The Open Computing Language (OpenCL)

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

Mike Bailey
mjb@cs.oregonstate.edu
Oregon State University

OpenCL

• 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 (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.

Who Is Behind OpenCL?
Members of Khronos’s OpenCL Working Group

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

OpenCL – Vendor-independent GPU Programming

OpenCL Programming Environment
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 );
}
```

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 n`, `int n`, `float n`, 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.)

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.

- 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. These are divided equally among all its Threads
This way of querying information is a recurring OpenCL pattern (get used to it):

```c
status = clGetPlatformIDs( numPlatforms, platforms, NULL         );
status = clGetPlatformIDs(        0,                 NULL,       &numPlatforms );
for( int i = 0; i < (int)numPlatforms; i++ )
    cl_uint numDevices;
    if( status != CL_SUCCESS )
        status = clGetPlatformIDs( numPlatforms, platforms, NULL );
    platforms = new cl_platform_id[ numPlatforms ];
    fprintf( OUTPUT, "Number of Platforms = %d
", numPlatforms );
    if( status != CL_SUCCESS )
        status = clGetPlatformIDs( 0, NULL, &numPlatforms );
        cl_uint numPlatforms;

// find out how many platforms are attached here and get their ids:
This way of querying information is a recurring OpenCL pattern (get used to it):

```
switch( type )
{
    case CL_DEVICE_TYPE_CPU:
        fprintf( OUTPUT, "CL_DEVICE_TYPE_CPU
" );
        break;
    case CL_DEVICE_TYPE_GPU:
        fprintf( OUTPUT, "CL_DEVICE_TYPE_GPU
" );
        break;
    case CL_DEVICE_TYPE_ACCELERATOR:
        fprintf( OUTPUT, "CL_DEVICE_TYPE_ACCELERATOR
" );
        break;
    default:
        fprintf( OUTPUT, "Other...
" );
        break;
}
clGetDeviceInfo( devices[ j ], CL_DEVICE_VENDOR_ID, sizeof(ui), &ui, NULL);
fprintf( OUTPUT, "Device Vendor ID = 0x%04x
", ui );
clGetDeviceInfo( devices[ j ], CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(ui), &ui, NULL);
fprintf( OUTPUT, "Device Maximum Compute Units = %d
", ui );
clGetDeviceInfo( devices[ j ], CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(ui), &ui, NULL);
fprintf( OUTPUT, "Device Maximum Work Item Dimensions = %d
", ui );
clGetDeviceInfo( devices[ j ], CL_DEVICE_MAX_WORK_ITEM_SIZES, sizeof(sizes), sizes, NULL);
fprintf( OUTPUT, "Device Maximum Work Item Sizes = %d x %d x %d
", sizes[0], sizes[1], sizes[2] );
clGetDeviceInfo( devices[ j ], CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size), &size, NULL);
fprintf( OUTPUT, "Device Maximum Work Group Size = %d
", size );
clGetDeviceInfo( devices[ j ], CL_DEVICE_MAX_CLOCK_FREQUENCY , sizeof(ui), &ui, NULL);
fprintf( OUTPUT, "Device Maximum Clock Frequency = %d MHz
", ui );
}

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 = 64 x 64 x 64
Kernel Local Memory Size = 0

size_t extensionSize;
clGetDeviceInfo( device, CL_DEVICE_EXTENSIONS, 0, NULL, &extensionSize );
char *extensions = new char [ extensionSize ];
clGetDeviceInfo( devices, CL_DEVICE_EXTENSIONS, extensionSize, extensions, NULL);
fprintf( stderr, "Device Extensions:
" );
for( int i = 0; i < (int)strlen(extensions); i++ )
{
    if( extensions[ i ] == ' ' )
        extensions[ i ] = '
';
}
fprintf( stderr, "%s
", extensions );
delete [] extensions;

Device Extensions:
cl_khr_byte_addressable_store
cl_khr_icd
cl_khr_gl_sharing
cl_nv_d3d9_sharing
cl_nv_d3d10_sharing
cl_khr_d3d10_sharing
cl_nv_d3d11_sharing
cl_khr_compiler_options
cl_nv_device_attribute_query
cl_nv pragma unroll
cl_khr_global_int32_base_atomics
cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics
cl_khr_local_int32_extended_atomics
cl_khr_fp64

This is the big one you are looking for. It shows that this OpenCL system can interoperate with OpenGL.
This one is handy too. It shows that this OpenCL system can support 64-bit floating point (i.e., double precision).

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

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <omp.h> // for timing
#include "cl.h"
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] = hB[i] = sqrtf((float) i);
}
```

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. So, you do not want to allocate a large array like this as a local variable.

