GPU COMPUTING
LECTURE 12 - GPU PROGRAMMING MODELS

Kazem Shekofteh
kazem.shekofteh@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg
Inspired from lectures by Holger Fröning

ANNOUNCEMENT OF THE EXAM

Date: Tue, February 28, 2023

Time: 10:00

Location: INF 252 gHS, SR A + SR B

PROGRAMMING A GPU FOR GENERAL-PURPOSE COMPUTATIONS

Up to now: CUDA

- Vendor-specific (NVIDIA), pros and cons
- First programming model for GPU computing (2006)
- Alternatives?

Similar approach: OpenCL (1)

- Since 2008
- Imperative language

Directive-based programming: OpenACC (2)

- Since 2012, to be integrated in OpenMP
- Declarative language

In general: domain-specific languages (DSL) vs. general-purpose languages

CUDA REVIEW

- Up to now we've learned most aspects of CUDA
    - Bottom-up approach for this course
- Main documentation is the CUDA Programming Guide
    - Also bottom-up
- CUDA: various methods to control execution
    - Plenty of opportunities, plenty of responsibilities
- OpenCL: similar, but not vendor-specific
    - Documentation: top-down

CUDA

OpenCL

OPENCL
(AND A NICE CUDA REVIEW)

OVERVIEW

Low-level, high-performance, portable abstraction

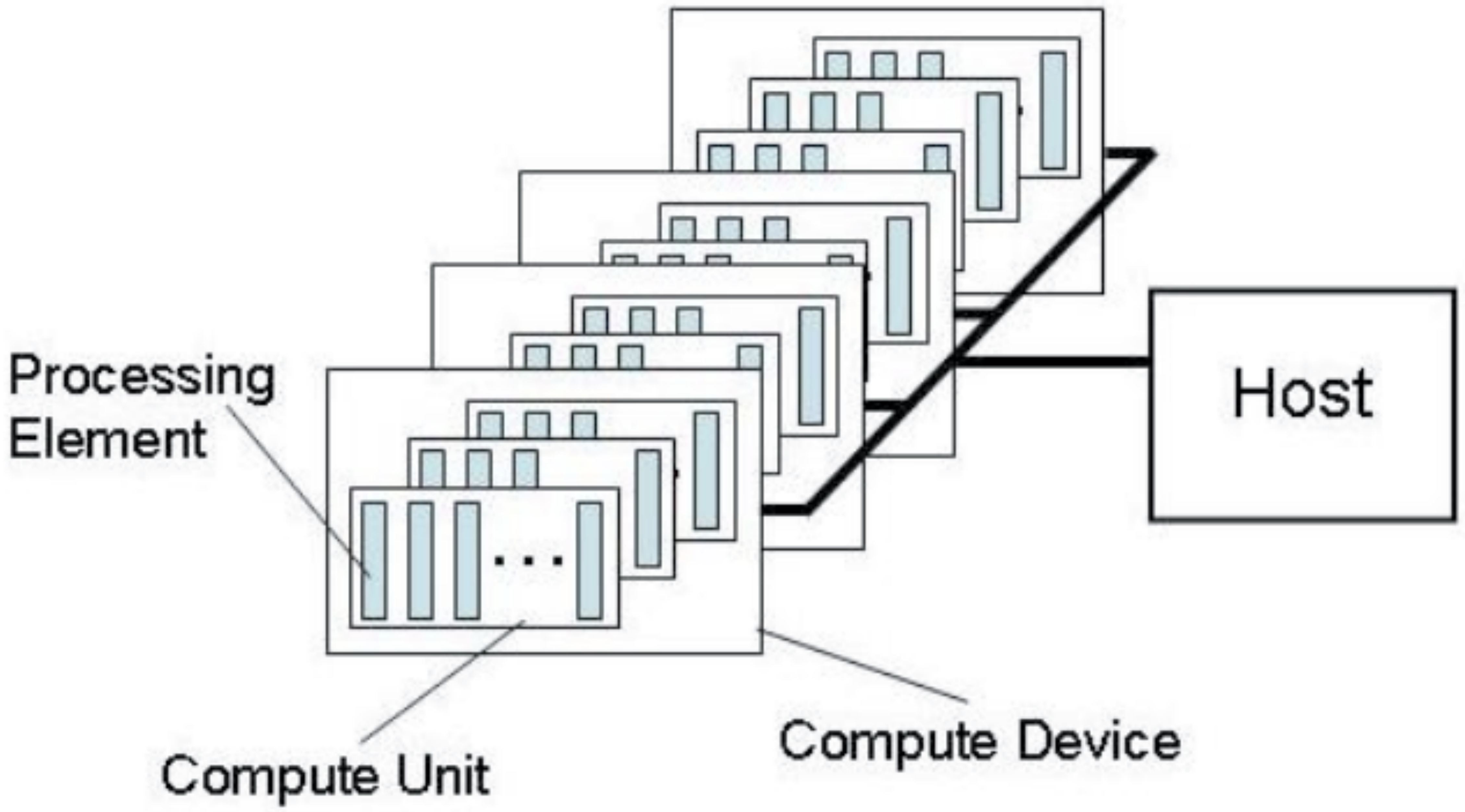
- API
- Cross-platform programming language

