Graphics Programming on the Web
WebCL Course Notes
Siggraph 2012

Mikaël Bourges-Sévenier[1]
Motorola Mobility, Inc.

Abstract

This document introduces WebCL [1], a new standard under development by the Khronos Group, for high-performance computing in web browsers. Since WebCL wraps OpenCL, the course starts by reviewing important OpenCL [2] concepts. Next, we detail how to program with WebCL in the browser and on devices such as GPUs. Finally, we discuss WebCL – WebGL [3] interoperability and provide complete examples of moving from WebGL shaders to WebCL. Last, we provide tips and tricks to ease such translation and to optimize WebCL code performance.

1 mikeseven@acm.org

Table of Content

1 What is WebCL?

In short, WebCL is to OpenCL what WebGL is to OpenGL. WebCL is a JavaScript API over OpenCL API; Khronos Group is defining all these international standards. Historically, OpenGL was defined as a standard for hardware accelerated graphics, hence Graphics Language. OpenGL was first a fixed pipeline a programmer could change various states to produce images. Then, OpenGL pipeline became programmable using shaders, pieces of C like code that can be inserted at some points of the OpenGL rendering pipeline.

As the need for more complex applications arise, programmers realized that shaders could be used for more general programming problems, taking advantage of the massively parallel nature of GPUs; this became known as GPGPU. But shaders can only provide limited features for such applications.

Few years ago, Apple proposed OpenCL to the Khronos Group, a more general framework for computing, hence the term Compute Language. Not only OpenCL allows usage of GPUs but also any devices that has a driver in the machine: CPUs, DSPs, accelerators, and so on.

It is important to note that OpenCL doesn't provide any rendering capability, unlike OpenGL; it only processes data, lots of data. The source of such data could be OpenGL buffers such as vertex buffers, pixel buffers, render buffers, and so on.

To understand WebCL, it is necessary to understand the OpenCL programming model.

2 Glossary and conventions

Work-item	The basic unit of work of an OpenCL device
Work-group	Work-items execute together as a work-group
Kernel	The code of a work-item, a C99 function
Program	A collection of kernels and other functions, same as a dynamic library
Context	The environment within which work-items executes. This includes devices, their memories, their command queues, and so on

In this course, we will use the following conventions:

- Code is a yellow box
  - All lines are numbered
  - WebCL/OpenCL keywords and methods are in **bold red**
  - Comments are in light green
  - Language keywords are in **bold dark purple**
  - Strings are in blue

```
__kernel
void multiply(__global const float *a, // a, b, c values are in global memory
```

- The method console.log() is simplified to log().
- All WebCL calls throw exceptions (unlike WebGL that return error codes). For simplicity, we may omit try/catch in this document, but you should not!
- OpenCL qualifiers start with __ (two '_'). For example, one could use __kernel or kernel interchangeably. In this document, we always use __kernel.
- We use interchangeably CL for OpenCL and WebCL, GL for OpenGL ES 2.x and WebGL.
- OpenCL files end with extension '.cl'. On web pages, we use <script type = "x-webcl">, although both are not defined by any standard.

3 Thinking in parallel

Programming a massively parallel device is challenging and, for many developers, may require learning new programming skills. By massively parallel, we mean that many hardware-processing units run at once or, said differently, many hardware threads are running concurrently. While CPUs tend to have 2, 4, or 8 cores, GPUs can have thousands of cores. Even on mobile devices, GPUs with hundred of cores are coming. For web developers used to sequential event-based programs, with JavaScript language not providing threading support, it is a radical shift.

The following example shows the main idea:

- A traditional loop over a (large) set of data can be replaced by a data-parallel kernel
- Each work-item runs a copy of the kernel function.
- With n work-items, the computation is executed in 1 pass vs. n passes with a traditional loop

The OpenCL concepts are introduced in the next section.

```
// in JavaScript

function multiply(a, b, n)
{
  var c=[];
  for(var i = 0; i < n; i++)
    c[i] = a[i] * b[i];

  return c;
}
```

```
// in OpenCL

__kernel
void multiply(__global const float *a, // a, b, c values are in global memory
              __global const float *b,
              __global float *c, int n)
{
  int id = get_global_id(0); // work-item globalID
  if(id >= n) return;        // make sure work-item don't read/write past array size
  c[id] = a[id] * b[id];
}
```

Code 1 – Representing a JavaScript method into a WebCL kernel.

4 OpenCL memory model

Before we enter into OpenCL programming details, it is important to understand its platform model:

- A **Host** contains one or more **Compute Devices**. A Host has its own memory.
- Each Compute Device (e.g. CPU, GPU, DSP, FPGA...) is composed of one or more **compute units** (e.g. cores). Each Compute Device has its own memory.
- Each **Compute Unit** is divided in one or more **Processing Elements** (e.g. hardware threads). Each processing element has its own memory.

In general, we will refer to Host for the device onto which the WebCL program is executed (i.e. within the browser). We refer to Device for a compute device onto which an OpenCL Kernel is executed. Hence, a CPU can be both a Host and a Compute Device.

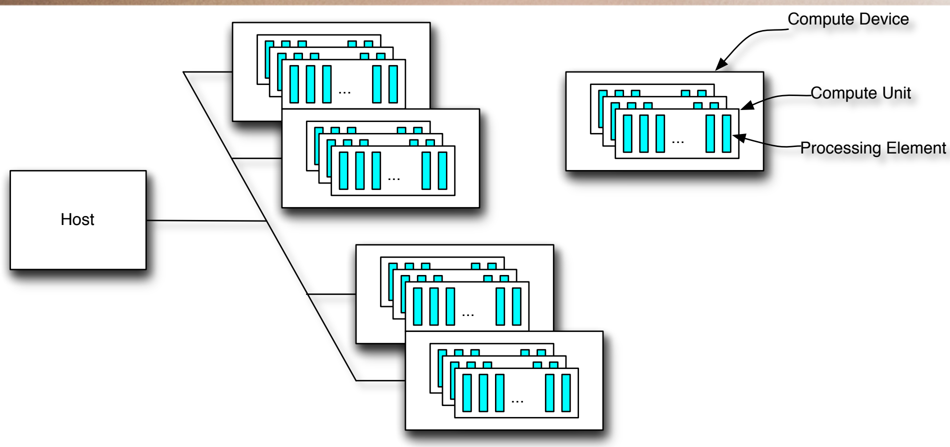

Figure 1 - OpenCL platform model

OpenCL defines 4 types of memory spaces within a Compute Device:

- **Global memory** – corresponds to the device RAM. This is where input data are stored. Available to all work groups/items. Similar to system memory over a slow bus, rather slow memory. Not cached.
- **Constant memory** – cached global memory
- **Local memory** – high-speed memory shared among work-items of a compute unit (i.e. for a work-group). Similar to L1 cache. Reasonably fast memory.
- **Private memory** – registers of a work-item. Very fast memory.

However, private memory is small and local memory is often no more than 64 KB. As a result, programmer must choose carefully which variables leave in a memory space for the best performance / memory access performance tradeoff.

Another type of memory is **Texture Memory**, which is similar to Global Memory but is cached, optimized for 2D spatial locality, and designed for streaming reads with constant latency. In other words, if your device has image support and your data can fit in texture memory, it may be better than using buffers in global memory.

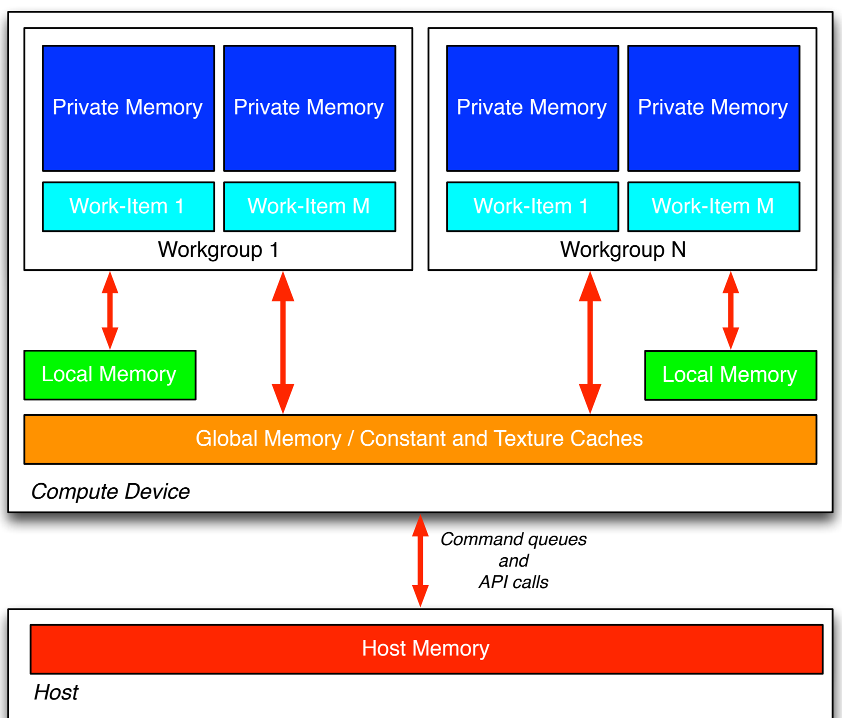

Figure 2 – OpenCL memory model

Finally, at an even lower level, work-items are scheduled as a group called *warp* (NVidia) or *wavefront* (AMD); this is the smallest unit of parallelism on a device. Individual work-items in a warp/wavefront start together at the same program address, but they have their own address counter and register state and are therefore free to branch and execute independently [8]. Threads on a CPU are generally heavyweight entities and context switches (when the operating system swap two threads on and off execution channels) are therefore expensive. By comparison, threads on a GPU (i.e. work-items) are extremely lightweight entities. Since registers are allocated to active threads, no swapping of registers and state occurs between GPU threads. Once threads complete, its resources are de-allocated.

Each work-item has a global ID into an N-Dimensional index space, where N can be 1, 2 or 3. An N-dimensional range (or NDRange) is defined by an array of N values specifying the extent of the index space in each dimension starting at an offset F (0 by default). Within a work-group, a work-item also has a local ID.

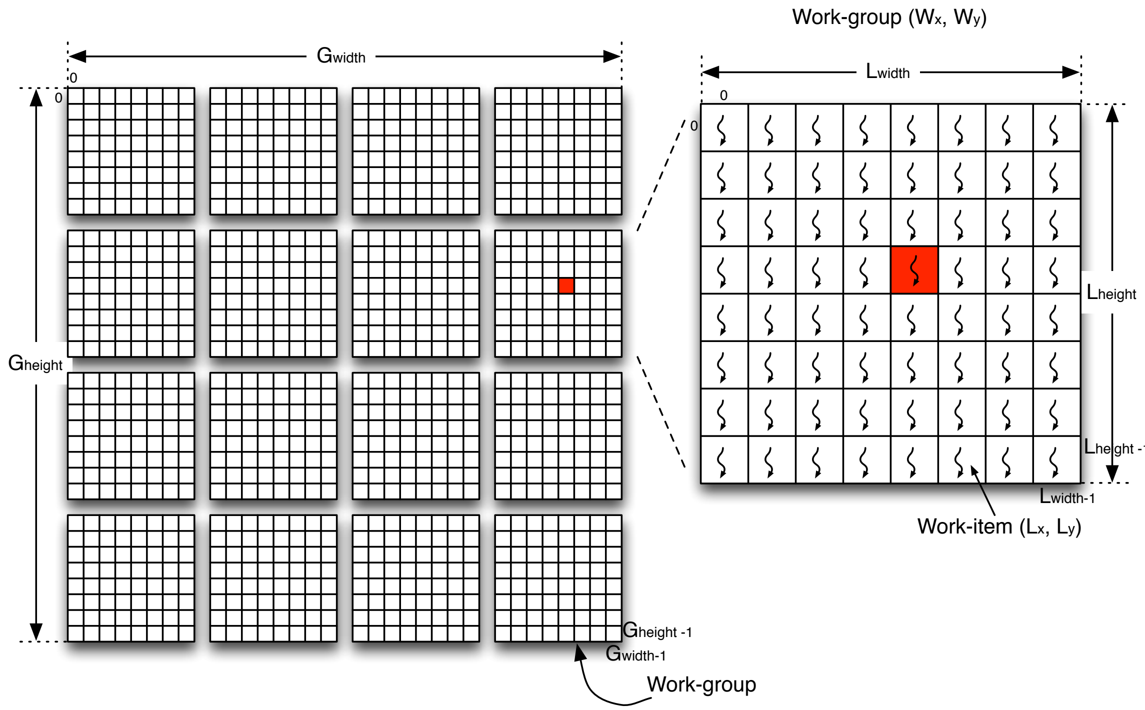

Figure 3 – Global and Local IDs for a 2D problem.

