OpenCL™
Parallel computing for CPUs and GPUs

Lee Howes
Advanced Micro Devices
OpenCL™

With OpenCL™ you can...

- Leverage CPUs and GPUs to accelerate parallel computation
- Get dramatic speedups for computationally intensive applications
- Write accelerated portable code across different devices and architectures

With AMD’s implementations you can...

- Leverage CPUs, AMD’s GPUs, to accelerate parallel computation
- OpenCL Public release for multi-core CPU and AMD’s GPU’s December 2009
- The closely related DirectX® 11 public release supporting DirectCompute on AMD GPUs in October 2009, as part of Win7 Launch
The Heterogeneous Computing Software Ecosystem

Advanced Optimizations & Load Balancing

- Load balance across CPUs and GPUs; leverage Fusion performance advantages

End-user Applications

- High Level Frameworks
- Middleware/Libraries: Video, Imaging, Math/Sciences, Physics
- OpenCL™ & DirectCompute
- Hardware & Drivers: Fusion, Discrete CPUs/GPUs

- Increase ease of application development
- Drive new features into industry standards

Tools: HLL compilers, Debuggers, Profilers

Middleware/Libraries: Video, Imaging, Math/Sciences, Physics

Advanced Optimizations & Load Balancing

Performance advantages
What is OpenCL™?
- The OpenCL Platform, Execution and Memory Model
Using the OpenCL framework
- Setup
- Resource Allocation
- Compilation and execution of kernels
The OpenCL C language
- Language Features
- Built-in Functions
An OpenCL source example
Adding abstraction: C++ bindings for OpenCL
Two examples of use:
- SPH
- Soft body simulation
What is OpenCL™?
OpenCL™ Platform Model

A host connected to one or more OpenCL devices
An OpenCL device is
- A collection of one or more compute units (arguably cores)
- A compute unit is composed of one or more processing elements
- Processing elements execute code as SIMD or SPMD
OpenCL™ Execution Model

Kernel
- Basic unit of executable code - similar to a C function
- Data-parallel or task-parallel

Program
- Collection of kernels and other functions
- Analogous to a dynamic library

Applications queue kernel execution instances
- Queued in-order
- Executed in-order or out-of-order
Define N-dimensional computation domain (N = 1, 2 or 3)

- Each independent element of execution in N-D domain is called a work-item
- The N-D domain defines the total number of work-items that execute in parallel

E.g., process a 1024 x 1024 image: Global problem dimensions: 1024 x 1024 = 1 kernel execution per pixel: 1,048,576 total kernel executions

```
void scalar_mul(int n,  
    const float *a,    
    const float *b,    
    float *result) {  
    int i;  
    for (i=0; i<n; i++)  
        result[i] = a[i] * b[i]; 
}
```

```
kernel void dp_mul(  
    global const float *a,  
    global const float *b,  
    global float *result) {  
    int id = get_global_id(0);  
    result[id] = a[id] * b[id];  
    // execute dp_mul over "n" work-items
```
Expressing Data-Parallelism in OpenCL™ (continued)

Kernels executed across a global domain of work-items
- Global dimensions define the range of computation
- One work-item per computation, executed in parallel

Work-items are grouped in local workgroups
- Local dimensions define the size of the workgroups
- Executed together on one device
- Share local memory and synchronization

Caveats
- Global work-items must be independent: no global synchronization
- Synchronization can be done within a workgroup
Global Dimensions: 1024 x 1024
Local Dimensions: 128 x 128

Synchronization between work-items possible only within workgroups: barriers and memory fences

Can not synchronize outside of a workgroup

Choose the dimensions that are “best” for your algorithm
Task-Parallelism in OpenCL™

A task kernel executes as a single work-item
- No data-parallel iteration space

Flexible source language:
- Can be written in OpenCL™ C
- Or be native compiled from C/C++

Native functions can then be queued in the OpenCL queuing model (see later...)
OpenCL™ Memory Model

Private Memory
- Per work-item

Local Memory
- At least 32kB split into blocks each available to any work-item in a given work-group

Local Global/Constant Memory
- Not synchronized

Host Memory
- On the CPU

Memory management is explicit:
You must move data from host -> global -> local and back
OpenCL™ uses Dynamic/Runtime compilation model (like OpenGL®):

1. The code is compiled to an Intermediate Representation (IR)
   - Usually an assembler or a virtual machine
   - Known as offline compilation
2. The IR is compiled to a machine code for execution.
   - This step is much shorter.
   - It is known as online compilation.

