Northeastern University

PROGRAMMING AND SIMULATING HETEROGENEOUS DEVICES - OPENCL AND MULTI2SIM

Rafael Ubal, Dana Schaa, Perhaad Mistry, David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA

ICPE 2012 – Boston, MA

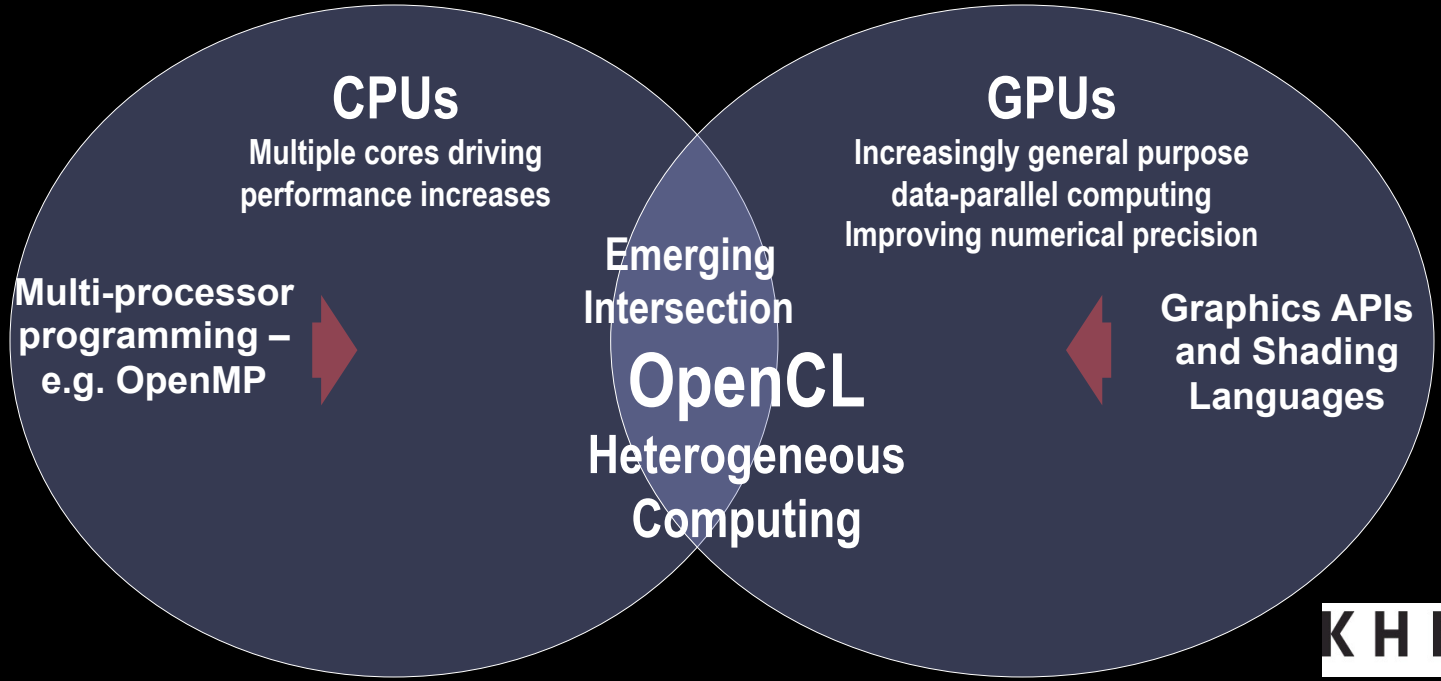
AGENDA

Northeastern University

- Part 1 – Programming with OpenCL
  - What is OpenCL ?
  - OpenCL platform, memory and programming models
  - OpenCL programming walkthrough
  - Simple OpenCL optimization example
  - Multidevice Programming
  - OpenCL Programming on a APU
  - Details about OpenCL v1.2
- Part 2 – Multi2Sim

PROCESSOR PARALLELISM

Northeastern University



KHRONOS GROUP

OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

WHAT IS OPENCL ?

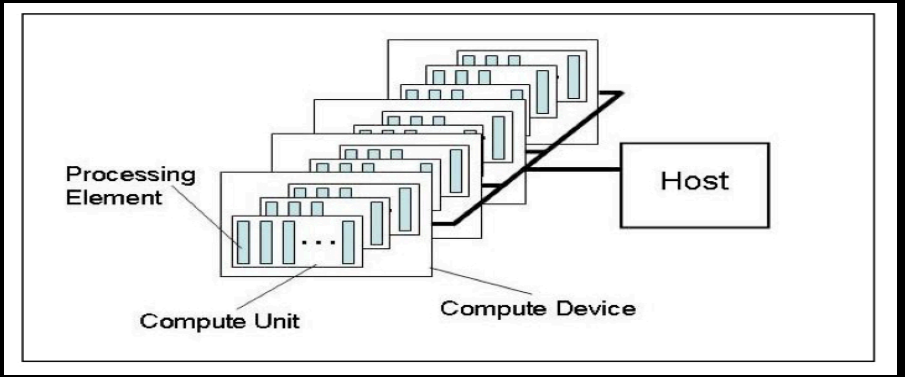
Northeastern University

- With OpenCL™ you can…
  - Leverage CPUs, GPUs, other processors such as Cell. DSPs to accelerate parallel computation
  - Get dramatic speedups for computationally intensive applications
  - Write accelerated portable code across different devices and architectures
  - Royalty free, cross-platform, vendor neutral managed by Khronos OpenCL working group
- Defined in four parts
  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model

HOST-DEVICE MODEL (PLATFORM MODEL)

Northeastern University

- The platform model consists of a host connected to one or more OpenCL devices
- A device is divided into one or more compute units
- Compute units are divided into one or more processing elements
- The host is whatever the OpenCL library runs on
  - Usually x86 CPUs
- Devices are processors that the library can talk to
  - CPUs, GPUs, and other accelerators
- For AMD
  - All CPUs are 1 device (each core is a compute unit and processing element)
  - Each GPU is a separate device



DISCOVERING PLATFORMS AND DEVICES

Northeastern University

- Obtaining Platform Information
  - To get the number of platforms available to the implementation
- Obtaining Device Information
  - Once a platform is selected, we can query for the devices present
  - Specify types of devices interested in (e.g. all devices, CPUs only, GPUs only)
- These functions are called twice each time
  - First call is to determine the number of platforms / devices
  - Second retrieves platform / device objects

```
cl_int  clGetPlatformIDs (cl_uint num_entries,
                          cl_platform_id *platforms,
                          cl_uint *num_platforms)
```

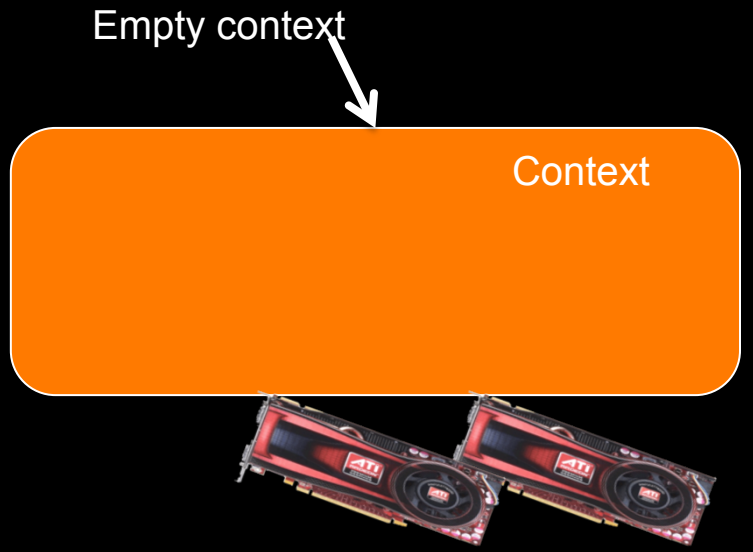
Get Platform Information

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

Get Device Information

CONTEXTS

- A context is associated with a list of devices
  - All OpenCL resources will be associated with a context as they are created
- 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 operated on by the device
  - Command queues: coordinators of execution of the kernels on the devices



CREATING A CONTEXT

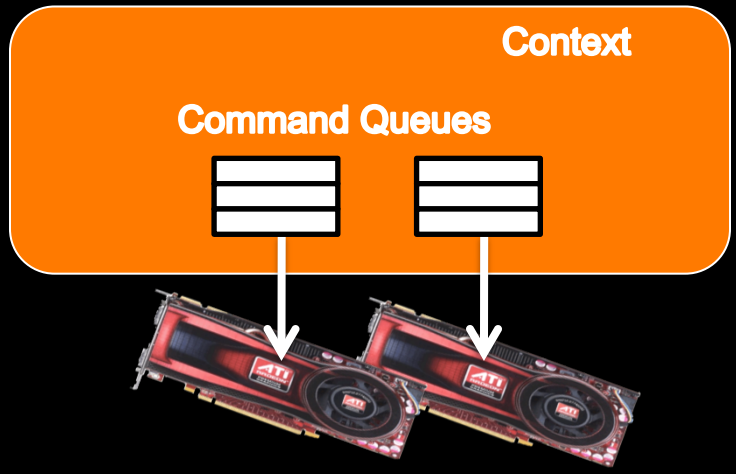
Northeastern University

- This function creates a context given a list of devices
- The properties argument specifies which platform to use
- The function also provides a callback mechanism for reporting errors to the user

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

CREATING A COMMAND QUEUE

- By supplying a command queue as an argument, the device being targeted can be determined
- The command queue properties specify:
  - If out-of-order execution of commands is allowed
  - If profiling is enabled
- Creating multiple command queues to a device is possible

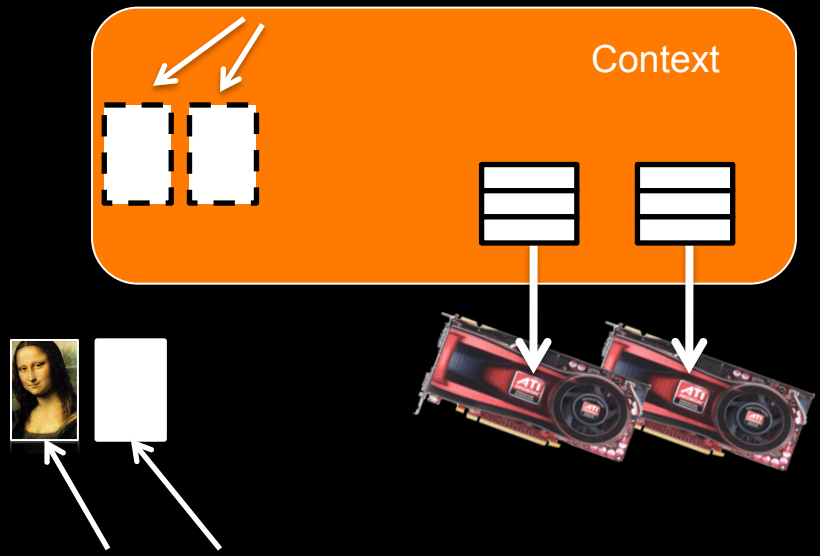



```
cl_command_queue  clCreateCommandQueue (cl_context context,
                                        cl_device_id device,
                                        cl_command_queue_properties properties,
                                        cl_int *errcode_ret)
```

MEMORY OBJECTS

- Memory objects are OpenCL data that can be moved on and off devices
- Classified as either buffers or images
- Buffers
  - Contiguous memory – stored sequentially and 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

Uninitialized OpenCL buffers - original data will be transferred to/from these objects



Original input/output data
(not OpenCL memory objects)

