



EEP 524

Applied High-Performance GPU Computing

LECTURE 3 PART 1 :

Instructor: Dr. Colin Reinhardt

EE Professional Masters Program

Winter 2021



UPDATES



- ▶ AWS Cloud VDE online!
 - ▶ Please respond to Poll!
 - ▶ Intro videos Part 1 & 2: nvcc command line and VS CUDA projects
 - ▶ posted on class web Panopto
- ▶ Class DUE Dates revised
 - ▶ see class web Home page
- ▶ Schedule (topics coverage) also being revised



Lecture 3 : Outline

- ▶ Fundamental concepts and techniques
 - ▶ data dimensionality and memory representation
 - ▶ computational domain (index space) dimensionality
 - ▶ mapping problem & index spaces to HW execution
- ▶ The Full vector addition Application (OpenCL)
 - ▶ Host application API
 - ▶ Device kernel language
- ▶ DISCUSSION
- ▶ Ex2 & Hw1
 - ▶ DUE 27 JAN (WED), 11:59 PM
 - ▶ submission will be online via class web Assignments page
 - ▶ There will be a Lec 3 – Part 2 to cover Ex2/Hw1 in more depth

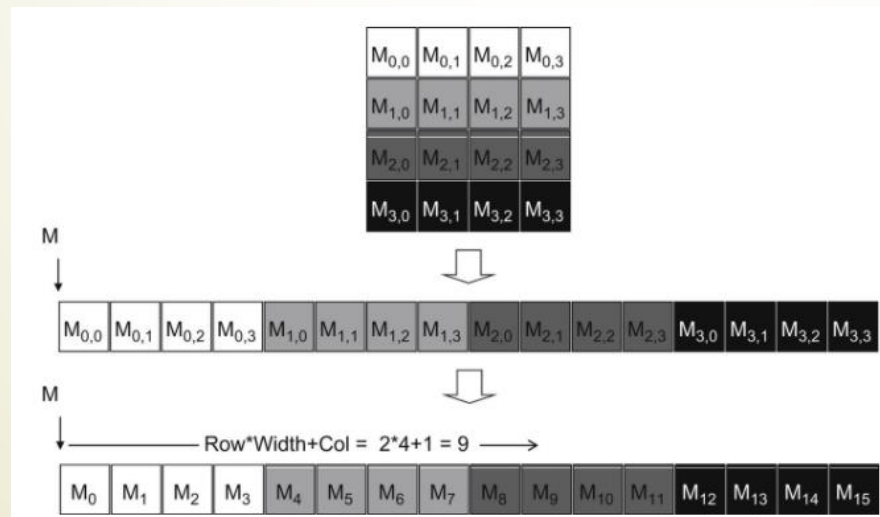


Data Dimensionality: application view and computer view

- ▶ An application domain expert may think of the data for a problem in its “natural” dimensionality
 - ▶ image processing: 2D $h(x,y)$
 - ▶ physical problem: 3D $f(x,y,z)$
- ▶ However, computer memory space is a “flat” address space.
 - ▶ the compiler and processor reference memory using a base + offset
 - ▶ linear, contiguous
- ▶ Thus, all higher-dimensional data representations are ultimately “linearized”
 - ▶ “flattened” into equivalent 1D arrays
 - ▶ multi-dimensional indexing syntax translated into 1D offsets
 - ▶ consider: c-style multi-dimensional arrays
 - ▶ $x[i][j]$
 - ▶ $y[i][j][k]$

“flattened” (linearized) arrays

- Consider 2D matrix (2D C array) : $M(\text{row}, \text{col}) = M(y, x) = M_{y,x} = m[y][x]$
- There are two choices to linearize into a 1D representation
 - row-major : concatenate successive rows
 - column-major : concatenate successive columns
- standard C, CUDA, and OpenCL all adopt the **row-major** convention



NxM matrix

$M(\text{row}, \text{col}) = \text{row} * M + \text{col}$

row-major

$M(\text{row}, \text{col}) = \text{col} * N + \text{row}$

col-major

- How does this translate into 3D...?

$M(z, y, x) = M(\text{slice}, \text{row}, \text{col})$

CUDA and OpenCL memory arrays

Both CUDA and OpenCL provide a generic/arbitrary 1D memory array

This is generally the primary way to exchange data between host and device

- ▶ *except for some specialized cases, which will discuss later: images, textures*
- ▶ **CUDA** : arbitrary linear array of memory bytes

```
float* h_pData = (float*)malloc(sizeof(float)*N); // alloc on host  
cudaMalloc( &d_pData, sizeof(float)*N); // alloc on device  
cudaMemcpy( d_pData, h_pData, sizeof(float)*N, cudaMemcpyHostToDevice); // copy HtoD
```

- ▶ **OpenCL** buffer object : a one-dimensional collection of elements
- ▶ elements can be a scalar type, vector type, or user-defined structure

```
float* h_pData = (float*)malloc(sizeof(float)*N); // alloc on host  
// alloc on device & copy HtoD  
cl_mem d_gpubuf = clCreateBuffer(clCtx, CL_MEM_USE_HOST_PTR, sizeof(float)*N, h_pData, &err );
```


Compute index space dimensionality

- ▶ There is a direct parallel to the application data – memory space relation
- ▶ The computational index space is also flattened/linearized when we calculate a gpu kernel (thread) global (linear) index
 - ▶ this is mathematically identical to finding the offset into a flattened/linearized multi-dimensional space

- ▶ Recall

- ▶ the total index space is the grid size : can be 1D, 2D, or 3D

- ▶ 2D built-in

- ▶ `dim3 numblocksInGrid(Nx, Ny, 1);` `size_t global_work_size = { Gx, Gy, Gz } // WGs`
 - ▶ `dim3 numthreadsInBlock(Tx,Ty, 1);` `size_t local_work_size = {Lx, Ly, Lz } // WIs`
 - ▶ `threadIdx.x, threadIdx.y, threadIdx.z` `get_local_id(dimIdx)`
 - ▶ `blockDim.x, blockDim.y` `get_local_size(dimIdx) // num WIs in WG dim`
 - ▶ `blockIdx.x, blockIdx.y` `get_group_id(dimIdx)`
 - ▶ `gridDim.x, .y, .z` `get_num_groups(dimIdx) // num WG/TB in grid`
 - ▶ `warpSize`

