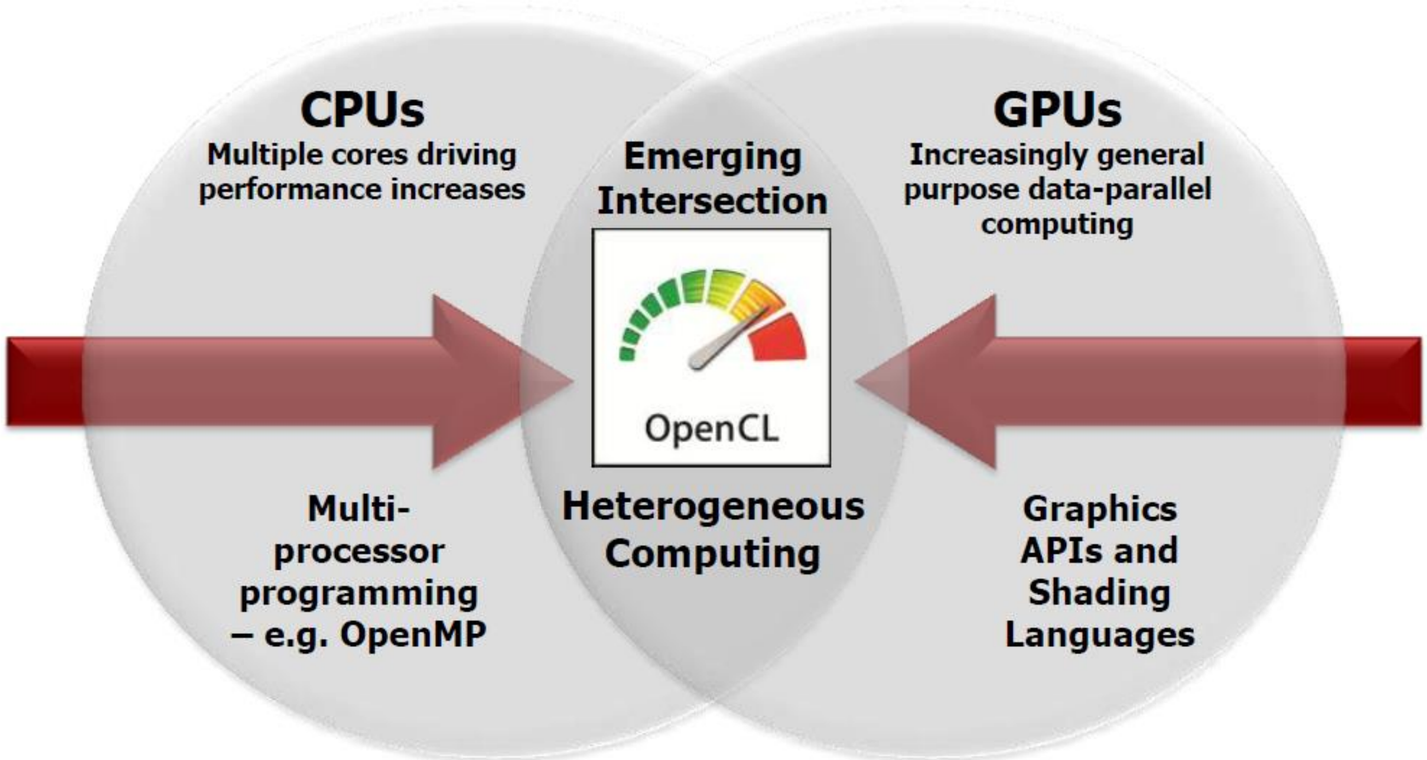
Introduction to OpenCL

Dr.Alptekin Temizel

atemizel@metu.edu.tr

INFORMATICS INSTITUTE

Processor Parallelism



OpenCL is a programming framework for heterogeneous compute resources

OpenCL Ecosystem

Implementers
Desktop/Mobile/Embedded/FPGA

SYCL Single Source C++ Programming

OpenCL Core API and Language Specs

SPIR Portable Kernel Intermediate Language

Working Group Members
Apps/Tools/Tests/Courseware

OpenCL 2.2 - Top to Bottom C++

Single Source C++ Programming
Full support for features in C++14-based Kernel Language

API and Language Specs
Brings C++14-based Kernel Language into core specification

Portable Kernel Intermediate Language
Support for C++14-based kernel language e.g. constructors/destructors

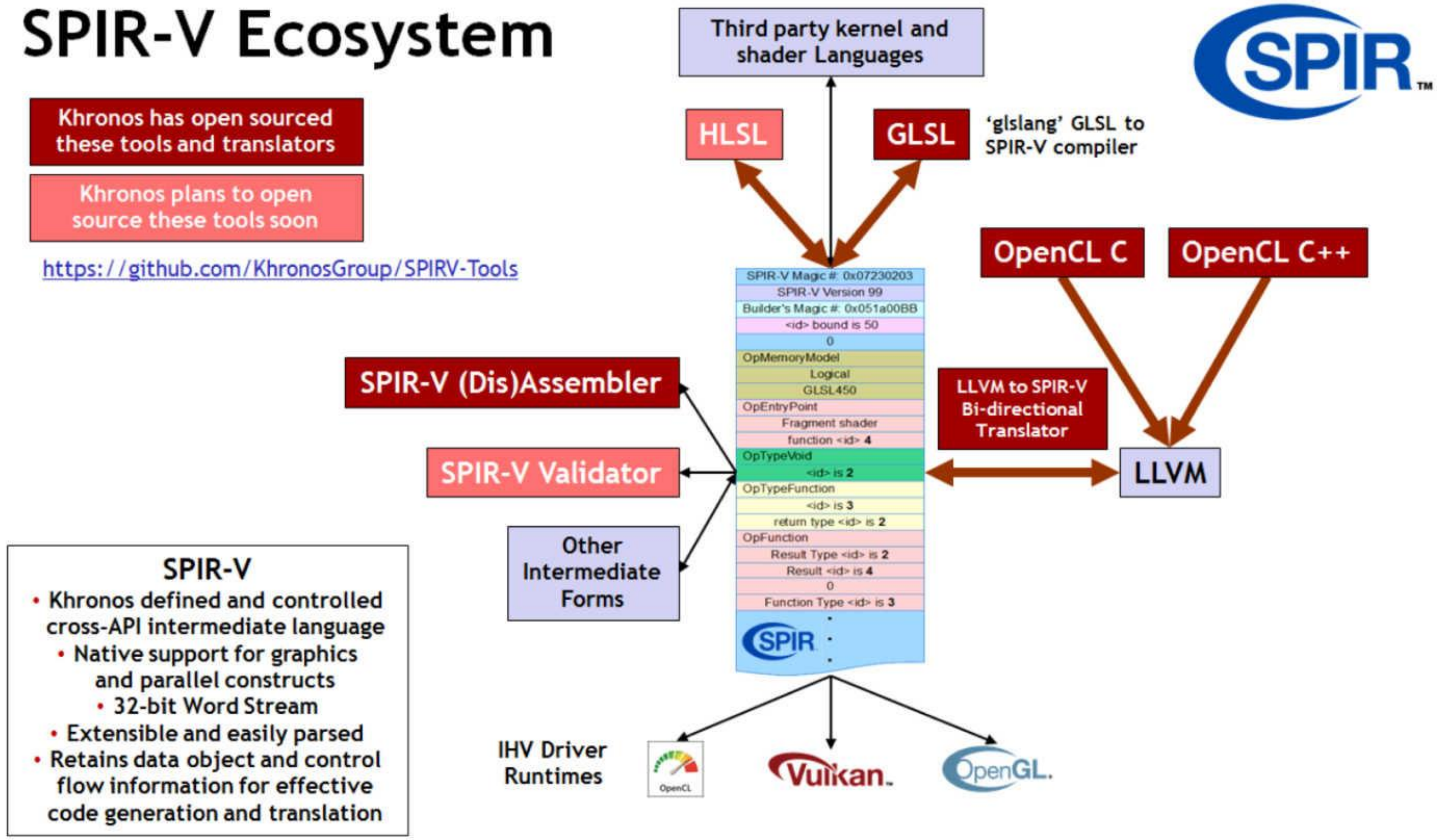
OpenCL C++ Kernel Language
SPIR-V 1.2 with C++ support
SYCL 2.2 single source C++
Pipes
Efficient device-scope communication between kernels
Code Gen Optimizations:
- Specialization constants at SPIR-V compilation time
- Constructors and destructors of program scope global objects
- User callbacks can be set at program release time

