EEP 524
Applied High-Performance GPU Computing

LECTURE 2 – Part 1 : Wednesday, January 13, 2021

Instructor: Dr. Colin Reinhardt

EE Professional Masters Program

Winter 2021

Lecture 2 – Part 1 : Outline

- Lecture 1 and EX1 Review/Summary
- Just Enough GPU History
- GPU Compute API Conceptual Models
  - with GPU Hardware Architecture 101
  - *OpenCL*
  - *CUDA*
- The Full vector addition Application (CUDA-DRV+OCL)
  - Host application
  - Device kernel
- Ex2 & Hw1 – Draft posted, due date TBD

Lecture 1 and EX1 Review/Summary

- **Lecture 1 Part 1**: (live lecture, Wed 1/6, posted on Zoom cloud)
  - Course intro, logistics, syllabus, schedule, expectations, TA
  - Background: why multi-/many-core HW revolution and parallel programming?
  - basic concepts of parallel execution
  - computational "index space" (CUDA=grid, OpenCL=NDRange)
  - vector addition GPU kernel function
- **Lecture 1 Part 2**: (Panopto recording, posted Sun 1/10)
  - Host and Device
  - GPU Application development overview
  - CUDA Runtime API
- **READINGS**: Week1
- **EX1**
  - VS2019 IDE and project setup
  - Query Platform/ Device using OpenCL and CUDA Driver API
  - HelloGPUWorld with CUDA Runtime API
  - Simplest vector addition with CUDA Runtime API

Remember our course approach is a series of passes,
each with progressively more depth & detail.

Just Enough GPU History

Early GPUs were quietly leading a parallel life…

What GPUs were originally designed to do: 3D rendering

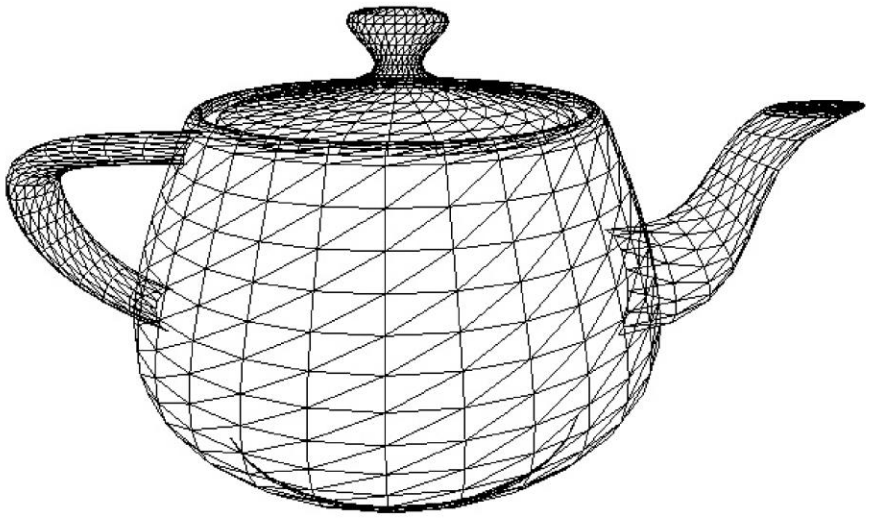


Image credit: Henrik Wann Jensen

Input: description of a scene:

3D surface geometry (e.g., triangle mesh)
surface materials, lights, camera, etc.

Output: image of the scene

Simple definition of rendering task: computing how each triangle in 3D mesh contributes to appearance of each pixel in the image?

What GPUs are still designed to do

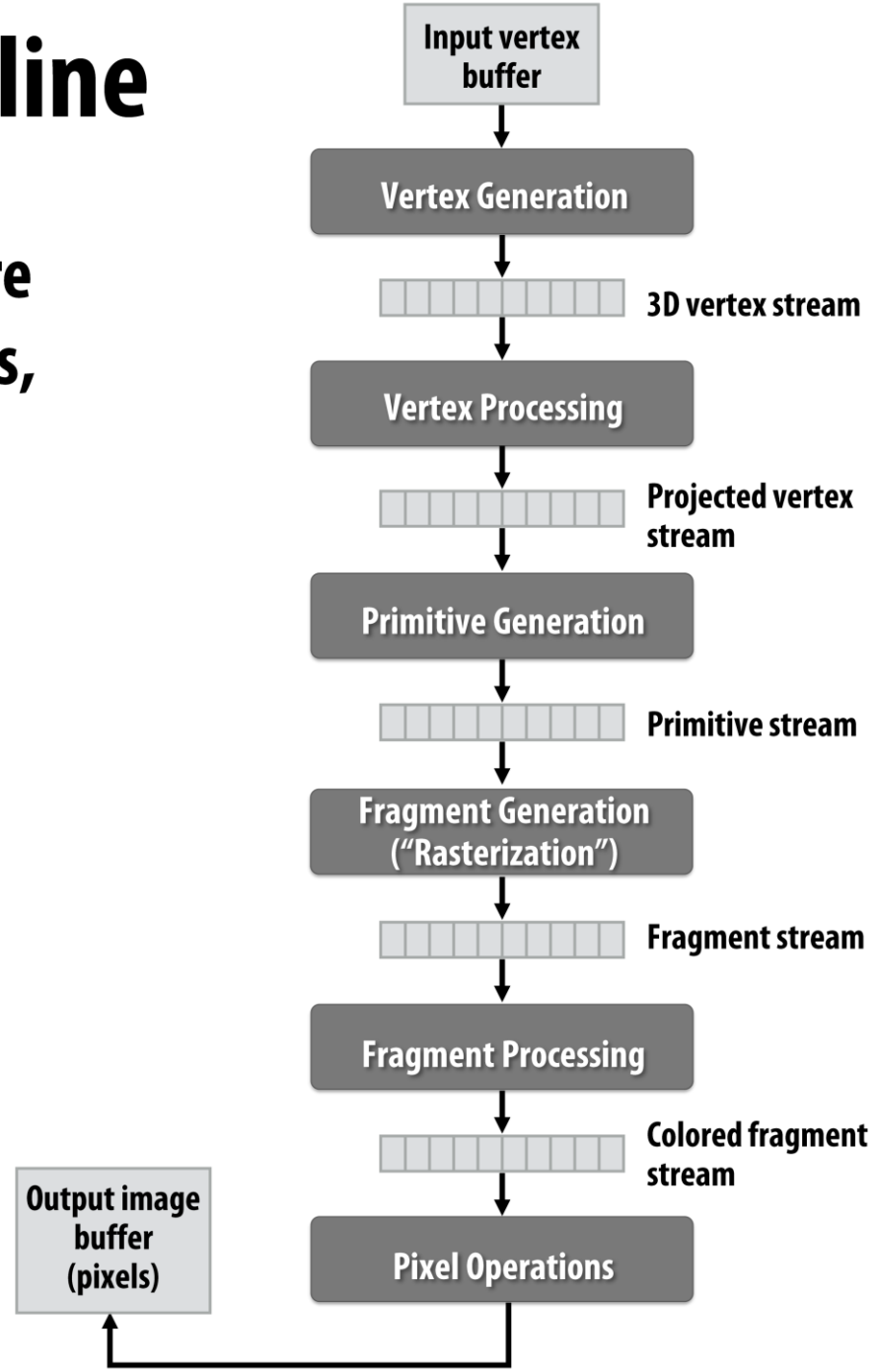
Unreal Engine Kite Demo (Epic Games 2015)

