目录

圖

表

版權信息

致謝

OpenCL 規範中凝聚了許多人的貢獻，涉及桌面、手持以及嵌入式等多個計算機工業領域；下面列出了部分人員，以及當時所在的公司:

David Neto, Altera
Benedict Gaster, AMD
Bill Licea Kane, AMD
Ed Buckingham, AMD
Jan Civlin, AMD
Laurent Morichetti, AMD
Mark Fowler, AMD
Michael Houston, AMD
Michael Mantor, AMD
Norm Rubin, AMD
Ofer Rosenberg, AMD
Victor Odintsov, AMD
Aaftab Munshi, Apple
Anna Tikhonova, Apple
Abe Stephens, Apple
Bob Beretta, Apple
Geoff Stahl, Apple
Ian Ollmann, Apple
Inam Rahman, Apple
James Shearer, Apple
Jeremy Sandmel, Apple
John Stauffer, Apple
Kathleen Danielson, Apple
Michael Larson, Apple
MonPing Wang, Apple
Nate Begeman, Apple
Tanya Lattner, Apple
Travis Brown, Apple
Anton Lokhmotov, ARM
Dave Shreiner, ARM
Hedley Francis, ARM
Robert Elliott, ARM
Scott Moyers, ARM
Tom Olson, ARM
Alastair Donaldson, Codeplay
Andrew Richards, Codeplay
Stephen Frye, Electronic Arts
Eric Schenk, Electronic Arts
Brian Murray, Freescale
Brian Horton, IBM
Brian Watt, IBM
Dan Brokenshire, IBM
Gordon Fossum, IBM
Greg Bellows, IBM
Joaquin Madruga, IBM
Mark Nutter, IBM
Joe Molleson, Imagination Technologies
Jon Parr, Imagination Technologies
Robert Quill, Imagination Technologies
James McCarthy, Imagination Technologies
Aaron Lefohn, Intel
Adam Lake, Intel
Andrew Brownsword, 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
Michael McCool, Intel
Murali Sundaresan, Intel
Paul Lalonde, Intel
Stefanus Du Toit, Intel
Stephen Junkins, Intel
Tim Foley, Intel
Timothy Mattson, Intel
Yariv Aridor, Intel
Bill Bush, Kestrel Institute
Lindsay Errington, Kestrel Institute
Jon Leech, Khronos
Benjamin Bergen, Los Alamos National Laboratory
Marcus Daniels, Los Alamos National Laboratory
Michael Bourges Sevenier, Motorola
Jyrki Leskelä, Nokia
Jari Nikara, Nokia
Amit Rao, NVIDIA
Ashish Srivastava, NVIDIA
Bastiaan Aarts, NVIDIA
Chris Cameron, NVIDIA
Christopher Lamb, NVIDIA
Dibyapran Sanyal, NVIDIA
Guatam Chakrabarti, NVIDIA
Ian Buck, NVIDIA
Jason Sanders, NVIDIA
Jaydeep Marathe, NVIDIA
Jian-Zhong Wang, NVIDIA
Karthik Raghavan Ravi, NVIDIA
Kedar Patil, NVIDIA
Manjunath Kudlur, NVIDIA
Mark Harris, NVIDIA
Michael Gold, NVIDIA
Neil Trevett, NVIDIA
Rahul Joshi, NVIDIA
Richard Johnson, NVIDIA
Sean Lee, NVIDIA
Tushar Kashalikar, NVIDIA
Vinod Grover, NVIDIA
Xiangyun Kong, NVIDIA
Yogesh Kini, NVIDIA
Yuan Lin, NVIDIA
Alex Bourd, QUALCOMM
Andrzej Mamona, QUALCOMM
Chihong Zhang, QUALCOMM
David Garcia, QUALCOMM
David Ligon, QUALCOMM
Robert Simpson, QUALCOMM
Yanjun Zhang, S3 Graphics
Tasneem Brutch, Samsung
Thierry Lepley, STMicroelectronics
Alan Ward, Texas Instruments
Madhukar Budagavi, Texas Instruments
Brian Hutsell Vivante
Mike Cai, Vivante
Sumeet Kumar, Vivante

Henry Styles, Xilinx

第 1 章
簡介

在現代處理器架構中，將並行視為提高性能的重要途徑之一。有鑒於在固定功率內提升時鐘頻率面臨很大技術挑戰，目前 CPU 都通過增加核心數目來提高性能。而 GPU 原來只是具有特定功能的渲染設備，現在也變成了可編程的並行處理器。今天的計算機系統通常包含 CPU、GPU 和其他類型的處理器，且可以高度並行，這就引出了一個重要問題：怎樣讓軟件開發人員將這些異構處理平台充分利用起來。

多核 CPU 和 GPU 的傳統編程方法差異較大，因此為異構並行處理平台開發應用就成為一個比較棘手的問題。雖然基於 CPU 的並行編程模型通常是有標準的，但是一般都假設共享位址空間並且不包含矢量運算。而通用目的的 GPU 編程模型一般都具有複雜的內存分級尋址、矢量運算，但通常都是基於某個特定平台、供應商或硬件的。這些限制使得開發人員很難通過同一套多平台的源碼庫使用這些異構處理器（CPU、GPU 和其他類型的處理器）。更進一步，要讓軟件開發人員充分、高效的利用異構處理平台——從高性能計算服務器、桌面計算機系統，一直到手持設備——這些設備中有並行 CPU、GPU 和其他處理器（如 DSP 和 Cell/B.E. 處理器），組合方式有很多種。

OpenCL（Open Computing Language）是一種開放的免稅標準，用於在 CPU、GPU 和其他處理器上進行通用目的的並行編程，他使軟件開發人員可以以一種高效可移植的方式使用這些異構處理平台。

OpenCL 支持廣泛的應用，從嵌入式、消費級的軟件到 HPC 解決方案都涵蓋在內，而這是通過一個底層、高性能、可移植的抽象來完成的。通過創建一個高效、更接近於硬件的編程接口，OpenCL 會在並行計算生態系統中形成一個基礎層，此系統所包含的工具、中間件和應用都是跨平台的。有一種新興的交互式圖形應用，將通用的並行計算算法和圖形渲染管線結合在一起；OpenCL 特別適合於這種應用，在其中扮演的角色愈發重要。

OpenCL 由一系列 API 和一個跨平台的編程語言組成；其中 API 可以協調異構處理器間的並行計算；而編程語言則明確規定了計算環境。OpenCL 標準：

- 同時支持基於數據和基於任務的並行編程模型；
- 使用 ISO C99 的一個子集，並為並行做了擴展；
- 定義了與 IEEE 754 一致的數值需求；
- 為手持和嵌入式設備定義了一個配置檔案；
- 可以與 OpenGL、OpenGL ES 以及其他圖形 API 進行高效的互操作。

本文檔先對一些基本概念和 OpenCL 架構進行概述，接着再詳細描述其執行模型、內存模型以及對同步的支持。然後討論 OpenCL 平台和運行時 API，緊跟着是對 OpenCL C 編程語言的詳細描述，並就如何用 OpenCL 編程進行採樣計算給出了一些示例。此規範劃分為三部分：一是核心規格，所有符合 OpenCL 的實作都必須支持；二是手持/嵌入式規格，降低了對手持和嵌入式設備的要求；三是一些可選擴展，這些擴展可能會在後續修訂 OpenCL 規範時被納入核心規格。

第 2 章
術語

應用（Application）

運行在主機和 OpenCL 設備上的程式。

阻塞的入隊 API 調用（Blocking Enqueue API calls）

參見非阻塞的入隊 *API* 調用。

非阻塞的入隊 API 調用（Non-Blocking Enqueue API calls）

非阻塞的入隊 *API* 調用在命令隊列中放置一個命令後會立刻返回到主機中。而阻塞的入隊 *API* 調用則會等到命令結束後再返回。

屏障（Barrier）

有兩種屏障——命令隊列屏障和作業組屏障。

- 命令隊列屏障本身是一個命令，OpenCL API 提供有函式可以將其入隊。他可以保證在此之前入隊的所有命令都執行完畢後，在此之後入隊的命令才會開始執行。
- OpenCL C 編程語言中有一個內建的作業組屏障函式。內核在設備上運行時，可以利用此函式在同一作業組中的不同作業項間進行同步 (這些作業項正在執行此內核)。對這個作業組而言，其中任一作業項要想越過此屏障繼續執行，則他所包含的所有作業項都必須先執行此屏障。

緩衝對象（Buffer Object）

一種內存對象，其中存儲的是線性字節序列。內核在設備上運行時，可以通過指針來訪問他。主機可以調用 OpenCL API 來操控緩衝對象。緩衝對象封裝了以下資訊:

- 字節數;
- 一些屬性，用來描述使用情況以及分配自哪個區域;
- 緩衝數據。

內建內核（Built-in Kernel）

一種內核，可以在 OpenCL 設備上運行，也可以在具有固定功能的自定義設備上運行，甚至可以在固件中運行。應用可以查詢設備或自定義設備所支持的內建內核。程式對象可以包含用 OpenCL C 寫就的內核，也可以包含內建內核，但不能同時包含兩者。請參考內核和程式。

命令（Command）

即 OpenCL 運算，可以提交給命令隊列執行。例如，一些 OpenCL 命令可以在計算設備上執行內核、操控內存對象等等。

命令隊列（Command-queue）

這種對象中持有命令，而這些命令將在某個特定設備上執行。命令隊列是在上下文中的某個特定設備上創建的。命令按序入隊，但不一定按順序執行。參見順序執行和亂序執行。

命令隊列屏障（Command-queue Barrier）

參見屏障。

計算設備內存（Compute Device Memory）

附着到計算設備上的一塊或多塊內存。

計算器件（Compute Unit）

一個 OpenCL 设备中包含一個或多個計算器件。一個作業組只能在單個計算器件上執行。一個計算器件由一個或多個處理元件以及局部內存組成。計算器件也可能包含專用的材質濾波器件，此器件可以被他的處理元件訪問。

並發（Concurrency）

系統的一個屬性，這樣的系統中，多個任務可以同時保持活躍並取得進展。運行程式時要想並發執行，程式員必須找出問題中可以並發的部分，並在源碼中標示出來，用一個支持並發的符號進行開發。

不變內存（Constant Memory）

全局內存中的一個區域，在內核執行過程中保持不變。放入其中的內存對象都是由主機分配並初始化的。

上下文（Context）

內核在執行時所處的環境，同步和內存管理就是在此概念上定義的。上下文包括一組设备、這些设备可以訪問的內存、相應的內存屬性、以及一個或多個命令隊列（用來對內核的執行和內存對象上的相關運算進行調度）。

自定義設備（Custom Device）

一種 OpenCL 设备，完整地實現了 OpenCL 運行時，但不支持用 OpenCL C 編寫的程式。自定義設備可能是不可編程的專用硬件，高效節能，僅可執行定向任務；也可能具有有限的編程能力，如專用的 DSP。自定義設備並不符合 OpenCL。自定義設備可能支持在線編譯器。可以使用 OopenCL 運行時 API，由源碼（如果支持在線編譯器）和/或二元碼、或者內建內核來創建針對這種設備的程式。請參考设备。

數據並行編程模型（Data Parallel Programming Model）

傳統意義上，這種編程模型中，並發由單個程式中的指令表示，這些指令會作用到一組數據結構中的多個元素上。在 OpenCL 中，對這個術語進行了推廣，此模型中，單個程式中的一組指令同時作用到一個索引抽象域中的所有點上。

设备（Device）

设备是一組計算器件的集合。命令隊列可用來將命令在设备上進行排隊。這些命令包括執行內核或讀寫內存對象。典型的 OpenCL 设备指 GPU、多核 CPU 以及其他處理器（如 DSP 和 Cell/B.E. 處理器）。

事件對象（Event Object）

事件對象封裝了運算（如一個命令）的狀態。他可以用來對上下文中的運算進行同步。

事件等待序列（Event Wait List）

事件等待序列是事件對象的序列，可用來控制何時執行命令。

框架（Framework）

一個軟件系統，包含一套用以支持軟件開發和執行的組件。典型的框架包括庫、API、運行時系統、編譯器等等。

全局 ID（Global ID）

全局 *ID*用來唯一標識一個作業項，源自執行內核時所指定的全局作業項數目。全局 *ID*是一個 N 維的值，起自 $(0, 0, \cdots 0)$。請參考局部 *ID*。

全局內存（Global Memory）

一塊內存區域，上下文中的所有作業項均可訪問。主機可以使用命令來訪問他（如讀、寫以及映射）。

GL 共享組（GL share group）

這種對象用來管理共享的 OpenGL 或 OpenGL ES 資源（如材質、緩衝、幀緩衝或渲染緩衝），並與一個或多個 GL 上下文對象關聯在一起。典型的*GL 共享組*是不透明的，不能直接訪問。

句柄（Handle）

一種不透明的型別，用來引用 OpenCL 所分配的對象。對對象的任何操作都是通過引用其句柄來進行的。

主機（host）

主機使用 OpenCL API 與上下文進行交互。

主機指針（Host pointer）

一種指針，指向主機上虛擬位址空間中的內存。

違規（Illegal）

明確規定不允許的系統行為，遇到這種問題時，OpenCL 會將其視為錯誤進行報告。

圖像對象（Image Object）

一種內存對象，用來存儲二維或三維的結構化陣列。只能通過讀寫函式訪問其中的圖像數據。讀取函式需要使用採樣器。

圖像對象封裝了以下資訊:

- 圖像的維數;
- 圖像中每個元素的描述;
- 一些屬性，用來描述使用情況以及分配自哪個區域;
- 圖像數據。

圖像中的所有元素均從預定義的一系列圖像格式中選取。

依賴於具體實作（Implementation Defined）

在符合 OpenCL 標準的不同實作中，明確規定可以不同的那些行為。對於這種行為，要求 OpenCL 的實作者提供相關文檔。

順序執行（In-order Execution）

OpenCL 中的一種執行模型，命令隊列中的命令按提交的順序執行，只有當正在運行的命令完成後，下一個才能開始運行。參看亂序執行。

內核（Kernel）

程式中聲明的一種函式，可以在 OpenCL 設備上執行。在程式中定義的任何函式，都可以通過加上限定符 `__kernel` 或 `kernel` 將其標識成內核。

內核對象（Kernel Object）

內核對象封裝了程式中聲明的一個 `__kernel` 函式以及執行此函式時所需的引數。

局部 ID（Local ID）

作業項的一種 ID，在所處作業組中是唯一的。局部 *ID*是一個 N 維的值，起自 $(0, 0, \cdots 0)$。請參考全局 *ID*。

局部內存（Local Memory）

一塊內存區域，隸屬於某個作業組，只有這個作業組中的作業項才能訪問。

標註（Marker）

一種命令，可用來給之前入隊的所有命令打標籤。此命令會返回一個事件，應用可以等在這個事件上，例如等待之前入隊的命令全部完成。

內存對象（Memory Object）

某塊帶有引用計數的全局內存的句柄。請參考緩衝對象和圖像對象。

內存區域（Memory Region）

OpenCL 中一個確切的位址空間。不同的內存區域在物理上可能重疊，但在邏輯上 OpenCL 會將其視為不同。分為私有內存、局部內存、不變內存以及全局內存。

內存池（Memory Pool）

同內存區域。

對象（Object）

這是對 OpenCL API 可以操控的資源的一種抽象表示。包括程式對象、內核對象以及內存對象。

亂序執行（Out-of-Order Execution）

一種執行模型，命令隊列中的命令可以任意順序開始和結束執行，只要符合事件等待序列和命令隊列屏障的限制即可。參見順序執行。

父設備（Parent Device）

一種 OpenCL 設备，被劃分成了多個子設備。並不是所有的父設備都是根設備。根設備可以被拆分，子設備可以繼續被拆分。這種情況下，第一組子設備可能是第二組子設備的父設備，但不是根設備。請參考设备、子設備和根設備。

平台（platform）

包括主機以及 OpenCL 框架所管理的一些设备。應用可以在這些设备上共享資源、執行內核。

私有內存（Private Memory）

專屬某個作業項的一塊內存。在一個作業項的私有內存中定義的變量對另一個作業項是不可見的。

處理元件（Processing Element）

一種虛擬的標量處理器。一個作業項可以在一個或多個處理元件上執行。

程式（Program）

由一組內核組成，可能還包含一些輔助函式（由 `__kernel` 函式調用）和常量数据。

程式對象（Program Object）

程式對象封裝有以下資訊：

- 對所屬上下文的引用。
- 程式源碼或二元碼。

- 最近成功構建的程式執行體，用來運行此程式的设备清單，所用的構建選項以及構建日誌。
- 當前附着其上的内核對象的數目。

引用計數（Reference Count）

OpenCL 對象的生命周期由其引用計數決定，這個內部計數記錄了此對象被引用的次數。在 OpenCL 中創建一個對象時，其引用計數被置為0。后续对保留 API（如 **clRetainContext**、**clRetainCommandQueue**）的調用會使其引用計數增一。而對釋放 API 如 **clReleaseContext**、**clReleaseCommandQueue**）的調用會使其引用計數減一。當引用計數降為 0 後，OpenCL 會回收此對象所佔用的資源。

放寬的一致性（Relaxed Consistency）

一種內存一致性模型。對同一塊內存而言，不同的作業項或命令所看到的內容可能不同，當然，屏障和其他顯式同步點除外。

資源（Resource）

OpenCL 所定義的一類對象。資源的實體就是對象。最常用的資源是上下文、命令隊列、程式對象、内核對象以及内存對象。計算用資源主要指那些參與推進程式計數器動作的硬件元件。例如主機、设备、計算器件以及處理元件。

保留（Retain）

參見釋放。

釋放（Release）

會使 OpenCL 對象的引用計數增一（retain）或減一（release）。這是一個記賬功能，可以保證在使用某個對象的所有實體都結束之前，系統不會移除此對象。參見引用計數。

根設備（Root Device）

一種 OpenCL 设备，還未劃分。請參考设备、父設備和子設備。

採樣器（Sampler）

一種對象，用來描述内核讀取圖像時怎樣對其採樣。讀取圖像的函式將採樣器作為一個引數。採樣器指定圖像的尋址方式，如圖像坐標溢出時如何處置、濾波模式、輸入的圖像坐標是否已歸一化。

單指令多數據（SIMD）

一種編程模型，一個内核在多個處理元件上並發執行，每個處理元件上都有自己的數據，並共享一個程式計數器。所有處理元件所執行的指令嚴格一致。

單程式多數據（SPMD）

一種编程模型，一個内核在多個處理元件上並發執行，每個處理元件上都有自己的數據和自己的程式計數器。因此，運行同一個内核的所有計算資源都會維護自己的指令計數器；同時由於内核中分支的存在，這些處理元件中的實際指令序列可能會有很大不同。

子設備（Sub Device）

一個 OpenCL 设备可以劃分成多個子設備。按照劃分方案，新的子設備是父設備中一組計算器件的別名。任何情況下，只要可以使用其父設備，就可以使用這些子設備。劃分设备不會摧毀父設備，可以繼續和其子設備混合使用。請參考设备、父設備和根設備。

任務並行編程模型（Task Parallel Programming Model）

一種編程模型，用多個並發的任務來表示計算，其中任務就是單個作業組（大小是1）中所執行的內核。這些並發的任務可以運行不同的內核。

線程安全（Thread-safe）

當一個 OpenCL API 被多個主機線程同時調用時，如果 OpenCL 所管理的內部狀態始終保持一致，就認為他是線程安全的。如果一个 OpenCL API 是線程安全的，那麼就允許應用在多個主機線程中同時調用他，而不必在這些線程間實施互斥，即他們也是重入安全的。

未定義（Undefined）

調用 OpenCL API、內核中使用內建函式或者執行內核時，如果 OpenCL 沒有對其行為顯式定義，那麼這種行為就是未定義的。至於在 OpenCL 中碰到這種情況時會發生什麼，不要求實作明確指出。

作業組（Work-group）

一組相關的作業項，在同一個計算器件上執行。其中所有作業項執行同一個內核，並共享局部內存和作業组屏障。

作業组屏障（Work-group Barrier）

參見屏障。

作業項（Work-item）

內核的並行執行體中的一個。作為在計算器件上執行的作業組中的一部分，一個作業項可以由一個或多個處理元件執行。用其全局 *ID*和局部 *ID*來區分作業項。

节 2.1 OpenCL 类图

圖 *2.1*使用統一建模語言[1]（UML）以類圖的方式對 OpenCL 規範進行了描述。圖中的節點和邊分別代表類和類之間的關係。簡化起見，類中沒有特性和操作。抽象類帶有注釋“{abstract}”。圖中顯示了類之間的關係，有聚集（aggregation，帶有實心方塊）、關聯（association，沒有註解）和繼承（inheritance，帶有開放的箭頭）。邊兩端帶有關係的基數。其中“*****”代表“多個”，“**1**”代表“一個且只有一個”，“**0..1**”代表“可選的一個”，而“**1..***”代表“一個或多個”。普通箭頭代表關係的方向。

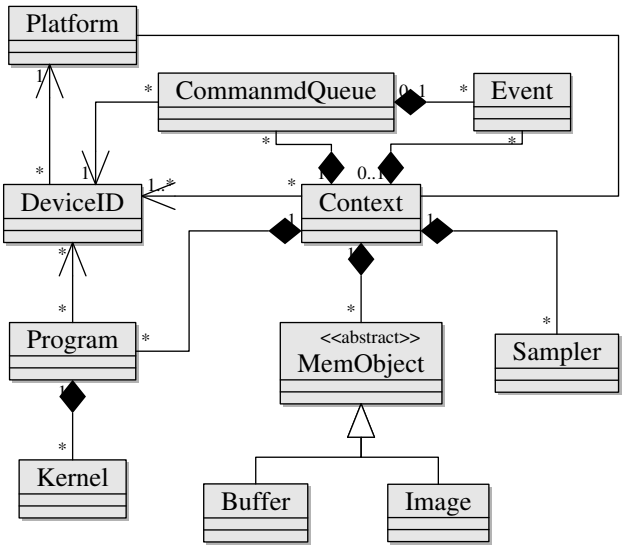


圖 2.1 **OpenCL 類 圖**

[1] 統一建模語言（http://www.uml.org/）是 Object Management Group（OMG）的商標。

第 3 章
OpenCL 架構

OpenCL是一個開放的工業標準，可以為 CPU、GPU 以及其他分散的計算設備（這些設備被組織到同一平台中）所組成的異構群進行編程。他不只是一種語言。 OpenCL 是一個並行編程框架，包括一種語言、API、庫以及一個運行時系統來支持軟件開發。例如，使用 OpenCL，程式員可以寫出一個能在 GPU 上執行的通用程式，而不必將其算法映射到 3D 圖形 API（如 OpenGL 或 DirectX）上。

OpenCL 的目標是讓程式員可以寫出可移植並且高效的代碼。這包括庫作者、中間件供應商，以及以性能為導向的應用程式員。因此 OpenCL 提供了底層的硬件抽象和一個框架來支持編程，同時也暴露了底層硬件的許多細節。

我們將使用如下的模型體系來描述 OpenCL 背後的核心理念:

- 平台模型
- 內存模型
- 執行模型
- 編程模型

节 3.1 平台模型

OpenCL 平台模型的定義可以查看圖 *3.1*。此模型中有一個**主機**，並且有一個或多個 **OpenCL** 设备與其相連。每個 OpenCL 设备可劃分成一個或多個計算器件（CU），每個計算器件又可劃分成一個或多個處理元件（PE）。设备上的計算是在處理元件中進行的。

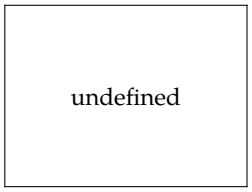


圖 3.1 平台模型

OpenCL 應用會按照主機平台的原生模型在這個主機上運行。主機上的 OpenCL 應用提交**命令**給设备中的處理元件以執行計算任務。計算器件中的處理元件會作為 SIMD 單元（執行指令流的步伐一致）或 SPMD 單元（每個 PE 維護自己的程式計數器）執行指令流。

3.1.1 平台對多版本的支持

OpenCL 的設計目標就是要支持同一平台中多種具有不同能力的設備。這些設備可以符合不同版本的 OpenCL 規範。對於一個 OpenCL 系統，有三個重要的版本 ID 要考慮: 平台版本、设备版本、设备所支持的 OpenCL C 語言的版本。

平台版本表明了所支持的 OpenCL 運行時的版本。這包括可由主機用來與 OpenCL 運行時交互的所有 API，像上下文、內存對象、设备、命令隊列。

设备版本表明了设备的能力，獨立於運行時和編譯器的版本，由 **`clGetDeviceInfo`** 所返回的设备資訊來描述。有很多特性都與设备版本有關，如資源限制和擴展功能。所返回的版本號即為此设备所符合的 OpenCL 規範的最高版本號，但不會高於平台版本。

語言版本，可以讓開發人員據此知道此设备所支持的 OpenCL 編程語言具備哪些特性。此版本會是所支持語言的最高版本。

OpenCL C 被設計為向後兼容的，因此對於一個设备而言，只要支持語言的某一個版本，就可以說他符合標準。如果某個设备支持多個語言版本，編譯器缺省使用最高的那個版本。語言的版本不會高於平台的版本，但可能高於设备的版本（參見節 *5.6.4.5*）。

节 3.2 執行模型

OpenCL 程式的執行分為兩種情況: 在一個或多個 **OpenCL 设备**上執行**內核**; 在主機上執行**主機程式**。主機程式為內核定義了上下文並管理內核的執行。

OpenCL 執行模型的核心就是內核是怎麼執行的。主機提交內核時會定義一個索引空間。內核的實體會在此空間中的所有點上執行。內核的實體稱為**作業項**，通過在索引空間中的坐標來標識，這個坐標就是作業項的全局 ID。所有作業項都會執行相同的代碼，但是代碼的執行路徑和參與運算的數據可能會不同。

作業項被組織到**作業組**中。作業組以更粗粒度對索引空間進行了分解。作業組帶有一個唯一的 ID，他與作業項所使用的索引空間具有同樣的維數。作業項具有一個局部 ID，此 ID 在其所隸屬的作業組中是唯一的；因此任一作業項都可以通過其全局 ID 或其局部 ID加作業組 ID 來唯一標識。同一作業組中的作業項會在同一計算器件中的多個處理元件上並發執行。

在 OpenCL 中，索引空間又叫做 NDRange。 NDRange 是一個 N 維的索引空間，其中 N 可以是一、二或者三。 NDRange 由一個長度為 N 的整數陣列來定義，他指定了索引空間各維度的寬度（起自偏移索引 F，F 缺省為 0）。每個作業項的全局 ID和局部 ID都是 N 維元組。全局 ID的取值範圍從 F 開始，直到 F 加相應維度上的元素個數減一。

作業組的 ID 跟作業項的全局 ID差不多。一個長度為 N 的陣列定義了每個維度上作業組的數目。作業項在所隸屬的作業組中有一個局部 ID，此 ID 中各維度的取值範圍為0到作業組在相應維度上的大小減一。因此，作業組的 ID 加上其中一個局部 ID可以唯一確定一個作業項。有兩種途徑來標識一個作業項: 根據全局索引，或根據作業組索引加一個局部索引。

接下來請看圖 *3.2*中的二維索引空間，其中包括作業項、其全局 ID以及相應的 ID 元組: 作業組 ID 和局部 ID。作業項的索引空間為 (G_x, G_y)，每個作業組的大小是 (S_x, S_y)，全局 ID的偏移量是 (F_x, F_y)。全局索引定義了一個 G_x 乘 G_y 的索引空間，所能容納的作業項總數是 G_x 和 G_y 的乘積。局部索引定義了一個 S_x 乘 S_y 的索引空間，一個作業組中所能容納作業項的數目是 S_x 和 S_y 的乘積。如果知道每個作業組的大小和作業項的總數，就能算出有多少作業組。作業組是由一個二維的索引空間來唯一標識的。作業項可以用他的全局 ID (g_x, g_y) 標識，或用作業組 ID (w_x, w_y)、作業組的大小 (S_x, S_y) 和在作業組中的局部 ID (s_x, s_y) 三項組合起來標識:

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

作業組的數目可以這樣計算:

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

給定全局 ID和作業組大小，作業項所屬作業組的 ID 為:

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

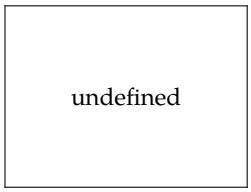


圖 3.2 **NDRange 索引空间示例**

很多編程模型都可以映射到這個執行模型上。 OpenCL 明確支持的有兩種: **數據並行編程模型**和**任務並行編程模型**。

3.2.1 執行模型：上下文和命令隊列

主機會為執行內核定義一個上下文。上下文包括以下資源:

1. **设备**: 主機可以使用的 OpenCL 设备集。
2. **內核**: 運行在 OpenCL 设备上的 OpenCL 函式。
3. **程式對象**: 實現內核的程式源碼和執行體。
4. **內存對象**: 一組內存對象，對主機和 OpenCL 设备可見。內存對象包含一些值，內核實體可以在其上進行運算。

主機使用 OpenCL API 中的函式來創建並操控上下文。主機會創建一個稱為命令隊列的數據結構來協調设备上內核的執行。主機還會將命令入隊，這些命令將在上下文中的设备上被調度。這些命令包括:

- **內核執行命令**: 在设备的處理元件上執行內核。
- **內存命令**: 讀寫內存對象或者在內存對象間傳輸數據，或者從主機的位址空間中映射、解映射內存對象。
- **同步命令**: 限制命令的執行順序。

命令隊列負責命令的調度，使其可以在设备上執行。在主機和设备上，命令的執行是異步的。命令的執行有兩種模式:

- **順序執行**: 命令嚴格按照在命令隊列中出現的順序開始和結束執行。換言之，前面的命令結束後，才能執行後面的命令。這使隊列中命令的執行順序串行化。
- **亂序執行**: 按順序執行命令，但後續命令執行前不必等待前面命令結束。任何順序上的限制都是由程式員通過顯式的同步命令强加的。

提交給隊列的內核執行命令和內存命令都會生成事件對象。這些事件對象可以用來控制命令的執行順序、協調命令在主機和设备間的運行。

一個上下文中可以有多個隊列。這些隊列並發運行、相互獨立，OpenCL 中沒有顯式的機制來對他們進行同步。

3.2.2 執行模型：內核的種類

OpenCL 執行模型支持兩種內核:

- **OpenCL 內核**，用 OpenCL C 編程語言編寫，並用 OpenCL C 編譯器編譯而成。所有 OpenCL 的實作都支持 OpenCL 內核。實作也可能提供其他機制創建 OpenCL 內核。
- **原生內核**，通過主機函式指針訪問。原生內核與 OpenCL 內核一起入隊在设备上執行，並共享內存對象。例如，這些原生內核可以是應用代碼中定義的函式，也可以是從庫中導出的函式。注意，在 OpenCL 中，執行原生內核的能力是一個可選功能，原生內核的語義依賴於具體實作。可以使用 OpenCL API 中的一些函式來查詢设备的能力並確定设备是否具備某種能力。

节 3.3 內存模型

作業項在執行內核時可以訪問四塊不同的內存區域:

- **全局內存**: 所有作業組中的所有作業項都可以對其進行讀寫。作業項可以讀寫此中內存對象的任意元素。對全局內存的讀寫可能會被緩存起來，這取決於设备的能力。
- **不變內存**: 全局內存中的一塊區域，在內核的執行過程中保持不變。主機負責對此中內存對象的分配和初始化。
- **局部內存**: 隸屬於某個作業組。可以用來分配一些變量，這些變量由此作業組中的所有作業項共享。在 OpenCL 设备上，可能會將其實現成一塊專用的內存區域，也可能將其映射到全局內存中。
- **私有內存**: 隸屬於某個作業項。一個作業項的私有內存中所定義的變量對另外一個作業項而言是不可見的。

*表 3.1*列出了這些資訊: 內核或主機是否可以從某個內存區域中分配內存、怎樣分配（靜態編譯時 vs. 動態運行時）以及允許如何訪問（即內核或主機是否可以對其進行讀寫）。

表 3.1 內存區域——分配以及訪問

	全局內存	不變內存	局部內存	私有內存
主機	動態分配 可讀可寫	動態分配 可讀可寫	動態分配 不可訪問	不可分配 不可訪問
內核	不可分配 可讀可寫	靜態分配 只讀	靜態分配 可讀可寫	靜態分配 可讀可寫

圖 *3.3*描述了內存區域以及與平台模型的關係。圖中含有處理元件（PE）、計算器件和设备，但是沒有畫出主機。

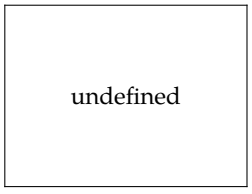


圖 3.3 OpenCL 設備架構的概念模型

應用在主機上運行時，使用 OpenCL API 在全局內存中創建內存對象，並將內存命令(節 *3.2.1*中有所描述)入隊以操作他們。

多數情況下，主機和 OpenCL 设备的內存模型是相互獨立的。其必然性主要在於 OpenCL 沒有囊括主機的定義。然而，有時他們確實需要交互。有兩種交互方式：顯式拷貝數據、將內存對象的部分区域映射和解映射。

為了顯式拷貝數據，主機會將一些命令插入隊列，用來在內存對象和主機內存之間傳輸數據。這些用於傳輸內存的命令可以是阻塞式的，也可以是非阻塞式的。對於前者，一旦主機上相關內存資源可以被安全的重用，OpenCL 函式調用就會立刻返回。而對於後者，一旦命令入隊，OpenCL 函式調用就會返回，而不管主機內存是否可以安全使用。

用映射、解映射的方法處理主機和 OpenCL 內存對象的交互時，主機可以將內存對象的某個區域映射到自己的位址空間中。內存映射命令可能是阻塞的，也可能是非阻塞的。一旦映射了內存對象的某個區域，主機就可以讀寫這塊區域。當主機對這塊區域的訪問（讀和/或寫）結束後，就會將其解映射。

3.3.1 內存一致性

OpenCL 所使用的一致性內存模型比較寬鬆；即，不保證不同作業項所看到的內存狀態始終一致。

在作業項內部，內存具有裝載、存儲的一致性。在隸屬於同一作業組的作業項之間，局部內存在作業组屏障上是一致的。全局內存亦是如此，但對於執行同一內核的不同作業組，則不保證其內存一致性。

對於已經入隊的命令所共享的內存對象，其內存一致性由同步點來強制實施。

节 3.4 編程模型

OpenCL 執行模型支持數據並行編程模型和任務並行編程模型，同時也支持這兩種模型的混合體。對於 OpenCL 而言，驅動其設計的首要模型是數據並行。

3.4.1 數據並行編程模型

數據並行編程模型依據同時應用到內存對象的多個元素上的指令序列來定義計算（computation）。OpenCL 執行模型所關聯的索引空間定義了作業項，以及數據怎樣映射到作業項上。在嚴格的數據並行模型中，作業項和內存對象的元素間的映射關係為一對一，內核可在這些元素上並行執行。對於數據並行編程模型，OpenCL 所實現的版本則比較寬鬆，不要求嚴格的一對一的映射。

OpenCL 所提供的數據並行編程模型是分級的。有兩種方式來進行分級。一種是顯式方式，程式員定義可以並行執行的作業項的總數、以及怎樣將這些作業項劃分到作業組中。另一種是隱式方式，程式員僅指定前者，後者由 OpenCL 實作來管理。

3.4.2 任務並行編程模型

在 OpenCL 的任務並行編程模型中，內核的實體在執行時獨立於任何索引空間。這在邏輯上等同於：在計算器件上執行內核時，相應作業組中只有一個作業項。這種模型下，用戶以如下方式表示並行：

- 使用设备所實現的矢量數據型別；
- 將多個任務入隊，和/或
- 將多個原生內核入隊（他們是使用一個與 OpenCL 正交的編程模型開發的）。

3.4.3 同步

在 OpenCL 中，有兩方面的同步：

- 隸屬同一作業組的作業項之間；
- 隸屬同一上下文的命令隊列中的命令之間。

前者是通過作業组屏障實現的。對於同一作業組中的所有作業項來說，任意一個要想越過屏障繼續執行，所有作業項都必須先執行這個屏障。注意，在同一作業組中，所有正在執行內核的作業項必須都能執行到這個作業组屏障，或者都不會去執行。作業組之間沒有同步機制。

命令隊列中命令間的同步點是：

- 命令隊列屏障。他保證：所有之前排隊的命令都執行完畢，並且他們對內存對象的所有更新，在後續命令開始執行前都是可見的。他只能在隸屬同一命令隊列的命令間進行同步。

- 等在一個事件上。所有會將命令入隊的 OpenCL API 函式都會返回一個事件（用來標識這個命令以及他所更新的內存對象）。如果某個後續命令正在等待那個事件，可以保證在其開始執行前，可以見到對那些內存對象的所有更新。

节 3.5 內存對象

內存對象分成兩類：*緩衝對象*和*圖像對象*。*緩衝對象*中所存儲的元素是一維的，而*圖像對象*則用來存儲二維或三維的材質、幀緩衝（frame-buffer）或圖像。

*緩衝對象*中的元素可以是標量數據型別（如 `int`、`float`）、矢量數據型別或用戶自定義的結構體。*圖像對象*用來表示材質、幀緩衝等緩衝。圖像對象中元素的格式必須從預定義格式中選取。內存對象中至少要有一個元素。

*緩衝對象*和*圖像對象*的根本區別是：

- *緩衝對象*中的元素是順序存儲的，內核在设备上運行時可以用指針訪問這些元素。而*圖像對象*中元素的存儲格式對用戶是透明的，不能通過指針直接訪問。可以使用 OpenCL C 編程語言提供的內建函式來讀寫圖像對象。
- 對於*緩衝對象*，內核按其存儲格式訪問其中的數據。而對於*圖像對象*，其元素的存儲格式可能與內核中使用的數據格式不一樣。內核中的圖像元素始終是四元矢量（每一元都可以是浮點型別或者帶符號/無符號整形）。內建函式在讀寫圖像元素時會進行相應的格式轉換。

內存對象是用 `cl_mem` 來表示的。內核的輸入和輸出都是內存對象。

节 3.6 OpenCL 框架

OpenCL 框架中，一個主機、加上不少於一個的 OpenCL 设备就可以構成一個異構並行計算機系統，並由應用來使用。這個框架包含以下組件：

- **OpenCL 平台層**：主機程式可以發現 OpenCL 设备及其能力，也可以創建上下文。
- **OpenCL 運行時**：一旦創建了上下文，主機程式就可以操控他。
- **OpenCL 編譯器**：OpenCL 編譯器可以創建含有 OpenCL 內核的程式執行體。他所實現的 OpenCL C 編程語言支持 ISO C99 的一個子集，並帶有並行擴展。

第 4 章
OpenCL 平台層

本章將介紹 OpenCL 平台層，他實現了具體平台相關的特性，允許應用查詢 OpenCL 设备、设备配置資訊，以及創建 OpenCL 上下文來使用這些设备。

节 4.1 查詢平台資訊

clGetPlatformIDs

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

可以使用此函式獲取可用平台的清單。

num_entries 是 *platforms* 中可以容納 cl_platform_id 表項的數目。如果 *platforms* 不是 NULL，*num_entries* 必須大於零。

platforms 會返回所找到的 OpenCL 平台清單。*platforms* 中的每個 cl_platform_id 都用來標識某個特定的 OpenCL 平台。如果 *platforms* 是 NULL，則被忽略。所返回 OpenCL 平台的數目是 *num_entries* 和實際數目中較小的那個。

num_platforms 返回實際可用的 OpenCL 平台的數目。如果 *num_platforms* 是 NULL，則被忽略。

如果執行成功，**clGetPlatformIDs** 會返回 CL_SUCCESS。否則，返回下列錯誤之一：

- CL_INVALID_VALUE，如果 *num_entries* 等於零，且 *platforms* 不是 NULL；或者 *num_platforms* 和 *platforms* 都是 NULL。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clGetPlatformInfo

```
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)
```

此函式可用來獲取 OpenCL 平台的特定資訊。這些資訊如表 *4.1*所示。

platform 即 **clGetPlatformIDs** 所返回的平台 ID，指明要查詢哪個平台，也可以是 NULL。而如果是 NULL，其行為依賴於具體實作。

param_name 是一個枚舉常量，指明要查詢什麼資訊。其值如表 *4.1*所示。

param_value 是一個指針，所指內存用來存儲 *param_name* 所對應的資訊。如果是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小（單位：字節）。其值必須 >= 表 *4.1*中返回型別的大小。

param_value_size_ret 返回所查詢資訊的實際大小。如果是 NULL，則忽略。

如果執行成功，**clGetPlatformInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤之一[2]：

- CL_INVALID_PLATFORM，如果 *platform* 無效。
- CL_INVALID_VALUE，如果不支持 *param_name*，或 *param_value_size* < 表 *4.1*中返回值的大小且 *param_value* 不是 NULL。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

节 4.2 查詢設備

clGetDeviceIDs

[2] OpenCL 規範沒有規定 API 調用返回錯誤碼時的優先順序。

表 4.1 OpenCL 平 台 查 詢

cl_platform_info	返回型別[1]
CL_PLATFORM_PROFILE	char[]
OpenCL 規格字串。返回實作所支持的規格名。可以是下列之一: • FULL_PROFILE——表示實作支持 OpenCL 規範（核心規範所定義的功能，無須支持任何擴展）。 • EMBEDDED_PROFILE——表示實作支持 OpenCL 嵌入式規格。他是對應版本 OpenCL 的一個子集。 OpenCL 1.2的嵌入式規格請參考第 *10*章。	
CL_PLATFORM_VERSION	char[]
OpenCL 版本字串。返回所支持的 OpenCL 版本。其格式如下: *OpenCL<space><major_version.minor_version><space><platform-specific information>* 所返回的 *major_version.minor_version* 將是 1.2。	
CL_PLATFORM_NAME	char[]
平台名字。	
CL_PLATFORM_VENDOR	char[]
平台供應商的名字。	
CL_PLATFORM_EXTENSIONS	char[]
返回平台所支持的擴展名，以空格分隔（擴展名本身不包含空格）。此平台關聯的所有设备都要支持此處定義的擴展。	

[1] 對於 OpenCL 查詢函式，如果所查詢資訊的型別為 char[]，則會返回一個以 null 終止的字串。

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

函式 **clGetDeviceIDs**[3] 可用來獲取一個平台上所有可用设备的清單。

platform 即 **clGetPlatformIDs** 所返回的平台 ID，也可能是 NULL。如果是 NULL，則其行為依賴於具體實作。

device_type 是位欄（bitfield），用來標識 OpenCL 设备的類型。可以用來查詢某種 OpenCL 设备，也可以查詢所有的。請參考表 *4.2*。

表 4.2 OpenCL 設 備 種 類 清 單

cl_device_type
CL_DEVICE_TYPE_CPU
主機處理器。OpenCL 實作運行其上，是單核或多核 CPU。
CL_DEVICE_TYPE_GPU
GPU。這意味着此设备也可以用來加速 3D API （如 OpenGL 或 DirectX）。
CL_DEVICE_TYPE_ACCELERATOR
OpenCL 專用加速器（如 IBM CELL Blade)。這些設備通過外圍設備互聯總線（如 PCIe）與主機處理器通信。
CL_DEVICE_TYPE_CUSTOM
一種專用加速器，但是不支持用 OpenCL C 編寫的程式。
CL_DEVICE_TYPE_DEFAULT
系統中默認的 OpenCL 设备。不能是 CL_DEVICE_TYPE_CUSTOM 類型的設備。
CL_DEVICE_TYPE_ALL
系統中所有可用的 OpenCL 设备。不包括 CL_DEVICE_TYPE_CUSTOM 類型的設備。

num_entries 是 *devices*中所能容納 cl_device 表項的數目。如果 *devices* 不是 NULL，則 *num_entries* 必須大於零。

devices 用來返回所找到的 OpenCL 设备。 *devices* 中返回的 cl_device_id 用來標識一個 OpenCL 设备。如果參數 *devices* 是 NULL，則忽略。所返回的 OpenCL 设备數目為 *num_entries* 或符合 device_type 的设备数目，取二者中較小的那個。

[3] **clGetDeviceIDs** 可能返回 *platform* 中所有與 *device_type* 匹配的真正的物理設備，也可能只是其中一個子集。

num_devices 返回符合 device_type 的所有 OpenCL 设备的數目。如果 *num_devices* 是 NULL，則忽略。

如果執行成功，則 **clGetDeviceIDs** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_PLATFORM，如果 *platform* 無效。
- CL_INVALID_DEVICE_TYPE，如果 *device_type* 無效。
- CL_INVALID_VALUE，如果 *num_entries* 等於零且 *devices* 不是 NULL，或者 *num_devices* 和 *devices* 都是 NULL。
- CL_DEVICE_NOT_FOUND，如果沒有找到任何符合 *device_type* 的 OpenCL 设备。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

對於 **clGetDeviceIDs** 所返回的 OpenCL 设备，應用可以查詢其能力，從而決定使用哪些设备。

clGetDeviceInfo

```
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)
```

函数 **clGetDeviceInfo** 可用來獲取 OpenCL 设备的特定資訊，如表 4.3所示。

device 是 **clGetDeviceIDs** 所返回的设备，或者由 **clCreateSubDevices** 創建的子設備。如果是子設備，則返回子設備的資訊。

param_name 是枚舉常量，用來指明要查詢什麼資訊，參見表 4.3。

param_value 是指針，所指內存用來存儲 *param_name* 所對應的值（參見表 4.3）。如果 *param_value* 是 NULL，則忽略。

param_value_size 就是 *param_value* 所指內存塊的字節數。其值必須 >= 表 4.3中所列返回型別的大小。

param_value_size_ret 返回 *param_value* 所對應數據的實際大小。如果 *param_value_size_ret* 是 NULL，則忽略。

表 4.3α OpenCL 設 備 查 詢

cl_device_info	返回型別
CL_DEVICE_TYPE	cl_device_type
OpenCL 设备類型，當前支持: • CL_DEVICE_TYPE_CPU， • CL_DEVICE_TYPE_GPU， • CL_DEVICE_TYPE_ACCELERATOR， • CL_DEVICE_TYPE_DEFAULT，或者 • 以上值的组合，或者 • CL_DEVICE_TYPE_CUSTOM。	
CL_DEVICE_VENDOR_ID	cl_uint
唯一的设备供應商標識符。例如可以是 PCIe ID。	
CL_DEVICE_MAX_COMPUTE_UNITS	cl_uint
OpenCL 设备上的並行計算器件的數目。最小值是 1。	
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	cl_uint
數據並行編程模型中所用全局和局部作業項 ID 的最大維數（參見 **clEnqueueNDRangeKernel**）。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，其最小值是 3。	
CL_DEVICE_MAX_WORK_ITEM_SIZES	size_t[]
對**clEnqueueNDRangeKernel**而言，作業組中每個維度上可以指派作業項的最大數目。 返回 *n* 個型別為 size_t 的表項。其中 *n* 是查詢 CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS 時所返回的值。 對於不是 CL_DEVICE_TYPE_CUSTOM 的设备，最小值是$(1, 1, 1)$。	
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t

表 4.3β OpenCL 設 備 查 詢

用數據並行編程模型執行內核時，作業組中所能容納作業項的最大數目（參见 **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_DEVICE_PREFERRED_VECTOR_WIDTH_HALF	cl_uint
可以放入矢量中的內建標量型別所期望的原生矢量的寬度。矢量寬度定義為可以可以容納標量元素的數目。 如果不支持雙精度浮點數，CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE 必須返回0。 如果不支持擴展 **cl_khr_fp16**，CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF 必須返回0。	
CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT CL_DEVICE_NATIVE_VECTOR_WIDTH_INT CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF	cl_uint
返回原生 ISA 矢量寬度。此矢量寬度定義為所能容納標量元素的數目。 如果不支持雙精度浮點數，CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE 必須返回0。 如果不支持擴展 **cl_khr_fp16**，CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF 必須返回0。	
CL_DEVICE_MAX_CLOCK_FREQUENCY	cl_uint
设备的時鐘頻率可以配置成的最大值，單位：MHz。	
CL_DEVICE_ADDRESS_BITS	cl_uint
計算設備的位址空間缺省大小，無符號整數，單位：bit。當前支持 32 位或 64 位。	
CL_DEVICE_MAX_MEM_ALLOC_SIZE	cl_ulong
所能分配的內存對象大小的最大值，單位：字節（byte）。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，此值最小為： $max($CL_DEVICE_GLOBAL_MEM_SIZE $* 1/4, 128 * 1024 * 1024)$	
CL_DEVICE_IMAGE_SUPPORT	cl_bool
如果 OpenCL 设备支持圖像，則為 CL_TRUE，否則為 CL_FALSE。	
CL_DEVICE_MAX_READ_IMAGE_ARGS	cl_uint
內核可以同時讀取多少圖像對象。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 128。	
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	cl_uint
內核可以同時寫入多少圖像對象。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 8。	
CL_DEVICE_IMAGE2D_MAX_WIDTH	size_t
2D 图像或非緩衝對象所創建的 1D 圖像的最大寬度，單位：像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 8192。	
CL_DEVICE_IMAGE2D_MAX_HEIGHT	size_t
2D 圖像的最大高度，單位：像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 8192。	
CL_DEVICE_IMAGE3D_MAX_WIDTH	size_t
3D 圖像的最大寬度，單位：像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 2048。	
CL_DEVICE_IMAGE3D_MAX_HEIGHT	size_t
3D 圖像的最大高度，單位：像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 2048。	
CL_DEVICE_IMAGE3D_MAX_DEPTH	size_t
3D 圖像的最大深度，單位：像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 2048。	
CL_DEVICE_IMAGE_MAX_BUFFER_SIZE	size_t
由緩衝對象所創建的 1D 圖像的最大像素數。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 65536。	

表 4.3γ OpenCL 設 備 查 詢

CL_DEVICE_IMAGE_MAX_ARRAY_SIZE	size_t
1D 或 2D 圖像陣列中圖像的最大數目。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 2048。	
CL_DEVICE_MAX_SAMPLERS	cl_uint
一個內核內最多可以使用多少個採樣器。關於採樣器的細節請參考節 *6.12.14*。 如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 16。	
CL_DEVICE_MAX_PARAMETER_SIZE	size_t
內核引數的最大字節數。 如果設備類型不是 CL_DEVICE_TYPE_CUSTOM，則此值至少要是 1024。如果是1024，則內核參數最多是128個。	
CL_DEVICE_MEM_BASE_ADDR_ALIGN	cl_uint
如果設備類型不是 CL_DEVICE_TYPE_CUSTOM，至少要是設備所支持的 OpenCL 內建數據型別中最大的那種的大小，單位: bit。（FULL 規格中是 long16，EMBEDDED 規格中是 long16 或 int16）	
CL_DEVICE_SINGLE_FP_CONFIG	cl_device_fp_config
描述设备的單精度浮點能力。此位欄支持下列值: ● CL_FP_DENORM——支持去規格化數（denorm）。 ● CL_FP_INF_NAN——支持 INF 和 qNaN。 ● CL_FP_ROUND_TO_NEAREST——支持捨入為最近偶數。 ● CL_FP_ROUND_TO_ZERO——支持向零捨入。 ● CL_FP_ROUND_TO_INF——支持向正無窮和負無窮捨入。 ● CL_FP_FMA——支持 IEEE754-2008 中的積和熔加运算（fused multiply-add, FMA）。 ● CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT——除法和開方可以按 IEEE754 規範進行正確的捨入。 ● CL_FP_SOFT_FLOAT——軟件中實現了基本的浮點運算（加、減、乘）。 如果设备類型不是 CL_DEVICE_TYPE_CUSTOM，其浮點能力至少要是: CL_FP_ROUND_TO_NEAREST | CL_FP_INF_NAN。	
CL_DEVICE_DOUBLE_FP_CONFIG	cl_device_fp_config
描述设备的雙精度浮點能力。此位欄支持下列值: ● CL_FP_DENORM——支持去規格化數。 ● CL_FP_INF_NAN——支持 INF 和 qNaN。 ● CL_FP_ROUND_TO_NEAREST——支持捨入為最近偶數。 ● CL_FP_ROUND_TO_ZERO——支持向零捨入。 ● CL_FP_ROUND_TO_INF——支持向正無窮和負無窮捨入。 ● CL_FP_FMA——支持 IEEE75-2008 中的積和熔加运算（fused multiply-add, FMA）。 ● CL_FP_SOFT_FLOAT——軟件中實現了基本的浮點運算（加、減、乘）。 由於雙精度浮點是一個可選特性，所以最小的雙精度浮點能力可以是 0。 而如果设备支持雙精度浮點，則其能力至少要是: 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。	
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	cl_device_mem_cache_type
所支持的全局內存緩存的類型。其值可以是: ● CL_NONE, ● CL_READ_ONLY_CACHE 和 ● CL_READ_WRITE_CACHE。	
CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE	cl_uint
全局內存緩存列（cache line）的字節數。	
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE	cl_ulong
全局內存緩存的字節數。	

表 4.36 OpenCL 設 備 查 詢

CL_DEVICE_GLOBAL_MEM_SIZE	cl_ulong
全局设备内存的字節數。	
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	cl_ulong
一次所能分配的常量緩衝區的最大字節數。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，最小值是 64KB。	
CL_DEVICE_MAX_CONSTANT_ARGS	cl_uint
單個內核中，最多能有多少個參數在聲明時帶有限定符__*constant*。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，最小值是 8。	
CL_DEVICE_LOCAL_MEM_TYPE	cl_device_local_mem_type
所支持的局部內存的類型。可以是 CL_LOCAL（意指專用的局部內存，如 SRAM ）或 CL_GLOBAL。 對於自定義設備，如果不支持局部內存，可以返回 CL_NONE。	
CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong
局部內存區的字節數。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，最小值是 32KB。	
CL_DEVICE_ERROR_CORRECTION_SUPPORT	cl_bool
如果所有對計算設備內存（包括全局內存和不變內存）的訪問，都可以由设备進行糾錯，則為 CL_TRUE，否則為 CL_FALSE。	
CL_DEVICE_HOST_UNIFIED_MEMORY	cl_bool
如果设备和主機共有一個統一的內存子系統，則為 CL_TRUE，否則為 CL_FALSE。	
CL_DEVICE_PROFILING_TIMER_RESOLUTION	size_t
设备定時器的精度。單位是納秒。詳情參見節 *5.12*。	
CL_DEVICE_ENDIAN_LITTLE	cl_bool
如果 OpenCL 设备是小端（little-endian）的，則為 CL_TRUE，否則為CL_FALSE。	
CL_DEVICE_AVAILABLE	cl_bool
如果设备可用，則為 CL_TRUE，否則為CL_FALSE。	
CL_DEVICE_COMPILER_AVAILABLE	cl_bool
如果沒有可用的編譯器來編譯程式源碼，則為 CL_FALSE，否則為 CL_TRUE。 只有嵌入式平台的規格才可以是 CL_FALSE。	
CL_DEVICE_LINKER_AVAILABLE	cl_bool
如果沒有可用的鏈接器，則為 CL_FALSE，否則為 CL_TRUE。 只有在嵌入式平台規格中才可以是 CL_FALSE。 如果 CL_DEVICE_COMPILER_AVAILABLE 是 CL_TRUE，則他必須是 CL_TRUE。	
CL_DEVICE_EXECUTION_CAPABILITIES	cl_device_exec_capabilities
描述设备的執行能力。此位欄包含以下值: • CL_EXEC_KERNEL——這個 OpenCL 设备可以執行 OpenCL 內核。 • CL_EXEC_NATIVE_KERNEL——這個 OpenCL 设备可以執行原生內核。 其中 CL_EXEC_KERNEL 是必需的。	
CL_DEVICE_QUEUE_PROPERTIES	cl_command_queue_properties

表 4.3ε **OpenCL 設 備 查 詢**

命令隊列的屬性。此位欄包含以下值: ● CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE ● CL_QUEUE_PROFILING_ENABLE 參見表 *5.1*。 其中 CL_QUEUE_PROFILING_ENABLE 是必需的。	
CL_DEVICE_BUILT_IN_KERNELS	char[]
设备所支持的內建內核的清單, 以分號分隔。如果不支持內建內核, 則返回空字串。	
CL_DEVICE_PLATFORM	cl_platform_id
此设备所關聯的平台。	
CL_DEVICE_NAME	char[]
设备的名字。	
CL_DEVICE_VENDOR	char[]
供應商的名字。	
CL_DEVICE_VERSION	char[]
OpenCL 軟件驅動的版本, 格式為: *major_number.minor_number*。	
CL_DEVICE_PROFILE	char[]
OpenCL 規格字串。返回设备所支持的規格名稱。可以是下列字串之一: ● FULL_PROFILE——如果设备支持 OpenCL 規範 (核心規格所定義的功能, 不要求支持任何擴展)。 ● EMBEDDED_PROFILE——如果设备支持 OpenCL 嵌入式規格。 返回的是 OpenCL 框架所實現了的規格。如果返回的是 FULL_PROFILE, 則 OpenCL 框架支持符合 FULL_PROFILE 的设备, 可能也支持符合 EMBEDDED_PROFILE 的设备。所有设备都得有可用的編譯器, 即 CL_DEVICE_COMPILER_AVAILABLE 必須是 CL_TRUE。而如果返回的是 EMBEDDED_PROFILE, 則僅支持符合 EMBEDDED_PROFILE 的设备。	
CL_DEVICE_VERSION	char[]
OpenCL 版本字串。返回设备所支持的 OpenCL 版本。格式如下: *OpenCL<space><major_version.minor_version><space><vendor-specific information>* 所返回的 *major_version.minor_version* 的值將是1.2。	
CL_DEVICE_OPENCL_C_VERSION	char[]
OpenCL C 版本字串。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备, 返回編譯器在其上所支持的 OpenCL C 的最高版本。格式如下: *OpenCL<space>C<space><major_version.minor_version><space><vendor-specific information>* 如果 CL_DEVICE_VERSION 是 OpenCL 1.2, 則 *major_version.minor_version* 必須是 1.2。如果 CL_DEVICE_VERSION 是 OpenCL 1.1, 則 *major_version.minor_version* 必須是 1.1。如果 CL_DEVICE_VERSION 是 OpenCL 1.0, 則 *major_version.minor_version* 可以是 1.0 或 1.1。	
CL_DEVICE_EXTENSIONS	char[]

表 4.3ζ OpenCL 設 備 查 詢

返回设备所支持的擴展名清單，以空格分隔（擴展名本身不包含空格）。所返回的清單可能包含供應商支持的擴展名，也可能是下列已獲 Khronos 批准的擴展名:

- **cl_khr_int64_base_atomics**
- **cl_khr_int64_extended_atomics**
- **cl_khr_fp16**
- **cl_khr_gl_sharing**
- **cl_khr_gl_event**
- **cl_khr_d3d10_sharing**
- **cl_khr_media_sharing**
- **cl_khr_d3d11_sharing**

對於支持 OpenCL C 1.2的设备，所返回的清單中必須包含下列已獲 Khronos 批准的擴展名:

- **cl_khr_global_int32_base_atomics**
- **cl_khr_global_int32_extended_atomics**
- **cl_khr_local_int32_base_atomics**
- **cl_khr_local_int32_extended_atomics**
- **cl_khr_byte_addressable_store**
- **cl_khr_fp64**（如果支持雙精度浮點，為向後兼容必須支持此擴展）

詳情請參考《OpenCL 1.2擴展規範》。

CL_DEVICE_PRINTF_BUFFER_SIZE	size_t
內核調用 printf 時，由一個內部緩衝區存儲其輸出，此區域大小的最大值。對於 FULL 規格，最小為 1MB。	
CL_DEVICE_PREFERRED_INTEROP_USER_SYNC	cl_bool
OpenCL 和其他 API（如 DirectX ）間共享內存對象時，如果设备的偏好是讓用戶自己負責同步，則其值為 CL_TRUE; 而如果设备或實作已經具備有效的方式來進行同步，則其值為 CL_FALSE。	
CL_DEVICE_PARENT_DEVICE	cl_device_id
返回此子設備所屬父設備的 cl_device_id。如果 *device* 是根設備，則返回 NULL。	
CL_DEVICE_PARTITION_MAX_SUB_DEVICES	cl_uint
劃分设备時，所能創建的子設備的最大數目。 所返回的值不能超過 CL_DEVICE_MAX_COMPUTE_UNITS。	
CL_DEVICE_PARTITION_PROPERTIES	cl_device_partition_property[]
返回 *device* 所支持的劃分方式。這是一個陣列，元素型別為 cl_device_partition_property，其值可以是: • CL_DEVICE_PARTITION_EQUALLY • CL_DEVICE_PARTITION_BY_COUNTS • CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN 如果此设备不支持任何劃分方式，則返回 0。	
CL_DEVICE_PARTITION_AFFINITY_DOMAIN	cl_device_affinity_domain
用 CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN 劃分 *device* 時，所支持的相似域（ affinity domain)。此位欄的值如下所示: • CL_DEVICE_AFFINITY_DOMAIN_NUMA • CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE • CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE • CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE • CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE • CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE 如果以上都不支持，就返回 0。	
CL_DEVICE_PARTITION_TYPE	cl_device_partition_property[]
如果 *device* 是子設備，則會返回調用 **clCreateSubDevices** 時所指定的引數 *properties*。否則返回的 *param_value_size_ret* 可能是 0 即不存在任何劃分方式; 或者 *param_value* 所指內存中的屬性值是 0（0 用來終止屬性清單）。	
CL_DEVICE_REFERENCE_COUNT	cl_uint
返回 *device* 的引用計數。如果是根設備，則返回 1。	

無論對於根設備（即 **clGetDeviceIDs** 所返回的设备）還是由他創建的子設備，表 4.3中所列查詢都會返回相同的資訊，不過下列查詢例外：

- CL_DEVICE_GLOBAL_MEM_CACHE_SIZE
- CL_DEVICE_BUILT_IN_KERNELS
- CL_DEVICE_PARENT_DEVICE
- CL_DEVICE_PARTITION_TYPE
- CL_DEVICE_REFERENCE_COUNT

如果執行成功，**clGetDeviceInfo** 會返回 CL_SUCCESS；否則，返回下列錯誤碼之一：

- CL_INVALID_DEVICE——如果 *device* 無效。
- CL_INVALID_VALUE——如果不支持 *param_name*；或者 *param_value_size* 的值 < 表 4.3中所列返回型別的大小，並且 *param_value* 不是 NULL；或者 *param_name* 指的是某個擴展，但是此设备不支持此擴展。
- CL_OUT_OF_RESOURCES——如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY——如果在為主機上的 OpenCL 實作分配資源時失敗。

节 4.3 劃分設備

clCreateSubDevices

```
cl_int clCreateSubDevices (
                cl_device_id in_device,
                const cl_device_partition_property *properties,
                cl_uint num_devices,
                cl_device_id *out_devices,
                cl_uint *num_devices_ret)
```

此函式會按照 *properties* 給定的劃分方案，由 *in_device* 創建一個子設備陣列，其中每個子設備都包含一組計算器件，且他們之間沒有交集。子設備的使用方式跟根設備（或其父設備）一樣，如可以用來創建上下文、構建程式、進一步調用 **clCreateSubDevices** 以及創建命令隊列。如果用子設備創建命令隊列，則其中的命令只能在這個子設備上執行。

in_device 即為要劃分的設備。

properties 則指明怎樣劃分。每個劃分方式緊跟相應的值。此清單以 0 終止。表 4.4列出了所支持的劃分策略。*properties*只能從中選擇一個。

表 4.4 **clCreateSubDevices 所支持的劃分策略**

cl_device_partition_property	**partition value**
CL_DEVICE_PARTITION_EQUALLY	unsigned int
將一個大的设备集合分割成許多小的设备集合，每個都含有 n 個計算器件。 n 伴隨此屬性一起傳遞。如果不能均勻平分 CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS，則不會使用剩下的計算器件。	
CL_DEVICE_PARTITION_BY_COUNTS	unsigned int
此屬性後面跟着的是計算器件數目清單，以 CL_DEVICE_PARTITION_BY_COUNTS_LIST_END 終止。對於清單中每個非零的 m，都會創建一個具有 m 個計算器件的子設備。 CL_DEVICE_PARTITION_BY_COUNTS_LIST_END的值是 0。清單中非零值的數目不能超過 CL_DEVICE_PARTITION_MAX_SUB_DEVICES。計算器件的總數不能超過 CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS。	
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN	cl_device_affinity_domain
將设备分割成許多小的设备集合，每個集合中包含一個或多個計算器件。他們共享部分緩存體系。跟隨此屬性的值從下列值中選取： • CL_DEVICE_AFFINITY_DOMAIN_NUMA，子設備中的計算器件之間共享一個 NUMA 節點。 • CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE，子設備中的計算器件之間共享一個 4 級數據緩存。 • CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE，子設備中的計算器件之間共享一個 3 級數據緩存。 • CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE，子設備中的計算器件之間共享一個 2 級數據緩存。 • CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE，子設備中的計算器件之間共享一個 1 級數據緩存。 • CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE，將设备按下一個可再分的相似域進行分割。實作會判定此设备或者子設備可以怎樣進一步細分，按 NUMA、L4、L3、L2、L1 的順序，取最前面的那個。對於所劃分成的子設備，其中的計算器件間共享這一級的內存子系統。 用戶可以通過對子設備調用 **clGetDeviceInfo** (CL_DEVICE_PARTITION_TYPE) 來確定發生了什麼。	

num_devices 是 *out_devices* 指向的內存塊所能容納 cl_device_id 的數目。

out_devices 是一塊緩衝區，用來存儲所返回的 OpenCL 子設備。如果 *out_devices* 是 NULL，則忽略。否則， *num_devices* 必須大於等於（按照 *properties* 所指定的劃分策略）所劃分成的子設備的數目。

num_devices_ret 返回（按照 *properties* 所指定的劃分策略） *device*可以劃分成子設備的數目。如果 *num_devices_ret* 是 NULL，則忽略。

如果劃分成功， **clCreateSubDevices** 會返回 CL_SUCCESS。否則，返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_DEVICE，如果 *in_devices* 無效。
- CL_INVALID_VALUE，如果 *properties* 中的值無效，或者雖然有效但是此设备不支持。
- CL_INVALID_VALUE，如果 *out_devices* 不是 NULL，且 *num_devices* 小於所創建子設備的數目。
- CL_DEVICE_PARTITION_FAILED，如果實作支持此劃分方式，但是 *in_device* 不能再細分了。
- CL_INVALID_DEVICE_PARTITION_COUNT，如果 *properties* 所指定的劃分策略是 CL_DEVICE_PARTITION_BY_COUNTS，且所要求的子設備數目超過了 CL_DEVICE_PARTITION_MAX_SUB_DEVICES，或者所要求的計算器件總數超過了 *in_device* 的 CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS，或者所要求的某個子設備中計算器件的數目小於 0 或者超過了 *in_device* 的 CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

下面就如何為 **clCreateSubDevices** 指定參數 *properties* 給出了一些示例：

要將一個包含 16 個計算器件的设备劃分成兩個子設備，每個包含 8 個計算器件，這樣指定 *properties*：

```
1          { CL_DEVICE_PARTITION_EQUALLY, 8, 0 }
```

要將一個包含 4 個計算器件的设备劃分成兩個子設備，其中一個子設備包含 3 個計算器件，另一個只包含 1 個計算器件，這樣指定 *properties*：

```
1          { CL_DEVICE_PARTITION_BY_COUNTS,
2            3, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0 }
```

按最外層的緩存列（如果有的話）劃分设备，這樣指定 *properties*：

```
1          { CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN,
2            CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE,
3            0 }
```

clRetainDevice

```
cl_int clRetainDevice (cl_device_id device)
```

如果 *device* 是通過調用 **clCreateSubDevices** 創建的一個子設備，則 **clRetainDevice** 會使 *device* 的引用計數增一。如果 *device* 是一個根設備，即由 **clGetDeviceIDs** 所返回的一個 cl_device_id，則 *device* 的引用計數保持不變。如果執行成功或者 *device* 是一個根設備， **clRetainDevice** 會返回 CL_SUCCESS，否則，返回下列錯誤碼之一：

- CL_INVALID_DEVICE，如果 *device* 是一個無效的子設備。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clReleaseDevice

```
cl_int clReleaseDevice (cl_device_id device)
```

如果 *device* 是通過調用 **clCreateSubDevices** 創建的一個子設備，則 **clReleaseDevice** 會使 *device* 的引用計數減一。如果 *device* 是一個根設備，即 **clGetDeviceIDs** 所返回的一個 cl_device_id，則 *device* 的引用計數保持不變。如果執行成功，**clReleaseDevice** 會返回 CL_SUCCESS，否則，返回下列錯誤碼之一：

- CL_INVALID_DEVICE，如果 *device* 是一個無效的子設備。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

在 *device* 的引用計數降為 0 並且所有附着其上的對象（如命令隊列）都被釋放後，就會刪除 *device* 對象。

节 4.4 上下文

clCreateContext

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

此函式可以創建一個 OpenCL 上下文。OpenCL 上下文是在一個或多個设备上創建的。OpenCL 運行時使用上下文來管理對象（如命令隊列、內存對象、程式對象以及內核對象）；並在上下文中所指定的设备上執行內核。

properties 指定了上下文的一系列屬性名和對應的值。每個屬性名後面緊跟相應的期望值。此清單以 0 終止。表 4.5列出了所支持的屬性。如果 *properties* 是 NULL，選擇哪個平台依賴於具體實作。

表 4.5 clCreateContext 所支持的屬性清單

cl_context_properties	屬性值
CL_CONTEXT_PLATFORM	cl_platform_id
指定要使用哪個平台。	
CL_CONTEXT_INTEROP_USER_SYNC	cl_bool
OpenCL 和其他 API 之間的同步是否由用戶自己負責。對於其使用限制，請參考《OpenCL 1.2擴展規範》中相應章節。 如果沒有指定此屬性，則缺省值為 CL_FALSE。	

num_devices 是參數 *devices* 中设备的數目。

devices 指向一個设备清單，其中的設備都是唯一的[4]，都是由 **clGetDeviceIDs** 所返回的，或者是用 **clCreateSubDevices** 創建的子設備。

pfn_notify 是應用所註冊的回調函式。對於此上下文，無論創建時還是運行時，只要發生了錯誤，OpenCL 的實作都會調用此函式，但調用可能是異步的。應用需要保證此函式是線程安全的。此函式的參數為：

