Illinois UPCRC Summer School 2010

The OpenCL Programming Model

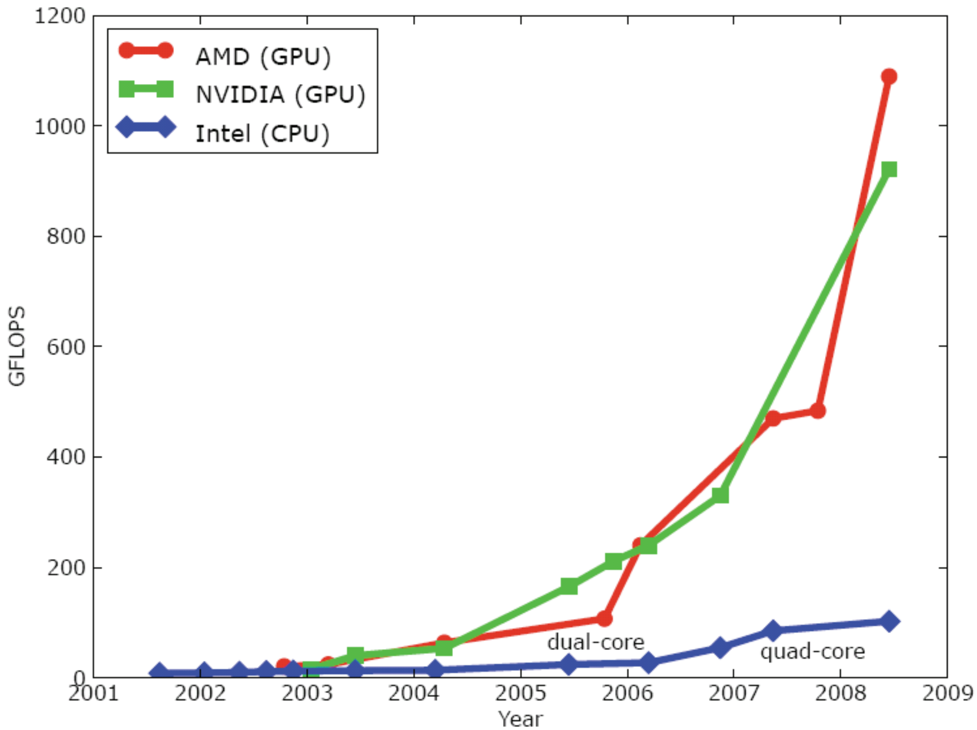
Part 1: Basic Concepts

Wen-mei Hwu and John Stone
with special contributions from
Deepthi Nandakumar

UPCRC Illinois
Universal Parallel Computing Research Center

Why GPU Computing

- An enlarging peak performance advantage:
  - Calculation: 1 TFLOPS vs. 100 GFLOPS
  - Memory Bandwidth: 100-150 GB/s vs. 32-64 GB/s