NVIDIA RTX Real-time Ray Tracing

WATCH to (dis)BELIEVE: https://youtu.be/NgcYLIvlp_k

Real-time graphics pipeline

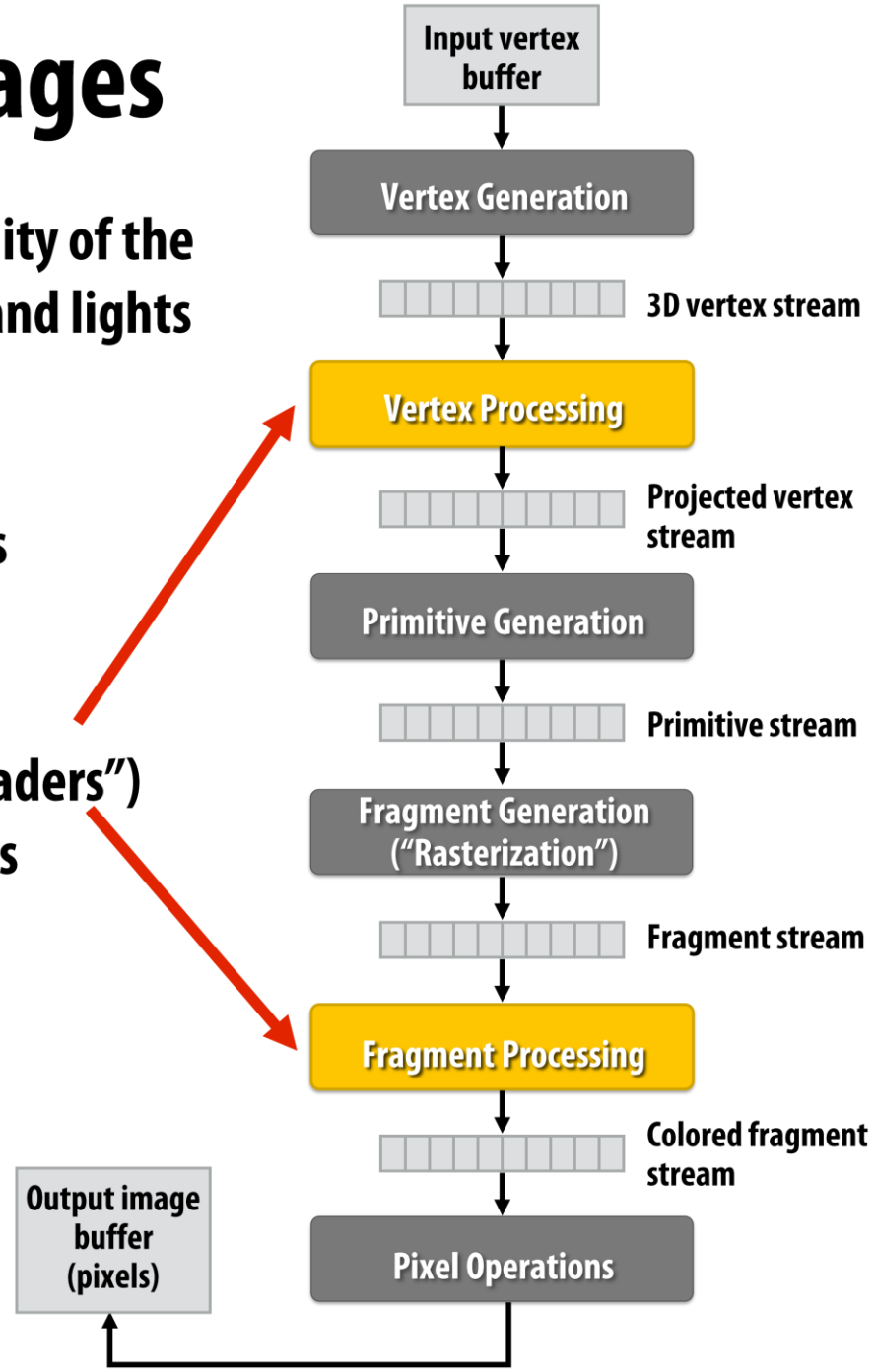
Abstracts process of rendering a picture as a sequence of operations on vertices, primitives, fragments, and pixels.



Graphics shading languages

- Allow application to extend the functionality of the graphics pipeline by specifying materials and lights programmatically!
  - Support diversity in materials
  - Support diversity in lighting conditions

- Programmer provides mini-programs ("shaders") that define pipeline logic for certain stages
  - Pipeline maps shader function onto all elements of input stream



Hack! early GPU-based scientific computation

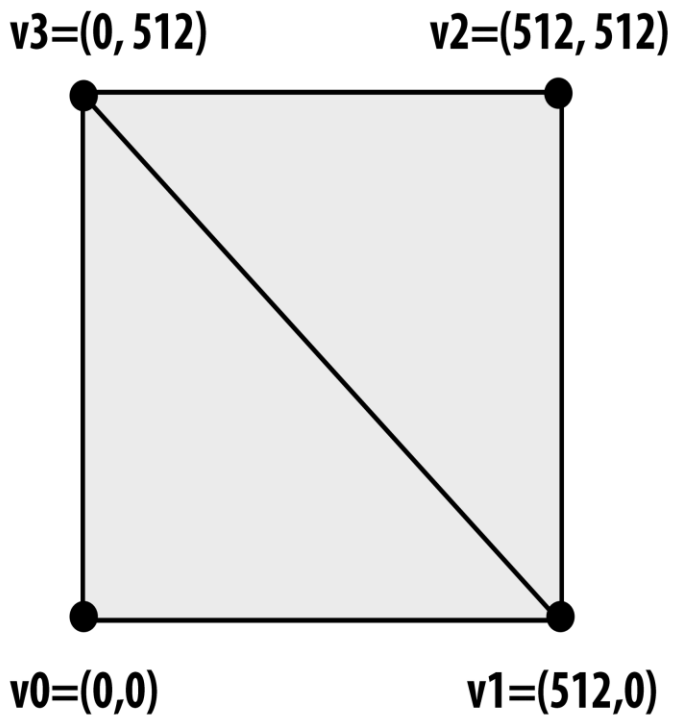
Set OpenGL output image size to be output array size (e.g., 512 x 512)

Render 2 triangles that exactly cover screen
(one shader computation per pixel = one shader computation output image element)

We now can use the GPU like a data-parallel programming system.

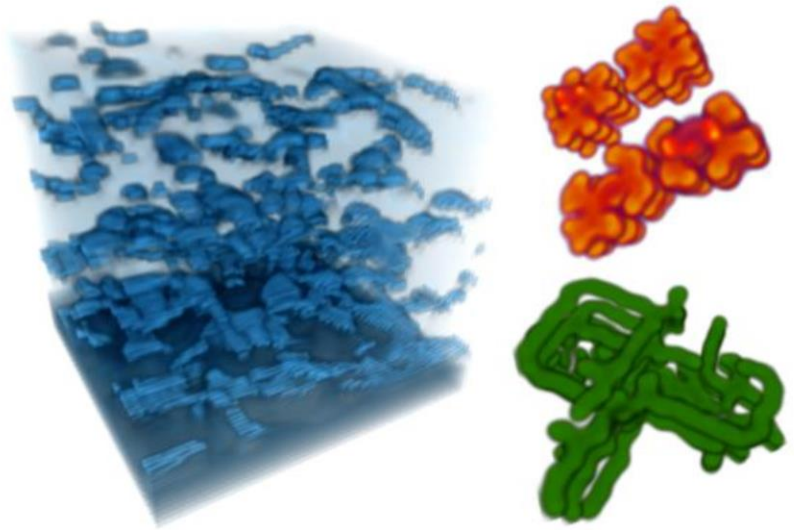
Fragment shader function is mapped over 512 x 512 element collection.

Hack!

