OpenCL Base Course

Ing. Marco Stefano Scroppo, PhD Student at University of Catania

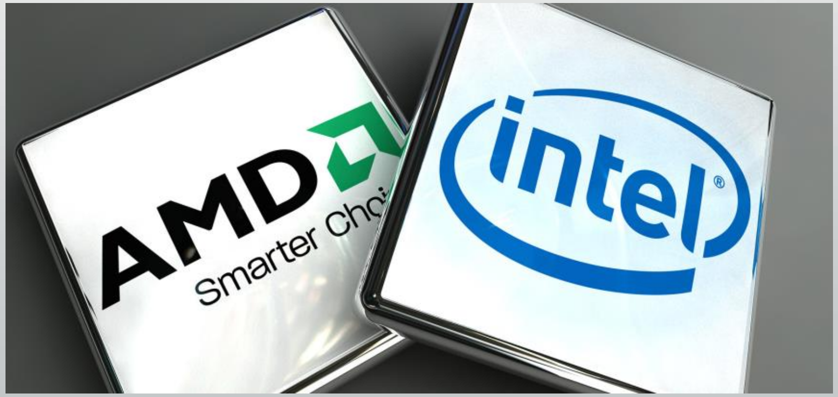
Course Overview

This OpenCL base course is structured as follows:

- Introduction to GPGPU programming, parallel programming and heterogeneous programming
- The OpenCL framework
- OpenCL programming
  - Introduction to OpenCL programming
  - Guidelines through examples
  - OpenCL programming - students hands-on

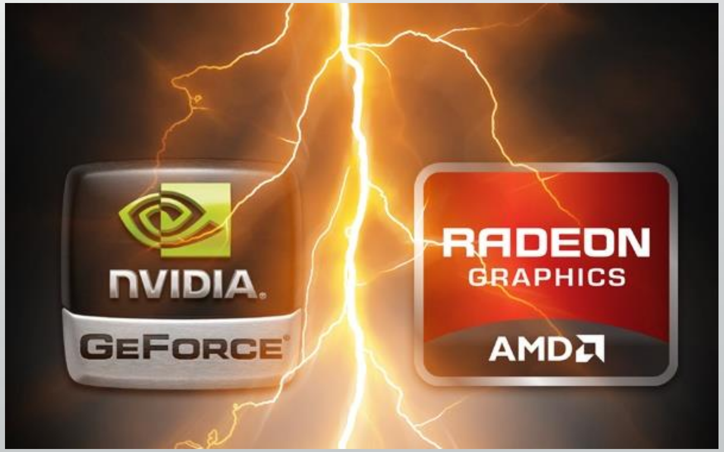
CPU

- Central Processing Unit (CPU) of a computer system must be able to perform a wide variety of tasks efficiently.
- The increase of CPU performance corresponds to the increase of the core clock frequency.
  - But now it has reached the limit and it is no longer possible due to power requirements.
- Today, processor cores are not getting any faster, but instead we are getting increasing numbers of cores per chip.

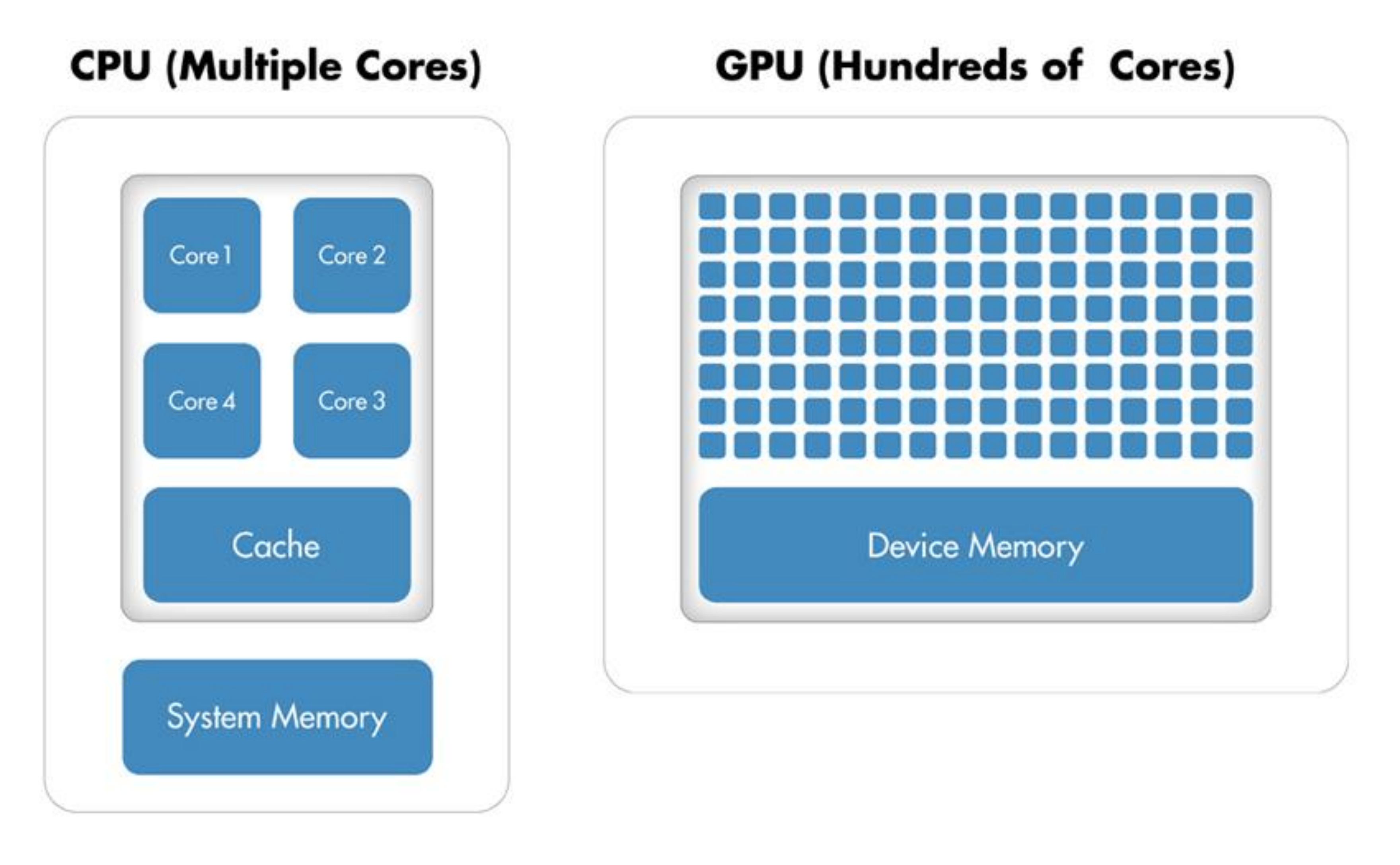


GPU

- A Graphics Processing Unit (GPU) is a kind of processor primarily used to manage and boost the performance of video and graphics
  - The main feature of a GPU is the presence of a high number (hundreds) of simplistic cores
- GPUs work in tandem with the CPU, and are responsible for generating the graphical output display (computing pixel values)
- Inherently parallel - each core computes a certain set of pixels
  - Architecture has evolved for this purpose



CPU vs GPU



CPU (Multiple Cores)
GPU (Hundreds of Cores)
Core 1
Core 2
Core 4
Core 3
Cache
System Memory
Device Memory

