CS 677: Parallel Programming for Many-core Processors
Lecture 11

Instructor: Philippos Mordohai
Webpage: www.cs.stevens.edu/~mordohai
E-mail: Philippos.Mordohai@stevens.edu

Outline

- More CUDA Libraries
- OpenGL Interface
- Introduction to OpenCL

CUDA Libraries

Joseph Kider
University of Pennsylvania
CIS 565 - Spring 2011

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.gpuarray.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

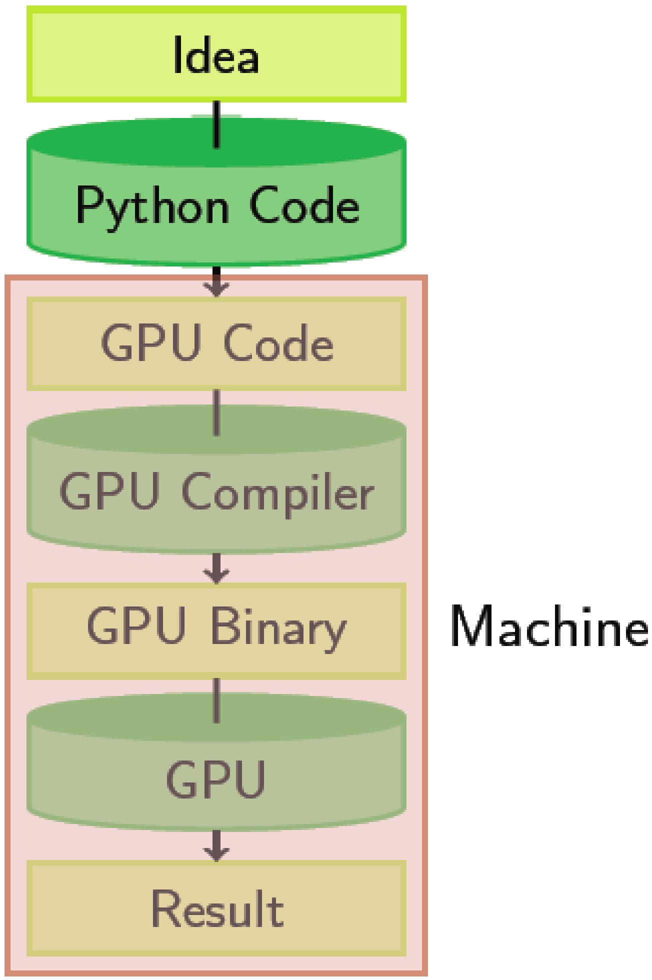
```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

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
```

Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *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

```
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
cudppScan(scanplan,
          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

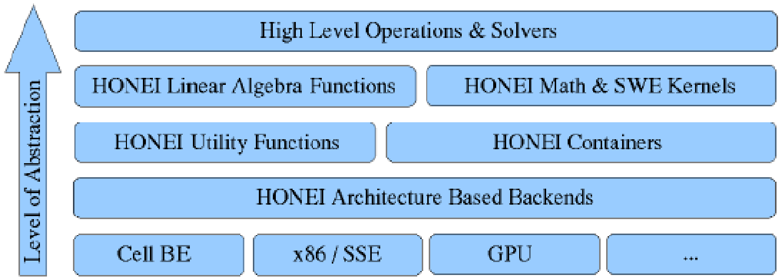
- CULA is EM Photonics' GPU-accelerated numerical linear algebra library that contains a growing list of LAPACK functions.
- LAPACK stands for Linear Algebra PACKage. It is an industry standard computational library that has been in development for over 15 years and provides a large number of routines 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(GLuintbuffObj);
   - 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, GLuintbuffObj);
   - 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:

```
// 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

```
float4 *dptr;
cudaGLMapBufferObject( (void**)&dptr, vbo));
```