In dynamic compilation, step 1 is done usually only once, and the IR is stored. The App loads the IR and performs step 2 during the App’s runtime (hence the term…)
Using the OpenCL™ framework
OpenCL™ Objects

Setup
- Devices — GPU, CPU, Cell/B.E.
- Contexts — Collection of devices
- Queues — Submit work to the device

Memory
- Buffers — Blocks of memory
- Images — 2D or 3D formatted images

Execution
- Programs — Collections of kernels
- Kernels — Argument/execution instances

Synchronization/profiling
- Events
OpenCL™ Framework

Context

Programs

Kernels

Memory Objects

Command Queues

In Order Queue

Out of Order Queue

GPU

Compile code

Create data & arguments

Send to execution

_kernel void add(
global const float *a,
global const float *b,
global float *c) {
    int id=get_global_id(0);
    c[id]=a[id]+b[id];
}

add
CPU program binary

add
GPU program binary

args
arg[0] value
arg[1] value
arg[2] value

Images

Buffers

In Order
Queue

Out of Order
Queue

GPU
1. Get the device(s)
2. Create a context
3. Create command queue(s)

```c
cl_uint num_devices_returned;
cl_device_id devices[2];
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1, &devices[0], &num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, &devices[1], &num_devices_returned);

cl_context context;
context = clCreateContext(0, 2, devices, NULL, NULL, &err);

cl_command_queue queue_gpu, queue_cpu;
queue_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
```
Setup: Notes

Devices
- Multiple cores on CPU or GPU together are a single device
- OpenCL™ executes kernels across all cores in a data-parallel manner

Contexts
- Enable sharing of memory between devices
- To share between devices, both devices must be in the same context

Queues
- All work submitted through queues
- Each device must have a queue
A system may have several devices—which is best?
The “best” device is algorithm- and hardware-dependent

Query device info with: `clGetDeviceInfo(device, param_name, *value)`

- Number of compute units: `CL_DEVICE_MAX_COMPUTE_UNITS`
- Clock frequency: `CL_DEVICE_MAX_CLOCK_FREQUENCY`
- Memory size: `CL_DEVICE_GLOBAL_MEM_SIZE`
- Extensions (double precision, atomics, etc.)

Pick the best device for your algorithm
Memory Resources

Buffers
- Simple chunks of memory
- Kernels can access however they like (array, pointers, structs)
- Kernels can read and write buffers

Images
- Opaque 2D or 3D formatted data structures
- Kernels access only via `read_image()` and `write_image()`
- Each image can be read or written in a kernel, but not both
Image Formats and Samplers

Formats
- Channel orders: CL_A, CL_RG, CL_RGB, CL_RGBA, etc.
- Channel data type: CL_UNORM_INT8, CL_FLOAT, etc.
- `clGetSupportedImageFormats()` returns supported formats

Samplers (for reading images)
- Filter mode: linear or nearest
- Addressing: clamp, clamp-to-edge, repeat, or none
- Normalized: true or false

Benefit from image access hardware on GPUs
cl_image_format format;
format.image_channel_data_type = CL_FLOAT;
format.image_channel_order = CL_RGBA

cl_mem input_image;
input_image = clCreateImage2D(context, CL_MEM_READ_ONLY, &format,
   image_width, image_height, 0, NULL,
   &err);

cl_mem output_image;
output_image = clCreateImage2D(context, CL_MEM_WRITE_ONLY, &format,
   image_width, image_height, 0, NULL, &err);

cl_mem input_buffer;
input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY,
   sizeof(cl_float)*4*image_width*image_height, NULL, &err);

cl_mem output_buffer;
output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
   sizeof(cl_float)*4*image_width*image_height, NULL, &err);
Reading and Writing Memory Object Data

Explicit commands to access memory object data
- Read from a region in memory object to host memory
  - `clEnqueueReadBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- Write to a region in memory object from host memory
  - `clEnqueueWriteBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- Map a region in memory object to host address space
  - `clEnqueueMapBuffer(queue, object, blocking, flags, offset, size, ...)`
- Copy regions of memory objects
  - `clEnqueueCopyBuffer(queue, srcobj, dstobj, src_offset, dst_offset, ...)`

Operate synchronously (`blocking = CL_TRUE`) or asynchronously
Compilation and execution of kernels
Program and Kernel Objects

Program objects encapsulate
- a program source or binary
- list of devices and latest successfully built executable for each device
- a list of kernel objects

Kernel objects encapsulate
- a specific kernel function in a program
  - declared with the kernel qualifier
- argument values
- kernel objects created after the program executable has been built
kernel void horizontal_reflect(read_only image2d_t src,
write_only image2d_t dst)
{
    int x = get_global_id(0);  // x-coord
    int y = get_global_id(1);  // y-coord
    int width = get_image_width(src);
    float4 src_val = read_imagef(src, sampler,
                                (int2)(width-1-x, y));
    write_imagef(dst, (int2)(x, y), src_val);
}
Executing Kernels

1. Set the kernel arguments
2. Enqueue the kernel

```c
err = clSetKernelArg(kernel, 0, sizeof(input), &input);
err = clSetKernelArg(kernel, 1, sizeof(output), &output);
size_t global[3] = {image_width, image_height, 0};
err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global, NULL, 0, NULL, NULL);
```

