

Programming in OpenCL

Timo Stich, NVIDIA

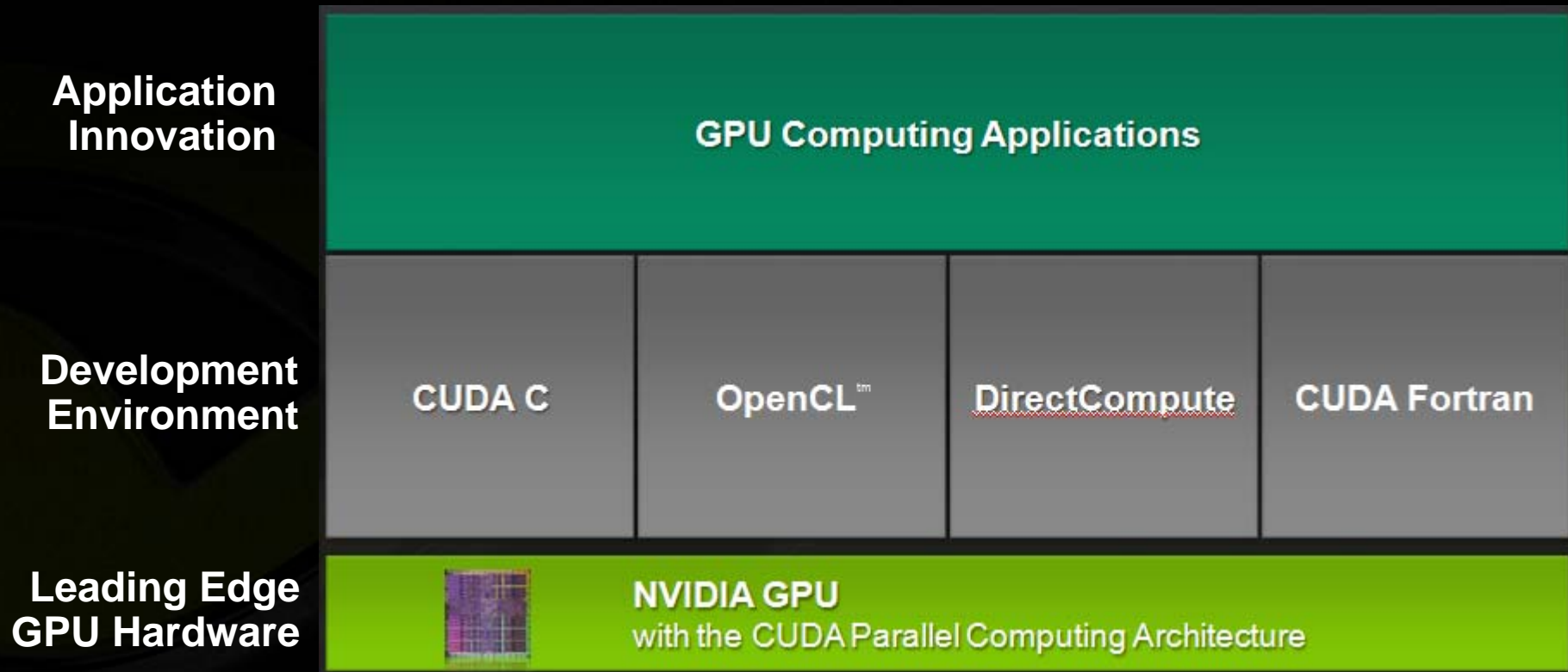


Outline



- **Introduction to OpenCL**
- **OpenCL API Overview**
- **Performance Tuning on NVIDIA GPUs**
- **OpenCL Programming Tools & Resources**

OpenCL and the CUDA Architecture



OpenCL Portability



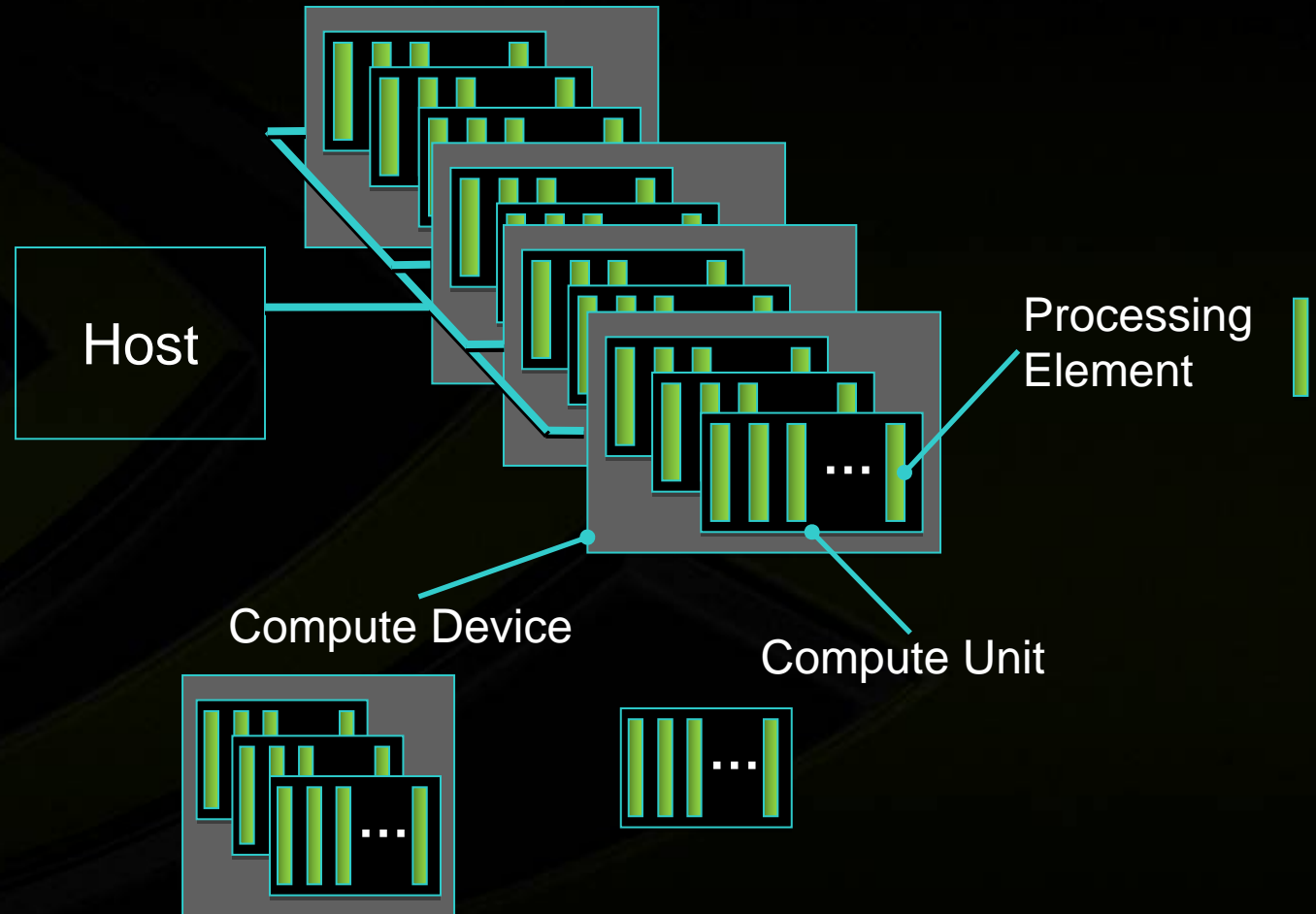
- **Portable code across multiple devices**
 - GPU, CPU, Cell, mobiles, embedded systems, ...