Using a 2D example, as depicted in Figure 3, with a global NDRange of size [G_{width}, G_{height}] and local NDRange of size [L_{width}, L_{height}],

1. Indexes always go from 0 to range-1 in each dimension
2. localID of work-item at index (l_x, l_y) is $l_x + l_y * L_{width}$
3. globalID of work-item at index (g_x, g_y) is $g_x + g_y * G_{width}$

To favor memory coalescing (i.e. the device accesses memory in a batch rather than individual accesses that would require serialized accesses to memory), it is useful to keep:

1. The G_{width} of the problem as a multiple of the maximum work-group size, eventually adding extra columns with appropriate padding. The maximum work-group size is given by cl.KERNEL_WORK_GROUP_SIZE
2. The L_{width} of a work-group as a multiple of the warp/wavefront size. This value is given by cl.KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE

Both limits can be queried on a WebCLKernel object once it is created. They are extremely important for maximum throughput.

Host and devices communicate via buffers defined in an OpenCL context. Commands are sent to devices via command-queues. Commands are used for memory transfers from host and devices, between memory objects in a device, and to execute programs.

5 Programming with WebCL

Programming with WebCL is composed of 2 parts:

- The host side (e.g. in the web browser) that sets up and controls the execution of the program
- The device side (e.g. on a GPU) that runs computations i.e. kernels.

5.1 Host/Browser side

All WebCL methods may throw exceptions (rather than error codes as in WebGL), so you should wrap your WebCL methods with try/catch, even though for simplicity we will omit them in this document.

```
try {
  webclobject.method(…);
}
catch(ex) {
  // an exception occurred
  log(ex);
}
```

Code 2 – Always wrap WebCL method calls with try/catch!

Unlike WebGL, WebCL is a global object so that it can be used in a Web page or within a Web Worker. Consequently, we first need to create a WebCL object:

```
var cl = new WebCL();
```

Code 3 – Creating the WebCL object.

The remainder of this section will detail how to use all WebCL objects in Figure 4.

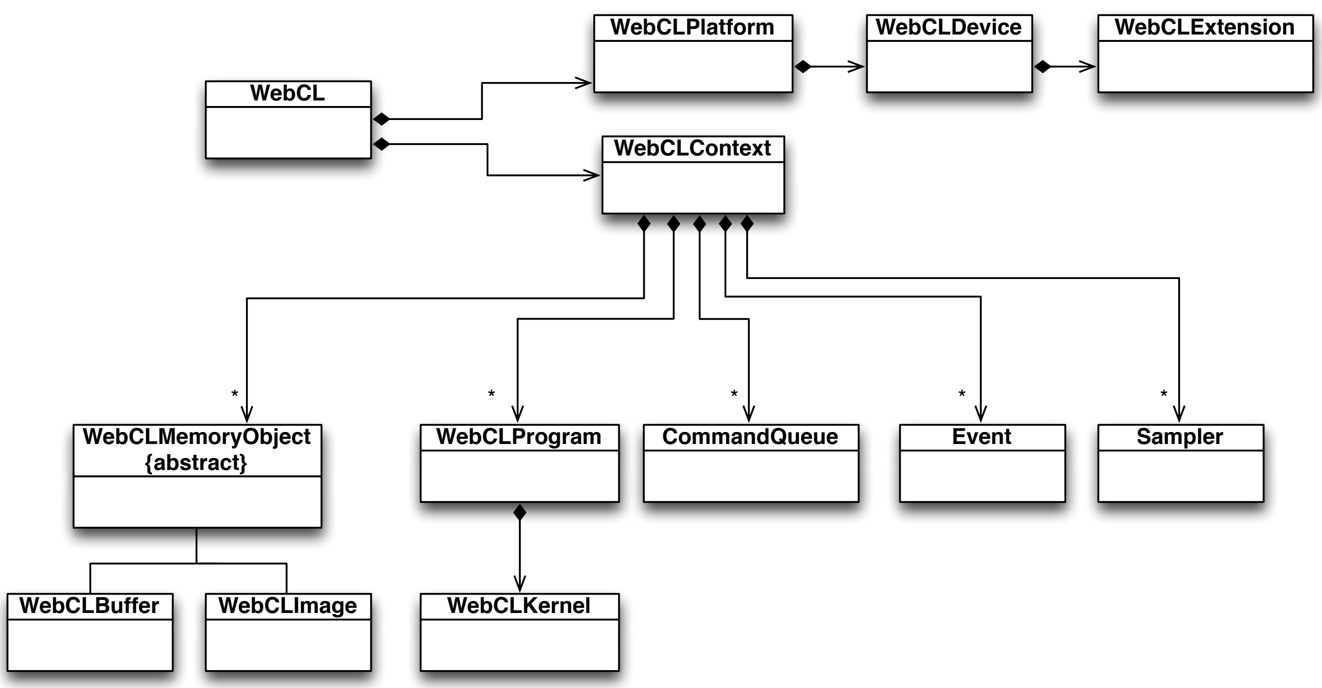

Figure 4 – WebCL objects.

The typical workflow is described in Figure 5 and consists in 3 phases:

- Initialize the platform layer
- Load and compile programs/kernels
- Interact with devices through the runtime layer

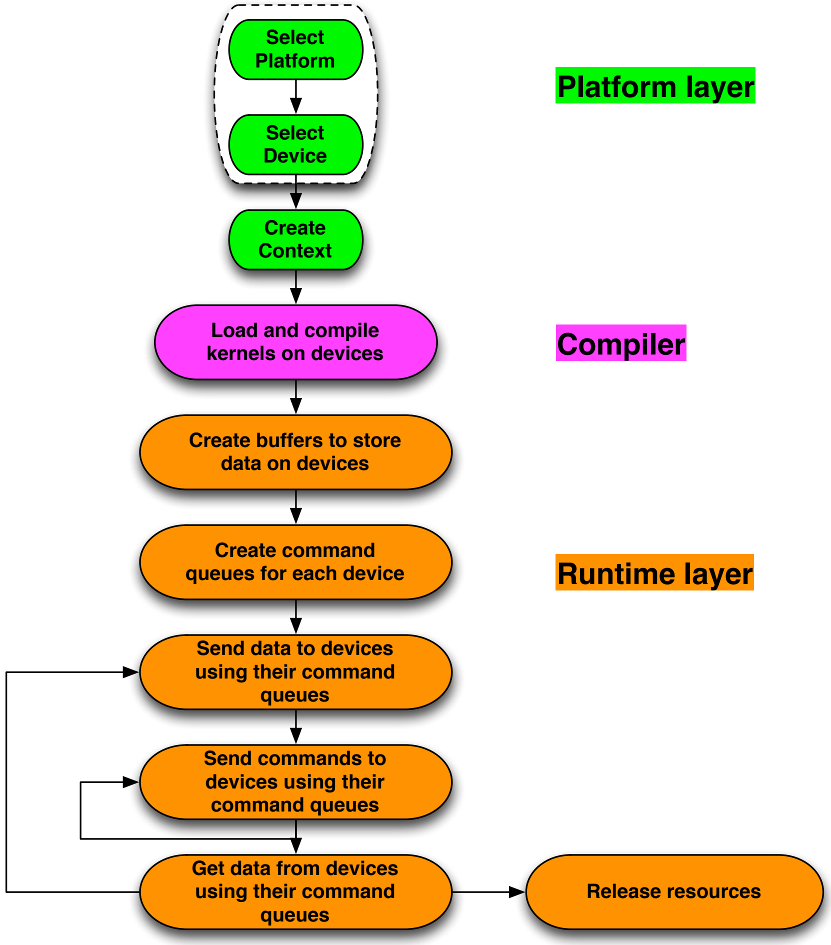

Figure 5 – OpenCL startup sequence

5.1.1 Platform layer

The OpenCL platform layer implements platform-specific features. They allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

```
// let's get all platforms on this machine
var platforms = cl.getPlatforms();

// dump information about each platform
for (var i = 0, il = platforms.length; i < il; ++i) {
  var p = platforms[i];
  var profile = p.getInfo(WebCL.PLATFORM_PROFILE);
  var version = p.getInfo(WebCL.PLATFORM_VERSION);
  var extensions = p.getInfo(WebCL.PLATFORM_EXTENSIONS);

  // list of devices on this platform p
  var devices = p.getDevices(WebCL.DEVICE_TYPE_ALL);

  // find appropriate device
  for (var j = 0, jl = devices.length; j < jl; ++j) {
    var d = devices[j];
    var devExts = d.getInfo(WebCL.DEVICE_EXTENSIONS);
    var devGMem = d.getInfo( WebCL.DEVICE_GLOBAL_MEM_SIZE);
    var devLMem = d.getInfo( WebCL.DEVICE_LOCAL_MEM_SIZE);
    var devCompUnits = d.getInfo( WebCL.DEVICE_MAX_COMPUTE_UNITS);
    var dev_hasImage = d.getInfo( WebCL.DEVICE_IMAGE_SUPPORT);
    var devHasImage = d.getInfo( WebCL.DEVICE_IMAGE_SUPPORT);

    // select device that match your requirenements
    …
```

```
  }

  // assuming we found the best device, we can create the context
  var context = cl.createContext( {
  'platform': platform,
  'device': device,
} );
```

Code 4 – Query WebCL platforms and devices features.

In general, to ensure your algorithm is portable across various devices (even on the same machine!), it is necessary to know details about features on each device. For example, if you require image support, ensure the device you choose support them and up to what size, and how many images can be supported at once. If your kernel requires atomics, make sure device's extensions return 'cl_khr_int64_base_atomics'. On embedded devices, knowing that 'cl_khr_fp16' is supported (i.e. 16-bit floats or half-floats) can lead to twice more performance. When optimizing algorithms, knowing the maximum workgroup size, the number of work-items per dimension, the number of parameters to a kernel function, the maximum size of a memory object, and other features, are crucial elements to adapt your applications at runtime.

On the other end, if you just want to use the best device on the machine and let the browser find it for you, you could just do:

```
var ctx = cl.createContext( {
  deviceType : cl.DEVICE_TYPE_GPU
} );

// query the platform/device found by the browser
try {
  devices = ctx.getInfo(cl.CONTEXT_DEVICES);
catch(ex) {
  throw "Error: Failed to retrieve compute devices for context!";
}

var device = null, platform = null;

for(var i=0, il=devices.length; i < il; ++i) {
  device_type = devices[i].getInfo(cl.DEVICE_TYPE);
  if (device_type == cl.DEVICE_TYPE_GPU) {
    device = devices[i];
    break;
  }
}

if (device)
 platform = device.getInfo(cl.DEVICE_PLATFORM);
```

Code 5 – Let the browser figures the best platform/device for a context.

Note: in practice, the algorithm in Code 5 is often simplified with

```
var devices = ctx.getInfo(cl.CONTEXT_DEVICES);
var device = devices[0];
var platform = device.getInfo(cl.DEVICE_PLATFORM);
```

but this assumes the machine has only 1 GPU device!

Now that we have created a WebCLContext object, we need to set it up for our program and run it!

5.1.2 Runtime layer

The runtime layer manages OpenCL objects such as command-queues, memory objects, program objects, kernel objects in a program and calls that allow you to enqueue commands to a command-queue such as executing a kernel, reading, or writing a memory object.

WebCL defines the following objects:

- Command Queues
- Memory objects (Buffer and Images)
- Sampler objects describe how to sample an image being read by a kernel
- Program objects that contain a set of kernel functions identified with __kernel qualifier in the program source

- Kernel objects encapsulate the specific __kernel functions declared in a program source and its argument values to be used when executing the __kernel function
- Event objects used to track the execution status of a command as well as to profile a command
- Command synchronization objects such as Markers and Barriers

5.1.2.1 Loading and building programs

WebCL, like WebGL 1.0, assumes program to be provided in source code form i.e. a large string. Currently, any WebCL device is required to have an internal compiler. The source code is first loaded to the device, then compiled. As with any compiler, CL defines standard compilation options including the standard –D (predefined name and value) and –I (include directory). Code 6 shows how to properly catch compilation errors using WebCLProgram.getBuildInfo().

```
// Create the compute program from the source strings
program = ctx.createProgram(source);

// Build the program executable with relaxed math flag
try {
  program.build(device, "-cl-fast-relaxed-math");
} catch (err) {
  throw 'Error building program: ' + err
        + program.getBuildInfo(device, cl.PROGRAM_BUILD_LOG));
}
```

Code 6 – Load and build a CL program.