- Note: Your kernel is executed **asynchronously**
  - Nothing may happen—you have just enqueued your kernel
- To ensure execution:
  - Use a blocking read `clEnqueueRead*(... CL_TRUE ...)`
  - Use events to track the execution status
OpenCL™ defines a command queue
- Created on a single device
- Within the scope of a context

Commands are enqueued to a specific queue
- Kernels Execution
- Memory Operations

Events
- Each command can be created with an event associated
- Each command execution can be dependent in a list of pre-created events

Two types of queues
- In order queue: commands are executed in the order of issuing
- Out of order queue: command execution is dependent only on its event list completion

Multiple queues can be created on the same device
Commands can be dependent on events created on other queues/contexts
In the example above:
- C3 from Q1,1 depends on C1 & C2 from Q1,2
- C1 from Q1,4 depends on C2 from Q1,2
- In Q1,4, C3 depends on C2
Synchronization Between Commands

Each individual queue can execute in order or out of order
- For in-order queue, all commands execute in order
- Behaves as expected (as long as you’re enqueuing from one thread)

You must explicitly synchronize between queues
- Multiple devices each have their own queue
- Use events to synchronize

Events
- Commands return events and obey waitlists
  - clEnqueue*(... , num_events_in_waitlist, *event_waitlist, *event_out)
Synchronization: One Device/Queue

Example: Kernel 2 uses the results of Kernel 1

Kernel 2 waits in the queue until Kernel 1 is finished
Synchronization: Two Devices/Queues

- Kernel 1
  - Output
  - GPU
- Kernel 2
  - Input
  - CPU

Explicit dependency: Kernel 1 must finish before Kernel 2 starts
Synchronization: Two Devices/Queues

Kernel 2 issued with no associated wait. Starts before the results from Kernel 1 are ready

Kernel 2 waits for an event from Kernel 1. Does not start until the results from Kernel 1 are ready
Using Events on the Host

- `clWaitForEvents(num_events, *event_list)`
  - Blocks until events are complete

- `clEnqueueMarker(queue, *event)`
  - Returns an event for a marker that moves through the queue

- `clEnqueueWaitForEvents(queue, num_events, *event_list)`
  - Inserts a “WaitForEvents” into the queue

- `clGetEventInfo()`
  - Command type and status: `CL_QUEUED`, `CL_SUBMITTED`, `CL_RUNNING`, `CL_COMPLETE`, or error code

- `clGetEventProfilingInfo()`
  - Command queue, submit, start, and end times
The OpenCL™ C language
OpenCL™ C Language

Derived from ISO C99
- No standard C99 headers, function pointers, recursion, variable length arrays, and bit fields

Additions to the language for parallelism
- Work-items and workgroups
- Vector types
- Synchronization

Address space qualifiers
Optimized image access
Built-in functions
Kernel

A data-parallel function executed for each work-item

```
kernel void square(__global float* input, __global float* output)
{
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```
Work-Items and Workgroup Functions

get_work_dim = 1
get_global_size = 26

get_num_groups = 2
get_local_size = 13
get_group_id = 0

get_group_id = 1
get_local_id = 8
get_global_id = 21
Data Types

Scalar data types
- char, uchar, short, ushort, int, uint, long, ulong,
- bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)

Image types
- image2d_t, image3d_t, sampler_t

Vector data types
- Vector length of 2, 4, 8, and 16
- Aligned at vector length
- Vector operations and built-in

```c
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);
vi0.lo = vi1.hi;
int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);
```
Vector Operations

Vector literal

```c
int4 vi0 = (int4) -7;
int4 vil = (int4)(0, 1, 2, 3);
```

Vector components

```c
vi0.lo = vil.hl;
int8 v8 = (int8)(vi0, vil.s01, vil.odd);
```

Vector ops

```c
vi0 += vil;
vi0 = abs(vi0);
```

```
v0 = abs(vi0);
```

```
vi0 = abs(vi0);
```
Address Spaces

**Kernel** pointer arguments must use **global**, **local**, or **constant**

- `kernel void distance(global float8* stars, local float8* local_stars)`
- `kernel void sum(private int* p) // Illegal because it uses private`

Default address space for arguments and local variables is **private**

- `kernel void smooth(global float* io) { float temp; ...`

image2d_t and image3d_t are always in **global** address space

- `kernel void average(read_only global image_t in, write_only image2d_t out)`
Program (global) variables must be in **constant** address space

```cpp
constant float bigG = 6.67428E-11;
global float time; // Illegal non constant
kernel void force(global float4 mass) { time = 1.7643E18f; }
```

Casting between different address spaces is undefined