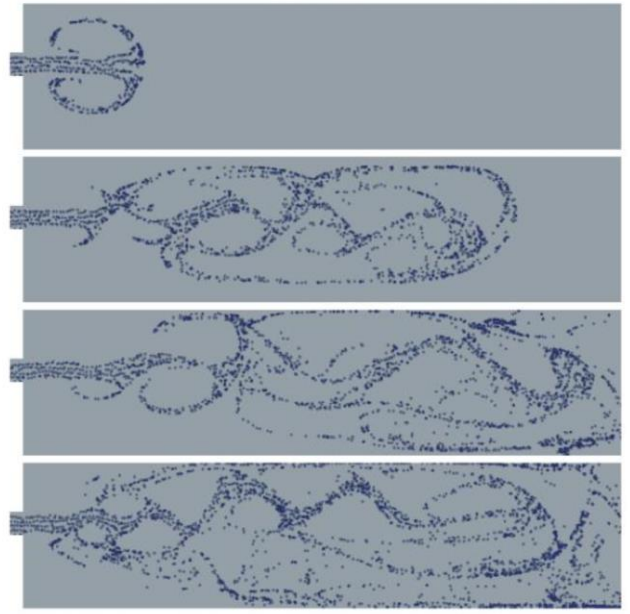


“GPGPU” 2002-2003

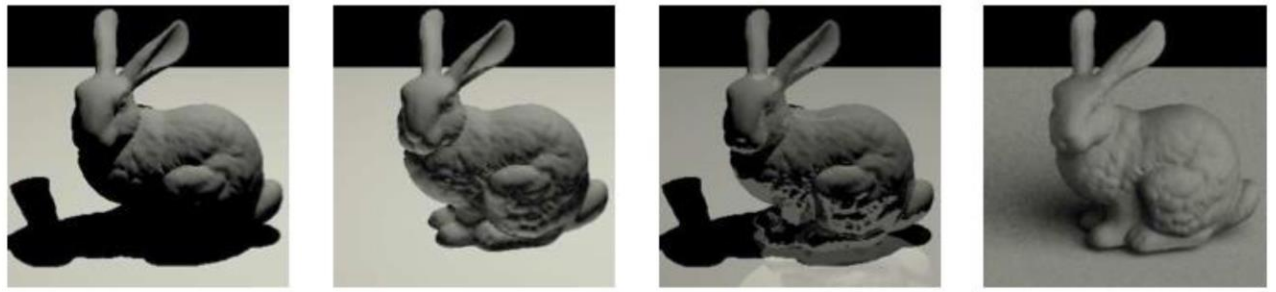
GPGPU = “general purpose” computation on GPUs



Coupled Map Lattice Simulation [Harris 02]



Sparse Matrix Solvers [Bolz 03]



Ray Tracing on Programmable Graphics Hardware [Purcell 02]

NVIDIA Tesla architecture (2007)

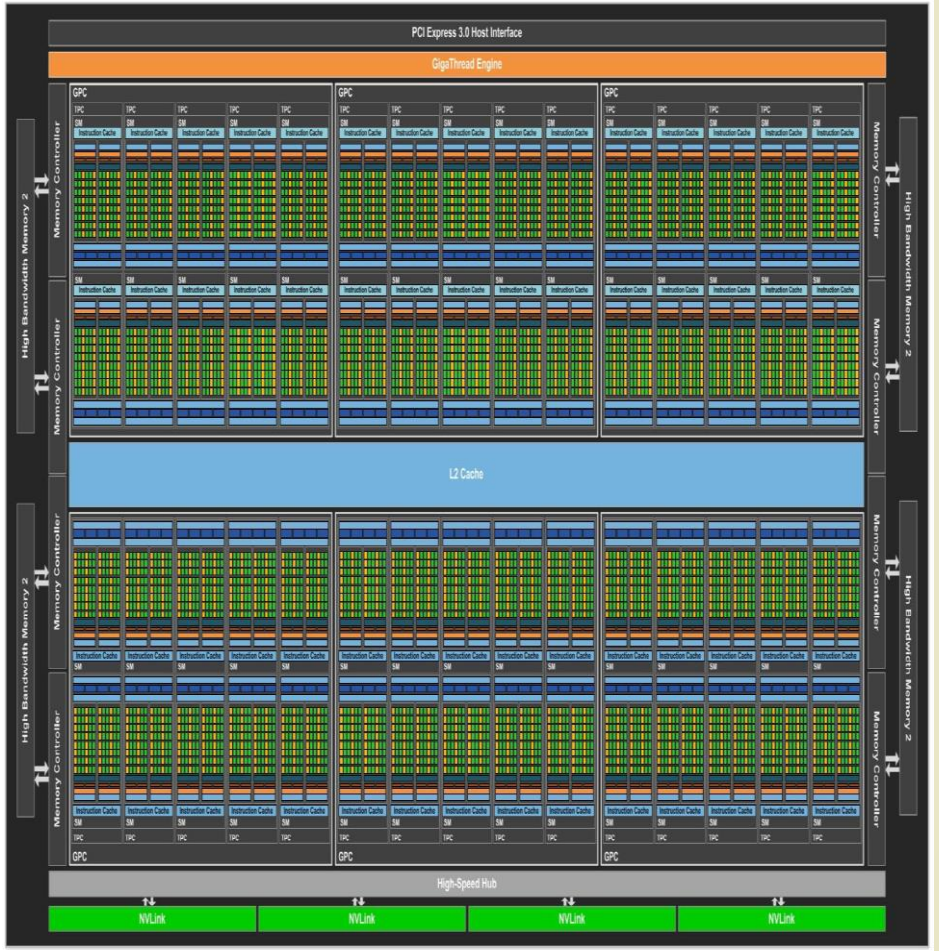
(GeForce 8xxx series GPUs)

First alternative, non-graphics-specific ("compute mode") interface to GPU hardware

Lets say a user wants to run a non-graphics program on the GPU's programmable cores…

- Application can allocate buffers in GPU memory and copy data to/from buffers
- Application (via graphics driver) provides GPU a single kernel program binary
- Application tells GPU to run the kernel in an SPMD fashion ("run N instances")
- Go! (launch(myKernel, N))

Pascal GP100 with 60 SM units



CUDA programming language

- Introduced in 2007 with NVIDIA Tesla architecture
- "C-like" language to express programs that run on GPUs using the compute-mode hardware interface
- Relatively low-level: CUDA's abstractions closely match the capabilities/performance characteristics of modern GPUs (design goal: maintain low abstraction distance)
- Note: OpenCL is an open standards version of CUDA
  - CUDA only runs on NVIDIA GPUs
  - OpenCL runs on CPUs and GPUs from many vendors

The origins of OpenCL

AMD

ATI

Merged, needed commonality across products

NVIDIA

GPU vendor – wants market share from CPU

Intel

CPU vendor – wants market share from GPU

Apple

Was tired of recoding for many core, GPUs. Pushed vendors to standardize.