GPGPU

- GPGPU: General Purpose computation on Graphics Processing Units.
- Idea: using GPU for generic computations
- GPU acts as an “accelerator” to the CPU (Heterogeneous System)
  - Most lines of code are executed on the CPU
  - Key computational kernels are executed on the GPU
    - Taking advantage of the large number of cores and high graphics memory bandwidth
  - AIM: code performs better than use of CPU alone.
- Nvidia was the pioneer for GPGPU.
  - It created CUDA Language (based on C) and the guidelines to follow

Heterogeneous Computing

- Heterogeneous computing exploit the capabilities of different computing resources in a system like
  - CPU
  - GPU
  - Multicore Microprocessor
  - Digital Signal Processor
  - Reconfigurable Hardware (field-programmable gate arrays)

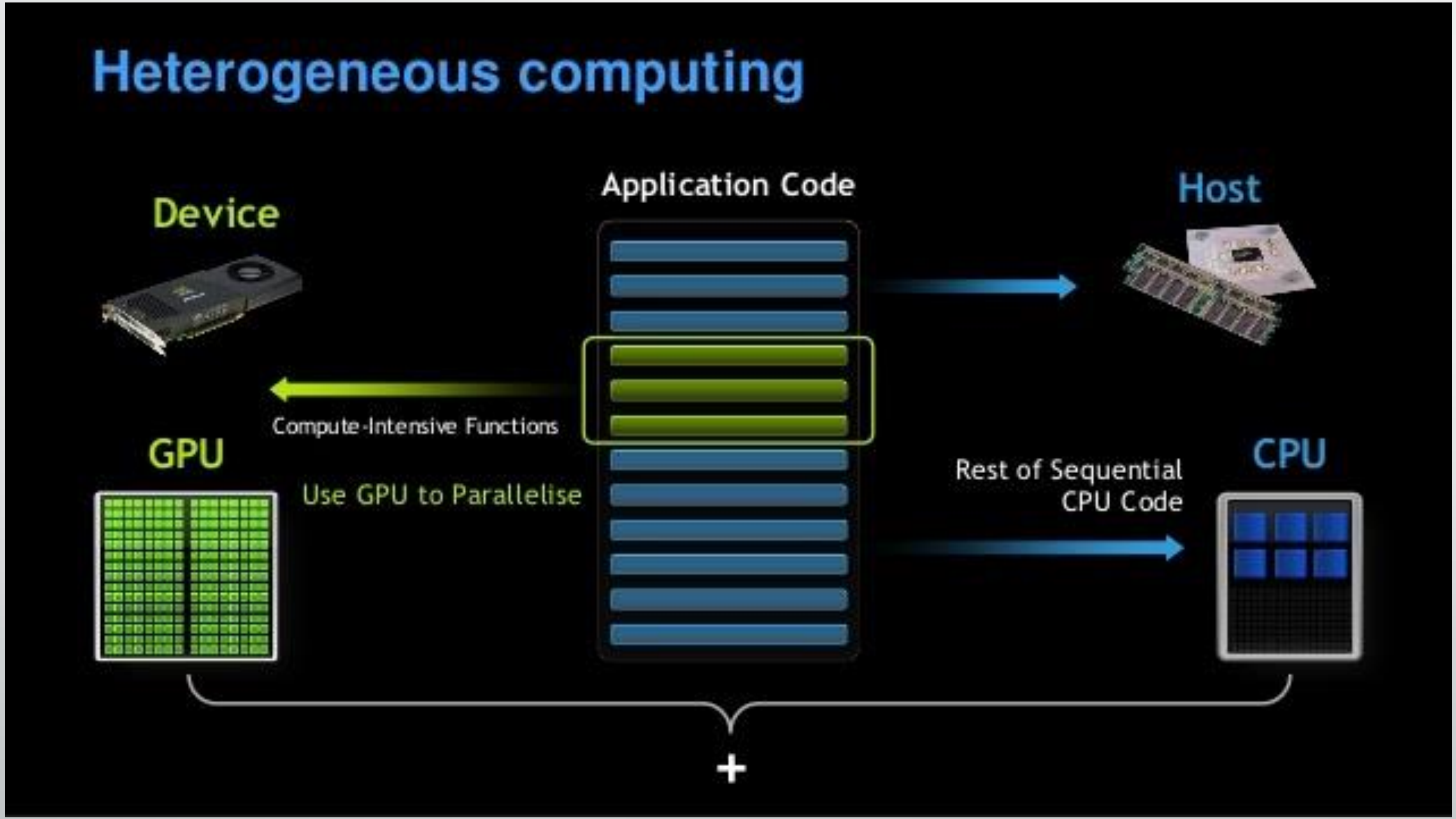
Heterogeneous Computing

- Heterogeneous applications commonly include a mix of workload behaviors:
  - control intensive (e.g. searching, sorting, and parsing)
  - data intensive (e.g. image processing, simulation and modeling, and data mining)
  - compute intensive (e.g. iterative methods, numerical methods, and financial modeling)
- Each of these workload classes executes most efficiently on a specific style of hardware architecture and no single device is best for running all classes of workloads. For example:
  - Control-intensive applications tend to run faster on superscalar CPUs
    - They use branch prediction mechanism that are very powerful on this hardware
  - Data-intensive applications tend to run faster on vector architectures
    - In this kind of application the same operation is applied to multiple data items, and on vector architecturs multiple operations can be executed in parallel

Heterogeneous Computing

- Heterogeneous computing is usually used to obtain a high level of parallelization
  - This increase performance in applications where there are several workloads
- The use of a graphics processing unit (GPU) together with a CPU to accelerate scientific, analytics, engineering, consumer, and enterprise applications is a simple and common scenario of the heterogeneous programming