- ▶ `cudaThreadIdxFrom1D = blockDim.x * blockIdx.x + threadIdx.x`

EXAMPLE

➤ 1D:

- Num Workgroups/Blocks = 4, WorkGroup/Block size = 4

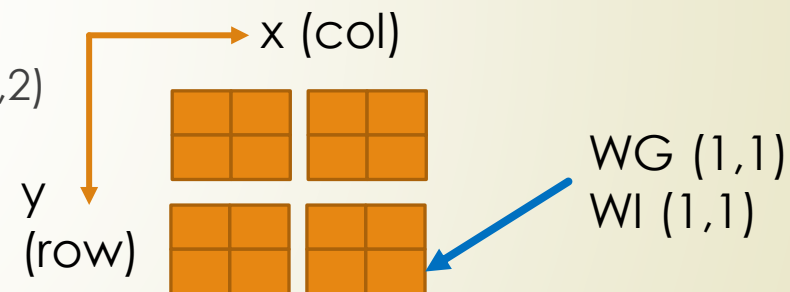


$$\text{globalLinearID} = \text{blockDim.x} * \text{blockIdx.x} + \text{threadIdx.x}$$

WG (2), WI (3)

➤ 2D:

- Num WG/BLK = (2,2), WI/Threads = (2,2)



- $$\begin{aligned} \text{globalLinearID} = & \text{gridDim.x} * \text{blockDim.x} * \text{blockDim.y} * \text{blockIdx.y} \\ & + \text{gridDim.x} * \text{blockDim.x} * \text{threadIdx.y} \\ & + \text{blockDim.x} * \text{blockIdx.x} + \text{threadIdx.x} \end{aligned}$$



Different (but related) spaces

Conceptually, we can think of three distinct but related spaces.
They can each have their own dimensionality.
In practice, they are all related.

1. application-space problem size
2. computational index space
 - ▶ related to how we decompose the problem for parallelization
 - ▶ and also to HW
3. hardware execution capabilities/resources

EXAMPLE

- ▶ 3D MRI volume medical imaging
 - ▶ Application-space: 3D physical coordinates (x,y,z)
 - ▶ device sampling resolution provides 1001x1001x511 volumetric data
 - ▶ Computational index space:
 - ▶ decompose/partition full 3D grid / 3D NDRange into work units
 - ▶ workgroups (threadblocks) / workitems (threads)
 - ▶ N workgroups, M workitems/workgroup
 - ▶ these choices are guided by
 - ▶ the specific parallelization algorithm being implemented
 - ▶ HW resources : device CUs and PEs, device memory (global, shared/local, private)
 - ▶ choices tend to use **powers-of-two**.
 - ▶ We'll pick closest power-of-two: $1024 \times 1024 \times 512 = 128 \times 8 \times 128 \times 8 \times 64 \times 8$
 - ▶ so WORKGROUP/NumThreadBlock : $128 \times 128 \times 64 = 1,048,576 = 1024 \text{ Ki}$
 - ▶ and WorkItem / Threads size: $8 \times 8 \times 8 = 512$

OPENCL

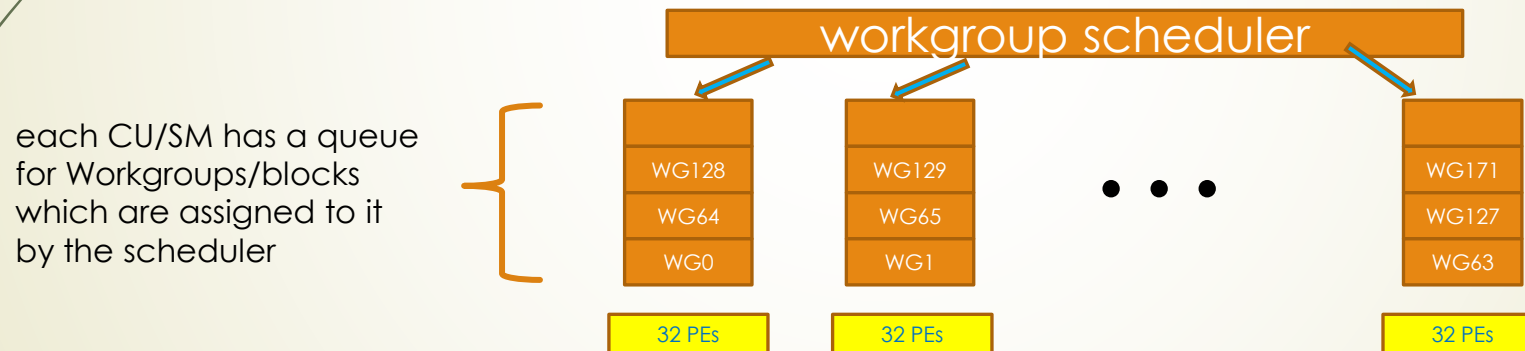
```
size_t global0[] = { 128, 128, 64 };  
size_t local0[] = { 8, 8, 8 };  
err = clEnqueueNDRangeKernel(cmd0, knl0, 3, NULL, global0, local0, 0, NULL, NULL);
```

CUDA

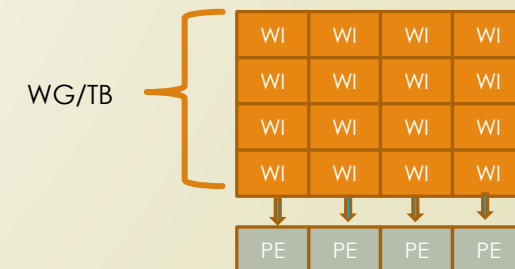
```
dim3 numBlocks{ 128, 128, 64 };  
dim3 threadsPerBlock{ 8, 8, 8 };  
myKnl<<<numBlocks,threadsPerBlock>>>(a,b,c)
```

EXAMPLE (cont...)

- ▶ We started with 1001x1001x511 volumetric MRI data
- ▶ To a compute index space of: $1024 \times 1024 \times 512 = 128 \times 8 \times 128 \times 8 \times 64 \times 8$
 - ▶ Number of WorkGroups (threadblocks) : $128 \times 128 \times 64 = 1,048,576$ (= 1024 Ki)
 - ▶ Number of WorkItems / (threads) size: $8 \times 8 \times 8 = 512$
- ▶ SUPPOSE we have a Hypothetical GPU
 - ▶ **64** CUs (SMs) with **32** PEs (cores) each = 2048 cores total



- ▶ Since a single WG has 512 WIs, but each CU only has 32 PEs,
 - ▶ the WG will be processed in "strips" of 32, in 16 steps.
 - ▶ each strip of 32 WIs is a warp (CUDA term)



Grid/NDRange : 2D Example

Boundary Handling

- In general the size of the compute domain may not be evenly divisible by the block (and warp) dimensions.