Note: WebCL currently only supports source code as a set of strings.

At this point, our program is compiled, and contains one or more kernel functions. These kernel functions are the entry points of our program, similar to entry points of a shared library. To refer to each kernel function, we create a WebCLKernel object:

```
// Create the compute kernels from within the program
kernel = program.createKernel('kernel_function_name');
```

Code 7 – Create a kernel object for each kernel function in the program.

In the next section, we will discover how to pass arguments to the kernel functions.

5.1.2.2 Passing arguments to kernels

A kernel function may have one or more arguments, like any function. Since JavaScript only offers the type Number for numerical values, we need to pass the type of such value to the kernel object for each argument. For other type of values, we must use WebCL objects:

- WebCLBuffer and WebCLImage that wrap a Typed Array [1]
- WebCLSampler for sampling an image

5.1.2.3 Creating memory objects

A WebCLBuffer object stores a one-dimensional collection of elements. Elements of a buffer can be scalar type (e.g. int, float), vector data type, or user-defined structure.

```
// create a 1D buffer
var buffer = ctx.createBuffer(flags, sizeInBytes, optional srcBuffer);
```

Flag	Description
cl.MEM_READ_WRITE	Default. Memory object is read and written by kernel
cl.MEM_WRITE_ONLY	Memory object only written by kernel
cl.MEM_READ_ONLY	Memory object only read by kernel
cl.MEM_USE_HOST_PTR	Implementation uses storage memory in srcBuffer. srcBuffer must be specified.
cl.MEM_ALLOC_HOST_PTR	Implementation requests OpenCL to allocate host memory.
cl.MEM_COPY_HOST_PTR	Implementation request OpenCL to allocate host memory and copy

	data from srcBuffer memory. srcBuffer must be specified.

Reading from a WRITE_ONLY memory object, or Writing to a READ_ONLY memory object, is undefined. These flags are mutually exclusive.

srcBuffer must be a Typed Array already allocated by the application and sizeInBytes ≥ srcBuffer.byteLength.

MEM_USE_HOST_PTR is mutually exclusive with MEM_ALLOC_HOST_PTR and MEM_COPY_HOST_PTR. However, MEM_COPY_HOST_PTR can be specified with MEM_ALLOC_HOST_PTR. On AMD and NVidia GPUs and on some operating systems, using MEM_ALLOC_HOST_PTR may result in pinned host memory to be used, which may result in improved performance [8][9].

A sub-buffer can be created from an existing WebCLBuffer object as a new WebCLBuffer object.

```
// create a sub-buffer
var subbuffer = buffer.createSubBuffer(flags, offset, size);
```

Note: only reading from a buffer object and its sub-buffer objects or reading from multiple overlapping sub-buffer objects is defined. All other concurrent reading or writing is undefined.

A WebCLImage is used to store a 1D, 2D, or 3D dimensional texture, render-buffer, or image. The elements of an image object are selected from a predefined list of image formats. However, currently, WebCL only supports 2D images.

```
// create a 32-bit RGBA WebCLImage object
// first, we define the format of the image
var InputFormat = {
  'order' : cl.RGBA,
  'data_type' : cl.UNSIGNED_INT8,
  'size': [ image_width, image_height ],
  'rowPitch': image_pitch
};

// Image on device
var image = ctx.createImage(cl.MEM_READ_ONLY | cl.MEM_USE_HOST_PTR, format, imageBuffer);
```

'order' refers to the memory layout in which pixel data channels are stored in the image. 'data_type' is the type of the channel data type.

'size' refers to the image size.

'rowPitch' refers to the scan-line pitch in bytes. If imageBuffer is null, it must be 0. Otherwise, it must be at least image_width * sizeInBytesOfChannelElement, which is the default if rowPitch is not specified.

imageBuffer is a Typed Array that contain the image data already allocated by the application. imageBuffer.byteLength >= rowPitch * image_height. The size of each element in bytes must be a power of 2.

A WebCLSampler describes how to sample an image when the image is read in a kernel function. It is similar to WebGL samplers.

```
// create a sampler object
var sampler = ctx.createSampler(normalizedCoords, addressingMode, filterMode);
```

normalizedCoords is cl.TRUE or true indicates image coordinates specified are normalized.

addressingMode indicated how out-of-range image coordinates are handled when reading an image. This can be set to CL_ADDRESS_MIRRORED_REPEAT, CL_ADDRESS_REPEAT, CL_ADDRESS_CLAMP_TO_EDGE, CL_ADDRESS_CLAMP and CL_ADDRESS_NONE.

filterMode specifies the type of filter to apply when reading an image. This can be cl.FILTER_NEAREST or cl.FILTER_LINEAR.

5.1.2.4 Passing arguments to a kernel

Passing arguments to a kernel function is complicated by JavaScript un-typed nature: JavaScript provides a Number object and there is no way to know if this is a 32-bit integer, a 16-bit short, a 32-bit float, and so on. In fact, JavaScript numbers are typically 64-bit double. As a result, developers must provide the type of arguments used in a kernel function.

The WebCLKernel.setArg() method has two definitions: one for scalar and vector types and one for memory objects (buffers and images) and sampler objects. Table 1 provides the relationships between OpenCL C types and values used in kernel methods' arguments and setArg() arguments.

Values referring to local memory use the special type cl.type.LOCAL_MEMORY_SIZE because local variables can't be initialized by host or device but host can tell the device how many bytes to allocate for a kernel argument.

As a rule of thumb, scalar values are passed by value directly in setArg(). Buffers/Images/Vectors values are passed by commands to transfer their host memory to the device memory.

```
// Sets value of kernel argument idx with value of scalar/vector type
kernel.setArg(idx, value, type);

// Sets value of kernel argument idx with value as memory object or sampler
kernel.setArg(idx, a_webCLObject);
```

Code 8 – WebCLKernel.setArg() definition

For example,

```
// Sets value of argument 0 to the integer value 5
kernel.setArg(0, 5, cl.type.INT);

// Sets value of argument 1 to the float value 1.34
kernel.setArg(1, 1.34, cl.type.FLOAT);

// Sets value of argument 2 as a 3-float vector
// buffer should be a FloatBuffer
kernel.setArg(2, buffer, cl.type.FLOAT | cl.type.VEC3);

// Sets value of argument 3 to a buffer (same for image and sampler)
kernel.setArg(3, buffer);

// Allocate 4096 bytes of local memory for argument 4
kernel.setArg(4, 4096, cl.LOCAL_MEMORY_SIZE);
```

Code 9 – Setting kernel arguments.

Kernel argument type	setArg() value	setArg() cl.type	Remarks
char, uchar	scalar	CHAR, UCHAR	1 byte
short, ushort	scalar	SHORT, USHORT	2 bytes
int, uint	scalar	INT, UINT	4 bytes
long, ulong	scalar	LONG, ULONG	4 bytes
float	scalar	FLOAT	4 bytes
half, double	scalar	HALF, DOUBLE	No on all implementations 2 bytes (half), 8 bytes (double)
<char…double>*N*	WebCLBuffer	VEC*N*	*N* = 2, 3, 4, 8, 16 May be null if global or constant value
char, …, double *	WebCLBuffer		May be null if global or constant value
image2d_t	WebCLImage		
sampler_t	WebCLSampler		
__local		LOCAL_MEMORY_SIZE	Size initialized in kernel

Table 1 – Relationships between C types used in kernels and setArg()'s cl.type.*

If the argument of a kernel function is declared with the __constant qualifier, the size in bytes of the memory object cannot exceed cl.DEVICE_MAX_CONSTANT_BUFFER_SIZE.

Note 1: OpenCL allows passing structures as byte arrays to kernels but WebCL currently doesn't for portability. The main reason is that endianness between host and devices may be different and this would require developers to format their data for each device's endianness even on the same machine.

Note 2: all WebCL API calls are thread-safe, except kernel.setArg(). However, kernel.setArg() is safe as long as concurrent calls operate on different WebCLKernel objects. Behavior is undefined if multiple threads call on the same WebCLKernel object at the same time.

5.1.2.5 Controlling device execution with command queues

Operations on WebCL objects such as memory, program and kernel objects are performed using command queues. A command queue contains a set of operations or commands. Applications may use multiple independent command queues without synchronization as long as commands don't apply on shared objects between command queues. Otherwise, synchronization is required.

Commands are queued in order but execution may be in order (default) or out of order. This means that if a command-queue contains command A and command B, an in-order command-queue object guarantees that command B is executed when command A finishes. If an application configures a command-queue to be out-of-order, there is no guarantee that commands finish in the order they were queued. For out-of-order queues, a wait for events or a barrier command can be enqueued in the command-queue to guarantee previous commands finish before the next batch of commands is executed. Out-of-order queues are an advanced topic we won't cover in this course. Interested readers should refer to Derek Gerstmann Siggraph Asia 2009 on Advanced OpenCL Event Model Usage [20]. Moreover, device support for out-of-order queues is optional in OpenCL and many current drivers don't support it. It is useful to test for out-of-order support and, if an exception is thrown, then create an in-order queue.

```
// Create an in-order command queue (default)
var queue = ctx.createCommandQueue(device);

// Create an in-order command queue with profiling of commands enabled
var queue = ctx.createCommandQueue(device, cl.QUEUE_PROFILING_ENABLE);

// Create an out-of-order command queue
var queue = ctx.createCommandQueue(device, cl.QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);
```

Note: a command queue is attached to a specific device. And multiple command queues can be used per device. One application is to overlap kernel execution with data transfers between host and device [8]. Figure 6 shows the timing benefit if a problem could be separated in half:

- The first half of the data is transferred from host to, taking half the time of the full data set. Then kernel is executed, possibly in half time needed with the full data set. And finally result is transferred back to device in half the time of the full result set.
- Just after the first half is transferred, the second half is transferred from host to device, and the same process is repeated.

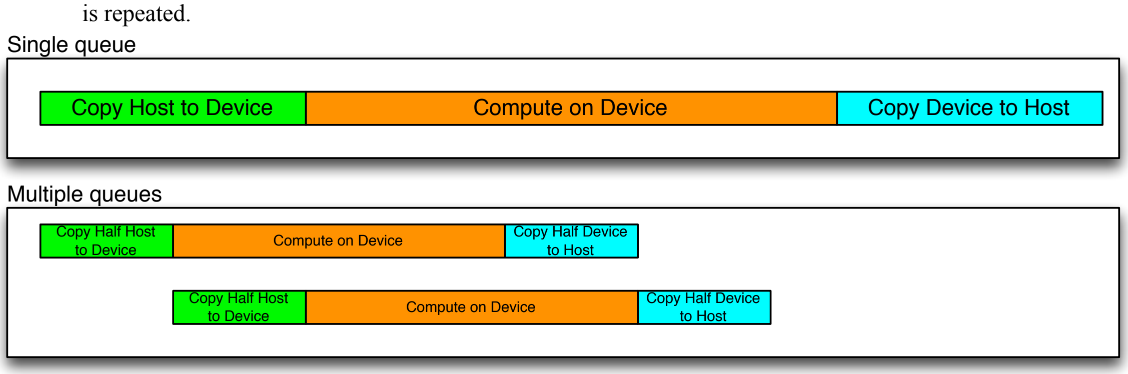

Figure 6 – Using multiple command-queues for overlapped data transfer.

5.1.2.6 Command-queue execution

Once a set of commands have been queued, WebCL offers two ways to execute the command-queue:

```
// execute a task
queue.enqueueTask(kernel);

// execute a NDRange
```

```
queue.enqueueNDRange(kernel, offsets, globals, locals);
```

With enqueueTask(), the kernel is executed using a single work-item. This is a very restricted form of enqueueNDRange().

enqueueNDRange() has first parameters:

- kernel – the kernel to execute
- offsets – offsets to apply to globals. If null, then offsets=[0, 0, 0]
- globals – the problem size per dimension
- locals – the number of work-items per work-group per dimension. If null, the device will choose the appropriate number of work-items

Recall Figure 3 where globals and locals relationships are depicted. If we want to execute a kernel over an image of size (width, height), then globals may be [width, height] and locals may be [16, 16].

Since enqueueNDRange() will fail if locals size is more than cl.KERNEL_WORK_GROUP_SIZE, in practice, it may be useful to do

```
locals[0] = kernel.getWorkGroupInfo(device, cl.KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE);
locals[1] = kernel.getWorkGroupInfo(device, cl.KERNEL_WORK_GROUP_SIZE) / locals[0];
globals[0] = locals[0] * divUp(width, locals[0]);
globals[1] = locals[1] * divUp(height, locals[1]);

// Helper to get next up value for integer division of x/y
function divUp(x, y) {
  return (x % y == 0) ? (x / y) : (x / y + 1);
}
```

Code 10 – A way to optimally setup locals and globals NDRanges.