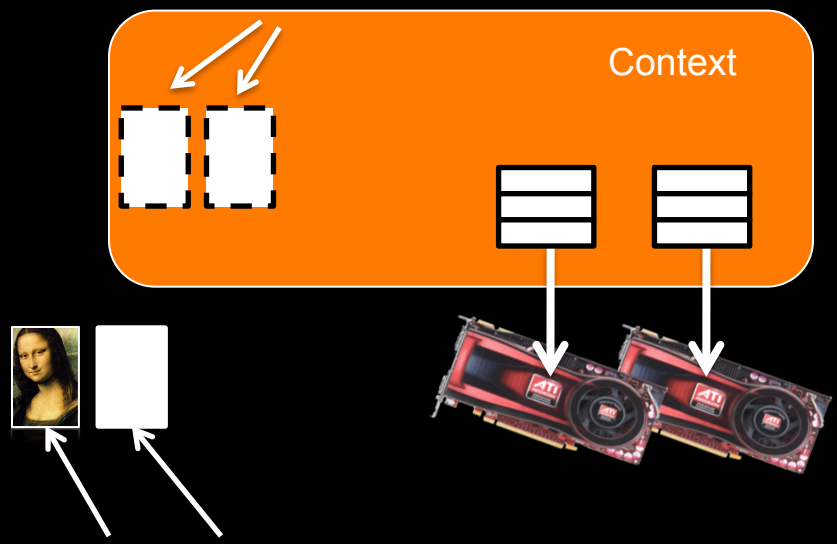
MEMORY OBJECTS

- Memory objects are associated with a context
  - They must be explicitly copied to a device prior to execution (covered next)

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

- `cl_mem_flags` specify:
- Combination of reading and writing allowed on data
- If the host pointer itself should be used to store the data
- If the data should be copied from the host pointer

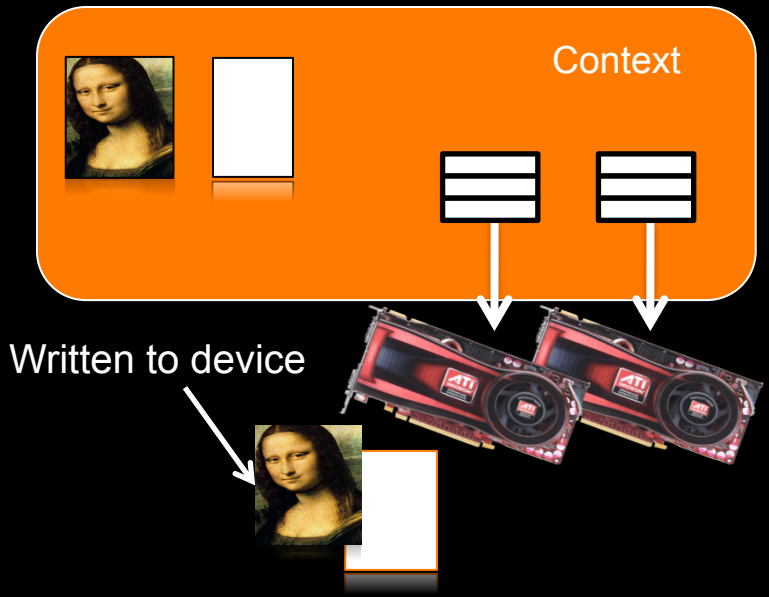
Uninitialized OpenCL buffers - original data will be transferred to/from these objects



Original input/output data (not OpenCL memory objects)

TRANSFERRING DATA

- OpenCL provides commands to transfer data to and from devices
  - clEnqueue{Read|Write}{Buffer|Image}
- Objects are transferred to devices by specifying an action (read or write) and a command queue
  - Data moved from host array into OpenCL buffer
  - Validity of objects on multiple devices is undefined by the OpenCL spec (i.e. are vendor specific)



Images are redundant show that they are part of the context and physically on the device

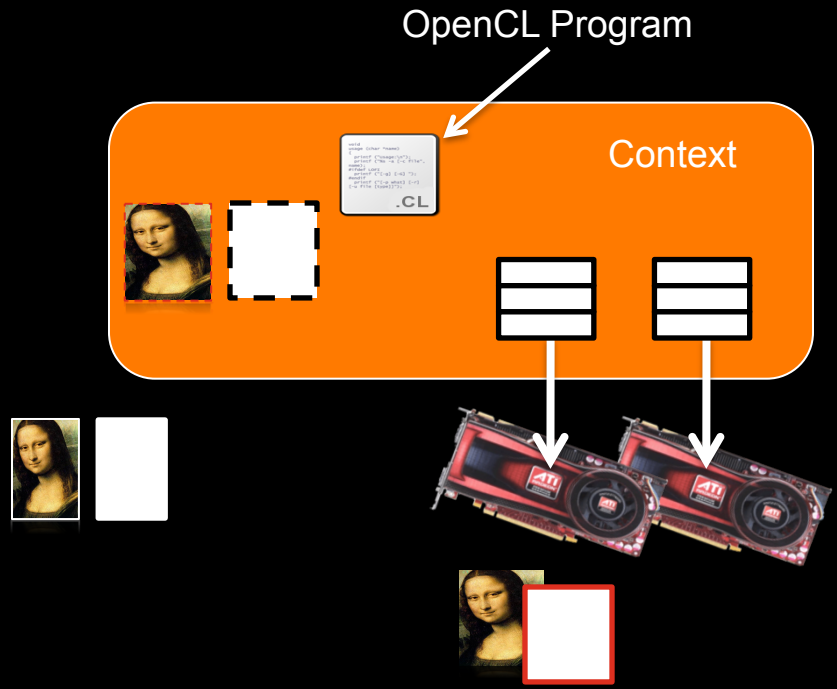
TRANSFERRING DATA

- This command initializes the OpenCL memory object and writes data to the device associated with the command queue
  - The command will write data 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
- Events can specify which commands should be completed before this one runs

```
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)
```

PROGRAMS AND KERNELS

- A program object is basically a collection of OpenCL kernels
  - Can be source code (text) or precompiled binary
  - Can also contain constant data and auxiliary functions
- Creating a program object requires either reading in a string (source code) or a precompiled binary
  - A program object is created by selecting which devices to target



CREATING A PROGRAM

Northeastern University

- This function creates a program object from strings of source code
  - *count* specifies the number of strings
  - The user must create a function to read in the source code to a string
  - Programmer can pass in compiler flags (optional)
- The *lengths* fields are used to specify the string lengths

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

BUILDING A PROGRAM

Northeastern University

- This function compiles and links an executable from the program object for each device in the context
  - Program is compiled for each device
  - If `device_list` is supplied, then only those devices are targeted
- Optional preprocessor, optimization, and other options can be supplied by the *options* argument

