Sign up for the lab platform:

Brev invite link (click or scan)

Open this doc where we can share info & ask questions:

app.sli.do/event/opMgqcvf7VPETGx9gVjF3p

WIFI: SSIDSSIDSSIDSSID

PW: PASSWORDPASSWORD



PYTHON FOR HPC TUTORIAL

Sign up for the lab platform:

Brev invite link (click or scan)

Open this doc where we can share info & ask questions:

app.sli.do/event/opMgqcvf7VPETGx9gVjF3p

WIFI: SSIDSSIDSSIDSSID

PW: PASSWORDPASSWORD



PYTHON FOR HPC TUTORIAL



ANDY TERELL
Product Manager @ NVIDIA
CUDA Python

Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido

Brev: invite link Q&A: app.sli.do/event/opMgqcvf7VPETGx9gVjF3p



ANDY TERELL
Product Manager @ NVIDIA
CUDA Python



DAVIN POTTS
Pythonista @ Appliomics
DragonHPC

Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido

Brev: invite link Q&A: app.sli.do/event/opMgqcvf7VPETGx9gVjF3p



ANDY TERELL
Product Manager @ NVIDIA
CUDA Python



DAVIN POTTS
Pythonista @ Appliomics
DragonHPC



AARON JOMY
System Software @ CERN
CppJIT, CppInterOp

Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido

Brev: invite link Q&A: app.sli.do/event/opMgqcvf7VPETGx9gVjF3p



ANDY TERELL
Product Manager @ NVIDIA
CUDA Python



DAVIN POTTS
Pythonista @ Appliomics
DragonHPC



AARON JOMY
System Software @ CERN
CppJIT, CppInterOp



VASSIL VASSILEV
Research Software @ Princeton/CERN
Clang-REPL, Clad

Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido

This course is web-based; no local install needed.

Recommended browser: Chrome (28+).

Javascript and WebRTC must be enabled.

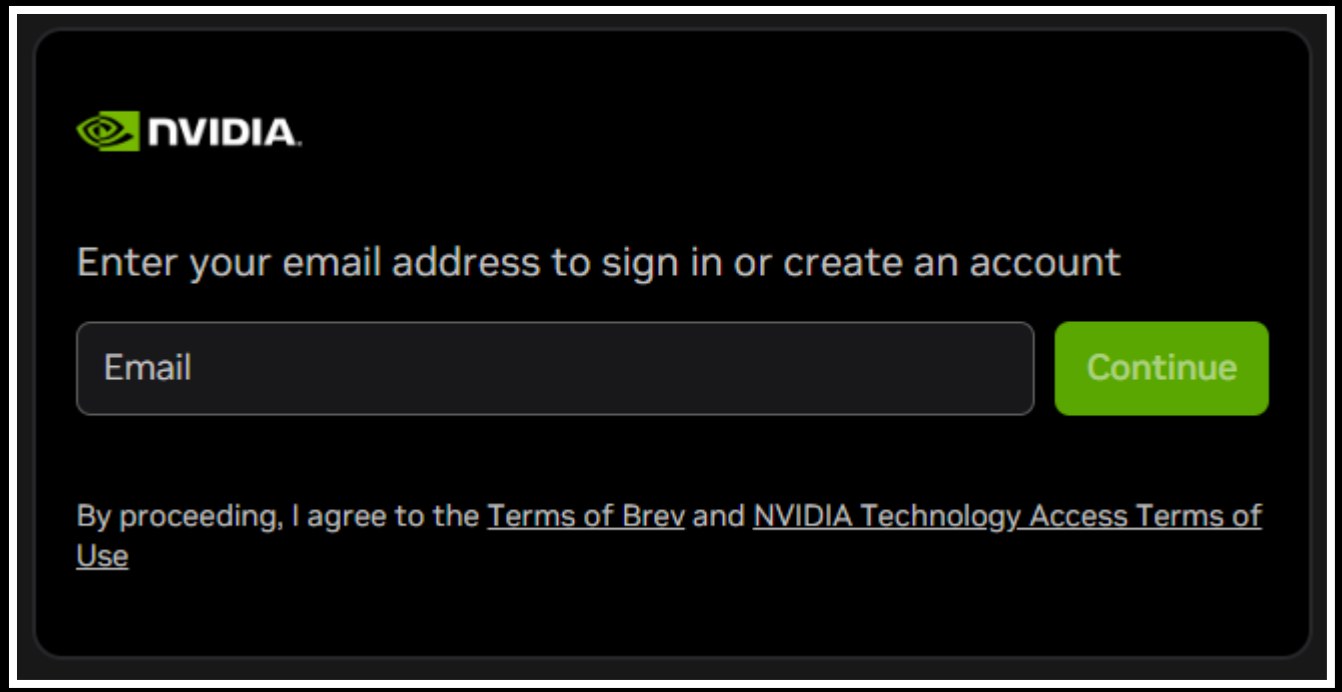
Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido

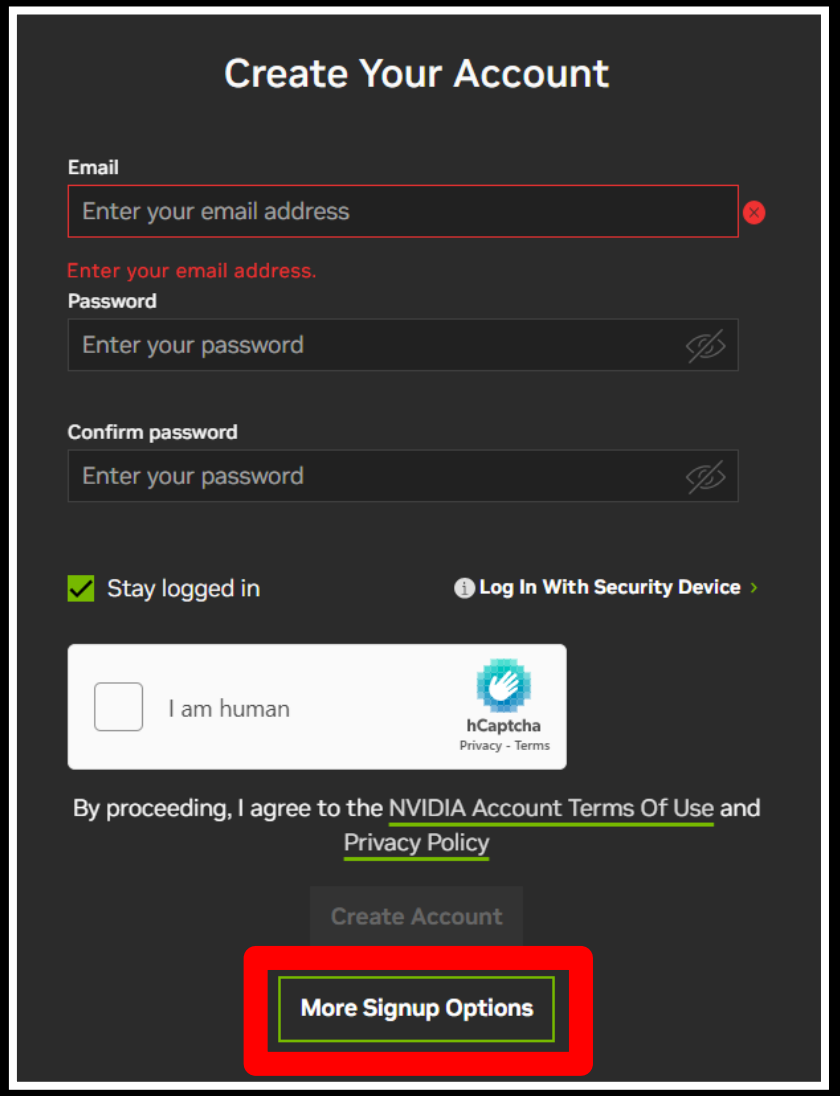
All content is open source and available on GitHub:

github.com/pyHPC/sc2026-tutorial

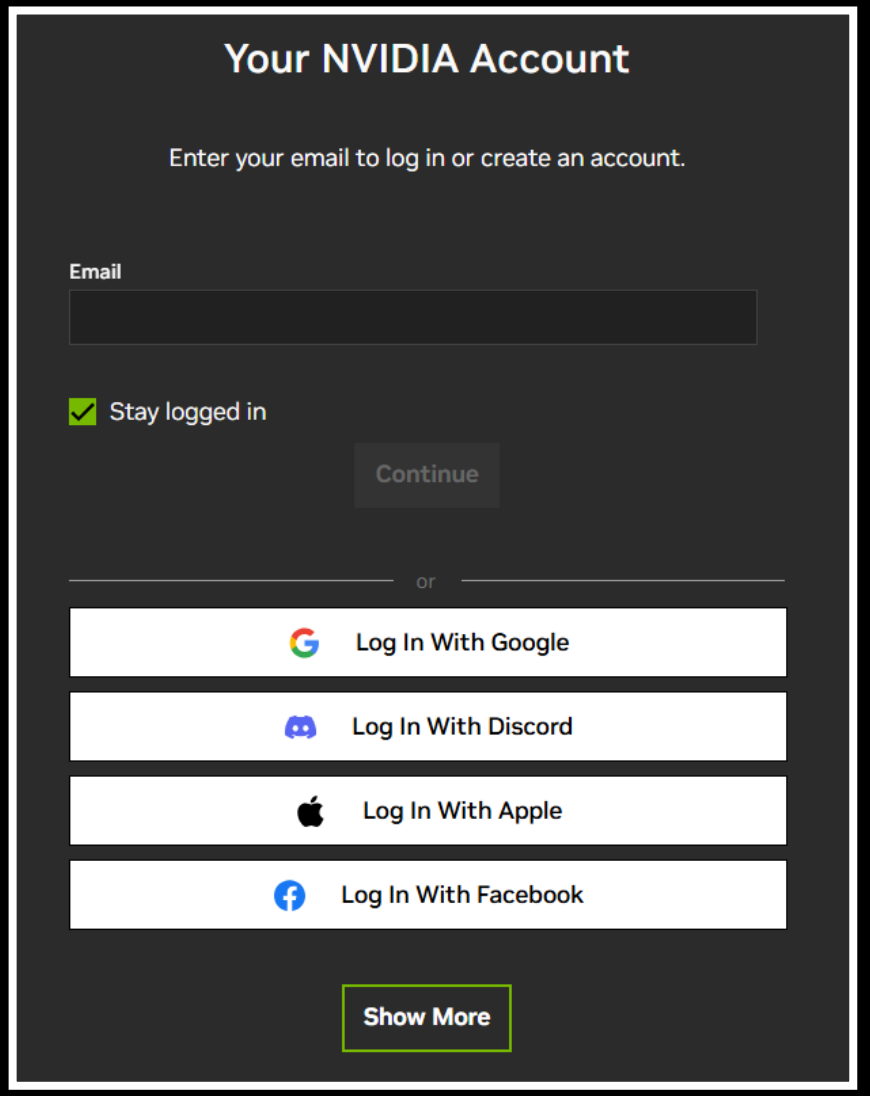
Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido

Brev: invite link Q&A: app.sli.do/event/opMgqcvf7VPETGx9gVjF3p

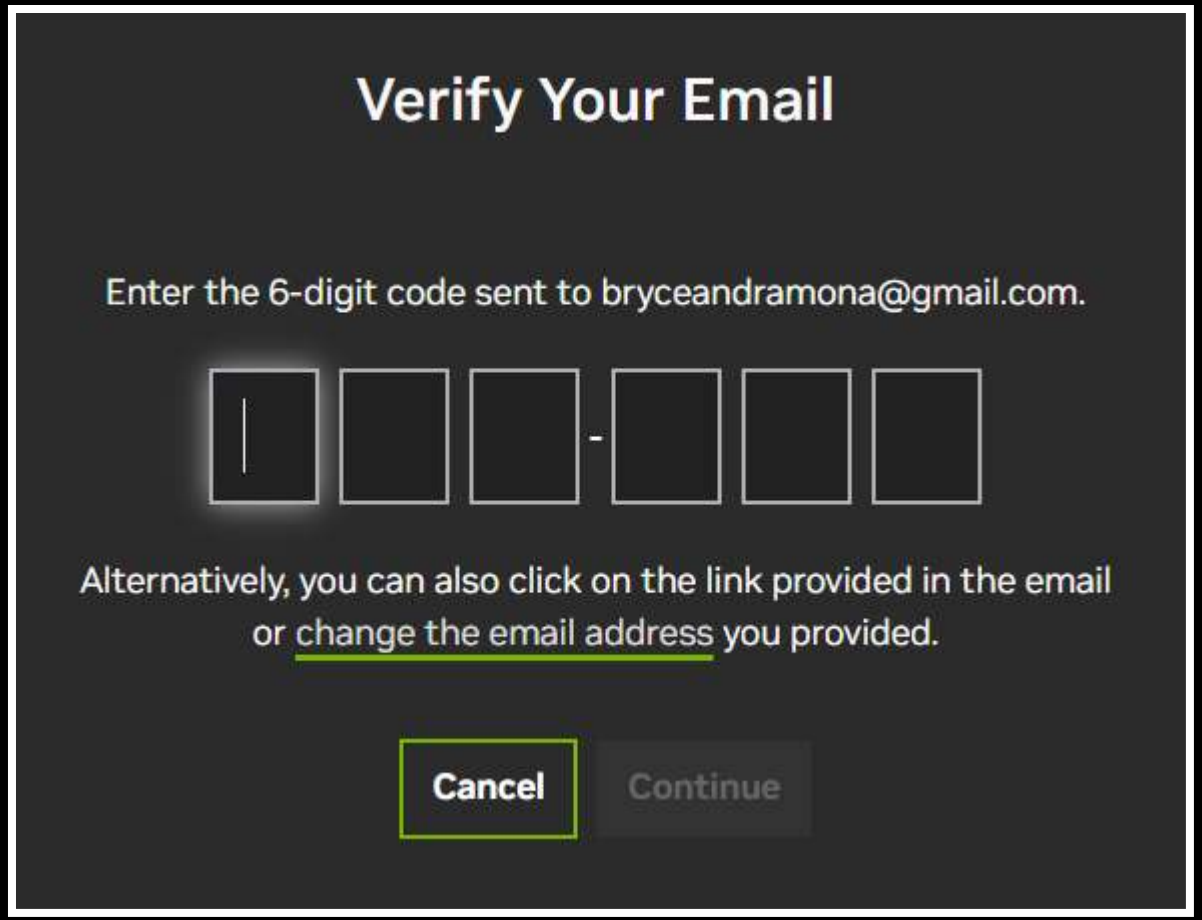



Create Your Account
Email
Enter your email address
Enter your email address.
Password
Enter your password
Confirm password
Enter your password
Stay logged in
Log In With Security Device
I am human
hCaptcha
Privacy - Terms
By proceeding, I agree to the NVIDIA Account Terms Of Use and Privacy Policy
Create Account
More Signup Options

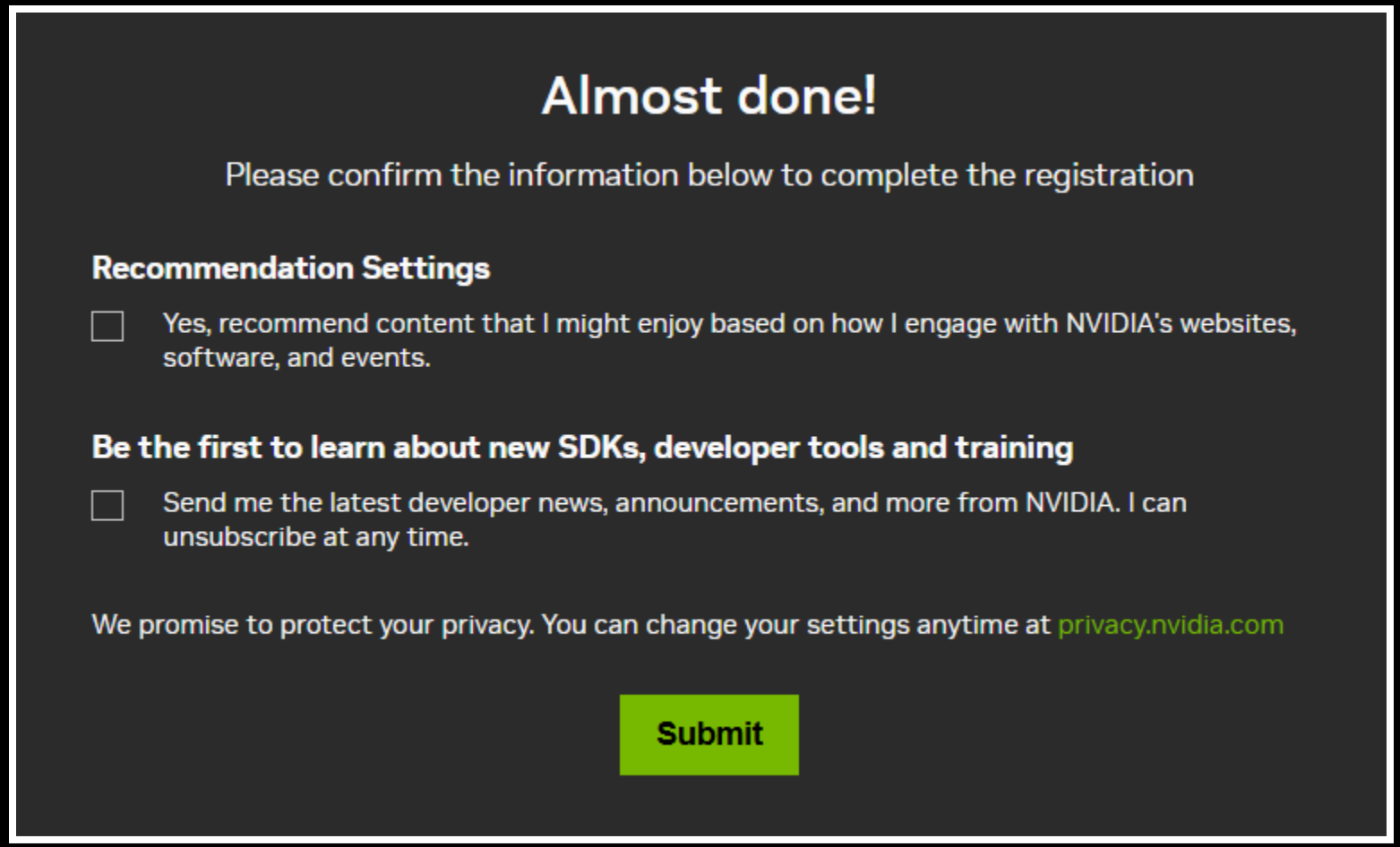
Brev: invite link Q&A: app.sli.do/event/opMgqcvf7VPETGx9gVjF3p



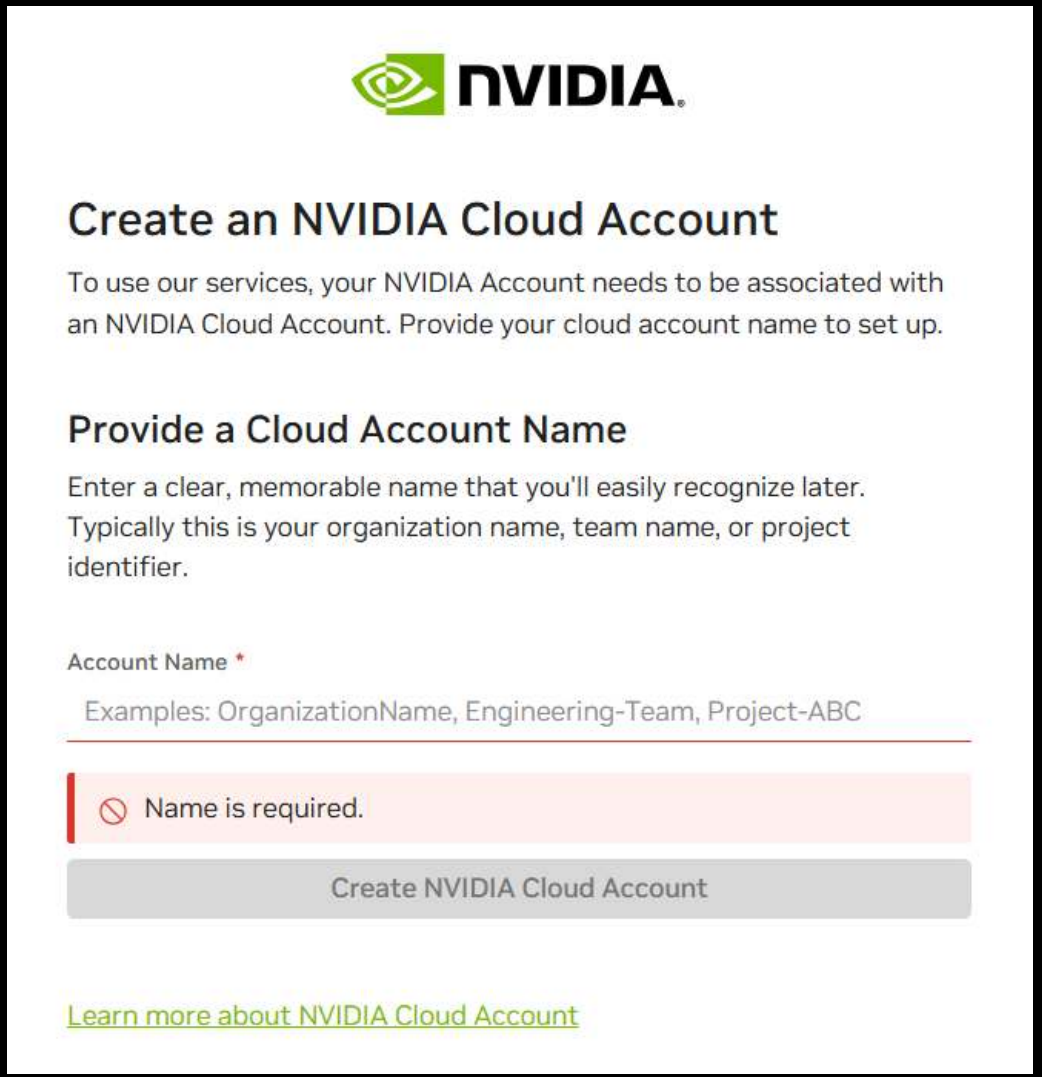
Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido



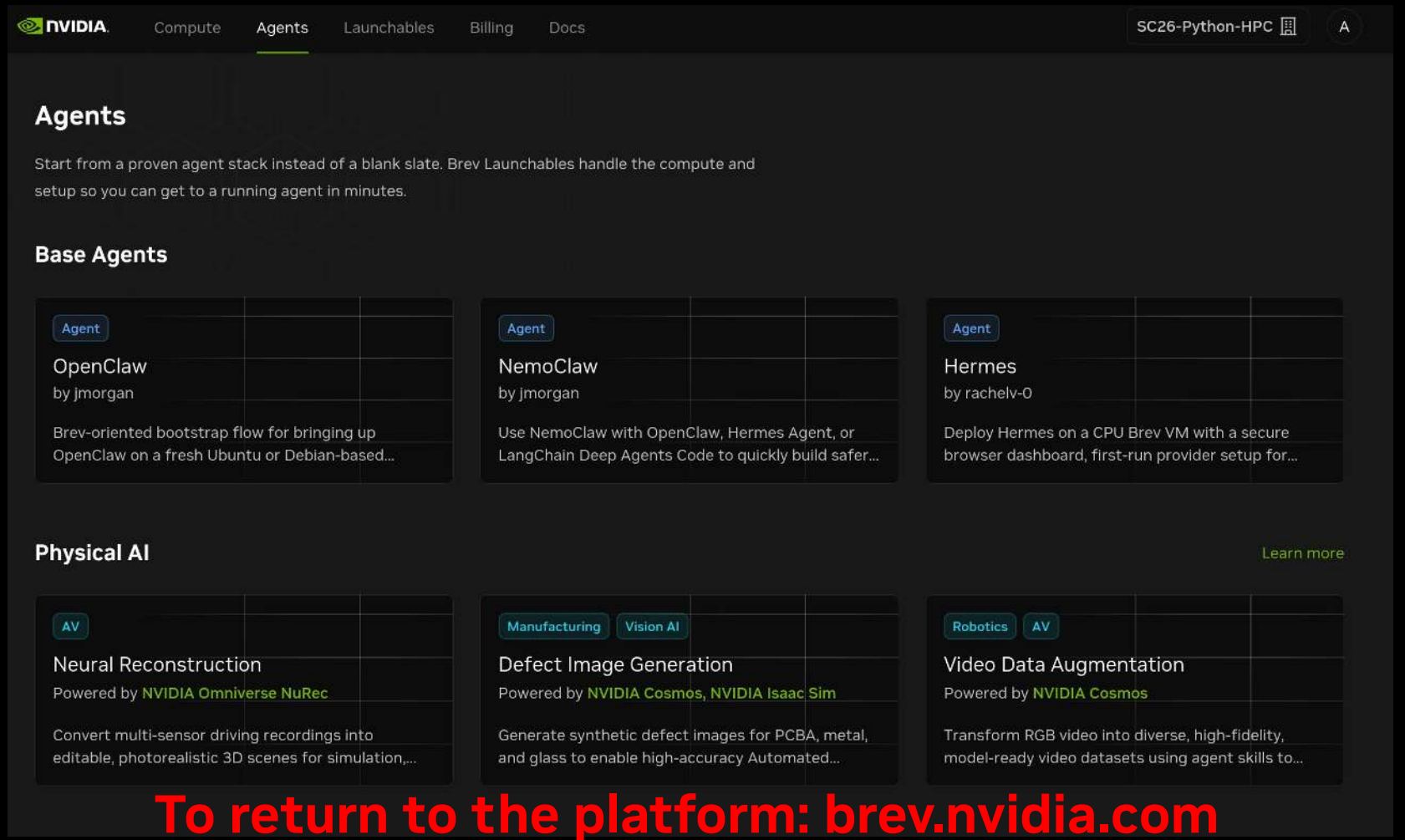
Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido


Almost done!
Please confirm the information below to complete the registration
Recommendation Settings
Yes, recommend content that I might enjoy based on how I engage with NVIDIA's websites, software, and events.
Be the first to learn about new SDKs, developer tools and training
Send me the latest developer news, announcements, and more from NVIDIA. I can unsubscribe at any time.
We promise to protect your privacy. You can change your settings anytime at privacy.nvidia.com
Submit

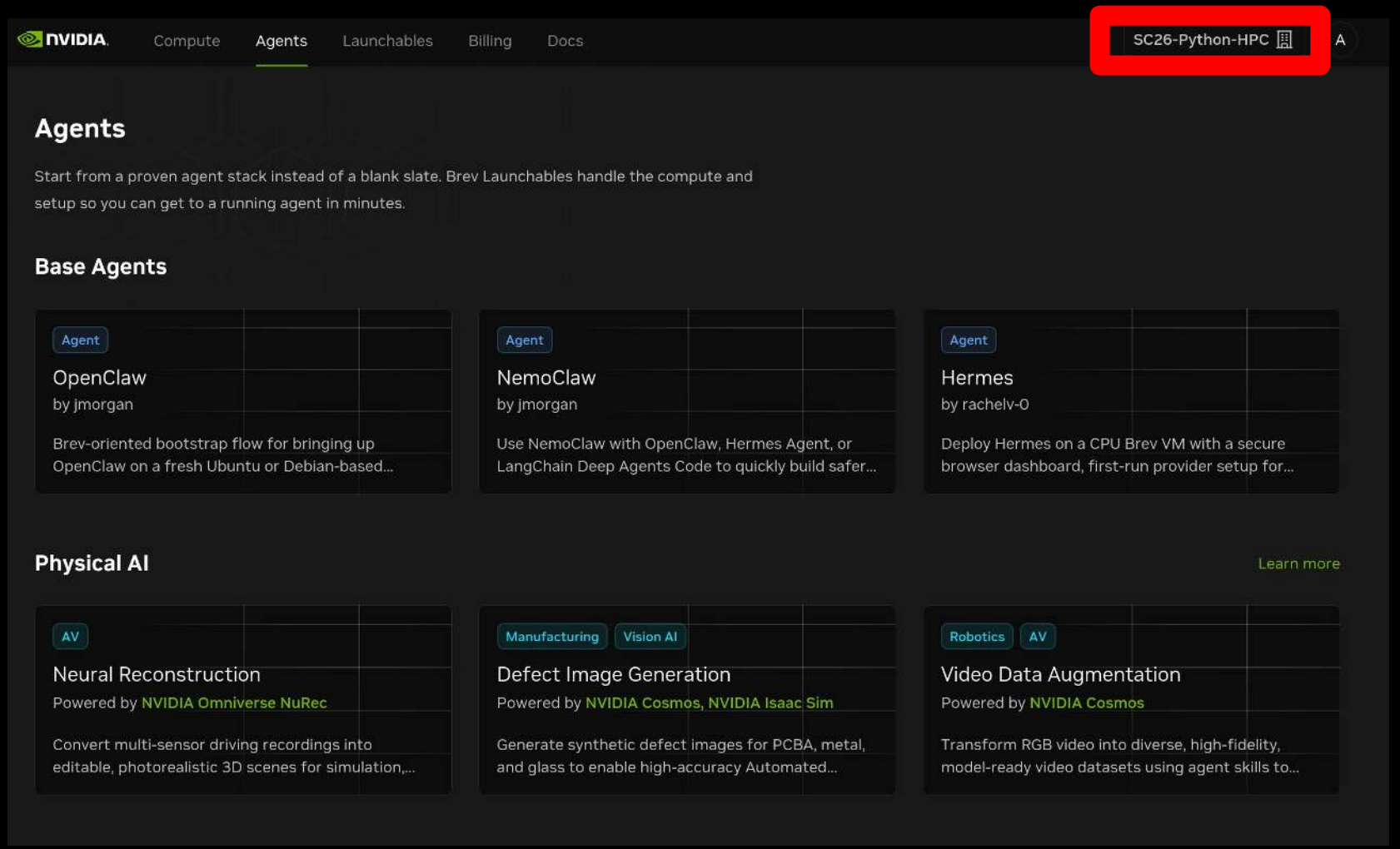
Please enter your first & last name here!



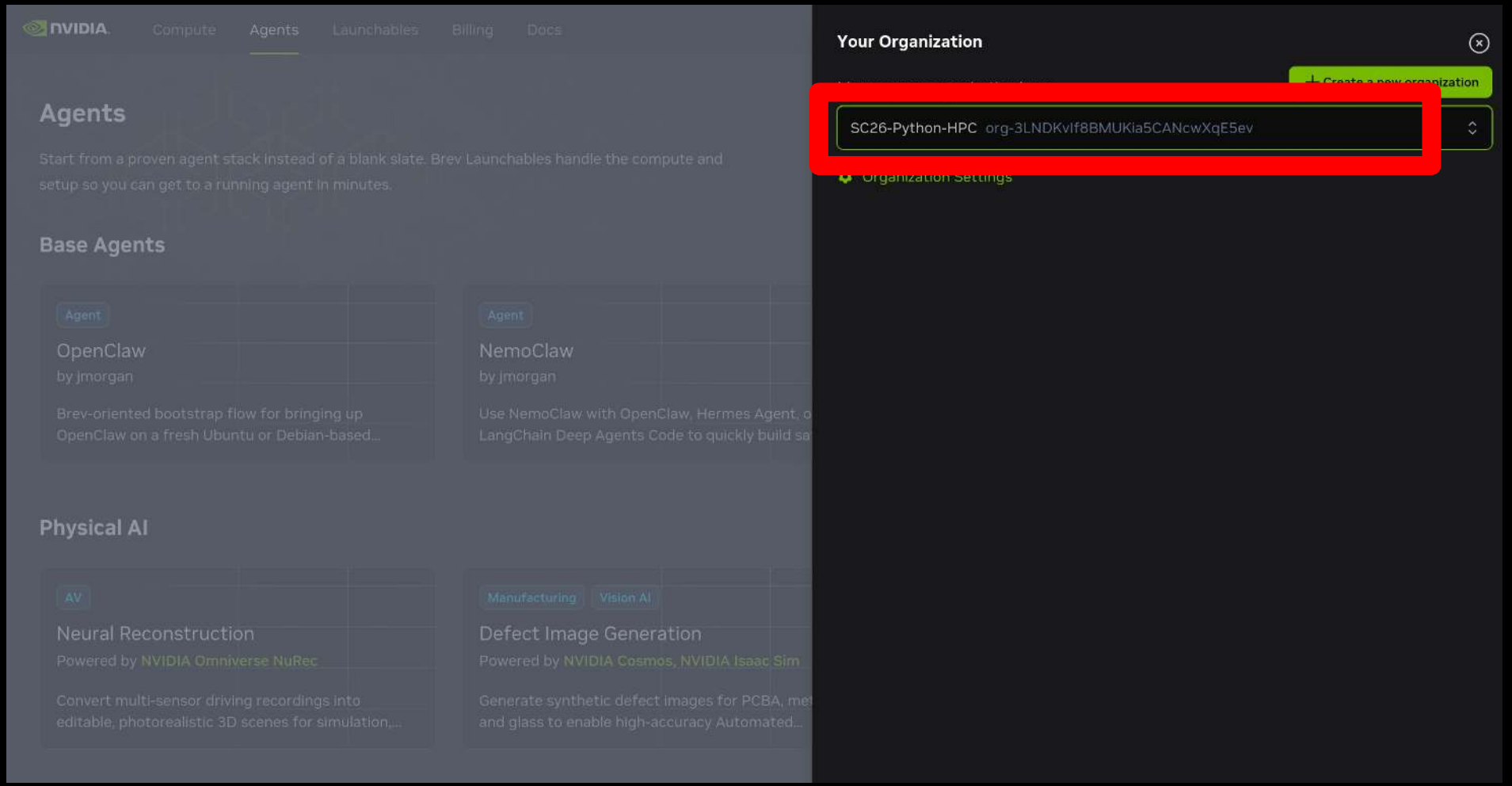
Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido


NVIDIA
Compute
Agents
Launchables
Billing
Docs
SC26-Python-HPC
A
Agents
Start from a proven agent stack instead of a blank slate. Brev Launchables handle the compute and setup so you can get to a running agent in minutes.
Base Agents
Agent
OpenClaw
by jmorgan
Brev-oriented bootstrap flow for bringing up OpenClaw on a fresh Ubuntu or Debian-based...
Agent
NemoClaw
by jmorgan
Use NemoClaw with OpenClaw, Hermes Agent, or LangChain Deep Agents Code to quickly build safer...
Agent
Hermes
by rachelv-0
Deploy Hermes on a CPU Brev VM with a secure browser dashboard, first-run provider setup for...
Physical AI
Learn more
AV
Neural Reconstruction
Powered by NVIDIA Omniverse NuRec
Convert multi-sensor driving recordings into editable, photorealistic 3D scenes for simulation,...
Manufacturing
Vision AI
Defect Image Generation
Powered by NVIDIA Cosmos, NVIDIA Isaac Sim
Generate synthetic defect images for PCBA, metal, and glass to enable high-accuracy Automated...
Robotics
AV
Video Data Augmentation
Powered by NVIDIA Cosmos
Transform RGB video into diverse, high-fidelity, model-ready video datasets using agent skills to...
To return to the platform: brev.nvidia.com

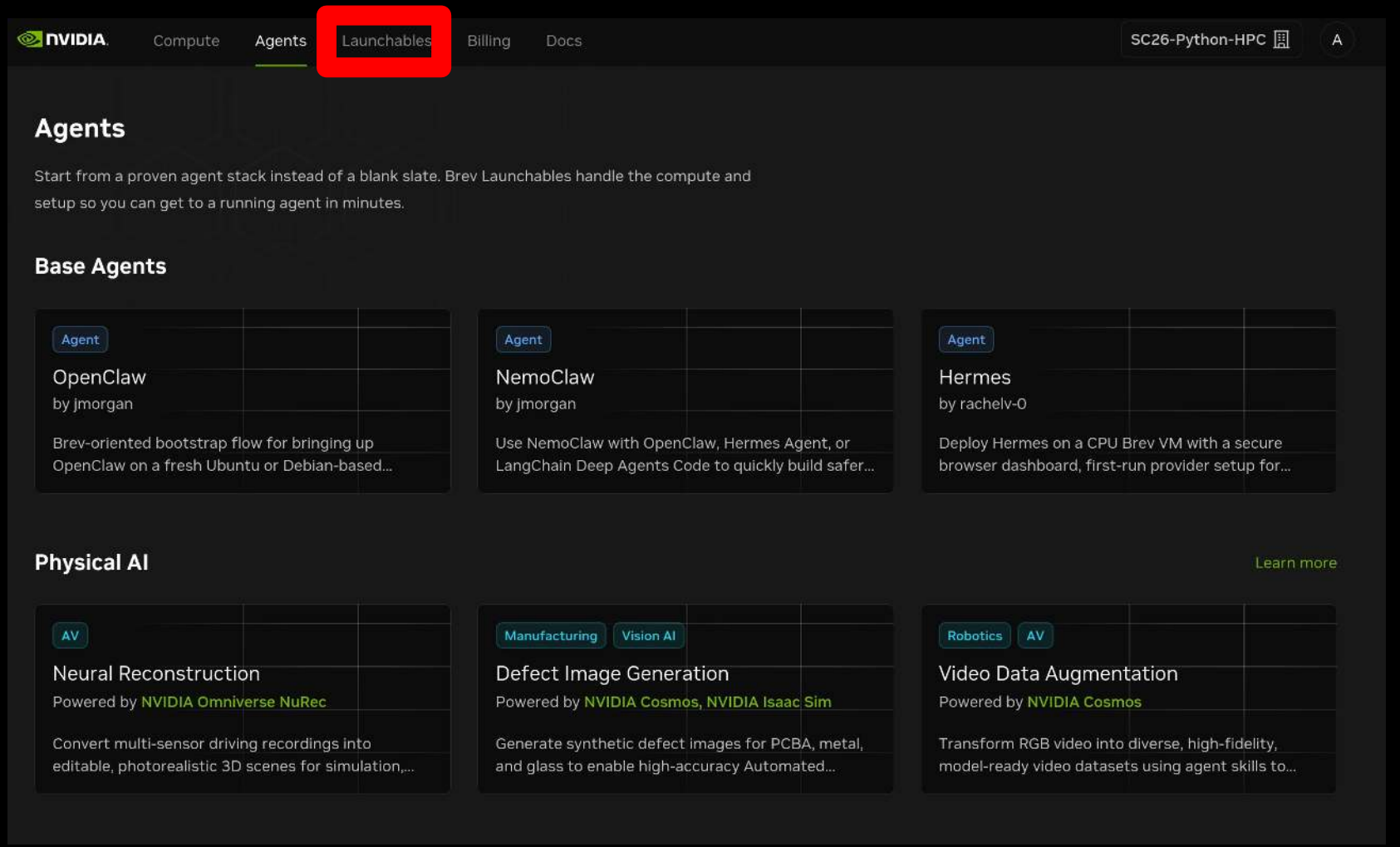
Brev: invite link Q&A: app.sli.do/event/opMgqcvf7VPETGx9gVjF3p