Release	Interval to next	Features
Dec08 OpenCL 1.0 Specification	18 months	
Jun10 OpenCL 1.1 Specification	18 months	3-component vectors; Additional image formats; Multiple hosts and devices; Buffer region operations; Enhanced event-driven execution; Additional OpenCL C built-ins; Improved OpenGL data/event interop
Nov11 OpenCL 1.2 Specification	24 months	Device partitioning; Separate compilation and linking; Enhanced image support; Built-in kernels / custom devices; Enhanced DX and OpenGL Interop
Nov13 OpenCL 2.0 Specification	24 months	Shared Virtual Memory; On-device dispatch; Generic Address Space; Enhanced Image Support; C11 Atomics; Pipes; Android ICD
Nov15 OpenCL 2.1 Specification	18 months	SPIR-V in Core; Subgroups into core; Subgroup query operations; clCloneKernel; Low-latency device timer queries
May17 OpenCL 2.2 Specification		

OpenCL 2.0

- November 18, 2013: the Khronos Group announced the ratification and public release of the finalized OpenCL 2.0 specification.
- August 29, 2014: Intel releases HD Graphics 5300 driver that supports OpenCL 2.0.
- September 25, 2014: AMD releases Catalyst 14.41 RC1, which includes an OpenCL 2.0 driver.

OpenCL 2.2



SPIR-V is the first open standard, cross-API intermediate language for natively representing parallel compute and graphics.

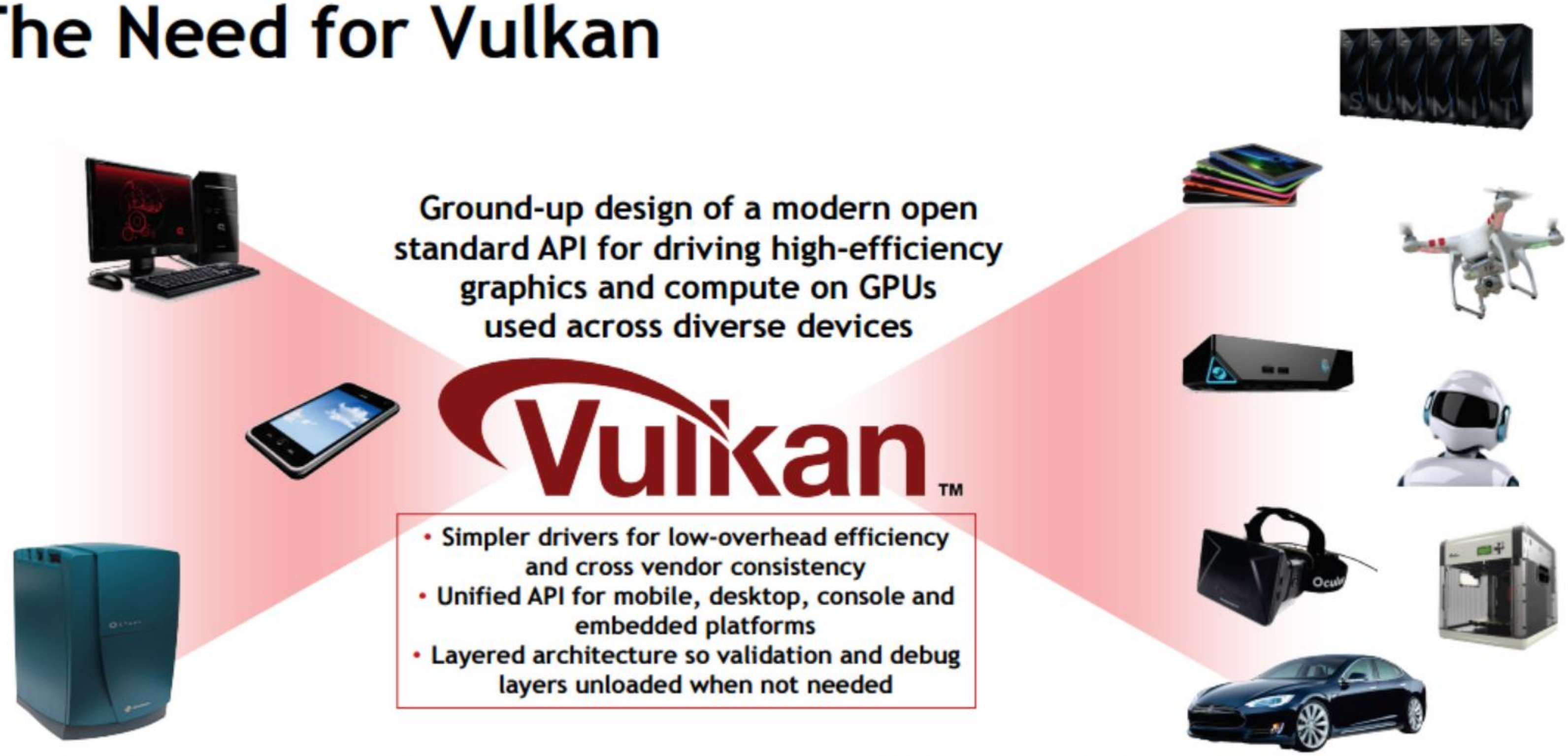
Incorporated as part of the core specification of both OpenCL, OpenGL and the new Vulkan graphics and compute API.