5.1.2.7 *Command Synchronization*

Nearly all commands available in WebCLCommandQueue class have two final parameters:

- event_list – an array of WebCLEvents
- event – an event returned by the device to monitor the execution status of a command

By default, event_list and event are null for a command, meaning that the command is executed as blocking the host thread until it is queued in the device's command queue. If a programmer doesn't want to block the host thread while a command is being executed, the device can return an event and the host code can register a callback to be notified once the command complete.

```
// Enqueue kernel
try {
  kernel_event=new cl.WebCLEvent();
  queue.enqueueTask(kernel, null, kernel_event);
} catch(ex) {
  throw "Couldn't enqueue the kernel. "+ex;
}

// Set kernel event handling routines
try {
  kernel_event.setCallback(cl.COMPLETE, kernel_complete, "The kernel finished successfully.");
} catch(ex) {
  throw "Couldn't set callback for event. "+ex;
}

// Read the buffer
var data=new Float32Array(4096);
try {
  read_event=new cl.WebCLEvent();
  queue.enqueueReadBuffer(clBuffer, false, 0, 4096*4, data, null, read_event);
} catch(ex) {
  throw "Couldn't read the buffer. "+ex;
}

// register a callback on completion of read_event
read_event.setCallback(cl.COMPLETE, read_complete, “Read complete”);

// wait for both events to complete
queue.waitForEvents([kernel_event, read_event]);

// kernel callback
function kernel_complete(event, data) {
```

```
33     // event.status = cl.COMPLETE or error if negative
34     // event.data is null
35     // data should contain "The kernel finished successfully."
36   }
37
38   // read buffer callback
39   function read_complete(event, data) {
40     // event.status = cl.COMPLETE or error if negative
41     // event.data contains a WebCLMemoryObject with values from device
42     // data contains "Read complete"
43   }
```

Code 11 –Using WebCLEvent callbacks.

In Code 11, for the commands we wish to get notified on their cl.COMPLETE status, we first create a WebCLEvent object, pass it to the command, then register a JavaScript callback function.

Note 1: the last argument of WebCLEvent.setCallback() can be anything. And this argument is passed untouched as the last argument of the callback function.

Note 2: in the case of enqueue Read/Write WebCLBuffers or WebCLImages, as in line 22, clBuffer ownership is transferred from host to device. Thus, when read_complete() callback is called, clBuffer ownership is transferred back from device to host. This means that once the ownership of clBuffer is transferred (line 22), the host cannot access or use this buffer any more. Once the callback is called, line 40, the host can use the buffer again.

5.1.2.8 Profiling commands

To enable timing of commands, one creates a command-queue with option cl.QUEUE_PROFILING_ENABLE. Then, WebCLEvents can be used to time a command. Code 12 shows how to profile an enqueueReadBuffer() command.

```
// Create a command queue for profiling
try {
  queue = context.createCommandQueue(device, cl.QUEUE_PROFILING_ENABLE);
} catch(ex) {
  throw "Couldn't create a command queue for profiling. "+ex;
}

// Read the buffer with a profiling event
var prof_event=new cl.WebCLEvent();
try {
  queue.enqueueReadBuffer(data_buffer, true, 0, data.byteLength, data, null, prof_event);
} catch(ex) {
  throw "Couldn't read the buffer. "+ex;
}

// Get profiling information in nanoseconds
time_start = prof_event.getProfilingInfo(cl.PROFILING_COMMAND_START);
time_end = prof_event.getProfilingInfo(cl.PROFILING_COMMAND_END);
total_time = time_end - time_start;
```

Code 12 – How to profile a command.

Note: timestamps are given in nanoseconds (10^{-9} seconds).

Likewise, to profile the duration of a kernel:

```
// Enqueue kernel
try {
  queue.enqueueNDRangeKernel(kernel, null, globals, locals, null, prof_event);
} catch(ex) {
  throw "Couldn't enqueue the kernel. "+ex;
}
```

Code 13 – Profiling a kernel.

5.2 Device side

Kernels are written in a derivative of C99 with the following caveats:

- A file may have multiple __kernel functions (similar to a library with multiple entry points)
- No recursion since there is no call stack on devices
- All functions are inlined to the kernel functions
- No dynamic memory (e.g. malloc(), free()…)

- No function pointer
- No standard libc libraries (e.g. memcpy(), strcmp()…)
- No standard data structures (except vector operations)
- Helper functions
    - Barriers
    - Work-item functions
    - Atomic operations
    - Vector operations
    - Math operations and fast native (hardware accelerated) math operations
    - IEEE754 floating-point
    - 16-bit floats and doubles (optional)
- Built-in data types
    - 8, 16, 32, 64-bit values
    - Image 2D (and 3D but not in WebCL 1.0), Sampler, Event
    - 2, 3, 4, 8, 16-component vectors
- New keywords
    - Function qualifiers: __kernel
    - Address space qualifiers: __global, __local, __constant, __private (default),
    - Access qualifiers: __read_only, __write_only, __read_write,
- Preprocessor directives (#define, #pragma)

Appendices A.3 and A.4 provide examples of kernels.

6 Interoperability with WebGL

Recall that WebCL is for computing, not for rendering. However, if your data already resides in the GPU and you need to render it, wouldn't it be faster to tell OpenGL to use it rather than reading it from the GPU memory to CPU memory and send it again to OpenGL on your GPU? This is where WebGL comes in.

Since WebCL is using data from WebGL, the WebGL context must be created first. Then, a shared WebCL context can be created. This GL share group object manages shared GL and CL resources such as

- Textures objects – contain texture data in image form,
- Vertex buffers objects (VBOs) – contains vertex data such as coordinates, colors, and normal vectors,
- Renderbuffer objects – contain images used with GL framebuffer objects.

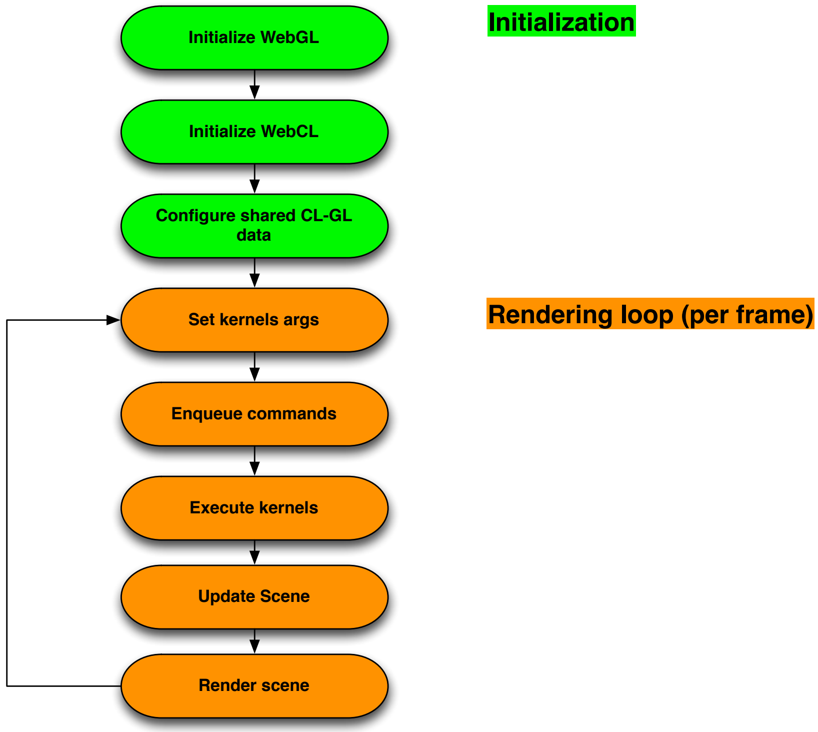

Figure 7 – Typical algorithm for WebCL – WebGL applications

6.1 Fun with 2 triangles

Applications such as image processing and ray tracing produce an output image whose pixels are drawn onto the screen. For such applications, it suffices to map the output image onto 2 unlit screen-aligned triangles rendered by GL. A compute kernel provides more flexible ways to optimize generic computations than a fragment shader. More importantly, texture memory is cached and thus provides a faster way to access data than regular (global) memory. However, in devices without image memory support, one should use WebCLBuffers and update GL textures with Pixel Buffer Objects.

In this section, we use Iñigo Quilez excellent ShaderToy's Mandelbulb fragment shader [24] converted as a CL kernel, depicted in Figure 8. The whole WebGL scene consists in 2 textured triangles filling a canvas. WebCL generates the texture at each frame. Therefore, for a canvas of dimension [width, height], WebCL will generate width * height pixels. We will detail each step and the full program is given in Appendix A. In [24], you can find more cool shaders that you can easily convert by the following the guidelines for this sample.

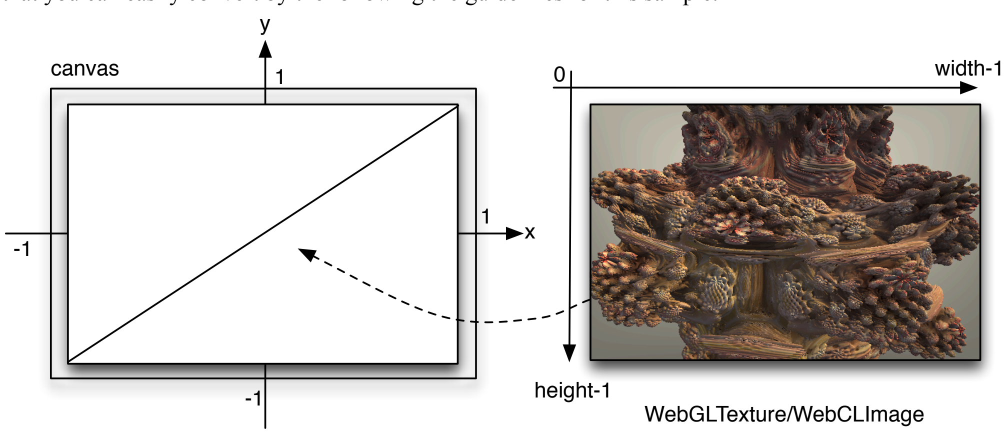

Figure 8 – Two triangles filling the canvas to draw a WebCL generated image.

6.1.1 General CL-GL interoperability algorithm

Since CL uses GL buffers for compute, WebGL context must first be initialized and then WebCL context is created sharing that WebGL context. Once both contexts are initialized, it is possible to create shared objects by creating first the WebGL object, then the corresponding WebCL object from the WebGL object.

The general algorithm is as follows:

```
function Init_GL() {
  // Create WebGL context
  // Init GL shaders
  // Init GL buffers
  // Init GL textures
}

function Init_CL() {
  // Create WebCL context from WebGLContext
  // Compile programs/kernels
  // Create command queues
  // Create buffers
}

function Create_shared_CLGL_objects {
  // Create WebGL object glObj (vertex array, texture, renderbuffer)
  // Create WebCL object clObj from WebGL object glObj
}

// called during rendering, possibly at each frame
// or in a separate Web Worker
function Execute_kernel(…) {
  // Make sure all GL commands are finished
  gl.flush();

  // acquire shared WebCL object
  queue.enqueueAcquireGLObjects(clObj);

  // Execute CL kernel
  // set global and local parameters
  try {
    queue.enqueueNDRangeKernel(kernel, null, global, local);
  } catch (err) {
    throw "Failed to enqueue kernel! " + err;
}

  // Release CL object
  queue.enqueueReleaseGLObjects(clObj);

  // make sure all CL commands are finished
  queue.flush();
}

// This is the main rendering method called at
// each frame
function display(timestamp) {
  // Execute some GL commands
  …

  Execute_kernel( … );

  // Execute more CL and GL commands
  …
}
```

Code 14 – General algorithm for WebCL-WebGL interoperability.

The remainder of this section will focus on how to create shared CLGL objects and how to use them.

