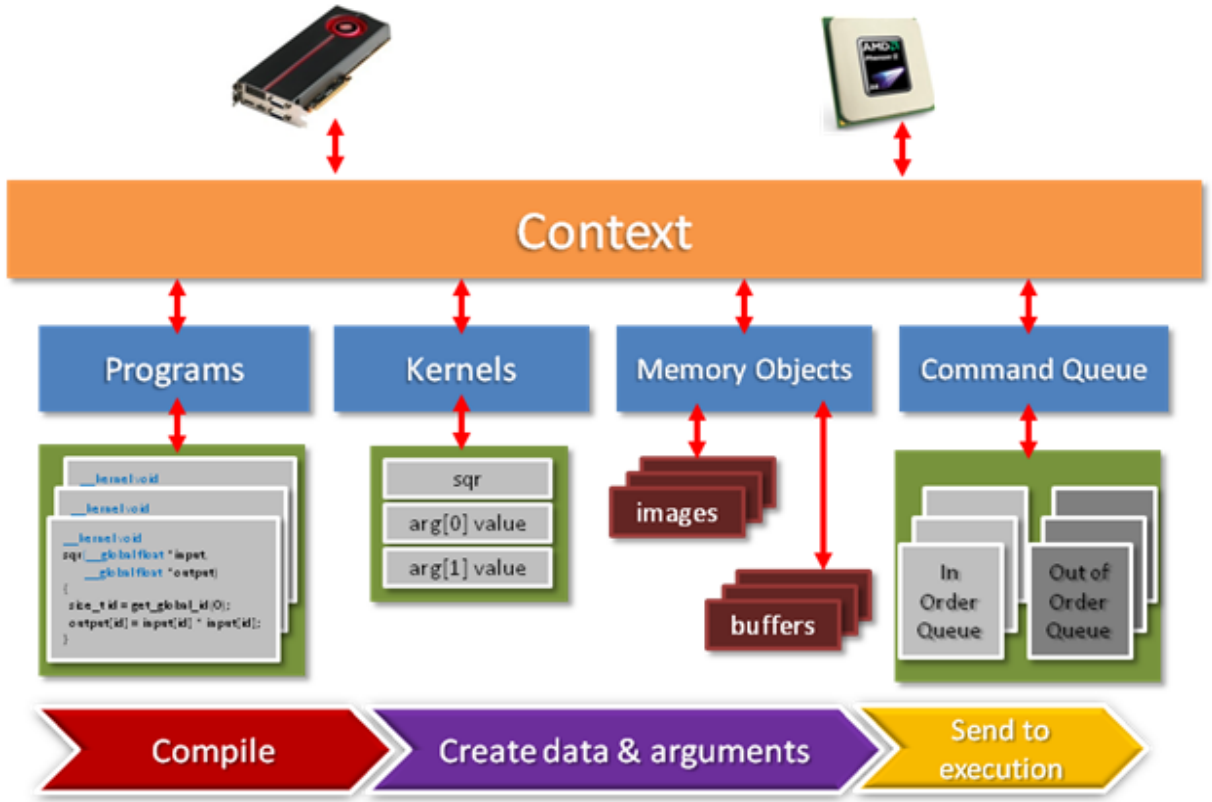
CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

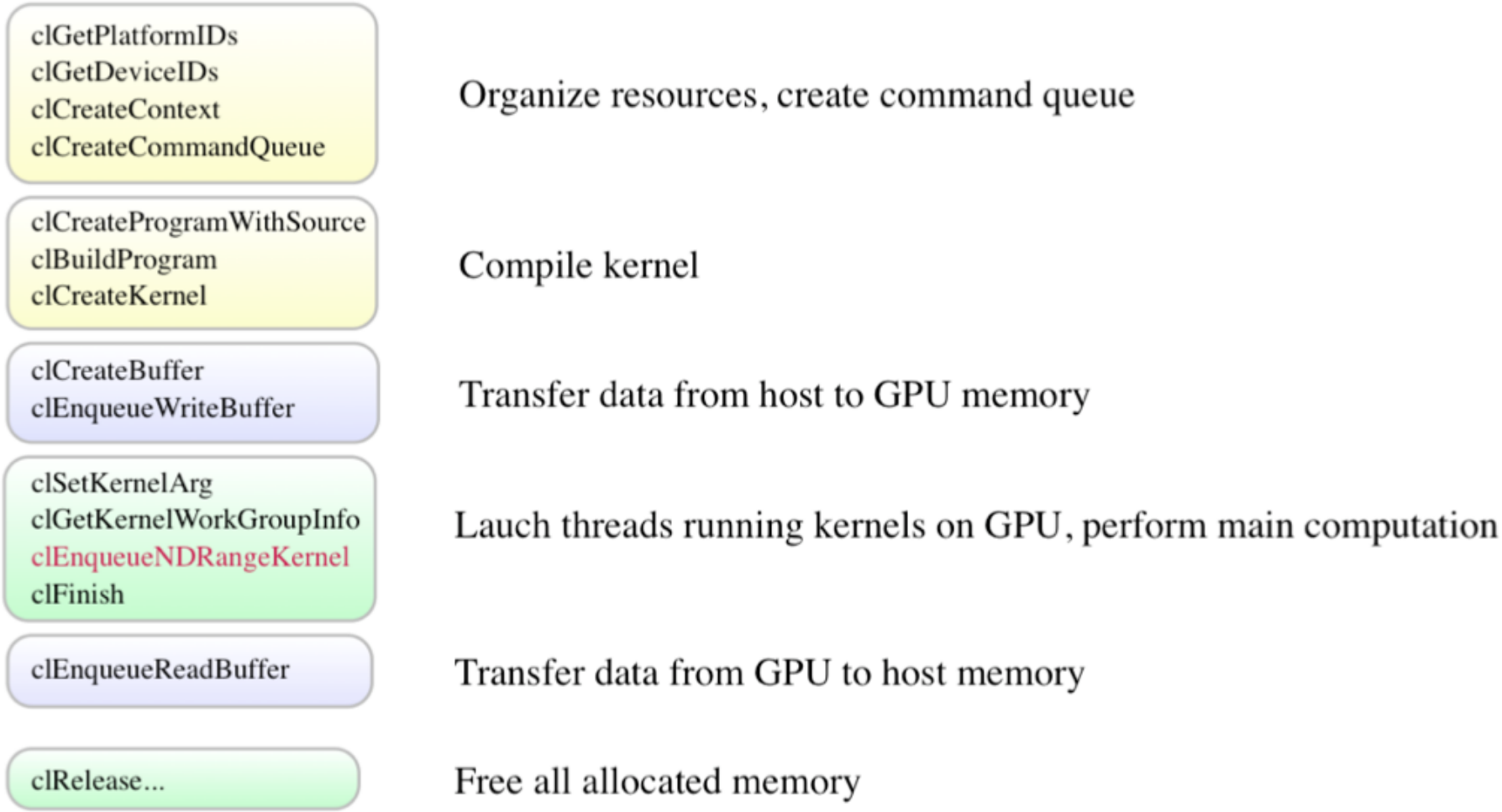
Lecture 15: (OpenCL) Kernels, Memories, Synchronization and Events

OpenCL Objects

- Setup
  - Devices – GPU, CPU, DSP
  - Contexts – Collection of devices
  - Queues – Submit work to the device
- Memory
  - Buffers – Blocks of memory
  - Images – 2D or 3D formatted images
- Execution
  - Programs – Collections of kernels
  - Kernels – Argument/execution instances
- Synchronization/profiling
  - Events



OpenCL Program Flow



Kernels

Working with Kernels

- The kernels are where all the action is in an OpenCL program
- Steps to using kernels:
  1. Load kernel source code into a program object from a file
  2. Make a kernel functor from a function within the program
  3. Initialize device memory
  4. Call the kernel functor, specifying memory objects and global/local sizes
  5. Read results back from the device
- Note the kernel function argument list must match the kernel definition on the host

Expressing Parallelism

- Define our problem domain as an N-Dimensional domain (N=1, 2 or 3), e.g. :
  - 1D: Add two vectors/arrays
  - 2D: Process an image (e.g., 2K x 2K pixels = 4M kernel executions)
  - 3D: Process volumetric data acquired from an x-ray scanner
- Execute kernel at each point in the problem domain
  - Sometimes called a work-grid or index space
  - No creating threads within kernel – write your code for a single thread
- Choose dimension to match your algorithm

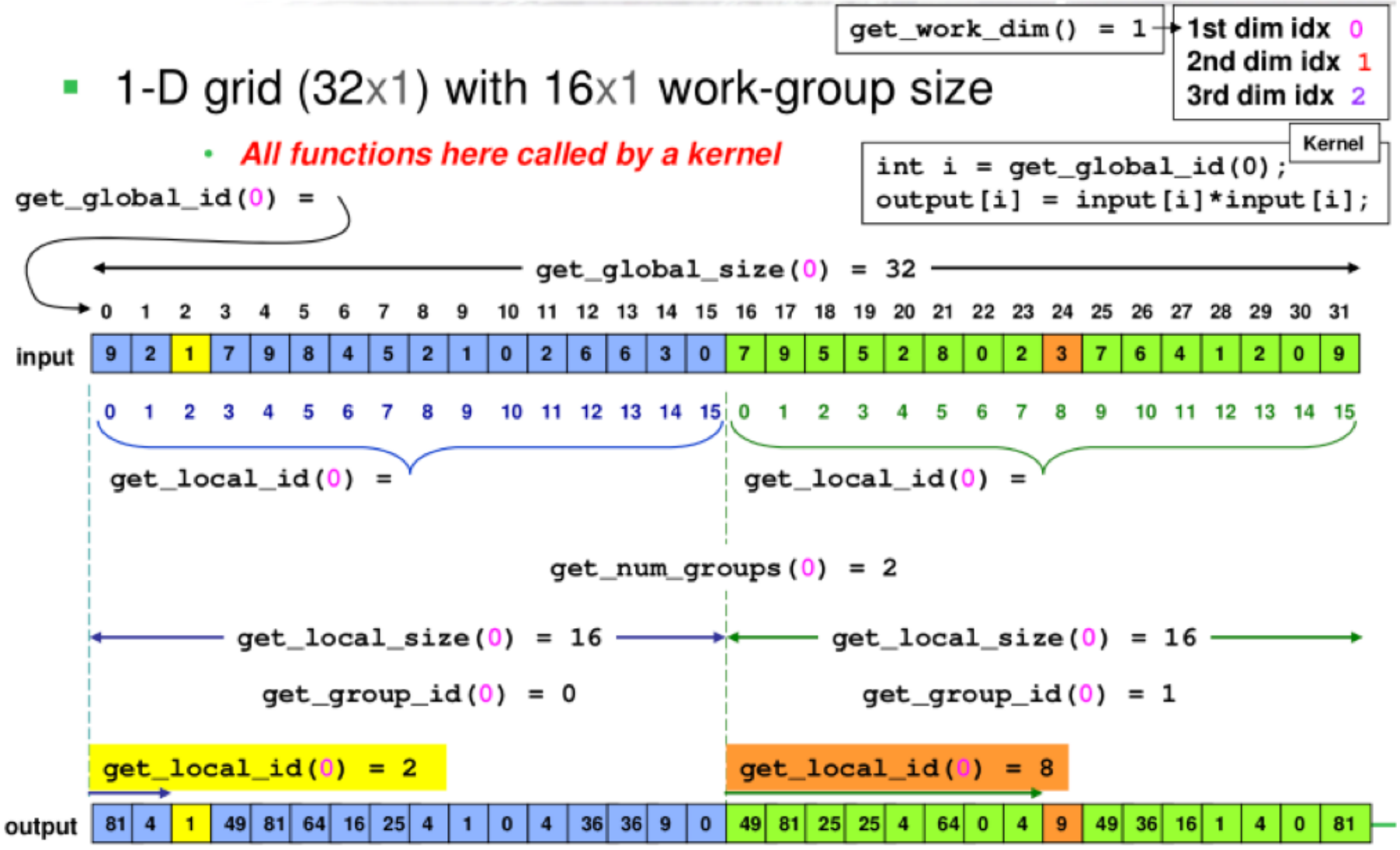
Understanding Kernels, Work-items and Work-groups