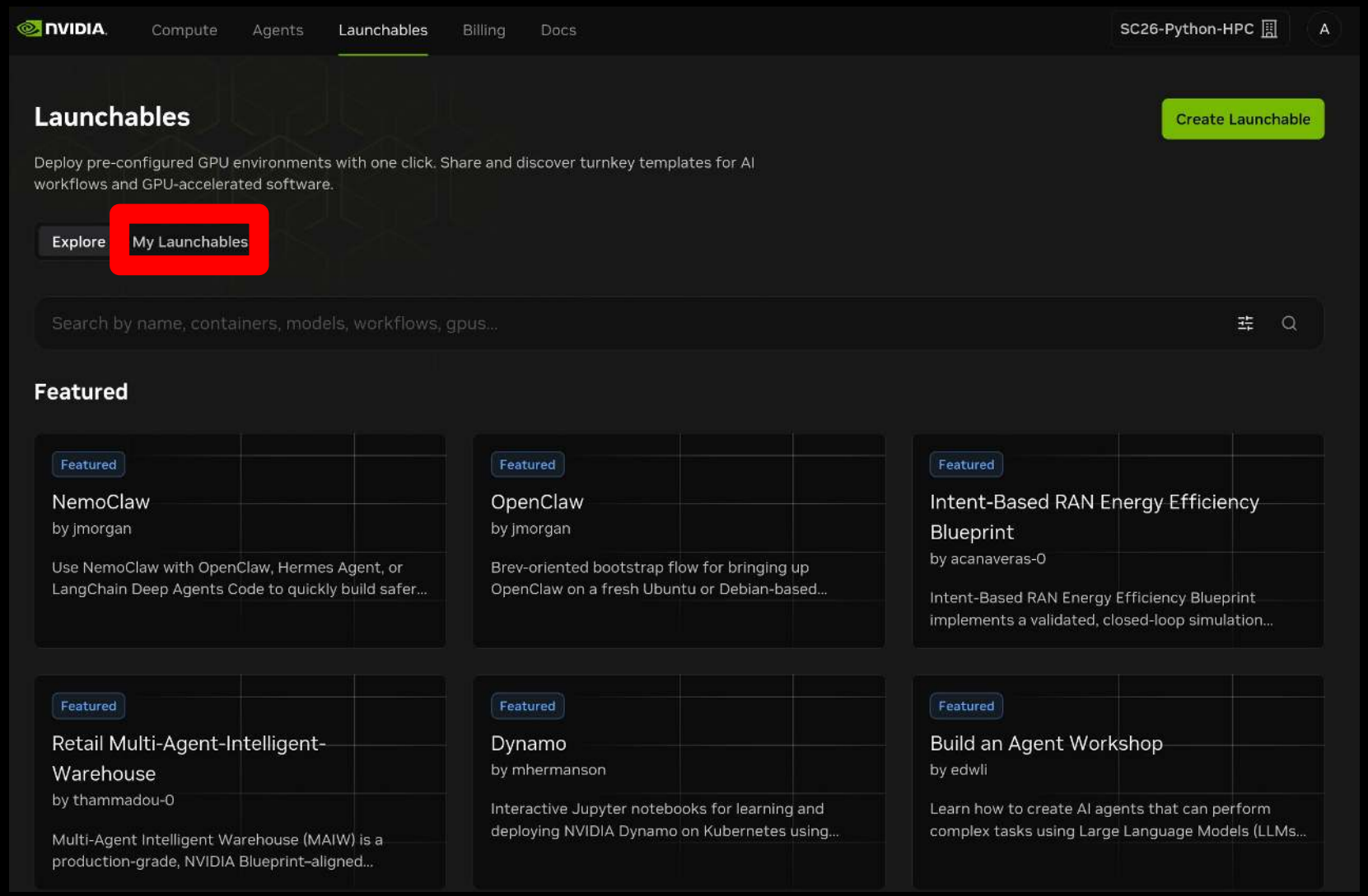
Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido

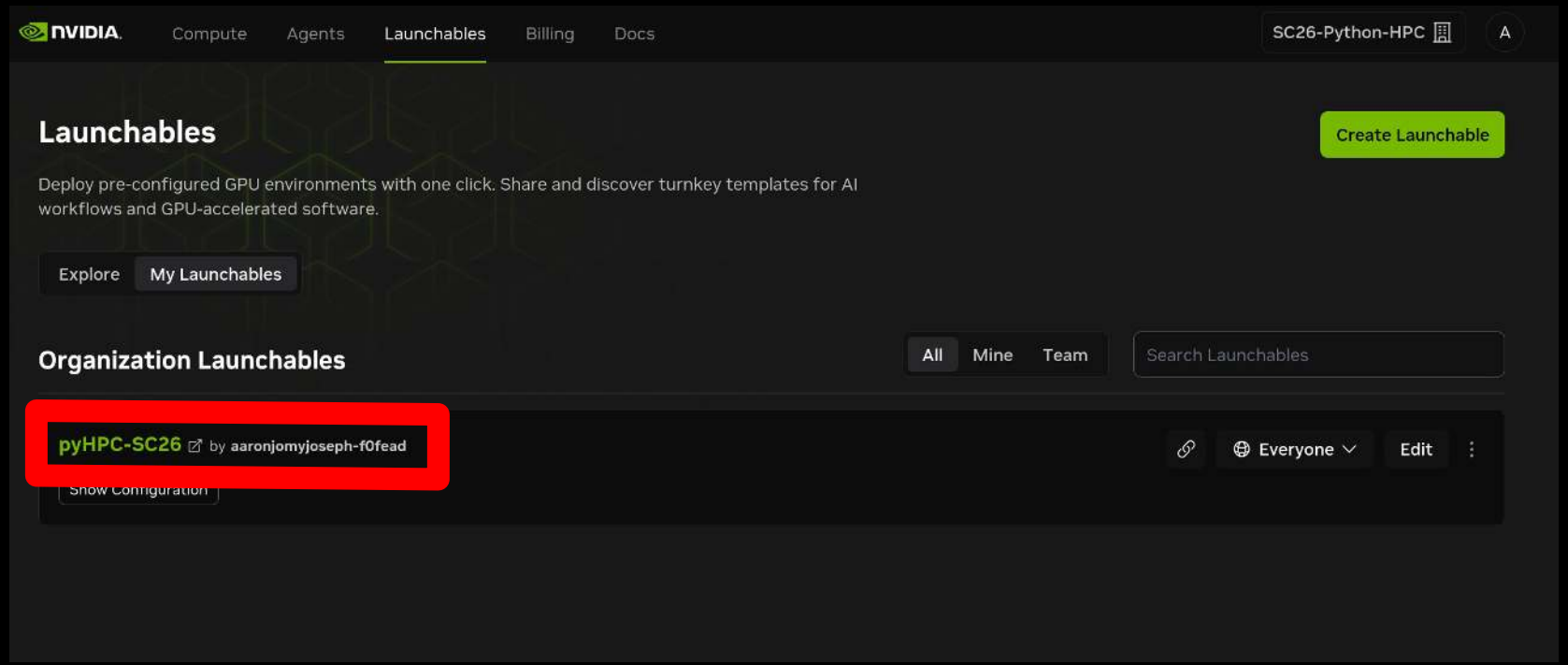

NVIDIA
Compute
Agents
Launchables
Billing
Docs
Agents
Start from a proven agent stack instead of a blank slate. Brev Launchables handle the compute and setup so you can get to a running agent in minutes.
Base Agents
Agent
OpenClaw
by jmorgan
Brev-oriented bootstrap flow for bringing up OpenClaw on a fresh Ubuntu or Debian-based...
Agent
NemoClaw
by jmorgan
Use NemoClaw with OpenClaw, Hermes Agent, o
LangChain Deep Agents Code to quickly build sa
Physical AI
AV
Neural Reconstruction
Powered by NVIDIA Omniverse NuRec
Convert multi-sensor driving recordings into editable, photorealistic 3D scenes for simulation,...
Manufacturing
Vision AI
Defect Image Generation
Powered by NVIDIA Cosmos, NVIDIA Isaac Sim
Generate synthetic defect images for PCBA, me
and glass to enable high-accuracy Automated...
Your Organization
Create a new organization
SC26-Python-HPC org-3LNDKvIf8BMUKia5CANcwXqE5ev
Organization Settings

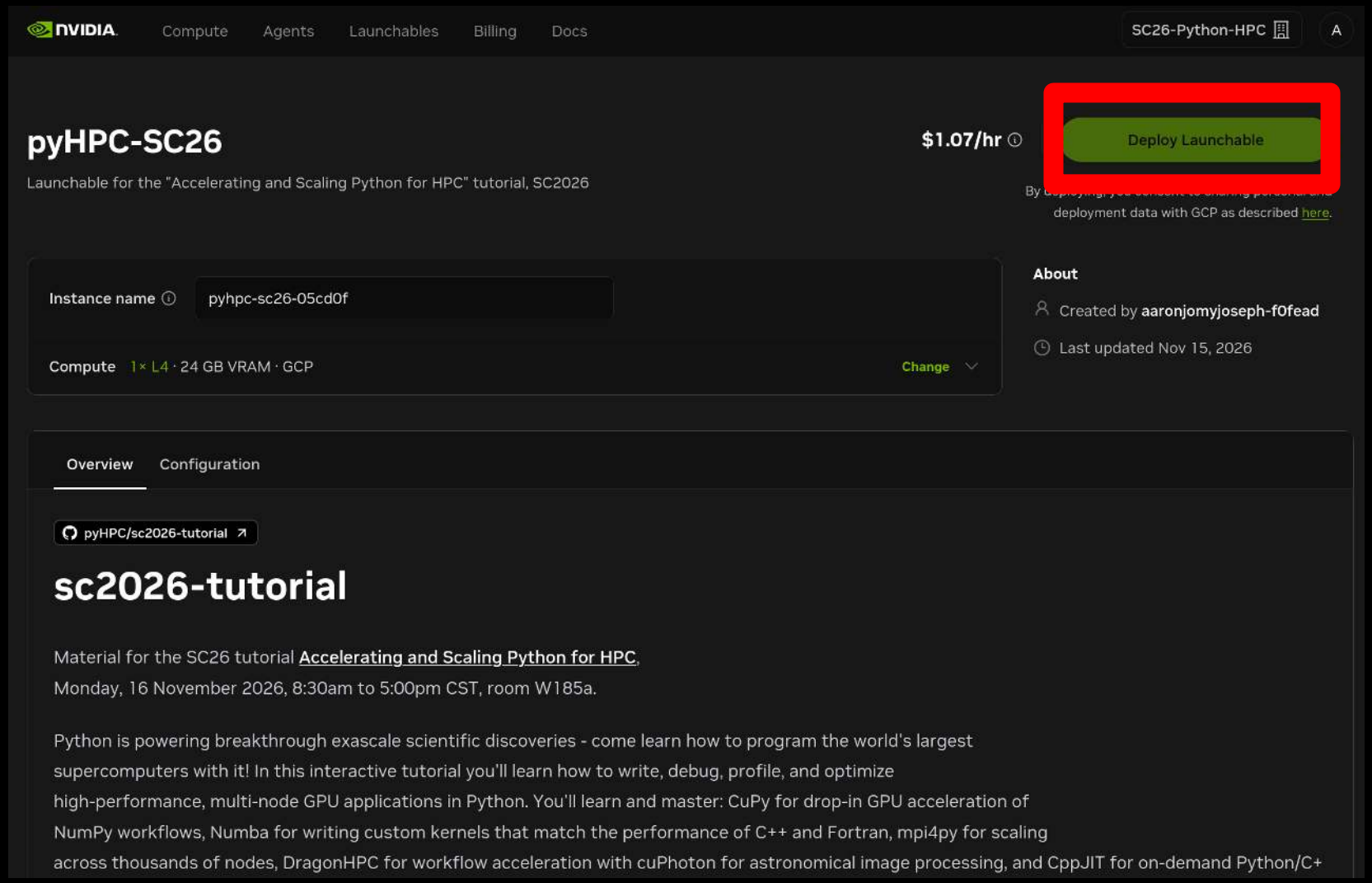
Brev: invite link Q&A: app.sli.do/event/opMgqcvf7VPETGx9gVjF3p



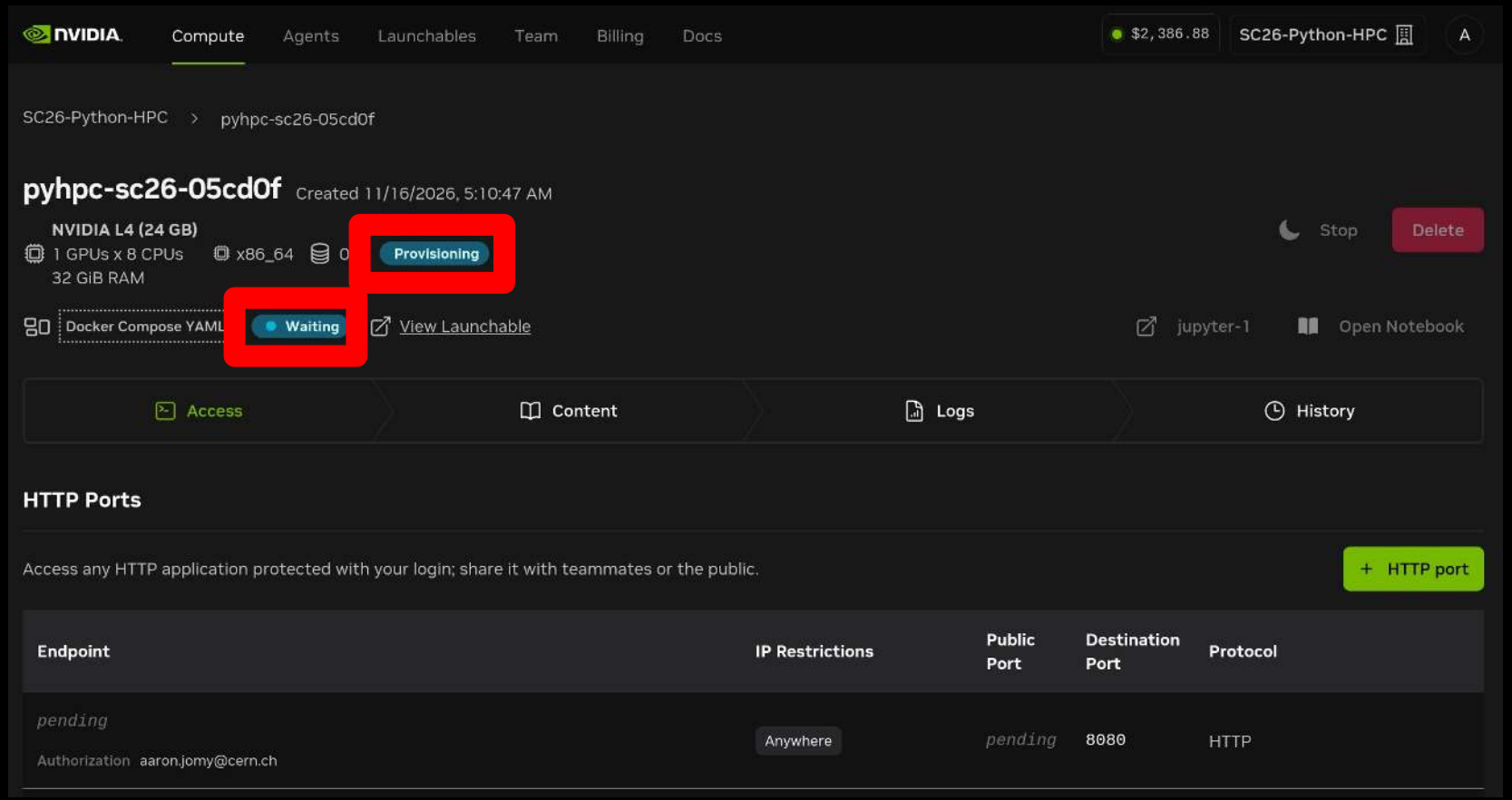
Copyright (C) 2026 NVIDIA | Brev: brev.nvidia.com | Q&A: Slido

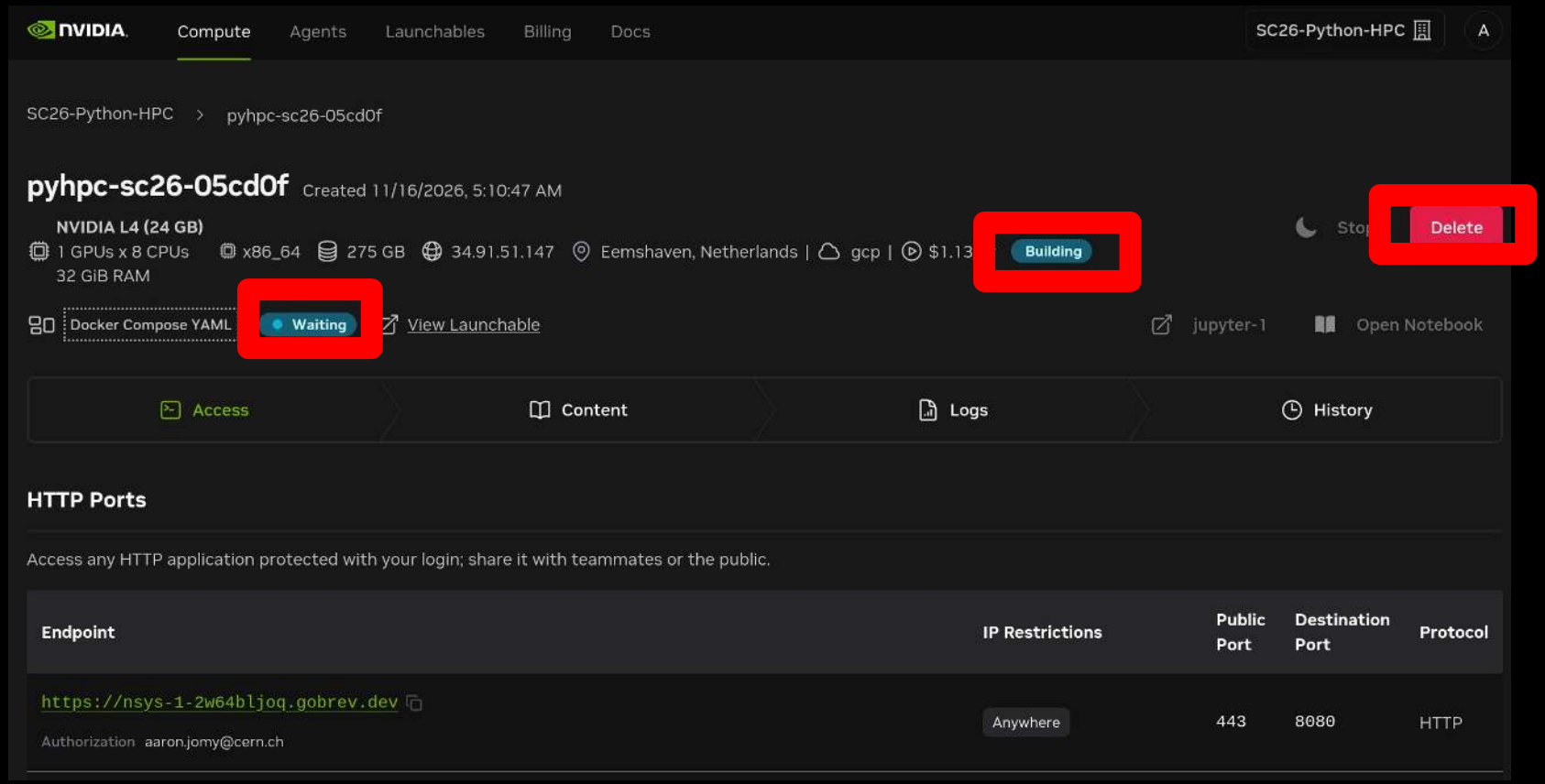

NVIDIA
Compute
Agents
Launchables
Billing
Docs
SC26-Python-HPC
A
Launchables
Create Launchable
Deploy pre-configured GPU environments with one click. Share and discover turnkey templates for AI workflows and GPU-accelerated software.
Explore
My Launchables
Search by name, containers, models, workflows, gpus...
Featured
Featured
NemoClaw
by jmorgan
Use NemoClaw with OpenClaw, Hermes Agent, or LangChain Deep Agents Code to quickly build safer...
Featured
OpenClaw
by jmorgan
Brev-oriented bootstrap flow for bringing up OpenClaw on a fresh Ubuntu or Debian-based...
Featured
Intent-Based RAN Energy Efficiency Blueprint
by acanaveras-0
Intent-Based RAN Energy Efficiency Blueprint implements a validated, closed-loop simulation...
Featured
Retail Multi-Agent-Intelligent-Warehouse
by thammadou-0
Multi-Agent Intelligent Warehouse (MAIW) is a production-grade, NVIDIA Blueprint-aligned...
Featured
Dynamo
by mhermanson
Interactive Jupyter notebooks for learning and deploying NVIDIA Dynamo on Kubernetes using...
Featured
Build an Agent Workshop
by edwli
Learn how to create AI agents that can perform complex tasks using Large Language Models (LLMs...


NVIDIA
Compute
Agents
Launchables
Billing
Docs
SC26-Python-HPC
A
Launchables
Create Launchable
Deploy pre-configured GPU environments with one click. Share and discover turnkey templates for AI workflows and GPU-accelerated software.
Explore
My Launchables
Organization Launchables
All
Mine
Team
Search Launchables
pyHPC-SC26
by aaronjomyjoseph-f0fead
Everyone
Edit
Show Configuration

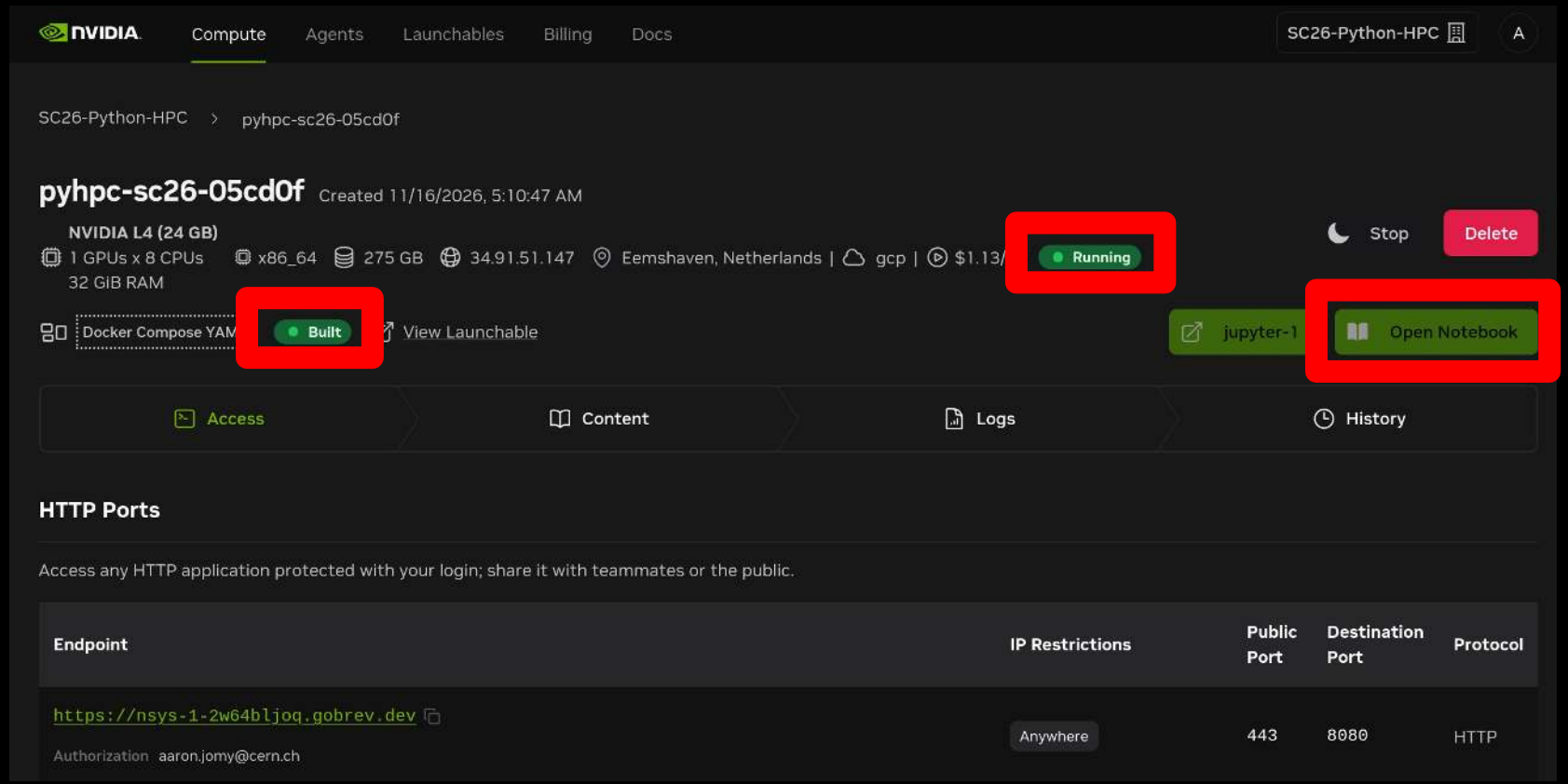

NVIDIA
Compute
Agents
Launchables
Billing
Docs
SC26-Python-HPC
A
pyHPC-SC26
Launchable for the "Accelerating and Scaling Python for HPC" tutorial, SC2026
$1.07/hr
Deploy Launchable
deployment data with GCP as described here.
Instance name
pyhpc-sc26-05cd0f
Compute
1× L4 · 24 GB VRAM · GCP
Change
About
Created by aaronjomyjoseph-f0fead
Last updated Nov 15, 2026
Overview
Configuration
pyHPC/sc2026-tutorial
sc2026-tutorial
Material for the SC26 tutorial Accelerating and Scaling Python for HPC,
Monday, 16 November 2026, 8:30am to 5:00pm CST, room W185a.
Python is powering breakthrough exascale scientific discoveries - come learn how to program the world's largest supercomputers with it! In this interactive tutorial you'll learn how to write, debug, profile, and optimize high-performance, multi-node GPU applications in Python. You'll learn and master: CuPy for drop-in GPU acceleration of NumPy workflows, Numba for writing custom kernels that match the performance of C++ and Fortran, mpi4py for scaling across thousands of nodes, DragonHPC for workflow acceleration with cuPhoton for astronomical image processing, and CppJIT for on-demand Python/C+

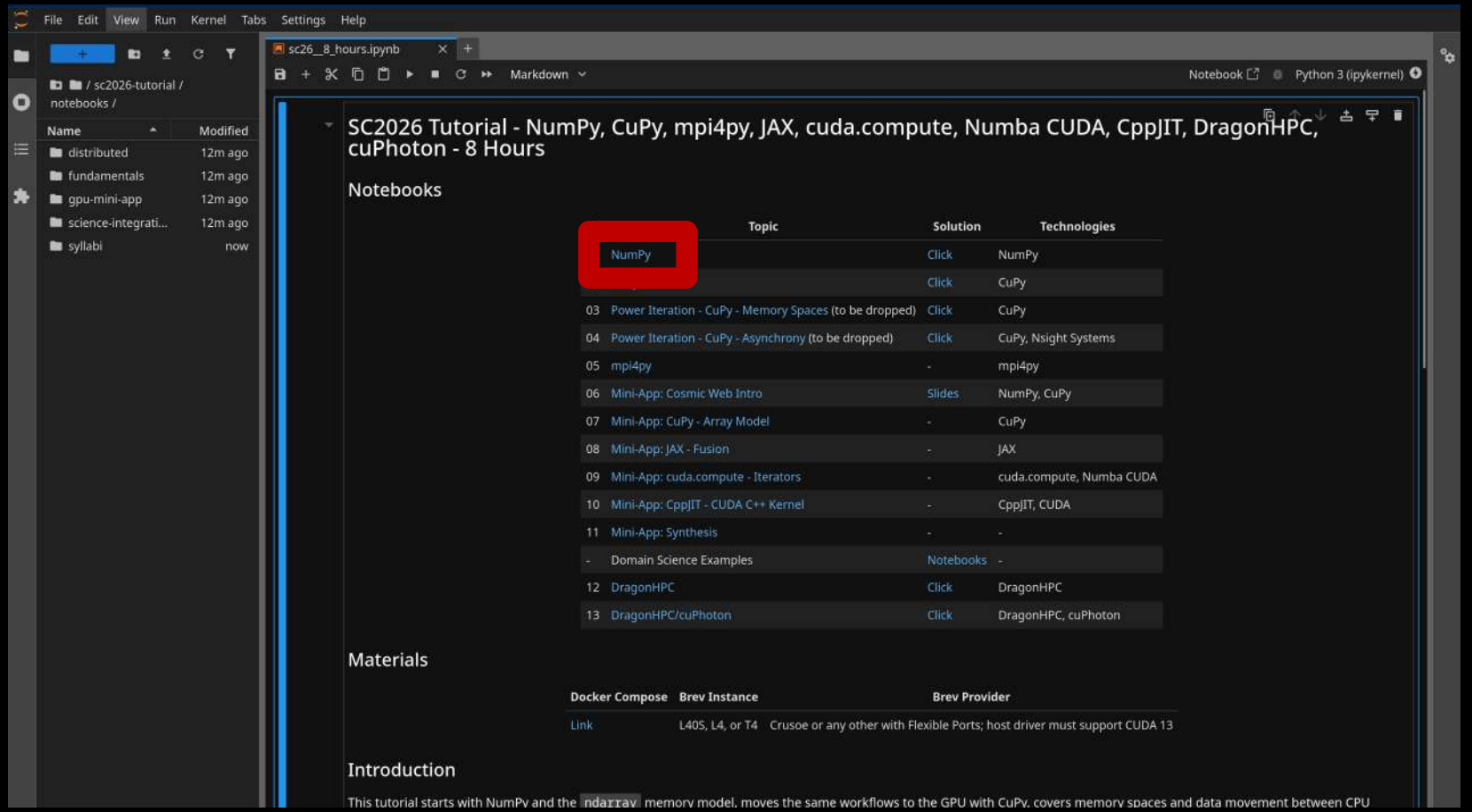
Brev: invite link Q&A: app.sli.do/event/opMgqcvf7VPETGx9gVjF3p

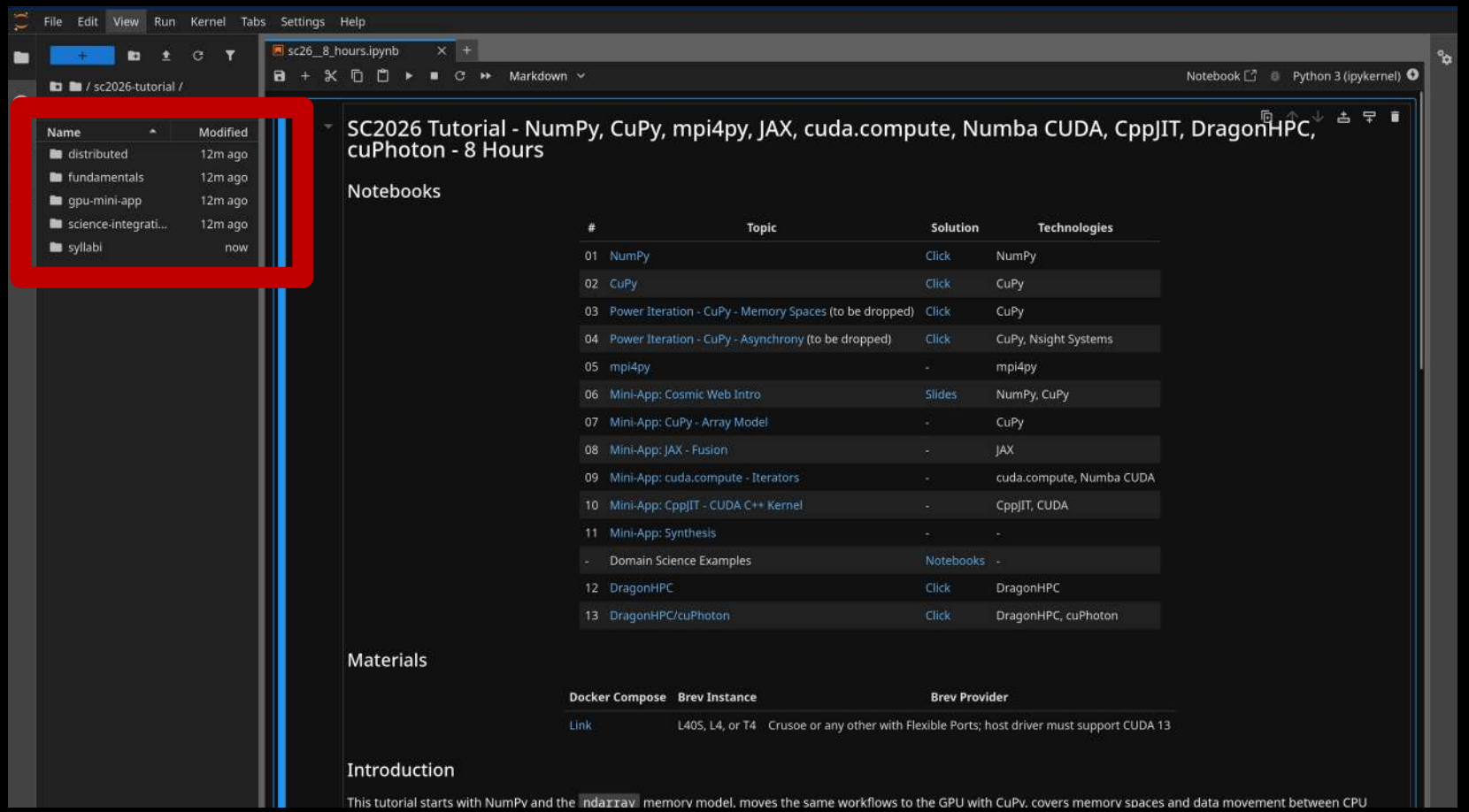



NVIDIA
Compute
Agents
Launchables
Billing
Docs
SC26-Python-HPC
A
SC26-Python-HPC > pyhpc-sc26-05cd0f
pyhpc-sc26-05cd0f
Created 11/16/2026, 5:10:47 AM
NVIDIA L4 (24 GB)
1 GPUs x 8 CPUs
x86_64
275 GB
34.91.51.147
Eemshaven, Netherlands | gcp | $1.13
32 GiB RAM
Building
Delete
Docker Compose YAML
Waiting
View Launchable
jupyter-1
Open Notebook
Access
Content
Logs
History
HTTP Ports
Access any HTTP application protected with your login; share it with teammates or the public.
Endpoint
IP Restrictions
Public Port
Destination Port
Protocol
https://nsys-1-2w64bljoq.gobrev.dev
Authorization aaron.jomy@cern.ch
Anywhere
443
8080
HTTP

Brev: invite link Q&A: app.sli.do/event/opMgqcvf7VPETGx9gVjF3p




File Edit View Run Kernel Tabs Settings Help
sc26__8_hours.ipynb
Markdown
Notebook
Python 3 (ipykernel)
/ sc2026-tutorial / notebooks /
Name
Modified
distributed 12m ago
fundamentals 12m ago
gpu-mini-app 12m ago
science-integrati... 12m ago
syllabi now
SC2026 Tutorial - NumPy, CuPy, mpi4py, JAX, cuda.compute, Numba CUDA, CppJIT, DragonHPC, cuPhoton - 8 Hours
Notebooks
Topic
Solution
Technologies
NumPy
Click
NumPy
Click
CuPy
03 Power Iteration - CuPy - Memory Spaces (to be dropped)
Click
CuPy
04 Power Iteration - CuPy - Asynchrony (to be dropped)
Click
CuPy, Nsight Systems
05 mpi4py
-
mpi4py
06 Mini-App: Cosmic Web Intro
Slides
NumPy, CuPy
07 Mini-App: CuPy - Array Model
-
CuPy
08 Mini-App: JAX - Fusion
-
JAX
09 Mini-App: cuda.compute - Iterators
-
cuda.compute, Numba CUDA
10 Mini-App: CppJIT - CUDA C++ Kernel
-
CppJIT, CUDA
11 Mini-App: Synthesis
-
-
- Domain Science Examples
Notebooks
-
12 DragonHPC
Click
DragonHPC
13 DragonHPC/cuPhoton
Click
DragonHPC, cuPhoton
Materials
Docker Compose
Brev Instance
Brev Provider
Link
L40S, L4, or T4
Crusoe or any other with Flexible Ports; host driver must support CUDA 13
Introduction
This tutorial starts with NumPy and the ndarray memory model, moves the same workflows to the GPU with CuPy, covers memory spaces and data movement between CPU


File Edit View Run Kernel Tabs Settings Help
sc26__8_hours.ipynb
Markdown
Notebook
Python 3 (ipykernel)
/ sc2026-tutorial /
Name
Modified
distributed 12m ago
fundamentals 12m ago
gpu-mini-app 12m ago
science-integrati... 12m ago
syllabi now
SC2026 Tutorial - NumPy, CuPy, mpi4py, JAX, cuda.compute, Numba CUDA, CppJIT, DragonHPC, cuPhoton - 8 Hours
Notebooks
Topic Solution Technologies
01 NumPy Click NumPy
02 CuPy Click CuPy
03 Power Iteration - CuPy - Memory Spaces (to be dropped) Click CuPy
04 Power Iteration - CuPy - Asynchrony (to be dropped) Click CuPy, Nsight Systems
05 mpi4py - mpi4py
06 Mini-App: Cosmic Web Intro Slides NumPy, CuPy
07 Mini-App: CuPy - Array Model - CuPy
08 Mini-App: JAX - Fusion - JAX
09 Mini-App: cuda.compute - Iterators - cuda.compute, Numba CUDA
10 Mini-App: CppJIT - CUDA C++ Kernel - CppJIT, CUDA
11 Mini-App: Synthesis - -
- Domain Science Examples Notebooks -
12 DragonHPC Click DragonHPC
13 DragonHPC/cuPhoton Click DragonHPC, cuPhoton
Materials
Docker Compose Brev Instance Brev Provider
Link L40S, L4, or T4 Crusoe or any other with Flexible Ports; host driver must support CUDA 13
Introduction
This tutorial starts with NumPy and the ndarray memory model, moves the same workflows to the GPU with CuPy, covers memory spaces and data movement between CPU



Accelerating and Scaling Python for HPC Basics

Dr Vassil Vassilev, CERN

compiler-research.org

Prerequisites

- Be interactive, ask questions!
- Please share your own experience!



