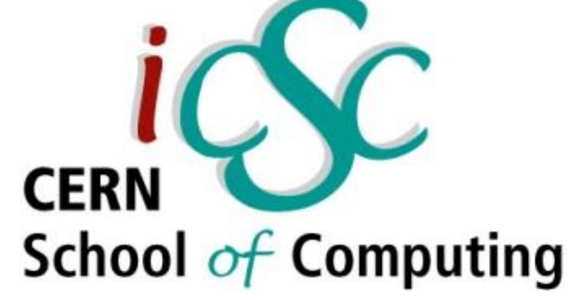
Multiplatform Programming with Python

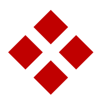
Peter Kicsiny

08/03/2023

Special thanks to: Giovanni Iadarola, Riccardo de Maria, Alberto Pace



6-9 March 2023



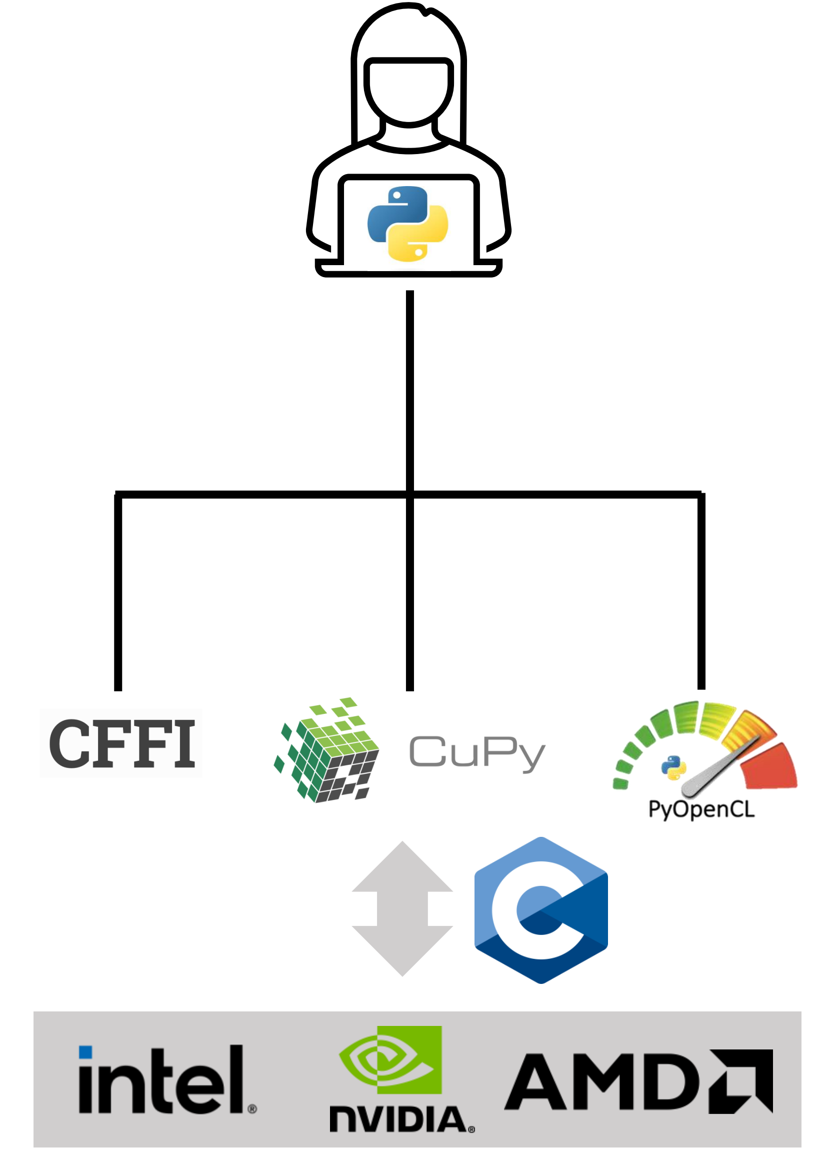
What is this course about?

Lecture

- Short recap on heterogeneous architectures & CUDA + OpenCL programming models
- Introduction to 3 Python libraries: **CFFI**, **CuPy** & **PyOpenCL**

Tutorial

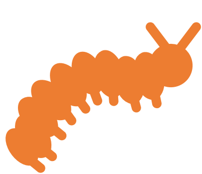
- Try out the libraries on toy examples



Lecture outline

Heterogeneous Programming Recap

- Motivation
- Hardware accelerators
- Heterogeneous systems

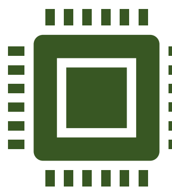


CPU Programming with Python

- Python & C
- CFFI intro

GPU Programming with Python

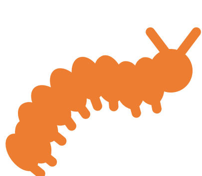
- CUDA & OpenCL models
- CuPy intro
- PyOpenCL intro



Lecture outline

Heterogeneous Programming Recap

- Motivation
- Hardware accelerators
- Heterogeneous systems

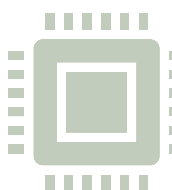


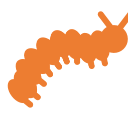
CPU Programming with Python

- Python & C
- CFFI intro

GPU Programming with Python

- CUDA & OpenCL models
- CuPy intro
- PyOpenCL intro





Motivation: HEP & particle accelerator physics

- Increasing demand for computing power
  - Petabytes of HEP data generated per hour: online & offline data processing
  - Heavy numerical simulations: Monte Carlo, finite element, ML, multiparticle tracking

Particle physics

Run 3 (μ=55) Run 4 (μ=88-140) Run 5 (μ=165-200)

ATLAS Preliminary
2022 Computing Model - CPU

- Conservative R&D
- Aggressive R&D
- Sustained budget model (+10% +20% capacity/year)

Annual CPU Consumption [MHS06years]

0 10 20 30 40 50

2020 2022 2024 2026 2028 2030 2032 2034 2036

Year

Image source: https://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/UPGRADE/CERN-LHCC-2022-005/

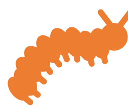
Beam dynamics & particle accelerator design

HiLumi HL-LHC PROJECT

FUTURE CIRCULAR COLLIDER

International MUON Collider Collaboration

Image source: https://abpcomputing.web.cern.ch/



Motivation: hardware trends

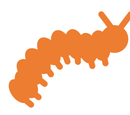
- Processing data has become much faster than accessing it in memory
  - Bring memory on the chip: cache
- Single CPU core performance saturates
  - Trend for many core CPUs

Energy consumption & heat dissipation limits increase of single core clock frequency

Performance gap

Performance
100,000
10,000
1000
100
10
1
Processor
Memory
1980 1985 1990 1995 2000 2005 2010
Year

Image source: Computer Architecture, A quantitative Approach, John L. Hennessy and David A. Patterson



Hardware accelerators

- CPU speed has flattened out
- Hardware accelerators can further speed up computations
- Specialized for specific types of tasks

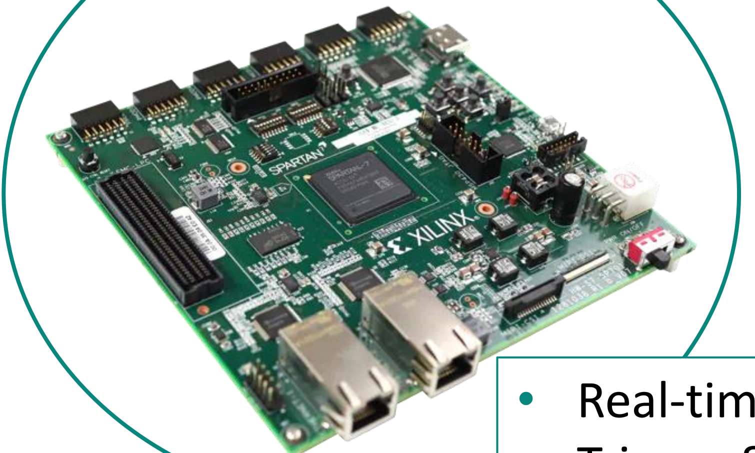
Tensor Processing Unit (TPU)

- Artificial intelligence

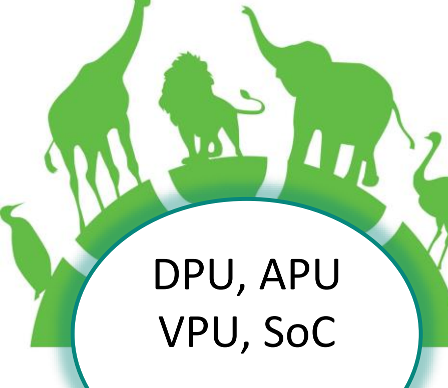
Graphics Processing Unit (GPU)

- Computer graphics
- General Purpose GPU (GPGPU): scientific computing

Field Programmable Gate Array (FPGA)



- Real-time systems
- Trigger & DAQ



DPU, APU
VPU, SoC
…

CPU vs GPU: peak performance

- Since 2010: GPUs are getting better at double precision computations
- Achievable speedup depends on code structure (Amdahl's law)

