The Open Computing Language (OpenCL)

- OpenCL consists of two parts: a C/C++-callable API and a C-ish programming language.
- The OpenCL programming language can run on NVIDIA GPUs, AMD GPUs, Intel CPUs, Intel GPUs, mobile devices, and (supposedly) FPGAs (Field-Programmable Gate Arrays).
- But, OpenCL is at its best on compute devices with large amounts of data parallelism, which usually implies GPU usage.
- You break your computational problem up into lots and lots of small pieces. Each piece gets farmed out to threads on the GPU.
- Each thread wakes up and is able to ask questions about where it lives in the entire collection of (thousands of) threads. From that, it can tell what it is supposed to be working on.
- OpenCL can share data, and interoperate, with OpenGL.
- There is a JavaScript implementation of OpenCL, called WebCL.
- There is a JavaScript implementation of OpenGL, called WebGL.
- WebCL can share data, and interoperate, with WebGL.
- The GPU does not have a stack, and so the OpenCL C-ish programming language cannot do recursion and cannot make function calls. It also can’t use pointers.

The Khronos Group

http://www.khronos.org/opencl/
http://en.wikipedia.org/wiki/OpenCL

Active OpenCL Members

Example of using OpenCL in a System-on-a-Chip:
Qualcomm Node – Full Linux and OpenCL

The OpenCL Paradigm
OpenCL wants you to break the problem up into Pieces

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

If you were writing in OpenCL, you would say:

```c
kernel void ArrayMult( global float *dA, global float *dB, global float *dC)
{
    int gid = get_global_id ( 0 );
}
```

Think of this as having an implied for-loop around it, looping through all possible values of gid

The OpenCL Language also supports Vector Parallelism

OpenCL can be vector-oriented, meaning that it can perform a single instruction on multiple data values at the same time (SIMD).

Vector data types are: char, int, float, where n = 2, 4, 8, or 16.

```c
float4 f, g;
f = (float4)( 1.f, 2.f, 3.f, 4.f );
float16 a16, x16, y16, z16;
f.x = 0.;
f.xy = g.zw;
x16.s89ab = f;
float16 a16 = x16 * y16 + z16;
```

(Note: just because the language supports it, doesn’t mean the hardware does.)
Threads can share memory with the other Threads in the same Work-Group
• Each Work-Group has a maximum number of registers it can use. These are divided equally among all its Threads
• Each Thread has registers and private memory
• Global and Constant memory is often cached inside a Work-Group
• Global and Constant memory is accessible by all Threads in all Work-Groups
• Threads can synchronize with other Threads in the same Work-Group
• Threads can share memory with the other Threads in the same Work-Group

Rules

OpenCL Error Codes

This one is weird. It has no zeros. All the others are negative.

CL_SUCCESS
CL_DEVICE_NOT_FOUND
CL_DEVICE_NOT_AVAILABLE
CL_COMPILER_NOT_AVAILABLE
CL_MEM_SUCCESS
CL_BUILD_PROGRAM_FAILURE
CL_IMAGE_FORMAT_NOT_SUPPORTED
CL_MEM_COPY_OVERLAP
CL_PROFILING_INFO_NOT_AVAILABLE
CL_OUT_OF_HOST_MEMORY
CL_MEM_OBJECT_ALLOCATION_FAILURE
CL_COMPILER_NOT_AVAILABLE
CL_DEVICE_NOT_AVAILABLE
CL_DEVICE_NOT_FOUND
CL_SUCCESS

Getting Just the GPU Device(s)

status = clGetDeviceIDs( platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL );

Status

CL_INVALID_WORK_DIMENSION
CL_INVALID_KERNEL_ARGS
CL_INVALID_ARG_SIZE
CL_INVALID_ARG_VALUE
CL_INVALID_ARG_INDEX
CL_INVALID_KERNEL
CL_INVALID_KERNEL_DEFINITION
CL_INVALID_KERNEL_NAME
CL_INVALID_PROGRAM
CL_INVALID_BUILD_OPTIONS
CL_INVALID_SAMPLER
CL_INVALID_IMAGE_SIZE
CL_INVALID_MEM_OBJECT
CL_INVALID_HOST_PTR
CL_INVALID_COMMAND_QUEUE
CL_INVALID_QUEUE_PROPERTIES