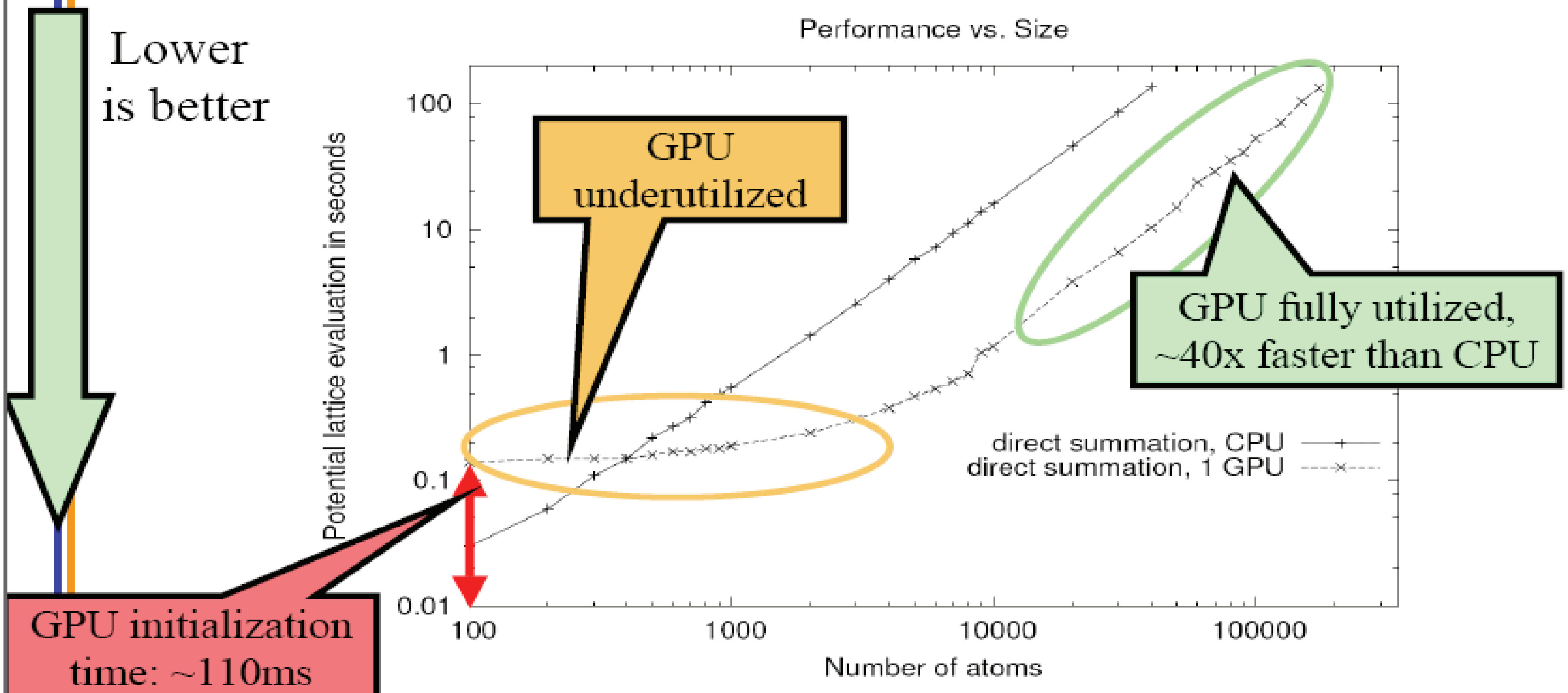
Courtesy: John Owens

  - GPU in every PC and workstation – massive volume and potential impact

Universal Parallel Computing Research Center

Role of GPUs - large data sets

Direct Coulomb Summation Runtime



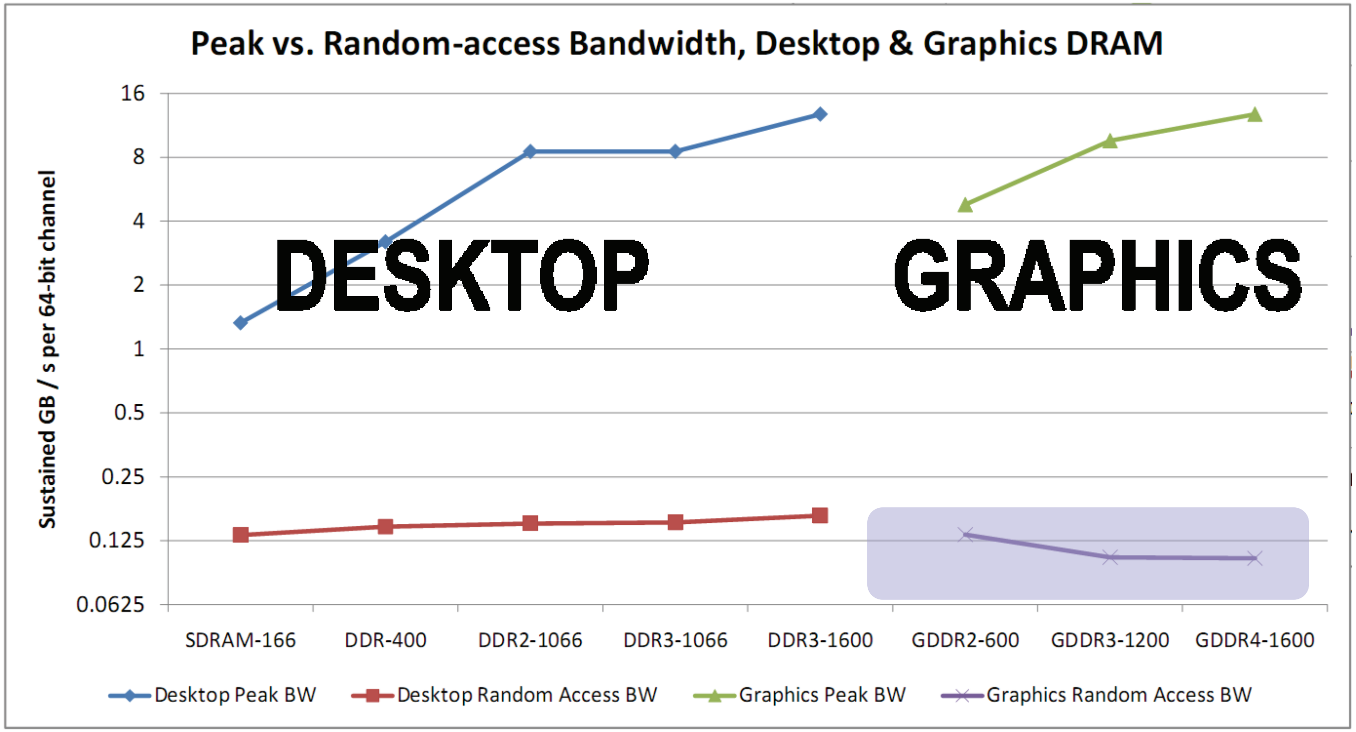
Accelerating molecular modeling applications with graphics processors. J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten. *J. Comp. Chem.*, 28:2618-2640, 2007.

Future Apps Reflect a Concurrent World

- Exciting applications in future computing have been traditionally considered "supercomputing applications"
  - Video and audio synthesis/analysis, 3D imaging and visualization, consumer game physics, virtual reality products, computational financing, molecular dynamics simulation, computational fluid dynamics
  - These "Super-apps" represent and model the physical, concurrent world
- Various granularities of parallelism exist, but…
  - programming model must not hinder scalable implementation
  - data delivery needs careful management

UPCRC Illinois
Universal Parallel Computing Research Center

DRAM Bandwidth Trends Sets Programming Agenda



- **Random access BW** 1.2% **of peak for DDR3-1600,** 0.8% **for GDDR4-1600 (and falling)**
- **3D stacking and optical interconnects will unlikely help.**

Research Center

UIUC/NCSA AC Cluster

- 32 nodes
  - 4-GPU (GTX280, Tesla), 1-FPGA, quad-core Opteron node at NCSA
  - GPUs donated by NVIDIA
  - FPGA donated by Xilinx
  - 128 TFLOPS single precision, 10 TFLOPS double precision
- Coulomb Summation:
  - 1.78 TFLOPS/node
  - 271x speedup vs. Intel QX6700 CPU core w/ SSE

UIUC/NCSA AC Cluster

http://www.ncsa.uiuc.edu/Projects/GPUcluster/

A partnership between NCSA and academic departments.

UPCRC Illinois
Universal Parallel Computing Research Center

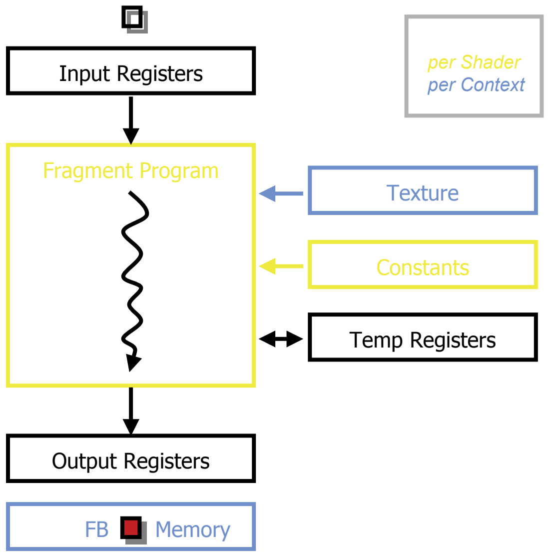
What is (Historical) GPGPU ?

- General Purpose computation using GPU and graphics API in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

UPCRC Illinois
Universal Parallel Computing Research Center

Previous GPGPU Constraints

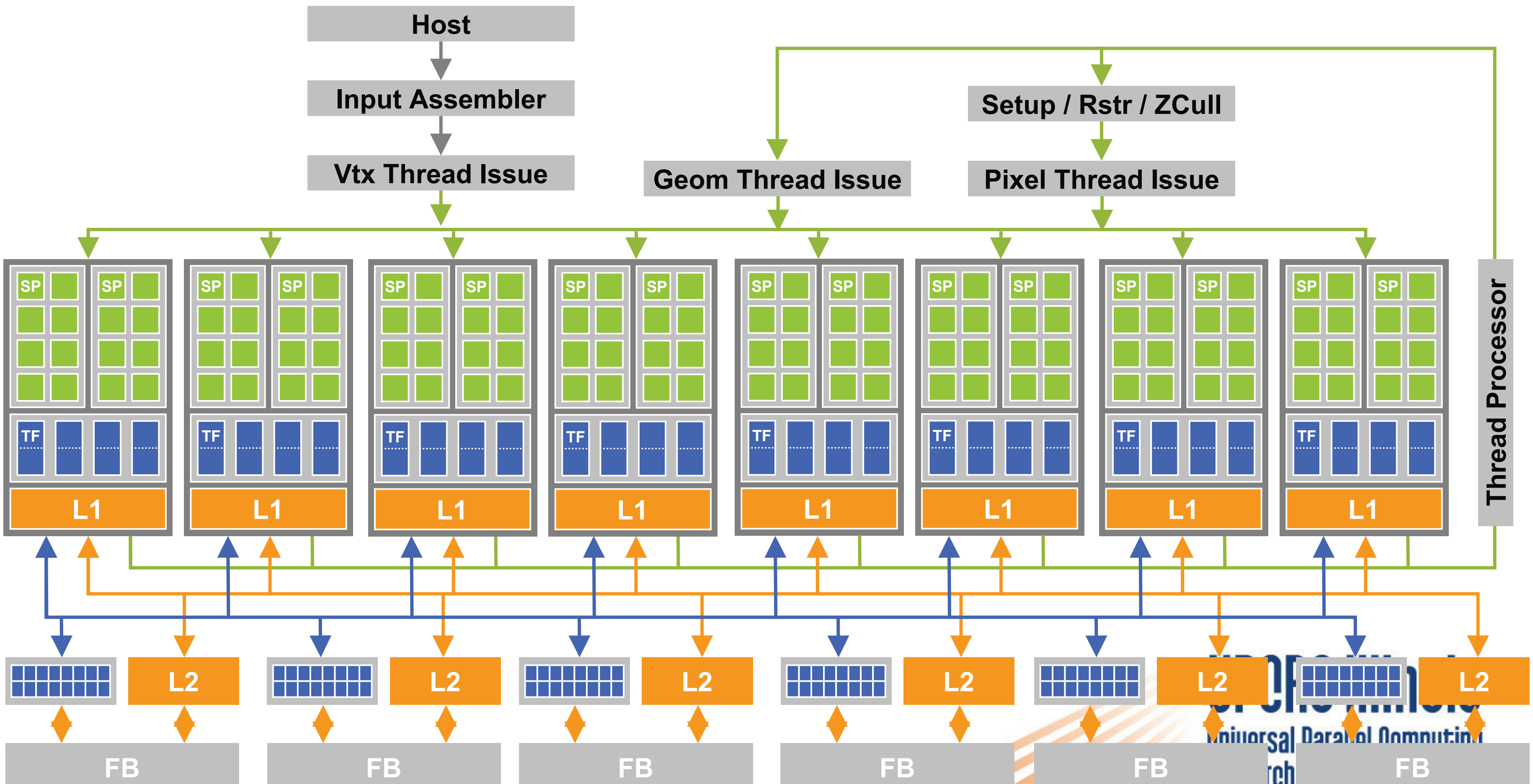
- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Lack of Integer & bit ops
- Communication limited
  - Between pixels
  - Scatter a[i] = p



