Outline

• More CUDA Libraries

• OpenGL Interface

• Introduction to OpenCL
CUDA Libraries

Joseph Kider
University of Pennsylvania
CIS 565 - Spring 2011

Libraries
CUBLAS
CUFFT
MAGMA
CULA
Thrust
...

3
CUDA Specialized Libraries: PyCUDA

- PyCUDA lets you access Nvidia’s CUDA parallel computation API from Python
PyCUDA

- Third party open source, written by Andreas Klöckner
- Exposes all of CUDA via Python bindings
- Compiles CUDA on the fly
  - CUDA is presented as an interpreted language
- Integrated with numpy
- Handles memory management, resource allocation
- CUDA programs are Python strings
  - Metaprogramming - modify source code on the fly

https://developer.nvidia.com/pycuda
PyCUDA - Differences

• Object cleanup tied to lifetime of objects
  – Easier to write correct, leak- and crash-free code
  – PyCUDA knows about dependencies, too, so it won’t detach from a context before all memory allocated in it is also freed

• Convenience: Abstractions like `pycuda.driver.SourceModule` and `pycuda.gpudarray.GPUArray` make CUDA programming even more convenient than with Nvidia’s C-based runtime

• Completeness: PyCUDA provides the full power of CUDA’s driver API

• Automatic Error Checking: All CUDA errors are automatically translated into Python exceptions

• Speed: PyCUDA’s base layer is written in C++
PyCUDA - Example

```python
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.size, a.dtype.itemsize)
cuda.memcpy_htod(a_gpu, a)

mod = cuda.SourceModule('"
__global__ void doublify(float *a)
{
  int idx = threadIdx.x + threadIdx.y*4;
a[idx] *= 2.0f;
}
"
')
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```
In GPU scripting, GPU code does \textit{not} need to be a compile-time constant.

(Key: Code is data—it \textit{wants} to be reasoned about at run time)
CUDA Specialized Libraries: CUDPP

- CUDPP: CUDA Data Parallel Primitives Library
  - CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum ("scan"), parallel sort and parallel reduction

http://cudpp.github.io/
CUDPP - Design Goals

• CUDPP is implemented as 4 layers:
  – The **Public Interface** is the external library interface, which is the intended entry point for most applications. The public interface calls into the Application-Level API.
  – The **Application-Level API** comprises functions callable from CPU code. These functions execute code jointly on the CPU (host) and the GPU by calling into the Kernel-Level API below them.
  – The **Kernel-Level API** comprises functions that run entirely on the GPU across an entire grid of thread blocks. These functions may call into the CTA-Level API below them.
  – The **CTA-Level API** comprises functions that run entirely on the GPU within a single Cooperative Thread Array (CTA, aka thread block). These are low-level functions that implement core data-parallel algorithms, typically by processing data within shared memory.
CUDPP + Thrust

- CUDPP's interface is optimized for performance while Thrust is oriented towards productivity

```c
int main(void)
{
    unsigned int numElements = 32768;

