Sign up for the lab platform:

nvda.ws/XXXXXXX

Join the Zoom:

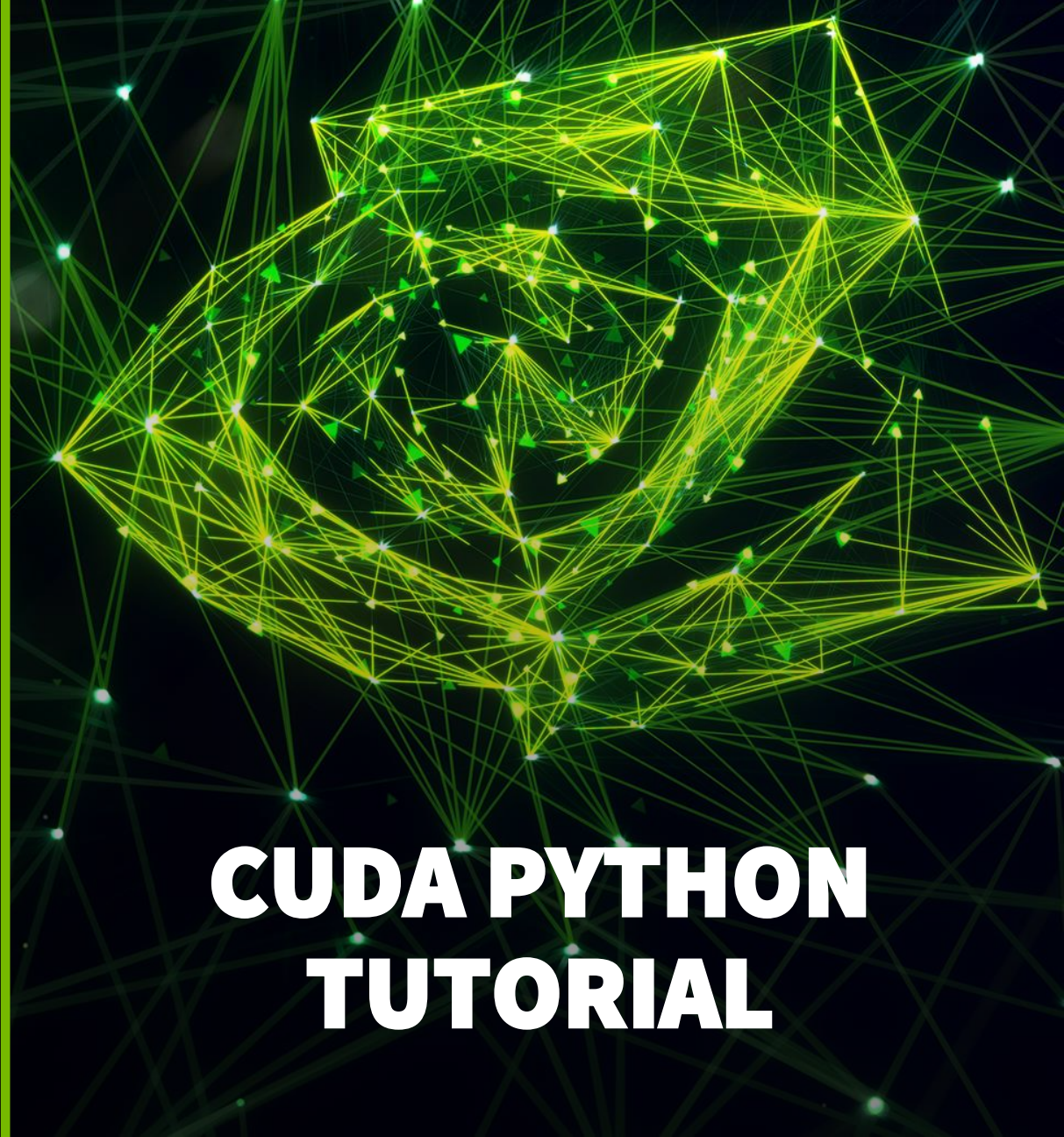
nvda.ws/XXXXXXX

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

nvda.ws/XXXXXXX

WIFI: SSIDSSIDSSIDSSID

PW: PASSWORDPASSWORD



CUDA PYTHON TUTORIAL

Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD

Bryce Adelstein Lelbach
Principal Engineer
Vanguard Programming

Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD

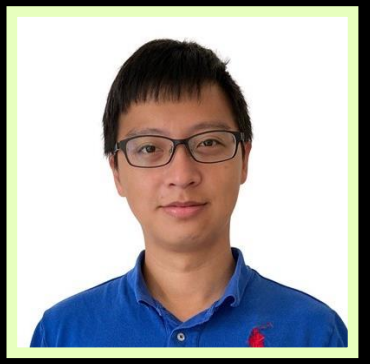
Bryce Adelstein Lelbach
Principal Engineer
Vanguard Programming

Andy Terrel
Product Manager
CUDA Python

Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD

Bryce Adelstein Lelbach
Principal Engineer
Vanguard Programming

Andy Terrel
Product Manager
CUDA Python

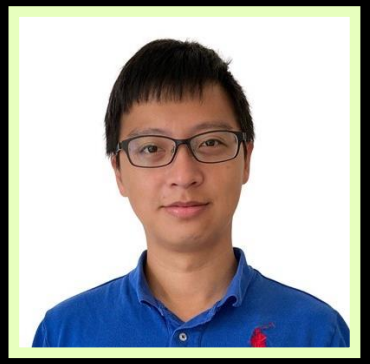


Leo Fang
Senior Software Engineer
CUDA Python

Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD

Bryce Adelstein Lelbach
Principal Engineer
Vanguard Programming

Andy Terrel
Product Manager
CUDA Python



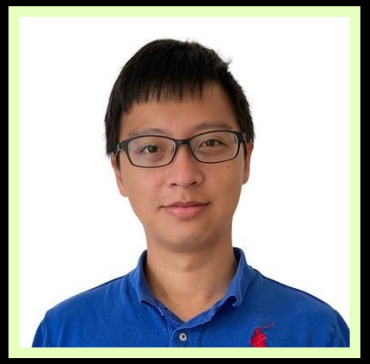
Leo Fang
Senior Software Engineer
CUDA Python

Mike O'Keeffe
Senior Solutions Architect
Higher Education & Research

Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD

Bryce Adelstein Lelbach
Principal Engineer
Vanguard Programming

Andy Terrel
Product Manager
CUDA Python



Leo Fang
Senior Software Engineer
CUDA Python

Mike O'Keeffe
Senior Solutions Architect
Higher Education & Research

Eliot Eshelman
Senior Account Manager

NVIDIA.

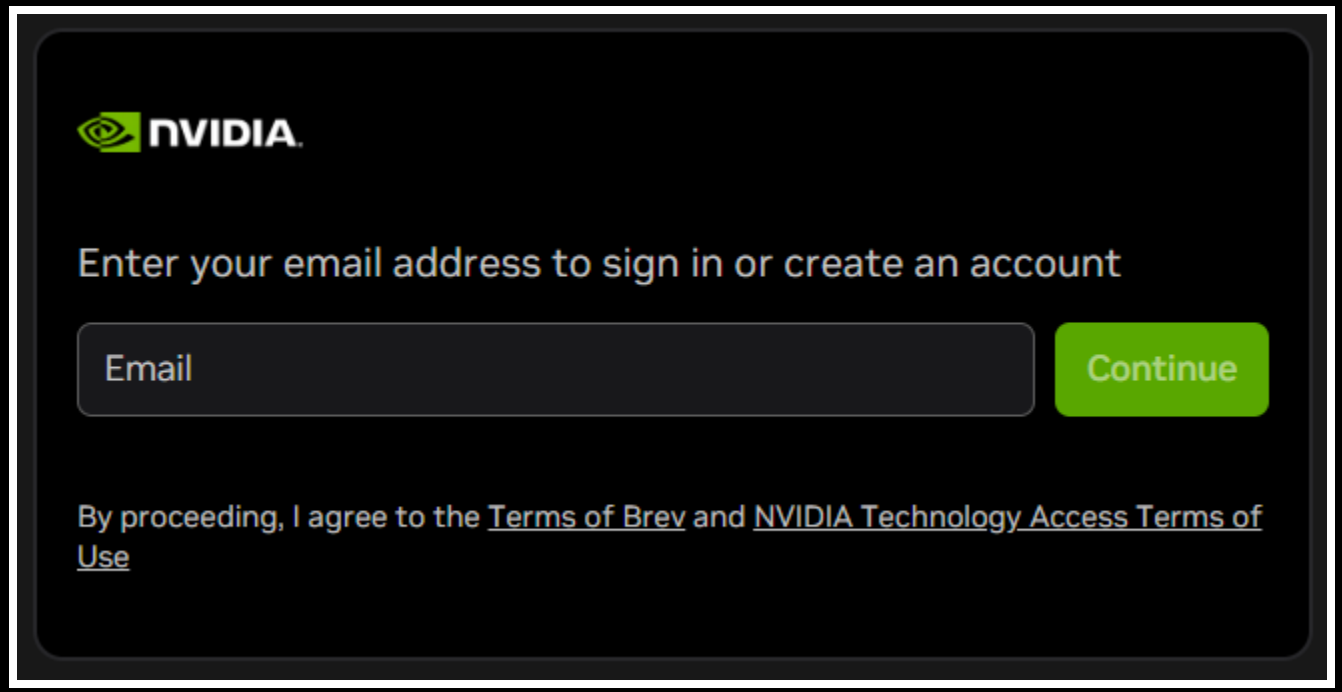
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD

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

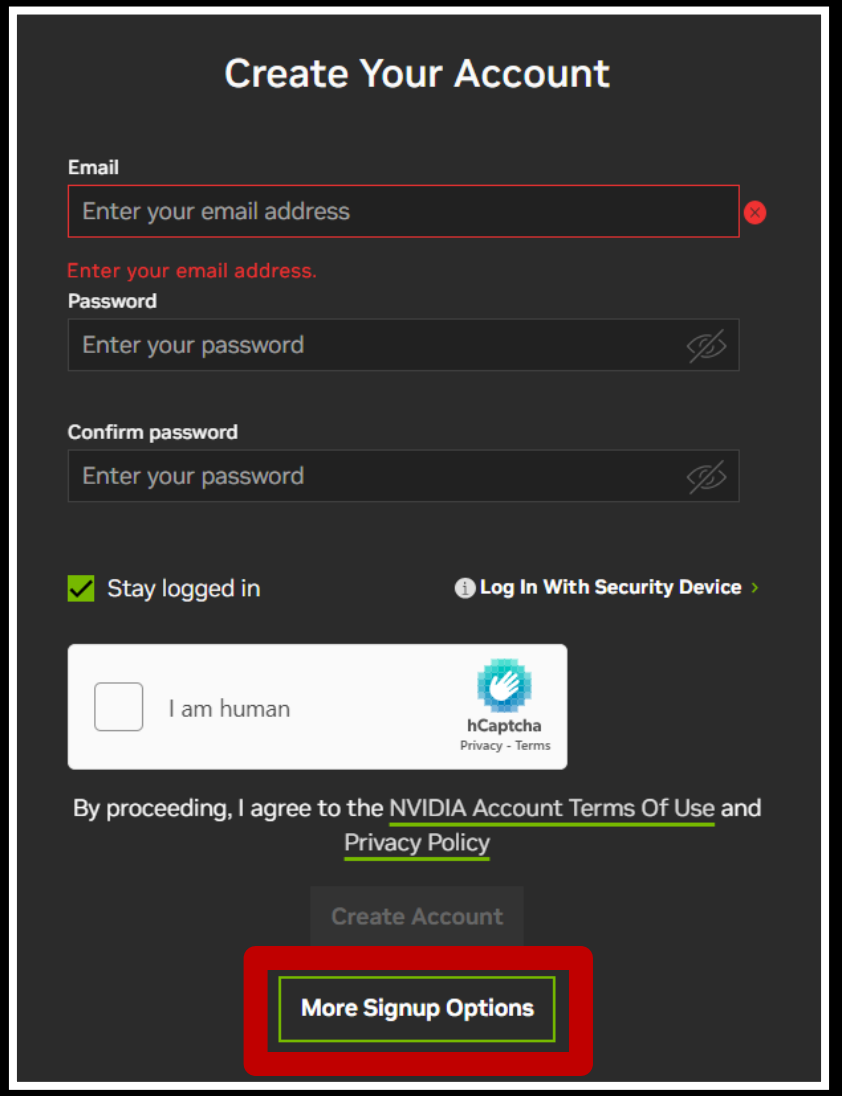
Recommended browser: Chrome (28+).

Javascript and WebRTC must be enabled.

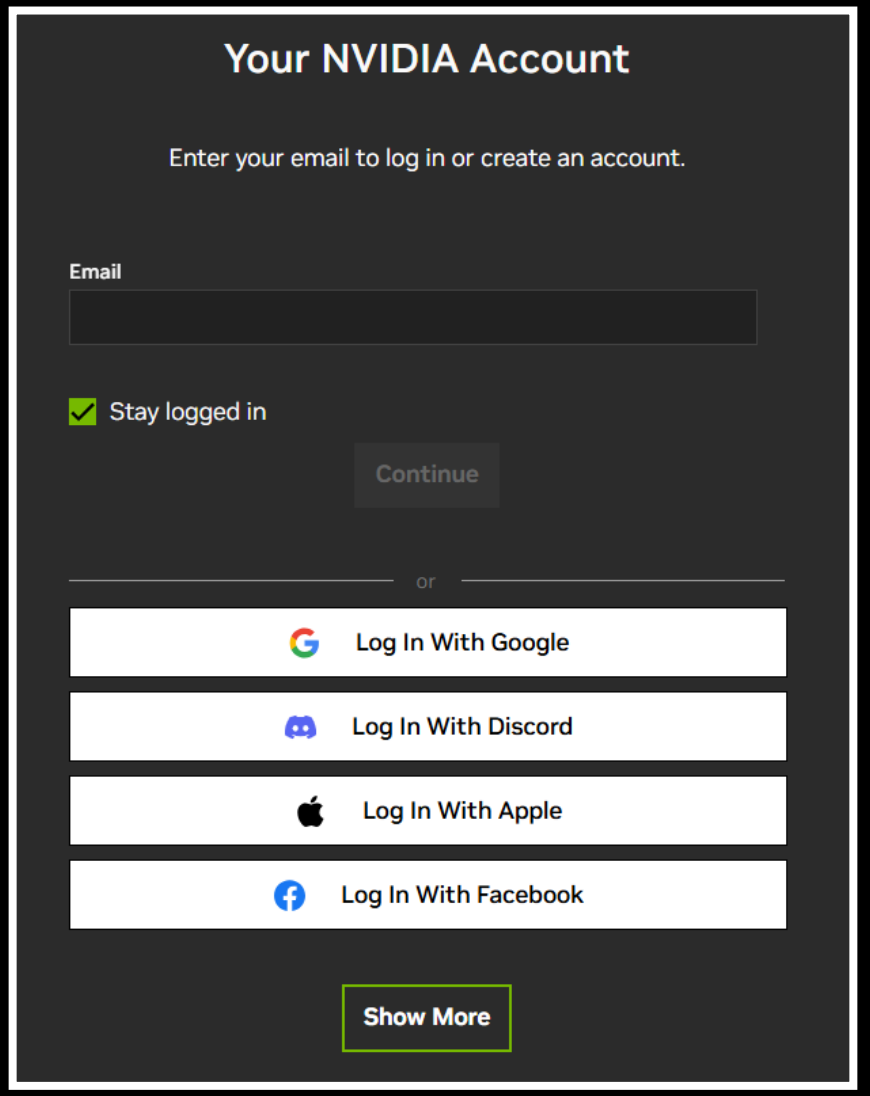
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



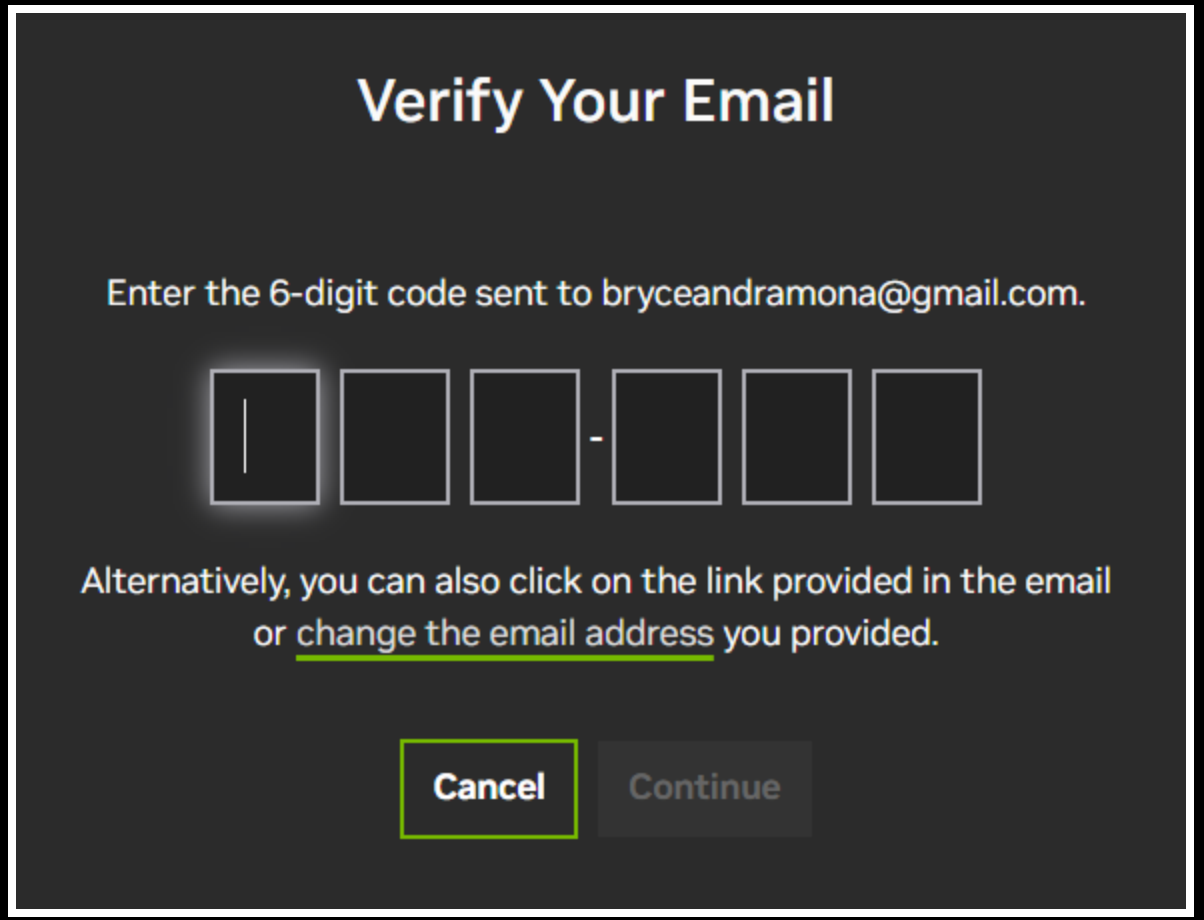
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD

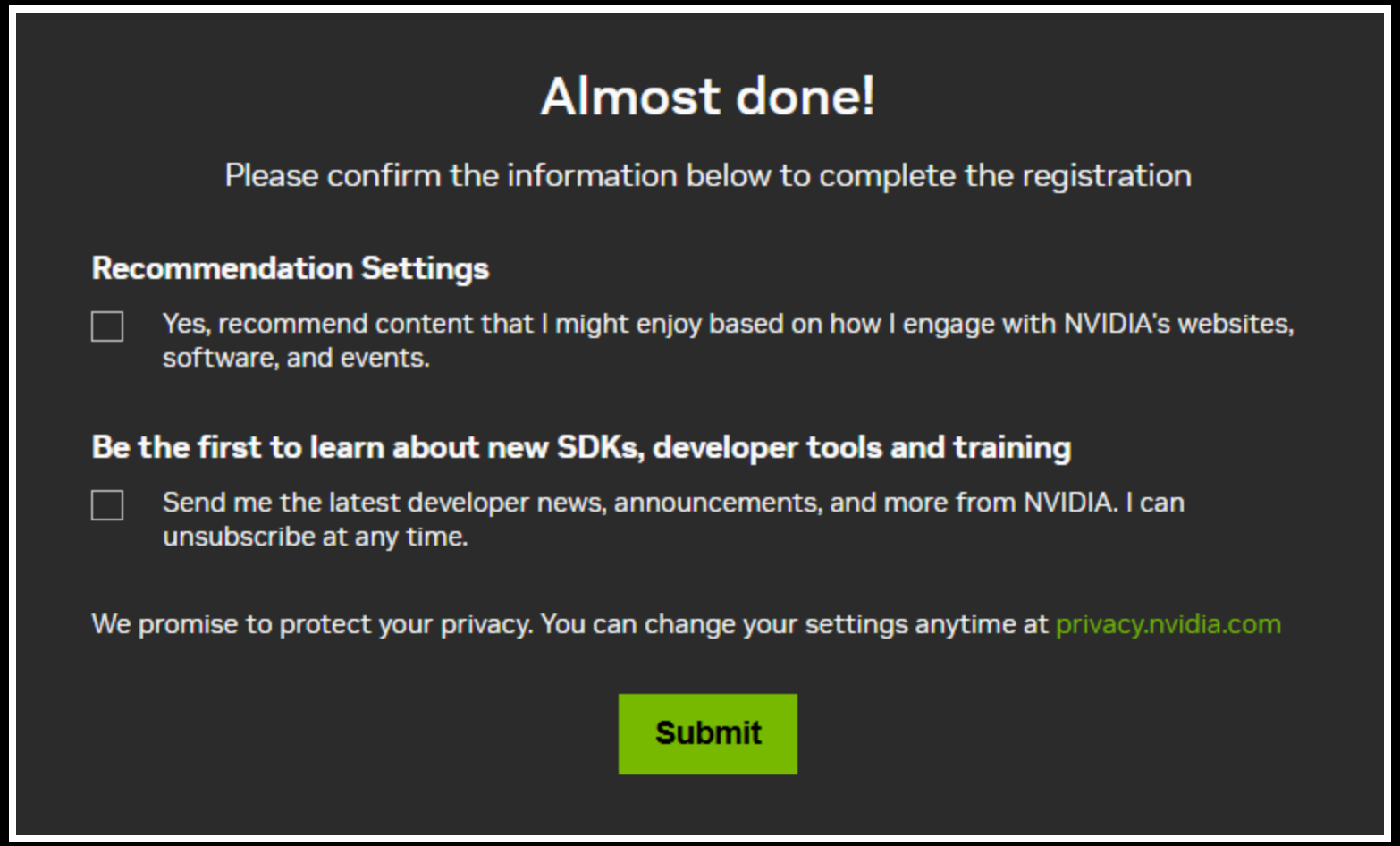


Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



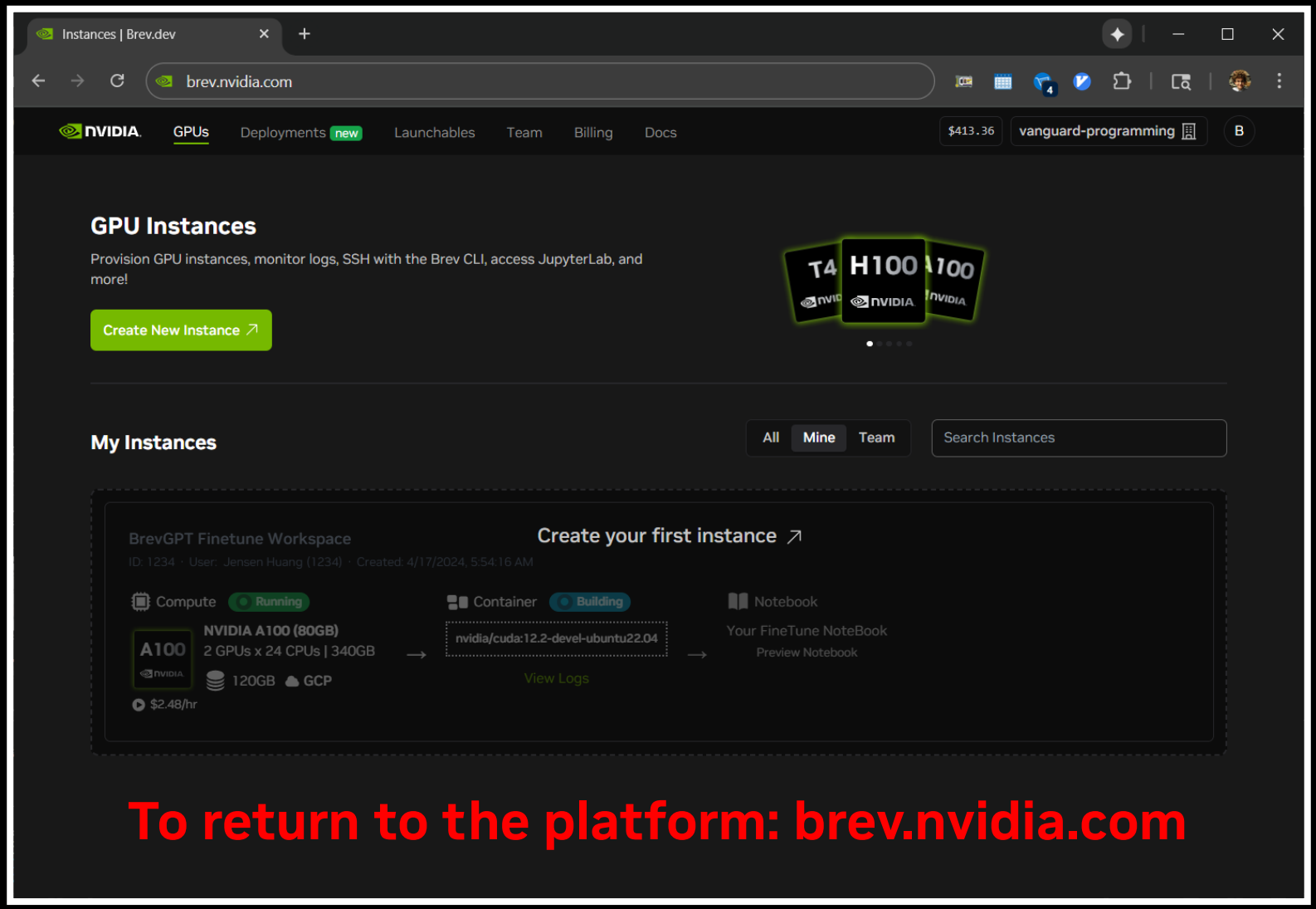
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



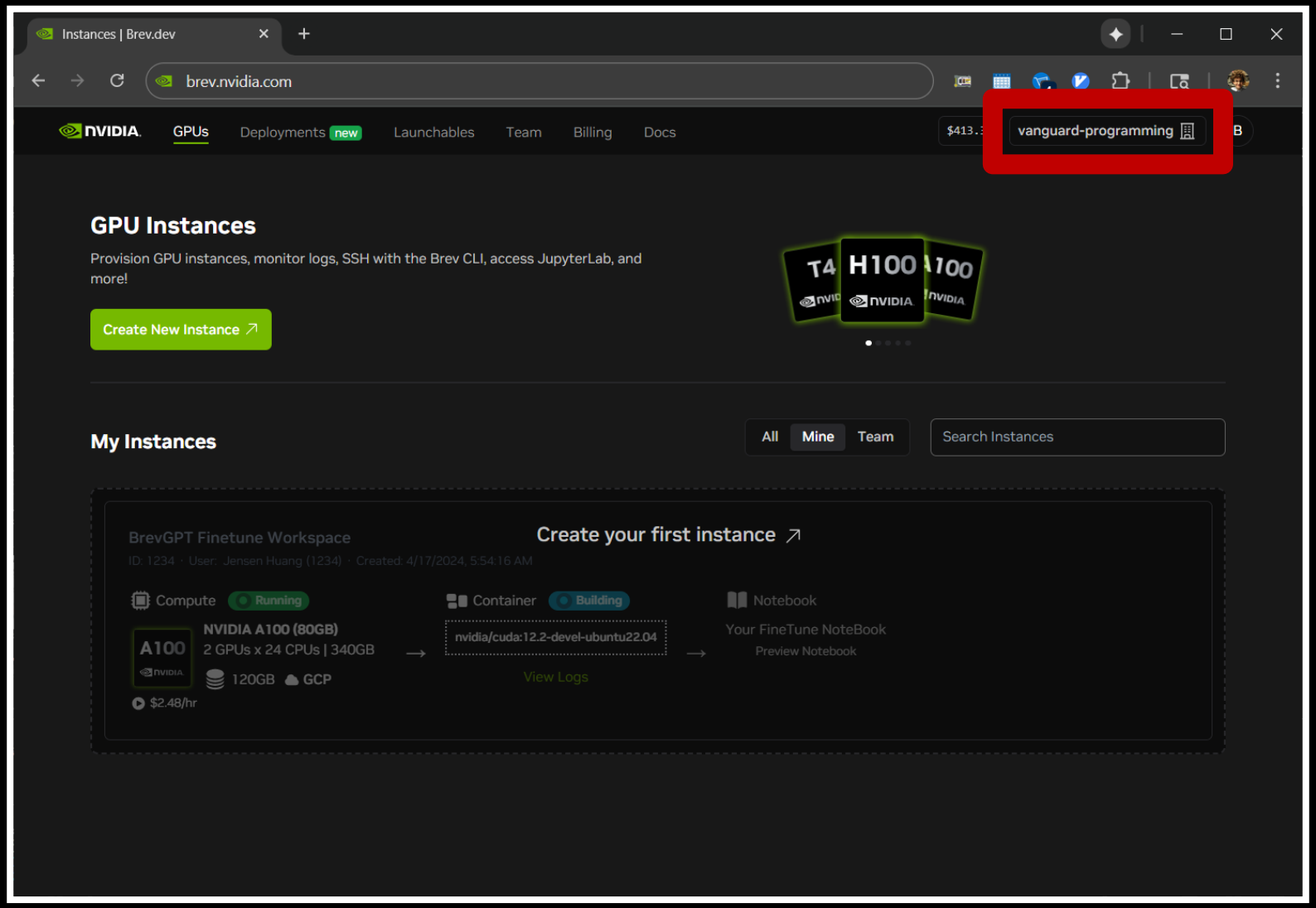


Please enter your first & last name here!

