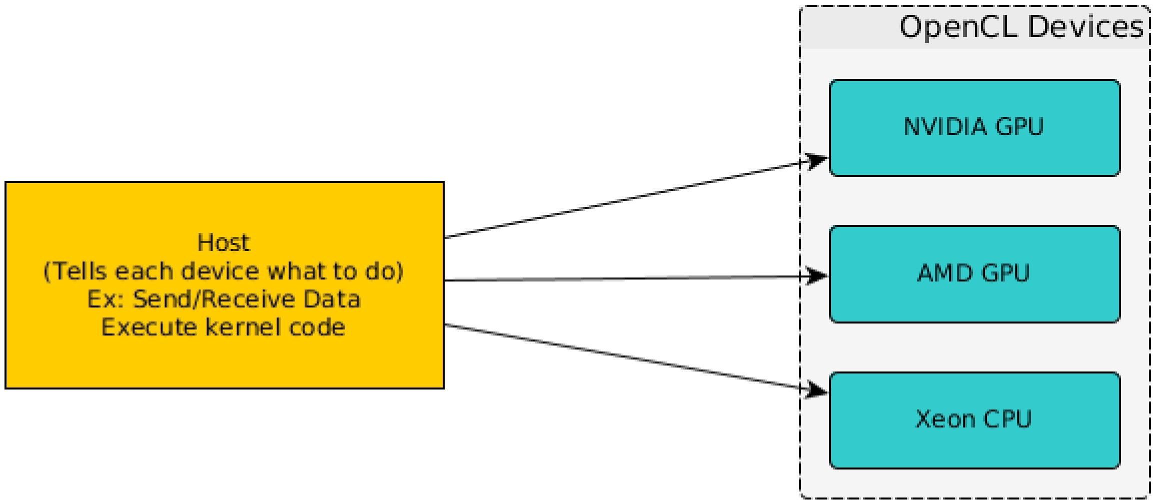
Introduction to OpenCL

Open Computing Language is a framework for writing programs that execute across heterogeneous platforms. They consist for example of CPUs GPUs DSPs and FPGAs. OpenCL specifies a programming language (based on C99) for programming these devices and application programming interfaces (APIs) to control the platform and execute programs on the compute devices. OpenCL provides a standard interface for parallel computing using task-based and data-based parallelism.

First thing to notice

While OpenCL can natively talk to a large range of devices, that doesn't mean that your code will run optimally on all of them without any effort on your part. In fact, there's no guarantee it will even run at all, given that different CL devices have very different feature sets. If you stick to the OpenCL spec and avoid vendor-specific extensions, your code should be portable, if not tuned for speed.

Actors on OpenCL system



Use cases

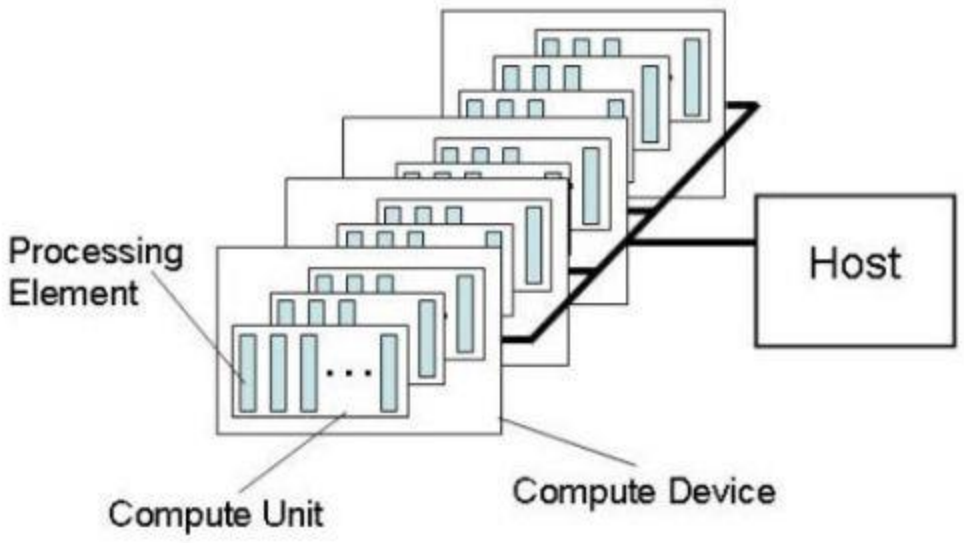
This are the normal cases where you should use GPUs for computation.

