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
- GPUs do 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.

What Standardization Efforts Does the Khronos Group Sponsor?

The Khronos Group

http://www.khronos.org/opencl/
http://en.wikipedia.org/wiki/OpenCL

Who is Part of the Khronos Group?

PROMOTER MEMBERS

Over 100 members worldwide
Any company is welcome to join

Active *OpenCL* Members

Implementers
Desktop/Mobile/Embedded/FPGA

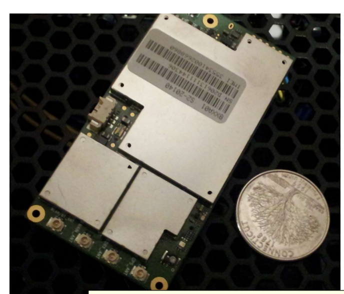
Single Source C++ Programming

Core API and Language Specs

Portable Kernel Intermediate Language

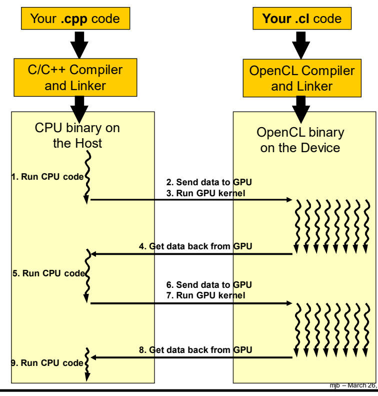
Working Group Members
Apps/Tools/Tests/Courseware

An example of using OpenCL in a System-on-a-Chip: Qualcomm Snapdragon – Full Linux and OpenCL



I used this as part of a project a few years ago. When I needed to take it from place-to-place, I put it in a very large Ziploc bag so I wouldn't accidentally lose it! ☺

The OpenCL Paradigm



OpenCL wants you to break the problem up into Pieces

If you were writing in **C/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 ];
}
```

If you were writing in **OpenCL**, you would say:

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

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

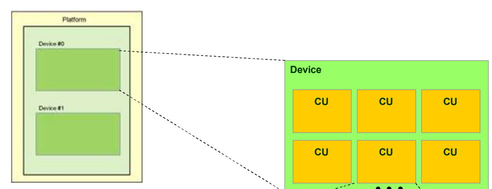
OpenCL code can be vector-oriented, meaning that it can perform a single instruction on multiple data values at the same time (SIMD).

Vector data types are: char*n*, int*n*, float*n*, where *n* = 2, 4, 8, or 16.

```
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.)

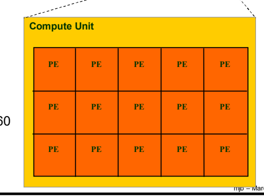
Compute Units and Processing Elements are Arranged in Grids



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 Blackwell 5090 has 170 Compute Units, each of which has 128 Processing Elements, for a grand total of 21,760 Processing Elements.

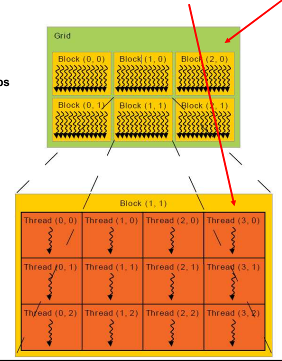


Work-Groups are Arranged in Grids

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

Grid of Work-Groups

Grid of Work-Items



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

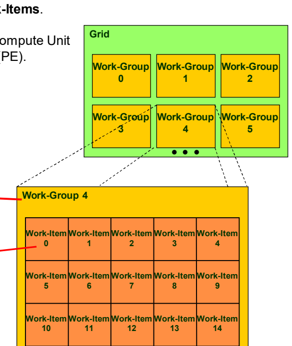
An OpenCL program is organized as a grid of **Work-Groups**.

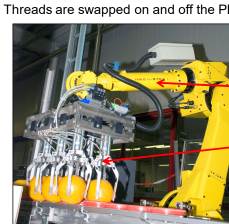
Each Work-Group is organized as a grid of **Work-Items**.

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.

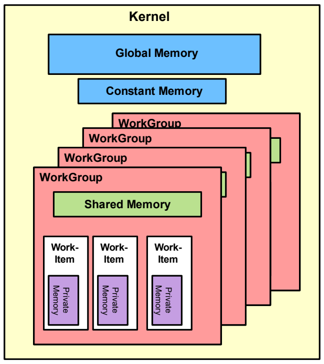
Threads are swapped on and off the PEs.