OpenCL architecture

- Platform model (see below)
- Execution model (1)
- Memory model (2)
- Programming model (3)

Platform model

- An abstraction how OpenCL sees the HW
- Host and device kernel code
- Converged and diverged control flow

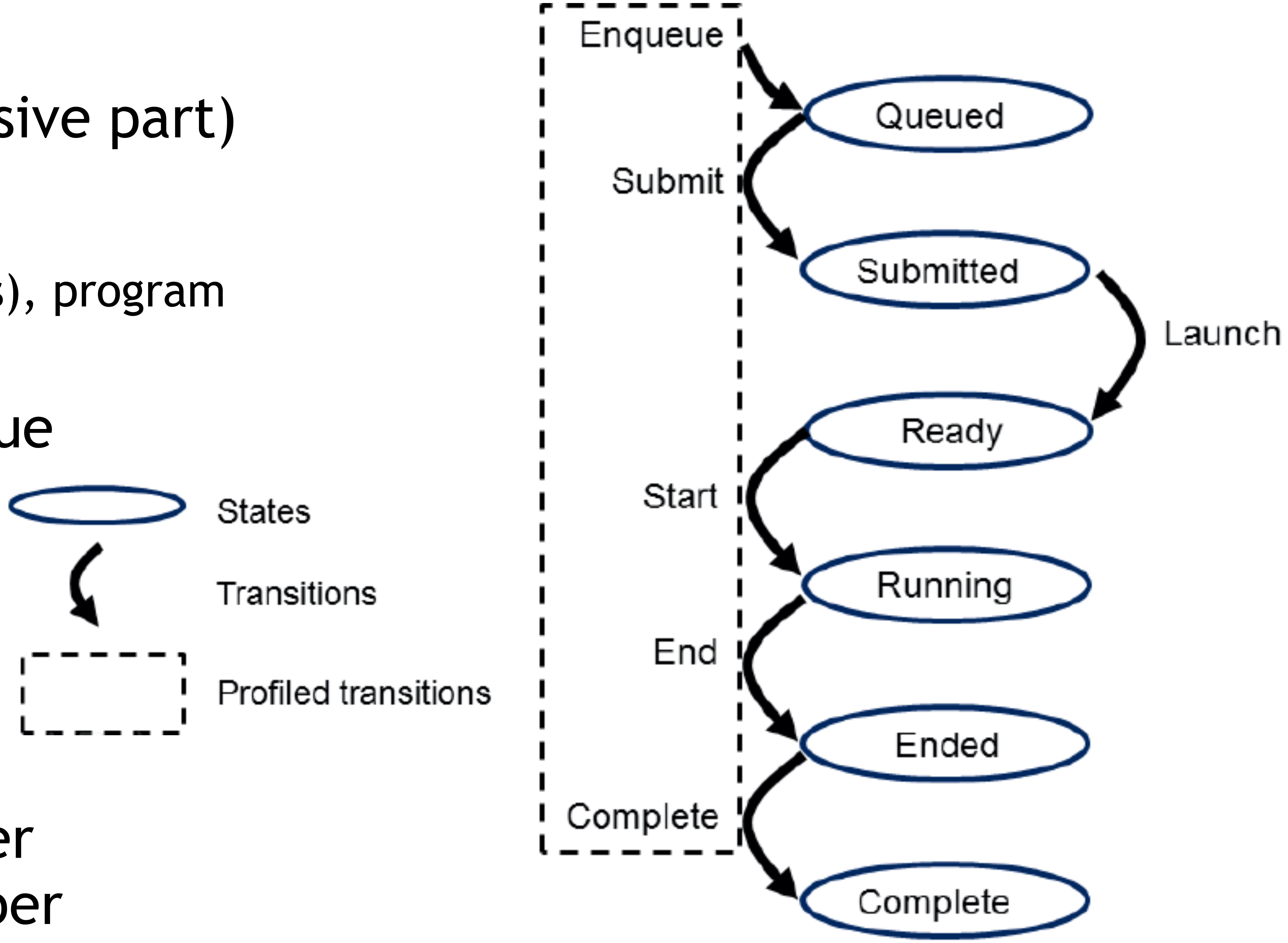


Platform model

Khronos OpenCL Working Group, „The OpenCL Specification“, Version: 2.0, Document Revision: 19

EXECUTION MODEL - OVERVIEW

- Host code (sequential parts, control)
- Kernels -> device (computational intensive part)
- Context
  - Devices, kernel objects (OpenCL functions), program objects, memory objects
- Each device has a (host) command queue
  - Kernel-enqueue commands
  - Memory commands
  - Synchronization commands
- Additionally: device command queues
- Event objects, in-order and out-of-order execution, multiple command queues per context



Khronos OpenCL Working Group, „The OpenCL Specification“, Version: 2.0, Document Revision: 19

EXECUTION MODEL - NDRANGE

Work item: kernel function in execution (kernel instance) for a single point in the defined index space

- Global ID based on its coordinates in the index space
- Or: work group ID + local ID

