iterator<char>()));

cl::Program::Source source(1, std::make\_pair(sourceCode.c\_str(), sourceCode.length() + 1);

// Create the program from the source code

cl::Program program = cl::Program(context, source);

// Build the program for the devices

program.build(devices);

// Create the kernel

cl::Kernel vecadd\_kernel(program, "vecadd");

// Set the kernel arguments

vecadd\_kernel.setArg(0, bufferA);

vecadd\_kernel.setArg(1, bufferB);

vecadd\_kernel.setArg(2, bufferC);

// Execute the kernel

cl::NDRange gloabl(elements);

cl::NDRange local(256);

queue.enqueueNDRangeKernel(vecadd\_kernel, cl::NullRange, gloabl, local);

// Copy the output data back to the host

queue.enqueueReadBuffer(bufferC, CL\_TRUE, 0, datasize, C);

} catch(cl::Error error){

std::cout << error.what() << “(“ << error.err() << “)” << std::endl;

}

}

# 第4章 OpenCL案例

## OpenCL计算直方图

### 思路

1、在每个工作组中创建一份**局部积分图**。【局部内存中的数据，每个工作组中的所有工作项都可以共享访问。局部内存一般会分布在GPU的片上内存中，其访问速度要比访问全局内存快的多。】

2、当工作组完成局部积分图时，其会传递给全局内存，并使用原子加操作将对应位置上的数据原子加到全局内存中。不过，这种实现方式也有问题：**对局部内存的访问上存在条件竞争**。这里需要你对目标设备的架构有所了解。对于很多GPU来说，原子操作访问局部内存的效率很高。在AMD Radeon GPU上，原子单元位于片上暂存式存储器中。因此，局部内存上的原子操作的效率要比全局原子操作的效率高很多。

### 计算直方图的OpenCL内核代码

1、初始化局部直方图内的值为0 (第14行)

2、同步工作项，确保相应的数据全部更新完毕 (第23行)

3、计算局部直方图 (第26行)

4、再次同步工作项，确保相应的数据全部更新完毕 (第35行)

5、将局部直方图写入到全局内存中 (第39行)

define HIST\_BINS 256

kernel

void histogram(global int data, int numData, \_\_global int histogram){

\_\_local int localHistorgram[HIST\_BINS];

int lid = get\_local\_id(0);

int gid = get\_glaobal\_id(0);

/ Initialize local histogram to zero /

for (int i = lid; i < HIST\_BINS; i += get\_local\_size(0)){

localHistorgram[i] = 0;

}

/\* Wait nutil all work-items within

the work-group have completed their stores \*/

barrier(CLK\_LOCAL\_MEM\_FENCE);

/ Compute local histogram /

for (int i = gid; i < numData; i += get\_glaobal\_size(0)){

atomic\_add(&localHistorgram[data[i]], 1);

}

/\* Wait nutil all work-items within

the work-group have completed their stores \*/

barrier(CLK\_LOCAL\_MEM\_FENCE);

/\* Write the local histogram out to

the global histogram \*/

for (int i = lid; i < HIST\_BINS; i += get\_glaobal\_size(0)){

atomic\_add(&histogram[i], localHistorgram[i]);

}

}

当我们需要工作项需要访问不同的内存位置时，我们可以以工作项的唯一标识ID为基准，然后加上所有工作项的数量作为跨度(例如，工作组内以工作组中工作项的数量，计算对应工作项所要访问的局部内存位置。或以NDRange中的尺寸，访问全局内存)。

### 内存栅栏

能确保工作组中的所有工作项都要到达该栅栏处，只要有线程没有达到，已达到的线程就不能执行下面的操作。

**局部内存栅栏**就是用来保证所有工作项都到达栅栏处，以代表局部直方图更新完毕。

### clEnqueueFillBuffer()

对数据进行初始化。

该API类似于C中的memset()函数。

**buffer**参数就是要初始化的数组对象，具体的值由pattern指定。与memset()不同，pattern可以指定为任意的OpenCL支持类型，比如：标量、整型向量或浮点类型。

**pattern\_size**用来指定pattern所占空间。

**size**参数用来指定数组内初始化的字节数，其值必须是pattern\_size的整数倍。

**offset**参数用来指定数组起始初始化的位置或偏移。

格式：

cl\_int

clEnqueueFillBuffer(

cl\_command\_queue command\_queue,

cl\_mem **buffer**,

const void \***pattern**,

size\_t **offset**,

size\_t **size**,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event \*event\_wait\_list,

cl\_event \*event)

### 直方图统计的主机端代码

注意，check(cl\_int status)是用来检查之前执行命令的状态是否为CL\_SUCCESS。

/ System includes /

/ OpenCL includes /

/ Utility functions /

include “utils.h”

include “bmp\_utils.h”

static const int HIST\_BINS = 256;

int main(int argc, char \*argv[]){

/ Host data /

int hInputImage = NULL;

int hOutputHistogram = NULL;

/\* Allocate space for the input image and read the

data from disk /

int imageRows;

int imageCols;

hInputImage = readBmp(“../../Images/cat.bmp”, &imageRows, &imageCols);

const int imageElements = imageRows imageCols;

const size\_t imageSize = imageElements \* sizeof(int);

/ Allocate space for the histogram on the host /

const int histogramSize = HIST\_BINS sizeof(int);

hOutputHistogram = (int )malloc(histogramSize);

if (!hOutputHistogram){ exit(-1); }

/ Use this check the output of each API call /

cl\_int status;

/ Get the first platform /

cl\_platform\_id platform;

status = clGetPlatformIDs(1, &platform, NULL);

check(status);

/ Get the first device /

cl\_device\_id device;

status = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_GPU, 1, &device, NULL);

check(status);

/ Create a command-queue and associate it with the device /

cl\_command\_queue cmdQueue;

context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);

check(status);

/ Create a buffer object for the output histogram /

cl\_mem bufOutputHistogram;

bufOutputHistogram = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, histogramSize, NULL, &status);

check(status);

/ Write the input image to the device /

status = clEnqueueWriteBuffer(cmdQueue, bufInputImage, CL\_TRUE, 0, imageSize, hInputImage, 0, NULL, NULL);

check(status);

/ Initialize the output histogram with zero /

int zero = 0;

status = clEnqueueFillBuffer(cmdQueue, bufOutputHistogram, &zero, sizeof(int), 0, histogramSize, 0, NULL, NULL);

check(status);

/ Create a program with source code /

char programSource = readFile(“histogram.cl”);

size\_t prograSourceLen = strlen(programSource);

cl\_program program = clCreateProgramWithSouce(context, 1, (const char \*)&programSource, &prograSourceLen, &status);

check(status);

/ Build (compile) the program for the device /

status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);

if (status != CL\_SUCCESS){

printCompilerError(program, device);

exit(-1);

}

/ Create the kernel /

cl\_kernel kernel;

kernel = clCreateKernel(program, “histogram”, &status);

check(status);

/ Set the kernel arguments /

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), &bufInputImage);

status |= clSetKernelArg(kernel, 1, sizeof(int), &imageElements);

status |= clSetKernelArg(kernel, 2, sizeof(cl\_mem), &bufOutputHistogram);

/ Define the index space and work-group size /

size\_t globalWorkSize[1];

globalWorkSize[0] = 1024;

size\_t localWorkSize[1];

localWorkSize[0] = 64;

/ Enqueue the kernel for execution /

status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);

check(status);

/ Read the output histogram buffer to the host /

status = clEnqueuReadBuffer(cmdQueue, bufOutputHistogram, CL\_TRUE, 0, histogramSize, hOutputHistogram, 0, NULL, NULL);

check(status);

/ Free OpenCL resources /

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseCommandQueue(cmdQueue);

clReleaseMemObject(bufInputImage);

clReleaseMemObject(bufOutputHistogram);

clReleaseContext(context);

/ Free host resource /

free(hInputImage);

free(hOutputHistogram);

free(programSource);

return 0;

}

## OpenCL旋转图像

### 图像旋转内核

\_\_constant sampler\_t sampler =

CLK\_NORMALIZED\_COORDS\_FALSE |

CLK\_FILTER\_LINEAR |

CLK\_ADDRESS\_CLAMP;

kernel

void rotation( read\_only image2d\_t inputImage,

\_\_write\_only image2d\_t ouputImage,

int imageWidth,

int imageHeigt,

float theta)

{

/ Get global ID for ouput coordinates /

int x = get\_global\_id(0);

int y = get\_global\_id(1);

/ Compute image center /

float x0 = imageWidth / 2.0f;

float y0 = imageHeight / 2.0f;

/\* Compute the work-item’s location relative

to the image center \*/

int xprime = x - x0;

int yprime = y - y0;

/ Compute sine and cosine /

float sinTheta = sin(theta);

float cosTheta = cos(theta);

/ Compute the input location /

float2 readCoord;

readCoord.x = xprime cosTheta - yprime sinTheta + x0;

readCoord.y = xprime sinTheta + yprime cosTheta + y0;

/ Read the input image /

float value;

value = read\_imagef(inputImage, sampler, readCoord).x;

/ Write the output image /

write\_imagef(outputImage, (int2)(x, y), (float4)(value, 0.f, 0.f, 0.f));

}

### 图像采样器(sampler\_t sampler)

用来描述如何访问图像。采样器指定如何处理访问到的图像位置，比如，当访问到图像之外的区域，或是当访问到多个坐标时，不进行差值操作。

### CLK\_NORMALIZED\_COORDS\_FALSE

标识指定基于像素地址的寻址。

### CL\_ADDRESS\_CLAMP

用于处理跨边界访问寻址方式。CL\_ADDRESS\_CLAMP表示当访问到图像之外的区域，会将RGB三个通道的值设置成0，并且将A通道设置成1或0(由图像格式决定)。所以，超出范围的像素将会返回黑色。

### CLK\_FILTER\_NEAREST和CLK\_FILTER\_LINEAR

一种过滤模式。过滤模式将决定将如何返回图像所取到的值。选项CLK\_FILTER\_NEAREST只是**简单的返回离所提供左边最近的图像元素**。

或者使用**CLK\_FILTER\_LINEAR**将**坐标附近的像素进行线性差值**。

### 为旋转例程创建图像对象

/\* The image descriptor describes how the data will be stored in memory. This descriptor initializes a 2D image with no pitch \*/

cl\_image\_desc desc;

desc.image\_type = CL\_MEM\_OBJECT\_IMAGE2D;

desc.image\_width = width;

desc.image\_height = height;

desc.image\_depth = 0;

desc.image\_array\_size = 0;

desc.image\_row\_pitch = 0;

desc.image\_slice\_pitch = 0;

desc.num\_mip\_levels = 0;

desc.num\_samples = 0;

desc.buffer = NULL;

/ The image format descibes the properties of each pixel /

cl\_image\_format format;

format.image\_channel\_order = CL\_R; // single channel

format.image\_channel\_data\_type = CL\_FLOAT;

/\* Create the input image and initialize it using a pointer to the image data on the host \*/

cl\_mem inputImage = clCreateImage(context, CL\_MEM\_READ\_ONLY, &format, &desc, NULL, NULL);

/ Create the output image /

cl\_mem ouputImage = clCreateImage(context, CL\_MEM\_WRITE\_ONLY, &formatm, &desc, NULL, NULL);

/ Copy the host image data to the device /

size\_t origin[3] = {0,0,0}; // Offset within the image to copy form

size\_t region[3] = {width, height, 1}; // Elements to per dimension

clEnqueueWriteImage(queue, inputImage, CL\_TRUE,

origin, region, 0 / row-pitch /, 0 / slice-pitch /,

hostInputImage, 0, NULL, NULL);

### 图像旋转主机端的完整代码

/ System includes /

/ OpenCL includes /

/ Utility functions /

include “utils.h”

include “bmp-utils.h”

int main(int argc, char \*argv)

{

/ Host data /

float hInputImage = NULL;

float \*hOutputImage = NULL;

/ Angle for rotation (degrees) /

const float theta = 45.f;

/\* Allocate space for the input image and read the

data from disk /

int imageRows;

int imageCols;

hInputImage = readBmpFloat(“cat.bmp”, &imageRow, &imageCols);

const int imageElements = imageRows imageCols;

const size\_t imageSize = imageElements \* sizeof(float);

/ Allocate space for the ouput image /

hOutputImage = (float \*)malloc(imageSize);

if (!hOutputImage){ exit(-1); }

/ Use this to check the output of each API call /

cl\_int status;

/ Get the first platform /

cl\_platform\_id platform;

status = clGetPlatformIDs(1, &platform, NULL);

check(status);

/ Get the first device /

cl\_device\_id device;

status = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_GPU, 1, &device, NULL);

check(status);

/ Create a context and associate it with the device /

cl\_context context;

context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);

check(status);

/ Create a command-queue and associate it with the device /

cl\_command\_queue cmdQueue;

cmdQueue = clCreateCommandQueue(context, device, 0, &status);

check(status);

/\* The image descriptor describes how the data will be stored

in memory. This descriptor initializes a 2D image with no pitch \*/

cl\_image\_desc desc;

desc.image\_type = CL\_MEM\_OBJECT\_IMAGE2D;

desc.image\_width = width;

desc.image\_height = height;

desc.image\_depth = 0;

desc.image\_array\_size = 0;

desc.image\_row\_pitch = 0;

desc.image\_slice\_pitch = 0;

desc.num\_mip\_levels = 0;

desc.num\_samples = 0;

desc.buffer = NULL;

/ The image format describes the properties of each pixel /

cl\_image\_format format;

format.image\_channel\_order = CL\_R; // single channel

format.image\_channel\_data\_type = CL\_FLOAT;

/\* Create the input image and initialize it using a

pointer to the image data on the host \*/

cl\_mem inputImage = clCreateImage(context, CL\_MEM\_READ\_ONLY, &format, &desc, NULL, NULL);

/ Create the ouput image /

cl\_mem outputImage = clCreateImage(context, CL\_MEM\_WRITE\_ONLY, &format, &desc, NULL, NULL);

/ Copy the host image data to the device /

size\_t origin[3] = {0,0,0}; // Offset within the image to copy from

size\_t region[3] = {imageCols, imageRows, 1}; // Elements to per dimension

clEnqueueWriteImage(cmdQueue, inputImage, CL\_TRUE,

origin, region, 0 / row-pitch /, 0 / slice-pitch /, hInputImage, 0, NULL, NULL);

/ Create a program with source code /

char programSource = readFile(“image-rotation.cl”);

size\_t programSourceLen = strlen(programSource);

cl\_program program = clCreateProgramWithSource(context, 1, (const char \*)&programSource, &programSourceLen, &status);

check(status);

/ Build (compile) the program for the device /

status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);

if (status != CL\_SUCCESS){

printCompilerError(program, device);

exit(-1);

}

/ Create the kernel /

cl\_kernel kernel;

kernel = clCreateKernel(program, “rotation”, &status);

check(status);

/ Set the kernel arguments /

status = clSetkernelArg(kernel, 0, sizeof(cl\_mem), &inputImage);

status |= clSetKernelArg(kernel, 1, sizeof(cl\_mem), &outputImage);

status |= clSetKernelArg(kernel, 2, sizeof(int), &imageCols);

status |= clSetKernelArg(kernel, 3, sizeof(int), &imageRows);

status |= clSetKernelArg(kernel, 4, sizeof(float), &theta);

check(status);

/ Define the index space and work-group size /

size\_t globalWorkSize[2];

globalWorkSize[0] = imageCols;

globalWorkSize[1] = imageRows;

size\_t localWorkSize[2];

localWorkSize[0] = 8;

localWorkSize[1] = 8;

/ Enqueue the kernel for execution /

status = clEnqueueReadImage(cmdQueue, outputImage, CL\_TRUE, origin, region, 0 / row-pitch /, 0 / slice-pitch /, hOutputImage, 0, NULL, NULL);

check(status);

/ Write the output image to file /

writeBmpFloat(hOutputImage, “rotated-cat.bmp”, imageRows, imageCols, “cat.bmp”);