    // allocate host memory
    thrust::host_vector<float> h_idata(numElements);
    // initialize the memory
    thrust::generate(h_idata.begin(), h_idata.end(), rand);
```
CUDPP + Thrust

// set up plan
CUDPPConfiguration config;
config.op = CUDPP_ADD;
config.datatype = CUDPP_FLOAT;
config.algorithm = CUDPP_SCAN;
config.options = CUDPP_OPTION_FORWARD | CUDPP_OPTION_EXCLUSIVE;

CUDPPHandle scanplan = 0;
CUDPPResult result = cudppPlan(&scanplan, config, numElements, 1, 0);

if(CUDPP_SUCCESS != result)
{
    printf("Error creating CUDPPPlan\n");
    exit(-1);
}

// Run the scan
thrust::raw_pointer_cast(&d_odata[0]),
thrust::raw_pointer_cast(&d_idata[0]),
numElements);
CUDA Specialized Libraries: CUBLAS

- CUDA accelerated BLAS (Basic Linear Algebra Subprograms)

https://developer.nvidia.com/cublas
CUBLAS

• GPU Variant 100 times faster than CPU version
• Matrix size is limited by graphics card memory and texture size
• Although taking advantage of sparse matrices would help reduce memory consumption, sparse matrix storage is not implemented by CUBLAS
CUDA Specialized Libraries: CUFFT

- Cuda Based Fast Fourier Transform Library
- The FFT is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets
- One of the most important and widely used numerical algorithms, with applications that include computational physics and general signal processing
CUFFT

• Computes parallel FFT on the GPU
• Uses “plans” like FFTW*
  – A plan contains information about optimal configuration for a given transform
  – Plans can prevent recalculation
  – Good fit for CUFFT because different kinds of FFTs require different thread/block configurations

* FFTW is a popular CPU library for FFT
CUFFT

• 1D, 2D and 3D transforms of complex and real-valued data
• Batched execution for doing multiple 1D transforms in parallel
• 1D transform size up to 8M elements
• 2D and 3D transform sizes in the range [2, 16384]
• In-place and out-of-place transforms
CUDA Specialized Libraries: CULA

- CULA is EM Photonics' GPU-accelerated numerical linear algebra library that contains a growing list of LAPACK functions.
- LAPACK stands for Linear Algebra PACKage. It is an industry standard computational library that has been in development for over 15 years and provides a large number of routines for factorization, decomposition, system solvers, and eigenvalue problems.
CUDA Specialized Libraries: HONEI
(Hardware oriented numerics, efficiently implemented)

A collection of libraries for numerical computations targeting multiple processor architectures
HONEI

• HONEI is an open-source collection of libraries offering a hardware oriented approach to numerical calculations.

• HONEI abstracts the hardware, and applications written on top of HONEI can be executed on a wide range of computer architectures such as CPUs, GPUs and the Cell processor.
  – The most important frontend library is libhoneila, HONEI's linear algebra library.
  – The numerics and math library libhoneimath contains high performance kernels for iterative linear system solvers as well as other useful components like interpolation and approximation.
OpenGL Interface

Utah CS 6235
by Mary Hall
OpenGL Rendering

• OpenGL buffer objects can be mapped into the CUDA address space and then used as global memory
  – Vertex buffer objects
  – Pixel buffer objects
• Allows direct visualization of data from computation
  – No device to host transfer
  – Data stays in device memory -very fast compute / viz cycle
• Data can be accessed from the kernel like any other global data (in device memory)
OpenGL Interoperability

1. Register a buffer object with CUDA
   - `cudaGLRegisterBufferObject(GLuint buffObj);`
   - OpenGL can use a registered buffer only as a source
   - Unregister the buffer prior to rendering to it by OpenGL

2. Map the buffer object to CUDA memory
   - `cudaGLMapBufferObject(void** devPtr, GLuint buffObj);`
   - Returns an address in global memory
   - Buffer must be registered prior to mapping
OpenGL Interoperability

3. Launch a CUDA kernel to process the buffer
   – Unmap the buffer object prior to use by OpenGL
   – cudaGLUnmapBufferObject(GLuintbuffObj);

4. Unregister the buffer object
   – cudaGLUnregisterBufferObject(GLuintbuffObj);
   – Optional: needed if the buffer is a render target

5. Use the buffer object in OpenGL code
Example from simpleGL in SDK

1. GL calls to create and initialize buffer, then register with CUDA:

```c
// create buffer object
glGenBuffers( 1, vbo);
glBindBuffer( GL_ARRAY_BUFFER, *vbo);

// initialize buffer object
unsigned int size = mesh_width * mesh_height * 4 * sizeof( float)*2;
glBufferData( GL_ARRAY_BUFFER, size, 0,
               GL_DYNAMIC_DRAW);
glBindBuffer( GL_ARRAY_BUFFER, 0);