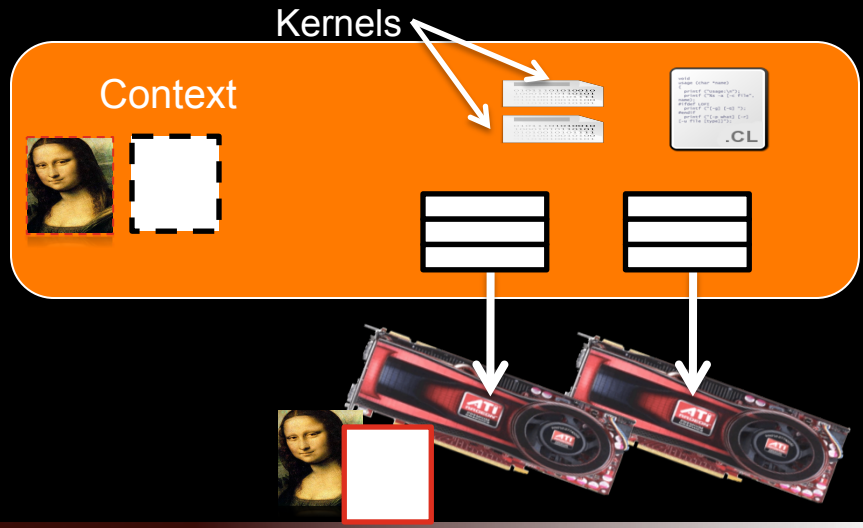
```
cl_int      clBuildProgram (cl_program program,
                            cl_uint num_devices,
                            const cl_device_id *device_list,
                            const char *options,
                            void (CL_CALLBACK *pfn_notify)(cl_program program,
          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

- Compilation failure is determined by an error value returned from `clBuildProgram()`
- clGetProgramBuildInfo() with the program object and the parameter `CL_PROGRAM_BUILD_STATUS` returns a string with the compiler output

CREATING A KERNEL

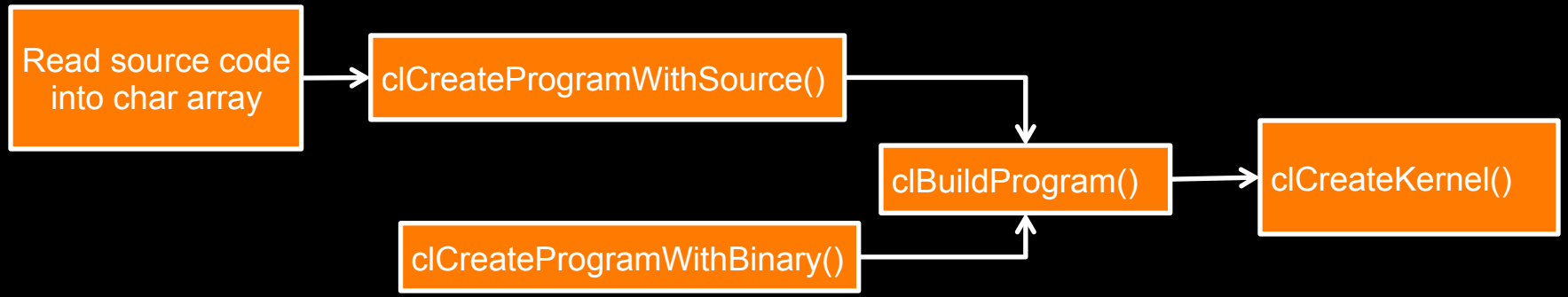
- A kernel is a function declared in a program that is executed on an OpenCL device
  - A kernel object is a kernel function along with its associated arguments
  - A kernel object is created from a compiled program object by specifying the name of the kernel function
  - The kernel is created is specified by a string that matches the name of the function within the program
- Must explicitly associate arguments (memory objects, primitives, etc.) with the kernel object

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



SUMMARIZING RUNTIME COMPILATION

- Runtime compilation is necessary due to the range of devices from different vendors
- There is a high overhead for compiling programs and creating kernels
  - Each operation only has to be performed once (at the beginning of the program)
  - The kernel objects can be reused any number of times by setting different arguments



PROGRAMMING MODEL

Northeastern University

- Data parallel
  - One-to-one mapping between work-items and elements in a memory object
  - Work-groups can be defined explicitly (like CUDA) or implicitly (specify the number of work-items and OpenCL creates the work-groups)
- Task parallel
  - Kernel is executed independent of an index space
  - Other ways to express parallelism: enqueueing multiple tasks to the device,

A SCALABLE THREAD STRUCTURE

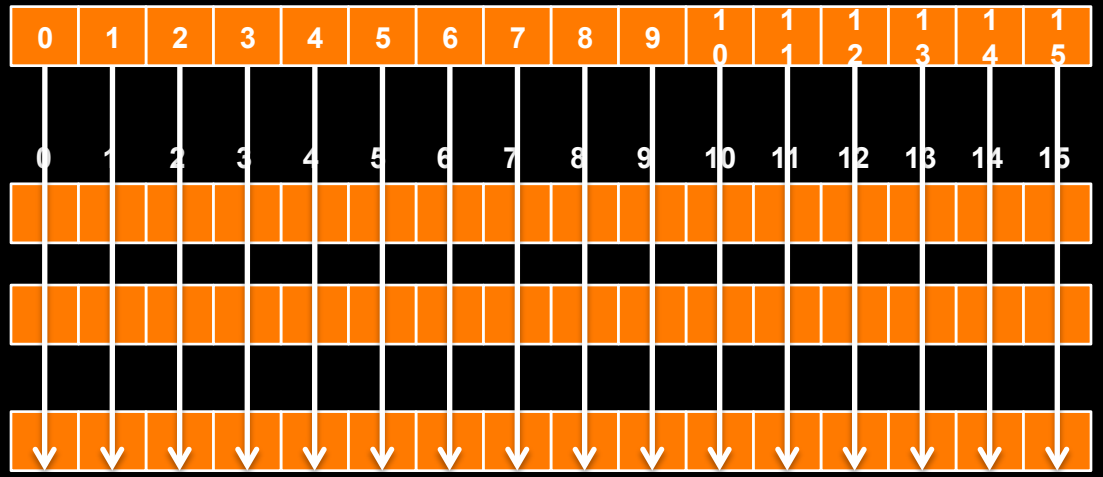
Northeastern University

- Each thread is responsible for adding the indices corresponding to its ID
- Each instance of a kernel is called a work-item (though “thread” is commonly used as well)
- Work-items are organized as work-groups
  - Work-groups are independent from one-another (this is where scalability comes from)

Thread structure:

Vector Addition:

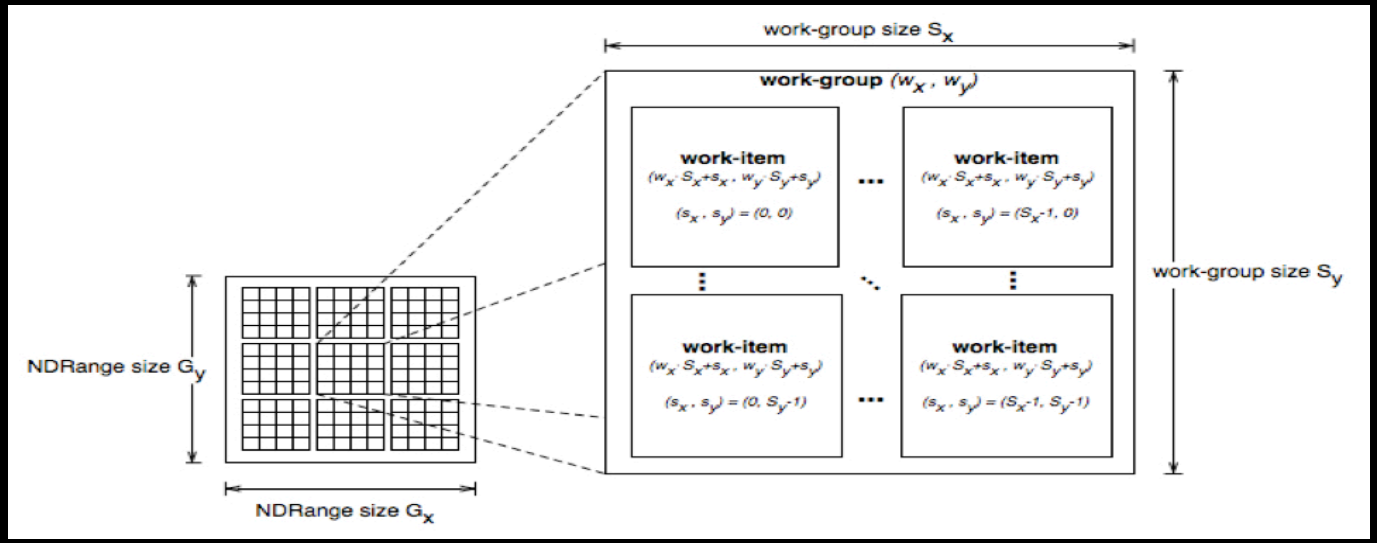
A
+
B
=
C



THREAD STRUCTURE

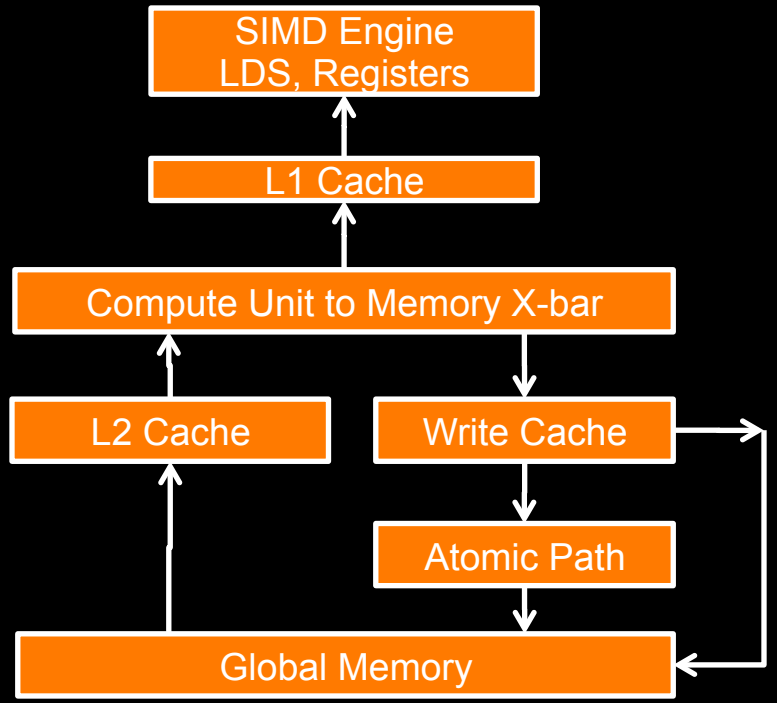
Northeastern University

- An index space defines a hierarchy of work-groups and work-items
- Work-items can uniquely identify themselves based on:
  - A global id (unique within the index space)
  - A work-group ID and a local ID within the work-group

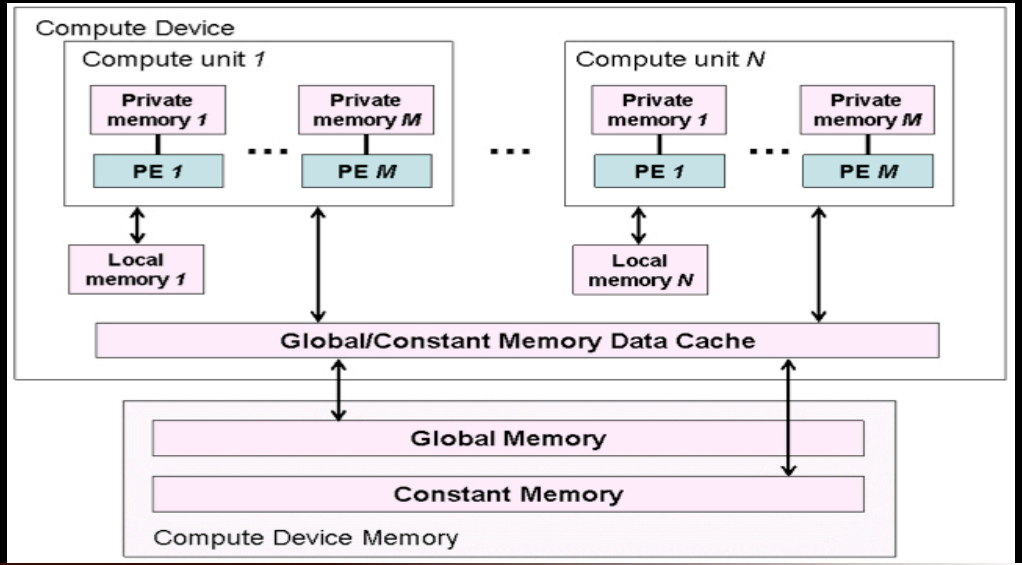


MEMORY MODEL

- The OpenCL memory model is closely related to a real GPU memory hierarchy



Memory	Description
Global	Accessible by all work-items
Constant	Read-only, global
Local	Local to a work-group
Private	Private to a work-item



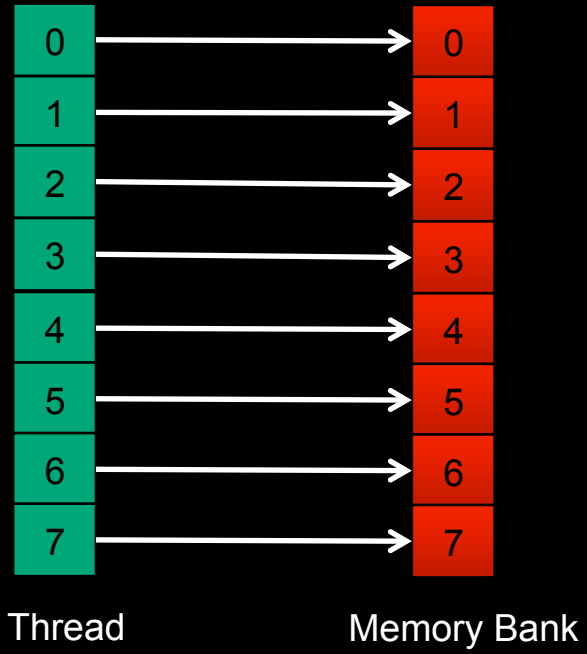
THE OPENCL KERNEL…FINALLY

- Memory Space Qualifiers
  - __global – memory allocated from global space
  - __constant – a special type of read-only memory
  - __local – memory shared by a work-group
  - __private – private per work-item memory
  - __read_only /__write_only – used for images
- Kernel arguments that are memory objects must be global, local, or constant
- Kernels execute asynchronously from the host
- Synchronization
  - Between items in a work-group
  - Between commands in a command queue

```
//Simple vector addition kernel:

__kernel
void vecadd(   __global int* A,
               __global int* B,
               __global int* C)
{
  int tid = get_global_id(0);
  C[tid] = A[tid] + B[tid];
}
```

MEMORY MODEL

- Memory management is explicit
  - Must move data from host to device global memory, from global memory to local memory, and back
- Work-groups are assigned to execute on compute-units
  - No guaranteed communication/coherency between different work-groups
- Memory is made up of *banks*
  - Memory banks are the hardware units that actually store data

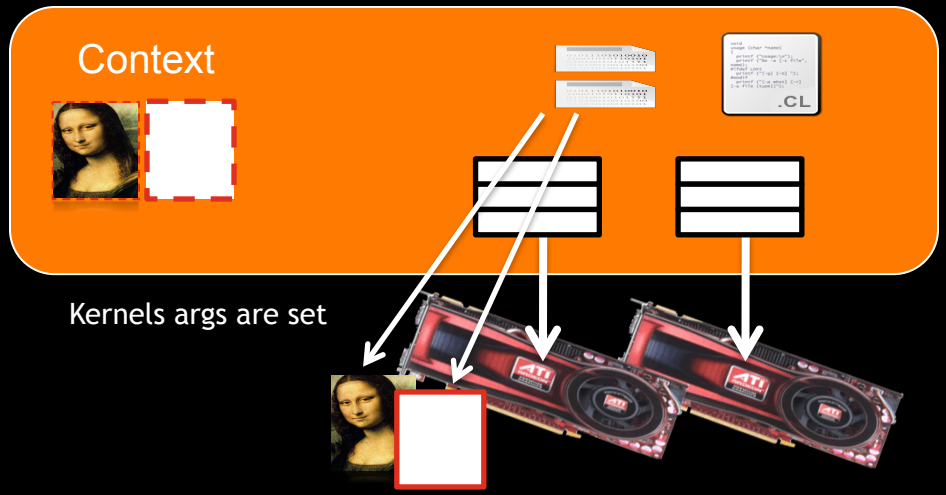


SETTING KERNEL ARGUMENTS

Northeastern University

- Each call provides the index of the argument as in the function signature, size, and a pointer to the data
- Examples:
  - clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_iImage);
  - clSetKernelArg(kernel, 1, sizeof(int), (void*)&a);
- CUDA avoids this by using a preprocessor

```
cl_int  clSetKernelArg (cl_kernel kernel,
                        cl_uint arg_index,
                        size_t arg_size,
                        const void *arg_value)
```

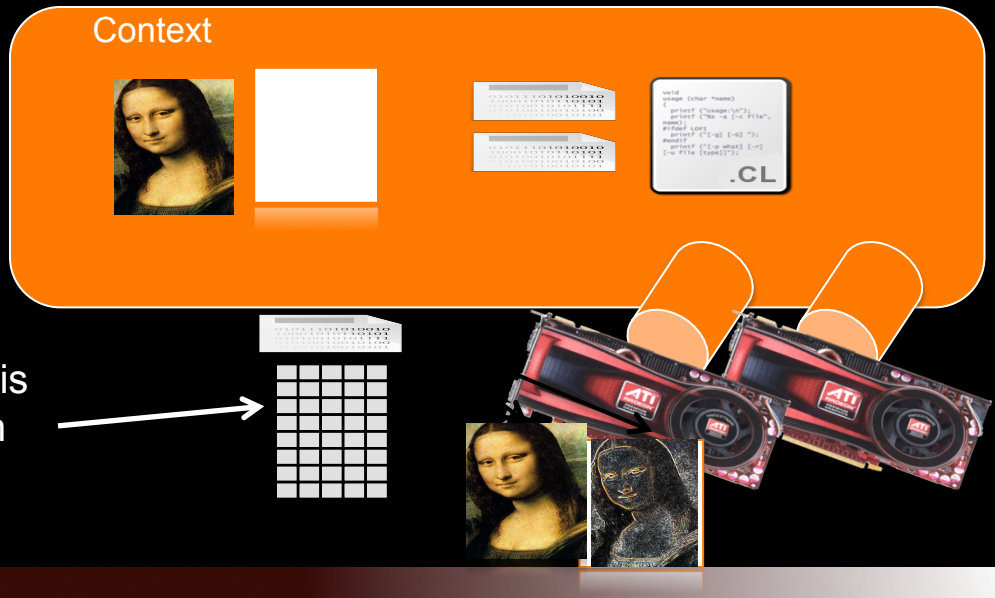


EXECUTING THE KERNEL

Northeastern University

- A thread structure defined by the index-space that is created
  - Each thread executes the same kernel on different data