Wrote a rough draft straw man API

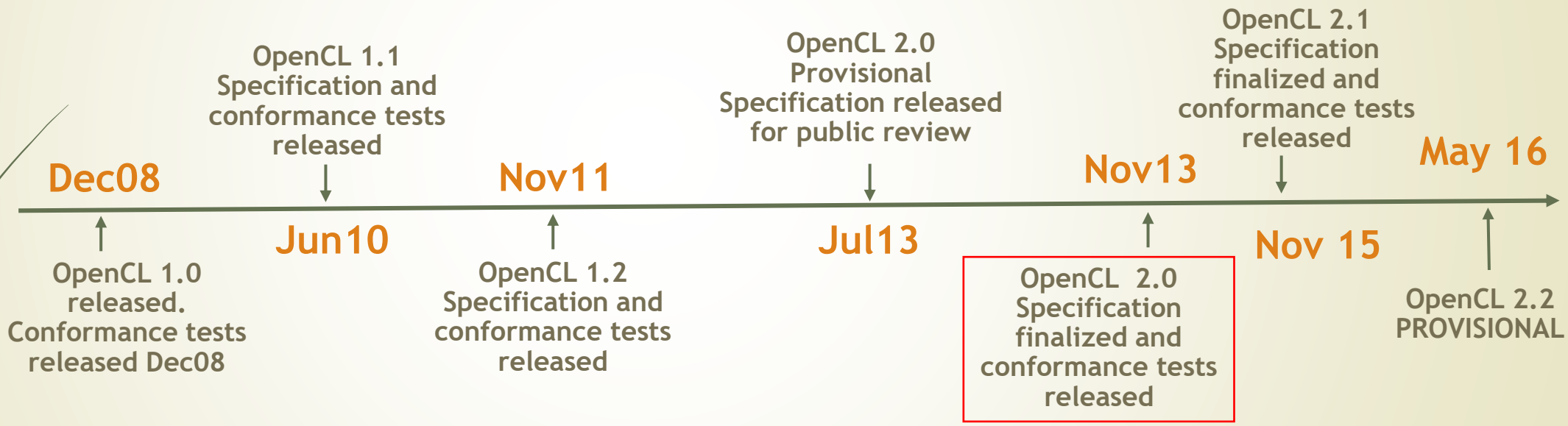
Khronos Compute group formed

- **ARM**
- **Nokia**
- **IBM**
- **Sony**
- **Qualcomm**
- **Imagination**
- **TI**
- **+ many more**

OpenCL

Incidentally, OpenCL is a trademark of Apple Inc.

OpenCL Timeline



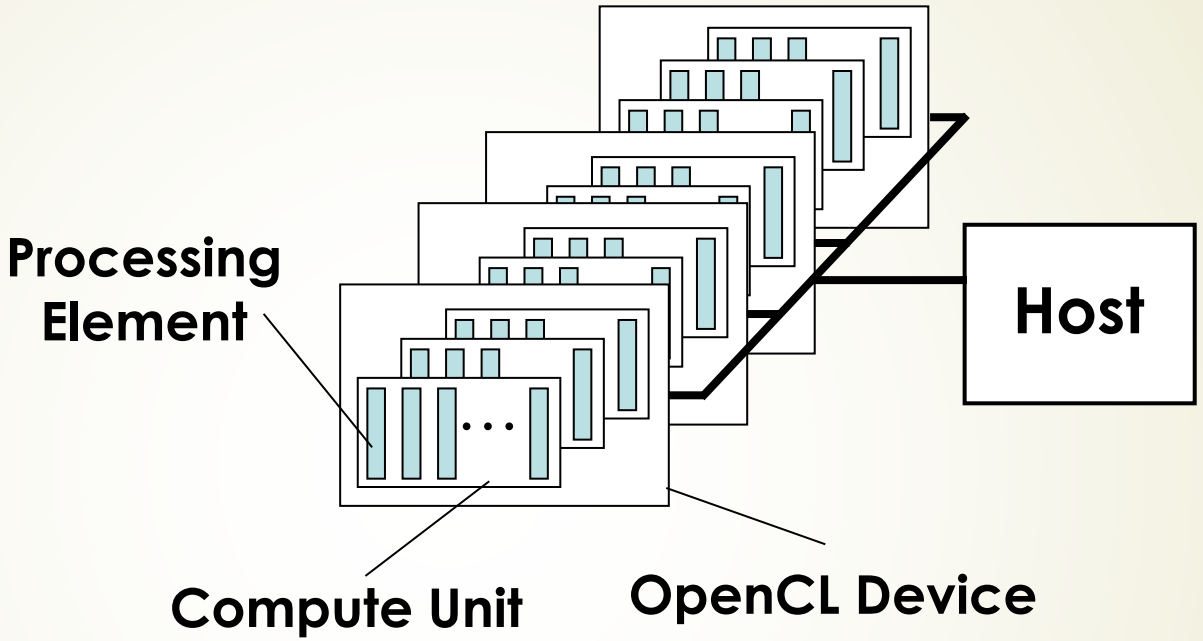
BREAK

GPU Framework Models, Host APIs, and device kernel languages

VIEWS: OpenCL and CUDA

- We will look at several conceptual views/models of the GPU framework
  - Platform
  - Execution
  - Memory
  - Programming model – which will be interleaved throughout the other views
- For each view/model, we'll start with OpenCL API
  - since it is more general, in order to be vendor & hardware 'agnostic'
    - and thus it is also more complex
- Then we'll compare the NVIDIA CUDA Driver API equivalents and terminology

OpenCL Platform Model



- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units (CUs)***
    - Each Compute Unit is divided into one or more ***Processing Elements (PEs)***
- Memory divided into ***host memory*** and ***device memory***

OpenCL Platform Example (One node, two CPU sockets, two GPUs)

CPUs:

- Treated as one OpenCL device
  - One CU per core
  - 1 PE per CU, or if PEs mapped to SIMD lanes, *n* PEs per CU, where *n* matches the SIMD width
- Remember:
  - the CPU will also have to be its own host!

GPUs:

- Each GPU is a separate OpenCL device
- Can use CPU and all GPU devices concurrently through OpenCL

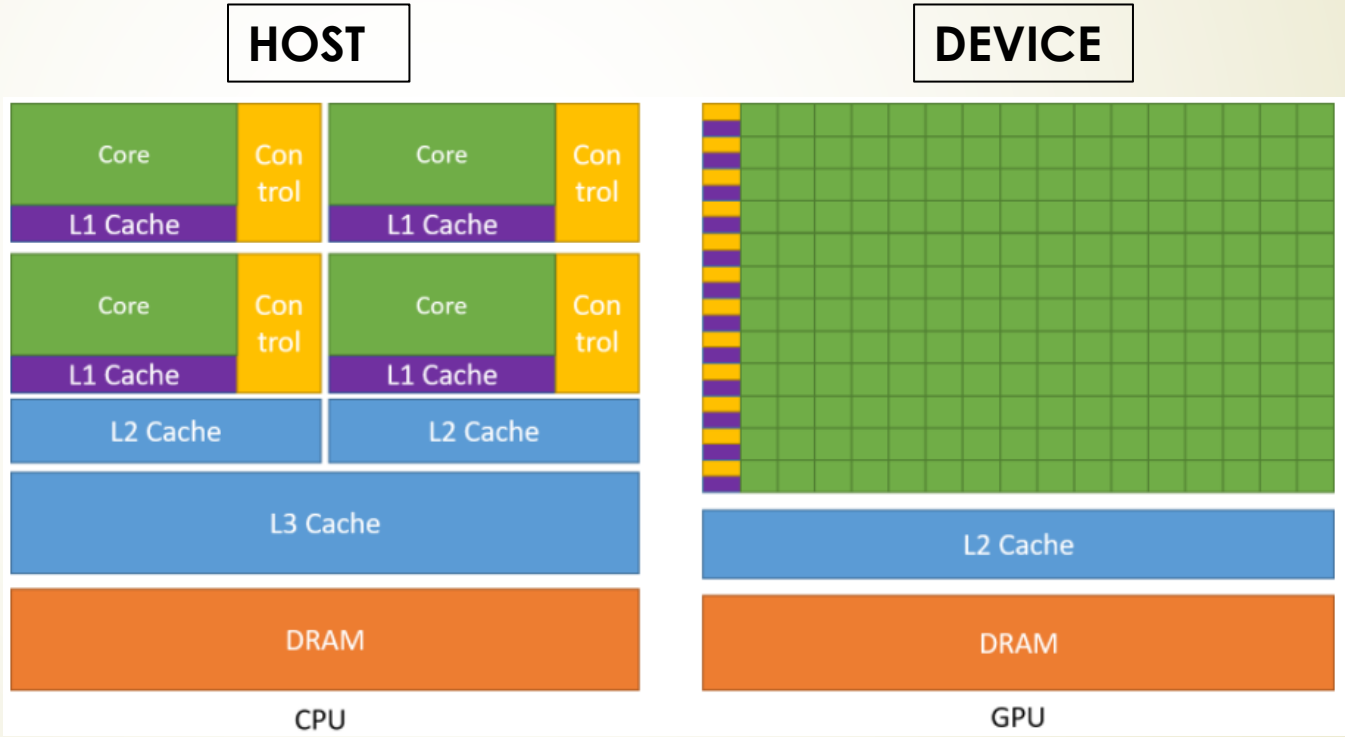
CU = Compute Unit; PE = Processing Element