- A kernel (function) executes on an OpenCL device
  - Written in OpenCL C
  - Many instances of a kernel execute in parallel but on different data
- A work-item is an instance of the kernel
  - Executed by a Processing Element (PE) in the device
  - Given a unique global id to distinguish it from other work-items
- A work-group is a collection of work-items
  - Its work-items execute concurrently on the same Compute Unit
  - Grouping affects efficiency, sharing local memory, barriers
  - Allows scheduling of device resources, providing scalability
  - Given a unique group id
  - Each work-item within a work-group given a unique local id

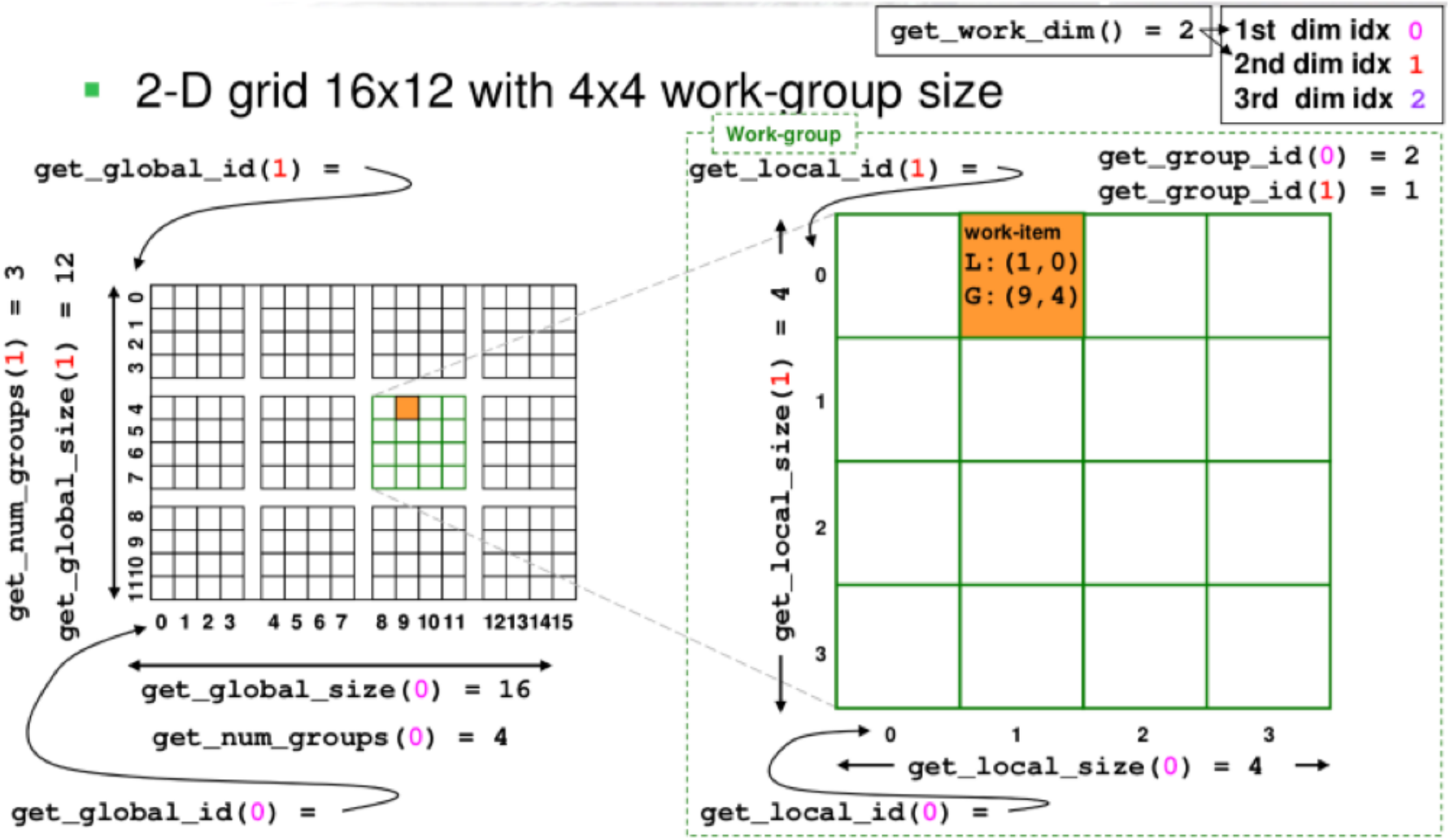
Work-item/group Built-In Functions

uint get_work_dim()	The number of dimensions
size_t get_global_id(uint dimidx)	The ID of the current work-item [0,WI) in dimension dimidx
size_t get_global_size(uint dimidx)	The total number of work-items (WI) in dimension dimidx
size_t get_global_offset(uint dimidx)	The offset as specified in the enqueueNDRangeKernel API in dimension dimidx
size_t get_group_id(uint dimidx)	The ID of the current work-group [0, WG) in dimension dimidx
size_t get_local_id(uint dimidx)	The ID of the work-item within the work-group [0, WI/WG) in dimension dimidx
size_t get_local_size (uint dimidx)	The number of work-items per work-group = WI/WG in dimension dimidx
size_t get_num_groups(uint dimidx)	The total number of work-groups (WG) in dimension dimidx

Work-item/group Built-In Functions (1-D Example)

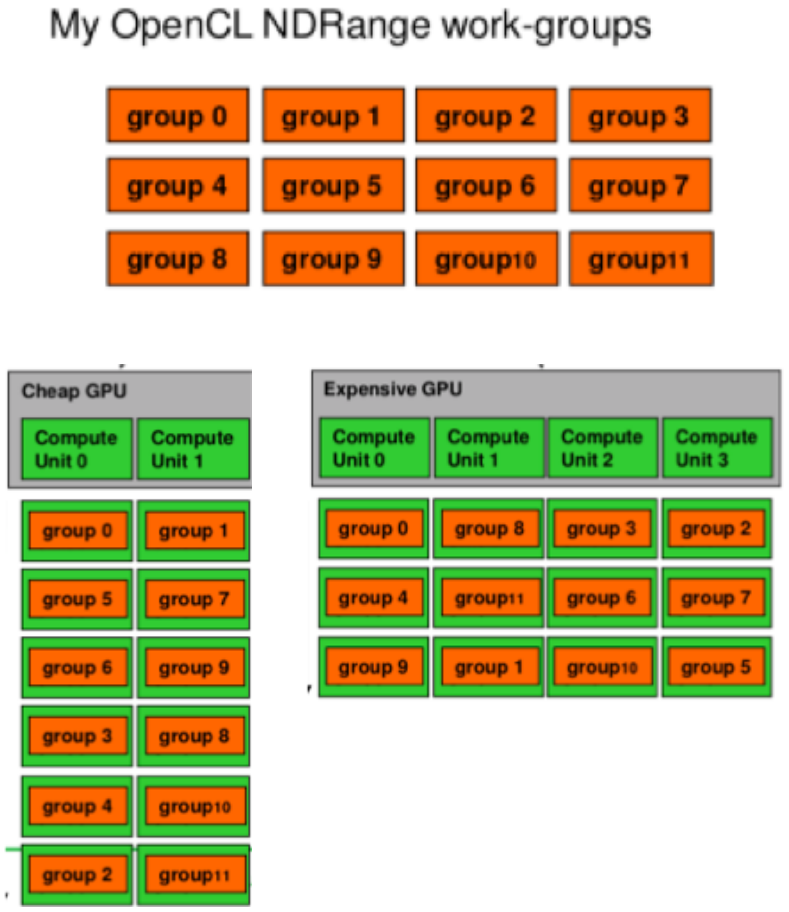


Work-Item/Group Built-In Functions (2-D Example)

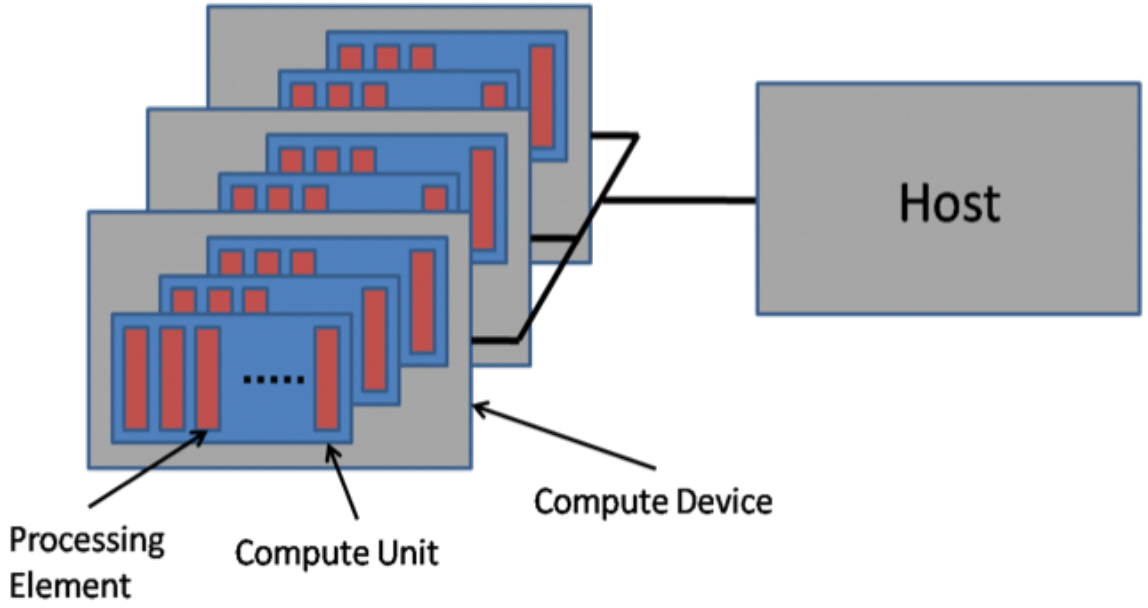


Work-groups Concurrency

- N work-groups execute concurrently
  - Run in any order
  - More Compute Units = more work-groups running in parallel
  - Provides scalability (automatic speed-up, linear in theory)
- All work-groups have same number of work-items
  - Work-items within a work-group execute concurrently
  - Hardware specifies maximum number of work-items allowed
  - Cannot change number of work-items once kernel is running



Work-items Scheduling



- Compute Unit is a SIMD/vector unit
  - Work-items in a work-group grouped into
    - Warps (Nvidia: 32)
    - Wavefronts (AMD: 64)
  - A warp contains consecutive work-item IDs
    - Warps: Work-items 0-31, 32-63, ...
  - Same instruction from your kernel executed at a time in SIMD unit
    - Each work-item executed by one element of the vector unit
    - Lock-step execution
    - Entire warp processed at once by a vector unit
  - If work-group contains more work-items (often does!)
    - Warps of work-items swapped in and out on the vector unit
      - E.g.: when a warp stalls waiting for memory transfer
    - Very little cost associated with swapping (hardware to do it)

Work-groups Numbers

- You specify total number of work-items (global dimensions)
- And optionally number of work-items per work-group (i.e., local dimensions)
  - Local_work_size arg can be NULL
  - OpenCL will choose the ”best” value for the number of work-groups
  - Often recommended to be multiple of warp or wavefront size
    - NVIDIA suggests multiple warps per work-group
  - The local-size must divide evenly the global size in each dimension!

