GPU TECHNOLOGY CONFERENCE

OpenCL on the GPU

San Jose, CA | September 30, 2009

Neil Trevett and Cyril Zeller, NVIDIA

NVIDIA.

Welcome to the OpenCL Tutorial!

- Khronos and industry perspective on OpenCL
  - Neil Trevett
    Khronos Group President
    OpenCL Working Group Chair
    NVIDIA Vice President Mobile Content
- NVIDIA and OpenCL
  - Cyril Zeller
    NVIDIA Manager of Compute Developer Technology

GPU TECHNOLOGY CONFERENCE

OpenCL

Khronos and the OpenCL Standard

Neil Trevett
OpenCL Working Group Chair, Khronos President
NVIDIA Vice President Mobile Content

Who is the Khronos Group?

- Consortium creating open API standards ‘by the industry, for the industry’
  - Non-profit founded nine years ago – over 100 members - any company welcome
- Enabling software to leverage silicon acceleration
  - Low-level graphics, media and compute acceleration APIs
- Strong commercial focus
  - Enabling members and the wider industry to grow markets
- Commitment to royalty-free standards
  - Industry makes money through enabled products – not from standards themselves

Silicon Community

Software Community

GPU TECHNOLOGY CONFERENCE

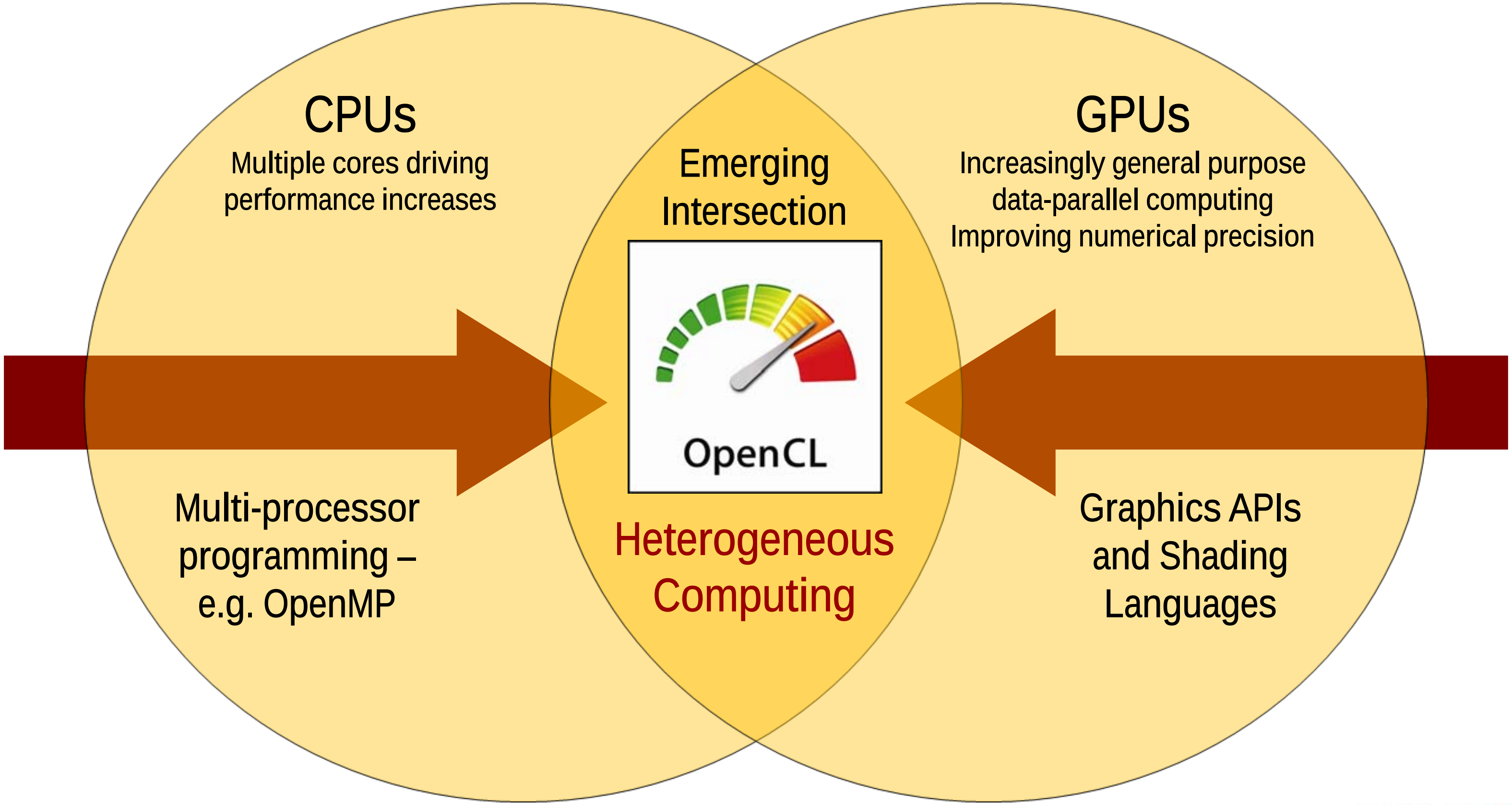
KHRONOS GROUP

Over 100 companies creating authoring and acceleration standards

Board of Promoters

GPU TECHNOLOGY CONFERENCE

Processor Parallelism



OpenCL Commercial Objectives

- Grow the market for parallel computing
- Create a foundation layer for a parallel computing ecosystem
- Enable use of diverse parallel computation resources in a system
- Support a wide diversity of applications
- Application portability across diverse systems from many vendors
- Close coordination with silicon roadmaps
  - OpenCL 1.0 designed to run on current GPU hardware for fast roll-out
  - THEN evolve specification to expose and inspire future silicon capabilities

GPU TECHNOLOGY CONFERENCE

OpenCL Working Group

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers
- Many industry-leading experts involved in OpenCL’s design
  - A healthy diversity of industry perspectives
- Apple made initial proposal and is very active in the working group
  - Serving as specification editor

GPU TECHNOLOGY CONFERENCE

The Khronos API Ecosystem

Hundreds of man years invested by industry experts in coordinated ecosystem

Desktop 3D Ecosystem

COLLADA
3D Asset Interchange Format

OpenGL
Cross platform desktop 3D

OpenCL
Heterogeneous Parallel Programming

Parallel computing and visualization in scientific and consumer applications

Streamlined APIs for mobile and embedded graphics, media and compute acceleration

OpenGL ES
Embedded 3D

OpenMAX
Streaming Media and Image Processing

OpenVG
Vector 2D

OpenSL ES
Enhanced Audio

EGL
Surface and synch abstraction

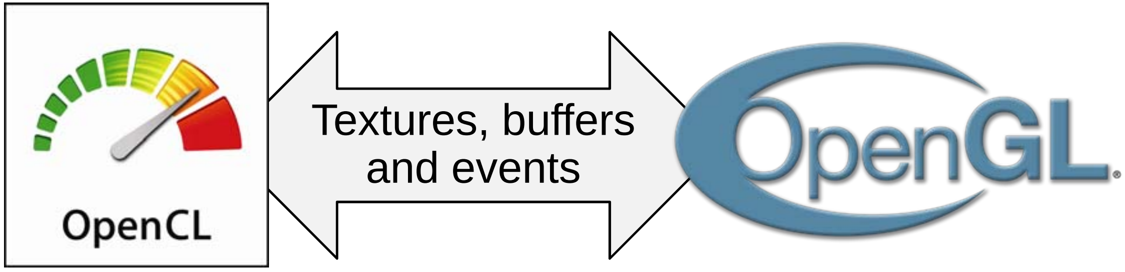
OpenKODE
Mobile OS Abstraction

OpenCL - the center of an visual computing ecosystem with parallel computation, 3D, video, audio and image processing on desktop, embedded and mobile systems