- *errinfo* 指向一個錯誤字串。
- *private_info* 指向一塊二進制數據，其大小為 *cb*，這塊數據由 OpenCL 實作所返回，可用來記錄一些附加資訊來幫助調試錯誤。
- *user_data* 指向用戶提供的數據。

如果 *pfn_notify* 是 NULL，則表示不註冊回調函式。

[4] *devices* 中重複的设备會被忽略。

很多情況下，即使在上下文外發生了錯誤，也需要發出錯誤通告。這種情況 *pfn_notify* 就無能為力了，怎樣發出這種錯誤通告依賴於具體實作。

user_data 會在調用 *pfn_notify* 時作為引數 *user_data* 使用。 *user_data* 可以是 NULL。

errcode_ret 用來返回相應錯誤碼。如果 *errcode_ret* 是 NULL，則不會返回錯誤碼。

如果成功創建了上下文，**clCreateContext** 會將其返回（非零），並將 *errcode_ret* 置為 CL_SUCCESS。否則返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_PLATFORM，如果 *properties* 是 NULL 且沒有可選的平台，或者 *properties* 中平台的值無效。
- CL_INVALID_PROPERTY，如果 *properties* 中的屬性名不受支持，或者支持此屬性但其值無效，或者同一屬性名出現多次。
- CL_INVALID_VALUE，如果 *devices* 是 NULL。
- CL_INVALID_VALUE，如果 *num_devices* 等於零。
- CL_INVALID_VALUE，如果 *pfn_notify* 是 NULL 但 *user_data* 不是 NULL。
- CL_INVALID_DEVICE，如果 *devices* 中有無效的设备。
- CL_DEVICE_NOT_AVAILABLE，如果 *devices* 中的某個设备當前不可用，即使此设备是由 **clGetDeviceIDs** 返回的。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clCreateContextFromType

```
cl_context clCreateContextFromType(
                const cl_context_properties *properties,
                cl_device_type device_type,
                void (CL_CALLBACK *pfn_notify)(
                        const char *errinfo,
                        const void *private_info,
                        size_t cb,
                        void *user_data),
                void *user_data,
                cl_int *errcode_ret)
```

函数 **clCreateContextFromType**[5]會由特定類型的设备創建一個 OpenCL 上下文，此上下文不會引用由這些设备所創建的子設備。

properties 指定了上下文的一系列屬性及其相應的值。每個屬性名後面緊跟其對應的期望值。表 4.5列出了所支持的屬性。如果 *properties* 是 NULL，選擇哪個平台依賴於具體實作。

device_type 是位欄，用來標識设备類型，參見中的表 4.2。

pfn_notify 和 *user_data* 跟 **clCreateContext** 中所描述的一樣。

errcode_ret 用來返回相應的錯誤碼，如果 *errcode_ret* 是 NULL，則不返回錯誤碼。

如果成功創建了上下文，**clCreateContextFromType** 會將其返回，並將 *errcode_ret* 置為 CL_SUCCESS。否則返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_PLATFORM，如果 *properties* 是 NULL 並且沒有平台可選，或者 *properties* 中平台的值無效。
- CL_INVALID_PROPERTY，如果 *properties* 中的上下文屬性名不受支持，或者支持此屬性但其值無效，或者同一屬性名出現多次。
- CL_INVALID_VALUE，如果 *pfn_notify* 是 NULL 但 *user_data* 不是 NULL。
- CL_INVALID_DEVICE_TYPE，如果 *device_type* 的值無效。
- CL_DEVICE_NOT_AVAILABLE，如果當前沒有同時符合 *device_type* 以及 *properties* 中屬性值的设备可用。

[5] **clCreateContextfromType** 可能返回平台中現有符合 *device_type* 的所有實際物理設備，也可能只返回其中一個子集。

- CL_DEVICE_NOT_FOUND，如果沒有找到同時符合 *device_type* 以及 *properties* 中的屬性值的设备。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clRetainContext

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

函式 **clRetainContext** 會使 *context* 的引用計數增一。

如果執行成功，**clRetainContext** 會返回 CL_SUCCESS。否則返回下列錯誤碼之一:

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clCreateContext 和 **clCreateContextFromType** 會實施隱式的保留。這對第三方庫非常有用，這樣應用就可以將上下文傳給他們使用。然而，應用可能會在沒有通知他們的情況下刪除此上下文。通過使用函式保留或釋放上下文，在庫所使用的上下文不再有效時就不會出問題。

clReleaseContext

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

函式 **clReleaseContext** 會使 *context* 的引用計數減一。

如果執行成功，**clReleaseContext** 會返回CL_SUCCESS。否則返回下列錯誤碼之一:

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

如果 *context* 的引用計數變成了零，且所有附着其上的對象（如內存對象、命令隊列）都被釋放了的時候，*context* 就會被刪除。

clGetContextInfo

```
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)
```

函式 **clGetContextInfo** 可用來查詢上下文的相關資訊。

context 指定要查詢哪個 OpenCL 上下文。

param_name 是枚舉常量，指定要查詢什麼資訊。

param_value 指向的內存用來存儲查詢結果。如果 *param_value* 是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小。必須大於等於*表 4.6*中所列返回型別的大小。

param_value_size_ret 會返回 *param_value* 所存查詢結果的實際字節數。如果 *param_value_size_ret* 是 NULL，則忽略。

*表 4.6*中列出了所支持的 *param_name* 和 **clGetContextInfo** 返回的 *param_value* 中的資訊。

如果執行成功，**clGetContextInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_CONTEXT，如果 *context* 不是一個有效的 OpenCL 上下文。
- CL_INVALID_VALUE，如果 *param_name* 的值不受支持，或者 *param_value_size* 的值 < *表 4.6*中返回型別的大小且 *param_value* 不是 NULL。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

表 4.6 **clGetContextInfo** 所支持的 *param_names*

cl_context_info	返回型別
CL_CONTEXT_REFERENCE_COUNT	cl_unit
返回 *context* 的引用計數[1]。	
CL_CONTEXT_NUM_DEVICES	cl_unit
返回 *context* 中设备的數目。	
CL_CONTEXT_DEVICES	cl_device_id[]
返回 *context* 中设备的清單。	
CL_CONTEXT_PROPERTIES	cl_context_properties[]
返回調用 **clCreateContext** 或 **clCreateContextFromType** 時所指定的引數 *properties*。 對於調用 **clCreateContext** 或 **clCreateContextFromType** 創建 *context* 時所指定的引數 *properties* 而言，如果此引數不是 NULL，實作必須返回此引數的值。而如果此引數是 NULL，實作可以選擇將 *param_value_size_ret* 置為 0，即没有返回属性值，也可以將 *param_value* 的內容置為 0（0 用作上下文屬性清單的終止標記）。	

[1] 在返回的那一刻，此引用計數就已過時。應用中一般不太適用。提供此特性主要是為了檢測內存泄漏。

第 5 章
OpenCL runtime

本節將要介紹的 API 可用來管理 OpenCL 對象（如命令隊列、內存對象、程式對象、內核對象）或者將命令插入命令隊列（如執行內核，讀寫內存對象）。

节 5.1 命令隊列

OpenCL 對象，如內存對象、程式對象和內核對象都是用上下文創建的。對這些對象的操控都是通過命令隊列實施的。用命令隊列可以將一系列操作（叫做命令）按序排隊。如果有多個命令隊列，應用可以將多個相互獨立的命令分別排隊而無須同步。然而這僅在沒有共享任何對象時才成立。如果要在多個命令隊列間共享對象，就要求應用實施相應的同步。這在附錄 *A*中會有所描述。

clCreateCommandQueue

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

函式 **clCreateCommandQueue** 可用來在某個设备上創建命令隊列。

context 必須是一個有效的 OpenCL 上下文。

表 5.1 cl_command_queue_property 的有效值及其描述

命令隊列屬性
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
用來確定命令隊列中的命令是順序執行還是亂序執行。如果設置了此屬性，就亂序執行，否則順序執行。 詳情請參考節 *5.11*。
CL_QUEUE_PROFILING_ENABLE
使能或去能對命令的評測（profiling）。如果設置了此屬性，則使能，否則去能。 詳情請參考節 *5.12*。

device 必須是與 *context* 關聯的设备。他要麼是用 **clCreateContext** 創建 *context* 時所指定的设备清單中的一個，要麼其類型與用 **clCreateContextFromType** 創建 *context* 時所指定的设备類型相同。

properties 指定了命令隊列的一系列屬性。他是位欄，參見表 *5.1*。其值只能從表 *5.1*所列屬性中選取，否則無效。

errcode_ret 用來返回錯誤碼。當然如果 *errcode_ret* 是 NULL，就不會返回錯誤碼了。

如果成功創建了命令隊列，則 **clCreateCommandQueue** 會將其返回，同時將 *errcode_ret* 置為 CL_SUCCESS。否則返回 NULL，同時將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_INVALID_DEVICE，如果 *device* 無效或未與 *context* 關聯。
- CL_INVALID_VALUE，如果 *properties* 的值有效，但是此设备不支持。
- CL_INVALID_QUEUE_PROPERTIES，如果 *properties* 本身沒問題，但 *device* 不支持。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clRetainCommandQueue

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

此函式會使 *command_queue* 的引用計數增一。

如果執行成功，**clRetainCommandQueue** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clCreateCommandQueue 會實施隱式的 Retain。這對第三方庫非常有用，這樣應用可以將命令隊列傳給他們使用。然而，應用可能在沒有通知庫的情況下刪除命令隊列。通過保留或釋放命令隊列，在庫所使用的命令隊列不再有效時就不會出現問題。

clReleaseCommandQueue

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

此函式會使 *command_queue* 的引用計數減一。

如果執行成功，**clReleaseCommandQueue** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

当 *command_queue* 的引用計數降為 0，並且其中的所有命令全部執行完畢（如執行內核、更新內存對象等）時，此命令隊列就會被刪除。

clReleaseCommandQueue會實施隱式的刷新（flush），這會觸發所有之前入隊的 OpenCL 命令。

clGetCommandQueueInfo

```
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)
```

此函式可用來查詢命令隊列的資訊。

command_queue 指定要查詢哪個命令隊列。

param_name 指定要查詢什麼資訊。

param_value 所指內存用來存儲查詢結果。如果是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小（單位：字節）。其值必須 >= 表 5.2中返回型別的大小。如果 *param_value* 是 NULL，則將其忽略。

param_value_size_ret 會返回查詢結果的實際大小。如果是 NULL，則忽略。

clGetCommandQueueInfo 所支持的 *param_name* 以及 *param_value* 中所返回的資訊如表 5.2所示。

表 5.2 **clGetCommandQueueInfo所支持的***param_names*

cl_command_queue_info	**返回类型**
CL_QUEUE_CONTEXT	cl_context
返回創建命令隊列時所指定的上下文。	
CL_QUEUE_DEVICE	cl_device_id
返回創建命令隊列時所指定的设备。	
CL_QUEUE_REFERENCE_COUNT	cl_uint
返回命令隊列的引用計數[1]。	
CL_QUEUE_PROPERTIES	cl_command_queue_properties
返回命令隊列當前的屬性。這些屬性就是 **clCreateCommandQueue** 的引數 *properties*。	

[1] 在返回的那一刻，此引用計數就已過時。應用中一般不太適用。提供此特性主要是為了檢測內存泄漏。

如果執行成功，**clGetCommandQueueInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_VALUE，如果 *param_name* 不在支持之列，或者 *param_value_size* 的值 < *表 5.2*中返回型別的大小且 *param_value* 不是 NULL。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

> 如果創建了上下文和命令隊列並且已經有命令入隊了，而這時他們所使用的设备可能變的不可用了。這種情況下，對於使用這個上下文（和命令隊列）的 *OpenCL API* 而言，其行為依賴於具體實作。在设备變的不可用時，如果創建上下文時用戶指定了回調函式，則可以在傳遞給他的引數 *errinfo*、*private_info* 中記錄相應的資訊。

节 5.2 緩衝對象

緩衝對象中的元素按一維存儲，他們可以是標量數據型別（如 int、float ）、矢量數據型別，或者用戶自定義的結構體。

5.2.1 創建緩衝對象

clCreateBuffer

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

此函式可用來創建**緩衝對象**。

context 是 OpenCL 上下文，用來創建緩衝對象。

flags 是位欄，用來表明如何分配（如使用哪個內存區域分配緩衝對象）以及怎樣使用。*表 5.3*中列出了 *flags* 可能的值。如果 *flags* 的值是 0，則使用缺省值 CL_MEM_READ_WRITE。

表 5.3α *cl_mem_flags*的 值

cl_mem_flags
CL_MEM_READ_WRITE
表明此內存對象可讀可寫。這是缺省值。
CL_MEM_WRITE_ONLY
只能寫不能讀，對這種內存對象的讀操作是未定義的。 CL_MEM_READ_WRITE 和 CL_MEM_WRITE_ONLY 是互斥的。
CL_MEM_READ_ONLY
只能讀不能寫，對這種內存對象的寫操作是未定義的。 CL_MEM_READ_WRITE 或者 CL_MEM_WRITE_ONLY 都與 CL_MEM_READ_ONLY 互斥。
CL_MEM_USE_HOST_PTR
僅當 *host_ptr* 不是 NULL 時才有效。他表明應用想讓 OpenCL 實作使用 *host_ptr* 所引用的內存來存儲內存對象的內容。 OpenCL 實作可以在设备內存中保存一份 *host_ptr* 所引用的內容用作緩存。內核在设备上執行時可以使用這份拷貝。 如果多個緩衝對象由同一 *host_ptr* 創建，或者有重疊區域，那麼 OpenCL 命令操作這些緩衝對象時，其結果未定義。 用 CL_MEM_USE_HOST_PTR 創建內存對象時， *host_ptr* 的對齊規則請參考 附錄 *4.3*。
CL_MEM_ALLOC_HOST_PTR
表明應用想讓 OpenCL 實作在主機可以訪問的內存中分配內存。 他與 CL_MEM_USE_HOST_PTR 互斥。
CL_MEM_COPY_HOST_PTR

表 5.3β *cl_mem_flags*的 值

僅當 *host_ptr* 不是 NULL 時才有效。他表明應用想讓 OpenCL 實作使用 *host_ptr* 所引用的內存來為內存對象分配內存並拷貝數據。 他與 CL_MEM_USE_HOST_PTR 互斥。 他與 CL_MEM_ALLOC_HOST_PTR 一起使用時，可以對由主機可訪問內存（如 PCIe ）分配的 cl_mem 對象進行初始化。
CL_MEM_HOST_WRITE_ONLY
表明主機只會對此內存對象進行寫入。可用來對主機的寫操作進行優化（如對於通過系統總線如 PCIe 與主機進行通信的设备，分配內存對象時使能 Write-combining）。
CL_MEM_HOST_READ_ONLY
表明主機只會對此內存對象進行讀取。 他與 CL_MEM_HOST_WRITE_ONLY 互斥。
CL_MEM_HOST_NO_ACCESS
表明主機不會對此內存對象進行讀寫。 CL_MEM_HOST_WRITE_ONLY 或者 CL_MEM_HOST_READ_ONLY 都與 CL_MEM_HOST_NO_ACCESS 互斥。

size 即所要分配緩衝對象的大小，單位：字節。

host_ptr 指向可能已經由應用分配好了的緩衝數據。其大小必須 >= *size*。

errcode_ret 用來返回錯誤碼。如果是 NULL，則不會返回錯誤碼。

如果成功創建了緩衝對象，**clCreateBuffer** 會將其返回，並將 *errcode_ret* 置為 CL_SUCCESS。否則返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_INVALID_VALUE，如果 *flags* 的值不是表 5.3中定義的。
- CL_INVALID_BUFFER_SIZE，如果 *size* 是 0[6]。
- CL_INVLAID_HOST_PTR，如果 *host_ptr* 是 NULL，但是 *flags* 中设置了 CL_MEM_USE_HOST_PTR 或 CL_MEM_COPY_HOST_PTR；或者 *host_ptr* 不是 NULL，但是 *flags* 中沒有設置 CL_MEM_USE_HOST_PTR 或 CL_MEM_COPY_HOST_PTR。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為緩衝對象分配內存失敗。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clCreateSubBuffer

```
cl_mem clCreateSubBuffer(
                cl_mem buffer,
                cl_mem_flags flags,
                cl_buffer_create_type buffer_create_type,
                const void *buffer_create_info,
                cl_int *errcode_ret)
```

此函式可以由一個現有的緩衝對象創建一個新的緩衝對象（叫做子緩衝對象，sub-buffer object）。

buffer 必須是一個有效的緩衝對象，且不能是子緩衝對象。

flags 是位欄，用來表明如何分配以及怎樣使用內存對象，參見表 5.3。如果 *flags* 中沒有設置 CL_MEM_READ_WRITE、CL_MEM_READ_ONLY 或 CL_MEM_WRITE_ONLY，則會從 *buffer* 中繼承這些屬性。而 *flags* 中不能設置 CL_MEM_USE_HOST_PTR、CL_MEM_ALLOC_HOST_PTR 和 CL_MEM_COPY_HOST_PTR，這些也會由 *buffer* 繼承。即使 *buffer* 的內存訪問限定符中有 CL_MEM_COPY_HOST_PTR，也並不意味着創建 sub-buffer 時會有額外的拷貝。如果 *flags* 中沒有設置 CL_MEM_HOST_WRITE_ONLY、CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_NO_ACCESS，則會從 *buffer* 中繼承這些屬性。

buffer_create_type 和 *buffer_create_info* 表明了所要創建的緩衝對象的類型。表 5.4中列出了所支持的 *buffer_create_type* 以及 *buffer_create_info* 中對應的內容。

[6] 如果 *size* 比 *context* 中所有设备的 CL_DEVICE_MAX_MEM_ALLOC_SIZE（参见表 4.3）都大，實作可能返回 CL_INVALID_BUFFER_SIZE。

表 5.4 **clCreateSubBuffer 所支持的創建類型**

cl_buffer_create_type
CL_BUFFER_CREATE_TYPE_REGION

用 *buffer* 中的特定區域創建緩衝對象。
buffer_create_info 指向如下數據結構:
 	
`struct _cl_buffer_region {`
`  size_t origin;`
`  size_t size;`
`} cl_buffer_region;`
 	
(*origin*, *size*) 就是在 *buffer* 中的偏移量和字節數。
如果 *buffer* 是用 CL_MEM_USE_HOST_PTR 創建的, 所返回緩衝對象的 *host_ptr* 就是 *host_ptr* + *origin*。
所返回的緩衝對象引用了為 *buffer* 分配的數據存儲空間, 並指向其中的特定區域 (*origin*, *size*)。
如果在 *buffer* 中, 區域 (*origin*, *size*) 越界了, 則會在 *errcode_ret* 中返回 CL_INVALID_VALUE。
如果 *size* 是 0, 則返回 CL_INVALID_BUFFER_SIZE。
如果與 *buffer* 相關聯的上下文中沒有一個設備的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 與 *origin* 對齊, 則會在 *errcode_ret* 中返回 CL_MISALIGNED_SUB_BUFFER_OFFSET。

如果執行成功, **clCreateSubBuffer** 會返回 CL_SUCCESS。否則會將 *errcode_ret* 置為下列錯誤碼之一:

- CL_INVALID_MEM_OBJECT, 如果 *buffer* 無效或者是一個子緩衝對象。
- CL_INVALID_VALUE, 如果 *buffer* 是用 CL_MEM_WRITE_ONLY 創建的, 但 *flags* 中設置了 CL_MEM_READ_WRITE 或 CL_MEM_READ_ONLY; 或者 *buffer* 是用 CL_MEM_READ_ONLY 創建的, 但 *flags* 中設置了 CL_MEM_READ_WRITE 或 CL_MEM_WRITE_ONLY; 或者 *flags* 中設置了 CL_MEM_USE_HOST_PRT、CL_MEM_ALLOC_HOST_PTR 或 CL_MEM_COPY_HOST_PTR。
- CL_INVALID_VALUE, 如果 *buffer* 是用 CL_MEM_HOST_WRITE_ONLY 創建的, 但 *flags* 中設置了 CL_MEM_HOST_READ_ONLY; 或者 *buffer* 是用 CL_MEM_HOST_READ_ONLY 創建的, 但 *flags* 中設置了 CL_MEM_HOST_WRITE_ONLY; 或者 *buffer* 是用 CL_MEM_HOST_NO_ACCESS 創建的, 但 *flags* 中設置了 CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_WRITE_ONLY。
- CL_INVALID_VALUE, 如果 *buffer_create_type* 的值無效。
- CL_INVALID_VALUE, 如果 *buffer_create_info* 中的值無效 (對 *buffer_create_type* 而言), 或者 *buffer_create_info* 是 NULL。
- CL_INVALID_BUFFER_SIZE, 如果 *size* 是 0。
- CL_MEM_OBJECT_ALLOCATION_FAILURE, 如果為子緩衝對象分配內存失敗。
- CL_OUT_OF_RESOURCES, 如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY, 如果在為主機上的 OpenCL 實作分配資源時失敗。

> 對一個緩衝對象及其子緩衝對象的並發讀、寫、拷貝是未定義的。對於由同一緩衝對象創建的互相重疊的子緩衝對象的並發讀、寫、拷貝也是未定義的。只有讀操作是定義了的。

5.2.2 讀、寫以及拷貝緩衝對象

clEnqueueReadBuffer

clEnqueueWriteBuffer

```
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)
```

這兩個函式會將一個用於讀寫緩衝對象的命令入隊，此命令可以將緩衝對象的內容讀到主機內存中，或者由主機內存寫入到緩衝對象中。

command_queue 就是命令要進入的隊列。 *command_queue* 和 *buffer* 必須是由同一個 OpenCL 上下文創建的。

buffer 是一個緩衝對象。

blocking_read 和 *blocking_write* 表明讀寫操作是阻塞的還是非阻塞的。

如果 *blocking_read* 是 CL_TRUE，即讀命令是阻塞的，直到 *buffer* 中的數據完全拷貝到 *ptr* 所指內存中後，**clEnqueueReadBuffer** 才會返回。

如果 *blocking_read* 是 CL_FALSE，即讀命令是非阻塞的，**clEnqueueReadBuffer** 將此命令入隊後就會返回。只有等到讀命令執行完畢，才能繼續使用 *ptr* 所指向的內容。參數 *event* 會返回一個事件對象，可用來查詢讀命令的執行情況。讀命令完成後，應用就可以繼續使用 *ptr* 所指向的內容了。

如果 *blocking_write* 是 CL_TRUE，OpenCL 實作會拷貝 *ptr* 所指向的數據，並將一個寫命令入隊。當 **clEnqueueWriteBuffer** 返回後，應用就可以繼續使用 *ptr* 所指向的內存了。

如果 *blocking_write* 是 CL_FALSE，OpenCL 實作會用 *ptr* 實施非阻塞的寫操作。既然非阻塞，實作就可以立即返回。返回後，應用還不能立刻就使用 *ptr* 所指內存。參數 *event* 會返回一個事件對象，可用來查詢寫命令的執行情況。寫命令完成後，應用就可以重新使用 *ptr* 所指向的內存了。

offset 是讀寫區域在緩衝對象中的偏移量，單位字節。

cb 是要讀寫數據的大小，單位字節。

ptr 指向主機中的一塊內存，用作讀命令的數據源以及寫命令的目的地。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。 *event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，就可以回收並重新使用 *event_wait_list* 所關聯的內存了。

event 會返回一個事件對象，用來標識此讀、寫命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueReadBuffer** 和 **clEnqueueWriteBuffer** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *buffer* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *buffer* 無效。
- CL_INVALID_VALUE，如果 (*offset*, *size*) 所指定的區域越限，或者 *ptr* 是 NULL，或者 *size* 是 0。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項：

  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *buffer* 是子緩衝對象，且創建此對象時所指定的 *offset* 沒有與 *queue* 所關聯设备的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST，如果讀寫操作是阻塞的，且 *event_wait_list* 中任一命令的執行狀態是負整數。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *buffer* 分配內存失敗。
- CL_INVALID_OPERATION，如果調用的是 **clEnqueueReadBuffer**，但創建 *buffer* 時指定了 CL_MEM_HOST_WRITE_ONLY 或 CL_MEM_HOST_NO_ACCESS。
- CL_INVALID_OPERATION，如果調用的是 **clEnqueueWriteBuffer**，但創建 *buffer* 時指定了 CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_NO_ACCESS。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clEnqueueReadBufferRect

clEnqueueWriteBufferRect

```
cl_int clEnqueueReadBufferRect (
                          cl_command_queue command_queue,
                          cl_mem buffer,
                          cl_bool blocking_read,
                          const size_t *buffer_origin,
                          const size_t *host_origin,
                          const size_t *region,
                          size_t buffer_row_pitch,
                          size_t buffer_slice_pitch,
                          size_t host_row_pitch,
                          size_t host_slice_pitch,
                          void *ptr,
                          cl_uint num_events_in_wait_list,
                          const cl_event *event_wait_list,
                          cl_event *event)

cl_int clEnqueueWriteBufferRect (
                          cl_command_queue command_queue,
                          cl_mem buffer,
                          cl_bool blocking_write,
                          const size_t *buffer_origin,
                          const size_t *host_origin,
                          const size_t *region,
                          size_t buffer_row_pitch,
                          size_t buffer_slice_pitch,
                          size_t host_row_pitch,
                          size_t host_slice_pitch,
                          const void *ptr,
                          cl_uint num_events_in_wait_list,
                          const cl_event *event_wait_list,
                          cl_event *event)
```

這兩個函式所入隊的命令可以讀寫緩衝對象中 2D 或 3D 矩形區域。

command_queue 就是命令要進入的隊列。*command_queue* 和 *buffer* 必須是由同一個 OpenCL 上下文創建的。

buffer 是一個緩衝對象。

blocking_read 和 *blocking_write* 表明讀寫操作是阻塞的還是非阻塞的。

如果 *blocking_read* 是 CL_TRUE，即讀命令是阻塞的，直到 *buffer* 中的數據完全拷貝到 *ptr* 所指內存中後，**clEnqueueReadBufferRect** 才會返回。

如果 *blocking_read* 是 CL_FALSE，即讀命令是非阻塞的，**clEnqueueReadBufferRect** 將此命令入隊後就會返回。只有等到讀命令執行完畢，才能繼續使用 *ptr* 所指向的內容。參數 *event* 會返回一個事件對象，可用來查詢讀命令的執行情況。讀命令完成後，應用就可以繼續使用 *ptr* 所指向的內容了。

如果 *blocking_write* 是 CL_TRUE，OpenCL 實作會拷貝 *ptr* 所指向的數據，並將一個寫命令入隊。當 **clEnqueueWriteBufferRect** 返回後，應用就可以繼續使用 *ptr* 所指向的內存了。

如果 *blocking_write* 是 CL_FALSE，OpenCL 實作會用 *ptr* 實施非阻塞的寫操作。既然非阻塞，實作就可以立即返回。返回後，應用還不能立刻就使用 *ptr* 所指內存。參數 *event* 會返回一個事件對象，可用來查詢寫命令的執行情況。寫命令完成後，應用就可以重新使用 *ptr* 所指向的內存了。

buffer_origin 定義了要讀寫的內存區域在 *buffer* 中的偏移量 (x, y, z)。對於 2D 矩形區域，z 的值即 *buffer_origin*[2] 應該是 0。偏移量這樣計算:

$$\begin{aligned} &\mathit{buffer\_origin}[2] * \mathit{buffer\_slice\_pitch} \\ +\ &\mathit{buffer\_origin}[1] * \mathit{buffer\_row\_pitch} \\ +\ &\mathit{buffer\_origin}[0] \end{aligned}$$

host_origin 定義了要讀寫的內存區域在 *ptr* 中的偏移量 (x, y, z)。對於 2D 矩形區域，z 的值即 *host_origin*[2] 應該是 0。偏移量這樣計算:

$$\begin{aligned} &\mathit{host\_origin}[2] * \mathit{host\_slice\_pitch} \\ +\ &\mathit{host\_origin}[1] * \mathit{host\_row\_pitch} \\ +\ &\mathit{host\_origin}[0] \end{aligned}$$

region 定義了要讀寫的 2D 或 3D 區域: $(width, height, depth)$，其單位分別是 byte、row、slice。對於 2D 區域，*depth* 的值，即 *region*[2] 應該是 1。

buffer_row_pitch 是 *buffer* 中每一行數據的字節數。如果 *buffer_row_pitch* 是 0，則在計算偏移量時用 *region*[0] 替代。

buffer_slice_pitch 是 *buffer* 中每個 2D 平面（slice）的字節數。如果 *buffer_slice_pitch* 是 0，則在計算偏移量時用 *region*[1] * *buffer_row_pitch* 替代。

host_row_pitch 是 *ptr* 中每一行數據的字節數。如果 *host_row_pitch* 是 0，則在計算偏移量時用 *region*[0] 替代。

host_slice_pitch 是 *ptr* 中每個 2D 平面（slice）的字節數。如果 *host_slice_pitch* 是 0，則在計算偏移量時用 *region*[1] * *host_row_pitch* 替代。

ptr 指向主機中的一塊內存，用作讀命令的數據源以及寫命令的目的地。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，就可以回收並重新使用 *event_wait_list* 所關聯的內存了。

event 會返回一個事件對象，用來標識此讀、寫命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueReadBufferRect** 和 **clEnqueueWriteBufferRect** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *buffer* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *buffer* 無效。
- CL_INVALID_VALUE，如果下列區域越限:

(*buffer_origin*, *region*, *buffer_row_pitch*, *buffer_slice_pitch*)

- CL_INVALID_VALUE，如果 *ptr* 是 NULL。
- CL_INVALID_VALUE，如果 *region* 中的任何元素是 0。
- CL_INVALID_VALUE，如果 *buffer_row_pitch* 不是 0 且小於 *region*[0]。
- CL_INVALID_VALUE，如果 *host_row_pitch* 不是 0 且小於 *region*[0]。
- CL_INVALID_VALUE，如果 *buffer_slice_pitch* 滿足下列所有條件:
  - 不是 0、
  - 小於 *region*[1] * *buffer_row_pitch*、
  - 且不是 *buffer_row_pitch* 的整數倍。
- CL_INVALID_VALUE，如果 *host_slice_pitch* 滿足下列條件:
  - 不是 0、
  - 小於 *region*[1] * *host_row_pitch*、
  - 且不是 *host_row_pitch* 的整數倍。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *buffer* 是子緩衝對象，且創建此對象時所指定的 *offset* 沒有與 *command_queue* 所關聯設备的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST，如果讀寫操作是阻塞的，且 *event_wait_list* 中任何命令的執行狀態是負整數。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *buffer* 分配內存失敗。
- CL_INVALID_OPERATION，如果調用的是 **clEnqueueReadBufferRect**，但創建 *buffer* 時指定了 CL_MEM_HOST_WRITE_ONLY 或 CL_MEM_HOST_NO_ACCESS。
- CL_INVALID_OPERATION，如果調用的是 **clEnqueueWriteBufferRect**，但創建 *buffer* 時指定了 CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_NO_ACCESS。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

如果調用 **clEnqueueReadBuffer** 時將引數 *ptr* 設置為 *host_ptr* + *offset*（其中 *host_ptr* 是用 CL_MEM_USE_HOST_PTR 創建所要讀取的緩衝對象時指定的），為避免未定義的行為，必須滿足下列要求:

- 在讀命令開始執行前，所有使用此緩衝對象（或由此緩衝對象創建的內存對象，包括緩衝對象和圖像對象）的命令都已經執行完畢。
- 所有用此緩衝對象創建的緩衝對象或圖像對象都沒有被映射。
- 在讀命令執行完之前，所有用此緩衝對象創建的緩衝對象或圖像對象都沒有被任何命令隊列使用。

如果調用 **clEnqueueReadBufferRect** 時將引數 *ptr* 設置為 *host_ptr*，並且 *host_origin* 和 *buffer_origin* 相同，其中 *host_ptr* 就是用 CL_MEM_USE_HOST_PTR 創建所要讀取的緩衝對象時指定的，要滿足的要求跟 **clEnqueueReadBuffer** 一樣。

如果調用 **clEnqueueWriteBuffer** 時將引數 *ptr* 置為 *host_ptr* + *offset*，其中 *host_ptr* 就是用 CL_MEM_USE_HOST_PTR 創建所要寫入的緩衝對象時指定的，為避免未定義的行為，必須滿足下列要求:

- 在寫命令開始執行前，(*host_ptr* + *offset*, *cb*) 所劃定的主機內存區域必須含有最新的數據。
- 所有用此緩衝對象創建的緩衝對象或圖像對象都沒有被映射。
- 在寫命令執行完之前，所有用此緩衝對象創建的緩衝對象或圖像對象都沒有被任何命令隊列使用。

如果調用 **clEnqueueWriteBufferRect** 時將引數 *ptr* 設置為 *host_ptr*，並且 *host_origin* 和 *buffer_origin* 相同，其中 *host_ptr* 就是用 CL_MEM_USE_HOST_PTR 創建所要讀取的緩衝對象時指定的，為避免未定義的行為，必須滿足下列要求:

- 在寫命令開始執行前，(*buffer_origin*, *region*) 所劃定的主機內存區域必須含有最新的數據。
- 所有用此緩衝對象創建的緩衝對象或圖像對象都沒有被映射。
- 在寫命令執行完之前，所有用此緩衝對象創建的緩衝對象或圖像對象都沒有被任何命令隊列使用。

clEnqueueCopyBuffer

```
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 size,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
```

此函式會將一個拷貝命令入隊，此命令會將緩衝對象 *src_buffer* 的內容拷貝到 *dst_buffer* 中。

command_queue 即拷貝命令所要插入的命令隊列。*command_queue*、*src_buffer* 和 *dst_buffer* 必須位於同一 OpenCL 上下文中。

src_offset 即在 *src_buffer* 中的什麼位置開始讀取數據。

dst_offset 即在 *dst_buffer* 中的什麼位置開始寫入數據。

size 即所要拷貝數據的字節數。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，就可以回收並重新使用 *event_wait_list* 所關聯的內存了。

event 會返回一個事件對象，用來標識此拷貝命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。不過可以用 **clEnqueueBarrierWithWaitList** 來代替。如果 *event_wait_list* 和 *event* 都不是 NULL，則 *event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueCopyBuffer** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue*、*src_buffer* 以及 *dst_buffer* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *src_buffer* 或 *dst_buffer* 無效。
- CL_INVALID_VALUE，如果 *src_offset*、*dst_offset*、*size*、*src_offset* + *size* 或 *dst_offset* + *size* 越限。
- CL_INVALID_VALUE，如果 *size* 是 0。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項：
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *src_buffer* 是子緩衝對象，且創建此對象時所指定的 *offset* 沒有與 *command_queue* 所關聯设备的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *dst_buffer* 是子緩衝對象，且創建此對象時所指定的 *offset* 沒有與 *command_queue* 所關聯设备的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_MEM_COPY_OVERLAP，如果 *src_buffer* 和 *dst_buffer* 是同一個緩衝對象或子緩衝對象，且源和目的區域重疊。或者 *src_buffer* 和 *dst_buffer* 是同一緩衝對象的不同

子緩衝對象，且他們重疊。當 *src_offset* ≤ *dst_offset* ≤ *src_offset* + *size* − 1 或 *dst_offset* ≤ *src_offset* ≤ *dst_offset* + *size* − 1 時就認為重疊了。

- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *src_buffer* 或 *dst_buffer* 分配內存失敗。
- CL_INVALID_OPERATION，如果調用的是 **clEnqueueReadBufferRect**，但創建 *buffer* 時指定了 CL_MEM_HOST_WRITE_ONLY 或 CL_MEM_HOST_NO_ACCESS。
- CL_INVALID_OPERATION，如果調用的是 **clEnqueueWriteBufferRect**，但創建 *buffer* 時指定了 CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_NO_ACCESS。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clEnqueueCopyBufferRect

```
cl_int clEnqueueCopyBufferRect (
                            cl_command_queue command_queue,
                            cl_mem src_buffer,
                            cl_mem dst_buffer,
                            const size_t *src_origin,
                            const size_t *dst_origin,
                            const size_t *region,
                            size_t src_row_pitch,
                            size_t src_slice_pitch,
                            size_t dst_row_pitch,
                            size_t dst_slice_pitch,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)
```

此函式會將一個拷貝命令入隊，此命令會將緩衝對象 *src_buffer* 的一個 2D 或 3D 矩形區域拷貝到 *dst_buffer* 中。拷貝時源和宿的偏移量的計算方式在後面 *src_origin* 和 *dst_origin* 的描述中會涉及。每次拷貝的字節數等於區域寬度，然後源和宿的偏移量會增加相應的行間距（row pitch）。每拷貝完一個 2D 矩形，源和宿的偏移量會增加相應的面間距（slice pitch）。

> 如果 *src_buffer* 和 *dst_buffer* 是同一個緩衝對象，*src_row_pitch* 和 *dst_row_pitch* 必須相同，*src_slice_pitch* 和 *dst_slice_pitch* 也必須相同。

command_queue 就是拷貝命令要進入的隊列。*command_queue*、*src_buffer*、*dst_buffer* 必須位於同一 OpenCL 上下文中。

src_origin 定義了 *src_buffer* 中內存區域的偏移量 (x, y, z)。對於 2D 矩形區域，z 的值即 *src_origin*[2] 應該是 0。偏移量這樣計算：*src_origin*[2] * *src_slice_pitch* + *src_origin*[1] * *src_row_pitch* + *src_origin*[0]。

dst_origin 定義了 *dst_buffer* 中內存區域的偏移量 (x, y, z)。對於 2D 矩形區域，z 的值即 *dst_origin*[2] 應該是 0。偏移量這樣計算：*dst_origin*[2] * *dst_slice_pitch* + *dst_origin*[1] * *dst_row_pitch* + *dst_origin*[0]。

region 定義了要拷貝的 2D 或 3D 區域：$(width, height, depth)$，其單位分別是 byte、row、slice。對於 2D 區域，*depth* 的值，即 *region*[2] 應該是 1。

src_row_pitch 是 *src_buffer* 中每一行數據的字節數。如果 *src_row_pitch* 是 0，則在計算偏移量時用 *region*[0] 替代。

src_slice_pitch 是 *src_buffer* 中每個 2D 平面（slice）的字節數。如果 *src_slice_pitch* 是 0，則在計算偏移量時用 *region*[1] * *src_row_pitch* 替代。

dst_row_pitch 是 *dst_buffer* 中每一行數據的字節數。如果 *dst_row_pitch* 是 0，則在計算偏移量時用 *region*[0] 替代。

dst_slice_pitch 是 *dst_buffer* 中每個 2D 平面的字節數。如果 *dst_slice_pitch* 是 0，則在計算偏移量時用 *region*[1] * *src_row_pitch* 替代。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_*

list 必須是0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，就可以回收並重新使用 *event_wait_list* 所關聯的內存了。

event 會返回一個事件對象，用來標識此讀、寫命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。這時可以用 **clEnqueueBarrierWithWaitList** 來代替。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueCopyBufferRect** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue*、*src_buffer* 和 *dst_buffer* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *src_buffer* 或 *dst_buffer* 無效。
- CL_INVALID_VALUE，如果下列兩個區域中任意一個越限:
  - (*src_origin*, *region*, *src_row_pitch*, *src_slice_pitch*)
  - (*dst_origin*, *region*, *dst_row_pitch*, *dst_slice_pitch*)
- CL_INVALID_VALUE，如果 *region* 中的任何元素是 0。
- CL_INVALID_VALUE，如果 *src_row_pitch* 不是 0 且小於 *region*[0]。
- CL_INVALID_VALUE，如果 *dst_row_pitch* 不是 0 且小於 *region*[0]。
- CL_INVALID_VALUE，如果 *src_slice_pitch* 不是 0，並且滿足下列條件之一:
  - 小於 *region*[1] * *src_row_pitch*;
  - 或者不是 *src_row_pitch* 的整數倍。
- CL_INVALID_VALUE，如果 *dst_slice_pitch* 不是 0，並且滿足下列條件之一:
  - 小於 *region*[1] * *dst_row_pitch*;
  - 或者不是 *dst_row_pitch* 的整數倍。
- CL_INVALID_VALUE，如果 *src_buffer* 和 *dst_buffer* 是同一個緩衝對象，且 *src_slice_pitch* 不等於 *dst_slice_pitch* 或者 *src_row_pitch* 不等於 *dst_row_pitch*。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_MEM_COPY_OVERLAP，如果 *src_buffer* 和 *dst_buffer* 是同一個緩衝對象或子緩衝對象，且源和目的區域重疊。或者 *src_buffer* 和 *dst_buffer* 是同一緩衝對象的不同子緩衝對象，且他們重疊。對於怎樣確定源和目的區域重疊，請參考 附錄 *F*。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *src_buffer* 是子緩衝對象，且創建此對象時所指定的 *offset* 沒有與 *command_queue* 所關聯設備的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *dst_buffer* 是子緩衝對象，且創建此對象時所指定的 *offset* 沒有與 *command_queue* 所關聯設備的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *src_buffer* 或 *dst_bufer* 分配內存失敗。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.2.3 填充緩衝對象

clEnqueueFillBuffer

```
cl_int clEnqueueFillBuffer (
                        cl_command_queue command_queue,
                        cl_mem buffer,
                        const void *pattern,
                        size_t pattern_size,
                        size_t offset,
                        size_t size,
                        cl_uint num_events_in_wait_list,
                        const cl_event *event_wait_list,
                        cl_event *event)
```

此函式所入隊的命令可以按給定的範式（pattern）填充緩衝對象。*buffer* 的一些用法資訊（如是否可由內核或主機讀寫、以及創建 *buffer* 時所指定的引數 cl_mem_flags）會被 **clEnqueueFillBuffer** 忽略。

command_queue 即這個填充命令要插入的隊列。*command_queue* 和 *buffer* 必須位於用一個 OpenCL 上下文中。

buffer 是一個緩衝對象。

pattern 指向數據範式（data pattern），其大小為 *pattern_size*。用 *pattern* 填充的區域在 *buffer* 中的偏移量為 *offset*，大小是 *size*。數據範式必須是 OpenCL 所支持的標量或矢量的整數或浮點數型別，參見節 *6.1.1*和節 *6.1.2*。例如，如果數據範式是 float4，則 *pattern* 指向一個型別為 cl_float4 的值，且 *pattern_size* 是 sizeof(cl_float4)。*pattern_size* 的最大值就是 OpenCL 設備所支持的最大的整數或浮點數矢量數據型別的大小。函式返回後，就可以重用或者釋放 *pattern* 所指向的內存了。

offset 即填充區域在 *buffer* 中的偏移量，單位：字節，他必須是 *pattern_size* 的整數倍。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，就可以回收並重新使用 *event_wait_list* 所關聯的內存了。

event 會返回一個事件對象，用來標識此命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。不過可以用 **clEnqueueBarrierWithWaitList** 代替。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueFillBuffer** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *buffer* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *buffer* 無效。
- CL_INVALID_VALUE，如果 (*offset*, *size*) 所指定的區域越限。
- CL_INVALID_VALUE，如果 *pattern* 是 NULL，或者 *pattern_size* 是 0，或者 *pattern_size* 不屬於 {1, 2, 4, 8, 16, 32, 64, 128}。
- CL_INVALID_VALUE，如果 *offset* 和 *size* 不是 *pattern_size* 的整數倍。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項：
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *buffer* 是子緩衝對象，且創建此對象時所指定的 *offset* 沒有與 *queue* 所關聯設備的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *buffer* 分配內存失敗。
- CL_OUT_OF_RESOURCES，如果在為設備上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.2.4 映射緩衝對象

clEnqueueMapBuffer

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

此函式所入隊的命令會將 *buffer* 的某個區域映射到主機的位址空間中，並返回新位址。

command_queue 必須是一個有效的命令。

blocking_map 表明此映射操作是**阻塞的**還是**非阻塞的**。

如果 *blocking_map* 是 CL_TRUE，即映射命令是阻塞的，直到映射完成，**clEnqueueMapBuffer** 才會返回，應用可以用返回的指針訪問所映射區域的內容。

如果 *blocking_map* 是 CL_FALSE，即映射命令是非阻塞的，直到映射命令完成後，才能使用返回的指針。參數 *event* 會返回一個事件對象，可用來查詢映射命令的執行情況。映射命令完成後，應用就可以使用返回的指針訪問映射區域的內容了。

map_flags 是位欄，參見表 *5.5*。

buffer 是一個緩衝對象，必須與 *command_queue* 位於同一 OpenCL 上下文中。

offset 和 *size* 即所要映射的區域在緩衝對象中的偏移量和大小，單位都是字節。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，就可以回收並重新使用 *event_wait_list* 所關聯的內存了。

event 會返回一個事件對象，用來標識此命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

errcode_ret 用來返回錯誤碼。如果是 NULL，則不會返回錯誤碼。

表 5.5 所支持的 cl_map_flags

cl_map_flags
CL_MAP_READ
表明所映射區域是用來讀的。當 **clEnqueueMap{Buffer | Image}** 所入隊的命令完成時，保證所返回的指針中所映射區域的內容是最新的。
CL_MAP_WRITE
表明所映射區域是用來寫的。當 **clEnqueueMap{Buffer | Image}** 所入隊的命令完成時，保證所返回的指針中所映射區域的內容是最新的。
CL_MAP_WRITE_INVALIDATE_REGION
表明所映射區域是用來寫的。所映射區域中的原有內容會被丟棄。一種典型的情況就是主機會複寫（overwrite）其內容。此標誌告訴實作可以不用保證 **clEnqueueMap{Buffer | Image}** 所返回的指針中所映射區域的內容是最新的，這會極大提升性能。 他和 CL_MAP_WRITE 是互斥的。

如果執行成功，**clEnqueueMapBuffer** 會返回映射區域的指針，並將 *errcode_ret* 置為 CL_SUCCESS。否則，返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *buffer* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *buffer* 無效。
- CL_INVALID_VALUE，如果 (*offset*, *size*) 所指定的區域越限，或者 *size* 是 0，或者 *map_flags* 的值無效。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *buffer* 是子緩衝對象，且創建此對象時所指定的 *offset* 沒有與 *command_queue* 所關聯設备的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_MAP_FAILURE，如果映射失敗。對於用 CL_MEM_USE_HOST_PTR 或 CL_MEM_ALLOC_HOST_PTR 創建的緩衝對象不能出現此錯誤。
- CL_INVALID_OPERATION，如果創建 *buffer* 時指定了 CL_MEM_HOST_WRITE_ONLY 或 CL_MEM_HOST_NO_ACCESS，而在 *map_flags* 中設置了 CL_MAP_READ；或者創建 *buffer* 時指定了 CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_NO_ACCESS，而在 *map_flags* 中設置了 CL_MAP_WRITE 或 CL_MAP_WRITE_INVALIDATE_REGION。
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST，如果映射操作是阻塞的，且 *event_wait_list* 中任一事件的執行狀態是負整數。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *buffer* 分配內存失敗。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

所返回的指針僅可用於訪問所映射的區域 (*offset*, *size*)，雖然實作可能映射更大的區域。對於訪問此區域外的內容，其結果未定義。

> 如果創建緩衝對象時在 *mem_flags* 中指定了 CL_MEM_USE_HOST_PTR，則:
> - 當 **clEnqueueMapBuffer** 所入隊的命令完成後，保證 **clCreateBuffer** 的 *host_ptr* 中所映射區域的內容是最新的。
> - **clEnqueueMapBuffer** 所返回的指針得自創建緩衝對象時所用的 *host_ptr*。

被映射過的緩衝對象使用 **clEnqueueUnmapMemObject** 進行解映射。參見節 *5.4.2*。

节 5.3 圖像對象

圖像對象用來存儲一維、二維或三維的材質、幀緩衝（frame-buffer）或圖像。其中元素的類型是在預定義圖像格式中選取的。每個圖像對象中至少要有一個元素。

5.3.1 創建圖像對象

clCreateImage

```
cl_mem clCreateImage (cl_context context,
                      cl_mem_flags flags,
                      const cl_image_format *image_format,
                      const cl_image_desc *image_desc,
                      void *host_ptr,
                      cl_int *errcode_ret)
```

此函式可用來創建 1D 圖像、1D 圖像緩衝 (image buffer)、1D 圖像陣列 (image array)、2D 圖像、2D 圖像陣列以及 3D 圖像對象。

context 即將要創建的圖像對象所處的 OpenCL 上下文。

flags 是位欄，用來指明如何分配以及怎樣使用將要創建的圖像對象，參見表 *5.3*。

對於所有類型的圖像對象，除了 CL_MEM_OBJECT_IMAGE1D_BUFFER，如果 *flags* 的值為 0，則使用缺省值 CL_MEM_READ_WRITE。

對於類型為 CL_MEM_OBJECT_IMAGE1D_BUFFER 的圖像對象，如果 *flags* 中沒有設置 CL_MEM_READ_WRITE、CL_MEM_READ_ONLY 或 CL_MEM_WRITE_ONLY，則會從 *buffer* 中繼承這些屬性。而 *flags* 中不能設置 CL_MEM_USE_HOST_PTR、CL_MEM_ALLOC_HOST_PTR 和 CL_MEM_COPY_HOST_PTR，這些也會由 *buffer* 繼承。即使 *buffer* 的內存訪問限定符中有 CL_MEM_COPY_HOST_PTR，也並不意味着創建子緩衝對象時會有額外的拷貝。如果 *flags* 中沒有設置 CL_MEM_HOST_WRITE_ONLY、CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_NO_ACCESS，則會從 *buffer* 中繼承這些屬性。

image_format 指明圖像的格式。參見節 *5.3.1.1*。

image_desc 指明圖像的類型以及維數。參見節 *5.3.1.2*。

host_ptr 指向圖像數據 (可能已由應用分配好)。下表列出了 *host_ptr* 所指緩衝大小的一些限制。

圖像類型	***host_ptr* 所指緩衝的大小**
CL_MEM_OBJECT_IMAGE1D	>= image_row_pitch
CL_MEM_OBJECT_IMAGE1D_BUFFER	>= image_row_pitch
CL_MEM_OBJECT_IMAGE2D	>= image_row_pitch * image_height
CL_MEM_OBJECT_IMAGE3D	>= image_slice_pitch * image_depth
CL_MEM_OBJECT_IMAGE1D_ARRAY	>= image_slice_pitch * image_array_size
CL_MEM_OBJECT_IMAGE2D_ARRAY	>= image_slice_pitch * image_array_size

對於 3D 圖像或 2D 圖像陣列，*host_ptr* 所指圖像數據分別按相鄰的 2D 圖像平面或 2D 圖像線性序列進行存儲。每個 2D 圖像都是相鄰掃描列 (scanline) 的線性序列。每個掃描列都是相鄰圖像元素的線性序列。

對於 2D 圖像，*host_ptr* 所指圖像數據按相鄰掃描列的線性序列進行存儲。每個掃描列都是相鄰圖像元素的線性序列。

對於 1D 圖像陣列，*host_ptr* 所指圖像數據按相鄰 1D 圖像的線性序列進行存儲。每個 1D 圖像或 1D 圖像緩衝就是一個掃描列，是相鄰圖像元素的線性序列。

errcode_ret 會返回相應的錯誤碼。如果 *errcode_ret* 是 NULL，不會返回錯誤碼。

如果成功創建了圖像對象，**clCreateImage** 會將其返回，並將 *errcode_ret* 置為 CL_SUCCESS。否則，返回 NULL 並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_INVALID_VALUE，如果 *flags* 的值無效。
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR，如果 *image_format* 中的值無效，或者 *image_format* 是 NULL。
- CL_INVALID_IMAGE_DESCRIPTOR，如果 *image_desc* 中的值無效，或者 *image_desc* 是 NULL。
- CL_INVALID_IMAGE_SIZE，如果 *image_desc* 中圖像任一維度的值超過了 *context* 中任一设备在相應維度上的最大值 (參見表 *4.3*)。
- CL_INVALID_HOST_PTR，如果 *image_desc* 中 *host_ptr* 是 NULL，但 *flags* 中設置了 CL_MEM_USE_HOST_PTR 或 CL_MEM_COPY_HOST_PTR；或者 *host_ptr* 不是 NULL，但 *flags* 中設置了 CL_MEM_COPY_HOST_PTR 或 CL_MEM_USE_HOST_PTR。
- CL_INVALID_VALUE，如果要創建的是 1D 圖像緩衝，為其指定 CL_MEM_WRITE_ONLY 的同時 *flags* 中設置了 CL_MEM_READ_WRITE 或 CL_MEM_READ_ONLY；或者為其指定 CL_MEM_READ_ONLY 的同時 *flags* 中設置了 CL_MEM_READ_WRITE 或 CL_MEM_WRITE_ONLY；或者 *flags* 中設置了 CL_MEM_USE_HOST_PTR、CL_MEM_ALLOC_HOST_PTR、或 CL_MEM_COPY_HOST_PTR。
- CL_INVALID_VALUE，如果要創建的是 1D 圖像緩衝，為其指定 CL_MEM_HOST_WRITE_ONLY 的同時 *flags* 中設置了 CL_MEM_HOST_READ_ONLY；或者為其指定 CL_MEM_HOST_READ_ONLY 的同時 *flags* 中設置了 CL_MEM_HOST_WRITE_ONLY；或者為其指定 CL_MEM_HOST_NO_ACCESS 的同時 *flags* 中設置了 CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_WRITE_ONLY。
- CL_IMAGE_FORMAT_NOT_SUPPORTED，如果 *image_format* 的值不受支持。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為圖像對象分配內存失敗。

- CL_INVALID_OPERATION，如果 *context* 中的所有设备都不支持圖像（即表 *4.3*中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE）。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.3.1.1 圖像格式描述符

圖像格式描述符的結構定義如下:

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

image_channel_order 指定了通道（channel）數目以及通道布局，即圖像中通道存儲的內存布局。其有效值參見表 *5.6*。

image_channel_data_type 即通道的數據類型。其有效值參見表 *5.7*。

由以上兩者所確定的圖像元素的位數必須是 2 的指數。

表 5.6 圖 像 通 道 順 序

image_channel_order 中可指定的枚舉值	只有通道數據類型為下列值時才能使用此格式
CL_R、CL_Rx、CL_A	
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、CL_RGx、CL_RA	
CL_RGB 或者 CL_RGBx	CL_UNORM_SHORT_565、CL_UNORM_SHORT_555、CL_UNORM_INT_101010
CL_RGBA	CL_UNORM_INT8、CL_SNORM_INT8、CL_SIGNED_INT8、CL_UNSIGNED_INT8

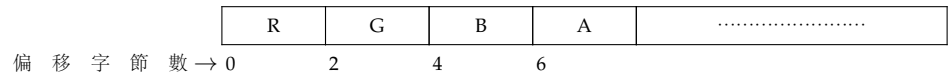
表 5.7 圖 像 通 道 數 據 類 型

圖像通道數據類型	描述
CL_SNORM_INT8	各通道均為歸一化帶符號 8 位整數
CL_SNORM_INT16	各通道均為歸一化帶符號 16 位整數
CL_UNORM_INT8	各通道均為歸一化無符號 8 位整數
CL_UNORM_INT16	各通道均為歸一化無符號 16 位整數
CL_UNORM_SHORT_565	歸一化 5-6-6 3 通道 RGB 圖像。通道順序必須是 CL_RGB 或 CL_RGBx。
CL_UNORM_INT_101010	歸一化 x-10-10-10 4 通道 xRGB 圖像。通道順序必須是 CL_RGB 或 CL_RGBx。
CL_SIGNED_INT8	各通道均為非歸一化帶符號 8 位整數
CL_SIGNED_INT16	各通道均為非歸一化帶符號 16 位整數
CL_SIGNED_INT32	各通道均為非歸一化帶符號 32 位整數
CL_UNSIGNED_INT8	各通道均為非歸一化無符號 8 位整數
CL_UNSIGNED_INT16	各通道均為非歸一化無符號 16 位整數
CL_UNSIGNED_INT32	各通道均為非歸一化無符號 32 位整數
CL_HALF_FLOAT	個通道均為 16 位半浮點數
CL_FLOAT	個通道均為單精度浮點數

例如，如果 image_channel_order = CL_RGBA，image_channel_data_type = CL_UNORM_INT8，則其內存布局為:

R	G	B	A	……
偏移字節數 → 0	1	2	3	

類似，如果 image_channel_order = CL_RGBA，image_channel_data_type = CL_SIGNED_INT16，則其內存布局為:



CL_UNORM_SHORT_565、CL_UNORM_SHORT_555 和 CL_UNORM_INT_101010 這三種通道數據類型比較特殊，他們都是壓縮過的圖像格式，每個元素的所有通道數據被壓縮到一個無符號短整形或無符號整形中。在壓縮時，一般第一個通道佔據最高位（most significant bit），後續通道相繼佔據次高位。對於 CL_UNORM_SHORT_565，R 佔據比特 15:11；G 佔據比特 10:5；B 佔據比特 4:0。對於 CL_UNORM_SHORT_555，比特 15 未定義；R 佔據比特 14:10；G 佔據比特 9:5；B 佔據比特 4:0。對於 CL_UNORM_INT_101010，比特 31:30 未定義；R 佔據比特 29:20；G 佔據比特 19:10；B 佔據比特 9:0。

OpenCL 實作必須能夠保持 image_channel_data_type 的位數所指定的最小精度。如果 OpenCL 實作不支持由 image_channel_order 和 image_channel_data_type 所定義的圖像格式，則 **clCreateImage** 會返回 NULL。

5.3.1.2 圖像描述符

圖像描述符描述了圖像或圖像陣列的類型和維數，其定義如下：

```
1  typedef struct _cl_image_desc {
2          cl_mem_object_type      image_type,
3          size_t                  image_width;
4          size_t                  image_height;
5          size_t                  image_depth;
6          size_t                  image_array_size;
7          size_t                  image_row_pitch;
8          size_t                  image_slice_pitch;
9          cl_uint                 num_mip_levels;
10         cl_uint                 num_samples;
11         cl_mem                  buffer;
12 } cl_image_desc;
```

image_type 即圖像類型，其值必須是 CL_MEM_OBJECT_IMAGE1D、CL_MEM_OBJECT_IMAGE1D_BUFFER、CL_MEM_OBJECT_IMAGE1D_ARRAY、CL_MEM_OBJECT_IMAGE2D、CL_MEM_OBJECT_IMAGE2D_ARRAY 或 CL_MEM_OBJECT_IMAGE3D。

image_width 即圖像寬度，單位：像素。對於 2D 圖像或圖像陣列，其寬度必須 <= CL_DEVICE_IMAGE2D_MAX_WIDTH。對於 3D 圖像，其寬度必須 <= CL_DEVICE_IMAGE3D_MAX_WIDTH。對於 1D 圖像緩衝，其寬度必須 <= CL_DEVICE_IMAGE_MAX_BUFFER_SIZE。對於 1D 圖像和 1D 圖像陣列，其寬度必須 <= CL_DEVICE_IMAGE2D_MAX_WIDTH。

image_height 即圖像高度，單位：像素。只有圖像為 2D、3D 或 2D 圖像陣列時才有效。對於 2D 圖像或圖像陣列，其寬度必須 <= CL_DEVICE_IMAGE2D_MAX_HEIGHT。對於 3D 圖像，其寬度必須 <= CL_DEVICE_IMAGE3D_MAX_HEIGHT。

image_depth 即圖像深度，單位：像素。只有圖像為 3D 圖像時才有效，其值必須 >= 1 且 <= CL_DEVICE_IMAGE3D_MAX_DEPTH。

image_array_size[7] 即圖像陣列中圖像個數。只有圖像為 1D 或 2D 圖像陣列時才有效。如果設置了 image_array_size，其值必須 >= 1 且 <= CL_DEVICE_IMAGE_MAX_ARRAY_SIZE。

image_row_pitch 即掃描列間隔，單位字節。如果 *host_ptr* 是 NULL，其值必須是 0；如果 *host_ptr* 不是 NULL，則可以是 0 或者 >= image_width * 元素大小。如果 *host_ptr* 不是 NULL，並且 image_row_pitch = 0，則用 image_width * 元素大小取代 image_row_pitch。如果 imag_row_pitch 不是 0，則必須是圖像元素大小的整數倍。

image_slice_pitch 即 3D 圖像中每個 2D 平面的大小，或者 1D、2D 圖像陣列中每個圖像的大小。單位字節。如果 *host_ptr* 是 NULL，其值必須是 0；如果 *host_ptr* 不是 NULL，對於 2D 圖像陣列或 3D 圖像，可以是 0 或者 >= image_row_pitch * image_height；對於 1D 圖像陣列，可以是 0 或者 >= image_row_pitch。如果 *host_ptr* 不是 NULL，並且 image_slice_pitch = 0，對於 2D 圖像陣列或 3D 圖像，則用 image_row_pitch * image_height 取代 image_slice_pitch；對於 1D 圖像陣列，則用 image_row_pitch。如果 imag_slice_pitch 不是 0，則必須是 image_row_pitch 的整數倍。

num_mip_levels 和 num_samples 必須是 0。

[7] 對於內核而言，以 *image_array_size* = 1 讀寫 2D 圖像陣列時，其性能可能會比直接讀寫 2D 圖像要低。

如果 image_type 是 CL_MEM_OBJECT_IMAGE1D_BUFFER，則 buffer 引用一個緩衝對象，否則 buffer 必須是 NULL。對於 1D 圖像緩衝對象，圖像的所有像素均取自緩衝對象的數據。在相應的同步點上，對緩衝對象中數據的改動也會反映到 1D 圖像緩衝對象的內容上，反之亦然。image_width * 元素大小必須 <= 緩衝對象中的數據大小。

> 對一個緩衝對象以及與其關聯的 *1D* 圖像緩衝對象的並發讀、寫以及拷貝是未定義的。只有讀是定義了的。

5.3.2 查詢所支持的圖像格式

clGetSupportedImageFormats

```
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)
```

此函式可用來查詢 OpenCL 實作所支持的圖像格式，對於圖像對象，需要指定如下資訊:

- 上下文
- 圖像類型——1D、2D 或 3D 圖像，1D 圖像緩衝，1D 或 2D 圖像陣列
- 圖像對象的分配資訊

clGetSupportedImageFormats 會返回一個聯合體（union），其中的圖像格式是 *contex* 中所有设备都支持的。

context 是 OpenCL 上下文，將在其上創建圖像對象。

flags 是位欄，用來描述將要創建的圖像對象的分配和用法資訊。參見表 *5.5*。

image_type 即圖像類型，其值可以是：CL_MEM_OBJECT_IMAGE1D、CL_MEM_OBJECT_IMAGE1D_BUFFER、CL_MEM_OBJECT_IMAGE2D、CL_MEM_OBJECT_IMAGE3D、CL_MEM_OBJECT_IMAGE1D_ARRAY 或 CL_MEM_OBJECT_IMAGE2D_ARRAY。

num_entries 即 *image_formats* 所能容納的表項數目。

image_formats 用來存儲所返回的圖像格式。每一項都是 cl_image_format 結構體。如果 *image_formats* 是 NULL，則忽略。

num_image_formats 即所返回的圖像格式的實際數目。如果是 NULL，則忽略。

如果執行成功，則 **clGetSupportedImageFormats** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_INVALID_VALUE，如果 *flags* 或 *image_type* 無效，或者 *num_entries* 是 0 且 *image_formats* 是 NULL。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

如果 CL_DEVICE_IMAGE_SUPPORT（參見表 *4.3*）是 CL_TRUE，則 OpenCL 實作賦予下列變量的值必須大於或等於表 *4.3*中所規定的最小值:

- 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
- CL_DEVICE_MAX_SAMPLERS

5.3.2.1 必須要支持的圖像格式

對於所有支持圖像的设备而言，必須要支持的 2D 和 3D 圖像對象的格式（無論讀寫）如表 *5.8*所示。

表 5.8 必須要支持的圖像格式

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

5.3.3 讀、寫以及拷貝圖像對象

clEnqueueReadImage

clEnqueueWriteImage

```
cl_int clEnqueueReadImage (cl_command_queue command_queue,
                           cl_mem image,
                           cl_bool blocking_read,
                           const size_t *origin,
                           const size_t *region,
                           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,
                           const size_t *region,
                           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)
```

這兩個函式所入隊的命令可用於讀寫圖像對象，此命令可以將圖像或圖像陣列的內容讀取到主機內存中，或者由主機內存寫入到圖像或圖像陣列中。

command_queue 就是命令要進入的隊列。*command_queue* 和 *image* 必須是由同一個 OpenCL 上下文創建的。

image 是圖像對象或圖像陣列對象。

blocking_read 和 *blocking_write* 表明讀寫操作是阻塞的還是非阻塞的。

如果 *blocking_read* 是 CL_TRUE，即讀命令是阻塞的，直到將圖像數據完全拷貝到 *ptr* 所指內存中後，**clEnqueueReadImage** 才會返回。

如果 *blocking_read* 是 CL_FALSE，即讀命令是非阻塞的，**clEnqueueReadBuffer** 將命令入隊後就會返回。只有等到讀命令執行完畢，才能繼續使用 *ptr* 所指向的內容。參數 *event* 所返回的事件對象可用來查詢讀命令的執行情況。讀命令完成後，應用就可以繼續使用 *ptr* 所指向的內容了。

如果 *blocking_write* 是 CL_TRUE，OpenCL 實作會拷貝 *ptr* 所指向的數據，並將寫命令入隊。**clEnqueueWriteBuffer** 返回後，應用就可以重新使用 *ptr* 所指向的內存了。

如果 *blocking_write* 是 CL_FALSE，OpenCL 實作會用 *ptr* 實施非阻塞的寫操作。既然非阻塞，實作就可以立即返回。返回後，應用還不能立刻就使用 *ptr* 所指內存。參數 *event* 所返回的事件對象可用來查詢寫命令的執行情況。寫命令完成後，應用就可以重新使用 *ptr* 所指向的內存了。

origin 定義了所要讀寫區域的起始位置 (x, y, z)，單位像素；對於 2D 圖像陣列是偏移量 (x, y) 以及圖像索引；對於 1D 圖像陣列是偏移量 (x) 以及圖像索引。如果 *image* 是 2D 圖像對象，*origin*[2] 必須是 0。如果 *image* 是 1D 圖像對象或 1D 圖像緩衝對象，*origin*[1] 和 *origin*[2] 都必須是 0。如果 *image* 是 1D 圖像陣列對象，*origin*[2] 必須是 0，*origin*[1] 是圖像在陣列中的索引。如果 *image* 是 2D 圖像陣列對象，則 *origin*[2] 是圖像在陣列中的索引。

region 定義了要讀寫區域的大小 $(width, height, depth)$，單位像素。對於 2D 圖像陣列對象，即 $(width, height)$ 以及圖像個數。對於 1D 圖像陣列對象，即 $(width)$ 以及圖像個數。如果 *image* 是 2D 圖像對象或 1D 圖像陣列對象，*region*[2] 必須是 1。如果 *image* 是 1D 圖像對象或 1D 圖像緩衝對象，*region*[1] 和 *region*[2] 都必須是 0。

clEnqueueReadImage 中的 *row_pitch* 和 **clEnqueueWriteImage** 中的 *input_row_pitch* 即每行的長度，單位字節。其值必須大於或等於 *元素大小* * *width*。如果 *row_pitch* 或 *input_row_pitch* 是 0，則會根據 *元素大小* * *width* 自動計算行間距。

clEnqueueReadImage 中的 *slice_pitch* 和 **clEnqueueWriteImage** 中的 *input_slice_pitch* 即 3D 圖像的 3D 區域中 2D 平面大小，或者 1D、2D 圖像陣列中每個圖像的大小，單位字節。如果 *image* 是 1D 或 2D 圖像，其值必須是 0，否則必須大於或等於 *row_pitch* * *height*。如果 *slice_pitch* 或 *input_slice_pitch* 是 0，則會根據 *row_pitch* * *height* 自動計算面間距。

ptr 指向主機中的一塊內存，用作讀命令的數據源以及寫命令的目的地。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一上下文中。此函式返回後，就可以回收並重新使用 *event_wait_list* 所關聯的內存了。

event 會返回一個事件對象，用來標識此讀、寫命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueReadImage** 和 **clEnqueueWriteImage** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *image* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *image* 無效。
- CL_INVALID_VALUE，如果 (*origin*, *region*) 所指定的區域越限，或者 *ptr* 是 NULL。
- CL_INVALID_VALUE，如果 *origin* 和 *region* 沒有遵守上面所描述的相應規則。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項：
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_INVALID_IMAGE_SIZE，如果圖像的大小（寬度、高度、指定的或自動計算的行間距或面間距）不被 *command_queue* 所關聯设备支持。
- CL_IMAGE_FORMAT_NOT_SUPPORTED，如果 *image* 的圖像格式（圖像通道順序和數據類型）不被 *command_queue* 所關聯设备支持。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *image* 分配內存失敗。
- CL_INVALID_OPERATION，如果 *command_queue* 所關聯设备不支持圖像（即表 *4.3*中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE)。
- CL_INVALID_OPERATION，如果調用的是 **clEnqueueReadImage**，但創建 *image* 時指定了 CL_MEM_HOST_WRITE_ONLY 或 CL_MEM_HOST_NO_ACCESS。
- CL_INVALID_OPERATION，如果調用的是 **clEnqueueWriteImage**，但創建 *image* 時指定了 CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_NO_ACCESS。

- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST，如果讀寫操作是阻塞的，並且 *event_wait_list* 中任一事件的執行結果是負整數。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

如果調用 **clEnqueueReadImage** 時，參數 *ptr* 的值為 *host_ptr* + (*origin*[2] * *image_slice_pitch* + *origin*[1] * *image_row_pitch* + *origin*[0] * 單像素字節數，其中 *host_ptr* 是用 CL_MEM_USE_HOST_PTR 創建 *image* 時指定的，為避免未定義行為必須滿足下列要求：

- 在讀命令開始執行前，所有使用此圖像對象的命令都已執行完畢。
- **clEnqueueReadImage** 的 *row_pitch* 和 *slice_pitch* 必須設置成圖像的行間距和面間距。
- 此圖像對象沒有被映射。
- 在讀命令執行完以前，任何命令隊列都不能使用此圖像對象。

如果調用 **clEnqueueWriteImage** 時，參數 *ptr* 的值為 *host_ptr* + (*origin*[2] * *image_slice_pitch* + *origin*[1] * *image_row_pitch* + *origin*[0] * 單像素字節數，其中 *host_ptr* 是用 CL_MEM_USE_HOST_PTR 創建 *image* 時指定的，為避免未定義行為必須滿足下列要求：