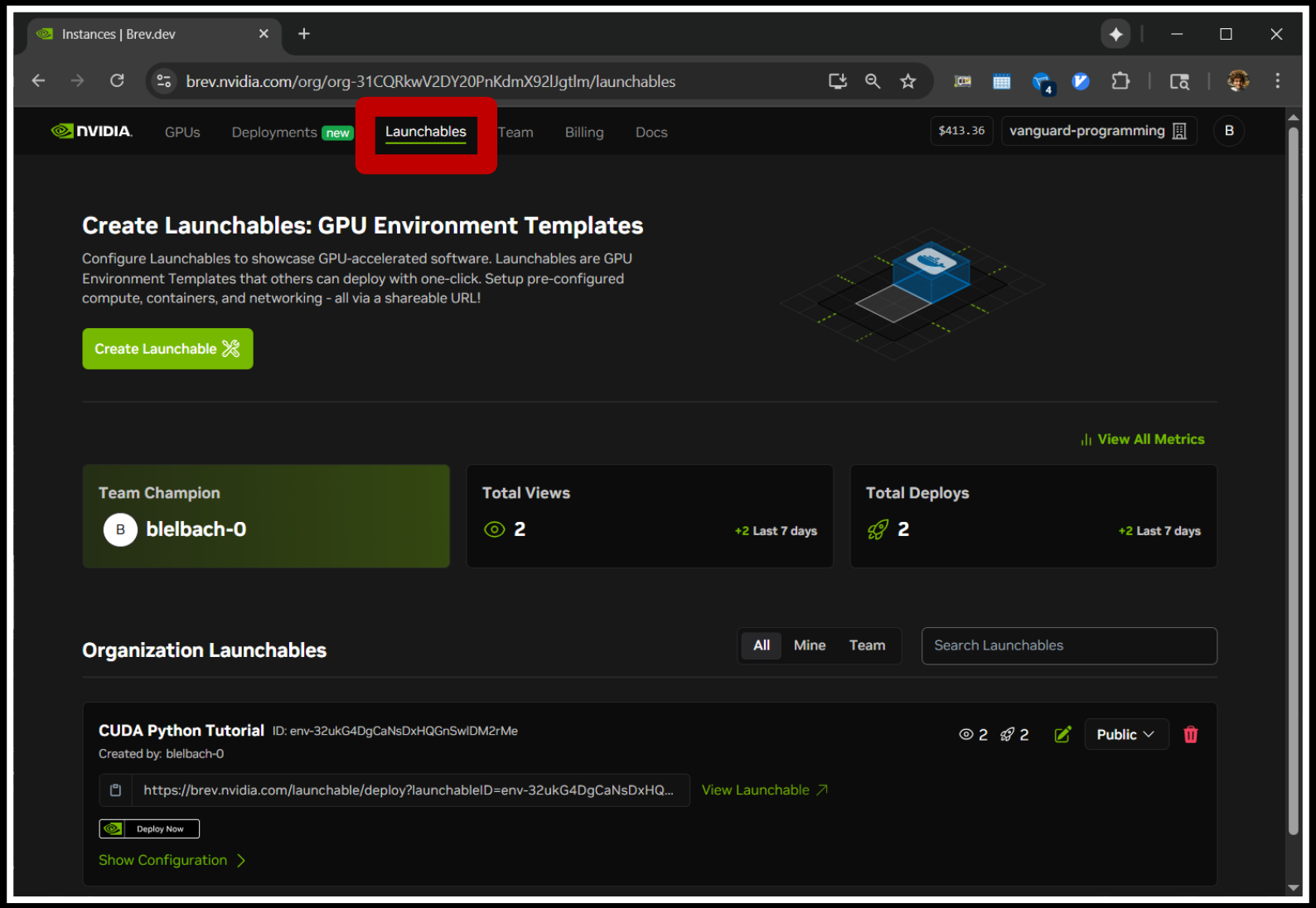
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



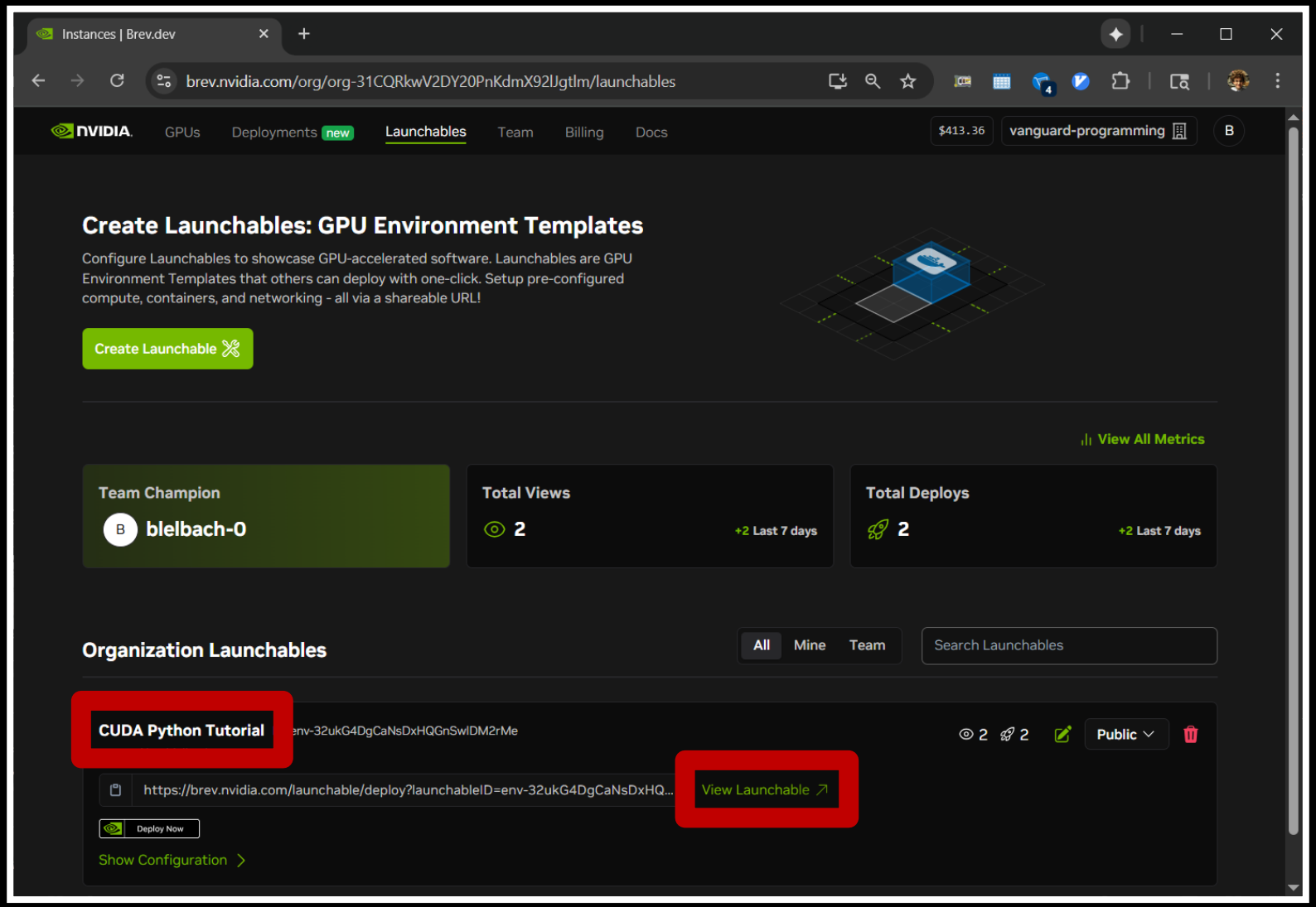
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



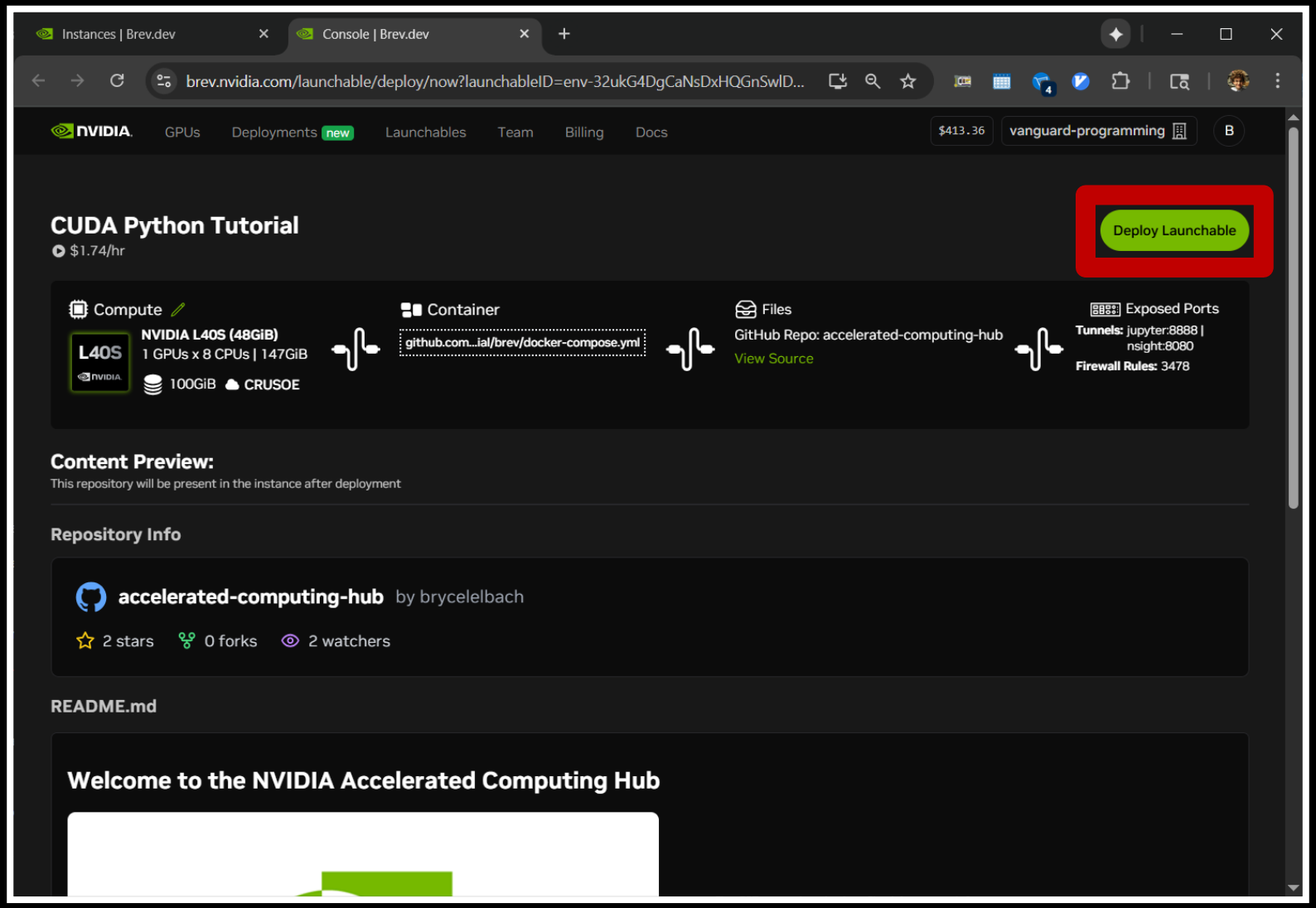
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



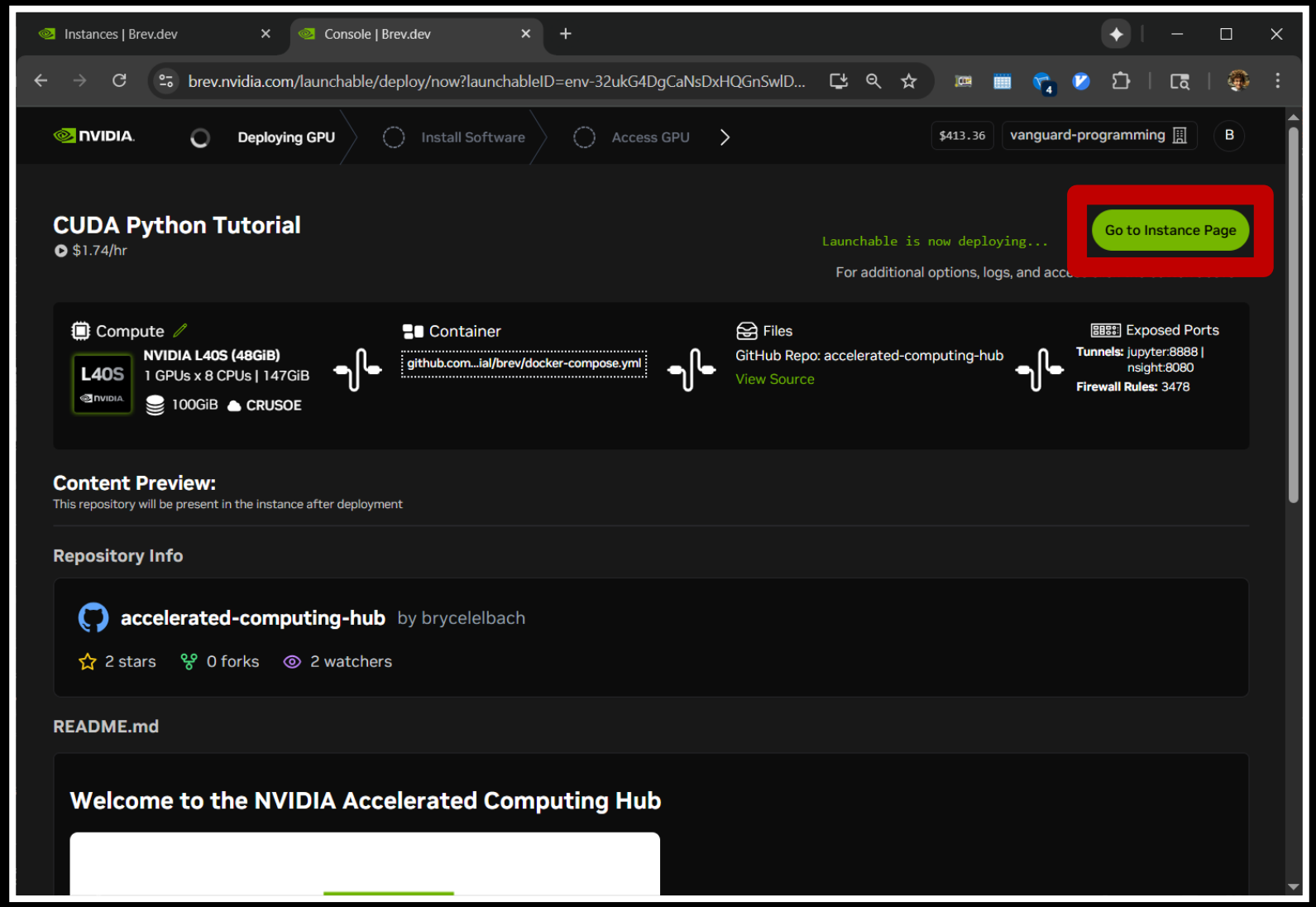
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



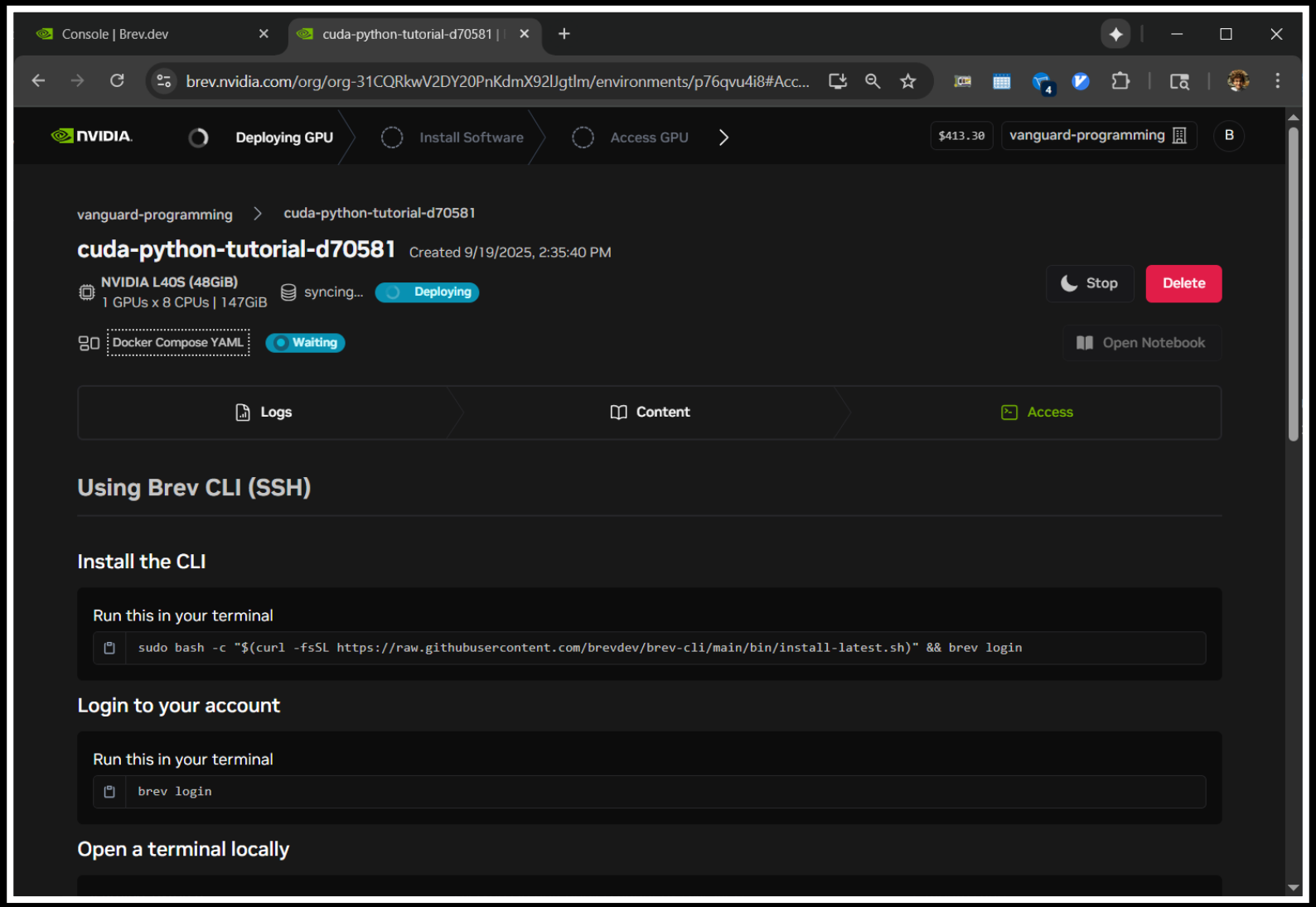
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



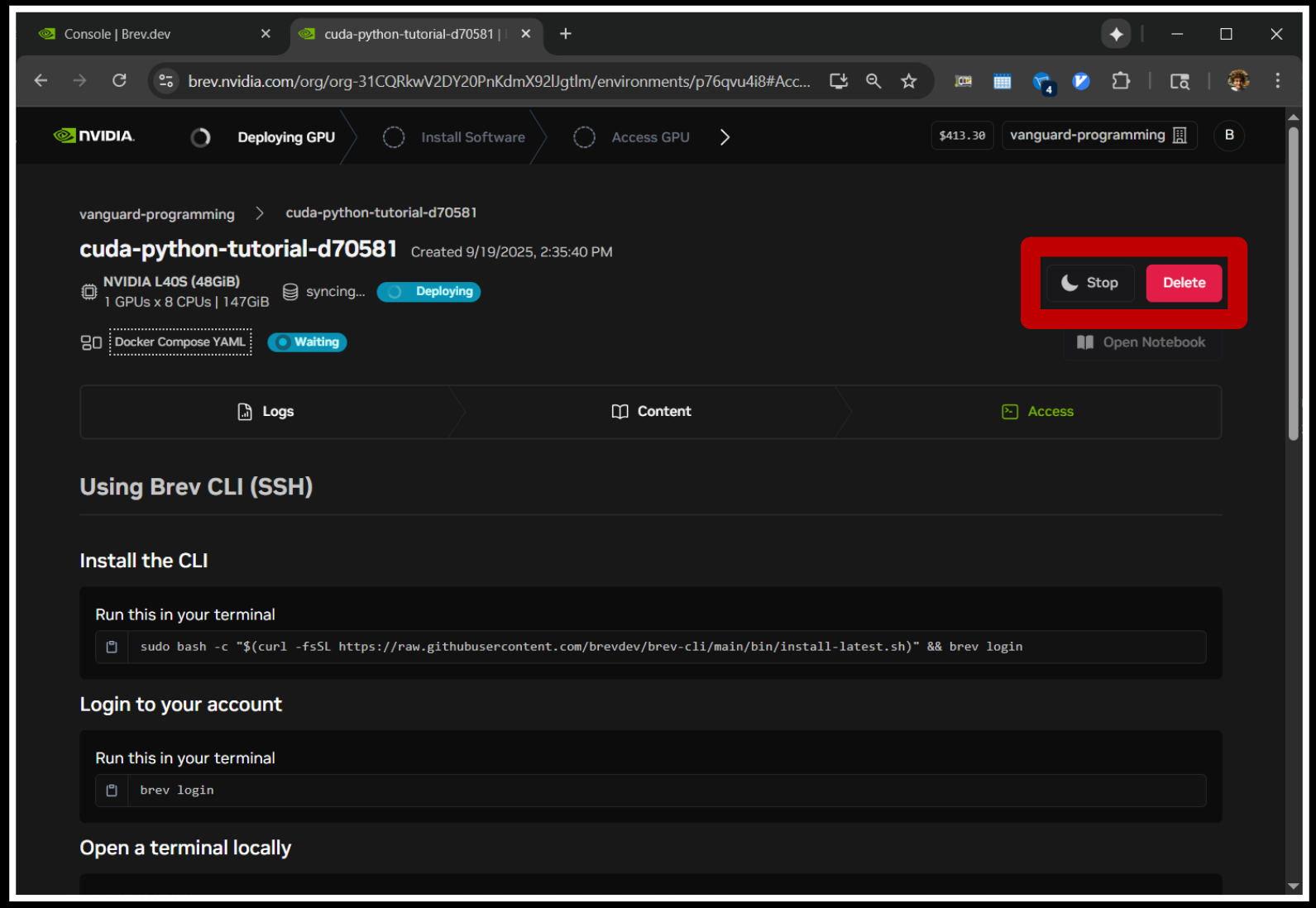
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



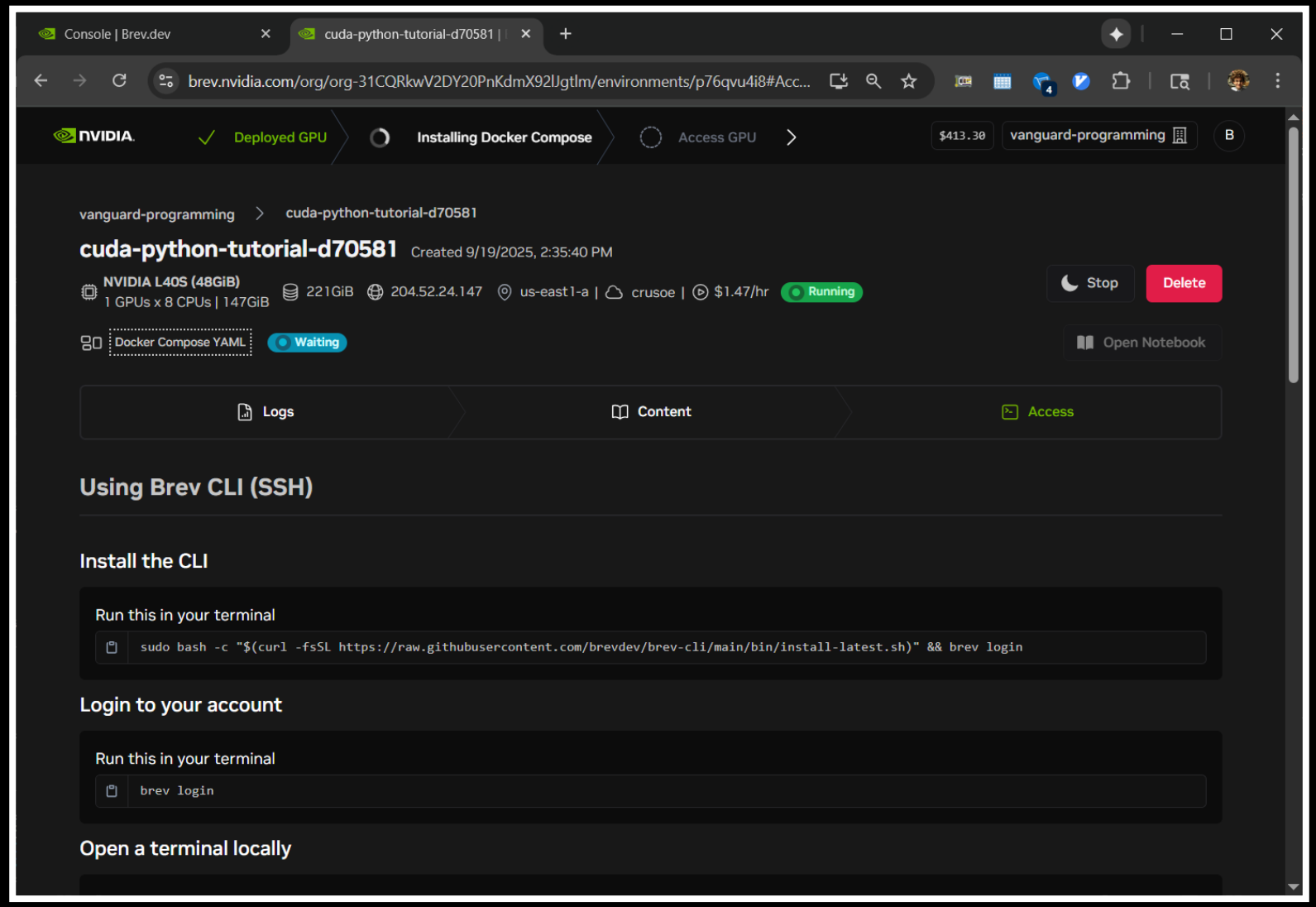
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



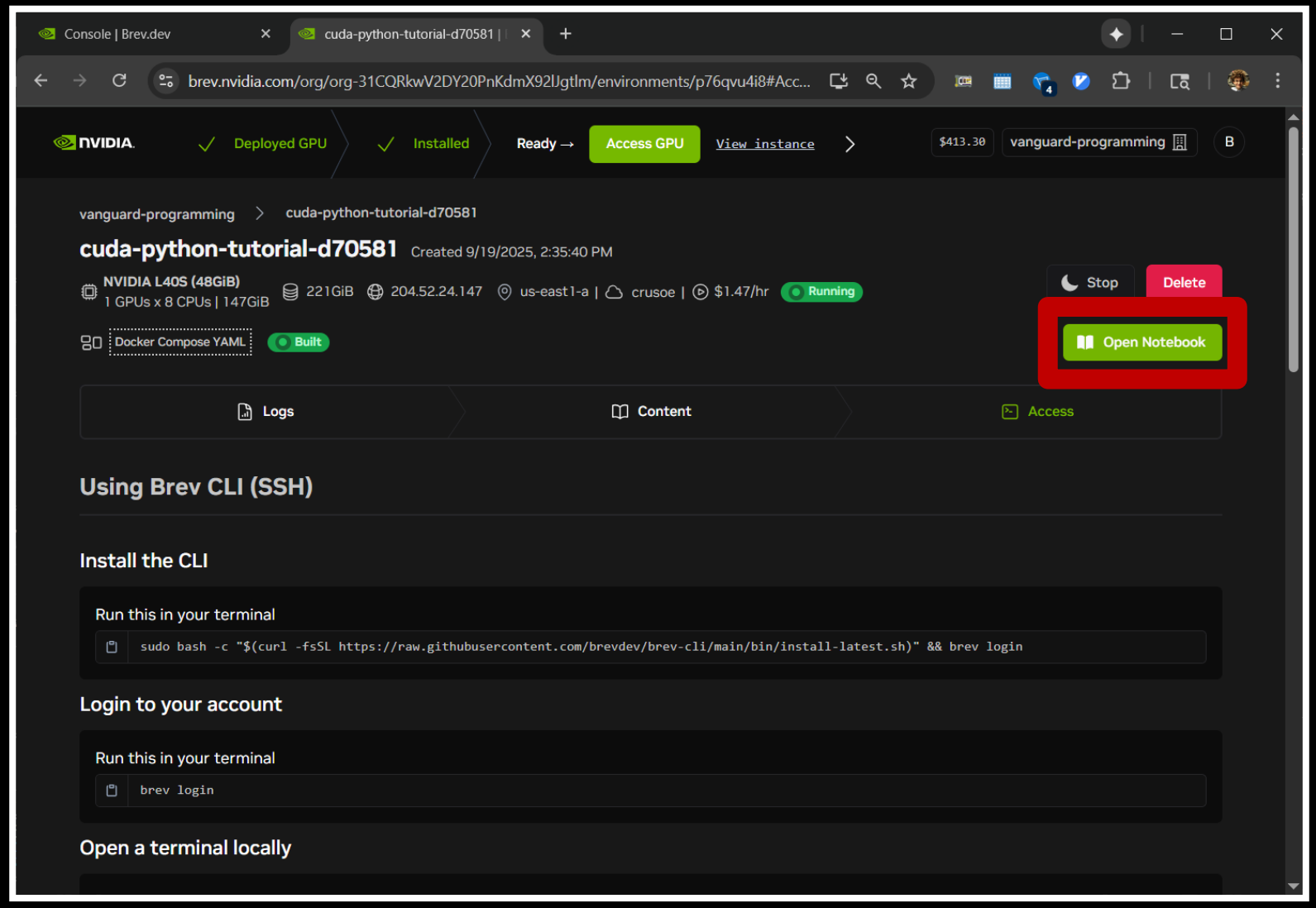
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