KHRONOS GROUP

OpenCL / OpenGL Interoperability

- OpenCL can efficiently share resources with OpenGL
  - Applications use a graphics or compute API that best fits each part of their problem
- Data is shared, not copied between the two APIs
  - OpenCL objects are created from OpenGL objects
  - Textures, Buffer Objects and Renderbuffers
- Applications can select devices to run OpenGL and OpenCL
  - Efficient queuing of OpenCL and OpenGL commands into the hardware
  - Flexible scheduling and synchronization
  - Works on single GPU and multi-GPU systems



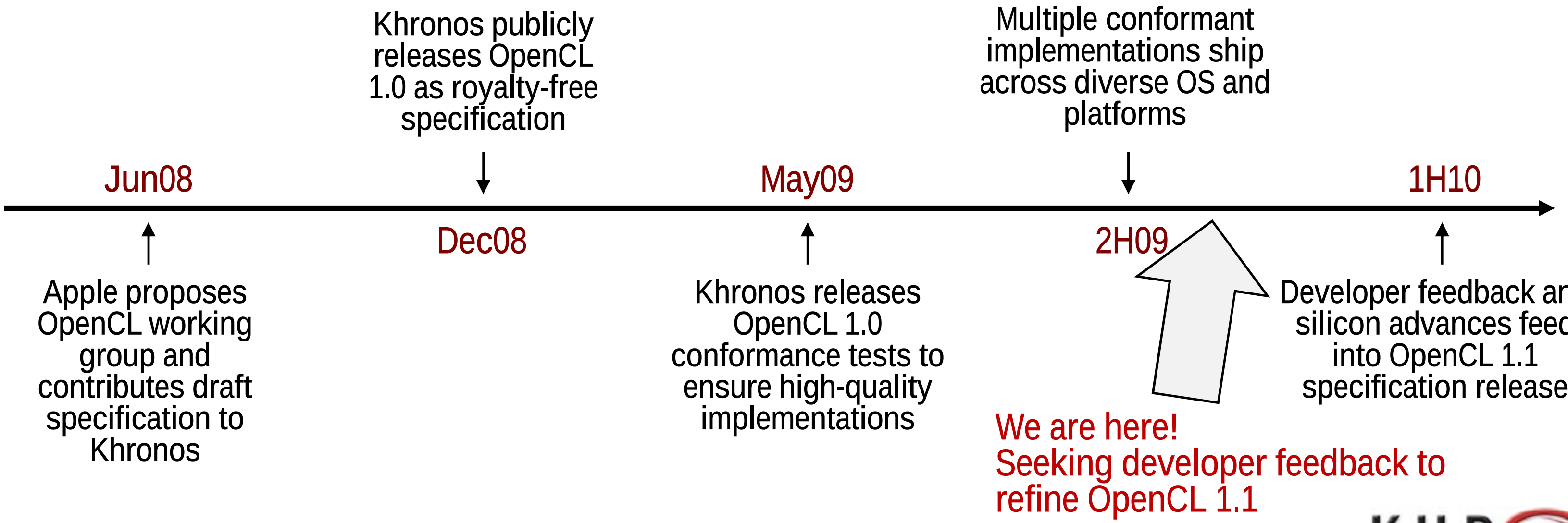
OpenCL 1.0 Embedded Profile

- OpenCL is not just for ‘big iron’
- Embedded profile relaxes some data type and precision requirements
- Intent to enable OpenCL on mobile and embedded silicon in next few years
- Avoids the need for a separate “ES” spec
- Khronos mobile API ecosystem defining tightly interoperable compute, imaging & graphics
- Watch out for OpenCL in mobile phones, automotive, avionics…

A concept GPS phone processes images to recognize buildings and landmarks and uses the internet to supply relevant data

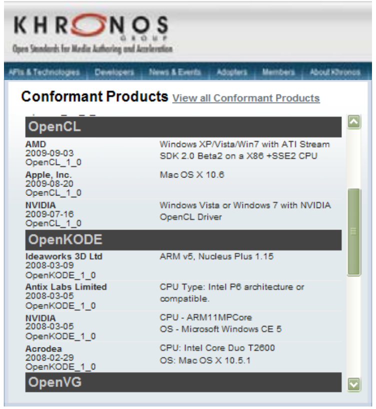
OpenCL Timeline

- Six months from proposal to released OpenCL 1.0 specification
  - Due to a strong initial proposal and a shared commercial incentive
- Apple’s Mac OS X Snow Leopard now ships with OpenCL
  - Improving speed and responsiveness for a wide spectrum of applications



OpenCL Conformance

- A standard without strong testing for conformance is not a standard at all
  - Strengthens consistency of cross-vendor implementations
  - Creates a reliable platform for software developers
- OpenCL has a an exhaustive set of conformance tests
  - Precision and functionality testing
- Khronos Administers an OpenCL Adopters Program
  - Full source access to tests for small fee
  - Peer review of uploaded results by OpenCL working group
- Only passing implementations licensed to use the OpenCL trademark
  - Watch for the OpenCL logo!
  - List of conformant implementations can be found at www.khronos.org



GPU TECHNOLOGY CONFERENCE

KHRONOS GROUP

Khronos OpenCL Resources

- OpenCL is 100% free for developers
  - Download drivers from your silicon vendor
- OpenCL Registry
  - www.khronos.org/registry/cl/
- OpenCL Reference Card
  - PDF version www.khronos.org/files/opencl-quick-reference-card.pdf
  - Pick up your physical copy today!
  - Man pages coming soon!
- OpenCL Developer Forums
  - www.khronos.org/message_boards/
  - Give us your feedback!

The Industry Impact of OpenCL

- OpenCL
  - Multi-vendor, royalty free API for heterogeneous parallel programming
- For software developers
  - More programming choice to tap the power of parallel computing
  - Ecosystem foundation for a wider choice of parallel tools, libraries, middleware
- For silicon vendors and OEMs
  - Catalyze a wide range of software and tools to drive hardware demand
- .. and most importantly - end-users will benefit
  - A wide range of innovative parallel computing applications

- **If this is relevant to your company please join Khronos and have a voice in OpenCL's evolution!**

GPU TECHNOLOGY CONFERENCE

OpenCL on the GPU

San Jose, CA | September 30, 2009

Cyril Zeller

NVIDIA.

GPU TECHNOLOGY CONFERENCE

Outline

- General considerations
- API overview
- Performance primer
- Next steps

GPU TECHNOLOGY CONFERENCE

NVIDIA's OpenCL Timeline

Dec08

NVIDIA shows 1st operable OpenCL demo on GPU (SIGGRAPH Asia)

Khronos releases OpenCL 1.0 specification

Apr09

NVIDIA releases drivers and code samples, to developers

May09

NVIDIA submits 1st GPU implementation for conformance

Khronos releases OpenCL 1.0 conformance tests

Jun09

NVIDIA releases 1st conformant GPU implementation for Windows and Linux

Sep09

NVIDIA releases OpenCL Visual Profiler

NVIDIA drivers, code samples, and programming guides, now publicly available at developer.nvidia.com/object/get-opencl.html