- 在寫命令開始執行前，主機內存中的內容已經最新。
- **clEnqueueWriteImage** 的 *input_row_pitch* 和 *input_slice_pitch* 必須設置成圖像的行間距和面間距。
- 此圖像對象沒有被映射。
- 在寫命令執行完以前，任何命令隊列都不能使用此圖像對象。

clEnqueueCopyImage

```
cl_int clEnqueueCopyImage (cl_command_queue command_queue,
                           cl_mem src_image,
                           cl_mem dst_image,
                           const size_t *src_origin,
                           const size_t *dst_origin,
                           const size_t *region,
                           cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list,
                           cl_event *event)
```

此函式會將一個拷貝命令入隊。 *src_image* 和 *dst_image* 可以是 1D、2D、3D 圖像或者 1D、2D 圖像陣列對象，允許我們實施下列拷貝動作：

- 由 1D 圖像對象拷貝到 1D 圖像對象。
- 由 1D 圖像對象拷貝到 2D 圖像對象的掃描列，反之亦可。
- 由 1D 圖像對象拷貝到 3D 圖像對象的 2D 平面的掃描列，反之亦可。
- 由 1D 圖像對象拷貝到 1D 或 2D 圖像陣列對象中某個圖像的掃描列，反之亦可。
- 由 2D 圖像對象拷貝到 2D 圖像對象。
- 由 2D 圖像對象拷貝到 3D 圖像對象的 2D 平面，反之亦可。
- 由 2D 圖像對象拷貝到 2D 圖像陣列對象中的某個圖像，反之亦可。
- 由 1D 圖像陣列對象拷貝到 1D 圖像陣列對象，反之亦可。
- 由 2D 圖像陣列對象拷貝到 2D 圖像陣列對象，反之亦可。
- 由 3D 圖像對象拷貝到 3D 圖像對象。

command_queue 就是拷貝命令要進入的隊列。 *command_queue*、*src_image*、*dst_image* 必須位於同一 OpenCL 上下文中。

src_origin 定義了源區域的偏移量。對於 1D、2D 或 3D 圖像，即 (x, y, z)，單位像素；對於 2D 圖像陣列，即 (x, y) 以及圖像索引；對於 1D 圖像陣列，即 (x) 以及圖像索引。如果 *src_image* 是 2D 圖像對象，*src_origin*[2] 必須是 0。如果 *src_image* 是 1D 圖像對象，*src_origin*[1] 和 *src_origin*[2] 都必須是 0。如果 *src_image* 是 1D 圖像陣列對象，*src_origin*[1] 即圖像索引。如果 *src_image* 是 2D 圖像陣列對象，*src_origin*[2] 即圖像索引。

dst_origin 除了對應的是 *dst_image*，其他都跟 *src_origin* 一樣。

region 定義了要拷貝的 1D、2D 或 3D 區域 $(width, height, depth)$，單位像素；對於 2D 圖像陣列，即 2D 區域 $(width, height)$ 以及圖像索引；對於 1D 圖像陣列，即 1D 區域 $(width)$ 以及圖像索引。如果 *src_image* 或 *dst_image* 是 2D 圖像對象或 1D 圖像陣列對象，*region*[2] 必

須是 1。如果 *src_image* 或 *dst_image* 是 1D 圖像或 1D 圖像緩衝對象，*src_origin*[1] 和 *region*[2] 必須是 1。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，就可以回收並重新使用 *event_wait_list* 所關聯的內存了。

event 會返回一個事件對象，用來標識此拷貝命令，可用來查詢或等待此命令結束。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。這時可以用 **clEnqueueBarrierWithWaitList** 來代替。如果 *event_wait_list* 和 *event* 都不是 NULL，則 *event* 不能屬於 *event_wait_list*。

目前要求 *src_image* 和 *dst_image* 具有完全相同的圖像格式（即創建 *src_image* 和 *dst_image* 時所指定的 cl_image_format 必須相同）。

如果執行成功，**clEnqueueCopyImage** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue*、*src_image* 和 *dst_image* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *src_image* 或 *dst_image* 無效。
- CL_IMAGE_FORMAT_MISMATCH，如果 *src_image* 和 *dst_image* 所使用的圖像格式不同。
- CL_INVALID_VALUE，如果 (*src_origin*, *region*) 所指定的區域在 *src_image* 中越限，或者 (*dst_origin*, *region*) 所指定的區域在 *dst_image* 中越限。
- CL_INVALID_VALUE，如果 *src_origin*、*dst_origin* 和 *region* 沒有遵守上面所描述的相應規則。
- CL_INVALID_VALUE，如果 *src_buffer* 和 *dst_buffer* 是同一個緩衝對象，且 *src_slice_pitch* 不等於 *dst_slice_pitch* 或者 *src_row_pitch* 不等於 *dst_row_pitch*。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項：
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_INVALID_IMAGE_SIZE，如果圖像的大小（寬度、高度、指定的或自動計算的行間距或面間距）不被 *queue* 所關聯设备支持。
- CL_IMAGE_FORMAT_NOT_SUPPORTED，如果 *src_image* 或 *dst_image* 的圖像格式(圖像通道順序和數據類型）不被 *queue* 所關聯设备支持。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *src_image* 或 *dst_image* 分配內存失敗。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。
- CL_INVALID_OPERATION，如果調用的是 **clEnqueueReadBufferRect**，但創建 *buffer* 時指定了 CL_MEM_HOST_WRITE_ONLY 或 CL_MEM_HOST_NO_ACCESS。
- CL_INVALID_OPERATION，如果 *command_queue* 所關聯设备不支持圖像（即，表 4.3中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE)。）。
- CL_MEM_COPY_OVERLAP，如果 *src_image* 和 *dst_image* 是同一個圖像對象，並且源區域和宿區域重疊。

5.3.4 填充圖像對象

clEnqueueFillImage

```
cl_int clEnqueueFillImage (cl_command_queue command_queue,
```

```
cl_mem image,
const void *fill_color,
const size_t *origin,
const size_t *region,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
```

此函式所入隊的命令可以將圖像對象填充成某種顏色。*image* 的一些用法資訊（如是否可讀、是否可寫，以及創建 *image* 時所指定的參數 cl_mem_flags）會被 **clEnqueueFillImage** 忽略。

command_queue 即填充命令所要插入的隊列。*command_queue* 和 *image* 必須位於用一個 OpenCL 上下文中。

image 是一個圖像對象。

fill_color 即填充色。如果 *image* 通道數據類型不是非歸一化帶符號或無符號整形，則他是四元 RGBA 浮點顏色值；如果 *image* 通道數據類型是非歸一化帶符號整形，則他是四元帶符號整形值；如果 *image* 通道數據類型是非歸一化無符號整形，則他是四元無符號整形值。填充時會根據 *image* 的圖像通道格式以及順序對此顏色進行轉換，參見節 *6.12.14*和節 *8.3*。

origin 定義了所要填充區域的起始位置 (x, y, z)，單位像素；對於 2D 圖像陣列是 (x, y) 以及圖像索引；對於 1D 圖像陣列是 (x) 以及圖像索引。如果 *image* 是 2D 圖像對象，*origin*[2] 必須是 0。如果 *image* 是 1D 圖像對象或 1D 圖像緩衝對象，*origin*[1] 和 *origin*[2] 都必須是 0。如果 *image* 是 1D 圖像陣列對象，則 *origin*[2] 必須是 0，*origin*[1] 是圖像在陣列中的索引。如果 *image* 是 2D 圖像陣列對象，則 *origin*[2] 是圖像在陣列中的索引。

region 定義了要填充區域的大小 $(width, height, depth)$，單位像素。對於 2D 圖像陣列對象，即 $(width, height)$ 以及圖像個數。對於 1D 圖像陣列對象，即 $(width)$ 以及圖像個數。如果 *image* 是 2D 圖像對象或 1D 圖像陣列對象，*region*[2] 必須是 1。如果 *image* 是 1D 圖像或 1D 圖像緩衝對象，*region*[1] 和 *region*[2] 都必須是 0。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，就可以回收並重新使用 *event_wait_list* 所關聯的內存了。

event 會返回一個事件對象，用來標識此命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。不過可以用 **clEnqueueBarrierWithWaitList** 代替。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueFillImage** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *image* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *image* 無效。
- CL_INVALID_VALUE，如果 *fill_color* 是 NULL。
- CL_INVALID_VALUE，如果 (*origin*, *region*) 所指定的區域越限或者 *ptr* 是 NULL。
- CL_INVALID_VALUE，如果 *origin* 和 *region* 沒有遵守前面參數描述中所列規則。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項：
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。

- CL_INVALID_IMAGE_SIZE，如果圖像的大小（寬度、高度、指定的或自動計算的行間距或面間距）不被 *queue* 所關聯设备支持。
- CL_IMAGE_FORMAT_NOT_SUPPORTED，如果 *image* 的圖像格式（圖像通道順序和數據型別）不被 *queue* 所關聯设备支持。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *image* 分配內存失敗。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.3.5 圖像對象和緩衝對象間的拷貝

clEnqueueCopyImageToBuffer

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

此函式所入隊的命令用來將圖像對象拷貝到緩衝對象中。

command_queue 就是拷貝命令要進入的隊列。*command_queue*、*src_image*、*dst_buffer* 必須位於同一 OpenCL 上下文中。

src_image 是一個圖像對象。

dst_buffer 是一個緩衝對象。

src_origin 定義了源區域的起始位置 (x, y, z)，單位像素；對於 2D 圖像陣列，即 (x, y) 以及圖像索引；對於 1D 圖像陣列，即 (x) 以及圖像索引。如果 *src_image* 是 2D 圖像對象，*src_origin*[2] 必須是 0。如果 *src_image* 是 1D 圖像對象或 1D 圖像緩衝對象，*src_origin*[1] 和 *src_origin*[2] 都必須是 0。如果 *src_image* 是 1D 圖像陣列對象，*src_origin*[1] 即圖像索引。如果 *src_image* 是 2D 圖像陣列對象，*src_origin*[2] 即圖像索引。

region 定義了所拷貝區域的大小 $(width, height, depth)$，單位像素；對於 2D 圖像陣列，即 $(width, height)$ 以及圖像索引；對於 1D 圖像陣列，即 $(width)$ 以及圖像索引。如果 *src_image* 是 2D 圖像對象或 1D 圖像陣列對象，*region*[2] 必須是 1。如果 *src_image* 是 1D 圖像或 1D 圖像緩衝對象，*src_origin*[1] 和 *region*[2] 必須是 1。

dst_offset 即拷貝時目的區域在 *dst_buffer* 中的起始位置。目的區域的大小，即 *dst_cb* 這樣計算：如果 *src_image* 是 3D 圖像對象，則為 $width * height * depth * bytes/imageelement$；如果 *src_image* 是 2D 圖像對象，則為 $width * height * bytes/imageelement$；如果 *src_image* 是 2D 圖像陣列對象，則為 $width * height * arraysize * bytes/imageelement$；如果 *src_image* 是 1D 圖像或 1D 圖像緩衝對象，則為 $width * bytes/imageelement$；如果 *src_image* 是 1D 圖像陣列對象，則為 $width * arraysize * bytes/imageelement$；

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，即可回收並重新使用 *event_wait_list* 所關聯的內存。

event 會返回一個事件對象，用來標識此拷貝命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。這時可以用 **clEnqueueBarrierWithWaitList** 來代替。如果 *event_wait_list* 和 *event* 都不是 NULL，則 *event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueCopyImageToBuffer** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue*、*src_image* 和 *dst_buffer* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *src_image* 或 *dst_buffer* 無效；或者 *src_image* 是由 *dst_buffer* 創建的 1D 圖像 buffer 對象。
- CL_INVALID_VALUE，如果 $(src\_origin, region)$ 所指定的區域在 *src_image* 中越限，或者 (dst_offset, dst_cb) 所指定的區域在 *dst_buffer* 中越限。
- CL_INVALID_VALUE，如果 *src_origin* 和 *region* 沒有遵守上面所描述的相應規則。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *dst_buffer* 是一個子緩衝對象，並且創建他時所指定的參數 *offset* 沒有和 *queue* 所在设备的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_INVALID_IMAGE_SIZE，如果圖像的大小（寬度、高度、指定的或自動計算的行間距或面間距）不被 *queue* 所在设备支持。
- CL_IMAGE_FORMAT_NOT_SUPPORTED，如果 *src_image* 的圖像格式（圖像通道順序和數據類型）不被 *queue* 所在设备支持。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *src_image* 或 *dst_buffer* 分配內存失敗。
- CL_INVALID_OPERATION，如果 *command_queue* 所在设备不支持圖像（即表 4.3中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE）。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clEnqueueCopyBufferToImage

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

此函式所入隊的命令用來將緩衝對象拷貝到圖像對象中。

command_queue 就是拷貝命令要進入的隊列。*command_queue*、*src_buffer*、*dst_image* 必須位於同一 OpenCL 上下文中。

src_buffer 是一個緩衝對象。

dst_image 是一個圖像對象。

src_offset 即拷貝時源區域在 *src_buffer* 中的起始位置。

dst_origin 定義了目的區域的起始位置 (x, y, z)，單位像素；對於 2D 圖像陣列，即 (x, y) 以及圖像索引；對於 1D 圖像陣列，即 (x) 以及圖像索引。如果 *dst_image* 是 2D 圖像對象，*dst_origin*[2] 必須是 0。如果 *dst_image* 是 1D 圖像對象或 1D 圖像緩衝對象，*dst_origin*[1] 和 *dst_origin*[2] 都必須是 0。如果 *dst_image* 是 1D 圖像陣列對象，*dst_origin*[1] 即圖像索引。如果 *dst_image* 是 2D 圖像陣列對象，*dst_origin*[2] 即圖像索引。

region 定義了所拷貝區域的大小 $(width, height, depth)$，單位像素；對於 2D 圖像陣列，即 $(width, height)$ 以及圖像索引；對於 1D 圖像陣列，即 $(width)$ 以及圖像索引。如果 *dst_image* 是 2D 圖像對象或 1D 圖像陣列對象，*region*[2] 必須是 1。如果 *dst_image* 是 1D 圖像對象或 1D 圖像緩衝對象，*dst_origin*[1] 和 *region*[2] 必須是 1。

所拷貝的源區域的大小，即 *src_cb* 這樣計算：如果 *dst_image* 是 3D 圖像對象，則為 *width* * *height* * *depth* * *bytes/imageelement*；如果 *dst_image* 是 2D 圖像對象，則為 *width* * *height* * *bytes/imageelement*；如果 *dst_image* 是 2D 圖像陣列對象，則為 *width* * *height* * *arraysize* * *bytes/imageelement*；如果 *dst_image* 是 1D 圖像對象或 1D 圖像緩衝對象，則為 *width* * *bytes/imageelement*；如果 *dst_image* 是 1D 圖像陣列對象，則為 *width* * *arraysize* * *bytes/imageelement*；

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，即可回收並重新使用 *event_wait_list* 所關聯的內存。

event 會返回一個事件對象，用來標識此讀、寫命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。這時可以用 **clEnqueueBarrierWithWaitList** 來代替。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueCopyBufferToImage** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue*、*src_buffer* 和 *dst_image* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *src_buffer* 或 *dst_image* 無效；或者 *dst_image* 是由 *src_buffer* 創建的 1D 圖像緩衝對象。
- CL_INVALID_VALUE，如果 $(dst\_origin, region)$ 所指定的區域在 *dst_image* 中越限，或者 (src_offset, src_cb) 所指定的區域在 *src_buffer* 中越限。
- CL_INVALID_VALUE，如果 *dst_origin* 和 *region* 沒有遵守參數說明中描述的相應規則。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項：
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果 *src_buffer* 是一個子緩衝對象，並且創建他時所指定的參數 *offset* 沒有和 *queue* 所在设备的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 對齊。
- CL_INVALID_IMAGE_SIZE，如果 *dst_image* 的圖像大小（寬度、高度、指定的或自動計算的行間距或面間距）不被 *queue* 所在设备支持。
- CL_IMAGE_FORMAT_NOT_SUPPORTED，如果 *dst_image* 的圖像格式（圖像通道順序和數據類型）不被 *queue* 所在设备支持。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *src_buffer* 或 *dst_image* 分配內存失敗。
- CL_INVALID_OPERATION，如果 *command_queue* 所在设备不支持圖像（即，表 4.3中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE）。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.3.6 映射圖像對象

clEnqueueMapImage

```
void * clEnqueueMapImage (cl_command_queue command_queue,
                          cl_mem image,
                          cl_bool blocking_map,
                          cl_map_flags map_flags,
```

```
const size_t *origin,
const size_t *region,
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)
```

此函式所入隊的命令會將 *image* 的某個區域映射到主機的位址空間中，並返回新位址。

command_queue 必須是一個有效的命令隊列。

image 是一個圖像對象。 *command_queue* 與 *image* 必須位於同一 OpenCL 上下文中。

blocking_map 表明此映射操作是阻塞的還是非阻塞的。

如果 *blocking_map* 是 CL_TRUE，即映射命令是阻塞的，直到映射完成，**clEnqueueMap Image** 才會返回，應用可以用返回的指針訪問所映射區域的內容。

如果 *blocking_map* 是 CL_FALSE，即映射命令是非阻塞的，直到映射命令完成後，才能使用 **clEnqueueMapImage** 所返回的指針。參數 *event* 會返回一個事件對象，可用來查詢映射命令的執行情況。映射命令完成後，應用就可以使用 **clEnqueueMapImage** 所返回的指針訪問映射區域的內容了。

map_flags 是位欄，參見表 *5.5*。

origin 定義了所要映射區域的起始位置 (x, y, z)，單位像素；對於 2D 圖像陣列是 (x, y) 以及圖像索引；對於 1D 圖像陣列是 (x) 以及圖像索引。如果 *image* 是 2D 圖像對象，$origin[2]$ 必須是 0。如果 *image* 是 1D 圖像對象或 1D 圖像緩衝對象，$origin[1]$ 和 $origin[2]$ 都必須是 0。如果 *image* 是 1D 圖像陣列對象，$origin[2]$ 必須是 0，$origin[1]$ 是圖像在陣列中的索引。如果 *image* 是 2D 圖像陣列對象，$origin[2]$ 是圖像在陣列中的索引。

region 定義了映射區域的大小 $(width, height, depth)$，單位像素。對於 2D 圖像陣列對象，即 $(width, height)$ 以及圖像個數。對於 1D 圖像陣列對象，即 $(width)$ 以及圖像個數。如果 *image* 是 2D 圖像對象或 1D 圖像陣列對象，$region[2]$ 必須是 1。如果 *image* 是 1D 圖像對象或 1D 圖像緩衝對象，$region[1]$ 和 $region[2]$ 都必須是 0。

image_row_pitch 即所映射區域的掃描列間距，單位字節，不能是 NULL。

image_slice_pitch 對於 3D 圖像而言是 2D 平面的大小，對於 1D 或 2D 圖像陣列而言分別是每個 1D 或 2D 圖像的大小。對於 1D 或 2D 圖像，如果此參數不是 NULL 則返回零。對於 3D 圖像、1D 或 2D 圖像陣列，*image_slice_pitch* 不能是 NULL。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，即可回收並重新使用 *event_wait_list* 所關聯的內存。

event 會返回一個事件對象，用來標識此命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

errcode_ret 用來返回錯誤碼。如果是 NULL，則不會返回錯誤碼。

如果執行成功，**clEnqueueMapImage** 會返回所映射區域的指針，並將 *errcode_ret* 置為 CL_SUCCESS。否則，返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *image* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *image* 無效。
- CL_INVALID_VALUE，如果 $(origin, region)$ 所指定的區域越限，或者 *map_flags* 的值無效。
- CL_INVALID_VALUE，如果 *image_row_pitch* 是 NULL。
- CL_INVALID_VALUE，如果 *image* 是 3D 圖像、1D 或 2D 圖像陣列對象，而 *image_slice_pitch* 是 NULL。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項：

  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_INVALID_IMAGE_SIZE，如果 *image* 的大小（圖像寬度、高度、指定的或自動計算的行間距或面間距）不被 *queue* 所在设备支持。
- CL_IMAGE_FORMAT_NOT_SUPPORTED，如果 *image* 的圖像格式（圖像通道順序和數據類型）不被 *queue* 所在设备支持。
- CL_MAP_FAILURE，如果映射失敗。對於用 CL_MEM_USE_HOST_PTR 或 CL_MEM_ALLOC_HOST_PTR 創建的圖像對象不能出現此錯誤。
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST，如果映射操作是阻塞的，且 *event_wait_list* 中任一事件的執行狀態是負整數。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *buffer* 分配內存失敗。
- CL_INVALID_OPERATION，如果 *command_queue* 所在设备不支持圖像（即 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE，參見表 *4.3*）。
- CL_INVALID_OPERATION，如果創建 *image* 時指定了 CL_MEM_HOST_WRITE_ONLY 或 CL_MEM_HOST_NO_ACCESS，而在 *map_flags* 中設置了 CL_MAP_READ；或者創建 *image* 時指定了 CL_MEM_HOST_READ_ONLY 或 CL_MEM_HOST_NO_ACCESS，而在 *map_flags* 中設置了 CL_MAP_WRITE 或 CL_MAP_WRITE_INVALIDATE_REGION。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

所映射的區域起自 *origin*，對於 1D 圖像對象、1D 圖像緩衝對象或 1D 圖像陣列，其大小至少為 *region*[0]；對於 2D 圖像對象或 2D 圖像陣列，其大小至少為 (*image_row_pitch* * *region*[1])；對於 3D 圖像對象，其大小至少為 (*image_slice_pitch* * *region*[1])。如果訪問此區域外的內容，其結果未定義。

> 如果創建緩衝對象時在 *mem_flags* 中指定了 CL_MEM_USE_HOST_PTR，則：
> - 當 **clEnqueueMapImage** 的命令完成後，保證 **clCreateImage** 的 *host_ptr* 中所映射區域的內容是最新的。
> - **clEnqueueMapImage** 所返回的指針得自創建圖像對象時所用的 *host_ptr*。

被映射過的圖像對象使用 **clEnqueueUnmapMemObject** 進行解映射。參見節 *5.4.2*。

5.3.7 圖像對象查詢

對於所有內存對象都適用的資訊可以使用 **clGetMemObjectInfo** 來獲取，參見節 *5.4.5*。

而對於由 **clCreateImage** 創建的圖像對象而言，一些特定資訊使用如下函式獲取：

clGetImageInfo

```
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 即被查詢的圖像對象。

param_name 指明要查詢什麼資訊。他所支持的類型以及在 *param_value* 中返回的資訊如表 *5.9*所示。

param_value 用來存儲查詢結果。如果 *param_value* 是 NULL，則忽略。

param_value_size，即 *param_value* 內存塊的大小。其值必須 >= 表 *5.9*中返回型別的大小。

param_value_size_ret，即查詢結果的實際大小。如果 *param_value_size_ret* 是 NULL，則忽略。

如果執行成功，**clGetImageInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_VALUE，如果 param_name 的值無效，或者 param_value_size 的值 < 表 5.9中返回型別的大小並且 param_value 不是 NULL，或者 *param_name* 指的是某個擴展，但是此设备不支持相應擴展。
- CL_INVALID_MEM_OBJECT，如果 *image* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

表 5.9 **clGetImageInfo 所支持的** *param_names*

cl_image_info	**返回型別**
CL_IMAGE_FORMAT	cl_image_format
返回用 **clCreateImage** 創建 *image* 時所指定的圖像格式描述符。	
CL_IMAGE_ELEMENT_SIZE	size_t
返回 *image* 中每個元素的大小。一個元素由 n 個通道組成，其中 n 由 cl_image_format 給出。	
CL_IMAGE_ROW_PITCH	size_t
返回 *image* 中圖像元素的行間距。	
CL_IMAGE_SLICE_PITCH	size_t
對於 3D 圖像對象，返回 2D 面間距；對於 1D 或 2D 圖像陣列，返回每個圖像的大小；對於 1D 圖像對象、1D 圖像緩衝對象以及 2D 圖像對象，返回 0。	
CL_IMAGE_WIDTH	size_t
返回圖像寬度，單位像素。	
CL_IMAGE_HEIGHT	size_t
返回圖像高度，單位像素。對於 1D 圖像對象、1D 圖像緩衝對象以及 1D 圖像陣列對象，高度為 0。	
CL_IMAGE_DEPTH	size_t
返回圖像深度，單位像素。對於 1D 圖像對象、1D 圖像緩衝對象、2D 圖像對象，以及 1D 或 2D 圖像陣列對象，深度為 0。	
CL_IMAGE_ARRAY_SIZE	size_t
返回圖像陣列中圖像的個數。如果 *image* 不是圖像陣列，則返回 0。	
CL_IMAGE_BUFFER	cl_mem
返回 *image* 所關聯的緩衝對象。	
CL_IMAGE_NUM_MIP_LEVELS	cl_uint
返回 *image* 所關聯的 *num_mip_levels*。	
CL_IMAGE_NUM_SAMPLES	cl_uint
返回 *image* 所關聯的 *num_samples*。	

节 5.4 內存對象的查詢、解映射、遷移、保留和釋放

5.4.1 保留以及釋放內存對象

clRetainMemObject

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

此函式會使 *memobj* 的引用計數增一。執行成功後會返回 CL_SUCCESS。否則，返回下列錯誤的一種：

- CL_INVALID_MEM_OBJECT，如果 *memobj* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clCreateBuffer、**clCreateSubBuffer** 以及 **clCreateImage** 會實施隱式的保留。

clReleaseMemObject

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

此函式會使 *memobj* 的引用計數減一。執行成功後會返回 CL_SUCCESS。否則，返回下列錯誤的一種：

- CL_INVALID_MEM_OBJECT，如果 *memobj* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

一旦 *memobj* 的引用計數變成零，並且所有使用他的命令都執行完畢，此內存對象就會被刪除。如果 *memobj* 是一個緩衝對象，只有等到其所有子緩衝對象都被刪除後，才能將其刪除。

clSetMemObjectDestructorCallback

```
cl_int clSetMemObjectDestructorCallback (cl_mem memobj,
                 void (CL_CALLBACK *pfn_notify)(cl_mem memobj,
                       void *user_data),
                 void *user_data)
```

此函式會向 *memobj* 註冊一個回調函式。每次調用 **clSetMemObjectDestructorCallback** 都會將回調函式註冊到 *memobj* 的回調棧上。回調函式的調用順序與其註冊順序相反。對於一個內存對象而言，這些回調函式是在其資源被釋放以及將其刪除之前調用的。這樣就提供了一種機制，當 *host_ptr* (創建 *memobj* 時指定) 所引用的內存 (用來存儲內存對象的內容) 可以重新使用或是被釋放時，正在使用 *memobj* 的應用 (以及庫) 可以收到通知。

memobj 是一個內存對象。

pfn_notify 即應用所註冊的回調函式。此函式可能會被 OpenCL 實作異步調用。應用需要保證此函式是線程安全的。此函式的參數有：

- *memobj* 就是要被刪除的內存對象。當此函式被調用時，這個內存對象就已經無效了。此處提供他只是出於參考的目的。
- *user_data* 指向用戶提供的數據。

user_data 將在調用 *pfn_notify* 時作為引數 *user_data* 傳入。 *user_data* 可以是 NULL。

如果執行成功，**clSetMemObjectDestructorCallback** 會返回 CL_SUCCESS。否則返回下列錯誤碼之一：

- CL_INVALID_MEM_OBJECT，如果 *memobj* 無效。
- CL_INVALID_VALUE，如果 *pfn_notify* 是 NULL。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

當 *OpenCL* 實作調用回調函式時，*host_ptr* (如果是用 CL_MEM_USE_HOST_PTR 創建的內存對象) 所指向的內容是未定義的。此回調函式的一種典型應用就是用來釋放或重新使用 *host_ptr* 所指向的內存。

如果在回調函式中調用了下列 *OpenCL API*，其行為是未定義的：

- **clFinish**
- **clWaitForEvents**

如果在回調函式中有對下列 *OpenCL API* 的阻塞式調用，其行為也是未定義的：

- **clEnqueueReadBuffer**
- **clEnqueueReadBufferRect**
- **clEnqueueWriteBuffer**
- **clEnqueueWriteBufferRect**
- **clEnqueueReadImage**
- **clEnqueueWriteImage**
- **clEnqueueMapBuffer**
- **clEnqueueMapImage**
- **clBuildProgram**
- **clCompileProgram**
- **clLinkProgram**

如果應用需要在回調函式中等待上面某一例程執行完畢，請使用相應例程的非阻塞方式，並設置回調來完成剩下的工作。需要注意的是，如果回調 (或其他代碼) 在命令隊列中插入了命令，在隊列刷新 (*flush*) 前不要求這些命令已經開始執行。在標準用法中，阻塞式的入隊調用通過顯式的刷新隊列來達到目的。由於回調中不允許阻塞式的調用，對於那些會在隊列中插入命令的回調而言，需要在返回前調用 **clFlush** 或者將其安排在其他線程中隨後調用。

回調函式不能再用引數 *memobj* 調用其他 *OpenCL API*，否則其行為未定義。

5.4.2 內存對象的解映射

clEnqueueUnmapMemObject

```
cl_int clEnqueueUnmapMemObject (cl_command_queue command_queue,
                                cl_mem memobj,
                                void *mapped_ptr,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

此函式所入隊的命令可以將內存對象之前映射的區域解映射。假定使用 **clEnqueueMapBuffer** 或 **clEnqueueMapImage** 所返回的指針對 *memobj* 所進行的讀寫已經完畢。

command_queue 就是解映射命令要進入的隊列。

memobj 是一個內存對象，他必須與 *command_queue* 位於同一 OpenCL 上下文中。

mapped_ptr 即之前調用 **clEnqueueMapBuffer** 或 **clEnqueueMapImage** 所返回的主機位址。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，即可回收並重新使用 *event_wait_list* 所關聯的內存。

event 會返回一個事件對象，用來標識此命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。不過可以用 **clEnqueueBarrierWithWaitList** 來代替。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueUnmapMemObject** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_MEM_OBJECT，如果 *memobj* 無效。
- CL_INVALID_VALUE，如果 *mapped_ptr* 無效。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *memobj* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。

clEnqueueMapBuffer 和 **clEnqueueMapImage** 會使內存對象的映射計數增一。映射計數的初始值是零。對同一內存對象多次調用 **clEnqueueMapBuffer** 或 **clEnqueueMapImage**，會使其映射計數增加相應的數目。**clEnqueueUnmapMemObject** 會使內存對象的映射計數減一。

對於緩衝對象中被映射的那部分而言，**clEnqueueMapBuffer** 和 **clEnqueueMapImage** 充當同步點。

5.4.3 訪問所映射的區域

本節將介紹 OpenCL 命令是如何訪問所映射區域的。

如果某內存對象的某區域映射為可寫（即 **clEnqueueMap{Buffer | Image}** 的參數 *map_flags* 中設置了 CL_MAP_WRITE 或 CL_MAP_WIRTE_INVALIDATE_REGION），在將其解映射之前，對於此內存對象以及與其相關的內存對象（與此區域有重疊的子緩衝對象或 1D 圖像緩衝對象）而言，此區域的內容是未定義的。

對於一個內存對象以及與其相關的內存對象（與此區域有重疊的子緩衝對象或 1D 圖像緩衝對象）而言，命令隊列中的多個命令都可以將其中某個區域或重疊區域映射為可讀（即 *mem*

_flags = CL_MAP_READ）。對於內存對象中映射為可讀的區域，內核以及其他 OpenCL 命令（像 **clEnqueueCopyBuffer**）都可以讀取其內容。

對於一個內存對象以及與其相關的內存對象（與此區域有重疊的子緩衝對象或 1D 圖像緩衝對象）而言，如果映射（或解映射）的區域有重疊，則認為是錯誤，會導致 **clEnqueueMap{Buffer | Image}** 返回 CL_INVALID_OPERATION。

如果內存對象映射為可寫，則應用要保證在其他會讀寫其內容的內核或命令開始執行前將其解映射，否則其行為未定義。這些行為包括讀寫此內存對象，讀寫與其相關的內存對象（子緩衝對象或 1D 圖像緩衝對象），讀寫其父對象（如果此內存對象本身是子緩衝對象或 1D 圖像緩衝對象）。

而如果內存對象映射為可讀，則要保證其他會改寫其內容的內核或命令開始執行前將其解映射，否則其行為未定義。這些行為包括對此內存對象的寫操作，對與其相關的內存對象（子緩衝對象或 1D 圖像緩衝對象）的寫操作，對其父對象（如果此內存對象本身是子緩衝對象或 1D 圖像緩衝對象）的寫操作。

在解映射後，如果繼續使用之前映射後的指針來訪問所映射的區域，其行為未定義。

在遵守上面所列規則的前提下，可以將 **clEnqueueMap{Buffer | Image}** 所返回的指針作為引數 *ptr* 調用 **clEnqueue{Read | Write}Buffer**、**clEnqeue{Read | Write}BufferRect** 或 **clEnqueue{Read | Write}Image**。

5.4.4 內存對象的遷移

本節所描述的機制可用來將內存對象指派給某個设备。在創建內存對象時，用戶可能希望顯式的控制其位置。此機制可用來：

- 在使用前，保證在某個特定设备上分配對象。
- 將內存對象從一個设备遷移到另一個上面。

clEnqueueMigrateMemObjects

```
cl_int clEnqueueMigrateMemObjects(
                          cl_command_queue command_queue,
                          cl_uint num_mem_objects,
                          const cl_mem *mem_objects,
                          cl_mem_migration_flags flags,
                          cl_uint num_events_in_wait_list,
                          const cl_event *event_wait_list,
                          cl_event *event)
```

此函式所入隊的命令可以將一組內存對象強制關聯到某個设备上。典型的，在设备上執行命令時，會將所使用的內存對象隱式的遷移到此设备上。**clEnqueueMigrateMemObjects** 使得可以在相應命令執行前實施顯式的遷移。這使得用戶可以通過命令隊列的正規調度，提前改變內存對象的關聯，為後續命令做準備，同時可以避免隱式遷移所帶來的延遲。一旦 **clEnqueueMigrateMemObjects** 所返回的事件被標註為 CL_COMPLETE，*mem_objects* 中的所有內存對象就都成功遷移到了 *command_queue* 所在设备上。這些對象會一直在這個设备上，直到另一命令顯式或隱式地將其再次遷移。

clEnqueueMigrateMemObjects 也可以用來控制內存對象在創建後的初始位置，這樣可以避免第一條命令將其實體化時可能的開銷。

為了避免對內存對象的訪問相互衝突，與此命令相關的所有事件間的依賴關係由用戶負責。如果依賴關係有問題，則 **clEnqueueMigrateMemObjects** 會導致未定義的結果。

command_queue 是一個命令隊列。*mem_objects* 中的內存對象會被遷移到 *command_queue* 所在的设备上。如果設置了 CL_MIGRATE_MEM_OBJECT_HOST，則會遷移到主機上。

num_mem_objects 即 *mem_objects* 中內存對象的數目。

mem_objects 指向一組內存對象。

flags 是位欄，用來指定遷移選項。表 5.4中列出了可能的值。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，即可回收並重新使用 *event_wait_list* 所關聯的內存。

表 5.10 cl_mem_migration 的 值

cl_mem_migration
CL_MIGRATE_MEM_OBJECT_HOST
表明要將內存對象遷移到主機上，無視命令隊列。
CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED
表明遷移後，內存對象的內容是未定義的。遷移時不會招致遷移內容所帶來的開銷。

event 會返回一個事件對象，用來標識此命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。不過可以用 **clEnqueueBarrierWithWaitList** 來代替。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果執行成功，**clEnqueueMigrateMemObjects** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *mem_objects* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_MEM_OBJECT，如果 *mem_objects* 中的任一內存對象無效。
- CL_INVALID_VALUE，如果 *num_mem_objects* 是零或者 *mem_objects* 是 NULL。
- CL_INVALID_VALUE，如果 *flags* 的值無效。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *mem_objects* 中的內存對象分配內存失敗。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.4.5 內存對象的相關查詢

對於所有型別的內存對象（緩衝對象和圖像對象）所通用的一些資訊，可以由以下函式獲取:

clGetMemObjectInfo

```
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 指定所要查詢的內存對象。

param_name 指定要查詢什麼資訊。所支持資訊的種類以及 *param_value* 中返回的內容如表 *5.11*所示。

param_value 指向的內存用來存儲查詢結果。如果是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小（單位: 字節）。其值必須 >= 表 *5.11*中返回型別的大小。如果 *param_value* 是 NULL，則忽略。

param_value_size_ret 會返回查詢結果的實際大小。如果是 NULL，則忽略。

如果執行成功，**clGetMemObjectInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_VALUE，如果 *param_name* 不在支持之列，或者 *param_value_size* 的值 < 表 *5.11*中返回型別的大小且 *param_value* 不是 NULL。
- CL_INVALID_MEM_OBJECT，如果 *memobj* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

表 5.11 **clGetMemObjectInfo**所支持的*param_names*

cl_mem_info	返回类型
CL_MEM_TYPE	cl_mem_object_type
返回下列值中的一個: • CL_MEM_OBJECT_BUFFER，如果 *memobj* 是用 **clCreate{Buffer | SubBuffer}** 創建的。 • cl_image_desc.image_type，如果 *memobj* 是用 **clCreateImage** 創建的。	
CL_MEM_FLAGS	cl_mem_flags
返回用 **clCreate{Buffer | SubBuffer | Image}** 創建 *memobj* 時所指定的引數 *flags*。 如果 *memobj* 是子緩衝對象，則同時會返回由父對象所繼承的內存訪問限定符。	
CL_MEM_SIZE	size_t
返回 *memobj* 中用於數據存儲的內存的實際大小。單位: 字節。	
CL_MEM_HOST_PTR	void *
如果 *memobj* 是由 **clCreate{Buffer | Image}** 創建，且 *mem_flags* 中設置了 CL_MEM_USE_HOST_PTR，則返回引數 *host_ptr* 的值。否則返回 NULL。 如果 *memobj* 是由 **clCreateSubBuffer** 創建，則返回 $host\_ptr + origin$。*host_ptr* 即用 **clCreateBuffer** 創建 *memobj* 的父對象時 (*mem_flags* 中設置了 CL_MEM_USE_HOST_PTR) 引數的值。否則返回 NULL。	
CL_MEM_MAP_COUNT	cl_uint
映射計數。[1]	
CL_MEM_REFERENCE_COUNT	cl_uint
返回 *memobj* 的引用計數。[2]	
CL_MEM_CONTEXT	cl_context
返回創建內存對象時所指定的上下文。如果 *memobj* 是用 **clCreateSubBuffer** 創建的，則返回引數 *buffer* 所在的上下文。	
CL_MEM_ASSOCIATED_MEMOBJECT	cl_mem
返回 *memobj* 的父對象。即 **clCreateSubBuffer** 的引數 *buffer*。 否則返回 NULL。	
CL_MEM_OFFSET	size_t
如果 *memobj* 是由 **clCreateSubBuffer** 創建的子緩衝對象，則返回其偏移量。否則返回 0。	

[1] 在返回的那一刻，此映射計數就已過時。應用中一般不太適用。提供此特性主要是為了調試。
[2] 在返回的那一刻，此引用計數就已過時。應用中一般不太適用。提供此特性主要是為了檢測內存泄漏。

节 5.5 採樣器對象

採樣器對象所描述的是當內核讀取圖像時怎樣對其採樣。內核中使用內建函式讀取圖像時將採樣器作為引數之一。此引數可以是用 OpenCL 函式創建的採樣器對象，也可以是內核中聲明的採樣器。本節我們將討論如何利用 OpenCL 函式創建採樣器對象。

5.5.1 創建採樣器對象

clCreateSampler

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

此函式可以創建一個採樣器對象。至於採樣器怎樣工作請參考節 *6.12.14.1*。

context 是一個 OpenCL 上下文。

normalized_coords 指明圖像坐標是否已歸一化（如果 *normalized_coords* 是 CL_TRUE，則代表已歸一化，如果是 CL_FALSE，則未歸一化）。

addressing_mode 指明讀取圖像時如果坐標溢出如何處置。可以設置成 CL_ADDRESS_MIRRORED_REPEAT、CL_ADDRESS_REPEAT、CL_ADDRESS_CLAMP_TO_EDGE、CL_ADDRESS_CLAMP 或 CL_ADDRESS_NONE。

filter_mode 指明讀取圖像時採用哪種濾波模式。可以是 CL_FILTER_NEAREST 或 CL_FILTER_LINEAR。

errcode_ret 會返回相應的錯誤碼。如果是 NULL，則不會返回錯誤碼。

如果成功創建了採樣器對象，**clCreateSampler** 會將其返回，並將 *errcode_ret* 置為 CL_SUCCESS。否則，返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_INVALID_VALUE，如果下列三項中的任一項或其組合無效：
  - *addressing_mode*
  - *filter_mode*
  - *normalized_coords*
- CL_INVALID_OPERATION，如果 *context* 中的所有设备都不支持圖像（即 CL_DEVICE_IMAGE_SUPPORT 是 CL_FALSE，參見表 *4.3*）。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clRetainSampler

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

此函式會使 *sampler* 的引用計數增一。**clCreateSampler** 會實施隱式的 Retain。

如果執行成功，**clRetainSampler** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_SAMPLER，如果 *sampler* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clReleaseSampler

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

此函式會使 *sampler* 的引用計數減一。当 *sampler* 的引用計數降為 0，並且所有使用他的命令全部執行完畢，此採樣器就會被刪除。

如果執行成功，**clReleaseSampler** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_SAMPLER，如果 *sampler* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.5.2 採樣器對象的相關查詢

clGetSamplerInfo

```
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)
```

此函式會返回採樣器對象的相關資訊。

sampler 即所要查詢的採樣器。

param_name 指定要查詢什麼資訊。所支持的資訊類型以及 *param_value* 中所返回的內容如表 *5.12*所示。

param_value 指向的內存用來存儲查詢結果。如果是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小（單位：字節）。其值必須 >= 表 *5.12*中返回型別的大小。

param_value_size_ret 會返回查詢結果的實際大小。如果是 NULL，則忽略。

clGetCommandQueueInfo 所支持的 *param_name* 的值以及 *param_value* 中所返回的資訊如表 *5.12*所示。

表 5.12 **clGetSamplerInfo**所支持的*param_names*

cl_sampler_info	返回类型
CL_SAMPLER_REFERENCE_COUNT	cl_uint
返回 *sampler* 的引用計數。[1]	
CL_SAMPLER_CONTEXT	cl_context
返回創建採樣器時所指定的上下文。	
CL_SAMPLER_NORMALIZED_COORDS	cl_bool
sampler 的坐標是否歸一化。	
CL_SAMPLER_ADDRESSING_MODE	cl_addressing_mode
返回 *sampler* 的尋址模式。	
CL_SAMPLER_FILTER_MODE	cl_filter_mode
返回 *sampler* 的濾波模式。	

[1] 在返回的那一刻，此引用計數就已過時。應用中一般不太適用。提供此特性主要是為了檢測內存泄漏。

如果執行成功，**clGetSamplerInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_VALUE，如果 *param_name* 不在支持之列，或者 *param_value_size* 的值 < 表 5.12中返回型別的大小且 *param_value* 不是 NULL。
- CL_INVALID_SAMPLER，如果 *sampler* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

节 5.6 程式對象

一個 OpenCL 程式由一組內核組成，而這些內核就是程式源碼中以限定符 *__kernel* 聲明的函式。*__kernel* 函式可能會用到一些輔助函式以及常量數據，他們也是程式的一部分。OpenCL 編譯器可以為相應的目標设备以**在線**（**online**）或**離線**（**offline**）的方式生成程式執行體。

一個程式對象封裝了以下資訊:

- 所關聯的上下文;
- 程式源碼或二元碼;
- 最近成功構建的程式執行體、庫或編譯過的二元碼，他們所針對的设备清單，以及所使用的構建選項和構建日誌;
- 附着其上的內核對象的數目。

5.6.1 創建程式對象

clCreateProgramWithSource

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

此函式可以創建一個程式對象，並將 *strings* 中的源碼裝載到這個對象中。此對象所關聯的设备就是 *context* 所關聯的设备。*strings* 中的源碼是 OpenCL C 程式源碼、頭檔，或者自定義的代碼（對應與自定義设备，並且需要在線編譯器的支持）。

context 是一個 OpenCL 上下文。

strings 是一個字串陣列，有 *count* 個元素，字串可以 null 終止，這些字串就構成了源碼。

lengths 也是一個陣列，其中的元素代表字符個數（字串長度）。如果某個元素是零，則相應的字串以 null 終止。如果 *lengths* 是 NULL，則 *strings* 中所有字串都以 null 終止。所有大於零的值都不包括 null 終止符。

errcode_ret 會返回相應的錯誤碼。如果 *errcode_ret* 是 NULL，則不會返回錯誤碼。

如果成功創建了程式對象，則 **clCreateProgramWidthSource** 會將其返回，並將 *errcode_ret* 置為 CL_SUCCESS。否則，返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_INVALID_VALUE，如果 *count* 是零，或者 *strings* 或其中任一項是 NULL。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clCreateProgramWithBinary

```
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)
```

此函式會創建一個程式對象，並將 *binary* 中的內容裝載到此對象中。

context 是一個 OpenCL 上下文。

device_list 指向 *context* 中的设备。 *device_list* 不能是 NULL。所裝載的二元碼會在此清單中的设备上執行。

num_devices 即 *device_list* 中设备的數目。

此程式對象所關聯的设备就是 *device_list* 中的设备。而 *device_list* 中的设备又必須是 *context* 關聯的设备。

lengths 是一個陣列，其元素是所裝載程式二元碼的字節數。

binaries 也是一個陣列，每個元素都指向所要加載的程式二元碼。 *binaries*[*i*] 對應於 *device_list*[*i*]，其大小為 *lengths*[*i*]。 *lengths*[*i*] 不能是零，並且 *binaries*[*i*] 不能是 NULL。

binaries 中的內容為下列之一：

- 程式執行體，可以運行在 *context* 所關聯的设备上；
- 為 *context* 中的设备編譯過的程式；或
- 為 *context* 中的设备編譯過的庫。

程式二元碼可能包含下列之一，或二者兼有：

- 針對特定设备（device-specific）的代碼，和/或
- 針對特定實作（implementation-specific）的中間表示（intermediate representation，简称 IR）（將被轉換成針對特定设备的代碼）。

binary_status 將返回 *device_list* 中设备所對應的程式二元碼是否裝載成功。他是一個有 *num_devices* 個元素的陣列。如果 *device_list*[*i*] 所對應的程式二元碼裝載成功，則 *binary_status*[*i*] 為 CL_SUCCESS；如果 *lengths*[*i*] 是零或者 *binaries*[*i*] 是 NULL，則 *binary_status*[*i*] 為 CL_INVALID_VALUE；如果程式二元碼無效，則 *binary_status*[*i*] 為 CL_INVALID_BINARY。

errcode_ret 會返回相應的錯誤碼。如果 *errcode_ret* 是 NULL，則不會返回錯誤碼。

如果成功創建了程式對象，則 **clCreateProgramWithBinary** 會將其返回，並將 *errcode_ret* 置為 CL_SUCCESS。否則，返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_INVALID_VALUE，如果 *device_list* 是 NULL，或者 *num_devices* 是零。
- CL_INVALID_DEVICE，如果 *device_list* 中的 OpenCL 设备不在 *context* 中。
- CL_INVALID_VALUE，如果 *lengths* 或 *binaries* 是 NULL，或者任一項 *lengths*[*i*] 是零，或者 *binaries*[*i*] 是 NULL。
- CL_INVALID_BINARY，如果任一程式二元碼無效。 *binary_status* 會返回對應於每個设备的狀態。

- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

OpenCL 中，應用可以使用程式源碼或二元碼創建程式對象，並構建相應的程式執行體。這非常有用，當程式在系統中相應设备上第一次使用時，應用可以裝載程式源碼並編譯、鏈接，在線生成程式執行體。然後應用就可以查詢這些執行體並將其緩存起來，後續再使用此程式時就不必編譯、鏈接了。應用可以讀取並加載緩存起來的執行體，從而很大程度上減少應用花在初始化上的時間。

clCreateProgramWithBuiltInKernels

```
cl_program clCreateProgramWithBuiltInKernels(
                          cl_context context,
                          cl_uint num_devices,
                          const cl_device_id *device_list,
                          const char *kernel_names,
                          cl_int *errcode_ret)
```

此函式會創建一個程式對象，並將內建內核的相關資訊裝載到此對象中。

context 是一個 OpenCL 上下文。

device_list 指向 *context* 中的设备。 *device_list* 不能是 NULL。所加載的內建內核會在此清單中的设备上執行。

num_devices 即 *device_list* 中设备的數目。

此程式對象所關聯的设备就是 *device_list* 中的设备。而 *device_list* 中的设备又必須是 *context* 所關聯的设备。

kernel_names 是一組內建內核的名字，以分號分隔。

errcode_ret 會返回相應的錯誤碼。如果 *errcode_ret* 是 NULL，則不會返回錯誤碼。

如果成功創建了程式對象，則 **clCreateProgramWithBuiltInKernels** 會將其返回，並將 *errcode_ret* 置為 CL_SUCCESS。否則，返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_INVALID_VALUE，如果 *device_list* 是 NULL，或者 *num_devices* 是零。
- CL_INVALID_VALUE，如果 *kernel_names* 是 NULL，或者對於 *kernel_names* 中的任一內核， *device_list* 中的所有设备都不支持。
- CL_INVALID_DEVICE，如果 *device_list* 中的 OpenCL 设备不在 *context* 中。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clRetainProgram

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

此函式會使 *program* 的引用計數增一。 **clRetainProgram** 會實施隱式的保留。執行成功後會返回 CL_SUCCESS。否則，返回下列錯誤的一種：

- CL_INVALID_PROGRAM，如果 *program* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clReleaseProgram

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

此函式會使 *program* 的引用計數減一。當所有與 *program* 關聯的內核對象都被刪除，並且 *program* 的引用計數變成零，此程式對象就會被刪除。執行成功後會返回 CL_SUCCESS。否則，返回下列錯誤的一種：

- CL_INVALID_PROGRAM，如果 *program* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.6.2 構建程式執行體

clBuildProgram

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

此函式會由程式源碼或二元碼構建（編譯 & 鏈接）程式執行體，他可以在 *program* 所在上下文中的所有或部分设备上執行。對於用 **clCreateProgramWithSource** 或 **clCreateProgramWithBinary** 創建的 *program*，必須調用 **clBuildProgram** 來構建程式執行體。如果是用 **clCreateProgramWithBinary** 創建的 *program*，程式二元碼必須是可執行的（而不是編譯後的二元碼或庫）。

對於可執行的二元碼而言，可以用 **clGetProgramInfo**(*program*, CL_PROGRAM_BINARIES, ...) 進行查詢，也可以由 **clCreateProgramWithBinary** 用來創建新的程式對象。

program 就是程式對象。

device_list 指向 *program* 所關聯的设备。如果 *device_list* 是 NULL，則會為 *program* 所關聯的所有设备構建程式執行體。否則，僅為此清單中的设备構建程式執行體。

num_devices 即 *device_list* 中设备的數目。

options 指向一個以 null 終止的字串，用來描述構建選項。所支持的選項請參閱節 *5.6.4*。

pfn_notify 是應用所註冊的一個回調函式。在構建完程式執行體後（無論成功還是失敗）會被調用。如果 *pfn_notify* 不是 NULL，一旦可以開始構建，**clBuildProgram** 就會立刻返回，而不必等待構建完成。如果上下文、所要編譯鏈接的程式、设备清單以及構建選項都是有效的，以及實施構建所需的主機和设备資源都可用，那麼就可以開始構建了。如果 *pfn_notify* 是 NULL，直到構建完畢，**clBuildProgram** 才會返回。對此函式的調用可能是異步的。應用需要保證此函式是線程安全的。

user_data 在調用 *pfn_notify* 時作為引數傳入，可以是 NULL。

如果執行成功，則 **clBuildProgram** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_PROGRAM，如果 *program* 無效。
- CL_INVALID_VALUE，如果 *device_list* 是 NULL 而 *num_devices* 大於零，或者 *device_list* 不是 NULL 而 *num_devices* 等於零。
- CL_INVALID_VALUE，如果 *pfn_notify* 是 NULL，而 *user_data* 不是 NULL。
- CL_INVALID_DEVICE，如果 *device_list* 中的 OpenCL 设备不是 *program* 所關聯设备。
- CL_INVALID_BINARY，如果 *program* 是用 **clCreateProgramWithBinary** 創建的，但沒有為 *device_list* 中的设备裝載相應的程式二元碼。
- CL_INVALID_BUILD_OPTIONS，如果 *options* 所指定的構建選項無效。
- CL_INVALID_OPERATION，如果對於 *device_list* 中任一设备而言，之前調用 **clBuildProgram** 為 *program* 構建他所對應的程式執行體的動作還未完成。
- CL_COMPILER_NOT_AVAILABLE，如果是用 **clCreateProgramWithSource** 創建的 *program*，但是沒有可用的編譯器，即 CL_DEVICE_COMPILER_AVAILABLE 是 CL_FALSE，參見表 *4.3*。
- CL_BUILD_PROGRAM_FAILURE，如果構建程式執行體失敗。如果 **clBuildProgram** 沒有將此錯誤返回，則在構建完成時會將其返回。
- CL_INVALID_OPERATION，如果有附着到 *program* 上的內核對象。

- CL_INVALID_OPERATION，如果 *program* 不是由 **clCreateProgramWith{Source | Binary}** 創建的。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.6.3 編譯和鏈接的分離

OpenCL 1.2 對如何編譯和鏈接程式做了擴展，從而支持:

- 編譯階段、鏈接階段的分離。編譯階段可以將程式源碼編譯成二進制目標碼，而鏈接階段可以將其與其他目標碼鏈接成程式執行體。
- 內嵌頭檔。在 OpenCL 1.0 和 1.1 中，對於程式源碼所包含的頭檔而言，構建選項 -I 可以用來指定其搜索路徑。而 OpenCL 1.2 對其進行了擴展，允許目標碼中內嵌頭檔的源碼。
- 庫。可以使用鏈接器將目標碼和庫鏈接成程式執行體，或者創建一個新庫。

clCompileProgram

```
cl_int clCompileProgram(
                cl_program program,
                cl_uint num_devices,
                const cl_device_id *device_list,
                const char *options,
                cl_uint num_input_headers,
                const cl_program *input_headers,
                const char **header_include_names,
                void (CL_CALLBACK *pfn_notify)(
                                cl_program program,
                                void *user_data),
                void *user_data)
```

此函式可以編譯程式源碼。編譯前會先運行預處理器。編譯工作的目標设备是 *program* 關聯的所有设备或指定的那些设备。對於編譯後的二進制目標碼，可以使用 **clGetProgramInfo**(*program*, CL_PROGRAM_BINARIES, …) 對其進行查詢，也可以由 **clCreateProgramWithBinary** 用來創建新的程式對象。

program 是一個程式對象，即編譯的目標。

device_list 指向 *program* 所關聯的设备。如果 *device_list* 是 NULL，則編譯動作的目標设备為 *program* 關聯的所有设备。否則，僅為此清單中的设备實施編譯。

num_devices 即 *device_list* 中设备的數目。

options 指向一個以 null 終止的字串，用來描述編譯選項。所支持的選項請參閱節 *5.6.4*。

num_input_headers 即陣列 *input_headers* 中程式的數目。

input_headers 是一個陣列，元素為內嵌頭檔的程式，這些程式由 **clCreateProgramWithSource** 創建。

header_include_names 也是一個陣列，並與 *input_headers* 一一對應。每一項就是 *program* 源碼中一個內嵌頭檔的名字。*input_headers* 中對應項即為包含此頭檔源碼的程式對象。搜索頭檔時，內嵌頭檔比編譯選項 -I 所列目錄（節 *5.6.4.1*）有更高優先級。如果 *header_include_names* 中有多項名字相同的頭檔，則使用最先搜索到的。

例如，假設有如下程式源碼:

```
1  #include <foo.h>
2  #include <mydir/myinc.h>
3
4  __kernel void image_filter (int n, int m,
5                              __constant float *filter_weights,
6                              __read_only image2d_t src_image,
7                              __write_only image2d_t dst_image)
8  {
9          ...
10 }
```

這個內核包含了兩個頭檔 foo.h 和 mydir/myinc.h。下面來看怎樣將其傳遞並內嵌到程式對象中:

```
cl_program foo_pg = clCreateProgramWithSource(context,
                                1, &foo_header_src, NULL, &err);
cl_program myinc_pg = clCreateProgramWithSource(context,
                                1, &myinc_header_src, NULL, &err);

// let's assume the program source described above is given
// by program_A and is loaded via clCreateProgramWithSource

cl_program input_headers[2] = { foo_pg, myinc_pg };
char * input_header_names[2] = { "foo.h", "mydir/myinc.h" };
clCompileProgram(program_A,
                 0, NULL,         // num_devices & device_list
                 NULL,            // compile_options
                 2,               // num_input_headers
                 input_headers,
                 input_header_names,
                 NULL, NULL);     // pfn_notify & user_data
```

pfn_notify 是應用所註冊的一個回調函式。在編譯完成後（無論成功還是失敗）會被調用。如果 *pfn_notify* 不是 NULL，一旦可以開始編譯，**clCompileProgram** 就會立刻返回，而不必等待編譯完成。如果上下文、所要編譯的程式源碼、设备清單、所輸入的頭檔以及相應的程式，還有構建選項都是有效的，並且實施編譯所需的主機和设备資源都可用，那麼就可以開始進行編譯了。如果 *pfn_notify* 是 NULL，直到編譯完畢，**clCompileProgram** 才會返回。對此函式的調用可能是異步的。應用需要保證此函式是線程安全的。

user_data 在調用 *pfn_notify* 時作為引數傳入，可以是 NULL。

如果執行成功，則 **clCompileProgram** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_PROGRAM，如果 *program* 無效。
- CL_INVALID_VALUE，如果 *device_list* 是 NULL 而 *num_devices* 大於零，或者 *device_list* 不是 NULL 而 *num_devices* 等於零。
- CL_INVALID_VALUE，如果 *num_input_headers* 是零，而 *header_include_names* 或 *input_headers* 不是 NULL；或者 *num_input_headers* 不是零，而 *header_include_names* 或 *input_headers* 是 NULL。
- CL_INVALID_VALUE，如果 *pfn_notify* 是 NULL，而 *user_data* 不是 NULL。
- CL_INVALID_DEVICE，如果 *device_list* 中的 OpenCL 设备不是 *program* 所關聯设备。
- CL_INVALID_COMPILER_OPTIONS，如果 *options* 所指定的編譯選項無效。
- CL_INVALID_OPERATION，如果對於 *device_list* 中任一设备而言，之前調用 **clCompileProgram** 或 **clBuildProgram** 為 *program* 編譯或構建他所對應的程式執行體的動作還未完成。
- CL_COMPILER_NOT_AVAILABLE，如果沒有可用的編譯器，即 CL_DEVICE_COMPILER_AVAILABLE 是 CL_FALSE，參見表 *4.3*。
- CL_COMPILE_PROGRAM_FAILURE，如果編譯程式源碼失敗。如果 **clCompileProgram** 沒有將此錯誤返回，則在編譯結束後會將其返回。
- CL_INVALID_OPERATION，如果有附着到 *program* 上的內核對象。
- CL_INVALID_OPERATION，如果沒有 *program* 的源碼，即 *program* 不是由 **clCreateProgramWithSource** 創建的。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clLinkProgram

```
cl_program clLinkProgram (cl_context context,
                          cl_uint num_devices,
                          const cl_device_id *device_list,
```

```
                                const char *options,
                                cl_uint num_input_programs,
                                const cl_program *input_programs,
                                void (CL_CALLBACK *pfn_notify)(
                                                        cl_program program,
                                                        void *user_data),
                                void *user_data,
                                cl_int *errcode_ret)
```

此函式可以將一組目標碼和庫鏈接成執行體，並創建一個包含這個執行體的程式對象。對於這個執行體，可以使用 **clGetProgramInfo**(*program*, CL_PROGRAM_BINARIES, ...) 對其進行查詢，也可以由 **clCreateProgramWithBinary** 用來創建新的程式對象。

如果 *device_list* 不是 NULL，則所返回的程式對象關聯的就是 *device_list* 中的设备；否則，關聯的就是 *context* 中的设备。

context 是一個 OpenCL 上下文。

device_list 指向 *context* 中的设备。如果 *device_list* 是 NULL，則鏈接動作的目標设备是 *context* 中的所有设备。否則，僅為此清單中的设备實施鏈接。

num_devices 即 *device_list* 中设备的數目。

options 指向一個以 null 終止的字串，用來描述鏈接選項。所支持的選項請參閱節 *5.6.5*。

num_input_programs 即 *input_programs* 中程式的數目。

input_programs 是一個陣列，每一項都是一個程式對象，他們是編譯過的二元碼或庫，將被鏈接成程式執行體。對於目標設備而言，有以下幾種情況:

- *input_programs* 中的所有程式都包含針對此设备的二元碼或庫。這種情況下，就為此设备實施鏈接來生成程式執行體。
- 所有程式都不包含針對此设备的二元碼或庫。這種情況下，不會為此设备實施鏈接，也不會生成對應的程式執行體。
- 所有其他情況都會返回錯誤 CL_INVALID_OPERATION。

pfn_notify 是應用所註冊的一個回調函式。在鏈接完成後 (無論成功還是失敗) 會被調用。

如果 *pfn_notify* 不是 NULL，一旦可以開始鏈接，**clLinkProgram** 就會立刻返回，而不必等待鏈接完成。一旦鏈接完成，*pfn_notify* 就會被調用，並返回一個程式對象 (跟 **clLinkProgram** 返回的一樣)。應用可以查詢鏈接的狀態以及日誌。此函式可能會被異步調用。應用需要保證此函式是線程安全的。

如果 *pfn_notify* 是 NULL，直到鏈接完畢，**clLinkProgram** 才會返回。

user_data 在調用 *pfn_notify* 時作為引數傳入，可以是 NULL。

如果上下文、设备、輸入的程式以及鏈接選項都是有效的，並且實施鏈接所需的主機和设备資源都可用，那麼就可以開始進行鏈接了。如果可以開始鏈接，**clLinkProgram** 會返回一個非零的程式對象。

如果 *pfn_notify* 是 NULL，鏈接成功會將 *errcode_ret* 置為 CL_SUCCESS，鏈接失敗會將其置為 CL_LINK_FAILURE。

如果 *pfn_notify* 不是 NULL，**clLinkProgram** 不必等到鏈接完成，一旦可以開始鏈接，就可以將 *errcode_ret* 置為 CL_SUCCESS 並返回。*pfn_notify* 會返回 CL_SUCCESS 或 CL_LINK_FAILURE 以表示鏈接成功與否。

否則 **clLinkProgram** 會返回 NULL，並在 *errcode_ret* 中返回相應的錯誤碼。應用應當查詢此程式對象的鏈接狀態，以判定鏈接是否成功。返回的錯誤碼可以是:

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_INVALID_VALUE，如果 *device_list* 是 NULL 而 *num_devices* 大於零，或者 *device_list* 不是 NULL 而 *num_devices* 等於零。
- CL_INVALID_VALUE，如果 *num_input_programs* 是零，且 *input_programs* 是 NULL；或者 *num_input_programs* 是零，而 *input_programs* 不是 NULL；或者 *num_input_programs* 不是零，而 *input_programs* 是 NULL。
- CL_INVALID_PROGRAM，如果 *input_programs* 中的程式對象無效。
- CL_INVALID_VALUE，如果 *pfn_notify* 是 NULL，而 *user_data* 不是 NULL。
- CL_INVALID_DEVICE，如果 *device_list* 中的任一设备不屬於 *context*。

- CL_INVALID_LINKER_OPTIONS，如果 *options* 中的連接器選項無效。
- CL_INVALID_OPERATION，如果沒有遵守上面對參數 *input_programs* 的介紹中所列的規則。
- CL_LINKER_NOT_AVAILABLE，如果沒有可用的連接器，即 CL_DEVICE_LINKER_AVAILABLE 是 CL_FALSE，參見表 *4.3*。
- CL_LINK_PROGRAM_FAILURE，如果鏈接失敗。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.6.4 編譯器選項

編譯器選項分為以下幾種：預處理器選項、數學本徵選項、優化控制選項以及其他選項。本規範定義了一套標準選項，用於在線或離線構建程式執行體，所有 OpenCL C 編譯器都必須支持。也可以通過增加一些針對特定供應商或特定平台的選項對其進行擴展。

5.6.4.1 預處理器選項

這些選項用於控制預處理器，在真正編譯前對源碼進行預處理。

-D *name*

預定義一個名為 *name* 的巨集，其值為 1。

-D *name*=*definition*

就像預處理指示（directive） #define 那樣，在翻譯的第三個階段中會將 *definition* 的內容符號化並處理。不過遇到內嵌的換行符會進行截斷。

對於選項 -D，會按照在 **clBuildProgram** 或 **clCompileProgram** 的引數 *options* 中出現的順序進行處理。

-I *dir*

將 *dir* 加入到頭檔的搜索路徑中。

5.6.4.2 數學本徵選項

這些選項會控制編譯器中浮點算術相關的行為。他們會影響速度與正確性之間的權衡。

-cl-single-precision-constant

將雙精度浮點常數視為單精度浮點常數。

-cl-denorms-are-zero

此選項用來控制如何處理單、雙精度浮點去規格化數（denormalized number）。如果作為構建選項，單精度去規格化數會被刷成零；此時如果支持雙精度，則雙精度去規格化數也可能會被刷成零。此選項僅作為性能建議，如果设备支持單精度（或雙精度）去規格化數，OpenCL 編譯器可以選擇不將其刷成零。

如果设备不支持單精度（或雙精度）去規格化數，即 CL_DEVICE_SINGLE_FP_CONFIG 中沒有設置 CL_FP_DENORM，則對於單精度數此選項會被忽略。

如果设备不支持雙精度，或者支持雙精度但不支持雙精度（或雙精度）去規格化數，即 CL_DEVICE_DOUBLE_FP_CONFIG 中沒有設置 CL_FP_DENORM，則對於雙精度數此選項會被忽略。

此選項僅對程式中的標量或矢量浮點變數以及其上的運算起作用，對於讀寫圖像對象則無效。

-cl-fp32-correctly-rounded-divide-sqrt

對於程式源碼中的單精度浮點除法（x/y 和 1/x）和 sqrt 而言，應用可以使用此選項來保證對他們進行正確的捨入；而如果沒有使用此選項，則他們的最小數值精確度在節 *7.4*中定義。

只有在對應设备的 CL_DEVICE_SINGLE_FP_CONFIG（參見表 *4.3*）中設置了 CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT 時，才能使用此選項；否則編譯會失敗。

5.6.4.3 優化選項

這些選項控制各種優化。打開優化選項會使編譯器嘗試改進性能和/或代碼大小，代價就是犧牲編譯時間，還可能降低調試程式的能力。

-cl-opt-disable

此選項會去能所有優化。缺省情況下優化是打開的。

下列選項會控制編譯器中浮點算術相關的行為。這些選項以犧牲正確性為代價來換取性能的提升，必須明確使能。由於他們可能會導致不正確的輸出，而這些輸出則依賴於數學函式對 IEEE 754 標準的具體實作，因此缺省這些選項都是關閉的。

-cl-mad-enable

允許用 `mad` 代替 `a * b + c`。`mad` 會用較低的精確度來計算 `a * b + c`。例如，一些 OpenCL 设备實現的 `mad` 在加 `c` 前會對 `a * b` 進行截斷。

-cl-no-signed-zeros

允許在進行浮點算術運算時忽略零的正負。在 IEEE 754 中，對於 `+0.0` 和 `-0.0` 的行為是不同的，這會導致不能對像 `x + 0.0` 或 `0.0 * x` 這樣的算式進行簡化（即使只帶有 `-cl-finite-math`）。

-cl-unsafe-math-optimizations

允許對浮點算術的優化：

- 假定參數和結果都是有效的；
- 可能違反 IEEE 754 的標準；以及
- 可能違反節 *7.4*中所定義的對單雙精度浮點數的數值符合性的要求，還有節 *7.5*中邊界條件下的行為。

此選項包含了 `-cl-no-signed-zeros` 和 `-cl-mad-enable`。

-cl-finite-math-only

在執行浮點算術運算時，可以假定引數和結果都不是 NaN 或 $\pm\infty$。這可能違反節 *7.4*中所定義的對單雙精度浮點數的數值符合性的要求，還有節 *7.5*中邊界條件下的行為。

-cl-fast-relaxed-math

相當於 `-cl-finite-math-only` 和 `-cl-unsafe-math-optimizations` 的組合。對浮點算術的優化可以違反 IEEE 754 的標準，也可以違反節 *7.4*中所定義的對單雙精度浮點數的數值符合性的要求，還有節 *7.5*中邊界條件下的行為。此選項會使得程式中定義預處理器巨集 `__FAST_RELAXED_MATH__`。

5.6.4.4 請求或抑制警告的選項

警告（warning）是一些診斷訊息，表示一些構造不屬於錯誤但是有風險，或者可能是錯誤。下列選項是獨立於語言的，不是用來使能特定警告，而是用來控制 OpenCL 編譯器所產生的診斷訊息的種類。

-w

禁止所有警告訊息。

-Werror

把所有警告都當成錯誤。

5.6.4.5 控制 OpenCL C 版本的選項

下面的選項控制編譯器所使用的 OpenCL C 的版本。

-cl-std=

確定所使用的 OpenCL C 語言的版本。必須為此選項提供一個值。有效的值有：

- CL1.1 ——所有 OpenCL C 程式都使用 OpenCL 1.1 規範第 *6*章中所定義的特性。
- CL1.2 ——所有 OpenCL C 程式都使用 OpenCL 1.2 規範第 *6*章中所定義的特性。

對於任何一個设备，如果其 CL_DEVICE_OPENCL_C_VERSION 為 OpenCL C 1.0，使用的選項卻是 -cl-std=1.1，則對 **clBuildProgram** 或 **clCompileProgram** 的調用都會**失敗**；而如果其 CL_DEVICE_OPENCL_C_VERSION 為 OpenCL C 1.1，使用選項卻是 -cl-std=1.2，對 **clBuildProgram** 或 **clCompileProgram** 的調用也都會**失敗**。

如果沒有指定選項 -cl-std，則使用相應设备的 CL_DEVICE_OPENCL_C_VERSION 作為 OpenCL C 的版本為其編譯程式。

5.6.4.6 用於查詢內核引數資訊的選項

-cl-kernel-arg-info

此選項允許編譯器將內核引數的資訊存儲到程式執行體中。所存儲的資訊包括引數名、引數型別、所使用的位址限定符以及訪問限定符。至於怎樣查詢這些資訊請參考 **clGetKernelArgInfo** 的描述。

5.6.5 連接器選項

此規範定義了一組標準的連接器選項，OpenCL C 編譯器在鏈接程式時必須支持。這些選項分成兩種：庫鏈接選項和程式鏈接選項。也可以通過增加一些針對特定供應商或特定平台的選項對其進行擴展。

5.6.5.1 庫鏈接選項

將編譯後的二元碼鏈接成庫時可以指定下列選項。

-create-library

將編譯後的二元碼（由 **clLinkProgram** 的引數 *input_programs* 提供）鏈接成庫。

-enable-link-options

鏈接器在將庫與程式執行體進行鏈接時，可以按照鏈接選項修正庫的行為（參見節 *5.6.5.2*）。此選項必須與 -create-library 一起使用。

5.6.5.2 程式鏈接選項

在鏈接程式執行體時可以使用下列選項：

- -cl-denorms-are-zero
- -cl-no-signed-zeroes
- -cl-unsafe-math-optimizations
- -cl-finite-math-only
- -cl-fast-relaxed-math

這些選項在節 *5.6.4.2*和 節 *5.6.4.3*中有所描述。鏈接器可能會將這些選項應用到 **clLinkProgram** 中的所有目標碼上；也可能只應用到以 -enable-link-option 創建的庫上。

5.6.6 卸載編譯器

clUnloadPlatformCompiler