// register buffer object with CUDA
cudaGLRegisterBufferObject(*vbo);
```
Example from simpleGL in SDK

2. Map OpenGL buffer object for writing from CUDA
   
   ```c
   float4 *dptr;
   cudaGLMapBufferObject( (void**)&dptr, vbo));
   ```

3. Execute the kernel to compute values for dptr
   
   ```c
   dim3 block(8, 8, 1);
   dim3 grid(mesh_width / block.x, mesh_height / block.y, 1);
   kernel<<< grid, block>>>(dptr, mesh_width, mesh_height, anim);
   ```

4. Unregister the OpenGL buffer object and return to OpenGL
   
   ```c
   cudaGLUnmapBufferObject( vbo);
   ```
OpenCL

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011

with additional material from
Joseph Kider
University of Pennsylvania
CIS 565 - Spring 2009
Processor Parallelism

CPUs
Multiple cores driving performance increases

Multi-processor programming – e.g. OpenMP

Emerging Intersection

OpenCL

GPUs
Increasingly general purpose data-parallel computing

Heterogeneous Computing

Graphics APIs and Shading Languages

OpenCL is a programming framework for heterogeneous compute resources
OpenCL

• **Open Compute Language**
• For heterogeneous parallel-computing systems
• Cross-platform
  – Implementations for
    • ATI GPUs
    • NVIDIA GPUs
    • x86 CPUs
  – Is cross-platform really *one size fits all*?

OpenCL

- Standardized
- Initiated by Apple
- Developed by the Khronos Group
OpenCL Working Group

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers

- Many industry-leading experts involved in OpenCL’s design
  - A healthy diversity of industry perspectives

- Apple made initial proposal and is very active in the working group
  - Serving as specification editor
Spir

• Standard Portable Intermediate Representation
  – SPIR-V is first open standard, cross-API, intermediate language for natively representing parallel compute and graphics
  – Part of the core specification of:
    • OpenCL 2.1
    • the new Vulkan graphics and compute API
## Vulkan

<table>
<thead>
<tr>
<th>OpenGL</th>
<th>Vulkan</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Originally architected for graphics workstations with direct renderers and split memory</strong></td>
<td>Matches architecture of modern platforms including mobile platforms with unified memory, tiled rendering</td>
</tr>
<tr>
<td>Driver does lots of work: state validation, dependency tracking, error checking. Limits and randomizes performance</td>
<td>Explicit API – the application has direct, predictable control over the operation of the GPU</td>
</tr>
<tr>
<td>Threading model doesn’t enable generation of graphics commands in parallel to command execution</td>
<td>Multi-core friendly with multiple command buffers that can be created in parallel</td>
</tr>
<tr>
<td>Syntax evolved over twenty years – complex API choices can obscure optimal performance path</td>
<td>Removing legacy requirements simplifies API design, reduces specification size and enables clear usage guidance</td>
</tr>
<tr>
<td>Shader language compiler built into driver. Only GLSL supported. Have to ship shader source</td>
<td>SPIR-V as compiler target simplifies driver and enables front-end language flexibility and reliability</td>
</tr>
<tr>
<td>Despite conformance testing developers must often handle implementation variability between vendors</td>
<td>Simpler API, common language front-ends, more rigorous testing increase cross vendor functional/performance portability</td>
</tr>
</tbody>
</table>

Source: www.khronos.org/assets/uploads-developers-library-overview/2015_vulkan_v1_Overview.pdf
Vulkan

Complex drivers lead to driver overhead and cross vendor unpredictability
Error management is always active
Driver processes full shading language source
Separate APIs for desktop and mobile markets

Application
Traditional graphics drivers include significant context, memory and error management

Application responsible for memory allocation and thread management to generate command buffers

Direct GPU Control

Simpler drivers for low-overhead efficiency and cross vendor portability
Layered architecture so validation and debug layers can be unloaded when not needed
Run-time only has to ingest SPIR-V intermediate language
Unified API for mobile, desktop, console and embedded platforms

Vulkan Language Ecosystem

GLSL Shader Source
- GLSL will be first shading language supported by Vulkan

GLSL to SPIR-V Translator

Game Engines
- Can flexibly target SPIR-V and Vulkan back-ends

Future diversity in domain-specific languages, frameworks and tools
- E.g. C++ Shading Language

SPIR-V supported in Vulkan core

Vulkan Runtime

Device X
Device Y
Device Z

Source: www.khronos.org/assets/uploads developers/library/overview/2015_vulkan_v1_Overview.pdf
Design Goals of OpenCL

• Use all computational resources in the system
  – GPUs and CPUs as peers
  – Data- and task-parallel computing

• Efficient parallel programming model
  – Based on C
  – Abstract the specifics of underlying hardware
  – Define maximum allowable errors of math functions

• Drive future hardware requirements
OpenCL

• API similar to OpenGL
• Based on the C language
• Easy transition form CUDA to OpenCL
OpenCL and CUDA

- Many OpenCL features have a one to one mapping to CUDA features
- OpenCL
  - More complex platform and device management
  - More complex kernel launch

- OpenCL is more complex due to its support for multiplatform and multivendor portability
OpenCL and CUDA

• **Compute Unit (CU)** corresponds to
  – CUDA streaming multiprocessor (SMs)
  – CPU core
  – etc.

• **Processing Element** corresponds to
  – CUDA streaming processor (SP)
  – CPU ALU
OpenCL and CUDA

OpenCL and CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>Kernel</td>
</tr>
<tr>
<td>Host program</td>
<td>Host program</td>
</tr>
<tr>
<td>Thread</td>
<td>Work item</td>
</tr>
<tr>
<td>Block</td>
<td>Work group</td>
</tr>
<tr>
<td>Grid</td>
<td>NDRange (index space)</td>
</tr>
</tbody>