/ Free OpenCL resources /

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseCommandQueue(cmdQueue);

clReleaseMemObject(inputImage);

clReleaseMemObject(outputImage);

clReleaseContext(context);

/ Free host resources /

free(hInputImage);

free(hOutputImage);

free(programSource);

return 0;

}

OpenCL图像卷积

使用OpenCL C实现的图像卷积

kerenl

void convolution( read\_only image2d\_t inputImage,

write\_only image2d\_t outputImage,

int rows,

int cols, consetant float filter,

int filterWidth,

sampler\_t sampler)

{

/ Store each work-item’s unique row and column \*/

int column = get\_global\_id(0);

int row = get\_global\_id(1);

/\* Half the width of the filter is needed for indexing

memory later \*/

int halfWidth = (int)(filterWidth / 2);

/\* All accesses to images return data as four-element vectors

(i.e., float4), although only the x component will contain

meaningful data in this code \*/

float4 sum = {0.0f,0.0f,0.0f,0.0f};

/ Iterator for the filter /

int filterIdx = 0;

/\* Each work-item iterates around its local area on the basis of the

size of the filter\*/

int2 coords; // Coordinates for accessing the image

/ Iterate the filter rows/

for (int i = -halfWidth; i <= halfWidth; i++)

{

coords.y = row + i;

/ Iterate over the filter columns /

for (int j = -halfWidth; j <= halfWidth; j++)

{

coords.x - column + 1;

/\*Read a pixel from the image. A single-channel image

stores the pixel in the x coordinate of the reatured

vector. \*/

pixel = read\_imagef(inputImage, sampler, coords);

sum.x += pixel.x filter[filterIdx++];

}

}

/ Copy the data to the output image /

coords.x = column;

coords.y = row;

write\_imagef(outputImage, coords, sum);

}

### 主机端创建采样器

cl\_sampler clCreateSampler(

cl\_context context,

cl\_bool normalized\_coords,

cl\_addressing\_mode addressing\_mode,

cl\_filter\_mode filter\_mode,

cl\_int \*errcode\_ret)

**C++ API:**

cl::Sampler::Sampler(

const Context &context,

cl\_bool normalized\_coords,

cl\_addressing\_mode addressing\_mode,

cl\_filter\_mode filter\_mode,

cl\_int \*err = NULL)

**使用C++创建采样器:**

cl::Sampler sampler = new cl::Sampler(context, CL\_FALSE, CL\_ADDRESS\_CLAMP\_TO\_EDGE, CL\_FILTER\_NEAREST);

### 图像卷积主机端完整代码

define \_\_CL\_ENABLE\_EXCEPTIONS

include “utils.h”

include “bmp\_utils.h”

static const char \*inputImagePath = “../../Images/cat.bmp”;

static float gaussianBlurFilter[25] = {

1.0f / 273.0f, 4.0f / 273.0f, 7.0f / 273.0f, 4.0f / 273.0f, 1.0f / 273.0f,

4.0f / 273.0f, 16.0f / 273.0f, 26.0f / 273.0f, 16.0f / 273.0f, 4.0f / 273.0f,

7.0f / 273.0f, 26.0f / 273.0f, 41.0f / 273.0f, 26.0f / 273.0f, 7.0f / 273.0f,

4.0f / 273.0f, 16.0f / 273.0f, 26.0f / 273.0f, 16.0f / 273.0f, 4.0f / 273.0f,

1.0f / 273.0f, 4.0f / 273.0f, 7.0f / 273.0f, 4.0f / 273.0f, 1.0f / 273.0f

};

static const int gaussianBlurFilterWidth = 5;

int main()

{

float hInputImage;

float hOutpueImage;

int imageRows;

int imageCols;

/ Set the filter here /

int filterWidth = gaussianBlurFilterWidth;

float \*filter = gaussianBlurFilter;

/ Read in the BMP image /

hInputImage = readBmpFloat(inputImagePath, &imageRows, &imageCols);

/ Allocate space for the output image /

hOutputImage = new float[imageRows \* imageCols];

try

{

/ Query for platforms /

std::vector platforms;

cl::Platform::get(&platdorms);

/\* Get a list of devices on this platform \*/

std::vector<cl::Device> device;

platforms[0].getDevices(CL\_DEVICE\_TYPE\_GPU, &devices);

/\* Create a context for the devices \*/

cl::Context context(devices);

/\* Create a command-queue for the first device \*/

cl::CommandQueueu queue = cl::CommandQueue(context, devices[0]);

/\* Create the images \*/

cl::ImageFormat imageFormat = cl::ImageFormat(CL\_R, CL\_FLOAT);

cl::Image2D inputImage = cl::Image2D(context, CL\_MEM\_READ\_ONLY,

imageFormat, imageCols, imageRows);

cl::Image2D outputImage = cl::Image2D(context,

CL\_MEM\_WRITE\_ONLY,

imageFormat, imageCols, imageRows);

/\* Create a buffer for the filter \*/

cl::Buffer filterBuffer = cl::Buffer(context, CL\_MEM\_READ\_ONLY,

filterWidth \* filterWidth \* sizeof(float));

/\* Copy the input data to the input image \*/

cl::size<3> origin;

origin[0] = 0;

origin[1] = 0;

origin[2] = 0;

cl::size<3> region;

region[0] = 0;

region[1] = 0;

region[2] = 0;

queue.enqueueWriteImage(inputImage, CL\_TRUE, origin, region,

0, 0,

hInputImage);

/\* Copy the filter to the buffer\*/

queue.enqueueWriteBuffer(filterBuffer, CL\_TRUE, 0,

filterWidth \* filterWidth \* sizeof(float), filter);

/\* Create the sampler \*/

cl::Sampler sampler = cl::Sampler(context, CL\_FALSE,

CL\_ADDRESS\_CLAMP\_TO\_EDGE, CL\_FILTER\_NEAREST);

/\* Read the program source \*/

std::ifstream sourceFile("image-convolution.cl");

std::string sourceCode(std::istreambuf\_iterator<char>(sourceFile),

(std::istreambuf\_iterator<char>()));

cl::Program::Source source(1,

std::make\_pair(sourceCode.c\_str(),

sourceCode.length() + 1));

/\* Make program form the source code \*/

cl::Program program = cl::Program(context, source);

/\* Create the kernel \*/

cl::Kernel kernel(program, "convolution");

/\* Set the kernel arguments \*/

kernel.setArg(0, inputImage);

kernel.setArg(1, ouputImage);

kernel.setArg(2, filterBuffer);

kernel.setArg(3, filterWIdth);

kernel.setArg(4, sampler);

/\* Execute the kernel \*/

cl::NDRange global(imageCols, imageRows);

cl::NDRange local(8, 8);

queue.enqueueNDRangeKernel(kernel, cl::NullRange, global,

local);

/\* Copy the output data back to the host \*/

queue.enqueueReadImage(outputImage, CL\_TRUE, origin, region,

0, 0,

hOutputImage);

/\* Save the output BMP image \*/

writeBmpFloat(hOutputImage, "cat-filtered.bmp", imageRows, imageCols, inputImagePath);

}

catch(cl::Error error){

std::cout << error.what() << “(“ << error.err() << “)” << std::endl;

}

free(hInputImage);

delete hOutputImage;

return 0;

}

## 生产者-消费者

### 管道

**管道内存中的数据(称为packets)**组织为**先入先出(FIFO)**结构。

管道对象的内存在**全局内存**上开辟，所以可以被多个内核同时访问。这里需要注意的是，管道上存储的数据，主机端无法访问。

内核中管道属性可能是**只读(read\_only)**或**只写(write\_only)**，不过**不能是读写**。如果管道对象没有指定是只读或只写，那么编译器将**默认其为只读**。

管道在内核的参数列表中，通过使用**关键字pipe**进行声明，后跟**数据访问类型和数据包的数据类型**。

### 卷积内核(生产者)

\_\_constant sampler\_t sampler =

CLK\_NORMALIZED\_COORDS\_FALSE |

CLK\_FILTER\_NEAREST |

CLK\_ADDRESS\_CLAMP\_TO\_EDGE;

kernel

void producerKernel(

image2d\_t read\_only inputImage,

pipe write\_only float \*outputPipe, constant float filter,

int filterWidth)

{

/ Store each work-item’s unique row and column \*/

int column = get\_global\_id(0);

int row = get\_global\_id(1);

/\* Half the width of the filter is needed for indexing memory later\*/

int halfWidth = (int)(filterWidth / 2);

/ Used to hold the value of the output pixel /

float sum = 0.0f;

/ Iterator for the filter /

int filterIdx = 0;

/\* Each work-item iterates around its local area on the basis of the size of the filter \*/

int2 coords; // Coordinates for accessing the image

/ Iterate the filter rows /

for (int i = -halfWidth; i <= halfWidth; i++)

{

coords.y = row + i;

/ Iterate over the filter columns /

for (int j = -halfWidth; j <= halfWidth; j++)

{

coords.x = column + j;

/\* Read a pixel from the image. A single channel image

stores the pixel in the x coordinate of the returned

vector. /

float4 pixel;

pixel = read\_imagef(inputImage, sampler, coords);

sum += pixel.x filter[filterIdx++];

}

}

/ Write the output pixel to the pipe /

write\_pipe(outputPipe, &sum);

}

### 卷积内核(消费者)

kernel

void consumerKernel(

pipe read\_only float inputPipe,

int totalPixels,

\_\_global int histogram)

{

int pixelCnt;

float pixel;

/ Loop to process all pixels from the producer kernel /

for (pixelCnt = 0; pixelCnt < totalPixels; pixelCnt++)

{

/\* Keep trying to read a pixel from the pipe until one becomes available \*/

while(read\_pipe(inputPipe, &pixel));

/\* Add the pixel value to the histogram \*/

histogram[(int)pixel]++;

}

}

### 创建管道对象

cl\_pipe clCreatePipe(

cl\_context context,

cl\_mem\_flags flags,

cl\_uint pipe\_packet\_size,

cl\_uint pipe\_max\_packets,

const cl\_pipe\_properties \*properties,

cl\_int \*errcode\_ret)

cl\_mem pipe = clCreatepipe(context, 0, sizeof(float), imageRows \* imageCols, NULL, &status);

### 生产者-消费者主机端完整代码

/ System includes /

/ OpenCL includes /

/ Utility functions /

include “utils.h”

include “bmp-utils.h”

/ Filter for the convolution /

static float gaussianBlurFilter[25] = {

1.0f / 273.0f, 4.0f / 273.0f, 7.0f / 273.0f, 4.0f / 273.0f, 1.0f / 273.0f,

4.0f / 273.0f, 16.0f / 273.0f, 26.0f / 273.0f, 16.0f / 273.0f, 4.0f / 273.0f,

7.0f / 273.0f, 26.0f / 273.0f, 41.0f / 273.0f, 26.0f / 273.0f, 7.0f / 273.0f,

4.0f / 273.0f, 16.0f / 273.0f, 26.0f / 273.0f, 16.0f / 273.0f, 4.0f / 273.0f,

1.0f / 273.0f, 4.0f / 273.0f, 7.0f / 273.0f, 4.0f / 273.0f, 1.0f / 273.0f

};

static const int filterWidth = 5;

static const int filterSize = 25 \* sizeof(float);

/ Number of histogram bins /

static const int HIST\_BINS = 256;

int main(int argc, char argv[])

{

/ Host data /

float hInputImage = NULL;

int \*hOutputHistogram = NULL;

/\* Allocate space for the input image and read the data from dist /

int imageRows;

int imageCols;

hInputImage = readBmpFloat(“../../Images/cat.bmp”, &imageRows, &imageCols);

const int imageElements = imageRows imageCols;

const size\_t imageSize = imageElements \* sizeof(float);

/ Allocate space for the histogram on the host /

const int histogramSize = HIST\_BINS sizeof(int);

hOutputHistogram = (int )malloc(histogramSize);

if (!hOutputHistogram){ exit(-1); }

/ Use this to check the output of each API call /

cl\_int status;

/ Get the first platform /

cl\_platform\_id platform;

status = clGetPlatformIDs(1, &platform, NULL);

check(status);

/ Get the devices /

cl\_device\_id devices[2];

cl\_device\_id gpuDevice;

cl\_device\_id cpuDevice;

status = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_CPU, 1, &gpuDevice, NULL);

status = clGetDeviceIDs(platform, CL\_DEVICE\_TYPE\_GPU, 1, &cpuDevice, NULL);

check(status);

devices[0] = gpuDevice;

devices[1] = cpuDevice;

/ Create a context and associate it with the devices /

cl\_context context;

context = clCreateContext(NULL, 2, devices, NULL, NULL, &status);

check(status);

/ Create the command-queues /

cl\_command\_queue gpuQueue;

cl\_command\_queue cpuQueue;

gpuQueue = clCreateCommandQueue(context, gpuDevice, 0, &status);

check(status);

cpuQueue = clCreateCommandQueue(context, cpuDevice, 0, &status);

check(status);

/\* The image desriptor describes how the data will be stored in memory. This descriptor initializes a 2D image with no pitch\*/

cl\_image\_desc desc;

desc.image\_type = CL\_MEM\_OBJECT\_IMAGE2D;

desc.image\_width = imageCols;

desc.image\_height = imageRows;

desc.image\_depth = 0;

desc.image\_array\_size = 0;

desc.image\_row\_pitch = 0;

desc.image\_slice\_pitch = 0;

desc.num\_mip\_levels = 0;

desc.num\_samples = 0;

desc.buffer = NULL;

/ The image format descibes the properties of each pixel /

cl\_image\_format format;

format.image\_channel\_order = CL\_R; // single channel

format.image\_channel\_data\_type = CL\_FLOAT;

/\* Create the input image and initialize it using a

pointer to the image data on the host. \*/

cl\_mem inputImage;

inputImage = clCreateImage(context, CL\_MEM\_READ\_ONLY, &format, &desc, NULL, NULL);

/ Create a buffer object for the ouput histogram /

cl\_mem ouputHistogram;

outputHisrogram = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, &format, &desc, NULL, NULL);

/ Create a buffer for the filter /

cl\_mem filter;

filter = clCreateBuffer(context, cl\_MEM\_READ\_ONLY, filterSize, NULL, &status);

check(status);

cl\_mem pipe;

pipe = clCreatePipe(context, 0, sizeof(float), imageRows \* imageCols, NULL, &status);

/ Copy the host image data to the GPU /

size\_t origin[3] = {0,0,0}; // Offset within the image to copy from

size\_t region[3] = {imageCols, imageRows, 1}; // Elements to per dimension

status = clEnqueueWriteImage(gpuQueue, inputImage, CL\_TRUE, origin, region, 0, 0, hInputImage, 0, NULL, NULL);

check(status);

/ Write the filter to the GPU /

status = clEnqueueWriteBuffer(gpuQueue, filter, CL\_TRUE, 0, filterSize, gaussianBlurFilter, 0, NULL, NULL);

check(status);

/ Initialize the output istogram with zeros /

int zero = 0;

status = clEnqueueFillBuffer(cpuQueue, outputHistogram, &zero, sizeof(int), 0, histogramSize, 0, NULL, NULL);

check(status);

/ Create a program with source code /

char programSource = readFile(“producer-consumer.cl”);

size\_t programSourceSize = strlen(programSource);

cl\_program program = clCreateProgramWithSource(context, 1, (const char\*)&programSource, &programSourceLen, &status);

check(status);

/ Build (compile) the program for the devices /

status = clBuildProgram(program, 2, devices, NULL, NULL, NULL);

if (status != CL\_SUCCESS)

{

printCompilerError(program, gpuDevice);

exit(-1);

}

/ Create the kernel /

cl\_kernel producerKernel;

cl\_kernel consumerKernel;

producerKernel = clCreateKernel(program, “producerKernel”, &status);

check(status);

consumerKernel = clCreateKernel(program, “consumerKernel”, &status);

check(status);