(Here, it’s being done on the heap.)

3. Create an OpenCL Context

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

### Properties
- `NULL`: No context properties.
- `1`: Number of devices.
- `device`: The device context.
- `NULL`: No user data.
- `NULL`: No status object.

4. Create an OpenCL Command Queue

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

### Properties
- `context`: The context object.
- `device`: The device object.
- `0`: No event wait list.

5. Allocate the Device Memory Buffers

```c
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);
```

# bytes
- `dataSize`: The size of the data.
- `CL_MEM_READ_ONLY`: The buffer can be read but not written.
- `CL_MEM_WRITE_ONLY`: The buffer can be written to but not read.
- `CL_MEM_READ_WRITE`: The buffer can be both read and written.

The read and write terminology is with respect to the OpenCL device. So, `CL_MEM_READONLY` 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_WRITEONLY` and `CL_MEM_READWRITE`.

6. 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);
status = clEnqueueWriteBuffer(cmdQueue, dB, CL_FALSE, 0, dataSize, hB, 0, NULL, NULL);
```

Enqueuing Works Like a Conveyer Belt

- **Read Buffer dA**
- **Execute Kernel**
- **Write Buffer dB**
- **Write Buffer dA**
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.

gid = which element we are dealing with right now.

OpenCL code is compiled in the Driver...

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

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

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

"r" should work, since the .cl file is pure ASCII text, but some people report that it doesn't work unless you use "rb"

Watch out for the 'r' + 'n' problem (See the next slide.)

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:

cd < 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:

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

Then run loop1.csh

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

dos2unix < loop.csh > loop1.csh

Also, you can run csh scripts (or .cpp programs) directly on Windows, and the 'r' is ignored, because csh is aware of this problem.

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

8. Compile and Link the Kernel Code

if create the kernel program on the device:

cfg "strings[ ]": if an array of strings
strings[ ] = createProgramWithSource
cfg (context, 1, (const char *[])"arraymult.cl", NULL, &status);
delete (1) configText

cfg (build the kernel program on the device:

cfg "options n"[ ]": if not NULL, then the program is built

Sorry about this. Unfortunately, this is a fact of life when you mix Windows and Linux.
How does the 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...

c_program program = clCreateProgramWithSource( context, 1, (const char **) ArrayOfStrings, NULL, &status );
```

These are two ways to provide a single character buffer:

```c
char *buffer[1];
buffer[0] = "...the entire OpenCL code...";

c_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

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

9. Create the Kernel Object

```c
c_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 );
```

11. Enqueue the Kernel Object for Execution

```c
size_t globalWorkSize[3] = { NUM_ELEMENT, 1, 1 };
size_t localWorkSize[3] = { LOCAL_SIZE, 1, 1 };
status = clEnqueueNDRangeKernel( cmdQueue, kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL );
```

Work-Groups, Local IDs, and Global IDs

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

```
# Dimensions # Events Event Object
status = clEnqueueNDRangeKernel( cmdQueue, kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL );
```

#WorkGroups = GlobalIndexSpaceSize
WorkGroupSize

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

Wx = 4

Lx = 4

#WorkGroups = GlobalIndexSpaceSize
WorkGroupSize

5x4 = 20
4
```
Work-Groups, Local IDs, and Global IDs

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

<table>
<thead>
<tr>
<th>GlobalIndexSpaceSize</th>
<th>WorkGroups</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WorkGroupSize</td>
</tr>
<tr>
<td></td>
<td>Lx = 4</td>
</tr>
<tr>
<td></td>
<td>Ly = 3</td>
</tr>
<tr>
<td></td>
<td>Wx = 5</td>
</tr>
<tr>
<td></td>
<td>Wy = 4</td>
</tr>
</tbody>
</table>

Work-Groups, Local IDs, and Global IDs

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

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

```c
uint get_work_dim();
size_t 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

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

GigaMultiplications/Second

Array Size (K)

Work-Group Size

Array Size (M)

Billion OpenCL Multiplies per Second on rabbit’s NVIDIA Titan Black

Local Size

Billion OpenCL Multiplies per Second on rabbit’s NVIDIA Titan Black

(Local Size = 64)