</table>
OpenCL and CUDA

- **Work Item** (CUDA thread) - executes kernel code
- **Index Space** (CUDA grid) - defines work items and how data is mapped to them
- **Work Group** (CUDA block) - work items in a work group can synchronize
OpenCL and CUDA

- **CUDA**: `threadIdx` and `blockIdx`
  - Combine to create a global thread ID
  - Example
    - `blockIdx.x * blockDim.x + threadIdx.x`
OpenCL and CUDA

- OpenCL: each thread has a unique global index
  - Retrieve with `get_global_id()`

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>threadIdx.x</code></td>
<td><code>get_local_id(0)</code></td>
</tr>
<tr>
<td><code>blockIdx.x * blockDim.x + threadIdx.x</code></td>
<td><code>get_global_id(0)</code></td>
</tr>
</tbody>
</table>
OpenCL and CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gridDim.x</code></td>
<td><code>get_num_groups(0)</code></td>
</tr>
<tr>
<td><code>blockIdx.x</code></td>
<td><code>get_group_id(0)</code></td>
</tr>
<tr>
<td><code>blockDim.x</code></td>
<td><code>get_local_size(0)</code></td>
</tr>
<tr>
<td><code>gridDim.x * blockDim.x</code></td>
<td><code>get_global_size(0)</code></td>
</tr>
</tbody>
</table>
OpenCL and CUDA

• Recall CUDA:

Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf
OpenCL and CUDA

• In OpenCL:

  - get_global_size(0)
  - get_local_size(0)

Index Space

Work Group (0,0)

Work Item (0, 0)  Work Item (1, 0)  Work Item (2, 0)  Work Item (3, 0)  Work Item (4, 0)
Work Item (0, 1)  Work Item (1, 1)  Work Item (2, 1)  Work Item (3, 1)  Work Item (4, 1)
Work Item (0, 2)  Work Item (1, 2)  Work Item (2, 2)  Work Item (3, 2)  Work Item (4, 2)

Work Group (0, 0)  Work Group (1, 0)  Work Group (2, 0)
Work Group (0, 1)  Work Group (1, 1)  Work Group (2, 1)

get_global_size(1)  get_local_size(1)
Kernels: Work-item and Work-group Example

- Dimension: 2
- Global size: 32x32 = 1024
- Number of groups: 16

Local id: (4, 2)
Global id: (28, 10)
OpenCL and CUDA

• Recall the CUDA memory model:
OpenCL and CUDA

• In OpenCL:
## OpenCL and CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global memory</td>
<td>Global memory</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Constant memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
</tr>
<tr>
<td>Local memory</td>
<td>Private memory</td>
</tr>
</tbody>
</table>

![Diagram of CUDA and OpenCL memory types]

51
## OpenCL and CUDA

<table>
<thead>
<tr>
<th>CUDA</th>
<th>Host Access</th>
<th>Device Access</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global memory</td>
<td>Dynamic allocation; read/write access</td>
<td>No allocation; read/write access by all work items in all work groups; large and slow but may be cached in some devices</td>
<td>Global memory</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Dynamic allocation; read/write access</td>
<td>Static allocation; read only access by all work items</td>
<td>Constant memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Dynamic allocation; no access</td>
<td>Static allocation; shared read/write access by all work items in a work group</td>
<td>Local memory</td>
</tr>
<tr>
<td>Local memory</td>
<td>No allocation; no access</td>
<td>Static allocation; read/write access by a single work item</td>
<td>Private memory</td>
</tr>
</tbody>
</table>

**CUDA**

- **Global memory**
  - Host: Dynamic allocation; read/write access
  - Device: No allocation; read/write access by all work items in all work groups; large and slow but may be cached in some devices

- **Constant memory**
  - Host: Dynamic allocation; read/write access
  - Device: Static allocation; read only access by all work items

- **Shared memory**
  - Host: Dynamic allocation; no access
  - Device: Static allocation; shared read/write access by all work items in a work group

- **Local memory**
  - Host: No allocation; no access
  - Device: Static allocation; read/write access by a single work item
OpenCL and CUDA

- Both also have **Fences**
  - In OpenCL
    - `mem_fence()`
    - `read_mem_fence()`
    - `write_mem_fence()`
OpenCL Fence Examples

- `mem_fence(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)`
  - waits until all reads/writes to local and/or global memory made by the calling work item prior to `mem_fence()` are visible to all threads in the work-group

- `barrier(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)`
  - waits until all work-items in the work-group have reached this point and calls `mem_fence(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)`
Porting CUDA to OpenCL™

- Qualifiers

<table>
<thead>
<tr>
<th>C for CUDA Terminology</th>
<th>OpenCL™ Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong> function</td>
<td>__kernel function</td>
</tr>
<tr>
<td><strong>device</strong> function</td>
<td>function (no qualifier required)</td>
</tr>
<tr>
<td><strong>constant</strong> variable declaration</td>
<td>__constant variable declaration</td>
</tr>
<tr>
<td><strong>device</strong> variable declaration</td>
<td>__global variable declaration</td>
</tr>
<tr>
<td><strong>shared</strong> variable declaration</td>
<td>__local variable declaration</td>
</tr>
</tbody>
</table>

## Data Types

<table>
<thead>
<tr>
<th>Scalar Type</th>
<th>Vector Type (n = 2, 4, 8, 16)</th>
<th>API Type for host app</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, uchar</td>
<td>charn, ucharn</td>
<td>cl_char&lt;n&gt;, cl_uchar&lt;n&gt;</td>
</tr>
<tr>
<td>short, ushort</td>
<td>shortn, ushortn</td>
<td>cl_short&lt;n&gt;, cl_ushort&lt;n&gt;</td>
</tr>
<tr>
<td>int, uint</td>
<td>intn, uintn</td>
<td>cl_int&lt;n&gt;, cl_uint&lt;n&gt;</td>
</tr>
<tr>
<td>long, ulong</td>
<td>longn, ulongn</td>
<td>cl_long&lt;n&gt;, cl_ulong&lt;n&gt;</td>
</tr>
<tr>
<td>float</td>
<td>floatn</td>
<td>cl_float&lt;n&gt;</td>
</tr>
</tbody>
</table>
Accessing Vector Components

- Accessing components for vector types with 2 or 4 components
  - \texttt{<vector2>.xy, <vector4>.xyzw}