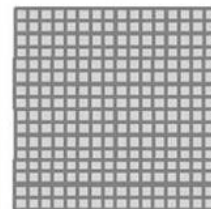
Example

- Suppose we want to process an image of size $(M \times N) = 62 \times 76$
- If we use following

```
dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1);  
dim3 dimBlock(16, 16, 1); // threads per block
```

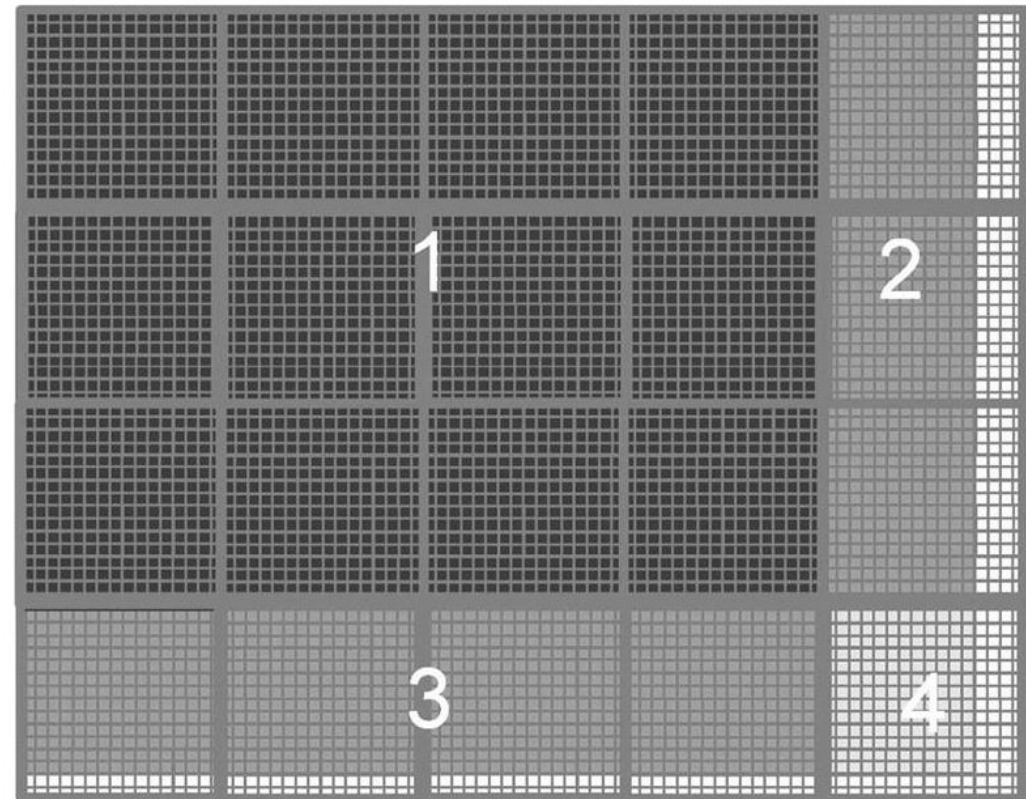
- We'll have a grid size of

- 64x80 threads
- 4x5 blocks



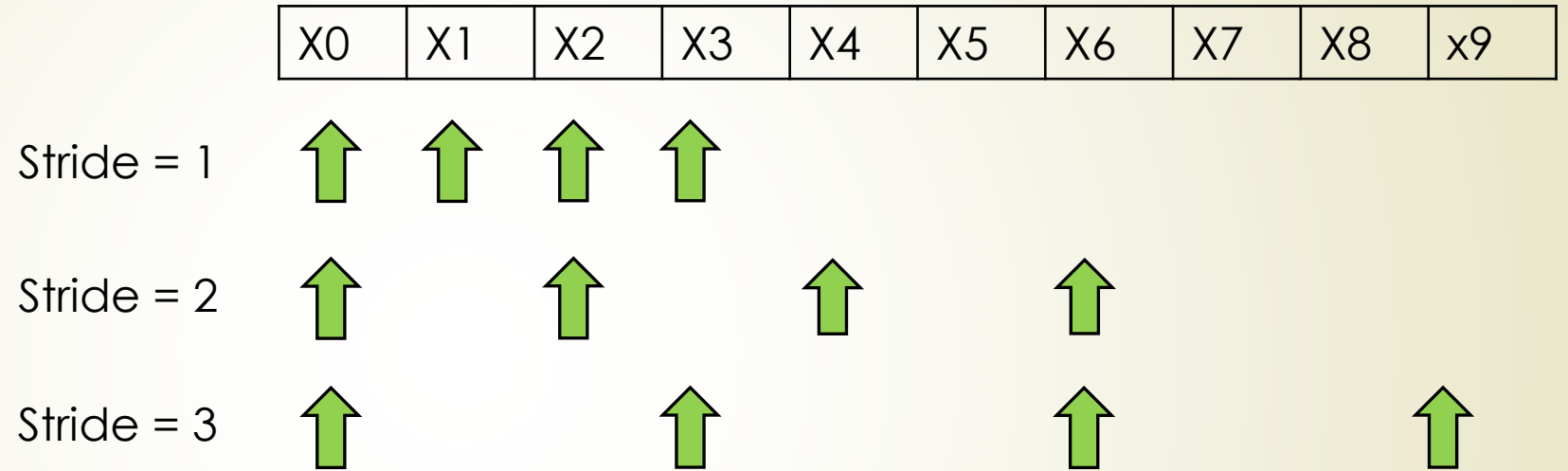
16x16 block

So there will be regions where the kernel will need to explicitly guard against out-of-bound accesses.