6.1.2 Using shared textures

Initialize a WebCLImage object from a WebGLImage object as follows:

```
// Create OpenGL texture object
Texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, Texture);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, TextureWidth, TextureHeight, 0, gl.RGBA,
gl.UNSIGNED_BYTE, null);
gl.bindTexture(gl.TEXTURE_2D, null);

// Create OpenCL representation of OpenGL texture
try {
  clTexture = ctx.createFromGLTexture2D(cl.MEM_WRITE_ONLY, gl.TEXTURE_2D, 0, Texture);
}
catch(ex) {
  throw "Error: Failed to create WebCLImage. "+ex;
}

// To use this texture, somewhere in your code, do as usual:
glBindTexture(gl.TEXTURE_2D, Texture)
```

Code 15 – Initialize a WebCLImage object from a WebGLImage object.

Set the WebCLImage as an argument of your kernel:

```
kernel.setArg(0, clTexture);
kernel.setArg(1, TextureWidth, cl.type.UINT);
kernel.setArg(2, TextureHeight, cl.type.UINT);
```

Finally, here is how to use this WebCLImage inside your kernel code:

```
__kernel
void compute(__write_only image2d_t pix, uint width, uint height)
{
  const int x = get_global_id(0);
  const int y = get_global_id(1);

  // compute pixel color as a float4

  write_imagef(pix, (int2)(x,y), color);
}
```

Code 16 – Using a WebCLImage data inside a kernel.

Note: it should be possible to use write_imagei() or write_imageui() with int4 colors. However, at the time of writing (May 2012), this doesn't seem to work with latest AMD and NVidia drivers. The code presented in this section is the only way I found to work with textures between CL and GL.

6.1.3 Using shared buffers

A WebCLBuffer is created from a WebGLBuffer as follows. On line 6, it is important to specify the correct sizeInBytes of the buffer.

```
1   // create a WebGLBuffer
2   pbo = gl.createBuffer();
3   gl.bindBuffer(gl.ARRAY_BUFFER, pbo);
4
5   // buffer data
6   gl.bufferData( gl.ARRAY_BUFFER, sizeInBytes, gl.DYNAMIC_DRAW);
7   gl.bindBuffer( gl.ARRAY_BUFFER, null);
8
9   // Create WebCLBuffer from WebGLBuffer
10  try {
11    clPBO = context.createFromGLBuffer( cl.MEM_WRITE_ONLY, pbo);
12  }
13  catch(ex) {
14    throw "Error: Failed to create WebCLBuffer. "+ex;
15  }
```

Code 17 – Using a WebCLImage data inside a kernel.

Since a GL ARRAY_BUFFER can be used for vertices, normals, colors, texture coordinates, texture data, and more, WebCL can be used to schedule processing of these buffers.

If the device doesn't support texture interoperability between CL and GL, a buffer can be used to update a WebGLImage sub-texture as follows with the assumption that WebCLBuffer contains RGBA values for each pixel.

```
// Create OpenGL texture object
Texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, Texture);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
```

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, TextureWidth, TextureHeight, 0, gl.RGBA,
gl.UNSIGNED_BYTE, null);
gl.bindTexture(gl.TEXTURE_2D, null);

// To render using this texture
gl.bindTexture(gl.TEXTURE_2D, TextureId);
gl.bindBuffer(gl.PIXEL_UNPACK_BUFFER, pbo);
gl.texSubImage2D(gl.TEXTURE_2D, 0, 0, 0, TextureWidth, TextureHeight, gl.RGBA, gl.UNSIGNED_BYTE,
null);
```

Code 18 – Updating a texture from a WebGLBuffer

6.1.4 Example

The following example consists of 2 module objects, whose code is given in Appendix A:

- Graphics – encapsulates WebGL calls
- Compute – encapsulates WebCL calls

The code is rather large for just setting up WebCL and WebGL but, fear not, this is just boilerplate you can reuse!

The main method works as follows:

- Create a Canvas object
- Instantiate Graphics and Compute objects
- Launch the main rendering method
    - If the window is resized, we call Graphics to configure the shared GL texture. Then, we call Compute to configure the CL texture from this GL texture.
    - At each frame, we reset the kernel argument with the current timestamp in seconds. Then, the kernel is executed.
    - Finally, Graphics renders the frame

```
var COMPUTE_KERNEL_ID = "mandelbulb.cl"; // <script> id
var COMPUTE_KERNEL_NAME = "compute";     // name of __kernel
var Width;
var Height;
var Reshaped = true;
var log = console.log;

/*
 * reshape() is called if document is resized
 */
function reshape(evt) {
  Width = evt.width;
  Height = evt.height;
  Reshaped = true;
}

(function main() {
  log('Initializing');

  document.setTitle("Mandelbulb demo");
  var canvas = document.createElement("mycanvas", Width, Height);

  // install UX callbacks
  document.addEventListener('resize', reshape);

  // init WebGL
  var gfx=Graphics();
  try {
    gfx.init(canvas);
  }
  catch(err) {
    alert('[Error] While initializing GL: '+err);
    gfx.clean();
    return;
  }

  // init WebCL
  var compute=Compute();
  try {
    compute.init(gfx.gl(), COMPUTE_KERNEL_ID, COMPUTE_KERNEL_NAME);
  }
  catch(err) {
    alert('[Error] While initializing CL: '+err);
```

```
      compute.clean();
      return;
    }

    // render scene
    var startTime=-1;
    var fpsFrame=0, fpsTo=0;

    (function update(timestamp) {
      if(timestamp) {
        if(startTime==-1) {
          startTime=fpsTo=timestamp;
        }
        var ltime = timestamp-startTime;
      }

      // reinit shared data if document is resized
      if (Reshaped) {
        log('reshaping texture');
        try {
          var glTexture=gfx.configure_shared_data(Width,Height);
          var clTexture=compute.configure_shared_data(gfx.gl(), glTexture);
          Reshaped=false;
        }
        catch(err) {
          alert('[Error] While reshaping shared data: '+ex);
          return;
        }
      }

      // set kernel arguments
      compute.resetKernelArgs(ltime/1000.0, Width, Height);

      // compute texture for this timestamp
      try {
        compute.execute_kernel(gfx.gl());
      }
      catch(err) {
        alert('[Error] While executing kernel: '+ex);
        return;
      }

      // render scene with updated texture from CL
      try {
        gfx.display(ltime);
      }
      catch(err) {
        alert('[Error] While rendering scene '+err);
        return;
      }

      // Calculate framerate
      fpsFrame++;
      var dt=timestamp - fpsTo;
      if( dt>1000 ) {
          var ffps = 1000.0 * fpsFrame / dt;
          log("myFramerate: " + ffps.toFixed(1) + " fps");
          fpsFrame = 0;
          fpsTo = timestamp;
      }

      requestAnimationFrame(update);
    })();
  })();
```

Code 19 – Main method for CL-GL program.

The kernel for such applications has the form:

```
__kernel
void compute(__write_only image2d_t pix, const float time)
{
  const int x = get_global_id(0);
  const int y = get_global_id(1);
  const int xl = get_local_id(0);
```

```
7      const int yl = get_local_id(1);
8      const int tid = xl+yl*get_local_size(0); // local work-item ID
9      const int width = get_global_size(0);
10     const int height = get_global_size(1);
11
12     // init local memory
13     …
14     // perform interesting computations for pixel (x,y)
15     …
16     // write (r,g,b,a) value at pixel (x,y)
17     write_imagef(pix, (int2)(x,y), rgba);
18   }
```

Code 20 – Kernel for texture-based rendering.

Note 1: we don't pass the size of the shared texture since the dimension of our problem is the full size of the texture itself. In other words, when executing the kernel with enqueueNDRange(), the globals argument is [width, height], and that's what we retrieve in lines 9 and 10 in Code 20.

Note 2: for this example, we only pass the timestamp of the current frame to the kernel. For user interactivity, one should also pass mouse coordinates, window size, and other user/application attributes.

In Appendix, we provide the fragment shader code of the Mandelbulb shader by Iñigo Quilez [24], as well as the direct transformation to OpenCL and an optimized OpenCL version. We chose this example because the ray-marching algorithm (also known as sphere tracing [25]) used to render the mandelbulb fractal requires lots of operations per pixel; a good candidate for CL optimizations. Note that this is not necessarily the fastest way to render such mathematical objects. On our machine, this leads to 6 fps for WebGL version [24], 8 fps for non-optimized OpenCL version (Appendix A.2), and 12 fps for the optimized OpenCL version (Appendix A.4).

6.2 Other applications

In general, CL applications perform many matrix operations, whether the result is to be rendered directly onto the screen (e.g. in a texture) or not. For example, the famous N-body simulation calculates at each frame the position of astronomical objects, which are then rendered by GL [23]. An array of structures that contains position and other attributes is shared between host and device; the device performing all the calculations of the interactions between objects.

CL can also share vertex buffers and render buffers with GL. This allows developers to do all kind of complex geometry and special effects that can be inserted in GL's rendering pipeline.

7 Tips and tricks

NVidia and AMD excellent programming guides [8][9] provide lots of tips to optimize OpenCL programs. In our experience, we recommend following this strategy:

- Use Host code for serial code, use Device code for parallel code
- Write your code in serialized form (i.e. test it on a host CPU) and identify the areas that are good candidates for data-parallel optimizations
  - As a rule of thumb: identify where iterations are repeated on data, these are good candidates for data-parallel optimizations
- In your kernel, initialize first local memory with data from global memory that will be used often in your algorithm
- Group memory transfers together, this favors memory coalescing
- Identify where synchronization between work-items of the same work-group is necessary
- At the end of your kernel, write results from local memory back to global memory
- Rewrite your algorithm to minimize control flow divergence (i.e. if, switch, for, do, while). If threads in the same warp/wavefront take different execution paths, these execution paths will be serialized, thereby reducing throughput until the execution paths converge again.

7.1 From GLSL to OpenCL C

In converting GLSL shader to OpenCL C, we recommend following these guidelines:

- GLSL's vec*N* type are changed to OpenCL's float*N* type
    - Initializations in OpenCL are: (float*N*)(val1,...,valN) instead of vec*N*(val1,…,valN) in GLSL
- by default all floating point values are double in CL, make sure to add 'f' at the end.
- out arguments of methods must be pointers
- if numerical precision is not too important, compile with –cl-fast-relaxed-math, -cl-mad-enable, and use native_* functions (i.e. native_sin() instead of sin()).
- Use rsqrt() instead of 1.0f/sqrt()

7.2 Barriers

Barriers are an important mechanism to wait for all work-items to be synchronized at points in the code. However, it is very important NOT to use barriers in if/else constructs. The reason is that some work items may not sync at the barrier, while others may block at the barrier; resulting in a deadlock of the GPU (i.e. you would have to reset your machine!).

The pattern to use a barrier is:

- Load values into local memory
- barrier(CL_LOCAL_MEM_FENCE); // wait for load to finish
- Use local memory in your algorithm
- barrier(CL_LOCAL_MEM_FENCE); // wait for all work-items

7.3 Local work-group size

When running a kernel, the method enqueueNDRangeKernel(), takes the parameters:

- global_work_size – the global number of work-items in N dimensions i.e. the size of the problem.
- local_work_size – the number of work-items that make up a work-group. Synchronization between work-items (with barriers) can only be within a work-group.

If local_work_size[0] * local_work_size[1] * … * local_work_size[N-1] > kernel.getWorkGroupInfo(device, cl.KERNEL_WORK_GROUP_SIZE), the program won't execute!

cl.KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE can be used to make block-size multiple of that size. AMD calls that size wavefront size and NVidia calls it warp size. Note: this value is often 32 for NVidia GPUs, 64 for AMD GPUs.

Since kernels can't allocate memory dynamically, one trick could be to compile a small program to get such kernel dependent values, add them on top of your real program code as constants (or #define) before compiling it.

7.4 Learn parallel patterns!

Parallel programming is not new. In fact, it might be as old as modern computers. Since the 60s lots of work has been done on supercomputers and many patterns have been found but they are still an active area of research. Learning how to use these patterns can simplify your code and more importantly lead to faster performance for your programs 0[13][21]. Algorithms such as map, reduce, scan, scatter/gather, stencils, pack [21], Berkely Parallel Computing Laboratory's pattern language for parallel computing [32], and Murray Cole's algorithmic skeletons 0 are examples of such parallel algorithms and methodologies you need to know.

8 WebCL implementations

At the time of writing, the following prototypes are available:

- Nokia WebCL prototype [16] as a Mozilla FireFox extension
- Mozilla FireFox implementation [18]
- Samsung WebKit prototype [17]
- Motorola Mobility node-webcl module [15], a Node.JS based implementation.

Motorola Mobility node-webcl implementation is based on Node.JS, which uses Google V8 JavaScript engine, as in Google Chrome browser. This implementation is up to date with the latest WebCL specification and allows quick prototyping of WebCL features before they become available in browsers. Coupled with Node.JS features, it also enables server-side applications using WebCL. All examples in this course have been developed and tested first with node-webcl.

9 Perspectives

This course provided the foundations for developers to experiment with the exciting world of high-performance computing on the web. OpenCL is a rather young technology and it is not uncommon to find bugs in current implementations. However, WebCL implementations would abstract these technical issues for safer, more robust, more secure, and more portable applications, as the specification mature with feedback from users, hardware manufacturers and browser vendors. Meanwhile, prototype WebCL implementations are already available and we hope this course gave you all the excitement to start hacking your GPUs today for cool applications tomorrow ☺

Appendix A CL-GL code

This appendix provides source code for applications described in section 6.1.4. The first two sections provide the Graphics and Compute module objects. The third section is a direct translation from GLSL to OpenCL kernel language using techniques described in section 7.1. The last section is an example optimized version using local memory and work-groups.

A.1 Graphics object

