OpenCL

Open Computing Language

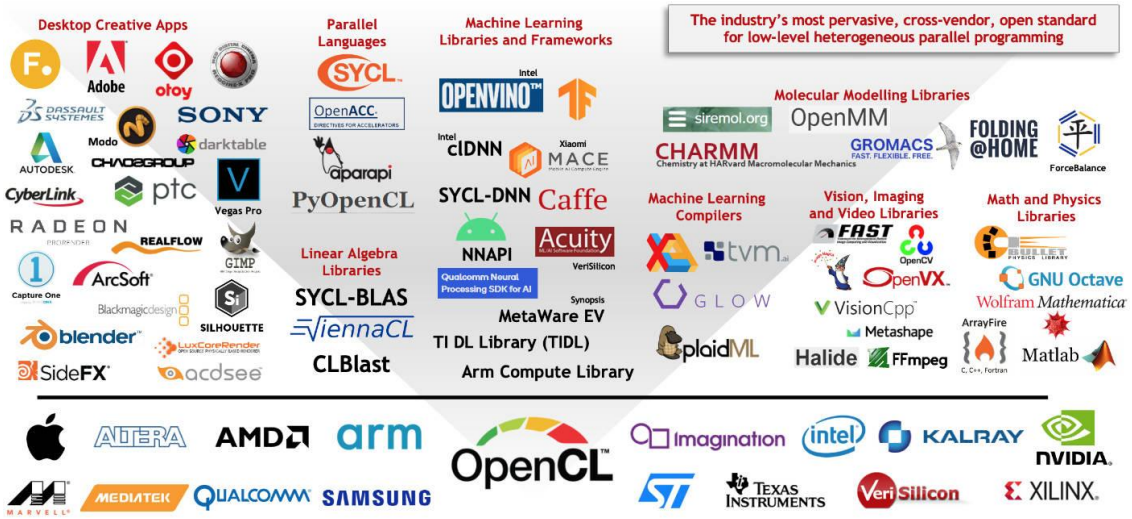
André Mora José Fonseca

Advertising… or maybe not…

- Are you interested in getting orders-of-magnitude performance increases over standard multi-core processors, while programming with a high-level language such as C or C#?

OpenCL multi-threaded platform might be the answer!

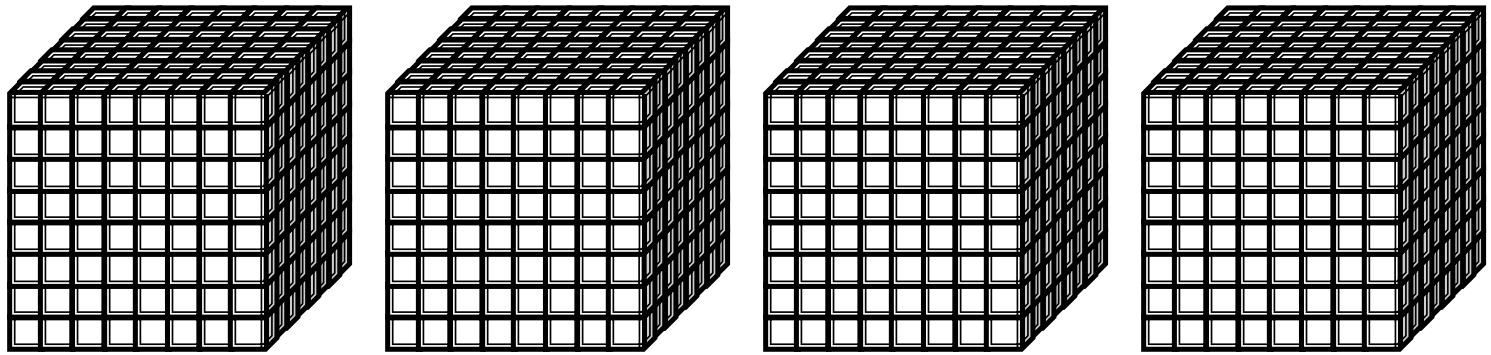
- However, not all problems can be mapped efficiently onto multi-threaded hardware…



What is GPU for?

GPU is especially well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel

= MASSIVE PARALELISM =



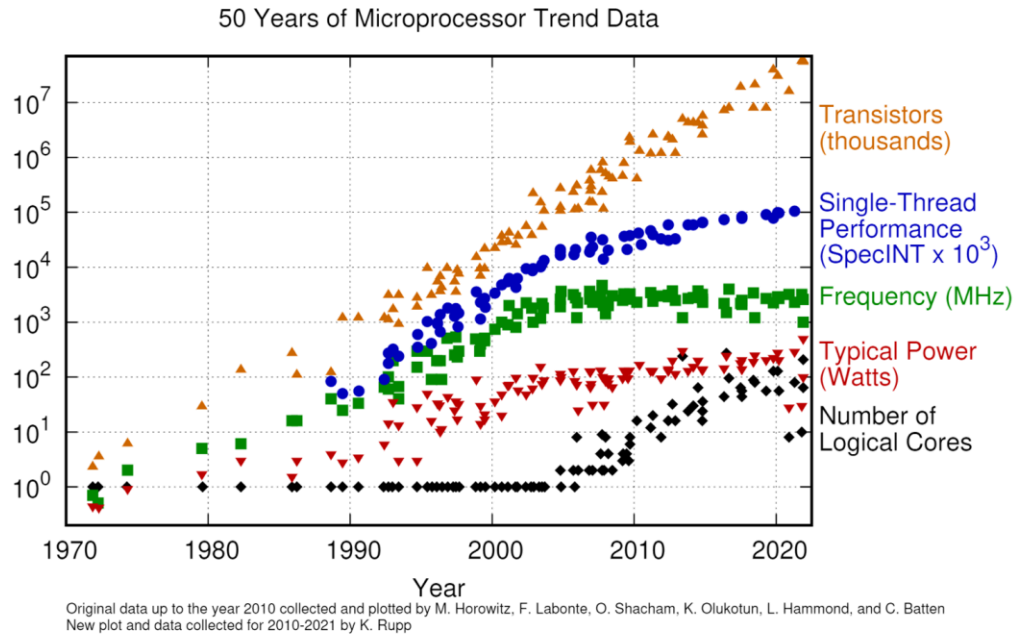
Grid of 4 3-dimensional blocks of CUDA threads

Recent history

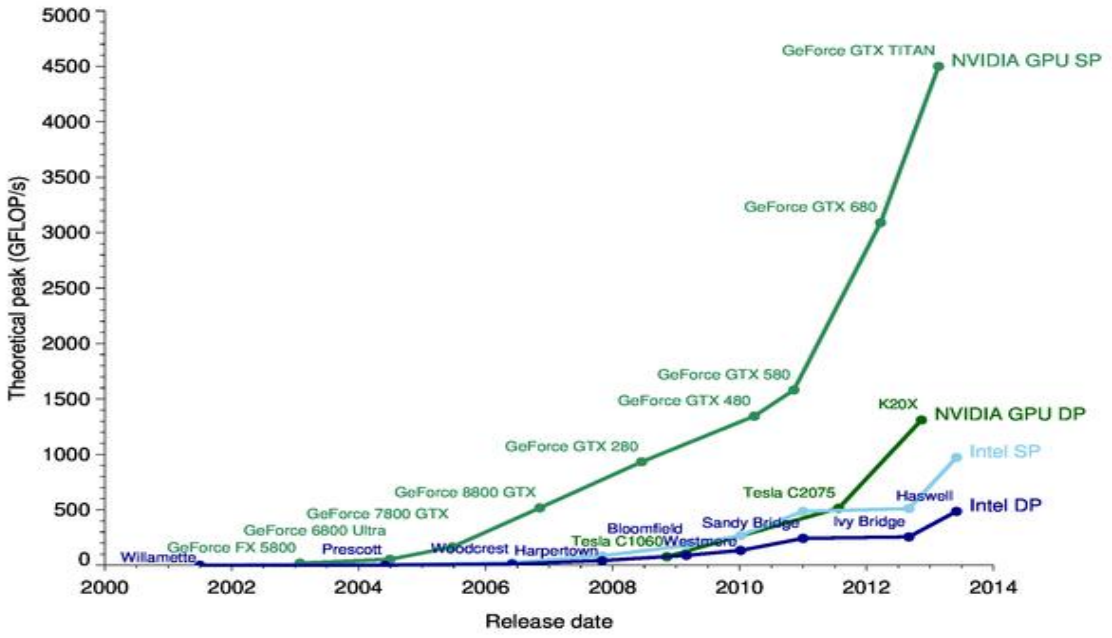
- Specialized machines faded out (e.g. CRAY)
  - Cost, economies of scale
- Intel and AMD CPUs designed for home/office use
  - Increasing clock frequencies gave increasing performance, but it is close to the technological limit
  - Multi-core is become the solution
- Computer gaming (and more recently AI) drives Graphics Processing Unit (GPU)
  - NVIDIA and AMD