3. Execute the kernel to compute values for dptr

```
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 Open GL

```
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

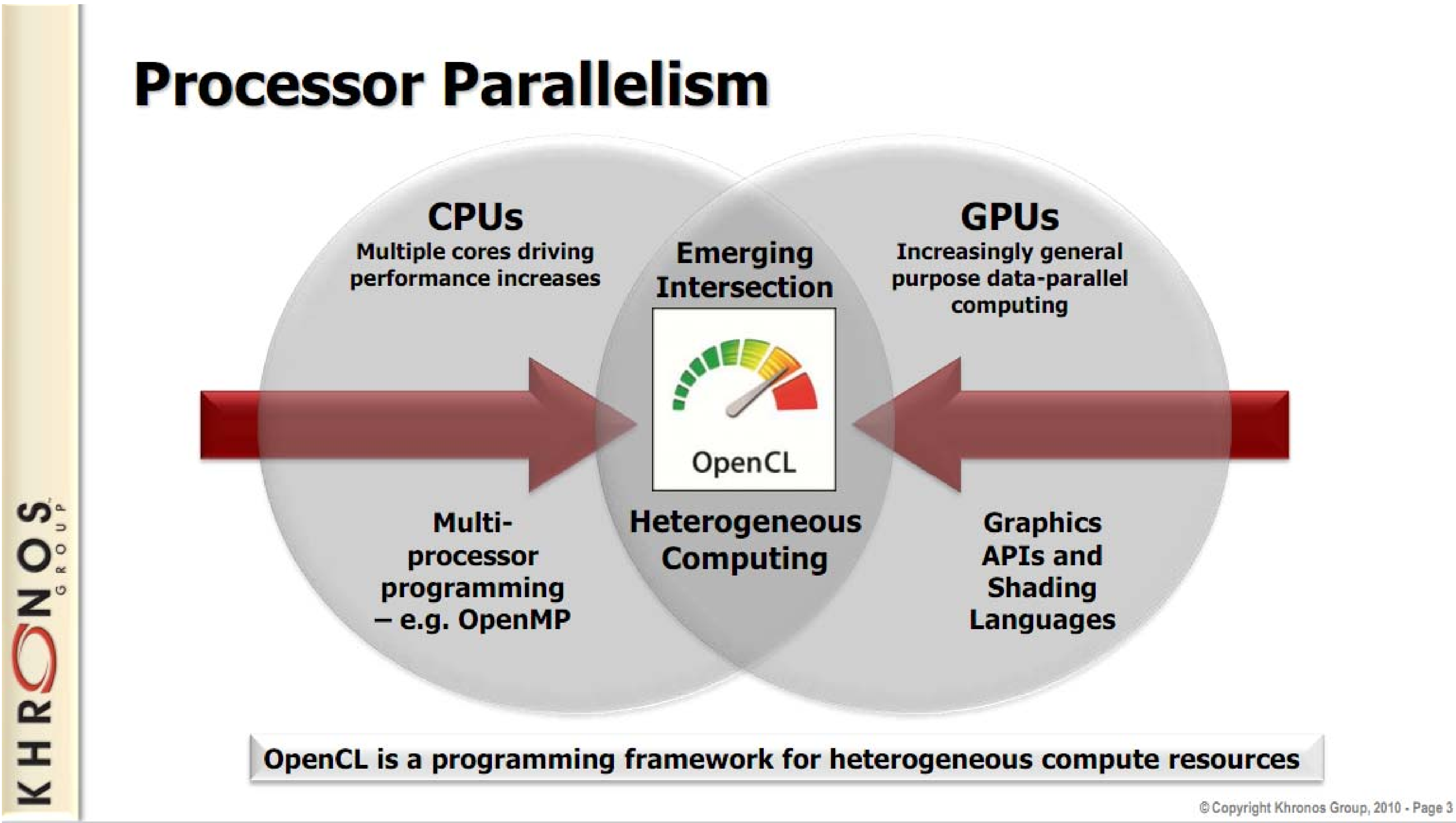


Image from: http://www.khronos.org/developers/library/overview/opencl_overview.pdf

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*?

Image from: http://developer.apple.com/softwarelicensing/agreements/opencl.html

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

Image from: http://www.khronos.org/developers/library/overview/opencl_overview.pdf

SPIR

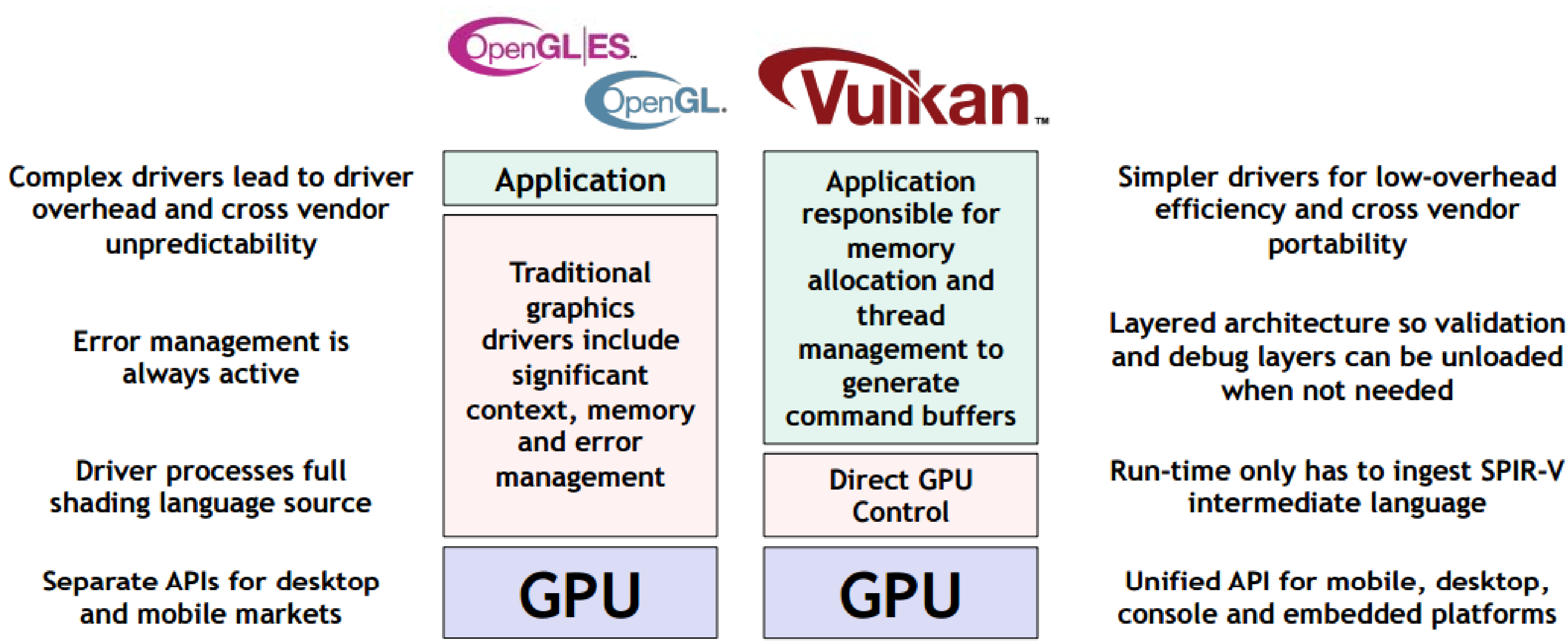
- Standard Portable Intermediate Representation
  - SPIR-V is first open standard, cross-API, intermediate language for natively representing parallel compute and graphics
  - Part of the core specification of:
    - OpenCL 2.1
    - the new Vulkan graphics and compute API

