The OpenGL graphics system is a software interface to graphics hardware. (“GL” stands for “Graphics Library.”) It allows you to create interactive programs that produce color images of moving, three-dimensional objects. With OpenGL, you can control computer-graphics technology to produce realistic pictures, or ones that depart from reality in imaginative ways.

The **OpenGL Series** from Addison-Wesley Professional comprises tutorial and reference books that help programmers gain a practical understanding of OpenGL standards, along with the insight needed to unlock OpenGL’s full potential.

Visit [informit.com/opengl](http://informit.com/opengl) for a complete list of available products.
Contents

Figures ................................................................. xv
Tables ................................................................. xxvi
Listings ................................................................. xxv
Foreword ............................................................... xxix
Preface ................................................................. xxxiii
Acknowledgments ..................................................... xli
About the Authors ................................................... xliii

Part I  The OpenCL 1.1 Language and API .................... 1

1. An Introduction to OpenCL ................................. 3
   What Is OpenCL, or . . . Why You Need This Book ........ 3
   Our Many-Core Future: Heterogeneous Platforms .......... 4
   Software in a Many-Core World ............................. 7
   Conceptual Foundations of OpenCL ......................... 11
      Platform Model .............................................. 12
      Execution Model .......................................... 13
      Memory Model ............................................. 21
      Programming Models .................................... 24
   OpenCL and Graphics ........................................ 29
   The Contents of OpenCL ..................................... 30
      Platform API .............................................. 31
      Runtime API .............................................. 31
      Kernel Programming Language ....................... 32
      OpenCL Summary ........................................ 34
   The Embedded Profile ...................................... 35
   Learning OpenCL ............................................. 36
2. **HelloWorld: An OpenCL Example** ............................................ 39
   
   Building the Examples ................................................. 40
   Prerequisites ................................................................. 40
   Mac OS X and Code::Blocks ............................................. 41
   Microsoft Windows and Visual Studio ................................. 42
   Linux and Eclipse .......................................................... 44
   HelloWorld Example ...................................................... 45
   Choosing an OpenCL Platform and Creating a Context .......... 49
   Choosing a Device and Creating a Command-Queue ................. 50
   Creating and Building a Program Object ............................ 52
   Creating Kernel and Memory Objects ................................. 54
   Executing a Kernel ....................................................... 55
   Checking for Errors in OpenCL ......................................... 57

3. **Platforms, Contexts, and Devices** ............................. 63
   
   OpenCL Platforms ........................................................ 63
   OpenCL Devices ............................................................. 68
   OpenCL Contexts ............................................................ 83

4. **Programming with OpenCL C** .................................... 97
   
   Writing a Data-Parallel Kernel Using OpenCL C .................. 97
   Scalar Data Types ........................................................ 99
      The half Data Type .................................................... 101
   Vector Data Types ....................................................... 102
      Vector Literals ....................................................... 104
      Vector Components .................................................. 106
   Other Data Types ....................................................... 108
   Derived Types ............................................................ 109
   Implicit Type Conversions ............................................. 110
      Usual Arithmetic Conversions .................................... 114
   Explicit Casts ........................................................... 116
   Explicit Conversions .................................................... 117
   Reinterpreting Data as Another Type ............................... 121
   Vector Operators ........................................................ 123
      Arithmetic Operators .............................................. 124
      Relational and Equality Operators ............................... 127
5. OpenCL C Built-In Functions ........................................ 149
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. Programs and Kernels</td>
<td>217</td>
</tr>
<tr>
<td>Program and Kernel Object Overview</td>
<td>217</td>
</tr>
<tr>
<td>Program Objects</td>
<td>218</td>
</tr>
<tr>
<td>Creating and Building Programs</td>
<td>218</td>
</tr>
<tr>
<td>Program Build Options</td>
<td>222</td>
</tr>
<tr>
<td>Creating Programs from Binaries</td>
<td>227</td>
</tr>
<tr>
<td>Managing and Querying Programs</td>
<td>236</td>
</tr>
<tr>
<td>Kernel Objects</td>
<td>237</td>
</tr>
<tr>
<td>Creating Kernel Objects and Setting Kernel Arguments</td>
<td>237</td>
</tr>
<tr>
<td>Thread Safety</td>
<td>241</td>
</tr>
<tr>
<td>Managing and Querying Kernels</td>
<td>242</td>
</tr>
<tr>
<td>7. Buffers and Sub-Buffers</td>
<td>247</td>
</tr>
<tr>
<td>Memory Objects, Buffers, and Sub-Buffers Overview</td>
<td>247</td>
</tr>
<tr>
<td>Creating Buffers and Sub-Buffers</td>
<td>249</td>
</tr>
<tr>
<td>Querying Buffers and Sub-Buffers</td>
<td>257</td>
</tr>
<tr>
<td>Reading, Writing, and Copying Buffers and Sub-Buffers</td>
<td>259</td>
</tr>
<tr>
<td>Mapping Buffers and Sub-Buffers</td>
<td>276</td>
</tr>
<tr>
<td>8. Images and Samplers</td>
<td>281</td>
</tr>
<tr>
<td>Image and Sampler Object Overview</td>
<td>281</td>
</tr>
<tr>
<td>Creating Image Objects</td>
<td>283</td>
</tr>
<tr>
<td>Image Formats</td>
<td>287</td>
</tr>
<tr>
<td>Querying for Image Support</td>
<td>291</td>
</tr>
<tr>
<td>Creating Sampler Objects</td>
<td>292</td>
</tr>
<tr>
<td>OpenCL C Functions for Working with Images</td>
<td>295</td>
</tr>
<tr>
<td>Transferring Image Objects</td>
<td>299</td>
</tr>
<tr>
<td>9. Events</td>
<td>309</td>
</tr>
<tr>
<td>Commands, Queues, and Events Overview</td>
<td>309</td>
</tr>
<tr>
<td>Events and Command-Queues</td>
<td>311</td>
</tr>
<tr>
<td>Event Objects</td>
<td>317</td>
</tr>
</tbody>
</table>
13. OpenCL Embedded Profile ............................................. 383
   OpenCL Profile Overview ............................................. 383
   64-Bit Integers ...................................................... 385
   Images ................................................................. 386
   Built-In Atomic Functions ......................................... 387
   Mandated Minimum Single-Precision Floating-Point
   Capabilities ............................................................. 387
   Determining the Profile Supported by a Device in an
   OpenCL C Program .................................................... 390

Part II  OpenCL 1.1 Case Studies ................................. 391
14. Image Histogram ..................................................... 393
   Computing an Image Histogram ...................................... 393
   Parallelizing the Image Histogram ................................... 395
   Additional Optimizations to the Parallel Image Histogram ...... 400
   Computing Histograms with Half-Float or Float Values for
   Each Channel .......................................................... 403
15. Sobel Edge Detection Filter ........................................ 407
   What Is a Sobel Edge Detection Filter? ............................ 407
   Implementing the Sobel Filter as an OpenCL Kernel .......... 407
16. Parallelizing Dijkstra’s Single-Source Shortest-Path
    Graph Algorithm ..................................................... 411
   Graph Data Structures ................................................ 412
   Kernels ..................................................................... 414
   Leveraging Multiple Compute Devices ................................ 417
17. Cloth Simulation in the Bullet Physics SDK .................... 425
   An Introduction to Cloth Simulation ............................... 425
   Simulating the Soft Body .............................................. 429
   Executing the Simulation on the CPU ............................. 431
   Changes Necessary for Basic GPU Execution .................... 432
   Two-Layered Batching .................................................. 438
Optimizing for SIMD Computation and Local Memory .......... 441
Adding OpenGL Interoperation ........................................ 446

18. Simulating the Ocean with Fast Fourier Transform .......... 449
   An Overview of the Ocean Application .......................... 450
   Phillips Spectrum Generation ..................................... 453
   An OpenCL Discrete Fourier Transform .......................... 457
      Determining 2D Decomposition ................................ 457
      Using Local Memory ............................................ 459
      Determining the Sub-Transform Size ......................... 459
      Determining the Work-Group Size .............................. 460
      Obtaining the Twiddle Factors ................................. 461
      Determining How Much Local Memory Is Needed ............. 462
      Avoiding Local Memory Bank Conflicts ....................... 463
      Using Images .................................................. 463
      A Closer Look at the FFT Kernel .............................. 463
      A Closer Look at the Transpose Kernel ....................... 467

19. Optical Flow ............................................................ 469
   Optical Flow Problem Overview ................................. 469
   Sub-Pixel Accuracy with Hardware Linear Interpolation .... 480
   Application of the Texture Cache ................................ 480
   Using Local Memory ................................................ 481
   Early Exit and Hardware Scheduling ............................. 483
   Efficient Visualization with OpenGL Interop .................. 483
   Performance ......................................................... 484

20. Using OpenCL with PyOpenCL ....................................... 487
   Introducing PyOpenCL .............................................. 487
   Running the PyImageFilter2D Example ......................... 488
   PyImageFilter2D Code ............................................. 488
   Context and Command-Queue Creation .......................... 492
   Loading to an Image Object ...................................... 493
   Creating and Building a Program ............................... 494
   Setting Kernel Arguments and Executing a Kernel ............ 495
   Reading the Results .............................................. 496
## 21. Matrix Multiplication with OpenCL ............................................. 499

- The Basic Matrix Multiplication Algorithm ........................................ 499
- A Direct Translation into OpenCL .................................................... 501
- Increasing the Amount of Work per Kernel ........................................ 506
- Optimizing Memory Movement: Local Memory .................................... 509
- Performance Results and Optimizing the Original CPU Code .................. 511

## 22. Sparse Matrix-Vector Multiplication ............................................. 515

- Sparse Matrix-Vector Multiplication (SpMV) Algorithm ........................ 515
- Description of This Implementation .................................................. 518
- Tiled and Packetized Sparse Matrix Representation .............................. 519
- Header Structure .................................................................................. 522
- Tiled and Packetized Sparse Matrix Design Considerations .................... 523
- Optional Team Information .................................................................... 524
- Tested Hardware Devices and Results .................................................. 524
- Additional Areas of Optimization ......................................................... 538

## A. Summary of OpenCL 1.1 ............................................................... 541

- The OpenCL Platform Layer ............................................................... 541
  - Contexts .......................................................................................... 541
  - Querying Platform Information and Devices ...................................... 542
- The OpenCL Runtime ............................................................................ 543
  - Command-Queues ............................................................................ 543
- Buffer Objects ...................................................................................... 544
  - Create Buffer Objects ....................................................................... 544
  - Read, Write, and Copy Buffer Objects .............................................. 544
  - Map Buffer Objects .......................................................................... 545
  - Manage Buffer Objects ..................................................................... 545
  - Query Buffer Objects ....................................................................... 545
- Program Objects .................................................................................... 546
  - Create Program Objects ................................................................. 546
  - Build Program Executable ............................................................... 546
  - Build Options .................................................................................. 546
  - Query Program Objects .................................................................... 547
  - Unload the OpenCL Compiler ............................................................ 547
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel and Event Objects</td>
<td>547</td>
</tr>
<tr>
<td>Create Kernel Objects</td>
<td>547</td>
</tr>
<tr>
<td>Kernel Arguments and Object Queries</td>
<td>548</td>
</tr>
<tr>
<td>Execute Kernels</td>
<td>548</td>
</tr>
<tr>
<td>Event Objects</td>
<td>549</td>
</tr>
<tr>
<td>Out-of-Order Execution of Kernels and Memory Object Commands</td>
<td>549</td>
</tr>
<tr>
<td>Profiling Operations</td>
<td>549</td>
</tr>
<tr>
<td>Flush and Finish</td>
<td>550</td>
</tr>
<tr>
<td>Supported Data Types</td>
<td>550</td>
</tr>
<tr>
<td>Built-In Scalar Data Types</td>
<td>550</td>
</tr>
<tr>
<td>Built-In Vector Data Types</td>
<td>551</td>
</tr>
<tr>
<td>Other Built-In Data Types</td>
<td>551</td>
</tr>
<tr>
<td>Reserved Data Types</td>
<td>551</td>
</tr>
<tr>
<td>Vector Component Addressing</td>
<td>552</td>
</tr>
<tr>
<td>Vector Components</td>
<td>552</td>
</tr>
<tr>
<td>Vector Addressing Equivalencies</td>
<td>553</td>
</tr>
<tr>
<td>Conversions and Type Casting Examples</td>
<td>554</td>
</tr>
<tr>
<td>Operators</td>
<td>554</td>
</tr>
<tr>
<td>Address Space Qualifiers</td>
<td>554</td>
</tr>
<tr>
<td>Function Qualifiers</td>
<td>554</td>
</tr>
<tr>
<td>Preprocessor Directives and Macros</td>
<td>555</td>
</tr>
<tr>
<td>Specify Type Attributes</td>
<td>555</td>
</tr>
<tr>
<td>Math Constants</td>
<td>556</td>
</tr>
<tr>
<td>Work-Item Built-In Functions</td>
<td>557</td>
</tr>
<tr>
<td>Integer Built-In Functions</td>
<td>557</td>
</tr>
<tr>
<td>Common Built-In Functions</td>
<td>559</td>
</tr>
<tr>
<td>Math Built-In Functions</td>
<td>560</td>
</tr>
<tr>
<td>Geometric Built-In Functions</td>
<td>563</td>
</tr>
<tr>
<td>Relational Built-In Functions</td>
<td>564</td>
</tr>
<tr>
<td>Vector Data Load/Store Functions</td>
<td>567</td>
</tr>
<tr>
<td>Atomic Functions</td>
<td>568</td>
</tr>
<tr>
<td>Async Copies and Prefetch Functions</td>
<td>570</td>
</tr>
<tr>
<td>Synchronization, Explicit Memory Fence</td>
<td>570</td>
</tr>
<tr>
<td>Miscellaneous Vector Built-In Functions</td>
<td>571</td>
</tr>
<tr>
<td>Image Read and Write Built-In Functions</td>
<td>572</td>
</tr>
</tbody>
</table>
Image Objects .............................................................. 573
  Create Image Objects ............................................. 573
  Query List of Supported Image Formats ................. 574
  Copy between Image, Buffer Objects ...................... 574
  Map and Unmap Image Objects ............................... 574
  Read, Write, Copy Image Objects ......................... 575
  Query Image Objects ........................................... 575
Image Formats .......................................................... 576
Access Qualifiers ....................................................... 576
Sampler Objects ......................................................... 576
Sampler Declaration Fields ....................................... 577
OpenCL Device Architecture Diagram ....................... 577
OpenCL/OpenGL Sharing APIs ................................. 577
  CL Buffer Objects > GL Buffer Objects ................. 578
  CL Image Objects > GL Textures .......................... 578
  CL Image Objects > GL Renderbuffers .................... 578
  Query Information ............................................... 578
Share Objects .......................................................... 579
  CL Event Objects > GL Sync Objects ...................... 579
  CL Context > GL Context, Sharegroup ................. 579
OpenCL/Direct3D 10 Sharing APIs .......................... 579

Index ......................................................................... 581
Figures

**Figure 1.1** The rate at which instructions are retired is the same in these two cases, but the power is much less with two cores running at half the frequency of a single core. ................................................................. 5

**Figure 1.2** A plot of peak performance versus power at the thermal design point for three processors produced on a 65nm process technology. Note: This is not to say that one processor is better or worse than the others. The point is that the more specialized the core, the more power-efficient it is. ......................... 6

**Figure 1.3** Block diagram of a modern desktop PC with multiple CPUs (potentially different) and a GPU, demonstrating that systems today are frequently heterogeneous .......................... 7

**Figure 1.4** A simple example of data parallelism where a single task is applied concurrently to each element of a vector to produce a new vector. ......................... 9

**Figure 1.5** Task parallelism showing two ways of mapping six independent tasks onto three PEs. A computation is not done until every task is complete, so the goal should be a well-balanced load, that is, to have the time spent computing by each PE be the same. .......... 10

**Figure 1.6** The OpenCL platform model with one host and one or more OpenCL devices. Each OpenCL device has one or more compute units, each of which has one or more processing elements. ......................... 12
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.7</td>
<td>An example of how the global IDs, local IDs, and work-group indices are related for a two-dimensional NDRange. Other parameters of the index space are defined in the figure. The shaded block has a global ID of ((g_x, g_y) = (6, 5)) and a work-group plus local ID of ((w_x, w_y) = (1, 1)) and ((l_x, l_y) = (2, 1)).</td>
</tr>
<tr>
<td>1.8</td>
<td>A summary of the memory model in OpenCL and how the different memory regions interact with the platform model.</td>
</tr>
<tr>
<td>1.9</td>
<td>This block diagram summarizes the components of OpenCL and the actions that occur on the host during an OpenCL application.</td>
</tr>
<tr>
<td>2.1</td>
<td>CodeBlocks CL_Book project.</td>
</tr>
<tr>
<td>2.2</td>
<td>Using cmake-gui to generate Visual Studio projects.</td>
</tr>
<tr>
<td>2.3</td>
<td>Microsoft Visual Studio 2008 Project.</td>
</tr>
<tr>
<td>2.4</td>
<td>Eclipse CL_Book project.</td>
</tr>
<tr>
<td>3.1</td>
<td>Platform, devices, and contexts.</td>
</tr>
<tr>
<td>3.2</td>
<td>Convolution of an 8×8 signal with a 3×3 filter, resulting in a 6×6 signal.</td>
</tr>
<tr>
<td>4.1</td>
<td>Mapping <code>get_global_id</code> to a work-item.</td>
</tr>
<tr>
<td>4.2</td>
<td>Converting a <code>float4</code> to a <code>ushort4</code> with round-to-nearest rounding and saturation.</td>
</tr>
<tr>
<td>4.3</td>
<td>Adding two vectors.</td>
</tr>
<tr>
<td>4.4</td>
<td>Multiplying a vector and a scalar with widening.</td>
</tr>
<tr>
<td>4.5</td>
<td>Multiplying a vector and a scalar with conversion and widening.</td>
</tr>
<tr>
<td>5.1</td>
<td>Example of the work-item functions.</td>
</tr>
<tr>
<td>7.1</td>
<td>(a) 2D array represented as an OpenCL buffer; (b) 2D slice into the same buffer.</td>
</tr>
</tbody>
</table>
Figure 9.1 A failed attempt to use the `clEnqueueBarrier()` command to establish a barrier between two command-queues. This doesn't work because the barrier command in OpenCL applies only to the queue within which it is placed. 316

Figure 9.2 Creating a barrier between queues using `clEnqueueMarker()` to post the barrier in one queue with its exported event to connect to a `clEnqueueWaitForEvent()` function in the other queue. Because `clEnqueueWaitForEvents()` does not imply a barrier, it must be preceded by an explicit `clEnqueueBarrier()`. 317

Figure 10.1 A program demonstrating OpenCL/OpenGL interop. The positions of the vertices in the sine wave and the background texture color values are computed by kernels in OpenCL and displayed using Direct3D. 344

Figure 11.1 A program demonstrating OpenCL/D3D interop. The sine positions of the vertices in the sine wave and the texture color values are programmatically set by kernels in OpenCL and displayed using Direct3D. 368

Figure 12.1 C++ Wrapper API class hierarchy 370

Figure 15.1 OpenCL Sobel kernel: input image and output image after applying the Sobel filter. 409

Figure 16.1 Summary of data in Table 16.1: NV GTX 295 (1 GPU, 2 GPU) and Intel Core i7 performance. 419

Figure 16.2 Using one GPU versus two GPUs: NV GTX 295 (1 GPU, 2 GPU) and Intel Core i7 performance. 420

Figure 16.3 Summary of data in Table 16.2: NV GTX 295 (1 GPU, 2 GPU) and Intel Core i7 performance—10 edges per vertex. 421

Figure 16.4 Summary of data in Table 16.3: comparison of dual GPU, dual GPU + multicore CPU, multicore CPU, and CPU at vertex degree 1. 423
| Figure 17.1 | AMD's Samari demo, courtesy of Jason Yang | 426 |
| Figure 17.2 | Masses and connecting links, similar to a mass/spring model for soft bodies | 426 |
| Figure 17.3 | Creating a simulation structure from a cloth mesh | 427 |
| Figure 17.4 | Cloth link structure | 428 |
| Figure 17.5 | Cloth mesh with both structural links that stop stretching and bend links that resist folding of the material | 428 |
| Figure 17.6 | Solving the mesh of a rope. Note how the motion applied between (a) and (b) propagates during solver iterations (c) and (d) until, eventually, the entire rope has been affected | 429 |
| Figure 17.7 | The stages of Gauss-Seidel iteration on a set of soft-body links and vertices. In (a) we see the mesh at the start of the solver iteration. In (b) we apply the effects of the first link on its vertices. In (c) we apply those of another link, noting that we work from the positions computed in (b). | 432 |
| Figure 17.8 | The same mesh as in Figure 17.7 is shown in (a). In (b) the update shown in Figure 17.7(c) has occurred as well as a second update represented by the dark mass and dotted lines | 433 |
| Figure 17.9 | A mesh with structural links taken from the input triangle mesh and bend links created across triangle boundaries with one possible coloring into independent batches | 434 |
| Figure 17.10 | Dividing the mesh into larger chunks and applying a coloring to those. Note that fewer colors are needed than in the direct link coloring approach. This pattern can repeat infinitely with the same four colors | 439 |
| Figure 18.1 | A single frame from the Ocean demonstration | 450 |
Figure 19.1  A pair of test images of a car trunk being closed. The first (a) and fifth (b) images of the test sequence are shown. .......................... 470

Figure 19.2  Optical flow vectors recovered from the test images of a car trunk being closed. The fourth and fifth images in the sequence were used to generate this result. .......................... 471

Figure 19.3  Pyramidal Lucas-Kanade optical flow algorithm ............... 473

Figure 21.1  A matrix multiplication operation to compute a single element of the product matrix, $C$. This corresponds to summing into each element $C_{i,j}$ the dot product from the $i$th row of $A$ with the $j$th column of $B$. .................. 500

Figure 21.2  Matrix multiplication where each work-item computes an entire row of the $C$ matrix. This requires a change from a 2D NDRange of size 1000×1000 to a 1D NDRange of size 1000. We set the work-group size to 250, resulting in four work-groups (one for each compute unit in our GPU). .......... 506

Figure 21.3  Matrix multiplication where each work-item computes an entire row of the $C$ matrix. The same row of $A$ is used for elements in the row of $C$ so memory movement overhead can be dramatically reduced by copying a row of $A$ into private memory.... 508

Figure 21.4  Matrix multiplication where each work-item computes an entire row of the $C$ matrix. Memory traffic to global memory is minimized by copying a row of $A$ into each work-item's private memory and copying rows of $B$ into local memory for each work-group. .................. 510

Figure 22.1  Sparse matrix example. .......................... 516

Figure 22.2  A tile in a matrix and its relationship with input and output vectors. .......................... 520