Warmup

- How many of you have used Python before?
  - How many of you have a project > 1000 lines of code?
  - > 5000 lines?
  - > 15000 lines?
  - How many of you worked in a team of 2 on such project?
    - > 5
    - > 10
- How many of you coded in a different language than Python?
  - In a low-level language such as C++, C, or assembly?
- What's your average data set? MB, GB, TB, PB, EB?

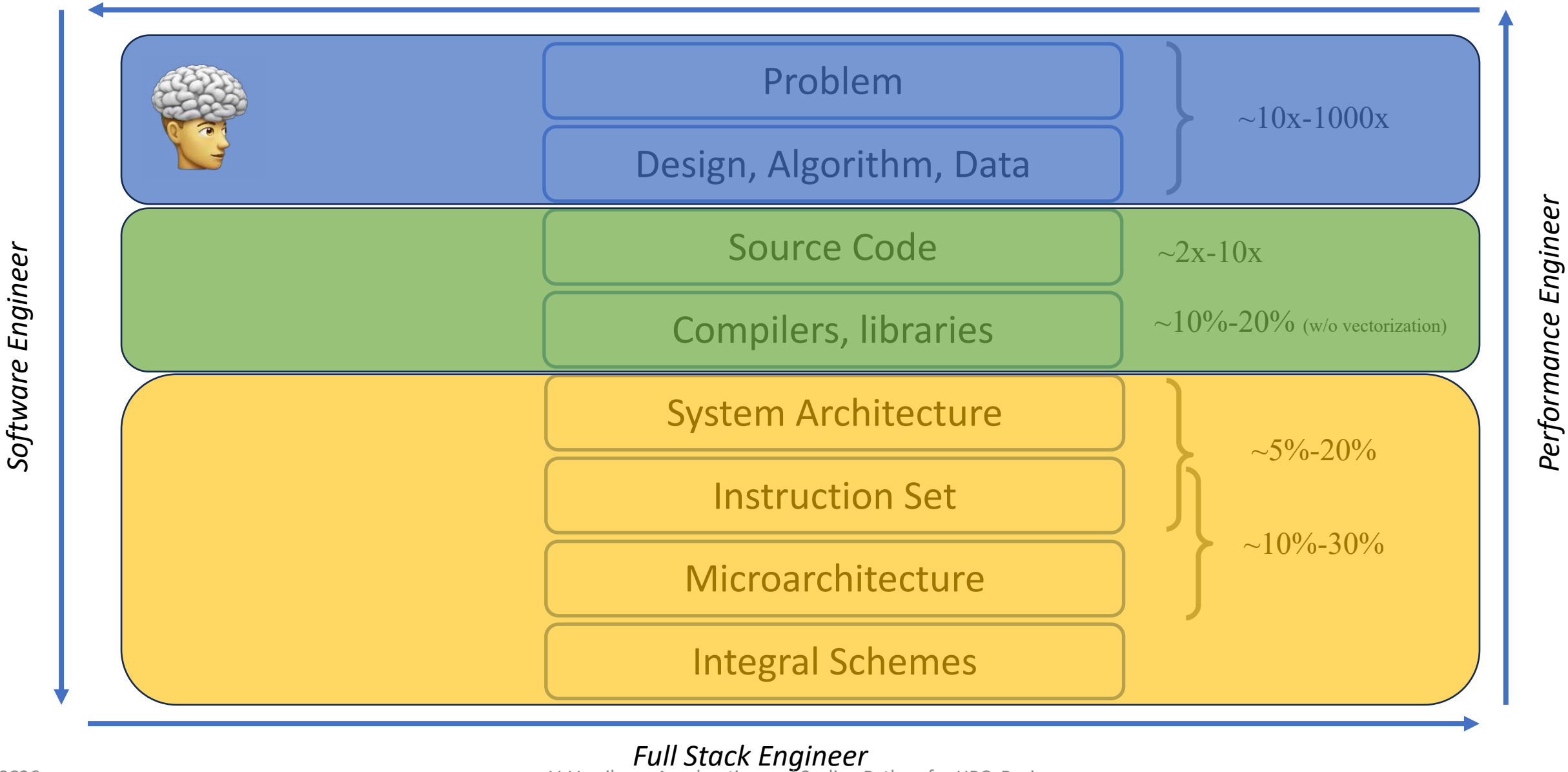
Goals

- Understand the general principles behind high-performance programming
- Recognize and explore performance optimization opportunities
- Build intuition about computer program execution
- Practice

Just Enough Performance

```
def f(N = 100, M = 1000, L = 10000):
    for i in range(N):
        for j in range(M):
            for k in range(L):
                g(i, j, k)
```

Outline


Software Engineer
Performance Engineer
Problem
Design, Algorithm, Data
~10x-1000x
Source Code
Compilers, libraries
~2x-10x
~10%-20% (w/o vectorization)
System Architecture
Instruction Set
Microarchitecture
Integral Schemes
~5%-20%
~10%-30%
Full Stack Engineer

If I had an hour to solve a problem I'd spend 55 minutes thinking about the problem and 5 minutes thinking about the solutions

- *We have no proof Albert Einstein ever said*

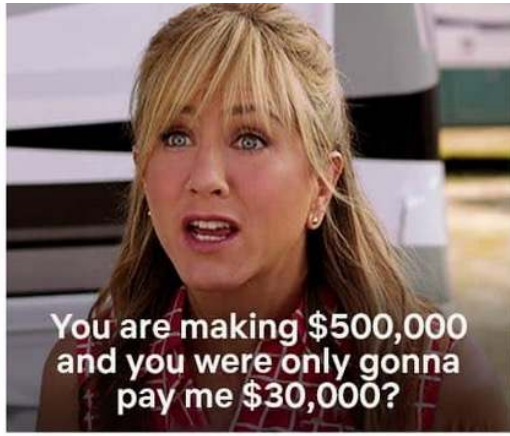
Problem Defining

- Is a problem worth solving?
- What would be the impact of having a solution?
- Is the problem new to the domain?
- Does the problem exist anywhere else? How it is solved in other domains?
- What domain knowledge it requires and can it define away some issues with existing solutions?
- Can I decompose it
  - Pontryagin's suboptimality principle
  - Morphology analysis

Solution Designing

- What's the mathematical foundation of my solution?
- What are the implementation requirements?
  - Do I need ~~3 years~~ 100B tokens to implement it?
  - What skills?
  - How many people?









Solution Designing

- How do I translate the solution into code?
  - What's my input data?
  - What algorithms I should use? What's their complexity?
  - What are my data structures?
  - How to scale?

Debugging is 3 times more difficult than coding the algorithm.
What happens if coded a solution at the limit of our skills?

Algorithm

What makes one program better than another?

- ***Correctness***
- Speed
- Resources it takes
- What else?



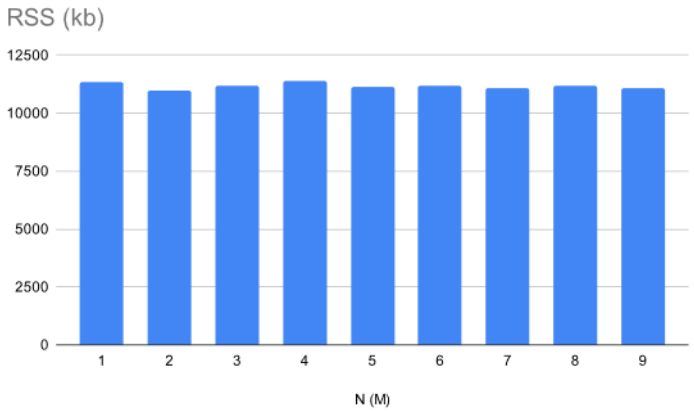
Algorithm Analysis. Example

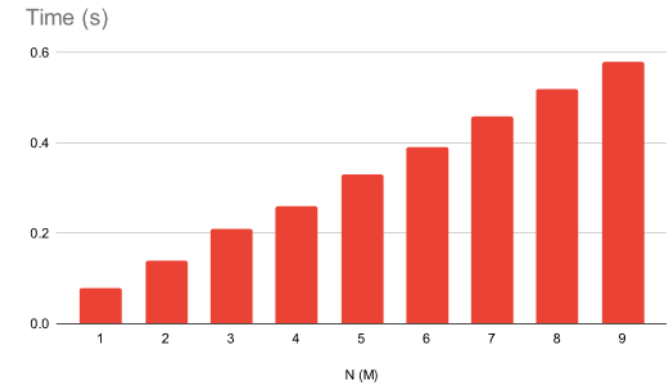
Find the sum of the first N numbers.

$$\sum_{i=0}^{n} i$$

```
def sum_to_n(n):
    total = 0
    for i in range(n + 1):
        total += i
    return total
```

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

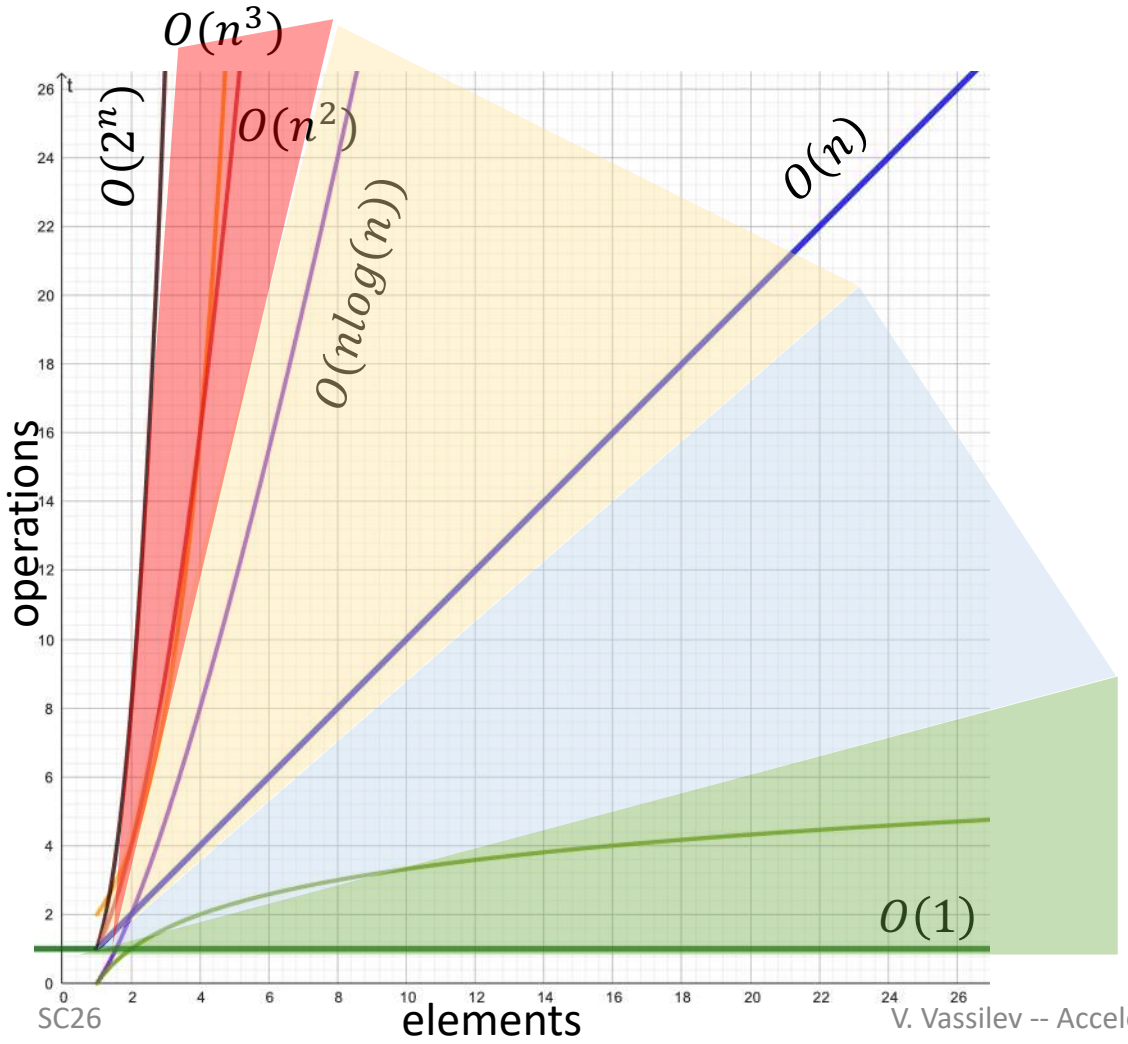




N	Time (s)	RSS (kb)
1M	0.08	11320
2M	0.14	10952
3M	0.21	11172
4M	0.26	11416
5M	0.33	11156
6M	0.39	11180
7M	0.46	11056
8M	0.52	11188
9M	0.58	11088

The Big Picture With Big O

Worst-case algorithm complexity analysis.



Big O is an asymptotic notation which is used to describe the limiting behavior of a function.

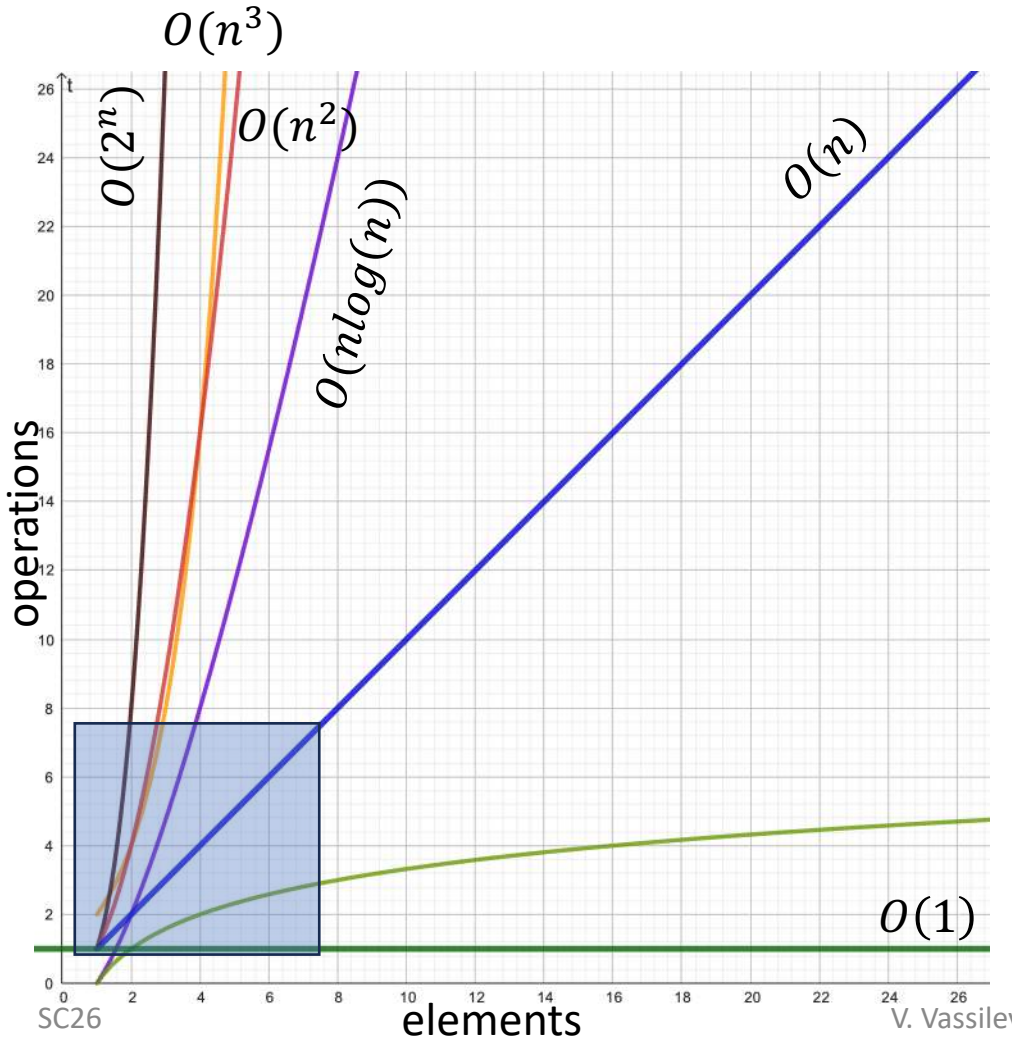
In CS it's used to classify algorithms according to how their run time or space requirements grow as the input size grows.

A very important tool in the programmer's toolbox.

Handout time

The Big Picture With Big O

Worst-case algorithm complexity analysis. O



f(n)	Name	Example
1	Constant	dict/set lookup; array index (Task 6's x in set)
logn	Logarithmic	binary search in sorted data (Task 2 ex2)
n	Linear	1 pass over an array; linear search (Task 2 ex1)
nlogn	Log Linear	any comparison sort; merge sort (Task 3 ex3)
n^2	Quadratic	bubble sort (Task 3 ex1);
n^3	Cubic	naive matrix multiply; triple-nested loop
2^n	Exponential	Halting problem/Generalized Chess, Go
n!	Factorial	Brute forcing travel salesman

Big O. Limitations.

```
list = [1, ..., 63]
list.append(64) # Would that be O(1)?
```



- $O(M*N^2)$ can matter
- Worst-case scenarios happen rarely
- Depending on the properties of the input data the algorithmic complexity can vary

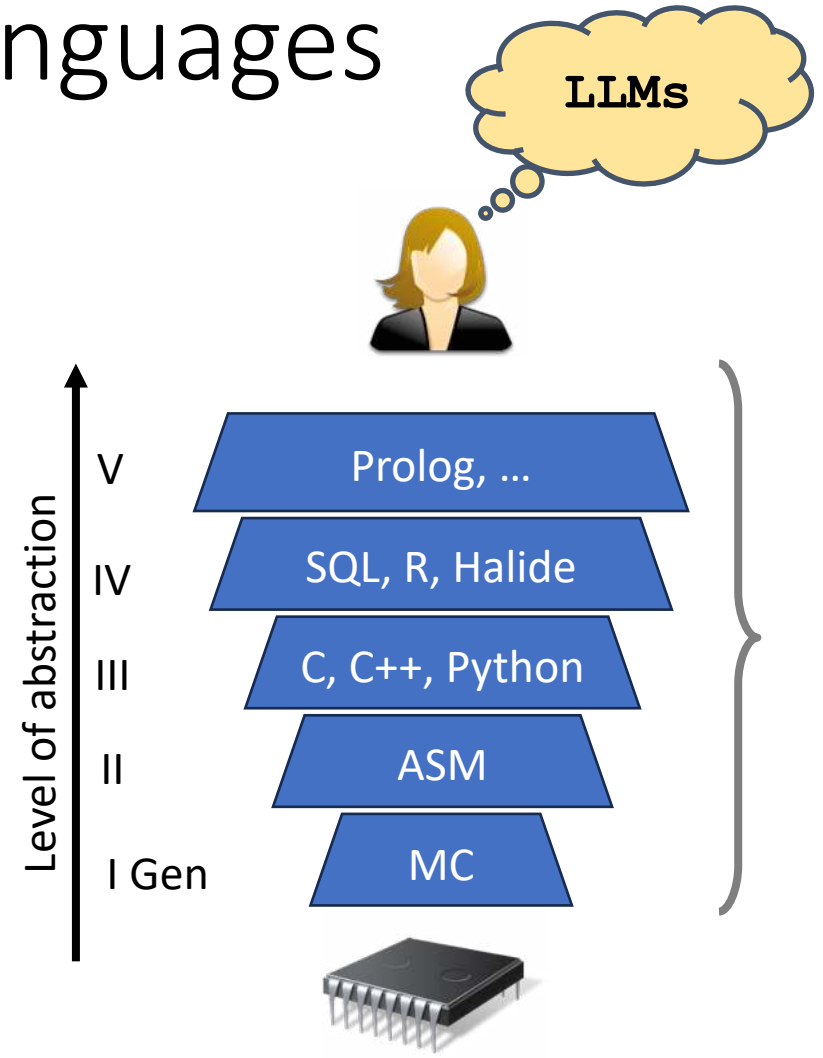
Programming is the art of replacing old bugs with new ones.

- My personal experience

Brief History of Programming Languages

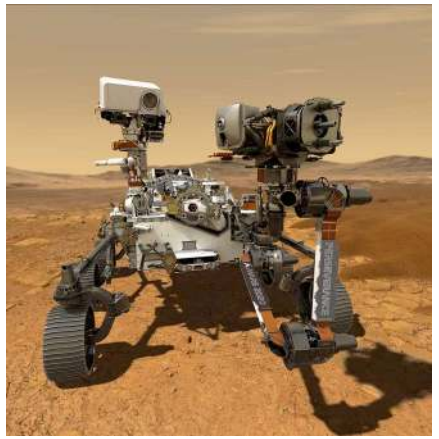
- **1957** Fortran — first compilers (arithmetic expressions, statements, procedures)
- **1960** Algol — first formal definition of PL (BNF grammars, block structure, recursion)
- **1970** Pascal — user-defined types, virtual machines
- **1972** C — structured programming, static type system
- **1985** C++ — object-oriented, exceptions, templates
- **1991** Python — duck typing, ease of use

- ***Important steps in imperative PL**



Know Your Toolchain









C++
- Efficiency
- Stability
- Backward compatibility

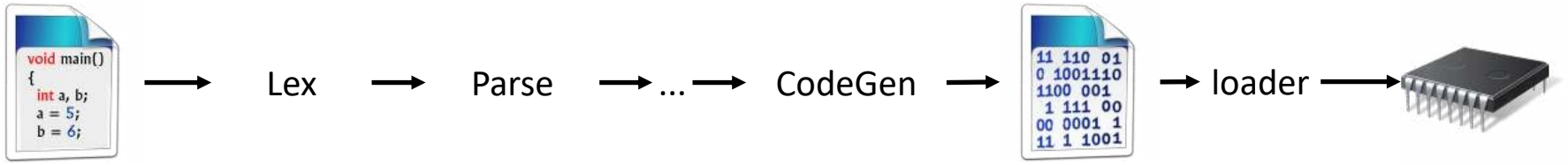
“Prioritizes Performance over Surprise which is sometimes surprising“ T. Winters Link

Python
- Readability
- Simplicity
- Flexibility

“Special cases aren't special enough to break the rules” Zen of Python Link

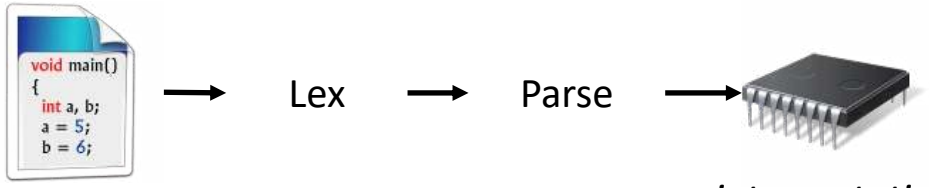
Translator Classification

Compiler translates to machine code



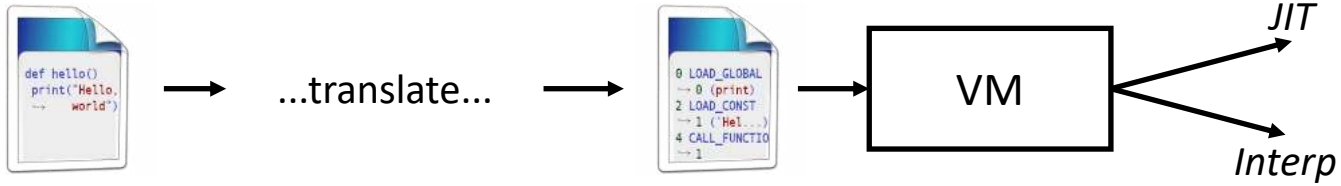
Source *Machine code*

Interpreter executes the source "directly"



Source *interpretation*

Hybrid compiler interprets intermediate representation



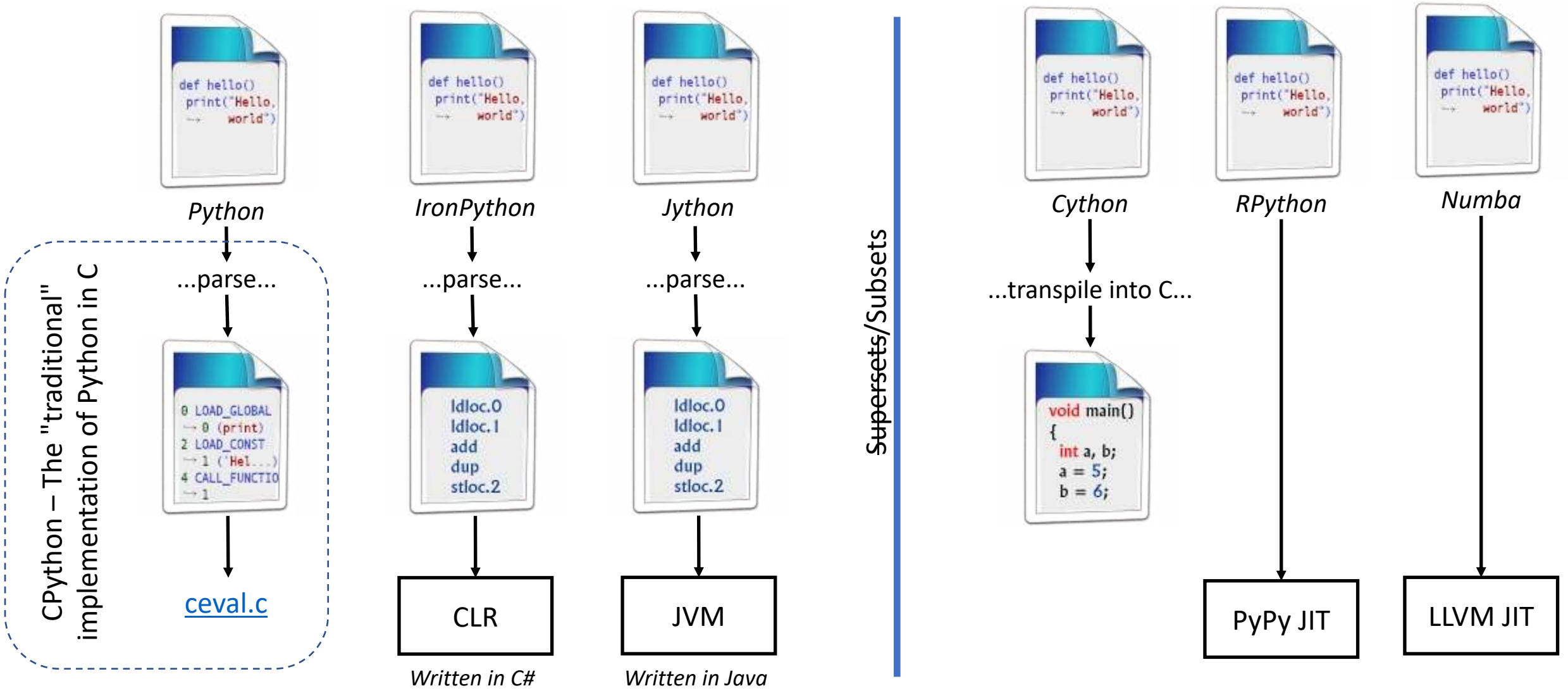
Source *Intermediate Representation (eg. bytecode)*

- The operators in loops are analyzed again and again

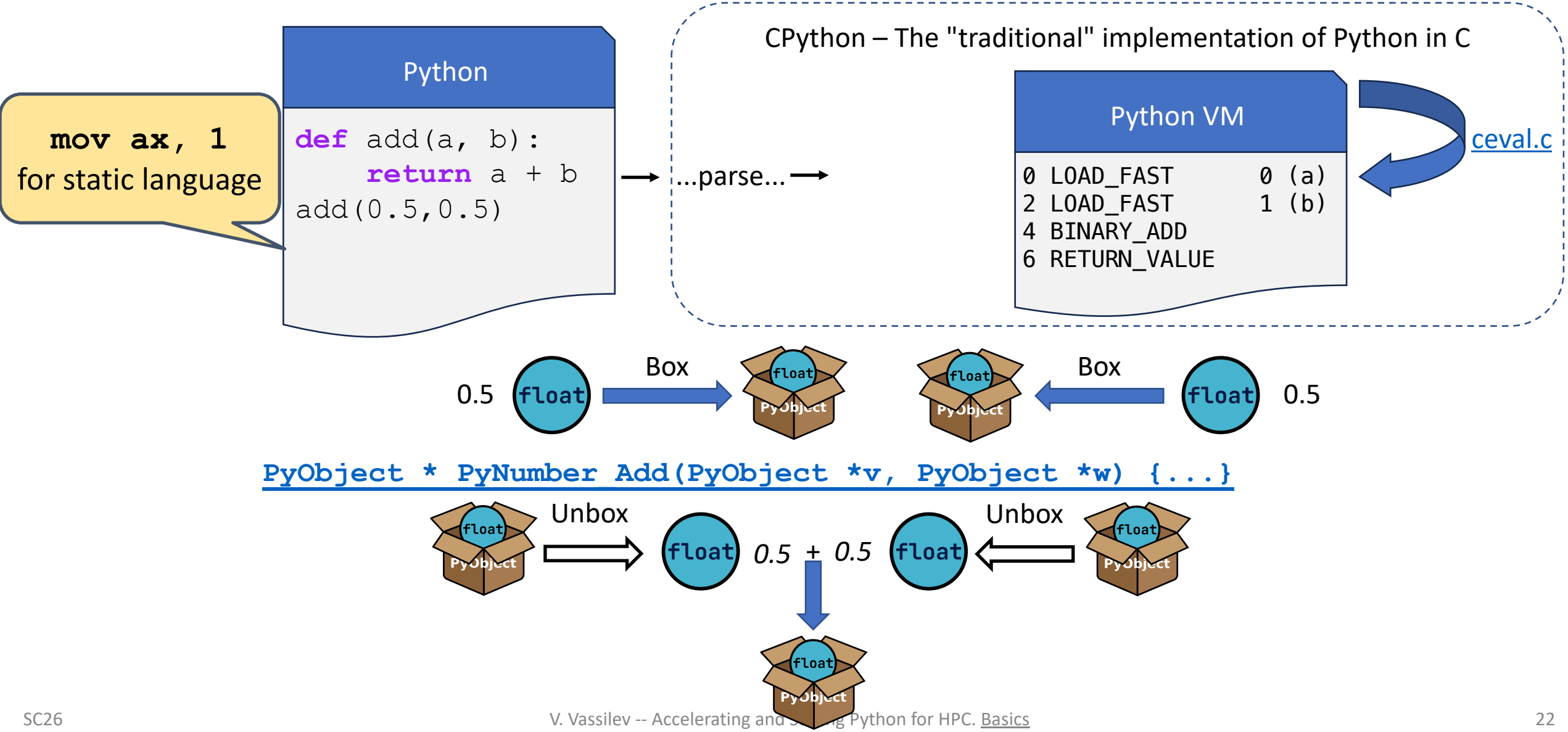
- The source translates to code for virtual machine (VM)
- The VM runtime interprets the code simulating real machine
- The Just-In-Time compiler translates the representation at startup time.

Python Jungle

Trading Flexibility for Performance



Trading Performance For Flexibility



Expressing Optimization Assumptions

- Use built-in functions and libraries
  They are heavily optimized and implemented in C. Eg. use sum() instead of manually iterating
- Avoid global state
  Global variables are slower to access and hinder optimizations
- Minimize the use of loops
  Use map, filter, reduce from libraries such as NumPy
- Use proper data structures
  Choose the correct data structure for the task. Prefer immutable types such as tuple and frozenset.
  Prefer pre-allocated, pre-resized types to avoid amortization effects
- Be mindful with string operations such as concatenation

Expressing Optimization Assumptions

- Avoid excessive object creation
  Think about the created objects especially in tight loops
- Avoid memory leaks in resource release
  Use the `with` construct to manage resources such as files or network connections
- Avoid using try/except constructs for control flow
  Exceptions are designed for error handling and less so for other things
- Avoid unnecessary abstractions
  Abstractions introduce often indirection which can be inefficient
- Use declarative style
  List comprehensions and generator expressions allow sometimes better performance.
- Use memoisation techniques to avoid recomputation
  Cache results with `functools.lru_cache`

YOUR PYTHON — CPython interprets and orchestrates

ecosystem: NumPy · pandas · ML stack · your analysis code

AOT · vertical

Packaged libraries

NumPy · CuPy · mpi4py · Dragon

```
cp.add(x, y)
```

glue: shipped
kernel: shipped
when: before install

AOT · vertical

Bindings you write

ctypes · pybind11 · Cython

```
mylib.fast_add(x, y)
```

glue: you write it
kernel: existing C/C++
when: your build step

JIT · vertical

Compiled Python

Numba — @njit · @cuda.jit · JAX - jax.jit

```
add(x, y) # @njit
```

glue: generated
kernel: Python → LLVM
when: JIT, first call

JIT · horizontal

4 · Runtime interop

cppjit (CppInterOp)

```
cppjit.gbl.add(x, y)
```

glue: built from C++ AST
kernel: C++ cppdef'd live
when: JIT, first use

no shaft of its own — objects, templates, inheritance, callbacks cross both ways

C++ ECOSYSTEM

Eigen · CUDA
30 years of HPC code
your lab's libraries

compiles to native like everything else

COMPILED NATIVE CODE — kernels run with the GIL released

CPU SIMD · GPU (CUDA) · interconnect (MPI) · cluster processes

“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; *premature optimization is the root of all evil (or at least most of it) in programming*.”

- Donald Knuth



NumPy: Array Programming

```
import numpy as np

# Riemann sum of ∫₀¹x²dx
x = np.linspace(0, 1, 100)
y = x**2
dx = x[1] - x[0]
I = np.sum(y) * dx
```

The de facto standard for array data in Python.

Every HPC and ML library references NumPy in some way!

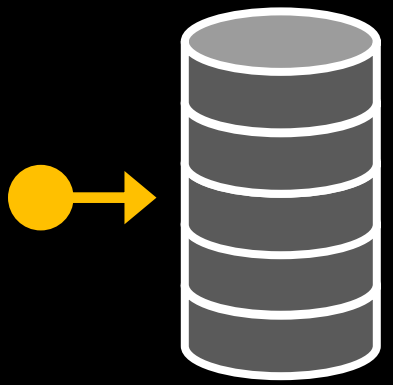
NumPy specifies:

- `ndarray`: Multidimensional array type.
- Array indexing and slicing.
- A common set of array functions.
- How to define your own array functions.

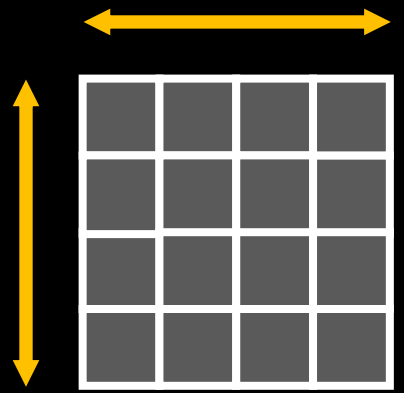
ndarray



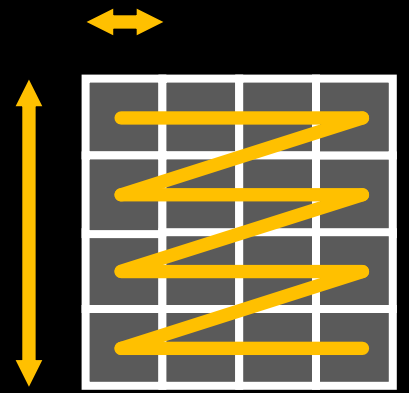
dtype: the data type of all elements of this array.
Ex: `int32`



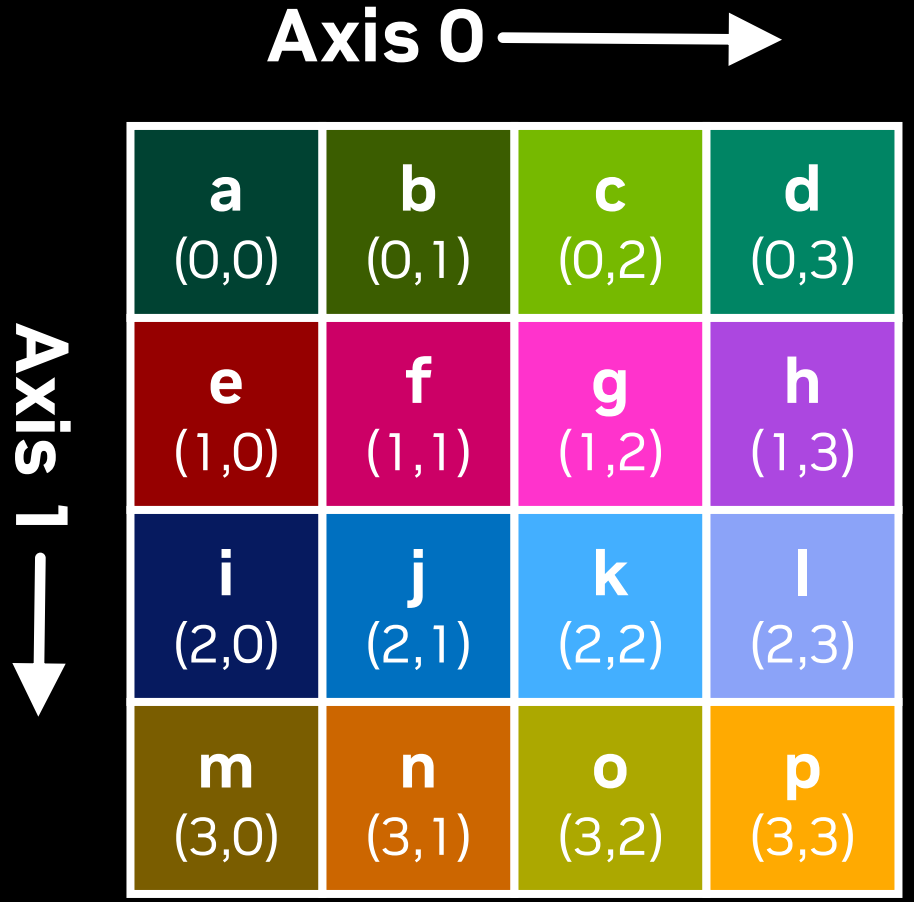
data: a pointer to the block of memory holding all the elements in this array.



shape: the number of dimensions and the size of each dimension.
Ex: `(4, 3)`



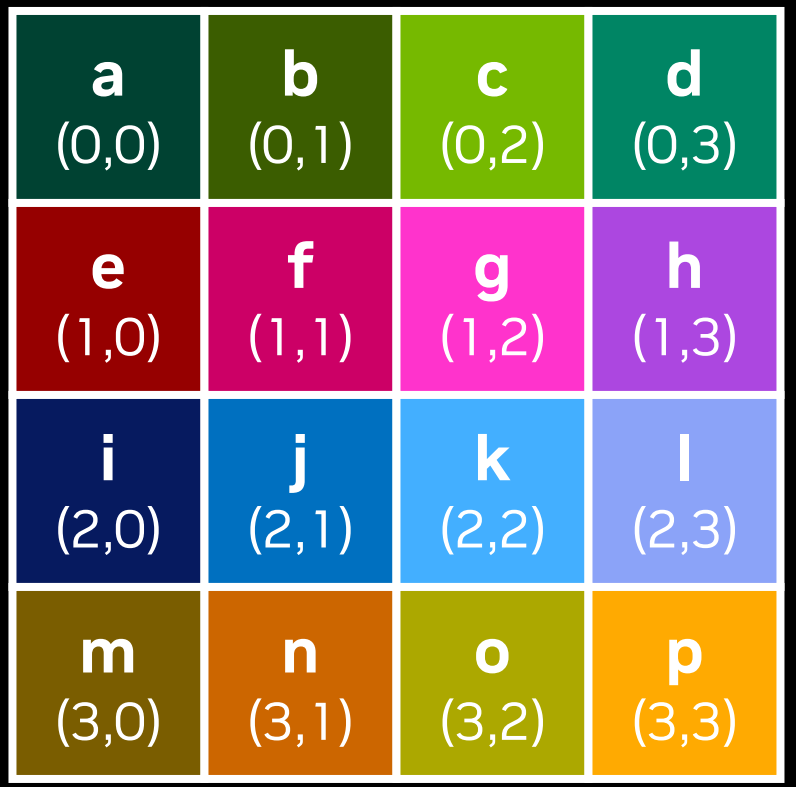
strides: the number of bytes between elements along each dimension.
Ex: `(12, 4)`



Each axis of the array can have whatever meaning makes sense for your situation:

- 1D - temperature data: [time step]
- 2D - greyscale image: [row, col]
- 3D - color image: [row, col, rgb]
- 4D - video: [time, row, col, rgb]
- 4D - image stack: [img#, row, col, rgb]

Axis ordering and memory layout are independent!