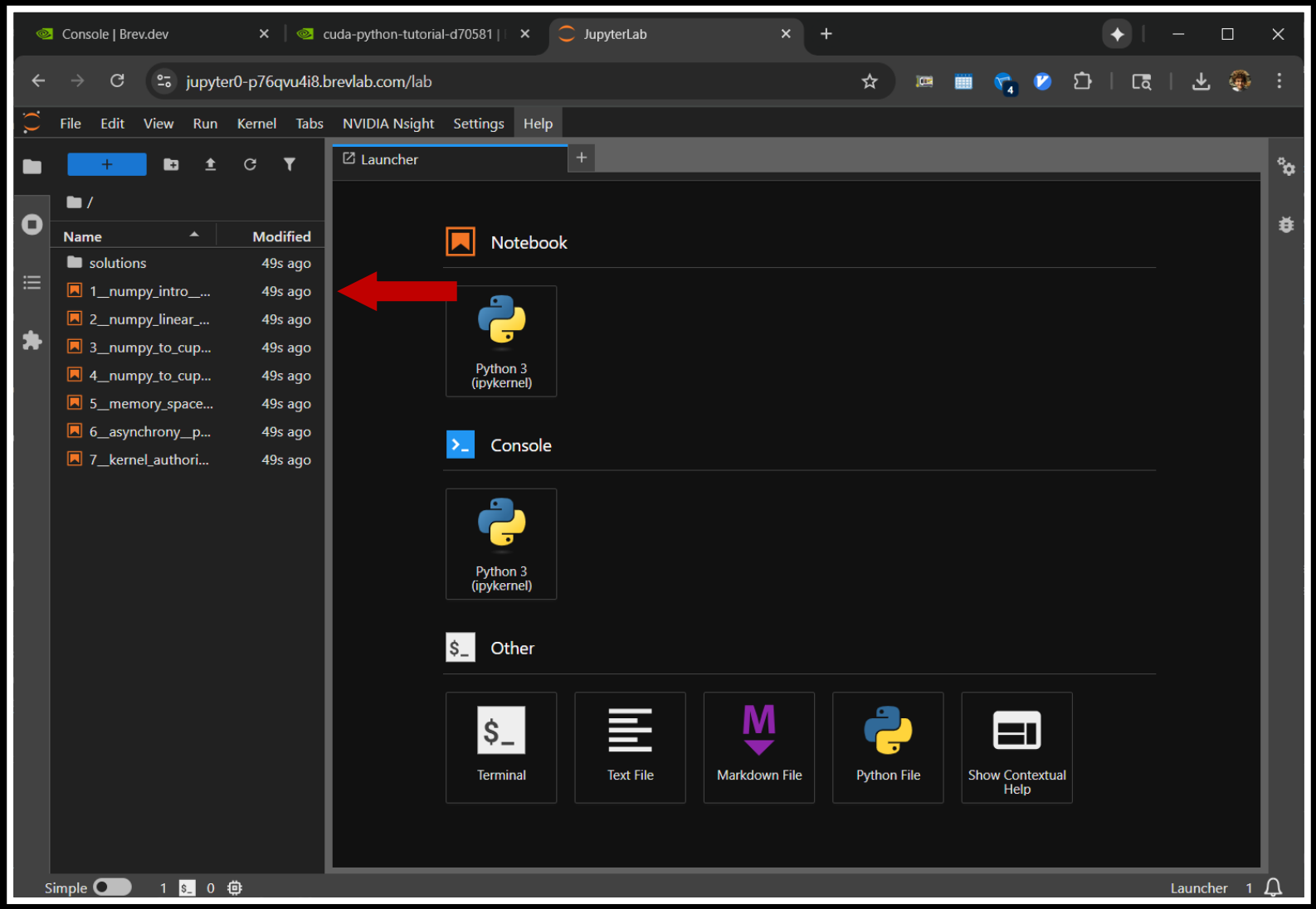
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



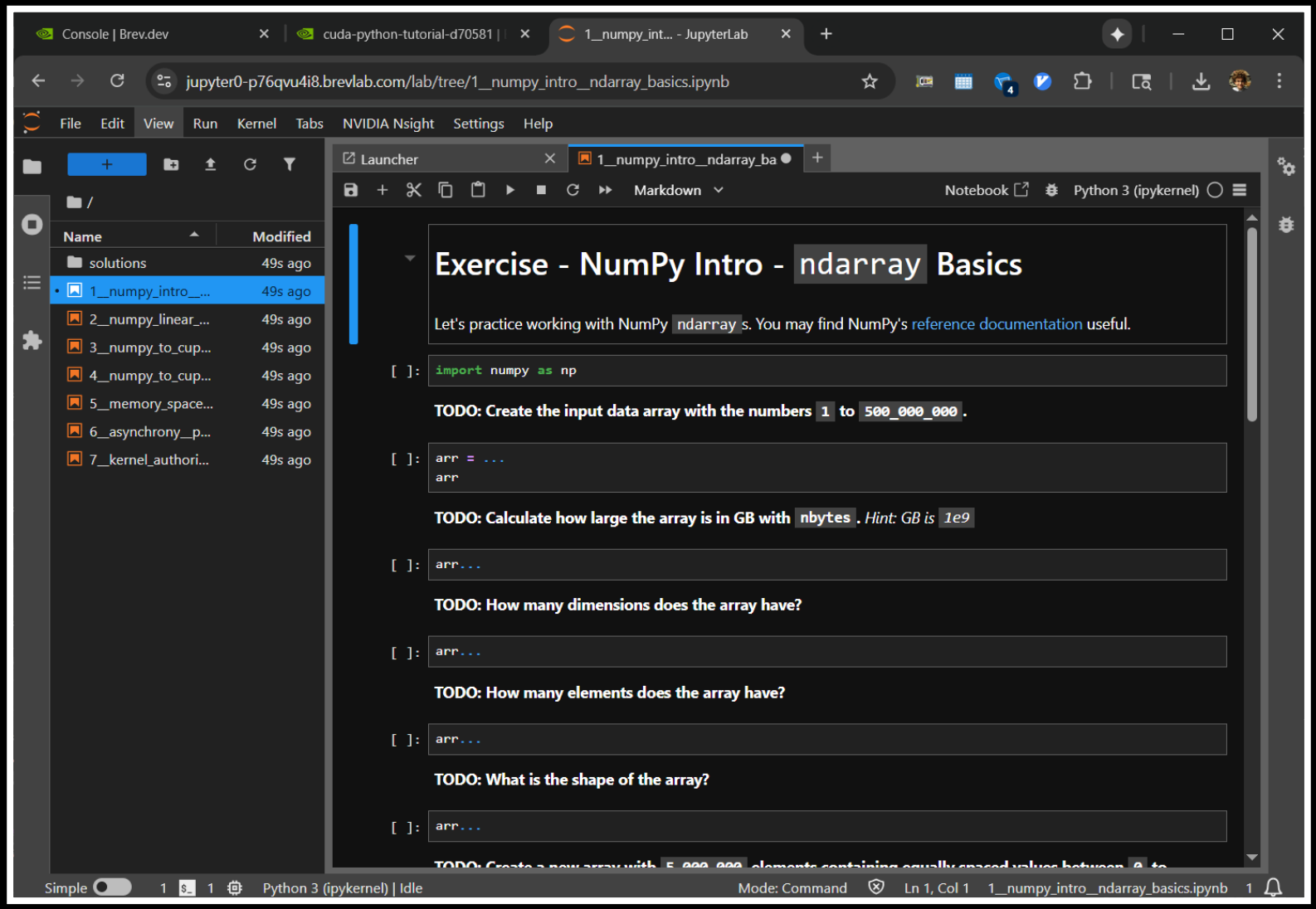
Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



Lab: nvda.ws/XXXXXXX Q&A: nvda.ws/XXXXXXX WIFI: SSIDSSIDSSID PW: PASSWORD



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.

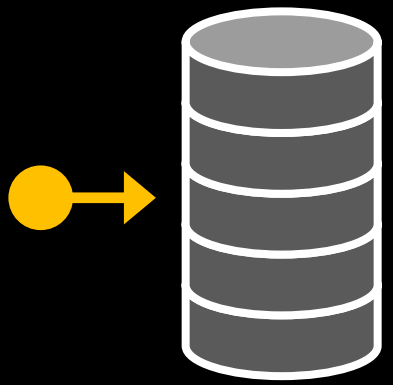
Every HPC and ML library references NumPy in some way!

NumPy specifies:

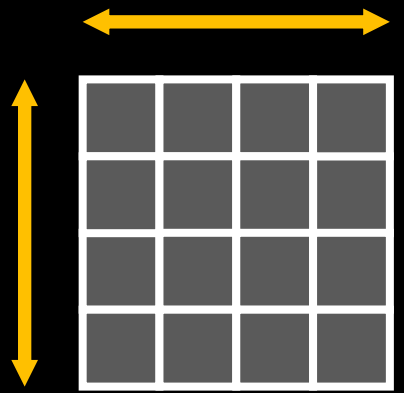
- `ndarray`: Multidimensional array type.
- Array indexing and slicing.
- A common set of array functions.
- How to define your own array functions.

ndarray

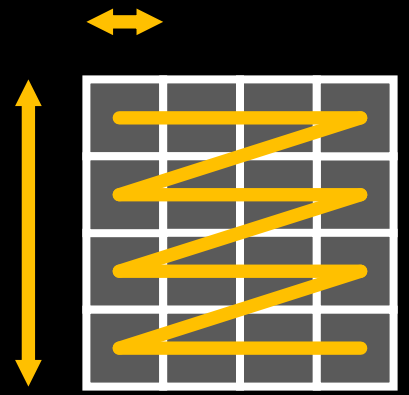
dtype: the data type of all elements of this array.
Ex: `int32`



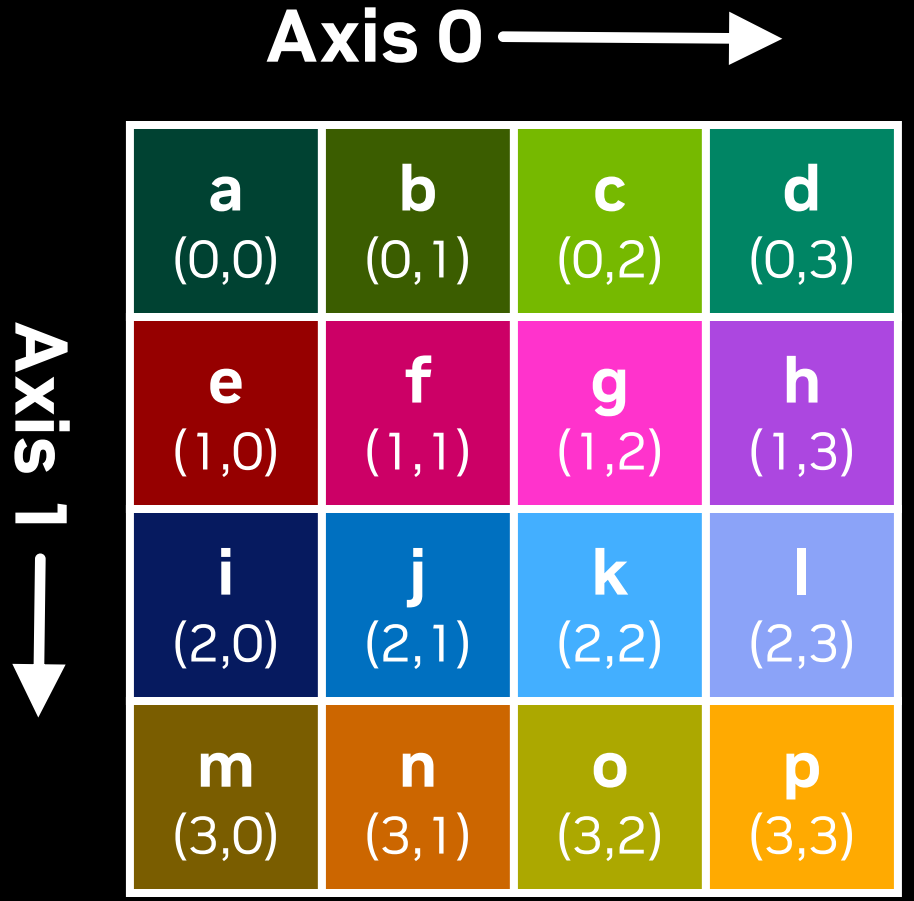
data: a pointer to the block of memory holding all the elements in this array.



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



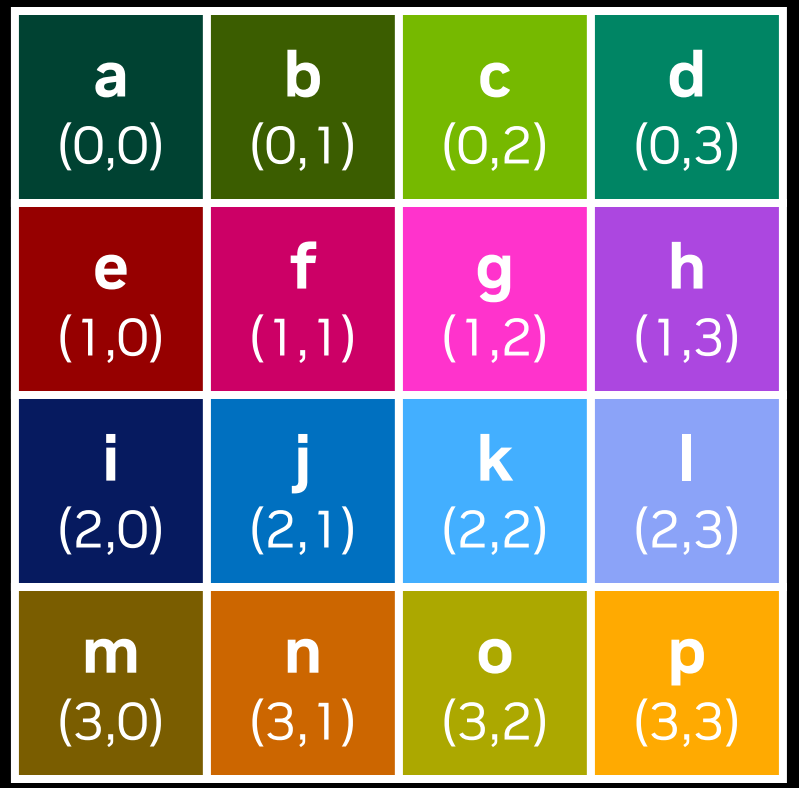
strides: the number of bytes between elements along each dimension.
Ex: `(12, 4)`



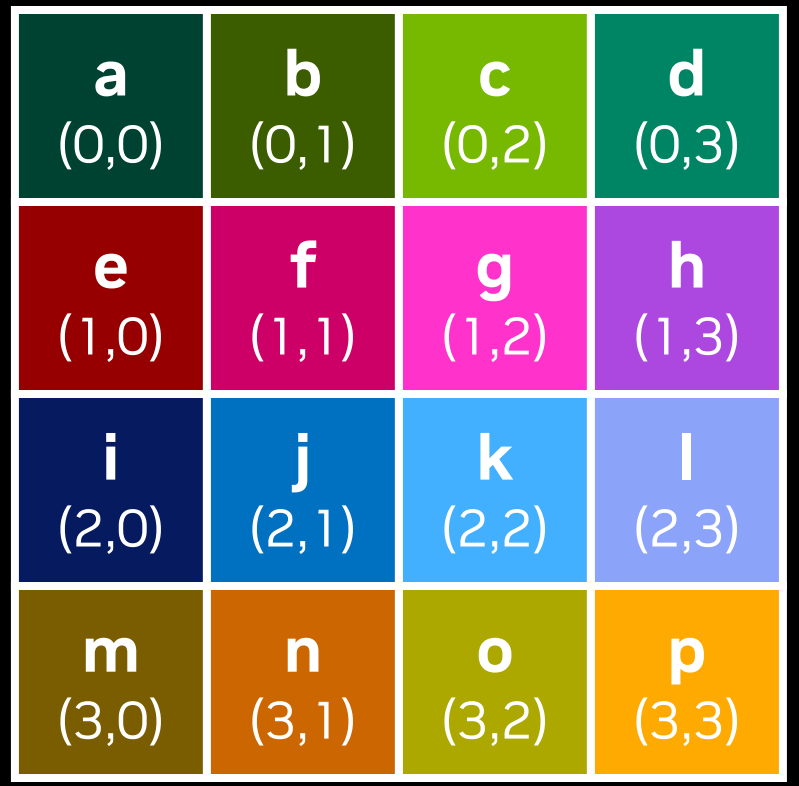
Each axis of the array can have whatever meaning makes sense for your situation:

- 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



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

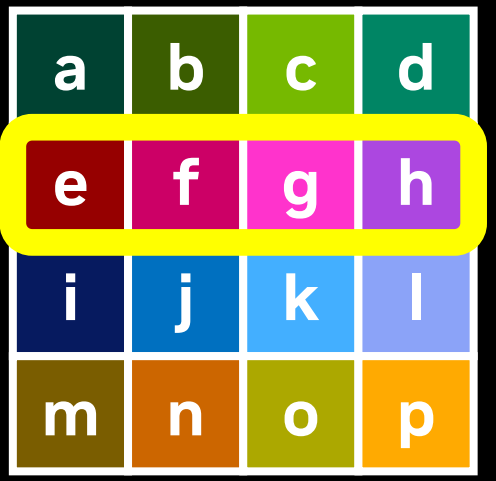
Views of the original data, *not a copy*

```
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)



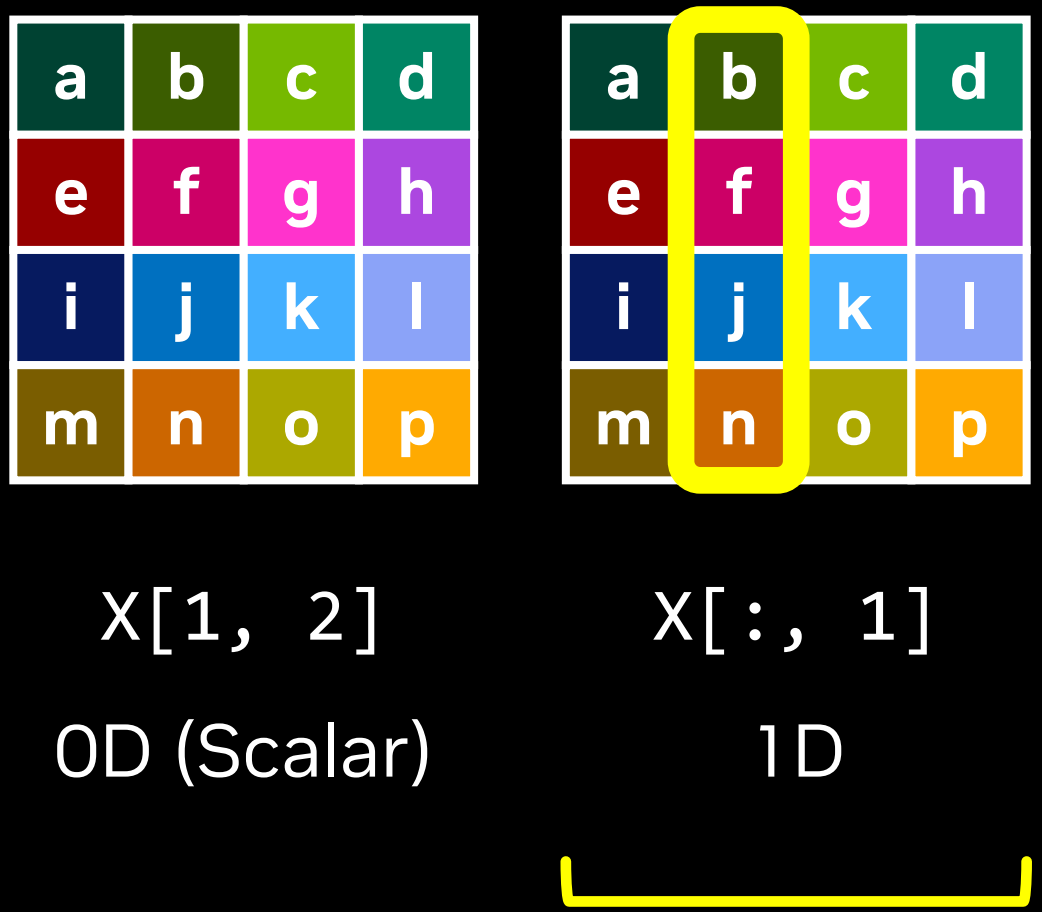
`X[1, 2]`

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)

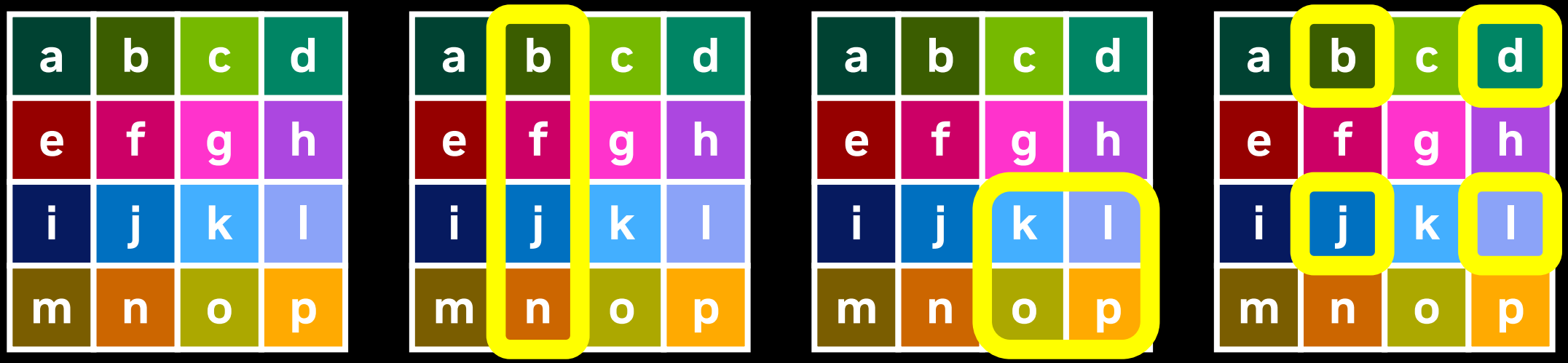
`X[:, 1]`

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

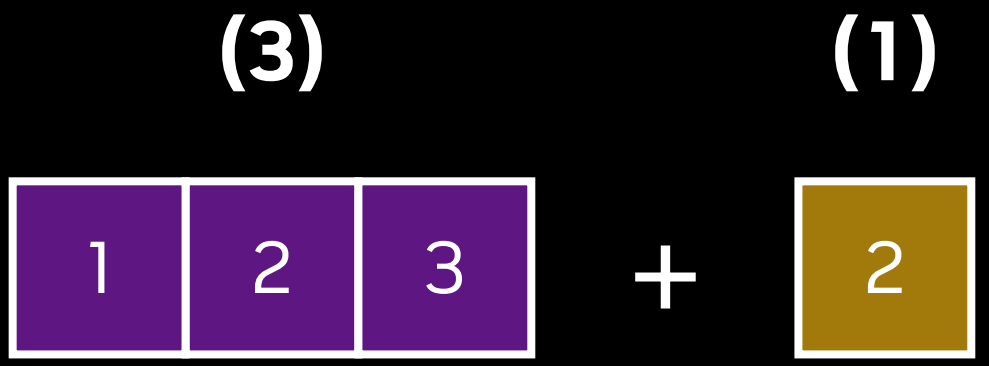
`X[2:, 2:]`

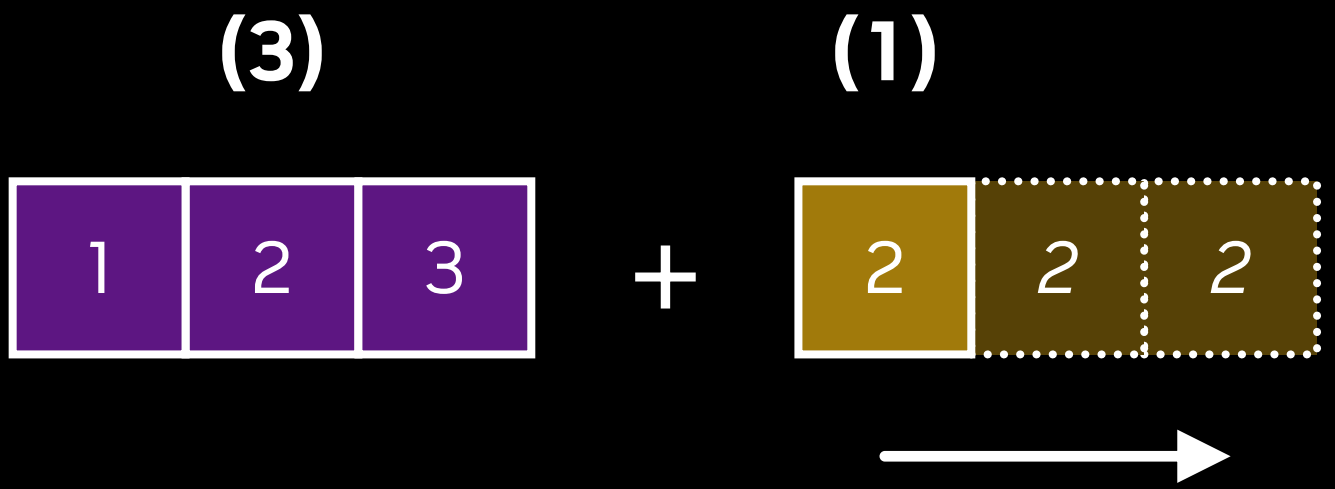
2D

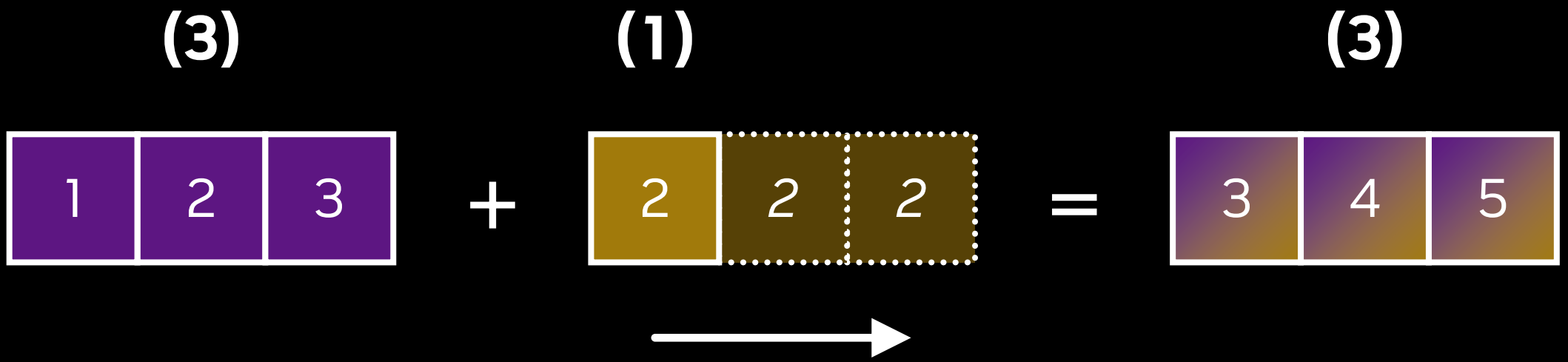
`A[::2, 1::2]`

2D

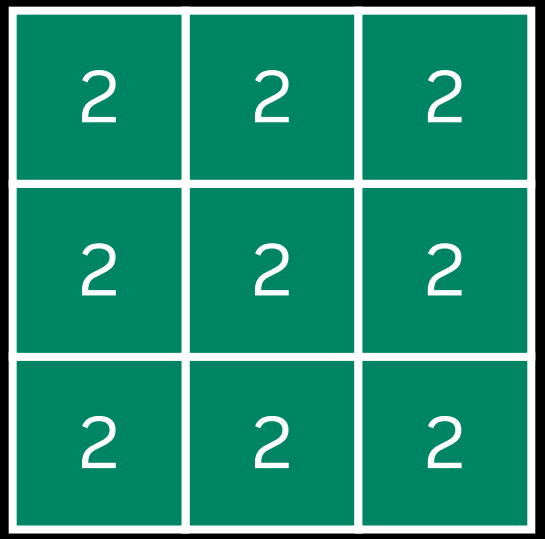
`start:stop:step`





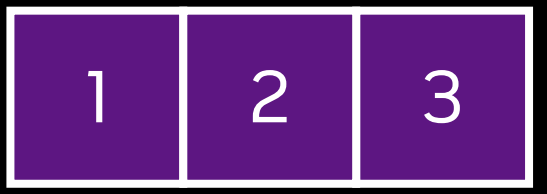


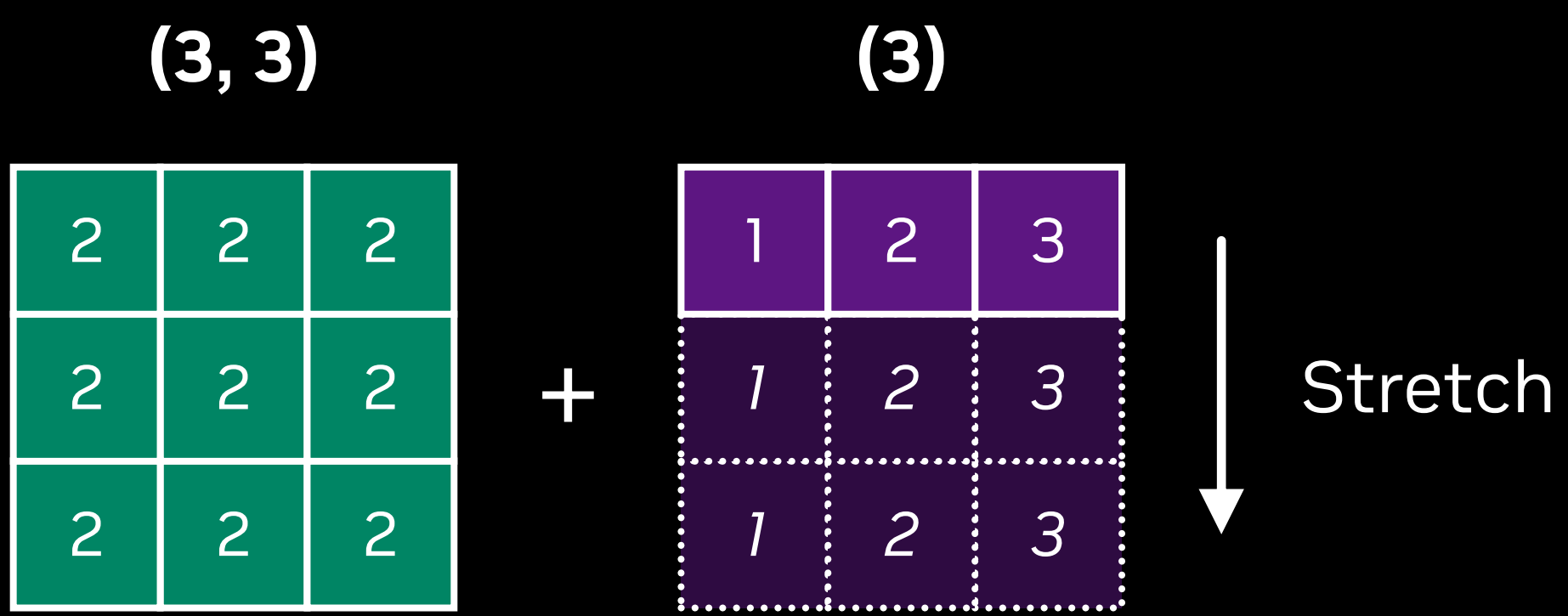
(3, 3)

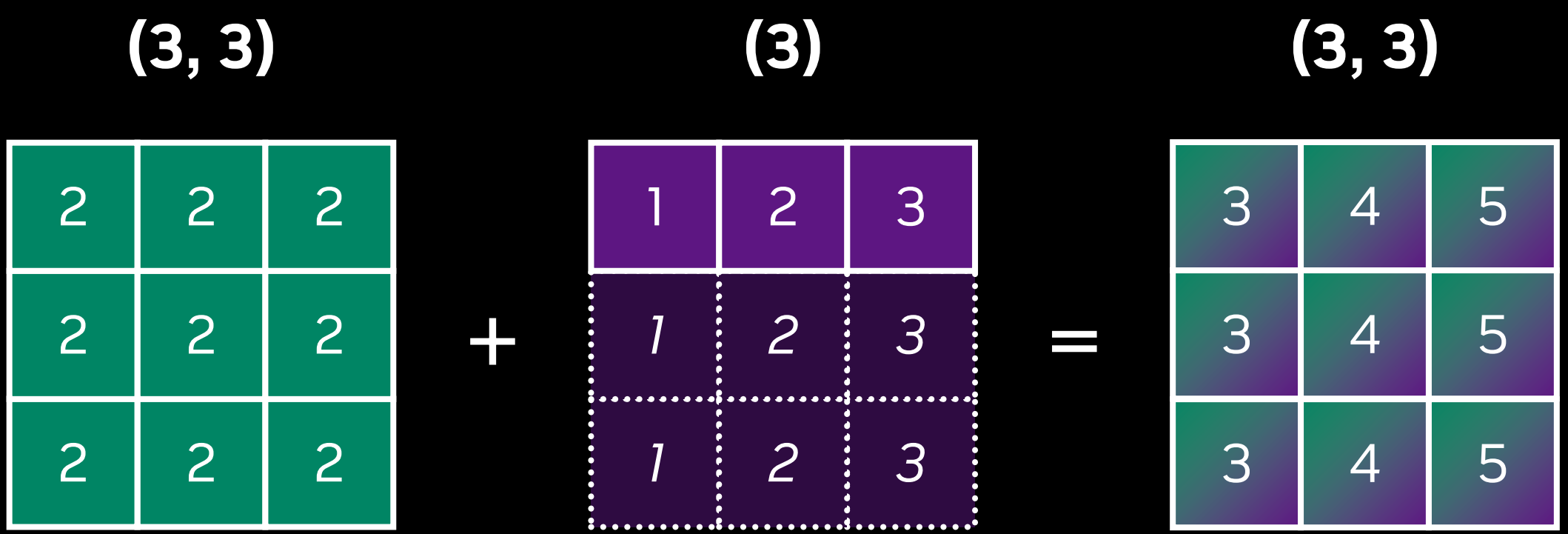


+

(3)







(3, 2)

-1	1
1	-1
-1	1

+

(3)

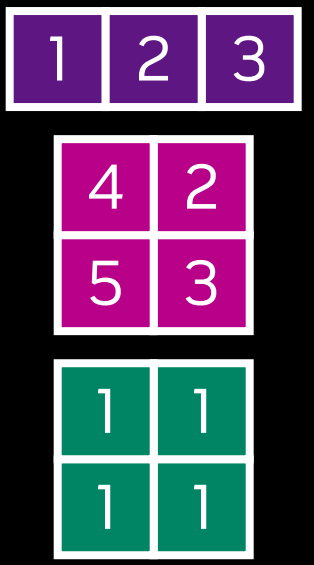
1	2	3

=

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

Array Functions apply operations to the contents of arrays

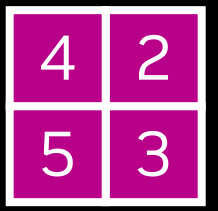
```
A = np.arange(1, 3)

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.

