A Jump Start to OpenCL

Another Language to Program Parallel Computing Devices

March 15, 2009

CIS 565/665 – GPU Computing and Architecture

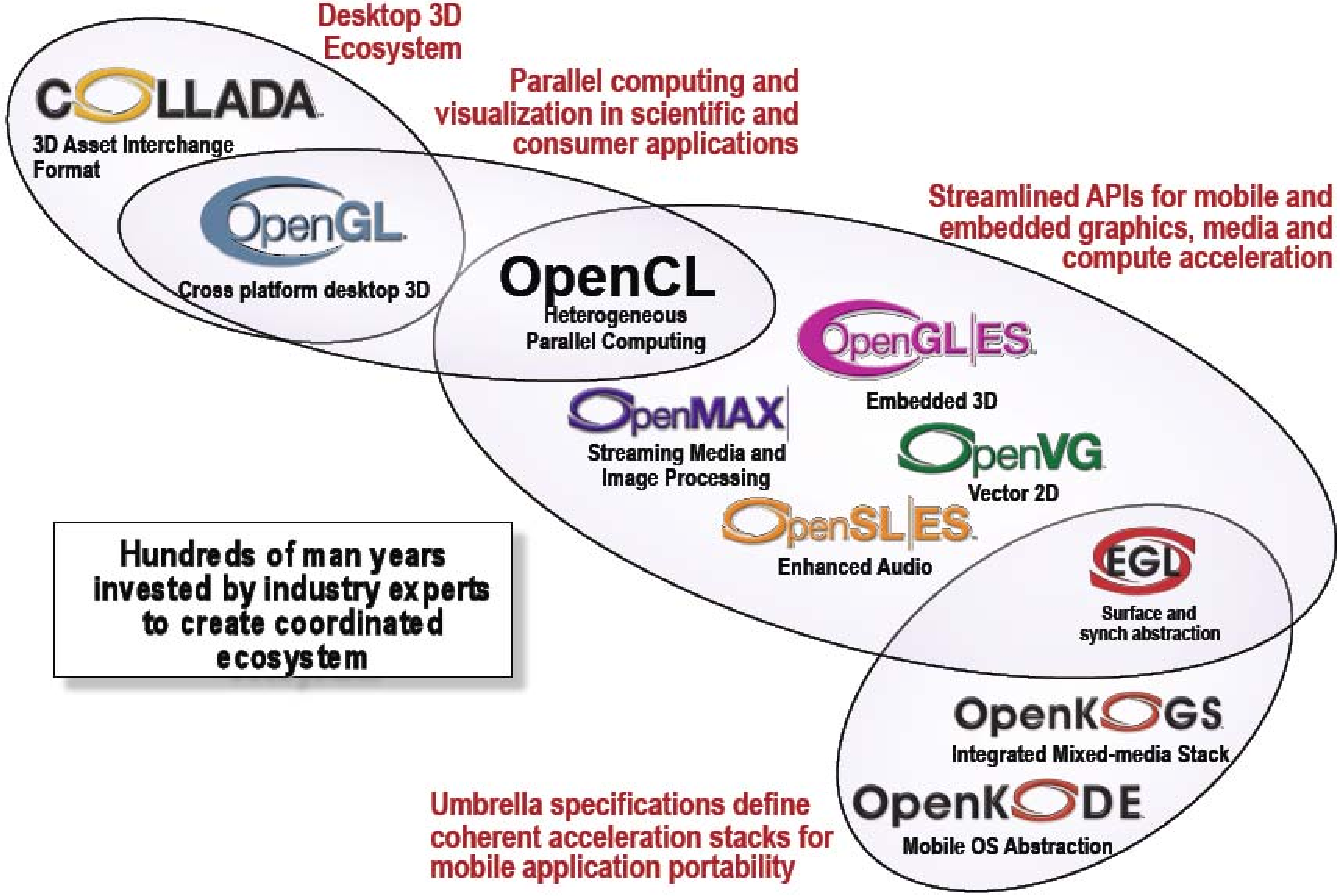
Sources

- OpenCL Tutorial - Introduction to OpenCL
- OpenCL for NVIDIA GPUs – Chris Lamb
- OpenCL – Parallel Computing for Heterogeneous Devices (SIGGASIA) – Kronos Group
- NVIDIA OpenCL Jump Start Guide
- OpenCL – Making Use of What You've Got
- OpenCL Basics and Advanced (PPAM 2009) – Domink Behr

Sources

- OpenCL Tutorial - Introduction to OpenCL
- OpenCL for NVIDIA GPUs – Chris Lamb
- OpenCL – Parallel Computing for Heterogeneous Devices (SIGGASIA) – Kronos Group
- NVIDIA OpenCL Jump Start Guide
- OpenCL – Making Use of What You've Got
- OpenCL Basics and Advanced (PPAM 2009) – Domink Behr

The Khronos API Ecosystem



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 initially proposed and is very active in the working group
  - Serving as specification editor
- Here are some of the other companies in the OpenCL working group

3DLABS SEMICONDUCTOR · ACTIVISION BLIZZARD · AMD · ARM · BROADCOM · EA

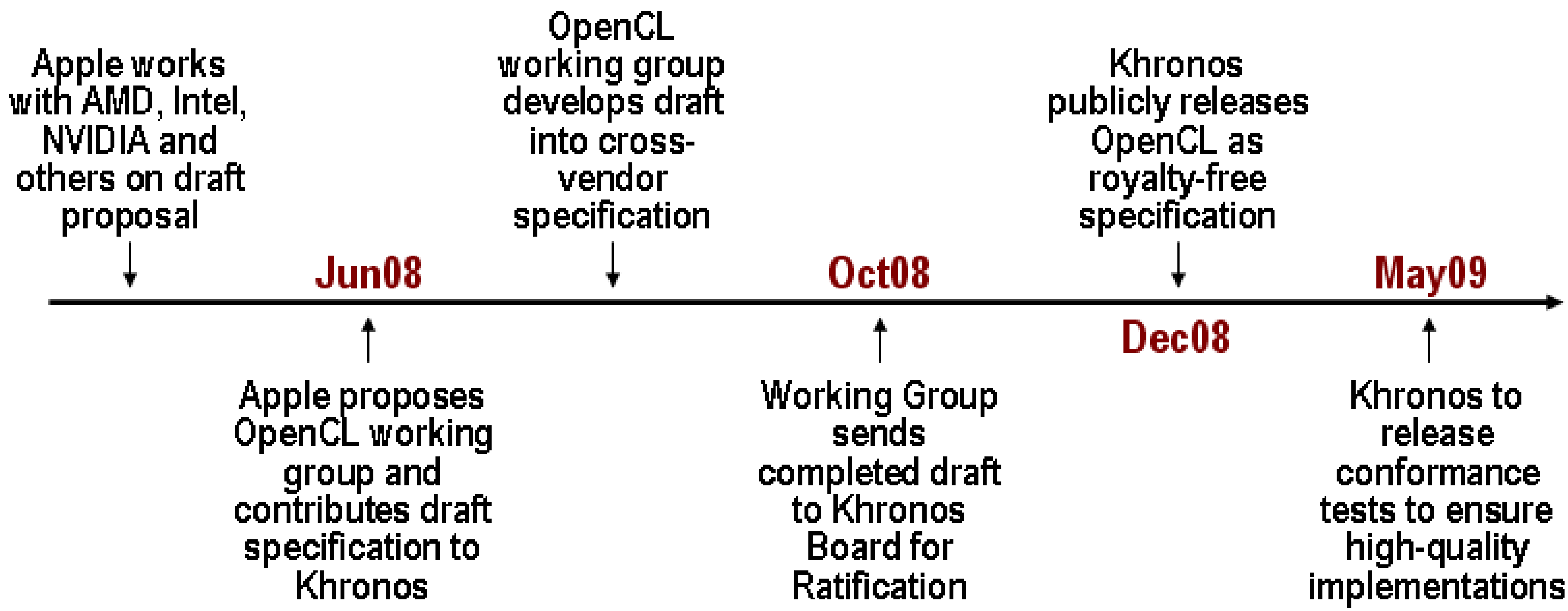
codeplay · ERICSSON · freescale semiconductor · HI CORP. · IBM · intel

Imagination TECHNOLOGIES · MOTOROLA · movidia · NOKIA · NVIDIA · QNX SOFTWARE SYSTEMS

RAPIDMIND · SAMSUNG · Seaweed SYSTEMS · TAKUMI · TEXAS INSTRUMENTS · UMEÅ UNIVERSITET

OpenCL Timeline

- **Six months from proposal to released specification**
  - Due to a strong initial proposal and a shared commercial incentive to work quickly
- **Apple's Mac OS X Snow Leopard will include OpenCL**
  - Improving speed and responsiveness for a wide spectrum of applications
- **Multiple OpenCL implementations expected in the next 12 months**
  - On diverse platforms



CUDA Working Group

- Because of Nexus and Visual Studio Integration….