Physical	Logical	Value
0	(0,0)	a
1	(0,1)	b
2	(0,2)	c
3	(0,3)	d
4	(1,0)	e
5	(1,1)	f
6	(1,2)	g
7	(1,3)	h
8	(2,0)	i
9	(2,1)	j
10	(2,2)	k
11	(2,3)	l
12	(3,0)	m
13	(3,1)	n
14	(3,2)	o
15	(3,3)	p

a (0,0)	**b** (0,1)	**c** (0,2)	**d** (0,3)
e (1,0)	**f** (1,1)	**g** (1,2)	**h** (1,3)
i (2,0)	**j** (2,1)	**k** (2,2)	**l** (2,3)
m (3,0)	**n** (3,1)	**o** (3,2)	**p** (3,3)

Physical	Logical	Value
0	(0,0)	a
1	(1,0)	e
2	(2,0)	i
3	(3,0)	m
4	(0,1)	b
5	(1,1)	f
6	(2,1)	j
7	(3,1)	n
8	(0,2)	c
9	(1,2)	g
10	(2,2)	k
11	(3,2)	o
12	(0,3)	d
13	(1,3)	h
14	(2,3)	l
15	(3,3)	p

Storage

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

```
X = np.ndarray((4,4))
```

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

```
np.transpose(X)
X.transpose()
X.T
```

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

Views of the original data, *not a copy*

```
np.reshape(X, (2,8))
X.reshape((2,8))
```

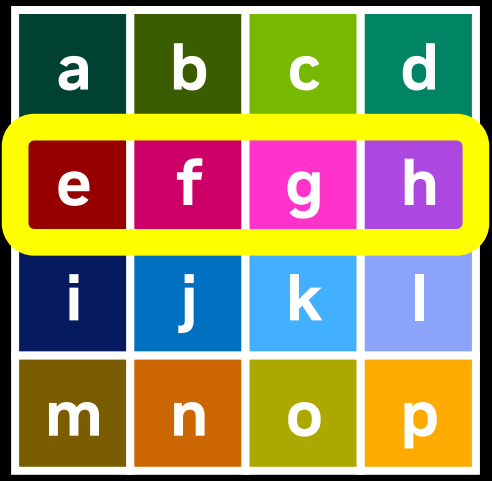
a	b
c	d
e	f
g	h
i	j
k	l
m	n
o	p



`X[1, 2]`

0D (Scalar)





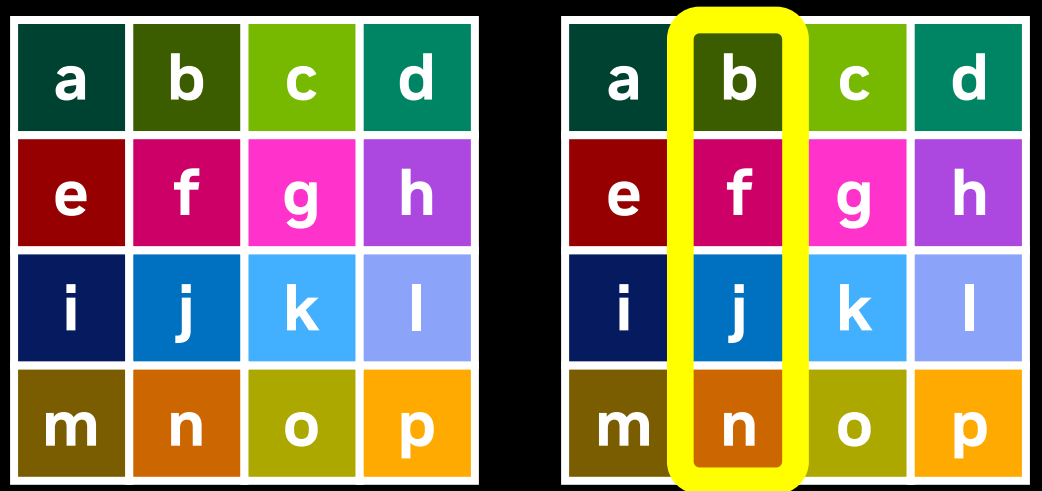
`X[1, 2]`

0D (Scalar)

`X[1]`

1D

Views of the original data, *not a copy*



`X[1, 2]`

0D (Scalar)

`X[:, 1]`

1D

Views of the original data, *not a copy*



`X[1, 2]`

0D (Scalar)

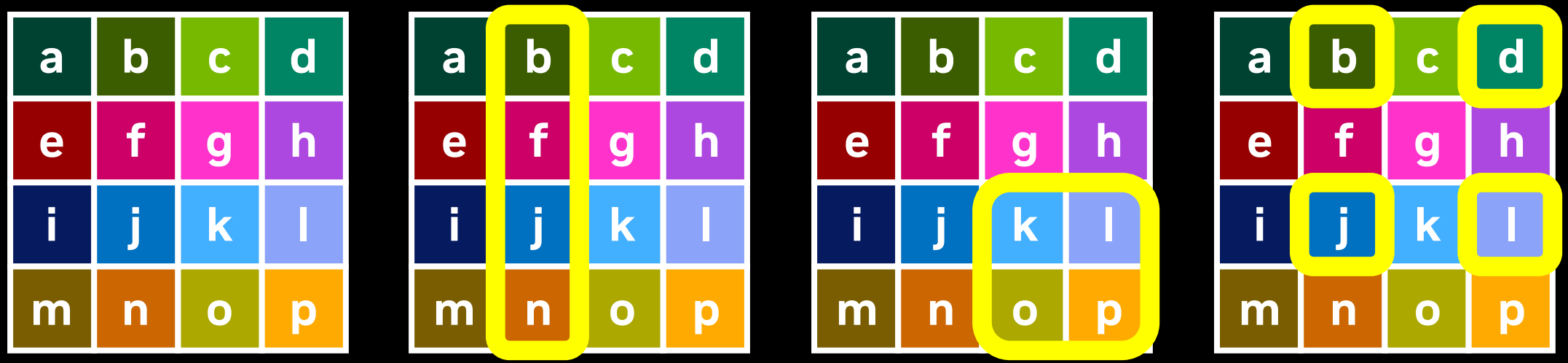
`X[:, 1]`

1D

`X[2:, 2:]`

2D

Views of the original data, *not a copy*



`X[1, 2]`

0D (Scalar)

`X[:, 1]`

1D

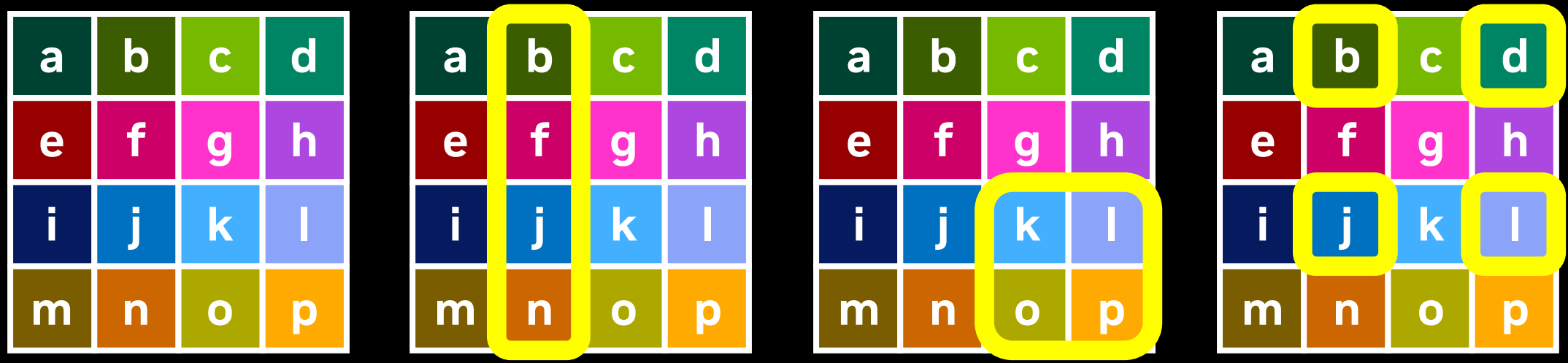
`X[2:, 2:]`

2D

`A[::2, 1::2]`

2D

Views of the original data, *not a copy*



`X[1, 2]`

0D (Scalar)

`X[:, 1]`

1D

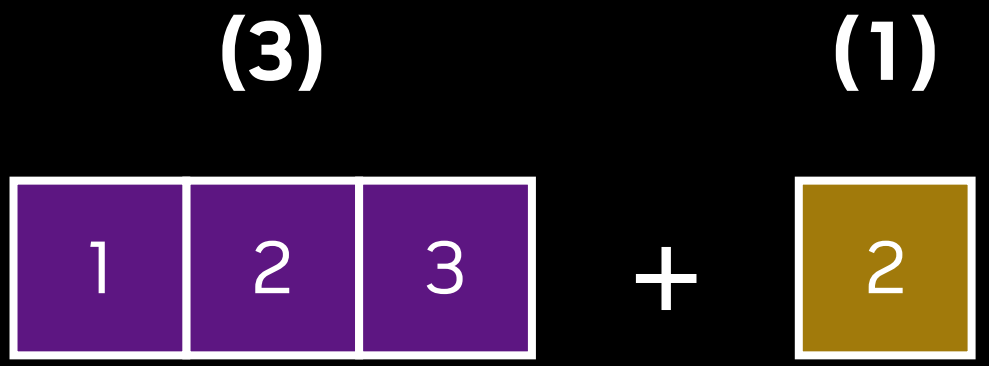
`X[2:, 2:]`

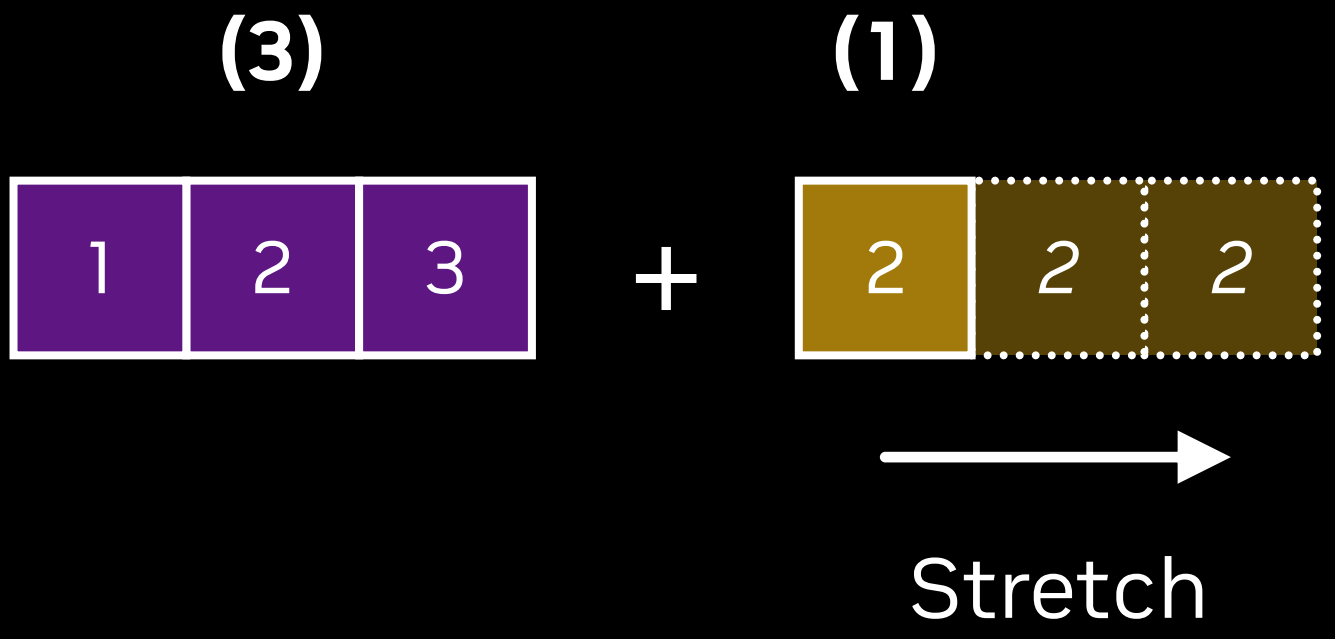
2D

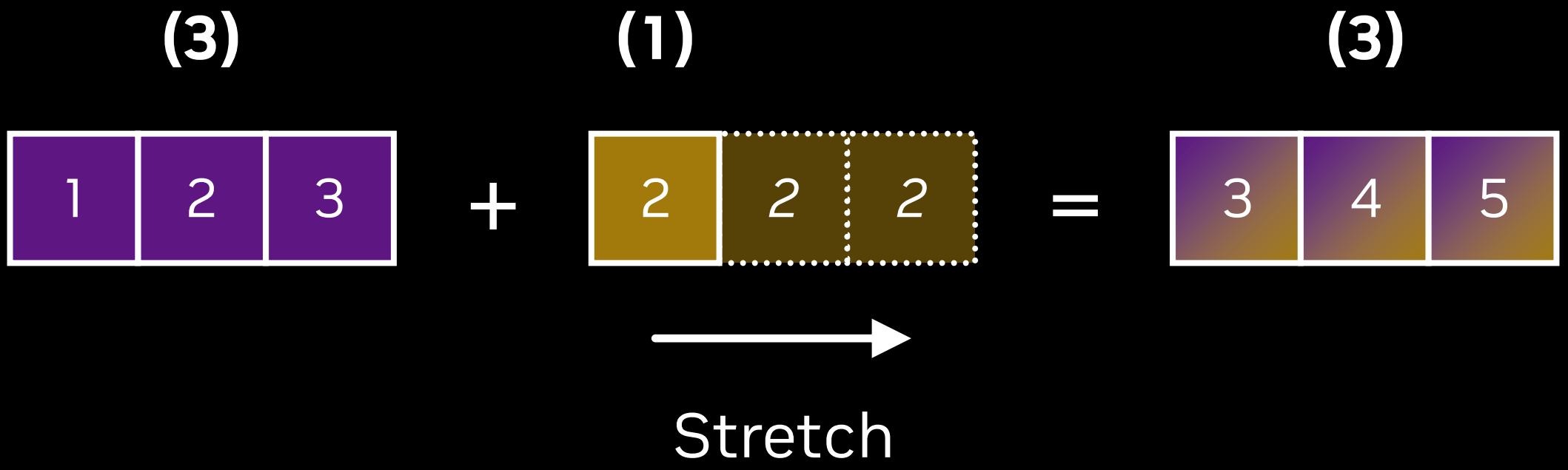
`A[::2, 1::2]`

2D

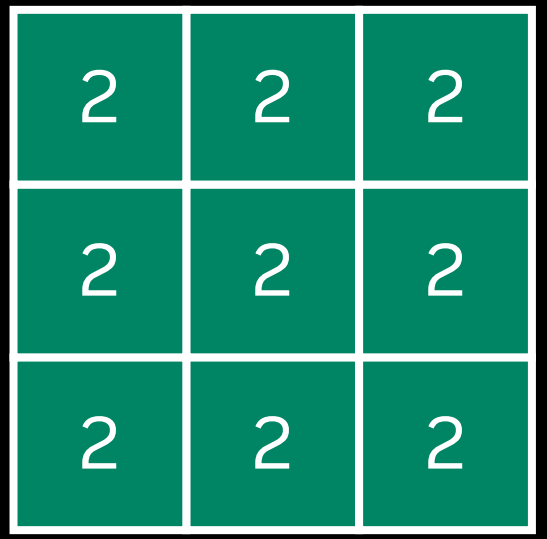
`start:stop:step`


(3)
(1)
1
2
3
+
2


(3)
(1)
1
2
3
+
2
2
2
Stretch

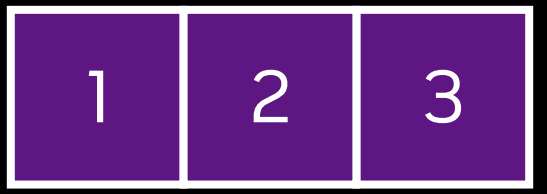

(3)
(1)
(3)
1
2
3
+
2
2
2
=
3
4
5
Stretch

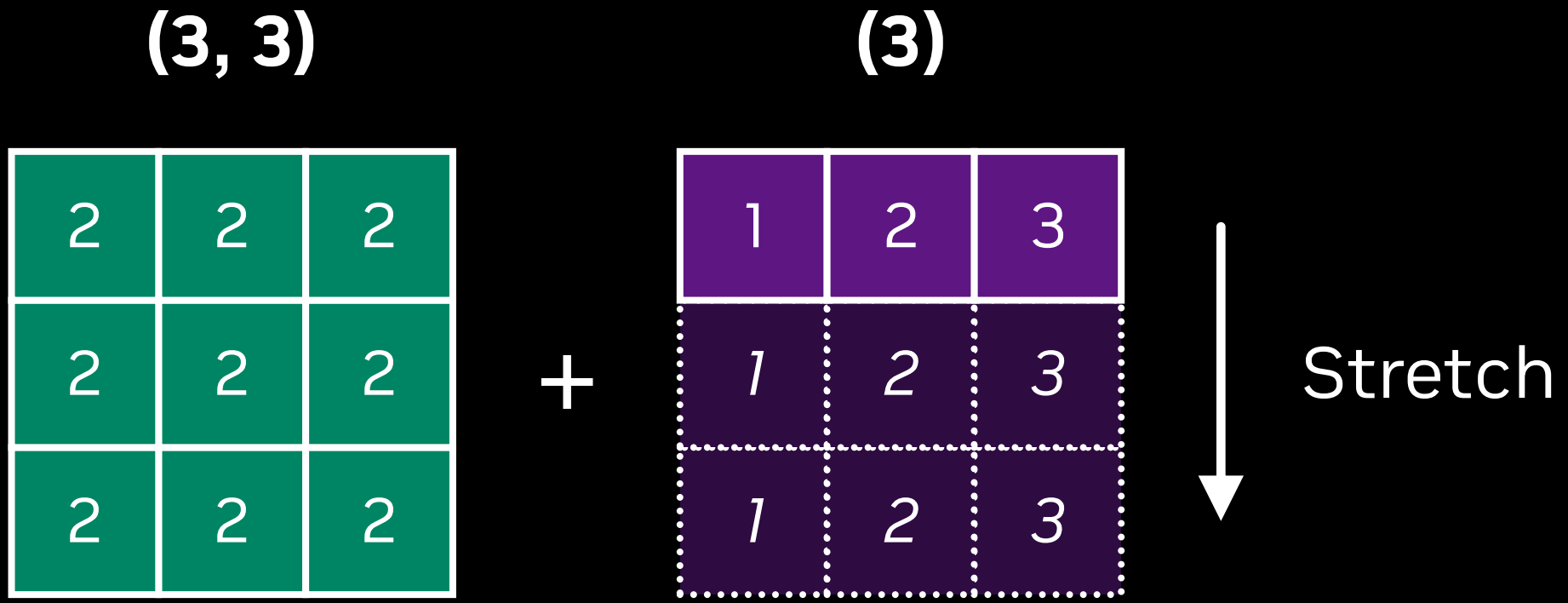
(3, 3)



+

(3)

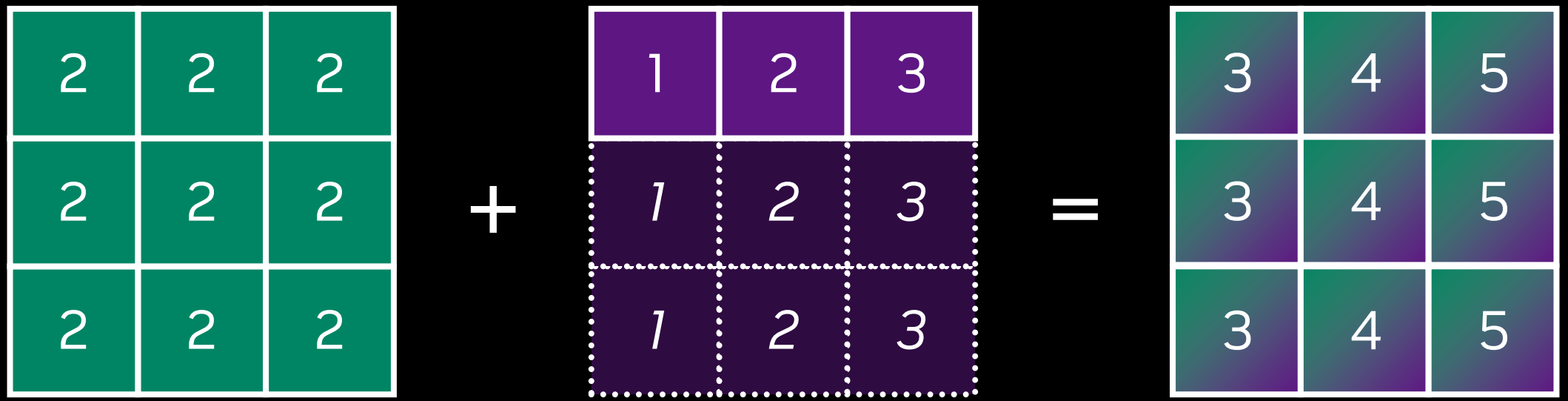



(3, 3)
(3)
2 2 2
2 2 2
2 2 2
+
1 2 3
1 2 3
1 2 3
Stretch

(3, 3)

(3)

(3, 3)



(3, 2)

-1	1
1	-1
-1	1

+

(3)

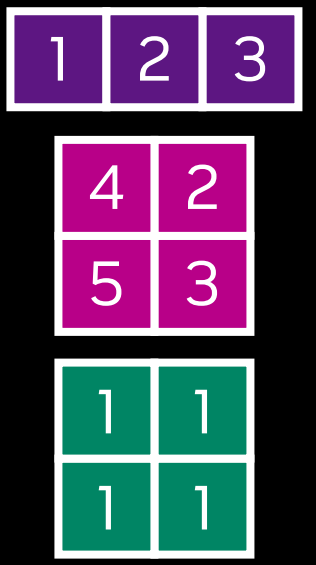
1	2	3

=

`ValueError:`
operands could not be broadcast together with shapes (3,2) (3,)

Array Functions apply operations to the contents of arrays

```
A = np.arange(1, 3)

B = np.array([[4, 2],
              [5, 3]])

C = np.ones((2, 2),
      dtype=np.int32)
```

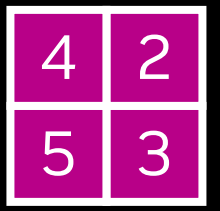


Some create new arrays. Specify the type with the dtype parameter.

Array Functions apply operations to the contents of arrays

```
A = np.arange(1, 3)

B = np.array([[4, 2],
              [5, 3]])

C = np.ones((2, 2),
      dtype=np.int32)
```

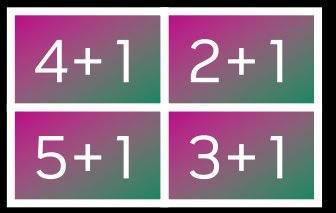






Some create new arrays. Specify the type with the dtype parameter.

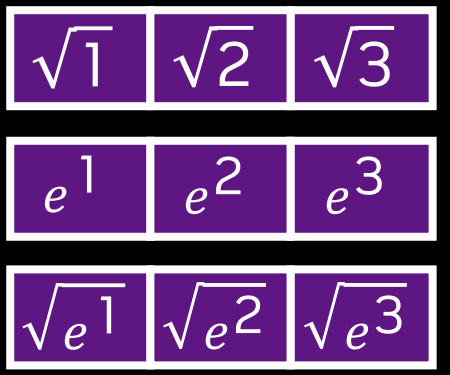
```
D = np.add(B, C)

D = B + C
```



Arithmetic can be done with operators.

Array Functions apply operations to the contents of arrays

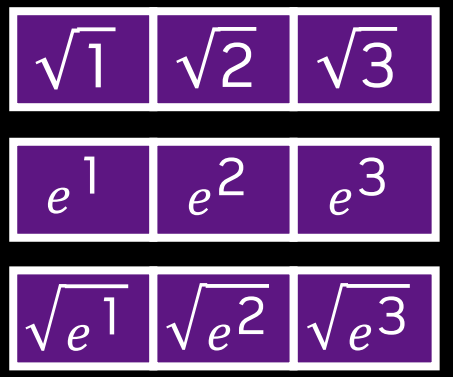
```
F = np.sqrt(A)
G = np.exp(A)

np.sqrt(A, out=A)
np.exp(A, out=A)
```



Most are elementwise and return a new array. Use the out parameter to avoid an allocation and/or operate inplace.

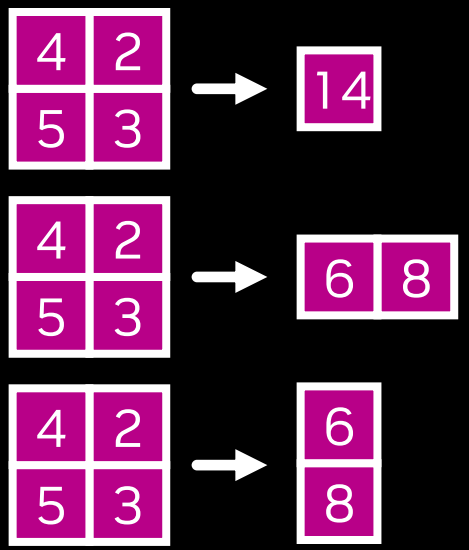
Array Functions apply operations to the contents of arrays

```
F = np.sqrt(A)
G = np.exp(A)

np.sqrt(A, out=A)
np.exp(A, out=A)
```



Most are elementwise and return a new array. Use the out parameter to avoid an allocation and/or operate inplace.

```
X = np.sum(A))

Y = np.sum(A, axis=1))

Z = np.sum(A, axis=1,
       keepdims=True))
```



Some can be applied to a certain axis and may reduce rank. Use the keepdims parameter to preserve the rank.

Array Functions are faster than for loops

```
A = np.random.random((1024, 1024))

def loop(A):
  for i in range(A.shape[0]):
    for j in range(A.shape[1]):
      A[i,j] = A[i,j] + A[i,j]**2

def array_ops(A):
  return A + A**2

%timeit loop(A)
%timeit array_ops(A)
```

Array functions are implemented with optimized native code in a language like C.

689 ms ± 8.76 ms per loop
(mean ± std. dev. of 7 runs, 1 loop each)
2.63 ms ± 98.7 µs per loop
(mean ± std. dev. of 7 runs, 100 loops each)

Google Colab T4 VM, Python 3.11

Exercise 01

NumPy Intro

ndarray Basics

Array Creation
(`np.empty, np.zeros, np.arange, …`)

Linear Algebra
(`np.dot, np.linalg.eig, …`)

Indexing / Slicing
(`np.take, np.put, np.choose, …`)

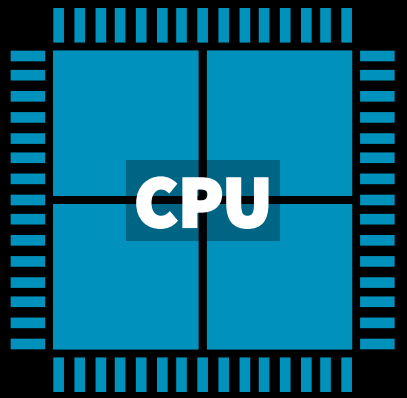
Input / Output
(`np.loadtxt, np.save, …`)

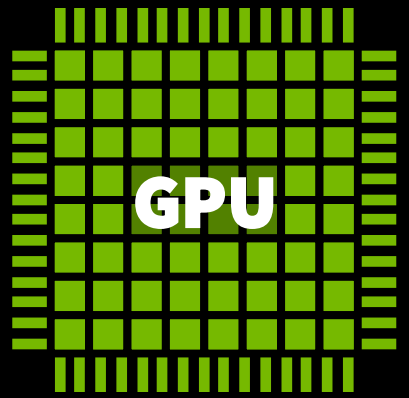
Random Numbers & Sampling
(`np.random.rand, np.random.normal,…`)

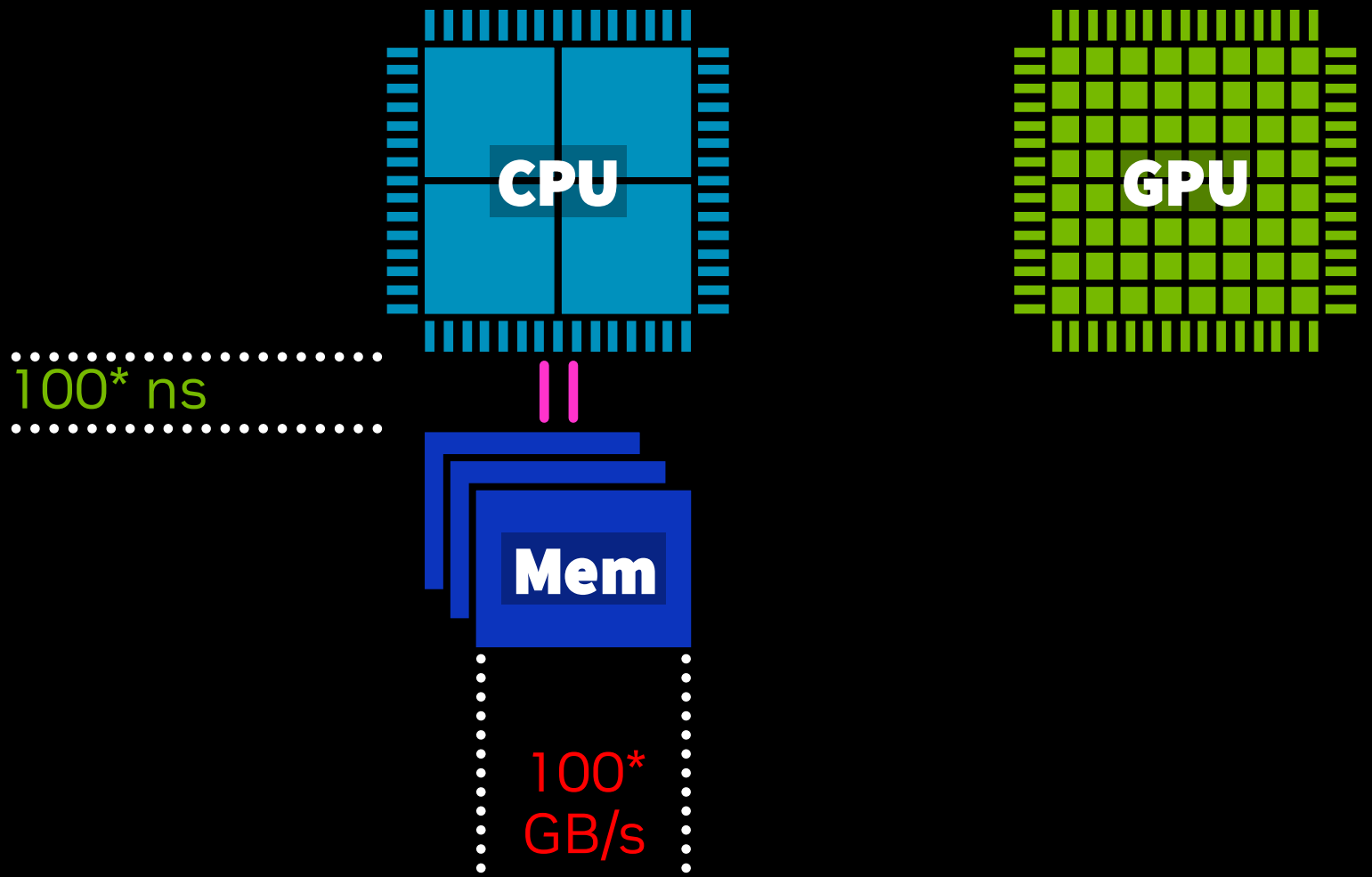
Shape Manipulation
(`np.transpose, np.hstack, …`)

Math / Logic Functions
(`np.exp, np.sin, np.isnan, …`)

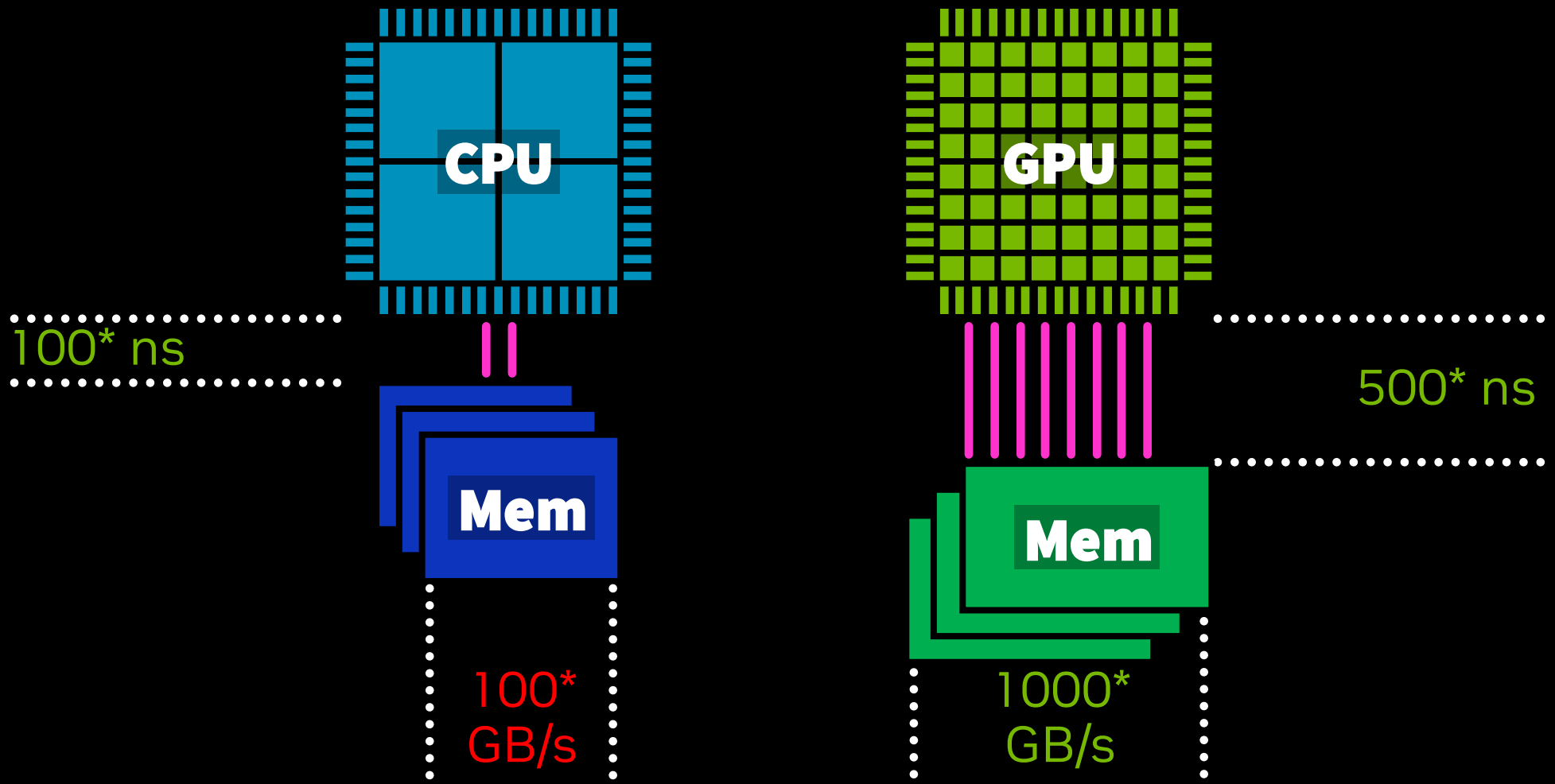
…and many more categories
(and SciPy has *even more*)


CPU


GPU



Numbers are made up and
for expository purposes only.

