CSCI-GA.3033-004

Graphics Processing Units (GPUs): Architecture and Programming

OpenCL

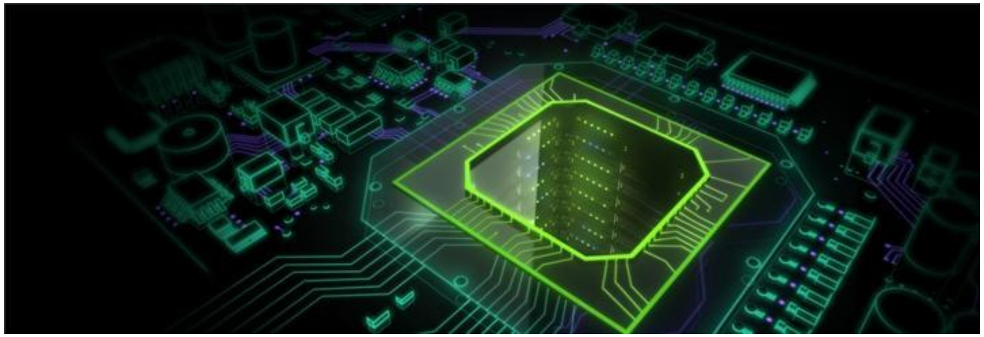
Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

http://www.mzahran.com

Many slides from this lecture are adapted from:

- http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf
- http://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/calling-the-kernel/
- Slides with the book "Heterogeneous Computing with OpenCL 2.0" 3rd edition



OpenCL Working Group

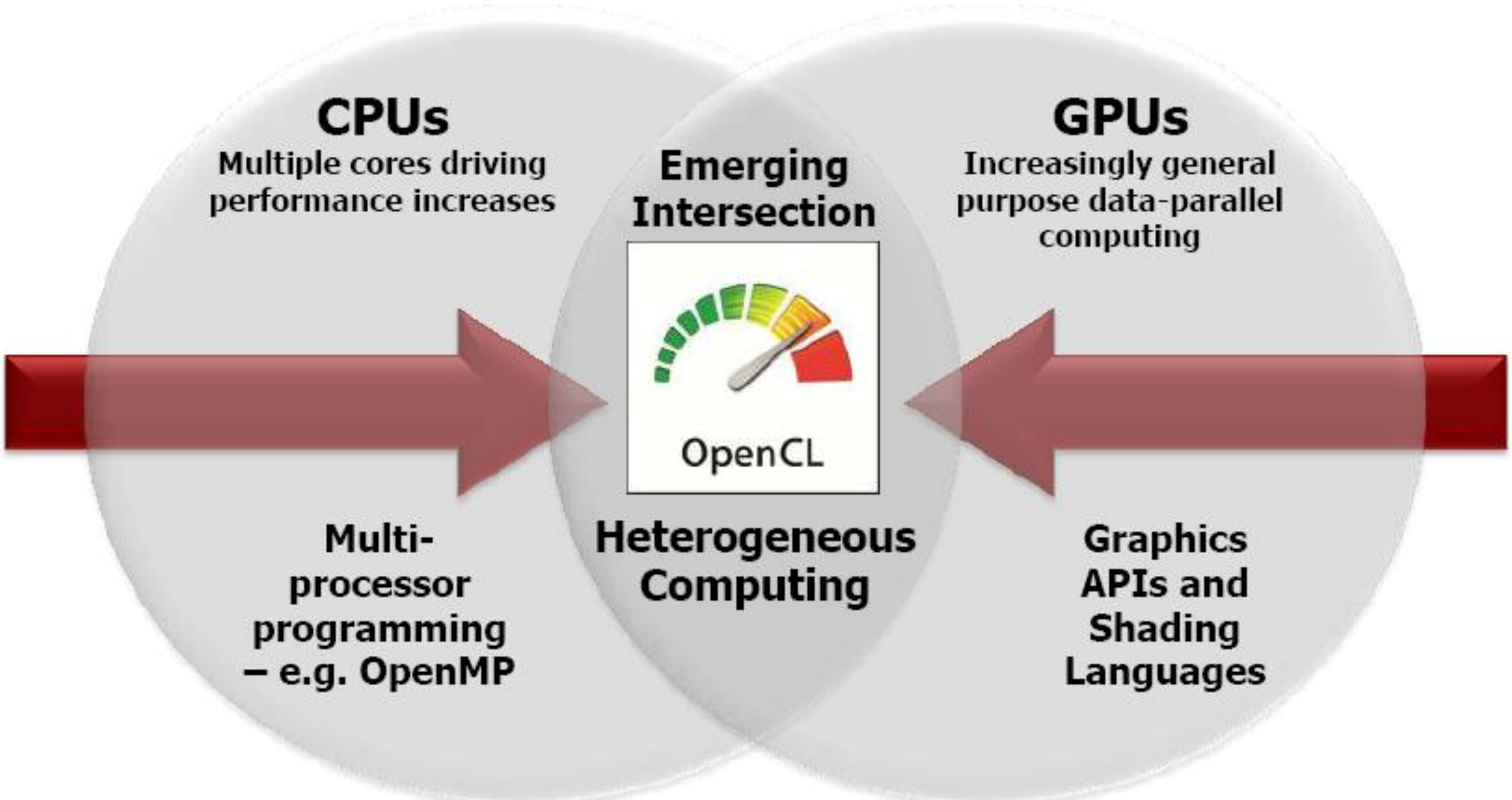
- **Diverse industry participation**
  - Processor vendors, system OEMs, middleware vendors, application developers
- **Many industry-leading experts involved in OpenCL's design**
  - A healthy diversity of industry perspectives
- **Apple initially proposed and is very active in the working group**
  - Serving as specification editor
- **Here are some of the other companies in the OpenCL working group**

OpenCL logo is trademark Apple Inc

Background

- OpenCL was initiated by Apple and maintained by the Khronos Group (also home of OpenGL).
- OpenCL draws heavily on CUDA
  - Easy to learn for CUDA programmers
- OpenCL host code is much more complex and tedious due to desire to maximize portability and to minimize burden on vendors

Processor Parallelism



Design Goals

- Use all computation resources in the system (GPUs, CPUs, Accelerators, FPGAs, …)
- Data parallel model (SPMD) and task parallel model
  - Efficient programming
  - Extension to C
- Abstract underlying parallelism
- Drive future hardware requirements

Implementation

Each OpenCL implementation (OpenCL library from AMD, NVIDIA, etc.) defines *platforms* which enable the host system to interact with OpenCL-capable devices

OpenCL Anatomy

Platform Layer API

- Hardware abstraction layer
- Query, select, and initialize compute device
- create compute contexts and task queues