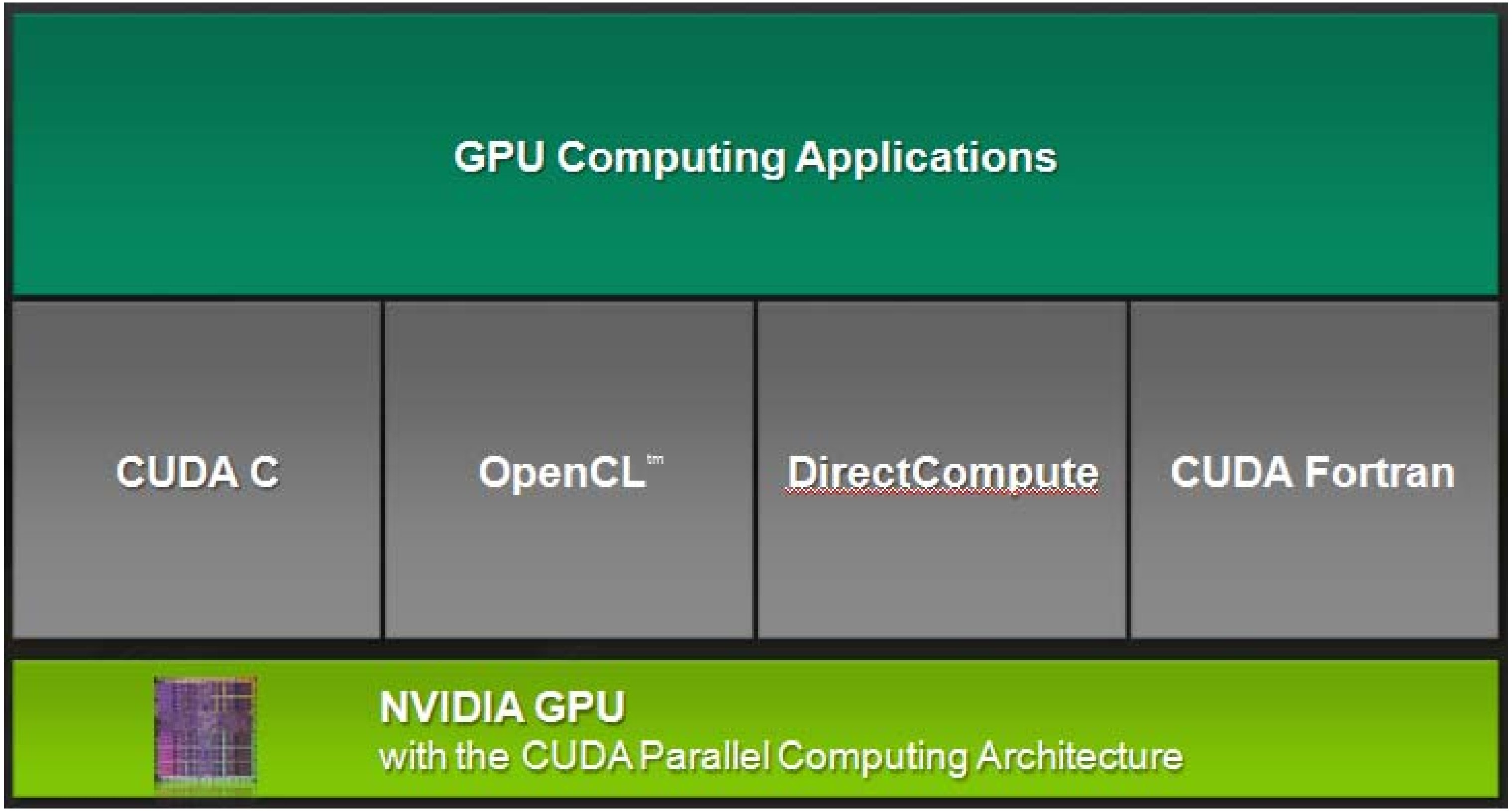
NVIDIA.