```
cl_int  clEnqueueNDRangeKernel (cl_command_queue command_queue,
                                cl_kernel kernel,
                                cl_uint  work_dim,
                                const size_t *global_work_offset,
                                const size_t *global_work_size,
                                const size_t *local_work_size,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

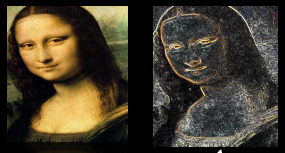


An index space of threads is created (dimensions match the data)

COPYING DATA BACK

- The last step is to copy the data back from the device to the host
- Similar call as writing a buffer to a device, but data will be transferred back to the host

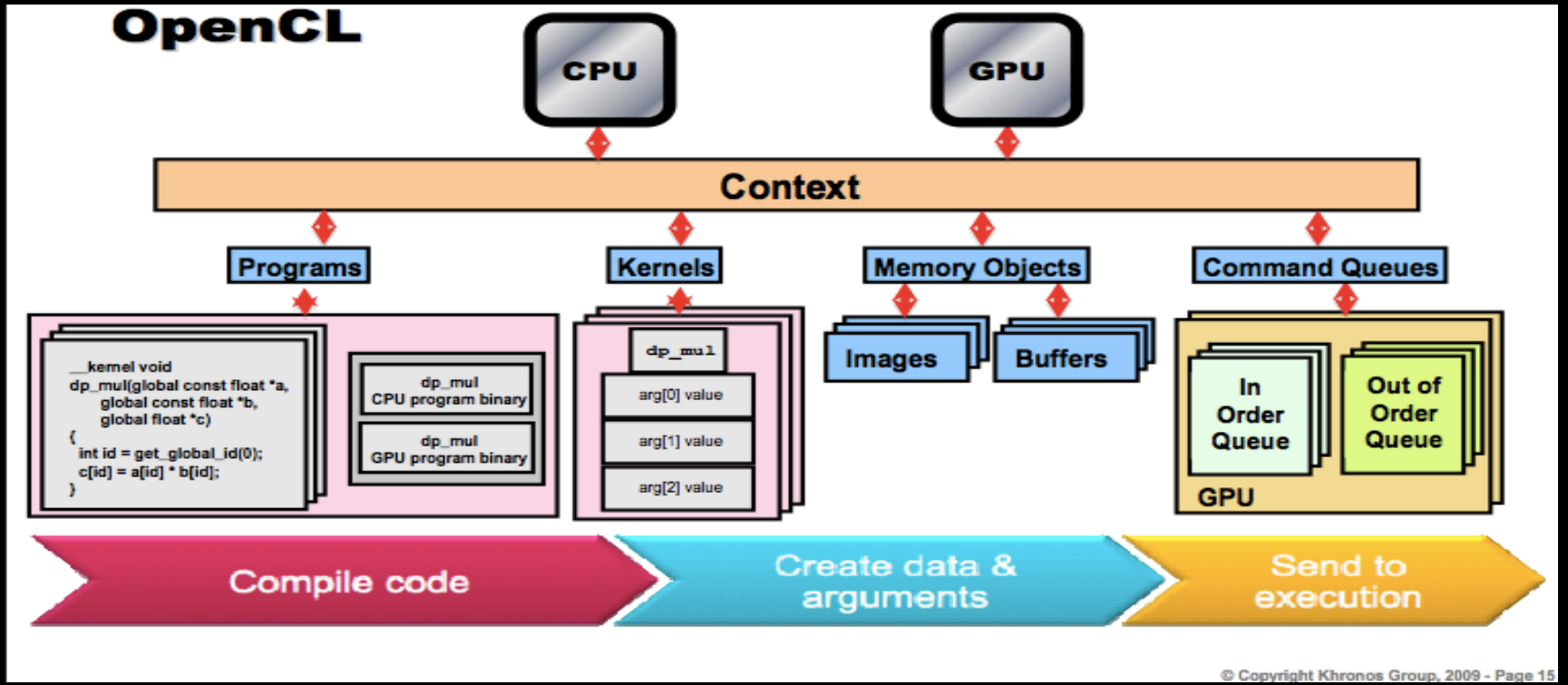
```
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_read,
                            size_t offset,
                            size_t cb,
                            void *ptr,
                     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



Copied back from GPU

BIG PICTURE

Northeastern University



EXAMPLE 1 - IMAGE ROTATION

- A common image processing routine
- Applications in matching, alignment, etc.
- New coordinates of point (x1,y1) when rotated by an angle Θ around (x0,y0)

$$x_2 = \cos(\theta) * (x_1 - x_0) - \sin(\theta) * (y_1 - y_0) + x_0$$
$$y_2 = \sin(\theta) * (x_1 - x_0) + \cos(\theta) * (y_1 - y_0) + x_0$$

- By rotating the image about the origin (0,0) we get

$$x_2 = \cos(\theta) * (x_1) - \sin(\theta) * (y_1)$$
$$y_2 = \sin(\theta) * (x_1) + \cos(\theta) * (y_1)$$

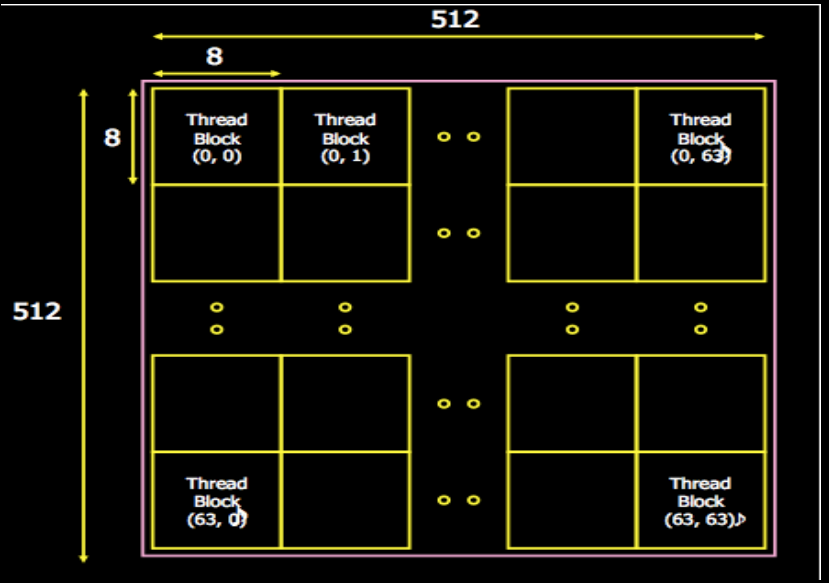
- Each coordinate for every point in the image can be calculated independently

Original Image

Rotated Image (90°)

IMAGE ROTATION

- Input: To copy to device
  - Image (2D Matrix of floats)
  - Rotation parameters
  - Image dimensions
- Output: From device
  - Rotated Image
- Main Steps
  - Copy image to device by enqueueing a write to a buffer on the device from the host
  - Run the Image rotation kernel on input image
  - Copy output image to host by enqueueing a read from a buffer on the device



THE OPENCL KERNEL

- Parallel portion of the algorithm off-loaded to device
  - Most thought provoking part of coding process
- Steps to be done in Image Rotation kernel
  - Obtain coordinates of work item in work group
  - Read rotation parameters
  - Calculate destination coordinates
  - Read input and write rotated output at calculated coordinates
- Parallel kernel is not always this obvious.
  - Profiling is often necessary to find the bottlenecks and locate the data parallelism
  - In this example grid of output image decomposed into work items
  - Not all parts of the input image copied to the output image after rotation, corners of I/P image could be lost after rotation

OPENCL KERNEL

Northeastern University

```
__kernel  void image_rotate(
   __global float * src_data, __global float * dest_data, //Data in global memory
    int W,    int H,                                      //Image Dimensions
     float sinTheta, float cosTheta )                     //Rotation Parameters
{
  //Thread gets its index within index space
  const int ix = get_global_id(0);
  const int iy = get_global_id(1);

  //Calculate location of data to move into ix and iy – Output decomposition as mentioned
  float xpos = (  ((float) ix)*cosTheta + ((float)iy )*sinTheta);
  float ypos = (  ((float) iy)*cosTheta -  ((float)ix)*sinTheta);

  if (( ((int)xpos>=0) && ((int)xpos< W)))  //Bound Checking
                                  && (((int)ypos>=0) && ((int)ypos< H)))
  {
      //Read (xpos,ypos) src_data and store at (ix,iy) in dest_data
      dest_data[iy*W+ix] =  src_data[(int)(floor(ypos*W+xpos))];
   }
}
```

STEP0: INITIALIZE DEVICE

- Declare context

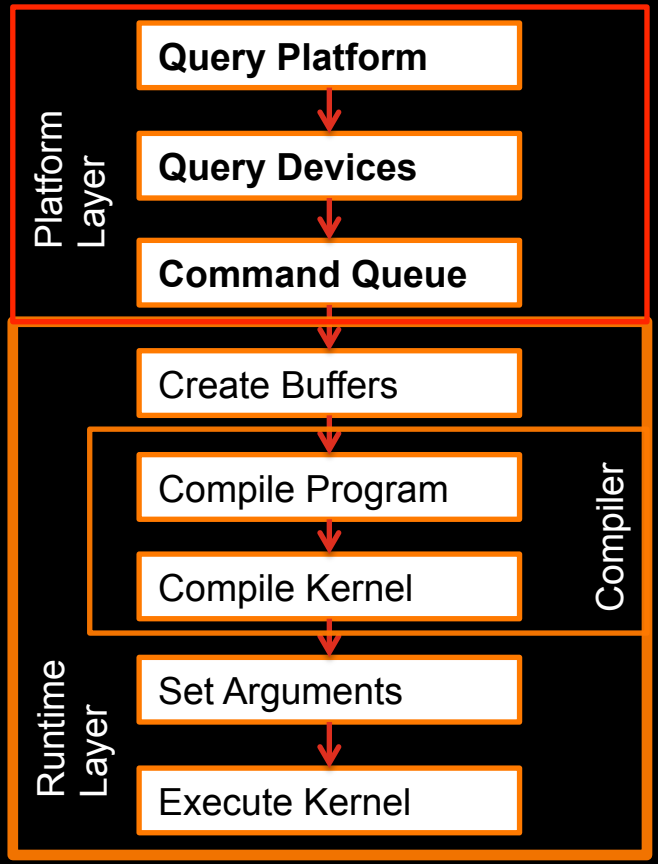
```
cl_context myctx = clCreateContextFromType(
          0, CL_DEVICE_TYPE_GPU,
          NULL, NULL, &ciErrNum);
```

- Choose a device from context

```
ciErrNum = clGetDeviceIDs(
         0,CL_DEVICE_TYPE_GPU,
         1, &device,  cl_uint *num_devices)
```

- Using device and context create a command queue