Figure 22.3  Format of a single-precision 128-byte packet ............... 521
| Figure 22.4 | Format of a double-precision 192-byte packet | 522 |
| Figure 22.5 | Format of the header block of a tiled and packetized sparse matrix | 523 |
| Figure 22.6 | Single-precision SpMV performance across 22 matrices on seven platforms | 528 |
| Figure 22.7 | Double-precision SpMV performance across 22 matrices on five platforms | 528 |
Tables

Table 2.1  OpenCL Error Codes.................................58
Table 3.1  OpenCL Platform Queries..........................65
Table 3.2  OpenCL Devices...................................68
Table 3.3  OpenCL Device Queries............................71
Table 3.4  Properties Supported by clCreateContext........85
Table 3.5  Context Information Queries.......................87
Table 4.1  Built-In Scalar Data Types........................100
Table 4.2  Built-In Vector Data Types.........................103
Table 4.3  Application Data Types.............................103
Table 4.4  Accessing Vector Components.......................106
Table 4.5  Numeric Indices for Built-In Vector Data Types...107
Table 4.6  Other Built-In Data Types..........................108
Table 4.7  Rounding Modes for Conversions....................119
Table 4.8  Operators That Can Be Used with Vector Data Types.123
Table 4.9  Optional Extension Behavior Description..........144
Table 5.1  Built-In Work-Item Functions......................151
Table 5.2  Built-In Math Functions............................154
Table 5.3  Built-In half_ and native_ Math Functions...........160
<table>
<thead>
<tr>
<th>Table 5.4</th>
<th>Single- and Double-Precision Floating-Point Constants</th>
<th>162</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 5.5</td>
<td>( \text{ulp} ) Values for Basic Operations and Built-In Math Functions</td>
<td>164</td>
</tr>
<tr>
<td>Table 5.6</td>
<td>Built-In Integer Functions</td>
<td>169</td>
</tr>
<tr>
<td>Table 5.7</td>
<td>Built-In Common Functions</td>
<td>173</td>
</tr>
<tr>
<td>Table 5.8</td>
<td>Built-In Geometric Functions</td>
<td>176</td>
</tr>
<tr>
<td>Table 5.9</td>
<td>Built-In Relational Functions</td>
<td>178</td>
</tr>
<tr>
<td>Table 5.10</td>
<td>Additional Built-In Relational Functions</td>
<td>180</td>
</tr>
<tr>
<td>Table 5.11</td>
<td>Built-In Vector Data Load and Store Functions</td>
<td>181</td>
</tr>
<tr>
<td>Table 5.12</td>
<td>Built-In Synchronization Functions</td>
<td>190</td>
</tr>
<tr>
<td>Table 5.13</td>
<td>Built-In Async Copy and Prefetch Functions</td>
<td>192</td>
</tr>
<tr>
<td>Table 5.14</td>
<td>Built-In Atomic Functions</td>
<td>195</td>
</tr>
<tr>
<td>Table 5.15</td>
<td>Built-In Miscellaneous Vector Functions</td>
<td>200</td>
</tr>
<tr>
<td>Table 5.16</td>
<td>Built-In Image 2D Read Functions</td>
<td>202</td>
</tr>
<tr>
<td>Table 5.17</td>
<td>Built-In Image 3D Read Functions</td>
<td>204</td>
</tr>
<tr>
<td>Table 5.18</td>
<td>Image Channel Order and Values for Missing Components</td>
<td>206</td>
</tr>
<tr>
<td>Table 5.19</td>
<td>Sampler Addressing Mode</td>
<td>207</td>
</tr>
<tr>
<td>Table 5.20</td>
<td>Image Channel Order and Corresponding Bolor Color Value</td>
<td>209</td>
</tr>
<tr>
<td>Table 5.21</td>
<td>Built-In Image 2D Write Functions</td>
<td>211</td>
</tr>
<tr>
<td>Table 5.22</td>
<td>Built-In Image 3D Write Functions</td>
<td>212</td>
</tr>
<tr>
<td>Table 5.23</td>
<td>Built-In Image Query Functions</td>
<td>214</td>
</tr>
</tbody>
</table>
Table 6.1  Preprocessor Build Options ......................... 223
Table 6.2  Floating-Point Options (Math Intrinsics) ........ 224
Table 6.3  Optimization Options ............................... 225
Table 6.4  Miscellaneous Options .............................. 226
Table 7.1  Supported Values for cl_mem_flags ............... 249
Table 7.2  Supported Names and Values for clCreateSubBuffer . . 254
Table 7.3  OpenCL Buffer and Sub-Buffer Queries .......... 257
Table 7.4  Supported Values for cl_map_flags ............... 277
Table 8.1  Image Channel Order ............................... 287
Table 8.2  Image Channel Data Type .......................... 289
Table 8.3  Mandatory Supported Image Formats ............. 290
Table 9.1  Queries on Events Supported in clGetEventInfo() ... 319
Table 9.2  Profiling Information and Return Types .......... 329
Table 10.1 OpenGL Texture Format Mappings to OpenCL Image Formats ......................... 346
Table 10.2 Supported param_name Types and Information Returned ......................... 348
Table 11.1 Direct3D Texture Format Mappings to OpenCL Image Formats ......................... 360
Table 12.1 Preprocessor Error Macros and Their Defaults .......... 372
Table 13.1 Required Image Formats for Embedded Profile ......... 387
Table 13.2 Accuracy of Math Functions for Embedded Profile versus Full Profile ................. 388
Table 13.3 Device Properties: Minimum Maximum Values for Full Profile versus Embedded Profile .......... 389
| Table 16.1 | Comparison of Data at Vertex Degree 5 | 418 |
| Table 16.2 | Comparison of Data at Vertex Degree 10 | 420 |
| Table 16.3 | Comparison of Dual GPU, Dual GPU + Multicore CPU, Multicore CPU, and CPU at Vertex Degree 10 | 422 |
| Table 18.1 | Kernel Elapsed Times for Varying Work-Group Sizes | 458 |
| Table 18.2 | Load and Store Bank Calculations | 465 |
| Table 19.1 | GPU Optical Flow Performance | 485 |
| Table 21.1 | Matrix Multiplication (Order-1000 Matrices) Results Reported as MFLOPS and as Speedup Relative to the Unoptimized Sequential C Program (i.e., the Speedups Are “Unfair”) | 512 |
| Table 22.1 | Hardware Device Information | 525 |
| Table 22.2 | Sparse Matrix Description | 526 |
| Table 22.3 | Optimal Performance Histogram for Various Matrix Sizes | 529 |
### Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 2.1</td>
<td>HelloWorld OpenCL Kernel and Main Function</td>
<td>46</td>
</tr>
<tr>
<td>Listing 2.2</td>
<td>Choosing a Platform and Creating a Context</td>
<td>49</td>
</tr>
<tr>
<td>Listing 2.3</td>
<td>Choosing the First Available Device and Creating a Command-Queue</td>
<td>51</td>
</tr>
<tr>
<td>Listing 2.4</td>
<td>Loading a Kernel Source File from Disk and Creating and Building a Program Object</td>
<td>53</td>
</tr>
<tr>
<td>Listing 2.5</td>
<td>Creating a Kernel</td>
<td>54</td>
</tr>
<tr>
<td>Listing 2.6</td>
<td>Creating Memory Objects</td>
<td>55</td>
</tr>
<tr>
<td>Listing 2.7</td>
<td>Setting the Kernel Arguments, Executing the Kernel, and Reading Back the Results</td>
<td>56</td>
</tr>
<tr>
<td>Listing 3.1</td>
<td>Enumerating the List of Platforms</td>
<td>66</td>
</tr>
<tr>
<td>Listing 3.2</td>
<td>Querying and Displaying Platform-Specific Information</td>
<td>67</td>
</tr>
<tr>
<td>Listing 3.3</td>
<td>Example of Querying and Displaying Platform-Specific Information</td>
<td>79</td>
</tr>
<tr>
<td>Listing 3.4</td>
<td>Using Platform, Devices, and Contexts—Simple Convolution Kernel</td>
<td>90</td>
</tr>
<tr>
<td>Listing 3.5</td>
<td>Example of Using Platform, Devices, and Contexts—Simple Convolution</td>
<td>91</td>
</tr>
<tr>
<td>Listing 6.1</td>
<td>Creating and Building a Program Object</td>
<td>221</td>
</tr>
<tr>
<td>Listing 6.2</td>
<td>Caching the Program Binary on First Run</td>
<td>229</td>
</tr>
<tr>
<td>Listing 6.3</td>
<td>Querying for and Storing the Program Binary</td>
<td>230</td>
</tr>
<tr>
<td>Listing 6.4</td>
<td>Example Program Binary for <em>HelloWorld.cl</em> (NVIDIA)</td>
<td>233</td>
</tr>
<tr>
<td>Listing 6.5</td>
<td>Creating a Program from Binary</td>
<td>235</td>
</tr>
<tr>
<td>Listing 7.1</td>
<td>Creating, Writing, and Reading Buffers and Sub-Buffers Example Kernel Code</td>
<td>262</td>
</tr>
<tr>
<td>Listing 7.2</td>
<td>Creating, Writing, and Reading Buffers and Sub-Buffers Example Host Code</td>
<td>262</td>
</tr>
<tr>
<td>Listing 8.1</td>
<td>Creating a 2D Image Object from a File</td>
<td>284</td>
</tr>
<tr>
<td>Listing 8.2</td>
<td>Creating a 2D Image Object for Output</td>
<td>285</td>
</tr>
<tr>
<td>Listing 8.3</td>
<td>Query for Device Image Support</td>
<td>291</td>
</tr>
<tr>
<td>Listing 8.4</td>
<td>Creating a Sampler Object</td>
<td>293</td>
</tr>
<tr>
<td>Listing 8.5</td>
<td>Gaussian Filter Kernel</td>
<td>295</td>
</tr>
<tr>
<td>Listing 8.6</td>
<td>Queue Gaussian Kernel for Execution</td>
<td>297</td>
</tr>
<tr>
<td>Listing 8.7</td>
<td>Read Image Back to Host Memory</td>
<td>300</td>
</tr>
<tr>
<td>Listing 8.8</td>
<td>Mapping Image Results to a Host Memory Pointer</td>
<td>307</td>
</tr>
<tr>
<td>Listing 12.1</td>
<td>Vector Add Example Program Using the C++ Wrapper API</td>
<td>379</td>
</tr>
<tr>
<td>Listing 13.1</td>
<td>Querying Platform and Device Profiles</td>
<td>384</td>
</tr>
<tr>
<td>Listing 14.1</td>
<td>Sequential Implementation of RGB Histogram</td>
<td>393</td>
</tr>
<tr>
<td>Listing 14.2</td>
<td>A Parallel Version of the RGB Histogram—Compute Partial Histograms</td>
<td>395</td>
</tr>
<tr>
<td>Listing 14.3</td>
<td>A Parallel Version of the RGB Histogram—Sum Partial Histograms</td>
<td>397</td>
</tr>
<tr>
<td>Listing 14.4</td>
<td>Host Code of CL API Calls to Enqueue Histogram Kernels</td>
<td>398</td>
</tr>
<tr>
<td>Listing 14.5</td>
<td>A Parallel Version of the RGB Histogram—Optimized Version</td>
<td>400</td>
</tr>
</tbody>
</table>
Listing 14.6  A Parallel Version of the RGB Histogram for Half-Float and Float Channels ...........................................403
Listing 15.1  An OpenCL Sobel Filter ..................................................408
Listing 15.2  An OpenCL Sobel Filter Producing a Grayscale Image .................................................................410
Listing 16.1  Data Structure and Interface for Dijkstra’s Algorithm .................................................................413
Listing 16.2  Pseudo Code for High-Level Loop That Executes Dijkstra’s Algorithm .............................................414
Listing 16.3  Kernel to Initialize Buffers before Each Run of Dijkstra’s Algorithm ..................................................415
Listing 16.4  Two Kernel Phases That Compute Dijkstra’s Algorithm .................................................................416
Listing 20.1  ImageFilter2D.py .............................................................489
Listing 20.2  Creating a Context ............................................................492
Listing 20.3  Loading an Image ..............................................................494
Listing 20.4  Creating and Building a Program ........................................495
Listing 20.5  Executing the Kernel ..........................................................496
Listing 20.6  Reading the Image into a Numpy Array ........................................496
Listing 21.1  A C Function Implementing Sequential Matrix Multiplication .......................................................500
Listing 21.2  A kernel to compute the matrix product of $A$ and $B$ summing the result into a third matrix, $C$. Each work-item is responsible for a single element of the $C$ matrix. The matrices are stored in global memory. . . 501
Listing 21.3  The Host Program for the Matrix Multiplication Program ...............................................................503
Listing 21.4  Each work-item updates a full row of $C$. The kernel code is shown as well as changes to the host code from the base host program in Listing 21.3. The only change required in the host code was to the dimensions of the NDRRange.

Listing 21.5  Each work-item manages the update to a full row of $C$, but before doing so the relevant row of the $A$ matrix is copied into private memory from global memory.

Listing 21.6  Each work-item manages the update to a full row of $C$. Private memory is used for the row of $A$ and local memory ($B_{wrk}$) is used by all work-items in a work-group to hold a column of $B$. The host code is the same as before other than the addition of a new argument for the $B$-column local memory.

Listing 21.7  Different Versions of the Matrix Multiplication Functions Showing the Permutations of the Loop Orderings

Listing 22.1  Sparse Matrix-Vector Multiplication OpenCL Kernels
During the past few years, heterogeneous computers composed of CPUs and GPUs have revolutionized computing. By matching different parts of a workload to the most suitable processor, tremendous performance gains have been achieved.

Much of this revolution has been driven by the emergence of many-core processors such as GPUs. For example, it is now possible to buy a graphics card that can execute more than a trillion floating point operations per second (teraflops). These GPUs were designed to render beautiful images, but for the right workloads, they can also be used as high-performance computing engines for applications from scientific computing to augmented reality.

A natural question is why these many-core processors are so fast compared to traditional single core CPUs. The fundamental driving force is innovative parallel hardware. Parallel computing is more efficient than sequential computing because chips are fundamentally parallel. Modern chips contain billions of transistors. Many-core processors organize these transistors into many parallel processors consisting of hundreds of floating point units. Another important reason for their speed advantage is new parallel software. Utilizing all these computing resources requires that we develop parallel programs. The efficiency gains due to software and hardware allow us to get more FLOPs per Watt or per dollar than a single-core CPU.

Computing systems are a symbiotic combination of hardware and software. Hardware is not useful without a good programming model. The success of CPUs has been tied to the success of their programming models, as exemplified by the C language and its successors. C nicely abstracts a sequential computer. To fully exploit heterogeneous computers, we need new programming models that nicely abstract a modern parallel computer. And we can look to techniques established in graphics as a guide to the new programming models we need for heterogeneous computing.

I have been interested in programming models for graphics for many years. It started in 1988 when I was a software engineer at PIXAR, where I developed the RenderMan shading language. A decade later graphics
systems became fast enough that we could consider developing shading languages for GPUs. With Kekoa Proudfoot and Bill Mark, we developed a real-time shading language, RTSL. RTSL ran on graphics hardware by compiling shading language programs into pixel shader programs, the assembly language for graphics hardware of the day. Bill Mark subsequently went to work at NVIDIA, where he developed Cg. More recently, I have been working with Tim Foley at Intel, who has developed a new shading language called Spark. Spark takes shading languages to the next level by abstracting complex graphics pipelines with new capabilities such as tesselation.

While developing these languages, I always knew that GPUs could be used for much more than graphics. Several other groups had demonstrated that graphics hardware could be used for applications beyond graphics. This led to the GPGPU (General-Purpose GPU) movement. The demonstrations were hacked together using the graphics library. For GPUs to be used more widely, they needed a more general programming environment that was not tied to graphics. To meet this need, we started the Brook for GPU Project at Stanford. The basic idea behind Brook was to treat the GPU as a data-parallel processor. Data-parallel programming has been extremely successful for parallel computing, and with Brook we were able to show that data-parallel programming primitives could be implemented on a GPU. Brook made it possible for a developer to write an application in a widely used parallel programming model.

Brook was built as a proof of concept. Ian Buck, a graduate student at Stanford, went on to NVIDIA to develop CUDA. CUDA extended Brook in important ways. It introduced the concept of cooperating thread arrays, or thread blocks. A cooperating thread array captured the locality in a GPU core, where a block of threads executing the same program could also communicate through local memory and synchronize through barriers. More importantly, CUDA created an environment for GPU Computing that has enabled a rich ecosystem of application developers, middleware providers, and vendors.

OpenCL (Open Computing Language) provides a logical extension of the core ideas from GPU Computing—the era of ubiquitous heterogeneous parallel computing. OpenCL has been carefully designed by the Khronos Group with input from many vendors and software experts. OpenCL benefits from the experience gained using CUDA in creating a software standard that can be implemented by many vendors. OpenCL implementations run now on widely used hardware, including CPUs and GPUs from NVIDIA, AMD, and Intel, as well as platforms based on DSPs and FPGAs.
By standardizing the programming model, developers can count on more software tools and hardware platforms.

What is most exciting about OpenCL is that it doesn’t only standardize what has been done, but represents the efforts of an active community that is pushing the frontier of parallel computing. For example, OpenCL provides innovative capabilities for scheduling tasks on the GPU. The developers of OpenCL have have combined the best features of task-parallel and data-parallel computing. I expect future versions of OpenCL to be equally innovative. Like its father, OpenGL, OpenCL will likely grow over time with new versions with more and more capability.

This book describes the complete OpenCL Programming Model. One of the coauthors, Aaftab, was the key mind behind the system. He has joined forces with other key designers of OpenCL to write an accessible authoritative guide. Welcome to the new world of heterogeneous computing.

—Pat Hanrahan  
Stanford University
Industry pundits love drama. New products don’t build on the status quo to make things better. They “revolutionize” or, better yet, define a “new paradigm.” And, of course, given the way technology evolves, the results rarely are as dramatic as the pundits make it seem.

Over the past decade, however, something revolutionary has happened. The drama is real. CPUs with multiple cores have made parallel hardware ubiquitous. GPUs are no longer just specialized graphics processors; they are heavyweight compute engines. And their combination, the so-called heterogeneous platform, truly is redefining the standard building blocks of computing.

We appear to be midway through a revolution in computing on a par with that seen with the birth of the PC. Or more precisely, we have the potential for a revolution because the high levels of parallelism provided by heterogeneous hardware are meaningless without parallel software; and the fact of the matter is that outside of specific niches, parallel software is rare.

To create a parallel software revolution that keeps pace with the ongoing (parallel) heterogeneous computing revolution, we need a parallel software industry. That industry, however, can flourish only if software can move between platforms, both cross-vendor and cross-generational. The solution is an industry standard for heterogeneous computing.

OpenCL is that industry standard. Created within the Khronos Group (known for OpenGL and other standards), OpenCL emerged from a collaboration among software vendors, computer system designers (including designers of mobile platforms), and microprocessor (embedded, accelerator, CPU, and GPU) manufacturers. It is an answer to the question “How can a person program a heterogeneous platform with the confidence that software created today will be relevant tomorrow?”

Born in 2008, OpenCL is now available from multiple sources on a wide range of platforms. It is evolving steadily to remain aligned with the latest microprocessor developments. In this book we focus on OpenCL 1.1. We describe the full scope of the standard with copious examples to explain how OpenCL is used in practice. Join us. Vive la révolution.


**Intended Audience**

This book is written by programmers for programmers. It is a pragmatic guide for people interested in writing code. We assume the reader is comfortable with C and, for parts of the book, C++. Finally, we assume the reader is familiar with the basic concepts of parallel programming. We assume our readers have a computer nearby so they can write software and explore ideas as they read. Hence, this book is overflowing with programs and fragments of code.

We cover the entire OpenCL 1.1 specification and explain how it can be used to express a wide range of parallel algorithms. After finishing this book, you will be able to write complex parallel programs that decompose a workload across multiple devices in a heterogeneous platform. You will understand the basics of performance optimization in OpenCL and how to write software that probes the hardware and adapts to maximize performance.

**Organization of the Book**

The OpenCL specification is almost 400 pages. It’s a dense and complex document full of tediously specific details. Explaining this specification is not easy, but we think that we’ve pulled it off nicely.

The book is divided into two parts. The first describes the OpenCL specification. It begins with two chapters to introduce the core ideas behind OpenCL and the basics of writing an OpenCL program. We then launch into a systematic exploration of the OpenCL 1.1 specification. The tone of the book changes as we incorporate reference material with explanatory discourse. The second part of the book provides a sequence of case studies. These range from simple pedagogical examples that provide insights into how aspects of OpenCL work to complex applications showing how OpenCL is used in serious application projects. The following provides more detail to help you navigate through the book:

**Part I: The OpenCL 1.1 Language and API**

- **Chapter 1, “An Introduction to OpenCL”:** This chapter provides a high-level overview of OpenCL. It begins by carefully explaining why heterogeneous parallel platforms are destined to dominate computing into the foreseeable future. Then the core models and concepts behind OpenCL are described. Along the way, the terminology used in OpenCL is presented, making this chapter an important one to read.
even if your goal is to skim through the book and use it as a reference guide to OpenCL.

• **Chapter 2, “HelloWorld: An OpenCL Example”:** Real programmers learn by writing code. Therefore, we complete our introduction to OpenCL with a chapter that explores a working OpenCL program. It has become standard to introduce a programming language by printing “hello world” to the screen. This makes no sense in OpenCL (which doesn't include a print statement). In the data-parallel programming world, the analog to “hello world” is a program to complete the element-wise addition of two arrays. That program is the core of this chapter. By the end of the chapter, you will understand OpenCL well enough to start writing your own simple programs. And we urge you to do exactly that. You can't learn a programming language by reading a book alone. Write code.

• **Chapter 3, “Platforms, Contexts, and Devices”:** With this chapter, we begin our systematic exploration of the OpenCL specification. Before an OpenCL program can do anything “interesting,” it needs to discover available resources and then prepare them to do useful work. In other words, a program must discover the platform, define the context for the OpenCL program, and decide how to work with the devices at its disposal. These important topics are explored in this chapter, where the OpenCL Platform API is described in detail.

• **Chapter 4, “Programming with OpenCL C”:** Code that runs on an OpenCL device is in most cases written using the OpenCL C programming language. Based on a subset of C99, the OpenCL C programming language provides what a kernel needs to effectively exploit an OpenCL device, including a rich set of vector instructions. This chapter explains this programming language in detail.

• **Chapter 5, “OpenCL C Built-In Functions”:** The OpenCL C programming language API defines a large and complex set of built-in functions. These are described in this chapter.

• **Chapter 6, “Programs and Kernels”:** Once we have covered the languages used to write kernels, we move on to the runtime API defined by OpenCL. We start with the process of creating programs and kernels. Remember, the word *program* is overloaded by OpenCL. In OpenCL, the word *program* refers specifically to the “dynamic library” from which the functions are pulled for the kernels.

• **Chapter 7, “Buffers and Sub-Buffers”:** In the next chapter we move to the buffer memory objects, one-dimensional arrays, including a careful discussion of sub-buffers. The latter is a new feature in
OpenCL 1.1, so programmers experienced with OpenCL 1.0 will find this chapter particularly useful.

