OpenCL Basics

Wolfram Schenck | Faculty of Eng. and Math., Bielefeld University of Applied Sciences |

Vectorisation and Portable Programming using OpenCL, 21.-22.11.2017

Overview of the Lecture

OpenCL Overview

OpenCL

OpenCL (Open Computing Language)

Programming framework for CPUs, GPUs, DSPs, FPGAs with programming language „OpenCL C“

- Started by Apple, subsequent development with AMD, IBM, Intel, and NVIDIA, meanwhile managed by Khronos Group
  - ➔ Open and royalty–free standard
- **Goal:** Programming framework for portable, parallel programming of devices in heterogeneous environments (CPUs, GPUs, and other processors; from smartphone to supercomputer)
- Dec. 2008: OpenCL 1.0
- June 2010: OpenCL 1.1
- Nov. 2011: **OpenCL 1.2**
- Nov. 2013: OpenCL 2.0
- Nov. 2015: OpenCL 2.1

CUDA C vs. OpenCL

CUDA C

PRO

- ✓ Mature and efficient
- ✓ Many tools and extra libraries

CONTRA

- ✗ Only usable for GPUs by NVIDIA

OpenCL

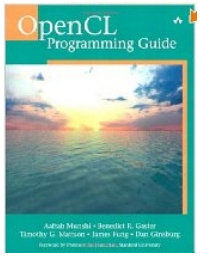
PRO

- ✓ For various processor types (independent from manufacturer)
- ✓ Supports heterogeneous platforms

CONTRA

- ✗ Not as mature and as widely used as CUDA C
- ✗ Partly long–winded programming necessary

Additional Information



OpenCL Programming Guide
Aaftab Munshi, Benedict Gaster, Timothy G. Mattson,
James Fung, Dan Ginsburg
Addison-Wesley, 2011, ISBN: 978-0321749642

- Khronos website:
  http://www.khronos.org/opencl
  (News, specifications, **MAN pages**)
- AMD OpenCL Zone:
  http://developer.amd.com/tools-and-sdks/opencl-zone/
- Intel Developer Zone on OpenCL:
  https://software.intel.com/de-de/forums/opencl
- EBook on OpenCL:
  http://www.fixstars.com/en/opencl/book/. . .
  . . . /OpenCLProgrammingBook/contents/

Tutorials

- OpenCL — Introduction for HPC Programmers:
  https://software.intel.com/en-us/articles/. . .
  . . . /tutorial-opencl-introduction-for-hpc-programmers
  (from V. Kartoshkin and T. Mattson)
- OpenCL–Tutorials on PRACE website:
  http://www.training.prace-ri.eu/. . .
  . . . /training_material/?tx_pracetmo_pi1[tag]=opencl
  - Heterogeneous Programming. OpenCL and High Level Programming Tools
  - PATC Course: Programming paradigms for new hybrid architectures
  - CUDA and OpenCL

Architecture of OpenCL

At the conceptional level:

- Platform model
- Execution model
- Memory model
- Programming model

At the programming level:

- OpenCL Platform API
- OpenCL Runtime API
- OpenCL C (programming language)

Platform Model

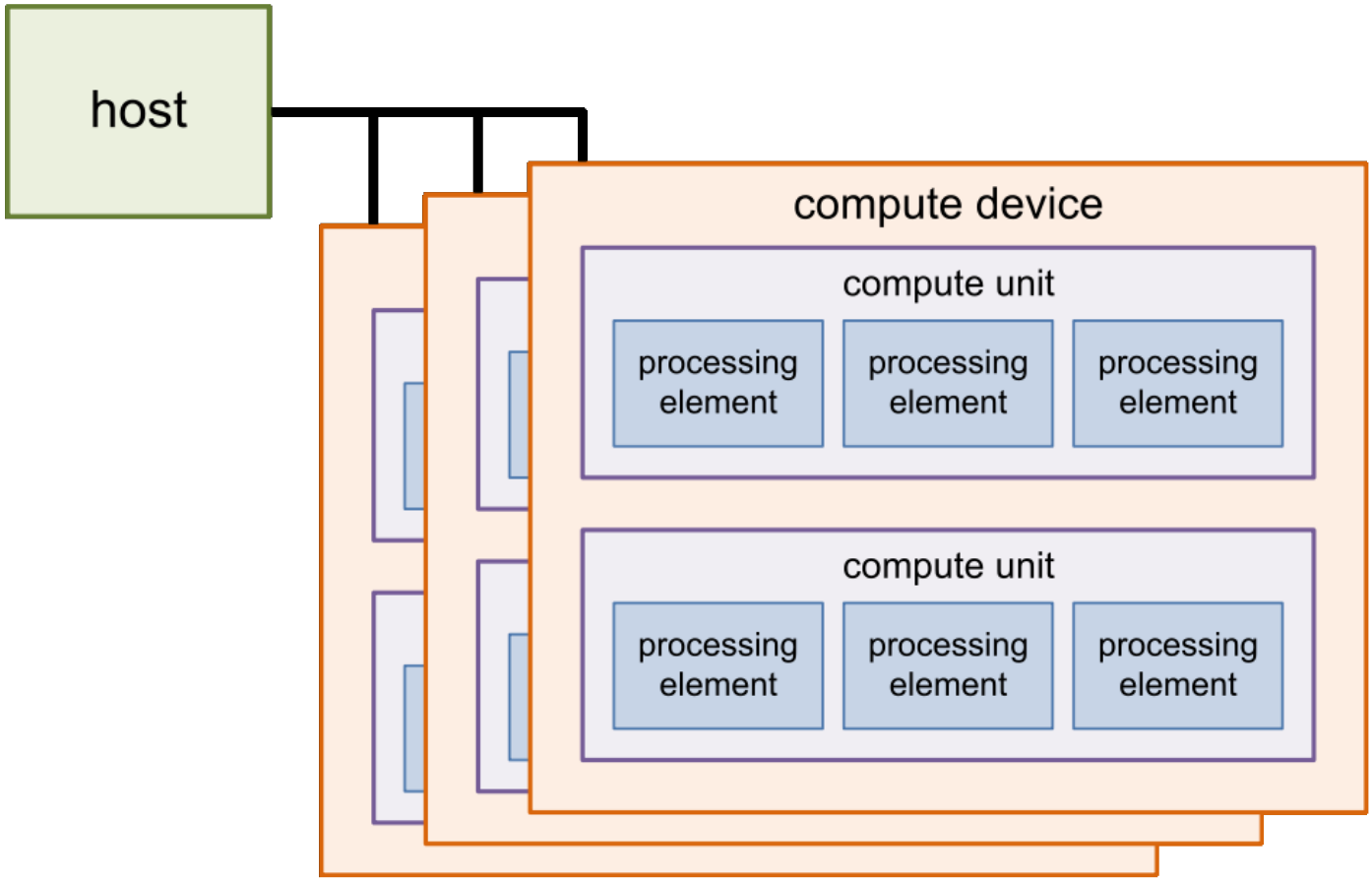


Fig.: Wikipedia

Platform Model (cont.)