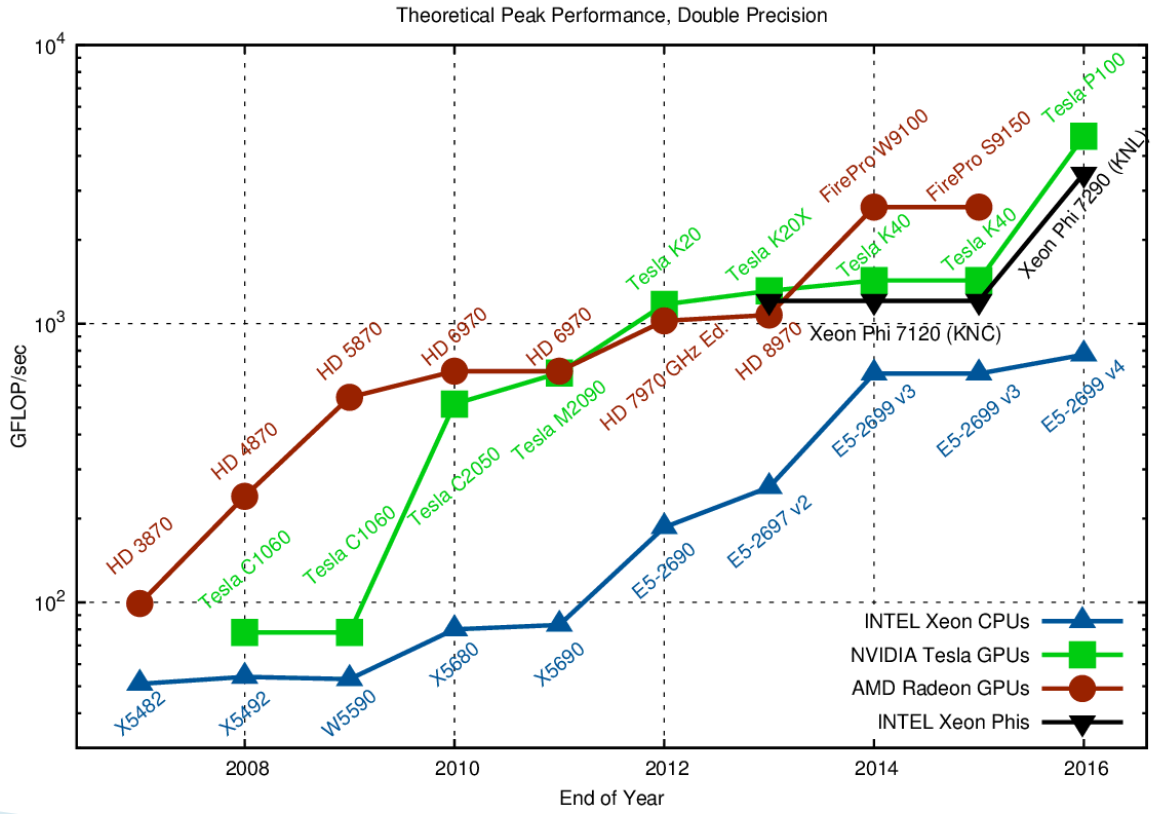
Present

- Clock frequency no longer increasing
  - more transistors -> more operations per second
  - size decreasing -> clock frequency increasing
  - size decreasing stalled at 2003
- Augment clock frequency, rises the dissipated heat and the electrical consumption imposing limits
- Processor parallelism is the way to increase the performance
- Most programs don't parallelize its algorithms wasting the resources of these new processors