A Way to Print OpenCL Error Codes

struct errorcode
{
    cl_int statusCode;
    char * meaning;
}

Get Used to It:

This way of querying information is a recurring OpenCL pattern (get used to it):

status = clGetPlatformIDs( numPlatforms, platforms, NULL );

How many there are

Querying the Number of Devices on a Platform

status = clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, numDevices, devices, NULL );

Getting Just the GPU Device(s)

status = clGetDeviceIDs( platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL );

Rules

Cl device

status = clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, numDevices, devices, NULL );

void

A Way to Print OpenCL Error Codes – get this from the Class Announcements

status = clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, numDevices, devices, NULL );
Typical Values from Querying the Device

```
Number of Platforms = 1
Platform 0:
  Name = 'NVIDIA CUDA'
  Vendor = 'NVIDIA Corporation'
  Version  = 'OpenCL 1.1 CUDA 4.1.1'
  Profile = 'FULL_PROFILE'

Device 0:
  Type = 0x0004 = CL_DEVICE_TYPE_GPU
  Device Vendor ID = 0x10de
  Device Maximum Compute Units = 15
  Device Maximum Work Item Dimensions = 3
  Device Maximum Work Item Sizes = 1024 x 1024 x 64
  Device Maximum Work Group Size = 1024
  Device Maximum Clock Frequency = 1401 MHz
  Kernel Maximum Work Group Size = 1024
  Kernel Local Memory Size = 0
```

Steps in Creating and Running an OpenCL program

1. Program header
2. Allocate the host memory buffers
3. Create an OpenCL context
4. Create an OpenCL command queue
5. Allocate the device memory buffers
6. Write the data from the host buffers to the device buffers
7. Read the kernel code from a file
8. Compile and link the kernel code
9. Create the kernel object
10. Setup the arguments to the kernel object
11. Enqueue the kernel object for execution
12. Read the results buffer back from the device to the host
13. Clean everything up
```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "cl.h"

// allocate the host memory buffers:
float * hA = new float [ NUM_ELEMENTS ];
float * hB = new float [ NUM_ELEMENTS ];
float * hC = new float [ NUM_ELEMENTS ];

// fill the host memory buffers:
for( int i = 0; i < NUM_ELEMENTS; i++ )
{
    hA[i] = hB[i] = sqrtf( (float) i );
}

// array size in bytes (will need this later):
size_t dataSize = NUM_ELEMENTS * sizeof( float );

// opencl function return status:
cl_int status; // test against CL_SUCCESS

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

// create a command queue:
cl_command_queue cmdQueue = clCreateCommandQueue( context, device, 0, &status );

// allocate memory buffers on the device:
cl_mem dA = clCreateBuffer( context, CL_MEM_READ_ONLY, dataSize, NULL, &status );
cl_mem dB = clCreateBuffer( context, CL_MEM_READ_ONLY, dataSize, NULL, &status );
cl_mem dC = clCreateBuffer( context, CL_MEM_WRITE_ONLY, dataSize, NULL, &status );

// enqueue the 2 commands to write data into the device buffers:
status = clEnqueueWriteBuffer( cmdQueue, dA, CL_FALSE, 0, dataSize, hA, 0, NULL, NULL );
status = clEnqueueWriteBuffer( cmdQueue, dB, CL_FALSE, 0, dataSize, hB, 0, NULL, NULL );
```

**Note:**

- The read and write terminology is with respect to the OpenCL device. So, `CL_MEM_READ_ONLY` means that the OpenCL device can only get this data — it can't send it back to the host CPU. Other options are `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE`.

- Global memory and the heap (typically) have lots more space than the stack does. So, typically, you do not want to allocate a large array like this as a local variable.

- Here, it's being done on the heap. It could also have been done in global memory.
Enqueuing Works Like a Conveyor Belt

Whop-a, whopp-a

- Read Buffer dC
- Execute Kernel
- Write Buffer dB
- Write Buffer dA

The .cl File

```c
kernel
void
ArrayMult( global const float *dA, global const float *dB, global float *dC )
{
    int gid = get_global_id( 0 );
}
```

Which dimension's index are we fetching?