/ Set the kernel arguments /

status = clSetKernelArg(producerKernel, 0, sizeof(cl\_mem), &inputImage);

status |= clSetKernelArg(producerKernel, 1, sizeof(cl\_mem), &pipe);

status |= clSetKernelArg(producerKernel, 2, sizeof(int), &filterWidth);

check(status);

status |= clSetKernelArg(consumerKernel, 0, sizeof(cl\_mem), &pipe);

status |= clSetKernelArg(consumerKernel, 1, sizeof(int), &imageElements);

status |= clSetKernelArg(consumerKernel, 2, sizeof(cl\_mem), &outputHistogram);

check(status);

/ Define the index space and work-group size /

size\_t producerGlobalSize[2];

producerGlobalSize[0] = imageCols;

producerGlobalSize[1] = imageRows;

size\_t producerLocalSize[2];

producerLocalSize[0] = 8;

producerLocalSize[1] = 8;

size\_t consumerGlobalSize[1];

consumerGlobalSize[0] = 1;

size\_t consumerLocalSize[1];

consumerLocalSize[0] = 1;

/ Enqueue the kernels for execution /

status = clEnqueueNDRangeKernel(gpuQueue, producerKernel, 2, NULL, producerGlobalSize, producerLocalSize, 0, NULL, NULL);

status = clEnqueueNDRangeKernel(cpuQueue, consumerKernel, 2, NULL, consumerGlobalSize, consumerLocalSize, 0, NULL, NULL);

/ Read the output histogram buffer to the host /

status = clEnqueueReadBuffer(cpuQueue, outputHistogram, CL\_TRUE, 0, histogramSize, hOutputHistogram, 0, NULL, NULL);

check(status);

/ Free OpenCL resources /

clReleaseKernel(producerKernel);

clReleaseKernel(consumerKernel);

clReleaseProgram(program);

clReleaseCommandQueue(gpuQueue);

clReleaseCommandQueue(cpuQueue);

clReleaseMemObject(inputImage);

clReleaseMemObject(outputHistogram);

clReleaseMemObject(filter);

clReleaseMemObject(pipe);

clReleaseContext(context);

/ Free host resources /

free(hInputImage);

free(hOutputHistogram);

free(programSource);

return 0;

}

## 打印编译错误信息

### clGetProgramBuildInfo()

当OpenCL程序对象编译失败，一个构建日志将会产生，并保存在程序对象中。该日志可以通过API clGetProgramBuildInfo()检索出，传入CL\_PROGRAM\_BUILD\_LOG到param\_name，得到相应日志内容。

### clProgramBuildInfo()

与clGetProgramBuildInfo()类似的API，clProgramBuildInfo()需要调用两次：第一次是获取日志的大小，分配对应大小的数组，用来放置日志内容；第二次是将日志中的具体内容取出。

### 查询程序对象编译日志的函数封装

void printCompilerError(cl\_program program, cl\_device\_id device)

{

cl\_int status;

size\_t logSize;

char \*log;

/ Get the log size /

status = clGetProgramBuildInfo(program, device, CL\_PROGRAM\_BUILD\_LOG, 0, NULL, &logSize);

check(status);

/ Allocate space for the log /

log = (char \*)malloc(logSize);

if (!log){

exit(-1);

}

/ Read the log /

status = clGetPeogramBuildInfo(program, device, CL\_PROGRAM\_BUILD\_LOG, logSize, log, NULL);

check(status);

/ Print the log /

printf(“%s\n”, log);

}

### 创建一个程序字符串

char readFile(const char filename)

{

FILE fp;

char fileData;

long fileSize;

/ Open the file /

fp = fopen(filename, “r”);

if (!fp){

printf(“Could not open file: %s\n”, filename);

exit(-1);

}

/ Determine the file size /

if (fseek(fp, 0, SEEK\_END)){

printf(“Error read the file\n”);

exit(-1);

}

fileSize = ftell(fp);

if (fileSize < 0){

printf(“Error read the file\n”);

exit(-1);

}

if (fseek(fp, 0, SEEK\_SET)){

printf(“Error read the file\n”);

exit(-1);

}

/ Read the contents /

fileData = (char \*)malloc(fileSize + 1);

if (!fileData){

exit(-1);

}

if (fread(fileData, fileSize, 1, fp) != 1){

printf(“Error reading the file\n”);

exit(-1);

}

/ Terminate the string /

fileData[fileSize] = ‘\0’;

/ Close the file /

if (fclose(fp)){

printf(“Error closing the file\n”);

exit(-1);

}

return fileData;

}

# 第5章 OpenCL运行时和并发模型

### OpenCL命令队列的主要同步点

1、使用指定OpenCL时间等待某个命令完成

2、调用clFinish()函数，将阻塞主机端的线程，直到命令队列中的命令全部执行完毕

3、执行一次阻塞式内存操作

### clEnqueueReadBuffer()中的blocking\_read

控制是否使用阻塞方式进行数据传输。

将数据从设备端获取，或是传输到设备端，通常都会使用同步的方式进行内存操作。因此，blocking\_read参数可以设置成**CL\_TRUE**，用以阻塞主线程，直到数据传输完成主线程才能继续进行下面的操作。使用这种方式进行同步，可以直接获取数据，之后就不需要在进行额外的同步了。

格式：

cl\_int

clEnqueueReadBuffer(

cl\_command\_queue command\_queue,

cl\_mem buffer,

cl\_bool blocking\_read,

size\_t offset,

size\_t size,

const void \*ptr,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event \*event\_wait\_list,

cl\_event \*event)

### 事件命令状态

1、**已入队(Queued)**：该命令已经在命令队列中占据了一席之地

2、**已提交(Submitted)**：该命令已经从命令队列中移除，并提交给设备执行

3、**已就绪(Ready)**：该命令已经准备好在设备端执行

4、**已运行(Running)**：该命令已经在设备端执行，但并未完成

5、**已完成(Ended)**：该命令已经在设备端执行完成

6、**已结束(Complete)**：该命令以及其子命令都已经执行完成

### clGetEventInfo()

查询命令相关错误码，将CL\_EVENT\_COMMAND\_STATUS\_EXECUTION\_STATUS传递给param\_name参数即可。

格式：

cl\_int

clGetEventInfo(

cl\_event event,

cl\_event\_info param\_name,

size\_t param\_value\_size，

void \*param\_value,

size\_t \*param\_value\_size\_ret)

### clWaitForEvents()

阻塞主机端的执行，直到指定的事件对象链表上相关的命令全部解除，才会解除对主机主线程的阻塞。

格式：

cl\_int

clWaitForEvent(

cl\_uint num\_events,

const cl\_event \*event\_list)

### clEnqueueBarrierWithWaitList()

入队栅栏，可将命令队列列表当做参数输入。如果没有提供命令队列，栅栏会等待之前入队的所有命令完成(后续入队的命令将不会执行，直到前面所有的命令执行完成)。

### clEnqueueMarkerWithWaitList()

入队标识(Marker)。

栅栏和标识的区别在于，**标识不会阻塞之后入队的命令**。因此，当设备完成所有指定事件时，标识就允许编程者去查询指定事件状态，而不会阻碍其中一些命令的执行。

### clSetEventCallback()

**为事件对象注册回调函数**。当事件对象到达某一指定状态时，回调函数便会调用。**command\_exec\_callback\_type**用来指定**回调函数在何时调用**。可能的参数只有：CL\_SUBMITTED，CL\_RUNNING和CL\_COMPLETE。

格式：

cl\_int

clSetEventCallback(

cl\_event event,

cl\_int command\_exec\_callback\_type,

void (CL\_CALLBACK \*pfn\_event\_notify)(

cl\_event event,

cl\_int event\_command\_exec\_status,

void \*user\_data),

void \*user\_data)

### 开启命令队列的计时功能

在创建命令队列时，将**CL\_QUEUE\_PROFILING\_ENABLE**加入**properties**参数内，提供给**clCreateCommandQueueWithProperties()**

### clGetEventProfilingInof()

获取与命令相关的时间对象的性能信息。

为了**确定命令整体耗时情况**，可以将**CL\_PROFILING\_COMMAND\_START**和**CL\_PROFILING\_COMMAND\_END**作为实参传入**param\_name**中。如果需要**将子内核执行的时间计算在内**，就需要传递**CL\_PROFILING\_COMMAND\_COMPLETE**作为实参。

OpenCL定义的计时器，其计时的精度必须是**纳秒**级别。

格式：

cl\_int

clGetEventProfilingInfo(

cl\_event event,

cl\_profiling\_info param\_name,

size\_t param\_value\_size,

void \*param\_value,

size\_t \*param\_value\_size\_ret)

### clCreateUserEvent()

创建用户事件的对象。

格式：

cl\_event

clCreateUserEvent(

cl\_context context,

cl\_int \*errcode\_ret)

### clSetUeserEventStatus()

**设置用户事件的状态**。

execution\_status参数指定需要设置的新执行状态。用户事件对象可以将状态设置为两种，第一种可为**CL\_COMPLETE**；第二种可为**一个负数，表示一个错误**。设置成负值意味着**所有入队的命令，需要等待该用户事件终止才能退出**。

clSetUserEventStatus()**只能对一个用户事件对象使用一次**，也就是**对一个对象只能设置一次执行状态**。

格式：

cl\_int clSetUserEventStatus(

cl\_event event,

cl\_int execution\_status)

### 设置乱序的标志位

创建命令队列的API(clCreateCommandQueueWithProperties())可以设置乱序的标志位。只要属性参数中包含**CL\_QUEUE\_OUT\_OF\_ORDER\_EXEC\_MODE\_ENABLE**就可以产生支持乱序执行的命令队列。

支持乱序的队列需要**创建在支持乱序执行的设备上**。

乱序命令队列并不保证其执行顺序就是乱序的。鲁棒性较好的应该避免死锁的发生，当意识到有死锁发生的可能，那最好的方式就是**串行执行入队的命令**。

### OpenCL同步事件对象

使用OpenCL事件对象进行同步时，只能针对**同一上下文对象中的命令**。如果是不同的上下文中的设备，那么事件对象的同步功能将会失效，并且要在这种情况下共享数据，就需要在两个设备之间进行**显式的拷贝**。

### OpenCL下的多设备编程(使用异构设备进行并行编程)

1、**流水执行**：两个或多个设备以流水方式工作，这样就需要设备间互相等待结果。

2、**单独执行**：这种方式就是每个设备各自做各自的任务，每个设备间的任务并无相关性。

### clEnqueueNDRangeKernel()

内核的执行需要运行时提供对应的启动/调度接口，该函数就是clEnqueueNDRangeKernel()。

内核在调度中会产生大量**工作项**，共同执行内核“函数体”。

使用启动接口之后，就会产生一个**NDRange**，其包括了**内核执行的维度**，以及**每个维度上的工作项的数量**。一个NDRange可以定义为**1，2和3维**，用于规划处工作项的“格子图”，工作项简单和直接的结构非常适合并行执行。

将OpenCL模型映射到硬件端，**每个工作项都运行在一个硬件单元上**，这个单元称为**“执行元素”(Processing element, PE)**。OpenCL内核执行时，可能有**多个**工作项**依次**工作在同一个PE上。

### 分歧

工作项也有可能在同一波或束中，走进不同的分支中，这时**硬件有责任将不该执行的分支结果舍弃**。这种情况就是众所周知的分歧(divergence)，这会极大**影响内核执行的效率**，因为工作项执行了冗余的操作，并且这些操作的结果最后都要舍弃。

### clGetKernelWorkGroupInfo()

**工作组尺寸查询**，需要**CL\_KERNEL\_PREFERRED\_WORK\_GROUP\_SIZE\_MULTIPLE**作为实参传入**param\_name**中。

### 获取工作项在执行区域内的具体位置

OpenCL定义了一些内置函数，可以在内核内部获取工作项在执行区域内的具体位置。这些函数通常都有一个维度参数，这里定义为**uint dimension**。其值可以为**0、1和2**，可以通过**维度的设置**获取**NDRange参数在入队时的设置**：

1、**uint get\_work\_dime()**:返回入队时所设定的工作维度数量

2、**size\_t get\_global\_size(uint dimension)**:返回对应维度上的全局工作项数量

3、**size\_t get\_global\_id(uint dimension)**:返回对应维度上对应全局工作项索引

4、**size\_t get\_local\_size(uint dimension)**:返回对应维度上工作组的数量。如果内核所分配的工作项在工作组内数量不均匀，那么剩余工作组(之后章节中讨论)将返回与均匀工作项不同的值

5、**size\_t get\_enqueued\_local\_size(uint dimension)**:返回对应维度上均匀区域内工作项数量

6、**size\_t get\_num\_groups(uint dimension):**返回当前工作项索引，以该工作项所属组的起始位置为偏移

7、**size\_t get\_group\_id(uint dimension)**:返回当前工作组的索引。也可通过获取该工作组的第一个元素的全局索引，然后除以工作组的大小得到当前工作组索引

### get\_local\_size()

返回工作组实际的大小

### get\_enqueue\_local\_size()

返回剩余的工作组大小。

### size\_t get\_global\_linear\_id()

返回工作项的全局线性索引

### size\_t get\_local\_linear\_id()

返回工作项的组内线性索引

### work\_group\_barrier()

让提前到达的线程等待未到达对应位置的线程，直到所有线程到达该位置，则继续进行下面的操作。不同的工作组中完成等待的时间都是相互独立的。

内置函数work\_group\_barrier()的参数列表如下所示：

void work\_group\_barrier(cl\_mem\_fence\_flag flags)

void work\_group\_barrier(cl\_mem\_fence\_flag flags, memory\_scope scope)

### flags可以设置成的实参

1、CLK\_LOCAL\_MEM\_FENCE：需要对于局部内存的所有访问组内可见

2、CLK\_GLOBAL\_MEM\_FENCE：需要对全局内存的所有访问组内可见

3、CLK\_IMAGE\_MEM\_FENCE：需要对图像的所有访问组内可见

### 三类工作组评估函数

1、谓词评估函数

2、广播函数

3、并行原语函数

### 谓词评估函数

谓词评估函数评估**工作组中的所有工作项**，如果满足相关的条件则返回一个非零值。

评估的工作组中有一个工作项满足条件，则函数work\_group\_any()返回一个非零值。当评估的工作组中所有工作项满足条件，则函数work\_group\_all()返回一个非零值。

格式：

int work\_group\_any(int predicate)

int work\_group\_all(int predicate)

### 广播函数

广播函数是**将一个工作项的数据传输给工作组内其他的工作项**。函数用坐标来标定对应的工作项，然后共享该工作项的数据x。该函数将广播返回值到每个工作项上。

函数参数在维度上进行了重载：

一维：gentype work\_group\_broadcast(gentype x, size\_t local\_id)

二维：gentype work\_group\_broadcast(gentype x, size\_t local\_id\_x, size\_t local\_id\_y)

三维：gentype work\_group\_broadcast(gentype x, size\_t local\_id\_x, size\_t local\_id\_y, size\_t local\_id\_z)

### 并行原语函数

OpenCL支持两类内置并行原语函数：**归约**和**扫描**。这个函数会用在很多并行应用上，这两个函数的实现都由设备供应商使用高性能代码提供，这样对于开发者来说会省去自己去优化的风险和工作量。

并行原语函数**无法保证浮点数操作的顺序**，所以该函数未与浮点数相关联。

格式：

gentype work\_group\_reduce\_<op>(gentype x)

gentype work\_group\_scan\_inclusive\_<op>(gentype x)

gentype work\_group\_scan\_exclusive\_<op>(gentype x)

函数中的<op>可以替换为add，min或max。这样就可以使用该函数找到局部数组的最大值，就如下面的代码所示：

// float max:

max = work\_group\_reduce\_max(local\_data[get\_local\_id(0)]);