- Fast Permutation: Devices moves memory faster than Host
- Data Translation: Change from one format to another
- Numerical Acceleration: Devices calculate faster than Host big chunks of data

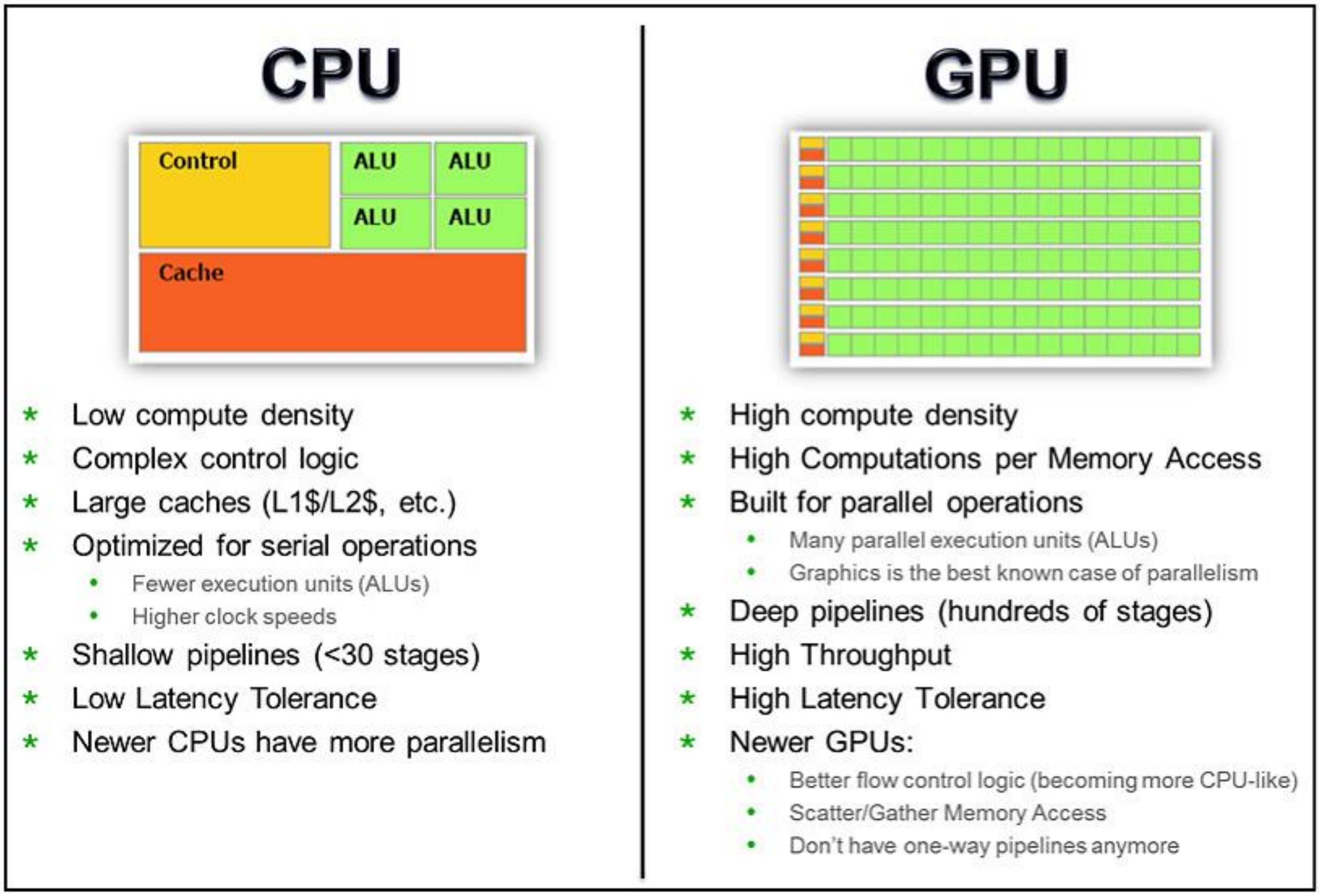
Heterogeneous systems

It's a system composed of multiple computing systems. For example a desktop system with a Multicore CPU and GPU.