UPCRC Illinois
Universal Parallel Computing Research Center

G80 – Graphics Mode

- The future of GPUs is programmable processing
- So – build the architecture around the processor

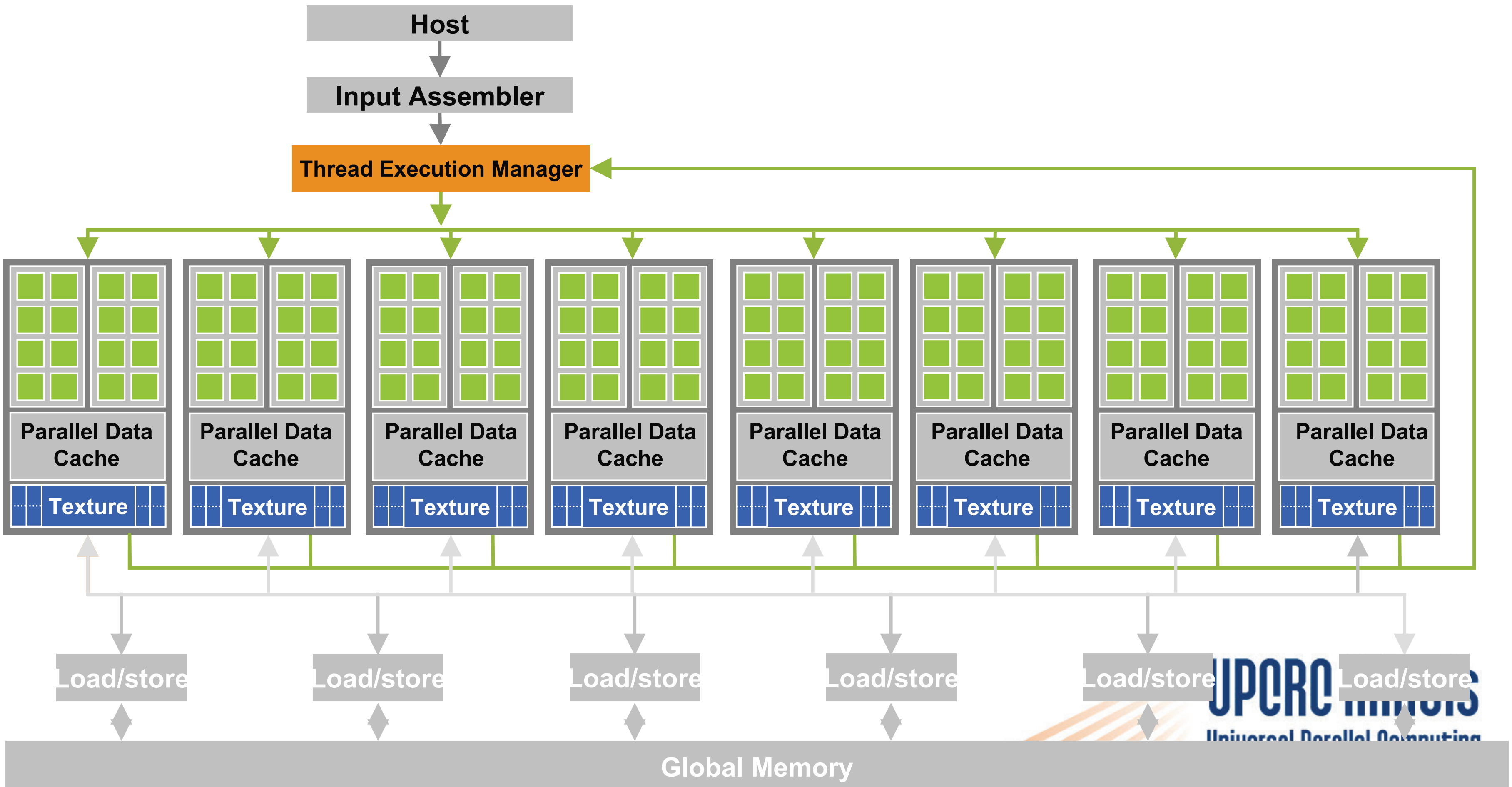


CUDA – Recent OpenCL Predecessor

- "Compute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute – graphics-free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

G80 CUDA mode – A **Device** Example

- Processors execute computing threads
- New operating mode/HW interface for computing



What is OpenCL?

- Cross-platform parallel computing API and C-like language for heterogeneous computing devices
- Code is portable across various target devices:
  - Correctness is guaranteed
  - Performance of a given kernel is not guaranteed across differing target devices
- OpenCL implementations already exist for AMD and NVIDIA GPUs, x86 CPUs
- In principle, OpenCL could also target DSPs, Cell, and perhaps also FPGAs

UPCRC Illinois
Universal Parallel Computing Research Center

More on Multi-Platform Targeting

- Targets a broader range of CPU-like and GPU-like devices than CUDA
  - Targets devices produced by multiple vendors
  - Many features of OpenCL are optional and may not be supported on all devices
- OpenCL codes must be prepared to deal with much greater hardware diversity
- A single OpenCL kernel will likely not achieve peak performance on all device types

UPCRC Illinois
Universal Parallel Computing Research Center

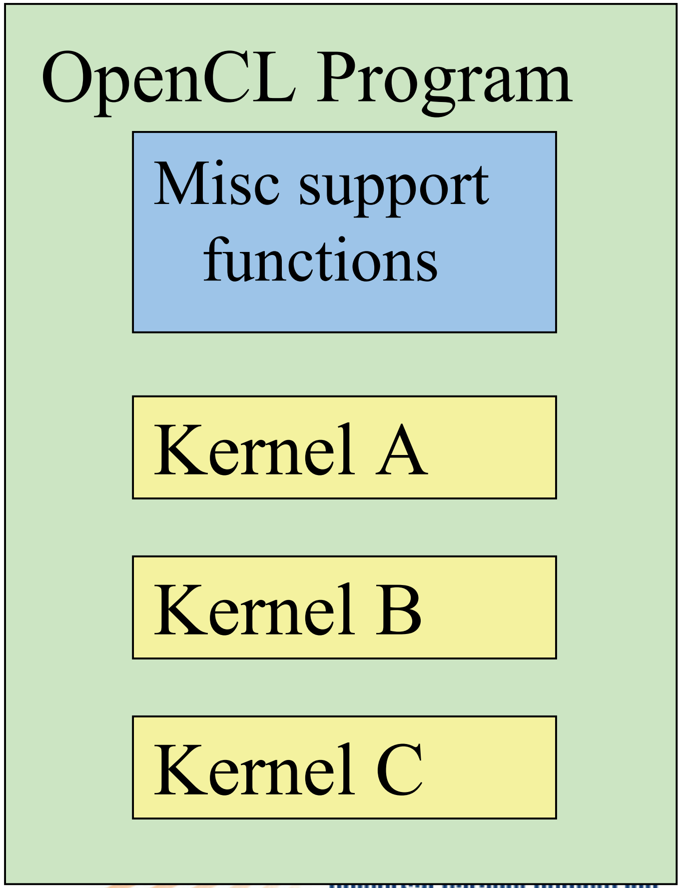
Overview

- OpenCL programming model – basic concepts and data types
- OpenCL application programming interface - basic
- Simple examples to illustrate basic concepts and functionalities
- Case study to illustrate performance considerations