OpenCL and the CUDA Architecture

Application Innovation

Development Environment

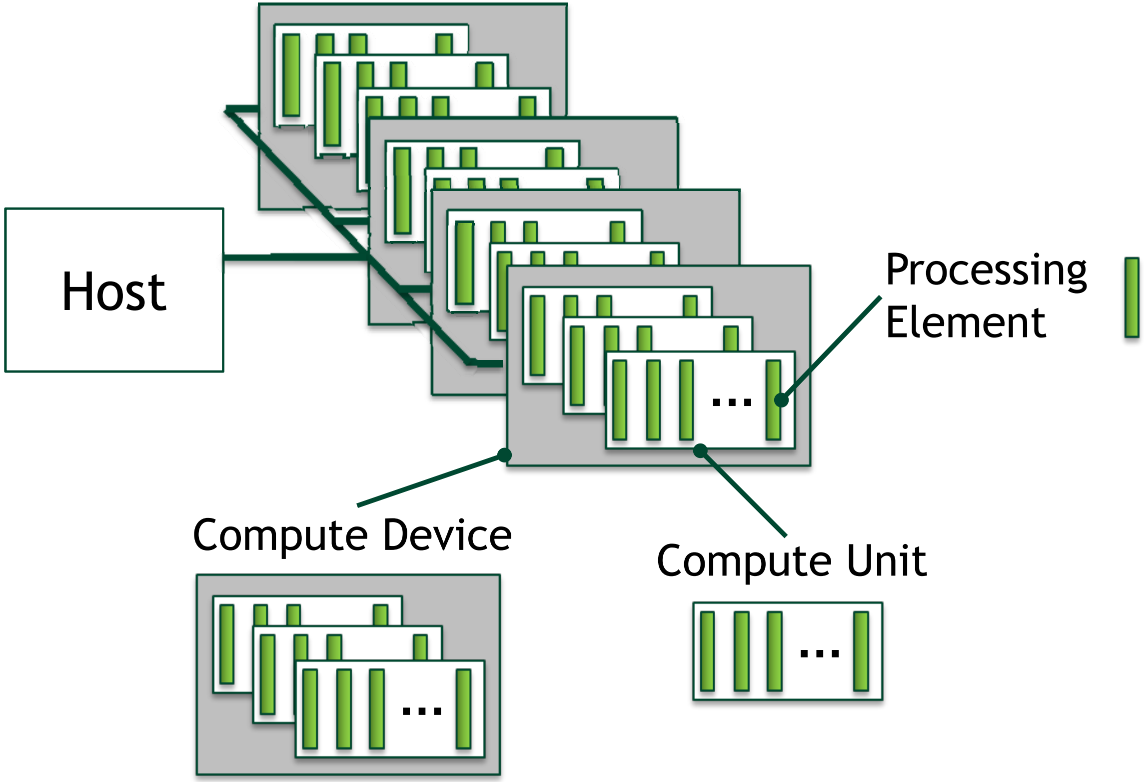
Leading Edge GPU Hardware



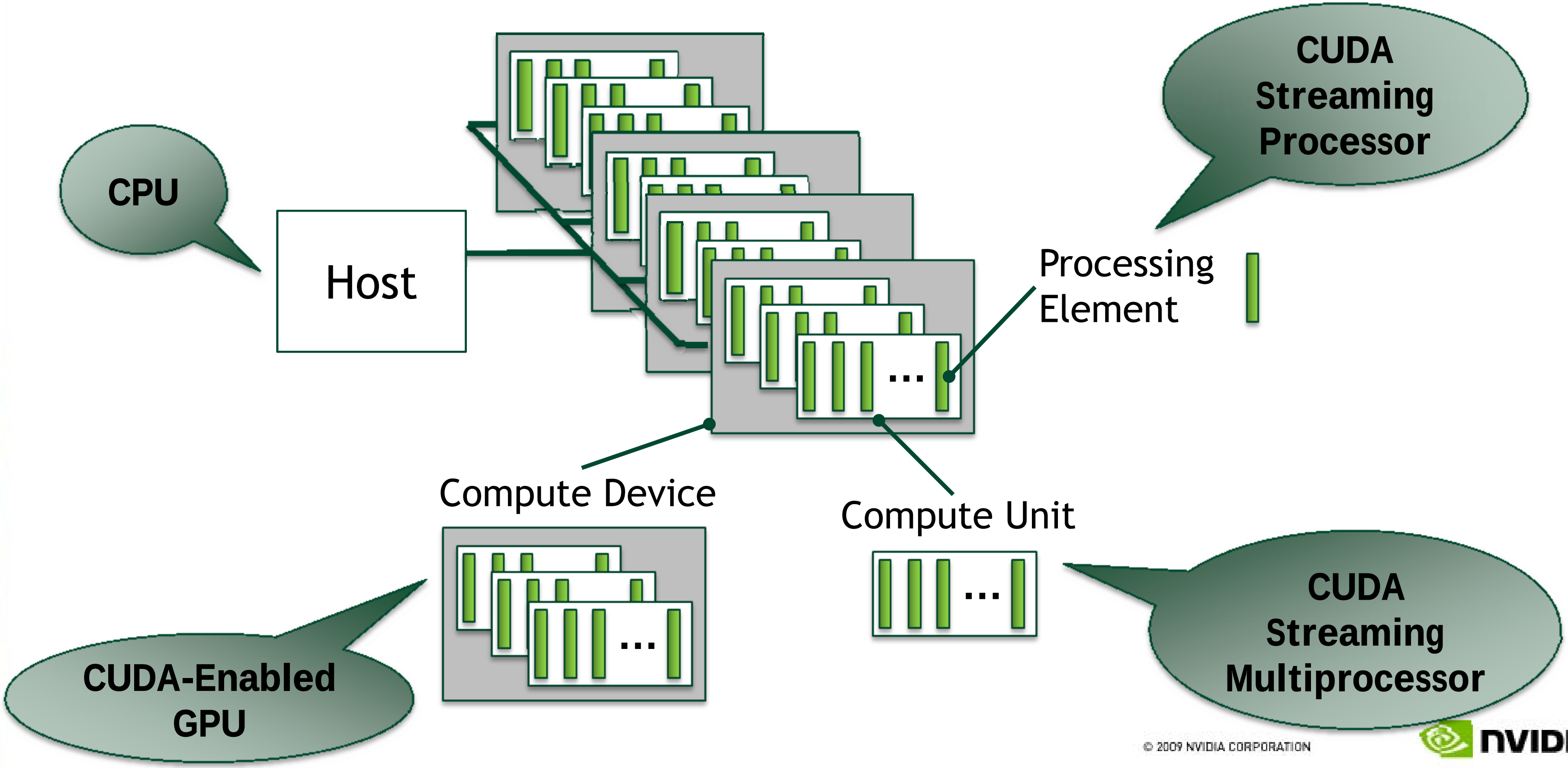
OpenCL Portability

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

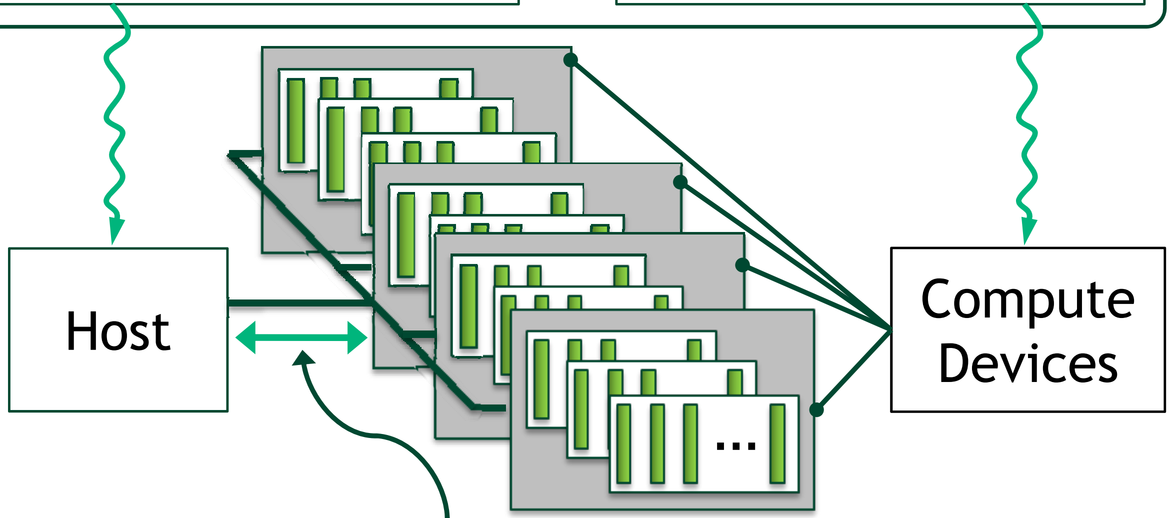


Anatomy of an OpenCL Application

OpenCL Application

Host Code
- Written in C/C++
- Executes on the host

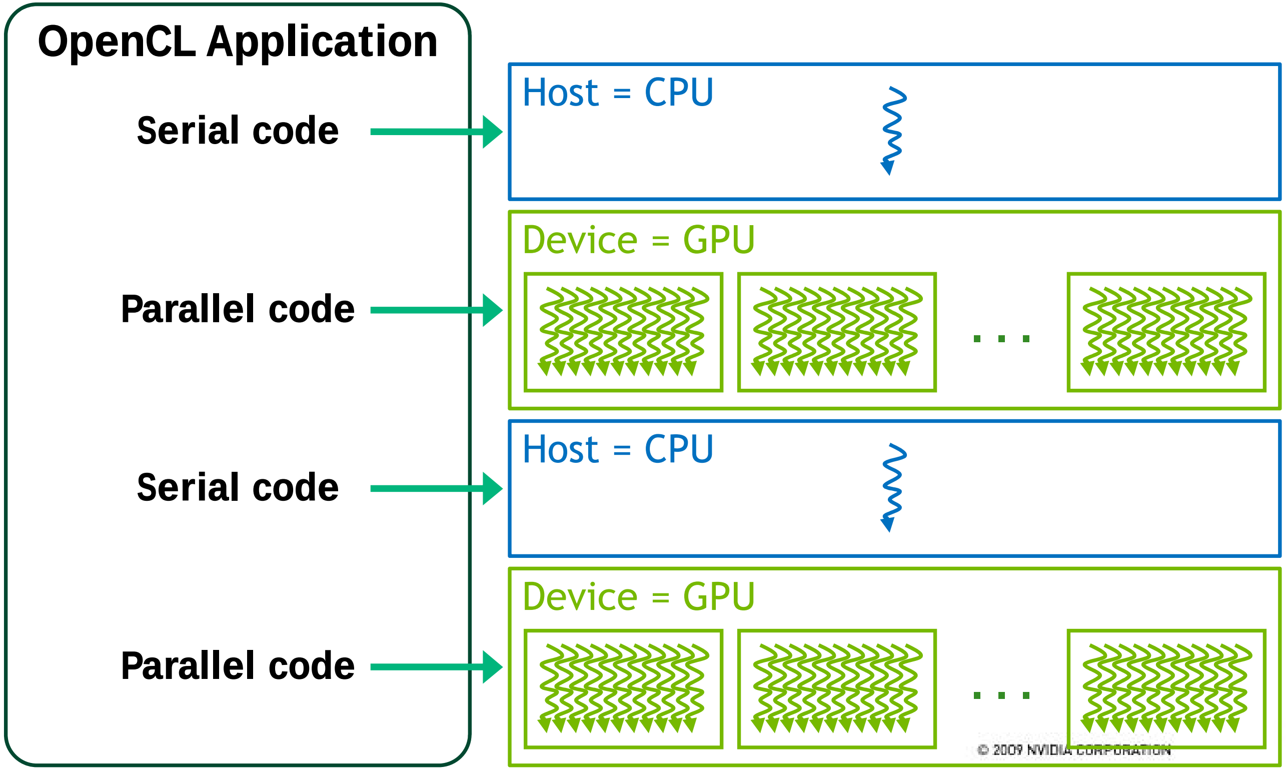
Device Code
- Written in OpenCL C
- Executes on the device



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
  - Platform query and context creation
- Runtime
  - Memory management and command execution within a context
- Compiler for OpenCL C

Platform Layer

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

Context =
cl_context

One or more devices
cl_device_id

Memory and device code shared by these devices
cl_mem cl_program

Command queues to send commands to these devices
cl_command_queue

Error Handling, Resource Deallocation

- Error handling:
  - All host functions return an error code
  - Context error callback
- 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
cl_context* CreateContext() {
    return clCreateContextFromType(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, cb, device_id, NULL);
    return device_id;
}
```

Error callback

User data

Error code

Runtime

- Command queues creation and management
- Device 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
  - Event object can be queried to track execution status of associated command and get profiling information

Memory Objects

- Two types of memory objects (cl_mem):
  - **Buffer** objects
  - **Image** objects
- 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

GPU TECHNOLOGY CONFERENCE

Buffer Object

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

NVIDIA.

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