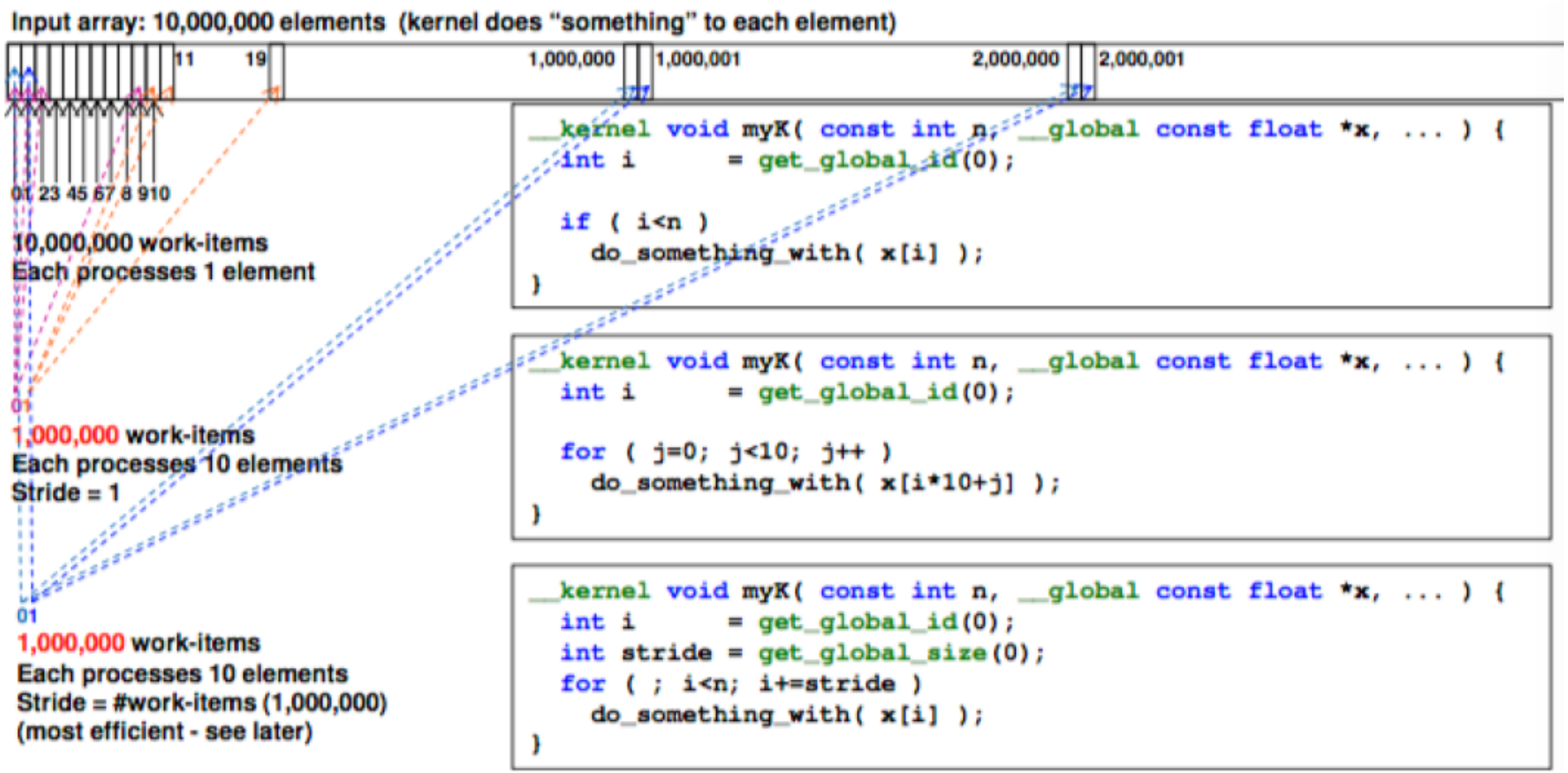
More Work-Items than Needed

- What if the global/local numbers don't divide?
  - Silly example: your array length is prime
  - More realistic: global_work_size[0] = 100,000, local_work_size[0] = 64
  - *Asking OpenCL for 1562.5 work-groups! (Error!)*
- Specify more work-items than required
  - **int num_work_groups = (global_work_size[0] + local_work_size[0] - 1) / local_work_size[0];**
  - **global_work_size[0] = num_work_groups * local_work_size[0];**
  - Set global_work_size[0] = 100032 (gives 1563 work-groups)
  - Kernel must check for going out of bounds in your arrays

```
__kernel void arrayAddOCL( const int n, __global const float *x,
                           __global const float *y, __global float *z )
{
  int i = get_global_id(0);          // 100032 work-items hence i could = 100000+
  if ( i < n )                       // Pass in n=100000 i.e., the true array len
    z[i] = x[i] + y[i];
}
```

Fewer Work-items than Tasks

- Work-item processes more than one element
- Decouples index-space from data



OpenCL Kernels

- Remember: One instance of the kernel created for each work-item

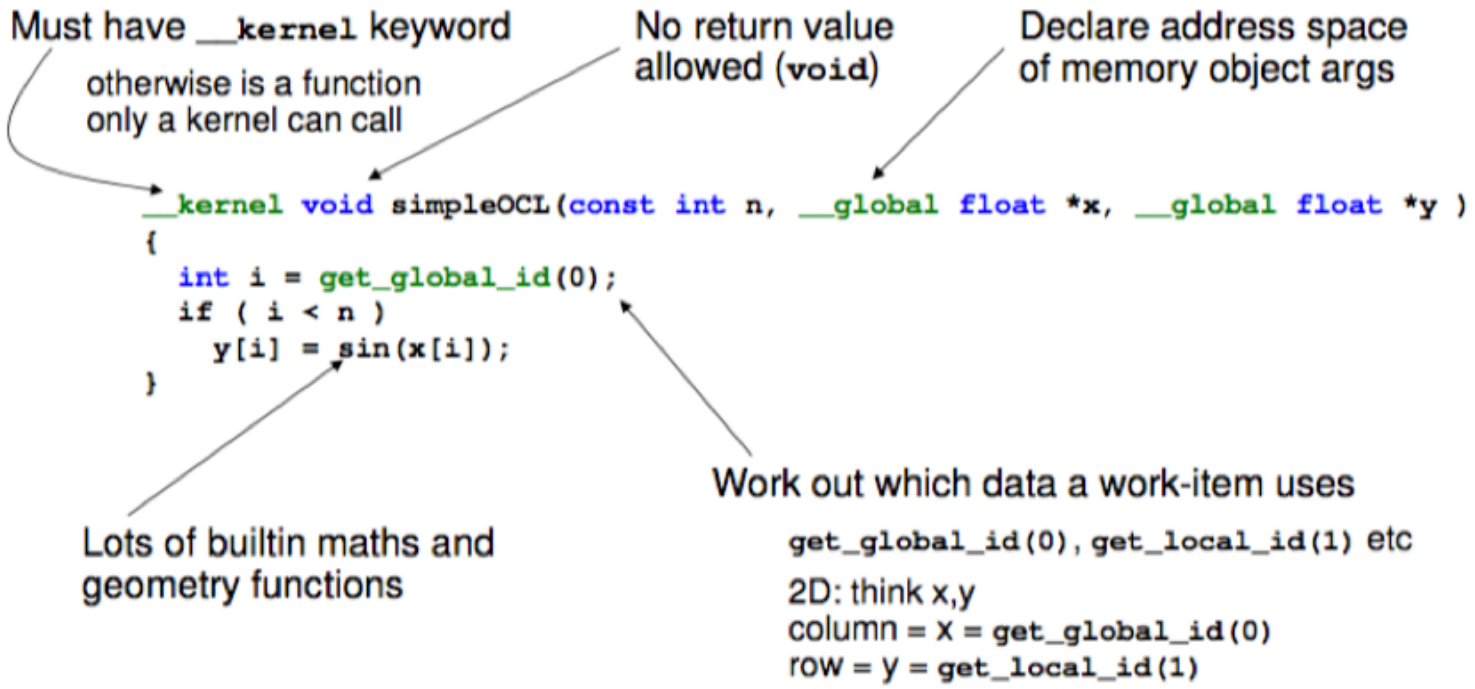


Image Qualifiers

__kernel void myfunc(__read_only image2d_t inputImage, __write_only image2d_t outputImage

OpenCL C for Compute Kernels

- Restricted version of ISO C99
  - No recursion, function pointers, variable length arrays or functions from the C99 standard headers
- Preprocessing directives defined by C99 are supported
  - #include, #ifdef
- Built-in data types
  - Scalar and vector data types, pointers
    - E.g., float4, int16 (with host equivalents: cl_float4, cl_int16)
  - Image types
    - image2d_t, image3d_t, and sampler_t
- Built-in functions
  - Mandatory functions: Work-item/group functions, math.h, read and write images, relational, geometric functions, synchronization functions, printf
  - Optional functions (extensions) up to vendors: e.g., double precision, atomics to global and local memory, selection of rounding mode, write to image3d_t surface

Enable Double Precision in Kernels

- Some features of OpenCL are optional
  - Not all vendors support them (yet)
  - May be promoted to core features in subsequent specs
  - Many exist. Check the string returned by
    - clGetDeviceInfo(..., CL_DEVICE_EXTENSIONS, ...);
- To control the OpenCL compiler's use of extensions use #pragma

```
#ifdef cl_khr_fp64
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#else
#error Kernel requires double precision
// Or could #define REAL to be float or double accordingly
// Can also test for cl_amd_fp64 on AMD h/w
#endif
__kernel void DoubleKernel(__global const double *vecA, ...)
{
  double kFactor;
  int i = get_local_id(0);
  kFactor = sqrt( sin(vecA[i]) + … );
```

Vector Data Types and Notation in Kernels

- Portable version of scalar data types extended to vectors
  - Various types: char, uchar, short, ushort, int, uint, long, ulong, float, (double)
  - Various lengths: 2, 4, 8, 16

OpenCL C Type	OpenCL API Type for host app	Description
charn	cl_charn	signed two's complement 8-bit integer vector
ucharn	cl_ucharn	unsigned 8-bit integer vector
shortn	cl_shortn	signed two's complement 16-bit integer vector
ushortn	cl_ushortn	unsigned 16-bit integer vector
intn	cl_intn	signed two's complement 32-bit integer vector
uintn	cl_uintn	unsigned 32-bit integer vector
longn	cl_longn	signed two's complement 64-bit integer vector
ulongn	cl_ulongn	unsigned 64-bit integer vector
floatn	cl_floatn	floating point vector