GPU: A Highly Parallel, Multithreaded, Manycore Processor



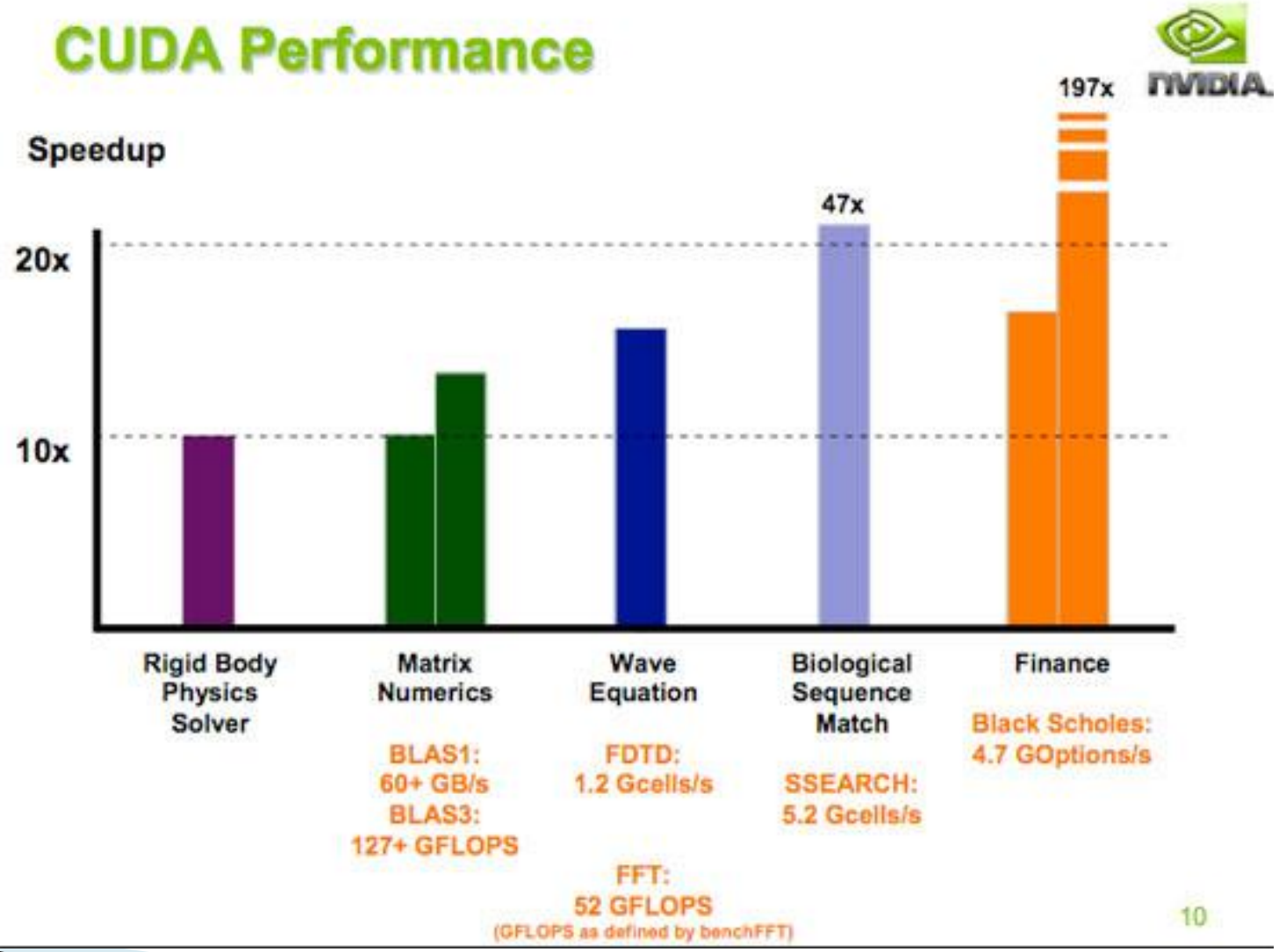


NVidia state-of-art processors

Features	DGX GB300 NVL72	HGX A100	Tesla V100	Tesla P100	Tesla K80	GeForce RTX 4090
Number and Type of GPU	72 NVIDIA Blackwell Ultra GPUs, 36 NVIDIA Grace CPUs	16 Ampere A100	1 Volta	1 Pascal GP100	2 Kepler GK210	Ada Lovelace
Peak double precision floating point performance	100 TFLOPS	312 TFLOPS	7.8 TFLOPS	5.31 Tflops (GPU Boost Clocks)	2.91 Tflops (GPU Boost Clocks)	
Peak single precision floating point performance	6PFLOPS	10 Pflops	15.7 TFLOPS	10.6 Tflops (GPU Boost Clocks)	8.74 Tflops (GPU Boost Clocks) 5.6 Tflops (Base Clocks)	
Memory bandwidth	21 TB/sec	2 TB/sec	900GB/sec	720 GB/sec	480 GB/sec (240 each)	
Memory size		80GB	32GB HBM2	16 GB	24 GB	24 GB
CUDA cores	>18000	8192	5,120	3586	4992 (2496 each)	16384
Tensor Cores	>564	512	640	-	-	

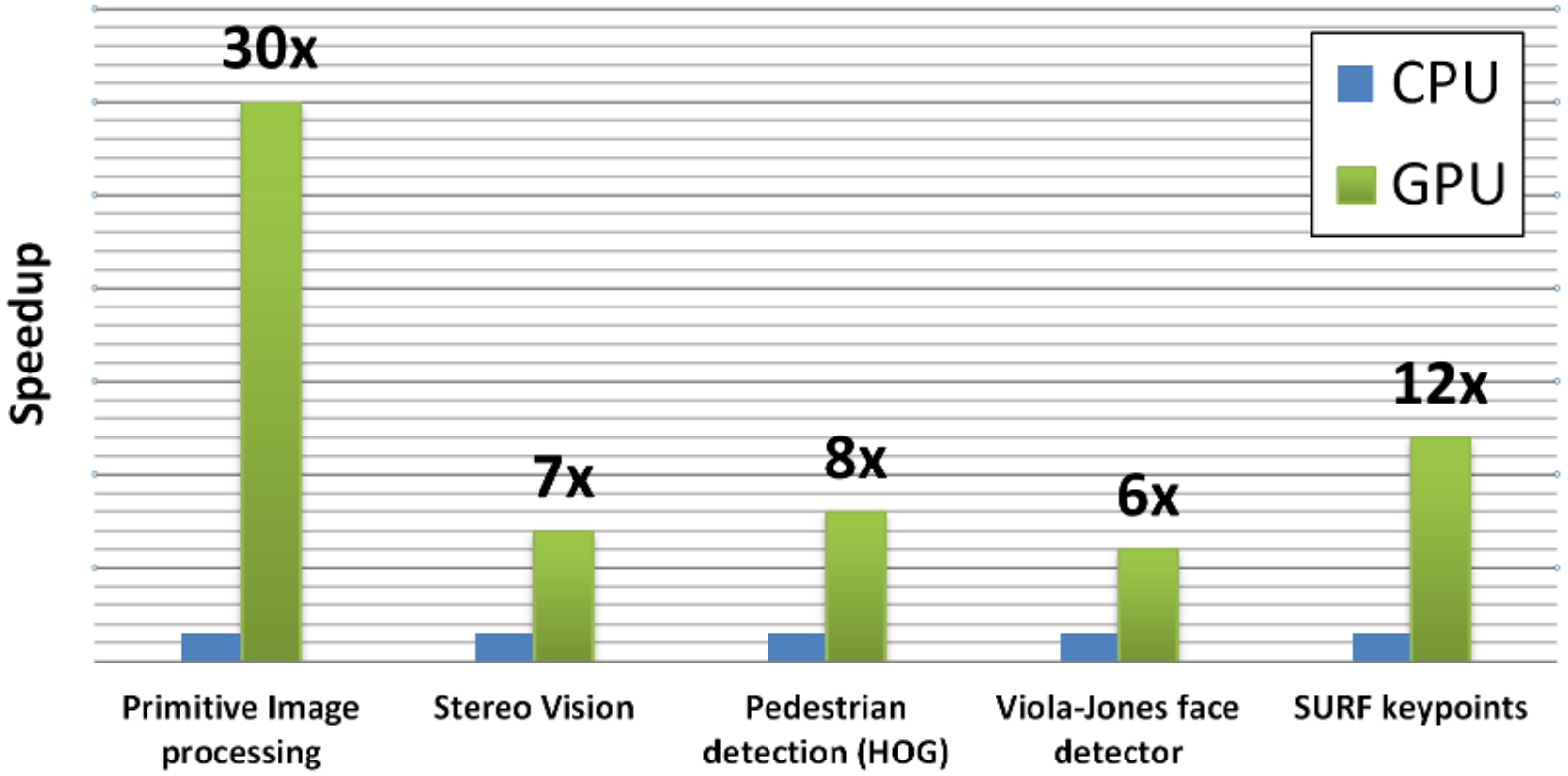
http://www.nvidia.com/
April - 2025

Successful applications



https://www.anandtech.com/show/2586

OpenCV implementation speedup

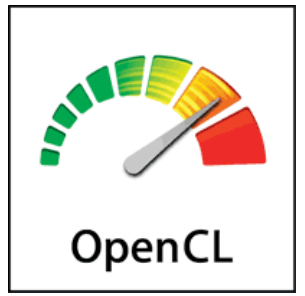


What is CUDA and OpenCL?

What is CUDA?

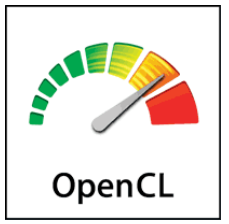
- CUDA is NVIDIA's parallel computing architecture.
- It enables dramatic increases in computing performance by harnessing the power of the GPU.
- Applications:
  - image and video processing
  - Artificial intelligence (deep learning)
  - computational biology and chemistry
  - fluid dynamics simulation
  - CT image reconstruction
  - and much more.

OpenCL
Open Computing Language



- OpenCL is an API for coordinating parallel computation across heterogeneous processors
- Cross-vendor, non-proprietary solution
- The OpenCL architecture is a close match to the CUDA architecture

OpenCL History



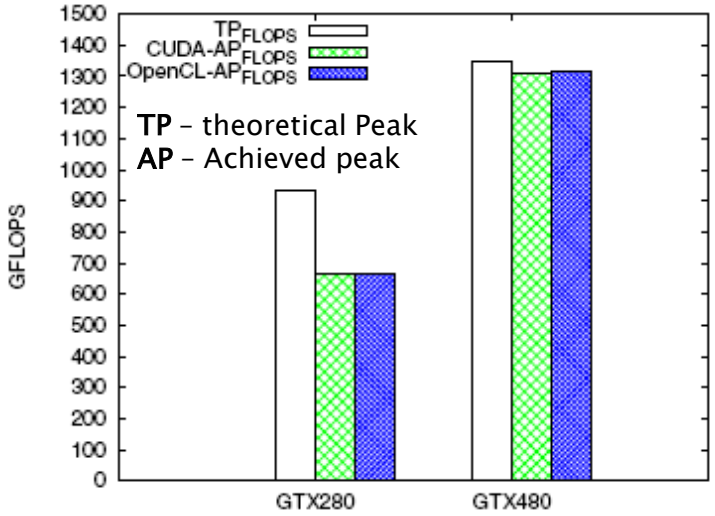
- It maintained by the non-profit technology consortium Khronos Group (www.khronos.org)
- It was proposed to Khronos in the summer of 2008, by Apple, Nvidia, Intel, and AMD
  - (CUDA was born earlier in 2006…)
- It has been adopted by Intel, AMD, NVidia, and ARM Holdings
- Interfaces to independent vendors are made through vendors' drivers
- Most up-to-date version is OpenCL 3.0 – (released April-2020)