```
cl_command_queue myqueue ;
myqueue = clCreateCommandQueue(
         myctx,
         device, 0, &ciErrNum);
```



STEP1: CREATE BUFFERS

Northeastern University

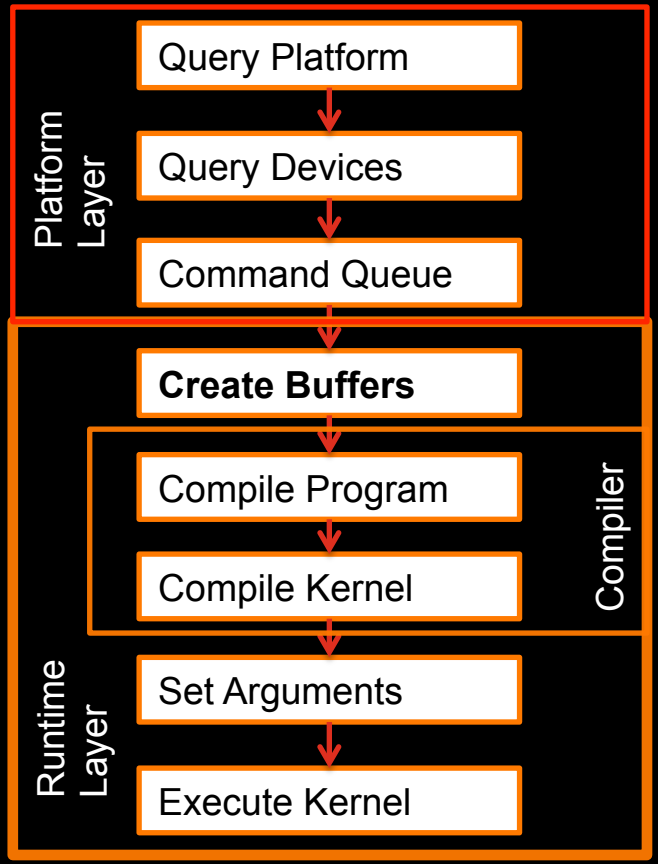
- Create buffers on device
- Input data is read-only
- Output data is write-only

```
cl_mem d_ip = clCreateBuffer( myctx,
        CL_MEM_READ_ONLY, mem_size,  NULL,
        &ciErrNum);
```

```
cl_mem d_op = clCreateBuffer( myctx,
        CL_MEM_WRITE_ONLY, mem_size,  NULL,
        &ciErrNum);
```

- Transfer input data to the devicea

```
ciErrNum = clEnqueueWriteBuffer(
        myqueue , d_ip, CL_TRUE,
        0,mem_size, (void *)src_image,
        0, NULL,  NULL)
```



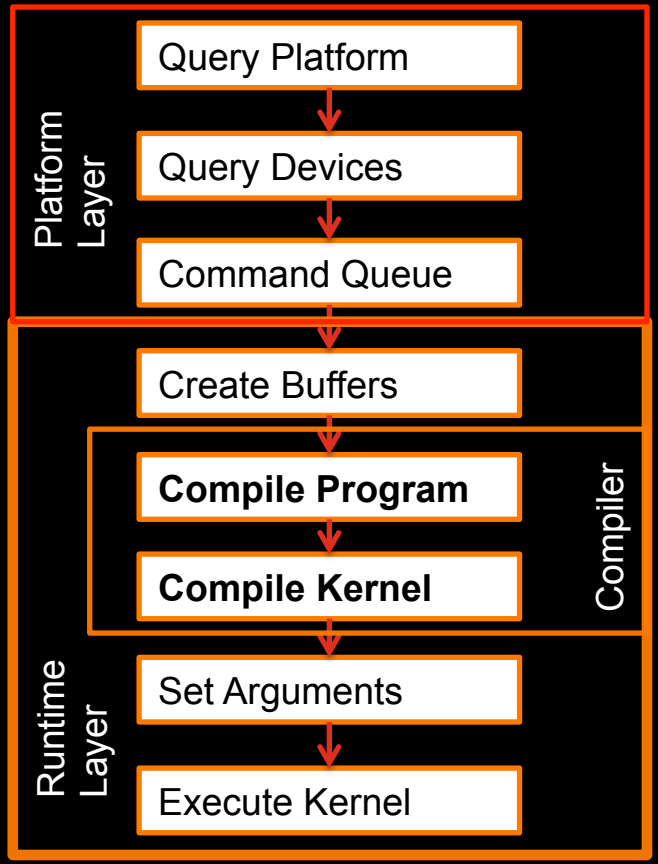
STEP2: BUILD PROGRAM, SELECT KERNEL

Northeastern University

```
// Create the program
cl_program myprog= clCreateProgramWithSource
        ( myctx,1, (const char **)&source,
        &program_length, &ciErrNum);
```

```
// Build the program
ciErrNum = clBuildProgram( myprog, 0,  NULL,
        NULL, NULL, NULL);
```

```
// The "image_rotate" function as the kernel
cl_kernel mykernel = clCreateKerne( myprog ,
        "image_rotate" , error_code)
```



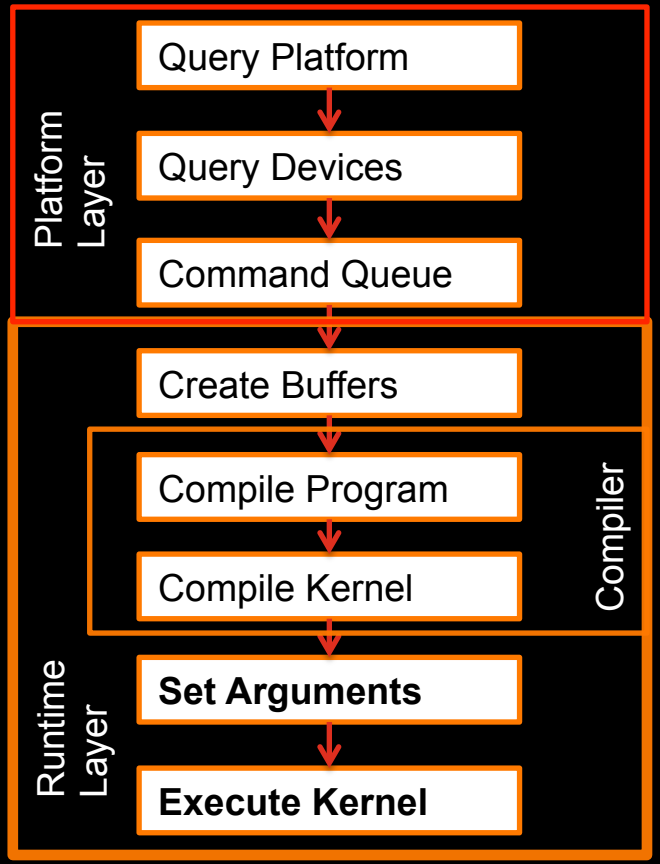
STEP3: SET ARGUMENTS, ENQUEUE KERNEL

Northeastern University

```
// Set Arguments
clSetKernelArg(mykernel, 0, sizeof(cl_mem),
          (void *)&d_ip);
clSetKernelArg(mykernel, 1, sizeof(cl_mem),
          (void *)&d_op);
 clSetKernelArg(mykernel, 2, sizeof(cl_int),
          (void *)&W);
```

```
//Set local and global workgroup sizes
size_t localws[2] = {16,16} ;
//Assume divisible by 16
size_t globalws[2]={W, H};
```

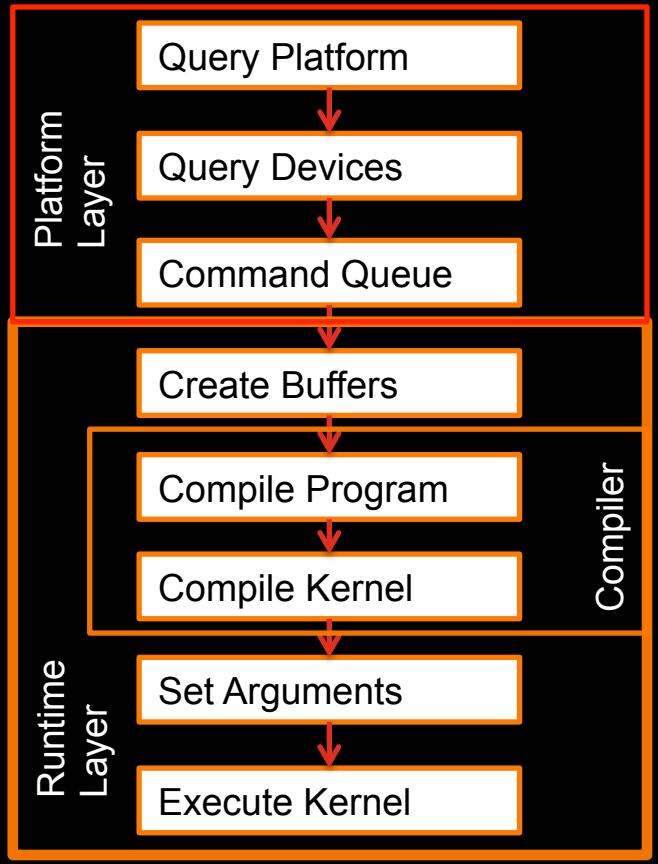
```
// execute kernel
clEnqueueNDRangeKernel(  myqueue , myKernel,
          2, 0, globalws, localws,
          0, NULL, NULL);
```



STEP4: READ BACK RESULT

- Only necessary for data required on the host
- Data output from one kernel can be reused for another kernel
- Avoid redundant host-device IO

```
// copy results from device back to host
clEnqueueReadBuffer(
        myctx, d_op,
        CL_TRUE, //Blocking Read Back
        0, mem_size,(void *) op_data,
        NULL, NULL, NULL);
```



OPENCL TIMING

- OpenCL provides “events” which can be used for timing kernels
- We pass an event to the OpenCL enqueue kernel function to capture timestamps
- Code snippet provided can be used to time a kernel
- Add profiling enable flag to create command queue
- By taking differences of the start and end timestamps we discount overheads like time spent in the command queue

```
cl_event event_timer;
clEnqueueNDRangeKernel(myqueue ,
          myKernel,2, 0, globalws, localws,
          0, NULL, &event_timer);
```

```
unsigned long starttime, endtime;
```

```
clGetEventProfilingInfo( event_time,
          CL_PROFILING_COMMAND_START,
          sizeof(cl_ulong),&starttime, NULL);
```

```
clGetEventProfilingInfo(event_time,
          CL_PROFILING_COMMAND_END,
          sizeof(cl_ulong), &endtime, NULL);
```

```
unsigned long elapsed =
(unsigned long)(endtime - starttime);
```

OPENCL OPTIMIZATION - THREAD MAPPING

Northeastern University

- Thread mapping determines which threads will access which data
  - Proper mappings can align with hardware and improve performance
  - Improper mappings can be disastrous to performance
- Using mappings, the same thread can be assigned to access different data elements
  - Examples below show three different possible mappings of threads to data (assuming the thread id is used to access an element)

Mapping

```
int tid =
get_global_id(1) *
get_global_size(0) +
get_global_id(0);
```

Thread IDs

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
int tid =
get_global_id(0) *
get_global_size(1) +
get_global_id(1);
```

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

```
int group_size =
get_local_size(0) *
get_local_size(1);

int tid =
get_group_id(1) *
get_num_groups(0) *
group_size +
get_group_id(0) *
group_size +
get_local_id(1) *
get_local_size(0) +
get_local_id(0)
```

0	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

*assuming 2x2 groups

THREAD MAPPING

- Consider a serial matrix multiplication algorithm

```
for (i1=0; i1 < M; i1++)
    for (i2=0; i2< N; i2++)
        for (i3=0; i3< P; i3++)
            C[i1][i2] += A[i1][i3]*B[i3][i2];
```

- This algorithm is suited for output data decomposition
  - We will create *NM* threads - effectively removing the outer two loops
  - Each thread will perform *P* calculations - The inner loop will remain as part of the kernel
- Should the index space be MxN or NxM?