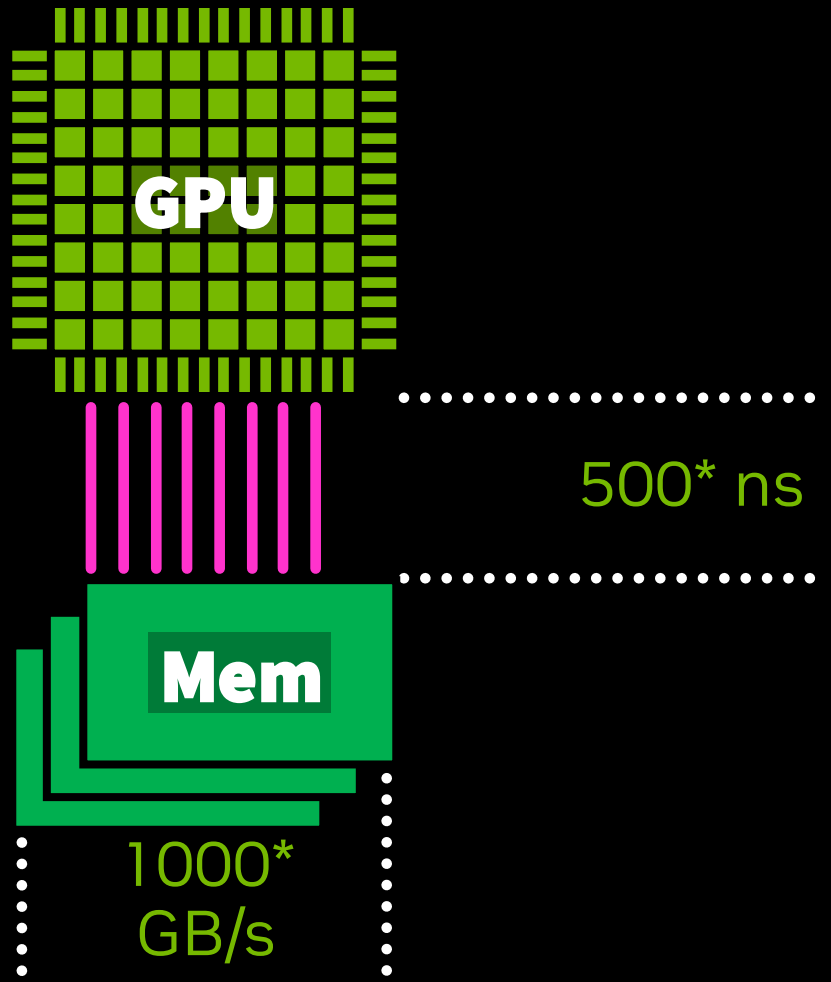


Numbers are made up and
for expository purposes only.

You need a sufficient workload to benefit from GPUs.

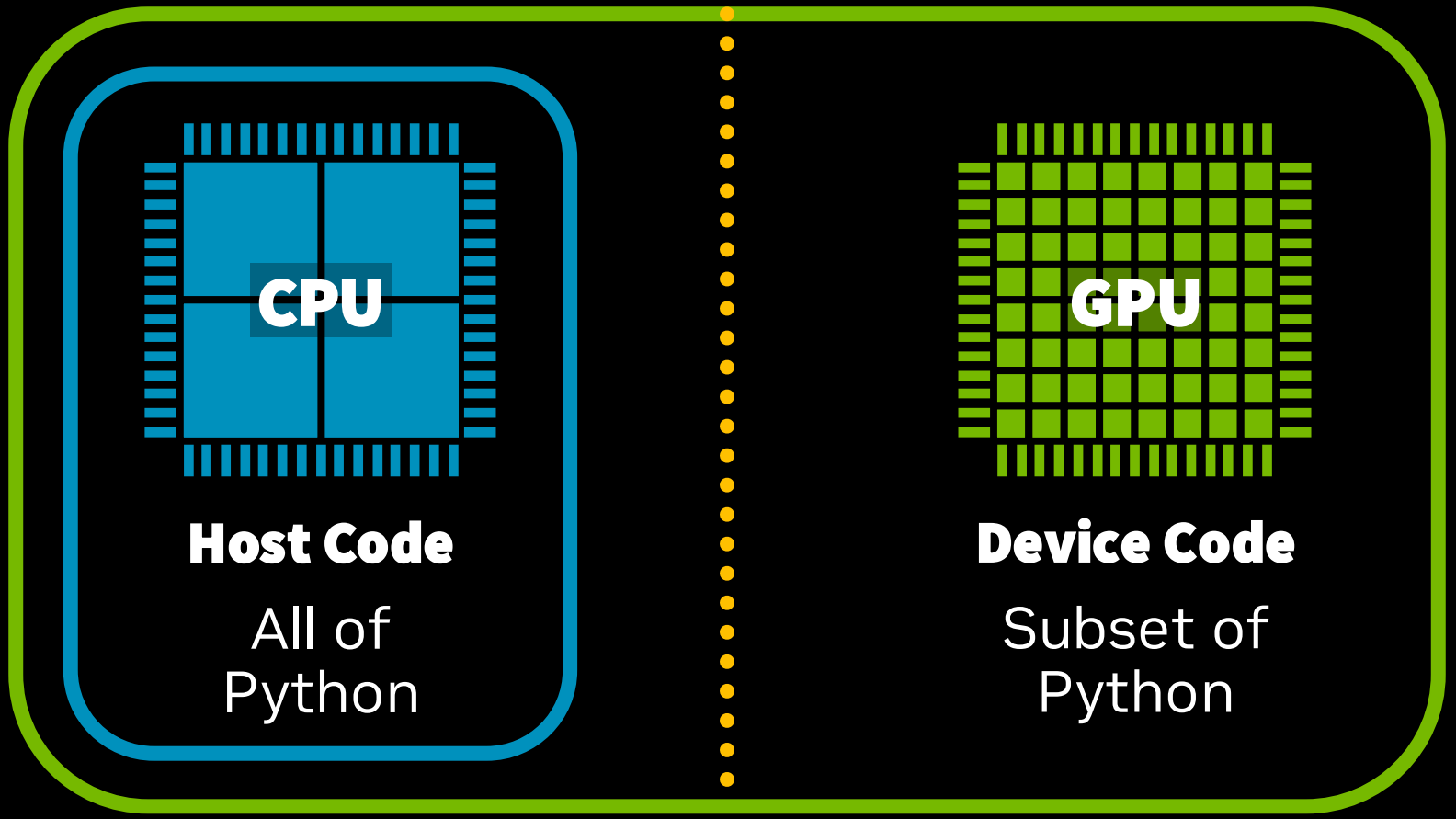
O(N) compute => GBs of data.

More than O(N) compute => less data.



Numbers are made up and for expository purposes only.

CUDA Python is an extension of Python for writing programs that simultaneously run on CPUs (hosts) & GPUs (devices).





NumPy: Array Programming

```
import numpy as np

# Riemann sum of ∫₀¹x²dx
x = np.linspace(0, 1, 100)
y = x**2
dx = x[1] - x[0]
I = np.sum(y) * dx
```

The de facto standard for array data in Python.

Every HPC and ML library references NumPy in some way!

NumPy specifies:

- `ndarray`: Multidimensional array type.
- Array indexing and slicing.
- A common set of array functions.
- How to define your own array functions.



CuPy: NumPy API for GPUs

- A NumPy / SciPy library that runs *on the GPU.*
- 80/90% compatible; we'll learn some of the limitations today.
- Handles allocating memory on the GPU & running GPU kernels for you.
- Open source, NVIDIA CUDA & AMD ROCm backends.

```
import numpy as np

current_state = (np.random.uniform(size=1000*1000) > 0.8).reshape((1000,1000))

def life_step_array_func(X):
    nbrs_count = sum(np.roll(np.roll(X, i, 0), j, 1)
                     for i in (-1, 0, 1) for j in (-1, 0, 1)
                     if (i != 0 or j != 0))
    return (nbrs_count == 3) | (X & (nbrs_count == 2))

%timeit life_step_array_func(current_state)
```

```
11.1 ms ± 11.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
import cupy as cp

current_state = (cp.random.uniform(size=1000*1000) > 0.8).reshape((1000,1000))

def life_step_array_func_cupy(X):
    nbrs_count = sum(cp.roll(cp.roll(X, i, 0), j, 1)
                     for i in (-1, 0, 1) for j in (-1, 0, 1)
                     if (i != 0 or j != 0))
    return (nbrs_count == 3) | (X & (nbrs_count == 2))

%timeit life_step_array_func_cupy(current_state)
```

```
955 µs ± 1.95 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Nearly identical code, **10x faster** (Quadro RTX 8000)

Exercise 03

NumPy to CuPy

ndarray Basics

CuPy Isn't *Always* Just Like NumPy

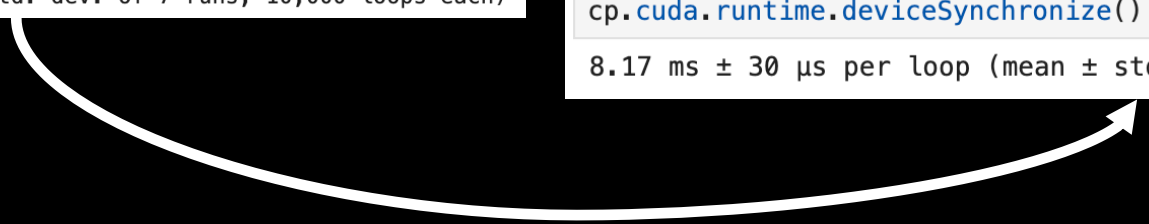
- Not all NumPy / SciPy functions exist (but most do).
- Structured object, and string dtypes are not supported.
- Out of bounds access wraps around back to the beginning of the array.
- Memory transfers are necessary:
  - Arrays from NumPy and libraries like OpenCV need to be sent to the GPU to use with CuPy.
  - CuPy arrays need to be brought back to the CPU to use with other libraries, like matplotlib.
- Different promotion rules.
  - NumPy is safe even if slow, CuPy prefers speed and avoids implicit slowness.
- Not bit-for-bit equivalent.
  - Some algorithms like pseudo-random number generation give different results.

Avoid Serial For Loops!

```
a_cpu = np.zeros(1000, dtype=np.float32)
```

```
%%timeit
for i in range(a_cpu.shape[0]):
    a_cpu[i] = i
```

```
82.4 µs ± 220 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```



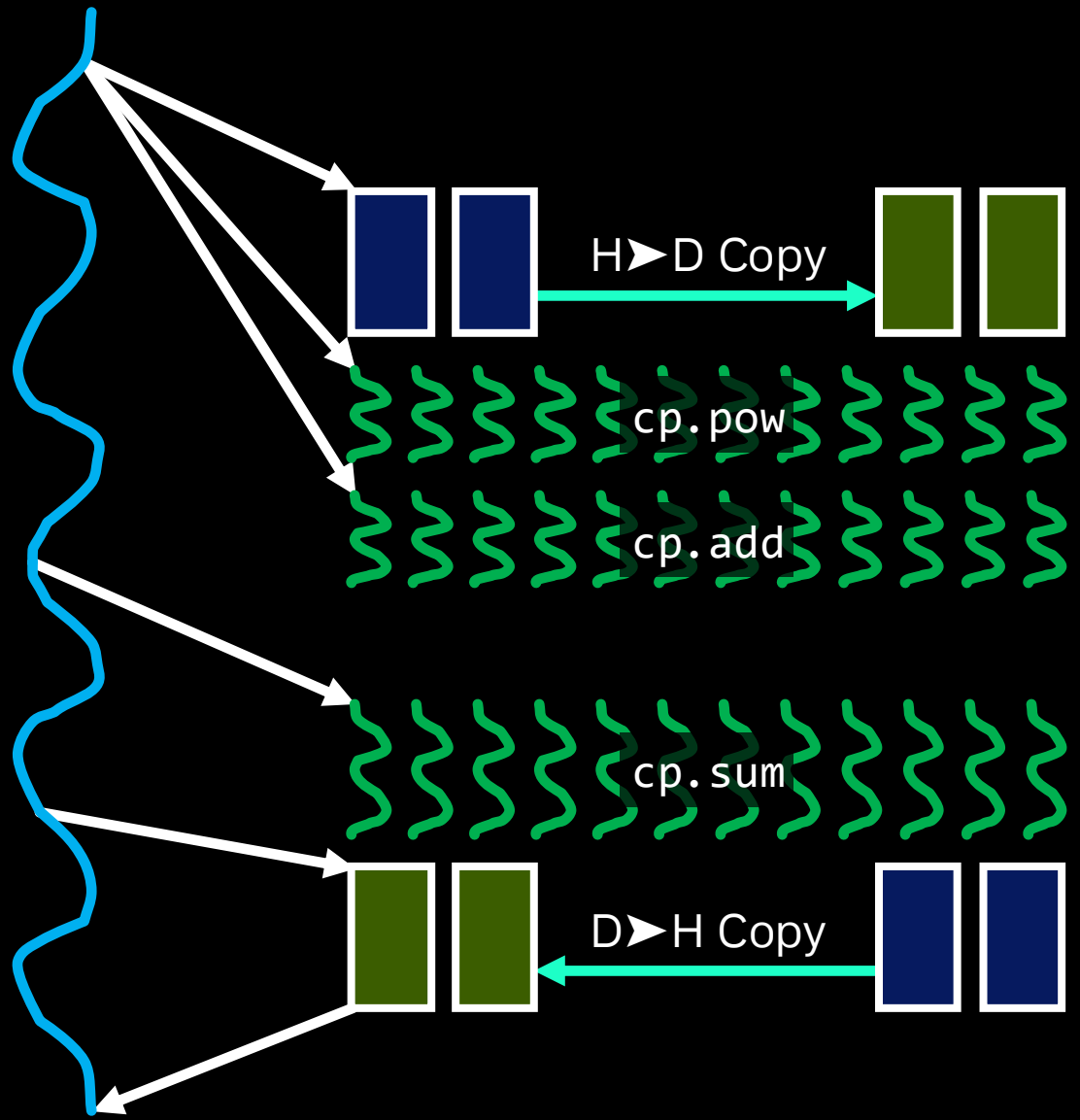
```
a_gpu = cp.zeros(1000, dtype=cp.float32)
```

```
%%timeit
for i in range(a_gpu.shape[0]):
    a_gpu[i] = i

## need to make sure GPU code is done for benchmarking purposes
cp.cuda.runtime.deviceSynchronize()
```

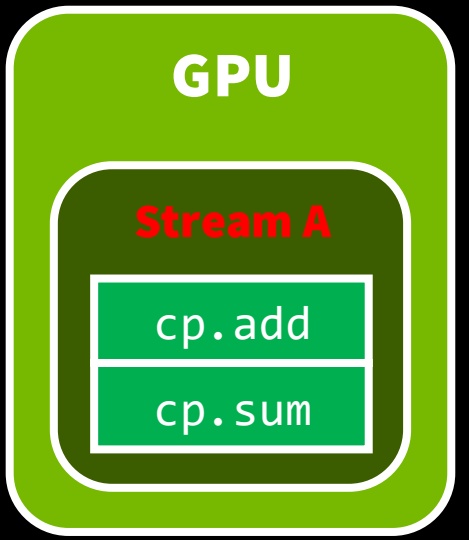
```
8.17 ms ± 30 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Elementwise operations on arrays are **100x worse** on the GPU than the CPU.

```
A = cp.asarray(…)

A = A + A**2

do_cpu_stuff()

x = A.sum(axis=1)

do_cpu_other_stuff()

x_host = cp.asnumpy(x)
```

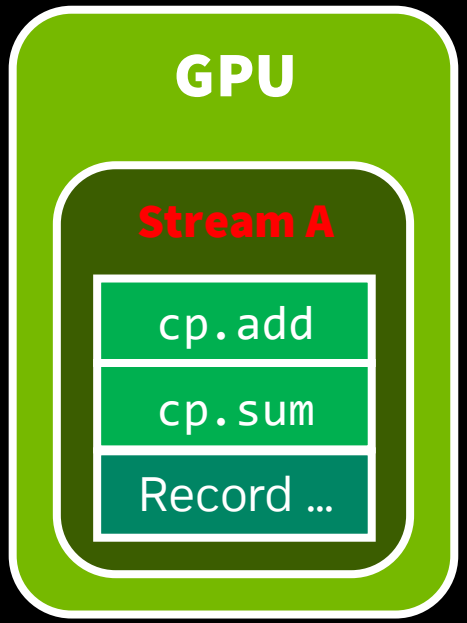


```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  cp.sum(…)
```



A stream is a chain of device work that is executed in order.

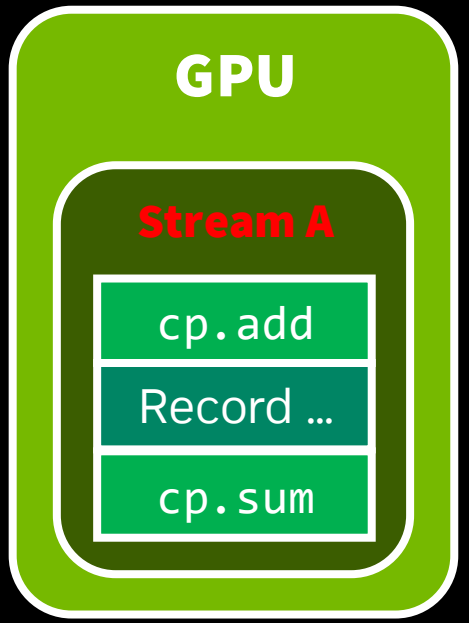
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  cp.sum(…)
  A.synchronize() # Host waits.
```



A stream is a chain of device work that is executed in order.

They can record events that host code can sync with.

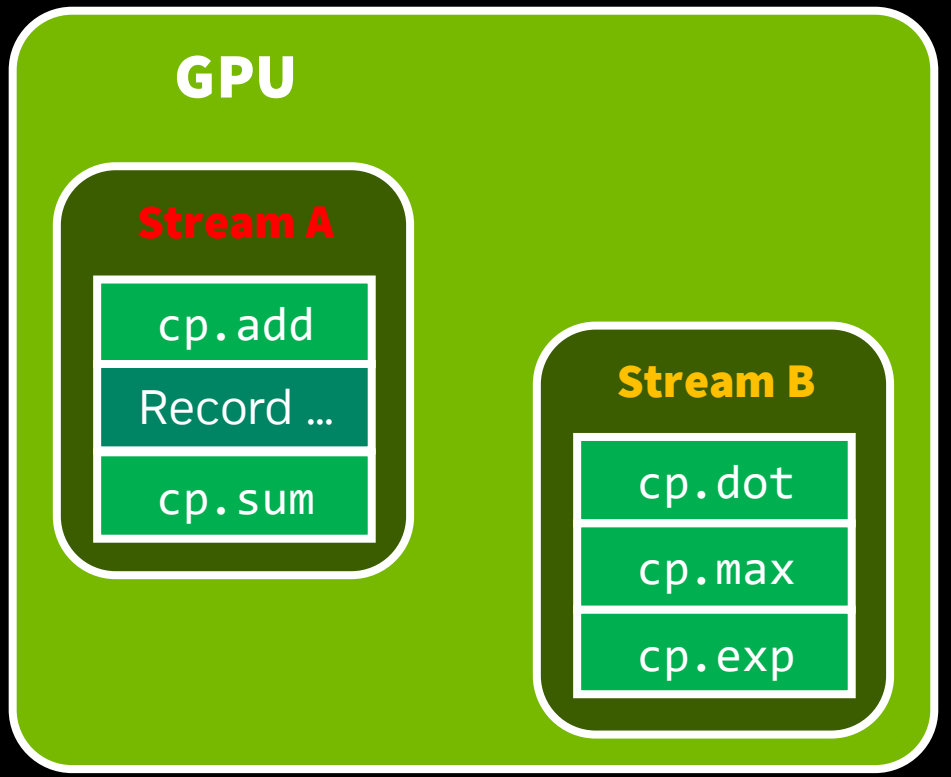
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)
```



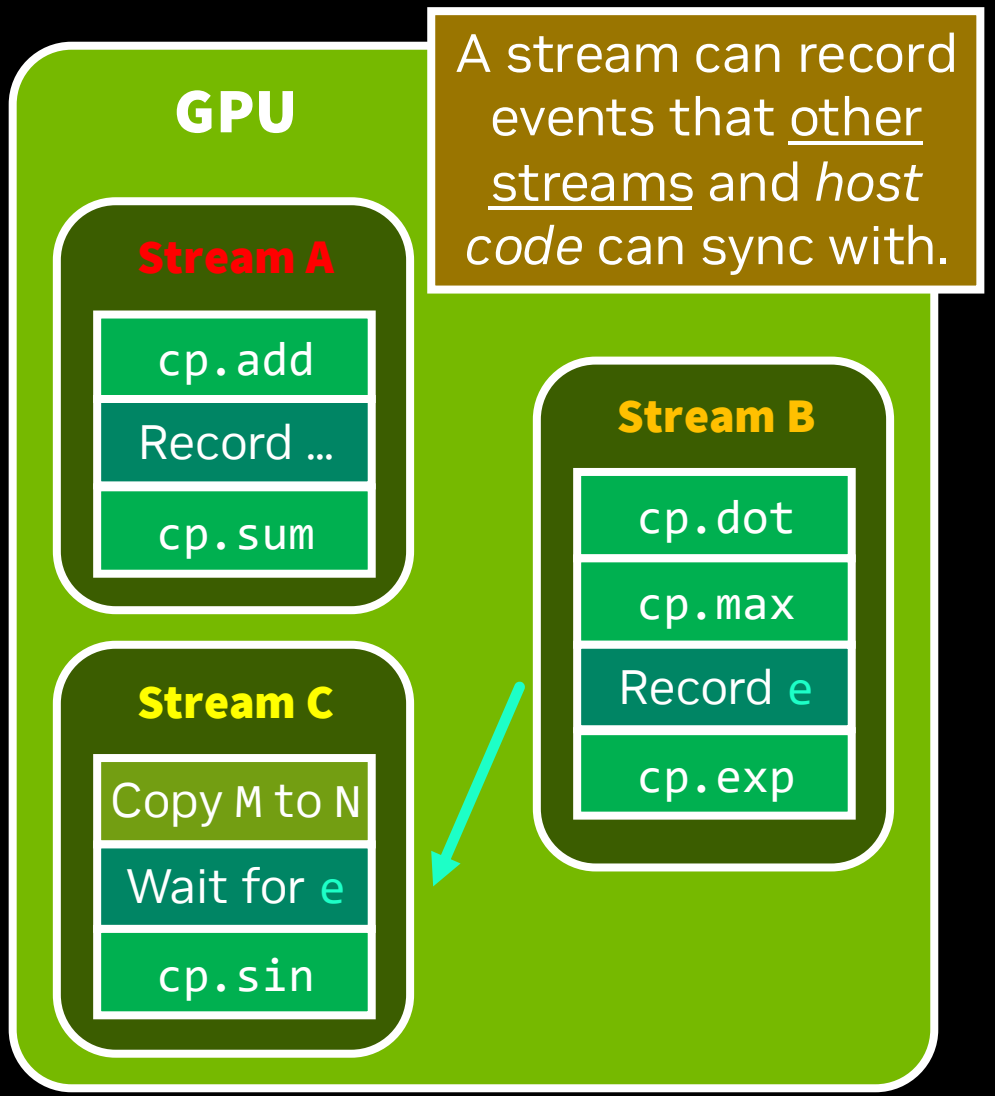
A stream is a chain of device work that is executed in order.

They can record events that host code can sync with.

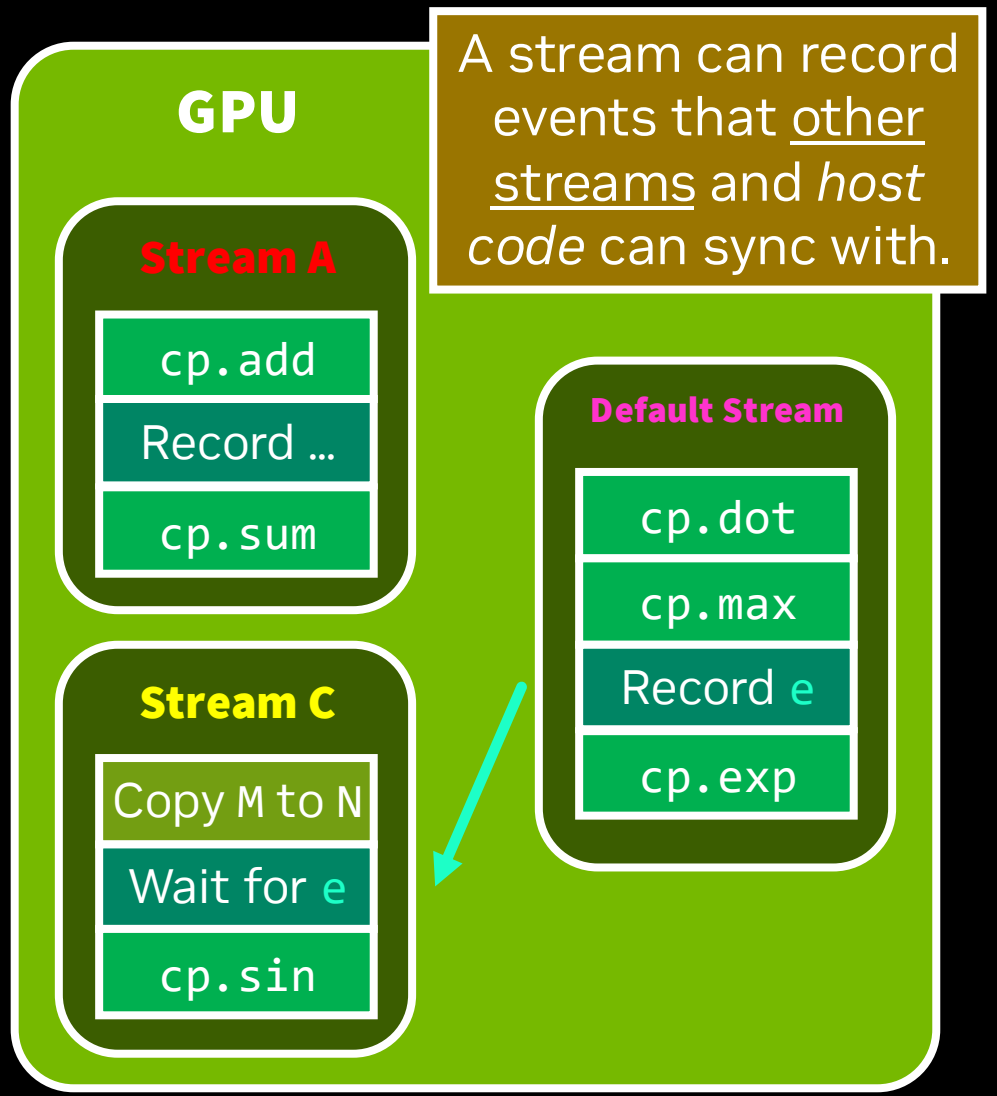
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)

B = cp.cuda.Stream(non_blocking=True)
with B:
  cp.dot(…)
  cp.max(…)
  cp.exp(…)
```



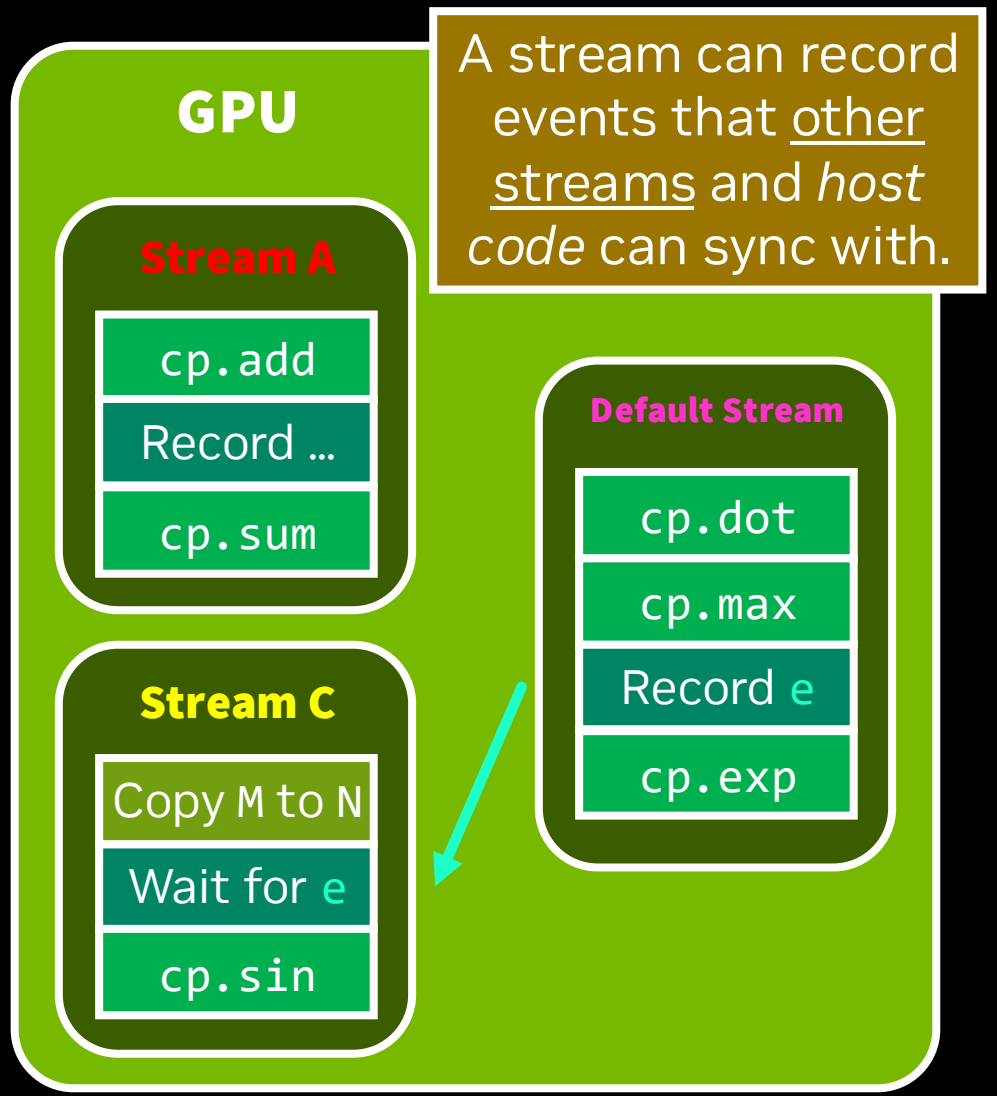
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)

B = cp.cuda.Stream(non_blocking=True)
with B:
  cp.dot(…)
  cp.max(…)
  e = B.record()
  cp.exp(…)

C = cp.cuda.Stream(non_blocking=True)
with C:
  N = cp.asnumpy(M)
  C.wait_event(e) # Device waits.
  cp.sin(…)

e.synchronize() # Host waits.
```



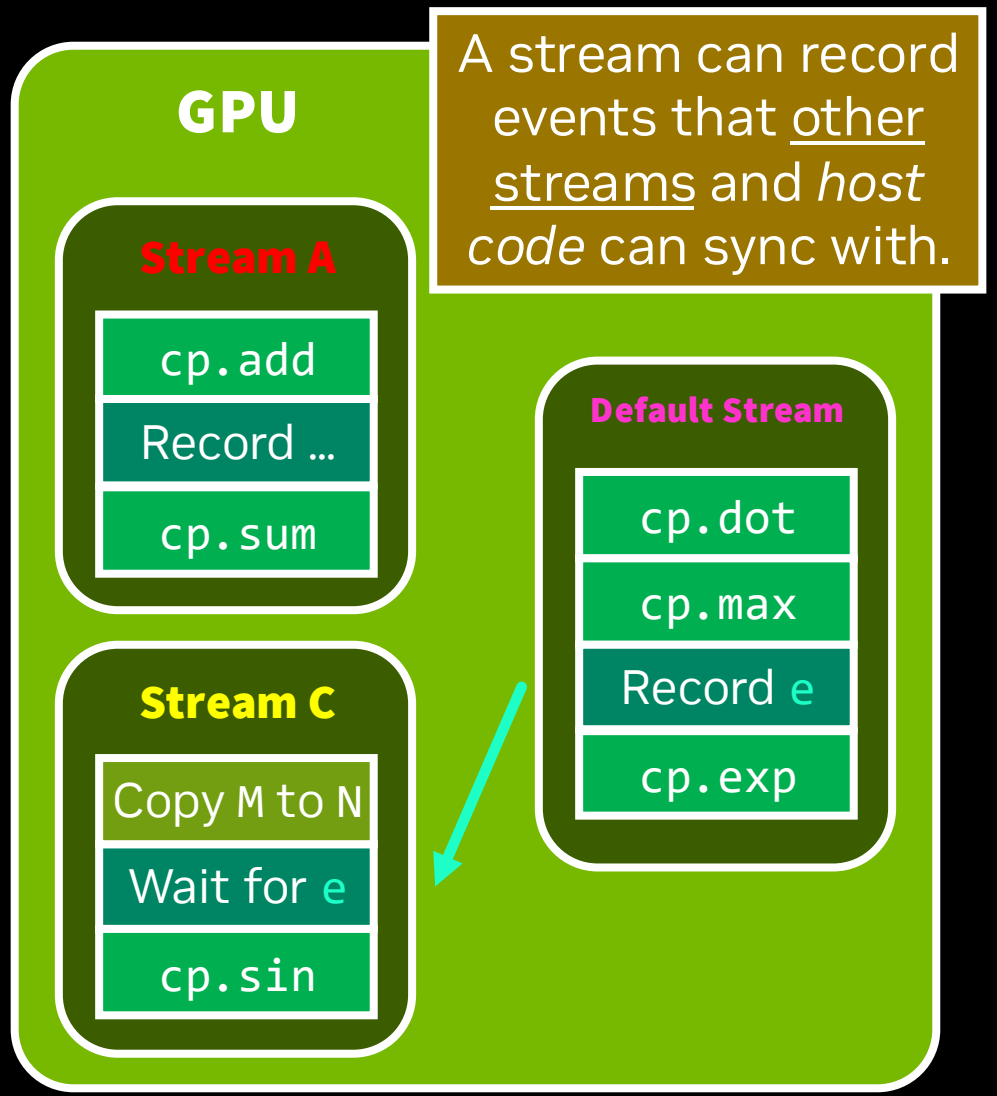
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)

cp.dot(…)
cp.max(…)
e = cp.cuda.get_current_stream().record()
cp.exp(…)

C = cp.cuda.Stream(non_blocking=True)
with C:
  N = cp.asnumpy(M)
  C.wait_event(e) # Device waits.
  cp.sin(…)

e.synchronize() # Host waits.
```



```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)

cp.dot(…)
cp.max(…)
e = cp.cuda.get_current_stream().record()
cp.exp(…)

C = cp.cuda.Stream(non_blocking=True)
with C:
  N = cp.asnumpy(M)
  C.wait_event(e) # Device waits.
  cp.sin(…)

e.synchronize() # Host waits.
```



```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)

cp.dot(…)
cp.max(…)
e = cp.cuda.get_current_stream().record()
cp.exp(…)

C = cp.cuda.Stream(non_blocking=True)
with C:
  N = cp.asnumpy(M, blocking=False)
  C.wait_event(e) # Device waits.
  cp.sin(…)

e.synchronize() # Host waits.
```

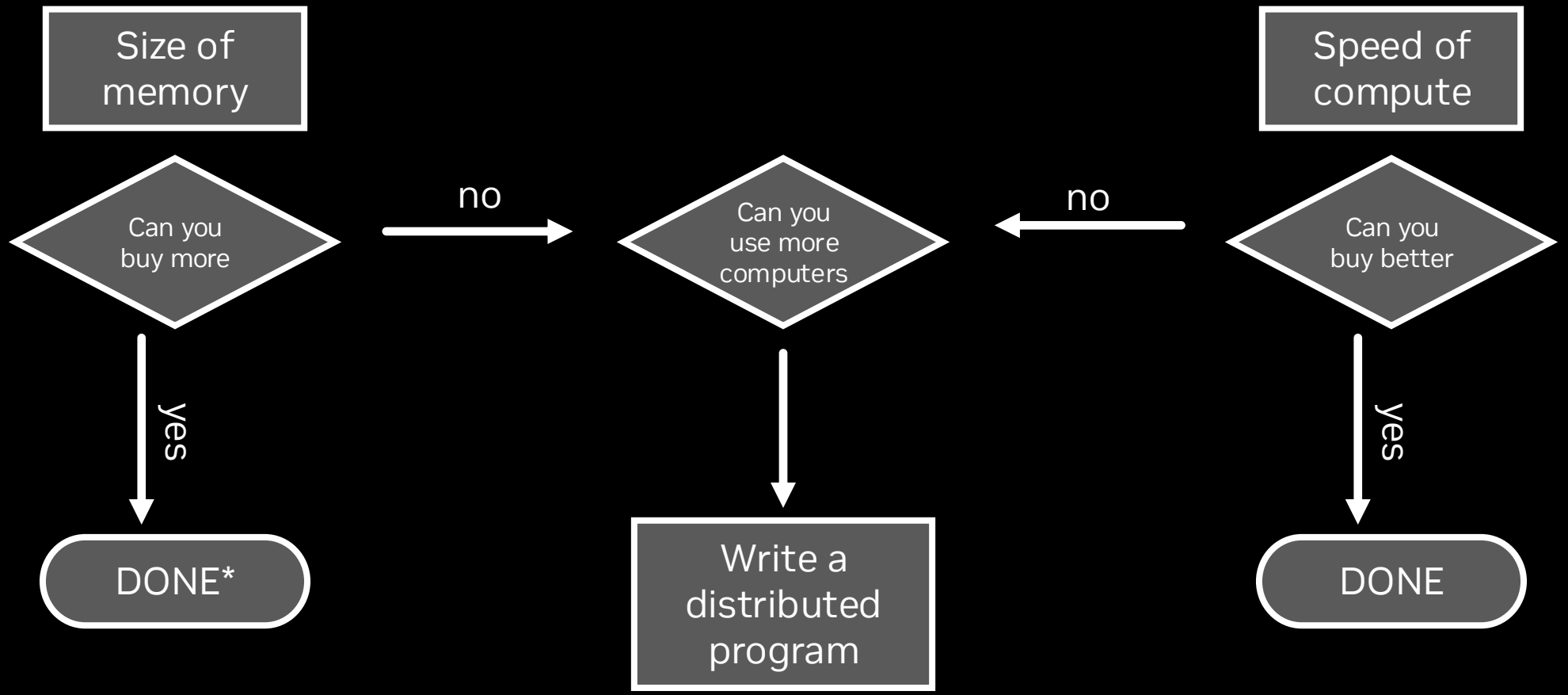


Exercise 06

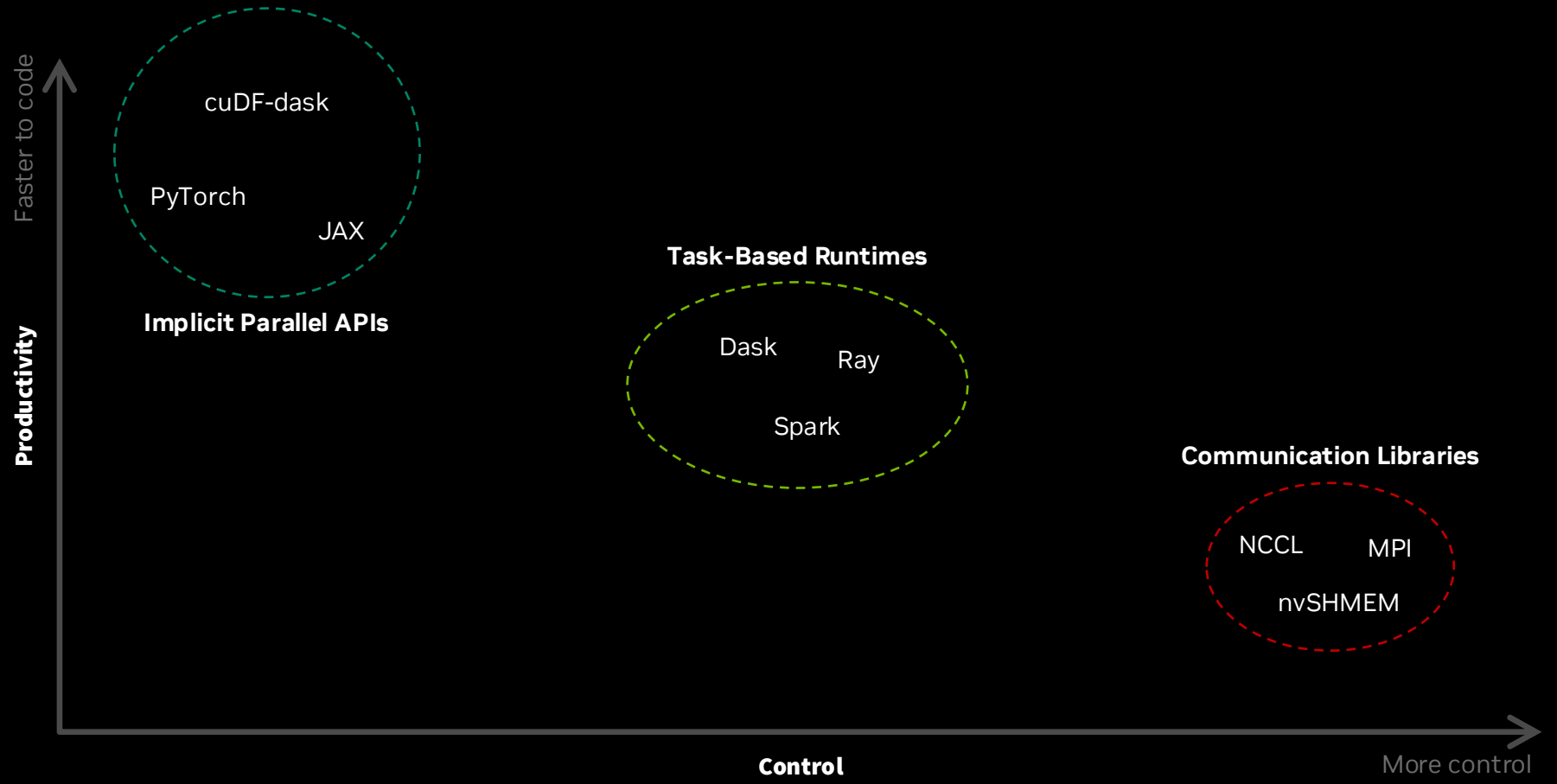
Asynchrony

Power Iteration

What are some limits to scale?



