>>> Easy GPU computing with jax and cupy

Rico K. L. Lo[†] (Center of Gravity, Niels Bohr Institute)
@ Tycho Science Day 2026

[†]kalok.lo@nbi.ku.dk

>>> Outline

1. Objectives
2. GPUs and General-purpose GPU programming
3. Brief introduction to high-level GPGPU libraries
   - cupy
   - jax
   - pytorch
4. Hands-on demonstrations and exercises
5. Concluding remarks

>>> Objectives

Learn how to

- accelerate your scientific computations with GPUs in Python
  - *in theory:* working principles and techniques
  - *in practice:* using high-level libraries, such as `cupy` and `jax`
- use the GPUs available at the KU HPC cluster

Disclaimers:

- This technical workshop is for ***beginners***
  - assume only basic knowledge in Python and familiarity with `numpy`
  - no prior knowledge on GPU technology is needed
- Only a brief introduction (1.5 hours) and not a technical deep dive
- Not about machine/deep learning

GPUs and General-purpose GPU programming

what, why and how

>>> What is a GPU?

- A graphics processing unit (GPU) was originally designed to accelerate computer graphics rendering (hence its name)
  - by offloading the calculations from the central processing unit (CPU) to a separate piece of hardware, usually with ***separate memory*** modules
  - calculations are often done ***asynchronously*** with the rest of the program
  - CPU and GPU work alongside each other/in tandem
- Popular vendors for GPUs: Nvidia, AMD, Intel, Apple

Figure: Nvidia's GeForce RTX 5090 (image credit: Nvidia)

>>> Advantages of GPUs over CPUs

- Instead of having only a few but powerful and versatile processors, like in the case of a CPU, a GPU has less powerful but ***a lot more*** processing units
  - as an example, a top-of-the-line GPU from Nvidia, RTX 5090, has a total of 21,760 of those cores,
  - while a top consumer-grade CPU from AMD, Ryzen 9950X, has only 16 cores
- If a computation on a large dataset can be split into ***independent operations*** on smaller subsets, it can be executed in parallel on a GPU, resulting in significant performance gains
  - this kind of parallelism is known as Single Instruction, Multiple Threads (SIMT), or “embarrassingly parallel”
  - each operation (usually) takes longer to run on a GPU, but the ***wall clock time (“time to result”) is much smaller***
  - GPUs are optimized for *high throughput*, and not on latency

>>> General-purpose GPU programming

- We can give a GPU some instructions, known as *kernels*, on what operations to perform on a given set of data
  - do not have to be about rendering graphics, hence the name *general-purpose* GPU programming
- Accelerate (usually) *parts* of a bigger calculation
  - copy the data from CPU memory to GPU memory
  - then execute kernels on those data
  - transfer the result back to CPU once everything is done
- Different GPU vendors support, unfortunately different and oftentimes their own, interfaces for this, such as
  - **Nvidia's CUDA** (most used and well-established)
  - AMD's ROCm
  - Apple's Metal
  - Intel's oneAPI
- Many higher-level abstractions are available, such as `cupy`, `jax` and `pytorch` that we will talk about for the rest of the workshop
- Of course, you can also write your own kernel for more fine-grained controls (not covered in here)

Gentle introduction to high-level GPGPU libraries in python

jax, cupy, pytorch and all that

>>> Overarching introduction

- Provide abstraction in Python, freeing users the need of writing customs kernels to enjoy speedup from GPUs with *minimal* efforts
  - you can “just write like Python”
- Specifically, `cupy` and `jax` both offer `numpy`-like `ndarray` objects for array operations, as well as `scipy`-like routines
  - both supports multiple backends like CUDA and ROCm
  - that being said, the two are *quite different* in terms of their philosophies, capabilities and their ecosystems
  - will be covered in a bit more details next
- `pytorch` uses their own `torch.Tensor` objects and works slightly differently
  - the three libraries are actually *interoperable*, and that data can remain at the GPU memory as much as possible

>>> cupy: an almost drop-in replacement of numpy

- An *almost* drop-in replacement of numpy and scipy that uses GPU acceleration
  - for instance, use `cupy.dot` instead of `numpy.dot`
  - cursed way of "upgrading" to GPU – `import cupy as np`
- Support Python wrapping of user-defined CUDA kernels written in C/C++
- Despite the name, it also supports AMD's ROCm now
- Documentation: `https://docs.cupy.dev/en/stable/overview.html`

```python
import cupy as cp

# Check if CUDA is available
cp.cuda.is_available()
```

>>> cupy: an almost drop-in replacement of numpy

- `cupy.ndarray` is the central object
- For example, to represent this matrix

we write

$$x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
x = cp.array([[1,2],[3,4]])
```

- By default, the data reside on the GPU when it is available
- This can be checked using

```
x.device # Should return <CUDA Device 0>
```

>>> Data transfer and synchronization in cupy

- For `cupy`, data need to be created or transferred ***explicitly*** using `cp.array()` or `cp.asarray()` to the GPU
- Calculation results, in the same spirit, need to be moved back to the CPU using `x.get()` or `cp.asnumpy()`
- Kernels are launched ***asynchronously***, need to be synchronized if mixing CPU+GPU using `cp.cuda.Device().synchronize()`
  - otherwise you could get an error, or a wrong result (without knowing) in the worst case

>>> jax: more than just a replacement of numpy

- Unofficially, `jax` stands for
  - *j*ust-in-time compilation
  - *a*uto-differentiation or auto-gradient
  - *x*LA, or accelerated linear algebra
- Provide also numpy-like arrays and routines
  - cursed way of "upgrading" to GPU – `import jax.numpy as np`
- Support CPU, GPU (Nvidia's using CUDA and AMD's using ROCm) and TPU (Google's)
- Documentation: `https://docs.jax.dev/en/latest/`

