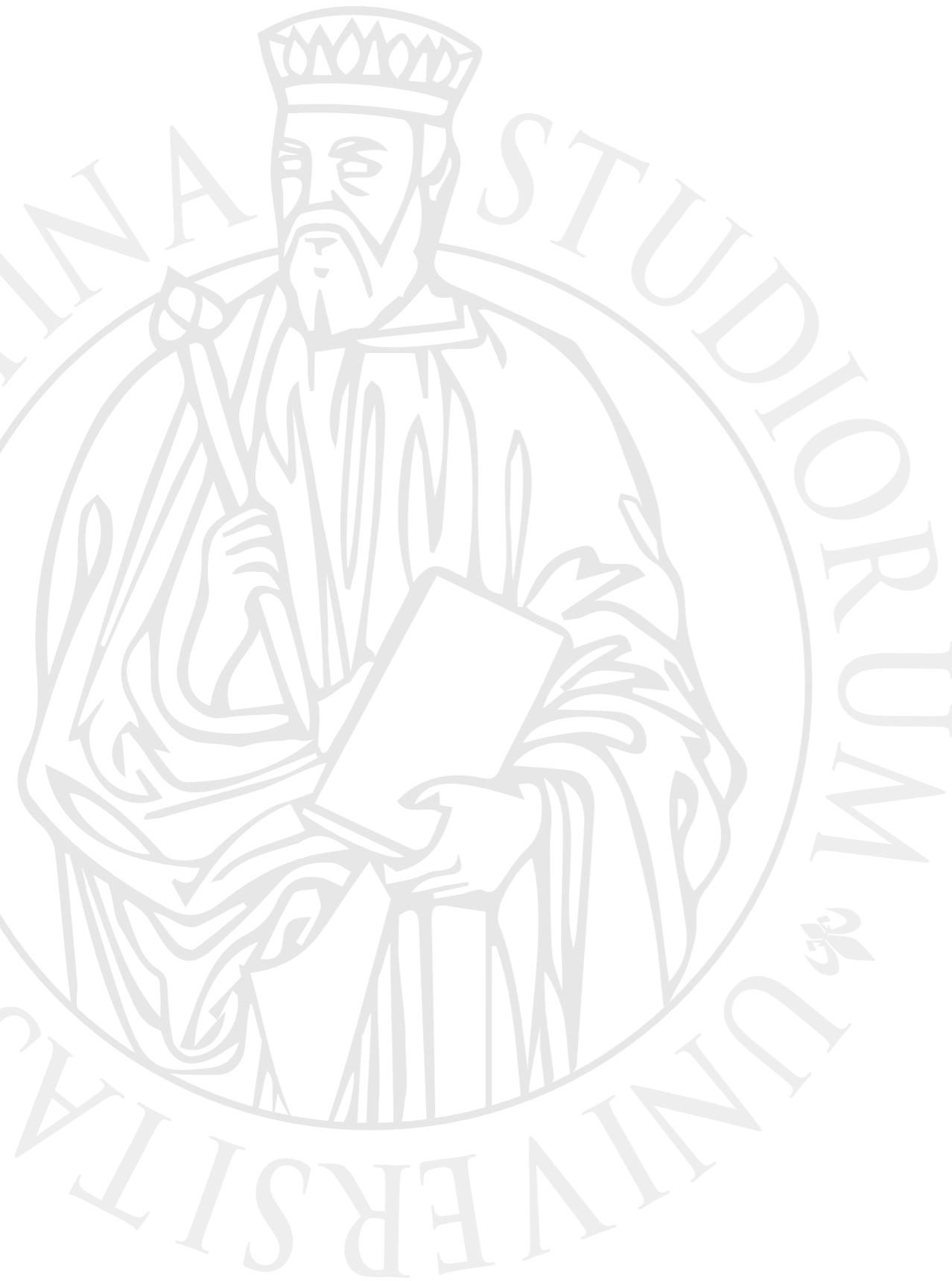




UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

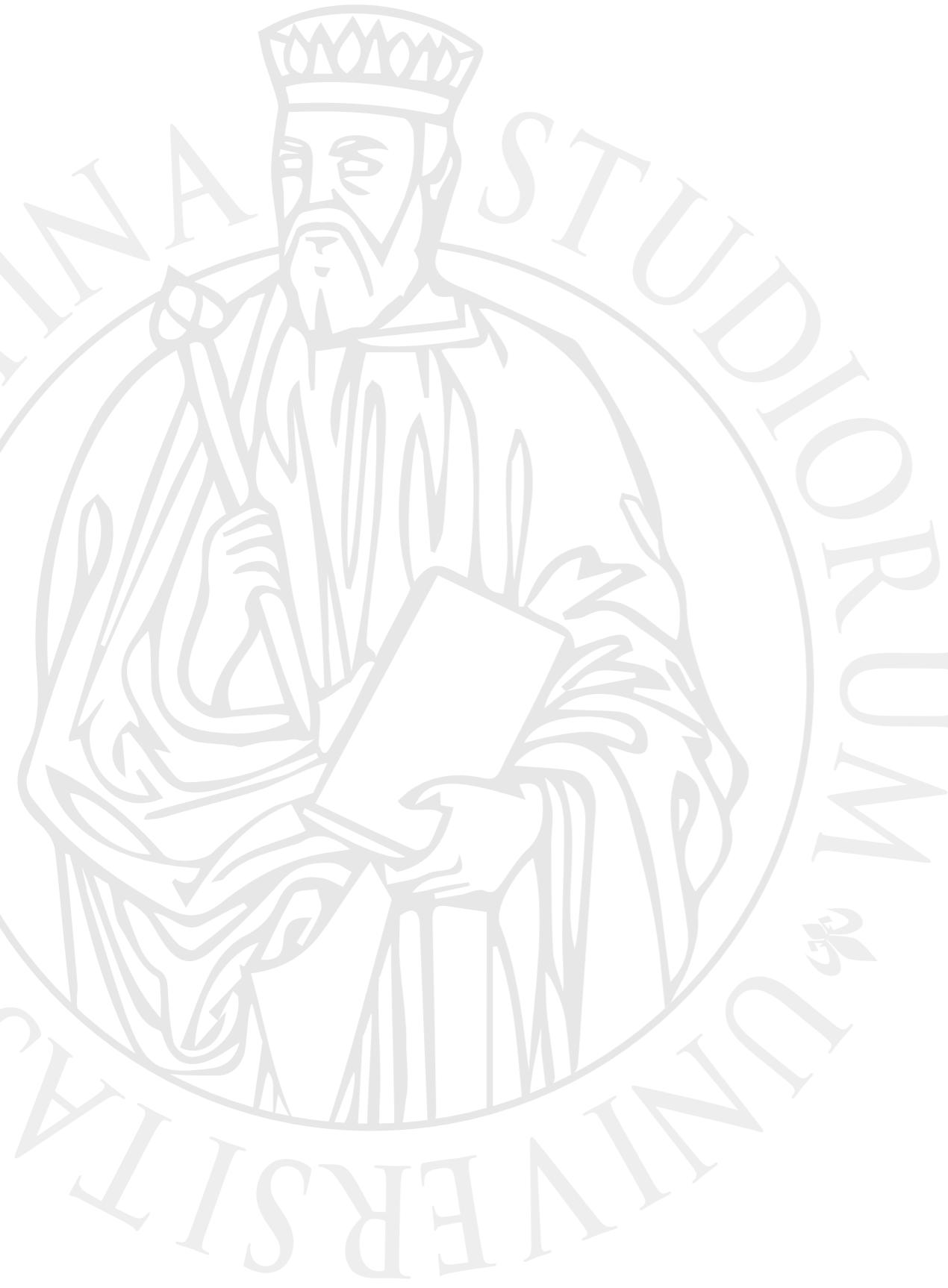


# Parallel Programming

Prof. Marco Bertini



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE



# Python: GPU programming



# Python and CUDA libraries

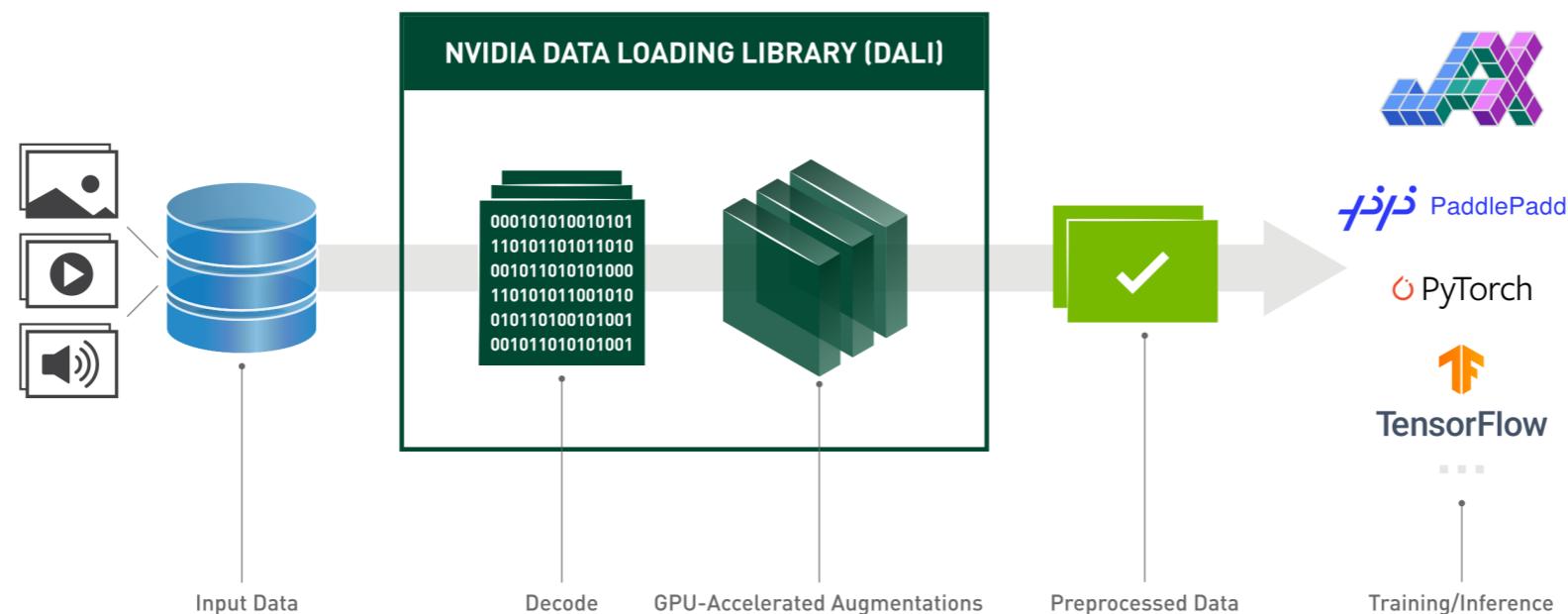


# CUDA enabled libraries for Python

- Nvidia provides many CUDA-enabled libraries with Python bindings/interfaces. Check RAPIDS.ai
  - cuDF: Python drop-in Pandas replacement built on CUDA.
  - cuML: GPU-acceleration of popular ML algorithms e.g. XGBoost, scikit-learn like interface
  - cuVS: algorithms for approximate nearest neighbors and clustering on the GPU.