```c
float2 pos;
pos.x = 1.0f;
pos.y = 1.0f;
pos.z = 1.0f; // illegal since vector only has 2 components

float4 c;
c.x = 1.0f;
c.y = 1.0f;
c.z = 1.0f;
c.w = 1.0f;
```
## Accessing Vector with Numeric Index

<table>
<thead>
<tr>
<th>Vector components</th>
<th>Numeric indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 components</td>
<td>0, 1</td>
</tr>
<tr>
<td>4 components</td>
<td>0, 1, 2, 3</td>
</tr>
<tr>
<td>8 components</td>
<td>0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>16 components</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F</td>
</tr>
</tbody>
</table>

```c
float8 f;
f.s0 = 1.0f; // the 1st component in the vector
f.s7 = 1.0f; // the 8th component in the vector

float16 x;
f.sa = 1.0f; // or f.sA is the 10th component in the vector
f.sF = 1.0f; // or f.sF is the 16th component in the vector
```
### Handy addressing of Vector Components

<table>
<thead>
<tr>
<th>Vector access suffix</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>.lo</td>
<td>Returns the lower half of a vector</td>
</tr>
<tr>
<td>.hi</td>
<td>Returns the upper half of a vector</td>
</tr>
<tr>
<td>.odd</td>
<td>Returns the odd components of a vector</td>
</tr>
<tr>
<td>.even</td>
<td>Returns the even components of a vector</td>
</tr>
</tbody>
</table>