Universal Parallel Computing Research Center

OpenCL Programs

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



Universal Parallel Computing Research Center

OpenCL Execution Model

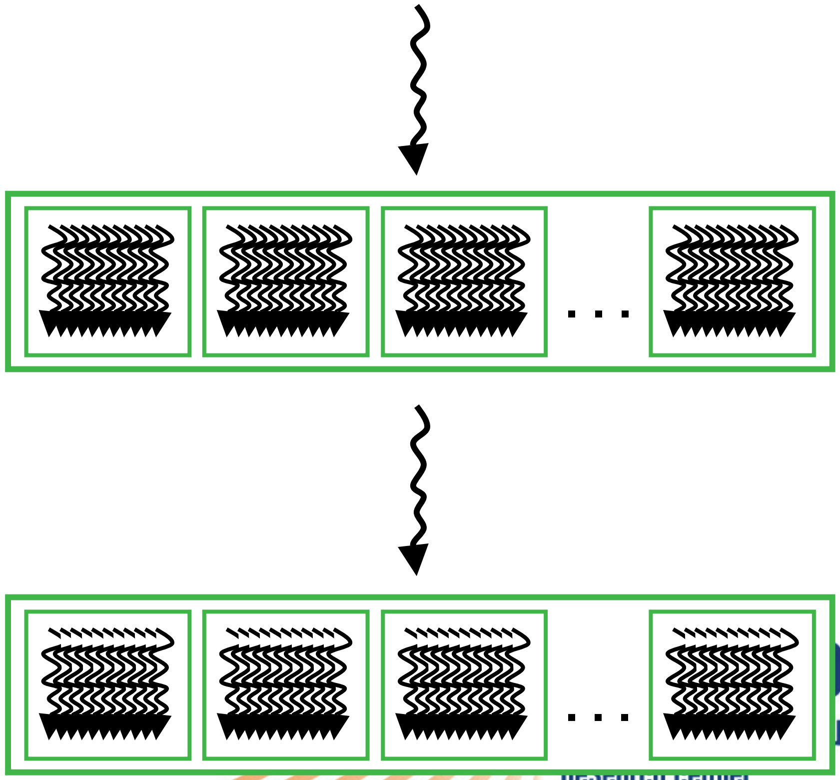
- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);



OpenCL Kernels

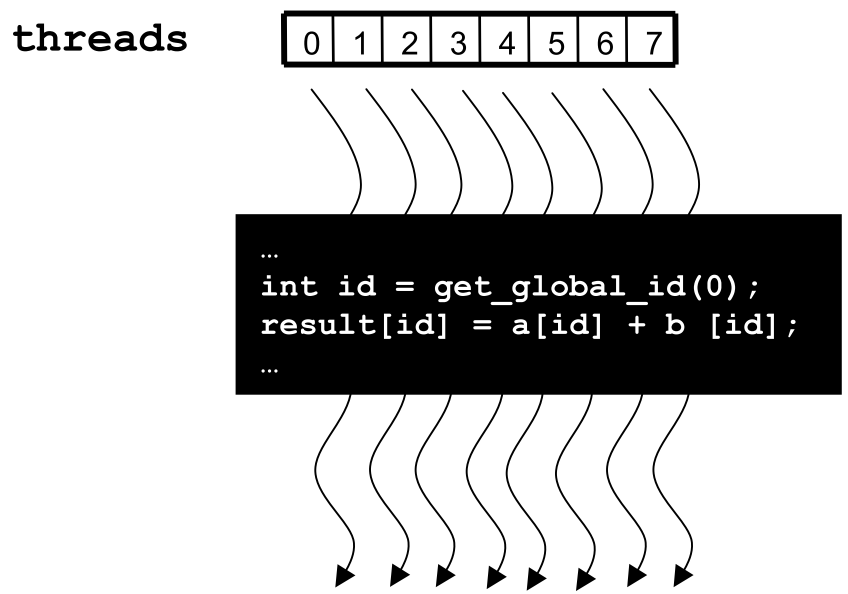
- Code that actually executes on target devices
- Kernel body is instantiated once for each work item
  - An OpenCL work item is equivalent to a CUDA thread
- Each OpenCL work item gets a unique index

```
__kernel void
vadd(__global const float *a,
     __global const float *b,
     __global float *result) {
 int id = get_global_id(0);
 result[id] = a[id] + b[id];
}
```

UPCRC Illinois
Universal Parallel Computing Research Center

Array of Parallel Work Items

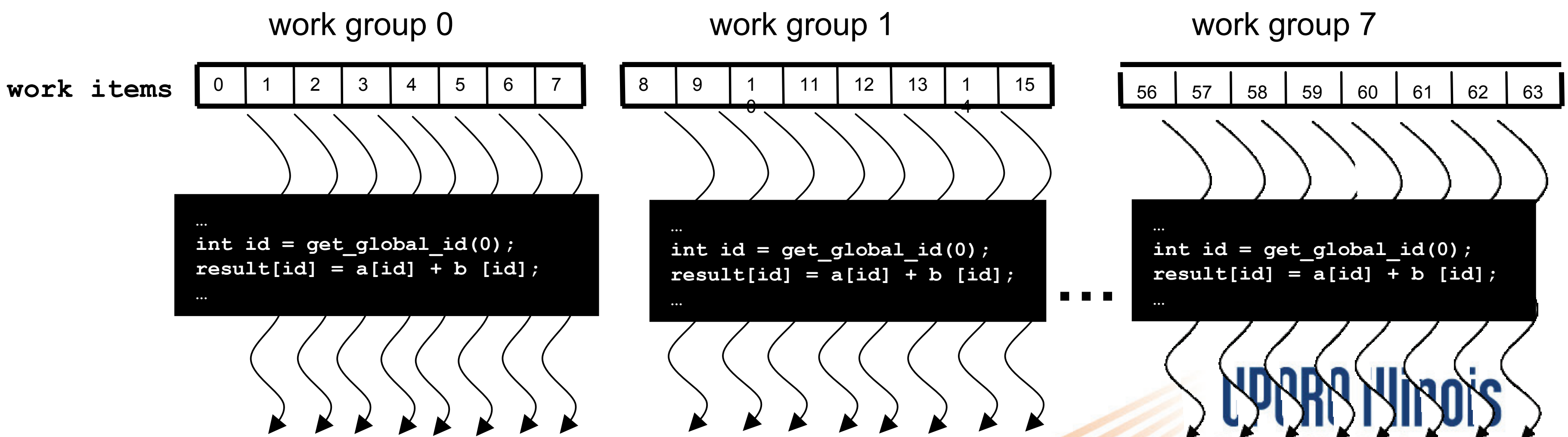
- An OpenCL kernel is executed by an array of work items
  - All work items run the same code (SPMD)
  - Each work item has an index that it uses to compute memory addresses and make control decisions




```
…
int id = get_global_id(0);
result[id] = a[id] + b [id];
…
```

Work Groups: Scalable Cooperation

- Divide monolithic work item array into work groups
  - Work items within a work group cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Work items in different work groups cannot cooperate



UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Data Parallel Model Summary

- Parallel work is submitted to devices by launching kernels
- Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”
- Work items executing within the same work group can synchronize with each other with barriers or memory fences
- Work items in different work groups can’t sync with each other, except by launching a new kernel

