

OpenCL 规范

版本： 1.0

修订版本： 48

Khronos OpenCL Working Group

编者：Aaftab Munshi

2009-10-6

译者： 倪庆亮

20091023

Copyright (c) 2008-2009 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be re-formatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group web-site should be included whenever possible with specification distributions.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Khronos, OpenKODE, OpenKOGS, OpenVG, OpenMAX, OpenGL ES, gLFX and OpenWF are trademarks of the Khronos Group Inc. COLLADA is a trademark of Sony Computer Entertainment Inc. used by permission by Khronos. OpenGL and OpenML are registered trademarks and the OpenGL ES logo is a trademark of

Silicon Graphics Inc. used by permission by Khronos. OpenCL is a trademark of Apple Inc. used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

设计者：倪庆亮

致谢

OpenCL 规范是许多人贡献的结果，涉及桌面、手持和嵌入式等多个计算机工业领域；下面列出了部分人员，以及当时所在的公司。

Andrzej Mamona, AMD

Benedict Gaster, AMD

Bill Licea Kane, AMD

David Garcia, AMD

Ed Buckingham, AMD

Jan Civlin, AMD

Laurent Morichetti, AMD

Mark Fowler, AMD

Michael Houston, AMD

Michael Mantor, AMD

Norm Rubin, AMD

Robert Simpson, AMD

Aaftab Munshi, Apple

Benjamin Lipchak, Apple

Bob Beretta, Apple

Daniel N. Gessel, Apple

David Black-Schaffer, Apple

Derek Gerstmann, Apple

Geoff Stahl, Apple

Ian Ollmann, Apple

Inam Rahman, Apple

Jeff Kidder, Apple

Jeremy Sandmel, Apple

John Stauffer, Apple

Kathleen Danielson, Apple

Michael Larson, Apple

MonPing Wang, Apple

Nate Begeman, Apple

Nick Burns, Apple

Nicolas Moss, Apple

Ralph Brunner, Apple

Stephen Canon, Apple

Travis Brown, Apple

Andrew Cox, ARM

Dave Shreiner, ARM

Eivind Liland, ARM

Roger Nixon, Broadcom

Rob Barris, Blizzard

Alastair Donaldson, Codeplay

Andrew Richards, Codeplay

Andrew Brownsword, Electronic Arts

Eric Schenk, Electronic Arts

Erik Noreke, Ericsson

Jacob Strom, Ericsson

Teddie Stenvi, Ericsson

Brian Murray, Freescale

Barry Minor, IBM

Brian Watt, IBM

Dan Brokenshire, IBM

Joaquin Madruga, IBM

Mark Nutter, IBM

Joe Molleson, Imagination Technologies

Aaron Lefohn, Intel

Andrew Lauritzen, Intel

Craig Kolb, Intel

Geoff Berry, Intel

John Kessenich, Intel

Josh Fryman, Intel

Hong Jiang, Intel

Larry Seiler, Intel

Matt Pharr, Intel

Ofer Rosenberg, Intel

Paul Lalonde, Intel

Stephen Junkins, Intel

Tim Foley, Intel

Timothy Mattson, Intel

Bill Bush, Kestrel Institute

Lindsay Errington, Kestrel Institute

Jon Leech, Khronos

Cormac Brick, Movidia

David Donohoe, Movidia

Jyrki Leskelä, Nokia

Kari Pulli, Nokia

Amit Rao, NVIDIA

Chris Cameron, NVIDIA

Christopher Lamb, NVIDIA

Ian Buck, NVIDIA

Jason Sanders, NVIDIA

Mark Harris, NVIDIA

Michael Gold, NVIDIA

Neil Trevett, NVIDIA

Alex Bourd, Qualcomm

Michael McCool, RapidMind

Stefanus Du Toit, RapidMind

Jonathan Grant, Renesas

Robert Schulman, Seaweed Systems

John Bates, Sony

Ajit Kamat, Symbian

Madhukar Budagavi, Texas Instruments

Tom Olson, Texas Instruments

目录

1	介绍	1
2	术语	2
3	OPENCL架构	7
3.1	平台模型	7
3.2	执行模型	8
3.2.1	执行模型：上下文和命令队列	10
3.2.2	执行模型：内核的种类	11
3.3	内存模型	11
3.3.1	内存一致性	13
3.4	编程模型	13
3.4.1	数据并行编程模型	13
3.4.2	任务并行编程模型	14
3.4.3	同步	14
3.5	OPENCL框架	14
4	OPENCL平台层	15
4.1	查询平台信息	15
4.2	查询设备	17
4.3	上下文	23
5	OPENCL运行时	27
5.1	命令队列	27
5.2	内存对象	31

5.2.1	创建缓冲对象	32
5.2.2	读、写和拷贝缓冲对象	33
5.2.3	保留和释放内存对象	36
5.2.4	创建图像对象	37
5.2.4.1	图像格式描述符	40
5.2.5	查询所支持的图像格式	42
5.2.5.1	图像格式最小支持列表	42
5.2.6	读、写和拷贝图像对象	43
5.2.7	图像对象和缓冲对象间的拷贝	48
5.2.8	映射和解映射内存对象	51
5.2.8.1	用来访问内存对象所映射区域的OpenCL命令的行为	57
5.2.9	内存对象查询	57
5.3	采样器对象	59
5.4	程序对象	62
5.4.1	创建程序对象	62
5.4.2	构建程序执行体	65
5.4.3	构建选项	66
5.4.3.1	预处理选项	66
5.4.3.2	固有数学选项	67
5.4.3.3	优化选项	67
5.4.3.4	请求或抑制告警的选项	68
5.4.4	卸载OpenCL编译器	69
5.4.5	程序对象查询	69

5.5	内核对象	72
5.5.1	创建内核对象	72
5.5.2	设置内核参数	74
5.5.3	内核对象查询	76
5.6	执行内核	78
5.7	事件对象	84
5.8	内核和内存对象命令的乱序执行	87
5.9	内存对象和内核上的PROFILE操作	89
5.10	刷新 (FLUSH) 和完成 (FINISH)	90
6	OPENCL C编程语言	91
6.1	支持的数据类型	91
6.1.1	内建标量数据类型	91
6.1.1.1	数据类型half	93
6.1.2	内建矢量数据类型	94
6.1.3	其它内建数据类型	95
6.1.4	保留 (reserved) 数据类型	95
6.1.5	类型对齐	96
6.1.6	矢量字面量 (vector literals)	96
6.1.7	矢量组件 (component)	97
6.2	变换 (CONVERSION) 和类型转换 (CAST)	99
6.2.1	隐式变换	99
6.2.2	显式转换	99

6.2.3	显式变换	100
6.2.3.1	数据类型	101
6.2.3.2	舍入模式	101
6.2.3.3	溢出行为和饱和变换	102
6.2.3.4	显式变换示例	102
6.2.4	将数据重新诠释为另一种类型	103
6.2.4.1	使用联合重新诠释类型	103
6.2.4.2	使用as_typed()重新诠释类型	104
6.2.5	指针转换	105
6.3	运算符	105
6.4	矢量运算	110
6.5	地址空间限定符	111
6.5.1	__global (或global)	111
6.5.2	__local (或local)	112
6.5.3	__constant (或constant)	112
6.5.4	__private (或private)	112
6.6	图像访问限定符	113
6.7	函数限定符	113
6.7.1	__kernel (或kernel)	113
6.7.2	可选的特性 (attribute) 限定符	113
6.8	限制	115
6.9	预处理器指令和宏	116
6.10	特性限定符	118

6.10.1	为类型指定特性	118
6.10.2	为函数指定特性	120
6.10.3	为变量指定特性	120
6.10.4	为块和控制流语句指定特性	122
6.10.5	扩展的特性限定符	122
6.11	内建函数	122
6.11.1	工作项函数	123
6.11.2	数学函数	124
6.11.2.1	math.h的浮点宏和编译指示	130
6.11.3	整数函数	131
6.11.4	公共函数	134
6.11.5	几何函数	135
6.11.6	关系函数	136
6.11.7	加载和存储矢量数据的函数	137
6.11.8	读写图像的函数	141
6.11.8.1	采样器	141
6.11.8.2	内建图像函数	143
6.11.9	同步函数	147
6.11.10	显式内存屏障函数	148
6.11.11	全局内存和局部内存间的异步拷贝，以及预取	148
7	OPENCL数值一致性	149
7.1	舍入模式	149

7.2	INF、NAN和去规格化数	150
7.3	浮点异常	150
7.4	相对误差即ULP	150
7.5	边界情况的行为	153
7.5.1	C99 TC2 之外的附加需求	153
7.5.2	对C99, TC2 的行为作出的改变	155
7.5.3	在flush-to-zero模式中, 边界情况的行为	155
8	图像寻址和过滤	156
8.1	规范化坐标	156
8.2	寻址模式和过滤	156
8.3	变换规则	160
8.3.1	对于规范化整型通道数据类型的变换规则	160
8.3.1.1	规范化整型通道数据类型到浮点值得变换	160
8.3.1.2	从浮点值到规范化整型通道数据类型的变换	161
8.3.2	half浮点通道数据类型的变换	162
8.3.3	浮点通道数据类型的变换	163
8.3.4	带符号和无符号 8 位、16 位和 32 位整型通道数据类型的变换	163
9	可选扩展	164
9.1	对可选扩展的编译器指令	164
9.2	获取OPENCL扩展函数的指针	165
9.3	双精度浮点数	166
9.3.1	变换	167

9.3.2	数学函数.....	167
9.3.3	公共函数.....	169
9.3.4	几何函数.....	169
9.3.5	关系函数.....	169
9.3.6	加载和存储矢量数据的函数.....	170
9.3.7	全局内存和局部内存间的异步拷贝，以及预取.....	173
9.3.8	IEEE 754 一致性.....	173
9.3.9	相对误差即ULP.....	174
9.4	选择舍入模式.....	175
9.5	32 位整数的原子函数.....	177
9.6	32 位整数的局部原子操作.....	178
9.7	64 位原子操作.....	180
9.8	写入 3D图像对象.....	182
9.9	可按字节寻址的存储.....	183
9.10	半浮点.....	184
9.10.1	变换.....	184
9.10.2	数学函数.....	185
9.10.3	公共函数.....	185
9.10.4	几何函数.....	186
9.10.5	关系函数.....	186
9.10.6	读写图像的函数.....	188
9.10.7	加载和存储矢量数据的函数.....	190

9.10.8	全局内存和局部内存间的异步拷贝，以及预取	190
9.10.9	IEEE 754 一致性	190
9.10.10	相对误差即ULP.....	191
9.11	由GL上下文或共享组创建CL上下文.....	193
9.11.1	概述.....	193
9.11.2	新过程和新函数	193
9.11.3	新记号	193
9.11.4	对第 4 章的补充	194
9.12	与OPENGL/OPENGL ES缓冲对象、材质和渲染缓存对象共享内存对象.....	200
9.12.1	共享对象的生命周期	201
9.12.2	CL缓冲对象→GL缓冲对象.....	201
9.12.3	CL图像对象→GL材质.....	202
9.12.3.1	OpenGL和OpenCL图像格式对应关系	205
9.12.4	CL图像对象→GL渲染缓存.....	205
9.12.5	由CL内存对象查询GL对象信息	207
9.12.6	在GL和CL上下文间共享映射到GL对象的内存对象	208
9.12.6.1	同步OpenCL和OpenGL对共享对象的访问	210
10	OPENCL嵌入式简档.....	211
11	参考文献.....	218
附录 A	共享	221
A.1	共享的OPENCL对象.....	221
A.2	多个宿主机线程	221

附录 B	移植性.....	222
附录 C	范例	225
C.1	一个简单的OPENCL内核.....	226
C.2	矩阵转置	229
C.3	矩阵降阶	232

1 介绍

现代处理器架构中将并行视为一种提高性能的重要途径。由于固定功率下提升时钟频率所面临的技术挑战，目前只能通过增加 CPU 核心数目来提高性能。GPU 也从只具有固定功能的渲染设备变成了可编程的并行处理器。鉴于今天的计算机系统通常包含高度并行的 CPU、GPU 和其它类型的处理器，让软件开发人员完全利用这些异构处理平台的优势就变得非常重要。

由于传统的多核 CPU 和 GPU 的编程方法彼此有很大不同，因此为异构并行处理平台创建应用是一个挑战。虽然基于 CPU 的并行编程模型一般都是建立在某个标准之上的，但是它往往假定有一个共享的地址空间而且不包含矢量运算。通用目的的 GPU 编程模型具有复杂的内存分级寻址和矢量运算，但通常基于某个特定平台、供应商或硬件。这些限制使得开发人员很难通过单独的多平台源码库使用异构 CPU、GPU 和其它类型的处理器。更进一步，需要让软件开发人员充分利用异构处理平台的优势：从高性能的计算服务器，桌面计算机系统，到手持设备，这些设备中包含多种不同的并行 CPU、GPU 和其它处理器（像 DSP 和 Cell/B.E.处理器）。

OpenCL（开放式计算语言）是一种开放的免税标准，使用它可以在 CPU、GPU 和其它处理器上进行通用目的的并行编程，它使软件开发者可以更加方便高效的使用这些异构处理平台。

OpenCL 支持广泛的应用，它通过一个低级别、高性能、可移植的抽象，使得从嵌入式和消费软件到 HPC 解决方案都可以得到支持。通过创建一个高效、底层的编程接口，OpenCL 会在并行计算生态系统中形成一个基础层，而这个生态系统中会包含独立于平台的工具、中间件和应用。OpenCL 特别适合在新兴的交互式图形应用中扮演越来越重要的角色，而这些应用会将通用的并行计算算法和图形渲染管线组合起来使用。

OpenCL 含有一个 API，用来协调异构处理器间的并行计算；还含有一个交叉平台编程语言，此语言带有一个规定详尽的计算环境。OpenCL 标准：

- ✚ 支持基于数据和任务的并行编程模型
- ✚ 使用一个带有并行扩展的 ISO C99 的子集。
- ✚ 定义了一致的数值需求（基于 IEEE 754）。
- ✚ 为手持和嵌入式设备定义了一个配置简档（profile）。
- ✚ 可以与 OpenGL、OpenGL ES 和其它图形 API 进行高效的互操作。

本文档先对一些基本的概念和 OpenCL 的架构进行一个概述，然后再详细描述它的执行模型、内存模型和对同步的支持。那时再讨论 OpenCL 平台和运行时 API，紧接着是对 OpenCL C 语言的详细描述。同时就如何用 OpenCL 编程进行采样计算给出了一些示例。

此规范划分为以下几部分：一是核心规格，任何兼容 OpenCL 的实现都必须支持；二是手持/嵌入式简档，降低了对手持和嵌入式设备兼容 OpenCL 的要求；三是一些可选扩展，这些扩展在后续修订 OpenCL 规范时可能会变成核心规范。

2 术语

应用 (Application): 运行在宿主机和 OpenCL 设备上的程序。

阻塞和非阻塞的入队 API 调用 (Blocking and Non-Blocking Enqueue API calls): 一个非阻塞的入队 API 调用将会在命令队列中放置一个命令后立刻返回。而对于阻塞模式的入队 API 调用，直到命令完成后才会返回。

隔层 (Barrier): 有两种类型的隔层——命令队列隔层 (command-queue barrier , 以下简称 CQB) 和工作组隔层 (work-group barrier , 以下简称 WGB)。

- ✚ OpenCL API 提供一个函数用来入队一个命令——命令队列隔层。这个隔层命令可以保证只有之前入队的所有命令都执行完毕后，后续命令才能开始执行。
- ✚ OpenCL C 语言提供了一个内建的工作组隔层函数。运行在设备上的内核可以利用这个函数在 (正在执行此内核的) 工作组 (work-group) 中的工作项 (work-item) 间执行同步。在隔层外执行工作组中的任意一个工作项之前，必须先为其中所有工作项构建隔层。

缓冲对象 (Buffer object): 存储一个线性字节序列的内存对象。在一个 (正在设备上运行的) 内核中，可以通过一个指针来访问缓冲对象。宿主机可以用 OpenCL API 调用来操控缓冲对象。一个缓冲对象包含以下信息：

- ✚ 字节数
- ✚ 用来描述使用信息和分配自哪个区域的属性。
- ✚ 缓冲数据

命令 (Command): 提交给命令队列来执行的 OpenCL 运算。例如，将内核提交给计算设备执行、操控内存对象等 OpenCL 命令。

命令队列 (Command-queue): 一个用来保存命令的对象，这些命令将在某个设备上执行。命令队列是在一个上下文中的某个设备上创建的。命令按序入队，但可能顺序执行也可能乱序执行。参见顺序执行和乱序执行 (In-order Execution and Out-of-order Execution)。

命令队列隔层 (Command-queue Barrier): 请看隔层。

计算单元 (Compute Unit): 一个 OpenCL 设备会包含一个或多个计算单元。一个工作组只能在一个计算单元上执行。一个计算单元由一个或多个处理元件组成。一个计算单元也可能包含专注于材质过滤的单元, 此单元可以被它的处理元件访问。

并发 (Concurrency): 系统的一个属性, 而此系统中会有一个任务的集合, 这些任务可以同时保持活跃并取得进展。为了在运行程序时利用并发执行, 程序员必须找出可以并发的部分, 在源码中标示出来, 用一个支持并发的标记进行开发。

不变内存 (Constant Memory): 全局内存的一个区域, 在内核执行过程中保持不变。宿主机分配并初始化内存对象, 然后放到不变内存中。

上下文 (Context): 内核执行时所处的环境, 同步和内存管理就是在此范围内定义的。上下文包含设备、这些设备可以访问的内存、相应的内存属性、一个或多个命令队列(用来对内核的执行和内存对象相关操作进行调度)。

数据并行编程模型 (Data Parallel Programming Model): 传统上, 这个术语涉及一个编程模型, 此模型中, 并发由单个程序中应用到数据结构中多个元素上的指令来表示。在 OpenCL 中, 对这个术语进行了推广, 此模型中, 单个程序的一套指令同时应用到索引的抽象域中的每一点上。

设备 (Device): 设备是计算单元的集合。一个命令队列被用来对操作设备的命令进行排队, 例如执行内核或读写内存对象等命令。典型的 OpenCL 设备有 GPU、多核 CPU 和其它处理器 (如 DSP 和 Cell/B.E.处理器)。

事件对象 (Event Object): 事件对象封装了操作 (如一个命令) 的状态。它可以用来对某个上下文中的操作进行同步。

事件等待列表 (Event Wait List): 事件等待列表是事件对象的列表, 可用来控制何时执行命令。

框架 (Framework): 一个软件系统, 含有一套组件用以支持软件开发和执行。一个典型的框架包含库、API、运行时系统、编译器等等。

全局 ID (Global ID): 一个全局 ID 用来唯一标识一个工作项, 源自执行内核时所指定的全局工作项数目。全局 ID 是一个 N 维的值, 起自 (0, 0, ... 0)。请参看局部 ID (Local ID)。

全局内存 (Global Memory): 同一上下文中的所有工作项均可访问的内存区域。宿主机可以用一些命令 (如 read、write 和 map) 来访问它。

GL 共享组 (GL share group) : GL 共享组对象用来管理共享的 OpenGL 或 OpenGL ES 资源 (如材质、缓存、帧缓存或渲染缓存), 并涉及一个或多个 GL 上下文对象。典型的 GL 共享组是一个不透明的对象, 不能直接访问。

句柄 (Handle) : 一个不透明的类型, 用来引用 OpenCL 所分配的对象。对对象的所有操作都是通过引用其句柄来进行的。

宿主机 (host) : 宿主机使用 OpenCL API 与上下文进行交互。

宿主机指针 (Host pointer) : 指针, 指向宿主机上的虚拟地址空间中的内存。

违规 (Illegal) : 明显不允许的行为, OpenCL 会将其视为错误进行报告。

图像对象 (Image Object) : 一个内存对象, 用来存储二维或三维的结构化数组。只有通过读写函数才能访问图像数据。读函数需要使用采样器 (sampler)。

图像对象封装了以下信息:

- ✚ 图像的维数。
- ✚ 图像中每个元素的描述。
- ✚ 一些属性, 用来描述使用情况和分配自哪个区域。
- ✚ 图像数据

图像中的所有元素均选自一个预定义图像格式的列表。

依赖于具体实现 (Implementation Defined) : 明确被允许可以不同的行为, 要求 OpenCL 的实现者为其提供相关文档。

顺序执行 (In-order Execution) : OpenCL 中的执行模型, 命令队列中的命令按提交的顺序执行, 只有当正在运行的命令完成后, 下一个才能开始运行。参看乱序执行 (Out-of-order Execution)。

内核 (Kernel) : 内核是程序中声明的一个函数, 可以在 OpenCL 设备上执行。对于程序中定义的任何函数, 都可以通过加上限定符 `_kernel` 标识成内核。

内核对象 (Kernel Object) : 一个内核对象封装了程序中声明的一个 `_kernel` 函数和执行此函数所用的参数。

局部 ID (Local ID) : 一个局部 ID 属于一个正在执行内核的工作组, 且在此工作组内是唯一的。局部 ID 是一个 N 维的值, 起于 (0, 0, ... 0)。参见全局 ID。

局部内存 (Local Memory): 属于某个工作组内存区域, 只有那个工作组中的工作项才可以访问它。

记号 (Marker): 排在命令队列中的一个命令, 可以用来给之前入队的命令做标记。此命令会返回一个事件, 应用可以等在这个事件上, 例如等待处在记号命令之前的命令全部完成。

内存对象 (Memory Objects): 全局内存中某区域的句柄, 此区域带有引用计数。参见缓冲对象 (Buffer Object) 和图像对象 (Image Object)。

内存区域/内存池 (Memory Region/Memory Pool): OpenCL 中的一个明确的地址空间。不同的内存区域在物理上可能重叠, 但在逻辑上 OpenCL 会将其视为不同。可以将其指定为私有的 (private)、局部的 (local)、不变的 (constant) 和全局的 (global)。

对象 (Object): 对 OpenCL API 可以操控的资源的一种抽象描述。包含程序对象 (program object)、内核对象 (kernel object)、内存对象 (memory object)。

乱序执行 (Out-of-Order Execution): 工作队列中的命令开始和完成执行可以以任意顺序进行, 只要与事件等待列表和命令队列隔层相容即可。参见顺序执行 (In-order Execution)。

平台 (platform): 宿主机加一些可以被 OpenCL 框架所管理的设备。应用可以在平台的设备上共享资源 (resource) 和执行内核。

私有内存 (Private Memory): 专属于某个工作项的一块内存区域。一个工作项的私有内存中定义的变量对另一个工作项是不可见的。

处理元件 (Processing Element): 一个虚拟的标量处理器。一个工作项可以在一个或多个处理元件上执行。

程序 (Program): 一个 OpenCL 程序由一套内核组成, 还可能包含一些辅助函数 (由 `_kernel` 函数调用) 和常量数据。

程序对象 (Program Object): 一个程序对象封装有以下信息:

- ✚ 对所关联上下文的引用。
- ✚ 程序源码或二进制。
- ✚ 最近成功构建的可执行程序, 设备列表 (用来运行此程序), 所用的构建选项和构建日志。
- ✚ 当前所附内核对象的数目。

引用计数 (Reference Count) : 一个 OpenCL 对象的生命周期由其引用计数来决定, 这个内部计数会记录所有对此对象的引用的数目。当你在 OpenCL 中创建一个对象时, 其引用计数被设成 1。后续对保留 (retain) API (如 `clRetainContext`, `clRetainCommandQueue`) 的调用会增大引用计数。对释放 (release) API (如 `clReleaseContext`, `clReleaseCommandQueue`) 的调用会减小引用计数。当引用计数降为 0 后, OpenCL 会回收此对象的资源。

放宽的一致性 (Relaxed Consistency) : 在内存一致性模型中, 不同的工作项或命令所看到的内容也可能不同, 当然, 隔层和其他显式同步点除外。

资源 (Resource) : OpenCL 所定义的一类对象。一个资源的实例就是一个对象。最常用的资源是上下文 (context)、命令队列 (command-queue)、程序对象 (program object)、内核对象 (kernel object) 和内存对象 (memory object)。计算用资源主要指那些参与推进程序计数器动作的硬件元件, 包括宿主机、设备、计算单元和处理元件。

保留、释放 (Retain、Release) : 动作, 会增大 (retain) 和减小 (release) 一个 OpenCL 对象的引用计数。这是一个记账功能, 它可以保证在使用这个对象的所有实例都完成之前, 系统不会移除这个对象。参见引用计数 (Reference Count)。

采样器 (Sampler) : 此对象用来描述在内核读取图像时怎样对其采样。读取图像的函数将采样器作为一个参数。采样器指定图像的寻址方式, 如图像的坐标越限时怎样处理、滤波模式、输入的图像坐标是否已规范化。

SIMD : 单指令多数据。这是一个编程模型, 一个内核在多个处理元件上并发执行, 每个处理元件上都有自己的数据, 还有一个共享的程序计数器。所有处理元件执行严格一致的一套指令。

SPMD : 单程序多数据。此编程模型中, 一个内核在多个处理元件上并发执行, 每个处理元件上都有自己的数据和自己的程序计数器。因此, 运行同一个内核的所有计算资源都会维护自己的指令计数器; 同时由于内核中的不同分支, 这些处理元件中的实际指令序列可能会有很大不同。

任务并行编程模型 (Task Parallel Programming Model) : 这个编程模型中, 计算表示为多个并发的任务, 其中, 一个任务就是单个工作组 (大小是 1) 中执行的那个内核。这些并发的任务可以运行不同的内核。

线程安全 (Thread-safe) : 对于一个 OpenCL API 调用, 只有在被多个宿主机线程同时调用时, OpenCL 所管理的内部状态保持一致, 才认为它是线程安全的。如果一个 OpenCL API 调用是线程安全的, 那么就允许应用多个宿主机线程中同时调用它, 而不必在这些线程间实施互斥。

未定义 (Undefined): 指 OpenCL 没有显式定义那些 API 调用、内建函数 (由内核使用或用来执行内核) 的行为。至于当 OpenCL 碰到一个未定义的构造时会发生什么, 不要求实现指定。

工作组 (Work-group): 相关工作项的集合, 这些工作项在同一个计算单元上执行。一个工作组中的所有工作项执行同一个内核, 共享局部内存和工作组隔层。

工作组隔层 (Work-group Barrier): 参见隔层 (Barrier)。

工作项 (Work-item): 内核的并行执行体中的一个。作为一个计算单元中所执行的一个工作组中的一部分, 一个工作项可以由一个或多个处理元件执行。用其全局 ID 和局部 ID 来区分一个工作项和所在工作组中的其他工作项。

3 OpenCL 架构

OpenCL 是一个开放的工业标准, 可以为 CPU、GPU 和其它分离的计算设备 (这些设备被组织到单个平台中) 所组成的异构群进行编程。它不只是一种语言。OpenCL 是一个并行编程的框架, 包括一种语言、API、库和一个运行时系统来支持软件开发。例如, 使用 OpenCL, 程序员可以写出一个能在 GPU 上执行的通用程序, 而不必将其算法映射到 3D 图形 API (如 OpenGL 或 DirectX) 上。

OpenCL 的目标是使那些想写出可移植且高效的代码的程序员成为专家。这包括库作者、中间件提供商和面向性能的应用程序员。因此 OpenCL 提供了底层的硬件抽象和一个框架来支持编程, 同时也暴露了下面硬件的许多细节。

为了描述 OpenCL 背后的核心思想, 我们将使用一个分级模型:

-  平台模型
-  内存模型
-  执行模型
-  编程模型

3.1 平台模型

OpenCL 的平台模型的定义可以查看图 3.1。此模型中, 一个**宿主机**连接到了一个或多个 **OpenCL 设备**上。一个 OpenCL 设备被划分成一个或多个**计算单元 (CU)**, 每个计算单元又被划分为一个或多个**处理元件 (PE)**。设备上的计算发生在处理元件中。

OpenCL 应用会按照宿主机平台的原生模型在这个宿主机上运行。OpenCL 应用会从

宿主主机上提交命令给设备上的处理元件来执行计算。一个计算单元中的所有处理元件作为 SIMD 单元或 SPMD 单元（每个 PE 维护自己的程序计数器）执行单个指令流。

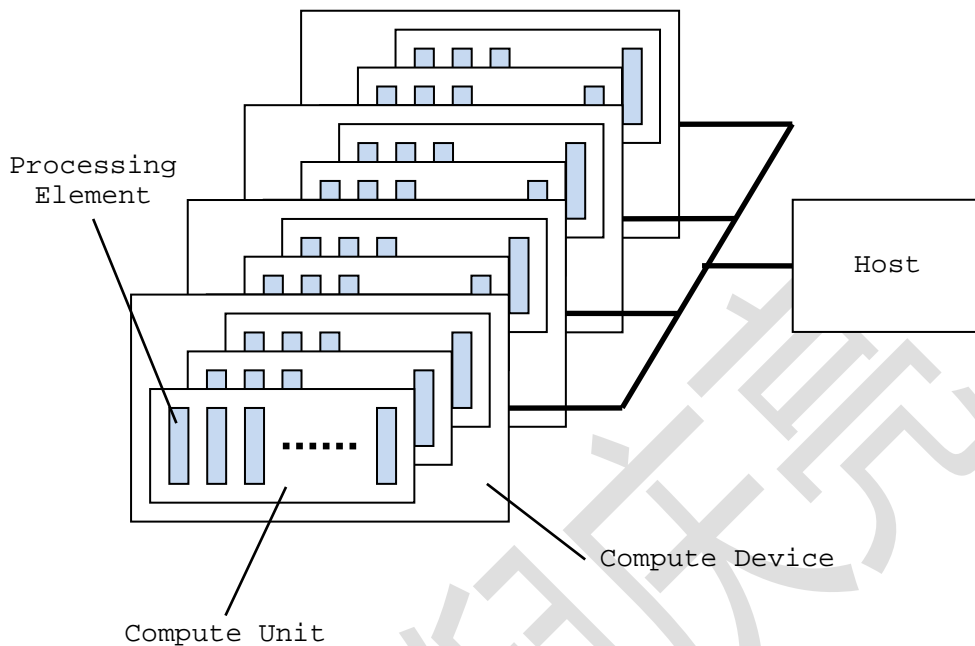


图 3.1 平台模型……一个宿主主机加上一个或多个计算设备，每个设备具有一个或多个计算单元，每个计算单元具有一个或多个处理元件。

3.2 执行模型

有两处会执行 OpenCL 程序：在一个或多个 **OpenCL 设备** 上执行**内核**、在宿主主机上执行**宿主主机程序**。宿主主机程序为内核定义了上下文并管理内核的执行。

OpenCL 执行模型的核心是通过内核怎样执行来定义的。当宿主主机提交内核来执行时，就会定义一个索引空间，内核的实例会为其中所有点而执行。这个内核实例就是一个**工作项**，通过它在索引空间中的点来标识，并且为这个工作项提供了一个全局 ID。所有工作项都会执行相同的代码，但是代码的执行路径和用来运算的数据可能会不同。

工作项被组织到**工作组**中。工作组提供了对索引空间更粗粒度的分解。工作组有一个唯一的工作组 ID，此 ID 和工作项所使用的索引空间具有同样的维数。另外工作项还有一个局部 ID，此 ID 在其所在工作组中是唯一的，所以单个工作项可以通过其全局 ID 或通过其局部 ID 加工作组 ID 来唯一标识。一个给定的工作组中的工作项会在单个计算单元中的多个处理元件上并发执行。

OpenCL 1.0 所支持的索引空间叫做一个 NDRange。NDRange 是一个 N 维的索引空间，其中 N 是 1、2 或 3。一个 NDRange 由一个长度为 N 的整数数组来定义，N 指定了

索引空间每个维度的范围。每个工作项的全局 ID 和局部 ID 都是 N 维的元组。全局 ID 的取值范围从 0 开始，到这个维度上元素个数减 1。

通过跟工作项全局 ID 相似的途径可以给工作组指定一个 ID。一个长度为 N 的数组定义了每个维度上工作组的数目。将工作项指定给一个工作组时，会为其分配一个局部 ID，其范围从 0 开始到工作组在那个维度上的大小减 1。因此，工作组 ID 加上此工作组中的一个局部 ID 可以唯一确定一个工作项。有两种途径来识别一个工作项 通过全局索引的方式，或通过一个工作组索引加上此工作组中的一个局部索引的方式。

例如，考虑一下图 3.2 中的二维索引空间。

我们为工作项 (G_x, G_y) 和每个工作组 (S_x, S_y) 的大小输入索引空间。全局索引定义了一个 G_x, G_y 索引空间，其中工作项的数目是 G_x 和 G_y 的乘积。局部索引定义了一个 S_x, S_y 索引空间，单个工作组中工作项的数目是 S_x 和 S_y 的乘积。给定每个工作组的大小，和工作项的总数，我们可以算出工作组的数目。一个二维的索引空间用来唯一标识一个工作组。每个工作项通过其全局 ID (g_x, g_y) 来标识，或通过工作组 ID (w_x, w_y) 、每个工作组的大小 (S_x, S_y) 和此工作组中的局部 ID (s_x, s_y) 来标识，如：

$$(g_x, g_y) = (w_x * S_x + s_x, w_y * S_y + s_y)$$

工作组的数目可以这样计算：

$$(W_x, W_y) = (G_x / S_x, G_y / S_y)$$

给定一个全局 ID 和工作组大小，一个工作项的工作组 ID 可以这样计算：

$$(w_x, w_y) = ((g_x - s_x) / S_x, (g_y - s_y) / S_y)$$

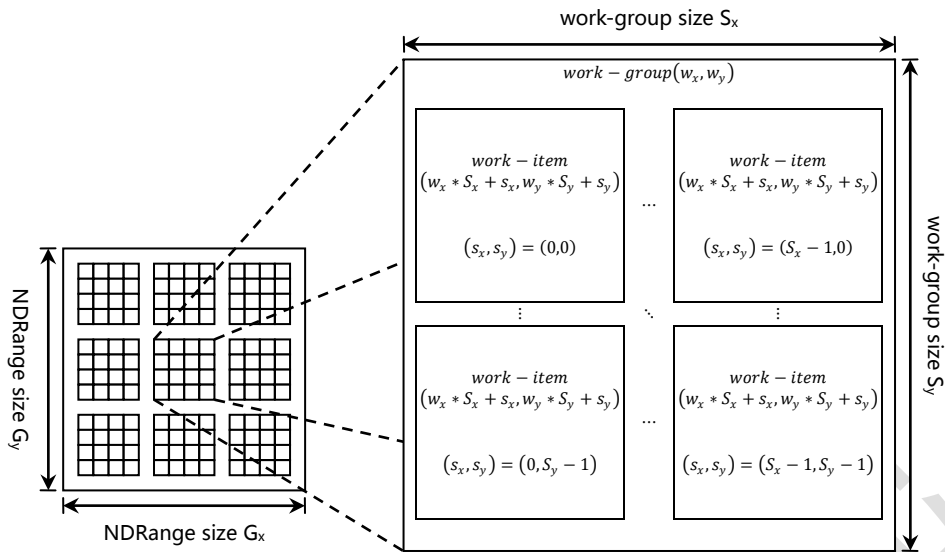


图 3.2 NDRange 索引空间的一个例子，上面有工作项、他们的全局 ID 和所映射到的一对工作组 ID 和局部 ID




大部分编程模型都可以映射到这个执行模型上，在 OpenCL 中明确支持的有两种：**数据并行编程模型**和**任务并行编程模型**。

3.2.1 执行模型：上下文和命令队列

宿主机为内核的执行定义了一个上下文。此上下文包括以下资源：

1. **设备**：宿主机可以使用的 OpenCL 设备集。
2. **内核**：运行在 OpenCL 设备上的 OpenCL 函数。
3. **程序对象**：程序源码和实现内核的执行体。
4. **内存对象**：对宿主机和 OpenCL 设备可见的一组内存对象。这些内存对象包含一些值，内核实例可以在这些值上进行运算。

宿主机使用 OpenCL API 中的函数来创建和操控上下文。宿主机创建一个叫做命令队列的数据结构来协调设备上内核的执行。宿主机将命令放入命令队列，在设备上的上下文中进行调度。这包括：

-  **内核执行命令**：在设备的处理元件上执行内核。
-  **内存命令**：读写内存对象，或者自宿主机地址空间映射和解映射内存对象。
-  **同步命令**：限制命令的执行顺序。

命令队列对在设备上执行的命令进行调度。这些命令在宿主机和设备上异步执行。执行时，命令间的关系属于下面两种模式之一：

- ✚ **顺序执行**：按照在命令队列中出现的顺序来发起命令，并按顺序结束。换言之，前面的命令完成后，后面的命令才能开始。这将队列中命令的执行顺序串行化。
- ✚ **乱序执行**：按顺序发起命令，但后续命令执行前不必等待前面命令完成。任何顺序上的限制都是由程序员通过显式的同步命令强加的。

内核的执行和提交给一个队列的内存命令会生成事件对象。这用来控制命令的执行、协调宿主主机和设备的运行。

可以将多个队列关联到同一个上下文上。这些队列并发且独立运行，OpenCL 中没有显式的机制来对它们进行同步。

3.2.2 执行模型：内核的种类

OpenCL 执行模型支持两类内核：

- ✚ **OpenCL 内核**是用 OpenCL C 编程语言所写就，并用 OpenCL 编译器编译的。所有 OpenCL 的实现都支持 OpenCL 内核。实现可能会提供其他机制来创建 OpenCL 内核。
- ✚ **原生内核**是通过一个宿主机函数指针来访问的。原生内核与设备上的 OpenCL 内核一起入队执行，并共享内存对象。例如，这些原生内核可以是应用代码中定义的函数，也可以是从库中导出的函数。注意，执行原生内核的能力是 OpenCL 的一个可选功能，原生内核的语义依赖于具体实现。OpenCL API 中的一些函数可以用来查询设备能力和决定设备是否支持这个能力。

3.3 内存模型

正在执行内核的工作项可以访问四块不同的内存区域：

- ✚ **全局内存**：所有工作组中的所有工作项都可以对其进行读写。工作项可以读写此中内存对象的任意元素。对全局内存的读写可能会被缓存，这取决于设备的能力。
- ✚ **不变内存**：全局内存中的一块区域，在内核的执行过程中保持不变。宿主机负责对此中内存对象的分配和初始化。
- ✚ **局部内存**：隶属于一个工作组的内存区域。它可以用来分配一些变量，这些变量由此工作组中的所有工作项共享。在 OpenCL 设备上，可能会将其实现成一块专有的内存区域，也可能将其映射到全局内存中。
- ✚ **私有内存**：隶属于一个工作项的内存区域。一个工作项的私有内存中所定义的变量对另外一个工作项来说是不可见的。

表 3.1 描述内核或宿主机是否可以从一个内存区域中分配内存、怎样分配（静态如编译时 VS 动态如运行时）和允许怎样访问（如内核或宿主机是否可以对一个内存区域进行读

写)。

表 3.1 内存区域的分配和内存访问能力

		全局	不变	本地	私有
宿主机	分配	动态	动态	动态	No
	访问	读/写	读/写	No	No
内核	分配	No	静态	静态	静态
	访问	读/写	只读	读/写	读/写

图 3.3 描述了内存区域以及怎样与平台模型关联在一起。

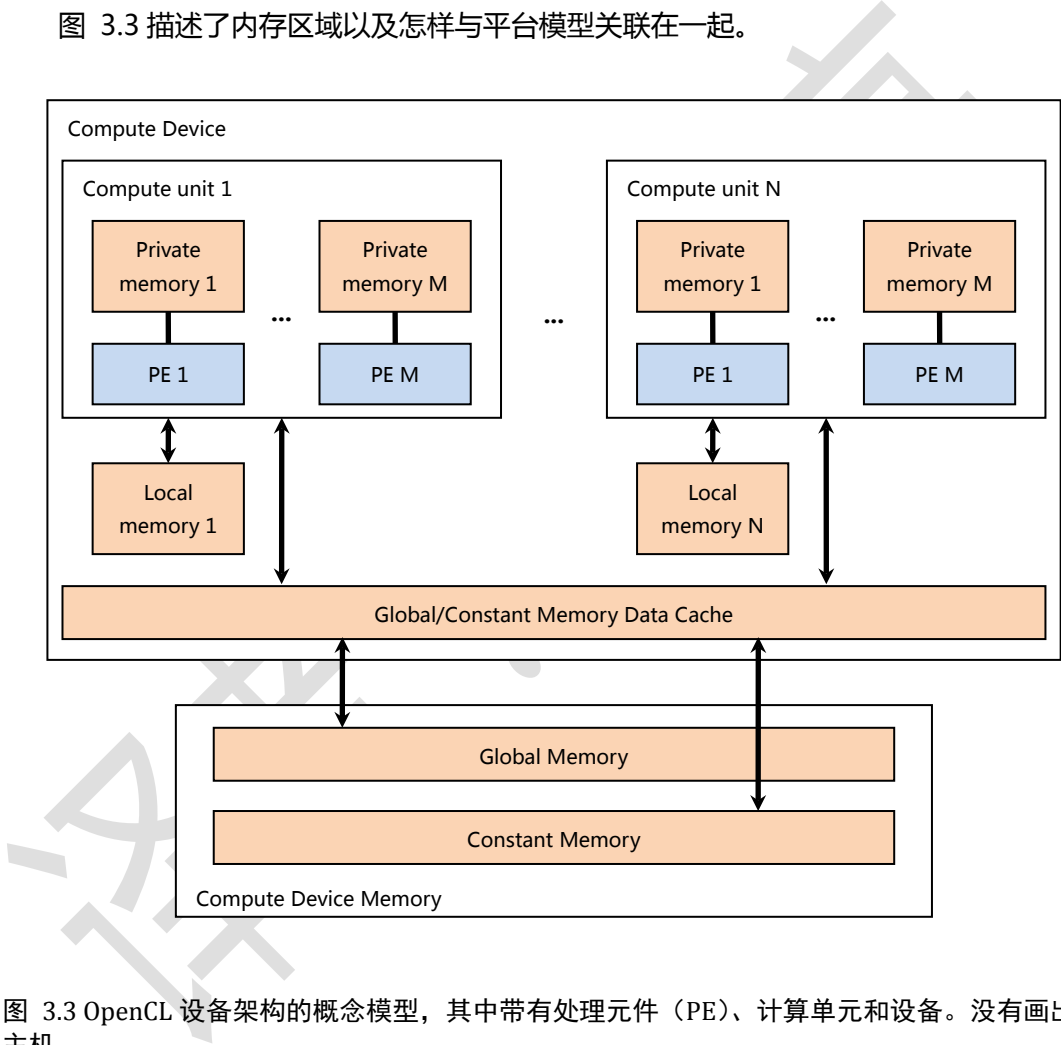


图 3.3 OpenCL 设备架构的概念模型，其中带有处理元件（PE）、计算单元和设备。没有画出宿主机。

宿主机上运行的应用，使用 OpenCL API 在全局内存中创建内存对象，并将内存命令（节 3.2.1 有所描述）入队以操作这些内存对象。

大多数情况下，宿主机和 OpenCL 设备的内存模型是互相独立的。这是由一个必要性来决定的，那就是假定宿主机是在 OpenCL 以外定义的。然而，有时它们确实需要交互；有两种途径：显式拷贝数据、映射和解映射一个内存对象的区域。

要想显式拷贝数据，宿主机会将命令入队，用以在内存对象和主机内存间传输数据。这些内存传输命令可能是阻塞的，也可能是非阻塞的。对于阻塞的内存传输，一旦主机上的相关内存资源可以安全的使用，OpenCL 函数调用会立刻返回。而对于非阻塞的内存传输，一旦命令入队，OpenCL 函数调用就会返回，无视主机内存是否可以安全使用。

主机和 OpenCL 内存对象交互时，其映射、解映射的方法允许主机将内存对象的区域映射到自己的地址空间中。内存映射命令可能是阻塞的、也可能是非阻塞的。一旦内存对象的区域被映射了，主机就可以读写这块区域。当主机完成对这块映射区域的访问（读和/或写）后，主机就会将这块区域解映射。

3.3.1 内存一致性

OpenCL 使用一个较宽松的一致性内存模型；即，不保证一个工作项所看到的内存状态跟其他工作项所看到的在所有时刻都是一致的。

在一个工作项内部，内存具有装载/存储的一致性。在一个工作组隔层上，单个工作组中的工作项之间，局部内存是一致的。全局内存也是如此，但是不保证执行相同内核的不同工作组间的内存一致性。

对于由入队的命令所共享的内存对象，其内存一致性由一个同步点来强制实施。

3.4 编程模型

OpenCL 执行模型支持**数据并行编程模型**和**任务并行编程模型**，同时也支持这两种模型的混合体。用以驱动 OpenCL 的设计的首要模型是数据并行。

3.4.1 数据并行编程模型

在数据并行编程模型中，一个指令序列会应用到一个内存对象的多个元素上，就按照这个指令序列来定义一个计算。与 OpenCL 执行模型相关联的索引空间定义了工作项，以及数据怎样映射到工作项上。在一个严格的数据并行模型中，工作项和内存对象的元素间有一对一的映射关系，内核可以在上面并行执行。对于数据并行编程模型，OpenCL 实现了一个较宽松的版本，其中不要求严格的一对一的映射。

OpenCL 提供了一个分级的数据并行编程模型。有两种途径来指定这种分层的细分。在显式模型中，程序员定义并行执行的工作项的总数，和这些工作项怎样划分成工作组。在隐式模型中，程序员仅指定前者，后者由 OpenCL 的实现来管理。

3.4.2 任务并行编程模型

在 OpenCL 的任务并行编程模型中，内核的单个实例在执行时是独立于任何索引空间的。这在逻辑上等同于在一个计算单元上执行一个内核，其工作组中只有单个工作项。在这种模型下，用户用以下方式表示并行：

- ✚ 使用设备所实现的矢量数据类型。
- ✚ 多个任务入队，和/或
- ✚ 多个原生内核入队，这些原生内核是使用一个与 OpenCL 正交的编程模型所开发的。

3.4.3 同步

在 OpenCL 中，有两个领域的同步：

- ✚ 同一工作组中的工作项
- ✚ 命令队列中处在同一个上下文中的命令

单个工作组内的工作项之间的同步是用工作组隔层来完成的。对于一个工作组内的所有工作项来说，任意一个要想在隔层外继续执行，所有工作项都必须先执行这个隔层。注意，执行内核的工作组中的所有工作项都必须碰到工作组隔层，否则全然不通。工作组之间没有同步机制。

命令队列中的命令间的同步点是：

- ✚ 命令队列隔层。命令队列隔层保证：所有原来排队的命令都执行完毕，并且对内存对象的所有更新，对于后续命令来说，在其开始执行之前，都是可见的。这个隔层只能在单个命令队列中的命令间进行同步。
- ✚ 等待一个事件。排队命令中的所有 OpenCL API 函数都会返回一个事件，用来标识这个命令和其更新的内存对象。如果某个后续命令正在等待那个事件，可以保证在其开始执行之前，那些内存对象的更新都是可见的。

3.5 OpenCL 框架

OpenCL 框架允许应用将宿主机和一个或多个 OpenCL 设备作为单个异构并行计算机系统来使用。这个框架包含以下组件：

- ✚ OpenCL 平台层：平台层允许宿主机程序发现 OpenCL 设备及其能力并创建上下文。

- ✚ OpenCL 运行时：一旦创建了上下文，运行时允许宿主机程序操控它。
- ✚ OpenCL 编译器：OpenCL 编译器可以创建包含 OpenCL 内核的可执行程序。此编译器所实现的 OpenCL C 编程语言支持 ISO C99 语言的一个子集，并带有并行扩展。

4 OpenCL 平台层

本节将介绍 OpenCL 平台层，它实现了平台特有的特性，允许应用来查询 OpenCL 设备、设备配置信息，而且可以创建 OpenCL 上下文来使用一个或多个设备。

4.1 查询平台信息

可以使用下面的函数获取可用平台的列表。

```
cl_int clGetPlatformIDs (cl_uint num_entries,
                        cl_platform_id *platforms,
                        cl_uint *num_platforms)
```

- ✚ *num_entries* 是可以加入 *platforms* 的 *cl_platform_id* 表项的数目。如果 *platforms* 不是 NULL，*num_entries* 必须大于 0。
- ✚ *platforms* 会返回所找到的 OpenCL 平台的列表。*platforms* 中 *cl_platform_id* 的值可以用来标识一个特定的 OpenCL 平台。如果 *platforms* 是 NULL，则被忽略。所返回的 OpenCL 平台的数目是 *num_entries* 和实际可用数目中较小的那个。
- ✚ *num_platforms* 返回实际可用的 OpenCL 平台的数目。如果 *num_platforms* 是 NULL，则被忽略。

如果 *num_entries* 等于 0 且 *platforms* 不是 NULL 或者 *num_platforms* 和 *platforms* 都是 NULL，**clGetPlatformIDs** 会返回 CL_INVALID_VALUE；而如果执行成功，则返回 CL_SUCCESS。

这个函数

```
cl_int clGetPlatformInfo (cl_platform_id platform,
                        cl_platform_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret)
```

获取 OpenCL 平台的特定信息。表 4.1 列出了可以用 **clGetPlatformInfo** 来查询的信息。

- ✚ *platform* 是 **clGetPlatformIDs** 所返回的一个平台 ID 或者是 NULL，如果是 NULL，

则其行为依赖于具体实现。

- ✚ *param_name* 是一个枚举，标识要查询的平台信息。它可以在表 4.1 所列值中任意选择。
- ✚ *param_value* 是一个内存指针，所指内存用来存放返回的值，此值是与给定的 *param_name* 相对应的一个合适的值（由表 4.1 所指定）。如果 *param_value* 是 NULL，则被忽略。
- ✚ *param_value_size* 用来指定 *param_value* 所指内存块的字节数。其值必须 \geq 返回类型（由表 4.1 指定）的大小。
- ✚ *param_value_size_ret* 返回 *param_value* 所查询的数据的实际大小（字节数）。如果 *param_value_size_ret* 是 NULL，则被忽略。

表 4.1 OpenCL 平台查询

cl_platform_info	返回类型	描述
CL_PLATFORM_PROFILE	char[] ¹	OpenCL 简档字符串。返回实现所支持的简档名称。所返回的简档名称是下列字符串中的一个： FULL_PROFILE——如果实现支持 OpenCL 规范（核心规范所定义的功能，且不需要支持任何扩展）。 EMBEDDED_PROFILE——如果实现支持 OpenCL 嵌入式简档。嵌入式简档被定义为所对应版本 OpenCL 的一个子集。OpenCL 1.0 所对应的嵌入式简档在第 10 节中有所描述。
CL_PLATFORM_VERSION	char[]	OpenCL 版本字符串。返回实现所支持的 OpenCL 版本。版本字符串格式如下： <i>OpenCL</i> <space> <major_version.minor_version> <space> <platform-specific information> 所返回的 <i>major_version.minor_version</i> 的值将是 1.0。
CL_PLATFORM_NAME	char[]	平台的名字
CL_PLATFORM_VENDOR	char[]	平台供应商
CL_PLATFORM_EXTENSIONS	char[]	返回平台所支持的扩展名字的列表，以空格来分隔（扩展名字本身不包含空格）。与这个平台相关的所有设备都要支持此处定义的扩展。

clGetPlatformInfo 如果执行成功，则返回 CL_SUCCESS。如果 *platform* 不是一个有效的平台，则返回 CL_INVALID_PLATFORM²。如果 *param_name* 不在被支持之列，或者 *param_value_size* 所指定的字节数 < 表 4.1 所指定的返回类型的大小，且 *param_value* 不

¹ 如果返回的被查询的信息类型是 char[]，则 OpenCL 查询函数会返回一个以 null 结尾的字符串。

² OpenCL 规范没有描述 API 调用所返回的错误码的优先顺序。

是 NULL，则返回 CL_INVALID_VALUE。

4.2 查询设备

可用设备列表可以用如下函数获取：

```
cl_int clGetDeviceIDs3 (cl_platform_id platform,
                        cl_device_type device_type,
                        cl_uint num_entries,
                        cl_device_id *devices,
                        cl_uint *num_devices)
```

- ✚ *platform* 是 **clGetPlatformIDs** 所返回的平台 ID 或者是 NULL。如果是 NULL，则其行为依赖于具体实现。
- ✚ *device_type* 是位域，用来标识 OpenCL 设备的类型。这个设备类型可以用来指定要查询什么类型的 OpenCL 设备。表 4.2 列出了 *device_type* 所有合法值。

表 4.2 OpenCL 设备类型列表

cl_device_type	描述
CL_DEVICE_TYPE_CPU	宿主处理器。在其上运行 OpenCL 实现，是单核或多核 CPU。
CL_DEVICE_TYPE_GPU	GPU。这意味着此设备也可以用来加速一个 3D API（如 OpenGL 或 DirectX）。
CL_DEVICE_TYPE_ACCELERATOR	OpenGL 专用加速器（如 IBM 的 CELL Blade）。这些设备通过外围内联（如 PCIe）与宿主处理器通信。
CL_DEVICE_TYPE_DEFAULT	系统中默认的 OpenCL 设备。
CL_DEVICE_TYPE_ALL	系统中所有可用的 OpenCL 设备。

- ✚ *num_entries* 是 *cl_device* 表项的数目，此表项可以添进 *devices*。如果 *devices* 不是 NULL，则 *num_entries* 必须大于 0。
- ✚ *devices* 返回一个列表，其中存放所找到的 OpenCL 设备。*devices* 中的 *cl_device_id* 的值可以用来标识一个特定的 OpenCL 设备。如果参数 *devices* 是 NULL，则忽略此参数。所返回的 OpenCL 设备的数目是如下两个数目中较小的一个：*num_entries*，类型为 *device_type* 的 OpenCL 设备的数目。
- ✚ *num_devices* 返回与 *device_type* 相匹配的可用 OpenCL 设备的数目。如果 *num_devices* 是 NULL，则忽略此参数。

³ **clGetDeviceIDs** 可能返回 platform 中所有与 *device_type* 匹配的实际物理设备，也可能是其中一个子集。

对于 **clGetDeviceIDs** 来说，如果 *device_type* 不是一个合法值，则返回 `CL_INVALID_DEVICE_TYPE`；如果 *num_entries* 等于 0 而且 *devices* 不是 NULL，或者 *num_devices* 和 *devices* 都是 NULL，则返回 `CL_INVALID_VALUE`；如果没有发现与 *device_type* 相匹配的 OpenCL 设备，则返回 `CL_DEVICE_NOT_FOUND`。如果执行成功，则返回 `CL_SUCCESS`。

对于 **clGetDeviceIDs** 返回的 OpenCL 设备，应用可以查询其能力。应用可以据其来决定使用哪些设备。

这个函数

```
cl_int clGetDeviceInfo (cl_device_id device,
                       cl_device_info param_name,
                       size_t param_value_size,
                       void *param_value,
                       size_t *param_value_size_ret)
```

可以获取一个 OpenCL 设备的特定信息。可以用 **clGetDeviceInfo** 来查询的信息如表 4.3 所示。

- ✚ *device* 是 **clGetDeviceIDs** 所返回的一个设备。
- ✚ *param_name* 是一个枚举，用来标识所查询的设备信息，可以在表 4.3 所列数值中选取。
- ✚ *param_value* 是一个指针，所指内存中存储有给定的 *param_name* (由表 4.3 指定) 所对应的值。如果 *param_value* 是 NULL，则忽略。
- ✚ *param_value_size* 指定了 *param_value* 所指内存的字节数，其值必须 \geq 表 4.3 所指定的返回类型的大小。
- ✚ *param_value_size_ret* 返回 *param_value* 所查询的数据的实际大小。如果 *param_value_size_ret* 是 NULL，则忽略。

表 4.3 OpenCL 设备查询

cl_device_info	返回类型	描述
CL_DEVICE_TYPE	cl_device_type	OpenCL 设备类型。当前所支持的值有： CL_DEVICE_TYPE_CPU、 CL_DEVICE_TYPE_GPU、 CL_DEVICE_TYPE_ACCELERATOR、 CL_DEVICE_TYPE_DEFAULT 或它们的组合。
CL_DEVICE_VENDOR_ID	cl_uint	唯一设备供应商标识符。例如可以是 PCIe ID。
CL_DEVICE_MAX_COMPUTE_UNITS	cl_uint	OpenCL 设备上的并行计算核心的数目。最小值是 1。
CL_DEVICE_MAX_WORK_ITEM	cl_uint	数据并行执行模型中所用的全局和局部工

M_DIMENSIONS		作项 ID 的最大维数。(参见 clEnqueueNDRangeKernel)。最小值是 3。
CL_DEVICE_MAX_WORK_ITEM_SIZES	size_t []	给 clEnqueueNDRangeKernel 所指定的工作组中每个维度上工作项的最大数目。 返回 n 个 size_t 表项。其中 n 是查询 CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS 所返回的值。 最小值是(1, 1, 1)。
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t	一个执行内核(使用数据并行执行模型)的工作组中所能存放的工作项的最大数目。(参见 clEnqueueNDRangeKernel)。最小值是 1。
CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE	cl_uint	可以放入矢量中的内建标量类型所期望的原生矢量的宽度。矢量宽度定义为可以存储到矢量中的标量元素的数目。 如果不支持扩展 cl_khr_fp64 , 则 CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE 必须返回 0。
CL_DEVICE_MAX_CLOCK_FREQUENCY	cl_uint	设备所配置的最高时钟频率(单位 MHz)。
CL_DEVICE_ADDRESS_BITS	cl_uint	设备地址空间的默认大小。其值是无符号整型, 目前所支持的值是 32 和 64。
CL_DEVICE_MAX_MEM_ALLOC_SIZE	cl_ulong	内存对象的最大字节数。最小值是 $\max(\text{CL_DEVICE_GLOBAL_MEM_SIZE}/4, 128*1024*1024)$ 。
CL_DEVICE_IMAGE_SUPPORT	cl_bool	如果 OpenCL 设备支持图像, 就是 CL_TRUE, 否则就是 CL_FALSE。
CL_DEVICE_MAX_READ_IMAGE_ARGS	cl_uint	内核可以同时读取的图像对象的最大数目。 如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE, 最小值是 128。
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	cl_uint	内核可以同时写入的图像对象的最大数目。 如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE, 最小值是 8。
CL_DEVICE_IMAGE2D_MAX_WIDTH	size_t	2D 图像的最大宽度(单位: 像素)。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE, 最小值是 8192。
CL_DEVICE_IMAGE2D_MAX_HEIGHT	size_t	2D 图像的最大高度(单位: 像素)。如果

HEIGHT		CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE , 最小值是 8192。
CL_DEVICE_IMAGE3D_MAX_WIDTH	size_t	3D 图像的最大宽度 (单位: 像素)。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE , 最小值是 2048。
CL_DEVICE_IMAGE3D_MAX_HEIGHT	size_t	3D 图像的最大高度 (单位: 像素)。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE , 最小值是 2048。
CL_DEVICE_IMAGE3D_MAX_DEPTH	size_t	3D 图像的最大深度 (单位: 像素)。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE , 最小值是 2048。
CL_DEVICE_MAX_SAMPLERS	cl_uint	一个内核可以使用的采样器的最大数目。关于采样器的细节描述可参见节 6.11.8。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE , 则最小值是 16。
CL_DEVICE_MAX_PARAMETER_SIZE	size_t	传给内核的参数最大字节数。最小值是 256。
CL_DEVICE_MEM_BASE_ADDR_ALIGN	cl_uint	描述分配任意一个内存对象时, 其基地址的对齐方式 (单位 bit)。
CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE	cl_uint	最小字节对齐的数目, 可以由所有数据类型使用。
CL_DEVICE_SINGLE_FP_CONFIG	cl_device_fp_config	<p>描述设备的单精度浮点能力。这是一个位域, 描述下列值中的一个或多个:</p> <p>CL_FP_DENORM——支持去规格化数</p> <p>CL_FP_INF_NAN——支持 INF 和 QNaN (quiet NaN)</p> <p>CL_FP_ROUND_TO_NEAREST——支持舍入到最近偶数 (round to nearest even)</p> <p>CL_FP_ROUND_TO_ZERO——支持向零舍入 (round to zero)</p> <p>CL_FP_ROUND_TO_INF——支持向正负无穷舍入 (round to +ve and -ve infinity)</p> <p>CL_FP_FMA——支持 IEEE 754-2008 熔加运算 (fused multiply-add, FMA)</p> <p>要求至少要支持的浮点能力是:</p> <p>CL_FP_ROUND_TO_NEAREST CL_FP_INF_NAN。</p>
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	cl_device_mem_cache_type	所支持的全局内存缓存的类型。合法值有: CL_NONE、CL_READ_ONLY_CACHE 和 CL_READ_WRITE_CACHE。

CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE	cl_uint	全局内存缓存行 (cache line) 的字节数。
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE	cl_ulong	全局内存缓存的字节数。
CL_DEVICE_GLOBAL_MEM_SIZE	cl_ulong	全局设备内存的字节数。
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	cl_ulong	所分配常量缓存的最大字节数。最小值是 64KB。
CL_DEVICE_MAX_CONSTANT_ARGS	cl_uint	一个内核中最多能有多少参数带限定符 <code>_constant</code> 。最小值是 8。
CL_DEVICE_LOCAL_MEM_TYPE	cl_device_local_mem_type	所支持的局部内存的类型。可以设置成 <code>CL_LOCAL</code> (意指专用的局部内存, 如 SRAM) 或 <code>CL_GLOBAL</code> 。
CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong	局部内存区的大小。最小值是 16KB。
CL_DEVICE_ERROR_CORRECTION_SUPPORT	cl_bool	如果设备实现了对内存、缓存、寄存器等的错误修正, 则是 <code>CL_TRUE</code> 。如果设备没有实现错误修正, 则是 <code>CL_FALSE</code> 。这可能是 OpenCL 的某个客户端的需求。
CL_DEVICE_PROFILING_TIMER_RESOLUTION	size_t	描述设备定时器的分辨率。单位是纳秒。其细节请参见节 5.9。
CL_DEVICE_ENDIAN_LITTLE	cl_bool	如果 OpenCL 设备是小端设备, 则为 <code>CL_TRUE</code> , 否则为 <code>CL_FALSE</code> 。
CL_DEVICE_AVAILABLE	cl_bool	如果设备可用, 则为 <code>CL_TRUE</code> , 否则就是 <code>CL_FALSE</code> 。
CL_DEVICE_COMPILER_AVAILABLE	cl_bool	如果 OpenCL 的实现没有可用编译器来编译程序源码, 则为 <code>CL_FALSE</code> , 否则是 <code>CL_TRUE</code> 。只有 OpenCL ES 简档中才能是 <code>CL_FALSE</code> 。
CL_DEVICE_EXECUTION_CAPABILITIES	cl_device_exec_capabilities	描述设备的执行能力。这是一个位域, 有以下值: <code>CL_EXEC_KERNEL</code> ——此 OpenCL 设备可以执行 OpenCL 内核。 <code>CL_EXEC_NATIVE_KERNEL</code> ——此 OpenCL 设备可以执行原生内核。 其能力至少要为: <code>CL_EXEC_KERNEL</code> 。
CL_DEVICE_QUEUE_PROPERTIES	cl_command_queue_properties	描述设备所支持的命令队列属性。这是一个位域, 包含以下值: <code>CL_QUEUE_OUT_OF_ORDER_EXEC_MODE</code>

		DE_ENABLE CL_QUEUE_PROFILING_ENABLE 这些属性在表 5.1 中有所描述。 其能力至少要为： CL_QUEUE_PROFILING_ENABLE。
CL_DEVICE_PLATFORM	cl_platform_id	此设备所关联的平台
CL_DEVICE_NAME	char[]	设备名称字符串
CL_DEVICE_VENDOR	char[]	供应商名称字符串
CL_DRIVER_VERSION	char[]	OpenCL 软件驱动版本字符串，形如 <i>major_number.minor_number</i> 。
CL_DEVICE_PROFILE ⁴	char[]	OpenCL 简档字符串。返回设备所支持的简档名称。所返回的简档名称可以是如下字符串中的一个： FULL_PROFILE——如果设备支持 OpenCL 规范（核心规范所定义的功能，不需要支持任何扩展）。 EMBEDDED_PROFILE——如果设备支持 OpenCL 嵌入式简档。
CL_DEVICE_VERSION	char[]	OpenCL 版本字符串。返回设备所支持的 OpenCL 版本。形如： <i>OpenCL <space> <major_version.minor_version> <space> <vendor-specific information></i> 所返回的 <i>major_version.minor_version</i> 的值将是 1.0。
CL_DEVICE_EXTENSIONS	char[]	返回一个扩展名称列表，以空格来分隔（扩展名称本身不包含空格）。 所返回的扩展名称列表当前可能包括下列已获批准的扩展名称中的一个或多个： cl_khr_fp64 cl_khr_select_fprounding_mode cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics

⁴ 平台简档返回 OpenCL 框架所实现的简档。如果返回的是 FULL_PROFILE，则 OpenCL 框架支持 FULL_PROFILE 的设备，可能也支持 EMBEDDED_PROFILE 的设备。编译器必须对所有设备可用，即 CL_DEVICE_COMPILER_AVAILABLE 必须是 CL_TRUE。如果平台简档是 EMBEDDED_PROFILE，则只支持 EMBEDDED_PROFILE 的设备。

		<div>cl_khr_int64_base_atomics</div> <div>cl_khr_int64_extended_atomics</div> <div>cl_khr_3d_image_writes</div> <div>cl_khr_byte_addressable_store</div> <div>cl_khr_fp16</div> <div>cl_khr_gl_sharing</div> <div>对于这些扩展的详细描述请参见节 9。</div>
--	--	--

