Computer Graphics

The Open Computing Language (OpenCL)

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

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

Who is Part of the Khronos Group?

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

The OpenCL Paradigm

The OpenCL Language also supports Vector Parallelism

OpenCL wants you to break the problem up into Pieces

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.

Think of this as having an implied for-loop around it, looping through all possible values of \( \text{gid} \).
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. 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.
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. These are divided equally among all its Threads

OpenCL Error Codes

This one is #define'd as zero.

All the others are negative.

CL_SUCCESS
CL_DEVICE_NOT_FOUND
CL_DEVICE_NOT_AVAILABLE
CL_COMPILER_NOT_AVAILABLE
CL_MEM_OBJECT_ALLOCATION_FAILURE
CL_OUT_OF_RESOURCES
CL_OUT_OF_HOST_MEMORY
CL_PROFILING_INFO_NOT_AVAILABLE
CL_MEM_COPY_OVERLAP
CL_IMAGE_FORMAT_MISMATCH
CL_IMAGE_FORMAT_NOT_SUPPORTED
CL_BUILD_PROGRAM_FAILURE
CL_MAP_FAILURE
CL_INVALID_QUEUE_PROPS
CL_INVALID_COMMAND_QUEUE
CL_INVALID_HOST_PTR
CL_INVALID_MEM_OBJECT
CL_INVALID_IMAGE_FORMAT_DESCRIPTOR
CL_INVALID_IMAGE_SIZE
CL_INVALID_SAMPLER
CL_INVALID_BINARY
CL_INVALID_BUILD_OPTIONS
CL_INVALID_PROGRAM
CL_INVALID_PROGRAM_EXECUTABLE
CL_INVALID_KERNEL_DEFINITION
CL_INVALID_KERNEL
CL_INVALID_ARG_INDEX
CL_INVALID_ARG_VALUE
CL_INVALID_ARG_SIZE
CL_INVALID_KERNEL_ARGS
CL_INVALID_WORK_DIMENSION

Querying the Number of Platforms (usually one)

cl_uint numPlatforms;
status = clGetPlatformIDs( 0, NULL, &numPlatforms );
if( status != CL_SUCCESS )
fprintf( stderr, "clGetPlatformIDs failed (1)\n" );
fprintf( stderr, "Number of Platforms = %d\n", numPlatforms );
cl_platform_id * platforms = new cl_platform_id[ numPlatforms ];
status = clGetPlatformIDs( numPlatforms, platforms, NULL );
if( status != CL_SUCCESS )
fprintf( stderr, "clGetPlatformIDs failed (2)\n" );

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

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

struct errorcode
{
    cl_int statusCode;
    char *meaning;
}
ErrorCodes[] =
{
   { CL_SUCCESS,                                        "\n"                                                },
   { CL_DEVICE_NOT_FOUND,                    "Device Not Found"                    },
   { CL_DEVICE_NOT_AVAILABLE,              "Device Not Available"                },
   . . .
   { CL_INVALID_MIP_LEVEL,                       "Invalid MIP Level"                     },
   { CL_INVALID_GLOBAL_WORK_SIZE,      "Invalid Global Work Size"         },
};
void
PrintCLError( cl_int errorCode,  char * prefix,  FILE *fp )
{
if( errorCode == CL_SUCCESS )
return;

const int numErrorCodes = sizeof( ErrorCodes ) / sizeof( struct errorcode );
char * meaning = " ";
for( int i = 0; i < numErrorCodes; i++ )
{
    if( errorCode == ErrorCodes[i].statusCode )
    {
        meaning = ErrorCodes[i].meaning;
        break;
    }
}
fprintf( fp, "%s %s\n", prefix, meaning );
Querying the Number of Devices on a Platform

```c
// find out how many devices are attached to each platform and get their ids:
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices);
devices = new cl_device_id[numDevices];
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, numDevices, devices, NULL);
```

Getting Just the GPU Device(s)