WebCL

- WebCL 1.0 specification released on March 19, 2014
- It is a JavaScript binding to OpenCL for heterogeneous parallel computing within a web browser without the use of plug-ins.
- Though no browsers natively support WebCL yet!
- WebCL allows web applications to utilize multi-core CPUs and GPUs to make computationally intensive programs feasible in the browser, e.g. physics engines, canvas element and video editing.

Vulkan

The Need for Vulkan

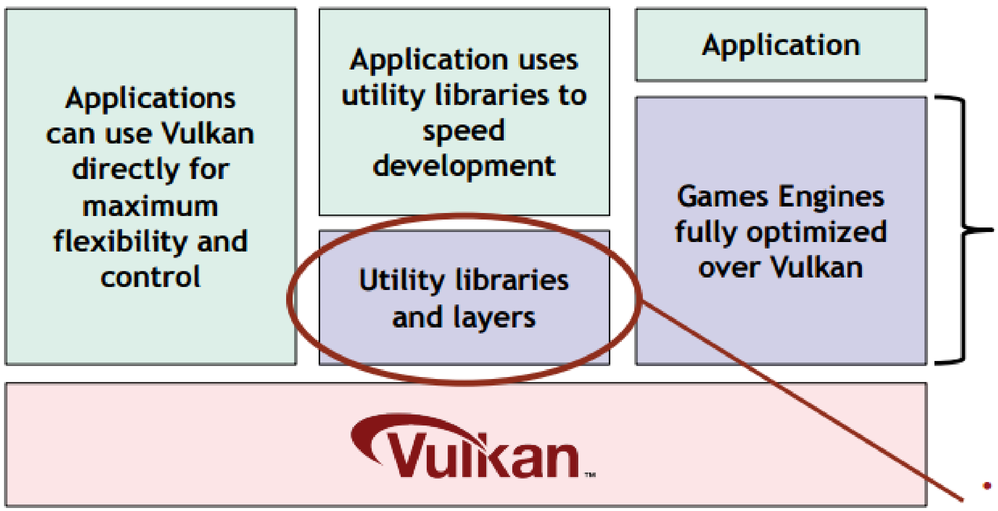


In the twenty two years since OpenGL was invented - the architecture of GPUs and platforms has changed radically

GPUs being used for graphics, compute and vision processing on a rapidly *increasing* diversity of platforms - *increasing* the need for cross-platform standards

Vulkan

Vulkan Layered Ecosystem

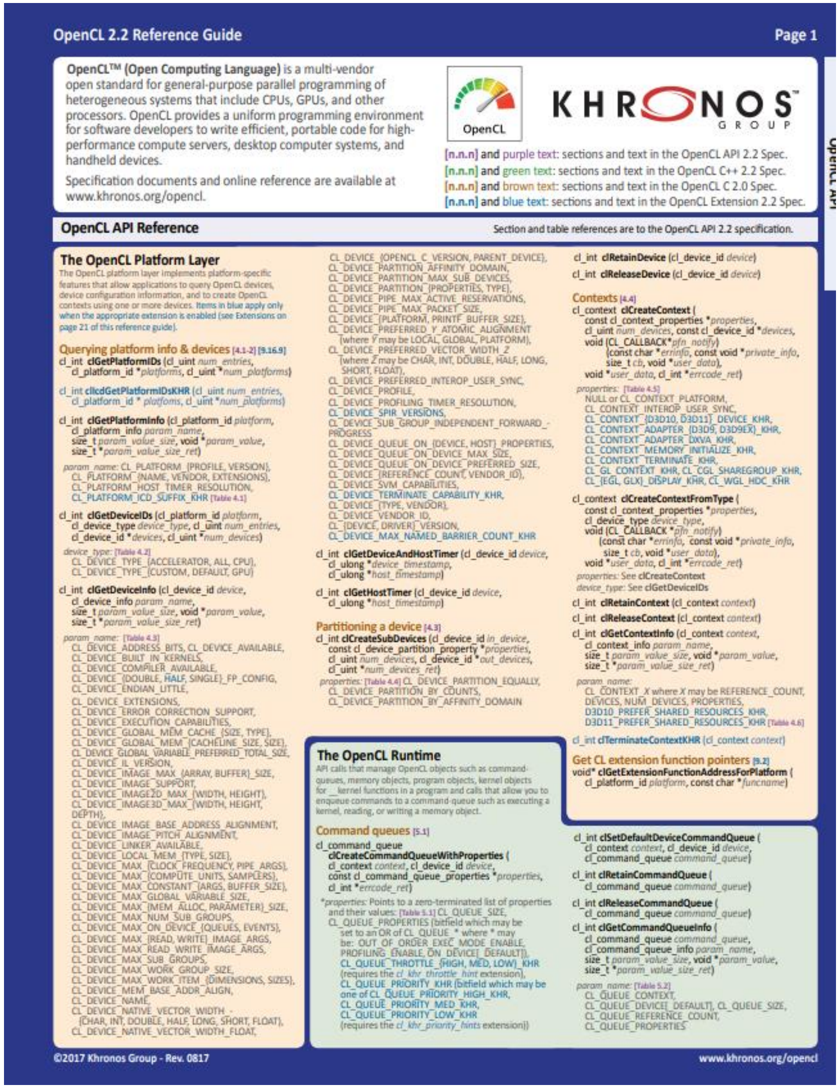


Developers can choose at which level to use the Vulkan Ecosystem

The industry's leading games and engine vendors are participating in the Vulkan working group

Rich Area for Innovation

- Many utilities and layers will be in open source
- Layers to ease transition from OpenGL
- Domain specific flexibility


OpenCL 2.2 Reference Guide

Page 1

OpenCL™ (Open Computing Language) is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs, and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems, and handheld devices.

Specification documents and online reference are available at www.khronos.org/opencl.

[n.n.n] and purple text: sections and text in the OpenCL API 2.2 Spec.
[n.n.n] and green text: sections and text in the OpenCL C++ 2.2 Spec.
[n.n.n] and brown text: sections and text in the OpenCL C 2.0 Spec.
[n.n.n] and blue text: sections and text in the OpenCL Extension 2.2 Spec.

OpenCL API Reference

Section and table references are to the OpenCL API 2.2 specification.