如果函数执行成功，**clGetDeviceInfo** 会返回 CL_SUCCESS。如果设备无效，则返回 CL_INVALID_DEVICE。如果 *param_name* 不被支持，或者 *param_value_size* 所指定的字节数 < 表 4.3 所指定的返回类型的大小且 *param_value* 不是 NULL，则返回 CL_INVALID_VALUE。

4.3 上下文

这个函数

```
cl_context clCreateContext (  
    const cl_context_properties *properties,  
    cl_uint num_devices,  
    const cl_device_id *devices,  
    void (*pfn_notify)(const char *errinfo,  
                       const void *private_info, size_t cb,  
                       void *user_data),  
    void *user_data,  
    cl_int *errcode_ret)
```

会创建一个 OpenCL 上下文。一个 OpenCL 上下文与一个或多个设备一起创建。OpenCL 运行时会使用上下文来管理命令队列、内存、程序和内核等对象，并在上下文所指定的一个或多个设备上执行内核。

properties 指向一个列表，其中有上下文属性名称及其对应的值。每个属性名称后面紧跟其对应的期望值。此列表以 0 结尾。表 4.4 列出了所支持的属性。*properties* 为 NULL 时，其行为依赖于具体实现。

表 4.4 clCreatContext 所支持的属性列表

cl_context_properties 枚举	属性值	描述
CL_CONTEXT_PLATFORM	cl_platform_id	指定要使用的平台

num_devices 是参数 *devices* 中设备的数目。

*devices*是一个指针，指向**clGetDeviceIDs**所返回的设备的列表⁵。

*pfn_notify*是应用所注册的一个回调函数。OpenCL 的实现可以用这个回调函数来报告此上下文中所发生的错误。OpenCL 的实现可能会异步调用此回调函数。应用负责保证回调函数的线程安全。这个回调函数的参数是：

- ✚ *errinfo* 是一个指针，指向一个错误字符串。
- ✚ *private_info* 和 *cb* 会提供一个指向二进制数据的指针，这些数据由 OpenCL 的实现所返回，可以用来记录一些附加信息来帮助调试错误。
- ✚ *user_data* 指向用户所提供的数据。

如果 *pfn_notify* 是 NULL，就是没有注册回调函数。

当 *pfn_notify* 被调用时，会将 *user_data* 作为参数 *user_data* 传递给 *pfn_notify*。*user_data* 可以是 NULL。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL，就不会返回错误码。

如果成功创建了上下文，**clCreateContext** 会返回一个有效的非 0 上下文，并且将 *errcode_ret* 设置成 CL_SUCCESS。当返回 NULL 时，*errcode_ret* 会返回下列错误值：

- ✚ CL_INVALID_PLATFORM，如果 *properties* 是 NULL 而且没有可选择的平台，或者 *properties* 中所指定的平台无效。
- ✚ CL_INVALID_VALUE，如果 *properties* 中的上下文名称不在支持之列，或者虽然支持但是值无效，或者属性名称由重复。
- ✚ CL_INVALID_VALUE，如果 *devices* 为 NULL。
- ✚ CL_INVALID_VALUE，如果 *num_devices* 等于 0。
- ✚ CL_INVALID_VALUE，如果 *pfn_notify* 是 NULL，但 *user_data* 不是 NULL。
- ✚ CL_INVALID_DEVICE 如果 *devices* 中有无效设备或者没有关联到指定的平台上。
- ✚ CL_DEVICE_NOT_AVAILABLE，如果 *devices* 中的某个设备当前无效（即使这个设备是由 **clGetDeviceIDs** 返回的）。
- ✚ CL_OUT_OF_HOST_MEMORY，如果在为宿主机上的 OpenCL 实现分配其所需资源时失败。

这个函数

<code>cl_context clCreateContextFromType⁶ (</code>

⁵ 会忽略重复的设备。

```

cl_context_properties *properties,
cl_device_type device_type,
void (*pfn_notify)(const char *errinfo,
                   const void *private_info,
                   size_t cb,
                   void *user_data),

void *user_data,
cl_int *errcode_ret)

```

会根据设备类型创建一个 OpenCL 上下文，此设备类型用来标识要使用的设备。

properties 指向一个列表，其中列出了上下文属性名称及其对应的值。每个属性名称后面紧跟其对应的期望值。此列表以 0 结尾。表 4.4.列出了所支持的属性。*properties* 为 NULL 时，其行为依赖于具体实现。

device_type 是位域，用来标识设备的类型。在节 4.2 中的表 4.2 中对其有所描述。

pfn_notify 和 *user_data* 在 **clCreateContext** 中有所描述。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL，就不会返回错误码。

如果成功创建了上下文，**clCreateContextFromType** 会返回一个有效的非 0 上下文，且 *errcode_ret* 会被设置成 CL_SUCCESS。如果返回的是 NULL，则 *errcode_ret* 可能会是下列错误值：

- ✚ CL_INVALID_VALUE，如果 *properties* 是 NULL 而且没有可选择的平台，或者 *properties* 所指定的平台无效。
- ✚ CL_INVALID_VALUE，如果 *properties* 中的上下文属性名称不在支持之列，或者虽然支持但是值无效，或者属性名称由重复。
- ✚ CL_INVALID_VALUE，如果 *pfn_notify* 是 NULL，但 *user_data* 不是 NULL。
- ✚ CL_INVALID_DEVICE_TYPE，如果 *device_type* 无效。
- ✚ CL_DEVICE_NOT_AVAILABLE，如果当前没有有效且与 *device_type* 相匹配的设备。
- ✚ CL_DEVICE_NOT_FOUND，如果没有找到与 *device_type* 相匹配的设备。
- ✚ CL_OUT_OF_HOST_MEMORY，如果在为宿主机上的 OpenCL 实现分配其所需资源时失败。

这个函数

⁶ **clCreateContextfromType** 可能返回平台中现有的与 *device_type* 相匹配的所有实际物理设备，也可能是其中一个子集。

```
cl_int clRetainContext (cl_context context)
```

会增大 *context* 的引用计数。如果执行成功, **clRetainContext** 会返回 CL_SUCCESS。如果 *context* 无效, 则会返回 CL_INVALID_CONTEXT。

clCreateContext 和 **clCreateContextFromType** 会执行隐式的保留。这对第三方库非常有用, 使其可以直接获取由应用传给它们的上下文。然而, 应用可能会在没有通知库的情况下删除上下文。允许函数附到上下文上 (如保留) 或释放上下文, 这样, 在库所使用的上下文不再有效时就不会出问题。

这个函数

```
cl_int clReleaseContext (cl_context context)
```

会减小 *context* 的引用计数。如果执行成功, **clReleaseContext** 会返回 CL_SUCCESS。如果 *context* 无效, 则会返回 CL_INVALID_CONTEXT。

当 *context* 的引用计数减小到 0, 且所有附到其上的对象 (如内存对象、命令队列) 都被释放了的时候, *context* 会被删除。

这个函数

```
cl_int clGetContextInfo (cl_context context,
                          cl_context_info param_name,
                          size_t param_value_size,
                          void *param_value,
                          size_t *param_value_size_ret)
```

可以用来查询一个上下文的相关信息。

context 指定所要查询的 OpenCL 上下文。

param_name 是一个枚举, 指定所要查询的信息。

param_value 是一个指针, 指向所返回的查询结果。如果 *param_value* 是 NULL, 则被忽略。

param_value_size 指定 *param_value* 所指内存的字节数。它必须大于等于表 4.5 中所描述的返回类型的大小。

param_value_size_ret 返回 *param_value* 所查询数据的实际字节数。如果 *param_value_size_ret* 是 NULL, 则被忽略。

表 4.5 列出了 *param_name* 的值和 **clGetContextInfo** 返回的 *param_value* 中的信

息。

表 4.5 clGetContextInfo 所支持的 param_names 的列表

cl_context_info	返回类型	param_value 中的信息
CL_CONTEXT_REFERENCE_COUNT ⁷	cl_unit	<i>context</i> 的引用计数
CL_CONTEXT_DEVICES	cl_device_id[]	<i>context</i> 中的设备列表
CL_CONTEXT_PROPERTIES	cl_context_properties[]	clCreateContext 或 clCreateContextFromType 所指定的参数 <i>properties</i> 。 如果此参数不是 NULL, 则必须将其值返回。而如果此参数是 NULL, 实现可以选择将 <i>param_value_size_ret</i> 置为 0, 即没有返回属性值, 或者仅在 <i>param_value points</i> 所指内存中返回属性值 0 (0 用作属性列表的终止标记)。

如果执行成功, **clGetContextInfo** 会返回 CL_SUCCESS。如果 *context* 无效, 则返回 CL_INVALID_CONTEXT。如果 *param_names* 不在被支持之列, 或者 *param_value_size* 所指定的大小 < 表 4.5 所指定的返回类型的大小, 而且 *param_value* 不是 NULL, 则返回 CL_INVALID_VALUE。

5 OpenCL 运行时

本节中, 我们将描述用来管理 OpenCL 对象 (如命令队列、内存对象、程序对象、执行 `_kernel` 函数的内核对象) 的 API 调用, 和用来将命令入队的调用, 如执行内核, 读写内存对象。

5.1 命令队列

OpenCL 对象, 如内存对象、程序对象和内核对象, 都是用上下文来创建的。对这些对象的操作都是通过命令队列来执行的。可以用命令队列将一系列操作 (称作命令) 按顺序排队。如果有多个命令队列, 则允许应用将多个命令分别排队而无需同步。注意, 在这些对象被共享之前, 这个说法永远成立。如果多个命令队列间共享对象, 则需要应用进行适当的同步。221 附录 A 对此有所描述。

⁷ 所返回的引用计数会立刻过时, 不适合应用中通常的用途。提供这个特性主要是为了定位内存泄露。

函数

```
cl_command_queue clCreateCommandQueue (  
    cl_context context,  
    cl_device_id device,  
    cl_command_queue_properties properties,  
    cl_int *errcode_ret)
```

用来在特定设备上创建一个命令队列。

context 必须是一个有效的上下文。

表 5.1 cl_command_queue_property 支持列表

命令队列属性	描述
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	用来确定命令队列中的命令是顺序执行还是乱序执行。如果设置了，就乱序执行，否则顺序执行。 详细描述请参考节 5.8。
CL_QUEUE_PROFILING_ENABLE	使能或禁止对命令队列中命令的 profile。如果设置了，则使能 profile，否则，禁止 profile。 详细描述请参考节 5.3。

device 必须是与 *context* 关联的一个设备。它可以是使用 **clCreateContext** 创建 *context* 时所指定的设备列表中的一个。或者具有使用 **clCreateContextFromType** 创建 *context* 时所指定的设备类型。

properties 指定命令队列的一系列属性。它是一个位域，如表 5.1 所示。只能对表 5.1 中所列属性进行设置，否则认为 *properties* 无效。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL，就不会返回错误码。

如果创建成功，**clCreateCommandQueue** 会返回一个非 0 的命令队列，同时将 *errCode_ret* 置为 CL_SUCCESS。否则会返回 NULL，并将 *errCode_ret* 置为下列值中的一个：

- CL_INVALID_CONTEXT，如果 *context* 无效。
- CL_INVALID_DEVICE，如果 *device* 无效或者没有与 *context* 关联在一起。
- CL_INVALID_VALUE，如果 *properties* 的值无效。
- CL_INVALID_QUEUE_PROPERTIES，如果 *properties* 的值有效但是此设备不支持。
- CL_OUT_OF_HOST_MEMORY，如果宿主机上的 OpenCL 实现在分配资源时失败。

函数

```
cl_int clRetainCommandQueue (cl_command_queue command_queue)
```

会增加 *command_queue* 的引用计数。如果执行成功，**clRetainCommandQueue** 会返回 CL_SUCCESS。如果 *command_queue* 不是一个有效的命令队列，则返回 CL_INVALID_COMMAND_QUEUE。

clCreateCommandQueue 会隐式执行保留。这对第三方库非常有用，使其可以直接获取由应用传给它们的命令队列。然而，应用可能会在没有通知库的情况下删除命令队列。允许函数附加到命令队列上（如保留）或释放命令队列，这样，在库所使用的命令队列不再有效时就不会出现问题。

函数

```
cl_int clReleaseCommandQueue (cl_command_queue command_queue)
```

会减小 *command_queue* 的引用计数。如果执行成功，**clReleaseCommandQueue** 会返回 CL_SUCCESS。如果 *command_queue* 不是一个有效的命令队列，则返回 CL_INVALID_COMMAND_QUEUE。

当 *command_queue* 的引用计数变为 0，而且其中的所有命令（如执行内核，更新内存对象等）都执行完毕时，此命令队列会被删除。

函数

```
cl_int clGetCommandQueueInfo (cl_command_queue command_queue,  
                                cl_command_queue_info param_name,  
                                size_t param_value_size,  
                                void *param_value,  
                                size_t *param_value_size_ret)
```

可以用来查询一个命令队列的相关信息。

command_queue 指定要查询的命令队列。

param_name 指定要查询什么信息。

param_value 指向存储结果的内存。如果是 NULL，则忽略。

param_value_size 指定 *param_value* 所指内存的字节数。其大小必须 ≥ 表 5.2 中所列返回类型的大小。如果是 NULL，则忽略。

param_value_size_ret 返回 *param_value* 所查数据的实际字节数。如果是 NULL，则

忽略。

clGetCommandQueueInfo 所支持的 *param_name* 的值和 *param_value* 所返回的信息如表 5.2 所示。

表 5.2 clGetCommandQueueInfo 中 param_names 支持列表

cl_command_queue_info	返回类型	param_value 所返回的信息
CL_QUEUE_CONTEXT	cl_context	创建命令队列时所指定的上下文
CL_QUEUE_DEVICE	cl_device_id	创建命令队列时所指定的设备
CL_QUEUE_REFERENCE_COUNT ⁸	cl_unit	命令队列的引用计数
CL_QUEUE_PROPERTIES	cl_command_queue_properties	命令队列的当前属性。这些属性是由 clCreateCommandQueue 的参数 <i>properties</i> 所指定，且可以用 clSetCommandQueueProperty 进行修改。

如果执行成功，**clGetCommandQueueInfo** 返回 CL_SUCCESS。如果 *command_queue* 无效，则返回 CL_INVALID_COMMAND_QUEUE。如果 *param_name* 不在被支持之列，或者 *param_value_size* 的大小 < 表 5.2 所列返回类型的大小而且 *param_value* 不是 NULL，则返回 CL_INVALID_VALUE。

函数

```
cl_int clSetCommandQueueProperty (
    cl_command_queue command_queue,
    cl_command_queue_properties properties,
    cl_bool enable,
    cl_command_queue_properties *old_properties)
```

用来使能或禁止命令队列的属性。

command_queue 指定要查询的命令队列。

properties 指定 *command_queue* 的新属性。只有表 5.1 所列属性可以设置；否则认为 *properties* 无效。

enable 确定使能（如果 *enable* 是 CL_TRUE）或禁止（如果 *enable* 是 CL_FALSE）*properties* 中的属性值。属性值参见表 5.1。

⁸ 所返回的引用计数会立刻过时，不适合应用中通常的用途。提供这个特性是为了定位内存泄露。

old_properties 返回命令队列 (被 **clSetCommandQueueProperty** 改变) 之前的属性。如果是 NULL , 则忽略。

如表 5.1 所示, 属性 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE 决定了命令队列中命令的执行顺序。更改这个属性会导致 OpenCL 实现的阻塞, 直到 *command_queue* 中之前入队的命令全部完成。由于此操作代价太大, 除非绝对有必要, 否则不要更改此属性。

如果成功更新了命令队列的属性, **clSetCommandQueueProperty** 会返回 CL_SUCCESS。如果 *command_queue* 无效, 则返回 CL_INVALID_COMMAND_QUEUE。如果 *properties* 的值无效, 则返回 CL_INVALID_VALUE。如果此设备不支持 *properties* 的值, 则返回 CL_INVALID_QUEUE_PROPERTIES。

注意:

当使用某个设备的上下文和命令队列被创建后并且有命令入队, 这时此设备可能变成了不可用状态。这种情况下, 使用此上下文 (和命令队列) 的 OpenCL API 调用的行为依赖于具体实现。创建上下文时可以指定了用户的回调函数, 当设备变成不可用时, 可以用此回调函数的参数 *errinfo* 和 *private_info* 来记录相应的信息。

5.2 内存对象

内存对象分为两类: 缓冲对象和图像对象。缓冲对象存储一维元素集合, 而图像对象存储二维或三维的材质、帧缓冲或图像。

缓冲对象的元素可以是标量数据类型 (如 int , float)、矢量数据类型或用户定义的结构体。图像对象用来描述可用作材质或帧缓冲的缓冲区。图像对象的元素从一个预定义的图像格式列表选取。内存对象中最少要有一个元素。

缓冲对象和图像对象基本的不同点是:

- ✚ 缓冲对象中的元素顺序存储, 内核可以使用指针对其进行访问。而图像对象的存储格式对用户是不透明的, 不能用指针直接访问, 而要用 OpenCL C 语言提供的内建函数对其进行读写。
- ✚ 缓冲对象的存储格式与内核对其的访问方式是一致的。图像对象存储图像的数据格式与内核所用数据格式可能不同。内核中的图像元素通常是四元矢量 (每元可以是浮点数或带符号/无符号的整型)。内建函数读取图像后, 会将其由存储格式转换为四元矢量。同样的, 内建函数在写入图像前, 会将其由四元矢量转换为其存储格式, 如四个 8 位元素。

内存对象用 **cl_mem** 对象来表示。内核将内存对象作为输入，并输出一个或多个内存对象。

5.2.1 创建缓冲对象

缓冲对象由下列函数创建：

```
cl_mem clCreateBuffer (cl_context context,
                        cl_mem_flags flags,
                        size_t size,
                        void *host_ptr,
                        cl_int *errcode_ret)
```

context 是一个有效的 OpenCL 上下文，用来创建缓冲对象。

flags 是一个位域，用来指定分配和使用信息，如在哪个内存区域中分配缓冲对象以及怎样使用。表 5.3 列出了 *flags* 可能的值：

表 5.3 cl_mem_flags 支持列表

cl_mem_flags	描述
CL_MEM_READ_WRITE	内存对象可读可写，这是默认值。
CL_MEM_WRITE_ONLY	只能写不能读，对这种内存对象的读操作是未定义的。
CL_MEM_READ_ONLY	只能读不能写，对这种内存对象的写操作是未定义的。
CL_MEM_USE_HOST_PTR	只有 <i>host_ptr</i> 不是 NULL 时，这个标志位才有效。表示用 <i>host_ptr</i> 所指内存来存储内存对象。 允许 OpenCL 实现在设备内存中保存一份 <i>host_ptr</i> 所指内容的拷贝。当内核在设备上执行时可以使用这份拷贝。 如果多个缓冲对象由同一个 <i>host_ptr</i> 创建，或者有重叠区域，当 OpenCL 命令操作这些缓冲对象时，其结果未定义。
CL_MEM_ALLOC_HOST_PTR	这个标志标明应用想让 OpenCL 实现从宿主机的可访问内存中分配内存。 与 CL_MEM_USE_HOST_PTR 互斥。
CL_MEM_COPY_HOST_PTR	只有 <i>host_ptr</i> 不是 NULL 时，此标志才有效。它表明应用想让 OpenCL 实现为内存对象分配内存并从 <i>host_ptr</i> 所引用内存中拷贝数据。 与 CL_MEM_USE_HOST_PTR 互斥。 可以与 CL_MEM_ALLOC_HOST_PTR 一起使用，对用宿主机可访问内存（如 PCIe）分配的 cl_mem 对象进行初始化。

size 是所分配缓冲对象的字节数。

host_ptr 指向由应用所分配的缓冲数据。其大小必须 $\geq size$ 。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL，就不会返回错误码。

如果缓冲对象创建成功，**clCreateBuffer** 会返回一个非 0 的缓冲对象，且会将 *errcode_ret* 置为 CL_SUCCESS。否则，返回 NULL 并将 *errcode_ret* 置为下列值之一：

- ✚ CL_INVALID_CONTEXT，如果 *context* 无效。
- ✚ CL_INVALID_VALUE，如果 *flags* 无效。
- ✚ CL_INVALID_BUFFER_SIZE，如果 *size* 是 0，或者大于 *context* 中所有设备的 CL_DEVICE_MAX_MEM_ALLOC_SIZE（参见表 4.3）。
- ✚ CL_INVALID_HOST_PTR，如果 *host_ptr* 是 NULL 且 *flags* 中设置了 CL_MEM_USE_HOST_PTR 或 CL_MEM_COPY_HOST_PTR，或者 *host_ptr* 不是 NULL 但 *flags* 中设置了 CL_MEM_COPY_HOST_PTR 或 CL_MEM_USE_HOST_PTR。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE，如果为缓冲对象分配内存失败。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主机上的 OpenCL 实现分配其所需资源失败。

5.2.2 读、写和拷贝缓冲对象

下面两个函数会将两个命令入队，一个负责将缓冲对象读入宿主机内存，另一个可以将宿主机内存中的数据写入缓冲对象。

```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
                             cl_mem buffer,
                             cl_bool blocking_read,
                             size_t offset,
                             size_t cb,
                             void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
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)
```

command_queue 指容纳读写命令的命令队列。*command_queue* 和 *buffer* 必须创建自同一个 OpenCL 上下文。

buffer 指一个有效的缓冲对象。

blocking_read 和 *blocking_write* 指明读写操作是阻塞的（blocking）还是非阻塞的

(non-blocking)。

如果 *blocking_read* 是 CL_TRUE , 也就是说读命令是阻塞的 , 只有当缓冲对象被读取并拷贝到 *ptr* 所指内存后 , **clEnqueueReadBuffer** 才会返回。

如果 *blocking_read* 是 CL_FALSE , 也就是说读命令是非阻塞的 , **clEnqueueReadBuffer** 会将非阻塞的读命令入队并返回。只有当读命令完成后才能使用 *ptr* 所指内容。参数 *event* 返回一个事件对象 , 此对象可以用来查询读命令的执行状态。读命令完成后 , 应用就可以使用 *ptr* 所指内容了。

如果 *blocking_write* 是 CL_TRUE , OpenCL 实现会拷贝 *ptr* 所指数据并将写操作入队。**clEnqueueWriteBuffer** 返回后 , 应用可以继续使用 *ptr* 所指内存。

如果 *blocking_write* 是 CL_FALSE , OpenCL 实现将使用 *ptr* 执行非阻塞写操作。由于非阻塞 , 实现可以立即返回。所以在其返回后 , 应用不能使用 *ptr* 所指内存。参数 *event* 会返回一个事件对象 , 用来查询写命令的执行状态。当写命令完成后 , 应用可以重新使用 *ptr* 所指内存。

offset 是对缓冲对象读写的偏移量 , 以字节为单位。

cb 是要读写的字节数。

ptr 指向要读写的宿主机内存。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 NULL , 那么此命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL , 则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL , *event_wait_list* 所指事件列表必须是有效的 , 且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 会返回一个事件对象用来标识此读写命令 , 也可以用来查询或等待此命令的完成。*event* 也可以是 NULL , 但这样一来 , 应用就不能查询或等待此命令的完成了。

如果执行成功 , **clEnqueueReadBuffer** 和 **clEnqueueWriteBuffer** 会返回 CL_SUCCESS。否则 , 它会返回以下错误 :

- ✚ CL_INVALID_COMMAND_QUEUE , 如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT , 如果 *command_queue* 所关联的上下文与 *buffer* 所关联的上下文不一致 , 或者与 *event_wait_list* 中事件所关联的上下文不一致。
- ✚ CL_INVALID_MEM_OBJECT , 如果 *buffer* 无效。

- ✚ CL_INVALID_VALUE ,如果(*offset*,*cb*)所指定的读写区域越界 ,或 *ptr* 是 NULL。
- ✚ CL_INVALID_EVENT_WAIT_LIST , 如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list*>0 , 或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0 , 或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE , 如果为 *buffer* 所关联的数据仓库 (data store) 分配内存失败。
- ✚ CL_OUT_OF_HOST_MEMORY , 如果为宿主机上的 OpenCL 实现分配其所需资源失败。

注意：

当调用 **clEnqueueReadBuffer** 读取缓冲对象 (参数 *ptr* 的值为 *host_ptr + offset* , *host_ptr* 指向的内存区域是使用 CL_MEM_USE_HOST_PTR 创建缓冲对象时所指定的) 时 , 为避免未定义行为 , 必须满足以下要求：

- ✚ 当读命令开始执行时 , 所有使用此缓冲对象的命令必须都已经执行完毕。
- ✚ 此缓冲对象没有被映射。
- ✚ 在读命令执行完毕前 , 任何命令队列都不会使用此缓冲对象。

当调用 **clEnqueueWriteBuffer** 更新缓冲对象 (参数 *ptr* 的值为 *host_ptr + offset* , *host_ptr* 指向的内存区域是使用 CL_MEM_USE_HOST_PTR 创建缓冲对象时所指定的。) 中某部分最新内容 (latest bits) 时 , 为避免未定义行为 , 必须满足以下要求：

- ✚ 当写命令开始执行时 , 宿主机内存区域(*host_ptr + offset*,*cb*)包含最新内容。
- ✚ 此缓冲对象没有被映射。
- ✚ 在写命令执行完毕前 , 任何命令队列都不会使用此缓冲对象。

函数

```
cl_int clEnqueueCopyBuffer (cl_command_queue command_queue ,
                             cl_mem src_buffer ,
                             cl_mem dst_buffer ,
                             size_t src_offset ,
                             size_t dst_offset ,
                             size_t cb ,
                             cl_uint num_events_in_wait_list ,
                             const cl_event *event_wait_list ,
                             cl_event *event)
```

将一个命令入队 , 此命令会将缓冲对象 *src_buffer* 拷贝到缓冲对象 *dst_buffer* 中。

command_queue 指的是拷贝命令要加入的命令队列。它与 *src_buffer* 和 *dst_buffer* 所关联的 OpenCL 上下文必须一致。

src_offset 指从 *src_buffer* 的什么地方开始拷贝数据。

dst_offset 指从 *dst_buffer* 的什么地方开始写入数据。

cb 指要拷贝数据的字节数。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 NULL ,那么此命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL , 则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL , *event_wait_list* 所指事件列表必须是有效的, 且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 会返回一个事件对象用来标识此读写命令, 也可以用来查询或等待此命令的完成。*event* 也可以是 NULL, 但这样一来, 应用就不能查询或等待此命令的完成了。也可以用 **clEnqueueBarrier** 来代替。

如果执行成功, **clEnqueueCopyBuffer** 会返回 CL_SUCCESS。否则, 它会返回以下错误:

- ✚ CL_INVALID_COMMAND_QUEUE, 如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT, 如果 *command_queue* 所关联的上下文与 *src_buffer*、*dst_buffer* 所关联的上下文不一致, 或者与 *event_wait_list* 中事件所关联的上下文不一致。
- ✚ CL_INVALID_MEM_OBJECT, 如果 *src_buffer* 或 *dst_buffer* 无效。
- ✚ CL_INVALID_VALUE, 如果 *src_offset*、*dst_offset*、*cb*、*src_offset+cb* 或 *dst_offset+cb* 越界。
- ✚ CL_INVALID_EVENT_WAIT_LIST, 如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list*>0, 或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0, 或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_MEM_COPY_OVERLAP, 如果 *src_buffer* 和 *dst_buffer* 是同一个缓冲对象或者源和目的区域重叠。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE, 如果为 *src_buffer* 或 *dst_buffer* 所关联的数据仓库分配内存失败。
- ✚ CL_OUT_OF_HOST_MEMORY, 如果为宿主机上的 OpenCL 实现分配其所需资源失败。

5.2.3 保留和释放内存对象

函数


```
cl_int clRetainMemObject (cl_mem memobj)
```

会增加 *memobj* 的引用计数。如果执行成功, **clRetainMemObject** 会返回 CL_SUCCESS。如果 *memobj* 无效, 返回 CL_INVALID_MEM_OBJECT。**clCreateBuffer** 和 **clCreateImage{2D|3D}** 会隐式执行保留。

函数

```
cl_int clReleaseMemObject (cl_mem memobj)
```

会减小 *memobj* 的引用计数。当 *memobj* 的引用计数减小到 0, 并且命令队列中使用它的命令都执行完毕, 这个内存对象就会被删除。如果执行成功, **clReleaseMemObject** 会返回 CL_SUCCESS。如果 *memobj* 无效, 返回 CL_INVALID_MEM_OBJECT。

5.2.4 创建图像对象

使用下列函数创建图像对象 (一维或二维):

```
cl_mem clCreateImage2D (cl_context context,
                        cl_mem_flags flags,
                        const cl_image_format *image_format,
                        size_t image_width,
                        size_t image_height,
                        size_t image_row_pitch,
                        void *host_ptr,
                        cl_int *errcode_ret)
```

context 是一个有效的 OpenCL 上下文, 图像对象将在其上创建。

flags 是一个位域, 用来描述图像对象的分配和使用信息, 参见表 5.3。

image_format 指向一个结构体, 用来描述将要创建的图像对象的格式属性。详细内容参见节 5.2.4.1。

image_width 和 *image_height* 是图像的宽和高, 单位像素。必须大于等于 1。

image_row_pitch 是扫描线间距, 单位字节。如果 *host_ptr* 是 NULL, 则它必须是 0, 否则既可以是 0, 也可以 $\geq \text{image_width} \times \text{单个元素的字节数}$ 。如果 *host_ptr* 是 NULL 且 *image_row_pitch* 是 0, *image_row_pitch* 会被计算成 $\text{image_width} \times \text{单个元素的字节数}$ 。如果 *image_row_pitch* 不是 0, 那么它必须是单个图像元素字节数的整倍数。

host_ptr 指向可能已由应用分配了的图像数据。*host_ptr* 所指缓存的大小必须 $\geq \text{image_row_pitch} \times \text{image_height}$ 。每个元素的大小 (单位字节) 必须是 2 的幂。*host_ptr* 所指图像数据是按相邻扫描线线性存储的, 而每条扫描线则按图像元素线性存储。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL，就不会返回错误码。

如果成功创建了图像对象，**clCreateImage2D** 会返回此图像对象，并且将 *errcode_ret* 置为 CL_SUCCESS。否则，返回 NULL，并且将 *errcode_ret* 置为下列值中的一个：

- ✚ CL_INVALID_CONTEXT，如果 *context* 无效。
- ✚ CL_INVALID_VALUE，如果 *flags* 无效。
- ✚ CL_INVALID_IMAGE_FORMAT_DESCRIPTOR，如果 *image_format* 是 NULL 或无效。
- ✚ CL_INVALID_IMAGE_SIZE，如果 *image_width* 或 *image_height* 是 0，或者分别超过了 *context* 中所有设备的 CL_DEVICE_IMAGE2D_MAX_WIDTH 或 CL_DEVICE_IMAGE2D_MAX_HEIGHT，或者 *image_row_pitch* 没有遵循上面参数描述中的规则。
- ✚ CL_INVALID_HOST_PTR，如果 *host_ptr* 是 NULL 且 *flags* 中设置了 CL_MEM_USE_HOST_PTR 或 CL_MEM_COPY_HOST_PTR，或者 *host_ptr* 不是 NULL 但 *flags* 中设置了 CL_MEM_COPY_HOST_PTR 或 CL_MEM_USE_HOST_PTR。
- ✚ CL_IMAGE_FORMAT_NOT_SUPPORTED，如果不支持 *image_format*。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE，如果为图像对象分配内存时失败。
- ✚ CL_INVALID_OPERATION，如果 *context* 中的所有设备都不支持图像（如表 4.3 中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE）。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主主机上的 OpenCL 实现分配其所需资源失败。

使用下列函数可以创建一个三维图像对象：

```
cl_mem clCreateImage3D (cl_context context,
                        cl_mem_flags flags,
                        const cl_image_format *image_format,
                        size_t image_width,
                        size_t image_height,
                        size_t image_depth,
                        size_t image_row_pitch,
                        size_t image_slice_pitch,
                        void *host_ptr,
                        cl_int *errcode_ret)
```

context 是一个有效的 OpenCL 上下文，图像对象将在其上创建。

flags 是一个位域，用来描述图像对象的分配和使用信息，参见表 5.3。

image_format 指向一个结构体，用来描述将要创建的图像对象的格式属性。详细内容参见节 5.2.4.1。

image_width 和 *image_height* 是图像的宽和高，单位像素。必须大于等于 1。

image_depth 是图像的深度，单位像素，必须大于 1。

image_row_pitch 是扫描线间距，单位字节。如果 *host_ptr* 是 NULL，则它必须是 0，否则既可以是 0，也可以 $\geq \text{image_width} \times \text{单个元素的字节数}$ 。如果 *host_ptr* 是 NULL 且 *image_row_pitch* 是 0，*image_row_pitch* 会被计算成 $\text{image_width} \times \text{单个元素的字节数}$ 。如果 *image_row_pitch* 不是 0，那么它必须是单个图像元素字节数的整倍数。

image_slice_pitch 是三维图像中每个二维切片的字节数。如果 *host_ptr* 是 NULL，则它必须是 0，否则既可以是 0，也可以 $\geq \text{image_row_pitch} \times \text{image_height}$ 。如果 *host_ptr* 是 NULL 且 *image_slice_pitch* 是 0，*image_row_pitch* 会被计算成 $\text{image_row_pitch} \times \text{image_height}$ 。如果 *image_slice_pitch* 不是 0，那么它必须是 *image_row_pitch* 的整倍数。

host_ptr 指向可能已由应用分配了的图像数据。*host_ptr* 所指缓存的大小必须 $\geq \text{image_slice_pitch} \times \text{image_depth}$ 。每个元素的大小（单位字节）必须是 2 的幂。*host_ptr* 所指图像数据是按相邻二维切片线性存储的，每个二维切片按相邻扫描线线性存储，而每条扫描线则按图像元素线性存储。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL，就不会返回错误码。

如果成功创建了图像对象，**clCreateImage3D** 会返回此图像对象，并且将 *errcode_ret* 置为 CL_SUCCESS。否则，返回 NULL，并且将 *errcode_ret* 置为下列值中的一个：

- ✚ CL_INVALID_CONTEXT，如果 *context* 无效。
- ✚ CL_INVALID_VALUE，如果 *flags* 无效。
- ✚ CL_INVALID_IMAGE_FORMAT_DESCRIPTOR，如果 *image_format* 是 NULL 或无效。
- ✚ CL_INVALID_IMAGE_SIZE，如果 *image_width*、*image_height* 是 0，或 $\text{image_depth} \leq 1$ ，或者分别超过了 *context* 中所有设备的 CL_DEVICE_IMAGE3D_MAX_WIDTH、CL_DEVICE_IMAGE3D_MAX_HEIGHT 和 CL_DEVICE_IMAGE3D_MAX_DEPTH，或者 *image_row_pitch* 或 *image_slice_pitch* 没有遵循上面参数描述中的规则。
- ✚ CL_INVALID_HOST_PTR，如果 *host_ptr* 是 NULL 且 *flags* 中设置了 CL_MEM_USE_HOST_PTR 或 CL_MEM_COPY_HOST_PTR，或者 *host_ptr* 不是 NULL 但 *flags* 中设置了 CL_MEM_COPY_HOST_PTR 或 CL_MEM_USE_HOST_PTR。
- ✚ CL_IMAGE_FORMAT_NOT_SUPPORTED，如果不支持 *image_format*。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE，如果为图像对象分配内存时失败。

- ✚ CL_INVALID_OPERATION , 如果 *context* 中的所有设备都不支持图像 (如表 4.3 中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE)。
- ✚ CL_OUT_OF_HOST_MEMORY , 如果为宿主机上的 OpenCL 实现分配其所需资源失败。

5.2.4.1 图像格式描述符

结构体-图像格式描述符的定义如下：

```
typedef struct _cl_image_format {
    cl_channel_order    image_channel_order;
    cl_channel_type     image_channel_data_type;
} cl_image_format;
```

image_channel_order 指定通道数目和通道布局，即图像中的通道的内存布局。其所有合法值如表 5.4 所示。

表 5.4 cl_channel_order 支持列表

channel_order 可以指定的枚举值	此格式只有在 channel_data_type 是以下值时才可用：
CL_R 或 CL_A	全部
CL_INTENSITY	CL_UNORM_INT8、CL_UNORM_INT16、CL_SNORM_INT8、CL_SNORM_INT16、CL_HALF_FLOAT、CL_FLOAT。
CL_LUMINANCE	CL_UNORM_INT8、CL_UNORM_INT16、CL_SNORM_INT8、CL_SNORM_INT16、CL_HALF_FLOAT、CL_FLOAT。
CL_RG 或 CL_RA	全部
CL_RGB	CL_UNORM_SHORT_565、CL_UNORM_SHORT_555、CL_UNORM_INT_101010。
CL_RGBA	全部
CL_ARGB 或 CL_BGRA	CL_UNORM_INT8、CL_SNORM_INT8、CL_SIGNED_INT8、CL_UNSIGNED_INT8。

image_channel_data_type 描述通道数据类型的大小，其值如表 5.5 所示。由 *image_channel_data_type* 和 *image_channel_order* 所确定的单元素的比特数必须是 2 的幂。

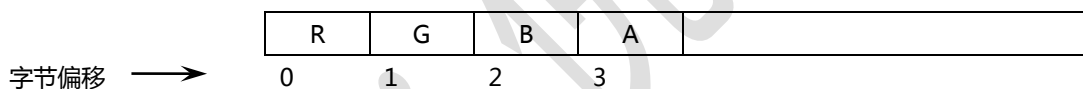
表 5.5 图像通道数据类型支持列表

图像通道数据类型	描述
CL_SNORM_INT8	各通道都是规范化带符号 8 位整型值
CL_SNORM_INT16	各通道都是规范化带符号 8 位整型值
CL_UNORM_INT8	各通道都是规范化无符号 8 位整型值
CL_UNORM_INT16	各通道都是规范化无符号 8 位整型值

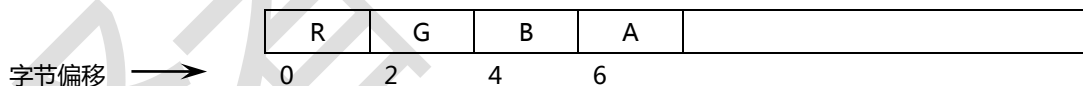
CL_UNORM_SHORT_565	规范化 5-6-5 三通道 RGB 图像。通道顺序必须是 CL_RGB。
CL_UNORM_SHORT_555	规范化 x-5-5-5 四通道 xRGB 图像。通道顺序必须是 CL_RGB。
CL_UNORM_INT_101010	规范化 x-10-10-10 四通道 xRGB 图像。通道顺序必须是 CL_RGB。
CL_SIGNED_INT8	各通道都是非规范化带符号 8 位整型值。
CL_SIGNED_INT16	各通道都是非规范化带符号 16 位整型值。
CL_SIGNED_INT32	各通道都是非规范化带符号 32 位整型值。
CL_UNSIGNED_INT8	各通道都是非规范化无符号 8 位整型值。
CL_UNSIGNED_INT16	各通道都是非规范化无符号 16 位整型值。
CL_UNSIGNED_INT32	各通道都是非规范化无符号 32 位整型值。
CL_HALF_FLOAT	各通道都是 16 位半浮点值。
CL_FLOAT	各通道都是单精度浮点值。

例如，各通道都是规范化无符号 8 位整型值的 RGBA 图像这样表示：

image_channel_order = CL_RGBA, *image_channel_data_type* = CL_UNORM_INT8。
这种图像格式的内存布局如下图所示：



类似的，如果 *image_channel_order* = CL_RGBA, *image_channel_data_type* = CL_UNORM_INT16，这种图像格式的内存布局如下图所示：



image_channel_data_type 的值：CL_UNORM_SHORT_565、CL_UNORM_SHORT_555 和 CL_UNORM_INT_101010 是几种特殊的压缩图像格式，每个元素的所有通道都打包到一个无符号短整型或无符号整型中。对于这些压缩图像格式，打包时一般是第一个通道占据最高位，后续通道相继占据次高位。对于 CL_UNORM_SHORT_565，R 占据比特 15:11，G 占据比特 10:5 而 B 则占据比特 4:0。对于 CL_UNORM_SHORT_555，比特 15 未定义，R 占据比特 14:10，G 占据比特 9:5 而 B 则占据比特 4:0。对于 CL_UNORM_INT_101010，比特 31:30 未定义，R 占据比特 29:20，G 占据比特 19:10 而 B 则占据比特 9:0。

OpenCL 实现必须维护 *image_channel_data_type* 中比特数的最小精度。如果 OpenCL 实现不支持 *image_channel_order* 和 *image_channel_data_type* 所指定的图像格式，*clCreateImage2D* 或 *clCreateImage3D* 会返回 NULL。

5.2.5 查询所支持的图像格式

函数

```
cl_int clGetSupportedImageFormats (cl_context context,
                                   cl_mem_flags flags,
                                   cl_mem_object_type image_type,
                                   cl_uint num_entries,
                                   cl_image_format *image_formats,
                                   cl_uint *num_image_formats)
```

可以用来获取 OpenCL 实现所支持的图像格式列表，同时可以指定图像对象的以下信息：

- ✚ 上下文
- ✚ 图像类型——二维或三维
- ✚ 图像对象的分配信息

context 是一个有效的 OpenCL 上下文，图像对象将在其上创建。

flags 是一个位域，用来指定所创图像对象的分配和使用信息，参见表 5.3。

image_type 指的是图像类型，必须是 CL_MEM_OBJECT_IMAGE2D 或 CL_MEM_OBJECT_IMAGE3D。

num_entries 指的是 *image_formats* 所指内存最多可以存储多少条图像格式。

image_formats 指向一块内存，用来存储返回的图像格式支持列表。每一条都是 OpenCL 实现所支持的 *cl_image_format* 结构。如果 *image_formats* 是 NULL 则被忽略。

num_image_formats 是对于特定 *context* 和 *flags* 所支持图像格式的实际数目。如果 *num_image_formats* 是 NULL，则被忽略。

如果执行成功，**clGetSupportedImageFormats** 返回 CL_SUCCESS；如果 *context* 无效则返回 CL_INVALID_CONTEXT；如果 *flags* 或 *image_type* 无效，或 *num_entries* 是 0 但 *image_formats* 不是 NULL，就返回 CL_INVALID_VALUE。

5.2.5.1 图像格式最小支持列表

如果 CL_DEVICE_IMAGE_SUPPORT (参见表 4.3) 是 CL_TRUE，那么实现给 CL_DEVICE_MAX_READ_IMAGE_ARGS、CL_DEVICE_MAX_WRITE_IMAGE_ARGS、CL_DEVICE_IMAGE2D_MAX_WIDTH、CL_DEVICE_IMAGE2D_MAX_HEIGHT、CL_DEVICE_IMAGE3D_MAX_WIDTH、CL_DEVICE_IMAGE3D_MAX_HEIGHT、

CL_DEVICE_IMAGE3D_MAX_DEPTH 和 CL_DEVICE_MAX_SAMPLERS 所指定的值必须大于等于表 4.3 所指定的最小值。OpenCL 实现必须支持下列图像格式。

对于只读的二维或三维图像，其图像格式的最小支持列表如表 5.6 所示：

表 5.6 对于只读图像，其图像格式的最小支持列表

image_num_channels	image_channel_order	image_channel_data_type
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
4	CL_BGRA	CL_UNORM_INT8

对于可读可写或只可写的二维图像，其图像格式的最小支持列表如表 5.7 所示：

表 5.7 对于可读可写或只可写的二维图像，其图像格式的最小支持列表

image_num_channels	image_channel_order	image_channel_data_type
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
4	CL_BGRA	CL_UNORM_INT8

5.2.6 读、写和拷贝图像对象

下列函数会将一个命令入队，此命令用来将 2D 或 3D 图像对象读入宿主机内存，或将

宿主机内存中的数据写入 2D 或 3D 图像对象中。

```
cl_int clEnqueueReadImage (cl_command_queue command_queue,
                             cl_mem image,
                             cl_bool blocking_read,
                             const size_t origin[3],
                             const size_t region[3],
                             size_t row_pitch,
                             size_t slice_pitch,
                             void *ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
cl_int clEnqueueWriteImage (cl_command_queue command_queue,
                             cl_mem image,
                             cl_bool blocking_write,
                             const size_t origin[3],
                             const size_t region[3],
                             size_t input_row_pitch,
                             size_t input_slice_pitch,
                             const void * ptr,
                             cl_uint num_events_in_wait_list,
                             const cl_event *event_wait_list,
                             cl_event *event)
```

command_queue 指容纳读写命令的命令队列。*command_queue* 和 *image* 必须创建自同一个 OpenCL 上下文。

image 指一个有效的 2D 或 3D 图像对象。

blocking_read 和 *blocking_write* 指明读写操作是阻塞的 (blocking) 还是非阻塞的 (non-blocking)。

如果 *blocking_read* 是 CL_TRUE , 也就是说读命令是阻塞的 , 只有当缓冲对象被读取并拷贝到 *ptr* 所指内存后 , **clEnqueueReadImage** 才会返回。

如果 *blocking_read* 是 CL_FALSE , 也就是说读命令是非阻塞的 , **clEnqueueReadImage** 会将非阻塞的读命令入队并返回。只有当读命令完成后才能使用 *ptr* 所指内容。参数 *event* 返回一个事件对象 , 此对象可以用来查询读命令的执行状态。读命令完成后 , 应用就可以使用 *ptr* 所指内容了。

如果 *blocking_write* 是 CL_TRUE , OpenCL 实现会拷贝 *ptr* 所指数据并将写操作入队。**clEnqueueWriteImage** 返回后 , 应用可以继续使用 *ptr* 所指内存。

如果 *blocking_write* 是 CL_FALSE , OpenCL 实现将使用 *ptr* 执行非阻塞写操作。由于非阻塞 , 实现可以立即返回。所以在其返回后 , 应用不能使用 *ptr* 所指内存。参数 *event* 会返回一个事件对象 , 用来查询写命令的执行状态。当写命令完成后 , 应用可以重新使用

*ptr*所指内存。

*origin*是对图像对象读写的偏移量(x, y, z)，以像素为单位。如果 *image* 是 2D 图像对象，那么 *origin*[2]所指定的 *z* 值必须是 0。

*region*是要读写的 2D 或 3D 矩形 (*width*, *height*, *depth*)，以像素为单位。如果 *image* 是 2D 图像，那么 *region*[2]所指定的 *depth* 必须是 1。

clEnqueueReadImage 中的 *row_pitch*和 **clEnqueueWriteImage** 中的 *input_row_pitch* 是每行的字节数。其值必须大于等于每个元素的字节数* *width*。如果是 0，则会根据单个元素的字节数* *width* 自动计算。

clEnqueueReadImage 中的 *slice_pitch*和 **clEnqueueWriteImage** 中的 *input_slice_pitch* 分别是要读写的 2D 切片或 3D 区域的字节数。如果 *image* 是 2D 图像，则必须是 0。另外，必须大于等于 *row_pitch* * *height*。如果是 0，则会根据 *row_pitch* * *height* 自动计算。

*ptr*指向要读写的宿主机内存。

*event_wait_list*和 *num_events_in_wait_list*指定在执行此命令前必须完成的事件。如果 *event_wait_list*是 NULL，那么此命令不用等待任何事件的完成。如果 *event_wait_list*是 NULL，则 *num_events_in_wait_list*必须是 0。如果 *event_wait_list*不是 NULL，*event_wait_list*所指事件列表必须是有效的，且 *num_events_in_wait_list*必须大于 0。*event_wait_list*中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

*event*会返回一个事件对象用来标识此读写命令，也可以用来查询或等待此命令的完成。*event*也可以是 NULL，但这样一来，应用就不能查询或等待此命令的完成了。

如果执行成功，**clEnqueueReadImage** 和 **clEnqueueWriteImage** 会返回 CL_SUCCESS。否则，它会返回以下错误：

- ✚ CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT，如果 *command_queue* 所关联的上下文与 *image* 所关联的上下文不一致，或者与 *event_wait_list* 中事件所关联的上下文不一致。
- ✚ CL_INVALID_MEM_OBJECT，如果 *image* 无效。
- ✚ CL_INVALID_VALUE，如果 *origin* 和 *region* 所指定的读写区域越界，或 *ptr* 是 NULL。
- ✚ CL_INVALID_VALUE，如果 *image* 是 2D 图像对象但 *origin*[2]不是 0 或者 *region*[2]不是 1 或者 *slice_pitch* 不是 0。

- ✚ CL_INVALID_EVENT_WAIT_LIST, 如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list* > 0, 或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0, 或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE, 如果为 *image* 所关联的数据仓库分配内存失败。
- ✚ CL_INVALID_OPERATION, 如果 *command_queue* 所关联的设备不支持图像 (即表 4.3 中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE)。
- ✚ CL_OUT_OF_HOST_MEMORY, 如果为宿主机上的 OpenCL 实现分配其所需资源失败。

注意：

当调用 **clEnqueueReadImage** 读取图像对象的某个区域时, 参数 *ptr* 的值为 $host_ptr + (origin[2] * \text{图像切片间距} + origin[1] * \text{图像行间距} + origin[0] * \text{单个像素的字节数})$, *host_ptr* 指向的内存区域是使用 CL_MEM_USE_HOST_PTR 创建图像对象时所指定的, 为避免未定义行为, 必须满足以下要求：

- ✚ 当读命令开始执行时, 所有使用此图像对象的命令必须都已经执行完毕。
- ✚ **clEnqueueReadImage** 的参数 *row_pitch* 和 *slice_pitch* 必须设置成图像的行间距和切片间距。
- ✚ 此图像对象没有被映射。
- ✚ 在读命令执行完毕前, 任何命令队列都不会使用此图像对象。

当调用 **clEnqueueWriteImage** 更新图像对象中某部分最新内容时, 参数 *ptr* 的值为 $host_ptr + (origin[2] * \text{图像切片间距} + origin[1] * \text{图像行间距} + origin[0] * \text{单个像素的字节数})$, *host_ptr* 指向的内存区域是使用 CL_MEM_USE_HOST_PTR 创建图像对象时所指定的, 为避免未定义行为, 必须满足以下要求：

- ✚ 当写命令开始执行时, 宿主机内存区域包含最新内容。
- ✚ **clEnqueueWriteImage** 中的参数 *input_row_pitch* 和 *input_slice_pitch* 必须设置成图像行间距和切片间距。
- ✚ 此缓冲对象没有被映射。
- ✚ 在写命令执行完毕前, 任何命令队列都不会使用此图像对象。

函数

```
cl_int clEnqueueCopyImage (cl_command_queue command_queue,
                           cl_mem src_image,
                           cl_mem dst_image,
                           const size_t src_origin[3],
                           const size_t dst_origin[3],
```

```
const size_t region[3],
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
```

会将一个用来拷贝图像对象的命令入队。*src_image*和*dst_image*是允许我们执行下列操作的 2D 或 3D 图像对象：

- ✚ 拷贝 2D 图像对象到 2D 图像对象。
- ✚ 拷贝 2D 图像对象到 3D 图像对象的 2D 切片。
- ✚ 拷贝 3D 图像对象的 2D 切片到 2D 图像对象。
- ✚ 拷贝 3D 图像对象到 3D 图像对象。

command_queue 指的是拷贝命令要加入的命令队列。它与 *src_image* 和 *dst_image* 所关联的 OpenCL 上下文必须一致。

src_origin 是对 *src_image* 中开始数据拷贝的起始位置(*x, y, z*)，以像素为单位。如果 *src_image* 是 2D 图像对象，那么 *src_origin*[2] 所指定的 *z* 值必须是 0。

dst_origin 是对 *src_image* 中开始数据拷贝的起始位置(*x, y, z*)，以像素为单位。如果 *dst_image* 是 2D 图像对象，那么 *dst_origin*[2] 所指定的 *z* 值必须是 0。

region 定义了要拷贝的 2D 或 3D 区域(*width, height, depth*)，以像素为单位。如果 *src_image* 或 *dst_image* 是 2D 图像对象，*region*[2] 给定的 *depth* 的值必须是 1。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 NULL，那么此命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL，则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL，*event_wait_list* 所指事件列表必须是有效的，且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 会返回一个事件对象用来标识此读写命令，也可以用来查询或等待此命令的完成。*event* 也可以是 NULL，但这样一来，应用就不能查询或等待此命令的完成了。也可以用 **clEnqueueBarrier** 来代替。

当前要求 **clEnqueueCopyImage** 的参数 *src_image* 和 *dst_image* 必须具有相同的图像格式（即创建 *src_image* 和 *dst_image* 时所指定的 *cl_image_format* 必须相匹配）。

如果执行成功，**clEnqueueCopyImage** 会返回 CL_SUCCESS。否则，它会返回以下错误：

- ✚ CL_INVALID_COMMAND_QUEUE , 如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT , 如果 *command_queue* 所关联的上下文与 *src_image*、*dst_image* 所关联的上下文不一致, 或者与 *event_wait_list* 中事件所关联的上下文不一致。
- ✚ CL_INVALID_MEM_OBJECT , 如果 *src_image* 或 *st_image* 无效。
- ✚ CL_IMAGE_FORMAT_MISMATCH , 如果 *src_image* 或 *st_image* 的图像格式不一致。
- ✚ CL_INVALID_VALUE , 如果 2D 或 3D 矩形区域(*src_origin*, *src_origin* + *region*)超出了 *src_image* , 或者(*dst_origin*, *dst_origin* + *region*)超出了 *dst_image*。
- ✚ CL_INVALID_VALUE , 如果 *src_image* 是 2D 图像对象, 但 *origin*[2]不是 0 或者 *region*[2]不是 1。
- ✚ CL_INVALID_VALUE , 如果 *dst_image* 是 2D 图像对象, 但 *dst_origin*[2]不是 0 或者 *region*[2]不是 1。
- ✚ CL_INVALID_EVENT_WAIT_LIST , 如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list*>0 , 或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0 , 或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE , 如果为 *src_buffer* 或 *dst_buffer* 所关联的数据仓库分配内存失败。
- ✚ CL_OUT_OF_HOST_MEMORY , 如果为宿主机上的 OpenCL 实现分配其所需资源失败。
- ✚ CL_INVALID_OPERATION , 如果 *command_queue* 所关联的设备不支持图像(即表 4.3 中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE)。
- ✚ CL_MEM_COPY_OVERLAP , 如果 *src_image* 和 *dst_image* 是同一个图像对象, 或者源和目的区域重叠。

5.2.7 图像对象和缓冲对象间的拷贝

函数

```
cl_int clEnqueueCopyImageToBuffer (cl_command_queue command_queue,
                                     cl_mem src_image,
                                     cl_mem dst_buffer,
                                     const size_t src_origin[3],
                                     const size_t region[3],
                                     size_t dst_offset,
                                     cl_uint num_events_in_wait_list,
                                     const cl_event *event_wait_list,
                                     cl_event *event)
```

将一个命令入队, 此命令可以将图像对象拷贝到缓冲对象中。

command_queue 必须有效。它与 *src_image* 和 *dst_buffer* 所关联的 OpenCL 上下

文必须一致。

src_image 是一个有效的图像对象。

dst_buffer 是一个有效的缓冲对象。

src_origin 是对 *src_image* 中开始数据拷贝的起始位置(*x*, *y*, *z*)，以像素为单位。如果 *src_image* 是 2D 图像对象，那么 *src_origin*[2] 所指定的 *z* 值必须是 0。

region 定义了要拷贝的 2D 或 3D 区域(*width*, *height*, *depth*)，以像素为单位。如果 *src_image* 是 2D 图像对象，*region*[2] 给定的 *depth* 的值必须是 1。

dst_offset 指从 *dst_buffer* 的什么地方开始写入数据。待拷贝区域的字节数 (即 *dst_cb*) 为 $width * height * depth * bytes/image\ element$ (如果 *src_image* 是 3D 图像对象) 或者 $width * height * bytes/image\ element$ (如果 *src_image* 是 2D 图像对象)。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 NULL，那么此命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL，则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL，*event_wait_list* 所指事件列表必须是有效的，且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 会返回一个事件对象用来标识此读写命令，也可以用来查询或等待此命令的完成。*event* 也可以是 NULL，但这样一来，应用就不能查询或等待此命令的完成了。也可以用 **clEnqueueBarrier** 来代替。

如果执行成功，**clEnqueueCopyImageToBuffer** 会返回 CL_SUCCESS。否则，它会返回以下错误：

- ✚ CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT，如果 *command_queue* 所关联的上下文与 *src_image*、*dst_buffer* 所关联的上下文不一致，或者与 *event_wait_list* 中事件所关联的上下文不一致。
- ✚ CL_INVALID_MEM_OBJECT，如果 *src_image* 或 *dst_buffer* 无效。
- ✚ CL_IMAGE_FORMAT_MISMATCH，如果 *src_image* 或 *dst_image* 的图像格式不一致。
- ✚ CL_INVALID_VALUE，如果 2D 或 3D 矩形区域(*src_origin*, *src_origin* + *region*) 超出了 *src_image*，或者(*dst_offset*, *dst_offset* + *dst_cb*)超出了 *dst_buffer*。
- ✚ CL_INVALID_VALUE，如果 *src_image* 是 2D 图像对象，但 *origin*[2] 不是 0 或者

region[2]不是 1。

- ✚ CL_INVALID_EVENT_WAIT_LIST, 如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list*>0, 或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0, 或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE, 如果为 *src_image* 或 *dst_buffer* 所关联的数据仓库分配内存失败。
- ✚ CL_INVALID_OPERATION, 如果 *command_queue* 所关联的设备不支持图像 (即表 4.3 中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE)。
- ✚ CL_OUT_OF_HOST_MEMORY, 如果为宿主机上的 OpenCL 实现分配其所需资源失败。

函数

```
cl_int clEnqueueCopyBufferToImage (cl_command_queue command_queue,
                                     cl_mem src_buffer,
                                     cl_mem dst_image,
                                     size_t src_offset,
                                     const size_t dst_origin[3],
                                     const size_t region[3],
                                     cl_uint num_events_in_wait_list,
                                     const cl_event *event_wait_list,
                                     cl_event *event)
```

将一个命令入队, 此命令可以将缓冲对象拷贝到图像对象中。

command_queue 必须有效。它与 *src_buffer* 和 *dst_image* 所关联的 OpenCL 上下文必须一致。

src_buffer 是一个有效的缓冲对象。

dst_image 是一个有效的图像对象。

src_offset 指从 *src_buffer* 的什么地方开始拷贝数据。

dst_origin 是 *dst_image* 中开始数据拷贝的起始位置(*x, y, z*), 以像素为单位。如果 *dst_image* 是 2D 图像对象, 那么 *dst_origin*[2]所指定的 *z* 值必须是 0。

region 定义了要拷贝的 2D 或 3D 区域(*width, height, depth*), 以像素为单位。如果 *dst_image* 是 2D 图像对象, *region*[2]给定的 *depth* 的值必须是 1。

src_buffer 待拷贝区域的字节数 (即 *src_cb*) 为 *width * height * depth * bytes/image element* (如果 *dst_image* 是 3D 图像对象) 或者 *width * height * bytes/image element* (如果 *dst_image* 是 2D 图像对象)。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 NULL ,那么此命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL , 则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL , *event_wait_list* 所指事件列表必须是有效的 , 且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 会返回一个事件对象用来标识此读写命令 ,也可以用来查询或等待此命令的完成。*event* 也可以是 NULL , 但这样一来 , 应用就不能查询或等待此命令的完成了。也可以用 **clEnqueueBarrier** 来代替。

如果执行成功, **clEnqueueCopyBufferToImage** 会返回 CL_SUCCESS。否则, 它会返回以下错误 :

- ✚ CL_INVALID_COMMAND_QUEUE , 如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT 如果 *command_queue* 所关联的上下文与 *src_buffer*、*dst_image* 所关联的上下文不一致 , 或者与 *event_wait_list* 中事件所关联的上下文不一致。
- ✚ CL_INVALID_MEM_OBJECT , 如果 *src_buffer* 或 *dst_image* 无效。
- ✚ CL_IMAGE_FORMAT_MISMATCH , 如果 *src_image* 或 *st_image* 的图像格式不一致。
- ✚ CL_INVALID_VALUE , 如果 2D 或 3D 矩形区域(*dst_origin*, *dst_origin* + *region*) 超出了 *dst_image* , 或者(*src_offset*, *src_offset* + *src_cb*)超出了 *src_buffer*。
- ✚ CL_INVALID_VALUE , 如果 *dst_image* 是 2D 图像对象 , 但 *dst_origin*[2]不是 0 或者 *region*[2]不是 1。
- ✚ CL_INVALID_EVENT_WAIT_LIST , 如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list*>0 , 或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0 , 或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE , 如果为 *src_image* 或 *dst_buffer* 所关联的数据仓库分配内存失败。
- ✚ CL_INVALID_OPERATION 如果 *command_queue* 所关联的设备不支持图像(即表 4.3 中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE)。
- ✚ CL_OUT_OF_HOST_MEMORY , 如果为宿主主机上的 OpenCL 实现分配其所需资源失败。

5.2.8 映射和解映射内存对象

函数

```

void * clEnqueueMapBuffer (cl_command_queue command_queue,
                           cl_mem buffer,
                           cl_bool blocking_map,
                           cl_map_flags map_flags,
                           size_t offset,
                           size_t cb,
                           cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list,
                           cl_event *event,
                           cl_int *errcode_ret)

```

会将一个命令入队，此命令可以将缓冲对象 *buffer* 的某个区域映射到宿主机地址空间中，并返回一个指向映射区域的指针。

command_queue 必须是一个有效的命令队列。

blocking_map 指明映射操作是阻塞的 (blocking) 还是非阻塞的 (non-blocking)。

如果 *blocking_map* 是 CL_TRUE，只有当 *buffer* 的指定区域被映射后，**clEnqueueMapBuffer** 才会返回。

如果 *blocking_map* 是 CL_FALSE，也就是说映射操作是非阻塞的，只有当映射命令完成后才能使用 **clEnqueueMapBuffer** 所返回的指向映射区域的指针。参数 *event* 返回一个事件对象，此对象可以用来查询映射命令的执行状态。映射命令完成后，应用就可以使用 **clEnqueueMapBuffer** 所返回的指针访问映射区域的内容了。

map_flags 是一个位域，如果设置了 CL_MAP_READ 表明此缓冲对象中由 (*offset*, *cb*) 指定的区域在映射后可读，如果设置了 CL_MAP_WRITE 表明此缓冲对象中由 (*offset*, *cb*) 指定的区域在映射后可写。

buffer 指一个有效的缓冲对象。*command_queue* 和 *buffer* 所关联的 OpenCL 上下文必须相同。

offset 和 *cb* 是对缓冲对象中要映射区域的偏移字节数和大小。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 NULL，那么此命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL，则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL，*event_wait_list* 所指事件列表必须是有效的，且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 会返回一个事件对象用来标识此读写命令，也可以用来查询或等待此命令的完成。*event* 也可以是 NULL，但这样一来，应用就不能查询或等待此命令的完成了。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL，就不会返回错误码。

如果执行成功，**clEnqueueMapBuffer** 会返回指向映射区域的指针，并将 *errcode_ret* 置为 CL_SUCCESS。否则，返回 NULL 指针，并将 *errcode_ret* 置为下列错误值中的一个：

- ✚ CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT，如果 *command_queue* 所关联的上下文与 *buffer* 所关联的上下文不一致，或者与 *event_wait_list* 中事件所关联的上下文不一致。
- ✚ CL_INVALID_MEM_OBJECT，如果 *buffer* 无效。
- ✚ CL_INVALID_VALUE，如果(*offset*, *cb*)所指定的映射区域越界，或 *map_flags* 无效。
- ✚ CL_INVALID_EVENT_WAIT_LIST，如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list* > 0，或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0，或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_MAP_FAILURE，如果映射失败。如果缓冲对象是使用 CL_MEM_USE_HOST_PTR 或 CL_MEM_ALLOC_HOST_PTR 创建的，就不会返回此错误。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE，如果为 *buffer* 所关联的数据仓库分配内存失败。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主主机上的 OpenCL 实现分配其所需资源失败。

所返回的指针所映射的区域起自 *offset*，大小至少是 *cb*（字节）。对此区域之外内存的访问，其结果未定义。

函数

```
void * clEnqueueMapImage (cl_command_queue command_queue,
                           cl_mem image,
                           cl_bool blocking_map,
                           cl_map_flags map_flags,
                           const size_t origin[3],
                           const size_t region[3],
                           size_t *image_row_pitch,
                           size_t *image_slice_pitch,
                           cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list,
                           cl_event *event,
                           cl_int *errcode_ret)
```

会将一个命令入队，此命令可以将图像对象 *image* 的某个区域映射到宿主机地址空间中，并返回一个指向映射区域的指针。

command_queue 必须是一个有效的命令队列。

image 指一个有效的图像对象。*command_queue* 和 *image* 所关联的 OpenCL 上下文必须相同。

blocking_map 指明映射操作是阻塞的 (blocking) 还是非阻塞的 (non-blocking)。

如果 *blocking_map* 是 CL_TRUE , 只有当 *image* 的指定区域被映射后 , **clEnqueueMapImage** 才会返回。

如果 *blocking_map* 是 CL_FALSE , 也就是说映射操作是非阻塞的 , 只有当映射命令完成后才能使用 **clEnqueueMapImage** 所返回的指向映射区域的指针。参数 *event* 返回一个事件对象 , 此对象可以用来查询映射命令的执行状态。映射命令完成后 , 应用就可以使用 **clEnqueueMapImage** 所返回的指针访问映射区域的内容了。

map_flags 是一个位域 , 如果设置了 CL_MAP_READ 表明此图像对象中由 (*origin, region*) 指定的区域在映射后可读 , 如果设置了 CL_MAP_WRITE 表明此图像对象中由 (*origin, region*) 指定的区域在映射后可写。