New in OpenCL 2.0

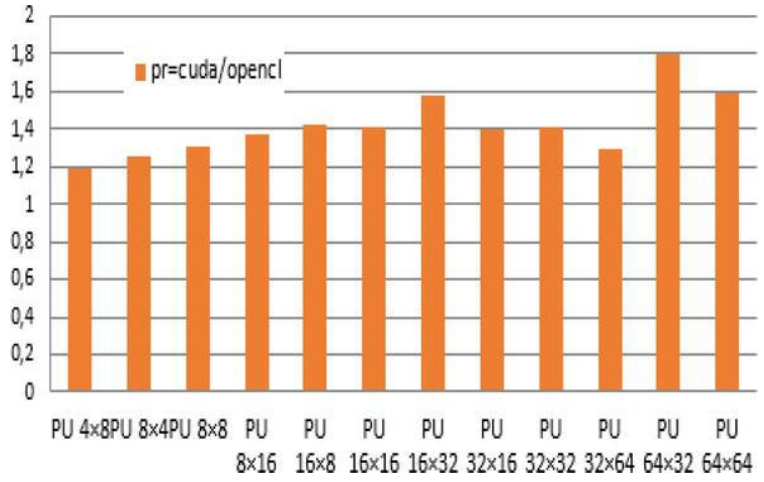
- **Shared Virtual Memory** – Host and device kernels can directly share complex, pointer-containing data structures
- **Nested Parallelism** – Device kernels can enqueue kernels to the same device with no host interaction
- **Generic Address Space** – Functions can be written without specifying a named address space for arguments
- **Other features:**
  - **Images** – Improved image support including sRGB images and 3D image writes.
  - **C11 Atomics** – A subset of C11 atomics and synchronization operations to enable assignments in one work-item to be visible to other work-items in a work-group, across work-groups executing on a device or for sharing data between the OpenCL device and host.
  - **Pipes** – Pipes are memory objects that store data organized as a FIFO and OpenCL 2.0 provides built-in functions for kernels to read from or write to a pipe.

OpenCL vs CUDA

- *"From the results and analysis above, we can see that there is no reason for OpenCL to obtain worse performance than CUDA under a fair comparison. "*



Jianbin Fang; Varbanescu, A.L.; Sips, H., "A Comprehensive Performance Comparison of CUDA and OpenCL," *Parallel Processing (ICPP), 2011 International Conference on* , vol., no., pp.216,225,2011



Khemiri, R., Bouaafia, S., Bahba, A., Nasr, M., & Ezahra Sayadi, F. Performance Analysis of OpenCL and CUDA Programming Models for the High Efficiency Video Coding. *IntechOpen*. doi: 10.5772/intechopen.99823 (2022).

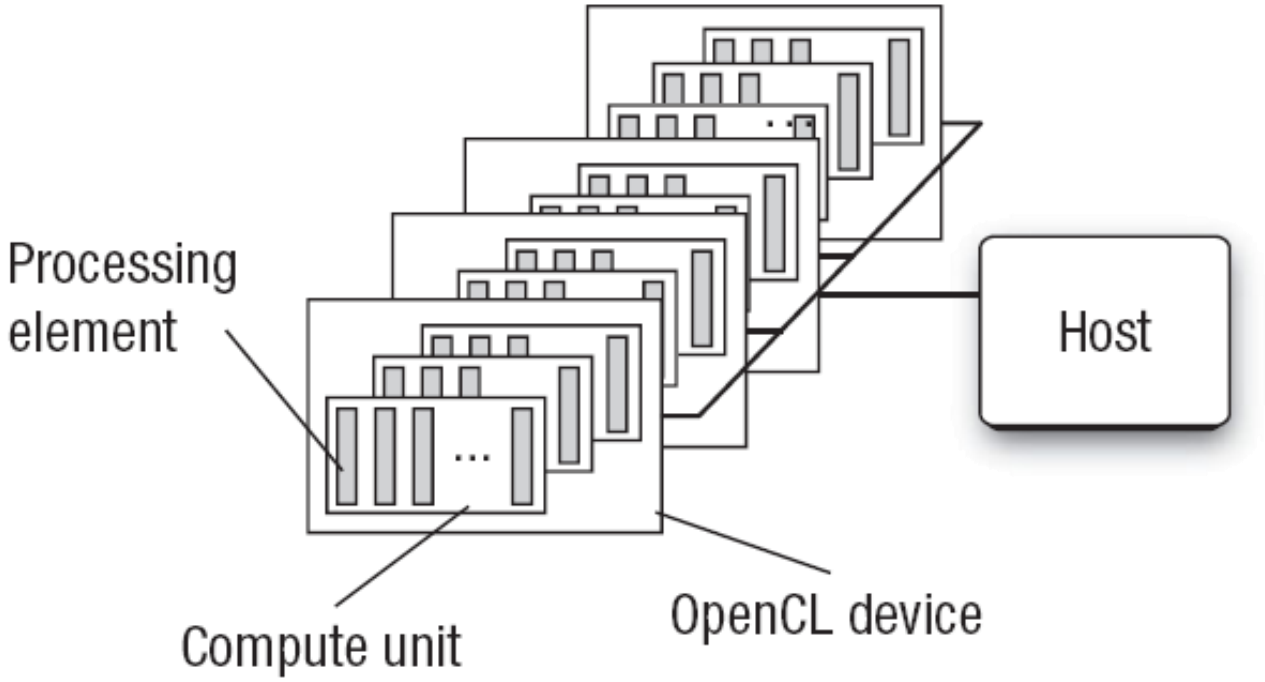
TABLE I
A COMPARISON OF GENERAL TERMS [13]

CUDA terminology	OpenCL terminology
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory
Thread	Work-item
Thread-block	Work-group

Basic concepts

Basic concepts

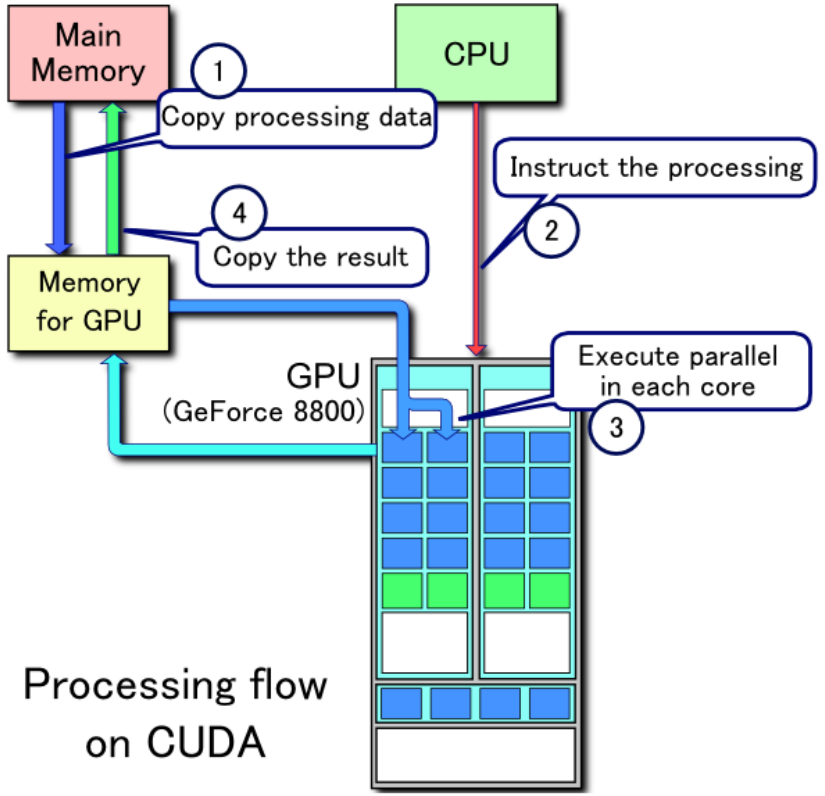
- **What are the computing units?**
  - In OpenCL it is GPU, CPU or even an FPGA
  - Several devices (GPUs + CPU) can co-exist on the same system



The co-paradigm

- Computing is evolving from "central processing" on the CPU to "co-processing" on the CPU and GPU.