Design Goals of OpenCL

- Use all computational resources in system
  - GPUs and CPUs as peers
  - Data- and task- parallel compute model
- Efficient parallel programming model
  - Based on C
  - Abstract the specifics of underlying hardware
- Specify accuracy of floating-point computations
  - IEEE 754 compliant rounding behavior
  - Define maximum allowable error of math functions
- Drive future hardware requirements

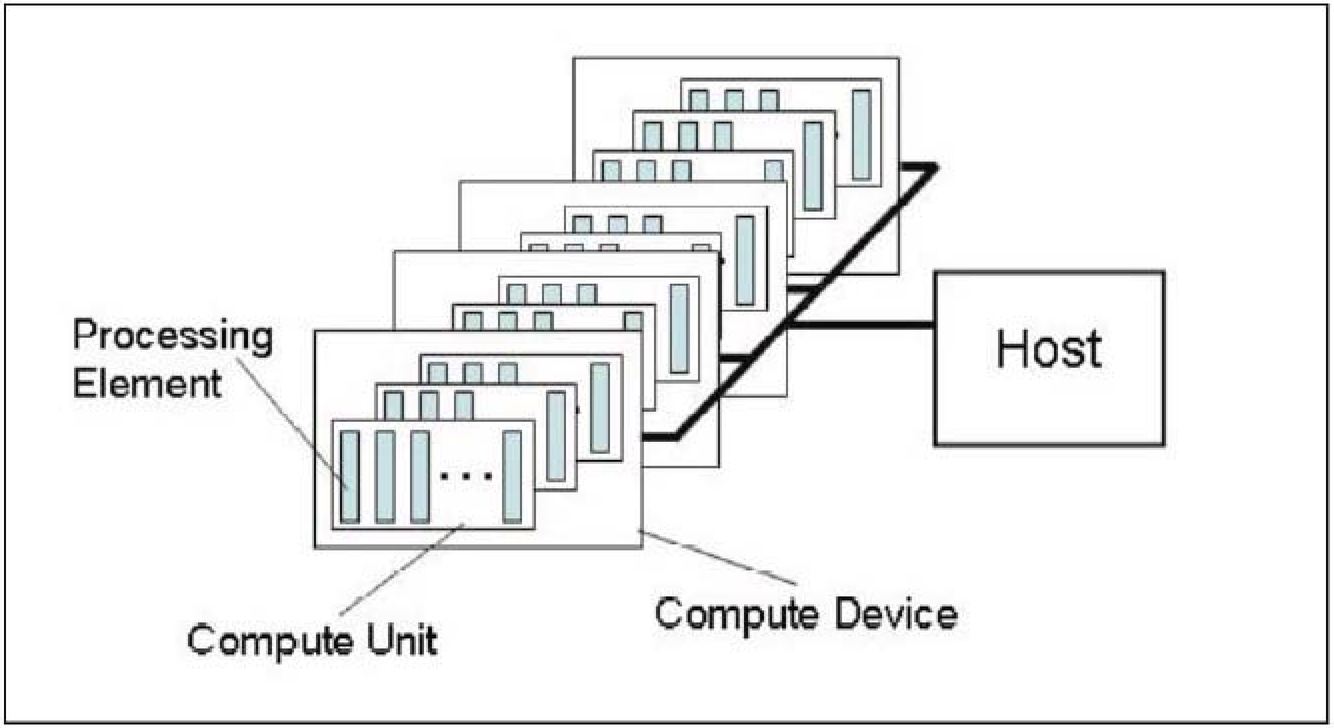
Anatomy of OpenCL

- Language Specification
  - C-based cross-platform programming interface
  - Subset of ISO C99 with language extensions - familiar to developers
  - Well-defined numerical accuracy (IEEE 754 rounding with specified max error)
  - Online or offline compilation and build of compute kernel executables
  - Includes a rich set of built-in functions

- Platform Layer API
  - A hardware abstraction layer over diverse computational resources
  - Query, select and initialize *compute devices*
  - Create *compute contexts* and *work-queues*

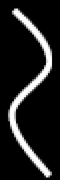
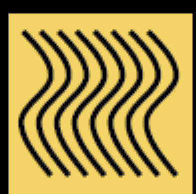
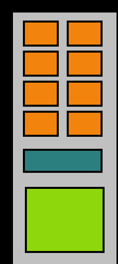
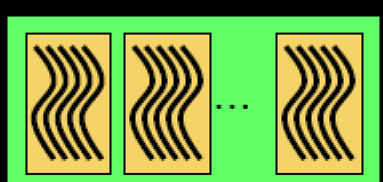
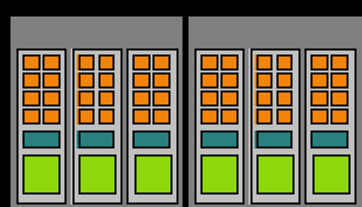
- Runtime API
  - Execute *compute kernels*
  - Manage scheduling, compute, and memory resources

OpenCL Platform Model (Section 3.1)

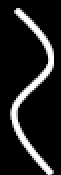
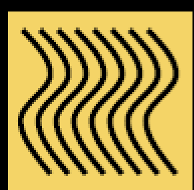
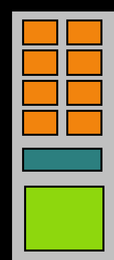
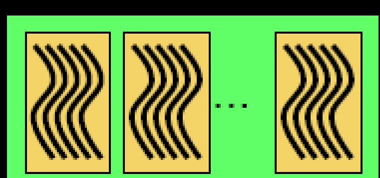
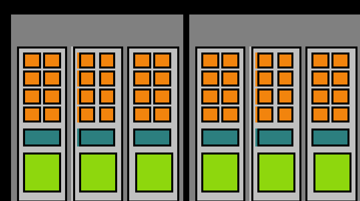


- One <u>Host</u> + one or more <u>Compute Devices</u>
  - Each **Compute Device** is composed of one or more **Compute Units**
    - Each **Compute Unit** is further divided into one or more **Processing Elements**

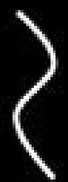
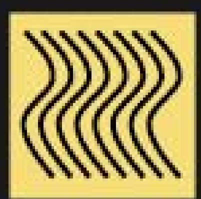
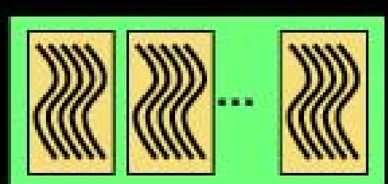
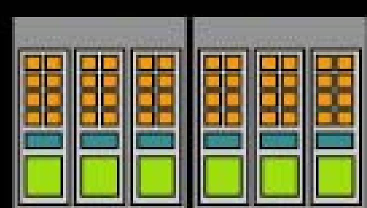
OpenCL Memory Model on NVIDIA

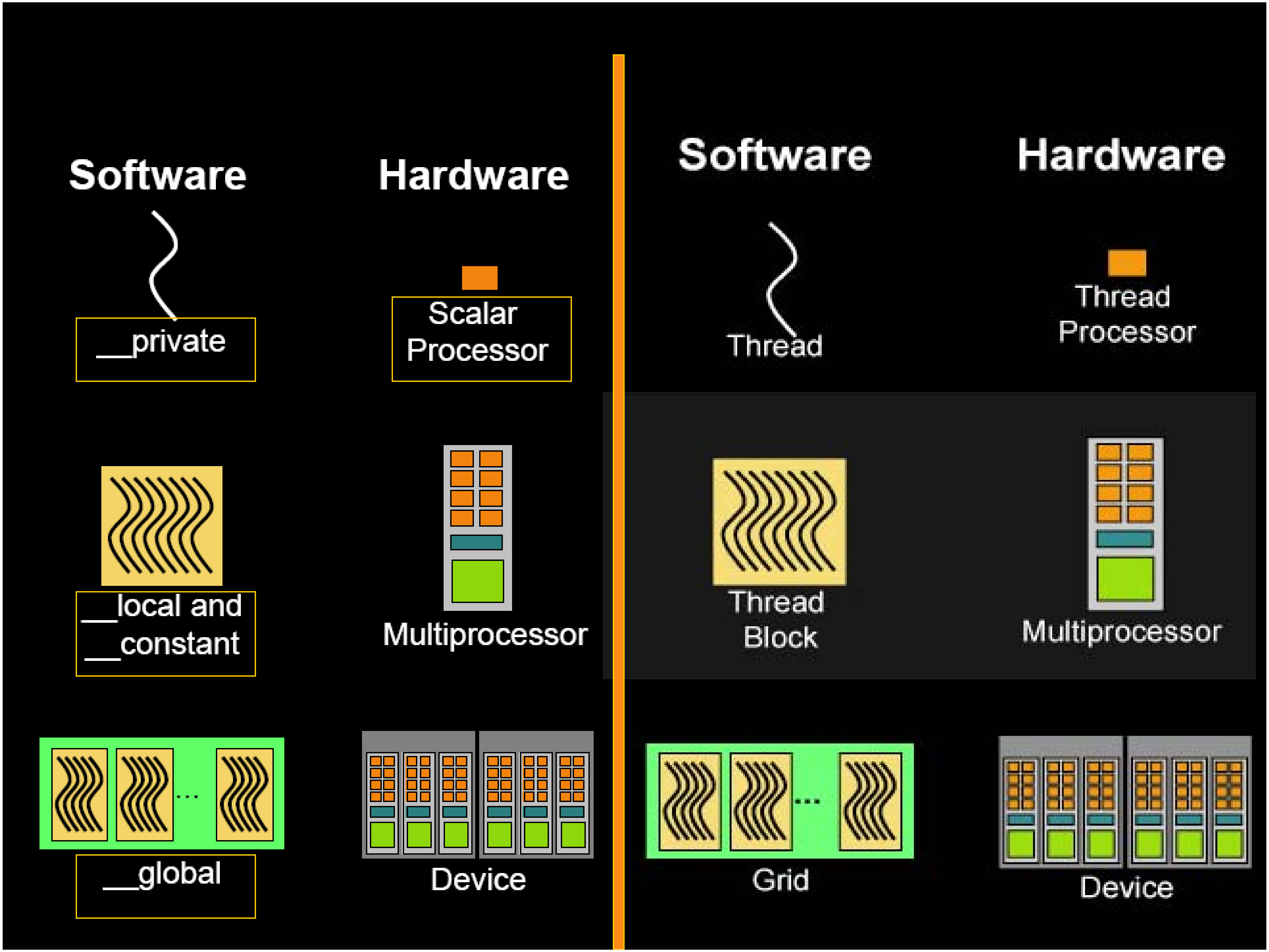
Software	Hardware	
 __private	 Scalar Processor	• Each hardware thread has a dedicated __private region for stack
 __local and __constant	 Multiprocessor	• Each multiprocessor has dedicated storage for __local memory and __constant caches • Work-items running on a multiprocessor can communicate through __local memory
 __global	 Device	• All work-groups on the device can access __global memory • Atomic operations allow powerful forms of global communication

OpenCL Synchronization on NVIDIA

Software	Hardware	
 mem_fence() atom_*()	 Scalar Processor	• Independent atomic operations and memory system control • Write collective operations in a familiar C-style
 barrier() work_group_copy()	 Multiprocessor	• Single instruction fast barrier support directly in HW • Collective operations leverage the entire multi-processor
 EnqueueNDRange cl_event	 Device	• Direct HW support for scheduling NDRange grids • Direct HW support for scheduling enqueued commands using cl_events