### 闭扫描和开扫描

闭扫描和开扫描的不同在于，当前位置上的元素是否参与累加。

闭扫描版本生成的数组**会包含当前位置的元素**。

开扫描版本则不会包含当前位置的元素。

每个工作项都可用该函数，且会在工作组内返回相关线性索引的值。

### 原生内核

原生内核是一种回调的机制，其能更简洁的集成进OpenCL的执行模型中。原生内核允许使用传统编译器去编译C标准函数(与OpenCL不同)，并将编译好的C函数在放入OpenCL的任务执行图中，由事件来触发下一个事件。原生内核可以在一个设备上入队执行，并且与OpenCL内核共享内存对象。

原生内核与OpenCL内核的区别在于**设置参数方面**。原生内核使用对应的API进行入队(**clEnqueueNativeKernel()**)，将标准C函数通过指针的方式进行传入。参数列表需要连同其大小，打包传入设备。

格式：

cl\_int

clEnqueueNativeKernel(

cl\_command\_queue command\_queue,

void (CL\_CALLBACK \*user\_func)(void \*),

void \*args,

size\_t cb\_args,

cl\_uint num\_mem\_objects,

const cl\_mem \*mem\_list,

const void \*\*args\_mem\_loc,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event \*event\_wait\_list,

cl\_event \*event)

常规OpenCL内核可以将数组和图像作为参数传入，原生内核同样可以使用对应的数组和图像作为输入。OpenCL中向原生内核传递参数完成后，会通过一种方式进行解包。可以通过**设置mem\_list的实参**，向原生内核传入一个**内存对象链表**；**args\_mem\_loc参数**存储一个**指针链表**，用于存储解包之后的内存对象。

### 内置内核

内置内核与设备是捆绑关系，其也不会在运行时由源码进行构建。通常的内置内核会展示硬件对固定函数的加速能力，这个硬件可能是一种支持OpenCL的特殊嵌入式设备，或是自定义设备。

内置内核属于OpenCL定义之外的内容，因此内置内核实现的最总解释权还在硬件供应商那里。

### 嵌套并行机制(nested parallelism)

设备端任务队列的好处是启用了嵌套并行机制(nested parallelism)——一个并行程序中的某个线程，再去开启多个线程。嵌套并行机制常用于**不确定内部某个算法要使用多少个线程的应用**中。单独调用的方式，执行完任务后，线程的生命周期就结束了。嵌套并行将会在任务执行中间产生更多的线程，并在某个子任务完成后结束线程。

与单独使用分叉-连接(fork-join)机制相比，嵌套并行机制中产生的线程会在其任务完成后销毁。

### 设备端入队的好处

1、**内核可以在设备端直接入队**。这样就不需要同步，或与主机的交互，并且会隐式的减少数据传输

2、**更加自然的表达算法**。当算法包括递归，不规则循环结构，或其他单层并行是固定不均匀的，现在都可以在OpenCL中完美的实现

3、**更细粒度的并行调度，以及动态负载平衡**。设备能更好的相应数据驱动决策，以及适应动态负载

### enqueue\_kernel()

让子内核入队。每个调用该内置函数的工作项都会入队一个子内核。

格式：

int

enqueue\_kernel(

queue\_t queue,

kernel\_enqueue\_flags\_t flags,

const ndrange\_t ndrange,

void (^block)(void))

需要传递一个命令队列。**flags参数**用来执行子内核何时开始执行。该参数有三个语义可以选择：

1、**CLK\_ENQUEUE\_FLAGS\_NO\_WAIT**:子内核立刻执行

2、**CLK\_ENQUEUE\_FLAGS\_WAIT\_KERNEL**:子内核需要等到父内核到到ENDED点时执行。这就意味着子内核在设备上运行时，父内核已经执行完成

3、**CLK\_ENQUEUE\_FLAGS\_WAIT\_WORK\_GROUP**:子内核必须要等到入队的工作组执行完成后，才能执行。

enqueue\_kernel()也需要使用**NDRange**来**指定维度信息**(传递给ndrange参数)。与主机端调用一样，**全局偏移**和**工作组数量**是可选的。这时需要在内核上创建ndrange\_t类型的对象来执行执行单元的配置，这里使用到了一组内置函数：

ndrange\_t ndrange\_<N>D(const size\_t global\_work\_size[<N>])

ndrange\_t ndrange\_<N>D(const size\_t global\_work\_size[<N>], const size\_t global\_work\_size[<N>])

ndrange\_t ndrange\_<N>D(const size\_t global\_work\_size[<N>], const size\_t global\_work\_size[<N>], const size\_t local\_work\_size\_[<N>])

其中<N>可为1,2和3。

最终，enqueue\_kernel()的最后一个参数**block**，其为指定入队的内核。这里指定内核的方式称为**“Clang块”**。

如主机端API一样，enqueue\_kernel()会返回一个整数，代表其执行是否成功。返回**CLK\_SUCCESS**为成功，返回**CLK\_ENQUEUE\_FAILURE**则为失败。编程者想要了解失败入队的更多原因的话，需要在**clBuildProgram()传入”-g”参数**，或是**clCompileProgram()调用会启用细粒度错误报告**，会有更加具体的错误码返回，例如：CLK\_INVALID\_NDRANGE或CLK\_DEVICE\_QUEUE\_FULL。

### Clang块

Clang块在OpenCL标准中，作为一种方式对内核进行封装，并且Clang块可以进行参数设置，能让子内核顺利入队。Clang块是一种**传递代码和作用域的方法**。其语法与闭包函数和匿名函数类似。块类型使用到一个结果类型和一系列参数类型(类似lambda表达式)。这种语法让“块”看起来更像是一个函数类型声明。“^”操作用来**声明一个块变量(block variable)**(该块用来引用内核)，或是**用来表明作用域的开始**(使用内核代码直接进行声明)。

### 设备端设置内核参数需要动态分配局部内存时

enqueue\_kernel()有重载的函数：

int

enqueue\_kerenl(

queue\_t queue,

kernel\_enqueue\_flags\_t flags,

const ndrange\_t ndrange,

void (^block)(local void \*, ...),

uint size0, ...)

该函数可用来**创建局部内存指针**，标准中块可以是一个可变长参数的表达式(一个可以接受可变长参数的函数)。其中每个参数的类型必须是local void \*。注意，声明中，函数列表可以被void类型替代。

enqueue\_kernel()函数同样也是可变长的，末尾提供的数值是用来表示**每个局部素组的大小**。

# 第6章 OpenCL主机端内存模型

### 图像内存对象的优势

1、GPU上的层级缓存和数据流结构就是为了优化访问图像类型数组所准备

2、GPU驱动会在硬件层面上优化图像数据的排布，从而提升访问图像数据的效率，尤其是二维图像模式

3、硬件支持图像是一个很复杂的数据访问过程，在这个过程中硬件会将一些存储的数据进行压缩

### clCreateBuffer()

创建数组对象，只需要提供**上下文对象**，**数组大小**，以及一些**标识**。

函数会**返回一个数组对象**，如果需要将错误码传出，则需要传入最后一个参数。

flags参数可以将数组配置成只读或只写的数据，以及设置其他分配选项。错误码将从err传出，对应的错误码在OpenCL标准文档中都有定义。通常OpenCL函数执行成功，都会以CL\_SUCCESS作为错误码返回。

格式：

cl\_mem

clCreateBuffer(

cl\_context context,

cl\_mem\_flags flags,

size\_t size,

void \*host\_ptr,

cl\_int \*err)

### 子数组对象(subbuffer)

OpenCL也支持子数组对象(subbuffer)，也就是可以**将单独的数组对象再进行划分为更小的数组对象**，这些数组对象可以**相互覆盖**，可以**读或写和拷贝**，以及和其父数组以相同的方式使用。

注意有覆**盖和包含关系**的子数组对象，会让其父数组对象结构变的更加**复杂**，并且在实际使用过程中这种情况会造成一些未定义的行为。

### OpenCL中图像与数组对象的不同

1、图像数据排布的不透明性，使其不能直接在内核中使用指针进行数据读取

2、多维结构

3、图像对数据成员有一定的要求，并不能像数组那让接受任意数据类型

### clEnqueueReadImage()

主机端访问图像对象，对图像对象操作的主机函数支持多维度的寻址。

clEnqueueReadImage()更像clEnqueueReadBufferRect()，而非clEnqueuReadBuffer()。

### clGetSupportedImageFormats()

获取图像格式支持的列表。

### clCreateImage()

创建图像对象。

context，flags和host\_ptr这些与创建数组对象所需要的参数一致。

图像类型(image\_format)和图像描述符(image\_desc)参数定义了图像的**维度**，**数据格式**和**数据分布**。

格式：

cl\_mem

clCreateImage(

cl\_context context,

cl\_mem\_flags flags,

const cl\_image\_format \*image\_format,

const cl\_image\_desc \*image\_desc,

void \*host\_ptr,

cl\_int \*errcode\_ret)

### clCreatePipe()

创建管道对象。

管道数据通常称为**包(packets)**，其包含了OpenCL C或用户自定义的类型。

当创建一个管道，需要提供**包大小(pipe\_packed\_size)**和**最大包数(pipe\_max\_packets)**。如同其他创建内存对象的API，这个API也需要设置一些**与内存相关标识**。对于管道对象来说，只有设置**CL\_MEM\_READ\_WRITE**标识是合法的，其也是管道对象默认的标识参数。以后，管道对象**不可在主机端访问**，即便是编程者没有意识到，也需要使用CL\_MEM\_HOST\_NO\_ACCESS来显式表明管道对象在主机端不可访问。

格式：

cl\_mem

clCreatePipe(

cl\_context context,

cl\_mem\_flags flags,

cl\_uint pipe\_packet\_size,

cl\_uint pipe\_max\_packets,

const cl\_pipe\_properties \*properties,

cl\_int \*errcode\_ret)

### 在数组对象创建和初始化、传递内核参数，从内存对象中读回数据中，数据转移的过程

注意，运行时也直接在创建和初始化了设备端的内存对象。

(a)使用主机内存创建和初始化一个数组对象。

(b)内核之前，隐式的将主机端的数据搬运到设备端。

(c)显式的将设备内存中的数据搬回主机端。

### 显式的从主机端或设备端将数据拷贝到设备端或主机端

(a)创建一个为初始化的内存对象。

(b)内核执行之前，将主机端数据传入设备端。

(c)内核执行之后，将设备端数据传回主机端。

API：**clEnqueueWriteBuffer()**、**clEnqueueReadBuffer()**。

格式：

cl\_int

clEnqueueWriteBuffer(

cl\_command\_queue command\_queue,

cl\_mem buffer,

cl\_bool blocking\_write,

size\_t offset,

size\_t size,

const void \*ptr,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event \*event\_wait\_list,

cl\_event \*event)

clEnqueueWriteBuffer()和clEnqueueReadBuffer()的声明很类似，除了使用blocking\_write和blocking\_read。声明中可以看出，数据是在**buffer**和**ptr**之间进行传输。

其中**写命令**就将**主机端的数据在设备端进行拷贝**(在全局内存中进行备份)，并且**读命令**就将**设备端的数据在主机端进行拷贝**。

注意这里需要有命令队列参与。这样就需要指定设备，对其内存副本进行初始化。这种设计也让运行时能更快的将相应的数据传递到对应的设备，以便在内核执行时供内核使用。

编程者可以通过设置**offset和size**，来决定**从哪里拷贝多少byte个数据**。注意ptr是作为**主机端读或写的起始地址**，offset用来决定**数组对象的起始地址**，而**ptr的初始地址则由编程者决定**。

这种数据传输是可以是**异步**的。

当使用异步方式调用clEnqueueReadBuffer()时，函数返回时我们无法知晓拷贝过程是否完成，直到我们通过同步机制——通过**事件机制**，或**调用clFinish()**。

如果我们想**在函数返回时就完成拷贝**，则需要**将CL\_TRUE作为实参传入blocking\_write或blocking\_read中**。

同步对于OpenCL内存模型尤为重要。修改中的内存不保证可见，且不保证内存状态的一致性，直到用一个事件来表明该命令结束。主机指针和设备内存间的数据互传，我们不能在拷贝的同时，对内存数据进行其他的操作，直到我们确定互传完成。

与**设备内存**相关的是**上下文对象**，而非设备对象。

通过clEnqueueWriteBuffer()入队一个命令，直到其完成，过程中不能确定数据是否已经完全搬运到设备上，而能确定的是，主机端的数据已经开始进行转移。与其他API不同，**数据互传命令**也可以指定为是**同步**的。我们只需要简单的将之前的调用进行修改即可：

clEnqueueReadBuffer(

commandQueue,

outputBuffer,

CL\_TRUE, // blocking read

0,

sizeof(int) \* 16，

returnedArray,

0,

0,

&readEvent);

这样的调用，API需要**保证设备端的数据完全传回主机端后才进行返回**，并且**在返回之后，主机端才能对读回的数据进行操作**。

### clEnqueueMigrateMemObjects()

进行主机和设备间的数据转换，用来从当前地址(无论主机还是设备)转移到指定设备上。如果一个系统中有多个异构设备，那么该API也能用来进行不同设备间的数据数据交互。

注意**设备间数据交互**不是使用clEnqueueReadBuffer()和clEnqueueWriteBuffer()，只有**设备和主机端的数据进行交互**时，才会使用这两个API。

格式：

cl\_int

clEnqueueMigrateMemObjects(

cl\_command\_queue command\_queue,

cl\_uint num\_mem\_objects,

const cl\_mem \*mem\_objects,

cl\_mem\_migrationg\_flags flags,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event \*event\_wiat\_list,

cl\_event \*event)

clEnqueueMigrateMemObjects()需要使用**内存对象数组**作为**实参**传入，其可以通过一条命令转移多个内存对象。如同所有clEnqueue\*开头的函数一样，该函数也能**产生事件对象，指定依赖关系**。当事件对象的状态设置为**CL\_COMPLETE**时，代表着**相对应的设备端内存，已经传递到参数中的command\_queue命令队列上**了。

除了显式告诉运行时进行数据转移，该命令也可以进行更高效的**隐式转移**。当编程者入队该命令后，恰好**设备端在执行的任务与该数据转移命令没有任何关系**(例如，内核执行时不包括该数据转移的内存)，数据转移会在任务执行时进行，从而**隐藏了传输的延迟**。这种隐式传递的方式**clEnqueueReadBuffer()和clEnqueueWriteBuffer()也适用**。

### CL\_MIGRATE\_MEM\_OBJECT\_HOST

对于转移的内存对象，标准提供了一个标识：CL\_MIGRATE\_MEM\_OBJECT\_HOST。这个标识以为这告诉运行时，数据需要传输到主机端。如果设置该标识，那么**传入的命令队列对象，将被API忽略**。

### 零拷贝(zero-copy)数据

一个系统中具有一个CPU和一个离散GPU，这种情况下GPU就需要**通过PCIe总线**对主存数据进行访问。当设备能用这种方式直接访问主机端内存，那么这种数据通常被称为零拷贝(zero-copy)数据。

### clEnqueueMapBuffer()

对一个内存对象进行**设备端的映射**。

能够直接操作主机端或设备端数据，而无需显式使用API进行读取和写入。这里的映射实现并非意味着进行了拷贝。有了这种机制，零拷贝内存才算完美实现。设备端在修改零拷贝内存的过程对主机不可见，直到其完成修改才再对主机可见。

格式：

void \*

clEnqueueMapBuffer(

cl\_command\_queue command\_queue,

cl\_mem buffer,

cl\_bool blocking\_map,

cl\_map\_flags map\_flags,

size\_t offset,

size\_t size,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event \*event\_wait\_list,

cl\_event \*event,

cl\_int \*errcode\_ret)

调用clEnqueueMapBuffer()时，其会**返回一个可供主机端访问内存的指针**。当clEnqueueMapBuffer()产生的事件对象的状态为**CL\_COMPLETE**时，意味着**主机端可以安全使用返回指针，进行数据访问**。