```
void VecAdd(a, b, c, n) {
    for (int i = 0; i < n; ++i)
        c[i] = a[i] + b[i];
}
```

Sequential execution by CPU thread

NDRange

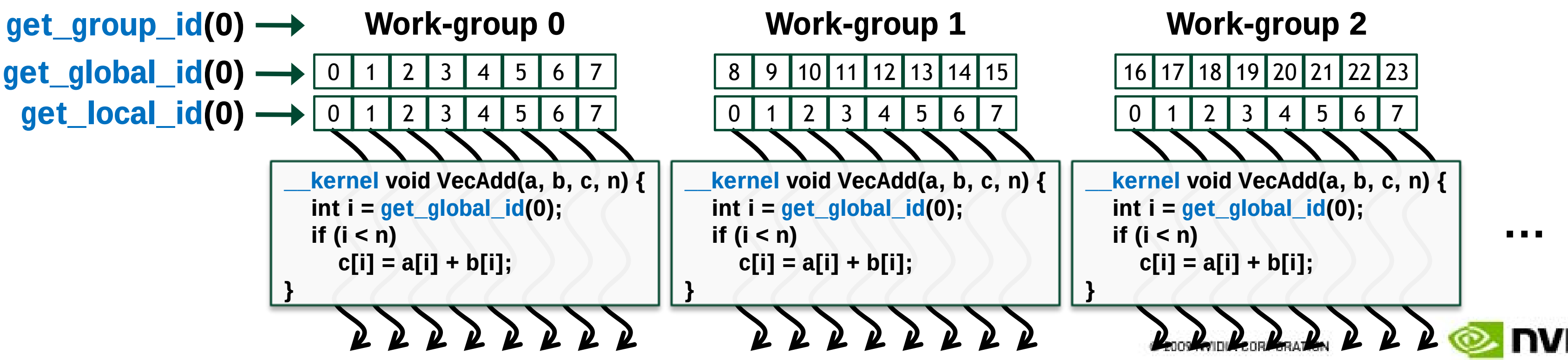
get_global_id(0) → | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |

```
__kernel void VecAdd(a, b, c, n) {
    int i = get_global_id(0);
    if (i < n)
        c[i] = a[i] + b[i];
}
```

Parallel execution by multiple work-items

Kernel Execution: Work-Groups

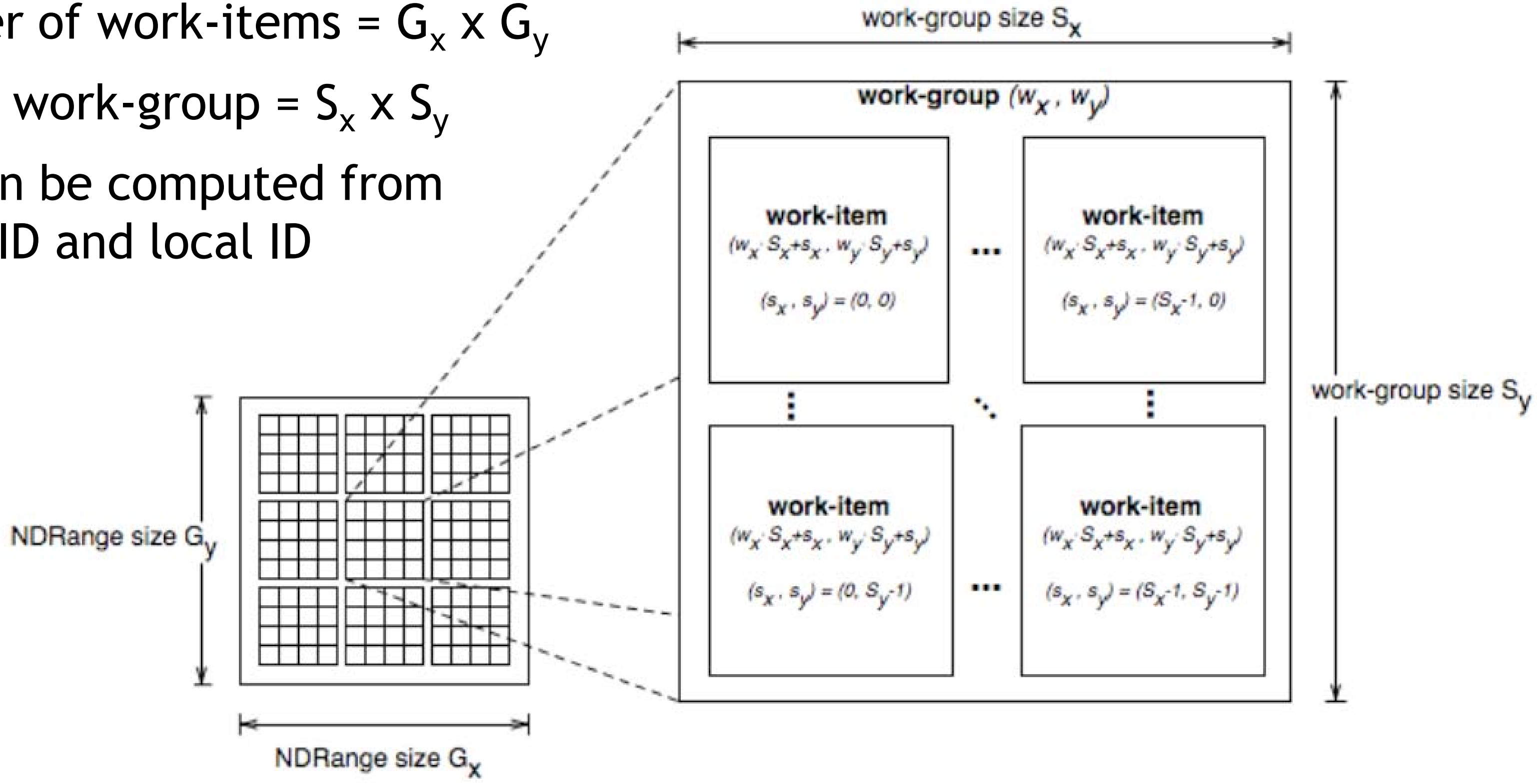
- Work-items are grouped into *work-groups*
  - Each work-group has a unique work-group ID (accessible from device code via get_group_id())
  - Each work-item has a unique local ID within a work-group (accessible from device code via get_local_id())
  - Work-group has same dimensionality as NDRange



GPU TECHNOLOGY CONFERENCE

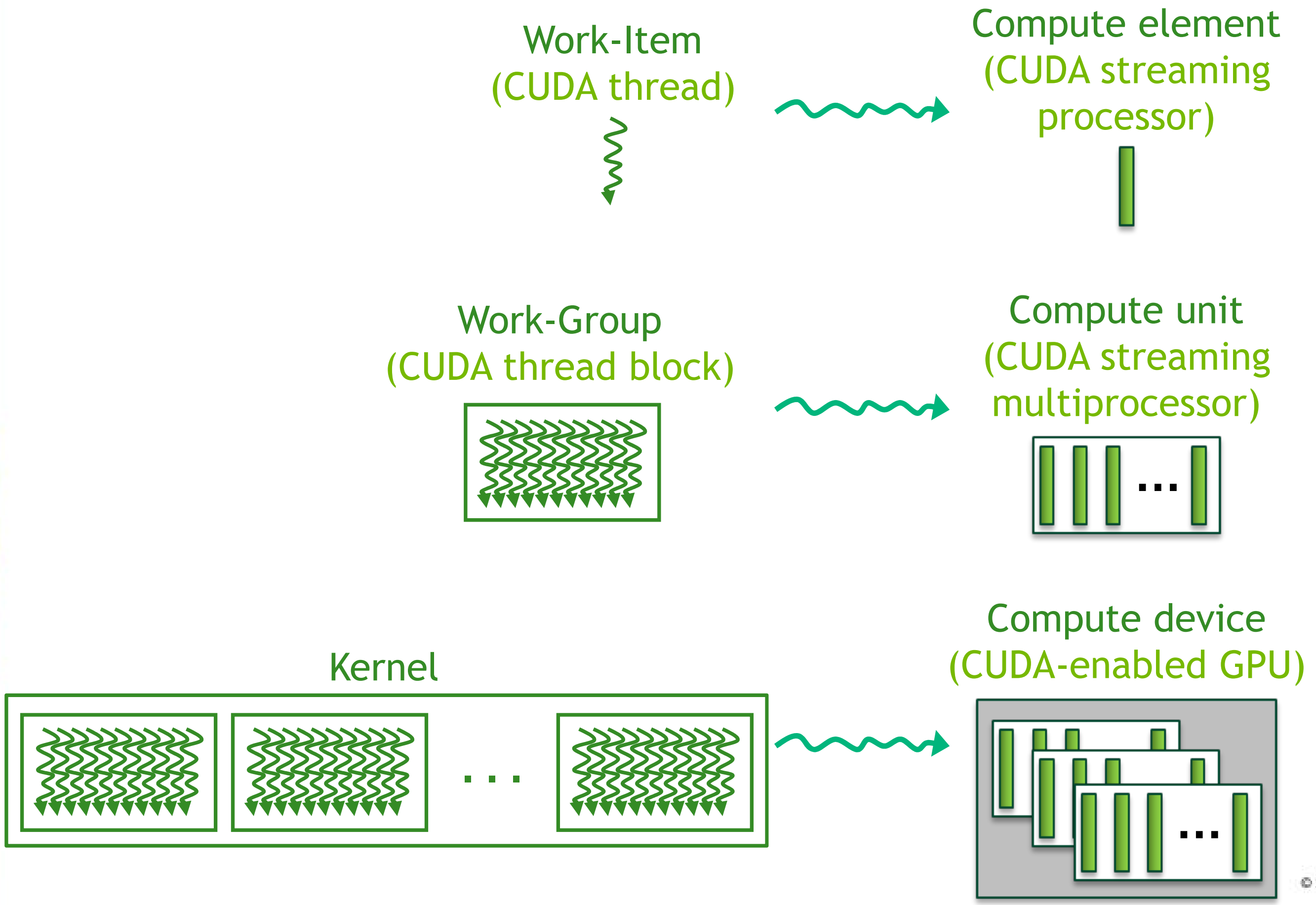
Example of 2D NDRange

- Total number of work-items = $G_x \times G_y$
- Size of each work-group = $S_x \times S_y$
- Global ID can be computed from work-group ID and local ID



GPU TECHNOLOGY CONFERENCE

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

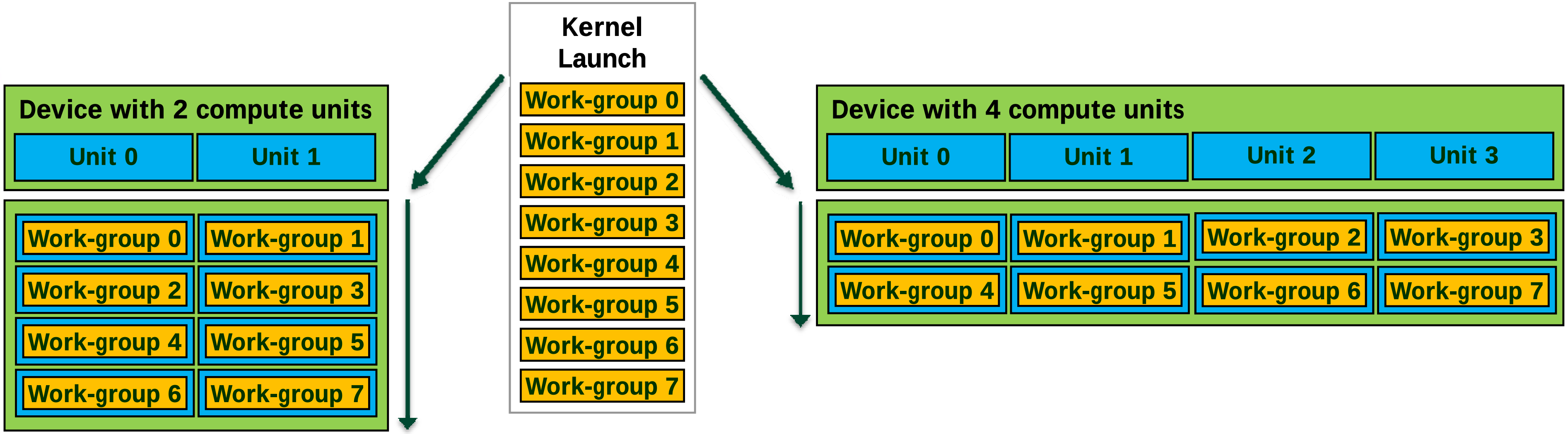
NVIDIA.

Benefits of Work-Groups

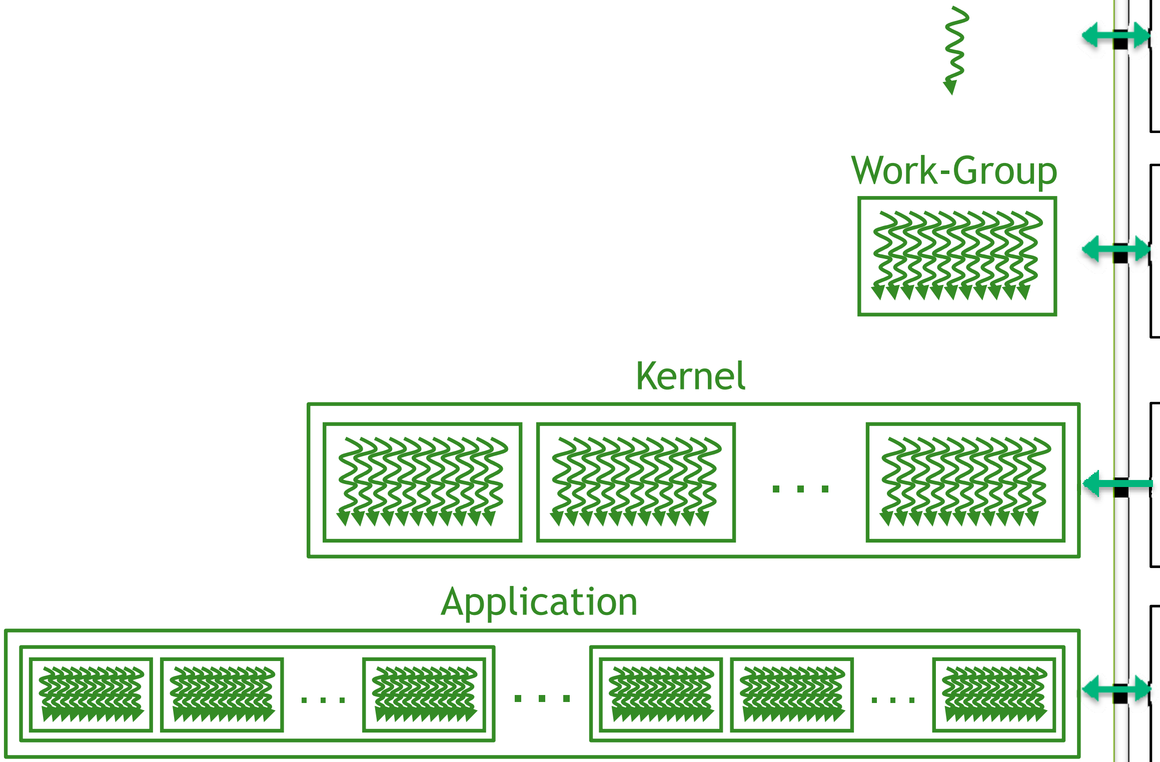
- Automatic scalability across devices with different numbers of compute units
- Efficient cooperation between work-items of same work-group
  - Fast shared memory and synchronization

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



Scope and Lifetime	OpenCL Terminology	CUDA Architecture
Work-Item	Private Memory __private	• Registers • 16 K (Tesla arch) 32 K (Fermi arch) of 32-bit registers per compute unit
Work-Group	Local Memory __local	• On-chip • CUDA shared memory • 16 KB (Tesla arch) 48 KB (Fermi arch) per compute unit
Kernel	Constant Memory __constant	• Off-chip, cached • CUDA constant memory • 64 KB
Application	Global Memory __global	• Off-chip • CUDA global memory • Up to 4 GB

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

GPU TECHNOLOGY CONFERENCE

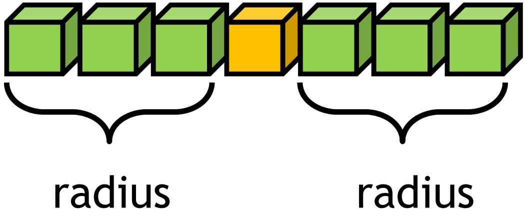
NVIDIA.

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

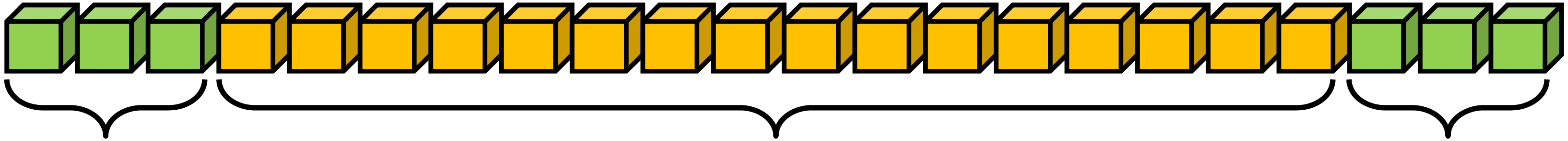
Example of Using Local Memory

- 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



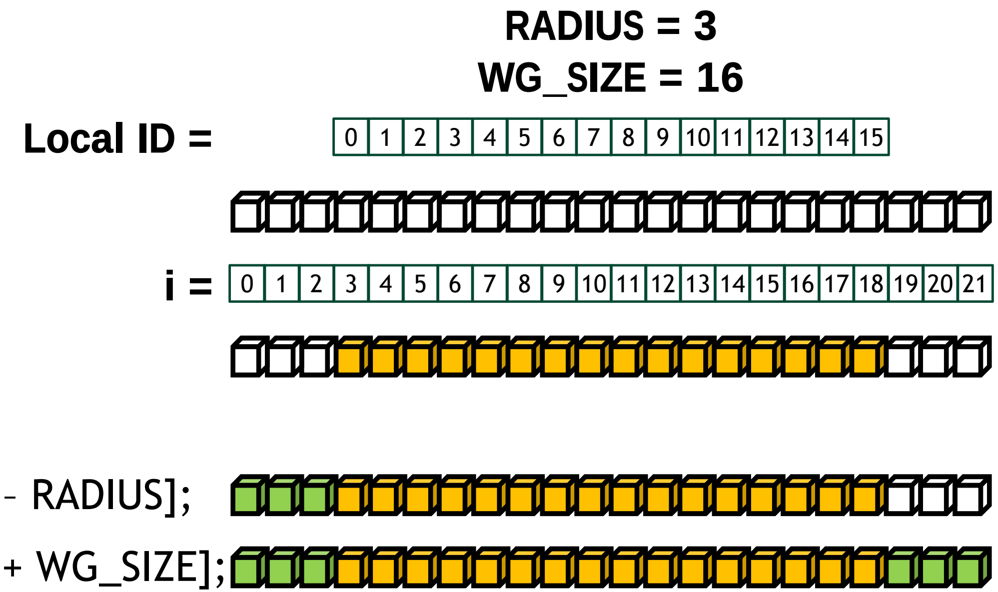
“halo”
= RADIUS elements
on the left

The WG_SIZE input elements
corresponding to the output elements

“halo”
= RADIUS elements
on the right

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); // Ensure all work-items are done writing to local memory
   int value = 0;
   for (offset = - RADIUS; offset <= RADIUS; ++offset)
      value += local[i + offset];
   output[get_global_id(0)] = value;          }
```