And in CUDA…

- **Host** and **Device** terms are also used.
  - and each has its own separate memory system
  - at present all NVIDIA devices are "discrete" GPUs and are connected to the Host by the PCI Express (PCIe) interface
- the *Platform* level is absent, since only NVIDIA platform is supported.
- Compute Unit (CU) = **Streaming Multiprocessor** (SM)
- Processing Element (PE) = **CUDA cores**
  - current NVIDIA GPUs have several types of cores in each SM
    - FP32, INT32, FP64, Tensor cores (and multiple instances of each)

HOST (CPU) and DEVICE (GPU)

- Simplified conceptual architecture view
  - notice relative use of chip area



- CPU: minimum latency optimized
- GPU: maximum throughput optimized

OpenCL Host API : Platform Model

```
// Get OpenCL platform and device information
cl_platform_id * platforms = NULL;
cl_uint num_platforms = 0;

// Set up the Platform
cl_int clStatus = clGetPlatformIDs(0, NULL, &num_platforms);
platforms = (cl_platform_id *)malloc(sizeof(cl_platform_id)*num_platforms);
clStatus = clGetPlatformIDs(num_platforms, platforms, NULL);

// Get the devices list and choose the device you want to run on
cl_device_id *device_list = NULL;
cl_uint num_devices = 0;
clStatus = clGetDeviceIDs( platforms[0], CL_DEVICE_TYPE_GPU, 0,NULL, &num_devices);
device_list = (cl_device_id *)malloc(sizeof(cl_device_id)*num_devices);
clStatus = clGetDeviceIDs( platforms[0],CL_DEVICE_TYPE_GPU, num_devices, device_list, NULL);
```

Common Host-API pattern: query for size, allocate mem, call again to retrieve data

And in CUDA… (DRIVER API)

```
cuChk(cuInit(0));


// Get number of CUDA devices
int deviceCount = 0;
cuChk(cuDeviceGetCount(&deviceCount));


// Get handle for device 0
CUdevice cuDev;
cuChk(cuDeviceGet(&cuDev, 0));
```

OpenCL Execution Model

- Defined in terms of two distinct units of execution
  - **Host program**: executes on host
  - **Kernels**: execute on one or more OpenCL devices
    - Where the work associated with computation occurs
    - **Work-items** executing in **work-group**s
- **Context**: defines environment within which kernels execute
  - Devices
  - Kernel objects
  - Program objects
  - Memory objects
- **Command Queue**: enable host to interact with a device
  - Each associated with single device
  - **Command** types placed into command-queue
    - Kernel-enqueue, Memory, Synchronization

And in CUDA…

- The Host program and the kernels are conceptually the same

- work-group = **thread block**
- work-item = **thread**

- Both OpenCL and CUDA have a context
- command queue = CUDA **stream**
  - the CUDA context automatically creates a default stream 0 (NULL stream)

Kernel execution

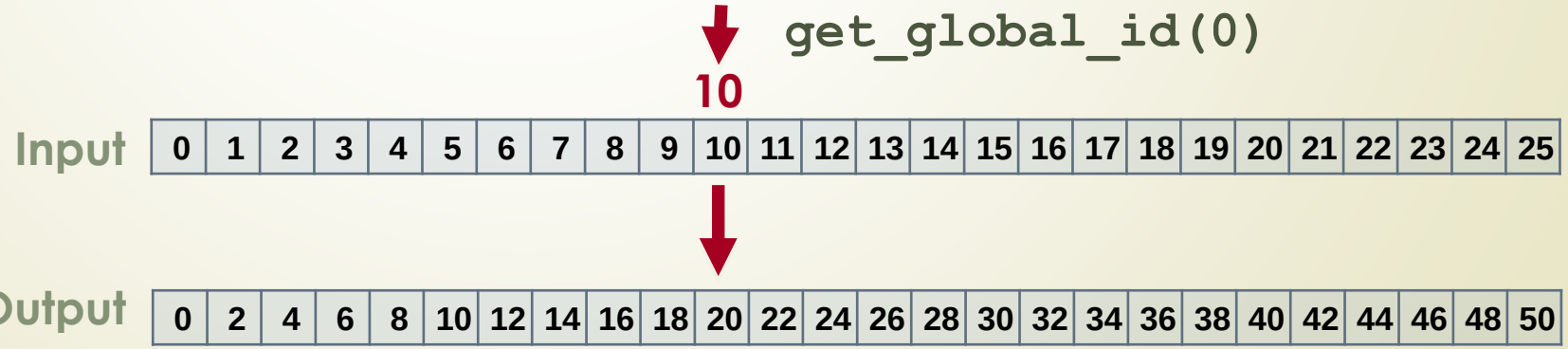
"The core of the OpenCL Execution Model"

- ***kernel-instance***
  - A kernel (function) with argument values and index space parameters
- *Work-item (WI)*: a specific kernel instance for a specific point in index space
  - WIs are executed by PEs
- Work Group (WG)
  - Group of WIs associated with a given kernel-instance managed by the device
  - A coarse-grained decomp of Index space
  - WGs share block of high-speed local memory
  - WIs within a WG can be synchronized with *fences* and *barriers*
    - We will cover synchronization details later in the class
  - Platform Model Compute Units (CUs) are hardware processing resources which support WGs
    - each WG executes on single CU
    - each CU executes only one WG at a time*

Execution model (kernels)