- 0 = X
- 1 = Y
- 2 = Z

Since this is a 1D problem, X is the only index we need to get.

The .cl File

void main()
{
    vec3 newcolor = texture2D( uTexUnit, vST ).rgb;
    newcolor = mix( newcolor, vColor.rgb, uBlend );
    gl_FragColor = vec4(u_LightIntensity*newcolor, 1.);
}

A Warning about Editing on Windows and Running on Linux

Some of you will end up having strange, unexplainable problems with your csh scripts, .cpp programs, or .cl programs. This could be because you are typing your code in on Windows (using Notepad or Wordpad or Word) and then running it on Linux. Windows likes to insert an extra carriage return ('') at the end of each line, which Linux interprets as a garbage character.

You can test this by typing the Linux command:
```
od -c  loop.csh
```
which will show you all the characters, even the '' (which you don't want) and the '
' (newlines, which you do want).

To get rid of the carriage returns, enter the Linux command:
```
tr -d  ''  <  loop.csh  >  loop1.csh
```

Or, on some systems, there is a utility called dos2unix which does this for you:
```
dos2unix < loop.csh > loop1.csh
```

Sorry about this. Unfortunately, this is a fact of life when you mix Windows and Linux.
**Something new: Intermediate Compilation**

- You pre-compile your OpenCL code with an external compiler.
- Your OpenCL code gets turned into an intermediate form known as SPIR-V.
- SPIR-V gets turned into fully-compiled code at runtime.

**Advantages:**
1. Software vendors don’t need to ship their OpenCL source.
2. Syntax errors appear during the SPIR-V step, not during runtime.
3. Software can launch faster because half of the compilation has already taken place.
4. This guarantees a common front-end syntax.
5. This allows for other language front-ends.

---

**How does that array-of-strings thing actually work?**

```c
char *ArrayOfStrings[3];
ArrayOfStrings[0] = …one commonly-used function…
ArrayOfStrings[1] = “. . . another commonly-used function. . . ”;
ArrayOfStrings[2] = “. . . the real OpenCL code . . . ”;
cl_program program = clCreateProgramWithSource( context, 1, (const char **) ArrayOfStrings, NULL, &status );
```

These are two ways to provide a single character buffer:

```c
// the entire OpenCL code ...
cl_program program = clCreateProgramWithSource( context, 1, (const char **) buffer, NULL, &status );
```

---

**Why use an array of strings to hold the OpenCL program, instead of just a single string?**

1. You can use the same OpenCL source and insert the appropriate “#defines” at the beginning.
2. You can insert a common header file (= a .h file).
3. You can simulate a “#include” to re-use common pieces of code.

---

**8. Compile and Link the Kernel Code**

```c
if (create the kernel program on the device:)
if (an array of strings)
strings[0] = clProgramText;
cl_program program = clCreateProgramWithSource( context, 1, (const char **)strings, NULL, &status );
delete [] clProgramText;
if (build the kernel program on the device:)
if (options = “”);
status = clBuildProgram( program, 1, &device, options, NULL, NULL );
if (status != CL_SUCCESS )
{
retrieved and print the error messages:
size_t size;
clGetProgramBuildInfo( program, devices[0], CL_PROGRAM_BUILD_LOG, 0, NULL, &size );
cl_char *log = new cl_char[ size ];
clGetProgramBuildInfo( program, devices[0], CL_PROGRAM_BUILD_LOG, size, log, NULL );
fprintf( stderr, “clBuildProgram failed: %s
”, log );
delete [] log;
}
```