Here are the main components of the system:



Host: Your desktop system Compute Device: CPU, GPU, FPGA, DSP. Compute Unit: Number of cores Processing Elements: ALUs on each core.
You don't need to think too much on how the OpenCL device model fit on a specific hardware, this is the responsibility of the hardware vendor. Don't think that Processing Element is a "Processor" or CPU Core.



CPU

* Low compute density
* Complex control logic
* Large caches (L1$/L2$, etc.)
* Optimized for serial operations
  - Fewer execution units (ALUs)
  - Higher clock speeds
* Shallow pipelines (<30 stages)
* Low Latency Tolerance
* Newer CPUs have more parallelism

GPU

* High compute density
* High Computations per Memory Access
* Built for parallel operations
  - Many parallel execution units (ALUs)
  - Graphics is the best known case of parallelism
* Deep pipelines (hundreds of stages)
* High Throughput
* High Latency Tolerance
* Newer GPUs:
  - Better flow control logic (becoming more CPU-like)
  - Scatter/Gather Memory Access
  - Don't have one-way pipelines anymore

OpenCL Models

First to understand OpenCL we need to understand the following models.

- Device Model: How the device look inside.
- Execution Model: How work get done on devices.
- Memory Model: How devices and host see data.
- Host API: How the host control the devices.

OpenCL components

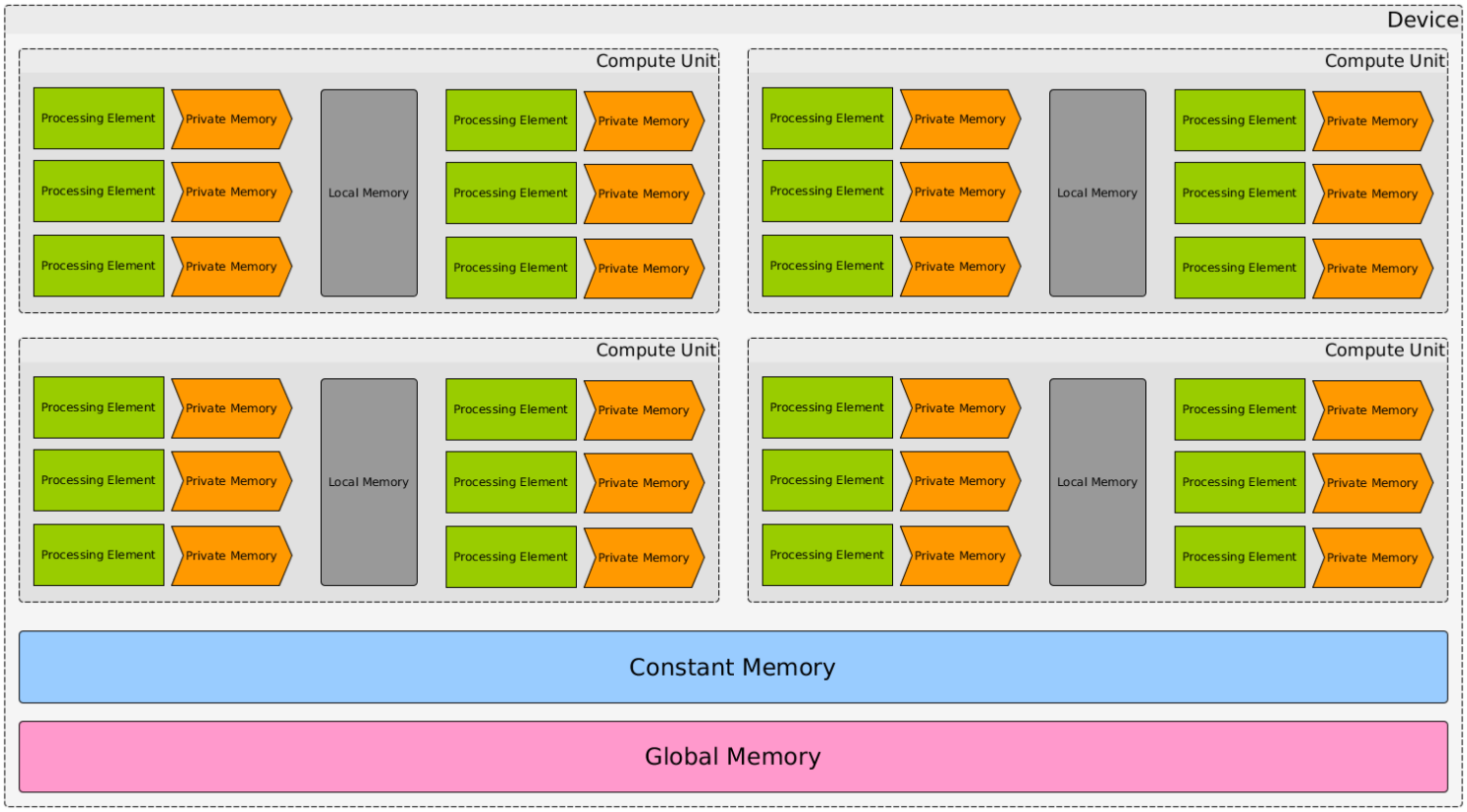
1.