THREAD MAPPING

- Thread mapping 1: with an MxN index space, the kernel would be:

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

- Thread mapping 2: with an NxM index space, the kernel would be:

```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

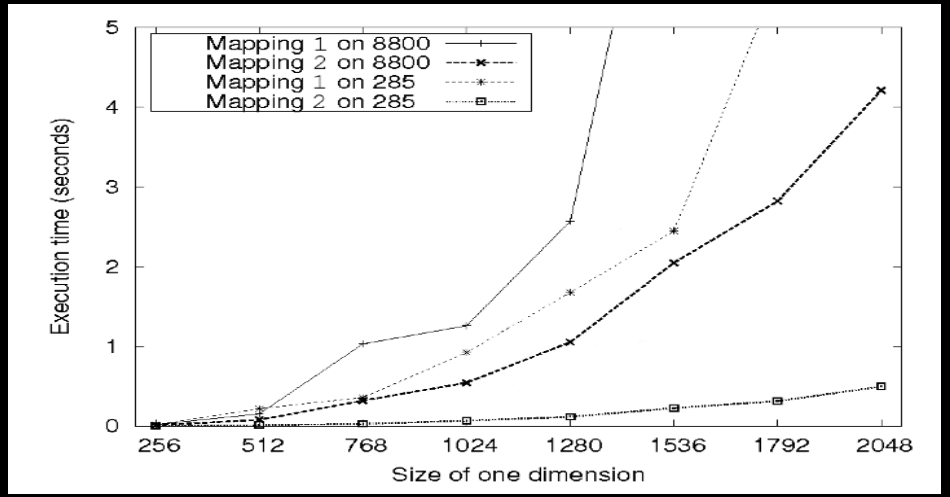
- Both mappings produce functionally equivalent versions of the program

THREAD MAPPING

Northeastern University

- This figure shows the execution of the two thread mappings on NVIDIA GeForce 285 and 8800 GPUs

- Notice that mapping 2 is far superior in performance for both GPUs



SO FAR

Northeastern University

- We know how to optimize a program in OpenCL by taking advantage of the underlying architecture
- We have seen how to utilize threads to hide latency
- We have also seen how to take advantage of the different memory spaces available in today's GPUs.

N-BODY ALGORITHMS

- The gravitational attraction between two bodies in space is an example of an N-body problem
  - Body represents a galaxy / star, and bodies attract each other through gravitational force
  - Bodies attract each other through force (F)
- $O(N^2)$ algorithm: N*N interactions need to be calculated
- All-pairs technique is used to calculate close-field forces

$$F = G * \left(\frac{m_i * m_j}{|| r_{ij} ||^2} \right) * \frac{r_{ij}}{|| r_{ij} ||}$$

F = Resultant Force Vector between particles i and j
G = Gravitational Constant
m_i = Mass of particle i
m_j = Mass of particle j
r_{ij} = Distance of particle i and j

For each particle this becomes

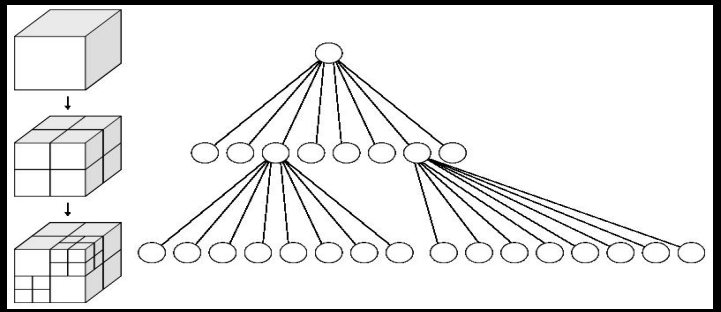
$$F_i = (G * m_i) * \sum_{j=1 \to N} \left(\frac{m_j}{|| r_{ij} ||^2} * \left(\frac{r_{ij}}{|| r_{ij} ||} \right) \right)$$

```
for(i=0; i<n; i++) {
    ax = ay = az = 0;
   // Loop over all particles "j"
    for (j=0; j<n; j++)  {

                //Calculate Displacement
                dx=x[j]-x[i];
                dy=y[j]-y[i];
                dz=z[j]-z[i];

               // small eps is delta added for dx,dy,dz = 0
               invr= 1.0/sqrt(dx*dx+dy*dy+dz*dz +eps);
               invr3 = invr*invr*invr;
               f=m[ j ]*invr3;

               // Accumulate acceleration
               ax += f*dx;
               ay += f*dy;
               az += f*dx;
    }
   // Use ax, ay, az to update particle positions
}
```

N-BODY ALGORITHMS

- For large counts, the N^2 method calculates of force contribution of distant particles
  - Distant particles hardly affect resultant force
- Algorithms like Barnes Hut reduce number of particle interactions calculated
  - Nearby cells treated individually
  - Distant cells treated as a single large particle
- We restrict ourselves to a simple all pair simulation of particles with gravitational forces
  - Near field still uses all pairs
  - So, implementing all pairs improves both near and far field calculations

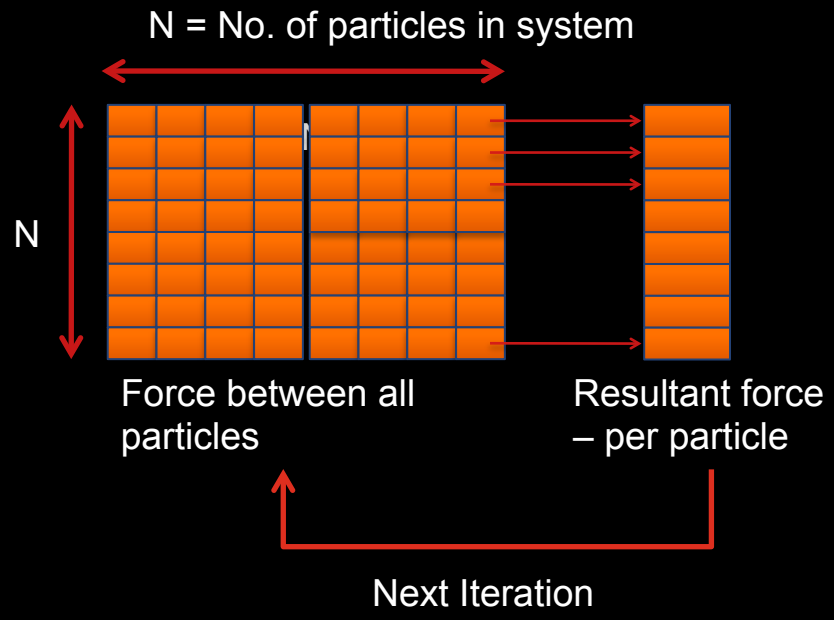
- Volume divided into cubic cells in an octree
- A octree is a tree where a node has exactly 8 children
- Used to subdivide a 3D space



PARALLEL IMPLEMENTATION

Northeastern University

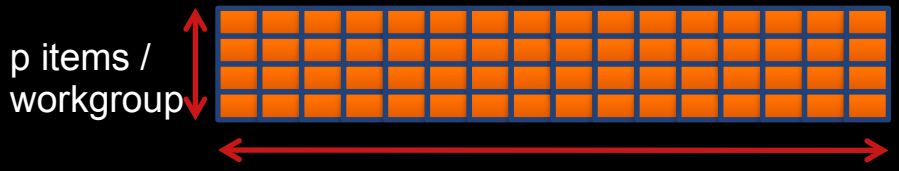
- Embarrassingly parallel algorithm
- Forces of each particle can be computed independently
- Accumulate results in local memory
- Add accumulated results to previous position of particles
- New position used as input to the next time step to calculate new forces acting between particles



NAÏVE PARALLEL IMPLEMENTATION

Northeastern University

- Disadvantages of implementation where each work item reads data independently
- No reuse since redundant reads of parameters for multiple work-items
- Memory access= N reads*N threads= N2
- Similar to naïve non blocking matrix multiplication



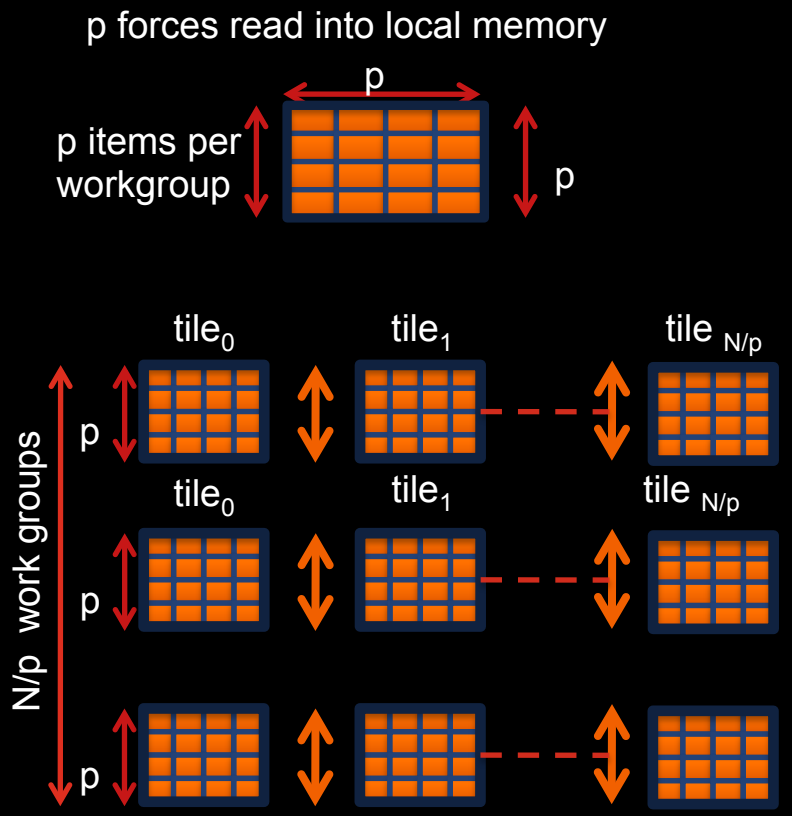
N = No. of particles
All N particles read in by each work item

```
__kernel void nbody(
                __global float4 * initial_pos,
                __global float4 * final_pos,
                Int N, __local float4 * result) {

   int localid = get_local_id(0);
   int globalid = get_global_id(0);
   result [localid] = 0;

   for( int i=0 ; i<N;i++)  {
    //! Calculate interaction between
    //! particle globalid and particle i
    GetForce( globalid, i, initial_pos, final_pos,
                &result [localid]) ;
    }
    finalpos[ globalid] = result[ localid];
}
```

LOCAL MEMORY OPTIMIZATIONS

Northeastern University

- Data Reuse
  - Any particle read into compute unit can be used by all p bodies
- Computational tile:
  - Square region of the grid of forces consisting of size p
  - 2p forces required to evaluate all p^2 interactions in tile
- p work items (in vertical direction) read in p forces
- Interactions on p bodies captured as an update to p acceleration vectors
- Intra-work group synchronization shown in orange required since all work items use data read by each work item



OPENCL IMPLEMENTATION

- Data reuse using local memory
  - Without reuse N*p items read per work group
  - With reuse p*(N/p) = N items read per work group
- All work items use data read in by each work item
  - SIGNIFICANT improvement: p is work group size (at least 128 in OpenCL, discussed in occupancy)
  - Loop nest shows how a work item traverses all tiles
  - Inner loop accumulates contribution of all particles within tile

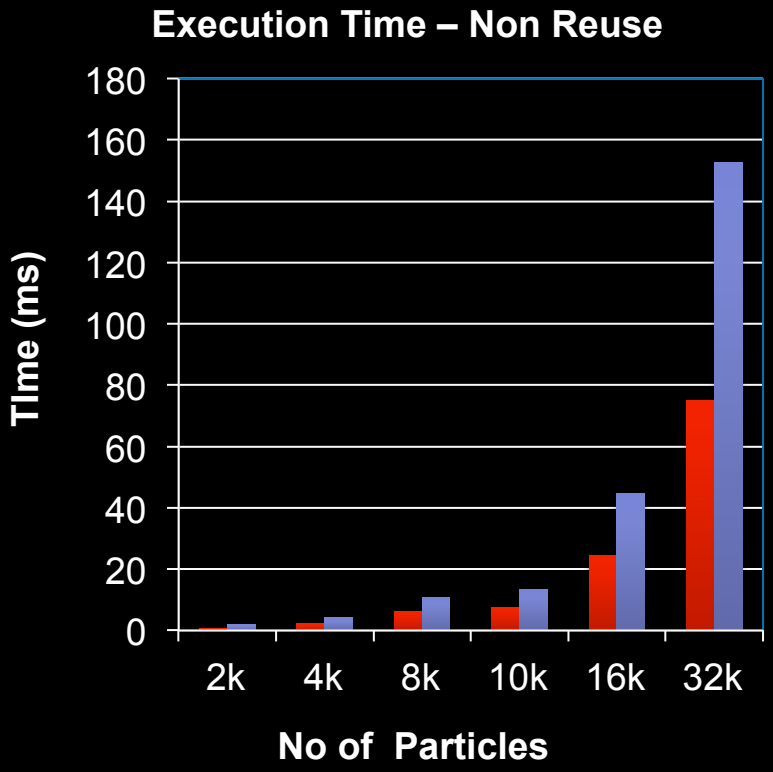
```
for (int i = 0; i < numTiles; ++i)  {
      // load one tile into local memory
      int idx = i * localSize + tid;
      localPos[tid] = pos[idx];

     barrier(CLK_LOCAL_MEM_FENCE);