对于要映射的 2D 或 3D 矩形区域 , *origin* 定义了其偏移 (*x, y, z*) , 而 *region* 定义了 (*width, height, depth*) , 单位均为像素。如果 *image* 是 2D 图像对象 , *origin*[2] 指定的 *z* 值必须是 0 , 且 *region*[2] 所指定的 *depth* 必须是 1。

image_row_pitch 返回所映射区域的扫描线间距 , 单位字节 , 不能是 NULL。

image_slice_pitch 返回所映射区域每个 2D 切片的大小 , 单位字节。对于 2D 图像 , 如果此参数不是 NULL 就会返回 0 ; 而对于 3D 图像 , 此参数不能是 NULL。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 NULL , 那么此命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL , 则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL , *event_wait_list* 所指事件列表必须是有效的 , 且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 会返回一个事件对象用来标识此读写命令 , 也可以用来查询或等待此命令的完成。*event* 也可以是 NULL , 但这样一来 , 应用就不能查询或等待此命令的完成了。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL , 就不会返回错误码。

如果执行成功 , **clEnqueueMapImage** 会返回指向映射区域的指针 , 并将 *errcode_ret* 设置成 CL_SUCCESS。否则 , 返回 NULL 指针 , 并将 *errcode_ret* 设置成以下错误值中的一个 :

- ✚ CL_INVALID_COMMAND_QUEUE , 如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT , 如果 *command_queue* 所关联的上下文与 *image* 所关联的上下文不一致, 或者与 *event_wait_list* 中事件所关联的上下文不一致。
- ✚ CL_INVALID_MEM_OBJECT , 如果 *image* 无效。
- ✚ CL_INVALID_VALUE , 如果(*origin*, *origin + region*)所指定的映射区域越界, 或 *map_flags* 无效。
- ✚ CL_INVALID_VALUE , 如果 *image* 是 2D 图像, *origin*[2] 不是 0 或 *region*[2] 不是 1。
- ✚ CL_INVALID_VALUE , 如果 *image_row_pitch* 是 NULL。
- ✚ CL_INVALID_VALUE , 如果 *image* 是 3D 图像, 但 *image_slice_pitch* 是 NULL。
- ✚ CL_INVALID_EVENT_WAIT_LIST , 如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list* > 0, 或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0, 或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_MAP_FAILURE , 如果映射失败。如果图像对象是使用 CL_MEM_USE_HOST_PTR 或 CL_MEM_ALLOC_HOST_PTR 创建的, 就不会返回此错误。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE , 如果为 *image* 所关联的数据仓库分配内存失败。
- ✚ CL_INVALID_OPERATION , 如果 *command_queue* 所关联的设备不支持图像(即表 4.3 中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE)。
- ✚ CL_OUT_OF_HOST_MEMORY , 如果为宿主机上的 OpenCL 实现分配其所需资源失败。

所返回的指针所映射的 2D 或 3D 区域起自 *origin*, 对于 2D 图像大小至少是 $image_row_pitch * region[1] + region[0]$ (像素), 对于 3D 图像大小至少是 $image_slice_pitch * region[2] + image_row_pitch * region[1] + region[0]$ (像素)。对此区域之外内存的访问, 其结果未定义。

如果在创建缓冲对象或图像对象时, *mem_flags* 中设置了 CL_MEM_USE_HOST_PTR, 就会:

当 **clEnqueueMapBuffer** 或 **clEnqueueMapImage** 的命令完成时, 保证 **clCreateBuffer** 或 **clCreateImage{2D|3D}** 中的 *host_ptr* 含有所映射区域的最新内容。

clEnqueueMapBuffer 或 **clEnqueueMapImage** 返回的指针会从创建缓冲对象或图像对象时制定的 *host_ptr* 继承。

函数

<code>cl_int clEnqueueUnmapMemObject (cl_command_queue <i>command_queue</i>,</code>
--

```
cl_mem memobj,
void *mapped_ptr,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
```

入队一个命令，对之前所映射的内存对象区域进行解映射。使用 **clEnqueueMapBuffer** 或 **clEnqueueMapImage** 所返回的指针对宿主机的所有读写操作必须在之前全部完成。

command_queue 必须是一个有效的命令队列。

mem_obj 必须是一个有效的内存对象。*command_queue* 和 *mem_obj* 所关联的 OpenCL 上下文必须相同。

mapped_ptr 是之前调用 **clEnqueueMapBuffer** 或 **clEnqueueMapImage** 返回的宿主机地址。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 NULL，那么此命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL，则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL，*event_wait_list* 所指事件列表必须是有效的，且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 会返回一个事件对象用来标识此读写命令，也可以用来查询或等待此命令的完成。*event* 也可以是 NULL，但这样一来，应用就不能查询或等待此命令的完成了。也可以用 **clEnqueueBarrier** 来代替。

如果执行成功，**clEnqueueUnmapMemObject** 会返回 CL_SUCCESS。否则，返回下列错误之一：

- ✚ CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 无效。
- ✚ CL_INVALID_MEM_OBJECT，如果 *memobj* 无效。
- ✚ CL_INVALID_VALUE，如果 **clEnqueueMapBuffer** 或 **clEnqueueMapImage** 为 *memobj* 所返回的 *mapped_ptr* 无效。
- ✚ CL_INVALID_EVENT_WAIT_LIST，如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list* > 0，或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0，或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主机上的 OpenCL 实现分配其所需资源失败。
- ✚ CL_INVALID_CONTEXT，如果 *command_queue* 所关联的上下文与 *memobj*

所关联的上下文不一致，或者与 *event_wait_list* 中事件所关联的上下文不一致。

clEnqueueMapBuffer 和 **clEnqueueMapImage** 会增加内存对象的映射计数。内存对象的映射计数的初始值是 0。对同一个内存对象多次调用 **clEnqueueMapBuffer** 或 **clEnqueueMapImage** 会将映射计数增大相应的调用次数。

clEnqueueUnmapMemObject 会减小内存对象的映射计数。

clEnqueueMapBuffer 和 **clEnqueueMapImage** 充当内存对象所映射区域的同步点。

5.2.8.1 用来访问内存对象所映射区域的OpenCL命令的行为

如果 **clEnqueueMapBuffer** 或 **clEnqueueMapImage** 的参数 *mem_flags* 设置了 **CL_MAP_WRITE**，那么在解映射前，所映射区域的内容是未定义的。设备上所运行的内核对其进行的任何读写操作，其结果均是未定义的。

多个命令队列可以将某个内存对象的某个区域或重叠区域映射为可读（如 *mem_flags*=**CL_MAP_READ**）。对于映射为可读的内存对象某区域，设备上执行的内核可以读取其内容。设备上执行的内核对映射区域的写操作的行为是未定义的。将重叠区域映射（或解映射）为可写是未定义的。

对于一个 OpenCL 函数调用，如果它会将一个读写映射区域的命令入队，其结果是未定义的。

5.2.9 内存对象查询

使用下列函数可以获取所有内存对象共用的信息：

```
cl_int clGetMemObjectInfo (cl_mem memobj,
                           cl_mem_info param_name,
                           size_t param_value_size,
                           void *param_value,
                           size_t *param_value_size_ret)
```

memobj 指定要查询的内存对象。

param_name 指定要查询的信息。所支持的类型和 *param_value* 所返回的信息如表 5.8 所示。

param_value 指向存储查询结果的内存。如果是 NULL，则忽略。

param_value_size 用来指定 *param_value* 所指内存的字节数。必须 ≥ 表 5.8 所列返回类型的大小。

param_value_size_ret 返回 *param_value* 所存储的查询数据的实际字节数。如果是 NULL，则忽略。

如果执行成功，**clGetMemObjectInfo** 返回 CL_SUCCESS。如果 *param_name* 无效，或者 *param_value_size* 的值 < 表 5.8 所列返回类型的大小，且 *param_value* 不是 NULL，则返回 CL_INVALID_VALUE；如果 *memobj* 无效，则返回 CL_INVALID_MEM_OBJECT。

表 5.8 clGetMemObjectInfo 的 param_names 支持列表

cl_mem_info	返回类型	param_value 所返回的信息
CL_MEM_TYPE	cl_mem_object_type	如果 <i>memobj</i> 是用 clCreateBuffer 创建，则返回 CL_MEM_OBJECT_BUFFER；如果是用 clCreateImage2D 创建则返回 CL_MEM_OBJECT_IMAGE2D；如果是用 clCreateImage3D 创建则返回 CL_MEM_OBJECT_IMAGE3D。
CL_MEM_FLAGS	cl_mem_flags	返回用 clCreateBuffer 或 clCreateImage{2D 3D} 创建 <i>memobj</i> 时所指定的参数 <i>flags</i> 。
CL_MEM_SIZE	size_t	返回 <i>memobj</i> 所关联的数据仓库的实际大小，单位字节。
CL_MEM_HOST_PTR	void *	返回创建 <i>memobj</i> 时所指定的参数 <i>host_ptr</i> 。
CL_MEM_MAP_COUNT ⁹	cl_unit	映射计数。
CL_MEM_REFERENCE_COUNT ¹⁰	cl_unit	返回 <i>memobj</i> 的引用计数。
CL_MEM_CONTEXT	cl_context	返回创建 <i>memobj</i> 时所指定的上下文。

使用下列函数可以获取用 **clCreateImage{2D|3D}** 所创建的图像对象的特定信息：

```
cl_int clGetImageInfo (cl_mem image,
                        cl_image_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret)
```

image 指定要查询的图像对象。

param_name 指定要查询的信息。所支持的类型和 *param_value* 所返回的信息如表

⁹ 所返回的映射计数会立刻过时，不适合应用中通常的用途。提供这个特性是为了调试。

¹⁰ 所返回的引用计数会立刻过时，不适合应用中通常的用途。提供这个特性是为了定位内存泄露。

5.9 所示。

param_value 指向存储查询结果的内存。如果是 NULL，则忽略。

param_value_size 用来指定 *param_value* 所指内存的字节数。必须 \geq 表 5.9 所列返回类型的大小。

param_value_size_ret 返回 *param_value* 所存储的查询数据的实际字节数。如果是 NULL，则忽略。

如果执行成功，**clGetImageInfo** 返回 CL_SUCCESS；如果 *param_name* 无效，或者 *param_value_size* 的值 $<$ 表 5.9 所列返回类型的大小，且 *param_value* 不是 NULL，则返回 CL_INVALID_VALUE；如果 *image* 无效，则返回 CL_INVALID_MEM_OBJECT。

表 5.9 clGetImageInfo 的 param_names 支持列表

cl_image_info	返回类型	param_value 所返回的信息
CL_IMAGE_FORMAT	cl_image_format	返回用 clCreateImage{2D 3D} 创建 <i>image</i> 时所指定的图像格式描述符。
CL_IMAGE_ELEMENT_SIZE	size_t	返回 <i>image</i> 中每个元素的大小。每个元素由 n 个通道组成，其中 n 由 cl_image_format 指定。
CL_IMAGE_ROW_PITCH	size_t	返回 <i>image</i> 每行元素的大小。
CL_IMAGE_SLICE_PITCH	size_t	如果 <i>image</i> 是 3D 图像对象，则返回其 2D 切片的大小，如果是 2D 图像对象，则返回 0。
CL_IMAGE_WIDTH	size_t	返回图像的宽，单位像素。
CL_IMAGE_HEIGHT	size_t	返回图像的高，单位像素。
CL_IMAGE_DEPTH	size_t	返回图像的深度，单位像素；对于 2D 图像对象，返回 0。

5.3 采样器对象

采样器对象所描述的是在将图像读入内核时怎样对其采样。内建函数在将图像读入内核时将采样器作为一个参数。此参数可以是使用 OpenCL 函数创建的采样器对象，也可以是在内核中声明的采样器。本节将介绍怎样使用 OpenCL 函数创建采样器对象。

函数

```
cl_sampler clCreateSampler (cl_context context,
                             cl_bool normalized_coords,
                             cl_addressing_mode addressing_mode,
                             cl_filter_mode filter_mode,
                             cl_int *errcode_ret)
```

会创建一个采样器对象。对于采样器怎样工作请参见节 6.11.8.1。

context 必须是一个有效的 OpenCL 上下文。

normalized_coords 决定所指定的图像坐标是规范化的 (CL_TRUE) 还是非规范化的 (CL_FALSE)。

addressing_mode 决定了读取图像时怎样处理越界的图像坐标。可以设置成 CL_ADDRESS_REPEAT、CL_ADDRESS_CLAMP_TO_EDGE、CL_ADDRESS_CLAMP 和 CL_ADDRESS_NONE。

filtering_mode 决定了读取图像时所采用的滤波模式。可以是 CL_FILTER_NEAREST 或 CL_FILTER_LINEAR。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL，就不会返回错误码。

如果成功创建了采样器对象，**clCreateSampler** 会返回此对象，并将 *errcode_ret* 置为 CL_SUCCESS。否则，返回 NULL，并将 *errcode_ret* 置为下列错误值之一：

- ✚ CL_INVALID_CONTEXT，如果 *context* 无效。
- ✚ CL_INVALID_VALUE，如果 *addressing_mode*、*filter_mode* 或 *normalized_coords*，或者这些参数的组合无效。
- ✚ CL_INVALID_OPERATION，如果 *context* 所关联的所有设备都不支持图像（如表 4.3 中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE）。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主机上的 OpenCL 实现分配其所需资源失败。

函数

```
cl_int clRetainSampler (cl_sampler sampler)
```

会增大 *sampler* 的引用计数。**clCreateSampler** 会执行隐式的保留。如果执行成功，**clRetainSampler** 会返回 CL_SUCCESS。如果 *sampler* 无效，则返回 CL_INVALID_SAMPLER。

函数


```
cl_int clReleaseSampler (cl_sampler sampler)
```

会减小 *sampler* 的引用计数。如果引用计数减小到 0，而且所有要使用 *sampler* 的命令都执行完毕，此采样器对象会被删除。如果执行成功，**clReleaseSampler** 返回 CL_SUCCESS。如果 *sampler* 无效，则返回 CL_INVALID_SAMPLER。

函数

```
cl_int clGetSamplerInfo (cl_sampler sampler,
                          cl_sampler_info param_name,
                          size_t param_value_size,
                          void *param_value,
                          size_t *param_value_size_ret)
```

返回 *sampler* 的相关信息。

sampler 指定要查询的采样器。

param_name 指定要查询的信息。**clGetSamplerInfo** 所支持的 *param_name* 类型以及 *param_value* 中所返回的信息如表 5.10 所示。

param_value 指向存储查询结果的内存。如果是 NULL，则忽略。

param_value_size 表明 *param_value* 所指内存的大小。必须 ≥ 表 5.10 所列返回类型的大小。

param_value_size_ret 返回 *param_value* 中所存数据的实际大小。如果是 NULL，则忽略。

表 5.10 clGetSamplerInfo 参数查询

cl_sampler_info	返回类型	param_value 所返回的信息
CL_SAMPLER_REFERENCE_COUNT ¹¹	cl_uint	返回 <i>sampler</i> 的引用计数。
CL_SAMPLER_CONTEXT	cl_context	返回创建 <i>sampler</i> 时所指定的上下文。
CL_SAMPLER_ADDRESSING_MODE	cl_addressing_mode	返回 clCreateSampler 的参数 <i>addressing_mode</i> 。
CL_SAMPLER_FILTER_MODE	cl_filter_mode	返回 clCreateSampler 的参数 <i>filter_mode</i> 。
CL_SAMPLER_NORMALIZED_COORDS	cl_bool	返回 clCreateSampler 的参数 <i>normalized_coords</i> 。

¹¹ 所返回的引用计数会立刻过时，不适合应用中通常的用途。提供这个特性是为了定位内存泄露。

如果执行成功, **clGetSamplerInfo** 返回 CL_SUCCESS。如果 *param_name* 无效, 或者 *param_value_size* 的值 < 表 5.10 所列返回类型的大小但 *param_value* 不是 NULL, 则返回 CL_INVALID_VALUE; 如果 *sampler* 无效, 则返回 CL_INVALID_SAMPLER。

5.4 程序对象

一个 OpenCL 程序由一套内核组成, 即程序代码中带限定符 `_kernel` 的函数。OpenCL 程序也可能包含一些辅助函数和常量数据, 这些都由 `_kernel` 函数使用。OpenCL 编译器可以为相应的目标设备在线或离线生成可执行程序。

一个程序对象封装了以下信息:

- ✚ 所关联的上下文。
- ✚ 程序源码或二进制。
- ✚ 最近成功构建的可执行程序, 设备列表 (用来运行此程序), 所使用的构建选项和一个构建日志。

5.4.1 创建程序对象

函数

```
cl_program clCreateProgramWithSource (cl_context context,
                                     cl_uint count,
                                     const char **strings,
                                     const size_t *lengths,
                                     cl_int *errcode_ret)
```

为一个上下文创建程序对象, 并载入 *strings* 中文本字符串所指定的程序源码。此程序对象所关联的设备就是 *context* 所关联的设备。

context 必须是有效的 OpenCL 上下文。

strings 是源码, 由 *count* 个字符串指针组成的数组, 其中字符串可以选择用 null 终止符表示结束。

参数 *lengths* 是一个数组, 存放的是每个字符串中字符的数目 (即字符串长度)。如果 *lengths* 中某个元素是 0, 则相应的字符串以 null 结尾。如果 *lengths* 是 NULL, 则参数 *strings* 中所有字符串都是以 null 结尾。所有大于 0 的长度值都不包括 null 终止符。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL, 就不会返回错误码。

如果成功创建了程序对象, **clCreateProgramWithSource** 会将此对象返回, 并将

errcode_ret 置为 CL_SUCCESS。否则，返回 NULL，并将 *errcode_ret* 置为下列错误值之一：

- ✚ CL_INVALID_CONTEXT，如果 *context* 无效。
- ✚ CL_INVALID_VALUE，如果 *count* 是 0，或者 *strings* 或其中任一项是 NULL。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主机上的 OpenCL 实现分配其所需资源失败。

函数

```
cl_program clCreateProgramWithBinary (cl_context context,
                                       cl_uint num_devices,
                                       const cl_device_id *device_list,
                                       const size_t *lengths,
                                       const unsigned char **binaries,
                                       cl_int *binary_status,
                                       cl_int *errcode_ret)
```

为一个上下文创建程序对象，并载入 *binaries* 中的二进制。

context 必须是有效的 OpenCL 上下文。

device_list 指向 *context* 中的设备列表，不能是 NULL。所装载的二进制就是为这些设备准备的。

num_devices 是 *device_list* 中设备的数目。

此程序对象所关联的设备就是 *device_list* 中所列设备。而 *device_list* 中所列设备必须是 *context* 所关联的设备。

lengths 是一个数组，其中元素是为 *device_list* 中的设备所装载的程序二进制的大小。

binaries 是一个指针数组，其中元素指向为 *device_list* 中的设备所装载的程序二进制。对于 *device_list* 中的每个设备 *device_list[i]*，*binaries[i]* 指向此设备的二进制，而其大小则为 *lengths[i]*。*lengths[i]* 不能是 0，且 *binaries[i]* 不能是 NULL。

binaries 所指定的程序二进制包含描述可执行程序的比特，这些程序将会在 *context* 所关联的设备上运行。程序二进制可能包含下列之一，也可能二者兼有：

- ✚ 特定设备 (device-specific) 的执行体 (executable)，和/或
- ✚ 特定实现 (implementation-specific) 的中间表示 (intermediate representation，简称 IR)，将被转换成特定设备的执行体。

binary_status 表示 *device_list* 中每个设备的程序二进制是否装载成功。它是一个数组，有 *num_devices* 个元素；如果设备 *device_list[i]* 的二进制装载成功，则 *binary_status[i]* 就是 CL_SUCCESS。如果 *lengths[i]* 是 0 或 *binaries[i]* 是 NULL，则返回 CL_INVALID_VALUE；如果指定设备的程序二进制无效，则返回 CL_INVALID_BINARY。如果 *binary_status* 是 NULL，则忽略。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL，就不会返回错误码。

如果成功创建了程序对象，**clCreateProgramWithBinary** 会将此对象返回，并将 *errcode_ret* 置为 CL_SUCCESS。否则，返回 NULL，并将 *errcode_ret* 置为下列错误值之一：

- ✚ CL_INVALID_CONTEXT，如果 *context* 无效。
- ✚ CL_INVALID_VALUE，如果 *device_list* 是 NULL 或 *num_devices* 是 0。
- ✚ CL_INVALID_DEVICE，如果 *device_list* 中的设备不是 *context* 所关联的设备。
- ✚ CL_INVALID_VALUE，如果 *lengths* 或 *binaries* 是 NULL，或者 *lengths[i]* 是 0 或 *binaries[i]* 是 NULL。
- ✚ CL_INVALID_BINARY，如果任何一个程序二进制无效。*binary_status* 会返回相应的状态。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主机上的 OpenCL 实现分配其所需资源失败。

OpenCL 允许应用使用程序源码或二进制创建程序对象，并构建成相应的程序执行体。这样就可以让应用自己决定是使用之前离线构建的二进制，还是装载并编译程序源码从而使用在线编译链接的执行体。它允许应用为程序的第一个实例在线装载并构建程序执行体，这是非常有用的。这样，应用就可以查询这些执行体并将其缓存起来。后续所需实例就不必重新构建了。应用可以读取并加载缓存起来的执行体，这样可以在很大程度上减少应用在初始化上所用的时间。

函数

```
cl_int clRetainProgram (cl_program program)
```

会增大 *program* 的引用计数。**clCreateProgram** 会执行隐式的保留。如果执行成，会返回 CL_SUCCESS。而如果 *program* 无效，则返回 CL_INVALID_PROGRAM。

函数

```
cl_int clReleaseProgram (cl_program program)
```

会减小 *program* 的引用计数。当所有与 *program* 关联的内核对象全都被删除且

program 的引用计数减小到 0 时, *program* 就会被删除。如果执行成功, 会返回 CL_SUCCESS。而如果 *program* 无效, 则返回 CL_INVALID_PROGRAM。

5.4.2 构建程序执行体

函数

```
cl_int clBuildProgram (cl_program program,
                      cl_uint num_devices,
                      const cl_device_id *device_list,
                      const char *options,
                      void (*pfn_notify)(cl_program program,
                                          void *user_data),
                      void *user_data)
```

为 *program* 关联的上下文中的所有设备或某个设备构建(编译和链接)程序执行体(由程序源码)或二进制。OpenCL 运行使用源码或二进制构建执行体。无论 *program* 是由 **clCreateProgramWithSource** 还是由 **clCreateProgramWithBinary** 创建的, 都必须调用 **clBuildProgram** 为其所关联的设备构建程序执行体。

program 是一个程序对象。

device_list 指向 *program* 所关联的设备列表。如果是 NULL, 只要加载了相应的源码或二进制, 且是 *program* 所关联的设备, 就会为这样的设备构建程序执行体。如果不是 NULL, 只要加载了相应的源码或二进制, 且在这个列表中, 就会为这样的设备构建程序执行体。

num_devices 是 *device_list* 中所列设备的数目。

options 指向一个字符串, 此字符串用来描述构建程序时所用的构建选项。所支持的选项参见节 5.4.3。

pfn_notify 是函数指针, 指向要通知的例程。此例程是一个回调函数, 应用可以自己注册, 当构建完程序执行体后(无论成功与否)会调用它。如果不是 NULL, **clBuildProgram** 不必等待构建完成, 可以立即返回。如果是 NULL, 在构建完成之前, **clBuildProgram** 不会返回。此回调函数可能被 OpenCL 实现异步调用。应用负责保证回调函数的线程安全。

user_data 会作为 *pfn_notify* 的参数传递(当 *pfn_notify* 被调用时)。可以是 NULL。

如果执行成功, **clBuildProgram** 会返回 CL_SUCCESS。否则, 返回下列错误之一:

- ✚ CL_INVALID_PROGRAM, 如果 *program* 无效。
- ✚ CL_INVALID_VALUE, 如果 *device_list* 是 NULL 且 *num_devices* 大于 0, 或者

device_list 不是 NULL 但 *num_devices* 是 0。

- ✚ CL_INVALID_VALUE , 如果 *pfn_notify* 是 NULL 但 *user_data* 不是 NULL。
- ✚ CL_INVALID_DEVICE , 如果 *device_list* 中所列 OpenCL 设备不是 *program* 所关联的设备。
- ✚ CL_INVALID_BINARY 如果 *program* 是由 **clCreateWithProgramBinary** 创建, 且对于 *device_list* 中所列设备没有加载相应的有效程序二进制。
- ✚ CL_INVALID_BUILD_OPTIONS , 如果 *options* 所指定的构建选项无效。
- ✚ CL_INVALID_OPERATION , 如果之前针对 *program* 的调用 **clBuildProgram** 为 *device_list* 中所列任意设备构建程序执行体没有完成。
- ✚ CL_COMPILER_NOT_AVAILABLE , 如果 *program* 由 **clCreateProgramWithSource** 创建, 且没有有效的编译器, 如: 表 4.3 中的 CL_DEVICE_COMPILER_AVAILABLE 被置为 CL_FALSE。
- ✚ CL_BUILD_PROGRAM_FAILURE , 如果构建程序执行体失败。如果构建完成后 **clBuildProgram** 仍没有返回, 就会返回这个错误。
- ✚ CL_INVALID_OPERATION , 如果有内核对象附到了 *program* 上。
- ✚ CL_OUT_OF_HOST_MEMORY , 如果为宿主主机上的 OpenCL 实现分配其所需资源失败。

5.4.3 构建选项

构建选项可以分为预处理选项、固有数学选项、控制优化的选项以及其他选项。本规范定义了在线或离线构建程序执行体时, OpenCL 编译器必须支持的一套标准选项。也可以加一些特定供应商或特定平台的选项对其进行扩展。

5.4.3.1 预处理选项

这些选项控制 OpenCL 预处理器, 实际编译前作用于程序源码上。

-D *name*

将 *name* 预定义为一个宏, 其值为 1。

-D *name=definition*

将 *definition* 的内容符号化, 在翻译阶段遇到 '#define' 指令就会处理。实际上, 如果遇到内嵌的 newline 字符会发生截断。

-D 选项会按 **clBuildProgram** 参数中给定的顺序进行处理。

-I *dir*

将 *dir* 加入搜索头文件的目录列表。

5.4.3.2 固有数学选项

这些选项控制跟浮点算术有关的编译器行为。这些选项会影响速度和正确性之间的权衡。

`-cl-single-precision-constant`

将双精度浮点常数视为单精度浮点常数。

`-cl-denorms-are-zero`

此选项控制怎样处理单精度和双精度去规格化数 (denormalized number)。如果将其作为构建选项, 单精度去规格化数会被刷新成 0, 而如果选项扩展支持双精度, 则双精度去规格化数也会被刷新成 0。此项仅作为性能暗示, 如果设备支持单精度 (或双精度) 去规格化数, OpenCL 编译器可以自己决定是否将其刷新成 0。

如果设备不支持单精度去规格化数, 如 `CL_DEVICE_SINGLE_FP_CONFIG` 中没有设置比特 `CL_FP_DENORM`, 在处理单精度去规格化数时会忽略此选项。

如果设备不支持双精度去规格化数, 或者设备支持但是 `CL_DEVICE_DOUBLE_FP_CONFIG` 中没有设置 `CL_FP_DENORM`, 那么处理双精度去规格化数时会忽略此选项。

只有在处理程序中的标量和矢量单精度浮点变量及其上的运算时才会应用此标志, 而在读写图像对象时没任何作用。

5.4.3.3 优化选项

这些选项控制各种优化。打开优化标志会使编译器尝试改进性能和/或代码大小, 这些都是以增加编译时间为代价的, 也可能改进调试程序的能力。

`-cl-opt-disable`

此选项会关闭所有优化, 默认情况下会使能优化。

`-cl-strict-aliasing`

此选项运行编译器对别名规则做最严格的假设。

下列选项控制与浮点算术相关的编译器行为。这些选项会在性能和正确性之间取得平衡, 必须明确使能。由于这些选项可能导致不正确的输出, 而这些输出依赖于对 IEEE 754 有关

数学函数的规则/规范的具体实现，所以默认情况下不会打开这些选项。

`-cl-mad-enable`

允许用 `mad` 代替 $a * b + c$ 。`mad` 会使用弱化的精确度去计算 $a * b + c$ 。例如，一些 OpenCL 设备在实现 `mad` 时，在加 `c` 前会将 $a * b$ 的结果进行截断。

`-cl-no-signed-zeros`

允许在做浮点算术时忽略 0 的正负。IEEE 754 的算术定义了对于 $+0.0$ 和 -0.0 的不同行为，这样就会禁止对表达式 $x+0.0$ 或 $0.0*x$ （即使只带有 `-cl-finite-math`）进行简化。此选项暗示 0 的符号并不重要。

`-cl-unsafe-math-optimizations`

允许对浮点算术的优化：(a) 假定参数和结果都是有效的，(b) 可能违反了 IEEE 754 标准，(c) 可能违反了 OpenCL 数值一致性（numerical compliance）要求（对于单精度浮点数请参见节 7.4，对于双精度浮点数请参见节 9.3.9，对于边界条件的行为请参见节 7.5）。此选项包含 `-cl-no-signed-zeros` 和 `-cl-mad-enable` options。

`-cl-finite-math-only`

允许在对浮点算术优化时假定参数和结果都不会是 NaN 或 $\pm\infty$ 。此选项可能会违反 OpenCL 数值一致性要求（对于单精度浮点数请参见节 7.4，对于双精度浮点数请参见节 9.3.9，对于边界条件的行为请参见节 7.5）。

`-cl-fast-relaxed-math`

设置优化选项 `-cl-finite-math-only` 和 `-cl-unsafe-math-optimizations`。对浮点算术的优化可能违反 IEEE 754 标准和 OpenCL 数值一致性要求（对于单精度浮点数请参见节 7.4，对于双精度浮点数请参见节 9.3.9，对于边界条件的行为请参见节 7.5）。此选项会在 OpenCL 程序中定义预处理器宏 `_FAST_RELAXED_MATH_`。

5.4.3.4 请求或抑制告警的选项

告警是一些诊断信息，用来报告一些构造虽然不属于错误但是有风险，或者可能是错误。下列是一些独立于语言的选项，不会使能特定的告警，但可以控制 OpenCL 编译器所生成的诊断信息的种类。

`-w`

禁止所有告警消息。

-Werror

将所有告警都当错误报告。

5.4.4 卸载OpenCL编译器

函数

```
cl_int clUnloadCompiler (void)
```

允许实现释放 OpenCL 编译器所分配的资源。这是由应用提供的，仅作为一个暗示，并不保证将来再也不使用编译器，也不保证实现会真正卸载编译器。如有必要，在调用 **clUnloadCompiler** 后也可以调用 **clBuildProgram** 重写加载编译器，来构建相应的程序执行体。目前此调用永远都返回 CL_SUCCESS。

5.4.5 程序对象查询

函数

```
cl_int clGetProgramInfo (cl_program program,
                        cl_program_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret)
```

返回此程序对象的信息。

program 指定要查询的程序对象。

param_name 指定要查询的信息。所支持的类型和返回的信息如表 5.11 所示。

param_value 指向存储查询结果的内存。如果是 NULL，则忽略。

param_value_size 表明 *param_value* 所指内存的大小。其值必须 \geq 表 5.11 所列返回类型的大小。

param_value_size_ret 会返回拷贝到 *param_value* 中的数据的大小。如果是 NULL，则忽略。

表 5.11 clGetProgramInfo 参数查询

cl_program_info	返回类型	param_value 所返回的信息
CL_PROGRAM_REFERENCE_COUNT ¹²	cl_uint	返回 <i>program</i> 的引用计数。
CL_PROGRAM_CONTEXT	cl_context	返回创建程序对象时所指定的上下文。
CL_PROGRAM_NUM_DEVICES	cl_uint	返回 <i>program</i> 所关联设备的数目。
CL_PROGRAM_DEVICES	cl_device_id[]	返回程序对象所关联的设备列表。可以是程序对象的上下文所关联的设备,也可以是使用 clCreateProgramWithBinary 创建程序对象时所指定的设备的子集。
CL_PROGRAM_SOURCE	char[]	返回 clCreateProgramWithSource 所指定的程序源码。返回的是 clCreateProgramWithSource 使用的所有源码字符串的级联,并带有 null 终止符。 <i>param_value_size_ret</i> 中存储的是源码的实际字符数 (包含 null 终止符)。
CL_PROGRAM_BINARY_SIZES	size_t[]	返回一个数组,存放 <i>program</i> 所关联设备的程序二进制的大小。 <i>program</i> 所关联设备的数目就是此数组的元素个数。如果某个二进制无效,相应的数组元素就是 0。
CL_PROGRAM_BINARIES	unsigned char *[]	返回 <i>program</i> 所关联的所有设备的程序二进制。对于 <i>program</i> 中的每个设备,返回的可能是使用 clCreateProgramWithBinary 创建 <i>program</i> 时所指定的二进制,也可能是由 clBuildProgram 生成的可执行二进制;如果 <i>program</i> 是由 clCreateProgramWithSource 所创建,则返回的就是后者。所返回的比特可能是特定实现的中间表示 (a.k.a. IR),也可能是特定设备的可执行比特,或者二者兼有。具体要返回哪个由 OpenCL 实现来确定。 <i>param_value</i> 指向有 n 个元素的数组,每个元素都是一个指针,其中 n 是 <i>program</i> 所关联设备的数目。这些指针所指缓存的大小可以此表中的 CL_PROGRAM_BINARY_SIZES 来查询。 实现会将特定设备的二进制拷贝到相应数组元素所指内存中 (如果有的话)。要想知道数组中的程序二进制指的是哪个设备,使用 CL_PROGRAM_DEVICES 来获取设备列表。

¹² 所返回的引用计数会立刻过时,不适合应用中通常的用途。提供这个特性是为了定位内存泄露。

		CL_PROGRAM_BINARIES所返回的n个指针和 CL_PROGRAM_DEVICES 所返回的设备列表具有一对一的关系。
--	--	---

如果执行成功，**clGetProgramInfo** 返回 CL_SUCCESS；如果 *param_name* 无效，或者 *param_value_size* 的值 < 表 5.11 所列返回类型的大小且 *param_value* 不是 NULL，则返回 CL_INVALID_VALUE；如果 *program* 无效，则返回 CL_INVALID_PROGRAM。

函数

```
cl_int clGetProgramBuildInfo (cl_program program,
                                cl_device_id device,
                                cl_program_build_info param_name,
                                size_t param_value_size,
                                void *param_value,
                                size_t *param_value_size_ret)
```

返回程序对象中每个设备的构建信息。

program 指定要查询的程序对象。

device 表明要查询哪个设备的构建信息。*device* 必须是 *program* 所关联的有效设备。

param_name 指定要查询的信息类型。所支持的类型和返回的信息如表 5.12 所示。

param_value 指向存储查询结果所用的内存。如果是 NULL，则忽略。

param_value_size 表明 *param_value* 所指内存的大小。其值必须 ≥ 表 5.12 所列返回类型的大小。

param_value_size_ret 会返回拷贝到 *param_value* 中的数据的大小。如果是 NULL，则忽略。

表 5.12 clGetProgramBuildInfo 参数查询

cl_program_build_info	返回类型	param_value 中所返回的信息
CL_PROGRAM_BUILD_STATUS	cl_build_status	返回为 <i>device</i> 构建 <i>program</i> 的状态。此状态可能是下列之一： <ul style="list-style-type: none"> CL_BUILD_NONE，没有进行构建。 CL_BUILD_ERROR，如果最后一次调用 clBuildProgram(以 <i>program</i> 和 <i>device</i> 作为参数) 产生了错误。 CL_BUILD_SUCCESS，如果最后一次调用 clBuildProgram(以 <i>program</i> 和 <i>device</i> 作为参数) 是成功的。

		CL_BUILD_IN_PROGRESS, 如果最后一次调用 clBuildProgram (以 <i>program</i> 和 <i>device</i> 作为参数) 还没有完成。
CL_PROGRAM_BUILD_OPTIONS	char[]	返回 clBuildProgram 的参数 <i>options</i> 。 如果为 <i>device</i> 构建 <i>program</i> 的状态是 CL_BUILD_NONE, 则返回空字符串。
CL_PROGRAM_BUILD_LOG	char[]	返回以 <i>device</i> 为参数调用 clBuildProgram 所产生的构建日志。 如果为 <i>device</i> 构建 <i>program</i> 的状态是 CL_BUILD_NONE, 则返回空字符串。

如果执行成功, **clGetProgramBuildInfo** 会返回 CL_SUCCESS; 如果 *device* 不是 *program* 所关联的设备, 则返回 CL_INVALID_DEVICE; 如果 *param_name* 无效, 或者 *param_value_size* 的值 < 表 5.12 所列返回类型的大小且 *param_value* 不是 NULL, 则返回 CL_INVALID_VALUE; 如果 *program* 无效, 则返回 CL_INVALID_PROGRAM。

5.5 内核对象

内核就是程序中所声明的函数。程序中任何函数都可以通过增加限定符 `__kernel` 变成一个内核。一个内核对象包括一个 `__kernel` 函数和执行此函数时所用的参数值。

5.5.1 创建内核对象

使用此函数可以创建内核对象：

```
cl_kernel clCreateKernel (cl_program program,
                          const char *kernel_name,
                          cl_int *errcode_ret)
```

program 是带有成功构建的执行体的程序对象。

kernel_name 是程序中带有 `__kernel` 限定符的函数名字。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL, 就不会返回错误码。

如果成功创建了内核对象, **clCreateKernel** 会返回此对象并将 *errcode_ret* 置为 CL_SUCCESS。否则, 返回 NULL 并将 *errcode_ret* 置为下列错误值之一：

- ✚ CL_INVALID_PROGRAM, 如果 *program* 无效。
- ✚ CL_INVALID_PROGRAM_EXECUTABLE, 如果没有对 *program* 成功构建的执行体。
- ✚ CL_INVALID_KERNEL_NAME, 如果在 *program* 中没有找到 *kernel_name*。

- ✚ CL_INVALID_KERNEL_DEFINITION, 如果在所有设备为 *program* 所构建得执行体中, *kernel_name* 所指的 `_kernel` 函数的定义不一致, 如参数个数不同, 或参数类型不同等。
- ✚ CL_INVALID_VALUE, 如果 *kernel_name* 是 NULL。
- ✚ CL_OUT_OF_HOST_MEMORY, 如果为宿主机上的 OpenCL 实现分配其所需资源时失败。

函数

```
cl_int clCreateKernelsInProgram (cl_program program,
cl_uint num_kernels,
cl_kernel *kernels,
cl_uint *num_kernels_ret)
```

会为 *program* 中所有内核函数创建内核对象。对于 *program* 中的任一 `_kernel` 函数, 在为其成功构建程序执行体的所有设备上, 如果其定义不一致就不会为其创建内核对象。

program 是带有成功构建的执行体的程序对象。

num_kernels 是 *kernels* 所指内存的大小, 即 `cl_kernel` 的数目。

errcode_ret 用来返回错误码。如果 *errcode_ret* 是 NULL, 就不会返回错误码。

kernels 用来存储所返回的内核对象。如果是 NULL, 则忽略。否则, 必须大于等于 *program* 中内核的数目。

num_kernels_ret 是 *program* 中内核的数目。如果是 NULL, 则忽略。

如果成功分配了内核对象, 则 **clCreateKernelsInProgram** 会返回 CL_SUCCESS; 如果 *program* 无效, 则返回 CL_INVALID_PROGRAM; 如果没有为 *program* 中任何设备成功构建的执行体, 则返回 CL_INVALID_PROGRAM_EXECUTABLE; 如果 *kernel* 不是 NULL 且 *num_kernels* 小于 *program* 中内核的数目, 则返回 CL_INVALID_VALUE; 如果为宿主机上的 OpenCL 实现分配其所需资源失败, 则返回 CL_OUT_OF_HOST_MEMORY

只有当一个程序对象加载了有效的程序源码或二进制, 并为程序所关联的一个或多个设备成功构建了程序执行体, 这时才能创建内核对象。对于一个程序对象, 如果有内核对象与其相关联, 则不允许改变其程序执行体。这意味着, 如果有内核对象附到了程序对象上, 调用 **clBuildProgram** 将返回 CL_INVALID_OPERATION。 *program* 所关联的上下文也是 *kernel* 所关联的上下文。 *program* 所关联的设备也是 *kernel* 所关联的设备。对于程序对象所关联的设备, 如果为其构建了有效的程序执行体, 则可以用来执行此程序对象中声明的内核。

函数

```
cl_int clRetainKernel (cl_kernel s)
```

会增加 *kernel* 的引用计数。如果执行成功，**clRetainKernel** 返回 CL_SUCCESS。如果 *kernel* 无效，则返回 CL_INVALID_KERNEL；**clCreateKernel** 或 **clCreateKernelsInProgram** 会执行隐式保留。

函数

```
cl_int clReleaseKernel (cl_kernel kernel)
```

会减小 *kernel* 的引用计数。如果执行成功，**clReleaseKernel** 会返回 CL_SUCCESS；如果 *kernel* 无效，则返回 CL_INVALID_KERNEL。当引用计数减小到 0 并且所有命令都不再使用它时，此内核对象会被删除。

5.5.2 设置内核参数

要执行内核，必须设置内核参数。

函数

```
cl_int clSetKernelArg (cl_kernel kernel,  
                        cl_uint arg_index,  
                        size_t arg_size,  
                        const void *arg_value)
```

用来设置内核某个参数的值。

kernel 是一个有效的内核对象。

arg_index 是参数的索引。内核参数的索引从最左边参数的 0 到 n-1，其中 n 是内核所声明的参数的个数。

例如，下面的内核：

```
__kernel void  
image_filter (int n, int m,  
              __constant float *filter_weights,  
              __read_only image2d_t src_image,  
              __write_only image2d_t dst_image)  
{  
    ...  
}
```

对于 *image_filter* 的参数来讲，n 的索引是 0，m 的是 1，*filter_weights* 的是

2, `src_image` 的是 3, 而 `dst_image` 的是 4。

`arg_value` 所指数据用来作为索引为 `arg_index` 的参数。在 `clSetKernelArg` 返回后, `arg_value` 所指数据已经被拷贝, 其内存可以由应用重新使用。其参数值会被所有将 `kernel` 入队的 API 调用 (`clEnqueueNDRangeKernel` 和 `clEnqueueTask`) 所使用, 直到为 `kernel` 调用 `clSetKernelArg` 对其作出改变。

如果参数是一个内存对象, `arg_value` 就会指向相应对象。此对象必须是用 `kernel` 所关联的上下文创建的。如果参数是缓冲对象, 也可以将 `arg_value` 置为 `NULL`, 这时, 这个 `NULL` 就会用来作为内核中声明为 `__global` 或 `__constant` 内存指针的值。如果此参数在声明时带有限定符 `__local`, 则 `arg_value` 必须是 `NULL`。如果参数类型是 `sampler_t`, 则 `arg_value` 必须指向采样器对象。对于其他内核参数, `arg_value` 所指与实际参数类型必须一致。

如果所声明的参数类型是带有 `__global` 或 `__constant` 限定符的内建类型或用户自定义类型, 那么用作参数的内存对象必须是缓冲对象 (或 `NULL`)。如果参数带有限定符 `__constant`, 则此内存对象的大小不能超过 `CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE`, 且这样的参数的个数不能超过 `CL_DEVICE_MAX_CONSTANT_ARGS`。

如果所声明的参数类型是 `image2d_t`, 则指定为参数的内存对象必须是一个 2D 图像对象。而如果所声明的参数类型是 `image3d_t`, 则必须是一个 3D 图像对象。

`arg_size` 表明参数值的大小。如果参数是内存对象, 则其值为缓冲对象或图像对象的大小。如果所声明参数带有限定符 `__local`, 其值将是为 `__local` 参数所分配的缓存的大小。如果参数类型是 `sampler_t`, `arg_size` 的值必须等于 `sizeof(cl_sampler)`。对于其他参数, 必须是相应参数类型的大小。

注意: 内核对象不会更新对象的引用计数, 如由 `clSetKernelArg` 设置为参数的内存、采样器对象; 用户不要指望内核对象会对作为其参数的对象进行保留操作¹³。

如果执行成功, `clSetKernelArg` 会返回 `CL_SUCCESS`。否则, 返回下列错误之一:

- ✚ `CL_INVALID_KERNEL`, 如果 `kernel` 无效。
- ✚ `CL_INVALID_ARG_INDEX`, 如果 `arg_index` 无效。
- ✚ `CL_INVALID_ARG_VALUE`, 如果参数在声明时不带限定符 `__local`, 但 `arg_value`

¹³ 实现不能让 `cl_kernel` 对象拥有其参数的引用计数。因为没有为用户提供任何机制去告诉内核释放此所有权。如果内核拥有其参数的所有权, 用户就不可能确切知道什么时候可以安全释放为其所分配的资源, 如用 `CL_MEM_USE_HOST_PTR` 对 `cl_mem` 进行回写。

是 NULL，反之亦然。

- ✚ CL_INVALID_MEM_OBJECT，如果一个参数其类型是内存对象，但 *arg_value* 所指内存对象却无效。
- ✚ CL_INVALID_SAMPLER，如果一个参数其类型是 *sampler_t*，但 *arg_value* 所指采样器对象却无效。
- ✚ CL_INVALID_ARG_SIZE，如果一个参数不是内存对象，且 *arg_size* 与数据类型大小不一致；或者是内存对象，但 *arg_size* != sizeof(*cl_mem*)；或者 *arg_size* 是 0 且参数在声明时带有限定符 *_local*；或者参数是采样器，但 *arg_size* != sizeof(*cl_sampler*)。

5.5.3 内核对象查询

函数

```
cl_int clGetKernelInfo (cl_kernel kernel,
                        cl_kernel_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret)
```

返回内核对象的信息。

kernel 指定要查询的内核对象。

param_name 指定要查询的信息。所支持的类型和返回的信息如表 5.13 所示。

param_value 指向存储查询结果的内存。如果是 NULL，则忽略。

param_value_size 表明 *param_value* 所指内存的大小。其值必须 ≥ 表 5.13 所列返回类型的大小。

param_value_size_ret 会返回拷贝到 *param_value* 中的数据的大小。如果是 NULL，则忽略。

表 5.13 clGetKernelInfo 参数查询

cl_kernel_info	返回类型	param_value 中返回的信息
CL_KERNEL_FUNCTION_NAME	char[]	返回内核函数的名字。
CL_KERNEL_NUM_ARGS	cl_uint	返回 <i>kernel</i> 参数的数目。
CL_KERNEL_REFERENCE_COUNT ¹⁴	cl_uint	返回 <i>kernel</i> 的引用计数。

¹⁴ 所返回的引用计数会立刻过时，不适合应用中通常的用途。提供这个特性是为了定位内存泄露。

CL_KERNEL_CONTEXT	cl_context	返回 <i>kernel</i> 所关联的上下文。
CL_KERNEL_PROGRAM	cl_program	返回 <i>kernel</i> 所关联的程序对象。

如果执行成功, **clGetKernelInfo** 返回 CL_SUCCESS ; 如果 *param_name* 无效, 或者 *param_value_size* 的值 < 表 5.13 所列返回类型的大小且 *param_value* 不是 NULL, 则返回 CL_INVALID_VALUE ; 如果 *kernel* 无效, 则返回 CL_INVALID_KERNEL。

函数

```
cl_int clGetKernelWorkGroupInfo (cl_kernel kernel,
                                cl_device_id device,
                                cl_kernel_work_group_info param_name,
                                size_t param_value_size,
                                void *param_value,
                                size_t *param_value_size_ret)
```

会返回某个特定设备上的内核对象的信息。

kernel 指定要查询的内核对象。

device 标识 *kernel* 所关联的设备列表中的某个特定设备。这个设备列表是 *kernel* 所关联的 OpenCL 上下文中的设备列表。如果 *kernel* 所关联的设备列表中只有一个设备, *device* 可以是 NULL。

param_name 指定要查询的信息。所支持的类型和返回的信息如表 5.14 所示。

param_value 指向存储查询结果的内存。如果是 NULL, 则忽略。

param_value_size 表明 *param_value* 所指内存的大小。其值必须 ≥ 表 5.14 所列返回类型的大小。

param_value_size_ret 会返回拷贝到 *param_value* 中的数据的大小。如果是 NULL, 则忽略。

表 5.14 clGetKernelWorkGroupInfo 参数查询

cl_kernel_work_group_info	返回类型	param_value 中所返回的信息
CL_KERNEL_WORK_GROUP_SIZE	size_t	为应用提供一种机制, 来查询工作组大小的最大值, 这些工作组可以用来在 <i>device</i> 上执行内核。OpenCL 实现会根据内核的资源需求 (如寄存器的使用等) 来确定工作组需要多大。
CL_KERNEL_COMPILE_WORK_GROUP_SIZE	size_t[3]	返回由限定符 <code>__attribute__((reqd_work_group_size</code>

		(<i>x</i> , <i>y</i> , <i>z</i>)))所指定的工作组大小。参见节 6.7.2。 如果没有用上面的特性限定符指定大小, 则返回(0,0,0)。
CL_KERNEL_LOCAL_MEM_SIZE	cl_ulong	返回内核所用局部内存的大小, 这包括实现执行内核时所需局部内存、内核中所声明的带有 <code>__local</code> 地址限定符的变量、要为内核中声明为带 <code>__local</code> 地址限定符的指针参数所分配的局部内存 (其大小由 clSetKernelArg 指定)。 对于任一内核中声明为带 <code>__local</code> 地址限定符的指针参数, 如果没有指定其大小, 则假定为 0。

如果执行成功, **clGetKernelWorkGroupInfo** 返回 CL_SUCCESS; 如果 *device* 不在 *kernel* 所关联设备列表之中, 或者 *device* 是 NULL 但 *kernel* 所关联的设备多于一个, 则返回 CL_INVALID_DEVICE; 如果 *param_name* 无效, 或者 *param_value_size* 的值 < 表 5.14 所列返回类型的大小且 *param_value* 不是 NULL, 则返回 CL_INVALID_VALUE; 如果 *kernel* 无效, 则返回 CL_INVALID_KERNEL。

5.6 执行内核

函数

```
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)
```

会将一个命令入队, 此命令用来在设备上执行内核。

command_queue 是一个有效的命令队列。内核将被入队并在 *command_queue* 所关联的设备上执行。

kernel 是一个有效的内核对象。*kernel* 所关联的 OpenCL 上下文必须和 *command_queue* 所关联的保持一致。

work_dim 是维数, 用来指定全局工作项和工作组中的工作项。*work_dim* 必须大于 0, 且小于等于 3。

global_work_offset 在当前必须是 NULL。将来修订 OpenCL 时, 此参数可能用来指

定一组 *work_dim* 无符号值作为偏移量，用来计算工作项的**全局 ID**，而不必非得知道从偏移位置(0,0, ...,0)起始的全局 ID。

*global_work_size*¹⁵指向一组 *work_dim* 无符号值，描述在 *work_dim* 维度中用来执行内核函数的全局工作项的数目。全局工作项的数目这样计算：*global_work_size*[0] * ... * *global_work_size*[*work_dim* - 1]。

*local_work_size*指向一组 *work_dim* 无符号值，描述组成一个工作组（会执行 *kernel*）的工作项的数目（也是指工作组的大小）。工作组中工作项的总数可以这样算得 *local_work_size*[0] * ... * *local_work_size*[*work_dim* - 1]，此数目必须小于等于 *CL_DEVICE_MAX_WORK_GROUP_SIZE*（参见表 4.3），而且 *local_work_size*[*x*], *x* ∈ [0, *work_dim* - 1] 必须小于等于相应的 *CL_DEVICE_MAX_WORK_ITEM_SIZES*[*x*]。*local_work_size* 将用来确定怎样将 *global_work_size* 所指定的全局工作项放入恰当的工作组实例中。如果指定了 *local_work_size*，*global_work_size*[*x*], *x* ∈ [0, *work_dim* - 1] 的值必须可以被相应的 *local_work_size*[*x*] 整除。

kernel 所用工作组的大小可以在程序源码中用限定符 `__attribute__((reqd_work_group_size(X, Y, Z)))` 来指定（参见节 6.7.2）。这种情况下，*local_work_size* 所指定的工作组大小必须与属性限定符 *reqd_work_group_size* 所指定的值相一致。

local_work_size 也可以是 NULL，这样将由 OpenCL 实现决定怎样将全局工作项放入恰当的工作组实例中。

这些工作组实例可能在多个计算单元上并行执行，也可能在同一个计算单元上并发执行。

每个工作项都可以由一个唯一的全局 ID 来识别。这个全局 ID 可以在内核内读取，由 *global_work_size* 和 *global_work_offset* 来计算。在 OpenCL 1.0 中，全局 ID 都是从 (0,0, ...,0) 开始的。另外，一个工作项也可以由所在工作组中的一个唯一局部 ID 来识别。局部 ID 也可以由内核读取，用 *local_work_size* 来计算。局部 ID 都是从 (0,0, ...,0) 开始。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 NULL，那么此命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL，则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL，

¹⁵ *global_work_size* 的值不能超过设备（内核在其上执行）的 *sizeof(size_t)* 的范围。设备的 *sizeof(size_t)* 可以用表 4.3 中的 *CL_DEVICE_ADDRESS_BITS* 来确定。如果，例如，*CL_DEVICE_ADDRESS_BITS* = 32，即，设备使用 32 位的地址空间，*size_t* 是一个 32 位无符号整形，且 *global_work_size* 的值必须在 $1 \dots 2^{32} - 1$ 的范围内。此范围以外的值会导致返回 *CL_OUT_OF_RESOURCES* error。

*event_wait_list*所指事件列表必须是有效的，且 *num_events_in_wait_list* 必须大于 0。
event_wait_list 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 返回一个事件对象，用来识别特定内核执行实例。事件对象是唯一的，稍后可以用来识别特定内核执行实例。如果 *event* 是 NULL，不会为此内核执行实例创建任何事件，因此应用不能查询或等待此特定内核执行实例。

如果内核执行体成功进行排队，则 **clEnqueueNDRangeKernel** 会返回 CL_SUCCESS。否则，返回下列错误之一：

- ✚ CL_INVALID_PROGRAM_EXECUTABLE，如果没有为 *command_queue* 所关联设备成功构建的程序执行体。
- ✚ CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 无效。
- ✚ CL_INVALID_KERNEL，如果 *kernel* 无效。
- ✚ CL_INVALID_CONTEXT，如果 *command_queue* 所关联的上下文与 *kernel* 所关联的不一致，或者与 *event_wait_list* 中事件所关联的不一致。
- ✚ CL_INVALID_KERNEL_ARGS，如果没有指定内核参数值。
- ✚ CL_INVALID_WORK_DIMENSION，如果 *work_dim* 的值无效（即不在 1 和 3 之间）。
- ✚ CL_INVALID_GLOBAL_WORK_SIZE，如果 *global_work_size* 是 NULL，或者 *global_work_size*[0], ... *global_work_size*[*work_dim* - 1] 中的任何一个为 0 或者超过了用来执行内核的设备的 `sizeof(size_t)` 的范围。for the device on which the kernel execution will be enqueued.
- ✚ CL_INVALID_WORK_GROUP_SIZE，如果指定了 *local_work_size*，且工作项的数目（*global_work_size*）不能被工作组的大小（*local_work_size*）所整除，或者与程序源码中用限定符 `__attribute__((reqd_work_group_size(X, Y, Z)))` 为 *kernel* 指定的工作组大小不匹配。
- ✚ CL_INVALID_WORK_GROUP_SIZE，如果指定了 *local_work_size*，且工作组中的工作项总数（*local_work_size*[0] * ... * *local_work_size*[*work_dim* - 1]）大于 CL_DEVICE_MAX_WORK_GROUP_SIZE（参见表 4.3）。
- ✚ CL_INVALID_WORK_GROUP_SIZE，如果 *local_work_size* 是 NULL，且程序源码中使用限定符 `__attribute__((reqd_work_group_size(X, Y, Z)))` 为 *kernel* 指定了工作组大小。
- ✚ CL_INVALID_WORK_ITEM_SIZE，如果 *local_work_size*[*x*], *x* ∈ [0, *work_dim* - 1] 中的任意一个大于相应的 CL_DEVICE_MAX_WORK_ITEM_SIZES[*x*]。
- ✚ CL_INVALID_GLOBAL_OFFSET，如果 *global_work_offset* 是 NULL。
- ✚ CL_OUT_OF_RESOURCES，如果将 *kernel* 的可执行实例加入到命令队列时，由于执行内核所需的资源不足而失败。例如，由于寄存器或局部内存等资源不足，指

定的 *local_work_size* 导致执行内核失败。另一个例子是 *kernel* 中所用只读图像参数的数目超过了设备的 `CL_DEVICE_MAX_READ_IMAGE_ARGS`，或者 *kernel* 所用只写图像参数的数目超过了设备的 `CL_DEVICE_MAX_WRITE_IMAGE_ARGS`，或者 *kernel* 所用采样器的数目超过了设备的 `CL_DEVICE_MAX_SAMPLERS`。

- ✚ `CL_MEM_OBJECT_ALLOCATION_FAILURE`，如果在为 *kernel* 参数中的内存对象分配数据仓库所需内存时失败。
- ✚ `CL_INVALID_EVENT_WAIT_LIST`，如果 *event_wait_list* 是 `NULL` 且 *num_events_in_wait_list* > 0，或者 *event_wait_list* 不是 `NULL` 且 *num_events_in_wait_list* 是 0，或者 *event_wait_list* 中的事件对象无效。
- ✚ `CL_OUT_OF_HOST_MEMORY`，如果为宿主主机上的 OpenCL 实现分配所需资源时失败。

函数

```
cl_int clEnqueueTask (cl_command_queue command_queue,
                      cl_kernel kernel,
                      cl_uint num_events_in_wait_list,
                      const cl_event *event_wait_list,
                      cl_event *event)
```

会将一个命令入队，此命令用于在设备上执行内核。内核在执行时仅使用单个工作项。

command_queue 是一个有效的命令队列。内核将被入队并在 *command_queue* 所关联的设备上执行。

kernel 是一个有效的内核对象。*kernel* 所关联的 OpenCL 上下文必须和 *command_queue* 所关联的保持一致。

event_wait_list 和 *num_events_in_wait_list* 指定在执行此命令前必须完成的事件。如果 *event_wait_list* 是 `NULL`，那么此殊命令不用等待任何事件的完成。如果 *event_wait_list* 是 `NULL`，则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 `NULL`，*event_wait_list* 所指事件列表必须是有效的，且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。*event_wait_list* 中的事件和 *command_queue* 所关联的上下文必须一致。

event 返回一个事件对象，用来识别特定内核执行实例。事件对象是唯一的，稍后可以用来识别特定内核执行实例。如果 *event* 是 `NULL`，不会为此内核执行实例创建任何事件，因此应用不能查询或等待此特定内核执行实例。

clEnqueueTask 相当于调用 **clEnqueueNDRangeKernel** 时 *with work_dim = 1*，*global_work_offset = NULL*，*global_work_size[0]* 置为 1，*local_work_size[0]* 置为 1。

如果内核执行体成功进行排队，则 **clEnqueueTask** 会返回 CL_SUCCESS。否则，返回下列错误之一：

- ✚ CL_INVALID_PROGRAM_EXECUTABLE，如果没有为 *command_queue* 所关联设备成功构建的程序执行体。
- ✚ CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 无效。
- ✚ CL_INVALID_KERNEL，如果 *kernel* 无效。
- ✚ CL_INVALID_CONTEXT，如果 *command_queue* 所关联的上下文与 *kernel* 所关联的不一致，或者与 *event_wait_list* 中事件所关联的不一致。
- ✚ CL_INVALID_KERNEL_ARGS，如果没有指定设备参数值。
- ✚ CL_INVALID_WORK_GROUP_SIZE，如果程序源码中用限定符 `__attribute__((reqd_work_group_size(X, Y, Z)))` 为 *kernel* 指定了工作组大小，且不是(1,1,1)。
- ✚ CL_OUT_OF_RESOURCES，如果将 *kernel* 的可执行实例加入到命令队列时，由于执行内核所需的资源不足而失败。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE，如果在为 *kernel* 参数中的内存对象分配数据仓库所需内存时失败。
- ✚ CL_INVALID_EVENT_WAIT_LIST，如果 *event_wait_list* 是 NULL 且 *num_events_in_wait_list* > 0，或者 *event_wait_list* 不是 NULL 且 *num_events_in_wait_list* 是 0，或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主主机上的 OpenCL 实现分配其所需资源时失败。

函数

```
cl_int clEnqueueNativeKernel (cl_command_queue command_queue,
                               void (*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)
```

入队一个命令，来执行一个原生的（不是用 OpenCL 编译器编译的）C/C++ 函数。

command_queue 是一个有效的命令队列。原生的用户函数要想在此命令队列中执行，则设备的 CL_DEVICE_EXECUTION_CAPABILITIES 中必须设置了 CL_EXEC_NATIVE_KERNEL（参见表 4.3）。

user_func 指向一个可被宿主主机调用的用户函数。

args 指向调用 *user_func* 时所用的参数列表。

cb_args 是 *args* 所指的参数列表的大小。

arg 所指的数据 (大小是 *cb_args*) 会被拷贝一份, 会将指向这份拷贝的指针传递给 *user_func*。为什么要拷贝一份, 因为 *args* 可能包含一些内存对象 (*cl_mem*), 这些对象可能会被修改并被指向全局内存的指针所替代。当 **clEnqueueNativeKernel** 返回时, *args* 所指的内存区域可以被应用重新使用。

num_mem_objects 是 *args* 所传递的缓冲对象的数目。

mem_list 是一组有效的缓冲对象, 如果 *num_mem_objects* > 0。 *mem_list* 中的缓冲对象由 **clCreateBuffer** 所返回的内存对象句柄 (*cl_mem*) 或是 NULL。

args_mem_loc 所指位置即 *args* 所指用来存储内存对象句柄 (*cl_mem*) 的位置。在执行用户函数之前, 这些内存对象句柄会被指向全局内存的指针所代替。

event_wait_list, *num_events_in_wait_list* 和 *event* 与 **clEnqueueNDRangeKernel** 中所描述的一致。

如果用户函数的执行实例成功入队, **clEnqueueNativeKernel** 会返回 CL_SUCCESS。否则, 返回下列错误之一:

- ✚ CL_INVALID_COMMAND_QUEUE, 如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT, 如果 *command_queue* 所关联的上下文与 *event_wait_list* 中的事件所关联的上下文不一致。
- ✚ CL_INVALID_VALUE, 如果 *user_func* 是 NULL。
- ✚ CL_INVALID_VALUE, 如果 *args* 是 NULL 且 *cb_args* > 0, 或者 *args* 是 NULL 且 *num_mem_objects* > 0。
- ✚ CL_INVALID_VALUE, 如果 *args* 不是 NULL 且 *cb_args* 是 0。
- ✚ CL_INVALID_VALUE, 如果 *num_mem_objects* > 0 且 *mem_list* 或 *args_mem_loc* 是 NULL。
- ✚ CL_INVALID_VALUE, 如果 *num_mem_objects* = 0 且 *mem_list* 或 *args_mem_loc* 不是 NULL。
- ✚ CL_INVALID_OPERATION, 如果 *device* 不能执行原生函数。
- ✚ CL_INVALID_MEM_OBJECT, 如果 *mem_list* 中的一个或多个内存对象无效或者不是缓冲对象。
- ✚ CL_OUT_OF_RESOURCES, 如果由于执行内核所需资源不足, 将 *kernel* 的执行实例入队时失败。
- ✚ CL_MEM_OBJECT_ALLOCATION_FAILURE, 如果为 *kernel* 参数中的缓冲对象分

配数据仓库所需内存时失败。

- ✚ CL_INVALID_EVENT_WAIT_LIST, 如果 *event_wait_list* 是 NULL 且 *num_events_in_wait_list* > 0, 或者 *event_wait_list* 不是 NULL 且 *num_events_in_wait_list* 是 0, 或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_OUT_OF_HOST_MEMORY, 如果为宿主机上的 OpenCL 实现分配其所需资源失败。

5.7 事件对象

事件对象可以用来指内核执行命令(**clEnqueueNDRangeKernel**, **clEnqueueTask**, **clEnqueueNativeKernel**), 或读、写、映射和拷贝内存对象的命令 (**clEnqueue{Read|Write|Map}{Buffer|Image}**, **clEnqueueCopy{Buffer|Image}**, **clEnqueueCopyBufferToImage**, 或 **clEnqueueCopyImageToBuffer**)。

事件对象可以用来跟踪命令的执行状态。将命令入队的 API 调用会创建一个新的事件对象, 并在参数 *event* 中返回。当然如果将命令入队失败, 则不会返回事件对象。

在任意时刻, 所入队命令的执行状态可能是:

- ✚ CL_QUEUED, 命令已经入队;
- ✚ CL_SUBMITTED, 所入队命令已经由宿主机提交给命令队列所关联的设备;
- ✚ CL_RUNNING, 设备正在执行此命令;
- ✚ CL_COMPLETE, 命令已经成功完成; 或者
- ✚ 错误码, 如果命令异常终止 (可能由无效内存访问等所引起)。