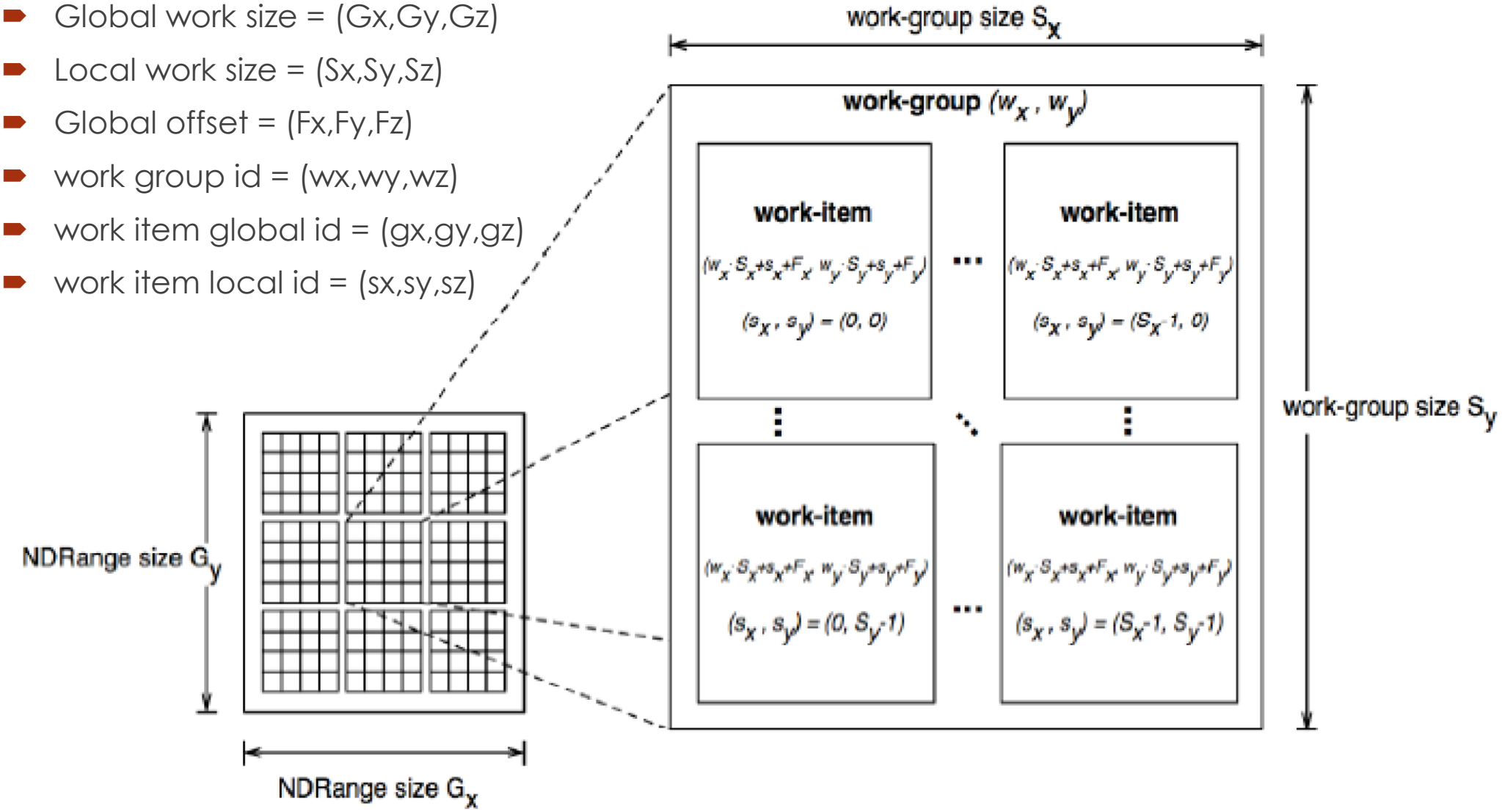
- OpenCL execution model … define a problem computation domain and execute an instance of a **kernel** for each point in the domain

```
__kernel void times_two(__global float* input,
                        __global float* output)
{
    int i = get_global_id(0);
    output[i] = 2.0f * input[i];
}
```



Data Partitioning: NDRange Index Space

- Global work size = (Gx,Gy,Gz)
- Local work size = (Sx,Sy,Sz)
- Global offset = (Fx,Fy,Fz)
- work group id = (wx,wy,wz)
- work item global id = (gx,gy,gz)
- work item local id = (sx,sy,sz)



Data Partitioning: NDRange Index Space

Worked Example: consider the 2D index space

Each work item is uniquely identified by

1. Global ID: (gx, gy)
2. Combination of {work-group ID (wx,wy), size of work-group (Sx,Sy), local ID (sx,sy)}

```
(gx, gy) = (wx*Sx + sx + Fx,  wy*Sy + sy + Fy)
```

The number of work-groups can be computed as:

```
(Wx, Wy) = [ ceil(Gx/Sx),  ceil(Gy/Sy) ]
```

Given the global ID and work-group size, the work-group ID for a work-item is:

```
(wx, wy) = [ (gx -sx - Fx)/Sx,  (gy -sy -Fy)/Sy ]
```

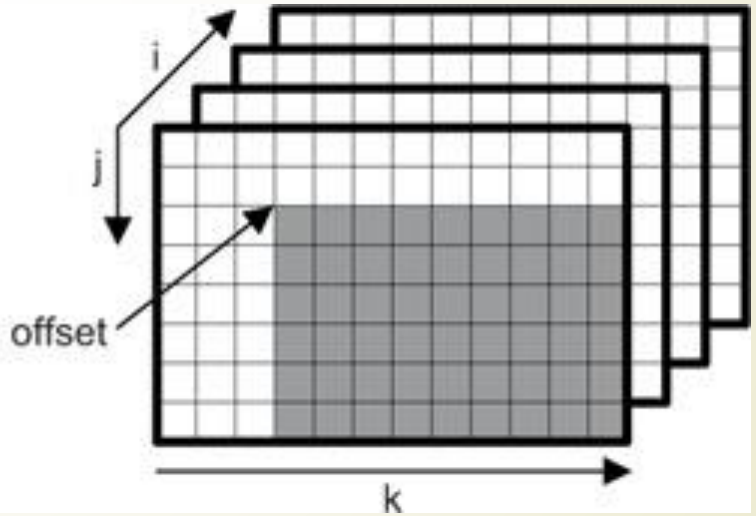
Data Partitioning

```
for(i=0; i<Z; i++) {
   for(j=0; j<Y; j++) {
      for(k=0; k<X; k++) {
         process(point[i][j][k]);
      }
   }
}
```

```
int i = get_global_id(0);
int j = get_global_id(1);
int k = get_global_id(2);
process(point[i][j][k]);
```

Example

```
for(i=0; i<4; i++) {
   for(j=2; j<8; j++) {
      for(k=3; k<12; k++) {
         process(point[i][j][k]);
      }
   }
}
```




```
size_t global_work_size[3] = {4, 6, 9};

size_t global_work_offset[3] = {0, 2, 3};
```