```
cl_int clUnloadPlatformCompiler (cl_platform_id platform)
```

此函式可用來釋放 OpenCL 編譯器為 *platform* 分配的資源。這僅作為應用的建議，不保證將來不再使用編譯器，也不保證真正卸載編譯器。如果在 **clUnloadPlatformCompiler** 之後調用 **cl{Build | Compile | Link}Program** 來構建程式執行體，有必要的話會重新加載編譯器。

如果執行成功，**clUnloadPlatformCompiler** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_PLATFORM，如果 *platform* 無效。

5.6.7 程式對象相關的查詢

clGetProgramInfo

```
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)
```

此函式會返回程式對象的相關資訊。

program 即所要查詢的程式對象。

param_name 指定要查詢什麼資訊。對於所支持的資訊類型以及 *param_value* 中返回的資訊，請參見表 *5.13*。

param_value 指向的內存用來存儲查詢結果。如果 *param_value* 是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小。其值必須 >= 表 *5.13*中返回型別的大小。

param_value_size_ret 返回查詢結果的實際大小。如果 *param_value_size_ret* 是 NULL，則忽略。

表 5.13α **clGetProgramInfo 所支持的** *param_names*

cl_program_info	返回型別
CL_PROGRAM_REFERENCE_COUNT	cl_uint
返回 *program* 的引用計數。[1]	
CL_PROGRAM_CONTEXT	cl_context
返回創建程式對象時所指定的上下文。	
CL_PROGRAM_NUM_DEVICES	cl_uint
返回 *program* 所關聯設備的數目。	
CL_PROGRAM_DEVICES	cl_device_id[]
返回 *program* 所關聯的設備。可以是創建程式對象時所用上下文中的設備；也可以是用 **clCreateProgramWithBinary** 創建程式對象時所指定的設備中的一個子集。	
CL_PROGRAM_SOURCE	char[]
返回傳給 **clCreateProgramWithSource** 的程式源碼。返回的字串中將所有源碼都串在了一起，並以 null 終止（原有的 null 會被剝離）。 如果 *program* 是用 **clCreateProgramWithBinary** 或 **clCreateProgramWithBuiltinKernels** 創建的，可能返回空字串，或者相應的程式源碼，這取決於二元碼中是否包含程式源碼。 *param_value_size_ret* 中會返回源碼中字符的實際數目，包含 null 終止符。	
CL_PROGRAM_BINARY_SIZES	size_t[]
返回一個陣列，內含 *program* 所關聯的所有設備對應的程式二元碼（可以是可執行二元碼、編譯過的二元碼或者庫的二元碼）的大小。陣列的大小等於與 *program* 所關聯的設備的個數。如果任一設備沒有對應的二元碼，則返回零。 如果 *program* 是用 **clCreateProgramWithBuiltinKernels** 創建的，可能陣列中的所有元素都是零。	
CL_PROGRAM_BINARIES	unsigned char *[]
返回一個陣列，內含 *program* 所關聯的所有設備對應的程式二元碼（可以是可執行二元碼、編譯過的二元碼或者庫的二元碼）。對於 *program* 所關聯的每個設備而言，所返回的二元碼可能是用 **clCreateProgramWithBinary** 創建 *program* 時所指定的二元碼，或者是用 **clBuildProgram** 或 **clLinkProgram** 所生成的可執行二元碼。如果是用 **clCreateProgramWithSource** 生成的 *program*，則返回的是 **clBuildProgram**、**clCompileProgram** 或 **clLinkProgram** 所生成的二元碼。所返回的可能是針對特定實作的中間表示（又叫做 IR），或針對特定設備的可執行二元碼，也可能二者兼有。至於二元碼中會返回哪種資訊由 OpenCL 實作來決定。 *param_value* 指向一個包含 n 個指針的陣列，所有指針都由調用者分配，其中 n 就是 *program* 所關聯設備的數目。對於每個指針需要分配多少內存，可以通過此表中的 CL_PROGRAM_BINARY_SIZES 進行查詢。 實作可以使用陣列中的元素來存儲特定設備所對應的程式二元碼，如果有的話。至於陣列中的元素都對應於哪個設備，可以使用 CL_PROGRAM_DEVICES 進行查詢。 CL_PROGRAM_BINARIES 和 CL_PROGRAM_DEVICES 所返回的陣列具有一對一的關係。	

表 5.13β　clGetProgramInfo 所支持的 *param_names*

CL_PROGRAM_NUM_KERNELS	size_t
返回 *program* 中聲明的內核總數。對於 *program* 所關聯的设备而言，至少要為其中之一成功構建了可執行程式，然後才能使用此資訊。	
CL_PROGRAM_KERNEL_NAMES	char[]
返回 *program* 中內核的名字，以分號間隔。對於 *program* 所關聯的设备而言，至少要為其中之一成功構建了可執行程式，然後才能使用此資訊。	

[1] 在返回的那一刻，此引用計數就已過時。應用中一般不太適用。提供此特性主要是為了檢測內存泄漏。

如果執行成功，**clGetProgramInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_VALUE，如果 *param_name* 無效，或者 *param_value_size* < 表 *5.13*中返回型別的大小，且 *param_value* 不是 NULL。
- CL_INVALID_PROGRAM，如果 *program* 無效。
- CL_INVALID_PROGRAM_EXECUTABLE，如果 *param_name* 是：
  - CL_PROGRAM_NUM_KERNELS
  - 或 CL_PROGRAM_KERNEL_NAMES，

  並且在 *program* 所關聯的设备中至少有一個设备所對應的程式執行體沒有成功構建。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clGetProgramBuildInfo

```
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)
```

此函式會返回程式對象中某個设备所對應的構建資訊。

program 即所要查詢的程式對象。

device 指定要查詢哪個设备的構建資訊。 *device* 必須是 *program* 所關聯的设备。

param_name 指定要查詢什麼資訊。對於所支持的資訊類型以及 *param_value* 中返回的資訊，請參見表 *5.14*。

param_value 指向的內存用來存儲查詢結果。如果 *param_value* 是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小。其值必須 >= 表 *5.14*中返回型別的大小。

param_value_size_ret 返回查詢結果的實際大小。如果 *param_value_size_ret* 是 NULL，則忽略。

如果執行成功，**clGetProgramBuildInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_DEVICE，如果 *device* 不是與 *program* 相關聯的设备。
- CL_INVALID_VALUE，如果 *param_name* 無效，或者 *param_value_size* < 表 *5.14*中返回型別的大小，且 *param_value* 不是 NULL。
- CL_INVALID_PROGRAM，如果 *program* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

為父設備構建的程式二元碼（編譯過的二元碼、庫的二元碼或者可執行的二元碼）可以被他的所有子設備所使用。如果沒有為某個子設備構建相應的程式二元碼，則會使用其父設備的。

一個设备的程式二元碼，無論是為 **clCreateProgramWithBinary** 所指定的，還是用 **clGetProgramInfo** 查詢得到的，都可以用到相應根設備、及其任意級別的子設備上。

表 5.14 **clGetProgramBuildInfo** 所支持的 *param_names*

cl_program_buid_info	返回型別
CL_PROGRAM_BUILD_STATUS	cl_build_status
返回構建、編譯或鏈接的狀態，在 *program* 上為 *device* 最後實施的那個（**clBuildProgram**、**clCompile Program** 或 **clLinkProgram**）。 可以是下列之一： • CL_BUILD_NONE。如果三個都沒有實施過。 • CL_BUILD_ERROR。如果最後實施的那個產生了錯誤。 • CL_BUILD_SUCCESS。如果最後實施的那個成功了。 • CL_BUILD_IN_PROGRESS。如果最後實施的那個還未完成。	
CL_PROGRAM_BUILD_OPTIONS	char[]
返回構建、編譯或鏈接的選項，即最後實施的那個的參數 *options*。如果狀態是 CL_BUILD_NONE，則返回空字串。	
CL_PROGRAM_BUILD_LOG	char[]
返回最後實施的那個的日誌。如果狀態是 CL_BUILD_NONE，則返回空字串。	
CL_PROGRAM_BINARY_TYPE	cl_program_binary_type
返回 *device* 所對應的二元碼的類型。可以是下列之一： • CL_PROGRAM_BINARY_TYPE_NONE，沒有對應的二元碼。 • CL_PROGRAM_BINARY_TYPE_COMPILED_OBJECT，有編譯過的二元碼。如果 *program* 是使用 **clCreate ProgramWithSource** 創建並使用 **clCompileProgram** 編譯的，或者是用 **clCreateProgramWithBinary** 裝載的編譯過的二元碼，都屬於這種情況。 • CL_PROGRAM_BINARY_TYPE_LIBRARY，有庫的二元碼。如果用 **clLinkProgram** 創建 *program* 時指定了鏈接選項 -create-library，或者是用 **clCreateProgramWithBinary** 裝載的庫的二元碼，都屬於這種情況。 • CL_PROGRAM_BINARY_TYPE_EXECUTABLE，有可執行的二元碼。如果用 **clLinkProgram** 創建 *program* 時沒有指定鏈接選項 -create-library，或者是用 **clCreateProgramWithBinary** 裝載的可執行的二元碼，都屬於這種情況。	

节 5.7 內核對象

內核就是程式中聲明的一個函式。對於程式中的任一函式，都可以通過加上限定符 *__kernel* 將其標識為內核。內核對象中封裝了程式中的某個 *__kernel* 函式以及執行此函式時所用的引數。

5.7.1 創建內核對象

clCreateKernel

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

此函式可用來創建內核對象。

program 是一個程式對象，帶有成功構建的執行體。

kernel_name 是程式中一個函式的名字，聲明時帶有限定符 *__kernel*。

errcode_ret 會返回相應的錯誤碼。如果 *errcode_ret* 是 NULL，則不會返回錯誤碼。

如果成功創建了內核對象，則 **clCreateKernel** 會將其返回，並將 *errcode_ret* 置為 CL_SUCCESS。否則，返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_PROGRAM，如果 *program* 無效。
- CL_INVALID_PROGRAM_EXECUTABLE，如果沒有為 *program* 成功構建的執行體。
- CL_INVALID_KERNEL_NAME，如果 *program* 中找不到 *kernel_name*。
- CL_INVALID_KERNEL_DEFINITION，如果 *kernel_name* 的函式定義（如參數個數、參數型別）在已經具有對應執行體的那些设备上有所區別。
- CL_INVALID_VALUE，如果 *kernel_name* 是 NULL。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clCreateKernelsInProgram

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

此函式會為 *program* 中的所有內核函式創建對應的內核對象。如果某個 *__kernel* 函式的定義在已經具有執行體的那些设备上不完全一樣，則不會為其創建內核對象。

program 是一個程式對象，具有成功構建的執行體。

num_kernels 即 *kernels* 中 cl_kernel 的數目。

kernels 用來存儲所返回的內核對象。如果 *kernels* 是 NULL，則忽略；否則，*num_kernels* 的值必須大於或等於 *program* 中內核的數目。

num_kernels_ret 即 *program* 中內核的數目。如果 *num_kernels_ret* 是 NULL，則忽略。

如果執行成功，**clCreateKernelsInProgram** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_PROGRAM，如果 *program* 無效。
- CL_INVALID_PROGRAM_EXECUTABLE，如果 *program* 中的任一设备沒有對應的執行體。
- CL_INVALID_VALUE，如果 *kernels* 不是 NULL，或者 *num_kernels* 的值小於 *program* 中內核的數目。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

只有當已將有效的程式源碼或者二元碼加載到了程式對象中，並且至少為一個此對象所關聯的设备成功構建了執行體時，才能創建內核對象。

對於一個程式對象而言，如果有關聯的內核對象，則不允許改變程式執行體，也就是說此時調用 **clBuildProgram** 或 **clCompileProgram** 會返回 CL_INVALID_OPERATION。*program* 所在的 OpenCL 上下文即為 *kernel* 所在的上下文。*program* 所關聯的设备即為 *kernel* 所關聯的设备。對於程式對象中的设备而言，只要有對應的程式執行體，就可以執行此對象中聲明的內核。

clRetainKernel

```
cl_int clRetainKernel (cl_kernel kernel)
```

此函式會使 *kernel* 的引用計數增一。執行成功後會返回 CL_SUCCESS。否則，返回下列錯誤的一種：

- CL_INVALID_KERNEL，如果 *kernel* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clCreateKernel 或 **clCreateKernelsInProgram** 會實施隱式的保留。

clReleaseKernel

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

此函式會使 *kernel* 的引用計數減一。執行成功後會返回 CL_SUCCESS。否則，返回下列錯誤的一種：

- CL_INVALID_KERNEL，如果 *kernel* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

當 *kernel* 的引用計數變成零，隊列中的所有命令都不再需要此對象時，此對象就會被刪除。

5.7.2 設置內核引數

要想執行內核，必須設置內核引數。

clSetKernelArg

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

此函式可用來設置內核的某個引數。

kernel 是一個內核對象。

arg_index 是引數索引。內核引數數的索引值從最左邊的 0 開始，一直到 $n-1$，其中 n 是引數的總數。

例如，試想如下內核:

```
1 kernel void
2 image_filter (int n, int m,
3                 __constant float *filter_weights,
4                 __read_only image2d_t src_image,
5                 __write_only image2d_t dst_image)
6 {
7         ...
8 }
```

在 image_filter 中，引數 n、m、filter_weights、src_image、dst_image 的索引分別為 0、1、2、3、4。

arg_value 所指數據用作索引為 *arg_index* 的引數的值。**clSetKernelArg** 返回後，*arg_value* 所指數據已經被複製，其內存可以由應用重新使用。所有會將 *kernel* 入隊的 API 調用（**clEnqueueNDRangeKernel** 和 **clEnqueueTask**）都會使用這個引數值，直到再次調用 **clSetKernelArg** 將其改變。

如果此引數是一個內存對象（緩衝對象、圖像對象或圖像陣列），*arg_value* 會指向相應的對象。此對象必須是用 *kernel* 所在的上下文創建的。如果引數是一個緩衝對象，*arg_value* 可以是 NULL，也可以指向一個為 NULL 的值，這時，如果內核引數聲明時帶有限定符 *__global* 或 *__constant*，則使用 NULL 值作為引數值。而如果引數聲明時所帶限定符為 *__local*，*arg_value* 必須是 NULL。如果引數型別是 sampler_t，則 *arg_value* 必須指向採樣器對象。

如果引數是一個指針，指向的是全局或不變位址空間中的內建標量或矢量型別、或用戶自定義結構體，則作為引數值的內存對象必須是一個緩衝對象（或 NULL）。如果引數聲明時帶有限定符 *__constant*，則內存對象的大小不能超過 CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE，且這種引數的個數不能超過 CL_DEVICE_MAX_CONSTANT_ARGS。

如果引數型別為 image2d_t，則作為引數值的內存對象必須是 2D 圖像對象。如果引數型別為 image3d_t，則作為引數值的內存對象必須是 3D 圖像對象。如果引數型別為 image1d_t，則作為引數值的內存對象必須是 1D 圖像對象。如果引數型別為 image1d_buffer_t，則作為引數值的內存對象必須是 1D 圖像緩衝對象。如果引數型別為 image1d_array_t，則作為引數值的內存對象必須是 1D 圖像陣列對象。如果引數型別為 image2d_array_t，則作為引數值的內存對象必須是 2D 圖像陣列對象。

對於其他型別的引數，*arg_value* 必須指向作為引數值的實際數據。

arg_size 即引數值的大小。如果引數是內存對象，則為緩衝對象或圖像對象的大小。如果引數在聲明時帶有限定符 *__local*，其值將是為 *__local* 引數所分配緩衝區的大小。如果引數型別是 sampler_t，則 *arg_size* 的值必須是 sizeof(cl_sampler)。對於其他型別的引數，*arg_size* 即為引數型別的大小。

> 對於 **clSetKernelArg** 所使用的作為引數值的對象（如內存對象、採樣器對象）而言，內核對象不會更新其引用計數。用戶不要指望內核對象會為內核引數中的對象執行保留操作。
>
> 實作不能讓 cl_kernel 對象擁有其引數的引用計數，因為沒有為用戶提供任何機制來告訴內核去釋放此所有權。如果內核把持了其引數的所有權，用戶就不可能確切的知道什麼時候可以安全的釋放為其分配的資源，如用 CL_MEM_USE_HOST_PTR 創建的 cl_mem 對象就屬於這種情況。

如果執行成功，**clSetKernelArg** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_KERNEL，如果 *kernel* 無效。
- CL_INVALID_ARG_INDEX，如果 *arg_index* 無效。
- CL_INVALID_ARG_VALUE，如果 *arg_value* 無效。
- CL_INVALID_MEM_OBJECT，如果引數型別是內存對象，而 *arg_value* 無效。
- CL_INVALID_SAMPLER，如果引數型別是 sampler_t，而 *arg_value* 無效。
- CL_INVALID_ARG_SIZE，如果引數不是內存對象，而 *arg_size* 的值與引數大小不一致；或者引數是內存對象，而 arg_size != sizeof(cl_mem)；或者引數聲明時帶有限定符 *__local*，而 *arg_size* 是零；或者引數是採樣器，而 arg_size != sizeof(cl_sampler)。
- CL_INVALID_ARG_VALUE，如果引數是帶有限定符 *read_only* 的圖像，而創建圖像對象 *arg_value* 時在 cl_mem_flags 中設置了 CL_MEM_WRITE。或者引數是帶有限定符 *write_only* 的圖像，而創建圖像對象 *arg_value* 時在 cl_mem_flags 中設置了 CL_MEM_READ。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

5.7.3 內核對象的相關查詢

clGetKernelInfo

```
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)
```

此函式會返回內核對象的相關資訊。

kernel 即所要查詢的內核。

param_name 指定要查詢什麼資訊。對於所支持的資訊類型以及 *param_value* 中返回的資訊，請參見表 *5.15*。

param_value 指向的內存用來存儲查詢結果。如果 *param_value* 是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小。其值必須 >= 表 *5.15*中返回型別的大小。

param_value_size_ret 返回查詢結果的實際大小。如果 *param_value_size_ret* 是 NULL，則忽略。

如果執行成功，**clGetKernelInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_VALUE，如果 *param_name* 無效；或者 *param_value_size* 的值 < 表 *5.15*中返回型別的大小，且 *param_value* 不是 NULL。
- CL_INVALID_KERNEL，如果 *kernel* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clGetKernelWorkGroupInfo

```
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)
```

此函式會返回內核對象針對某個设备的資訊。

kernel 即所要查詢的內核。

表 5.15 **clGetKernelInfo 所支持的** *param_names*

cl_kernel_info	返回型別
CL_KERNEL_FUNCTION_NAME	char[]
返回內核函式的名字。	
CL_KERNEL_NUM_ARGS	cl_uint
返回 *kernel* 的參數個數。	
CL_KERNEL_REFERENCE_COUNT	cl_uint
返回 *kernel* 的引用計數[1]。	
CL_KERNEL_CONTEXT	cl_context
返回 *kernel* 所在的上下文。	
CL_KERNEL_PROGRAM	cl_program
返回 *kernel* 所關聯的程式對象。	
CL_KERNEL_ATTRIBUTES	char[]
返回程式源碼中聲明內核函式時所有通過限定符 *__attribute__* 指定的特性。這些特性包括節 *6.11.2*中所列特性以及實作所支持的其他特性。 所返回的特性就是聲明時 __attribute__((...)) 中的內容，但是會移除兩頭的空格以及內嵌的換行。如果有多個特性，則在所返回的字串中以空格來分隔。	

[1] 在返回的那一刻，此引用計數就已過時。應用中一般不太適用。提供此特性主要是為了檢測內存泄漏。

device 是 *kernel* 所關聯的设备之一。 *kernel* 所關聯的设备即 *kernel* 所在上下文中的设备。如果 *kernel* 所關聯的设备只有一個，則 *device* 可以是 NULL。

param_name 指定要查詢什麼資訊。對於所支持的資訊類型以及 *param_value* 中返回的資訊，請參見表 *5.16*。

param_value 指向的內存用來存儲查詢結果。如果 *param_value* 是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小。其值必須 >= 表 *5.16*中返回型別的大小。

param_value_size_ret 返回查詢結果的實際大小。如果 *param_value_size_ret* 是 NULL，則忽略。

表 5.16 **clGetKernelWorkGroupInfo 所支持的** *param_names*

cl_kernel_work_group_info	返回型別
CL_KERNEL_GLOBAL_WORK_SIZE	size_t[3]
利用此機制，應用可以查詢用來在 *device* 上執行內核的全局索引空間的大小（即 **clEnqueueNDRangeKernel** 的引數 *global_work_size*）。 這要求 *device* 是自定義設備或所執行的內核是內建的，否則 **clGetKernelWorkGroupInfo** 會返回 CL_INVALID_VALUE。	
CL_KERNEL_WORK_GROUP_SIZE	size_t
利用此機制，應用可以查詢用來在 *device* 上執行內核的作業組的大小。 OpenCL 實作可以用內核的資源需求（寄存器的使用情況等）來確定作業組的大小。	
CL_KERNEL_COMPILE_WORK_GROUP_SIZE	size_t[3]
返回限定符 __attribute__((reqd_work_group_size(X, Y, Z))) 所指定的作業組的大小。參見節 *6.7.2*。 如果沒有用上述特性限定符指定作業組的大小，則返回$(0, 0, 0)$。	
CL_KERNEL_LOCAL_MEM_SIZE	cl_ulong
返回內核所使用的局部內存的大小。他包括實作執行內核所需的內存、內核中所聲明的帶有位址限定符 *__local* 的變量、以及為帶有位址限定符 *__local* 的指針引數所分配的內存（其大小由 **clSetKernelArg** 指定）。 對於任一帶有位址限定符 *__local* 的指針引數，如果沒有為其指定內存大小，則假定為 0。	
CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE	size_t
返回所期望的作業組大小的粒度。僅為性能建議。在調用 **clEnqueueNDRangeKernel** 時，如果給引數 *local_work_size* 所指定的作業組大小不是此查詢結果的倍數，並不會導致函式執行失敗，除非此值超過了设备的數目。	
CL_KERNEL_PRIVATE_MEM_SIZE	cl_ulong
內核中每個作業項至少要使用多少私有內存。他包括實作執行內核所需的所有私有內存；包括語言本身所需的私有內存，以及內核中聲明的 *__private* 變量。	

如果執行成功，**clGetKernelWorkGroupInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_DEVICE，如果 *device* 不是 *kernel* 所關聯的設备；或者 *device* 是 NULL，但 *kernel* 所關聯的设备多於一個。
- CL_INVALID_VALUE，如果 *param_name* 無效；或者 *param_value_size* 的值 < 表 5.16中返回型別的大小，且 *param_value* 不是 NULL。
- CL_INVALID_VALUE，如果 *param_name* 是 CL_KERNEL_GLOBAL_WORK_SIZE，且 *device* 不是自定義设备或 *kernel* 不是內建內核。
- CL_INVALID_KERNEL，如果 *kernel* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clGetKernelArgInfo

```
cl_int clGetKernelArgInfo (cl_kernel kernel,
                           cl_uint arg_indx,
                           cl_kernel_arg_info param_name,
                           size_t param_value_size,
                           void *param_value,
                           size_t *param_value_size_ret)
```

此函式會返回內核引數的相關資訊。只有滿足下列條件時，內核引數資訊才可用：

- *kernel* 所關聯的程式對象是用 **clCreateProgramWithSource** 創建的；
- 用 **cl{Build | Compile}Program** 構建程式執行體時，在引數 *options* 中指定了 -cl-kernel-arg-info。

kernel 即所要查詢的內核對象。

arg_indx 即引數的索引。內核引數的索引從最左邊的 0 一直到$n-1$，其中n是內核引數的總數。

param_name 指定要查詢什麼資訊。對於所支持的資訊類型以及 *param_value* 中返回的資訊，請參見表 5.17。

param_value 指向的內存用來存儲查詢結果。如果 *param_value* 是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小。其值必須 >= 表 5.17中返回型別的大小。

param_value_size_ret 返回查詢結果的實際大小。如果 *param_value_size_ret* 是 NULL，則忽略。

表 5.17α　**clGetKernelArgInfo** 所支持的 *param_names*

cl_kernel_arg_info	返回型別
CL_KERNEL_ARG_ADDRESS_QUALIFIER	cl_kernel_arg_address_qualifier
返回參數的位址限定符。所返回的值可以是下列之一： • CL_KERNEL_ARG_ADDRESS_GLOBAL • CL_KERNEL_ARG_ADDRESS_LOCAL • CL_KERNEL_ARG_ADDRESS_CONSTANT • CL_KERNEL_ARG_ADDRESS_PRIVATE 如果沒有指定位址限定符，則返回缺省的位址限定符 CL_KERNEL_ARG_ADDRESS_PRIVATE。	
CL_KERNEL_ARG_ACCESS_QUALIFIER	cl_kernel_arg_access_qualifier
返回參數的訪問限定符。所返回的值可以是下列之一： • CL_KERNEL_ARG_ACCESS_READ_ONLY • CL_KERNEL_ARG_ACCESS_WRITE_ONLY • CL_KERNEL_ARG_ACCESS_READ_WRITE • CL_KERNEL_ARG_ACCESS_NONE 如果參數型別不是圖像，則返回 CL_KERNEL_ARG_ACCESS_NONE。如果參數型別是圖像，則會返回所指定的訪問限定符或缺省的訪問限定符。	
CL_KERNEL_ARG_TYPE_NAME	char[]

表 5.17β **clGetKernelArgInfo** 所支持的 *param_names*

返回參數的型別名，即所聲明的型別名（會移除空白）。如果參數是無符號標量型別（即 unsigned char， unsigned short， unsigned int， unsigned long），則會返回 uchar， ushort， uint 和 ulong。所返回的型別名中不包括任何型別限定符。	
CL_KERNEL_ARG_TYPE_QUALIFIER	cl_kernel_arg_type_qualifier
返回參數的型別限定符。可能是: • CL_KERNEL_ARG_TYPE_CONST • CL_KERNEL_ARG_TYPE_RESTRICT • CL_KERNEL_ARG_TYPE_VOLATILE • 以上三種的組合; 或者 • CL_KERNEL_ARG_TYPE_NONE 注意: 如果參數是指針，且所引用的型別聲明時帶有限定符 *volatile*，則會返回 CL_KERNEL_ARG_TYPE_VOLATILE。例如，如果內核參數聲明為 global int volatile *x，則會返回 CL_KERNEL_ARG_TYPE_VOLATILE; 而對於聲明為 global int *volatile x 的內核參數，則不會返回 CL_KERNEL_ARG_TYPE_VOLATILE。	
CL_KERNEL_ARG_NAME	char[]
返回參數的名字。	

如果執行成功， **clGetKernelArgInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_ARG_INDEX，如果 *arg_indx* 無效。
- CL_INVALID_VALUE， 如果 *param_name* 無效； 或者 *param_value_size* 的值 < 表 5.17中返回型別的大小，且 *param_value* 不是 NULL。
- CL_KERNEL_ARG_INFO_NOT_AVAILABLE，如果沒有可用的引數資訊。
- CL_INVALID_KERNEL，如果 *kernel* 無效。

节 5.8 執行內核

clEnqueueNDRangeKernel

```
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)
```

此函式所入隊的命令可以在设备上執行內核。

command_queue 是一個命令隊列。排隊的內核會在 *command_queue* 所關聯的设备上執行。

kernel 是一個內核對象。 *kernel* 和 *command_queue* 必須位於同一 OpenCL 上下文中。

work_dim 用來指定全局作業項以及作業組中作業項的維數。其值必須大於零並且小於等於 CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS。

global_work_offset 是一個陣列，有 *work_dim* 個元素，且元素為無符號值; 所描述的偏移量可用來計算作業項的全局 ID。如果 *global_work_offset* 是 NULL，則全局 ID起自偏移量$(0, 0, ...0)$。

global_work_size 也是一個陣列，有 *work_dim* 個元素，且元素均為無符號值; 對於將要執行 *kernel* 的全局作業項而言，他在某個維度上的數目由陣列中的相應元素表示，總數為:

$$\prod_{i=0}^{\text{work\_dim}-1} global\_work\_size[i]$$

local_work_size 也是一個陣列，有 *work_dim* 個元素，且元素均為無符號值；對於將要執行 *kernel* 的作業組而言，其中作業項的數目由陣列中的相應元素表示，總數為：

$$\prod_{i=0}^{work\_dim-1} local\_work\_size[i]$$

這個總數必須小於或等於 *表 4.3*中的 CL_DEVICE_MAX_WORK_GROUP_SIZE，而且 *local_work_size*[*i*] 必須小於或等於相應的 CL_DEVICE_MAX_WORK_ITEM_SIZES[*i*]，其中 $0 \leq i \leq work\_dim - 1$。顯式指定的 *local_work_size* 可用來確定怎樣將 *global_work_size* 所指定的全局作業項劃分成多個作業組實體。如果指定了 *local_work_size*，*global_work_size*[*i*] 必須能被相應的 local_work_size[*i*] 整除，其中 $0 \leq i \leq work\_dim - 1$。

也可以在程式源碼中通過限定符 __attribute__((reqd_work_group_size(X, Y, Z))) (參見節 *6.7.2*) 為 *kernel* 指定作業組的大小。這種情況下，*local_work_size* 的值必須與此特性限定符所指定的值相匹配。

local_work_size 也可以是 NULL，這樣的話 OpenCL 實作將自己決定如何將全局作業項劃分成多個作業組實體。

這些作業組實體將在多個計算器件上並行執行，或在單個計算器件上並發執行。

每個作業項都有一個唯一的全局 ID。在內核中，可以通過對 *global_work_size* 和 *global_work_offset* 的運算得到這個全局 ID。另外，每個作業項在作業組中還有一個唯一的局部 ID。在內核中，可以通過對 *local_work_size* 的運算得到這個局部 ID。局部 ID始終起自 $(0, 0, ...0)$。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必須與 *command_queue* 位於同一個上下文中。此函式返回後，即可回收並重新使用 *event_wait_list* 所關聯的內存。

event 會返回一個事件對象，用來標識此拷貝命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。不過可以用 **clEnqueueBarrierWithWaitList** 來代替。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

如果內核成功排隊，**clEnqueueNDRangeKernel** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_PROGRAM_EXECUTABLE，如果 *command_queue* 所關聯的设备沒有對應的程式執行體。
- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_KERNEL，如果 *kernel* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *kernel* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_KERNEL_ARGS，如果還未指定內核參數。
- CL_INVALID_WORK_DIMENSION，如果 *work_dim* 的值無效 (即不在 1 到 3 的範圍內)。
- CL_INVALID_GLOBAL_WORK_SIZE，如果 *global_work_size* 是 NULL；或者 *global_work_size* 中的任意一個是 0 或超過了用來執行內核的设备的 sizeof(size_t) 所給定的範圍。
- CL_INVALID_GLOBAL_OFFSET，如果 i 的有效值中有任意一個使得下列公式的值超過了用來執行內核的设备的 sizeof(size_t) 所給定的範圍：

  $$global\_work\_size[i] + global\_work\_offset[i], 其中\ 0 \leq i \leq work\_dim - 1$$

- CL_INVALID_WORK_GROUP_SIZE，如果指定了 *local_work_size*，並且 *global_work_size* 不能被 *local_work_size* 所整除或者與程式源碼中的特性 __attribute__((reqd_work_group_size(X, Y, Z))) 不匹配。
- CL_INVALID_WORK_GROUP_SIZE，如果指定了 *local_work_size*，並且作業組中作業項的總數（即 $\prod_{i=0}^{\text{work\_dim}-1} local\_work\_size[i]$）大於表 4.3中的 CL_DEVICE_MAX_WORK_GROUP_SIZE。
- CL_INVALID_WORK_GROUP_SIZE，如果 *local_work_size* 是 NULL，並且程式源碼用限定符 __attribute__((reqd_work_group_size(X, Y, Z))) 聲明了作業組的大小。
- CL_INVALID_WORK_ITEM_SIZE，如果 i 的有效值中有任意一個使得下列公式的值超過了對應的 CL_DEVICE_MAX_WORK_ITEM_SIZES[i]:

$$local\_work\_size[i], \text{其中 } 0 \leq i \leq work\_dim - 1$$

- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果給型別為緩衝對象的參數指定了一個子緩衝對象作為其值，並且創建此子緩衝對象時所指定的 *offset* 沒有按 *command_queue* 所關聯设备的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 進行對齊。
- CL_INVALID_IMAGE_SIZE，如果參數是圖像對象，但是圖像的大小（圖像的寬度、高度、指定的或計算出來的行間距和/或面間距）不被 *command_queue* 所關聯的设备所支持。
- CL_IMAGE_FORMAT_NOT_SUPPORTED，如果參數是圖像對象，但是圖像的格式（圖像通道順序和數據型別）不被 *command_queue* 所關聯的设备所支持。
- CL_OUT_OF_RESOURCES，如果由於執行內核所需的資源不足，而無法將 *kernel* 的執行實體入隊。例如，顯式指定了 *local_work_size*，但是沒有足夠的資源（如寄存器、局部內存）從而導致失敗。另一個例子，*kernel*中的只讀圖像參數的數目超過了设备的 CL_DEVICE_MAX_READ_IMAGE_ARGS，或者 *kernel*中的只寫圖像參數的數目超過了设备的 CL_DEVICE_MAX_WRITE_IMAGE_ARGS，或者 *kernel*中所使用的採樣器的數目超過了设备的 CL_DEVICE_MAX_SAMPLERS。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *kernel* 參數中的圖像對象或緩衝對象分配內存時失敗。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clEnqueueTask

```
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)
```

此函式所入隊的命令可以在设备上執行內核。執行內核時使用單個作業項。

command_queue 是一個命令隊列。所排隊的內核會在 *command_queue* 所關聯的设备上執行。

kernel 是一個內核對象。*kernel* 和 *command_queue* 必須位於同一個 OpenCL 上下文中。

event_wait_list 和 *num_events_in_wait_list* 中列出了執行此命令前要等待的事件。如果 *event_wait_list* 是 NULL，則無須等待任何事件，並且 *num_events_in_wait_list* 必須是0。如果 *event_wait_list* 不是 NULL，則其中所有事件都必須是有效的，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點，並且必

須與 *command_queue* 位於同一個上下文中。此函式返回後，即可回收並重新使用 *event_wait_list* 所關聯的內存。

event 會返回一個事件對象，用來標識此拷貝命令，可用來查詢或等待此命令完成。而如果 *event* 是 NULL，就沒辦法查詢此命令的狀態或等待其完成了。不過可以用 **clEnqueueBarrierWithWaitList** 來代替。如果 *event_wait_list* 和 *event* 都不是 NULL，*event* 不能屬於 *event_wait_list*。

clEnqueueTask 相當於在調用 **clEnqueueNDRangeKernel** 時，參數 *work_dim* 為 1，*global_work_offset* 為 NULL，*global_work_size*[0] 為 1，*local_work_size*[0] 為 1。

如果內核成功排隊，**clEnqueueTask** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_PROGRAM_EXECUTABLE，如果 *command_queue* 所關聯的设备沒有對應的程式執行體。
- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_KERNEL，如果 *kernel* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *kernel* 位於不同的上下文中，或者 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_KERNEL_ARGS，如果還未指定內核參數。
- CL_INVALID_WORK_GROUP_SIZE，如果通過限定符 __attribute__((reqd_work_group_size(X, Y, Z))) 為 *kernel* 指定的作業組大小不是 $(1, 1, 1)$。
- CL_MISALIGNED_SUB_BUFFER_OFFSET，如果給型別為緩衝對象的參數指定了一個子緩衝對象作為其值，並且創建此子緩衝對象時所指定的 *offset* 沒有按 *command_queue* 所關聯设备的 CL_DEVICE_MEM_BASE_ADDR_ALIGN 進行對齊。
- CL_INVALID_IMAGE_SIZE，如果參數是圖像對象，但是圖像的大小（圖像的寬度、高度、指定的或計算出來的行間距和/或面間距）不被 *command_queue* 所關聯的设备所支持。
- CL_IMAGE_FORMAT_NOT_SUPPORTED，如果參數是圖像對象，但是圖像的格式（圖像通道順序和數據型別）不被 *command_queue* 所關聯的设备所支持。
- CL_OUT_OF_RESOURCES，如果由於執行內核所需的資源不足，而無法將 *kernel* 的執行實體入隊。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為 *kernel* 參數中的圖像對象或緩衝對象分配內存時失敗。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項：
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clEnqueueNativeKernel

此函式所入隊的命令將執行一個原生的 C/C++ 函式（所謂原生的即不是用 OpenCL 編譯器編譯的）。

command_queue 是一個命令隊列。原生的用戶函式要想在其中執行，此隊列所在设备必須具有 CL_EXEC_NATIVE_KERNEL 的能力（參見表 4.3中的 CL_DEVICE_EXECUTION_CAPABILITIES）。

user_func 指向一個可以被主機調用的用戶函式。

args 指向調用 *user_func* 時所需的參數。

cb_args 即 *args* 所指參數的大小。

args 所指的數據（大小為 *cb_args*）會被拷貝一份，這份拷貝的指針將被傳遞給 *user_func*。之所以要拷貝，是因為 *args* 可能包含有內存對象（cl_mem），而且需要通過一個指向全局內存的指針對其進行修改和取代。一旦 **clEnqueueNativeKernel** 返回，應用就可以重新使用 *args* 所指向的內存區域了。

num_mem_objects 即 *args* 中緩衝對象的數目。

mem_list 指向一組緩衝對象，要求 *num_mem_objects* > 0。其中的緩衝對象可能是 **clCreateBuffer** 所返回的內存對象的句柄（即 cl_mem），也可能是 NULL。

args_mem_loc 指向 *args* 中存儲內存對象句柄（即 cl_mem）的位置。在執行用戶函式之前，會用執行全局內存的指針取代這些句柄。

下列三個參數與 **clEnqueueNDRangeKernel** 中描述的一樣:

- *event_wait_list*,
- *num_events_in_wait_list*, 和
- *event*。

如果用戶函式的執行實體成功排隊，**clEnqueueNativeKernel** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_CONTEXT，如果 *command_queue* 和 *event_wait_list* 中的事件位於不同的上下文中。
- CL_INVALID_VALUE，如果 *user_func* 是 NULL。
- CL_INVALID_VALUE，如果 *args* 是 NULL，並且 *cb_args* > 0；或者 *args* 是一個 NULL 值，並且 *num_mem_objects* > 0。
- CL_INVALID_VALUE，如果 *args* 是 NULL，並且 *cb_args* 是 0。
- CL_INVALID_VALUE，如果 *num_mem_objects* > 0，並且 *mem_list* 或 *args_mem_loc* 是 NULL。
- CL_INVALID_VALUE，如果 *num_mem_objects* = 0，並且 *mem_list* 或 *args_mem_loc* 不是 NULL。
- CL_INVALID_OPERATION，如果 *command_queue* 所關聯的设备不能執行這個原生內核。
- CL_INVALID_MEM_OBJECT，如果 *mem_list* 中的任一內存對象無效或者不是緩衝對象。
- CL_OUT_OF_RESOURCES，如果由於執行內核所需的資源不足，而無法將內核的執行實體入隊。
- CL_MEM_OBJECT_ALLOCATION_FAILURE，如果為內核參數中的緩衝對象分配內存時失敗。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

> 內核參數中只讀圖像的數目不能超過 CL_DEVICE_MAX_READ_IMAGE_ARGS。內核參數中帶有限定符 *read_only* 的 *2D* 圖像陣列記為一個圖像。
> 內核參數中只寫圖像的數目不能超過 CL_DEVICE_MAX_WRITE_IMAGE_ARGS。內核參數中帶有限定符 *write_only* 的 *2D* 圖像陣列記為一個圖像。

节 5.9 事件對象

事件對象可用來指代執行內核的命令

- **clEnqueueNDRangeKernel**
- **clEnqueueTask**
- **clEnqueueNativeKernel**

讀、寫、映射以及拷貝內存對象的命令

- **clEnqueue{Read | Write | Map}Buffer**
- **clEnqueueUnmapMemObject**

- **clEnqueue{Read | Write}BufferRect**
- **clEnqueue{Read | Write | Map}Image**
- **clEnqueueCopy{Buffer | Image}**
- **clEnqueueCopyBufferRect**
- **clEnqueueCopyBufferToImage**
- **clEnqueueCopyImageToBuffer**

以及

- **clEnqueueMarkerWithWaitList**
- **clEnqueueBarrierWithWaitList**

還有用戶事件。

事件對象可用來跟蹤命令的執行狀態。那些會將命令入隊的 API 調用會創建一個新的事件對象，並在引數 *event* 中將其返回；如果將命令插入隊列時出現了錯誤，則不會返回事件對象。

在任意給定的時間點上，所入隊的命令的執行狀態將是下列之一：

- CL_QUEUED，這表示命令已經入隊。這是所有事件的初始狀態（用戶事件除外）。
- CL_SUBMITTED，這是所有用戶事件的初始狀態。對於其他事件，這表明主機已經將命令提交給了设备。
- CL_RUNNING，這表明设备已經開始執行命令。命令的執行狀態要想從 CL_SUBMITTED 變成 CL_RUNNING，他所等待的所有事件必須都已經成功完成，即他們的執行狀態必須是 CL_COMPLETE。
- CL_COMPLETE，這表明命令已經成功完成。
- 錯誤碼，錯誤碼是一個負整數，表明命令異常終止。異常終止的原因有很多，如非法的內存訪問。

> 命令的執行狀態是 CL_COMPLETE 或者負整數都表示已經完成。

如果命令被終止執行，那麼他所關聯的命令隊列、上下文（包括其中的其他命令隊列）就不再可用。這時如果 OpenCL API 調用要使用這個上下文（和其中的命令隊列），則其行為依賴於具體實作。創建上下文時用戶所註冊的回調函式可用來報告相應的錯誤資訊。

clCreateUserEvent

```
cl_event clCreateUserEvent (cl_context context,
                cl_int *errcode_ret)
```

此函式可用來創建用戶自己的事件對象。有了用戶事件，應用就可以讓所入隊的命令先等待用戶事件完成，然後再由设备來執行。

context 是 OpenCL 上下文。

errcode_ret 會返回相應的錯誤碼。如果 *errcode_ret* 是 NULL，則不會返回錯誤碼。

如果成功創建了用戶事件對象，**clCreateUserEvent** 會將其返回，並將 *errcode_ret* 置為 CL_SUCCESS。否則，返回 NULL，並將 *errcode_ret* 置為下列錯誤碼之一：

- CL_INVALID_CONTEXT，如果 *context* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

用戶事件對象在創建後，其執行狀態缺省為 CL_SUBMITTED。

clSetUserEventStatus

```
cl_int clSetUserEventStatus (cl_event event,
                cl_int execution_status)
```

此函式可用來設置用戶事件對象的執行狀態。

event 即用 **clCreateUserEvent** 創建的用戶事件對象。

execution_status 即將要設置的新的執行狀態，可以是 CL_COMPLETE，或者一個用來表示錯誤的負整數。負整數會導致所有已經入隊、並且等待此事件的命令被終止。要想改變 *event* 的執行狀態，**clSetUserEventStatus** 只能被調用一次。

如果執行成功，**clSetUserEventStatus** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_EVENT，如果 *event* 無效。
- CL_INVALID_VALUE，如果 *execution_status* 既不是 CL_COMPLETE，也不是負整數。
- CL_INVALID_OPERATION，如果之前已經調用 **clSetUserEventStatus** 改變過 *event* 的執行狀態。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

使用 **clEnqueue***** 時，如果引數 *event_wait_list* 中有用戶事件，那麼對於所入隊的命令而言，在調用會釋放 *OpenCL* 對象（事件對象）的 *OpenCL API* 之前，必須保證已經用 **clSetUserEventStatus** 設置過這些用戶事件的狀態；否則其行為未定義。

例如，下列代碼序列會導致 **clReleaseMemObject** 的未定義行為：

```
1 ev1 = clCreateUserEvent(ctx, NULL);
2 clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ...,
3                       1, &ev1, NULL);
4 clEnqueueWriteBuffer(cq, buf2, CL_FALSE,...);
5 clReleaseMemObject(buf2);
6 clSetUserEventStatus(ev1, CL_COMPLETE);
```

而下列代碼序列則可以正確工作：

```
1 ev1 = clCreateUserEvent(ctx, NULL);
2 clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ...,
3                       1, &ev1, NULL);
4 clEnqueueWriteBuffer(cq, buf2, CL_FALSE,...);
5 clSetUserEventStatus(ev1, CL_COMPLETE);
6 clReleaseMemObject(buf2);
```

clWaitForEvents

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

此函式會使主機線程等待 *event_list* 中的事件對象所標識的命令完成。對於一個命令而言，如果其執行狀態是 CL_COMPLETE 或負整數，則任務已經完成了。*event_list* 中的事件充當同步點。

如果 *event_list* 中的所有事件的執行狀態都是 CL_COMPLETE，則 **clWaitForEvents** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_VALUE，如果 *num_events* 是零或者 *event_list* 是 NULL。
- CL_INVALID_CONTEXT，如果 *event_list* 中的事件分屬不同的上下文。
- CL_INVALID_EVENT，如果 *event_list* 中的事件對象無效。
- CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST，如果 *event_list* 中的任一事件的執行狀態是負整數。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clGetEventInfo

```
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)
```

此函式會返回事件對象的相關資訊。

event 即所要查詢的事件對象。

param_name 指定要查詢什麼資訊。對於所支持的資訊類型以及 *param_value* 中返回的資訊，請參見表 *5.18*。

param_value 指向的內存用來存儲查詢結果。如果 *param_value* 是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小。其值必須 >= 表 *5.18*中返回型別的大小。

param_value_size_ret 返回查詢結果的實際大小。如果 *param_value_size_ret* 是 NULL，則忽略。

表 5.18 **clGetEventInfo 所支持的** *param_names*

cl_event_info	返回型別
CL_EVENT_COMMAND_QUEUE	cl_command_queue
返回 *event* 所關聯的命令隊列。對於用戶事件對象，會返回 NULL。	
CL_EVENT_CONTEXT	cl_context
返回 *event* 所關聯的上下文。	
CL_EVENT_COMMAND_TYPE	cl_command_type
返回 *event* 所關聯的命令。可以是下列之一: • 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_TO_BUFFER • CL_COMMAND_MAP_BUFFER • CL_COMMAND_MAP_IMAGE • CL_COMMAND_UNMAP_MEM_OBJECT • CL_COMMAND_MARKER • CL_COMMAND_ACQUIRE_GL_OBJECTS • CL_COMMAND_RELEASE_GL_OBJECTS • CL_COMMAND_READ_BUFFER_RECT • CL_COMMAND_WRITE_BUFFER_RECT • CL_COMMAND_COPY_BUFFER_RECT • CL_COMMAND_USER • CL_COMMAND_BARRIER • CL_COMMAND_MIGRATE_MEM_OBJECTS • CL_COMMAND_FILL_BUFFER • CL_COMMAND_FILL_IMAGE	
CL_EVENT_COMMAND_EXECUTION_STATUS	cl_int
返回 *event* 所標識的命令的執行狀態[1]。有效值為: • CL_QUEUED (命令已經入隊); • CL_SUBMITTED (主機已經將所入隊的命令提交給了命令隊列所關聯的设备); • CL_RUNNING (设备正在執行這個命令); • CL_COMPLETED (命令已經完成); 或 • 錯誤碼，一個負整數 (命令異常終止——可能由非法內存訪問所導致)。與平台或運行時 API 調用所返回的值或 *errcode_ret* 的值使用同一套錯誤碼。	
CL_EVENT_REFERENCE_COUNT	cl_uint
返回 *event* 的引用計數[2]。	

[1] 錯誤碼的值是負的，事件狀態的值是正的。事件狀態的值這樣變化：從第一個或初始狀態，即最大值（CL_QUEUED），一直到最後一個或完成的狀態，即最小值（CL_COMPLETE 或負整數）。CL_COMPLETE 跟 CL_SUCCESS 一樣。

[2] 在返回的那一刻，此引用計數就已過時。應用中一般不太適用。提供此特性主要是為了檢測內存泄漏。

可以使用 **clGetEventInfo** 來確定 *event* 所標識的命令是否執行完畢（即 CL_EVENT_COMMAND_EXECUTION_STATUS 返回 CL_COMPLETE），但這不是同步點。*event* 所關聯的命令可能會對內存對象做一些修改，不保證這些修改對其他已經入隊的命令是可見的。

如果執行成功，**clGetEventInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_VALUE，如果 *param_name* 無效，或者 *param_value_size* < 表 *5.18*中返回型別的大小，且 *param_value* 不是 NULL。
- CL_INVALID_VALUE，如果對於 *event* 而言，所要查詢的資訊還不能被查詢。
- CL_INVALID_EVENT，如果 *event* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clSetEventCallback

```
cl_int clSetEventCallback (
        cl_event event,
        cl_int command_exec_callback_type,
        void (CL_CALLBACK *pfn_event_notify)(
                cl_event event,
                cl_int event_command_exec_status,
                void *user_data),
        void *user_data)
```

此函式可以為命令的某個執行狀態註冊一個用戶回調函式。當 *event* 所關聯的命令的執行狀態變成或越過 *command_exec_status* 時，所註冊的回調函式就會被調用。

調用 **clSetEventCallback** 會將用戶回調函式註冊到 *event* 的回調棧上。而對於所註冊用戶回調函式的調用順序，則沒有定義。

event 是一個事件對象。

command_exec_callback_type 指定命令的執行狀態，回調就註冊到此狀態上。能註冊回調的狀態為：CL_SUBMITTED、CL_RUNNING 或 CL_COMPLETE[8]。不保證按執行狀態的變化順序來調用相應的回調函式。進而，需要注意的是，調用回調時，事件的狀態可能不是 CL_COMPLETE，但這絕不意味着 OpenCL 規範所定義的內存模型或執行模型發生了變化。例如，除非事件的狀態是 CL_COMPLETE，否則不能假設相應的內存遷移已經完成。

pfn_event_notify 即應用所註冊的事件回調函式。此函式可能會被 OpenCL 實作異步調用。應用要保證此函式是線程安全的。此函式的參數為：

- *event* 即此函式所關注的事件。
- *event_command_exec_status* 即此函式所關注的命令執行狀態。關於命令的執行狀態請參見表 *5.18*。如果是由於命令異常終止而調用的此函式，那麼傳給 *event_command_exec_status* 的將是相應的錯誤碼。
- *user_data* 指向用戶提供的數據。

user_data 將在調用 *pfn_event_notify* 時作為其引數 *user_data* 傳入。*user_data* 可以是 NULL。

註冊到一個事件對象上所有回調都必須被調用，而且是在銷毀事件對象之前被調用。回調必須儘快返回。如果在回調中調用了一些代價高昂的系統例程，如創建上下文或命令隊列的 OpenCL API，或者下列會阻塞的 OpenCL 操作，則其行為是未定義的。

- **clFinish**
- **clWaitForEvents**
- 對下列 API 的阻塞式調用：
  - **clEnqueueReadBuffer**
  - **clEnqueueReadBufferRect**
  - **clEnqueueWriteBuffer**

[8] 如果 *command_exec_callback* 是 CL_COMPLETE，當命令執行成功或者異常終止時都會調用所註冊的回調函式。

  - **clEnqueueWriteBufferRect**
- 對下列 API 的阻塞式調用:
  - **clEnqueueReadImage**
  - **clEnqueueWriteImage**
- 對下列 API 的阻塞式調用:
  - **clEnqueueMapBuffer**
  - **clEnqueueMapImage**
- 對下列 API 的阻塞式調用:
  - **clBuildProgram**
  - **clCompileProgram**
  - **clLinkProgram**

如果應用想等待上述例程的完成，請使用其非阻塞的形式，並設置一個回調來完成剩餘的工作。注意，如果回調（或者其他代碼）入隊了命令，可能在刷新隊列時才開始執行這些命令。在標準用法中，阻塞的入隊調用通過顯式的刷新隊列來達到此效果。由於回調中不運行有阻塞的調用，那些會入隊命令的回調應該在返回前調用 **clFlush** 或者將 **clFlush** 安排在另一個線程中晚些時候再調用。

如果執行成功，**clSetEventCallback** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_EVENT，如果 *event* 無效。
- CL_INVALID_VALUE，如果滿足下列任一條件:
  - *pfn_event_notify* 是 NULL;
  - 或者 *command_exec_callback_type* 不是 CL_COMPLETE。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clRetainEvent

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

此函式會使 *memobj* 的引用計數增一。那些會返回事件的 OpenCL 命令會實施隱式的保留。

如果執行成功，**clRetainEvent** 會返回 CL_SUCCESS。否則，返回下列錯誤的一種:

- CL_INVALID_EVENT，如果 *event* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clReleaseEvent

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

此函式會使 *event* 的引用計數減一。

如果執行成功，**clReleaseEvent** 會返回 CL_SUCCESS。否則，返回下列錯誤的一種:

- CL_INVALID_EVENT，如果 *event* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

一旦 *event* 的引用計數變成零，並且他所標識的命令執行完畢（或終止），同時沒有正在等待他完成的命令，則此事件就會被刪除。

> 對於用 **clCreateUserEvent** 創建的事件而言，如果還未將其狀態置為 CL_COMPLETE 或者一個錯誤碼，這時要釋放他的最後一個引用計數，開發人員就要小心了。如果用戶事件作為引數 *event_wait_list* 傳給了 **clEnqueue***，或者另一個主機線程正在用 **clWaitForEvents** 等待他，那麼即使用戶釋放了此對象，這些命令和主機線程也會等待其狀態變為 CL_COMPLETE 或錯誤。這種情況下，如果開發人員釋放了用戶事件的最後一個引用計數，理論上他就不能改變事件的狀態以解除其他部分的阻塞。結果就是正在等待的任務會永遠等下去，相關的事件、cl_mem 對象、命令隊列和上下文多半就泄漏了。順序執行的命令隊列碰到這種死鎖就會停止做任何事情。

节 5.10 標註、屏障以及對事件的等待

clEnqueueMarkerWithWaitList

```
cl_int clEnqueueMarkerWithWaitList(
                          cl_command_queue command_queue,
                          cl_uint num_events_in_wait_list,
                          const cl_event *event_wait_list,
                          cl_event *event)
```

此函式會將一個標註命令入隊，此命令會等待 *event_wait_list* 中的事件完成，如果 *event_wait_list* 為空，則等待 *command_queue* 中之前入隊的命令完成。此命令會返回一個可以等待的事件，即可以等在這個事件上，來確定 *event_wait_list* 中的事件完成，或者 *command_queue* 中之前入隊的命令完成。

command_queue 是一個命令隊列。

event_wait_list 和 *num_events_in_wait_list* 即執行這個命令前需要完成的事件。

如果 *event_wait_list* 是 NULL，*num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，*event_wait_list* 中的事件必須有效，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點。*event_wait_list* 中的事件與 *command_queue* 必須位於同一上下文中。當此函式返回後，就可以重新使用或者釋放 *event_wait_list* 所關聯的內存了。

如果 *event_wait_list* 是 NULL，則此命令會等到 *command_queue* 中所有之前入隊的命令完成。

event 會返回一個事件對象，可用來標識此命令。如果 *event* 是 NULL，那麼應用就無法查詢此命令的狀態，或者等待此命令的完成。如果 *event_wait_list* 和 *event* 都不是 NULL，參數 *event* 不能是 *event_wait_list* 中的事件。

如果執行成功，**clEnqueueMarkerWithWaitList** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

clEnqueueBarrierWithWaitList

```
cl_int clEnqueueBarrierWithWaitList(
                          cl_command_queue command_queue,
                          cl_uint num_events_in_wait_list,
                          const cl_event *event_wait_list,
                          cl_event *event)
```

此函式會將一個屏障命令入隊，此命令會等待 *event_wait_list* 中的事件完成，如果 *event_wait_list* 為空，則等待 *command_queue* 中之前入隊的命令完成。此命令會阻塞命令的執行，即，所遇之後入隊的命令都得等到他完成後才能執行。此命令會返回一個可以等待的事件，即可以等在這個事件上，來確定 *event_wait_list* 中的事件完成，或者 *command_queue* 中之前入隊的命令完成。

command_queue 是一個命令隊列。

event_wait_list 和 *num_events_in_wait_list* 即執行這個命令前需要完成的事件。

如果 *event_wait_list* 是 NULL，*num_events_in_wait_list* 必須是 0。如果 *event_wait_list* 不是 NULL，*event_wait_list* 中的事件必須有效，並且 *num_events_in_wait_list* 必須大於 0。*event_wait_list* 中的事件充當同步點。*event_wait_list* 中的事件與 *command_queue* 必須位於同一上下文中。當此函式返回後，就可以重新使用或者釋放 *event_wait_list* 所關聯的內存了。

如果 *event_wait_list* 是 NULL，則此命令會等到 *command_queue* 中所有之前入隊的命令完成。

event 會返回一個事件對象，可用來標識此命令。如果 *event* 是 NULL，那麼應用就無法查詢此命令的狀態，或者等待此命令的完成。如果 *event_wait_list* 和 *event* 都不是 NULL，參數 *event* 不能是 *event_wait_list* 中的事件。

如果執行成功，**clEnqueueBarrierWithWaitList** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一:

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_INVALID_EVENT_WAIT_LIST，如果滿足下列條件中的任一項:
  - *event_wait_list* 是 NULL，但 *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* 中有無效的事件。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

节 5.11 內核和內存對象命令的亂序執行

提交給命令隊列的 OpenCL 函式按被調用的順序入隊，但執行順序卻是可配置的，既可以順序執行，也可以亂序執行。**clCreateCommandQueue** 的參數 *properties* 可用來指定執行順序。

如果沒有為命令隊列設置屬性 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE，則其中的命令順序執行。例如，如果應用先調用 **clEnqueueNDRangeKernel** 執行內核 A，然後調用 **clEnqueueNDRangeKernel** 執行內核 B，則應用可以假定 A 執行完後 B 才開始執行。如果內核 A 所輸出的內存對象是內核 B 的輸入，那麼就通過執行內核 A 所產生的內存對象而言，內核 B 就可以看到其中正確的數據。如果為命令隊列設置了屬性 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE，則不保證內核 B 開始執行前，內核 A 一定執行完畢。

應用可以通過設置命令隊列的屬性 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE，將其中的命令配置成亂序執行。創建命令隊列時就可以設置此屬性。在亂序執行模式下，不保證命令按入隊的順序執行完畢。由於不保證內核會順序執行（這裡的順序是指調用 **clEnqueueNDRangeKernel** 的順序），因此很可能先入隊的內核 A（用事件 A 來標識）開始執行和/或執行完畢要比後入隊的內核 B 晚。要想保證內核的執行順序，可以等待某個事件（這裡就是事件 A）。對事件 A 的等待可以通過為內核 B 調用 **clEnqueueNDRangeKernel** 時設置其參數 *event_wait_list* 來實現。

另外，OpenCL 還提供下列命令:

- 標註，由 **clEnqueueMarkerWithWaitList** 入隊，用於等待一些事件;
- 屏障，由 **clEnqueueBarrierWithWaitList** 入隊;

等待事件的命令可以保證事件所標識的命令完成後，下一批命令才開始執行。而屏障命令則保證命令隊列中所有之前入隊的命令都執行完畢後，下一批命令才開始執行。

類似的，如果設置了屬性 CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE，那麼在 **clEnqueueNDRangeKernel**、**clEnqueueTask** 或 **clEnqueueNativeKernel** 之後入隊的一些讀、寫、拷貝或映射內存對象的命令可能不會等待內核執行完畢。要想保證命令的正確順序，可以使用標註命令等待 **clEnqueueNDRangeKernel**、**clEnqueueTask** 或 **clEnqueueNativeKernel** 所返回的事件對象，或者使用屏障命令，這樣只有在內核執行完畢後，才會開始讀寫內存對象。

节 5.12 對內存對象以及內核的評測

一些 OpenCL 函式可以作為命令插入到命令隊列中，所入隊的命令是通過唯一的事件對象來標識的。本節將描述對這些函式的評測（profiling）。這些函式[9]是:

- **clEnqueue{Read|Write|Map}Buffer**
- **clEnqueue{Read|Write}BufferRect**
- **clEnqueue{Read|Write|Map}Image**

[9] 《OpenCL 1.2 擴展規範》的節 *9.6.6* 中定義的 **clEnqueueAcquireGLObjects** 和 **clEnqueueReleaseGLObjects** 也包含在內。

- **clEnqueueUnmapMemObject**
- **clEnqueueCopyBuffer**
- **clEnqueueCopyBufferRect**
- **clEnqueueCopyImage**
- **clEnqueueCopyImageToBuffer**
- **clEnqueueCopyBufferToImage**
- **clEnqueueNDRangeKernel**
- **clEnqueueTask**
- **clEnqueueNativeKernel**。

clGetEventProfilingInfo

```
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)
```

如果使能了評測，則此函式會返回 *event* 所指代命令的評測資訊。

event 即事件對象。

param_name 指定所要查詢的評測數據。對於所支持的資訊類型以及 *param_value* 中返回的資訊，請參見表 *5.19*。

param_value 指向的內存用來存儲查詢結果。如果 *param_value* 是 NULL，則忽略。

param_value_size 即 *param_value* 所指內存塊的大小。其值必須 >= 表 *5.19*中返回型別的大小。

param_value_size_ret 返回查詢結果的實際大小。如果 *param_value_size_ret* 是 NULL，則忽略。

表 5.19 **clGetEventProfilingInfo 所支持的** *param_names*

cl_profiling_info	**返回型別**
CL_PROFILING_COMMAND_QUEUED	cl_ulong
一個 64 位值，描述的是當 *event* 所標識的命令入隊時，設備上時間計數器的值。單位：納秒。	
CL_PROFILING_COMMAND_SUBMIT	cl_ulong
一個 64 位值，描述的是當 *event* 所標識的命令提交給命令隊列所在設備時，設備上時間計數器的值。單位：納秒。	
CL_PROFILING_COMMAND_START	cl_ulong
一個 64 位值，描述的是當 *event* 所標識的命令在設備開始執行時，設備上時間計數器的值。單位：納秒。	
CL_PROFILING_COMMAND_QUEUED	cl_ulong
一個 64 位值，描述的是當 *event* 所標識的命令在設備上執行完畢時，設備上時間計數器的值。單位：納秒。	

所返回的無符號 64 位值可用來衡量 OpenCL 命令所消耗的時間，單位：納秒。

在设备的頻率和電源的狀態發生變化時，要求 OpenCL 设备能夠正確地跟蹤時間。CL_DEVICE_PROFILING_TIMER_RESOLUTION 指定了時鐘的精度，即，定時器增一時所經過的納秒數。

如果執行成功，且記錄了評測資訊，則 **clGetEventProfilingInfo** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_PROFILING_INFO_NOT_AVAILABLE，如果命令隊列沒有設置 CL_QUEUE_PROFILING_ENABLE；或者 *event* 所標識命令的執行狀態不是 CL_COMPLETE；或者 *event* 是用戶事件對象。
- CL_INVALID_VALUE，如果 *param_name* 無效；或者 *param_value_size* 的值 < 表 *5.19*中返回型別的大小，且 *param_value* 不是 NULL。
- CL_INVALID_EVENT，如果 *event* 無效。

- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

节 5.13 刷新和完結

clFlush

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

此函式會將 *command_queue* 中所有之前入隊的 OpenCL 命令都提交給 *command_queue* 所在设备去執行。 **clFlush** 返回時僅保證命令已經提交，不保證執行完畢。

如果執行成功，**clFlush** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

命令隊列中所有阻塞的命令以及 **clReleaseCommandQueue** 都會對命令隊列實施隱式的刷新（flush）。這些阻塞的命令包括：

- **clEnqueueReadBuffer**、**clEnqueueReadBufferRect**、**clEnqueueReadImage**，其中參數 *blocking_read* 設置成 CL_TRUE。
- **clEnqueueWriteBuffer**、**clEnqueueWriteBufferRect**、**clEnqueueWriteImage**，其中參數 *blocking_write* 設置成 CL_TRUE。
- **clEnqueueMapBuffer**、**clEnqueueMapImage**，其中參數 *blocking_map* 設置成 CL_TRUE。
- **clWaitForEvents**。

如果命令隊列中有一個命令 A，而另一個命令隊列中有命令 B 在等待用來指代命令 A 的事件對象，那麼應用必須對命令 A 所在的命令隊列調用 **clFlush**，或者任何會實施隱式刷新的的阻塞命令。

clFinish

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

此函式是阻塞的，等到 *command_queue* 中所有之前入隊的命令都被提交給相應的设备並執行完畢後才會返回。 **clFinish** 也是一個同步點。

如果執行成功，**clFinish** 會返回 CL_SUCCESS。否則，返回下列錯誤碼之一：

- CL_INVALID_COMMAND_QUEUE，如果 *command_queue* 無效。
- CL_OUT_OF_RESOURCES，如果在為设备上的 OpenCL 實作分配資源時失敗。
- CL_OUT_OF_HOST_MEMORY，如果在為主機上的 OpenCL 實作分配資源時失敗。

第 6 章
OpenCL C 編程語言

本章將描述 OpenCL C 編程語言，用他可以創建運行在 OpenCL 设备上的內核。 OpenCL C 編程語言（簡稱 OpenCL C）基於 ISO/IEC 9899:1999 C 語言規範（又名 C99 規範），同時又做了一些擴展和限制。關於語言的語法，其詳細描述請參考 ISO/IEC 9899:1999 規範。本章僅描述 Open CL C 對他所做的修改和限制。

节 6.1 所支持的數據型別

所支持的數據型別如下所示。

6.1.1 內建標量數據型別

表 *6.1*中列出了內建的標量數據型別。

表 6.1 內 建 標 量 數 據 型 別

型別	描述
`bool`[1]	一種條件數據型別，值為 `true` 或 `false`。其中 `true` 展開後為整形常量 1；而 `false` 展開後為整形常量 0。
`char`	帶符號 8 位整數，為二的補碼。
`unsigned char` `uchar`	無符號 8 位整數。
`short`	帶符號 16 位整數，為二的補碼。
`unsigned short` `ushort`	無符號 16 位整數。
`int`	帶符號 32 位整數，為二的補碼。
`unsigned int` `uint`	無符號 32 位整數。
`long`	帶符號 64 位整數，為二的補碼。
`unsigned long` `ulong`	無符號 64 位整數。
`float`	32 位浮點數。必須符合 IEEE 754 中的單精度存儲格式。
`double`[2]	64 位浮點數。必須符合 IEEE 754 中的雙精度存儲格式。
`half`	16 位浮點數。必須符合 IEEE 754-2008 中的半精度存儲格式。
`size_t`	無符號整數，算子 `sizeof` 的結果。如果 `CL_DEVICE_ADDRESS_BITS`（參見表 *4.3*）是 32 位，則此型別為 32 位無符號整數；如果 `CL_DEVICE_ADDRESS_BITS` 是 64 位，則此型別為 64 位無符號整數。
`ptrdiff_t`	帶符號整形，兩個指針相減的結果。如果 `CL_DEVICE_ADDRESS_BITS`（參見表 *4.3*）是 32 位，則此型別為 32 位帶符號整數；如果 `CL_DEVICE_ADDRESS_BITS` 是 64 位，則此型別為 64 位帶符號整數。
`intptr_t`	帶符號整形，任意指向 `void` 的有效指針都能轉換為此型別，然後還可以轉換回指向 `void` 的指針，其結果與原始指針相同。如果 `CL_DEVICE_ADDRESS_BITS`（參見表 *4.3*）是 32 位，則此型別為 32 位帶符號整數；如果 `CL_DEVICE_ADDRESS_BITS` 是 64 位，則此型別為 64 位帶符號整數。
`uintptr_t`	無符號整形，任意指向 `void` 的有效指針都能轉換為此型別，然後還可以轉換回指向 `void` 的指針，其結果與原始指針相同。如果 `CL_DEVICE_ADDRESS_BITS`（參見表 *4.3*）是 32 位，則此型別為 32 位無符號整數；如果 `CL_DEVICE_ADDRESS_BITS` 是 64 位，則此型別為 64 位無符號整數。
`void`	此型別不包含任何值；他是一種不完全型別，不能被補全。

[1] 任意標量值轉換為 `bool` 時，如果原始值等於 0，則結果為 0；否則，結果為 1。

[2] `double` 是可選型別，只有設備的 `CL_DEVICE_DOUBLE_FP_CONFIG`（參見表 *4.3*）不是零時才需要支持。

在 OpenCL API（以及頭檔）中，大多數內建標量型別都被聲明為其他型別，以更好的為應用所用。表 *6.1*中列出了 OpenCL C 編程語言中的內建標量數據型別與應用所用型別間的對應關係。

表 6.2α 內建標量數據型別與應用所用型別的對應關係

OpenCL 語言中的型別	應用所用 API 中的型別
`bool`	`n/a`

表 6.2β 內建標量數據型別與應用所用型別的對應關係

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
double	cl_double
half	cl_half
size_t	n/a
ptrdiff_t	n/a
intptr_t	n/a
uintptr_t	n/a
void	n/a

6.1.1.1 數據型別 half

數據型別 half 必須符合 IEEE 754-2008。型別為 half 的數含有 1 個符號位，5 個指數位以及 10 個尾數位。符號、指數、尾數的含義與 IEEE 754 浮點數類似。指數偏置值為 15。數據型別 half 必須能夠表示有限規格化數，去規格化數，無限以及 NaN。不能將 half 型別的去規格化數 (可能是用 vstore_half 將 float 轉換成 half 時生成的，也可能是用 vload_half 將 half 轉換成 float 時生成的) 刷成 0。從 float 到 half 的轉換會將尾數捨入成 11 位精度。從 half 到 float 的轉換是無損的；所有 half 數都可精確地表示成 float 值。

數據型別 half 只能用來聲明指向含有 half 值的緩衝區的指針。下面是幾個例子。