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

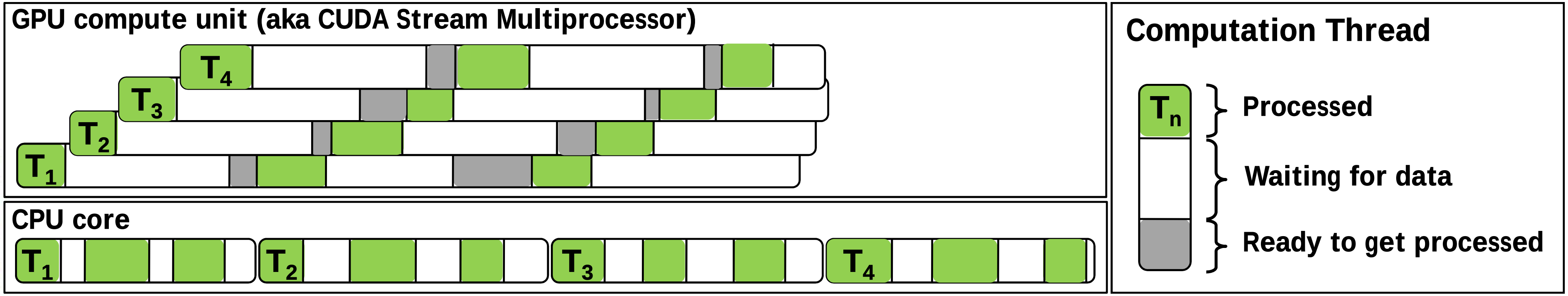
- OpenCL is about performance
  - Giving software developers access to the massive computing power of parallel processors like GPUs
- But, performance is generally not portable across devices:
  - There are multiple ways of implementing a given algorithm in OpenCL and these multiple implementations can 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

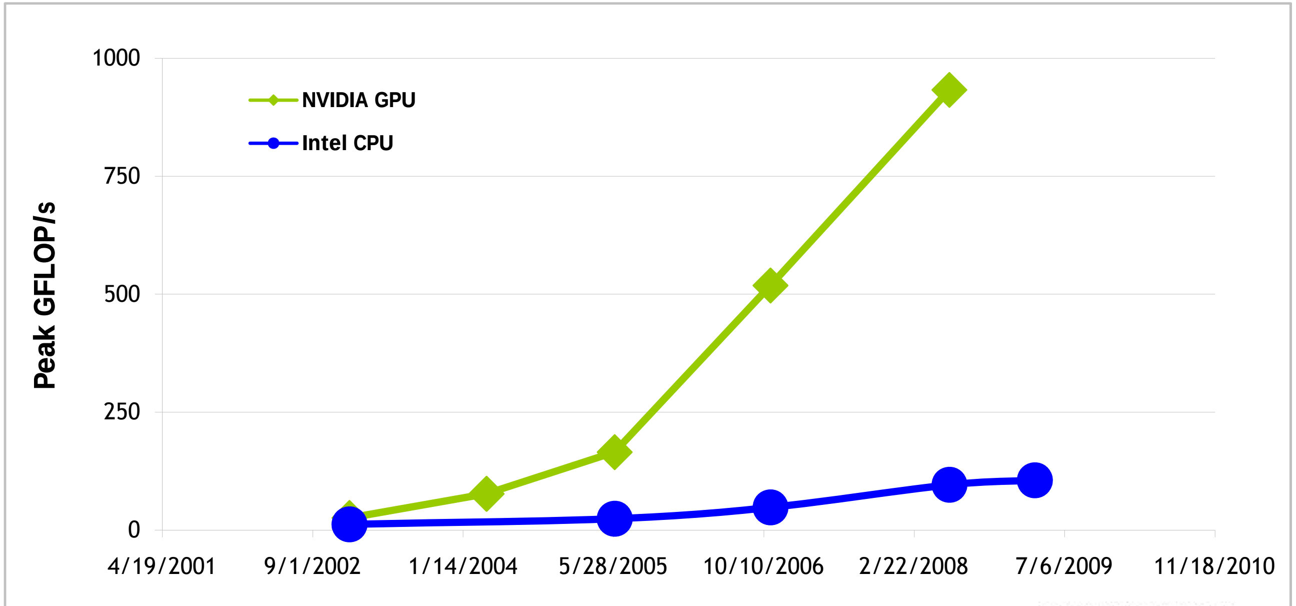


- Therefore, kernels must be launched with hundreds of work-items per compute unit for good performance
  - Minimal work-group size of 64; higher is usually better (typically 1.2 to 1.5 speedup)
  - Number of work-groups is typically 100 or more