- Vector literals
  - float4 fvec = (float 4) (1.0f, 2.0f, 3.0f, 4.0f);
  - uint4 uvec = (uvec4) (1); // uvec = (1, 1, 1, 1)
  - float4 fvec = (float4) ((float2)(1.0f, 2.0f), (float2) (3.0f, 4.0f));

Vector Component Access in Kernels

- Components: if 2, use .xy; if 4, access using .xyzw
  - float4 fvec = (float4) (1.0f, 2.0f, 3.0f, 4.0f);
  - fvec.z = 1.0f; // fvec = (1.0f, 2.0f, 1.0f, 4.0f);
  - fvec.xy = (float2) (5.0f, 8.0f); // fvec = (5.0, 8.0, 1.0, 4.0)
  - fvec.wx = (float2) (7.0f, 8.0f); // fvec = (8.0, 8.0, 1.0, 7.0)
  - float4 swiz = fvec.wzxy; // swiz = (7.0, 1.0, 8.0, 8.0);
  - float4 gvec = fvec.xxyy; // gvec = (8.0, 8.0, 8.0, 8.0);
- Components: numeric index *sn* where n = 0, 1, …, vecsize-1
  - float4 avec, fvec = (float4) (1.0f, 2.0f, 3.0f, 4.0f);
  - fvec.s0 = 2.0f; // fvec = (2.0, 2.0, 3.0, 7.0)
  - fvec.s23 = (float2) (8.0f, 9.0f); // fvec = (2.0, 2.0, 8.0, 9.0)
  - avec.xyzw = fvec.s0123; // fvec = (2.0, 2.0, 8.0, 9.0)
- Components: quick addressing to a group of components
  - float4 f = (float4) (1.0f, 2.0f, 3.0f, 4.0f);
  - float2 low = f.lo; // returns f.xy (1.0f, 2.0f)
  - float2 high = f.hi; // returns f.zw (3.0f, 4.0f)
  - float2 o = f.odd; // returns f.yw (2.0f, 4.0f)
  - float2 e = f.even; // returns f.xz (1.0f, 3.0f)

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

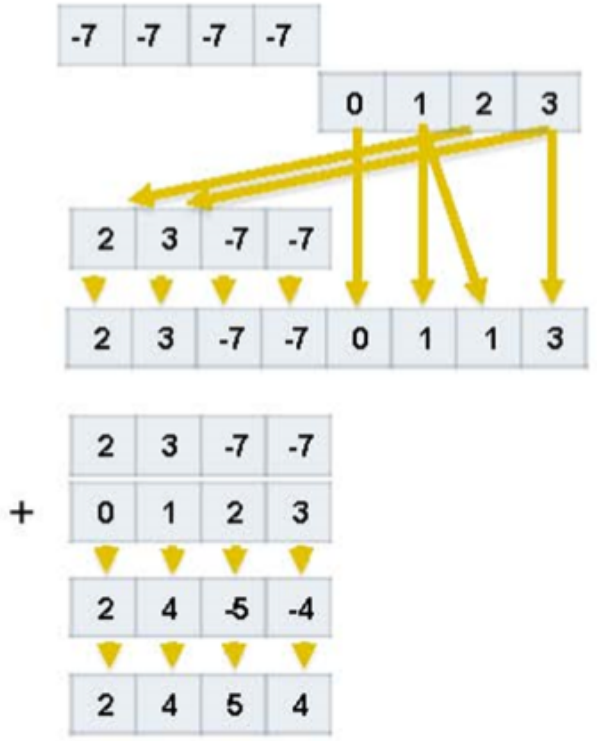
Vector access suffix	Description
.lo	Returns the lower half of a vector
.hi	Returns the upper half of a vector
.odd	Returns the odd components of a vector
.even	Returns the even components of a vector

Vector Operations in Kernels

- Operations and functions accepts vectors
  - float4 fvec = (float4) (1.0f, 2.0f, 7.0f, 8.0f);
  - float4 gvec = (float4) (4.0f, 3.0f, 2.0f, 1.0f);
  - fvec -= gvec; // fvec = (-3.0, -1.0, 5.0, 7.0)
  - gvec = abs(fvec); // gvec = (3.0, 1.0, 5.0, 7.0)
  - float c = 3.0f;
  - fvec = gvec + c; // fvec = (6.0, 4.0, 8.0, 10.0)
  - fvec *= 3.142; // Component-wise multiplication
  - gvec = sin(fvec); // Sin() on each component
  - float4 svec = sin(fvec.xxyy); // Sin() on x, x, y, y components
  - float4 dotp = dot(fvec, gvec) // Dot projduct

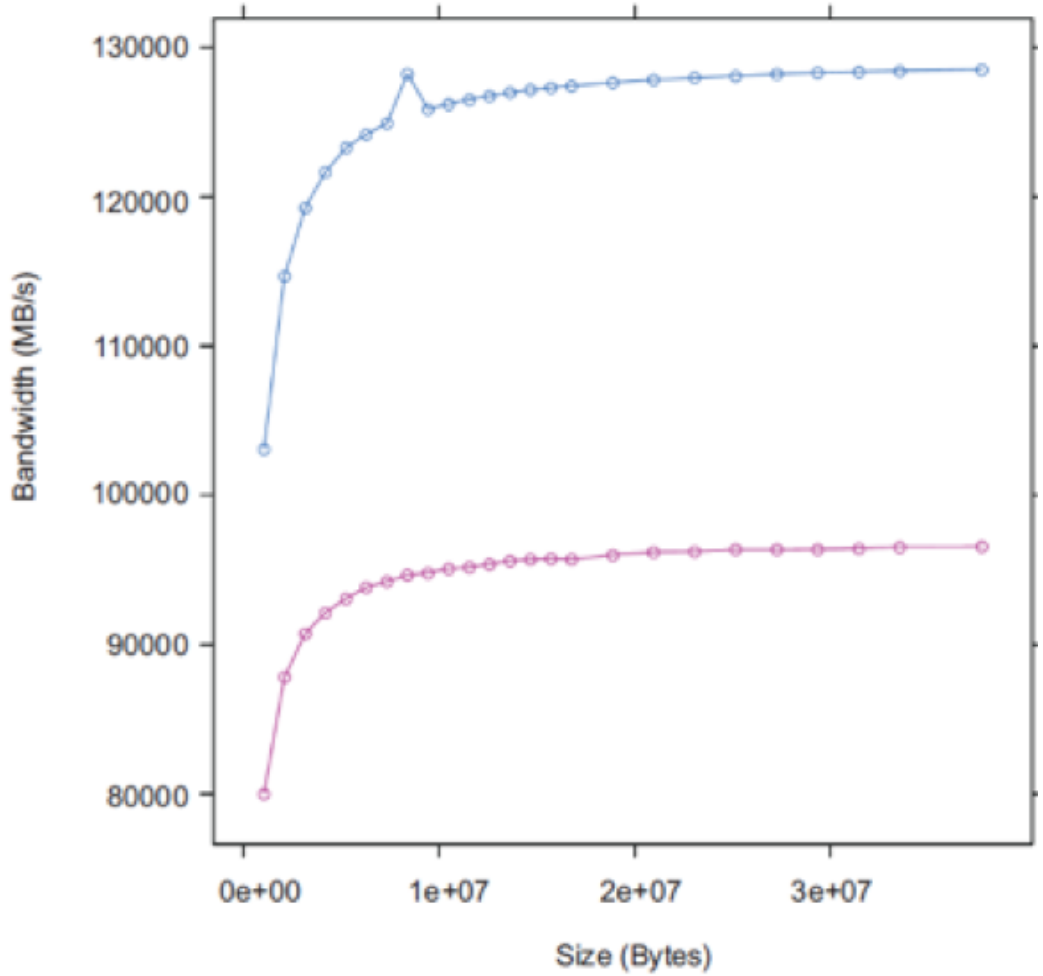
A Combined Vector Operations Example

- Vector literal
  - int4 vi0 = (int4) -7;
  - int4 vi1 = (int4) (0, 1, 2, 3);
- Vector components
  - vi0.lo = vi1.hi;
  - int8 v8 = (int8) (vi0, vi1.s01, vi1.odd);
- Vector operations
  - vi0 += vi1;
  - vi0 = abs(vi0);



Vector Data Type Improves Bandwidth

```
__kernel void
Copy4(__global const float4 * input,
      __global float4 * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
__kernel void
Copy1(__global const float * input,
      __global float * output)
{
  int gid = get_global_id(0);
  output[gid] = input[gid];
  return;
}
```

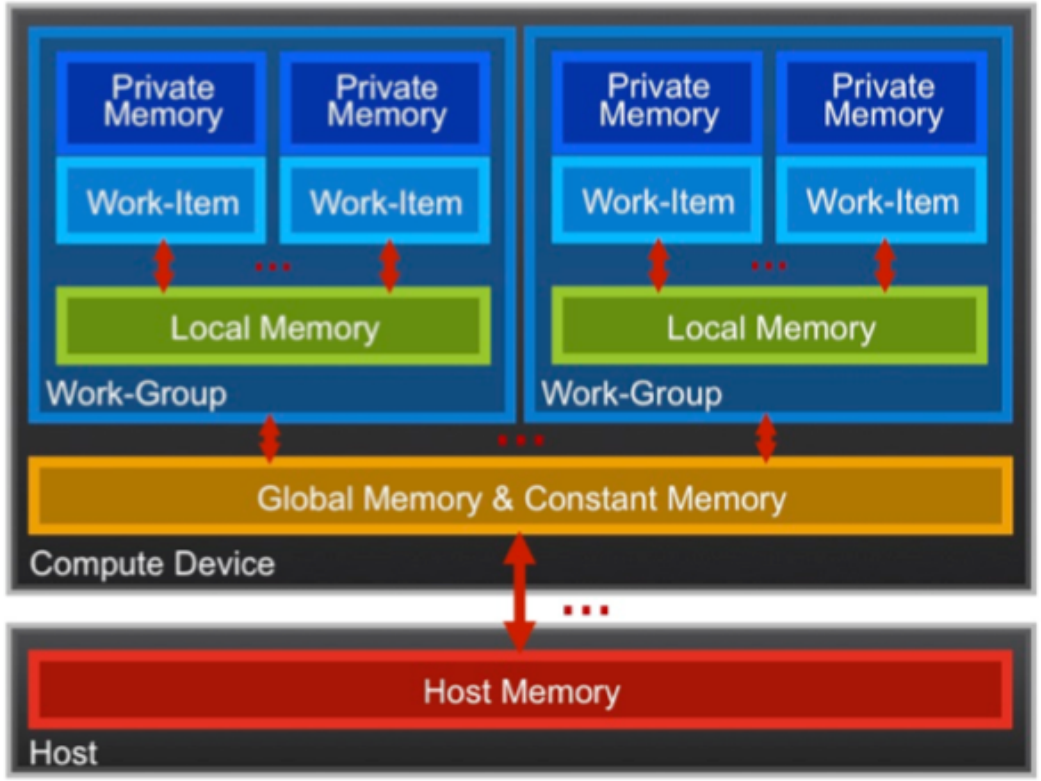


Two Kernels: One Using float4 (blue), the Other float1 (red)