```
1  void bar (__global half *p)
2  {
3          ....
4  }
5  __kernel void foo (__global half *pg, __local half *pl)
6  {
7          __global half *ptr;
8          int offset;
9
10         ptr = pg + offset;
11         bar(ptr);
12 }
```

下面的例子是對型別 half 的不當應用:

```
1  half a;
2  half b[100];
3
4  half *p;
5  a = *p;          <- not allowed. must use vload_half function
```

函式 **vload_half**、**vload_halfn**、**vloada_halfn** 和 **vstore_half**、**vstore_halfn**、**vstorea_halfn** 可分別裝載和存儲 half 指針，參見節 *6.12.7*。裝載函式可從內存中讀取標量或矢量 half 值並將其轉換成 float 值。而存儲函式則將標量或矢量 float 值作為輸入，並 (以恰當的捨入模式) 將其轉換成 half 標量或矢量值後寫入內存中。

6.1.2 內建矢量數據型別[10]

支持的矢量數據型別有：char、unsigned char、short、unsigned short、integer、unsigned integer、long、unsigned long、float。矢量數據型別是通過在型別名（即 char、uchar、short、ushort、int、uint、float、long、ulong）後面跟一個常值 n 來定義的（其中 n 表示矢量元素的數目）。對於所有矢量數據型別而言，這個 n 可以是 2、3、4、8 和 16。

表 *6.3*中列出了內建的矢量數據型別。

表 6.3 內建矢量數據型別

型別	描述
char*n*	帶符號 8 位整數矢量，為二的補碼。
uchar*n*	無符號 8 位整數矢量。
short*n*	帶符號 16 位整數矢量，為二的補碼。
ushort*n*	無符號 16 位整數矢量。
int*n*	帶符號 32 位整數矢量，為二的補碼。
uint*n*	無符號 32 位整數矢量。
long*n*	帶符號 64 位整數矢量，為二的補碼。
ulong*n*	無符號 64 位整數矢量。
float*n*	32 位浮點數矢量。
double*n*[1]	64 位浮點數矢量。

[1] double*n* 是可選型別，只有設備的 CL_DEVICE_DOUBLE_FP_CONFIG（參見表 *4.3*）不是零時才需要支持。

在 OpenCL API（以及頭檔）中，大多數內建矢量型別都被聲明為其他型別，以更好的為應用所用。下表列出了 OpenCL C 編程語言中的內建矢量數據型別與應用所用型別間的對應關係。

OpenCL 語言中的型別	應用所用 API 中的型別
char*n*	cl_char*n*
uchar*n*	cl_uchar*n*
short*n*	cl_short*n*
ushort*n*	cl_ushort*n*
int*n*	cl_int*n*
uint*n*	cl_uint*n*
long*n*	cl_long*n*
ulong*n*	cl_ulong*n*
float*n*	cl_float*n*
double*n*	cl_double*n*

6.1.3 其他內建數據型別

表 *6.4*中列出了 OpenCL 所支持的其他內建數據型別。

只有當設備支持圖像時（即 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，參見表 *4.3*），才會定義型別 image2d_t、image3d_t、image2d_array_t、image1d_t、image1d_buffer_t、image1d_array_t、sampler_t。

C99 的衍生型別（陣列、結構體、聯合體、函式以及指針）也在支持之列，不過必須是由節 *6.1.1*、節 *6.1.2* 和節 *6.1.3*中所描述的內建數據型別所構造的，同時有節 *6.9*中所描述的限制。

6.1.4 保留的數據型別

表 *6.5* 中所列的數據型別名都是保留的，應用不能將其作為型別名使用。對於表 *6.3* 中所列的矢量數據型別名而言，當 n 是除 2、3、4、8 和 16 之外的其他值時，也是保留的。

[10] 對於這些內建的矢量數據型別，即使下層的計算設備不支持，OpenCL 實作也要支持。設備編譯器需要將這些型別翻譯成恰當的指令，以計算設備原生支持的內建型別。附錄 C 描述了矢量型別所含組件在內存中的順序。

表 6.5 保留的數據型別

型別	描述
`bool`*n*	布爾矢量。
`half`*n*	16 位浮點數矢量。
`quad` `quad`*n*	128 位浮點標量和矢量。
`complex half` `complex half`*n*	16 位浮點複數標量和矢量。
`imaginary half` `imaginary half`*n*	16 位浮點虛數標量和矢量。
`complex float` `complex float`*n*	32 位浮點複數標量和矢量。
`imaginary float` `imaginary float`*n*	32 位浮點虛數標量和矢量。
`complex double` `complex double`*n*	64 位浮點複數標量和矢量。
`imaginary double` `imaginary double`*n*	64 位浮點虛數標量和矢量。
`complex quad` `complex quad`*n*	128 位浮點複數標量和矢量。
`imaginary quad` `imaginary quad`*n*	128 位浮點虛數標量和矢量。
`float`*n*`x`*m*	單精度浮點數的 $n \times m$ 矩陣，以列優先存儲。
`double`*n*`x`*m*	雙精度浮點數的 $n \times m$ 矩陣，以列優先存儲。
`long double` `long double`*n*	浮點標量和矢量。其精度和範圍至少為 `double`，但不超過 `quad`。
`long long` `long long`*n*	128 位帶符號整形標量和矢量。

`unsigned long long`
`ulong long`
`ulong long`*n*
	128 位無符號整形標量和矢量。

6.1.5 型別齊位

內存中所聲明的數據項會按起型別的字節數進行對齊。例如，型別為 `float4` 的變量會對齊到 16 字節邊界；而 `char2` 則會對齊到 2 字節邊界。

對於具備 3 個組件的矢量數據型別，其大小為 $4 \times sizeof(component)$。這意味着此型別會按 $4 \times sizeof(component)$ 進行對齊。內建函式 **`vload3`** 和 **`vstore3`** 可用來讀寫這種數據。

如果內建數據型別的大小不是二的冪，則按相鄰較大的二的冪進行對齊。此規則僅對內建型別有效，不會作用於結構體或聯合體上。

OpenCL 編譯器負責數據項的對齊。如果 *`__kernel`* 函式的參數聲明為某種數據型別的指針，那麼 OpenCL 編譯器就可以假定這個指針已經按照相應數據型別的要求進行了對齊。如果裝載或存儲時沒有對齊，則其行為是未定義的，節 *6.12.7*中定義的函式 **`vloadn`**、**`vload_halfn`**、**`vstoren`** 和 **`vstore_halfn`** 例外。矢量裝載函式可以從已經按矢量元素型別對齊的位址中讀取矢量數據。矢量存儲函式可以將矢量數據寫入已經按矢量元素型別對齊的位址中。

6.1.6 常值矢量

可使用常值矢量由一組標量、矢量或其混合體來創建矢量。常值矢量可用來對矢量進行初始化，也可以用作主算式。常值矢量不能用作左值 (L-value)。

常值矢量的書寫形式：一個帶括號的矢量型別，緊跟一組帶括號參數，參數以逗號分隔。常值矢量像重載的函式一樣參與運算。這個函式的參數型別就是最終矢量中的元素型別，參數個數就是最終矢量中的元素個數。另外，還有一種形式只帶有一個標量，其型別與最終矢量中的元素型別一樣。例如，`float4` 有下列形式：

表 6.4 其他內建數據型別

型別	描述
image2d_t	2D 圖像。節 *6.12.14*對於使用此型別的內建函式有詳細描述。
image3d_t	3D 圖像。節 *6.12.14*對於使用此型別的內建函式有詳細描述。
image2d_array_t	2D 圖像陣列。節 *6.12.14*對於使用此型別的內建函式有詳細描述。
image1d_t	1D 圖像。節 *6.12.14*對於使用此型別的內建函式有詳細描述。
image1d_buffer_t	由緩衝對象所創建的 1D 圖像。節 *6.12.14*對於使用此型別的內建函式有詳細描述。
image1d_array_t	1D 圖像陣列。節 *6.12.14*對於使用此型別的內建函式有詳細描述。
sampler_t	採樣器型別。節 *6.12.14*對於使用此型別的內建函式有詳細描述。
event_t	事件。可用來標識全局內存和局部內存之間的異步拷貝，參見節 *6.12.10*。

```
(float4)( float, float, float, float )
(float4)( float2, float, float )
(float4)( float, float2, float )
(float4)( float, float, float2 )
(float4)( float2, float2 )
(float4)( float3, float )
(float4)( float, float3 )

(float4)( float )
```

除了按照函式求值的標準規則對算元進行求值，還會對標量型別進行隱式拓寬。算元的求值順序是未定義的。會按照內存中的順序將算元賦值給最終矢量的相應元素。即，將第一個算元的第一個元素賦值給 result.x，將第一個算元的第二個元素（如果第一個算元是標量，則此處為第二個算元的第一個元素）賦值給 result.y，等等。如果只有單個標量算元，則會將其複製並賦值給最終矢量的所有元素。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, (float2)(2.0f, 3.0f), 4.0f);

float4  f = (float4)(1.0f, 2.0f);        <- error
```

6.1.7 矢量組件

對於具有 1 到 4 個組件的矢量數據型別，其組件可以用 <vector_data_type>.xyzw 來尋址。矢量數據型別 char2、uchar2、short2、ushort2、int2、uint2、long2、ulong2 和 float2 可以訪問元素 .xy。矢量數據型別 char3、uchar3、short3、ushort3、int3、uint3、long3、ulong3 和 float3 可以訪問元素 .xyz。矢量數據型別 char4、uchar4、short4、ushort4、int4、uint4、long4、ulong4 和 float4 可以訪問元素 .xyzw。

如果所訪問的組件在矢量型別中並不具備，則視為錯誤，例如：

```
float2 pos;     // is legal
pos.x = 1.0f;   // is illegal
pos.z = 1.0f;
float3 pos;     // is legal
pos.z = 1.0f;
pos.w = 1.0f;   // is illegal
```

在組件選擇文法中，可以將多個組件的名字附到句點 (.) 後面從而選擇多個組件。

```
float4 c;

c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
```

```
c.z = 1.0f;
c.xy = (float2)(3.0f, 4.0f);
c.xyz = (float3)(3.0f, 4.0f, 5.0f);
```

在組件選擇文法中，也可以將組件的順序打亂或重複。

```
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)
```

組件組符號可以出現在算式的左手邊。要形當左值用，必須對矢量組件進行重排（swizzling），並且不能重複，最終形成的左值是標量還是矢量取決於組件的數目。每個組件都必須是 OpenCL 所支持的標量或矢量型別。

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
pos.xw = (float2)(5.0f, 6.0f);// 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.xyz = (float3)(3.0f, 5.0f, 9.0f); // pos = (3.0f, 5.0f, 9.0f, 4.0f)
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);

float4 a, b, c, d;
float16 x;
x = (float16)(a, b, c, d);
x = (float16)(a.xxxx, b.xyz, c.xyz, d.xyz, a.yzw);

// illegal - component a.xxxxxxx is not a valid vector type
x = (float16)(a.xxxxxxx, b.xyz, c.xyz, d.xyz);
```

也可以用數值索引來訪問矢量數據型別的元素。表 6.6給出了可以使用的數值索引：

表 6.6 內建矢量數據型別的數值索引

矢量組件	可用的數值索引
2-component	0、1
3-component	0、1、2
4-component	0、1、2、3
8-component	0、1、2、3、4、5、6、7
16-component	0、1、2、3、4、5、6、7、8、9、a、A、b、B、c、C、d、D、e、E、f、F

數值索引前面必須加上字符 s 或 S。

下面例子中

```
float8  f;
```

f.s0 指代 float8 變量 f 的第一個元素，而 f.s7 則指代 float8 變量 f 的第八個元素。

下面例子中

```
float16 x;
```

x.sa（或 x.sA）指代 float16 變量 f 的第十一個元素，而 x.sf（或 x.sF）指代 float16 變量 f 的第十六個元素。

數值索引不能與.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
```

矢量數據型別可以使用後綴.lo（或.even）和.hi（或.odd）得到小號的矢量型別或者將小號的數據型別組合成大號的矢量型別。可以使用多級後綴.lo（或.even）和.hi（或.odd）直到變成標量。

後綴.lo 指代的是矢量中索引值較小的那一半。而後綴.hi 指代的是矢量中索引值較大的那一半。

後綴.even 指代的是矢量中索引值為偶數的元素。而後綴.odd 指代的是矢量中索引值為奇數的元素。

下面的例子有助於我們理解他:

```
float4 vf;

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

float2 even = vf.even;  // returns vf.xz
float2 odd = vf.odd;    // returns vf.yw
```

對於 3 組件的矢量型別應用後綴.lo (或.even) 和.hi (或.odd) 時, 會將 3 組件矢量型別當成 4 組件矢量型別使用, 且組件 w 未定義。下面給出了一些例子:

```
float8  vf;
float4  odd = vf.odd;
float4  even = 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] }
}

float3  vf = (float3)(1.0f, 2.0f, 3.0f);
float2  low = vf.lo; // (1.0f, 2.0f);
float2  high = vf.hi; // (3.0f, undefined);
```

不能對矢量元素取址, 否則會導致編譯錯誤。例如:

```
float8  vf;

float   *f = &vf.x;             // is illegal
```

```
4  float2  *f2 = &vf.s07;          // is illegal
5
6  float4  *odd = &vf.odd;         // is illegal
7  float4  *even = &vf.even;       // is illegal
8  float2  *high = &vf.even.hi;    // is illegal
9  float2  *low = &vf.odd.lo;      // is illegal
```

6.1.8 別名規則

OpenCL C 程式符合 C99 中基於型別的別名規則（在 C99 規範的節 6.5 的第 7 項中定義）。為了應用這些別名規則，OpenCL C 內建矢量數據型別會被當成聚合（aggregate）型別[11]。

6.1.9 關鍵字

下列名字將作為 OpenCL C 中的關鍵字而保留，不能挪作他用。

- C99 中作為關鍵字所保留的名字。
- *表 6.3*、*表 6.4* 和*表 6.5*中定義的 OpenCL C 數據型別。
- 位址空間限定符：*__global*、*global*、*__local*、*local*、*__constant*、*constant*、*__private* 和 *private*。
- 函式限定符：*__kernel* 和 *kernel*。
- 訪問限定符：*__read_only*、*read_only*、*__write_only*、*write_only*、*__read_write* 和 *read_write*。

節 6.2 轉換以及轉型

6.2.1 隱式轉換

OpenCL 支持*表 6.1*中所定義的內建標量型別間的隱式轉換（conversion）（`void` 和 `half`[12] 除外）。隱式轉換不僅是對算式值的重釋（reinterpret），同時會將其轉換成另一個型別與其相等的值。例如，可以將整數 `5` 轉換成浮點數 `5.0`。

對於內建矢量數據型別，不允許進行隱式轉換。

對於指針型別的隱式轉換遵循 C99 規範中所描述的規則。

6.2.2 顯式轉型

對於*表 6.1*中所定義的內建標量型別，標準的轉型（type-casting）動作會實施恰當的轉換（`void` 和 `half`[13] 除外）。下面例子中：

```
1  float   f = 1.0f;
2  int     i = (int)f;
```

`f` 存儲的是 `0x3F800000`，而 `i` 存儲的是 `0x1`，即將 `f` 中的浮點數 `1.0f` 轉換成了整數值。

對於矢量型別的顯式轉型是非法的。下面的例子會導致編譯錯誤。

```
1  int4    i;
2  uint4   u = (uint4) i;  <- not allowed
3
4  float4  f;
5  int4    i = (int4) f;   <- not allowed
6
7  float4  f;
8  int8    i = (int8) f;   <- not allowed
```

標量到矢量的轉換可以通過轉型來實施。轉型同時會實施恰當的算數轉換。轉換成內建整形矢量時會向零捨入。轉換成浮點矢量時會使用缺省的捨入模式。將 `bool` 轉型成整形矢量時，如果 `bool` 值是 `true`，會將矢量組件置為 −1（即設置了所有位），否則將矢量組件置為 0。

[11] 也就是說，出於使用基於型別的別名規則的目的，認為內建矢量數據型別等同於對應的陣列型別。
[12] 如果支持擴展 **cl_khr_fp16**，則 `half` 也在支持之列。
[13] 如果支持擴展 **cl_khr_fp16**，則 `half` 也在支持之列。

下面是顯式轉型的一些正確示例。

```
loat    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).

uchar4  vtrue = (uchar4)true;
// vtrue is a uchar4 vector with elements (0xff, 0xff,
//                                         0xff, 0xff).
```

6.2.3 顯式轉換

下面一組函式可用來實施顯式轉換

```
convert_destType(sourceType)
```

這組函式可以支持所有所支持型別（參加節 *6.1.1*、節 *6.1.2*和節 *6.1.3*）間的型別轉換，這些型別除外：`bool`、`half`、`size_t`、`ptrdiff_t`、`intptr_t`、`uintptr_t` 和 `void`。

源矢量和目的矢量的元素數目必須一樣。

下面例子中:

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

`convert_int4` 將 `uchar4` 矢量 `u` 轉換成了 `int4` 矢量 `c`。

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

`convert_int` 將 `float` 標量 `f` 轉換成了 `int` 標量 `i`。

可以通過兩個可選的修飾符（modifier）來改變轉換的行為，其中一個修飾符可以對溢出的輸入使用飽和算法（saturation），另一個可以指定捨入模式。

完整的標量轉換函式形如:

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

完整的矢量轉換函式形如:

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

6.2.3.1 數據型別

對於標量型別 `char`、`uchar`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`float` 以及由此所衍生的內建矢量型別，都可以進行轉換。算元和結果的型別中所含元素的數目必須相同。算元和結果的型別可能相同，這種情況下不會進行轉換，算式的型別和值都不變。

整數之間的轉換遵循 C99 規範的節 *6.3.1.1* 和節 *6.3.1.3* 中的轉換規則，不過節 *6.2.3.3*中所描述的溢出行為和飽和轉換例外。

6.2.3.2 捨入模式

與浮點型別有關的轉換會按照 IEEE-754 中的捨入規則進行捨入。轉換可以使用表 *6.7*中的修飾符指定捨入模式。

表 6.7 捨 入 模 式

修飾符	捨入模式描述
_rte	捨入為最近偶數。
_rtz	向零捨入。
_rtp	向正無窮捨入。
_rtn	向負無窮捨入。
沒有指定修飾符	使用目標型別的缺省捨入模式，轉換成整形時使用 _rtz，轉換成浮點型別時使用缺省的捨入模式。

缺省情況下，轉換成整形時使用捨入模式 _rtz，而轉換成浮點型別[14] 時使用缺省的捨入模式。所支持的唯一一個缺省的浮點捨入模式是捨入為最近偶數，即浮點型別的缺省捨入模式是 _rte。

6.2.3.3 溢出行為和飽和轉換

轉換時，如果算元大於目標型別所能表示的最大值，或者小於目標型別所能表示的最小值，就認為是溢出。其結果由 C99 規範的節 *6.3* 中的轉換規則來確定。如果要將浮點型別轉換成整形，則其行為依賴於具體實作。

轉換成整形時可以使用飽和模式 (在轉換函式的名字後面加上修飾符 _sat)。此模式下，如果發生溢出，結果將是目標格式所能表示的值中與源算元最近的那個 (會將 NaN 轉換成 0)。

轉換成浮點型別時將遵循 IEEE-754 中的捨入規則。轉換成浮點格式時不能使用修飾符 _sta。

6.2.3.4 顯式轉換的例子

例 6.1

```
short4  s;

// negative 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 6.2

```
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 );
```

[14] 轉換成浮點格式時，如果源的值超過了目標型別所能表示的最大有限浮點值，捨入模式會影響轉換結果，根據 IEEE-754 的捨入規則來確定結果是最大有限浮點值，還是和源有相同正負號的無窮值。

例 6.3

```
int4    i;

// convert ints to floats using the default 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 將數據重釋為其他型別

在 OpenCL 中，經常有需要將某一種數據型別重釋為另一種數據型別。在需要直接訪問浮點型別中的位元時就需要這樣做，例如，屏蔽掉（mask off）浮點數據的符號位，或者使用浮點矢量關係算子的結果（參加節 *6.3*中的第 *d*項）[15]。對於這種（反）轉換，C 語言中經常使用的有這幾種方法：指針別名（pointer aliasing）、聯合體（union）和內存拷貝（memcpy）。對於 C99，這些方法中只有內存拷貝是嚴格正確的。由於 OpenCL 沒有提供 **memcpy**，所以需要其他手段。

6.2.4.1 使用聯合體重釋型別

OpenCL 語言對聯合體（union）進行了擴充，允許程式使用其中某個成員訪問另一個型別不同的成員。對象中的相關字節會被視為訪問時所用型別的對象。如果訪問時所用型別大於對象的實際型別，則額外字節的值是未定義的。

例 6.4

```
union{ float f; uint u; double d[16];} 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
                 // 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_type() 和 as_typen() 重釋型別

表 6.1、*表 6.3*中所描述的所有型別（bool、half[17] 和 void 除外）都可以重釋為另一種大小相同的數據型別，如果是標量型別則使用 **as_type**，如果是矢量型別則使用 **as_typen**[18]。如果算

[15] 另外，對於一些設計用來支持特定矢量 ISA（如 AltiVec™、CELL Broadband Engine™ 架構）的 C 語言擴展，會將這樣的轉換與重排算子（swizzle oprator）一起使用對型別進行還原（unconversion）。為支持這種遺留代碼，**as_typen**() 允許在大小相同但元素數目不同的矢量間進行轉換，即使這種轉換的行為不能移植到其他硬件架構的 OpenCL 實作上。 AltiVec™ 是 Motorola Inc. 的商標。 Cell Broadband Engine 是 Sony Computer Entertainment Inc. 的商標。

[16] 僅當支持雙精度時有效。

[17] 除非支持擴展 **cl_khr_fp16**。

[18] 聯合體用來反應數據在內存中是怎麼組織的，而 **as_type** 和 **as_typen** 則是用來反應數據在寄存器中是怎麼組織的。如果在設備上算元和結果共享同一個寄存器檔（register file），則 **as_type** 和 **as_typen** 在編譯和應該不會產生任何指令。注意，內存中數據組織方式的不同大部分都是由於端序（endianness）的不同所導致，因此基於寄存器的表示方式也可能會由於寄存器中元素大小的原因而不同。（例如，在某種架構上，可能將 char 裝載進 32 位寄存器中，或者將 char 矢量裝載進元素大小固定為 32 位的 SIMD 矢量寄存器。）如果元素數目不同，則實作可以根據直觀感受選擇恰當的數據表示方式，如同在包含源型別數據的寄存器上應用相應的型別算子一樣。如果元素數目一樣，則 **as_typen** 的行為必須與用內存拷貝或聯合體重釋類似的數據型別相一致。所以，例如，如果某種實作中所有單精度數據在寄存器中都是以雙精度存儲的，那麼他所實現的 as_int(float) 就應該先將雙精度數據轉換成單精度，然後（如果有必要）再將單精度數據的位元轉移到合適的寄存器中以參與整形數據運算。如果不同位址空間中所存儲的數據具有不同的端序，則按設備的主端序（dominant endianness）進行處理。

元和結果的型別所含元素數目相同，則會直接返回算元中的位元，而不會將其修正為新型別。通常對函式參數都會實施型別晉陞（type promotion），而這裡不會。

例如，`as_float(0x3f800000)` 返回 `1.0f`，而 `1.0f` 正式將 `0x3f800000` 看作 IEEE-754 單精度浮點數時的值。

如果算元和結果類型所包含元素的數目不同，則結果依賴於具體實作，但將四元矢量轉換為三元矢量則例外。這種情況下，會直接返回算元中的位元，不會將其修正為新類型。也就是說，如果算元和結果類型所包含元素的數目不同，符合 OpenCL 的實作需要將其行為顯式的定義清楚，但兩種實作的行為不必相同。實作所定義的結果中可能包含所有或部分原始位元，也可能完全沒有原始位元，而且位元的順序也可由實作自由選擇。不能使用算子 **as_type** 和 **as_type*n*** 將數據重釋為另一種字節數不同的型別。

例 6.5

```
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
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 sizes
double4 g = as_double4[19](f);

float4 f;
// Legal. g.xyz will have same values as f.xyz.
float3 g = as_float3(f);
```

6.2.5 指針轉型

分別指向新舊型別的指針可以相互轉換。將指針轉型成新的型別時會斷言位址已經正確對齊，雖然實際並沒有做檢查。開發人員還是需要知道 OpenCL 设备的端序和數據的端序，從而可以確定標量和矢量數據元素在內存中的存儲方式。

6.2.6 常見的算術轉換

許多期望算元為算術類型的算子所引起轉換並得到結果的方式都類似。其目的是為算元和結果確定一種兩者通用的實型（real type，即實數）。在不改變型域（type domain）的情況下，所有算元都會進行轉換，其目標型別所對應的實型即為上面所說的通用實型。出於這個目的，所有的矢量型別都比標量具有更高的轉換級別（conversion rank）。除非顯式規定，否則上面所說的通用實型也是結果所對應的實型，如果與算元的實型相同，則其型域也相同，否則為複數。這種模式稱作常用的算術轉換（usual arithmetic conversion）。如果型別為矢量的算元多於一個，則會發生錯誤。按照節 *6.2.1*，不允許在矢量型別間進行隱式轉換。

[19] 僅當支持雙精度時才有效。

否則，如果算元中只有一個是矢量型別，其他都是標量型別，則會將標量型別轉換成矢量元素的型別，然後將其拓寬成一個新的矢量，其元素數目與矢量算元相同，且所有元素的值都與標量值相同。如果任一標量算元的轉換級別比矢量算元元素的級別高，那麼就會發生錯誤。出於此目的，轉換級別的次序定義如下：

1. 對於兩個浮點型別，如果第一個可以精確的表示另一個型別的所有數值，則第一個的級別比另一個高。（出於此目的，使用的是浮點值的完整編碼方案，而设备所用的可能是其子集。）
2. 任一浮點型別的級別都比任一整數型別的高。
3. 對於兩個整數型別，精度較高的那個的級別比另一個要高。
4. 對於精度相同的兩個整數型別，無符號的級別比有符號的要高[20]。
5. `bool` 的級別比其他所有型別都低。
6. 枚舉型別的級別和與其兼容的整數型別的級別相同。
7. 對於所有型別，T1、T2 和 T3，如果 T1 的級別比 T2 高，並且 T2 的級別比 T3 高，那麼 T1 的級別也比 T3 高。

而如果所有算元都是標量，則按照 C99 標準的節 *6.3.1.8* 應用算術轉換。

> 經過審查，*C99* 的節 *6.3.1.8* 和節 *6.3.1.1* 中的標準次序都被駁回了。如果使用這裡所定義的整形轉換級別，`int4 + 0U` 是合法的且所返回的型別為 `int4`。如果對於標量使用 *C99* 中標準的算術轉換規則，則會對整形矢量的元素實施標準的整數型別晉陞，`short8 + char` 所返回的型別為 `int8`，或者違規。

节 6.3 算子

a. 算術算子加 (+)、減 (-)、乘 (*)、除 (/) 都可以在內建的整數、浮點數標量和矢量數據型別上進行運算。而模除 (%) 則只能在內建的整數標量和整數矢量數據型別上進行運算。在算元的型別轉換後，所有算術算子所返回的結果都與算元具有相同的型別，且都是某種內建型別 (整數或浮點數)。轉換後，有如下情況：

   - 兩個算元都是標量。這種情況下，運算結果也是標量。
   - 一個算元是標量，另一個是矢量。這種情況下，會對標量算元進行算術轉換，使其型別與矢量算元的元素型別相同。然後將標量算元拓寬成矢量，其組件數目與矢量算元相同。最後按組件逐一進行運算，結果是同樣大小的矢量。
   - 兩個算元都是矢量，且型別相同。這種情況下，會按組件逐一進行運算，結果是同樣大小的矢量。

   任何其他情況下的隱式轉換都是違規的。對於整數型別的除法，如果結果無法用此型別表示，即發生了溢出 (上溢或下溢)，並不會引起異常，但結果的值沒有指定。這跟整數除以零的效果一樣。而浮點數除以零則會導致 $\pm\infty$ 或 NaN，就像 IEEE-754 標準中所規定的那樣。內建函式 **`dot`** 和 **`cross`** 分別為矢量點乘和矢量叉乘。

b. 算術單元算子 (+ 和 -) 可在內建標量和矢量類型上進行運算。

c. 算術算子中的後置式或前置式遞增和遞減算子可以在內建的標量和矢量型別上進行運算，但內建的標量和矢量浮點型別除外[21]。所有單元算子都會在算元的所有組件上進行運算。結果的型別與算元的型別相同。對於後置式或前置式遞增和遞減算子，算式必須是可賦值的 (即左值)。前置式遞增或前置式遞減算子會對所操作算式的內容加 1 或減 1，整個算式的值就是修改後的值。而後置式遞增或後置式遞減算子也會對所操作算式的內容加 1 或減 1，但整個算式的值是修改前的值。

d. 關係算子[22] 大於 (>)、小於 (<)、大於等於 (>=)、小於等於 (<=) 可在標量或矢量型別上進行運算。所有關係算子的結果都是整數型別。在對算元進行型別轉換後，有如下情況：

[20] 這與 C99 TC2 的節 *6.3.1.1* 中所描述的標準的整數轉換級別不同。

[21] 在浮點數上使用前置式和後置式遞增算子時可能會導致意料之外的行為，因此 OpenCL 中不支持對浮點標量和矢量型別運用此算子。例如，如果一個變量為浮點型別，其值為 `0x1.0p25f`，`a++` 返回的還是 `0x1.0p25f`。同時，如果 `a` 中含有小數，不保證 `(a++) - a` 會返回 `a`。如果不是缺省的捨入模式，則 `(a++) - a` 的結果可能跟 `a++` 或 `a - a` 的結果相同。

[22] 對於矢量關係運算，結果也是矢量，要想知道其中的元素是否有 `true` 或者是否所有元素都是 `true`，例如，在 `if ()` 語句的上下文中使用，請查閱節 *6.12.6*中的內建函式 **`any`** 和 **`all`**。

- 兩個算元都是標量。這種情況下，運算結果是 int 標量。
- 一個算元是標量，另一個是矢量。這種情況下，會對標量算元進行算術轉換，使其型別與矢量算元的元素型別相同。然後將標量算元拓寬成矢量，其組件數目與矢量算元相同。最後按組件逐一進行運算，結果是同樣大小的矢量。
- 兩個算元都是矢量，且型別相同。這種情況下，會按組件逐一進行運算，結果是同樣大小的矢量。

任何其他情況下的隱式轉換都是違規的。如果算元都是標量，則結果是標量帶符號整數型別 int。而如果算元都是矢量型別，則結果是矢量帶符號整數型別，其元素數目與算元相同。如果矢量算元的元素型別為 char*n* 和 uchar*n*，則結果為 char*n*。如果矢量算元的元素型別為 short*n* 和 ushort*n*，則結果為 short*n*。如果矢量算元的元素型別為 int*n*、uint*n* 和 float*n*，則結果為 int*n*。如果矢量算元的元素型別為 long*n*、ulong*n* 和 double*n*，則結果為 long*n*。對於標量型別的關係運算，如果所指定的關係為 false 則結果為 0，否則結果為 1。對於矢量型別的關係運算，如果所指定的關係為 false 則結果為 0，否則結果為 -1 (即設置了所有位)。如果任一引數不是數字 (NaN)，則關係算子的結果就為 0。

e. 等號算子[23] 相等 (==)、不等 (!=) 可對所有內建標量和矢量型別進行運算。所有等號算子的結果均為整數型別。在對算元進行型別轉換後，有如下情況:

- 兩個算元都是標量。這種情況下，運算結果也是標量。
- 一個算元是標量，另一個是矢量。這種情況下，會對標量算元進行算術轉換，使其型別與矢量算元的元素型別相同。然後將標量算元拓寬成矢量，其組件數目與矢量算元相同。最後按組件逐一進行運算，結果是同樣大小的矢量。
- 兩個算元都是矢量，且型別相同。這種情況下，會按組件逐一進行運算，結果是同樣大小的矢量。

任何其他情況下的隱式轉換都是違規的。如果算元都是標量，則結果是標量帶符號整數型別 int。而如果算元都是矢量型別，則結果是矢量帶符號整數型別，其元素數目與算元相同。如果矢量算元的元素型別為 char*n* 和 uchar*n*，則結果為 char*n*。如果矢量算元的元素型別為 short*n* 和 ushort*n*，則結果為 short*n*。如果矢量算元的元素型別為 int*n*、uint*n* 和 float*n*，則結果為 int*n*。如果矢量算元的元素型別為 long*n*、ulong*n* 和 double*n*，則結果為 long*n*。對於標量型別的等號運算，如果所指定的關係為 false 則結果為 0，否則結果為 1。對於矢量型別的等號運算，如果所指定的關係為 false 則結果為 0，否則結果為 -1 (即設置了所有位)。如果任一引數不是數字 (NaN)，則相等算子 (==) 的結果就為 0。如果任一引數不是數字 (NaN)，則不等算子 (!=) 的結果就為 1 (對於標量算元) 或 -1 (對於矢量算元)。

f. 位元算子與 (&)、或 (|)、異或 (^)、非 (~) 可以對所有內建的標量或矢量型別進行運算，但是浮點型別除外 (無論標量還是矢量)。對於矢量內建型別，這些算子會按組件逐一運算。如果一個算元為標量，而另一個是矢量，則會對標量算元進行算術轉換，使其型別與矢量算元的元素型別相同。然後將標量算元拓寬成矢量，其組件數目與矢量算元相同。最後按組件逐一進行運算，結果是同樣大小的矢量。

g. 邏輯算子與 (&&)、或 (||) 可對所有內建標量或矢量型別進行運算。對於內建的標量型別，如果左手的算元不等於 0，則邏輯與算子 (&&) 只對右手的算元進行求值。對於內建的標量型別，如果左手的算元不等於 0，則邏輯或算子 (||) 只對右手的算元進行求值。對於內建的矢量型別，會對兩個算元都求值，並按組件逐一進行運算。如果一個算元是標量，另一個是矢量，則會對標量算元進行算術轉換，使其型別與矢量算元的元素型別相同。然後將標量算元拓寬成矢量，其組件數目與矢量算元相同。最後按組件逐一進行運算，結果是同樣大小的矢量。

邏輯異或算子 (^^) 是保留關鍵字。

如果算元都是標量，則結果是標量帶符號整數型別 int。而如果算元都是矢量型別，則結果是矢量帶符號整數型別，其元素數目與算元相同。如果矢量算元的元素型別為 char*n* 和 uchar*n*，則結果為 char*n*。如果矢量算元的元素型別為 short*n* 和 ushort*n*，則結果為 short*n*。如果矢量算元的元素型別為 int*n*、uint*n* 和 float*n*，則結果為 int*n*。如果矢量算元的元素型別為 long*n*、ulong*n* 和 double*n*，則結果為 long*n*。

對於標量型別的邏輯運算，如果所指定的關係為 false 則結果為 0，否則結果為 1。對於矢量型別的邏輯運算，如果所指定的關係為 false 則結果為 0，否則結果為 -1 (即設置了所有位)。

h. 邏輯單元算子非 (!) 可對所有內建標量或矢量型別進行運算。對於內建矢量型別，會按組件逐一運算。

[23] 對於矢量關係運算，結果也是矢量，要想知道其中的元素是否有 true 或者是否所有元素都是 true，例如，在 if () 語句的上下文中使用，請查閱節 *6.12.6*中的內建函式 **any** 和 **all**。

如果算元是標量，則結果是標量帶符號整數型別 int。而如果算元是矢量型別，則結果是矢量帶符號整數型別，其元素數目與算元相同。如果矢量算元的元素型別為 char*n* 和 uchar*n*，則結果為 char*n*。如果矢量算元的元素型別為 short*n* 和 ushort*n*，則結果為 short*n*。如果矢量算元的元素型別為 int*n*、uint*n* 和 float*n*，則結果為 int*n*。如果矢量算元的元素型別為 long*n*、ulong*n* 和 double*n*，則結果為 long*n*。

對於標量型別的邏輯非運算，如果算元不等於 0 則結果為 0，否則結果為 1。對於矢量型別的邏輯非運算，如果算元不等於 0 則結果為 0，否則結果為 -1（即設置了所有位）。

i. 三元選擇算子（?:）會在三個算式上進行運算（exp1 ? exp2 : exp3）。此算子會先對 exp1 進行求值，結果可能是標量或矢量，但不會是浮點數。如果結果是標量，且不等於 0，那麼接着對第二個算式進行求值，否則對第三個算式求值。如果結果是矢量，則等同於調用 **select**(exp3, exp2, exp1)。函式 **select** 在表 *6.17*中有所描述。第二個和第三個算式的結果可以是任意型別，只要型別匹配即可；如果不匹配，則會進行隱式轉換（參見節 *6.2.1*）使其匹配；而如果一個是矢量，另一個是標量，則會對標量進行算術轉換，使其型別與矢量算元的元素型別相同，然後將其拓寬成型別與矢量算元相同的矢量。其結果的型別就是整個算式的型別。

j. 算子右移（>>）和左移（<<）可對所有內建的標量或矢量型別進行運算，但浮點型別除外。對於內建的矢量型別，會按組件逐一進行運算。對於右移算子（>>）和左移算子（<<），如果第一個算元是標量，則最右邊的算元也必須是標量；如果第一個算元是矢量，則最右邊的算元即可以是矢量，也可以是標量。

E1 << E2 的結果就是將 E1 左移 x 位，x 等於將 E2 視為無符號整數時其低 $log_2 N$ 位的值；如果 E1 是標量，則 N 就是在型別晉陞[24] 後，表示 E1 的數據型別所用的位數，如果 E1 是矢量，則 N 為表示 E1 的元素所用的位數。騰出的位元會填零。

E1 >> E2 的結果就是將 E1 右移 x 位，x 等於將 E2 視為無符號整數時其低 $log_2 N$ 位的值；如果 E1 是標量，則 N 就是在型別晉陞後，表示 E1 的數據型別所用的位數，如果 E1 是矢量，則 N 為表示 E1 的元素所用的位數。如果 E1 是無符號型別，或者是帶符號型別但其值非負，則騰出的位元會填零。如果 E1 是帶符號型別，且是負值，則騰出的位元會填一。

k. 算子 sizeof 的結果就是算元的大小（單位字節），其中包括對齊時所需的填補字節（參見節 *6.1.5*）。算元可能是算式，也可能是帶小括號的型別。這個大小是由算元的型別所決定的。結果的型別為 size_t。如果算元的型別為變長陣列[25]，則會對算元進行求值；否則，不求值，並且結果是整形常數。

使用此算子時，如果算元的型別為 char、uchar，結果為 1；如果算元的型別為 short、ushort 或 half，則結果為 2；如果算元的型別為 int、uint 或 float，則結果為 4；如果算元的型別為 long、ulong 或 double，則結果為 8；如果算元是矢量，則結果為組件數目與每個標量組件大小的乘積[26]；如果算元是陣列，則結果為整個陣列的大小；如果算元的型別為結構體或聯合體，結果就是這樣一個物件的大小，包括內部或尾部的填補字節。算元不能是函式型別或不完整的型別的算式，也不能是帶小括號的這種型別的名字，位欄結構體成員也不行[27]。

如果算元的型別為 bool、image2d_t、image3d_t、image2d_array_t、image1d_t、image1d_buffer_t、image1d_array_t、sampler_t 或 event_t，則結果依賴於具體實作。

l. 算子逗號（,）在運算時，會返回以逗號分隔的算式列中最右邊那個的型別和值。會按從左到右的順序對所有算式進行求值。

m. 單元算子（*）是一種間接指示。如果算元指向一個對象，則結果為左值，指代此對象。如果此對象的型別為“type”，則結果的型別也是“type”。如果賦給指針的值無效，則單元算子（*）的行為未定義[28]。

n. 單元算子（&）會返回算元的位址。如果算元的型別為“type”，則結果的型別也是“type”。如果算元是單元算子 * 的結果，則不會對算子 * 和 & 進行求值，如同將兩者都省略，但算子的相關限制還是有效，並且結果依然不是左值。類似的，如果算元是算子 [] 的結果，則不會對算

[24] 整數的型別晉陞在 ISO/IEC 9899: 1999 的節 6.3.1.1 中有所描述。
[25] OpenCL 1.1 不支持變長陣列，參見節 *6.9*中的第 *d*項。
[26] 三元矢量是個例外，其大小定義為 4 與每個標量組件大小的乘積。
[27] OpenCL 1.1 不支持位欄結構體成員，參見節 *6.9*中的第 *c*項。
[28] 無效值指的是算元為 null 指針、位址沒有根據對象的型別正確對齊、所指對象的生命周期已經結束。如果 *P 是左值，而 T 是某種對象指針型別的名字，則 *(T)P 是左值，且其型別與 T 所指對象的型別相兼容。

子 & 和算子 [] 所隱含的 * 進行求值，如同移除了算子 &，並將算子 [] 改成了算子 +。否則，結果將是指針，指向算元所指代的對象[29]。

o. 賦值算子 (=) 可以給變量賦值，如

```
lvalue = expression
```

賦值算子會將 expression 的值存入 lvalue。expression 和 lvalue 的型別必須相同，如果不同，則 expression 的型別必須列於*表 6.1*中，這種情況下，賦值前會先進行隱式轉換。

如果 expression 是標量，則 lvalue 是矢量，則會將標量轉換成矢量元素的型別，然後將標量算元拓寬成矢量，其組件數目與矢量算元相同。最後按組件逐一賦值。

要想進行其他轉換，必須顯式指定。左值必須是可寫的。如果變量是內建型別、整個結構體或陣列、結構體欄位、用欄位選擇器 (.) 對組件進行選擇或重排 (欄位互不重複) 所形成的左值、括號中的左值、或者用陣列下標算子 ([]) 提領出的左值，那麼這個變量就是左值。其他單元或二元算式、函式名、帶有重複欄位的重排以及常量都不可為左值。三元算子 (?:) 也不能是左值。

沒有規定對算元求值的順序。如果嘗試修改賦值算子的結果，或者在下個時序點 (sequence point) 後面訪問他，其行為未定義。其他賦值算子有 +=、 -=、 *=、 /=、 %=、 <<=、 >>=、 &=、 |=、 ^=。

算式

```
lvalue op= expression
```

等同於

```
lvalue = lvalue op expression
```

其中 lvalue 和 expression 必須同時滿足算子 op 和 = 的要求。

> 除了算子 sizeof，本節所列的任一算子都不能用於數據型別 half。

节 6.4 矢量運算

矢量運算是按組件逐一進行的。通常，當算子在矢量上運算時，矢量的每個組件都是獨立進行的。

例如:

```
float4  v, u;
float   f;

v = u + f;
```

等同於

```
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;
```

等同於

```
w.x = v.x + u.x;
```

[29] 因此，&*E 等同於 E (即使 E 是 null 指針)，而 &(E1[E2]) 等同於 ((E1)+(E2))。如果 E 是左值，並且作為算子 & 的算元是有效的，那麼 *&E 也是左值，並且等同於 E，這永遠成立。

```
w.y = v.y + u.y;
w.z = v.z + u.z;
w.w = v.w + u.w;
```

對於大多數算子以及所有整數、浮點數矢量型別都是這樣的。

节 6.5 位址空間限定符

OpenCL 實現了下列相互分離的位址空間：**__global**、**__local**、**__constant**、**__private**。在聲明變量時，可以使用位址空間限定符來指定用哪塊區域的內存來分配對象。OpenCL 對 C 中型別限定符的語法做了擴展，OpenCL 中的型別限定符可以包含一個位址空間名。如果某個對象的型別中帶有位址空間名，則在指定的位址空間中分配此對象；否則，在缺省的位址空間中分配此對象。

也可以使用不帶前綴 __ 的位址空間限定符即 **global**、**local**、**constant**、**private** 來代替相應帶前綴 __ 的位址空間限定符。

對於程式中的函式引數以及函式的局部變量而言，缺省的位址空間為 *__private*。所有函式引數必須位於 *__private* 位址空間中。

如果 *__kernel* 函式的引數聲明為指針或某種型別的陣列，則他指向的位址空間必須是 *global*、*local*、*constant*。兩個指針要想相互賦值，他們必須指向同一個位址空間。將指向位址空間 A 的指針轉型成為指向位址空間 B 的指針時的。

如果函式引數的型別為 *image2d_t*、*image3d_t*、*image2d_array_t*、*image1d_t*、*image1d_buffer_t* 或 *image1d_array_t*，則必須在 *__global* 位址空間中分配。

如果變量的作用域是整個程式，則沒有對應的缺省位址空間。這樣的變量在聲明時必須加上 *__constant*。

例：

```
// 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];
```

函式所返回的值沒有對應的位址空間。聲明函式時，如果在返回的型別上加了位址空間限定符，則會產生編譯錯誤；除非函式返回的是指針，並且限定符是用在指針指向的對象上。

例：

```
__private int f() {... } // should generate an error
__local int *f() {... } // allowed
__local int * __private f() {... }; // should generate an error.
```

6.5.1 __global（或 global）

位址空間名 **__global** 或 **global** 用來指代分配自全局內存的內存對象（緩衝對象或圖像對象）。

緩衝對象可聲明為指針，指向標量、矢量或用戶自定義的結構體。這樣內核就可以讀寫緩衝區中任意位置的內容。

在主機代碼中通過調用相應 API 分配內存對象陣列時，他的實際大小就確定了。

一些例子：

```
__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.
```

如果聲明時帶有此限定符的引數上附着的是圖像對象，則根據所附着對象的類型不同，聲明此引數時其型別必須是 image2d_t (2D 圖像對象)、image3d_t (3D 圖像對象)、image2d_array

_t (2D 圖像陣列對象)、image1d_t (1D 圖像對象)、image1d_buffer_t (1D 圖像緩衝對象)或 image1d_array_t (1D 圖像陣列對象)。不能直接訪問圖像對象的元素。 OpenCL 提供有內建函式可用來讀寫圖像對象。

限定符 **const** 可以與 **__global** 一起使用來聲明一個只讀的緩衝對象。

6.5.2 __local（或 local）

如果要在局部內存中分配變量，並在某個作業組中的所有作業項間共享，則可以使用位址空間名 **__local** 或 **local**。指向 **__local** 位址空間的指針可以作為函式（包括內核函式）的引數。對於在內核函式內聲明的 **__local** 變量，其作用域僅為此內核函式。

關於在內核函式內聲明的 **__local** 變量，下面是一些例子:

```
__kernel void my_func(...)
{
        __local float   a;      // A single float allocated
                                 // in local address space
        __local float   b[10];  // An array of 10 floats
                                 // allocated in local address space.

        if (...)
        {
                // example of variable in __local address space
                // but not declared at __kernel function scope.
                __local float   c;      <- not allowed.
        }
}
```

不能對在內核函式內聲明的 **__local** 變量進行初始化:

```
__kernel void my_func(...)
{
        __local float   a = 1;  <- not allowed

        __local float   b;
        b = 1;                   <- allowed
}
```

> 如果內核函式內聲明了 **__local** 變量，則會為執行此內核的每個作業組都分配一個，並且只在作業組的生命周期內才存在。

6.5.3 __constant（或 constant）

位址空間名 **__constant** 或 **constant** 所描述的變量分配自全局內存，並且在內核中作為只讀變量來訪問。所有作業項（包括全局作業項）在執行內核時都可以讀取這些變量。

所有常值字串都存儲在 *__constant* 位址空間中。

> 在給內核的常數引數進行計數時，每個指向 *__constant* 位址空間的指針引數都會使計數增一。參見表 4.3中的 CL_DEVICE_MAX_CONSTANT_ARGS。

如果變量的作用域是整個程式或內核函式的最外層，則可以在 *__constant* 位址空間中聲明他。必須對這種變量進行初始化，並且所用的值必須是編譯時常數。對這種變量的寫操作會導致編譯時錯誤。

不要求實作將這種聲明聚合進幾個常數引數中，以使引數個數最少。這種行為依賴於具體實作。

想要代碼是可移植的，就必須做最保守的假定，即每個在函式或程式的作用域中聲明的 **__constant** 變量在計數時都算是一個單獨的引數。

6.5.4 __private（或 private）

內核函式中所聲明的不帶位址空間限定符的變量，非內核函式中的所有變量，以及所有函式引數都位於 **__private** 或 **private** 位址空間中。如果所聲明的變量是指針，並且沒有指定位址空間限定符，則認為他指向的是 **__private** 位址空間。

所保留的關鍵字 *__global*、*__constant*、*__local*、*__private*、*global*、*constant*、*local* 和 *private* 只作為位址空間限定符使用，不作他用。

节 6.6 訪問限定符

內核參數中的圖像對象可以聲明為只讀的或只寫的。內核不能對一個圖像對象即讀又寫。對於內核參數中的圖像對象，如果內核要讀取他，則聲明時要加上限定符 **__read_only**（或 **read_only**）；如果內核要寫入他，則聲明時要加上限定符 **__write_only**（或 **write_only**）。缺省的限定符為 **__read_only**。

下面例子:

```
1  __kernel void foo (read_only image2d_t imageA,
2                     write_only image2d_t imageB)
3  {
4          ....
5  }
```

imageA 是一個只讀的 2D 圖像對象，而 imageB 是一個只寫的 2D 圖像對象。

保留關鍵字 *__read_only*、*__write_only*、*__read_write*、*read_only*、*write_only* 和 *read_write* 只能用作訪問限定符，不能挪作他用。

节 6.7 函式限定符

6.7.1 __kernel（或 kernel）

限定符 **__kernel**（或 **kernel**）可以將函式聲明為內核，使其可由應用在 OpenCL 设备上執行。對於用此限定符聲明的函式有如下規定:

- 僅可在设备上執行;
- 可由主機調用;
- 當被另一個內核函式調用時，*__kernel* 函式僅僅是一個常規函式。

> 如果內核函式中聲明的變量帶有限定符 *__local* 或 *local*，則主機可以使用恰當的 *API* 像 **clEnqueueNDRangeKernel** 和 **clEnqueueTask** 來調用他。
>
> 如果內核函式中聲明的變量帶有限定符 *__local* 或 *local*，那麼在其他內核函式中調用他會發生什麼依賴於具體實作。
>
> 保留關鍵字 **__kernel** 和 **kernel** 只能用作函式的限定符，不能挪作他用。

6.7.2 可選特性限定符

限定符 *__kernel* 可以與關鍵字 *__attribute__* 一起使用，從而為內核函式聲明額外的資訊，如下所述。

可選特性 __attribute__((vec_type_hint(<type>)))[30] 可以給編譯器暗示 *__kernel* 可計算的寬度，編譯器在對代碼進行自動矢量化時，可以將其作為基準來計算處理器的可利用帶寬。其中 <type> 是表 6.3中所列的內建矢量型別，或者是構成其元素的標量型別。如果沒有指定 vec_type_hint (<type>)，則假定內核具有限定符 __attribute__((vec_type_hint(int)))。

例如，如果開發人員指定的寬度是 float4，則編譯器應當假定通常使用四路浮點矢量進行計算，並可以決定合併作業項或者甚至可能將一個作業項分成多個線程，以更好的匹配硬件的能力。對於合格的實作，不要求可以對代碼自動矢量化，但應當支持這個暗示。即使沒有這個暗示，編譯

[30] 自動矢量化（autovectorization）時會做隱式的假設：*__kernel* 中調用的所有庫都必須可在運行時重新編譯，只有這樣，編譯器才可以合併或分離作業項。這可能意味着，這樣的庫不能是編死的二元碼，如果是編死的二元碼，則必須帶有源碼或可重定向的中間表示。在某些情況下，這可能導致代碼安全問題。

器也可能自動矢量化。如果實作將 N 個作業項合併到一個線程中，則他要負責正確地處理這種情況：全局或局部作業項的數目在任一維度上模 N 不為零。

例：

```
1  // autovectorize assuming float4 as the
2  // basic computation width
3  __kernel __attribute__((vec_type_hint(float4)))
4  void foo( __global float4 *p ) {.... }
5
6  // autovectorize assuming double as the
7  // basic computation width
8  __kernel __attribute__((vec_type_hint(double)))
9  void foo( __global float4 *p ) {.... }
10
11 // autovectorize assuming int (default)
12 // as the basic computation width
13 __kernel
14 void foo( __global float4 *p ) {.... }
```

舉個例子，如果聲明 *__kernel* 函式時帶有 `__attribute__((vec_type_hint (float4)))`（即 *__kernel* 函式中大部分運算都已使用 `float4` 顯式矢量化），並且此內核在使用 Intel® AVX（Intel® Advanced Vector Instructions，實現了 8 個 `float` 寬度的矢量單元），則自動矢量化時可能選擇將兩個作業項合併成一個線程，第二個作業項將在 256 位 AVX 寄存器的高 128 位中運行。

另一個例子，Power4 機器具有兩個雙精度浮點單元，均為六級流水線結構（6-cycle deep pipe）。針對這種機器，自動矢量化時可能選擇將六個聲明時帶有限定符 `__attribute__(( vec_type_hint (double2)))` 的內核間插到同一個硬件線程中，從而可以讓 FPU 可以一直 12 路全速並行。考慮到資源利用或者一些偏好，如 2 的指數，即僅將 4 個或 8 個（或者其他數目的）作業項進行合併，只要自動矢量化時認定這種抉擇更好就行。

可選特性 `__attribute__((work_group_size_hint(X, Y, Z)))` 可以給編譯器暗示作業組的大小，即最有可能傳給 **clEnqueueNDRangeKernel** 的引數 *local_work_size* 的值。例如，`__attribute__((work_group_size_hint(1, 1, 1)))` 就是暗示編譯器內核最有可能在大小為 1 的作業組中執行。

可選特性 `__attribute__((reqd_work_group_size(X, Y, Z)))` 則是 **clEnqueueNDRangeKernel** 的引數 *local_work_size* 必須使用的值。這樣編譯器就可以針對此內核對生成的代碼進行適當的優化。如果內核是通過 **clEnqueueTask** 執行的，則指定此特性時必須是 `__attribute__((reqd_work_group_size(1, 1, 1)))`。

如果 `Z` 是一，則 **clEnqueueNDRangeKernel** 的引數 *work_dim* 可以是 2 或 3。如果 `Y` 和 `Z` 都是一，則 **clEnqueueNDRangeKernel** 的引數 *work_dim* 可以是 1、2 或 3。

节 6.8 存儲類別限定符

所支持的存儲類別限定符有 *typedef*、*extern* 和 *static*。而 *auto* 和 *register* 則不在支持之列。

存儲類別限定符 *extern* 只可用在函式（包括內核函式和非內核函式）、在程式作用域內聲明的全局變量以及函式（包括內核函式和非內核函式）內部聲明的變量上。而存儲類別限定符 *static* 只可用在非內核函式和在程式作用域內聲明的全局變量上。

例：

```
1  extern constant float4 noise_table[256];
2  static constant float4 color_table[256];
3
4  extern kernel void my_foo(image2d_t img);
5  extern void my_bar(global float *a);
6
7  kernel void my_func(image2d_t img, global float *a)
8  {
9          extern constant float4 a;
10         static constant float4 b;       // error.
```

```
11          static float c;                 // error.
12
13          ...
14          my_foo(img);
15          ...
16          my_bar(a);
17          ...
18  }
```

节 6.9 限制[31]

a. 對於指針的使用有一點限制。規定如下:
   - 對於內核函式，程式中聲明其參數時必須帶有限定符 *__global*、*__constant* 或 *__local*。
   - 兩個指針相互賦值時，在聲明他們時其限定符必須同為 *__global*、*__constant* 或 *__local*。
   - 不允許指針指向函式。
   - 在程式中，內核函式的參數不能聲明為指針的指針。但是函式內部的變量或者非內核函式的參數可以是指針的指針。

b. 只有函式參數的型別才能是圖像（image2d_t, image3d_t, image2d_array_t, image1d_t, image1d_buffer_t or image1d_array_t）。不能修改這種參數。並且只能用內建函式來訪問其元素（參見節 *6.12.14*）。

   圖像型別不能用來聲明變量、結構體或聯合體的欄位，也不能聲明圖像陣列或圖像指針，函式返回值的型別也不能是圖像。圖像型別不能與位址空間限定符 *__private*、*__local* 和 *__constant* 一起使用。型別 image3d_t 不能與訪問限定符 *__write_only* 一起使用，除非使能了擴展 **cl_khr_3d_image_writes**。圖像型別不能與訪問限定符 *__read_write* 一起使用，這種用法暫時保留，將來可能會使用。

   採樣器型別（sampler_t）只能作為函式參數的型別，或者用來聲明作用域為整個程式或內核函式最外層的變量。如果內核函式中所聲明的採樣器變量沒有位於最外層，則其行為依賴於具體實作。如果引數或變量是採樣器，則不能修改他。

   採樣器型別不能用來聲明結構體或聯合體的欄位，也不能聲明採樣器陣列或採樣器指針，函式返回值的型別也不能是採樣器。採樣器型別不能與位址空間限定符 *__local* 或 *__global* 一起使用。

c. 不支持結構體中的位欄（bit-field）成員。

d. 不支持變長陣列和帶有彈性（沒有指明大小的）陣列的結構體。

e. 不支持變參巨集和變參函式。

f. 程式中不能包含下列 C99 標準頭檔，也不能使用其中的庫函式:
   - 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、

[31] 加了刪除線的項是 OpenCL 1.0 中的限制，但在 OpenCL 1.1 中已經移除。

- stdlib.h、
- string.h、
- tgmath.h、
- time.h、
- wchar.h
- wctype.h。

g. 不支持存儲類別限定符 *auto* 和 *register*。

h. ~~不支持預定義標識符。~~

i. 不支持遞迴（recursion）。

j. 源碼中帶有限定符 *__kernel* 的函式所返回的型別只能是 void。

k. 內核函式參數的型別不能是內建標量型別 bool、half、size_t、ptrdiff_t、intptr_t 和 uintptr_t，如果是結構體或聯合體，其中的欄位也不能是這些標量型別。這些型別的大小（除了 half）依賴於具體實作，而且在 OpenCL 设备和主機處理器上也可能不同，這樣如果內核參數聲明為這些型別的指針，就很難為這樣的參數分配緩衝對象。這裡不支持 half，是因為他僅僅是一種存儲格式[32]，不能作為一種數據型別來參與算術運算。

l. 不可規約的控制流（irreducible control flow）是否違規依賴於具體實作。

m. ~~對於大小小於 32 位的內建型別，即 char、uchar、char2、uchar2、short、ushort 和 half，有如下限制：~~

- ~~不能對 char、uchar、char2、uchar2、short、ushort 和 half 的指針（或陣列）或者型別為 char、uchar、char2、uchar2、short 或 ushort 的結構體成員進行寫操作。參見節 9.9。~~

  ~~下面的內核示例中展示了少於 32 位的內建型別不支持哪些內存操作。~~

```
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.
}
```

n. 程式中內核函式引數的型別不能是 event_t。

o. 結構體或聯合體的元素必須屬於同一個位址空間中，否則違規。

p. 如果內核函式的引數是結構體或聯合體，其元素不能是 OpenCL 對象。

q. 支持 C99 中定義的型別限定符 *const*、*restrict* 和 *volatile*。這些限定符不能與下列型別一起使用：image2d_t、image3d_t、image2d_array_t、image1d_t、image1d_buffer_t、image1d_array_t。非指針型別不能使用限定符 *restrict*。

r. 內核函式引數的型別不能是事件（event_t）。事件型別不能用來聲明作用域為整個程式的變量。事件型別也不能用來聲明結構體或聯合體的欄位。事件型別不能與位址空間限定符 *__local*、*__constant* 或 *__global* 一起使用。

节 6.10 預處理指示和巨集

支持 C99 規範中定義的預處理指示。

其中是這樣描述 **# pragma** 指示的：

```
# pragma pp-tokens_opt new-line
```

[32] 除非支持擴展 **cl_khr_fp16**。

在進行巨集代換（macro replacement）之前，如果在指令 **# pragma** 中，**pragma** 後面緊跟的不是預處理符記（token）**OpenCL**（用來取代 **STDC**），則其行為依賴於具體實作。此行為可能會導致翻譯失敗，也可能導致翻譯器或最終程式的行為不符合規範。如果實作不能識別這樣的 **pragma**，則將其忽略。在進行巨集代換之前，如果 **pragma** 後面緊跟的是預處理符記 **OpenCL**，則不會在此指示之上實施任何巨集代換，並且此指示只能是下列形式之一 (其意義在別處有描述)：

```
#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
```

可用的預定義巨集的名字如下所示：

__FILE__ 當前源檔的名字（常值字串）。

__LINE__ 當前源檔中當前源碼行的行號（整形常數）。

__OPENCL_VERSION__ 一個整數，用來反映 OpenCL 设备所支持的 OpenCL 規範的版本號。如果是本文所述的 OpenCL，則 __OPENCL_VERSION__ 所對應的整數是 120。

CL_VERSION_1_0 整數 100，反映的是 OpenCL 1.0 規範。

CL_VERSION_1_1 整數 110，反映的是 OpenCL 1.1 規範。

CL_VERSION_1_2 整數 120，反映的是 OpenCL 1.2 規範。

__OPENCL_C_VERSION__ 一個整數，用來反映 OpenCL C 的版本，此版本號由 **clBuildProgram** 或 **clCompileProgram** 所用構建選項 -cl-std 來指定。如果沒有指定此構建選項，則使用這個 OpenCL 设备所用編譯器所支持的 OpenCL C 的版本。如果是本文所述的 OpenCL C，則 __OPENCL_C_VERSION__ 所對應的整數是 120。

__ENDIAN_LITTLE__ 用來確定 OpenCL 设备的架構是大端的還是小端的（如果是小端的，則為 1，否則未定義）。參見表 *4.3*中的 CL_DEVICE_ENDIAN_LITTLE。

__kernel_exec(X, type*n*)（和 kernel_exec(X, type*n*)）定義為：

```
__kernel __attribute__((work_group_size_hint(X, 1, 1))) \
         __attribute__((vec_type_hint(typen)))
```

__IMAGE_SUPPORT__ 用來確定 OpenCL 设备是否支持圖像。如果支持圖像則為整形常數 1，否則未定義。參見表 *4.3*中的 CL_DEVICE_IMAGE_SUPPORT。

__FAST_RELAXED_MATH__ 用來確定給 **clBuildProgram** 或 **clCompileProgram** 所指定的構建選項中是否有優化選項 -cl-fast-relaxed-math。如果有則為整形常數 1，否則未定義。

對於那些 C99 規範中有定義，但目前 OpenCL 還不支持的巨集名字，全部保留，以備將來使用。

預定義的標識符 __func__ 可用。

节 6.11 特性限定符

本節描述 __*attribute*__ 的相關語法以及他所綁定的構件。

特性限定符的形式為 __attribute__ ((attribute-list))。

特性列定義如下：

```
attribute-list:
        attribute_opt
        attribute-list , attribute_opt

attribute:
        attribute-token attribute-argument-clause_opt

attribute-token:
        identifier

attribute-argument-clause:
        ( attribute-argument-list )
```

```
attribute-argument-list:
        attribute-argument
        attribute-argument-list, attribute-argument

attribute-argument:
        assignment-expression
```

此語法直接取自 GCC，但跟 GCC 又有不同。在 GCC 中，特性只能用在函式、型別和變量上，而 OpenCL 特性可與下列項相關聯:

- 型別
- 函式
- 變量
- 區塊
- 控制流語句

通常，對於在給定的上下文中如何綁定特性，其規則有很多值得仔細研究的地方，關於細節讀者可以參考 GCC 的文檔[5]以及 Maurer 和 Wong 的論文[14]。

6.11.1 用於型別的特性

定義 enum、struct 和 union 型別時，可以用關鍵字 *__atrribute__* 為其指定某些特性。此關鍵字後緊跟用雙層括號括起來的特性規格。目前型別有兩種特性: aligned 和 packed。

在聲明或定義 enum、struct 或 union 時，或者 typedef 其他型別時都可以指定型別特性。

對於 enum、struct 和 union 型別，特性限定符可以位於標籤 enum、struct 或 union 和型別名字中間，也可以跟在右大括號之後。優選前者。

aligned (*alignment*)

此特性用來指定某種型別變量的最小對齊字節數。

```
struct S { short f[3]; } __attribute__ ((aligned (8)));
typedef int more_aligned_int __attribute__ ((aligned (8)));
```

迫使編譯器保證（盡其所能）在分配型別為 struct S 或 more_aligned_int 的變量時**至少**按 8 字節邊界進行對齊。

譯者注: 是已經作此要求，還是正在考慮作此要求？

需要注意的是，ISO C 標準要求任一給定的 struct 或 union 型別的齊位都至少要是其所有成員齊位（alignment）的最小公倍數的整數倍，同時必須是二的冪。這意味着對於 struct 或 union 型別，你**可以**通過給其任一成員附加特性 aligned 來有效的調整整個 struct 或 union 的齊位。不過如果要想調整整個 struct 或 union 型別的齊位，上面示例所演示的方式無疑更明顯、更直觀、可讀性更好。

如同前例所示，你可以為某個 struct 或 union 型別顯式指定你期望編譯器使用的齊位字節數。另外，你也可以不指定齊位因子，讓編譯器按目標機器上所用過的最大齊位進行對齊。例如，你可以這樣寫:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

無論何時，只要你在特性 *aligned* 的規格描述中沒有指定齊位因子，則編譯器自行選擇在目標機器上曾經使用過的齊位中最大的那個。上例中，每個 short 大小為 2 字節，整個 struct S 為 6 字節。 2 的冪中，大於等於 6 的最小的那個是 8，因此編譯器將 struct S 的齊位設置為 8。

注意，特性 *aligned* 的有效性可能受限於 OpenCL 设备和編譯器的固有限制。編譯器所支持的齊位可能有一個上限。如果編譯器支持最大為 8 字節的齊位，即使在 *__attribute__* 中指定了 aligned(16)，也只能按 8 字節進行對齊。請查閱您所用平台的文檔以獲取進一步的信息。

特性 *aligned* 只能增大齊位；但你可以用 *packed* 來減小齊位。看下面。

packed

此特性可在定義 struct 或 union 時使用，以指定結構體或聯合體的每個成員如何放置，從而使佔用的內存最少。當用在 enum 定義上時，則表明應當使用的最小的整數型別。

為某個 struct 或 union 指定此特性，等同於為其所有成員都指定此特性。

下面例子中，struct my_packed_struct 的成員都壓縮到了一起，但其成員 s 的內部布局沒有壓縮。要想壓縮 s，則 struct my_unpacked_struct 也要使用特性 *packed*。

```
1  struct my_unpacked_struct
2  {
3          char c;
4          int i;
5  };
6
7  struct __attribute__ ((packed)) my_packed_struct
8  {
9          char c;
10         int i;
11         struct my_unpacked_struct s;
12 };
```

你只需在定義 enum、struct 或 union 時使用 *packed*，而在 typedef 時則無需使用，因為 typedef 並沒有定義新的枚舉、結構體或聯合體。

6.11.2 用於函式的特性

當前所支持的函式特性限定符請參見節 *6.7*。

6.11.3 用於變量的特性

關鍵字 *__attribute__* 也可用來為變量或結構體的欄位指定特性。此關鍵字後緊跟用雙層括號括起來的特性規格。目前定義了下列特性限定符：

aligned (*alignment*)

此特性用來指定變量或結構體欄位的最小齊位，單位：字節。例如，下列聲明：

```
1  int x __attribute__ ((aligned (16))) = 0;
```

使編譯器將全局變量 x 分配在 16 字節邊界上。所指定的齊位值必須是二的冪。

你也可以為結構體的欄位指定齊位。例如，要想創建按雙字對齊的 int 對組，你可以這樣寫：

```
1  struct foo { int x[2] __attribute__ ((aligned (8))); };
```

除了這種方式，你也可以創建具有 double 成員的聯合體，從而迫使聯合體按雙字對齊。

如同前例所示，你可以為某個變量或結構體欄位顯式指定你期望編譯器使用的齊位字節數。另外，你也可以不指定齊位因子，讓編譯器按目標機器上所用過的最大齊位進行對齊。例如，你可以這樣寫：

```
1  short array[3] __attribute__ ((aligned));
```

無論何時，只要你在特性 *aligned* 的規格描述中沒有指定齊位因子，則編譯器自行選擇在目標機器上曾經使用過的齊位中最大的那個。

當用在 struct 或結構體成員上時，特性 *aligned* 只能增大齊位；要想減小齊位，只能使用 *packed*。而用在 typedef 上時，特性 *aligned* 既能增大齊位，也能減小齊位，如果使用特性 *packed* 則會產生警告。

注意，特性 *aligned* 的有效性可能受限於 OpenCL 设备和編譯器的固有限制。編譯器所支持的齊位可能有一個上限。如果編譯器支持最大為 8 字節的齊位，即使在 *__attribute__* 中指定了 aligned(16)，也只能按 8 字節進行對齊。請查閱您所用平台的文檔以獲取進一步的信息。

packed

此特性用在變量或結構體欄位上時，會使其按最小齊位——單字節進行對齊，除非你用特性 *aligned* 指定了更大的值。

這裡有一個結構體，其欄位 x 上用了 *packed*，因此它會緊跟在 a 後面：