Runtime API

- Execute compute kernels
- Manage scheduling, compute, and memory resources

Language Specs

- C-based
- Rich set of built-in functions

Simply Speaking

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
```

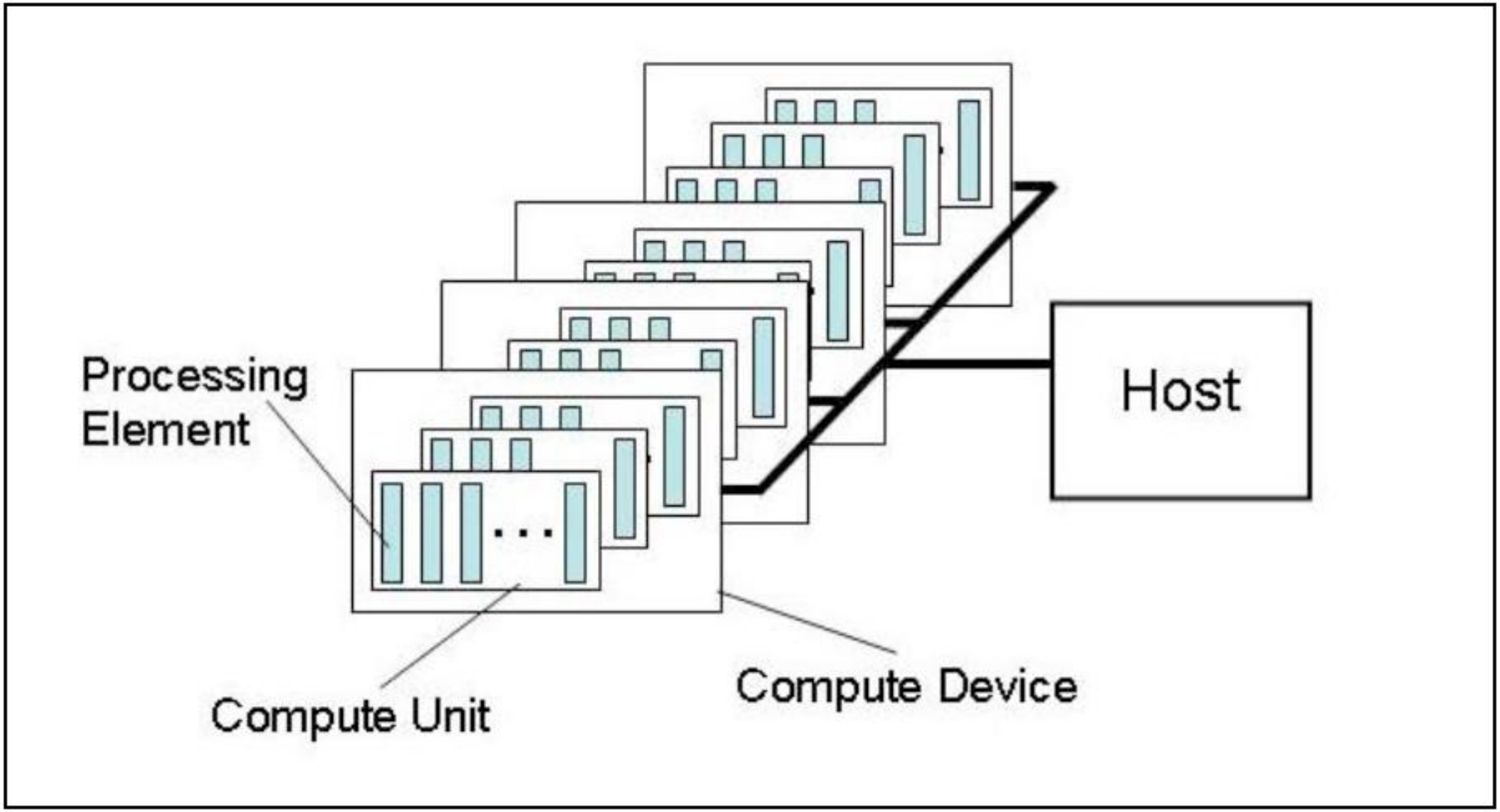
OpenCL Platform Model

1 Host + 1 or more compute devices

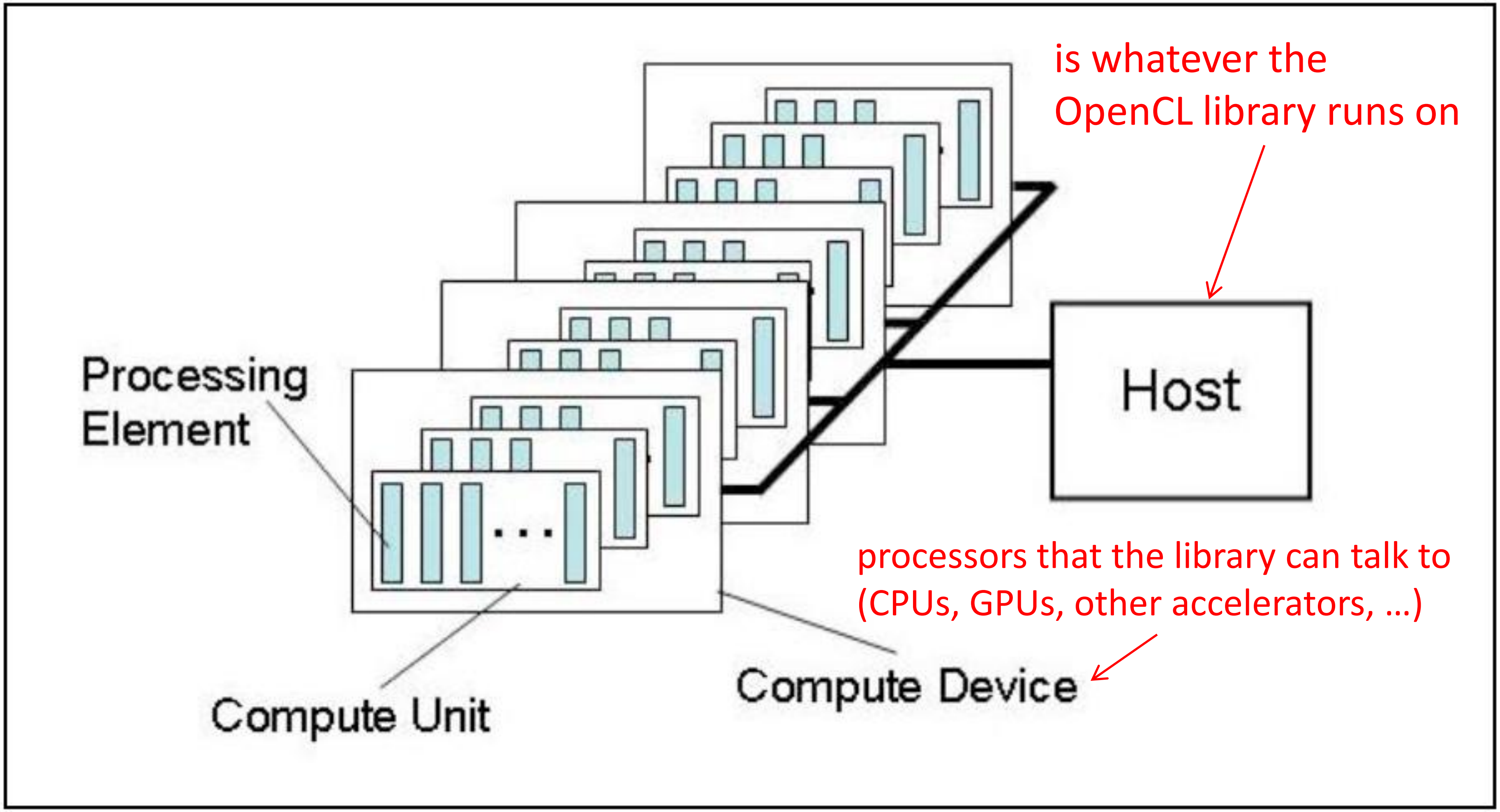
1 or more compute units

1 or more processing elements

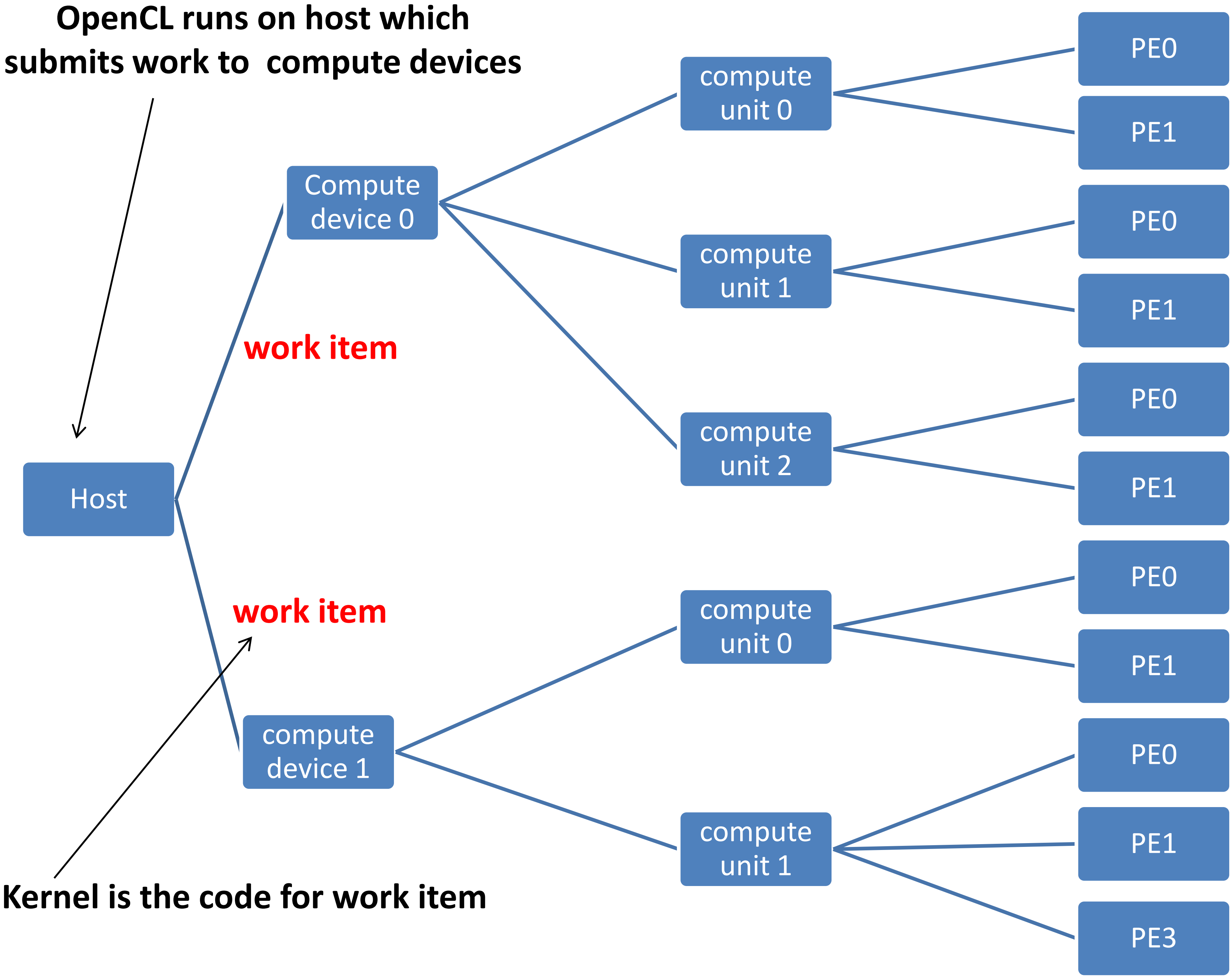
OpenCL Platform Model



OpenCL Platform Model



Each processing element maintains its own program counter.

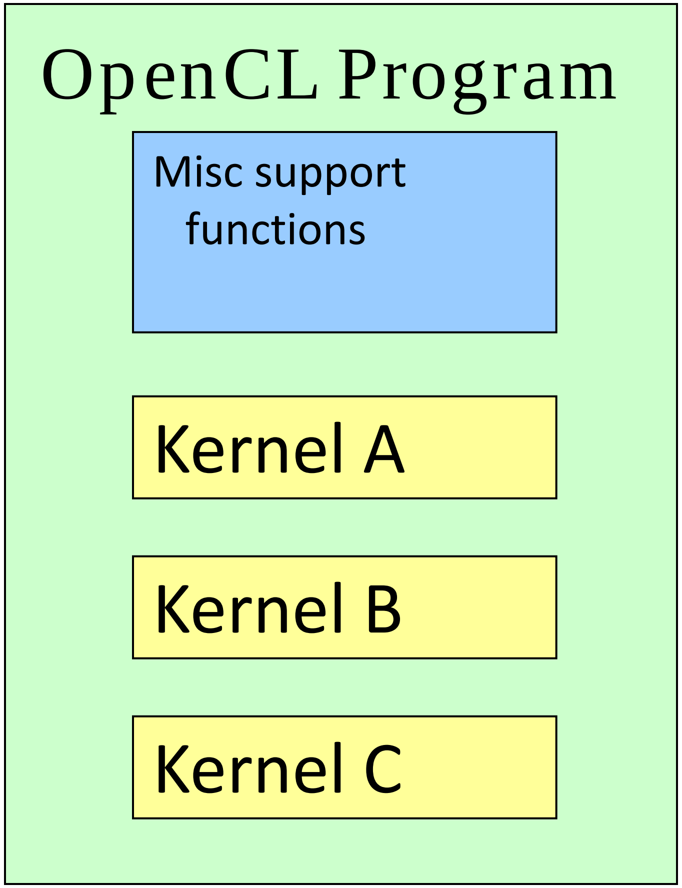

OpenCL runs on host which
submits work to compute devices
Host
Compute
device 0
compute
device 1
work item
work item
compute
unit 0
compute
unit 1
compute
unit 2
compute
unit 0
compute
unit 1
PE0
PE1
PE0
PE1
PE0
PE1
PE0
PE1
PE0
PE1
PE3
Kernel is the code for work item

A Bit of Vocabulary

- Kernel: Smallest unit of execution, like a C function
- Host program: A collection of kernels
- Work group: a collection of work items
  - Has a unique work-group ID
  - work items can synchronize
- Work item: an instance of kernel at run time
  - Has a unique ID within the work-group

OpenCL Programs

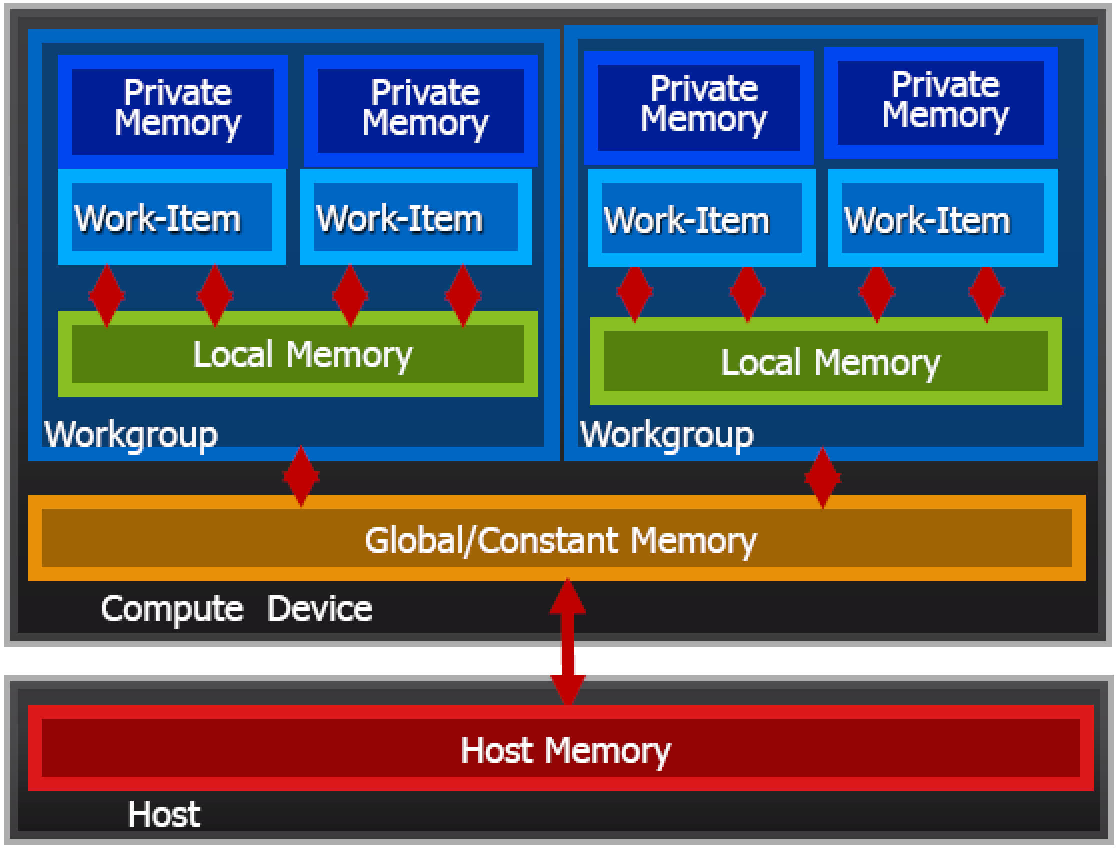
- An OpenCL "program" contains one or more "kernels" and any supporting routines that run on a target device
- An OpenCL kernel is the basic unit of parallel code that can be executed on a target device



Mapping OpenCL concepts to CUDA

OpenCL	CUDA
• kernel	• kernel
• Host program	• Host program
• NDRange	• Grid
• work item	• Thread
• work group	• Block

OpenCL Memory Model



- Relaxed consistency model
- Implementations map this hierarchy to available memories

OpenCL Memory Model