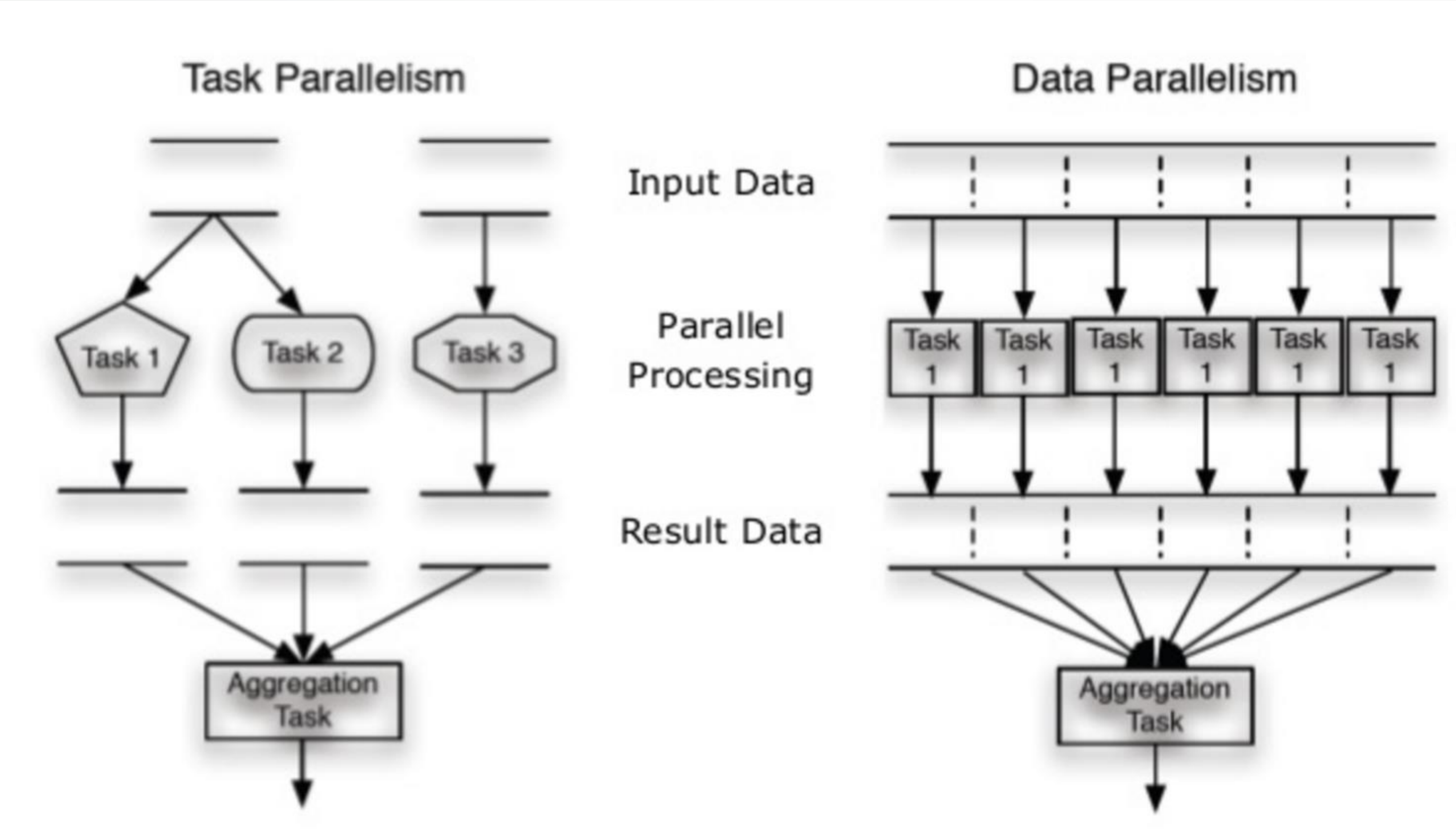
Heterogeneous Computing



Parallel Programming

- Most applications are first programmed to run on a single processor
  - But sometimes applications can be parallelized
- The parallel programming is the ability to use multiple computing resources to speed up the computation
- Two kinds of parallelism:
  - Task-based: each unit carries out a different job.
  - Data-based: all units do the same work on different subsets of the data

Parallel Programming



Parallel Programming

- A problem can be parallelized only if it can be divided into independent subproblems
- If the problem can be divided, it's possible to use a decomposition method
- Two main decomposition methods:
  - Divide-and-conquer
    - iteratively break a problem into smaller subproblems until the subproblems fit well on the computational resources provided
  - Scatter-gather
    - send a subset of the input data to each parallel resource, and then collect the results of the computation and combine them into a result data set

Parallel Programming

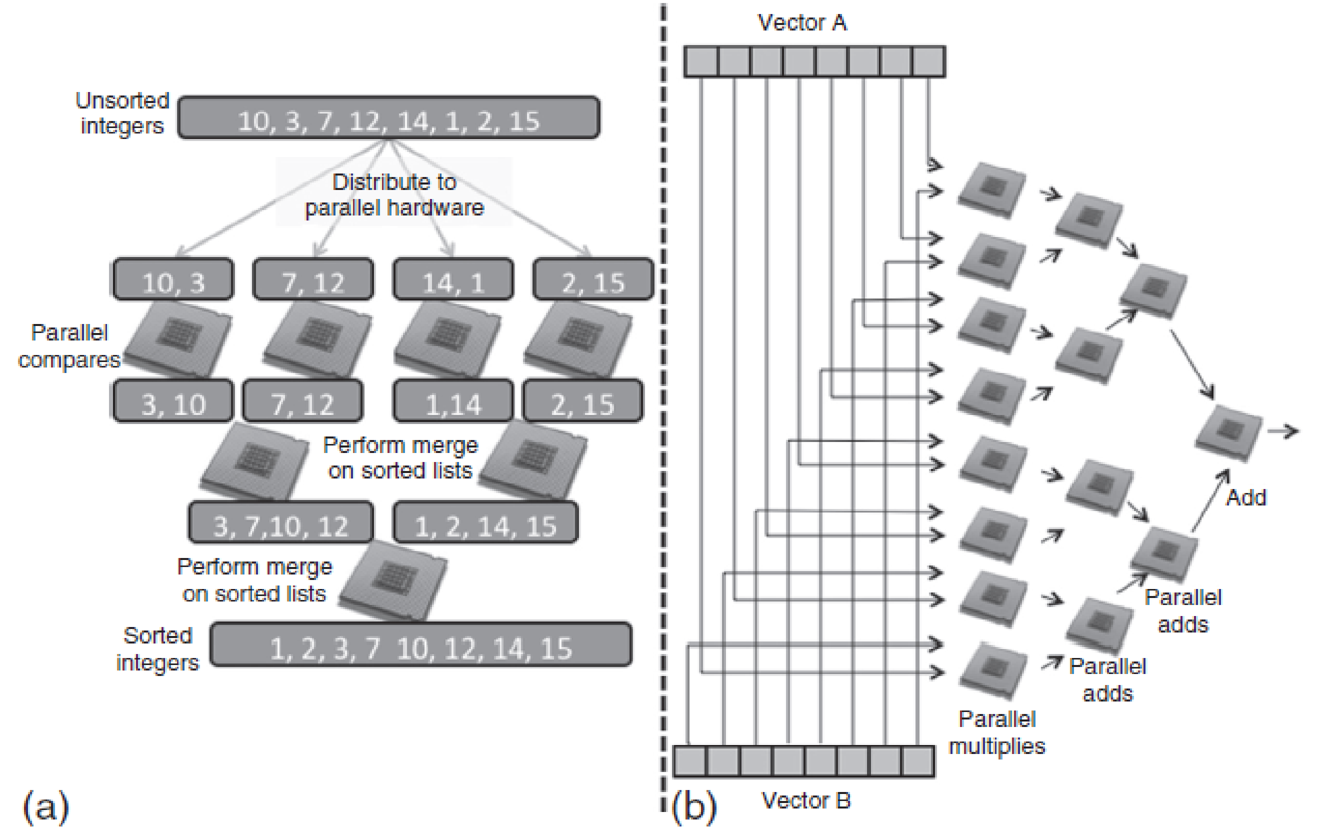


FIGURE 1.1

(a) Simple sorting: a divide-and-conquer implementation, breaking the list into shorter lists, sorting them, and then merging the shorter sorted lists. (b) Vector-scalar multiply: scattering the multiplies and then gathering the results to be summed up in a series of steps.