```cpp
kernel void calcEMF(global float4* particles) {
    global float* particle_ptr = (global float*) particles;
    float* private_ptr = (float*) particles; // Undefined behavior -
    float particle = *private_ptr; // different address space
}
```
Scalar and pointer conversions follow C99 rules

- No implicit conversions for vector types
  
  ```
  float4 f4 = int4_vec; // Illegal implicit conversion
  ```

- No casts for vector types (different semantics for vectors)
  
  ```
  float4 f4 = (float4) int4_vec; // Illegal cast
  ```

- Casts have other problems
  
  ```
  float x;
  int i = (int)(x + 0.5f); // Round float to nearest integer
  ```

  Wrong for:
  - 0.5f - 1 ulp (rounds up not down)
  - negative numbers (wrong answer)

- There is hardware to do it on nearly every machine
Conversions

Explicit conversions: convert_destType<_sat><_roundingMode>

- Scalar and vector types
- No ambiguity

Options:

- Saturate to 0
- Round down to nearest even
- Round up to nearest value
- Saturated to 255

```c
uchar4 c4 = convert_uchar4_sat_rte(f4);
```

<table>
<thead>
<tr>
<th>f4</th>
<th>0</th>
<th>254</th>
<th>255</th>
<th>1.2E9f</th>
</tr>
</thead>
<tbody>
<tr>
<td>c4</td>
<td>-5.0f</td>
<td>254.5f</td>
<td>254.6</td>
<td>255</td>
</tr>
</tbody>
</table>
Reinterpret the bits to another type
Types must be the same size

```c
// f[i] = f[i] < g[i] ? f[i] : 0.0f
float4 f, g;
int4 is_less = f < g;
f = as_float4(as_int4(f) & is_less);
```

OpenCL™ provides a `select` built-in
Built-in Math Functions

IEEE 754 compatible rounding behavior for single precision floating-point
IEEE 754 compliant behavior for double precision floating-point
Defines maximum error of math functions as ULP values
Handle ambiguous C99 library edge cases

Commonly used single precision math functions come in three flavors

- \( \log(x) \)
  - Full precision \( \leq 3 \text{ ulps} \)
  - Half precision/faster. \ half\_log\ —minimum 11 bits of accuracy, \( \leq 8192 \text{ ulps} \)
  - Native precision/fastest. \ native\_log\: accuracy is implementation defined

- Choose between accuracy and performance
Built-in Workgroup Functions

Synchronization

- barrier
- mem_fence, read_mem_fence, write_mem_fence

Work-group functions

- Encountered by all work-items in the work-group
- With the same argument values

```c
kernel read(global int* g, local int* shared) {
    if (get_global_id(0) < 5)
        barrier(CLK_GLOBAL_MEM_FENCE);
    else
        k = array[0];
}
```

Illegal since not all work-items encounter barrier
async_work_group_copy
- Copy from global to local or local to global memory
- Use DMA engine or do a memcpy across work-items in work-group
- Returns an event object

wait_group_events
- wait for events that identify async_work_group_copy operations to complete
Built-in Functions

Integer functions
- abs, abs_diff, add_sat, hadd, rhadd, clz, mad_hi, mad_sat, max, min, mul_hi, rotate, sub_sat, upsample

Image functions
- read_image[f | i | ui]
- write_image[f | i | ui]
- get_image_[width | height | depth]

Common, Geometric and Relational Functions

Vector Data Load and Store Functions
- eg. vload_half, vstore_half, vload_halfn, vstore_halfn, ...
Extensions

Atomic functions to global and local memory
- add, sub, xchg, inc, dec, cmp_xchg, min, max, and, or, xor
- 32-bit/64-bit integers

Select rounding mode for a group of instructions at compile time
- For instructions that operate on floating-point or produce floating-point values
  - \#pragma opencl_select_rounding_mode rounding_mode
- All 4 rounding modes supported

Extension: Check \texttt{clGetDeviceInfo} with \texttt{CL\_DEVICE\_EXTENSIONS}
Programming advice
Compiling programs can be expensive
- Reuse programs where possible
- Precompile binaries if this makes sense

Moving data to/from some devices can be expensive
- Discrete GPU over PCIe®
- Keep data on the device

Starting a kernel can be expensive
- Try to make individual kernels do a large amount of work

On some devices events can be expensive
- Don’t over-use them
Large global work sizes help hide memory latency and overheads
- 1000+ work items preferred

Trade off mathematics performance/precision using half/native versions of functions

Divergent execution can be bad on some devices
- Try to make all work-items in a group use the same control path

Handle data reuse using local memory where possible

Access memory sequentially across work items
- Allows memory coalescing
- Offers bandwidth improvements
Debugging

Start on the CPU

Be very careful about reading/writing out-of-bounds on the GPU
- Add explicit address checks around reads and writes if kernel is crashing

Play nicely with other apps
- GPUs are not preemptively scheduled

Use extra output buffers to record intermediate values

Set a context/call-back function to report API errors
OpenCL™ Basic Source Example
__kernel void vec_add (__global const float *a,
__global const float *b,
__global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
// Enumerate platforms
cl_context
cl_platform_id
err =
cl_platform_id *platforms =
nPlatforms;
platform;
clGetPlatformIDs(0,NULL,&nPlatforms);
malloc(nPlatforms);

// get list of all platforms
err = clGetPlatformIDs(nPlatforms,platforms,NULL);
cl_context_properties p[3] = {CL_CONTEXT_PLATFORM, (cl_context_properties)platform[0], 0 };
// create the OpenCL context on all devices
cl_context context = clCreateContextFromType(
    p, CL_DEVICE_TYPE_ALL, NULL, NULL, NULL);

// get the list of all devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);
// create a command-queue
cl_cmd_queue cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
cl_mem memobjs[3];
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, 
sizeof(cl_float)*n, NULL, NULL);
Vector Addition - Host API (3)