GPU TECHNOLOGY CONFERENCE

GPU Computing: High Arithmetic Intensity

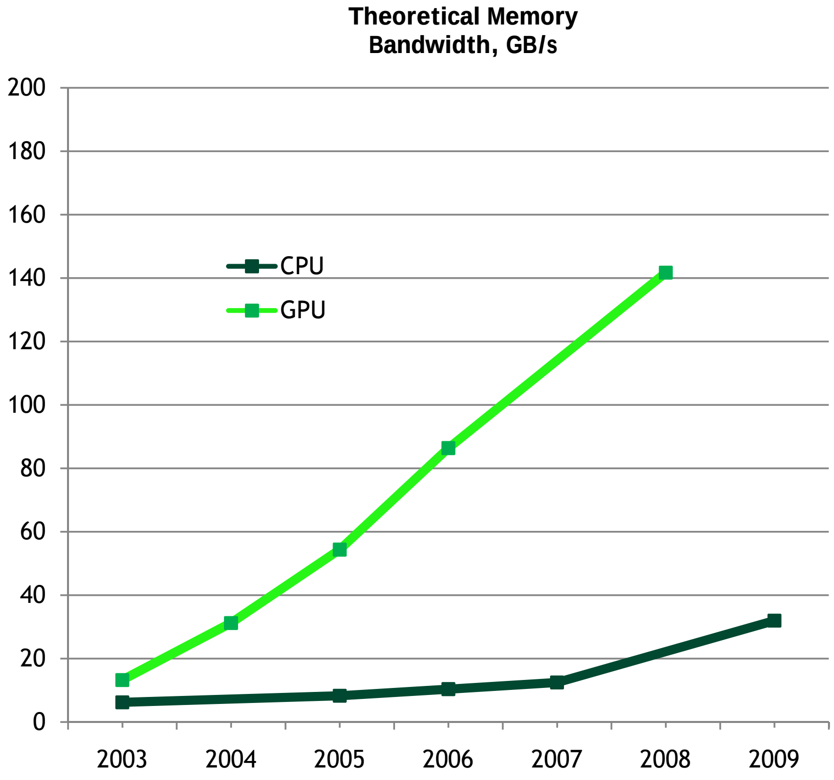
- GPU devotes many more transistors than CPU to arithmetic units ⇒ high arithmetic intensity



GPU TECHNOLOGY CONFERENCE

GPU Computing: High Memory Bandwidth

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



GPU TECHNOLOGY CONFERENCE

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 (called **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

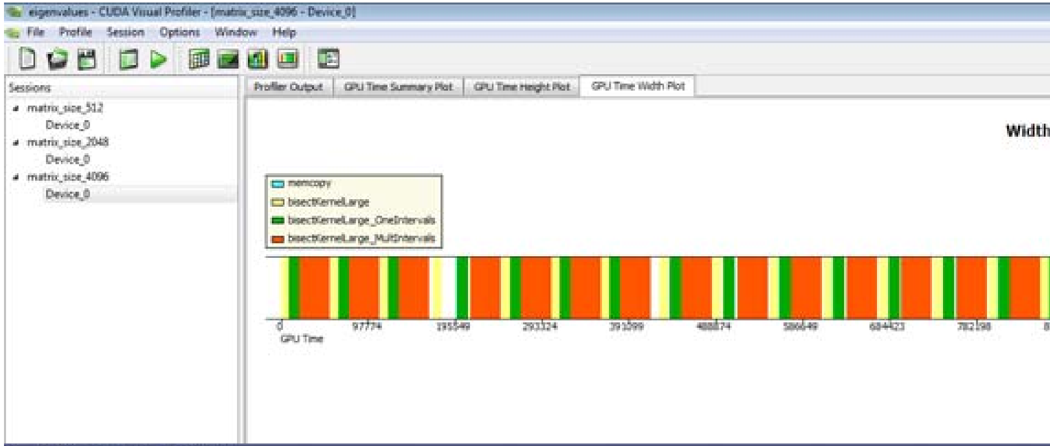
- Use vector types for convenience, not performance
- Generally want more work-items rather than large vectors per work-item

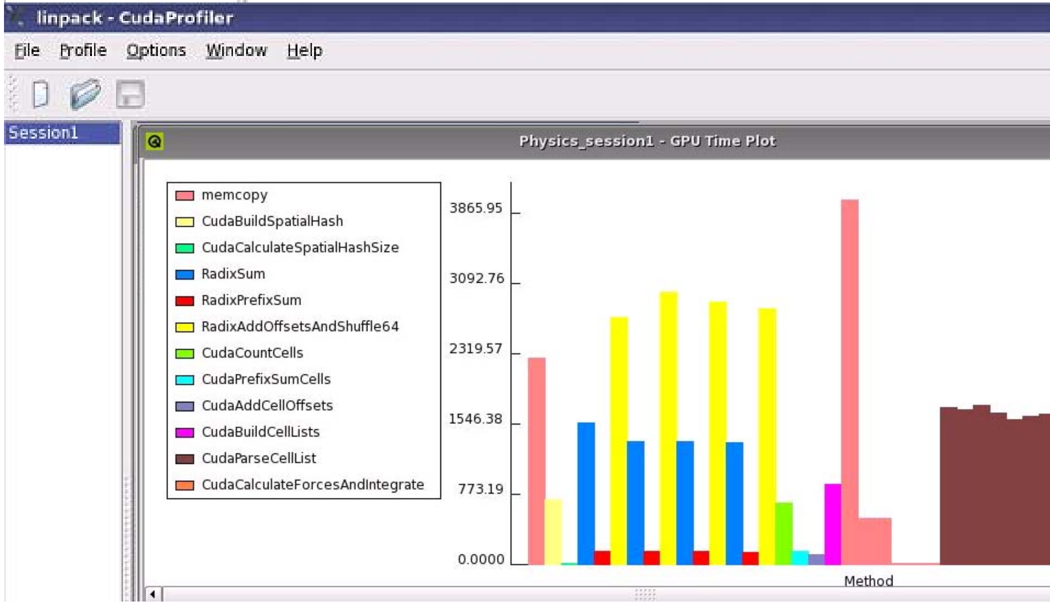
Maximize Instruction Throughput

- 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





Next Steps

- Begin hands-on development with our publicly available OpenCL driver and GPU Computing SDK
- Read OpenCL Specification and extensive documentation provided with the SDK
- Read and contribute to OpenCL forums at Khronos and NVIDIA
- Attend these GTC talks:
  - "The Art of Debugging for the CUDA Architecture" on Thursday @ 5 PM
  - "NEXUS: A Powerful IDE for GPU Computing on Windows " on Friday @ 1 PM
  - "OpenCL Optimization" on Friday @ 2 PM

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-1.0.43.pdf
- Khronos OpenCL Forum:
  - http://www.khronos.org/message_boards/viewforum.php?f=28