```python
import jax

# Check what devices are available
jax.devices()
```

>>> jax: more than just a replacement of numpy

- `jax.Array` is `jax`'s implementation of `ndarray`
- For example, to represent this matrix

we write

$$x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
import jax.numpy as jnp
x = jnp.array([[1,2],[3,4]])
```

- By default, the data reside on the GPU when it is available
- This can be checked using

```
x.device # Should return CudaDevice(id=0)
```

>>> Data transfer and synchronization in jax

- For `jax`, data transfer is **implicit** and handled by `jax` itself
- Explicit transfer can be done using
  - `jax.device_put(x)` for transferring to the GPU
  - `jax.device_get(x)` for transferring to the CPU
- Same “issue” with the need for synchronization
  - using `jax.block_until_ready()`

```
(x @ x).block_until_ready() # or alternatively,
jax.block_until_ready(x @ x)
```

>>> Just-in-time compilation (JIT)

- Python is, at least out-of-the-box, a *dynamically interpreted* programming language
  - the interpreter (i.e., `python`) reads and executes your code line-by-line
  - basically synonymous to being **slow**
  - especially when using loops, where the same translation happens over and over again unnecessarily
- Just-in-time compilation solves this problem
  - by compiling the code to machine code during program execution
  - and using the compiled code in subsequent calls
- In `jax`, this can be done using the `@jax.jit` decorator

```
import jax

@jax.jit
def f(x, y):
    return jax.numpy.sin(x) + y**2
```

>>> jax: a functional programming paradigm

- In order for `jax`'s JIT, auto-vectorization and auto-differentiation to work correctly 100% of the time, *functions need to be "pure"*
  - think of that roughly as mathematical functions
  - the output of a function f depends only on its input x
  - given the same input, the output $f(x)$ is always the same
- This means that a function cannot have any "side effects", for example –
  - cannot access global variables/external states
  - cannot modify input variables
  - performing I/O calls

```
# This is an example of a _non-pure_ function
tax_rate = 0.2

def price_with_tax(price):
    return price * (1 + tax_rate)
```

>>> pytorch: a graph-based compute engine

- A (very) popular machine learning framework
- `torch.Tensor` is the central "array" object in the library
  - "philosophically" different from `cupy` and `jax`
- Perform auto-differentiation by creating a compute graph
- Support a wide range of accelerators, even with Apple's (using Metal Performance Shader, or MPS)
- Documentation: `https://docs.pytorch.org/docs/stable/index.html`

```
import torch
# Check if CUDA is available
torch.cuda.is_available()
# Check if MPS is available
torch.backends.mps.is_available()
```

>>> Data transfer and synchronization in pytorch

- For `pytorch`, data need to be created or transferred ***explicitly*** (just like `cupy`) using `x.to(device='cuda')` to the GPU
- Calculation results, in the same spirit, need to be moved back to the CPU using the same interface `x.cpu()`
  - if `x` is part of a compute graph, then `x.detach().cpu()` is needed
- Kernels are launched ***asynchronously***, need to be synchronized if mixing CPU+GPU using `torch.cuda.synchronize()`

Hands-on demonstrations and exercises

>>> GPU hardware at the KU HPC cluster

- GPU-equipped headnodes/frontends:
  - `astro01`: 4 × Nvidia A100 40 GB
  - `astro02`: 3 × Nvidia A30 24 GB
  - `astro04`: 4 × Nvidia RTX A6000 48 GB
  - for more information (e.g., usage), run `nvidia-smi`
- SLRUM partitions:
  - `astro_gpu` and `astro_gpu_interactive`: 6 × Nvidia H100 NVL 94 GB
  - `astro2_gpu`: 11 × Nvidia A100 40 GB

To submit a job to those partitions, do for example –

```
srun -p astro2_gpu --gres=gpu:1 nvidia-smi
```

Disclaimer: These are accurate *as of* 28/01/2026. You should always check the latest setup and availability of GPUs

>>> Getting ready

- Have access to a GPU –
  - Tycho
  - Google colab
  - Your own computer (running Linux/macOS)

```
# Miniconda
curl -O https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
# Change the name of the environment as needed
conda create -n tycho_workshop_demo python=3.10 numpy scipy ipython
# Install JAX
pip install -U "jax[cuda12]"
# Install cupy
pip install -U cupy-cuda12x
# Install torch
pip install -U torch torchvision
```

- Demonstration notebooks (and the slides) are available at the UCPH_Tycho Teams and also `https://github.com/ricokaloklo/tycho-workshop-2026-gpu`
- “Etiquette”: specify the environment variable `CUDA_VISIBLE_DEVICES` when using the GPUs on the headnodes

Concluding remarks

Some practical suggestions on which library to use in your work

>>> Summary

- GPU is perfect for doing simpler and *independent* tasks in parallel, reducing the wall clock time
- There are many high-level Python libraries (e.g., `cupy`, `jax`, `pytorch`) for GPGPU programming with minimal efforts
- GPUs are available at the KU HPC cluster (some are often under-utilized)
- Practical suggestions on accelerating your code using GPUs – if you
  - starting fresh: use `cupy` or `jax`
  - upgrading existing code-base: use `jax` for its implicit data movement
  - need to run of all sort of devices: use `jax` or `pytorch`
  - have absolutely no idea: give `jax` a try