- **Basic structure:** Host which is connected to several devices
- **Host:** Computational unit on which the host program runs
  - Usually: CPU of the computer system
- **Device:** Computational unit which is accessed via OpenCL library
  - Examples: CPUs, GPUs, DSPs, FPGAs
- **Further subdivision:**
  - Device $\longrightarrow$ „Compute Units“
  - Compute Unit $\longrightarrow$ „Processing Elements“

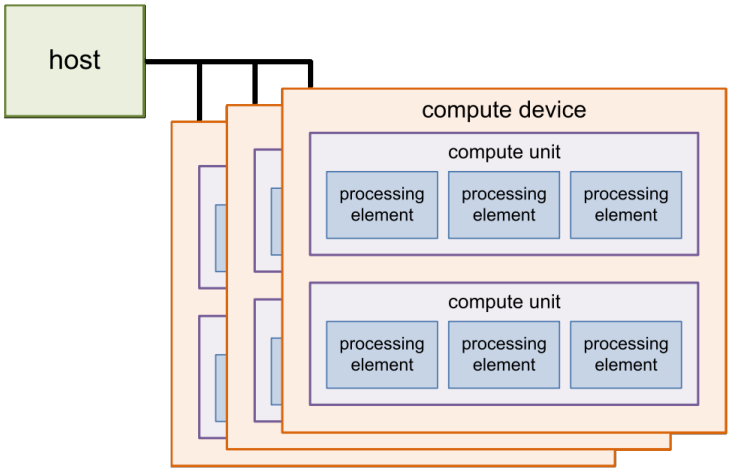


Fig.: Wikipedia

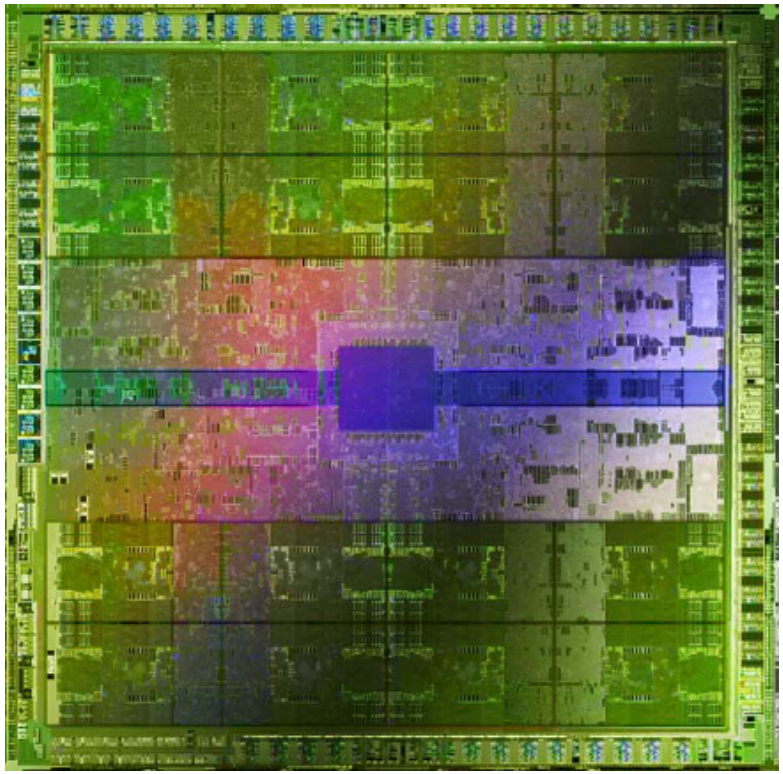
Platform Model (CPU)

CPU

- **Device:** All CPUs on the mainboard of the computer system
- **Compute unit (CU):** One CU per core (or per hardware thread)
- **Processing element (PE):** 1 PE per CU, or if PEs are mapped to SIMD lanes, *n* PEs per CU, where *n* matches the SIMD width

Fermi: GF100/GF110

16 Streaming Multi–Processors (SM)



Die shot

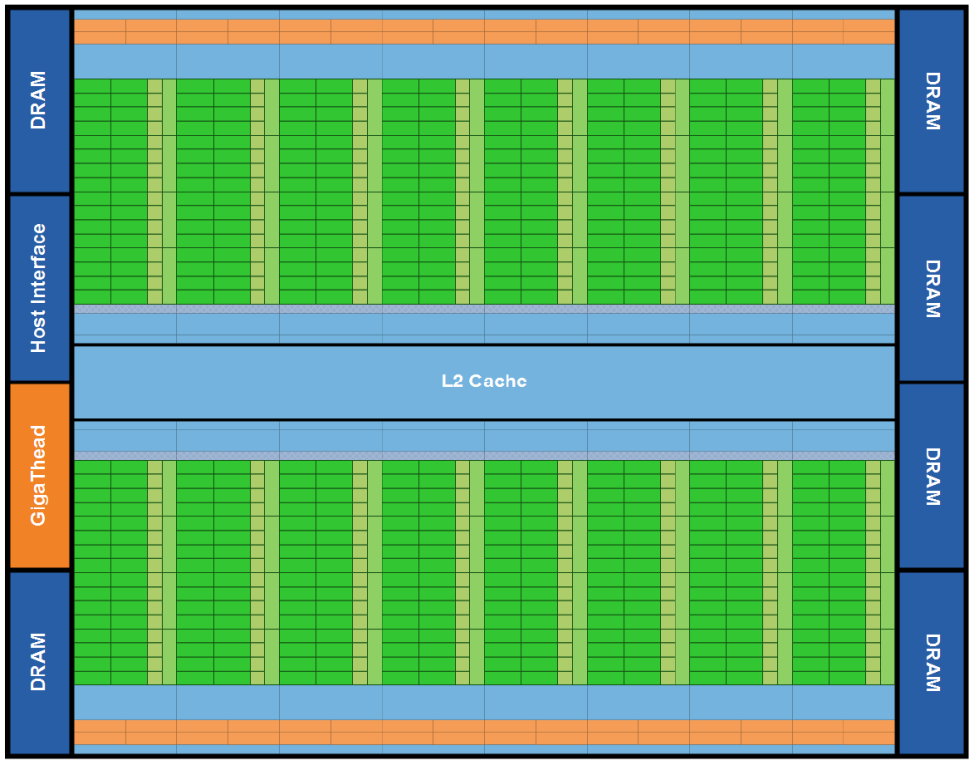
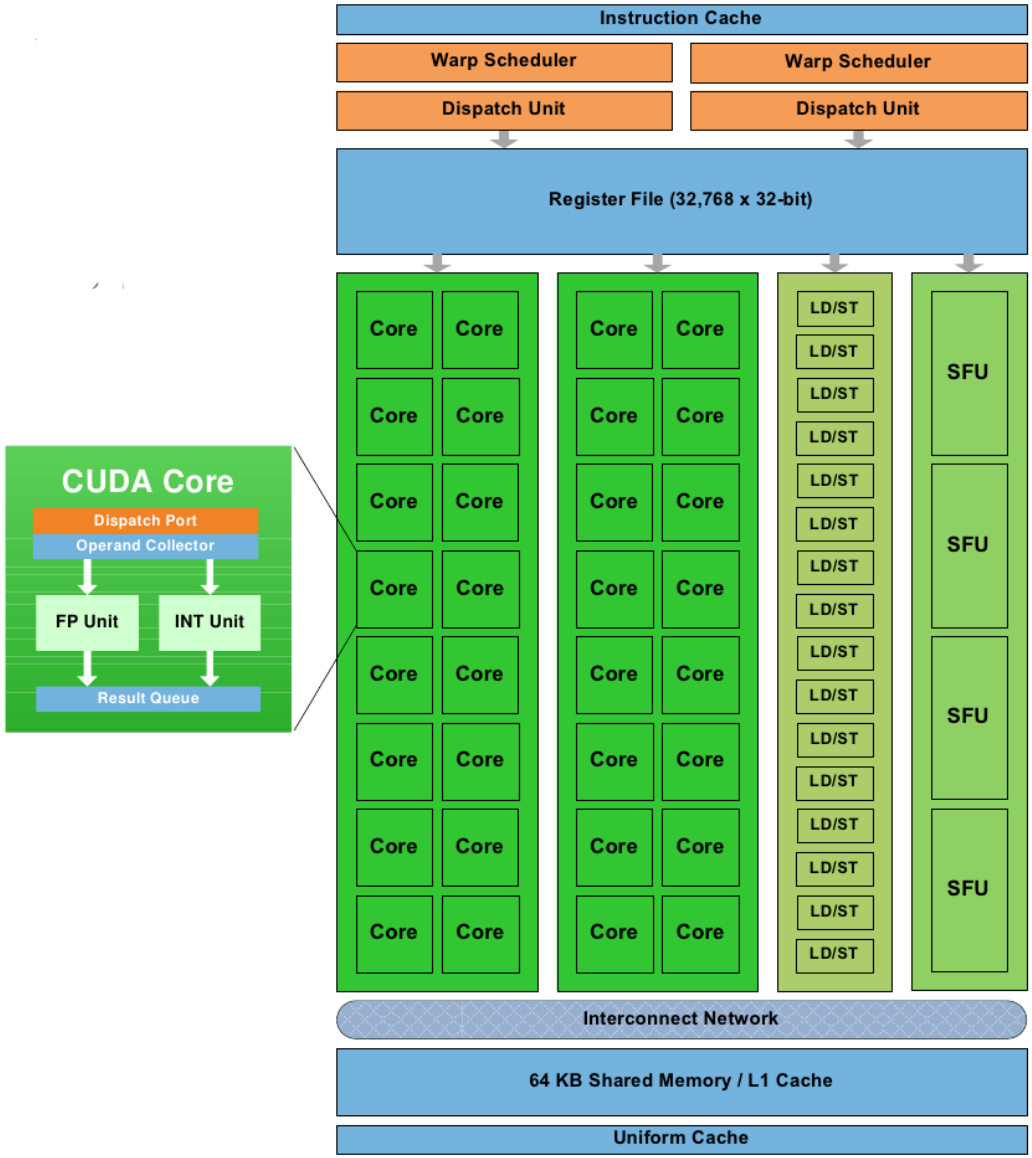