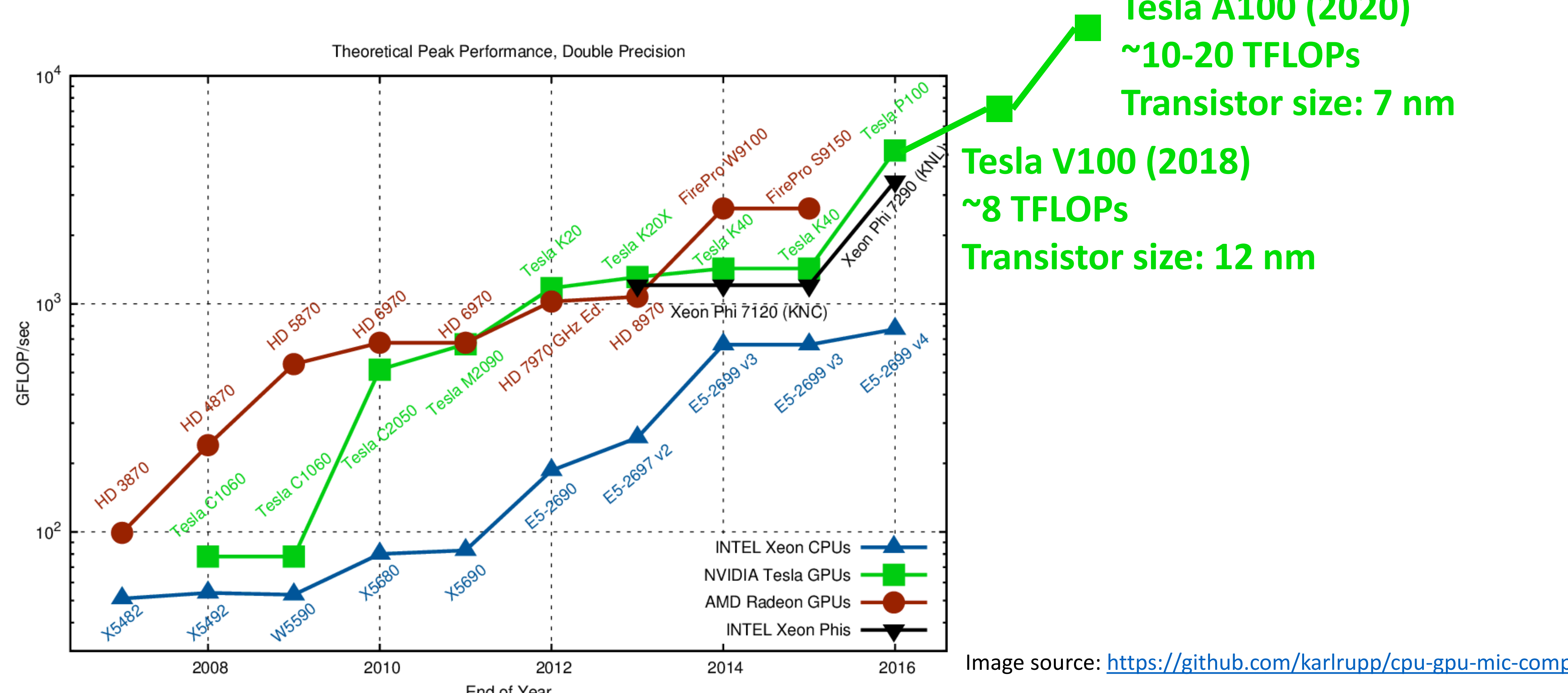
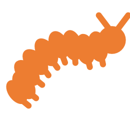
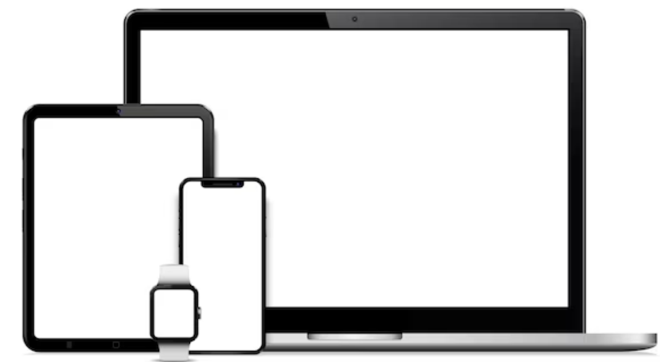


Image source: https://github.com/karlrupp/cpu-gpu-mic-comparison

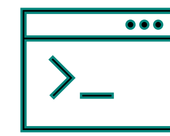


Heterogeneous systems

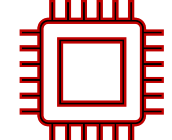
- Combine CPU & hardware accelerators
- Can be realized on chip, node or cluster level
- It is everywhere: compute clusters, PCs, laptops, phones, smart devices



- Main challenges:

Software 

- Compilers
- Portability of code
- Precision & reproducibility

Hardware 

- Memory access
- Communication
- Synchronization

Lecture outline

Heterogeneous Programming Recap

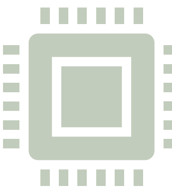
- Motivation
- Hardware accelerators
- Heterogeneous systems

CPU Programming with Python

- Python & C
- CFFI intro

GPU Programming with Python

- CUDA & OpenCL models
- CuPy intro
- PyOpenCL intro

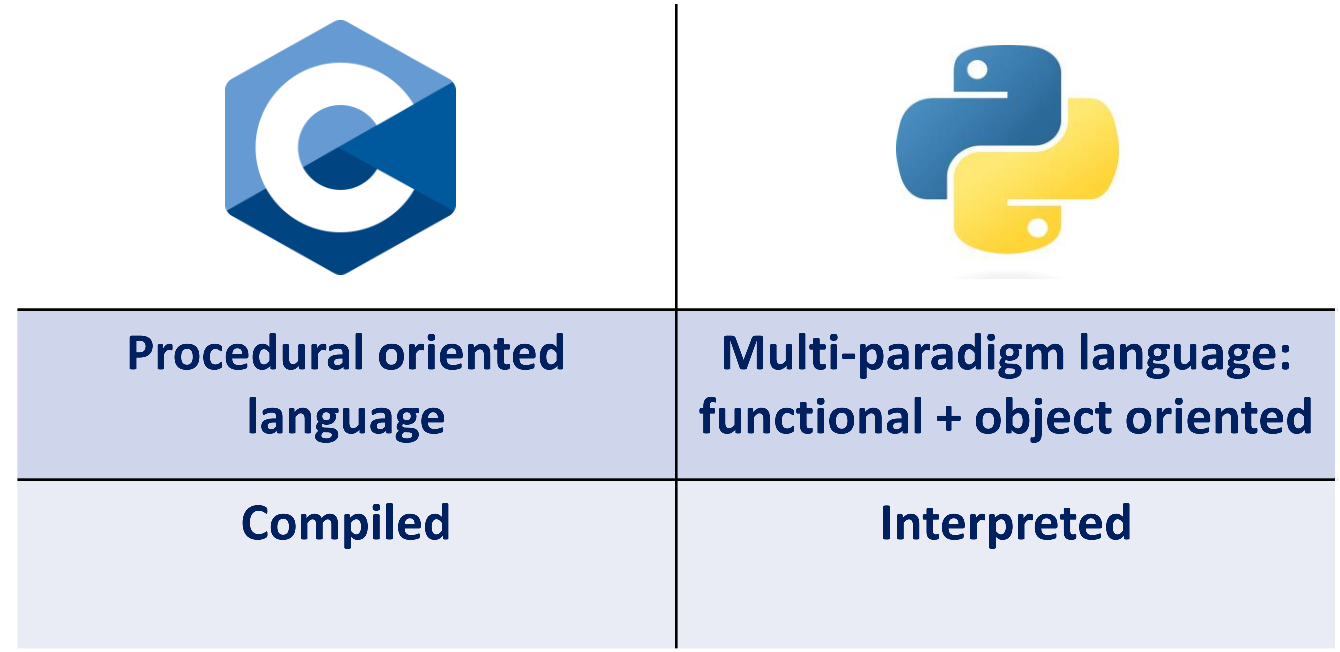


C vs Python: a basic comparison

Procedural oriented language	Multi-paradigm language: functional + object oriented

- Procedural oriented:
  - Imperative
  - Series of routines/functions to be executed
- Object oriented:
  - Imperative
  - Code organized in classes
  - Core concept is an object: contains functions & data which are updated
- Functional:
  - Declarative
  - More focus on (higher level) functions
  - Immutable data: instead of updates copies are created

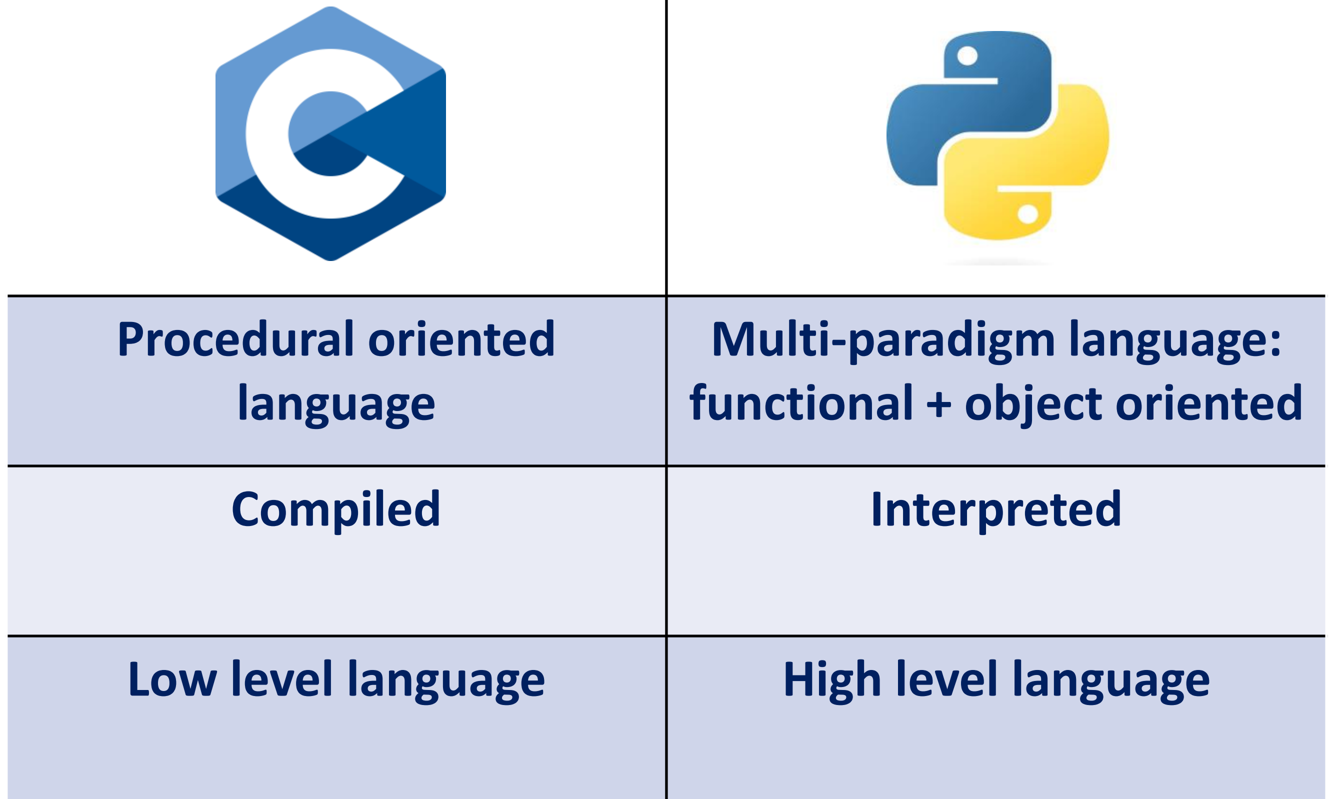
C vs Python: a basic comparison



C	Python
Procedural oriented language	**Multi-paradigm language: functional + object oriented**
Compiled	**Interpreted**

- Compiled:
  - Code translated into machine language before runtime
  - Platform specific
- Interpreted:
  - Code translated line by line during runtime: overhead
  - Platform independent

C vs Python: a basic comparison



C	Python
Procedural oriented language	**Multi-paradigm language: functional + object oriented**
Compiled	**Interpreted**
Low level language	**High level language**