```c
cl_program program = clCreateProgramWithSource(
    context, 1, &program_source, NULL, NULL);

cl_int err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

cl_kernel kernel = clCreateKernel(program, "vec_add", NULL);
err = clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2], sizeof(cl_mem));
```

Spec Guide

- Creating program objects: Section 5.4.1
- Building program executables: Section 5.4.2
- Creating kernel objects: Section 5.5.1
- Setting kernel arguments: Section 5.5.2
// set work-item dimensions
size_t global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(
    cmd_queue, kernel, 1, NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(
    context, memobjs[2], CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
Adding abstraction
The OpenCL™ C++ bindings
“In my experience, C++ is alive and well -- thriving, even. This surprises many people. It is not uncommon for me to be asked, essentially, why somebody would choose to program in C++ instead of in a simpler language with more extensive "standard" library support, e.g., Java or C#.”

Scott Meyers

C++ offers:

- Data abstraction
- Object-oriented programming
- Generic templates
Lightweight, providing access to the low-level features of the original OpenCL™ C API

Compatible with standard C++ compilers (GCC 4.x and VS 2008)

C++ features that may be considered acceptable by all, e.g. exceptions, should not be required but may be supported by the use of polices that are not enabled by default

Goals

Should not require the use of the Standard Template Library

- Many organizations are wary of using the STL in their products

The bindings should be defined completely within a header, cl.hpp

- No linking to precompiled code should be necessary
A simple example: Hello from OpenCL™ C++ bindings

#define __CL_ENABLE_EXCEPTIONS
#define __NO_STD_VECTOR
#define __NO_STD_STRING
#if defined(__APPLE__) || defined(__MACOSX)
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif

#include <cstdio>
#include <cstdlib>
#include <iostream>

const char * helloStr = "__kernel void hello(void) { }\n";
A simple example: Hello from OpenCL™ C++ bindings (2)

```c
int main(void) {
    try {
        cl::Context context(CL_DEVICE_TYPE_GPU, 0, NULL, NULL, &err);
        cl::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
        cl::Program::Sources source(1, std::make_pair(helloStr,strlen(helloStr)));
        cl::Program program_ = cl::Program(context, source);
        program_.build(devices);
        cl::Kernel kernel(program_, "hello", &err);
        cl::CommandQueue queue(context, devices[0], 0, &err);
        cl::KernelFunctor func = kernel.bind(queue, cl::NDRange(4, 4),
                                              cl::NDRange(2, 2));
        func().wait();
    } catch (cl::Error err) {
        std::cerr << "ERROR: " << err.what() << "(" << err.err() << ")" <<
                  std::endl;
    }
    return EXIT_SUCCESS;
}
```
FFT algorithms are common and often computationally intensive. They are processed in a data-parallel manner. Today’s GPUs are excellent at executing FFTs. A 1k x 1k size is straightforward, but a 2k x 2k size is still manageable. However, real-time rendering becomes a challenge for a 2k x 2k sim (around 2Mn polygons).
Real-time ocean simulator (2)

Jerry Tenssendorf’s work:
- Simulating Ocean Water, SIGGRAPH 1999
- Waterworld, Titanic, and many others (2kx2k FFTs)
- Works with sums of sinusoids but starts in Fourier domain
- Can evaluate at any time t without having to evaluate other times
- Use the Phillips Spectrum
- Roughness of waves is a function of wind velocity

Jason L. Mitchell’s work:
- Real-Time Synthesis and Rendering of Ocean Water, 2005
- DX-9 Demo 256x256 FFTs, low and high frequencies separated
Real-time ocean simulator (3)
The first task is to obtain a platform.

```cpp
cl::vector<cl::Platform> platforms;
err = cl::Platform::get(&platforms);
checkErr(err && (platforms.size() == 0 ? -1 : CL_SUCCESS), "cl::Platform::get()");
```

// As cl::vector (implements std::vector interface) straightforward to determine number of platforms, no need for additional variables as required by clGetPlatformIDs.
```cpp
std::cout << "Number of platforms: \t " << platforms.size() << std::endl;
for (cl::vector<cl::Platform>::iterator i=platforms.begin(); i!=platforms.end(); ++i) {
    // pick a platform and do something
    std::cout << " Platform Name: " << (*i).getInfo<CL_PLATFORM_NAME>().c_str() << std::endl;
}
```
clGetXInfo functions are provided in two flavors:

- A static version of the form:
  
  ```
  template <cl_int name> typename
detail::param_traits<detail::cl_device_info, name>::param_type
  getInfo(cl_int* err = NULL) const
  ```

- A dynamic version of the form:
  
  ```
  template <typename T> cl_int getInfo(cl_device_info name, T* param) const
  ```

Unlike the C API the C++ bindings return info values directly:

- No need to call info function to find memory requirements
- Mapping from cl_X_info enums to C++ types, so for cl_platform_info:
  
  ```
  F(cl_platform_info, CL_PLATFORM_PROFILE, STRING_CLASS) \
  F(cl_platform_info, CL_PLATFORM_VERSION, STRING_CLASS) \
  F(cl_platform_info, CL_PLATFORM_NAME, STRING_CLASS) \
  F(cl_platform_info, CL_PLATFORM_VENDOR, STRING_CLASS) \
  F(cl_platform_info, CL_PLATFORM_EXTENSIONS, STRING_CLASS)
  ```
Continue by creating a context over the platform’s set of devices