已经终止的命令所返回的错误码是一个负整数。如果命令的执行状态是 CL_COMPLETE 或一个负整数, 则认为此命令已经完成。

如果命令的执行已经终止, 此命令所关联的命令队列, 和所关联的上下文 (和此上下文中的所有其他命令队列) 可能不再可用。现在, 使用此上下文 (或与此上下文所关联的命令队列) 的 OpenCL API 调用的行为依赖于具体实现。创建上下文时所指定的用户注册的回调函数可以用来报告相应的错误信息。

函数

```
cl_int clWaitForEvents (cl_uint num_events,
                       const cl_event *event_list)
```

等待执行命令 (由 *event_list* 中的事件对象来识别) 的宿主机线程完成。对于一个命令, 如果其执行状态是 CL_COMPLETE 或负整数, 则可以认为此命令已经完成。 *event_list* 中的事件可以充当同步点。

如果函数执行成功 ,**clWaitForEvents** 会返回 CL_SUCCESS。如果 *num_events* 是 0 , 则返回 CL_INVALID_VALUE ; 如果 *event_list* 中的事件不属于同一个上下文 , 则返回 CL_INVALID_CONTEXT 如果 *event_list* 中的事件对象无效 则返回 CL_INVALID_EVENT。

函数

```
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)
```

返回某事件对象的信息。

event 指定所要查询的事件对象。

param_name 指定所要查询的信息。所支持的类型和返回的信息见表 5.15。

param_value 指向存储返回信息的内存。如果是 NULL , 则忽略。

param_value_size 表明 *param_value* 所指内存的大小。其值必须>=表 5.15 所列返回类型的大小。

param_value_size_ret 会返回拷贝到 *param_value* 中的数据的大小。如果是 NULL , 则忽略。

表 5.15 clGetEventInfo 参数查询

cl_event_info	返回类型	param_value 中所返回的信息
CL_EVENT_COMMAND_QUEUE	cl_command_queue	返回与 <i>event</i> 相关联的命令队列。
CL_EVENT_COMMAND_TYPE	cl_command_type	返回 <i>event</i> 所关联命令的类型： CL_COMMAND_NDRANGE_KERNEL CL_COMMAND_TASK CL_COMMAND_NATIVE_KERNEL CL_COMMAND_READ_BUFFER CL_COMMAND_WRITE_BUFFER CL_COMMAND_COPY_BUFFER CL_COMMAND_READ_IMAGE CL_COMMAND_WRITE_IMAGE CL_COMMAND_COPY_IMAGE CL_COMMAND_COPY_BUFFER_TO_IMAGE

		CL_COMMAND_COPY_IMAGE_T O_BUFFER CL_COMMAND_MAP_BUFFER CL_COMMAND_MAP_IMAGE CL_COMMAND_UNMAP_MEM_ OBJECT CL_COMMAND_MARKER CL_COMMAND_ACQUIRE_GL_O BJECTS CL_COMMAND_RELEASE_GL_O BJECTS
CL_EVENT_COMMAND_EXECUTION_S TATUS	cl_int	返回由 <i>event</i> 来识别的命令的执行 状态： CL_QUEUED CL_SUBMITTED CL_RUNNING CL_COMPLETE 错误码（负整数）
CL_EVENT_REFERENCE_COUNT ¹⁶	cl_uint	返回 <i>event</i> 的引用计数

使用 **clGetEventInfo** 来确定一个由 *event* 来识别的命令是否执行完毕（即 CL_EVENT_COMMAND_EXECUTION_STATUS 返回 CL_COMPLETE），这不是一个同步点。不保证 *event* 所关联的命令要修改的内存对象对其他命令可见。

如果执行成功，**clGetEventInfo** 返回 CL_SUCCESS；如果 *param_name* 无效，或者 *param_value_size* 的值 < 表 5.15 所列返回类型的大小且 *param_value* 不是 NULL，则返回 CL_INVALID_VALUE；如果 *event* 无效，则返回 CL_INVALID_EVENT。

函数

```
cl_int clRetainEvent (cl_event event)
```

会增加 *event* 的引用计数。如果执行成功，**clRetainEvent** 返回 CL_SUCCESS。如果 *event* 无效，则返回 CL_INVALID_EVENT，返回一个事件的 OpenCL 命令会执行隐式保留。

要释放一个事件，使用下列函数

```
cl_int clReleaseEvent (cl_event event)
```

会减小 *event* 的引用计数。如果执行成功，**clReleaseEvent** 会返回 CL_SUCCESS；如

¹⁶ 所返回的引用计数会立刻过时，不适合应用中通常的用途。提供这个特性是为了定位内存泄露。

果 *event* 无效，则返回 `CL_INVALID_EVENT`。当引用计数减小到 0 并，此事件所标识的命令已经完成（或终止）且命令队列中没有命令需要等待此事件完成，此事件对象会被删除。

5.8 内核和内存对象命令的乱序执行

提交给一个命令队列的 OpenCL 函数是按被调用的顺序入队的，但可以配置成顺序执行或乱序执行。`clCreateCommandQueue` 的参数 *properties* 可以用来指定执行顺序。

如果某个命令队列的属性 `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` 没有设置，则其中的命令会顺序执行。例如，如果应用先调用 `clEnqueueNDRangeKernel` 将内核 A 入队，随后调用 `clEnqueueNDRangeKernel` 将内核 B 入队，应用可以假定内核 A 先完成，然后才执行内核 B。如果内核 A 所输出的内核对象是内核 B 的输入，则执行内核 A 所产生的内核对象中的正确数据可以被内核 B 所看到。而如果设置了属性 `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`，则不保证开始执行内核 B 之前，内核 A 会完成。

通过设置命令队列的属性 `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`，应用可以将其中的命令配置成乱序执行。这可以在创建命令队列时指定，也可以用 `clSetCommandQueueProperty` 进行动态修改。在乱序执行模式下，不保证命令按入队的顺序完成执行。鉴于不保证内核顺序执行，即，如果命令队列中调用了 `clEnqueueNDRangeKernel`，之前调用 `clEnqueueNDRangeKernel` 来执行内核 A（用事件 A 来标识），后来又调用 `clEnqueueNDRangeKernel` 执行内核 B，但是 A 的执行和/或完毕可能会晚于内核 B。要保证内核按特定顺序执行，可以使用等待事件的方法（此时是事件 A）。对于内核 B，调用 `clEnqueueNDRangeKernel` 时，参数 *event_wait_list* 中可以指定要等待的事件 A。

另外，可以将等待事件或隔层命令入队。等待事件的命令可以保证，在下一批命令执行之前，之前入队的命令（由要等待的事件列表来标识）已经完成。隔层命令可以保证，在下一批命令执行之前，之前入队的命令已经全部完成。

类似的，如果读、写、拷贝或映射内存对象的命令在命令 `clEnqueueNDRangeKernel`、`clEnqueueTask` 或 `clEnqueueNativeKernel` 之后入队，不保证会等待被调度执行的内核完成（如果设置了属性 `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`）。要保证命令的正确顺序，`clEnqueueNDRangeKernel`、`clEnqueueTask` 或 `clEnqueueNativeKernel` 所返回的事件对象可以用来将对事件的等待入队，或者可以入队一个隔层命令，在读写内存对象前此命令必须完成。

函数

```
cl_int clEnqueueMarker (cl_command_queue command_queue,
                       cl_event *event)
```

会入队一个记号命令。此命令所返回的 *event* 可以用来入队一个等待，等待此记号事件，即，等待在此记号命令之前入队的所有命令全部完成。

如果执行成功，**clEnqueueMarker** 会返回 CL_SUCCESS。如果 *command_queue* 无效则返回 CL_INVALID_COMMAND_QUEUE；如果 *event* 是 NULL，则返回 CL_INVALID_VALUE；如果为宿主机上的 OpenCL 实现分配其所需资源失败，则返回 CL_OUT_OF_HOST_MEMORY。

函数

```
cl_int clEnqueueWaitForEvents (cl_command_queue command_queue,
                                cl_uint num_events,
                                const cl_event *event_list)
```

会入队一个等待命令，此后入队的命令在执行之前，必须等待某个事件或一系列事件完成。*num_events* 表明 *event_list* 中事件的数目。*event_list* 中的每个事件都必须是之前调用下列函数所返回的一个有效事件对象：**clEnqueueNDRangeKernel**、**clEnqueueTask**、**clEnqueueNativeKernel**、**clEnqueue{Read|Write|Map}{Buffer|Image}**、**clEnqueueCopy{Buffer|Image}**、**clEnqueueCopyBufferToImage**、**clEnqueueCopyImageToBuffer** 或 **clEnqueueMarker**。*event_list* 中的事件所关联的上下文必须和 *command_queue* 所关联的上下文保持一致。

event_list 中的事件充当同步点。

如果执行成功，**clEnqueueWaitForEvents** 会返回 CL_SUCCESS。如果 *command_queue* 无效则返回 CL_INVALID_COMMAND_QUEUE；如果 *command_queue* 所关联的上下文与 *event_list* 中的事件所关联的上下文不一致，则返回 CL_INVALID_CONTEXT；如果 *num_events* 是 0 或 *event_list* 是 NULL，则返回 CL_INVALID_VALUE；如果 *event_list* 中的事件对象无效，则返回 CL_INVALID_EVENT；如果为宿主机上的 OpenCL 实现分配其所需资源失败，则返回 CL_OUT_OF_HOST_MEMORY。

函数

```
cl_int clEnqueueBarrier (cl_command_queue command_queue)
```

入队一个隔层操作。此命令可以保证，在下一批命令执行之前，*command_queue* 中的所有命令一定完成。命令 **clEnqueueBarrier** 是一个同步点。

如果执行成功，**clEnqueueBarrier** 返回 CL_SUCCESS。如果 *command_queue* 无效，则返回 CL_INVALID_COMMAND_QUEUE；而如果为宿主机上的 OpenCL 实现分配其所需资源失败，则返回 CL_OUT_OF_HOST_MEMORY。

5.9 内存对象和内核上的profile操作

本节将描述对于作为命令入队的OpenCL函数的profile。这些函数¹⁷指的是：
clEnqueue{Read|Write|Map}Buffer、**clEnqueue{Read|Write|Map}Image**、
clEnqueueCopy{Buffer|Image}、**clEnqueueCopyImageToBuffer**、
clEnqueueCopyBufferToImage、**clEnqueueNDRangeKernel**、**clEnqueueTask**、
clEnqueueNativeKernel和**clEnqueueMarker**。

事件对象可以用来捕获（度量命令执行时间的）profile 信息。用 **clCreateCommandQueue** 创建命令队列时，在参数 *properties* 中设置标志位 **CL_QUEUE_PROFILING_ENABLE**，或者在 **clSetCommandQueueProperty** 的参数 *properties* 中设置标志位 **CL_QUEUE_PROFILING_ENABLE**，这两种方法都可以使能对 OpenCL 命令的 profile。

函数

```
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)
```

返回 *event* 所关联命令的 profile 信息。

event 指定事件对象。

param_name 指定要查询的信息。所支持的类型和返回的信息如表 5.16 所示。

param_value 指向存储查询结果的内存。如果是 NULL，则忽略。

param_value_size 表明 *param_value* 所指内存的大小。其值必须 ≥ 表 5.16 所列返回类型的大小。

param_value_size_ret 会返回拷贝到 *param_value* 中的数据的大小。如果是 NULL，则忽略。

表 5.16 clGetEventProfilingInfo 参数查询

cl_profiling_info	返回类型	param_value 中所返回的信息
	型	

¹⁷ 节 9.12.6 中定义的 **clEnqueueAcquireGLObjects** 和 **clEnqueueReleaseGLObjects** 也包括在内。

CL_PROFILING_COMMAND_QUEUED	cl_ulong	一个 64 位值，用来描述当 <i>event</i> 所标识的命令入队时，设备上的时间计数器的值，单位纳秒。
CL_PROFILING_COMMAND_SUBMIT	cl_ulong	一个 64 位值，用来描述当宿主机将 <i>event</i> 所标识的命令提交给命令队列所关联的设备时，设备上的时间计数器的值，单位纳秒。
CL_PROFILING_COMMAND_START	cl_ulong	一个 64 位值，用来描述当宿主机将 <i>event</i> 所标识的命令在设备上开始执行时，设备上的时间计数器的值，单位纳秒。
CL_PROFILING_COMMAND_END	cl_ulong	一个 64 位值，用来描述当宿主机将 <i>event</i> 所标识的命令在设备上执行完毕时，设备上的时间计数器的值，单位纳秒。

所返回的无符号 64 位值可以用来度量 OpenCL 命令所耗费的时间（单位为纳秒）。

要求在设备频率和电源状态发生变化时，OpenCL 设备可以正确的跟踪时间。

CL_DEVICE_PROFILING_TIMER_RESOLUTION 指定定时器的分辨率，即，增加定时器的计数时所过去的纳秒数。

如果函数执行成功，且记录了 profile 信息，**clGetEventProfilingInfo** 会返回 CL_SUCCESS；如果没有为命令队列设置标志位 CL_QUEUE_PROFILING_ENABLE，且 profile 信息当前不可用（由于 *event* 所标识的命令没有完成），则返回 CL_PROFILING_INFO_NOT_AVAILABLE；如果 *param_name* 无效，或者 *param_value_size* 的值 < 表 5.16 所列返回类型的大小且 *param_value* 不是 NULL，则返回 CL_INVALID_VALUE；如果 *event* 无效，则返回 CL_INVALID_EVENT。

5.10 刷新（flush）和完成（finish）

函数

```
cl_int clFlush (cl_command_queue command_queue)
```

将之前入队的 OpenCL 命令提交给命令队列关联的设备。**clFlush** 仅保证将命令提交给设备，但不保证这些命令会在 **clFlush** 返回前完成。

如果执行成功，**clFlush** 会返回 CL_SUCCESS。如果 *command_queue* 无效，则返回 CL_INVALID_COMMAND_QUEUE；如果为宿主机上的 OpenCL 实现分配其所需资源失败，则返回 CL_OUT_OF_HOST_MEMORY。

命令队列中任何阻塞的命令，如 **clEnqueueRead{Image|Buffer}**（*blocking_read* 设置为 CL_TRUE）、**clEnqueueWrite{Image|Buffer}**（*blocking_write* 设置为 CL_TRUE）、**clEnqueueMap{Buffer|Image}**（*blocking_map* 设置为 CL_TRUE）或者 **clWaitForEvents**，都会对此命令队列执行隐式的刷新。

对于事件对象,如果命令队列中存在等待它的命令(此命令由另一个命令队列中的命令入队),要想使用它,应用必须调用 **clFlush** 或任一阻塞的命令(对命令队列执行隐式刷新,此处的命令指入队的事件对象)。

函数

```
cl_int clFinish (cl_command_queue command_queue)
```

会阻塞,直到之前入队的所有 OpenCL 命令全部提交并完成。直到 *command_queue* 中所有命令被处理并完成后, **clFinish** 才会返回。**clFinish** 也是一个同步点。

如果执行成功, **clFinish** 会返回 CL_SUCCESS。如果 *command_queue* 无效,则返回 CL_INVALID_COMMAND_QUEUE;如果为宿主机上的 OpenCL 实现分配其所需资源失败,则返回 CL_OUT_OF_HOST_MEMORY。

6 OpenCL C 编程语言

本节将描述 OpenCL C 编程语言,它用来创建在 OpenCL 设备上执行的内核。OpenCL C 编程语言(简称 OpenCL C)基于 ISO/IEC 9899:1999 C 语言规范(a.k.a. C99 规范),同时做了一些扩展和限制。关于语法方面的问题,请参考 ISO/IEC 9899:1999 规范。本节仅描述 OpenCL C 对 ISO/IEC 9899:1999 所做的改动和限制。

6.1 支持的数据类型

支持下列数据类型。

6.1.1 内建标量数据类型

表 6.1 列出了所支持的内建标量数据类型。

表 6.1 内建标量数据类型

类型	描述
bool ¹⁸	一种条件数据类型,值为 true 或 false。true 可以为整型常量 1,而 false 可以为整型常量 0。
char	带符号 8 位整型,2 的补值。
unsigned char	无符号 8 位整型。

¹⁸ 将任何标量值变换成 **bool** 值时,只要其值等于 0,结果就是 0,否则结果是 1。

uchar	
short	带符号 16 位整型，2 的补值。
unsigned short ushort	无符号 16 位整型。
int	带符号 32 位整型，2 的补值。
unsigned int uint	无符号 32 位整型。
long	带符号 64 位整型，2 的补值。
unsigned long ulong	无符号 64 位整型。
float	单精度浮点数。必须符合 IEEE 754 单精度存储格式。
half	16 位浮点数。必须符合 IEEE 754-2008 半精度存储格式。
size_t	无符号整型，sizeof 操作符所返回的类型。是 32 位无符号整型（如果 CL_DEVICE_ADDRESS_BITS 是 32 位）或 64 位无符号整型（如果 CL_DEVICE_ADDRESS_BITS 是 64 位）（参见表 4.3）。
ptrdiff_t	带符号整型，两个指针相减的结果。是 32 位带符号整型（如果 CL_DEVICE_ADDRESS_BITS 是 32 位）或 64 位带符号整型（如果 CL_DEVICE_ADDRESS_BITS 是 64 位）（参见表 4.3）。
intptr_t	带符号整型，任何指向 void 的指针都可以转换成此类型，且可以转换回指向 void 的指针，结果与原指针一样。
uintptr_t	无符号整型，任何指向 void 的指针都可以转换成此类型，且可以转换回指向 void 的指针，结果与原指针一样。
void	void 不包含任何值，是一种不完整的数据类型，且不能补全。

大多数内建标量数据类型同时也被声明为 OpenCL API（和头文件）中对应的类型，从而可以由应用来使用。下表描述了 OpenCL C 编程语言的內建标量数据类型与应用可用数据类型的对应关系：

OpenCL 语言中的类型	提供给应用的 API 中的类型
bool	n/a
char	cl_char
unsigned char , uchar	cl_uchar
short	cl_short
unsigned short , ushort	cl_ushort
int	cl_int
unsigned int , uint	cl_uint
long	cl_long
unsigned long , ulong	cl_ulong
float	cl_float
half	cl_half
size_t	n/a
ptrdiff_t	n/a
intptr_t	n/a

uintptr_t	n/a
void	void

6.1.1.1 数据类型half

数据类型 `half` 必须遵循 IEEE 754-2008。`half` 类型的数具有 1 个符号位, 5 个指数位和 10 个尾数位。对于符号、指数、尾数的解释与 IEEE 754 的浮点数相类似。指数偏差 (exponent bias) 是 15。数据类型 `half` 必须可以表示有限标准数、去规格化数、无穷和 NaN。(使用 `vstore_half` 将 `float` 转换成 `half` 和使用 `vload_half` 将 `half` 转换成 `float` 时所生成的) 数据类型为 `half` 的去规格化数不能被刷新成 0。从 `float` 到 `half` 的转换将尾数四舍五入成 11 位精度。从 `half` 到 `float` 的转换时无损的; 所有 `half` 数都可以示为 `float` 值。

数据类型 `half` 只能用来声明含有 `half` 值的缓存指针。下面是几个例子:

```
void
bar (__global half *p)
{
    ....
}
__kernel void
foo (__global half *pg, __local half *pl)
{
    __global half *ptr;
    int offset;
    ptr = pg + offset;
    bar(ptr);
}
```

下面例子是对类型 `half` 的不当应用:

```
half a;
half a[100];

half *p;
a = *p;    ← not allowed. must use vload_half function
```

对于 `half` 指针的加载和存储分别可以使用函数 `vload_half`、`vload_halfn`、`vloada_halfn` 和 `vstore_half`、`vstore_halfn`、`vstorea_halfn` (参见节 6.11.8)。加载函数从内存中读取标量或矢量 `half` 值并将其转换成标量或矢量 `float` 值。存储函数将标量或矢量 `float` 值作为输入, 并将其转换成标量或矢量 `half` 值 (使用恰当的舍入模式), 并将标量或矢量 `half` 值写入内存。

6.1.2 内建矢量数据类型 ¹⁹

支持的矢量数据类型有 char、unsigned char、short、unsigned short、integer、unsigned integer、long、unsigned long 和 float。矢量数据类型是用这些类型名字定义的，即 char、uchar、short、ushort、int、uint、float、long 和 ulong 后面跟一个数字 *n* (*n* 表示元素的数目)。对于 *n*，所支持的值是 2、4、8 和 16。

表 6.2 是内建矢量数据类型的列表。

表 6.2 内建矢量数据类型列表

类型	描述
char <i>n</i>	8 位带符号整型矢量，2 的补值。
uchar <i>n</i>	8 位无符号整型矢量。
short <i>n</i>	16 位带符号整型矢量，2 的补值。
ushort <i>n</i>	16 位无符号整型矢量。
int <i>n</i>	32 位带符号整型矢量，2 的补值。
uint <i>n</i>	32 位无符号整型矢量。
long <i>n</i>	64 位带符号整型矢量，2 的补值。
ulong <i>n</i>	64 位无符号整型矢量。
float <i>n</i>	浮点矢量

内建矢量数据类型同时也被声明为 OpenCL API (和头文件) 中对应的类型，从而可以由应用来使用。下表描述了 OpenCL C 编程语言的 内建矢量数据类型 与应用可用数据类型的对应关系：

OpenCL 语言中的类型	提供给应用的 API 中的类型
char <i>n</i>	cl_char <i>n</i>
uchar <i>n</i>	cl_uchar <i>n</i>
short <i>n</i>	cl_short <i>n</i>
ushort <i>n</i>	cl_ushort <i>n</i>
int <i>n</i>	cl_int <i>n</i>
uint <i>n</i>	cl_uint <i>n</i>
long <i>n</i>	cl_long <i>n</i>
ulong <i>n</i>	cl_ulong <i>n</i>
float <i>n</i>	cl_float <i>n</i>

¹⁹ 即使下面的计算设备不支持某个或全部矢量数据类型，OpenCL 实现也支持这些内建矢量数据类型。设备的编译器会将它们转换成恰当的指令（使用计算设备原生支持的内建类型）。

6.1.3 其它内建数据类型

表 6.3 列出了 OpenCL 支持的其他内建数据类型。

表 6.3 其它内建数据类型

类型	描述
image2d_t	2D 图像。参见节 6.11.8。
image3d_t	3D 图像。参见节 6.11.8。
sampler_t	采样器类型。参见节 6.11.8。
event_t	事件。可以用来标识从全局内存到局部内存的异步拷贝，反之亦然。参见节 6.11.11。

6.1.4 保留 (reserved) 数据类型

表 6.4 所列数据类型的名字是保留的，不能用作用户自定义类型的名字。表 6.2 中定义了矢量数据类型的名字，但是其中的 n 即便不是 2、4、8 和 16，也是保留的。

表 6.4 保留数据类型

类型	描述
bool n	布尔矢量。
double、double n	双精度浮点数和双精度矢量。
half n	16 位浮点矢量。
quad、quad n	128 位浮点数和矢量。
complex half complex half n imaginary half imaginary half n	16 位浮点复数，16 位浮点虚数，16 位浮点复数矢量和虚数矢量。
complex float complex float n imaginary float imaginary float n	单精度浮点复数、单精度浮点虚数，单精度浮点复数矢量和虚数矢量。
complex double complex double n imaginary double imaginary double n	双精度浮点复数、双精度浮点虚数，双精度浮点复数矢量和虚数矢量。
complex quad complex quad n imaginary quad imaginary quad n	128 位浮点复数、128 位浮点虚数，128 位浮点复数矢量和虚数矢量。
float nxm	$n*m$ 的单精度浮点值的矩阵，以行优先存储。
double nxm	$n*m$ 的双精度浮点值的矩阵，以行优先存储。
long double	浮点标量和矢量类型，精度和范围至少是 double，

long double <i>n</i>	最多是 quad。
long long long long <i>n</i>	128 位带符号整型标量和矢量。
unsigned long long ulong long ulong long <i>n</i>	128 位无符号标量和矢量。

C99 的衍生类型 (数组、结构体、联合、函数和指针), 由节 6.1.1、节 6.1.2 和节 6.1.3 所描述的内建数据类型构成, 也同樣在被支持之列。

支持 C99 规格中所定义的类型限定符 `const`、`restrict` 和 `volatile`。这些限定符不能用于类型 `image2d_t` 和 `image3d_t`。指针外的其它类型不能使用限定符 `restrict`。

6.1.5 类型对齐

内存中声明为某种类型的数据项会一直按照此类型的字节数对齐。例如, 一个 `float4` 变量会按 16 字节边界对齐。一个 `char2` 变量会按 2 字节边界对齐。

如果一个内建数据类型的大小不是 2 的幂, 则会按紧接的 2 的幂进行对齐。此规则仅限内建数据类型, 结构体和联合除外。

OpenCL 编译器负责数据项的对齐。如果加载、存储没有对齐, 则其行为时未定义的, 节 6.11.7 中所定义的加载和存储矢量数据的函数例外。这些加载、存储矢量数据的函数在读写矢量数据时所用的地址, 即可以对齐到此矢量数据类型的大小, 也可以对齐到此类型中标量元素的大小。

6.1.6 矢量字面量 (vector literals)

可以使用矢量字面量由一组标量或矢量来创建矢量。矢量字面量的书写形式: 一个带括号的矢量类型, 后面紧跟带括号的一组表达式。矢量字面量即可以在初始化语句中使用, 也可以用作可执行语句中的常量。

可以仅指定一个字面值, 即一个标量值, 或者必须与要创建的矢量类型的大小相匹配。如果仅指定了一个标量字面值, 此值会被复制到矢量类型的所有组件中。

例如:

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uint4 u = (uint4)(1); ← u will be (1, 1, 1, 1).
float4 f = (float4)((float2)(1.0f, 2.0f),
                    (float2)(3.0f, 4.0f));
float4 f = (float4)(1.0f, 2.0f); ← error
```

6.1.7 矢量组件 (component)

如果一个矢量数据类型具有 1 到 4 个组件，则其组件可以这样寻址：

`<vector_data_type>.xyzw`。矢量数据类型 `char2`、`uchar2`、`short2`、`ushort2`、`int2`、`uint2`、`long2`、`ulong2` 和 `float2` 可以访问元素 `.xy`。矢量数据类型 `char4`、`uchar4`、`short4`、`ushort4`、`int4`、`uint4`、`long4`、`ulong4` 和 `float4` 可以访问元素 `.xyzw`。

如果访问越界（超过了此矢量数据类型所声明的组件）则会发生错误，例如：

```
float2 pos;

pos.x = 1.0f; // is legal
pos.z = 1.0f; // is illegal
```

组件选择文法允许通过将其名字附到句点 (.) 后面从而选择多个组件。

```
float4 c, a, b;

c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f; // is a float
c.xy = (float2)(3.0f, 4.0f); // is a float2
```

组件的顺序可以与正常顺序不同，可以搅拌，也可以重复：

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float4 swiz= pos.wzyx; // swiz = (4.0f, 3.0f, 2.0f, 1.0f)
float4 dup = pos.xxyy; // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

组件组符号可以出现在表达式的左手边。要形成左值，搅拌 (swizzle) 必须应用到矢量类型的左值上，不能有重复组件，从而产生一个标量或矢量类型的左值，当然这依赖于所指定组件的数目。

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

pos.xw = (float2)(5.0, 6.0); // pos =(5.0f, 2.0f, 3.0f, 6.0f)
pos.wx = (float2)(7.0f, 8.0f); // pos =(8.0f, 2.0f, 3.0f, 7.0f)
pos.xx = (float2)(3.0f, 4.0f); // illegal - 'x' used twice

// illegal - mismatch between float2 and float4
pos.xy = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
```

也可以使用数值索引来访问矢量数据类型的元素。可以使用的数值索引如下表：

矢量组件	可以使用的数值索引
2-component	0、1
4-component	0、1、2、3

8-component	0、1、2、3、4、5、6、7、8
16-component	0、1、2、3、4、5、6、7、8、9、a、A、 b、B、c、C、d、D、e、E、f、F

数值索引前面必须加上字母 `s` 或 `S`。

在下面例子中：

```
float8 f;
```

`f.s0` 指 `float8` 变量 `f` 的第 1 个元素，而 `f.s7` 指第 8 个元素。

在下面例子中：

```
float16 x;
```

`x.sa` (或 `x.sA`) 指 `float16` 变量 `x` 的第 10 个元素，而 `x.sf` (或 `x.sF`) 指第 16 个元素。

在访问具有 1 到 4 个组件的矢量的元素时，数值索引不能与符号 `.xyzw` 混合使用。

例如

```
float4      f, a;
a = f.x12w;    // illegal use of numeric indices with .xyzw
a.xyzw = f.s0123; // valid
```

矢量数据类型可以使用下标 `.lo` (或 `.odd`) 和 `.hi` (或 `.even`) 来获取小号的矢量类型或者将小号的矢量类型组合成大号的矢量类型。可以使用多级下标 `.lo` (或 `.odd`) 和 `.hi` (或 `.even`) 直到变成标量。

下标 `.lo` 指矢量类型中索引较小的那一半组件，而下标 `.hi` 指索引较大的那一半。

下面例子可以帮助说明这一点：

```
float4 vf;

float2 low = vf.lo; // returns vf.xy
float2 high = vf.hi // returns vf.zw
```

下标 `.odd` 指矢量类型中索引为奇数的元素，而下标 `.even` 指索引为偶数的元素。

下面是一些例子。

```
float8 vf;
float4 left = vf.odd;
```

```

float4    right = vf.even;
float2    high = vf.even.hi;
float2    low = vf.odd.lo;

// interleave L+R stereo stream
float4    left, right;
float8    interleaved;
interleaved.even = left;
interleaved.odd = right;

// deinterleave
left = interleaved.even;
right = interleaved.odd;

// transpose a 4x4 matrix
void transpose( float4 m[4] )
{
    // read matrix into a float16 vector
    float16 x = (float16)( m[0], m[1], m[2], m[3] );
    float16 t;
    //transpose
    t.even = x.lo;
    t.odd = x.hi;
    x.even = t.lo;
    x.odd = t.hi;
    //write back
    m[0] = x.lo.lo; // { m[0][0], m[1][0], m[2][0], m[3][0] }
    m[1] = x.lo.hi; // { m[0][1], m[1][1], m[2][1], m[3][1] }
    m[2] = x.hi.lo; // { m[0][2], m[1][2], m[2][2], m[3][2] }
    m[3] = x.hi.hi; // { m[0][3], m[1][3], m[2][3], m[3][3] }
}

```

6.2 变换（conversion）和类型转换（cast）

6.2.1 隐式变换

支持表 6.1 中所定义的标量内建类型之间的隐式变换。指针类型的隐式变换遵循 C99 规范中所描述的规则。

内建矢量数据类型不支持隐式变换。数组和结构体的成员也不支持隐式变换。例如，`int` 数组不能隐式变换成 `float` 数组。对于隐式变换，不仅仅是对表达式值的重新诠释，同时也是将其值变换成新类型相等的值。例如，整型值 5 变换成浮点值 5.0。

6.2.2 显式转换

对内建标量数据类型的标准类型转换会执行适当的变换。

下面例子中：

```

float f = 1.0f;
int i = (int)f;

```

`f` 存储的是 `0x3F800000`，而 `i` 存储的是 `0x1`，即将 `f` 中的浮点值 `1.0f` 到整型值 `1` 的变换。

矢量类型间的显式转换是不合法的。下面例子会导致一个编译器错误：

```
float4    f;
int4      i = (int4) f;  ← not allowed
```

将标量转换成期望的矢量数据类型就可以完成标量到矢量的变换。类型转换也会执行恰当的算术变换。到内建整型矢量的变换会使用向零舍入（round to zero）模式。而到浮点矢量类型的变换会使用当前的舍入模式。将 `bool` 转换成矢量整型时 如果 `bool` 值是 `true`，会将所有矢量组件置为 `-1`（即所有比特都是 `1`），否则所有组件都被置为 `0`。例如：

```
float f = 1.0f;
float4 va = (float4)f;
// va is a float4 vector with elements (f, f, f, f).

uchar u = 0xFF;
float4 vb = (float4)u;
// vb is a float4 vector with elements((float)u, (float)u,
//                                     (float)u, (float)u).

float f = 2.0f;
int2 vc = (int2)f;
// vc is an int2 vector with elements ((int)f, (int)f).
```

6.2.3 显式变换

可以用下列一组函数

```
convert_<dest type name>(srctype)
```

来执行显式变换。对所支持类型（见节 6.1.1）提供了全套的类型变换，下列类型除外：`bool`、`half`、`size_t`、`ptrdiff_t`、`intptr_t`、`uintptr_t` 和 `void`。

源矢量和宿矢量的元素数目必须一致。

下面例子中

```
uchar4    u;
int4      c = convert_int4(u);
```

`convert_int4` 会执行从 `uchar4` 矢量 `u` 到 `int4` 矢量 `c` 的变换。

```
float      f;
int        i = convert_int(f);
```


`convert_int` 会执行从 `float` 标量 `f` 到 `int` 标量 `i` 的变换。

对于从一种类型到同种类型的变换，无论是表达式的类型，还是表达式的值都不会发生任何变化。

变换的行为可以通过两个可选的修正符 (`modifier`) 来改变，一个可以指定溢出输入的饱和度 (`saturation`)，另一个可以指定舍入模式。

变量变换函数的完整形式如：

```
destType convert_destType<_sat><_roundingMode> (sourceType)
```

矢量变换函数的完整形式如：

```
destTypen convert_destTypen<_sat><_roundingMode> (sourceTypen)
```

6.2.3.1 数据类型

对于标量类型 `char`、`uchar`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`float` 以及来源于它们的内建矢量类型，可以进行变换。对于操作数和结果，必须具有相同数目的元素。操作数和结果的类型可能相同。

6.2.3.2 舍入模式

浮点类型相关的变换要遵循 IEEE 754 的舍入规则。对于源操作数或目标类型是浮点或整型的变换可能带有一个可选的舍入模式修正符。如下表所示：

表 6.5 舍入模式

修正符	舍入模式的描述
<code>_rte</code>	舍入到最近偶数。
<code>_rtz</code>	向零舍入。
<code>_rtp</code>	向正无穷舍入。
<code>_rtn</code>	向负无穷舍入。
没有指定修正符	使用目标类型的默认舍入模式。向整型的变换使用 <code>_rtz</code> ；向浮点类型的变换使用当前舍入模式。

默认情况下，向整型的变换使用舍入模式 `_rtz` (向零舍入)，而向浮点类型的变换²⁰使

²⁰ 对于向浮点格式的变换，当一个有限的源超过了目标类型所能表示的最大有限浮点值时，舍入模式会影响变换的结果，根据 IEEE 754 的舍入规则来确定结果到底是目标类型所能表示的最大有限浮点值，还是和源具有相同正负号的无穷值。

用当前舍入模式。所支持的唯一一个默认浮点舍入模式是 `_rte` (舍入到最近偶数), 即浮点类型的当前舍入模式将是 `_rte`。

6.2.3.3 溢出行为和饱和变换

当变换的操作数大于目标类型所能表示的最大值, 或者小于目标类型所能表示的最小值, 这都叫溢出。对于整型间的变换, 如果发生了溢出, 将截取源操作数元素的低位填充到相应的目标元素中。而对于浮点类型到整型的变换, 则依赖于具体实现。

到整型的变换可以选择使用饱和模式 (在变换函数名字后面加上修正符 `_sat`)。这种模式下如果发生了溢出, 结果将是目标类型所能表示的值中与源操作数最相近的值 (NaN 将变成 0)。

到浮点类型的变换遵循 IEEE 754 的舍入规则。这种变化不能使用修正符 `_sat`。

6.2.3.4 显式变换示例

例 1:

```
short4 s;

// -ve values clamped to 0
ushort4 u = convert_ushort4_sat( s );

// values > CHAR_MAX converted to CHAR_MAX
// values < CHAR_MIN converted to CHAR_MIN
char4 c = convert_char4_sat( s );
```

例 2:

```
float4 f;

// values implementation defined for
// f > INT_MAX, f < INT_MIN or NaN
int4 i = convert_int4( f );

// values > INT_MAX clamp to INT_MAX, values < INT_MIN clamp
// to INT_MIN. NaN should produce 0.
// The _rtz rounding mode is
// used to produce the integer values.
int4 i2 = convert_int4_sat( f );

// similar to convert_int4, except that
// floating-point values are rounded to the nearest
// integer instead of truncated
int4 i3 = convert_int4_rte( f );

// similar to convert_int4_sat, except that
// floating-point values are rounded to the
// nearest integer instead of truncated
```

```
int4 i4 = convert_int4_sat_rte( f );
```

例 3：

```
int4 i;

// convert ints to floats using the current rounding mode.
float4 f = convert_float4( i );

// convert ints to floats. integer values that cannot
// be exactly represented as floats should round up to the
// next representable float.
float4 f = convert_float4_rtp( i );
```

6.2.4 将数据重新诠释为另一种类型

在OpenCL中，经常需要将某数据类型中的比特重新诠释为另一种类型。有时需要直接访问浮点类型中的比特，这是一种非常典型的需求，例如，掩掉浮点类型的符号位，将结果用作矢量相关的运算（参见节 6.3.d）²¹。对于这种（反）变换，C语言中已经有了一些方法对其进行了实践，包括指针别名（pointer aliasing）、联合（union）和memcpy。对于C99，这些方法中只有memcpy是严格正确的。由于OpenCL没有提供memcpy，所有需要一些别的方法。

6.2.4.1 使用联合重新诠释类型

OpenCL 语言对联合做了扩展，允许程序使用另外一种类型的成员来访问一个联合对象的成员。此对象中的相关字节将被视为访问时所用类型的对象。如果访问时所用类型大于对象的实际类型，则额外字节的值是未定义的。

例子：

```
union{ float f; uint u; double d; } u;

u.u = 1;           // u.f contains 2**-149. u.d is undefined --
                   // depending on endianness the low or high half
                   // of d is unknown

u.f = 1.0f;        // u.u contains 0x3f800000, u.d contains an
                   // undefined value -- depending on endianness
```

²¹ 另外，对C语言的其它扩展（被设计用来支持特定的矢量ISA，即AltiVec™、CELL Broadband Engine™架构）使用这些变换与搅拌运算符一起实现类型的反变换。为支持这种遗留代码，as_typed()支持在大小相同但元素个数不同的矢量类型间进行变换，即使这种变换的行为不能移植到不同硬件架构的OpenCL实现上。AltiVec™是Motorola Inc.的商标。Cell Broadband Engine是Sony Computer Entertainment Inc.的商标。

```

// the low or high half of d is unknown
u.d = 1.0;    // u.u contains 0x3ff00000 (big endian) or 0
               // (little endian). u.f contains either 0x1.ep0f
               // (big endian) or 0.0f (little endian)

```

6.2.4.2 使用as_typen()重新诠释类型

使用运算符**as_typen()**可以将表 6.1 和表 6.2 中定义的任何数据类型 (bool、half 和void除外) 重新诠释为另一种相同大小的数据类型²²。如果操作数和返回类型所包含的元素数目相同, 会将操作数中的所有比特直接作为新类型返回, 而不做任何修改。对于函数参数, 通常的类型提升 (type promotion) 不会被执行。

例如, `as_float(0x3f800000)` 返回 `1.0f`, `1.0f` 正式将 `0x3f800000` 看做 IEEE 754 单精度浮点数时的值。

如果操作数和结果类型的元素数目不一样, 其结果依赖于具体实现。对此, 实现必须显式定义其行为, 但两种实现的行为可以不同。实现可以定义结果中含有多少原始比特 (全部、部分或空) 以及其顺序。`as_typen()` 不能对大小不同的类型进行转换。

例如:

```

float f = 1.0f;
uint u = as_uint(f); // Legal. Contains: 0x3f800000

float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
// Legal. Contains: (int4)
// (0x3f800000, 0x40000000, 0x40400000, 0x40800000)
int4 i = as_int4(f);

float4 f, g;
int4 is_less = f < g;

// Legal. f[i] = f[i] < g[i] ? f[i] : 0.0f

```

²² 联合 (union) 用来反映数据在内存中是怎么组织的, 而 `as_typen()` 用来反映数据在寄存器中是怎么组织的。通过在设备上使用共享的寄存器文件 (此文件对操作数和结果两种类型都可以进行操作), `as_typen()` 在编译后不会产生任何指令。注意由于大部分内存中数据组织方式的不同都是由于端模式 (endian) 的不同所引起, 基于寄存器的表示方式也会由于寄存器中元素大小的不同而不同。(例如, 某种架构中, 会将 char 加载到一个 32 位寄存器中, 或将 char 矢量加载到元素大小固定 32 位的 SIMD 矢量寄存器中。) 如果元素数目不匹配, 实现可以根据直观感受选择恰当的数据表示方式。如果元素数目匹配, `as_typen()` 会忠诚的复制重新诠释单个元素时的行为。例如, 如果实现会将单精度数据按双精度存储到寄存器中, 那么 `as_int(float)` 就会先将双精度先行转换成单精度数据, 然后 (如果有必要) 将单精度数据转移到可以操作整形数据的寄存器中。如果数据所在地址空间具有不同的端模式, 则按设备的主要端模式 (dominant endianness) 进行处理。

```
f = as_float4(as_int4(f) & is_less);

int i;
// Legal. Result is implementation-defined.
short2 j = as_short2(i);

int4 i;
// Legal. Result is implementation-defined.
short8 j = as_short8(i);

float4 f;
//Error. result and operand have different size
double4 g = as_double4(f);
```

6.2.5 指针转换

指向旧类型和新类型的指针可以互相转换。将指针转换成新类型时，会假定已经正确对齐了地址，虽然实际上没有检查过。开发人员也需要知道设备的端模式和数据的端模式，以决定标量和矢量数据元素在内存中是如何存储的。

6.3 运算符

a) 算术运算符加 (+)、减 (-)、乘 (*)、除 (/) 可以对内建整型和浮点标量、矢量数据类型进行操作。而取余 (%) 仅能对内建整型标量、矢量数据类型进行操作。在对操作数进行类型变换后，所有算术运算符都会得出与操作数同样基本类型（整型或浮点类型）的结果。变换后，下列情况是有效的：

- ✚ 两个操作数都是标量，则运算后的结果也是标量。
- ✚ 一个操作数是标量，另一个是矢量。这种情况下，标量操作数会晋升和/或向上变换为矢量操作数的类型（标量的向下变换是非法的，且会导致编译时错误）。标量操作数扩展成矢量，且其组件数目和矢量操作数一样。此运算的结果是同样大小的矢量。
- ✚ 两个操作数都是矢量，且大小一样。这种情况下，此运算的结果是同样大小的矢量。

所有其它情况都是非法的。对于整数除法，如果结果发生了溢出（包括上溢和下溢），不会产生异常但结果是一个未指明的值，这跟整数除以零的效果一样。浮点数除以零，其结果将是 IEEE 754 标准中所规定的 $\pm\infty$ 或 NaN。内建函数 **dot** 和 **cross** 可以分别对矢量进行点乘和叉乘。

- b) 算术一元运算符 (+ 或 -) 可以对内建标量和矢量类型进行操作。
- c) 前缀和后缀增量、减量算术运算符 (++ 和 --) 可以对内建标量和矢量类型进行操作 (内

建标量和矢量浮点类型除外)²³。所有一元运算符会作用于其操作数的所有组件上，结果的类型与操作数一致。对于前缀和后缀增量、减量运算符，其操作数表达式必须是可赋值的（即左值）。前缀增量、减量运算符会将所操作表达式的内容加 1 或减 1，整个表达式的值是加 1 或减 1 以后的值。而对于后缀增量、减量运算符，同样会对所操作表达式的内容加 1 或减 1，只不过整个表达式的值是执行加 1 或减 1 之前的值。

d) 关系运算符大于 (>)、小于 (<)、大于等于 (>=)、小于等于 (<=) 可以对标量和矢量类型进行操作。如果源操作数是浮点矢量，则结果是带符号整型矢量。两个操作数的类型必须一致，或者在隐式变换（参见节 6.2.1）后一致。

如果源操作数都是标量，那么变换结果将是带符号整型标量 `int`；如果源操作数都是矢量，则变换结果是带符号整型矢量（与源操作数大小相同）。如果源操作数是矢量类型，则变换结果类型如下表所示：

源操作数类型（矢量）	结果类型
<code>char_n</code> 、 <code>uchar_n</code>	<code>char_n</code>
<code>short_n</code> 、 <code>ushort_n</code>	<code>short_n</code>
<code>int_n</code> 、 <code>uint_n</code> 、 <code>float_n</code>	<code>int_n</code>
<code>long_n</code> 、 <code>ulong_n</code>	<code>long_n</code>

对于标量类型，如果所指关系是 *false* 则返回 0，否则返回 1。对于矢量类型，如果所指关系是 *false* 则返回 0，否则返回 -1（即所有比特置位）。只要任何一个参数不是数（即 NaN），则关系运算符返回 0。

e) 判等运算符等于 (==) 和不等 (!=) 可以对内建标量和矢量类型进行操作。如果操作数类型不匹配，则会对其中一个进行变换（参见节 6.2.1）使两者匹配。对于内建矢量类型，此运算会作用到其所有组件上。

如果源操作数都是标量，那么变换结果将是带符号整型标量 `int`；如果源操作数都是矢量，则变换结果是带符号整型矢量（与源操作数大小相同）。如果源操作数是矢量类型，则变换结果类型如下表所示：

源操作数类型（矢量）	结果类型
<code>char_n</code> 、 <code>uchar_n</code>	<code>char_n</code>
<code>short_n</code> 、 <code>ushort_n</code>	<code>short_n</code>
<code>int_n</code> 、 <code>uint_n</code> 、 <code>float_n</code>	<code>int_n</code>
<code>long_n</code> 、 <code>ulong_n</code>	<code>long_n</code>

²³ 对于浮点类型的前缀、后缀增量操作可能会导致意外的行为，因此不支持对内建标量和矢量浮点类型的这种操作。例如，`0x1.0p25f++` 应该返回 `0x1.0p25f`。同样，如果 `a` 带有小数部分，不保证 `(a++)--` 会返回 `a`。在非默认舍入模式下，对于较大的 `a`，`(a++)--` 的结果可能是 `a++`，也可能是 `a--`。

对于标量类型，如果所指关系是 *false* 则返回 0，否则返回 1。对于矢量类型，如果所指关系是 *false* 则返回 0，否则返回 -1（即所有比特置位）。只要任何一个参数不是数（即 NaN），则关系运算符返回 0。

如果一个参数不是数（即 NaN）或两个参数都不是数，运算符等于（==）返回 0，而运算符不等于（!=）返回 1（对于标量源操作数）或 -1（对于矢量源操作数）。

f) 按位运算符与（&）或（|）异或（^）和非（~）可以对所有内建标量和矢量类型进行操作（内建标量和矢量浮点类型除外）。对于内建矢量类型，此运算符会作用于所有组件上。

g) 逻辑运算符与（&&）或（||）可以对所有内建标量和矢量类型进行操作（内建标量和矢量浮点类型除外）。如果左手边操作数不等于 0，与（&&）仅评估右手边操作数。如果左手边操作数等于 0，或（||）仅评估右手边操作数。对于内建矢量类型，此运算符会作用于所有组件上。

逻辑运算符异或（^^）保留。

如果源操作数都是标量，那么变换结果将是带符号整型标量 *int*；如果源操作数都是矢量，则变换结果是带符号整型矢量（与源操作数大小相同）。如果源操作数是矢量类型，则变换结果类型如下表所示：

源操作数类型（矢量）	结果类型
<i>charn</i> 、 <i>ucharn</i>	<i>charn</i>
<i>shortn</i> 、 <i>ushortn</i>	<i>shortn</i>
<i>intn</i> 、 <i>uintn</i> 、 <i>floatn</i>	<i>intn</i>
<i>longn</i> 、 <i>ulongn</i>	<i>longn</i>

对于标量类型，如果逻辑运算结果是 *false* 则返回 0，否则返回 1。对于矢量类型，如果逻辑运算结果是 *false* 则返回 0，否则返回 -1（即所有比特置位）。

h) 逻辑一元运算符非（!）可以对所有内建标量和矢量类型进行操作（内建标量和矢量浮点类型除外）。对于内建矢量类型，此运算符会作用于所有组件上。

如果源操作数是标量，那么变换结果将是带符号整型标量 *int*；如果源操作数是矢量，则变换结果是带符号整型矢量（与源操作数大小相同）。如果源操作数是矢量类型，则变换结果类型如下表所示：

源操作数类型（矢量）	结果类型
<i>charn</i> 、 <i>ucharn</i>	<i>charn</i>
<i>shortn</i> 、 <i>ushortn</i>	<i>shortn</i>
<i>intn</i> 、 <i>uintn</i> 、 <i>floatn</i>	<i>intn</i>

longn, ulongn	longn
---------------	-------

对于标量类型，如果操作数不等于 0，则一元逻辑运算结果是 0，否则返回 1。对于矢量类型，如果操作数不等于 0 则返回 0，否则返回-1（即所有比特置位）。

i) 三元选择运算符（?:）会在三个表达式上进行操作，形如（*exp1? exp2: exp3*）。此运算符会计算第一个表达式 *exp1*，其结果可能是标量或矢量（浮点类型除外）。对于标量结果，如果是 *true* 则计算第二个表达式，否则计算第三个表达式。对于矢量结果，等同于调用 **select**(*exp2, exp3, exp1*)（参见表 6.13）。第二和第三个表达式可以是任意类型，但其类型必须一致，或者经过变换（参见节 6.2.1）后一致。此类型就是整个表达式的类型。

j) 运算符（~）、右移（>>）、左移（<<）可以对所有内建标量和矢量类型进行操作（内建标量和矢量浮点类型除外）。对于内建矢量类型，此运算符会作用于所有组件上。对于右移（>>）、左移（<<）运算符，如果第一个操作数是标量，则最右操作数也必须是标量，而如果第一个操作数是矢量，最右操作数即可以是矢量也可以是标量。

$E1 \ll E2$ 的结果就是将 *E1* 左移 *x* 位，*x* 等于将 *E2* 视为无符号整型时其低 $\log_2 N$ 位的值，*N* 是表示此标量数据类型或矢量数据类型的单个组件所用比特的数目；腾出的比特会填零。

$E1 \gg E2$ 的结果就是将 *E1* 右移 *x* 位，*x* 等于将 *E2* 视为无符号整型时其低 $\log_2 N$ 位的值，*N* 是表示此标量数据类型或矢量数据类型的单个组件所用比特的数目；如果 *E1* 是带符号类型且是负值，则空出的比特位会被置为全 1，否则空出的比特位会被清零。

k) 运算符 **sizeof** 会产生操作数的字节数，包括对齐所需的所有填充字节（参见节 6.1.5），其操作数可能是一个表达式，也可能是一个括起来的类型名称。大小由操作数得类型所决定。结果是整数。如果操作数是变长数组²⁴，会进行求值，否则不进行求值，结果是整型常量。内建基本类型的 **sizeof** 结果如下表所示：

内建基本类型	sizeof 结果
char、uchar	1
short、ushort、half	2
int、uint、float	4
long、ulong、double	8

如果操作数是矢量，则结果是组件数目*单个标量组件的大小。如果操作数是数组类型，结果是数组的字节总数。如果操作数是结构体或联合，结果是这样一个对象的字节总数，包括内部和尾部的填充字节。如果表达式是函数类型或不完整类型，或者括起来的这种类型名

²⁴ OpenCL 1.0 还不支持变长数组，请参见节 6.8.d。

称，或者指定的位域成员，对于这些都不能应用 `sizeof` 运算符。

l) 运算符逗号 (,) 可以操作表达式，对于以逗号分隔的一系列表达式，将返回最右表达式的类型和值。会按从左至右的顺序对所有表达式求值。

m) 一元运算符 (*) 是一种间接指示。如果操作数指向函数，则结果是函数指示符 (designator)；如果指向一个对象，则结果是一个 (指示此对象的) 左值。如果操作数的类型是“指向 *type* 的指针”，结果的类型就是“*type*”。如果指针的值无效，则 * 运算的行为未定义²⁵。

n) 一元运算符 (&) 返回操作数的地址。如果操作数类型为“*type*”，则结果为“指向 *type* 的指针”。如果操作数是一元运算符 * 的结果，则 * 和 & 都不会被求值，这两个运算符都会被忽略，但是运算符的限制仍然存在，且结果不是一个左值。如果操作数是运算符 [] 的结果，[] 所暗示的 & 或 * 都不会被求值，结果就像是移除了运算符 &，且 [] 变成了运算符 +。否则，结果是一个指针，指向操作数所指示的对象或函数²⁶。

o) 给变量名赋值是通过赋值运算符 (=) 来完成的。如

```
lvalue = expression
```

赋值运算符将 *expression* 的值存储到 *lvalue* 中。*expression* 和 *lvalue* 的类型必须一致，或者 *expression* 的类型是表 6.1 所列类型之一，这种情况下，在赋值完成之前会对表达式进行隐式变换。

如果 *expression* 是标量类型，而 *lvalue* 是矢量类型。标量表达式会被提升和/或向上变换为矢量操作数的类型（标量的向下变换是非法的，且会导致编译时错误）。标量操作数扩展成矢量，且其组件数目和矢量操作数一样。此运算的结果是同样大小的矢量。

其它任何类型变换必须显式指定。左值必须是可写的。下列都是左值：内建类型、所有结构体或数组类型的变量，结构体的域、左值加域选择器 (.) 所选的组件或搅拌（域不能重复）、带括号的左值、用数组下标运算符 ([]) 解引用的左值。其它一元或二元表达式、函数名、域有所重复的搅拌、常量等都不可作为左值。三元运算符 (?:) 也不能作为左值。

没有规定操作数的求值顺序。如果尝试修改赋值运算符的结果，或者在下个序列点

²⁵ 用一元运算符 * 对指针解引用时，其无效值包括下列情况：NULL 指针、地址没有根据对象类型正确对齐、所指对象生命期已经结束。

²⁶ 因此，&*E 等价于 E（即使 E 是 NULL 指针），而 &(E1[E2]) 等价于 ((E1)+(E2))。如果 E 是函数指示符或一个对 & 有效的左值，那么 *&E 就是函数指示符或一个与 E 相等的左值。如果 *p 是左值，且 T 是某种对象指针类型的名字，那么 *(T)p 是左值，且其类型与 T 所指类型相兼容。用一元运算符 * 对指针解引用时，其无效值包括下列情况：NULL 指针、地址没有根据对象类型正确对齐、所指对象生命期已经结束。

(sequence point) 后访问它，其行为未定义。其它赋值运算符有 +=、-=、*=、/=、%=、<=、>=、&=、|=、^=。

表达式

```
lvalue op= expression
```

等价于

```
lvalue = lvalue op expression
```

，且 *lvalue* 和 *expression* 必须满足 *op* 和赋值运算符 (=) 的语义要求。

注意：

除运算符 **sizeof** 外，数据类型 `half` 不能与本节所描述的任意运算符一起使用。

6.4 矢量运算

矢量运算是组件级的。通常，当一个运算符操作矢量时，将以组件级的方式独立操作矢量的每个组件。

例如，

```
float4    v, u;
float      f;

v = u + f;
```

等价于

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
v.w = u.w + f;
```

；而

```
float4    v, u, w;

w = v + u;
```

等价于

```
w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;
w.w = v.w + u.w;
```

；而且对于大多数运算符和所有整型、浮点矢量类型都一样。

6.5 地址空间限定符

OpenCL 实现了下列分离的地址空间：`__global`、`__local`、`__constant` 和 `__private`。声明变量时可以使用地址空间限定符来指定用来分配对象的内存区域。OpenCL 对 C 语言中类型限定符的文法做了扩展，包含一个地址空间名称作为一个有效的类型限定符。如果这个地址空间名称对一个对象的类型做了限定，那么就会在指定的地址空间内分配此对象；否则，就会在通用地址空间中分配此对象。

不带前缀`__`的地址空间名称，即 `global`、`local`、`constant` 和 `private`，可以用来代替相应的带有前缀`__`的地址空间名称。

程序中函数的参数、局部变量所用的通用地址空间是`__private`。`__kernel` 函数的所有参数都位于`__private` 地址空间内。

如果`__kernel` 函数的参数是指针类型，那么只能指向下列地址空间之一：`__global`、`__local`、`__constant`。指向地址空间 A 的指针只能赋给同样指向地址空间 A 的另一个指针。将指向地址空间 A 的指针转换成指向地址空间 B 的指针，这是非法的。

如果`__kernel` 函数的参数声明为 `image2d_t` 或 `image3d_t`，那么一定是指向 `__global` 地址空间。

所有程序范围的变量必须在`__constant` 地址空间中声明。

例子：

```
// declares a pointer p in the __private address space that
// points to an int object in address space __global
__global int *p;

// declares an array of 4 floats in the __private address space.
float x[4];
```

6.5.1 `__global` (或 `global`)

地址空间名称`__global` 或 `global` 用来指分配自全局内存池的内存对象（缓冲对象或图像对象）。

可以将缓冲对象声明为一个指针，指向标量、矢量或用户自定义结构体。

数组内存对象的实际大小在（宿主机代码中通过恰当的 API 调用）分配此对象时确定。

一些例子如：

```
__global float4 *color; // An array of float4 elements
typedef struct {
    float a[3];
    int b[2];
} foo_t;
__global foo_t *my_info; // An array of foo_t elements.
__global image2d_t texture; // A 2D texture image
```

如果一个图像对象附到了一个在声明时带有限定符的参数上，对于 2D 图像对象，那么此参数必须是 `image2d_t`，而对于 3D 图像对象，那么此参数必须是 `image3d_t`。不能直接访问图像对象的元素。提供了一些内建函数来读写图像对象。

限定符 `const` 可以和限定符 `__global` 一起使用来指定一个只读的缓冲对象。

6.5.2 __local (或local)

地址空间名称 `__local` 和 `local` 可以用来描述分配自局部内存的变量，这种变量由工作组中的所有工作项共享。此限定符可以用在声明为指针的函数参数上（包括 `__kernel` 函数），也可以用在 `__kernel` 函数中声明的变量上。

6.5.3 __constant (或constant)

地址空间名称 `__constant` 和 `constant` 可以用来描述分配自全局内存且在内核中只读的变量。在内核执行时，这些只读变量可以被其所有（全局）工作项访问。此限定符可以用在声明为指针的函数参数上（包括 `__kernel` 函数），也可以用在 `__kernel` 函数中声明为指针的局部变量上，全局变量也可以。程序源码中所声明的带有限定符 `__constant` 的全局变量需要初始化。

在 OpenCL 程序源码中，对带有限定符 `__constant` 的变量进行写操作会导致一个编译时错误。

6.5.4 __private (或private)

函数（包括 `__kernel` 函数）中的所有变量及传给此函数的所有参数都在 `__private` 和 `private` 地址空间中。对于没有指定地址空间限定符且声明为指针的变量，如果类型是 `image2d_t` 或 `image3d_t` 则指向 `__global` 地址空间，否则 `__private` 地址空间。

名字 `__global`、`__constant`、`__local`、`__private`、`global`、`constant`、`local` 和 `private` 被保留用作地址空间限定符，否则不能使用它们。

6.6 图像访问限定符

内核参数中的图像对象可以声明为只读或只写的。内核不能对一个图像对象既读又写。可以使用限定符 `__read_only` (或 `read_only`) 和 `__write_only` (或 `write_only`) 来表明图像对象是只读的还是只写的。默认的限定符是 `__read_only`。

下面例子中

```
__kernel void
foo (read_only image2d_t imageA,
     write_only image2d_t imageB)
{
    ....
}
```

`imageA` 是只读的, `imageB` 是只写的。

名字 `__read_only`、`__write_only`、`__read_write`、`read_only`、`write_only` 和 `read_write` 被保留用作地址空间限定符, 否则不能使用它们。

6.7 函数限定符

6.7.1 `__kernel` (或 `kernel`)

限定符 `__kernel` (或 `kernel`) 可以将函数声明为内核, 此内核可以在 OpenCL 设备上由应用来执行。符合下列规则的函数才能使用此限定符:

- ✚ 只可以在此设备上执行。
- ✚ 可以被宿主机调用。
- ✚ 如果 `__kernel` 函数被另一个内核函数调用, 则仅仅是一个普通函数调用。

注意:

如果内核函数中声明了带有限定符 `__local` 或 `local` 的变量, 宿主机可以使用 `clEnqueueNDRangeKernel`、`clEnqueueTask` 等 API 来调用这些内核函数; 而如果由其它内核函数调用它们, 则其行为依赖于具体实现。

名字 `__kernel` 和 `kernel` 被保留用作地址空间限定符, 否则不能使用它们。

6.7.2 可选的特性 (`attribute`) 限定符

如下面所说, 用 `__kernel` 声明内核函数时, 可以加上 `__attribute__` 来声明一些附加信息。

可选的 `__attribute__((vec_type_hint(<typen>)))`²⁷ 可以给编译器一个暗示，表示 `__kernel` 的可计算的 width，同时在编译器试图将代码自动矢量化时，它也是计算可利用的处理器带宽的基础。 `vec_type_hint(<typen>)` 应当是内建标量和矢量数据类型之一（见表 6.1 和表 6.2），而如果没有指定 `vec_type_hint(<typen>)`，默认值是 `int`。

默认类型是 `__attribute__((vec_type_hint(int)))`。

例如，如果开发人员指定了一个 `float4` 的宽度，编译器假定通常使用 4 路浮点矢量进行计算，并且可能将工作项合并或分离到多个线程中以更好的匹配硬件性能。不要求实现一定会自动矢量化代码，但要支持这种暗示。即使没有提供暗示，编译器也可能自动矢量化代码。如果实现将 N 个工作项合并到了一个线程中，那么它要负责正确处理这种情况：任一维度上的全局或局部工作项的数目模 N 不是 0。

例如：

```
// autovectorize assuming float4 as the
// basic computation width
__kernel __attribute__((vec_type_hint(float4)))
void foo( __global float4 *p ) { ....

// autovectorize assuming double as the
// basic computation width
__kernel __attribute__((vec_type_hint(double)))
void foo( __global float4 *p ) { ....