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

Querying the Device (this is really useful!), I

```c
// find out how many platforms are attached here and get their ids:
cl_uint numPlatforms;
status = clGetPlatformIDs(0, NULL, &numPlatforms);
if (status != CL_SUCCESS)
    fprintf(stderr, "clGetPlatformIDs failed (1)\n");
fprintf(OUTPUT, "Number of Platforms = %d\n", numPlatforms);
cl_platform_id *platforms = new cl_platform_id[numPlatforms];
status = clGetPlatformIDs(numPlatforms, platforms, NULL);
if (status != CL_SUCCESS)
    fprintf(stderr, "clGetPlatformIDs failed (2)\n");
```

Querying the Device, II

```c
clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, 0, NULL, &size);
str = new char[size];
clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, size, str, NULL);
fprintf(OUTPUT, "Name = '%s'\n", str);
delete[] str;
clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, 0, NULL, &size);
str = new char[size];
clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, size, str, NULL);
fprintf(OUTPUT, "Vendor = '%s'\n", str);
delete[] str;
```

Querying the Device, III

```c
switch(type)
{
    case CL_DEVICE_TYPE_CPU:
        fprintf(OUTPUT, "CL_DEVICE_TYPE_CPU\n");
        break;
    case CL_DEVICE_TYPE_GPU:
        fprintf(OUTPUT, "CL_DEVICE_TYPE_GPU\n");
        break;
    case CL_DEVICE_TYPE_ACCELERATOR:
        fprintf(OUTPUT, "CL_DEVICE_TYPE_ACCELERATOR\n");
        break;
    default:
        fprintf(OUTPUT, "Other...\n");
        break;
}
```
### 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 Compile Work Group Size = 0 x 0 x 0
  - Kernel Local Memory Size = 0

### Querying to see what extensions are supported on this device

```c
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] = '\n';
}
fprintf( stderr, "%s\n", extensions );
delete [] extensions;
```

Typical Values from Querying the Device

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).

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

1. .cpp Program Header

// 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 );

3. Create an OpenCL Context

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

4. Create an OpenCL Command Queue

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

2. Allocate the Host Memory Buffers
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 );
```

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 );
```

---

**Enqueuing Works Like a Conveyer Belt**

```plaintext
Read Buffer dC  | Execute Kernel  | Write Buffer dB  | Write Buffer dA
```

Whopp-a, whopp-a

**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.
OpenCL code in a separate file

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

GLSL Driver does the Compile and Link

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:
```
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:
```
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
```

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

// create the kernel program on the device:
char * strings[1] = { ... an array of strings };
dcl_program program = clCreateProgramWithSource(context, 1, (const char **)strings, NULL, &status);
delete [] strings;

// 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);
    clChar *log = new clChar[size];
    clGetProgramBuildInfo(program, devices[0], CL_PROGRAM_BUILD_LOG, size, log, NULL);
    fprintf(stderr, "clBuildProgram failed:
%s
", log);
    delete [] log;
}

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

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

These are two ways to provide a single character buffer:

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

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( );
```

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

- **G = 20**
- **W = 5**
- **L = 4**

- **GlobalIndexSpaceSize**
  - #WorkGroups = \( \frac{GlobalIndexSpaceSize}{WorkGroupsSize} \)
  - \( 5 \times 4 = \frac{20}{4} \)
Work-Groups, Local IDs, and Global IDs

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

\[
\text{WorkGroups} = \frac{\text{GlobalIndexSpaceSize}}{\text{WorkGroupSize}}
\]

\[
5 \times 4 = \frac{20 \times 12}{4 \times 3}
\]

Lx = 4

Ly = 3

Wx = 5

Wy = 4

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 ) ;
```

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 );
```

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

// clean everything up:
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>8</td>
<td>120</td>
</tr>
<tr>
<td>16</td>
<td>240</td>
</tr>
<tr>
<td>32</td>
<td>360</td>
</tr>
<tr>
<td>64</td>
<td>480</td>
</tr>
<tr>
<td>128</td>
<td>600</td>
</tr>
<tr>
<td>256</td>
<td>720</td>
</tr>
<tr>
<td>512</td>
<td>840</td>
</tr>
<tr>
<td>1024</td>
<td>960</td>
</tr>
</tbody>
</table>

Writing the .cl Program's Binary Code

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

You would do this:

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

And you still have to do this:

```c
char * options = { ""};
status = clBuildProgram( program, 1, &device, options, NULL, &status );
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.\n\n\", log );
    delete [ ] log;
}
```

Importing that Binary Code back In:

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;
```