C Host API: C API used to control the devices. (Ex: Memory transfer, kernel compilation)

2.

OpenCL C: Used on the device (Kernel Language)

Device Model

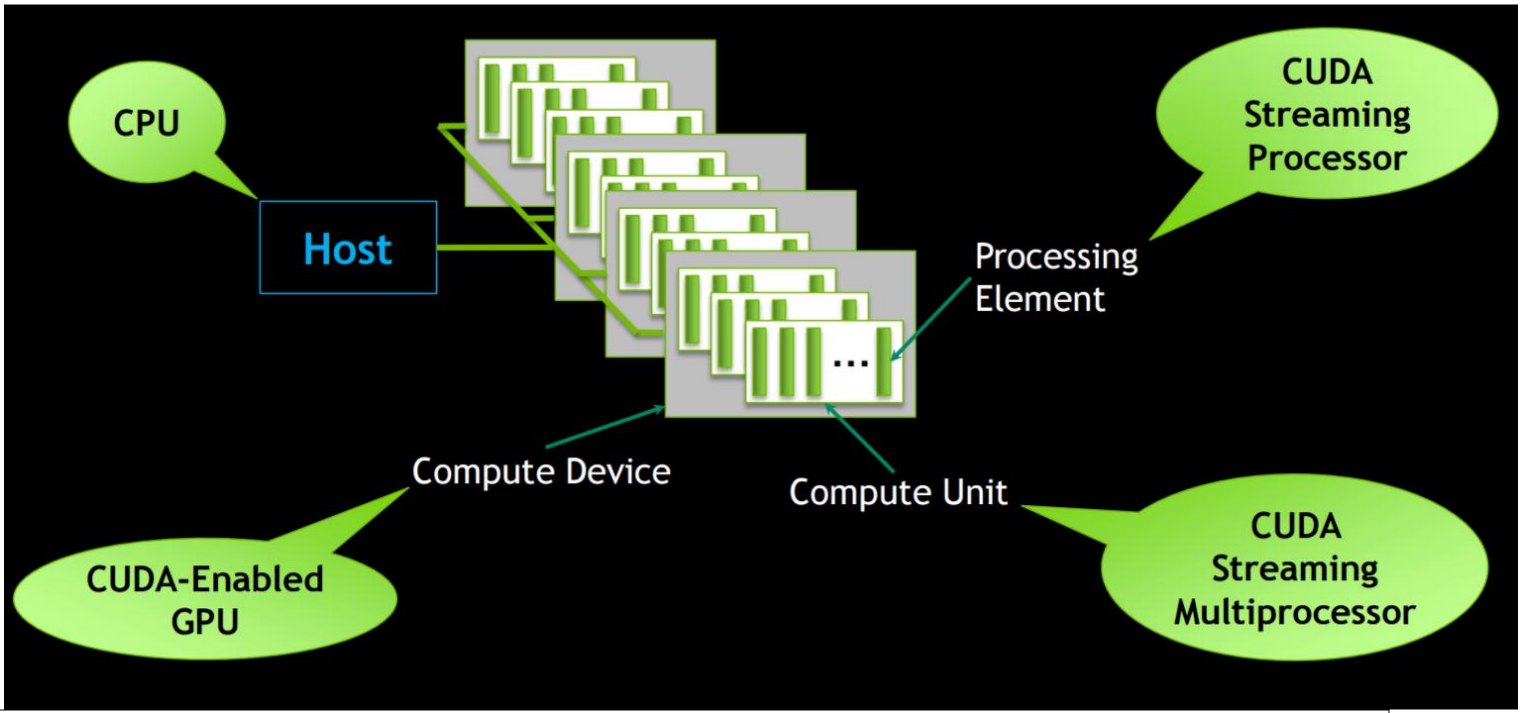


Some words about our memories:

- Global Memory: Shared with all Device, but slow. And is persistent between kernel calls.
- Constant Memory: Faster than global memory, use it for filter parameters
- Local Memory: Private to each compute unit, and shared to all processing elements.
- Private Memory: Faster but local to each processing element.

The Constant, Local, and private memory are scratch space so each you cannot save data there to be used by other kernels.

If you are coming from the CUDA word, this is how the OpenCl model fit on Cuda compute architecture.



Execution Model

OpenCl applications run on the Host, which submit work to the compute devices.

1. 1.

Work Item: Basic unit of work on a compute device

2. 2.

Kernel: The code that runs on a work item (Basically a C function)