Grid/NDRange Additional Considerations

- Strides and striding : 1D data len $N = 10$. gridsize = 1 block, blocksize = 4 threads



- Suppose our total data size N is larger than the total grid size
 - We can have kernels loop where each thread strides by the grid size
 - “Grid-stride loop” formalism with data boundary handling:

```
__global__ void vecAdd_gridstride(float *result, float *a, float *b, int N) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for(int i = index; i < N; i += stride)
    {
        result[i] = a[i] + b[i];
    }
}
```


Vector Addition – Host-side (OCL-focus)

- The host program is the code that runs on the host system to:
 - Setup the environment for the OpenCL/CUDA program
 - Create and manage kernels, and associated arguments and memory objects
- 6 steps in a basic host program:
 1. Define the **platform** ... platform = devices+context+queues
 2. Create and Build the **program** (dynamic library for kernels)
 3. Setup **memory** objects to manage input/output data (host-device)
 4. Define and configure the **kernel** (attach arguments to kernel fns)
 5. Enqueue **commands** ... transfer memory objects and execute kernels
 6. Retrieve results

1. Define the platform

- Grab the first available platform:

```
err = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
```

- Use the first GPU device the platform provides:

```
err = clGetDeviceIDs(firstPlatformId,  
                     CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
```

- Create a simple context with a single device:

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

- Create a simple command-queue to feed our device:

```
commands = clCreateCommandQueue(context, device_id, 0, &err); // < OCL 2.0
```

```
commands = clCreateCommandQueueWithProperties(context, dev_id, qProps, &err);
```

2. Create and Build the program

- Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (for real applications with non-trivial or multiple kernels & aux fns).

- (HW1 read_source() header file)

- Create the program object:

```
program = clCreateProgramWithSource(context, 1  
                                     (const char**) &KernelSource, NULL, &err);
```

- Compile & link the program to create a “dynamic library” from which specific kernels can be pulled:

```
err = clBuildProgram(program, dev_id, 1, NULL, NULL, NULL);
```

Build error messages

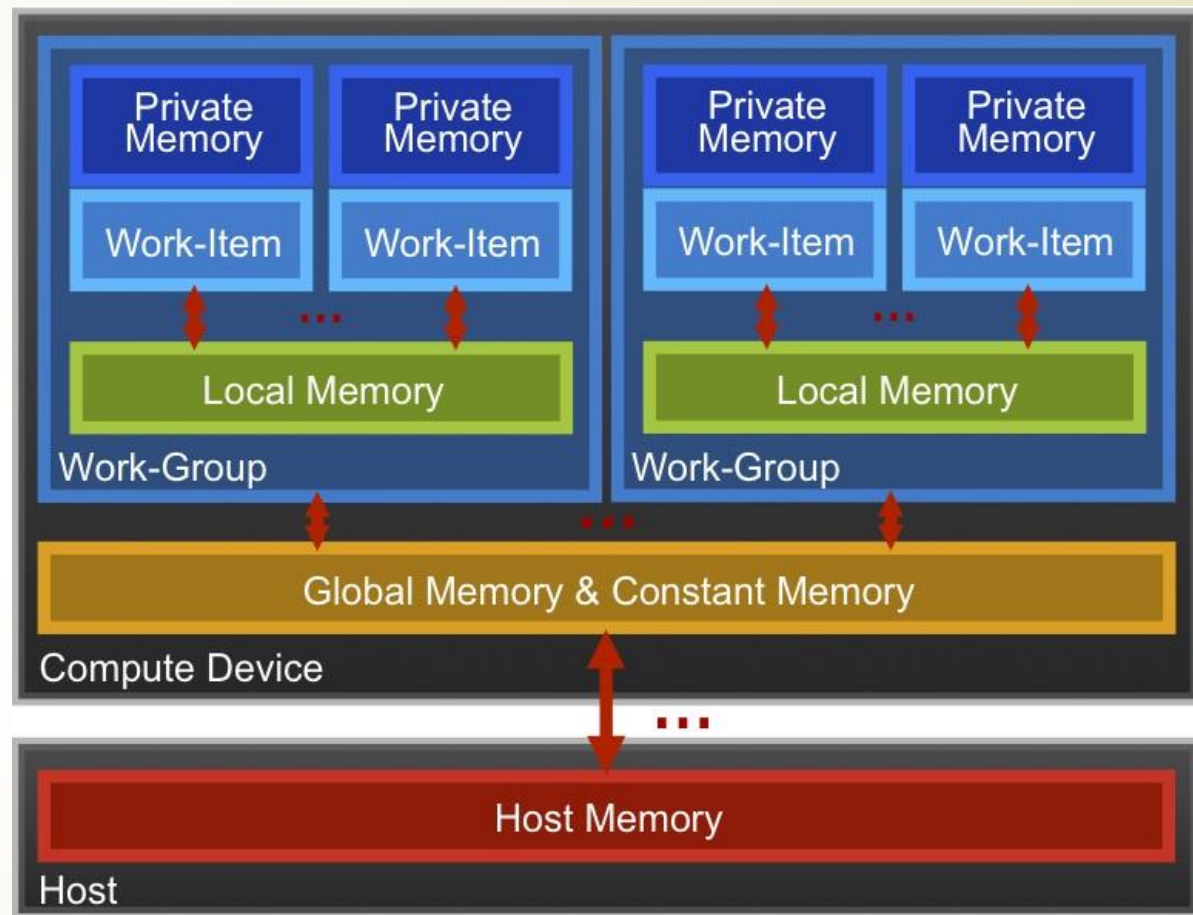
- Fetch and print error messages:

```
if (err != CL_SUCCESS) {  
    size_t len;  
    char buffer[2048];  
    clGetProgramBuildInfo(program, device_id,  
                          CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);  
    printf("%s\n", buffer);  
}
```

- Important to check all your OpenCL API error messages!

OpenCL Memory model

- **Private Memory**
 - Per work-item
- **Local Memory**
 - Shared within work-group
- **Global Memory**
- **Constant Memory**
 - Visible to all work-groups
- **Host memory**
 - On the CPU



Memory management is **explicit**:
Programmer is responsible for moving data from
host → global → local (and back)

And in CUDA...

The memory terminology is a bit of mess between CUDA and OpenCL.
We'll spend a lot of time studying the memory system throughout the course
It's extremely important for performance.