The OpenCL Platform Layer

Querying platform info & devices

Partitioning a device

Contexts

Get CL extension function pointers

The OpenCL Runtime

Command queues

©2017 Khronos Group - Rev. 0817

www.khronos.org/opencl

OpenCL Reference Card

https://www.khronos.org/files/opencl22-reference-guide.pdf

OpenCL Desktop Implementations

- http://developer.amd.com/zones/OpenCLZone/
- http://software.intel.com/en-us/articles/opencl-sdk/
- http://developer.nvidia.com/opencl

OpenCL on FPGAs

- Altera Stratix V devices now support OpenCL
- OpenCL on FPGAs for GPU Programmers:

http://design.altera.com/Acceleware_OpenCL_FPGA_WP

The BIG Idea behind OpenCL

- **OpenCL execution model …**
  - Define N-dimensional computation domain
  - Execute a kernel at each point in computation domain

Traditional loops

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

Data Parallel OpenCL

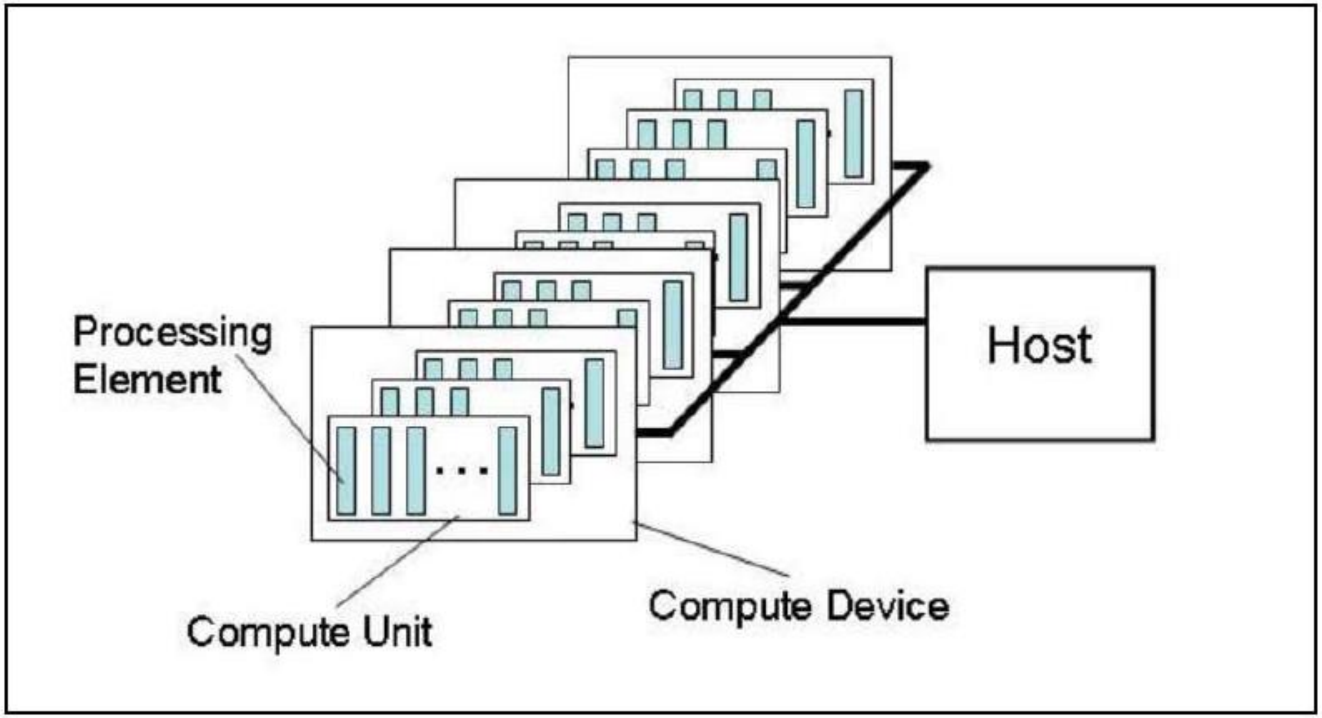
```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

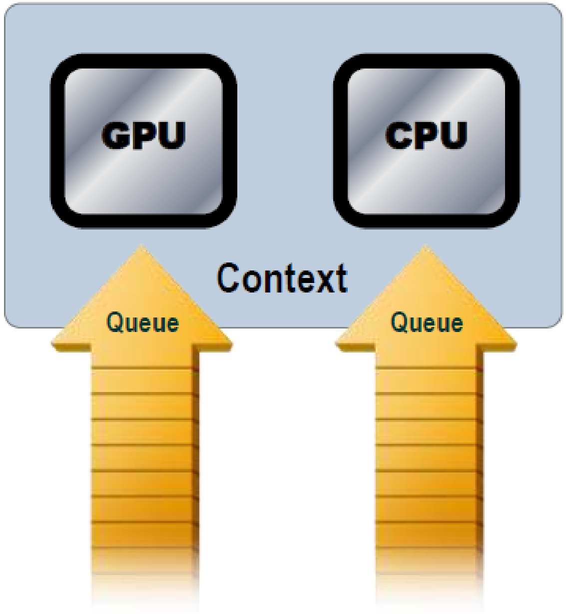
OpenCL Platform Model

- **One Host + one or more Compute Devices**
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements



OpenCL Execution Model