Vulkan

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

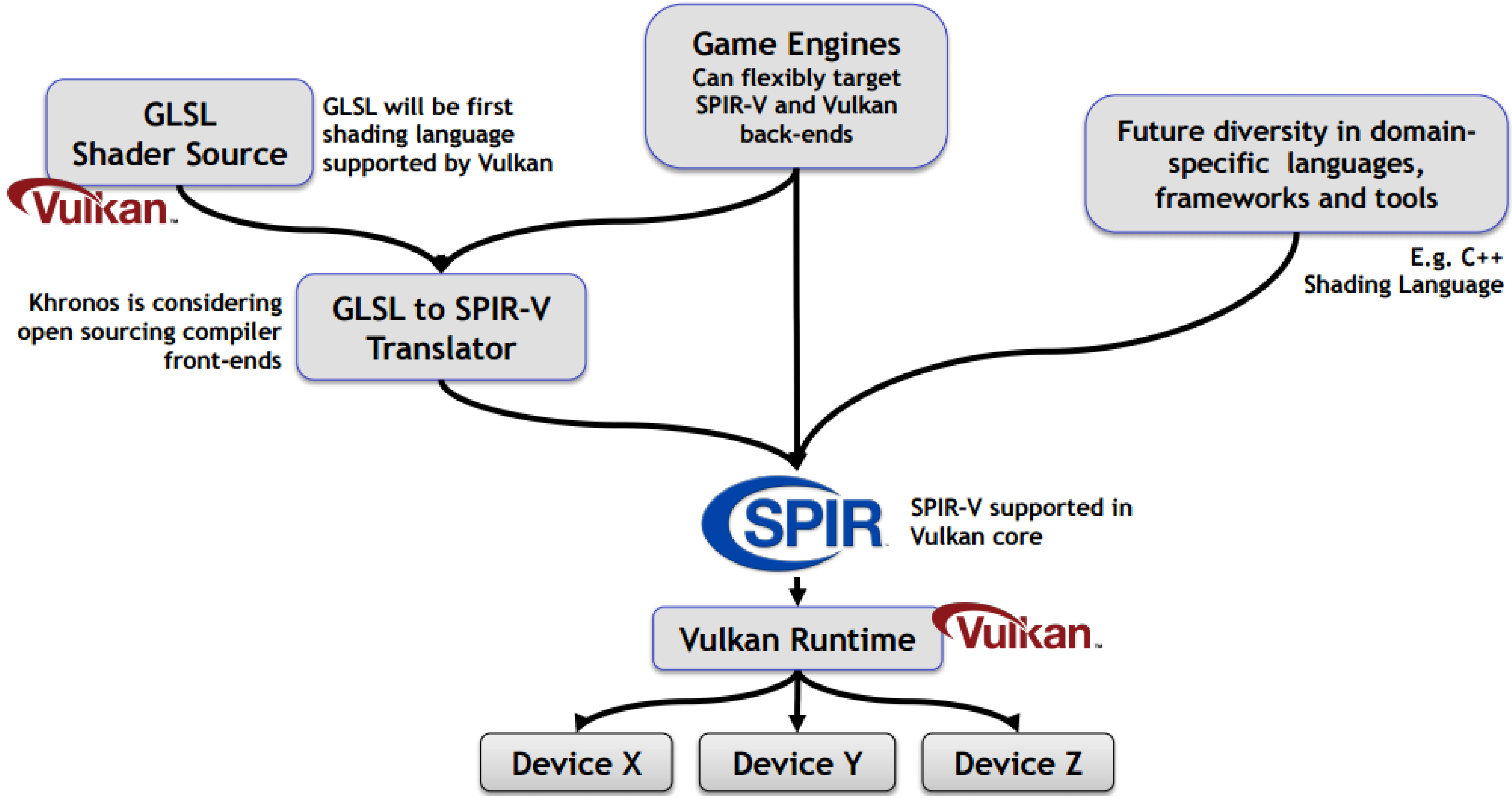
Source: www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf

Vulkan



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

Vulkan Language Ecosystem



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

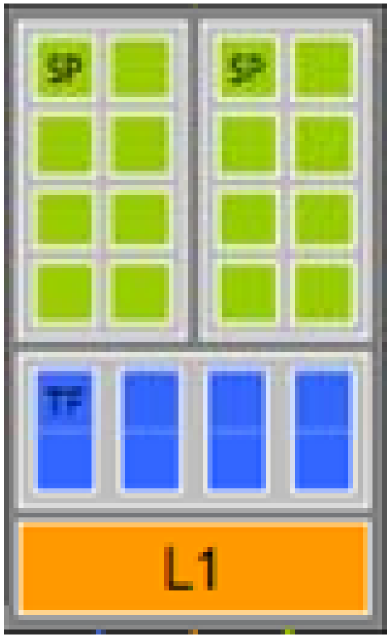
OpenCL and CUDA

- 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

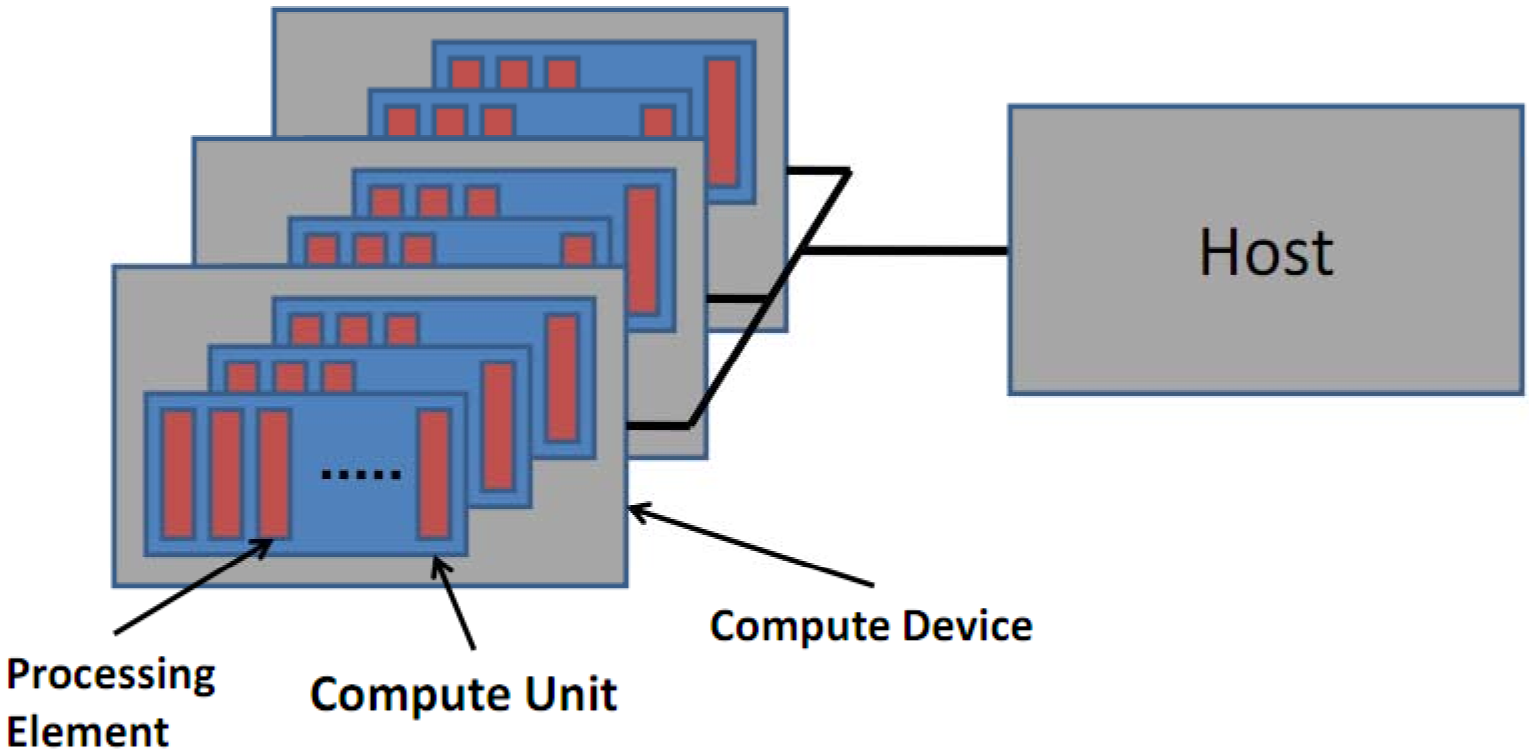


Image from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL and CUDA

CUDA	OpenCL
Kernel	Kernel
Host program	Host program
Thread	Work item
Block	Work group
Grid	NDRange (index space)

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()`

CUDA	OpenCL
`threadIdx.x`	`get_local_id(0)`
`blockIdx.x * blockDim.x + threadIdx.x`	`get_global_id(0)`

OpenCL and CUDA

CUDA	OpenCL
`gridDim.x`	`get_num_groups(0)`
`blockIdx.x`	`get_group_id(0)`
`blockDim.x`	`get_local_size(0)`
`gridDim.x * blockDim.x`	`get_global_size(0)`

OpenCL and CUDA

- Recall CUDA:

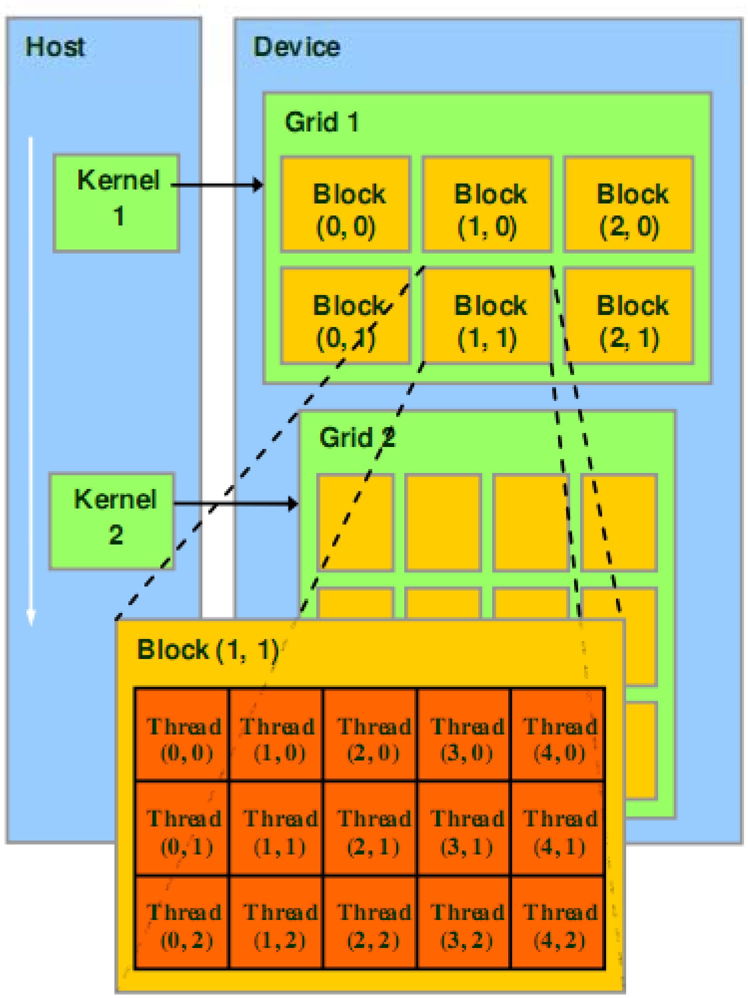
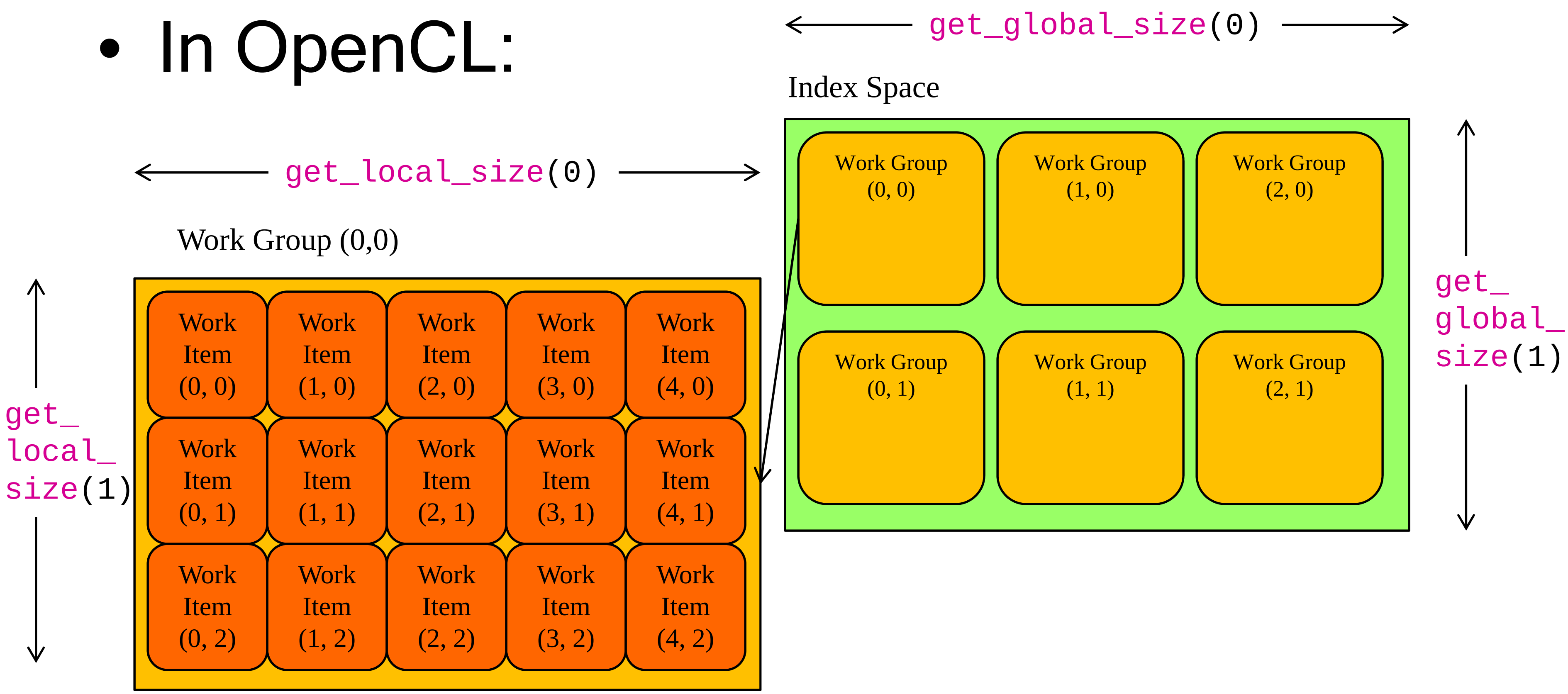