CUDA	OpenCL	Description
Host memory	Host memory	
Global or Device memory	Global memory	DRAM
Local memory	Global memory*	DRAM
Constant memory	Constant memory	
Texture memory	Global memory*	
Shared memory	Local memory	SRAM, on-chip
Registers	Private memory	per thread/work-item

3. Setup Memory Objects

- ▶ For vector addition we need 3 memory objects, one each for input vectors A and B, and one for the output vector C.
- ▶ Create input vectors and assign values on the host:

```
float h_a[LENGTH], h_b[LENGTH], h_c[LENGTH];  
    // or malloc(sizeBytes) or _aligned_malloc(sizeBytes,alignSz)  
for (i = 0; i < length; i++) {  
    h_a[i] = rand() / (float)RAND_MAX;  
    h_b[i] = rand() / (float)RAND_MAX; }
```

- ▶ Define OpenCL device memory objects:

```
cl_mem d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float)*count, NULL, NULL);  
cl_mem d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float)*count, NULL, NULL);  
cl_mem d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float)*count, NULL, NULL);
```


What do we put in device memory?

Memory Objects:

- A handle to a reference-counted region of **global device** memory.

In OpenCL 2.0 there are three kinds of memory objects

- **Buffer** object:
 - Defines a linear collection of bytes ("*just a C array*"). Block of contiguous memory.
 - The contents of buffer objects are fully exposed within kernels and can be accessed using pointers
- **Image** object:
 - Defines a two- or three-dimensional region of memory.
 - Image data can **only** be accessed with read and write functions, i.e. these are opaque data structures. The read functions use a sampler.
- **Pipe** object:
 - Ordered sequence of data items
 - Two-endpoints: write (data items inserted), read (data items removed)
 - Only one kernel instance may write, and one may read. Must be different.

Memory Object Creation

We will focus on the **Buffer** object type (but it's similar for **Image** and **Pipe** objects)

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

Allocate memory on the GPU and possibly copy host data to device

Flags (bit-field):

- CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, CL_MEM_READ_ONLY
- CL_MEM_HOST_WRITE_ONLY, CL_MEM_HOST_READ_ONLY, CL_MEM_HOST_NO_ACCESS
- CL_MEM_USE_HOST_PTR : *Intel zero-copy buffer requires manual alignment
- CL_MEM_ALLOC_HOST_PTR : *runtime will create zero-copy allocation buffer
- CL_MEM_COPY_HOST_PTR

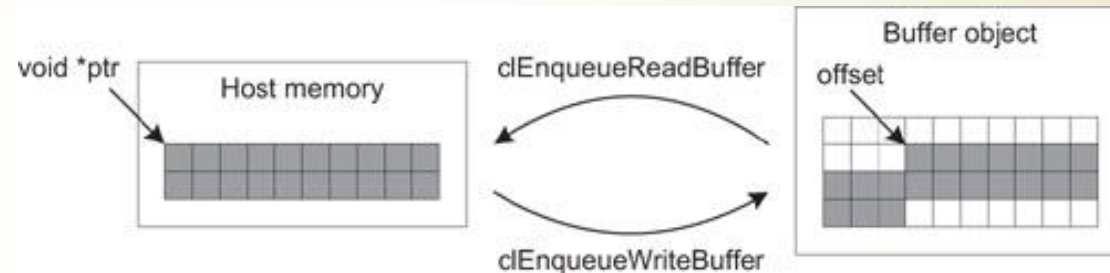
Example:

```
cl_mem_flags flags =  
    CL_MEM_READ_ONLY | CL_MEM_HOST_WRITE_ONLY | CL_MEM_USE_HOST_PTR
```

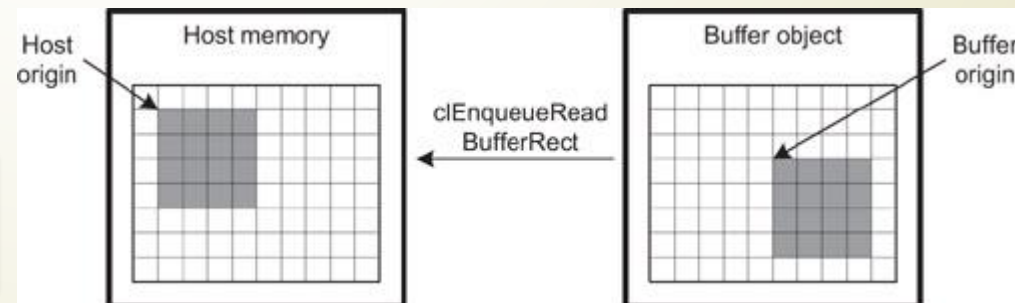
Memory Object Transfer Commands

Read/write data transfers:

```
cl_int clEnqueueRead/WriteBuffer (cl_command_queue command_queue, cl_mem  
buffer, cl_bool blocking_read, size_t offset, size_t size, void *ptr,  
cl_uint num_events_in_wait_list, const cl_event *event_wait_list,  
cl_event *event)
```



```
cl_int clEnqueueRead/WriteBufferRect (cl_command_queue command_queue, cl_mem  
buffer, cl_bool blocking_read, const size_t *buffer_origin, const size_t *host_origin, const  
size_t *region, size_t buffer_row_pitch, size_t buffer_slice_pitch, size_t host_row_pitch, size_t  
host_slice_pitch, void *ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list,  
cl_event *event)
```