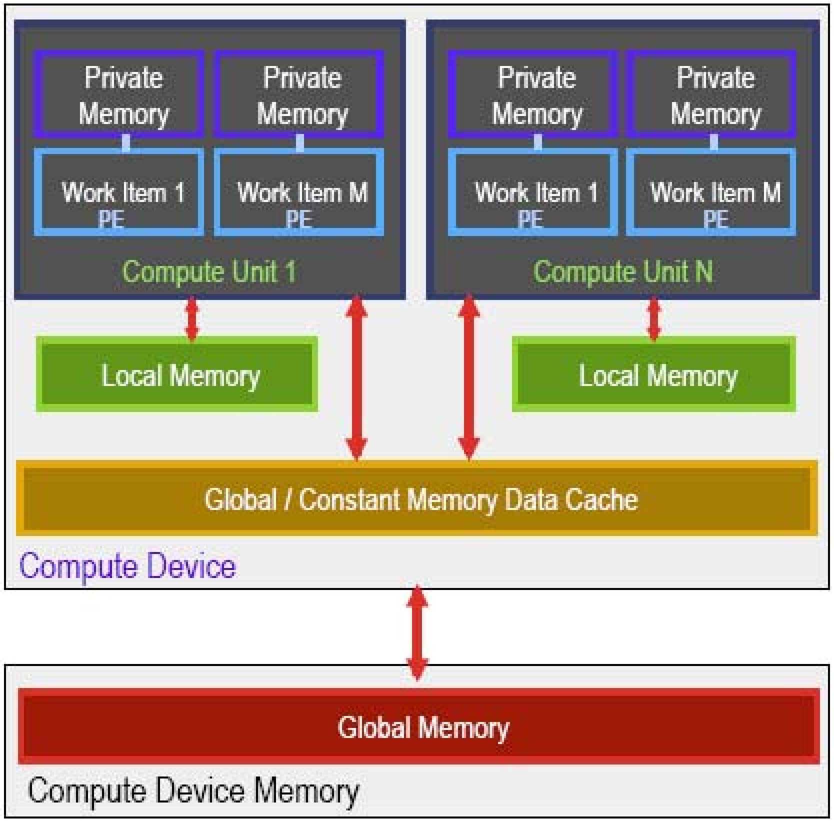
Execution Model CUDA

Software	Hardware	
 Thread	 Thread Processor	Threads are executed by thread processors
 Thread Block	 Multiprocessor	Thread blocks are executed on multiprocessors Thread blocks do not migrate Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)
 Grid	 Device	A kernel is launched as a grid of thread blocks Only one kernel can execute on a device at one time

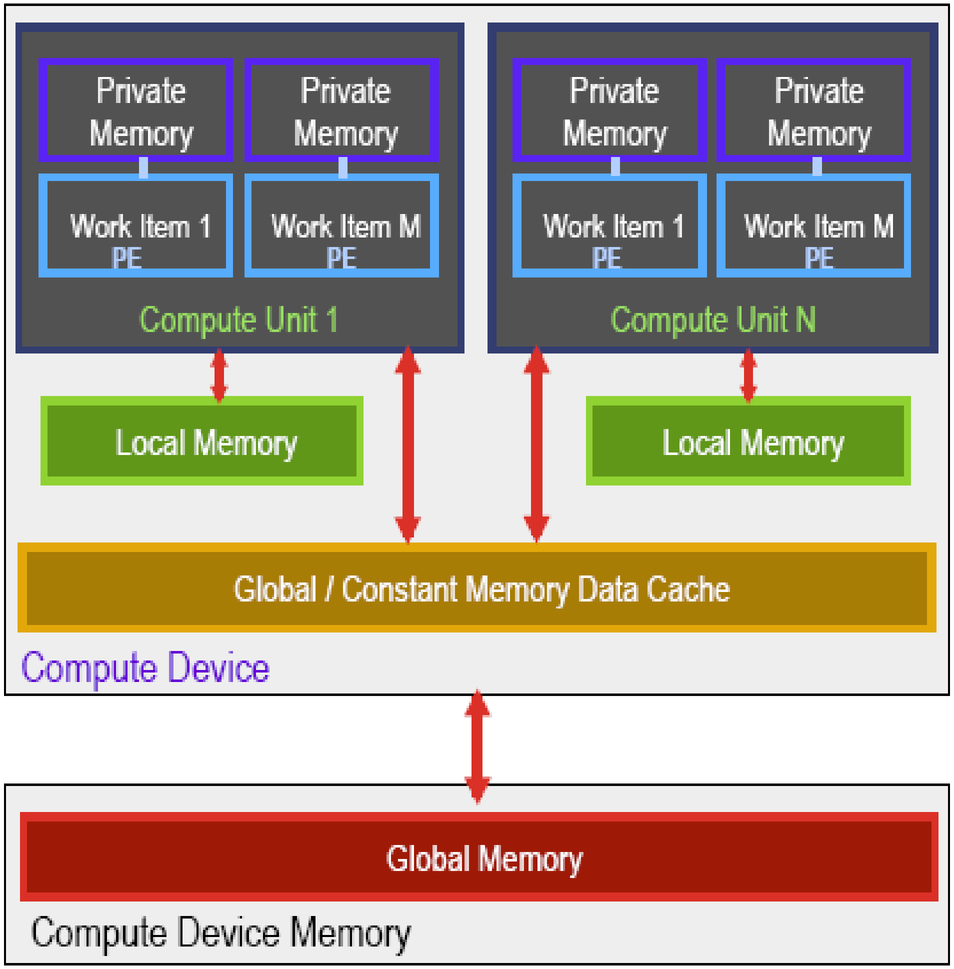

Software
Hardware
__private
Scalar Processor
__local and
__constant
Multiprocessor
__global
Device
Software
Hardware
Thread
Thread Processor
Thread Block
Multiprocessor
Grid
Device

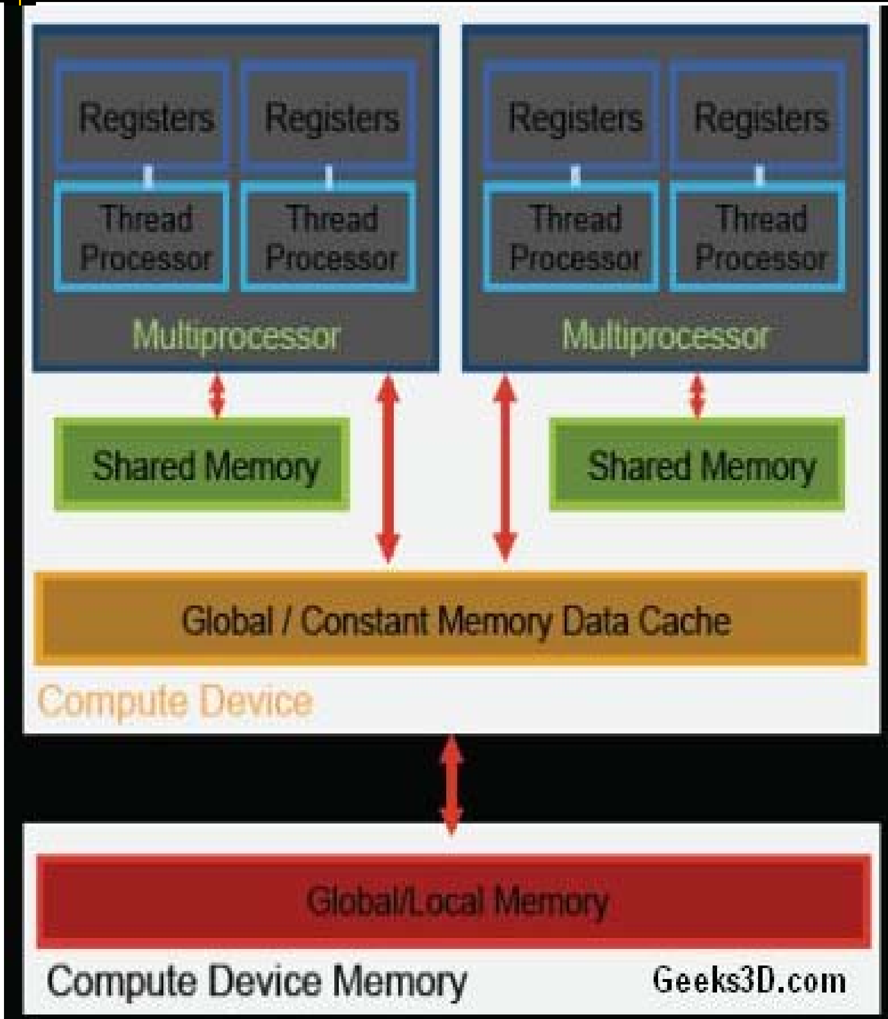
OpenCL Memory Model (Section 3.3)

- **Shared memory model**
  - Relaxed consistency
- **Multiple distinct address spaces**
  - Address spaces can be collapsed depending on the device's memory subsystem
- **Address spaces**
  - Private - private to a *work-item*
  - Local - local to a *work-group*
  - Global - accessible by all work-items in all work-groups
  - Constant - read only global space
- **Implementations map this hierarchy**
  - To available physical memories



Memory Model Comparison





OpenCL

CUDA

CUDA vs OpenCL

CUDA term	OpenCL term
GPU	Device
Multiprocessor	Compute Unit
Scalar core	Processing element
Global memory	Global memory
Shared (per-block) memory	Local memory
Local memory (automatic, or __local__)	Private memory
kernel	program
block	work-group
thread	work item

- Syntactic differences in kernel code
- C host-side API like CUDA C API
- Nothing like the CUDA language extensions!