```
struct foo
{
        char a;
        int x[2] __attribute__ ((packed));
};
```

對於用戶自定義的型別，如果特性列位於他的開始，則會作用於此型別的變量上，而不會作用於此型別上；而如果特性列位於型別本體之後，則會作用於此型別上。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* 決定了變量的字節順序。可將 *endiantype* 設置為 `host`，表明使用主機處理器的端序；也可將其設置為 `device`，表明使用用以執行內核的设备的端序。缺省值為 `device`。

例如：

```
float4 *p __attribute__ ((endian(host)));
```

指明 p 所指內存中存儲的數據格式為主機端序。

6.11.4 用於區塊和控制流語句的特性

正在考慮在基本區塊和控制流語句的前面放置特性，例如：

```
__attribute__((attr1)) {...}

for __attribute__((attr2)) (...) __attribute__((attr3)) {...}
```

這裡，`attr1` 作用於大括號中的區塊上，而 `attr2` 和 `attr3` 則分別作用於迴圈的控制構件以及本體上。

對於區塊和控制流語句，目前還沒有定義任何特性限定符。

6.11.5 對特性限定符的擴展

可以為標準語言擴展和特定供應商的擴展對特性語法做一些擴充。任何擴展都要遵守《OpenCL 1.2 擴展規範》的第九章所列的命名約定。

對於編譯器而言，特性提供了非常有用的暗示。我們認為，OpenCL 的實作可以無視所有特性，但所生成的可執行二元碼必須能產生同樣的結果。這並不妨礙實作使用特性所提供的附加資訊以實施優化或者其他轉化，只要他認為合適就行。這種情況下，程式員就有責任保證所提供的資訊在某種意義上是正確的。

节 6.12 內建函式

OpenCL C 編程語言提供了一套豐富的內建函式用於標量和矢量運算。這些函式中有很多跟通用 C 庫中提供的函式名字類似，不同的是他們所支持的參數型別即可是標量，也可是矢量。應用應儘可能使用這些內建函式，而不是去實現自己的版本。

用戶自定義的 OpenCL C 函式，按照函式的 C 標準規則運行（C99-TC2-節 *6.9.1*）。在函式入口處，對於所有可動態修改的參數（variably modified parameter），都會求出其大小，並且按照節 *6.2.6*中描述的常見算術轉換規則將所有引數算式的值都轉換成對應參數的型別。本節所描述的

內建函式行為類似，不過由於同一個內建函式可能有多種形式，為避免歧義，不會有隱式的標量拓寬。然而需要注意的是，可能某些內建函式的某些形式是針對標量以及矢量型別的混合運算。

6.12.1 作業項函式

表 6.8列出了內建的作業項函式，這些函式可用來查詢指定給 **clEnqueueNDRangeKernel** 的維數、全局和局部的索引空間大小、以及在设备上執行此內核時每個作業項的全局 ID和局部 ID。如果使用的是 **clEnqueueTask**，則維數、全局和局部索引空間的大小都是一。

表 6.8 作 業 項 函 式 表

函式	描述
`uint get_work_dim()`	返回所用的維度數目。即 **clEnqueueNDRangeKernel** 的引數 *work_dim* 的值。 如果用的是 **clEnqueueTask**，則返回 1。
`size_t get_global_size(uint dimindx)`	返回在 *dimindx* 所標識的維度上指定的全局作業項的數目。即 **clEnqueueNDRangeKernel** 的引數 *global_work_size*。*dimindx* 的有效範圍為 0 到 **get_work_dim**() − 1。如果 *dimindx* 是其他值，則 **get_global_size** 返回 1。 如果用的是 **clEnqueueTask**，則返回 1。
`size_t get_global_id(uint dimindx)`	返回作業項在 *dimindx* 所標識的維度上的唯一全局 ID。此 ID 基於用來執行此內核的全局作業項的數目。 *dimindx* 的有效範圍為 0 到 **get_work_dim**() − 1。如果 *dimindx* 是其他值，則 **get_global_id** 返回 0。 如果用的是 **clEnqueueTask**，則返回 0。
`size_t get_local_size(uint dimindx)`	返回在 *dimindx* 所標識的維度上指定的局部作業項的數目。如果 **clEnqueueNDRangeKernel** 的引數 *local_work_size* 不是 NULL，則返回的就是此引數的值；否則 OpenCL 實作會選擇一個恰當的 *local_work_size* 並將其返回。 *dimindx* 的有效範圍為 0 到 **get_work_dim**() − 1。如果 *dimindx* 是其他值，則 **get_local_size** 返回 1。 如果用的是 **clEnqueueTask**，則返回 1。
`size_t get_local_id(uint dimindx)`	返回作業項在作業組內 *dimindx* 所標識的維度上的唯一局部 ID。 *dimindx* 的有效範圍為 0 到 **get_work_dim**() − 1。如果 *dimindx* 是其他值，則 **get_local_id** 返回 0。 如果用的是 **clEnqueueTask**，則返回 0。
`size_t get_num_groups(uint dimindx)`	返回執行此內核的作業組在 *dimindx* 所標識的維度上的數目。*dimindx* 的有效範圍為 0 到 **get_work_dim**() − 1。如果 *dimindx* 是其他值，則 **get_num_groups** 返回 1。 如果用的是 **clEnqueueTask**，則返回 1。
`size_t get_group_id(uint dimindx)`	所返回的作業組 ID 取值範圍為 0...**get_work_dim**(*dimindx*) − 1。 *dimindx* 的有效範圍為 0 到 **get_work_dim**() − 1。如果 *dimindx* 是其他值，則 **get_group_id** 返回 0。 如果用的是 **clEnqueueTask**，則返回 0。
`size_t get_global_offset (uint dimindx)`	返回的是為 **clEnqueueNDRangeKernel** 的引數 *global_work_offset* 所指定的偏移值。 *dimindx* 的有效範圍為 0 到 **get_work_dim**() − 1。如果 *dimindx* 是其他值，則 **get_group_id** 返回 0。 如果用的是 **clEnqueueTask**，則返回 0。

6.12.2 數學函式

表 6.9中列出了內建的數學函式。內建的數學函式分為兩種：

- 第一種函式有兩個版本，一個版本的引數是標量，一個版本的引數是矢量；
- 第二種函式只有一個版本，引數為標量浮點數。

矢量版本的數學函式按組件逐一進行運算。描述也是針對單個組件的。

無論調用環境中使用哪種捨入模式，內建數學函式始終捨入為最近偶數，返回的結果也始終如一。

表 6.9中所列函式即可接受標量引數，也可接受矢量引數。泛型 gentype 表示函式引數的型別可以是 float、float2、float3、float4、float8、float16、double、double2、double3、double4、double8 或 double16。泛型 gentypef 表示函式引數的型別可以是 float、float2、float3、float4、float8 或 float16。泛型 gentyped 表示函式引數的型別可以是 double、double2、double3、double4、double8 或 double16。如果沒有特殊說明，函式的返回值與引數的型別都相同。

表 6.9α　引數既可為標量，也可為矢量的內建數學函式表

函式	描述
gentype **acos**(gentype)	反餘弦函數。
gentype **acosh**(gentype)	反雙曲餘弦函數。
gentype **acospi**(gentype *x*)	計算 $acos(x)/\pi$。
gentype **asin**(gentype)	反正弦函數。
gentype **asinh**(gentype)	反雙曲正弦函數。
gentype **asinpi**(gentype *x*)	計算 $asin(x)/\pi$。
gentype **atan** (gentype *y_over_x*)	反正切函數。
gentype **atan2** (gentype *y*, gentype *x*)	y/x 的反正切。
gentype **atanh** (gentype)	反雙曲正切函數。
gentype **atanpi** (gentype *x*)	計算 $atan(x)/\pi$。
gentype **atan2pi** (gentype *y*, gentype *x*)	計算 $atan2(y,x)/\pi$。
gentype **cbrt**(gentype)	計算立方根。
gentype **ceil**(gentype)	向正無窮捨入成整數值。
gentype **copysign**(gentype *x*, gentype *y*)	將 *x* 的符號改成 *y* 的，並將其返回。
gentype **cos**(gentype)	計算餘弦。
gentype **cosh**(gentype)	計算雙曲餘弦。
gentype **cospi**(gentype *x*)	計算 $cos(\pi x)$。
gentype **erfc**(gentype *x*)	餘補誤差函數 $1 - \mathbf{erf}(x) = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-\theta^2}d\theta$。
gentype **erf**(gentype *x*)	誤差函數，表示正態分布的積分 $\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-\theta^2}d\theta$。
gentype **exp**(gentype *x*)	計算 e 的 *x* 次冪 e^x。
gentype **exp2**(gentype)	底數為 2 的冪。
gentype **exp10**(gentype)	底數為 10 的冪。
gentype **expm1**(gentype *x*)	計算 $e^x - 1.0$。
gentype **fabs**(gentype)	計算浮點數的絕對值。
gentype **fdim**(gentype *x*, gentype *y*)	如果 $x < y$，返回 $x - y$；否則返回 $+0$。
gentype **floor**(gentype)	向負無窮捨入成整數值。
gentype **fma**(gentype *a*, gentype *b*, gentype *c*)	返回 $a \times b + c$，其中乘法具有無限精度即不會進行捨入，但會對加法進行正確地捨入。邊界條件下的行為遵循 IEEE 754-2008 標準。
gentype **fmax** (gentype *x*, gentype *y*) gentypef **fmax** (gentypef *x*, float *y*) gentyped **fmax** (gentyped *x*, double *y*)	如果 $x < y$，則返回 *y*，否則返回 *x*。如果一個引數是 NaN，則返回另一個引數；如果兩個引數都是 NaN，則返回 NaN。
gentype **fmin** (gentype *x*, gentype *y*) gentypef **fmin** (gentypef *x*, float *y*) gentyped **fmin** (gentyped *x*, double *y*)	如果 $y < x$，則返回 *y*，否則返回 *x*。如果一個引數是 NaN，則返回另一個引數；如果兩個引數都是 NaN，則返回 NaN[1]。
gentype **fmod** (gentyped *x*, gentype *y*)	模。返回 $x - y \times \mathbf{trunc}(x/y)$。

表 6.9β 引數既可為標量，也可為矢量的內建數學函式表

`gentype fract (gentype x, __global gentype *iptr)`
`gentype fract (gentype x, __local gentype *iptr)`
`gentype fract (gentype x, __private gentype *iptr)`
	返回 **fmin**$(x - \textbf{floor}(x), \texttt{0x1.fffffep - 1f})$。而 **floor**$(x)$ 將在 *iptr* 中返回[2]。

`floatn frexp (floatn x, __global intn *exp)`
`floatn frexp (floatn x, __local intn *exp)`
`floatn frexp (floatn x, __private intn *exp)`
`float frexp (float x, __global int *exp)`
`float frexp (float x, __local int *exp)`
`float frexp (float x, __private int *exp)`
	從 x 中分離出尾數和指數。返回的尾數（記為 m)型別為 `float`，值為 0 或者屬於 $[1/2, 1)$。x 的每個組件都等於 $m \times 2^{\text{exp}}$。

`doublen frexp (doublen x, __global intn *exp)`
`doublen frexp (doublen x, __local intn *exp)`
`doublen frexp (doublen x, __private intn *exp)`
`double frexp (double x, __global int *exp)`
`double frexp (double x, __local int *exp)`
`double frexp (double x, __private int *exp)`
	從 x 中分離出尾數和指數。返回的尾數（記為 m)型別為 `double`，值為 0 或者屬於 $[1/2, 1)$。x 的每個組件都等於 $m \times 2^{\text{exp}}$。

`gentype hypot (gentype x, gentype y)`	計算 $\sqrt{x^2 + y^2}$，不會有過分的上溢或下溢。

`intn ilogb (floatn x)`
`int ilogb (float x)`
`intn ilogb (doublen x)`
`int ilogb (double x)`
	返回整形對數。

`floatn ldexp (floatn x, intn k)`
`floatn ldexp (floatn x, int k)`
`float ldexp (float x, int k)`
`doublen ldexp (doublen x, intn k)`
`doublen ldexp (doublen x, int k)`
`double ldexp (double x, int k)`
	返回 $x \times 2^{\text{k}}$。

表 6.9γ 引數既可為標量，也可為矢量的內建數學函式表

gentype **lgamma** (gentype *x*) floatn **lgamma_r** (floatn *x*, __global intn *signp) floatn **lgamma_r** (floatn *x*, __local intn *signp) floatn **lgamma_r** (floatn *x*, __private intn *signp) float **lgamma_r** (float *x*, __global int *signp) float **lgamma_r** (float *x*, __local int *signp) float **lgamma_r** (float *x*, __private int *signp) doublen **lgamma_r** (doublen *x*, __global intn *signp) doublen **lgamma_r** (doublen *x*, __local intn *signp) doublen **lgamma_r** (doublen *x*, __private intn *signp) double **lgamma_r** (double *x*, __global int *signp) double **lgamma_r** (double *x*, __local int *signp) double **lgamma_r** (double *x*, __private int *signp)	返回伽馬函數絕對值的自然對數。**lgamma_r** 的引數 *signp* 中會返回伽馬函數的符號。$ln\|\Gamma(x)\|$
gentype **log** (gentype)	計算自然對數。
gentype **log2** (gentype)	計算以 2 為底的對數。
gentype **log10** (gentype)	計算以 10 為底的對數。
gentype **log1p** (gentype *x*)	計算 $log_e(1.0 + x)$。
gentype **logb** (gentype *x*)	$log_r\|x\|$的整數部分。
gentype **mad** (gentype *a*, gentype *b*, gentype *c*)	**mad** 逼近 $a \times b + c$。至於 $a \times b$ 是否要捨入、怎樣捨入，以及如何處理超常（supernormal）或次常（subnormal）的乘法都沒有定義。在那些速度比準確更重要的地方，就可以使用 **mad**[3]。
gentype **maxmag** (gentype *x*, gentype *y*)	如果 $\|x\| > \|y\|$，則返回 *x*；如果 $\|y\| > \|x\|$，則返回 *y*；否則返回 $\mathsf{fmax}(x, y)$。
gentype **minmag** (gentype *x*, gentype *y*)	如果 $\|x\| < \|y\|$，則返回 *x*；如果 $\|y\| < \|x\|$，則返回 *y*；否則返回 $\mathsf{fmin}(x, y)$。
gentype **modf** (gentype *x*, __global gentype *iptr) gentype **modf** (gentype *x*, __local gentype *iptr) gentype **modf** (gentype *x*, __private gentype *iptr)	分解浮點數，將引數 *x* 分解為整數部分和小數部分，兩部分的符號都與 *x* 相同。整數部分存儲在 *iptr* 所指對象中。
floatn **nan** (uintn *nancode*) float **nan** (uint *nancode*) doublen **nan** (ulongn *nancode*) double **nan** (ulong *nancode*)	返回 quiet NaN。其中將 *nancode* 放在尾數的位置。
gentype **nextafter** (gentype *x*, gentype *y*)	返回 *x* 在往 *y* 的方向上下一個可表示的單精度浮點值。因此，如果 *y* 小於 *x*，則返回小於 *x* 的最大的可表示的浮點數。
gentype **pow** (gentype *x*, gentype *y*)	計算 x^y。
floatn **pown** (floatn *x*, intn *y*) float **pown** (float *x*, int *y*) doublen **pown** (doublen *x*, intn *y*) double **pown** (double *x*, int *y*)	計算 x^y，其中 *y* 是整數。

表 6.96 引數既可為標量，也可為矢量的內建數學函式表

`gentype powr (gentype x, gentype y)`	計算 x^y，其中 $x \geq 0$。
`gentype remainder (gentype x, gentype y)`	返回值記為 r，則 $r = x - n \times y$。其中 n 是最接近精確值 x/y 的整數。如果有兩個這樣的整數，則選擇偶數作為 n。如果 r 是零，則符號與 x 一樣。

`floatn remquo (floatn x, floatn y, __global intn *quo)`
`floatn remquo (floatn x, floatn y, __local intn *quo)`
`floatn remquo (floatn x, floatn y, __private intn *quo)`
`float remquo (float x, float y, __global int *quo)`
`float remquo (float x, float y, __local int *quo)`
`float remquo (float x, float y, __private int *quo)`
	返回值記為 r，則 $r = x - k \times y$。其中 k 是最接近精確值 x/y 的整數。如果有兩個這樣的整數，則選擇偶數作為 k。如果 r 是零，則符號與 x 一樣。r 跟 **remainder** 返回的值一樣。區別就是 **remquo** 還會計算整數商 x/y 的最低七位，連帶 x/y 的符號一同存儲在 *quo* 所指對象中。

`doublen remquo (doublen x, doublen y, __global intn *quo)`
`doublen remquo (doublen x, doublen y, __local intn *quo)`
`doublen remquo (doublen x, doublen y, __private intn *quo)`
`double remquo (double x, double y, __global int *quo)`
`double remquo (double x, double y, __local int *quo)`
`double remquo (double x, double y, __private int *quo)`
	返回值記為 r，則 $r = x - k \times y$。其中 k 是最接近精確值 x/y 的整數。如果有兩個這樣的整數，則選擇偶數作為 k。如果 r 是零，則符號與 x 一樣。r 跟 **remainder** 返回的值一樣。區別就是 **remquo** 還會計算整數商 x/y 的最低七位，連帶 x/y 的符號一同存儲在 *quo* 所指對象中。

`gentype rint (gentype)`	捨入為最近偶數，雖然值為整數，但用的是浮點格式。關於捨入模式請參考節 7.1。

`floatn rootn (floatn x, intn y)`
`float rootn (float x, int y)`
 	
`doublen rootn (doublen x, intn y)`
`doublen rootn (double x, int y)`
	計算 $x^{1/y}$。

`gentype rsqrt (gentype)`	計算 $1/\sqrt{x}$。
`gentype sin (gentype)`	計算正弦。

`gentype sincos (gentype x, __global gentype *cosval)`
`gentype sincos (gentype x, __local gentype *cosval)`
`gentype sincos (gentype x, __private gentype *cosval)`
	計算 x 的正弦和餘弦。返回的是正弦，餘弦放到 *cosval* 中。

表 6.9ε　引數既可為標量，也可為矢量的內建數學函式表

gentype **sinh** (gentype)	計算雙曲正弦。
gentype **sinpi** (gentype *x*)	計算 **sin**(πx)。
gentype **sqrt** (gentype)	計算平方根。
gentype **tan** (gentype)	計算正切。
gentype **tanh** (gentype)	計算雙曲正切。
gentype **tanpi** (gentype)	計算 **tan**(πx)。
gentype **tgamma** (gentype)	計算 $\Gamma(x)$。
gentype **trunc** (gentype)	向零捨入成整數值。

[1] 在處理 sNaN（signaling NaN）時，**fmin** 和 **fmax** 的行為遵守 C99 中的定義，但可能與 IEEE 754-2008 中定義的 **minNum** 和 **maxNum** 處理方式不同。特別是，可能將 sNaN 當成 qNaN（quiet NaN）。

[2] 此處的 **min**() 是為了避免 **fract**(-small) 返回 1.0。有了 **min**()，這種情況就會返回小於 1.0 的最大正浮點數。

[3] 需要提醒用戶的是，對於一些情況，如 **mad**$(a, b, -a \times b)$，**mad** 定義的非常寬鬆，以至於當 *a* 和 *b* 為某些特定值時，返回任何值都有可能

*表 6.10*中列出了下列函式：

- *表 6.9*中的部分函式，但定義時帶有前綴 `half_`。實現這些函式時，精度至少要有 10 位，即所有 ULP 值都要小於等於 8192 ulp（ULP: units in the last place，最後一位的進退位）。
- *表 6.9*中的部分函式，但定義時帶有前綴 `native_`。這些函式可能會映射到一條或多條原生的设备指令上，性能通常比對應的不帶前綴 `native_` 的函式更好。這些函式的精度（以及某些情況下的輸入範圍）依賴於具體實作。
- 用於除法和倒數運算的 `half_` 和 `native_` 函式。

在*表 6.10*中，泛型 `gentype` 表示函式引數的型別可以是 `float`、`float2`、`float3`、`float4`、`float8` 或 `float16`。

表 6.10α　內建的 `half_` 和 `native_` 數學函式

函式	描述
gentype **half_cos** (gentype *x*)	計算餘弦。*x* 的取值範圍為 $-2^{16} \ldots +2^{16}$。
gentype **half_divide** (gentype *x*, gentype *y*)	計算 x/y。
gentype **half_exp** (gentype *x*)	計算 e^x。
gentype **half_exp2** (gentype *x*)	計算 2^x。
gentype **half_exp10** (gentype *x*)	計算 10^x。
gentype **half_log** (gentype *x*)	計算自然對數。
gentype **half_log2** (gentype *x*)	計算底為 2 的對數。
gentype **half_log10** (gentype *x*)	計算底為 10 的對數。
gentype **half_powr** (gentype *x*, gentype *y*)	計算 x^y，其中 $x \geq 0$。
gentype **half_recip** (gentype *x*)	計算倒數。
gentype **half_rsqrt** (gentype *x*)	計算 $1/\sqrt{x}$。
gentype **half_sin** (gentype *x*)	計算正弦。*x* 的取值範圍為 $-2^{16} \ldots +2^{16}$。
gentype **half_sqrt** (gentype *x*)	計算 $\sqrt{x}$。
gentype **half_tan** (gentype *x*)	計算正切。*x* 的取值範圍為 $-2^{16} \ldots +2^{16}$。
gentype **native_cos** (gentype *x*)	計算餘弦。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_divide** (gentype *x*, gentype *y*)	計算 x/y。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_exp** (gentype *x*)	計算 e^x。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_exp2** (gentype *x*)	計算 2^x。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。

表 6.10β 內建的 half_ 和 native_ 數學函式

gentype **native_exp10** (gentype *x*)	計算 10^x。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_log** (gentype *x*)	計算自然對數。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_log2** (gentype *x*)	計算底為 2 的對數。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_log10** (gentype *x*)	計算底為 10 的對數。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_powr** (gentype *x*, gentype *y*)	計算 x^y，其中 $x \geq 0$。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_recip** (gentype *x*)	計算倒數。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_rsqrt** (gentype *x*)	計算 $1/\sqrt{x}$。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_sin** (gentype *x*)	計算正弦。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_sqrt** (gentype *x*)	計算 $\sqrt{x}$。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。
gentype **native_tan** (gentype *x*)	計算正切。參數的取值範圍以及取最大值時會產生什麼錯誤都依賴於具體實作。

實作可以自行決定 **half_** 函式是否支持去規格化值。如果引數是去規格化數，**half_** 函式可以返回任何值，只要節 *7.5.3*允許就行，無論 `-cl-denorms-are-zero`（參見節 *5.6.4.2*）是否有效。

有下列符號常量可用。這些值的型別都是 `float`，在單精度浮點數的精度內是精確的。

常量名	描述
MAXFLOAT	最大的有限單精度浮點數。
HUGE_VALF	正浮點常量算式。其求值結果為 $+\infty$，由內建數學函式用作返回值表明錯誤。
INFINITY	常量算式，型別為 `float`，表示正的或無符號的無窮。
NAN	常量算式，型別為 `float`，表示 quiet NaN。

如果设备支持雙精度浮點數，還有下列符號常量可用:

常量名	描述
HUGE_VAL	正浮點常量算式。型別為 `double`。其求值結果為 $+\infty$，由內建數學函式用作返回值表明錯誤。

6.12.2.1 浮點巨集和雜注

雜注（pragma）**FP_CONTRACT** 可用來允許（如果狀態是 `on`）或禁止（如果狀態是 `off`）實作化簡算式。他可位於外部聲明的外面，也可位於複合語句中的顯式聲明或語句的前面。當在外部聲明外面時，在遇到下一個 **FP_CONTRACT** 或者翻譯單元結束時就無效了。當在複合語句中時，在遇到下一個 **FP_CONTRACT**（包括嵌套的複合語句中的 **FP_CONTRACT**）或者複合語句結束時就無效了；在複合語句末尾處，會恢復成此語句之前的狀態。在其他任何上下文中使用此雜注，其行為都是未定義的。

這樣設置 **FP_CONTRACT**:

```
1 #pragma OPENCL FP_CONTRACT on-off-switch
2
3 on-off-switch is one of:
4         ON, OFF or DEFAULT.
5         The DEFAULT value is ON.
```

巨集 **FP_FAST_FMAF** 用來指明對於單精度浮點數，函式 **fma** 是否比直接編碼更快。如果定義了此巨集，則表明對 `float` 算元的乘、加運算，函式 **fma** 一般跟直接編碼一樣快，或者更快。

OpenCL C 編程語言定義了如下巨集，他們必須使用指定的值。可以在預處理指示 `#if` 中使用這些常量算式。

```
#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
```

表 *6.11*中給出了上面所列巨集與應用所用的巨集名字之間的對應關係。

表 6.11 單精度浮點巨集與應用所用巨集的對應關係

OpenCL 語言中的巨集	應用所用的巨集
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_EPSILSON	CL_FLT_EPSILSON

下列兩個巨集將會展開成整數常量算式。如果 x 是 0 或 NaN，則 **ilogb**(x) 會分別返回這兩個值。

- **FP_ILOGB0** 為 {INT_MIN} 或 -{INT_MAX}。
- **FP_ILOGBNAN** 為 {INT_MAX} 或 {INT_MIN}。

除此之外，還有下列常量可用。他們的型別都是 float，在 float 型別的精度內是精確的。

常量	描述
M_E_F	e
M_LOG2E_F	$log_2 e$
M_LOG10E_F	$log_{10} e$
M_LN2_F	$log_e 2$
M_LN10_F	$log_e 10$
M_PI_F	π
M_PI_2_F	$\pi/2$
M_PI_4_F	$\pi/4$
M_1_PI_F	$1/\pi$
M_2_PI_F	$2/\pi$
M_2_SQRTPI_F	$2/\sqrt{\pi}$
M_SQRT2_F	$\sqrt{2}$
M_SQRT1_2_F	$1/\sqrt{2}$

如果设备支持雙精度浮點數，還有下列巨集和常量可用:

- 巨集 **FP_FAST_FMA** 指明處理雙精度浮點數時，**fma** 系列函式是否比直接編碼更快。如果定義了此巨集，則表明對 double 算元的乘、加運算，函式 **fma** 一般跟直接編碼一樣快，或者更快。

OpenCL C 編程語言定義了如下巨集，他們必須使用指定的值。可以在預處理指示 #if 中使用這些常量算式。

```
#define DBL_DIG          15
#define DBL_MANT_DIG     53
#define DBL_MAX_10_EXP   +308
#define DBL_MAX_EXP      +1024
#define DBL_MIN_10_EXP   -307
#define DBL_MIN_EXP      -1021
#define DBL_MAX          0x1.fffffffffffffp1023
#define DBL_MIN          0x1.0p-1022
#define DBL_EPSILON      0x1.0p-52
```

表 *6.12*中給出了上面所列巨集與應用所用的巨集名字之間的對應關係。

表 6.12 雙精度浮點巨集與應用所用巨集的對應關係

OpenCL 語言中的巨集	應用所用的巨集
DBL_DIG	CL_DBL_DIG
DBL_MANT_DIG	CL_DBL_MANT_DIG
DBL_MAX_10_EXP	CL_DBL_MAX_10_EXP
DBL_MAX_EXP	CL_DBL_MAX_EXP
DBL_MIN_10_EXP	CL_DBL_MIN_10_EXP
DBL_MIN_EXP	CL_DBL_MIN_EXP
DBL_MAX	CL_DBL_MAX
DBL_MIN	CL_DBL_MIN
DBL_EPSILSON	CL_DBL_EPSILSON

除此之外，還有下列常量可用。他們的型別都是 double，在 double 型別的精度內是精確的。

常量	描述
M_E	e
M_LOG2E	$log_2 e$
M_LOG10E	$log_{10} e$
M_LN2	$log_e 2$
M_LN10	$log_e 10$
M_PI	π
M_PI_2	$\pi/2$
M_PI_4	$\pi/4$
M_1_PI	$1/\pi$
M_2_PI	$2/\pi$
M_2_SQRTPI	$2/\sqrt{\pi}$
M_SQRT2	$\sqrt{2}$
M_SQRT1_2	$1/\sqrt{2}$

6.12.3 整數函式

表 *6.13*中列出了內建的整數函式，其引數即可為標量，亦可為矢量。矢量版本的整數函式按組件逐一運算。其中的描述針對單個組件的。

泛型 gentype 表示函式引數的型別可以是 char、char{2|3|4|8|16}、uchar、uchar{2|3|4|8|16}、short、short{2|3|4|8|16}、ushort、ushort{2|3|4|8|16}、int、int{2|3|4|8|16}、uint、uint{2|3|4|8|16}、long、long{2|3|4|8|16}、ulong 或 ulong{2|3|4|8|16}。泛型 ugentype 指代無符號版本的 gentype。例如，如果 gentype 為 char4，則 ugentype 為 uchar4。同時，泛型 sgentype 指明函式的引數可以是標量（即 char、uchar、short、ushort、int、uint、long 或 ulong）。對於既有 gentype 引數，又有 sgentype 引數的內建整數函式，gentype 必須是標量或矢量版本的 sgentype。例如，如

果 sgentype 是 uchar，則 gentype 必須是 uchar 或 uchar{2|3|4|8|16}。對於矢量版本，sgentype 只是簡單的拓寬成 gentype，參見節 *6.3*中的第 *a*項。

對於任一函式的任一具體應用，所有引數以及返回值的型別均相同，除非明確指定了其型別。

表 6.13 引數既可為標量整數，也可為矢量整數的內建函式

函式	描述
ugentype **abs**(gentype *x*)	返回 $\|x\|$。
ugentype **abs_diff**(gentype *x*, gentype *y*)	返回 $\|x-y\|$，不會有模上溢。
gentype **add_sat**(gentype *x*, gentype *y*)	返回 $x+y$，並使結果飽和。
gentype **hadd** (gentype *x*, gentype *y*)	返回 $(x+y) >> 1$，中間的加法不會有模上溢。
gentype **rhadd** (gentype *x*, gentype *y*)	返回 $(x+y) >> 1$，中間的加法不會有模上溢[1]。
gentype **clamp** (gentype *x*, gentype *minval*, gentype *maxval*) gentype **clamp** (gentype *x*, sgentype *minval*, sgentype *maxval*)	返回 **min**(**max**(*x*, *minval*), *maxval*)。如果 *minval* > *maxval*，則結果未定義。
gentype **clz** (gentype *x*)	返回 *x* 中前導 0 的位數，從最高有效位開始。
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*，並使結果飽和。
gentype **max** (gentype *x*, gentype *y*) gentype **max** (gentype *x*, sgentype *y*)	如果 *x* < *y*，則返回 *y*，否則返回 *x*。
gentype **min** (gentype *x*, gentype *y*) gentype **min** (gentype *x*, sgentype *y*)	如果 *y* < *x*，則返回 *y*，否則返回 *x*。
gentype **mul_hi** (gentype *x*, gentype *y*)	計算 *x* × *y*，並返回乘積的高位半部（high half）。
gentype **rotate** (gentype *v*, gentype *i*)	將 *v* 中的每個元素都左移，其位數就是 *i* 中對應元素的值（遵守節 *6.3*中的移位取模規則）。由左側移出的位再從右側移入。
gentype **sub_sat** (gentype *x*, gentype *y*)	返回 *x* − *y*，並使結果飽和。
short **upsample** (char *hi*, uchar *lo*) ushort **upsample** (uchar *hi*, uchar *lo*) shortn **upsample** (charn *hi*, ucharn *lo*) ushortn **upsample** (ucharn *hi*, ucharn *lo*)	$result[i] = ((short)hi[i] << 8) \| lo[i]$ $result[i] = ((ushort)hi[i] << 8) \| lo[i]$
int **upsample** (short *hi*, ushort *lo*) uint **upsample** (ushort *hi*, ushort *lo*) intn **upsample** (shortn *hi*, ushortn *lo*) uintn **upsample** (ushortn *hi*, ushortn *lo*)	$result[i] = ((int)hi[i] << 16) \| lo[i]$ $result[i] = ((uint)hi[i] << 16) \| lo[i]$
long **upsample** (int *hi*, uint *lo*) ulong **upsample** (uint *hi*, uint *lo*) longn **upsample** (intn *hi*, uintn *lo*) ulongn **upsample** (uintn *hi*, uintn *lo*)	$result[i] = ((long)hi[i] << 32) \| lo[i]$ $result[i] = ((ulong)hi[i] << 32) \| lo[i]$
gentype **popcount** (gentype *x*)	返回 *x* 中非零位的數目。

[1] 矢量運算經常需要 $n+1$ 個臨時位才能算出結果。而 **rhadd** 指令具有這個額外的位，從而無需上升取樣（upsample）或下降取樣（downsample）。其性能優勢非常大。

表 6.14中列出了優化內核性能時可用的快速整數函式。泛型 gentype 表明函式引數的型別可以是 int、int2、int3、int4、int8、int16、uint、uint2、uint3、uint4、uint8 或 uint16。

表 6.14 內建的快速整數函式

函式	描述
gentype **mad24** (gentype *x*, gentype *y*, gentype *z*)	計算 $x \times y + z$，其中 x 和 y 均為 24 位整數，z 和返回值均為 32 位整數。至於如何實施 24 位整數乘法，請參考 **mul24** 的定義。
gentype **mul24** (gentype *x*, gentype *y*)	計算 x 和 y 的乘積。其中 x 和 y 均為 32 位整數，但僅使用低 24 位。z 和返回值均為 32 位整數。至於如何實施 24 位整數乘法，請參考 **mul24** 的定義。要使用此函式，必須滿足以下條件：如果 x 和 y 是帶符號整數，則其值必須在區間 $[-2^{23}, 2^{23} - 1]$ 內；如果 x 和 y 是無符號整數，則其值必須在區間 $[0, 2^{24} - 1]$ 內。如果不滿足上述條件，則其結果依賴於具體實作。

OpenCL C 編程語言定義了如下巨集，他們必須使用指定的值。可以在預處理指示 #if 中使用這些常量算式。

```
#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
```

下表給出了上面所列巨集與應用所用的巨集名字之間的對應關係。

OpenCL 語言中的巨集	應用所用的巨集
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
SHRT_MIN	CL_SHRT_MIN
UCHAR_MAX	CL_UCHAR_MAX
USHRT_MAX	CL_USHRT_MAX
UINT_MAX	CL_UINT_MAX
ULONG_MAX	CL_ULONG_MAX

6.12.4 公共函式[33]

表 *6.15*列出了内建的公共函式。這些函式都是按組件逐一運算的，其中的描述也是針對單個組件的。泛型 gentype 表明函式的引數可以是 float、float2、float3、float4、float8、float16、double、double2、double3、double4、double8 或 double16。泛型 gentypef 表明函式的引數可以是 float、float2、float3、float4、float8 或 float16。泛型 gentyped 表明函式的引數可以是 double、double2、double3、double4、double8 或 double16。

內建的公共函式實現時用的捨入模式是捨入為最近偶數。

表 6.15α　引數既可為標量整數，也可為矢量整數的內建公共函式

函式	描述
gentype **clamp** (gentype *x*, gentype *minval*, gentype *maxval*) gentypef **clamp** (gentypef *x*, float *minval*, float *maxval*) gentyped **clamp** (gentyped *x*, double *minval*, double *maxval*)	返回 **fmin**(**fmax**(*x*, *minval*), *maxval*)。如果 *minval* > *maxval*，則結果未定義。
gentype **degrees** (gentype *radians*)	將弧度轉換成角度，即 $(180/\pi) \times$ *radians*。
gentype **max** (gentype *x*, gentype *y*) gentypef **max** (gentypef *x*, float *y*) gentyped **max** (gentyped *x*, double *y*)	如果 $x < y$，則返回 *y*，否則返回 *x*。如果 *x* 或 *y* 是無窮或 NaN，則返回的值未定義。
gentype **min** (gentype *x*, gentype *y*) gentypef **min** (gentypef *x*, float *y*) gentyped **min** (gentyped *x*, double *y*)	如果 $y < x$，則返回 *y*，否則返回 *x*。如果 *x* 或 *y* 是無窮或 NaN，則返回的值未定義。
gentype **mix** (gentype *x*, gentype *y*, gentype *a*) gentypef **mix** (gentypef *x*, gentypef *y*, float *a*) gentyped **mix** (gentyped *x*, gentyped *y*, double *a*)	返回 $x + (y - x) \times a$。其中 *a* 必須在區間 0.0…1.0 內，否則返回值未定義。
gentype **radians** (gentype *degrees*)	將角度轉換成弧度，即 $(\pi/180) \times$ *degrees*。
gentype **step** (gentype *edge*, gentype *x*) gentypef **step** (float *edge*, gentypef *x*) gentyped **step** (double *edge*, gentyped *x*)	如果 $x <$ *edge*，則返回 0.0，否則返回 1.0。

gentype **smoothstep** (gentype *edge0*,
gentype *edge1*,
gentype *x*)
gentypef **smoothstep** (float *edge0*,
float *edge1*,
gentypef *x*)
gentyped **smoothstep** (double *edge0*,
double *edge1*,
gentyped *x*)
	如果 $x \le$ *edge0*，則返回 0.0；
	如果 $x \ge$ *edge1*，則返回 1.0；
	如果 *edge0* < *x* < *edge1*，則實施平滑埃爾米特插值（smooth Hermite interpolation）。在某些地方需要能進行平滑過渡的臨界函式，這時就可使用此函式。
	此函式等同於：
	`gentype t;`
	`t = clamp((x - edge0) / (edge1 - edge1), 0, 1);`
	`return t * t * (3 - 2 * t);`
	如果 *edge0* ≥ *edge1*，或者 *x*、*edge0*、*edge1* 中的任意一個是 NaN，則結果未定義。

[33] 可以使用化簡（如 **mad** 或 **fma**）來實現 **mix** 和 **smoothstep**。

表 6.15β 引數既可為標量整數，也可為矢量整數的內建公共函式

`gentype sign (gentype x)`	如果 $x > 0$，則返回 1.0；如果 $x = -0.0$，則返回 -0.0；如果 $x = +0.0$，則返回 $+0.0$；如果 $x < 0$，則返回 -1.0；如果 x 是 NaN，則返回 0.0。

6.12.5 幾何函式[34]

表 *6.16*中列出了內建的幾何函式。這些函式都是按組件逐一運算的，其中的描述也是針對單個組件的。`floatn` 表示 `float`、`float2`、`float3` 或 `float4`；而 `doublen` 則表示 `double`、`double2`、`double3` 或 `double4`。

內建的幾何函式實現時用的捨入模式是捨入為最近偶數。

表 6.16 引數既可為標量，也可為矢量的內建幾何函式

函式	描述

`float4 cross (float4 p0, float4 p1)`
`float3 cross (float3 p0, float3 p1)`
`double4 cross (double4 p0, double4 p1)`
`double3 cross (double3 p0, double3 p1)`
	返回 $p0.xyz$ 和 $p1.xyz$ 的叉乘。所返回的 `float4` 中的組件 w 為 0.0。

`float dot (floatn p0, floatn p1)` `double dot (doublen p0, doublen p1)`	計算點乘。
`float distance (floatn p0, floatn p1)` `double distance (doublen p0, doublen p1)`	返回 *p0* 和 *p1* 的距離。即 **length**$(p0 - p1)$。
`float length (floatn p)` `double length (gentype p)`	返回矢量 *p* 的長度，即 $\sqrt{p.x^2 + p.y^2 + \cdots}$。
`floatn normalize (floatn p)` `doublen normalize (doublen p)`	返回的矢量方向與 *p* 相同，長度為 1。
`float fast_distance (floatn p0, floatn p1)`	返回 **fast_length**$(p0 - p1)$。
`float fast_length (floatn p)`	返回 **half_sqrt**$(p.x^2 + p.y^2 + \cdots)$。

`floatn fast_normalize (floatn p)`	返回的矢量方向與 p 相同，長度為 1。**fast_normalize** 是這樣計算的：
	p * **half_rsqrt**$(p.x^2 + p.y^2 + \cdots)$
	返回值與下面語句結果（具有無窮精度）的誤差在 8192 ulp 內：
	`if (`**all**$(p == 0.0f)$`)`
	$result = p;$
	`else`
	$result = p/$**sqrt**$(p.x^2 + p.y^2 + \cdots);$
	但是下列情況例外：
	1. 如果平方和大於 `FLT_MAX`，則結果中對應的浮點值未定義。
	2. 如果平方和小於 `FLT_MIN`，則實作可能直接返回 p。
	3. 如果设备處於“將去規格化數刷成零”的模式，則在計算前會將算元中量級小於 **sqrt**(`FLT_MIN`) 的元素刷成零。

6.12.6 關係函式

可以使用關係算子和相等算子（`<`、`<=`、`>`、`>=`、`!=`、`==`）對內建標量和矢量型別進行關係運算，所產生的結果分別為標量或矢量帶符號整形，參見節 *6.3*。

表 *6.17*中所列函式[35]可以內建標量或矢量型別為引數，返回的結果為標量或矢量整形。泛型 `gentype` 指代下列內建型別：`char`、`charn`、`uchar`、`ucharn`、`short`、`shortn`、`ushort`、`ushortn`、`int`、`intn`、`uint`、`uintn`、`long`、`longn`、`ulong`、`ulongn`、`float`、`floatn`、

[34] 可以使用化簡（如 **mad** 或 **fma**）來實現幾何函式。

[35] 如果實作對規範進行了擴充，從而支持 IEEE-754 標志和異常，則當有一個或多個算數是 NaN 時，表 *6.17*中所定義的內建函式不會引發無效（*invalid*）浮點異常。

double 和 double*n*。泛型 igentype 指代内建带符號整形，即：char、char*n*、short、short*n*、int、int*n*、long 和 long*n*。泛型 ugentype 指代内建無符號整形，即：uchar、uchar*n*、ushort、ushort*n*、uint、uint*n*、ulong 和 ulong*n*。其中 *n* 為 2、3、4、8 或 16。

對於標量型別的引數，如果所指定的關係為 *false*，則表 *6.17*中的函式 **isequal**、**isnotequal**、**isgreater**、**isgreaterequal**、**isless**、**islessequal**、**islessgreater**、**isfinite**、**isinf**、**isnan**、**isnormal**、**isordered**、**isunordered** 和 **signbit** 會返回 0，否則返回 1。而對於矢量型別的引數，如果所指定的關係為 *false*，則返回 0，否則返回 -1（即所有位都是 1）。

如果任一引數為 NaN，則關係函式 **isequal**、**isgreater**、**isgreaterequal**、**isless**、**islessequal** 和 **islessgreater** 返回 0。如果引數為標量，則當任一引數為 NaN 時，**isnotequal** 返回 1；而如果引數為矢量，則當任一引數為 NaN 時，**isnotequal** 返回 -1。

表 6.17α　標量和矢量關係函式

函式	描述
int **isequal** (float *x*, float *y*) intn **isequal** (floatn *x*, floatn *y*) int **isequal** (double *x*, double *y*) longn **isequal** (doublen *x*, doublen *y*)	按組件逐一比較 $x == y$。
int **isnotequal** (float *x*, float *y*) intn **isnotequal** (floatn *x*, floatn *y*) int **isnotequal** (double *x*, double *y*) longn **isnotequal** (doublen *x*, doublen *y*)	按組件逐一比較 $x! = y$。
int **isgreater** (float *x*, float *y*) intn **isgreater** (floatn *x*, floatn *y*) int **isgreater** (double *x*, double *y*) longn **isgreater** (doublen *x*, doublen *y*)	按組件逐一比較 $x > y$。
int **isgreaterequal** (float *x*, float *y*) intn **isgreaterequal** (floatn *x*, floatn *y*) int **isgreaterequal** (double *x*, double *y*) longn **isgreaterequal** (doublen *x*, doublen *y*)	按組件逐一比較 $x \geq y$。
int **isless** (float *x*, float *y*) intn **isless** (floatn *x*, floatn *y*) int **isless** (double *x*, double *y*) longn **isless** (doublen *x*, doublen *y*)	按組件逐一比較 $x < y$。
int **islessequal** (float *x*, float *y*) intn **islessequal** (floatn *x*, floatn *y*) int **islessequal** (double *x*, double *y*) longn **islessequal** (doublen *x*, doublen *y*)	按組件逐一比較 $x \leq y$。
int **islessgreater** (float *x*, float *y*) intn **islessgreater** (floatn *x*, floatn *y*) int **islessgreater** (double *x*, double *y*) longn **islessgreater** (doublen *x*, doublen *y*)	按組件逐一比較 $(x < y) \mid\mid (x > y)$。
int **isfinite** (float) intn **isfinite** (floatn) int **isfinite** (double) longn **isfinite** (doublen)	測試引數是否為有限值。
int **isinf** (float) intn **isinf** (floatn) int **isinf** (double) longn **isinf** (doublen)	測試引數是否為無限值（正數或負數）。
int **isnan** (float) intn **isnan** (floatn) int **isnan** (double) longn **isnan** (doublen)	測試引數是否為 NaN。

表 6.17β 標量和矢量關係函式

`int isnormal (float)`
`intn isnormal (floatn)`
`int isnormal (double)`
`longn isnormal (doublen)`
	測試引數是否為規格化值。

`int isordered (float x, float y)`
`intn isordered (floatn x, floatn y)`
`int isordered (double x, double y)`
`longn isordered (doublen x, doublen y)`
	測試引數是否規則。相當於 **isequal**(x, x)&&**isequal**(y, y)。

`int isunordered (float x, float y)`
`intn isunordered (floatn x, floatn y)`
`int isunordered (double x, double y)`
`longn isunordered (doublen x, doublen y)`
	測試引數是否不規則。如果引數 x 或 y 是 NaN，則返回非零值，否則返回零。

`int signbit (float)`
`intn signbit (floatn)`
`int signbit (double)`
`longn signbit (doublen)`
	測試符號位。對於此函式的標量版本，如果設置了符號位，則返回 1，否則返回 0。而在此函式的標量版本中，對於矢量的每個組件，如果設置了符號位則返回 -1（即所有位都是 1），否則返回 0。

`int any (igentype x)`	如果 x 中任一組件的最高位是 1，則返回 1；否則返回 0。
`int all (igentype x)`	如果 x 中所有組件的最高位都是 1，則返回 1；否則返回 0。

`gentype bitselect (gentype a,`
`gentype b,`
`gentype c)`
	如果 c 中的某一位為 0，則選取 a 中的對應位作為結果中對應位的值；否則選取 b 中的對應位作為結果中對應位的值。

`gentype select (gentype a,`
`gentype b,`
`igentype c)`
`gentype select (gentype a,`
`gentype b,`
`ugentype c)`
	對於矢量型別中的每個組件，如果 $c[i]$ 的最高位為 1，則結果為 $b[i]$，否則為 $a[i]$。
	對於標量型別，$result = c?b : a$。
	igentype 和 ugentype 的元素數目以及元素的位數都必須與 gentype 相同。

6.12.7 矢量數據裝載和存儲函式

表 6.18中列出了用來讀寫矢量型別數據的內建函式。泛型 gentype 表示內建數據型別 char、uchar、short、ushort、int、uint、long、ulong、float 或 double。泛型 gentype*n* 表示具有 n 個 gentype 元素的矢量。我們用 half*n* 表示具有 n 個 half 元素的矢量[36]。函式名中也有後綴 n（即 **vload*n***、**vstore*n*** 等），其中 n 為 2、3、4、8 或 16。

> **vload3**、**vload_half3**、**vstore3** 和 **vstore_half3** 所用位址為 $(p + (offset \times 3))$；而 **vloada_half3** 和 **vstorea_half3** 所用位址為 $(p + (offset \times 4))$。

表 6.18α 矢量數據裝載、存儲函式表

函式	描述

`gentypen vloadn (size_t offset,`
`const __global gentype *p)`
`gentypen vloadn (size_t offset,`
`const __local gentype *p)`
	由位址 $(p + (offset \times n))$ 讀取 **sizeof**(gentype*n*) 字節的數據並將其返回。對於位址 $(p + (offset \times n))$ 而言，如果 gentype 為 char、uchar，則他必須按 8 位對齊；如果 gentype 為 short、ushort，則他必須按 16 位對齊；如果 gentype 為 int、uint、float，則他必須按 32 位對齊；如果 gentype 為 long、ulong，則他必須按 64 位對齊。

[36] 僅在擴展 **cl_khr_fp16** 中才定義有 half*n*（參見《OpenCL 1.2 擴展規範》的節 *9.5*）。

表 6.18β　矢量數據裝載、存儲函式表

`void vstoren (gentypen data, size_t offset, __global gentype *p)`
`void vstoren (gentypen data, size_t offset, __local gentype *p)`
`void vstoren (gentypen data, size_t offset, __private gentype *p)`
	將 *data* 中 **sizeof**(gentype*n*) 字節的數據寫入位址 $(p + (offset \times n))$ 中。對於位址 $(p + (offset \times n))$ 而言，如果 gentype 為 char、uchar，則他必須按 8 位對齊；如果 gentype 為 short、ushort，則他必須按 16 位對齊；如果 gentype 為 int、uint、float，則他必須按 32 位對齊；如果 gentype 為 long、ulong，則他必須按 64 位對齊。

`float vload_half (size_t offset, const __global half *p)` `float vload_half (size_t offset, const __local half *p)`	由位址 $(p + offset)$ 讀取 **sizeof**(half) 字節的數據。將讀到的數據按 half 值解釋，將其轉換為 float 後返回。位址 $(p + offset)$ 必須按 16 位對齊。
`floatn vload_halfn (size_t offset, const __global half *p)` `floatn vload_halfn (size_t offset, const __local half *p)`	由位址 $(p + (offset \times n))$ 讀取 **sizeof**(half*n*) 字節的數據。將讀到的數據按 half*n* 值解釋，將其轉換為 float*n* 後返回。位址 $(p + (offset \times n))$ 必須按 16 位對齊。

表 6.18γ 矢量數據裝載、存儲函式表

`void vstore_half (float data, size_t offset, __global half *p)`
`void vstore_half_rte (float data, size_t offset, __global half *p)`
`void vstore_half_rtz (float data, size_t offset, __global half *p)`
`void vstore_half_rtp (float data, size_t offset, __global half *p)`
`void vstore_half_rtn (float data, size_t offset, __global half *p)`
 	
`void vstore_half (float data, size_t offset, __local half *p)`
`void vstore_half_rte (float data, size_t offset, __local half *p)`
`void vstore_half_rtz (float data, size_t offset, __local half *p)`
`void vstore_half_rtp (float data, size_t offset, __local half *p)`
`void vstore_half_rtn (float data, size_t offset, __local half *p)`
 	
`void vstore_half (float data, size_t offset, __private half *p)`
`void vstore_half_rte (float data, size_t offset, __private half *p)`
`void vstore_half_rtz (float data, size_t offset, __private half *p)`
`void vstore_half_rtp (float data, size_t offset, __private half *p)`
`void vstore_half_rtn (float data, size_t offset, __private half *p)`
	先按某種捨入模式將 *data* 中的 float 值轉換為 half 值。然後將其寫入位址 *p* + *offset* 中。位址 *p* + *offset* 必須按 16 位對齊。
	vstore_half 使用缺省的捨入模式。缺省的捨入模式為捨入為最近偶數。

表 6.18δ 矢量數據裝載、存儲函式表

`void vstore_halfn (floatn data, size_t offset, __global half *p)`
`void vstore_halfn_rte (floatn data, size_t offset, __global half *p)`
`void vstore_halfn_rtz (floatn data, size_t offset, __global half *p)`
`void vstore_halfn_rtp (floatn data, size_t offset, __global half *p)`
`void vstore_halfn_rtn (floatn data, size_t offset, __global half *p)`
 	
`void vstore_halfn (floatn data, size_t offset, __local half *p)`
`void vstore_halfn_rte (floatn data, size_t offset, __local half *p)`
`void vstore_halfn_rtz (floatn data, size_t offset, __local half *p)`
`void vstore_halfn_rtp (floatn data, size_t offset, __local half *p)`
`void vstore_halfn_rtn (floatn data, size_t offset, __local half *p)`
 	
`void vstore_halfn (floatn data, size_t offset, __private half *p)`
`void vstore_halfn_rte (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)`
	先按某種捨入模式將 *data* 中的 floatn 值轉換為 halfn 值。然後將其寫入位址 $(p + (offset \times n))$ 中。位址 $(p + (offset \times n))$ 必須按 16 位對齊。
	vstore_halfn 使用缺省的捨入模式。缺省的捨入模式為捨入為最近偶數。

表 6.18ε 矢量數據裝載、存儲函式表

`void vstore_half (double data, size_t offset, __global half *p)`
`void vstore_half_rte (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_rte (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_rte (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)`
	先按某種捨入模式將 *data* 中的 double 值轉換為 half 值。然後將其寫入位址 *p* + *offset* 中。位址 *p* + *offset* 必須按 16 位對齊。
	vstore_half 使用缺省的捨入模式。缺省的捨入模式為捨入為最近偶數。

表 6.18ç 矢量數據裝載、存儲函式表

`void vstore_halfn (doublen data, size_t offset, __global half *p)`
`void vstore_halfn_rte (doublen data, size_t offset, __global half *p)`
`void vstore_halfn_rtz (doublen data, size_t offset, __global half *p)`
`void vstore_halfn_rtp (doublen data, size_t offset, __global half *p)`
`void vstore_halfn_rtn (doublen data, size_t offset, __global half *p)`
 	
`void vstore_halfn (doublen data, size_t offset, __local half *p)`
`void vstore_halfn_rte (doublen data, size_t offset, __local half *p)`
`void vstore_halfn_rtz (doublen data, size_t offset, __local half *p)`
`void vstore_halfn_rtp (doublen data, size_t offset, __local half *p)`
`void vstore_halfn_rtn (doublen data, size_t offset, __local half *p)`
 	
`void vstore_halfn (doublen data, size_t offset, __private half *p)`
`void vstore_halfn_rte (doublen data, size_t offset, __private half *p)`
`void vstore_halfn_rtz (doublen data, size_t offset, __private half *p)`
`void vstore_halfn_rtp (doublen data, size_t offset, __private half *p)`
`void vstore_halfn_rtn (doublen data, size_t offset, __private half *p)`
	先按某種捨入模式將 *data* 中的 doublen 值轉換為 halfn 值。然後將其寫入位址 $(p + (offset \times n))$ 中。位址 $(p + (offset \times n))$ 必須按 16 位對齊。
	vstore_halfn 使用缺省的捨入模式。缺省的捨入模式為捨入為最近偶數。

`floatn vloada_halfn (size_t offset, const __global half *p)` `floatn vloada_halfn (size_t offset, const __local half *p)`	對於 n 為 1、2、4、8 和 16，由位址 $(p + (offset \times n))$ 讀取 **sizeof**(halfn) 字節的數據。讀到的數據解釋為 halfn 值，將其轉換為 floatn 值後返回。 位址 $(p + (offset \times n))$ 必須按 **sizeof**(halfn) 字節對齊。 如果 n = 3，則由位址 $(p + (offset \times 4))$ 讀取 half3 並返回 float3。位址 $(p + (offset \times 4))$ 按 **sizeof**(half) $\times$ 4 字節對齊。

表 6.18η 矢量數據裝載、存儲函式表

`void vstorea_halfn (floatn data, size_t offset, __global half *p)`
`void vstorea_halfn_rte (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_rte (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_rte (floatn data, 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)`
	按某種捨入模式將 *data* 中的 floatn 轉換為 halfn。
	如果 n 為 1、2、4、8 和 16，則將 halfn 值寫入位址 $(p + (offset \times n))$ 位址 $(p + (offset \times n))$ 必須按 **sizeof**(halfn) 字節對齊。
	如果 n = 3，則將 half3 值寫入位址 $(p + (offset \times 4))$ 位址 $(p + (offset \times 4))$ 按 **sizeof**(half) × 4 字節對齊。
	vstorea_halfn 使用缺省的捨入模式。缺省的捨入模式為捨入為最近偶數。

表 6.180 矢量數據裝載、存儲函式表

`void vstorea_halfn (doublen data, size_t offset, __global half *p)`
`void vstorea_halfn_rte (doublen data, size_t offset, __global half *p)`
`void vstorea_halfn_rtz (doublen data, size_t offset, __global half *p)`
`void vstorea_halfn_rtp (doublen data, size_t offset, __global half *p)`
`void vstorea_halfn_rtn (doublen data, size_t offset, __global half *p)`
 	
`void vstorea_halfn (doublen data, size_t offset, __local half *p)`
`void vstorea_halfn_rte (doublen data, size_t offset, __local half *p)`
`void vstorea_halfn_rtz (doublen data, size_t offset, __local half *p)`
`void vstorea_halfn_rtp (doublen data, size_t offset, __local half *p)`
`void vstorea_halfn_rtn (doublen data, size_t offset, __local half *p)`
 	
`void vstorea_halfn (doublen data, size_t offset, __private half *p)`
`void vstorea_halfn_rte (doublen data, size_t offset, __private half *p)`
`void vstorea_halfn_rtz (doublen data, size_t offset, __private half *p)`
`void vstorea_halfn_rtp (doublen data, size_t offset, __private half *p)`
`void vstorea_halfn_rtn (doublen data, size_t offset, __private half *p)`
	按某種捨入模式將 *data* 中的 doublen 轉換為 halfn。
	如果 n 為 1、2、4、8 和 16，則將 halfn 值寫入位址 $(p + (offset \times n))$ 位址 $(p + (offset \times n))$ 必須按 **sizeof**(halfn) 字節對齊。
	如果 n = 3，則將 half3 值寫入位址 $(p + (offset \times 4))$ 位址 $(p + (offset \times 4))$ 按 **sizeof**(half) × 4 字節對齊。
	vstorea_halfn 使用缺省的捨入模式。缺省的捨入模式為捨入為最近偶數。

使用這些函式裝載、存儲矢量數據時，如果所讀寫的位址沒有按表 *6.18*中所描述的方式對齊，則結果未定義。表 *6.18*中存儲函式的指針引數 *p* 可以指向 *__global*、*__local* 或 *__private* 內存。表 *6.18*中裝載函式的指針引數 *p* 可以指向 *__global*、*__local*、*__constant* 或 *__private* 內存。

6.12.8 同步函式

OpenCL C 編程語言實現了如下同步函式。

表 6.19 內建同步函式

函式	描述
`void barrier (cl_mem_fence_flags flags)`	同一作業組中的作業項在處理器上執行此內核時，其中任一作業項要想越過 **barrier** 繼續執行，所有作業項都得先執行此函式。所有作業項必須都能執行到此函式。 如果 **barrier** 在條件語句內，只要有一個作業項會進入此條件語句具有並執行 **barrier**，那麼所有作業項都必須進入此條件語句。 如果 **barrier** 在迴圈語句內，那麼在每一次迭代過程中，任一作業項要想越過 **barrier** 繼續執行，所有作業項都得先執行此函式。 **barrier** 函式還會用內存隔柵（讀、寫都包括）來確保局部、全局內存操作的正確順序。 引數 *flags* 指定內存位址空間，可以是下列常值的組合: • CLK_LOCAL_MEM_FENCE，函式 **barrier** 會通過刷新存儲在局部內存中的所有變量，或者用內存隔柵確保局部內存操作的正確順序。 • CLK_GLOBAL_MEM_FENCE，函式 **barrier** 會用內存隔柵確保全局內存操作的正確順序。例如，作業項寫入緩衝對象或圖像對象後又想讀取更新過的數據，這時此功能就派上用場了。

6.12.9 顯式內存隔柵函式

OpenCL C 編程語言實現了如下顯式內存隔柵函式，可對作業項中的內存操作進行定序。

表 6.20 內建顯式內存隔柵函式

函式	描述
`void mem_fence (cl_mem_fence_flags flags)`	作業項執行內核時，為其中的裝載和存儲進行定序。這意味着在執行 **mem_fence** 之後的裝載和存儲之前，會先將 **mem_fence** 之前的裝載和存儲提交給內存。 引數 *flags* 指定內存位址空間，可以是下列常值的組合: • CLK_LOCAL_MEM_FENCE • CLK_GLOBAL_MEM_FENCE
`void read_mem_fence (cl_mem_fence_flags flags)`	讀內存屏障，僅對裝載定序。 引數 *flags* 指定內存位址空間，可以是下列常值的組合: • CLK_LOCAL_MEM_FENCE • CLK_GLOBAL_MEM_FENCE
`void write_mem_fence (cl_mem_fence_flags flags)`	寫內存屏障，僅對存儲定序。 引數 *flags* 指定內存位址空間，可以是下列常值的組合: • CLK_LOCAL_MEM_FENCE • CLK_GLOBAL_MEM_FENCE

6.12.10 在全局內存和局部內存間的異步拷貝以及預取

OpenCL C 編程語言實現了下列函式，可在全局內存和局部內存間進行異步拷貝，以及從全局內存中預取（prefetch）。

如無特殊說明，泛型 `gentype` 表示函式引數可以是下列內建數據型別:

- `char`、`charn`、`uchar`、`ucharn`、
- `short`、`shortn`、`ushort`、`ushortn`、
- `int`、`intn`、`uint`、`uintn`、
- `long`、`longn`、`ulong`、`ulongn`、
- `float`、`floatn` 或 `double`、`doublen`，

其中 n 可以是 2、3[37]、4、8、16。

[37] 對 **async_work_group_copy** 和 **async_work_group_strided_copy** 而言，矢量型別的組件數目是 3 還是 4 沒有什麼區別。

表 6.21　內建異步拷貝和預取函式

函式	描述
`event_t async_work_group_copy ( __local gentype *dst, const __global gentype *src, size_t num_gentypes, event_t event)` `event_t async_work_group_copy ( __global gentype *dst, const __local gentype *src, size_t num_gentypes, event_t event)`	從 *src* 異步拷貝 *num_gentypes* 個 gentype 元素到 *dst* 中。作業組中的所有作業項都會實施此異步拷貝，因此在作業組中，使用相同引數值執行內核的所有作業項必須都能執行到此函式，否則結果未定義。 返回的事件對象可由 **wait_group_events** 用來等待異步拷貝完畢。可以使用引數 *event* 將 **async_work_group_copy** 與之前的異步拷貝關聯在一起，從而使得多個異步拷貝間可共享同一事件；否則 *event* 必須是零。 如果引數 *event* 非零，則會將其中的事件對象返回。 此函式不會對源數據實施隱式同步，如在拷貝前執行 **barrier**。
`event_t async_work_group_strided_copy ( __local gentype *dst, const __global gentype *src, size_t num_gentypes, size_t src_stride, event_t event)` `event_t async_work_group_strided_copy ( __global gentype *dst, const __local gentype *src, size_t num_gentypes, size_t dst_stride, event_t event)`	從 *src* 異步採集 *num_gentypes* 個 gentype 元素到 *dst* 中。參數 *src_stride* 為從 *src* 中讀取元素時所用跨距。參數 *dst_stride* 為將元素寫入 *dst* 中時所用跨距。作業組中的所有作業項都會實施此異步採集，因此在作業組中，使用相同引數值執行內核的所有作業項必須都能執行到此函式，否則結果未定義。 返回的事件對象可由 **wait_group_events** 用來等待異步拷貝完畢。可以使用引數 *event* 將 **async_work_group_strided_copy** 與之前的異步拷貝關聯在一起，從而使得多個異步拷貝間可共享同一事件；否則 *event* 必須是零。 如果引數 *event* 非零，則會將其中的事件對象返回。 此函式不會對源數據實施隱式同步，如在拷貝前執行 **barrier**。 如果 *src_stride* 或 *dst_stride* 是 0，或者拷貝時，*src_stride* 或 *dst_stride* 使得 *src* 或 *dst* 指針超過了位址空間的上界，則 **async_work_group_strided_copy** 的行為未定義。
`void wait_group_events ( int num_events, event_t *event_list)`	等待用來表示 **async_work_group_copy** 操作完成的事件。實施等待後會釋放 *event_list* 中的事件對象。對於某個作業組中的作業項而言，如果執行內核時用的 *num_events* 以及 *event_list* 中的事件對象一樣，則它們必須都能執行到此函式；否則結果未定義。
`void prefetch ( const __global gentype *p, size_t num_gentypes)`	預取 *num_gentypes* × **sizeof**(gentype) 字節到全局緩存中。預取指令會作用到作業組中的作業項上，不會影響內核的功能性行為。

> 內核必須用內建函式 **wait_group_events** 等待所有異步拷貝全部完成後再退出，否則其行為未定義。

6.12.11 原子函式

OpenCL C 編程語言實現了下列函式，可用來對位於 *__global* 或 *__local* 內存中的 32 位帶符號、無符號整數以及單精度浮點數[38]進行原子操作。

表 6.22α　內建異步拷貝和預取函式

函式	描述

[38] 只有 **atomic_xchg** 才支持單精度浮點數據型別。

表 6.22β 內建異步拷貝和預取函式

`int atomic_add ( volatile __global int *p, int val) unsigned int atomic_add ( volatile __global unsigned int *p, unsigned int val)`
 	
`int atomic_add ( volatile __local int *p, int val) unsigned int atomic_add ( volatile __local unsigned int *p, unsigned int val)`
	讀取 p 所指向的 32 位值 (記為 old)。計算 $(old + val)$ 並將結果存儲到 p 所指位置中。此函式返回 old。

`int atomic_sub ( volatile __global int *p, int val) unsigned int atomic_sub ( volatile __global unsigned int *p, unsigned int val)`
 	
`int atomic_sub ( volatile __local int *p, int val) unsigned int atomic_sub ( volatile __local unsigned int *p, unsigned int val)`
	讀取 p 所指向的 32 位值 (記為 old)。計算 $(old - val)$ 並將結果存儲到 p 所指位置中。此函式返回 old。

`int atomic_xchg ( volatile __global int *p, int val) unsigned int atomic_xchg ( volatile __global unsigned int *p, unsigned int val) float atomic_xchg ( volatile __global float *p, float val)`
 	
`int atomic_xchg ( volatile __local int *p, int val) unsigned int atomic_xchg ( volatile __local unsigned int *p, unsigned int val) float atomic_xchg ( volatile __local float *p, float val)`
	將位置 p 中所存儲的值 old 和 val 中的新值相互交換。返回 old。

`int atomic_inc (volatile __global int *p) unsigned int atomic_inc ( volatile __global unsigned int *p)`
 	
`int atomic_inc (volatile __local int *p) unsigned int atomic_inc ( volatile __local unsigned int *p)`
	讀取 p 所指向的 32 位值 (記為 old)。計算 $(old + 1)$ 並將結果存儲到 p 所指位置中。此函式返回 old。

表 6.22γ　內建異步拷貝和預取函式

`int atomic_dec (volatile __global int *p)`
`unsigned int atomic_dec (`
`volatile __global unsigned int *p)`
 	
`int atomic_dec (volatile __local int *p)`
`unsigned int atomic_dec (`
`volatile __local unsigned int *p)`
	讀取 p 所指向的 32 位值（記為 old）。計算 $(old - 1)$ 並將結果存儲到 p 所指位置中。此函式返回 old。

`int atomic_min (`
`volatile __global int *p,`
`int val)`
`unsigned int atomic_min (`
`volatile __global unsigned int *p,`
`unsigned int val)`
 	
`int atomic_min (`
`volatile __local int *p,`
`int val)`
`unsigned int atomic_min (`
`volatile __local unsigned int *p,`
`unsigned int val)`
	讀取 p 所指向的 32 位值（記為 old）。計算 **min**(old, val) 並將結果存儲到 p 所指位置中。此函式返回 old。

`int atomic_max (`
`volatile __global int *p,`
`int val)`
`unsigned int atomic_max (`
`volatile __global unsigned int *p,`
`unsigned int val)`
 	
`int atomic_max (`
`volatile __local int *p,`
`int val)`
`unsigned int atomic_max (`
`volatile __local unsigned int *p,`
`unsigned int val)`
	讀取 p 所指向的 32 位值（記為 old）。計算 **max**(old, val) 並將結果存儲到 p 所指位置中。此函式返回 old。

`int atomic_and (`
`volatile __global int *p,`
`int val)`
`unsigned int atomic_and (`
`volatile __global unsigned int *p,`
`unsigned int val)`
 	
`int atomic_and (`
`volatile __local int *p,`
`int val)`
`unsigned int atomic_and (`
`volatile __local unsigned int *p,`
`unsigned int val)`
	讀取 p 所指向的 32 位值（記為 old）。計算 $(old \& val)$ 並將結果存儲到 p 所指位置中。此函式返回 old。

表 6.22δ 內建異步拷貝和預取函式

`int atomic_or ( volatile __global int *p, int val) unsigned int atomic_or ( volatile __global unsigned int *p, unsigned int val)`
 	
`int atomic_or ( volatile __local int *p, int val) unsigned int atomic_or ( volatile __local unsigned int *p, unsigned int val)`
	讀取 p 所指向的 32 位值（記為 old）。計算 $(old \mid val)$ 並將結果存儲到 p 所指位置中。此函式返回 old。

`int atomic_xor ( volatile __global int *p, int val) unsigned int atomic_xor ( volatile __global unsigned int *p, unsigned int val)`
 	
`int atomic_xor ( volatile __local int *p, int val) unsigned int atomic_xor ( volatile __local unsigned int *p, unsigned int val)`
	讀取 p 所指向的 32 位值（記為 old）。計算 $(old \hat{} val)$ 並將結果存儲到 p 所指位置中。此函式返回 old。

OpenCL 1.0 規範的節 *9.5* 和節 *9.6* 中列有如下擴展：
- **cl_khr_global_int32_base_atomics**
- **cl_khr_global_int32_extended_atomics**
- **cl_khr_local_int32_base_atomics**
- **cl_khr_local_int32_extended_atomics**

其中所定義的帶有前綴 **atom_** 的內建原子函式也在支持之列。

6.12.12 雜類矢量函式

另外，OpenCL C 編程語言還實現了下列內建矢量函式。如無特殊說明，泛型 `gentypen`（或 `gentypem`）表示函式引數的型別可以為：

- `char{2|4|8|16}`、`uchar{2|4|8|16}`、
- `short{2|4|8|16}`、`ushort{2|4|8|16}`、
- `half{2|4|8|16}`[39]、
- `int{2|4|8|16}`、`uint{2|4|8|16}`、
- `long{2|4|8|16}`、`ulong{2|4|8|16}`、
- `float{2|4|8|16}` 或 `double{2|4|8|16}`[40]。

而泛型 `ugentypen` 則為內建無符號整形。

表 6.23α 內建雜類矢量函式

函式	描述

[39] 僅當支持擴展 **cl_khr_fp16** 時才有效。
[40] 僅當支持雙精度浮點數時才有效

表 6.23β　內建雜類矢量函式

`int vec_step (gentypen a)`
 	
`int vec_step (char3 a)`
`int vec_step (uchar3 a)`
`int vec_step (short3 a)`
`int vec_step (ushort3 a)`
`int vec_step (half3 a)`
`int vec_step (int3 a)`
`int vec_step (uint3 a)`
`int vec_step (long3 a)`
`int vec_step (ulong3 a)`
`int vec_step (float3 a)`
`int vec_step(double3 a)`
 	
`int vec_step(type)`
	此函式的引數為標量或矢量，返回值為引數中的元素數目。
	對於所有標量型別，返回值為 1。
	對於 3 組件矢量，返回值為 4。
	引數也可以是純型別，如 **vec_step**(float2)。

`gentypen shuffle (`
`gentypem x,`
`ugentypen mask)`
`gentypen shuffle2 (`
`gentypem x,`
`gentypem y,`
`ugentypen mask)`
	輸入為一個或兩個型別相同的矢量，由其中的元素構造一個排列並返回，所返回的矢量元素型別與輸入相同，矢量長度與掩碼相同。掩碼中每個元素的大小必須與結果中元素大小相同。
	對於 **shuffle**，只考慮 *mask* 中每個元素的 **ilogb**$(2m-1)$ 個最低有效位。而對於 **shuffle2**，則會考慮 *mask* 中每個元素的 **ilogb**$(2m-1)+1$ 個最低有效位。忽略 *mask* 中的其他位。
	因為要對輸入矢量中的元素從左到右進行編號（第二個輸入矢量的編號續接第一個的），因此需要用 **vec_step**(gentypem) 得到矢量元素數目。引數 *mask* 用來確定結果中的相應元素選自輸入矢量中的哪個元素。
	例:
	
	`uint4 mask = (uint4)(3, 2, 1, 0);`
	`float4 a;`
	`float4 r = shuffle(a, mask);`
	`// r.s0123 = a.wzyx`
	
	`uint8 mask = (uint8)(0, 1, 2, 3, 4, 5, 6, 7);`
	`float4 a, b;`
	`float8 r = shuffle2(a, b, mask);`
	`// r.s0123 = a.xyzw`
	`// r.s4567 = b.xyzw`
	
	`uint4 mask;`
	`float8 a;`
	`float4 b;`
	`b = shuffle(a, mask);`
	
	而無效的例子有:
	
	`uint8 mask;`
	`short16 a;`
	`short8 b;`
	`b = shuffle(a, mask); <- not valid`

6.12.13 printf

OpenCL C 編程語言還實現有 **printf** 函式。

表 6.24α　內建 printf 函式

函式	描述

表 6.24β 內建 printf 函式

`int printf(constant char * restrict format, ...)`	在 *format* 所指字串（指定後續引數如何轉換）的控制下，將輸出寫入實作所定義的數據流，如 stdout。如果引數不足（對格式字串而言），則其行為未定義。而如果引數有剩餘，則會對剩餘的引數求值（一直這樣）但將其忽略。此函式解析完格式字串就會返回。 如果執行成功，此函式返回 0，否則返回 -1。

6.12.13.1 printf 輸出同步

當特定內核所關聯的事件完成後，此內核調用 **printf** 得到的輸出會刷入實作所定義的輸出數據流。在命令隊列上調用 **clFinish** 會將所有擱置的 **printf** 輸出（由之前入隊並完成的命令產生）刷入實作所定義的輸出數據流。在多個作業項並發執行 **printf** 的情況下，寫入數據的順序沒有任何保證。例如，全局 ID為 $(0, 0, 1)$的作業項的輸出與全局 ID為 $(0, 0, 4)$的作業項的輸出很可能混雜在一起。

6.12.13.2 printf 格式字串

格式是一個字符序列，其開始和結束均位於其初始轉義狀態（shift state）下。此格式可能包含零個或多個指示：普通字符（非 **%**），不作改變直接拷貝到輸出數據流中；以及轉換規約，每個都會導致讀取零個或多個後續引數，可能還會根據對應的轉換限定符將其轉換（如果可以的話），然後將結果寫入輸出數據流。格式字串必須位於 *constant* 位址空間中，因此必須在編譯時就需要確定下來，不能由可執行程式動態創建。

每個轉換規約都以字符 **%** 引入，下面按順序列出了跟在 **%** 後面的內容：

- 零個或多個**標誌**（*flag*）（順序任意），可修正轉換規約的意義。
- 最小**欄寬**（*field width*），可選。如果轉換後，字符數小於欄寬，則在左側以空格填補達到欄寬（缺省使用空格，如果有後面所描述的左側調整標誌，則會在右側填補）。欄寬為非負十進制整數[41]。
- **精度**（*precision*），可選。如果出現在 **d**、**i**、**o**、**u**、**x** 和 **X** 轉換中，則作為數字的最小位數；如果出現在 **a**、**A**、**e**、**E**、**f** 和 **F** 轉換中，則作為小數點後面數字的位數；如果出現在 **g** 和 **G** 轉換中，則作為尾數數字的最大位數；如果出現在 **s** 轉換中，則為要寫入的最大字節數。精度的形式為點（.）後跟一個可選的十進制整數；如果只有點，則用零作精度。如果精度與其他任一轉換規約一同出現，則其行為未定義。
- **矢量說明符**（*vector specifier*），可選。
- **長度修飾符**（*length modifier*），指定引數的大小。長度修飾符與矢量說明符一起指定矢量型別。不允許在矢量型別間進行隱式轉換（遵守節 *6.2.1*）。如果沒有指定矢量說明符，則長度修飾符是可選的。
- **轉換說明符**（*conversion specifier*），用來指定要應用哪種轉換。

標誌字符及其意義如下：

– 轉換結果在欄位內左對齊（如果沒有此標誌則為右對齊）。

+ 帶符號的轉換，結果總會帶有正負號（如果沒有此標誌，則只有負數才帶有符號[42]）。

space 如果帶符號轉換的第一個字符不是正負號，或者帶符號轉換的結果為空，則在結果前添一個空格。如果同時指定了 space 和 +，則忽略 space。

結果將是“另一種形式”。對於 **o** 轉換，他會增加精度強制結果的第一個數字為零（僅在有必要時才這樣做，如果值本身就是 0，則只打印一個 0）。對於 **x**（或 **X**）轉換，非零結果將帶有前綴 **0x**（或 **0X**）。對於 **a**、**A**、**e**、**E**、**f**、**F**、**g** 和 **G** 轉換，浮點數的轉換結果始終帶有小數點，即使後面沒有數字（通常，後面有數字時才有小數點）。對於 **g** 和 **G** 轉換，不會移除結果中尾隨的零。對於其他轉換，其行為未定義。

0 對於 **d**、**i**、**o**、**u**、**x**、**X**、**a**、**A**、**e**、**E**、**f**、**F**、**g** 和 **G** 轉換，用前導零（跟在正負號或尾數後面）而不是空格將結果填補到欄寬，轉換無窮或 NaN 時除外。如果 **0** 和 **–** 同時存在，

[41] 注意，如果欄寬以**0**開頭，則將**0**當作標誌。
[42] 轉換浮點數時，負零以及捨入為零的負值都會使結果帶有負號。

則忽略 **0**。對於 **d**、**i**、**o**、**u**、**x** 和 **X** 轉換，如果指定了精度，則忽略 **0**；對於其他轉換，其行為未定義。

矢量說明符及其意義如下：

v*n* 表明後面的 **a**、**A**、**e**、**E**、**f**、**F**、**g**、**G**、**d**、**i**、**o**、**u**、**x** 或 **X** 轉換說明符作用到矢量引數上，其中 ***n*** 是矢量的大小，必須是 2、3、4、8 或 16。

矢量值按如下形式顯示：