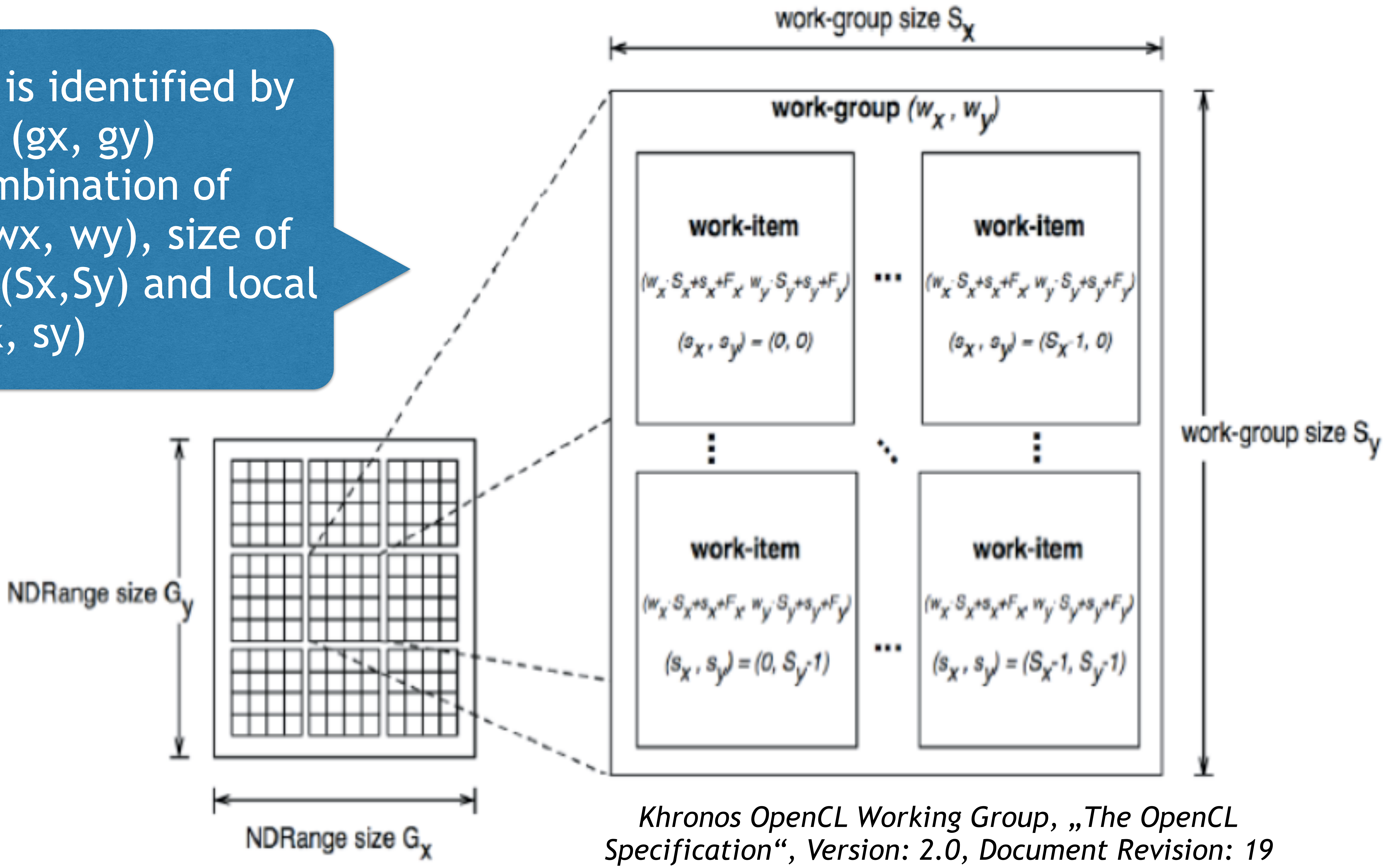
Work group: organization structure of work items with a given kernel instance (coarse grained decomposition of the index space)

NDRange: N-dimensional index space supported by OpenCL

- Decomposed into work groups
- Defined by size of each dimension, offset indices (F) per dimension, work group size for each dimension, global ID: N-dimensional tuple ([F;F+size-1])

EXECUTION MODEL - NDRANGE

Each work-item is identified by global ID (gx, gy)
or by the combination of work group ID (wx, wy), size of each work group (Sx,Sy) and local ID (sx, sy)



Khronos OpenCL Working Group, „The OpenCL Specification“, Version: 2.0, Document Revision: 19

EXECUTION MODEL – EXECUTION OF KERNEL INSTANCES

Kernel-enqueue command: host program enqueues kernel object with NDRange and work group decomposition to command queue