- **NOTE:**

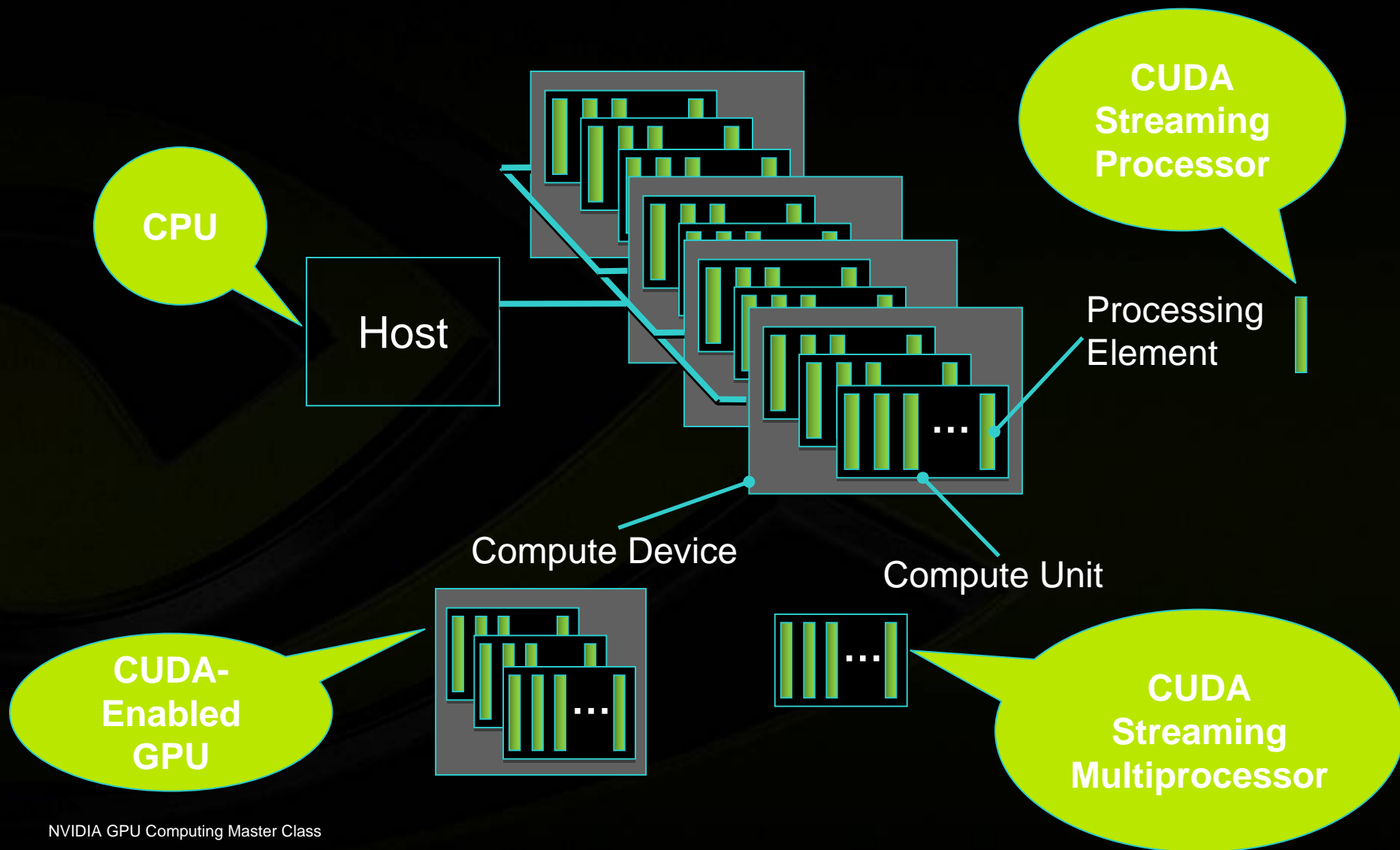
functional portability != performance portability

- **Different code for each device is necessary to get good performance**
- **Even for GPUs from different vendors!**