Scalar Architecture

- NVIDIA GPUs have a scalar architecture
  - Use vector types in OpenCL for convenience, not performance
  - Generally want more work-items rather than large vectors per work-item

- Optimize performance by overlapping memory accesses with HW computation
  - High arithmetic intensity programs (i.e. high ratio of math to memory transactions)
  - Many concurrent work-items

Take Advantage of __local Memory

- Hundreds of times faster than __global memory
- Work-items can cooperate via __local memory
  - barrier() only needs CLK_LOCAL_MEM_FENCE, which is much lower overhead
- Use it to manage locality
  - Stage loads and stores in shared memory to optimize reuse

Optimize Memory Access

- Assess locality of __global memory access patterns
  - HW coalescing of accesses within 128-byte memory blocks
  - 1st Order performance effect
- Optimize for spatial locality of accesses in cached texture memory (OpenCL Images)
  - Image reads may benefit from processing as 2D blocks
  - Experiment with work-group aspect ratio to discover what's best
- Let OpenCL allocate memory optimally
  - CL_MEM_ALLOC_HOST_PTR
  - The implementation can optimize alignment and location
  - Can still get access for the host via clEnqueueMap{Buffer|Image}

Architecture – Execution Model

- Kernel – Smallest unit of execution, like a C function
- Host program – A collection of kernels
- Work item, an instance of kernel at run time
- Work group, a collection of work items

OpenCL Execution Model (Section 3.2)

- OpenCL *Program*:
  - Kernels
    - Basic unit of executable code — similar to C functions, CUDA kernels, etc.
    - Data-parallel or task-parallel
  - Host Program
    - Collection of compute kernels and internal functions
    - Analogous to a dynamic library
- *Kernel* Execution
  - The host program invokes a kernel over an index space called an ***NDRange***
    - NDRange, "N-Dimensional Range", can be a 1D, 2D, or 3D space
  - A single kernel instance at a point in the index space is called a ***work-item***
    - Work-items have unique global IDs from the index space
  - Work-items are further grouped into ***work-groups***
    - Work-groups have a unique work-group ID
    - Work-items have a unique local ID within a work-group

OpenCL Execution Model (Section 3.2)

- OpenCL *Program*:
  - Kernels
    - Basic unit of executable code — similar to C functions, CUDA kernels, etc.
    - Data-parallel or task-parallel
  - Host Program
    - Collection of compute kernels and internal functions
    - Analogous to a dynamic library
- *Kernel* Execution
  - The host program invokes a kernel over an index space called an ***NDRange***
    - NDRange, "N-Dimensional Range", can be a 1D, 2D, or 3D space
  - A single kernel instance at a point in the index space is called a ***work-item***
    - Work-items have unique global IDs from the index space
  - Work-items are further grouped into ***work-groups***
    - Work-groups have a unique work-group ID
    - Work-items have a unique local ID within a work-group

Expressing Data-Parallelism in OpenCL

- Define N-dimensional computation domain (N = 1, 2 or 3)
  - Each independent element of execution in N-D domain is called a work-item
  - The N-D domain defines the total number of work-items that execute in parallel
- E.g., process a 1024 x 1024 image: Global problem dimensions: 1024 x 1024 = 1 kernel execution per pixel: 1,048,576 total kernel executions

Scalar

```
void
scalar_mul(int n,
           const float *a,
           const float *b,
           float *result)
{
  int i;
  for (i=0; i<n; i++)
    result[i] = a[i] * b[i];
}
```

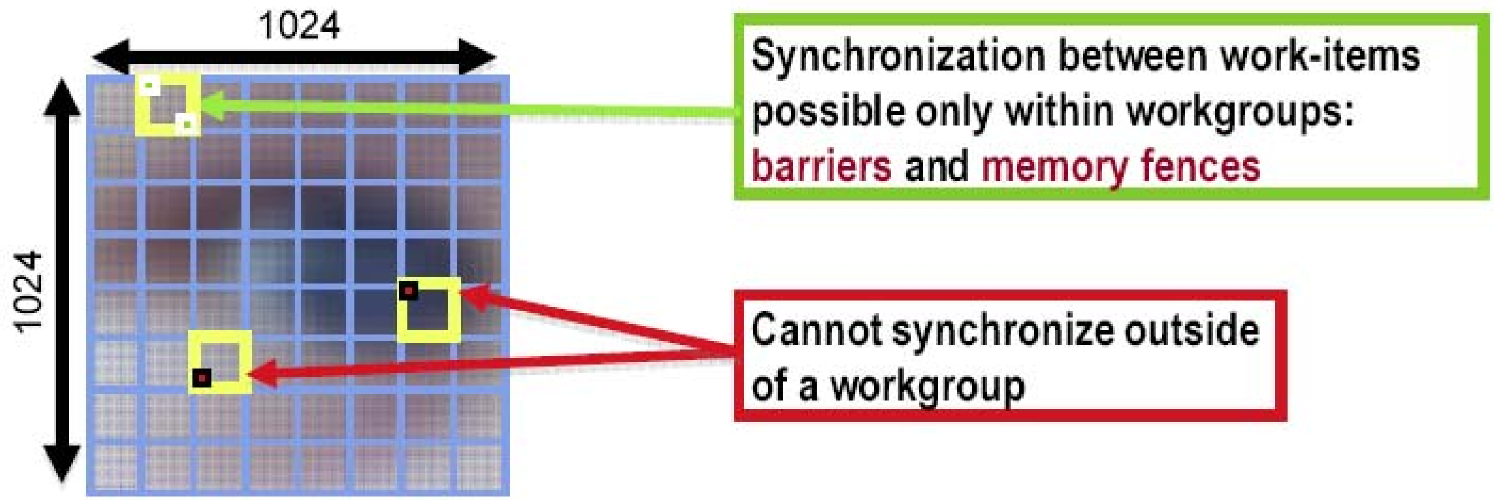
Data Parallel

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *result)
{
  int id = get_global_id(0);