Universal Parallel Computing Research Center

OpenCL Host Code

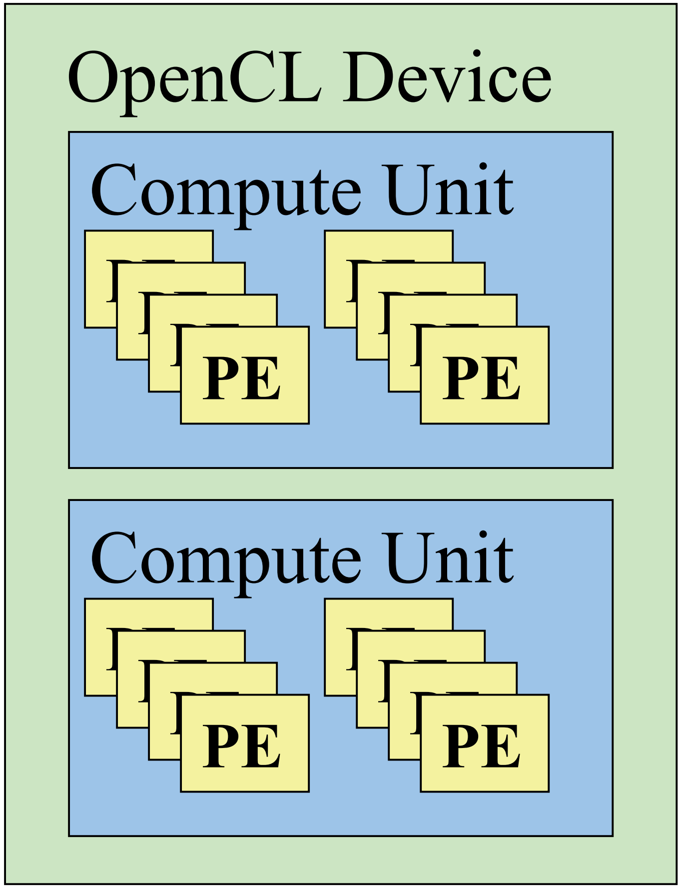
- Prepare and trigger device code execution
  - Create and manage device context(s) and associate work queue(s), etc…
  - Memory allocations, memory copies, etc
  - Kernel launch

- OpenCL programs are normally compiled entirely at runtime, which must be managed by host code

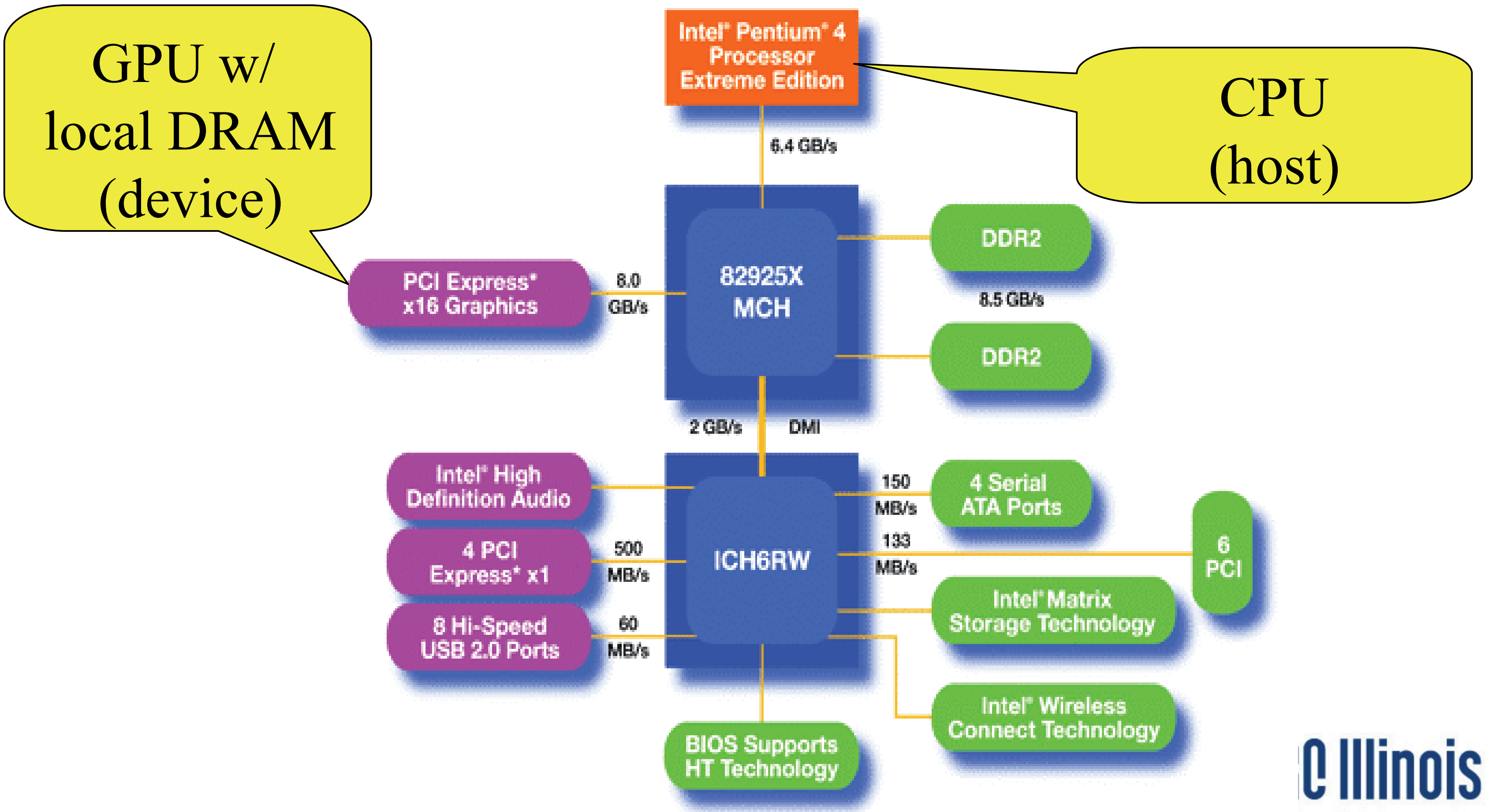
UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Hardware Abstraction

- OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”
- Each “device” contains one or more “compute units”, i.e. cores, SMs, etc...
- Each “compute unit” contains one or more SIMD “processing elements”



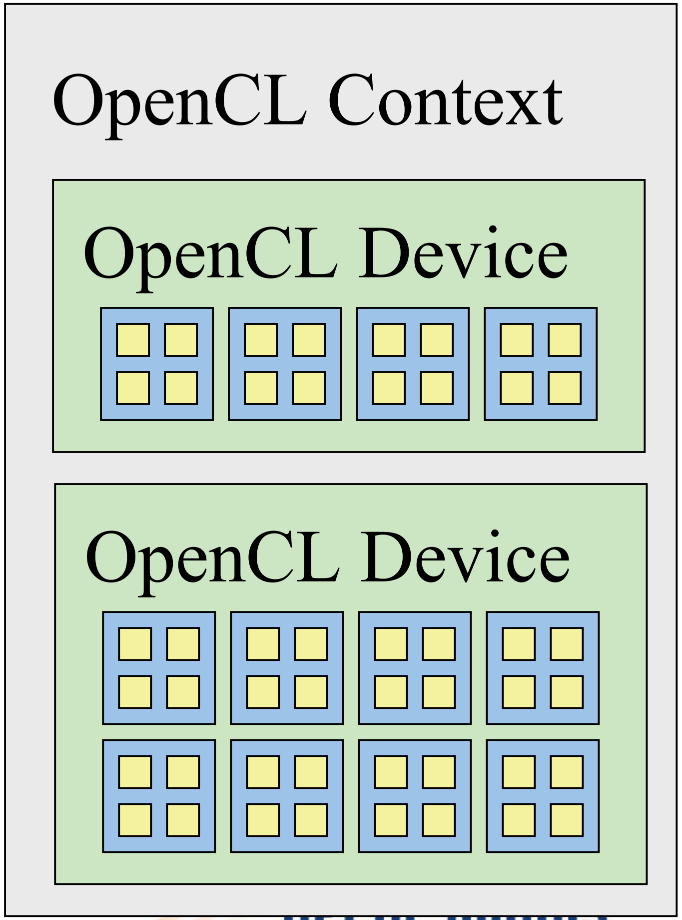
An Example of Physical Reality Behind OpenCL Abstraction