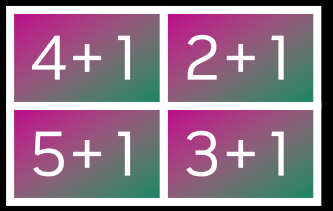
Array Functions apply operations to the contents of arrays

```
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.

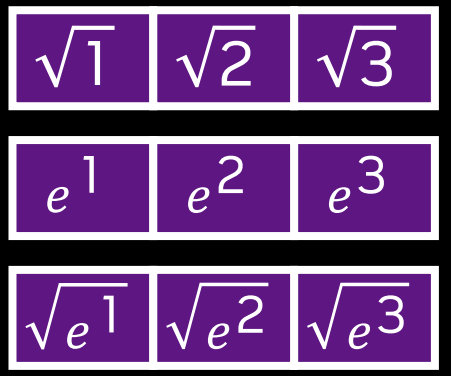
```
D = np.add(B, C)

D = B + C
```



Arithmetic can be done with operators.

Array Functions apply operations to the contents of arrays

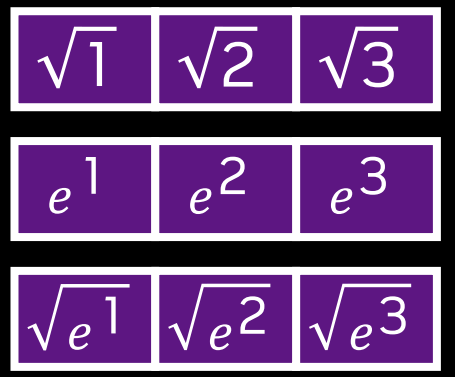
```
F = np.sqrt(A)
G = np.exp(A)

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.

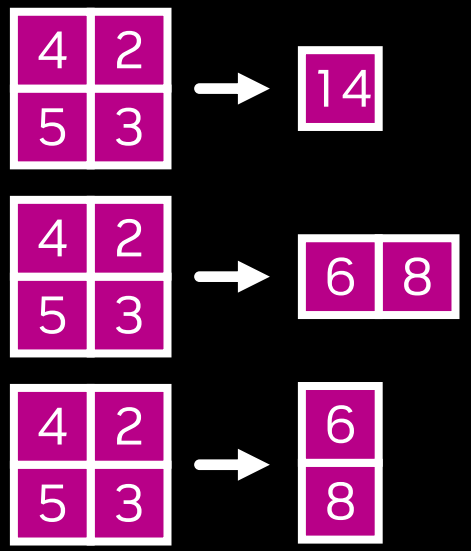
Array Functions apply operations to the contents of arrays

```
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 1

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, …`)

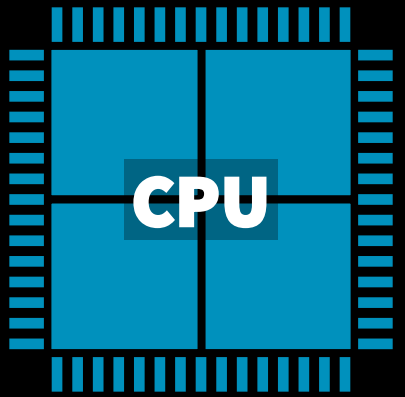
Math / Logic Functions
(`np.exp, np.sin, np.isnan, …`)

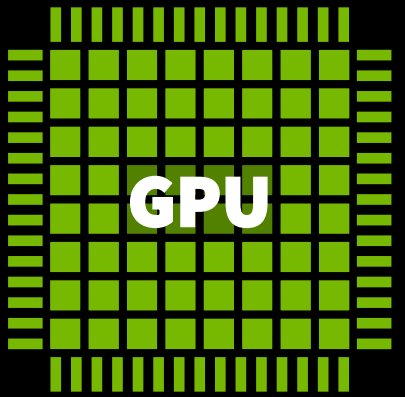
…and many more categories
(and SciPy has *even more*)

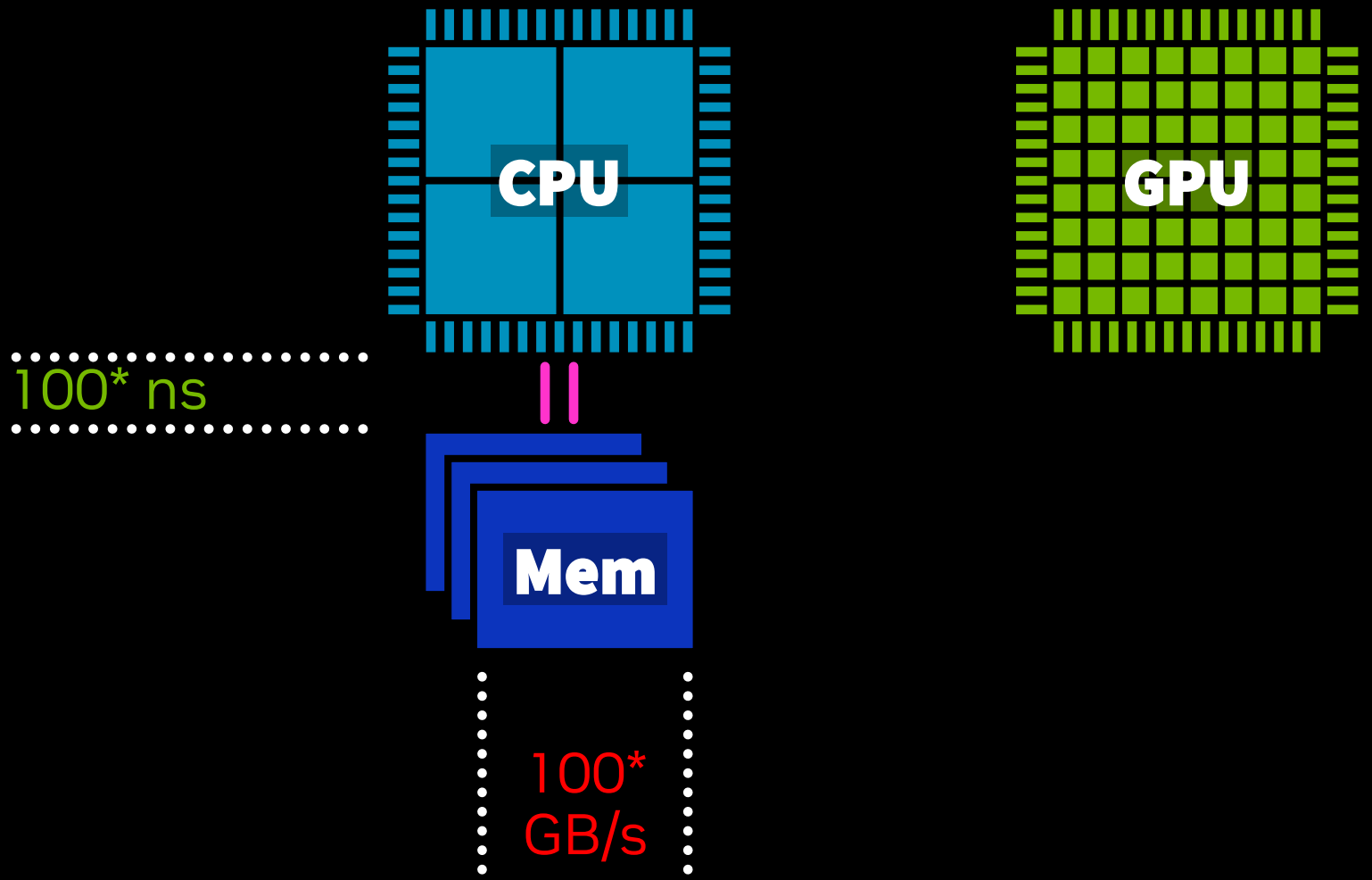
Exercise 2

NumPy Linear Algebra

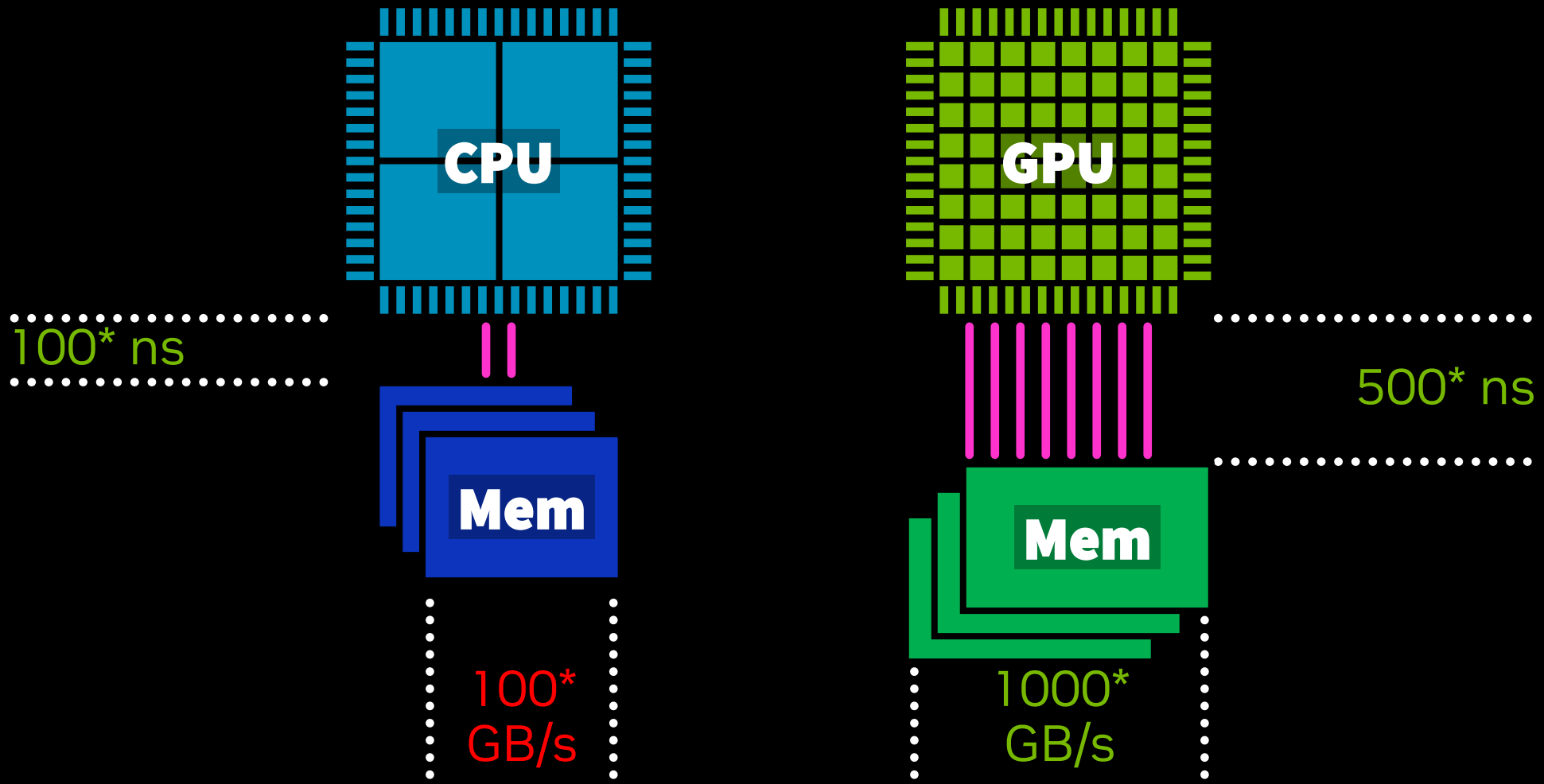
SVD Reconstruction







Numbers are made up and
for expository purposes only.

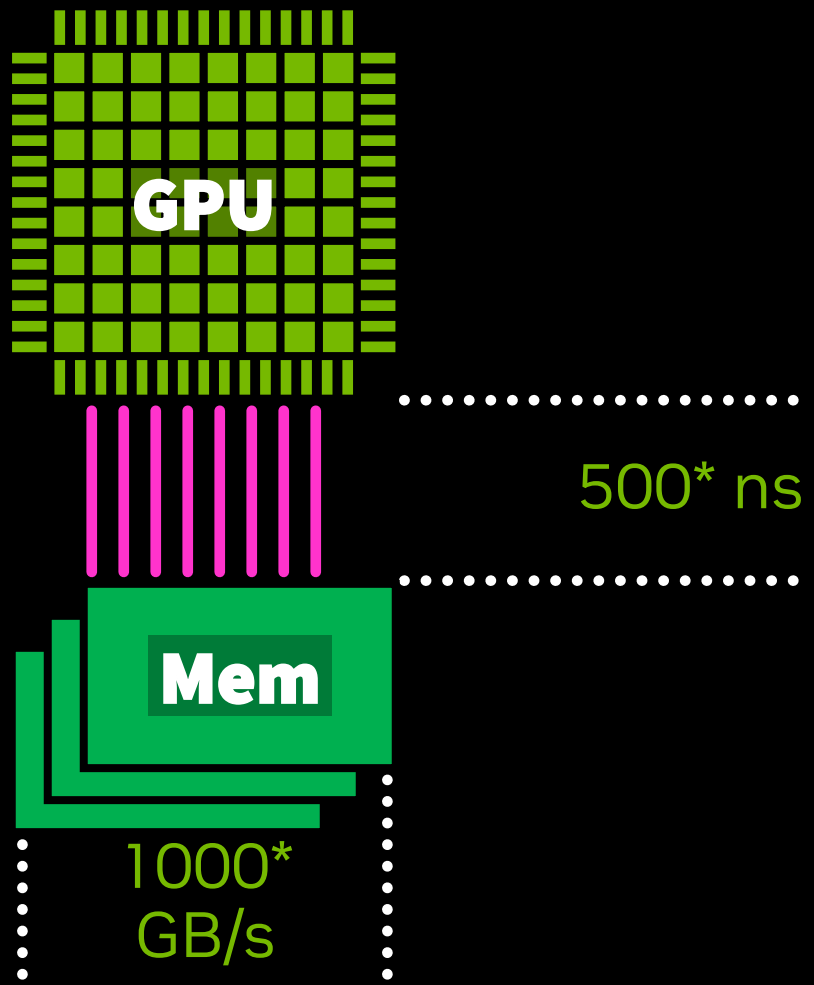


Numbers are made up and
for expository purposes only.

You need a sufficient workload to benefit from GPUs.

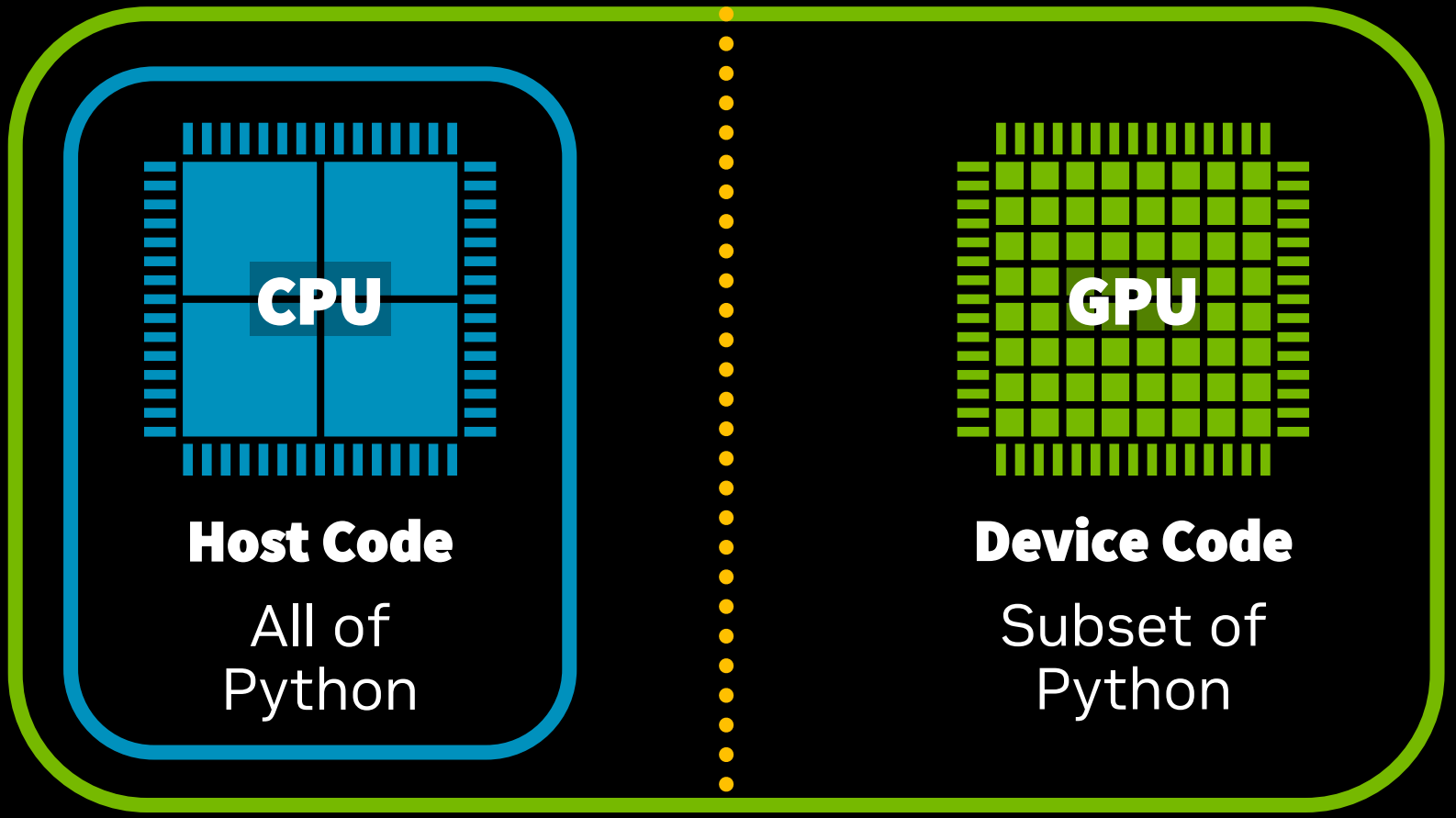
O(N) compute => GBs of data.

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.

```python
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)
```

```python
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 3

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)
```

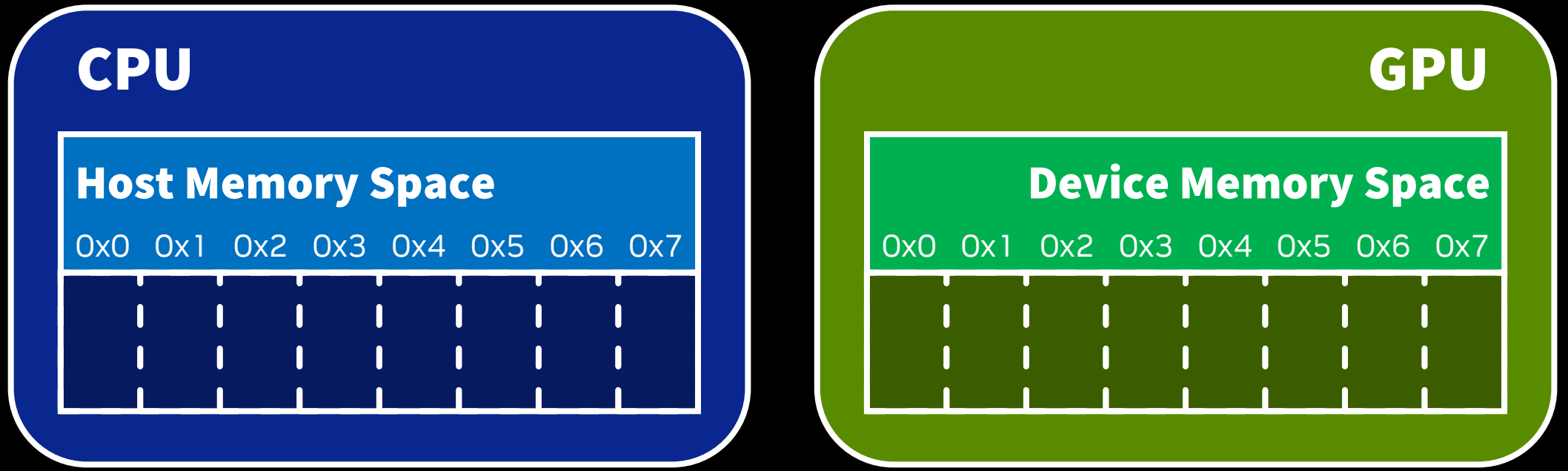
```
a_gpu = cp.zeros(1000, dtype=cp.float32)
```

```
%%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()
```

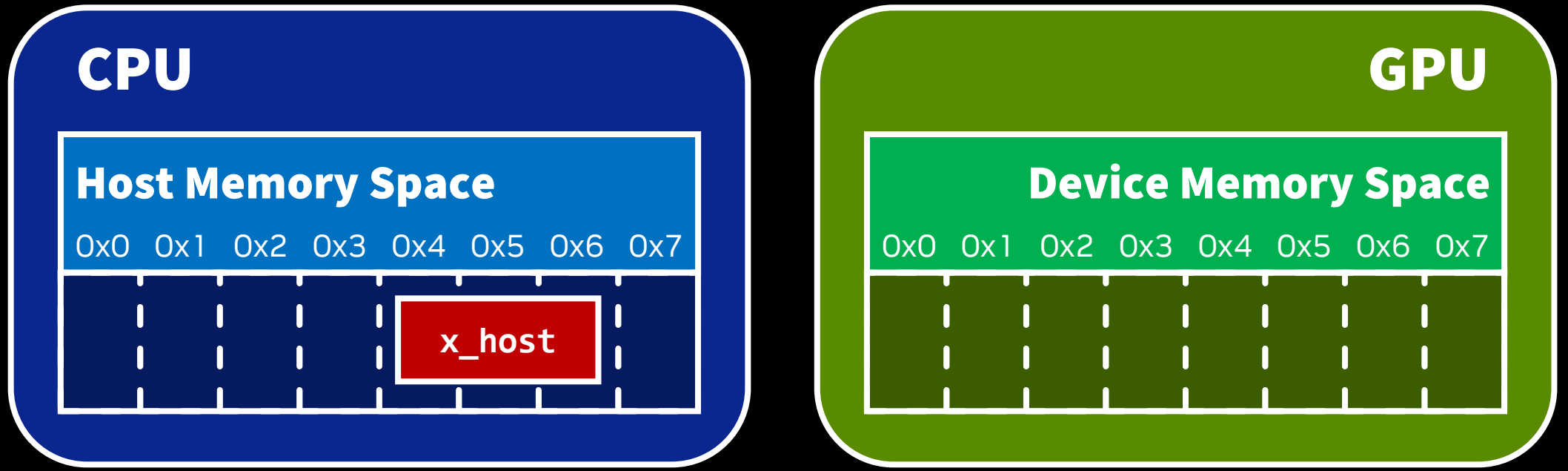
```
8.17 ms ± 30 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

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

Heterogeneous systems have multiple kinds of memory.

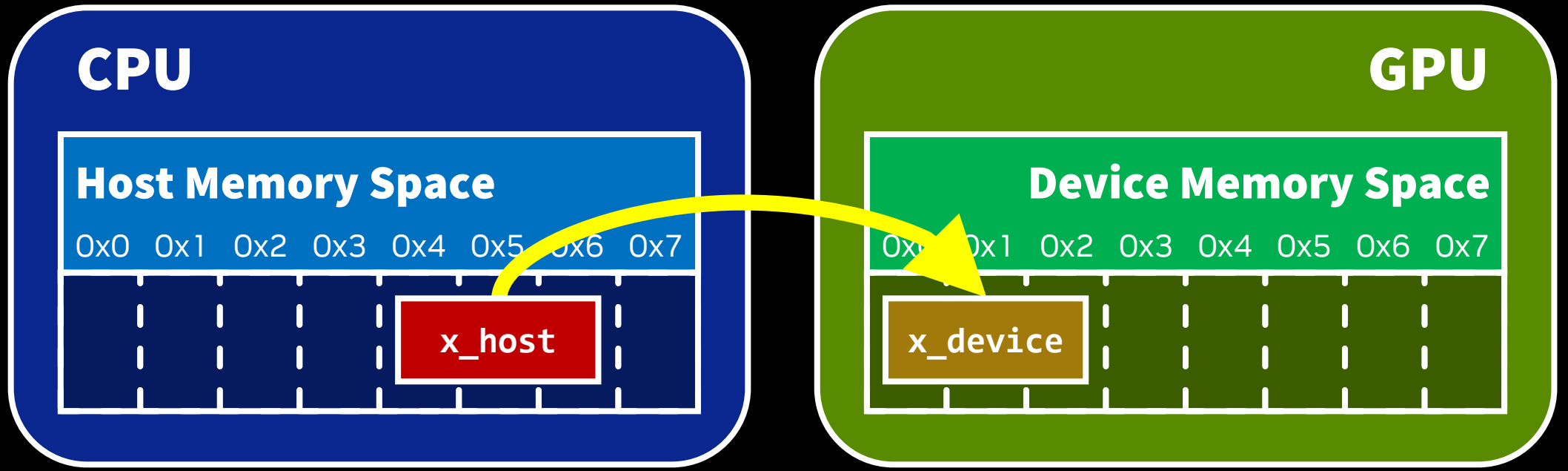


CPUs & GPUs may only have access to a subset of memory.



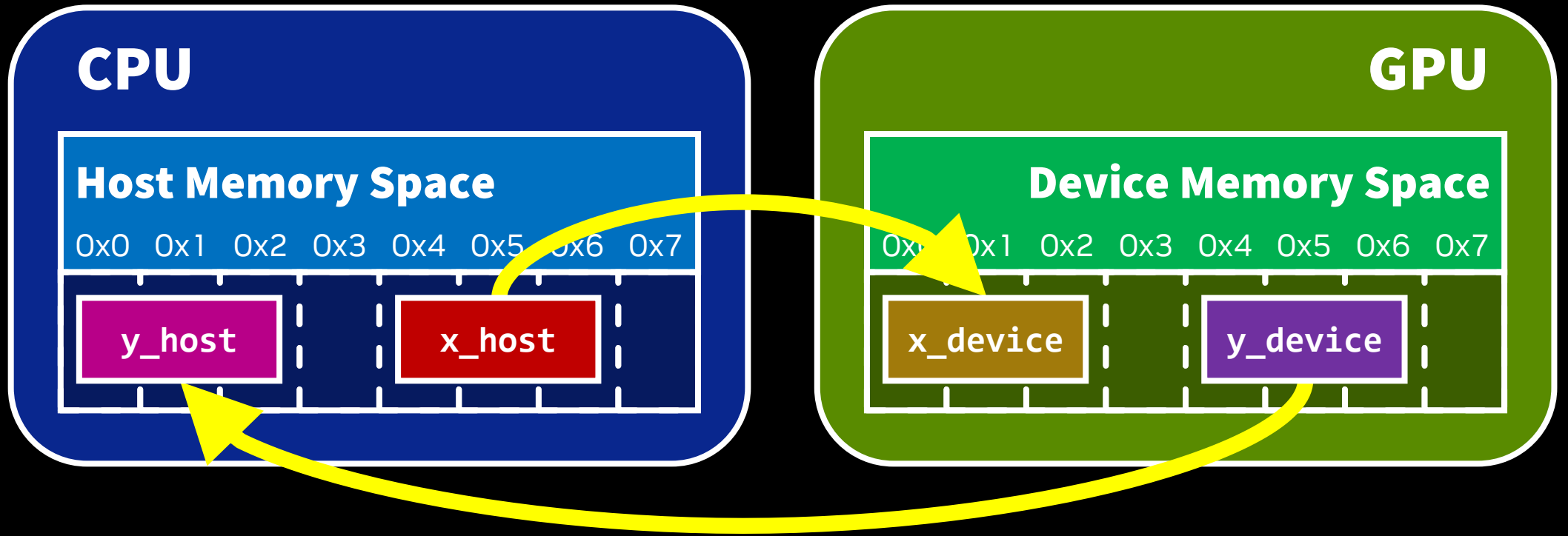
How do we transfer data between host and device?

```
x_device = cp.asarray(x_host)
```



How do we transfer data between host and device?

```
x_device = cp.asarray(x_host)
```




```
y_host = cp.asnumpy(y_device)
```

Exercise 4

NumPy to CuPy

SVD Reconstruction

Beware Implicit Memory Transfers

CuPy silently transfers & ***synchronizes*** when:

Beware Implicit Memory Transfers

CuPy silently transfers & ***synchronizes*** when:

- Printing or representing (`print(x)`, `str(x)`, f-string interpolation).