```cpp
cl::vector<cl::Platform>::iterator p = platforms.begin();
...
// Get all GPU devices supported by a particular platform
cl::vector<cl::Device> devices;
p->getDevices(CL_DEVICE_TYPE_GPU | CL_DEVICE_TYPE_CPU, &devices);

// Create a single context for all devices
cl::Context context(devices, NULL, NULL, NULL, &err);
checkErr(err, "Context::Context()");

// Create work-queues for CPU and GPU devices
queueCPU = cl::CommandQueue(context, devices[0], 0, &err);
checkErr(err, "CommandQueue::CommandQueue(CPU)");
queueGPU = cl::CommandQueue(context, devices[1], 0, &err);
checkErr(err, "CommandQueue::CommandQueue(GPU)");
```
Load and build a program for the devices

```cpp
std::ifstream file("ocean_kernels.cl");
checkErr(file.is_open() ? CL_SUCCESS : -1, "reading ocean_kernels.cl");
std::string prog(std::istreambuf_iterator<char>(file),
                 (std::istreambuf_iterator<char>()));

cl::Program::Sources source(1, std::make_pair(prog.c_str(), prog.length()+1));
cl::Program program(context, source);
err = program.build(devices);

if (err != CL_SUCCESS) {
    std::cout << "Info: " << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices);
    checkErr(err, "Program::build()");
}
```
The program consists of four kernels

- Two data-parallel FFT kernels
  
  \[
  \begin{align*}
  \text{__kernel __attribute__((reqd_work_group_size (64,1,1)))} \\
  \text{void kfft(__global float *greal, __global float *gimag)} \\
  \text{__kernel __attribute__((reqd_work_group_size (64,1,1)))} \\
  \text{void ktran(__global float *greal, __global float *gimag)}
  \end{align*}
  \]

- One kernel for calculating the partial differences, i.e. slopes, that is used to calculate the normals from the height map for light shading
  
  \[
  \text{__kernel void kPartialDiffs} \\
  \text{(__global float* h, __global float2 *slopeOut, uint width, uint height)}
  \]

- One (task) kernel for generating Phillips Spectrum
  
  \[
  \text{__kernel void phillips} \\
  \text{(__global float2 * buffer, __global float2 *const randomNums, float windSpeed,} \\
  \text{float windDir, unsigned int height, unsigned int width)}
  \]
The kernels must be built and invariant arguments may be set

```
kfftKernel = cl::Kernel(program, "kfft", &err);
checkErr(err, "Kernel::Kernel(kfft)");

ktranKernel = cl::Kernel(program, "ktran", &err);
checkErr(err, "Kernel::Kernel(ktrans)");

phillipsKernel = cl::Kernel(program, "kphillips", &err);
checkErr(err, "Kernel::Kernel(kphillips)");

partialDiffsKernel = cl::Kernel(program, "kPartialDiffs", &err);
checkErr(err, "Kernel::Kernel(kPartialDiffs)");
```
Memory buffers are allocated for a specific device and will later be passed to kernels for use

```cpp
imag = cl::Buffer(context, CL_MEM_READ_WRITE, 1024*1024*sizeof(float), 0, &err);
checkErr(err, "Buffer::Buffer(imag)");

spectrum = cl::Buffer(context, CL_MEM_READ_WRITE, 1024*1024*sizeof(cl_float2), 0, &err);
checkErr(err, "Buffer::Buffer(spectrum)");
```

// Height map and partial differences (i.e. slopes) generated directly into a GL buffer for rendering

```cpp
real = cl::BufferGL(context, CL_MEM_READ_WRITE, heightVBO, &err);
checkErr(err, "BufferGL::BufferGL(height)");

slopes = cl::BufferGL(context, CL_MEM_READ_WRITE, partialDiffsVBO, &err);
checkErr(err, "BufferGL::BufferGL(partialDiffs)");
```
Real-time ocean simulator: Doing the work

At some point all of this setup must be put to good use... first by waiting for the phillips computation to complete and then by acquiring the OpenGL® output buffer

// make sure spectrum is up-to-date, i.e. account for wind changes. in practice // we double buffer to avoid causing delays due to continual wind changes phillipsEvent.wait();

cl::vector<cl::Memory> v;
v.push_back(real); v.push_back(partialDifs);

err = queueGPU.enqueueAcquireGLObjects(&v);
checkErr(err, "Queue::enqueueAcquireGLObjects()");
Then continue by executing the FFT kernel itself