  result[id] = a[id] * b[id];
}
// execute dp_mul over "n" work-items
```

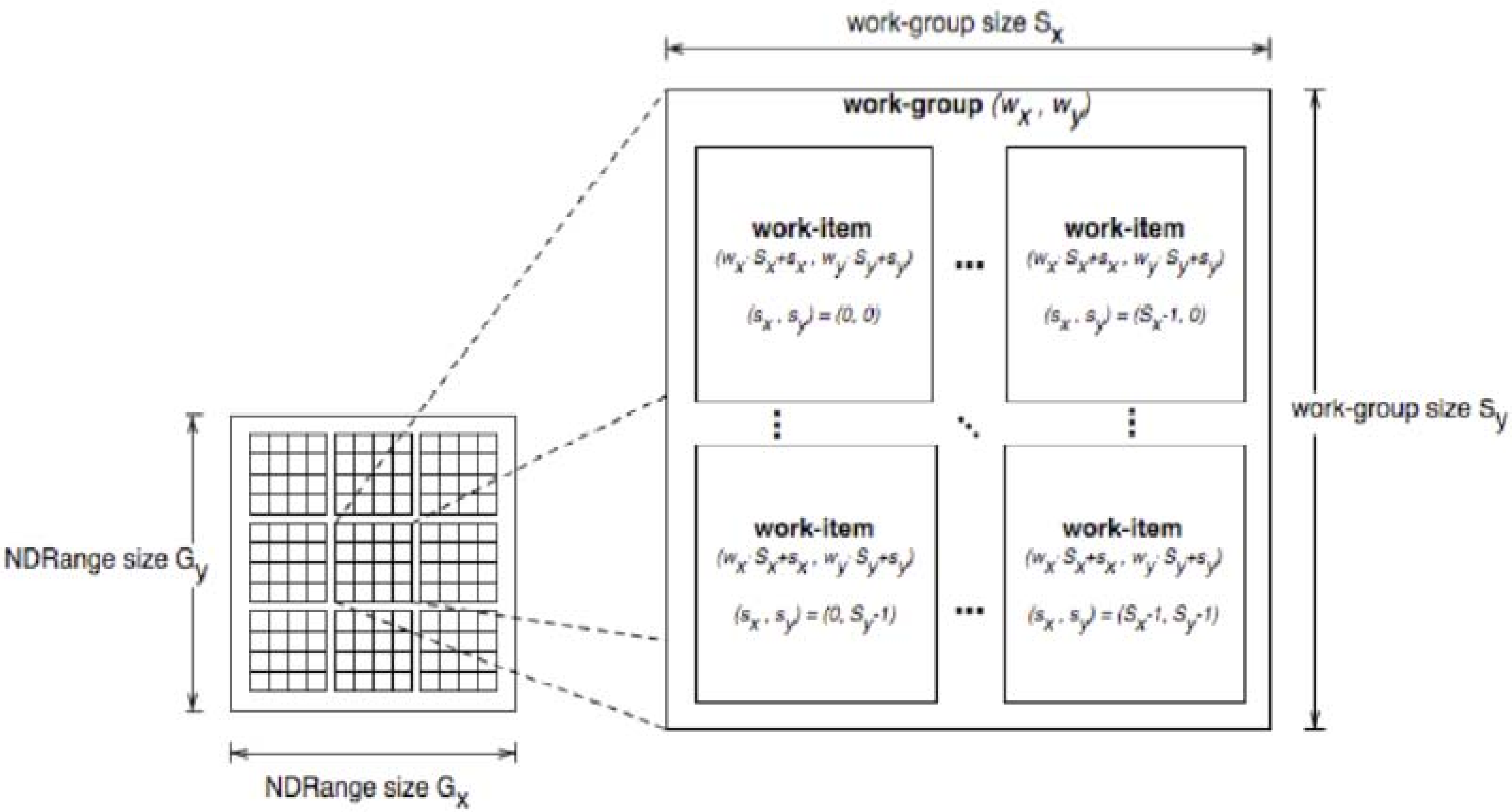
Global and Local Dimensions

- Global Dimensions: 1024 x 1024 (whole problem space)
- Local Dimensions: 128 x 128 (executed together)



- Choose the dimensions that are "best" for your algorithm

Kernel Execution



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

Programming Model

Data-Parallel Model (Section 3.4.1)

- **Must be implemented by *all* OpenCL compute devices**

- **Define N-Dimensional computation domain**
  - Each independent element of execution in an N-Dimensional domain is called a *work-item*
  - N-Dimensional domain defines total # of work-items that execute in parallel
    = *global work size*

- **Work-items can be grouped together — *work-group***
  - Work-items in group can communicate with each other
  - Can synchronize execution among work-items in group to coordinate memory access

- **Execute multiple work-groups in parallel**
  - Mapping of global work size to work-group can be implicit or explicit

Programming Model

Task-Parallel Model (Section 3.4.2)

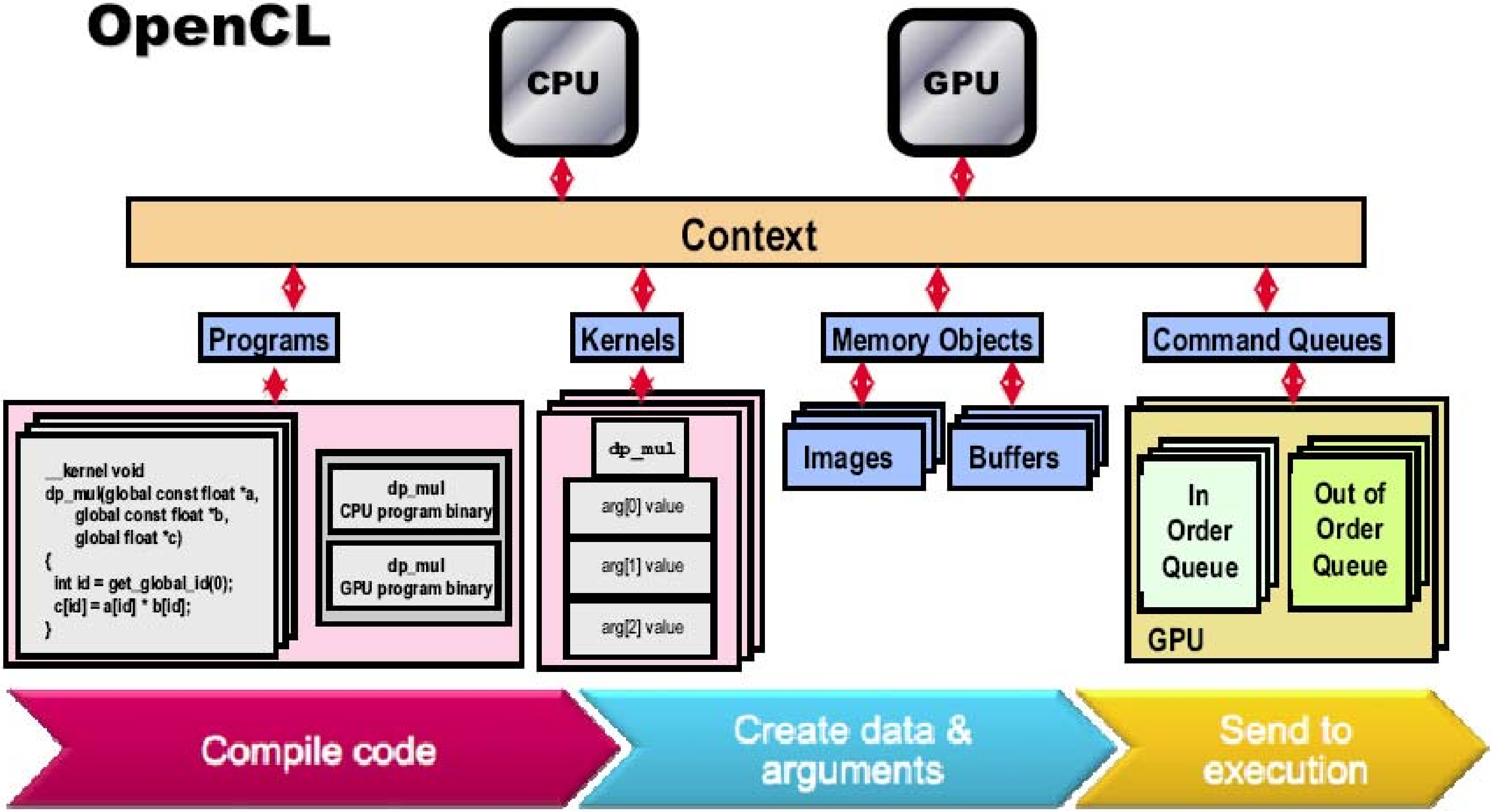
- **Some compute devices can also execute task-parallel compute kernels**

- **Execute as a *single* work-item**
  - A compute kernel written in OpenCL
  - A native C / C++ function

OpenCL Objects

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

OpenCL


CPU
GPU
Context
Programs
Kernels
Memory Objects
Command Queues
__kernel void
dp_mul(global const float *a,
global const float *b,
global float *c)
{
int id = get_global_id(0);
c[id] = a[id] * b[id];
}
dp_mul
CPU program binary
dp_mul
GPU program binary
dp_mul
arg[0] value
arg[1] value
arg[2] value
Images
Buffers
In Order Queue
Out of Order Queue
GPU
Compile code
Create data & arguments
Send to execution

Basic OpenCL Program Structure

- Host program
  - Query compute devices
  - Create contexts

  → **Platform Layer**

  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources

  → **Runtime**

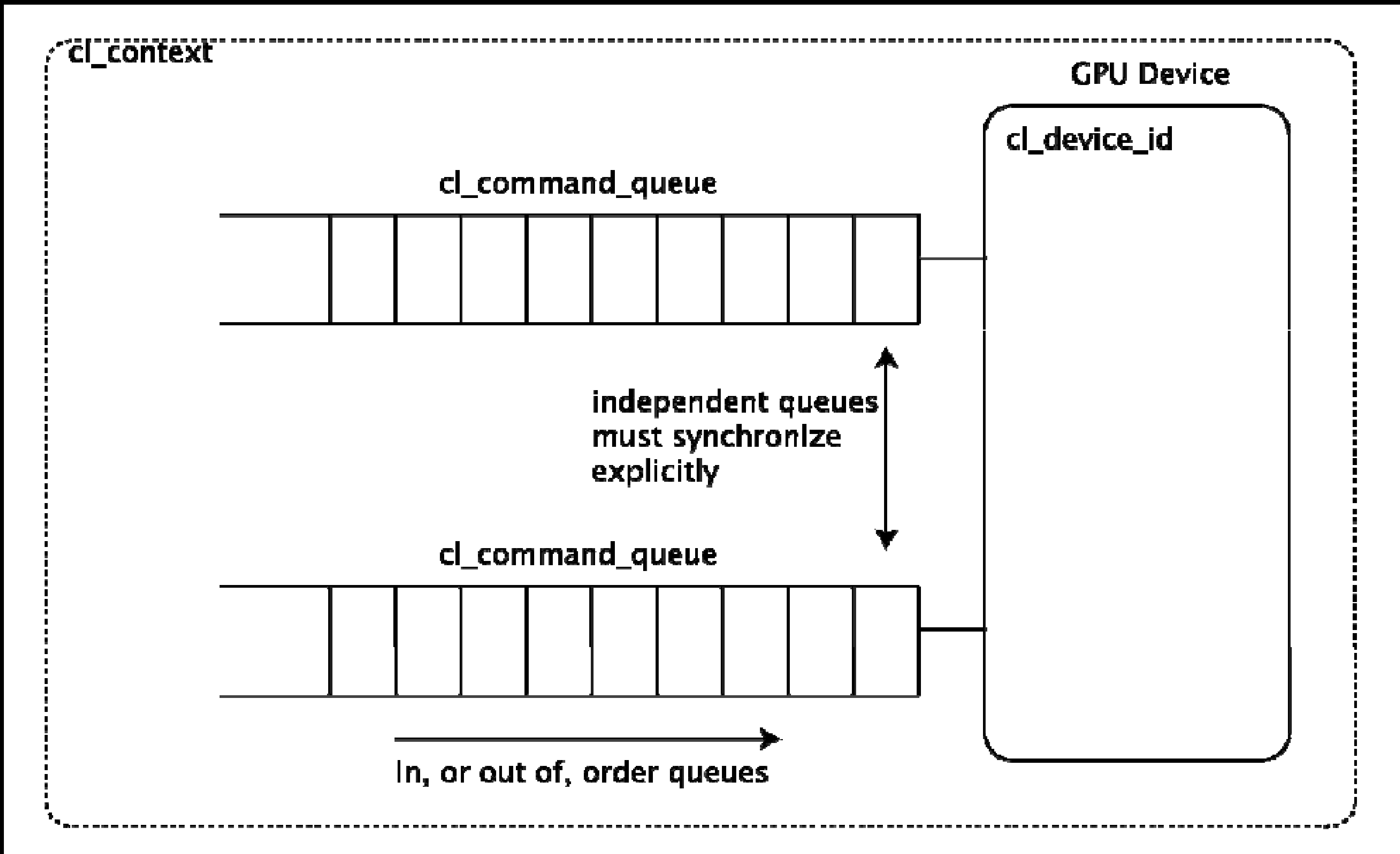
- Kernels
  - C code with some restrictions and extensions

  → **Language**

Memory Objects (Section 5.2)

- Buffer objects
  - 1D collection of objects (like C arrays)
  - Scalar types & Vector types, as well as user-defined Structures
  - Buffer objects accessed via pointers in the kernel
- Image objects
  - 2D or 3D texture, frame-buffer, or images
  - Must be addressed through built-in functions
- Sampler objects
  - Describe how to sample an image in the kernel
    - Addressing modes
    - Filtering modes

Command Queues



Getting started

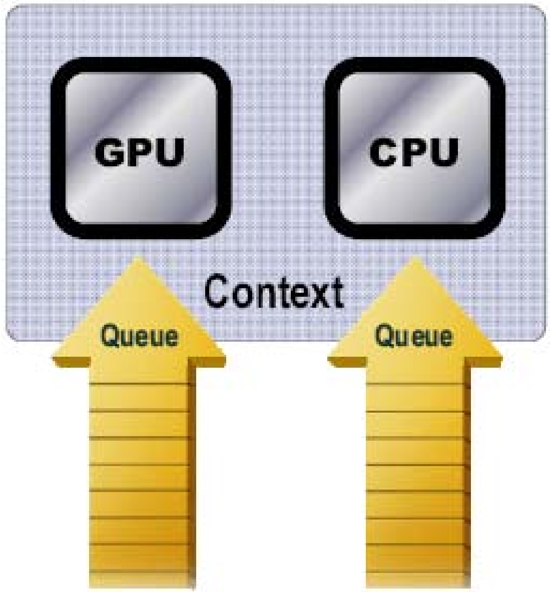
- Initialization
- Creating of memory objects
- Transfering (input) data
- Execution
- Synchronization
- Transfering (output) data
- Cleanup

Getting started
initialization

- Get platform
  - clGetPlatformIDs
- Get devices for platform
  - clGetDeviceIDs
- Create context for devices
  - clCreateContext
- Create command queue on a device within context
  - clCreateCommandQueue

Setup

1. Get the device(s)
2. Create a context
3. Create command queue(s)