- Low level:
  - More difficult syntax: you have to be explicit e.g. manual memory management
  - Allows more optimization: faster
  - Same program takes more lines of code
- High level:
  - Easier syntax: hides details from the programmer e.g. automatic memory management
  - Allows less optimization: slower
  - Same program takes less lines of code

C vs Python: a basic comparison

Procedural oriented language	**Multi-paradigm language: functional + object oriented**
Compiled	**Interpreted**
Low level language	**High level language**
Used for performance critical applications	**Used as “glue code” for scripting**

- Python is mainstream:
  - Processors are already fast: human cost > computational cost
  - Large developer community
  - General purpose language with a large set of libraries
  - Portable & flexible

C vs Python: a basic comparison

C	Python
Procedural oriented language	**Multi-paradigm language: functional + object oriented**
Compiled	**Interpreted**
Low level language	**High level language**
Used for performance critical applications	**Used as "glue code" for scripting**
"80% of the time spent on 20% of the code"	

- Computationally intensive parts can be written in C/C++(/Fortran)
- Rest (configuration, plotting, ...) can easily be scripted in Python
- Interface:
  - On CPU: Cython, ctypes, **CFFI**
  - On GPU: PyCUDA, **CuPy**, **PyOpenCL**

Application Programming Interfaces (APIs)

- From Wikipedia: *“An API is a way for two or more computer programs to communicate with each other.”*
  - Like a Python program with a C program, through the functions of a Python library
  - Or a program running on a CPU host with a GPU device’s parallel programming elements, through the functions of a parallel programming model
- APIs simplify the programming by abstracting (hiding) the details of the underlying communication

CFFI

- C Foreign Function Interface for Python
- Enables to import existing C libraries in Python
- Enables to build new modules from custom C code & use them in Python

- API mode (more common)
- ABI (Application Binary Interface) mode (less common)

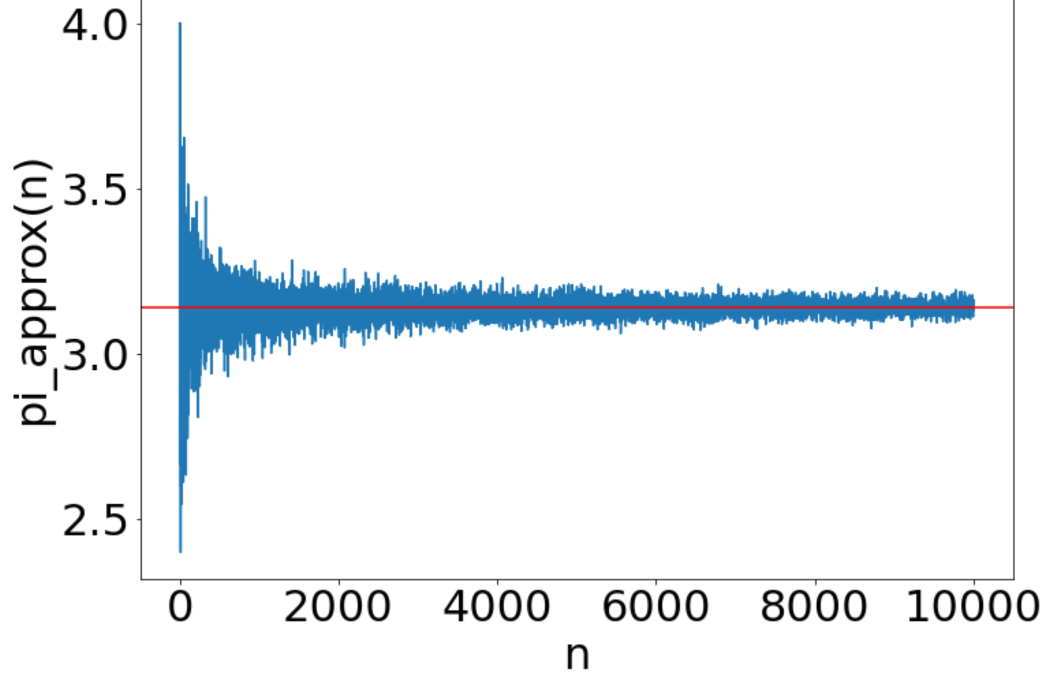
- Out-of-line mode (API/ABI) (more common)
- In-line mode (API/ABI) (less common)

CFFI: calling C sources (out-of-line API mode)

- C routine to approximate π

```c
/* filename: pi.c*/
# include <stdlib.h>
# include <math.h>

/* Returns a very crude
approximation of Pi
   given a int: a number of
iteration */
float pi_approx(int n){

  double i,x,y,sum=0;

  for(i=0;i<n;i++){
    x=rand();
    y=rand();
    if (sqrt(x*x+y*y) <
sqrt((double)RAND_MAX*RAND_MAX))
      sum++;}

  return 4*(float)sum/(float)n;}
```

```c
/* filename: pi.h*/
float pi_approx(int n);
```



CFFI: calling C sources (out-of-line API mode)

```c
/* filename: pi.h*/
float pi_approx(int n);
```

```c
/* filename: pi.c*/
# include <stdlib.h>
# include <math.h>

/* Returns a very crude
approximation of Pi
   given a int: a number of
iteration */
float pi_approx(int n){

  double i,x,y,sum=0;

  for(i=0;i<n;i++){
    x=rand();
    y=rand();
    if (sqrt(x*x+y*y) <
sqrt((double)RAND_MAX*RAND_MAX))
      sum++;}

  return 4*(float)sum/(float)n;}
```

```python
from cffi import FFI
ffi = FFI()

ffi.cdef(
    "float pi_approx(int n);")

ffi.set_source("_pi",
r"""
    #include "pi.h"
""",
sources=['pi.c'])

ffi.compile(verbose=True)
```

Create an FFI class instance

CFFI: calling C sources (out-of-line API mode)

```c
/* filename: pi.h*/
float pi_approx(int n);
```

```c
/* filename: pi.c*/
# include <stdlib.h>
# include <math.h>

/* Returns a very crude approximation of Pi
   given a int: a number of iteration */
float pi_approx(int n){

  double i,x,y,sum=0;

  for(i=0;i<n;i++){
    x=rand();
    y=rand();
    if (sqrt(x*x+y*y) < sqrt((double)RAND_MAX*RAND_MAX))
      sum++;}

  return 4*(float)sum/(float)n;}
```

```python
from cffi import FFI
ffi = FFI()

ffi.cdef(
    "float pi_approx(int n);")

ffi.set_source("_pi",
r"""
    #include "pi.h"
""",
sources=['pi.c'])

ffi.compile(verbose=True)
```

CFFI

This is where the C header content goes

ffi.cdef():

- Declare function signatures, types, constants & global variables, like in a C header file
- Registers the C code that you want to use in Python

CFFI: calling C sources (out-of-line API mode)

```c
/* filename: pi.h*/
float pi_approx(int n);
```

```c
/* filename: pi.c*/
# include <stdlib.h>
# include <math.h>

/* Returns a very crude
approximation of Pi
   given a int: a number of
iteration */
float pi_approx(int n){

  double i,x,y,sum=0;

  for(i=0;i<n;i++){
    x=rand();
    y=rand();
    if (sqrt(x*x+y*y) <
sqrt((double)RAND_MAX*RAND_MAX))
      sum++;}

  return 4*(float)sum/(float)n;}
```

```python
from cffi import FFI
ffi = FFI()

ffi.cdef(
    "float pi_approx(int n);")

ffi.set_source("_pi",
r"""
    #include "pi.h"
""",
sources=['pi.c'])

ffi.compile(verbose=True)
```

CFFI

- Name of the C extension module
- C source code that will be pasted in the module
- Additional content that gets pasted in the module

ffi.set_source()

- Prepares a new C module (called extension module)
- Use Python raw strings for literal characters e.g. \n

CFFI: calling C sources (out-of-line API mode)

```c
/* filename: pi.h*/
float pi_approx(int n);
```

```c
/* filename: pi.c*/
# include <stdlib.h>
# include <math.h>

/* Returns a very crude
approximation of Pi
    given a int: a number of
iteration */
float pi_approx(int n){

  double i,x,y,sum=0;

  for(i=0;i<n;i++){
    x=rand();
    y=rand();
    if (sqrt(x*x+y*y) <
sqrt((double)RAND_MAX*RAND_MAX))
      sum++;}

  return 4*(float)sum/(float)n;}
```

```python
from cffi import FFI
ffi = FFI()

ffi.cdef(
    "float pi_approx(int n);")

ffi.set_source("_pi",
r"""
    #include "pi.h"
""",
sources=['pi.c'])

ffi.compile(verbose=True)
```

CFFI

Compile C code

ffi.compile()

- Generates the C file _pi.c
- Compiles it to get the C extension module; a binary shared library object with .so/.dll extension

CFFI: calling C sources (out-of-line API mode)

CFFI

```c
/* filename: pi.h*/
float pi_approx(int n);
```

```c
/* filename: pi.c*/
# include <stdlib.h>
# include <math.h>

/* Returns a very crude approximation of Pi
   given a int: a number of iteration */
float pi_approx(int n){

  double i,x,y,sum=0;

  for(i=0;i<n;i++){
    x=rand();
    y=rand();
    if (sqrt(x*x+y*y) < sqrt((double)RAND_MAX*RAND_MAX))
      sum++;}

  return 4*(float)sum/(float)n;}
```

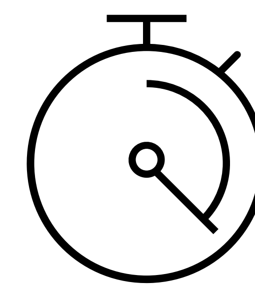
```python
from cffi import FFI
ffi = FFI()

ffi.cdef(
    "float pi_approx(int n);")

ffi.set_source("_pi",
r"""
    #include "pi.h"
""",
sources=['pi.c'])

ffi.compile(verbose=True)
```

```python
from _pi import ffi, lib
a = lib.pi_approx(100)
```

Import the extension module like a Python module

Profiling

Python's timeit module



- Simple timing of small pieces of Python code
- Avoids common pitfalls in measuring execution times
- Command line interface & Python interface

```
$ python3 -m timeit -s "from _pi import ffi, lib" "lib.pi_approx(100)"
200000 loops, best of 5: 1.46 usec per loop
```