- Memory management is explicit
  - Must move data from host memory to device global memory, from global memory to local memory, and back
- Work-groups are assigned to execute on compute-units
  - No guaranteed coherency among different work-groups

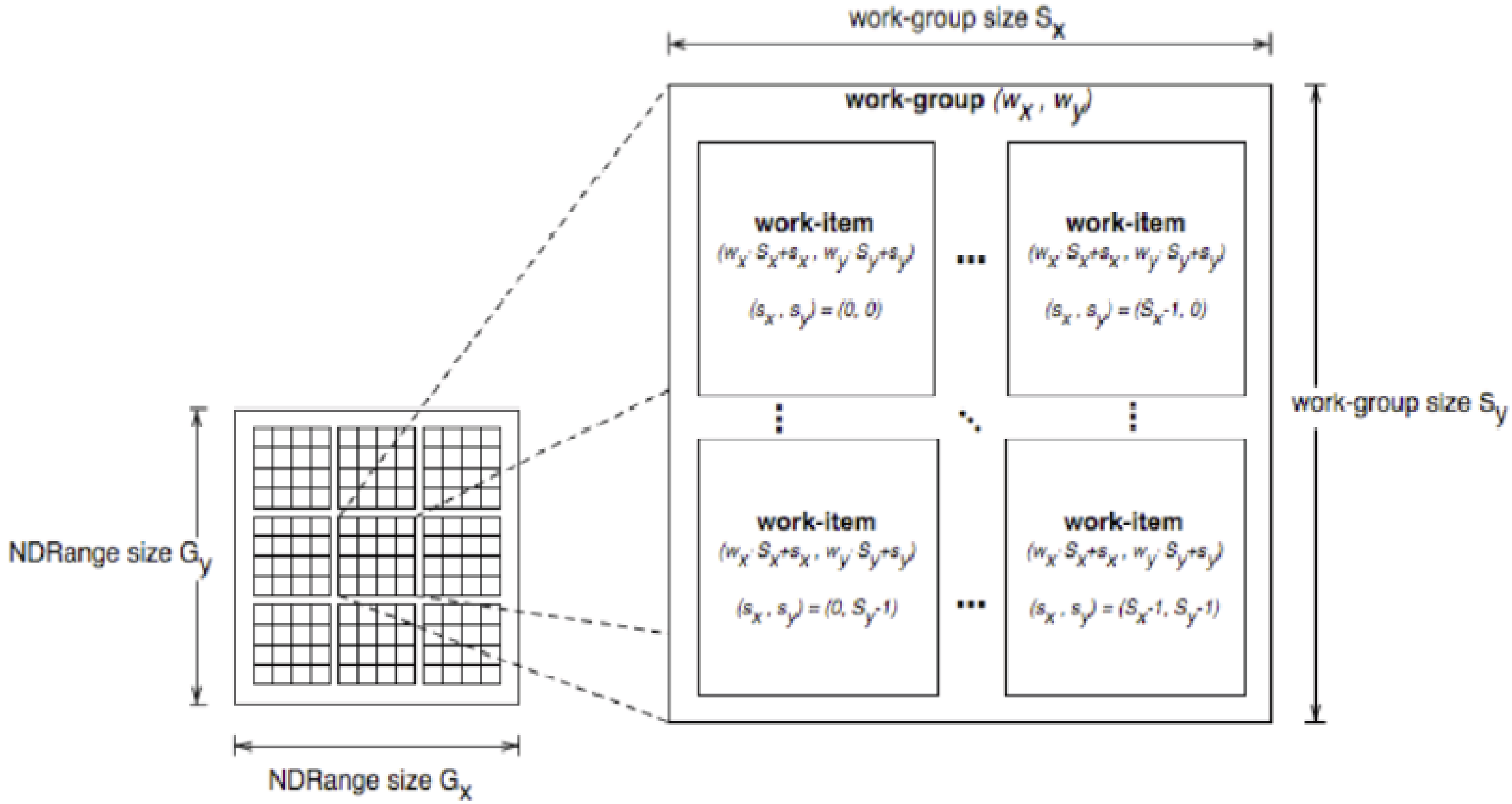
OpenCL Memories

- __global – large, long latency
- __private – on-chip device registers
- __local – memory accessible from multiple PEs or work items. May be SRAM or DRAM.
- __constant – read-only constant cache
- Device memory is managed explicitly by the programmer, as with CUDA

NDRange

- N-Dimensional Range
- N = 1D, 2D, or 3D
- An index space in which kernels are executed
- A work-item is a single kernel instance at a point in the index space

Kernel Execution



In OpenCL, unlike CUDA, a thread can have its own global unique ID by calling: **get_global_id(x)** where x is the dimension (0, 1, or 2)

Mapping OpenCL Dimensions to CUDA Dimensions

OpenCL	CUDA
get_global_id(0)	blockIdx.x*blockDim.x+threadIdx.x
get_local_id(0)	threadIdx.x
get_global_size(0)	gridDim.x*blockDim.x
get_local_size(0)	blockDim.x

Structure of OpenCL main program

1. Get information about platform and devices available on system
2. Select devices to use
3. Create an OpenCL command queue
4. Create memory buffers on device
5. Transfer data from host to device memory buffers
6. Create kernel program object
7. Build (compile) kernel in-line (or load precompiled binary)
8. Create OpenCL kernel object
9. Set kernel arguments
10. Execute kernel
11. Read kernel memory and copy to host memory.

A Platform Is:

"The host plus a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform."

clGetPlatformIDs()

Simple code for identifying platform