- The main idea is:
  1. pass information from the CPU to the OpenCL Device (GPU)
  2. get it computed by the OpenCL Device (GPU)
  3. get back the results from the Device (GPU) to the CPU



OpenCL Philosophy

OpenCL extends C (or C#) by allowing the programmer to define functions, called ***kernels,*** that, when called, are executed N times in parallel by N different ***threads,*** in OpenCL-compliant ***devices*** *as* opposed to only once like regular C functions

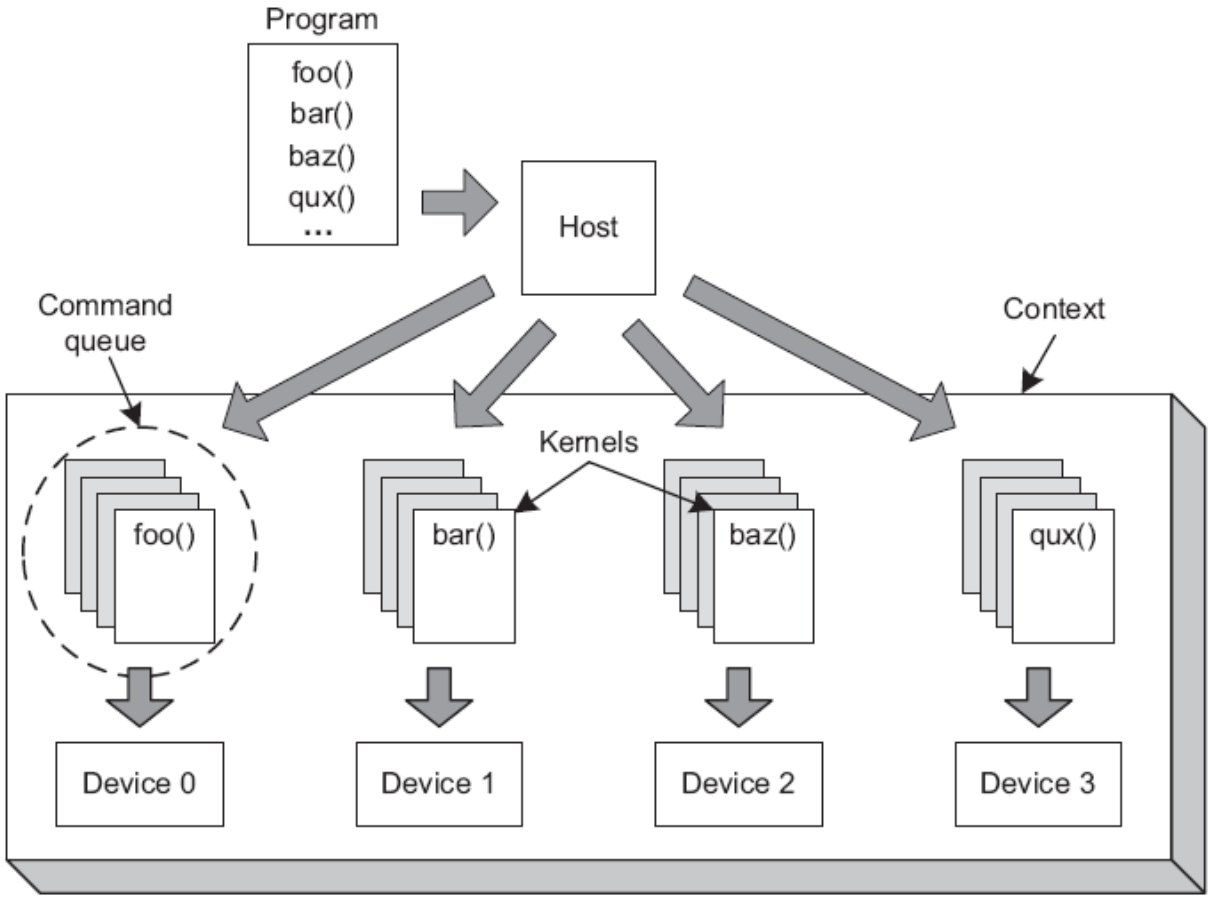
Thread vs process

- Both threads and processes are methods of parallelizing an application
  - **Processes** are independent execution units that contain their own state information, use their own address spaces, and only interact with each other via interprocess communication mechanisms
  - **Threads** share the same state and same memory space and can communicate with each other directly, because they share the same variables (a single process might contain multiple threads)

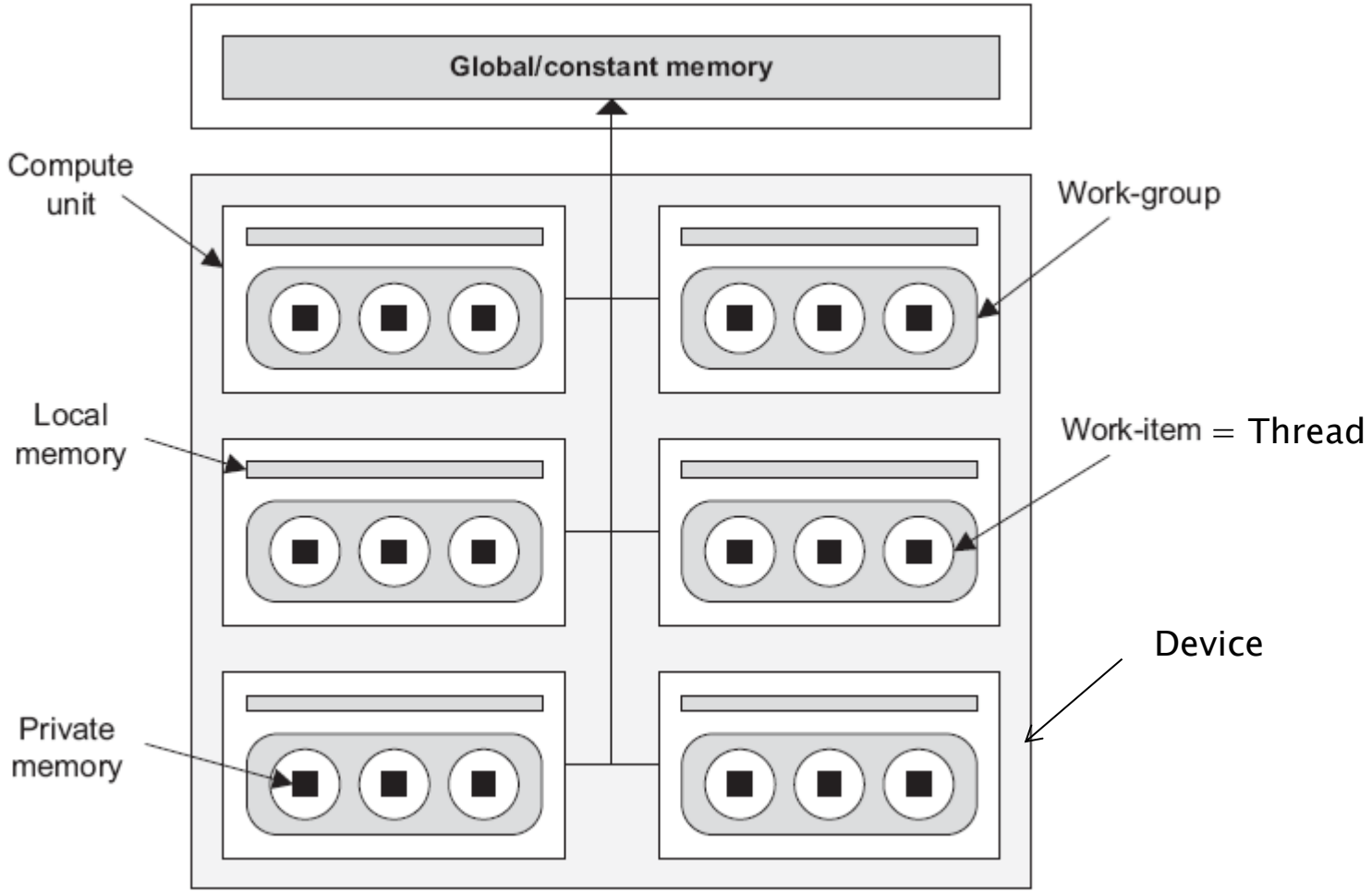
Basic concepts

- What is a OpenCL kernel?
  - It is a function callable from the host and executed on the OpenCL device
  - Kernels are simultaneous executions of **threads** in parallel
  - It is possible to define how many **threads** in a group of threads and the number of groups for each kernel execution
  - Kernel execution is asynchronous from the host
  - It is possible to synchronize host and device trough the adequate functions