     // calculate acceleration effect due to each body
    for( int j = 0; j < localSize; ++j )    {
        // Calculate acceleration caused by particle j on i
        float4 r = localPos[j] – myPos;

        float distSqr = r.x * r.x  +  r.y * r.y  +  r.z * r.z;
        float invDist = 1.0f / sqrt(distSqr + epsSqr);
                float s = localPos[j].w * invDistCube;

        // accumulate effect of all particles
        acc += s * r;
      }
      // Synchronize so that next tile can be loaded
      barrier(CLK_LOCAL_MEM_FENCE);
   }
}
```

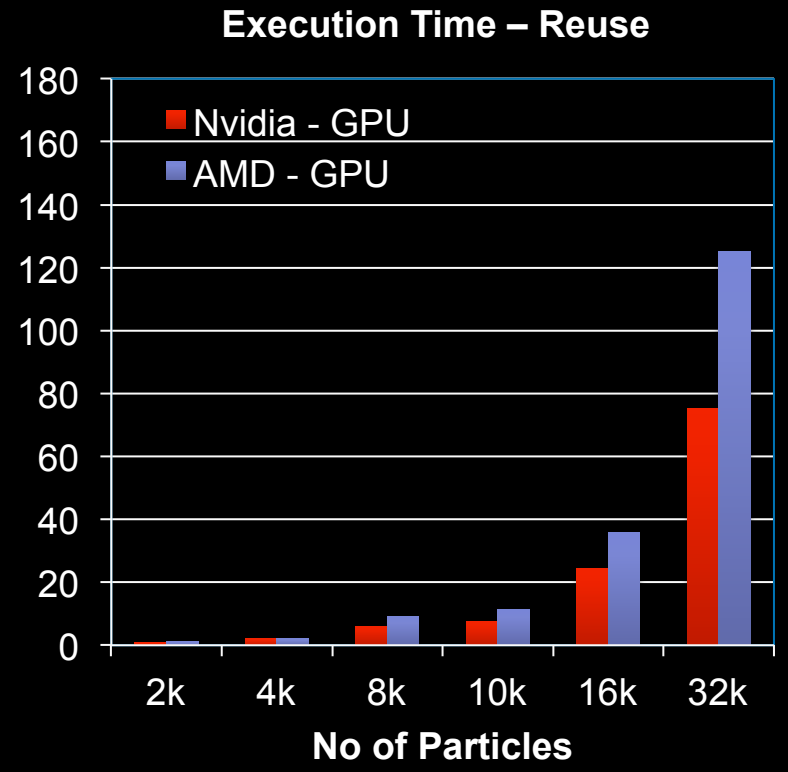
PERFORMANCE

- Effect of optimizations compared for two GPU platforms
  - **Exactly same** code, only recompiled for platform
- Devices Compared
  - AMD GPU 5870
  - Nvidia GPU GTX 480
- Time measured for OpenCL kernel using OpenCL event counters
  - Device IO and other overheads like compilation time are not relevant to our discussion of optimizing a compute kernel
  - Events are provided in the OpenCL spec to query obtain timestamps for different state of OpenCL commands

EFFECT OF REUSE ON KERNEL PERFORMANCE

Northeastern University



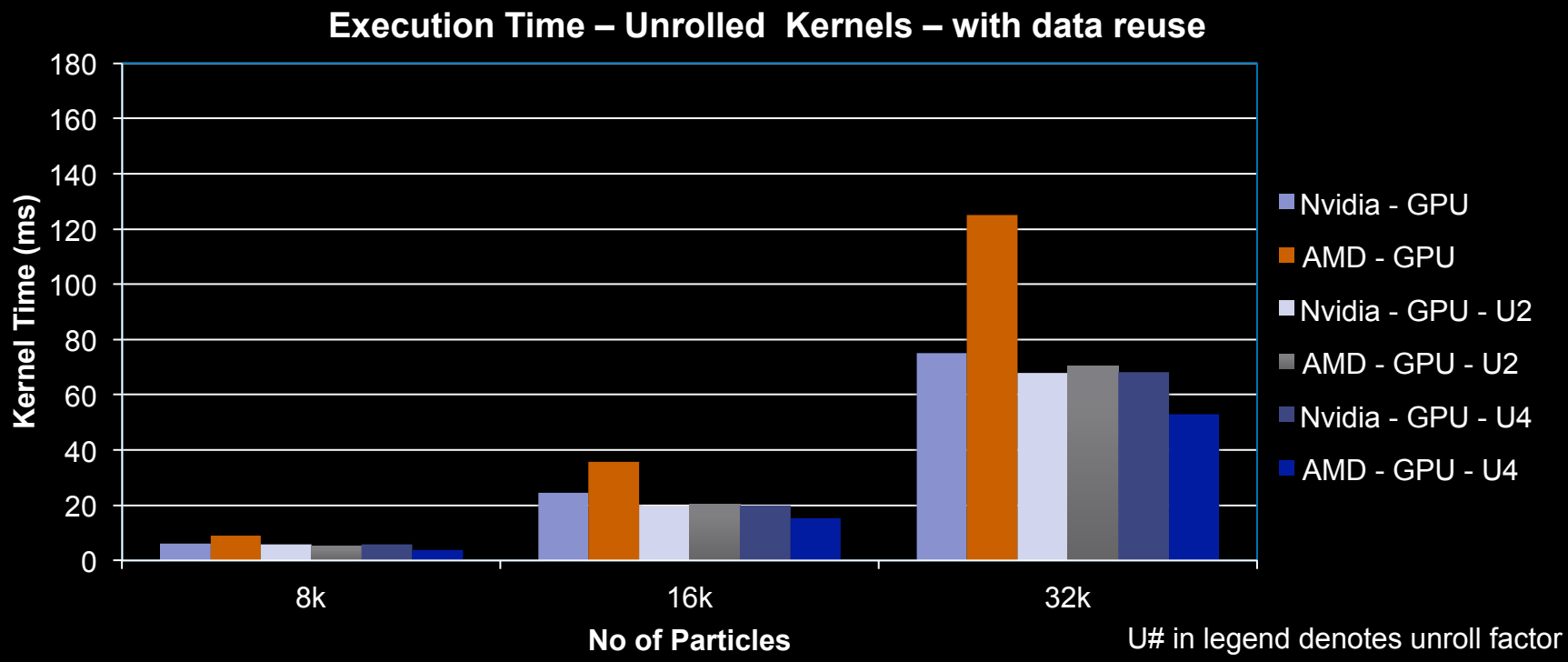


PERFORMANCE - LOOP UNROLLING

- We also attempt loop unrolling of the reuse local memory implementation
  - We unroll the innermost loop within the thread
- Loop unrolling can be used to improve performance by removing overhead of branching
  - Beneficial only for tight loops where branching overhead is comparable to the size of the loop body
  - Experiment on optimized local memory implementation
  - Executable size is not a concern for GPU kernels
- We implement unrolling by factors of 2 and 4 and we see substantial performance gains across platforms
  - Decreasing returns for larger unrolling factors seen

EFFECT OF UNROLL ON KERNEL PERFORMANCE

Northeastern University



OPENCL C++ API

- C++ Bindings provide
  - Abstractions
  - Object oriented progamming
  - Templates
- Lightweight, providing access to the low-level features of the original OpenCL™ C API
- Compatible with standard C++ compilers (GCC 4.x and VS 2008)

```
int main(void) {
try {
  cl::Context context (CL_DEVICE_TYPE_GPU, 0, NULL, NULL, &err);
  cl::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
  cl::Program::Sources source(1, std::make_pair(helloStr,strlen(helloStr)));
  cl::Program program_ = cl::Program(context, source);
  program_.build(devices);
  cl::Kernel kernel(program_, "hello", &err);
  cl::CommandQueue queue(context, devices[0], 0, &err);
  cl::KernelFunctor func = kernel.bind(queue, cl::NDRange(4, 4),
  cl::NDRange(2, 2));
  func().wait();
}
catch (cl::Error err) {
  std::cerr << "ERROR: " << err.what() << "(" << err.err() << ")“ <<
  std::endl;
}
return EXIT
}
```

EXTENSIONS TO OPENCL

- An OpenCL Extension is a feature, which might be supported by a device but is not a part of the OpenCL specification
  - Extensions allow vendors to expose device specific features without being concerned about compatibility with specification and other vendor features
- Check `clGetDeviceInfo` with `CL_DEVICE_EXTENSIONS`