// autovectorize assuming int (default)
// as the basic computation width
__kernel
void foo( __global float4 *p ) { ....
```

如果，例如，声明 `__kernel` 时带有 `__attribute__((vec_type_hint(float4)))`（意味着 `__kernel` 中大多数运算都明显使用 `float4` 矢量化），且内核在运行时使用 Intel® 高级矢量指令（Intel® Advanced Vector Instructions，简称 Intel® AVX，实现了 8 个浮点宽的矢量单元），那么自动矢量化可能选择将两个工作项合并到一个线程中，第二个工作项在 256 位 AVX 寄存器的高 128 位中运行。

另一个例子，Power4 机器有两个标量双精度浮点单元，带有一个深度为 6 个周期的管道。Power4 机器的自动矢量化可能选择将六个 `__attribute__((vec_type_hint(double2))) __kernels` 交错插入一个硬件线程中，来保证一直有 12 路并行性可用来

²⁷ 隐式的自动矢量化会假设内核中调用的所有库都需要在运行时重新编译，这可以处理这种情况：编译器决定合并或分立工作项。这可能意味着，这些库不能是硬编码的二进制，否则必须带有源码或一些可重定向的中间表示。这可能会导致代码安全问题。

喂饱 FPU。也可能根据所涉及的资源利用或一些首选项的二分性，选择将 4 到 8 个（或其它数目的）工作项合并，如果这样选择更好。

可选的 `__attribute__((work_group_size_hint(X, Y, Z)))` 可以给编译器一个暗示，表示可能使用的工作组大小，即最可能由 `clEnqueueNDRangeKernel` 的参数 `local_work_size` 所指定的值。例如 `__attribute__((work_group_size_hint(1, 1, 1)))` 将暗示编译器，执行内核所用的工作组大小最可能是 1。

可选的 `__attribute__((reqd_work_group_size(X, Y, Z)))` 表示工作组大小，必须用作 `clEnqueueNDRangeKernel` 的参数 `local_work_size`。这允许编译器对此内核做适当的优化。对于通过 `clEnqueueTask` 执行的内核，如果指定了 `__attribute__((reqd_work_group_size(X, Y, Z)))`，那么必须是 (1, 1, 1)。

如果 `z` 是 1，那么 `clEnqueueNDRangeKernel` 的参数 `work_dim` 可以是 2 或 3。如果 `y` 和 `z` 是 1，那么 `clEnqueueNDRangeKernel` 的参数 `work_dim` 可以是 1、2 或 3。

6.8 限制

a) 对指针的一点限制：

- ✚ `__kernel` 函数的参数，如果是指针，必须带有限定符 `__global`、`__constant` 或 `__local`。
- ✚ 带有限定符 `__global`、`__constant` 或 `__local` 的指针只能赋值给另一个带有同样限定符的指针。
- ✚ 不允许指针指向函数。
- ✚ `__kernel` 函数的参数不能是指向指针的指针。函数的内部变量或非 `__kernel` 函数的参数可以是指向指针的指针。

b) 对于类型为 `image2d_t` 或 `image3d_t` 的变量只能作为函数的参数。不能直接访问图像的元素。提供有内建函数对图像的任意位置进行读写（参见节 6.11.9）。不允许指针指向 `image2d_t` 或 `image3d_t`。结构体成员不能是 `image2d_t` 或 `image3d_t`。局部变量和函数的返回值不能是 `image2d_t` 或 `image3d_t`。`image2d_t` 或 `image3d_t` 在作为函数参数时不能被修改。

不能声明类型为采样器的数组、指针，函数的局部变量或返回值也不能是采样器。对于 `__kernel` 函数所调用的函数，其参数不能是采样器。采样器作为 `__kernel` 函数的参数时不能被修改。

c) 目前不支持位域。

d) 不支持变长数组和变长结构体（最后一个成员是未指定长度的数组）。

- e) 不支持参数个数不定的宏和函数。
- f) 不能使用 C99 标准头文件 `assert.h`、`ctype.h`、`complex.h`、`errno.h`、`fenv.h`、`float.h`、`inttypes.h`、`limits.h`、`locale.h`、`setjmp.h`、`signal.h`、`stdarg.h`、`stdio.h`、`stdlib.h`、`string.h`、`tgmath.h`、`time.h`、`wchar.h` 和 `wctype.h`
- g) 不支持的存储类别限定符 `extern`、`static`、`auto` 和 `register`。
- h) 不支持预定义标识符。
- i) 不支持递归。
- j) 带有限定符 `__kernel` 的函数只能返回 `void`。
- k) `__kernel` 函数的参数类型不能是内建标量类型 `bool`、`half`、`size_t`、`ptrdiff_t`、`intptr_t` 和 `uintptr_t`。除 `half` 外, 这些类型的大小都依赖于具体实现, 而且在 OpenCL 设备和宿主机处理器上可能是不同的, 这导致很难为其分配缓冲对象并作为参数传递给内核 (参数声明为指向这些类型的指针)。不支持 `half`, 是因为 `half` 仅可用作一种存储格式, 不能将其作为一种数据类型在其上执行浮点算术。
- l) 不规则的控制流是否合法依赖于具体实现。
- m) 少于 32 比特的内建数据类型 (即 `char`、`uchar`、`char2`、`uchar2`、`short`、`ushort`、`half`) 有下列限制。

- ✚ 不支持对下列数据的写操作: 类型为 `char`、`uchar`、`char2`、`uchar2`、`short`、`ushort`、`half` 的指针或数组, 类型为 `char`、`uchar`、`char2`、`uchar2`、`short`、`ushort` 的结构体成员。其他信息请参考节 9.9。

下面的内核实例展示了对于少于 32 比特的内建数据类型, 不支持哪些内存操作:

```
kernel void
do_proc (__global char *pA, short b, __global short *pB)
{
    char          x[100];
    __private char *px = x;
    int           id = (int)get_global_id(0);
    short         f;

    f = pB[id] + b;  ← is allowed
    px[1] = pA[1];  ← error. px cannot be written.
    pB[id] = b;     ← error. pB cannot be written
}
```

- p) 对于 `__kernel` 函数的任意参数, 其类型不能为 `event_t`。
- q) 结构体或联合的所有成员必须属于同一个地址空间。否则是非法的。

6.9 预处理器指令和宏

支持 C99 规范中定义的预处理器指令。

指令 `#pragma` 是这样描述的:


```
#pragma pp-tokensopt new-line
```

在指令**#pragma**中,如果预处理记号**OPENCL**(用来代替**STDC**)没有紧跟在**pragma**后面(比任何宏的位置都靠前),那么实现的行为方式将由其自己决定。但是其行为可能导致翻译失败,或者导致翻译器或最终的程序的行为方式与规范不一致。实现所不认识的任何**pragma**指令,都会被忽略。如果预处理记号**OPENCL**紧跟在**pragma**后面(比任何宏的位置都靠前),那么此指令不会执行任何宏替换操作,此指令为下列形式之一(其含义在其它地方说明):

```
#pragma OPENCL FP_CONTRACT on-off-switch
on-off-switch: one of ON OFF DEFAULT
#pragma OPENCL EXTENSION extensionname : behavior
#pragma OPENCL EXTENSION all : behavior
```

可用的预定义宏如下:

`__FILE__`, 当前源文件的名字(字符串)。

`__LINE__`, 当前源文件中的当前代码行(整型常量)。

`__OPENCL_VERSION__`, OpenCL 设备所支持的 OpenCL 版本号,是一个整数。此文档所描述的 OpenCL 版本中,`__OPENCL_VERSION__`的值为 100。

`__ENDIAN_LITTLE__`, 用来确定 OpenCL 设备是小端架构还是大端架构(1 代表小端,否则未定义)。同时请参考表 4.3 中的 `CL_DEVICE_ENDIAN_LITTLE`。

`__ROUNDING_MODE__`,用来确定当前舍入模式,被置为 `rte`。`__ROUNDING_MODE__`只影响到浮点类型的变换。

`__kernel_exec(X, typen)`(和 `kernel_exec(X, typen)`)被定义为

```
__kernel __attribute__((work_group_size_hint(X, 1, 1))) \
__attribute__((vec_type_hint(typen)))
```

`__IMAGE_SUPPORT__`, 用来确定 OpenCL 设备是否支持图像。如果是 1 则支持,否则未定义。同时可以参考表 4.3 中的 `CL_DEVICE_IMAGE_SUPPORT`。

`__FAST_RELAXED_MATH__`,用来确定 `clBuildProgram` 的构建选项中是否指定了优化选项 `-cl-fast-relaxed-math`。如果是整型常量 1 则表示指定了构建选项 `-cl-fast-relaxed-math`, 否则未定义。

C99 规范中定义了但当前 OpenCL 没有支持的那些宏的名字将被保留,以备将来使用。

6.10 特性限定符

本节将介绍使用 `__attribute__` 的文法，以及绑定了特性限定符的构件。

特性限定符的形式如：`__attribute__ ((attribute-list))`。

特性列表定义为：

```
attribute-list:
    attributeopt
    attribute-list , attributeopt

attribute:
    attribute-token attribute-argument-clauseopt

attribute-token:
    identifier

attribute-argument-clause:
    ( attribute-argument-list )

attribute-argument-list:
    attribute-argument
    attribute-argument-list, attribute-argument

attribute-argument:
    assignment-expression
```

这个文法直接取自 GCC，但与 GCC 有些不同（GCC 只允许对函数、类型和变量应用特性），OpenCL 的特性可以与下列项关联：

- ✚ 类型；
- ✚ 函数；
- ✚ 变量；
- ✚ 块；和
- ✚ 控制流语句。

通常，对于一个给定的上下文，怎样绑定一个特性，其规则有很多需要仔细研究的地方，关于细节可以参考一下 GCC 的文档以及 Maurer 和 Wong 的论文【节 11——参考文献的第 16 项和第 17 项】。

6.10.1 为类型指定特性

关键字 `__attribute__` 允许你定义 `struct` 和 `union` 类型时为其制定特性。此关键字后面紧跟特性规格（放在双层括号内）。目前对于类型而言，有两种特性可用：`aligned` 和 `packed`。

你可以在声明或定义 `enum`、`struct` 或 `union` 时，或者用 `typedef` 声明其它类型时指定类型特性，

对于 `enum`、`struct` 或 `union` 类型，你可以将特性放在 `enum`、`struct` 或 `union` 标签和类型名称中间，或者放在定义的后面（右大括号后面）。最好使用前者。

`aligned (alignment)`

对于指定类型的变量，此特性会指定其最小对齐字节数 `alignment`。例如，下面的声明：

```
struct S { short f[3]; } __attribute__((aligned (8)));  
typedef int more_aligned_int __attribute__((aligned (8)));
```

将强迫编译器尽其所能保证在分配所有类型为 `struct S` 或 `more_aligned_int` 的变量时，都按至少按 8 字节边界进行对齐。

需要注意的是，对于任意 `struct` 或 `union` 类型，ISO C 标准要求其 `alignment` 至少要是其所有成员 `alignment` 的最小公倍数的整数倍，同时必须是 2 的幂。这意味着，你可以通过给 `struct` 或 `union` 的任意一个成员加上特性 `aligned` 来有效的调整 `struct` 或 `union` 类型的 `alignment`，但是如果调整整个 `struct` 或 `union` 类型的 `alignment`，上面例子的表达方式无疑更明白、更直观、可读性更高。

如同前一个例子所示，对于一个 `struct` 或 `union` 类型而言，你可以显式指定 `alignment` 让编译器使用。当然，你也可以不指定，让编译器自己选择一个在目标机器上最常用的值。例如，你可以这样写：

```
struct S { short f[3]; } __attribute__((aligned));
```

如果你没有为特性 `aligned` 指定 `alignment`，那么编译器将会在为目标机器编译时曾经使用的值中选择最大的一个。上面例子中，每个 `short` 大小是 2 字节，因此整个 `struct S` 大小是 6 字节。在 2 的幂中，大于等于 6 的最小的是 8，所以编译器将 `struct S` 的 `alignment` 设置成了 8。

需要注意的是 特性 `aligned` 会受限 OpenCL 设备和编译器本身所固有的一些限制。对于一些设备，OpenCL 编译器可能只允许将变量按最大 `alignment` 进行对齐。如果 OpenCL 编译器只能按 8 字节对齐，即使指定 `aligned(16)` 也只能按 8 字节进行对齐。请查看特定平台的文档以获取进一步信息。

特性 `aligned` 只能增大 `alignment`；但是你可以用特性 `packed` 来减小 `alignment`。看下面。

packed

在定义 struct 或 union 类型时，此特性指定每个成员怎么放置以最小化内存占用。如果定义 enum 时使用了此特性，则表明应当使用最小的基本类型。

对于 struct 和 union 类型使用特性 packed，相当于对于每个成员都是用此特性。

下面例子中，struct my_packed_struct 的成员都压缩在了一起，但是成员 s 的内部布局没有进行压缩。要想 s 的内部布局也进行压缩，必须对 struct my_unpacked_struct 使用特性 packed。

```
struct my_unpacked_struct
{
    char c;
    int i;
};

struct __attribute__((packed)) my_packed_struct
{
    char c;
    int i;
    struct my_unpacked_struct s;
};
```

你可以仅在定义 enum、struct 或 union 时使用此特性，而对于 typedef 则不必使用。

6.10.2 为函数指定特性

当前所支持的函数特性请参考节 6.7.2。

6.10.3 为变量指定特性

关键字 __attribute__ 也可以用在变量或结构体的域上。此关键字后面紧跟双层括号，括号内就是特性规格。目前定义了下列特性限定符：

aligned (alignment)

此特性指定变量或结构体域的最小 alignment（单位：字节）。例如，下面的声明：

```
int x __attribute__((aligned (16))) = 0;
```

会使编译器在 16 字节边界处为全局变量 x 分配内存。对齐的值必须是 2 的幂。

你也可以为结构体的域指定 alignment。例如，要想创建按 8 字节对齐的两个 int，

可以这样写：

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

这样也可以创建一个带有 double 成员的 union，并使其按 8 字节对齐。

如上面例子所示，你可以显式指定 alignment；另外，也可以不显式指定，这样编译器会使用目标机器上最常用的 alignment。例如，你可以这样写：

```
short array[3] __attribute__ ((aligned));
```

如果你没有显式指定 alignment，OpenCL 编译器自动将为目标机器曾用过的最大 alignment 作为所声明的变量或结构体域的 alignment。。

当用在 struct 或其成员上时，特性 aligned 只能增大 alignment；为了减小它，必须使用特性 packed。当作为 typedef 的一部分时，特性 aligned 既可以增大也可以减小 alignment，此时如果使用特性 packed 会产生告警。

需要注意的是 特性 aligned 会受限 OpenCL 设备和编译器本身所固有的一些限制。对于一些设备，OpenCL 编译器可能只允许将变量按最大 alignment 进行对齐。如果 OpenCL 编译器只能按 8 字节对齐，即使指定 aligned(16)也只能按 8 字节进行对齐。请查看特定平台的文档以获取进一步信息。

packed

特性 packed 会使变量或结构体域按最小的 alignment——1 个字节进行对齐，除非你用特性 aligned 设置了更大的值。

这里有一个结构体，域 x 带有特性 packed，因此它紧跟域 a：

```
struct foo
{
    char a;
    int x[2] __attribute__ ((packed));
};
```

如果特性列表位于用户自定义类型之前，则将作用于此类型的变量之上，而不会作用于此类型上；如果特性列表紧跟类型主体，则就会作用于此类型上。例如：

```
/* a has alignment of 128 */
__attribute__((aligned(128))) struct A {int i;} a;

/* b has alignment of 16 */
__attribute__((aligned(16))) struct B {double d;}
__attribute__((aligned(32))) b ;
```

```
struct A a1; /* a1 has alignment of 4 */
struct B b1; /* b1 has alignment of 32 */
```

endian (endiantype)

特性 `endian` 用来确定变量的字节序。`endiantype` 可以置为 `host` ,表示变量使用宿主处理器的端模式 ;也可以指望 `device` ,表示变量将使用执行内核的设备的端模式。默认值是 `device`。

例如 :

```
float4 *p __attribute__((endian(host)));
```

表示 `p` 所指内存中的数据是按宿主机的端模式进行存储的。

6.10.4 为块和控制流语句指定特性

对于基本块和控制流语句 , 其特性需要放在前面 , 例如 :

```
__attribute__((attr1)) {...}
for __attribute__((attr2)) (...) __attribute__((attr3)) {...}
```

这里 `attr1` 作用于大括号中的程序块上 , `attr2` 和 `attr3` 分别作用于循环控制构件和循环体上。

目前还没有为块和控制流语句定义任何特性限定符。

6.10.5 扩展的特性限定符

可以为标准的语言扩展和供应商特定的扩展为特性的文法做一些扩展。任何扩展都必须遵守节 9 的命名约定。

对于编译器而言 , 特性提供了非常有用的暗示。我们认为 , 对于某个特定的 OpenCL 实现而言 , 它可以忽略所有特性 , 但是生成的可执行二进制必须能产生同样的结果。这并不妨碍实现使用特性所提供的附加信息并执行优化或其它转化 , 只要在它自己看来是合适的。这种情况下 , 程序员有责任保证所提供的信息在某种意义上是正确的。

6.11 内建函数

OpenCL C 编程语言提供了一套丰富的函数来对标量和矢量进行操作。这些函数中有很多跟通用的 C 库提供的函数名字相似 , 所不同的是它们支持标量和矢量参数类型。应用应

当尽可能使用这些内建函数，而不要去实现自己的版本。

6.11.1 工作项函数

表 6.6 列出了所有工作项函数，可以用来查询指定给 **clEnqueueNDRangeKernel** 的维度的数目、全局和局部工作大小，以及在设备上执行内核时每个工作项的全局 ID 和局部 ID。当使用 **clEnqueueTask** 执行内核时，维度的数目、全局和局部工作大小都是 1。

表 6.6 工作项函数列表

函数	描述
<code>uint get_work_dim()</code>	返回所用的维度数目，即给 clEnqueueNDRangeKernel 的参数 <i>work_dim</i> 所指定的值。 对于 clEnqueueTask ，返回 1。
<code>size_t get_global_size(uint dimindx)</code>	返回在维度 <i>dimindx</i> 上的全局工作项的数目，即给 clEnqueueNDRangeKernel 的参数 <i>global_work_size</i> 所指定的值。 <i>dimindx</i> 的取值范围为 0 到 get_work_dim()-1 ，否则 get_global_size() 返回 1。 对于 clEnqueueTask ，总是返回 1。
<code>size_t get_global_id(uint dimindx)</code>	返回维度 <i>dimindx</i> 上的唯一全局工作项 ID 的值。此 ID 基于指定来执行内核的全局工作项的数目。 <i>dimindx</i> 的取值范围是从 0 到 get_work_dim()-1 ，否则 get_global_id() 返回 0。
<code>size_t get_local_size(uint dimindx)</code>	返回维度 <i>dimindx</i> 上的唯一局部工作项的数目。如果 clEnqueueNDRangeKernel 的参数 <i>local_work_size</i> 不是 NULL，则就是 <i>local_work_size</i> 的值；否则 OpenCL 实现会选择一个恰当的 <i>local_work_size</i> 并由此函数返回。此 ID 基于指定来执行内核的全局工作项的数目。 <i>dimindx</i> 的取值范围是从 0 到 get_work_dim()-1 ，否则 get_local_size() 返回 0。 对于 clEnqueueTask ，总是返回 1。
<code>size_t get_local_id(uint dimindx)</code>	返回维度 <i>dimindx</i> 上的唯一局部工作项 ID 的值（即某个特定工作组中的一个工作项）。此 ID 基于指定来执行内核的全局工作项的数目。 <i>dimindx</i> 的取值范围是从 0 到 get_work_dim()-1 ，否则 get_local_id() 返回 0。 对于 clEnqueueTask ，返回 0。
<code>size_t get_num_groups(uint dimindx)</code>	返回在维度 <i>dimindx</i> 上执行内核的工作组的数目。 <i>dimindx</i> 的取值范围是从 0 到 get_work_dim()-1 ，否则 get_num_groups() 返回 1。

	对于 clEnqueueTask ，总是返回 1。
size_t get_group_id (uint <i>dimindx</i>)	get_group_id 返回工作组 ID，取值范围为从 0 到 get_num_groups (<i>dimindx</i>)-1。 dimindx 的取值范围是从 0 到 get_work_dim ()-1，否则 get_group_id () 返回 0。 对于 clEnqueueTask ，返回 0。

6.11.2 数学函数

内建的数学函数由表 6.7 列出，可以分为两类：

- 有参数为标量和参数为矢量的两个版本。
- 参数只能是标量浮点数。

矢量版本的数学函数会做组件级的运算。描述是针对每个组件的。

内建数学函数不受调用环境中的舍入模式的影响，总会按舍入到最近偶数运算并返回同样的值。

表 6.7 所列的内建数学函数的参数可以是标量或矢量。我们使用一个通用的类型名称 *gentype* 来表明函数参数的类型可以是 *float*、*float2*、*float4*、*float8* 或 *float16*。对于一个函数而言，无论参数是哪种类型，所有参数和返回值的类型都必须一样，除非明确指定了类型。

表 6.7 参数为标量和矢量的内建数学函数表

gentype acos (gentype <i>x</i>)	反余弦函数 $\cos^{-1} x$
gentype acosh (gentype <i>x</i>)	反双曲余弦 $\cosh^{-1} x$
gentype acospi (gentype <i>x</i>)	计算 $\frac{\cos^{-1} x}{\pi}$
gentype asin (gentype <i>x</i>)	反正弦函数 $\sin^{-1} x$
gentype asinh (gentype <i>x</i>)	反双曲正弦 $\sinh^{-1} x$
gentype asinpi (gentype <i>x</i>)	计算 $\frac{\sin^{-1} x}{\pi}$
gentype atan (gentype <i>y_over_x</i>)	反正切函数 $\tan^{-1} y_over_x$
gentype atan2 (gentype <i>y</i> , gentype <i>x</i>)	<i>y</i> / <i>x</i> 的反正切 $\tan^{-1} \frac{y}{x}$
gentype atanh (gentype <i>x</i>)	双曲反正切 $\tanh^{-1} x$

gentype atanpi (gentype x)	计算 $\frac{\tan^{-1} x}{\pi}$
gentype atan2pi (gentype y , gentype x)	计算 $\frac{\tan^{-1} \frac{y}{x}}{\pi}$
gentype cbrt (gentype x)	计算立方根 $\sqrt[3]{x}$
gentype ceil (gentype x)	向上取整 $\lceil x \rceil$
gentype copysign (gentype x , gentype y)	将 x 的符号变成跟 y 一样。
gentype cos (gentype x)	计算余弦 $\cos x$
gentype cosh (gentype x)	计算双曲余弦 $\cosh x$
gentype cospi (gentype x)	计算 $\cos \pi x$
gentype erfc (gentype x)	余补误差函数 $1 - \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-\theta^2} d\theta$
gentype erf (gentype x)	误差函数，表示正态分布的积分 $\frac{2}{\sqrt{\pi}} \int_0^x e^{-\theta^2} d\theta$ 有 $\operatorname{erf}(\infty) = 1$, $\operatorname{erf}(-x) = -x$
gentype exp (gentype x)	计算 e 的 x 次幂 e^x
gentype exp2 (gentype x)	底数为 2 的指数 2^x
gentype exp10 (gentype x)	底数为 10 的指数 10^x
gentype expm1 (gentype x)	计算 $e^x - 1.0$
gentype fabs (gentype x)	计算浮点数的绝对值 $ x $
gentype fdim (gentype x , gentype y)	$\begin{cases} x - y, & x > 0 \\ +0, & x \leq 0 \end{cases}$
gentype floor (gentype x)	向下取整 $\lfloor x \rfloor$
gentype fma (gentype a , gentype b , gentype c)	浮点乘加函数。不会对中间的乘积进行舍入，乘积具有无限精度，仅对最后的和进行正确的舍入。边界情况遵循 IEEE 754-2008 标准。 $a * b + c$
gentype fmax (gentype x , gentype y) gentype fmax (gentype x , float y)	如果一个参数是 NaN，则返回另一个参数。如果两个参数都是 NaN，则返回 NaN。否则： $fmax(x, y) = \begin{cases} y, & x < y \\ x, & x \geq y \end{cases}$
gentype fmin ²⁸ (gentype x , gentype y)	如果一个参数是 NaN，则返回另一个参数。如果两

²⁸ fmin 和 fmax 的行为遵循 C99 的定义，可能不符合 IEEE 754-2008 中关于 minNum 和 maxNum 就 SNaN (signaling NaN) 的定义。特别是，SNaN 的行为可能会和 QNaN (quiet NaN) 一样。

gentype fmin (gentype <i>x</i> , float <i>y</i>)	个参数都是 NaN , 则返回 NaN。否则： $fmin(x, y) = \begin{cases} x, & x < y \\ y, & x \geq y \end{cases}$
gentype fmod (gentype <i>x</i> , gentype <i>y</i>)	模。返回 $x - y * trunc(\frac{x}{y})$
gentype fract (gentype <i>x</i> , __global gentype * <i>iptr</i>) ²⁹ gentype fract (gentype <i>x</i> , __local gentype * <i>iptr</i>) gentype fract (gentype <i>x</i> , __private gentype * <i>iptr</i>)	返回 $fmin(x - floor(x), 0x1.fffffep - 1f)$ 。 <i>iptr</i> 中将返回 $floor(x)$ 。
gentype frexp (gentype <i>x</i> , __global int <i>n</i> * <i>exp</i>) gentype frexp (gentype <i>x</i> , __local int <i>n</i> * <i>exp</i>) gentype frexp (gentype <i>x</i> , __private int <i>n</i> * <i>exp</i>)	从 <i>x</i> 中分解出尾数和指数。对于每个组件, 所返回的尾数 (记为 <i>m</i>) 要么是 0, 要么属于 $[\frac{1}{2}, 1)$ 。 <i>x</i> 的每个组件都等于 $m * 2^{exp}$
gentype hypot (gentype <i>x</i> , gentype <i>y</i>)	计算 $\sqrt{x^2 + y^2}$ 不会有过分的上溢或下溢。用途如可以计算直角三角形的斜边长。
int <i>n</i> ilogb (gentype <i>x</i>)	返回 <i>x</i> 的指数 (整型)。
gentype ldexp (gentype <i>x</i> , int <i>n</i>) gentype ldexp (gentype <i>x</i> , int <i>n</i>)	返回 $x * 2^n$
gentype lgamma (gentype <i>x</i>) gentype lgamma_r (gentype <i>x</i> , __global int <i>n</i> * <i>signp</i>) gentype lgamma_r (gentype <i>x</i> , __local int <i>n</i> * <i>signp</i>) gentype lgamma_r (gentype <i>x</i> , __private int <i>n</i> * <i>signp</i>)	返回伽马函数绝对值的自然对数。 <i>signp</i> 中会返回伽马函数的符号。 $\ln \Gamma(x) $
gentype log (gentype)	计算自然对数。
gentype log2 (gentype)	计算以 2 为底的对数。
gentype log10 (gentype)	计算以 10 为底的对数。
gentype log1p (gentype <i>x</i>)	计算 $\log_e(1.0 + x)$
gentype logb (gentype <i>x</i>)	计算 <i>x</i> 的指数, 即 $\log_r x $ 的整数部分。
gentype mad (gentype <i>a</i> , gentype <i>b</i> , gentype <i>c</i>)	mad 逼近 $a * b + c$ 。至于中间的乘积 $a * b$ 是否要舍入或者怎样舍入, 还有怎样处理正常情况怎样处理异常情况, 这些都没有定义。当速度比准确性优先时就可以使用 mad ³⁰ 。

²⁹ 此处的运算符 min()是为了避免 fract(-small)返回 1.0。此时会返回小于 1.0 的最大正浮点数。

³⁰ 告诫用户：对于一些情况, 如 mad(*a*, *b*, -*a***b*), mad()的定义非常宽松, 以至于 *a* 和 *b* 为某些特定值

gentype modf (gentype <i>x</i> , __global gentype * <i>iptr</i>) gentype modf (gentype <i>x</i> , __local gentype * <i>iptr</i>) gentype modf (gentype <i>x</i> , __private gentype * <i>iptr</i>)	分解浮点数。 modf 会将参数 <i>x</i> 分解为整数部分和小数部分，这两部分的符号都与参数 <i>x</i> 一样。整数部分存储在 <i>iptr</i> 所指对象中。
gentype nan (uint <i>n nancode</i>)	返回 QNaN (quiet NaN)。 <i>nancode</i> 可能置于返回值的显著位置。
gentype nextafter (gentype <i>x</i> , gentype <i>y</i>)	计算紧跟 <i>x</i> 的下一个可表示的单精度浮点值 (往 <i>y</i> 的方向)。 因此，如果 <i>y</i> 小于 <i>x</i> ， nextafter() 返回小于 <i>x</i> 的可表示的最大浮点数。
gentype pow (gentype <i>x</i> , gentype <i>y</i>)	计算 x^y
gentype pown (gentype <i>x</i> , int <i>n y</i>)	计算 x^y 其中 <i>y</i> 是整数。
gentype powr (gentype <i>x</i> , gentype <i>y</i>)	计算 x^y 其中 $x \geq 0$ 。
gentype remainder (gentype <i>x</i> , gentype <i>y</i>)	返回值记为 <i>r</i> ，则 $r = x - n * y$ 其中 <i>n</i> 是最接近 <i>x/y</i> 的整数。如果最接近 <i>x/y</i> 的整数有两个，那么 <i>n</i> 是为偶数的那个。如果 <i>r</i> 是 0，则其符号同 <i>x</i> 一样。
gentype remquo (gentype <i>x</i> , gentype <i>y</i> , __global int <i>n *quo</i>) gentype remquo (gentype <i>x</i> , gentype <i>y</i> , __local int <i>n *quo</i>) gentype remquo (gentype <i>x</i> , gentype <i>y</i> , __private int <i>n *quo</i>)	返回值记为 <i>r</i> ，则 $r = x - n * y$ 其中 <i>n</i> 是最接近 <i>x/y</i> 的整数。如果最接近 <i>x/y</i> 的整数有两个，那么 <i>n</i> 是为偶数的那个。如果 <i>r</i> 是 0，则其符号同 <i>x</i> 一样。返回值同 remainder 一样。 remquo 同样也会计算 <i>x/y</i> 的整数商的低 7 比特，且其符号同 <i>x/y</i> 一样。此带符号值存储在 <i>quo</i> 所指对象中。
gentype rint (gentype)	舍入成浮点格式的整数值 (使用舍入到最近偶数的舍入模式)。关于舍入模式请参考节 7.1。
gentype rootn (gentype <i>x</i> , int <i>n y</i>)	计算 $x^{1/y}$
gentype round (gentype <i>x</i>)	返回整数值，从 0 往外舍入，无视当前舍入方向。
gentype rsqrt (gentype <i>x</i>)	计算平方根的倒数 $\frac{1}{\sqrt{x}}$
gentype sin (gentype <i>x</i>)	计算正弦 $\sin x$

时，允许 **mad()** 返回任何结果。

gentype sincos (gentype x , __global gentype * $cosval$) gentype sincos (gentype x , __local gentype * $cosval$) gentype sincos (gentype x , __private gentype * $cosval$)	计算 x 的正弦和余弦。返回值为正弦，余弦放在 $cosval$ 中。
gentype sinh (gentype x)	计算双曲正弦 $\sinh x$
gentype sinpi (gentype x)	计算 $\sin(\pi x)$
gentype sqrt (gentype x)	计算平方根 \sqrt{x}
gentype tan (gentype x)	计算正切 $\tan x$
gentype tanh (gentype x)	计算双曲正切 $\tanh x$
gentype tanpi (gentype x)	计算 $\tan(\pi x)$
gentype tgamma (gentype x)	计算伽马函数 $\Gamma(x)$
gentype trunc (gentype x)	舍入到整数值（使用舍入到 0 的舍入模式）。

表 6.8 描述下列函数：

- ✚ 表 6.7 中所列函数的一个子集，在定义时带有前缀 `half_`。这些函数的实现至少具有 10 比特的精度，即 ULP 值 $\leq 8192 \text{ulp}$ （ULP：units in the last place，最后一位的进退位）。
- ✚ 表 6.7 中所列函数的一个子集，在定义时带有前缀 `native_`。这些函数可能映射到一条或多条原生的设备指令，且一般都比表 6.7 所列相应函数（不带前缀 `native_` 的函数）的性能要高一些。这些函数的精度（包括一些情况下输入的范围）依赖于具体实现。
- ✚ 用于下列基本操作：除和倒数的 `half_` 和 `native_` 函数。

表 6.8 参数为标量和矢量的内建 `half_` 和 `native_` 数学函数

函数	描述
gentype half_cos (gentype x)	计算余弦。且 x 必须在范围 $-2^{16} \dots +2^{16}$ 内。
gentype half_divide (gentype x , gentype y)	计算 x/y 。
gentype half_exp (gentype x)	计算 e^x 。
gentype half_exp2 (gentype x)	计算 2^x 。
gentype half_exp10 (gentype x)	计算 10^x 。
gentype half_log (gentype x)	计算自然对数。
gentype half_log2 (gentype x)	计算以 2 为底的对数。
gentype half_log10 (gentype x)	计算以 10 为底的对数。
gentype half_powr (gentype x , gentype y)	计算 x^y ，其中 $x \geq 0$ 。

gentype half_recip (gentype x)	计算倒数。
gentype half_rsqrt (gentype x)	计算平方根的倒数。
gentype half_sin (gentype x)	计算正弦。且 x 必须在范围 $-2^{16} \dots + 2^{16}$ 内。
gentype half_sqrt (gentype x)	计算平方根。
gentype half_tan (gentype x)	计算正切。且 x 必须在范围 $-2^{16} \dots + 2^{16}$ 内。
gentype native_cos (gentype x)	计算余弦。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_divide (gentype x , gentype y)	计算 x/y 。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_exp (gentype x)	计算 e^x 。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_exp2 (gentype x)	计算 2^x 。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_exp10 (gentype x)	计算 10^x 。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_log (gentype x)	计算自然对数。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_log2 (gentype x)	计算以 2 为底的对数。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_log10 (gentype x)	计算以 10 为底的对数。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_powr (gentype x , gentype y)	计算 x^y ，其中 $x \geq 0$ 。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_recip (gentype x)	计算倒数。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_rsqrt (gentype x)	计算平方根的倒数。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_sin (gentype x)	计算正弦。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_sqrt (gentype x)	计算平方根。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。
gentype native_tan (gentype x)	计算正切。参数的范围依赖于具体实现。最大值的错误也依赖于具体实现。

half 函数对于非正规值的支持是可选的。**half** 函数可能返回节 7.5.3 所允许的任意结果，即使 `-cl-denorms-are-zero` (参见节 5.4.3.2) 无效。**native** 函数对于非正规数的支持依赖于具体实现。

有下列符号常量可用。其类型是 `float`，在单精度浮点数的精度内是准确的。

常量名称	描述
MAXFLOAT	最大非无穷单精度浮点数。
HUGE_VALF	正浮点常量表达式。其值为 <code>+infinity</code> 。用作内建数

	学函数所返回的错误值。
INFINITY	浮点类型的常量表达式，表示正的或无符号的无穷。
NAN	浮点类型的常量表达式，表示 QNaN。

6.11.2.1 math.h的浮点宏和编译指示

可以用编译指示 **FP_CONTRACT** 来允许(如果其状态时 on)或不准(其状态是 off) 实现收缩表达式。每个编译指示既可以写在外部声明的外面，也可以写在复合语句中显式声明和语句的前面。当用在外部声明的外面时，从其出现之处开始，直到遇到另一个编译指示 **FP_CONTRACT**，或者直到此翻译单元的结束之处，此编译指示一直起作用。当用在复合语句中时，从其出现之处开始，直到遇到另一个编译指示 **FP_CONTRACT** (即使出现在一个内嵌复合语句中)，或者直到此复合语句的结束之处，此编译指示一直起作用；在复合语句的结束之处，此编译指示的状态恢复成在此复合语句之前的状态。如果在任何其它上下文中使用此编译指示，其行为未定义。

设置 FP_CONTRACT 的编译指示的定义如下：

```
#pragma OPENCL FP_CONTRACT on-off-switch

on-off-switch is one of:
    ON, OFF or DEFAULT.
The DEFAULT value is ON.
```

宏 **FP_FAST_FMAF** 指示函数 **fma** 是否比对于单精度浮点数直接编码要快。如果定义了此宏，此宏将指示函数 **fma** 一般同对于 float 操作数的一次乘法和一次加法一样快，或者更快。

下表宏必须使用指定的值。这些常量表达式适宜用在处理指令 **#if** 中。

```
#define FLT_DIG 6
#define FLT_MANT_DIG 24
#define FLT_MAX_10_EXP +38
#define FLT_MAX_EXP +128
#define FLT_MIN_10_EXP -37
#define FLT_MIN_EXP -125
#define FLT_RADIX 2
#define FLT_MAX 0x1.fffffep127f
#define FLT_MIN 0x1.0p-126f
#define FLT_EPSILON 0x1.0p-23f
```

下表列出了 OpenCL C 编程语言中内建的宏（前面已经给出）和应用可以使用的宏的对应关系。

OpenCL 语言中的宏	应用的宏
FLT_DIG	CL_FLT_DIG
FLT_MANT_DIG	CL_FLT_MANT_DIG

FLT_MAX_10_EXP	CL_FLT_MAX_10_EXP
FLT_MAX_EXP	CL_FLT_MAX_EXP
FLT_MIN_10_EXP	CL_FLT_MIN_10_EXP
FLT_MIN_EXP	CL_FLT_MIN_EXP
FLT_RADIX	CL_FLT_RADIX
FLT_MAX	CL_FLT_MAX
FLT_MIN	CL_FLT_MIN
FLT_EPSILON	CL_FLT_EPSILON

下列两个宏将扩展成整型常量表达式；如果 x 是 0 或 NaN，则 **ilogb(x)** 会分别返回这两个常量。**FP_ILOGB0** 将是 {INT_MIN} 或 -{INT_MAX}。**FP_ILOGBNAN** 将是 {INT_MAX} 或 {INT_MIN}。

6.11.3 整数函数

表 6.9 列出了内建的整数函数，其参数可以标量或矢量。我们使用一个通用的类型名称 **gentype** 来表明函数的参数可以是下列类型：char、char{2|4|8|16}、uchar、uchar{2|4|8|16}、short、short{2|4|8|16}、ushort、ushort{2|4|8|16}、int、int{2|4|8|16}、uint、uint{2|4|8|16}、long、long{2|4|8|16}、ulong 或 ulong{2|4|8|16}。我们用一个通用的类型名称 **ugentype** 指 **gentype** 的无符号版本。例如，如果 **gentype** 是 char4，则 **ugentype** 是 uchar4。

对于一个函数的任何特定用途，所有参数和返回值的实际类型必须一样，除非明确指定了其类型。

表 6.9 参数为标量和矢量整数的内建函数

函数	描述
ugentype abs (gentype x)	返回 $ x $
ugentype abs_diff (gentype x , gentype y)	返回 $ x - y $ 不会有模上溢。
gentype add_sat (gentype x , gentype y)	返回 $ x + y $ ，且使结果饱和。
gentype hadd (gentype x , gentype y)	返回 $(x + y) \gg 1$ ，且中间的和不会模上溢。
gentype rhadd (gentype x , gentype y) ³¹	返回

³¹ 通常矢量运算暂时需要 $n+1$ 比特来计算结果。rhadd 会指令提供一个额外的比特，而无需 upsample

	$(x + y + 1) \gg 1$ ，且中间的和不会模上溢。
gentype clz (gentype x)	返回 x 中主要 0 比特的数目，从最高有效位开始。
gentype mad_hi (gentype a , gentype b , gentype c)	返回 $mul_hi(a, b) + c$
gentype mad_sat (gentype a , gentype b , gentype c)	返回 $a * b + c$ ，且使结果饱和。
gentype max (gentype x , gentype y)	返回结果如下： $\begin{cases} y, & x < y \\ x, & x \geq y \end{cases}$
gentype min (gentype x , gentype y)	返回结果如下： $\begin{cases} y, & y < x \\ x, & y \geq x \end{cases}$
gentype min_hi (gentype x , gentype y)	计算 $x * y$ ，并返回乘积的高一半的值。
gentype rotate (gentype v , gentype i)	对于 v 中的每个元素，按 i 中相应的元素所给定的比特数目进行循环左移（参考节 6.3 中的移位取模规则）。从左边移出的比特再从右边移入。
gentype sub_sat (gentype x , gentype y)	返回 $ x - y $ ，且使结果饱和。
short upsample (char hi , uchar lo) ushort upsample (uchar hi , uchar lo) shortn upsample (charn hi , uchar lo) ushortn upsample (ucharn hi , uchar lo) int upsample (short hi , ushort lo) uint upsample (ushort hi , ushort lo) intn upsample (shortn hi , ushortn lo) uintn upsample (ushortn hi , ushortn lo) long upsample (int hi , uint lo) ulong upsample (uint hi , uint lo) longn upsample (intn hi , uintn lo) ulongn upsample (uintn hi , uintn lo)	元素 $result[i] = ((short)hi[i] \ll 8) lo[i]$ 元素 $result[i] = ((ushort)hi[i] \ll 8) lo[i]$ 元素 $result[i] = ((int)hi[i] \ll 16) lo[i]$ 元素 $result[i] = ((uint)hi[i] \ll 16) lo[i]$ 元素 $result[i] = ((long)hi[i] \ll 32) lo[i]$ 元素 $result[i] = ((ulong)hi[i] \ll 32) lo[i]$

表 6.10 列出了快速整数函数，可以用来优化内核的性能。我们使用一个通用的类型名称 gentype 来表示此函数的参数可以是下列类型：int、int2、int4、int8、int16、uint、uint2、uint4、uint8 或 uint16。

表 6.10 内建的快速整数函数

函数	描述
----	----

和 downsample。这可作为一个深层的性能优势。

gentype mad24 (gentype x , gentype y , gentype z)	两个 24 位整数 x 和 y 的乘积加上 32 位的整数返回值得到 32 位整数 z 。请参考 mul24 的定义，查看 24 位整数乘法是怎样执行的。
gentype mul24 (gentype x , gentype y)	计算两个 24 位整数 x 和 y 的乘积。 x 和 y 都是 32 位整数，但只有低 24 位用来执行乘法。只有当 x 和 y 是带符号整数且在范围 $[-2^{23}, 2^{23} - 1]$ 内，或者 x 和 y 是无符号整数且在范围 $[0, 2^{24} - 1]$ 内时，才能使用 mul24 ；否则，乘积依赖于具体实现。

下列宏必须使用指定的值。这些常量表达式适宜用在处理指令 `#if` 中。

```
#define CHAR_BIT      8
#define CHAR_MAX      SCHAR_MAX
#define CHAR_MIN      SCHAR_MIN
#define INT_MAX        2147483647
#define INT_MIN        (-2147483647 - 1)
#define LONG_MAX        0x7fffffffffffffffL
#define LONG_MIN        (-0x7fffffffffffffffL - 1)
#define SCHAR_MAX      127
#define SCHAR_MIN      (-127 - 1)
#define SHRT_MAX        32767
#define SHRT_MIN        (-32767 - 1)
#define UCHAR_MAX      255
#define USHRT_MAX       65535
#define UINT_MAX        0xffffffff
#define ULONG_MAX       0xffffffffffffffffUL
```

下表列出了 OpenCL C 编程语言中内建的宏（前面已经给出）和应用可以使用的宏的对应关系。

OpenCL 语言中的宏	应用的宏
CHAR_BIT	CL_CHAR_BIT
CHAR_MAX	CL_CHAR_MAX
CHAR_MIN	CL_CHAR_MIN
INT_MAX	CL_INT_MAX
INT_MIN	CL_INT_MIN
LONG_MAX	CL_LONG_MAX
LONG_MIN	CL_LONG_MIN
SCHAR_MAX	CL_SCHAR_MAX
SCHAR_MIN	CL_SCHAR_MIN
SHRT_MAX	CL_SHRT_MAX
SHAR_MIN	CL_SHAR_MIN
UCHAR_MAX	CL_UCHAR_MAX
USHRT_MAX	CL_USHRT_MAX
UINT_MAX	CL_UINT_MAX
ULONG_MAX	CL_ULONG_MAX

6.11.4 公共函数³²

表 6.11 列出了内建的公共函数。这些函数所做的运算都是组件级的，描述部分都是针对单个组件的。我们使用一个通用的类型名称 `gentype` 来指示参数可以是这些类型 `float`、`float2`、`float4`、`float8` 或 `float16`。

这些内建的公共函数的实现所使用的舍入模式是舍入到最近的偶数。

表 6.11 参数可以为标量和适量的内建公共函数

函数	描述
<code>gentype clamp(gentype x,</code> <code> gentype minval,</code> <code> gentype maxval)</code> <code>gentype clamp(gentype x,</code> <code> float minval,</code> <code> float maxval)</code>	返回 $\text{fmin}(\text{fmax}(x, \text{minval}), \text{maxval})$ 如果 $\text{minval} > \text{maxval}$ ，则结果未定义。
<code>gentype degrees(gentype radians)</code>	将 <code>radians</code> 转换成度数，即 $\left(\frac{180}{\pi}\right) * \text{radians}$
<code>gentype max(gentype x, gentype y)</code> <code>gentype max(gentype x, float y)</code>	结果为 $\begin{cases} y, & x < y \\ x, & x \geq y \end{cases}$ 如果 x 和 y 是无穷或 NaN，返回值未定义。
<code>gentype min(gentype x, gentype y)</code> <code>gentype min(gentype x, float y)</code>	结果为 $\begin{cases} y, & y < x \\ x, & y \geq x \end{cases}$ 如果 x 和 y 是无穷或 NaN，返回值未定义。
<code>gentype mix(gentype x,</code> <code> gentype y, gentype a)</code> <code>gentype mix(gentype x,</code> <code> gentype y, float a)</code>	返回 x 和 y 的线性混合，实现为： $x + (y - x) * a$ a 必须在范围 0.0 到 1.0 之内，否则，结果未定义。
<code>gentype radians(gentype degrees)</code>	将 <code>degrees</code> 转换为弧度，即 $\left(\frac{\pi}{180}\right) * \text{degrees}$
<code>gentype step(gentype edge, gentype x)</code> <code>gentype step(float edge, gentype x)</code>	结果为： $\begin{cases} 0.0, & x < \text{edge} \\ 1.0, & x \geq \text{edge} \end{cases}$
<code>gentype smoothstep(gentype edge0,</code> <code> gentype edge1,</code> <code> gentype x)</code> <code>gentype smoothstep(float edge0,</code> <code> float edge1,</code>	如果 $x \leq \text{edge0}$ ，返回 0.0；如果 $x \geq \text{edge1}$ ，返回 1.0；如果 $\text{edge0} < x < \text{edge1}$ ，会在 0 和 1 之间执行平滑 hermite 插值。如果你想要一个临界值函数平滑过渡，就可以使用这个函数。 这等价于：

³² 函数 `mix` 和 `smoothstep` 在实现时可以使用收缩，如同 `mad` 或 `fma` 一样。

gentype x)	<pre>gentype t; t = clamp((x - edge0) / (edge1 - edge0), 0, 1); return t * t * (3 - 2 * t);</pre> <p>如果 $edge0 \geq edge1$，或者 x、$edge0$ 或 $edge1$ 是 NaN，则结果未定义。</p>
gentype sign (gentype x)	<p>结果为：</p> $A = \pi \begin{cases} 1.0, & x > 0 \\ -0.0, & x = -0.0 \\ +0.0, & x = +0.0 \\ -1.0, & x < 0 \\ 0.0, & x = NaN \end{cases}$

6.11.5 几何函数³³

表 6.11 列出了内建的几何函数。这些函数所做的运算都是组件级的，描述部分都是针对单个组件的。我们使用一个通用的类型名称 `gentype` 来指示参数类型可以是：`float`、`float2` 或 `float4`。

这些内建的几何函数的实现所使用的舍入模式是舍入到最近的偶数。

表 6.12 参数可以是标量和矢量的内建几何函数

函数	描述
float4 cross (float4 $p0$, float4 $p1$)	返回 $p0.xyz$ 和 $p1.xyz$ 的叉乘。所返回的 float4 中的组件 w 将会是 0.0。
float dot (gentype $p0$, gentype $p1$)	计算点乘。
float distance (gentype $p0$, gentype $p1$)	返回 $p0$ 和 $p1$ 的距离，同 length ($p0 - p1$) 一样。
float length (gentype p)	返回矢量 p 的长度，即 $\sqrt{p.x^2 + p.y^2 + \dots}$
gentype normalize (gentype p)	返回一个与 p 的方向相同的矢量，但是长度为 1。
float fast_distance (gentype $p0$, gentype $p1$)	返回 <code>fast_length(p0 - p1)</code> 。
float fast_length (gentype p)	返回矢量 p 的长度，这样计算： $half_sqrt(p.x^2 + p.y^2 + \dots)$
gentype fast_normalize (gentype p)	<p>返回一个方向与 p 一样的矢量，但长度是 1。这样计算：</p> $p * half_sqrt(p.x^2 + p.y^2 + \dots)$ <p>其结果与下面具有无穷精度结果的误差将在 8192ulp 内：</p> <pre>if (all(p == 0.0f)) result = p; else</pre>

³³ 几何函数在实现时可以使用收缩，如同 `mad` 或 `fma` 一样。

	<pre>result = p/sqrt(p.x² + p.y² + ...);</pre> <p>但是下列情况例外：</p> <p>a) 如果平方和大于 FLT_MAX ,那么结果中的浮点值未定义。</p> <p>b) 如果平方和小于 FLT_MIN ,那么实现将返回 p。</p> <p>c) 对于处于“非规范化数刷成零”模式下的设备，如果某个操作数元素小于$\text{sqrt}(\text{FLT_MIN})$,则可能在计算之前先被刷成零。</p>
--	--

6.11.6 关系函数

关系运算符和相等运算符 (<、<=、>、>=、!=、==) 可以与内建标量和矢量类型一起使用，所产生的结果是一个标量或矢量带符号整型，参见节 6.3。

表 6.13 中函数的参数可以是内建标量或矢量类型，返回的结果是标量或矢量整型。参数类型 *gentype* 可以是 *char*、*charn*、*uchar*、*ucharn*、*short*、*shortn*、*ushort*、*ushortn*、*int*、*intn*、*uint*、*uintn*、*long*、*longn*、*ulong*、*ulongn*、*float* 和 *floatn*。参数类型 *igentype* 指带符号整型矢量，即 *char*、*charn*、*short*、*shortn*、*int*、*intn*、*long* 和 *longn*。参数类型 *ugentype* 指无符号整型矢量，即 *uchar*、*ucharn*、*ushort*、*ushortn*、*uint*、*uintn*、*ulong* 和 *ulongn*。

对于表 6.13 中的函数 **isequal**、**isnotequal**、**isgreater**、**isgreaterequal**、**isless**、**islessequal**、**islessgreater**、**isfinite**、**isinf**、**isnan**、**isnormal**、**isordered**、**isunordered** 和 **signbit**，如果参数是标量，关系是 *false* 时返回 0，否则返回 1；如果参数是矢量，关系是 *false* 时返回 0，否则返回-1（即所有比特全 1）。

对于关系函数 **isequal**、**isgreater**、**isgreaterequal**、**isless**、**islessequal** 和 **islessgreater**，如果任何一个参数是 NaN，则返回 0。对于 **isnotequal**，如果任何一个参数是 NaN，参数是标量则返回 1，参数是矢量则返回-1。

表 6.13 标量和矢量关系函数

函数	描述
int isequal (float <i>x</i> , float <i>y</i>) intn isequal (floatn <i>x</i> , floatn <i>y</i>)	返回组件级的比较 $x == y$ 。
int isnotequal (float <i>x</i> , float <i>y</i>) intn isnotequal (floatn <i>x</i> , floatn <i>y</i>)	返回组件级的比较 $x \neq y$ 。
int isgreater (float <i>x</i> , float <i>y</i>) intn isgreater (floatn <i>x</i> , floatn <i>y</i>)	返回组件级的比较 $x > y$ 。
int isgreaterequal (float <i>x</i> , float <i>y</i>) intn isgreaterequal (floatn <i>x</i> , floatn <i>y</i>)	返回组件级的比较 $x \geq y$ 。

int isless (float x , float y) intn isless (floatn x , floatn y)	返回组件级的比较 $x < y$ 。
int islessequal (float x , float y) intn islessequal (floatn x , floatn y)	返回组件级的比较 $x \leq y$ 。
int islessgreater (float x , float y) intn islessgreater (floatn x , floatn y)	返回组件级的比较 $(x < y) (x > y)$ 。
int isfinite (float) intn isfinite (floatn)	测试参数是否一个有限值。
int isinf (float) intn isinf (floatn)	测试参数是否一个无穷值 (+ve 或 -ve)。
int isnan (float) intn isnan (floatn)	测试参数是否 NaN。
int isnormal (float) intn isnormal (floatn)	测试参数是否一个正常值。
int isordered (float x , float y) intn isordered (floatn x , floatn y)	测试参数是否有序。参数为 x 和 y ，结果是 $isequal(x, x) \&\& isequal(y, y)$
int isunordered (float x , float y) intn isunordered (floatn x , floatn y)	测试参数是否无序。参数为 x 和 y ，任何一个 NaN 则返回一个非零值，否则返回 0。
int signbit (float) intn signbit (floatn)	测试符号位。对于此函数的标量版本，如果设置了参数的符号位，则返回 1，否则返回 0。而对于矢量版本的每个组件：如果设置了符号位则返回 -1，否则返回 0。
int any (ignetype x)	如果任何一个组件设置了最高有效位，则返回 1，否则返回 0。
int all (ignetype x)	如果设置了所有组件都设置了最高有效位，则返回 1，否则返回 0。
gentype bitselect (gentype a , gentype b , gentype c)	对于结果中的每个比特，如果 c 中相应比特是 0，则返回 a 中的相应比特，否则返回 b 中的相应比特。
gentype select (gentype a , gentype b , ignetype c) gentype select (gentype a , gentype b , ugentype c)	对于矢量类型中的每个组件，如果 c 中相应组件设置了 MSB，则返回 b 中相应组件，否则返回 a 中相应组件。 对于标量类型， $result = c ? b : a$ 。

6.11.7 加载和存储矢量数据的函数

表 6.14 中所列函数允许你通过某个内存指针读写矢量数据。我们用 `gentype` 指代下列内建数据类型：`char`、`uchar`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`float`。用 `gentypen` 指代内建数据类型 `char{2|4|8|16}`、`uchar{2|4|8|16}`、`short{2|4|8|16}`、`ushort{2|4|8|16}`、`int{2|4|8|16}`、`uint{2|4|8|16}`、`long{2|4|8|16}`、`ulong{2|4|8|16}` 或 `float{2|4|8|16}`。`gentypen` 或函数名(即

`vloadn`、`vstoren` 等) 的后缀 n 表示内建数据类型中的元素个数 ($n=2$ 、 4 、 8 或 16)。

表 6.14 用来加载和存储矢量数据的函数

函数	描述
<code>gentypen vloadn (size_t offset,</code> <code> const __global gentype *p)</code> <code>gentypen vloadn (size_t offset,</code> <code> const __local gentype *p)</code> <code>gentypen vloadn (size_t offset,</code> <code> const __constant gentype *p)</code> <code>gentypen vloadn (size_t offset,</code> <code> const __private gentype *p)</code>	<p>返回从位置($p + (offset * n)$)读取的 <code>sizeof(gentype)n</code> 字节的数据。对于所计算的地址 ($p + (offset * n)$)，如果 <code>gentype</code> 是 <code>charn</code>、<code>ucharn</code>，则必须按 8 比特对齐；如果 <code>gentype</code> 是 <code>shortn</code>、<code>ushortn</code>，则必须按 16 比特对齐；如果 <code>gentype</code> 是 <code>intn</code>、<code>uintn</code>，则必须按 32 比特对齐；如果 <code>gentype</code> 是 <code>longn</code>、<code>ulongn</code>，则必须按 64 比特对齐。</p>
<code>void vstoren (gentypen data,</code> <code> size_t offset, __global gentype *p)</code> <code>void vstoren (gentypen data,</code> <code> size_t offset, __local gentype *p)</code> <code>void vstoren (gentypen data,</code> <code> size_t offset, __private gentype *p)</code>	<p>将 <code>data</code> 中 <code>sizeof(gentypen)</code> 字节的数据写入地址 ($p + (offset * n)$) 中。对于所计算的地址 ($p + (offset * n)$)，如果 <code>gentype</code> 是 <code>charn</code>、<code>ucharn</code>，则必须按 8 比特对齐；如果 <code>gentype</code> 是 <code>shortn</code>、<code>ushortn</code>，则必须按 16 比特对齐；如果 <code>gentype</code> 是 <code>intn</code>、<code>uintn</code>，则必须按 32 比特对齐；如果 <code>gentype</code> 是 <code>longn</code>、<code>ulongn</code>，则必须按 64 比特对齐。</p>
<code>float vload_half (size_t offset,</code> <code> const __global half *p)</code> <code>float vload_half (size_t offset,</code> <code> const __local half *p)</code> <code>float vload_half (size_t offset,</code> <code> const __constant half *p)</code> <code>float vload_half (size_t offset,</code> <code> const __private half *p)</code>	<p>返回从位置($p + offset$)读取的 <code>sizeof(float)</code> 字节的数据。 会将读到的 <code>half</code> 值变换成单精度浮点数。所计算的地址($p + offset$)，必须按 16 比特对齐。</p>
<code>floatn vload_halfn (size_t offset,</code> <code> const __global half *p)</code> <code>floatn vload_halfn (size_t offset,</code> <code> const __local half *p)</code> <code>floatn vload_halfn (size_t offset,</code> <code> const __constant half *p)</code> <code>floatn vload_halfn (size_t offset,</code> <code> const __private half *p)</code>	<p>返回从位置($p + (offset * n)$)读取的 <code>sizeof(float)n</code> 字节的数据。 会将读到的 <code>halfn</code> 值变换成单精度浮点数。所计算的地址($p + (offset * n)$)，必须按 16 比特对齐。</p>
<code>void vstore_half (float data,</code> <code> size_t offset, __global half *p)</code> <code>void vstore_half_ret (float data,</code> <code> size_t offset, __global half *p)</code> <code>void vstore_half_rtz (float data,</code> <code> size_t offset, __global half *p)</code> <code>void vstore_half_rtp (float data,</code> <code> size_t offset, __global half *p)</code>	<p>先将 <code>data</code> 中的单精度浮点值变换成 <code>half</code> 值(使用相应的舍入模式)，然后将 <code>half</code> 值写入地址 ($p + offset$)，此地址按 16 比特进行对齐。 vstore_half 使用当前的舍入模式。默认当前舍入模式是舍入到最近的偶数。</p>

<pre> void vstore_half_rtn (float <i>data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstore_half (float <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_half_ret (float <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_half_rtz (float <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_half_rtp (float <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_half_rtn (float <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_half (float <i>data</i>, size_t <i>offset</i>, __private half *<i>p</i>) void vstore_half_ret (float <i>data</i>, size_t <i>offset</i>, __private half *<i>p</i>) void vstore_half_rtz (float <i>data</i>, size_t <i>offset</i>, __private half *<i>p</i>) void vstore_half_rtp (float <i>data</i>, size_t <i>offset</i>, __private half *<i>p</i>) void vstore_half_rtn (float <i>data</i>, size_t <i>offset</i>, __private half *<i>p</i>) </pre>	
<pre> void vstore_halfn (floatn <i>data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstore_halfn_ret (floatn <i>data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstore_halfn_rtz (floatn <i>data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstore_halfn_rtp (floatn <i>data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstore_halfn_rtn (floatn <i>data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstore_halfn (floatn <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_halfn_ret (floatn <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_halfn_rtz (floatn <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_halfn_rtp (floatn <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_halfn_rtn (floatn <i>data</i>, size_t <i>offset</i>, __local half *<i>p</i>) </pre>	<p>先将 <i>data</i> 中的单精度浮点值转换成 half 值(使用相应的舍入模式), 然后将 halfn 值写入地址 $(p + (offset * n))$, 此地址按 16 比特进行对齐。 vstore_halfn 使用当前的舍入模式。默认当前舍入模式是舍入到最近的偶数。</p>

<pre> void vstore_halfn (floatn data, size_t offset, __private half *p) void vstore_halfn_ret (floatn data, size_t offset, __private half *p) void vstore_halfn_rtz (floatn data, size_t offset, __private half *p) void vstore_halfn_rtp (floatn data, size_t offset, __private half *p) void vstore_halfn_rtn (floatn data, size_t offset, __private half *p) </pre>	
<pre> floatn vloada_halfn (size_t offset, const __global half *p) floatn vloada_halfn (size_t offset, const __local half *p) floatn vloada_halfn (size_t offset, const __constant half *p) floatn vloada_halfn (size_t offset, const __private half *p) </pre>	<p>返回从位置($p + (offset * n)$)读取的 <code>sizeof(floatn)</code> 字节的数据。</p> <p>会将读到的 <code>halfn</code> 值转换成单精度浮点数。所计算的地址($p + (offset * n)$)，必须按<code>sizeof(half) * n</code>对齐。</p>
<pre> void vstorea_halfn (floatn data, size_t offset, __global half *p) void vstorea_halfn_ret (floatn data, size_t offset, __global half *p) void vstorea_halfn_rtz (floatn data, size_t offset, __global half *p) void vstorea_halfn_rtp (floatn data, size_t offset, __global half *p) void vstorea_halfn_rtn (floatn data, size_t offset, __global half *p) void vstorea_halfn (floatn data, size_t offset, __local half *p) void vstorea_halfn_ret (floatn data, size_t offset, __local half *p) void vstorea_halfn_rtz (floatn data, size_t offset, __local half *p) void vstorea_halfn_rtp (floatn data, size_t offset, __local half *p) void vstorea_halfn_rtn (floatn data, size_t offset, __local half *p) void vstorea_halfn (floatn data, size_t offset, __private half *p) void vstorea_halfn_ret (floatn data, </pre>	<p>先将 <code>data</code> 中的单精度浮点值转换成 <code>half</code> 值(使用相应的舍入模式)，然后将 <code>halfn</code> 值写入地址($p + (offset * n)$)，此地址按<code>sizeof(half) * n</code>进行对齐。</p> <p>vstore_halfn 使用当前的舍入模式。默认当前舍入模式是舍入到最近的偶数。</p>

<pre> size_t offset, __private half *p) void vstorea_halfn_rtz (floatn data, size_t offset, __private half *p) void vstorea_halfn_rtp (floatn data, size_t offset, __private half *p) void vstorea_halfn_rtn (floatn data, size_t offset, __private half *p) </pre>	
--	--

对于这些函数,如果要读写的地址没有按上表中所说的方式进行对齐,则结果是未定义的。对于表 6.14 中的存储函数,指针参数 *p* 可以指向 `__global`、`__local` 或 `__private` 内存,而对于加载函数,除此之外,还可以指向 `__constant` 内存。

6.11.8 读写图像的函数

本节的内建函数仅能用于由 `clCreateImage{2D|3D}` 所创建的图像对象。可以调用特定函数访问图像对象对其某个特定位置进行读写。




如果图像对象对于内核是只读的,则在声明时必须加上限定符 `__read_only`,对于这样的图像对象调用 `write_image` 将会产生一个编译错误;而对于只写的图像对象,在声明时必须加上限定符 `__write_only`,对于这样的图像对象调用 `read_image` 也会产生一个编译错误。对于一个内核中的图像对象,要么调用 `read_image`,要么调用 `write_image`,不能同时使用两者。

`read_image` 返回一个具有四个组件的浮点、整型或无符号整型的色彩值,此值定义为 *x*、*y*、*z*、*w*,其中 *x* 指红色通道,*y* 指绿色通道,*z* 指蓝色通道,*w* 指 alpha 通道。

6.11.8.1 采样器

读图像的函数会将采样器作为一个参数。可以使用 `clSetKernelArg` 将采样器作为参数传递给内核,程序源码中也可以声明一个类型为 `sampler_t` 的常量。

程序中所声明的采样器变量类型必须是 `sampler_t`。类型 `sampler_t` 是一个 32 位无符号整型常量,按位域的方式解释为下列属性:

-  寻址模式
-  过滤模式
-  规格化坐标

这些属性控制着 `read_image{f|i|ui}` 怎样读取一个 2D 或 3D 图像对象的元素。

程序源码中使用下列文法声明类型为采样器的全局常量:

```
const sampler_t <sampler name> = <value>
```

采样器的域如表 6.15 所示。

表 6.15 采样器描述符

采样器状态	描述
<normalized coords>	指定传入的 x 、 y 和 z 坐标是规格化值还是非规格化值。必须是下列枚举中的一个： CLK_NORMALIZED_COORDS_TRUE 或 CLK_NORMALIZED_COORDS_FALSE。 在 OpenCL 1.0 中，在一个内核中对同一个图像多次调用 read_image(f i ui) 时，采样器参数必须有相同的 <normalized coords>。
<address mode>	指定图像的寻址模式，即怎样处理坐标越界。必须是字面值或下列枚举之一： CLK_ADDRESS_REPEAT——越界坐标会被包裹成一个有效值。此模式只能用于规格化坐标。如果使用的不是规格化坐标，此模式可能产生一些未定义的图像坐标。 CLK_ADDRESS_CLAMP_TO_EDGE——越界坐标会被压缩到边缘上。 CLK_ADDRESS_CLAMP ³⁴ ——越界坐标会返回一个边界色彩。如果图像通道次序是 CL_A、CL_INTENSITY、CL_RA、CL_ARGB、CL_BGRA 或 CL_RGBA，则此色彩是 (0.0f, 0.0f, 0.0f, 0.0f)；而如果图像通道次序是 CL_R, CL_RG、CL_RGB 或 CL_LUMINANCE，则此色彩是 (0.0f, 0.0f, 0.0f, 1.0f)。 CLK_ADDRESS_NONE——由程序员保证坐标的有效性，否则结果是未定义的。
<filter mode>	指定使用的过滤模式。必须是一个字面值或者下列枚举之一： CLK_FILTER_NEAREST 或 CLK_FILTER_LINEAR。 参见节 8.2。

不能声明类型为采样器的数组、指针，函数中的局部变量或函数的返回值都不能是采样器。如果一个函数被 `__kernel` 函数所调用，则其参数不能是采样器。`__kernel` 函数的参数如果是采样器，则此参数不能被修改。

例子：

³⁴ 与寻址模式 GL_ADDRESS_CLAMP_TO_BORDER 相似。