- **OpenCL application runs on a host which submits work to the compute devices**
  - **Context**: The environment within which work-items executes … includes devices and their memories and command queues
  - **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
  - **Kernel**: the code for a work item. Basically a C function
  - **Work item**: the basic unit of work on an OpenCL device
- **Applications queue kernel execution**
  - Executed in-order or out-of-order



Compilation Model

- **OpenCL™ uses Dynamic/Runtime compilation model (like OpenGL®):**
  1. The code is complied 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.

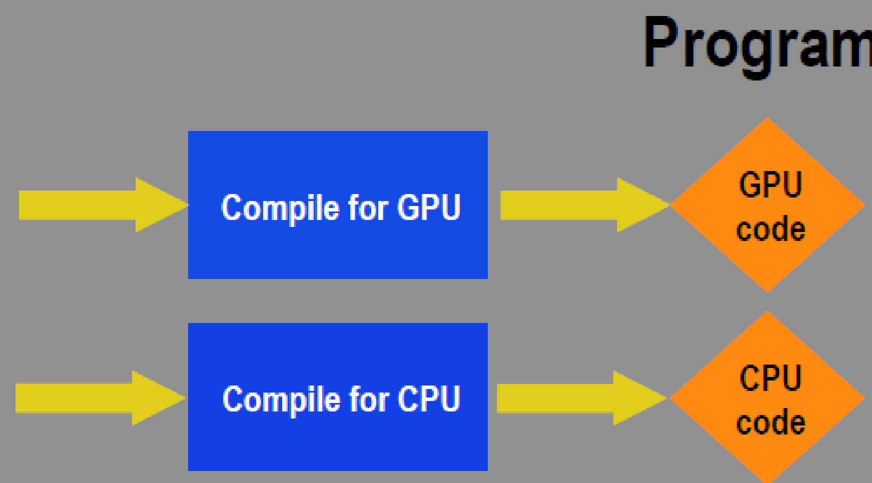
- **In dynamic compilation, step 1 is done usually only once, and the IR is stored.**
- **The App loads the IR and performs step 2 during the App's runtime (hence the term…)**

Executing Code

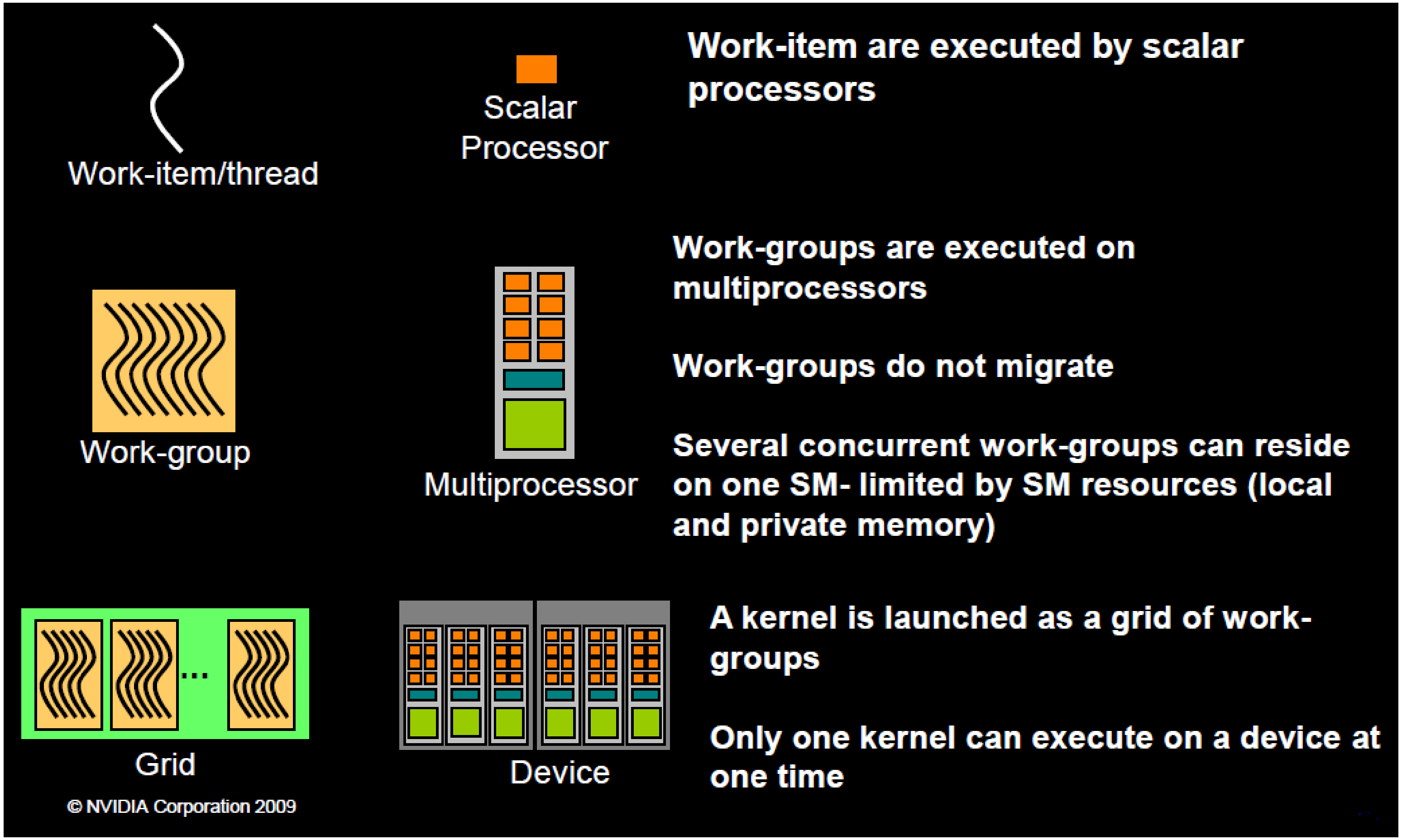
- Programs build executable code for multiple devices
- Execute the same code on different devices

Kernel Code

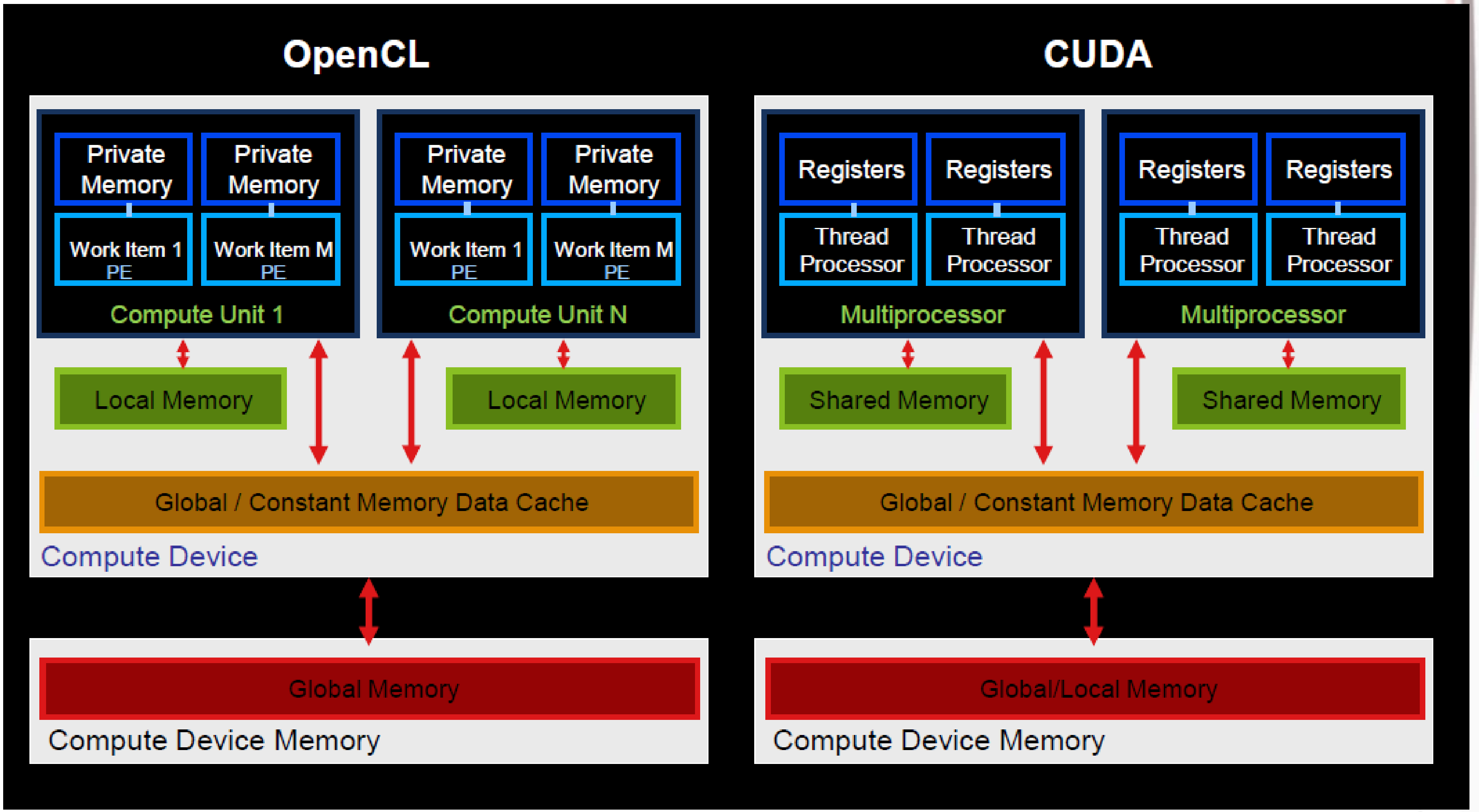
```
kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                               (int2)(width-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