Fig.: NVIDIA

GF100/GF110: Streaming Multi–Processor (SM)



Fermi Streaming Multiprocessor (SM)

Fig.: NVIDIA

SM properties

- 32 CUDA cores (Streaming processors/SP)
- 16 Load/store units
- 4 Special function units (SFU)
- 2 Warp scheduler

→ 512 ALUs/FPUs available

Platform Model (GPU and MIC)

GPU

- **Device:** Each GPU in the system acts as single device
- **Compute unit (CU):** One CU per multi–processor (NVIDIA)
- **Processing element (PE):** 1 PE per CUDA core (NVIDIA) or "SIMD lane" (AMD)

MIC

- **Device:** Each MIC in the system acts as single device
- **Compute unit (CU):** One CU per hardware thread ($= 4 \times$ [# of cores $- 1$])
- **Processing element (PE):** 1 PE per CU, or if PEs are mapped to SIMD lanes, *n* PEs per CU, where *n* matches the SIMD width

Platform Model (cont.)

Platform

- Every OpenCL implementation (with underlying OpenCL library) defines a so–called „platform“.
- Each specific platform enables the host to control the devices belonging to it.
- Platforms of various manufacturers can coexist on one host and may be used from within a single application (**ICD:** „installable client driver model“).

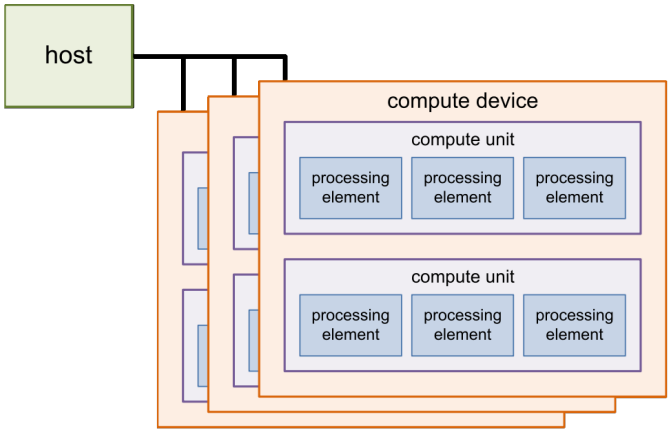


Fig.: Wikipedia

Platform Model

Practical Hints

Get OpenCL running under Linux

- **Header files:** Get from Khronos website (e.g.)
  - Central file: `CL/cl.h`
- **OpenCL library stub with ICD loader:**
  Get from one of the vendors of your OpenCL devices
  - Central file: `libOpenCL.so`
- **ICD definition files and platform–specific OpenCL libraries:**
  Get from all the vendors of your OpenCL devices
  - ICD files usually located in: `/etc/OpenCL/vendors/`
- **Mechanism at runtime:**
  - `libOpenCL.so` is dynamically linked to your application at runtime
  - ICD loader uses `dlopen(..)` to open all required platform–specific OpenCL libraries
  - Calls to OpenCL library functions are routed to the correct implementation

Execution Model

Example: 2D–Arrangement of Work–Items

S_x, S_y : Number of work–items within a single work–group

s_x, s_y : Indices of a work–item within a work–group

w_x, w_y : Indices of a work–group within the NDRange

F_x, F_y : Global index offset

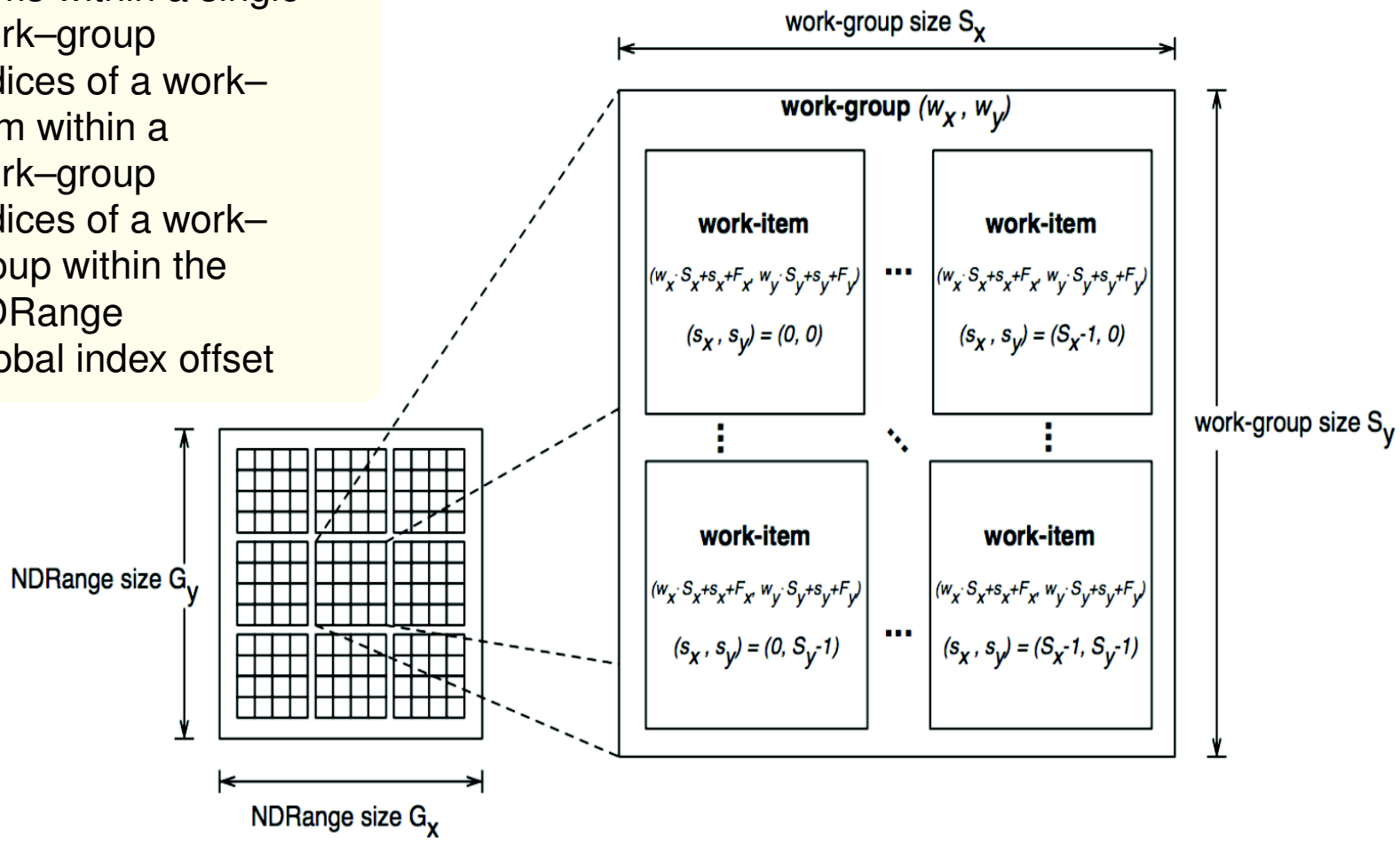


Fig.: „The OpenCL Specification 1.1“, Fig. 3.2

Excursus: Thread Management on GPUs

Kernel

- Function for execution on the device (here: GPU)
- Typical scenario: Many kernel instantiations running simultaneously in parallel threads