```
value1 C value2 C ... C valuen
```

其中 C 是分隔符，此分隔符為逗號（,）。

如果沒有使用矢量說明符，長度修飾符及其意義如下：

hh 表明後面的 **d**、**i**、**o**、**u**、**x** 或 **X** 轉換說明符作用到 **char** 或 **uchar** 引數上（引數還是先按整數相應規則進行型別晉陞，不過在打印前會先轉換成 **char** 或 **uchar**）。

h 表明後面的 **d**、**i**、**o**、**u**、**x** 或 **X** 轉換說明符作用到 **short** 或 **ushort** 引數上（引數還是先按整數相應規則進行型別晉陞，不過在打印前會先轉換成 **short** 或 **ushort**）。

l（ell）表明後面的 **d**、**i**、**o**、**u**、**x** 或 **X** 轉換說明符作用到 **long** 或 **ulong** 引數上。完整規格的 OpenCL 是支持修飾符 **l** 的。而對於嵌入式規格的 OpenCL，僅當设备支持 64 位整數時才支持修飾符 **l**。

如果使用了矢量說明符，長度修飾符及其意義如下：

hh 表明後面的 **d**、**i**、**o**、**u**、**x** 或 **X** 轉換說明符作用到 **char*n*** 或 **uchar*n*** 引數上（不會對引數進行型別晉陞）。

h 表明後面的 **d**、**i**、**o**、**u**、**x** 或 **X** 轉換說明符作用到 **short*n*** 或 **ushort*n*** 引數上（不會對引數進行型別晉陞）；而後面的 **a**、**A**、**e**、**E**、**f**、**F**、**g** 或 **G** 轉換說明符作用到 **half*n*** 引數上。

hl 此修飾符可以與矢量說明符一起使用。表明後面的 **d**、**i**、**o**、**u**、**x** 或 **X** 轉換說明符作用到 **int*n*** 或 **uint*n*** 引數上；而後面的 **a**、**A**、**e**、**E**、**f**、**F**、**g** 或 **G** 轉換說明符作用到 **float*n*** 引數上。

l（ell）表明後面的 **d**、**i**、**o**、**u**、**x** 或 **X** 轉換說明符作用到 **long*n*** 或 **ulong*n*** 引數上；而後面的 **a**、**A**、**e**、**E**、**f**、**F**、**g** 或 **G** 轉換說明符作用到 **double*n*** 引數上。完整規格的 OpenCL 是支持修飾符 **l** 的。而對於嵌入式規格的 OpenCL，僅當设备支持 64 位整數或雙精度浮點數時才支持修飾符 **l**。

> 如果沒有長度說明符，只有矢量說明符，則其行為未定義。矢量說明符以及長度修飾符所描述的矢量數據型別必須與引數的型別一致；否則其行為未定義。
> 如果長度修飾符不是與上述轉換說明符一起出現的，則其行為未定義。

轉換說明符及其意義如下：

d, i 將 **int**、**char*n***、**short*n***、**int*n*** 或 **long*n*** 型別的引數轉換成帶符號十進制數，樣式為 *[-]dddd*。精度指定所顯式數字的最少位數；如果所轉換的值可以用更少的數字表示，則添加前導零。缺省精度為 1。用精度零轉換零值，其結果為空，沒有任何字符。

o, u, x, X 將 **unsigned int**、**uchar*n***、**ushort*n***、**uint*n*** 或 **ulong*n*** 型別的引數轉換成無符號八進制數（**o**）、無符號十進制數（**u**）或無符號十六進制數（**x** 或 **X**），樣式為 *dddd*；**x** 使用字母 **abcdef**，而 **X** 使用字母 **ABCDEF**。精度指定所顯式數字的最少位數；如果所轉換的值可以用更少的數字表示，則添加前導零。缺省精度為 1。用精度零轉換零值，其結果為空，沒有任何字符。

f, F 將 **double**、**half*n***、**float*n*** 或 **double*n*** 型別的浮點引數轉換成十進制符號，樣式為 *[-]ddd.dddd*，其中小數點後面數字的位數等於精度。如果沒有指定精度，則使用缺省值

6；如果精度為零，並且沒有指定 **#** 標誌，則不會有小數部分（包括小數點）。如果有小數點，則小數點前至少有一位數字。會對引數進行捨入。如果引數為無窮，則轉換後樣式為 *[-]inf* 或 *[-]infinity*——至於是哪一種依賴於具體實作。如果引數為 NaN，則轉換後樣式為 *[-]nan* 或 *[-]nan(n-char-sequence)* ——至於是哪一種，以及*n-char-sequence*的意義都依賴於具體實作。 轉換說明符 **F** 則會產生 **INF**、**INFINITY** 或 **NAN**，分別代替 **inf**、**infinity** 或 **nan**[43]。

e、**E**　將 **double**、**half***n*、**float***n* 或 **double***n* 型別的浮點引數轉換成十進制符號，樣式為 *[-]d.dddd e±dd*，小數點前面有一位數字（如果引數非零，則此數字也非零），小數點後面數字的位數等於精度；如果沒有指定精度，則使用缺省值 6；如果精度為零，並且沒有指定 **#** 標誌，則不會有小數部分（包括小數點）。會對引數進行捨入。轉換說明符 **E** 所產生的結果會將指數前面的 **e** 換成 **E**。指數至少包含兩位數字，如果多於兩位，則只包含表示指數所必要的數字。如果引數的值為零，則指數也是零。如果引數為無窮或 NaN，則轉換結果與轉換說明符 **f** 或 **F** 一樣。

g、**G**　將 **double**、**half***n*、**float***n* 或 **double***n* 型別的浮點引數按 **f** 或 **e** 的樣式進行轉換（如果是 **G**，則按 **F** 或 **E** 的樣式進行轉換），具體按哪種轉換取決於所要轉換的值以及精度。如果精度非零，將其記為 P；如果省略了精度，則讓 P 為 6；如果精度為零，則讓 P 為 1。然後，將按 **E** 轉換時的指數記為 X，如果 $P > X \geq -4$，則按 **f**（或 **F**）以及精度 $P-(X+1)$ 進行轉換。否則，按 **e**（或 **E**）以及精度 $P-1$ 進行轉換。最後，除非使用了標誌 **#**，否則移除小數部分中所有尾隨的零，如果這樣小數部分為空，就連小數點一併移除。如果引數為無窮或 NaN，則轉換結果與轉換說明符 **f** 或 **F** 一樣。

a、**A**　**double**、**half***n*、**float***n* 或 **double***n* 型別的浮點引數轉換後的樣式為：

*[-]***0xh.***hhhh* **p**±d

其中小數點前有一位十六進制數字（如果引數為規格化浮點數，則此數字非零，否則未規定[44]），小數點後十六進制數字的位數與精度相同；如果沒有指定精度，則選用剛好能確切表示其值的精度；如果精度為零，而且沒有指定標誌 **#**，則不會有小數部分（包括小數點）。 **a** 使用字母 **abcdef**，**A** 使用字母 **ABCDEF**。**A** 會用 **X** 和 **P** 分別取代樣式中的 **x** 和 **p**。指數至少包含一位數字，否則僅為表示十進制指數所必要的數字。如果引數為零，則指數也是零。如果引數為無窮或 NaN，則轉換結果與轉換說明符 **f** 或 **F** 一樣。

僅當支持數據型別 double 時，轉換說明符 **e**、**E**、**g**、**G**、**a**、**A** 才會將 float 或 half 型別的引數轉換成 double。如果不支持 double，則用 **float** 取代 **double** 作為引數，同時會將 half 轉換為 float。

c　會將 **int** 引數轉換為 **unsigned char**，並輸出字符。

s　引數必須是常值字串[45]。常值字串陣列中的字符都會被寫入輸出數據流，直到遇到 null 終止符，終止符不會被寫入。如果指定了精度，所寫入字符的數目不會超過精度值。如果沒有指定精度或者精度值大於陣列的大小，陣列將包含 null 字符。

p　引數必須是指向 **void** 的指針。此指針可以指代 *global*、*constant*、*local* 或 *private* 位址空間中的某個內存區域。指針的值轉換成一個可打印字符的序列，其內容依賴於具體實作。

%　直接輸出 **%**，不會轉換任何引數。完整的轉換規約為 **%%**。

如果轉換規約無效，其行為未定義。如果任一轉換規約的對應引數型別不正確，其行為也是未定義的。

任何情況下，欄寬較小和欄位不存在都不會引起欄位的截斷；如果轉換結果比欄寬寬，則會對此欄位進行擴充，以包含整個轉換結果。

對於 **a** 和 **A** 轉換，會將引數正確捨入成具有給定精度的十六進制浮點數。

[43] 當作用於無窮和 NaN 值時，**-**、**+** 和 **space** 標誌的意義不變；而 **#** 和 **0** 標誌沒有效果。
[44] 實作可以自行選擇小數點左邊的數字，以使後續數字按半字節（4 位）邊界對齊。
[45] 沒有專門針對多字節字符的條款。對於 **printf** 的轉換說明符 **s** 而言，如果引數不是常值字串，則其行為未定義。

下面給出了 **printf** 的一些例子。

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uchar4 uc = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);

printf("f4 = %2.2v4hlf\n", f);
printf("uc = %#v4hhx\n", uc);
```

上面兩次 **printf** 調用打印結果如下:

```
f4 = 1.00,2.00,3.00,4.00
uc = 0xfa,0xfb,0xfc,0xfd
```

下面是關於 **printf** 的轉換說明符 **s** 的合法用例。引數必須是常值字串:

```
kernel void my_kernel(... )
{
        printf(“%s\n”, “this is a test string\n”);
}
```

下面是關於 **printf** 的轉換說明符 **s** 的無效用例。

```
kernel void my_kernel(global char *s, ... )
{
        printf(“%s\n”, s);
}

constant char *p = “this is a test string\n”;
printf(“%s\n”, p);
printf(“%s\n”, &p[3]);
```

下面也是一個 **printf** 的無效用例, 矢量說明符和長度修飾符所確定的數據型別與引數的型別不一致:

```
kernel void my_kernel(global char *s, ... )
{
        uint2 ui = (uint2)(0x12345678, 0x87654321);
        printf("unsigned short value = (%#v2hx)\n", ui)
        printf("unsigned char value = (%#v2hhx)\n", ui)
}
```

6.12.13.3 printf 在 OpenCL 和 C99 中的差異

- OpenCL C 中, 不支持在轉換說明符 **c** 或 **s** 前使用修飾符 **l**。
- OpenCL C 不支持長度說明符 **ll**、**j**、**z**、**t** 以及 **L**, 不過暫時保留以備將來可能使用。
- OpenCL C 添加了可選的矢量說明符 ***vn*** 用以支持矢量型別的打印。
- 僅當支持數據型別 double 時, 轉換說明符 **f**、**F**、**e**、**E**、**g**、**G**、**a**、**A** 才會將 float 引數轉換成 double。參見表 *4.3* 中的 CL_DEVICE_DOUBLE_FP_CONFIG。如果不支持 double, 則引數為 float 而不是 double。
- 對於嵌入式規格, 僅當支持 64 位整數時才支持長度說明符 **l**。
- 在 OpenCL C 中, 如果執行成功, **printf** 會返回 0, 否則返回 -1。而在 C99 中, **printf** 會返回所打印字符的數目, 如果發生了輸出錯誤或編碼錯誤則返回一個負值。
- 在 OpenCL C 中, 轉換說明符 **s** 只能用於常值字串。

6.12.14 圖像讀寫函式

本節中所定義的內建函式僅能與圖像對象一起使用。

聲明可被內核讀取的圖像對象時應帶有限定符 *__read_only*。對帶有 *__read_only* 的圖像對象調用 **write_image** 會造成編譯錯誤。聲明可被內核寫入的圖像對象時應帶有限定符 *__write_only*。對帶有 *__write_only* 的圖像對象調用 **read_image** 會造成編譯錯誤。不支持在同一內核中對同一圖像對象同時調用 **read_image** 和 **write_image**。

read_image 返回的是一個四組件矢量浮點數、整數或無符號整數顏色值。此值用 x、y、z、w 來標識，其中 x 指代紅色分量，y 指代綠色分量，z 指代藍色分量，w 指代 alpha 分量，每個分量都是一個矢量組件。

6.12.14.1 採樣器

圖像讀取函式的引數中有一個就是採樣器。採樣器可作為引數由 **clSetKernelArg** 傳給內核，也可以在 *kernel* 函式的最外層聲明採樣器，或者是程式源碼中聲明的型別為 sampler_t 的常量。

程式中所聲明採樣器變量的型別為 sampler_t。這種變量必須用 32 位無符號整形常數進行初始化，按位欄解釋此常量，其位欄指定了下列屬性:

- 尋址模式
- 濾波模式
- 歸一化坐標

這些數學控制着 **read_image{f|i|ui}** 如何讀取圖像中的元素。

也可在程式源碼中用如下幾種語法將採樣器聲明為全局常量:

```
const sampler_t         <sampler name> = <value>
or
constant sampler_t      <sampler name> = <value>
or
__constant sampler_t    <sampler_name> = <value>
```

在計算每個設備中指向常數位址空間的引數數目或常數位址空間的大小時，不考慮帶有限定符 *constant* 的採樣器（參見表 *4.3*中的 CL_DEVICE_MAX_CONSTANT_ARGS 和 CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE）。

表 6.25 採 樣 器 描 述 符

採樣器屬性	描述
<normalized coords>	指定所傳入的坐標 x、y 和 z 是否已歸一化。他必須是常值，可以是下列預定義枚舉中的一個: • CLK_NORMALIZED_COORDS_TRUE • CLK_NORMALIZED_COORDS_FALSE 在單個內核中針對同意圖像多次調用 **read_image{f|i|ui}** 時，所用採樣器中 <normalized coords> 的值必須相同。
<addressing mode>	指定圖像的尋址模式，即圖像坐標溢出時如何處置。他必須是常值，可以是下列預定義枚舉中的一個: • CLK_ADDRESS_MIRRORED_REPEAT——在整數接點處翻轉圖像坐標。這種尋址模式只能用於歸一化坐標。如果使用的不是歸一化坐標，則此模式生成的圖像坐標未定義。 • CLK_ADDRESS_REPEAT——溢出的坐標會繞回到有效區間內。這種尋址模式只能用於歸一化坐標。如果使用的不是歸一化坐標，則此模式生成的圖像坐標未定義。 • CLK_ADDRESS_CLAMP_TO_EDGE——溢出的坐標會被壓入有效範圍內。 • CLK_ADDRESS_CLAMP[1]——溢出的坐標會返回顏色極值。 • CLK_ADDRESS_NONE——此模式下，由程式員保證坐標不會溢出，否則結果未定義。 對於 1D 和 2D 圖像陣列，尋址模式僅對坐標 x 和 (x, y) 有效。坐標中的陣列索引所用尋址模式始終是 CLK_ADDRESS_CLAMP_TO_EDGE。
filter mode	指定所用的濾波模式。他必須是常值，可以是下列預定義枚舉中的一個: • CLK_FILTER_NEAREST • CLK_FILTER_LINEAR 對於這些濾波模式的描述，請參見節 *8.2*。

[1] 與尋址模式CLK_ADDRESS_CLAMP_TO_EDGE類似。

例:

```
const sampler_t samplerA = CLK_NORMALIZED_COORDS_TRUE
                         | CLK_ADDRESS_REPEAT
                         | CLK_FILTER_NEAREST;
```

採樣器 **samplerA** 用的是規格化坐標、重複尋址模式和最近濾波。

對於一個內核中所能聲明採樣器的最大數目，可以用 **clGetDeviceInfo** 以 CL_DEVICE_MAX_SAMPLERS 進行查詢。

6.12.14.1.1 確定顏色極值

如果採樣器中的 <addressing mode> 是 CLK_ADDRESS_CLAMP，則溢出的圖像坐標會返回顏色極值。所選顏色極值取決於圖像通道順序，可以是下列中的一個：

- 如果圖像通道順序為 CL_A、CL_INTENSITY、CL_Rx、CL_RA、CL_RGx、CL_RGBx，CL_ARGB、CL_BGRA 或 CL_RGBA，則所選顏色極值為 (0.0f, 0.0f, 0.0f, 0.0f)。
- 如果圖像通道順序為 CL_R、CL_RG、CL_RGB 或 CL_LUMINANCE，則所選顏色極值為 (0.0f, 0.0f, 0.0f, 1.0f)。

6.12.14.2 內建圖像讀取函式

下列帶有採樣器的內建函式可用於讀取圖像。

表 6.26α 內建圖像讀取函式

函式	描述
float4 **read_imagef** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) float4 **read_imagef** (image2d_t *image*, sampler_t *sampler*, float2 *coord*)	用坐標 $(coord.x, coord.y)$ 查找 2D 圖像對象 *image* 中的元素。 如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。 對於使用整數坐標的 **read_imagef** 所用採樣器而言，其濾波模式必須是 CLK_FILTER_NEAREST，歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；如果是其他值，結果未定義。 如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。
int4 **read_imagei** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) int4 **read_imagei** (image2d_t *image*, sampler_t *sampler*, float2 *coord*) uint4 **read_imageui** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) uint4 **read_imageui** (image2d_t *image*, sampler_t *sampler*, float2 *coord*)	用坐標 $(coord.x, coord.y)$ 查找 2D 圖像對象 *image* 中的元素。 **read_imagei** 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。 對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一： • CL_SIGNED_INT8 • CL_SIGNED_INT16 • CL_SIGNED_INT32 如果 *image_channel_data_type* 不在上述值之列，則結果未定義。 對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一： • CL_UNSIGNED_INT8 • CL_UNSIGNED_INT16 • CL_UNSIGNED_INT32 如果 *image_channel_data_type* 不在上述值之列，則結果未定義。 **read_image{i|ui}** 僅支持最近濾波。即 *sampler* 中的濾波模式必須是 CLK_FILTER_NEAREST；否則結果未定義。 對於使用整數坐標的 **read_image{i|ui}** 所用採樣器而言，其歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；否則結果未定義。

表 6.26β 內建圖像讀取函式

`float4 read_imagef ( image3d_t image, sampler_t sampler, int4 coord )` `float4 read_imagef ( image3d_t image, sampler_t sampler, float4 coord)`	用坐標 $(coord.x, coord.y, coord.z)$ 查找 3D 圖像對象 *image* 中的元素。其中 $coord.w$ 被忽略。 如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。 對於使用整數坐標的 **read_imagef** 所用採樣器而言，其濾波模式必須是 CLK_FILTER_NEAREST，歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；如果是其他值，結果未定義。 如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。

`int4 read_imagei ( image3d_t image, sampler_t sampler, int4 coord)`
`int4 read_imagei ( image3d_t image, sampler_t sampler, float4 coord)`
`uint4 read_imageui ( image3d_t image, sampler_t sampler, int4 coord)`
`uint4 read_imageui ( image3d_t image, sampler_t sampler, float4 coord)`
	用坐標 $(coord.x, coord.y, coord.z)$ 查找 3D 圖像對象 *image* 中的元素。其中 $coord.w$ 被忽略。
	read_imagei 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。每個通道的值都是 32 位整數。
	對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	read_image{i|ui} 僅支持最近濾波。即 *sampler* 中的濾波模式必須是 CLK_FILTER_NEAREST；否則結果未定義。
	對於使用整數坐標的 **read_image{i|ui}** 所用採樣器而言，其歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；否則結果未定義。

`float4 read_imagef ( image2d_array_t image, sampler_t sampler, int4 coord )` `float4 read_imagef ( image2d_array_t image, sampler_t sampler, float4 coord)`	用坐標 $coord.z$ 確定 2D 圖像陣列 *image* 中的某一個 2D 圖像；用坐標 $(corrd.x, coord.y)$ 來查找此 2D 圖像中的元素。其中 $coord.w$ 被忽略。 如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。 對於使用整數坐標的 **read_imagef** 所用採樣器而言，其濾波模式必須是 CLK_FILTER_NEAREST，歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；如果是其他值，結果未定義。 如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。

表 6.26γ　內建圖像讀取函式

`int4 read_imagei (`
`  image2d_array_t image,`
`  sampler_t sampler,`
`  int4 coord)`
`int4 read_imagei (`
`  image2d_array_t image,`
`  sampler_t sampler,`
`  float4 coord)`
`uint4 read_imageui (`
`  image2d_array_t image,`
`  sampler_t sampler,`
`  int4 coord)`
`uint4 read_imageui (`
`  image2d_array_t image,`
`  sampler_t sampler,`
`  float4 coord)`
	用坐標 $coord.z$ 確定 2D 圖像陣列 *image* 中的某一個 2D 圖像；用坐標 $(corrd.x, coord.y)$ 來查找此 2D 圖像中的元素。其中 $coord.w$ 被忽略。
	read_imagei 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。每個通道的值都是 32 位整數。
	對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	read_image{i|ui} 僅支持最近濾波。即 *sampler* 中的濾波模式必須是 CLK_FILTER_NEAREST；否則結果未定義。
	對於使用整數坐標的 **read_image{i|ui}** 所用採樣器而言，其歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；否則結果未定義。

`float4 read_imagef (`
`  image1d_t image,`
`  sampler_t sampler,`
`  int coord)`
`float4 read_imagef (`
`  image1d_t image,`
`  sampler_t sampler,`
`  float coord)`
	用坐標 $coord$ 查找 1D 圖像對象 *image* 中的元素。
	如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。
	如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。
	如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。
	對於使用整數坐標的 **read_imagef** 所用採樣器而言，其濾波模式必須是 CLK_FILTER_NEAREST，歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；如果是其他值，結果未定義。
	如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。

`int4 read_imagei (`
`  image1d_t image,`
`  sampler_t sampler,`
`  int coord)`
`int4 read_imagei (`
`  image1d_t image,`
`  sampler_t sampler,`
`  float coord)`
 	
`uint4 read_imageui (`
`  image1d_t image,`
`  sampler_t sampler,`
`  int coord)`
`uint4 read_imageui (`
`  image1d_t image,`
`  sampler_t sampler,`
`  float coord)`
	用坐標 $coord$ 查找 1D 圖像對象 *image* 中的元素。
	read_imagei 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。每個通道的值都是 32 位整數。
	對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	read_image{i|ui} 僅支持最近濾波。即 *sampler* 中的濾波模式必須是 CLK_FILTER_NEAREST；否則結果未定義。
	對於使用整數坐標的 **read_image{i|ui}** 所用採樣器而言，其歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；否則結果未定義。

表 6.26δ 內建圖像讀取函式

`float4 read_imagef (`
`image1d_array_t image,`
`sampler_t sampler,`
`int2 coord)`
`float4 read_imagef (`
`image1d_array_t image,`
`sampler_t sampler,`
`float2 coord)`
	用坐標 $coord.y$ 確定 1D 圖像陣列 *image* 中的某一個 1D 圖像；用坐標 $corrd.x$ 來查找此 1D 圖像中的元素。
	如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。
	如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。
	如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。
	對於使用整數坐標的 **read_imagef** 所用採樣器而言，其濾波模式必須是 CLK_FILTER_NEAREST，歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；如果是其他值，結果未定義。
	如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。

`int4 read_imagef (`
`image1d_array_t image,`
`sampler_t sampler,`
`int2 coord)`
`int4 read_imagef (`
`image1d_array_t image,`
`sampler_t sampler,`
`float2 coord)`
`uint4 read_imagef (`
`image1d_array_t image,`
`sampler_t sampler,`
`int2 coord)`
`uint4 read_imagef (`
`image1d_array_t image,`
`sampler_t sampler,`
`float2 coord)`
	用坐標 $coord.y$ 確定 1D 圖像陣列 *image* 中的某一個 1D 圖像；用坐標 $corrd.x$ 來查找此 1D 圖像中的元素。
	read_imagei 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。每個通道的值都是 32 位整數。
	對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	read_image{i|ui} 僅支持最近濾波。即 *sampler* 中的濾波模式必須是 CLK_FILTER_NEAREST；否則結果未定義。
	對於使用整數坐標的 **read_image{i|ui}** 所用採樣器而言，其歸一化坐標必須是 CLK_NORMALIZED_COORDS_FALSE，尋址模式必須是 CLK_ADDRESS_CLAMP_TO_EDGE、CLK_ADDRESS_CLAMP 或 CLK_ADDRESS_NONE；否則結果未定義。

6.12.14.3 內建無採樣器圖像讀取函式

下列無採樣器的內建函式也可用於讀取圖像。其行為與節 *6.12.14.2*中坐標為整數、並帶有採樣器的對應函式一樣，相當於採樣器的濾波模式為 CLK_FILTER_NEAREST、歸一化坐標為 CLK_NORMALIZED_COORDS_FALSE、尋址模式為 CLK_ADDRESS_NONE。

表 6.27α 內建無採樣器圖像讀取函式

函式	描述

`float4 read_imagef (`
`image2d_t image,`
`int2 coord)`
	用坐標 $(coord.x, coord.y)$ 查找 2D 圖像對象 *image* 中的元素。
	如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。
	如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。
	如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。
	如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。

表 6.27β 內建無採樣器圖像讀取函式

`int4 read_imagei ( image2d_t image, int2 coord)` `uint4 read_imageui ( image2d_t image, int2 coord)`	用坐標 $(coord.x, coord.y)$ 查找 2D 圖像對象 *image* 中的元素。 **read_imagei** 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。 對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一： • CL_SIGNED_INT8 • CL_SIGNED_INT16 • CL_SIGNED_INT32 如果 *image_channel_data_type* 不在上述值之列，則結果未定義。 對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一： • CL_UNSIGNED_INT8 • CL_UNSIGNED_INT16 • CL_UNSIGNED_INT32 如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
`float4 read_imagef ( image3d_t image, int4 coord )`	用坐標 $(coord.x, coord.y, coord.z)$ 查找 3D 圖像對象 *image* 中的元素。其中 $coord.w$ 被忽略。 如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。 如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。
`int4 read_imagei ( image3d_t image, int4 coord)` `uint4 read_imageui ( image3d_t image, int4 coord)`	用坐標 $(coord.x, coord.y, coord.z)$ 查找 3D 圖像對象 *image* 中的元素。其中 $coord.w$ 被忽略。 **read_imagei** 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。每個通道的值都是 32 位整數。 對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一： • CL_SIGNED_INT8 • CL_SIGNED_INT16 • CL_SIGNED_INT32 如果 *image_channel_data_type* 不在上述值之列，則結果未定義。 對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一： • CL_UNSIGNED_INT8 • CL_UNSIGNED_INT16 • CL_UNSIGNED_INT32 如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
`float4 read_imagef ( image2d_array_t image, int4 coord)`	用坐標 $coord.z$ 確定 2D 圖像陣列 *image* 中的某一個 2D 圖像；用坐標 $(corrd.x, coord.y)$ 來查找此 2D 圖像中的元素。其中 $coord.w$ 被忽略。 如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。 如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。

表 6.27y 內建無採樣器圖像讀取函式

`int4 read_imagei ( image2d_array_t image, int4 coord)` `uint4 read_imageui ( image2d_array_t image, int4 coord)`	用坐標 $coord.z$ 確定 2D 圖像陣列 *image* 中的某一個 2D 圖像；用坐標 $(corrd.x, coord.y)$ 來查找此 2D 圖像中的元素。其中 $coord.w$ 被忽略。 **read_imagei** 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。每個通道的值都是 32 位整數。 對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一： • CL_SIGNED_INT8 • CL_SIGNED_INT16 • CL_SIGNED_INT32 如果 *image_channel_data_type* 不在上述值之列，則結果未定義。 對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一： • CL_UNSIGNED_INT8 • CL_UNSIGNED_INT16 • CL_UNSIGNED_INT32 如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
`float4 read_imagef ( image1d_t image, int coord)` `float4 read_imagef ( image1d_buffer_t image, int coord)`	用坐標 *coord* 查找 1D 圖像對象或 1D 圖像緩衝對象 *image* 中的元素。 如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。 如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。

`int4 read_imagei ( image1d_t image, int coord)`
`uint4 read_imageui ( image1d_t image, int coord)`
`int4 read_imagei ( image1d_buffer_t image, int coord)`
`uint4 read_imageui ( image1d_buffer_t image, int coord)`
	用坐標 *coord* 查找 1D 圖像對象或 1D 圖像緩衝對象 *image* 中的元素。
	read_imagei 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。每個通道的值都是 32 位整數。
	對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。

`float4 read_imagef ( image1d_array_t image, int2 coord)`	用坐標 $coord.y$ 確定 1D 圖像陣列 *image* 中的某一個 1D 圖像；用坐標 $corrd.x$ 來查找此 1D 圖像中的元素。 如果創建圖像對象時所用的 *image_channel_data_type* 是預定義的壓縮過的格式或 CL_UNORM_INT8 或 CL_UNORM_INT16，則返回的浮點值在區間 $[0.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_SNORM_INT8 或 CL_SNORM_INT16，則返回的浮點值在區間 $[-1.0 \cdots 1.0]$ 內。 如果創建圖像對象時所用的 *image_channel_data_type* 是 CL_HALF_FLOAT 或 CL_FLOAT，則返回原始浮點值。 如果創建圖像對象時所用的 *image_channel_data_type* 不再上面所列範圍內，則結果未定義。

表 6.27δ　內建無採樣器圖像讀取函式

`int4 read_imagei (`
`  image1d_array_t image,`
`  int2 coord)`
`uint4 read_imageui (`
`  image1d_array_t image,`
`  int2 coord)`
	用坐標 $coord.y$ 確定 1D 圖像陣列 *image* 中的某一個 1D 圖像；用坐標 $corrd.x$ 來查找此 1D 圖像中的元素。
	read_imagei 和 **read_imageui** 所返回的值分別為非歸一化帶符號整數和非歸一化無符號整數。每個通道的值都是 32 位整數。
	對於 **read_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。
	對於 **read_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果 *image_channel_data_type* 不在上述值之列，則結果未定義。

6.12.14.4 內建圖像寫入函式

下列內建函式可用於寫入圖像。

表 6.28α　內建圖像寫入函式

函式	描述

`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,`
`  uint4 color)`
	將 *color* 寫入 2D 圖像對象 *image* 中由坐標 $(coord.x, coord.y)$ 所指定的位置上。寫之前會進行適當的格式轉換。 $coord.x$ 和 $coord.y$ 會被當成非歸一化坐標，其值必須分別在區間 $0 \cdots$image width $- 1$ 和 $0 \cdots$image height $- 1$ 內。
	對於 **write_imagef**，創建圖像對象時所用的 *image_channel_data_type* 必須是預定義壓縮過的格式或者 CL_SNORM_INT8、CL_UNORM_INT8、CL_SNORM_INT16、CL_UNORM_INT16、CL_HALF_FLOAT 或 CL_FLOAT。
	對於 **write_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	對於 **write_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果創建圖像對象時所用的 *image_channel_data_type* 不再上述所列範圍內，或者坐標 (x, y) 不在 $(0 \cdots$image width $- 1, 0 \cdots$image height $- 1)$ 範圍內，則 **write_imagef**、**write_imagei** 和 **write_imageui** 的行為未定義。

表 6.28β 內建圖像寫入函式

`void write_imagef ( image2d_array_t image, int4 coord, float4 color)`
`void write_imagei ( image2d_array_t image, int4 coord, int4 color)`
`void write_imageui ( image2d_array_t image, int4 coord, uint4 color)`
	用 $coord.z$ 確定 2D 圖像陣列 *image* 中的某一 2D 圖像，將 *color* 寫入此 2D 圖像中由坐標 $(coord.x, coord.y)$ 所指定的位置上。寫之前會進行適當的格式轉換。 $coord.x$、$coord.y$ 以及 $coord.z$ 會被當成非歸一化坐標，其值必須分別在區間 0…image width − 1、0…image height − 1 和 0…image number − 1 內。
	對於 **write_imagef**，創建圖像對象時所用的 *image_channel_data_type* 必須是預定義壓縮過的格式或者 CL_SNORM_INT8、CL_UNORM_INT8、CL_SNORM_INT16、CL_UNORM_INT16、CL_HALF_FLOAT 或 CL_FLOAT。
	對於 **write_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	對於 **write_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果創建圖像對象時所用的 *image_channel_data_type* 不再上述所列範圍內，或者坐標 (x, y, z) 不在 $(0…\text{image width} - 1, 0…\text{image height} - 1, 0…\text{image number} - 1)$ 範圍內，則 **write_imagef**、**write_imagei** 和 **write_imageui** 的行為未定義。

`void write_imagef ( image1d_t image, int coord, float4 color)`
`void write_imagei ( image1d_t image, int coord, int4 color)`
`void write_imageui ( image1d_t image, int coord, uint4 color)`
`void write_imagef ( image1d_buffer_t image, int coord, float4 color)`
`void write_imagei ( image1d_buffer_t image, int coord, int4 color)`
`void write_imageui ( image1d_buffer_t image, int coord, uint4 color)`
	將 *color* 寫入 1D 圖像對象 *image* 中由坐標 *coord* 所指定的位置上。寫之前會進行適當的格式轉換。 *coord* 會被當成非歸一化坐標，其值必須在區間 0…image width − 1 內。
	對於 **write_imagef**，創建圖像對象時所用的 *image_channel_data_type* 必須是預定義壓縮過的格式或者 CL_SNORM_INT8、CL_UNORM_INT8、CL_SNORM_INT16、CL_UNORM_INT16、CL_HALF_FLOAT 或 CL_FLOAT。
	對於 **write_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	對於 **write_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果創建圖像對象時所用的 *image_channel_data_type* 不再上述所列範圍內，或者坐標不在 0…image width − 1 範圍內，則 **write_imagef**、**write_imagei** 和 **write_imageui** 的行為未定義。

表 6.28γ　內建圖像寫入函式

`void write_imagef (`
`  image1d_array_t image,`
`  int2 coord,`
`  float4 color)`
`void write_imagei (`
`  image1d_array_t image,`
`  int2 coord,`
`  int4 color)`
`void write_imageui (`
`  image1d_array_t image,`
`  int2 coord,`
`  uint4 color)`
	用 $coord.y$ 確定 1D 圖像陣列 *image* 中的某一 1D 圖像，將 *color* 寫入此 1D 圖像中由坐標 $coord.x$ 所指定的位置上。寫之前會進行適當的格式轉換。 $coord.x$ 和 $coord.y$ 會被當成非歸一化坐標，其值必須分別在區間 $0 \cdots$ image width -1 和 $0 \cdots$ image number -1 之間。
	對於 **write_imagef**，創建圖像對象時所用的 *image_channel_data_type* 必須是預定義壓縮過的格式或者 CL_SNORM_INT8、CL_UNORM_INT8、CL_SNORM_INT16、CL_UNORM_INT16、CL_HALF_FLOAT 或 CL_FLOAT。
	對於 **write_imagei** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_SIGNED_INT8
	• CL_SIGNED_INT16
	• CL_SIGNED_INT32
	對於 **write_imageui** 而言，創建圖像對象時所用的 *image_channel_data_type* 必須是下列值之一：
	• CL_UNSIGNED_INT8
	• CL_UNSIGNED_INT16
	• CL_UNSIGNED_INT32
	如果創建圖像對象時所用的 *image_channel_data_type* 不再上述所列範圍內，或者坐標 (x, y) 不在 $(0 \cdots$ image width $-1, 0 \cdots$ image number $-1)$ 範圍內，則 **write_imagef**、**write_imagei** 和 **write_imageui** 的行為未定義。

6.12.14.5 內建圖像查詢函式

下列內建函式可用於查詢圖像資訊。

表 6.29α　內建圖像查詢函式

函式	描述

`int get_image_width (image1d_t image)`
`int get_image_width (`
`  image1d_buffer_t image)`
`int get_image_width (image2d_t image)`
`int get_image_width (image3d_t image)`
`int get_image_width (`
`  image1d_array_t image)`
`int get_image_width (`
`  image2d_array_t image)`
	返回圖像寬度，單位像素。

`int get_image_height (image2d_t image)`
`int get_image_height (image3d_t image)`
`int get_image_height (`
`  image2d_array_t image)`
	返回圖像高度，單位像素。

`int get_image_depth (image3d_t image)`	返回圖像深度，單位像素。

表 6.29β 內建圖像查詢函式

`int get_image_channel_data_type ( image1d_t image)`
`int get_image_channel_data_type ( image1d_buffer_t image)`
`int get_image_channel_data_type ( image2d_t image)`
`int get_image_channel_data_type ( image3d_t image)`
`int get_image_channel_data_type ( image1d_array_t image)`
`int get_image_channel_data_type ( image2d_array_t image)`
	返回通道數據類型，有效值有:
	• CLK_SNORM_INT8
	• CLK_SNORM_INT16
	• CLK_UNORM_INT8
	• CLK_UNORM_INT16
	• CLK_UNORM_SHORT_565
	• CLK_UNORM_SHORT_555
	• CLK_UNORM_SHORT_101010
	• CLK_SIGNED_INT8
	• CLK_SIGNED_INT16
	• CLK_SIGNED_INT32
	• CLK_UNSIGNED_INT8
	• CLK_UNSIGNED_INT16
	• CLK_UNSIGNED_INT32
	• CLK_HALF_FLOAT
	• CLK_FLOAT

`int get_image_channel_order ( image1d_t image)`
`int get_image_channel_order ( image1d_buffer_t image)`
`int get_image_channel_order ( image2d_t image)`
`int get_image_channel_order ( image3d_t image)`
`int get_image_channel_order ( image1d_array_t image)`
`int get_image_channel_order ( image2d_array_t image)`
	返回通道順序，有效值有:
	• CLK_A
	• CLK_R
	• CLK_Rx
	• CLK_RG
	• CLK_RGx
	• CLK_RA
	• CLK_RGB
	• CLK_RGBx
	• CLK_RGBA
	• CLK_ARGB
	• CLK_BGRA
	• CLK_INTENSITY
	• CLK_LUMINANCE

`int2 get_image_dim (image2d_t image)` `int2 get_image_dim ( image2d_array_t image)`	將 2D 圖像的寬度和高度存入 int2 返回。其中組件 *x* 是寬度，組件 *y* 是高度。
`int4 get_image_dim (image3d_t image)`	將 3D 圖像的寬度、高度和深度存入 int4 返回。其中組件 *x* 是寬度，組件 *y* 是高度，組件 *z* 是深度，組件 *w* 是 0。
`size_t get_image_array_size( image2d_array_t image)`	返回 2D 圖像陣列中圖像的個數。
`size_t get_image_array_size( image1d_array_t image)`	返回 1D 圖像陣列中圖像的個數。

表 *6.29*中，**get_image_channel_data_type** 和 **get_image_channel_order** 所返回的帶有前綴 CLK_ 的值分別對應於表 *5.7*和表 *5.6*中帶有前綴 CL_ 的值。例如，CL_UNORM_INT8 和 CLK_UNORM_INT8 都是指通道數據類型為非歸一化的 8 位整數。

表 *6.30*中列出了圖像元素的各通道的顏色值與 float4、int4 或 uint4 中組件的映射關係，這些矢量由 **read_image{f|i|ui}** 返回或作為 **write_image{f|i|ui}** 的參數 *color*。對於未映射的組件，如果是紅、綠、藍幾個通道，則將其值置為 0.0，而如果是 alpha 通道，則將其值置為 1.0。

表 6.30α 矢量組件與圖像通道的對應關係

通道順序	矢量組件中的通道數據
CL_R、CL_Rx	(r, 0.0, 0.0, 1.0)
CL_A	(0.0, 0.0, 0.0, a)
CL_RGB、CL_RGBx	(r, g, 0.0, 1.0)
CL_RA	(r, 0.0, 0.0, a)
CL_RG、CL_RGx	(r, g, b, 1.0)
CL_RGBA、CL_BGRA、CL_ARGB	(r, g, b, a)

表 6.30β 矢量組件與圖像通道的對應關係

CL_INTENSITY	(I, I, I, I)
CL_LUMINANCE	(L, L, L, 1.0)

如果內核對多個圖像使用同一個尋址模式為 CL_ADDRESS_CLAMP 的採樣器，則可能導致實作內部使用額外的採樣器。如果通過 **read_image{f | i | ui}** 對多個圖像使用同一採樣器，則實作可能需要分配額外的採樣器來處理不同的顏色極值（這取決於所用的圖像格式）。在計算设备所支持採樣器的最大數目時（CL_DEVICE_MAX_SAMPLERS），會將這些實作自行分配的採樣器考慮在內。如果所入隊的內核需要的採樣器超過了這個最大值，則會導致返回 CL_OUT_OF_RESOURCES。

第 7 章
OpenCL 數值符合性

本章將描述 C99 和 IEEE 754 的一些特性，所有符合 OpenCL 的设备都必須支持這些特性。

本章所描述的功能都是針對單精度浮點數的。目前，只要求支持單精度浮點數，而雙精度浮點數是一個可選特性。

节 7.1 捨入模式

设备內部可能以更高的精度執行讀點運算，然後捨入成目標型別。 IEEE 754 中定義了四種捨入模式:

- 捨入為最近偶數
- 向 $+\infty$ 捨入
- 向 $-\infty$ 捨入
- 向零捨入

目前，OpenCL 規範對單、雙精度浮點數只要求支持捨入為最近偶數，因此他也是缺省的捨入模式。

另外，僅支持靜態選擇捨入模式。不支持 IEEE 754 中對捨入模式的重新配置。

节 7.2 INF、NaN 以及去規格化數

INF 和 NaN 是必須要支持的。但對 sNaN 不作要求。

對去規格化單精度浮點數的支持是可選的。對於加、減、乘、除，以及節 *6.12.2*、節 *6.12.4*和節 *6.12.5*中定義的函式，去規格化單精度浮點數無論作為輸入還是輸出都有可能被刷成零。

节 7.3 浮點異常

浮點異常在 OpenCL 中被去能了。浮點異常的結果必須與 IEEE 754 中未使能異常的情況保持一致。至於實作是否以及如何設置浮點標誌或引發異常依賴於具體實作。對於查詢、清除或設置浮點標誌以及為異常設陷，本規範中沒有提供任何方法。由於陷阱機制的性能問題、不可移植，以及在矢量上下文中提供精準異常的不切實際 (尤其是在異構硬件上)，即使具備這樣的特性，也不鼓勵使用。

然而，如果實作通過對標誌的擴展支持這種操作，那麼初始化時應當將所有異常標誌清除，並將異常掩碼置位，這樣當算術運算引發異常時就不會導致陷入。然而，如果實作重用了下層的作業項，在進入內核前，實作不負責重新清除異常標誌或將異常掩碼重置為缺省值。也就是說，如果內核不檢查異常標誌，或者不使能陷阱，那麼他就可以期望算術運算不會觸發陷阱。而對於那些會檢查異常標誌或使能陷阱的內核，在將控制權返還給實作時，他們要自己負責清除異常標誌以及去能所有陷阱。是否以及何時重用下層的作業項 (以及伴隨的全局浮點狀態，如果有的話) 依賴於具體實作。

本規範中，算式 **math_errorhandling** 和 MATH_ERREXCEPT 被保留使用，還未定義。如果實作通過擴充本規範支持了浮點異常，則應該遵守 ISO / IEC 9899:TC2 定義 **math_errorhandling** 和 MATH_ERREXCEPT。

节 7.4 相對誤差即 ULP

本節中，我們將討論相對誤差 (定義為 ulp，即 units in the last place，浮點數間的最小間隔) 的最大值。整數和單精度浮點數間的加、減、乘、積和熔加以及轉換都符合 IEEE 754，因此可以正確捨入。浮點格式間的轉換以及節 *6.2.3*中的顯式轉換都必須正確捨入。

ULP 的定義如下:

如果 x 是位於兩個有限連續浮點數 a 和 b 之間的實數，並且與 a 和 b 都不相等，則 $ulp(x) = |b - a|$。否則 $ulp(x)$ 是這兩個離 x 最近且互不相等的有限浮點數間的距離。此外，$ulp(NaN)$ 就是 *NaN*。

歸因：此定義獲得了 *Jean-Michel Muller* 的認可，不過他對在零處的行為做了一點澄清。請參考 *ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5504.pdf*。

表 *7.1*[46]中以 ULP 的形式給出了單精度浮點算術運算的最小精確度。計算 ULP 值時參考的是無限精確的結果。其中 0 ulp 表示相應函式無需捨入。

[46] 內建數學函式 **lgamma** 和 **lgamma_r** 的 ULP 值目前還未定義。

表 7.1α　單精度內建數學函式的 ULP 值

函　式	最小精度——ULP 值
$x + y$	正確捨入
$x - y$	正確捨入
$x * y$	正確捨入
$1.0/y$	正確捨入
x/y	正確捨入
acos	<= 4 ulp
acospi	<= 5 ulp
asin	<= 4 ulp
asinpi	<= 5 ulp
atan	<= 5 ulp
atan2	<= 6 ulp
atanpi	<= 5 ulp
atan2pi	<= 6 ulp
acosh	<= 4 ulp
asinh	<= 4 ulp
atanh	<= 5 ulp
cbrt	<= 2 ulp
ceil	正確捨入
copysign	0 ulp
cos	<= 4 ulp
cosh	<= 4 ulp
cospi	<= 4 ulp
erfc	<= 16 ulp
erf	<= 16 ulp
exp	<= 3 ulp
exp2	<= 3 ulp
exp10	<= 3 ulp
expm1	<= 3 ulp
fabs	0 ulp
fdim	正確捨入
floor	正確捨入
fma	正確捨入
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	正確捨入
frexp	0 ulp
hypot	<= 4 ulp
ilogb	0 ulp
ldexp	正確捨入
log	<= 3 ulp
log2	<= 3 ulp
log10	<= 3 ulp
log1p	<= 2 ulp
logb	0 ulp
mad	所允許的任何值（無窮 ulp）

表 7.1β 單精度內建數學函式的 ULP 值

maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	<= 16 ulp
pown(x, y)	<= 16 ulp
powr(x, y)	<= 16 ulp
remainder	0 ulp
remquo	0 ulp
rint	正確捨入
rootn	<= 16 ulp
round	正確捨入
rsqrt	<= 2 ulp
sin	<= 4 ulp
sincos	正弦值和餘弦值都是 <= 4 ulp
sinh	<= 4 ulp
sinpi	<= 4 ulp
sqrt	正確捨入
tan	<= 5 ulp
tanh	<= 5 ulp
tanpi	<= 6 ulp
tgamma	<= 16 ulp
trunc	正確捨入
half_cos	<= 8192 ulp
half_divide	<= 8192 ulp
half_exp	<= 8192 ulp
half_exp2	<= 8192 ulp
half_exp10	<= 8192 ulp
half_log	<= 8192 ulp
half_log2	<= 8192 ulp
half_log10	<= 8192 ulp
half_powr	<= 8192 ulp
half_recip	<= 8192 ulp
half_rsqrt	<= 8192 ulp
half_sin	<= 8192 ulp
half_sqrt	<= 8192 ulp
half_tan	<= 8192 ulp
native_cos	依賴於具體實作
native_divide	依賴於具體實作
native_exp	依賴於具體實作
native_exp2	依賴於具體實作
native_exp10	依賴於具體實作
native_log	依賴於具體實作
native_log2	依賴於具體實作
native_log10	依賴於具體實作
native_powr	依賴於具體實作

表 7.1γ　單精度內建數學函式的 ULP 值

native_recip	依賴於具體實作
native_rsqrt	依賴於具體實作
native_sin	依賴於具體實作
native_sqrt	依賴於具體實作
native_tan	依賴於具體實作

表 7.2中以 ULP 的形式給出了雙精度浮點算術運算的最小精確度。計算 ULP 值時參考的是無限精確的結果。其中 0 ulp 表示相應函式無需捨入。

表 7.2α　雙精度內建數學函式的 ULP 值

函　式	最小精度——ULP 值
$x + y$	正確捨入
$x - y$	正確捨入
$x * y$	正確捨入
$1.0/y$	正確捨入
x/y	正確捨入
acos	<= 4 ulp
acospi	<= 5 ulp
asin	<= 4 ulp
asinpi	<= 5 ulp
atan	<= 5 ulp
atan2	<= 6 ulp
atanpi	<= 5 ulp
atan2pi	<= 6 ulp
acosh	<= 4 ulp
asinh	<= 4 ulp
atanh	<= 5 ulp
cbrt	<= 2 ulp
ceil	正確捨入
copysign	0 ulp
cos	<= 4 ulp
cosh	<= 4 ulp
cospi	<= 4 ulp
erfc	<= 16 ulp
erf	<= 16 ulp
exp	<= 3 ulp
exp2	<= 3 ulp
exp10	<= 3 ulp
expm1	<= 3 ulp
fabs	0 ulp
fdim	正確捨入
floor	正確捨入
fma	正確捨入
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	正確捨入
frexp	0 ulp
hypot	<= 4 ulp

表 7.2β 雙精度內建數學函式的 ULP 值

ilogb	0 ulp
ldexp	正確捨入
log	<= 3 ulp
log2	<= 3 ulp
log10	<= 3 ulp
log1p	<= 2 ulp
logb	0 ulp
mad	所允許的任何值（無窮 ulp）
maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	<= 16 ulp
pown(x, y)	<= 16 ulp
powr(x, y)	<= 16 ulp
remainder	0 ulp
remquo	0 ulp
rint	正確捨入
rootn	<= 16 ulp
round	正確捨入
rsqrt	<= 2 ulp
sin	<= 4 ulp
sincos	正弦值和餘弦值都是 <= 4 ulp
sinh	<= 4 ulp
sinpi	<= 4 ulp
sqrt	正確捨入
tan	<= 5 ulp
tanh	<= 5 ulp
tanpi	<= 6 ulp
tgamma	<= 16 ulp
trunc	正確捨入

节 7.5 邊界條件下的行為

數學函式（節 *6.12.2*）在邊界條件下的行為要符合《ISO/IEC 9899: TC2》（通常稱作 C99, TC2）中的節 *F.9* 和節 *G.6*，節 *7.5.1*中所列之處除外。

7.5.1 C99 TC2 之外的附加要求

如果函式會返回 NaN，且有多個算元為 NaN，則應當返回其中一個 NaN 算元。如果是 sNaN，會返回 NaN 算元的函式可能會使其消音。非 sNaN 轉換後應當還是非 sNaN。 sNaN 轉換後應當是 NaN，但可能被轉換成非 sNaN。而 NaN 的淨荷位和正負號如何轉換則未定義。

函式 **half_<funcname>** 所起作用與不帶前綴 **half_** 的同名函式一樣。他們必須符合同一邊界條件需求（參見《C99, TC2》中的節 *F.9* 和節 *G.6*）。對於其他情況，除了明確提到的地方，准許單精度函式的最大誤差為 8192 ulp（以單精度結果來衡量），儘管鼓勵更高的精確度。

對於那些明確規定了其結果的值（參見《C99, TC2》中的節 *F.9*，或者節 *7.5.1*以及後面的節 *7.5.3*）（如：`ceil(-1 < x < 0)` 會返回 `-0`），捨入誤差（節 *7.4*）或刷新行為（節 *7.5.3*）的容差不會起作用。那些值肯定產生規定的答案，不會是其他的。使用 ± 的地方，正負號會保留下來。例如，$sin(\pm 0) = \pm 0$ 的意思就是：$sin(+0) = +0$ 以及 $sin(-0) = -0$。

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 (±∞) = ±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 (±y, -∞) returns ± 1 for finite y > 0.

atan2pi (±y, +∞) returns ± 0 for finite y > 0.

atan2pi (±∞, x) returns ± 0.5 for finite x.

atan2pi (±∞, -∞) returns ±0.75.

atan2pi (±∞, +∞) 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 (±∞) returns a NaN.

exp10 (±0) returns 1.

exp10 (-∞) returns +0.

exp10 (+∞) returns +∞.

distance (x, y) calculates the distance from x to y without overflow or extraordinary precision loss due to underflow.

fdim (any, NaN) returns NaN.

fdim (NaN, any) returns NaN.

fmod (±0, NaN) returns NaN.

frexp (±∞, exp) returns ±∞ and stores 0 in exp.

frexp (NaN, exp) returns the NaN and stores 0 in exp.

fract (x, iptr) shall not return a value greater than or equal to 1.0, and shall not return a value less than 0.

fract (+0, iptr) returns +0 and +0 in iptr.

fract (-0, iptr) returns -0 and -0 in iptr.

fract (+inf, iptr) returns +0 and +inf in iptr.

fract (-inf, iptr) returns -0 and -inf in iptr.

fract (NaN, iptr) returns the NaN and NaN in iptr.

length calculates the length of a vector without overflow or extraordinary precision loss due to underflow.

lgamma_r (x, signp) returns 0 in signp if x is zero or a negative integer.

nextafter (-0, y > 0) returns smallest positive denormal value.

nextafter (+0, y < 0) returns smallest negative denormal value.

normalize shall reduce the vector to unit length, pointing in the same direction without overflow or extraordinary precision loss due to underflow.

normalize (v) returns v if all elements of v are zero.

normalize (v) returns a vector full of NaNs if any element is a NaN.

normalize (v) for which any element in v is infinite shall proceed as if the elements in v were replaced as follows:

```
1 for( i = 0; i < sizeof(v) / sizeof(v[0] ); i++ )
2         v[i] = isinf(v[i] ) ? copysign(1.0, v[i]) : 0.0 * v [i];
```

pow (±0, -∞) returns +∞

pown (x, 0) is 1 for any x, even zero, NaN or infinity.

pown (±0, n) is ±∞ for odd n < 0.

pown (±0, n) is +∞ for even n < 0.

pown (±0, n) is +0 for even n > 0.

pown (±0, n) is ±0 for odd n > 0.

powr (x, ±0) is 1 for finite x > 0.

powr (±0, y) is +∞ for finite y < 0.

powr (±0, -∞) is +∞.

powr (±0, y) is +0 for y > 0.

powr (+1, y) is 1 for finite y.

powr (x, y) returns NaN for x < 0.

powr (±0, ±0) returns NaN.

powr (+∞, ±0) returns NaN.

powr (+1, ±∞) returns NaN.

powr (x, NaN) returns the NaN for x >= 0.

powr (NaN, y) returns the NaN.

rint (-0.5 <= x < 0) returns -0.

remquo (x, y, &quo) returns a NaN and 0 in quo if x is ±∞, or if y is 0 and the other argument is non-NaN or if either argument is a NaN.

rootn (±0, n) is ±∞ for odd n < 0.

rootn (±0, n) is +∞ 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 (±∞) returns a NaN.

tanpi (±0) returns ±0.

tanpi (±∞) 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 +∞ where n + 0.5 is representable.

tanpi (n + 0.5) for odd integer n is -∞ where n + 0.5 is representable.

trunc (-1 < x < 0) returns -0.

7.5.2 對 C99 TC2 的行為做出的改變

modf 的作用如同如下實作:

```
1  gentype modf(gentype value, gentype *iptr)
2  {
3          *iptr = trunc(value);
4          return copysign(isinf( value ) ? 0.0 : value - *iptr, value);
5  }
```

rint 始終捨入為最近偶數, 即使其調用者處於其他捨入模式下。

7.5.3 Flush-To-Zero 模式中，邊界條件下的行為

如果去規格化數被刷成了零, 則函式可能返回下列結果之一:

1. 任何符合 non-flush-to-zero 模式的結果;
2. 如果捨入前, 第 *1*項所給出的結果是次規格化數, 則可能會將其刷成零;
3. 如果有一個或多個次規格化算元被刷成了零, 則結果為符合此函式的任意未刷新的值。
4. 如果捨入前, 第 *3*項的結果是次規格化數, 則可能將其刷成零。

在上述任一情況中, 如果某個算元或結果被刷成了零, 零的正負號未定義。

如果次規格化數被刷成了零, 則设备可能選擇使用下列方式來處理 **nextafter** 的邊界情況, 以取代節 *7.5.1*中的那些:

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.