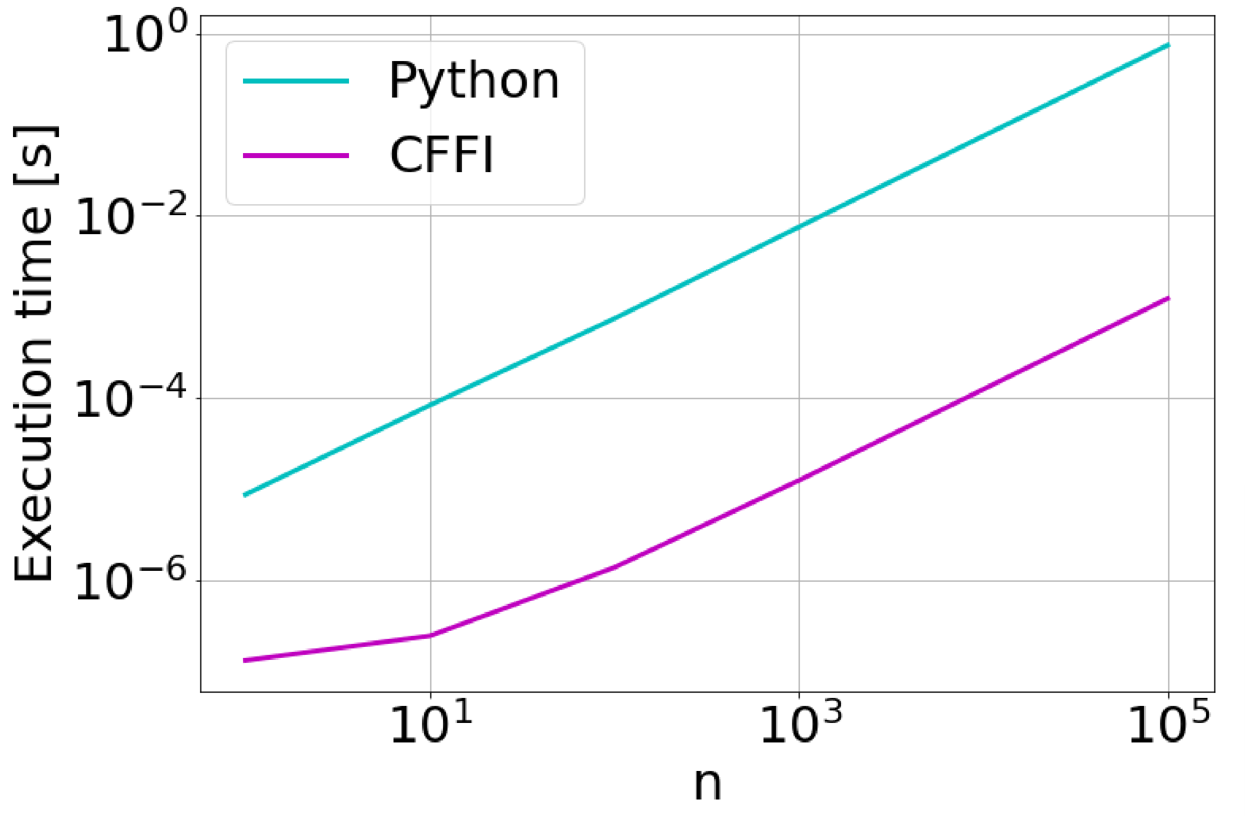
```
>>> import timeit
>>> r = int(1e6)   # repeat this many times
>>> timeit.timeit("lib.pi_approx(100)", setup="from _pi import ffi, lib", number=r)/r
1.5003994679999552e-06
```

- %timeit magic in IPython

```
%timeit lib.pi_approx(100)
1.43 µs ± 41.7 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

CFFI: π approximation benchmark

- CFFI can achieve ~10^2-10^3 typical speedup over standard Python routines



Benchmark on:

- macOS Ventura v13.2
- Intel i7 CPU 2.3 GHz
- clang v14.0.0
- Python v3.9.7
- NumPy v1.23.0
- CFFI v1.15.0

CFFI: raw C kernels (out-of-line API mode)

```python
# file: "cffi_example_raw.py"
import cffi

ffi_interface = cffi.FFI()
ffi_interface.cdef(
    """
    double sumVec(double*, int);
    """
)
ffi_interface.set_source("_sumVec",
r"""
double sumVec(double* v, int n){
  double sum = 0.0;
  for (int i=0; i<n; i++){
    sum += v[i];
  }
  return sum;
}
"""
)
ffi_interface.compile(verbose=True)
```

- You can also write raw C source code as a Python string

Source code that will be pasted in the C extension module

CFFI: raw C kernels (out-of-line API mode)

CFFI

```
from _sumVec import ffi, lib
import numpy as np

x = np.random.rand(10)
x_cffi = ffi.cast( "double *", ffi.from_buffer(x))

y = lib.sumVec(x_cffi, len(x))
```

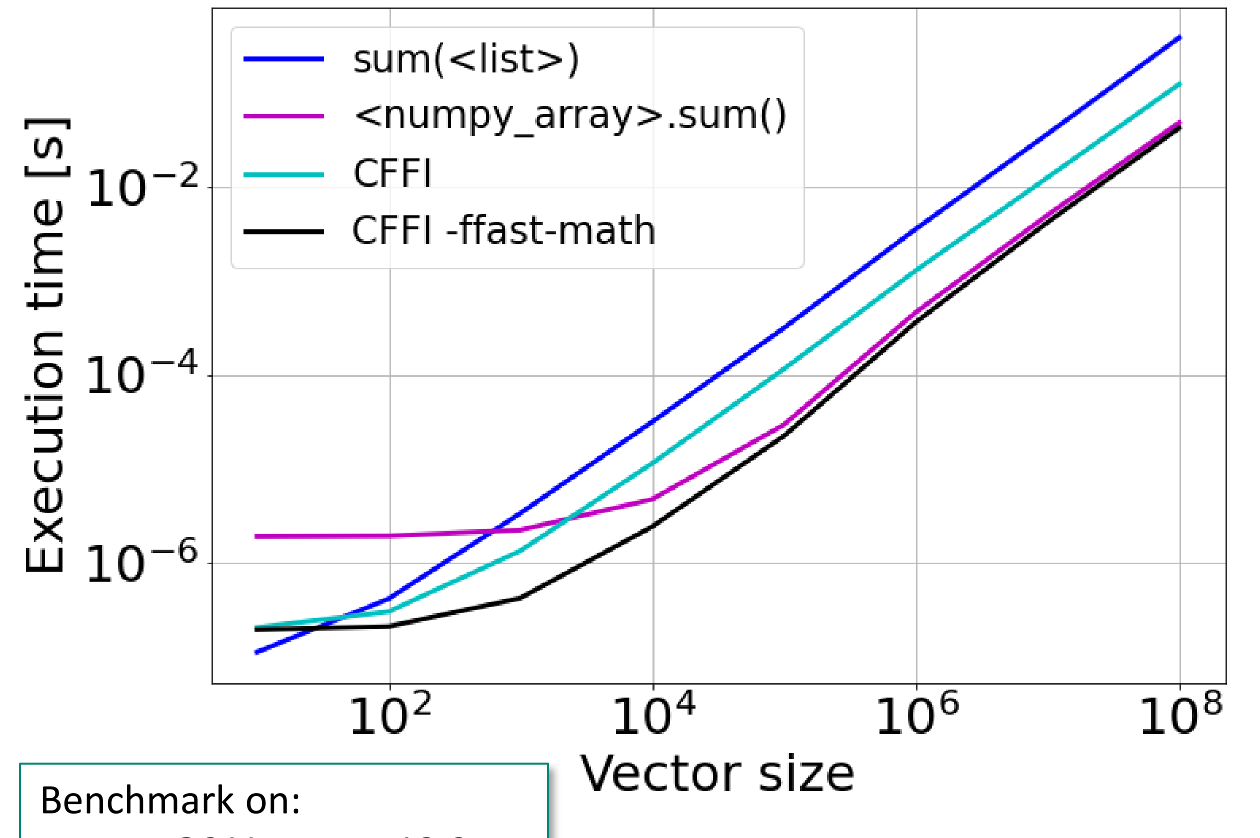
Import the extension module

Create input array & cast into a C array pointer

Call the C function from Python

C	Python
double	float / numpy.float64 / numpy.double
float	numpy.float32
int64_t	int / numpy.int64
int / int32_t	numpy.int32

CFFI: vector sum benchmark



CFFI

- NumPy functions are well optimized
  - Naive C loop becomes slower than NumPy for long vectors
- With some tricks CFFI can still become comparable to numpy.sum():