- **Chapter 8, “Images and Samplers”:** Next we move to the very important topic of our other memory object, images. Given the close relationship between graphics and OpenCL, these memory objects are important for a large fraction of OpenCL programmers.

- **Chapter 9, “Events”:** This chapter presents a detailed discussion of the event model in OpenCL. These objects are used to enforce ordering constraints in OpenCL. At a basic level, events let you write concurrent code that generates correct answers regardless of how work is scheduled by the runtime. At a more algorithmically profound level, however, events support the construction of programs as directed acyclic graphs spanning multiple devices.

- **Chapter 10, “Interoperability with OpenGL”:** Many applications may seek to use graphics APIs to display the results of OpenCL processing, or even use OpenCL to postprocess scenes generated by graphics. The OpenCL specification allows interoperation with the OpenGL graphics API. This chapter will discuss how to set up OpenGL/OpenCL sharing and how data can be shared and synchronized.

- **Chapter 11, “Interoperability with Direct3D”:** The Microsoft family of platforms is a common target for OpenCL applications. When applications include graphics, they may need to connect to Microsoft’s native graphics API. In OpenCL 1.1, we define how to connect an OpenCL application to the DirectX 10 API. This chapter will demonstrate how to set up OpenCL/Direct3D sharing and how data can be shared and synchronized.

- **Chapter 12, “C++ Wrapper API”:** We then discuss the OpenCL C++ API Wrapper. This greatly simplifies the host programs written in C++, addressing automatic reference counting and a unified interface for querying OpenCL object information. Once the C++ interface is mastered, it’s hard to go back to the regular C interface.

- **Chapter 13, “OpenCL Embedded Profile”:** OpenCL was created for an unusually wide range of devices, with a reach extending from cell phones to the nodes in a massively parallel supercomputer. Most of the OpenCL specification applies without modification to each of these devices. There are a small number of changes to OpenCL, however, needed to fit the reduced capabilities of low-power processors used in embedded devices. This chapter describes these changes, referred to in the OpenCL specification as the OpenCL embedded profile.
Part II: OpenCL 1.1 Case Studies

- **Chapter 14, “Image Histogram”**: A histogram reports the frequency of occurrence of values within a data set. For example, in this chapter, we compute the histogram for R, G, and B channel values of a color image. To generate a histogram in parallel, you compute values over local regions of a data set and then sum these local values to generate the final result. The goal of this chapter is twofold: (1) we demonstrate how to manipulate images in OpenCL, and (2) we explore techniques to efficiently carry out a histogram’s global summation within an OpenCL program.

- **Chapter 15, “Sobel Edge Detection Filter”**: The Sobel edge filter is a directional edge detector filter that computes image gradients along the x- and y-axes. In this chapter, we use a kernel to apply the Sobel edge filter as a simple example of how kernels work with images in OpenCL.

- **Chapter 16, “Parallelizing Dijkstra’s Single-Source Shortest-Path Graph Algorithm”**: In this chapter, we present an implementation of Dijkstra’s Single-Source Shortest-Path graph algorithm implemented in OpenCL capable of utilizing both CPU and multiple GPU devices. Graph data structures find their way into many problems, from artificial intelligence to neuroimaging. This particular implementation was developed as part of FreeSurfer, a neuroimaging application, in order to improve the performance of an algorithm that measures the curvature of a triangle mesh structural reconstruction of the cortical surface of the brain. This example is illustrative of how to work with multiple OpenCL devices and split workloads across CPUs, multiple GPUs, or all devices at once.

- **Chapter 17, “Cloth Simulation in the Bullet Physics SDK”**: Physics simulation is a growing addition to modern video games, and in this chapter we present an approach to simulating cloth, such as a warrior’s clothing, using OpenCL that is part of the Bullet Physics SDK. There are many ways of simulating soft bodies; the simulation method used in Bullet is similar to a mass/spring model and is optimized for execution on modern GPUs while integrating smoothly with other Bullet SDK components that are not written in OpenCL. We show an important technique, called batching, that transforms the particle meshes for performant execution on wide SIMD architectures, such as the GPU, while preserving dependences within the mass/spring model.
• Chapter 18, “Simulating the Ocean with Fast Fourier Transform”: In this chapter we present the details of AMD’s Ocean simulation. Ocean is an OpenCL demonstration that uses an inverse discrete Fourier transform to simulate, in real time, the sea. The fast Fourier transform is applied to random noise, generated over time as a frequency-dependent phase shift. We describe an implementation based on the approach originally developed by Jerry Tessendorf that has appeared in a number of feature films, including *Waterworld*, *Titanic*, and *Fifth Element*. We show the development of an optimized 2D DFFT, including a number of important optimizations useful when programming with OpenCL, and the integration of this algorithm into the application itself and using interoperability between OpenCL and OpenGL.

• Chapter 19, “Optical Flow”: In this chapter, we present an implementation of optical flow in OpenCL, which is a fundamental concept in computer vision that describes motion in images. Optical flow has uses in image stabilization, temporal upsampling, and as an input to higher-level algorithms such as object tracking and gesture recognition. This chapter presents the pyramidal Lucas-Kanade optical flow algorithm in OpenCL. The implementation demonstrates how image objects can be used to access texture features of GPU hardware. We will show how the texture-filtering hardware on the GPU can be used to perform linear interpolation of data, achieve the required sub-pixel accuracy, and thereby provide significant speedups. Additionally, we will discuss how shared memory can be used to cache data that is repeatedly accessed and how early kernel exit techniques provide additional efficiency.

• Chapter 20, “Using OpenCL with PyOpenCL”: The purpose of this chapter is to introduce you to the basics of working with OpenCL in Python. The majority of the book focuses on using OpenCL from C/C++, but bindings are available for other languages including Python. In this chapter, PyOpenCL is introduced by walking through the steps required to port the Gaussian image-filtering example from Chapter 8 to Python. In addition to covering the changes required to port from C++ to Python, the chapter discusses some of the advantages of using OpenCL in a dynamically typed language such as Python.

• Chapter 21, “Matrix Multiplication with OpenCL”: In this chapter, we discuss a program that multiplies two square matrices. The program is very simple, so it is easy to follow the changes made to the program as we optimize its performance. These optimizations focus
on the OpenCL memory model and how we can work with the model to minimize the cost of data movement in an OpenCL program.

• Chapter 22, “Sparse Matrix-Vector Multiplication”: In this chapter, we describe an optimized implementation of the Sparse Matrix-Vector Multiplication algorithm using OpenCL. Sparse matrices are defined as large, two-dimensional matrices in which the vast majority of the elements of the matrix are equal to zero. They are used to characterize and solve problems in a wide variety of domains such as computational fluid dynamics, computer graphics/vision, robotics/kinematics, financial modeling, acoustics, and quantum chemistry. The implementation demonstrates OpenCL’s ability to bridge the gap between hardware-specific code (fast, but not portable) and single-source code (very portable, but slow), yielding a high-performance, efficient implementation on a variety of hardware that is almost as fast as a hardware-specific implementation. These results are accomplished with kernels written in OpenCL C that can be compiled and run on any conforming OpenCL platform.

Appendix

• Appendix A, “Summary of OpenCL 1.1”: The OpenCL specification defines an overwhelming collection of functions, named constants, and types. Even expert OpenCL programmers need to look up these details when writing code. To aid in this process, we’ve included an appendix where we pull together all these details in one place.

Example Code

This book is filled with example programs. You can download many of the examples from the book’s Web site at www.openclprogrammingguide.com.

Errata

If you find something in the book that you believe is in error, please send us a note at errors@opencl-book.com. The list of errata for the book can be found on the book’s Web site at www.openclprogrammingguide.com.
This page intentionally left blank
Acknowledgments

From Aaftab Munshi

It has been a great privilege working with Ben, Dan, Tim, and James on this book. I want to thank our reviewers, Andrew Brownsword, Yahya H. Mizra, Dave Shreiner, and Michael Thurston, who took the time to review this book and provided valuable feedback that has improved the book tremendously. I want to thank our editor at Pearson, Debra Williams Cauley, for all her help in making this book happen.

I also want to thank my daughters, Hannah and Ellie, and the love of my life, Karen, without whom this book would not be possible.

From Benedict R. Gaster

I would like to thank AMD for supporting my work on OpenCL. There are four people in particular who have guided my understanding of the GPGPU revolution: Mike Houston, Justin Hensley, Lee Howes, and Laurent Morichetti.

This book would not have been possible without the continued enjoyment of life in Santa Cruz and going to the beach with Miranda, Maude, Polly, and Meg. Thanks!

From Timothy G. Mattson

I would like to thank Intel for giving me the freedom to pursue work on OpenCL. In particular, I want to thank Aaron Lefohn of Intel for bringing me into this project in the early days as it was just getting started. Most of all, however, I want to thank the amazing people in the OpenCL working group. I have learned a huge amount from this dedicated team of professionals.
From James Fung

It’s been a privilege to work alongside my coauthors and contribute to this book. I would also like to thank NVIDIA for all its support during writing as well as family and friends for their support and encouragement.

From Dan Ginsburg

I would like to thank Dr. Rudolph Pienaar and Dr. Ellen Grant at Children’s Hospital Boston for supporting me in writing this book and for their valuable contributions and insights. It has been an honor and a great privilege to work on this book with Affie, Ben, Tim, and James, who represent some of the sharpest minds in the parallel computing business. I also want to thank our editor, Debra Williams Cauley, for her unending patience and dedication, which were critical to the success of this project.
About the Authors


Benedict R. Gaster is a software architect working on programming models for next-generation heterogeneous processors, in particular looking at high-level abstractions for parallel programming on the emerging class of processors that contain both CPUs and accelerators such as GPUs. Benedict has contributed extensively to the OpenCL’s design and has represented AMD at the Khronos Group open standard consortium. Benedict has a Ph.D. in computer science for his work on type systems for extensible records and variants. He has been working at AMD since 2008.

Timothy G. Mattson is an old-fashioned parallel programmer, having started in the mid-eighties with the Caltech Cosmic Cube and continuing to the present. Along the way, he has worked with most classes of parallel computers (vector supercomputers, SMP, VLIW, NUMA, MPP, clusters, and many-core processors). Tim has published extensively, including the books *Patterns for Parallel Programming* (with Beverly Sanders and Berna Massingill, published by Addison-Wesley, 2004) and *An Introduction to Concurrency in Programming Languages* (with Matthew J. Sottile and Craig E Rasmussen, published by CRC Press, 2009). Tim has a Ph.D. in chemistry for his work on molecular scattering theory. He has been working at Intel since 1993.

James Fung has been developing computer vision on the GPU as it progressed from graphics to general-purpose computation. James has a Ph.D. in electrical and computer engineering from the University of Toronto and numerous IEEE and ACM publications in the areas of parallel GPU Computer Vision and Mediated Reality. He is currently a Developer Technology Engineer at NVIDIA, where he examines computer vision and image processing on graphics hardware.

Dan Ginsburg currently works at Children’s Hospital Boston as a Principal Software Architect in the Fetal-Neonatal Neuroimaging and Development Science Center, where he uses OpenCL for accelerating
neuroimaging algorithms. Previously, he worked for Still River Systems
developing GPU-accelerated image registration software for the Monarch
250 proton beam radiotherapy system. Dan was also Senior Member of
Technical Staff at AMD, where he worked for over eight years in a vari-
eity of roles, including developing OpenGL drivers, creating desktop and
hand-held 3D demos, and leading the development of handheld GPU
developer tools. Dan holds a B.S. in computer science from Worcester
Polytechnic Institute and an M.B.A. from Bentley University.
Chapter 4

Programming with OpenCL C

The OpenCL C programming language is used to create programs that describe data-parallel kernels and tasks that can be executed on one or more heterogeneous devices such as CPUs, GPUs, and other processors referred to as accelerators such as DSPs and the Cell Broadband Engine (B.E.) processor. An OpenCL program is similar to a dynamic library, and an OpenCL kernel is similar to an exported function from the dynamic library. Applications directly call the functions exported by a dynamic library from their code. Applications, however, cannot call an OpenCL kernel directly but instead queue the execution of the kernel to a command-queue created for a device. The kernel is executed asynchronously with the application code running on the host CPU.

OpenCL C is based on the ISO/IEC 9899:1999 C language specification (referred to in short as C99) with some restrictions and specific extensions to the language for parallelism. In this chapter, we describe how to write data-parallel kernels using OpenCL C and cover the features supported by OpenCL C.

Writing a Data-Parallel Kernel Using OpenCL C

As described in Chapter 1, data parallelism in OpenCL is expressed as an N-dimensional computation domain, where $N = 1, 2, \text{ or } 3$. The N-D domain defines the total number of work-items that can execute in parallel. Let’s look at how a data-parallel kernel would be written in OpenCL C by taking a simple example of summing two arrays of floats. A sequential version of this code would perform the sum by summing individual elements of both arrays inside a for loop:

```c
void scalar_add (int n, const float *a, const float *b, float *result) {
    int i;
```
for (i=0; i<n; i++)
    result[i] = a[i] + b[i];

A data-parallel version of the code in OpenCL C would look like this:

```
kernel void
scalar_add (global const float *a,
            global const float *b,
            global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] + b[id];
}
```

The `scalar_add` function declaration uses the `kernel` qualifier to indicate that this is an OpenCL C kernel. Note that the `scalar_add` kernel includes only the code to compute the sum of each individual element, aka the inner loop. The N-D domain will be a one-dimensional domain set to `n`. The kernel is executed for each of the `n` work-items to produce the sum of arrays `a` and `b`. In order for this to work, each executing work-item needs to know which individual elements from arrays `a` and `b` need to be summed. This must be a unique value for each work-item and should be derived from the N-D domain specified when queuing the kernel for execution. The `get_global_id(0)` returns the one-dimensional global ID for each work-item. Ignore the `global` qualifiers specified in the kernel for now; they will be discussed later in this chapter.

Figure 4.1 shows how `get_global_id` can be used to identify a unique work-item from the list of work-items executing a kernel.

![Figure 4.1](mapping.png)

**Figure 4.1** Mapping `get_global_id` to a work-item
The OpenCL C language with examples is described in depth in the sections that follow. The language is derived from C99 with restrictions that are described at the end of this chapter.

OpenCL C also adds the following features to C99:

- **Vector data types.** A number of OpenCL devices such as Intel SSE, Altivec for POWER and Cell, and ARM NEON support a vector instruction set. This vector instruction set is accessed in C/C++ code through built-in functions (some of which may be device-specific) or device-specific assembly instructions. In OpenCL C, vector data types can be used in the same way scalar types are used in C. This makes it much easier for developers to write vector code because similar operators can be used for both vector and scalar data types. It also makes it easy to write portable vector code because the OpenCL compiler is now responsible for mapping the vector operations in OpenCL C to the appropriate vector ISA for a device. Vectorizing code also helps improve memory bandwidth because of regular memory accesses and better coalescing of these memory accesses.

- **Address space qualifiers.** OpenCL devices such as GPUs implement a memory hierarchy. The address space qualifiers are used to identify a specific memory region in the hierarchy.

- **Additions to the language for parallelism.** These include support for work-items, work-groups, and synchronization between work-items in a work-group.

- **Images.** OpenCL C adds image and sampler data types and built-in functions to read and write images.

- **An extensive set of built-in functions such as math, integer, geometric, and relational functions.** These are described in detail in Chapter 5.

### Scalar Data Types

The C99 scalar data types supported by OpenCL C are described in Table 4.1. Unlike C, OpenCL C describes the sizes, that is, the exact number of bits for the integer and floating-point data types.
### Table 4.1  Built-In Scalar Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>A conditional data type that is either true or false. The value <code>true</code> expands to the integer constant 1, and the value <code>false</code> expands to the integer constant 0.</td>
</tr>
<tr>
<td>char</td>
<td>A signed two's complement 8-bit integer.</td>
</tr>
<tr>
<td>unsigned char, uchar</td>
<td>An unsigned 8-bit integer.</td>
</tr>
<tr>
<td>short</td>
<td>A signed two's complement 16-bit integer.</td>
</tr>
<tr>
<td>unsigned short, ushort</td>
<td>An unsigned 16-bit integer.</td>
</tr>
<tr>
<td>int</td>
<td>A signed two's complement 32-bit integer.</td>
</tr>
<tr>
<td>unsigned int, uint</td>
<td>An unsigned 32-bit integer.</td>
</tr>
<tr>
<td>long</td>
<td>A signed two's complement 64-bit integer.</td>
</tr>
<tr>
<td>unsigned long, ulong</td>
<td>An unsigned 64-bit integer.</td>
</tr>
<tr>
<td>float</td>
<td>A 32-bit floating-point. The float data type must conform to the IEEE 754 single-precision storage format.</td>
</tr>
<tr>
<td>double</td>
<td>A 64-bit floating-point. The <code>double</code> data type must conform to the IEEE 754 double-precision storage format. This is an optional format and is available only if the double-precision extension (<code>cl_khr_fp64</code>) is supported by the device.</td>
</tr>
<tr>
<td>half</td>
<td>A 16-bit floating-point. The <code>half</code> data type must conform to the IEEE 754-2008 half-precision storage format.</td>
</tr>
<tr>
<td>size_t</td>
<td>The unsigned integer type of the result of the <code>sizeof</code> operator. This is a 32-bit unsigned integer if the address space of the device is 32 bits and is a 64-bit unsigned integer if the address space of the device is 64 bits.</td>
</tr>
<tr>
<td>ptrdiff_t</td>
<td>A signed integer type that is the result of subtracting two pointers. This is a 32-bit signed integer if the address space of the device is 32 bits and is a 64-bit signed integer if the address space of the device is 64 bits.</td>
</tr>
<tr>
<td>intptr_t</td>
<td>A signed integer type with the property that any valid pointer to <code>void</code> can be converted to this type, then converted back to a pointer to <code>void</code>, and the result will compare equal to the original pointer.</td>
</tr>
</tbody>
</table>
The **half** Data Type

The `half` data type must be IEEE 754-2008-compliant. `half` numbers have 1 sign bit, 5 exponent bits, and 10 mantissa bits. The interpretation of the sign, exponent, and mantissa is analogous to that of IEEE 754 floating-point numbers. The exponent bias is 15. The `half` data type must represent finite and normal numbers, denormalized numbers, infinities, and NaN. Denormalized numbers for the `half` data type, which may be generated when converting a `float` to a `half` using the built-in function `vstore_half` and converting a `half` to a `float` using the built-in function `vload_half`, cannot be flushed to zero.

Conversions from `float` to `half` correctly round the mantissa to 11 bits of precision. Conversions from `half` to `float` are lossless; all `half` numbers are exactly representable as `float` values.

The `half` data type can be used only to declare a pointer to a buffer that contains `half` values. A few valid examples are given here:

```c
void
bar(global half *p)
{
    ...
}

void
foo(global half *pg, local half *pl)
{
    global half *ptr;
    int offset;

    ptr = pg + offset;
    bar(ptr);
}
```

### Table 4.1  Built-In Scalar Data Types (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uintptr_t</code></td>
<td>An unsigned integer type with the property that any valid pointer to <code>void</code> can be converted to this type, then converted back to a pointer to <code>void</code>, and the result will compare equal to the original pointer.</td>
</tr>
<tr>
<td><code>void</code></td>
<td>The <code>void</code> type constitutes an empty set of values; it is an incomplete type that cannot be completed.</td>
</tr>
</tbody>
</table>
Following is an example that is not a valid usage of the `half` type:

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

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

Loads from a pointer to a `half` and stores to a pointer to a `half` can be performed using the `vload_half`, `vload_halfn`, `vloada_halfn` and `vstore_half`, `vstore_halfn`, and `vstorea_halfn` functions, respectively. The load functions read scalar or vector `half` values from memory and convert them to a scalar or vector float value. The store functions take a scalar or vector float value as input, convert it to a `half` scalar or vector value (with appropriate rounding mode), and write the `half` scalar or vector value to memory.

### Vector Data Types

For the scalar integer and floating-point data types described in Table 4.1, OpenCL C adds support for vector data types. The vector data type is defined with the type name, that is, `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `float`, `long`, or `ulong` followed by a literal value `n` that defines the number of elements in the vector. Supported values of `n` are 2, 3, 4, 8, and 16 for all vector data types. Optionally, vector data types are also defined for `double` and `half`. These are available only if the device supports the double-precision and half-precision extensions. The supported vector data types are described in Table 4.2.

Variables declared to be a scalar or vector data type are always aligned to the size of the data type used in bytes. Built-in data types must be aligned to a power of 2 bytes in size. A built-in data type that is not a power of 2 bytes in size must be aligned to the next-larger power of 2. This rule does not apply to structs or unions.

For example, a `float4` variable will be aligned to a 16-byte boundary and a `char2` variable will be aligned to a 2-byte boundary. For 3-component vector data types, the size of the data type is $4 \times \text{sizeof}(\text{component})$. This means that a 3-component vector data type will be aligned to a $4 \times \text{sizeof}(\text{component})$ boundary.

The OpenCL compiler is responsible for aligning data items appropriately as required by the data type. The only exception is for an argument to a
Table 4.2  Built-In Vector Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charn</td>
<td>A vector of n 8-bit signed integer values</td>
</tr>
<tr>
<td>ucharn</td>
<td>A vector of n 8-bit unsigned integer values</td>
</tr>
<tr>
<td>shortn</td>
<td>A vector of n 16-bit signed integer values</td>
</tr>
<tr>
<td>ushortn</td>
<td>A vector of n 16-bit unsigned integer values</td>
</tr>
<tr>
<td>intn</td>
<td>A vector of n 32-bit signed integer values</td>
</tr>
<tr>
<td>uintn</td>
<td>A vector of n 32-bit unsigned integer values</td>
</tr>
<tr>
<td>longn</td>
<td>A vector of n 64-bit signed integer values</td>
</tr>
<tr>
<td>ulongn</td>
<td>A vector of n 64-bit unsigned integer values</td>
</tr>
<tr>
<td>floatn</td>
<td>A vector of n 32-bit floating-point values</td>
</tr>
<tr>
<td>doublen</td>
<td>A vector of n 64-bit floating-point values</td>
</tr>
<tr>
<td>halfn</td>
<td>A vector of n 16-bit floating-point values</td>
</tr>
</tbody>
</table>

kernel function that is declared to be a pointer to a data type. For such functions, the compiler can assume that the pointee is always appropriately aligned as required by the data type.

For application convenience and to ensure that the data store is appropriately aligned, the data types listed in Table 4.3 are made available to the application.

Table 4.3  Application Data Types

<table>
<thead>
<tr>
<th>Type in OpenCL Language</th>
<th>API Type for Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>cl_char</td>
</tr>
<tr>
<td>uchar</td>
<td>cl_uchar</td>
</tr>
<tr>
<td>short</td>
<td>cl_short</td>
</tr>
<tr>
<td>ushort</td>
<td>cl_ushort</td>
</tr>
<tr>
<td>int</td>
<td>cl_int</td>
</tr>
</tbody>
</table>

continues
Vector Literals

Vector literals can be used to create vectors from a list of scalars, vectors, or a combination of scalar and vectors. A vector literal can be used either as a vector initializer or as a primary expression. A vector literal cannot be used as an l-value.