```
const sampler_t samplerA = CLK_NORMALIZED_COORDS_TRUE
                           CLK_ADDRESS_REPEAT
                           CLK_FILTER_NEAREST;
```

samplerA 所指定的采样器使用规格化坐标，repeat 寻址模式和 nearest 过滤模式。

一个内核中所能声明的采样器的最大数目可以使用 **clGetDeviceInfo** 及 **CL_DEVICE_MAX_SAMPLERS** 来查询。

6.11.8.2 内建图像函数

下列内建函数可以用来读写图像。

表 6.16 内建图像读写函数

图像	描述
float4 read_imagef (image2d_t <i>image</i> , sampler_t <i>sampler</i> , int2 <i>coord</i>) float4 read_imagef (image2d_t <i>image</i> , sampler_t <i>sampler</i> , float2 <i>coord</i>)	<p>用坐标(<i>x</i>, <i>y</i>)来查找 <i>image</i> 中的元素。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 CL_UNORM_INT8 或 CL_UNORM_INT16, 则 read_imagef 所返回的浮点值在范围[0.0 ... 1.0]内。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 CL_SNORM_INT8 或 CL_SNORM_INT16, 则 read_imagef 所返回的浮点值在范围[-1.0 ... 1.0]内。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 CL_HALF_FLOAT 或 CL_FLOAT, 则 read_imagef 返回浮点值。</p> <p>调用 read_imagef 时如果使用整数坐标, 则采样器必须使用过滤模式 CLK_FILTER_NEAREST, 规格化坐标设置为 CLK_NORMALIZED_COORDS_FALSE, 寻址模式设置为 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE; 否则, 返回值未定义。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 的值不是上面所说的那些, 则 read_imagef 的返回值未定义。</p>
int4 read_imagei (image2d_t <i>image</i> , sampler_t <i>sampler</i> , int2 <i>coord</i>) int4 read_imagei (image2d_t <i>image</i> , sampler_t <i>sampler</i> ,	<p>用坐标(<i>x</i>, <i>y</i>)来查找 <i>image</i> 中的元素。</p> <p>read_imagei 和 read_imageui 分别返回非规格化带符号整型和无符号整型值。每个通道都会存储在一个 32 位整型中。</p>

<pre> float2 coord) unsigned int4 read_imageui (image2d_t image, sampler_t sampler, int2 coord) unsigned int4 read_imageui (image2d_t image, sampler_t sampler, float2 coord) </pre>	<p>只有创建图像对象时 <i>image_channel_data_type</i> 为 CL_SIGNED_INT8、CL_SIGNED_INT16 或 CL_SIGNED_INT32，才能使用 read_imagei；否则结果未定义。</p> <p>只有创建图像对象时 <i>image_channel_data_type</i> 为 CL_UNSIGNED_INT8、CL_UNSIGNED_INT16 或 CL_UNSIGNED_INT32，才能使用 read_imageui；否则结果未定义。</p> <p>read_image{i ui}仅支持 nearest 过滤模式，所以 <i>sampler</i> 中的过滤模式必须是 CLK_FILTER_NEAREST；否则结果未定义。</p> <p>对于使用整数坐标作为参数的 read_image{i ui}，<i>sampler</i> 参数中的规格化坐标域必须是 CLK_NORMALIZED_COORDS_FALSE，寻址模式域必须是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；否则结果未定义。</p>
<pre> void write_imagef (image2d_t image, int2 coord, float4 color) void write_imagei (image2d_t image, int2 coord, int4 color) void write_imageui (image2d_t image, int2 coord, unsigned int4 color) </pre>	<p>将 <i>color</i> 的值写到 <i>image</i> 中坐标(<i>x</i>,<i>y</i>)处。在写入前会进行必要的数据格式变换。<i>x</i>&<i>y</i>被看做非规格化坐标，且必须分别在范围0 ... <i>image_width</i> - 1和 0 ... <i>image_height</i> - 1内。</p> <p>只有创建图像对象时 <i>image_channel_data_type</i> 为 CL_SNORM_INT8、CL_UNORM_INT8、CL_SNORM_INT16、CL_UNORM_INT16、CL_HALF_FLOAT 或 CL_FLOAT，才能使用 write_imagef；会将个通道数据从浮点类型变换为实际的存储类型。</p> <p>只有创建图像对象时 <i>image_channel_data_type</i> 为 CL_SIGNED_INT8、CL_SIGNED_INT16 或 CL_SIGNED_INT32，才能使用 write_imagei。</p> <p>只有创建图像对象时 <i>image_channel_data_type</i> 为 CL_UNSIGNED_INT8、CL_UNSIGNED_INT16 或 CL_UNSIGNED_INT32，才能使用 write_imageui。</p> <p>对于这三个函数，如果创建 <i>image</i> 时的 <i>image_channel_data_type</i> 不是以上所列的值或者坐标(<i>x</i>,<i>y</i>)不在范围(0 ... <i>image_width</i> - 1, 0 ... <i>image_height</i> - 1)内，且结果未定义。</p>

<pre>float4 read_imagef (image3d_t image, sampler_t sampler, int4 coord) float4 read_imagef (image3d_t image, sampler_t sampler, float4 coord)</pre>	<p>用坐标(<i>coord.x, coord.y, coord.z</i>)来查找 <i>image</i> 中的元素。<i>coord.w</i> 则被忽略。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 CL_UNORM_INT8 或 CL_UNORM_INT16 或预定义压缩格式之一, 则 read_imagef 所返回的浮点值在范围[0.0 ... 1.0]内。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 CL_SNORM_INT8 或 CL_SNORM_INT16, 则 read_imagef 所返回的浮点值在范围[-1.0 ... 1.0]内。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 CL_HALF_FLOAT 或 CL_FLOAT, 则 read_imagef 返回浮点值。</p> <p>调用 read_imagef 时如果使用整数坐标, 则采样器必须使用过滤模式 CLK_FILTER_NEAREST, 规格化坐标设置为 CLK_NORMALIZED_COORDS_FALSE, 寻址模式设置为 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE; 否则, 返回值未定义。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 的值不是上面所说的那些, 则 read_imagef 的返回值未定义。</p>
<pre>int4 read_imagei (image3d_t image, sampler_t sampler, int4 coord) int4 read_imagei (image3d_t image, sampler_t sampler, float4 coord) unsigned int4 read_imageui (image3d_t image, sampler_t sampler, int4 coord) unsigned int4 read_imageui (image3d_t image, sampler_t sampler, float4 coord)</pre>	<p>用坐标(<i>coord.x, coord.y, coord.z</i>)来查找 <i>image</i> 中的元素。<i>coord.w</i> 则被忽略。</p> <p>read_imagei 和 read_imageui 分别返回非规格化带符号整型和无符号整型值。每个通道都会存储在一个 32 位整型中。</p> <p>只有创建图像对象时 <i>image_channel_data_type</i> 为 CL_SIGNED_INT8、CL_SIGNED_INT16 或 CL_SIGNED_INT32, 才能使用 read_imagei; 否则结果未定义。</p> <p>只有创建图像对象时 <i>image_channel_data_type</i> 为 CL_UNSIGNED_INT8、CL_UNSIGNED_INT16 或 CL_UNSIGNED_INT32, 才能使用 read_imageui; 否则结果未定义。</p> <p>read_image(i ui) 仅支持 nearest 过滤模式, 所以 <i>sampler</i> 中的过滤模式必须是 CLK_FILTER_NEAREST; 否则结果未定义。</p>

表 6.16 中 `get_image_channel_data_type` 和 `get_image_channel_order` 所返回的带有前缀 CLK_ 的值与表 5.4 和表 5.5 中带有前缀 CL_ 的图像通道次序和数据类型一一对应。例如，CL_UNORM_INT8 和 CLK_UNORM_INT8 都表示图像的通道数据类型是非规格化无符号 8 位整型。

下表描述了一个图像元素的通道和 float4、int4 或 unsigned int4 矢量数据类型（由 `read_image{f|i|ui}` 返回或提供给 `write_image{f|i|ui}` 的色彩值）之间的映射关系。对于未进行映射的组件，如果是红、绿、蓝通道，则置为 0.0；如果是 alpha 通道，则置为 1.0。

通道次序	存储通道数据的 float4、int4 或 unsigned int4 组件
CL_R	(r, 0.0, 0.0, 1.0)
CL_A	(0.0, 0.0, 0.0, a)
CL_RG	(r, g, 0.0, 1.0)
CL_RA	(r, 0.0, 0.0, a)
CL_RGB	(r, g, b, 1.0)
CL_RGBA, CL_BGRA, CL_ARGB	(r, g, b, a)
CL_INTENSITY	(I, I, I, I)
CL_LUMINANCE	(L, L, L, 1.0)

6.11.9 同步函数

OpenCL C 编程语言实现了下列同步函数。

表 6.17 内建同步函数

函数	描述
void barrier (cl_mem_fence_flags <i>flags</i>)	<p>在一个工作组中，对于所有在处理器上执行内核的工作项而言，任何一个要想在此 barrier 之外继续执行，必须先执行此函数。</p> <p>如果 barrier 位于条件语句中，任何一个工作项进入此条件语句，则所有工作项都必须进入此语句，且执行 barrier。</p> <p>如果 barrier 在循环中，任何一个工作项要想在 barrier 外继续执行，则对于每一次循环，所有工作项都必须执行此 barrier。</p> <p>函数 barrier 会将一个内存屏障（读或写）入队，来保证对局部或全局内存的正确操作顺序。</p> <p>参数 <i>flags</i> 指定内存地址空间，可以是下列字面值的组合：</p> <p>CLK_LOCAL_MEM_FENCE——函数 barrier 将会刷新局部内存所存储的任意变量，或将一个内存屏障入队来保证对于局部内存的正确操作顺序。</p> <p>CLK_GLOBAL_MEM_FENCE——函数 barrier 会将</p>

	一个内存屏障入队，来保证对全局内存的正确操作顺序。当工作项，例如，写入缓冲对象或图像对象，然后想读取更新后的数据，这时就可以用 barrier 。
--	--

6.11.10 显式内存屏障函数

OpenCL C 编程语言实现了下列显式内存屏障函数 ,为工作项的内存操作提供次序保证。

表 6.18 内建显式内存屏障函数

函数	描述
void mem_fence (cl_mem_fence_flags <i>flags</i>)	对工作项的加载和存储进行排序。这意味着执行 mem_fence 之后的任何加载和存储前，会先提交 mem_fence 之前的加载和存储操作。 参数 <i>flags</i> 指定内存地址空间，可以是下列字面值的组合： CLK_LOCAL_MEM_FENCE CLK_GLOBAL_MEM_FENCE
void read_mem_fence (cl_mem_fence_flags <i>flags</i>)	读内存隔层，仅对加载进行排序。 参数 <i>flags</i> 指定内存地址空间，可以是下列字面值的组合： CLK_LOCAL_MEM_FENCE CLK_GLOBAL_MEM_FENCE
void write_mem_fence (cl_mem_fence_flags <i>flags</i>)	写内存隔层，仅对存储进行排序。 参数 <i>flags</i> 指定内存地址空间，可以是下列字面值的组合： CLK_LOCAL_MEM_FENCE CLK_GLOBAL_MEM_FENCE

6.11.11 全局内存和局部内存间的异步拷贝，以及预取

OpenCL C 编程语言实现了下列函数，提供在全局内存和局部内存间的异步拷贝，以及从全局内存的预取。

我们用 *gentype* 来指代参数类型可以是 `char`、`char{2|4|8|16}`、`uchar`、`uchar{2|4|8|16}`、`short`、`short{2|4|8|16}`、`ushort`、`ushort{2|4|8|16}`、`int`、`int{2|4|8|16}`、`uint`、`uint{2|4|8|16}`、`long`、`long{2|4|8|16}`、`ulong`、`ulong{2|4|8|16}`或 `float`、`float{2|4|8|16}`，除非明确指定。

表 6.19 内建异步拷贝和预取函数

函数	描述
<code>event_t async_work_group_copy (</code>	将 <i>num_elements</i> 个 <i>gentype</i> 元素从 <i>src</i> 异步拷贝

<pre> __local gentype *dst, const __global gentype *src, size_t num_elements, event_t event) event_t async_work_group_copy (__global gentype *dst, const __local gentype *src, size_t num_elements, event_t event) </pre>	<p>到 <i>dst</i>。一个工作组中的所有工作项都要执行此异步拷贝，而且要使用相同的参数，否则结果未定义。wait_group_events 可以使用返回的事件对象来等待异步拷贝完成。可以使用参数 <i>event</i> 将 async_work_group_copy 与之前的异步拷贝关联起来，这样多个异步拷贝可以共享一个事件对象，否则应将 <i>event</i> 置为 0。</p> <p>如果参数 <i>event</i> 不是 0，则会返回 <i>event</i>。此函数不会对源数据执行任何隐式的同步，如在执行拷贝前使用 barrier。</p>
<pre> void wait_group_events (int num_events, event_t *event_list) </pre>	<p>等待事件，这些事件用来标识操作 async_work_group_copy 的完成。执行此等待后，将会释放 <i>event_list</i> 中的事件对象。</p> <p>在一个工作组中，所有工作项都要使用相同的参数调用此函数，否则结果未定义。</p>
<pre> void prefetch (const __global gentype *p, size_t num_elements) </pre>	<p>将 $\text{num_elements} * \text{sizeof}(\text{gentype})$ 字节预取到全局 cache 中。预取指令会应用到工作组中的一个工作项上，同时不会影响内核的功能。</p>





7 OpenCL 数值一致性

本节描述 C99 和 IEEE 754 标准的一些特性，所有遵循 OpenCL 的设备都必须支持这些特性。

本节所描述的内容是关于单精度浮点数的。当前，仅单精度浮点数是必须的，而双精度浮点数仅是一个可选的扩展。

7.1 舍入模式

执行浮点计算时，内部可能会有额外的精度，然后会进行舍入来适应目标类型。IEEE 754 定义了四种可能的舍入模式。

-  舍入到最近偶数
-  向 $+\infty$ 舍入
-  向 $-\infty$ 舍入
-  向 0 舍入

目前，OpenCL 规格仅要求第一项舍入到最近偶数，同时也是默认的舍入模式。另外，仅支持静态选择舍入模式。IEEE 754 规范中所描述的动态重新配置舍入模式不是必需的。

7.2 INF、NaN和去规格化数

INF 和 NaN 是必须支持的。带符号 NaN 不是必需的。

对单精度去规格化数的支持是可选的。对于单精度浮点运算加、减、乘、除,和节 6.11.2 (数学函数)、节 6.11.4 (公共函数)、节 6.11.5 (几何函数) 中的函数,作为输入或输出的单精度浮点去规格化数可能会被刷新成零。

7.3 浮点异常

OpenCL 中禁用了浮点异常。浮点异常的结果必须与 IEEE 754 规范中禁用了异常的情况一致。是否以及何时设置浮点标志或引发异常依赖于具体实现。此标准没有提供任何方法来查询、清除或设置浮点标志或陷入异常。由于陷入机制的不履行、不可移植,以及在矢量上下文中(尤其是异构硬件上)提供精确的异常是不切实际的,所以不鼓励使用这些特性。

然而实现可以通过对标准的扩展来支持这些操作,不过初始化时应当清空所有异常标志并且设置异常掩码,这样算术运算所引发的异常不会触发陷入。如果实现重用了下面的工作,但是实现并不负责进入内核前重新清除标志或将异常掩码重置成默认值。那就是说对于不检查标志或不使能陷入的内核,准许它期望算术不会触发陷入。而对于检查标志或使能陷入的内核,由其负责在将控制器返还给实现之前,清除标志和禁用所有陷入。是否或何时重用下面的工作项(和全局浮点状态,如果有的话)依赖于具体实现。

表达式 `math_errorhandling` 和 `MATH_ERREXCEPT` 被保留,由此标准使用,但是还未定义。如果实现对此规范做了扩展从而支持浮点异常,那它就要对每个 ISO / IEC 9899 : TC2 定义 `math_errorhandling` 和 `MATH_ERREXCEPT`。

7.4 相对误差即ULP

本节我们讨论定义为 ulp 的最大相对误差。加、减、乘、除、乘加以及整数和浮点格式间的变换都遵循 IEEE 754, 因此都可以正确的进行舍入。浮点格式间的变换和节 6.2.3 中的显式变换必须进行正确的舍入。

ULP 是这样定义的:

如果 x 是位于两个连续的有限浮点数 a 和 b 之间的实数, 不等于其中任何一个, 那么 $ulp(x) = |b - a|$, 否则 $ulp(x)$ 是这两个(离 x 最近的)不相等的有限浮点数的距离。此外, $ulp(NaN)$ 是 NaN。

来源: 此定义获得了 Jean-Michel Muller 的赞同, 并对 0 的行为做了一点澄清。请参考 <ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5504.pdf>。

表 7.1³⁵列出了单精度浮点算术运算的最小精确度，以ULP的形式给出。计算ULP值时所参考的是进行算数运算时具有无穷精度的结果。

表 7.1 内建数学函数的 ULP 值

函数	最小精确度——ULP值 ³⁶
$x + y$	正确舍入
$x - y$	正确舍入
$x * y$	正确舍入
$1.0/x$	$\leq 2.5ulp$
x/y	$\leq 2.5ulp$
$acos$	$\leq 4ulp$
$acospi$	$\leq 5ulp$
$asin$	$\leq 4ulp$
$asinpi$	$\leq 5ulp$
$atan$	$\leq 5ulp$
$atan2$	$\leq 6ulp$
$atanpi$	$\leq 5ulp$
$atan2pi$	$\leq 6ulp$
$acosh$	$\leq 4ulp$
$asinh$	$\leq 4ulp$
$atanh$	$\leq 5ulp$
$cbrt$	$\leq 2ulp$
$ceil$	正确舍入
$copysign$	$0ulp$
cos	$\leq 4ulp$
$cosh$	$\leq 4ulp$
$cospi$	$\leq 4ulp$
$erfc$	$\leq 16ulp$
erf	$\leq 16ulp$
exp	$\leq 3ulp$
$exp2$	$\leq 3ulp$
$exp10$	$\leq 3ulp$
$expm1$	$\leq 3ulp$
$fabs$	$0ulp$
$fdim$	正确舍入
$floor$	正确舍入
fma	正确舍入
$fmax$	$0ulp$
$fmin$	$0ulp$
$fmod$	$0ulp$
$fract$	$\leq 1ulp$
$frexp$	$0ulp$
$hypot$	$\leq 4ulp$
$ilogb$	$0ulp$
$ldexp$	正确舍入
log	$\leq 3ulp$

³⁵ 目前，内建数学函数 $lgamma$ 和 $lgamma_r$ 的 ULP 值还未定义。

³⁶ 对于不需要舍入的函数，所用的 ULP 值是 0。

<i>log2</i>	$\leq 3ulp$
<i>log10</i>	$\leq 3ulp$
<i>log1p</i>	$\leq 2ulp$
<i>logb</i>	$0ulp$
<i>mad</i>	所允许的任何值 (无穷 ulp)
<i>modf</i>	$0ulp$
<i>nan</i>	$0ulp$
<i>nextafter</i>	$0ulp$
<i>pow(x,y)</i>	$\leq 16ulp$
<i>pown(x,y)</i>	$\leq 16ulp$
<i>powr(x,y)</i>	$\leq 16ulp$
<i>remainder</i>	$0ulp$
<i>remquo</i>	$0ulp$
<i>rint</i>	正确舍入
<i>rootn</i>	$\leq 16ulp$
<i>round</i>	正确舍入
<i>rsqrt</i>	$\leq 2ulp$
<i>sin</i>	$\leq 4ulp$
<i>sincos</i>	对于正弦值和余弦值： $\leq 4ulp$
<i>sinh</i>	$\leq 4ulp$
<i>sinpi</i>	$\leq 4ulp$
<i>sqrt</i>	$\leq 3ulp$
<i>tan</i>	$\leq 5ulp$
<i>tanh</i>	$\leq 5ulp$
<i>tanpi</i>	$\leq 6ulp$
<i>tgamma</i>	$\leq 16ulp$
<i>trunc</i>	正确舍入
<i>half_cos</i> <i>half_divide</i> <i>half_exp</i> <i>half_exp2</i> <i>half_exp10</i> <i>half_log</i> <i>half_log2</i> <i>half_log10</i> <i>half_powr</i> <i>half_recip</i> <i>half_rsqrt</i> <i>half_sin</i> <i>half_sqrt</i> <i>half_tan</i>	$\leq 8192ulp$
<i>native_cos</i> <i>native_divide</i> <i>native_exp</i> <i>native_exp2</i> <i>native_exp10</i> <i>native_log</i> <i>native_log2</i> <i>native_log10</i> <i>native_powr</i> <i>native_recip</i> <i>native_rsqrt</i> <i>native_sin</i> <i>native_sqrt</i> <i>native_tan</i>	依赖于具体实现

7.5 边界情况的行为

对于节 6.11.2 的数学函数,边界情况的行为符合 ISO/IEC 9899:TC 2(通常称作 C99, TC2)的节 F.9 和 G.6,节 7.5.1 中所列之处例外。

7.5.1 C99 TC2 之外的附加需求

对于有多个 NaN 操作数且返回 NaN 的函数,应当返回其中一个 NaN 操作数。返回 NaN 操作数的函数可能会压制 NaN,如果是 sNaN。如果不是 sNaN,则应当会被转换成一个非 sNaN。sNaN 应当那个被转换成 NaN,但也许会被转换成非 sNaN。至于 NaN 的其它净荷比特或符号位怎么转换,这些未定义。

对于函数 `half_<funcname>` 和不带前缀 `half_` 的同名函数,其行为一致。它们必须满足相同的边界条件需求(参见 C99, TC2 的节 F.9 和 G.6)。对于其它情况,除了所提到的地方,这些单精度函数的误差最大可以到 8192ulp,尽管鼓励更好的精确度。

如果 C99, TC2 的节 F.9 或后面的节 7.5.1 和 7.5.3(和描述其它浮点精度的类似章节)指定了结果(如 `ceil(-1 < x < 0)` 返回 -0),则对于舍入误差(节 7.4)或刷新行为(节 7.5.3)而言,通常的容差无效。这些应当恰好产生指定的结果,不会是其它的。如果使用了 \pm ,则会保留正负号。例如, $\sin(\pm 0) = \pm 0$ 意味着: $\sin(+0)$ 是 +0,而 $\sin(-0)$ 是 -0。

acospi (1) = +0. acospi (x) returns a NaN for $ x > 1$.
asinpi (± 0) = ± 0 . asinpi (x) returns a NaN for $ x > 1$.
atanpi (± 0) = ± 0 . atanpi ($\pm \infty$) = ± 0.5 .
atan2pi (± 0 , -0) = ± 1 . atan2pi (± 0 , +0) = ± 0 . atan2pi (± 0 , x) returns ± 1 for $x < 0$. atan2pi (± 0 , x) returns ± 0 for $x > 0$. atan2pi (y, ± 0) returns -0.5 for $y < 0$. atan2pi (y, ± 0) returns 0.5 for $y > 0$. atan2pi ($\pm y$, - ∞) returns ± 1 for finite $y > 0$. atan2pi ($\pm y$, + ∞) returns ± 0 for finite $y > 0$. atan2pi ($\pm \infty$, x) returns ± 0.5 for finite x. atan2pi ($\pm \infty$, - ∞) returns ± 0.75 . atan2pi ($\pm \infty$, + ∞) returns ± 0.25 .
ceil (-1 < x < 0) returns -0.
cospi (± 0) returns 1 cospi (n + 0.5) is +0 for any integer n where n + 0.5 is representable. cospi ($\pm \infty$) returns a NaN.

<p>exp10 (± 0) returns 1.</p> <p>exp10 ($-\infty$) returns +0.</p> <p>exp10 ($+\infty$) returns $+\infty$.</p>
<p>distance (x, y) calculates the distance from x to y without overflow or extraordinary precision loss due to underflow.</p>
<p>fdim (any, NaN) returns NaN.</p> <p>fdim (NaN, any) returns NaN.</p>
<p>fmod (± 0, NaN) returns NaN.</p>
<p>frexp ($\pm\infty$, exp) returns $\pm\infty$ and stores 0 in exp.</p> <p>frexp (NaN, exp) returns the NaN and stores 0 in exp.</p>
<p>fract (x, $iptr$) shall not return a value greater than or equal to 1.0, and shall not return a value less than 0.</p> <p>fract (+0, $iptr$) returns +0 and +0 in $iptr$.</p> <p>fract (-0, $iptr$) returns -0 and -0 in $iptr$.</p> <p>fract ($+\infty$, $iptr$) returns +0 and $+\infty$ in $iptr$.</p> <p>fract ($-\infty$, $iptr$) returns -0 and $-\infty$ in $iptr$.</p> <p>fract (NaN, $iptr$) returns the NaN and NaN in $iptr$.</p>
<p>length calculates the length of a vector without overflow or extraordinary precision loss due to underflow.</p>
<p>nextafter (-0, $y > 0$) returns smallest positive denormal value.</p> <p>nextafter (+0, $y < 0$) returns smallest negative denormal value.</p>
<p>normalize shall reduce the vector to unit length, pointing in the same direction without overflow or extraordinary precision loss due to underflow.</p> <p>normalize (v) returns v if all elements of v are zero.</p> <p>normalize (v) returns a vector full of NaNs if any element is a NaN.</p>
<p>normalize (v) for which any element in v is infinite shall proceed as if the elements in v were replaced as follows:</p> <pre>for (i = 0; i < sizeof(v) / sizeof(v[0]); ++i) v[i] = isinf(v[i]) ? copysign(1.0, v[i]) : 0.0 * v[i];</pre>
<p>pow (± 0, $-\infty$) returns $+\infty$</p>
<p>pown (x, 0) is 1 for any x, even zero, NaN or infinity.</p> <p>pown (± 0, n) is $\pm\infty$ for odd $n < 0$.</p> <p>pown (± 0, n) is $+\infty$ for even $n < 0$.</p> <p>pown (± 0, n) is +0 for even $n > 0$.</p> <p>pown (± 0, n) is ± 0 for odd $n > 0$.</p>
<p>powr (x, ± 0) is 1 for finite $x > 0$.</p> <p>powr (± 0, y) is $+\infty$ for finite $y < 0$.</p> <p>powr (± 0, $-\infty$) is $+\infty$.</p> <p>powr (± 0, y) is +0 for $y > 0$.</p> <p>powr (+1, y) is 1 for finite y.</p> <p>powr (x, y) returns NaN for $x < 0$.</p> <p>powr (± 0, ± 0) returns NaN.</p> <p>powr ($+\infty$, ± 0) returns NaN.</p> <p>powr (+1, $\pm\infty$) returns NaN.</p> <p>powr (x, NaN) returns the NaN for $x \geq 0$.</p>

powr (NaN, y) returns the NaN.
rint ($-0.5 \leq x < 0$) returns -0.
remquo (x , y , & <i>quo</i>) returns a NaN and 0 in <i>quo</i> if x is $\pm\infty$, or if y is 0 and the other argument is non-NaN or if either argument is a NaN.
rootn (± 0 , n) is $\pm\infty$ for odd $n < 0$. rootn (± 0 , n) is $+\infty$ for even $n < 0$. rootn (± 0 , n) is $+0$ for even $n > 0$. rootn (± 0 , n) is ± 0 for odd $n > 0$. rootn (x , n) returns a NaN for $x < 0$ and n is even. rootn (x , 0) returns a NaN.
round ($-0.5 < x < 0$) returns -0.
sinpi (± 0) returns ± 0 . sinpi ($+n$) returns $+0$ for positive integers n . sinpi ($-n$) returns -0 for negative integers n . sinpi ($\pm\infty$) returns a NaN.
tanpi (± 0) returns ± 0 . tanpi ($\pm\infty$) returns a NaN. tanpi (n) is copysign (0.0, n) for even integers n . tanpi (n) is copysign (0.0, $-n$) for odd integers n . tanpi ($n + 0.5$) for even integer n is $+\infty$ where $n + 0.5$ is representable. tanpi ($n + 0.5$) for odd integer n is $-\infty$ where $n + 0.5$ is representable.
trunc ($-1 < x < 0$) returns -0.

7.5.2 对C99 , TC2 的行为作出的改变

modf 的行为应当跟下面的实现一样：

```
gentype modf ( gentype value, gentype *iptr )
{
    *iptr = trunc( value );
    return copysign( isinf( value ) ? 0.0 : value - *iptr, value );
}
```

rint 始终使用舍入模式：舍入到最近偶数，无视调用者的舍入模式。

7.5.3 在flush-to-zero模式中，边界情况的行为

如果将去规格化数刷新成了零，那么函数可能返回下列四种结果之一：

1. 任何符合 non-flush-to-zero 模式的结果。
2. 如果 1 所给出的结果在进行舍入前是次规格化 (sub-normal) 数，那么它可能会被刷新称零。
3. 如果一个或多个次规格化操作数被刷新成了零，则结果是未刷新的。

4. 如果 3 的结果在舍入前是次规格化数，那么结果可能被刷新成零。

以上任一情况中，如果一个操作数或结果被刷新为零，则零的正负未定义。

如果次规格化数被刷新为零，设备可能会选择下列规则处理 **nextafter** 的边界情况，而不是用节 7.5.1 中所列的那些。

```
nextafter ( +smallest normal, y < +smallest normal ) = +0.
nextafter ( -smallest normal, y > -smallest normal ) = -0.
nextafter ( -0, y > 0 ) returns smallest positive normal value.
nextafter ( +0, y < 0 ) returns smallest negative normal value.
```

为了更清楚一些，次规格化数和去规格化数定义为可表示数的一个字节，范围是 $0 < x < \text{TYPE_MIN}$ 和 $-\text{TYPE_MIN} < x < -0$ ，不包括 ± 0 。对于一个非零数，如果规格化后，其以 2 为底的幂小于 $(\text{TYPE_MIN_EXP} - 1)$ ，则就说它在舍入前是次规格化数。³⁷

8 图像寻址和过滤

用 w_t 、 h_t 和 d_t 分别代表图像的宽度、高度和深度，单位像素。用 `coord.xy` (也可以使用 `(s, t)`) 或 `coord.xyz` (也可以使用 `(s, t, r)`) 表示为 **read_image{f|i|ui}** 所指定的坐标。**read_image{f|i|ui}** 中所指定的采样器用来决定怎样对图像采样并返回恰当的色彩。

8.1 规范化坐标

这会影响怎样解释图像坐标。如果为 **read_image{f|i|ui}** 指定的坐标是规范化坐标，那么会将坐标 s 、 t 和 r 分别乘以 w_t 、 h_t 和 d_t 来生成非规范化坐标。

用 (u, v, w) 表示非规范化坐标浮点值。

8.2 寻址模式和过滤

我们先来看看如果寻址模式不是 `CLK_ADDRESS_REPEAT`，怎样用寻址和过滤模式生成恰当的采样位置以从图像中读取数据。

生成图像坐标 (u, v, w) 后，我们会用恰当的寻址和过滤模式从图像的恰当位置读取数据。

³⁷ 这里应当用相应浮点类型常量来代替 `TYPE_MIN` 和 `TYPE_MIN_EXP`，如 `FLT_MIN` 和 `FLT_MIN_EXP`。

如果 (u, v, w) 中的值是 INF 或 NaN，那么 `read_image{f|i|ui}` 的行为未定义。

Filter Mode = CLK_FILTER_NEAREST

如果过滤模式是 CLK_FILTER_NEAREST，将得到图像中离 (u, v, w) （曼哈顿距离）最近的元素。如果所返回元素的位置为 (i, j, k) ，则：

```
i = address_mode((int)floor(u))
j = address_mode((int)floor(v))
k = address_mode((int)floor(w))
```

对于三维图像，所返回的元素位置是 (i, j, k) ；对于二维图像，则为 (i, j) 。

关于函数 `address_mode` 如表 8.1 所示。

表 8.1 用寻址模式生成纹理位置

寻址模式	<code>address_mode(coord)</code> 的结果
CLK_ADDRESS_CLAMP_TO_EDGE	<code>clamp(coord, 0, size - 1)</code>
CLK_ADDRESS_CLAMP	<code>clamp(coord, -1, size)</code>
CLK_ADDRESS_NONE	<code>coord</code>

对于 u, v, w ，表 8.1 中的 `size` 分别是 w_t, h_t, d_t 。而表 8.1 中的函数 `clamp` 定义如下：

```
clamp(a, b, c) = return (a < b) ? b : ((a > c) ? c : a)
```

如果所选的纹理位置 (i, j, k) 越界，则用边界颜色作为纹理颜色。

Filter Mode = CLK_FILTER_LINEAR

如果过滤模式是 CLK_FILTER_LINEAR，对于 2D 图像会选中其中的 2×2 方阵，对于 3D 材质会选中 $2 \times 2 \times 2$ 的立方体。所返回的 2×2 方阵或 $2 \times 2 \times 2$ 立方体如下：

```
i0 = address_mode((int)floor(u - 0.5))
j0 = address_mode((int)floor(v - 0.5))
k0 = address_mode((int)floor(w - 0.5))
i1 = address_mode((int)floor(u - 0.5) + 1)
j1 = address_mode((int)floor(v - 0.5) + 1)
k1 = address_mode((int)floor(w - 0.5) + 1)
a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)
```

其中 `frac(x)` 指 x 的小数部分，即 $x - \text{floor}(x)$ 。

对于三维图像，返回值是：

```

T = (1 - a) * (1 - b) * (1 - c) * Ti0j0k0
  + a * (1 - b) * (1 - c) * Ti1j0k0
  + (1 - a) * b * (1 - c) * Ti0j1k0
  + a * b * (1 - c) * Ti1j1k0
  + (1 - a) * (1 - b) * c * Ti0j0k1
  + a * (1 - b) * c * Ti1j0k1
  + (1 - a) * b * c * Ti0j1k1
  + a * b * c * Ti1j1k1

```

其中 T_{ijk} 是三维图像中位置为 (i, j, k) 的元素。

对于二维图像，返回值是：

```

T = (1 - a) * (1 - b) * Ti0j0
  + a * (1 - b) * Ti1j0
  + (1 - a) * b * Ti0j1
  + a * b * Ti1j1

```

其中 T_{ij} 是二维图像中位置为 (i, j) 的元素。

如果上面的任何 T_{ijk} 或 T_{ij} 越界，则使用边界颜色作为 T_{ijk} 或 T_{ij} 。

现在我们来讨论一下，如果寻址模式是 CLK_ADDRESS_REPEAT，怎样用寻址和过滤模式生成恰当的采样位置以从图像中读取数据。

如果 (s, t, r) 中的值是 INF 或 NaN，那么 `read_image{f|i|ui}` 的行为未定义。

Filter Mode = CLK_FILTER_NEAREST

如果过滤模式是 CLK_FILTER_NEAREST，所返回图像元素的位置为 (i, j, k) ，则 i 、 j 和 k 为：

```

u = (s - floor(s)) * wt
i = (int)floor(u)
if (i > wt - 1)
    i = i - wt

v = (t - floor(t)) * ht
j = (int)floor(v)
if (j > ht - 1)
    j = j - ht

w = (r - floor(r)) * dt
k = (int)floor(w)
if (k > dt - 1)
    k = k - dt

```

对于三维图像，所返回的元素位置是 (i, j, k) ；对于二维图像，则为 (i, j) 。

Filter Mode = CLK_FILTER_LINEAR

如果过滤模式是 CLK_FILTER_LINEAR ,对于 2D 图像会选中其中的 2×2 方阵 ,对于 3D 材质会选中 $2 \times 2 \times 2$ 的立方体。所返回的 2×2 方阵或 $2 \times 2 \times 2$ 立方体如下：

```

u = (s - floor(s)) * wt
i0 = (int)floor(u - 0.5)
i1 = i0 + 1
if (i0 < 0)
    i0 = wt + i0
if (i1 > wt - 1)
    i1 = i1 - wt

v = (t - floor(t)) * ht
j0 = (int)floor(v - 0.5)
j1 = j0 + 1
if (j0 < 0)
    j0 = ht + j0
if (j1 > ht - 1)
    j1 = j1 - ht

w = (r - floor(r)) * dt
k0 = (int)floor(w - 0.5)
k1 = k0 + 1
if (k0 < 0)
    k0 = dt + k0
if (k1 > dt - 1)
    k1 = k1 - dt

a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)

```

其中 $\text{frac}(x)$ 指 x 的小数部分，即 $x - \text{floor}(x)$ 。

对于三维图像，返回值是：

```

T = (1 - a) * (1 - b) * (1 - c) * Ti0j0k0
+ a * (1 - b) * (1 - c) * Ti1j0k0
+ (1 - a) * b * (1 - c) * Ti0j1k0
+ a * b * (1 - c) * Ti1j1k0
+ (1 - a) * (1 - b) * c * Ti0j0k1
+ a * (1 - b) * c * Ti1j0k1
+ (1 - a) * b * c * Ti0j1k1
+ a * b * c * Ti1j1k1

```

其中 T_{ijk} 是三维图像中位置为 (i, j, k) 的元素。

对于二维图像，返回值是：

```

T = (1 - a) * (1 - b) * Ti0j0
+ a * (1 - b) * Ti1j0
+ (1 - a) * b * Ti0j1
+ a * b * Ti1j1

```

其中 T_{ij} 是二维图像中位置为 (i, j) 的元素。

注意：

如果采样器使用非规范化坐标(浮点或整型坐标),过滤模式为 CLK_FILTER_NEAREST, 且寻址模式是下列模式之一——CLK_ADDRESS_NONE、CLK_ADDRESS_CLAMP_TO_EDGE 或 CLK_ADDRESS_CLAMP, 那么节 8.2 中的元素位置 (i, j, k) 在计算时不会有精度损失。

如果采样器采用其它组合(规范化坐标或非规范化坐标、过滤模式、寻址模式), 则对于寻址模式的计算和图像过滤的运算而言, 在此修订版本的 OpenCL 规范中没有定义其相对误差或精度。为了使任何 OpenCL 设备在进行图像寻址和过滤的计算时都至少具有一个最小精度, 对于采样器的这些组合, 开发者应当在内核中将图像坐标去规范化, 并在内核中实现线性过滤(调用 `read_imagef(f[i][j][i])` 时采样器使用非规范化坐标, 过滤模式使用 CLK_FILTER_NEAREST, 寻址模式使用 CLK_ADDRESS_NONE、CLK_ADDRESS_CLAMP_TO_EDGE 或 CLK_ADDRESS_CLAMP, 并对读取的色彩值进行插值从而生成最终的色彩值)。

8.3 变换规则

这一节我们讨论在内核中读写图像时所应用的变换规则。

8.3.1 对于规范化整型通道数据类型的变换规则

本节我们讨论规范化整型通道数据类型和浮点值间的互相变换。

8.3.1.1 规范化整型通道数据类型到浮点值得变换

对于用图像通道数据类型 CL_UNORM_INT8 和 CL_UNORM_INT16 创建的图像, `read_imagef` 会将通道值从 8 位或 16 位无符号整型变换成范围为 $[0.0f \dots 1.0]$ 的浮点值。

而对于用图像通道数据类型 CL_SNORM_INT8 和 CL_SNORM_INT16 创建的图像, `read_imagef` 会将通道值从 8 位或 16 位带符号整型变换成范围为 $[-1.0 \dots 1.0]$ 的浮点值。

按下列方式执行变换：

```
CL_UNORM_INT8 (8-bit unsigned integer) → float
    normalized float value = (float)c / 255.0f
CL_UNORM_INT16 (16-bit unsigned integer) → float
```

```

normalized float value = (float)c / 65535.0f

CL_SNORM_INT8 (8-bit signed integer) → float

normalized float value = max(-1.0f, (float)c / 127.0f)

CL_SNORM_INT16 (16-bit signed integer) → float

normalized float value = max(-1.0f, (float)c / 32767.0f)

```

对于上面所列变换，其精度 $\leq 1.5\text{ulp}$ ，不过下列情况除外：

```

For CL_UNORM_INT8

0 must convert to 0.0f and
255 must convert to 1.0f

For CL_UNORM_INT16

0 must convert to 0.0f and
65535 must convert to 1.0f

For CL_SNORM_INT8

-128 and -127 must convert to -1.0f,
0 must convert to 0.0f and
127 must convert to 1.0f

For CL_SNORM_INT16

-32768 and -32767 must convert to -1.0f,
0 must convert to 0.0f and
32767 must convert to 1.0f

```

8.3.1.2 从浮点值到规范化整型通道数据类型的变换

对于用图像通道数据类型 CL_UNORM_INT8 和 CL_UNORM_INT16 创建的图像，**write_imagef** 会将浮点色彩值变换成 8 位或 16 位无符号整型。

而对于用图像通道数据类型 CL_SNORM_INT8 和 CL_SNORM_INT16 创建的图像，**write_imagef** 会将浮点色彩值变换成 8 位或 16 位带符号整型。

按下列方式执行变换：

```

float → CL_UNORM_INT8 (8-bit unsigned integer)
convert_uchar_sat_rte(f * 255.0f)
float → CL_UNORM_INT16 (16-bit unsigned integer)
convert_ushort_sat_rte(f * 65535.0f)
float → CL_SNORM_INT8 (8-bit signed integer)
convert_char_sat_rte(f * 127.0f)
float → CL_SNORM_INT16 (16-bit signed integer)
convert_short_sat_rte(f * 32767.0f)

```

对于越界行为和饱和变换规则，请参考节 6.2.3.3。

OpenCL 实现可能选择... 来逼近上面所列变换所用的舍入模式。如果使用的舍入模式不是舍入到最近偶数 (`_rte`)，则实际结果与舍入到最近偶数所产生结果的相对误差必须 ≤ 0.6 。

```
float → CL_UNORM_INT8 (8-bit unsigned integer)

    convert_uchar_sat_<impl-rounding-mode>(f * 255.0f)

    Let f_preferred = (f * 255.0f)
    Let f_approx = impl-rounding-mode(f * 255.0f)

    fabs(f_preferred - f_approx) must be <= 0.6

float → CL_UNORM_INT16 (16-bit unsigned integer)

    convert_ushort_sat_<impl-rounding-mode>(f * 65535.0f)

    Let f_preferred = (f * 65535.0f)
    Let f_approx = impl-rounding-mode(f * 65535.0f)

    fabs(f_preferred - f_approx) must be <= 0.6

float → CL_SNORM_INT8 (8-bit signed integer)

    convert_char_sat_<impl_rounding_mode>(f * 127.0f)

    Let f_preferred = (f * 127.0f)
    Let f_approx = impl-rounding-mode(f * 127.0f)

    fabs(f_preferred - f_approx) must be <= 0.6

float → CL_SNORM_INT16 (16-bit signed integer)

    convert_short_sat_<impl-rounding-mode>(f * 32767.0f)

    Let f_preferred = (f * 32767.0f)
    Let f_approx = impl-rounding-mode(f * 32767.0f)

    fabs(f_preferred - f_approx) must be <= 0.6
```

8.3.2 half浮点通道数据类型的变换

对于用通道数据类型 `CL_HALF_FLOAT` 创建的图像，从 `half` 到 `float` 的变换是无损的（如节 6.1.1.1 所示）。从 `float` 到 `half` 的变换会使用舍入到最近偶数或向零舍入模式对尾数进行舍入。对于从 `float` 到 `half` 的变换，所生成的类型为 `half` 的去规格化数可能会被刷新称零。必须将类型为 `float` 的 NaN 进行恰当的变换，成为类型为 `half` 的 NaN；对于 INF 也是一样。

8.3.3 浮点通道数据类型的变换

对于使用通道数据类型 `CL_FLOAT` 创建的图像，使用下列规则对其读写。

- ✚ 可能会将 NaN 变换成设备所支持的 NaN 值。
- ✚ 可以将去规格化数刷新成零。
- ✚ 所有其它值都必须保留。

8.3.4 带符号和无符号 8 位、16 位和 32 位整型通道数据类型的变换

使用通道数据类型 `CL_SIGNED_INT8`、`CL_SIGNED_INT16` 和 `CL_SIGNED_INT32` 调用 `read_imagei` 会返回图像中指定位置的未经修改的整型值。

使用通道数据类型 `CL_UNSIGNED_INT8`、`CL_UNSIGNED_INT16` 和 `CL_UNSIGNED_INT32` 调用 `read_imageui` 会返回图像中指定位置的未经修改的整型值。

调用 `write_imagei` 会执行下列变换之一：

```
32 bit signed integer → 8-bit signed integer
    convert_char_sat(i)
32 bit signed integer → 16-bit signed integer
    convert_short_sat(i)
32 bit signed integer → 32-bit signed integer
    no conversion is performed
```

调用 `write_imageui` 会执行下列变换之一：

```
32 bit unsigned integer → 8-bit unsigned integer
    convert_uchar_sat(i)
32 bit unsigned integer → 16-bit unsigned integer
    convert_ushort_sat(i)
32 bit unsigned integer → 32-bit unsigned integer
    no conversion is performed
```

必须正确的使本节所描述的变换饱和。

9 可选扩展

本节描述 OpenCL 规范所支持的可选特性。前一节中讨论了所有实现都必须支持的特性。一些 OpenCL 设备可能支持下面的可选扩展。不要求但期望 OpenCL 实现支持这些扩展；这些功能可能在以后的 OpenCL 修订版本中变成必须要支持的。下面对怎样定义 OpenCL 扩展做了一个简短的描述。

对于 OpenCL 工作组批准的 OpenCL 扩展，使用下列命名规则：

- ✚ 每个扩展都必须有一个唯一的名字，形式如 “**cl_khr_<name>**”。如果某个实现支持此扩展，则 CL_PLATFORM_EXTENSIONS 或 CL_DEVICE_EXTENSIONS (见表 4.3) 中将会包含此字符串。
- ✚ 对于扩展所定义的所有 API 函数，其名字形如 **cl<FunctionName>KHR**。
- ✚ 对于扩展所定义的所有枚举，其名字形如 **cl_<enum_name>_KHR**。

OpenCL 工作组所批准的 OpenCL 扩展，在 OpenCL 的后续修订版本中可能会变成必须的核心特性。如果是这样，那么此扩展的规范会整合到核心规范中，同时将此扩展所定义的函数和枚举移除其词缀 **KHR**。即使如此，CL_PLATFORM_EXTENSIONS 或 CL_DEVICE_EXTENSIONS 中仍然会有此扩展的名字，并且为了帮助迁移仍然支持带词缀 **KHR** 的函数和枚举。

对于厂商的扩展，使用下列命名规则：

- ✚ 每个扩展都必须有一个唯一的名字，形式如 “**cl_<vendor_name>_<name>**”。如果某个实现支持此扩展，则 CL_PLATFORM_EXTENSIONS 或 CL_DEVICE_EXTENSIONS (见表 4.3) 中将会包含此字符串。
- ✚ 对于扩展所定义的所有 API 函数，其名字形如 **cl<FunctionName><vendor_name>**。
- ✚ 对于扩展所定义的所有枚举，其名字形如 **cl_<enum_name>_<vendor_name>**。

9.1 对可选扩展的编译器指令

指令 `#pragma OPENCL EXTENSION` 可以控制 OpenCL 编译器的行为。此指令定义如下：

```
#pragma OPENCL EXTENSION extension_name : behavior
#pragma OPENCL EXTENSION all : behavior
```

其中 `extension_name` 是扩展的名字；对于由 OpenCL 工作组批准的扩展，其形式为 **cl_khr_<name>**，而对于厂商的扩展，形式为 **cl_<vendor_name>_<name>**。第二行中

的“all”意味着“behavior”会作用到编译器所支持的所有扩展上。“behavior”的值如下表所示：

行为	描述
enable	其行为跟扩展 <i>extension_name</i> 一样。 如果不支持扩展 <i>extension_name</i> ，或指定了 all ，那么 #pragma OPENCL EXTENSION 会报告错误。
disable	其行为（包括产生错误和告警）如同扩展 <i>extension_name</i> 不是语言定义的一部分。 如果指定了 all ，则仅有语言核心，不带有任何扩展。 如果不支持 <i>extension_name</i> 则 #pragma OPENCL EXTENSION 会报告警。

指令**#pragma OPENCL EXTENSION** 是用来设置扩展行为的一种简单、低级的机制。它没有定义任何策略，如哪些组合是恰当的；那些必须在其它地方定义。指令的次序会影响扩展的行为。后面的指令会覆盖前面的。变量 **all** 会设置所有扩展的行为，会覆盖所有之前关于扩展的指令，不过仅在 *behavior* 是 **disable** 时是这样的。

编译器的初始状态跟这条指令所产生的效果一样：

```
#pragma OPENCL EXTENSION all : disable
```

它告诉编译器所有的错误和告警必须按此规范来报告，忽略任何扩展。

对于任何一个扩展，只要它会影响 OpenCL 语言的语义、文法或者会添加一些内建函数，那么就必须创建一个与此扩展名字相匹配的预处理器**#define**。只有给定的实现支持此扩展时，这个**#define** 才可用。

例如：

一个添加了字符串“**cl_khr_fp64**”的扩展同样会添加一个名叫 **cl_khr_fp64** 的预处理器**#define**。内核可以使用此预处理器**#define** 来做一些事情，如：

```
#ifdef cl_khr_fp64
    // do something using the extension
#else
    // do something else or #error!
#endif
```

9.2 获取OpenCL扩展函数的指针

函数

void* **clGetExtensionFunctionAddress**³⁸ (const char *funcname)

返回名为 funcname 的扩展函数的地址。应将所返回的指针转换成函数指针类型，且其类型与相应扩展规范和头文件中的定义相一致。如果返回了 NULL，则表明指定的函数不存在；即使不是 NULL，也不保证真正支持此函数。应用必须使用 **clGetPlatformInfo**(platform, CL_PLATFORM_EXTENSIONS, ...)或 **clGetDeviceInfo**(device, CL_DEVICE_EXTENSIONS, ...)进行相应的查询来确定某个 OpenCL 实现是否支持此扩展。

对于 OpenCL 的核心（不带任何扩展）函数，不能用 **clGetExtensionFunctionAddress** 进行查询。对于可以使用 **clGetExtensionFunctionAddress** 进行查询的函数，实现也可以选择从实现这些函数的目标库中将这函数静态导出。即使如此，如果应用想具备更好的移植性，就不要依赖于这一点。

9.3 双精度浮点数

对于一些用于科学计算的算法和应用而言，对双精度浮点数的支持是必须的。可以通过增加对双精度浮点数的支持（作为一个可选扩展）来使能这类应用。

OpenCL 1.0 中将对双精度浮点数的支持作为一个可选扩展。如果应用想使用 double，则在内核代码的任何双精度数据类型之前，需要先包含指令 **#pragma OPENCL EXTENSION cl_khr_fp64 : enable。**

对表 6.1 和表 6.2 中所列的内建标量和矢量数据类型做了扩展，增加了以下类型：

类型	描述
double	双精度浮点数。
double2	2-组件 double 矢量。
double4	4-组件 double 矢量。
double8	8-组件 double 矢量。
double16	16-组件 double 矢量。

在 OpenCL API（和头文件）中，内建的标量和矢量 double 数据类型同样被声明为响应的带前缀 cl_ 的类型，这些类型可以由应用使用。下表列出了其对应关系。

OpenCL 语言中的类型	应用可以使用的 API 类型
---------------	----------------

³⁸ 由于没有办法控制对设备的查询，所以对于不同设备上此扩展的所有实现，所返回的函数指针必须都能工作。如果设备不支持此扩展，那么调用此扩展的函数，其行为未定义。

double	cl_double
double2	cl_double2
double4	cl_double4
double8	cl_double8
double16	cl_double16

double 数据类型必须遵循 IEEE 754 中的双精度存储格式。

下列文字添加到了节 6.1.1.1 中。

从 double 到 half 的变换进行了正确舍入。从 half 到 double 的变换是无损的。

9.3.1 变换

现在,节 6.2.1 所描述的隐式变换规则也包含了标量数据类型 double 和矢量数据类型 doublen。

节 6.2.2 中所描述的显式转换也做了扩展,支持标量数据类型 double 和矢量数据类型 doublen。

节 6.2.3 中所描述的显式变换函数也做了扩展,支持标量数据类型 double 和矢量数据类型 doublen。

节 6.2.4.2 中的函数 as_typen() 也做了扩展,允许在 longn、ulongn 和 doublen 之间进行转换,这些转换与变换无关。

9.3.2 数学函数

对表 6.7 中定义的内建数学函数做了扩展,相应版本的函数可以将 double 和 double{2|4|8|16} 作为参数和返回值。现在 gentype 也包括 double、double2、double4、double8 和 double16。对一个函数的任何特定应用,所有参数和返回值的实际类型都必须一样。

另外,还增加了下列符号常量:

HUGE_VAL	-	正的 double 表达式,等价于正无穷。 用来作为一个错误值,由内建数学函数返回。
----------	---	---

宏 **FP_FAST_FMA** 用来指示,对于双精度浮点数,函数族 **fma()** 是否比直接编码快。如果定义了宏 **FP_FAST_FMA**,则表示,相比对 **double** 操作数的乘和加,函数 **fma()** 执行的一样快,甚至更快。

下列宏必须使用给定的值。这些常量表达式适合在预处理指令 `#if` 中使用。

<code>#define DBL_DIG</code>	<code>15</code>
<code>#define DBL_MANT_DIG</code>	<code>53</code>
<code>#define DBL_MAX_10_EXP</code>	<code>+308</code>
<code>#define DBL_MAX_EXP</code>	<code>+1024</code>
<code>#define DBL_MIN_10_EXP</code>	<code>-307</code>
<code>#define DBL_MIN_EXP</code>	<code>-1021</code>
<code>#define DBL_MAX</code>	<code>0x1.fffffffffffffp1023</code>
<code>#define DBL_MIN</code>	<code>0x1.0p-1022</code>
<code>#define DBL_EPSILON</code>	<code>0x1.0p-52</code>

下表描述了 OpenCL C 编程语言中的内建宏（上面所列）和应用可见宏的对应关系。

OpenCL 语言中的宏	应用所用的宏
<code>DBL_DIG</code>	<code>CL_DBL_DIG</code>
<code>DBL_MANT_DIG</code>	<code>CL_DBL_MANT_DIG</code>
<code>DBL_MAX_10_EXP</code>	<code>CL_DBL_MAX_10_EXP</code>
<code>DBL_MAX_EXP</code>	<code>CL_DBL_MAX_EXP</code>
<code>DBL_MIN_10_EXP</code>	<code>CL_DBL_MIN_10_EXP</code>
<code>DBL_MIN_EXP</code>	<code>CL_DBL_MIN_EXP</code>
<code>DBL_MAX</code>	<code>CL_DBL_MAX</code>
<code>DBL_MIN</code>	<code>CL_DBL_MIN</code>
<code>DBL_EPSILON</code>	<code>CL_DBL_EPSILON</code>

下列常量也是可用的。类型都是 `double`，在 `double` 的精度内都是准确的。

常量	描述
<code>M_E</code>	e
<code>M_LOG2E</code>	$\log_2 e$
<code>M_LOG10E</code>	$\log_{10} e$
<code>M_LN2</code>	$\log_e 2$
<code>M_LN10</code>	$\log_e 10$
<code>M_PI</code>	π
<code>M_PI_2</code>	$\pi/2$
<code>M_PI_4</code>	$\pi/4$
<code>M_1_PI</code>	$1/\pi$
<code>M_2_PI</code>	$2/\pi$
<code>M_2_SQRTPI</code>	$2/\sqrt{\pi}$
<code>M_SQRT2</code>	$\sqrt{2}$
<code>M_SQRT1_2</code>	$1/\sqrt{2}$

9.3.3 公共函数³⁹

对表 6.11 中所定义的内建公共函数做了扩展，相应版本的函数可以将 `double` 和 `double{2|4|8|16}` 作为参数和返回值。现在 `gentype` 也包括 `double`、`double2`、`double4`、`double8` 和 `double16`。

9.3.4 几何函数⁴⁰

对表 6.12 中所定义的内建几何函数做了扩展，相应版本的函数可以将 `double` 和 `double{2|4}` 作为参数和返回值。现在 `gentype` 也包括 `double`、`double2` 和 `double4`。

9.3.5 关系函数

对表 6.13 中所定义的内建关系函数做了扩展，相应版本的函数可以将 `double` 和 `double{2|4}` 作为参数。现在 `gentype` 也包括 `double`、`double2` 和 `double4`。

关系运算符和判等运算符 (`<`、`<=`、`>`、`>=`、`!=`、`==`) 可以与矢量类型 `doublen` 一起使用，所产生的结果类型为 `longn` (如节 6.3 中所描述的那样)。

函数	描述
<code>int isequal(double x, double y)</code> <code>longn isequal(doublen x, doublen y)</code>	返回组件级的比较 $x == y$ 。
<code>int isnotequal(double x, double y)</code> <code>longn isnotequal(doublen x, doublen y)</code>	返回组件级的比较 $x != y$ 。
<code>int isgreater(double x, double y)</code> <code>longn isgreater(doublen x, doublen y)</code>	返回组件级的比较 $x > y$ 。
<code>int isgreaterequal(double x, double y)</code> <code>longn isgreaterequal(doublen x, doublen y)</code>	返回组件级的比较 $x >= y$ 。
<code>int isless(double x, double y)</code> <code>longn isless(doublen x, doublen y)</code>	返回组件级的比较 $x < y$ 。
<code>int islessequal(double x, double y)</code> <code>longn islessequal(doublen x, doublen y)</code>	返回组件级的比较 $x <= y$ 。
<code>int islessgreater(double x, double y)</code> <code>longn islessgreater(doublen x, doublen y)</code>	返回组件级的比较 $(x < y) (x > y)$ 。
<code>int isfinite(double)</code>	测试参数是否一个有限值。

³⁹ 函数 `mix` 和 `smoothstep` 在实现时可以使用收缩，如同 `mad` 或 `fma` 一样。

⁴⁰ 几何函数在实现时可以使用收缩，如同 `mad` 或 `fma` 一样。

<code>longn isfinite(double n)</code>	
<code>int isinf(double)</code> <code>longn isinf(double n)</code>	测试参数是否一个无穷值 (+ve 或 -ve)。
<code>int isnan(double)</code> <code>longn isnan(double n)</code>	测试参数是否 NaN。
<code>int isnormal(double)</code> <code>longn isnormal(double n)</code>	测试参数是否一个正常值。
<code>int isordered(double x, double y)</code> <code>longn isordered(double n x, double n y)</code>	测试参数是否有序。参数为 x 和 y , 结果是 <code>isequal(x, x) && isequal(y, y)</code>
<code>int isunordered(double x, double y)</code> <code>longn isunordered(double n x, double n y)</code>	测试参数是否无序。参数为 x 和 y , 任何一个 NaN 则返回一个非零值, 否则返回 0。
<code>int signbit(double)</code> <code>longn signbit(double n)</code>	测试符号位。对于此函数的标量版本, 如果设置了参数的符号位, 则返回 1, 否则返回 0。而对于矢量版本的每个组件: 如果设置了符号位则返回 -1, 否则返回 0。
<code>double n bitselect (double n a, double n b, double n c)</code>	对于结果中的每个比特, 如果 c 中相应比特是 0, 则返回 a 中的相应比特, 否则返回 b 中的相应比特。
<code>double n select (double n a, double n b, long n c)</code> <code>double n select (double n a, double n b, ulong n c)</code>	对于每个组件, 如果 c 中相应组件设置了 MSB, 则返回 b 中相应组件, 否则返回 a 中相应组件。

9.3.6 加载和存储矢量数据的函数

对表 6.14 中所定义的矢量数据读写函数 (`vloadn` 和 `vstoren`) 做了扩展, 相应版本的函数可以读写标量或矢量 `double` 值。现在 `gentype` 也包括 `double`; 现在 `gentypen` 也包括 `double`、`double2`、`double4`、`double8` 和 `double16`。函数 `vstore_half`、`vstore_halfn` 和 `vstorea_halfn` 允许将双精度标量或矢量值当做 `half` 值写入内存。

函数	描述
<code>gentypen vloadn (size_t offset, const __global gentype *p)</code> <code>gentypen vloadn (size_t offset, const __local gentype *p)</code> <code>gentypen vloadn (size_t offset, const __constant gentype *p)</code> <code>gentypen vloadn (size_t offset, const __private gentype *p)</code>	返回从位置 $(p + (offset * n))$ 读取的 <code>sizeof(gentypen)</code> 字节的数据。对于所计算的地址 $(p + (offset * n))$, 如果 <code>gentype</code> 是 <code>char n</code> 、 <code>uchar n</code> , 则必须按 8 比特对齐; 如果 <code>gentype</code> 是 <code>short n</code> 、 <code>ushort n</code> , 则必须按 16 比特对齐; 如果 <code>gentype</code> 是 <code>int n</code> 、 <code>uint n</code> , 则必须按 32 比特对齐; 如果 <code>gentype</code> 是 <code>long n</code> 、 <code>ulong n</code> , 则必须按 64 比特对齐。
<code>void vstoren (gentypen data,</code>	将 <code>data</code> 中 <code>sizeof(gentypen)</code> 字节的数据写入地址

<pre> size_t offset, __global gentype *p) void vstoren(gentype n data, size_t offset, __local gentype *p) void vstoren(gentype n data, size_t offset, __private gentype *p) </pre>	<p>($p + (offset * n)$)中。对于所计算的地址 ($p + (offset * n)$)，如果 gentype 是 charn、ucharn，则必须按 8 比特对齐；如果 gentype 是 shortn、ushortn，则必须按 16 比特对齐；如果 gentype 是 intn、uintn，则必须按 32 比特对齐；如果 gentype 是 longn、ulongn，则必须按 64 比特对齐。</p>
<pre> void vstore_half(double data, size_t offset, __global half *p) void vstore_half_ret(double data, size_t offset, __global half *p) void vstore_half_rtz(double data, size_t offset, __global half *p) void vstore_half_rtp(double data, size_t offset, __global half *p) void vstore_half_rtn(double data, size_t offset, __global half *p) void vstore_half(double data, size_t offset, __local half *p) void vstore_half_ret(double data, size_t offset, __local half *p) void vstore_half_rtz(double data, size_t offset, __local half *p) void vstore_half_rtp(double data, size_t offset, __local half *p) void vstore_half_rtn(double data, size_t offset, __local half *p) void vstore_half(double data, size_t offset, __private half *p) void vstore_half_ret(double data, size_t offset, __private half *p) void vstore_half_rtz(double data, size_t offset, __private half *p) void vstore_half_rtp(double data, size_t offset, __private half *p) void vstore_half_rtn(double data, size_t offset, __private half *p) </pre>	<p>先将 $data$ 中的双精度浮点值变换成 half 值(使用相应的舍入模式)，然后将 half 值写入地址 ($p + offset$)，此地址按 16 比特进行对齐。 vstore_half 使用当前的舍入模式。默认当前舍入模式是舍入到最近的偶数。</p>
<pre> void vstore_halfn(double n data, size_t offset, __global half *p) void vstore_halfn_ret(double n data, size_t offset, __global half *p) void vstore_halfn_rtz(double n data, size_t offset, __global half *p) </pre>	<p>先将 $data$ 中的双精度浮点值变换成 halfn 值(使用相应的舍入模式)，然后将 halfn 值写入地址 ($p + (offset * n)$)，此地址按 16 比特进行对齐。 vstore_halfn 使用当前的舍入模式。默认当前舍入模式是舍入到最近的偶数。</p>

<pre> void vstore_halfn_rtp (double <i>n data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstore_halfn_rtn (double <i>n data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstore_halfn (double <i>n data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_halfn_ret (double <i>n data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_halfn_rtz (double <i>n data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_halfn_rtp (double <i>n data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_halfn_rtn (double <i>n data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstore_halfn (double <i>n data</i>, size_t <i>offset</i>, __private half *<i>p</i>) void vstore_halfn_ret (double <i>n data</i>, size_t <i>offset</i>, __private half *<i>p</i>) void vstore_halfn_rtz (double <i>n data</i>, size_t <i>offset</i>, __private half *<i>p</i>) void vstore_halfn_rtp (double <i>n data</i>, size_t <i>offset</i>, __private half *<i>p</i>) void vstore_halfn_rtn (double <i>n data</i>, size_t <i>offset</i>, __private half *<i>p</i>) </pre>	
<pre> void vstorea_halfn (double <i>n data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstorea_halfn_ret (double <i>n data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstorea_halfn_rtz (double <i>n data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstorea_halfn_rtp (double <i>n data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstorea_halfn_rtn (double <i>n data</i>, size_t <i>offset</i>, __global half *<i>p</i>) void vstorea_halfn (double <i>n data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstorea_halfn_ret (double <i>n data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstorea_halfn_rtz (double <i>n data</i>, size_t <i>offset</i>, __local half *<i>p</i>) void vstorea_halfn_rtp (double <i>n data</i>, size_t <i>offset</i>, __local half *<i>p</i>) </pre>	<p>先将 <i>data</i> 中的双精度浮点值转换成 half 值(使用相应的舍入模式), 然后将 half<i>n</i> 值写入地址 $(p + (offset * n))$, 此地址按 $sizeof(half) * n$ 进行对齐。</p> <p>vstore_halfn 使用当前的舍入模式。默认当前舍入模式是舍入到最近的偶数。</p>

<div>size_t offset, __ local half *p)</div> <div>void vstorea_halfn_rtn (doublen data,</div> <div>size_t offset, __ local half *p)</div> <div>void vstorea_halfn (doublen data,</div> <div>size_t offset, __ private half *p)</div> <div>void vstorea_halfn_ret (doublen data,</div> <div>size_t offset, __ private half *p)</div> <div>void vstorea_halfn_rtz (doublen data,</div> <div>size_t offset, __ private half *p)</div> <div>void vstorea_halfn_rtp (doublen data,</div> <div>size_t offset, __ private half *p)</div> <div>void vstorea_halfn_rtn (doublen data,</div> <div>size_t offset, __ private half *p)</div>	
---	--

9.3.7 全局内存和局部内存间的异步拷贝，以及预取

对表 6.19 中所定义的内建异步拷贝和预取函数做了扩展；现在 gentypen 也包括 double、double2、double4、double8 和 double16。

9.3.8 IEEE 754 一致性

作为对表 4.3 的补充，下表中的表项允许应用使用 **clGetDeviceInfo** 查询一个支持双精度浮点数的 OpenCL 设备的配置信息。

Op-code	返回类型	描述
CL_DEVICE_DOUBLE_FP_CONFIG	cl_device_fp_config	描述 OpenCL 设备的双精度浮点能力。这是一个位域，可以是下列值的一个或多个： CL_FP_DENORM——支持去规格化数 CL_FP_INF_NAN——支持 INF 和 NaN CL_FP_ROUND_TO_NEAREST——支持舍入模式：舍入到最近偶数 CL_FP_ROUND_TO_ZERO——支持向零舍入 CL_FP_ROUND_TO_INF——支持舍入模式：向+ve 和 -ve 无穷舍入 CP_FP_FMA——支持 IEEE 754-2008 中的混合乘加

		必须要支持的最小的双精度浮点能力是 CL_FP_FMA CL_FP_ROUND_TO_NEAREST CL_FP_ROUND_TO_ZERO CL_FP_ROUND_TO_INF CL_FP_INF_NAN CL_FP_DENORM。
--	--	--

对双精度浮点数及其上的运算而言，要求支持 IEEE 754 的混合乘加、去规格化数、INF 和 NaN。

9.3.9 相对误差即ULP

本节我们讨论定义为 ulp 的最大相对误差。加、减、乘、除、混合乘加以及整数和浮点格式间的变换都遵循 IEEE 754，因此都可以使用舍入模式：舍入到最近偶数正确的进行舍入。

下表列出了双精度浮点算术运算的最小精确度，以 ULP 的形式给出。计算 ULP 值时所参考的是进行算数运算时具有无穷精度的结果。

函数	最小精确度——ULP值 ⁴¹
$x + y$	正确舍入
$x - y$	正确舍入
$x * y$	正确舍入
$1.0/x$	正确舍入
x/y	正确舍入
acos	$\leq 4ulp$
acospi	$\leq 5ulp$
asin	$\leq 4ulp$
asinpi	$\leq 5ulp$
atan	$\leq 5ulp$
atan2	$\leq 6ulp$
atanpi	$\leq 5ulp$
atan2pi	$\leq 6ulp$
acosh	$\leq 4ulp$
asinh	$\leq 4ulp$
atanh	$\leq 5ulp$
cbrt	$\leq 2ulp$
ceil	正确舍入
copysign	$0ulp$
cos	$\leq 4ulp$
cosh	$\leq 4ulp$

⁴¹ 对于不需要舍入的函数，所用的 ULP 值是 0。

<i>cospi</i>	$\leq 4ulp$
<i>erfc</i>	$\leq 16ulp$
<i>erf</i>	$\leq 16ulp$
<i>exp</i>	$\leq 3ulp$
<i>exp2</i>	$\leq 3ulp$
<i>exp10</i>	$\leq 3ulp$
<i>expm1</i>	$\leq 3ulp$
<i>fabs</i>	$0ulp$
<i>fdim</i>	正确舍入
<i>floor</i>	正确舍入
<i>fma</i>	正确舍入
<i>fmax</i>	$0ulp$
<i>fmin</i>	$0ulp$
<i>fmod</i>	$0ulp$
<i>fract</i>	$\leq 1ulp$
<i>frexp</i>	$0ulp$
<i>hypot</i>	$\leq 4ulp$
<i>ilogb</i>	$0ulp$
<i>ldexp</i>	正确舍入
<i>log</i>	$\leq 3ulp$
<i>log2</i>	$\leq 3ulp$
<i>log10</i>	$\leq 3ulp$
<i>log1p</i>	$\leq 2ulp$
<i>logb</i>	$0ulp$
<i>mad</i>	所允许的任何值 (无穷 ulp)
<i>modf</i>	$0ulp$
<i>nan</i>	$0ulp$
<i>nextafter</i>	$0ulp$
<i>pow(x,y)</i>	$\leq 16ulp$
<i>pown(x,y)</i>	$\leq 16ulp$
<i>powr(x,y)</i>	$\leq 16ulp$
<i>remainder</i>	$0ulp$
<i>remquo</i>	$0ulp$
<i>rint</i>	正确舍入
<i>rootn</i>	$\leq 16ulp$
<i>round</i>	正确舍入
<i>rsqrt</i>	$\leq 2ulp$
<i>sin</i>	$\leq 4ulp$
<i>sincos</i>	对于正弦值和余弦值： $\leq 4ulp$
<i>sinh</i>	$\leq 4ulp$
<i>sinpi</i>	$\leq 4ulp$
<i>sqrt</i>	正确舍入
<i>tan</i>	$\leq 5ulp$
<i>tanh</i>	$\leq 5ulp$
<i>tanpi</i>	$\leq 6ulp$
<i>tgamma</i>	$\leq 16ulp$
<i>trunc</i>	正确舍入

9.4 选择舍入模式

作为 OpenCL 1.0 的一个扩展选项,允许在程序源码中为一条指令或一组指令指定舍入模式。如果想使用此特性,应用需要包括此指令:

```
#pragma OPENCL EXTENSION cl_khr_select_fprounding_mode : enable
```

如果支持扩展 **cl_khr_select_fprounding_mode**，OpenCL 实现必须支持对单精度浮点数的所有四种舍入模式，即表 4.3 中的 CL_DEVICE_SINGLE_FP_CONFIG 必须包括 CL_FP_ROUND_TO_ZERO 和 CL_FP_ROUND_TO_INF。对双精度浮点数也是一样。

在程序源码中使用下面的 pragma 来指定舍入模式：

```
#pragma OPENCL SELECT_ROUNDING_MODE rounding-mode
```

其中 *rounding-mode* 可以是下列值之一：

- ✚ **rte** —— 舍入到最近偶数
- ✚ **rtz** —— 向零舍入
- ✚ **rtp** —— 向正无穷舍入
- ✚ **rtn** —— 向负无穷舍入

在程序源码中，**#pragma OPENCL SELECT_ROUNDING_MODE** 会为后面所有浮点类型（标量或矢量类型）上的运算或产生浮点值的所有指令设置舍入模式，直到遇到下一个 **#pragma OPENCL SELECT_ROUNDING_MODE**。在编译时会识别为一个代码块指定的舍入模式。如果处于复合语句中，从其出现之处开始，此编译指示会一直有效，直到遇到另一个编译指示 **SELECT_ROUNDING_MODE**（包括处于内部嵌套复合语句中的），或者直到复合语句结束；在复合语句结束之处，舍入模式会恢复成此复合语句之前的状态。除非文档中做了明确说明，否则被调用的函数不会继承调用者的舍入模式。

如果使能了此扩展，会根据当前舍入模式定义预处理器符号 **__ROUNDING_MODE__**，其值为下列之一：

```
#define __ROUNDING_MODE__ rte
#define __ROUNDING_MODE__ rtz
#define __ROUNDING_MODE__ rtp
#define __ROUNDING_MODE__ rtn
```

这样我们可以用下面命令将 **foo()** 重新映射到 **foo_rte()** 上：

```
#define foo foo##__ROUNDING_MODE__
```

默认的舍入模式是舍入到最近偶数；节 6.11.2 中的内建数学函数，节 6.11.4 中的公共函数和节 6.11.5 中的几何函数在实现时使用的就是此舍入模式。如果没有指定舍入模式，包括内建的各种变换以及内建函数 **vstore_half** 和 **vstorea_half**，那么就继承当前的舍入模式。从浮点数到整型的变换始终使用 **rtz** 模式，除非用户明确要求使用另外一种舍入模式。

下面有一些例子，描述了舍入模式的选择是怎样工作的。

```
#pragma OPENCL SELECT_ROUNDING_MODE rtz
    float4 a = b * c;  ← uses round to zero rounding mode.