OpenCL Terms



OpenCL vs. CUDA



OpenCL to CUDA Model Mapping

OpenCL Parallelism Concept	CUDA Equivalent
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

Mapping of OpenCL Indices to CUDA

OpenCL API Call	Explanation	CUDA Equivalent
get_local_id(0)	local index of the work item within the work group in the x dimension	threadIdx.x
get_global_id(0);	global index of the work item in the x dimension	blockIdx.x * blockDim.x + threadIdx.x
get_global_size(0);	size of NDRange in the x dimension	gridDim.x * blockDim.x
get_local_size(0);	Size of each work group in the x dimension	blockDim.x

Basic Program Structure

- **Host program**
  - Create contexts
  - Query compute devices
  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources
- **Compute Kernel (runs on device)**
  - C code with some restrictions and extensions

How to Use OpenCL (1)

- Open an OpenCL context,
- Get and select the devices to execute on,
- Create a command queue to accept the execution and memory requests,
- Allocate OpenCL memory objects to hold the inputs and outputs for the compute kernel,
- **Online compile** and build the compute kernel code,
- Set up the arguments and execution domain,
- Kick off compute kernel execution,
- Collect the results,
- Clean up.

How to Use OpenCL (2)

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                                     NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
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);

// create the program
program = clCreateProgramWithSource(context, 1, &program_source, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
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));

// set work-item dimensions
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);
```

OpenCL Language Highlights

- **Function qualifiers**
  - "__kernel" qualifier declares a function as a kernel
- **Address space qualifiers**
  - __global, __local, __constant, __private
- **Work-item functions**
  - get_work_dim()
  - get_global_id(), get_local_id(), get_group_id(), get_local_size()
- **Image functions**
  - Images must be accessed through built-in functions
  - Reads/writes performed through sampler objects from host or defined in source
- **Synchronization functions**
  - Barriers - All Work Items within a Work Group must execute the barrier function before any Work Item in the Work Group can continue

Error Check

- Most OpenCL API either return an error code (type ***cl_int***) or they store the error code at a location passed by the user as a parameter to the call.

  It is important to check its behavior correctly in the case of error.

```
inline void checkErr(cl_int err, const char * name)
{
        if (err != CL_SUCCESS) {
          std::cerr << "ERROR: " << name
          << " (" << err << ")" << std::endl;
          exit(EXIT_FAILURE);
        }
}
```

clGetDeviceIDs

- Query the system and obtain list of devices available on a platform:

cl_int **clGetDeviceIDs**(
cl_platform_id platform,
cl_device_type device_type,
cl_uint num_entries,
cl_device_id *devices,
cl_uint *num_devices)

cl_device_type	Description
CL_DEVICE_TYPE_CPU	An OpenCL device that is the host processor.
CL_DEVICE_TYPE_GPU	An OpenCL device that is a GPU.
CL_DEVICE_TYPE_ACCELERATOR	Dedicated OpenCL accelerators (for example the IBM CELL Blade).
CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the system.
CL_DEVICE_TYPE_ALL	All OpenCL devices available in the system.

A list of OpenCL devices found.

clGetDeviceIDs

```
clGetDeviceIDs(
    cpPlatform,
    CL_DEVICE_TYPE_GPU,
    1,
    &cdDevice,
    NULL);
```

clGetDeviceInfo

Get information about an OpenCL device.

cl_int **clGetDeviceInfo**(

cl_device_id device,

cl_device_info param_name,

size_t param_value_size,

*void **param_value,

*size_t **param_value_size_ret)

Device Information

cl_device_info

CL_DEVICE_ADDRESS_BITS
CL_DEVICE_AVAILABLE
CL_DEVICE_COMPILER_AVAILABLE
CL_DEVICE_ENDIAN_LITTLE
CL_DEVICE_ERROR_CORRECTION_SUPPORT
CL_DEVICE_EXECUTION_CAPABILITIES
CL_DEVICE_EXTENSIONS
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE
CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE
CL_DEVICE_GLOBAL_MEM_SIZE
CL_DEVICE_IMAGE_SUPPORT
CL_DEVICE_IMAGE2D_MAX_HEIGHT
CL_DEVICE_IMAGE2D_MAX_WIDTH
CL_DEVICE_IMAGE3D_MAX_DEPTH
CL_DEVICE_IMAGE3D_MAX_HEIGHT
CL_DEVICE_IMAGE3D_MAX_WIDTH
CL_DEVICE_LOCAL_MEM_SIZE
CL_DEVICE_LOCAL_MEM_TYPE
...

clCreateContext

An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context.

cl_context **clCreateContext**(

cl_context_properties *properties,

cl_uint num_devices,

const cl_device_id *devices,

void *pfn_notify

void *user_data,

cl_int *errcode_ret)

clCreateContext

cxGPUContext = clCreateContext(0, **1, &cdDevice, NULL, NULL, &ciErr1);**

clCreateContextFromType

Create an OpenCL context from a device type that identifies the specific device(s) to use.

cl_context **clCreateContextFromType** (

const cl_context_properties *properties,

cl_device_type device_type,

*void (CL_CALLBACK ** pfn_notify),

*void ** user_data,

*cl_int ** errcode_ret)

clCreateCommandQueue

Create a command-queue on a specific device.

cl_command_queue **clCreateCommandQueue**(
cl_context context,
cl_device_id device,
cl_command_queue_properties properties,
cl_int **errcode_ret*)

Command-Queue Properties	Description
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	Determines whether the commands queued in the command-queue are executed in-order or out-of-order.
CL_QUEUE_PROFILING_ENABLE	Enable or disable profiling of commands in the command-queue.

clCreateBuffer

Creates a buffer object.

```
cl_mem clCreateBuffer (
    cl_context context,
    cl_mem_flags flags,
    size_t size,
    void *host_ptr,
    cl_int *errcode_ret)
```

cl_mem_flags

cl_mem_flags	Description
CL_MEM_READ_WRITE	**Memory object will be read and written by a kernel. This is the default.**
CL_MEM_WRITE_ONLY	Memory object will be written but not read by a kernel.
CL_MEM_READ_ONLY	Memory object is a read-only memory object when used inside a kernel.
CL_MEM_USE_HOST_PTR	This flag is valid only if host_ptr is not NULL. If specified, it indicates that the application wants the OpenCL implementation to use memory referenced by host_ptr as the storage bits for the memory object. OpenCL implementations are allowed to cache the buffer contents pointed to by host_ptr in device memory. This cached copy can be used when kernels are executed on a device. The result of OpenCL commands that operate on multiple buffer objects created with the same host_ptr or overlapping host regions is considered to be undefined.
CL_MEM_ALLOC_HOST_PTR	This flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory. CL_MEM_ALLOC_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.
CL_MEM_COPY_HOST_PTR	This flag is valid only if host_ptr is not NULL. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by host_ptr. CL_MEM_COPY_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive. CL_MEM_COPY_HOST_PTR can be used with CL_MEM_ALLOC_HOST_PTR to initialize the contents of the cl_mem object allocated using host-accessible (e.g. PCIe) memory.

Source Code

You can put your code into a string:

```
const char* OpenCLSource[] = {
"__kernel void VectorAdd(__global int* c, __global int* a,__global int* b)",
"{",
" // Index of the elements to add \n",
" unsigned int n = get_global_id(0);",
" // Sum the n'th element of vectors a and b and store in c \n",
" c[n] = a[n] + b[n];",
"}"
};
```

But this could be difficult when your source code gets longer...

Alternatively you can put your code into a file and load it into a string.

clCreateProgramWithSource

Creates a program object for a context, and loads the source code specified by the text strings in the strings array into the program object.

cl_program **clCreateProgramWithSource** (

cl_context context,

cl_uint count,

*const char ***strings,

*const size_t **lengths,

*cl_int **errcode_ret)

clCreateProgramWithBinary

Creates a program object for a context, and loads the binary bits specified by binary into the program object.

cl_program **clCreateProgramWithBinary** (

cl_context context,

cl_uint num_devices,

*const cl_device_id **device_list,

*const size_t **lengths,

*const unsigned char ***binaries,

*cl_int **binary_status,

*cl_int **errcode_ret)

clBuildProgram

Builds (compiles and links) a program executable from the program source or binary.

cl_int **clBuildProgram** (

cl_program program,

cl_uint num_devices,

*const cl_device_id ** device_list,

*const char ** options,

void (CL_CALLBACK *pfn_notify)(cl_program program, void *user_data),

*void ** user_data)

clCreateKernel

Creates a kernel object.

```
cl_kernel clCreateKernel (
    cl_program  program,
    const char *kernel_name,
    cl_int *errcode_ret)
```

A kernel is a function declared in a program. A kernel is identified by the **__kernel** qualifier applied to any function in a program. A kernel object encapsulates the specific __kernel function declared in a program and the argument values to be used when executing this __kernel function.

clSetKernelArg

Used to set the argument value for a specific argument of a kernel.

cl_int **clSetKernelArg** (

cl_kernel kernel,

cl_uint arg_index,

size_t arg_size,

*const void **arg_value)

clEnqueueNDRangeKernel

Enqueues a command to execute a kernel on a device.

```
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)
```

Launching Kernels

```
clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
NULL, global_work_size, NULL, 0, NULL, NULL);
```

```
kernel<<<grid, block, 0, 0>>>(...);
```

clEnqueueWriteBuffer

Enqueue commands to write into a buffer object from host memory (host to device copy).

cl_int **clEnqueueWriteBuffer** (
cl_command_queue command_queue,
cl_mem buffer,
cl_bool blocking_write,
size_t offset,
size_t cb,
*const void **ptr,
cl_uint num_events_in_wait_list,
*const cl_event **event_wait_list,
*cl_event **event)

clEnqueueReadBuffer

Enqueue commands to read from a buffer object to host memory. (device to host copy)

cl_int **clEnqueueReadBuffer** (
cl_command_queue command_queue,
cl_mem buffer,
cl_bool blocking_read,
size_t offset,
size_t cb,
*void **ptr,
cl_uint num_events_in_wait_list,
*const cl_event **event_wait_list,
*cl_event **event)

clFinish

Blocks until all previously queued OpenCL commands in a command-queue are issued to the associated device and have completed.

cl_int **clFinish** (*cl_command_queue* command_queue)

clFinish is also a synchronization point.

House cleaning

- clFinish
- clReleaseKernel
- clReleaseProgram
- clReleaseCommandQueue
- clReleaseContext

Sample Kernel Code-1D

- Source code for the computation kernel, stored in text file (read from file and compiled at run time, e.g. during app. init)

CUDA version

int idx = blockIdx.x*blockDim.x + threadIdx.x;

```
int idx = get_global_id(0);
// bound check
//(equivalent to the limit on a 'for' loop for serial C code
if (idx >= iNumElements)
{
   return;
}
  // add the vector elements
   c[idx] = a[idx] + b[idx];
}
```

Sample Kernel Code – 2D

```
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);
}
```

Kernel Execution Configuration (1)

- Host program launches kernel in index space called NDRange
  - NDRange ("N-Dimensional Range") is a multitude of kernel instances arranged into 1, 2 or 3 dimensions
  - A single kernel instance in the index space is called a *Work Item*
    - Each Work Item executes same compute kernel (on different data)
    - Work Items have unique global IDs from the index space
- Work-items are further grouped into *Work Groups*
  - Work Groups have a unique Work Group ID
  - Work Items have a unique Local ID within a Work Group
- Analogous to a C loop that calls a function many times. Except all iterations are called simultaneously & executed in parallel

Kernel Execution Configuration (2)

- The total number of elements (indexes) in the launch domain is called the ***global*** work size; individual elements are known as ***work-items***. ***Work-items*** can be grouped into ***work-groups*** when communication between ***work-items*** is required.
- size_t szGlobalWorkSize;

1D variable for total # of work items

- size_t szLocalWorkSize;

1D variable for # of work items in the work group

Kernel Execution Configuration (3)

- # of work-groups > # of SM
  - Each SM has at least one work-group to execute
- # of work-groups / # of SM > 2
  - Multi work-groups can run on a SM
- Work on another work-group if one work-group is waiting on barrier
  - # of work-groups / # of SM > 100 to scale well to future device

Kernel Execution Configuration (4)

NVIDIA Specific Constraints:

- The number of work-items per work-group should be a multiple of 32 (warp size)
- Want as many warps running as possible to hide latencies
- Minimum: 64
- Larger, e.g. 256 may be better
- Depends on the problem, do experiments!

Kernel Execution Configuration (5)

// set Local work size dimensions

szLocalWorkSize = 256;

// set Global work size dimensions

szGlobalWorkSize = height*width;

Make sure that **szGlobalWorkSize** > **szLocalWorkSize**

Code Walkthrough (1)

```
#include <CL/cl.h>

// Query platform ID
  cl_platform_id platform;
  clGetPlatformIDs (1, &platform, NULL);

// Setup context properties
  cl_context_properties props[3];
  props[0] = (cl_context_properties)CL_CONTEXT_PLATFORM;
  props[1] = (cl_context_properties)platform;
  props[2] = (cl_context_properties)0;

// Create a context to run OpenCL on our GPU
  cl_context GPUContext = clCreateContextFromType(props, CL_DEVICE_TYPE_GPU,NULL, NULL, NULL);

// Get the list of GPU devices associated with this context
  size_t ParmDataBytes;
  clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, 0, NULL, &ParmDataBytes);
  cl_device_id* GPUDevices = (cl_device_id*)malloc(ParmDataBytes);
  clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, ParmDataBytes, GPUDevices, NULL);