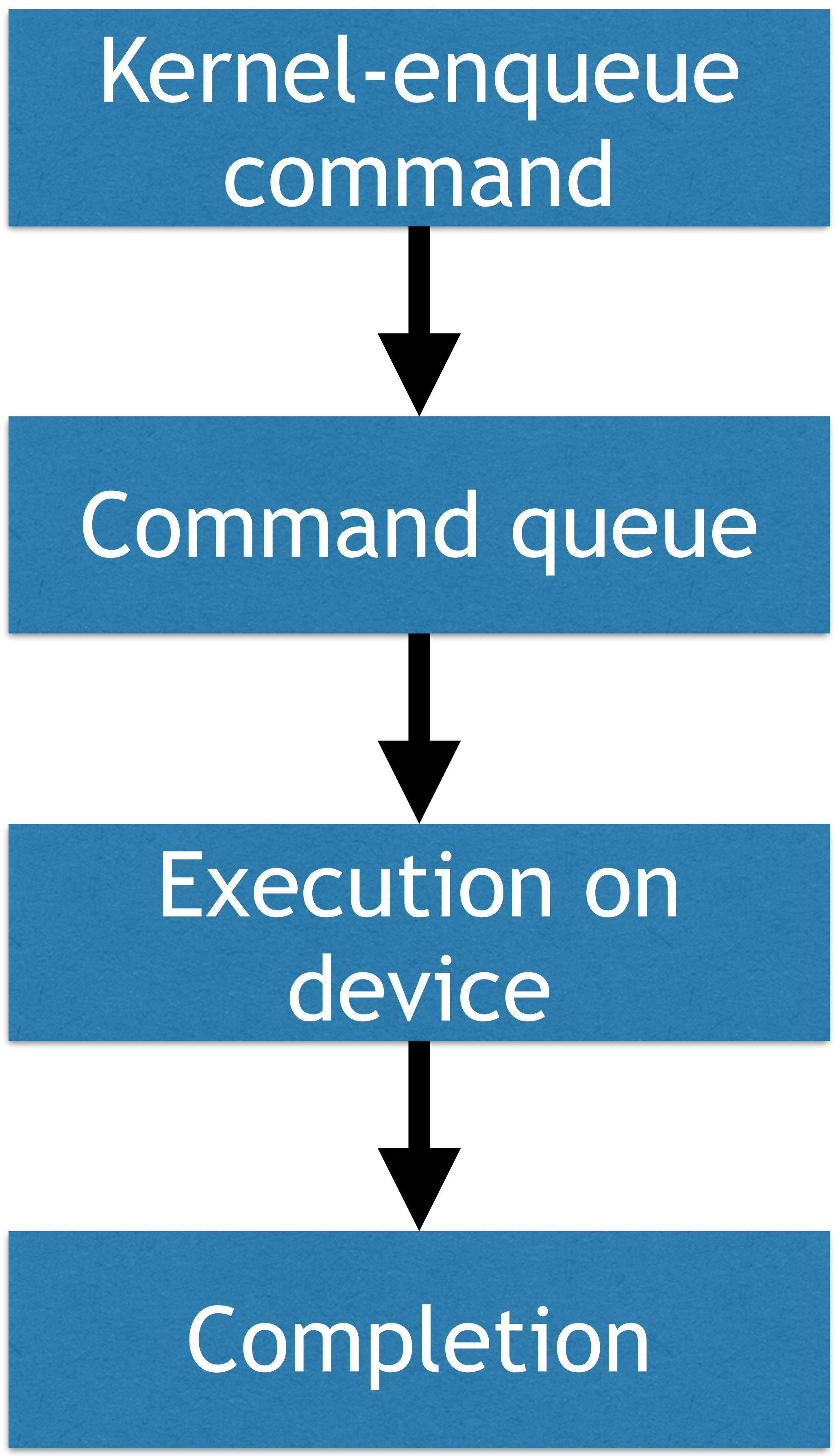
Command queue: determines when to submit kernel instance to device? (in-order queues, OOO queues)

Launching kernel instance: associated work groups are placed in the work pool (ready-to-execute work groups)

- Kernel enqueue command completes when all work groups have ended, updates to memory are globally visible, and device signals successful completion
- No constraints on scheduling as long as all work groups will eventually execute

Multiple command queues: feeding into a single work pool

Device-side enqueue using nested parallelism (OOO)



EXECUTION MODEL - SYNCHRONIZATION

Synchronization between work groups not possible

- Work groups may be serialized or not, no guarantees for parallel execution
- No guarantees for independent progress

Synchronization between work items of a single work group only using high-level constructs as barriers

- Again: no guarantees for independent progress. Thus any kind of active-wait synchronization (spin locks) is not portable

-> No forward progress or ordering relations between work groups

Work-group synchronization: constraints on the order of execution for work items in a single work group

- Work group function (collective)
- Barrier, reduction, broadcast, prefix sum, predicate evaluation

EXECUTION MODEL: KERNEL CATEGORIES

OpenCL kernels: kernel-objects associated with kernel functions within program-objects (user kernels)

Native kernels: execution along with OpenCL kernels on a device and shared memory objects

- Functions created outside of OpenCL, accessed within OpenCL through a function pointer

Built-in kernels: specific to a particular device, not built at run time (fixed-function hardware)

All use the command queue model and synchronization semantics

MEMORY MODEL - OVERVIEW

Memory regions: distinct memories visible to both host and device that share a context

Memory objects: objects defined by the OpenCL API and their management by the host and devices

Shared Virtual Memory (SVM): a virtual address space exposed to host and devices within a single context

Consistency model: constraints/guarantees on visibility of updates and reads, including atomic operations and memory fences