#pragma OPENCL SELECT_ROUNDING_MODE rtp
    float4 d = foo(a);  ← function foo uses rounding mode specified
                        where source for foo() is implemented.
```

9.5 32 位整数的原子函数

OpenCL 1.0 支持下面的可选扩展，以实现在 `__global` 内存中对 32 位带符号和无符号整型的原子操作：

cl_khr_global_int32_base_atomics 和 **cl_khr_global_int32_extended_atomics**

要想使用这些扩展，OpenCL 程序源码中需要包括 `#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable` 或 `#pragma OPENCL EXTENSION cl_khr_global_int32_extended_atomics : enable`。

表 9.1 列出了扩展 **cl_khr_global_int32_base_atomics** 所支持的原子函数；表 9.2 列出了扩展 **cl_khr_global_int32_extended_atomics** 所支持的原子函数；这些函数都会在一个原子事物中执行。

对于执行这些原子函数的设备而言，这些事务都是原子的。如果多个设备上同时执行相同内存位置上的原子操作，那么不保证其原子性。

表 9.1 扩展 **cl_khr_global_int32_base_atomics** 所实现的内置原子函数

函数	描述
int atom_add (__global int *p, int val) unsigned int atom_add (__global unsigned int *p, unsigned int val)	读取 <i>p</i> 所指位置存储的 32 位值（用 <i>old</i> 表示）。计算 (<i>old</i> + <i>val</i>) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_sub (__global int *p, int val) unsigned int atom_sub (__global unsigned int *p, unsigned int val)	读取 <i>p</i> 所指位置存储的 32 位值（用 <i>old</i> 表示）。计算 (<i>old</i> - <i>val</i>) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_xchg (__global int *p, int val) unsigned int atom_xchg (__global unsigned int *p, unsigned int val)	交换位置 <i>p</i> 所存储的值 <i>old</i> 和 <i>val</i> 。返回 <i>old</i> 。
int atom_inc (__global int *p) unsigned int atom_inc (__global unsigned int *p)	读取 <i>p</i> 所指位置存储的 32 位值（用 <i>old</i> 表示）。计算 (<i>old</i> + 1) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_dec (__global int *p) unsigned int atom_dec (__global unsigned int *p)	读取 <i>p</i> 所指位置存储的 32 位值（用 <i>old</i> 表示）。计算 (<i>old</i> - 1) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。

<code>__global unsigned int *p)</code>	<i>old</i> .
int atom_cmpxchg (<code>__global int *p,</code> <code>int cmp, int val)</code> unsigned int atom_cmpxchg (<code>__global unsigned int *p,</code> <code>unsigned int cmp,</code> <code>unsigned int val)</code>	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 $(old == cmp) ? val : old$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。

表 9.2 扩展 cl_khr_global_int32_extended_atomics 所实现的内置原子函数

函数	描述
int atom_min (<code>__global int *p, int val)</code> unsigned int atom_min (<code>__global unsigned int *p,</code> <code>unsigned int val)</code>	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 $\min(old, val)$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_max (<code>__global int *p, int val)</code> unsigned int atom_max (<code>__global unsigned int *p,</code> <code>unsigned int val)</code>	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 $\max(old, val)$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_and (<code>__global int *p, int val)</code> unsigned int atom_and (<code>__global unsigned int *p,</code> <code>unsigned int val)</code>	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 $(old \& val)$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_or (<code>__global int *p, int val)</code> unsigned int atom_or (<code>__global unsigned int *p,</code> <code>unsigned int val)</code>	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 $(old val)$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_xor (<code>__global int *p, int val)</code> unsigned int atom_xor (<code>__global unsigned int *p,</code> <code>unsigned int val)</code>	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 $(old \wedge val)$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。

9.6 32 位整数的局部原子操作

OpenCL 1.0 支持下面的可选扩展,以实现在 `__local` 内存中对 32 位带符号和无符号整型的原子操作:

cl_khr_local_int32_base_atomics 和 cl_khr_local_int32_extended_atomics

要想使用这些扩展, OpenCL 程序源码中需要包括 `#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable` 或 `#pragma OPENCL EXTENSION cl_khr_local_int32_extended_atomics : enable`。

表 9.3 列出了扩展 **cl_khr_local_int32_base_atomics** 所支持的原子函数；表 9.4 列出了扩展 **cl_khr_local_int32_extended_atomics** 所支持的原子函数；这些函数都会在一个原子事物中执行。

表 9.3 扩展 cl_khr_local_int32_base_atomics 所实现的内建原子函数

函数	描述
int atom_add (__local int * <i>p</i> , int <i>val</i>) unsigned int atom_add (__local unsigned int * <i>p</i> , unsigned int <i>val</i>)	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 (<i>old</i> + <i>val</i>) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_sub (__local int * <i>p</i> , int <i>val</i>) unsigned int atom_sub (__local unsigned int * <i>p</i> , unsigned int <i>val</i>)	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 (<i>old</i> - <i>val</i>) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_xchg (__local int * <i>p</i> , int <i>val</i>) unsigned int atom_xchg (__local unsigned int * <i>p</i> , unsigned int <i>val</i>)	交换位置 <i>p</i> 所存储的值 <i>old</i> 和 <i>val</i> 。返回 <i>old</i> 。
int atom_inc (__local int * <i>p</i>) unsigned int atom_inc (__local unsigned int * <i>p</i>)	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 (<i>old</i> + 1) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_dec (__local int * <i>p</i>) unsigned int atom_dec (__local unsigned int * <i>p</i>)	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 (<i>old</i> - 1) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_cmpxchg (__local int * <i>p</i> , int <i>cmp</i> , int <i>val</i>) unsigned int atom_cmpxchg (__local unsigned int * <i>p</i> , unsigned int <i>cmp</i> , unsigned int <i>val</i>)	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 (<i>old</i> == <i>cmp</i>) ? <i>val</i> : <i>old</i> 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。

表 9.4 扩展 cl_khr_local_int32_extended_atomics 所实现的内建原子函数

函数	描述
int atom_min (__local int * <i>p</i> , int <i>val</i>) unsigned int atom_min (__local unsigned int * <i>p</i> , unsigned int <i>val</i>)	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 min (<i>old</i> , <i>val</i>) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_max (__local int * <i>p</i> , int <i>val</i>) unsigned int atom_max (__local unsigned int * <i>p</i> ,	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算 max (<i>old</i> , <i>val</i>) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。

unsigned int <i>val</i>)	
int atom_and (__local int * <i>p</i> , int <i>val</i>) unsigned int atom_and (__local unsigned int * <i>p</i> , unsigned int <i>val</i>)	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算(<i>old</i> & <i>val</i>)并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_or (__local int * <i>p</i> , int <i>val</i>) unsigned int atom_or (__local unsigned int * <i>p</i> , unsigned int <i>val</i>)	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算(<i>old</i> <i>val</i>)并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
int atom_xor (__local int * <i>p</i> , int <i>val</i>) unsigned int atom_xor (__local unsigned int * <i>p</i> , unsigned int <i>val</i>)	读取 <i>p</i> 所指位置存储的 32 位值 (用 <i>old</i> 表示)。计算(<i>old</i> ^ <i>val</i>)并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。

9.7 64 位原子操作

OpenCL 1.0 支持下面的可选扩展,以实现在__global 和__local 内存中对 64 位带符号和无符号整型的原子操作:

cl_khr_int64_base_atomics 和 cl_khr_int64_extended_atomics

要想使用这些扩展, OpenCL 程序源码中需要包括 `#pragma OPENCL EXTENSION cl_khr_int64_base_atomics : enable` 或 `#pragma OPENCL EXTENSION cl_khr_int64_extended_atomics : enable`。

表 9.5 列出了扩展 **cl_khr_int64_base_atomics** 所支持的原子函数;表 9.6 列出了扩展 **cl_khr_int64_extended_atomics** 所支持的原子函数。这些函数都会在一个原子事物中执行。

对于执行这些原子函数的设备而言,这些事务都是原子的。如果多个设备上同时执行相同内存位置上的原子操作,那么不保证其原子性。

表 9.5 扩展 cl_khr_int64_base_atomics 所实现的内置原子函数

函数	描述
long atom_add (__global long * <i>p</i> , long <i>val</i>) long atom_add (__local long * <i>p</i> , long <i>val</i>) ulong atom_add (__global ulong * <i>p</i> , ulong <i>val</i>) ulong atom_add (__local ulong * <i>p</i> , ulong <i>val</i>)	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算(<i>old</i> + <i>val</i>)并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
long atom_sub (__global long * <i>p</i> , long <i>val</i>) long atom_sub (__local long * <i>p</i> , long <i>val</i>)	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算(<i>old</i> - <i>val</i>)并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。

<ul style="list-style-type: none"> ulong atom_sub (__global ulong *<i>p</i>, ulong <i>val</i>) ulong atom_sub (__local ulong *<i>p</i>, ulong <i>val</i>) 	
<ul style="list-style-type: none"> long atom_xchg (__global long *<i>p</i>, long <i>val</i>) long atom_xchg (__local long *<i>p</i>, long <i>val</i>) ulong atom_xchg (__global ulong *<i>p</i>, ulong <i>val</i>) ulong atom_xchg (__local ulong *<i>p</i>, ulong <i>val</i>) 	交换位置 <i>p</i> 所存储的值 <i>old</i> 和 <i>val</i> 。返回 <i>old</i> 。
<ul style="list-style-type: none"> long atom_inc (__global long *<i>p</i>) long atom_inc (__local long *<i>p</i>) ulong atom_inc (__global ulong *<i>p</i>) ulong atom_inc (__local ulong *<i>p</i>) 	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算 $(old + 1)$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
<ul style="list-style-type: none"> long atom_dec (__global long *<i>p</i>) long atom_dec (__local long *<i>p</i>) ulong atom_dec (__global ulong *<i>p</i>) ulong atom_dec (__local ulong *<i>p</i>) 	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算 $(old - 1)$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
<ul style="list-style-type: none"> long atom_cmpxchg (__global long *<i>p</i>, long <i>cmp</i>, long <i>val</i>) long atom_cmpxchg (__local long *<i>p</i>, long <i>cmp</i>, long <i>val</i>) ulong atom_cmpxchg (__global ulong *<i>p</i>, ulong <i>cmp</i>, ulong <i>val</i>) ulong atom_cmpxchg (__local ulong *<i>p</i>, ulong <i>cmp</i>, ulong <i>val</i>) 	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算 $(old == cmp) ? val : old$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。

表 9.6 扩展 cl_khr_int64_extended_atomics 所实现的内置原子函数

函数	描述
<ul style="list-style-type: none"> long atom_min (__global long *<i>p</i>, long <i>val</i>) long atom_min (__local long *<i>p</i>, long <i>val</i>) ulong atom_min (__global ulong *<i>p</i>, ulong <i>val</i>) ulong atom_min (__local ulong *<i>p</i>, ulong <i>val</i>) 	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算 $\min(old, val)$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
<ul style="list-style-type: none"> long atom_max (__global long *<i>p</i>, long <i>val</i>) long atom_max (__local long *<i>p</i>, long <i>val</i>) ulong atom_max (__global ulong *<i>p</i>, ulong <i>val</i>) ulong atom_max (__local ulong *<i>p</i>, ulong <i>val</i>) 	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算 $\max(old, val)$ 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> 。
<ul style="list-style-type: none"> long atom_and (__global long *<i>p</i>, long <i>val</i>) long atom_and (__local long *<i>p</i>, long <i>val</i>) 	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算 $(old \& val)$ 并将结果存储到 <i>p</i> 所指位置中。返回

<ul style="list-style-type: none"> ulong atom_and (__global ulong *<i>p</i>, ulong <i>val</i>) ulong atom_and (__local ulong *<i>p</i>, ulong <i>val</i>) 	<i>old</i> .
<ul style="list-style-type: none"> long atom_or (__global long *<i>p</i>, long <i>val</i>) long atom_or (__local long *<i>p</i>, long <i>val</i>) ulong atom_or (__global ulong *<i>p</i>, ulong <i>val</i>) ulong atom_or (__local ulong *<i>p</i>, ulong <i>val</i>) 	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算 (<i>old</i> <i>val</i>) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> .
<ul style="list-style-type: none"> long atom_xor (__global long *<i>p</i>, long <i>val</i>) long atom_xor (__local long *<i>p</i>, long <i>val</i>) ulong atom_xor (__global ulong *<i>p</i>, ulong <i>val</i>) ulong atom_xor (__local ulong *<i>p</i>, ulong <i>val</i>) 	读取 <i>p</i> 所指位置存储的 64 位值 (用 <i>old</i> 表示)。计算 (<i>old</i> ^ <i>val</i>) 并将结果存储到 <i>p</i> 所指位置中。返回 <i>old</i> .

9.8 写入 3D 图像对象

OpenCL 1.0 支持由内核对 2D 图像对象进行读写。在一个内核中，不能对同一个 2D 图像对象既读又写。OpenCL 1.0 同样支持由内核对 3D 图像对象进行读操作，但不允许对其进行写操作。扩展 `cl_khr_3d_image_writes` 实现了对 3D 图像对象的写操作。在一个内核中，不能对同一个 3D 图像对象既读又写。

要想使用此扩展对 3D 图像对象进行写操作，需要在 OpenCL 程序源码中包括指令 `#pragma OPENCL EXTENSION cl_khr_3d_image_writes : enable.`

扩展 `cl_khr_3d_image_writes` 所实现的内建函数如下表所示。

函数	描述
<ul style="list-style-type: none"> void write_imagef (image3d_t <i>image</i>, int4 <i>coord</i>, float4 <i>color</i>) void write_imagei (image3d_t <i>image</i>, int4 <i>coord</i>, int4 <i>color</i>) void write_imageui (image3d_t <i>image</i>, int4 <i>coord</i>, unsigned int4 <i>color</i>) 	<p>将 <i>color</i> 写到 3D 图像对象 <i>image</i> 中坐标 (<i>x</i>, <i>y</i>, <i>z</i>) 处。在写入之前会先进行适当的数据格式变换。<i>coord.x</i>、<i>coord.y</i>、<i>coord.z</i> 都是非规范化坐标，取值范围分别是 0 到图像宽度-1、0 到图像高度-1、0 到图像深度-1。</p> <p>write_imagef 的参数 <i>image</i> 在创建时所使用的 <i>image_channel_data_type</i> 必须是预定义的压缩格式之一或 CL_SNORM_INT8、CL_UNORM_INT8、CL_SNORM_INT16、CL_UNORM_INT16、CL_HALF_FLOAT 或 CL_FLOAT。</p> <p>write_imagei 的参数 <i>image</i> 在创建时所使用的 <i>image_channel_data_type</i> 必须是 CL_SIGNED_INT8、CL_SIGNED_INT16 或 CL_SIGNED_INT32。</p> <p>write_imageui 的参数 <i>image</i> 在创建时所使用的 <i>image_channel_data_type</i> 必须是</p>

	CL_UNSIGNED_INT8、CL_UNSIGNED_INT16 或 CL_UNSIGNED_INT32。 如果创建图像对象时 <i>image_channel_data_type</i> 的值不是上面所列的值，或者坐标(<i>x</i> , <i>y</i> , <i>z</i>)不在范围 (0 ... <i>width</i> - 1, 0 ... <i>height</i> - 1, 0 ... <i>depth</i> - 1)之内，则 write_imagef 、 write_imagei 和 write_imageui 的行为是未定义的。
--	--

对于可写的 3D 图像，至少要支持的图像格式如下表所示：

image_num_channels	image_channel_order	image_channel_data_type
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
4	CL_BGRA	CL_UNORM_INT8

9.9 可按字节寻址的存储

节 6.8.m 列出了对内建类型 `char`、`uchar`、`char2`、`uchar2`、`short` 和 `half` 的限制。OpenCL 扩展 `cl_khr_byte_addressable_store` 则移除了这些限制。对于应用而言，要想对类型为 `char`、`uchar`、`char2`、`uchar2`、`short`、`ushort` 和 `half` 的指针（或结构体）进行写操作，在任何执行此类写操作的代码之前需要包括 `#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable`；当然，节 6.8.m 可能不支持这类写操作。

下面是节 6.8.m 中所给的例子 如果支持扩展 `cl_khr_byte_addressable_store`，则例子中加粗的几行可以正确编译并正确工作。

```
__kernel void
do_proc (__global char *pA, short b, __global short *pB)
{
    char          x[100];
    __private char *px = x;
    int           id = (int)get_global_id(0);
    short         f;

    f = pB[id] + b;

    px[1] = pA[1];    ← no longer an error.
}
```

```
pB[id] = b;      ← no longer an error.
}
```

9.10 半浮点

在 OpenCL 1.0 的核心规范中，仅支持将 half 作为一种存储格式。在这个扩展中，将 half 标量和矢量类型作为内建类型，且可以用来进行算术运算、变换等。对于应用而言，要想使用类型 half 和 halfn 需要包括指令 `#pragma OPENCL EXTENSION cl_khr_fp16 : enable.`

此扩展对表 6.1 和表 6.2 中的内建标量和矢量数据类型做了扩展，增加了下列类型：

类型	描述
half2	2-组件半精度浮点矢量。
half4	4-组件半精度浮点矢量。
half8	8-组件半精度浮点矢量。
half16	16-组件半精度浮点矢量。

内建矢量数据类型 halfn 同时也在 OpenCL API(和头文件)中被声明为适当的类型，从而可以由应用来使用。下表描述了 OpenCL C 编程语言的 内建矢量数据类型 halfn 与应用可用数据类型的对应关系：

OpenCL 语言中的类型	应用所用的 API 类型
half2	cl_half2
half4	cl_half4
half8	cl_half8
half16	cl_half16

9.10.1 变换

现在，节 6.2.1 所描述的隐式变换规则也包含了标量数据类型 half 和矢量数据类型 halfn。

节 6.2.2 中所描述的显式转换也做了扩展，支持标量数据类型 half 和矢量数据类型 halfn。

节 6.2.3 中所描述的显式变换函数也做了扩展，支持标量数据类型 half 和矢量数据类型 halfn。

节 6.2.4.2 中的函数 as_typed() 也做了扩展，允许在 shortn、ushortn 和 halfn 之间进行转换，这些转换与变换无关。

9.10.2 数学函数

对表 6.7 中定义的内建数学函数做了扩展，相应版本的函数可以将 `half` 和 `half{2|4|8|16}` 作为参数和返回值。现在 `gentype` 也包括 `half`、`half2`、`half4`、`half8` 和 `half16`。对一个函数的任何特定应用，所有参数和返回值的实际类型都必须一样。

宏 `FP_FAST_FMA_HALF` 用来指示，对于半精度浮点数，函数族 `fma()` 是否比直接编码快。如果定义了宏 `FP_FAST_FMA_HALF`，则表示，相比对 `half` 操作数的乘和加，函数 `fma()` 执行的一样快，甚至更快。

9.10.3 公共函数⁴²

对表 6.11 中所定义的内建公共函数做了扩展，相应版本的函数可以将 `half` 和 `half{2|4|8|16}` 作为参数和返回值。现在 `gentype` 也包括 `half`、`half2`、`half4`、`half8` 和 `half16`。

函数	描述
<code>gentype clamp(gentype x,</code> <code> gentype minval,</code> <code> gentype maxval)</code> <code>gentype clamp(gentype x,</code> <code> half minval,</code> <code> half maxval)</code>	返回 $\min(\max(x, \text{minval}), \text{maxval})$ 如果 $\text{minval} > \text{maxval}$ ，则结果未定义。
<code>gentype degrees(gentype radians)</code>	将 <code>radians</code> 转换成度数，即 $\left(\frac{180}{\pi}\right) * \text{radians}$
<code>gentype max(gentype x, gentype y)</code> <code>gentype max(gentype x, half y)</code>	结果为 $\begin{cases} y, & x < y \\ x, & x \geq y \end{cases}$ 如果 x 和 y 是无穷或 NaN，返回值未定义。
<code>gentype min(gentype x, gentype y)</code> <code>gentype min(gentype x, half y)</code>	结果为 $\begin{cases} y, & y < x \\ x, & y \geq x \end{cases}$ 如果 x 和 y 是无穷或 NaN，返回值未定义。
<code>gentype mix(gentype x,</code> <code> gentype y, gentype a)</code> <code>gentype mix(gentype x,</code> <code> gentype y, half a)</code>	返回 x 和 y 的线性混合，实现为： $x + (y - x) * a$ a 必须在范围 0.0 到 1.0 之内，否则，结果未定义。
<code>gentype radians(gentype degrees)</code>	将 <code>degrees</code> 转换为弧度，即 $\left(\frac{\pi}{180}\right) * \text{degrees}$

⁴² 函数 `mix` 和 `smoothstep` 在实现时可以使用收缩，如同 `mad` 或 `fma` 一样。

<code>gentype step(gentype <i>edge</i>, gentype <i>x</i>)</code> <code>gentype step(half <i>edge</i>, gentype <i>x</i>)</code>	结果为： $\begin{cases} 0.0, & x < edge \\ 1.0, & x \geq edge \end{cases}$
<code>gentype smoothstep(gentype <i>edge0</i>, gentype <i>edge1</i>, gentype <i>x</i>)</code> <code>gentype smoothstep(half <i>edge0</i>, half <i>edge1</i>, gentype <i>x</i>)</code>	如果 $x \leq edge0$ ，返回 0.0；如果 $x \geq edge1$ ，返回 1.0；如果 $edge0 < x < edge1$ ，会在 0 和 1 之间执行平滑 hermite 插值。如果你想要一个临界值函数平滑过渡，就可以使用这个函数。 这等价于： <pre>gentype t; t = clamp((x - edge0) / (edge1 - edge0), 0, 1); return t * t * (3 - 2 * t);</pre> 如果 $edge0 \geq edge1$ ，或者 x 、 $edge0$ 或 $edge1$ 是 NaN，则结果未定义。
<code>gentype sign(gentype <i>x</i>)</code>	结果为： $A = \pi \begin{cases} 1.0, & x > 0 \\ -0.0, & x = -0.0 \\ +0.0, & x = +0.0 \\ -1.0, & x < 0 \\ 0.0, & x = NaN \end{cases}$

9.10.4 几何函数⁴³

对表 6.12 中所定义的内建几何函数做了扩展，相应版本的函数可以将 `half` 和 `half{2|4}` 作为参数和返回值。现在 `gentype` 也包括 `half`、`half2` 和 `half4`。

函数	描述
<code>half4 cross(half4 <i>p0</i>, half4 <i>p1</i>)</code>	返回 $p0.xyz$ 和 $p1.xyz$ 的叉乘。所返回的 <code>half4</code> 中的组件 w 将会是 0.0。
<code>half dot(gentype <i>p0</i>, gentype <i>p1</i>)</code>	计算点乘。
<code>half distance(gentype <i>p0</i>, gentype <i>p1</i>)</code>	返回 $p0$ 和 $p1$ 的距离，同 <code>length(p0-p1)</code> 一样。
<code>half length(gentype <i>p</i>)</code>	返回矢量 p 的长度，即 $\sqrt{p.x^2 + p.y^2 + \dots}$
<code>gentype normalize(gentype <i>p</i>)</code>	返回一个与 p 的方向相同的矢量，但是长度为 1。

9.10.5 关系函数

对表 6.13 中所定义的内建关系函数做了扩展，相应版本的函数可以将 `half` 和 `half{2|4|8|16}` 作为参数。

关系运算符和判等运算符 (`<`、`<=`、`>`、`>=`、`!=`、`==`) 可以与矢量类型 `halfn` 一起

⁴³ 几何函数在实现时可以使用收缩，如同 `mad` 或 `fma` 一样。

使用，所产生的结果类型为 `shortn` (如节 6.3 中所描述的那样)。

函数	描述
<code>int isequal(half x, half y)</code> <code>shortn isequal(halfn x, halfn y)</code>	返回组件级的比较 $x == y$ 。
<code>int isnotequal(half x, half y)</code> <code>shortn isnotequal(halfn x, halfn y)</code>	返回组件级的比较 $x \neq y$ 。
<code>int isgreater(half x, half y)</code> <code>shortn isgreater(halfn x, halfn y)</code>	返回组件级的比较 $x > y$ 。
<code>int isgreaterequal(half x, half y)</code> <code>shortn isgreaterequal(halfn x, halfn y)</code>	返回组件级的比较 $x \geq y$ 。
<code>int isless(half x, half y)</code> <code>shortn isless(halfn x, halfn y)</code>	返回组件级的比较 $x < y$ 。
<code>int islessequal(half x, half y)</code> <code>shortn islessequal(halfn x, halfn y)</code>	返回组件级的比较 $x \leq y$ 。
<code>int islessgreater(half x, half y)</code> <code>shortn islessgreater(halfn x, halfn y)</code>	返回组件级的比较 $(x < y) (x > y)$ 。
<code>int isfinite(half)</code> <code>shortn isfinite(halfn)</code>	测试参数是否一个有限值。
<code>int isinf(half)</code> <code>shortn isinf(halfn)</code>	测试参数是否一个无穷值 (+ve 或 -ve)。
<code>int isnan(half)</code> <code>shortn isnan(halfn)</code>	测试参数是否 NaN。
<code>int isnormal(half)</code> <code>shortn isnormal(halfn)</code>	测试参数是否一个正常值。
<code>int isordered(half x, half y)</code> <code>shortn isordered(halfn x, halfn y)</code>	测试参数是否有序。参数为 x 和 y ，结果是 $isequal(x, x) \&\& isequal(y, y)$
<code>int isunordered(half x, half y)</code> <code>shortn isunordered(halfn x, halfn y)</code>	测试参数是否无序。参数为 x 和 y ，任何一个 NaN 则返回一个非零值，否则返回 0。
<code>int signbit(half)</code> <code>shortn signbit(halfn)</code>	测试符号位。对于此函数的标量版本，如果设置了参数的符号位，则返回 1，否则返回 0。而对于矢量版本的每个组件：如果设置了符号位则返回 -1，否则返回 0。
<code>halfn bitselect (halfn a, halfn b, halfn c)</code>	对于结果中的每个比特，如果 c 中相应比特是 0，则返回 a 中的相应比特，否则返回 b 中的相应比特。
<code>halfn select (halfn a, halfn b, shortn c)</code> <code>halfn select (halfn a, halfn b, ushortn c)</code>	对于每个组件，如果 c 中相应组件设置了 MSB，则返回 b 中相应组件，否则返回 a 中相应组件。

9.10.6 读写图像的函数

对表 6.16 中所定义的读写图像的函数做了扩展，支持类型为 `half` 的图像色彩值。

图像	描述
<p><code>half4 read_imageh (image2d_t image, sampler_t sampler, int2 coord)</code></p> <p><code>half4 read_imageh (image2d_t image, sampler_t sampler, float2 coord)</code></p>	<p>用坐标(x, y)来查找 <i>image</i> 中的元素。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为预定义的压缩格式之一或 <code>CL_UNORM_INT8</code> 或 <code>CL_UNORM_INT16</code> ,则 read_imageh 所返回的半浮点值在范围$[0.0 \dots 1.0]$内。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 <code>CL_SNORM_INT8</code> 或 <code>CL_SNORM_INT16</code> ,则 read_imageh 所返回的半浮点值在范围$[-1.0 \dots 1.0]$内。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 <code>CL_HALF_FLOAT</code> ,则 read_imageh 返回半浮点值。</p> <p>调用 read_imageh 时如果使用整数坐标 ,则采样器必须使用过滤模式 <code>CLK_FILTER_NEAREST</code> ,规格化坐标设置为 <code>CLK_NORMALIZED_COORDS_FALSE</code> ,寻址模式设置为 <code>CLK_ADDRESS_CLAMP_TO_EDGE</code>、<code>CLK_ADDRESS_CLAMP</code> 或 <code>CLK_ADDRESS_NONE</code> ;否则 ,返回值未定义。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 的值不是上面所说的那些 ,则 read_imageh 的返回值未定义。</p>
<p><code>void write_imageh (image2d_t image, int2 coord, half4 color)</code></p>	<p>将 <i>color</i> 的值写到 <i>image</i> 中坐标(x, y)处。在写入前会进行必要的数据格式变换。x & y 被看做非规范化坐标 ,且必须分别在范围$0 \dots image_width - 1$和 $0 \dots image_height - 1$内。</p> <p>只有创建图像对象时 <i>image_channel_data_type</i> 为 <code>CL_SNORM_INT8</code>、<code>CL_UNORM_INT8</code>、<code>CL_SNORM_INT16</code>、<code>CL_UNORM_INT16</code> 或 <code>CL_HALF_FLOAT</code> ,才能使用 write_imageh ;会将个通道数据从半浮点类型变换为实际的存储类型。</p> <p>如果创建 <i>image</i> 时的 <i>image_channel_data_type</i> 不是以上所列的值或者坐标(x, y)不在范围 $(0 \dots image_width - 1, 0 \dots image_height - 1)$内 ,则</p>

	结果未定义。
half4 read_imageh (image3d_t <i>image</i> , sampler_t <i>sampler</i> , int4 <i>coord</i>)	用坐标(<i>coord.x</i> , <i>coord.y</i> , <i>coord.z</i>)来查找 <i>image</i> 中的元素。 <i>coord.w</i> 则被忽略。
half4 read_imageh (image3d_t <i>image</i> , sampler_t <i>sampler</i> , half4 <i>coord</i>)	<p>如果创建图像对象时 <i>image_channel_data_type</i> 为 CL_UNORM_INT8 或 CL_UNORM_INT16 或预定义压缩格式之一, 则 read_imageh 所返回的半浮点值在范围[0.0 ... 1.0]内。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 CL_SNORM_INT8 或 CL_SNORM_INT16, 则 read_imageh 所返回的半浮点值在范围[-1.0 ... 1.0]内。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 为 CL_HALF_FLOAT, 则 read_imageh 返回半浮点值。</p> <p>调用 read_imageh 时如果使用整数坐标, 则采样器必须使用过滤模式 CLK_FILTER_NEAREST, 规范化坐标设置为 CLK_NORMALIZED_COORDS_FALSE, 寻址模式设置为 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE; 否则, 返回值未定义。</p> <p>如果创建图像对象时 <i>image_channel_data_type</i> 的值不是上面所说的那些, 则 read_imageh 的返回值未定义。</p>

如果支持扩展 **cl_khr_3d_image_writes** 则扩展 **cl_khr_fp16** 也支持下列函数。

图像	描述
void write_imageh (image3d_t <i>image</i> , int4 <i>coord</i> , half4 <i>color</i>)	<p>将 <i>color</i> 的值写到 <i>image</i> 中坐标(<i>x</i>, <i>y</i>, <i>z</i>)处。在写入前会进行必要的格式转换。<i>coord.x</i>、<i>coord.y</i> 和 <i>coord.z</i> 被看做非规范化坐标, 且必须分别在范围 0 ... <i>image_width</i> - 1、0 ... <i>image_height</i> - 1 和 0 ... <i>image_depth</i> - 1 内。</p> <p>只有创建图像对象时 <i>image_channel_data_type</i> 为 CL_SNORM_INT8、CL_UNORM_INT8、CL_SNORM_INT16、CL_UNORM_INT16 或 CL_HALF_FLOAT, 才能使用 write_imageh; 会将个通道数据从半浮点类型变换为实际的存储类型。</p> <p>如果创建 <i>image</i> 时的 <i>image_channel_data_type</i></p>

	不是以上所列的值或者坐标(x, y, z)不在范围 ($0 \dots image_width - 1, 0 \dots image_height - 1, 0 \dots image_depth - 1$)内, 则结果未定义。
--	--

9.10.7 加载和存储矢量数据的函数

对表 6.14 中所定义的矢量数据读写函数 (**vload n** 和**vstore n**) 做了扩展, 相应版本的函数可以读写标量或矢量 half 值。现在 `gentype` 也包括 half; 现在 `gentypen` 也包括 half2、half4、half8 和 half16。

函数	描述
<code>gentypen vloadn (size_t offset, const __global gentype *p)</code> <code>gentypen vloadn (size_t offset, const __local gentype *p)</code> <code>gentypen vloadn (size_t offset, const __constant gentype *p)</code> <code>gentypen vloadn (size_t offset, const __private gentype *p)</code>	返回从位置($p + (offset * n)$)读取的 sizeof(<code>gentypen</code>)字节的数据。对于所计算的地址 ($p + (offset * n)$), 必须按 16 比特对齐。
<code>void vstoren (gentypen data, size_t offset, __global gentype *p)</code> <code>void vstoren (gentypen data, size_t offset, __local gentype *p)</code> <code>void vstoren (gentypen data, size_t offset, __private gentype *p)</code>	将 <code>data</code> 中 sizeof(<code>gentypen</code>)字节的数据写入地址 ($p + (offset * n)$)中。对于所计算的地址 ($p + (offset * n)$), 必须按 16 比特对齐。

9.10.8 全局内存和局部内存间的异步拷贝, 以及预取

对表 6.19 中所定义的内建异步拷贝和预取函数做了扩展, 现在 `gentype` 也包括 half、half2、half4、half8 和 half16。

9.10.9 IEEE 754 一致性

作为对表 4.3 的补充, 下表中的表项允许应用使用 **clGetDeviceInfo** 查询一个支持半精度浮点数的 OpenCL 设备的配置信息。

Op-code	返回类型	描述
CL_DEVICE_HALF_FP_CONFIG	cl_device_fp_config	描述 OpenCL 设备的半精度浮点能力。这是一个位域, 可以是下列值的一个或多个: CL_FP_DENORM——支持去规格化数

		CL_FP_INF_NAN——支持 INF 和 NaN CL_FP_ROUND_TO_NEAREST——支持舍入模式：舍入到最近偶数 CL_FP_ROUND_TO_ZERO——支持向零舍入 CL_FP_ROUND_TO_INF——支持舍入模式：向+ve 和-ve 无穷舍入 CP_FP_FMA——支持 IEEE 754-2008 中的混合乘加 必须要支持的最小的半精度浮点能力是 CL_FP_ROUND_TO_ZERO CL_FP_ROUND_TO_INF CL_FP_INF_NAN。
--	--	---

9.10.10 相对误差即ULP

本节我们讨论定义为 ulp 的最大相对误差。对于半浮点类型上的加、减、乘、除、混合乘加等运算，要求使用当前舍入模式对其进行正确舍入。对于到半浮点格式的变换，必须使用运算符 `conver_`所指舍入模式或当前舍入模式（如果此运算符没有指定舍入模式），或者 C 风格的转换对齐进行正确舍入。对于半浮点格式到整型的变换，必须使用运算符 `conver_`所指舍入模式或向零舍入（如果此运算符没有指定舍入模式），或者 C 风格的转换对齐进行正确舍入。所有从半浮点到浮点的变换都是准确的。

对于这些运算，可以换成使用向零舍入模式进行正确舍入。整数和半浮点格式间的变换必须使用舍入到最近偶数进行正确舍入。

下表列出了半精度浮点算术运算的最小精确度，以 ULP 的形式给出。计算 ULP 值时所参考的是进行算数运算时具有无穷精度的结果。

函数	最小精确度——ULP值 ⁴⁴
$x + y$	正确舍入
$x - y$	正确舍入
$x * y$	正确舍入
$1.0/x$	正确舍入

⁴⁴ 对于不需要舍入的函数，所用的 ULP 值是 0。

x/y	正确舍入
<i>acos</i>	$\leq 2ulp$
<i>acospi</i>	$\leq 2ulp$
<i>asin</i>	$\leq 2ulp$
<i>asimpi</i>	$\leq 2ulp$
<i>atan</i>	$\leq 2ulp$
<i>atan2</i>	$\leq 2ulp$
<i>atanpi</i>	$\leq 2ulp$
<i>atan2pi</i>	$\leq 2ulp$
<i>acosh</i>	$\leq 2ulp$
<i>asinh</i>	$\leq 2ulp$
<i>atanh</i>	$\leq 2ulp$
<i>cbrt</i>	$\leq 2ulp$
<i>ceil</i>	正确舍入
<i>copysign</i>	0ulp
<i>cos</i>	$\leq 2ulp$
<i>cosh</i>	$\leq 2ulp$
<i>cospi</i>	$\leq 2ulp$
<i>erfc</i>	$\leq 4ulp$
<i>erf</i>	$\leq 4ulp$
<i>exp</i>	$\leq 2ulp$
<i>exp2</i>	$\leq 2ulp$
<i>exp10</i>	$\leq 2ulp$
<i>expm1</i>	$\leq 2ulp$
<i>fabs</i>	0ulp
<i>fdim</i>	正确舍入
<i>floor</i>	正确舍入
<i>fma</i>	正确舍入
<i>fmax</i>	0ulp
<i>fmin</i>	0ulp
<i>fmod</i>	0ulp
<i>fract</i>	$\leq 1ulp$
<i>frexp</i>	0ulp
<i>hypot</i>	$\leq 2ulp$
<i>ilogb</i>	0ulp
<i>ldexp</i>	正确舍入
<i>log</i>	$\leq 2ulp$
<i>log2</i>	$\leq 2ulp$
<i>log10</i>	$\leq 2ulp$
<i>log1p</i>	$\leq 2ulp$
<i>logb</i>	0ulp
<i>mad</i>	所允许的任何值 (无穷 ulp)
<i>modf</i>	0ulp
<i>nan</i>	0ulp
<i>nextafter</i>	0ulp
<i>pow(x,y)</i>	$\leq 4ulp$
<i>pown(x,y)</i>	$\leq 4ulp$
<i>powr(x,y)</i>	$\leq 4ulp$
<i>remainder</i>	0ulp
<i>remquo</i>	0ulp
<i>rint</i>	正确舍入
<i>rootn</i>	$\leq 4ulp$
<i>round</i>	正确舍入
<i>rsqrt</i>	$\leq 1ulp$
<i>sin</i>	$\leq 2ulp$
<i>sincos</i>	对于正弦值和余弦值: $\leq 2ulp$
<i>sinh</i>	$\leq 2ulp$

<i>sinpi</i>	$\leq 2ulp$
<i>sqrt</i>	正确舍入
<i>tan</i>	$\leq 2ulp$
<i>tanh</i>	$\leq 2ulp$
<i>tanpi</i>	$\leq 2ulp$
<i>tgamma</i>	$\leq 4ulp$
<i>trunc</i>	正确舍入

注意：

对于 half 标量或矢量数据类型，实现可能会先将其变换为单精度浮点数，然后执行单精度浮点运算。这种情况下，仅在存储时使用 half 标量或矢量类型。

9.11 由GL上下文或共享组创建CL上下文

9.11.1 概述

节 9.12 中定义了怎样与并行 OpenGL 实现中的材质和缓冲对象共享数据，但没有定义怎样建立 GL 上下文或共享组与 CL 上下文间的关联。此扩展中定义了一些可选特性，可以由创建 OpenGL 上下文的例程使用来完成这个关联工作。如果某个实现支持此扩展，那么 CL_PLATFORM_EXTENSIONS 或 CL_DEVICE_EXTENSIONS (参见表 4.3) 中应有字符串 cl_khr_gl_sharing。

9.11.2 新过程和新函数

```
cl_int clGetGLContextInfoKHR (
    const cl_context_properties *properties,
    cl_gl_context_info param_name,
    size_t param_value_size,
    void *param_value,
    size_t *param_value_size_ret);
```

9.11.3 新记号

对于 clCreateContext、clCreateContextFromType 和 clGetGLContextInfoKHR，如果 properties 中的 OpenGL 上下文或共享组对象无效，则会返回：

CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR	-1000
--	-------

clGetGLContextInfoKHR 的参数 param_name 接受下列值：

CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR	0x2006
CL_DEVICES_FOR_GL_CONTEXT_KHR	0x2007

clCreateContext 和 clCreateContextFromType 的参数 properties 中接受下列特

性名称：

CL_GL_CONTEXT_KHR	0x2008
CL_EGL_DISPLAY_KHR	0x2009
CL_GLX_DISPLAY_KHR	0x200A
CL_WGL_HDC_KHR	0x200B
CL_CGL_SHAREGROUP_KHR	0x200C

9.11.4 对第 4 章的补充

在节 4.3 中，对 **clCreateContext** 中参数 *properties* 的描述，由下列文字代替：

“*properties* 指向一个特性列表，此列表是一个数组，元素为 <特性名称, 值>，以零结尾。如果 *properties* 中没有指定某个特性，则使用此特性的默认值（由表 4.4 列出）（成为隐式指定）。如果 *properties* 是 NULL 或空（所指列表中的第一个值是非零），则所有特性都用默认值。”

特性控制着怎样与 OpenGL 缓冲对象、材质对象和渲染缓存对象共享 OpenCL 内存对象（参见节 9.12）。依赖特定平台 API（用来将 OpenGL 上下文绑定到窗口系统上），可以设置下列特性来标识一个 OpenGL 上下文：

- ✚ 如果支持 CGL 绑定 API，对于一个 CGL 共享组对象，应当为 CGLShareGroup 句柄设置特性 CL_CGL_SHAREGROUP_KHR。
- ✚ 如果支持 EGL 绑定 API，对于 OpenGL ES 或 OpenGL 上下文，应当为 EGLContext 句柄设置特性 CL_GL_CONTEXT_KHR，而对于用来创建 OpenGL ES 或 OpenGL 上下文的显示屏，应当为 EGLDisplay 句柄设置特性 CL_EGL_DISPLAY_KHR。
- ✚ 如果支持 GLX 绑定 API，对于 OpenGL 上下文，应当为 GLXContext 设置特性 CL_GL_CONTEXT_KHR；而对于用来创建 OpenGL 上下文的 X 窗口系统，应当为 Display 句柄设置特性 CL_GLX_DISPLAY_KHR。
- ✚ 如果支持 WGL 绑定 API，对于 OpenGL 上下文，应当为 HGLRC 句柄设置特性 CL_GL_CONTEXT_KHR；而对于用来创建 OpenGL 上下文的显示屏，应当为 HDC 句柄设置特性 CL_WGL_HDC_KHR。

在上下文中创建的内存对象可以由指定的 OpenGL 或 OpenGL ES 上下文（此上下文的共享列表中的任意 OpenGL 上下文均可，其共享方式遵循 GLX 1.4 和 EGL 1.4 规范、微软 Windows 上的 OpenGL 实现的 WGL 文档）共享，也可以由为 CGL 显式标识的 OpenGL 共享组共享。如果特性列表中没有指定任何 OpenGL 或 OpenGL ES 上下文或共享组，就不能共享内存对象，调用节 9.12 中的任意命令都将导致错误 CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR。

表 4.4 要添加以下内容：

特性名称	允许的值（默认值加粗表示）	描述
CL_GL_CONTEXT_KHR	0 、OpenGL 上下文句柄	OpenGL 上下文，OpenCL 上下文与其相关联。
CL_CGL_SHAREGROUP_KHR	0 、CGL 共享组句柄	CGL 共享组，OpenCL 上下文与其相关联。
CL_EGL_DISPLAY_KHR	EGL_NO_DISPLAY 、EGLDisplay 句柄	EGLDisplay，用来创建 OpenGL 上下文。
CL_GLX_DISPLAY_KHR	None 、X 句柄	X 显示屏，用来创建 OpenGL 上下文。
CL_WGL_HDC_KHR	0 、HDC 句柄	HDC，用来创建 OpenGL 上下文。

表 4.4 clCreatContext 所支持的属性列表

用下列文字代替 **clCreateContext** 所返回的第一个错误：

errcode_ret 返回 CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR，如果一个上下文是通过下列途径中的任意一个所指定：

- ✚ 通过设置特性 CL_GL_CONTEXT_KHR 和 CL_EGL_DISPLAY_KHR 为基于 EGL 的 OpenGL ES 或 OpenGL 实现指定一个上下文。
- ✚ 通过设置特性 CL_GL_CONTEXT_KHR 和 CL_GLX_DISPLAY_KHR 为基于 GLX 的 OpenGL 实现指定一个上下文。
- ✚ 通过设置特性 CL_GL_CONTEXT_KHR 和 CL_WGL_HDC_KHR 为基于 WGL 的 OpenGL 实现指定一个上下文。

且满足下列任一条件：

- ✚ 所指定的显示屏和上下文特性不能标识一个有效的 OpenGL 或 OpenGL ES 上下文。
- ✚ 所指定的上下文不支持缓冲对象和渲染缓存对象。
- ✚ 所指定的上下文与所创建的 OpenCL 上下文不兼容（例如，存放于一个物理上不同的地址空间，如另一个硬件设备；或者由于实现所限不支持与 OpenCL 共享数据）。

errcode_ret 返回 CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR，如果通过设置属性 CL_CGL_SHAREGROUP_KHR，为基于 CGL 的 OpenGL 实现指定了一个共享组，且这个共享组不能标识一个有效的 CGL 共享组对象。

errcode_ret 返回 CL_INVALID_OPERATION，如果如上面所描述的那样指定了一个上下文，且满足下列任一条件：

- ✚ 为 CGL、EGL、GLX 或 WGL 中的一个指定了一个上下文或共享组对象 ,且 OpenGL 实现不支持那个窗口系统绑定 API。
- ✚ 特性 CL_CGL_SHAREGROUP_KHR、CL_EGL_DISPLAY_KHR、CL_GLX_DISPLAY_KHR 和 CL_WGL_HDC_KHR 中的多个被设置了默认值以外的值。
- ✚ 特性 CL_CGL_SHAREGROUP_KHR 和 CL_GL_CONTEXT_KHR 同时被设置了默认值以外的值。
- ✚ 参数 *devices* 中的任一设备不支持可以与 OpenGL 对象共享数据仓库的 OpenCL 对象 , 参见节 9.12。

errcode_ret 返回 CL_INVALID_VALUE 如果 *properties* 中的任一特性名称不在表 4.4 中。”

用下列文字代替对 **clCreateContextFromType** 的参数 *properties* 的描述 :

“*properties* 指向一个特性列表 , 其格式和内容同 **clCreateContext** 的参数 *properties* 一样。”

用上面所述 **clCreateContext** 的两个新错误代替 **clCreateContextFromType** 的第一个错误。

9.11.5 对节 9.12 的补充

添加新的一节 9.12.7 :

“可以查询与 OpenGL 上下文相对应的 OpenCL 设备。这样的设备可能不会一直存在 (例如 ,OpenGL 上下文所用 GPU 不支持 OpenCL 命令队列 ,但支持共享的 CL/GL 对象) , 即使存在 , 也可能随时间变化。如果存在这样的设备 , 对于在其对应的命令队列上获取和释放共享的 CL/GL 对象的操作 , 可能要比在其它设备所对应的命令队列上做同样操作快。要查询当前所对应的设备 , 可以使用函数 :

```
cl_int  clGetGLContextInfoKHR (
    const cl_context_properties *properties,
    cl_gl_context_info param_name,
    size_t param_value_size,
    void *param_value,
    size_t *param_value_size_ret)
```

properties 指向一个特性列表 , 其格式和内容与 **clCreateContext** 的参数 *properties* 一样。 *properties* 必须可以标识单个有效的 GL 上下文或 GL 共享组对象。

param_name 是一个常量 ,用来指定要查询的 GL 上下文信息 ,必须是表 9.7 中的值。

param_value 所指内存用来存储查询结果。如果是 NULL , 则忽略。

param_value_size 指定 *param_value* 所指内存的大小 ,单位字节。其值必须大于等于表 9.7 中所列返回类型的大小。

param_value_size_ret 返回所查数据的实际大小 ,单位字节。如果是 NULL ,则忽略。

表 9.7 可以用 clGetGLContextInfoKHR 来查询的 GL 上下文信息

param_name	返回类型	param_value 中所返回的信息
CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR	cl_device_id	返回当前与指定的 OpenGL 上下文相关联的 CL 设备。
CL_DEVICES_FOR_GL_CONTEXT_KHR	cl_device_id[]	列出可能与指定的 OpenGL 上下文相关联的所有 CL 设备。

如果执行成功 , **clGetGLContextInfoKHR** 会返回 CL_SUCCESS。如果不存在与 *param_name* 对应的设备 , 此调用不会失败 , 但是 *param_value_size_ret* 的值将是零。

clGetGLContextInfoKHR 会返回 CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR , 如果上下文是通过下列方式之一指定的 :

- ✚ 通过设置特性 CL_GL_CONTEXT_KHR 和 CL_EGL_DISPLAY_KHR 为基于 EGL 的 OpenGL ES 或 OpenGL 实现指定一个上下文。
- ✚ 通过设置特性 CL_GL_CONTEXT_KHR 和 CL_GLX_DISPLAY_KHR 为基于 GLX 的 OpenGL 实现指定一个上下文。
- ✚ 通过设置特性 CL_GL_CONTEXT_KHR 和 CL_WGL_HDC_KHR 为基于 WGL 的 OpenGL 实现指定一个上下文。

且满足下列任一条件 :

- ✚ 所指定的显示屏和上下文特性不能标识一个有效的 OpenGL 或 OpenGL ES 上下文。
- ✚ 所指定的上下文不支持缓冲对象和渲染缓存对象。
- ✚ 所指定的上下文与所创建的 OpenCL 上下文不兼容 (例如 , 存放于一个物理上不同的地址空间 , 如另一个硬件设备 ; 或者由于实现所限不支持与 OpenCL 共享数据)。

clGetGLContextInfoKHR 返回 `CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR` , 如果通过设置属性 `CL_CGL_SHAREGROUP_KHR` , 为基于 CGL 的 OpenGL 实现指定了一个共享组, 且这个共享组不能标识一个有效的 CGL 共享组对象。

clGetGLContextInfoKHR 返回 `CL_INVALID_OPERATION` , 如果如上面所描述的那样指定了一个上下文, 且满足下列任一条件:

- ✚ 为 CGL、EGL、GLX 或 WGL 中的一个指定了一个上下文或共享组对象, 且 OpenGL 实现不支持那个窗口系统绑定 API。
- ✚ 特性 `CL_CGL_SHAREGROUP_KHR`、`CL_EGL_DISPLAY_KHR`、`CL_GLX_DISPLAY_KHR` 和 `CL_WGL_HDC_KHR` 中的多个被设置了默认值以外的值。
- ✚ 特性 `CL_CGL_SHAREGROUP_KHR` 和 `CL_GL_CONTEXT_KHR` 同时被设置了默认值以外的值。
- ✚ 参数 *devices* 中的任一设备不支持可以与 OpenGL 对象共享数据仓库的 OpenCL 对象, 参见节 9.12。

clGetGLContextInfoKHR 返回 `CL_INVALID_VALUE` , 如果 *properties* 中的任一特性名称不在表 4.4 中。

另外, **clGetGLContextInfoKHR** 返回 `CL_INVALID_VALUE` , 如果 *param_name* 的值不在表 9.7 中, 或者 *param_value_size* 小于表 9.7 中返回类型的大小, 且 *param_value* 不是 NULL。”

9.11.6 问题

1. 在创建与其相关联的 OpenCL 上下文是, 怎样标识一个 OpenGL 上下文?

已解决: 通过使用一个特性对(显示屏、上下文句柄)来标识任一 OpenGL 或 OpenGL ES 上下文(与窗口系统绑定层 EGL、GLX 或 WGL 之一有关), 或使用共享组句柄来标识一个 CGL 共享组。如果指定了上下文, 对于调用 `clCreateContext` 的线程, 就不必这样做。

前面所建议的方法会使用布尔特性 `CL_USE_GL_CONTEXT_KHR` , 允许创建一个与当前所绑定的 OpenGL 上下文关联的上下文。这可能会被实现为一个独立的扩展, 在一些特定情况下, 如在绑定这个 GL 上下文的线程中执行, 可能允许效率更高的获取/释放。

2. 特性列表应是什么样的格式?

经过相当多的讨论, 我们认为特性列表可以以零结尾。此列表作为 `'cl_context_properties *properties'` 传入, 其中 `cl_context_properties` 的真实类型是

'`interp_t`'，见 `cl.h`。

这允许我们以非常有效的方式将宿主机 API 中的所有标量整型、指针和句柄的值编码进参数列表，同结构体和 EGL 特性列表非常相似。允许特性列表为 `NULL`。另一个与 EGL 类似的地方，任何没有显式传入的特性都会用其默认值。

根据从 EGL、GLX 和 WGL 中得到的经验，对于特性列表这种机制足够灵活和通用，可以满足管理调用的需要，如创建上下文。但它并不是完全通用（对浮点数和非标量特性值的编码不够直观），所以，对于 `getter/setter` 方法或不定长结构体的数组，建议使用其他途径，如不透明的特性列表。

3. 如果使用了所关联的其它上下文定义的资源，且这个上下文已经销毁，那么这时 OpenGL 或 OpenCL 上下文应当怎么办？

已解决：如节 9.12 所述，当创建 OpenCL 对象时，会放置一个对相应 GL 对象的数据仓库的引用。与那个数据仓库相对应的 GL 的名字可能会被删除，但只要引用它的 CL 对象不被销毁，数据仓库本身会一直存在。

然而，如果正在使用某个 CL 上下文，对于这个 CL 上下文所对应的共享组而言，销毁其中所有 GL 上下文所导致的结果依赖于具体实现，可能会导致程序终止。

4. 怎样与 D3D 共享？

D3D 与 OpenCL 间的共享会使用同样的特性列表机制，当然所用参数是不同的，将会是一个类似的并行 OpenCL 扩展。此扩展与本节的扩展会有一些交互，但能否创建一个 CL 上下文，使其同时共享 GL 和 D3D 对象，还不是很明确。

5. 什么情况会导致创建上下文时会由于共享的原因而失败？

已解决：已经列出了一些交叉平台的失败条件（GL 上下文或 CGL 共享组不存在、GL 上下文不支持节 9.12 中接口所需类型的 GL 对象、GL 上下文的实现不允许共享），还存在一些其它的原因可能导致失败，这些情况依赖于具体实现，如果以后发现了有这种情况，那么应该加到此扩展中。OpenCL 和 OpenGL 间的共享需要做驱动级别的整合。

6. `clEnqueueAcquire/ReleaseGLObjects` 可以放到什么样的命令队列中？

已解决：所有命令队列。在创建上下文时会强加此限制，而且如果所传入的任一设备不支持共享 CL/GL 对象，创建上下文会失败，并返回错误 `CL_INVALID_OPERATION`。

7. 应用怎样确定要将获取/释放操作放入哪个命令队列中？

已解决 : `clGetGLContextInfoKHR` 会返回当前与指定的 GL 上下文对应的 CL 设备 (典型的如显示屏, 此函数正在其上运行), 或一个设备列表, 其中存放的是指定的上下文可能会在其上运行的所有 CL 设备 (在多线程/“虚拟屏幕”环境中可能会非常有用)。此命令并没有简单的放在节 9.12 中, 因为它同指定一个 GL 上下文一样, 依赖于此扩展所引入的一个方法——属性列表。

如果没有返回设备, 意味着 GL 上下文所在 GPU 比较老, 不支持 OpenCL, 但是仍然可以在此 GPU 上运行的 GL 和其它地方运行的 CL 间共享对象。

8. 查找 `CL_DEVICES_FOR_GL_CONTEXT_KHR` 是什么意思?

已解决 : 曾经关联过某个 GL 上下文的所有 CL 设备。在一些平台上, 如 MacOS X, “虚拟屏幕”的概念允许多个 GPU 支持同一个虚拟显示屏; 其它窗口系统也可能实现了类似的功能, 如一个透明的异构多线程 X 服务器。因此这个查找的确切含义与正在使用的绑定层 API 有关。

9.12 与 OpenGL/OpenGL ES 缓冲对象、材质和渲染缓存对象共享内存对象

本节讨论一些 OpenCL 函数, 这些函数允许应用将 OpenGL 缓存、材质和渲染缓存对象作为 OpenCL 内存对象使用。这允许在 OpenCL 和 OpenGL 间高效共享数据。这些 OpenCL API 用来执行读写内存对象 (同样也是 OpenGL 对象) 的内核。

可以从 OpenGL 材质或渲染缓存对象创建 OpenCL 图像对象。可以从 OpenGL 缓冲对象创建 OpenCL 缓冲对象。

如果 OpenCL 上下文是由 OpenGL 共享组对象或上下文创建的, 而且只有这样, 那么可以由 OpenGL 对象创建 OpenCL 内存对象。OpenGL 共享组和上下文是使用特定平台 API (如 EGL、CGL、WGL 和 GLX) 创建的。在 MacOS X 上, 可以使用 OpenCL 平台扩展 **`cl_apple_gl_sharing`**, 由 OpenGL 共享组对象创建 OpenCL 上下文。在其它平台上, 包括 Microsoft Windows、Linux/Unix 等, 可以使用 Khronos 平台扩展 **`cl_khr_gl_sharing`**, 由 OpenGL 上下文创建 OpenCL 上下文。请参考您的 OpenCL 实现的平台文档或访问 Khronos Registry (地址为 <http://www.khronos.org/registry/cl/>) 以获取进一步信息。

对于所支持的任何 OpenGL 上下文, 只要是在 GL 共享组对象、或者与 GL 上下文相关联的共享组内定义的, 且 CL 上下文是由其创建的, 那么这些上下文就可以被共享; 默认的 OpenGL 对象 (即命名为零的对象) 例外, 这种对象不能被共享。

9.12.1 共享对象的生命周期

由 OpenGL 对象创建的 OpenCL 内存对象（下文中称作“共享的 CL/GL 对象”）会一直存在，直到相应的 GL 对象被删除。如果通过 GL API（如 **glDeleteBuffers**、**glDeleteTextures** 或 **glDeleteRenderbuffers**）删除 GL 对象，后面再使用 CL 缓冲对象或图像对象会导致未定义的行为，包括但不限于可能的 CL 错误和数据讹误，但不会导致程序终止。

CL 上下文和相应的命令队列依赖于 GL 共享组对象或与 GL 上下文相关联的共享组（CL 上下文由其创建）的存在。如果 GL 共享组对象或共享组中的所有 GL 上下文都被销毁了，只要使用 CL 上下文或命令队列就会导致未定义的行为，而且可能导致程序终止。在销毁相应的 GL 共享组或上下文之前，应用应当先销毁 CL 命令队列和 CL 上下文。

9.12.2 CL缓冲对象→GL缓冲对象

函数

```
cl_mem clCreateFromGLBuffer (cl_context context,
                             cl_mem_flags flags,
                             GLuint bufobj,
                             cl_int *errcode_ret)
```

可以由 OpenGL 缓冲对象创建 OpenCL 缓冲对象。

context 是由 OpenGL 上下文创建的 OpenCL 上下文。

flags 是位域，用来指定使用信息。请参考表 5.3。只能使用 CL_MEM_READ_ONLY、CL_MEM_WRITE_ONLY 和 CL_MEM_READ_WRITE。

bufobj 是一个 GL 缓冲对象的名字。此对象的数据仓库必须是之前通过调用 **glBufferData** 创建的，尽管其内容不需要初始化。数据仓库的大小将被用来决定 CL 缓冲对象的大小。

errcode_ret 会返回相应的错误码，下面有所描述。如果 *errcode_ret* 是 NULL，则不会返回错误码。

如果成功创建了缓冲对象，则 **clCreateFromGLBuffer** 会返回此对象，且 *errcode_ret* 被置为 CL_SUCCESS。否则，返回 NULL，且 *errcode_ret* 是下列错误值之一：

- ✚ CL_INVALID_CONTEXT，如果 *context* 无效或不是由 GL 上下文创建的。
- ✚ CL_INVALID_VALUE，如果 *flags* 的值无效。

- ✚ CL_INVALID_GL_OBJECT, 如果 *bufobj* 不是 GL 缓冲对象或虽然是 GL 缓冲对象但没有数据仓库。
- ✚ CL_OUT_OF_HOST_MEMORY, 如果为宿主机上的 OpenCL 实现分配其所需资源时失败。

在调用 **clCreateFromGLBuffer** 时, GL 缓冲对象数据仓库的大小将被用作所返回缓冲对象的大小。如果存在一个相应的 CL 缓冲对象, 且通过 GL API (如 **glBufferData**) 修改了 GL 缓冲对象的状态, 那么后面再使用 CL 缓冲对象会导致未定义的行为。

可以用函数 **clRetainMemObject** 和 **clReleaseMemObject** 来保留和释放缓冲对象。

9.12.3 CL图像对象→GL材质

函数