Illinois
Parallel Computing Research Center

OpenCL Context

- Contains one or more devices
- OpenCL memory objects are associated with a context, not a specific device
- clCreateBuffer() is the main data object allocation function
  - error if an allocation is too large for any device in the context
- Each device needs its own work queue(s)
- Memory transfers are associated with a command queue (thus a specific device)



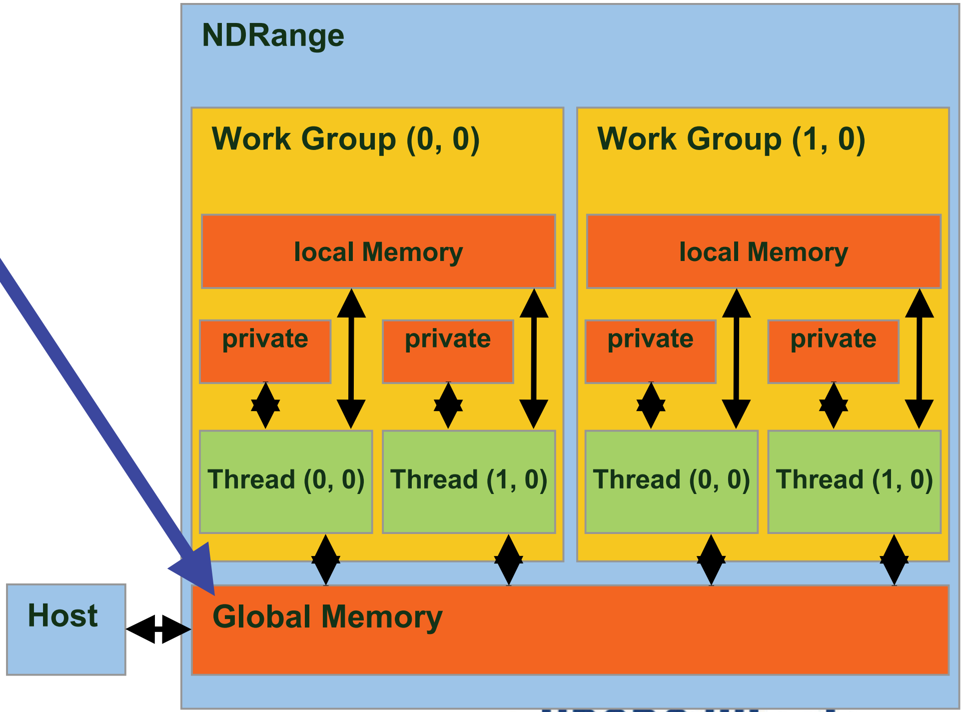
Universal Parallel Computing Research Center

OpenCL Context Setup Code (simple)

```
cl_int clerr = CL_SUCCESS;
cl_context clctx = clCreateContextFromType(0, CL_DEVICE_TYPE_ALL, NULL,
    NULL, &clerr);

size_t parmsz;
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);

cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz, cldevs,
    NULL);

cl_command_queue clcmdq = clCreateCommandQueue(clctx, cldevs[0], 0,
    &clerr);
```

OpenCL Memory Model Overview

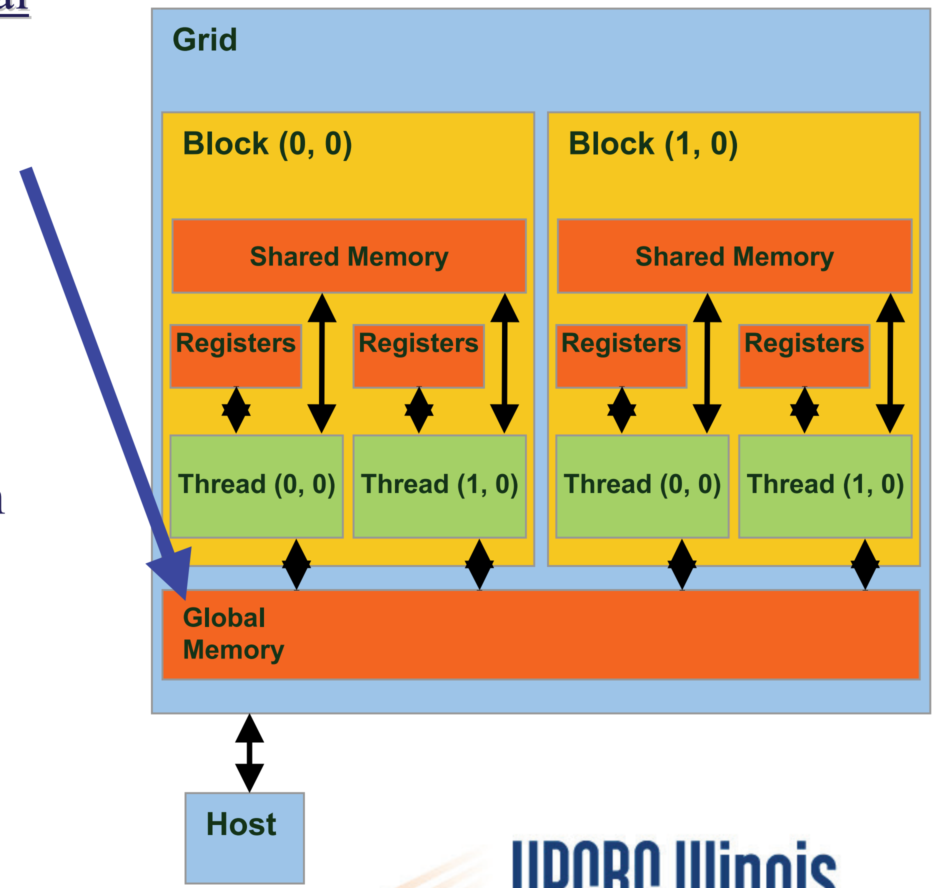
- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now



UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Device Memory Allocation

- **clCreateBuffer();**
  - Allocates object in the device Global Memory
  - Returns a pointer to the object
  - Requires five parameters
    - OpenCL context pointer
    - Flags for access type by device
    - Size of allocated object
    - Host memory pointer, if used in copy-from-host mode
    - Error code
- **clReleaseMemObject()**
  - Frees object
    - Pointer to freed object