与clEnqueueWriteBuffer()和clEnqueueReadBuffer()相同，clEnqueueMapBuffer()也可以设置**同步方式**，将**CL\_TRUE作为实参传入blocking\_map**时，函数将在主机端可以安全使用访存指针时，**将产生的指针返回**。

clEnqueueMapBuffer()还有一个**map\_flags**，这个标识可以设置的实参有：**CL\_MAP\_READ，CL\_MAP\_WRITE和CL\_MAP\_WRITE\_INVALIDATE\_REGION**。

CL\_MAP\_READ，主机只能对这块映射内存进行读取；

CL\_MAP\_WRITE和CL\_MAP\_WRITE\_INVALIDATE\_REGION都表示主机端只能对该指针内容进行修改。CL\_MAP\_WRITE\_INVALIDATE\_REGION是带有优化的选项，其会指定整个区域将会被修改或者忽略，并且运行时在其修改完之前不会对中间值进行映射。这种方式就无法保证数据的一致性状态，而运行时对于内存区域的潜在访问要快于CL\_MAP\_WRITE。

当主机端对映射数据修改完毕，其需要进行反映射API的调用。反映射时需要将制定内存对象和映射出的指针传入该API。clEnqueueUnmapMemObject()的声明如下：

cl\_int

clEnqueueUnmapMemObject(

cl\_command\_queue command\_queue,

cl\_mem memobj,

void \*mapped\_ptr,

cl\_uint num\_events\_in\_wait\_list,

const cl\_event \*event\_wait\_list,

cl\_event \*event)

反映射时需要将内存对象本身和映射指针传入该API中。如其他数据管理命令一样，当API产生的事件对象的状态为CL\_COMPLETE时，则代表数据更新完成。与其他API不同clEnqueueUnmapMemObject()没有阻塞参数。图6.3展示了一个内存对象从映射到反映射的过程。如果内存对象在设备端修改的时候进行映射，让主机端读取上面的数据，这种情况会引发一些未定义行为；反之亦然。

### AMD处理内存标志

默认的内存对象(并未提供任何标志)将直接在设备端分配。

当提供CL\_MEM\_COPY\_HOST\_PTR或CL\_MEM\_USE\_HOST\_PTR标志时，如果设备端支持虚拟内存，那么这块内存将创建为主机端的页锁定内存(不可分页)，并且也能被设备作为零拷贝内存直接访问。

如果设备不支持虚拟内存，那么就会像默认内存对象一样，在设备端创建内存。当编程者希望将数据分配在设备端，且能让主机端直接访问该内存的话，AMD提供了一种供应商指定扩展的标志——CL\_MEM\_USE\_PERSISTENT\_MEM\_AMD。当在AMD设备上使用了该标志，则可以在设备端分配内存，且主机端可以直接访问。

### OpenCL 2.0三种SVM类型

|  |  |  |  |
| --- | --- | --- | --- |
|  | **粗粒度SVM** | **细粒度SVM** | **系统细粒度SVM** |
| **OpenCL对象** | 数组 | 数组 | 无(任意主机端类型) |
| **共享粒度** | 数组 | 字节 | 字节 |
| **分配API** | clSVMAlloc | clSVMAlloc | malloc(或类似的C/C++函数或操作) |
| **一致性** | 同步点 | 同步点和选择性原子操作 | 同步点和选择性原子操作 |
| **显式同步设备端和主机端数据？** | 映射/反映射命令 | 无 | 无 |

**1、粗粒度SVM**：数据以OpenCL内存对象的形式共享。其具有一些同步点——内核执行、映射和逆映射。

**2、细粒度SVM**：数据以OpenCL内存对象的形式共享。不需要进行显式同步，不过在主机端和设备端对相应内存进行原子操作时，内存数据需要进行同步。

**3、系统细粒度SVM**：设备端和主机端使用的内存可以认为是完全一样的，内存对象使用的方式和C/C++一样。

### OpenCL共享数据的行为对比

|  |  |  |  |
| --- | --- | --- | --- |
| 操作 | 2.0之前 | 粗粒度SVM | 细粒度SVM |
| 数据拷贝到设备端 | clEnqueueWriteBuffer | 不需要 | 不需要 |
| 设备端执行原子操作对主机端可见 | 不适用 | 不可见 | 可见 |
| 设备端对数据进行修改对主机端可见 | 需要进行拷贝之后可见 | 内核执行完成时 | 内核执行完成后，或在设备端执行原子操作之后 |
| 数据从设备端拷贝到主机端 | clEnqueueReadBuffer | 不需要 | 不需要 |

### 粗粒度数组SVM

粗粒度数组SVM可以与**OpenCL内存对象**进行虚拟地址共享。

**粗粒度SVM内存**与**非SVM内存**的不同点在于，**主机和设备可以共用虚拟内存指针**。粗粒度SVM内存需要在主机端进行映射和反映射，这样才能**保证最后一次更新的数据对设备可见**。为了完成这个功能，主机端线程需要调用**clEnqueueMapBuffer()**将**指定的内存区域阻塞的进行映射**。当映射完成后，内核就可以对该内存进行使用。当clEnqueueMapBuffer()返回时，内核对该内存的任何操作，对于主机都是可见的。

### 粗粒度SVM使用

主机端需要调用**clSVMAlloc()**分配出SVM内存，这段内存可以在主机端和设备端共享。cl\_mem对象是通过**clCreateBuffer()**通过传入CL\_MEM\_USE\_HOST\_PTR参数和主机端内存指针进行创建。这里clSVMAlloc()分配出的指针就类似于传入的主机端指针。

同样，内存中的内容设备端和主机端也是能够自动共享的。这里就**不需要**再去调用clEnqueueWriteBuffer()和clEnqueueReadBuffer()来完成设备端和主机端数据进行共享了。

当不在需要SVM内存，可以通过**clReleaseMemObject()**API对粗粒度SVM对象进行释放。之后，还需要通过**clSVMFree()**API销毁SVM内存所开辟的空间。

### clSVMAlloc()

创建SVM内存。

SVM可以通过**标志**指定为：**只读**，**只写**和**可读写**。

**alignment**表示**内存对象需要在该系统上以最少多少字节对齐**。如果传入0，则因为这使用默认的对齐字节，那么将会是OpenCL运行时支持的**最大数据类型的大小**。

与clCreateBuffer()返回cl\_mem不同，clSVMAlloc()返回的是一个**void型指针**。就像C函数malloc()一样，clSVMAlloc()也会返回一个**非空的指针**来表示**内存分配成功**，否则分配失败。

将**CL\_MEM\_SVM\_FINE\_GRAIN\_BUFFER**标志传入**clSVMAlloc()**，就能创建**细粒度**数组SVM对象。

若要使用**SVM原子操作**，则需要将**CL\_MEM\_SVM\_ATOMICS**一并传入**flags**中。注意，**CL\_MEM\_SVM\_FINE\_GRAIN\_BUFFER**只能和**CL\_MEM\_SVM\_ATOMICS共同传入flags**，否则即为非法。

格式：

void \*

clSVMAlloc(

cl\_context context,

cl\_svm\_mem\_flags flags,

size\_t size,

unsigned int alignmet)

### clSVMFree()

释放SVM内存，其只需要传入**对应的上下文对象**和**SVM指针**即可。

clSVMFree()函数的调用会**瞬间结束**，而不需要等待什么命令结束。

将SVM内存使用clSVMFree()函数释放之后在进行访问，程序会出现**段错误**，这就和普通的C程序没有任何区别了。为了保证在一些**列命令使用完SVM之后，再对SVM进行释放**，OpenCL也提供了一种**入队释放**的方式：**clEnqueueSVMFree()**。

格式：

void

clSVMFree(

cl\_context context,

void \*svm\_pointer)

### 细粒度数组SVM

细粒度数组SVM支持**字节级别**的数据共享。

当设备支持**SVM原子操作**时，细粒度数组SVM内存对象可以**同时在主机端和设备端，**对**同一块内存空间**进行**读与写**。细粒度数组SVM也可被**同一或不同**设备，在同一时间对相同的区域进行并发访问。SVM原子操作可以为内存提供**同步点**，从而能保证OpenCL内存模型的一致性。

如果设备不支持SVM原子操作，主机端和设备端依旧可以对相同的区域进行并发的访问和修改，但这样的操作就会造成一些**数据的覆盖**。

细粒度**系统**SVM是对细粒度**数组**SVM的扩展，其将SVM的范围扩展到**主机端**的**整个内存区域**中——开辟OpenCL内存或主机端内存只需要使用**malloc()**就可以。如果设备支持细粒度系统SVM，那么对于OpenCL程序来说，内存对象的概念就不需要了，并且内存传入内核将是一件很简单的事情(如同CUDA内核函数的调用一样)。

查看设备支持哪种SVM，可以将**CL\_DEVICE\_SVM\_CAPABILITIES**标识传入**clGetDeviceInfo()**中进行**查询**。OpenCL标准规定，如果支持2.0及以上标准，则至少要支持粗粒度数组SVM。

# 第7章 OpenCL设备端内存模型

### OpenCL内存空间四种类型

1、全局内存

2、局部内存

3、常量内存

4、私有内存

### OpenCL C保证内存的一致性的同步操作

执行栅栏、内存栅栏、原子操作

### 层级的一致性描述

1、工作项内部，内存操作的顺序可预测：对于同一地址的**两次读写**将会被硬件或编译器**重新排序**。特别是对于图像对象的访问，即使被同一工作项操作，同步时也需要遵循**“读后写”**的顺序。

2、原子操作、内存栅栏或执行栅栏操作能保证同一工作组中的两个**工作项，看到的数据一致**。

3、原子操作、内存栅栏或执行栅栏操作能保证**工作组的数据一致**。不同工作组的工作项无法使用栅栏进行同步。

### work\_group\_barrier()

工作组中使用栅栏对工作组中的所有工作项进行同步。

栅栏要求工作组中的所有**工作项**都要达到指定位置，才能继续下面的工作。这样的操作能保证工作组内的数据保持一致(比如：将全局内存上的一个子集数据传输到局部内存中)。

其中**flags**用来指定需要使用栅栏来同步的**内存类型**。其有三个选项：**CLK\_LOCAL\_MEM\_FENCE**、**CLK\_GLOBAL\_MEM\_FENCE**和**CLK\_IMAGE\_MEM\_FENCE**，这三个选项分别对应能被整个工作组访问到的三种不同内存类型：**局部内存**、**全局内存**和**图像内存**。

第二版work\_group\_barrier()也可以指定内存范围。其结合flags可以进行更加**细粒度**的数据管理。

scope有两个有效参数：**memory\_scope\_work\_group**和**memory\_scope\_device**。当将memory\_scope\_work\_group和**CLK\_GLOBAL\_MEM\_FENCE**一起使用时，栅栏则能保证所有工作组中每个工作项在到达同步点时，可以看到**其他所有工作项完成的数据**。当将memory\_scope\_device和**CLK\_GLOBAL\_MEM\_FENCE**一起使用时，栅栏则能保证**内存可被整个设备进行访问**。CLK\_LOCAL\_MEM\_FENCE只能和memory\_scope\_work\_group一起使用，其只能保证工作**组内**的数据一致，无法对该工作组之外的工作项做任何保证。

格式：

void

work\_group\_barrier(

cl\_mem\_fence\_flags flags)

void

work\_group\_barrier(

cl\_mme\_fence\_flags flags,

memory\_scope scope)

### 原子操作的方式

原子“读改写”、原子加载、原子存储

### OpenCL 2.0原子操作

atomic\_int

atomic\_uint

atomic\_float

如果设备支持64bit原子扩展，那么就需要添加一些原子类型：

atomic\_long

atomic\_ulong

atomic\_double

atomic\_size\_t

atomic\_intptr\_t

atomic\_uintptr\_t

atomic\_ptrdiff\_t

64位原子指针类型只针对能够使用64位地址空间的计算设备。

### atomic\_fetch()

“**同步后修改**”原子函数。

其中**key**可以替换成add, sub, or, xor, and, min和max。

**object**传入的是原子类型的指针，**operand**传入的是要进行操作的数值。

**返回值C**是非原子版的A，其值是在对M操作之前A内存中的具体数值。

格式：

C atomic\_fetch\_<key>(volatile A \*object, M operand)

### 全局数组

\_\_global可以用来描述一个全局数组。

数组中可以存放任意类型的数据：标量、矢量或自定义类型。无论数组中存放着什么类型的数据，其是通过**指针顺序**的方式进行访问，并且执行内核可以对数组进行读写，也可以设置成**针对内核只读或只写**的数组。

### OpenCL图像类型

图像对象维度有一维、二维和三维，其分别对应图像类型image1d\_t、image2d\_t和image3d\_t

### 读取图像的内置函数

read\_imagef()、read\_imagei()和read\_imageui()，分别可读取**单浮点**、**整型**和**无符号整型**数据。可以传入整型和浮点型**坐标值**。

每个图像读取函数都具有三个参数：**图像对象**、**图像采样器**和**坐标位置**。返回值通常是**具有四个元素的矢量**。

这里传入的坐标使用的是int2和float2类型。因为指定的二维图像类型，image2d\_t。对于**一维**图像类型，传入**int和float**类型的坐标即可。对于**三维**图像类型，需要传入**int4和float4**类型的数据，其中最后一个分量不会是用到。

如果图像数据**不足四个通道**，那么大多数图像类型将**返回0**，用来表示未使用的颜色通道，**返回1**代表着**alpha(透明度)通道**。例如，读取单通道图像(CL\_R)，那么返回的float4类型中就会是这样(r, 0.0, 0.0, 1.0)。

读取函数的第二个参数是**采样器**。其表示**硬件或运行时系统如何看待图像对象**。创建采样器对象可以在**内核端**创建sampler\_t类型对象，也可以在**主机端**使用**clCreateSampler()**API进行创建。

格式：

float4

read\_imagef(

image2d\_t image,

sampler\_t samper,

int2 coord)

float4

read\_imagef(

image2d\_t image,

sampler\_t samper,

float2 coord)

float4

read\_imagef(

image2d\_t image,

int2 coord)

### 采样器对象

采样器对象决定了**如何对图像对象进行寻址**，以及**相应的滤波模式**，并且决定了**传入的坐标是否进行归一化**。

指定归一化坐标(CLK\_NORMALIZED\_COORDS\_TRUE)就是告诉采样器在**对应维度的[0,1]之间对图像进行寻址**。使用非归一化(CLK\_NORMALIZED\_COORDS\_FALSE)则直接使用**传入的坐标在对应维度上进行寻址**。

寻址模式用来解决当采样器采样到图像之外的范围时，应该返回何值。这对于一些不想进行边缘判断编程者来说，使用标志指定如何处理这种“越界访问”会很方便。**CLK\_ADDRESS\_CLAMP**会将超出部分截断，将返回一个边界值；**CLK\_ADDRESS\_REPEAT**超出部分会返回一个在有效范围内的值。

滤波模式有两个选项：**返回离给定坐标最近的图像元素值**(CLK\_FILTER\_NEAREST)，或**使用坐标周围图像点进行线性差值**(CLK\_FILTER\_LINEAR)。

### 写入图像函数

与读不同，写入函数就不需要传递采样器对象。取代采样器的是要写入图像的具体数值：(在写入图像时，所提供的坐标必须是非归一化的。)

格式：

void

write\_imagef(

image2d\_t image,

int2 coord,

float4 color)

### 管道读写函数

这些函数需要传入一个管道对象，还需要传入一个需要写入和读出的位置指针。当读或写的函数执行成功的话，函数就会返回0。

在有些情况下这些函数调用不会成功。当管道为**空**时，read\_pipe()将会返回一个**负值**；当管道**满**了时，write\_pipe()会返回一个**负值**。

格式：

int

read\_pipe(

pipe gentype p,

gentype \*ptr)

int

write\_pipe(

pipe gentype p,

const gentype \*ptr)

### reserve\_read\_pipe()和reserve\_write\_pipe()

编程者需要保证管道有足够的空间进行写入或者读出。reserve\_read\_pipe()和reserve\_write\_pipe()就会返回一个占位标识符(reservation identifier)，其类型为reserve\_id\_t。