Memories

Kernel Address Space Qualifiers

- Private mem: __private
  - R/W per work-item
- Local mem: __local
  - R/W by the work-items in a work-group
- Global mem: __global
  - R/W by work-items in all work-groups
- Const mem: __constant
  - R-only global mem visible to all work-items
  - Host allocs and inits
- Host mem:
  - On the CPU
- Explicit memory management
  - You move data explicitly host->global->local and back



Memory Objects

- Memory objects are defined to be in a separate space from the host CPU's memory
- Two types of memory objects:
  - Buffers
    - One-dimensional arrays in the traditional CPU sense
    - The contents of buffer objects are fully exposed within kernels and can be accessed using pointers
  - Images
    - Multidimensional structure
    - Opaque types (Can only be accessed with read and write functions)
    - Limited to a range of types relevant to graphic data

```
__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);
}
```

Global Memory

- Use to refer to memory objects (buffers or images)
  - Memory allocated with clCreateBuffer() is in global memory
  - Usually passed into kernel as an arg
  - Can pass to functions

```
float myFunc( int i, __global const float *x ) {
  float tmp = *(x+i);              // Could also use x[i]
  return 2.0*tmp*tmp*tmp+3.0*sin(tmp)-1.0;
}

__kernel void myKernel( const int n,
                        __global const float *inArr,
                        __global float *outArr )
{
  int i = get_global_id(0);
  outArr[i] = myFunc(i, inArr);
}
```

Constant Memory

- Similar to __global but read-only within a kernel
  - May be allocated in same global memory
  - Device may have separate constant mem cache
  - Vars with ‘program scope’ must be declared __constant

```
// Create a read-only cl_mem obj. Initialize with these host values.
const float table_h[5] = {1.0, 3.0, 5.0, 3.0, 1.0};
cl_mem table_d = clCreateBuffer( ctx, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
                                 sizeof(float)*5, table_h, NULL );

// Pass read-only buffer to kernel just like any other cl_mem obj
err  = clSetKernelArg( kernel, 0, sizeof(int),    &numElems ); // As earlier
err |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &vecA_d );   // As earlier
err |= clSetKernelArg( kernel, 2, sizeof(cl_mem), &table_d );  // constant
```

```
__constant int myConst = 123;      // Var has 'program scope' (use pre-processor)

__kernel void myKernel(const int n, __global float *x, __constant float *table)
{
  for ( i=0; i<5; i++ )
    // Do something with table[i]
}
```

Local Memory

- Use to share items between work-items in a work-group
  - Allows work-items to communicate (within their work-group)
  - Can save re-loading data from global memory
  - Can replace a __private (register) if same value in all work-items
  - Useful for (temp) arrays – quicker than using global memory

```
__kernel void myKernel( const int n, ... )
{
  __local float tmpScale;          // Cannot initialize on this line
  __local float wgValues[64];      // Hard-coded size in kernel (Bad-ish)
  int g_idx = get_global_id(0);    // global work-item id
  int l_idx = get_local_id(0);     // local work-item id in work-group
  tmpScale = 123.456;              // Bad: All work-items do this! Use __constant?

  wgValues[l_idx] = myComputation(g_idx, ...);
  ...
}
```

Private Memory

- A kernel's local variables and a function's args are private

```
float myFunc( float x, float y ) {
  return 2.0*x*x*x+3.0*y*y-1.0;
}
__kernel void myKernel( const int n, ... ) {
  float a, b;
  float temp[4];
 __global int *p;          // Is p private or global?
 int i = get_global_id(0);
 // Kernel body will call myFunc(a,b) at some point
}
```

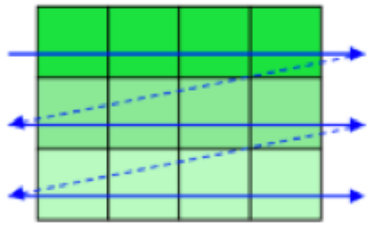
- GPUs alloc __privates in Compute Unit's register file
  - Fastest access times
  - Limited number of registers per CU (and hence per work-item)

Accessing Memory

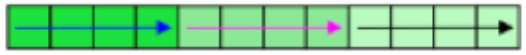
- Coalesced __global memory access for efficiency
  - Work-item i to access array[i]
  - Work-item i+1 to access array[i+1]

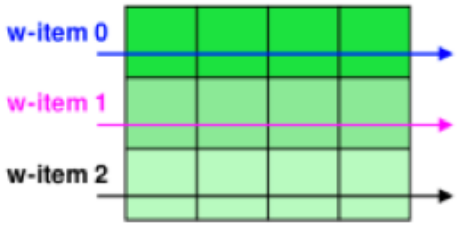
- Access to C matrices (row-major)
  - E.g., a 1D index-space where a single work-item processes an entire row of a C matrix

```
// C (CPU cache-friendly)
for (r=0; r<height; r++)
  for (c=0; c<width; c++)
    z = A[r*width+c];
```

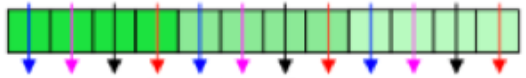


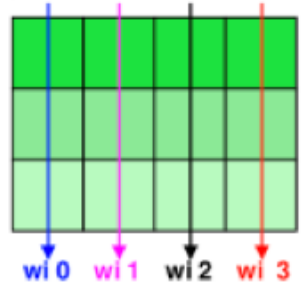
```
// Kernel (non-coalesced)
WI=get_global_id(0);
for (c=0; c<width; c++ )
  z = A[WI*width+c];
```






```
// Kernel (coalesced)
WI=get_global_id(0);
for (r=0; r<height; r++ )
  z = A[r*width+WI];
```





Memory Sizes

- Limits to various memory spaces
  - Can spill from faster to slower memory (or crash)

```
cl_ulong dinfo;   // Properties below of this type unless indicated
clGetDeviceInfo( &device_id, PROPERTY, sizeof(dinfo), &dinfo, NULL );
```

CL_DEVICE_GLOBAL_MEM_SIZE	Size of global device memory in bytes.
CL_DEVICE_MAX_MEM_ALLOC_SIZE	Max memory object allocation size in bytes. The minimum value is *max(1/4 CL_DEVICE_GLOBAL_MEM_SIZE , 128MB)*
CL_DEVICE_LOCAL_MEM_SIZE	Size of local memory arena in bytes. Minimum value is 32KB.
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	Size of constant buffer memory arena in bytes. Minimum value is 64KB
CL_DEVICE_MAX_CONSTANT_ARGS	(cl_uint) Max number of arguments declared with __constant qualifier in a kernel. The minimum value is 8.
CL_DEVICE_LOCAL_MEM_TYPE	(cl_device_local_mem_type) Type of local memory supported. Can be CL_LOCAL implying dedicated local mem storage such as SRAM, or CL_GLOBAL.

Work-item Synchronization

- Only possible within a work-group
  - Can't sync with work-items in other work-groups
  - Can't sync one work-group with another
- Use barrier(type) in kernel where *type* is
  - CLK_LOCAL_MEM_FENCE: ensure consistency in local mem
  - CLK_GLOBAL_MEM_FENCE: ensure consistency in global mem
- All work-items in work-group must issue the barrier() call and same number of calls

```
__kernel void BadKernel(...) {
  int i = get_global_id(0);
  ...
  // ERROR: Not all WIs reach barrier
  if ( i % 2 )
    barrier(CLK_GLOBAL_MEM_FENCE);
}
```

```
__kernel void BadKernel(...) {
  int i = get_local_id(0);
  ...
  // ERROR: WIs issue different number
  for ( j=0; j<=i; j++ )
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Use with __local Memory

- Barrier often used when initializing __local memory
  - Kernel must initialize local memory

```
__kernel void kMat( const int n, __global float *A, __local float *tmp_arr
{
  // Could also have some fixed size local array
  // __local float tmp_arr[64];

  int gbl_id = get_global_id(0);   // ID within entire index space
  int loc_id = get_local_id(0);    // ID within this work-group
  int loc_sz = get_local_size(0);  // Size of this work-group

  // For some reason we want to fill up the first half of the local array
  if ( loc_id < loc_sz/2 )
    tmp_arr[loc_id] = A[gbl_id];

  // All work-items must hit barrier. They'll all see a consistent tmp_arr[
  barrier(CLK_LOCAL_MEM_FENCE);

  // Each work-item can now use the elements from tmp_arr[] safely.
  // Often used if we'd be repeatedly accessing the same A[] elements.
  for ( j=0; j<loc_sz; j++ )
    my_compute( gbl_id, tmp_arr[j], A );
```

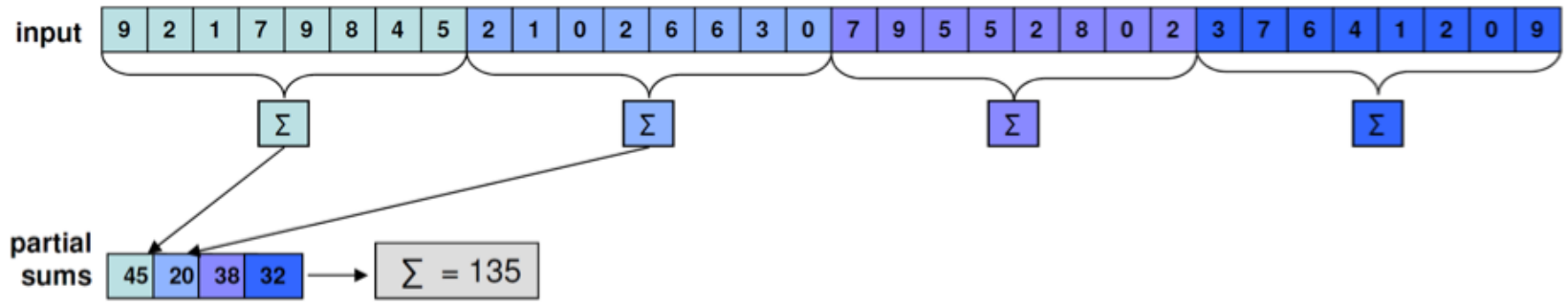
General Optimization Tips

- Use local memory
- Set work-group size explicitly
- Unroll loops
- Reduce data and instructions
- Use built-in vector types

Example: Parallel Reduction

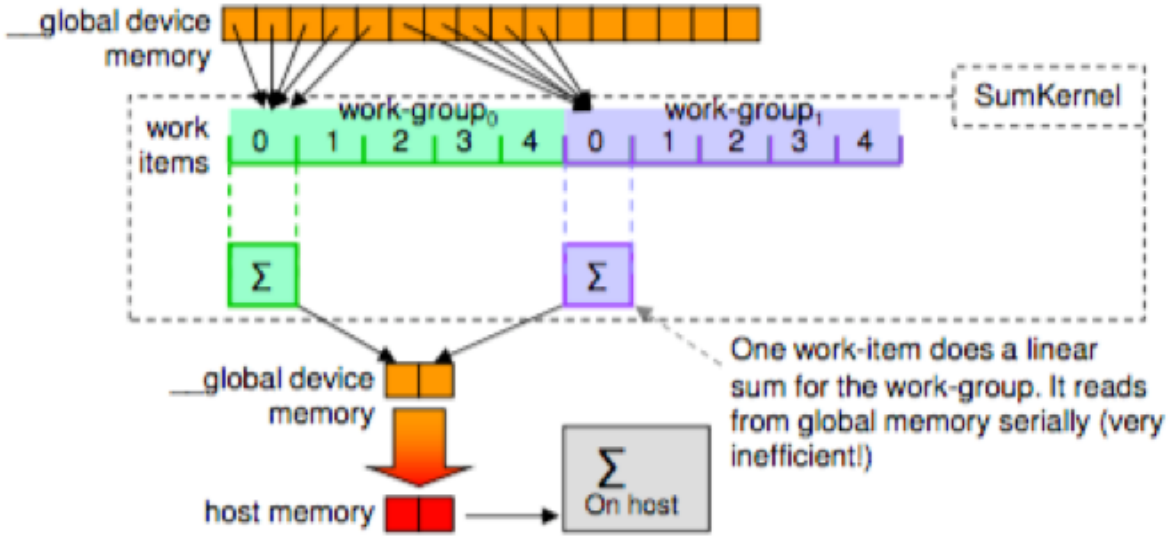
- Reduction of an array of numbers to a single value, e.g., sum