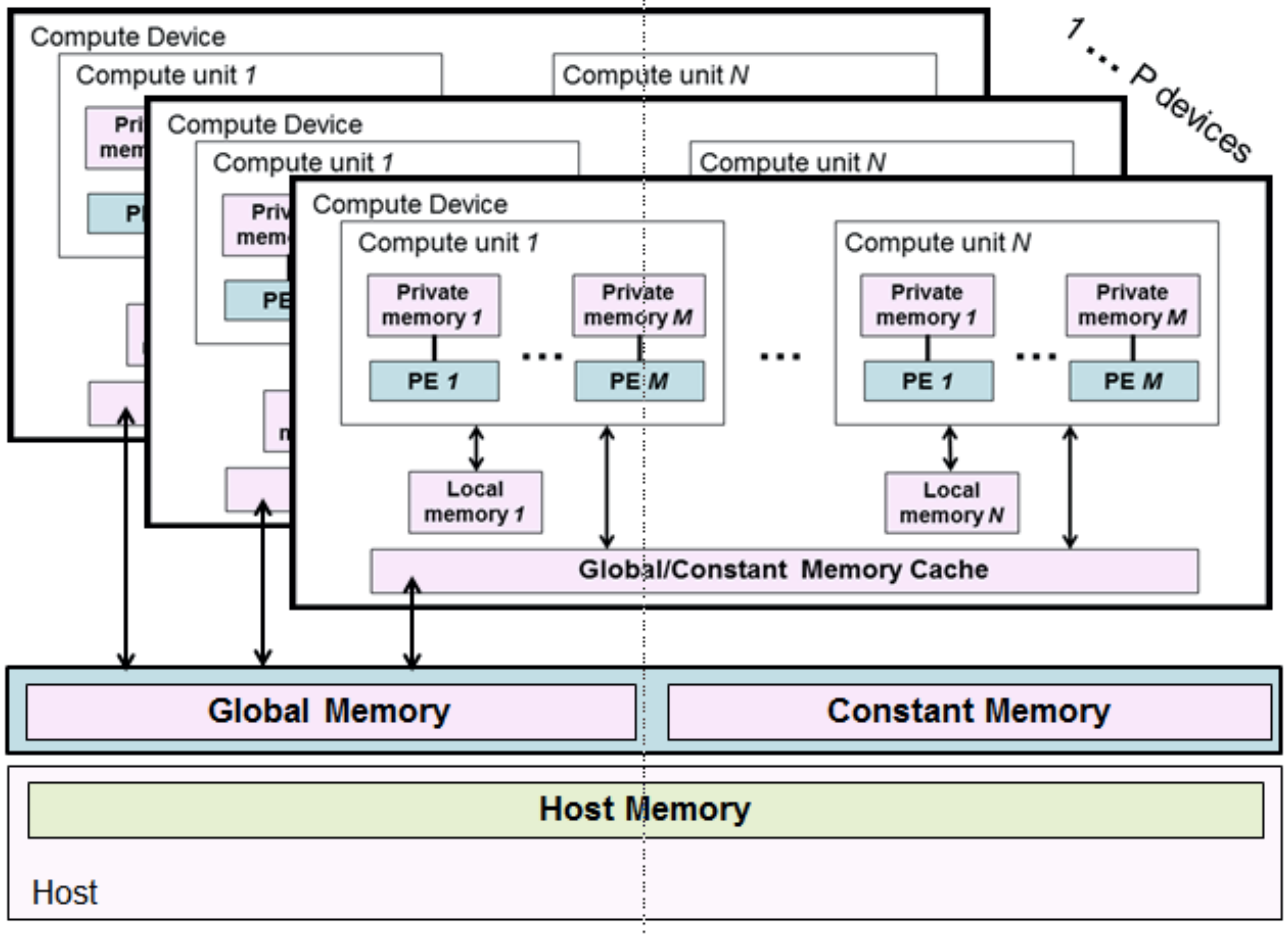
> "Consistency model defines constraints on the order in which memory operations must appear to be performed (become visible)"

MEMORY MODEL – NAMED ADDRESS SPACES

Global memory: addresses not preserved between kernel instances or host/device

SVM: alternative that logically extends global memory to include host memory

Optional: caches



Khronos OpenCL Working Group, „The OpenCL Specification“, Version: 2.0, Document Revision: 19

MEMORY MODEL – MEMORY OBJECTS

Buffer: a block of contiguous memory used as general purpose object

- Usually manipulation using pointers

Image: a buffer that holds one- to three-dimensional images as an opaque data structure managed by special functions

- RW access not supported
- OpenCL 2.0: read and write supported with special synchronization and fence operations

Pipe: an ordered sequence of data items with two endpoints (read and write)

- In particular supporting producer/consumer patterns

Allocated by host functions, modifications either using pointers or managed by the OpenCL runtime.