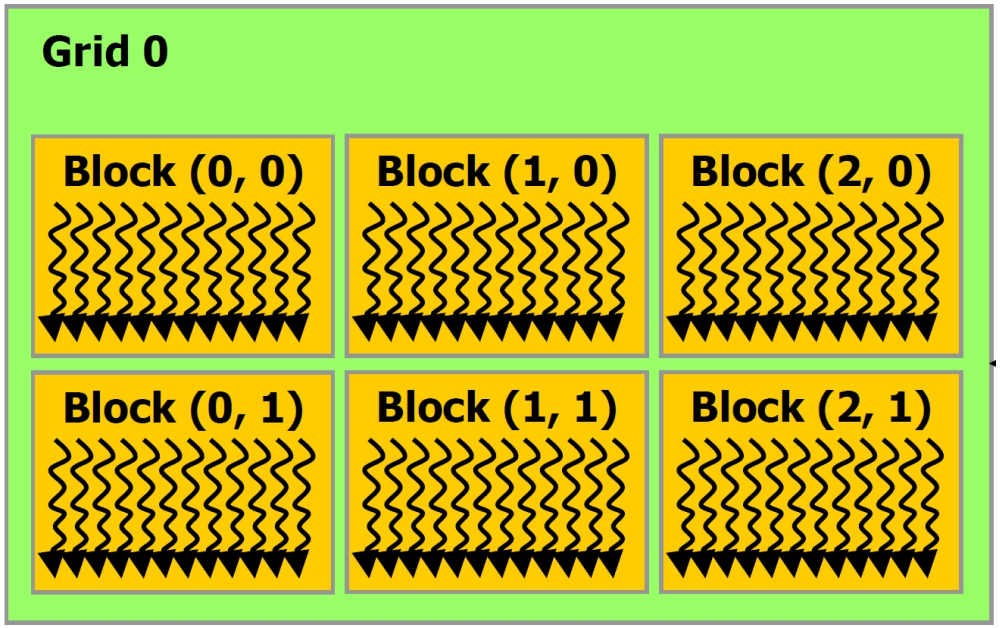
Challenge

Management of many thousands of threads

Solution

„Coarse Grained Parallelism“ → „Fine Grained Parallelism“

Thread Management (cont.)



Hierarchical thread organization

Upper level : Grid (equiv. to NDRange) $\Rightarrow$ Device
Medium level : Block (equiv. to work–group)
$\Rightarrow$ Streaming Multi–Processor (SM)
Lower level : Thread (equiv. to work–item)
$\Rightarrow$ Streaming Processor (SP)

Fig.: NVIDIA

Thread–Management (cont.)

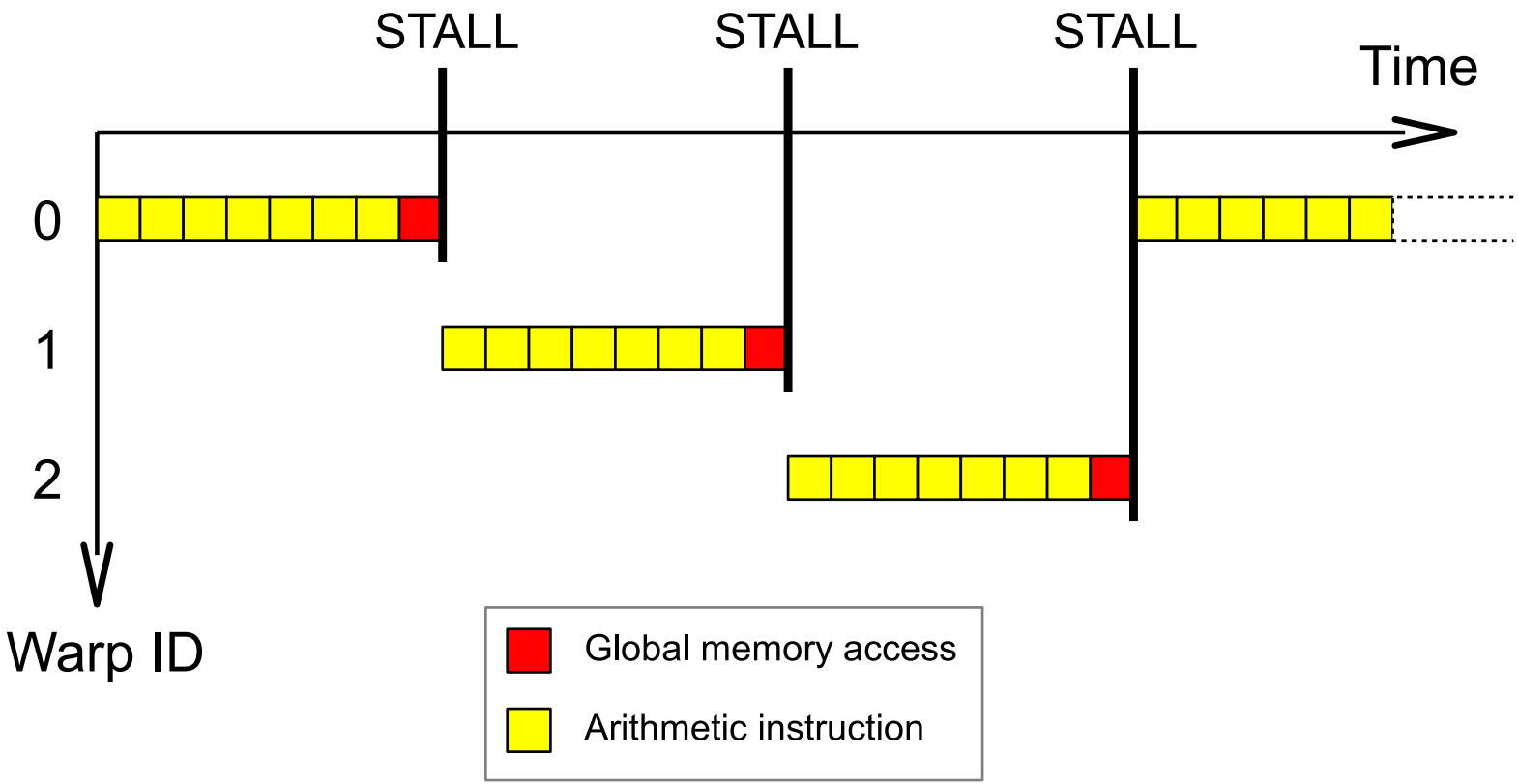
Block Scheduler ➔ „Coarse Grained Parallelism“ (NVIDIA)

- Distributes groups of work–items (“work–groups”) to SMs
- Takes free capacity into account (registers, local memory, number of work–items)
- Goal: Load–balancing (“round–robin” procedure)