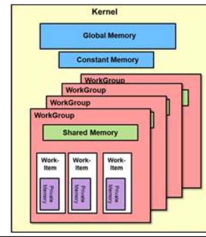
http://news.cision.com

OpenCL Memory Model



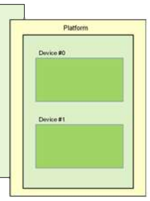
Rules

- 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



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):

	How many to get	Where to put them	How many total there are
status = **clGetPlatformIDs**(	0,	NULL,	&numPlatforms);
status = **clGetPlatformIDs**(	numPlatforms,	platforms,	NULL);

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_VALUE
CL_INVALID_DEVICE_TYPE
CL_INVALID_PLATFORM
CL_INVALID_DEVICE
CL_INVALID_CONTEXT

CL_INVALID_QUEUE_PROPERTIES
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_NAME
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

A Way to Print OpenCL Error Codes – get this from our Class Resources Page


```
struct errorcode
{
        cl_int          statusCode;
        char *           meaning;
}
ErrorCodes[ ] =
{
        { CL_SUCCESS,                          " "                              },
        { 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


```
// 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)

```
cl_device_id   device;

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

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

19

```
// 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" );

cl_uint numDevices;
cl_device_id *devices;

for( int i = 0; i < (int)numPlatforms; i++ )
{
	fprintf( OUTPUT, "Platform #%d:\n", i );
	size_t size;
	char *str;

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


Querying the Device, II

20

```
	clGetPlatformInfo( platforms[i], CL_PLATFORM_VERSION, 0, NULL, &size );
	str = new char [ size ];
	clGetPlatformInfo( platforms[i], CL_PLATFORM_VERSION, size, str, NULL );
	fprintf( OUTPUT, "\tVersion = '%s'\n", str );
	delete[ ] str;

	clGetPlatformInfo( platforms[i], CL_PLATFORM_PROFILE, 0, NULL, &size );
	str = new char [ size ];
	clGetPlatformInfo( platforms[i], CL_PLATFORM_PROFILE, size, str, NULL );
	fprintf( OUTPUT, "\tProfile = '%s'\n", str );
	delete[ ] str;

	// find out how many devices are attached to each platform and get their ids:

	status = clGetDeviceIDs( platforms[i], CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices );
	if( status != CL_SUCCESS )
		fprintf( stderr, "clGetDeviceIDs failed (2)\n" );

	devices = new cl_device_id[ numDevices ];
	status = clGetDeviceIDs( platforms[i], CL_DEVICE_TYPE_ALL, numDevices, devices, NULL );
	if( status != CL_SUCCESS )
		fprintf( stderr, "clGetDeviceIDs failed (2)\n" );

	for( int j = 0; j < (int)numDevices; j++ )
	{
		fprintf( OUTPUT, "\tDevice #%d:\n", j );
		size_t size;
		cl_device_type type;
		cl_uint ui;
		size_t sizes[3] = { 0, 0, 0 };

		clGetDeviceInfo( devices[j], CL_DEVICE_TYPE, sizeof(type), &type, NULL );
		fprintf( OUTPUT, "\t\tType = 0x%04x = ", type );
```


Querying the Device, III

21

```
		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;
		}
		clGetDeviceInfo( devices[j], CL_DEVICE_VENDOR_ID, sizeof(ui), &ui, NULL );
		fprintf( OUTPUT, "\t\tDevice Vendor ID = 0x%04x\n", ui );

		clGetDeviceInfo( devices[j], CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(ui), &ui, NULL );
		fprintf( OUTPUT, "\t\tDevice Maximum Compute Units = %d\n", ui );

		clGetDeviceInfo( devices[j], CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(ui), &ui, NULL );
		fprintf( OUTPUT, "\t\tDevice Maximum Work Item Dimensions = %d\n", ui );

		clGetDeviceInfo( devices[j], CL_DEVICE_MAX_WORK_ITEM_SIZES, sizeof(sizes), sizes, NULL );
		fprintf( OUTPUT, "\t\tDevice Maximum Work Item Sizes = %d x %d x %d\n", sizes[0], sizes[1], sizes[2] );

		clGetDeviceInfo( devices[j], CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size), &size, NULL );
		fprintf( OUTPUT, "\t\tDevice Maximum Work Group Size = %d\n", size );

		clGetDeviceInfo( devices[j], CL_DEVICE_MAX_CLOCK_FREQUENCY, sizeof(ui), &ui, NULL );
		fprintf( OUTPUT, "\t\tDevice Maximum Clock Frequency = %d MHz\n", ui );
	}
}
```