Parallelism Sample

- Classic Sample: multiplication of the elements of two arrays A and B (each with N elements) storing the result of each multiply in the corresponding element of array C
- The standard way to develop this sample is implementing a sequential solution

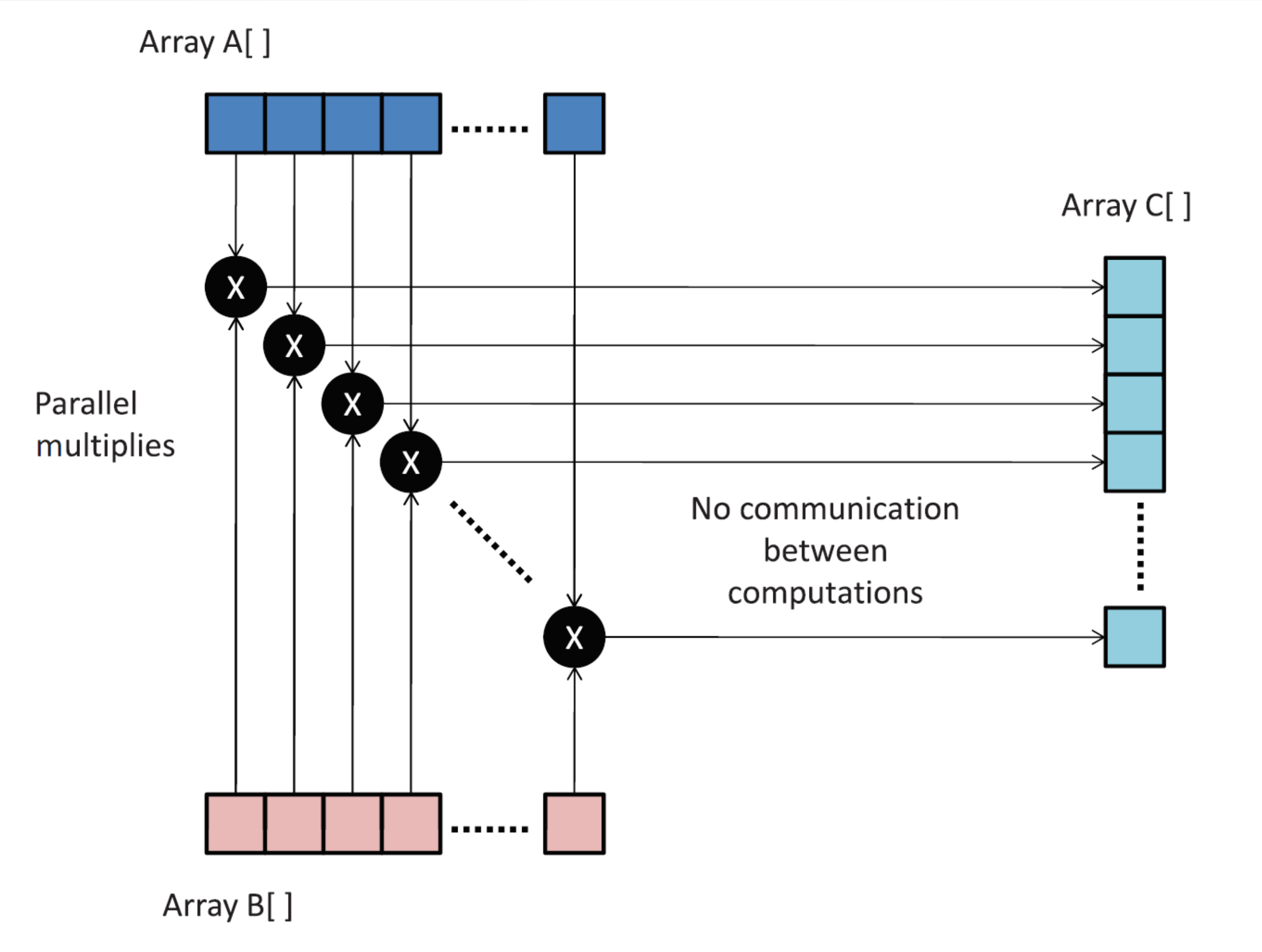
```
1   for (i=0; i<N; i++)
2       C[i] = A[i] * B[i];
```

- Problem: this solution execute N-times line 2 (one for each element in the array) without parallelism

Parallelism Sample

- **Question**: Is the sample parrallelizable? And why?
- **Answer**: Yes!!!!! The sample is parallelizable because the multiplication of each element in A and B is indipendent of every other element.
    - It's possible to create indipendent subproblems.
- **Solution**: Generate a separate execution instance to perform the computation of each element of C. This code possesses significant data-level parallelism because it's possible to perform the same operation in parallel to all the elements of A and B to produce C.

Parallelism Sample – Parallel Solution



Heterogeneous Computing - Problem

- For a class of algorithms developers write code in C or C++ and run it on a CPU.
- For another class of algorithms developers often write code in CUDA and use a GPU
- Two related approaches, but each worked on only one kind of processor and developers has to specialize in one and ignore the other.

how do you program such machines?

OpenCL

- The solution is **Open Computing Language** or **OpenCL**, a programming language developed specifically to support heterogeneous computing environments.

OpenCL - Introduction

- OpenCL is managed by the nonprofit technology consortium Khronos Group (Apple, IBM, NVIDIA, AMD, Intel, ARM, etc).
- The aim of OpenCL is enable the development of applications that can be executed across a range of devices made by different vendors.
  - Using the core language and correctly following the specification, any program designed for one vendor can execute on another vendor's hardware.

OpenCL - Introduction

- The first version of OpenCL, version 1.0, was released in 2008, and appeared in Apple's Mac OSX Snow Leopard.
  - AMD announced support for OpenCL in the same timeframe, and in 2009 IBM announced support for OpenCL in its XL compilers for the Power architecture.
- In 2010, the Khronos Group released version 1.1 of the OpenCL specification
- In 2011 released version 1.2
- In 2013 released version 2.0 (actual version).

OpenCL - Introduction

- OpenCL supports multiple levels of parallelism.
- It efficiently maps to
  - homogeneous or heterogeneous systems.
  - single- or multiple-device systems consisting of CPUs, GPUs, and other types of devices limited only by the imagination of vendors.
- OpenCL code is written in OpenCL C, a restricted version of the C99 language with extensions appropriate for executing data-parallel code on a variety of heterogeneous devices.

OpenCL or OpenGL

- OpenCL is similar to OpenGL but THEY ARE NOT THE SAME!!!!!!!
- OpenCL is specifically crafted to increase computing efficiency across platforms and it is typically used for image processing algorithms, physical simulations. It returns numerical results (**NO IMAGE RESULTS**).
- OpenGL is a graphical API that allows you to send rendering commands to the GPU. Typically, the goal is to show the rendering on screen.

OpenCL - Specification

The OpenCL specification is defined in four parts, which it refers to as *models*.

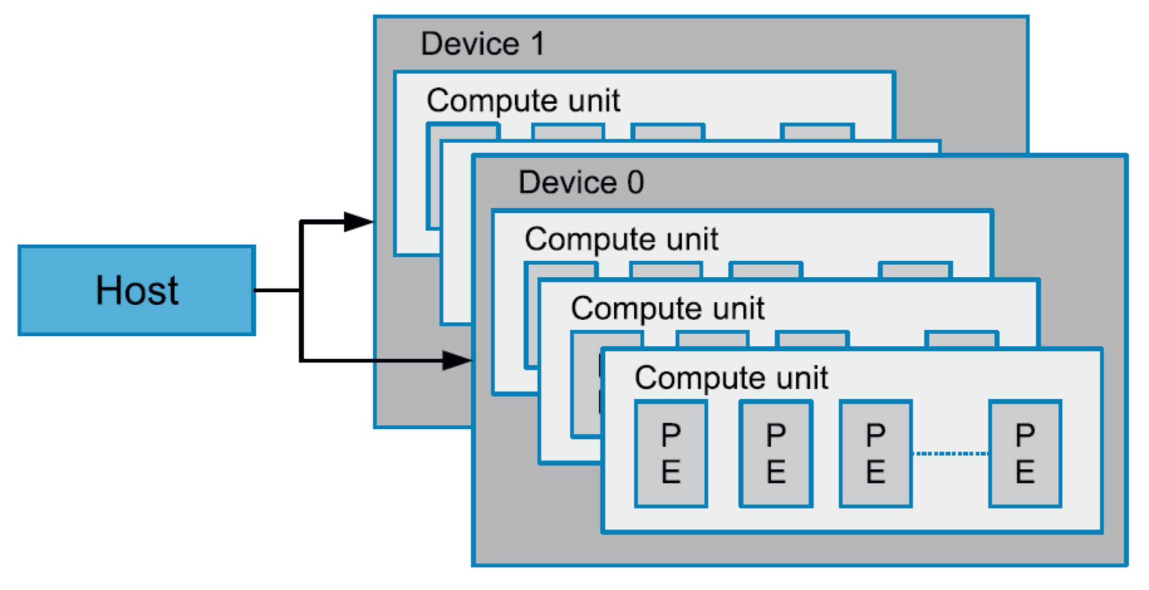
1. **Platform model**: Specifies that there is one host processor coordinating execution, and one or more device processors whose job it is to execute OpenCL C kernels. It also defines an abstract hardware model for devices.
2. **Execution model**: Defines how the OpenCL environment is configured by the host, and how the host may direct the devices to perform work. This includes defining an environment for execution on the host, mechanisms for host-device interaction, and a concurrency model used when configuring kernels. The concurrency model defines how an algorithm is decomposed into OpenCL work-items and work-groups.