```
//Platform
cl_platform_id   * platform;
cl_uint *nump, num;
clGetPlatformIDs (num, platform, nump);
```

Number of platform entries

List of OpenCL platforms found. (Platform IDs)
In our case just one platform, identified by &platform

Returns number of OpenCL platforms available. If NULL, this argument is ignored.

```
cl_int status;
//Retrieve number of platforms
cl_uint numPlatforms = 0;
status = clGetPlatformIDs(0, NULL, &numPlatforms);

//Allocate enough space for each platforms
cl_platform_id * platforms = NULL;
platforms = (cl_platform_id *)malloc(numPlatforms * sizeof(cl_platform_id));

//Fill in the platforms
status = clGetPlatformIDs(numPlatforms, platforms, NULL);
```

Once a platform is selected, we can then query for the devices that it knows how to interact with

clGetDeviceIDs[4] (cl_platform_id *platform*,
cl_device_type *device_type*,
cl_uint *num_entries*,
cl_device_id **devices*,
cl_uint **num_devices*)

We can specify which type of devices we are interested in (e.g. all devices, CPUs only, GPUs only) :

- CL_DEVICE_TYPE_CPU
- CL_DEVICE_TYPE_GPU
- CL_DEVICE_TYPE_ACCELERATOR
- CL_DEVICE_TYPE_DEFAULT
- CL_DEVICE_TYPE_ALL

```
cl_device_id devices[100];
cl_uint devices_n = 0;

clGetDeviceIDs(platforms[0],
          CL_DEVICE_TYPE_GPU,
                              100,
                          devices,
                      &devices_n);
```

Also check: clGetDeviceInfo()

A Context

- A context refers to the environment for managing OpenCL objects and resources
- To manage OpenCL programs, the following are associated with a context
  - Devices: the things doing the execution
  - **Program objects**: the program source that implements the kernels
  - Kernels: functions that run on OpenCL devices
  - **Memory objects**: data that are operated on by the device
  - Command queues: mechanisms for interaction with the devices

Create A Context

```
clCreateContext( cl_context_properties *properties,
                 cl_uint num_devices,
                 const cl_device_id *devices,
                 void *pfn_notify (const char *errinfo,
                                   const void *private_info,
                                   size_t cb,
                                   void *user_data),
                 void *user_data,
                 cl_int *errcode_ret)
```

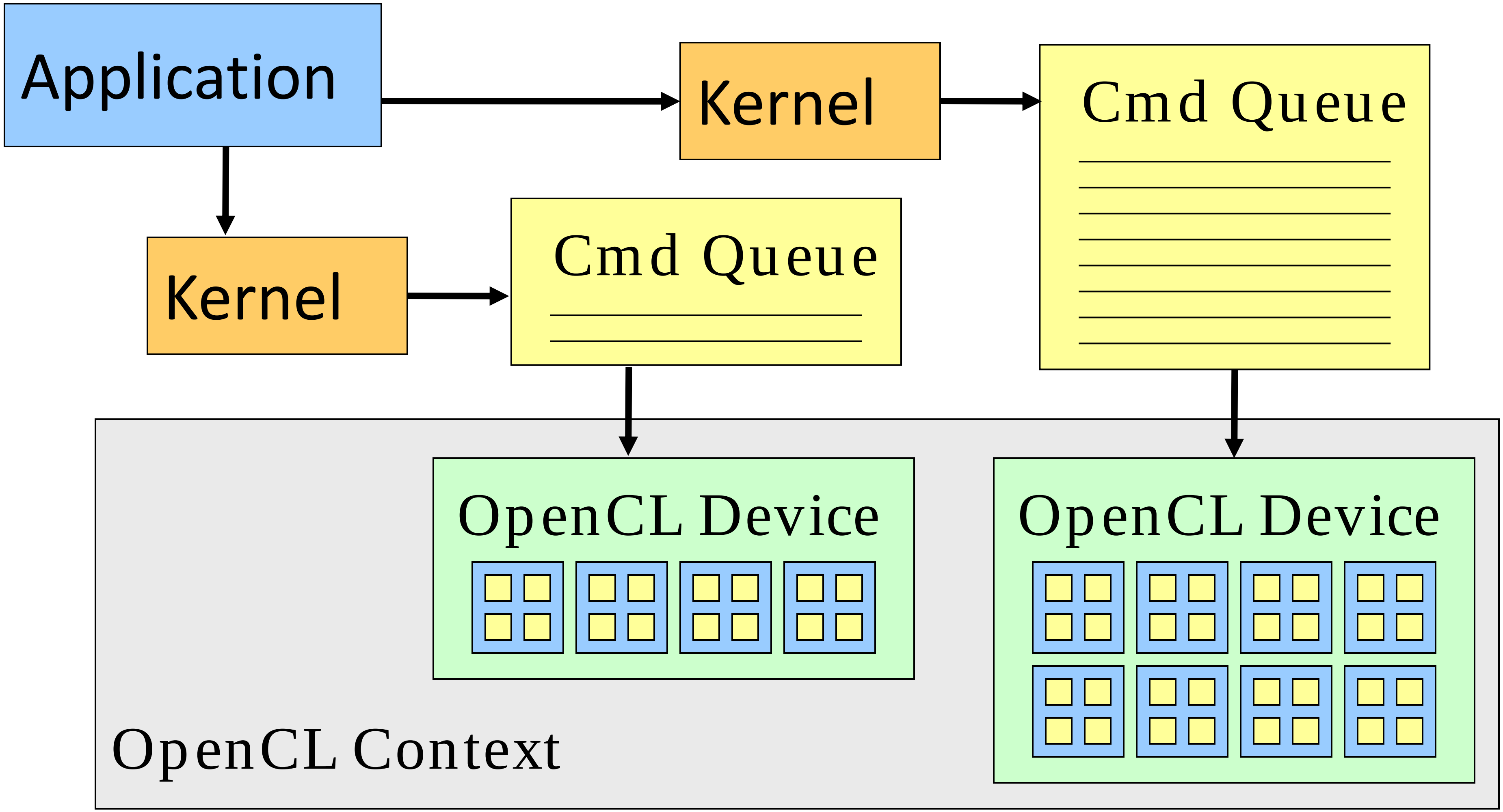
Callback function set by the OpenCL implementation to report information on errors that occur in this context.

All arguments above can be NULL except num_devices and device.

Command Queues

- *A command queue* is the mechanism for the host to request that an action be performed by the device
  - Perform a memory transfer, begin executing, etc.
- A separate command queue is required for each device
- Commands within the queue can be synchronous or asynchronous
- Commands can execute in-order or out-of-order

OpenCL Kernel Execution Launch



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

- Specify whether the queue is executed in-order or out-of-order.
- Default is in-order
- OtherwiseL CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE

Memory Objects

- Memory objects are OpenCL data that can be moved on and off devices
  - Objects are classified as either buffers or images
- Buffers
  - Contiguous chunks of memory – stored sequentially and can be accessed directly (arrays, pointers, structs)
  - Read/write capable
- Images
  - Opaque objects (2D or 3D)
  - Can only be accessed via **read_image()** and **write_image()**
  - Can either be read or written in a kernel, but not both

Allocating Memory on Device

Use clCreatBuffer:

OpenCL context

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

Bit field to specify type of allocation/usage (CL_MEM_READ_WRITE ,…)

Ptr to buffer data (May be previously allocated.)

Returns memory object

Returns error code if an error

Flags in clCreateBuffer()

- CL_MEM_READ_WRITE
- CL_MEM_WRITE_ONLY
- CL_MEM_READ_ONLY

From kernel perspective

- CL_MEM_COPY_HOST_PTR
  - valid only if host_ptr is not NULL
  - OpenCL allocates memory in the device for the memory object
  - copy the data from memory referenced by host_ptr.
- CL_MEM_ALLOC_HOST_PTR
  - only allocates memory on the host
  - which does not incur any transfer at all
  - Mutually exclusive with CL_MEM_USE_HOST_PTR
- CL_MEM_USE_HOST_PTR
  - valid only if host_ptr is not NULL
  - OpenCL uses a memory referenced by host_ptr as the storage for the memory object
  - OpenCL implementations are allowed to cache the buffer contents pointed to by host_ptr in device memory
  - most likely allocates *pinned* memory

Transferring Data

- OpenCL provides commands to transfer data to and from devices
  - clEnqueue{Read|Write}{Buffer|Image}
  - Copying from the host to a device is considered *writing*
  - Copying from a device to the host is *reading*

Transferring Data

```
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)
```

- This command initializes the OpenCL memory object and writes data (pointed by ptr) to the device (buffer starting from offset) associated with the command queue
- The command will write cb bytes from a host pointer (*ptr*) to the device
- The *blocking_write* parameter specifies whether or not the command should return before the data transfer is complete (CL_TRUE and CL_FLASE)
- event_wait_list and num_events_in_wait_list specify events that need to complete before this particular command can be executed.
- event returns an event object that identifies this particular write command.

Compilation Model

- More complicated than CUDA
- uses Dynamic/Runtime compilation model
  1. The code is complied to an Intermediate Representation (IR)
  2. The IR is compiled to a machine code for execution.
- Rule of thumb: Starting a kernel can be expensive, so try to make individual kernels do a large amount of work.

Typical OpenCL Program Flow

1. Select the desired devices (ex: all GPUs)
2. Create a context
3. Create command queues (per device)
4. Allocate memory on devices
5. Transfer data to devices
6. Compile programs
7. Create kernels
8. Execute
9. Transfer results back
10. Free memory on devices

- **clCreateProgramWithSource**
- **clBuildProgram**
- **clCreateKernel**

Let's See an Example

- Adding two vectors A and B and put the results in C.
- source:

http://www.thebigblob.com/getting-started-with-opencl-and-gpu-computing/

Let's write the kernel in separate file

```
__kernel void vector_add(__global const int *A,
                         __global const int *B,
                         __global int *C) {

    // Get the index of the current element to be processed
    int i = get_global_id(0);

   // Do the operation
   C[i] = A[i] + B[i];
}
```

file: kernel_add.cl

Setting The Stage

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

#define MAX_SOURCE_SIZE (0x100000)

int main(void) {
    // Create the two input vectors
    int i;
    const int LIST_SIZE = 1024;
    int *A = (int*)malloc(sizeof(int)*LIST_SIZE);
    int *B = (int*)malloc(sizeof(int)*LIST_SIZE);
    for(i = 0; i < LIST_SIZE; i++) {
        A[i] = i;
        B[i] = LIST_SIZE - i;
    }
```

Setting The Stage (cont'd)

```
// Load the kernel source code into the array source_str
FILE *fp;
char *source_str;
size_t source_size;

fp = fopen("kernel_add.cl", "r");
if (!fp) {
   fprintf(stderr, "Failed to load kernel.\n");
   exit(1);
}
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
fclose( fp );
```

Step 1: Select Desired Device

```
// Get platform and device information
  cl_platform_id platform_id = NULL;
  cl_device_id device_id = NULL;
  cl_uint ret_num_devices;
  cl_uint ret_num_platforms;

  cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);

  ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
        &device_id, &ret_num_devices);
```

Step 2: Create a Context

```
// Create an OpenCL context
   cl_context context = clCreateContext( NULL, 1, &device_id,
                                         NULL, NULL, &ret);
```

Step 3: Create Command Queues

```
// Create a command queue
   cl_command_queue command_queue =
clCreateCommandQueue(context, device_id, 0, &ret);
```

Step 4: Allocate Memory on Device

```
// Create memory buffers on the device for each vector
   cl_mem a_mem_obj = clCreateBuffer(context,
CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int), NULL, &ret);

   cl_mem b_mem_obj = clCreateBuffer(context,
CL_MEM_READ_ONLY, LIST_SIZE * sizeof(int), NULL, &ret);

   cl_mem c_mem_obj = clCreateBuffer(context,
CL_MEM_WRITE_ONLY, LIST_SIZE * sizeof(int), NULL, &ret);
```

Step 5: Transfer Data to Device(s)

```
// Copy the lists A and B to their respective memory buffers
  ret = clEnqueueWriteBuffer(command_queue, a_mem_obj,
CL_TRUE, 0, LIST_SIZE * sizeof(int), A, 0, NULL, NULL);

  ret = clEnqueueWriteBuffer(command_queue, b_mem_obj,
CL_TRUE, 0, LIST_SIZE * sizeof(int), B, 0, NULL, NULL);
```

Step 6 & 7:
Build Program and Compile Kernel

```
// Create a program from the kernel source
   cl_program program = clCreateProgramWithSource(context, 1, (const char **)&source_str, (const size_t *)&source_size, &ret);

   // Build the program
  ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

// Create the OpenCL kernel
   cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);

   // Set the arguments of the kernel
   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
   ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
   ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);
```

```
cl_program clCreateProgramWithSource
(           cl_context context,
            cl_uint count,
            const char **strings,
            const size_t *lengths,
            cl_int *errcode_ret)
```

A pointer to *count* arrays containing the source code of the kernel.