// Create a command-queue on the first GPU device
  cl_command_queue GPUCommandQueue = clCreateCommandQueue(GPUContext, GPUDevices[0], 0, NULL);
```

Code Walkthrough (2)

```
// Allocate GPU memory for source vectors AND initialize from CPU memory
cl_mem GPUVector1 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(int) * SIZE, HostVector1, NULL);
                cudaMalloc( (void**)&d_Vector1, sizeof(int) * SIZE );
                cudaMemcpy(d_Vector1, h_Vector1, sizeof(int) * SIZE, cudaMemcpyHostToDevice);
cl_mem GPUVector2 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, sizeof(int) * SIZE, HostVector2, NULL);
// Allocate output memory on GPU
cl_mem GPUOutVector = clCreateBuffer(GPUContext, CL_MEM_WRITE_ONLY,
   sizeof(int) * SIZE, NULL, NULL);
                cudaMalloc( (void**)&d_OutVector, sizeof(int) * SIZE );
// Create OpenCL program with source code
 cl_program OpenCLProgram = clCreateProgramWithSource(GPUContext, 7,
   OpenCLSource, NULL, NULL);
// Build the program (OpenCL JIT compilation)
 clBuildProgram(OpenCLProgram, 0, NULL, NULL, NULL, NULL)
```

Code Walkthrough (3)

```
// Create a handle to the compiled OpenCL function (Kernel)
cl_kernel OpenCLVectorAdd = clCreateKernel(OpenCLProgram, "VectorAdd", NULL);
// In the next step we associate the GPU memory with the Kernel arguments
clSetKernelArg(OpenCLVectorAdd, 0, sizeof(cl_mem), (void*)&GPUOutVector);
clSetKernelArg(OpenCLVectorAdd, 1, sizeof(cl_mem), (void*)&GPUVector1);
clSetKernelArg(OpenCLVectorAdd, 2, sizeof(cl_mem), (void*)&GPUVector2);
// Launch the Kernel on the GPU
size_t WorkSize[1] = {SIZE};
clEnqueueNDRangeKernel(GPUCommandQueue, OpenCLVectorAdd, 1, NULL, WorkSize,
   NULL, 0, NULL, NULL);
         dim3 dimBlock(64,1,1);
         dim3 dimGrid(ceil(SIZE/(float)64),1,1);
         CUDAVectorAdd<<<dimGrid,dimBlock>>>(d_Vector1, d_Vector2, d_OutVector);
// Copy the output in GPU memory back to CPU memory
int HostOutputVector[SIZE];
clEnqueueReadBuffer(GPUCommandQueue, GPUOutVector, CL_TRUE, 0, SIZE * sizeof(int),
   HostOutputVector, 0, NULL, NULL);
```

Code Walkthrough (4)

```
// Cleanup
    clReleaseKernel(OpenCLVectorAdd);
    clReleaseProgram(OpenCLProgram);
    clReleaseCommandQueue(GPUCommandQueue);
    clReleaseMemObject(GPUVector1);
                    cudaFree( d_Vector1);
    clReleaseMemObject(GPUVector2);
                    cudaFree( d_Vector2);
    clReleaseMemObject(GPUOutputVector);
                    cudaFree( d_OutVector);
    free(GPUDevices);
    clReleaseContext(GPUContext);
```

Vector addition kernel

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

OpenCL

```
__global__ void VectorAdd( int *c,
                           const int *a,
                           const int *b)
{
   int idx = blockIdx.x*blockDim.x + threadIdx.x;
   c[idx] = a[idx] + b[idx];
}
```

CUDA

Profiling

```
float executionTimeInseconds = 0;
for(no_of_iterations=0;no_of_iterations<10;no_of_iterations++){
   writeBufferH2D(&GPU_In, input[0].i_In);
   clEnqueueNDRangeKernel(commandQueue, newKernel, 1, NULL,    (size_t*)&globalWorkSize,    (size_t*)&localWorkSize,
0, NULL, &GPUExecution[0]);
   copyBuffer();
   for(int i=0; i<inputCount; i++){
            writeBufferH2D(&GPU_In, i_In);
            clEnqueueNDRangeKernel(commandQueue, newKernel, 1, NULL,
            (size_t*)&globalWorkSize, (size_t*)&localWorkSize, 0, NULL, NULL);
            copyBuffer();
   }
    clEnqueueNDRangeKernel(commandQueue, newKernel, 1, NULL,               (size_t*)&globalWorkSize,
(size_t*)&localWorkSize, 0, NULL,  &GPUExecution[1]);
   copyBuffer();
   clFinish(commandQueue);
   clGetEventProfilingInfo(GPUExecution[1], CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
   clGetEventProfilingInfo(GPUExecution[0], CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
   executionTimeInseconds += (end - start)*(1.0e-9f) ;
}
printf ("\nAverage GPU time for %.9f sec \n", (executionTimeInseconds)/10);
```

Scalar Architecture and Compiler

- NVIDIA GPUs have a scalar architecture
  - Use vector types in OpenCL for convenience, not performance
  - Generally want more work-items rather than large vectors per work-item

- Use the **-cl-mad-enable** compiler option
  - Permits use of FMADs (floating multiply-add), which can lead to large performance gains
  - The mad computes a * b + c with reduced accuracy. For example, some OpenCL devices implement mad as truncate the result of a * b before adding it to c.

- Investigate using the **-cl-fast-relaxed-math** compiler option
  - enables many aggressive compiler optimizations

Native Functions

- There are two types of runtime math libraries
  - Native_*() map directly to the hardware level: faster but lower accuracy
  - *(): slower but higher accuracy
- There are many native functions:
  - **native_sin**, **native_cos**, **native_powr**
- Use native math library whenever speed is more important than precision. For example, you can use for gaming applications.

Local Memory

Shared Memory

- Take Advantage of __local Memory
- Work-items can cooperate via __local memory
  - Synchronize to make sure each work-item is done updating : barrier(CLK_LOCAL_MEM_FENCE)

__syncthreads()

- Use it to manage locality
  - Stage loads and stores in shared memory to optimize reuse

Enabling extensions

- #pragma OPENCL EXTENSION *extension name* : *behavior*

  Behavior can be enable or disable. Extension name is the optional extension that should be enabled.

- #pragma OPENCL EXTENSION **cl_khr_fp64** : enable
  - enables double precision arithmetics and ALL math, geometric and common functions.

- #pragma OPENCL EXTENSION **cl_khr_byte_addressable_store** : enable
  - enables the application to write 1 byte data types into global memory. This is very important to effectively manipulate strings or monochromatic images.

Reminder

- Never ignore compiler warnings, spend some time to understand if it is safe.

OpenCL Profiler

- Profiler facilitates analysis and optimization of OpenCL programs by:
  - Reporting hardware counter values:
    - Number of various bus transactions
    - Branches
    - Effective Parallelism
- Computing per kernel statistics:
  - Effective instruction throughput
  - Effective memory throughput
- Visually displaying time spent in various GPU calls
- Requires no instrumentation of the source code