OpenCL - Specification

3. **Kernel programming model**: Defines how the concurrency model is mapped to physical hardware.
4. **Memory model**: Defines memory object types, and the abstract memory hierarchy that kernels use regardless of the actual underlying memory architecture. It also contains requirements for memory ordering and optional shared virtual memory between the host and devices.

OpenCL – Platform Model

- An OpenCL platform consists of a host connected to one or more OpenCL devices.
- A device is divided into one or more compute units (functionally independent), which are further divided into one or more processing elements.
- A system could have multiple platform present at the same time: for example, an AMD platform and an Intel platform present on the same machine.



OpenCL – Platform Model

The AMD Radeon R9 290X graphics card (device) comprises 44 vector processors (compute units). Each compute unit has four 16-lane SIMD (Single Instrunction Multiple Data) engines, for a total of 64 lanes (processing elements). Each SIMD lane on the Radeon R9 290X executes a scalar instruction.

This allows the GPU device to execute a total of $44 \times 16 \times 4 = 2816$ instructions at a time.

OpenCL – Platform Model API

OpenCL offers two API function for discovering platforms and devices:

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

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

Let's analyze their behavior in a example:
clInfoProgram1.c

Instructions for connecting to Cometa GPU Server

You must use a SSH client to connect to our server. In Linux or MacOS it is installed by default. From a Windows system you have to download and install an SSH client (putty or openssh).

Step for connection:

1. ssh –l guest unict-diit-ui-01.ct.pi2s2.it
   - Enter ‘cometaGuest’ as password
2. ssh –l username cometa-gpu-01.consorzio-cometa.it
   - ‘username’ is the login name you should have received from Cometa
   - Enter the password you received

OpenCL – Platform Model API

Remember:

- 3 step to get the platforms/devices
  - STEP 1: discovery quantity of platforms/devices
  - STEP 2: allocation of enough space
  - STEP 3: retrieval of the desired number of platforms/devices
- You can choose what device retrieve with *device_type* argument:
  - *CL_DEVICE_TYPE_CPU*
  - *CL_DEVICE_TYPE_GPU*
  - *CL_DEVICE_TYPE_ALL*

OpenCL – Execution and Programming Model

The Execution model define two main components:

- **Host program**: written in C or C++, it runs on the OpenCL host. The host program creates and queries the platform and the device attributes, defines a context for the kernels, builds the kernels, and manages the execution of these kernels.
- **Kernels**: written in OpenCL C, they are the basic units of executable code that run on the OpenCL device. Each instance of a OpenCL kernel is executed by a Compute Units.

OpenCL – Execution and Programming Model

On submission of the kernel by the host to the device, an N dimensional index space is defined (N = 1 2 or 3).

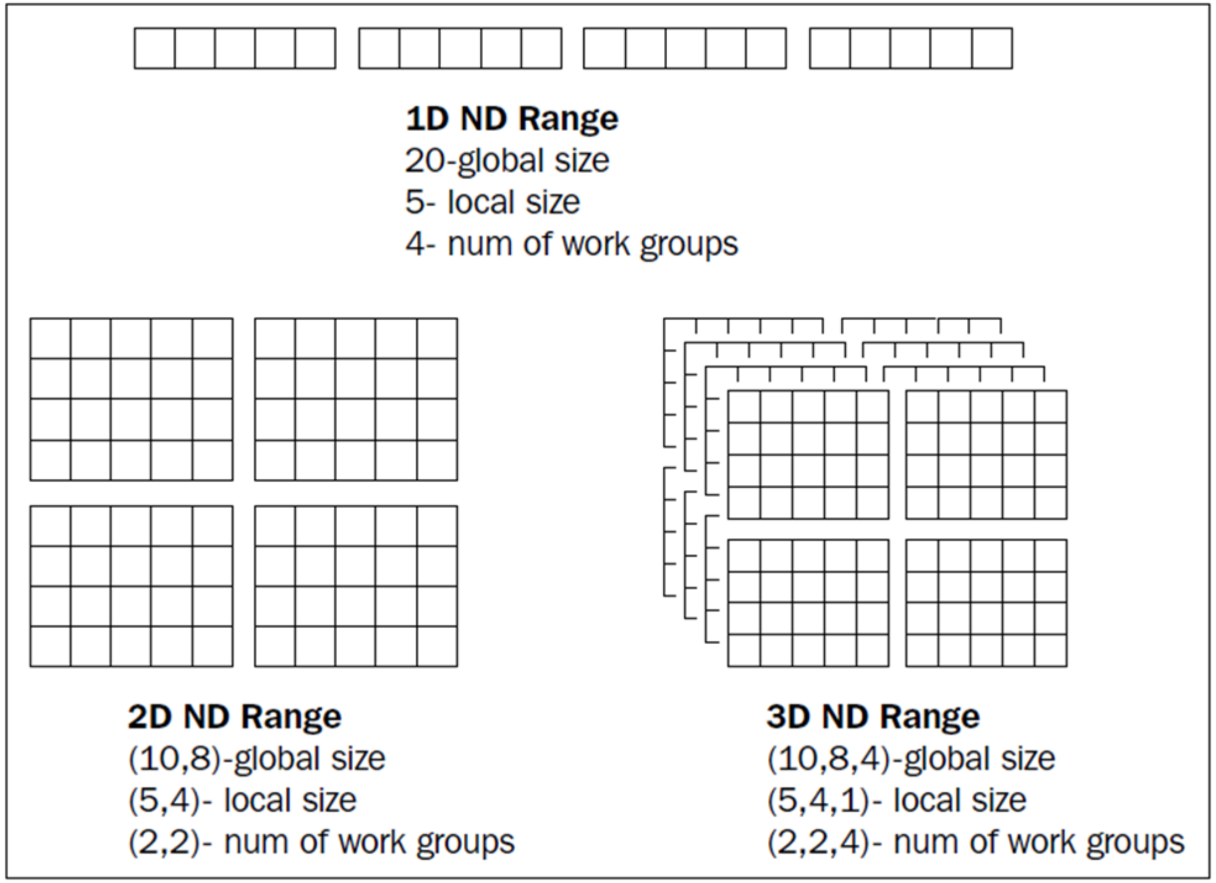
<u>The number of kernel instances is equal to the size of the index space</u>

<u>Each kernel instance is created at each of the coordinates of this index space.</u>

This instance is called as the "**work item**" and the index space is called as the **NDRange.** <u>The work-items are performed by the compute units.</u>