3. 3.

Program: Collection of kernels and other functions

4. 4.

Context: The environment where work-items execute (Devices, their memories and command queues)

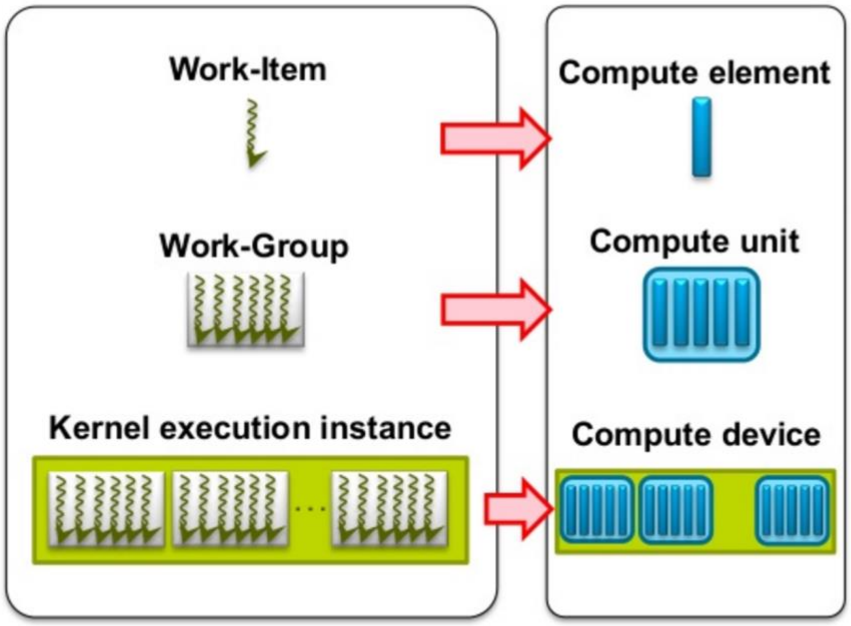
5. 5.

Command Queue: Queue used by the host to submit work (kernels, memory copies) to the device.

It's a framework that define how kernel execute on each point on a problem (N-Dimension vector). Or can be seen as the decomposition of a task in work-items.

What need to be defined:

- Global Work-size: Number of elements on your input vector.
- Global offset
- Work-group size: Size of your compute partition.



- Each **work-item** is executed by a **compute element**
- Each **work-group** is executed on a **compute unit**
- Several concurrent **work-groups** can reside on one **compute unit** depending on work-group's memory requirements and compute unit's memory resources
- Each **kernel** is executed on a **compute device**