```
/*
 * Graphics module object contains all WebGL initializations for a simple
 * 2-triangle textured screen aligned scene and its rendering.
 */
function Graphics() {
  var gl;
  var shaderProgram;
  var TextureId = null;
  var VertexPosBuffer, TexCoordsBuffer;

  /*
   * Init WebGL array buffers
   */
  function init_buffers() {
    log('  create buffers');
    var VertexPos = [ -1, -1,
                      1, -1,
                      1, 1,
                      -1, 1 ];
    var TexCoords = [ 0, 0,
                      1, 0,
                      1, 1,
                      0, 1 ];

    VertexPosBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, VertexPosBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(VertexPos), gl.STATIC_DRAW);
    VertexPosBuffer.itemSize = 2;
    VertexPosBuffer.numItems = 4;

    TexCoordsBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, TexCoordsBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(TexCoords), gl.STATIC_DRAW);
    TexCoordsBuffer.itemSize = 2;
    TexCoordsBuffer.numItems = 4;
  }

  /*
   * Compile vertex and fragment shaders
   *
   * @param gl WebGLContext
   * @param id <script> id where the source of the shader resides
   */
  function compile_shader(gl, id) {
    var shaders = {
      "shader-vs" : [
          "attribute vec3 aCoords;",
          "attribute vec2 aTexCoords;",
          "varying vec2 vTexCoords;",
          "void main(void) {",
          "    gl_Position = vec4(aCoords, 1.0);",
          "    vTexCoords = aTexCoords;",
          "}" ].join("\n"),
      "shader-fs" : [
           "#ifdef GL_ES",
           "  precision mediump float;",
           "#endif",
           "varying vec2 vTexCoords;",
           "uniform sampler2D uSampler;",
           "void main(void) {",
```

```
         "    gl_FragColor = texture2D(uSampler, vTexCoords.st);",
         "}" ].join("\n"),
  };

  var shader;
  var str = shaders[id];

  if (id.match(/-fs/)) {
    shader = gl.createShader(gl.FRAGMENT_SHADER);
  } else if (id.match(/-vs/)) {
    shader = gl.createShader(gl.VERTEX_SHADER);
  } else {
    throw 'Shader '+id+' not found';
  }

  gl.shaderSource(shader, str);
  gl.compileShader(shader);

  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    throw gl.getShaderInfoLog(shader);
  }

  return shader;
}

/*
 * Initialize vertex and fragment shaders, link program and scene objects
 */
function init_shaders() {
  log('  Init shaders');
  var fragmentShader = compile_shader(gl, "shader-fs");
  var vertexShader = compile_shader(gl, "shader-vs");

  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS))
    throw "Could not link shaders";

  gl.useProgram(shaderProgram);

  shaderProgram.vertexPositionAttribute = gl.getAttribLocation(shaderProgram, "aCoords");
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);

  shaderProgram.textureCoordAttribute = gl.getAttribLocation(shaderProgram, "aTexCoords");
  gl.enableVertexAttribArray(shaderProgram.textureCoordAttribute);

  shaderProgram.samplerUniform = gl.getUniformLocation(shaderProgram, "uSampler");
}

/*
 * Render the scene at a timestamp.
 *
 * @param timestamp in ms as given by new Date().getTime()
 */
function display(timestamp) {
 // we just draw a screen-aligned texture
 gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);

 gl.enable(gl.TEXTURE_2D);
 gl.bindTexture(gl.TEXTURE_2D, TextureId);

 // draw screen aligned quad
 gl.bindBuffer(gl.ARRAY_BUFFER, VertexPosBuffer);
 gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
     VertexPosBuffer.itemSize, gl.FLOAT, false, 0, 0);

 gl.bindBuffer(gl.ARRAY_BUFFER, TexCoordsBuffer);
 gl.vertexAttribPointer(shaderProgram.textureCoordAttribute,
     TexCoordsBuffer.itemSize, gl.FLOAT, false, 0, 0);

 gl.activeTexture(gl.TEXTURE0);
 gl.uniform1i(shaderProgram.samplerUniform, 0);
```

```

   gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);

   gl.bindTexture(gl.TEXTURE_2D, null);
   gl.disable(gl.TEXTURE_2D);

   gl.flush();
  }

  /*
   * Initialize WebGL
   *
   * @param canvas HTML5 canvas object
   */
  function init(canvas) {
    log('Init GL');
    gl = canvas.getContext("experimental-webgl");
    gl.viewportWidth = canvas.width;
    gl.viewportHeight = canvas.height;

    init_buffers();
    init_shaders();
  }

  /*
   * Configure shared data i.e. our WebGLImage
   *
   * @param TextureWidth width of the shared texture
   * @param TextureHeight height of the shared texture
   */
  function configure_shared_data(TextureWidth, TextureHeight) {
    if (TextureId) {
      gl.deleteTexture(TextureId);
      TextureId = null;
    }

    gl.viewportWidth = TextureWidth;
    gl.viewportHeight = TextureHeight;

    // Create OpenGL texture object
    gl.activeTexture(gl.TEXTURE0);
    TextureId = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, TextureId);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, TextureWidth, TextureHeight, 0,
        gl.RGBA, gl.UNSIGNED_BYTE, null);
    gl.bindTexture(gl.TEXTURE_2D, null);

    return TextureId;
  }

  return {
    'gl': function() { return gl; },
    'TextureId': function() { return TextureId; },
    'configure_shared_data': configure_shared_data,
    'init': init,
    'display': display,
    'clean': function() {}
  };
}
```

A.2 Compute object

The compute object reads a kernel from a string. The string may come from a <script type="x-webcl"> or from a file.

```
/*
 * Compute contains all WebCL initializations and runtime for our kernel
 * that update a texture.
 */
function Compute() {
  var cl=new WebCL();
  var /* cl_context */        clContext;
```

```
var /* cl_command_queue */  clQueue;
var /* cl_program */        clProgram;
var /* cl_device_id */      clDevice;
var /* cl_device_type */    clDeviceType = cl.DEVICE_TYPE_GPU;
var /* cl_image */          clTexture;
var /* cl_kernel */         clKernel;
var max_workgroup_size, max_workitem_sizes, warp_size;
var TextureWidth, TextureHeight;
var COMPUTE_KERNEL_ID;
var COMPUTE_KERNEL_NAME;
var nodejs = (typeof window === 'undefined');

/*
 * Initialize WebCL context sharing WebGL context
 *
 * @param gl WebGLContext
 * @param kernel_id the <script> id of the kernel source code
 * @param kernel_name name of the __kernel method
 */
function init(gl, kernel_id, kernel_name) {
  log('init CL');
  if(gl === 'undefined' || kernel_id === 'undefined'
    || kernel_name === 'undefined')
    throw 'Expecting init(gl, kernel_id, kernel_name)';

  COMPUTE_KERNEL_ID = kernel_id;
  COMPUTE_KERNEL_NAME = kernel_name;

  // Pick platform
  var platformList = cl.getPlatforms();
  var platform = platformList[0];

  // create the OpenCL context
  clContext = cl.createContext({
    deviceType: clDeviceType,
    shareGroup: gl,
    platform: platform });

  var device_ids = clContext.getInfo(cl.CONTEXT_DEVICES);
  if (!device_ids) {
    throw "Error: Failed to retrieve compute devices for context!";
  }

  var device_found = false;
  for(var i=0,l=device_ids.length;i<l;++i ) {
    device_type = device_ids[i].getInfo(cl.DEVICE_TYPE);
    if (device_type == clDeviceType) {
      clDevice = device_ids[i];
      device_found = true;
      break;
    }
  }

  if (!device_found)
    throw "Error: Failed to locate compute device!";

  // Create a command queue
  try {
    clQueue = clContext.createCommandQueue(clDevice, 0);
  }
  catch(ex) {
    throw "Error: Failed to create a command queue! "+ex;
  }

  // Report the device vendor and device name
  var vendor_name = clDevice.getInfo(cl.DEVICE_VENDOR);
  var device_name = clDevice.getInfo(cl.DEVICE_NAME);
  log("Connecting to " + vendor_name + " " + device_name);

  if (!clDevice.getInfo(cl.DEVICE_IMAGE_SUPPORT))
    throw "Application requires images: Images not supported on this device.";

  init_cl_buffers();
  init_cl_kernels();
}
```

```
  /*
   * Initialize WebCL kernels
   */
  function init_cl_kernels() {
    log('  setup CL kernel');

    clProgram = null;

    if(!nodejs) {
      var sourceScript = document.getElementById(COMPUTE_KERNEL_ID);
      if (!sourceScript)
        throw "Can't find CL source <script>";

      var str = "";
      var k = sourceScript.firstChild;
      while (k) {
        if (k.nodeType == 3) {
          str += k.textContent;
        }
        k = k.nextSibling;
      }
      if (sourceScript.type == "x-webcl")
        source = str;
      else
        throw "<script> type should be x-webcl";
    }
    else {
      log("Loading kernel source from file '" + COMPUTE_KERNEL_ID + "'...");
      source = fs.readFileSync(__dirname + '/' + COMPUTE_KERNEL_ID, 'ascii');
      if (!source)
        throw "Error: Failed to load kernel source!";
    }


    // Create the compute program from the source buffer
    try {
      clProgram = clContext.createProgram(source);
    }
    catch(ex) {
      throw "Error: Failed to create compute program! "+ex;
    }

    // Build the program executable
    try {
      clProgram.build(clDevice, '-cl-unsafe-math-optimizations -cl-single-precision-constant -cl-fast-relaxed-math -cl-mad-enable');
    } catch (err) {
      throw "Error: Failed to build program executable!\n"
          + clProgram.getBuildInfo(clDevice, cl.PROGRAM_BUILD_LOG);
    }

    // Create the compute kernels from within the program
    try {
      clKernel = clProgram.createKernel(COMPUTE_KERNEL_NAME);
    }
    catch(ex) {
      throw "Error: Failed to create compute row kernel! "+ex;
    }

    // Get the device intrinsics for executing the kernel on the device
    max_workgroup_size = clKernel.getWorkGroupInfo(clDevice, cl.KERNEL_WORK_GROUP_SIZE);
    max_workitem_sizes=clDevice.getInfo(cl.DEVICE_MAX_WORK_ITEM_SIZES);
    warp_size=clKernel.getWorkGroupInfo(clDevice, cl.KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE);
    log('  max workgroup size: '+max_workgroup_size);
    log('  max workitem sizes: '+max_workitem_sizes);
    log('  warp size: '+warp_size);
  }

  /*
   * (Re-)set kernel arguments
   *
   * @param time timestamp in ms (as given by new Date().getTime()
   * @param image_width width of the image
   * @param image_height height of the image
```

```
   */
  function resetKernelArgs(time, image_width, image_height) {
    TextureWidth = image_width;
    TextureHeight = image_height;

    // set the kernel args
    try {
      // Set the Argument values for the row kernel
      clKernel.setArg(0, clTexture);
      clKernel.setArg(1, time, cl.type.FLOAT);
    } catch (err) {
      throw "Failed to set row kernel args! " + err;
    }
  }

  /*
   * Initialize WebCL buffers
   */
  function init_cl_buffers() {
    //log('  create CL buffers');
  }

  /*
   * Configure shared data with WebGL i.e. our texture
   *
   * @param gl WebGLContext
   * @param glTexture WebGLTexture to share with WebCL
   */
  function configure_shared_data(gl, glTexture) {
    // Create OpenCL representation of OpenGL Texture
    clTexture = null;
    try {
      clTexture = clContext.createFromGLTexture2D(cl.MEM_WRITE_ONLY,
          gl.TEXTURE_2D, 0, glTexture);
    } catch (ex) {
      throw "Error: Failed to create CL Texture object. " + ex;
    }

    return clTexture;
  }

  /*
   * Execute kernel possibly at each frame before rendering results with WebGL
   *
   * @param gl WebGLContext
   */
  function execute_kernel(gl) {
    // Sync GL and acquire buffer from GL
    gl.flush();
    clQueue.enqueueAcquireGLObjects(clTexture);

    // Set global and local work sizes for kernel
    var local = [];
    local[0] = warp_size;
    local[1] = max_workgroup_size / local[0];
    var global = [ clu.DivUp(TextureWidth, local[0]) * local[0],
                   clu.DivUp(TextureHeight, local[1]) * local[1] ];

    // default values
    //var local = null;
    //var global = [ TextureWidth, TextureHeight ];

    try {
      clQueue.enqueueNDRangeKernel(clKernel, null, global, local);
    } catch (err) {
      throw "Failed to enqueue kernel! " + err;
    }

    // Release GL texture
    clQueue.enqueueReleaseGLObjects(clTexture);
    clQueue.flush();
  }

  return {
    'init':init,
```

```
    'configure_shared_data': configure_shared_data,
    'resetKernelArgs': resetKernelArgs,
    'execute_kernel': execute_kernel,
    'clean': function() {}
  }
}
```

A.3 Mandelbulb kernel (direct conversion)

The Mandelbulb 3D fractal, raymarched and colored with orbit traps and fake ambient occlusion by Iñigo Quilez [24] with authorization, is converted directly to an OpenCL kernel.