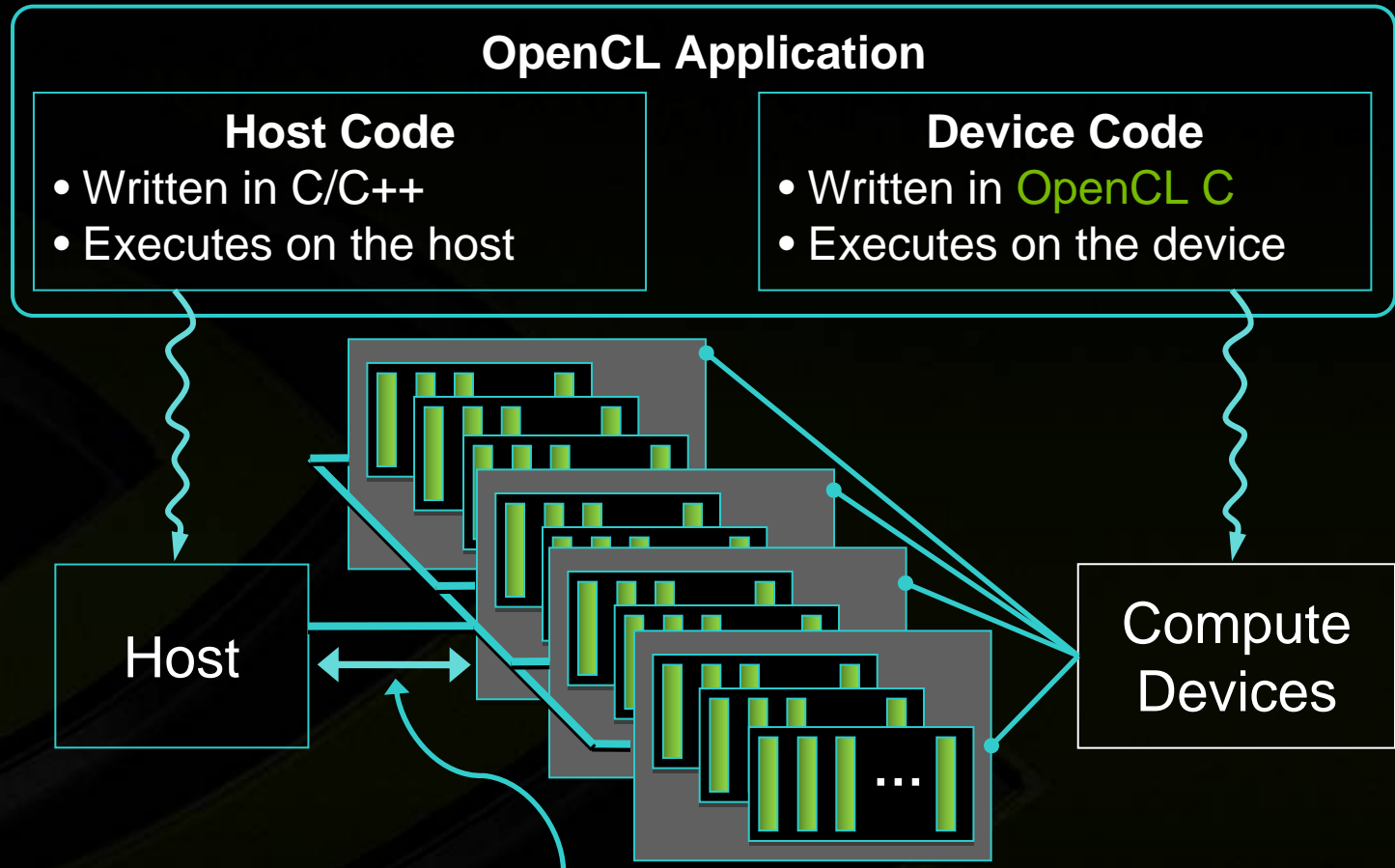
OpenCL Platform Model



OpenCL Platform Model on the CUDA Architecture



Anatomy of an OpenCL Application



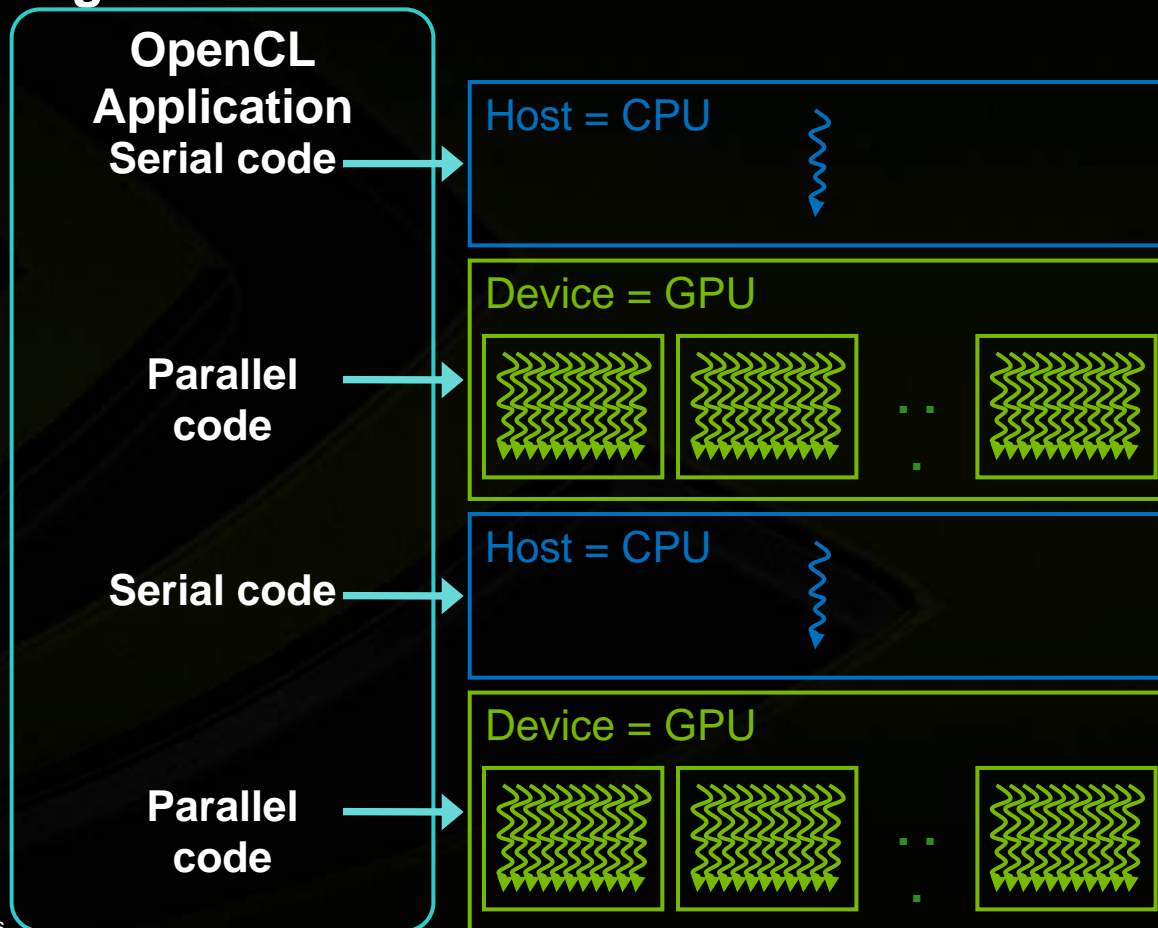
Host code sends commands to the devices:

- to transfer data between host memory and device memories
- to execute device code

Heterogeneous Computing



- **Serial** code executes in a **CPU** thread
- **Parallel** code executes in many **GPU** threads across multiple processing elements



OpenCL Framework

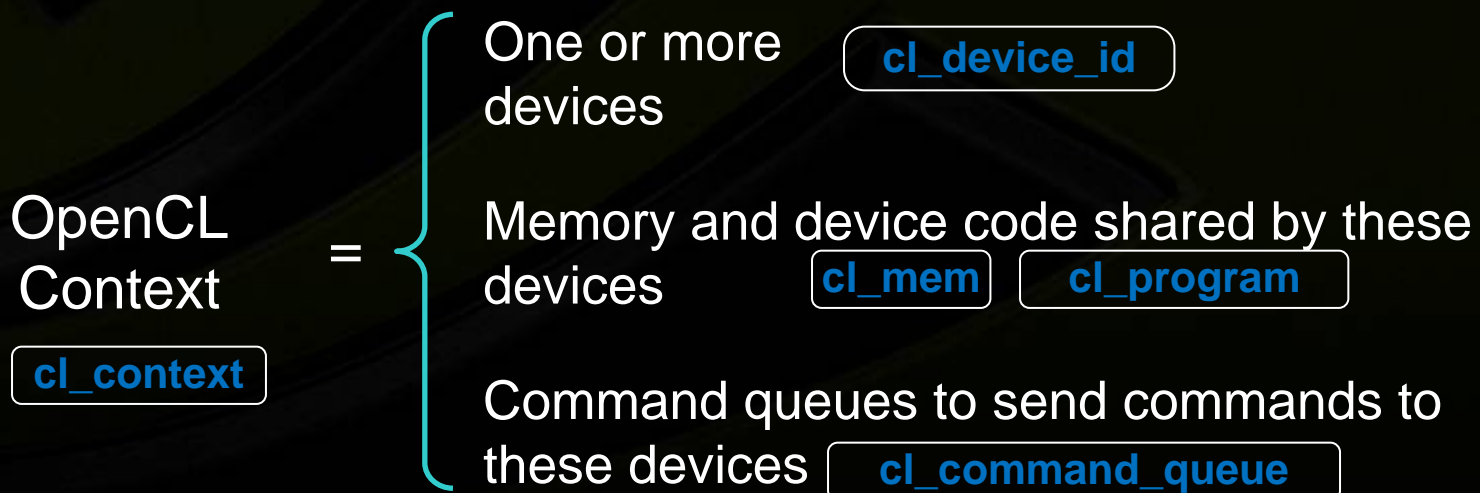


- **Platform layer**
 - Discover OpenCL devices and their capabilities and create contexts
- **Runtime layer**
 - Memory management and command execution within a context
- **OpenCL C Compiler**
 - Creates program executables that contain OpenCL kernels

Platform Layer



- Query platform information
 - `clGetPlatformIDs()`: list of platforms
 - `clGetPlatformInfo()`: profile, version, vendor, extensions
 - `clGetDeviceIDs()`: list of devices
 - `clGetDeviceInfo()`: type, capabilities
- Create OpenCL context on one or more devices of one platform



Error Handling, Resource Deallocation

- **Error handling:**
 - All host functions return an error code
 - Context error callback can be specified
- **Resource deallocation**
 - Reference counting API: `clRetain*()`, `clRelease*()`
- **Both are removed from code samples for clarity**
 - Please see SDK samples for complete code

Context Creation



// Create an OpenCL context for all GPU devices on the first Platform

```
cl_context* CreateContext() {  
    cl_platform_id platform_id;  
    clGetPlatformIDs(1, &platform_id, NULL);
```

```
    return clCreateContextFromType(  
        {CL_CONTEXT_PLATFORM, platform_id, 0},  
        CL_DEVICE_TYPE_GPU,  
        NULL, NULL, NULL);
```

```
}
```

// Get the list of GPU devices associated with a context

```
cl_device_id* GetDevices(cl_context context) {  
    size_t size;  
    clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &size);  
    cl_device_id* device_id = malloc(size);  
    clGetContextInfo(context, CL_CONTEXT_DEVICES, size, device_id,  
        NULL);  
    return device_id;
```

```
}
```

Runtime



- **Command queues creation and management**
- **Memory allocation and management**
- **Device code compilation and execution**
- **Event creation and management (synchronization, profiling)**

Command Queue



- **Sequence of commands scheduled for execution on a specific device**
 - Enqueuing functions: **clEnqueue*()**
 - Multiple queues can execute on the same device
- **Two modes of execution:**
 - **In-order:** Each command in the queue executes only when the preceding command has completed
 - Including all memory writes, so memory is consistent with all prior command executions
 - **Out-of-order:** No guaranteed order of completion for commands

Commands



- **Memory copy or mapping**
- **Device code execution**
- **Synchronization point**

Command Queue Creation

```
// Create a command-queue for a specific device
cl_command_queue CreateCommandQueue(cl_context context,
    cl_device_id device_id)
{
    return clCreateCommandQueue(context, device_id, 0, NULL);
}
```

Properties

Error
code

Command Synchronization



- Some **clEnqueue***() calls can be optionally blocking
- Queue barrier command
 - Any commands after the barrier start executing only after all commands before the barrier have completed
- An event object can be associated to each enqueued command
 - Any commands (or **clWaitForEvents()**) can wait on events before executing
 - Can be queried to track execution status and get profiling information

Memory Objects



- **Two types of memory objects (`cl_mem`):**
 - Buffer objects
 - Image objects
- **Associated with context, only implicitly with device**
- **Memory objects can be copied to host memory, from host memory, or to other memory objects**
- **Regions of a memory object can be accessed from host by mapping them into the host address space**

Buffer Object

- One-dimensional array
- Elements are scalars, vectors, or any user-defined structures
- Accessed within device code via pointers

```
__kernel void myKernel(__global int* buffer) {  
    <...>  
    // Access element in buffer object  
    int v = buffer[get_global_id(0)];  
    <...>  
}
```

Image Object



- **Two- or three-dimensional array**
- **Elements are 4-component vectors from a list of predefined formats**
- **Accessed within device code via built-in functions (storage format not exposed to application)**
 - **Sampler objects are used to configure how built-in functions sample images (addressing modes, filtering modes)**
- **Can be created from OpenGL texture or renderbuffer**