---

**9. Create the Kernel Object**

```c
cl_kernel kernel = clCreateKernel( program, “ArrayMult”, &status );
```

---

**10. Setup the Arguments to the Kernel Object**

```c
status = clSetKernelArg( kernel, 0, sizeof(cl_mem), &dA );
status = clSetKernelArg( kernel, 1, sizeof(cl_mem), &dB );
status = clSetKernelArg( kernel, 2, sizeof(cl_mem), &dC );
```

kernel

void

ArrayMult( global const float *dA, global const float *dB, global float *dC )
11. Enqueue the Kernel Object for Execution

```c
size_t globalWorkSize[3] = { NUM_ELEMENT, 1, 1 };
size_t localWorkSize[3] = { LOCAL_SIZE, 1, 1 };
Wait(cmdQueue); // will be covered in the OpenCL event notes
double time0 = omp_get_wtime();
status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);
Wait(cmdQueue); // will be covered in the OpenCL event notes
double time1 = omp_get_wtime();
```

12. Read the Results Buffer Back from the Device to the Host

```c
status = clEnqueueReadBuffer(cmdQueue, dC, CL_TRUE, 0, dataSize, hC, 0, NULL, NULL);
```
13. Clean Everything Up

```c
// clean everything up:
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
delreleaseMemObject(da);
delreleaseMemObject(db);
delreleaseMemObject(dc);
delete[] hA;
delete[] hB;
delete[] hC;
```

13. Clean Everything Up

Do this because we created these arrays with new:

```c
delreleaseMemObject(dA);
delreleaseMemObject(dB);
delreleaseMemObject(dC);
delete[] hA;
delete[] hB;
delete[] hC;
```

Array Multiplication Performance: What is a Good Work-Group Size?

<table>
<thead>
<tr>
<th>Array Size (K)</th>
<th>GigaMultiplications/Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>145</td>
</tr>
<tr>
<td>16</td>
<td>135</td>
</tr>
<tr>
<td>32</td>
<td>125</td>
</tr>
<tr>
<td>64</td>
<td>115</td>
</tr>
<tr>
<td>128</td>
<td>105</td>
</tr>
<tr>
<td>256</td>
<td>95</td>
</tr>
<tr>
<td>512</td>
<td>85</td>
</tr>
<tr>
<td>1024</td>
<td>75</td>
</tr>
</tbody>
</table>

Writing the .cl Program’s Binary Code

```c
size_t binary_sizes;  
status = clGetProgramInfo(Program, CL_PROGRAM_BINARY_SIZES, 0, NULL, &binary_sizes);  
size_t size;  
status = clGetProgramInfo(Program, CL_PROGRAM_BINARY_SIZES, sizeof(size_t), &size, NULL);  
unsigned char *binary = new unsigned char[size];  
status = clGetProgramInfo(Program, CL_PROGRAM_BINARIES, size, &binary, NULL);  
FILE *fpbin = fopen("particles.nv", "wb");  
if(fpbin == NULL)  
{
    fprintf(stderr, "Cannot create 'particles.bin'");  
}  
else  
{
    fwrite(binary, 1, size, fpbin);
    fclose(fpbin);
}  
delete[] binary;
```

Instead of doing this:

```c
char *strings[1] = {""};  
status = clBuildProgram(program, 1, &device, options, NULL, NULL);  
if(status != CL_SUCCESS)  
{
    size_t size;  
    clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL, &size);  
    cl_char *log = new cl_char[size];  
    clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, size, log, NULL);  
    fprintf(stderr, "clBuildProgram failed:
%s
", log);  
    delete[] log;  
}  
```

8. Compile and Link the Kernel Code

```c
unsigned char byteArray[numBytes];  
cl_program program = clCreateProgramWithBinary(context, 1, &device, &numBytes, byteArray, &binaryStatus, &status);  
delete[] byteArray;  
```

Instead of doing this:

You would do this:

```c
unsigned char binary[(numBytes)];  
cl_program program = clCreateProgramWithBinary(context, 1, &device, &numBytes, &binary, &binaryStatus, &status);  
delete[] binary;
```

And you still have to do this:

```c
char *options = "";  
status = clBuildProgram(program, 1, &device, options, NULL, NULL);  
if(status != CL_SUCCESS)  
{
    size_t size;  
    clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL, &size);  
    cl_char *log = new cl_char[size];  
    clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, size, log, NULL);  
    fprintf(stderr, "clBuildProgram failed:
%s
", log);  
    delete[] log;  
}
```