The lengths of strings in the source code.

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.

- A pointer to a list of devices that are in program.
- If device_list is NULL value, the program executable is built for all devices associated with program.

```
cl_int clBuildProgram (
    cl_program program,
    cl_uint num_devices,
    const cl_device_id *device_list,
    const char *options,
    void (*pfn_notify)(cl_program, void *user_data),
    void *user_data)
```

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

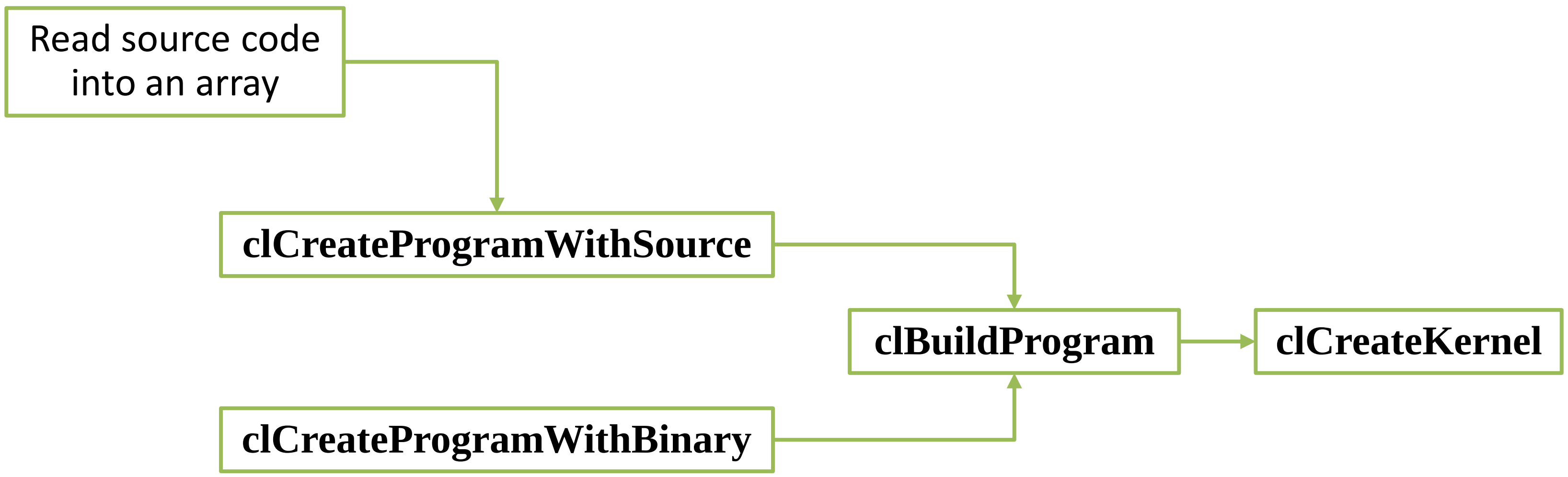
```
cl_kernel clCreateKernel (cl_program program,
                          const char *kernel_name,
                          cl_int *errcode_ret)
```

- **program**: a successfully built program.
- **kernel_name**: is a function name in kernel file(.cl file)
- **errcode_ret**: Returns an appropriate error code.
  If errcode_ret is NULL, no error code is returned.

clSetKernelArg(kernel, arg_index, sizeof(arg), &arg)

- from left to right
- one clSetKernelArg per argument per kernel

Runtime Compilation of OpenCL kernels



Step 8: Execute the Kernel

```
// Execute the OpenCL kernel on the list
 size_t global_item_size = LIST_SIZE; // Process the entire lists

 size_t local_item_size = 64; // Divide work items into groups of 64

ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
NULL, &global_item_size, &local_item_size, 0, NULL, NULL);
```

Always NULL for this revision. (offset)

An array of "dimension" elements. Each element specifies the number of items in that dimension.

Specify events that need to complete before this particular command can be executed.

Dimension must be 1, 2, or 3

[cl_int](#) **clEnqueueNDRangeKernel** ([cl_command_queue](#) *command,*
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*)

- ***work_dim**:* The number of dimensions used to specify the global work-items and work-items in the work-group.
  Must be greater than zero and less than or equal to three.
- **global_work_offset**: currently set to NULL
- ***global_work_size**:* Points to an array of work_dim unsigned values that describe the number of global work-items in work_dim dimensions that will execute the kernel function.
- ***local_work_size:*** Points to an array of work_dim unsigned values that describe the number of work-items that make up a work-group

Note About Local/Global Work Items

- If local_work_size is specified, the values specified in global_work_size[0],...global_work_size[work_dim - 1] must be evenly divisible by the corresponding values specified in local_work_size[0],... local_work_size[work_dim - 1].

- If local_work_size is NULL, the OpenCL implementation will determine how to be break the global work-items into appropriate work-group instances.

Step 9: Transfer Result Back

```
// Read the memory buffer C on the device to the local variable C
 int *C = (int*)malloc(sizeof(int)*LIST_SIZE);

ret = clEnqueueReadBuffer(command_queue, c_mem_obj,
CL_TRUE, 0, LIST_SIZE * sizeof(int), C, 0, NULL, NULL);
```

Step 10: Free Memory on Devices

```
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(a_mem_obj);
ret = clReleaseMemObject(b_mem_obj);
ret = clReleaseMemObject(c_mem_obj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);

free(A);
free(B);
free(C);
return 0; }
```

clFlush: Issues all previously queued OpenCL commands in a command-queue to the device associated with the command-queue.

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

Compiling an OpenCL program on CUDA GPU

Don't forget, in Linux, to:

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

```
nvcc  myprog.c -l OpenCL
```

That was data parallel mode. How about task parallelism?

- Several kernels with clCreateKernel()
  - e.g. kernel[0], kernel[1], ...
- Command queue can be out-of-order to increase chance of parallelism
- clEnqueueTask(command_queue, kernel[i], ...)
- Use events to manage data dependencies.

Events

- An event object contains information about the execution status of queued commands.
- This object is returned on all commands that start with *clEnqueue*
- *Example:*

```
cl_int clEnqueueTask (cl_command_queue command_queue,
                      cl_kernel kernel,
                      cl_uint num_events_in_wait_list,
                      const cl_event *event_wait_list,
                      cl_event *event)
```

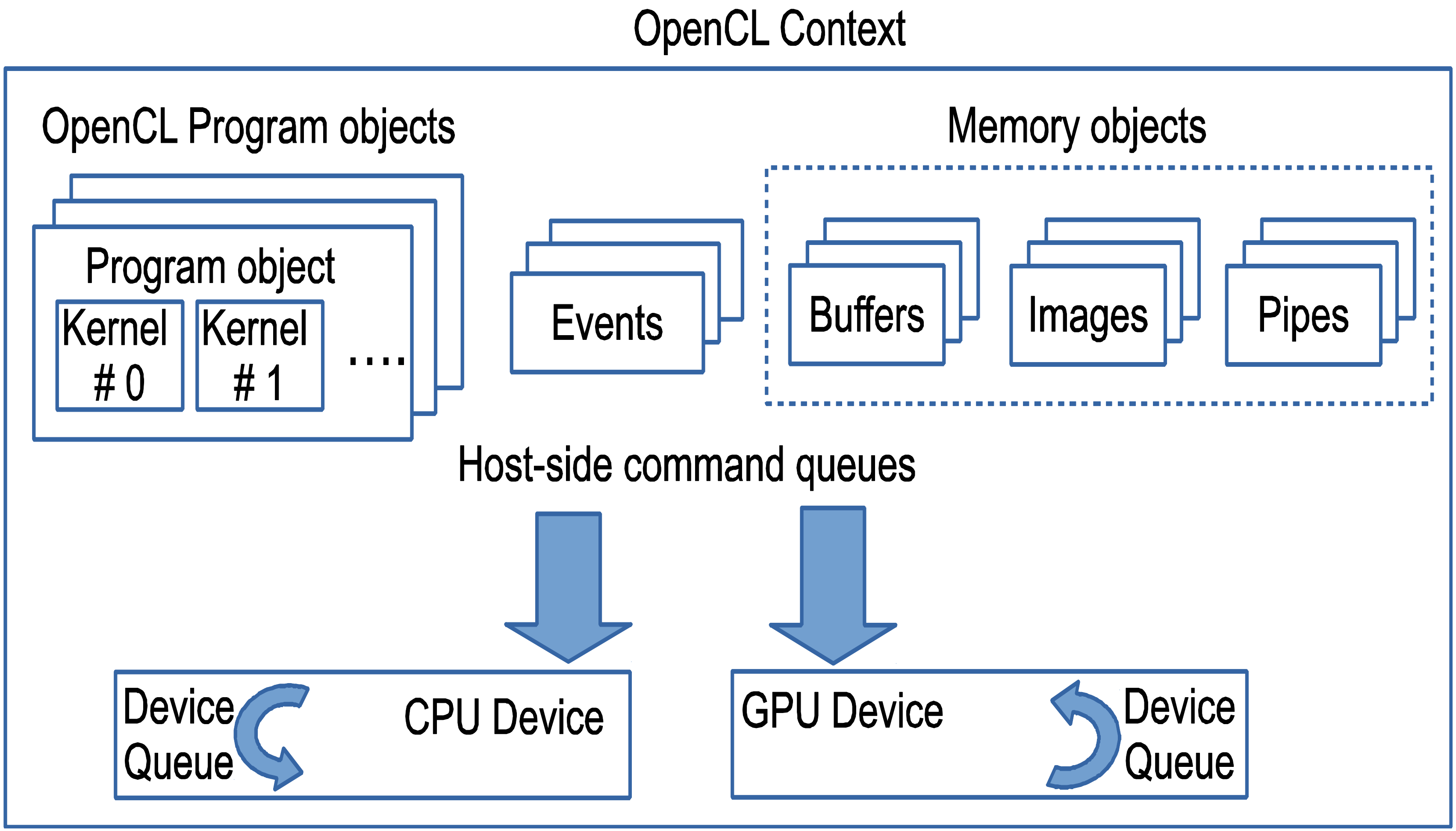
Events

- With an in-order command queue (default), each command will complete before the next command begins, so manually specifying dependencies is not required

