The Open Computing Language (OpenCL)

Mike Bailey
mjb@cs.oregonstate.edu

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.
OpenCL wants you to break the problem up into Pieces

```
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 C++, you would say:

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

The OpenCL Language also supports Vector Parallelism

```
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.

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

```
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.)

Compute Units and Processing Elements are Arranged in Grids

A GPU Device is organized as a grid of Compute Units. Each Compute Unit is organized as a grid of Processing Elements. So in NVIDIA terms, their Turing GPU has 68 Compute Units, each of which has 64 Processing Elements, for a grand total of 4,352 Processing Elements.

Work-Groups are Arranged in Grids

- The GPU’s workload is divided into a Grid of Work-Groups
- Each Block’s workload is divided into a Grid of Work-Items

OpenCL Software Terminology: Work-Groups and Work-Items are Arranged in Grids

An OpenCL program is organized as a grid of Work-Groups. Each Work-Group is organized as a grid of Work-Items.

In terms of hardware, a Work-Group runs on a Compute Unit and a Work-Item runs on a Processing Element (PE).

One thread is assigned to each Work-Item. Threads are swapped on and off the PEs.

OpenCL Memory Model

```
Kernel
  Global Memory
  Constant Memory
  WorkGroup
    WorkGroup
      WorkGroup
        WorkItem
        WorkItem
        WorkItem
```

http://news.cision.com
### Rules

- Threads can share memory with the other Threads in the same Work-Group.
- Threads can synchronize with other Threads in the same Work-Group.
- Global and Constant memory is accessible by all Threads in all Work-Groups.
- Each Thread has registers and private memory.
- Each Work-Group has a maximum number of registers it can use. These are divided equally among all its Threads.

### OpenCL Error Codes

This one is defined as zero. All the others are negative.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL_SUCCESS</td>
<td>Valid</td>
</tr>
<tr>
<td>CL_INVALID_QUEUE_PROPERTIES</td>
<td>Invalid queue properties</td>
</tr>
<tr>
<td>CL_INVALID_COMMAND_QUEUE</td>
<td>Invalid command queue</td>
</tr>
<tr>
<td>CL_INVALID_MEM_OBJECT</td>
<td>Invalid memory object</td>
</tr>
<tr>
<td>CL_INVALID_IMAGE_FORMAT_DESCRIPTOR</td>
<td>Invalid image format descriptor</td>
</tr>
<tr>
<td>CL_INVALID_IMAGE_SIZE</td>
<td>Invalid image size</td>
</tr>
<tr>
<td>CL_INVALID_IMAGE_FORMAT</td>
<td>Invalid image format</td>
</tr>
<tr>
<td>CL_INVALID_SAMPLER</td>
<td>Invalid sampler</td>
</tr>
<tr>
<td>CL_INVALID_BINARY</td>
<td>Invalid binary</td>
</tr>
<tr>
<td>CL_INVALID_HOST_PTR</td>
<td>Invalid host pointer</td>
</tr>
<tr>
<td>CL_INVALID_QUEUE_PROPERTIES</td>
<td>Invalid queue properties</td>
</tr>
<tr>
<td>CL_INVALID_CONTEXT</td>
<td>Invalid context</td>
</tr>
</tbody>
</table>

### Querying the Number of Platforms (usually one)

```c
cl_int numPlatforms;
status = clGetPlatformIDs( 0, NULL, &numPlatforms );
if( status != CL_SUCCESS )
    PrintCLError( ERROR, "clGetPlatformIDs failed (1)\n" );
status = clGetPlatformIDs( numPlatforms, platforms, NULL );
if( status != CL_SUCCESS )
    PrintCLError( ERROR, "clGetPlatformIDs failed (2)\n" );
```

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

```c
for( int i = 0; i < numPlatforms; i++ )
    clGetPlatformIDs( platforms[i], CL_PLATFORM_VENDOR, size, str, NULL );
    sprintf( OUTPUT , "Vendor = '%s'
", str );
    delete[ ] str;
    clGetPlatformIDs( platforms[i], CL_PLATFORM_NAME, size, str, NULL );
    sprintf( OUTPUT , "Name = '%s'
", str );
    delete[ ] str;
```

### Querying the Number of Devices on a Platform