Distributed Programming Landscape



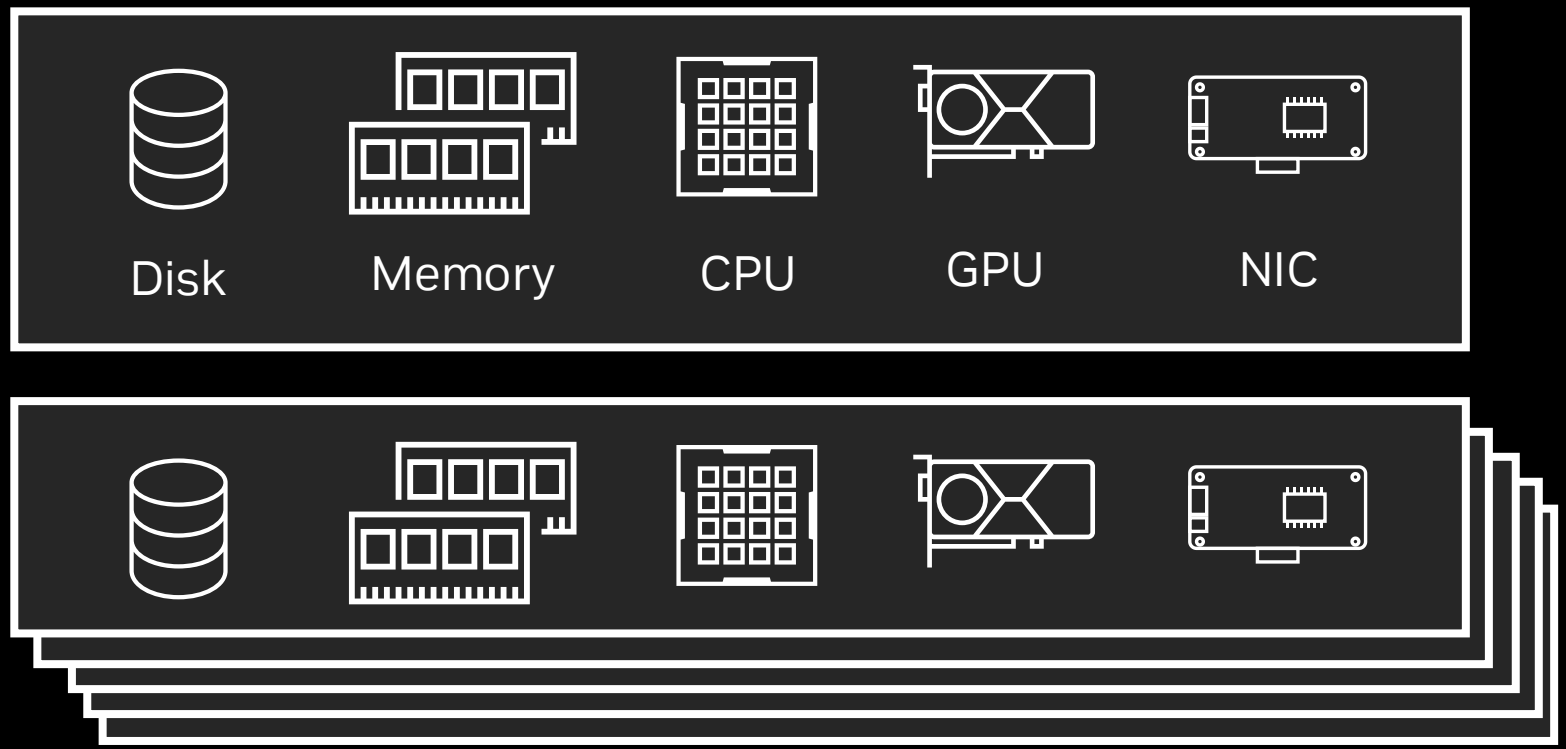
MPI – The HPC mainstay

- **communicators** encapsulate communication spaces for library safety
- **datatypes** reduce copying costs and permit heterogeneity
- multiple **communication modes** allow more control of memory buffer management
- extensive **collective operations** for scalable global communication
- **process topologies** permit efficient process placement, user views of process layout
- **profiling interface** encourages portable tools

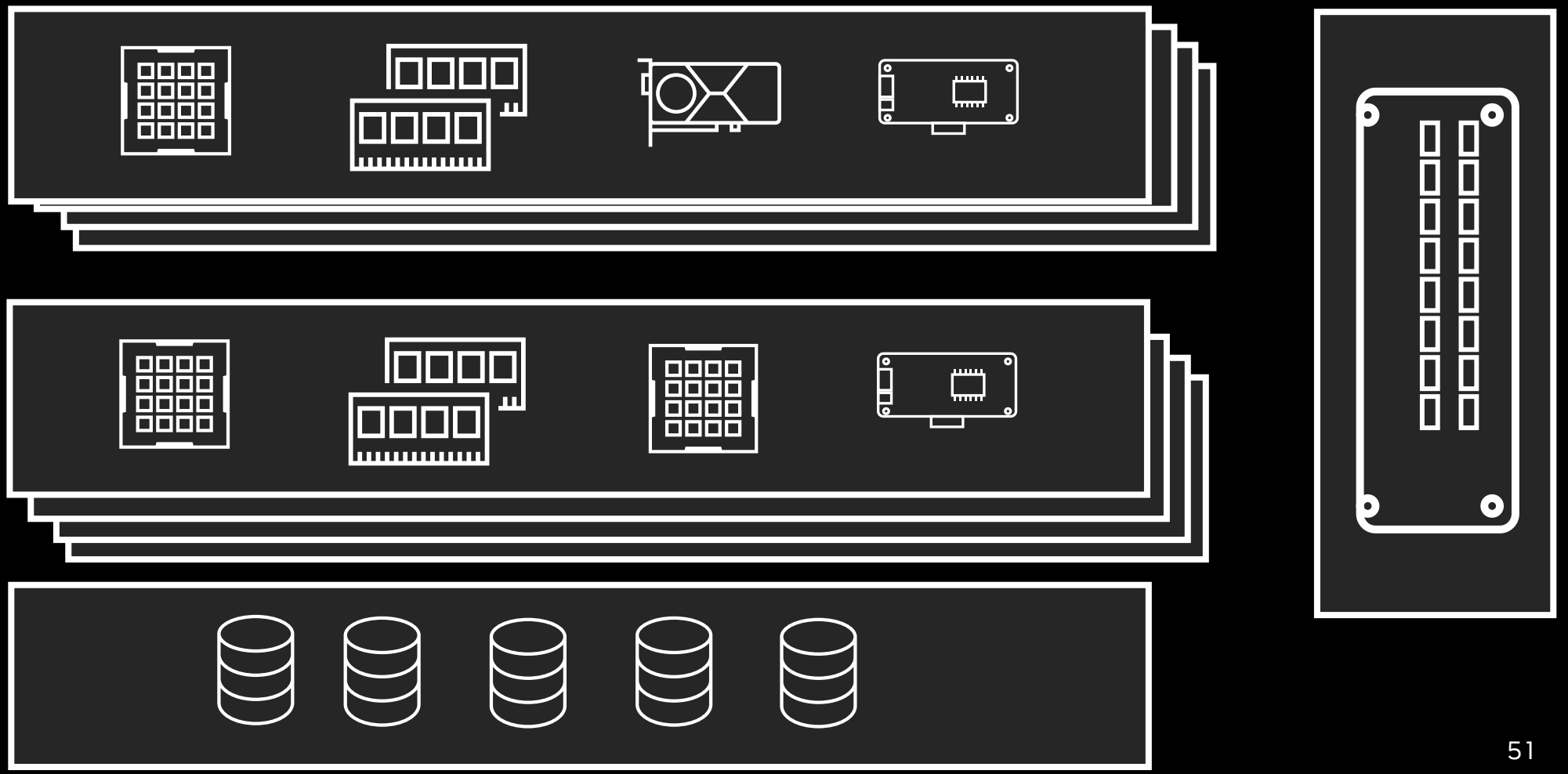
It Scales!

What is a Distributed System?

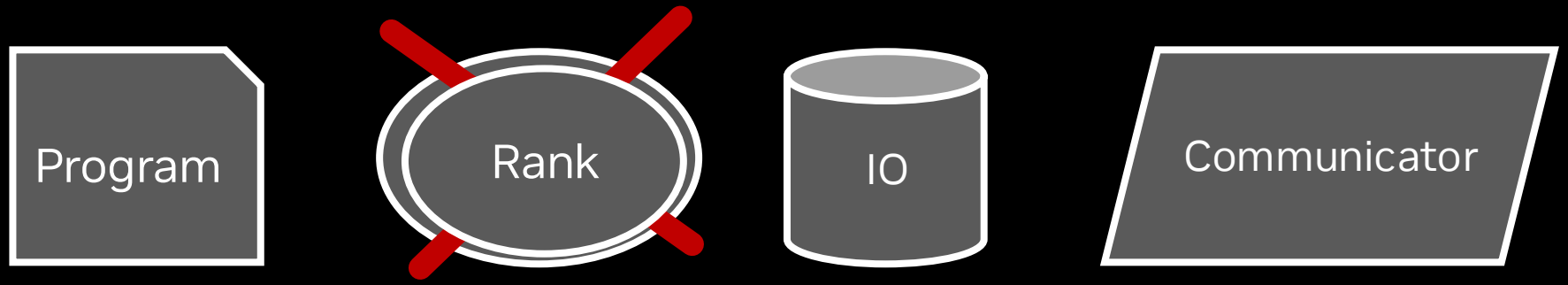
Node



Distributed Systems are often different



MPI Simplifies HPC



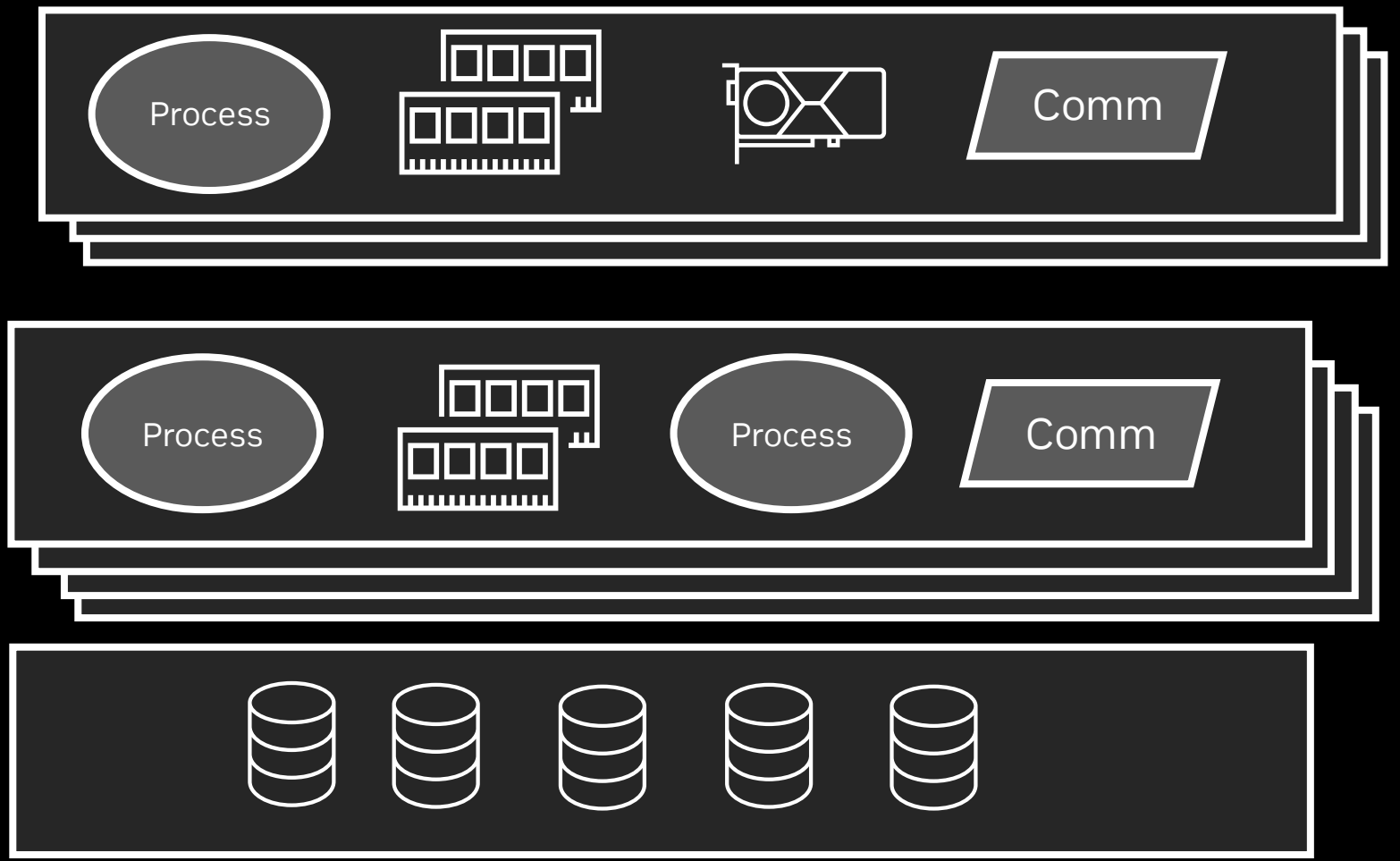
Most typical to have 1 program control the entire system

Typically, one per
- processor,
- socket, or
- gpu

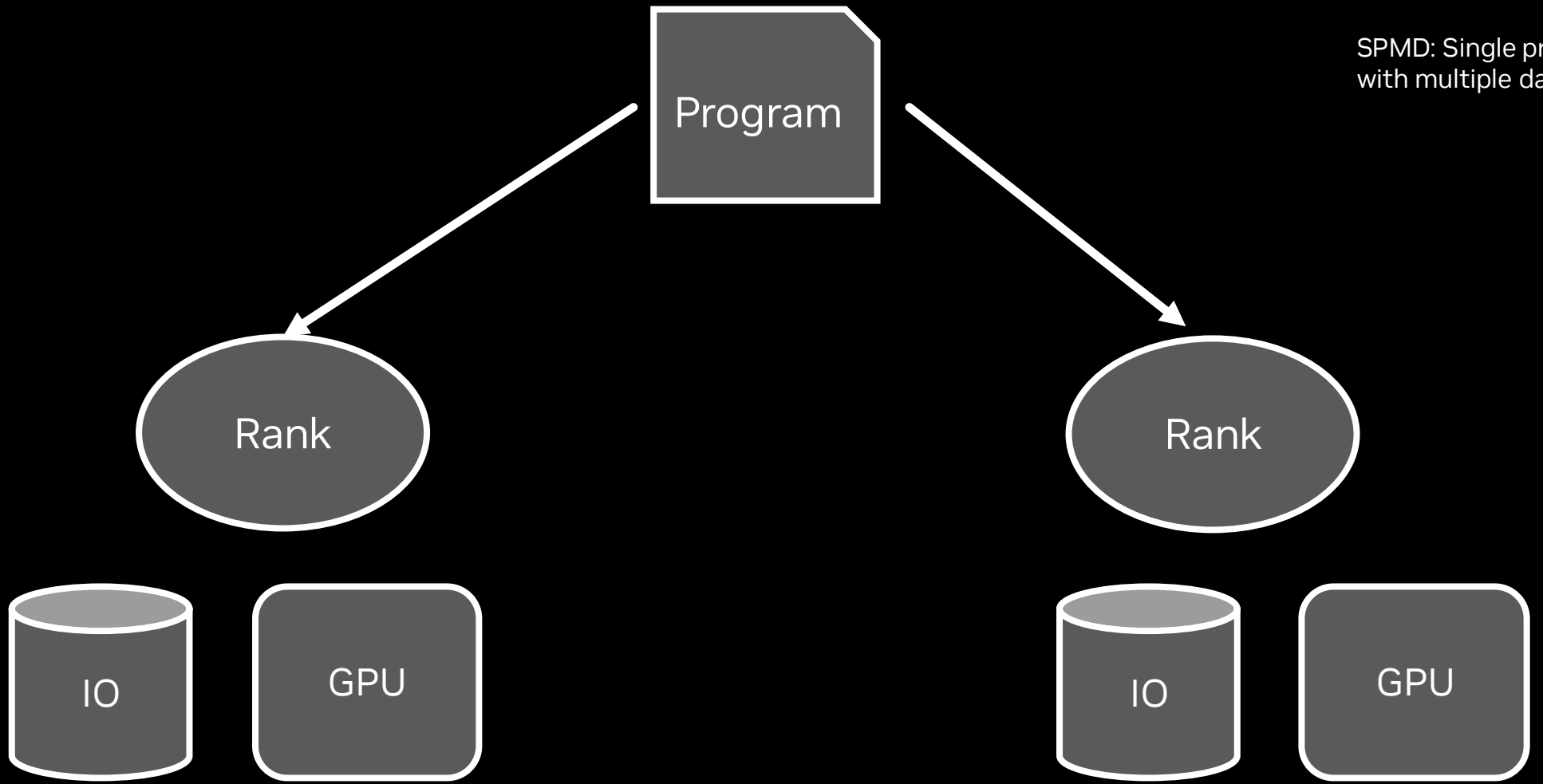
Beyond our scope but could be managed by filesystem or a parallel IO job via MPI

Hardware independent ways to organize communications.
Could be:
- Local memory
- Ethernet network
- Infiniband network
- etc

MPI Simplifies HPC



Message Passing Paradigm



SPMD: Single program with multiple data

Hello World

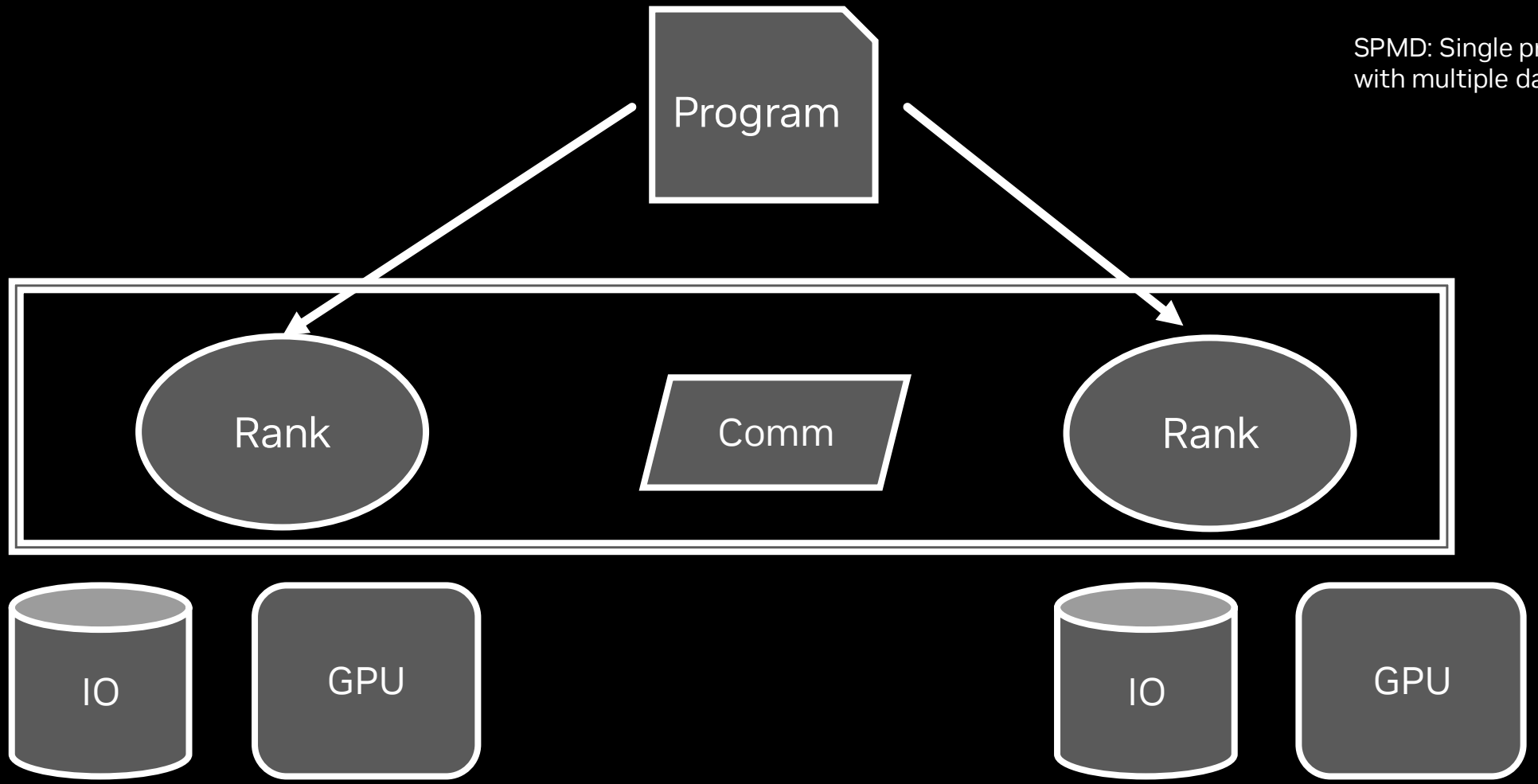
```
from mpi4py import MPI

print("Hello World!")
```

From a console in jupyter

> source /global/common/software/trn018/init-training-pyhpc-2025.sh

> srun -n 8 -c 8 python hello_world.py

Message Passing Paradigm



SPMD: Single program with multiple data

Hello World With Rank Data

```
import os

from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

# set up basic variables
name = MPI.Get_processor_name()
pid = os.getpid()

print(f"Hello World! I am process {rank} of {size} on {name} "
       "with pid {pid}.\n")
```

Communicators

- processes can be collected into **groups**
- each message is sent in a **context**, and must be received in the same context
- a **communicator** encapsulates a context for a specific group
- a given program may have many communicators with any level of overlap
- two initial communicators
  - MPI_COMM_WORLD (all processes)
  - MPI_COMM_SELF (current process)

mpi4py initialization

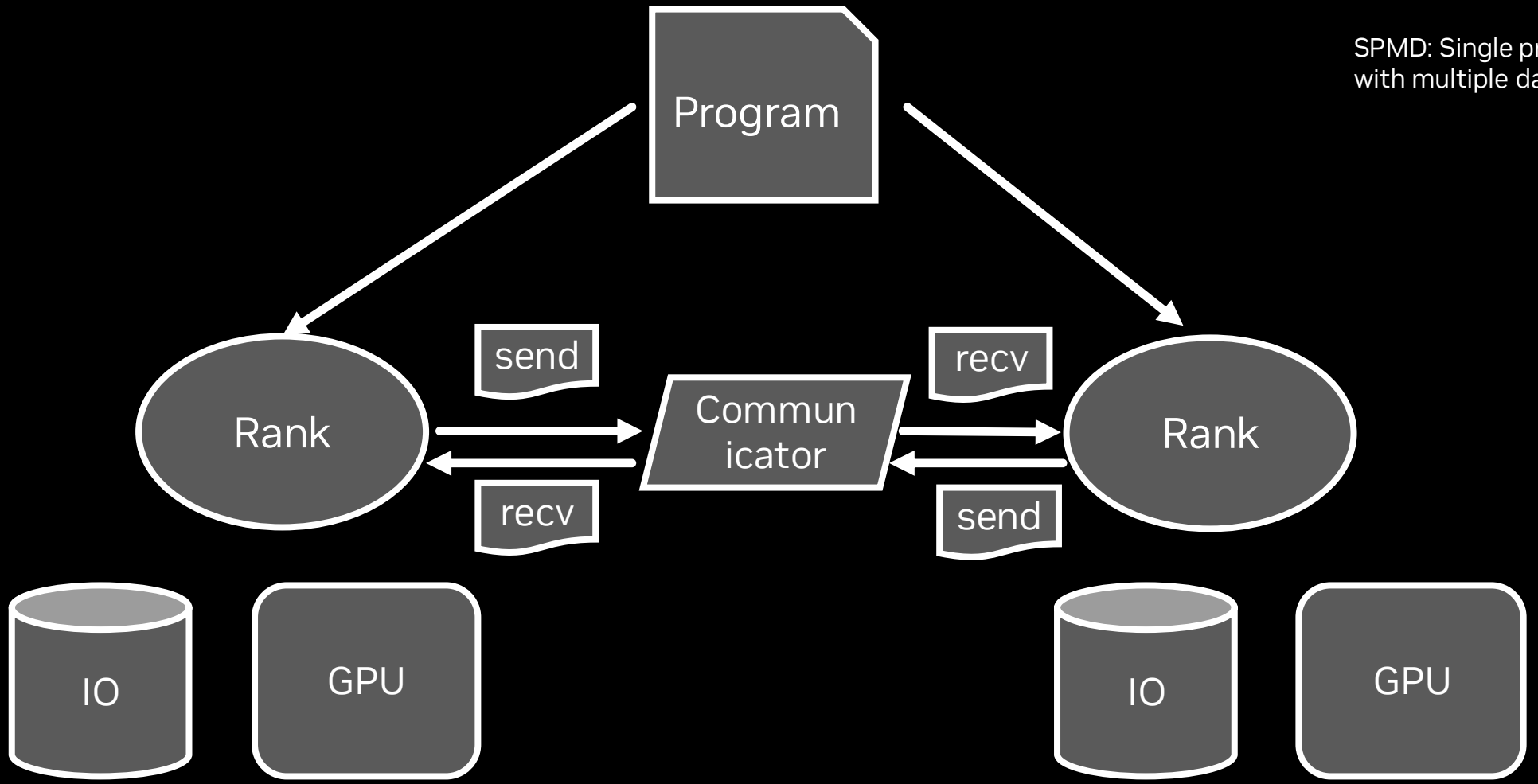
- mpi4py automatically calls MPI_Init() and MPI_Finalize()
- MPI_Init() is called when you import the MPI module from mpi4py
- MPI_Finalize() is called before the Python process ends
- If you need explicit control, use the mpi4py.rc module to configure before importing MPI

Exercise 1

Use the rank to print “fizz” if it divisible by 2, ”buzz” if divisible by 3, and “fizzbuzz” if both.

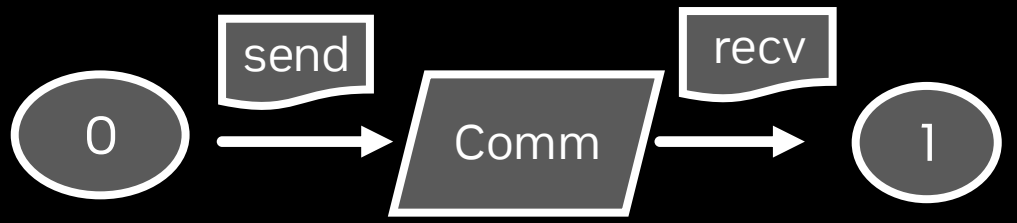
Message Passing Paradigm

SPMD: Single program with multiple data



mpi4py Blocking Send / Recv

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print(data)
```



mpi4py

Two interfaces for mpi commands:

- Generic API with lowercase function names, e.g. Comm.send
- Efficient API with titlecase function names, e.g. Comm.Send
  - The efficient API can still handle default arguments and type discovery for NumPy arrays and PEP-3118 buffers

Initialization

- Generic API with lowercase function names, e.g. Comm.send
- Efficient API with titlecase function names, e.g. Comm.Send
  - The efficient API can still handle default arguments and type discovery for NumPy arrays and PEP-3118 buffers

mpi4py Efficient API

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = np.arange(10, dtype='i’)
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = np.empty(10, dtype='i’)
    comm.Recv([data, MPI.INT], source=0, tag=77)
    print(data)
```

Message Data Types

- the data in a message to send or receive is described by address, count and datatype

- a datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous, strided block, or indexed array of blocks of MPI datatypes
  - an arbitrary structure of datatypes

- there are MPI functions to construct custom datatypes

Message Tags

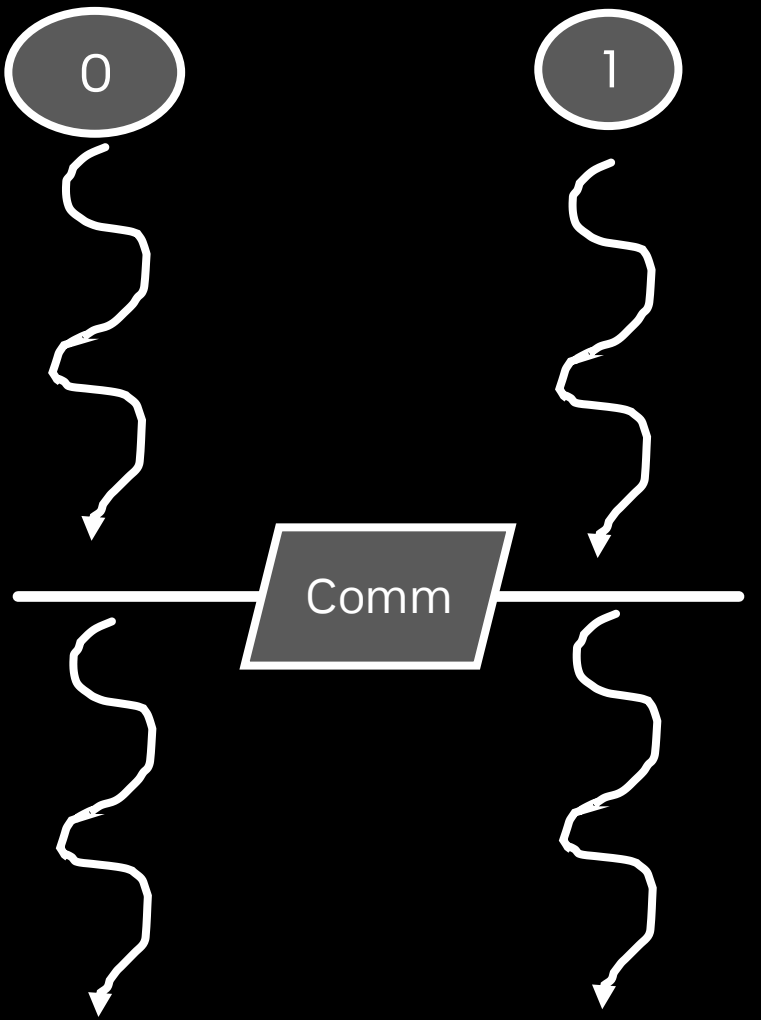
- messages are sent with an accompanying user-defined integer tag to assist the receiving process in identifying the message
- messages can be screened at the receiving end by specifying the expected tag, or not screened by using MPI_ANY_TAG

mpi4py numpy API

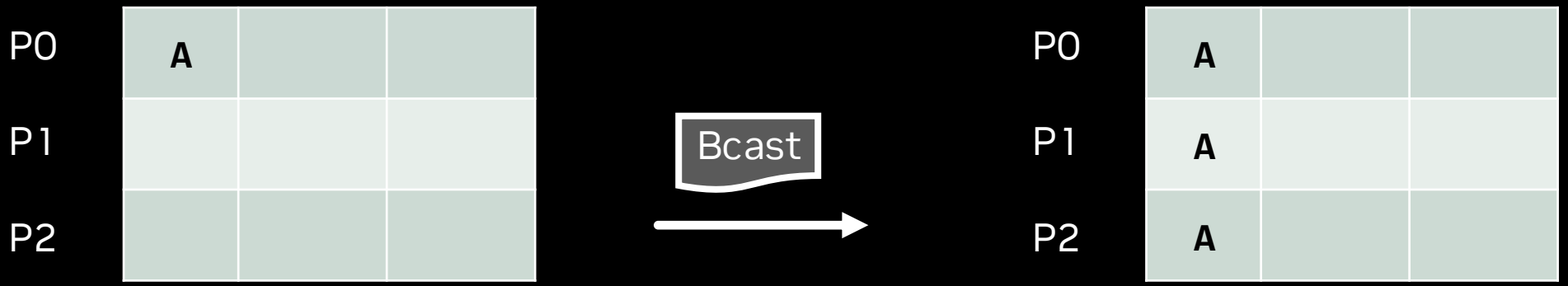
```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    data = np.arange(10, dtype=np.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = np.empty(10, dtype=np.float64)
    comm.Recv(data, source=0, tag=13)
    print(data)
```

Barriers

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

for r_id in range(comm.Get_size()):
    if rank == r_id:
        print(f"Hello from proc:")

comm.Barrier()
print(f"proc {rank} after barrier")
```



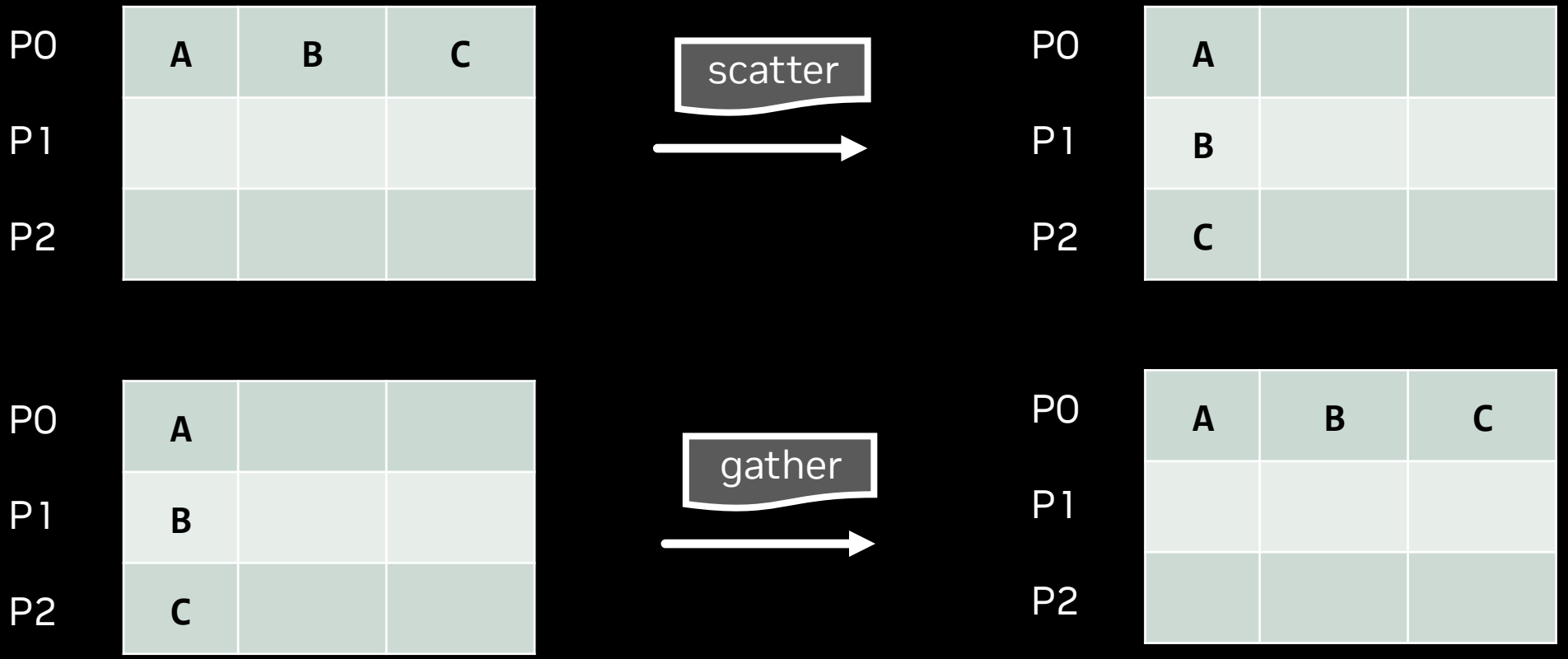
Broadcast



Bcast

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],
            'key2' : ( 'abc', 'xyz')}
else:
    data = None

data = comm.bcast(data, root=0)

print(f"bcast finished and data "
      "on rank %d is: {comm.rank}, {data}")
```