Kernel Distribution among Devices



Device General Architecture



PyOpenCL

PyOpenCL

- **PyOpenCL** lets you access GPUs and other massively parallel compute devices from Python. It tries to offer computing goodness in the spirit of its sister project PyCUDA.
- **PyOpenCL** is be available at:

  https://pypi.org/project/pyopencl/

- **PyOpenCL** offers:
  - **Object cleanup** – tied to lifetime of objects.
  - **Completeness** – the full OpenCL's API is available
  - **Automatic Error Checking** – All CL errors are automatically translated into Python exceptions.
  - **Speed** – **PyOpenCL's** base layer is written in C++
  - **Documentation**
  - **Broad support** – **PyOpenCL** was tested and works with Apple's, AMD's, and Nvidia's CL implementations.

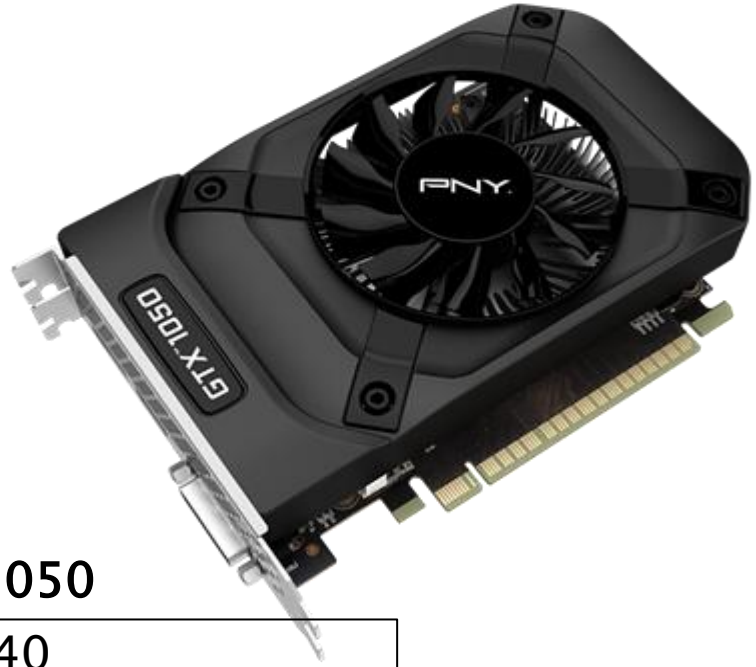
PyOpenCL Main Classes

- **Platform** – Represents the platform where OpenCL is being executed (Host) (Device Driver)
- **Device** – Represents a single device that is recognized by OpenCL in the system.
- **Program** – Contains the cl program and compiles it.
- **Kernel** – Represent the kernel, its parameters and controls its execution.
- **PyOpenCL** – API containing all OpenCL functions
- …

Getting device properties

Getting device properties

- Information that you can get:
  - Device Name
  - Vendor Name
  - Max Computing Units
  - Max Work Group Size
  - Global Memory Size
  - Local Memory Size
  - Image2DMaxHeight
  - Image2DMaxWidth
  - OpenCLVersion



GeForce GTX 1050

CUDA Cores 640
Graphics Clock (MHz) 1354
Computing Units: 2

GeForce GTX 3050 Ti

CUDA Cores 1280
Graphics Clock (MHz) 1035
Computing Units: 20

Platform & Device Information

```
import pyopencl as cl

platforms = cl.get_platforms()

for platform in platforms:
    name = platform.get_info(cl.platform_info.NAME)
    vendor = platform.get_info(cl.platform_info.VENDOR)
    version = platform.get_info(cl.platform_info.VERSION)

    displayStr = "Name: " + name + "\nVendor: " + vendor + "\nVersion: " + version + "\n"
    iF.showMessageBox(title="Platform Info", message=displayStr)

    devices = platform.get_devices()
(…)
```

Get drivers

Get platform info

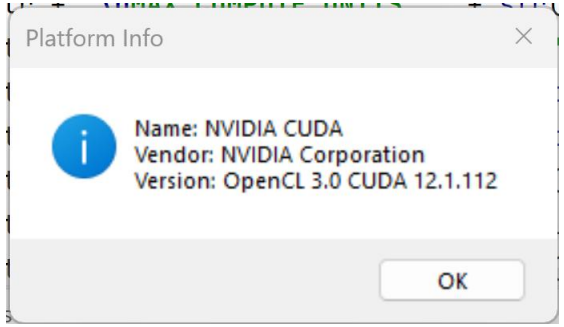
Get Devices

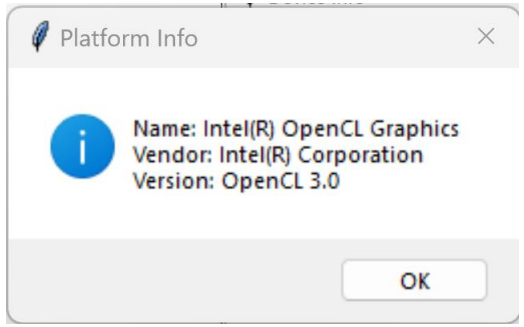
Platform & Device

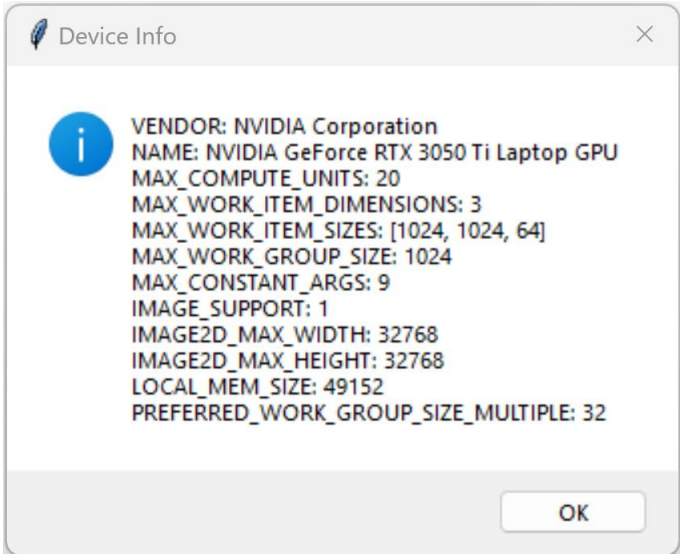
```
(...)
for device in devices:
    displayStr = "VENDOR: " + device.get_info(cl.device_info.VENDOR)
    displayStr = displayStr + "\nNAME: " + device.get_info(cl.device_info.NAME)
    displayStr = displayStr + "\nMAX_COMPUTE_UNITS: " + str(device.get_info(cl.device_info.MAX_COMPUTE_UNITS))
    displayStr = displayStr + "\nMAX_WORK_ITEM_DIMENSIONS: " +
                                        str(device.get_info(cl.device_info.MAX_WORK_ITEM_DIMENSIONS))
    displayStr = displayStr + "\nMAX_WORK_ITEM_SIZES: " + str(device.get_info(cl.device_info.MAX_WORK_ITEM_SIZES))
    displayStr = displayStr + "\nMAX_WORK_GROUP_SIZE: " + str(device.get_info(cl.device_info.MAX_WORK_GROUP_SIZE))
    displayStr = displayStr + "\nMAX_CONSTANT_ARGS: " + str(device.get_info(cl.device_info.MAX_CONSTANT_ARGS))
    displayStr = displayStr + "\nIMAGE_SUPPORT: " + str(device.get_info(cl.device_info.IMAGE_SUPPORT))
    displayStr = displayStr + "\nIMAGE2D_MAX_WIDTH: " + str(device.get_info(cl.device_info.IMAGE2D_MAX_WIDTH))
    displayStr = displayStr + "\nIMAGE2D_MAX_HEIGHT: " + str(device.get_info(cl.device_info.IMAGE2D_MAX_HEIGHT))
    displayStr = displayStr + "\nLOCAL_MEM_SIZE: " + str(device.get_info(cl.device_info.LOCAL_MEM_SIZE))
    displayStr = displayStr + "\nPREFERRED_WORK_GROUP_SIZE_MULTIPLE: " +
                                    str(device.get_info(cl.device_info.PREFERRED_WORK_GROUP_SIZE_MULTIPLE))


    iF.showMessageBox(title="Device Info", message=displayStr)
```