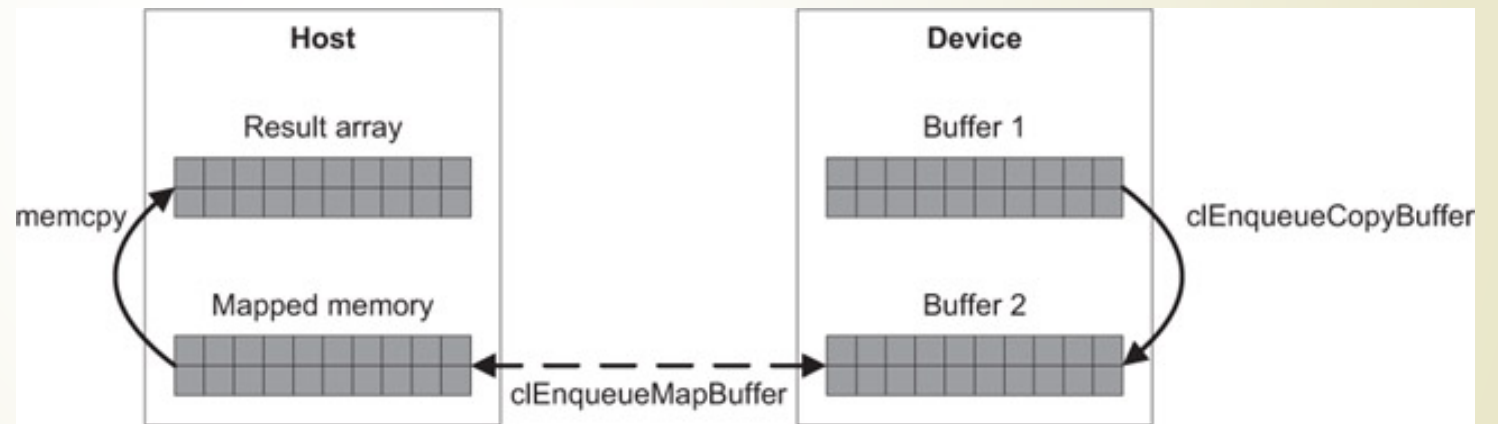


These commands make host
copies of buffers on Intel Skylake

Memory Object Transfer Commands

Mapping memory objects:

```
void * clEnqueueMapBuffer (cl_command_queue command_queue,  
cl_mem buffer, cl_bool blocking_map, cl_map_flags map_flags, size_t offset,  
size_t size, cl_uint num_events_in_wait_list, const cl_event *event_wait_list,  
cl_event *event, cl_int *errcode_ret)
```



```
cl_int clEnqueueUnmapMemObject (cl_command_queue command_queue,  
cl_mem memobj, void *mapped_ptr, cl_uint num_events_in_wait_list, const  
cl_event *event_wait_list, cl_event *event)
```

Creating and manipulating buffers

- ▶ Buffers are declared on the host as type: `cl_mem`
- ▶ Arrays in host memory hold your original host-side data:

```
float h_a[LENGTH], h_b[LENGTH];
```

- ▶ Create the device `buffer` (`d_a`), assign `sizeof(float)*count` bytes from host “`h_a`” to the buffer and copy it into device memory:

```
cl_mem d_a = clCreateBuffer(context,  
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  
    sizeof(float)*count, h_a, NULL);
```


Some conventions for naming buffers

and other variables...

- ▶ Code can often become unclear about whether a host variable is just a regular C array or an OpenCL buffer.
 - ▶ adopt good naming conventions from the start to avoid debug pain!
- ▶ A useful convention is to prefix the names of your regular **h**ost C arrays with “**h_**” and your OpenCL buffers which will live on the **d**evice with “**d_**”
- ▶ Also note that on the host, using OpenCL API functions, datatypes, etc are named using “cl” prefix, e.g. clCreateBuffer, cl_mem
 - ▶ this is not the case in the OpenCL C kernel language

Creating and manipulating buffers

- Other common **memory flags** include:

CL_MEM_WRITE_ONLY, CL_MEM_READ_WRITE

- These are (typically) from the point of view of the **device**

- Submit command to copy the buffer back to host memory at “h_c”:

➤ **CL_TRUE** = blocking, **CL_FALSE** = non-blocking

```
clEnqueueReadBuffer(queue, d_c, CL_TRUE,  
                    sizeof(float)*count, h_c,  
                    NULL, NULL, NULL);
```

4. Define and configure the kernel

- Create **kernel object** from the **kernel function** "vadd":

```
kernel = clCreateKernel(program, "vadd", &err);
```

- Attach kernel function arguments to memory objects:
 - for a kernel defined as

```
__kernel void vadd(__global int* a, __global int* b, __global int* c, uint N)
```

```
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
```

```
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
```

```
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
```

```
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
```

5. Enqueue commands

- Write **Buffers** from host into **global GPU (device)** memory (as **non-blocking** operations):

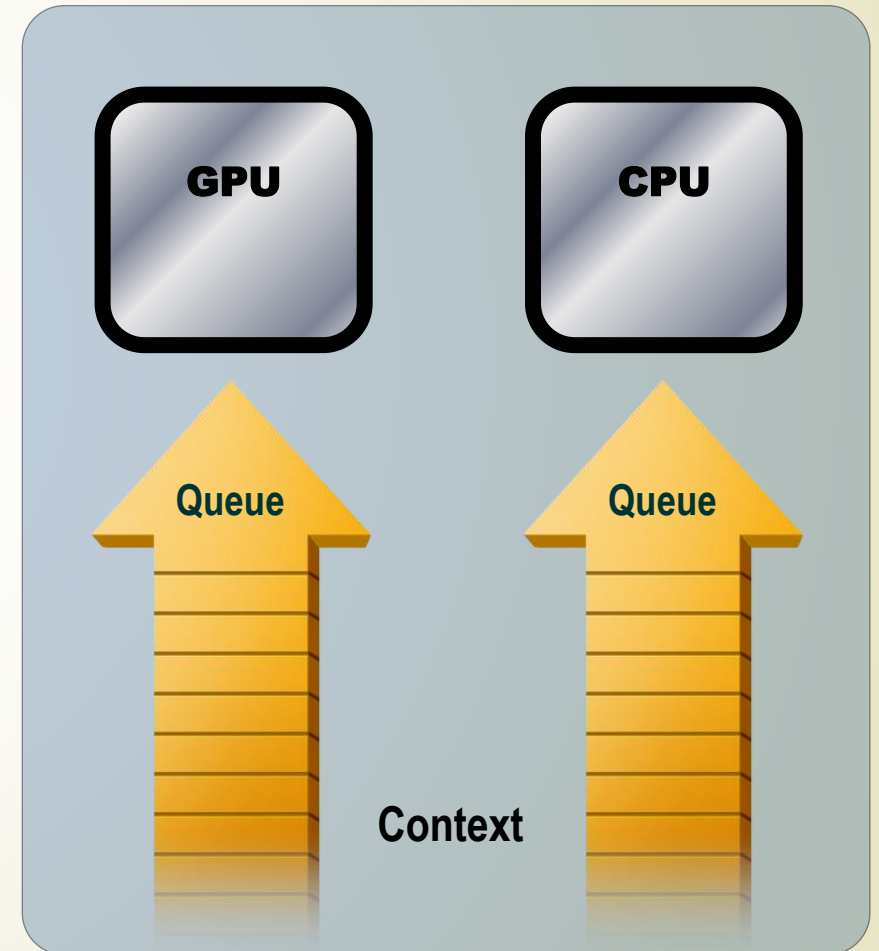
```
err = clEnqueueWriteBuffer(commands, d_a, CL_FALSE,  
                           0, sizeof(float)*count, h_a, 0, NULL, NULL);  
err = clEnqueueWriteBuffer(commands, d_b, CL_FALSE,  
                           0, sizeof(float)*count, h_b, 0, NULL, NULL);
```