MEMORY MODEL - MEMORY CONSISTENCY MODEL

Memory consistency model: guarantees for programmers and restrictions for compiler writers

- OpenCL consistency model is based on ISO C11

Release consistency (RC): “The system is said to provide RC, if all write operations by a certain node are seen by the other nodes after the former releases the object and before the latter acquire it.“

- Instead of globally updating memory, RC considers locks on areas of memory, and propagates only the locked memory as needed

Definition:

1. Before a non-sync access is performed, all previous acquires by the process must have completed
2. Before a release is performed, all previous reads/writes must have completed
3. Acquire/release is sequentially consistent (RCsc)

- Eager: actions guaranteed to happen for releases
- Lazy: actions guaranteed to happen for subsequent acquires

MEMORY MODEL - MEMORY CONSISTENCY MODEL

User can control memory relaxation: at least for synchronization operations like atomics, fences; user can also control scope

		store	load
memory_order_relaxed	implies no ordering constraints	-	-
memory_order_acquire	acquire semantics	-	acquire
memory_order_release	release semantics	release	-
memory_order_acq_rel	both acquire and release semantics	release	acquire
memory_order_seq_cst	implies sequential consistency	release	acquire

MEMORY MODEL - MEMORY CONSISTENCY MODEL

Good news: most programmers won't see these details

Instead, the following guidelines are sufficient (functionality & performance)

1. Only use synchronization points within command queues to ensure safe sharing of global memory objects
2. Only use work group functions (like barriers) to synchronize within work groups
3. Restrict use of consistency parameters to `memory_order_seq_cst` with `memory_scope_device`/`memory_scope_all_svm_devices`
4. Ensure that program is race-free

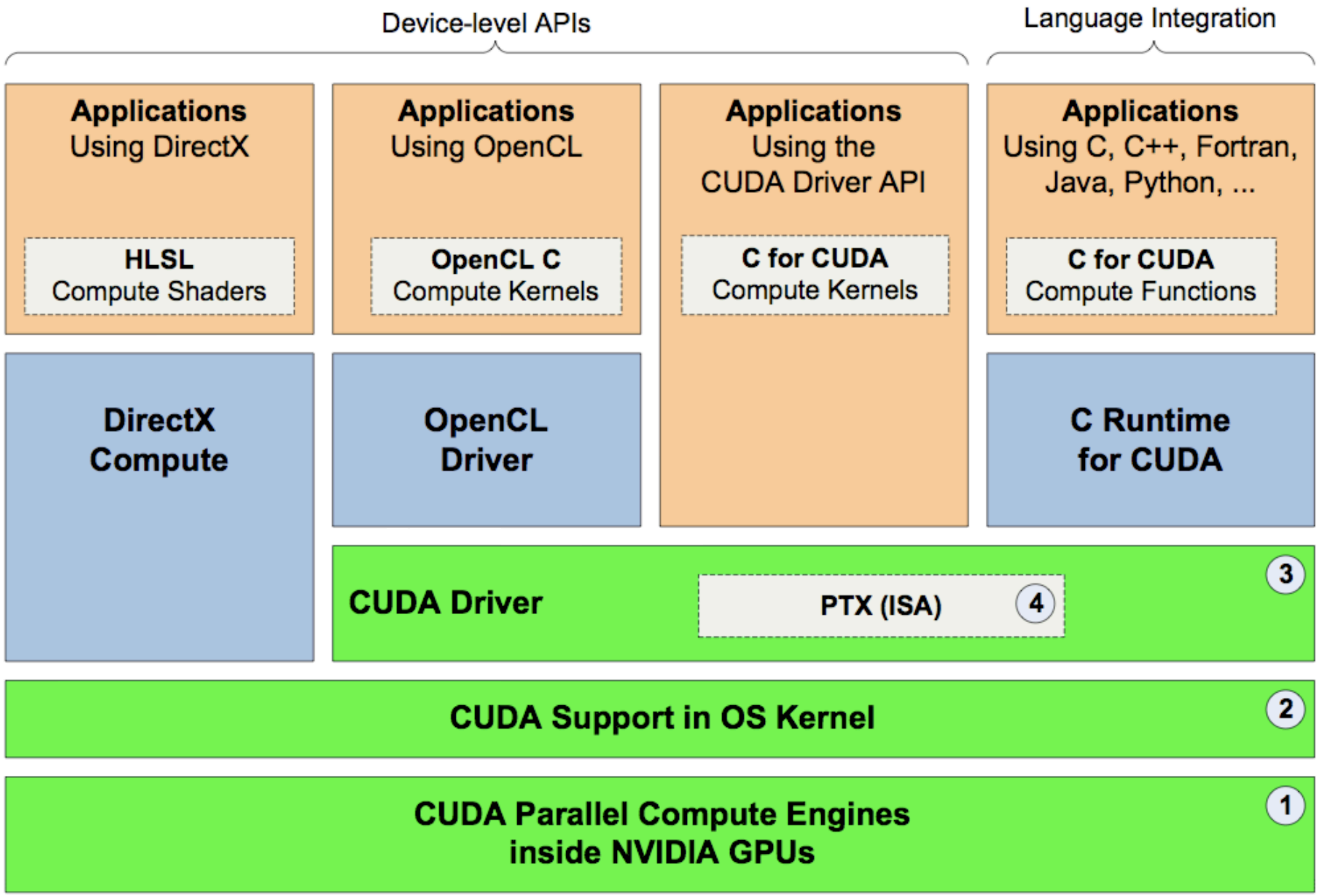
OPENCL VS. CUDA IN A NUTSHELL

BASICS COMPARED

	CUDA	OpenCL
What it is	HW architecture, ISA, programming language, API, SDK and tools	Open API and language specification
Proprietary or open technology	Proprietary	Open and royalty-free
When introduced	Q4 2006	Q4 2008
SDK vendor	Nvidia	Implementation vendors
Free SDK	Yes	Depends on vendor
Multiple implementation vendors	No, just Nvidia	Yes: Apple, Nvidia, AMD, IBM
Multiple OS support	Yes: Windows, Linux, Mac OS X; 32 and 64-bit	Depends on vendor
Heterogeneous device support	No, just Nvidia GPUs	Yes
Embedded profile available	No	Yes