Beware Implicit Memory Transfers

CuPy silently transfers & ***synchronizes*** when:

- Printing or representing (`print(x)`, `str(x)`, f-string interpolation).
- Converting to Python scalars or lists (`int(x)`, `float(x)`, `bool(x)`, `x.item()`).

Beware Implicit Memory Transfers

CuPy silently transfers & ***synchronizes*** when:

- Printing or representing (`print(x)`, `str(x)`, f-string interpolation).
- Converting to Python scalars or lists (`int(x)`, `float(x)`, `bool(x)`, `x.item()`).
- CuPy function that falls back to NumPy/SciPy because no GPU version exists (`save(x)`, `load(x)`).

Beware Implicit Memory Transfers

CuPy silently transfers & ***synchronizes*** when:

- Printing or representing (`print(x)`, `str(x)`, f-string interpolation).
- Converting to Python scalars or lists (`int(x)`, `float(x)`, `bool(x)`, `x.item()`).
- CuPy function that falls back to NumPy/SciPy because no GPU version exists (`save(x)`, `load(x)`).

```
N2 = np.zeros((2, 2))
C2 = cp.zeros((2, 2))
C0 = cp.sum(C1)
B = 3.14
```

Beware Implicit Memory Transfers

CuPy silently transfers & ***synchronizes*** when:

- Printing or representing (`print(x)`, `str(x)`, f-string interpolation).
- Converting to Python scalars or lists (`int(x)`, `float(x)`, `bool(x)`, `x.item()`).
- CuPy function that falls back to NumPy/SciPy because no GPU version exists (`save(x)`, `load(x)`).

```
N2 = np.zeros((2, 2))
C2 = cp.zeros((2, 2))
C0 = cp.sum(C1)
B = 3.14

# For arrays of rank 1 and greater:
C2 + C2 # CuPy + CuPy doesn't transfer/sync.
C2 + N2 # CuPy + NumPy is an ERROR.
```

Beware Implicit Memory Transfers

CuPy silently transfers & ***synchronizes*** when:

- Printing or representing (`print(x)`, `str(x)`, f-string interpolation).
- Converting to Python scalars or lists (`int(x)`, `float(x)`, `bool(x)`, `x.item()`).
- CuPy function that falls back to NumPy/SciPy because no GPU version exists (`save(x)`, `load(x)`).

```
N2 = np.zeros((2, 2))
C2 = cp.zeros((2, 2))
C0 = cp.sum(C1)
B = 3.14

# For arrays of rank 1 and greater:
C2 + C2 # CuPy + CuPy doesn't transfer/sync.
C2 + N2 # CuPy + NumPy is an ERROR.

# For 0D arrays (e.g. scalars):
C0 + N2 # CuPy + NumPy is an ERROR.
C0 + B  # CuPy + builtins transfers w/o sync.
```

Beware Implicit Memory Transfers

CuPy silently transfers & ***synchronizes*** when:

- Printing or representing (`print(x)`, `str(x)`, f-string interpolation).
- Converting to Python scalars or lists (`int(x)`, `float(x)`, `bool(x)`, `x.item()`).
- CuPy function that falls back to NumPy/SciPy because no GPU version exists (`save(x)`, `load(x)`).

```
N2 = np.zeros((2, 2))
C2 = cp.zeros((2, 2))
C0 = cp.sum(C1)
B = 3.14

# For arrays of rank 1 and greater:
C2 + C2 # CuPy + CuPy doesn't transfer/sync.
C2 + N2 # CuPy + NumPy is an ERROR.

# For 0D arrays (e.g. scalars):
C0 + N2 # CuPy + NumPy is an ERROR.
C0 + B  # CuPy + builtins transfers w/o sync.

print(f"{C2}") # Printing will transfer & sync.
if C0: pass    # Conditions will transfer & sync.
float(C0)      # Conversion will transfer & sync.
```

Beware Implicit Memory Transfers

CuPy silently transfers & ***synchronizes*** when:

- Printing or representing (`print(x)`, `str(x)`, f-string interpolation).
- Converting to Python scalars or lists (`int(x)`, `float(x)`, `bool(x)`, `x.item()`).
- CuPy function that falls back to NumPy/SciPy because no GPU version exists (`save(x)`, `load(x)`).

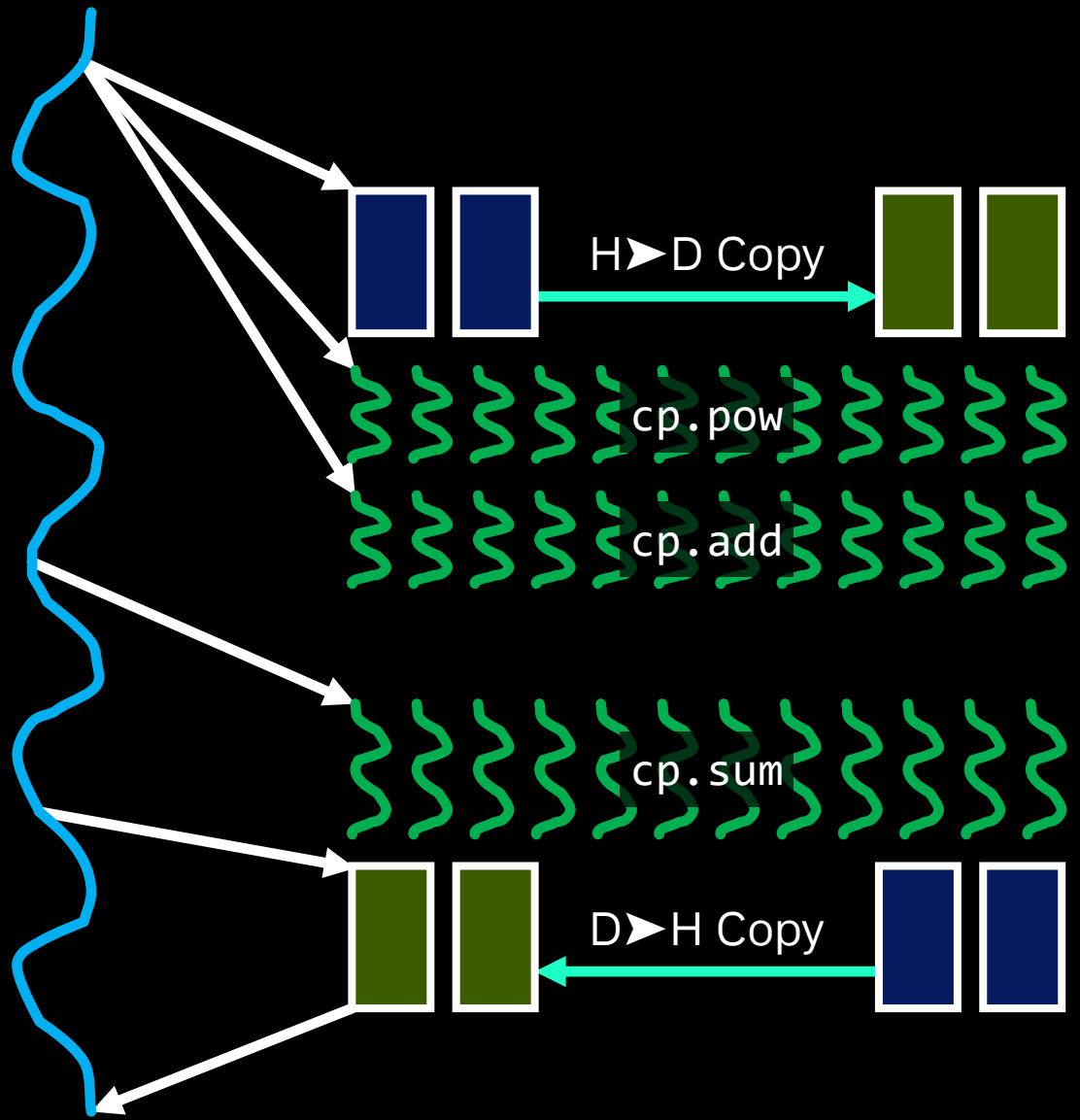
```
N2 = np.zeros((2, 2))
C2 = cp.zeros((2, 2))
C0 = cp.sum(C1)
B = 3.14

# For arrays of rank 1 and greater:
C2 + C2 # CuPy + CuPy doesn't transfer/sync.
C2 + N2 # CuPy + NumPy is an ERROR.

# For 0D arrays (e.g. scalars):
C0 + N2 # CuPy + NumPy is an ERROR.
C0 + B  # CuPy + builtins transfers w/o sync.

print(f"{C2}") # Printing will transfer & sync.
if C0: pass    # Conditions will transfer & sync.
float(C0)      # Conversion will transfer & sync.

C2[0, 0] # Indexing doesn't transfer.
C2[0, :] # Slicing doesn't transfer.
C2[0, 0] = 7 # Scalar assigns transfer w/o sync.
```

Exercise 5

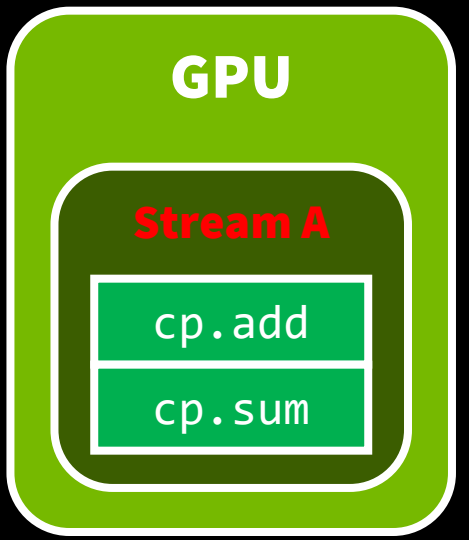
Memory Spaces

Power Iteration

```
A = cp.asarray(…)

A = A + A**2

do_cpu_stuff()

x = A.sum(axis=1)

do_cpu_other_stuff()

x_host = cp.asnumpy(x)
```

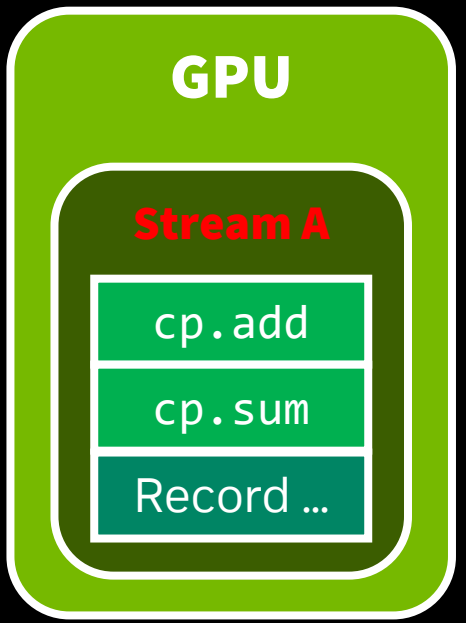


```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  cp.sum(…)
```



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

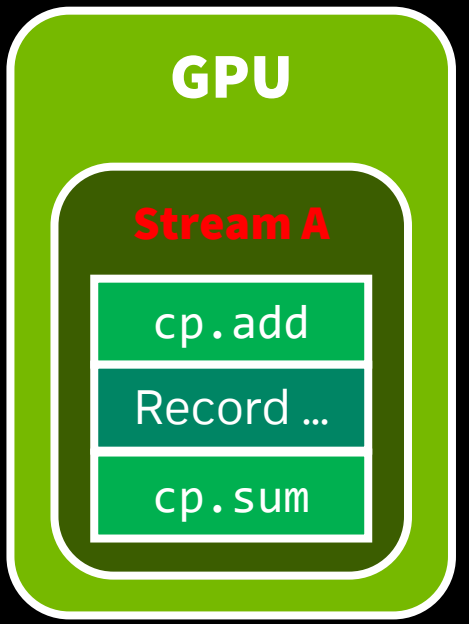
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  cp.sum(…)
  A.synchronize() # Host waits.
```



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

They can record events that host code can sync with.

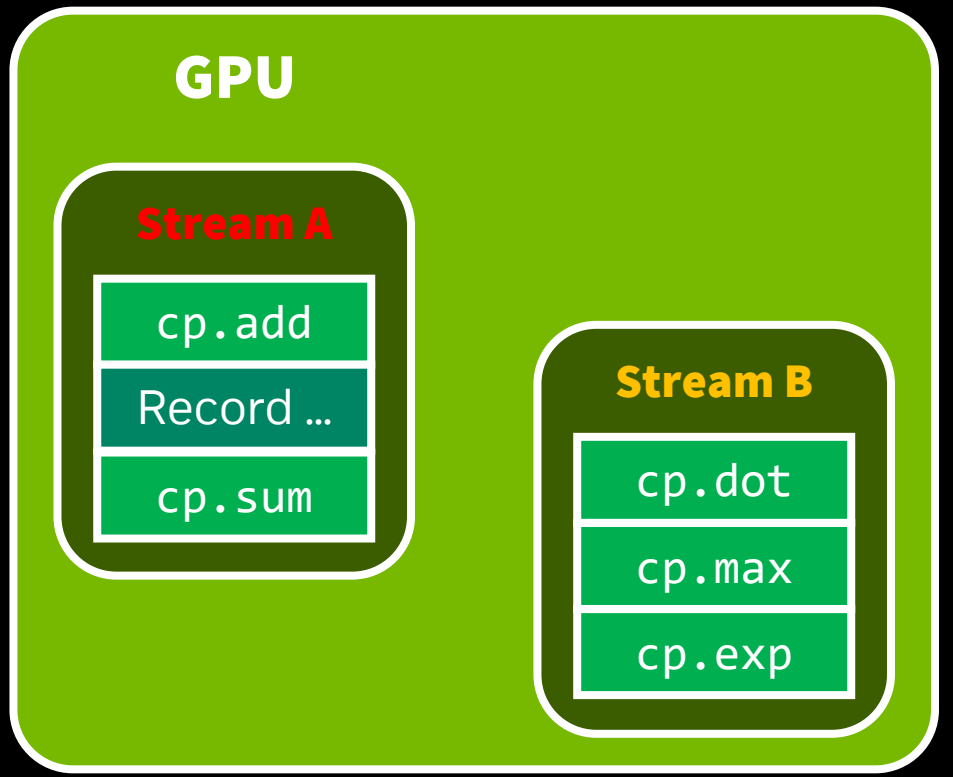
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)
```



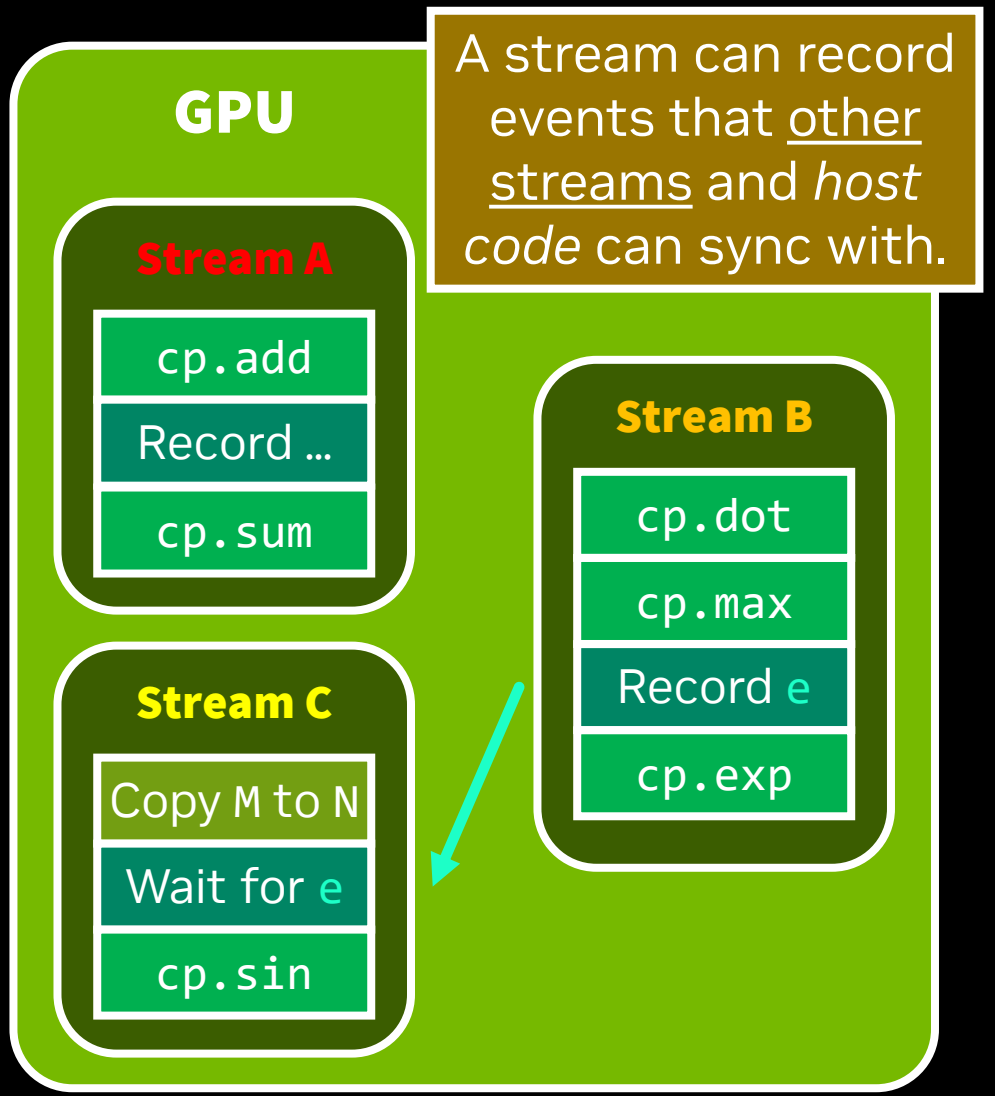
A stream is a chain of device work that is executed in order.

They can record events that host code can sync with.

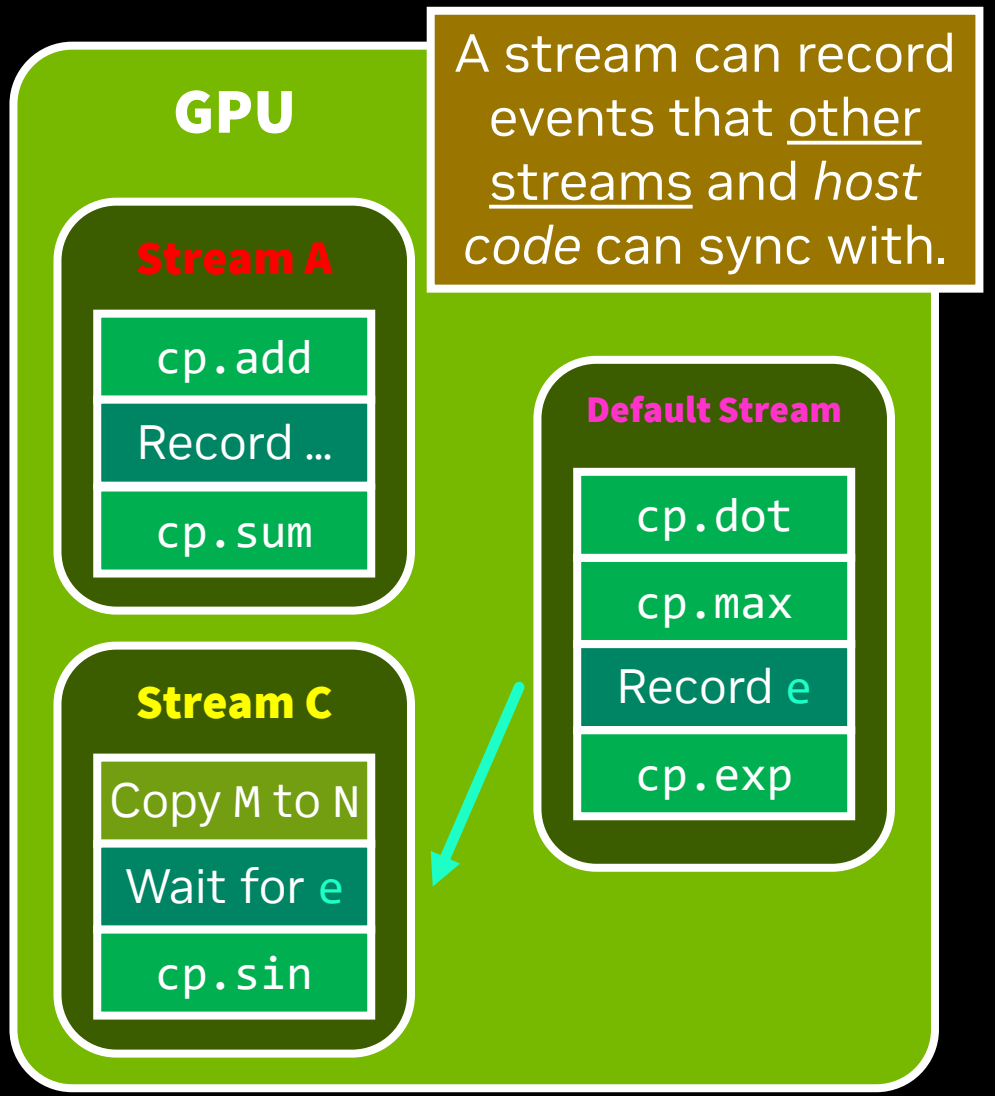
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)

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



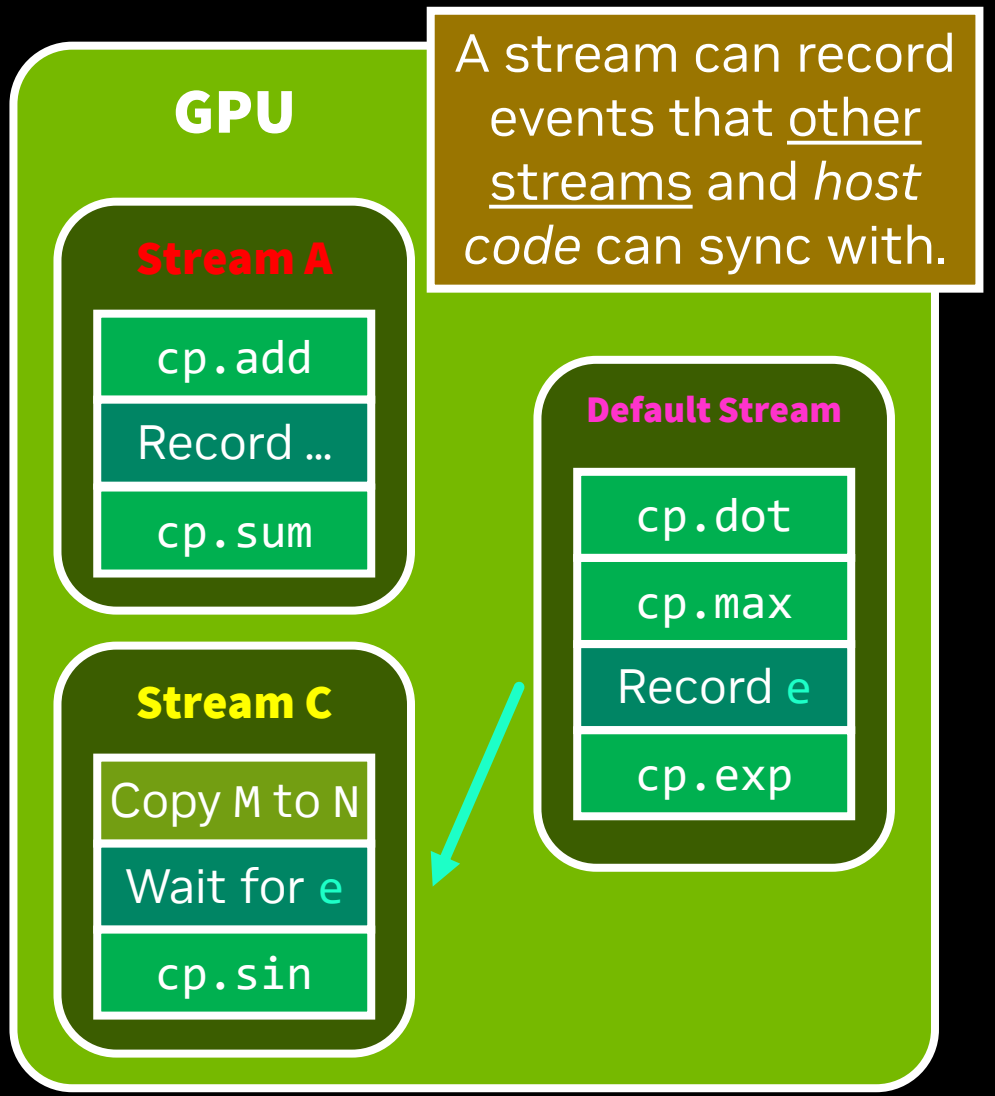
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)

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.
```



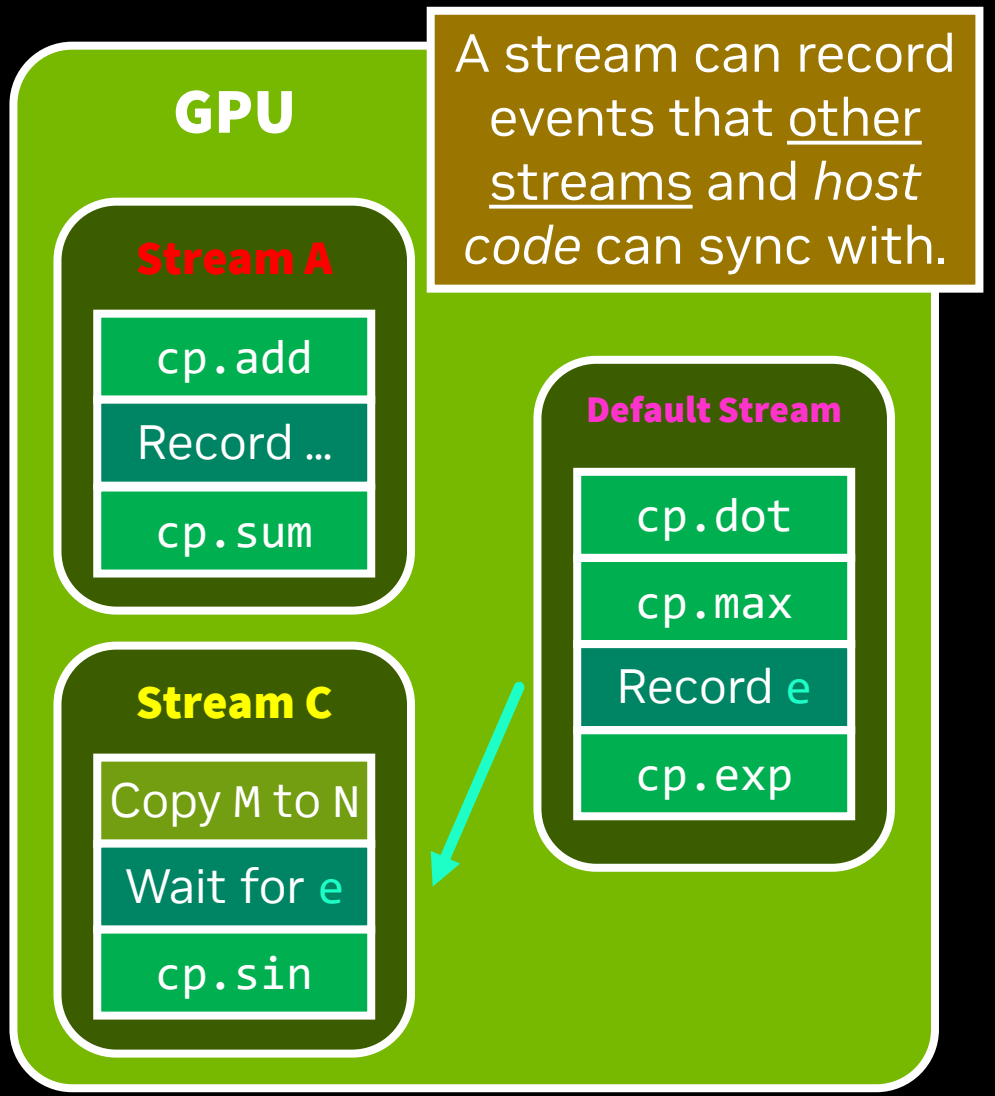
```
A = cp.cuda.Stream(non_blocking=True)
with A:
  cp.add(…)
  A.synchronize() # Host waits.
  cp.sum(…)

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 6

Asynchrony

Power Iteration

Pandas for Data Analysis

- Array and tensor frameworks like NumPy, CuPy and PyTorch are great when you have **n-dimensional, numeric** data
- Real data is messy!
  - Lives in CSV, Excel and JSON files
  - Strings, datetimes, timezones, missing values
- Pandas: tabular data analysis tools for Python

	id	name	score
1	P01	Mario	5
2	P02	Luigi	3
3	P03	Peach	7

Pandas DataFrames can hold more than just numeric data

cuDF: GPU-Accelerated Data Analysis

- A GPU-accelerated data analysis library
- Uses GPU algorithms for all operations: filtering, sorting, aggregating and combining data. I/O is accelerated.
- Supports strings, datetimes, timezones, and list-like data
- Offers both Pandas and Polars API
- Built on libcudf C++ library

RAPIDS

https://rapids.ai/

```python
import pandas as pd

tips = pd.read_csv("tips.csv")
tips["tip_percentage"] = (
   tips ["tip"] / tips ["total_bill"] * 100
)

# average tip by dining party size
print(tips.groupby("size").tip_percentage.mean())
```

```python
import cudf

tips = cudf.read_csv("tips.csv")
tips["tip_percentage"] = (
   tips ["tip"] / tips["total_bill"] * 100
)

# average tip by dining party size
print(tips.groupby("size").tip_percentage.mean())
```