A vector literal is written as a parenthesized vector type followed by a parenthesized comma-delimited list of parameters. A vector literal operates as an overloaded function. The forms of the function that are available are the set of possible argument lists for which all arguments have

<table>
<thead>
<tr>
<th>Type in OpenCL Language</th>
<th>API Type for Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint</td>
<td>cl_uint</td>
</tr>
<tr>
<td>long</td>
<td>cl_long</td>
</tr>
<tr>
<td>ulong</td>
<td>cl_ulong</td>
</tr>
<tr>
<td>float</td>
<td>cl_float</td>
</tr>
<tr>
<td>double</td>
<td>cl_double</td>
</tr>
<tr>
<td>half</td>
<td>cl_half</td>
</tr>
<tr>
<td>char</td>
<td>cl_char</td>
</tr>
<tr>
<td>uchar</td>
<td>cl_uchar</td>
</tr>
<tr>
<td>short</td>
<td>cl_short</td>
</tr>
<tr>
<td>ushort</td>
<td>cl_ushort</td>
</tr>
<tr>
<td>int</td>
<td>cl_int</td>
</tr>
<tr>
<td>uint</td>
<td>cl_uintn</td>
</tr>
<tr>
<td>long</td>
<td>cl_longn</td>
</tr>
<tr>
<td>ulong</td>
<td>cl_ulongn</td>
</tr>
<tr>
<td>float</td>
<td>cl_floata</td>
</tr>
<tr>
<td>double</td>
<td>cl_doublea</td>
</tr>
<tr>
<td>half</td>
<td>cl_halfa</td>
</tr>
</tbody>
</table>

Table 4.3 Application Data Types (Continued)
the same element type as the result vector, and the total number of elements is equal to the number of elements in the result vector. In addition, a form with a single scalar of the same type as the element type of the vector is available. For example, the following forms are available for float4:

(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 )

Operands are evaluated by standard rules for function evaluation, except that no implicit scalar widening occurs. The operands are assigned to their respective positions in the result vector as they appear in memory order. That is, the first element of the first operand is assigned to result.x, the second element of the first operand (or the first element of the second operand if the first operand was a scalar) is assigned to result.y, and so on. If the operand is a scalar, the operand is replicated across all lanes of the result vector.

The following example shows a vector float4 created from a list of scalars:

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

The following example shows a vector uint4 created from a scalar, which is replicated across the components of the vector:

uint4   u = (uint4)(1); // u will be (1, 1, 1, 1)

The following examples show more complex combinations of a vector being created using a scalar and smaller vector types:

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);

The following examples describe how not to create vector literals. All of these examples should result in a compilation error.

float4  f = (float4)(1.0f, 2.0f);
float4  f = (float2)(1.0f, 2.0f);
float4  f = (float4)(1.0f, (float2)(2.0f, 3.0f));
Vector Components

The components of vector data types with 1 to 4 components (aka elements) can be addressed as `<vector>.xyzw`. Table 4.4 lists the components that can be accessed for various vector types.

**Table 4.4  Accessing Vector Components**

<table>
<thead>
<tr>
<th>Vector Data Types</th>
<th>Accessible Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>char2, uchar2, short2, ushort2, int2, uint2, long2, ulong2, float2</td>
<td>.xy</td>
</tr>
<tr>
<td>char3, uchar3, short3, ushort3, int3, uint3, long3, ulong3, float3</td>
<td>.xyz</td>
</tr>
<tr>
<td>char4, uchar4, short4, ushort4, int4, uint4, long4, ulong4, float4</td>
<td>.xyzw</td>
</tr>
<tr>
<td>double2, half2</td>
<td>.xy</td>
</tr>
<tr>
<td>double3, half3</td>
<td>.xyz</td>
</tr>
<tr>
<td>double4, half4</td>
<td>.xyzw</td>
</tr>
</tbody>
</table>

Accessing components beyond those declared for the vector type is an error. The following describes legal and illegal examples of accessing vector components:

```c
float2 pos;
pos.x = 1.0f; // is legal
=pos.z = 1.0f; // is illegal
float3 pos;
pos.z = 1.0f; // is legal
=pos.w = 1.0f; // is illegal
```

The component selection syntax allows multiple components to be selected by appending their names after the period (.). A few examples that show how to use the component selection syntax are given here:

```c
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);
```
The component selection syntax also allows components to be permuted or replicated as shown in the following examples:

```c
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.xxxy;  // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

Vector components can also be accessed using a numeric index to refer to the appropriate elements in the vector. The numeric indices that can be used are listed in Table 4.5.

### Table 4.5 Numeric Indices for Built-In Vector Data Types

<table>
<thead>
<tr>
<th>Vector Components</th>
<th>Usable Numeric Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-component</td>
<td>0, 1</td>
</tr>
<tr>
<td>3-component</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>4-component</td>
<td>0, 1, 2, 3</td>
</tr>
<tr>
<td>8-component</td>
<td>0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>16-component</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F</td>
</tr>
</tbody>
</table>

All numeric indices must be preceded by the letter s or S. In the following example `f.s0` refers to the first element of the float8 variable `f` and `f.s7` refers to the eighth element of the float8 variable `f`:

```c
float8 f
```

In the following example `x.sa` (or `x.sA`) refers to the eleventh element of the float16 variable `x` and `x.sf` (or `x.sF`) refers to the sixteenth element of the float16 variable `x`:

```c
float16 x
```

The numeric indices cannot be intermixed with the `.xyzw` notation. For example:

```c
float4 f;
float4 v_A = f.xs123;  // is illegal
float4 v_B = f.s012w;  // is illegal
```

Vector data types can use the `.lo` (or `.odd`) and `.hi` (or `.even`) suffixes to get smaller vector types or to combine smaller vector types into a larger
vector type. Multiple levels of .lo (or .odd) and .hi (or .even) suffixes can be used until they refer to a scalar type.

The .lo suffix refers to the lower half of a given vector. The .hi suffix refers to the upper half of a given vector. The .odd suffix refers to the odd elements of a given vector. The .even suffix refers to the even elements of a given vector. Some examples to illustrate this concept are given here:

```c
float4 vf;

float2 low = vf.lo;    // returns vf.xy
float2 high = vf.hi;   // returns vf.zw
float x = low.low;     // returns low.x
float y = low.hi;      // returns low.y

float2 odd = vf.odd;   // returns vf.yw
float2 even = vf.even; // returns vf.xz
```

For a 3-component vector, the suffixes .lo (or .odd) and .hi (or .even) operate as if the 3-component vector were a 4-component vector with the value in the w component undefined.

## Other Data Types

The other data types supported by OpenCL C are described in Table 4.6.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image2d_t</td>
<td>A 2D image type.</td>
</tr>
<tr>
<td>image3d_t</td>
<td>A 3D image type.</td>
</tr>
<tr>
<td>sampler_t</td>
<td>An image sampler type.</td>
</tr>
<tr>
<td>event_t</td>
<td>An event type. These are used by built-in functions that perform async copies from global to local memory and vice versa. Each async copy operation returns an event and takes an event to wait for that identifies a previous async copy operation.</td>
</tr>
</tbody>
</table>
There are a few restrictions on the use of image and sampler types:

- The image and samplers types are defined only if the device supports images.

- Image and sampler types cannot be declared as arrays. Here are a couple of examples that show these illegal use cases:

```c
kernel void
foo(image2d_t imgA[10]) {  // error. images cannot be declared as arrays
    image2d_t imgB[4];     // error. images cannot be declared as arrays
    ...
}

kernel void
foo(sampler_t smpA[10]) {  // error. samplers cannot be declared as arrays
    sampler_t smpB[4];     // error. samplers cannot be declared as arrays
    ...
}
```

- The `image2d_t`, `image3d_t`, and `sampler_t` data types cannot be declared in a struct.

- Variables cannot be declared to be pointers of `image2d_t`, `image3d_t`, and `sampler_t` data types.

### Derived Types

The C99 derived types (arrays, structs, unions, and pointers) constructed from the built-in data types described in Tables 4.1 and 4.2 are supported. There are a few restrictions on the use of derived types:

- The struct type cannot contain any pointers if the struct or pointer to a struct is used as an argument type to a kernel function. For example, the following use case is invalid:

```c
typedef struct {
    int x;
    global float *f;
} mystruct_t;
```
• The struct type can contain pointers only if the struct or pointer to a struct is used as an argument type to a non-kernel function or declared as a variable inside a kernel or non-kernel function. For example, the following use case is valid:

```c
void
my_func(mystruct_t *p)
{
    ...
}
```

```c
kernel void
foo(global int *p1, global float *p2)
{
    mystruct_t s;

    s.x = p1[get_global_id(0)];
    s.f = p2;
    my_func(&s);
}
```

---

**Implicit Type Conversions**

Implicit type conversion is an automatic type conversion done by the compiler whenever data from different types is intermixed. Implicit conversions of scalar built-in types defined in Table 4.1 (except `void`, `double`, and `half`) are supported. When an implicit conversion is done, it is not just a reinterpretation of the expression’s value but a conversion of that value to an equivalent value in the new type.

Consider the following example:

```c
float f = 3;  // implicit conversion to float value 3.0
int   i = 5.23f;  // implicit conversion to integer value 5
```

---

1 Unless the double-precision extension (`cl_khr_fp64`) is supported by the device.

2 Unless the half-precision extension (`cl_khr_fp16`) is supported by the device.
In this example, the value 3 is converted to a float value 3.0f and then assigned to \textit{f}. The value 5.23f is converted to an int value 5 and then assigned to \textit{i}. In the second example, the fractional part of the float value is dropped because integers cannot support fractional values; this is an example of an unsafe type conversion.

\textbf{Warning} Note that some type conversions are inherently unsafe, and if the compiler can detect that an unsafe conversion is being implicitly requested, it will issue a warning.

Implicit conversions for pointer types follow the rules described in the C99 specification. Implicit conversions between built-in vector data types are disallowed. For example:

\begin{verbatim}
float4 f;
int4 i;

f = i; // illegal implicit conversion between vector data types
\end{verbatim}

There are graphics shading languages such as OpenGL Shading Language (GLSL) and the DirectX Shading Language (HLSL) that do allow implicit conversions between vector types. However, prior art for vector casts in C doesn't support conversion casts. The \textit{AltiVec Technology Programming Interface Manual} (www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf?fsrch=1), Section 2.4.6, describes the function of casts between vector types. The casts are conversion-free. Thus, any conforming AltiVec compiler has this behavior. Examples include XL C, GCC, MrC, Metrowerks, and Green Hills. IBM's Cell SPE C language extension (C/C++ Language Extensions for Cell Broadband Engine Architecture; see Section 1.4.5) has the same behavior. GCC and ICC have adopted the conversion-free cast model for SSE (http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Vector-Extensions.html#Vector-Extensions). The following code example shows the behavior of these compilers:

\begin{verbatim}
#include <stdio.h>

// Declare some vector types. This should work on most compilers
// that try to be GCC compatible. Alternatives are provided
// for those that don't conform to GCC behavior in vector
// type declaration.
// Here a vFloat is a vector of four floats, and
// a vInt is a vector of four 32-bit ints.
#if 1
    // This should work on most compilers that try
    // to be GCC compatible
    // cc main.c -Wall -pedantic
typedef float vFloat __attribute__ ((__vector_size__(16)));
#endif
\end{verbatim}
typedef int   vInt   __attribute__ ((__vector_size__(16)));
#define init_vFloat(a, b, c, d) (const vFloat) {a, b, c, d}
#else
//Not GCC compatible
#if defined( __SSE2__ )
// depending on compiler you might need to pass
// something like -msse2 to turn on SSE2
#include <emmintrin.h>
typedef __m128  vFloat;
typedef __m128i vInt;
static inline vFloat init_vFloat(float a, float b,
float c, float d);
static inline vFloat init_vFloat(float a, float b,
float c, float d)
{ union{ vFloat v; float f[4];}u;
u.f[0] = a; u.f[1] = b;
return u.v;
}
#elif defined( __VEC__ )
// depending on compiler you might need to pass
// something like -faltivec or -maltivec or
// "Enable AltiVec Extensions" to turn this part on
#include <altivec.h>
typedef vector float vFloat;
typedef vector int   vInt;
#if 1
// for compliant compilers
#define init_vFloat(a, b, c, d) 
    (const vFloat) (a, b, c, d)
#else
// for FSF GCC
#define init_vFloat(a, b, c, d) 
    (const vFloat) (a, b, c, d)
#endif
#endif
#endif
#endif

void
print_vInt(vInt v)
{
    union{ vInt v; int i[4]; }u;
u.v = v;

    printf("vInt: 0x%8.8x 0x%8.8x 0x%8.8x 0x%8.8x\n", 
    u.i[0], u.i[1], u.i[2], u.i[3]);
}
void
print_vFloat(vFloat v)
{
    union{ vFloat v; float i[4]; }u;
    u.v = v;

    printf("vFloat: %f %f %f %f\n", u.i[0], u.i[1], u.i[2], u.i[3]);
}

int
main(void)
{
    vFloat  f = init_vFloat(1.0f, 2.0f, 3.0f, 4.0f);
    vInt    i;

    print_vFloat(f);

    printf("assign with cast:  vInt i = (vInt) f;\n" );
    i = (vInt) f;

    print_vInt(i);

    return 0;
}

The output of this code example demonstrates that conversions between vector data types implemented by some C compilers\(^3\) such as GCC are cast-free.

vFloat: 1.000000 2.000000 3.000000 4.000000
assign with cast:  vInt i = (vInt) f;
vInt: 0x3f800000 0x40000000 0x40400000 0x40800000

So we have prior art in C where casts between vector data types do not perform conversions as opposed to graphics shading languages that do perform conversions. The OpenCL working group decided it was best to make implicit conversions between vector data types illegal. It turns out that this was the right thing to do for other reasons, as discussed in the section “Explicit Conversions” later in this chapter.

\(^3\) Some fiddling with compiler flags to get the vector extensions turned on may be required, for example, -msse2 or -faltivec. You might need to play with the #ifs. The problem is that there is no portable way to declare a vector type. Getting rid of the sort of portability headaches at the top of the code example is one of the major value-adds of OpenCL.
Usual Arithmetic Conversions

Many operators that expect operands of arithmetic types (integer or floating-point types) cause conversions and yield result types in a similar way. The purpose is to determine a common real type for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. For this purpose, all vector types are considered to have a higher conversion rank than scalars. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the usual arithmetic conversions.

If the operands are of more than one vector type, then a compile-time error will occur. Implicit conversions between vector types are not permitted.

Otherwise, if there is only a single vector type, and all other operands are scalar types, the scalar types are converted to the type of the vector element, and then widened into a new vector containing the same number of elements as the vector, by duplication of the scalar value across the width of the new vector. A compile-time error will occur if any scalar operand has greater rank than the type of the vector element. For this purpose, the rank order is defined as follows:

1. The rank of a floating-point type is greater than the rank of another floating-point type if the floating-point type can exactly represent all numeric values in the second floating-point type. (For this purpose, the encoding of the floating-point value is used, rather than the subset of the encoding usable by the device.)

2. The rank of any floating-point type is greater than the rank of any integer type.

3. The rank of an integer type is greater than the rank of an integer type with less precision.

4. The rank of an unsigned integer type is greater than the rank of a signed integer type with the same precision.

5. bool has a rank less than any other type.

6. The rank of an enumerated type is equal to the rank of the compatible integer type.
7. For all types $T_1$, $T_2$, and $T_3$, if $T_1$ has greater rank than $T_2$, and $T_2$ has greater rank than $T_3$, then $T_1$ has greater rank than $T_3$.

Otherwise, if all operands are scalar, the usual arithmetic conversions apply as defined by Section 6.3.1.8 of the C99 specification.

Following are a few examples of legal usual arithmetic conversions with vectors and vector and scalar operands:

```c
short a;
int4 b;
int4 c = b + a;
```

In this example, the variable $a$, which is of type `short`, is converted to an `int4` and the vector addition is then performed.

```c
int a;
float4 b;
float4 c = b + a;
```

In the preceding example, the variable $a$, which is of type `int`, is converted to a `float4` and the vector addition is then performed.

```c
float4 a;
float4 b;
float4 c = b + a;
```

In this example, no conversions need to be performed because $a$, $b$, and $c$ are all the same type.

Here are a few examples of illegal usual arithmetic conversions with vectors and vector and scalar operands:

```c
int a;
short4 b;
short4 c = b + a; // cannot convert & widen int to short4
```

```c
double a;
float4 b;
float4 c = b + a; // cannot convert & widen double to float4
```

```c
int4 a;
float4 b;
float4 c = b + a; // cannot cast between different vector types
```
Explicit Casts

Standard type casts for the built-in scalar data types defined in Table 4.1 will perform appropriate conversion (except void and half\(^4\)). In the next example, \(f\) stores 0x3F800000 and \(i\) stores 0x1, which is the floating-point value 1.0f in \(f\) converted to an integer value:

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

Explicit casts between vector types are not legal. The following examples will generate a compilation error:

```c
int4 i;
uint4 u = (uint4)i; // compile error

float4 f;
int4 i = (int4)f; // compile error

float4 f;
int8 i = (int8)f; // compile error
```

Scalar to vector conversions are performed by casting the scalar to the desired vector data type. Type casting will also perform the appropriate arithmetic conversion. Conversions to built-in integer vector types are performed with the round-toward-zero rounding mode. Conversions to built-in floating-point vector types are performed with the round-to-nearest rounding mode. When casting a bool to a vector integer data type, the vector components will be set to -1 (that is, all bits are set) if the bool value is true and 0 otherwise.

Here are some examples of explicit casts:

```c
float4 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 )
```

\(^4\) Unless the half-precision extension (cl_khr_fp16) is supported.
uchar4 vtrue = (uchar4)true;  // vtrue is a uchar4 vector with elements{0xFF, 0xFF, 0xFF, 0xFF}

**Explicit Conversions**

In the preceding sections we learned that implicit conversions and explicit casts do not allow conversions between vector types. However, there are many cases where we need to convert a vector type to another type. In addition, it may be necessary to specify the rounding mode that should be used to perform the conversion and whether the results of the conversion are to be saturated. This is useful for both scalar and vector data types.

Consider the following example:

```c
float x;
int i = (int)x;
```

In this example the value in `x` is truncated to an integer value and stored in `i`; that is, the cast performs round-toward-zero rounding when converting the floating-point value to an integer value.

Sometimes we need to round the floating-point value to the nearest integer. The following example shows how this is typically done:

```c
float x;
int i = (int)(x + 0.5f);
```

This works correctly for most values of `x` except when `x` is `0.5f - 1 ulp` or if `x` is a negative number. When `x` is `0.5f - 1 ulp`, `(int)(x + 0.5f)` returns `1`; that is, it rounds up instead of rounding down. When `x` is a negative number, `(int)(x + 0.5f)` rounds down instead of rounding up.

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <float.h>

int main(void)
{
    float a = 0.5f;
    float b = a - nextafterf(a, (float)-INFINITY); // a - 1 ulp
```

---

5 `ulp(x)` is the gap between two finite floating-point numbers. A detailed description of `ulp(x)` is given in Chapter 5 in the section “Math Functions,” subsection “Relative Error as ulps.”
printf("a = %8x, b = %8x\n",
     *(unsigned int *)&a, *(unsigned int *)&b);
printf("(int)(a + 0.5f) = %d \n", (int)(a + 0.5f));
printf("(int)(b + 0.5f) = %d \n", (int)(b + 0.5f));
}

The printed values are:

a = 3f000000, b = 3effffff  // where b = a – 1 ulp.
(int)(a + 0.5f) = 1,
(int)(b + 0.5f) = 1

We could fix these issues by adding appropriate checks to see what value
x is and then perform the correct conversion, but there is hardware to
do these conversions with rounding and saturation on most devices. It is
important from a performance perspective that OpenCL C allows devel-
opers to perform these conversions using the appropriate hardware ISA as
opposed to emulating in software. This is why OpenCL implements built-
in functions that perform conversions from one type to another with
options that select saturation and one of four rounding modes.

Explicit conversions may be performed using either of the following:

destType convert_destType<_sat><_roundingMode> (sourceType)
destType convert_destTypes<_sat><_roundingMode> (sourceTypen)

These provide a full set of type conversions for the following scalar types:
char, uchar, short, ushort, int, uint, long, ulong, float, double,6
half,7 and the built-in vector types derived therefrom. The operand and
result type must have the same number of elements. The operand and
result type may be the same type, in which case the conversion has no
effect on the type or value.

In the following example, convert_int4 converts a uchar4 vector u to
an int4 vector c:

uchar4 u;
int4 c = convert_int4(u);

In the next example, convert_int converts a float scalar f to an int
scalar i:

float f;
int i = convert_int(f);

6 Unless the double-precision extension (cl_khr_fp64) is supported.
7 Unless the half-precision extension (cl_khr_fp16) is supported.
The optional rounding mode modifier can be set to one of the values described in Table 4.7.

<table>
<thead>
<tr>
<th>Rounding Mode Modifier</th>
<th>Rounding Mode Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_rte</td>
<td>Round to nearest even.</td>
</tr>
<tr>
<td>_rtz</td>
<td>Round toward zero.</td>
</tr>
<tr>
<td>_rtp</td>
<td>Round toward positive infinity.</td>
</tr>
<tr>
<td>_rtn</td>
<td>Round toward negative infinity.</td>
</tr>
<tr>
<td>No modifier specified</td>
<td>Use the default rounding mode for this destination type: _rtz for conversion to integers or _rte for conversion to floating-point types.</td>
</tr>
</tbody>
</table>

The optional saturation modifier (_sat) can be used to specify that the results of the conversion must be saturated to the result type. When the conversion operand is either greater than the greatest representable destination value or less than the least representable destination value, it is said to be out of range. When converting between integer types, the resulting value for out-of-range inputs will be equal to the set of least significant bits in the source operand element that fits in the corresponding destination element. When converting from a floating-point type to an integer type, the behavior is implementation-defined.

Conversions to integer type may opt to convert using the optional saturated mode by appending the _sat modifier to the conversion function name. When in saturated mode, values that are outside the representable range clamp to the nearest representable value in the destination format. (NaN should be converted to 0.)

Conversions to a floating-point type conform to IEEE 754 rounding rules. The _sat modifier may not be used for conversions to floating-point formats.

Following are a few examples of using explicit conversion functions.