```
ffi.set_source("_sumVec", r"""...""",
    extra_compile_args=["-ffast-math"])
```

Benchmark on:
- macOS Ventura v13.2
- Intel i7 CPU 2.3 GHz
- clang v14.0.0
- Python v3.9.7
- NumPy v1.23.0
- CFFI v1.15.0

Lecture outline

Heterogeneous Programming Recap

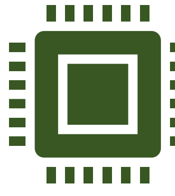
- Motivation
- Hardware accelerators
- Heterogeneous systems

CPU Programming with Python

- Python & C
- CFFI intro

GPU Programming with Python

- CUDA & OpenCL models
- CuPy intro
- PyOpenCL intro



CPU vs GPU design

- CPU: latency-oriented design
  - Large **caches**
  - Advanced **control logic** (e.g. out-of-order execution, branch prediction)
- GPU: throughput-oriented design
  - Chip area maximized for **floating-point calculations** by the Arithmetic Logic Units (ALUs)
  - Data parallelism

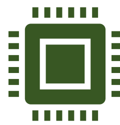
Host

Control
ALU
ALU
ALU
ALU
Cache
DRAM
CPU

DRAM
GPU

(**CUDA** / **OpenCL**) device

Image source: https://commons.wikimedia.org/wiki/File:Cpu-gpu.svg



Parallel programming models

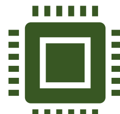
- OpenMP (Open Multi-Processing)
  - Multithreading on (multicore) CPUs
  - Shared memory

- OpenACC (Open ACCelerators)
  - Variation of OpenMP for heterogeneous systems

- MPI (Message Passing Interface)
  - Many-core computing used mainly on CPU clusters
  - No shared memory

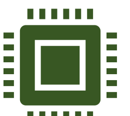
- CUDA (Compute Unified Device Architecture)
  - Most widely used GPU programming model
  - Only compatible with NVIDIA GPUs

- OpenCL (Open Computing Language)
  - Similar to CUDA but not limited to NVIDIA GPUs
  - Supports other platforms, e.g. CPU, FPGA



Overview: CUDA & OpenCL terminology

CUDA	OpenCL
Hardware model	**Platform model**
Streaming multiprocessor (SM)	Compute unit
Streaming processor (SP)	Processing element
Programming model	**Execution model**
Kernel	Kernel
Thread	Work-item
Block	Work-group
Grid	Computation domain (NDRange, grid)
Memory model	**Memory model**
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory



CUDA & OpenCL hardware abstractions

CUDA	OpenCL
Hardware model	Platform model
Streaming multiprocessor (SM)	Compute unit
Streaming processor (SP)	Processing element

Example: NVIDIA GA100 GPU

- 128 **Streaming multiprocessors (SM)** / **Compute units** (≈CPU cores)

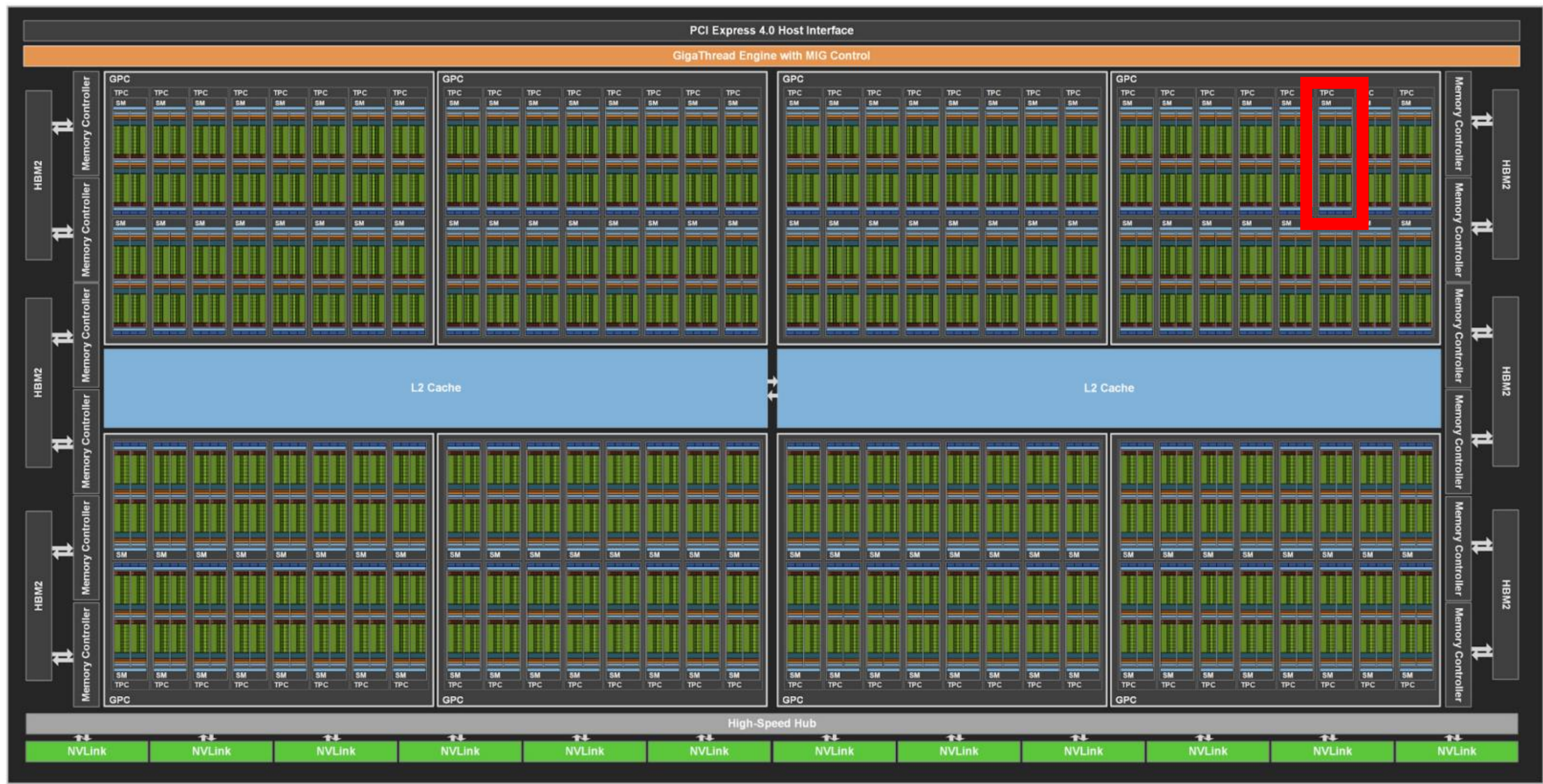
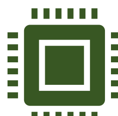


Image source: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf



CUDA & OpenCL hardware abstractions

CUDA	OpenCL
Hardware model	Platform model
Streaming multiprocessor (SM)	Compute unit
Streaming processor (SP)	Processing element

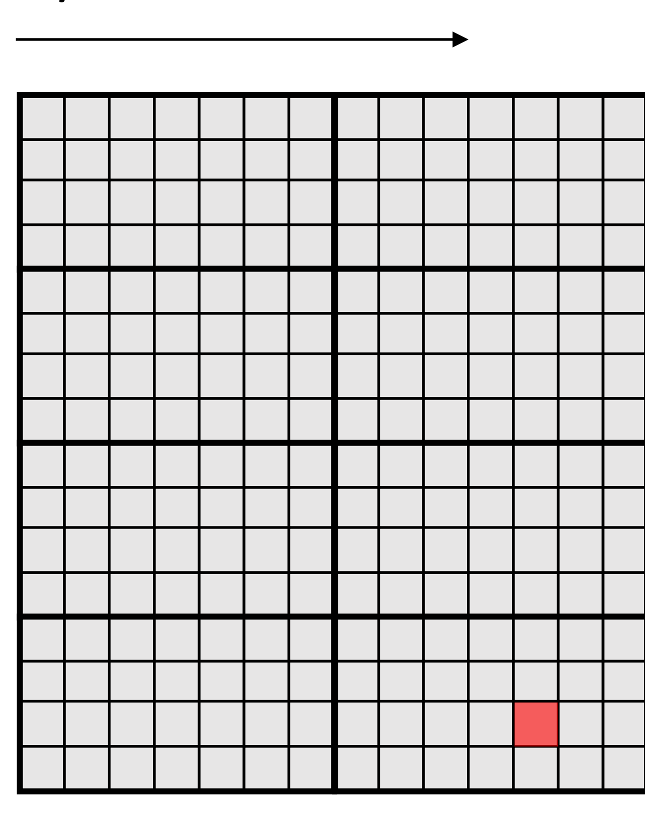
Example: NVIDIA GA100 GPU single SM

- Streaming processors (SP) / processing elements
  - Up to ~10^4 on a full GPU
  - Scalar & tensor “cores”

Image source: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

CUDA & OpenCL programming abstractions

CUDA	OpenCL
Programming model	Execution model
Kernel	Kernel
Thread	Work-item
Block	Work-group
Grid	Computation domain (NDRange, grid)

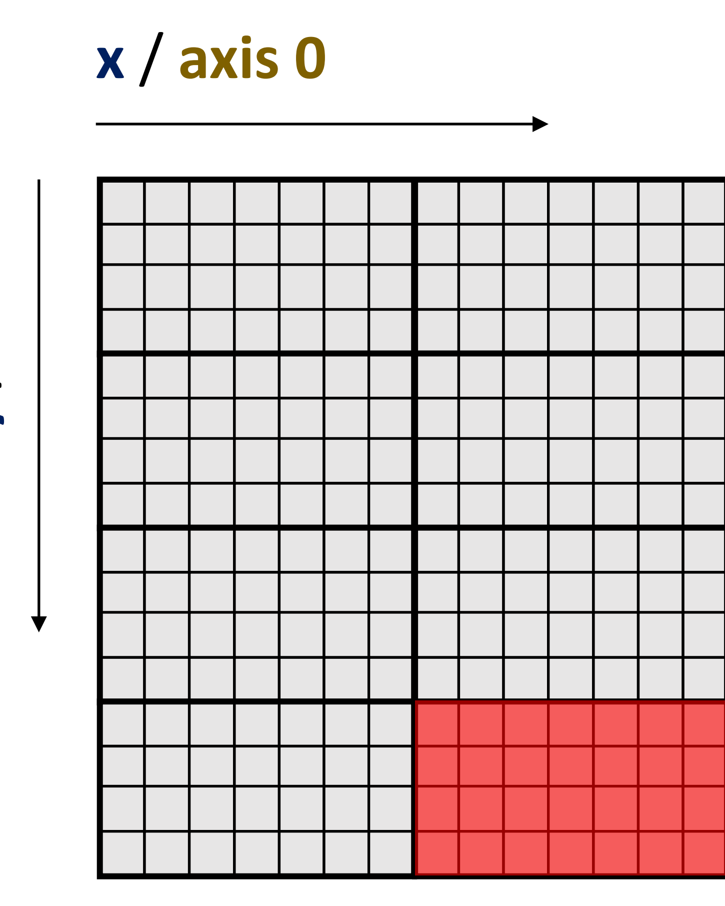


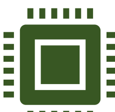
- The GPU can generate **threads** / **work-items**
- Abstraction of the program's sequential execution
- Threads process the same instruction on different data items

CUDA & OpenCL programming abstractions

CUDA	OpenCL
Programming model	**Execution model**
Kernel	Kernel
Thread	Work-item
Block	Work-group
Grid	Computation domain (NDRange, grid)

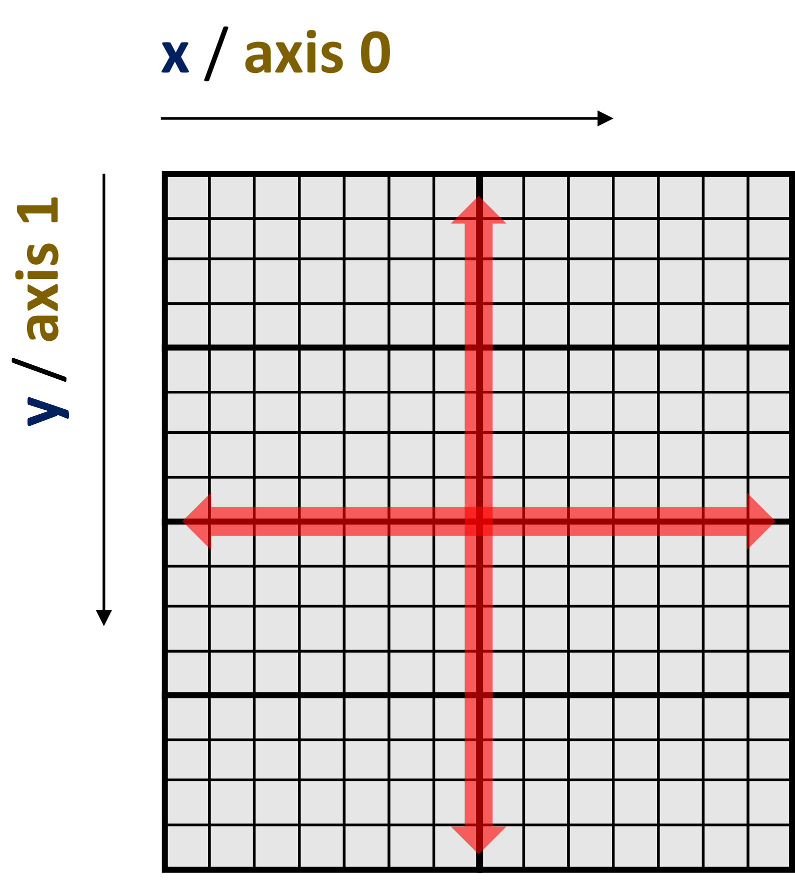
- Threads are grouped into **blocks** / **work-groups**
- Each block gets assigned to an SM
- If #blocks > #SMs: sequential execution in arbitrary order among SMs



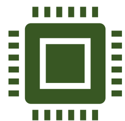


CUDA & OpenCL programming abstractions

CUDA	OpenCL
Programming model	Execution model
Kernel	Kernel
Thread	Work-item
Block	Work-group
Grid	Computation domain (NDRange, grid)

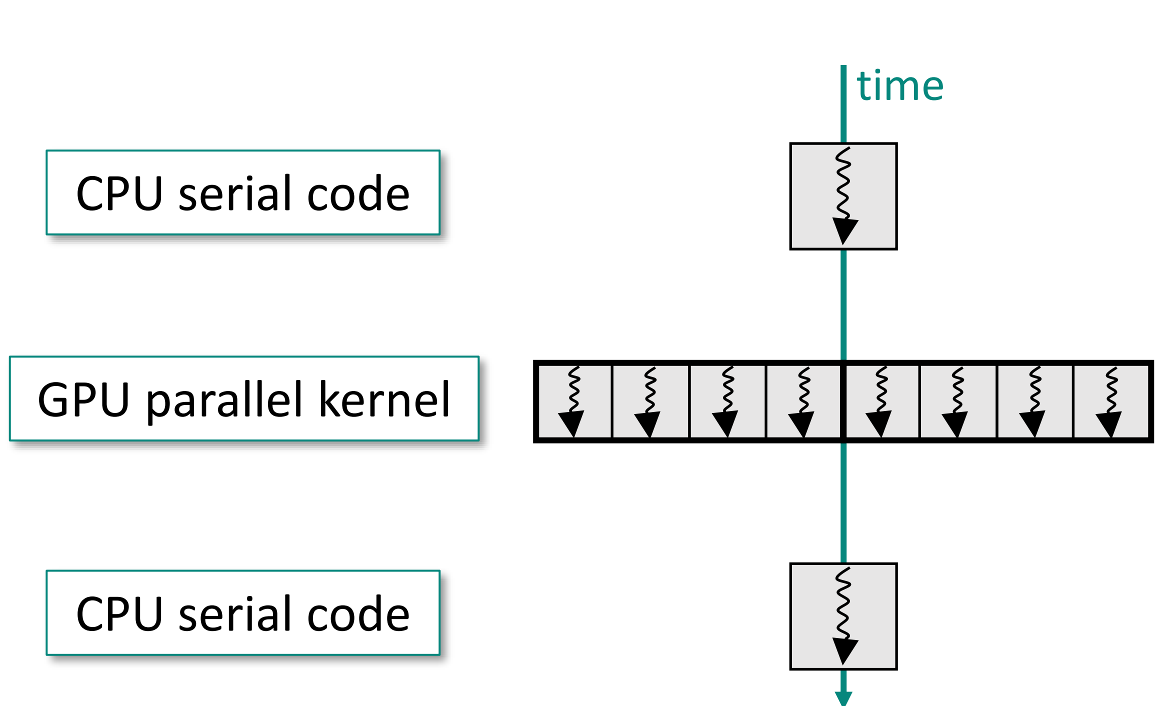


- Blocks make up a **grid** /
  N dimensional index space (NDRange, grid)

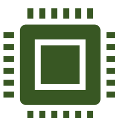


CUDA & OpenCL programming abstractions

- A typical parallel program contains commands to be executed by the host & by the device(s)
- Kernel: code submitted & executed in parallel on a device
  - Remember Amdahl's law