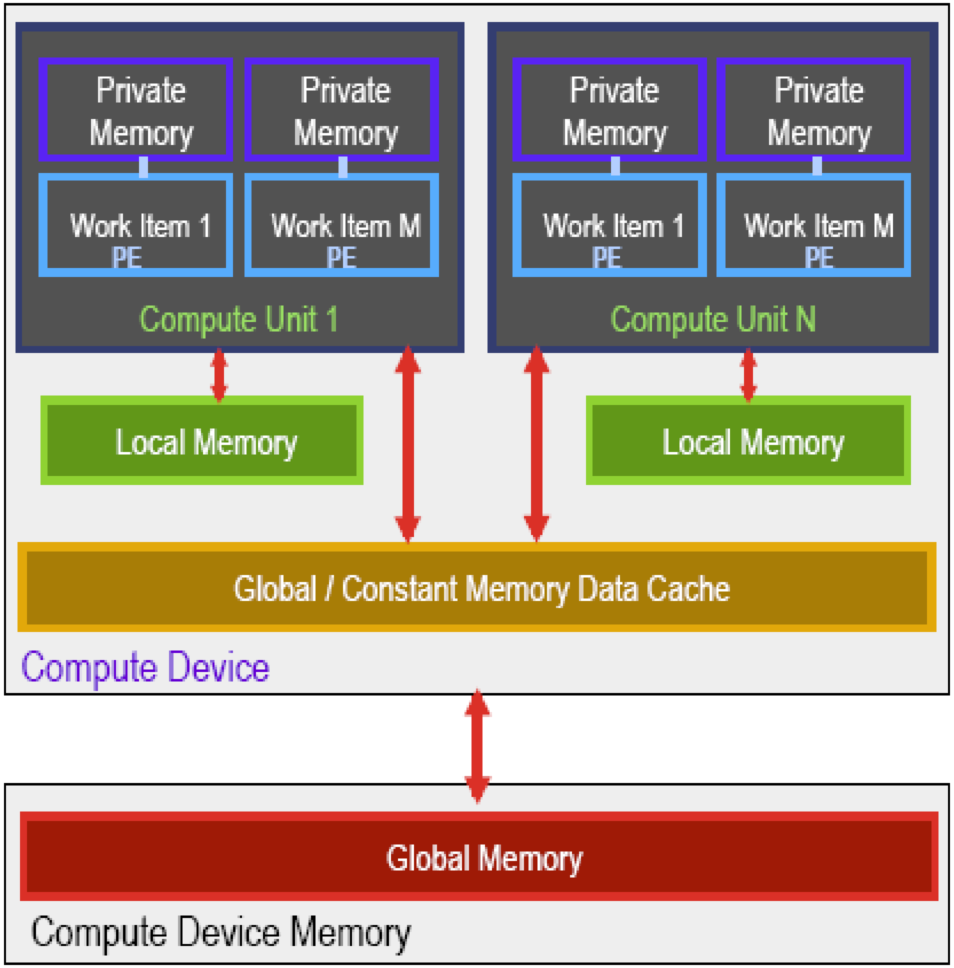
New in OpenCL 2.0

- Device side enqueuing
  - Allows a device to enqueue commands to itself
  - Device-side command queues are out-of-order
- New memory object: pipes
  - Ordered sequence of data items called packets
  - Stored on the basis of a first in, first out method
  - Can only be accessed via intrinsics **read_pipe()** and **write_pipe()**

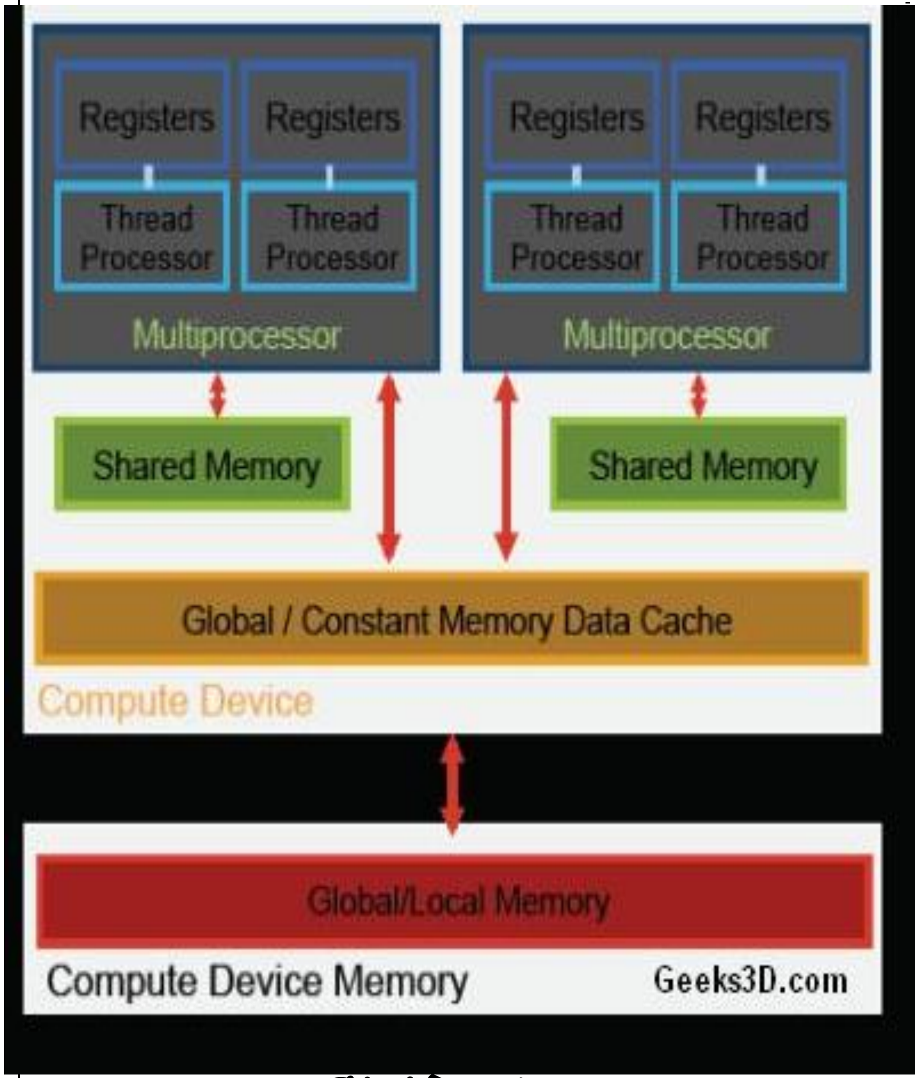
Big Picture


OpenCL Context
OpenCL Program objects
Program object
Kernel # 0
Kernel # 1
....
Events
Memory objects
Buffers
Images
Pipes
Host-side command queues
Device Queue
CPU Device
GPU Device
Device Queue

Memory Model Comparison



OpenCL

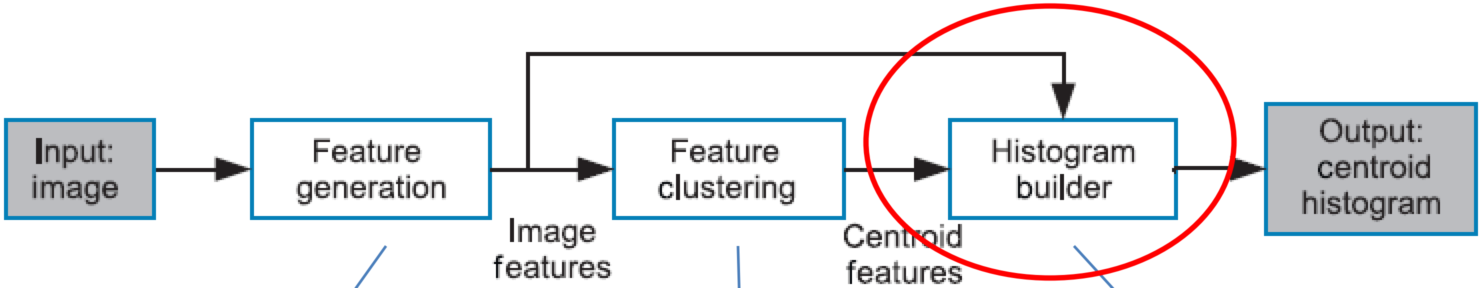


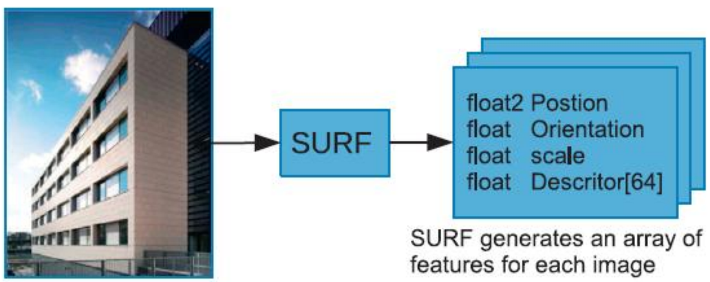
CUDA

Case Study: Image Clustering

Problem definition

- The bag-of-words (BoW) model:
  - One of the most popular approaches to image classification
  - Treats an image's features as words.
  - Represents the image as a vector of the occurrence counts of image features (words).
- We will discuss:
  - OpenCL implementation of the histogram builder → a very important component of BoW.