```
void sumCPU( const int n, const float *x,
             float *res )
{
  float sum = 0.0;
  for ( int i=0; i<n; i++ )
    sum += x[i];
  *res = sum;
}
```



Parallel Reduction on GPU (1st attempt)

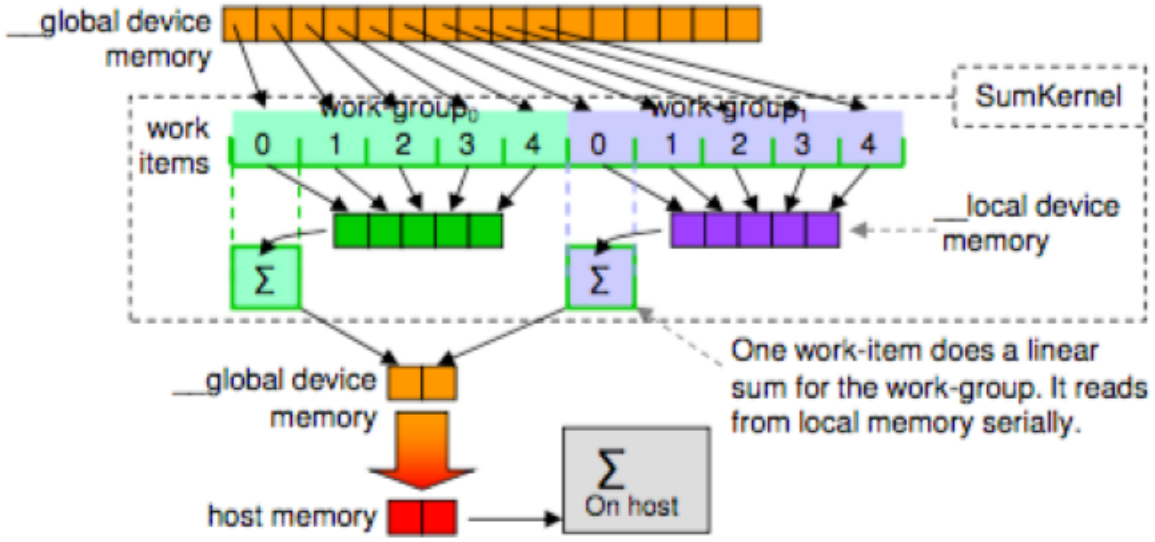
- Use one work-item to perform a sum within a work-group




```
__kernel void sumGPU1( const uint n, __global const float *x,
                       __global float *partialSums ) {
  if ( get_local_id(0) == 0 ) {                    // Many idle work-items!
    float group_sum = x[get_global_id(0)];
    for ( int i=1; i<get_local_size(0); i++ )
      group_sum += x[get_global_id(0)+i];          // Should check (gid+i) < n
    partialSums[get_group_id(0)] = group_sum;      // Write sum to output array
  }
  // Add barrier(CLK_GLOBAL_MEM_FENCE) if doing other work in kernel
}
```

Parallel Reduction on GPU (2nd attempt)

- Memory optimization (copy to __local memory in parallel)

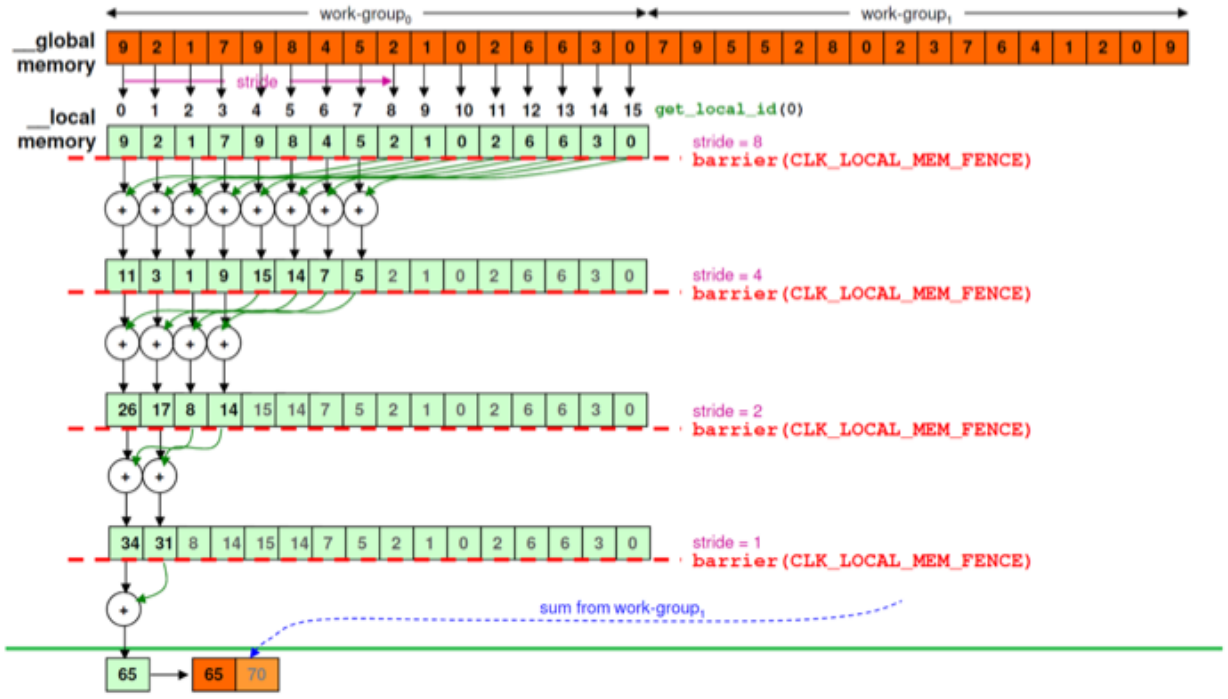



```
__kernel void sumGPU2( const uint n, __global const float *x,
                       __global float *partialSums, __local float *localCopy ) {
  localCopy[get_local_id(0)] = x[get_global_id(0)]; // Init the localCopy array
    barrier(CLK_LOCAL_MEM_FENCE);                   // All work-items must call
  if ( get_local_id(0) == 0 ) {                     // Many idle work-items!
    float group_sum = localCopy[0];
    for ( int i=1; i<get_local_size(0); i++ )
      group_sum += localCopy[i];                    // Sum up the local copy
    partialSums[get_group_id(0)] = group_sum;       // Write sum to output array
  } // Add barrier(CLK_GLOBAL_MEM_FENCE) if doing other work in kernel
}
```

Parallel Reduction on GPU (3rd attempt)

- Repeatedly half the work-group, adding one half to the other

```
__kernel void sumGPU3( const uint n, __global const float *x,
                       __global float *partialSums, __local float *localSums )
{
  uint local_id   = get_local_id(0);
  uint group_size = get_local_size(0);

  // Copy from global mem in to local memory (should check for out of bounds)
  localSums[local_id] = x[get_global_id(0)];
  for (uint stride=group_size/2; stride>0; stride /= 2) { // stride halved at loop

    // Synchronize all work-items so we know all writes to localSums have occurred
    barrier(CLK_LOCAL_MEM_FENCE);

    // First n work-items read from second n work-items (n=stride)
    if ( local_id < stride )
      localSums[local_id] += localSums[local_id + stride]
  }
  // Write result to nth position in global output array (n=work-group-id)
  if ( local_id == 0 )
    partialSum[get_group_id(0)] = localSums[0];
}
```



Parallel Reduction on GPU (4th attempt)

- Re-order to remove a couple of loop iterations

```
__kernel void sumGPU4( const uint n, __global float *x,
                       __global float *partialSums, __local float *localSums ) {
  uint global_id   = get_global_id(0);     // Gives where to read from
  uint global_size = get_global_size(0);   // Used to calc where to read from
  uint local_id    = get_local_id(0);      // Gives where to read/write local mem
  uint group_size  = get_local_size(0);    // Used to calc initial stride

  // Copy from global mem in to local memory (doing first iteration)
  localSums[local_id] = x[global_id] + x[global_id + global_size];
  barrier(CLK_LOCAL_MEM_FENCE);
  for (uint stride=group_size/2; stride>1; stride>>=1 ) { // >>=1 does same as /=2
    // First n work-items read from second n work-items (n=stride)
    if ( local_id < stride )
      localSums[local_id] += localSums[local_id + stride];

    // Synchronize so we know all work-items have written to localSums
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  // Last iter: write result to nth position in global x array (n=work-group id)
  if ( local_id == 0 )
    x[get_group_id(0)] = localSums[0]+localSums[1];
  }
```

Example: Linear Algebra

- Definition:
  - The branch of mathematics concerned with the study of vectors, vector spaces, linear transformations and systems of linear equations
- Example: Consider the following system of linear equations

$$x + 2y + z = 1$$

$$x + 3y + 3z = 2$$

$$x + y + 4z = 6$$

  - This system can be represented in terms of vectors and a matrix as the classic “Ax = b” problem

$$\begin{pmatrix} 1 & 2 & 1 \\ 1 & 3 & 3 \\ 1 & 1 & 4 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 6 \end{pmatrix}$$

Solving Ax=b

- LU Decomposition:
  - Transform a matrix into the product of a lower triangular and upper triangular matrix. It is used to solve a linear system of equations

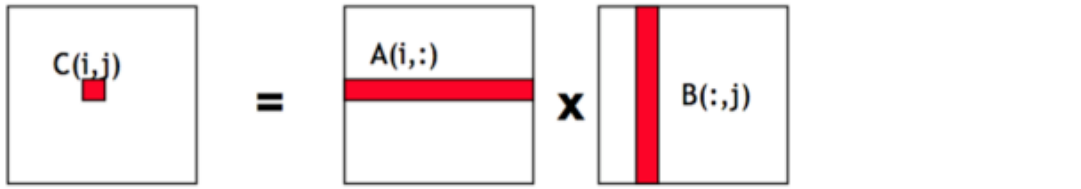
$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 3 & 3 \\ 1 & 2 & 4 \end{pmatrix}$$

  - We solve for x, given a problem Ax=b
    - Ax=b
    - LUx=b
    - $Ux=(L^{-1})b$
    - $x=(U^{-1})(L^{-1})b$

Matrix Multiplication (Sequential Version)

- Calculate C=AB, where all three matrices are NxN

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```



Dot product of a row of A and a column of B for each element of C

Matrix Multiplication (1st attempt)