Exercise 7

cudf

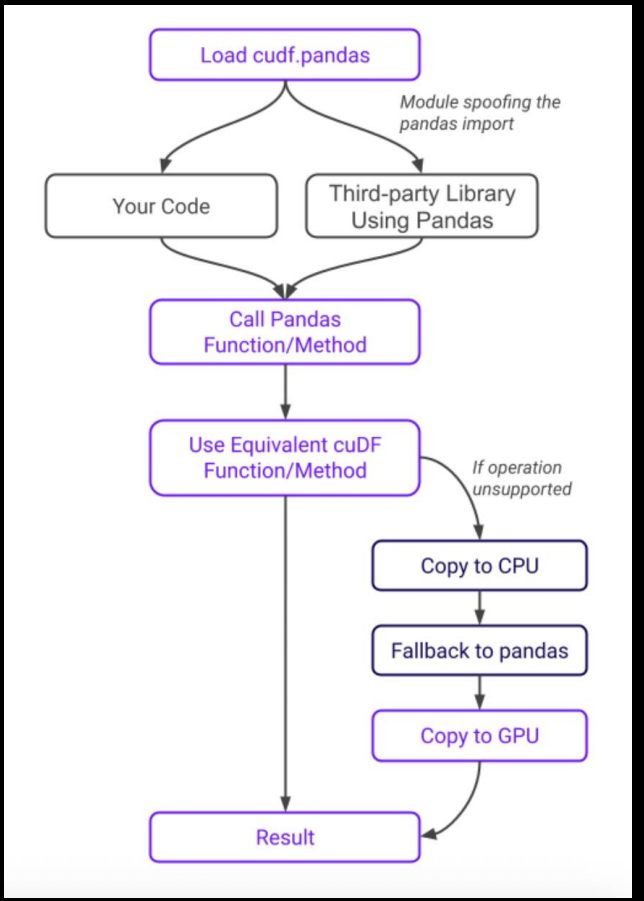
NYC Parking Violations

cudf.pandas

cudf.pandas lets you use pandas, but accelerates parts of the program that can use cudf

Automatically manages data movement between the CPU and GPU behind the scenes

```
%load_ext cudf.pandas
import pandas as pd

# uses cuDF:
data = pd.read_csv("filename.csv")
result = data.groupby("a").mean()

# uses pandas:
result.to_excel("filename.xlsx")
```



Exercise 8

cudf.pandas

NYC Parking Violations

Parallel Algorithms

Parallel Algorithms

Serial algorithms are relatively easy to reason about and implement

Serial Sum:

```
sum = 0
for num in seq:
    sum = sum + num
print(sum)
```

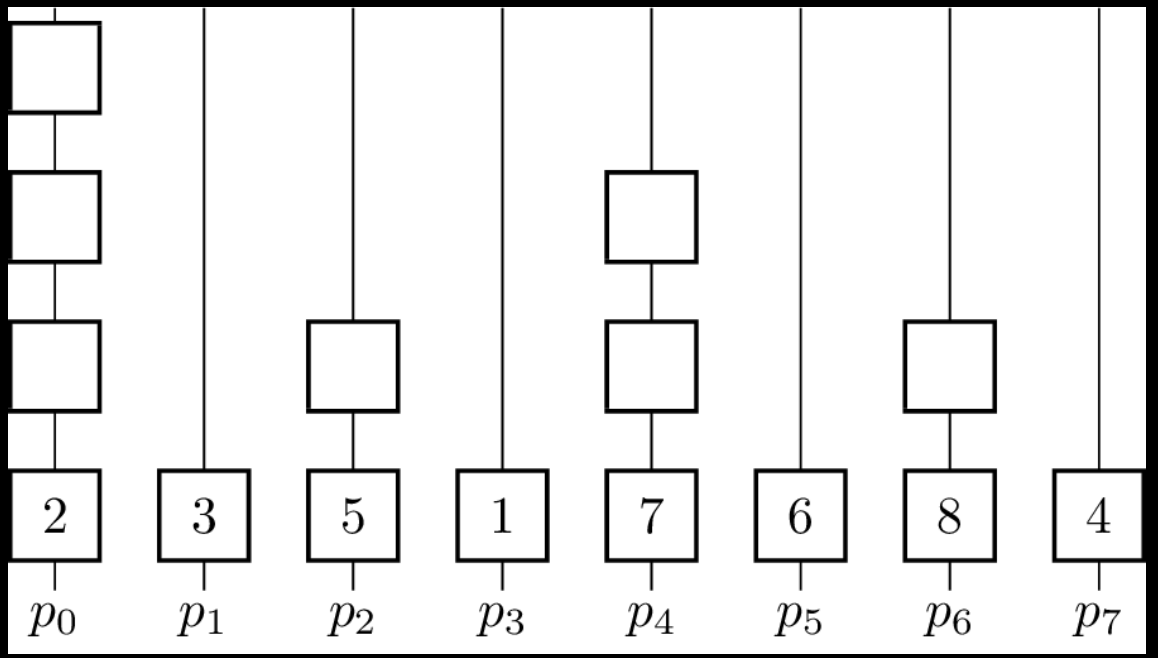
Parallel Algorithms

Serial algorithms are relatively easy to reason about and implement

Serial Sum:

```
sum = 0
for num in seq:
    sum = sum + num
print(sum)
```

Parallel algorithms are usually more Involved



Parallel reduction

Parallel Algorithms

Serial algorithms are relatively easy to reason about and implement

Serial Sum:

```
sum = 0
for num in seq:
    sum = sum + num
print(sum)
```

Parallel algorithms are usually more Involved

Implementation must be optimized for your specific GPU architecture

NVIDIA Hopper microarchitecture

Parallel Algorithms

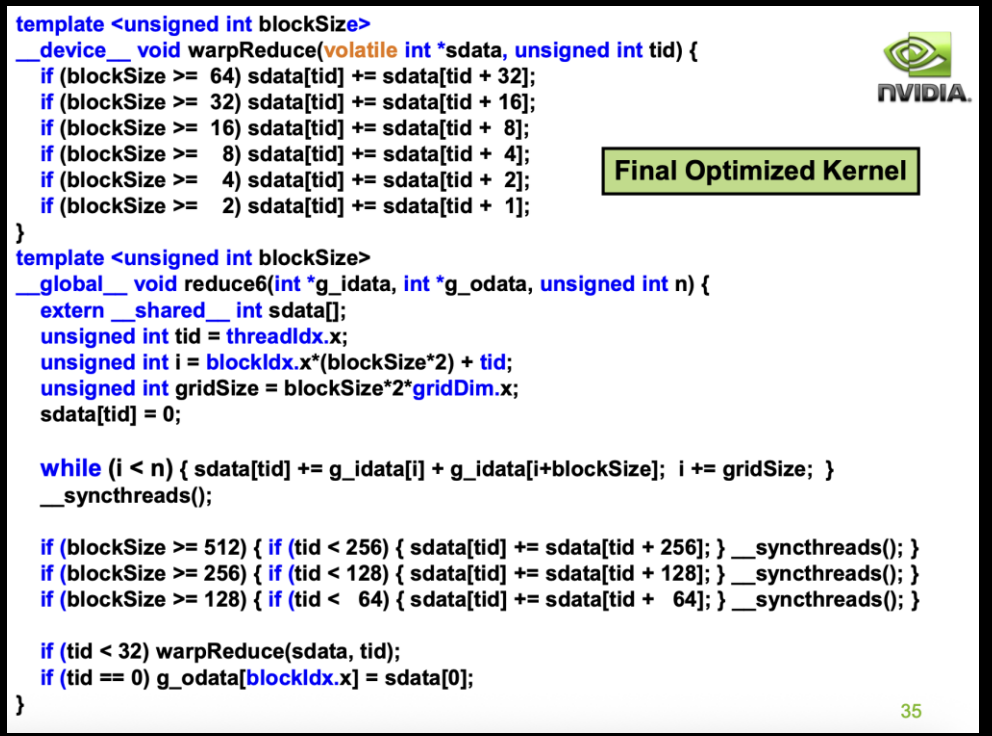
Serial algorithms are relatively easy to reason about and implement

Serial Sum:

```
sum = 0
for num in seq:
    sum = sum + num
print(sum)
```

Parallel algorithms are usually more Involved

Implementation must be optimized for your specific GPU architecture



Optimized CUDA reduction

Parallel Algorithms

"Umm… why don't you just use CuPy/PyTorch to compute `.sum()`?"

```
>>> seq = cp.array([2, 3, 5, 1, 7, 6, 8, 4]
>>> print(seq.sum())
36
```

Parallel Algorithms

"Umm… why don't you just use CuPy/PyTorch to compute `.sum()`?"

```
>>> seq = cp.array([2, 3, 5, 1, 7, 6, 8, 4]
>>> print(seq.sum())
36
```

Some algorithms can't be efficiently written using existing CuPy or PyTorch operations. For example, how would you sum just the even values of a sequence?

Parallel Algorithms

"Umm… why don't you just use CuPy/PyTorch to compute `.sum()`?"

```
>>> seq = cp.array([2, 3, 5, 1, 7, 6, 8, 4]
>>> print(seq.sum())
36
```

Some algorithms can't be efficiently written using existing CuPy or PyTorch operations. For example, how would you sum just the even values of a sequence?

Options

- Suboptimal implementation composed of array/tensor operations
- Write CUDA C++ and bind to it from Python
- Write a custom Numba CUDA/Triton kernel (next section!)
- Use `cuda.cccl` to build your algorithm (this section!)

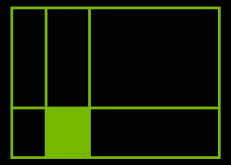
cuda.cccl

- In C++, the **CUDA Core Compute Libraries** provide core abstractions for programming the GPU
  - **Thrust** is a parallel algorithms library containing efficient building blocks like `reduce`, `sort` and `scan` for composing more complex algorithms
  - **CUB** is a cooperative algorithms library containing block-wide and warp-wide primitives like `block_reduce` and `warp_scan` for writing efficient CUDA C++ kernels
  - Venerable and mature – used by many CUDA projects including PyTorch and CuPy
- **`cuda.cccl`** provides the same abstractions for Python developers
  - `cuda.cccl.parallel` provides composable pieces to build up complex algorithms
  - `cuda.cccl.cooperative` provides block-wide and warp-wide primitives for writing efficient CUDA kernels in Python
  - Newer and experimental (API subject to change)

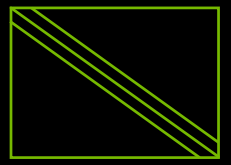
Exercise 9

cuda.cccl

Customizing Algorithms

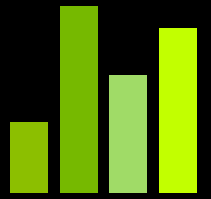


cuBLAS

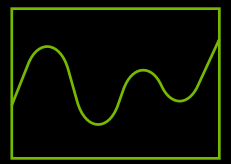


cuSPARSE

cuFile (GDS)



NVBench



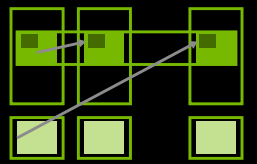
cuFFT

cuDSS

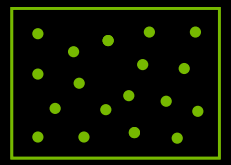
CV CUDA

cuLitho

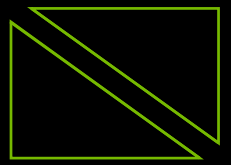
Cooperative Groups



NVSHMEM

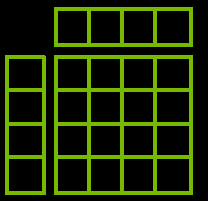


cuRAND

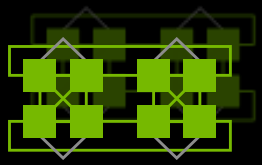


cuSOLVER

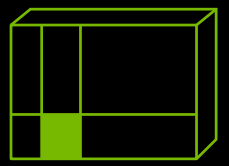
CUDA-Q



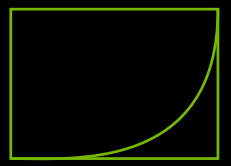
CUTLASS



NCCL



cuTENSOR



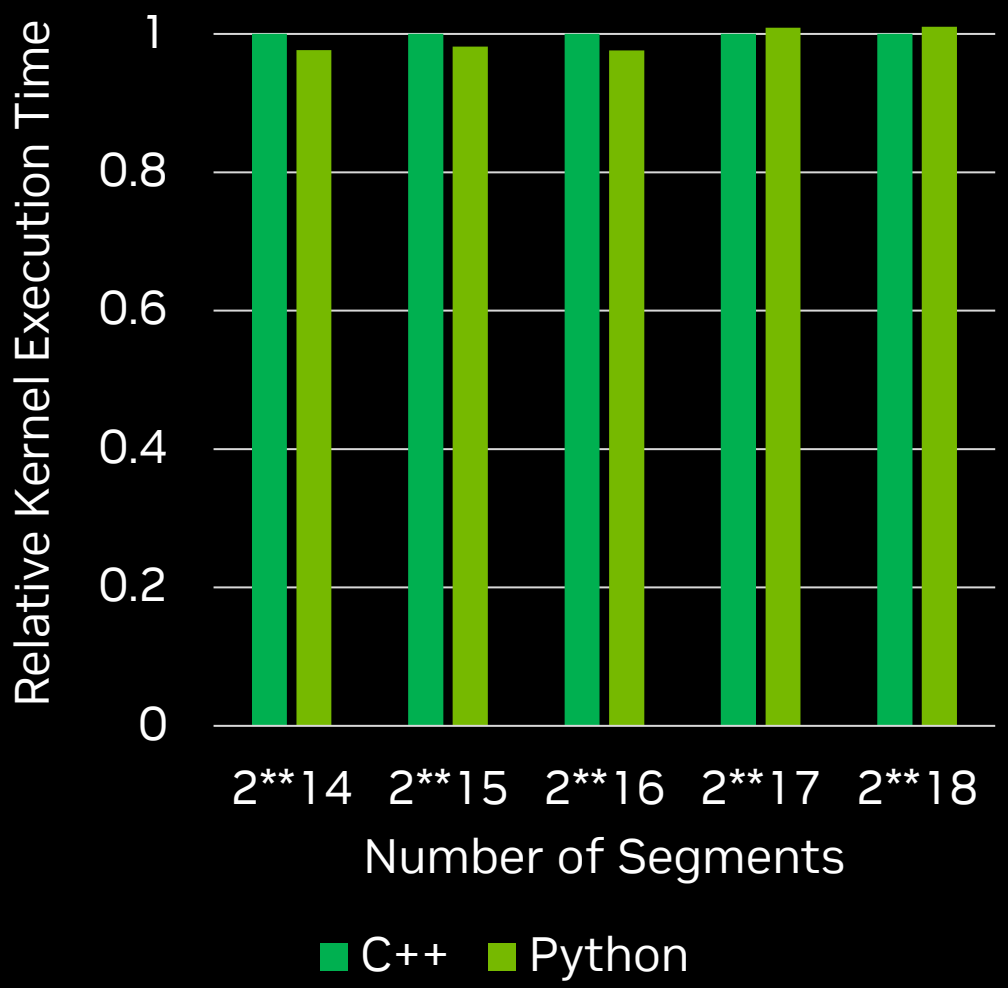
CUDA Math

CUDA Runtime

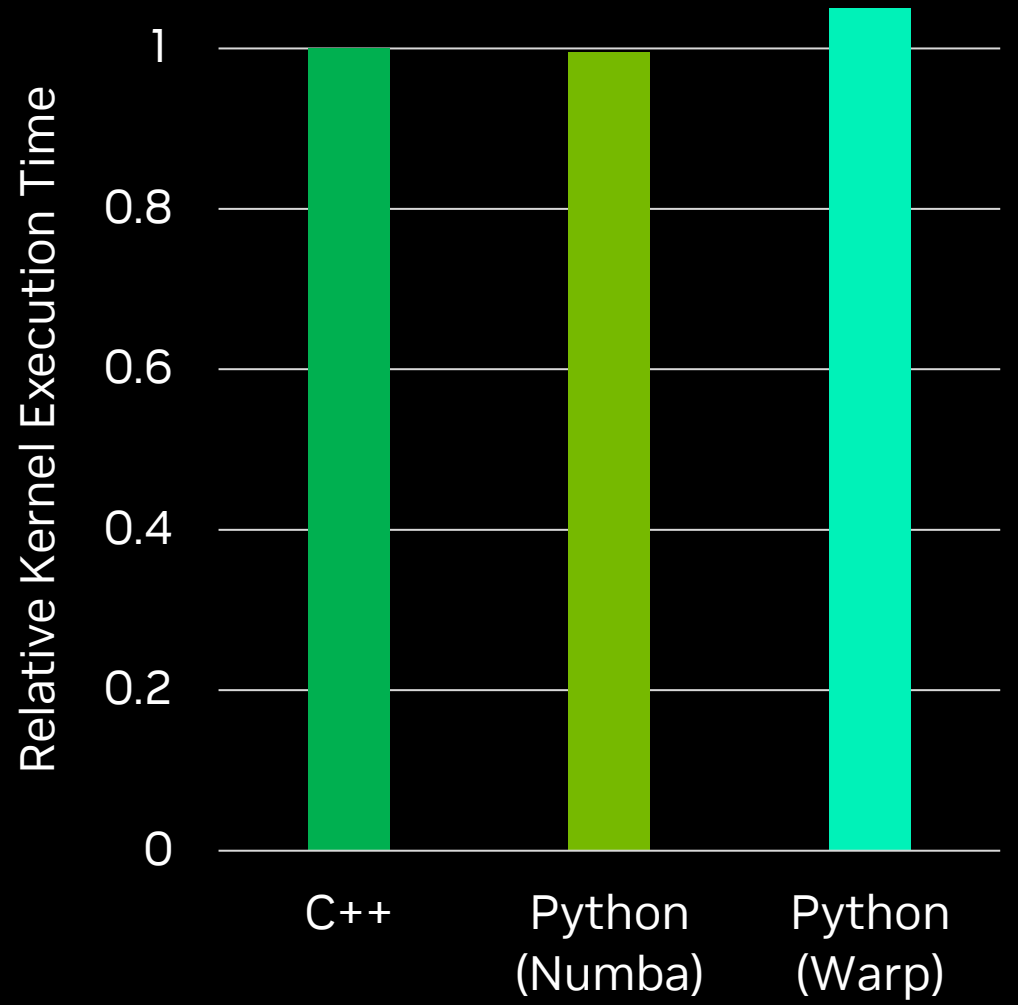
Prefer using libraries!

If no library exists or you can't get the perf you need with them, then...

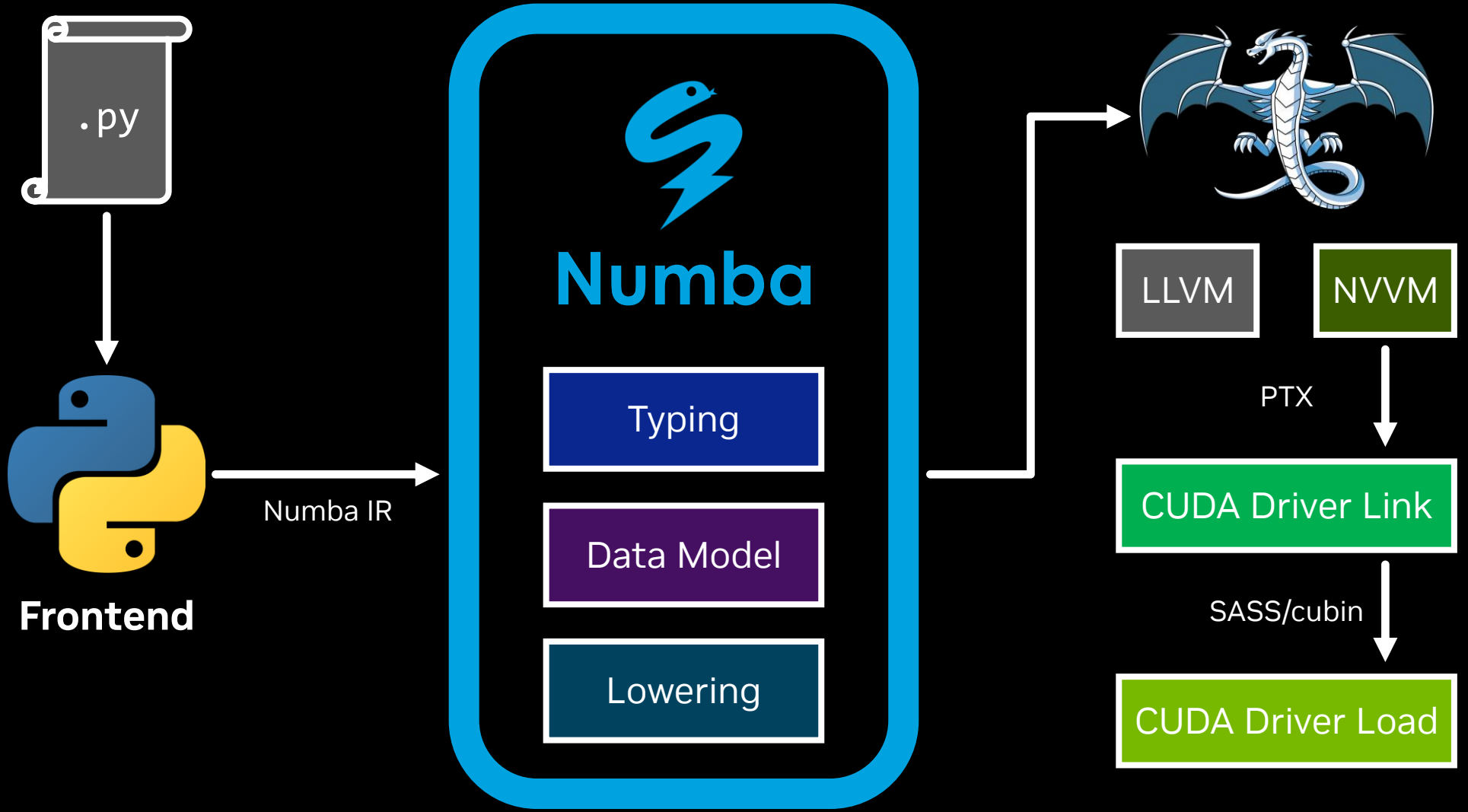
CUDA Segmented Sort Kernel

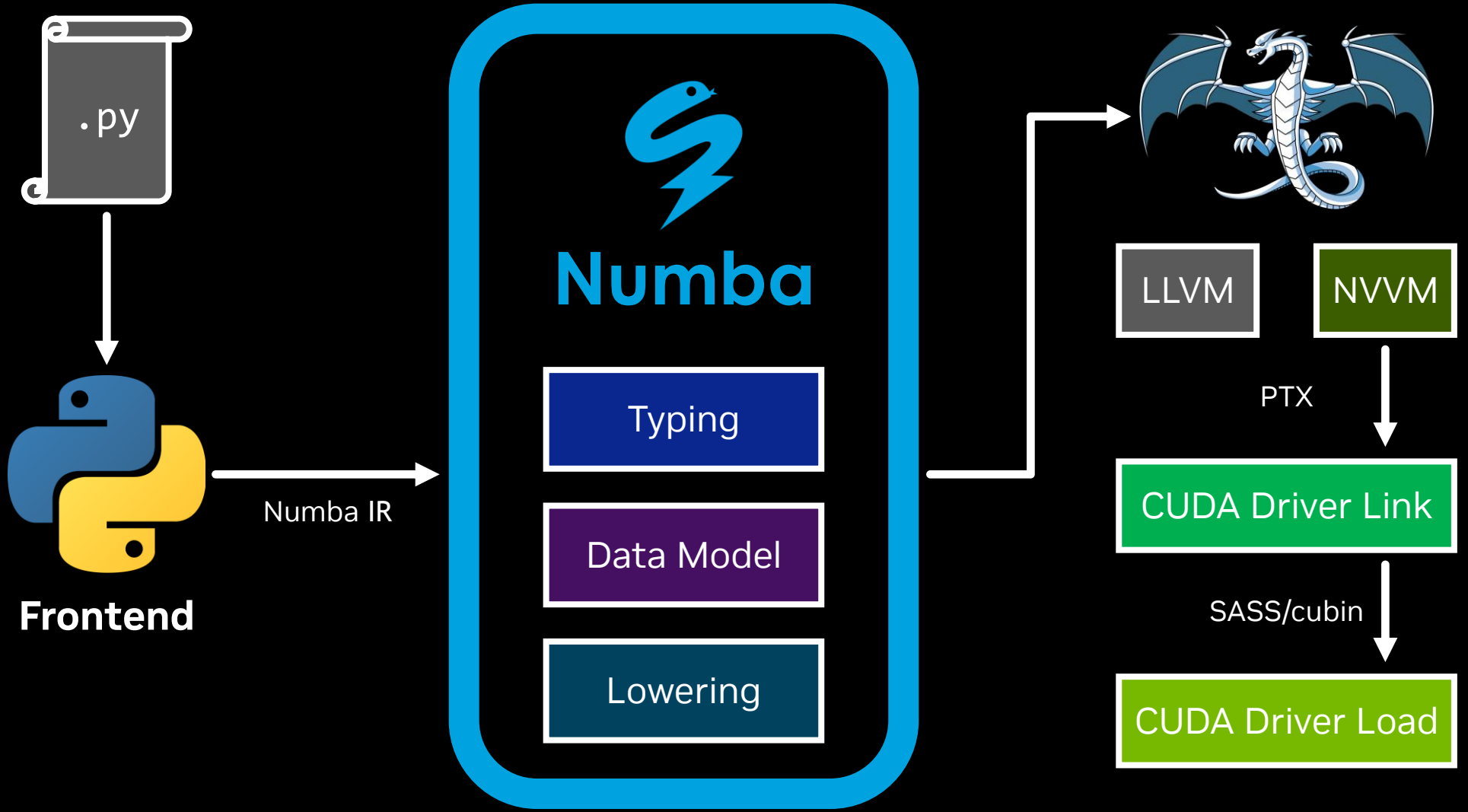


CUDA FFT Convolution Kernel



A6000, CUDA 12.6, "CUDA: New Features and Beyond", GTC 2025





```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]
```

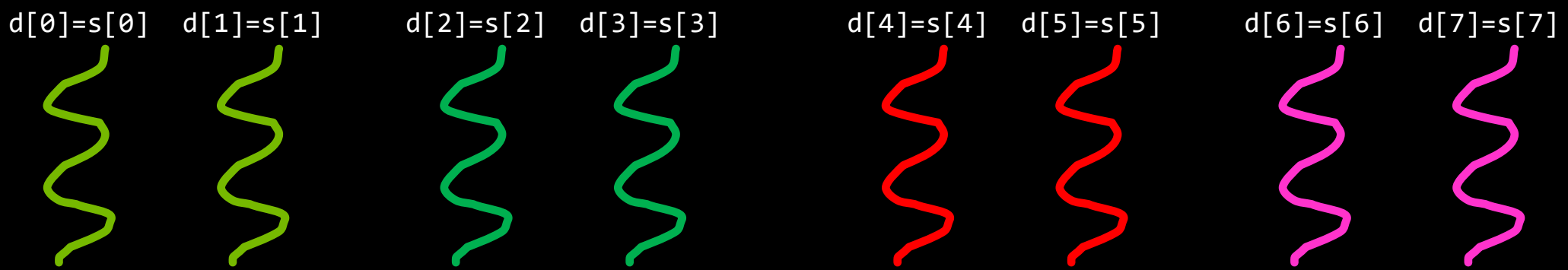
```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]
```

```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]
```

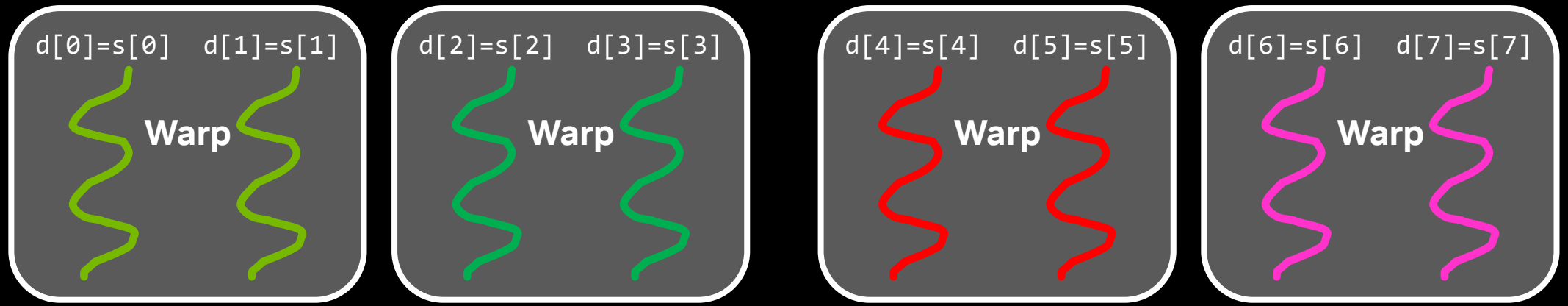
```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]
```

```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]
```

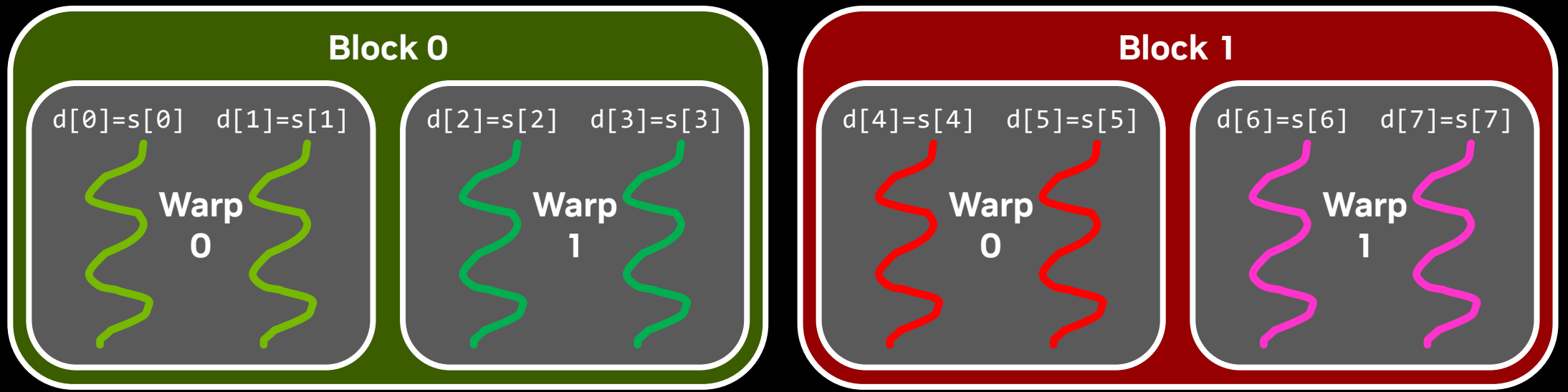
```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]
```