```
// forward declarations
bool isphere( float4 sph, float3 ro, float3 rd, float2 *t );
bool iterate( float3 q, float *resPot, float4 *resColor );
bool ifractal( float3 ro, float3 rd, float *rest, float maxt, float3 *resnor, float4 *rescol,
float fov );

inline bool isphere( float4 sph, float3 ro, float3 rd, float2 *t ) {
  float3 oc = ro - sph.xyz;
  float b = dot(oc,rd);
  float c = dot(oc,oc) - sph.w*sph.w;

  float h = b*b - c;
  if( h<0 )
    return false;

  float g = sqrt( h );
  t->x = - b - g;
  t->y = - b + g;

  return true;
}

#define NumIte 7
#define Bailout 100

inline bool iterate( float3 q, float *resPot, float4 *resColor ) {
  float4 trap = (float4)(100);
  float3 zz = q;
  float m = dot(zz,zz);
  if( m > Bailout ) {
    *resPot = 0.5f*log(m); ///pow(8.0f,0.0f);
    *resColor = (float4)(1);
    return false;
  }

  for( int i=1; i<NumIte; i++ ) {
    float x = zz.x; float x2 = x*x; float x4 = x2*x2;
    float y = zz.y; float y2 = y*y; float y4 = y2*y2;
    float z = zz.z; float z2 = z*z; float z4 = z2*z2;

    float k3 = x2 + z2;
    float k2 = rsqrt( k3*k3*k3*k3*k3*k3*k3 );
    float k1 = x4 + y4 + z4 - 6*y2*z2 - 6*x2*y2 + 2*z2*x2;
    float k4 = x2 - y2 + z2;

    zz.x = q.x + 64*x*y*z*(x2-z2)*k4*(x4-6.0*x2*z2+z4)*k1*k2;
    zz.y = q.y + -16*y2*k3*k4*k4 + k1*k1;
    zz.z = q.z + -8*y*k4*(x4*x4 - 28*x4*x2*z2 + 70*x4*z4 - 28*x2*z2*z4 + z4*z4)*k1*k2;

    m = dot(zz,zz);

    trap = min( trap, (float4)(zz.xyz*zz.xyz,m) );

    if( m > Bailout )
    {
      *resColor = trap;
      *resPot = 0.5f*log(m)/pow(8.0f,i);
      return false;
    }
  }

```

```
  *resColor = trap;
  *resPot = 0;
  return true;
}

inline bool ifractal( float3 ro, float3 rd, float *rest, float maxt,
    float3 *resnor, float4 *rescol, float fov ) {
  float4 sph = (float4)( 0.0, 0.0, 0.0, 1.25 );
  float2 dis;

  // bounding sphere
  if( !isphere(sph,ro,rd,&dis) )
  return false;

  // early skip
  if( dis.y<0.001f ) return false;

  // clip to near!
  if( dis.x<0.001f ) dis.x = 0.001f;

  if( dis.y>maxt) dis.y = maxt;

  float dt;
  float3 gra;
  float4 color;
  float4 col2;
  float pot1;
  float pot2, pot3, pot4;

  float fovfactor = 1.0f/sqrt(1+fov*fov);

  // raymarch!
  for( float t=dis.x; t<dis.y; ) {
    float3 p = ro + rd*t;

    float Surface = clamp( 0.001f*t*fovfactor, 0.000001f, 0.005f );

    float eps = Surface*0.1f;

    if( iterate(p,&pot1,&color) ) {
      *rest = t;
      *resnor=normalize(gra);
      *rescol = color;
      return true;
    }

    iterate(p+(float3)(eps,0.0,0.0),&pot2,&col2);
    iterate(p+(float3)(0.0,eps,0.0),&pot3,&col2);
    iterate(p+(float3)(0.0,0.0,eps),&pot4,&col2);

    gra = (float3)( pot2-pot1, pot3-pot1, pot4-pot1 );
    dt = 0.5f*pot1*eps/length(gra);

    if( dt<Surface ) {
      *rescol = color;
      *resnor = normalize( gra );
      *rest = t;
      return true;
    }

    t+=dt;
  }

  return false;
}

__kernel
void compute(__write_only image2d_t pix, float time) {
  int x=get_global_id(0), y=get_global_id(1);
  const int width = get_global_size(0);
  const int height = get_global_size(1);
  float2 resolution=(float2)(width,height);
  float2 gl_FragCoord=(float2)(x,y);

  float2 p = (float2)(-1.f + 2.f * gl_FragCoord.xy / resolution.xy);
```

```
  float2 s = p*(float2)(1.33,1.0);

  float3 light1 = (float3)( 0.577f, 0.577f, 0.577f );
  float3 light2 = (float3)( -0.707f, 0, 0.707f );

  float fov = 1;
  float r = 1.4f+0.2f*cospi(2.f*time/20.f);
  float3 campos = (float3)( r*sinpi(2.f*time/20.f),
                            0.3f-0.4f*sinpi(2.f*time/20.f),
                            r*cospi(2.f*time/20.f) );
  float3 camtar = (float3)(0,0.1,0);

  //camera matrix
  float3 cw = normalize(camtar-campos);
  float3 cp = (float3)(0,1,0);
  float3 cu = normalize(cross(cw,cp));
  float3 cv = normalize(cross(cu,cw));

  // ray dir
  float3 rd;
  float3 nor, rgb;
  float4 col;
  float t;

  rd = normalize( s.x*cu + s.y*cv + 1.5f*cw );

  bool res=ifractal(campos,rd,&t,1e20f,&nor,&col,fov);

  if( !res ) {
    rgb = 1.3f*(float3)(1,.98,0.9)*(0.7f+0.3f*rd.y);
  }
  else {
    float3 xyz = campos + t*rd;

    // sun light
    float dif1 = clamp( 0.2f + 0.8f*dot( light1, nor ), 0.f, 1.f );
    dif1=dif1*dif1;

    // back light
    float dif2 = clamp( 0.3f + 0.7f*dot( light2, nor ), 0.f, 1.f );

    // ambient occlusion
    float ao = clamp(1.25f*col.w-.4f,0.f,1.f);
    ao=0.5f*ao*(ao+1);

    // shadow
    if( dif1>0.001f ) {
      float lt1;
      float3 ln;
      float4 lc;
      if( ifractal(xyz,light1,&lt1,1e20,&ln,&lc,fov) )
        dif1 = 0.1f;
    }

    // material color
    rgb = (float3)(1);
    rgb = mix( rgb, (float3)(0.8,0.6,0.2), (float3)(sqrt(col.x)*1.25f) );
    rgb = mix( rgb, (float3)(0.8,0.3,0.3), (float3)(sqrt(col.y)*1.25f) );
    rgb = mix( rgb, (float3)(0.7,0.4,0.3), (float3)(sqrt(col.z)*1.25f) );

    // lighting
    rgb *= (0.5f+0.5f*nor.y)*
           (float3)(.14,.15,.16)*0.8f +
           dif1*( float3)(1.0,.85,.4) +
           0.5f*dif2*( float3)(.08,.10,.14);
    rgb *= (float3)( pow(ao,0.8f), pow(ao,1.00f), pow(ao,1.1f) );

    // gamma
    rgb = 1.5f*(rgb*0.15f + 0.85f*sqrt(rgb));
  }

  float2 uv = 0.5f*(p+1.f);
  rgb *= 0.7f + 0.3f*16.0f*uv.x*uv.y*(1.0f-uv.x)*(1.0f-uv.y);
  rgb = clamp( rgb, (float3)(0), (float3)(1) );
```

```
  write_imagef(pix,(int2)(x,y),(float4)(rgb,1.0f));
}
```

A.4 Mandelbulb kernel (optimized)

The main idea is to move to local memory all parameters necessary for computation.

```
#define WARPSIZE 256

typedef struct {
  float3 origin;
  float r;
  float2 dis;
} Sphere;

typedef struct {
  float3 origin;
  float3 dir;
  float3 nor;
  float4 col;
  float fovfactor;
  float t;
  float3 rgb;
  Sphere sph;
} __attribute__((aligned(16))) Ray;


// forward declarations
bool isphere( __local Ray *ray );
bool iterate( const float3 q, float *resPot, float4 *resColor );
bool ifractal( __local Ray *ray);

inline bool isphere( __local Ray *ray ) {
  const float3 oc = ray->origin - ray->sph.origin;
  const float b = dot(oc,ray->dir);
  const float c = dot(oc,oc) - ray->sph.r*ray->sph.r;

  const float h = b*b - c;
  if( h<0 )
    return false;

  const float g = native_sqrt( h );
  ray->sph.dis = (float2) ( - b - g, - b + g);

  return true;
}

__constant int NumIte=8;
__constant float Bailout=100;
__constant float EPS=0.001f;
__constant float MAXT=1e20f;
__constant float3 light1 = (float3)( 0.577f, 0.577f, 0.577f );
__constant float3 light2 = (float3)( -0.707f, 0, 0.707f );

inline bool iterate( const float3 q, float *resPot, float4 *resColor )
{
  float4 trap = (float4)(100);
  float3 zz = q;
  float m = dot(zz,zz);
  if( m > Bailout ) {
    *resPot = 0.5f*native_log(m); // /pow(8.0f,0.0f);
    *resColor = (float4)(1);
    return false;
  }

#pragma unroll 4
  for( int i=0; i<NumIte; i++ ) {
    const float x = zz.x; const float x2 = x*x; const float x4 = x2*x2;
    const float y = zz.y; const float y2 = y*y; const float y4 = y2*y2;
    const float z = zz.z; const float z2 = z*z; const float z4 = z2*z2;

    const float k3 = x2 + z2;
    const float k2 = rsqrt( k3*k3*k3*k3*k3*k3*k3 );
```

```
    const float k1 = x4 + y4 + z4 - 6*y2*z2 - 6*x2*y2 + 2*z2*x2;
    const float k4 = x2 - y2 + z2;

    zz.x = q.x + 64*x*y*z*(x2-z2)*k4*(x4-6.0*x2*z2+z4)*k1*k2;
    zz.y = q.y + -16*y2*k3*k4*k4 + k1*k1;
    zz.z = q.z + -8*y*k4*(x4*x4 - 28*x4*x2*z2 + 70*x4*z4 - 28*x2*z2*z4 + z4*z4)*k1*k2;

    m = dot(zz,zz);

    trap = min( trap, (float3)(zz.xyz*zz.xyz,m) );

    if( m > Bailout ) {
      *resColor = trap;
      *resPot = 0.5f*native_log(m)/native_powr(8.0f,i);
      return false;
    }
  }

  *resColor = trap;
  *resPot = 0;
  return true;
}

inline bool ifractal( __local Ray *ray) {
  __local Sphere *sph=&ray->sph;
  sph->origin = (float3)( 0);
  sph->r = 1.25f;

  // bounding sphere
  if( !isphere(ray) )
    return false;

  // early skip
  if( sph->dis.y<EPS ) return false;

  // clip to near!
  if( sph->dis.x<EPS ) sph->dis.x = EPS;

  if( sph->dis.y>MAXT) sph->dis.y = MAXT;

  float dt;
  float3 gra;
  float4 color, col2;
  float pot1, pot2, pot3, pot4;

