The Open Computing Language (OpenCL)

Also go look at the files first.cpp and first.cl!

OpenCL consists of two parts: a C/C++-callable API and a C-ish programming language.

The 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 GPUs.

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 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

OpenCL – Vendor-independent GPU Programming

This happens in the vendor-specific driver
The OpenCL Programming Environment

C/C++ program plus OpenCL code

Compiler and Linker

CPU binary on the host

Compiler and Linker

OpenCL binary on the GPU

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/C++, you would say:

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

If you were writing in OpenCL, you would say:

The OpenCL Language also supports Vector Parallelism

OpenCL code 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.

float4 f, g;
float4 d = (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 d16 = x16 * y16 + z16;

(Note: just because the language supports it, doesn’t mean the hardware does.)

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

Global Memory

Constant Memory

Local Memory
### 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.
- Global and constant memory is often cached inside a Work-Group.
- Each thread has registers and private memory.
- Each work-group has a maximum number of registers it can use.
- 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 other threads in the same work-group.

### Querying the Number of Devices on a Platform

```c
cl_uint numPlatforms;
status = clGetPlatformIDs( 0, NULL, &numPlatforms );
if( status != CL_SUCCESS )
    fprintf(stderr, "clGetPlatformIDs failed (2)
    
if( status != CL_SUCCESS )
    status = clGetPlatformIDs( 0, NULL, &numPlatforms );

// find out how many devices are attached here and get their ids:
cl_device_id * devices;
cl_uint numDevices;
status = clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, numDevices, devices, NULL);
if( status != CL_SUCCESS )
    fprintf(stderr, "clGetDeviceIDs failed

// get used to it:

// this way of querying information is a recurring OpenCL pattern
fprintf(stderr, "clGetDeviceIDs failed (1)

// querying the number of platforms
for( int i = 0; i < (int)numPlatforms; i++ )
    cl_device_id * devices;
    cl_uint numDevices;
    if( status != CL_SUCCESS )
        status = clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, numDevices, devices, NULL);```

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

- `numDevices` is the number of devices attached to the platform.
- `devices` is the array of device IDs.
- `platform` is the platform to query.
- `CL_DEVICE_TYPE_ALL` is the type of device to query.

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

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

```c
size_t extensionSize;
clGetDeviceInfo( device, CL_DEVICE_EXTENSIONS, 0, NULL, &extensionSize );
char *extensions = new char [extensionSize];
clGetDeviceInfo( device, CL_DEVICE_EXTENSIONS, extensionSize, extensions, NULL );
```

```
for( int i = 0; i < (int)strlen(extensions); i++ )
if( extensions[ i ] == ' ' )
extensions[ i ] = '
';
```

```
switch( type )
{
    case CL_DEVICE_TYPE_GPU:
        fprintf( OUTPUT, "CL_DEVICE_TYPE_GPU
" );
        break;
    case CL_DEVICE_TYPE_ACCELERATOR:
        fprintf( OUTPUT, "CL_DEVICE_TYPE_ACCELERATOR
" );
        break;
    case CL_DEVICE_TYPE_CPU:
        fprintf( OUTPUT, "CL_DEVICE_TYPE_CPU
" );
        break;
    default:
        fprintf( OUTPUT, "CL_DEVICE_TYPE_ACCELERATOR
" );
        break;
}
```

Device Extensions:
- `cl_khr_global_int32_base_atomics`
- `cl_khr_global_int32_extended_atomics`
- `cl_khr_local_int32_extended_atomics`

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
# CPP Program Header

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

1. 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] = hB[i] = sqrtf( (float) i );
}
// array size in bytes (will need this later):
size_t dataSize = NUM_ELEMENTS * sizeof( float );
```

This could have also been done like this:

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

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.

2. Create an OpenCL Context

```c
// create a context:
cl_context context = clCreateContext( NULL, 1, &device, NULL, NULL, &status );
```

3. Create an OpenCL Command Queue

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

4. Allocate the Device Memory Buffers

```c
// allocate memory buffers on the device:
cl_mem dA = clCreateBuffer( context, CL_MEM_READ_ONLY , dataSize, NULL, &status );
dataSize = NUM_ELEMENTS * sizeof( float );
```

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.

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

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

6. Write the Data from the Host Buffers to the Device Buffers
Enqueuing Works Like a Conveyer Belt

Write Buffer dC
Execute Kernel
Write Buffer dB
Write Buffer dA

OpenCL code is compiled in the Driver . . .

Application Program
OpenCL Driver does the Compile and Link

OpenCL code in a separate file

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

(. . . just like OpenGL's GLSL Shader code is compiled in the driver)

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.);
}

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

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:

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 ('\r') at the end of each line, which Linux interprets as a garbage character. You can test this by typing the Linux command:

```bash
od -c  loop.csh
```

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

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

```bash
tr -d  '\r'  <  loop.csh  >  loop1.csh
```

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

```bash
dos2unix < loop.csh > loop1.csh
```

Sorry about this. Unfortunately, this is a fact of life when you mix Windows and Linux.
Coming: 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

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 (`#include`) file
3. You can simulate a `#include` to re-use common pieces of code

9. Create the Kernel Object
10. Setup the Arguments to the Kernel Object

```
kernel void ArrayMult( global const float *dA, global const float *dB, global float *dC )
```

11. Enqueue the Kernel Object for Execution

```
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 ) {
    # dimensions
    global work offset (always NULL)
    # events
    event wait list
    # events
    event object
}
```

## Work-Groups, Local IDs, and Global IDs

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

```
Gx = 20
Lx = 4
Wx = 5
```

### NDRange Index Space can be 2D. This one is 2D.

```
Gx = 20
Gy = 12
Lx = 4
Ly = 3
Wy = 5
Wy = 4
```

### NDRange Index Space can be 3D. This one is 3D.

```
Gx = 20
Gy = 12
Gz = 3
Lx = 4
Ly = 3
Lz = 2
```

### Figuring Out What Thread You Are and What Your Thread Environment is Like

```
uint get_work_dim( ) ;
size_t get_global_id( uint dimindx ) ;
size_t get_local_id( uint dimindx ) ;
size_t get_group_id( uint dimindx ) ;
size_t get_num_groups( uint dimindx ) ;
size_t get_global_size( uint dimindx ) ;
size_t get_local_size( uint dimindx ) ;
```

0 ≤ dimindx ≤ 2
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
if(cleanEverythingUp:)
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmdQueue);
    clReleaseMemObject(dA);
    clReleaseMemObject(dB);
    clReleaseMemObject(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>Work-Group Size</th>
<th>GigaMultiplications/Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>30</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'
    return;
} else {
    fwrite(binary, 1, size, fpbin);
    fclose(fpbin);
}
delete [] binary;
```

Importing that Binary Code back In:

```c
8. Compile and Link the Kernel Code
unsigned char byteArray[numBytes];
status = clCreateProgramWithBinary(context, 1, &device, &numBytes, &byteArray, &binaryStatus, &status);
delete [] byteArray;
```

Instead of doing this:

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

You would do this:

```c
unsigned char *binary[1];
cl_program program = clCreateProgramWithBinary(context, 1, &device, NULL, &status, &binary);
delete [] binary[0];
```

And you still have to do this:

```c
cl_program program = clCreateProgramWithSource(context, 1, &device, NULL, &status);
delete [] options;
clEnqueueBuildProgram(program, 1, &device, options, NULL, NULL);
```
Billions of OpenCL Multiplies per Second on rabbit’s NVIDIA Titan Black
(Array size = 64M)

Billions of OpenCL Multiplies per Second on rabbit’s NVIDIA Titan Black
(Local Size = 64)