```
#pragma OPENCL EXTENSION extension_name : enable
```

- Atomic functions to global and local memory
  - add, sub, xchg, inc, dec, cmp_xchg, min, max, and, or, xor
  - 32-bit/64-bit integers
- Byte Addressable Stores
- Device Fission – Allows splitting up a compute device into multiple subdevices
- Media operations

APPROACHES TO MULTIPLE DEVICES

- Single context, multiple devices
  - Standard way to work with multiple devices in OpenCL
  - Associating specific devices with a context is done by passing a list of the desired devices to `clCreateContext()`
  - The call `clCreateContextFromType()` takes a device type (or combination of types) as a parameter and creates a context with all devices of that type:
- Multiple contexts, multiple devices - Computing on a cluster, multiple systems, etc.

cl_device_type	Description
CL_DEVICE_TYPE_CPU	An OpenCL device that is the host processor. The host processor runs the OpenCL implementations and is a single or multi-core CPU.
CL_DEVICE_TYPE_GPU	An OpenCL device that is a GPU. By this we mean that the device can also be used to accelerate a 3D API such as OpenGL or DirectX.
CL_DEVICE_TYPE_ACCELERATOR	Dedicated OpenCL accelerators (for example the IBM CELL Blade). These devices communicate with the host processor using a peripheral interconnect such as PCIe.
CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the system.
CL_DEVICE_TYPE_ALL	All OpenCL devices available in the system.

SINGLE CONTEXT, MULTIPLE DEVICES

Northeastern University

- When multiple devices are part of the same context, most OpenCL objects are shared
  - Memory objects, programs, kernels, etc.
- One command queue must exist per device and is supplied in OpenCL when the target GPU needs to be specified
  - Any clEnqueue* function takes a command queue as an argument

SINGLE CONTEXT, MULTIPLE DEVICES

- Memory objects are common to a context, they must be explicitly written to a device before being used
  - Whether or not the same object can be valid on multiple devices is vendor specific
- OpenCL does not assume that data can be transferred directly between devices, so commands only exists to move from a host to device, or device to host
  - Copying from one device to another requires an intermediate transfer to the host

SINGLE CONTEXT, MULTIPLE DEVICES

- The behavior of a memory object written to multiple devices is vendor-specific
- OpenCL does not define if a copy of the object is made or whether the object remains valid once written to a device
- A CPU would operate on a memory object in-place, while a GPU would make a copy (so the original would still be valid until it is explicitly written over)
- AMD/NVIDIA implementations allow an object to be copied to multiple devices
  - Programmer responsible for maintaining updated copy and merging data

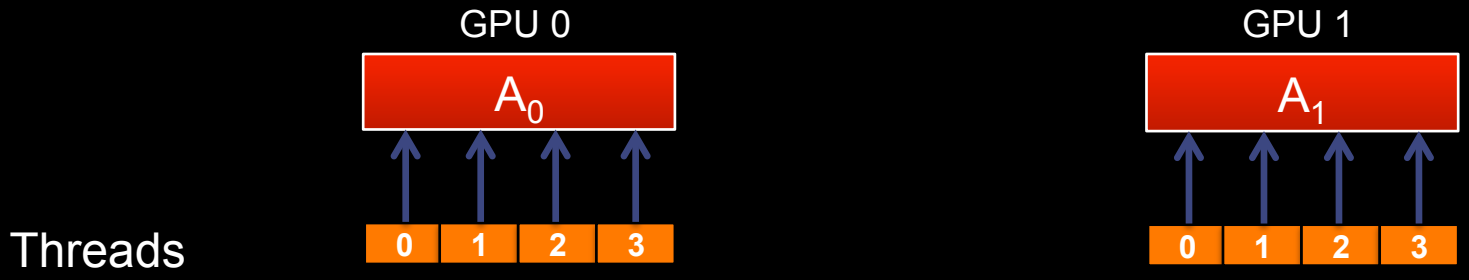
When writing data to a GPU, a copy is made, so multiple writes are valid

SINGLE CONTEXT, MULTIPLE DEVICES

- Just like writing a multi-threaded CPU program, we have two choices for designing multi-GPU programs

1. Redundantly copy all data and index using global offsets

2. Split the data into subsets and index into the subset



SINGLE CONTEXT, MULTIPLE DEVICES

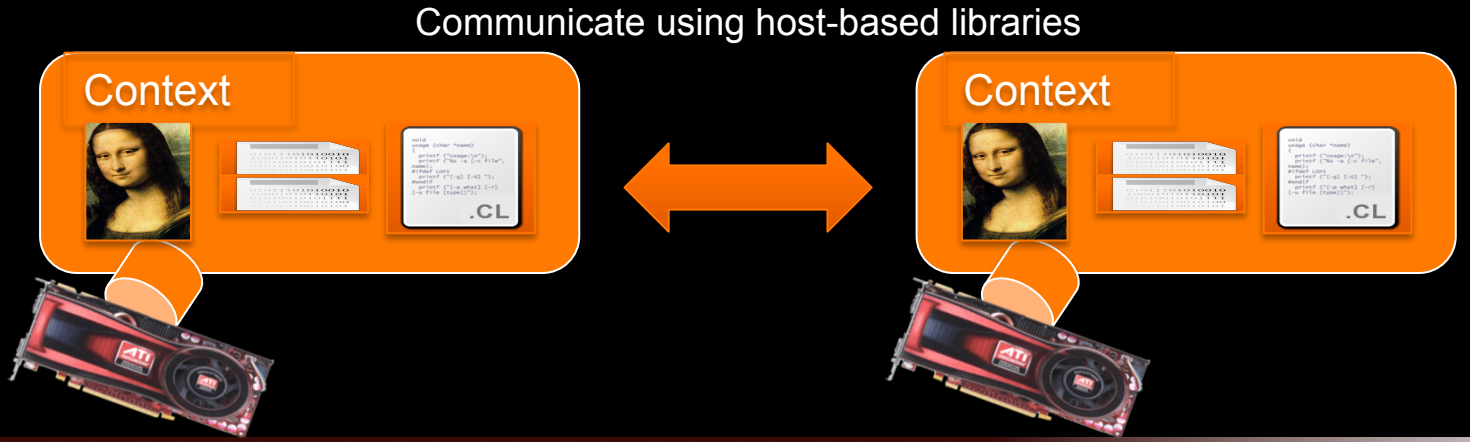
- OpenCL provides mechanisms to help with both multi-device techniques
  - `clEnqueueNDRangeKernel()` optionally takes offsets that are used when computing the global ID of a thread
    - Note that for this technique to work, any objects that are written to will have to be synchronized manually
  - *SubBuffers* were introduced in OpenCL 1.1 to allow a buffer to be split into multiple objects
    - This allows reading/writing to offsets within a buffer to avoid manually splitting and recombining data
- OpenCL *events* are used to synchronize execution on different devices within a context
- `clEnqueue*` function generates an event that identifies the operation
- `clEnqueue*` functions also take an optional list of events that must complete before that operation should occur
- `clEnqueueWaitForEvents()` is the specific call to wait for a list of events to complete

SYNCHRONIZATION BETWEEN COMMAND QUEUES

- individual queue can execute in order or out of order
  - In-order queue, all commands execute in order
  - Behaves as expected (as long as you're enqueuing from one thread)
- Multiple Queues
  - You must explicitly synchronize between queues
  - Multiple devices each have their own queue
- Use events to synchronize
- clWaitForEvents(num_events, *event_list)
  - Blocks until events are complete
- clEnqueueMarker(queue, *event)
  - Returns an event for a marker that moves through the queue
- clEnqueueWaitForEvents(queue, num_events, *event_list)
  - Inserts a "WaitForEvents" into the queue

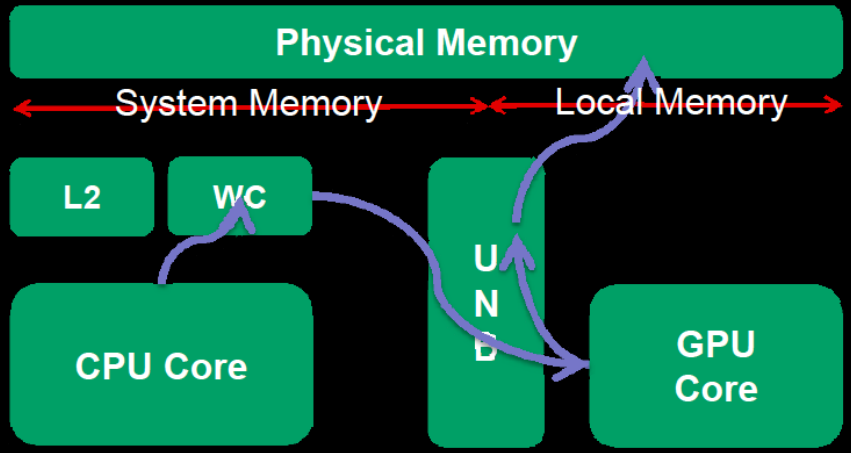
MULTIPLE CONTEXTS, MULTIPLE DEVICES

- An alternative approach is to create multiple OpenCL contexts (with associated objects) per device
- Distributed programming
  - If a framework such as MPI is used for communication, programs can be ran on multi-device machines or in distributed environments
  - Host libraries (e.g., pthreads, MPI) must be used for synchronization and communication
- In addition to PCI-Express transfers required to move data between host and device, extra memory and network communication may be required



PROGRAMMING SHARED MEMORY HETEROGENEOUS DEVICES

- Memory space subdivided into “System Memory” and “Local Memory”
  - Local memory: regions optimized for high bandwidth GPU accesses, driver managed
- Unified North Bridge arbitrates access
- clCreateBuffer calls allocate memory in either “System Memory” or “Local Memory”
  - Memory region defined by cl_mem_flags parameter
- Llano APUs provides different performance for each device based on regions used
  - To access Local memory from CPU

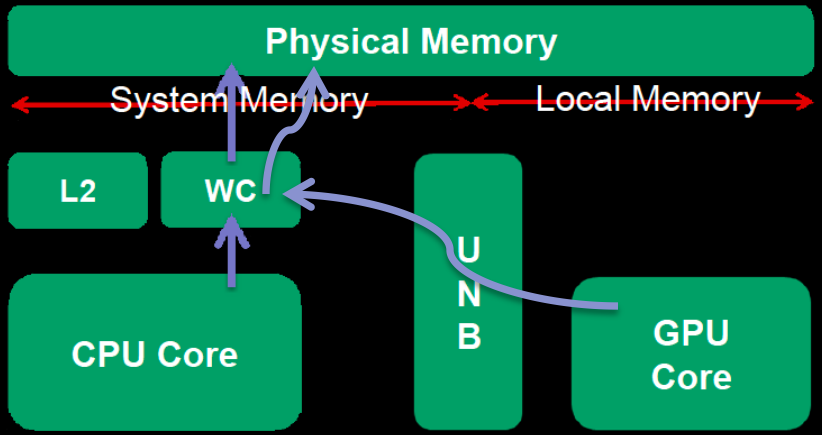



```
clCreateBuffer( myctx,
CL_MEM_READ_ONLY|CL_MEM_USE_PERSISTENT_MEM_AMD,
mem_size,  NULL, &ciErrNum);
```

MEMORY SYSTEM ON FUSION APUS - **Pierre Boudier & Graham Sellers.** AMD Fusion Developer Summit 2011

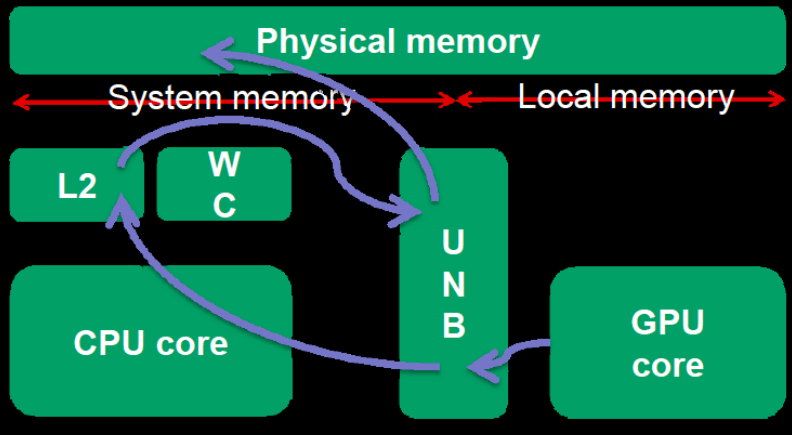
PROGRAMMING SHARED MEMORY HETEROGENEOUS DEVICES

Northeastern University



GPU / CPU access to uncached system memory

```
clCreateBuffer( myctx,
CL_MEM_READ_ONLY |
CL_MEM_ALLOC_HOST_PTR,
mem_size,  NULL, &ciErrNum);
```



GPU access to cached system memory

```
clCreateBuffer( myctx,
CL_MEM_READ_WRITE |
CL_MEM_ALLOC_HOST_PTR,
mem_size,  NULL, &ciErrNum);
```

OPENCL V1.2 - DIFFERENCES

- Device partitioning:
  - Partition a device into sub-devices so that work groups can be allocated to individual compute units.
  - Useful for reserving areas of the device to reduce latency for time-critical tasks.
- Separate compilation and linking of objects:
  - Functionality to compile OpenCL into external libraries for inclusion into other programs.
- Enhanced image support:
  - Support for 1D images and 1D/2D image arrays.
  - Extensions allow for OpenGL textures and arrays to be used to create OpenCL images
- Built-in kernels:
  - Custom devices that contain unique functionality are now integrated more closely into OpenCL
  - Kernels can be called to use specialised or non-programmable aspects of underlying hardware.
  - Examples include, video encoding/decoding, and digital signal processors.
- DirectX functionality: DX9 media surface sharing allows for efficient sharing between OpenCL and DX9

CONCLUSIONS ON HETEROGENEOUS COMPUTING

Northeastern University

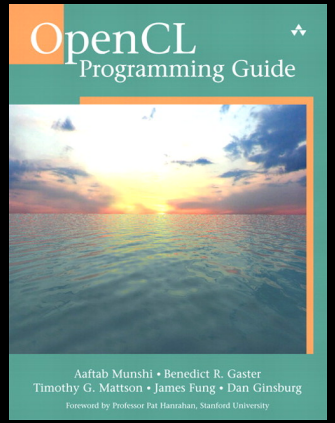
- Targeting heterogeneous devices (e.g., CPUs and GPUs at the same time) requires awareness of their different performance characteristics for an application
- Scheduling overhead
  - What is the startup time of each device?
- Location of data
  - Which device is the data currently resident on?
  - Data must be transferred across the PCI-Express bus
- Subdivision granularity of workloads across devices ?
  - Too large may execute slowly on a device, stalling overall completion
  - Too small may be dominated by startup overhead

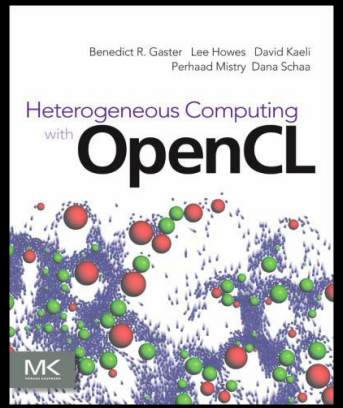
	CPUs	GPUs
Overhead	Low	High (depending on data)
Performance	Variable	High

RELEVANT RESOURCES FOR OPENCL

Northeastern University

- Books
  - Heterogenous Computing with OpenCL
  - OpenCL Programming Guide
- Tutorial on Fusion Memory model
  - *MEMORY SYSTEM ON FUSION APUS* - Pierre Boudier & Graham Sellers. AMD Fusion Developer Summit 2011
- Webinars. Lectures
  - AMD OpenCL University Toolkit
  - http://developer.amd.com/zones/OpenCLZone/Events/pages/OpenCLWebinars.aspx
- Debugging
  - GDEbugger - Windows Only
- Systems to test
  - NUCAR provides systems to test and run OpenCL code. Contact Prof. Kaeli





Northeastern University

THANK YOU !!

QUESTIONS ? COMMENTS ?

Perhaad Mistry
pmistry@ece.neu.edu