```c
float4 f = (float4) (1.0f, 2.0f, 3.0f, 4.0f);
float2 low, high;
float2 o, e;

low = f.lo;  // returns f.xy (1.0f, 2.0f)
high = f.hi;  // returns f.zw (3.0f, 4.0f)
o = f.odd;   // returns f.yw (2.0f, 4.0f)
e = f.even;  // returns f.xz (1.0f, 3.0f)
```
OpenCL API

• Walkthrough OpenCL host code for running vecAdd kernel:

```c
__kernel void vecAdd(__global const float *a, __global const float *b, __global float *c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0,
    CL_DEVICE_TYPE_GPU, 0, 0, 0);
OpenCL API

// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, 0, 0, 0);

Create a context for a GPU
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  0, 0, &nContextDescriptorSize);
cl_device_id aDevices =
  malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
  nContextDescriptorSize, aDevices, 0);
OpenCL API

// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES, 0, 0, &nContextDescriptorSize);
cl_device_id aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES, nContextDescriptorSize, aDevices, 0);

Retrieve an array of each GPU
Choosing Devices

• A system may have several devices - which is best?
• The “best” device is algorithm-dependent
• Query device info with: `clGetDeviceInfo(device, param_name, *value)`
  – Number of compute units `CL_DEVICE_MAX_COMPUTE_UNITS`
  – Clock frequency `CL_DEVICE_CLOCK_FREQUENCY`
  – Memory size `CL_DEVICE_GLOBAL_MEM_SIZE`
  – Extensions (double precision, atomics, etc.)
• Pick best device for your algorithm
// create a command queue for first device the context reported

cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);
// create a command queue for first device the context reported

cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);

Create a command queue (CUDA stream) for the first GPU
// create & compile program
cl_program hProgram;
hProgram =
    clCreateProgramWithSource(hContext, 1, source, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

- A program contains one or more kernels. Think dll.
- Provide kernel source as a string
- Can also compile offline
// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "vecAdd", 0);

Create kernel from program
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
// allocate host vectors
float* pA = new float[cnDimension];
float* pB = new float[cnDimension];
float* pC = new float[cnDimension];

// initialize host memory
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);
OpenCL API

cl_mem hDeviceMemA = clCreateBuffer(
    hContext,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    cnDimension * sizeof(cl_float),
    pA,  0);

cl_mem hDeviceMemB = /* ... */
OpenCL API

```c
cl_mem hDeviceMemA = clCreateBuffer(
    hContext,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    cnDimension * sizeof(cl_float),
    pA,  0);
```

```c
cl_mem hDeviceMemB = /* ... */
```

Create buffers for kernel input. Read only in the kernel. Written by the host.
hDeviceMemC = clCreateBuffer(hContext, CL_MEM_WRITE_ONLY, cnDimension * sizeof(cl_float), 0, 0);

Create buffer for kernel output.
// setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void*) &hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void*) &hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void*) &hDeviceMemC);

Kernel arguments set by index
OpenCL API

// execute kernel
clEnqueueNDRangeKernel(hCmdQueue,
    hKernel, 1, 0, &cnDimension, 0, 0, 0, 0);

// copy results from device back to host
clEnqueueReadBuffer(hContext,
    hDeviceMemC, CL_TRUE, 0,
    cnDimension * sizeof(cl_float),
    pC, 0, 0, 0);}
OpenCL API

// execute kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0, &cnDimension, 0, 0, 0, 0);

// copy results from device back to host
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0, cnDimension * sizeof(cl_float), pC, 0, 0, 0);
cl_int clEnqueueNDRangeKernel (  
    cl_command_queue command_queue,  
    cl_kernel kernel,  
    cl_uint work_dim, <=3  
    const size_t *global_work_offset, NULL  
    const size_t *global_work_size,  
    const size_t *local_work_size,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
delete [] pA;
delete [] pB;
delete [] pC;

clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
CUDA Pointer Traversal

```c
struct Node { Node* next; }
n = n->next; // undefined operation in OpenCL, // since ‘n’ here is a kernel input
```
OpenCL Pointer Traversal

```c
struct Node { unsigned int next; }
...
n = bufBase + n; // pointer arithmetic is fine, bufBase is
// a kernel input param to the buffer’s beginning
```
Intro OpenCL Tutorial