Data Transfer between Host and Device

```
int main() {
    cl_context context = CreateContext();
    cl_device_id* device_id = GetDevices(context);
    cl_command_queue command_queue =
    CreateCommandQueue(context, device_id[0]);
    size_t size = 100000 * sizeof(int);
    int* h_buffer = (int*)malloc(size);
    cl_mem* d_buffer = clCreateBuffer(context,
    CL_MEM_READ_WRITE, size, NULL, NULL);
    ... // Initialize host buffer h_buffer
    clEnqueueWriteBuffer(command_queue,
                        d_buffer, CL_FALSE, 0, size, h_buffer, 0, NULL,
    NULL);
    ... // Process device buffer d_buffer
    clEnqueueReadBuffer(command_queue,
                        d_buffer, CL_TRUE, 0, size, h_buffer, 0, NULL,
    NULL);
}
```

Device Code in OpenCL C



- **Derived from ISO C99**

- **A few restrictions: recursion, function pointers, functions in C99 standard headers**
- **Some extensions: built-in variables and functions, function qualifiers, address space qualifiers, e.g:**

```
__global float* a; // Pointer to device memory
```

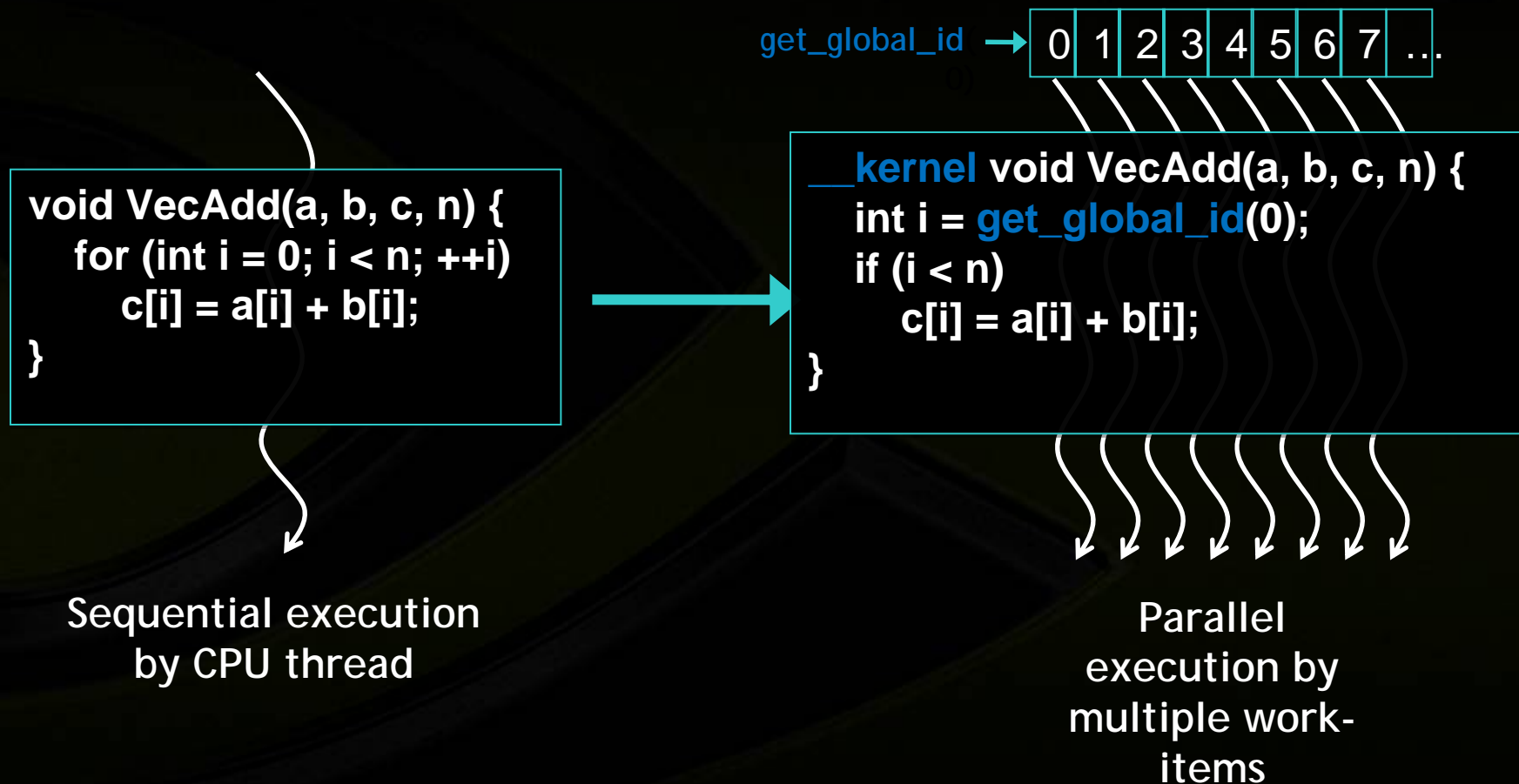
- **Functions qualified by `__kernel` keyword (a.k.a kernels) can be invoked by host code**

```
__kernel void MyKernel() { ... }
```

Kernel Execution: NDRange and Work-Items

- Host code invokes a kernel over an index space called an *NDRange*
 - NDRange = “N-Dimensional Range”
 - NDRange can be a 1-, 2-, or 3-dimensional space
- A single kernel instance at a point in the index space is called a *work-item*
 - Each work-item has a unique global ID within the index space (accessible from device code via `get_global_id()`)
 - Each work-item is free to execute a unique code path