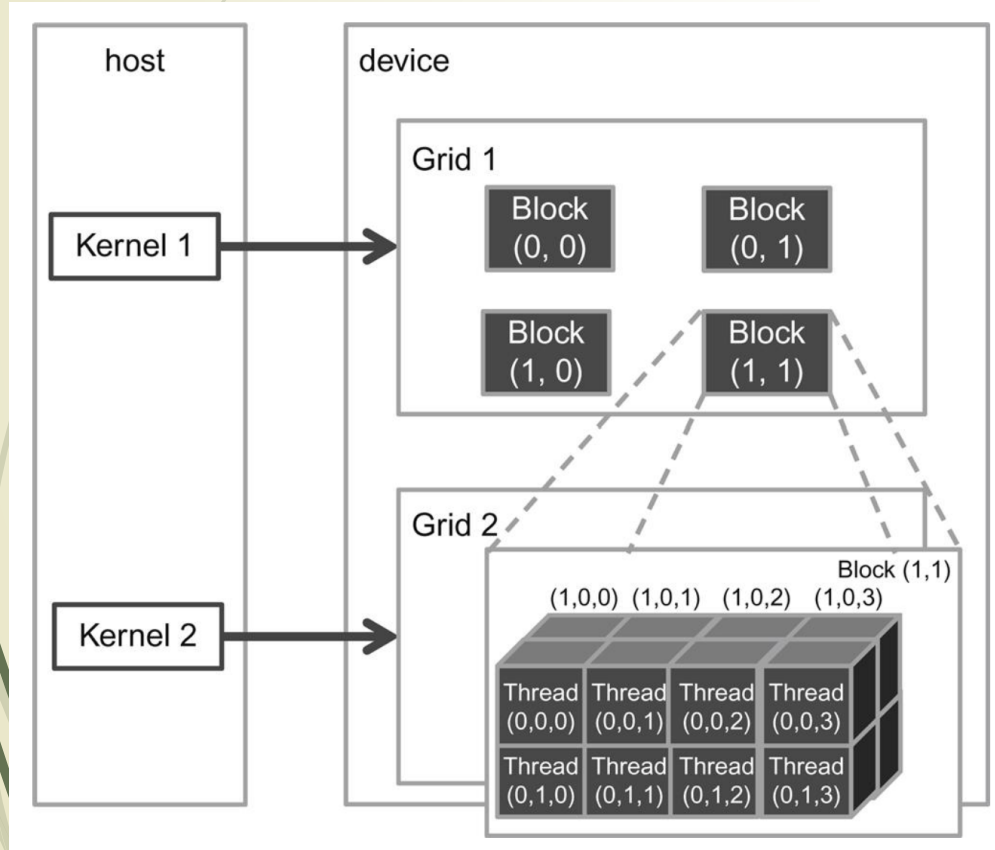
And in CUDA…

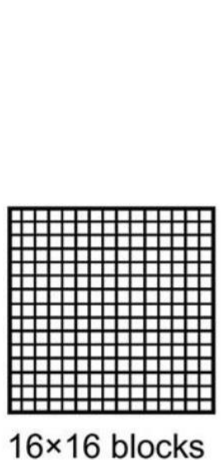
- As we discussed
  - work-group => thread block
  - work-item => thread
- However CUDA has an additional concept, closer to the HW
  - the **warp**
    - each warp has size 32 threads (at least currently on all NVIDIA GPUs)
  - a thread block may be broken (partitioned) into multiple warps
    - for example, if numThreadsPerBlock = 128, then there will be 4 warps.
  - a warp is scheduled and executed on a single CUDA core
    - all threads in a warp start together at the same program counter (PC)
    - each thread has its own PC and register state
    - the core can switch thread contexts very efficiently
  - Multiple warps from different thread blocks can be assigned to each core
    - and should be to maximize utilization (occupancy), as we'll discuss much more.

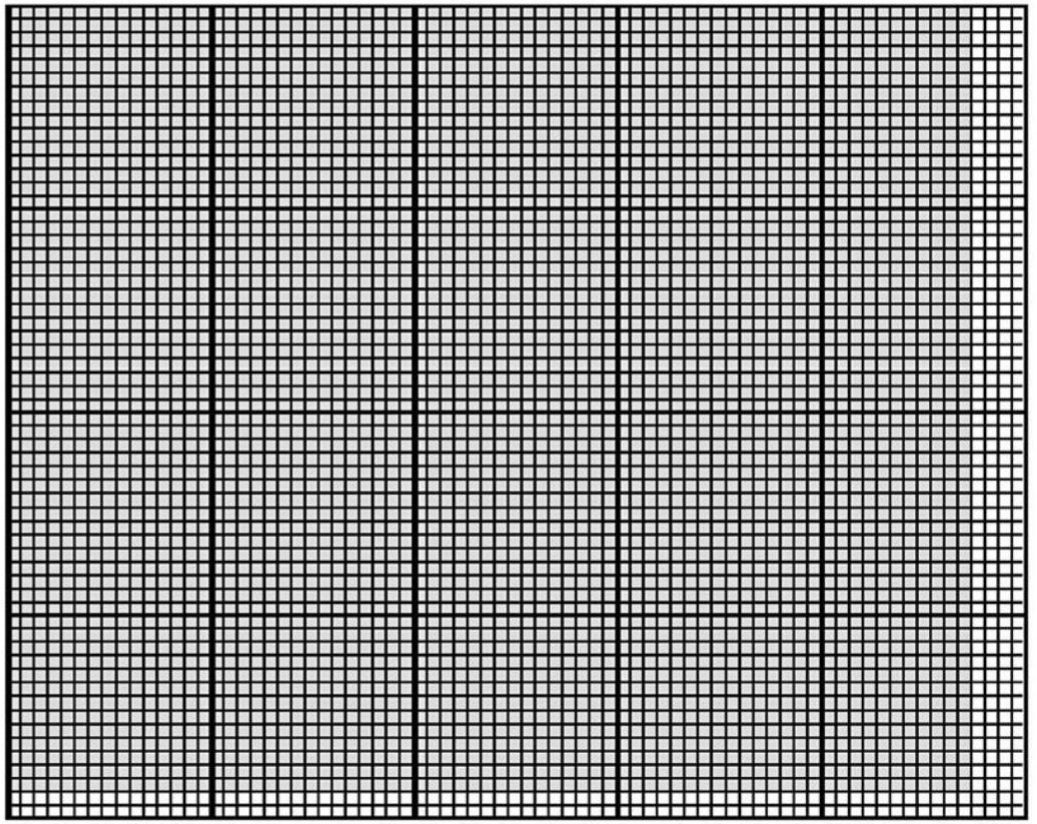
CUDA Thread Hierarchy

- Vital for establishing a kernel thread's position in "index space" at runtime

$$P_{\text{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x}} = P_{1*16+0,0*16+0} = P_{16,0}.$$

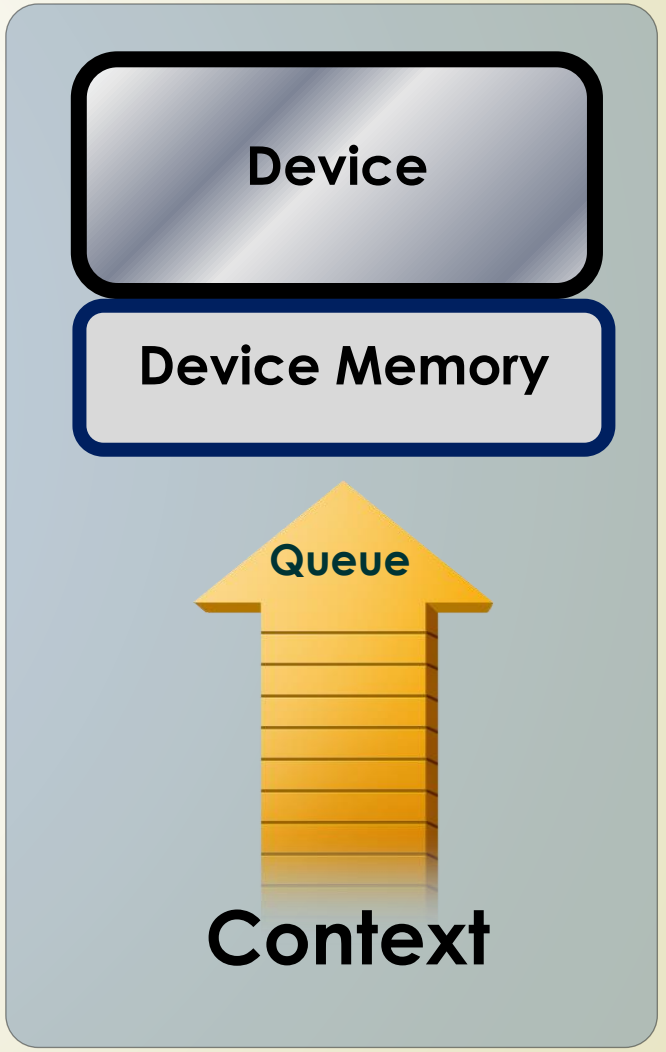






Context and Command-Queues

- ***Context***:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



OpenCL and in CUDA…

- OpenCL

```
cl_context ctx = clCreateContext(NULL, numDevices, devices, NULL, NULL, &status)
```