```
void foo(float* A, float* B, float* C, int n){
    // Allocate GPU memory & copy data from CPU to GPU
    <...>;
    // Generate threads / work-items & execute kernel
    <...>;
    // Copy results from GPU to CPU memory & deallocate GPU memory
    <...>;
}
```



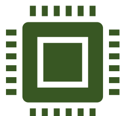
CUDA & OpenCL kernels & thread indexing

```
// Traditional C loop
void myadd(int n,
    const double* x1,
    const double* x2,
    double* y)
{
  for (int i=0; i<n; i++)
  {
    y[i] = x1[i] + x2[i];
  }
}
```

```
// CUDA kernel
__global__ void myadd(int n,
    const double* x1,
    const double* x2,
    double* y)
{
  int i = blockDim.x *
    blockIdx.x + threadIdx.x;
    y[i] = x1[i] + x2[i];
}
```

```
// OpenCL kernel
__kernel void myadd(int n,
    __global const double* x1,
    __global const double* x2,
    __global double* y)
{
  int i = get_global_id(0);
  y[i] = x1[i] + x2[i];
}
```

OpenCL API Call	Explanation	CUDA Equivalent
`get_global_id(0)`	Global index of the work item in the x dimension	`blockIdx.x*blockDim x+threadIdx.x`
`get_local_id(0)`	Local index of the work item within the work group in the x dimension	`threadIdx.x`
`get_global_size(0)`	Size of NDRange in the x dimension	`gridDim.x*blockDim.x`
`get_local_size(0)`	Size of each work group in the x dimension	`blockDim.x`



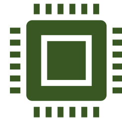
CUDA & OpenCL memory abstractions

CUDA memory model	OpenCL memory model	Function & scope	Host access	Device access
Global memory	Global memory	Per-grid communication	Read-write	Read-write
Constant memory	Constant memory	Per-grid kernel constants	Read-write	Read
Shared memory	Local memory	Per-block communication	-	Read-write
Local memory	Private memory	Per-thread private variables	-	Read-write

© Peter Kicsiny | Multiplatform Programming with Python @ inverted CERN School of Computing

CuPy

- CuPy is based on the CUDA libraries
- NumPy/SciPy compatible array library for GPU accelerated computing with Python
  - Covers most of the NumPy/SciPy API
  - Basically replace numpy.foo() by cupy.foo()
- Allows to run NumPy/SciPy on NVIDIA CUDA devices (more recently also on some AMD platforms)



CuPy: current device

- Default GPU on which the allocation, manipulation & calculation of arrays takes place

- Declare an array on the host:

```
import numpy as np
x_cpu = np.array([1, 2, 3])
```

- Or on the current device:

```
import cupy as cp
x_gpu = cp.array([1, 2, 3])
```

- Or let CuPy decide:

```
xp = cp.get_array_module(*args)
y_xp = xp.dot(*args)
```

  - If at least one argument is of type cupy.ndarray return the module CuPy else NumPy

CuPy: data transfer

- Transfer data between host & current device:

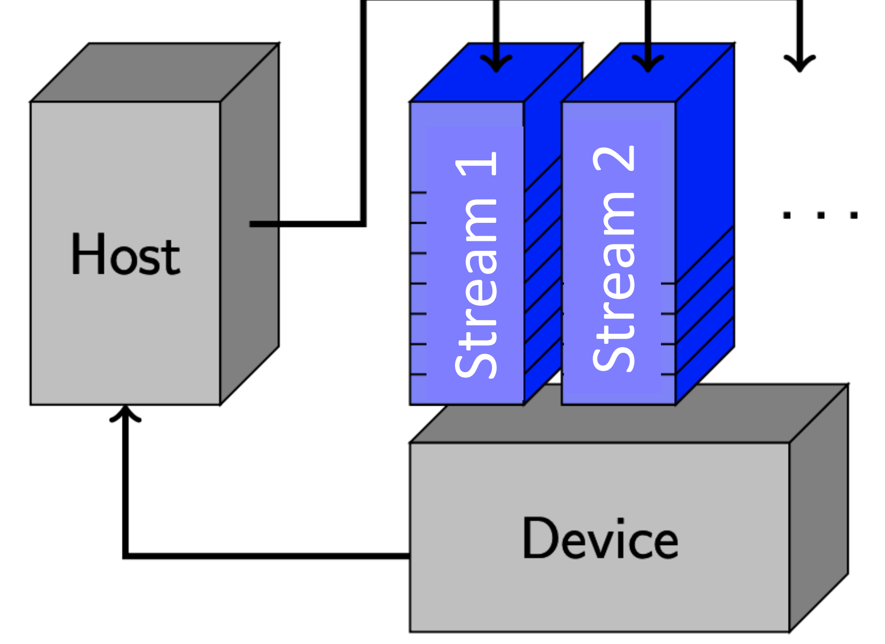
```
x_cpu = np.array([1, 2, 3])
x_gpu = cp.array(x_cpu)  # copy the data to the current device
y_cpu = x_gpu.get() # copy the data back to the host
```

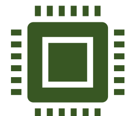
- Transfer data between devices:

```
with cp.cuda.Device(0):
    x_gpu_0 = cp.array([1, 2, 3])  # create an array on GPU 0
    ...
with cp.cuda.Device(1):
    x_gpu_1 = cp.array(x_gpu_0)  # copy the array to GPU 1
    ...
```

CuPy: streams

- An entry point to a compute device
- Current stream
  - All CUDA operations (e.g. data copies & kernels) are enqueued onto the current stream on the device
- The queued operations on a stream will be executed sequentially but **asynchronously** to the host
  - The control is given back to the Python interpreter immediately, while the GPU is still executing the operations





CuPy: example using only the API

```python
# file: "cupy_example_api.py"
import numpy as np
import cupy as cp

x1 = np.array([1, 3, 5, 7, 9], dtype=np.float64)
x2 = np.array([2, 4, 6, 8, 10], dtype=np.float64)

x1_dev = cp.array(x1)
x2_dev = cp.array(x2)

y_dev = x1_dev * x2_dev
print(y_dev, type(y_dev), y_dev.dtype)

y = y_dev.get()
print(y, type(y), y.dtype)
```

Create input arrays on host

Copy data to the current device

Do the multiplication on the device

```
[ 2. 12. 30. 56. 90.] <class 'cupy.ndarray'> float64
```

Copy result back to host