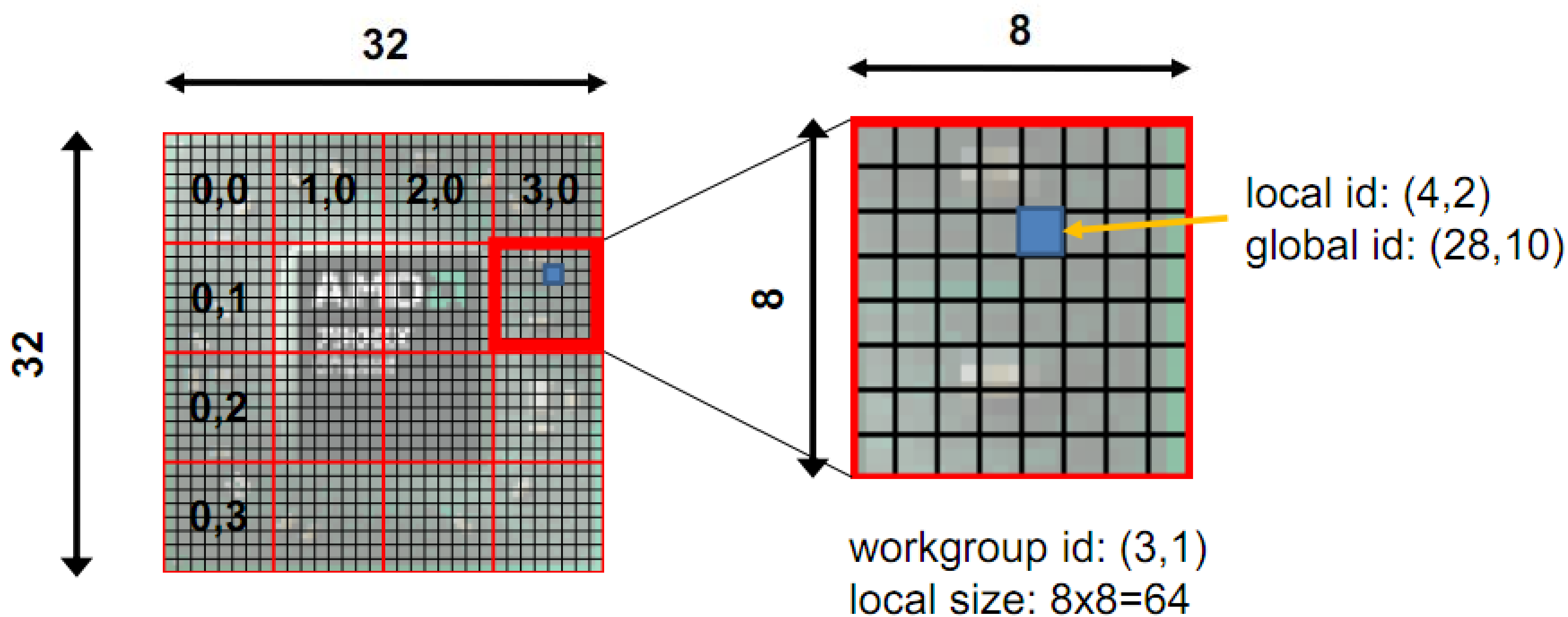
Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

OpenCL and CUDA

- In OpenCL:



Kernels: Work-item and Work-group Example

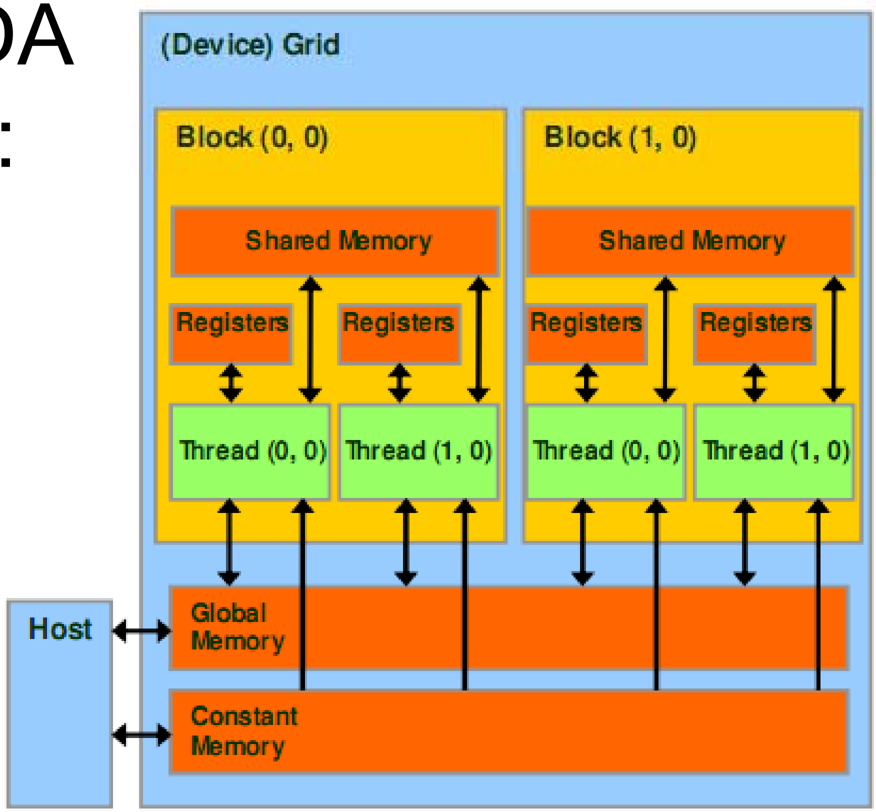


dimension: 2
global size: 32x32=1024
num of groups: 16

Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL and CUDA

- Recall the CUDA memory model:



OpenCL and CUDA

- In OpenCL:

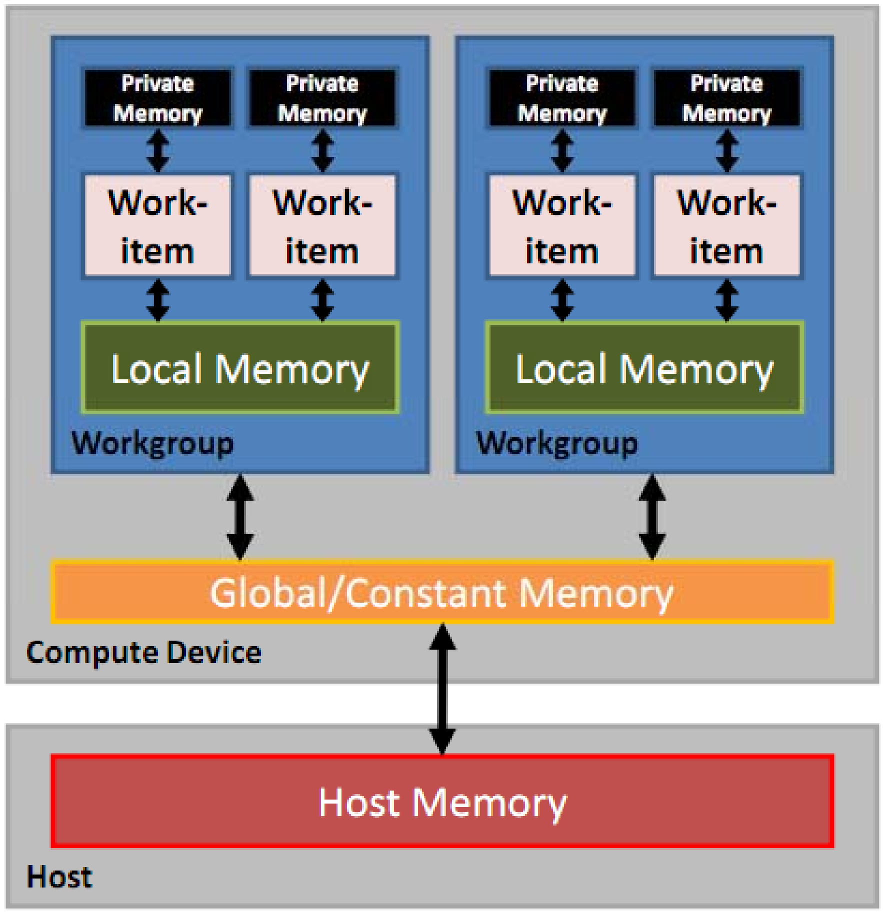
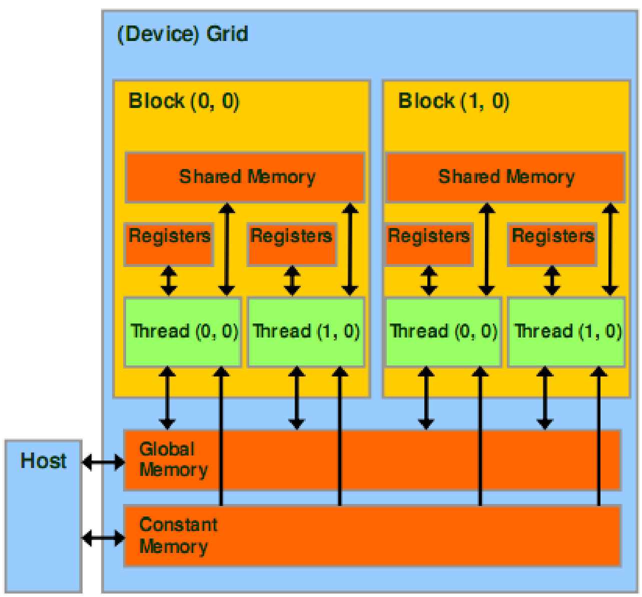
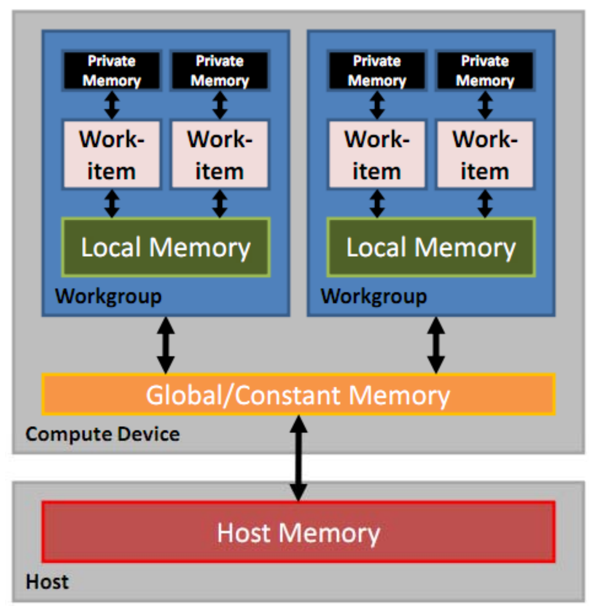


Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL and CUDA

CUDA	OpenCL
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory





OpenCL and CUDA

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

OpenCL and CUDA

CUDA	OpenCL
`__syncthreads()`	`__barrier()`

- 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

C for CUDA Terminology	OpenCL™ Terminology
__global__ function	__kernel function
__device__function	function (no qualifier required)
__constant__ variable declaration	__constant variable declaration
__device__ variable declaration	__global variable declaration
__shared__ variable declaration	__local variable declaration

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

Data Types

Scalar Type	Vector Type (n = 2, 4, 8, 16)	API Type for host app
char, uchar	charn, ucharn	cl_char<n>, cl_uchar<n>
short, ushort	shortn, ushortn	cl_short<n>, cl_ushort<n>
int, uint	intn, uintn	cl_int<n>, cl_uint<n>
long, ulong	longn, ulongn	cl_long<n>, cl_ulong<n>
float	floatn	cl_float<n>

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

Accessing Vector Components

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

```
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;
```


Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

Accessing Vector with Numeric Index

Vector components	Numeric indices
2 components	0, 1
4 components	0, 1, 2, 3
8 components	0, 1, 2, 3, 4, 5, 6, 7
16 components	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F

```
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
```


Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

Handy addressing of Vector Components

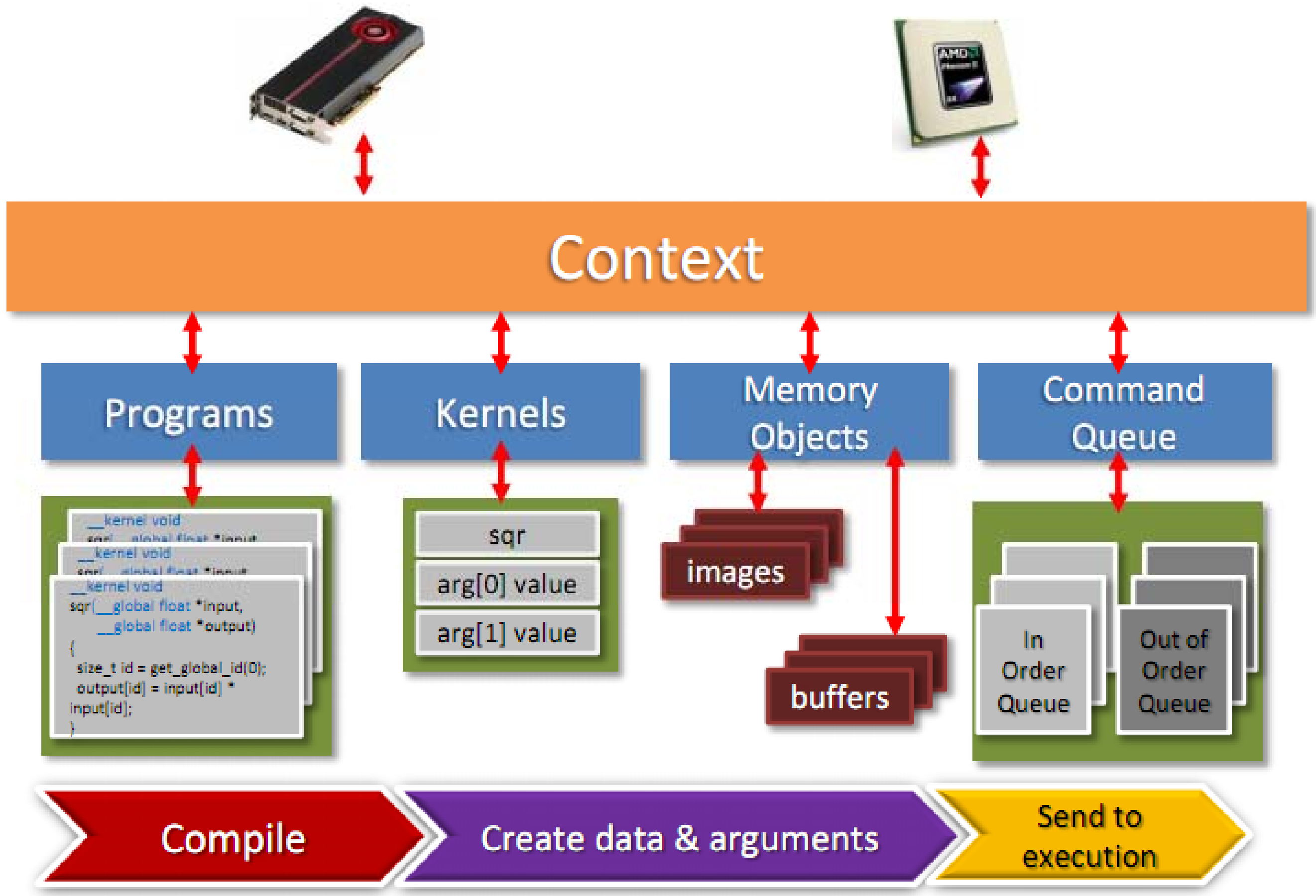
Vector access suffix	Returns
.lo	Returns the lower half of a vector
.hi	Returns the upper half of a vector
.odd	Returns the odd components of a vector
.even	Returns the even components of a vector

```
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)
```


Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

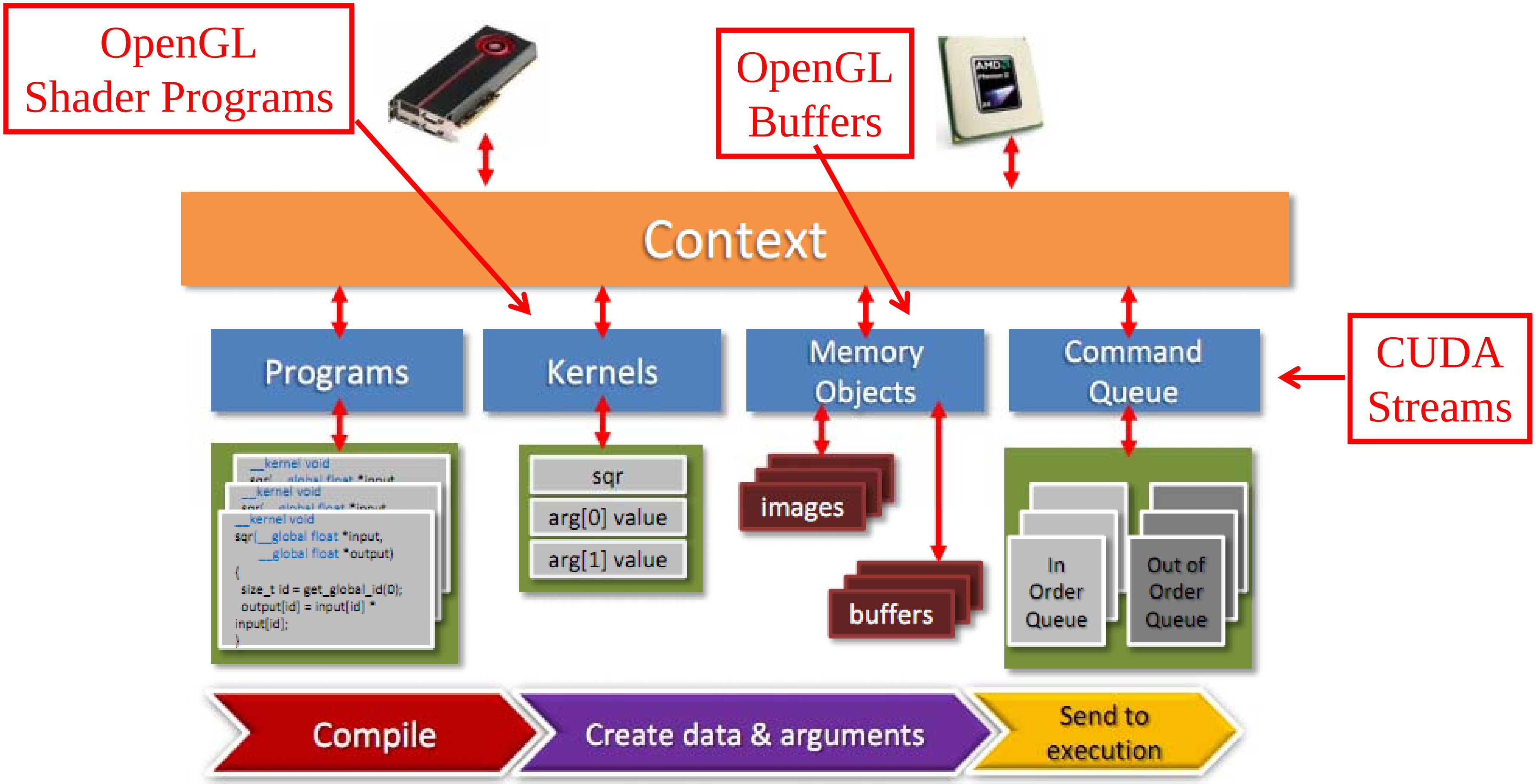
OpenCL™ Program Flow



AMD
The future is fusion

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL™ Program Flow



AMD
The future is fusion

Slide from: http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

OpenCL API

- Walkthrough OpenCL host code for running `vecAdd` kernel:

```
__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];
}
```

See NVIDIA OpenCL JumpStart Guide for full code example: ~~http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf~~

OpenCL API

```
// 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

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);
```

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

OpenCL API

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

OpenCL API

```
// 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

OpenCL API

```
// 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

OpenCL API

```
// 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

OpenCL API

```
// 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

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

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

Create buffers for kernel input. Read only in the kernel. Written by the host.

OpenCL API

```
hDeviceMemC = clCreateBuffer(hContext,
  CL_MEM_WRITE_ONLY,
   cnDimension * sizeof(cl_float),
   0, 0);
```

Create buffer for kernel output.

OpenCL API

```
// 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

Let OpenCL pick work group size

```
// 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);
```

Blocking read

clEnqueueNDRangeKernel

```
cl_int clEnqueueNDRangeKernel (
        cl_command_queue command_queue,
        cl_kernel kernel,
        cl_uint work_dim,
        const size_t *global_work_offset,
        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)
```

<=3

NULL

global_work_size must be divisible by local_work_size

OpenCL API

```
delete [] pA;
delete [] pB;
delete [] pC;
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
```

CUDA Pointer Traversal

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

OpenCL Pointer Traversal

```
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

Header Files

```
#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");
```

Error Handling

```
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

```
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

```
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

```
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

```
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

```
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,
  "ComamndQueue::enqueueNDRangeKernel()");
```

Reading the Results

```
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];
}
```