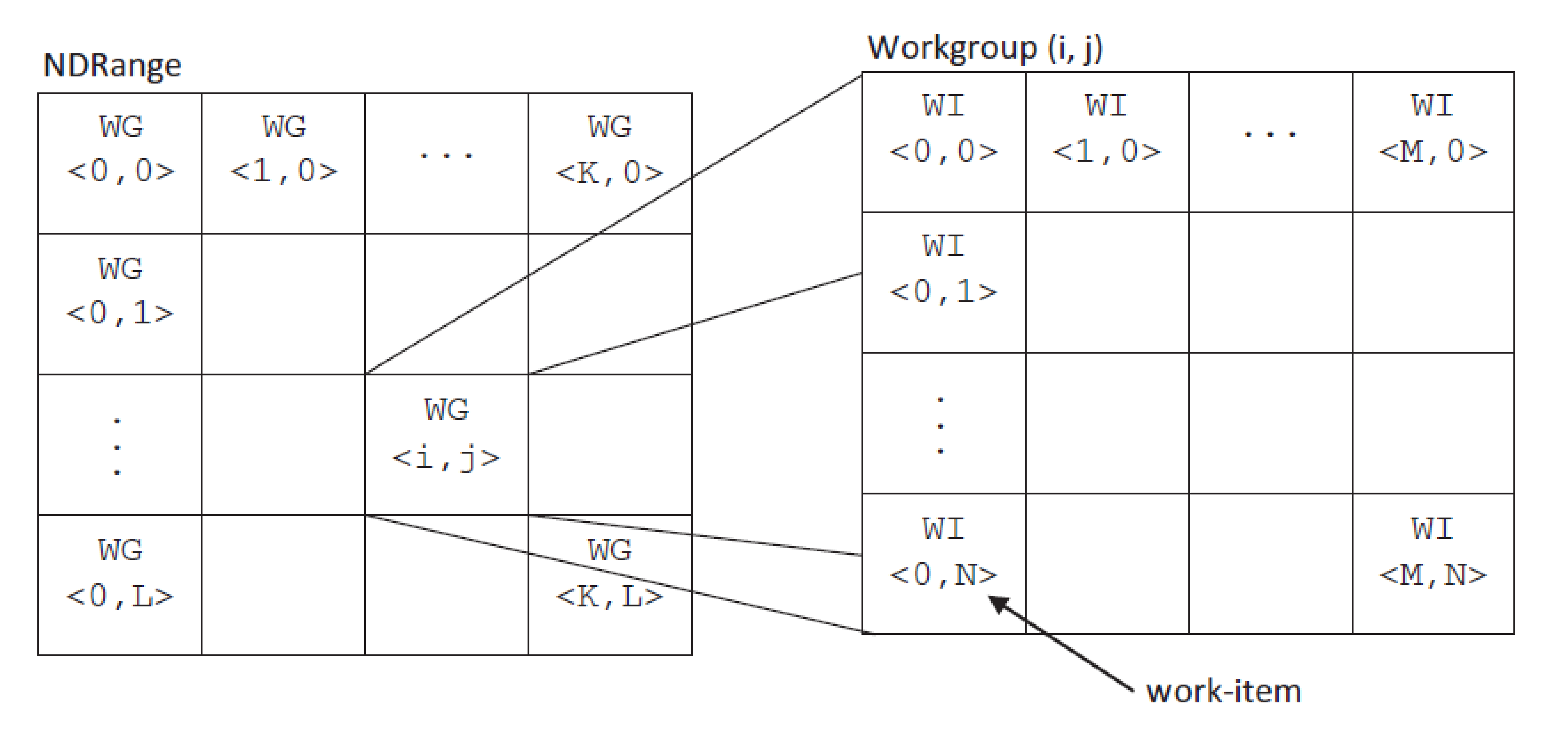
Work-items can be divided into smaller equally sized "**work-groups**"

OpenCL – Execution and Programming Model



OpenCL NDRange

OpenCL – Execution and Programming Model



OpenCL – Execution and Programming Model

- So for each work-item we can define two types of identifier:
  - **global-id**: A unique global ID given to each work item in the global NDRange
  - **local-id**: A unique local ID given to each work item within a work group

The ID is fundamental for the execution of the kernels in OpenCL

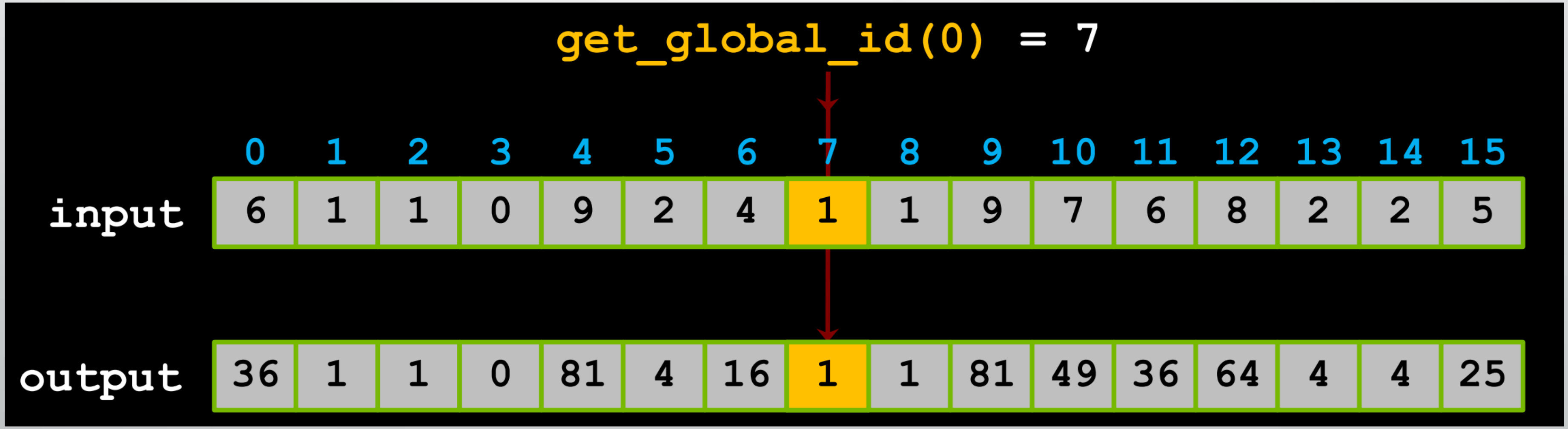
Scalar C Function	Data-Parallel Function

`void square(int n, const float *a, float *result)`
`{`
`  int i;`
`  for (i=0; i<n; i++)`
`    result[i] = a[i]*a[i];`
`}`
	`kernel void dp_square`
	`   (global const float *a, global float *result)`
	`{`
	`  int id= get_global_id(0);`
	`  result[id] = a[id]*a[id];`
	`}`
	`// dp_square execute over "n" work-items`

OpenCL – Execution and Programming Model

Data-Parallel Function

```
kernel void dp_square
    (global const float *a, global float *result)
{
  int id= get_global_id(0);
  result[id] = a[id]*a[id];
}
// dp_square execute over "n" work-items
```



OpenCL – Execution and Programming Model

In order for the host to request that a kernel is executed on a device, a **context** must be configured. It enables the host to pass commands and data to the device.

The API function to create a context is *clCreateContext()*.