Warp/Wavefront ➔ „Fine Grained Parallelism“

- **Warp** (NVIDIA): Group of 32 work–items which are scheduled and executed together (within a work–group/SM)
- **Wavefront** (AMD): Group of 64 work–items which are scheduled and executed together (within a work–group/CU)
- At this level: SIMD

Latency Hiding



Thread–Management (cont.)
(numbers for GF110)

- Up to 8 work–groups actively scheduled per SM
  - Up to 1024 work–items per work–group
    (does not result in 8×1024 ➔ see next item)
- Up to 1536 work–items per SM (organized as 48 Warps)
  - ➔ With 16 SMs:
    Max. $16 \times 1536 = 24576$ simultaneously scheduled work–items
  - Comparison with CPU:
    Several 100 threads simultaneously active
- In the grid: Up to 65535^3 work–items
  - ➔ Up to $65535^3 \times 1024 \approx 2.88 \cdot 10^{17}$ work–items per kernel call

NVIDIA Tesla Graphics Cards in Comparison

	GT200 (C1060)	GF110 (M2090)	GK110 (K20X)	GK110B (K40)
# of multi–procs. (SM/SMX)	30	16	14	15
# of cuda cores (per SM/SMX)	8	32	192	192
# of cuda cores (overall)	240	512	2688	2880
Clock (core/shader) [MHz]	602/1296	650/1300	735/735	745/875
GFLOPs (SP)	933	1331	3951	4290
GFLOPs (DP)	78	665	1317	1430
Memory bandwidth [GB/sec]	102	177	250	288
# of registers (per SM/SMX)	16384	32768	65536	65536
Shared mem. (per SM/X) [KB]	32	16–48	16–48	16–48
L1–cache (per SM/SMX) [KB]	0	16–48	16–48	16–48
L2–cache [KB]	0	768	1536	1536
Max threads per SM/SMX	1024	1536	2048	2048
Max blocks per SM/SMX	8	8	16	16
Max threads in flight	30720	24576	30720	30720

Execution Model

Components

- **Basic distinction:**
  - Host: Executes host program
  - Device: Executes device kernel
- **Hierarchy on device:**
  - NDRange $\longrightarrow$ Work–Group $\longrightarrow$ Work–Item
- Host defines **Context**:
  - Devices **(only from single platform!)**
  - Kernels (OpenCL–functions for execution on the device)
  - Program objects (kernel source code and kernels in compiled form)
  - Memory objects
- Host manages **Queues**:
  - Kernel execution
  - Operations on memory objects
  - Synchronization
    - Variants: In–order– und out–of–order execution

Memory Model

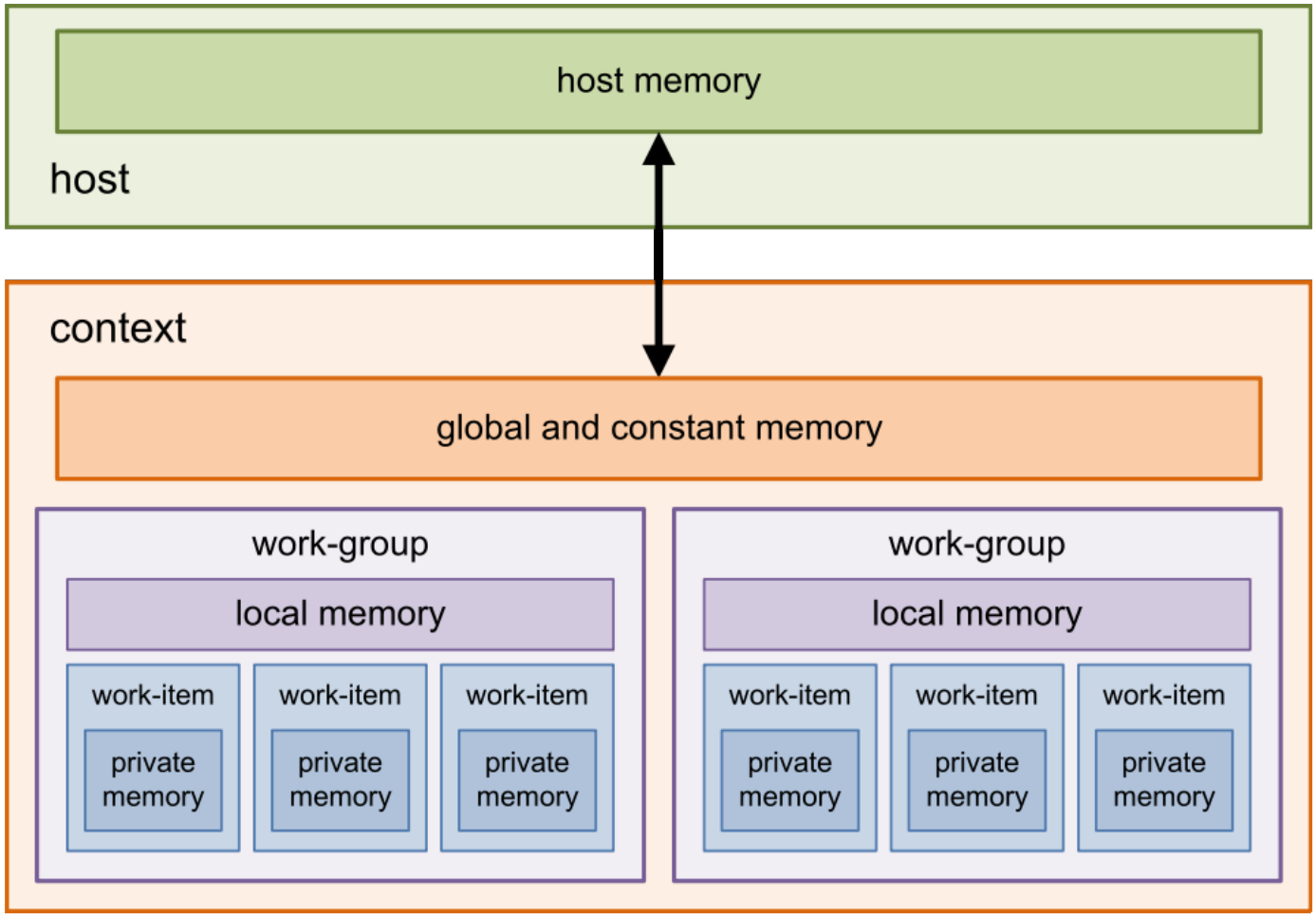


Fig.: Wikipedia

Memory Model

Allocation and Access

	Global	Constant	Local	Private
Host	Dynamic allocation Read / Write access	Dynamic allocation Read / Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read / Write access	Static allocation Read-only access	Static allocation Read / Write access	Static allocation Read / Write access

Weak Consistency Model

- **Consistency within work–group** for global and local memory: Only at synchronization points within work–group
- **Consistency between work–groups** for global memory: Only at synchronization points at host level

Table: „The OpenCL Specification 1.1“, Tab. 3.1

Programming Model

Supported Approaches

Data Parallel

- Possible mappings between data and NDRange:
  - Strict 1:1 mapping: For each data element one work–item
  - More flexible mappings also possible
- Favored device class: GPUs

Task Parallel

- Execution of only a single kernel instance
  (equivalent to an NDRange with only one work–item)
- Parallelism via:
  - SIMD units on the device (using OpenCL vector data types)
  - Multiple tasks in queue which are executed asynchronously
- Favored device class: Multi–core CPUs, multi–CPU systems