Benedict R. Gaster, AMD
Architect, OpenCL™
The “Hello World” program in OpenCL

- Programs are passed to the OpenCL runtime via API calls expecting values of type char *
- Often, it is convenient to keep these programs in separate source files
  - For this tutorial, device programs are stored in files with names of the form name_kernels.cl
  - The corresponding device programs are loaded at runtime and passed to the OpenCL API
#include <utility>
#define __NO_STD_VECTOR
// Use cl::vector instead of STL version
#include <CL/cl.hpp>

// additional C++ headers, which are agnostic to
// OpenCL.
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
#include <iterator>

const std::string hw("Hello World\n");
inline void checkErr(cl_int err, const char * name)
{
    if (err != CL_SUCCESS) {
        std::cerr << "ERROR: " << name
                   << " (" << err << ")" << std::endl;
        exit(EXIT_FAILURE);
    }
}
OpenCL Contexts

```c
int main(void)
{
    cl_int err;
    cl::vector< cl::Platform > platformList;
    cl::Platform::get(&platformList);
    checkErr(platformList.size() != 0 ? CL_SUCCESS : -1, "cl::Platform::get");
    std::cerr << "Platform number is: " << platformList.size() << std::endl;

    std::string platformVendor;
    platformList[0].getInfo((cl_platform_info)CL_PLATFORM_VENDOR, &platformVendor);
    std::cerr << "Platform is by: " << platformVendor << "\n";
}
```
OpenCL Contexts

```cpp
cl_context_properties cprops[3] =
    {
        CL_CONTEXT_PLATFORM,
        (cl_context_properties)(platformList[0]),
        0};

cl::Context context(
    CL_DEVICE_TYPE_CPU, 
    cprops, 
    NULL, 
    NULL, 
    &err);
checkErr(err, "Context::Context()");
```

Just pick first platform
OpenCL Buffer

char * outH = new char[hw.length()+1];
cl::Buffer outCL(
    context,
    CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
    hw.length()+1,
    outH,
    &err);
checkErr(err, "Buffer::Buffer()");
OpenCL Devices

```cpp
cl::vector<cl::Device> devices;
devices =
    context.getInfo<CL_CONTEXT_DEVICES>();
checkErr(devices.size() > 0 ? CL_SUCCESS : -1,
        "devices.size() > 0");
```

In OpenCL many operations are performed with respect to a given context. For example, buffer (1D regions of memory) and image (2D and 3D regions of memory) allocation are all context operations. But there are also device specific operations. For example, program compilation and kernel execution are on a per device basis, and for these a specific device handle is required.
Load Device Program

```cpp
std::ifstream file("lesson1_kernels.cl");
checkErr(file.is_open() ? CL_SUCCESS:-1, "lesson1_kernel.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,"");
checkErr(err, "Program::build()");
```
Kernel Objects

```
cl::Kernel kernel(program, "hello", &err);
checkErr(err, "Kernel::Kernel()");
err = kernel.setArg(0, outCL);
checkErr(err, "Kernel::setArg()");
```
Launching the Kernel

```cpp
cl::CommandQueue queue(context, devices[0], 0, &err);
checkErr(err, "CommandQueue::CommandQueue()");
cl::Event event;
err = queue.enqueueNDRangeKernel(
    kernel,
    cl::NullRange,
    cl::NDRange(hw.length() + 1),
    cl::NDRange(1, 1),
    NULL,
    &event);
checkErr(err,
    "CommandQueue::enqueueNDRangeKernel()");
```
event.wait();
err = queue.enqueueReadBuffer(
    outCL,
    CL_TRUE,
    0,
    hw.length()+1,
    outH);
checkErr(err,
    "ComamndQueue::enqueueReadBuffer()");
std::cout << outH;
return EXIT_SUCCESS;
}
The Kernel

#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable

__constant char hw[] = "Hello World\n";
__kernel void hello(__global char * out)
{
    size_t tid = get_global_id(0);
    out[tid] = hw[tid];
}