Scatter / Gather



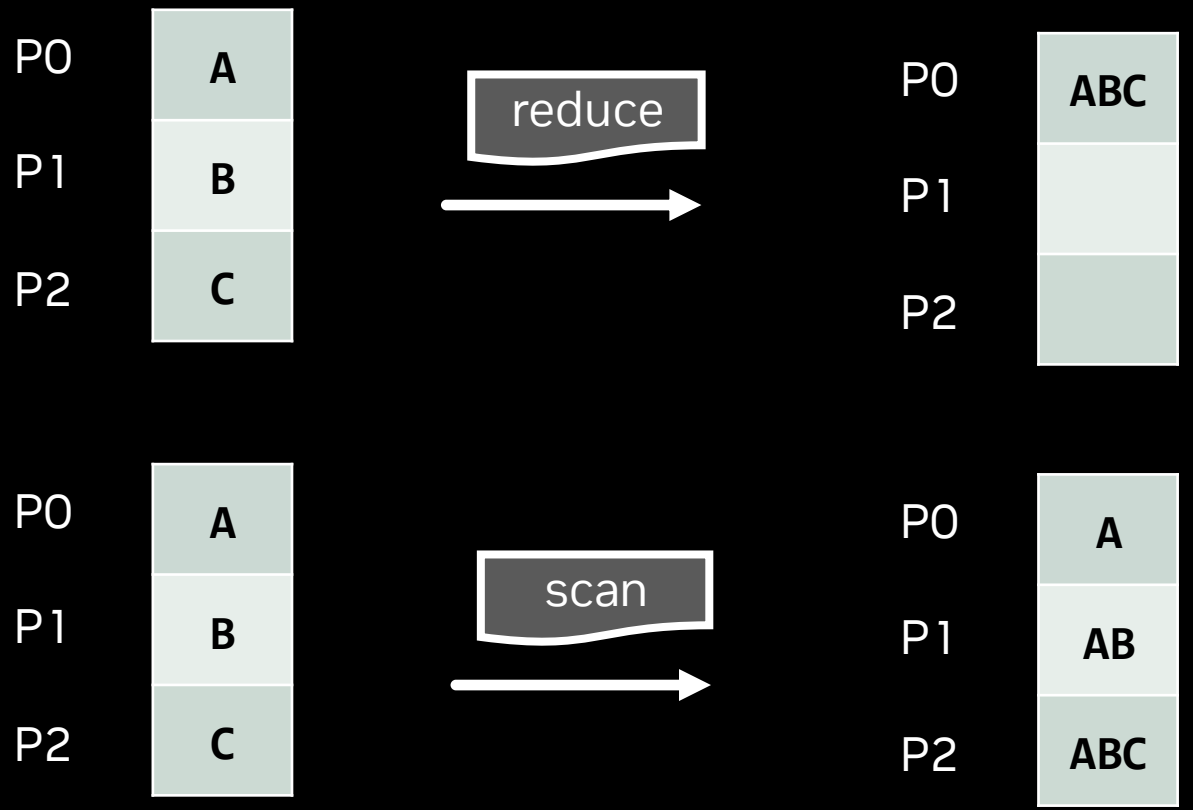
scatter

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    data = [(i+1)**2 for i in range(size)]
else:
    data = None

data = comm.scatter(data, root=0)

assert data == (rank+1)**2
print(f"data on rank %d is: {rank}, {data}")
```

gather

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

data = (rank+1)**2
print(f"before gather, data on rank %d is: {rank}, {data}")

comm.Barrier()
data = comm.gather(data, root=0)

if rank == 0:
    for i in range(size):
        assert data[i] == (i+1)**2
else:
    assert data is None

print(f"data on rank: {rank} is: {data}")
```

Reduce / Scan



reduce

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

sendmsg = comm.rank
recvmsg1 = comm.reduce(sendmsg, op=MPI.SUM, root=0)
recvmsg2 = comm.allreduce(sendmsg)
print recvmsg2
```

GPU Aware MPI

Several MPI implementations, including Open MPI and MVAPICH, support passing GPU pointers to MPI calls to avoid explicit data movement between host and device

This feature requires:

- mpi4py is built against a GPU-aware MPI library.
- The Python GPU arrays are compliant with either of the protocols.



CuPy







CuPy Arrays

```
from mpi4py import MPI
import cupy as cp


comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()


sendbuf = cp.arange(10, dtype='i')
recvbuf = cp.empty_like(sendbuf)
cp.cuda.get_current_stream().synchronize()
comm.Allreduce(sendbuf, recvbuf)


assert cp.allclose(recvbuf, sendbuf*size)
```

Computing multiple computers

GO BIGGER!!!!

Weak Scaling

Data is growing and I want more to add more computers as it grows

Bigger Data, Same time to compute

Limited by Denard Scaling

GO FASTER!!!!

Strong Scaling

Data is fixed and I want more computers to go faster

Same size data, Faster time to compute

Limited by Amdahl's Law

What We Learned Today

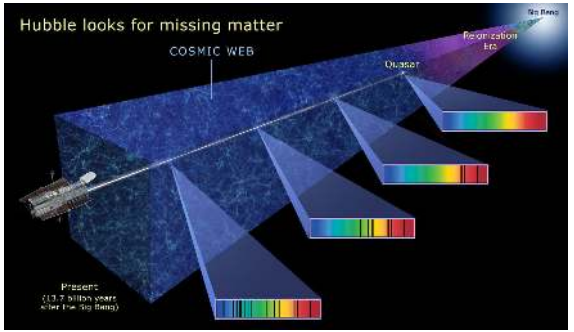
- NumPy: Array-oriented programming in Python.
- CuPy: CUDA-accelerated NumPy, linear algebra, & scientific algos.
- mpi4py: Scaling out with distributed programming.

SC26
hpc unites | Chicago

Accelerating and Scaling Python for HPC

mini-app: Topological Data Analysis of Cosmic Web data

Aaron Jomy, CERN
ROOT analysis framework, Compiler-Research Org



Libraries

Programming Models/Libraries in Python for HPC

ARRAY & TENSOR COMPUTATION
Vectorized Foundation Library








JIT & AOT COMPILATION
Native Machine Code Generation











INTEROPERABILITY BINDINGS
C / C++ / Rust Integration









DISTRIBUTED & TASK-BASED PARALLELISM
Multi-Node Scaling







ACCELERATED DATAFRAMES
Out-of-Core Data Processing





JAX

NumPy-style arrays with composable function transformations, compiled by XLA for GPUs, TPUs and CPUs.

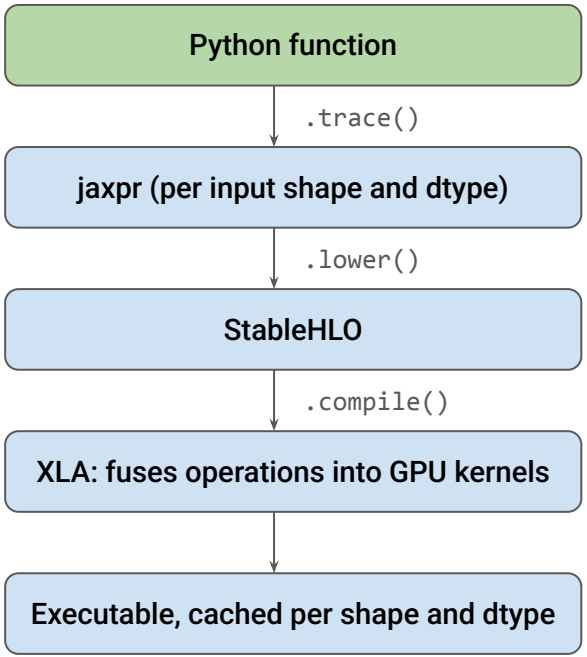
- `jit`: compiles the whole function and fuses operations, so fused intermediates never round-trip through global memory (HBM)
- `vmap`, `grad`: automatic batching and autodiff
- sharding: one program across many devices

`jit` expects functional code:

- immutable arrays, updates like `x.at[i].set(v)`
- loops via `lax.scan` or `lax.while_loop`
- static shapes: change in input shape/dtype triggers a recompile

Dispatch is asynchronous: call `.block_until_ready()` before timing.

jax 0.12 on PyPI: `pip install "jax[cuda13]"`



`jax.jit(f).trace(x).lower().compile()` walks these stages.
`.as_text()` prints the lowered and compiled programs.

cuda-cccl

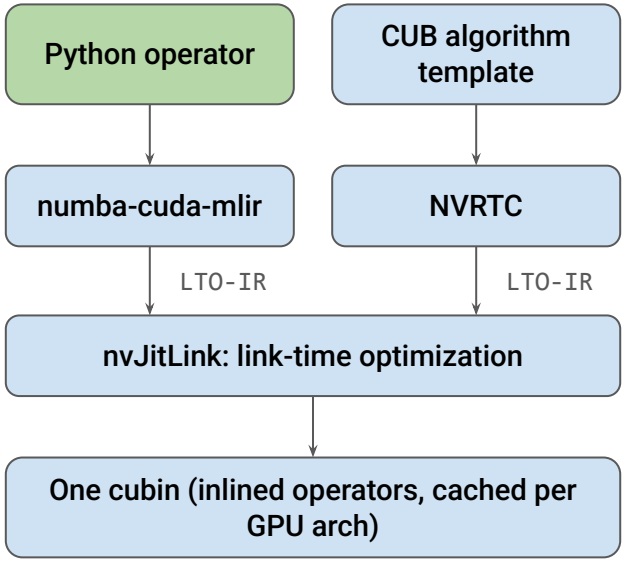
The CUDA Core Compute Libraries (Thrust, CUB) are the GPU building blocks used by PyTorch and CuPy. cuda-cccl brings them to Python.

- `cuda.compute`: device-wide reduce, scan, sort, histogram and transform on CuPy or PyTorch arrays

Inside `cuda.compute`:

- Python operators and `gpu_struct` types are JIT-compiled and inlined into the CUB kernel
- iterators (`CountingIterator`, `TransformIterator`, `ZipIterator`) fuse extra work into the same kernel
- `make_*` objects reuse the build and let you manage temporary storage

cuda-cccl 1.2 on PyPI: `pip install "cuda-cccl[cu13]"`



Python operators are compiled separately and linked into the CUB kernel when the algorithm is built.

numba-cuda

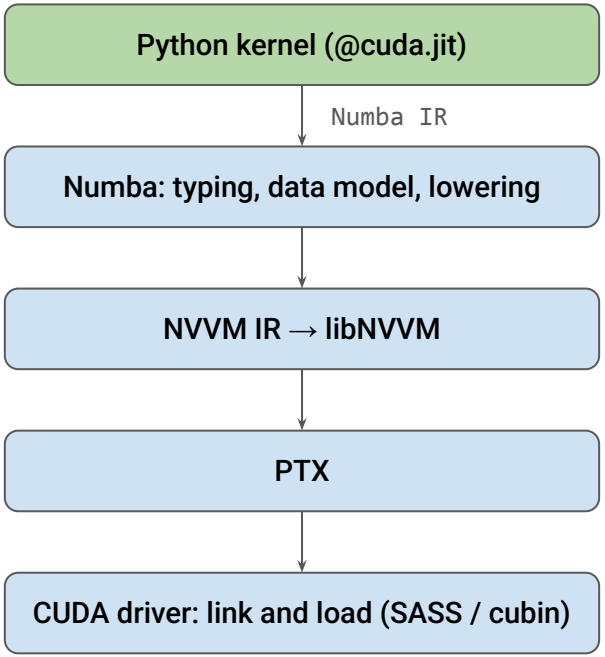
When no library algorithm fits, write the kernel yourself:
numba-cuda compiles a subset of Python into CUDA kernels.

The CUDA SIMT programming model in Python:

- `kernel[blocks, threads](args)` launches a `@cuda.jit` kernel asynchronously
- `cuda.grid(1)` gives the global thread index; results go into output arrays
- `cuda.shared.array`, `cuda.syncthreads()` and `cuda.atomic.add` for shared memory, barriers and atomics
- takes CuPy and PyTorch arrays directly via `__cuda_array_interface__`

numba-cuda 0.30 is in maintenance mode: security and critical fixes, through the CUDA 13 lifetime.

Its successor numba-cuda-mlir retains `@cuda.jit`.



numba-cuda-mlir generates MLIR instead of LLVM IR at the lowering stage.

CppJIT

Automatic bindings and interoperability with existing high-performance C++ libraries or your own C++ framework.

Given headers, libraries, or C++ code, CppJIT provides Python objects.

No interface files, bindings code, generated wrappers or build step.

Bindings are created *lazily*, on attribute access: pay for what you use.

CPython-based runtime:

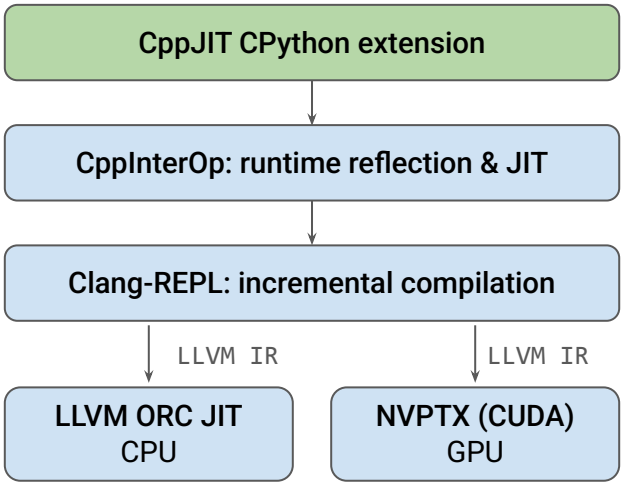
- proxies for scopes, methods, overloads, data members, enums
- converters and executors for the calling convention
- GIL management
- memory regulator keeps Python and C++ lifetimes in sync

Alpha release currently available for preview on PyPI.

Conda releases scheduled for EoY 2026.

```
import cppjit as cpp

cpp.load_library("myGameEngineLib.so")
cpp.include("myGameEngineLib.h")
cpp.gbl.Scene("World1").render(args)
```

class *constructor* *method*



CppJIT

Some use cases:

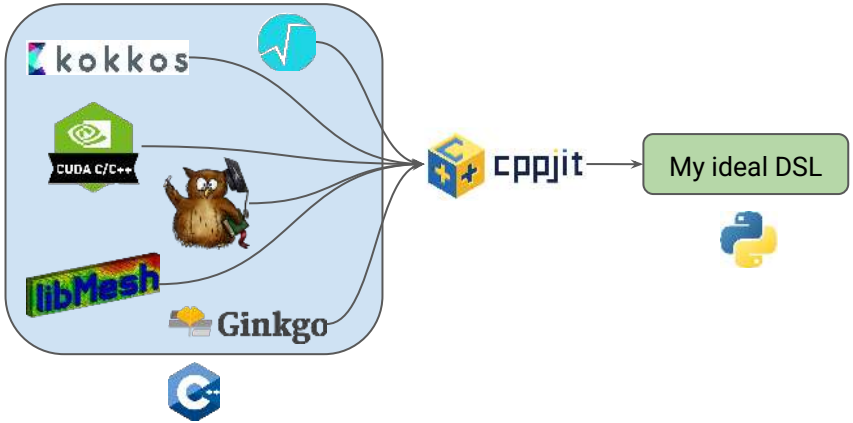
- **Numerics and data science:** Offload performance-critical code from Python to C++.
- **Template-heavy APIs:** STL, Boost, Eigen, and user templates instantiate lazily at call sites.
- **Building Domain-Specific Languages:** User-defined *"pythonization"* layer adapts bindings into Pythonic libraries.

CppJIT powers the ROOT analysis framework used at CERN: high-performance C++ libraries for particle physics data processing, statistical analysis and visualization automatically available as a Python library.

→ Discover the Higgs boson in a Python notebook

GPU code:

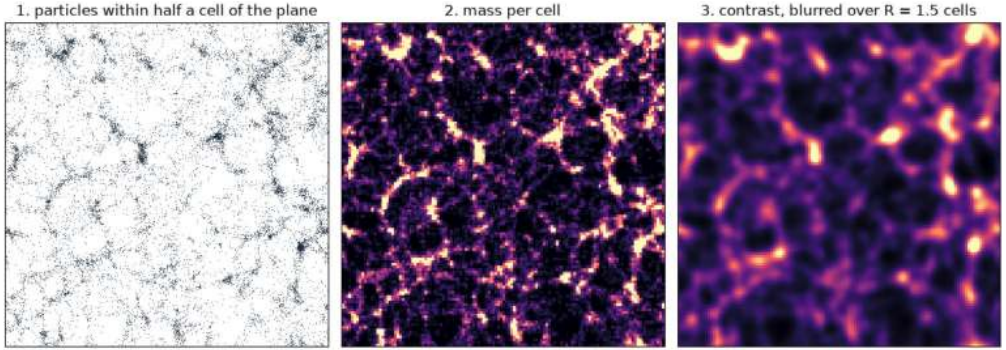
```
from cppjit.gbl import thrust

data = thrust.device_vector["double"](100000)
thrust.sequence(thrust.device, data.begin(), data.end())

total = thrust.reduce(thrust.device, data.begin(), data.end(), 0.0)

square = thrust.square["double"]()
sum_sq = thrust.transform_reduce(thrust.device, data.begin(),
                    data.end(), square, 0.0, thrust.plus["double"]())
```



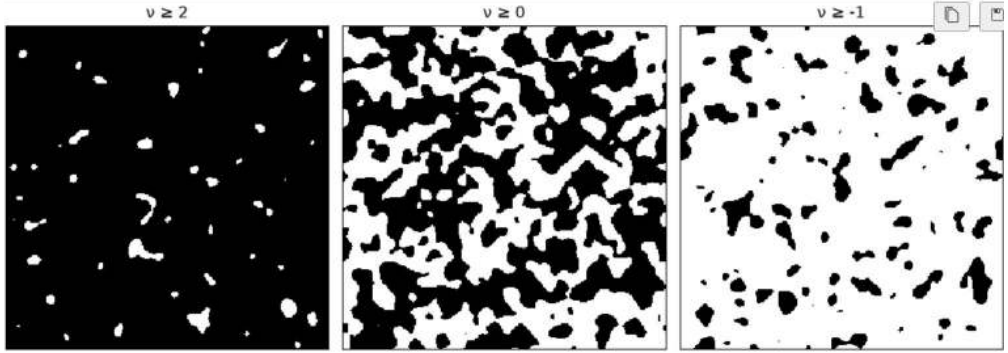
Problem Statement

Mini-app on simulated Cosmic Web data

How is matter in the universe connected? A slice through a simulated density field, thresholded at three levels:

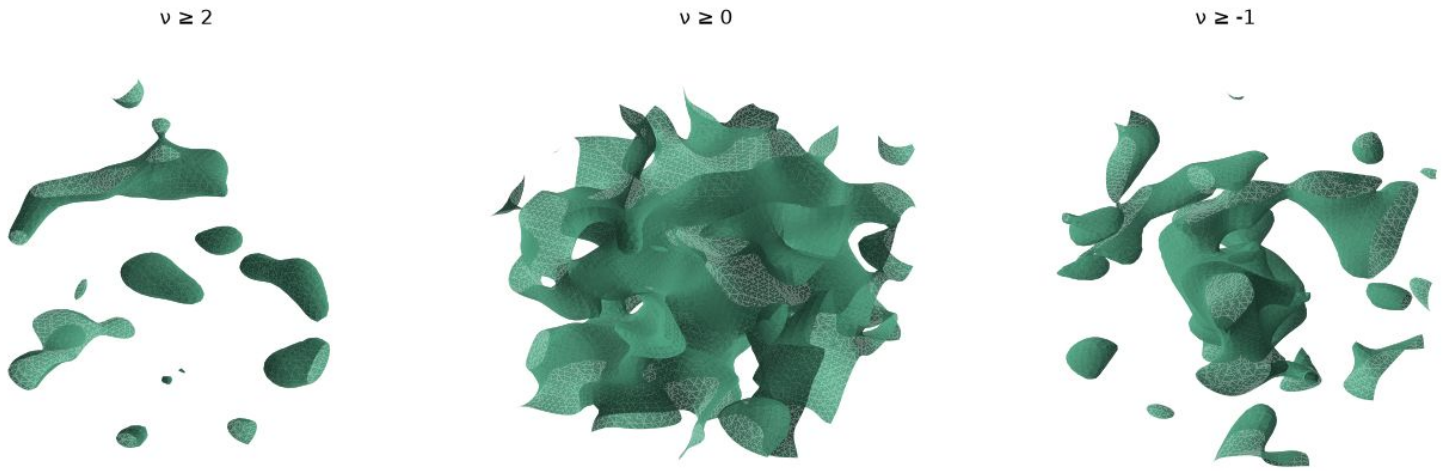


Each level shows a different structure: isolated **knots**, a connected **web**, isolated **voids**:



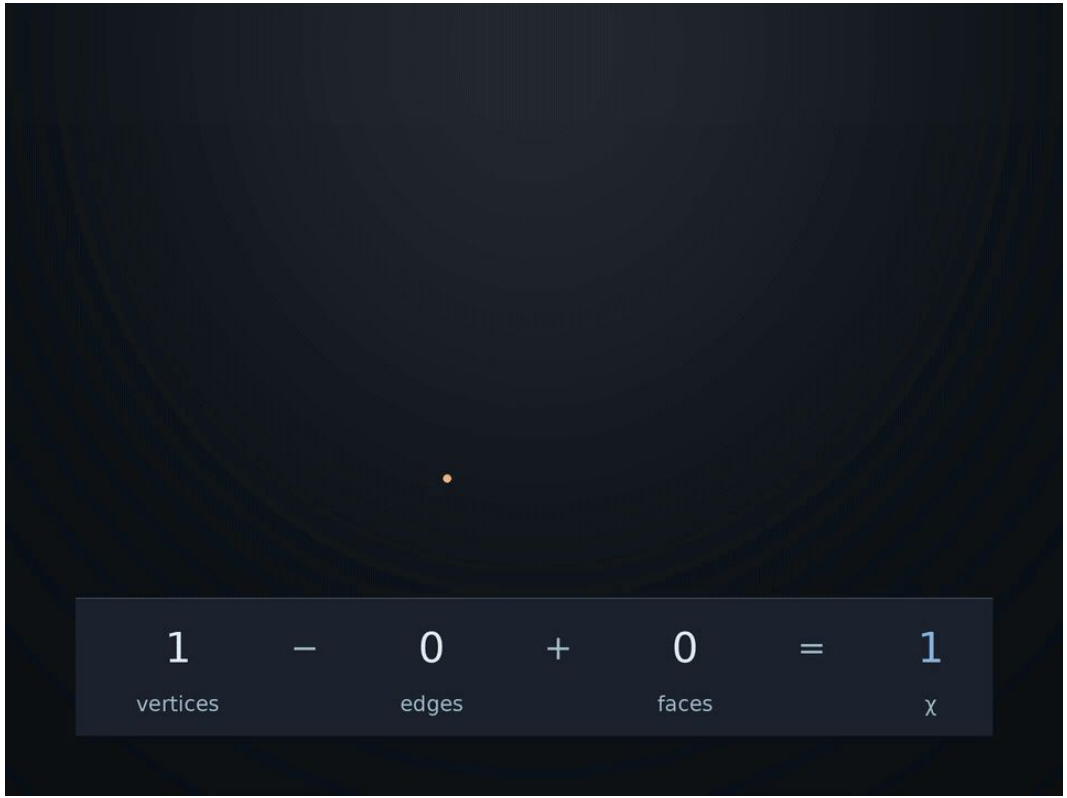
Mini-app on simulated Cosmic Web data

The same structures in 3D:



The **Euler Characteristic** summarizes how each shape is connected:
χ = components − tunnels + cavities

Euler Characteristic



Relevance beyond astrophysics

The Euler Characteristic is a simple invariant with applications across science and engineering:

Topology & Geometry: Classifies shapes by connectivity and holes.

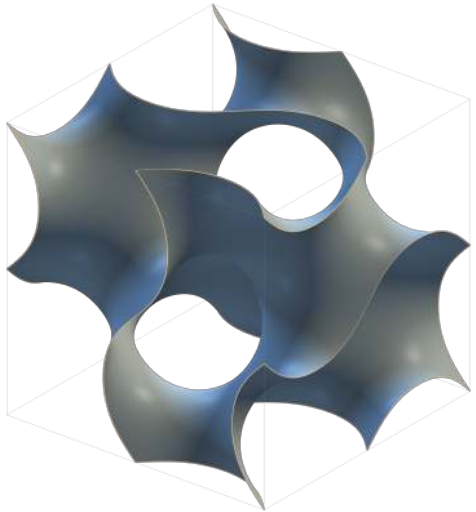
Computer Graphics: Detects defects in 3D meshes.

Materials Science: Characterizes pores and internal structures.

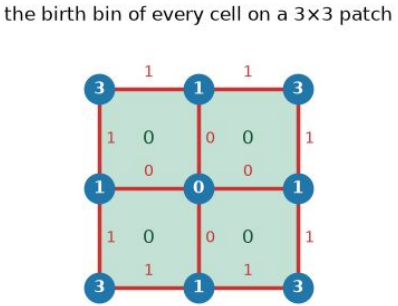
Biology & Medicine: Analyzes tissue and vascular morphology.

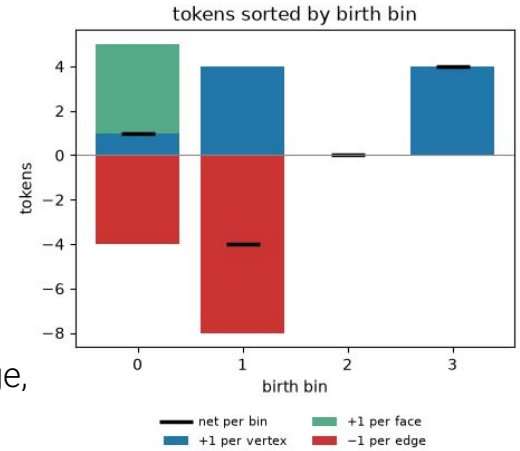
Machine Learning: Reveals structure in high-dimensional data.

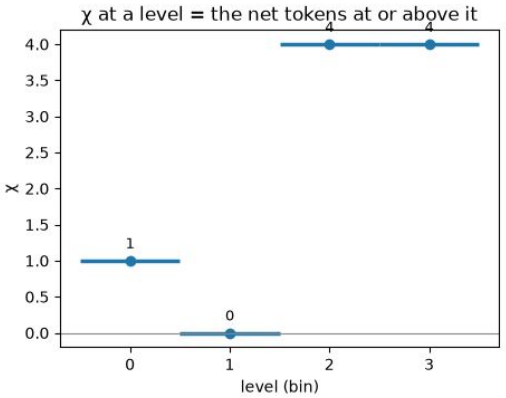
Physics: Studies phase transitions and complex structures.



What we’re going to solve





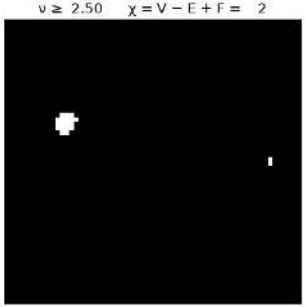


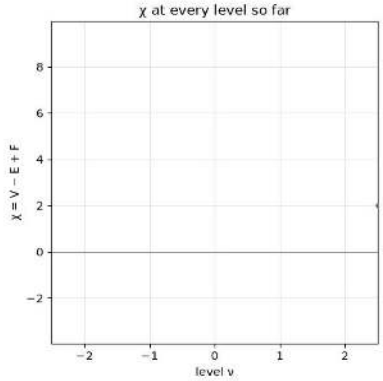
Local count:

- Each value is the level at which that point, edge, square (or cube, in 3D) joins the set.
- Counting depends only on each cell's immediate neighbors: a stencil that fills a histogram.

One pass:

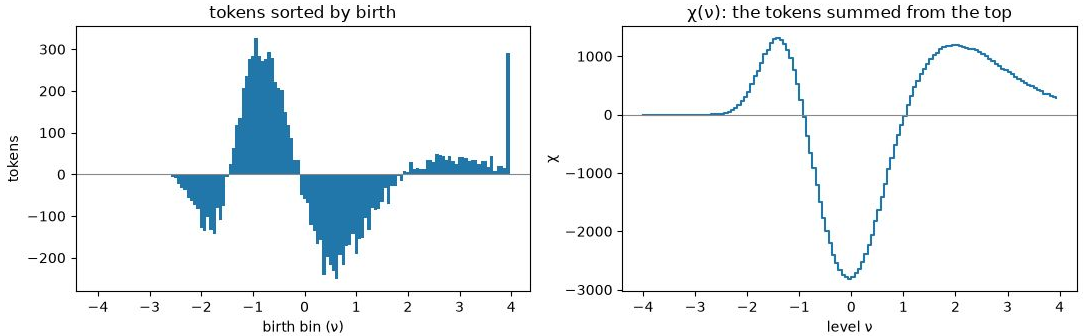
- A sweep over the grid fills a histogram whose running sum gives us **χ** at every level.
- We’ll compute this sweep on the GPU and aim to speed it up.





Goal of this tutorial

The resulting **Euler Characteristic Curve** looks like:



This mini-app implements a solver for this problem using different libraries and tools (Notebook 6 - 11)

- CuPy
- JAX
- cuda.compute
- numba-cuda
- CppJIT

Along the way, we'll identify **potential bottlenecks** in our code, use **profiling tools** and attempt to **improve performance.**

We conclude with a synthesis notebook that **compares the costs and throughput** achieved across the board.



Accelerating and Scaling Python for HPC Science Integration

Dr Vassil Vassilev, CERN

compiler-research.org

YOUR PYTHON — CPython interprets and orchestrates

ecosystem: NumPy · pandas · ML stack · your analysis code

AOT · vertical

Packaged libraries

NumPy · CuPy · mpi4py · Dragon

```
cp.add(x, y)
```

glue: shipped
kernel: shipped
when: before install

AOT · vertical

Bindings you write

ctypes · pybind11 · Cython

```
mylib.fast_add(x, y)
```

glue: you write it
kernel: existing C/C++
when: your build step

JIT · vertical

Compiled Python

Numba — @njit · @cuda.jit · JAX - jax.jit

```
add(x, y) # @njit
```

glue: generated
kernel: Python → LLVM
when: JIT, first call

JIT · horizontal

4 · Runtime interop

cppjit (CppInterOp)

```
cppjit.gbl.add(x, y)
```

glue: built from C++ AST
kernel: C++ cppdef'd live
when: JIT, first use

no shaft of its own — objects, templates, inheritance, callbacks cross both ways

C++ ECOSYSTEM

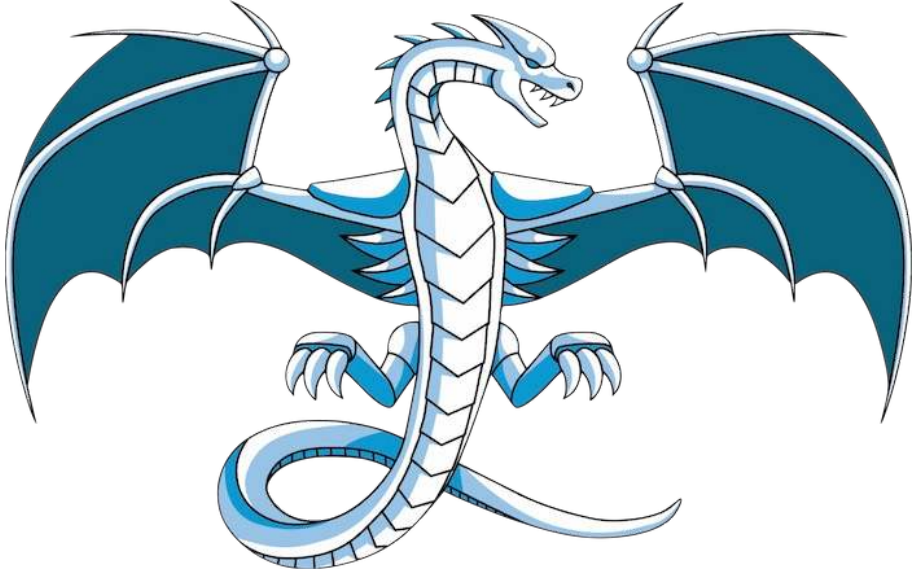
Eigen · CUDA
30 years of HPC code
your lab's libraries

compiles to native like everything else

COMPILED NATIVE CODE — kernels run with the GIL released

CPU SIMD · GPU (CUDA) · interconnect (MPI) · cluster processes

The compiler is in the club



```
import cppjit
cppjit.include("kernel/logistic_step.h")   # the header, as written

N = 50.0
for _ in range(60):
    N = cppjit.gbl.logistic_step["double"](N, 0.30, 2000.0)
print(N)                                   # 1999.999842435456
```

A single session where compilers, languages and tools compose: call, transform, compose.

Call it: not just fast but zero-copy

```
view = np.asarray(vec.data())

view.flags["OWNDATA"]                              # False
hex(view.ctypes.data) == hex(cppjit.addressof(vec.data()))   # True

view[0] = -1.0;  assert vec[0]  == -1.0      # write visible C++-side
vec[0]  = 50.0;  assert view[0] ==  50.0     # write visible NumPy-side
```

Transform it: Until model has no .backward()

```
def step_jax(N, r, K): return N + r * N * (1 - N / K)# the law, in JAX
jax.grad(step_jax, argnums=1)(50.0, 0.3, 2000.0)      # 48.75


cppjit.include("kernel/logistic_step.h")              # the law, as written
jax.grad(lambda r: cppjit.gbl.logistic_step["double"](50.0, r,
2000.0))(0.3)
# TypeError: Could not find "logistic_step<double>"
```

Deep learning gave Python the derivative of anything you write *in the framework*. Many big sciences frameworks (ROOT, Geant4, BioDynaMo) are not written in the framework, and the tracer cannot enter a compiled function.

Transform it: **C++ source**, dumped verbatim

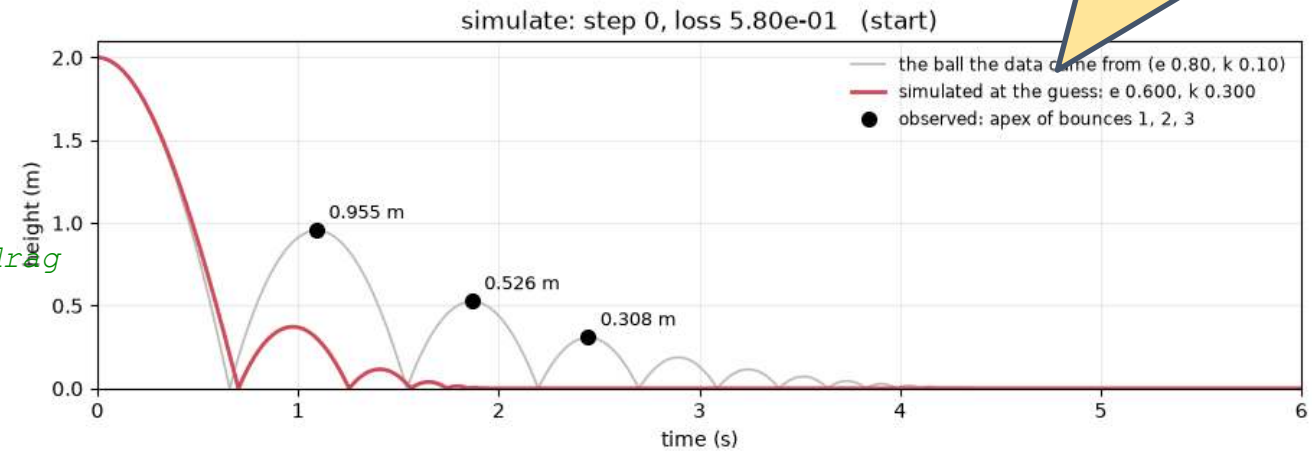
```
double step_darg1(double N, double r, double K) {
          double _d_N = 0; double _d_r = 1; double _d_K = 0;
          clad::ValueAndPushforward<double, double> _t0 =
              logistic_step_pushforward(N, r, K, _d_N, _d_r, _d_K);
          return _t0.pushforward;
      }
```

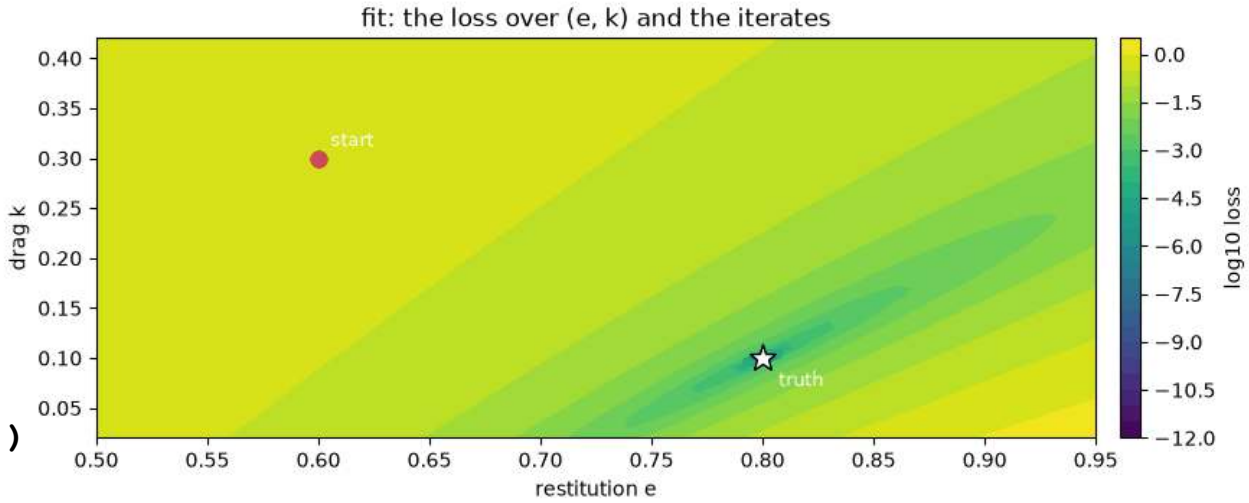
```
# Clad rides inside the session's compiler
os.environ["CPPINTEROP_EXTRA_INTERPRETER_ARGS"] = "-fplugin=clad.so -O2"
import cppjit; cppjit.cppdef(cell) # the cell above, compiled in the session
```

Simulate and fit it

BFGS in JAX with Clad Gradient