```c
// other arguments are invariant and set once during setup
err = kfftKernel.setArg(0, real);
err = queueGPU.enqueueNDRangeKernel(kfftKernel, cl::NullRange,
    cl::NDRange(1024*64), cl::NDRange(64));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(kfftKernel)");
```

```c
// other arguments are invariant and set once during setup
err = ktranKernel.setArg(0, real);
err = queueGPU.enqueueNDRangeKernel(ktranKernel, cl::NullRange,
    cl::NDRange(128*129/2 * 64), cl::NDRange(64));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(ktranKernel)");
```
Execute the FFT kernels again

// note, no need to set argument as they persist from previous calls
err = queueGPU.enqueueNDRangeKernel(kfftKernel, cl::NullRange,
    cl::NDRange(1024*64), cl::NDRange(64));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(kfftKernel2)");

err = queueGPU.enqueueNDRangeKernel(ktranKernel, cl::NullRange,
    cl::NDRange(128*129/2 * 64), cl::NDRange(64));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(ktranKernel2)");
Then calculate slopes and partial differences

```c
err = calculateSlopeKernel.setArg(0, real); err |= partialDiffsKernel.setArg(1, slopes);
err |= calculateSlopeKernel.setArg(2, width); err |= partialDiffsKernel.setArg(3, height);
checkErr(err, "Kernel::setArg(partialDiffsKernel)");

err = queueGPU.enqueueNDRangeKernel(partialDiffsKernel, cl::NullRange,
        cl::NDRange(width, height), cl::NDRange(8, 8));
checkErr(err, "CommandQueue::enqueueNDRangeKernel(partialDiffsKernel)" );
```
To allow the rendering code to display the output we must release the OpenGL® objects obtained earlier.

```c
// Do the release
err = queueGPU.enqueueReleaseGLObjects(&v);
checkErr(err, "Queue::enqueueReleaseGLObjects(GPU)");
// Finish the operations in the GPU queue - including the release
queueGPU.finish();
```

And if the wind changes we must execute the appropriate correction code...

```c
queueCPU.enqueueTask(phillipsKernel, NULL, &phillipsEvent);
```

The phillipsEvent will be correctly caught by the next run through the pipeline if it is created.
Optional features: Exceptions

Exceptions are controversial so inclusion is optional

They can be enabled by defining:
#define __CL_ENABLE_EXCEPTIONS

Before including cl.hpp
API calls that perviously returned a cl_int err will now throw cl::Error