SURF = Speed-UP Robust Feature

convert a list descriptors
into a histogram of visual words

- Features are quantized
- Typically by k-means clustering
- And mapped into clusters.
- The centroid of each cluster is known as a visual word

Histogram Builder

- **GOAL:** Determine to which centroid each descriptor belongs.
- **Input data:** Both the descriptors of the SURF features and the centroid have 64 dimensions.
- **Method:**
  1. Compute the Euclidean distance between the descriptor and all the centroids.
  2. Assign each SURF descriptor to its closest centroid.
- **Output:** The histogram is formed by counting the number of SURF descriptors assigned to each centroid.

First: CPU Implementation

```
// Loop over all the descriptors generated for the image
for(int i = 0; i < n_desc; i++)
{
  membership = 0;
  min_dist = FLT_MAX;
  // Loop over all the cluster centroids available
  for(j = 0 ; j < n_cluster; j++)
  {
    dist = 0;
    // n_features: No. of elements in each descriptor (64)
    // Calculate the distance between the descriptor and the centroid
    for(k = 0 ; k < n_features; k++)
    {
      dist_temp = surf[i][k]-cluster[j][k];
      dist += dist_temp * dist_temp;
    }
    // Update the minimum distance
    if(dist < min_dist)
    {
      min_dist = dist;
      membership = j;
    }
  }
  // Update the histogram location of the closest centroid
  histogram[membership] += 1;
}
```

Second: GPU Implementation #1 with OpenCL

```
__kernel
void kernelGPU1(
    __global float *descriptors,
    __global float *centroids,
    __global int *histogram,
    int n_descriptors,
    int n_centroids,
    int n_features)
{
    // Global ID identifies SURF descriptor
    int desc_id = get_global_id(0);

    int membership = 0;
    float min_dist = FLT_MAX;

    // For each cluster, compute the membership
    for(int j = 0; j < n_centroids; j++) {

        float dist = 0;

        // n_features: No. of elements in each descriptor (64)
        // Calculate the distance between the descriptor and the
           centroid
        for(int k = 0; k < n_features; k++) {
            float temp = descriptors[desc_id*n_features+k] -
                centroids[j*n_features+k];
            dist += temp*temp;
        }

        // Update the minimum distance
        if(dist < min_dist) {
            min_dist = dist;
            membership = j;
        }
    }

    // Atomic increment of histogram bin
    atomic_fetch_add_explicit(&histogram[membership], 1,
        memory_order_relaxed, memory_scope_device);
}
```

Atomically:

- fetch data pointed to by &histogram[]
- Add 1 to it
- Store the result back to & histogram[]

Second: GPU Implementation #1 with OpenCL

```
__kernel
void kernelGPU1(
    __global float *descriptors,
    __global float *centroids,
    __global int *histogram,
    int n_descriptors,
    int n_centroids,
    int n_features)
{
    // Global ID identifies SURF descriptor
    int desc_id = get_global_id(0);

    int membership = 0;
    float min_dist = FLT_MAX;

    // For each cluster, compute the membership
    for(int j = 0; j < n_centroids; j++) {

        float dist = 0;

        // n_features: No. of elements in each descriptor (64)
        // Calculate the distance between the descriptor and the
            centroid
        for(int k = 0; k < n_features; k++) {
            float temp = descriptors[desc_id*n_features+k] -
                centroids[j*n_features+k];
            dist += temp*temp;
        }

        // Update the minimum distance
        if(dist < min_dist) {
            min_dist = dist;
            membership = j;
        }
    }

    // Atomic increment of histogram bin
    atomic_fetch_add_explicit(&histogram[membership], 1,
        memory_order_relaxed, memory_scope_device);
}
```

What do you think about
the memory access of this line?

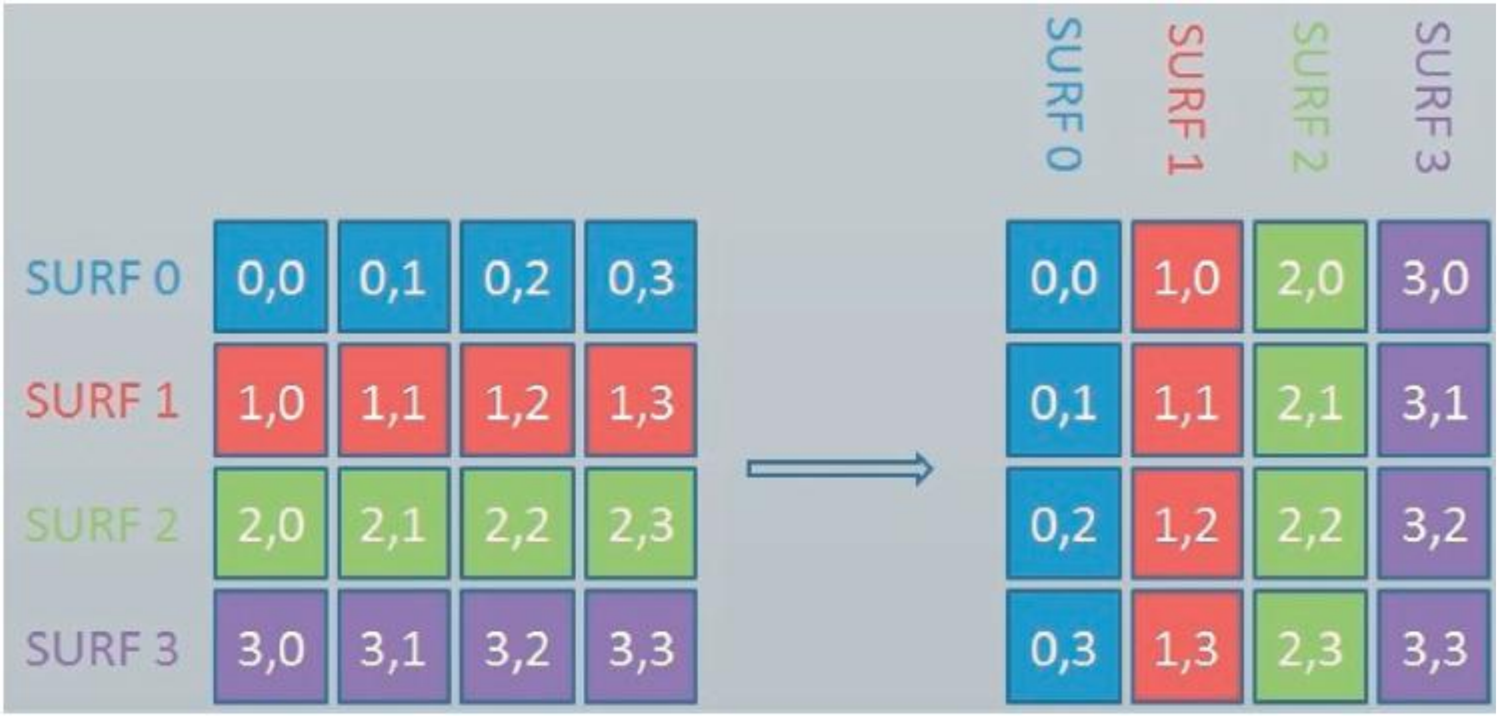
```
float temp = descriptors[desc_id*n_features+k] -
    centroids[j*n_features+k];
```

- Suppose we have 4 work-items in a workgroup.
- They have IDs: 0, 1, 2, and 3.
- n_features = 64
- Let's take the first iteration of k (i.e. k = 0)
- Those work items will access:
  - descriptors[0]
  - descriptors[64]
  - descriptors[128]
  - descriptors[192]

Pretty Bad .. Indeed!

Cannot be coalesced ☹

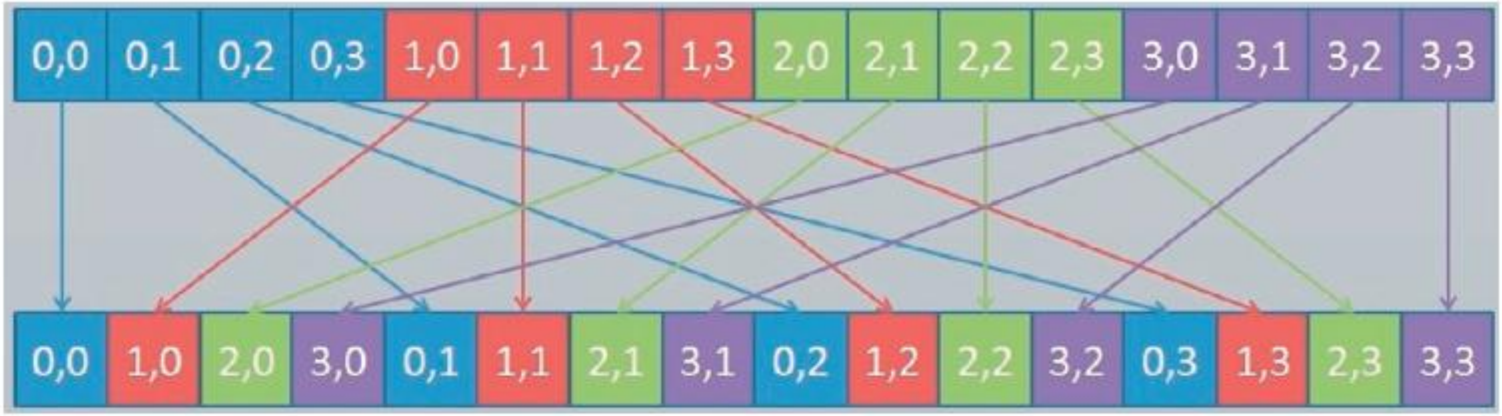
We need to change the way descriptors[] is accessed



A small kernel where each thread reads one element of the input matrix from global memory and writes back the same element at its transposed index in the global memory. Called before the main kernel of the historgram.