```
double apex(double e, double k, int n) {
  // drop from 2 m; apex of bounce n
  double y = 2.0, v = 0.0, dt = 1e-3, top = 0.0;
  int bounces = 0;
  for (int i = 0; i < 20000; ++i) {
    v -= (9.81 + k * v * std::fabs(v)) * dt;// gravity, air drag
    y += v * dt;
    if (y < 0) { y = 0; v = -e * v; ++bounces; }
    if (bounces == n && v > 0 && y > top) top = y;
  }
  return top;
}
// p = {e, k}; obs = the three apex heights
double loss(double* p, const double* obs) {
  double L = 0; for (int n = 1; n <= 3; ++n) {
  double d = apex(p[0], p[1], n) - obs[n-1]; L += d*d; }
  return L; }
auto d_loss = clad::gradient(loss, "p");

# forward: loss(); backward: d_loss
cpp_loss = jax.custom_vjp(...)
# JAX runs the optimiser; it never sees the C++
fit = optx.minimise(cpp_loss, optx.BFGS(), [0.6, 0.3])
```





Observed: the first three bounces reach 0.955, 0.526 and 0.308 m; unknown: the restitution e and the drag k. The cell is compiled by cppjit in the session, with Clad loaded as a compiler plugin.

Autodiff with Clad: .backward() for ”any” code

```
// clang -fplugin=/.../libclad.so
// Necessary for clad to work include
#include "clad/Differentiator/Differentiator.h"
double pow2(double x) { return x * x; }

double pow2_darg0(double); // to be filled by clad

int main() {
 auto dfdx = clad::differentiate(pow2, 0);
 // Function execution can happen in 3 ways:
 // 1) Using CladFunction::execute method.
 double res = dfdx.execute(3);

 printf("res is: %f\n", res);
 printf("The derivative code is: %s\n", dfdx.getCode());
 return res;
}
```

Clang Plugin

Tells the plugin to create pow2_darg0.

COMPILER EXPLORER

https://godbolt.org/z/h54nxjWc6

```
performance-test@vv-nuc ~/clad-build-llvm11 $ ./a.out
6.000000
The derivative code is: double pow2_darg0(double x) {
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}
```

Case Studies

The ball setup at scale. A model you can run, a few measurements, unknown dials, a loss, and the compiler's derivative of that loss through the model:

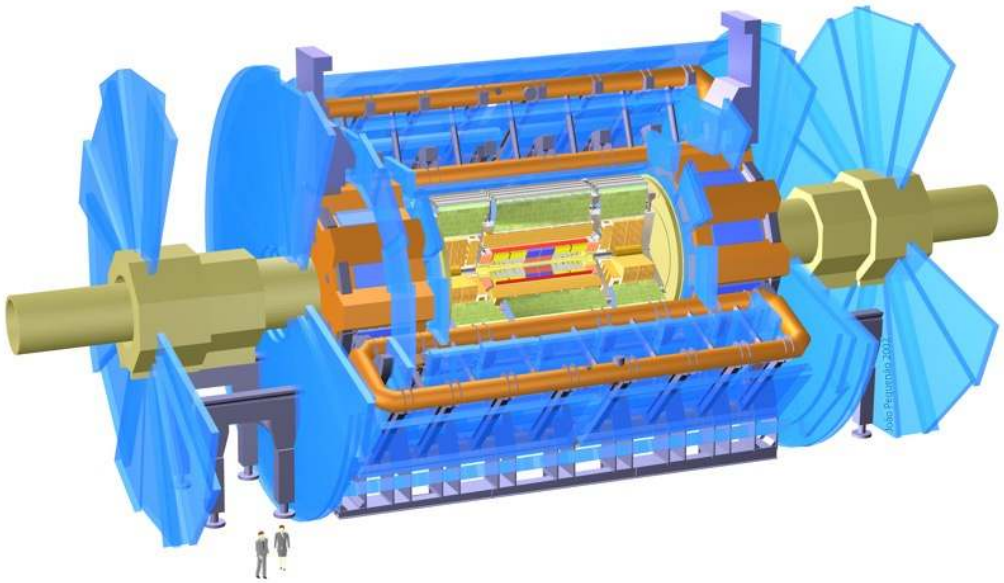
- Data Bound: High Energy Physics
  C++ likelihoods over enormous datasets; fits with hundreds of parameters
- Compute Bound: Cancer Modeling
  A mechanistic agent-based model of CAR T-cell therapy, C++ on BioDynaMo

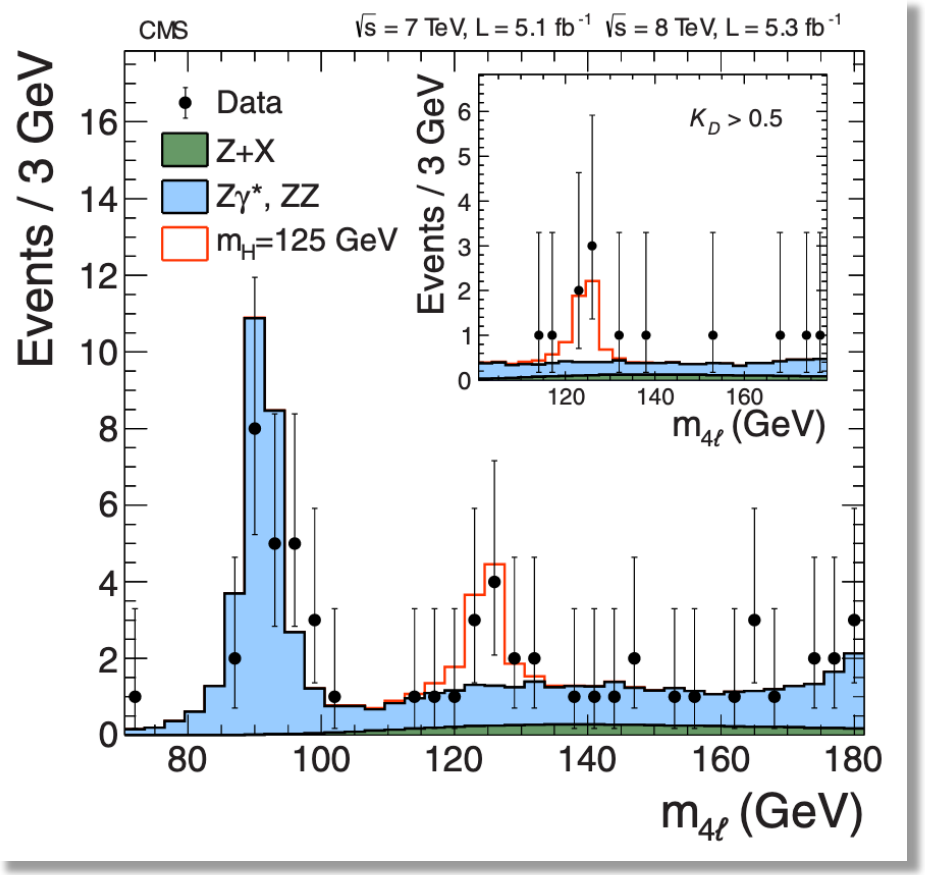


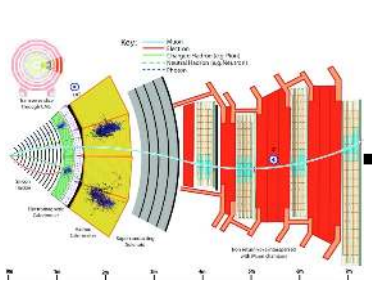
Example 1: High-energy physics

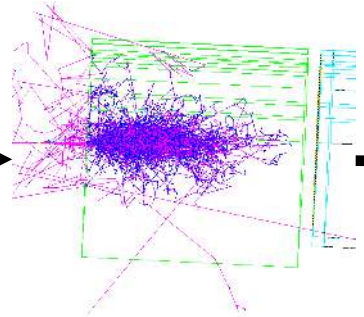
The pressure is data. Call the likelihood as written; transform it with Clad; compose it with JAX.

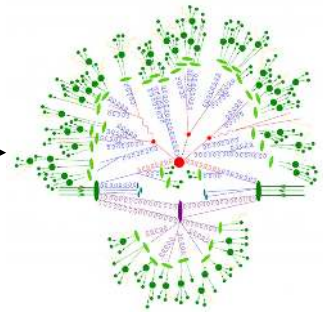
Particles Whisper

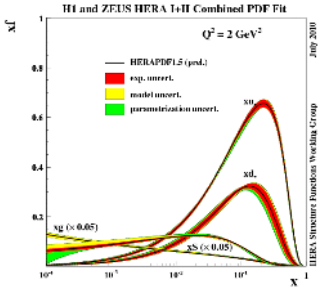


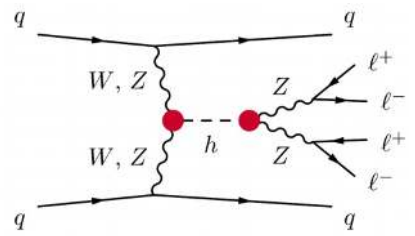












The ball equation but

Likelihoods are central for High Energy Physics

Same thing as the ball, at scale: the likelihood is the loss, the physics parameters are the dials, the recorded events are the data.

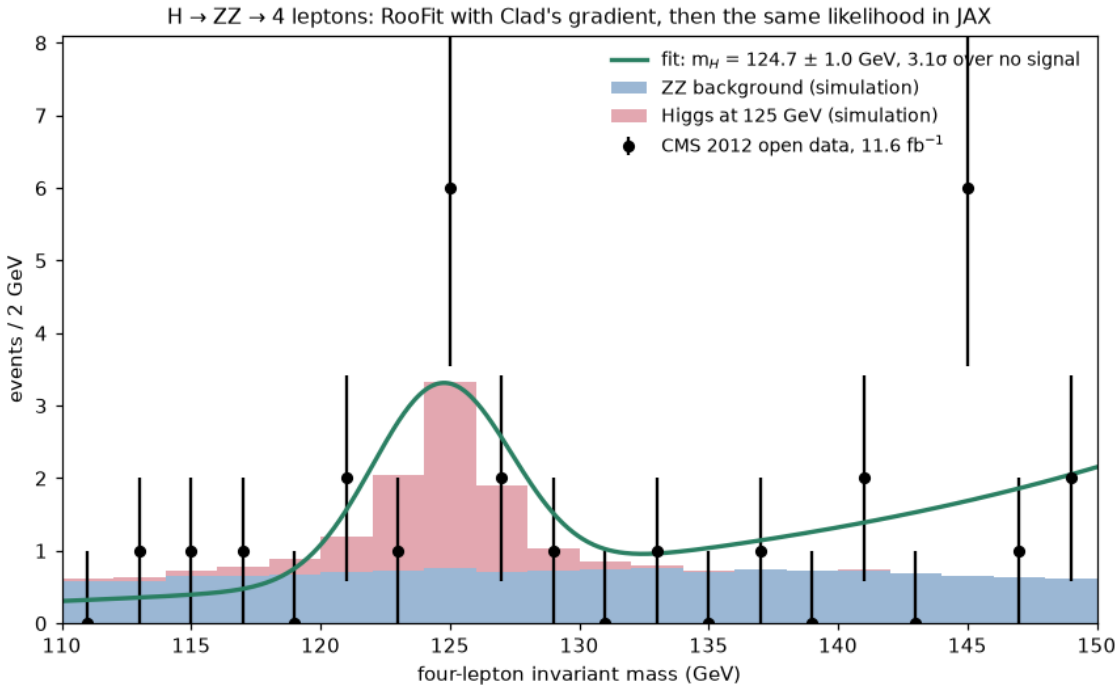
$$L(\vec{n}, \vec{a}|\vec{\eta}, \vec{\chi}) = \prod_{c \in unbinned\ ch} \prod_{i \in obs} \frac{f_c(\vec{x}_{ci}|\vec{\eta}, \vec{\chi})}{\int f_c(\vec{x}_{ci}|\vec{\eta}, \vec{\chi})\, d\vec{x}_c} \cdot \prod_{c \in binned\ ch(analytical)} \prod_{b \in obs} Pois(n_{cb}|\nu(\vec{\eta}, \vec{\chi})) \cdot \prod_{\chi \in \vec{\chi}} c_\chi(a_\chi|\chi)$$

$$\vec{n} : data, \vec{a} : auxilary\ data, \vec{\eta} : unconstrained\ parameters, \vec{\chi} : constrained\ parameters$$

CMS Combine Paper https://arxiv.org/pdf/2404.06614

Discover the Higgs boson, yourself

```
nll = model.createNLL(data, EvalBackend.Codegen())   # the likelihood out as C++; Clad writes its gradient
RooMinimizer(nll).minimize("Minuit2", "migrad")      # the standard minimiser, with that gradient
roofit_nll = jax.custom_vjp(...)                     # JAX: value and Clad's gradient, through a callback
optx.minimise(roofit_nll, optx.BFGS(), start)        # JAX's optimiser; it never sees RooFit
```

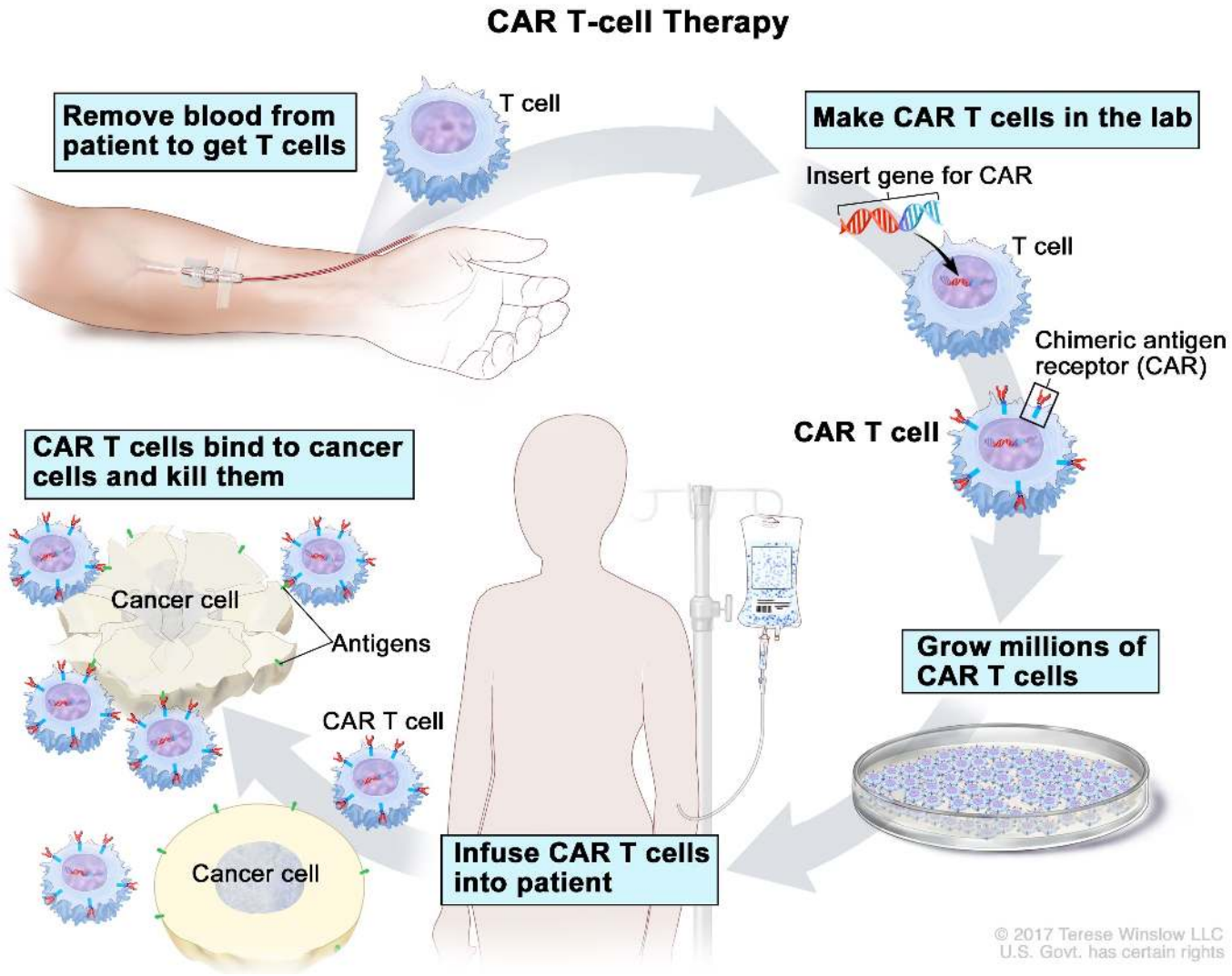


CMS 2012 open data, 11.6 fb^{-1}, the published four-lepton selection (ROOT's own tutorial of the CMS outreach analysis): 28 candidates between 110 and 150 GeV.
RooFit emits the likelihood as C++, Clad writes its gradient, Minuit2 finds m_h = 124.7 ± 1.0 GeV, 9 ± 4 signal events, 3.1σ over no signal, in 62 ms.
Then JAX: the same likelihood as a function with a derivative, its own optimiser, the same minimum to four decimals in 57 steps. Nothing was ported.
Local significance, one window, one selection; the 2012 observation combined more channels and data.

Example 2: Cancer research

The pressure is computation. Call the model as built; compose it from Python; transform its oxygen update, and fit it.

Cell Engineering



Effective in **leukemia** and other **blood** cancers. However, CAR T still **remains limited in solid tumors** due to unique tumor microenvironmental factors.

CARTopiaX

The Problem

- How a tumour organoid responds to CAR T-cell therapy: engineered immune cells that find, bind and kill cancer cells
- A mechanistic agent-based model: every cancer cell and every CAR T-cell is an agent in a 3-D tumour, with its own growth, death, movement and the oxygen it gets (Luque et al., Sci. Rep. 2024)
- **CARTopiaX**: the model implemented in C++17 on BioDynaMo, plus a calibration loop, exploration tools and a Python interface; the model supplies the rules, the platform the infrastructure
- 3,957 cells at the start, 12,372 after 10 days, each updated every 0.1 min with two 50^3 oxygen and signal grids: 14,400 steps a day, 432,000 for the published 30-day protocol; 0.9 h per simulated day here
- From python: the same 109 ms a step as the model's own binary (113 ms); 6 s of start-up once

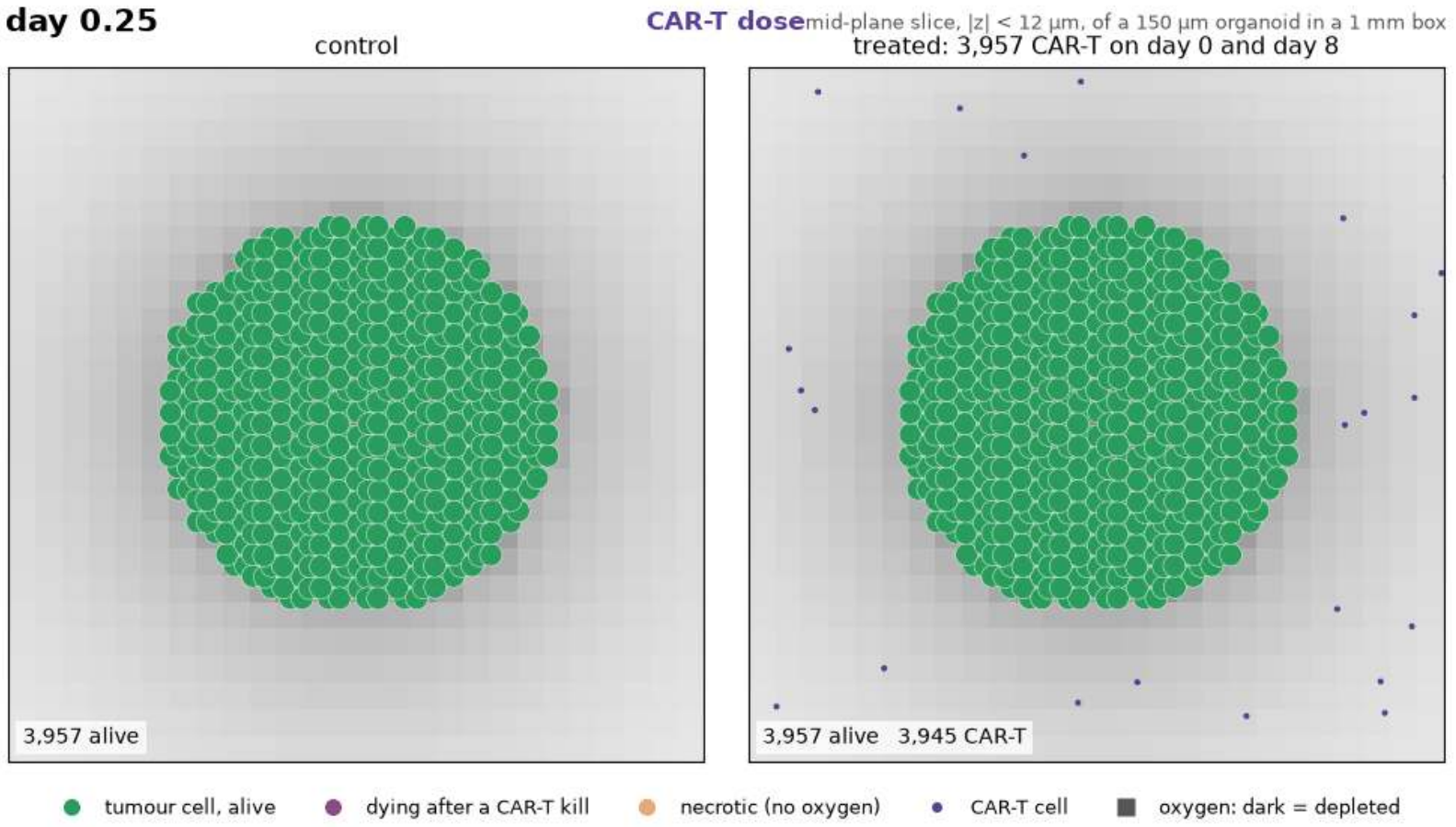
Same Story

```
import cppjit, ctypes
ctypes.CDLL("libbiodynamo.so", ctypes.RTLD_GLOBAL)    # the platform
ctypes.CDLL("libCARTopiaX.so", ctypes.RTLD_GLOBAL)    # the model, as built
cppjit.add_include_path("CARTopiaX/src")
cppjit.cppdef(open("organoid.cpp").read()) # its headers + build, dose, step, count

run = cppjit.gbl.sc26.run_organoid          # (minutes, every, radius_um, dose, seed)
control = run(14400, 360, 150, 0,    42)
treated = run(14400, 360, 150, 3957, 42)
# after 240 h: 12,372 alive tumour cells untreated, 721 treated
```

The model itself, from the session, through CARTopiaX: 12,372 living tumour cells in the untreated organoid after 10 simulated days, 721 in the one dosed with 3,957 CAR T-cells (114 killed, 94% fewer alive). Headers parsed in 6 s; the run averaged 0.23 s a step on 4 threads as the organoid grew, and one simulated day is 14,400 steps — this is why it is C++.

Modelling CAR T-Cell Therapy

CARTopiaX



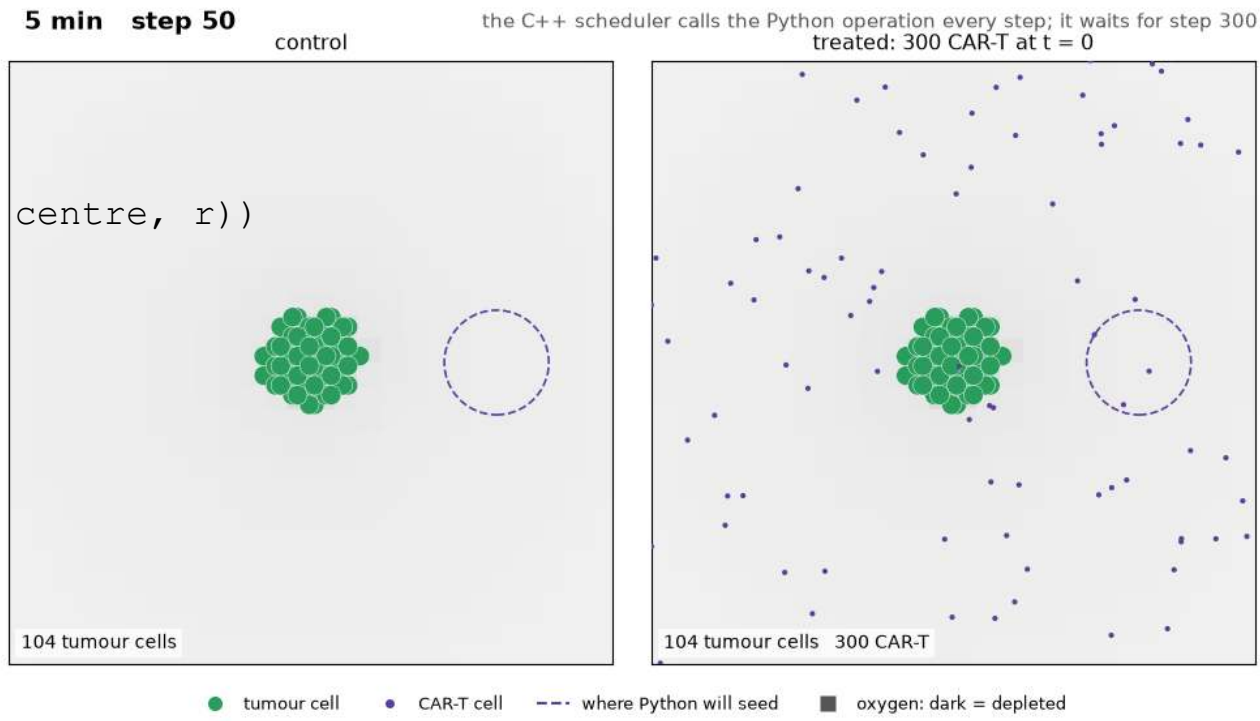
Luque's CAR T-cell model (Luque et al., Sci. Rep. 2024), run through CARTopiaX, the BioDynaMo platform built around it. The published protocol: a 150 µm organoid, 3,957 CAR T-cells at day 0 and day 8; after 10.0 simulated days, 12,372 living tumour cells untreated against 721 treated. The curve you saw fitted two slides ago came out of this same model at small size (60 µm), from this session. Every dot is one cell with its own rules: a tumour cell reads the oxygen in its voxel and divides, rests or dies by it; a CAR T-cell moves up a signal the tumour secretes, binds a neighbour within 14 µm and kills with a probability; the oxygen field is solved every substep. Nothing here is a curve fitted to data: it is the mechanism, run. Day 10 is the trough, not the outcome: by then the CAR T-cells are nearly spent, and in the model (and in CARTopiaX's own 30-day replication of the paper) the tumour regrows once they are gone.

A second organoid from a Python class

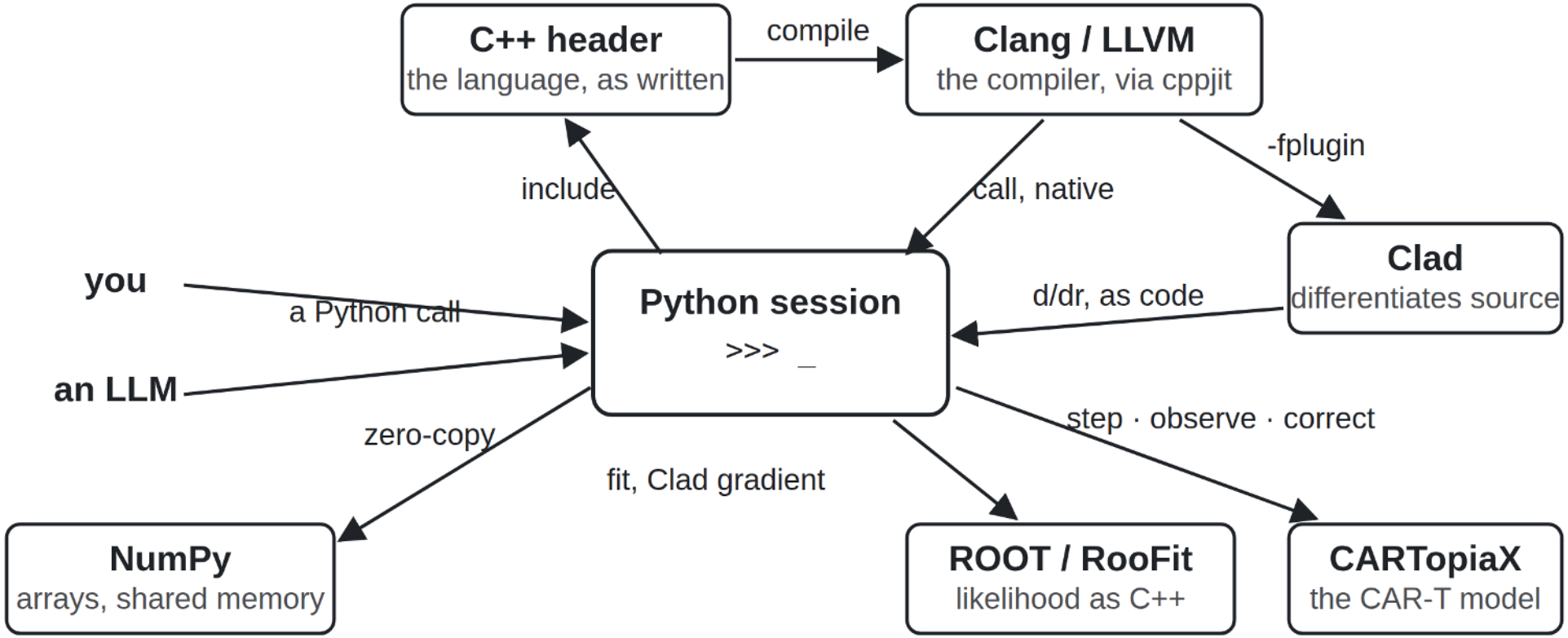
```
class SeedSecondOrganoid(sc26.PyOperation): # a BioDynaMo operation, in Python
  def Run(self):                            # the C++ scheduler calls it every step
    sim = bdm.Simulation.GetActive()
    if self.done or sim.GetScheduler().GetSimulatedSteps() < self.at_step:
      return
    ctxt = sim.GetExecutionContext()
    for pos in bdm.CreateSphereOfTumorCells(self.radius): # the model's packing
      p = bdm.Real3(pos); p[0] += self.centre[0]
      cell = bdm.TumorCell(p)                   # the model's own agent class
      cell.AddBehavior(bdm.StateControlGrowProliferate())
      ctxt.AddAgent(cell)
   self.done = True
```

```
op = bdm.Operation("seed second organoid (python)", 1)
op.AddOperationImpl(bdm.kCpu, SeedSecondOrganoid(at_step, centre, r))
sim.GetScheduler().ScheduleOp(op)# like any C++ operation
```

Steered from Python: a class inheriting BioDynaMo's operation base, scheduled like any C++ operation; at minute 30 the scheduler called it and it seeded a second organoid of 104 cells from the model's own TumorCell and behaviour classes at [160.0, 0.0, 0.0] µm. After 2 h: 208 tumour cells in the control, 208 in the treated (299 of 300 CAR T-cells alive).

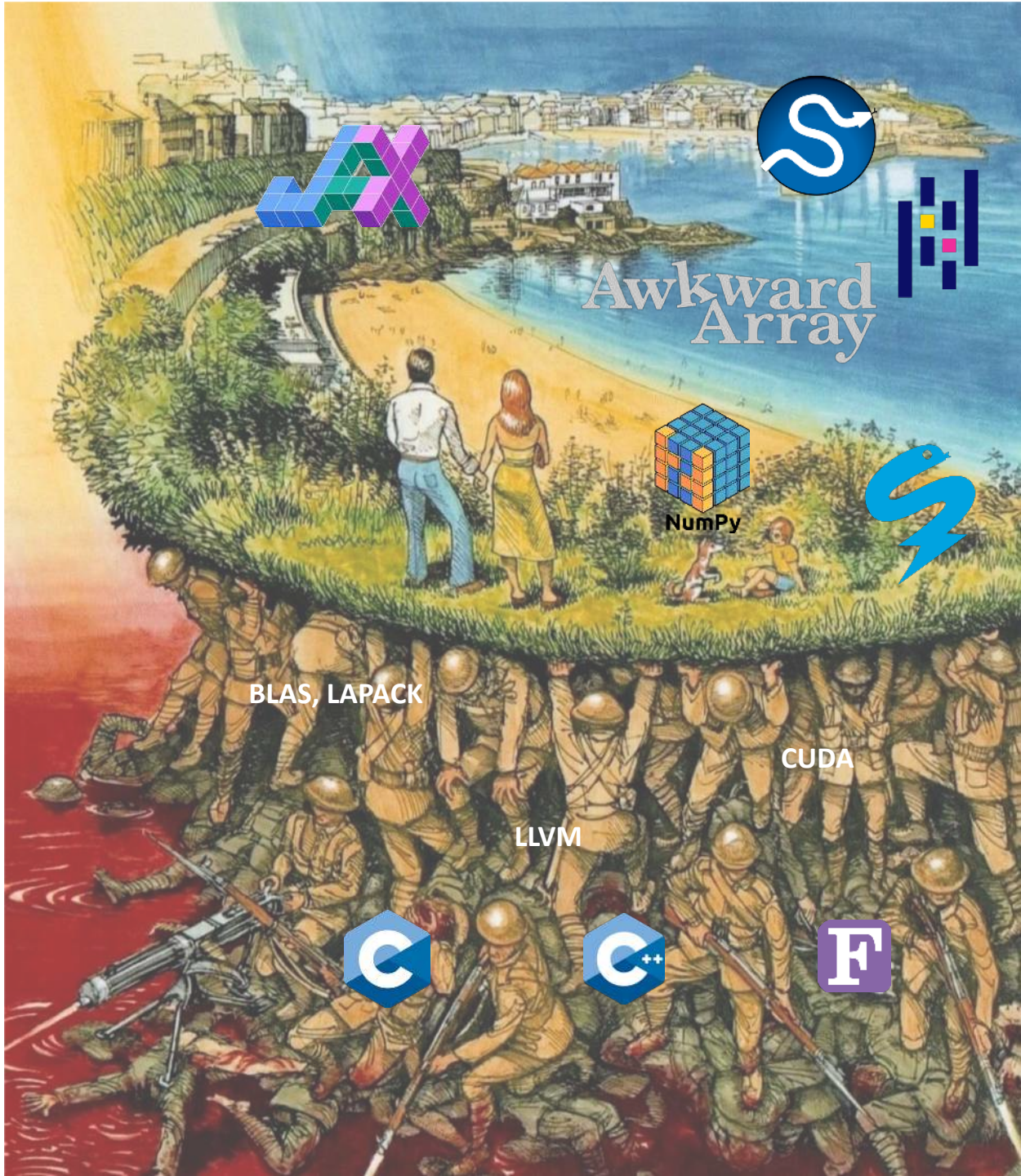


Architecture: one session, the compiler inside it



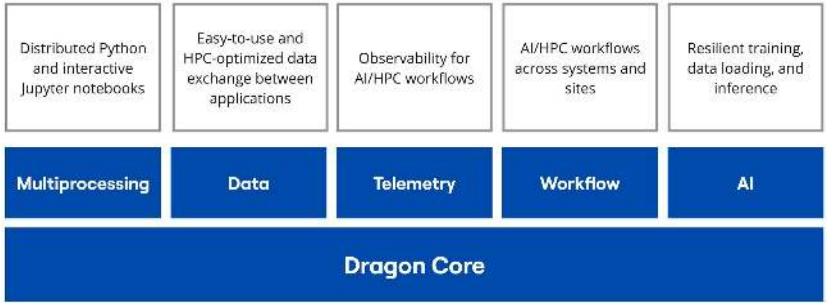
Example 1 on the map: the likelihood called as written (ROOT / RooFit), transformed by the compiler (Clad, a plugin of the session's Clang), composed with JAX. Example 2 walks the same three edges with a cancer model.

Thank you!



DragonHPC

An open-source composable distributed runtime for scalable, resilient HPC and AI applications and workflows, built on standard Python interfaces, especially `multiprocessing`. Scale from a laptop to a supercomputer.



Docs: https://dragonhpc.github.io/dragon/doc/_build/html/

GitHub: DragonHPC/dragon

cuPhoton

An open-source NVIDIA CUDA-X toolkit of GPU-accelerated workflows for astronomical imaging and X-ray analysis. Data stays on the GPU from sensor read through classification.

14,900× peak speedup in FITS image loading vs. an x86 CPU baseline (64 GB200 GPUs)

xDataReader → xRep → xPois → xFit → xScan

load · align · subtract · fit · classify

Blog: https://developer.nvidia.com/blog/faster-scientific-image-analysis-with-nvidia-cuphoton/

GitHub: NVIDIA/cuPhoton

Tutorial notebook: link available at start of tutorial