  // raymarch!
  float t=sph->dis.x, Surface, eps;
  float3 p = ray->origin + ray->dir * t;

  while(t < sph->dis.y) {
    if( iterate(p,&pot1,&color) ) {
      ray->t = t;
      ray->nor = fast_normalize(gra);
      ray->col = color;
      return true;
    }

    Surface = clamp( EPS*t*ray->fovfactor, 0.000001f, 0.005f );
    eps = Surface*0.1f;

    iterate(p+(float3)(eps,0.0,0.0),&pot2,&col2);
    iterate(p+(float3)(0.0,eps,0.0),&pot3,&col2);
    iterate(p+(float3)(0.0,0.0,eps),&pot4,&col2);

    gra = (float3)( pot2-pot1, pot3-pot1, pot4-pot1 );
    dt = 0.5f*pot1*eps/fast_length(gra);

    if( dt<Surface ) {
      ray->col = color;
      ray->nor = fast_normalize( gra );
      ray->t = t;
      return true;
    }

    t += dt;
```

```
    p += ray->dir * dt;
  }

  return false;
}

__kernel
void compute(__write_only image2d_t pix, const float time) {
  const int x = get_global_id(0);
  const int y = get_global_id(1);
  const int xl = get_local_id(0);
  const int yl = get_local_id(1);
  const int tid = xl+yl*get_local_size(0);
  const int width = get_global_size(0)-1;
  const int height = get_global_size(1)-1;

  const float2 resolution = (float2)(width,height);
  const float2 gl_FragCoord = (float2)(x,y);

  const float2 p = (float2)(-1.f + 2.f * gl_FragCoord / resolution);
  const float2 s = p*(float2)(1.33,1.0);

  const float fov = 0.5f, fovfactor = rsqrt(1+fov*fov);

  const float ct=native_cos(2*M_PI_F*time/20.f), st=native_sin(2*M_PI_F*time/20.f);
  const float r = 1.4f+0.2f*ct;
  const float3 campos = (float3)( r*st, 0.3f-0.4f*st, r*ct );
  const float3 camtar = (float3)(0,0.1,0);

  //camera matrix
  const float3 cw = fast_normalize(camtar-campos);
  const float3 cp = (float3)(0,1,0);
  const float3 cu = fast_normalize(cross(cw,cp));
  const float3 cv = fast_normalize(cross(cu,cw));

  // ray
  __local Ray rays[WARPSIZE+1],*ray=rays+tid;
  ray->origin=campos;
  ray->dir = fast_normalize( s.x*cu + s.y*cv + 1.5f*cw );
  ray->fovfactor = fovfactor;

  barrier(CLK_LOCAL_MEM_FENCE);

  const bool res=ifractal(ray);

  if( !res ) {
    // background color
    ray->rgb = 1.3f*(float3)(1,0.98,0.9)*(0.7f+0.3f*ray->dir.y);
  }
  else {
    // intersection point
    const float3 xyz = ray->origin + ray->t * ray->dir;

    // sun light
    float dif1 = clamp( 0.2f + 0.8f*dot( light1, ray->nor ), 0.f, 1.f );
    dif1=dif1*dif1;

    // back light
    const float dif2 = clamp( 0.3f + 0.7f*dot( light2, ray->nor ), 0.f, 1.f );

    // ambient occlusion
    const float aot = clamp(1.25f*ray->col.w-.4f, 0.f, 1.f);
    const float ao=0.5f*aot*(aot+1);

    // shadow: cast a lightray from intersection point
    if( dif1 > EPS ) {
      __local Ray *lray=rays+256;
      lray->origin=xyz;
      lray->dir=light1;
      lray->fovfactor = fovfactor;
      if( ifractal(lray) )
        dif1 = 0.1f;
    }

    // material color
```

```
        ray->rgb = (float3)(1);
        ray->rgb = mix( ray->rgb, (float3)(0.8,0.6,0.2), (float3)(native_sqrt(ray->col.x)*1.25f) );
        ray->rgb = mix( ray->rgb, (float3)(0.8,0.3,0.3), (float3)(native_sqrt(ray->col.y)*1.25f) );
        ray->rgb = mix( ray->rgb, (float3)(0.7,0.4,0.3), (float3)(native_sqrt(ray->col.z)*1.25f) );

        // lighting
        ray->rgb *= (0.5f+0.5f * ray->nor.y)*
                     (float3)(.14,.15,.16)*0.8f +
                     dif1*(float3)(1.0,.85,.4) +
                     0.5f*dif2*(float3)(.08,.10,.14);
        ray->rgb *= (float3)( native_powr(ao,0.8f), native_powr(ao,1.0f), native_powr(ao,1.1f) );

        // gamma
        ray->rgb = 1.5f*(ray->rgb*0.15f + 0.85f*native_sqrt(ray->rgb));
      }

      const float2 uv = 0.5f*(p+1.f);
      ray->rgb *= 0.7f + 0.3f*16.f*uv.x*uv.y*(1.f-uv.x)*(1.f-uv.y);

      ray->rgb = clamp( ray->rgb, (float3)(0), (float3)(1) );

      write_imagef(pix,(int2)(x,y),(float4)(ray->rgb,1.0f));
    }
```

Appendix B OpenCL and CUDA terminology

NVidia provides CUDA, an older API than OpenCL very used on their devices. CUDA and WebCL/OpenCL share similar concepts but a different terminology that we give below, borrowed from AMD article [30] and adapted to WebCL.

Terminology

WebCL/OpenCL	CUDA
Compute Unit (CU)	Streaming Multiprocessor (SM)
Processing Element (PE)	Streaming Processor (SP)
Work-item	Thread
Work-group	Thread block
Global memory	Global memory
Constant memory	Constant memory
Local memory	Shared memory
Private memory	Local memory

Writing kernels: qualifiers

WebCL/OpenCL	CUDA
__kernel function	__global__ function
(no annotation necessary)	__device__ function
__constant variable	__constant__ variable
__global variable	__device__ variable
__local variable	__shared__ variable

Writing kernels: indexing

WebCL/OpenCL	CUDA
get_num_groups()	gridDim
get_local_size()	blockDim
get_group_id()	blockIdx
get_local_id()	threadIdx
get_global_id()	No direct equivalent. Combine blockDim,

	blockIdx, and threadIdx to get a global index.
get_global_size()	No direct equivalent. Combine gridDim and blockDim to get the global size

Writing kernels: synchronization

WebCL/OpenCL	CUDA
barrier()	__syncthreads()
No equivalent	__threadfence()
mem_fence(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM-FENCE0	__threadfence_block()
read_mem_fence()	No equivalent
write_mem_fence()	No equivalent

Important API objects

WebCL/OpenCL	CUDA
WebCLDevice	CUdevice
WebCLContext	CUcontext
WebCLProgram	CUmodule
WebCLKernel	CUfunction
WebCLMemoryObject	CUdeviceptr
WebCLCommandQueue	No equivalent

Important API calls

WebCL/OpenCL	CUDA
No initialization required	cuInit()
WebCLContext.getInfo()	cuDeviceGet()
WebCLContext.create()	cuCtxCreate()
WebCLContext.createCommandQueue()	No equivalent
WebCLProgram.build()	No equivalent. CUDA programs are built off-line
WebCLContext.createKernel()	cuModuleGetFunction()
WebCLCommandQueue.enqueueWriteBuffer()	cuMemcpyHtoD()
WebCLCommandQueue,enqueueReadBuffer()	cuMemcpyDtoH()
Using locals of WebCLCommandQueue.enqueueNDRange()	cuFuncSetBlockShape()
WebCLKernel.setArg()	cuParamSet()
Using WebCLKernel.setArg()	cuParamSetSize()
WebCLCommandQueue.enqueueNDRangeKernel()	cuLaunchGrid()
Implicit through garbage collection	cuMemFree()

Bibliography

Specifications

[1] Aarnio, T. and Bourges-Sevenier, M. WebCL Working Draft. *Khronos WebCL Working Group*. https://cvs.khronos.org/svn/repos/registry/trunk/public/webcl/spec/latest/index.html.

[2] Munshi, A. OpenCL Specification 1.1. *Khronos OpenCL Working Group*. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[3] Marrin, C. WebGL Specification. *Khronos WebGL Working Group*. http://www.khronos.org/registry/webgl/specs/latest/.

[4] Munshi, A. and Leech, J. OpenGL ES 2.0.25. *Khronos Group*. http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf.

[5] Simpson, R.J. The OpenGL ES Shading Language. *Khronos Group*. http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf.

[6] Herman, D. and Russell, K., eds. Typed Array Specification. *Khronos.org*. http://www.khronos.org/registry/typedarray/specs/latest/.

[7] OpenCL 1.1 Reference Pages. OpenCL 1.1 Reference Pages. *Khronos.org*. http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/.

Programming guides

[8] NVidia OpenCL Programming Guide for the CUDA Architecture. 2012. *NVidia OpenCL Programming Guide for the CUDA Architecture*.

[9] AMD Accelerated Parallel Processing OpenCL. 2011. *AMD Accelerated Parallel Processing OpenCL*.

Books

[10] Gaster, B., Howes, L., Kaeli, D.R., Mistry, P., and Schaa, D. 2011. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann.

[11] Scarpino, M. 2011. OpenCL in Action: How to Accelerate Graphics and Computations. Manning Publications.

[12] Munshi, A., Gaster, B., Mattson, T.G., Fung, J., and Ginsburg, D. 2011. *OpenCL Programming Guide*. Addison-Wesley Professional.

[13] Kirk, D. and Hwu, W.-M. 2010. *Programming Massively Parallel Processors*. Morgan Kaufmann.

[14] Hillis, W.D. and Steele, G. 1986. *Data parallel algorithms*.

WebCL prototypes

[15] Motorola Mobility. Node-webcl, an implementation of Khronos WebCL specification using Node.JS. https://github.com/Motorola-Mobility/node-webcl

[16] Nokia Research. WebCL. http://webcl.nokiaresearch.com/

[17] Samsung Research. WebCL prototype for WebKit. http://code.google.com/p/webcl/

[18] Mozilla. FireFox WebCL branch. http://hg.mozilla.org/projects/webcl/

Articles and Presentations

[19] Cole, M.I. 1989. Algorithmic skeletons: structured management of parallel computation

[20] Gerstmann, D. 2009. *Advanced OpenCL*. Siggraph 2009.

[21] McCool, M.D. 2010. Structured parallel programming with deterministic patterns. *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, 5–5.

[22] Bordoloi, U.D. 2010. Optimization Techniques: Image Convolution. 1–25. http://developer.amd.com/zones/openclzone/events/assets/optimizations-imageconvolution1.pdf.

[23] BDT Nbody Tutorial. BDT Nbody Tutorial. *Brown Deer Technology*. http://www.browndeertechnology.com/docs/BDT_OpenCL_Tutorial_NBody-rev3.html.

[24] Iñigo Quilez. ShaderToy with Mandelbulb shader. http://www.iquilezles.org/apps/shadertoy/?p=mandelbulb

[25] Donnelly, W. GPU Gems - Chapter 8. Per-Pixel Displacement Mapping with Distance Functions. *developer.nvidia.com*. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter08.html.

[26] Mattson, T.G., Buck, I., Houston, M., and Gaster, B. 2009. OpenCL - A standard platform for programming heterogeneous parallel computers. *SC'09*. http://www.crc.nd.edu/~rich/SC09/docs/tut149/OpenCL-tut-sc09.pdf.

[27] Feng, W.-C., Lin, H., Scogland, T., and Zhang, J. 2012. OpenCL and the 13 dwarfs: a work in progress.

[28] Lefohn, A., Kniss, J., and Owens, J.D. Chapter 33. Implementing Efficient Parallel Data Structures on GPUs. In: *GPU Gems 2*. Addison-Wesley.

[29] Krüger, J. and Westermann, R. Chapter 44. A GPU Framework for Solving Systems of Linear Equations. In: *GPU Gems 2*. Addison-Wesley.

[30] Porting CUDA Applications to OpenCL. Porting CUDA Applications to OpenCL. *developer.amd.com*. http://developer.amd.com/zones/OpenCLZone/programming/pages/portingcudatoopencl.aspx.

[31] Hensley, J., Gerstmann, D., and Harada, T. OpenCL by Example. *SIGGRAPH Asia 2010*.

[32] A Pattern Language for Parallel Programming. A Pattern Language for Parallel Programming. *parlab.eecs.berkeley.edu*. http://parlab.eecs.berkeley.edu/wiki/patterns/patterns.