```
cl_uint num_devices_returned;
cl_device_id devices[2];
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1,
                     &devices[0], num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1,
                     &devices[1], &num_devices_returned);

cl_context context;
context = clCreateContext(0, 2, devices, NULL, NULL, &err);

cl_command_queue queue_gpu, queue_cpu;
queue_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
```

Choosing Devices

- A system may have several devices—which is best?
- The “best” device is algorithm- and hardware-dependent

- Query device info with: `clGetDeviceInfo(device, param_name, *value)`
  - Number of compute units `CL_DEVICE_MAX_COMPUTE_UNITS`
  - Clock frequency `CL_DEVICE_MAX_CLOCK_FREQUENCY`
  - Memory size `CL_DEVICE_GLOBAL_MEM_SIZE`
  - Extensions (double precision, atomics, etc.)

- Pick the best device for your algorithm

Getting started
create memory objects

- Create Buffer object for context
  - clCreateBuffer
- Create Image object for context
  - clCreateImage2D
  - clCreateImage3D

Allocating Images and Buffers

```
cl_image_format format;
format.image_channel_data_type = CL_FLOAT;
format.image_channel_order = CL_RGBA;

cl_mem input_image;
input_image  = clCreateImage2D(context, CL_MEM_READ_ONLY, &format,
                               image_width, image_height, 0, NULL, &err);
cl_mem output_image;
output_image = clCreateImage2D(context, CL_MEM_WRITE_ONLY, &format,
                               image_width, image_height, 0, NULL, &err);

cl_mem input_buffer;
input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              sizeof(cl_float)*4*image_width*image_height, NULL, &err);
cl_mem output_buffer;
output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                               sizeof(cl_float)*4*image_width*image_height, NULL, &err);
```

Memory Resources

- **Buffers**
  - Simple chunks of memory
  - Kernels can access however they like (array, pointers, structs)
  - Kernels can read and write buffers
- **Images**
  - Opaque 2D or 3D formatted data structures
  - Kernels access only via read_image() and write_image()
  - Each image can be read or written in a kernel, but not both

Image Formats and Samplers

- Formats
  - Channel orders: `CL_A`, `CL_RG`, `CL_RGB`, `CL_RGBA`, etc.
  - Channel data type: `CL_UNORM_INT8`, `CL_FLOAT`, etc.
  - `clGetSupportedImageFormats()` returns supported formats
- Samplers (for reading images)
  - Filter mode: linear or nearest
  - Addressing: clamp, clamp-to-edge, repeat or none
  - Normalized: true or false
- Benefit from image access hardware on GPUs

Getting started
transfer data

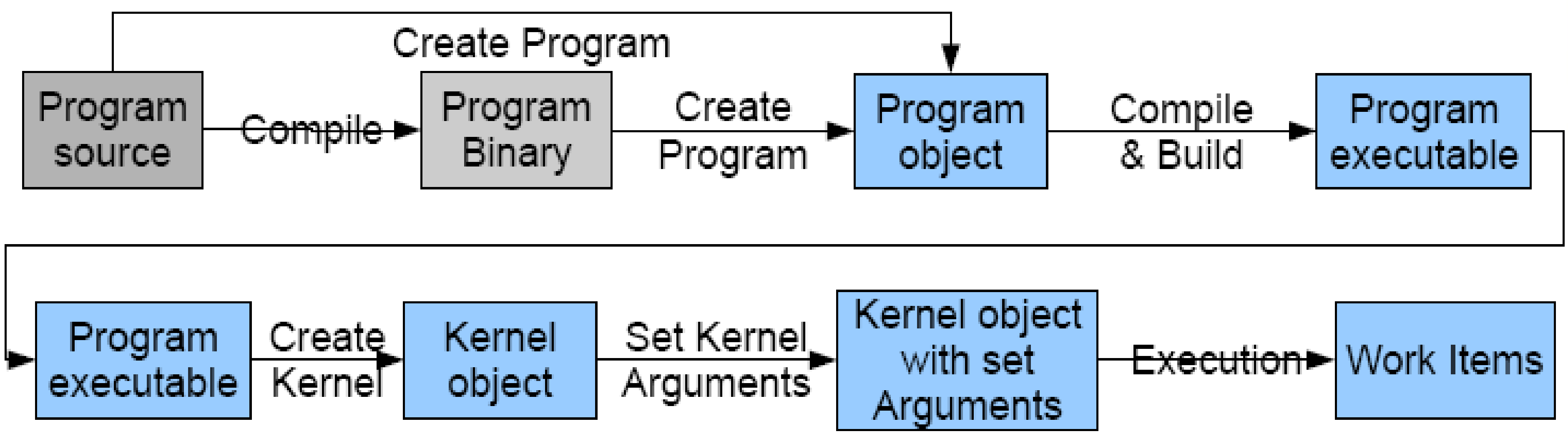
- Read/Write/Copy Buffer/Image
  - clEnqueueRead/Write/Copy Buffer/Image
  - Copy between buffer and image
    - clEnqueueCopyBufferToImage
    - clEnqueueCopyImageToBuffer
- Map/Unmap Buffer/Image
  - clEnqueueMapBuffer/Image
  - clEnqueueUnmapMemObject

Reading / Writing Memory Object Data

- Explicit commands to access memory object data
- Read from a region in memory object to host memory
  - `clEnqueueReadBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- Write to a region in memory object from host memory
  - `clEnqueueWriteBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- Map a region in memory object to host address space
  - `clEnqueueMapBuffer(queue, object, blocking, flags, offset, size, ...)`
- Copy regions of memory objects
  - `clEnqueueCopyBuffer(queue, srcobj, dstobj, src_offset, dst_offset, ...)`
- Operate synchronously (`blocking = CL_TRUE`) or asynchronously

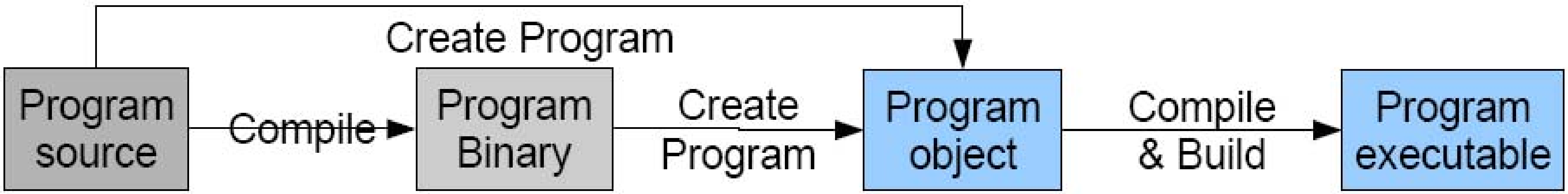
Execution overview

- Program source/binary, object, executable
- Kernel object
  - Create, Set arguments, Execute



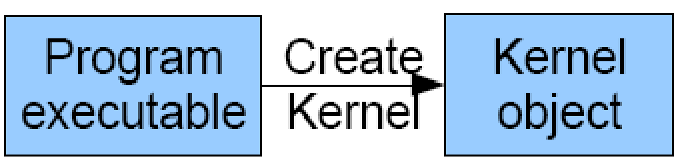
Program objects

- Create program for context and load source code/binary
  - clCreateProgramWithSource/Binary
- Compile and link program executable from source or binary for specified devices
  - clBuildProgram



Kernel objects

- Create kernel object for a kernel within program
  - clCreateKernel
- Create kernel objects for all kernels of a program
  - clCreateKernelsInProgram

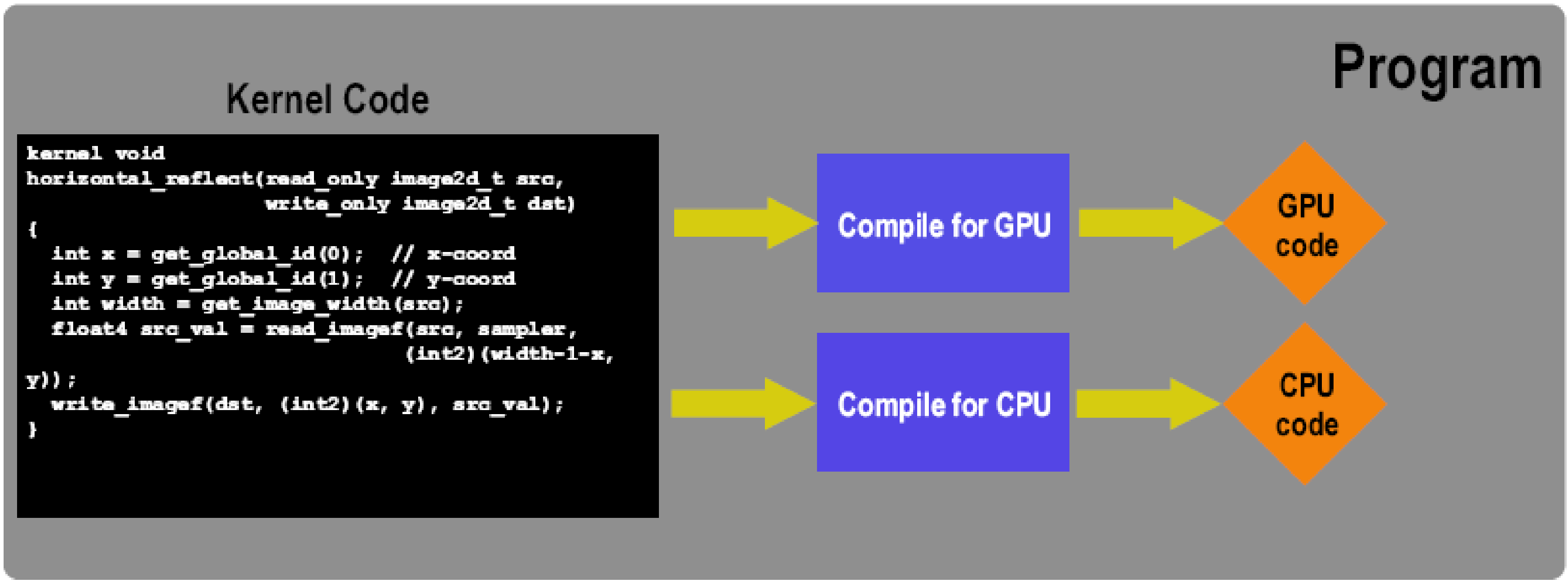


Program and Kernel Objects

- **Program objects encapsulate …**
  - a program source or binary
  - list of devices and latest successfully built executable for each device
  - a list of kernel objects
- **Kernel objects encapsulate …**
  - a specific kernel function in a program - declared with the kernel qualifier
  - argument values
  - kernel objects created after the program executable has been built

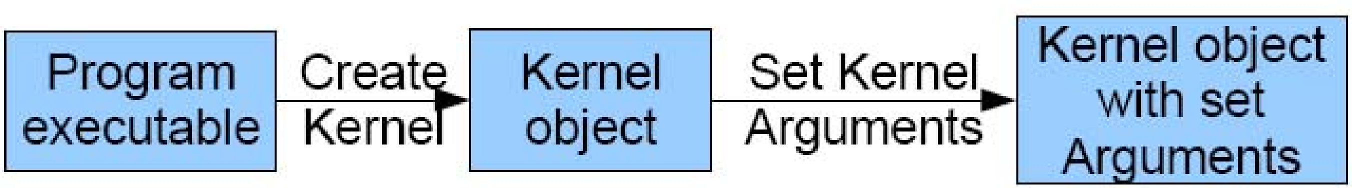
Executing Code

- Programs build executable code for multiple devices
- Execute the same code on different devices



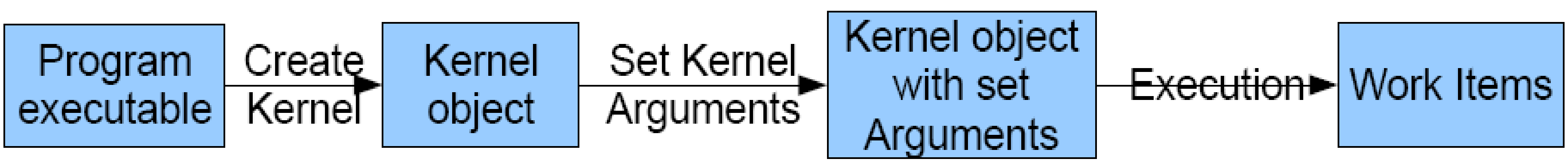
Kernel arguments

- Set kernel argument by index
  - clSetKernelArg



Kernel execution

- Enqueue execution of a kernel on a NDRange
  - ClEnqueueNDRangeKernel
- Enqueue execution of a single instance kernel
  - clEnqueueTask
- Enqueue execution of a native C/C++ function
  - clEnqueueNativeKernel



Executing Kernels

1. Set the kernel arguments
2. Enqueue the kernel