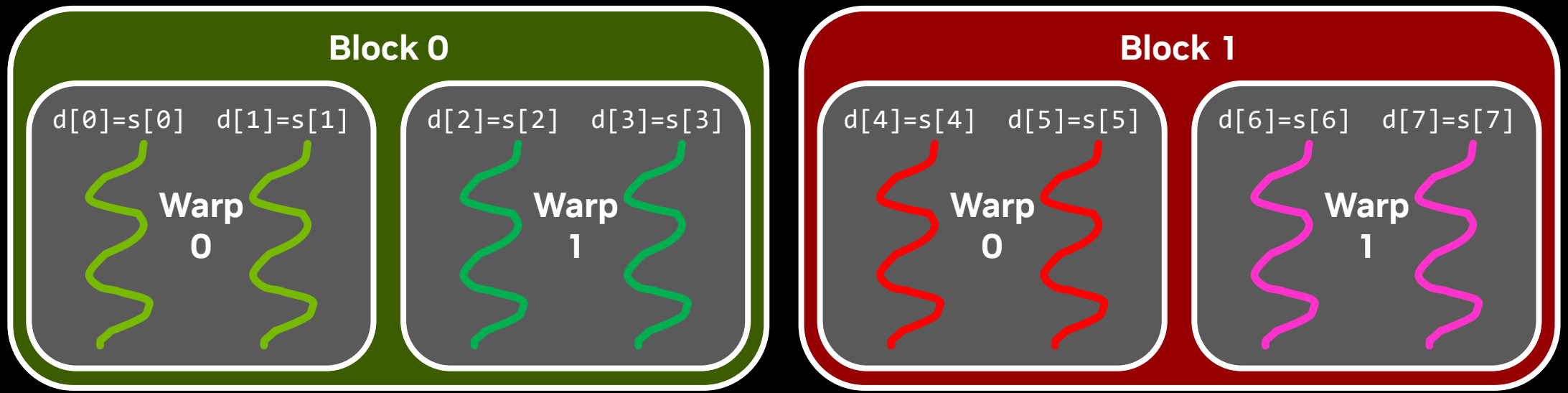
```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]
```



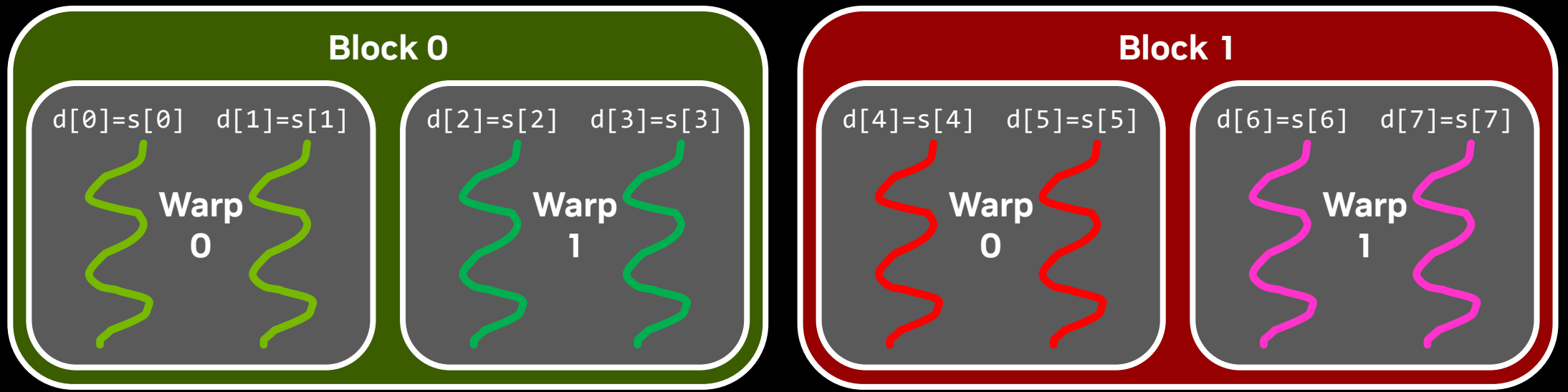
```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]
```



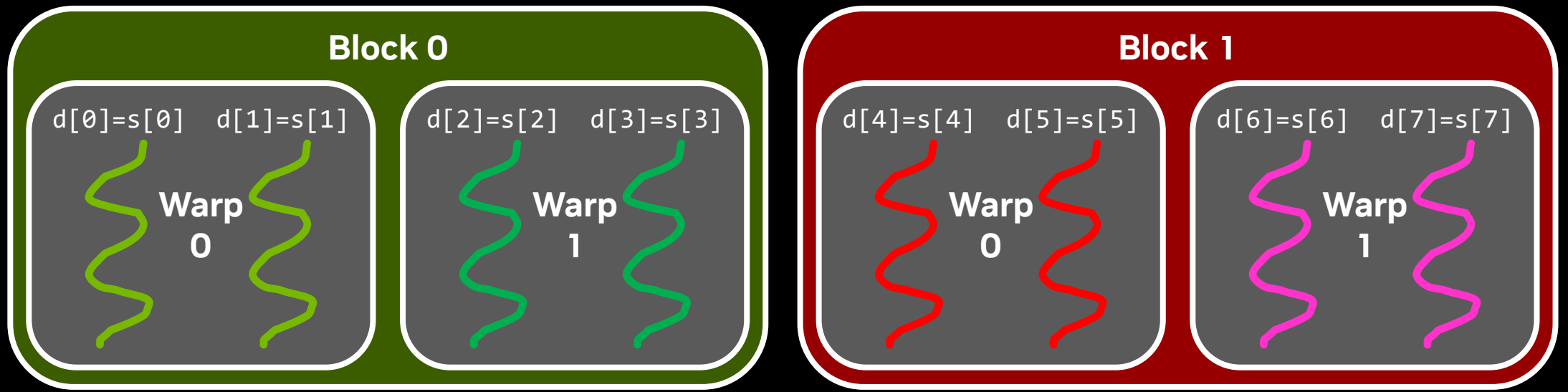
```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]

s = cp.arange(N)
d = cp.empty_like(s)
copy[2, 4](s, d)
```



```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]

s = cp.arange(N)
d = cp.empty_like(s)
copy[2, 4](s, d)
```



```
from numba import cuda

@cuda.jit
def copy(s, d):
  t = cuda.grid(1)
  d[t] = s[t]

s = cp.arange(N)
d = cp.empty_like(s)
copy[2, 4](s, d)
```



```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[…, …](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[…, …](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[…, …](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[…, …](s, d, T)
```

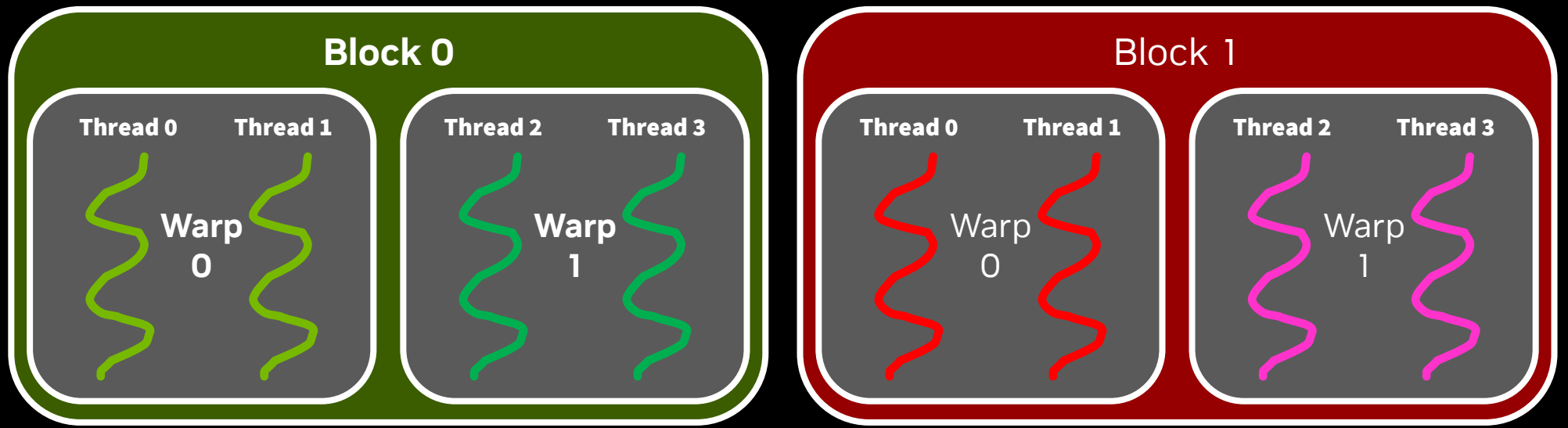
```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), …](s, d, T)
```

N = Size of the arrays
B = Number of thread blocks
T = Items per thread

```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

N = Size of the arrays
B = Number of thread blocks
T = Items per thread

```
cuda.grid(1)      = Global thread index (unique)
cuda.threadIdx.x  = Block-local thread index
cuda.threadDim.x  = # of threads per block
cuda.blockIdx.x   = Global block index (unique)
cuda.blockDim.x   = # of blocks
```



Exercise 10

Kernel Authoring

Copy

```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

Step	Thread 0	Thread 1	Thread 2	Thread 3
load	d[0]	d[4]	d[8]	d[12]
store	s[0]	s[4]	s[8]	s[12]
load	d[1]	d[5]	d[9]	d[13]
store	s[1]	s[5]	s[9]	s[13]
load	d[2]	d[6]	d[10]	d[14]
store	s[2]	s[6]	s[10]	s[14]
load	d[3]	d[7]	d[11]	d[15]
store	s[3]	s[7]	s[11]	d[15]
…	…	…	…	…

```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

Step	Thread 0	Thread 1	Thread 2	Thread 3
load	d[0]	d[4]	d[8]	d[12]
store	s[0]	s[4]	s[8]	s[12]
load	d[1]	d[5]	d[9]	d[13]
store	s[1]	s[5]	s[9]	s[13]
load	d[2]	d[6]	d[10]	d[14]
store	s[2]	s[6]	s[10]	s[14]
load	d[3]	d[7]	d[11]	d[15]
store	s[3]	s[7]	s[11]	d[15]
…	…	…	…	…

```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

Step	Thread 0	Thread 1	Thread 2	Thread 3
load	d[0]	d[4]	d[8]	d[12]
load	d[1]	d[5]	d[9]	d[13]
load	d[2]	d[6]	d[10]	d[14]
load	d[3]	d[7]	d[11]	d[15]
store	s[0]	s[4]	s[8]	s[12]
store	s[1]	s[5]	s[9]	s[13]
store	s[2]	s[6]	s[10]	s[14]
store	s[3]	s[7]	s[11]	d[15]
…	…	…	…	…

```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

Step	Thread 0	Thread 1	Thread 2	Thread 3
load	d[0:3]	d[4:7]	d[8:11]	d[12:15]
store	s[0:3]	s[4:7]	s[8:11]	s[12:15]
…	…	…	…	…

```
@cuda.jit
def copy(s, d, items):
 t = cuda.grid(1) * items
 for i in range(items):
   d[t + i] = s[t + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

Step	Thread 0	Thread 1	Thread 2	Thread 3
load	d[0]	d[4]	d[8]	d[12]
store	s[0]	s[4]	s[8]	s[12]
load	d[1]	d[5]	d[9]	d[13]
store	s[1]	s[5]	s[9]	s[13]
load	d[2]	d[6]	d[10]	d[14]
store	s[2]	s[6]	s[10]	s[14]
load	d[3]	d[7]	d[11]	d[15]
store	s[3]	s[7]	s[11]	d[15]
…	…	…	…	…

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

Step	Thread 0	Thread 1	Thread 2	Thread 3
load	d[0]	d[1]	d[2]	d[3]
store	s[0]	s[1]	s[2]	s[3]
load	d[4]	d[5]	d[6]	d[7]
store	s[4]	s[5]	s[6]	s[7]
load	d[8]	d[9]	d[10]	d[11]
store	s[8]	s[9]	s[10]	s[11]
load	d[12]	d[13]	d[14]	d[15]
store	s[12]	s[13]	s[14]	d[15]
…	…	…	…	…

```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

Step	Thread [0:3]
load	d[0:3]
store	s[0:3]
load	d[4:7]
store	s[4:7]
load	d[8:11]
store	s[8:11]
load	d[12:15]
store	s[12:15]
…	…

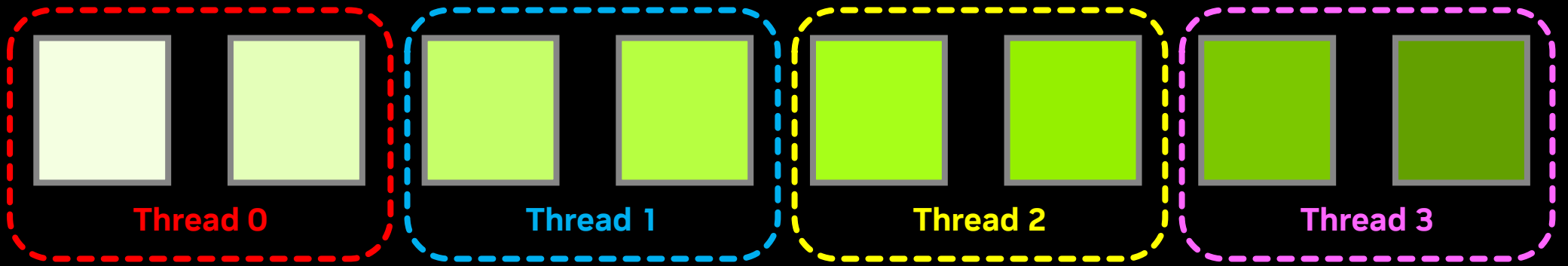
```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

Step	Thread [0:3]
load	d[0:3]
load	d[4:7]
load	d[8:11]
load	d[12:15]
store	s[0:3]
store	s[4:7]
store	s[8:11]
store	s[12:15]
…	…

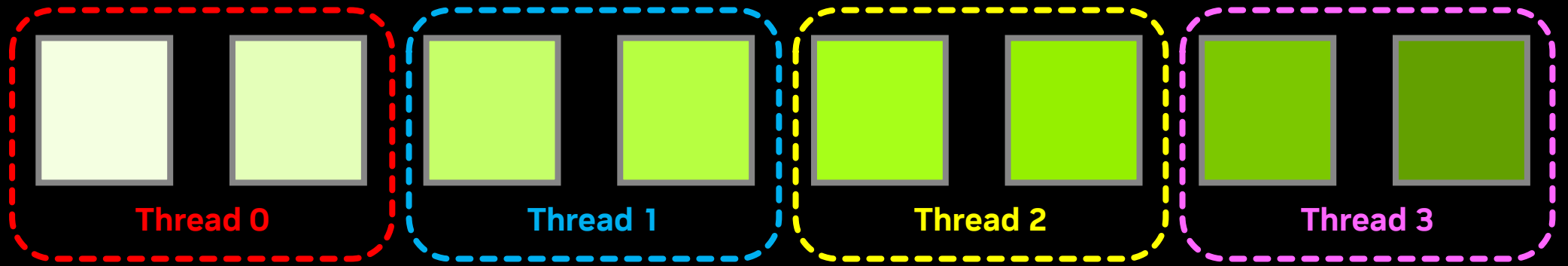
```
@cuda.jit
def copy(s, d, items):
  bd = cuda.blockDim.x
  bx = cuda.blockIdx.x
  tx = cuda.threadIdx.x
  tile = bd * items

  base = tx + bx * tile
  for i in range(0, tile, bd):
    d[base + i] = s[base + i]

s = cp.arange(N)
d = cp.empty_like(s)
copy[int(N/(B*T)), B](s, d, T)
```

Step	Thread [0:3]
load	d[0:15]
store	s[0:15]
…	…

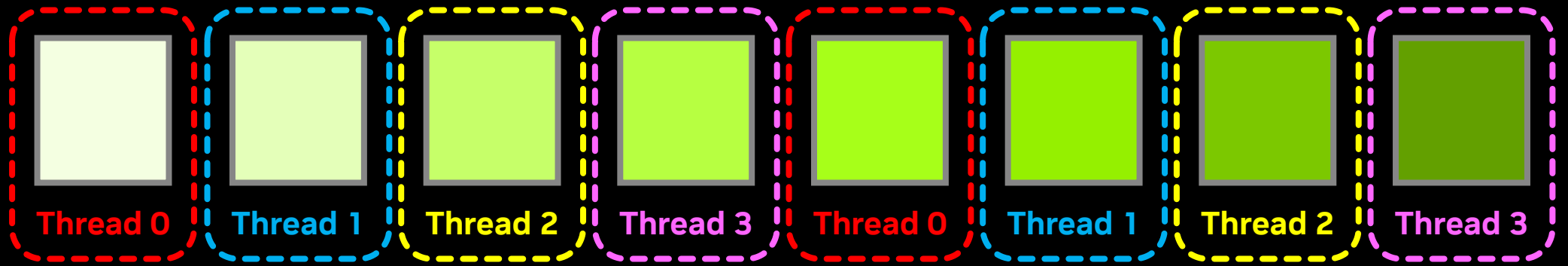
Blocked Arrangement

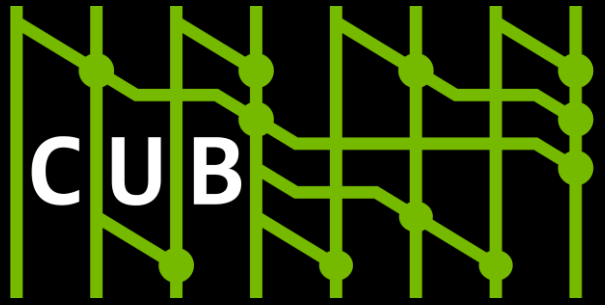


Blocked Arrangement



Striped Arrangement

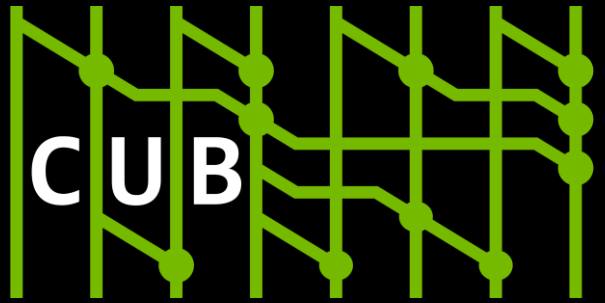




Cooperative primitives for CUDA C++

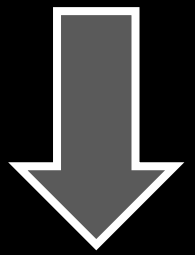
https://nvidia.github.io/cccl/cub

```
template <int BLOCK_THREADS, int ITEMS>
__global__ void BlockSortKernel(int *d_in, int *d_out)
{
  using BlockLoadT = cub::BlockLoad<
    int, BLOCK_THREADS, ITEMS, cub::BLOCK_LOAD_TRANSPOSE>;
  using BlockStoreT = cub::BlockStore<
    int, BLOCK_THREADS, ITEMS, cub::BLOCK_STORE_TRANSPOSE>;
  using BlockRadixSortT = cub::BlockRadixSort<
    int, BLOCK_THREADS, ITEMS>;

  // Allocate type-safe, repurposable shared memory
  __shared__ union {
    typename BlockLoadT::TempStorage       load;
    typename BlockStoreT::TempStorage      store;
    typename BlockRadixSortT::TempStorage  sort;
  } temp;

  // Obtain this block's segment of consecutive keys
  int thread_keys[ITEMS];
  int offset = blockIdx.x * (BLOCK_THREADS * ITEMS);
  BlockLoadT(temp.load).Load(d_in + offset, thread_keys);

  __syncthreads();        // Barrier for smem reuse

  // Collectively sort the keys
  BlockRadixSortT(temp.sort).Sort(thread_keys);

  __syncthreads();        // Barrier for smem reuse

  // Store the sorted segment
  BlockStoreT(temp.store).Store(d_out + offset, thread_keys);
}
```



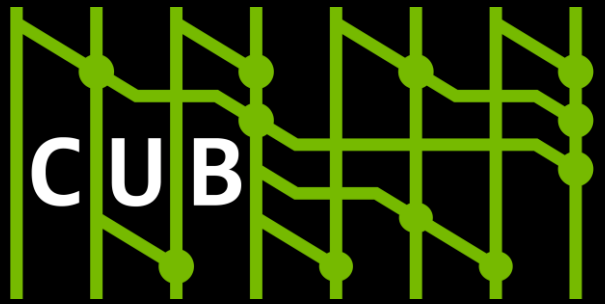
Cooperative primitives for CUDA C++

https://nvidia.github.io/cccl/cub

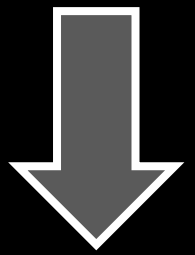


```
import cuda.cccl.cooperative
```

```
template <int BLOCK_THREADS, int ITEMS>
__global__ void BlockSortKernel(int *d_in, int *d_out)
{
  using BlockLoadT = cub::BlockLoad<
    int, BLOCK_THREADS, ITEMS, cub::BLOCK_LOAD_TRANSPOSE>;
  using BlockStoreT = cub::BlockStore<
    int, BLOCK_THREADS, ITEMS, cub::BLOCK_STORE_TRANSPOSE>;
  using BlockRadixSortT = cub::BlockRadixSort<
    int, BLOCK_THREADS, ITEMS>;

  // Allocate type-safe, repurposable shared memory
  __shared__ union {
    typename BlockLoadT::TempStorage       load;
    typename BlockStoreT::TempStorage      store;
    typename BlockRadixSortT::TempStorage  sort;
  } temp;

  // Obtain this block's segment of consecutive keys
  int thread_keys[ITEMS];
  int offset = blockIdx.x * (BLOCK_THREADS * ITEMS);
  BlockLoadT(temp.load).Load(d_in + offset, thread_keys);

  __syncthreads();        // Barrier for smem reuse

  // Collectively sort the keys
  BlockRadixSortT(temp.sort).Sort(thread_keys);

  __syncthreads();        // Barrier for smem reuse

  // Store the sorted segment
  BlockStoreT(temp.store).Store(d_out + offset, thread_keys);
}
```

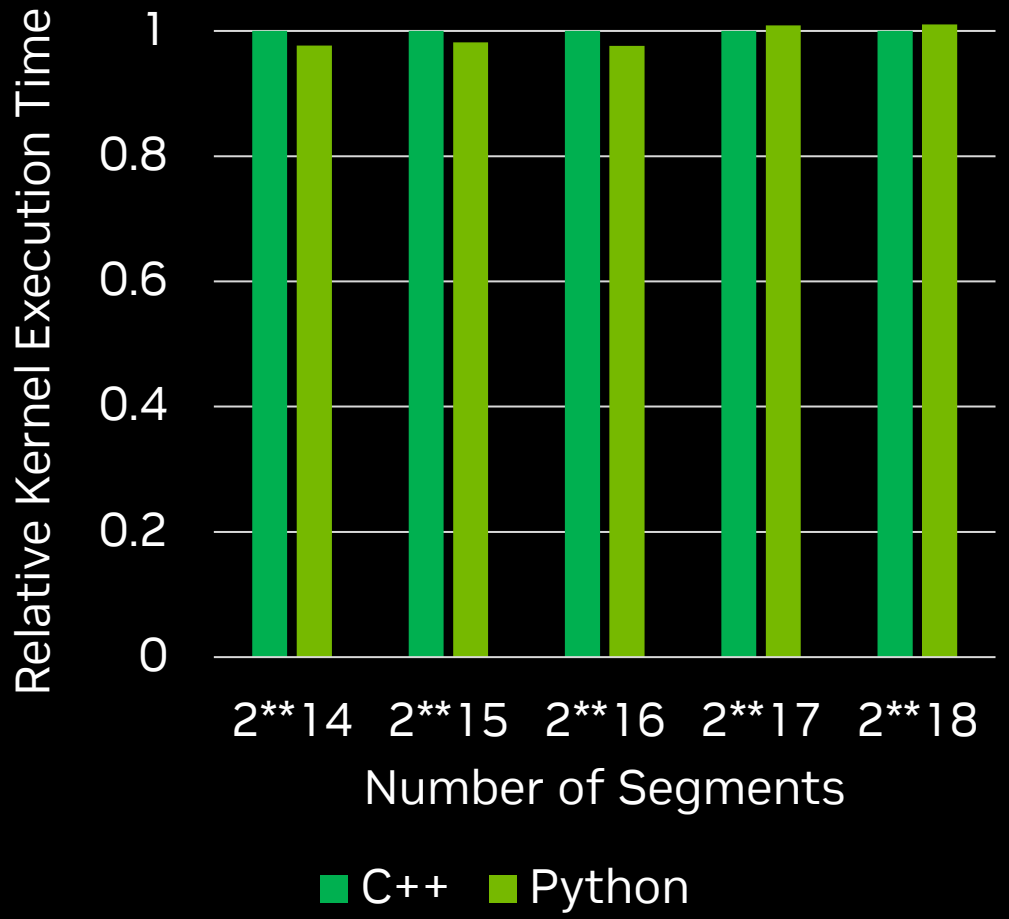


Cooperative primitives for CUDA C++
https://nvidia.github.io/cccl/cub



```
import cuda.cccl.cooperative
```

CUDA Segmented Sort Kernel



A6000, CUDA 12.6, "CUDA: New Features and Beyond", GTC 2025

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def copy(src, dst):
  …
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def copy(src, dst):
  locals = cuda.local.array(T, dtype=cp.int32)
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def copy(src, dst):
  locals = cuda.local.array(T, dtype=cp.int32)
  base = cuda.blockIdx.x * B * T
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def copy(src, dst):
  locals = cuda.local.array(T, dtype=cp.int32)
  base = cuda.blockIdx.x * B * T

  block_load(src[base : base + B * T], locals)
```

```python
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def copy(src, dst):
  locals = cuda.local.array(T, dtype=cp.int32)
  base = cuda.blockIdx.x * B * T

  block_load(src[base : base + B * T], locals)
  block_store(dst[base : base + B * T], locals)
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def copy(src, dst):
  locals = cuda.local.array(T, dtype=cp.int32)
  base = cuda.blockIdx.x * B * T

  block_load(src[base : base + B * T], locals)
  block_store(dst[base : base + B * T], locals)

src = np.arange(N, dtype='int32')
dst = np.empty_like(src)
copy[int(N / (B * T)), B](src, dst)
assert np.all(src == dst)
```

```
import cupy as cp
import numba.cuda as cuda
import cuda.cccl.cooperative.experimental as coop