The next example shows a conversion of a float4 to a ushort4 with round-to-nearest rounding mode and saturation. Figure 4.2 describes the values in f and the result of conversion in c.

```cpp
float4  f = (float4)(-5.0f, 254.5f, 254.6f, 1.2e9f);
ushort4 c = convert_uchar4_sat_rte(f);
```
The next example describes the behavior of the saturation modifier when converting a signed value to an unsigned value or performing a down-conversion with integer types:

```c
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);
```

The following example illustrates conversion from a floating-point to an integer with saturation and rounding mode modifiers:

```c
float4 f;
// values implementation-defined for f > INT_MAX, f < INT_MAX, 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);
```

**Figure 4.2** Converting a `float4` to a `ushort4` with round-to-nearest rounding and saturation
The final conversion example given here shows conversions from an integer to a floating-point value with and without the optional rounding mode modifier:

```c
int4 i;

// convert ints to floats using the round-to-nearest 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);
```

### Reinterpreting Data as Another Type

Consider the case where you want to mask off the sign bit of a floating-point type. There are multiple ways to solve this in C—using pointer aliasing, unions, or `memcpy`. Of these, only `memcpy` is strictly correct in C99. Because OpenCL C does not support `memcpy`, we need a different method to perform this masking-off operation. The general capability we need is the ability to reinterpret bits in a data type as another data type. In the example where we want to mask off the sign bit of a floating-point type, we want to reinterpret these bits as an unsigned integer type and then mask off the sign bit. Other examples include using the result of a vector relational operator and extracting the exponent or mantissa bits of a floating-point type.

The `as_type` and `as_typeN` built-in functions allow you to reinterpret bits of a data type as another data type of the same size. The `as_type` is used for scalar data types (except `bool` and `void`) and `as_typeN` for vector data types. `double` and `half` are supported only if the appropriate extensions are supported by the implementation.

The following example describes how you would mask off the sign bit of a floating-point type using the `as_type` built-in function:

```c
float f;
uint  u;

u = as_uint(f);
f = as_float(u & ~(1 << 31));
```

If the operand and result type contain the same number of elements, the bits in the operand are returned directly without modification as the new
type. If the operand and result type contain a different number of elements, two cases arise:

- The operand is a 4-component vector and the result is a 3-component vector. In this case, the xyz components of the operand and the result will have the same bits. The w component of the result is considered to be undefined.

- For all other cases, the behavior is implementation-defined.

We next describe a few examples that show how to use as_type and as_type_n. The following example shows how to reinterpret an int as a float:

```c
uint u = 0x3f800000;
float f = as_float(u);
```

The variable u, which is declared as an unsigned integer, contains the value 0x3f800000. This represents the single-precision floating-point value 1.0. The variable f now contains the floating-point value 1.0.

In the next example, we reinterpret a float4 as an int4:

```c
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
int4 i = as_int4(f);
```

The variable i, defined to be of type int4, will have the following values in its xyzw components: 0x3f800000, 0x40000000, 0x40400000, 0x40800000.

The next example shows how we can perform the ternary selection operator (?:) for floating-point vector types using as_type:

```c
// Perform the operation f = f < g ? f : 0 for components of a
// vector
float4 f, g;
int4 is_less = f < g;

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

The following example describes cases where the operand and result have a different number of results, in which case the behavior of as_type and as_type_n is implementation-defined:
int i;
short2 j = as_short2(i); // Legal. Result is implementation-defined

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

float4 f;
float3 g = as_float3(f); // Legal. g.xyz will have same values as
// f.xyz. g.w is undefined

This example describes reinterpreting a 4-component vector as a 3-component vector:

float4 f;
float3 g = as_float3(f); // Legal. g.xyz will have same values as
// f.xyz. g.w is undefined

The next example shows invalid ways of using as_type and as_type n, which should result in compilation errors:

float4 f;
double4 g = as_double4(f); // Error. Result and operand have
// different sizes.

float3 f;
float4 g = as_float4(f); // Error. Result and operand have
// different sizes

**Vector Operators**

Table 4.8 describes the list of operators that can be used with vector data types or a combination of vector and scalar data types.

<table>
<thead>
<tr>
<th>Operator Category</th>
<th>Operator Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic operators</td>
<td>Add (+)</td>
</tr>
<tr>
<td></td>
<td>Subtract (-)</td>
</tr>
<tr>
<td></td>
<td>Multiply (*)</td>
</tr>
<tr>
<td></td>
<td>Divide (/)</td>
</tr>
<tr>
<td></td>
<td>Remainder (%)</td>
</tr>
</tbody>
</table>

*continues*
Table 4.8  Operators That Can Be Used with Vector Data Types (Continued)

<table>
<thead>
<tr>
<th>Operator Category</th>
<th>Operator Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational operators</td>
<td>Greater than (&gt;)</td>
</tr>
<tr>
<td></td>
<td>Less than (&lt;)</td>
</tr>
<tr>
<td></td>
<td>Greater than or equal (&gt;=)</td>
</tr>
<tr>
<td></td>
<td>Less than or equal (&lt;=)</td>
</tr>
<tr>
<td>Equality operators</td>
<td>Equal (==)</td>
</tr>
<tr>
<td></td>
<td>Not equal (!=)</td>
</tr>
<tr>
<td>Bitwise operators</td>
<td>And (&amp;)</td>
</tr>
<tr>
<td></td>
<td>Or (</td>
</tr>
<tr>
<td></td>
<td>Exclusive or (^), not (~)</td>
</tr>
<tr>
<td>Logical operators</td>
<td>And (&amp;&amp;)</td>
</tr>
<tr>
<td></td>
<td>Or (</td>
</tr>
<tr>
<td>Conditional operator</td>
<td>Ternary selection operator (? :)</td>
</tr>
<tr>
<td>Shift operators</td>
<td>Right shift (&gt;&gt;)</td>
</tr>
<tr>
<td></td>
<td>Left shift (&lt;&lt;)</td>
</tr>
<tr>
<td>Unary operators</td>
<td>Arithmetic (+ or -)</td>
</tr>
<tr>
<td></td>
<td>Post- and pre-increment (++)</td>
</tr>
<tr>
<td></td>
<td>Post- and pre-decrement (--)</td>
</tr>
<tr>
<td></td>
<td>sizeof, not (!)</td>
</tr>
<tr>
<td></td>
<td>Comma operator (,)</td>
</tr>
<tr>
<td></td>
<td>Address and indirection operators (&amp;, *)</td>
</tr>
<tr>
<td>Assignment operators</td>
<td>=, *=, /=, +=, -=, &lt;&lt;=, &gt;&gt;=, &amp;=, ^=,</td>
</tr>
</tbody>
</table>

The behavior of these operators for scalar data types is as described by the C99 specification. The following sections discuss how each operator works with operands that are vector data types or vector and scalar data types.

Arithmetic Operators

The arithmetic operators—add (+), subtract (−), multiply (∗), and divide (/)—operate on built-in integer and floating-point scalar and vector data types. The remainder operator (%) operates on built-in integer scalar and vector data types only. The following cases arise:
• The two operands are scalars. In this case, the operation is applied according to C99 rules.

• One operand is a scalar and the other is a vector. The scalar operand may be subject to the usual arithmetic conversion to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.

• The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

For integer types, a divide by zero or a division that results in a value that is outside the range will not cause an exception but will result in an unspecified value. Division by zero for floating-point types will result in ±infinity or NaN as prescribed by the IEEE 754 standard.

A few examples will illustrate how the arithmetic operators work when one operand is a scalar and the other a vector, or when both operands are vectors.

The first example in Figure 4.3 shows two vectors being added:

```
int4 v_iA = (int4)(7, -3, -2, 5);
int4 v_iB = (int4)(1, 2, 3, 4);
int4 v_iC = v_iA + v_iB;
```

```
[7, -3, -2, 5] + [1, 2, 3, 4] = [8, -1, 1, 9]
```

**Figure 4.3** Adding two vectors

The result of the addition stored in vector \( v_iC \) is \( (8, -1, 1, 9) \).

The next example in Figure 4.4 shows a multiplication operation where operands are a vector and a scalar. In this example, the scalar is just
widened to the size of the vector and the components of each vector are multiplied:

```c
float4 vf = (float4)(3.0f, -1.0f, 1.0f, -2.0f);
float4 result = vf * 2.5f;
```

![Figure 4.4](image)

**Figure 4.4** Multiplying a vector and a scalar with widening

The result of the multiplication stored in vector `result` is `(7.5f, -2.5f, 2.5f, -5.0f)`.

The next example in Figure 4.5 shows how we can multiply a vector and a scalar where the scalar is implicitly converted and widened:

```c
float4 vf = (float4)(3.0f, -1.0f, 1.0f, -2.0f);
float4 result = vf * 2;
```

![Figure 4.5](image)

**Figure 4.5** Multiplying a vector and a scalar with conversion and widening

The result of the multiplication stored in the vector `result` is `(6.0f, -2.0f, 2.0f, -4.0f)`.
Relational and Equality Operators

The relational operators—greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=)—and equality operators—equal (==) and not equal (!=)—operate on built-in integer and floating-point scalar and vector data types. The result is an integer scalar or vector type. The following cases arise:

- The two operands are scalars. In this case, the operation is applied according to C99 rules.
- One operand is a scalar and the other is a vector. The scalar operand may be subject to the usual arithmetic conversion to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

The result is a scalar signed integer of type int if both source operands are scalar and a vector signed integer type of the same size as the vector source operand. The result is of type char n if the source operands are char n or uchar n; short n if the source operands are short n, short n, or half n; int n if the source operands are int n, uint n, or float n; long n if the source operands are long n, ulong n, or doubl n.

For scalar types, these operators return 0 if the specified relation is false and 1 if the specified relation is true. For vector types, these operators return 0 if the specified relation is false and -1 (i.e., all bits set) if the specified relation is true. The relational operators always return 0 if one or both arguments are not a number (NaN). The equality operator equal (==) returns 0 if one or both arguments are not a number (NaN), and the equality operator not equal (!=) returns 1 (for scalar source operands) or -1 (for vector source operands) if one or both arguments are not a number (NaN).

Bitwise Operators

The bitwise operators—and (&), or (|), exclusive or (^), and not (~)—operate on built-in integer scalar and vector data types. The result is an integer scalar or vector type. The following cases arise:

- The two operands are scalars. In this case, the operation is applied according to C99 rules.
• One operand is a scalar and the other is a vector. The scalar operand may be subject to the usual arithmetic conversion to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.

• The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

Logical Operators

The logical operators—and (&&), or (||)—operate on built-in integer scalar and vector data types. The result is an integer scalar or vector type. The following cases arise:

• The two operands are scalars. In this case, the operation is applied according to C99 rules.

• One operand is a scalar and the other is a vector. The scalar operand may be subject to the usual arithmetic conversion to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.

• The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

If both source operands are scalar, the logical operator and (&&) will evaluate the right-hand operand only if the left-hand operand compares unequal to 0, and the logical operator or (||) will evaluate the right-hand operand only if the left-hand operand compares equal to 0. If one or both source operands are vector types, both operands are evaluated.

The result is a scalar signed integer of type int if both source operands are scalar and a vector signed integer type of the same size as the vector source operand. The result is of type charn if the source operands are charn or ucharn; shortn if the source operands are shortn or ushortn; intn if the source operands are intn or uintn; or longn if the source operands are longn or ulongn.

For scalar types, these operators return 0 if the specified relation is false and 1 if the specified relation is true. For vector types, these operators return 0 if the specified relation is false and ~1 (i.e., all bits set) if the specified relation is true.

The logical exclusive operator (^) is reserved for future use.
Conditional Operator

The ternary selection operator (?:) operates on three expressions (expr1 ? expr2 : expr3). This operator evaluates the first expression, expr1, which can be a scalar or vector type except the built-in floating-point types. If the result is a scalar value, the second expression, expr2, is evaluated if the result compares unequal to 0; otherwise the third expression, expr3, is evaluated. If the result is a vector value, then (expr1 ? expr2 : expr3) is applied component-wise and is equivalent to calling the built-in function select(expr3, expr2, expr1). The second and third expressions can be any type as long as their types match or if an implicit conversion can be applied to one of the expressions to make their types match, or if one is a vector and the other is a scalar, in which case the usual arithmetic conversion followed by widening is applied to the scalar to match the vector operand type. This resulting matching type is the type of the entire expression.

A few examples will show how the ternary selection operator works with scalar and vector types:

```c
int4 va, vb, vc, vd;
int a, b, c, d;
float4 vf;

vc = d ? va : vb;  // vc = va if d is true, = vb if d is false
vc = vd ? va : vb; // vc.x = vd.x ? va.x : vb.x
// vc.y = vd.y ? va.y : vb.y
// vc.z = vd.z ? va.z : vb.z
// vc.w = vd.w ? va.w : vb.w
vc = vd ? a : vb;  // a is widened to an int4 first
// vc.x = vd.x ? a : vb.x
// vc.y = vd.y ? a : vb.y
// vc.z = vd.z ? a : vb.z
// vc.w = vd.w ? a : vb.w
vc = vd ? va : vf; // error – vector types va & vf do not match
```

Shift Operators

The shift operators—right shift (>>) and left shift (<<)—operate on built-in integer scalar and vector data types. The result is an integer scalar or vector type. The rightmost operand must be a scalar if the first operand is a scalar. For example:
```c
uint a, b, c;
uint2 r0, r1;

c = a << b; // legal – both operands are scalars
r1 = a << r0; // illegal – first operand is a scalar and
  // therefore second operand (r0) must also be scalar.
c = b << r0; // illegal – first operand is a scalar and
  // therefore second operand (r0) must also be scalar.
```

The rightmost operand can be a vector or scalar if the first operand is a vector. For vector types, the operators are applied component-wise.

If operands are scalar, the result of $E_1 << E_2$ is $E_1$ left-shifted by $\log_2(N)$ least significant bits in $E_2$. The vacated bits are filled with zeros. If $E_2$ is negative or has a value that is greater than or equal to the width of $E_1$, the C99 specification states that the behavior is undefined. Most implementations typically return 0.

Consider the following example:

```c
char x = 1;
char y = -2;
x = x << y;
```

When compiled using a C compiler such as GCC on an Intel x86 processor, $(x << y)$ will return 0. However, with OpenCL C, $(x << y)$ is implemented as $(x << (y & 0x7))$, which returns $0x40$.

For vector types, $N$ is the number of bits that can represent the type of elements in a vector type for $E_1$ used to perform the left shift. For example:

```c
char2 x = (uchar2)(1, 2);
char  y = -9;
x = x << y;
```

Because components of vector $x$ are an unsigned char, the vector shift operation is performed as $(1 << (y & 0x7)), (2 << (y & 0x7))$.

Similarly, if operands are scalar, the result of $E_1 >> E_2$ is $E_1$ right-shifted by $\log_2(N)$ least significant bits in $E_2$. If $E_2$ is negative or has a value that is greater than or equal to the width of $E_1$, the C99 specification states that the behavior is undefined. For vector types, $N$ is the number of bits that can represent the type of elements in a vector type for $E_1$ used to perform the right shift. The vacated bits are filled with zeros if $E_1$ is an unsigned type or is a signed type but is not a negative value. If $E_1$ is a signed type and a negative value, the vacated bits are filled with ones.
Unary Operators

The arithmetic unary operators (+ and −) operate on built-in scalar and vector types.

The arithmetic post- and pre- increment (++) and decrement (--) operators operate on built-in scalar and vector data types except the built-in scalar and vector floating-point data types. These operators work component-wise on their operands and result in the same type they operated on.

The logical unary operator not (!) operates on built-in scalar and vector data types except the built-in scalar and vector floating-point data types. These operators work component-wise on their operands. The result is a scalar signed integer of type int if both source operands are scalar and a vector signed integer type of the same size as the vector source operand. The result is of type char[n] if the source operands are char[n] or uchar[n]; short[n] if the source operands are short[n] or ushort[n]; int[n] if the source operands are int[n] or uint[n]; or long[n] if the source operands are long[n] or ulong[n].

For scalar types, these operators return 0 if the specified relation is false and 1 if the specified relation is true. For vector types, these operators return 0 if the specified relation is false and −1 (i.e., all bits set) if the specified relation is true.

The comma operator (,) operates on expressions by returning the type and value of the rightmost expression in a comma-separated list of expressions. All expressions are evaluated, in order, from left to right. For example:

```c
// comma acts as a separator not an operator.
int a = 1, b = 2, c = 3, x;

// comma acts as an operator
x = a += 2, a + b;      // a = 3, x = 5
x = (a, b, c);          // x = 3
```

The `sizeof` operator yields the size (in bytes) of its operand. The result is an integer value. The result is 1 if the operand is of type char or uchar; 2 if the operand is of type short, ushort, or half; 4 if the operand is of type int, uint, or float; and 8 if the operand is of type long, ulong, or double. The result is number of components in vector * size of each scalar component if the operand is a vector type except for 3-component vectors, which return 4 * size of each scalar component. If the operand is an array type, the result is the total number of bytes in the array, and if the operand is a structure or union type, the
result is the total number of bytes in such an object, including any internal or trailing padding.

The behavior of applying the sizeof operator to the image2d_t, image3d_t, sampler_t, and event_t types is implementation-defined. For some implementations, sizeof(sampler_t) = 4 and on some implementation this may result in a compile-time error. For portability across OpenCL implementations, it is recommended not to use the sizeof operator for these types.

The unary operator (*) denotes indirection. If the operand points to an object, the result is an l-value designating the object. If the operand has type “pointer to type,” the result has type type. If an invalid value has been assigned to the pointer, the behavior of the indirection operator is undefined.

The unary operator (&) returns the address of its operand.

**Assignment Operator**

Assignments of values to variables names are done with the assignment operator (=), such as

lvalue = expression

The assignment operator stores the value of expression into lvalue. The following cases arise:

- The two operands are scalars. In this case, the operation is applied according to C99 rules.
- One operand is a scalar and the other is a vector. The scalar operand is *explicitly converted* to the element type used by the vector operand and is then widened to a vector that has the same number of elements as the vector operand. The operation is applied component-wise, resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is applied component-wise, resulting in the same size vector.

The following expressions are equivalent:

lvalue op= expression
lvalue = lvalue op expression

The lvalue and expression must satisfy the requirements for both operator op and assignment (=).
Qualifiers

OpenCL C supports four types of qualifiers: function qualifiers, address space qualifiers, access qualifiers, and type qualifiers.

Function Qualifiers

OpenCL C adds the kernel (or __kernel) function qualifier. This qualifier is used to specify that a function in the program source is a kernel function. The following example demonstrates the use of the kernel qualifier:

```c
kernel void
parallel_add(global float *a, global float *b, global float *result)
{
    ...
}
```

// The following example is an example of an illegal kernel declaration and will result in a compile-time error.
// The kernel function has a return type of int instead of void.
kernel int
parallel_add(global float *a, global float *b, global float *result)
{
    ...
}
```

The following rules apply to kernel functions:

- The return type must be void. If the return type is not void, it will result in a compilation error.
- The function can be executed on a device by enqueuing a command to execute the kernel from the host.
- The function behaves as a regular function if it is called from a kernel function. The only restriction is that a kernel function with variables declared inside the function with the local qualifier cannot be called from another kernel function.

The following example shows a kernel function calling another kernel function that has variables declared with the local qualifier. The behavior is implementation-defined so it is not portable across implementations and should therefore be avoided.

```c
kernel void
my_func_a(global float *src, global float *dst)
```
{  
    local float l_var[32];
    ...
}

kernel void  
my_func_b(global float * src, global float *dst)
{
    my_func_a(src, dst); // implementation-defined behavior
}

A better way to implement this example that is also portable is to pass the local variable as an argument to the kernel:

kernel void  
my_func_a(global float *src, global float *dst, local float *l_var)  
{
    ...
}

kernel void  
my_func_b(global float * src, global float *dst, local float *l_var)  
{
    my_func_a(src, dst, l_var);
}

### Kernel Attribute Qualifiers

The kernel qualifier can be used with the keyword `__attribute__` to declare the following additional information about the kernel:

- `__attribute__((work_group_size_hint(X, Y, Z)))` is a hint to the compiler and is intended to specify the work-group size that will most likely be used, that is, the value specified in the `local_work_size` argument to `clEnqueueNDRangeKernel`.

- `__attribute__((reqd_work_group_size(X, Y, Z)))` is intended to specify the work-group size that will be used, that is, the value specified in the `local_work_size` argument to `clEnqueueNDRangeKernel`. This provides an opportunity for the compiler to perform specific optimizations that depend on knowing what the work-group size is.

- `__attribute__((vec_type_hint(<type>)))` is a hint to the compiler on the computational width of the kernel, that is, the size
of the data type the kernel is operating on. This serves as a hint to an auto-vectorizing compiler. The default value of `<type>` is `int`, indicating that the kernel is scalar in nature and the auto-vectorizer can therefore vectorize the code across the SIMD lanes of the vector unit for multiple work-items.

**Address Space Qualifiers**

Work-items executing a kernel have access to four distinct memory regions. These memory regions can be specified as a type qualifier. The type qualifier can be `global` (or `__global`), `local` (or `__local`), `constant` (or `__constant`), or `private` (or `__private`).

If the type of an object is qualified by an address space name, the object is allocated in the specified address space. If the address space name is not specified, then the object is allocated in the generic address space. The generic address space name (for arguments to functions in a program, or local variables in a function) is `private`.

A few examples that describe how to specify address space names follow:

```c
// declares a pointer p in the private address space that points to
// a float object in address space global
global float *p;

// declares an array of integers in the private address space
int  f[4];

// for my_func_a function we have the following arguments:
//
//   src - declares a pointer in the private address space that
//       points to a float object in address space constant
//
//   v   - allocate in the private address space
//
int
my_func_a(constant float *src, int4 v)
{
   float temp; // temp is allocated in the private address space.
}
```

Arguments to a kernel function that are declared to be a pointer of a type must point to one of the following address spaces only: `global`, `local`, or `constant`. Not specifying an address space name for such arguments will result in a compilation error. This limitation does not apply to non-kernel functions in a program.
A few examples of legal and illegal use cases are shown here:

```c
kernel void my_func(int *p) // illegal because generic address space
    // name for p is private.

kernel void
my_func(private int *p) // illegal because memory pointed to by
    // p is allocated in private.

void
my_func(int *p) // generic address space name for p is private.
    // legal as my_func is not a kernel function

void
my_func(private int *p) // legal as my_func is not a kernel function
```

**Global Address Space**

This address space name is used to refer to memory objects (buffers and images) allocated from the global memory region. This memory region allows read/write access to all work-items in all work-groups executing a kernel. This address space is identified by the `global` qualifier.

A buffer object can be declared as a pointer to a scalar, vector, or user-defined struct. Some examples are:

```c
global float4 *color;    // an array of float4 elements

typedef struct {
    float3 a;
    int2   b[2];
} foo_t;

global foo_t *my_info;    // an array of foo_t elements
```

The global address qualifier should not be used for image types.

Pointers to the global address space are allowed as arguments to functions (including kernel functions) and variables declared inside functions. Variables declared inside a function cannot be allocated in the global address space.