```
err = clSetKernelArg(kernel, 0, sizeof(input), &input);
err = clSetKernelArg(kernel, 1, sizeof(output), &output);

size_t global[3] = {image_width, image_height, 0};
err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global, NULL, 0, NULL, NULL;
```

- **Note: Your kernel is executed asynchronously**
  - Nothing may happen — you have just enqueued your kernel
  - Use a blocking read `clEnqueueRead*(... CL_TRUE ...)`
  - Use events to track the execution status

OpenCL C Language

- Data types
  - Scalar/Vector (2,4,8,16)
  - image2d_t/3d_t, sampler_t, event_t
- Adress space qualifiers
  - __global, __local, __constant, __private
- Image access qualifiers
  - __read_only, __write_only
- Function qualifiers
  - __kernel

Using Events on the Host

- `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
- `clGetEventInfo()`
  - Command type and status
    `CL_QUEUED, CL_SUBMITTED, CL_RUNNING, CL_COMPLETE`, or error code
- `clGetEventProfilingInfo()`
  - Command queue, submit, start, and end times

Address Spaces

- Kernel pointer arguments must use global, local or constant

```
kernel void distance(global float8* stars, local float8* local_stars)
kernel void sum(private int* p)    // Illegal because is uses private
```

- Default address space for arguments and local variables is private

```
kernel void smooth(global float* io) {
  float temp;
  ...
}
```

- image2d_t and image3d_t are always in global address space

```
kernel void average(read_only global image_t in, write_only image2d_t out)
```

CUDA vs OpenCL API Differences

- Naming Schemes
- How data gets passes to the API
- C for CUDA programs are compiled with an external tool (NVCC compiler)
- OpenCL compiler it typically invoked at runtime (you can offline compile too)

CUDA

```
cuInit(0);
cuDeviceGet(&hContext, 0);
cuCtxCreate(&hContext, 0, hDevice));

CUdeviceptr pDeviceMemA, pDeviceMemB,
pDeviceMemC;
cuMemAlloc(&pDeviceMemA, cnDimension *
sizeof(float));
cuMemAlloc(&pDeviceMemB, cnDimension *
sizeof(float));
cuMemAlloc(&pDeviceMemC, cnDimension *
sizeof(float));
// copy host vectors to device
cuMemcpyHtoD(pDeviceMemA, pA, cnDimension
* sizeof(float));
cuMemcpyHtoD(pDeviceMemB, pB, cnDimension
* sizeof(float));

cuFuncSetBlockShape(cuFunction, cnBlockSize,
1, 1);
cuLaunchGrid (cuFunction, cnBlocks, 1);
```

OpenCL

```
cl_context hContext;
hContext = clCreateContextFromType(0,
CL_DEVICE_DEVICE_TYPE_GPU, 0,0,0);

cl_mem hDeviceMemA, hDeviceMemB,
hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR,
cnDimension * sizeof(cl_float), pA, 0);
hDeviceMemB = clCreateBuffer(hContext,
CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR,
cnDimension * sizeof(cl_float), pA, 0);
hDeviceMemC = clCreateBuffer(hContext,
CL_MEM_WRITE_ONLY,
cnDimension * sizeof(cl_float) 0, 0);

clEnqueueNDRangeKernel(hCmdQueue,
hKernel, 1, 0,
&cnDimension, &cnBlockSize, 0, 0, 0);
```

CUDA Pointer Traversal

```
struct Node { Node* next; }
n = n->next; // undefined operation in OpenCL,
// since 'n' here is a kernel input
```

OpenCL Pointer Traversal

```
struct Node { unsigned int next; }
...
n = bufBase + n; // pointer arithmetic is fine, bufBase is
// a kernel input param to the buffer's beginning
```

Sample walkthrough oclVectorAdd

- Simple element by element vector addition

  For all i,

  C(i) = A(i) + B(i)

- Outline
  - Query compute devices
  - Create Context and Queue
  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources

CUDA Kernel code:

```
__global__ void
vectorAdd(const float * a, const float * b, float * c)
{
// Vector element index
int nIndex = blockIdx.x * blockDim.x + threadIdx.x;
c[nIndex] = a[nIndex] + b[nIndex];
}
```

OpenCL Kernel code:

```
__kernel void
vectorAdd(__global const float * a,
__global const float * b,
__global float * c)
{
// Vector element index
int nIndex = get_global_id(0);
c[nIndex] = a[nIndex] + b[nIndex];
}
```

CUDA kernel functions are declared using the “__global__” function modifier

OpenCL kernel functions are declared using “__kernel”.

CUDA Driver API Host code:

```
const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;
CUdevice hDevice;
CUcontext hContext;
CUmodule hModule;
CUfunction hFunction;
// create CUDA device & context
cuInit(0);
cuDeviceGet(&hContext, 0); // pick first device
cuCtxCreate(&hContext, 0, hDevice));
cuModuleLoad(&hModule, “vectorAdd.cubin”);
cuModuleGetFunction(&hFunction, hModule, "vectorAdd");
// allocate host vectors
float * pA = new float[cnDimension];
float * pB = new float[cnDimension];
float * pC = new float[cnDimension];
// initialize host memory
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);
// allocate memory on the device
CUdeviceptr pDeviceMemA, pDeviceMemB, pDeviceMemC;
cuMemAlloc(&pDeviceMemA, cnDimension * sizeof(float));
cuMemAlloc(&pDeviceMemB, cnDimension * sizeof(float));
cuMemAlloc(&pDeviceMemC, cnDimension * sizeof(float));
// copy host vectors to device
cuMemcpyHtoD(pDeviceMemA, pA, cnDimension * sizeof(float));
cuMemcpyHtoD(pDeviceMemB, pB, cnDimension * sizeof(float));
// setup parameter values
cuFuncSetBlockShape(cuFunction, cnBlockSize, 1, 1);
cuParamSeti(cuFunction, 0, pDeviceMemA);
cuParamSeti(cuFunction, 4, pDeviceMemB);
cuParamSeti(cuFunction, 8, pDeviceMemC);
cuParamSetSize(cuFunction, 12);
// execute kernel
cuLaunchGrid(cuFunction, cnBlocks, 1);
// copy the result from device back to host
cuMemcpyDtoH((void *) pC, pDeviceMemC, cnDimension * sizeof(float));
delete[] pA;  delete[] pB;  delete[] pC;
cuMemFree(pDeviceMemA);   cuMemFree(pDeviceMemB); cuMemFree(pDeviceMemC);
```

OpenCL Host Code:

```
const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
0, 0, 0);
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
0, 0, &nContextDescriptorSize);
cl_device_id * aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
nContextDescriptorSize, aDevices, 0);
// create a command queue for first device the context reported
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);
// create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1,
sProgramSource, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);
// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, “vectorAdd”, 0);
// allocate host vectors
float * pA = new float[cnDimension];
float * pB = new float[cnDimension];
float * pC = new float[cnDimension];
// initialize host memory
randomInit(pA, cnDimension);
randomInit(pB, cnDimension);
// allocate device memory
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, cnDimension * sizeof(cl_float), pA, 0);
hDeviceMemB = clCreateBuffer(hContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, cnDimension *
sizeof(cl_float), pA, 0);
hDeviceMemC = clCreateBuffer(hContext, CL_MEM_WRITE_ONLY, cnDimension * sizeof(cl_float), 0, 0);
// setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);
```

Declarations

```
cl_context cxMainContext;                  // OpenCL context
cl_command_queue cqCommandQue;             // OpenCL command que
cl_device_id* cdDevices;                   // OpenCL device list
cl_program cpProgram;                      // OpenCL program
cl_kernel ckKernel;                        // OpenCL kernel
cl_mem cmMemObjs[3];                       // OpenCL memory buffer objects
cl_int ciErrNum = 0;                       //  Error code var
size_t szGlobalWorkSize[1];                //  Global # of work items
size_t szLocalWorkSize[1];                 //  # of Work Items in Work Group
size_t szParmDataBytes;                    //  byte length of parameter storage
size_t szKernelLength;                     //  byte Length of kernel code
int iTestN = 10000;                  //  Length of demo test vectors
```

Contexts and Queues