The „Big Picture“

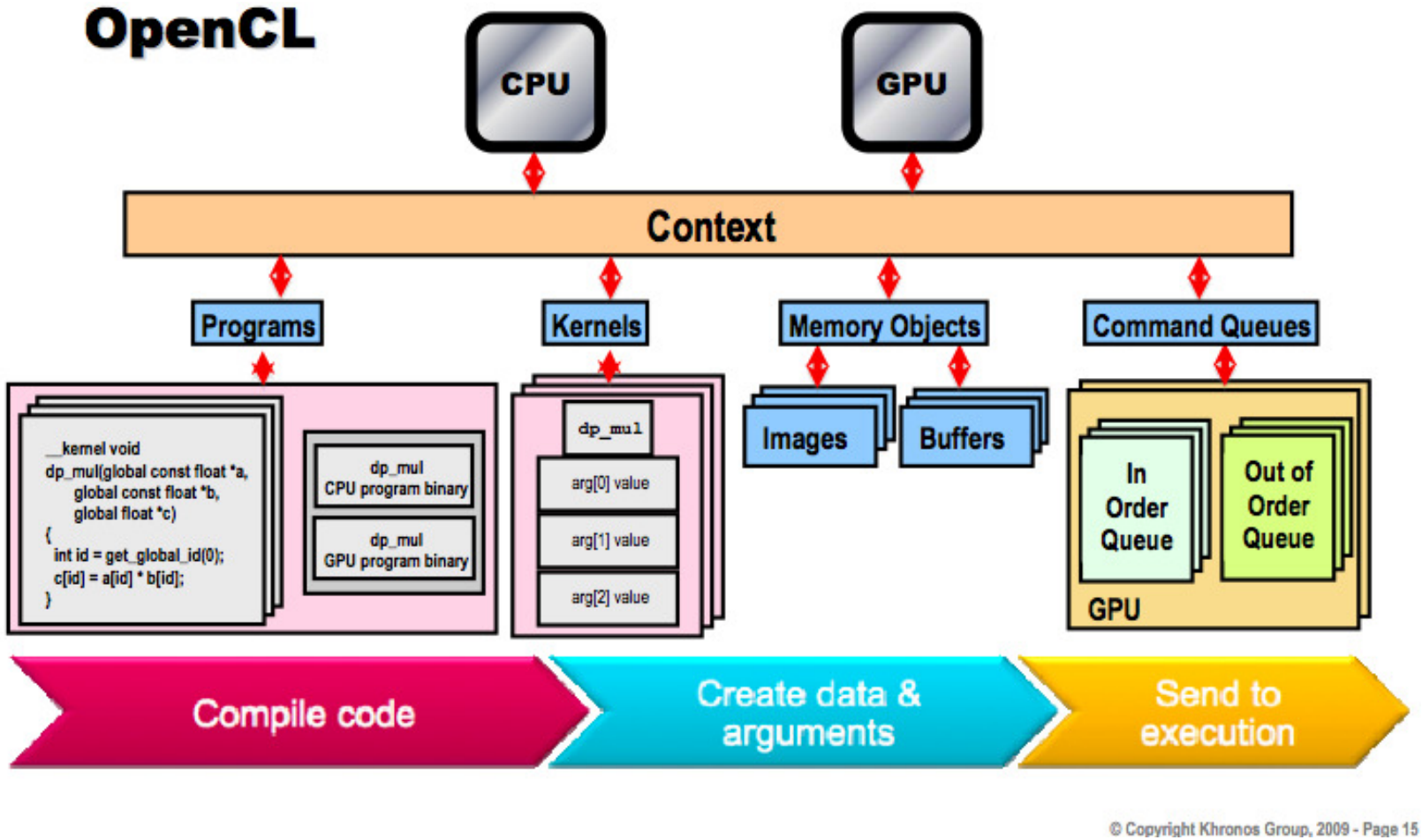


Abb.: Khronos Group

OpenCL

Basic Programming Steps in Host Code

1. Determine components of the heterogeneous system
2. Query specific properties of each component to adapt program execution dynamically during runtime
3. Compile and configure the OpenCL kernels
   - ➔ **Programming language for kernel code: OpenCL C**
4. Create and initialize memory objects (buffers, images)
5. Execution of the kernels in the correct order with the best suited device for each kernel
6. Collection of results

➔ **Functions for all these steps:**
OpenCL Platform and Runtime API

OpenCL Host API

Basic Programming Steps. . .

. . . in Practice

- Query platforms ➔ selection
- Query devices of the platform ➔ selection
- Create context for the devices
- Create queue (for context and device)
- Create program object (for context) ← from C string
  - Compile program
  - Create kernel (contained in program)
- Create memory objects (within context)
- Kernel execution:
  1. Set kernel arguments
  2. Put kernel into queue ➔ Execution
- Copy memory objects with results from device to host (invoke via queue)
- Clean up. . .

Query Platforms

```
cl_int clGetPlatformIDs( cl_uint num_entries,
                         cl_platform_id *platforms,
                         cl_uint *num_platforms );
```

Query all OpenCL platforms on the system

Return value : Error code (ideally equal to `CL_SUCCESS`)

`num_entries` : Number of pre-allocated elements of type `cl_platform_id` in the array `platforms`

`platforms` : Returns information about the platforms (for each platform one element in the array `platforms`)

`num_platforms`: Returns number of platforms

Query Platforms (cont.)

Double invocation of `clGetPlatformIDs(..)` necessary

- 1. invocation: `num_entries = 0, platforms = NULL` → Query `num_platforms`
- Allocate `num_platforms` elements of type `cl_platform_id` in the array `platforms`
- 2. invocation: `num_entries = num_platforms` → Query `platforms`

Related functions

- `clGetPlatformInfo(..)`

Query Devices

Precondition: Platform exists

```
cl_int clGetDeviceIDs( cl_platform_id platform,
                       cl_device_type device_type,
                       cl_uint num_entries,
                       cl_device_id *devices,
                       cl_uint *num_devices );
```

Query the devices belonging to the respective platform

Return value : Error code (ideally equal to `CL_SUCCESS`)

`platform` : Selected platform
`device_type` : Device category (e.g. `CL_DEVICE_TYPE_CPU`, `CL_DEVICE_TYPE_GPU`)
`num_entries` : Number of pre-allocated elements of type `cl_device_id` in the array `devices`
`devices` : Returns information about the devices (for each device one element in the array `devices`)
`num_devices` : Returns number of devices

Query Devices (cont.)

Double invocation of `clGetDeviceIDs(..)` necessary

- 1. invocation: `num_entries = 0, devices = NULL` → Query `num_devices`
- Allocate `num_devices` elements of type `cl_device_id` in the array `devices`
- 2. invocation: `num_entries = num_devices` → Query `devices`

Related functions

- `clGetDeviceInfo(..)`

Create Context

Precondition: Device exists

```
cl_context
clCreateContext( const cl_context_properties *properties,
                 cl_uint num_devices,
                 const cl_device_id *devices,
                 (voidCL_CALLBACK  *pfn_notify) (
                         const char *errinfo,
                         const void *private_info, size_t cb,
                         void *user_data
                 ),
                 void *user_data,
                 cl_int *errcode_ret );
```

Creation of a context

Return value : The created context

`properties` : Bit field for the definition of the desired properties of the context
`num_devices` : Number of devices for which the context shall be created
`devices` : Array with devices for which the context shall be created
`errcode_ret` : Returns the error code (ideally equal to `CL_SUCCESS`)

For more details → see OpenCL man pages

Create Queue

Precondition: Context and device exist

```
cl_command_queue
clCreateCommandQueue( cl_context context,
                      cl_device_id device,
                      cl_command_queue_properties properties,
                      cl_int *errcode_ret );
```

Creation of a queue

Return value : The created queue

`context` : Context within which the queue shall be created
`device` : Device for which the queue shall be created
`properties` : Bit field for the definition of the desired properties of the queue
`errcode_ret` : Returns the error code (ideally equal to `CL_SUCCESS`)

Hint

The default mode for queues is “in order execution” (other settings possible via parameter `properties`).