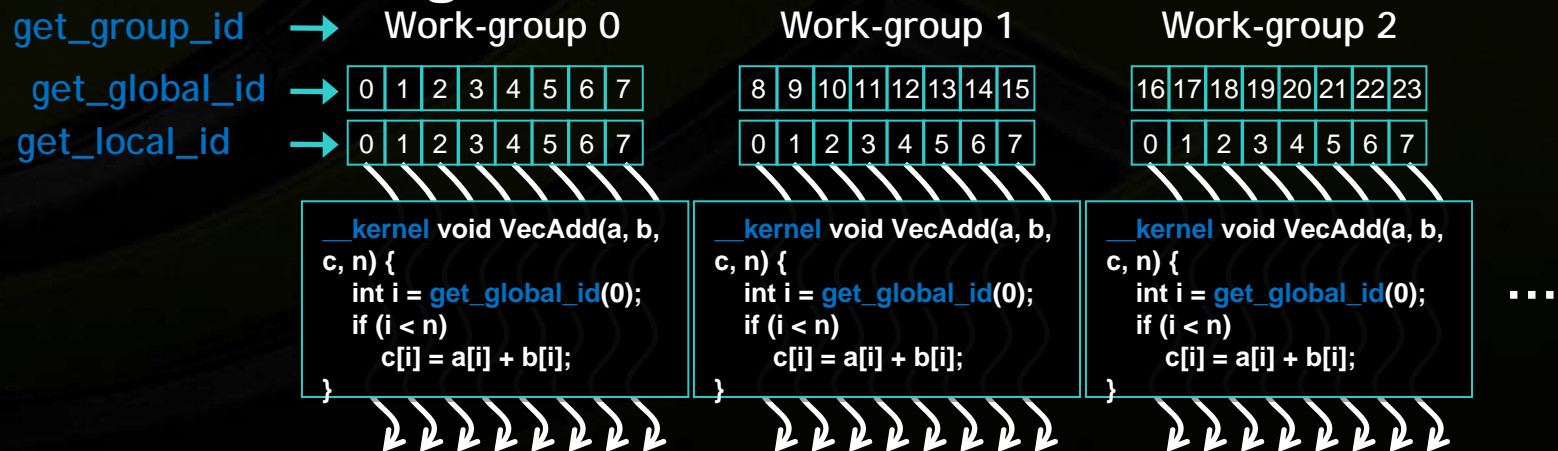
Example: Vector Addition



Kernel Execution: Work-Groups

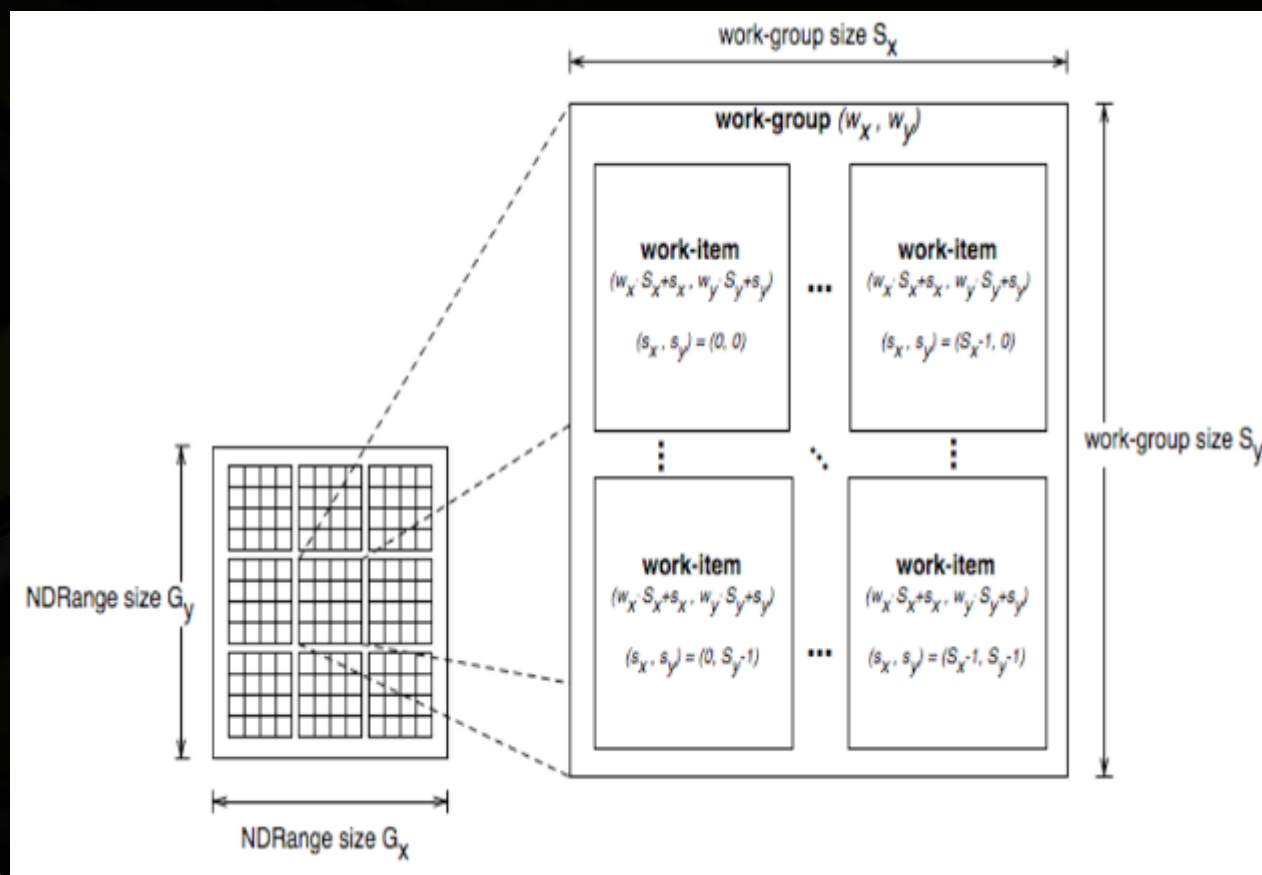


- Work-items are grouped into *work-groups*
 - Each work-group has a unique work-group ID (accessible from device code via `get_group_id()`)
 - Each work-item has a unique local ID within a work-group (accessible from device code via `get_local_id()`)
 - Work-group has same dimensionality as `NDRange`

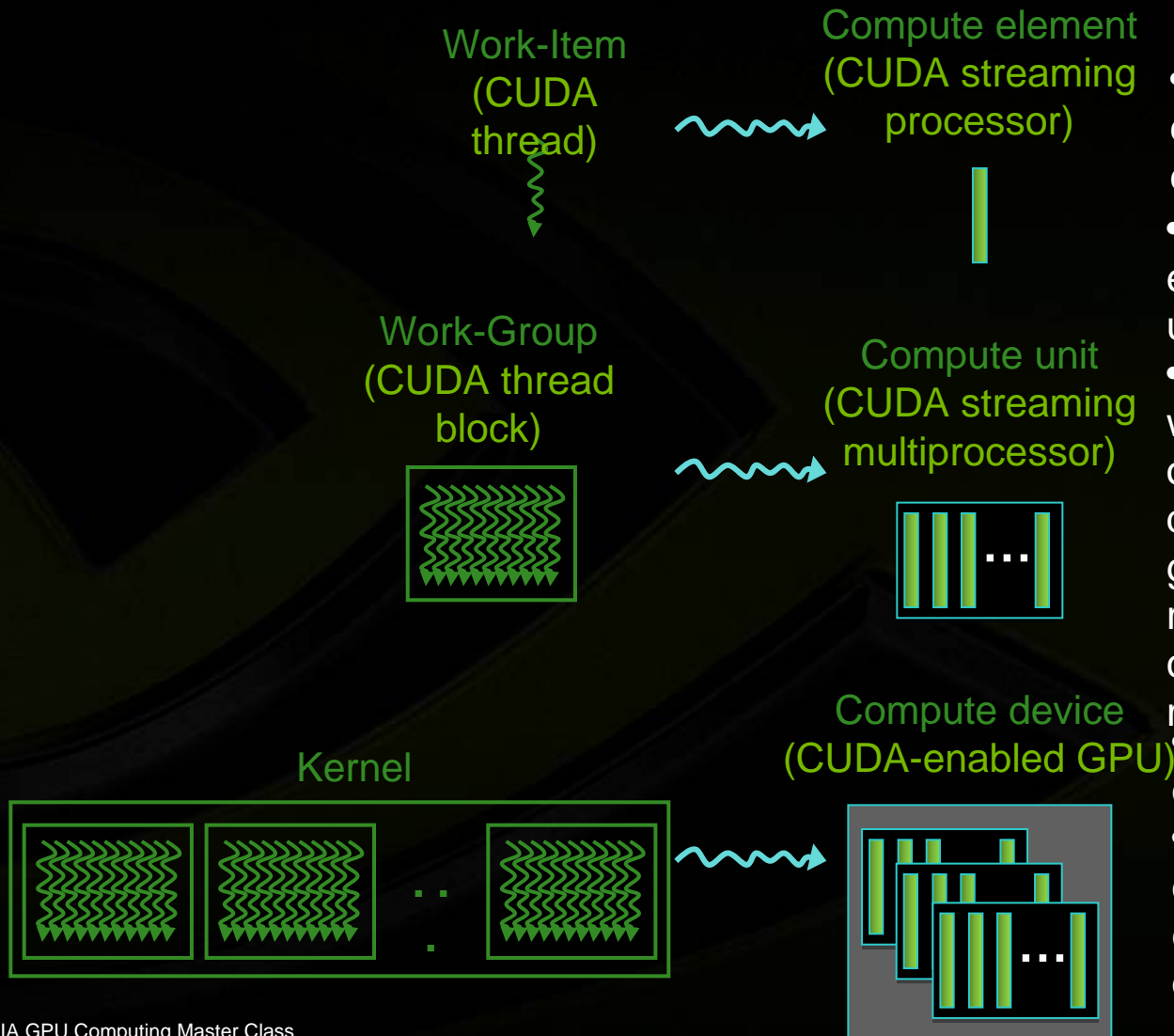


Example of 2D NDRange

- Total number of work-items = $G_x \times G_y$
- Size of each work-group = $S_x \times S_y$
- Number of work-groups = $(G_x / S_x) \times (G_y / S_y)$ (must be dividable)



Kernel Execution on Platform Model



- Each work-item is executed by a compute element
- Each work-group is executed on a compute unit
- Several concurrent work-groups can reside on one compute unit depending on work-group's memory requirements and compute unit's memory resources
- Each kernel is executed on a compute device
- On Tesla architecture, only one kernel can execute on a device at one time