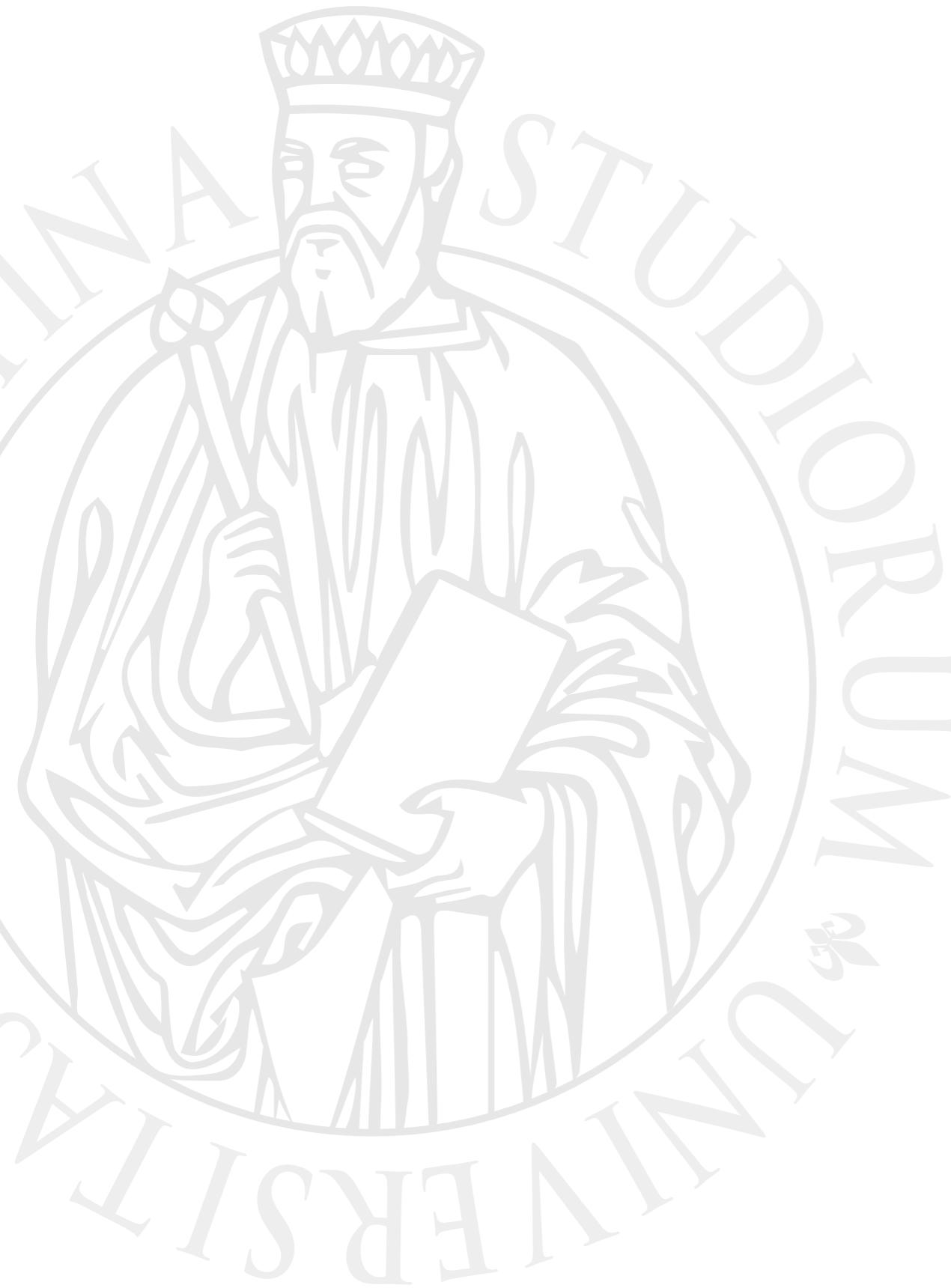
# Example: data loading

- The NVIDIA Data Loading Library (DALI) is a GPU-accelerated library for data loading and pre-processing to accelerate deep learning applications.
- It provides a collection of highly optimized building blocks for loading and processing image, video and audio data.
- It can be used as a portable drop-in replacement for built in data loaders and data iterators in popular deep learning frameworks.





UNIVERSITÀ  
DEGLI STUDI  
FIRENZE



# Python and CUDA programming

# PyCUDA vs. Numba

- PyCUDA is a Python interface to CUDA
  - Write CUDA code within your Python code
  - Low-level access and fine-grained control
  - Manages the memory, e.g. copying the Python data to/from device
  - Translates CUDA errors to Python exceptions
- Numba is a just-in-time compiler for Python that works on CPUs and GPUs
  - Best on code that uses NumPy arrays and functions, and loops.
  - Can work on NumPy *ufuncs*, i.e. element-wise operations

# PyCUDA Overview

- PyCUDA basics:
  - Write CUDA kernels in C-like language as strings.
  - Use `pycuda.driver` to allocate memory and launch kernels.
  - Use `pycuda.autoinit` for quick setup.
- Pros:
  - Very close to raw CUDA C (full control).
- Cons:
  - Need to write kernels in CUDA C, following CUDA syntax.



# Why PyCUDA ?

- Fine-grained control over memory management.
- Easy to call existing CUDA libraries from Python.
- Provides access to low-level CUDA features like streams, events, and memory allocation.



# Numba overview

- Numba basic:
  - `@cuda.jit` decorator that compiles a Python function to a CUDA kernel.
- Pros:
  - Write kernels in Python syntax.
  - Good integration with NumPy arrays.
- Cons:
  - Subset of Python supported inside kernels.
  - Some CUDA features less exposed than PyCUDA.



# Why Numba ?



- Easy to use with minimal boilerplate code.
- Supports automatic vectorization and parallelization on the CPU.
- CUDA support for writing GPU kernels with Python.
- Provides additional decorators that simplify development like `@reduce` for reduction



# Numba-cuda

- CUDA is now supported by Numba **in a separate package**: numba-cuda  
`pip install numba-cuda`  
`conda install conda-forge::numba-cuda`
- Version 0.6.1 supports it, 0.6.2 continues to provide it.
- Current Numba is 0.6.3