- Enqueue the kernel for execution (note: in-order so OK):

```
err = clEnqueueNDRangeKernel(commands, kernel, 1,  
                             NULL, &global, &local, 0, NULL, NULL);
```

Command-Queues and Commands

- ▶ Commands include:
 - ▶ Kernel executions
 - ▶ Memory object management
 - ▶ Synchronization
- ▶ The only way to submit **commands** to a device is through a command-queue.
- ▶ Each command-queue points to a **single** device within a context.
- ▶ Multiple command-queues can feed a single device.
 - ▶ Used to define independent streams of commands that don't require synchronization



Command-Queue execution

Command queues can be configured in different ways to control how commands execute

- ▶ **In-order queues:**

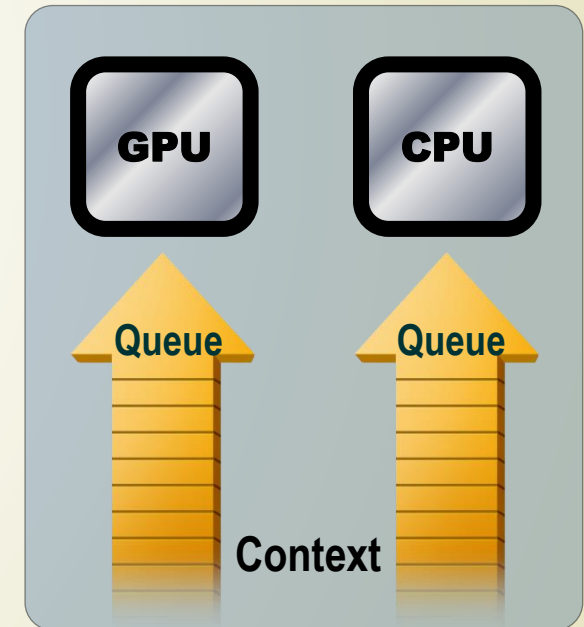
- ▶ Commands are enqueued and complete in the order they appear in the program (program-order)

- ▶ **Out-of-order queues:**

- ▶ Commands are enqueued in program-order but can execute (and hence complete) in any order.

- ▶ Execution of commands in the command-queue are guaranteed to be completed at synchronization points

- ▶ *details will be covered later*



clEnqueueNDRangeKernel(...)

```
cl_int clEnqueueNDRangeKernel(cl_command_queue queue, cl_kernel kernel,  
cl_uint work_dims, const size_t *global_work_offset, const size_t *global_work_size,  
const size_t *local_work_size, cl_uint num_events, const cl_event *wait_list,  
cl_event *thisEvent)
```

Index Space Arguments:

`work_dims`— number of dimensions in the data
`global_work_offset`— global ID offsets in each dimension
`global_work_size`— number of work-items in each dimension
`local_work_size`— number of work-items in a work-group, in each dimension

Kernel Object Information Queries

- `clGetKernelInfo()`
- `clGetKernelWorkGroupInfo()` – useful for dynamic performance tuning
- `clGetKernelArgInfo()` – requires program built with `-cl-kernel-arg-info` option

5. Enqueue commands (continued)

- ▶ Read back result (as a blocking operation). We have an in-order queue which assures the previous commands are completed before the read can begin.

```
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE,  
                           sizeof(float)*count, h_c, 0, NULL, NULL);
```

And in CUDA...

- ▶ Example of CUDA Driver API
 - ▶ Host application
 - ▶ demonstrating same basic host application steps as discussed for OpenCL API (see slide #14)
- ▶ Comments
 - ▶ cuModuleLoad takes PTX or CUBIN
 - ▶ must precompile .cu with nvcc
 - ▶ cuModuleGetFunction
 - ▶ must use the *mangled* name from PTX
 - ▶ cuLaunchKernel
 - ▶ uses the simpler form of kernel args
 - ▶ there is also an advanced form

```
int main(int argc, char** argv)
{
    // 0. always check and handle errors (IMPLIED step...)
    // 1. Define the platform (= obtain the CUDA device and context)
    // Initialize
    cudaChk(cuInit(0));

    // Get number of devices supporting CUDA
    int deviceCount = 0;
    cudaChk(cuDeviceGetCount(&deviceCount));
    if (deviceCount == 0) {
        printf("There is no device supporting CUDA.\n");
        exit(0);
    }

    // Get handle for device 0
    CUdevice cuDevice;
    cudaChk(cuDeviceGet(&cuDevice, 0));

    // Create context
    CUcontext cuContext;
    cudaChk(cuCtxCreate(&cuContext, 0, cuDevice));

    // 2. Create and build the Module and Function
    // Create module from binary file
    CUmodule cuModule;
    cudaChk(cuModuleLoad(&cuModule, "vecAdd_01.ptx")); // precompiled PTX or CUBIN from nvcc

    // Get function handle from module
    CUfunction vecAdd;
    cudaChk(cuModuleGetFunction(&vecAdd, cuModule, "_Z9vecAdd_01PfS_i")); // NOTE: mangled name

    // 3. Setup memory objects to manage the input-output host and device data
    // Allocate input vectors h_A etc. in host memory
    float* h_A = (float*)malloc(size);

    // init input vectors...
    // Allocate vectors in device memory
    CUdeviceptr d_A;
    cudaChk(cuMemAlloc(&d_A, size));

    // Copy vectors from host memory to device memory
    cudaChk(cuMemcpyHtoD(d_A, h_A, size));

    // 4. Configure the kernel for execution - set up arguments, grid/index hierarchy
    //// Invoke kernel
    int threadsPerBlock = 4;
    int blocksPerGrid = 4;
    // setup kernel arguments (using the simple kernel argument format)
    void* args[] = { &d_C, &d_A, &d_B, &N };

    // 5. Launch the kernel
    cudaChk(cuLaunchKernel(vecAdd, blocksPerGrid, 1, 1, threadsPerBlock, 1, 1, 0, 0, args, 0));
    cudaChk(cuCtxSynchronize());

    // 6. Retrieve results from device & verify/use
    cudaChk(cuMemcpyDtoH(h_C, d_C, size));

    // Do data result verification routine...

    return 0;
}
```

Vector Addition – Complete Host Program

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

Define platform and queues

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

cl_device_id[] 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_WRITE,
    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);
```

Define memory objects

Create the program

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

Build the program