Transpose

We need to change the way descriptors[] is accessed

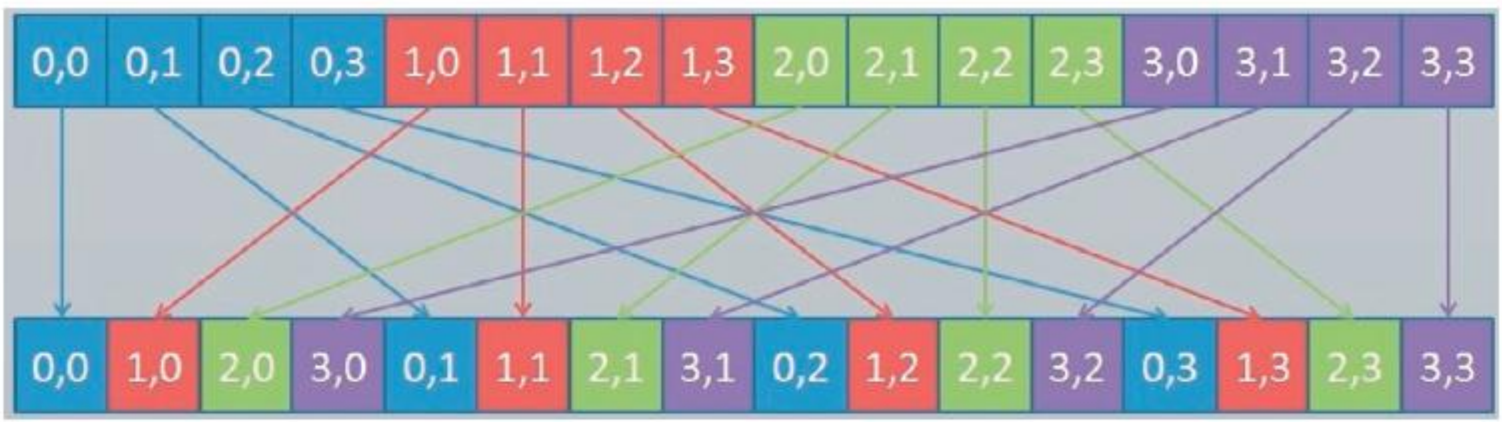


A small kernel where each thread reads one element of the input matrix from global memory and writes back the same element at its transposed index in the global memory. Called before the main kernel of the historgram.

Transpose

Now we make a small change in the main kernel

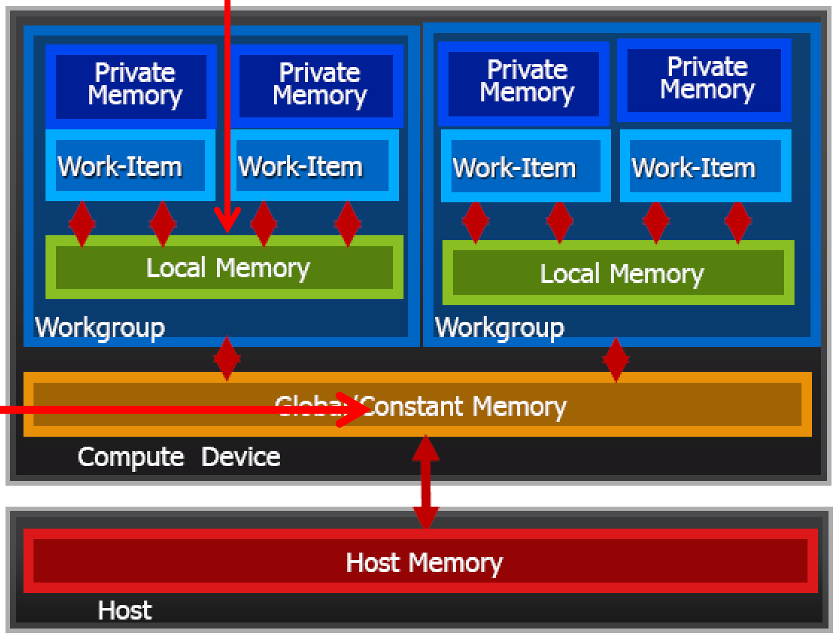
```
// n_features: No. of elements in each descriptor (64)
// Calculate the distance between the descriptor and the
    centroid
for(int k = 0; k < n_features; k++) {
    float temp = descriptors[k*n_descriptors+desc_id] -
        centroids[j*n_features+k];
    dist += temp*temp;
}
```



Another Optimization

```
for(int k = 0; k < n_features; k++) {
    float temp = descriptors[k*n_descriptors+desc_id] –
        centroids[j*n_features+k];
    dist += temp*temp;
}
```

- Data in these buffers are accessed multiple times
- centroids[] access is independent of thread ID



Local Memory (From OpenCL Perspective)

- Local memory is a high bandwidth low-latency memory used for sharing data among work-items within a work-group.
- However:
  - local memory has limited size
  - on AMDRadeon HD 7970 GPU there is 64 KB of local memory per compute unit, with the maximum allocation for a single work-group limited to 32 KB.

```c
__kernel
void kernelGPU4(
    __global float *descriptors,
    __global float *centroids,
    __global int *histogram,
    int n_descriptors,
    int n_centroids,
    int n_features)
{

    // Global ID identifies SURF descriptor
    int desc_id = get_global_id(0);
    int local_id = get_local_id(0);
    int local_size = get_local_size(0);

    // Store the descriptors in local memory
    __local float desc_local[4096]; // 64 descriptors * 64 work-items
    for(int i = 0; i < n_features; i++) {
        desc_local[i*local_size + local_id] =
            descriptors[i*n_descriptors + surf_id];
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    int membership = 0;
    float min_dist = FLT_MAX;

    // For each cluster, compute the membership
    for(int j = 0; j < n_centroids; j++) {

        float dist = 0;
        // n_features: No. of elements in each descriptor (64)
        // Calculate the distance between the descriptor and the centroid
        for(int k = 0; k < n_features; k++) {
            float temp = desc_local[k*local_size+local_id] -
                centroids[j*n_features+k];
            dist += temp*temp;
        }

        // Update the minimum distance
        if(dist < min_dist) {
            min_dist = dist;
            membership = j;
        }
    }

    // Atomic increment of histogram bin
    atomic_fetch_add_explicit(&histogram[membership], 1,
        memory_order_relaxed, memory_scope_device);
}
```

Returns the number of local work-items specified in dimension

```
// Store the descriptors in local memory
__local float desc_local[4096]; // 64 descriptors * 64 work-items
for(int i = 0; i < n_features; i++) {
    desc_local[i*local_size + local_id] =
        descriptors[i*n_descriptors + surf_id];
}
barrier(CLK_LOCAL_MEM_FENCE);
```

Mapping the centroids[] buffer to constant memory is as simple as changing the parameter declaration from _ _**global** to _ _**constant**

Constant Memory

- Constant memory is a memory space to hold data that is accessed simultaneously by all work-items
  - Usually maps to specialized caching hardware that has a fixed size
- Advantages for AMD hardware
  - If all work-items access the same address, then only one access request will be generated per wavefront
  - Constant memory can reduce pressure from L1 cache
  - Constant memory has lower latency than L1 cache

Performance on AMD Radeon 7970

No. of Features	Transform Kernel (ms)
4096	0.05
16,384	0.50
65,536	2.14

The small kernel to make the transpose

Performance on AMD Radeon 7970

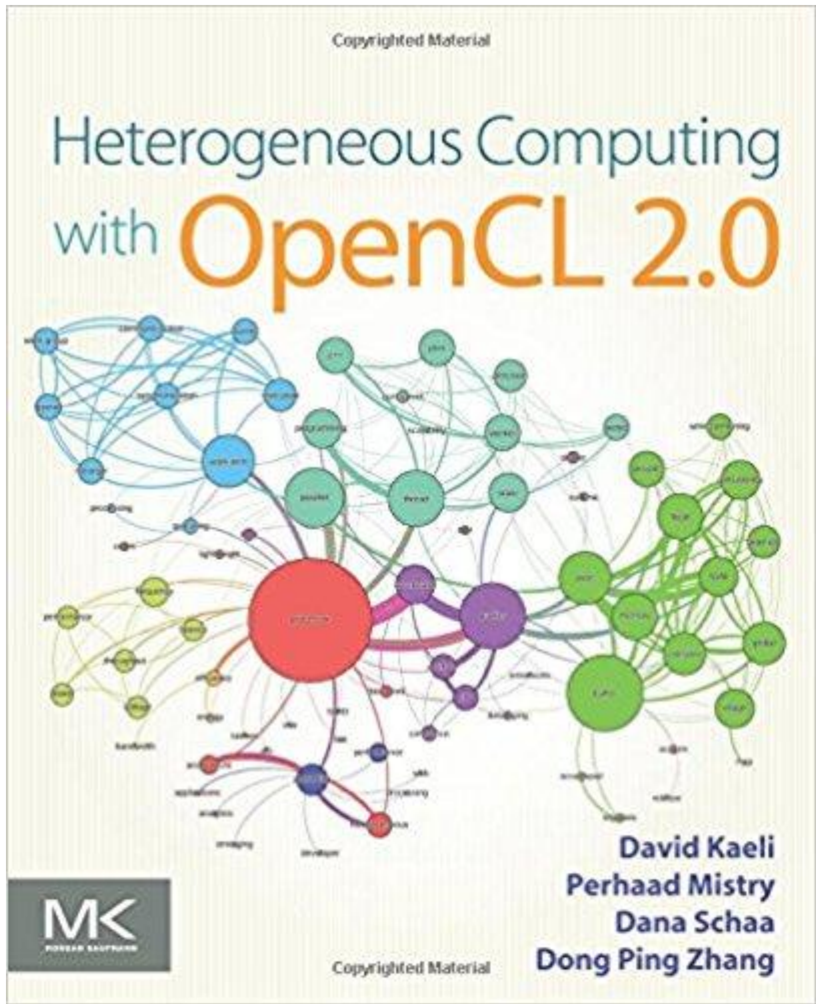
# of Clusters	# of SURF Descriptors	GPU Implementations				
		GPU1	GPU2	GPU3	GPU4	GPU5
8	4096	0.41	0.27	0.10	0.17	0.09
	16,384	3.60	0.28	0.17	0.69	0.19
	65,536	15.36	1.05	0.59	1.31	0.74
16	4096	0.77	0.53	0.19	0.28	0.14
	16,384	7.10	0.53	0.32	0.57	0.29
	65,536	30.41	1.47	1.17	2.26	1.12
64	4096	6.00	3.53	1.34	1.00	0.43
	16,384	28.28	2.11	1.20	2.96	0.86
	65,536	122.09	5.80	4.65	9.04	3.87
128	4096	4.96	4.04	1.47	1.95	0.81
	16,384	55.70	4.27	2.40	5.89	1.61
	65,536	243.30	11.63	9.29	17.46	6.43
256	4096	10.49	8.06	2.84	4.35	1.57
	16,384	109.67	8.62	4.77	11.44	3.13
	65,536	488.54	23.28	18.71	34.73	13.97

kernel running time in ms.

GPU1: original kernel **GPU2**: using transpose **GPU3**: vector (we did not cover)

GPU4: transpose + local mem **GPU5**: transpose + local mem + constant mem

A good book to start learning OpenCL



Conclusions

- OpenCL has high portability to many accelerators
- CUDA has been around for longer -> more libraries and OpenCL is playing catch-up

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