# PyCUDA vs. Numba

```
import pycuda.autoinit
import pycuda.driver as cuda
import numpy as np
from pycuda.compiler import SourceModule

# Compile the CUDA kernel code
mod = SourceModule("""
__global__ void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        y[i] += a * x[i];
    }
}
""")
saxpy_cuda = mod.get_function("saxpy") # Get the function pointer for the compiled kernel

# Initialize host data
a = 3.141
x = np.random.rand(1024 * 2048).astype(np.float32)
y = np.random.rand(1024 * 2048).astype(np.float32)

# Allocate memory for x and y on the GPU
d_x = cuda.mem_alloc(x.nbytes)
d_y = cuda.mem_alloc(y.nbytes)
# Copy data from CPU to GPU
cuda.memcpy_htod(d_x, x)
cuda.memcpy_htod(d_y, y)

# Kernel execution configuration
block_dim = (256, 1, 1) # Number of threads per block (1D block)
grid_dim = ((x.size + block_dim[0] - 1) // block_dim[0], 1) # Number of blocks needed

# Launch the kernel: (n, a, d_x, d_y)
saxpy_cuda(np.int32(x.size), np.float32(a), d_x, d_y, block=block_dim, grid=grid_dim)

# Copy the result back to CPU
cuda.memcpy_dtoh(y, d_y)
# Free GPU memory
d_x.free()
d_y.free()
```

PyCUDA saxpy

# PyCUDA vs. Numba

```
import numpy as np
from numba import vectorize
```

```
@vectorize([float32(float32, float32, float32)],
           target='cuda')
def saxpy(a, x, y):
    return a * x + y
```

Numba saxpy

```
a = 3.141
x = np.random.rand(1024, 2048).astype(np.float32)
y = np.random.rand(1024, 2048).astype(np.float32)
```

```
result = saxpy(a, x, y)
```



# PyCUDA

- pip install pycuda
- import pycuda.driver as cuda  
cuda.init()
- Or
- import pycuda.driver as cuda  
import pycuda.autoinit #  
    Automatically initializes  
    CUDA driver

# PyCUDA - memory

- Memory Allocation
  - `cuda.mem_alloc(size)`: Allocates memory on the GPU.
- Memory Transfer
  - `cuda.memcpy_htod(dst, src)`: Host to Device transfer.
  - `cuda.memcpy_dtoh(dst, src)`: Device to Host transfer.
- Synchronization
  - Ensure that memory operations are complete before moving to the next step.



# Numba

- pip install numba
- from numba import cuda
- Numba's `@jit` decorator can accelerate operations on the CPU without any GPU code.
- `@jit(nopython=True)` ensures that Numba compiles the function to machine code for maximum performance.
  - Automatically vectorizes the code without manually optimizing loops.
  - It's possible to even release the GIL, if the code doesn't work on Python objects
- Use the `@cuda.jit` decorator to compile a kernel for the GPU.
- Use `@vectorize` to vectorize CPU code, add `target='cuda'` to create GPU kernels. It creates ufunc functions (that operate element by element on whole arrays), the code seems to operate on scalars, Numba creates the loop (CPU) or CUDA kernel (GPU).



# Numba - memory

- Memory Management 
  - `cuda.to_device()` for sending data to the GPU.
  - `cuda.device_array()` for creating device arrays.
  - Use `.copy_to_host()` to copy from GPU array back to host
- Kernel Execution Configuration
  - Define the number of threads per block and number of blocks in a grid.
  - A syntax similar to CUDA... `numba.cuda.blockIdx`, `numba.cuda.threadIdx`, `numba.cuda.blockDim`, `numba.cuda.gridDim`

# PyCUDA vs. Numba - redux

- API and Level of Abstraction
  - PyCUDA: Direct interaction with CUDA APIs. More control over memory, kernel launch, and performance tuning.
  - Numba: Higher-level abstraction, with `@cuda.jit` for CUDA kernels. Less manual setup but also less fine-grained control.
- Ease of Use
  - PyCUDA: Requires understanding of CUDA memory management and kernel compilation.
  - Numba: Simple syntax, Pythonic interface, and automatic optimizations.

# CUDA support

- Numba, despite being high-level, support atomics, shared memory, streams, etc.
- Even though Numba can automatically transfer NumPy arrays to the device, it can only do so conservatively by always transferring device memory back to the host when a kernel finishes.
  - To avoid the unnecessary transfer for read-only arrays, use the explicit memory APIs



# Debugging in Numba

- Numba includes a **CUDA Simulator** that implements most of the semantics in CUDA Python using the Python interpreter and some additional Python code.
- Can be used to debug CUDA Python code, not only by adding print statements, but also by using the debugger to step through the execution of an individual thread.
- Enable the simulator setting the environment variable `NUMBA_ENABLE_CUDASIM` to 1 prior to importing Numba

# Credits

- These slides report material from:
  - NVIDIA GPU Teaching Kit
  - P. Graham, NVIDIA
  - PyCUDA manual
  - NUMBA manual