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

                            ),
                            void *user_data,
                            cl_int *errcode_ret)
```

OpenCL – Execution and Programming Model

The execution model specifies that devices perform tasks based on *commands* which are sent from the host to the device.

A **command-queue** is the communication mechanism that the host uses to request action by a device. Once the host has decided which devices to work with and a context has been created, one command-queue needs to be created per device. The API function *clCreateCommandQueue()* (deprecated in OpenCL 2.0 and substituted by *clCreateCommandQueueWithProprierties*) is used to create a command-queue.

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

OpenCL – Execution and Programming Model

Any API call that submits a command to a command-queue will begin with ***<u>clEnqueue</u>*** and require a command-queue as a parameter.

- For example, the *clEnqueueReadBuffer*() call requests that the device send data to the host, and *clEnqueueNDRangeKernel*() requests that a kernel is executed on the device.

In addition to API calls that submit commands to command-queues, OpenCL includes barrier operations that can be used to synchronize execution of command-queues. The API calls *clFlush*() and *clFinish*() are barrier operations for a command-queue.

OpenCL – Execution and Programming Model

The command put in a queue are handled through the use of **events**. Each command of *clEnqueue* type has three parameters in common:

- a pointer to a list of events that specify dependencies for the current command called *wait list.* It is used to specify dependencies for a command
- the number of events in the wait list
- a pointer to an event that will represent the execution of the current command

```
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
```

OpenCL – Execution and Programming Model

The OpenCL API also includes the function *clWaitForEvents()*, which causes the host to wait for all events specified in the wait list to complete execution.

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

OpenCL – Execution and Programming Model

OpenCL source code **<u>is compiled at runtime</u>** through a series of API calls.

The process of creating a kernel from source code is as follows:

1. The OpenCL C source code is stored in a character array. If the source code is stored in a file on a disk, it must be read into memory and stored as a character array.
   - Each kernel in a program source string or file is identified by a *__kernel* qualifier
2. The source code is turned into a program object, *cl_program*, by calling *clCreateProgramWithSource()*.
   - It's possible to create a program from binary source with *clCreateProgramWithBinary()*
3. The program object is then compiled, for one or more OpenCL devices, with *clBuildProgram()*. If there are compile errors, they will be reported here.
4. A kernel object, *cl_kernel*, is then created by calling *clCreateKernel* and specifying the program object and kernel name.

The final step of obtaining a *cl_kernel* object is similar to obtaining an exported function from a dynamic library.

OpenCL – Execution and Programming Model

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

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

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

OpenCL – Execution and Programming Model

Unlike invoking functions in C programs, we cannot simply call a kernel with a list of arguments. Executing a kernel requires dispatching it through an enqueue function. Owing to the syntax of C and the fact that kernel arguments are persistent (and hence we need not repeatedly set them to construct the argument list for such a dispatch), we must specify each kernel argument individually using *clSetKernelArg().*

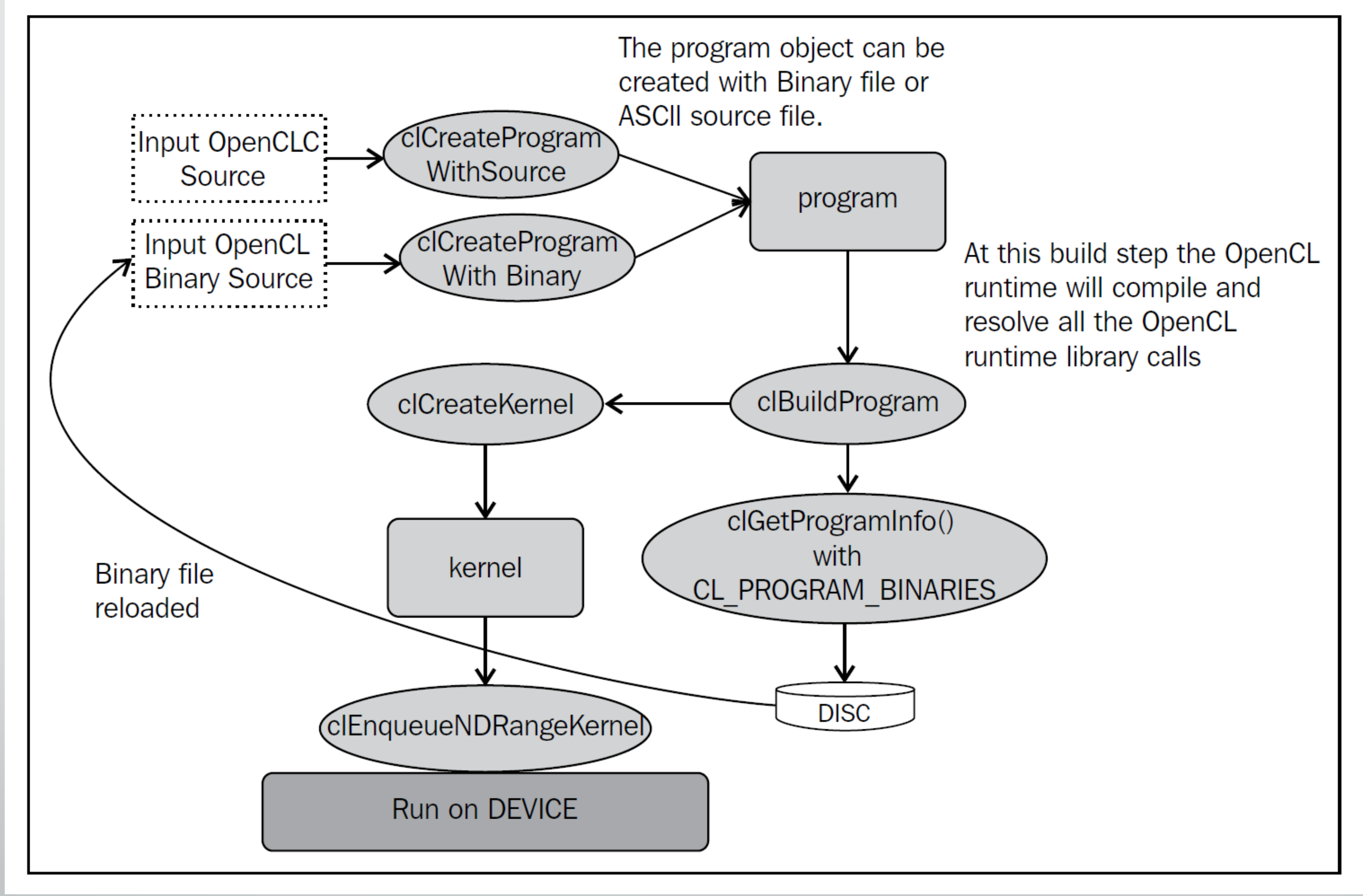
```
cl_int clSetKernelArg ( cl_kernel kernel,
                        cl_uint arg_index,
                        size_t arg_size,
                        const void *arg_value)
```

OpenCL – Execution and Programming Model

Enqueuing a command to a device to begin kernel execution is done with a call to *clEnqueueNDRangeKernel()*.