```
cl_mem clCreateFromGLTexture2D (cl_context context,
                                cl_mem_flags flags,
                                GLenum texture_target,
                                GLint miplevel,
                                GLuint texture,
                                cl_int *errcode_ret)
```

可以由 OpenGL 2D 材质对象或 OpenGL 立体贴图 (cubemap) 材质对象的单面来创建 OpenCL 2D 图像对象。

context 是由 OpenGL 上下文创建的 OpenCL 上下文。

flags 是位域, 用来指定使用信息。请参考表 5.3。只能使用 CL_MEM_READ_ONLY、CL_MEM_WRITE_ONLY 和 CL_MEM_READ_WRITE。

texture_target 必须是下列值之一: GL_TEXTURE_2D、GL_TEXTURE_CUBE_MAP_POSITIVE_X、GL_TEXTURE_CUBE_MAP_POSITIVE_Y、GL_TEXTURE_CUBE_MAP_POSITIVE_Z、GL_TEXTURE_CUBE_MAP_NEGATIVE_X、GL_TEXTURE_CUBE_MAP_NEGATIVE_Y、GL_TEXTURE_CUBE_MAP_NEGATIVE_Z 或 GL_TEXTURE_RECTANGLE⁴⁵。仅用来定义 *texture* 的图像类型。此参数不会造成对 GL 材质对象的引用, 当然也没有这个意思。

⁴⁵ 需要 OpenGL 3.1。或者, 如果支持 OpenGL 扩展 GL_ARB_texture_rectangle, 也可以用 GL_TEXTURE_RECTANGLE_ARB。


```
GLint miplevel,
GLuint texture,
cl_int *errcode_ret)
```

可以由 OpenGL 3D 材质对象创建 OpenCL 3D 图像对象。

context 是由 OpenGL 上下文创建的 OpenCL 上下文。

flags 是位域，用来指定使用信息。请参考表 5.3。只能使用 CL_MEM_READ_ONLY、CL_MEM_WRITE_ONLY 和 CL_MEM_READ_WRITE。

texture_target 必须是 GL_TEXTURE_3D。仅用来定义 *texture* 的图像类型。此参数不会造成对 GL 材质对象的引用，当然也没有这个意思。

miplevel 是所使用的 mipmap 级别。

texture 是 GL 3D 材质对象的名字。就材质完备性的每个 OpenGL 规则而言，此材质对象必须是完整材质。OpenGL 为材质的特定 *miplevel* 所定义的 *texture* 的格式和维度会被用来创建 3D 图像对象。只有当 GL 材质对象的内部格式可以映射到恰当的图像通道次序和数据类型（见表 5.4 和表 5.5）时，才能用来创建 3D 图像对象。

errcode_ret 会返回相应的错误码，下面有所描述。如果 *errcode_ret* 是 NULL，则不会返回错误码。

如果成功创建了图像对象，则 **clCreateFromGLTexture3D** 会返回此对象，且 *errcode_ret* 被置为 CL_SUCCESS。否则，返回 NULL，且 *errcode_ret* 是下列错误值之一：

- ✚ CL_INVALID_CONTEXT，如果 *context* 无效或不是由 GL 上下文创建的。
- ✚ CL_INVALID_VALUE，如果 *flags* 的值无效，或者 *texture_target* 的值无效。
- ✚ CL_INVALID_MIPLEVEL，如果 *miplevel*（对于 OpenGL 实现）小于 *level_base* 的值或者（对于 OpenGL ES 实现）是零；或者（对于 OpenGL 和 OpenGL ES）比 *q* 的值大。材质的 *level_base* 和 *q* 是在 OpenGL 2.1 规范的节 3.8.10（材质完备性）和 OpenGL ES 2.0 的节 3.7.10 中定义的。
- ✚ CL_INVALID_MIPLEVEL，如果 *miplevel* 的值大于零，且 OpenGL 实现不支持由非零的 mipmap 级别创建。
- ✚ CL_INVALID_GL_OBJECT，如果 *texture* 的类型与 *texture_target* 不匹配，如果 *texture* 的 *miplevel* 未定义，或者 *miplevel* 的宽或高是零。
- ✚ CL_INVALID_IMAGE_FORMAT_DESCRIPTOR，如果 OpenGL 材质的内部格式不能映射到所支持的 OpenCL 图像格式上。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主机上的 OpenCL 实现分配其所需资源时失败。

在调用 `clCreateFromGLBuffer` 时，GL 缓冲对象数据仓库的大小将被用作所返回缓冲对象的大小。如果存在一个相应的 CL 图像对象，且通过 GL API（如 `glTexImage2D`、`glTexImage3D` 或修改了材质参数 `GL_TEXTURE_BASE_LEVEL` 或 `GL_TEXTURE_MAX_LEVEL`）修改了 GL 材质对象的状态，那么后面再使用 CL 图像对象会导致未定义的行为。

可以用函数 `clRetainMemObject` 和 `clReleaseMemObject` 来保留和释放图像对象。

9.12.3.1 OpenGL和OpenCL图像格式对应关系

表 9.8 列出了 GL 材质内部格式和 CL 图像格式的对应关系。如果 OpenGL 成功创建了 GL 材质对象，且其内部格式是表 9.8 中的，那么就可以保证能将其映射到表中相应的 CL 图像格式上。如果材质对象是其它内部格式，那么可能（但不保证）映射到 CL 图像格式上；如果存在这样的映射，那么保证会保留所有色彩组件、数据类型，至少 OpenGL 为此格式实际分配的比特数/组件。

表 9.8 GL 内部格式和 CL 图像格式的映射

GL 内部格式	CL 图像格式（通道次序、通道数据类型）
GL_RGBA8	CL_RGBA、CL_UNORM_INT8 或 CL_BGRA、CL_UNORM_INT8
GL_RGBA16	CL_RGBA、CL_UNORM_INT16
GL_RGBA8I、GL_RGBA8I_EXT	CL_RGBA、CL_SIGNED_INT8
GL_RGBA16I、GL_RGBA16I_EXT	CL_RGBA、CL_SIGNED_INT16
GL_RGBA32I、GL_RGBA32I_EXT	CL_RGBA、CL_SIGNED_INT32
GL_RGBA8UI、GL_RGBA8UI_EXT	CL_RGBA、CL_UNSIGNED_INT8
GL_RGBA16UI、GL_RGBA16UI_EXT	CL_RGBA、CL_UNSIGNED_INT16
GL_RGBA32UI、GL_RGBA32UI_EXT	CL_RGBA、CL_UNSIGNED_INT32
GL_RGBA16F、GL_RGBA16F_ARB	CL_RGBA、CL_HALF_FLOAT
GL_RGBA32F、GL_RGBA32F_ARB	CL_RGBA、CL_FLOAT

9.12.4 CL图像对象→GL渲染缓存

函数

```
cl_mem clCreateFromGLRenderbuffer (cl_context context,
                                   cl_mem_flags flags,
                                   GLuint renderbuffer,
```

```
cl_int *errcode_ret)
```

可以由 OpenGL 渲染缓存创建 OpenCL 2D 图像对象。

context 是由 OpenGL 上下文创建的 OpenCL 上下文。

flags 是位域，用来指定使用信息。请参考表 5.3。只能使用 CL_MEM_READ_ONLY、CL_MEM_WRITE_ONLY 和 CL_MEM_READ_WRITE。

renderbuffer 是 GL 渲染缓存的名字。在创建图像对象前必须指定渲染缓存存储空间。OpenGL 所定义的 *renderbuffer* 的格式和维度会被用来创建 2D 图像对象。只有当 GL 渲染缓存的内部格式可以映射到恰当的图像通道次序和数据类型（见表 5.4 和表 5.5）时，才能用来创建 2D 图像对象。

errcode_ret 会返回相应的错误码，下面有所描述。如果 *errcode_ret* 是 NULL，则不会返回错误码。

如果成功创建了图像对象，则 **clCreateFromGLRenderBuffer** 会返回此对象，且 *errcode_ret* 被置为 CL_SUCCESS。否则，返回 NULL，且 *errcode_ret* 是下列错误值之一：

- ✚ CL_INVALID_CONTEXT，如果 *context* 无效或不是由 GL 上下文创建的。
- ✚ CL_INVALID_VALUE，如果 *flags* 的值无效。
- ✚ CL_INVALID_GL_OBJECT，如果 *renderbuffer* 不是 GL 渲染缓存对象或者 *renderbuffer* 的宽或高是零。
- ✚ CL_INVALID_IMAGE_FORMAT_DESCRIPTOR，如果 OpenGL 材质的内部格式不能映射到所支持的 OpenCL 图像格式上。
- ✚ CL_INVALID_OPERATION，如果 *renderbuffer* 是一个多重采样 (multi-sample) 的渲染缓存对象。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主机上的 OpenCL 实现分配其所需资源时失败。

如果存在一个相应的 CL 图像对象，且通过 GL API 修改了 GL 渲染缓存对象的状态（即用 GL API，如 **glRenderbufferStorage**，改变 GL 渲染缓存的维度或格式），那么后面再使用 CL 图像对象会导致未定义的行为。

可以用函数 **clRetainMemObject** 和 **clReleaseMemObject** 来保留和释放图像对象。

表 9.8 列出了 GL 材质内部格式和 CL 图像格式的对应关系。如果 OpenGL 成功创建了 GL 渲染缓存对象，且其内部格式是表 9.8 中的，那么就可以保证能将其映射到表中相应的 CL 图像格式上。如果渲染缓存对象是其它内部格式，那么可能（但不保证）映射到 CL 图像格式上；如果存在这样的映射，那么保证会保留所有色彩组件、数据类型，至少 OpenGL

为此格式实际分配的比特数/组件。

9.12.5 由CL内存对象查询GL对象信息

对于用来创建 OpenCL 内存对象的 OpenGL 对象，可以使用下面函数来查询其类型信息（即它是材质、渲染缓存，还是缓冲对象）。

```
cl_int  clGetGLObjectInfo (cl_mem memobj,
                           cl_gl_object_type *gl_object_type,
                           GLuint *gl_object_name)
```

gl_object_type 返回附到 *memobj* 上的 GL 对象的类型，可以是 CL_GL_OBJECT_BUFFER、CL_GL_OBJECT_TEXTURE2D、CL_GL_OBJECT_TEXTURE3D 或 CL_GL_OBJECT_RENDERBUFFER。如果是 NULL，则被忽略。

gl_object_name 返回用来创建 *memobj* 的 GL 对象的名字。如果是 NULL 则被忽略。

如果执行成功，则 **clGetGLObjectInfo** 会返回 CL_SUCCESS。如果 *memobj* 无效，则返回 CL_INVALID_MEM_OBJECT；如果没有 GL 对象与 *memobj* 相关联，则返回 CL_INVALID_GL_OBJECT。

函数

```
cl_int  clGetGLTextureInfo (cl_mem memobj,
                             cl_gl_texture_info param_name,
                             size_t param_value_size,
                             void *param_value,
                             size_t *param_value_size_ret)
```

会返回关于（与 *memobj* 相关联的）GL 材质对象的一些附加信息。

param_name 指定要查询什么附加信息。所支持的类型和 *param_value* 中所返回的信息如表 9.9 所示。

param_value 所指内存用来存放查询结果。如果是 NULL，则被忽略。

param_value_size 是 *param_value* 所指内存的大小，其值必须 ≥ 表 9.9 中返回类型的大小。

param_value_size_ret 是拷贝到 *param_value* 中数据的实际大小。如果是 NULL，则被忽略。

表 9.9 clGetGLTextureInfo 中的 param_name 支持列表

cl_gl_texture_info	返回类型	param_value 中返回的信息
CL_GL_TEXTURE_TARGET	GEnum	clCreateFromGLTexture2D 或 clCreateFromGLTexture3D 的参数 <i>texture_target</i> .
CL_GL_MIPMAP_LEVEL	GLint	clCreateFromGLTexture2D 或 clCreateFromGLTexture3D 的参数 <i>miplevel</i> .

如果执行成功，则 **clGetGLTextureInfo** 会返回 CL_SUCCESS。如果 *memobj* 无效，则返回 CL_INVALID_MEM_OBJECT；如果没有 GL 对象与 *memobj* 相关联，则返回 CL_INVALID_GL_OBJECT。如果 *param_name* 无效，或者 *param_value_size* 的值 < 表 9.9 中返回类型的大小，且 *param_value* 不是 NULL，或者 *param_value* 和 *param_value_size_ret* 都是 NULL，则返回 CL_INVALID_VALUE。

9.12.6 在GL和CL上下文间共享映射到GL对象的内存对象

函数

```
cl_int  clEnqueueAcquireGLObjects (cl_command_queue command_queue,
                                     cl_uint num_objects,
                                     const cl_mem *mem_objects,
                                     cl_uint num_events_in_wait_list,
                                     const cl_event *event_wait_list,
                                     cl_event *event)
```

用来获取由 OpenGL 对象创建的 OpenCL 内存对象。在使用这些对象之前，必须先获取它们。这些对象是由 OpenCL 上下文获取的，此上下文与 *command_queue* 相关联，因此所有命令队列，只要与此上下文相关联，就能使用这些对象。

command_queue 是一个有效的命令队列。对于用来创建与它相关联的 OpenCL 上下文的所有设备，都必须支持获取共享的 CL/GL 对象。在创建上下文时，此限制是强制性的。

num_objects 是 *mem_objects* 中要获取的内存对象的数目。

mem_objects 指向一个 CL 内存对象列表，这些对象与 GL 对象相对应。

event_wait_list 和 *num_events_in_wait_list* 指定在执行这个特殊命令前必须完成的事件。如果 *event_wait_list* 是 NULL，那么这个特殊命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL，则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL，*event_wait_list* 所指事件列表必须是有效的，且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。

event 会返回一个事件对象用来标识此读写命令，也可以用来查询或等待此命令的完成。

event 也可以是 NULL，但这样一来，应用就不能查询或等待此命令的完成了。

如果执行成功，**clEnqueueAcquireGLObjects** 会返回 CL_SUCCESS。如果 *num_objects* 是 0 且 *mem_objects* 是 NULL，则此函数什么都不做并返回 CL_SUCCESS。否则，返回下列错误之一：

- ✚ CL_INVALID_VALUE，如果 *num_objects* 是零且 *mem_objects* 不是 NULL，或者 *num_objects*>0 且 *mem_objects* 是 NULL。
- ✚ CL_INVALID_MEM_OBJECT，如果 *mem_objects* 中的内存对象无效。
- ✚ CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT，如果 *command_queue* 所关联的上下文不是由 OpenGL 上下文创建的。
- ✚ CL_INVALID_GL_OBJECT，如果 *mem_objects* 中的内存对象不是由 GL 对象创建的。
- ✚ CL_INVALID_EVENT_WAIT_LIST，如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list*>0，或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0，或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_OUT_OF_HOST_MEMORY，如果为宿主主机上的 OpenCL 实现分配其所需资源时失败。

函数

```
cl_int  clEnqueueReleaseGLObjects (cl_command_queue command_queue,
                                   cl_uint num_objects,
                                   const cl_mem *mem_objects,
                                   cl_uint num_events_in_wait_list,
                                   const cl_event *event_wait_list,
                                   cl_event *event)
```

用来释放由 OpenGL 对象创建的 OpenCL 内存对象。在 OpenGL 使用这些对象之前，必须先释放它们。这些对象是由与 *command_queue* 相关联的 OpenCL 上下文释放的。

num_objects 是 *mem_objects* 中要释放的内存对象的数目。

mem_objects 指向一个 CL 内存对象列表，这些对象与 GL 对象相对应。

event_wait_list 和 *num_events_in_wait_list* 指定在执行这个特殊命令前必须完成的事件。如果 *event_wait_list* 是 NULL，那么这个特殊命令不用等待任何事件的完成。如果 *event_wait_list* 是 NULL，则 *num_events_in_wait_list* 必须是 0。如果 *event_wait_list* 不是 NULL，*event_wait_list* 所指事件列表必须是有效的，且 *num_events_in_wait_list* 必须大于 0。*event_wait_list* 中的事件充当同步点。

event 会返回一个事件对象用来标识此读写命令,也可以用来查询或等待此命令的完成。*event* 也可以是 NULL,但这样一来,应用就不能查询或等待此命令的完成了。

如果执行成功, **clEnqueueReleaseGLObjects** 会返回 CL_SUCCESS。如果 *num_objects* 是 0 且 *mem_objects* 是 NULL 则此函数什么都不做并返回 CL_SUCCESS。否则,返回下列错误之一:

- ✚ CL_INVALID_VALUE, 如果 *num_objects* 是零且 *mem_objects* 不是 NULL, 或者 *num_objects*>0 且 *mem_objects* 是 NULL。
- ✚ CL_INVALID_MEM_OBJECT, 如果 *mem_objects* 中的内存对象无效。
- ✚ CL_INVALID_COMMAND_QUEUE, 如果 *command_queue* 无效。
- ✚ CL_INVALID_CONTEXT, 如果 *command_queue* 所关联的上下文不是由 OpenGL 上下文创建的。
- ✚ CL_INVALID_GL_OBJECT, 如果 *mem_objects* 中的内存对象不是由 GL 对象创建的。
- ✚ CL_INVALID_EVENT_WAIT_LIST, 如果 *event_wait_list* 是 NULL 而且 *num_events_in_wait_list*>0, 或者 *event_wait_list* 不是 NULL 但 *num_events_in_wait_list* 是 0, 或者 *event_wait_list* 中的事件对象无效。
- ✚ CL_OUT_OF_HOST_MEMORY, 如果为宿主机上的 OpenCL 实现分配其所需资源时失败。

9.12.6.1 同步OpenCL和OpenGL对共享对象的访问

为保证数据完整性,应用要负责同步对共享的 CL/GL 对象的访问(使用各自的 API)。如果没有提供这样的同步,就可能导致竞态条件(race condition)和其它未定义的行为,包括不可在实现间移植。

在调用 **clEnqueueAcquireGLObjects** 之前,应用必须确保所有要访问 *mem_objects* 中对象的 GL 操作都已经完成。这可以通过以下方式达到,且是可移植的:在所有上下文上发起一个 **glFinish** 命令并等待其完成,此命令带有对这些对象的未决引用。实现完全有可能提供更有效的同步方法;例如,在一些平台上,可能调用 **glFlush** 就足够了,或者线程内的同步可能是隐式的,或者可能有一个厂商特有的扩展,可以在 GL 命令流中放置一个屏障并等待其完成。注意,在现阶段,除 **glFinish** 外,其它同步方法都是不可移植的。

类似,在调用 **clEnqueueReleaseGLObjects** 后,由应用确保,在所有要访问 *mem_objects* 中对象且未决的 OpenCL 操作完成后,后续引用这些对象的 GL 命令才会执行。这可以通过下列方法来达成,这些方法是可移植的:使用 **clEnqueueReleaseGLObjects** 所返回的事件对象调用 **clWaitForEvents**, 或者调用 **clFinish**。像上面一样,一些实现可能提供更有效的方法。

如果 CL 和 GL 上下文处于不同的线程，则应用要负责维护操作的正确顺序。

如果一个 GL 上下文被绑定到了一个线程上，而此线程不是调用 **clEnqueueReleaseGLObjects** 的线程，那么如果应用不执行额外的步骤，对于此上下文而言，对 `mem_objects` 中任意对象的改变可能是不可见的。对于一个 OpenGL 3.1（或更高版本）上下文，在 OpenGL 3.1 规范的附录 G（“共享的对象和多个上下文”）中描述了对它的需求。对于之前版本的 OpenGL，需求依赖于具体实现。

当一个 OpenGL 对象被 OpenCL 获取后，在被释放之前，如果试图访问此对象的数据仓库将会导致未定义行为。类似的，当一个共享的 CL/GL 对象被 OpenCL 命令队列获取前，或被释放后，OpenCL 试图访问它将导致未定义行为。

10 OpenCL 嵌入式简档

OpenCL 1.0 规范描述了对桌面平台的特性需求。本节描述 OpenCL 1.0 嵌入式简档，它是 OpenCL 1.0 规范的一个子集，主要用于手持设备和嵌入式平台。节 8 中所定义的可扩展对两种简档都有用。

OpenCL 1.0 嵌入式简档有下列限制：

1. 64 位整型，即不支持 `long`、`ulong` 包括相应的矢量数据类型以及 64 位整型之上的运算。
2. 对 3D 图像的支持是可选的。

如果 `CL_DEVICE_IMAGE3D_MAX_WIDTH`、`CL_DEVICE_IMAGE3D_MAX_HEIGHT` 和 `CL_DEVICE_IMAGE3D_MAX_DEPTH` 是零，调用嵌入式简档中的 **clCreateImage3D** 创建 3D 图像会失败。**clCreateImage3D** 的参数 `errcode_ret` 会返回 `CL_INVALID_OPERATION`。在内核中声明类型为 `image3d_t` 的变量会导致编译错误。

如果 `CL_DEVICE_IMAGE3D_MAX_WIDTH`、`CL_DEVICE_IMAGE3D_MAX_HEIGHT` 和 `CL_DEVICE_IMAGE3D_MAX_DEPTH` 大于零，则 OpenCL 嵌入式简档的此实现支持 3D 图像。**clCreateImage3D** 会像 OpenCL 规范中所定义的那样工作。在内核中可以使用数据类型 `image3d_t`。

3. 如果创建 2D 和 3D 图像时，`image_channel_data_type` 的值是 `CL_FLOAT` 或 `CL_HALF_FLOAT`，那么所用采样器的过滤模式必须是 `CL_FILTER_NEAREST`；而如果采样器的过滤模式是 `CL_FILTER_LINEAR`，那么 **read_image** 和

read_imageh⁴⁸的返回值未定义。

4. 对于 2D 和 3D 图像，所支持的采样器寻址模式是：CLK_ADDRESS_NONE、CLK_ADDRESS_REPEAT 和 CLK_ADDRESS_CLAMP_TO_EDGE。
5. 由 CL_DEVICE_SINGLE_FP_CONFIG 给出的必须要支持的最小单精度浮点能力，即 CL_FP_ROUND_TO_ZERO 或 CL_FP_ROUND_TO_NEAREST。如果支持 CL_FP_ROUND_TO_NEAREST，则默认的舍入模式是舍入到最近偶数；否则默认舍入模式是向零舍入。
6. 单精度基本浮点运算（加、减和乘）应当进行正确舍入。表 10.1 给出了除法和开方的精度。

如果 CL_DEVICE_SINGLE_FP_CONFIG 中没有设置 CL_FP_INF_NAN，并且加减乘除的一个操作数或正确舍入后的结果是 INF 或 NaN，结果的值依赖于具体实现。同样，如果一个或多个操作数是 NaN，则单精度比较运算（<、>、<=、>=、==、!=）所返回的结果也依赖于具体实现。

在所有情况中，变换（节 6.2 和 6.11.7）应当和 FULL_PROFILE 中一样进行正确的舍入，包括那些使用或产生 INF 或 NaN 的变换。内建数学函数（节 6.11.2）也应当与 FULL_PROFILE 中的表现一样，包括节 7.5.1 中的边界情况的行为，不过精度由表 10.1 给出。

注意：

如果加减乘的默认舍入模式是向零舍入，则 **fract**、**fma** 和 **fdim** 也会按向零舍入进行运算。

对于基本浮点运算，放宽了 IEEE 754 的需求，尽管非常不情愿，但是这样可以为那些硬件预算有严格限制的嵌入式设备提供更好的灵活性。

7. 当使用 **vstore_half** 的变体执行从 float 到 half 的变换，或使用 **vload_half** 的变体执行从 half 到 float 的变换时，所生成的数据类型为 half 的去规格化数可能会被刷新称零。请参考 6.1.1.1。
8. 对于从 CL_UNORM_INT8、CL_SNORM_INT8、CL_UNORM_INT16 和 CL_SNORM_INT16 到 float 的变换，在嵌入式简档中其精度是 $\leq 2\text{ulp}$ （节 8.3.1.1 中为 $\leq 1.5\text{ulp}$ ）。对于嵌入式简档，节 8.3.1.1 中所描述的异常情况和下面给出的异常情况都有效。

For CL_UNORM_INT8

⁴⁸ 如果支持扩展 cl_khr_fp16。


```

0 must convert to 0.0f and
255 must convert to 1.0f

For CL_UNORM_INT16
0 must convert to 0.0f and
65535 must convert to 1.0f

For CL_SNORM_INT8
-128 and -127 must convert to -1.0f,
0 must convert to 0.0f and
127 must convert to 1.0f

For CL_SNORM_INT16
-32768 and -32767 must convert to -1.0f,
0 must convert to 0.0f and
32767 must convert to 1.0f

```

表 10.1 列出了对于嵌入式简档，单精度浮点算术运算的最小准确度，以 ULP 的形式给出。计算 ULP 值时所参考的是进行算数运算时具有无穷精度的结果。

表 10.1 内建数学函数的 ULP 值

函数	最小精度——ULP值 ⁴⁹
$x + y$	正确舍入
$x - y$	正确舍入
$x * y$	正确舍入
$1.0/x$	$\leq 1ulp$
x/y	$\leq 3ulp$
$acos$	$\leq 4ulp$
$acospi$	$\leq 5ulp$
$asin$	$\leq 4ulp$
$asinpi$	$\leq 5ulp$
$atan$	$\leq 5ulp$
$atan2$	$\leq 6ulp$
$atanpi$	$\leq 5ulp$
$atan2pi$	$\leq 6ulp$
$acosh$	$\leq 4ulp$
$asinh$	$\leq 4ulp$
$atanh$	$\leq 5ulp$
$cbrt$	$\leq 4ulp$
$ceil$	正确舍入
$copysign$	$0ulp$
cos	$\leq 4ulp$
$cosh$	$\leq 4ulp$
$cospi$	$\leq 4ulp$
$erfc$	$\leq 16ulp$
erf	$\leq 16ulp$
exp	$\leq 4ulp$
$exp2$	$\leq 4ulp$
$exp10$	$\leq 4ulp$

⁴⁹ 对于不需要舍入的函数，所用的 ULP 值是 0。

<i>expm1</i>	$\leq 4ulp$
<i>fabs</i>	$0ulp$
<i>fdim</i>	正确舍入
<i>floor</i>	正确舍入
<i>fma</i>	正确舍入
<i>fmax</i>	$0ulp$
<i>fmin</i>	$0ulp$
<i>fmod</i>	$0ulp$
<i>fract</i>	正确舍入
<i>frexp</i>	$0ulp$
<i>hypot</i>	$\leq 4ulp$
<i>ilogb</i>	$0ulp$
<i>ldexp</i>	正确舍入
<i>log</i>	$\leq 4ulp$
<i>log2</i>	$\leq 4ulp$
<i>log10</i>	$\leq 4ulp$
<i>log1p</i>	$\leq 4ulp$
<i>logb</i>	$0ulp$
<i>mad</i>	所允许的任何值 (无穷 ulp)
<i>modf</i>	$0ulp$
<i>nan</i>	$0ulp$
<i>nextafter</i>	$0ulp$
<i>pow(x,y)</i>	$\leq 16ulp$
<i>pown(x,y)</i>	$\leq 16ulp$
<i>powr(x,y)</i>	$\leq 16ulp$
<i>remainder</i>	$0ulp$
<i>remquo</i>	$0ulp$
<i>rint</i>	正确舍入
<i>rootn</i>	$\leq 16ulp$
<i>round</i>	正确舍入
<i>rsqrt</i>	$\leq 4ulp$
<i>sin</i>	$\leq 4ulp$
<i>sincos</i>	对于正弦值和余弦值 : $\leq 4ulp$
<i>sinh</i>	$\leq 4ulp$
<i>sinpi</i>	$\leq 4ulp$
<i>sqrt</i>	$\leq 4ulp$
<i>tan</i>	$\leq 5ulp$
<i>tanh</i>	$\leq 5ulp$
<i>tanpi</i>	$\leq 6ulp$
<i>tgamma</i>	$\leq 16ulp$
<i>trunc</i>	正确舍入
<i>half_cos</i> <i>half_divide</i> <i>half_exp</i> <i>half_exp2</i> <i>half_exp10</i> <i>half_log</i> <i>half_log2</i> <i>half_log10</i> <i>half_powr</i> <i>half_recip</i> <i>half_rsqrt</i> <i>half_sin</i> <i>half_sqrt</i> <i>half_tan</i>	$\leq 8192ulp$

<i>native_cos</i> <i>native_divide</i> <i>native_exp</i> <i>native_exp2</i> <i>native_exp10</i> <i>native_log</i> <i>native_log2</i> <i>native_log10</i> <i>native_powr</i> <i>native_recip</i> <i>native_rsqr</i> <i>native_sin</i> <i>native_sqr</i> <i>native_tan</i>	依赖于具体实现
---	---------

宏 `__EMBEDDED_PROFILE__` 被加入到语言中（参见节 6.9）。对于实现了嵌入式简档的 OpenCL 设备，它是整型常量 1，否则未定义。

如果 OpenCL 实现仅支持嵌入式简档，则表 4.1 中定义的 `CL_PLATFORM_PROFILE` 返回字符串 `EMBEDDED_PROFILE`。

在嵌入式简档中，对表 4.3 中的最小最大值做了修改，新的值是：

cl_device_info	返回类型	描述
<code>CL_DEVICE_MAX_COMPUTE_UNITS</code>	unsigned int	OpenCL 设备上的并行计算核心的数目。最小值是 1。
<code>CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS</code>	unsigned int	全局和局部工作项 ID 的最大维度。最小值是 3。
<code>CL_DEVICE_MAX_WORK_GROUP_SIZE</code>	size_t	一个执行内核（使用数据并行执行模型）的工作组中所能存放的工作项的最大数目。 （参见 <code>clEnqueueNDRangeKernel</code> ）。最小值是 1。
<code>CL_DEVICE_MAX_MEM_ALLOC_SIZE</code>	unsigned long long	分配内存对象的最大字节数。最小值是 $\max(\text{CL_DEVICE_GLOBAL_MEM_SIZE}/4, 1*1024*1024)$ 。
<code>CL_DEVICE_MAX_READ_IMAGE_ARGS</code>	unsigned int	内核可以同时读取的图像对象的最大数目。 如果 <code>CL_DEVICE_IMAGE_SUPPORT</code> 是 <code>CL_TRUE</code> ，最小值是 8。
<code>CL_DEVICE_MAX_WRITE_IMAGE_ARGS</code>	unsigned int	内核可以同时写入的图像对象的最大数目。 如果

		CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，最小值是 1。
CL_DEVICE_IMAGE2D_MAX_WIDTH	size_t	2D 图像的宽度（单位：像素）。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，最小值是 2048。
CL_DEVICE_IMAGE2D_MAX_HEIGHT	size_t	2D 图像的高度（单位：像素）。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，最小值是 2048。
CL_DEVICE_IMAGE3D_MAX_WIDTH	size_t	3D 图像的宽度（单位：像素）。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，最小值是 0。
CL_DEVICE_IMAGE3D_MAX_HEIGHT	size_t	3D 图像的高度（单位：像素）。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，最小值是 0。
CL_DEVICE_IMAGE3D_MAX_DEPTH	size_t	3D 图像的深度（单位：像素）。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，最小值是 0。
CL_DEVICE_MAX_SAMPLERS	unsigned int	一个内核可以使用的采样器的最大数目。关于采样器的细节描述可参见节 6.11.8。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，则最小值是 8。
CL_DEVICE_MAX_PARAMETER_SIZE	size_t	传给内核的参数最大字节数。最小值是 256。
CL_DEVICE_SINGLE_FP_CONFIG	cl_device_fp_config	<p>描述设备的单精度浮点能力。这是一个位域，描述下列值中的一个或多个：</p> <p>CL_FP_DENORM – 支持去规格化数</p> <p>CL_FP_INF_NAN – 支持 INF 和 qNaN</p> <p>CL_FP_ROUND_TO_NEAREST – 支持舍入到最近偶数</p> <p>CL_FP_ROUND_TO_ZERO – 支持向零舍入</p> <p>CL_FP_ROUND_TO_INF – 支持向 +ve 和 -ve 无穷舍入</p> <p>CL_FP_FMA – 支持 IEEE 754-2008 混合乘加</p> <p>要求至少要支持的浮点能力是：CL_FP_ROUND_TO_ZERO 或 CL_FP_ROUND_TO_NEAREST。</p>

CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	unsigned long long	分配不变缓冲区的最大字节数。最小值是 1KB。
CL_DEVICE_MAX_CONSTANT_ARGS	unsigned int	一个内核中带有 <code>_constant</code> 限定符的参数最大数目。最小值是 4。
CL_DEVICE_LOCAL_MEM_SIZE	unsigned long long	局部内存范围。最小值是 1KB。
CL_DEVICE_EXECUTION_CAPABILITIES	cl_device_exec_capabilities	描述设备的执行能力。这是一个位域，有以下值： CL_EXEC_KERNEL – 此 OpenCL 设备可以执行 OpenCL 内核。 CL_EXEC_NATIVE_KERNEL – 此 OpenCL 设备可以执行原生内核。 其能力至少要为： CL_EXEC_KERNEL。
CL_DEVICE_QUEUE_PROPERTIES	cl_command_queue_properties	描述设备所支持的命令队列属性。这是一个位域，包含以下值： CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE CL_QUEUE_PROFILING_ENABLE 这些属性在表 5.1 中有所描述。 其能力至少要为： CL_QUEUE_PROFILING_ENABLE。

对于嵌入式简档，如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，那么 CL_DEVICE_MAX_READ_IMAGE_ARGS、CL_DEVICE_MAX_WRITE_IMAGE_ARGS、CL_DEVICE_IMAGE2D_MAX_WIDTH、CL_DEVICE_IMAGE2D_MAX_HEIGHT、CL_DEVICE_IMAGE3D_MAX_WIDTH、CL_DEVICE_IMAGE3D_MAX_HEIGHT、CL_DEVICE_IMAGE3D_MAX_DEPTH 和 CL_DEVICE_MAX_SAMPLERS 的值必须大于等于上面所列的最小值。另外，对于 OpenCL 嵌入式简档的实现，必须支持下面所列图像格式。

对于只读 2D 图像和可选的 3D 图像，至少要支持的图像格式是：

image_num_channels	image_channel_order	image_channel_data_type
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8

		CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
--	--	---

对于可写的 2D 图像，至少要支持的图像格式是：

image_num_channels	image_channel_order	image_channel_data_type
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT

11 参考文献

1. The ISO/IEC 9899:1999 "C" Language Specification.
2. The ISO/IEC JTC1 SC22 WG14 N1169 Specification.
3. The ANSI/IEEE Std 754-1985 and 754-2008 Specifications.
4. The AltiVec™ Technology Programming Interface Manual.
5. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugarman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan. *Brook for GPUs: Stream Computing on Graphics Hardware*
6. Ian Buck. *Brook Specification v0.2*.
<http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf>
7. *NVIDIA CUDA Programming Guide*.
<http://developer.nvidia.com/object/cuda.html>
8. *ATI CTM Guide – Technical Reference Manual*
http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf
9. *OpenMP Application Program Interface*.
<http://www.openmp.org/drupal/mp-documents/spec25.pdf>

10. *The OpenGL Specification and the OpenGL Shading Language Specification* <http://www.opengl.org/registry/>
11. *NESL – A nested data parallel language.*
<http://www.cs.cmu.edu/~scandal/nsl.html>
12. *On the definition of ulp* (x) by Jean-Michel Muller
<ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5504.pdf>
13. Explicit Memory Fences
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2262.html>
14. Lefohn, Kniss, Strzodka, Sengupta, Owens, "Glift: Generic, Efficient, Random-Access GPU Data Structures," ACM Transactions on Graphics, Jan. 2006. pp 60--99.
15. Pharr, Lefohn, Kolb, Lalonde, Foley, Berry, "Programmable Graphics---The Future of Interactive Rendering," Neoptica Whitepaper, Mar. 2007.
16. Jens Maurer, Michael Wong. Towards support for attributes in C++ (Revision 4). March 2008. Proposed to WG21 "Programming Language C++, Core Working Group" .
17. GCC Attribute Syntax.
<http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html>.

附录 A 共享

A.1 共享的OpenCL对象

本节将描述：对于一个宿主机进程所创建的多个命令队列，那些对象可以在它们之间共享。

在创建 OpenCL 内存对象、程序对象和内核对象时用了个上下文，可以由使用此上下文的多个命令队列共享。当一个命令入队时可以创建一个事件对象；此事件对象可以由使用同一个上下文的多个命令队列共享。

应用需要实现线程间的同步，从而当多个命令队列都想改变一个共享对象（如命令队列、内存对象、程序对象和内核对象）的状态时，能够保证这些改动的正确次序（应用认为正确的次序）。

一个命令队列要想改变一个内存对象的状态时，可以先将这些改动暂存起来。要在多个命令队列间同步这些改动，应用必须做到：

如果命令队列中的命令想修改一个内存对象的状态，应用必须做到：

- ✚ 为这条命令获取相应的事件对象。
- ✚ 调用 API `clFlush`（或 `clFinish`）以发出此命令队列中所有未解决的命令。

对于一个内存对象，如果命令队列想同步到其最近的状态，则必须等待相应的事件对象，这些事件对象表示修改此对象的命令。这样就能保证，在之前的命令队列中使用此对象的命令都完成之后，才会执行此命令队列中使用此对象的命令。

如果一个共享资源正由其它命令队列使用，而这时修改此共享资源，则其结果未定义。

A.2 多个宿主机线程

在 OpenCL 实现中，创建、保留和释放对象（如上下文、命令队列、程序对象、内核对象和内存对象）的 API 调用是线程安全（*thread-safe*）的。而将命令入队或改变 OpenCL 对象（如命令队列、内存对象、程序对象和内核对象）状态的 OpenCL API 调用不是线程安全的。

在宿主机处理器上运行的应用中，OpenCL 实现可以为一个给定的 OpenCL 上下文和多个 OpenCL 上下文创建多个命令队列。

附录 B 移植性

OpenCL 被设计为可以移植到其它架构和硬件设计上。OpenCL 的核心是基于 C99 的编程语言。浮点算术基于 IEEE 754 和 **IEEE 754-2008** 标准。内存对象、指针限定符和弱序内存 (weakly-ordered memory) 被设计来为 OpenCL 设备所实现的分离式内存架构提供最大兼容性。命令队列和隔层允许在宿主机和 OpenCL 设备间进行同步。OpenCL 的设计、能力和限制很好的反应了下面硬件的能力。

不幸的是，在大量领域中，硬件平台的特质可能允许它做一些其它平台做不到的工作。拜常驻 CPU 的操作系统所赐，在 CPU 上执行的内核可以调出系统服务，像 printf，但是在 GPU 上相同的调用则会失败，至少目前如此 (请参见节 6.8)。出于调试的目的，这些服务是非常有优势的，实现可以使用 OpenCL 扩展机制来实现这些服务。

同样的，计算架构的不同可能意味着，例如，对于一个特定的循环结构，在 CPU 上执行时速度是可以接受的，但是在 GPU 上的速度可能会非常差。CPU 一般都是为多线程任务时延敏感算法设计的，执行这些任务 CPU 工作的很好，而普通的 GPU 可能会遭遇非常长的时延。如果开发者对编写可移植代码感兴趣，那么他们可能会发现，有必要在多种硬件上测试其代码，以确保关键算法在多种硬件上都能很好的工作。我们建议增加工作项的数目。希望在未来岁月中积累经验，能够产生一种统一的实践经验，从而可以在多种计算设备上应用。

我们可能更关注字节序问题。由于最开始支持 OpenCL 的大多数设备都是小端模式，开发者需要确保其内核在大端设备和小端设备上都做了测试，从而保证兼容性。OpenCL C 编程语言支持字节序特性限定符，允许开发者为数据指定使用宿主机的字节序还是使用 OpenCL 设备的字节序。这允许 OpenCL 编译器对加载和存储指令做适当的字节序转换。

我们也描述一下字节序怎样导致意料之外的结果：

当大端矢量机器 (如 AltiVec、CELL SPE) 加载矢量时，会保留数据的次序。即每个元素内的字节序和元素间的次序都同内存中的一样。当小端矢量机器 (如 SSE) 加载矢量时，会保留寄存器 (完成所有工作的地方) 中数据的次序。

在矢量中，每个元素内的字节序和元素间的次序都会被保留。

内存：

```
uint4 a = 

|            |            |            |            |
|------------|------------|------------|------------|
| 0x00010203 | 0x04050607 | 0x08090A0B | 0x0C0D0E0F |
|------------|------------|------------|------------|


```

寄存器中 (大端模式)：

uint4 a =	0x00010203	0x04050607	0x08090A0B	0x0C0D0E0F
-----------	------------	------------	------------	------------

寄存器中 (小端模式):

uint4 a =	0x0F0E0D0C	0x0B0A0908	0x07060504	0x03020100
-----------	------------	------------	------------	------------

这允许小端机器使用一个矢量负载来加载小端数据, 无视矢量中每块数据的大小。对于 `uchar16` 或 `ulong2` 也是一样。当然, 众所周知, 小端机器实际上⁵⁰按反向字节序存储数据, 来补偿数组元素的小端存储格式。

内存 (大端模式):

uint4 a =	0x00010203	0x04050607	0x08090A0B	0x0C0D0E0F
-----------	------------	------------	------------	------------

内存 (小端模式):

uint4 a =	0x03020100	0x07060504	0x0B0A0908	0x0F0E0D0C
-----------	------------	------------	------------	------------

一旦将数据装入了矢量中, 以此结束:

寄存器中 (大端模式):

uint4 a =	0x00010203	0x04050607	0x08090A0B	0x0C0D0E0F
-----------	------------	------------	------------	------------

寄存器中 (小端模式):

uint4 a =	0x0C0D0E0F	0x08090A0B	0x04050607	0x00010203
-----------	------------	------------	------------	------------

即, 在校正每个元素内的字节序的过程中, 机器终止了翻转元素间的次序。`0x00010203` 将出现在大端矢量的左边, 小端矢量的右边。

通过根据内存中的次序为元素编号, OpenCL 为不同的架构提供了一个一致的编程模型。这就有了像 `even/odd` 和 `high/low` 的概念。一旦将数据加载到了寄存器中, 我们发现元素 0 在大端矢量的左边, 小端矢量的右边。

<pre>float x[4]; float4 v = vload4(0, x);</pre>
Big-endian: v contains { x[0], x[1], x[2], x[3] }
Little-endian: v contains { x[3], x[2], x[1], x[0] }

⁵⁰ 注意这里我们谈论的是编程模型。事实上, 小端系统可能选择其它方式, 如从“右边”对字节进行寻址, 或翻转字节中比特的“次序”。这些选择中的任何一个都意味着硬件无需交换。

编译器知道此交换和按此引用元素。只要我们通过数值索引如.s0123456789abcdef或描述符如.xyzw、.hi、.lo、.even和.odd来引用它们，则所有都是透明的。当将数据存储回内存中时，任何次序的反转都会被撤销。用大端编程模型工作时，开发者可以忽略矢量中的元素次序问题。此机制依赖于以下事实：我们可以信赖一致的元素编号。一旦我们改变编号系统，例如通过与变换无关的转换（使用as_typed）将一个矢量转换成另外一个矢量（大小相同，但元素个数不同），那么不同的实现会有不同的结果，这取决于系统是大端、小端或者根本就没有矢量单元。（因此，对于到元素数目不同的矢量的比特转换，其行为依赖于具体实现，参考节 6.2.4）

下面是一个例子：

<pre>float x[4] = { 0.0f, 1.0f, 2.0f, 3.0f }; float4 v = vload4(0, x); uint4 y = (uint4) v; // legal, portable ushort8 z = (ushort8) v; // legal, not portable // element size changed</pre>
<pre>Big-endian: v contains { 0.0f, 1.0f, 2.0f, 3.0f } y contains { 0x00000000, 0x3f800000, 0x40000000, 0x40400000 } z contains { 0x0000, 0x0000, 0x3f80, 0x0000, 0x4000, 0x0000, 0x4040, 0x0000 } z.z is 0x3f80</pre>
<pre>Little-endian: v contains { 3.0f, 2.0f, 1.0f, 0.0f } y contains { 0x40400000, 0x40000000, 0x3f800000, 0x00000000 } z contains { 0x4040, 0x0000, 0x4000, 0x0000, 0x3f80, 0x0000, 0x0000, 0x0000 } z.z is 0</pre>

这里，在大端矢量机器和小端矢量机器间，z.z 中的值是不同的。

要想做与变换无关的转换——改变元素数目，以移植性的名义，OpenCL 会认为这是非法的。然而，虽然 OpenCL 提供了一套通用的运算符（取自矢量机器），但对于所有 ISA 都会以一致的可移植的方式提供的东西，并不是所有的都能用这套指令来访问。为一些特殊目的，许多矢量 ISA 提供一些指令，这些指令能极大的加速一些运算，如 DCT、SAD 或 3D 几何。我们打算让 OpenCL 过于笨重以致难以上手，如果这样，即使经验丰富的开发人员在编写对时间有严格要求、性能敏感的算法时，也不能使其接近最好的性能。开发人员倾向于将移植性抛在脑后，在代码中使用特定平台的指令。有鉴于此，OpenCL 允许传统的矢量 C 语言编程扩展，如作为对 OpenCL 的扩展，在 OpenCL 中可以直接使用 AltiVec C 编程接口或 Intel C 编程接口（在 emmintrin.h 中可以找到）来操作 OpenCL 数据类型。由于这些接口依赖于以下功能可以正常运作：与变换无关的转换——改变元素数目，所以 OpenCL 也允许这些功能。

作为一个普遍规则，任何操作，只要操作的是段中的矢量类型且段的大小与矢量元素大

小不同，在其它字节序不同或矢量架构不同的硬件上，就有可能中止。

例子可能包括：

- ✚ 使用 `.even` 和 `.odd` 将两个 `uchar8` 组合成 `ushort8`（请使用 `upsample()`，参见节 6.11.3）。
- ✚ 任何改变矢量元素数目的比特转换。（新类型上的运算是不可移植的。）
- ✚ 所使用区块（`chunk`）的大小与元素大小不同，改变数据次序的搅拌操作。

可移植运算的例子：

- ✚ 使用 `.even` 和 `.odd` 将两个 `uint8` 组合成 `uchar8`。例如对左右两个音频流执行间插操作。
- ✚ 任何不改变矢量元素数目的比特转换（如 `(float4)uint4`——为浮点类型定义存储格式）。
- ✚ 对与矢量元素大小一样的元素进行的的搅拌操作。

OpenCL 对 C 增加了一些东西，让应用的行为比 C 更可靠。值得注意的是，有几种情况下，OpenCL 为一些运算定义了一些行为，而这些在 C99 中是没有定义的：

- ✚ OpenCL 提供运算符 `convert_`，可以在所有类型间进行变换。对于将浮点类型变换成整型时会发生什么，C99 中没有定义，且舍入后的值依赖于整型可表示的范围。当使用变换的变体 `_sat` 时，会将浮点值变换成最近的可表示的整型值。类似的，针对 NaN 会发生什么，OpenCL 也给出了建议。对于提供了硬件方式饱和变换的硬件制造商，OpenCL 运算符 `convert_` 的饱和版本和非饱和版本可能都会使用此硬件。对于非饱和变换，如果在舍入后浮点操作数的值不在可表示的整形范围之内，OpenCL 没有定义这时会发生什么。
- ✚ 在 IEEE 754 标准草案中，将类型 `half`、`float` 和 `double` 定义为 `binary16`、`binary32` 和 `binary64` 格式（在现有 IEEE 754 标准中后两者相同）。你可能依赖于这些类型中比特的站位和意义。
- ✚ OpenCL 定义了越界的移位的行为。移位运算，如果大于等于第一个操作数的比特数，则先对其取模（译者注：即循环移位）。例如，如果要将 `int4` 移 33 位，OpenCL 将按移 $33 \% 32 = 1$ 位来对待。
- ✚ 对于数学库函数，许多边界情况比 C99 中定义的更严格。请参见节 7.5。

附录 C 范例

附录 D 中的例子不作为官方 OpenCL 规范的一部分。这些例子仅用来说明怎样使用 OpenCL API 在设备上执行内核和执行写为 OpenCL 内核的通用算法（如矩阵转置和矩阵

降阶), 其性能可能不足以实用。

C.1 一个简单的OpenCL内核

此例说明了怎样编写内核来操作内存对象的单独数据元素, 这跟 GLSL/HLSL 中所允许的类似。此例中的函数会计算两个 float4 数组的点乘, 并将结果写入一个 float 数组。

OpenCL 内核源码是:

```
__kernel void
dot_product (__global const float4 *a,
              __global const float4 *b, __global float *c)
{
    int gid = get_global_id(0);

    c[gid] = dot(a[gid], b[gid]);
}
```

下列代码描述了 OpenCL 运行时调用应用创建相应的内存对象、创建程序对象并为内核加载上面的程序源码, 构建程序执行体、创建内核对象、加载相应的参数值, 并在 GPU 设备上执行点乘内核。

```
void
delete_memobjs(cl_mem *memobjs, int n)
{
    int i;
    for (i=0; i<n; i++)
        clReleaseMemObject(memobjs[i]);
}

int
exec_dot_product_kernel(const char *program_source,
                        int n, void *srcA, void *srcB, void *dst)
{
    cl_context      context;
    cl_command_queue cmd_queue;
    cl_device_id    *devices;
    cl_program      program;
    cl_kernel       kernel;
    cl_mem          memobjs[3];
    size_t          global_work_size[1];
    size_t          local_work_size[1];
    size_t          cb;
    cl_int          err;

    // create the OpenCL context on a GPU device
    context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU,
                                      NULL, NULL, NULL);

    if (context == (cl_context)0)
        return -1;

    // get the list of GPU devices associated with context
    clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
```

```

devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES,
                  cb, devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);
if (cmd_queue == (cl_command_queue)0)
{
    clReleaseContext(context);
    free(devices);
    return -1;
}
free(devices);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
                           CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                           sizeof(cl_float4) * n, srcA, NULL);
if (memobjs[0] == (cl_mem)0)
{
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

memobjs[1] = clCreateBuffer(context,
                           CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                           sizeof(cl_float4) * n, srcB, NULL);
if (memobjs[1] == (cl_mem)0)
{
    delete_memobjs(memobjs, 1);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

memobjs[2] = clCreateBuffer(context,
                           CL_MEM_READ_WRITE,
                           sizeof(cl_float) * n, NULL, NULL);
if (memobjs[2] == (cl_mem)0)
{
    delete_memobjs(memobjs, 2);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// create the program
program = clCreateProgramWithSource(context,
                                    1, (const char**)&program_source, NULL, NULL);
if (program == (cl_program)0)
{
    delete_memobjs(memobjs, 3);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

```

```

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
    delete_memobjs(memobjs, 3);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// create the kernel
kernel = clCreateKernel(program, "dot_product", NULL);
if (kernel == (cl_kernel)0)
{
    delete_memobjs(memobjs, 3);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// set the args values
err = clSetKernelArg(kernel, 0,
                      sizeof(cl_mem), (void *) &memobjs[0]);
err |= clSetKernelArg(kernel, 1,
                      sizeof(cl_mem), (void *) &memobjs[1]);
err |= clSetKernelArg(kernel, 2,
                      sizeof(cl_mem), (void *) &memobjs[2]);

if (err != CL_SUCCESS)
{
    delete_memobjs(memobjs, 3);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// set work-item dimensions
global_work_size[0] = n;
local_work_size[0] = 1;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
                              global_work_size, local_work_size,
                              0, NULL, NULL);

if (err != CL_SUCCESS)
{
    delete_memobjs(memobjs, 3);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

```



```

// read output image
err = clEnqueueReadBuffer(cmd_queue, memobjs[2], CL_TRUE,
                          0, n * sizeof(cl_float), dst,
                          0, NULL, NULL);

if (err != CL_SUCCESS)
{
    delete_memobjs(memobjs, 3);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmd_queue);
    clReleaseContext(context);
    return -1;
}

// release kernel, program, and memory objects
delete_memobjs(memobjs, 3);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_queue);
clReleaseContext(context);
return 0; // success...
}

```

C.2 矩阵转置

本节我们描述在支持专有内存的 OpenCL 设备（具有代表性的如 GPU）上，对于由 2 的 MxN 次幂个元素组成的矩阵，怎样有效的对其执行转置。

此例中，会给每个工作组分配 64 个工作项，每个工作项独立计算 32x2 小部分来填充 32x32 的子矩阵（多于 8 次迭代）。最终的 32x32 子矩阵在局部内存中进行转置，同时带有一列 padding 以避免 bank 冲突。在局部内存中所执行的转置允许读写全局内存。额外的一列 padding 用来抵偿写地址，所以他们不会与读请求发生冲突。

使用 32（或者任何奇数倍的 GROUP_DIMX=32）的 padding，确保在一些 OpenCL 设备上读写全局内存中的元素将会被抵消，且不会操作同一个内存 bank/channel。就对全局内存的写操作而言，这是非常重要的，由于以列为主的索引时不连续的，可能引发全局内存 bank 冲突。

对于以行为主的元素，对其连续索引的全局内存读请求不会冲突。

下面给出了矩阵转置的 OpenCL 内核代码。

```

#define PADDING      (32)
#define GROUP_DIMX   (32)
#define LOG_GROUP_DIMX (5)
#define GROUP_DIMY   (2)
#define WIDTH        (256)
#define HEIGHT       (4096)

__kernel void matrix_transpose(

```



```

input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;
tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

tile[local_input] = input[input_index];
local_input += local_input_stride;
input_index += global_input_stride;

barrier(CLK_LOCAL_MEM_FENCE);

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];
local_output += local_output_stride;
output_index += global_output_stride;

output[output_index] = tile[local_output];

```



```

const unsigned int gid = get_group_id(0);
const unsigned int gsize = get_num_groups(0);

const unsigned int gs2 = GROUP_SIZE * 2;
const size_t stride = gs2 * gsize;

shared[lid] = 0.0f;

size_t i = gid * gs2 + lid;
while (i < n)
{
    shared[lid] += input[i] + input[(i+GROUP_SIZE)];
    i += stride;
}
barrier(CLK_LOCAL_MEM_FENCE);

#if (GROUP_SIZE >= 512)
if (lid < 256)
    shared[lid] += shared[lid + 256];
barrier(CLK_LOCAL_MEM_FENCE);
#endif

#if (GROUP_SIZE >= 256)
if (lid < 128)
    shared[lid] += shared[lid + 128];
barrier(CLK_LOCAL_MEM_FENCE);
#endif

#if (GROUP_SIZE >= 128)
if (lid < 64)
    shared[lid] += shared[lid + 64];
barrier(CLK_LOCAL_MEM_FENCE);
#endif

    if (lid < 32)
    {
#if (GROUP_SIZE >= 64)
        shared[lid] += shared[lid + 32];
        barrier(CLK_LOCAL_MEM_FENCE);
#endif
    }

#if (GROUP_SIZE >= 32)
    shared[lid] += shared[lid + 16];
    barrier(CLK_LOCAL_MEM_FENCE);
#endif

#if (GROUP_SIZE >= 16)
    shared[lid] += shared[lid + 8];
    barrier(CLK_LOCAL_MEM_FENCE);
#endif

#if (GROUP_SIZE >= 8)
    shared[lid] += shared[lid + 4];
    barrier(CLK_LOCAL_MEM_FENCE);
#endif

#if (GROUP_SIZE >= 4)
    shared[lid] += shared[lid + 2];
    barrier(CLK_LOCAL_MEM_FENCE);
#endif

#if (GROUP_SIZE >= 2)

```

```

        shared[lid] += shared[lid + 1];
        barrier(CLK_LOCAL_MEM_FENCE);
    #endif
    }

    if (lid == 0)
        output[gid] = shared[0];
}

```

注意，上面例子要求 n 是 `GROUP_SIZE` 的倍数。

下列代码描述了 OpenCL 运行时调用应用创建相应的内存对象、创建程序对象并为内核加载上面的程序源码，构建程序执行体、创建内核对象、加载相应的参数值，并在 GPU 设备上执行降阶内核。

```

#include <libc.h>
#include <stdbool.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <OpenCL/opencl.h>
#include <math.h>

////////////////////////////////////

#define MAX_GROUPS      (64)
#define MAX_WORK_ITEMS (64)

static int count = 1024 * 1024;

////////////////////////////////////

static char *
load_program_source(const char *filename) {
    struct stat statbuf;
    FILE *fh;
    char *source;

    fh = fopen(filename, "r");
    if (fh == 0)
        return 0;

    stat(filename, &statbuf);
    source = (char *) malloc(statbuf.st_size + 1);
    fread(source, statbuf.st_size, 1, fh);
    source[statbuf.st_size] = '\0';

    return source;
}

////////////////////////////////////
////

float reduce_float(float *data, int size) {

```

```

    int i;
    float sum = data[0];
    float c = (float) 0.0f;
    for (i = 1; i < size; i++) {
        float y = data[i] - c;
        float t = sum + y;
        c = (t - sum) - y;
        sum = t;
    }
    return sum;
}

////////////////////////////////////
/////

void create_reduction_pass_counts(int count, int max_groups,
    int max_work_items, int *level_count, size_t **group_counts,
    size_t **work_item_counts, int **entry_counts) {
    int work_items = (count < max_work_items * 2) ? count / 2
        : max_work_items;

    if (count < 1)
        work_items = 1;

    int groups = count / (work_items * 2);
    groups = max_groups < groups ? max_groups : groups;

    int max_levels = 1;
    int s = groups;

    while (s > 1) {
        int work_items = (s < max_work_items * 2) ? s / 2
            : max_work_items;
        s = s / (work_items * 2);
        max_levels++;
    }

    *group_counts = (size_t*) malloc(max_levels * sizeof(size_t));
    *work_item_counts = (size_t*) malloc(
        max_levels * sizeof(size_t));
    *entry_counts = (int*) malloc(max_levels * sizeof(int));

    (*level_count) = max_levels;
    (*group_counts)[0] = groups;
    (*work_item_counts)[0] = work_items;
    (*entry_counts)[0] = count;

    s = groups;
    int level = 1;

    while (s > 1) {
        int work_items = (s < max_work_items * 2) ? s / 2
            : max_work_items;
        int groups = s / (work_items * 2);
        groups = (max_groups < groups) ? max_groups : groups;

        (*group_counts)[level] = groups;
        (*work_item_counts)[level] = work_items;
        (*entry_counts)[level] = s;
    }
}

```

```

        s = s / (work_items * 2);
        level++;
    }
}

////////////////////////////////////
////////////////////////////////////

int main(int argc, char **argv) {
    cl_int err;
    cl_device_id device_id;
    cl_command_queue commands;
    cl_context context;
    cl_mem output;
    cl_mem input;
    cl_mem partials;
    int level_count = 0;
    size_t* group_counts = 0;
    size_t* work_item_counts = 0;
    int* entry_counts = 0;
    int i;

    // Create some random input data on the host
    //
    float *float_data = (float*) malloc(count * sizeof(float));
    for (i = 0; i < count; i++) {
        float_data[i] = ((float) rand() / (float) RAND_MAX);
    }

    // Connect to a GPU compute device
    //
    err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1,
        &device_id, NULL);
    if (err != CL_SUCCESS) {
        printf("Error: Failed to create a device group!\n");
        return EXIT_FAILURE;
    }

    // Create a compute context
    //
    context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);
    if (!context) {
        printf("Error: Failed to create a compute context!\n");
        return EXIT_FAILURE;
    }

    // Create a command commands
    //
    commands = clCreateCommandQueue(context, device_id, 0, &err);
    if (!commands) {
        printf("Error: Failed to create a command commands!\n");
        return EXIT_FAILURE;
    }

    // Load the compute program from disk into a cstring buffer
    //
    const char* filename = "reduce_kernel.cl";

```



```

printf("Loading program '%s'...\n", filename);
char *source = load_program_source(filename);
if (!source) {
    printf("Error: Failed to load compute program from file!\n");
    return EXIT_FAILURE;
}

// Create the input buffer on the device
//
input = clCreateBuffer(context, CL_MEM_READ_WRITE,
                      sizeof(float) * count, NULL, NULL);
if (!input)
{
    printf ("Error: Failed to allocate input data buffer on
device!\n");
    return EXIT_FAILURE;
}

// Fill the input buffer with the host allocated random data
//
err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0,
                          sizeof(float) * count,
                          (void *)float_data, 0, NULL, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to write to input data buffer on
device!\n");
    return EXIT_FAILURE;
}

// Create an intermediate data buffer for intra-level results
//
partials = clCreateBuffer(context, CL_MEM_READ_WRITE,
                          sizeof(float) * count, NULL, NULL);
if (!partials)
{
    printf("Error: Failed to allocate partial sum buffer on
device!\n");
    return EXIT_FAILURE;
}

// Create the output buffer on the device
//
output = clCreateBuffer(context, CL_MEM_READ_WRITE,
                       sizeof(float) * count, NULL, NULL);
if (!output)
{
    printf("Error: Failed to allocate result buffer on device!\n");
    return EXIT_FAILURE;
}

// Determine the global and local dimensions for the execution
//
create_reduction_pass_counts(count, MAX_GROUPS, MAX_WORK_ITEMS,
                             &level_count, &group_counts,
                             &work_item_counts, &entry_counts);

// Create programs and kernels for each level of the reduction

```

```

//
cl_program *programs =
(cl_program*)malloc(level_count * sizeof(cl_program));

memset(programs, 0, level_count * sizeof(cl_program));

cl_kernel *kernels = (cl_kernel*)malloc(
                                level_count * sizeof(cl_kernel));
memset(kernels, 0, level_count * sizeof(cl_kernel));

for(i = 0; i < level_count; i++)
{
    char *block_source = malloc(strlen(source) + 1024);
    size_t length = strlen(source) + 1024;
    memset(block_source, 0, length);

    // Insert macro definitions to specialize the kernel to a
    // particular group size
    //
    const char define[] = "#define GROUP_SIZE";
    sprintf(block_source, "%s (%d) \n%s\n", define,
            (int)group_counts[i], source);

    // Create the compute program from the source buffer
    //
    programs[i] = clCreateProgramWithSource(context, 1,
                                            (const char **) & block_source,
                                            NULL, &err);

    if (!programs[i] || err != CL_SUCCESS)
    {
        printf("Error: Failed to create compute program!\n");
        return EXIT_FAILURE;
    }

    // Build the program executable
    //
    err = clBuildProgram(programs[i], 0, NULL, NULL, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];
        printf("Error: Failed to build program executable!\n");
        clGetProgramBuildInfo(programs[i], device_id,
                                CL_PROGRAM_BUILD_LOG,
                                sizeof(buffer), buffer, &len);
        printf("%s\n", buffer);
        return EXIT_FAILURE;
    }

    // Create the compute kernel from within the program
    //
    kernels[i] = clCreateKernel(programs[i], "reduce", &err);
    if (!kernels[i] || err != CL_SUCCESS)
    {
        printf("Error: Failed to create compute kernel!\n");
        return EXIT_FAILURE;
    }
}

```

```

    free(block_source);
}

// Do the reduction for each level
//
printf("Performing Reduction [%d]...\n", count);

cl_mem pass_swap = output;
cl_mem pass_input = output;
cl_mem pass_output = input;

for(i = 0; i < level_count; i++)
{
    size_t global = group_counts[i] * work_item_counts[i];
    size_t local = work_item_counts[i];
    unsigned int entries = entry_counts[i];

    printf("Pass[%4d] Global[%4d] Local[%4d] Groups[%4d]
WorkItems[%4d] Entries[%d]\n", i,
        (int)global, (int)local, (int)group_counts[i],
        (int)work_item_counts[i], entries);

    // Swap the inputs and outputs for each pass
    //
    pass_swap = pass_input;
    pass_input = pass_output;
    pass_output = pass_swap;

    err = CL_SUCCESS;
    err |= clSetKernelArg(kernels[i], 0,
        sizeof(cl_mem), &pass_output);
    err |= clSetKernelArg(kernels[i], 1,
        sizeof(cl_mem), &pass_input);
    err |= clSetKernelArg(kernels[i], 2,
        sizeof(float) * work_item_counts[i], NULL);
    err |= clSetKernelArg(kernels[i], 3,
        sizeof(int), &entries);
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to set kernel arguments!\n");
        return EXIT_FAILURE;
    }

    // After the first pass, use the partial sums for the
    // next input values
    //
    if(pass_input == input)
        pass_input = pass_output;

    err = CL_SUCCESS;
    err |= clEnqueueNDRangeKernel(commands, kernels[i], 1, NULL,
        &global, &local, 0, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to execute kernel!\n");
        return EXIT_FAILURE;
    }
}

```

```

    }

    // Read back the final sum that was computed on the device
    //
    float computed_result = 0.0f;
    err = clEnqueueReadBuffer(commands, pass_output, CL_TRUE, 0,
                              sizeof(float), &computed_result,
                              0, NULL, NULL);

    if (err)
    {
        printf("Error: Failed to read back results from the
device!\n");
        return EXIT_FAILURE;
    }

    // Do our own reduction to compare the results
    //
    float reference = reduce_float(float_data, count);

    // Verify the results are correct
    //
    float error = fabs(reference - computed_result);

    // Report any incorrect results
    //
    if (error > 1e-5)
    {
        printf("Reference %f != Device Result %f\n",
               reference, computed_result);
        printf("Error: Incorrect results obtained! Max error = %f\n",
               error);
        return EXIT_FAILURE;
    }
    else
    {
        printf("Results Validated!\n");
    }

    // Shutdown and cleanup
    //
    for(i = 0; i < level_count; i++)
    {
        clReleaseKernel(kernels[i]);
        clReleaseProgram(programs[i]);
    }

    clReleaseMemObject(input);
    clReleaseMemObject(output);
    clReleaseMemObject(partials);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);

    free(group_counts);
    free(work_item_counts);
    free(entry_counts);
    free(kernels);

    return 0;

```

}

设备：假快部