```
__kernel void mat_mul(
 const int N,
 __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
    // C(i, j) = sum(over k) A(i,k) * B(k,j)
    for (k = 0; k < N; k++) {
      C[i*N+j] += A[i*N+k] * B[k*N+j];
    }
}
```

Mark as a kernel function and specify memory qualifiers

Remove outer loops and set work-item co-ordinates

Matrix Multiplication (2nd attempt)

- Rearrange and use a local scalar for intermediate C element values

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int k;
  int i = get_global_id(0);
  int j = get_global_id(1);
  float tmp = 0.0f;
  for (k = 0; k < N; k++)
   tmp += A[i*N+k]*B[k*N+j];
  }
  C[i*N+j] += tmp;
}
```

Matrix Multiplication (3rd attempt)

- One work item per row of C

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
   int j, k;
   int i = get_global_id(0);
   float tmp;
   for (j = 0; j < N; j++) {
    tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += A[i*N+k]*B[k*N+j];
    C[i*N+j] = tmp;
   }
}
```

Matrix Multiplication (4th attempt)

- Copy a row of A into private memory from global memory

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
  int j, k;
  int i =
    get_global_id(0);
  float tmp;
  float Awrk[1024];
  for (k = 0; k < N; k++)
    Awrk[k] = A[i*N+k];

  for (j = 0; j < N; j++) {
    tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += Awrk[k]*B[k*N+j];

    C[i*N+j] += tmp;
  }
}
```

Setup a work array for A in private memory*

Matrix Multiplication (5th attempt)

- B column shared between work-items

```
__kernel void mmul(
      const int N,
    __global float *A,
    __global float *B,
    __global float *C,
    __local  float *Bwrk)
{
 int j, k;
 int i =
    get_global_id(0);

 int iloc =
     get_local_id(0);

 int nloc =
     get_local_size(0);

 float tmp;
 float Awrk[1024];

 for (k = 0; k < N; k++)
   Awrk[k] = A[i*N+k];

 for (j = 0; j < N; j++) {

   for (k=iloc; k<N; k+=nloc)
     Bwrk[k] = B[k* N+j];

   barrier(CLK_LOCAL_MEM_FENCE);

   tmp = 0.0f;
   for (k = 0; k < N; k++)
     tmp += Awrk[k]*Bwrk[k];

   C[i*N+j] = tmp;

   barrier(CLK_LOCAL_MEM_FENCE);
 }
}
```

Pass a work array in local memory to hold a column of B. All the work-items do the copy "in parallel" using a cyclic loop distribution (hence why we need iloc and nloc)

Matrix Multiplication Performance

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3
C row per work-item, A private, B local	10,047.5	8,181.9

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

Synchronization and Events

Synchronization

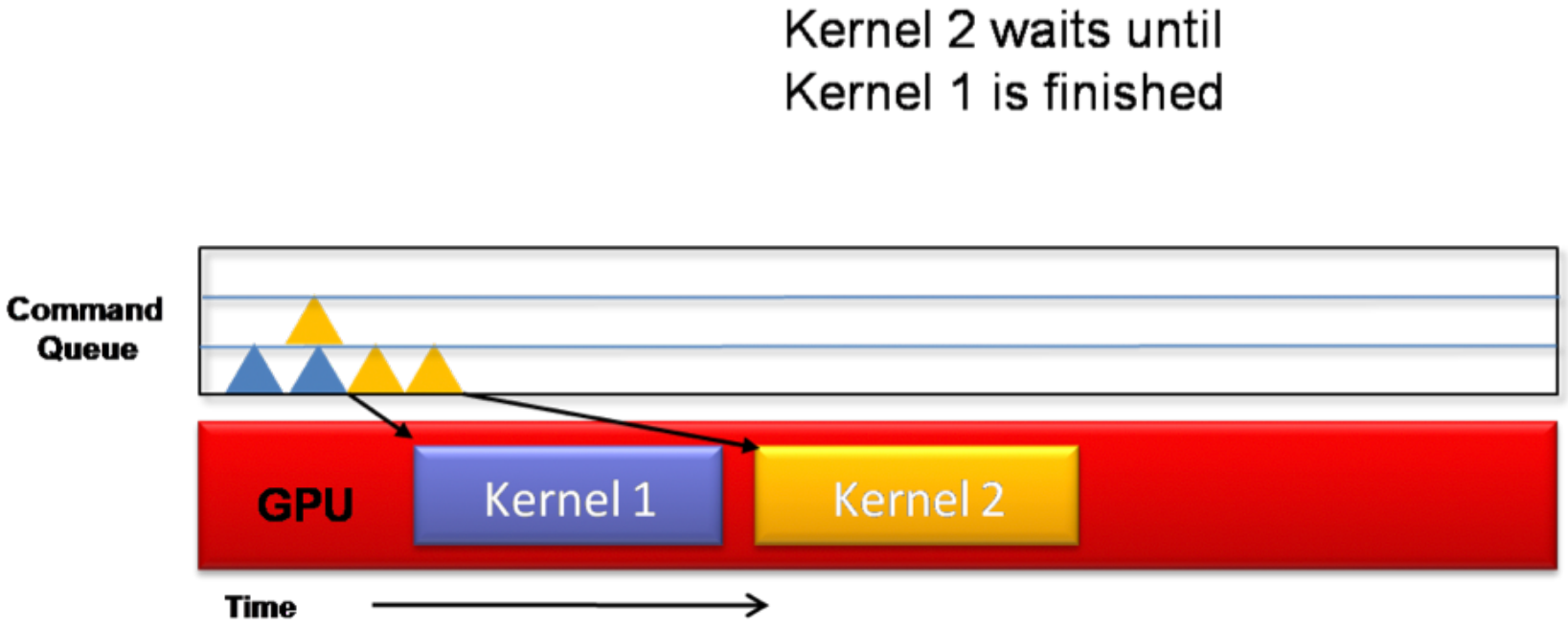
- Various synchronization
  - Between work-items in a work-groups
  - Between kernels (and other commands) in a queue
  - Between kernels (and other commands) in separate queues
  - Between the host and the queues
- Each individual queue can execute in order or out of order
  - In-order queue: commands executed in order submitted
  - Out-of-order queue: commands scheduled by OpenCL
- You must explicitly synchronize between queues
  - Multiple devices each have their own queue
  - Use events (an object that communicates the status of commands in OpenCL) to synchronize
  - clEnqueue*(…, num_events_in_waitlist, *event_waitlist, *event_out)

```
cl_event    k_events[2];

err = clEnqueueNDRangeKernel(commands, kernel1, 1,
 NULL, &global, &local, 0, NULL, &k_events[0]);

err = clEnqueueNDRangeKernel(commands, kernel2, 1,
 NULL, &global, &local, 0, NULL, &k_events[1]);

err = clEnqueueNDRangeKernel(commands, kernel3, 1,
 NULL, &global, &local, 2, k_events, NULL);
```

Enqueue two kernels that expose events

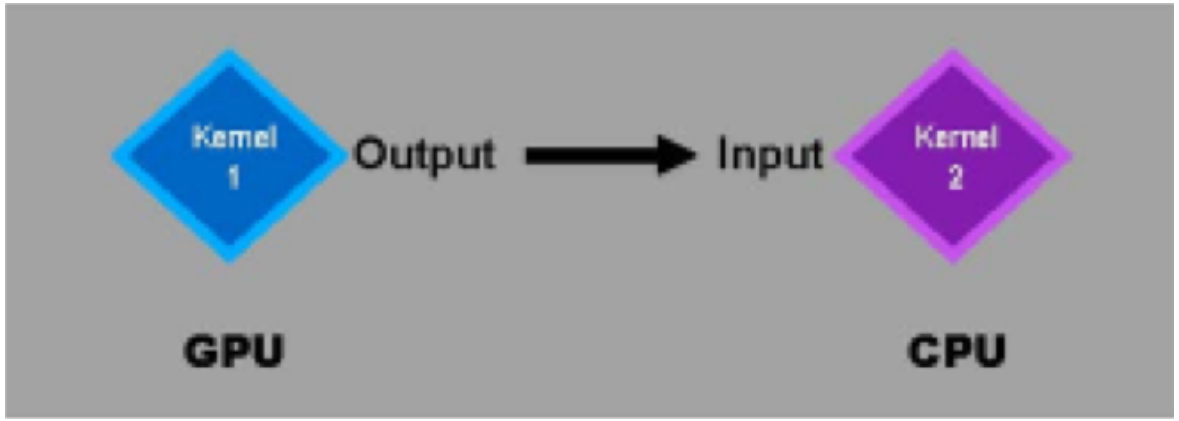
Wait to execute until two previous events complete