UPCRC Illinois
Universal Parallel Computing Research Center

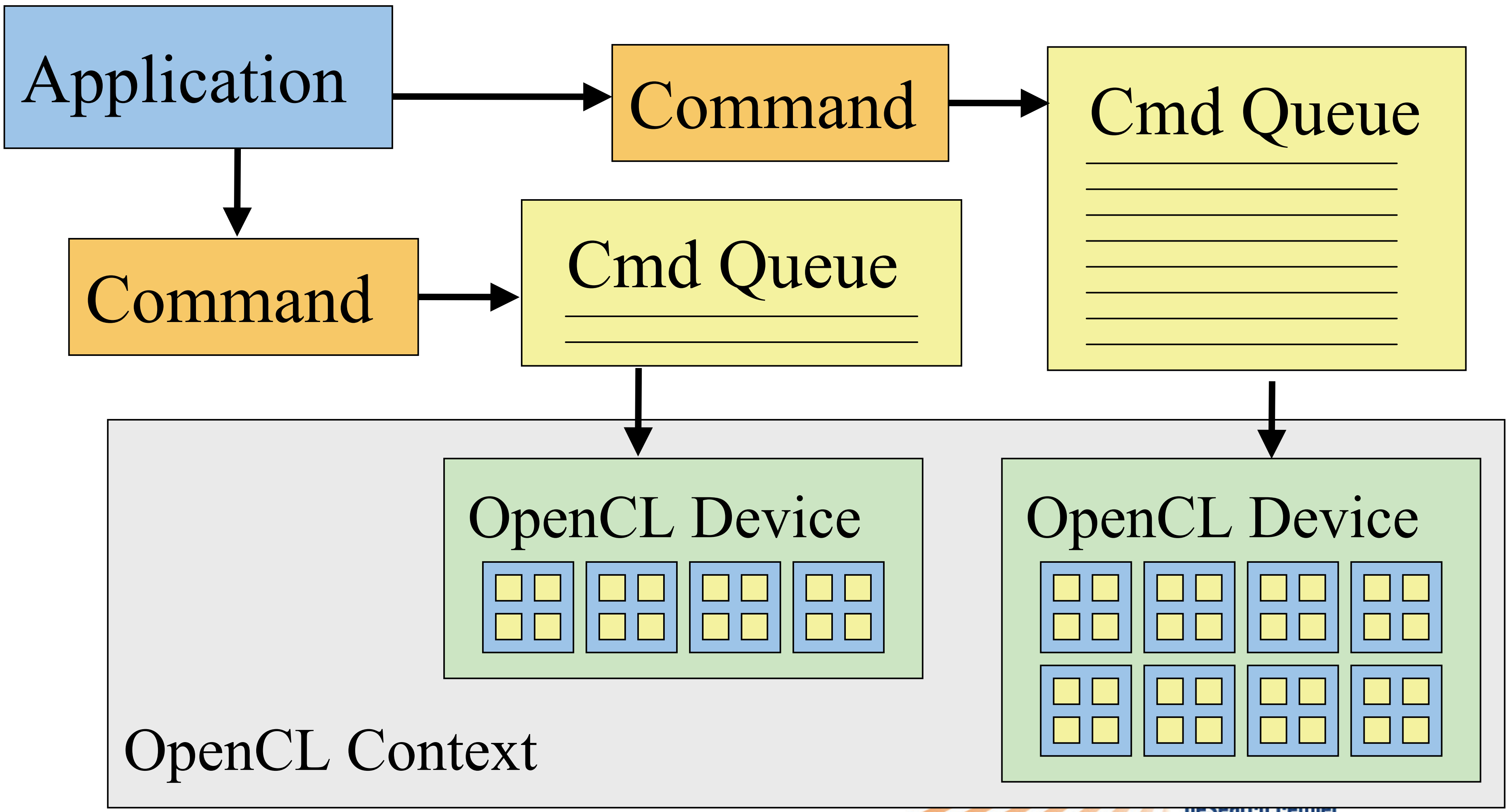
OpenCL Device Memory Allocation (cont.)

- Code example:
  - Allocate a 1024 single precision float array
  - Attach the allocated storage to d_a
  - “d” is often used to indicate a device data structure

```
VECTOR_SIZE = 1024;
cl_mem d_a;
int size = VECTOR_SIZE* sizeof(float);

d_a = clCreateBuffer(clctx, CL_MEM_READ_ONLY, size,
NULL, NULL);
clReleaseMemObject(d_a);
```

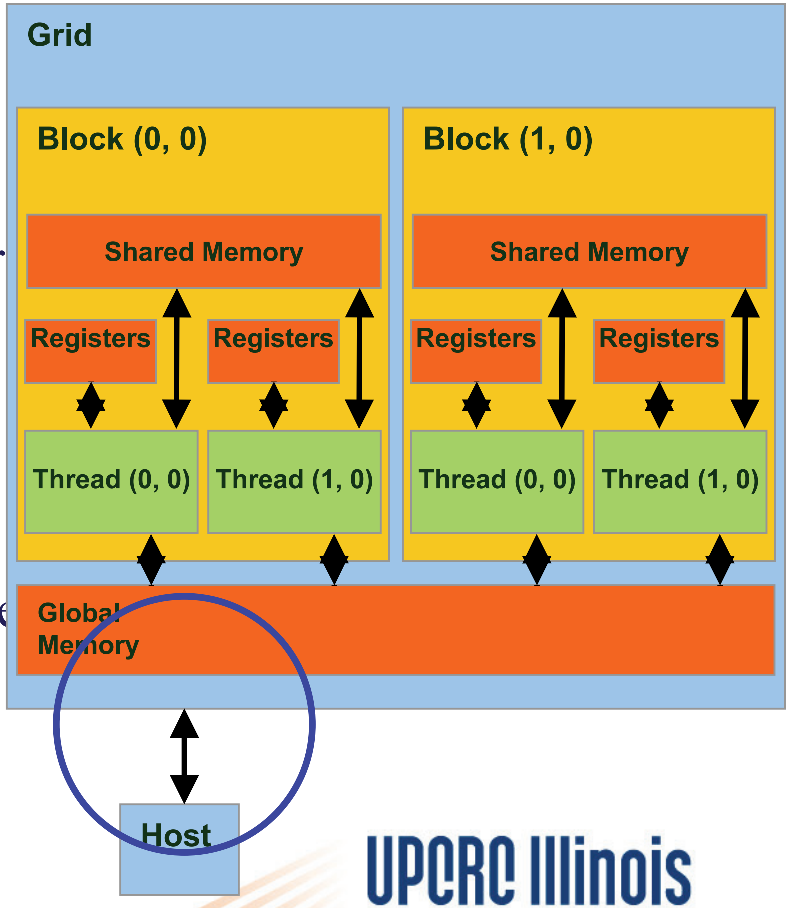
UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Device Command Execution



OpenCL Host-to-Device Data Transfer

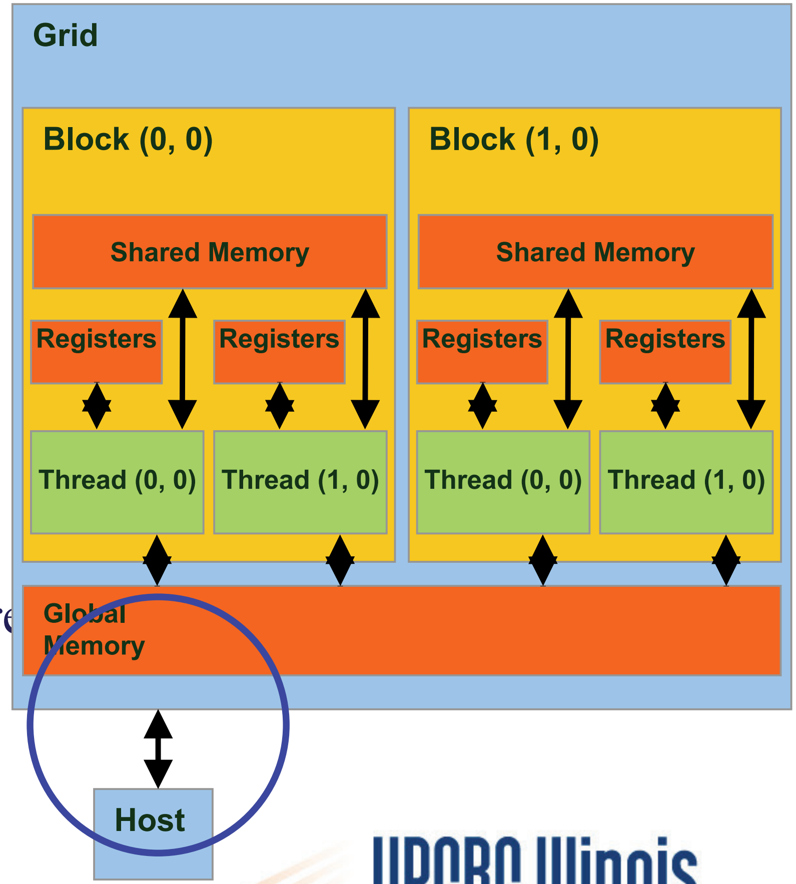
- clEnqueueWriteBuffer();
  - memory data transfer to device
  - Requires nine parameters
    - OpenCL command queue pointer
    - Destination OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size of bytes of written data
    - Host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command
- Asynchronous transfer later



UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Device-to-Host Data Transfer

- clEnqueueReadBuffer();
  - memory data transfer to host
  - Requires nine parameters
    - OpenCL command queue pointer
    - Source OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size of bytes of read data
    - Destination host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command
- Asynchronous transfer later



UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - a is in host memory and d_a is in device memory

```
clEnqueueWriteBuffer(clcmdq, d_a, CL_FALSE, 0,
    mem_size, (const void * )a, 0, 0, NULL);

clEnqueueReadBuffer(clcmdq, d_result, CL_FALSE, 0,
    mem_size, (void * ) host_result, 0, 0, NULL);
```

UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Host-Device Data Transfer (cont.)