Benefits of Work-Groups

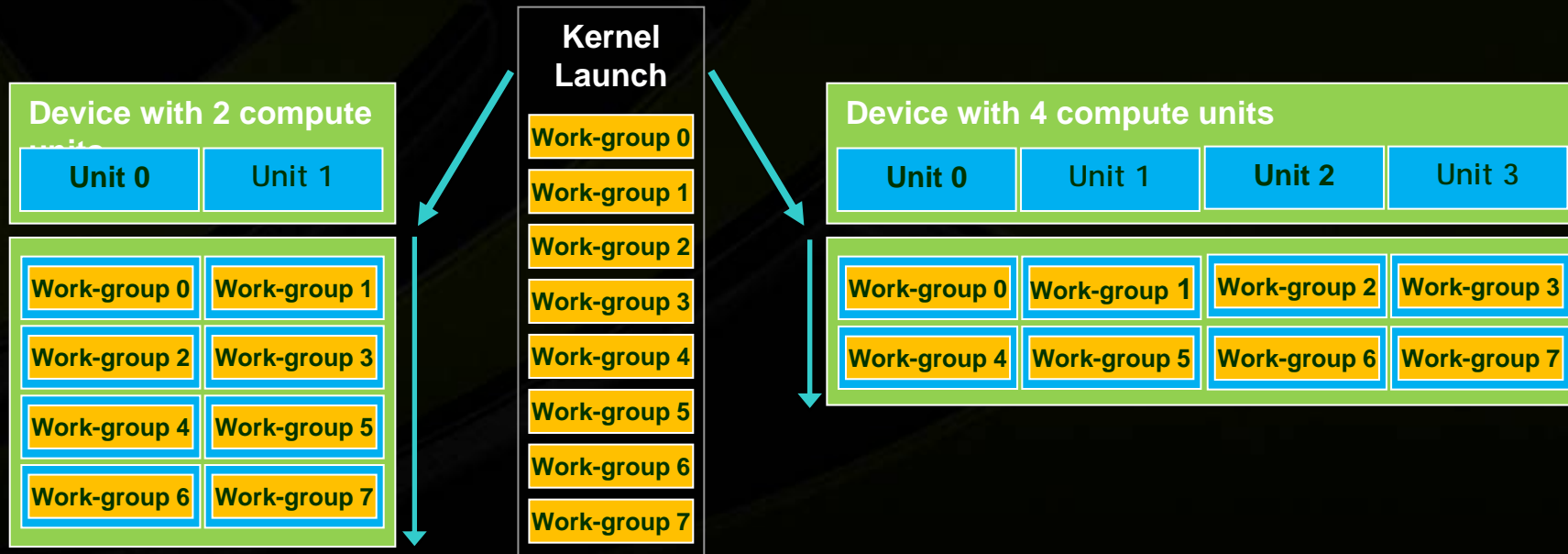


- **Automatic scalability across devices with different numbers of compute units**
- **Efficient cooperation between work-items of same work-group**
 - **Fast shared memory and synchronization**

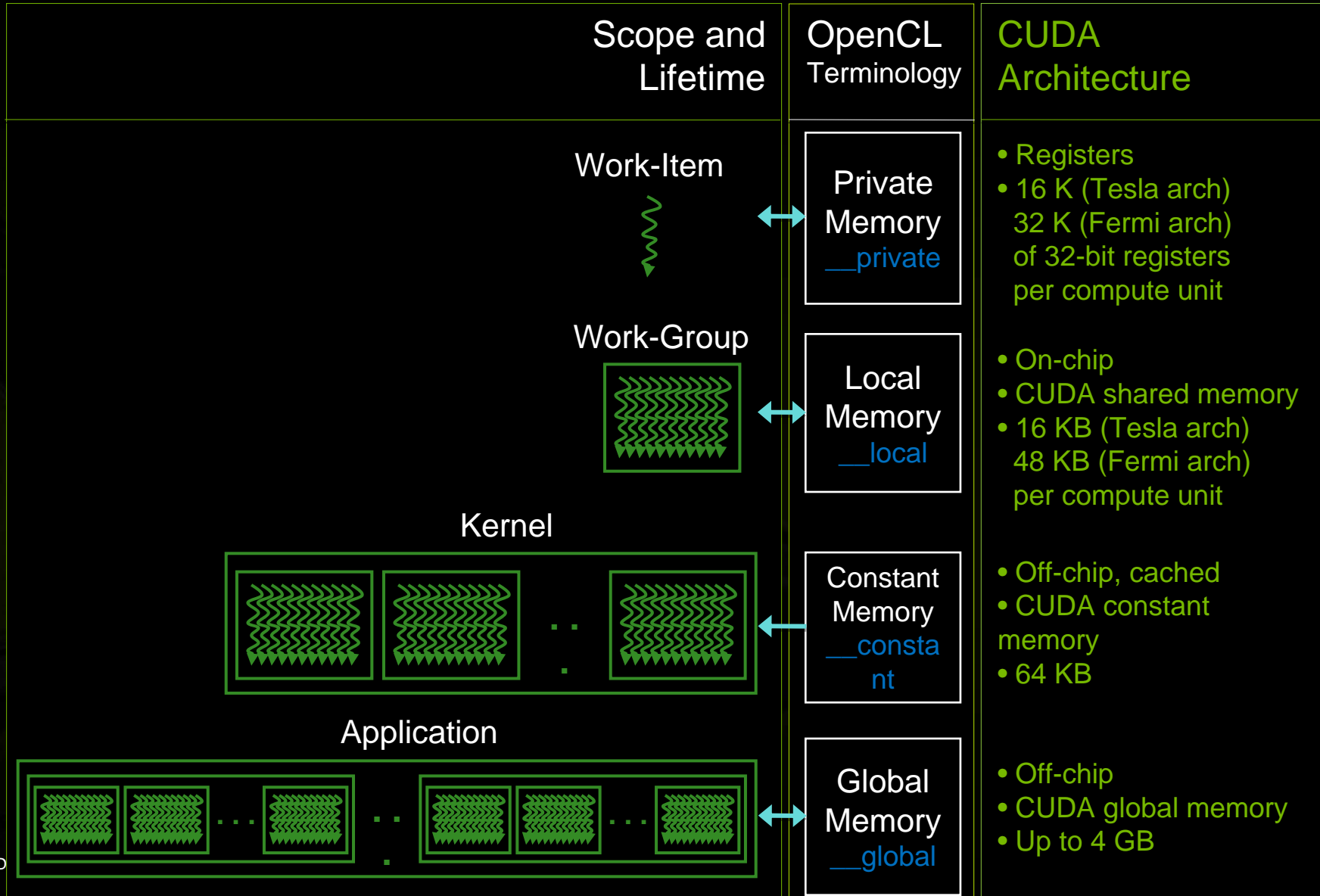
Scalability



- Work-groups can execute in any order, concurrently or sequentially
- This independence between work-groups gives scalability:
 - A kernel scales across any number of compute units



Memory Spaces



Cooperation between Work-Items of same Work-Group

- Built-in functions to order memory operations and synchronize execution:
 - **mem_fence**(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE): waits until all reads/writes to local and/or global memory made by the calling work-item prior to **mem_fence**() are visible to all threads in the work-group
 - **barrier**(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE): waits until all work-items in the work-group have reached this point and calls **mem_fence**(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)
- Used to coordinate accesses to local or global memory shared among work-items

Program and Kernel Objects



- A program object encapsulates some source code (with potentially several kernel functions) and its last successful build
 - `clCreateProgramWithSource()` // Create program from source
 - `clBuildProgram()` // Compile program
- A kernel object encapsulates the values of the kernel's arguments used when the kernel is executed
 - `clCreateKernel()` // Create kernel from successfully compiled // program
 - `clSetKernelArg()` // Set values of kernel's arguments

Kernel Invocation

```
int main() {  
    ... // Create context and command queue, allocate host and device  
        buffers of N elements  
    char* source = "__kernel void MyKernel(__global int* buffer, int N) {\n"  
        "    if (get_global_id(0) < N) buffer[get_global_id(0)] = 7;\n"  
        "}\n "  
    cl_program program = clCreateProgramWithSource(context, 1, &source,  
    NULL, NULL);  
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);  
    cl_kernel kernel = clCreateKernel(program, "MyKernel", NULL);  
    clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_buffer);  
    clSetKernelArg(kernel, 1, sizeof(int), (void*)&N);  
    size_t localWorkSize = 256; // Number of work-items in a work-group  
    int numWorkGroups = (N + localWorkSize - 1) / localWorkSize;  
    size_t globalWorkSize = numWorkGroups * localWorkSize;  
    clEnqueueNDRangeKernel(command_queue, kernel,  
        1, NULL, &globalWorkSize, &localWorkSize, 0,  
    NULL, NULL);  
    ... // Read back buffer  
}
```