格式：

reserve\_id\_t

reserve\_read\_pipe(

pipe gentype p,

nint num\_packets)

reserve\_id\_t

reserve\_write\_pipe(

pipe gentype p,

uint unm\_packets)

### commit\_read\_pipe()和commit\_write\_pipe()

当使用占位标识符时用来保证读出或写入过程完成的阻塞函数。

这些函数需要传入一个**占位标识符**和一个**管道对象**，并且**没有返回值**。当这些函数返回时，就能保证所有读或写操作完全提交。

格式：

void

commit\_read\_pipe(

read\_only pipe gentype p,

reserve\_id\_t reserve\_id)

void

commit\_write\_pipe(

write\_only pipe gentype p,

reserve\_id\_t reserve\_id)

### work\_group\_reserve\_write\_pipe()和work\_group\_reserve\_read\_pipe()

为了能正确的将管道对象上的数据删除，需要对工作项进行屏蔽，OpenCL C提供了相应的函数可以让每个工作组能够项管道对象进行预留和提交操作。

### work\_group\_commit\_read\_pipe()和work\_group\_commit\_write\_pipe()

为了保证访问管道对象正常完成。

### 使用常量地址空间的方式

1、可以通过数组创建的方式，之后将数组作为参数传入内核中。内核参数描述上，必须指定\_\_constant为对应指针的标识符。

2、内核端声明常量对象，并使用\_\_constant标识对其进行初始化，其属于编译时常量类型。

### OpenCL查询常量参数个数的限制、常量数组的最大尺寸

将CL\_DEVICE\_MAX\_CONSTANT\_ARGS和CL\_DEVICE\_MAX\_CONSTANT\_BUFFER\_SIZE传入**clGetDeviceInfo()**进行查询。

### clSetKernelArg()

为了分配内存使用clSetKernelArg()**传递局部内存的大小到内核中**。通常clSetKernelArg都是传递一个全局内存到内核端，现在这个内存参数需要置为NULL。这种方式意味着不需要全局变量的返回，所以其就为局部内存了。

格式：

ciErrNum = clSetKernelArg(

kernel object,

parameter index,

size in bytes,

NULL)

### 统一地址空间

包括全局、局部和私有地址空间。2.0标准中，并未将常量地址划分到同一地址空间中。

### OpenCL的一致性顺序

从弱到强：松散、获取-释放和顺序。

这些选项则由内存序选项指定：

1、**松散(memory\_order\_relaxed)**:这种内存序不会对内存序有**任何的约束**——编译器可以自由的对操作进行重排，包括后续的加载和存储操作。不过该方式可能会带来一些副作用，可能会造成结果错误。2.0之前的OpenCL标准中，原子操作就包含在松散的内存序中。因为缺少限制，所以编程者可能使用松散序获得最好的性能。

2、**获取(memory\_order\_acquire)**:**获取**操作和**加载**操作成对出现。当为同步操作指定该选项时，任何共享内存需要被其他执行单元(例如，其他工作项，或主机端线程)“释放”后才能进行存储。编译器需要将加载和存储操作移到同步操作之后。

3、**释放(memory\_order\_release)**:与获取操作不同，**释放**操作会和**存储**操作成对出现。当为同步操作指定释放序时，其会影响同步点之前的存储操作，使其操作对其他线程可见，并且在同步点之前的所有加载操作，必须在达到同步点前全部完成。编译器会将加载和同步操作移至同步点之前。

4、**获取-释放(memory\_order\_acq\_rel)**:该内存序具有**获取**和**释放**的属性：其会在获取到其他执行单元的内存时，释放自己所获取的内存。这个选项通常用于“读改写”操作。

5、**顺序(memory\_order\_seq\_cst)**:顺序一致性的内存序**不存在数据数据竞争**。该内存序中，加载和存储操作的执行顺序和程序的执行顺序一致，这样加载和存储操作也就是简单的交错与不同的执行单元中。该选项要比memory\_order\_acq\_rel更加严格，因为最后程序可以说是在串行执行。

### 可以作为内存域指定的选项

1、**工作项(memory\_scope\_work\_item)**:指定内存序要应用到每个工作项中。这里需要对图像对象进行行操作。

2、**工作组(memory\_scope\_work\_group)**:指定的内存序应用于工作组中的每个工作项。这个操作与栅栏操作相比，相当于一个轻量级的同步。

3、**设备(memory\_scope\_device)**:指定内存序用于某一个执行设备。

4、**所有设备(memory\_scope\_all\_svm\_devices)**:指定内存序应用于所有设备上的所有工作项，以及主机端(对细粒度SVM使用原子操作)。

### atomic\_compare\_exchange\_strong\_explicit()

“比较后交换”函数。

与之前看到函数声明不一样atomic\_compare\_exchange\_strong\_explicit()具有两个内存序参数——success和failure。这两个参数指定的是**当比较操作成功和没成功时所使用到的内存序**。编程者可以使用这种操作，来控制没有必要同步操作。比如，编程者将memory\_order\_relaxed传入failure，就是想在条件不成功的时候，不让工作项等待交换完成。

格式；

bool

atomic\_compare\_exchange\_strong\_explicit(

volatile A \*object,

C \*expected,

C desired,

memory\_order success,

memory\_order failure,

memory\_scope scope)

### OpenCL C对原子操作的操作域进行初始化

1、在程序范围内声明一个原子变量，可以使用ATOMIC\_VAR\_INIT()宏，该宏的声明如下所示：

#define ATOMIC\_VAR\_INIT(C value)

这种方式初始化的原子对象是在程序域内进行声明，且分配在去全局地址空间内。例如：

global atomic\_int sync = ATOMIC\_VAR\_INIT(0);

2、原子变量在内核端需要使用非原子函数atomic\_init()进行声明和初始化。注意，因为atomic\_init()是非原子函数，但是也不能被多个工作项同时调用。也就是，初始化需要串行且同步的进行，例如下面代码所示：

local atomic\_int sync;

if (get\_local\_id(0) == 0){

atomic\_init(&sync, 0);

}

work\_group\_barrier(CLK\_LOCAL\_MEM\_FENCE);

### atomic\_work\_item\_fence()

执行栅栏操作。

flags参数可以传入CLK\_GLOBAL\_MEM\_FENCE, CLK\_LOCAL\_MEM\_FENCE和CLK\_IMAGE\_MEM\_FENCE，或将这几个参数使用“位或”(OR)的方式共同传入。共同传入的方式，与单独传入的效果是一样的。

格式：

void

atomic\_work\_item\_fence(

cl\_mem\_fence\_flags flags,

memory\_order order,

memory\_scope scope)

# 第8章 异构系统下解析OpenCL

### GPU相比于CPU的特性

1、执行宽SIMD：大量执行单元对不同的数据，执行相同的指令。

2、大量多线程：GPU计算核支持并发大量线程。

3、硬件暂存式内存：物理内存可由编程者控制。

4、支持硬件同步：并发线程间支持细粒度交互。

5、硬件管理任务分配：工作队列管理与负载均和由硬件控制。

### 使用GPU处理图像获取较高性能的需求

1、提供大量的内核任务供硬件分发

2、如果内核过小，考虑将内核进行合并

### 合并访问

为了减少每个波面阵产生的请求，系统中的缓存会通过一种过滤机制进行合并读取操作，将写入操作合并，尽可能一次性写入更多的内容——这种机制被称为合并访问(coalesing)。

向量从规划好的内存(基于DRAM)中读取数据会更加的高效。

### 计算内核的带宽大小公式

8.3 OpenCL内存性能的考量 - 图1

EB代表**有效带宽**，Br代表**从全局内存上读取的数据量**(单位：byte)，Bw表示**写入全局内存的数据量(单位：byte)**，t代表**内核运行的时间**。

时间t的获取，可以通过一些性能测评工具，比如：ADM的CodeXL。Br和Bw可以通过每个工作项所读取或写入的数据量，然后乘以工作项的数量计算得出。所以，在某些情况下，这些读写的数据量都是估算出来的。

### 查询对应设备上局部内存的大小

OpenCL运行时API也支持查询对应设备上局部内存的大小。在编程者编译或暂存局部内存数据时，其可以作为OpenCL内核参数。下面代码第一个调用，是用来查询局部内存类型，其可以用来判断哪些内存属于局部内存或全局内存(哪些是可以缓存或不能缓存的)，第二个调用时用来返回局部内存的大小：

cl\_int err;

cl\_device\_local\_mem\_type type;

err = clGetDeviceInfo(

deviceId,

CL\_DEVICE\_LOCAL\_MEM\_TYPE,

sizeof(cl\_device\_local\_mem\_type),

&type,

0);

cl\_ulong size;

err = clGetDeviceInfo(

deviceId,

CL\_DEVICE\_LOCAL\_MEM\_SIZE,

sizeof(cl\_ulong),

&size,

0);

# 第9章 案例分析：图像聚类

### 图像聚类步骤

**1、生成特征**

我们已经将特征产生算法应用于BoW模型，将其称为加速稳健特征算法(SURF，Speeded Up Robust Features)。SURF算法在2006年，由Bay等人在欧洲计算机视觉会议(ECCV)上发表[1]，其对不同大小的图像进行尺度不变的变换。如图9.2所示，将图像出入SURF算法中，将会得到一组用来描述原始图像的特征值。每个特征值包含图像位置和描述矢量。每个特征中的描述符向量是一个64维的向量。没描述符中包括该特征颜色和特征位置周围颜色梯度关系。本章中的特征，都具有一个有着64个元素的描述符。其余的成员变量则不是本章关注的焦点。

**2、图像聚合**

SURF产生的描述符通常都经过量化，通常会经过k-means聚合，然后映射到群组中。每个群组的质心就是所谓的“视觉词”。

**3、直方图构建**

这一阶段的任务是将SURF算法产生的描述符设置到直方图的视觉词中去。为了完成这项工作，我们需要确定描述符和质心的对应关系。本例中，SURF产生的特征描述符和质心都由64维构成。我们通过计算描述符和所有质心的欧式距离，并且将描述符赋予距离最近的质心。直方图在这里就是用于统计每个质心被赋予描述符的次数。

### CPU串行实现SURF

// Loop over all the descriptors generated for the image

for (int i = 0; i < n\_des; i++){

membership = 0;

min\_dist = FLT\_MAX;

// Loop over all cluster centroids available

for (j = 0; j < n\_cluster; j++){

dist = 0;

// n\_featrues: No. of elements in each descriptor (64)

// Calculate the distance between the descriptor and the centroid

for (k = 0; k < n\_features; k++){

dist\_temp = surf[i][k] - cluster[j][k];

dist += dist\_temp \* dist\_temp;

}

// Update the minimum distance

if (dist < min\_dist){

min\_dist = dist;

membership = j;

}

}

// Update the histogram location of the closest centroid

histogram[membership] += 1;

}

第2行遍历了每个SURF特征的描述符

第7行遍历了集群的所有质心

第12行循环遍历当前描述符中的64个元素，并计算当前特征与当前集群质心的欧氏距离

第18行找到离集群质心最近的SURF特征，并将其设置为成员

CPU OpenMP并行实现

// Loop over all the descriptors generated for the image

for (int i = 0; i < n\_des; i++){

membership = 0;

min\_dist = FLT\_MAX;

// Loop over all cluster centroids available

for (j = 0; j < n\_cluster; j++){

dist = 0;

// n\_featrues: No. of elements in each descriptor (64)

// Calculate the distance between the descriptor and the centroid

for (k = 0; k < n\_features; k++){

dist\_temp = surf[i][k] - cluster[j][k];

dist += dist\_temp \* dist\_temp;

}

// Update the minimum distance

if (dist < min\_dist){

min\_dist = dist;

membership = j;

}

}

// Update the histogram location of the closest centroid

prargma omp atomic

histogram[membership] += 1;

}

编译标识出现在第2行，其作用就是将每一次循环迭代放置到一个线程中。

第18行的标识则告诉编译器，使用原子操作来更新共享内存。

### OpenCL原子操作实现SURF

kernel

void kernelGPU1(global float descriptors,

\_\_global float centroids,

\_\_global int \*histogram,

int n\_descriptors,

int n\_centroids,

int nfeatures){

// Global ID identifies SURF descriptor

int desc\_id = get\_global\_id(0);

int membership = 0;

float min\_dist = FLT\_MAX;

// For each cluster, compute the membership

for (int j = 0; j < n\_centroids; j++){

float dist = 0;

// n\_features: No. of elements in each descriptor(64)

// Calculate the distance between the descriptor and the centroid

for (int k = 0; k < n\_features; k++){

float temp = descriptors[desc\_id \* n\_features + k] -centroids[j \* n\_features + k];

dist += temp \* temp;

}

// Update the minimum distance

if (dist < min\_dist){

min\_dist = dist;

membership = j;

}

}

// Atomic increment of histogram bin

atomic\_fetch\_add\_explicit(&histogram[membership],1,memory\_order\_relaxed, memory\_scope\_device);

}

### OpenCL合并访问内存（转置矩阵）实现SURF

kernel

void kernelGPU2( global float descriptors,

\_\_global float centroids,

\_\_global int \*histogram,

int n\_descriptors,

int n\_centroids,

int nfeatures){

// Global ID identifies SURF descriptor

int desc\_id = get\_global\_id(0);

int membership = 0;

float min\_dist = FLT\_MAX;

// For each cluster, compute the membership

for (int j = 0; j < n\_centroids; j++){

float dist = 0;

// n\_features: No. of elements in each descriptor(64)

// Calculate the distance between the descriptor and the centroid

for (int k = 0; k < n\_features; k++){

float temp = descriptors[k \* n\_descriptors + desc\_id] -

centroids[j \* n\_features + k];

dist += temp \* temp;

}

// Update the minimum distance

if (dist < min\_dist){

min\_dist = dist;

membership = j;

}

}

// Atomic increment of histogram bin

atomic\_fetch\_add\_explicit(&histogram[membership], 1, memory\_order\_relaxed, memory\_scope\_device);

}

### OpenCL向量化实现SURF

kernel

void kernelGPU3( global float descriptors,

\_\_global float centroids,

\_\_global int \*histogram,

int n\_descriptors,

int n\_centroids,

int nfeatures){

// Global ID identifies SURF descriptor

int desc\_id = get\_global\_id(0);

int membership = 0;

float min\_dist = FLT\_MAX;

// For each cluster, compute the membership

for (int j = 0; j < n\_centroids; j++){

float dist = 0;

// n\_features: No. of elements in each descriptor(64)

// Calculate the distance between the descriptor and the centroid

// The increment of 4 is due to the explicit verctorization where

// the distance between 4 elements is calculated in each

// loop iteration

for (int k = 0; k < n\_features; k++){

float4 surf\_temp = (float4)(

descriptors[(k + 0) \* n\_descriptors + desc\_id],

descriptors[(k + 1) \* n\_descriptors + desc\_id],

descriptors[(k + 2) \* n\_descriptors + desc\_id],

descriptors[(k + 3) \* n\_descriptors + desc\_id]);

float4 cluster\_temp = (float4)(

centroids[j \* n\_feature + k],

centroids[j \* n\_feature + k + 1]

centroids[j \* n\_feature + k + 2]

centroids[j \* n\_feature + k + 3]);

float4 temp = surf\_temp - cluster\_temp;

temp = temp \* temp;

dist += temp.x + temp.y + temp.z + temp.w;

}

// Update the minimum distance

if (dist < min\_dist){

min\_dist = dist;

membership = j;

}

}

// Atomic increment of histogram bin

atomic\_fetch\_add\_explicit(&histogram[membership], 1, memory\_order\_relaxed, memory\_scope\_device);

}

### 局部内存的优势

1、局部内存与本地数据存(LDS)相对应，其比L1缓存大4倍。

2、再说缓存命中，LDS内存访存的延迟也要比L1缓存低很多。为了充分复用，数据放在LDS将会带来比放在L1缓存上更好的性能，因为LDS的延迟很小，即使是高频度访问也能轻松应对。