```c
cl_int numDevices;
status = clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, numDevices, devices, NULL );
if( status != CL_SUCCESS )
    PrintCLError( ERROR, "clGetDeviceIDs failed\n" );
```

### A Way to Print OpenCL Error Codes – get this from our Reference Page

```c
void PrintCLError( cl_int errorCode, char * prefix, FILE *fp )
{
    const int numErrorCodes = sizeof( ErrorCodes ) / sizeof( struct errorcode );
    char * meaning = ErrorCodes[errorCode].meaning;
    if( errorCode == CL_SUCCESS )
        return;
    for( int i = 0; i < numErrorCodes; i++ )
        if( errorCode == ErrorCodes[i].statusCode )
            meaning = ErrorCodes[i].meaning;
    fprintf( fp, "%s %s\n", prefix, meaning );
}
```

### Querying the Device (this is really useful!)

```c
cl_int numPlatforms;
status = clGetPlatformIDs( 0, NULL, &numPlatforms );
if( status != CL_SUCCESS )
    PrintCLError( ERROR, "clGetPlatformIDs failed\n" );
for( int i = 0; i < numPlatforms; i++ )
    clGetPlatformIDs( platforms[i], CL_PLATFORM_VENDOR, size, str, NULL );
    for( int j = 0; j < numDevices; j++ )
        clGetDeviceInfo( devices[j], CL_DEVICE_VENDOR, size, str, NULL );
```

### Getting Just the GPU Device(s)

```c
device_id device;
status = clGetDeviceIDs( platform, CL_DEVICE_TYPE_GPU, &device, NULL );
```
`clGetDeviceInfo(device, CL_DEVICE_EXTENSIONS, 0, NULL, &size);`  

Typical Values from Querying the Device  

<table>
<thead>
<tr>
<th>Number of Platforms: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform 0:</td>
</tr>
<tr>
<td>Name: 'NVidia CUDA'</td>
</tr>
<tr>
<td>Vendor: 'NVidia Corporation'</td>
</tr>
<tr>
<td>Version: 'OpenCL 1.1 CUDA 4.1.1'</td>
</tr>
<tr>
<td>Profile: FULL_PROFILE</td>
</tr>
<tr>
<td>Device 0:</td>
</tr>
<tr>
<td>Type: 0x0004 = CL_DEVICE_TYPE_GPU</td>
</tr>
<tr>
<td>Device Vendor ID: 0x10de</td>
</tr>
<tr>
<td>Device Maximum Compute Units: 15</td>
</tr>
<tr>
<td>Device Maximum Work Item Dimensions: 3</td>
</tr>
<tr>
<td>Device Maximum Work Item Sizes: 1024 x 1024 x 64</td>
</tr>
<tr>
<td>Device Maximum Work Group Size: 1024</td>
</tr>
<tr>
<td>Device Maximum Clock Frequency: 1401 MHz</td>
</tr>
<tr>
<td>Kernel Maximum Work Group Size: 1024</td>
</tr>
<tr>
<td>Kernel Local Memory Size: 0</td>
</tr>
</tbody>
</table>

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
1. *cpp Program Header*

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <omp.h> // for timing
```

2. Allocate the Host Memory Buffers

```c
// 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] = 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
```

This could have also been done like this:

```c
float hA[NUM_ELEMENTS];
```

3. Create an OpenCL Context

```c
c_context context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
```

4. Create an OpenCL Command Queue

```c
c_command_queue cmdQueue = clCreateCommandQueue(context, device, 0, &status);
```

5. Allocate the Device Memory Buffers

```c
c_mem dA = clCreateBuffer(context, CL_MEM_READ_ONLY, dataSize, NULL, &status);
c_mem dB = clCreateBuffer(context, CL_MEM_READ_ONLY, dataSize, NULL, &status);
c_mem dC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, dataSize, NULL, &status);
```

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.

6. Write the Data from the Host Buffers to the Device Buffers

```c
 status = clEnqueueWriteBuffer(cmdQueue, dA, CL_FALSE, 0, dataSize, hA, 0, NULL, NULL);
 status = clEnqueueWriteBuffer(cmdQueue, dB, CL_FALSE, 0, dataSize, hB, 0, NULL, NULL);
 status = clEnqueueWriteBuffer(cmdQueue, dC, CL_FALSE, 0, dataSize, hC, 0, NULL, NULL);
```
Enqueuing Works Like a Conveyer Belt

Write Buffer dA

Execute Kernel

Write Buffer dB

Write Buffer dC

Whoop-a, whoop-a

The .cl File

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.