- clCreateBuffer and clEnqueueWriteBuffer can be combined into a single command using special flags.
- Eg:

```
d_A=clCreateBuffer(clctxt, CL_MEM_READ_ONLY |
  CL_MEM_COPY_HOST_PTR, mem_size, h_A, NULL);
```

  - Combination of 2 flags here. CL_MEM_COPY_HOST_PTR to be used only if a valid host pointer is specified.
  - This creates a memory buffer on the device, and copies data from h_A into d_A.
  - Includes an implicit clEnqueueWriteBuffer operation, for all devices/command queues tied to the context clctxt.

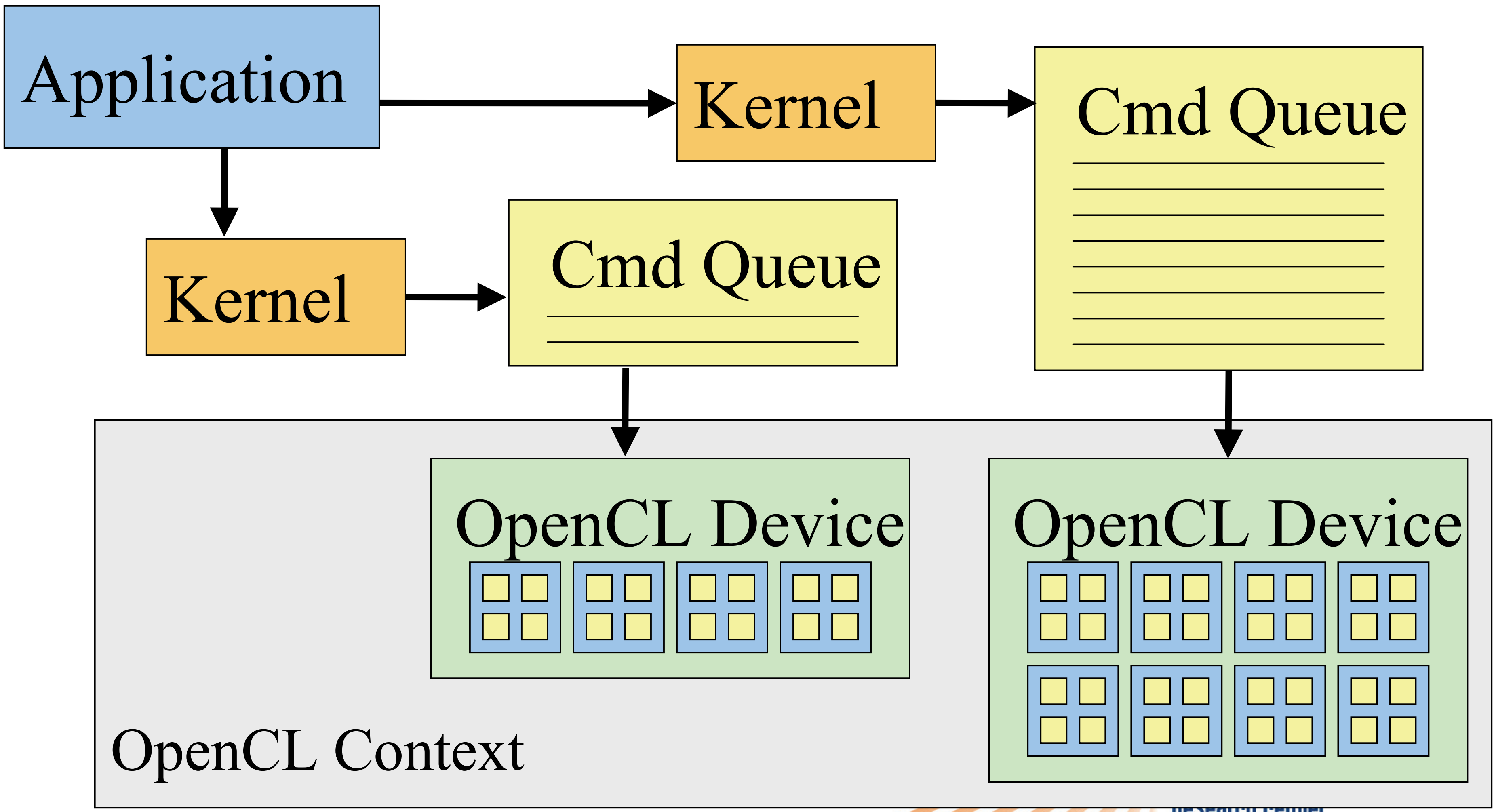
UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Memory Systems

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

UPCRC Illinois
Universal Parallel Computing Research Center

OpenCL Kernel Execution Launch



OpenCL Kernel Compilation Example

OpenCL kernel source code as a big string

```
const char* vaddsrc =
  "__kernel void vadd(__global float *d_A, __global float *d_B, __global float *d_C, int N) {
  \n"   […etc and so forth…]
cl_program clpgm;
clpgm = clCreateProgramWithSource(clctx, 1, &vaddsrc, NULL, &clerr);
char clcompileflags[4096];
sprintf(clcompileflags, "-cl-mad-enable");
clerr = clBuildProgram(clpgm, 0, NULL, clcompileflags, NULL, NULL);
cl_kernel clkern = clCreateKernel(clpgm, "vadd", &clerr);
```

Gives raw source code string(s) to OpenCL

Set compiler flags, compile source, and retreive a handle to the "vadd" kernel

Summary: Host code for `vadd`

```
int main()
{    // allocate and initialize host (CPU) memory
     float *h_A = …,    *h_B = …;
     // allocate device (GPU) memory
    cl_mem d_A, d_B, d_C;
    d_A = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
           CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_A, NULL);
    d_B = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
           CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_B, NULL);
    d_C = clCreateBuffer(clctx, CL_MEM_WRITE_ONLY, N *sizeof(float), NULL, NULL);
    clkern=clCreateKernel(clpgm, "vadd", NULL);
    clerr= clSetKernelArg(clkern, 0,sizeof(cl_mem), (void *) &d_A);
    clerr= clSetKernelArg(clkern, 1,sizeof(cl_mem), (void *) &d_B);
    clerr= clSetKernelArg(clkern, 2,sizeof(cl_mem), (void *) &d_C);
    clerr= clSetKernelArg(clkern, 3,sizeof(int), &N);
    cl_event event=NULL;
    clerr= clEnqueueNDRangeKernel(clcmdq,clkern, 2, NULL, Gsz,Bsz,
          0, NULL, &event);
    clerr= clWaitForEvents(1, &event);
    clEnqueueReadBuffer(clcmdq, d_C, CL_TRUE, 0, N*sizeof(float), h_C, 0,
                   NULL, NULL);
    clReleaseMemObject(d_A);
    clReleaseMemObject(d_B);
    clReleaseMemObject(d_C);
}
```

UPCRC Illinois
Universal Parallel Computing Research Center

Let's take a break!

UPCRC Illinois
Universal Parallel Computing Research Center