### OpenCL将SURF特征放入局部内存实现SURF

kernel

void kernelGPU4( global float descriptors,

\_\_global float centroids,

\_\_global int \*histogram,

int n\_descriptors,

int n\_centroids,

int nfeatures){

// Global ID identifies SURF descriptor

int desc\_id = get\_global\_id(0);

int local\_id = get\_local\_id(0);

int local\_size = get\_local\_size(0);

// Store the descriptors in local memory

\_\_local float desc\_local[4096]; // 64 descriptors 64 work-items

for (int i = 0; i < n\_features; i++){

desc\_local[i local\_size + local\_id] =

descriptors[i \* n\_descriptors + desc\_id];

}

barrier(CLK\_LOCAL\_MEM\_FENCE);

int membership = 0;

float min\_dist = FLT\_MAX;

// For each cluster, compute the membership

for (int j = 0; j < n\_centroids; j++){

float dist = 0;

// n\_features: No. of elements in each descriptor(64)

// Calculate the distance between the descriptor and the centroid

for (int k = 0; k < n\_features; k++){

float temp = descriptors[k \* local\_size + desc\_id] - centroids[j \* n\_features + k];

dist += temp \* temp;

}

// Update the minimum distance

if (dist < min\_dist){

min\_dist = dist;

membership = j;

}

}

// Atomic increment of histogram bin

atomic\_fetch\_add\_explicit(&histogram[membership], 1, memory\_order\_relaxed, memory\_scope\_device);

}

### OpenCL将聚类中点坐标放入常量内存实现SURF

kernel

void kernelGPU4( global float descriptors,

\_\_constant float centroids,

\_\_global int \*histogram,

int n\_descriptors,

int n\_centroids,

int nfeatures){

// Global ID identifies SURF descriptor

int desc\_id = get\_global\_id(0);

int local\_id = get\_local\_id(0);

int local\_size = get\_local\_size(0);

// Store the descriptors in local memory

\_\_local float desc\_local[4096]; // 64 descriptors 64 work-items

for (int i = 0; i < n\_features; i++){

desc\_local[i local\_size + local\_id] =

descriptors[i \* n\_descriptors + desc\_id];

}

barrier(CLK\_LOCAL\_MEM\_FENCE);

int membership = 0;

float min\_dist = FLT\_MAX;

// For each cluster, compute the membership

for (int j = 0; j < n\_centroids; j++){

float dist = 0;

// n\_features: No. of elements in each descriptor(64)

// Calculate the distance between the descriptor and the centroid

for (int k = 0; k < n\_features; k++){

float temp = descriptors[k \* local\_size + desc\_id] -centroids[j \* n\_features + k];

dist += temp \* temp;

}

// Update the minimum distance

if (dist < min\_dist){

min\_dist = dist;

membership = j;

}

}

// Atomic increment of histogram bin

atomic\_fetch\_add\_explicit(&histogram[membership], 1, memory\_order\_relaxed, memory\_scope\_device);

}

# 第10章 OpenCL的分析和调试

### 性能优化要点

1、当程序中有多个内核时，哪些内核需要去优化？

2、内核在命令队列中等待的时间与实际运行的时间

3、了解执行、初始化，以及内核编译在整个应用执行中的时间占比

4、主机与设备之间的数据I/O在整个应用执行中的时间占比

### clGetEventProfilingInfo()

通过事件的信息获取函数，其能提供命令的相关计时信息。

在创建**命令队列**的时候，需要设置**CL\_QUEUE\_PROFILING\_ENABLE**标识。一旦命令命令队列创建完成，就无法在对事件计时的功能进行开启或关闭。

格式：

cl\_int clGetEventProfilingInfo(

cl\_event event,

cl\_profling\_info param\_name,

size\_t param\_value\_size,

void \*param\_value,

size\_t \*param\_value\_size\_ret)

第一个参数，event事件对象时必须给定的，第二个参数是一个枚举值，用来描述描述所要获取相应的时间信息。

param\_name的值如下表：

|  |  |
| --- | --- |
| 事件状态 | param\_value返回的信息 |
| CL\_PROFILING\_COMMAND\_QUEUE | 使用一个64位的值对主机端将命令提交到命令队列的时间进行统计(单位：ns) |
| CL\_PROFILING\_COMMAND\_SUBMIT | 使用一个64位的值对命令从命令队列提交到相关的设备上的时间进行统计(单位：ns) |
| CL\_PROFILING\_COMMAND\_START | 使用一个64位的值对命令开始的时间进行记录(单位：ns) |
| CL\_PROFILING\_COMMAND\_END | 使用一个64位的值对命令完成的时间进行记录(单位：ns) |
| CL\_PROFILING\_COMMAND\_COMPLETE | 使用一个64位的值对命令及其相关子命令完成的时间进行记录(单位：ns) |

OpenCL命令队列是异步工作的，因此函数在命令入队时就返回了。所以在对事件对象进行计时查询时，需要调用一次clFinish()，以同步相关任务，让队列中的所有任务都完成。

### 使用OpenCL事件获取内核的时间信息

// Sample code that can be used for timing kernel execution duration

// Using different parameters for cl\_profiling\_info allows us to

// measure the wait time

cl\_event timing\_event;

cl\_int err\_code;

// !We ara timing the clEnqueueNDRangeKenrel call and timing

// information will be stored in timing\_event

err\_code = clEnqueueNDRangeKernel(

command\_queue,

kernel,

work\_dim,

global\_work\_offset,

global\_work\_size,

local\_work\_size,

0, NULL, &timing\_event);

cl\_ulong starttime, endtime;

err\_code = clGetEventProfilingInfo(timing\_event, CL\_PROFILING\_COMMAND\_START, sizeof(cl\_ulong), &starttime, NULL);

kerneltimer = clGetEventProfilingInfo(timing\_event, CL\_PROFILING\_COMMAND\_END, sizeof(cl\_ulong), &endtime, NULL);

unsigned long elapsed = (unsigned long)(endtime - starttime);

printf(“Kernel Execution\t%ld ns\n”, elapsed);

### CodeXL的主要功能

1、**性能分析模式**：CodeXL对OpenCL应用进行功能性的性能分析。CodeXL会将OpenCL运行时数据和AMD Radeon GPU执行的数据汇总。

2、**内核静态分析模式**：CodeXL可以视为一个静态分析工具，可以对OpenCL内核的编译、解析和汇编进行分析。这种模式下CodeXL也可以作为内核的原型工具。

3、**调试模式**：CodeXL可以用来调试OpenCL应用。CodeXL允许开发者对OpenCL内核源码和运行时API进行单步调试。这个模式下也可以观察函数参数，从而减少内存消耗。

### CodeXL的使用方式

1、**Visual Studio插件**：CodeXL可以对当前激活的解决方案进行分析。只需要在菜单栏找到插件，并使用插件运行程序即可进行调试。

2、**独立使用**：CodeXL也可以作为一个独立的软件，安装在Windows和Linux系统下。独立的软件使用方式有一个好处，就是不需要加载那么多的源文件。开发者只需要建立对应的CodeXL工程，并设置应用二进制文件的路径，命令行参数和内核源码所在位置即可。

### OpenCL应用的跟踪数据可了解的信息

1、通过时间线的角度，我们可以了解应用的高级架构。并且，可以确定应用中创建的OpenCL上下文数量，以及命令队列创建的数量和其在应用中的应用。内核执行和数据传输操作的时间也在时间线上可见。

2、总结页面可以帮助我们了解，当前应用的瓶颈是否在内核执行或数据传输上。我们找到最耗时的10个内核和数据传输操作，以及API调用(调用次数最多的API或执行最耗时的API)。

3、可以通过API跟踪的页面，来了解和调试所有API的输入参数和输出数据。

4、查看警告，并尝试使用最佳的方式调用OpenCL API。

### 时间线对于调试OpenCL的好处

1、你需要确定你应用中的外部框架是否正确。通过测试时间线，你可以确定队列和上下文对象的创建，与你的期望是否一致。

2、可以增加对同步操作的信心。例如，当内核A需要依赖一个内存操作，并且这个内存的数据来源于内核B，那么内核A就应该出现在内核B之后的时间线中。通过传统的调试方式，我们很难找到同步所引发的错误。

3、最后，你可以了解到应用是利用硬件是否高效。时间线展示了同时执行的独立内核和数据传输操作。

### 总结页面的主要汇总信息

**1、API总结页面**：页面展示了主机端调用的所用OpenCL API

**2、上下文总结页面**：页面统计了每个上下文对象上的所有内核和数据传输操作，也展示了数组和图像对象创建的个数。

**3、内核总结页面**：页面统计了应用所创建的所有内核。

**4、10个最耗时数据传输操作的总结页面**：页面将数据传输操作耗时进行排序，展示10个耗时最长的数据传输操作。

**5、10个最耗时内核的总结页面**：页面将内核执行耗时进行排序，展示了10个耗时最长的内核。

**6、警告/错误页面**：展示应用可能存在的问题。其能为未释放的OpenCL资源、OpenCL API失败的调用，以及如何改进能获取更优的性能。点击相关OpenCL API的超链接，就能看到相应API的一些信息。

### 性能计数器的作用

1、决定内核所需要分配的资源数量(通用寄存器，局部内存大小)。这些资源会受到正在GPU中运行的波面阵的影响。可以分配较高的波面阵数量来隐藏数据延迟。

2、决定在GPU运行指令时，所要是用ALU的数量，以及全局和局部内存的数量。

3、可以了解缓存的命中率，以及写入或读取全局内存中的数据量。

4、决定使用相应的向量ALU单元和内存单元。

5、了解任意局部内存(局部共享数据)的块冲突，SIMD单元尝试读取或写入同一个局部共享数据的相同位置，这样导致访存串行化，并且增加访问延迟。

### 内核分析器的收益

**1、使用OpenCL内核源码**：内核分析器不需要编译主机端源码，只需要编译OpenCL内核即可，其对于OpenCL内核源码来说，是一个很有用的工具。内核分析器包含一个离线编译器，这个编译器可以编译和反汇编OpenCL内核，并且通过分析工具能看到内核的ISA码。编译中出现的错误将出现在输出界面中。当不同的GPU设备支持不同的OpenCL扩展和内置函数时，内核分析器可以对内核进行检查，看其是否能在不同的GPU设备上编译通过。

**2、生成OpenCL二进制文件**：通常开发者不会希望将自己的内核源码进行发布。在这种情况下，OpenCL内核会以二进制文件和主要执行库或可执行文件一起发布。同样，OpenCL API也能对内核进行编译，并且保存成二进制文件。生成的二进制文件，只能用于同一平台上的设备。内核分析器命令行方式也可以生成二进制内核文件，用户使用这个工具可以生成AMD平台上支持的二进制内核。另外，内核分析器将一些选项的设置在内核二进制ELF文件中的某些字段中。这样就能避免以源码的形式发布内核了。这个二进制文件中只包含了ISA或LLVM的中间码，或源码。OpenCL内核二进制中不同字段所扮演的角色不同：

**· ISA字段**：如果开发者要包含一种特殊GPU设备的ISA码在二进制内核中，那么就要为其他OpenCL设备重新生成相应的二进制内核。

**· LLVM IR字段**：OpenCL二进制俄日那劲中的LLVM IR(或AMD IL)都支持大部分AMD设备。在OpenCL运行时，会将IR翻译成对应GPU的ISA。

**3、预先对内核的性能进行评估**：因为每个内核都能相对于主机端代码单独执行。这样就不需要知道太多OpenCL程序实现的细节。从而，能根据目标机器的信息对内核提前进行性能评估。

### ISA码的益处

1、可以看到**使用了多少通用寄存器**，并且了解**使用的寄存器数量是否会过多**。要是过多的话，应用就会使用全局内存来替代这些寄存器，从而造成访存的高延迟。寄存器使用的统计有助于我们对内核代码进行重构，从而减少或复用一定数量的通用寄存器，或者更多的使用局部内存。

2、可以看到**不同的显卡架构**，ISA指令中进行了不同次数的读取和存储，这些指令使用的数量，对于开发者来说是可控的。在了解指令数量之后，可以尽可能减少每个工作项读取或存储数据的尺寸。

3、可以通过类似**循环展开**的方式观察**生成ISA码有和不同**，从而达到优化的目的。而且开发者可以了解OpenCL的一些内置函数(比如原子操作)是如何进行实现的。

### 不让编译器对OpenCL内核进行任何的优化

使用-O0作为内核分析器的编译选项，不让编译器对OpenCL内核进行任何的优化。

### 内核分析器推断方式

**1、全真**：所有线程的分支状态为真——进入设计好的标签内。

**2、全假**：所有线程的分支状态为假——执行下一个分支。

**3、皆有**：一些线程的分支状态为假——执行所有分支。这样有些进入”else”分支的语句就完全不起作用。

### 异构应用的代码构成

1、API级别的代码(比如，clCreateBuffer(), clEnqueueNDRangeKenrel())。这些调用执行在主机端。

2、OpenCL命令，包括设备执行和数据传输。

### API级别调试的特性

**1、API函数断点**：CodeXL可以设置断点，断点设置和使用的方式与其他常用调试器相同。

**2、记录OpenCL API调用情况**：调试模式会让应用暂停执行，CodeXL会给我们展示在之前的运行过程中的最后一次OpenCL API的调用情况。图10.10展示了CodeXL作为调试器回溯该应用中所使用到的OpenCL命令。

**3、程序和内核信息**：OpenCL上下文对象包含多个程序对象和内核对象。CodeXL可以验证哪些程序对象与哪个上下文对象相关联。如果程序是使用clCreateProgramWithSource()创建，我们还可以看到传入到该函数的内核源码。

图像和数组对象数据：一个OpenCL上下文对象中包含多个数组对象和图像对象。CodeXL也可以让开发者直接查看这些对象中的数据。对于图像类型，CodeXL允许我们以图像的角度来查看图像对象中的数据。

**4、内存检查**：CodeXL可以让我们了解对应上下文对象的数组对象中内存内存使用情况。内存检查功能可以用来检查内存泄漏，因为在设备端内存不能直接被查看到，所以要查到相关问题会非常的困难。

**5、API使用统计**：CodeXL展示了当前选择的上下文对象中的统计的API使用信息。通过分解API在上下文中的调用，我们就能知道一个函数被调用了多少次。

### CodeXL有对OpenCL应用进行调试的方式

**1、OpenCL内核断点**：开发者可以在内核源码文件上进行断点设置。

**2、进入式调试**：开发者可以通过单步进入式调试的方式，直接从相关clEnqueueNDRangeKernel()函数进入相关内核中去。

**3、内核函数断点**：可以在断点对话框中添加相应的内核函数名作为函数断点。当内核运行匹配到相应函数名时，对应内核开始执行，调试结束是在内核函数开始运行的时候(这个也就是用来观察内核调用顺序，如果不在内核内部设置断点，调试过程是无法进入内核内部的)。

### 使用printf调试

对命令队列使用clFinish()能将所有入队内核中的printf()进行执行，打印出对应变量的信息。

和多线程程序一样，printf()打印出来的信息并不是顺序的。

# 第11章 高级语言映射到OpenCL2.0 —— 从编译器作者的角度

### C++ AMP为C++编程语言添加的规则

1、添加的函数限定于运行在GPU上；

2、允许GPU线程共享数据。

### C++ AMP向量相加

#include <amp.h>

#include <vector>

using namespace concurrency;

int main(void){

const int N = 10;

std::vector<float> a(N);

std::vector<float> b(N);

std::vector<float> c(N);

float sum = 0.f;

for (int i = 0; i < N; i++){

a[i] = 1.0f \* rand() / RAND\_MAX;

b[i] = 1.0f \* rand() / RAND\_MAX;