NDRange
dimension

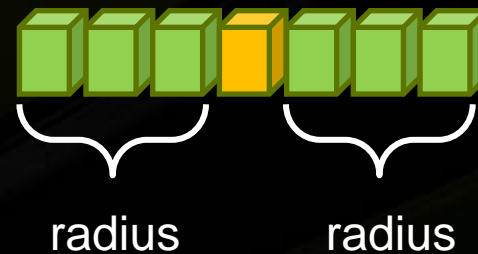
OpenCL Local Memory on the CUDA Architecture

- **On-chip memory (CUDA shared memory)**
 - 2 orders of magnitude lower latency than global memory
 - Order of magnitude higher bandwidth than global memory
 - 16 KB per compute unit on Tesla architecture (up to 30 compute units)
 - 48 KB per compute unit on Fermi architecture (up to 16 compute units)
- **Acts as a user-managed cache to reduce global memory accesses**
- **Typical usage pattern for work-items within a work-group:**
 - Read data from global memory to local memory; synchronize with barrier()
 - Process data within local memory; synchronize with barrier()
 - Write result to global memory

Example of Using Local Memory



- Applying a 1D stencil to a 1D array of elements:
 - Each output element is the sum of all elements within a radius
- For example, for radius = 3, each output element is the sum of 7 input elements:



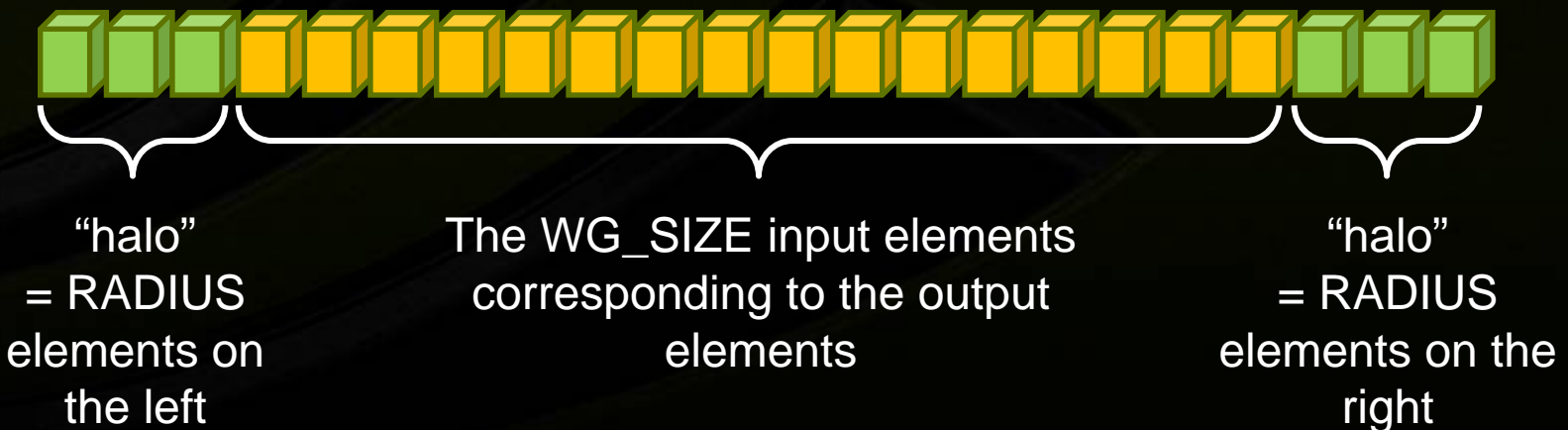
Implementation with Local Memory



- Each work-group outputs one element per work-item, so a total of WG_SIZE output elements

(WG_SIZE = number of work-items per work-group):

- Read $(WG_SIZE + 2 * RADIUS)$ elements from global memory to local memory
- Compute WG_SIZE output elements in local memory
- Write WG_SIZE output elements to global memory



Kernel Code



```
__kernel void stencil(__global int* input,
                    __global int* output) {
    __local int local[WG_SIZE + 2 * RADIUS];
    int i = get_local_id(0) + RADIUS;
    local[i] = input[get_global_id(0)];
    if (get_local_id(0) < RADIUS) {
        local[i - RADIUS] = input[get_global_id(0) - RADIUS];
        local[i + WG_SIZE] = input[get_global_id(0) + WG_SIZE];
    }
    barrier(CLK_LOCAL_MEM_FENCE); // Blocks until work-items are done writing to local
    memory
    int value = 0;
    for (offset = - RADIUS; offset <= RADIUS; ++offset) value += local[i + offset]; // Sum
    output[get_global_id(0)] = value;
}
```

RADIUS = 3

WG_SIZE = 16

Local ID =

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



i =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----



OpenCL C Language Restrictions



- **Pointers to functions are not allowed**
- **Pointers to pointers allowed within a kernel, but not as an argument**
- **Bit-fields are not supported**
- **Variable length arrays and structures are not supported**
- **Recursion is not supported**
- **Writes to a pointer of types less than 32-bit are not supported**
- **Double types are not supported, but reserved**
- **3D Image writes are not supported**

- **Some restrictions are addressed through extensions**

Optional Extensions



- **Extensions are optional features exposed through OpenCL**
- **The OpenCL working group has already approved many extensions that are supported by the OpenCL specification:**
 - **Double precision floating-point types (Section 9.3)**
 - **Built-in functions to support doubles**
 - **Atomic functions (Section 9.5, 9.6, 9.7)**
 - **3D Image writes (Section 9.8)**
 - **Byte addressable stores (write to pointers with types < 32-bits) (Section 9.9)**
 - **Built-in functions to support half types (Section 9.10)**

Performance Overview



- OpenCL is about performance
 - Standard to make use of the massive computing power of parallel processors like GPUs
- But, **performance is generally not portable across devices:**
 - There are multiple ways of implementing a given algorithm in OpenCL. Each can have vastly different performance characteristics for a given compute device!
- Achieving good performance on GPUs requires a basic understanding of GPU architecture

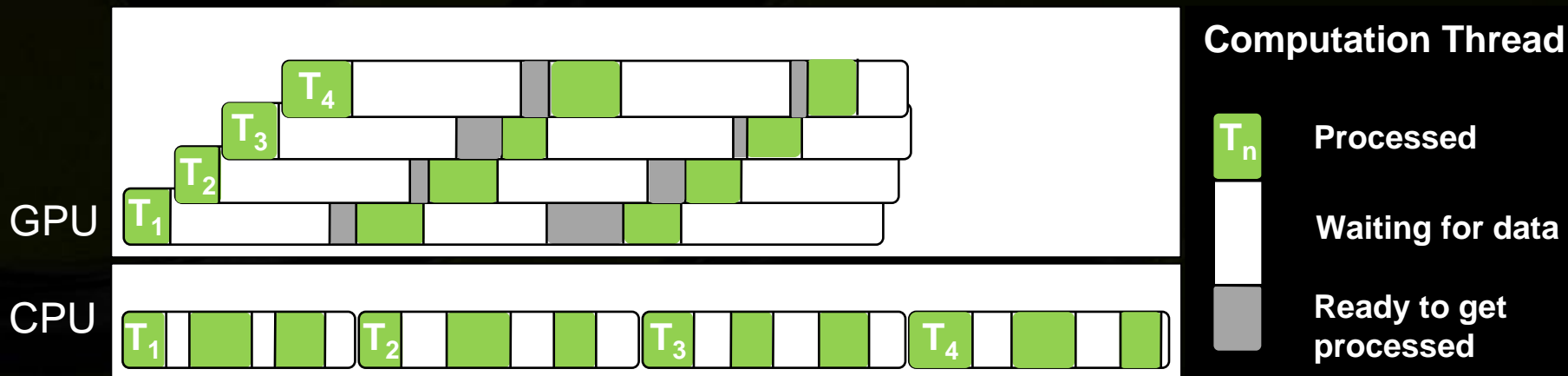
Heterogeneous Computing



- **Host + multiple devices = heterogeneous platform**
- **Distribute workload to:**
 - **Assign to each processor the type of work it does best**
 - **CPU = serial, GPU = parallel**
 - **Keep all processors busy at all times**
 - **Minimize data transfers between processors or hide them by overlapping them with kernel execution**
 - **Overlapping requires data allocated with `CL_MEM_ALLOC_HOST_PTR`**

GPU Computing: Highly Multithreaded

- GPU compute unit “hides” instruction and memory latency with computation
 - Switches from stalled threads to other threads at no cost (lightweight GPU threads)
 - Needs enough concurrent threads to hide latency
 - Radically different strategy than CPU core where memory latency is “reduced” via big caches