Application Program

OpenCL Driver does the Compile and Link

OpenCL code in a separate file

GPU

Application Program

GLSL Driver does the Compile and Link

GLSL shader code in a separate file

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

7. Read the Kernel Code from a File into a Character Array

const char *CL_FILE_NAME = { "arraymult.cl" };

FILE *fp = fopen( CL_FILE_NAME, "r" );
if( fp == NULL )
{
fprintf( stderr, "Cannot open OpenCL source file '%s'
return 1;
}
// read the characters from the opencl kernel program:
size_t fileSize = ftell( fp );
char *clProgramText = new char[ fileSize+1 ];
sz_t n = fread( clProgramText, 1, fileSize, fp );
fclose( fp );

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 ("y") at the end of each line, which Linux interprets as a garbage character.

You can test this by typing the Linux command:

do -c loop.csh

which will show you all the characters, even the "y" (which you don't want) and the "n" (newlines, which you do want).

To get rid of the carriage returns, enter the Linux command:

tr -d "y" < loop.csh > loop1.csh

Then run loop1.csh

Or, on some systems, there is a utility called dos2unix which does this for you:
ardos2unix < 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

8. Compile and Link the Kernel Code

```c
// create the kernel program on the device:
char *strings[1];
// an array of strings
strings[0] = clProgramText;
cl_program program = clCreateProgramWithSource( context, 1, (const char **)strings, NULL, &status );
delete [] clProgramText;

// build the kernel program on the device:
char *options = { "" };
status = clBuildProgram( program, 1, &device, options, NULL, NULL );
if( status != CL_SUCCESS ) { // retrieve 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 );
```

```c
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);  
```

---

Work-Groups, Local IDs, and Global IDs

"NDRange Index Space" can be 1D, 2D, or 3D. This one is 1D.

- GlobalIndexSpaceSize
- WorkGroups
- WorkGroupSize
- global work offset (always NULL)
- event wait list
- command queue
- device buffer
- host buffer
- event object
- # dimensions
- # events
- event wait list
- want to block until done?
- # bytes
- command queue
- offset
- host buffer
- event object
- offset
- status
- get_work_dim();
- get_global_size( uint dimindx;  
- size_t get_global_id( uint dimindx;  
- size_t get_local_size( uint dimindx;  
- size_t get_local_id( uint dimindx;  
- size_t get_num_groups( uint dimindx;  
- size_t get_group_id( uint dimindx;  
- size_t get_global_offset( uint dimindx;  
- 0 ≤ dimindx ≤ 2
13. Clean Everything Up

```cpp
clReleaseKernel( kernel );
clReleaseProgram( program );
clReleaseCommandQueue( cmdQueue );
clReleaseMemObject( dA );
clReleaseMemObject( dB );
clReleaseMemObject( dC );
delete [ ] hA;
delete [ ] hB;
delete [ ] hC;
```

Do this because we created these arrays with `new`.

---

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>64</td>
<td>164</td>
</tr>
<tr>
<td>32</td>
<td>102</td>
</tr>
<tr>
<td>16</td>
<td>68</td>
</tr>
<tr>
<td>8</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
</tr>
</tbody>
</table>

---

Writing out the .cl Program's Binary Code

```cpp
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'\n" );
} else {
    fwrite( binary, 1, size, fpbin );
    fclose( fpbin );
}
delete [ ] binary;
```

---

Importing that Binary Code back in:
8. Compile and Link the Kernel Code

```cpp
char *options = { "" }; status = clBuildProgram( program, 1, &device, options, NULL, NULL );
```

You would do this:

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

And you still have to do this:

```cpp
char *strings[ 1 ];
strings[0] = clProgramText;
cl_program program = clCreateProgramWithSource( context, 1, (const char **)strings, NULL, &status );
delete [ ] clProgramText;
```

---

Instead of doing this:

```cpp
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 in\n\n%s\n\n", log );
delete [ ] log;
```