Typical Values from Querying the Device

22

```
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
```

This is the GPU on *rabbit*

Querying to see what extensions are supported on this device

23

```
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, "\nDevice Extensions:\n" );
for( int i = 0; i < (int)strlen(extensions); i++ )
{
		if( extensions[ i ] == ' ' )
				extensions[ i ] = '\n';
}
fprintf( stderr, "%s\n", extensions );
delete [ ] extensions;
```


Querying to see what extensions are supported on this device

24

This is important. 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). This is important in simulation.

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_nv_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

Steps in Creating and Running an OpenCL program

1. Program header
2. Create 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


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

#include "cl.h"
```


2. Create the Host Memory Buffers

Global memory and the heap typically have *lots* more memory available than the stack does. So, typically, you do not want to allocate large arrays like this as *local* variables.

```
// global variables:

float hA[ NUM_ELEMENTS ];
float hB[ NUM_ELEMENTS ];
float hC[ NUM_ELEMENTS ];

• • •

// in the main program, 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 );

// opencl function return status:

cl_int status;            // test against CL_SUCCESS
```


3. Create an OpenCL Context

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

```
// create a context:

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

properties
the device
Pass in user data
one device
Callback
returned status

4. Create an OpenCL Command Queue


```
// create a command queue:

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

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

the context
properties
the device
returned status

5. Allocate the Device Memory Buffers


```
// allocate memory buffers on the device:

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 );
```

```
cl_mem dA = clCreateBuffer( context, CL_MEM_READ_ONLY, dataSize, NULL, &status );
```

how this buffer is restricted
buffer data already allocated
bytes
returned 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


```
// 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 );
```

command queue

want to block until done?

bytes

events

event object

status = clEnqueueWriteBuffer(cmdQueue, dA, CL_FALSE, 0, dataSize, hA, 0, NULL, NULL);

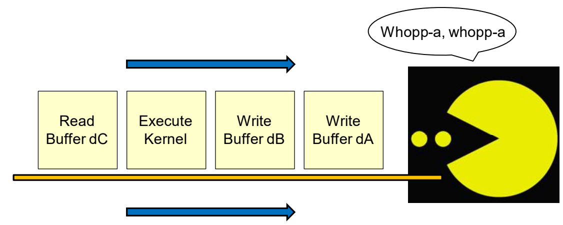
device buffer

offset

host buffer

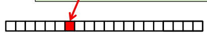
event wait list

Enqueuing Works Like a Conveyer Belt



The .cl File


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

          dC[gid] = dA[gid] * dB[gid];
}
```



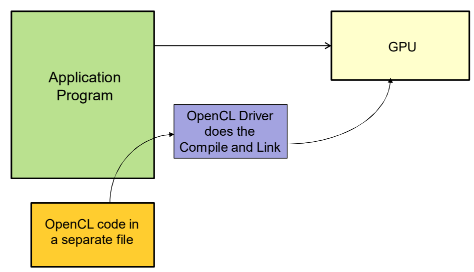
Which dimension's index are we fetching?

0 = X, 1 = Y, 2 = Z

Since this is a 1D problem, X is the only index we need to get.

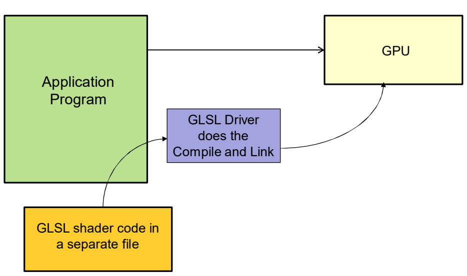
gid = which element we are dealing with right now.