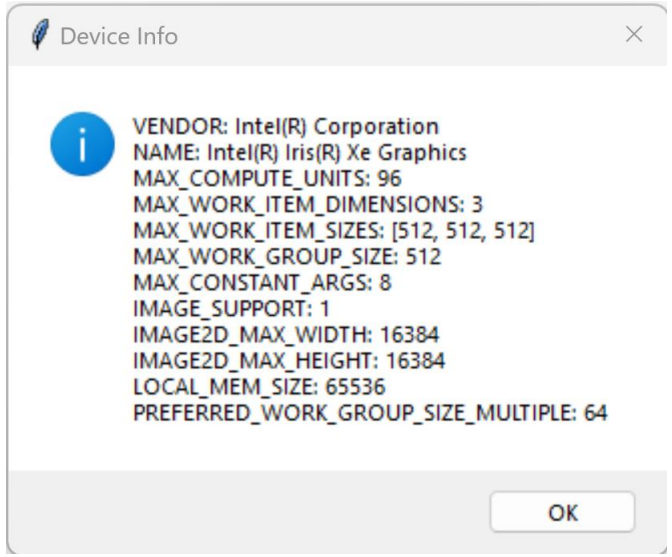
Get Device info

Output


Platform Info
Name: NVIDIA CUDA
Vendor: NVIDIA Corporation
Version: OpenCL 3.0 CUDA 12.1.112
OK


Platform Info
Name: Intel(R) OpenCL Graphics
Vendor: Intel(R) Corporation
Version: OpenCL 3.0
OK


Device Info
VENDOR: NVIDIA Corporation
NAME: NVIDIA GeForce RTX 3050 Ti Laptop GPU
MAX_COMPUTE_UNITS: 20
MAX_WORK_ITEM_DIMENSIONS: 3
MAX_WORK_ITEM_SIZES: [1024, 1024, 64]
MAX_WORK_GROUP_SIZE: 1024
MAX_CONSTANT_ARGS: 9
IMAGE_SUPPORT: 1
IMAGE2D_MAX_WIDTH: 32768
IMAGE2D_MAX_HEIGHT: 32768
LOCAL_MEM_SIZE: 49152
PREFERRED_WORK_GROUP_SIZE_MULTIPLE: 32
OK

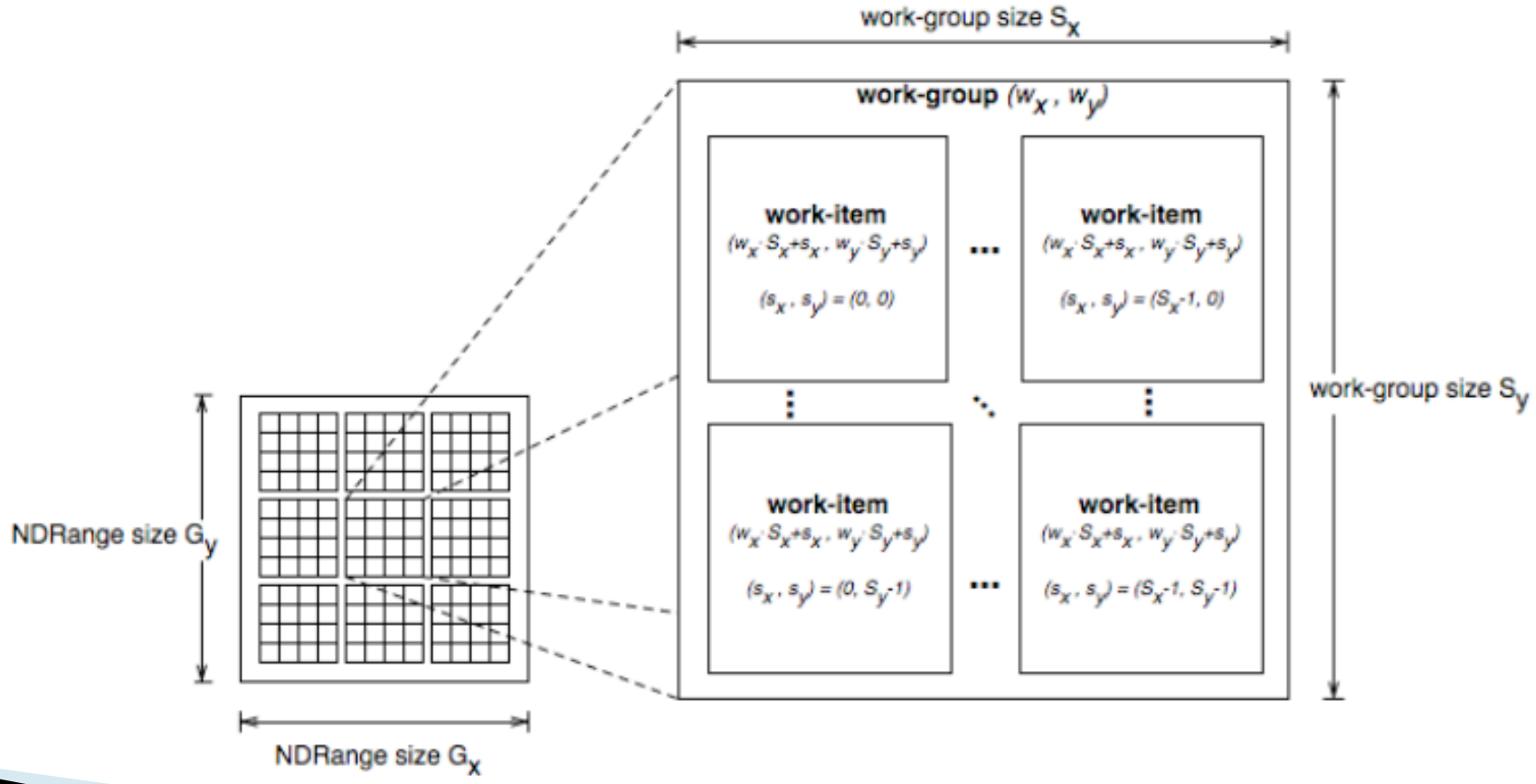

Device Info
VENDOR: Intel(R) Corporation
NAME: Intel(R) Iris(R) Xe Graphics
MAX_COMPUTE_UNITS: 96
MAX_WORK_ITEM_DIMENSIONS: 3
MAX_WORK_ITEM_SIZES: [512, 512, 512]
MAX_WORK_GROUP_SIZE: 512
MAX_CONSTANT_ARGS: 8
IMAGE_SUPPORT: 1
IMAGE2D_MAX_WIDTH: 16384
IMAGE2D_MAX_HEIGHT: 16384
LOCAL_MEM_SIZE: 65536
PREFERRED_WORK_GROUP_SIZE_MULTIPLE: 64
OK