```
// 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;
```

Create and setup kernel

```
// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
    global_work_size, NULL, 0, NULL, NULL);
```

Execute the kernel


```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
    CL_TRUE, 0,
    n*sizeof(cl_float),
    0, NULL, NULL);
```

Read results on the host

- It seems a little complicated the first time through, but most of this is “boilerplate”
- we will create templates to provide standard re-usable implementations.



Review: Basic OpenCL Program Flow

1. Allocate host-side memory for memory objects and initialize (**CUDA**)
 2. Get platform information
 3. Get device list from platform and select type of device(s) to run on (**CUDA**)
 4. Create OpenCL context for the device (**CUDA**)
 5. Create command queue(s) (**CUDA***)
 6. Create memory objects on the device (**CUDA**)
 7. Create program object from kernel source (**CUDA**)
 8. Build program and create OpenCL kernel object (**CUDA**)
 9. Set arguments of the kernel (**CUDA**)
 10. Enqueue kernel to execute on the device for index space (NDRange) (**CUDA**)
 11. Host retrieves results from device memory (**CUDA**)
 12. Clean up and wait for all commands to complete (receive events) (**CUDA**)
 13. Release all OpenCL allocated objects and free memory (**CUDA**)
- 

Kernel Languages 101 - Quick Ref

OpenCL C	CUDA C++
https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_C.pdf	https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-cplusplus-language-support
Built-in data types <ul style="list-style-type: none"> • scalar <ul style="list-style-type: none"> ◦ bool ◦ char, uchar (8-bit) ◦ short, ushort (16-bit) ◦ int, uint (32-bit integer) ◦ long, ulong (32-bit integer) ◦ float (32-bit FP) ◦ double (4-bit FP) ◦ half (16-bit FP) ◦ size_t (unsigned integer) • vector (n in {2,3,4,8,16}) <ul style="list-style-type: none"> ◦ charn, ucharn ◦ shortn, ushortn ◦ intn, uintn ◦ longn, ulongn ◦ floatn, doublen 	Built-in data types <ul style="list-style-type: none"> • vector (n in {1,2,3,4}) <ul style="list-style-type: none"> ◦ charn, ucharn ◦ shortn, ushortn ◦ intn, uintn ◦ longn, ulongn ◦ floatn, doublen • dim3 <ul style="list-style-type: none"> ◦ integer vector type ◦ used to specify dimensions Built-in variables <ul style="list-style-type: none"> • dim3 gridDim • uint3 blockDim • dim3 blockDim • uint3 threadIdx • int warpSize (warp size in threads)
qualifiers <ul style="list-style-type: none"> • address space <ul style="list-style-type: none"> ◦ __global ◦ __constant ◦ __local ◦ __private • access <ul style="list-style-type: none"> ◦ __read_only ◦ __write_only ◦ __read_write • function <ul style="list-style-type: none"> ◦ __kernel 	function execution space specifiers <ul style="list-style-type: none"> • __global__ • __device__ • __host__ variable memory space qualifiers <ul style="list-style-type: none"> • __device__ • __shared__ • __constant__
conversions <ul style="list-style-type: none"> • implicit conversions • explicit casts • explicit conversions 	
Built-in functions <ul style="list-style-type: none"> • Work-Item (<i>subset of...</i>) <ul style="list-style-type: none"> ◦ get_work_dim() ◦ get_global_size(dimIdx) ◦ get_global_id(dimIdx) ◦ get_local_size(dimIdx) ◦ get_local_id(dimIdx) ◦ get_num_groups(dimIdx) ◦ get_group_id(dimIdx) ◦ get_global_linear_id() • Math <ul style="list-style-type: none"> ◦ acos, cos, asin, sin, erf, exp, ... ◦ (integer, FP, vector versions) • Synchronization 	Built-in functions <ul style="list-style-type: none"> • Math • Synchronization
operators <ul style="list-style-type: none"> • arithmetic • pre-/post-increment (++ , --) • relational • equality • bitwise • logical 	

Just the essentials!!

For all the details, and when in doubt, consult the official refs (links provided)

Kernel Languages 101

Just the essentials – using vector data types

OpenCL C

- Some **operators overloaded** for vector types
- In general, the vector operations will act in component-wise fashion

<pre>float a[4], b[4], c[4]; for(int i=0; i<4; i++) { c[i] = a[i] + b[i]; }</pre>	➔	<pre>float4 a, b, c; c = a + b;</pre>
<pre>float4 v, u, w; w = v + u;</pre>	equivalent to ➔	<pre>w.x = v.x + u.x; w.y = v.y + u.y; w.z = v.z + u.z; w.w = v.w + u.w;</pre>

CUDA C++

```
// auto-determine number of thread blocks needed based on specified blocksize and data size (Nx, Ny, Nz)  
dim3 threads_per_block(8,8,1);  
dim3 number_of_blocks( (Nx+threads_per_block.x -1)/threads_per_block.x, (Ny+threads_per_block.y -1)/threads_per_block.y, 1);  
  
float4 f4 = make_float4(1,2,3,4);  
float f0 = f4.x; // (x,y,z,w) field components  
float f3 = f4.w;
```

- derived from the basic integer and floating-point types.
- They are structures and the 1st, 2nd, 3rd, and 4th components are accessible through the fields x, y, z, and w

Result verification: CPU v GPU

- ▶ Important part of analyzing your parallel computation implementation is checking you get the right answer, also called **verification**.
- ▶ When the same calculation is done on both CPU and GPU, the results may not be identical. Why?
 - ▶ different ordering of operations can affect results due to rounding, truncation/overflow
 - ▶ floating-point is not associative! $(A+B)+C$ is not necessarily equal to $A+(B+C)$
 - ▶ numerical accuracy and precision of different hardware
 - ▶ particularly evident for floating-point calculations
 - ▶ IEEE 754 compliance and specific options (rounding modes, etc)
- ▶ **THUS**
 - ▶ common to use a small error tolerance value for acceptable agreement threshold

```
float fTol = 1e-7;  
if( abs( cpuResult - gpuResult) <= fTol)  
    correct = TRUE;
```

Ex2 & Hw1 (due date extended)

DUE: 27 JAN (WED), 11:59 PM

See class web: Files/Exercises & Homeworks/Win21_EX2_HW1_FINAL.PDF

There will be a Lec 3 – Part 2 to cover Ex2/Hw1 in more depth

- hopefully posted tomorrow