  - OR (depending on OpenCL version used! 1.2 or 2.0)

```
cl_context_properties properties[3] = { CL_CONTEXT_PLATFORM, (cl_context_properties)platform, '\0' };

context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &err);

const cl_queue_properties queueProps[] = { CL_QUEUE_PROPERTIES,
(cl_command_queue_properties)CL_QUEUE_PROFILING_ENABLE, 0 };

//cl_command_queue commands0 = clCreateCommandQueue(context, device_id, 0, &err); // deprecated OpenCL 1.2
function

cl_command_queue commands0 = clCreateCommandQueueWithProperties(context, device_id, queueProps, &err);
```

- CUDA DRV

```
CUcontext cuCtx;

Custream cuStrm;

CUresult cuRes = cuCtxCreate(&cuCtx, uiFlags, cuDevice);

cuRes = cuStreamCreate(&cuStrm, CU_STREAM_NON_BLOCKING
```

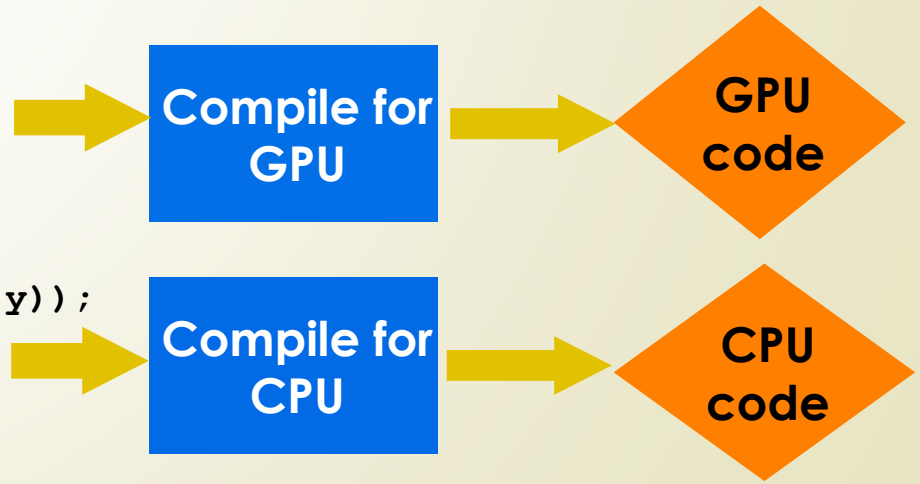
Building Program Objects

- The program object encapsulates:
  - A context
  - The program kernel source or binary
  - List of target devices and build options

OpenCL uses runtime compilation … because in general you don't know the details of the target device when you ship the program

- The C API build process to create a program object:
  - **clCreateProgramWithSource()**
  - **clCreateProgramWithBinary()**

```
__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 w = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,(int2)(w-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



And in CUDA…

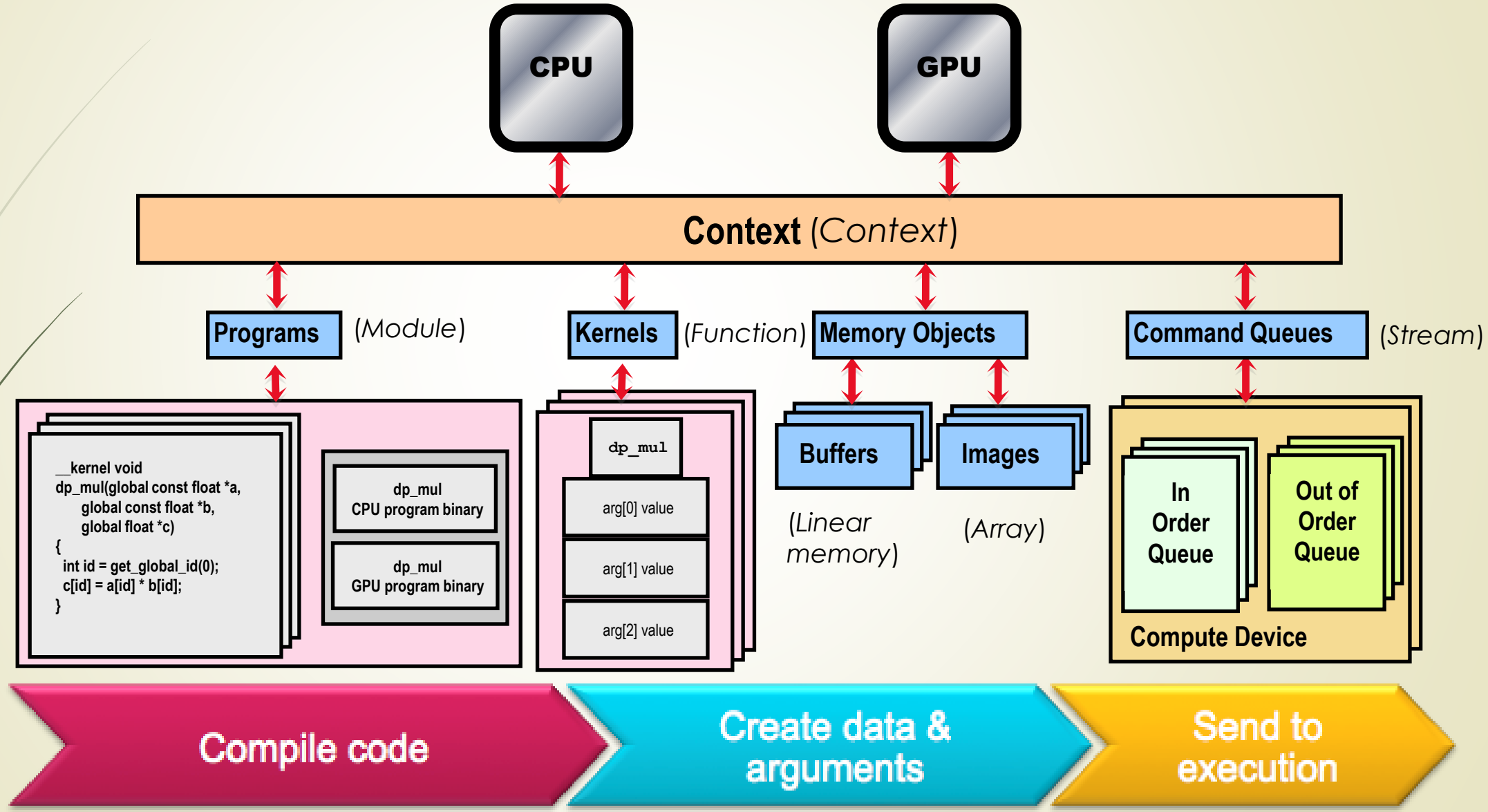
CUDA also can use Just-in-Time (JIT) compilation of device kernels

- Input source kernel code can be
  - PTX intermediate representation (IR) – output by **nvcc**
  - binary CUBIN or FATBIN format – output by **nvcc**
- Instead of a Program CUDA has a **Module** (CUmodule)
- The programmatic object for a kernel is a **Function** (CUfunction)

```
// Create module from binary file
CUmodule cuModule;
cuModuleLoad(&cuModule, "VecAdd.ptx");

// Get function handle from module
CUfunction vecAdd;
cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");
```

Basic OpenCL (*CUDA*) API Framework



GPU kernel launch/dispatch/execution

We haven't gotten to this quite yet!! Almost there.

- OpenCL
  - `clEnqueueNDRangeKernel( … )`
- CUDA Runtime API
  - `mykernel<<<numBlks, numThreads>>>(arg1, arg2, arg3)`
- CUDA Driver API
  - `cuLaunchKernel( … )`

Ex2 & Hw1, Readings Week2

DUE:

See class web: Files/Homework/

- Ex2/Hw1