}

array\_view<const float, 1> av(N, a);

array\_view<const float, 1> bv(N, b);

array\_view<float, 1> cv(N, c);

parallel\_for\_each(cv.get\_extent(),

[=](index<1> idx)restrict(amp){

cv[idx] = av[idx] + bv[idx];

});

cv.synchronize();

return 0;

}

### C++ AMP特性

1、C++ AMP类和函数都属于concurrency命名空间。使用“using”的方式可以让我们在这段简单的代码中，不用为相应的函数加上前缀concurrency::。

2、支持Lambda表达式，Lambda的使用可以让主机和加速器的代码放在一起，甚至放在同一个函数中。

3、C++ AMP提供了一个预设值模板extent——只需要设置一个整型作为模板参数，用来获取数据等级。具有维度的类模板，有些就支持指定特定的预设值指定一个或者多个整型数值。

### C++ AMP array\_view

C++ AMP中，类模板array\_view作为数据读写的传媒。一个array\_view对象是一个多维的数据集合。这种方式并不是将已有数据拷贝到一个新的位置，而是使用一种新的方式去访问原始数据所在的地址。

模板有两个参数：数据的**类型**和数据的**维度**。通过不同维度上的索引，可以访问到不同等级的类型或对象。

### C++ AMP parallel\_for\_each

parallel\_for\_each结构属于C++ AMP数据并行计算的代码段。其类似于**对OpenCL内核的启动**。C++ AMP中对于**数据集**的操作称为**计算区域(compute domain)**，并且定义了一个**预设值对象**。

与OpenCL类似，每个线程都调用的是同一个函数，并且线程间都由自己区域，是完全分开的(类似于NDRange)。

与C++ STL中标准算法for\_each类似，parallel\_for\_each函数模板就是将指定函数应用到对应的数据集上。

第一个参数为一个**预设值对象**，其用指定数据并行计算的范围。

第二个参数是一个C++函数对象(或仿函数)。parallel\_for\_each会将这个函数对象(或仿函数)应用到计算区域的每个数据中。

### C++ AMP restrict

C++ AMP中借用了C99中的关键字“restrict”，并且允许在其之后可以跟某个函数的函数列表(包括Lambda函数)。restrict关键字后面允许使用括号，括号内可以有**一个或者多个限定符**。

C++ AMP只定义了两个限定符：**amp和cpu**。从限定符的字面意思也能猜出，这无疑是指定哪种计算设备作为加速代码执行的设备，不是将加速代码编译成CPU代码，就是将其代码编译成C++语言子集中的某种代码。

restrict(**amp**)限定了指定函数必须以调用**硬件加速器**。当**没有进行限定**指定，那么就默认为restrict(**cpu**)方式。有些函数也可以进行**双重指定**，restrict(cpu, amp)，这种方式函数可能会在主机或加速器上执行，这需要对创建**两种限定设备**的上下文。

### CLamp开源的组件

**1、C++ AMP编译器**：该项目由Clang和LLVM衍生而出，其编译器支持C++ AMP作为C++语言的扩展，并且内核代码使用OpenCL C或可移植的中间码表示。

**2、C++ AMP头文件**：由C++ AMP标准定义的一系列头文件。其对一些OpenCL内置函数进行了包装，但是有些还是需要再认真的考虑一下。

**3、C++ AMP运行时**：用来连接主程序和内核的桥接库。连接可执行文件，其会加载和构建内核，设置内核参数和运行内核。

### clCreateProgramWithBinary()

为了使用SPIR格式对程序对象进行编译

### OpenCL的相关步骤在C++ AMP的对应情况

|  |  |
| --- | --- |
| OpenCL | C++ AMP |
| 内核 | parallel\_for\_each中定义Lambda函数，或将一个函数直接指定给parallel\_for\_each |
| 内核名 | 使用C++函数操作符直接调用给定函数或Lambda函数 |
| 内核启动 | parallel\_for\_each |
| 内核参数 | 使用Lambda函数获取变量 |
| cl\_mem内存 | concurrency::array\_view或array |

### CLamp编译器编译和链接C++ AMP程序

1、写完代码之后，将C++ AMP源码以**“设备端模式”**进行编译(所有C++ AMP指定的语言规则都会检查并应用)。CLamp编译器可以产生相应的OpenCL内核(基于AMP的约束函数)，并将其编译成LLVM的位码文件。内核所调用的函数都必须是内联函数。主机端程序也会编译成生成相应的位代码，然后生成对应的C++函数。

2、LLVM位代码通过一些变化来保证底层OpenCL程序的正确性。首先是**对主机端代码的修整**，然后确保**在OpenCL程序中内核和指令中所使用的指针地址空间的正确性**(global, constant, local, private)。这里需要注意的是，**C++ AMP和OpenCL的指针在地址空间上是不相同的**。OpenCL中，地址空间值指针类型的一种，而在C++ AMP中其为指针值的一种。因此，静态编译分析器会通过指针的负值和使用操作，自行推断所使用的指针的地址空间。另外，对LLVM位数据的转换也会使用到元数据，使其能与OpenCL SPIR格式兼容。

3、将LLVM位码编译成OpenCL SPIR位码之后，就可以在支持cl\_khr\_spir扩展特性的平台上直接链接和执行。编译后的二进制文件会以另外的形式保存，其格式与主机端程序的格式是不一样的。另外，OpenCL C格式的内核代码可以使用在任何支持OpenCL平台的设备上，即使对应的设备不支持SPIR模式。

4、输入的C++ AMP源码会以**“主机模式”**对主机端代码进行编译。C++ AMP头文件都是设计好的，所以不会有内核代码在主机端模式下进行编译。不过，程序会调用C++ AMP运行时API函数来取代内核执行部分的代码。

5、主机端和设备端代码最终都会链接在一起，并产生一个可执行文件

OpenCL SPIR版的向量相加

kernel void

ZZ6vecAddPfS\_S\_iEN3019cxxamptrampolineEiiS\_N11Concurrency11access\_typeEiiS\_S2\_iiS\_S2(

global float \*llvm\_cbe\_tmp1,

unsigned int llvm\_cbe\_tmp2, global float llvm\_cbe\_tmp3,

unsigned int llvm\_cbe\_tmp4,

\_\_global float llvm\_cbe\_tmp5,

unsigned int llvm\_cbe\_tmp6){

unsigned int llvm\_cbe\_tmp7;

float llvm\_cbe\_tmp8;

float llvm\_cbe\_tmp9;

llvm\_cbe\_tmp7 = /tail/get\_global\_id(0u);

llvm\_cbe\_tmp10 = \*((&llvm\_cbe\_tmp1[((signed int)llvm\_cbe\_tmp7)]));

llvm\_cbe\_tmp11 = ((&llvm\_cbe\_tmp3[((signed int)llvm\_cbe\_tmp7)])); ((&llvm\_cbe\_tmp5)[((signed int)llvm\_cbe\_tmp7)])) = (((float)(llvm\_cbe\_tmp10 + llvm\_cbe\_tmp11)));

return;

}

第1行：生成对应的内核名

第2-3行：序列化array\_view va

第4-5行：序列化array\_view vb

第6-7行：序列化array\_view vc

第11行：获取全局工作项索引，通过C++ AMPLambda函数中的idx获取

第12行：加载va[idx]

第13行：加载vb[idx]

第14行：计算va[idx]+vb[idx]，并将结果保存在vc[idx]

### 线程划分的显式隐式

1、隐式的方式可能会自动的减少在一个线程束中对内存的访问，并且通过透明或半透明的线程划分，来达到最佳内存形式，并且获取最佳的性能。

2、显式的方式需要用户显式定义不同的内存对象(可能有些内存在片上共享，有些内存是离散的)，相关的数据移动也由用户进行控制。

### 支持线程划分的显式编程模型的特点

1、通过某种方式将计算域划分成固定大小的小块

2、通过显式的方式进行数据内存的指定，通常是片上、离散或线程私有这几种形式。其在OpenCL中的对应为local, global和\_\_private

3、为固定大小的计算块提供同步机制，以便于不同工作项之间的协同工作

### C++ AMP和OpenCL地址空间的区别

1、OpenCL中，其是指针类型的一种，其指针使用local进行声明(不能使用private对一段内存进行声明)。

2、C++ AMP中地址空间时指针值的一部分。可以使用通用指针进行指定：float \*foo;指针foo可以指向一个使用tile\_static创建的内存(与OpenCL中的\_\_local等价)，因为一定的局限性同一个指针，只能指向全局内存中的一个值。

### OpenCL创建只读只写内存对象

1、OpenCL中**CL\_MEM\_READ\_ONLY**索引就代表了这段内存**不能在计算时进行修改**。如果使用这个索引创建的内存对象，需要在其中放置一些常量数据，这些数据在计算完成后也**不需要拷贝回主机**。

2、**CL\_MEM\_WRITE\_ONLY**索引所创建的内存，大多数情况下是用来存放结果数据的。如果使用这个索引创建出的内存对象，**不需要在加速器计算之前拷贝数据**到这个内存当中。

### discard\_data()

C++ AMP中discard\_data()是**array\_view**的一个**成员函数**。在运行时调用这个函数会**将对应对象上的数据进行复写**，因此就**没有必要**在计算开始之前，将数据拷贝到设备端。这种情况下，我们可以使用CL\_MEM\_WRITE\_ONLY创建一个内存对象。

### array\_view<const T, N>

如果一个array\_view对象的第一个模板参数的限定符是const，那么我们只能通过CL\_MEM\_READ\_ONLY创建相应的内存对象。这样的话，OpenCL运行时就能知道，哪段内存在计算的时候不能够被修改。因此，这个内存对象上所存储的数据，在计算完成后不需要拷贝回主机。

# 第12章 WebCL：使用OpenCL加速Web应用

### WebCL编程的构成

1、主机端(例如，Web浏览器)用来控制和执行JavaScript程序

2、设备端(例如，GPU)用来进行计算——OpenCL内核

### WebCL定义的对象

1、命令队列

2、内存对象(数组和图像)

3、采样器对象，其描述了在内核中如何对图像进行读取

4、程序对象，其包含了一些列内核函数

5、内核对象，在内核源码中使用\_\_kernel声明的函数，其作为真正的执行对象

6、事件对象，其用来追踪命令执行状态，以及对一个命令进行性能分析

7、命令同步对象，比如标记和栅栏

### WebCL处理对象

1、**WebCLBuffer**和**WebCLImage**，可以用来**包装数组**。一个WebCLBuffer对象可以将数据以一维的方式存储起来。数组中错存储的元素类型都可以使标量类型(比如：int或float)，向量类型，或是自定义结构体类型。

2、**WebCLSampler**可以**对图像进行采样**

setArg()中使用的webcl.type与C类型之间的关系

|  |  |  |  |
| --- | --- | --- | --- |
| 内核参数类型 | setArg()值的类型 | setArg()数组类型 | 注意 |
| char, uchar | scalar | Uint8Array, Int8Arrary | 1 byte |
| short, ushort | scalar | Uint16Array, Int16Array | 2 bytes |
| int, uint | scalar | Uint32Array, Int32Array | 4 bytes |
| long, ulong | scalar | Uint64Array, Int64Array | 8 bytes |
| float | scalar | Float32Array | 4 bytes |
| charN | vector | Int8Array for (u)charN | N = 2,3,4,8,16 |
| shortN | vector | Int16Array for (u)shortN | N = 2,3,4,8,16 |
| intN | vector | Int32Array for (u)intN | N = 2,3,4,8,16 |
| floatN | vector | Float32Array for floatN and halfN | N = 2,3,4,8,16 |
| doubleN | vector | Float64Array for (u)doubleN | N = 2,3,4,8,16 |
| char, …, double \* | WebCLBuffer |  |  |
| image2d\_t | WebCLImage |  |  |
| sampler\_t | WebCLSampler |  |  |
| \_\_local |  | Int32Array([size\_in\_bytes]) | 内核内部定义大小 |

### WebCL注意点

1、长整型是64位整数，其无法在JavaScript中表示。其只能表示成两个32位整数：低32位存储在数组的第一个元素中，高32位存储在第二个元素中。

2、如果使用\_\_constant对内核参数进行修饰，那么其大小就不能超多webcl.DEVICE\_MAX\_CONSTANT\_BUFFER\_SIZE。

3、OpenCL允许通过数组传递自定义结构体，不过为了可移植性WebCL还不支持自定义结构体的传递。其中很重要的原因是因为主机端和设备端的存储模式可能不同(大端或小端)，其还需要开发者对于不同端的设备进行数据整理，即使主机和设备位于同一设备上。

4、所有WebCL API都是线程安全的，除了kernel.setArg()。不过，kernel.setArg()在被不同的内核对象并发访问的时候也是安全的。未定义的行为会发生在多个线程调用同一个WebCLKernel对象时。

### 本设备的OpenCL是否支持乱序

可以尝试使用QUEUE\_OUT\_OF\_ORDER\_EXEC\_MODE\_ENABLE测试一下OpenCL实现是否支持乱序队列。如果得到的是INVALID\_QUEUE\_PROERTIES异常抛出的话，那么你所使用的设备就不支持乱序队列。

// Create an in-order command-queue(default)

var queue = context.createCommandQueue(device);

// Create an in-order command-queue with profiling of commands enabled

var queue = context.createCommandQueue(device, webcl.QUEUE\_PROFILING\_ENABLE)；

// Create an out-of-order command-queue

var queue = context.createCommandQueue(device, webcl.QUEUE\_OUT\_OF\_ORDER\_EXEC\_MODE\_ENABLE);

### WebCL中的enqueueNDRange

1、kernel——执行命令的内核对象。

2、offsets——对全局区域的偏移。如果传null，则代表offsets=[0, 0, 0]。

3、globals——内核所要解决问题的尺寸。

4、locals——每个维度上的工作组中，工作项的数量。如果传null，设备会对其自行设定。

### WebCL中webCLCommandQueue类的参数

1、event\_list——一个WebCLEevet数组

2、event——用来对设备执行命令的状态进行查询的事件对象

通常，这两参数传的都是null。不过，当有事件传递给某个命令时，主机端可以使用clWaitForEvents用来等待某个命令执行结束。编程者也可以使用事件的回调函数，在对应命令完成时进行提示。这要求主机端代码在事件中提前注册一个回调函数。当event\_list传递命令时，之后命令不会立即执行，只有等到event\_list上所有事件对象对应的命令执行完成，才会执行当前的命令。

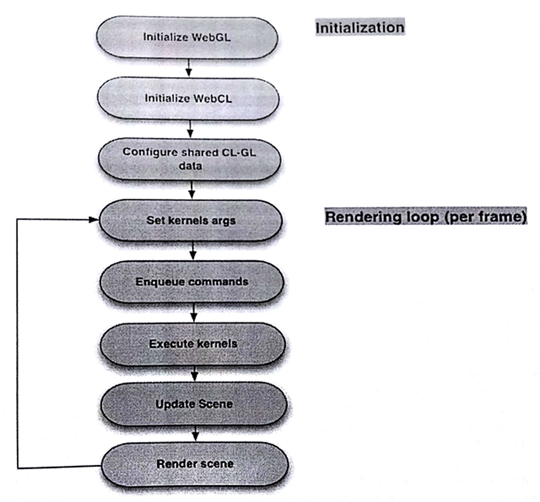
### WebGL、WebCL能够共享的内存对象

1、纹理对象：包含图像的纹理信息

2、顶点数组对象：包含顶点信息，比如坐标、颜色和法向量

3、渲染数组对象：包含图像对象所用到的WebGL帧缓存对象[4]

### 运行时调用WebCL和WebGL



### 两个能保护内存的准则

1、分配的内存需要初始化(程序不能访问旧数据)；

2、访问合法内存的代码可以置为可信。

### 保证程序安全性的机制

1、持续跟踪内存分配(如果平台支持动态分配，需要在运行时进行跟踪)

2、监视合法读取的内存范围

3、合法的数据砸编译和运行时，会显得更加高效

# 第13章 其他高级语言中OpenCL的使用