Create Program Object

Precondition: Context and source code exist

```
cl_program
clCreateProgramWithSource( cl_context context,
                           cl_uint count,
                           const char **strings,
                           const size_t *lengths,
                           cl_int *errcode_ret );
```

Creation of a program object

Return value : The created program object

`count` : Number of char buffers with source code (see `strings`)
`context` : Context within which the program object shall be created
`strings` : Array with pointers to the char buffers containing the source code
`length` : Array specifying the length of each char buffer (in bytes)
`errcode_ret` : Returns the error code (ideally equal to `CL_SUCCESS`)

Hint

Most often the char buffers have been read in before from text files with the OpenCL source code (file extension: `.cl`). For small applications, there is often only one char buffer which contains the complete source code.

Compile Program

Precondition: Program object and device(s) exist

```
cl_int
clBuildProgram( cl_program program,
                cl_uint num_devices,
                const cl_device_id *device_list,
                const char *options,
                void (CL_CALLBACK *pfn_notify)(
                        cl_program program, void *user_data
                ),
                void *user_data );
```

Compile the program for the listed devices

Return value : Error code (ideally equal to `CL_SUCCESS`)

`num_devices` : Number of devices for which the program shall be compiled
`device_list` : Array with devices for which the program shall be compiled
(these must belong to the same context as the program object!)
`options` : Char string with compiler options

Compile Program (cont.)

Hints

- For more details ➔ see OpenCL man pages
- Automagically, the right compiler implementation is used — the one from the OpenCL library which implements the platform for the devices which belong to the context of the program object.

Related functions

- `clGetProgramBuildInfo(..)`

➔ Query the build status and the compiler logs (with error messages, e.g. for syntax errors within the OpenCL device source code)

Create Kernel

Precondition: Program object with compiled code exists

```
cl_kernel clCreateKernel( cl_program program,
                          const char *kernel_name,
                          cl_int *errcode_ret );
```

Creation of a compute kernel

Return value : The created kernel

`program` : The program object which contains the compiled kernel code
`kernel_name` : Name of the kernel function (within the source code of the program object)
`errcode_ret` : Returns the error code (ideally equal to `CL_SUCCESS`)

Hint

The kernel is afterwards available for all devices which were contained in the `device_list` when calling `clBuildProgram(..)` before.

Create Memory Objects

Precondition: Context exists

Here: Creation of a buffer (alternatively: sub buffer, image)

```
cl_mem clCreateBuffer( cl_context context,
                       cl_mem_flags flags,
                       size_t size,
                       void *host_ptr,
                       cl_int *errcode_ret );
```

Creation of a buffer object

Return value : The created buffer

`context` : Context within which the buffer shall be created

`flags` : Bit field for the definition of the buffer properties and of the copy operations executed at creation

`size` : Length of the buffer (in bytes)

`host_ptr` : Pointer to the memory area in host memory which is used as source for copy operations or which is directly used for the buffer

`errcode_ret` : Returns the error code (ideally equal to `CL_SUCCESS`)

Explanation of the Parameter `flags` (Disjunct. within the Bit Field)

Flag	Meaning
`CL_MEM_READ_WRITE`	Memory object will be read and written by a kernel.
`CL_MEM_READ_ONLY`	Memory object will only be read by a kernel.
`CL_MEM_WRITE_ONLY`	Memory object will only be written by a kernel.
`CL_MEM_USE_HOST_PTR`	The buffer shall be located in host memory at address `host_ptr` (content may be cached in device memory). Not combinable with `CL_MEM_ALLOC_HOST_PTR` or `CL_MEM_COPY_HOST_PTR`.
`CL_MEM_ALLOC_HOST_PTR`	The buffer will be newly allocated in host memory (→ in some implementations page–locked memory!).
`CL_MEM_COPY_HOST_PTR`	The buffer will be initialized with the content of the memory region to which `host_ptr` points.

Set Kernel Arguments

Precondition: Kernel exists

```
cl_int clSetKernelArg( cl_kernel kernel,
                       cl_uint arg_index,
                       size_t arg_size,
                       const void *arg_value );
```

Set a single kernel argument

Return value : Error code (ideally equal to `CL_SUCCESS`)

`kernel` : The kernel for which the argument is set
`arg_index` : Index of the argument (starting with 0 for the first argument of the kernel function)
`arg_size` : Length of the value of the argument (in bytes)
`arg_value` : Pointer to the value of the argument

Hints

- If you want to pass a global memory buffer as kernel argument, you have to use the corresponding `cl_mem` object as value.
- In this case, `arg_size` has to be the size of the `cl_mem` object (**not** the length of the buffer)!

Execution Model (repeated)