清晰起見，將次規格化數或去規格化數定義為位於區間 `0 < x < TYPE_MIN` 和 `-TYPE_MIN < x < -0` 內的一組可表示的數。他們不包括 ±0。如果在捨入前，一個非零數規格化後，其以 2 為底的指數小於 `(TYPE_MIN_EXP - 1)`，就說他是次規格化數。[47]

[47] 此處要用相應浮點型別常量來替換 `TYPE_MIN` 和 `TYPE_MIN_EXP`，如對於 `float` 這兩個常量就應該是 `FLT_MIN` 和 `FLT_MIN_EXP`。

第 8 章
圖像尋址和濾波

分別用 w_t、h_t 和 d_t 表示圖像的寬度、高度 (或者 1D 圖像陣列的陣列大小) 和深度 (或 2D 圖像陣列的陣列大小)，單位均為像素。用 $coord.xy$ (或者 (s,t)) 或 $coord.xyz$ (或者 (s,t,r)) 表示 **read_image{f|i|ui}** 所用坐標。 **read_image{f|i|ui}** 中的採樣器用來確定如何對圖像採樣並返回恰當的顏色。

节 8.1 圖像坐標

這影響着對圖像坐標的解釋。如果 **read_image{f|i|ui}** 所用圖像坐標是歸一化的 (採樣器中會指定)，則將 s、t 和 r 的值分別乘以 w_t、h_t 和 d_t 就可以生成非歸一化的坐標值。

用 (u,v,w) 表示非歸一化的坐標值。

节 8.2 尋址模式和濾波模式

我們先來描述尋址模式既不是 CLK_ADDRESS_REPEAT 也不是 CLK_ADDRESS_MIRRORED_REPEAT 的情況下，怎樣利用尋址模式和濾波模式來生成恰當的採樣位置以讀取圖像。

生成圖像坐標 (u,v,w) 後，我們會使用恰當的尋址模式和濾波模式來生成恰當的採樣區以讀取圖像。

如果 (u,v,w) 中的值有 INF 或 NaN，則 **read_image{f|i|ui}** 的行為未定義。

Filter Mode = CLK_FILTER_NEAREST

如果濾波模式為 CLK_FILTER_NEAREST，則會得到離 (u,v,w) 最近 (Manhattan 距離) 的圖像元素。即返回坐標為 (i,j,k) 的元素，其中

```
1  i = address_mode((int)floor(u))
2  j = address_mode((int)floor(v))
3  k = address_mode((int)floor(w))
```

對於 3D 圖像，(i,j,k) 處的圖像元素即為所求。而對於 2D 圖像，(i,j) 處的圖像元素即為所求。

*表 8.1*中描述了函式 **address_mode**。

表 8.1 用來生成紋理位置的尋址模式

尋址模式	address_mode(coord) 的結果
CLK_ADDRESS_CLAMP_TO_EDGE	clamp (coord, 0, size − 1)
CLK_ADDRESS_CLAMP	clamp (coord, -1, size)
CLK_ADDRESS_NONE	coord

對於 u、v 和 w 而言，*表 8.1*中的 size 分別為 w_t、h_t 和 d_t。

*表 8.1*中的 clamp 定義為:

```
1  clamp(a, b, c) = return (a < b) ? b : ((a > c) ? c : a)
```

如果紋理位置 (i,j,k) 落到了圖像的外面，則用顏色極值作為此紋理的顏色。

Filter Mode = CLK_FILTER_LINEAR

如果濾波模式為 CLK_FILTER_LINEAR，則對於 2D 圖像會選擇一個 2×2 的方陣中的圖像元素，而對於 3D 圖像，則會選擇一個 $2 \times 2 \times 2$ 的立方體中的圖像元素。得到的 2×2 方陣或 $2 \times 2 \times 2$ 立方體如下所示。

設

```
1  i0 = address_mode((int)floor(u - 0.5))
2  j0 = address_mode((int)floor(v - 0.5))
3  k0 = address_mode((int)floor(w - 0.5))
4  i1 = address_mode((int)floor(u - 0.5) + 1)
5  j1 = address_mode((int)floor(v - 0.5) + 1)
6  k1 = address_mode((int)floor(w - 0.5) + 1)
7  a  = frac(u − 0.5)
8  b  = frac(v − 0.5)
```

```
c  = frac(w - 0.5)
```

其中 frac(x) 為 x 的小數部分，相當於 x - floor(x)。

對於 3D 圖像，用如下方式得到圖像元素:

```
T = (1 - a) * (1 - b) * (1 - c) * T_i0j0k0
    + a * (1 - b) * (1 - c) * T_i1j0k0
    + (1 - a) * b * (1 - c) * T_i0j1k0
    + a * b * (1 - c) * T_i1j1k0
    + (1 - a) * (1 - b) * c * T_i0j0k1
    + a * (1 - b) * c * T_i1j0k1
    + (1 - a) * b * c * T_i0j1k1
    + a * b * c * T_i1j1k1
```

其中 T_{ijk} 就是此 3D 圖像中位置 (i, j, k) 處的元素。

對於 2D 圖像，用如下方式得到圖像元素:

```
T = (1 - a) * (1 - b) * T_i0j0
    + a * (1 - b) * T_i1j0
    + (1 - a) * b * T_i0j1
    + a * b * T_i1j1
```

其中 T_{ij} 就是此 2D 圖像中位置 (i, j) 處的元素。

上面方程中，如果 T_{ijk} 或 T_{ij} 中任意一個所指代的位置落到了圖像外面，則用顏色極值作為 T_{ijk} 或 T_{ij} 處的顏色。

現在我們來討論尋址模式是 CLK_ADDRESS_REPEAT 的情況下，怎樣利用尋址模式和濾波模式來生成恰當的採樣位置以讀取圖像。

如果 (s, t, r) 中的值有 INF 或 NaN，則內建圖像讀取函式的行為未定義。

Filter Mode = CLK_FILTER_NEAREST

如果濾波模式為 CLK_FILTER_NEAREST，則使用位置 (i, j, k) 處的元素，其中 i、j 和 k 的計算方式如下:

```
u = (s - floor(s)) * w_t
i = (int)floor(u)
if (i > w_t - 1)
        i = i - w_t

v = (t - floor(t)) * h_t
j = (int)floor(v)
if (j > h_t - 1)
        j = j - h_t

w = (r - floor(r)) * d_t
k = (int)floor(w)
if (k > d_t - 1)
        k = k - d_t
```

對於 3D 圖像，(i, j, k) 處的圖像元素即為所求。而對於 2D 圖像，(i, j) 處的圖像元素即為所求。

Filter Mode = CLK_FILTER_LINEAR

如果濾波模式為 CLK_FILTER_LINEAR，則對於 2D 圖像會選擇一個 2×2 的方陣中的圖像元素，而對於 3D 圖像，則會選擇一個 $2 \times 2 \times 2$ 的立方體中的圖像元素。得到的 2×2 方陣或 $2 \times 2 \times 2$ 立方體如下所示。

設

```
u = (s - floor(s)) * w_t
i0 = (int)floor(u - 0.5)
i1 = i0 + 1
if (i0 < 0)
        i0 = w_t + i0
```

```
if (i1 > w_t - 1)
        i1 = i1 - w_t

v = (t - floor(t)) * h_t
j0 = (int)floor(v - 0.5)
j1 = j0 + 1
if (j0 < 0)
        j0 = h_t + j0
if (j1 > h_t - 1)
        j1 = j1 - h_t

w = (r - floor(r)) * d_t
k0 = (int)floor(w - 0.5)
k1 = k0 + 1
if (k0 < 0)
        k0 = d_t + k0
if (k1 > dt - 1)
        k1 = k1 - d_t

a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)
```

其中 `frac(x)` 為 x 的小數部分，相當於 `x - floor(x)`。

對於 3D 圖像，用如下方式得到圖像元素：

```
T = (1 - a) * (1 - b) * (1 - c) * T_i0j0k0
    + a * (1 - b) * (1 - c) * T_i1j0k0
    + (1 - a) * b * (1 - c) * T_i0j1k0
    + a * b * (1 - c) * T_i1j1k0
    + (1 - a) * (1 - b) * c * T_i0j0k1
    + a * (1 - b) * c * T_i1j0k1
    + (1 - a) * b * c * T_i0j1k1
    + a * b * c * T_i1j1k1
```

其中 T_{ijk} 就是此 3D 圖像中位置 (i, j, k) 處的元素。

對於 2D 圖像，用如下方式得到圖像元素：

```
T = (1 - a) * (1 - b) * T_i0j0
    + a * (1 - b) * T_i1j0
    + (1 - a) * b * T_i0j1
    + a * b * T_i1j1
```

其中 T_{ij} 就是此 2D 圖像中位置 (i, j) 處的元素。

現在我們來討論尋址模式為 `CLK_ADDRESS_MIRRORED_REPEAT` 的情況下，怎樣利用尋址模式和濾波模式來生成恰當的採樣位置以讀取圖像。這種情況下讀取圖像時，就如同圖像數據在整數處會翻轉平鋪一樣。例如，2 和 3 之間的坐標 (s, t, r) 如同從 1 降到 0 的坐標一樣。如果 (s, t, r) 中的值有 INF 或 NaN，則內建圖像讀取函式的行為未定義。

Filter Mode = CLK_FILTER_NEAREST

如果濾波模式為 `CLK_FILTER_NEAREST`，則使用位置 (i, j, k) 處的元素，其中 i、j 和 k 的計算方式如下：

```
s' = 2.0f * rint(0.5f * s)
s' = fabs(s - s')
u = s' * w_t
i = (int)floor(u)
i = min(i, w_t - 1)

t' = 2.0f * rint(0.5f * t)
t' = fabs(t - t')
v = t' * h_t
```

```
j = (int)floor(v)
j = min(j, h_t - 1)

r' = 2.0f * rint(0.5f * r)
r' = fabs(r - r')
w = r' * d_t
k = (int)floor(w)
k = min(k, d_t - 1)
```

對於 3D 圖像，(i, j, k) 處的圖像元素即為所求。而對於 2D 圖像，(i, j) 處的圖像元素即為所求。

Filter Mode = CLK_FILTER_LINEAR

如果濾波模式為 CLK_FILTER_LINEAR，則對於 2D 圖像會選擇一個 2×2 的方陣中的圖像元素，而對於 3D 圖像，則會選擇一個 $2 \times 2 \times 2$ 的立方體中的圖像元素。得到的 2×2 方陣或 $2 \times 2 \times 2$ 立方體如下所示。

設

```
s' = 2.0f * rint(0.5f * s)
s' = fabs(s - s')
u = s' * w_t
i0 = (int)floor(u - 0.5f)
i1 = i0 + 1
i0 = max(i0, 0)
i1 = min(i1, w_t - 1)

t' = 2.0f * rint(0.5f * t)
t' = fabs(t - t')
v = t' * h_t
j0 = (int)floor(v - 0.5f)
j1 = j0 + 1
j0 = max(j0, 0)
j1 = min(j1, h_t - 1)

r' = 2.0f * rint(0.5f * r)
r' = fabs(r - r')
w = r' * d_t
k0 = (int)floor(w - 0.5f)
k1 = k0 + 1
k0 = max(k0, 0)
k1 = min(k1, d_t - 1)

a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)
```

其中 frac(x) 為 x 的小數部分，相當於 x - floor(x)。

對於 3D 圖像，用如下方式得到圖像元素:

```
T = (1 - a) * (1 - b) * (1 - c) * T_i0j0k0
    + a * (1 - b) * (1 - c) * T_i1j0k0
    + (1 - a) * b * (1 - c) * T_i0j1k0
    + a * b * (1 - c) * T_i1j1k0
    + (1 - a) * (1 - b) * c * T_i0j0k1
    + a * (1 - b) * c * T_i1j0k1
    + (1 - a) * b * c * T_i0j1k1
    + a * b * c * T_i1j1k1
```

其中 T_{ijk} 就是此 3D 圖像中位置 (i, j, k) 處的元素。

對於 2D 圖像，用如下方式得到圖像元素:

```
T = (1 - a) * (1 - b) * T_i0j0
```

```
    + a * (1 - b) * T_i1j0
    + (1 - a) * b * T_i0j1
    + a * b * T_i1j1
```

其中 T_{ij} 就是此 2D 圖像中位置 (i, j) 處的元素。

注意:

如果在採樣器中指明使用的是非歸一化坐標（浮點數或整數坐標），濾波模式為 CLK_FILTER_NEAREST，尋址模式為 CLK_ADDRESS_NONE、CLK_ADDRESS_CLAMP_TO_EDGE 或 CLK_ADDRESS_CLAMP，則本節中計算圖像元素的位置 (i, j, k) 時不會損失精度。

如果採樣器採用的是其他組合方式（歸一化或非歸一化坐標、濾波模式、尋址模式），則對於尋址模式的計算以及圖像濾波的運算而言，在 OpenCL 規範的這個修訂版中沒有定義其相對誤差或精度。為了確保任何 OpenCL 设备在進行圖像尋址和濾波計算時都至少具有一個最小精度，對於採樣器的這些組合方式，開發人員應當在內核中將坐標去歸一化，並使用由非歸一化坐標、濾波模式為 CLK_FILTER_NEAREST、尋址模式為 CLK_ADDRESS_NONE、CLK_ADDRESS_CLAMP_TO_EDGE 或 CLK_ADDRESS_CLAMP 組合而成的採樣器調用 **read_image{f|i|ui}**，最後對從圖像中讀到的顏色進行插值來生成經過濾波的顏色值。

节 8.3 轉換規則

本節中我們將討論內核中讀寫圖像時所用的轉換規則。

8.3.1 歸一化整數通道數據類型的轉換規則

本節我們將討論歸一化整數通道數據類型與浮點值之間的相互轉換。

8.3.1.1 將歸一化整數通道數據類型轉換為浮點值

如果通道數據類型為 CL_UNORM_INT8 和 CL_UNORM_INT16。**read_imagef** 將把 8 位或 16 位無符號整數轉換為歸一化浮點值，其所在區間為 $[0.0f \cdots 1.0]$。

如果通道數據類型為 CL_SNORM_INT8 和 CL_SNORM_INT16。**read_imagef** 將把 8 位或 16 位帶符號整數轉換為歸一化浮點值，其所在區間為 $[-1.0 \cdots 1.0]$。

轉換方式如下:

CL_UNORM_INT8（**8 位無符號整數**）**->** float

歸一化浮點值 = (float) c / 255.0f

CL_UNORM_INT_101010（**10 位無符號整數**）**->** float

歸一化浮點值 = (float) c / 1023.0f

CL_UNORM_INT16（**16 位無符號整數**）**->** float

歸一化浮點值 = (float) c / 65535.0f

CL_SNORM_INT8（**8 位帶符號整數**）**->** float

歸一化浮點值 = max(-1.0f, (float)c / 127.0f)

CL_SNORM_INT16（**16 位帶符號整數**）**->** float

歸一化浮點值 = max(-1.0f, (float)c / 32767.0f)

除了下面所列情況，上述轉換的精度均 $\leq 1.5ulp$:

對於 CL_UNORM_INT8

0 必須轉換為 0.0f

255 必須轉換為 1.0f

對於 CL_UNORM_INT_101010

0 必須轉換為 0.0f

1023 必須轉換為 1.0f

對於 CL_UNORM_INT16

0 必須轉換為 0.0f

65535 必須轉換為 1.0f

對於 CL_SNORM_INT8

−128 和 −127 必須轉換為 −1.0f 0 必須轉換為 0.0f

127 必須轉換為 1.0f

對於 CL_SNORM_INT16

−32768 和 −32767 必須轉換為 −1.0f

0 必須轉換為 0.0f

32767 必須轉換為 1.0f

8.3.1.2 將浮點值轉換為歸一化整數通道數據類型

如果通道數據類型為 CL_UNORM_INT8 和 CL_UNORM_INT16。 **write_imagef** 將把浮點顏色值轉換為 8 位或 16 位無符號整數。

如果通道數據類型為 CL_SNORM_INT8 和 CL_SNORM_INT16。 **write_imagef** 將把浮點顏色值轉換為 8 位或 16 位帶符號整數。

建議按如下方式轉換:

float **->** CL_UNORM_INT8 （**8 位無符號整數**）

convert_uchar_sat_rte(f * 255.0f)

float **->** CL_UNORM_INT_101010 （**10 位無符號整數**）

min(convert_ushort_sat_rte(f * 1023.0f), 0x3ff)

float **->** CL_UNORM_INT16 （**16 位無符號整數**）

convert_ushort_sat_rte(f * 65535.0f)

float **->** CL_SNORM_INT8 （**8 位帶符號整數**）

convert_char_sat_rte(f * 127.0f)

float **->** CL_SNORM_INT16 （**16 位帶符號整數**）

convert_short_sat_rte(f * 32767.0f)

對於溢出時的行為以及飽和轉換規則請參考節 *6.2.3.3*。

OpenCL 實作也可能選擇其他捨入模式來逼近上述轉換結果。如果使用的捨入模式不是捨入為最近偶數 (_rte)，則實作所產生的結果與捨入模式 _rte 所產生結果的絕對誤差必須 ≤ 0.6。

```
1  float -> CL_UNORM_INT8 (8-bit unsigned integer)
2          Let f_preferred = convert_uchar_sat_rte(f * 255.0f)
3          Let f_approx =
4                  convert_uchar_sat_<impl-rounding-mode>(f * 255.0f)
5          fabs(f_preferred − f_approx) must be <= 0.6
6  float -> CL_UNORM_INT_101010 (10-bit unsigned integer)
7          Let f_preferred = convert_ushort_sat_rte(f * 1023.0f)
8          Let f_approx =
9                  convert_ushort_sat_<impl-rounding-mode>(f * 1023.0f)
10         fabs(f_preferred − f_approx) must be <= 0.6
11 float -> CL_UNORM_INT16 (16-bit unsigned integer)
12         Let f_preferred = convert_ushort_sat_rte(f * 65535.0f)
13         Let f_approx =
14                 convert_ushort_sat_<impl-rounding-mode>(f * 65535.0f)
15         fabs(f_preferred − f_approx) must be <= 0.6
16 float -> CL_SNORM_INT8 (8-bit signed integer)
```

```
        Let f_preferred = convert_char_sat_rte(f * 127.0f)
        Let f_approx =
                convert_char_sat_<impl_rounding_mode>(f * 127.0f)
        fabs(f_preferred - fapprox}/ETEX) must be <= 0.6
float -> CL_SNORM_INT16 (16-bit signed integer)
        Let f_preferred = convert_short_sat_rte(f * 32767.0f)
        Let f_approx =
                convert_short_sat_<impl-rounding-mode>(f * 32767.0f)
        fabs(f_preferred - f_approx) must be <= 0.6
```

8.3.2 半精度浮點通道數據類型的轉換規則

如果圖像通道數據類型為 CL_HALF_FLOAT，則由 half 到 float 的轉換是無損的（參見節 *6.1.1.1*）。由 float 到 half 的轉換中，捨入尾數所用的捨入模式為捨入為最近偶數或向零捨入。型別為 half 的去規格化數（可能是將 float 轉換為 half 時生成）可能會被刷成零。型別為 float 的 NaN 必須轉換為型別為 half 的 NaN。型別為 float 的 INF 必須轉換為型別為 half 的 INF。

8.3.3 浮點通道數據類型的轉換規則

如果圖像通道數據類型為 CL_FLOAT，則對其讀寫時要遵守下列規則：

- NaN 必須轉換為设备所支持的 NaN 值。
- 可以將去規格化數刷成零。
- 所有其他值都必須保留。

8.3.4 帶符號或無符號的 8/16/32 位整數通道數據類型的轉換規則

如果圖像通道數據類型為 CL_SIGNED_INT8、CL_SIGNED_INT16 或 CL_SIGNED_INT32，則 **read_imagei** 會返回圖像中指定位置所存儲的原始整數值。

如果圖像通道數據類型為 CL_UNSIGNED_INT8、CL_UNSIGNED_INT16 或 CL_UNSIGNED_INT32，則 **read_imageui** 會返回圖像中指定位置所存儲的原始整數值。

write_imagei 會進行下列轉換：

```
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
```

write_imageui 會進行下列轉換：

```
32 bit signed integer -> 8-bit signed integer
       convert_uchar_sat(i)
32 bit signed integer -> 16-bit signed integer
       convert_ushort_sat(i)
32 bit signed integer -> 32-bit signed integer
       no conversion is performed
```

對於本節所描述的轉換，必須使其正確飽和。

节 8.4 在圖像陣列中選擇圖像

讀寫 2D 圖像陣列中的 2D 圖像時，將非歸一化圖像坐標值記為 (u, v, w)。

選擇 2D 圖層時這樣計算：

```
layer = clamp(floor(w + 0.5f), 0, d_t - 1)
```

讀寫 1D 圖像陣列中的 1D 圖像時，將非歸一化圖像坐標值記為 (u, v)。

選擇 2D 圖層時這樣計算：

```
layer = clamp(floor(v + 0.5f), 0, h_t - 1)
```

第 9 章
可選擴展

OpenCL 1.2 所支持的可選特性在《OpenCL 1.2 擴展規範》中有所描述。

第 10 章
OpenCL 嵌入式規格

前面章節描述了對桌面平台的特性要求。本節則描述針對手持或嵌入式平台的 OpenCL 1.2 嵌入式規格，此規格是完整規格的一個子集。《OpenCL 1.2 擴展規範》中定義的可選擴展對兩種規格都有效。

OpenCL 1.2 嵌入式規格有如下局限：

1. 64 位整數，即 `long`、`ulong`，包括相應的矢量數據型別，以及相關的運算都是可選的。如果嵌入式規格的實作支持 64 位整數，則報告中有擴展字串 **cles_khr_int64**[48]。
2. 對 3D 圖像的支持是可選的。是否支持 3D 圖像主要取決於下列值：
   - `CL_DEVICE_IMAGE3D_MAX_WIDTH`、
   - `CL_DEVICE_IMAGE3D_MAX_HEIGHT` 和
   - `CL_DEVICE_IMAGE3D_MAX_DEPTH`。

   如果這些值為零，則嵌入式規格中調用 **clCreateImage** 創建 3D 圖像將失敗，並在引數 *errcode_ret* 中返回 `CL_INVALID_OPERATION`；而且在內核中聲明型別為 `image3d_t` 的引數將會導致編譯錯誤。

   如果這些值大於零，則表明嵌入式規格的實作是支持 3D 圖像的。**clCreateImage** 會和完整規格中定義的那樣正常工作。內核中可以使用數據型別 `image3d_t`。
3. 如果創建圖像和圖像陣列是所用的 `image_channel_data_type` 為 `CL_FLOAT` 或 `CL_HALF_FLOAT`，則他們只能使用濾波模式為 `CL_FILTER_NEAREST` 的採樣器。如果 2D 和 3D 圖像的 `image_channel_data_type` 為 `CL_FLOAT` 或 `CL_HALF_FLOAT`，而採樣器的 `filter_mode` 為 `CL_FILTER_LINEAR`，則 **read_imagef** 和 **read_imageh**[49] 的返回值未定義。
4. 對於圖像和圖像陣列，支持下列採樣器尋址模式：`CLK_ADDRESS_NONE`、`CLK_ADDRESS_MIRRORED_REPEAT`、`CLK_ADDRESS_REPEAT`、`CLK_ADDRESS_CLAMP_TO_EDGE`、`CLK_ADDRESS_CLAMP`。
5. 規定單精度浮點能力（由 `CL_DEVICE_SINGLE_FP_CONFIG` 給出）至少要為 `CL_FP_ROUND_TO_ZERO` 或 `CL_FP_ROUND_TO_NEAREST`。如果支持 `CL_FP_ROUND_TO_NEAREST`，則缺省的捨入模式為捨入為最近偶數，否則缺省的捨入模式為向零捨入。
6. 單精度浮點運算（加、減和乘）必須要能正確捨入。如果結果是零，則可能始終是 0.0。除法和開方的精確度由表 *10.1*給出。

   如果 `CL_DEVICE_SINGLE_FP_CONFIG` 中沒有設置 `CL_FP_INF_NAN`，但是某個算元或加、減、乘、除的結果引發了上溢或無效異常（參見 IEEE 754 規範），則結果的值依賴於具體實作。同樣，如果某個算元是 NaN，則單精度比較算子（`<`、`>`、`<=`、`>=`、`==`、`!=`）所返回的值也依賴於具體實作。

   所有情況下，轉換（節 *6.2*和節 *6.12.7*）都要像 `FULL_PROFILE` 一樣正確捨入，包括那些會小號或產生 INF 或 NaN 的轉換。內建數學函式（節 *6.12.2*）的表現也與 `FULL_PROFILE` 中描述的一樣，包括節 *7.5.1*中所描述的邊界條件下的行為，但是精確度則以表 *10.1*為准。

   > 如果減法和乘法缺省為向零捨入，**fract**、**fma** 和 **fdim** 所產生的結果應當已經按此模式正確捨入了。

   對於基本的浮點運算，上述內容實際是放寬了 IEEE 754 中的要求，儘管極度不情願，但是這樣可以為那些硬件預算有嚴格限制的嵌入式設備提供更大的靈活性。
7. 對於由 `CL_UNORM_INT8`、`CL_SNORM_INT8`、`CL_UNORM_INT16` 和 `CL_SNORM_INT16` 到 `float` 的轉換，在嵌入式規格中要求其精度 $\leq 2ulp$，這取代了節 *8.3.1.1*中的 $\leq 1.5ulp$。在嵌入式規格中，節 *8.3.1.1*中的異常情況以及下列所給處的異常情況都有效。

```
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
```

[48] 在不同的嵌入式設備上，64 位整數算術的性能可能有重大差異。
[49] 如果支持擴展 **cl_khr_fp16**。

```
        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
For CL_UNORM_INT_101010
        0 must convert to 0.0f and
        1023 must convert to 1.0f
```

8. 節 *6.12.11*中所定義的原子函式是可選的。

《OpenCL 1.2 擴展規範》中所定義的下列可選擴展對嵌入式規格同樣可用:

- **cl_khr_int64_base_atomics**
- **cl_khr_int64_extended_atomics**
- **cl_khr_fp16**
- **cles_khr_int64**。如果支持雙精度浮點數，即 `CL_DEVICE_DOUBLE_FP_CONFIG` 不是零，則必須支持 **cles_khr_int64**。

OpenCL 1.0 和 OpenCL 1.1 規範中所定義的下列擴展（第 *9* 章）對嵌入式規格同樣可用:

- **cl_khr_global_int32_base_atomics**
- **cl_khr_global_int32_extended_atomics**
- **cl_khr_local_int32_base_atomics**
- **cl_khr_local_int32_extended_atomics**

表 *10.1*中描述了嵌入式規格中，單精度浮點算術運算的最小精確度，以 ULP 值的形式給出。計算 ULP 值時參考的是無限精確的結果。其中 0 ulp 表示相應函式無需捨入。

表 10.1α　內建數學函式的 ULP 值

函　式	最小精度——ULP 值
$x + y$	正確捨入
$x - y$	正確捨入
$x * y$	正確捨入
$1.0/x$	<= 3 ulp
x/y	<= 3 ulp
acos	<= 4 ulp
acospi	<= 5 ulp
asin	<= 4 ulp
asinpi	<= 5 ulp
atan	<= 5 ulp
atan2	<= 6 ulp
atanpi	<= 5 ulp
atan2pi	<= 6 ulp
acosh	<= 4 ulp
asinh	<= 4 ulp
atanh	<= 5 ulp
cbrt	<= 4 ulp
ceil	正確捨入
copysign	0 ulp
cos	<= 4 ulp
cosh	<= 4 ulp
cospi	<= 4 ulp

表 10.1β 內建數學函式的 ULP 值

erfc	<= 16 ulp
erf	<= 16 ulp
exp	<= 4 ulp
exp2	<= 4 ulp
exp10	<= 4 ulp
expm1	<= 4 ulp
fabs	0 ulp
fdim	正確捨入
floor	正確捨入
fma	正確捨入
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	正確捨入
frexp	0 ulp
hypot	<= 4 ulp
ilogb	0 ulp
ldexp	正確捨入
log	<= 4 ulp
log2	<= 4 ulp
log10	<= 4 ulp
log1p	<= 4 ulp
logb	0 ulp
mad	所允許的任何值（無窮 ulp）
maxmag	0 ulp
minmag	0 ulp
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
pow(x, y)	<= 16 ulp
pown(x, y)	<= 16 ulp
powr(x, y)	<= 16 ulp
remainder	0 ulp
remquo	0 ulp
rint	正確捨入
rootn	<= 16 ulp
round	正確捨入
rsqrt	<= 4 ulp
sin	<= 4 ulp
sincos	正弦值和餘弦值都是 <= 4 ulp
sinh	<= 4 ulp
sinpi	<= 4 ulp
sqrt	<= 4 ulp
tan	<= 5 ulp
tanh	<= 5 ulp
tanpi	<= 6 ulp
tgamma	<= 16 ulp

表 10.1γ　內建數學函式的 ULP 值

trunc	正確捨入
half_cos	<= 8192 ulp
half_divide	<= 8192 ulp
half_exp	<= 8192 ulp
half_exp2	<= 8192 ulp
half_exp10	<= 8192 ulp
half_log	<= 8192 ulp
half_log2	<= 8192 ulp
half_log10	<= 8192 ulp
half_powr	<= 8192 ulp
half_recip	<= 8192 ulp
half_rsqrt	<= 8192 ulp
half_sin	<= 8192 ulp
half_sqrt	<= 8192 ulp
half_tan	<= 8192 ulp
native_cos	依賴於具體實作
native_divide	依賴於具體實作
native_exp	依賴於具體實作
native_exp2	依賴於具體實作
native_exp10	依賴於具體實作
native_log	依賴於具體實作
native_log2	依賴於具體實作
native_log10	依賴於具體實作
native_powr	依賴於具體實作
native_recip	依賴於具體實作
native_rsqrt	依賴於具體實作
native_sin	依賴於具體實作
native_sqrt	依賴於具體實作
native_tan	依賴於具體實作

語言中加入了巨集 __EMBEDDED_PROFILE__（參見節 *6.10*）。對於那些實現了嵌入式規格的 OpenCL 设备，此巨集為整數常量 1，否則未定義此巨集。

如果 OpenCL 實作僅支持嵌入式規格，則表 *4.1*中的 CL_PLATFORM_PROFILE 會返回字串 EMBEDDED_PROFILE。

嵌入式規格中，表 *4.3*所指定的最小值和最大值變化如下：

cl_device_info	**返回型別**
CL_DEVICE_TYPE	cl_device_type
OpenCL 设备類型，當前支持: • CL_DEVICE_TYPE_CPU， • CL_DEVICE_TYPE_GPU， • CL_DEVICE_TYPE_ACCELERATOR， • CL_DEVICE_TYPE_DEFAULT，或者 • 以上值的组合，或者 • CL_DEVICE_TYPE_CUSTOM。	
CL_DEVICE_VENDOR_ID	cl_uint
唯一的设备供應商標識符。例如可以是 PCIe ID。	
CL_DEVICE_MAX_COMPUTE_UNITS	cl_uint
OpenCL 设备上的並行計算器件的數目。最小值是 1。	
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	cl_uint

數據並行編程模型中所用全局和局部作業項 ID 的最大維數（參見 **clEnqueueNDRangeKernel**）。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，其最小值是 3。	
CL_DEVICE_MAX_WORK_ITEM_SIZES	size_t[]
對**clEnqueueNDRangeKernel**而言，作業組中每個維度上可以指派作業項的最大數目。 返回 *n* 個型別為 size_t 的表項。其中 *n* 是查詢 CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS 時所返回的值。 對於不是 CL_DEVICE_TYPE_CUSTOM 的设备，最小值是$(1, 1, 1)$。	
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t
用數據並行編程模型執行內核時，作業組中所能容納作業項的最大數目（參見 **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_DEVICE_PREFERRED_VECTOR_WIDTH_HALF	cl_uint
可以放入矢量中的內建標量型別所期望的原生矢量的寬度。矢量寬度定義為可以可以容納標量元素的數目。 如果不支持雙精度浮點數，CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE 必須返回0。 如果不支持擴展 **cl_khr_fp16**，CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF 必須返回0。	
CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT CL_DEVICE_NATIVE_VECTOR_WIDTH_INT CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF	cl_uint
返回原生 ISA 矢量寬度。此矢量寬度定義為所能容納標量元素的數目。 如果不支持雙精度浮點數，CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE 必須返回0。 如果不支持擴展 **cl_khr_fp16**，CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF 必須返回0。	
CL_DEVICE_MAX_CLOCK_FREQUENCY	cl_uint
设备的時鐘頻率可以配置成的最大值，單位: MHz。	
CL_DEVICE_ADDRESS_BITS	cl_uint
計算設備的位址空間缺省大小，無符號整數，單位: bit。當前支持 32 位或 64 位。如果嵌入式規格報告的值是 64，則必須支持擴展 **cles_khr_int64**。	
CL_DEVICE_MAX_MEM_ALLOC_SIZE	cl_ulong
所能分配的內存對象大小的最大值，單位: 字節。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，此值最小為: $max($CL_DEVICE_GLOBAL_MEM_SIZE $* 1/4, 1 * 1024 * 1024)$	
CL_DEVICE_IMAGE_SUPPORT	cl_bool
如果 OpenCL 设备支持圖像，則為 CL_TRUE，否則為 CL_FALSE。	
CL_DEVICE_MAX_READ_IMAGE_ARGS	cl_uint
內核可以同時讀取多少圖像對象。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 8。	
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	cl_uint
內核可以同時寫入多少圖像對象。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 1。	
CL_DEVICE_IMAGE2D_MAX_WIDTH	size_t
2D 图像的最大寬度，單位: 像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 2048。	
CL_DEVICE_IMAGE2D_MAX_HEIGHT	size_t
2D 圖像的最大高度，單位: 像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 2048。	
CL_DEVICE_IMAGE3D_MAX_WIDTH	size_t
3D 圖像的最大寬度，單位: 像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 0。	
CL_DEVICE_IMAGE3D_MAX_HEIGHT	size_t
3D 圖像的最大高度，單位: 像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 0。	

CL_DEVICE_IMAGE3D_MAX_DEPTH	size_t
3D 圖像的最大深度，單位：像素。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 0。	
CL_DEVICE_IMAGE_MAX_BUFFER_SIZE	size_t
由緩衝對象所創建的 1D 圖像的最大像素數。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 2048。	
CL_DEVICE_IMAGE_MAX_ARRAY_SIZE	size_t
1D 或 2D 圖像陣列中圖像的最大數目。如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 256。	
CL_DEVICE_MAX_SAMPLERS	cl_uint
一個內核內最多可以使用多少個採樣器。關於採樣器的細節請參考節 *6.12.14*。 如果 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則此值至少是 8。	
CL_DEVICE_MAX_PARAMETER_SIZE	size_t
內核引數的最大字節數。 如果設備類型不是 CL_DEVICE_TYPE_CUSTOM，則此值至少要是 256。	
CL_DEVICE_MEM_BASE_ADDR_ALIGN	cl_uint
如果設備類型不是 CL_DEVICE_TYPE_CUSTOM，至少要是設备所支持的 OpenCL 內建數據型別中最大的那種的大小，單位：bit。（FULL 規格中是 long16，EMBEDDED 規格中是 long16 或 int16）	
CL_DEVICE_SINGLE_FP_CONFIG	cl_device_fp_config
描述设备的單精度浮點能力。此位欄支持下列值： ● CL_FP_DENORM——支持去規格化數（denorm）。 ● CL_FP_INF_NAN——支持 INF 和 qNaN。 ● CL_FP_ROUND_TO_NEAREST——支持捨入為最近偶數。 ● CL_FP_ROUND_TO_ZERO——支持向零捨入。 ● CL_FP_ROUND_TO_INF——支持向正無窮和負無窮捨入。 ● CL_FP_FMA——支持 IEEE754-2008 中的積和熔加运算（fused multiply-add, FMA）。 ● CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT——除法和開方可以按 IEEE754 規範進行正確的捨入。 ● CL_FP_SOFT_FLOAT——軟件中實現了基本的浮點運算（加、減、乘）。 如果设备類型不是 CL_DEVICE_TYPE_CUSTOM，其浮點能力至少要是：CL_FP_ROUND_TO_ZERO 或 CL_FP_ROUND_TO_NEAREST。	
CL_DEVICE_DOUBLE_FP_CONFIG	cl_device_fp_config
描述设备的雙精度浮點能力。此位欄支持下列值： ● CL_FP_DENORM——支持去規格化數。 ● CL_FP_INF_NAN——支持 INF 和 qNaN。 ● CL_FP_ROUND_TO_NEAREST——支持捨入為最近偶數。 ● CL_FP_ROUND_TO_ZERO——支持向零捨入。 ● CL_FP_ROUND_TO_INF——支持向正無窮和負無窮捨入。 ● CL_FP_FMA——支持 IEEE75-2008 中的積和熔加运算（fused multiply-add, FMA）。 ● CL_FP_SOFT_FLOAT——軟件中實現了基本的浮點運算（加、減、乘）。 由於雙精度浮點是一個可選特性，所以最小的雙精度浮點能力可以是0。 而如果设备支持雙精度浮點，則其能力至少要是： 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。	
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	cl_device_mem_cache_type
所支持的全局內存緩存的類型。其值可以是： ● CL_NONE， ● CL_READ_ONLY_CACHE 和 ● CL_READ_WRITE_CACHE。	

CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE	cl_uint
全局內存緩存列（cache line）的字節數。	
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE	cl_ulong
全局內存緩存的字節數。	
CL_DEVICE_GLOBAL_MEM_SIZE	cl_ulong
全局设备內存的字節數。	
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	cl_ulong
一次所能分配的常量緩衝區的最大字節數。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，最小值是 1KB。	
CL_DEVICE_MAX_CONSTANT_ARGS	cl_uint
單個內核中，最多能有多少個參數在聲明時帶有限定符__*constant*。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，最小值是 4。	
CL_DEVICE_LOCAL_MEM_TYPE	cl_device_local_mem_type
所支持的局部內存的類型。可以是 CL_LOCAL（意指專用的局部內存，如 SRAM ）或 CL_GLOBAL。 對於自定義設備，如果不支持局部內存，可以返回 CL_NONE。	
CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong
局部內存區的字節數。對於類型不是 CL_DEVICE_TYPE_CUSTOM 的设备，最小值是 1KB。	
CL_DEVICE_ERROR_CORRECTION_SUPPORT	cl_bool
如果所有對計算設備內存（包括全局內存和不變內存）的訪問，都可以由设备進行糾錯，則為 CL_TRUE，否則為 CL_FALSE。	
CL_DEVICE_HOST_UNIFIED_MEMORY	cl_bool
如果设备和主機共有一個統一的內存子系統，則為 CL_TRUE，否則為 CL_FALSE。	
CL_DEVICE_PROFILING_TIMER_RESOLUTION	size_t
设备定時器的精度。單位是納秒。詳情參見節 *5.12*。	
CL_DEVICE_ENDIAN_LITTLE	cl_bool
如果 OpenCL 设备是小端（little-endian）的，則為 CL_TRUE，否則為CL_FALSE。	
CL_DEVICE_AVAILABLE	cl_bool
如果设备可用，則為 CL_TRUE，否則為CL_FALSE。	
CL_DEVICE_COMPILER_AVAILABLE	cl_bool
如果沒有可用的編譯器來編譯程式源碼，則為 CL_FALSE，否則為 CL_TRUE。 只有嵌入式平台的規格才可以是 CL_FALSE。	
CL_DEVICE_LINKER_AVAILABLE	cl_bool
如果沒有可用的鏈接器，則為 CL_FALSE，否則為 CL_TRUE。 只有嵌入式平台的規格才可以是 CL_FALSE。 如果 CL_DEVICE_COMPILER_AVAILABLE 是 CL_TRUE，則他必須是 CL_TRUE。	
CL_DEVICE_EXECUTION_CAPABILITIES	cl_device_exec_capabilities
描述设备的執行能力。此位欄包含以下值: • CL_EXEC_KERNEL——這個 OpenCL 设备可以執行 OpenCL 內核。 • CL_EXEC_NATIVE_KERNEL——這個 OpenCL 设备可以執行原生內核。 其中 CL_EXEC_KERNEL 是必需的。	
CL_DEVICE_QUEUE_PROPERTIES	cl_command_queue_properties

命令隊列的屬性。此位欄包含以下值: • CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE • CL_QUEUE_PROFILING_ENABLE 參見表 *5.1*。 其中 CL_QUEUE_PROFILING_ENABLE 是必需的。	
CL_DEVICE_BUILT_IN_KERNELS	char[]
设备所支持的內建內核的清單，以分號分隔。如果不支持內建內核，則返回空字串。	
CL_DEVICE_PRINTF_BUFFER_SIZE	size_t
內核調用 printf 時，由一個內部緩衝區存儲其輸出，此區域大小的最大值。對於嵌入式規格，最小為 1KB。	
CL_DEVICE_PREFERRED_INTEROP_USER_SYNC	cl_bool
OpenCL 和其他 API（如 DirectX）間共享內存對象時，如果设备的偏好是讓用戶自己負責同步，則其值為 CL_TRUE；而如果设备或實作已經具備有效的方式來進行同步，則其值為 CL_FALSE。	
CL_DEVICE_PARENT_DEVICE	cl_device_id
返回此子設備所屬父設備的 cl_device_id。如果 *device* 是根設備，則返回 NULL。	
CL_DEVICE_PARTITION_MAX_SUB_DEVICES	cl_uint
劃分设备時，所能創建的子設備的最大數目。 所返回的值不能超過 CL_DEVICE_MAX_COMPUTE_UNITS。	
CL_DEVICE_PARTITION_PROPERTIES	cl_device_partition_property[]
返回 *device* 所支持的劃分方式。這是一個陣列，元素型別為 cl_device_partition_property，其值可以是: • CL_DEVICE_PARTITION_EQUALLY • CL_DEVICE_PARTITION_BY_COUNTS • CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN 如果此设备不支持任何劃分方式，則返回 0。	
CL_DEVICE_PARTITION_AFFINITY_DOMAIN	cl_device_affinity_domain
用 CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN 劃分 *device* 時，所支持的相似域（affinity domain）。此位欄的值如下所示: • CL_DEVICE_AFFINITY_DOMAIN_NUMA • CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE • CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE • CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE • CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE • CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE 如果以上都不支持，就返回 0。	
CL_DEVICE_PARTITION_TYPE	cl_device_partition_property[]
如果 *device* 是子設備，則會返回調用 **clCreateSubDevices** 時所指定的引數 *properties*。否則返回的 *param_value_size_ret* 可能是 0 即不存在任何劃分方式；或者 *param_value* 所指內存中的屬性值是 0（0 用來終止屬性清單）。	
CL_DEVICE_REFERENCE_COUNT	cl_uint
返回 *device* 的引用計數。如果是根設備，則返回 1。	

如果表 *4.3*中的 CL_DEVICE_IMAGE_SUPPORT 是 CL_TRUE，則實作指派給下列項的值必須大於或等於上表中給出的最小值:

- 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、
- CL_DEVICE_MAX_SAMPLERS。

另外，OpenCL 嵌入式規格的實作必須支持下列圖像格式。

對於 2D 和可選的 3D 圖像，至少要支持下列圖像格式（讀寫都要支持）:

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

第 11 章
参考文獻

[1] .*The AltiVecTM Technology Programming Interface Manual*..

[2] .*The ANSI/IEEE Std 754-1985 and 754-2008 Specifications*..

[3] .*ATI CTM Guide – Technical Reference Manual*.. *http://ati.amd.com/companyinfo/researcher /documents/ATI_CTM_Guide.pdf*.

[4] .*Explicit Memory Fences*.. *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2262.html*.

[5] .*GCC Attribute Syntax*.. *http://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html*.

[6] .*The ISO/IEC 9899:1999 "C" Language Specification*..

[7] .*The ISO/IEC JTC1 SC22 WG14 N1169 Specification*..

[8] .*NESL – A nested data parallel language*.. *http://www.cs.cmu.edu/~scandal/nesl.html*.

[9] .*NVIDIA CUDA Programming Guide*.. *http://developer.nvidia.com/object/cuda.html*.

[10] .*The OpenGL Specification and the OpenGL Shading Language Specification*.. *http://www.opengl .org/registry/*.

[11] .*OpenMP Application Program Interface*.. *http://www.openmp.org/drupal/mp-documents/spec25 .pdf*.

[12] Ian, Buck.*Brook Specification v0.2*.. *http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf*.

[13] Daniel Horn Jeremy Sugarman Kayvon Fatahalian Mike Houston Pat Hanrahan, Ian Buck, Tim Foley.*Brook for GPUs: Stream Computing on Graphics Hardware*..

[14] Michael Wong, Jens Maurer (2008).Towards support for attributes in c++. . Proposed to WG21 "Programming Language C++, Core Working Group".

[15] Strzodka Sengupta Owens, Lefohn, Kniss (2006). Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, pp. 60-99.

[16] Jean-Michel, Muller.*On the definition of ulp (x)*.. *ftp://ftp.inria.fr/INRIA/publication/publi-pdf /RR/RR-5504.pdf*.

[17] Kolb Lalonde Foley Berry, Pharr, Lefohn (2007).*Programmable Graphics—The Future of Interactive Rendering*. Neoptica Whitepaper.

附錄 A
共享 OpenCL 對象

本節將描述有哪些對象可以由（在同一個主機進程中創建的）多個命令隊列共享。

OpenCL 內存對象、程式對象或內核對象都可以由多個命令隊列共享，只要命令隊列和這些對象是由同一個上下文創建的即可。命令入隊時會創建事件對象。這些事件對象也可以由使用同一上下文創建的多個命令隊列共享。

應用需要在主機處理器上的多個線程間進行同步，以確保當多個線程中的多個命令隊列都要改變所共享對象（像命令隊列對象、內存對象、程式對象、內核對象）的狀態時，這些改變按正確的次序進行（應用認為正確的次序）。

命令隊列可以將其中的命令對內存對象狀態的改變緩存起來。要想在多個命令隊列間同步這些變動，應用必須做到：

如果命令隊列中的命令會修改內存對象的狀態，那麼應用必須：

- 得到這些命令所對應的事件對象。
- 調用 **`clFlush`**（或 **`clFinish`**）以使此隊列中所有未完成的命令開始執行。

如果命令隊列想同步內存對象的最新狀態，應用所入隊的命令必須等在一些事件上，這些事件代表的是那些會修改共享對象狀態的命令。這樣可以確保在此隊列中的命令使用共享對象前，前一個隊列中使用此對象的命令都已執行完畢。

如果某個命令隊列要修改一個共享的資源，而另一個隊列正在使用此資源，那麼其行為未定義。

附錄 B
多個主機線程

請參考第2章中對線程安全的定義。在不同語境中這個名詞的定義可能不一樣。

除了 **clSetKernelArg**，所有的 OpenCL API 調用都是線程安全的。只要操作的是不同的 `cl_kernel` 對象，那麼在任何主機線程中調用 **clSetKernelArg** 都是安全的，而且是重入安全的。然而，如果多個主機線程同時調用 **clSetKernelArg** 以操作同一個 `cl_kernel` 對象，則其行為未定義。需要注意的是，如果是在 OpenCL 回調函式中調用 OpenCL API，還有其他限制，請參考節 *5.9*。

> 在 *OpenCL* 的設計中還存在一個天生的竞態：設置內核引數和使用內核（用 **clEnqueueNDRangeKernel** 或 **clEnqueueTask**）。在某個主機線程設置內核引數並將內核入隊時，在這兩個動作中間可能另一個線程改變了內核引數，從而導致所入隊的內核引數是錯誤的。相對於試圖在多個主機線程間共享 `cl_kernel` 對象，強烈建議應用為每個主機線程都創建一個獨立的 `cl_kernel` 對象。

如果是在中斷或信號處理函式中調用 OpenCL API，則其行為未定義。

OpenCL 實作應當能夠為一個給定的 OpenCL 上下文創建多個命令隊列，同時也可以在一個應用中創建多個 OpenCL 上下文。

附錄 C
可移植性

OpenCL 的設計目標之一就是要能移植到其他架構和硬件上。OpenCL 的核心是基於 C99 的編程語言。浮點算術基於 **IEEE-754** 和 **IEEE-754-2008** 標準。設計內存對象、指針限定符以及弱序內存（weakly ordered memory）的目的就是要為 OpenCL 设备所實現的分離式內存架構提供最大的兼容性。命令隊列和屏障允許在主機和 OpenCL 设备間進行同步。OpenCL 的設計、能力以及限制很好的反映了下層硬件的能力。

不幸的是，有很多事情某種硬件能夠做到，而對於其他硬件來說則無能為力。拜常駐 CPU 的操作系統所賜，在一些實作中，內核在 CPU 上執行時可以調出系統服務，而在 GPU 上同樣的調用則可能會失敗，至少目前如此。（參見第 9章）。出於調試的目的，這些服務還是非常有用的，實作可以用 OpenCL 的擴展機制來實現這些服務。

同樣，計算架構的不同可能意味着，一個迴圈結構在 CPU 上的執行速度還可以接受，但在 GPU 上執行時性能可能會很差。CPU 一般都是為單線程任務、時延敏感的算法設計的，因此這種任務執行的很好，而 GPU 則可能遭遇相當長的時延，甚至是指數級的惡化。志於編寫可移植代碼的開發人員可能會發現，要想確保關鍵算法在多種設備上都工作的很好，在每種設備上都要進行測試。我們建議增加作業項的數目。希望在未來的歲月中逐漸積累經驗，能夠產生一些統一的最佳實踐，讓我們能夠在多種計算設備上都能夠使用。

我們可能更關心的是端序。鑒於最開始 OpenCL 的一些實作都是針對小端設備的，開發人員需要確保其內核在大端、小端設備上都做過測試，從而保證源碼對現在以及將來的 OpenCL 設備的兼容性。OpenCL 編程語言還支持端序特性限定符，允許開發人員指定數據使用主機的端序還是 OpenCL 设备的端序。這樣使得 OpenCL 編譯器在裝載、存儲數據時可以進行適當的端序轉換。

下面我們來描述端序是怎樣導致意料之外的結果的：

當大端矢量機器（如 AltiVec，CELL SPE）裝載矢量時，數據的次序保持不變，即每個矢量元素內部的字節序以及元素間的次序都與內存中的一樣。當小端矢量機器（如 SSE）裝載矢量時，會將寄存器中的數據反序（所有工作都是在寄存器中完成的），每個矢量元素內部的字節序以及元素間的次序**都**會反序。

內存：

uint4 a =	**0x00010203**	0x04050607	0x08090A0B	0x0C0D0E0F

寄存器（大端）：

uint4 a =	**0x00010203**	0x04050607	0x08090A0B	0x0C0D0E0F

寄存器（小端）：

uint4 a =	0x0F0E0D0C	0x0B0A0908	0x07060504	**0x03020100**

無論矢量中每塊數據有多大，小端機器一次就能將其全部裝載。也就是說，矢量為 `uchar16` 還是 `ulong2`，所進行的變換都一樣。當然，眾所周知，對於陣列元素，小端機器實際上按反向字節序存儲數據，以作為小端存儲格式的補償：

注意，此處我們談論的是編程模型。事實上，小端系統可能選擇其他方式，如只是簡單地從“右側”尋址或將字節中的位反序。這兩種方式都意味着硬件上無需大的交換空間。

內存（大端）：

uint4 a =	**0x00010203**	0x04050607	0x08090A0B	0x0C0D0E0F

內存（小端）：

uint4 a =	**0x03020100**	0x07060504	0x0B0A0908	0x0F0E0D0C

一旦數據裝載到矢量中，以此結束：

寄存器（大端）：

uint4 a =	**0x00010203**	0x04050607	0x08090A0B	0x0C0D0E0F

寄存器（小端）：

uint4 a =	0x0C0D0E0F	0x08090A0B	0x04050607	**0x00010203**

在糾正每個元素內的字節序過程中，同時也會反轉元素間的次序。`0x00010203` 位於大端矢量的左側，但位於小端矢量的右側。

如果主機和设备具有不同的端序，則開發人員需要保證內核引數的值得到了正確處理。實作不一定會自動轉換內核引數的端序。至於這些情況下要如何處置內核引數，開發人員需要查閱供應商的文檔。

通過按元素在內存中的次序進行編號，OpenCL 為不同架構提供了一致的編程模型。`even/odd` 和 `high/low` 的概念也是這樣的。一旦將數據裝載到寄存器中，我們會發現元素 0 是大端矢量的最左側元素，卻是小端矢量的最右側元素。

```
float x[4];
float4 v = vload4( 0, x );
```

大端:

```
v contains { x[0], x[1], x[2], x[3] }
```

小端:

```
v contains { x[3], x[2], x[1], x[0] }
```

編譯器知道所發生的交換並據此引用元素。只要我們是通過數值索引(像.s0123456789abcdef)或描述符(像.xyzw、.hi、.lo、.even和.odd)來訪問這些元素,那麼這種交換就是透明的。當將數據存儲回內存中時,所有次序反轉都會被撤銷。對於大多數問題而言,開發人員都應當按大端編程模型工作,並忽略矢量元素的次序問題。此機制依賴於以下事實:我們可以信賴始終如一的元素編號。一旦我們改變的編號系統,例如通過無需轉換的轉型(使用as_type*n*)將一個矢量轉型成另外一個大小相同但元素個數不同的矢量,則在不同的實作中可能得到不同的結果,這取決於系統是大端的、小端的,還是根本沒有矢量單元。(因此,對於元素數目不同的矢量間的直轉(bitcast),其行為依賴於具體實作,參見節 *6.2.4*。)

看下面這個例子:

```
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

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

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
```

其中z.z中的值在大端機器和小端機器上是不同的。

OpenCL以移植性的名義規定,如果無需轉換的轉型改變了元素的數目,則違規。然而,雖然OpenCL提供了一組公共的矢量算子(取自矢量機器),還是不能以一種始終如一、統一且可移植的方式來訪問每個ISA都會提供的所有東西。許多矢量ISA提供了一些特定目的的指令,可以極大地加速一些特定運算,像DCT、SAD、或3D幾何。我們不打算讓OpenCL如此笨重以致難以上手,如果那樣的話,在編寫對事件有嚴格要求、性能敏感的算法時,即使是經驗豐富的開發人員也難以使其接近頂峰性能。開發人員更傾向於拋開移植性,在代碼中使用特定平台的指令。有鑒於此,OpenCL允許傳統的矢量C語言編程擴展作為OpenCL的擴展直接使用OpenCL數據型別,像AltiVec C編程接口,或Intel C編程接口(在emmintrin.h中可以找到)。由於這些接口要想正常運作,依賴於無需轉換、但會改變矢量元素數目的轉型,所以OpenCL也允許這種轉型。

作為通用規則,任何矢量型別的運算,只要元素數目與實際矢量型別的元素數目不同,那麼他在其他不同端序或不同矢量架構的硬件上運行時都可能終止。

這樣的例子有:

- 使用算子.even和.odd將兩個uchar8分別作為ushort的高字節和低字節組合成一個ushort8(請使用**upsample**,參見節 *6.12.3*)。
- 任何會改變矢量元素數目的直轉。(新型別上的運算是不可移植的。)
- 所用塊大小與元素大小不同,且會改變數據次序的重排運算。

下列運算則則是可移植的:

- 使用算子 `.even` 和 `.odd` 將兩個 `uint8` 組合成一個 `uchar16`。例如將左右兩個聲道的音頻流間插到一起。
- 任何不會改變矢量元素數目的直轉 (如 `(float4)uint4`，為浮點型別定義存儲格式)。
- 單位為矢量元素的重排運算。

OpenCL 為 C 增加了一些東西，時應用的行為更可靠。最主要的就是 OpenCL 中定義了下列運算的行為，而這些在 C99 中並沒有定義:

- OpenCL 為所有型別間的轉換都提供了 `convert_` 算子。在將浮點型別轉換成整數型別時，在捨入後浮點值可能仍然超出了整數可表示的範圍，這時會發生什麼在 C99 中沒有定義。如果使用了 `_sat` 轉換，則浮點值會被轉換成最近的可表示的整數。類似地，對於 NaN 會發生什麼，OpenCL 也給出了建議。對於提供有硬件實現的飽和轉換的硬件製造商，OpenCL `convert_` 算子的飽和版本和非飽和版本可能都會使用此硬件。而對於非飽和版本，如果浮點算元在捨入後仍然超出了整數可表示的範圍，這時會發生什麼 OpenCL 則沒有定義。
- 在 IEEE-754 標準草案中，型別 `half`、`float` 和 `double` 的格式定義為 binary16、binary32 和 binary64。(後兩個在現有的 IEEE-754 標準中是相同的。) 你可能要依賴於此定義中各個位的位置和意義。
- OpenCL 中定義了移位越界時的行為。如果移位運算的移位數大於或等於第一個算子的位數，則會對齊取模。例如，如果要將 `int4` 左移 `33` 位，則 OpenCL 將按左移 `33%32 = 1` 對待。
- 對於數學庫函式中的大量邊界條件，比 C99 定義的更加嚴格。參見節 *7.5*。

附錄 D
應用數據型別

本節中會列出為主機應用提供的數據結構、型別和常量定義。這些定義是所有平台和架構所共有的。這些額外細節可以證實我們要維護一個可移植編程環境的承諾，潛在地也會阻止對所供應頭檔的修改。

节 4.1 共享的應用標量數據型別

為了讓應用用起來更加方便，提供了下列應用標量型別：

```
cl_char
cl_uchar
cl_short
cl_ushort
cl_int
cl_uint
cl_long
cl_ulong
cl_half
cl_float
cl_double
```

节 4.2 所支持的應用矢量數據型別

應用矢量型別是聯合體，用來創建上一節中標量型別的矢量。為了讓應用用起來更加方便，提供了下列應用矢量型別：

```
cl_charn
cl_ucharn
cl_shortn
cl_ushortn
cl_intn
cl_uintn
cl_longn
cl_ulongn
cl_halfn
cl_floatn
cl_doublen
```

其中 *n* 可以是 2、3、4、8 或 16。

應用標量和矢量數據型別的定義在頭檔 **cl_platform.h** 中。

节 4.3 應用數據型別的齊位

用戶要負責確保傳入和傳出 OpenCL 緩衝的數據相對於緩衝的起始位址原生對齊，遵守節 *6.1.5*中的要求。其中隱含着，用 `CL_MEM_USE_HOST_PTR` 創建 OpenCL 緩衝時所提供的主機內存指針必須按內核中訪問這些緩衝時所用的數據型別進行對齊。此外，用戶也要負責傳入和傳出 OpenCL 圖像的數據按用來表示單個像素的數據粒度進行對齊（如：`image_num_channels * sizeof(image_channel_data_type)`，不過對於 `CL_RGB` 和 `CL_RGBx` 圖像，只需按像素的單個通道的粒度進行對齊即可（即 `sizeof(image_channel_data_type)`）。

對於那些由應用所定義的、既不是緩衝也不是圖像的數據型別，OpenCL 不對其齊位作任何要求，但是如果包含矢量（如 `__cl_float4`），必須要能用 `cl_type` 聯合體中欄位的名字直接訪問才行（節 *4.5*）。儘管如此，還是建議頭檔 **cl_platform.h** 將 OpenCL 所定義的應用數據型別（如 `cl_float4`）按其型別自然地對齊。

节 4.4 常值矢量

應用常值矢量可用來對矢量進行組件級賦值。使用常值矢量時編譯器可能會對齊進行轉換。

```
cl_float2 foo = {.s[1] = 2.0f };
```

```
2   cl_int8 bar = {{ 2, 4, 6, 8, 10, 12, 14, 16 }};
```

节 4.5 矢量組件

可以使用符號 <vector_name>.s[<index>] 對應用矢量型別的組件進行尋址。

例如:

```
1   foo.s[0] = 1.0f; // Sets the 1st vector component of foo
2   pos.s[6] = 2; // Sets the 7th vector component of bar
```

一些情況下，也可以使用下列符號訪問矢量組件。不保證所有實作都會支持這些符號，所以使用他們時要檢查一下相應的預處理器記號。

4.5.1 命名矢量組件符號

可以使用 .sN、.sn 或 .xyzw 來訪問矢量數據型別的組件，跟 OpenCL 語言中所用類似。.xyzw 僅可訪問前四個組件。可以通過預處理器記號 CL_HAS_NAMED_VECTOR_FIELDS 來識別是否支持這些符號。例如:

```
1   #ifdef CL_HAS_NAMED_VECTOR_FIELDS
2           cl_float4 foo;
3           cl_int16 bar;
4           foo.x = 1.0f;   // Set first component
5           foo.s0 = 1.0f;  // Same as above
6           bar.z = 3;      // Set third component
7           bar.se = 11;    // Same as bar.s[0xe]
8           bar.sD = 12;    // Same as bar.s[0xd]
9   #endif
```

與 OpenCL 語言不同的是，一次只能訪問一個組件。這個局限使得不能像 OpenCL 語言型別那樣對組件進行重排或複製。如果訪問時組件編號越界會導致失敗。

```
1   foo.xy          // illegal - illegal field name combination
2   bar.s1234       // illegal - illegal field name combination
3   foo.s7          // illegal - no component s7
```

4.5.2 High/Low 矢量組件符號

可以使用 .hi 和 .lo 來訪問矢量數據型別的組件，跟所支持的語言型別類似。可以通過預處理器記號 CL_HAS_HI_LO_VECTOR_FIELDS 來識別是否支持這些符號。例如:

```
1   #ifdef CL_HAS_HI_LO_VECTOR_FIELDS
2           cl_float4 foo;
3           cl_float2 new_hi = 2.0f, new_lo = 4.0f;
4           foo.hi = new_hi;
5           foo.lo = new_lo;
6   #endif
```

4.5.3 原生矢量型別符號

為了將矢量型別映射到內建矢量型別上，還定義了原生矢量型別。不同於上述的應用矢量型別，這些原生型別僅僅是在有限的基礎上才支持，取決於所支持的架構和編譯器。

這些型別不是聯合體，但可以更方便地映射到底層架構中的內建矢量型別上。原生型別的名字與對應的應用型別一樣，只不過帶有前綴雙下劃線“__”。

例如，__cl_float4 就是一個原生矢量型別，對應於應用矢量型別 cl_float4。用 __cl_float4 可能可以直接訪問內建的 __m128 或矢量浮點型別，而 cl_float4 則會被視為聯合體。

另外，為方便訪問，上述應用數據型別可能含有原生矢量數據型別的成員。用子矢量符號 .vN 訪問原生組件，其中 N 為子矢量中的元素編號。如果原生型別是一個更大型別（具有更多組件）的子集，則此符號就成為子矢量型別的一個基於索引的陣列。

通過與原生型別名相匹配的預處理器記號 __CL_TYPEN__ 來識別是否支持原生矢量型別。例如:

```
#ifdef __CL_FLOAT4__    // Check for native cl_float4 type
        cl_float8 foo;
        __cl_float4 bar; // Use of native type
        bar = foo.v4[1]; // Access the second native float4 vector
#endif
```

节 4.6 隱式轉換

不支持應用矢量型別間的隱式轉換。

节 4.7 顯式轉型

不支持對應用矢量型別 cl_type*n* 的顯式轉型。對原生矢量型別 __cl_type*n* 的顯式轉型則由外部編譯器定義。

节 4.8 其他算子和函式

對於應用矢量型別 cl_type*n* 和原生矢量型別 __cl_type*n* 而言，標準算子和函式的行為由外部編譯器定義。

节 4.9 應用常量的定義

除了以上應用型別的定義，還有下列常值定義。

常量	描述
CL_CHAR_BIT	字符位寬
CL_SCHAR_MAX	型別 cl_char 的最大值
CL_SCHAR_MIN	型別 cl_char 的最小值
CL_CHAR_MAX	型別 cl_char 的最大值
CL_CHAR_MIN	型別 cl_char 的最小值
CL_UCHAR_MAX	型別 cl_uchar 的最大值
CL_SHORT_MAX	型別 cl_short 的最大值
CL_SHORT_MIN	型別 cl_short 的最小值
CL_USHORT_MAX	型別 cl_ushort 的最大值
CL_INT_MAX	型別 cl_int 的最大值
CL_INT_MIN	型別 cl_int 的最小值
CL_UINT_MAX	型別 cl_uint 的最大值
CL_LONG_MAX	型別 cl_long 的最大值
CL_LONG_MIN	型別 cl_long 的最小值
CL_ULONG_MAX	型別 cl_ulong 的最大值
CL_FLT_DIAG	型別 cl_float 的精度，十進制小數的位數
CL_FLT_MANT_DIG	型別 cl_float 的尾數的數字位數
CL_FLT_MAX_10_EXP	型別 cl_float 所能表示的規格化浮點數中能以 $10^n - 1$ 表示的數中 n 的最大值，其中 n 為正整數。
CL_FLT_MAX_EXP	型別 cl_float 的最大指數值
CL_FLT_MIN_10_EXP	型別 cl_float 所能表示的規格化浮點數中能以 $10^n - 1$ 表示的數中 n 的最小值，其中 n 為負整數。
CL_FLT_MIN_EXP	型別 cl_float 的最小指數值
CL_FLT_RADIX	型別 cl_float 的底數
CL_FLT_MAX	型別 cl_float 的最大值
CL_FLT_MIN	型別 cl_float 的最小值

CL_FLT_EPSILON	型別為 cl_float 且能使 1.0 + CL_FLT_EPSILON != 1 為真的最小正浮點數
CL_DBL_DIAG	型別 cl_double 的精度, 十進制小數的位數
CL_DBL_MANT_DIG	型別 cl_double 的尾數的數字位數
CL_DBL_MAX_10_EXP	型別 cl_double 所能表示的規格化浮點數中能以 $10^n - 1$ 表示的數中 n 的最大值, 其中 n 為正整數。
CL_DBL_MAX_EXP	型別 cl_double 的最大指數值
CL_DBL_MIN_10_EXP	型別 cl_double 所能表示的規格化浮點數中能以 $10^n - 1$ 表示的數中 n 的最小值, 其中 n 為負整數。
CL_DBL_MIN_EXP	型別 cl_double 的最小指數值
CL_DBL_RADIX	型別 cl_double 的底數
CL_DBL_MAX	型別 cl_double 的最大值
CL_DBL_MIN	型別 cl_double 的最小值
CL_DBL_EPSILON	型別為 cl_double 且能使 1.0 + CL_DBL_EPSILON != 1 為真的最小正浮點數
CL_NAN	此巨集展開後的值表示 NaN
CL_HUGE_VALF	型別 cl_float 可表示的最大值
CL_HUGE_VAL	型別 cl_double 可表示的最大值
CL_MAXFLOAT	型別 cl_float 的最大值
CL_INFINITY	此巨集展開後的值表示 INF

這些常值定義位於頭檔 **cl_platform.h** 中。

附錄 E
OpenCL C++ 外覆 API

OpenCL C++ 外覆 API 為平台和運行時 API 提供了 C++ 接口。他構建於 OpenCL 1.2 C API (包括平台和運行時) 之上，而不是要取代 C API。**要求**所有 C++ 外覆 API 的實作都要調用底層的 C API，並假定 C API 是符合 OpenCL 1.2 規範的。

細節請參閱《OpenCL C++ 外覆 API 規範》。可由如下網址得到：*http://www.khronos.org/registry/cl/*。

附錄 F
CL_MEM_COPY_OVERLAP

下列代碼描述的是如何確定為 **clEnqueueCopyBufferRect** 指定的源矩形和宿矩形指向了同一個緩衝對象，並且相互重疊。


```
1  bool
2  check_copy_overlap(size_t src_offset[3],
3                  size_t dst_offset[3],
4                  size_t region[3],
5                  size_t row_pitch, size_t slice_pitch)
6  {
7          const size_t src_min[] = {src_offset[0], src_offset[1], src_offset[2]};
8          const size_t src_max[] = {src_offset[0] + region[0],
9                                    src_offset[1] + region[1],
10                                   src_offset[2] + region[2]};
11
12         const size_t dst_min[] = {dst_offset[0], dst_offset[1], dst_offset[2]};
13         const size_t dst_max[] = {dst_offset[0] + region[0],
14                                   dst_offset[1] + region[1],
15                                   dst_offset[2] + region[2]};
16
17         // Check for overlap
18         bool overlap = true;
19         unsigned i;
20         for (i=0; i != 3; ++i)
21         {
22                 overlap = overlap && (src_min[i] < dst_max[i])
23                         && (src_max[i] > dst_min[i]);
24         }
25         size_t dst_start = dst_offset[2] * slice_pitch +
26                         dst_offset[1] * row_pitch + dst_offset[0];
27         size_t dst_end = dst_start + (region[2] * slice_pitch +
28                         region[1] * row_pitch + region[0]);
29         size_t src_start = src_offset[2] * slice_pitch +
30                         src_offset[1] * row_pitch + src_offset[0];
31         size_t src_end = src_start + (region[2] * slice_pitch +
32                         region[1] * row_pitch + region[0]);
33
34         if (!overlap)
35         {
36                 size_t delta_src_x = (src_offset[0] + region[0] > row_pitch) ?
```

```
                                        src_offset[0] + region[0] - row_pitch : 0;
                size_t delta_dst_x = (dst_offset[0] + region[0] > row_pitch) ?
                                        dst_offset[0] + region[0] - row_pitch : 0;

                if ( (delta_src_x > 0 && delta_src_x > dst_offset[0]) ||
                        (delta_dst_x > 0 && delta_dst_x > src_offset[0]) )
                {
                        if ( (src_start <= dst_start && dst_start < src_end) ||
                                (dst_start <= src_start && src_start < dst_end) )
                                overlap = true;
                }

                if (region[2] > 1)
                {
                        size_t src_height = slice_pitch / row_pitch;
                        size_t dst_height = slice_pitch / row_pitch;
                        size_t delta_src_y = (src_offset[1] + region[1] > src_height) ?
                                                src_offset[1] + region[1] - src_height : 0;
                        size_t delta_dst_y = (dst_offset[1] + region[1] > dst_height) ?
                                                dst_offset[1] + region[1] - dst_height : 0;
                        if ( (delta_src_y > 0 && delta_src_y > dst_offset[1]) ||
                                (delta_dst_y > 0 && delta_dst_y > src_offset[1]) )
                        {
                                if ( (src_start <= dst_start && dst_start < src_end) ||
                                        (dst_start <= src_start && src_start < dst_end) )
                                        overlap = true;
                        }
                }
        }
        return overlap;
}
```

附錄 G
變化

节 7.1 自 OpenCL 1.0 發生的變化

OpenCL 1.1 的平台層和運行時（第 *4* 章和第 *5* 章）加入了如下特性:

- *表 4.III* 中加入了如下查詢:
  - CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR
  - CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT
  - CL_DEVICE_NATIVE_VECTOR_WIDTH_INT
  - CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG
  - CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT
  - CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE
  - CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF
  - CL_DEVICE_HOST_UNIFIED_MEMORY
  - CL_DEVICE_OPENCL_C_VERSION
- **clGetContextInfo** 中加入了查詢 CL_CONTEXT_NUM_DEVICES。
- 可選圖像格式: CL_Rx、CL_RGx 和 CL_RGBx。
- 對子緩衝對象的支持，可以用 **clCreateSubBuffer** 創建一個緩衝對象來指代另一緩衝對象的某個特定區域。
- 下列函式分別可以讀、寫、拷貝緩衝對象的某個矩形區域:
  - **clEnqueueReadBufferRect**、
  - **clEnqueueWriteBufferRect** 和
  - **clEnqueueCopyBufferRect**。
- **clSetMemObjectDestructorCallback** 允許用戶註冊一個回調函式，當內存對象被刪除、其資源被釋放時會調用此函式。
- 構建程式執行體時可以使用選項來控制所用的 OpenCL C 的版本。在節 *5.6.4.5* 中有所描述。
- **clGetKernelWorkGroupInfo** 中加入了查詢 CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE。
- 支持用戶事件。允許執行命令前等在用戶事件上。加入了新 API: **clCreateUserEvent** 和 **clSetUserEventStatus**。
- **clSetEventCallback** 可以為命令的特定執行狀態註冊回調函式。

OpenCL 1.1 的平台層和運行時（第 *4* 章和第 *5* 章）還進行了如下修正:

- *表 4.III* 中的下列查詢:
  - CL_DEVICE_MAX_PARAMETER_SIZE 由 256 變成了 1024 字節。
  - CL_DEVICE_LOCAL_MEM_SIZE 由 16KB 變成了 32KB。
- **clEnqueueNDRangeKernel** 的引數 *global_work_offset* 可以是非零值。
- 除了 **clSetKernelArg**，其他所有 API 都是線程安全的。

OpenCL 1.1 中的 OpenCL C 編程語言（第 *6* 章）加入了如下特性:

- 3 組件矢量數據型別
- 新的內建函式:
  - 作業項函式 **get_global_offset**，在節 *6.12.1*中定義。
  - 數學函式 **minmag** 和 **maxmag**，在節 *6.12.2*中定義。
  - 整數函式 **clamp**，在節 *6.12.3*中定義。
  - 整數函式 **min** 和 **max** 的矢量、標量變體，在節 *6.12.3*中定義。
  - **async_work_group_strided_copy**，在節 *6.12.10*中定義。
  - **vec_step**、**shuffle** 和 **shuffle2**，在節 *6.12.12*中定義。
- 擴展 **cl_khr_byte_addressable_store** 移入了核心規範。
- 下列擴展都移入了核心規範，內建原子函式的名字前綴由 **atom_** 變成了 **atomic_**。

  - **cl_khr_global_int32_base_atomics**、
  - **cl_khr_global_int32_extended_atomics**、
  - **cl_khr_local_int32_base_atomics** 和
  - **cl_khr_local_int32_extended_atomics** 。
- 巨集 `CL_VERSION_1_0` 和 `CL_VERSION_1_1`。

在 OpenCL 1.1 中，建議不再使用 OpenCL 1.0 的如下特性:

- 不再支持 API **clSetCommandQueueProperty**。
- 不再支持巨集 `__ROUNDING_MODE__`。
- **clBuildProgram** 的引數 *options* 不再支持選項 `-cl-strict-aliasing`。

OpenCL 1.1 的第 *9* 章中加入了如下擴展:

- **cl_khr_gl_event**，由 GL 同步對象創建 CL 事件對象。
- **cl_khr_d3d10_sharing**，與 Direct3D 10 共享內存對象。

OpenCL 1.1 的 OpenCL ES 規格（第 *10* 章）中做了如下修正:

- 對 64 位整數的支持是可選的。

节 7.2 自 OpenCL 1.1 發生的變化

OpenCL 1.2 的平台層和運行時（第 *4* 章和第 *5* 章）加入了如下特性:

- 支持自定義設備和內建內核。
- 可以依據设备所支持的劃分方案來劃分设备。
- 擴充了 *cl_mem_flags*，以描述主機如何訪問 `cl_mem` 對象中的數據。
- **clEnqueueFillBuffer** 和 **clEnqueueFillImage**，支持以某種範式填充緩衝對象或以某種顏色填充圖像對象。
- *cl_map_flags* 中加入了 `CL_MAP_WRITE_INVALIDATE_REGION`。規範中還對 `CL_MAP_WRITE` 的行為做了進一步澄清。
- 新的圖像型別：1D 圖像、由緩衝對象創建的 1D 圖像、1D 圖像陣列和 2D 圖像陣列。
- **clCreateImage**，可以創建圖像對象。
- API **clEnqueueMigrateMemObjects** 顯式地控制內存對象的位置，或者將其從一個设备遷移到另一個设备上。
- 編譯和鏈接的分離。
- **clGetProgramInfo** 中加入了一些查詢，用於在程式中可以查詢內核的數目以及內核的名字。
- **clGetProgramBuildInfo** 中加入了一些查詢，用於查詢編譯、鏈接的狀態和選項。
- API **clGetKernelArgInfo** 可以返回內核引數的資訊。
- **clEnqueueMarkerWithWaitList** 和 **clEnqueueBarrierWithWaitList**。

OpenCL 1.2 中的 OpenCL C 編程語言（第 *6* 章）加入了如下特性:

- 雙精度現在是一個可選的核心特性，而不再是擴展。
- 新的內建圖像型別：`image1d_t`、`image1d_array_t` 和 `image2d_array_t`。
- 新的內建函式:
  - 用於讀寫 1D 圖像、1D 和 2D 圖像陣列的函式，在節 *6.12.14.2*、節 *6.12.14.3*和 節 *6.12.14.4*中定義。
  - 無需採樣器的讀取圖像的函式，在節 *6.12.14.3*中定義。
  - 整數函式**popcount**，在節 *6.12.3*中定義。
  - 函式 **printf**，在節 *6.12.13*中定義。
- 存儲類別限定符 *extern* 和 *static*，在節 *6.8*中定義。
- 巨集 `CL_VERSION_1_2` 和 `__OPENCL_C_VERSION__`。

在 OpenCL 1.2 中，建議不再使用 OpenCL 1.1 的如下 API:

- **clEnqueueMarker**、**clEnqueueBarrier** 和 **clEnqueueWaitForEvents**。
- **clCreateImage2D** 和 **clCreateImage3D**。

- **clUnloadCompiler** 和 **clGetExtensionFunctionAddress**。
- **clCreateFromGLTexture2D** 和 **clCreateFromGLTexture3D**。

在 OpenCL 1.2 中, 建議不再使用如下查詢:

- *表 4.III*中 的 CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE （ 由 **clGetDeviceInfo** 使用)。

附錄 H
API 索引