OpenCL Programming

- A first simple application

WorkGroup Organization

- This is 2D organization but can be extended to 3D



A simple Kernel program

```
__kernel void power2(__global float* arr)
{
      int i = get_global_id(0);
      arr[i] = arr[i] * arr[i];
}
```

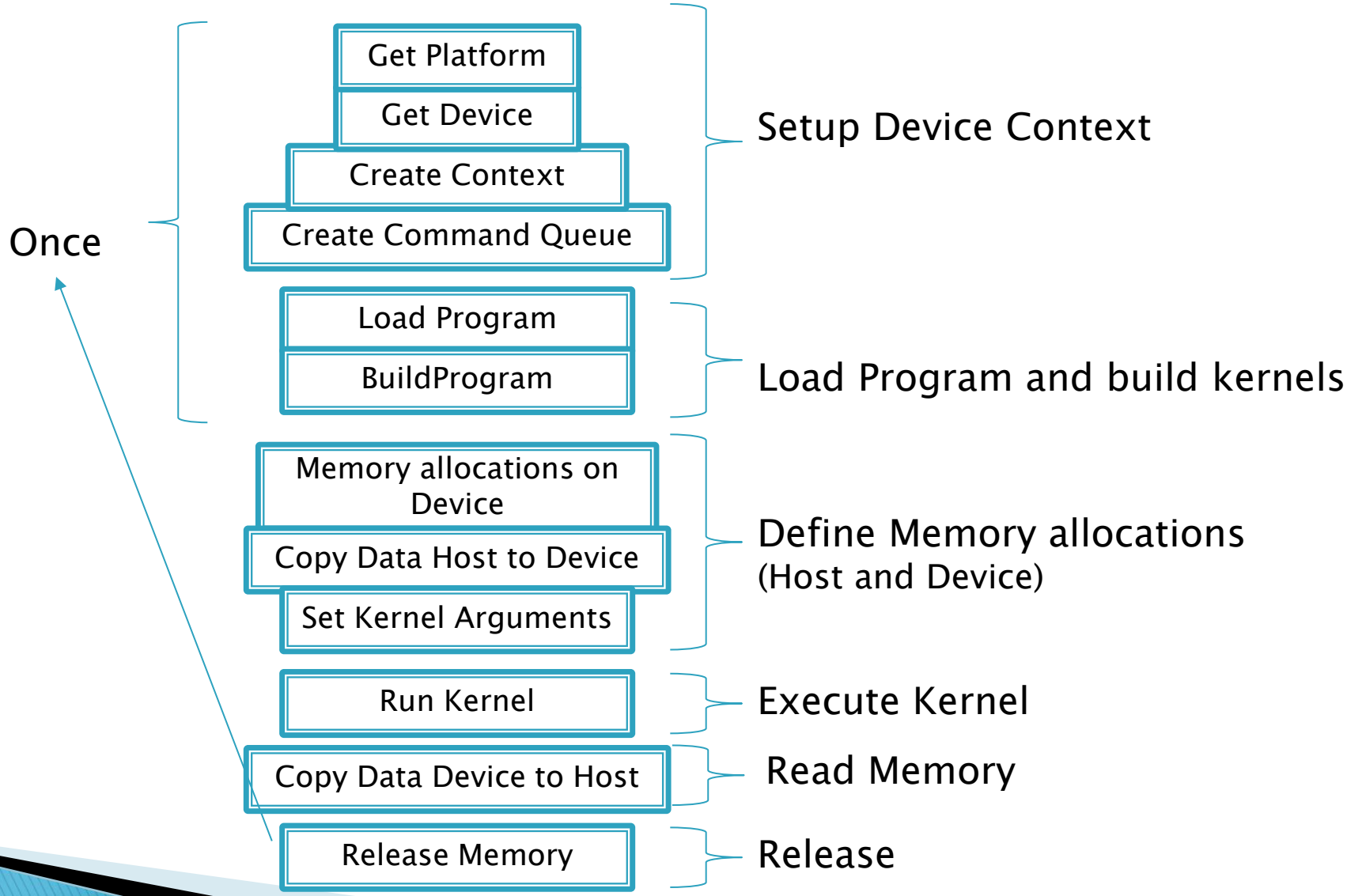
__global, __local or __constant
(for variables there is also __private)

Get Thread ID

- Built in variables:

	get_work_dim ()	*Number of coordinates (2 or 3)*
For each dimension	get_global_size (uint *dimindx)*	*Total number of work-items*
	get_num_groups (uint *dimindx)*	*Number of work-groups (Grid size)*
	get_local_size (uint *dimindx)*	*Work-group size (block size)*
	get_global_id (uint *dimindx)*	*Global thread id number*
	get_group_id (uint *dimindx)*	*Work-group id number*
	get_local_id (uint *dimindx)*	*Work-group thread id number*

Program sequence



Program – Setup

```
try:
    plaforms = cl.get_platforms()
    global plaform
    plaform = plaforms[0]

    devices = plaform.get_devices()
    global device
    device = devices[0]

    global ctx
    ctx = cl.Context(devices) # or dev_type=cl.device_type.ALL)

    global commQ
    commQ = cl.CommandQueue(ctx,device)

    (…)
```

Configure platform and device

Set Context / configuration – ctx

Create a command Queue – comQ

Program – Build

- Driver with built-in compiler -> Runtime compiler
  - No need for compiler

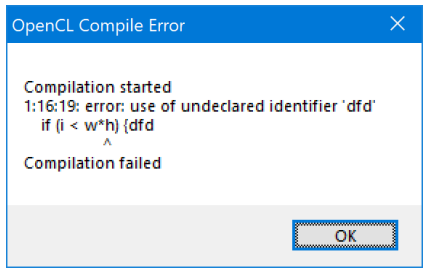
```
file = open("prog.cl","r")

    global prog
    prog = cl.Program(ctx,file.read())
    prog.build()


except Exception as e:
    print(e)
    return False

return True
```

File containing OpenCL Kernels

Load Program and Build
– Compile options are *optional*

– *Kernels are acessible through prog*

Catch OpenCL Exceptions



Program – Memory & Parameters

```
try:
    arrayIn = np.array([1,2,3,4,5,6,7,8,9,10], dtype=int)
    kernelName = prog.power2

    memBuffer = cl.Buffer(ctx,
            flags= cl.mem_flags.COPY_HOST_PTR | cl.mem_flags.READ_WRITE, hostbuf=arrayIn)

    kernelName.set_arg( 0, memBuffer)
```

Get Kernel

Create buffer and copy Host To Device

Set Kernel arguments
- argument Index
- argument value
  - buffer address
  - or numpy scalars

Example of multi-parameter setting

OpenCL Kernel:

```
__kernel void negative(__global uchar* image,
                       int w,
                       int h,
                       int padding,
                       __global uchar* imageOut)
```

PyOpenCL setting kernel arguments:

```
kernelName.set_arg( 0, bufferFilter)

kernelName.set_arg( 1, np.int32(width))

kernelName.set_arg( 2, np.int32(height))

kernelName.set_arg( 3, np.int32(padding))

kernelName.set_arg( 4, bufferFilterOut)
```

Program – Execute

```
globalWorkSize = (10,1)
workGroupSize = (5,1)

kernelEvent = cl.enqueue_nd_range_kernel( commQ, kernelName,
            global_work_size= globalWorkSize, local_work_size= workGroupSize )

kernelEvent.wait()

cl.enqueue_copy(commQ, arrayIn, memBuffer)

print(arrayIn)

memBuffer.release()
```

Total number of threads (work-items)

Number of threads per work-group (*block size*)

Execute Kernel & Wait to finish

Finally get the results from GPU and print

Release buffer memory

Dimensioning a Kernel for an image

How to dimension a GRID for processing an image with blocks of 32x8 threads (or any other size)?


```
localws = (32, 8) # openCV 32x8 = 256

globalws = (math.ceil(width / localws[0]) * localws[0],
                math.ceil(height / localws[1]) * localws[1])
```