The Open Computing Language (OpenCL)

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

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 getsfarmed 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 Part of 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 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.

```c
float f, g;
float4 f, g;
```

```c
float a16, x16, y16, z16;
float16 a16 = x16 * y16 + z16;
```

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

OpenCL Software Terminology:

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.

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 Memory Model

Kernel
Global Memory
Constant Memory
WorkGroup
WorkGroup
WorkGroup
Shared Memory
Work
Work
Work
Work
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

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

Querying the Number of Platforms (usually one)

```c
cl_platform_id *platforms = new cl_platform_id[numPlatforms];
if( status != CL_SUCCESS )
    fprintf( stderr, "clGetPlatformIDs failed (1)\n" );
if( status != CL_SUCCESS )
    cl_platform_id * platforms = new  cl_platform_id[numPlatforms];
    fprintf( stderr, "clGetPlatformIDs failed (1)\n" );
    fprintf( stderr, "clGetPlatformIDs failed (2)\n" );
    if( errorCode == CL_SUCCESS )
    {
        CL_INVALID_GLOBAL_WORK_SIZE, "Invalid Global Work Size"
    },
    CL_INVALID_MIP_LEVEL, "Invalid MIP Level"
```

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

```c
void PrintLError( cl_int errorCode, char * prefix, FILE *fp )
{
    struct errorcode ErrorCodes[] =
```

Querying the Number of Devices on a Platform

```c
int HowManyDevices = 0;
for( int i = 0; i < (int)numPlatforms; i++ )
```

Getting Just the GPU Device(s)

```c
cl_device_id *devices = new cl_device_id[numDevices];
```
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
  Vendor ID = 0x10de
  Device Name = 'NVIDIA CUDA'
  Device Maximum Clock Frequency = 1461 MHz
  Device Maximum Work Group Size = 1024 x 1024 x 64
  Device Maximum Work Item Dimensions = 3
  Device Local Memory Size = 0
  Device Maximum Compute Units = 15

Querying the Device, II

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

Typical Values from Querying the Device

Device Extensions:
- cl_khr_byte_addressable_store
- cl_khr_gl_int32_base_atomics
- cl_khr_gl_int32_extended_atomics
- cl_khr_global_int32_base_atomics
- cl_khr_global_int32_extended_atomics
- cl_khrileo
- cl_khr_openclext
- cl_khr_pragma_unroll
- cl_khr_pbg

Querying to see what extensions are supported on this device

Extensions: [cl_khr_byte_addressable_store, cl_khr_gl_int32_base_atomics, cl_khr_gl_int32_extended_atomics, cl_khr_global_int32_base_atomics, cl_khr_global_int32_extended_atomics, cl_khrileo, cl_khr_openclext, cl_khr_pragma_unroll, cl_khr_pbg]
1. .cpp Program Header

```c
#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

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.

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

3. Create an OpenCL Context

```c
cl_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
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 );
```

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 );
```

The command queue is like a task queue, where the devices can work on these tasks in any order. An event object describes the status of a command queue, and an event wait list allows the host to specify which event objects it needs to wait on before continuing.
Enqueuing Works Like a Conveyer Belt

Write Buffer dC
Execute Kernel
Write Buffer dB
Write Buffer dA

Whopp-a, whopp-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.

The .cl File

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

OpenCL code is compiled in the Driver . . .

Application Program

OpenCL code is in a separate file

OpenGL Driver does the Compile and Link

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

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
```
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

You do:

External
OpenCL
Compiler

Driver does:

Compiler in
driver

Vendor-specific
code

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

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

These are two ways to provide a single character buffer:

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);

cl_program program = clCreateProgramWithSource(context, 1, (const char **) &buffer, NULL, &status);

These are two ways to provide a single character buffer:

char *buffer[1];
buffer[0] = " . . . 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

9. Create the Kernel Object

cl_kernel kernel = clCreateKernel(program, "ArrayMult", &status);

10. Setup the Arguments to the Kernel Object

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();
```

11. Enqueue the Kernel Object for Execution

```
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

```
status = clEnqueueReadBuffer(cmdQueue, dC, CL_TRUE, 0, dataSize, hC, 0, NULL, NULL);
```

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

```
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.

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

- "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);
```

```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);
```
13. Clean Everything Up

```c
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

---


<table>
<thead>
<tr>
<th>Array Size (K)</th>
<th>Work-Group Size</th>
<th>GigaMultiplications/Second</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td></td>
</tr>
<tr>
<td></td>
<td>256</td>
<td></td>
</tr>
<tr>
<td></td>
<td>512</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td></td>
</tr>
</tbody>
</table>

---

15. 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;
```