OpenCL code is compiled in the Driver . . .




```
kernel void
ArrayMult( global float *A, global float *B, global float *C )
{
          int gid = get_global_id ( 0 );

          C[gid] = A[gid] * B[gid];
}
```


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




```
void main( )
{
          vec3 newcolor = texture( uTexUnit, vST) ).rgb;
          newcolor = mix( newcolor, uColor.rgb, uBlend );
          gl_FragColor = vec4( vLightIntensity*newcolor, 1. );
}
```


7. 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.)

```
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'\n", CL_FILE_NAME );
          return 1;
}

// read the characters from the opencl kernel program:

fseek( fp, 0, SEEK_END );
size_t  fileSize = ftell( fp );
fseek( fp, 0, SEEK_SET );
char *clProgramText =  new  char[ fileSize+1 ];
size_t  n = fread( clProgramText, 1, fileSize, fp );
clProgramText[fileSize] = '\0';
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 ('\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
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:\n%s\n", 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 . .. ";
cl_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 . . . ";
cl_program  program = clCreateProgramWithSource( context, 1, (const char **) buffer, NULL, &status );
```

```
char *buffer = " . . . 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 );
```

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


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( );
```

dimensions

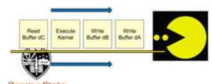
events

event object

status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);

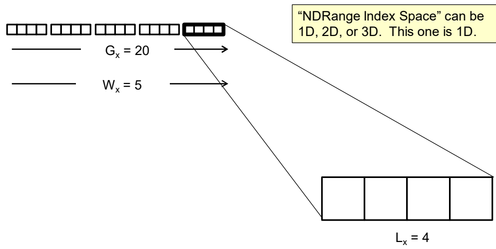
global work offset (always NULL)

event wait list



Work-Groups, Local IDs, and Global IDs

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

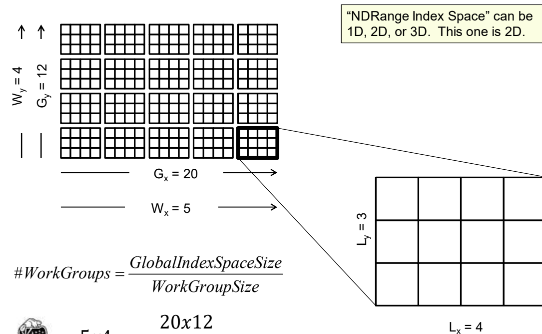


$$\#WorkGroups = \frac{GlobalIndexSpaceSize}{WorkGroupSize}$$

$$5 = \frac{20}{4}$$

Work-Groups, Local IDs, and Global IDs

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

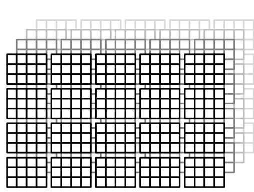


$$\#WorkGroups = \frac{GlobalIndexSpaceSize}{WorkGroupSize}$$

$$5x4 = \frac{20x12}{4x3}$$

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

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

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

command queue
want to block until done?
bytes
events
event object

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

device buffer
offset
host buffer
event wait list

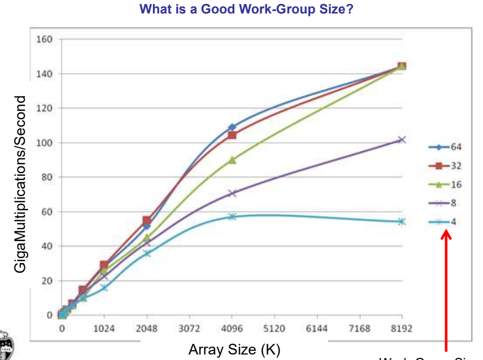
Oregon State University
Computer Graphics

13. Clean Everything Up

```
// clean everything up:

clReleaseKernel(            kernel   );
clReleaseProgram(           program  );
clReleaseCommandQueue(   cmdQueue );
clReleaseMemObject(     dA  );
clReleaseMemObject(     dB  );
clReleaseMemObject(     dC  );
```

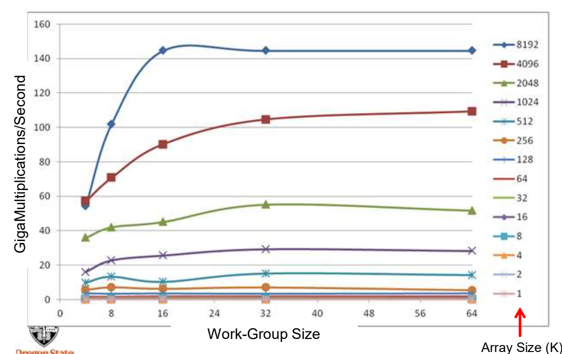
Oregon State University
Computer Graphics

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



Oregon State University
Computer Graphics

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



Oregon State University
Computer Graphics

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

Oregon State University
Computer Graphics

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

Instead of doing this:

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

You would do this:

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

And you still have to do this:

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

Computer Graphics