Sami Rosendahl, CUDA and OpenCL API comparison, https://wiki.aalto.fi/download/attachments/40025977/Cuda+and+OpenCL+API+comparison_presented.pdf

SYSTEM ARCHITECTURE



Sami Rosendahl, CUDA and OpenCL API comparison, https://wiki.aalto.fi/download/attachments/40025977/Cuda+and+OpenCL+API+comparison_presented.pdf

EXECUTION MODEL TERMINOLOGIES

CUDA	OpenCL
Grid	NDRange
Thread Block	Work group
Thread	Work item
Thread ID	Global ID
Block index	Block ID
Thread index	Local ID

Sami Rosendahl, CUDA and OpenCL API comparison, https://wiki.aalto.fi/download/attachments/40025977/Cuda+and+OpenCL+API+comparison_presented.pdf

MEMORY MODEL TERMINOLOGIES

CUDA	OpenCL
Host memory	Host memory
Global or Device memory	Global memory
Local memory	Global memory
Constant memory	Constant memory
Texture memory	Global memory
Shared memory	*Local* memory
Registers	Private memory

Sami Rosendahl, CUDA and OpenCL API comparison, https://wiki.aalto.fi/download/attachments/40025977/Cuda+and+OpenCL+API+comparison_presented.pdf

OPENCL EXAMPLES

OPENCL EXAMPLE CODE

```
int main (int argc , const char * argv [])
{
  // Select platform
  cl_uint num_platforms ; cl_platform_id platform ;
  cl_int err = clGetPlatformIDs  ( 1, &platform , &num_platforms );

  // Select device
  cl_device_id device ;
  clGetDeviceIDs ( platform, CL_DEVICE_TYPE_GPU, 1, &device , 0 );

  // Create context & command queue
  cl_context context = clCreateContext  ( 0, 1, &device , 0, 0, &err );
  cl_command_queue cmd_queue = clCreateCommandQueue ( context, device , 0, 0 );

  // Prepare kernel
  cl_program program = clCreateProgramWithSource (context, 1, &kernel_src, 0, &err);
  clBuildProgram ( program, 0, 0, 0, 0, 0 );
  cl_kernel kernel = clCreateKernel ( program, "example", &err );
...
```