```
[ 2. 12. 30. 56. 90.] <class 'numpy.ndarray'> float64
```

CuPy

CuPy: example with raw kernel #1

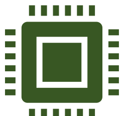
```
# file: "cupy_example_raw.py"
import numpy as np
import cupy as cp

x1 = np.array([1, 3, 5, 7, 9], dtype=np.float64)
x2 = np.array([2, 4, 6, 8, 10], dtype=np.float64)

x1_dev = cp.array(x1)
x2_dev = cp.array(x2)

y_dev = cp.zeros_like(x1_dev)

source_str = r"""
extern "C"{
__global__
void mymul(int n, const double* x1, const double* x2, double* y){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < n){
      y[tid] = x1[tid] * x2[tid];
    }
}}"""
```

So far the same

Initialize an output buffer on the device

Define CUDA kernel

CuPy



CuPy: example with raw kernel #2

Create a module from the CUDA source

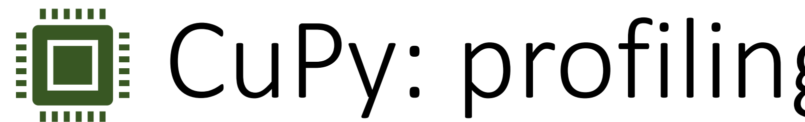
```
module = cp.RawModule(code=source_str)
mymul_kernel = module.get_function("mymul")

blocksize = 2
n_blocks = int(np.ceil(len(x1) / blocksize))
mymul_kernel(grid=(n_blocks,), block=(blocksize,),
args=(np.int32(len(x1)), x1_dev, x2_dev, y_dev))
print(y_dev, type(y_dev), y_dev.dtype)

y = y_dev.get()
print(y, type(y), y.dtype)
```

Compilation via CuPy's compiler at the first function call

Specify grid & block size & send kernel to the current stream

```
[ 2. 12. 30. 56. 90.] <class 'cupy.ndarray'> float64
```

```
[ 2. 12. 30. 56. 90.] <class 'numpy.ndarray'> float64
```



CuPy: profiling

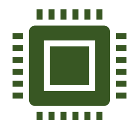
- cupyx.profiler.benchmark records the GPU time by synchronizing streams

```
from cupyx.profiler import benchmark

def mul(n_blocks, blocksize, n, a, b, c):
    mymul_kernel(grid=(n_blocks,), block=(blocksize,), args=(n, a, b, c))

print(benchmark(mul, (n_blocks, blocksize, np.int32(len(x1)), x1_dev,
x2_dev, y_dev), n_repeat=100))
```

Time spent on host

```
mul: CPU: 9.690 us +/- 2.269 (min: 8.040 / max: 24.411) us
GPU-0: 6403.901 us +/-253.549 (min: 6223.872 / max: 7040.000) us
```

Time spent on device executing the kernel (excluding memory transfer & compilation time)



CuPy: benchmarks against NumPy

CuPy speedup over NumPy (Quoted from RAPIDS AI)

(10000x10000) Array Size 800 MB

(1000x1000) Array Size 8 MB

Speedup

500
200
100
50
20
10
5
2
1

Elementwise
FFT
Array Slicing
Stencil
Sum
Matrix Multiplication
SVD
Standard Deviation

Image source: https://github.com/pentschev/pybench

- Benchmark of basic array operations
  - CPU: Intel Xeon E5–2698 v4 2.20GHz
  - GPU: NVIDIA Tesla V100 32 GB
- CuPy performs well out of the box
- With CuPy API: no need to write any parallel code by hand

PyOpenCL

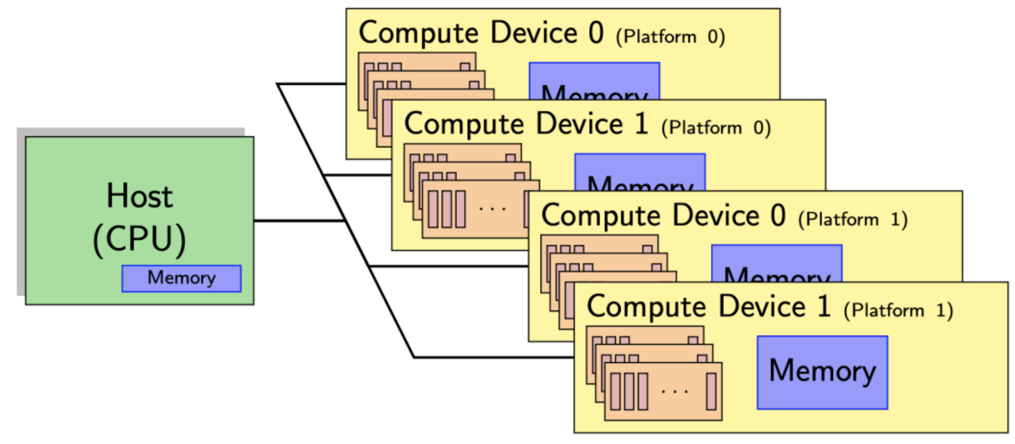
- PyOpenCL is based on the OpenCL libraries

- Platform independent
  - Specific OpenCL implementation for each
- pyopencl.array, pyopencl.clmath: (limited) NumPy-like API for array manipulations

PyOpenCL: platforms

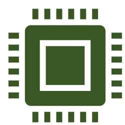
- Collection of devices from the same vendor
  - e.g. Platform 0: AMD CPUs & GPUs, Platform 1: NVIDIA GPUs

- All devices on a platform use the same OpenCL implementation

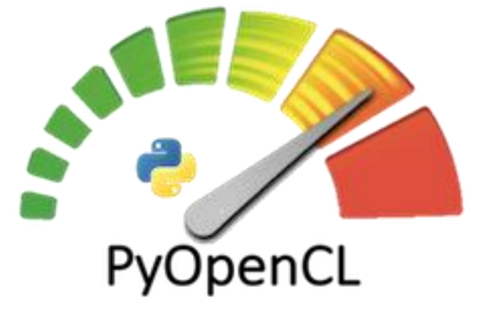


- Get list of available platforms:

```
import pyopencl as cl
platforms_list = cl.get_platforms()
```

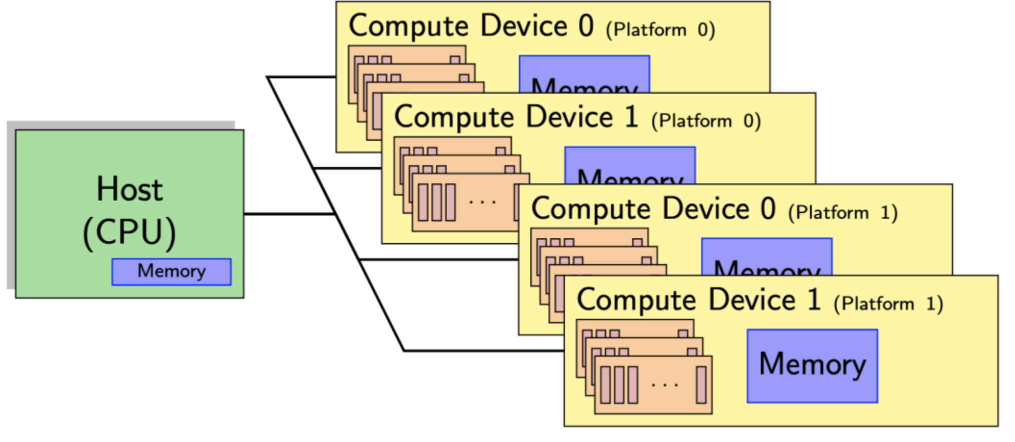


PyOpenCL: compute devices



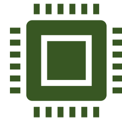
- A processor die with an interface to off-chip memory
  - CPU, GPU, other accelerators
  - They do the parallel computation

- Unlike CuPy, PyOpenCL has no default current device



- Get list of devices within a platform:

```
devices_list = platforms_list[0].get_devices()
```



PyOpenCL: contexts

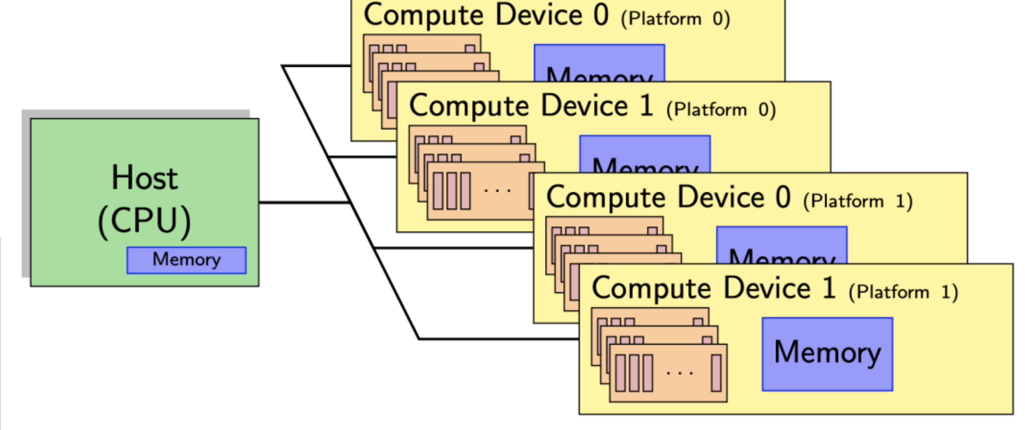
- The living space of all OpenCL objects (commands, kernels) & command queues

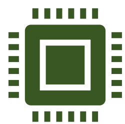
- Can be attached to 1 or more devices
- Unlike in CuPy, it must be declared manually:

```
import pyopencl as cl
platforms_list = cl.get_platforms()
devices_list = platforms_list[0].get_devices()
context = cl.Context(devices=devices_list)
```

or

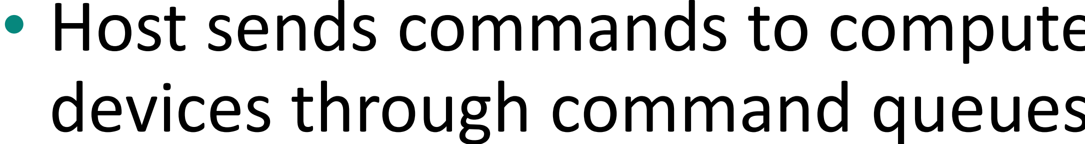
```
context = cl.create_some_context(interactive=True)
```





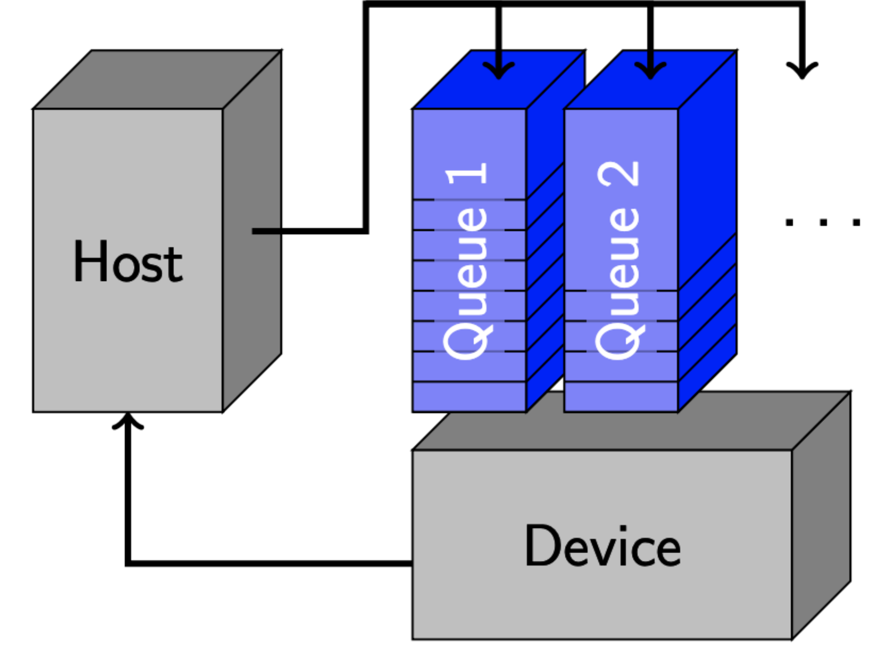
PyOpenCL: command queues

- Analog to CuPy streams



- Host sends commands to compute devices through command queues
- Sequential execution, no overlapping
- Unlike in CuPy, it must be set manually:

```
queue = cl.CommandQueue(context)
```



PyOpenCL: example using only the API

```
# file: "pyopencl_example_api.py"
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array

ctx = cl.create_some_context(interactive=False)
queue = cl.CommandQueue(ctx)

x1 = np.array([1, 3, 5, 7, 9], dtype=np.float64)
x2 = np.array([2, 4, 6, 8, 10], dtype=np.float64)

x1_dev = cl_array.to_device(queue, x1)
x2_dev = cl_array.to_device(queue, x2)

y_dev = x1_dev * x2_dev
print(y_dev, type(y_dev), y_dev.dtype)

y = y_dev.get()
print(y, type(y), y.dtype)
```

Works like NumPy arrays

PyOpenCL

Create context & command queue

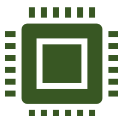
Copy data to the device queue

Do the multiplication on the device

```
[ 2. 12. 30. 56. 90.] <class 'pyopencl.array.Array'> float64
```

Copy result back to host

```
[ 2. 12. 30. 56. 90.] <class 'numpy.ndarray'> float64
```

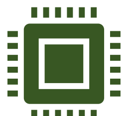


PyOpenCL: example with raw kernel #1

```python
# file: "pyopencl_example_raw.py"
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array

ctx = cl.create_some_context(interactive=False)
queue = cl.CommandQueue(ctx)

x1 = np.array([1, 3, 5, 7, 9], dtype=np.float64)
x2 = np.array([2, 4, 6, 8, 10], dtype=np.float64)

x1_dev = cl_array.to_device(queue, x1)
x2_dev = cl_array.to_device(queue, x2)
y_dev = cl_array.zeros(queue, len(x1_dev), dtype=np.float64)
```

So far the same

+output buffer



PyOpenCL: example with raw kernel #2

```
source_str = r"""
__kernel
void mymul(int n,
    __global const double* x1,
    __global const double* x2,
    __global double* y)
{
    int tid = get_global_id(0);
    y[tid] = x1[tid] * x2[tid];
}"""

prg = cl.Program(ctx, source_str).build()
mymul_kernel = prg.mymul
mymul_kernel(queue, (len(x1),), None, np.int32(len(x1)),
x1_dev.data, x2_dev.data, y_dev.data)
print(y_dev, type(y_dev), y_dev.dtype)

y = y_dev.get()
print(y, type(y), y.dtype)
```

Define OpenCL kernel

Compile with PyOpenCL's compiler

Specify grid & work-group size & send kernel to the queue

```
[ 2. 12. 30. 56. 90.] <class 'pyopencl.array.Array'> float64
```

```
[ 2. 12. 30. 56. 90.] <class 'numpy.ndarray'> float64
```

PyOpenCL

PyOpenCL: profiling

- Enable profiling when creating the command queue:

```
queue = cl.CommandQueue(ctx,
        properties=cl.command_queue_properties.PROFILING_ENABLE)
```

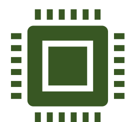
- Every enqueue operation returns an event object
  - Encapsulates the status of an operation

```
event = prg.mul(queue, (grid_size,), (workgroup_size,), a_buffer, b_buffer, c_buffer)
```

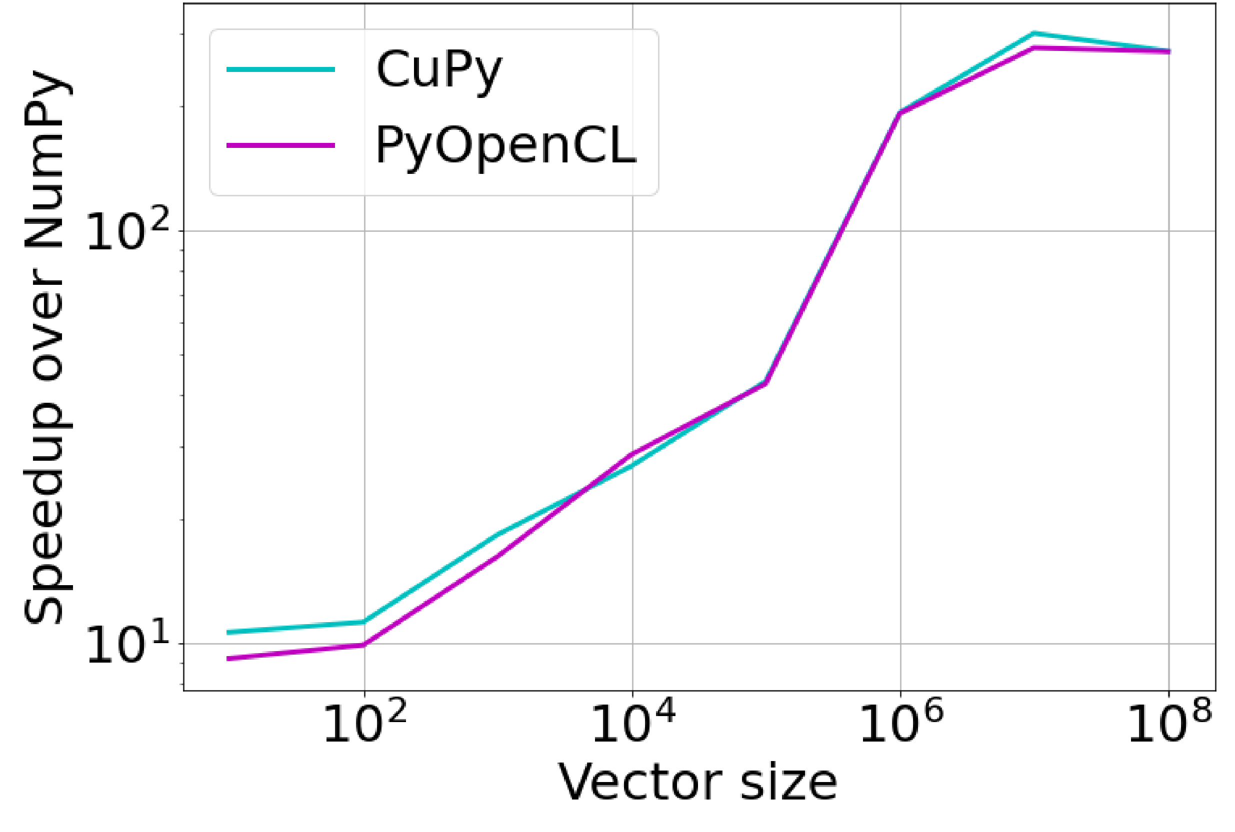
- Synchronize with event.wait():

```
event.wait()
elapsed = (event.profile.end - event.profile.start)*1e-3  # convert from [ns] to [us]
print(f"GPU kernel time: {int(elapsed)} us")
```

```
GPU kernel time: 6316 us
```



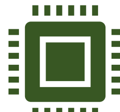
CuPy - PyOpenCL benchmark against NumPy



- Elementwise vector multiplication numpy.multiply(x,y)
- CuPy & PyOpenCL have comparable speedup

Benchmark on:

- Ubuntu 22.04.1 LTS
- CPU: AMD Ryzen 1950X 3.40 GHz
- GPU: NVIDIA TITAN V 12GB
- Python v3.10.6
- NumPy v1.21.5
- cupy-cuda11x v11.5.0
- PyOpenCL v2022.3.1



CuPy - PyOpenCL code similarity

```python
# file: "cupy_example_api.py"
import numpy as np
import cupy as cp

x1 = np.array([1, 3, 5, 7, 9], dtype=np.float64)
x2 = np.array([2, 4, 6, 8, 10], dtype=np.float64)

x1_dev = cp.array(x1)
x2_dev = cp.array(x2)

y_dev = x1_dev * x2_dev
print(y_dev, type(y_dev), y_dev.dtype)

y = y_dev.get()
print(y, type(y), y.dtype)
```

```python
# file: "pyopencl_example_api.py"
import numpy as np
import pyopencl as cl
import pyopencl.array as cl_array

ctx = cl.create_some_context(interactive=False)
queue = cl.CommandQueue(ctx)

x1 = np.array([1, 3, 5, 7, 9], dtype=np.float64)
x2 = np.array([2, 4, 6, 8, 10], dtype=np.float64)

x1_dev = cl_array.to_device(queue, x1)
x2_dev = cl_array.to_device(queue, x2)

y_dev = x1_dev * x2_dev
print(y_dev, type(y_dev), y_dev.dtype)

y = y_dev.get()
print(y, type(y), y.dtype)
```

Key takeaways

1. The future of HPC is heterogeneous & GPGPUs with fast double precision computing are getting more popular
2. Python is a high-level scripting language complementary to low-level languages used for performance critical code
3. CFFI helps to link custom C code to Python to boost performance
4. CuPy & PyOpenCL help to program GPUs in Python in a very similar & fairly simple way

Tutorial

https://github.com/pkicsiny/icsc23_course

References & resources

- Programming Massively Parallel Processors, **David B. Kirk, Wen-mei W. Hwu**
  https://safari.ethz.ch/architecture/fall2019/lib/exe/fetch.php?media=2013_programming_massively_parallel_processors_a_hands-on_approach_2nd.pdf
- CUDA Programming guide
  https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- OpenCL API Specification
  https://registry.khronos.org/OpenCL/
- Easy, Effective, Efficient GPU Programming with PyOpenCL and PyCUDA, **Andreas Klöckner**
  https://www.bu.edu/pasi/courses/gpu-programming-with-pyopencl-and-pycuda/
- CFFI documentation
  https://cffi.readthedocs.io/en/latest/index.html
- CuPy documentation
  https://docs.cupy.dev/en/stable/index.html
- PyOpenCL documentation
  https://documen.tician.de/pyopencl/index.html
- Tool used to create code snippet images for the lecture
  https://carbon.now.sh/