Synchronization with One Device and One Queue

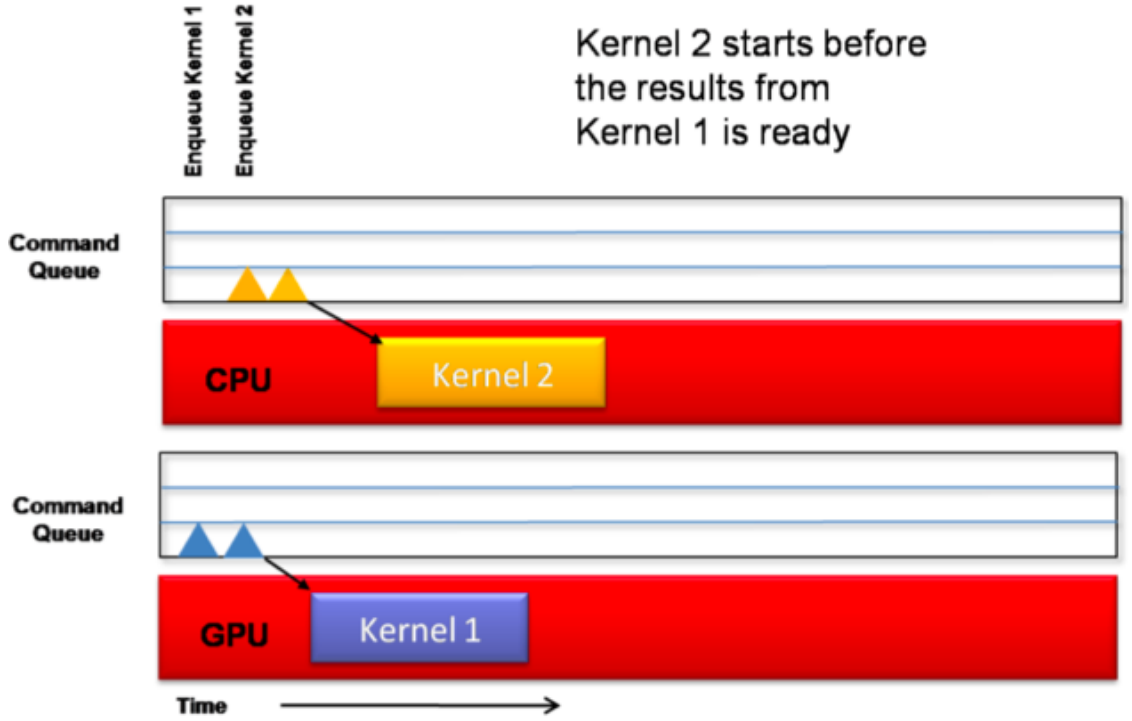


Synchronization with Two Devices and Two Queues

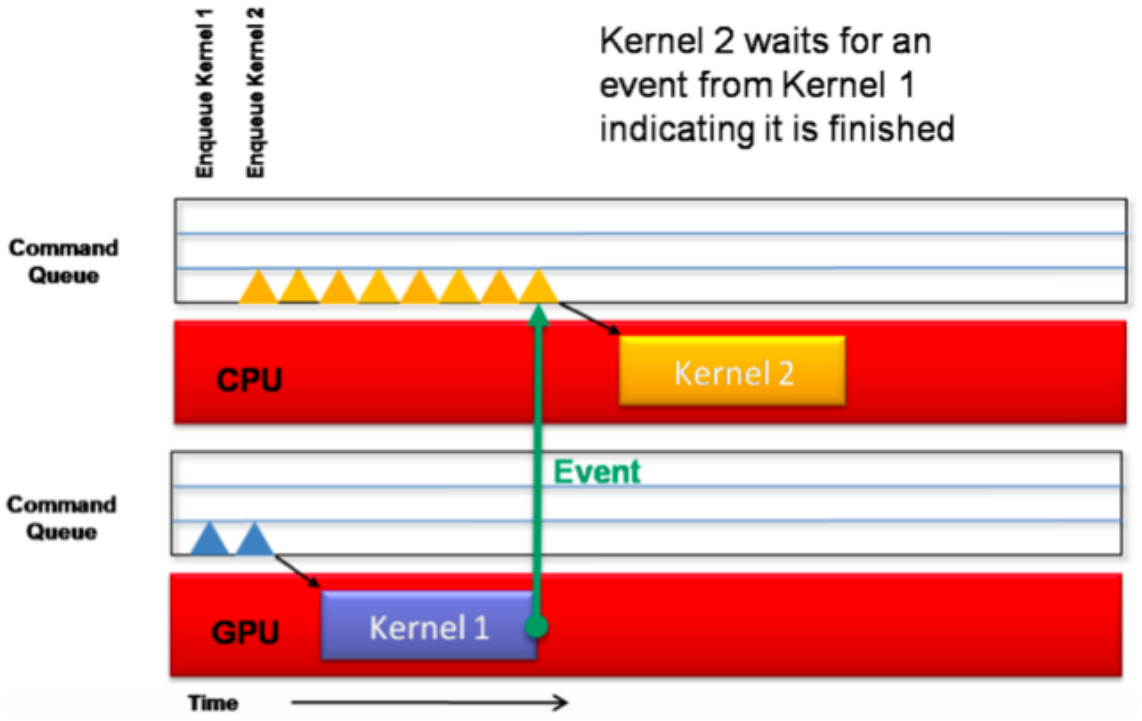
Explicit dependency: Kernel 1 must finish before Kernel 2 starts



Synchronization with Two Devices and Two Queues (Cont.)



Two Devices with Two Queues, unsynchronized



Two Devices with Two Queues, synchronized using events

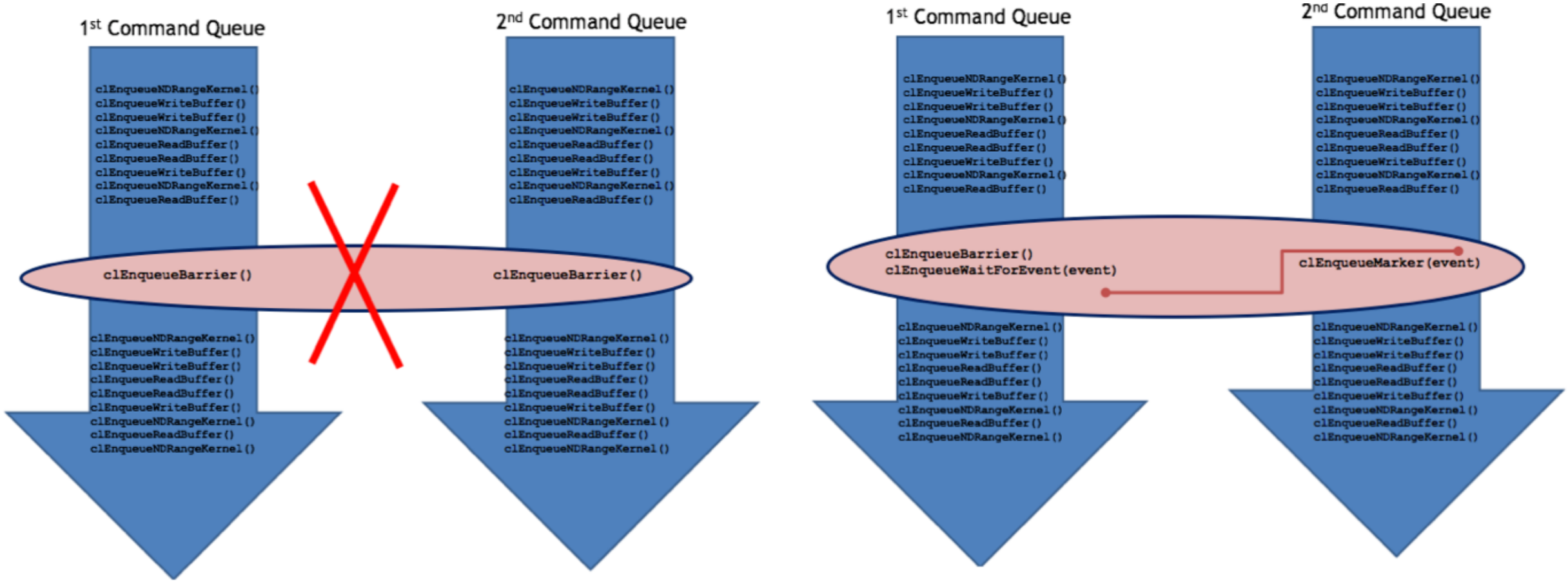
Using Events on the Host

- clWaitForEvents(num_events, *event_list)
  - Blocks until events are complete
- clEnqueueWaitForEvents(queue, num_events, *event_list)
  - Inserts a “WaitForEvents” into the queue
- clEnqueueMarker(queue, *event)
  - Returns an event for a marker that moves through the queue
- clGetEventInfo()
  - Command type and status: CL_QUEUED, CL_SUBMITTED, CL_RUNNING, CL_COMPLETE, or error code
- clSetEventCallback()
  - Called when command identified by event has completed

Why Events But Not Barrier?

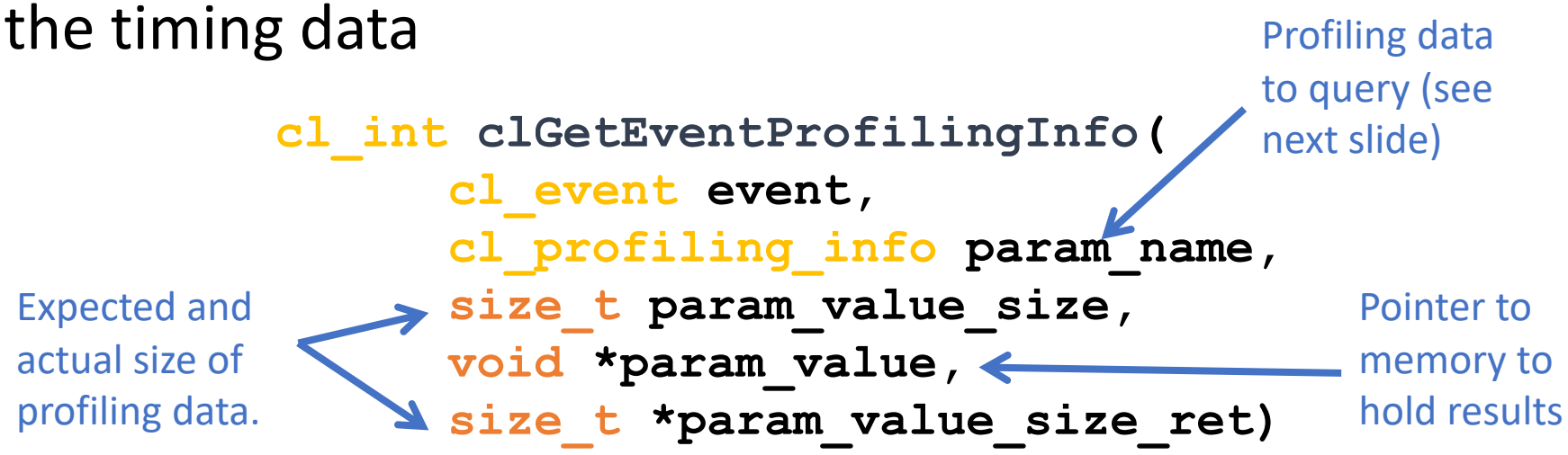
- A barrier defines a synchronization point … commands following a barrier wait to execute until all prior enqueued commands complete
  - Cl_int clEnqueueBarrier(cl_command_queue queue)
- Events provide fine grained control … this can really matter with an out-of-order queue
- Events work between commands in the different queues … as long as they share a context
- Events convey more information than a barrier … provide info on state of a command, not just whether it's complete or not

clEnqueueBarrier w/o Events



Profiling with Events

- Profiling works by turning an event into an opaque object to hold timing data
- Profiling is enabled when a queue is created with the CL_QUEUE_PROFILING_ENABLE flag set
- When profiling is enabled, the following function is used to extract the timing data



Cl_profiling_info values

- CL_PROFILING_COMMAND_QUEUED
  - The device time in nanoseconds when the command is enqueued in a command-queue by the host. (cl_ulong)
- CL_PROFILING_COMMAND_SUBMIT
  - The device time in nanoseconds when the command is submitted to compute device. (cl_ulong)
- CL_PROFILING_COMMAND_START
  - The device time in nanoseconds when the command starts execution on the device. (cl_ulong)
- CL_PROFILING_COMMAND_END
  - The device time in nanoseconds when the command has finished execution on the device. (cl_ulong)

Profiling Examples

```
cl_event prof_event;
cl_command_queue comm;

comm = clCreateCommandQueue(
        context, device_id,
        CL_QUEUE_PROFILING_ENABLE,
        &err);

err = clEnqueueNDRangeKernel(
          comm, kernel,
          nd, NULL, global, NULL,
          0, NULL, prof_event);

clFinish(comm);
err = clWaitForEvents(1,
&prof_event );
```

```
cl_ulong start_time, end_time;
size_t return_bytes;

err = clGetEventProfilingInfo(
          prof_event,
CL_PROFILING_COMMAND_QUEUED,
          sizeof(cl_ulong),
          &start_time,
          &return_bytes);

err = clGetEventProfilingInfo(
          prof_event,
          CL_PROFILING_COMMAND_END,
          sizeof(cl_ulong),
          &end_time,
          &return_bytes);

run_time =(double)(end_time -
start_time);
```

Events Inside Kernels

```
// A, B, C kernel args … global  buffers.
// Bwrk is a local buffer


for(k=0;k<Pdim;k++)
          Awrk[k] = A[i*Ndim+k];


for(j=0;j<Mdim;j++){
    event_t ev_cp  = async_work_group_copy(
        (__local float*) Bwrk, (__global float*) B,
        (size_t) Pdim, (event_t) 0);


    wait_group_events(1, &ev_cp);


    for(k=0, tmp= 0.0;k<Pdim;k++)
           tmp  += Awrk[k] *  Bwrk[k];
    C[i*Ndim+j] = tmp;
}
```

Compute a row of C = A * B

- One A col per work-item
- Work group shares rows of B

Start an async. copy for row of B returning an event to track progress.

Wait for async. copy to complete before proceeding.

Compute element of C using A from private memory and B from local memory.