```
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)
```

OpenCL – Execution and Programming Model



OpenCL – Memory Model

To support code portability, OpenCL's approach is to define an abstract memory model that programmers can target when writing code and vendors can map to their actual memory hardware.

OpenCL defines three types of memory objects: *buffers*, *images* and *pipes*.

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

OpenCL – Memory Model

```
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)
```

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

OpenCL – Memory Model

OpenCL classifies memory as either *host memory* or *device memory*.

OpenCL divides device memory into four named *memory regions.*

These memory regions are relevant within OpenCL kernels.

- *Global Memory:* visible to all work-items (similarly to the main memory on a CPU-based host system).
- *Costant Memory*: specifically designed for data where each element is accessed simultaneously by all work-item. It is modelled as a part of Global Memory
- *Local Memory:* memory that is shared between work-items within a work-group.
- *Private Memory:* memory that is unique to an individual work-item.

OpenCL – Memory Model

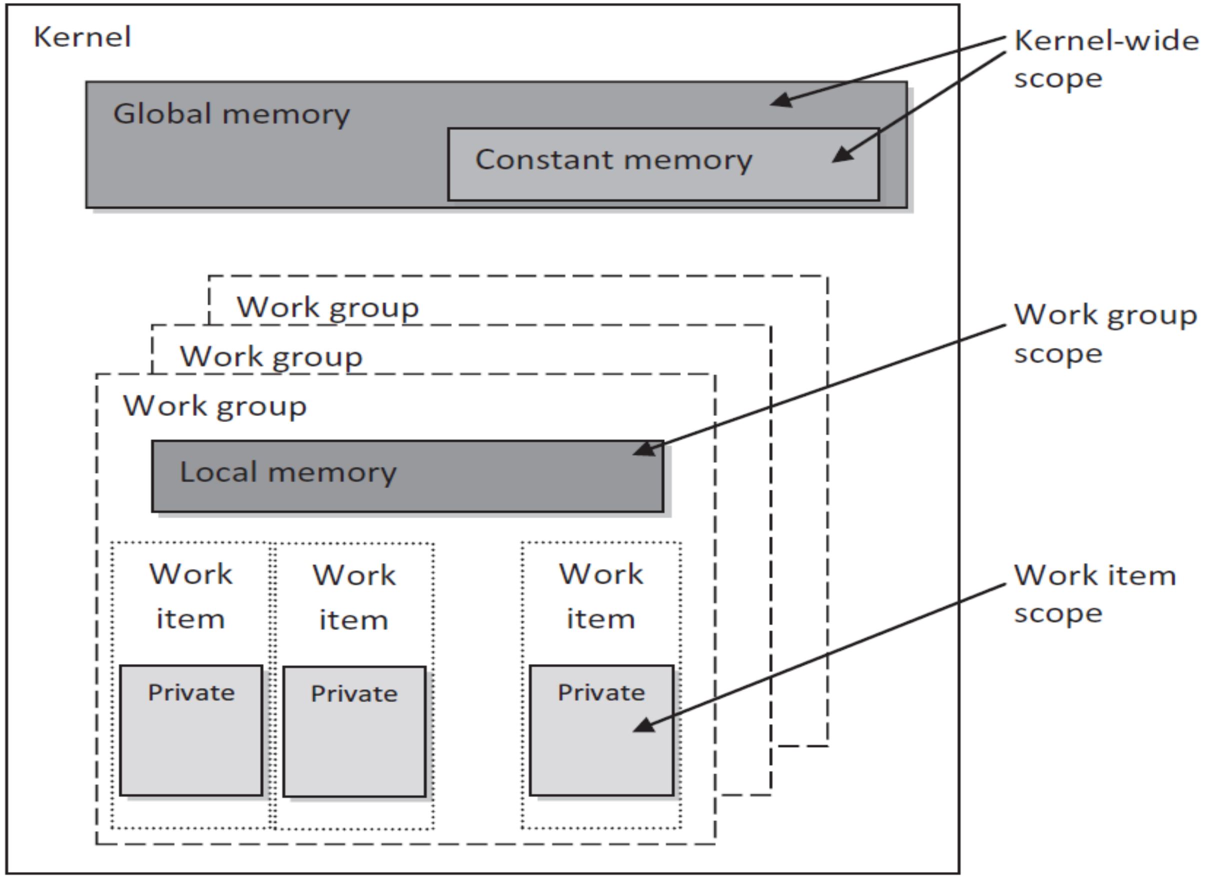


FIGURE 3.6

Memory regions and their scope in the OpenCL memory model.

OpenCL – Memory Model

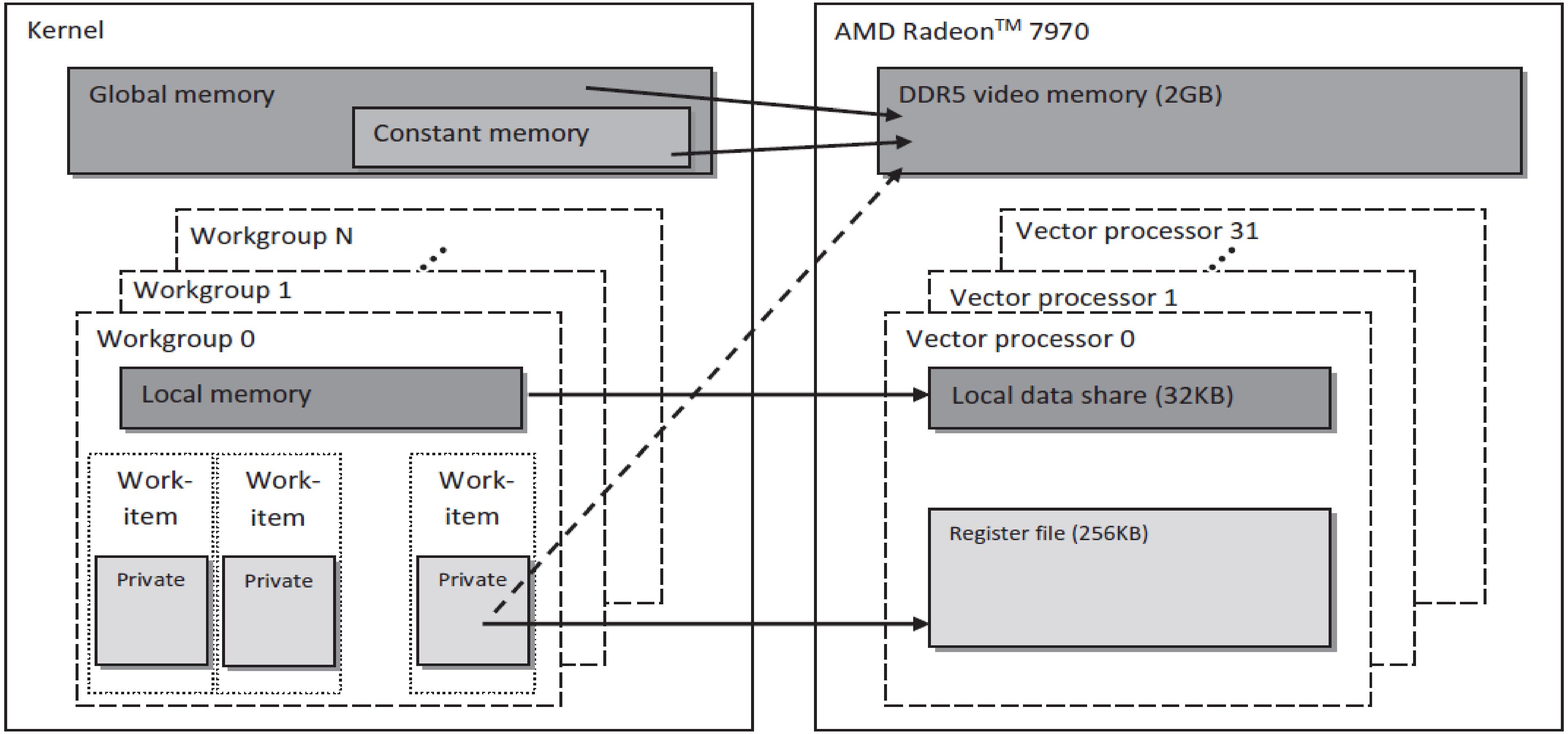


FIGURE 3.7

Mapping the OpenCL memory model to an AMD Radeon HD 7970 GPU.

THANK YOU FOR YOUR ATTENTION