Example: 2D–Arrangement of Work–Items

S_x, S_y : Number of work–items within a single work–group

s_x, s_y : Indices of a work–item within a work–group

w_x, w_y : Indices of a work–group within the NDRange

F_x, F_y : Global index offset

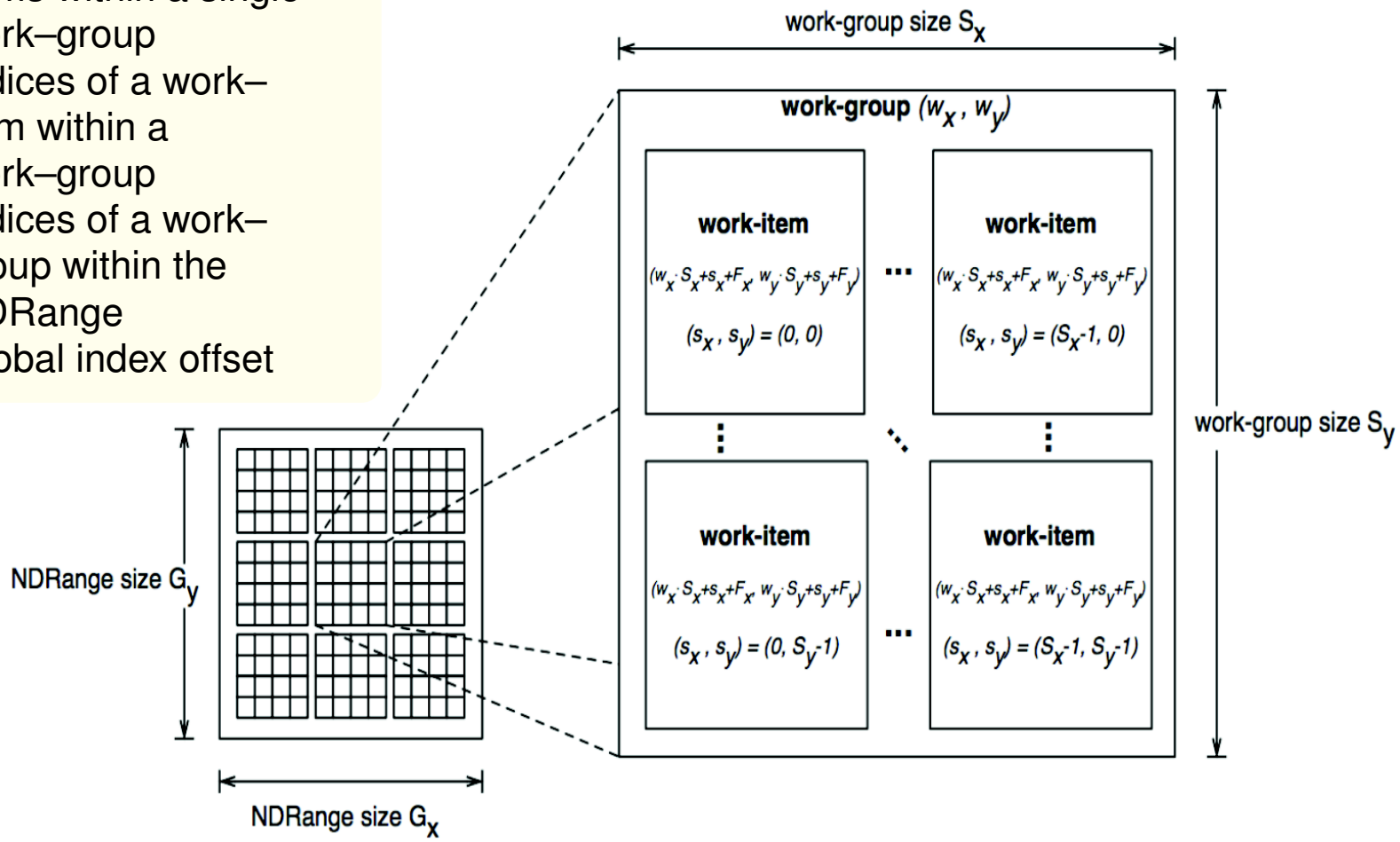


Fig.: „The OpenCL Specification 1.1“, Fig. 3.2

Kernel Execution

Precondition: Queue and kernel exist, kernel arguments already set

```
cl_int
clEnqueueNDRangeKernel( cl_command_queue command_queue,
                        cl_kernel kernel,
                        cl_uint work_dim,
                        const size_t *global_work_offset,
                        const size_t *global_work_size,
                        const size_t *local_work_size,
                        cl_uint num_events_in_wait_list,
                        const cl_event *event_wait_list,
                        cl_event *event );
```

Place a kernel for execution in a queue

Return value : Error code (ideally equal to `CL_SUCCESS`)

`command_queue` : Queue which shall be used for execution

`kernel` : The kernel to be executed

`work_dim` : Number of array dimensions (concerning the following three parameters)

`global_work_offset` : F_x (F_y, F_z) (see preceding slide)

`global_work_size` : G_x (G_y, G_z) (see preceding slide; overall number of work–items in each dimension across all work–groups!)

`local_work_size` : S_x (S_y, S_z) (see preceding slide; the ratios G_x/S_x, G_y/S_y, G_z/S_z need to be integer numbers!)

Kernel Execution (cont.)

Hints

- The size and structure of the NDRange are defined by the parameters `global_work_offset`, `global_work_size` and `local_work_size`.
- If `local_work_size` is set to `NULL`, the size of the work–groups will be automatically determined.
- For details on event handling ➔ see later slides

Transfer Data from Device to Host

Precondition: Queue exists

```
cl_int clEnqueueReadBuffer( cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_read,
                            size_t offset,
                            size_t cb,
                            void *ptr,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event );
```

Copy buffer content into host memory (e.g., buffer with results after kernel execution)

Return value : Error code (ideally equal to `CL_SUCCESS`)

`command_queue` : Queue which shall be used for execution
`buffer` : Buffer object which serves as **source** of the copy operation
`blocking_read` : If `true`, the function only returns after the copy operation has been finished (and therefore also all preceding commands in the queue if it operates in “in–order mode”)
`offset` : Read offset in the buffer (in bytes)
`cb` : Number of bytes to copy
`ptr` : Pointer to the **target region** in host memory (needs to be allocated in sufficient size before)

For details on event handling ➔ see later slides

Free OpenCL Resources
(Selection)

```
cl_int clReleaseContext( cl_context context );
cl_int clReleaseCommandQueue( cl_command_queue command_queue );
cl_int clReleaseProgram( cl_program program );
cl_int clReleaseKernel( cl_kernel kernel );
cl_int clReleaseMemObject( cl_mem memobj );
```

Release of different types of OpenCL objects

Return value : Error code (ideally equal to `CL_SUCCESS`)

Hint

In analogy to the release functions also retain functions exist for many types of OpenCL objects. The retain functions increase an object–internal counter, the release functions decrease it. Only after all retain calls were compensated by a release call, the next subsequent release call will ultimately free the resources of the object.

OpenCL for Compute Kernels

Basic Facts about “OpenCL C”

- Derived from ISO C99
- A few restrictions: No recursion, no function pointers, no functions from the C99 standard headers
- Preprocessing directives defined by C99 are supported (e.g., `#include`)
- Built–in data types:
  - Scalar and vector data types, pointers, images
- Mandatory built–in functions:
  - Work–item functions, math.h, reading and writing of images
  - Relational functions, geometric functions, synchronization functions
  - printf (v1.2 only)
- Optional built–in functions (called “extensions”)
  - Support for double precision, atomics to global and local memory

Qualifiers and Functions

- Function qualifiers:
  - `__kernel` qualifier declares a function as a kernel, i.e. makes it visible to host code so that it can be enqueued
- Address space qualifiers:
  - `__global`, `__local`, `__constant`, `__private`
  - Pointer kernel arguments must be declared with an address space qualifier (excl. `__private`)
- Work-item functions:
  - `get_work_dim()`, `get_global_id()`, `get_local_id()`, `get_group_id()`, etc.
- Synchronization functions:
  - Barriers — all work-items within a work-group must execute the barrier function before any work-item can continue:
    `barrier(cl_mem_fence_flags flags)`
  - Memory fences — provides ordering between memory operations:
    `mem_fence(cl_mem_fence_flags flags)`

Restrictions

- Recursion is not supported
- Pointers to functions are not allowed
- Pointers to pointers allowed within a kernel, but not as an argument to a kernel invocation
- Bit–fields are not supported
- Variable length arrays are not supported
- Structures and other data types have to be defined in both the host and device code (naturally, in exactly the same way; use common header files)
- Double types are optional in OpenCL v1.1, but the key word is reserved (note: Most implementations support double)

Exercise 1

Task and Hints

Task

- Implement the addition of three vectors instead of two!

Hints

- Copy project files from
  `train025@zam1069:OpenCL_Course/OpenCL_Basics/example`
- Use host code in `VectorAddition.C` as starting point
- Use device code in `vectoradd.cl` as starting point
- Adjust settings in `Makefile` to the computer system which you are using

Event Handling

Further Useful Functions...

...for Event Handling and Queues

```
cl_int clWaitForEvents ( cl_uint num_events,
                         const cl_event *event_list );
```

Wait for all events in `event_list`

Return value : Error code (ideally equal to `CL_SUCCESS`)

`num_events` : Number of elements in `event_list`
`event_list` : Array of events

```
cl_int clFlush( cl_command_queue command_queue );
```

Issues all previously queued OpenCL commands in `command_queue` to the device associated with `command_queue`

Return value : Error code (ideally equal to `CL_SUCCESS`)

```
cl_int clFinish( cl_command_queue command_queue );
```

Blocks until all previously queued OpenCL commands in `command_queue` are issued to the associated device and have completed. `clFinish` is also a synchronization point.

Return value : Error code (ideally equal to `CL_SUCCESS`)

Exercise 2

Task and Hints

Task

- Implement a second kernel for element–wise vector multiplication!
- Compute with both kernels (multiplication and pair–wise addition) the equation $e = a * b + c * d$ as element–wise vector operation!
- BONUS: Use an out–of–order queue instead of the default queue. . .
- . . . and ensure by using events that all commands are executed in the right order!

Hints

- Extend your code from exercise 1

Appendix: Notes on Nomenclature

Nomenclature

AMD vs. NVIDIA

AMD	NVIDIA
—	Texture Processing Cluster (TPC), Graphics Processing Cluster (GPC)
SIMD–Core *GCN–Arch.: Compute–Unit*	Streaming Multi–Processor (SM, SMX)
GCN–Arch.: SIMD	—
SIMD–Einheit („SIMD lane“)	Streaming Proc. (SP), CUDA Core
Wavefront	Warp
Local Data Share	Shared Memory
Global Data Share	—

Nomenclature

OpenCL vs. CUDA

OpenCL	CUDA
Work–Item	Thread
Work–Group	Block
NDRange (Workspace)	Grid
Local Memory	Shared Memory
Private Memory	Registers/Local Memory
Image	Texture
Queue	Stream
Event	Event