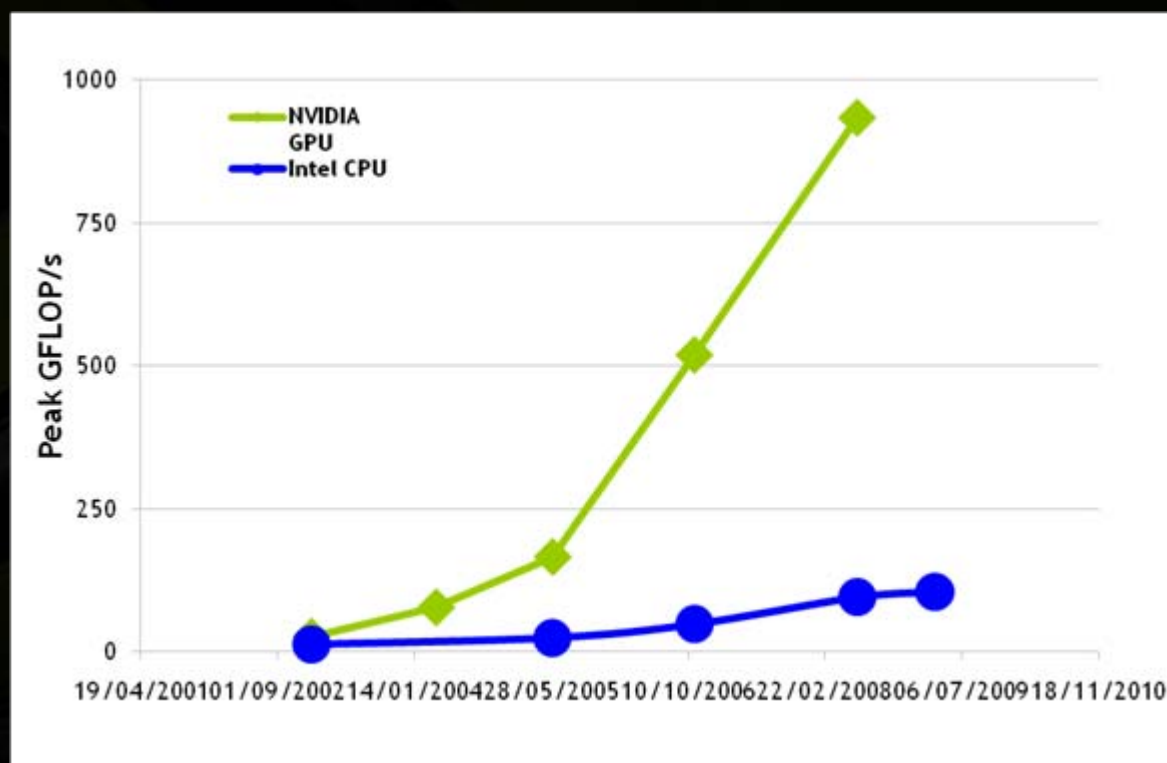


GPU Computing: Highly Multithreaded

- Latency hiding is only possible if there is other work that can be done in parallel
- Therefore, kernels **must** be launched with hundreds of work-items per compute unit for good performance
 - Minimal work-group size of 64; higher is usually better (typically 1.2 to 1.5 speedup)
 - Number of work-groups is typically 100 or more

GPU Computing: High Arithmetic Intensity

- GPU devotes many more transistors than CPU to arithmetic units \Rightarrow high arithmetic intensity



GPU Computing: High Memory Bandwidth

- GPUs offer high memory bandwidth, so applications can take advantage of high arithmetic intensity and achieve high arithmetic throughput



CUDA Memory Optimization



- Memory bandwidth will increase at a slower rate than arithmetic intensity in future processor architectures
- So, maximizing memory throughput is even more critical going forward
- Two important memory bandwidth optimizations:
 - Ensure global memory accesses are **coalesced**
 - **Up to an order of magnitude speedup!**
 - Replace global memory accesses by **shared memory** accesses whenever possible

CUDA = SIMT Architecture



- **Same Instruction Multiple Threads**
 - Threads running on a compute unit are partitioned into groups of 32 threads (warps) in which all threads execute the same instruction simultaneously
- **Minimize divergent branching within a warp**
 - Different code paths within a warp get serialized
- **Remove barrier calls when only threads within same warp need to communicate**
 - Threads within a warp are inherently synchronized

CUDA = Scalar Architecture



- **Use vector types for convenience, not performance**
- **Generally want more work-items rather than large vectors per work-item**

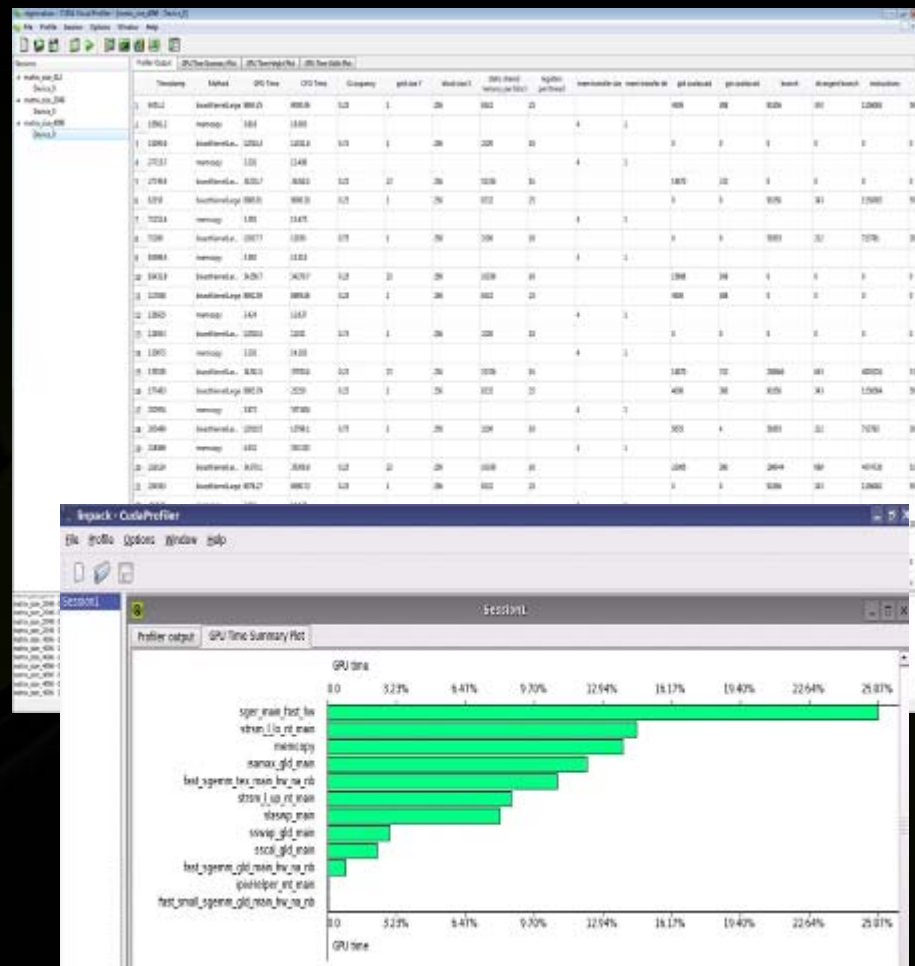
Maximize Instruction Throughput



- Favor **high-throughput instructions**
- Use **native_*()** math functions whenever speed is more important than precision
- Use **-cl-mad-enable** compiler option
 - Enables use of FMADs, which can lead to large performance gains
- Investigate using the **-cl-fast-relaxed-math** compiler option
 - Enables many aggressive compiler optimizations

OpenCL Visual Profiler

- Analyze GPU HW performance signals, kernel occupancy, instruction throughput, and more
- Highly configurable tables and graphical views
- Save/load profiler sessions or export to CSV for later analysis
- Compare results visually across multiple sessions to see improvements
- Supported on Windows and Linux
- Included in the CUDA Toolkit



OpenCL Information and Resources



- **NVIDIA OpenCL Web Page:**
 - http://www.nvidia.com/object/cuda_opencl.html
- **NVIDIA OpenCL Forum:**
 - <http://forums.nvidia.com/index.php?showforum=134>
- **NVIDIA driver, profiler, code samples for Windows and Linux:**
 - <https://nvdeveloper.nvidia.com/object/get-opencl.html>
- **Khronos (current specification):**
 - <http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf>
- **Khronos OpenCL Forum:**
 - http://www.khronos.org/message_boards/viewforum.php?f=28