A few examples of legal and illegal use cases are shown here:

```c
void
my_func(global float4 *vA, global float4 *vB)
{
    global float4 *p;   // legal
    global float4 a;    // illegal
}
Constant Address Space

This address space name is used to describe variables allocated in global memory that are accessed inside a kernel(s) as read-only variables. This memory region allows read-only access to all work-items in all work-groups executing a kernel. This address space is identified by the constant qualifier.

Image types cannot be allocated in the constant address space. The following example shows imgA allocated in the constant address space, which is illegal and will result in a compilation error:

```
kernel void
my_func(constant image2d_t imgA)
{
    ...
}
```

Pointers to the constant address space are allowed as arguments to functions (including kernel functions) and variables declared inside functions.

Variables in kernel function scope (i.e., the outermost scope of a kernel function) can be allocated in the constant address space. Variables in program scope (i.e., global variables in a program) can be allocated only in the constant address space. All such variables are required to be initialized, and the values used to initialize these variables must be compile-time constants. Writing to such a variable will result in a compile-time error.

Also, storage for all string literals declared in a program will be in the constant address space.

A few examples of legal and illegal use cases follow:

```
// legal - program scope variables can be allocated only
// in the constant address space
constant float wtsA[] = { 0, 1, 2, ... }; // program scope

// illegal - program scope variables can be allocated only
// in the constant address space
global float wtsB[] = { 0, 1, 2, ... };

kernel void
my_func(constant float4 *vA, constant float4 *vB)
{
    constant float4 *p = vA; // legal
    constant float a;         // illegal - not initialized
    constant float b = 2.0f;  // legal - initialized with a compile-time constant
```
p[0] = (float4)(1.0f);   // illegal – p cannot be modified

// the string "opencl version" is allocated in the
// constant address space
char *c = "opencl version";
}

**Note** The number of variables declared in the constant address space
that can be used by a kernel is limited to CL_DEVICE_MAX_CONSTANT_ARGS. OpenCL 1.1 describes that the minimum value
all implementations must support is eight. So up to eight variables
declared in the constant address space can be used by a kernel and
are guaranteed to work portably across all implementations. The
size of these eight constant arguments is given by CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE and is set to 64KB. It is therefore
possible that multiple constant declarations (especially those
defined in the program scope) can be merged into one constant
buffer as long as their total size is less than CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE. This aggregation of multiple variables
declared to be in the constant address space is not a required
behavior and so may not be implemented by all OpenCL imple-
mentations. For portable code, the developer should assume that
these variables do not get aggregated into a single constant buffer.

**Local Address Space**

This address space name is used to describe variables that need to be allo-
cated in local memory and are shared by all work-items of a work-group
but not across work-groups executing a kernel. This memory region allows
read/write access to all work-items in a work-group. This address space is
identified by the `local` qualifier.

A good analogy for local memory is a user-managed cache. Local memory
can significantly improve performance if a work-item or multiple work-
items in a work-group are reading from the same location in global mem-
ory. For example, when applying a Gaussian filter to an image, multiple
work-items read overlapping regions of the image. The overlap region size
is determined by the width of the filter. Instead of reading multiple times
from global memory (which is an order of magnitude slower), it is prefera-
ble to read the required data from global memory once into local memory
and then have the work-items read multiple times from local memory.

Pointers to the local address space are allowed as arguments to functions
(including kernel functions) and variables declared inside functions.
Variables declared inside a kernel function can be allocated in the local address space but with a few restrictions:

- These variable declarations must occur at kernel function scope.
- These variables cannot be initialized.

Note that variables in the local address space that are passed as pointer arguments to or declared inside a kernel function exist only for the lifetime of the work-group executing the kernel.

A few examples of legal and illegal use cases are shown here:

```c
kernel void my_func(global float4 *vA, local float4 *l)
{
    local float4 *p;   // legal
    local float4 a;    // legal
    a = 1;
    local float4 b = (float4)(0); // illegal – b cannot be initialized

    if (...)
    {
        local float c; // illegal – must be allocated at
                        // kernel function scope
        ...
    }
}
```

**Private Address Space**

This address space name is used to describe variables that are private to a work-item and cannot be shared between work-items in a work-group or across work-groups. This address space is identified by the `private` qualifier.

Variables inside a kernel function not declared with an address space qualifier, all variables declared inside non-kernel functions, and all function arguments are in the `private` address space.

**Casting between Address Spaces**

A pointer in an address space can be assigned to another pointer only in the same address space. Casting a pointer in one address space to a pointer in a different address space is illegal. For example:
kernel void
my_func(global float4 *particles)
{
    // legal – particle_ptr & particles are in the
    // same address space
    global float *particle_ptr = (global float *)particles;

    // illegal – private_ptr and particle_ptr are in different
    // address spaces
    float *private_ptr = (float *)particle_ptr;
}

Access Qualifiers

The access qualifiers can be specified with arguments that are an image
type. These qualifiers specify whether the image is a read-only (read_only
or __read_only) or write-only (write_only or __write_only) image. This is because of a limitation of current GPUs that do not allow
reading and writing to the same image in a kernel. The reason for this is
that image reads are cached in a texture cache, but writes to an image do
not update the texture cache.

In the following example imageA is a read-only 2D image object and
imageB is a write-only 2D image object:

kernel void
my_func(read_only image2d_t imageA, write_only image2d_t imageB)
{
    ...
}

Images declared with the read_only qualifier can be used with the
built-in functions that read from an image. However, these images cannot
be used with built-in functions that write to an image. Similarly, images
declared with the write_only qualifier can be used only to write to an
image and cannot be used to read from an image. The following examples
demonstrate this:

kernel void
my_func(read_only image2d_t imageA,
        write_only image2d_t imageB,
        sampler_t sampler)
{
    float4 clr;
    float2 coords;

    clr = read_imagef(imageA, sampler, coords); // legal
    clr = read_imagef(imageB, sampler, coords); // illegal
imageA is declared to be a read_only image so it cannot be passed as an argument to write_imagef. Similarly, imageB is declared to be a write_only image so it cannot be passed as an argument to read_imagef.

The read-write qualifier (read_write or __read_write) is reserved. Using this qualifier will result in a compile-time error.

Type Qualifiers

The type qualifiers const, restrict, and volatile as defined by the C99 specification are supported. These qualifiers cannot be used with the image2d_t and image3d_t type. Types other than pointer types cannot use the restrict qualifier.

Keywords

The following names are reserved for use as keywords in OpenCL C and cannot be used otherwise:

- Names already reserved as keywords by C99
- OpenCL C data types (defined in Tables 4.1, 4.2, and 4.6)
- Address space qualifiers: __global, global, __local, local, __constant, constant, __private, and private
- Function qualifiers: __kernel and kernel
- Access qualifiers: __read_only, read_only, __write_only, write_only, __read_write, and read_write

Preprocessor Directives and Macros

The preprocessing directives defined by the C99 specification are supported. These include

# non-directive
#if
#ifndef
The defined operator is also included.

The following example demonstrates the use of \#if, \#elif, \#else, and \#endif preprocessor macros. In this example, we use the preprocessor macros to determine which arithmetic operation to apply in the kernel. The kernel source is described here:

```c
#define OP_ADD          1
#define OP_SUBTRACT     2
#define OP_MULTIPLY     3
#define OP_DIVIDE       4

kernel void foo(global float *dst, global float *srcA, global float *srcB)
{
    size_t id = get_global_id(0);
    #if OP_TYPE == OP_ADD
        dst[id] = srcA[id] + srcB[id];
    #elif OP_TYPE == OP_SUBTRACT
        dst[id] = srcA[id] - srcB[id];
    #elif OP_TYPE == OP_MULTIPLY
        dst[id] = srcA[id] * srcB[id];
    #elif OP_TYPE == OP_DIVIDE
        dst[id] = srcA[id] / srcB[id];
    #else
        dst[id] = NAN;
    #endif
}
```

To build the program executable with the appropriate value for OP_TYPE, the application calls \texttt{clBuildProgram} as follows:

```c
// build program so that kernel foo does an add operation
err = clBuildProgram(program, 0, NULL,
                      "-DOP_TYPE=1", NULL, NULL);
```
Pragma Directives

The \#pragma directive is described as

\#pragma pp-tokensopt new-line

A \#pragma directive where the preprocessing token OPENCL (used instead of STDC) does not immediately follow \#pragma in the directive (prior to any macro replacement) causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a nonconforming manner. Any such \#pragma that is not recognized by the implementation is ignored. If the preprocessing token OPENCL does immediately follow \#pragma in the directive (prior to any macro replacement), then no macro replacement is performed on the directive.

The following standard pragma directives are available.

Floating-Point Pragma

The FP_CONTRACT floating-point pragma can be used to allow (if the state is on) or disallow (if the state is off) the implementation to contract expressions. The FP_CONTRACT pragma definition is

\#pragma OPENCL FP_CONTRACT on-off-switch
  on-off-switch: one of ON OFF DEFAULT

A detailed description of \#pragma OPENCL FP_CONTRACT is found in Chapter 5 in the section “Floating-Point Pragmas.”

Compiler Directives for Optional Extensions

The \#pragma OPENCL EXTENSION directive controls the behavior of the OpenCL compiler with respect to language extensions. The \#pragma OPENCL EXTENSION directive is defined as follows, where extension_name is the name of the extension:

\#pragma OPENCL EXTENSION extension_name: behavior
\#pragma OPENCL EXTENSION all : behavior
  behavior: enable or disable

The extension_name will have names of the form cl_khr_<name> for an extension (such as cl_khr_fp64) approved by the OpenCL working group and will have names of the form cl_<vendor_name>_<name> for vendor extensions. The token all means that the behavior applies to all extensions supported by the compiler. The behavior can be set to one of the values given in Table 4.9.
The \texttt{#pragma OPENCL EXTENSION} directive is a simple, low-level mechanism to set the behavior for each language extension. It does not define policies such as which combinations are appropriate; these are defined elsewhere. The order of directives matters in setting the behavior for each extension. Directives that occur later override those seen earlier. The all variant sets the behavior for all extensions, overriding all previously issued extension directives, but only if the behavior is set to disable.

An extension needs to be enabled before any language feature (such as preprocessor macros, data types, or built-in functions) of this extension is used in the OpenCL program source. The following example shows how to enable the double-precision floating-point extension:

\begin{verbatim}
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
double x = 2.0;
\end{verbatim}

If this extension is not supported, then a compilation error will be reported for \texttt{double x = 2.0}. If this extension is supported, this enables the use of double-precision floating-point extensions in the program source following this directive.

Similarly, the \texttt{cl_khr_3d_image_writes} extension adds new built-in functions that support writing to a 3D image:

\begin{verbatim}
#pragma OPENCL EXTENSION cl_khr_3d_image_writes : enable
kernel void my_func(write_only image3d_t img, ...)
{
    float4 coord, clr;
    ...
    write_imagef(img, coord, clr);
}
\end{verbatim}

\begin{table}[h]
\centering
\begin{tabular}{|c|p{0.7\textwidth}|}
\hline
\textbf{Behavior} & \textbf{Description} \\
\hline
\texttt{enable} & Enable the extension \texttt{extension\_name}. Report an error on the \texttt{#pragma OpenCL EXTENSION} if the \texttt{extension\_name} is not supported, or if all is specified. \\
\hline
\texttt{disable} & Behave (including issuing errors and warnings) as if the extension \texttt{extension\_name} is not part of the language definition. If all is specified, then behavior must revert back to that of the nonextended core version of the language being compiled to. Warn on the \texttt{#pragma OPENCL EXTENSION} if the \texttt{extension\_name} is not supported. \\
\hline
\end{tabular}
\caption{Optional Extension Behavior Description}
\end{table}
The built-in functions such as `write_imagef` with `image3d_t` in the preceding example can be called only if this extension is enabled; otherwise a compilation error will occur.

The initial state of the compiler is as if the following directive were issued, telling the compiler that all error and warning reporting must be done according to this specification, ignoring any extensions:

```c
#pragma OPENCL EXTENSION all : disable
```

Every extension that affects the OpenCL language semantics or syntax or adds built-in functions to the language must also create a preprocessor `#define` that matches the extension name string. This `#define` would be available in the language if and only if the extension is supported on a given implementation. For example, an extension that adds the extension string `cl_khr_fp64` should also add a preprocessor `#define` called `cl_khr_fp64`. A kernel can now use this preprocessor `#define` to do something like this:

```c
#ifdef cl_khr_fp64
   // do something using this extension
#else
   // do something else or #error
#endif
```

**Macros**

The following predefined macro names are available:

- `__FILE__` is the presumed name of the current source file (a character string literal).
- `__LINE__` is the presumed line number (within the current source file) of the current source line (an integer constant).
- `CL_VERSION_1_0` substitutes the integer 100, reflecting the OpenCL 1.0 version.
- `CL_VERSION_1_1` substitutes the integer 110, reflecting the OpenCL 1.1 version.
- `__OPENCL_VERSION__` substitutes an integer reflecting the version number of the OpenCL supported by the OpenCL device. This reflects both the language version supported and the device capabilities as given in Table 4.3 of the OpenCL 1.1 specification. The version of OpenCL described in this book will have `__OPENCL_VERSION__` substitute the integer 110.
• __ENDIAN_LITTLE__ is used to determine if the OpenCL device is a little endian architecture or a big endian architecture (an integer constant of 1 if the device is little endian and is undefined otherwise).

• __kernel_exec(X, typen) (and kernel_exec(X, typen)) is defined as
  __kernel __attribute__((work_group_size_hint(X, 1, 1))) \
  __attribute__((vec_type_hint(typen))).

• __IMAGE_SUPPORT__ is used to determine if the OpenCL device supports images. This is an integer constant of 1 if images are supported and is undefined otherwise.

• __FAST_RELAXED_MATH__ is used to determine if the –cl-fast-relaxed-math optimization option is specified in build options given to clBuildProgram. This is an integer constant of 1 if the –cl-fast-relaxed-math build option is specified and is undefined otherwise.

The macro names defined by the C99 specification but not currently supported by OpenCL are reserved for future use.

Restrictions

OpenCL C implements the following restrictions. Some of these restrictions have already been described in this chapter but are also included here to provide a single place where the language restrictions are described.

• Kernel functions have the following restrictions:
  • Arguments to kernel functions that are pointers must use the global, constant, or local qualifier.
  • An argument to a kernel function cannot be declared as a pointer to a pointer(s).
  • Arguments to kernel functions cannot be declared with the following built-in types: bool, half, size_t, ptrdiff_t, intptr_t, uintptr_t, or event_t.
  • The return type for a kernel function must be void.
  • Arguments to kernel functions that are declared to be a struct cannot pass OpenCL objects (such as buffers, images) as elements of the struct.
• Bit field struct members are not supported.
• Variable-length arrays and structures with flexible (or unsized) arrays are not supported.
• Variadic macros and functions are not supported.
• The `extern`, `static`, `auto`, and `register` storage class specifiers are not supported.
• Predefined identifiers such as `__func__` are not supported.
• Recursion is not supported.
• The library functions defined in the C99 standard headers—`assert.h`, `ctype.h`, `complex.h`, `errno.h`, `fenv.h`, `float.h`, `inttypes.h`, `limits.h`, `locale.h`, `setjmp.h`, `signal.h`, `stdarg.h`, `stdio.h`, `stdlib.h`, `string.h`, `tgmath.h`, `time.h`, `wchar.h`, and `wctype.h`—are not available and cannot be included by a program.
• The image types `image2d_t` and `image3d_t` can be specified only as the types of a function argument. They cannot be declared as local variables inside a function or as the return types of a function. An image function argument cannot be modified. An image type cannot be used with the `private`, `local`, and `constant` address space qualifiers. An image type cannot be used to declare a variable, a structure or union field, an array of images, a pointer to an image, or the return type of a function.
• The sampler type `sampler_t` can be specified only as the type of a function argument or a variable declared in the program scope or the outermost scope of a kernel function. The behavior of a sampler variable declared in a non-outermost scope of a kernel function is implementation-defined. A sampler argument or a variable cannot be modified. The sampler type cannot be used to declare a structure or union field, an array of samplers, a pointer to a sampler, or the return type of a function. The sampler type cannot be used with the `local` and `global` address space qualifiers.
• The event type `event_t` can be used as the type of a function argument except for kernel functions or a variable declared inside a function. The event type can be used to declare an array of events. The event type can be used to declare a pointer to an event, for example, `event_t *event_ptr`. An event argument or variable cannot be modified. The event type cannot be used to declare a structure or
union field, or for variables declared in the program scope. The event type cannot be used with the local, constant, and global address space qualifiers.

- The behavior of irreducible control flow in a kernel is implementation-defined. Irreducible control flow is typically encountered in code that uses `goto`. An example of irreducible control flow is a `goto` jumping inside a nested loop or a Duff’s device.
Symbols
-- (pre-increment) unary operator, 131
- (subtract) operator, 124–126
?: (ternary selection) operator, 129
- or -- (unary) operators, 131
| or || (or) operators, 127–128
+ (addition) operator, 124–126
+ or ++ (post-increment) unary operator, 131
!= (not equal) operator, 127
== (equal) operator, 127
% (remainder) operator, 124–126
& or && (and) operators, 127–128
* (multiply) operator, 124–126
^ (exclusive or) operator, 127–128
^^ (exclusive) operator, 128
~ (not) operator, 127–128
> (greater than) operator, 127
>= (greater than or equal) operator, 127
>> (right shift) operator, 129–130

Numbers
0 value, 64–65, 68
2D composition, in DFT, 457–458
64-bit integers, embedded profile, 385–386
754 formats, IEEE floating-point arithmetic, 34

A
accelerator devices
defined, 69
tiled and packetized sparse matrix design, 523, 534
access qualifiers
as keywords in OpenCL C, 141
overview of, 140–141
reference guide, 576
add (+) arithmetic operator, 124–126
address space qualifiers
casting between address spaces, 139–140
constant, 137–138
global, 136
as keywords in OpenCL C, 141
local, 138–139
overview of, 135–136
private, 139
reference guide, 554
supported, 99
addressing mode, sampler objects, 282, 292–295
ALL_BUILD project, Visual Studio, 43
AltiVec Technology Programming Interface Manual, 111–113
AMD
generating project in Linux, 40–41
generating project in Windows, 40–41
storing binaries in own format, 233
and (& or &&) operators, 127–128
Apple
initializing contexts for OpenGL interoperability, 338
querying number of platforms, 64
storing binaries in own format, 233
application data types, 103–104
ARB_cl_event extension, OpenGL, 349–350
architecture diagram, OpenCL device, 577
arguments
context, 85
device, 68
enqueuing commands, 313
guassian_kernel(), 296–297
kernel function restrictions, 146
reference guide for kernel, 548
setting kernel, 55–57, 237–240
arithmetic operators
  overview of, 124–126
  post- and pre-increment (++ and --)
    unary, 131
  symbols, 123
  unary (+ and -), 131
arrays
  parallelizing Dijkstra's algorithm, 412–414
  representing sparse matrix with
    binary data, 516
as_type(), 121–123
as_typeN(), 121–123
ASCII File, representing sparse matrix, 516–517
assignment (=) operator, 124, 132
async copy and prefetch functions, 191–195, 570
ATI Stream SDK
  generating project in Linux and
    Eclipse, 44–45
  generating project in Visual Studio, 42–44
  generating project in Windows, 40
  querying and selecting platform, 65–66
  querying context for devices, 89
  querying devices, 70
atomic built-in functions
  embedded profile options, 387
  overview of, 195–198
  reference guide, 568–569
_attribute_ keyword, kernel qualifier, 133–134
attributes, specifying type, 555
automatic load balancing, 20

B
barrier synchronization function, 190–191
batches
  executing cloth simulation on GPU, 433–441
  SpMV implementation, 518
behavior description, optional exten-
  sion, 144
bilinear sampling object, optical flow, 476
binaries, program
  creating, 235–236
  HelloBinaryWorld example, 229–230
  HelloWorld.cl (NVIDIA) example, 233–236
  overview of, 227–229
  querying and storing, 230–232
binary data arrays, sparse matrix, 516
bit field numbers, 147
bitwise operators, 124, 127–128
blocking enqueue calls, and callbacks, 327
blocking_read, executing kernel, 56
bool, rank order of, 113
border color, built-in functions, 209–210
bracket() operator, C++ Wrapper API, 370–371
buffers and sub-buffers
  computing Dijkstra's algorithm, 415
  copying, 274–276
  copying from image to, 299, 303–304
  creating, 249–256
  creating from OpenGL, 339–343
  creating kernel and memory objects, 377–378
  direct translation of matrix multipli-
    cation into OpenCL, 502
  executing Vector Add kernel, 377–378, 381
  mapping, 276–279
  in memory model, 21
  Ocean application, 451
  OpenCL/OpenGL sharing APIs, 446–448, 578
  overview of, 247–248
  querying, 257–259
  reading and writing, 259–274
  reference guide, 544–545
building program objects
  reference guide, 546–547
  using clBuildProgram(). see
    clBuildProgram()
built-in data types
  other, 108–109
  reference guide, 550–552
  scalar, 99–101
  vector, 102–103
built-in functions
  async copy and prefetch, 191–195
atomic, 195–198, 387, 568–569
border color, 209–210
common, 172–175, 559
floating-point constant, 162–163
floating-point pragma, 162
geometric, 175–177, 563–564
image read and write, 201–206, 572–573
integer, 168–172, 557–558
math, 153–161, 560–563
miscellaneous vector, 199–200, 571
overview of, 149
querying image information, 214–215
relational, 175, 178–181, 564–567
relative error as ulps, 163–168
samplers, 206–209
synchronization, 190–191
vector data load and store, 181–189
work-item, 150–152, 557
writing to image, 210–213

Bullet Physics SDK. see cloth simulation in Bullet Physics SDK
bytes, and vector data types, 102

c
C++ Wrapper API
defined, 369
exceptions, 371–374
Ocean application overview, 451
overview of, 369–371
C++ Wrapper API, Vector Add example
choosing device and creating command-queue, 375–377
choosing platform and creating context, 375
creating and building program object, 377
creating kernel and memory objects, 377–378
executing Vector Add kernel, 378–382
structure of OpenCL setup, 374–375
C99 language
OpenCL C derived from, 32–33, 97
OpenCL C features added to, 99
callbacks
creating OpenCL contexts, 85
event objects. see clSetEventCallback()
cl_khr_gl_event extension, 342, 348
cl_khr_gl_sharing extension, 336–337, 342
cl_map_flags, clEnqueueMapBuffer(), 276–277
cl_mem object, creating images, 284
CL_MEM_COPY_FROM_HOST_PTR, 377–378
cl_mem_flags, clCreateBuffer(), 249–250
CL_MEM_READ_ONLY | CL_MEM_COPY_FROM_HOST_PTR memory type, 55
CL_MEM_READ_WRITE, 308
CL_MEM_USE_HOST_PTR, 377–378
cl_platform error values, C++ Wrapper API, 371
cl_platform, 370–371
CL_PROFILING_COMMAND_END, 502
CL_PROFILING_COMMAND_START, 502
CL_QUEUE_PROFILING_ENABLE flag, 328
CL_QUEUE_PROFILING_ENABLE property, 502
CL_QUEUED value, command-queue, 311
CL_RUNNING value, command-queue, 311
CL_SUBMITTED value, command-queue, 311
CL_SUCCESS return value, clBuildProgram(), 220
_CL_USER_OVERRIDE_ERROR_STRINGS preprocessor macro, 372
classes, C++ Wrapper API hierarchy, 369–370
c1Barrier(), 313–316
c1Buffer(), 54
c1::Buffer(), 377–378, 381
c1::BuildProgram()
build options, 546–547
building program object, 219–220, 222
creating program from binary, 234–236
floating-point options, 224
miscellaneous options, 226–227
optimization options, 225–226
preprocessor build options, 223–224
querying program objects, 237
reference guide, 546
cl::commandQueue::enqueueUnmapObj(), 379, 382
c1::Context(), 375
c1::Context::getInfo(), 376
c1CreateBuffer()
creating buffers and sub-buffers, 249–251
creating memory objects, 54–55
direct translation of matrix multiplication into OpenCL, 502
reference guide, 544
setting kernel arguments, 239
c1CreateCommandQueue(), 51–52, 543
c1CreateContext(), 84–87, 541
c1CreateContextFromType()
creating contexts, 84–85
querying context for associated devices, 88
reference guide, 541
c1CreateEventFromGLsyncKHR() explicit synchronization, 349
reference guide, 579
synchronization between OpenCL/OpenGL, 350–351
c1CreateFromD3D10BufferKHR(), 580
c1CreateFromD3D10Texture2DKHR(), 580
c1CreateFromD3D10Texture3DKHR(), 580
c1CreateFromGL*(), 335, 448
c1CreateFromGLBuffer(), 339–343, 578
c1CreateFromGLRenderbuffer() creating memory objects from OpenGL, 341
reference guide, 578
sharing with OpenCL, 346–347
c1CreateFromGLTexture2D(), 341, 578
c1CreateFromGLTexture3D(), 341, 578
c1CreateImage2D()
creating 2D image from file, 284–285
creating image objects, 283–284
reference guide, 573–574
c1CreateImage3D(), 283–284, 574
c1CreateKernel()
creating kernel objects, 237–238
reference guide, 547
setting kernel arguments, 239–240
c1CreateKernelsInProgram(), 240–241, 547
clCreateProgram(), 221
clCreateProgramWithBinary()
creating programs from binaries, 228–229
HelloBinaryWorld example, 229–230
reference guide, 546
clCreateProgramWithSource()
creating and building program object, 52–53
creating program object from source, 218–219, 222
reference guide, 546
clCreateSampler(), 292–294, 576
c1CreateSubBuffer(), 253–256, 544
clCreateUserEvent()
generating events on host, 321–322
how to use, 323–324
reference guide, 549
clEnqueueAcquireD3D10ObjectsKHR(), 580
c1EnqueueAcquireGLOObjects()creating OpenGL buffers, 341–342
explicit synchronization, 349
implicit synchronization, 348–349
reference guide, 579
clEnqueueBarrier()
function of, 316–317
ordering constraints between commands, 313
reference guide, 549
clEnqueueCopyBuffer(), 275–276, 545
c1EnqueueCopyBufferToImage()
copying from buffer to image, 303–305
defined, 299
reference guide, 574
clEnqueueCopyImage()
copy image objects, 302–303
defined, 299
reference guide, 575
clEnqueueCopyImageToBuffer()
copying from image to buffer, 303–304
defined, 299
reference guide, 574
clEnqueueMapBuffer()mapping buffers and sub-buffers, 276–278
clEnqueueMarker()
defining synchronization points, 314
function of, 315–317
reference guide, 549
clEnqueueNativeKernel(), 548
c1EnqueueNDRangeKernel()
events and command-queues, 312
executing kernel, 56–57
reference guide, 548
work-items, 150
clEnqueueReadBuffer()
reading buffers, 260–261, 268–269
reading results back from kernel, 48, 56–57
reference guide, 544
clEnqueueReadBufferRect(), 269–272, 544
clEnqueueReadImage()
defined, 299–301
mapping image results to host memory pointer, 307–308
reference guide, 575
clEnqueueReleaseD3D10ObjectsKHR(), 580
c1EnqueueReleaseGLOObjects()
imPLICIT synchronization, 348–349
reference guide, 579
releasing objects acquired by OpenGL, 341–342
synchronization between OpenCL/OpenGL, 351
clEnqueueTask(), 150, 548
clEnqueueUnmapMapImage()
buffer mapping no longer required, 277–278
moving data to and from buffer, 278–279
reference guide, 545
clEnqueueUnmapMemObject()
releasing image data, 308

Index 585
clEnqueueWaitForEvents(), 314–317, 549
clEnqueueWriteBuffer() reference guide, 544
writing buffers, 259–260, 267
clEnqueueWriteBufferRect(), 272–273, 544–545
clEnqueueWriteImage() defined, 299
reference guide, 575
writing images from host to device memory, 301–302
cles_khr_int64 extension string, embedded profile, 385–386
c1Finish()
creating OpenCL buffers from OpenGL buffers, 342–343
OpenCL/OpenGL synchronization with, 348
OpenCL/OpenGL synchronization without, 351
preprocessor error macro for, 327
reference guide, 549
c1Flush()
preprocessor error macro for, 327
reference guide, 549
using callbacks with events, 327
c1.get_platforms(), PyOpenCL, 493
c1GetCommandQueueInfo(), 543
c1GetContextInfo() HelloWorld example, 50–51
querying context properties, 86–87
querying list of associated devices, 88
reference guide, 542
c1GetDeviceIDs()
convolution signal example, 91
querying devices, 68–69
translation of matrix multiplication into OpenCL, 502
c1GetDeviceIDsFromD3D10KHR(), 542
c1GetDeviceInfo()
querying for OpenGL sharing extension, 336–337
reference guide, 542–543, 579
steps in OpenCL usage, 83
c1GetEventInfo(), 319–320, 549
c1GetEventProfilingInfo() direct translation of matrix multiplication, 502
errors, 329–330
extracting timing data, 328
placing profiling functions inside callbacks, 332
profiling information and return types, 329
reference guide, 549
c1GetGLContextInfoKHR(), 579
c1GetGLObjectInfo(), 347–348, 578
c1GetGLTextureInfo(), 578
c1GetImageInfo(), 286
c1GetKernelInfo(), 242–243, 548
c1GetKernelWorkGroupInfo(), 243–244, 548
c1GetMemObjectInfo() querying buffers and sub-buffers, 257–259
querying image object, 286
reference guide, 545
c1GetPlatformIDs() querying context for associated devices, 88
querying platforms, 63–64
reference guide, 542
c1GetPlatformInfo() embedded profile, 384
querying and selecting platform, 65–67
reference guide, 542
c1GetProgramBuildInfo() creating and building program object, 52–53
detecting build error, 220–221, 222
direct translation of matrix multiplication, 502
reference guide, 547
c1GetProgramInfo() getting program binary back from built program, 227–228
reference guide, 547
c1GetSamplerInfo(), 294–295, 576
cloth simulation in Bullet Physics SDK
adding OpenGL interoperation,
446–448
executing on CPU, 431–432
executing on GPU, 432–438
introduction to, 425–428
optimizing for SIMD computation
and local memory, 441–446
overview of, 425
of soft body, 429–431
two-layered batching, 438–441
cl::Program(), 377
clReleaseCommandQueue(), 543
cReleaseContext(), 89, 542
clReleaseEvent(), 318–319, 549
clReleaseKernel(), 244–245, 548
clReleaseMemObject()
reference guide, 545
release buffer object, 339
release image object, 284
cReleaseProgram(), 236, 546
clReleaseSampler(), 294, 576
cRetainCommandQueue(), 543
cRetainContext(), 89, 541
cRetainEvent(), 318, 549
cRetainKernel(), 245, 548
cRetainMemObject(), 339, 545
cRetainProgram(), 236–237, 546
cRetainSampler(), 576
cSetEventCallback()
events impacting execution on host,
325–326
placing profiling functions inside
callbacks, 331–332
reference guide, 549
clSetKernelArg()
creating buffers and sub-buffers, 250,
255
executing kernel, 55–56
executing Vector Add kernel, 378
matrix multiplication using local
memory, 509–511
reference guide, 548
sampler declaration fields, 577
setting kernel arguments, 56, 237–240
thread safety and, 241–242
clSetMemObjectDestructor-
Callback(), 545
clSetUserEventStatus()
generating events on host, 322
how to use, 323–324
reference guide, 549
cUnloadComposer(), 237, 547
cWaitForEvents(), 323–324, 549
CMake tool
generating project in Linux and
Eclipse, 44–45
generating project in Visual Studio,
42–44
installing as cross-platform build tool,
40–41
Mac OS X and Code::Blocks, 40–41
cmake-gui, 42–44
Code::Blocks, 41–42
color, cloth simulation
executing on GPU, 433–438
in two-layered batching, 438–441
color images. see image histograms
coma operator (.), 131
command-queue
acquiring OpenGL objects, 341–342
as core of OpenCL, 309–310
creating, 50–52
creating after selecting set of devices,
377
creating in PyOpenCL, 493
defining consistency of memory
objects on, 24
direct translation of matrix multipli-
cation into OpenCL, 502
events and, 311–317
executing kernel, 56–57
in execution model, 18–21
execution of Vector Add kernel, 378,
380
OpenCL runtime reference guide, 543
runtime API setting up, 31–32
transferring image objects to,
299–300
common functions, 172–175
compiler
directives for optional extensions, 143–145
unloading OpenCL, 547
component selection syntax, vectors, 106–107
components, vector data type, 106–108
compute device, platform model, 12
compute units, platform model, 12
concurrency, 7–8
   exploiting in command-queues, 310
   parallel algorithm limitations, 28–29
conditional operator, 124, 129
costArray, Dijkstra’s algorithm, 413–414, 415–417
CPUs
   executing cloth simulation on, 431–432
   heterogeneous future of multicore, 4–7
   matrix multiplication and performance results, 511–513
   SpMV implementation, 518–519
CreateCommandQueue(), 50–51
CreateContext(), 49–50, 375
CreateMemObjects(), 54–55
CSR format, sparse matrix, 517

D
DAG (directed acyclic graph), command-queues and, 310
data load and store functions, vectors, 181–189
data structure, Dijkstra’s algorithm, 412–414
data types
   explicit casts, 116–117
   explicit conversions, 117–121
   implicit type conversions, 110–115
   reference guide for supported, 550–552
   reinterpreting data as other, 121–123
   reserved as keywords in OpenCL C, 141
   scalar. see scalar data types
   specifying attributes, 555
   vector. see vector data types
data-parallel programming model
   overview of, 8–9
   parallel algorithm limitations, 28–29
   understanding, 25–27
   writing kernel using OpenCL C, 97–99
decimation kernel, optical flow, 474
declaration fields, sampler, 577
default device, 69
#define preprocessor directive, 142, 145
denormalized numbers, 34, 388
dense matrix, 499
dense optical flow, 469
derived types, OpenCL C, 109–110
copy
   buffers and sub-buffers, 274–276, 545
   image objects, 302–305, 308, 575
design, for tiled and packetized sparse matrix, 523–524
device_type argument, querying devices, 68
devices
architecture diagram, 577
choosing first available, 50–52
convolution signal example, 89–97
creating context in execution model, 17–18
determining profile support by, 390
embedded profile for hand held, 383–385
executing kernel on, 13–17
execution of Vector Add kernel, 380
full profile for desktop, 383
in platform model, 12
selecting, 70–78
steps in OpenCL, 83–84
DFFT (discrete fast Fourier transform), 453
DFT. see discrete Fourier transform (DFT), Ocean simulation
Dijkstra’s algorithm, parallelizing graph data structures, 412–414
kernels, 414–417
leveraging multiple compute devices, 417–423
overview of, 411–412
dimensions, image object, 282
Direct3D, interoperability with. see interoperability with Direct3D
directed acyclic graph (DAG), command-queues and, 310
directional edge detector filter, Sobel, 407–410
directories, sample code for this book, 41
DirectX Shading Language (HLSL), 111–113
discrete fast Fourier transform (DFFT), 453
discrete Fourier transform (DFT), Ocean simulation
avoiding local memory bank conflicts, 463
determining 2D composition, 457–458
determining local memory needed, 462
determining sub-transform size, 459–460
determining work-group size, 460
obtaining twiddle factors, 461–462
overview of, 457
using images, 463
using local memory, 459
distance(), geometric functions, 175–176
divide (/) arithmetic operator, 124–126
double, vector data load and store, 181
DRAM, modern multicore CPUs, 6–7
dynamic libraries, OpenCL program vs., 97
E
early exit, optical flow algorithm, 483
Eclipse, generating project in, 44–45
directional edgeArray:, Dijkstra’s algorithm, 412–414
“Efficient Sparse Matrix-Vector Multiplication on CUDA” (Bell and Garland), 517
embedded profile
64-bit integers, 385–386
built-in atomic functions, 387
determining device supporting, 390
full profile vs., 383
images, 386–387
mandated minimum single-precision floating-point capabilities, 387–389
OpenCL programs for, 35–36
overview of, 383–385
platform queries, 65
MBEDDED_PROFILE_macro, 390
enumerated type
rank order of, 113
specifying attributes, 555
enumerating, list of platforms, 66–67
equal (==) operator, 127
equality operators, 124, 127
error codes
C++ Wrapper API exceptions, 371–374
c1Barrier(), 313
c1CreateUserEvent(), 321–322
c1EnqueueMarker(), 314
c1EnqueueWaitForEvents(), 314–315
error codes (continued)
  clGetEventProfilingInfo(), 329–330
  clGetProgramBuildInfo, 220–221
  clRetainEvent(), 318
  clSetEventCallback(), 326
  clWaitForEvents(), 323
  table of, 57–61
ERROR_CODE value, command-queue, 311
  .even suffix, vector data types, 107–108
  event data types, 108, 147–148
  event objects
    OpenCL/OpenGL sharing APIs, 579
    overview of, 317–320
    reference guide, 549–550
  event_t async_work_group_copy(), 192, 332–333
  event_t async_work_group_strided_copy(), 192, 332–333
  events
    command-queues and, 311–317
    defined, 310
    event objects. see event objects
    generating on host, 321–322
    impacting execution on host, 322–327
    inside kernels, 332–333
    from outside OpenCL, 333
    overview of, 309–310
    profiling using, 327–332
    in task-parallel programming model, 28
  exceptions
    C++ Wrapper API, 371–374
    execution of Vector Add kernel, 379
    exclusive (^) operator, 128
    exclusive or (^) operator, 127–128
  execution model
    command-queues, 18–21
    contexts, 17–18
    defined, 11
    how kernel executes OpenCL device, 13–17
    overview of, 13
    parallel algorithm limitations, 28–29
  explicit casts, 116–117
  explicit conversions, 117–121, 132
  explicit kernel, SpMV, 519
  explicit memory fence, 570–571
  explicit model, data parallelism, 26–27
  explicit synchronization, 349
  exponent, half data type, 101
  expression, assignment operator, 132
  extensions, compiler directives for
    optional, 143–145

F
  fast Fourier transform (FTT). see Ocean simulation, with FFT
  fast variants, geometric functions, 175
  FBO (frame buffer object), 347
  file, creating 2D image from, 284–285
  filter mode, sampler objects, 282, 292–295
  float channels, 403–406
  float data type, converting, 101
  float images, 386
  float type, math constants, 556
  floating-point arithmetic system, 33–34
  floating-point constants, 162–163
  floating-point data types, 113, 119–121
  floating-point options
    building program object, 224–225
    full vs. embedded profiles, 387–388
  floating-point pragmas, 143, 162
  floatn, vector data load and store
    functions, 181, 182–186
  fma, geometric functions, 175
  formats, image
    embedded profile, 387
    encapsulating information on, 282
    mapping OpenGL texture to OpenCL
      image, 346
    overview of, 287–291
    querying list of supported, 574
    reference guide for supported, 576
  formats, of program binaries, 227
  FP_CONTRACT pragma, 162
  frame buffer object (FBO), 347
  FreeImage library, 283, 284–285
  FreeSurfer. see Dijkstra's algorithm,
    parallelizing
  FTT (fast Fourier transform). see Ocean simulation, with FFT
  full profile
    built-in atomic functions, 387
    determining profile support by
      device, 390
embedded profile as strict subset of, 383–385
mandated minimum single-precision floating-point capabilities, 387–389
platform queries, 65
querying device support for images, 386–387
function qualifiers
overview of, 133–134
reference guide, 554
reserved as keywords, 141
functions. see built-in functions

G
Gaussian filter, 282–283, 295–299
Gauss-Seidel iteration, 432
GCC compiler, 111–113
general-purpose GPU (GPGPU), 10, 29
gentype
   barrier functions, 191–195
   built-in common functions, 173–175
   integer functions, 168–171
   miscellaneous vector functions, 199–200
   vector data load and store functions, 181–189
   work-items, 153–161
gentyped
   built-in common functions, 173–175
   built-in geometric functions, 175–176
   built-in math functions, 155–156
   defined, 153
gentypef
   built-in geometric functions, 175–177
   built-in math functions, 155–156, 160–161
   defined, 153
gentypei, 153, 158
gentypen, 181–182, 199–200
geometric built-in functions, 175–177, 563–564
get_global_id(), data-parallel kernel, 98–99
getInfo(), C++ Wrapper API, 375–377
gl_object_type parameter, query
   OpenGL objects, 347–348
glBuildProgram(), 52–53
   glCreateFromGLTexture2D(), 344–345
   glCreateFromGLTexture3D(), 344–345
   glCreateSyncFromCLeventARB(), 350–351
   glDeleteSync() function, 350
   GLEW toolkit, 336
   glFinish()
      creating OpenCL buffers from OpenGL buffers, 342
      OpenCL/OpenGL synchronization with, 348