OPENCL EXAMPLE CODE

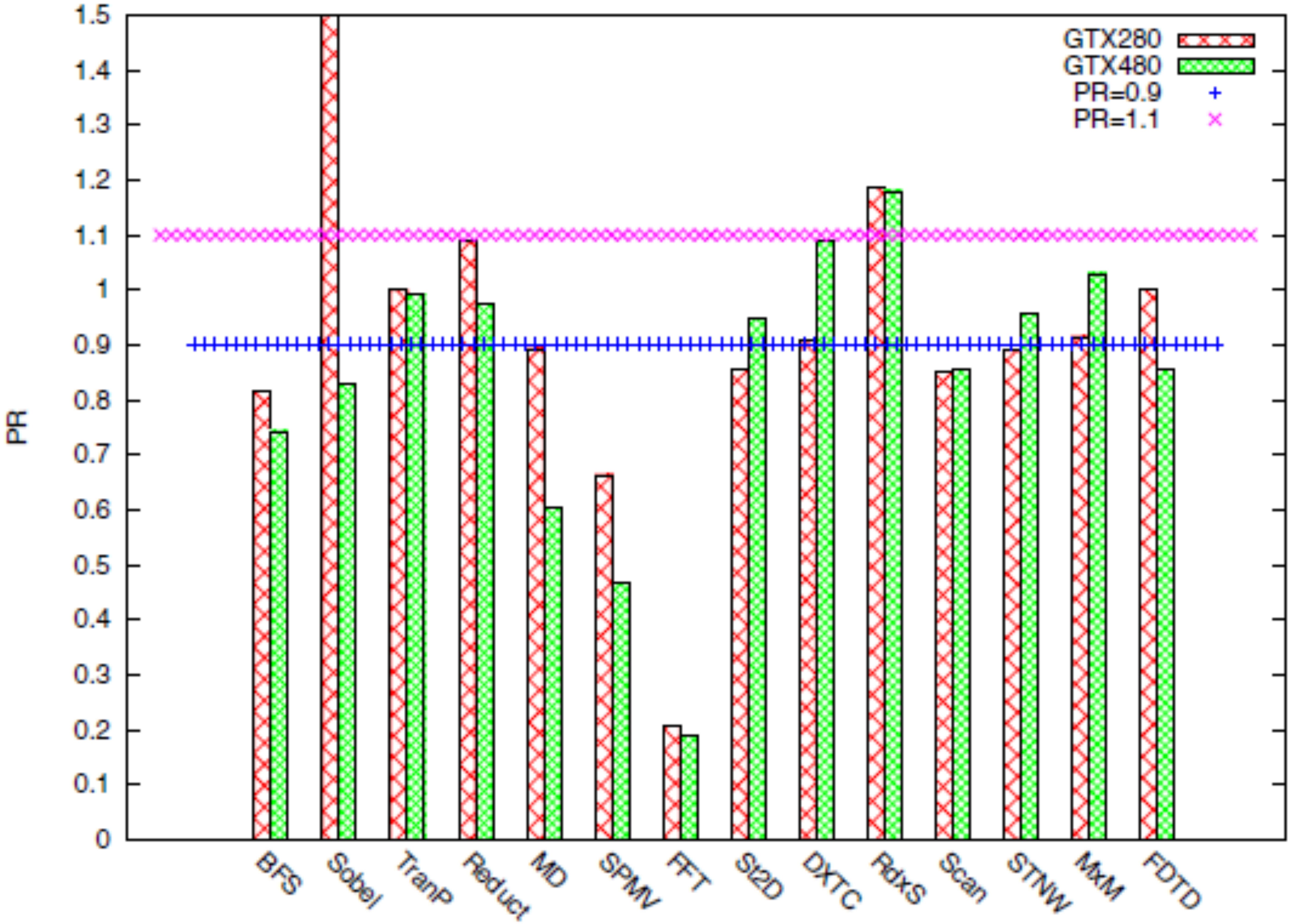
```
...
// Create buffers
cl_mem Ad = clCreateBuffer ( context, CL_MEM_READ_ONLY, sizeA, 0, 0 );

// Reserve memory
clSetKernelArg ( kernel, 0, sizeof ( cl_mem ), &d_A );

// Configure work group
size_t ws_global [] = { 512, 512 };
size_t ws_local [] =  {  16,  16 };  // 256 items per group

// Copy input data, execute kernel, copy output data back
clEnqueueWriteBuffer   ( cmd_queue, d_A, CL_FALSE, 0, sizeA, h_A, 0, 0, 0 );
clEnqueueNDRangeKernel ( cmd_queue, kernel, 2, 0, ws_global, ws_local, 0, 0, 0 );
clEnqueueReadBuffer    ( cmd_queue, d_A, CL_FALSE, 0, sizeA, h_A, 0, 0, 0 );
clFinish ( cmd_queue );
}
```

OPENCL EXAMPLE CODE

```
const char kernel_src [] =
  " __kernel void example ( __global const float *A, ..., int wA, int wB )"
  "{                                                                       "
  "int i = get_global_id (0);                                              "
  "int j = get_global_id (1);                                              "
  "                                                                        "
  "...                                                                     "
  "...                                                                     "
  "};                                                                      "
```

PERFORMANCE COMPARISON: OPENCL VS. CUDA

- Performance Ratio (PR) > 1: OpenCL is faster
- MD, SPMV: rely on texture memory
  - Remove -> similar performance results
- Sobel: GTX280 had no L1 cache
- FDTD: loop unrolling, CUDA unrolls more
  - Remove -> similar performance results
- FFT: see next slide



Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. 2011. A Comprehensive Performance Comparison of CUDA and OpenCL. In Proceedings of the 2011 International Conference on Parallel Processing (ICPP '11).

PERFORMANCE COMPARISON: FFT KERNEL

OpenCL performs much worse than CUDA, code relies on loop unrolling

TABLE V
STATISTIC FOR PTX INSTRUCTIONS

Class	Instructions	Instruction Count CUDA	Instruction Count OpenCL	Class	Instructions	Instruction Count CUDA	Instruction Count OpenCL
Arithmetic	add	93	191	Data Movement	cvt	16	16
	sub	83	95		mov	687	88
	mul	33	138		ld.param	1	1
	div	0	2		ld.local	97	64
	fma	0	37		ld.shared	32	32
	mad	2	22		ld.const	0	24
	neg	9	36		ld.global	8	8
	and	1	291		st.local	250	78
Sub-total		220	521		st.shared	32	32
Logic Shift	or	2	33		st.global	8	8
	not	0	4	Sub-total		1131	351
	xor	0	4	Flow Control	setp	2	80
	shl	0	50		selp	0	40
	shr	1	43		bra	2	68
Sub-total		4	163	Sub-total		4	188
Synchronization	bar	7	7	Total		1366	1230

2. slightly more for OpenCL

3. identical

4. much more for CUDA

5. much less for CUDA

1. almost the same

Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. 2011. A Comprehensive Performance Comparison of CUDA and OpenCL. In Proceedings of the 2011 International Conference on Parallel Processing (ICPP '11).

WRAPPING UP

SUMMARY

A wealth of features comes with a wealth of complexity

Geared to a variety of devices, from embedded mobile to supercomputers

Comprehensive execution model

Many similarities to CUDA

- Vendor-specific: faster updates, but limited usage

OpenCL: generic language

- NVIDIA, AMD GPUs, CELL, Intel MIC, CPUs, FPGAs, …

CUDA	OpenCL
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory
Thread	Work Item
Thread Block	Work Group