```cpp
catch (cl::Error err)
{
    std::cerr << "ERROR: " << err.what() << "(" << err.err() << ")" << std::endl;
}
```

err.what() returns an error string; err.err() returns the error code
Optional features: STL

Not everyone wants to use the STL in their applications

The OpenCL™ C++ bindings use std::string and std::vector

Support alternative versions:

- cl::string
- cl::vector

Also allow user defined replacements
Before including cl.hpp define:
#define __NO_STD_VECTOR

There is a new vector template that takes the same template parameters as std::vector

template cl::vector< typename T, unsigned int N = __MAX_DEFAULT_VECTOR_SIZE >;

__MAX_DEFAULT_VECTOR_SIZE defaults to 10. Can be overridden by defining before including cl.hpp:
#define __MAX_DEFAULT_VECTOR_SIZE 5

Developer can provide own version of vector class by defining:
#define __USE_DEV_VECTOR
#define VECTOR_CLASS userVector

Where userVector must be a template of the form given above for cl::vector
Optional features: Replacing std::string

Before including cl.hpp define:
#define __NO_STD_STRING

There is a new vector template that takes the same template parameters as std::string
cl::string

__MAX_DEFAULT_VECTOR_SIZE defaults to 10. Can be overridden by defining before including cl.hpp:
#define __MAX_DEFAULT_VECTOR_SIZE 5

Developer can provide own version of string class by defining:
#define __USE_DEV_STRING
#define STRING_CLASS userString

Where userString must be a class supporting the same interface as std::string.
Incorporation into the standard

Planned for adoption as part of OpenCL™ 1.1

Can already be downloaded from the Khronos web site:

- http://www.khronos.org/registry/cl

Multiple platform and vendor through use of underlying C API
A pair of examples

GPU Cloth and Fluid Simulations in Bullet Physics
OpenCL™ Fluid Simulation in Bullet Physics
Bullet Physics

Games Physics Simulation (http://bulletphysics.com)
- Open source engine (Zlib license)
- Rigid body dynamics, e.g.
  - Ragdolls, destruction, and vehicles
- Soft body dynamics
  - Cloth, rope, and deformable volumes
- Current version is 2.76 and is CPU only
- Future versions will add a turbo injection, with the help of OpenCL
Open Physics – OpenCL™ and Bullet

- Rigid body dynamics
  - Erwin Coumans and Sony team developing accelerated version
  - 2D/3D Demos already exist and work on productization

- Soft body dynamics
  - AMD developing DC/OpenCL™ Cloth acceleration

- Fluid simulation
  - Does not currently exist in Bullet
  - AMD developing OpenCL/DC SPH implementation

- DMM Finite Element method
  - Pixelux working on Bullet integration
Fluids and particle systems the basics

- Simply, highly parallel, thus map well to the GPU
- Particles store position, mass, velocity, age, density, etc
- Particles are moved by time stepping:
  - Euler or Leapfrog integration \( \frac{dv_i}{dt} = a_i \)
  - Acceleration \( a_i \) has contributions from gravity, pressure gradient, and viscosity
For correct simulation of fluids, inter-particle forces are required

Naïve implementation leads to a complexity of $O(n^2)$

[Muller03] reduce this to linear complexity by introducing a cutoff distance $k$
Smoothed Particle Hydrodynamics – Algorithm

- Build spatial grid on particles
  - Allow fast neighbor finding

- For each particle
  - Find neighbors

- For each particle
  - Compute density and pressure

- For each particle
  - Compute acceleration

- For each particle
  - Integrate

On the GPU each particle is worked on in parallel
SPH – Reducing $O(n^2)$ to $O(n)$ with spatial hashing

Fill particles into a grid with spacing 2*interaction distance
Search potential neighbors in adjacent cells only
Map cells $[i,j,k]$ into 1D array via hash function $h(i,j,k)$ [teschner03]

Implementation:
- Infinite 3D grid is mapped to finite grid
  - $10 \times 10 \times 10$ in the unit cube
  - $(x,y,z)$ maps $(x\%11,y\%11,z\%11)$
- Finding neighbors is fast
- Requires fast GPU sort
  - Today Radixsort
  - Future other sorts, work still needed to study sorting algorithms on GPU
SPH – Early prototype
Reverted to pure streaming implementation
- Data is always read coherently

Pipeline is automatically generated
- Using sequence description file
- Predict memory bandwidth, automatically

Still in early development, running but:
- Simulation still unstable, this is a maths issue so:
  - Problem is memory bound and so – 256,000 particles @ 60Hz doable!
No ‘good’ solution to date

Screen-space Curvature Flow
- [Malladi et al., 1995, Van der Laan et al., 2009]
- Evolve surface along normal direction with speed determined by mean surface curvature
- Flow moves surface only in Z
- Needs a lot of particles to look good!

Almost all post rendering processing (geometry shader is used to render screen point sprites), so:
- Why not use OpenCL™?
  - OpenCL C Geometry and Fragment Shaders, with fixed function rasterization
OpenCL™ Soft Body simulation
Particle-based soft body simulation

Mass/spring system:
- Large collection of masses or particles
- Connect using springs
- Layout and properties of springs changes properties of soft body

General spring types:
- Structural springs
- Shearing springs
- Bending springs
Subset of full range of soft bodies. Large number of particles lends itself to parallel processing.

- Force from a given link must be applied to both particles
- Requires batching to correctly update particles

Basic stages of simulation:

1. Original layout
2. Current layout: Compute forces as stretch from rest length
3. Apply forces to masses and compute velocities
4. Compute new positions based on velocities
Batch the nodes based on their interconnections:
- Graph colouring
- Complicated link structure = many batches

Compute the velocity:
- Over each batch in turn
- Over multiple cycles to allow solution to converge

Update positions, faces, normals etc based on computed velocities in a single pass
Adding wind to the simulation

Compute normals on a per-face basis

- Number of faces can be optimised to improve performance
  - We use a face per node triangle in the obvious fashion

Node normals and areas computed from surrounding faces

Compute wind force interaction based on velocity and density of the wind medium

- Wind velocity
- Relative velocity
- Scale by medium density

Node velocity

Wind velocity

Relative velocity

Scale by medium density
The “Übercloth”

Multi-pass multi-phase algorithm
- Large number of overall passes

Multiple cloths require all passes to be computed for each
- Separate kernel issues can end up being small

Merge cloths into single simulated cloth object
- Inefficient on the CPU due to caching
- Highly efficient on the GPU:
  - Higher occupancy
  - Massively parallel
Acknowledgements

Ocean sample by:
  Ben Gaster
  Brian Sumner
  Justin Hensley

Cloth example produced in collaboration with:
  Justin Hensley
  Abe Wiley
  Jason Yang

SPH by:
  Saif Ali
  Alan Heirich
  Ben Gaster
DISCLAIMER
The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION
© 2010 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, ATI, the ATI logo, AMD Opteron, Radeon, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft, Windows, Windows Vista, and DirectX™ are registered trademarks of Microsoft Corporation in the United States and/or other jurisdictions. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. Other names are for informational purposes only and may be trademarks of their respective owners.