      OpenCL/OpenGL synchronization without, 351
   global (_global) address space qualifier, 136, 141
   global index space, kernel execution model, 15–16
   global memory
      device architecture diagram, 577
      matrix multiplication, 507–509
      memory model, 21–23
   globalWorkSize, executing kernel, 56–57
   GLSL (OpenGL Shading Language), 111–113
   GLUT toolkit, 336, 450–451
   glWaitSync(), synchronization, 350–351
   GMCH (graphics/memory controller), 6–7
   gotos, irreducible control flow, 147
   GPGPU (general-purpose GPU), 10, 29
   GPU (graphics processing unit)
      advantages of image objects. see image objects
      defined, 69
      executing cloth simulation on, 432–438
      leveraging multiple compute devices, 417–423
      matrix multiplication and performance results, 511–513
      modern multicore CPUs as, 6–7
      OpenCL implementation for NVIDIA, 40
      optical flow performance, 484–485
      optimizing for SIMD computation and local memory, 441–446
      querying and selecting, 69–70
      SpMV implementation, 518–519
Index

GPU (graphics processing unit) (continued)
tiled and packetized sparse matrix
design, 523–524
tiled and packetized sparse matrix
team, 524
two-layered batching, 438–441
graph data structures, parallelizing
Dijkstra’s algorithm, 412–414
graphics. see also images
shading languages, 111–113
standards, 30–31
graphics processing unit. see GPU
(graphics processing unit)
graphics/memory controller (GMCH),
6–7
gray scale images, applying Sobel
OpenCL kernel to, 409–410
greater than (>) operator, 127
greater than or equal (>=) operator, 127

H
half data type, 101–102
half functions, 153
half-float channels, 403–406
half-float images, 386
halfn, 181, 182–186
hand held devices, embedded profile for.
see embedded profile
hardware
mapping program onto, 9–11
parallel computation as concurrency
enabled by, 8
SpMV kernel, 519
SpMV multiplication, 524–538
hardware abstraction layer, 11, 29
hardware linear interpolation, optical
flow algorithm, 480
hardware scheduling, optical flow
algorithm, 483
header structure, SpMV, 522–523
height map, Ocean application, 450
HelloWorld sample
checking for errors, 57–61
choosing device and creating com-
mand-queue, 50–52
choosing platform and creating
context, 49–50
creating and building program object,
52–53
creating kernel and memory objects,
54–55
downloading sample code, 39
executing kernel, 55–57
Linux and Eclipse, 44–45
Mac OS X and Code::Blocks, 41–42
Microsoft Windows and Visual
Studio, 42–44
overview of, 39, 45–48
prerequisites, 40–41
heterogeneous platforms, 4–7
.hl suffix, vector data types, 107–108
high level loop, Dijkstra’s algorithm,
414–417
histogram. see image histograms
histogram_partial_image_rgba_
unorm8 kernel, 400
histogram_partial_results_rgba_
unorm8 kernel, 400–402
histogram_sum_partial_results_
unorm8 kernel, 400
HLSL (DirectX Shading Language),
111–113
host
calls to enqueue histogram kernels,
398–400
creating, writing and reading buffers
and sub-buffers, 262–268
device architecture diagram, 577
events impacting execution on,
322–327
execution model, 13, 17–18
generating events on, 321–322
kernel execution model, 13
matrix multiplication, 502–505
platform model, 12
host memory
memory model, 21–23
reading image back to, 300–301
reading image from device to,
299–300
reading region of buffer into,
269–272
writing region into buffer from,
272–273
hybrid programming models, 29

592
I
ICC compiler, 111–113
ICD (installable client driver) model, 49, 375
IDs, kernel execution model, 14–15
IEEE standards, floating-point arithmetic, 33–34
image channel data type, image formats, 289–291
image channel order, image formats, 287–291
image data types, 108–109, 147
image difference, optical flow algorithm, 472
image functions
border color, 209–210
querying image information, 214–215
read and write, 201–206
samplers, 206–209
writing to images, 210–213
image histograms
additional optimizations to parallel, 400–402
computing, 393–395, 403–406
overview of, 393
parallelizing, 395–400
image objects
copy between buffer objects and, 574
creating, 283–286, 573–574
creating in OpenCL from OpenGL textures, 344–347
Gaussian filter example, 282–283
loading to in PyOpenCL, 493–494
mapping and ummapping, 305–308, 574
memory model, 21
OpenCL and, 30
OpenCL C functions for working with, 295–299
OpenCL/OpenGL sharing APIs, 578
overview of, 281–282
querying, 575
querying list of supported formats, 574
querying support for device images, 291
read, write, and copy, 575
specifying image formats, 287–291
transfering data, 299–308
image pyramids, optical flow algorithm, 472–479
image3d_t type, embedded profile, 386
ImageFilter2D example, 282–291, 488–492
images
access qualifiers for read-only or write-only, 140–141
describing motion between. see optical flow
DFT, 463
embedded profile device support for, 386–387
formats. see formats, image
as memory objects, 247
read and write built-in functions, 572–573
Sobel edge detection filter for, 407–410
supported by OpenCL C, 99
Image.tostring() method, PyOpenCL, 493–494
implicit kernel, SpMV, 518–519
implicit model, data parallelism, 26
implicit synchronization, OpenCL/OpenGL, 348–349
implicit type conversions, 110–115
index space, kernel execution model, 13–14
INF (infinity), floating-point arithmetic, 34
inheritance, C++ API, 369
initialization
Ocean application overview, 450–451
OpenCL/OpenGL interoperability, 338–340
parallelizing Dijkstra’s algorithm, 415
in-order command-queue, 19–20, 24
input vector, SpMV, 518
installable client driver (ICD) model, 49, 375
integer built-in functions, 168–172, 557–558
integer data types
arithmetic operators, 124–216
explicit conversions, 119–121
rank order of, 113
relational and equality operators, 127
intellectual property, program binaries protecting, 227
interoperability with Direct3D
acquiring/releasing Direct3D objects
in OpenCL, 361–363
creating memory objects from
Direct3D buffers/textures, 357–361
initializing context for, 354–357
overview of, 353
processing D3D vertex data in
OpenCL, 366–368
processing Direct3D texture in
OpenCL, 363–366
reference guide, 579–580
sharing overview, 353–354
interoperability with OpenGL
cloth simulation, 446–448
creating OpenCL buffers from
OpenGL buffers, 339–343
creating OpenCL image objects from
OpenGL textures, 344–347
initializing OpenCL context for,
338–339
optical flow algorithm, 483–484
overview of, 335
querying for OpenGL sharing
extension, 336–337
querying information about OpenGL
objects, 347–348
reference guide, 577–579
sharing overview, 335–336
synchronization, 348–351
irreducible control flow, restrictions,
147
iterations
executing cloth simulation on CPU,
431–432
executing cloth simulation on GPU,
434–435
pyramidal Lucas-Kanade optical flow,
472
simulating soft body, 429–431
creating, and setting kernel arguments, 237–241
executing, 548
managing and querying, 242–245
out-of-order execution of memory
object command and, 549
overview of, 237
program objects vs., 217–218
thread safety, 241–242
_kernel qualifier, 133–135, 141, 217
kernels
applying Phillips spectrum, 453–457
constant memory during execution
of, 21
creating, writing and reading buffers/sub-buffers, 262
creating context in execution model,
17–18
creating memory objects, 54–55,
377–378
in data-parallel programming model,
25–27
data-parallel version of, 97–99
defined, 13
in device architecture diagram, 577
events inside, 332–333
executing and reading result, 55–57
executing Ocean simulation application, 463–468
executing OpenCL device, 13–17
executing Sobel OpenCL, 407–410
executing Vector Add kernel, 381
in execution model, 13
leveraging multiple compute devices,
417–423
in matrix multiplication program,
501–509
parallel algorithm limitations, 28–29
parallelizing Dijkstra’s algorithm,
414–417
programming language and, 32–34
in PyOpenCL, 495–497
restrictions in OpenCL C, 146–148
in task-parallel programming model,
27–28
in tiled and packetized sparse matrix,
518–519, 523
keywords, OpenCL C, 141
Khronos, 29–30

K
kernel attribute qualifiers, 134–135
kernel execution commands, 19–20
kernel objects
arguments and object queries, 548
creating, 547–548

Index
learning OpenCL, 36–37
left shift (<<) operator, 129–130
length(), geometric functions, 175–177
less than (<) operator, 127
less than or equal (<=) operator, 127
library functions, restrictions in OpenCL C, 147

links
cloth simulation using two-layered batching, 438–441
executing cloth simulation on CPU, 431–432
executing cloth simulation on GPU, 433–438
introduction to cloth simulation, 426–428
simulating soft body, 429–431

Linux
generating project in, 44–45
initializing contexts for OpenGL interoperability, 338–339
OpenCL implementation in, 41

load balancing
automatic, 20
in parallel computing, 9
load/store functions, vector data, 567–568
local (_local) address space qualifier, 138–139, 141
local index space, kernel execution model, 15
local memory
device architecture diagram, 577
discrete Fourier transform, 459, 462–463
FFT kernel, 464
memory model, 21–24
optical flow algorithm, 481–482
optimizing in matrix multiplication, 509–511
SpMV implementation, 518–519
local WorkSize, executing kernel, 56–57

logical operators
overview of, 128
symbols, 124

unary not(!), 131
Lucas-Kanade. see pyramidal Lucas-Kanade optical flow algorithm
luminosity histogram, 393
lvalue, assignment operator, 132

Mac OS X
OpenCL implementation in, 40
using Code::Blocks, 41–42
macros
determining profile support by device, 390
integer functions, 172
OpenCL C, 145–146
preprocessor directives and, 555
preprocessor error, 372–374
mad, geometric functions, 175
magnitudes, wave, 454
main() function, HelloWorld OpenCL kernel and, 44–48
mandated minimum single-precision floating-point capabilities, 387–389
mantissa, half data type, 101
mapping
buffers and sub-buffers, 276–279
C++ classes to OpenCL C type, 369–370
image data, 305–308
image to host or memory pointer, 299
OpenGL texture to OpenCL image formats, 346
markers, synchronization point, 314
maskArray:, Dijkstra’s algorithm, 412–414, 415
masking off operation, 121–123
mass/spring model, for soft bodies, 425–427
math built-in functions
accuracy for embedded vs. full profile, 388
floating-point constant, 162–163
floating-point pragma, 162
overview of, 153–161
reference guide, 560–563
relative error as ulps in, 163–168
math constants, reference guide, 556
math intrinsics, program build options, 547
math_ functions, 153
Matrix Market (MM) exchange format, 517–518
matrix multiplication
  basic algorithm, 499–501
  direct translation into OpenCL, 501–505
  increasing amount of work per kernel, 506–509
  overview of, 499
  performance results and optimizing original CPU code, 511–513
  sparse matrix-vector. see sparse matrix-vector multiplication (SpMV)
  using local memory, 509–511
memory access flags, 282–284
memory commands, 19
memory consistency, 23–24, 191
memory latency, SpMV, 518–519
memory model, 12, 21–24
memory objects
  buffers and sub-buffers as, 247–248
  creating context in execution model, 17–18
  creating kernel and, 54–55, 377–378
  matrix multiplication and, 502
  in memory model, 21–24
  out-of-order execution of kernels and, 549
  querying to determine type of, 258–259
  runtime API managing, 32
mesh
  executing cloth simulation on CPU, 431–432
  executing cloth simulation on GPU, 433
  introduction to cloth simulation, 425–428
  simulating soft body, 429–431
  two-layered batching, 438–441
MFLOPS, 512–513
Microsoft Windows
  generating project in Visual Studio, 42–44
  OpenCL implementation in, 40
  OpenGL interoperability, 338–339
mismatch vector, optical flow algorithm, 472
MM (Matrix Market) exchange format, 517–518
multicore chips, power-efficiency of, 4–5
  multiplication matrix. see matrix multiplication
  sparse matrix-vector. see sparse matrix-vector multiplication (SpMV)
multiply (*) arithmetic operator, 124–126

N
n suffix, 181
names, reserved as keywords, 141
NaN (Not a Number), floating-point arithmetic, 34
native kernels, 13
NDRange
  data-parallel programming model, 25
  kernel execution model, 14–16
  matrix multiplication, 502, 506–509
  task-parallel programming model, 27
normalize(), geometric functions, 175–176
not (~) operator, 127–128
not equal (!=) operator, 127
NULL value, 64–65, 68
num_entries, 64, 68
numeric indices, built-in vector data types, 107
numpy, PyOpenCL, 488, 496–497
NVIDIA GPU Computing SDK
  generating project in Linux, 41
  generating project in Linux and Eclipse, 44–45
  generating project in Visual Studio, 42
  generating project in Windows, 40
  OpenCL/OpenGL interoperability, 336
objects, OpenCL/OpenGL sharing API, 579
Ocean simulation, with FFT
  FFT kernel, 463–467
generating Phillips spectrum, 453–457
OpenCL DFT. see discrete Fourier transform (DFT), Ocean simulation overview of, 449–453
transpose kernel, 467–468
odd suffix, vector data types, 107–108
OpenCL, introduction conceptual foundation of, 11–12 data-parallel programming model, 25–27
embedded profile, 35–36
execution model, 13–21
graphics, 30–31
heterogeneous platforms of, 4–7
kernel programming language, 32–34
learning, 36–37
memory model, 21–24
other programming models, 29
parallel algorithm limitations, 28–29
platform API, 31
platform model, 12
runtime API, 31–32
software, 7–10
summary review, 34–35
task-parallel programming model, 27–28
understanding, 3–4
OpenCL C access qualifiers, 140–141
address space qualifiers, 135–140
built-in functions. see built-in functions
derived types, 109–110
explicit casts, 116–117
explicit conversions, 117–121
function qualifiers, 133–134
functions for working with images, 295–299
implicit type conversions, 110
kernel attribute qualifiers, 134–135
as kernel programming language, 32–34
keywords, 141
macros, 145–146
other data types supported by, 108–109
overview of, 97
optimization options
clBuildProgram(), 225–226
partial image histogram, 400–402
program build options, 546
“Optimizing Power Using Transformations” (Chandrakasan et al.), 4–5
“Optimizing Sparse Matrix-Vector Multiplication on GPUs” (Baskaran and Bordawekar), 517
optional extensions, compiler directives for, 143–145
preprocessor directives, 141–144
reinterpreting data as another type, 121–123
restrictions, 146–148
scalar data types, 99–102
type qualifiers, 141
vector data types, 102–108
vector operators. see vector operators
writing data-parallel kernel using, 97–99
OPENCL EXTENSION directive, 143–145
OpenGL interoperability between OpenCL and. see interoperability with Direct3D; interoperability with OpenGL
Ocean application, 450–453
OpenCL and graphics standards, 30
reference guide for sharing APIs, 577–579
synchronization between OpenCL, 333
OpenGL Shading Language (GLSL), 111–113
operands, vector literals, 105
operators, vector. see vector operators
optical flow application of texture cache, 480–481
early exit and hardware scheduling, 483
efficient visualization with OpenGL interoper, 483–484
performance, 484–485
problem of, 469–479
sub-pixel accuracy with hardware linear interpolation, 480
understanding, 469
using local memory, 481–482
optional extensions, compiler directives for, 143–145
Index 597
or (()) operator, 127–128
or (||) operator, 128
out-of-order command-queue
  automatic load balancing, 20
data-parallel programming model, 24
execution model, 20
reference guide, 549
task-parallel programming model, 28
output, creating 2D image for, 285–286
output vector, SpMV, 518
overloaded function, vector literal as, 104–105

P
packets
  optimizing sparse matrix-vector multiplication, 538–539
tiled and packetized sparse matrix, 519–522
tiled and packetized sparse matrix design, 523–524
tiled and packetized sparse matrix team, 524
pad to 128-boundary, tiled and packetized sparse matrix, 523–524
parallel algorithm limitations, 28–29
parallel computation
  as concurrency enabled by software, 8
  of image histogram, 395–400
  image histogram optimizations, 400–402
parallel programming, using models for, 8
parallelism, 8
param_name values, querying platforms, 64–65
partial histograms
  computing, 395–397
  optimizing by reducing number of, 400–402
  summing to generate final histogram, 397–398
partitioning workload, for multiple compute devices, 417–423
Patterns for Parallel Programming (Mattson), 20
performance
  heterogeneous future of, 4–7
  leveraging multiple compute devices, 417–423
matrix multiplication results, 511–513
  optical flow algorithm and, 484–485
  soft body simulation and, 430–431
  sparse matrix-vector multiplication and, 518, 524–538
  using events for profiling, 327–332
  using matrix multiplication for high.
  see matrix multiplication
PEs (processing elements), platform model, 12
phillips function, 455–457
Phillips spectrum generation, 453–457
platform API, 30–31
platform model, 11–12
platforms
  choosing, 49–50
  choosing and creating context, 375
  convolution signal example, 89–97
  embedded profile, 383–385
  enumerating and querying, 63–67
  querying and displaying specific information, 78–83
  querying list of devices associated with, 68
  reference guide, 541–543
  steps in OpenCL, 83–84
pointer data types, implicit conversions, 111
post-increment (++) unary operator, 131
power
  efficiency of specialized core, 5–6
  of multicore chips, 4–5
#pragma directives, OpenCL C, 143–145
predefined identifiers, not supported, 147
prefetch functions, 191–195, 570
pre-increment (--) unary operator, 131
preprocessor build options, 223–224
preprocessor directives
  OpenCL C, 141–142
  program object build options, 546–547
preprocessor error macros, C++ Wrapper API, 372–374
private (_private) address space qualifier, 139, 141
private memory, 21–23, 577
processing elements (PEs), platform model, 12
profiles
associated with platforms, 63–67
commands for events, 327–332
embedded. see embedded profile
reference guide, 549
program objects
build options, 222–227
creating and building, 52–53, 377
creating and building from binaries,
227–236
creating and building from source
code, 218–222
creating and building in PyOpenCL,
494–495
creating context in execution model,
17–18
kernel objects vs., 217–218
managing and querying, 236–237
reference guide, 546–547
runtime API creating, 32
programming language. see also OpenCL
C; PyOpenCL, 32–34
programming models
data-parallel, 25–27
defined, 12
other, 29
parallel algorithm limitations, 28–29
task-parallel, 27–28
properties
device, 70
querying context, 85–87
PyImageFilter2D, PyOpenCL, 488–492
PyOpenCL
context and command-queue
creation, 492–493
creating and building program,
494–495
introduction to, 487–488
loading to image object, 493–494
overview of, 487
PyImageFilter2D code, 488–492
reading results, 496
running PyImageFilter2D example,
488
setting kernel arguments/executing
kernel, 495–496
pyopencl vo-92+, 488
pyopencl.create_some_context(), 492
pyramidal Lucas-Kanade optical flow
algorithm, 469, 471–473
Python, using OpenCL in. see
PyOpenCL
Python Image Library (PIL), 488,
493–494
Q
qualifiers
access, 140–141
address space, 135–140
function, 133–134
kernel attribute, 134–135
type, 141
queries
buffer and sub-buffer, 257–259, 545
device, 542–543
device image support, 291
event object, 319–320
image object, 214–215, 286, 575
kernel, 242–245, 548
OpenCL/OpenGL sharing APIs, 578
OpenGL objects, 347–348
platform, 63–66, 542–543
program object, 241–242, 547
storing program binary and, 230–232
supported image formats, 574
R
R,G, B color histogram
computing, 393–395, 403–406
optimizing, 400–402
overview of, 393
parallelizing, 395–400
rank order, usual arithmetic conversions,
113–115
read
buffers and sub-buffers, 259–268, 544
image back to host memory, 300–301
image built-in functions, 201–206,
298, 572–573
image from device to host memory,
299–300
image objects, 575
memory objects, 248
results in PyOpenCL, 496–497
read_imagef(), 298–299
read-only qualifier, 140–141
read-write qualifier, 141
recursion, not supported in OpenCL C, 147
reference counts
  buffers and sub-buffers, 256
  contexts, 89
  event objects, 318
  regions, memory model, 21–23
relational built-in functions, 175, 178–181, 564–567
relational operators, 124, 127
relaxed consistency model, memory objects, 24
remainder (%) arithmetic operator, 124–126
render buffers, 346–347, 578
rendering of height map, Ocean application, 450
reserved data types, 550–552
restrict type qualifier, 141
restrictions, OpenCL C, 146–148
return type, kernel function restrictions, 146
RGB images, applying Sobel OpenCL kernel to, 409
RGBA-formatted image, loading in PyOpenCL, 493–494
right shift (>>) operator, 129–130
rounding mode modifier
  explicit conversions, 119–121
  vector data load and store functions, 182–189
_rte suffix, 183, 187
runCLSimulation(), 451–457
runtime API, 30–32, 543

S
sampler data types
  determining border color, 209–210
  functions, 206–209
  restrictions in OpenCL C, 108–109, 147
sampler objects. see also image objects
  creating, 292–294
  declaration fields, 577
  functions of, 282
  overview of, 281–282
reference guide, 576–577
  releasing and querying, 294–295
_sat (saturation) modifier, explicit conversions, 119–120
SaveProgramBinary(), creating programs, 230–231
scalar data types
  creating vectors with vector literals, 104–105
  explicit casts of, 116–117
  explicit conversions of, 117–121
  half data type, 101–102
  implicit conversions of, 110–111
  integer functions, 172
  reference guide, 550
  supported by OpenCL C, 99–101
  usual arithmetic conversions with, 113–115
  vector arithmetic operators with. see vector operators
scalar_add(), writing data-parallel kernel, 97–98
754 formats, IEEE floating-point arithmetic, 34
sgentype
  integer functions, 172
  relational functions, 181
shape matching, soft bodies, 425
sharing APIs, OpenCL/OpenGL, 577–579
shift operators, 124, 129–130
shuffle, illegal usage of, 214
shuffle2, illegal usage of, 214
sign, half data type, 101
SIMD (Single Instruction Multiple Data) model, 26–27, 465
simulation
  cloth. see cloth simulation in Bullet Physics SDK
  ocean. see Ocean simulation, with FFT
Single Instruction Multiple Data (SIMD) model, 26–27, 465
Single Program Multiple Data (SPMD) model, 26
single-source shortest-path graph algorithm. see Dijkstra’s algorithm, parallelizing
64-bit integers, embedded profile, 385–386
sizeof operator, 131–132
slab, tiled and packetized sparse matrix, 519
Sobel edge detection filter, 407–410
soft bodies
executing cloth simulation on CPU, 431–432
executing cloth simulation on GPU, 432–438
interoperability with OpenGL, 446–448
introduction to cloth simulation, 425–428
simulating, 429–431
software, parallel, 7–10
solveConstraints, cloth simulation on GPU, 435
solveLinksForPosition, cloth simulation on GPU, 435
source code
creating and building programs from, 218–222
program binary as compiled version of, 227
sparse matrix-vector multiplication (SpMV)
algorithm, 515–517
defined, 515
description of, 518–519
header structure, 522–523
optional team information, 524
other areas of optimization, 538–539
overview of, 515
tested hardware devices and results, 524–538
tiled and packetized design, 523–524
tiled and packetized representation of, 519–522
specify type attributes, 555
SPMD (Single Program Multiple Data) model, 26
SpMV. see sparse matrix-vector multiplication (SpMV)
storage
image layout, 308
sparse matrix formats, 517
strips, tiled and packetized sparse matrix, 519
struct type
restrictions on use of, 109–110, 146
specifying attributes, 555
sub-buffers. see buffers and sub-buffers
sub-pixel accuracy, optical flow algorithm, 480
subregions, of memory objects, 21
subtract (−) arithmetic operator, 124–126
sub-transform size, DFT, 459–460
suffixes, vector data types, 107–108
synchronization
commands, 19–21
computing Dijkstra’s algorithm with kernel, 415–417
explicit memory fence, 570–571
functions, 190–191
OpenCL/OpenGL, 342, 348–351
primitives, 248
synchronization points
defining when enqueuing commands, 312–315
in out-of-order command-queue, 24

T
T1 to T3 data types, rank order of, 114
task-parallel programming model
overview of, 9–10
parallel algorithm limitations, 28–29
understanding, 27–28
team information, tiled and packetized sparse matrix, 524
ternary selection (?,?,?): operator, 129
Tessendorf, Jerry, 449, 454
tetrahedra, soft bodies, 425–428
texture cache, optical flow algorithm, 480–482
texture objects, OpenGL. see also image objects
creating image objects in OpenCL from, 344–347
Ocean application creating, 451
OpenCL/OpenGL sharing APIs, 578
querying information about, 347–348
thread safety, kernel objects, 241–242
tiled and packetized sparse matrix
defined, 515
design considerations, 523–524
tiled and packetized sparse matrix
(continued)
header structure of, 522–523
overview of, 519–522
SpMV implementation, 517–518
timing data, profiling events, 328
traits, C++ template, 376
transpose kernel, simulating ocean, 467–468
twiddle factors, DFT
FFT kernel, 464–466
obtaining, 461–462
using local memory, 463
2D composition, in DFT, 457–458
two-layered batching, cloth simulation, 438–441
type casting, vector component, 554
type qualifiers, 141

U
ugentype, 168–169, 181
ugentypen, 214–215
ulp values, 163–168
unary operators, 124, 131–132
union type, specifying attributes, 555
updatingCostArray::, Dijkstra’s algorithm, 413–417
usual arithmetic conversions, 113–115

V
vadd() kernel, Vector Add kernel, 378
variable-length arrays, not supported in
OpenCL C, 147
variadic macros and functions, not
supported in OpenCL C, 147
VBO (vertex buffer object), 340–344, 446–448
vbo_cl_mem, creating VBO in OpenGL, 340–341
Vector Add example. see C++ Wrapper
API, Vector Add example
vector data types
application, 103–104
built-in, 102–103
components, 106–108, 552–554
data load and store functions,
181–189
explicit casts, 116–117
explicit conversions, 117–121
implicit conversions between,
110–113
literals, 104–105
load/store functions reference,
567–568
miscellaneous built-in functions,
199–200, 571
operators. see vector operators
optical flow algorithm, 470–472
reference guide, 550
supported by OpenCL C, 99
usual arithmetic conversions with,
113–115
vector literals, 104–105
vector operators
arithmetic operators, 124–126
assignment operator, 132
bitwise operators, 127–128
conditional operator, 129
logical operators, 128
overview of, 123–124
reference guide, 554
relational and equality operators, 127
shift operators, 129–130
unary operators, 131–132
vertex buffer object (VBO), 340–344, 446–448
vertexArray::, Dijkstra’s algorithm, 412–414
vertical filtering, optical flow, 474
vertices
introduction to cloth simulation, 425–428
simulating soft body, 429–431
Visual Studio, generating project in, 42–44
vload_half(), 101, 182, 567
vload_halfn(), 182, 567
vloada_half(), 185–186, 568
vloadn(), 181, 567
void return type, kernel functions, 146
void wait_group_events(), 193, 332–333
volatile type qualifier, 141
voltage, multicore chip, 4–5
vstore_half()
half data type, 101
reference guide, 568
vector store functions, 183, 187
vstore_halfn(), 184, 186–188, 568
vstorea_halfn(), 186, 188–189, 568
vstoren(), 182, 567
VSTRIDE, FFT kernel, 464

W
wave amplitudes, 454
weightArray:, Dijkstra’s algorithm, 412–414
Windows. see Microsoft Windows
work-group barrier, 25–27
work-groups
data-parallel programming model, 25–27
global memory for, 21
kernel execution model, 14–16
local memory for, 21, 23
SpMV implementation, 518
tiled and packetized sparse matrix team, 524
work-items
barrier functions, 190–191
built-in functions, 557
data-parallel programming model, 25–27
functions, 150–152
global memory for, 21
kernel execution model, 13–15
local memory for, 23
mapping get_global_id to, 98–99
matrix multiplication, 501–509
private memory for, 21
task-parallel programming model, 27
write
buffers and sub-buffers, 259–268, 544–545
image from host to device memory, 301–302
image objects, 575
memory objects, 248
write_imagef(), 298–299
write-only qualifier, 140–141

Z
0 value, 64–65, 68