```
// create the OpenCL context on a GPU device
cxMainContext = clCreateContextFromType (0, CL_DEVICE_TYPE_GPU,  NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo (cxMainContext, CL_CONTEXT_DEVICES, 0, NULL,  &szParmDataBytes);
cdDevices = (cl_device_id*)malloc(szParmDataBytes);
clGetContextInfo (cxMainContext, CL_CONTEXT_DEVICES, szParmDataBytes, cdDevices, NULL);

// create a command-queue
cqCommandQue = clCreateCommandQueue (cxMainContext, cdDevices[0], 0, NULL);
```

Create Memory Objects

```
// allocate the first source buffer memory object… source data, so read only
cmMemObjs[0] = clCreateBuffer (cxMainContext,
                               CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                               sizeof(cl_float) * iTestN, srcA, NULL);

// allocate the second source buffer memory object … source data, so read only
cmMemObjs[1] = clCreateBuffer (cxMainContext,
                               CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                               sizeof(cl_float) * iTestN, srcB, NULL);

// allocate the destination buffer memory object … result data, so write only
cmMemObjs[2] = clCreateBuffer (cxMainContext, CL_MEM_WRITE_ONLY,
                               sizeof(cl_float) * iTestN, NULL, NULL);
```

Create Program and Kernel

```
// create the program, in this case from OpenCL C source string array
cpProgram = clCreateProgramWithSource (cxMainContext, SOURCE_NUM_LINES,
                                        cVectorAdd, NULL, &ciErrNum);

// build the program
ciErrNum = clBuildProgram (cpProgram, 0, NULL, NULL, NULL, NULL);

// create the kernel
ckKernel = clCreateKernel (cpProgram, "VectorAdd", &ciErrNum);

// set the kernel Argument values
ciErrNum = clSetKernelArg (ckKernel, 0, sizeof(cl_mem), (void*)&cmMemObjs[0] );
ciErrNum |= clSetKernelArg (ckKernel, 1, sizeof(cl_mem), (void*)&cmMemObjs[1] );
ciErrNum |= clSetKernelArg (ckKernel, 2, sizeof(cl_mem), (void*)&cmMemObjs[2] );
```

Launch Kernel and Read Results

```
// set work-item dimensions
szGlobalWorkSize[0] = iTestN;
szLocalWorkSize[0]= 1;

// execute kernel
ciErrNum = clEnqueueNDRangeKernel (cqCommandQue, ckKernel, 1, NULL,
                                   szGlobalWorkSize, szLocalWorkSize,
                          0, NULL, NULL);

// read output
ciErrNum = clEnqueueReadBuffer(cqCommandQue, cmMemObjs[2], CL_TRUE,
                                   0, iTestN * sizeof(cl_float), dst, 0, NULL, NULL);
```

Cleanup

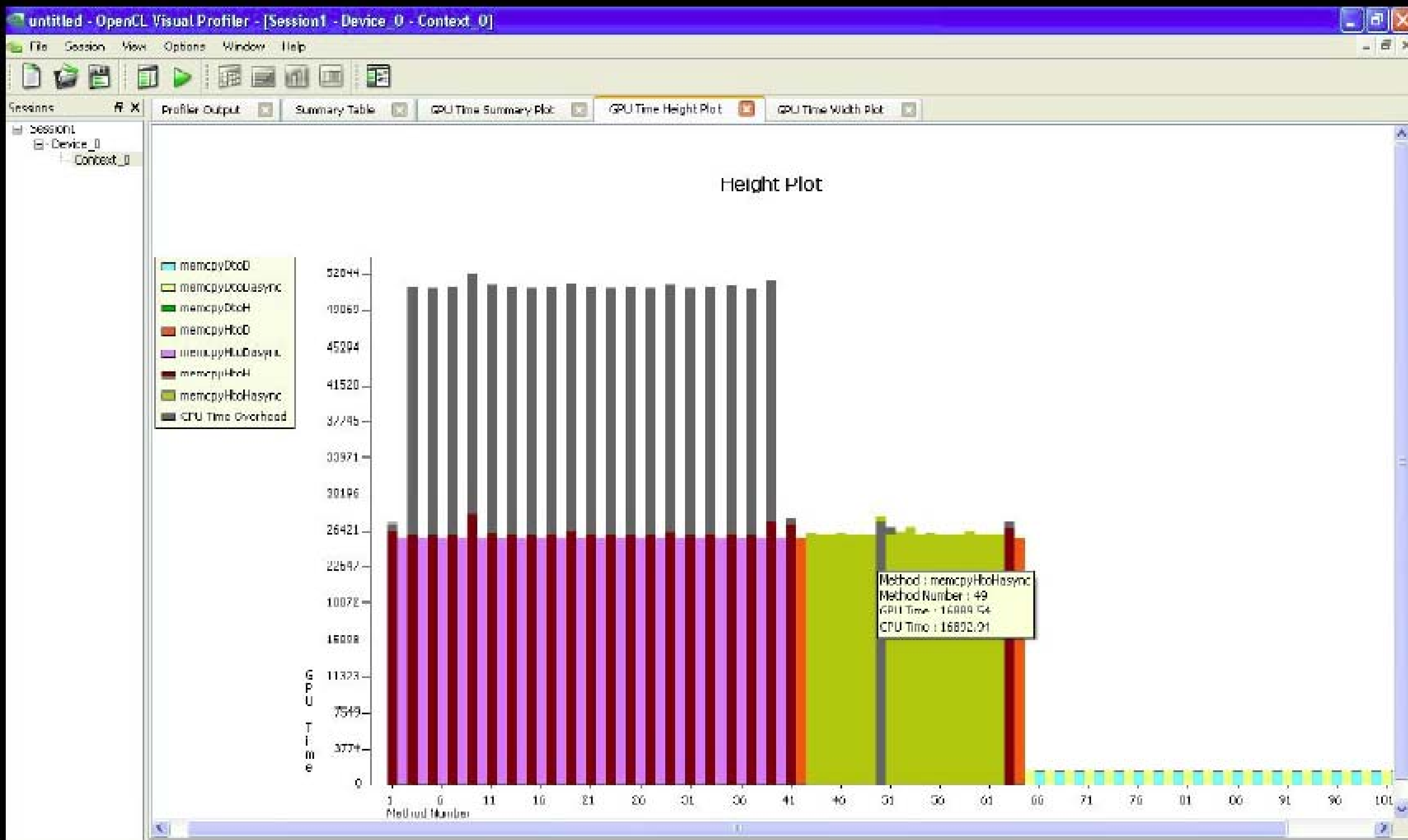
```
// release kernel, program, and memory objects
DeleteMemobjs (cmMemObjs, 3);
free (cdDevices);
clReleaseKernel (ckKernel);
clReleaseProgram (cpProgram);
clReleaseCommandQueue (cqCommandQue);
clReleaseContext (cxMainContext);
```

OpenCL Profiler Overview

- Profiler facilitates analysis and optimization of OpenCL programs by:
  - Reporting hardware counter values:
    - Number of various bus transactions
    - Branches
    - Effective Parallelism
    - Etc.
  - Computing per kernel statistics:
    - Effective instruction throughput
    - Effective memory throughput
  - Visually displaying time spent in various GPU calls
- Requires no instrumentation of the source code

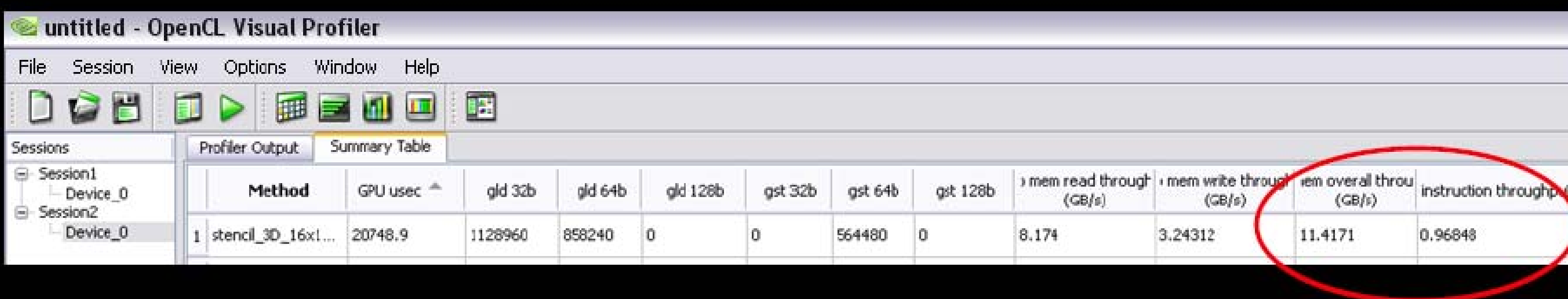
OpenCL Profiler Example

Time profile of GPU operations



OpenCL Profiler Sample Uses

- Determining whether kernel performance is bound by instruction or memory throughput
- Determining whether performance is limited by kernel execution or data transfer times
- Determining percentage of the application time spent in each kernel



Personal Aside…

- I'm a bit skeptical…
- 1) slower

Source: Matt Harvey Porting CUDA to OpenCL

Stage	CUDA	Nvidia OCL	Speedup
Bonded terms	0.396	0.477	-1.1x
Binning	0.863	3.833	-4.4x
Nonbonded terms	26.548	39.408	-1.5x
Integration	0.090	0.184	-2.0x
Total	28.506	43.924	-1.5x

NVidia Tesla C1060, HP xw6600, 2 × Xeon 5430, Centos 5.4, CUDA 3.0 beta
Model: Gramicidin-A 29042 atoms, cutoff=12Å switch=10.5Å

- 2) NVIDIA has to fully commit…

More Performance notes...

Stage	CUDA	Nvidia OCL	ATI OCL	Speedup
Bonded terms	0.396	0.477	1.930	-2.2x
Binning*	16.438	21.160	61.981	-3.8x
Nonbonded terms	26.548	39.408	168.342	-6.3x
			137.94	-5.1x
Integration	0.090	0.184	0.489	-5.4x
Total	44.081	61.251	234.196	-5.3x

NVidia Tesla C1060, HP xw6600, 2 x Xeon 5430, Centos 5.4, CUDA 3.0 beta
ATI 4850 (≈1TFLOP), HP xw6600, 2 x Xeon 5430, CentOS 5.4, ATI OpenCL beta 4
Model: Gramicidin-A 29042 atoms, cutoff=12Å switch=10.5Å

- Slow alogrithm for binning (no atomic memory operations)

Source: Matt Harvey Porting CUDA to OpenCL