N = 2**28 # Size of the input/output
B = 256   # Number of thread in each block
T = 64    # Number of items in each thread's registers

block_load  = coop.block.load(cp.int32, B, T, 'striped')
block_store = coop.block.store(cp.int32, B, T, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def copy(src, dst):
  locals = cuda.local.array(T, dtype=cp.int32)
  base = cuda.blockIdx.x * B * T

  block_load(src[base : base + B * T], locals)
  block_store(dst[base : base + B * T], locals)

src = np.arange(N, dtype='int32')
dst = np.empty_like(src)
copy[int(N / (B * T)), B](src, dst)
assert np.all(src == dst)
```

"Whose woods these are I think I know."

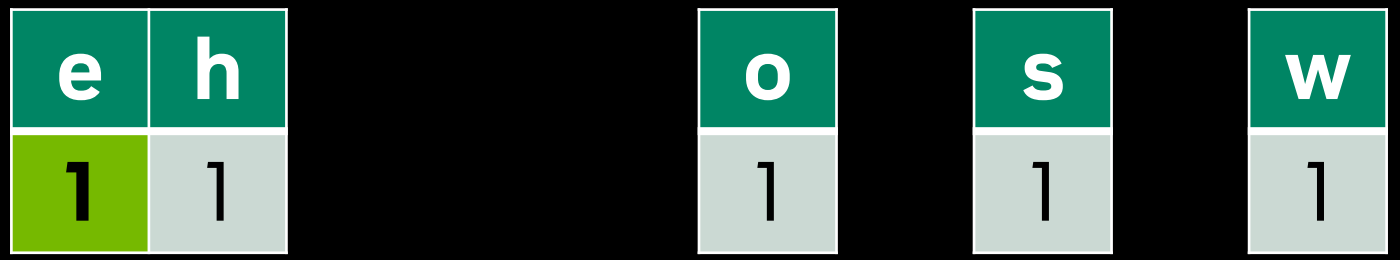
"Whose woods these are I think I know."

"Whose woods these are I think I know."

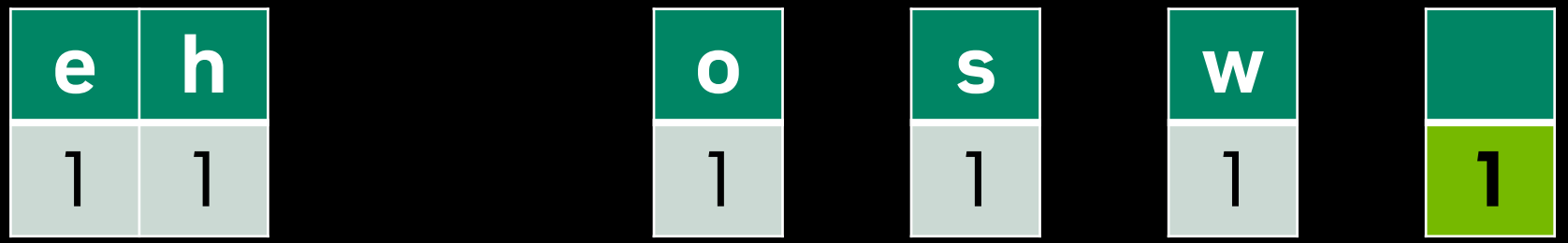
"Whose woods these are I think I know."

"Whose woods these are I think I know."

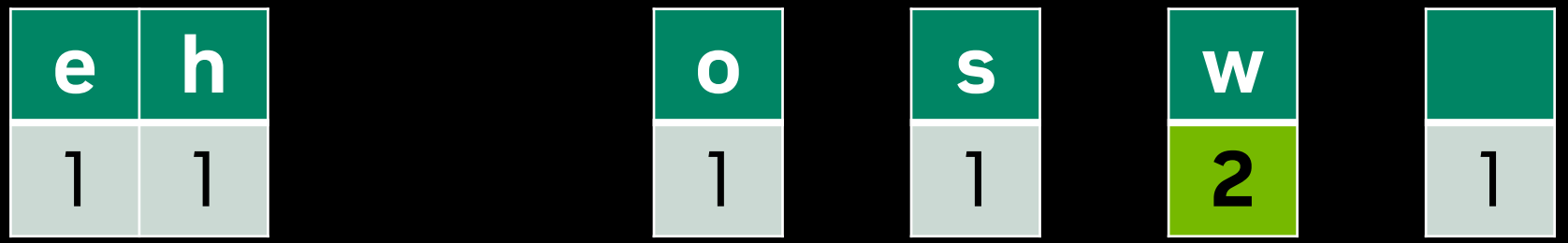
"Whose woods these are I think I know."



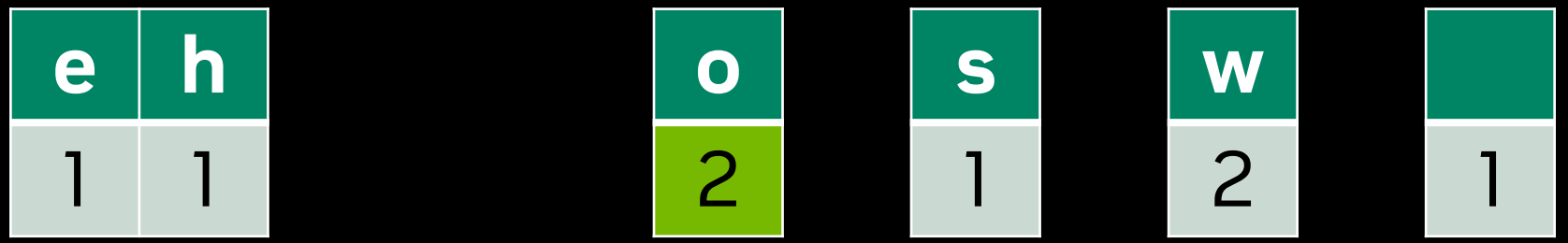
"Whose woods these are I think I know."



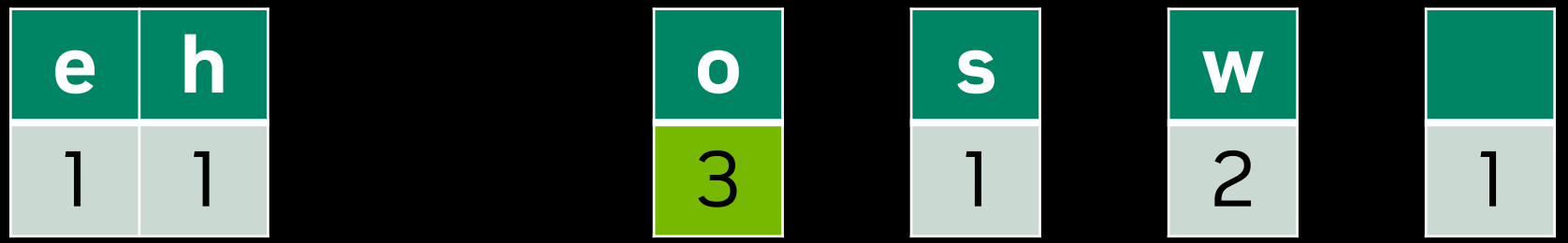
"Whose woods these are I think I know."



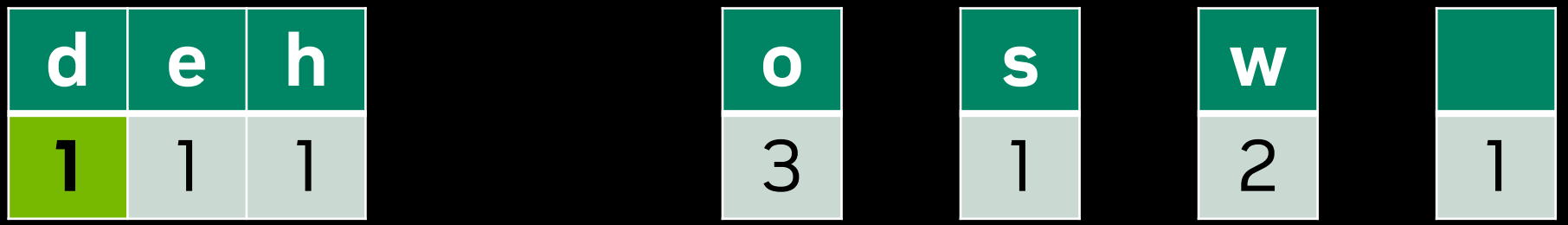
"Whose woods these are I think I know."



"Whose woods these are I think I know."



"Whose woods these are I think I know."



"Whose woods these are I think I know."

d	e	h
1	1	1

o	s	w	
3	2	2	1

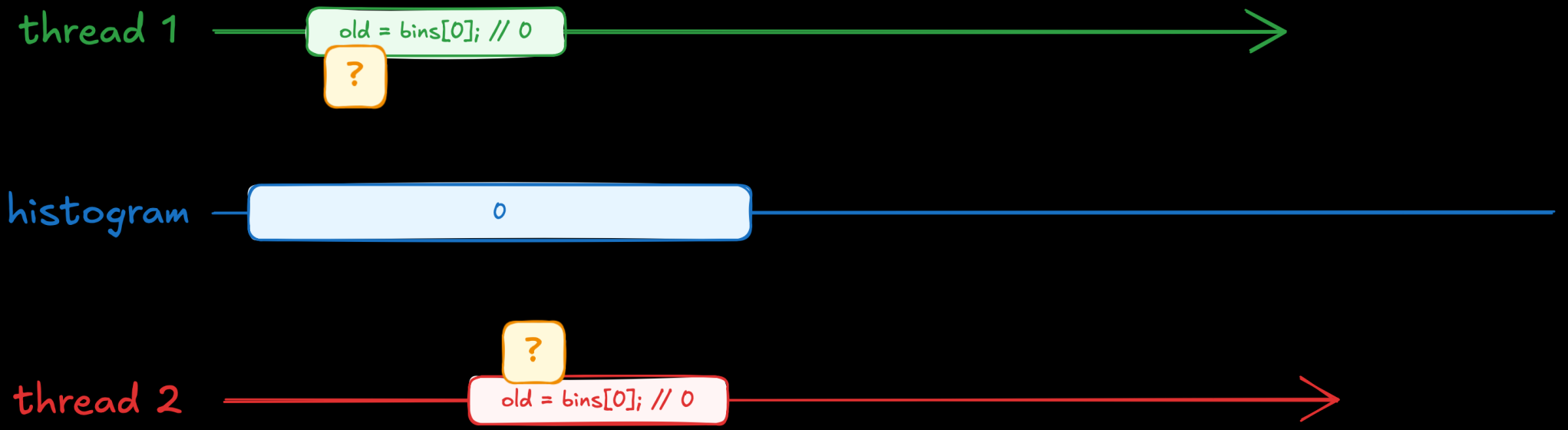
"Whose woods these are I think I know."

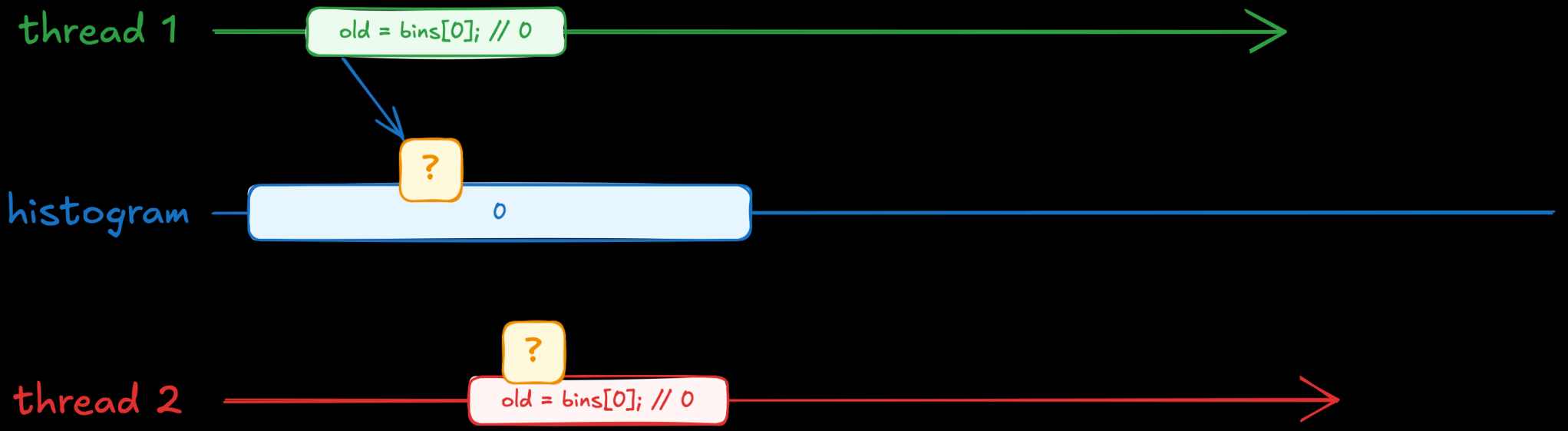
a	d	e	h	i	k	n	o	r	s	t	w	.	
1	1	4	3	3	2	2	4	1	3	2	3	1	7

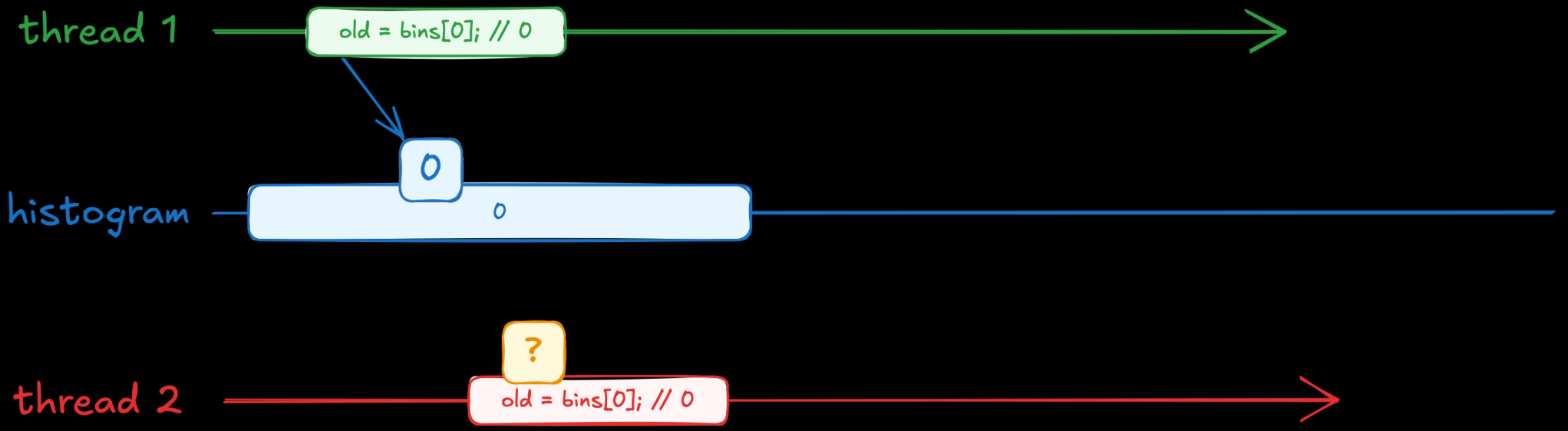
Exercise 11

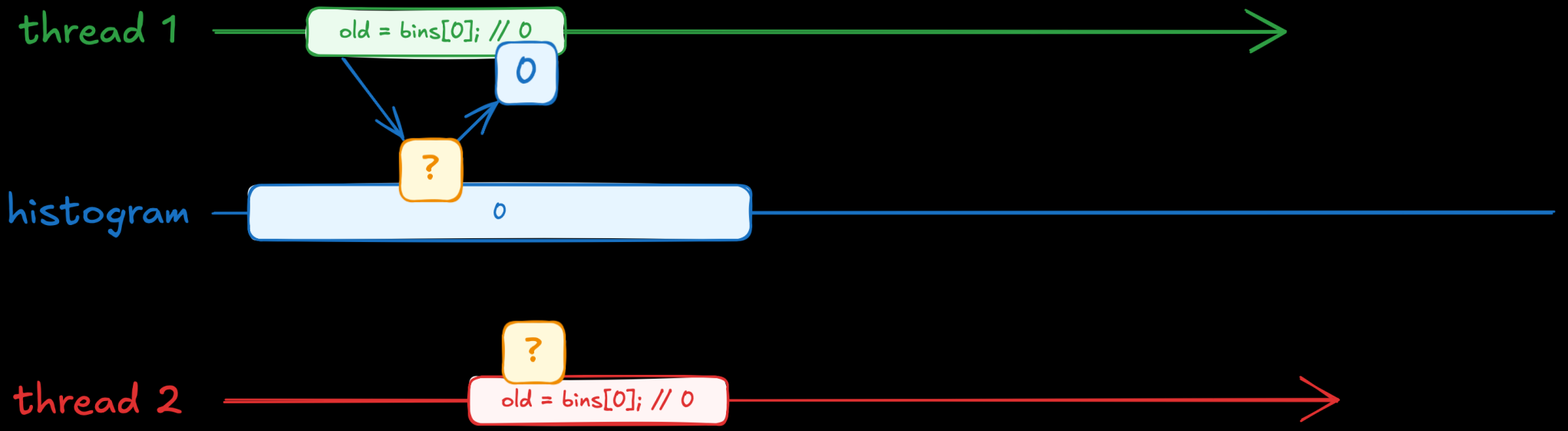
Kernel Authoring

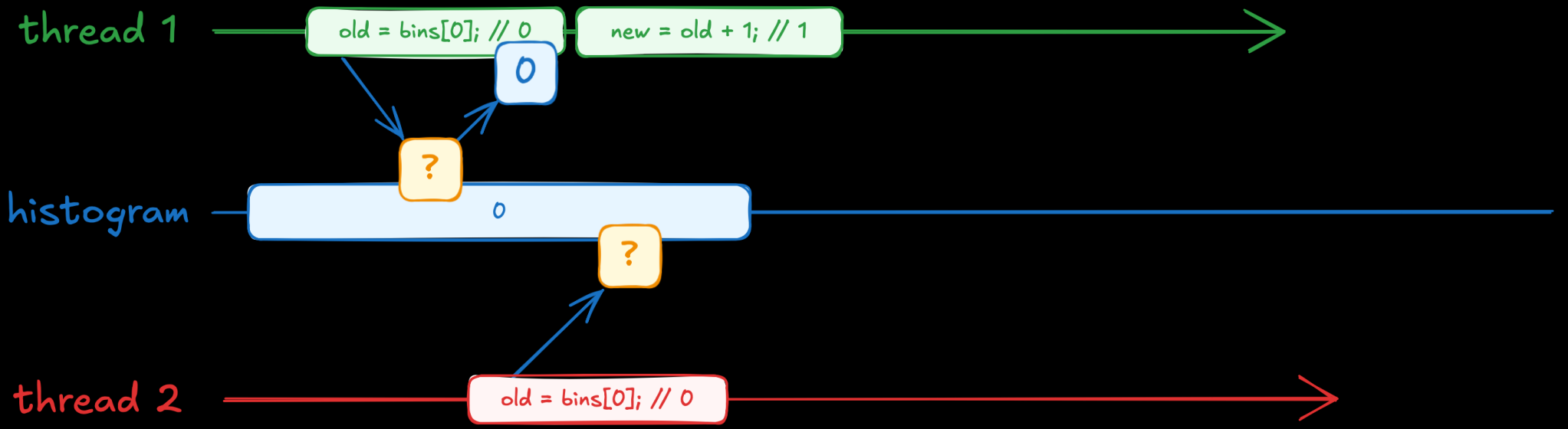
Book Histogram

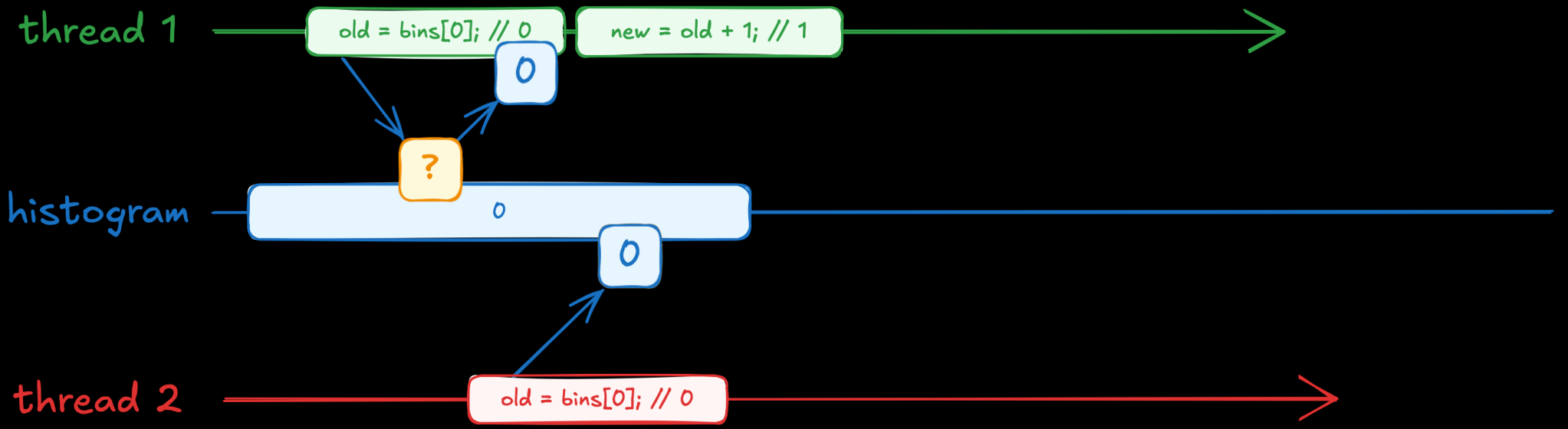


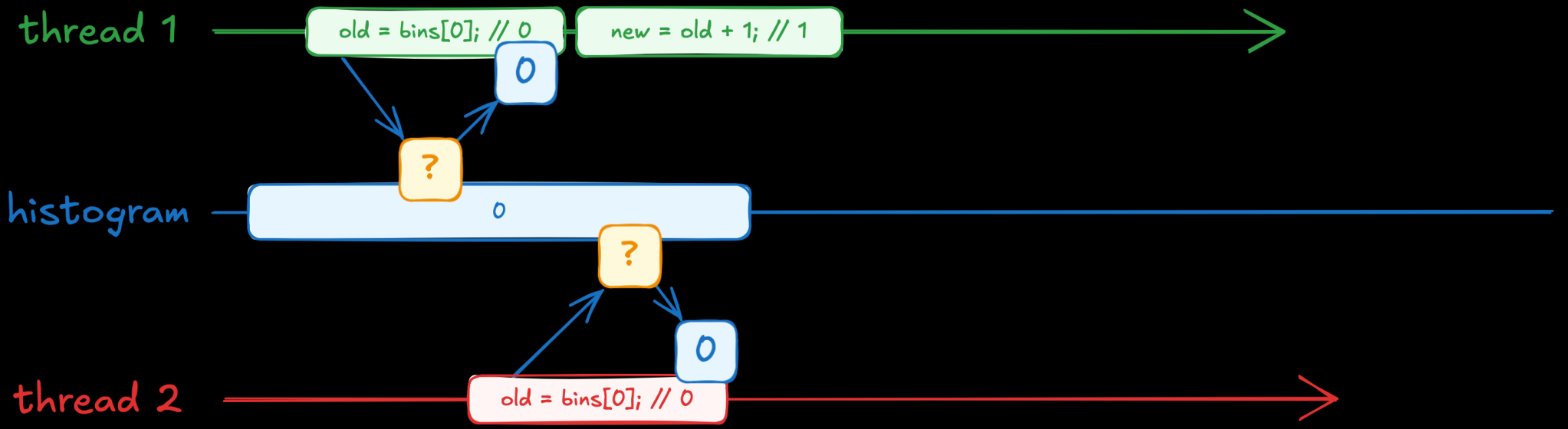

thread 1
old = bins[0]; // 0
?
histogram
0
?
thread 2
old = bins[0]; // 0

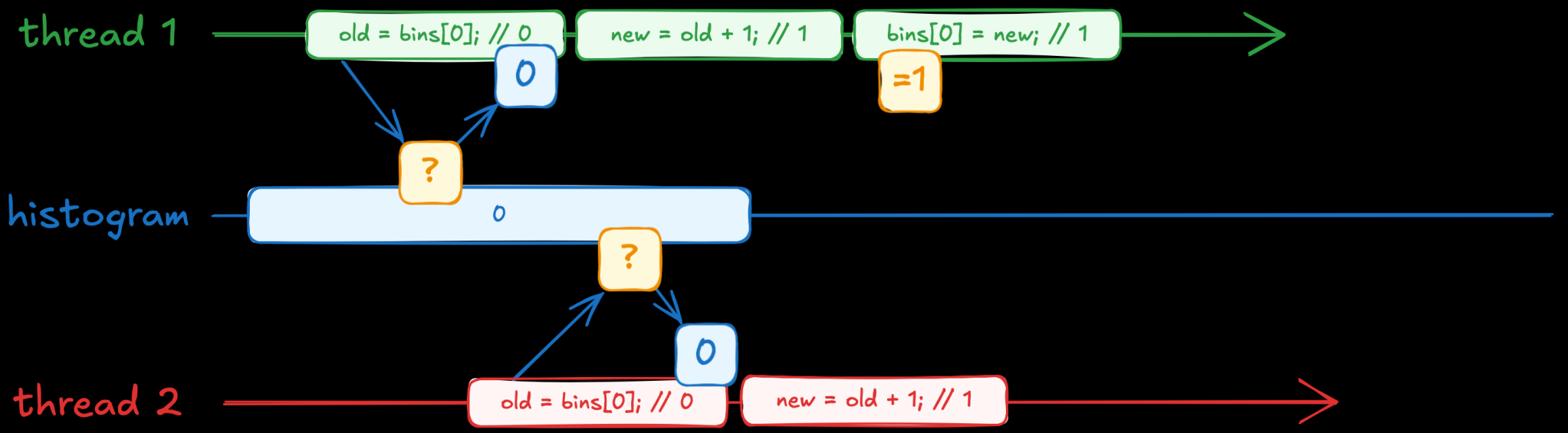


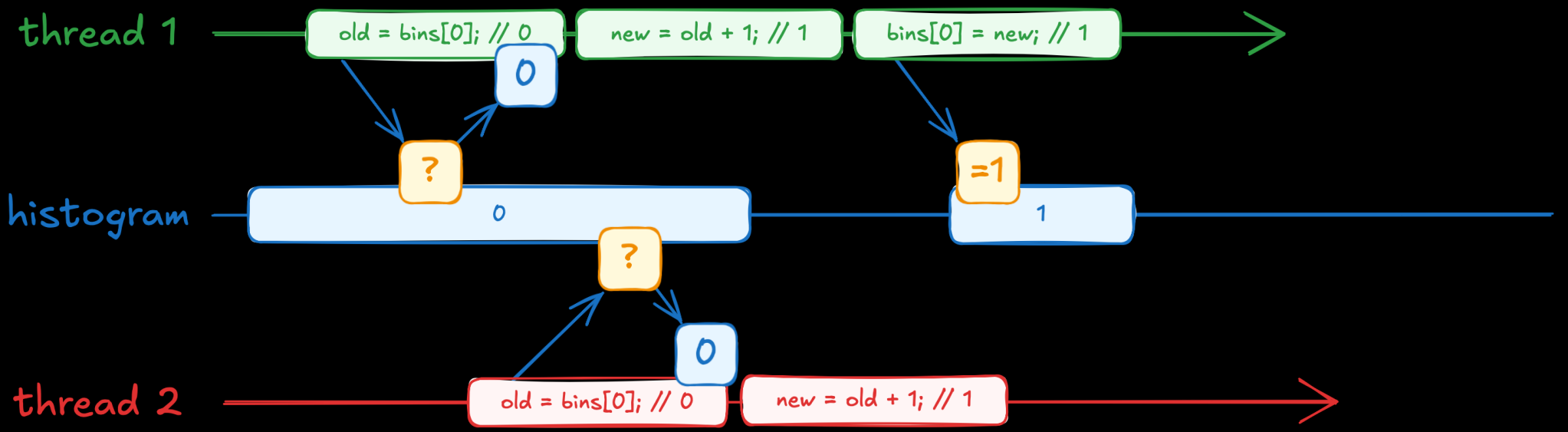


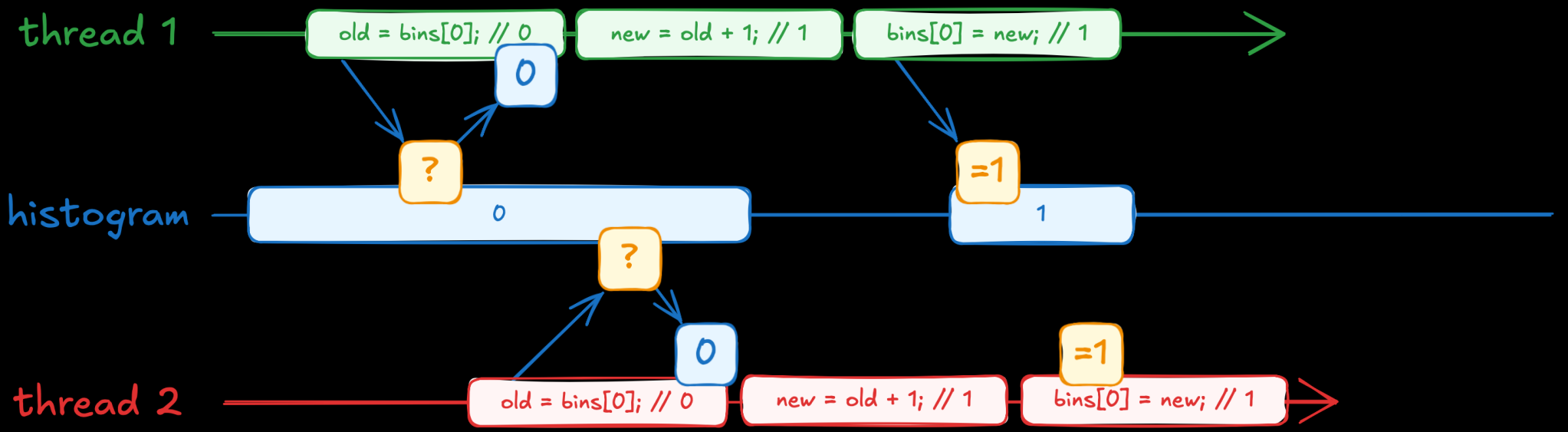

thread 1
old = bins[0]; // 0
new = old + 1; // 1
0
?
histogram
0
?
thread 2
old = bins[0]; // 0

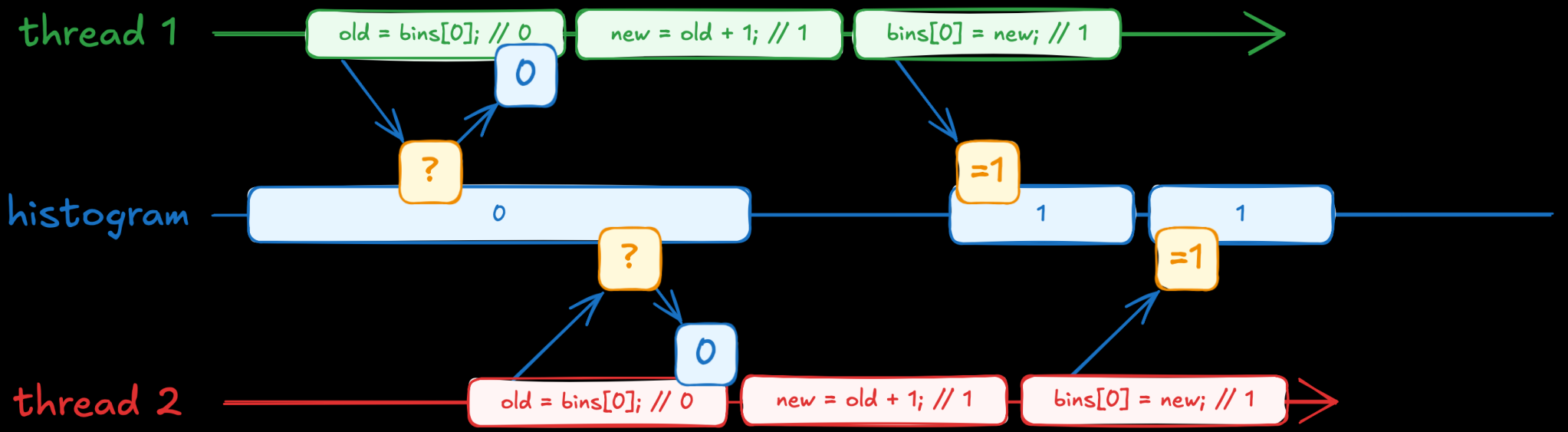

thread 1
old = bins[0]; // 0
new = old + 1; // 1
0
?
histogram
0
0
thread 2
old = bins[0]; // 0


thread 1
old = bins[0]; // 0
new = old + 1; // 1
0
?
histogram
0
?
0
thread 2
old = bins[0]; // 0


thread 1
old = bins[0]; // 0
new = old + 1; // 1
bins[0] = new; // 1
0
=1
?
histogram
0
?
0
thread 2
old = bins[0]; // 0
new = old + 1; // 1


thread 1
old = bins[0]; // 0
new = old + 1; // 1
bins[0] = new; // 1
0
?
=1
histogram
0
1
?
0
thread 2
old = bins[0]; // 0
new = old + 1; // 1

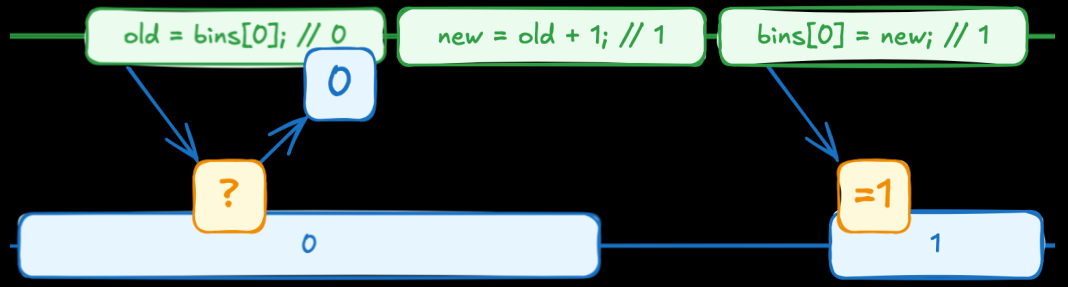

thread 1
old = bins[0]; // 0
new = old + 1; // 1
bins[0] = new; // 1
0
?
=1
histogram
0
1
?
0
=1
thread 2
old = bins[0]; // 0
new = old + 1; // 1
bins[0] = new; // 1

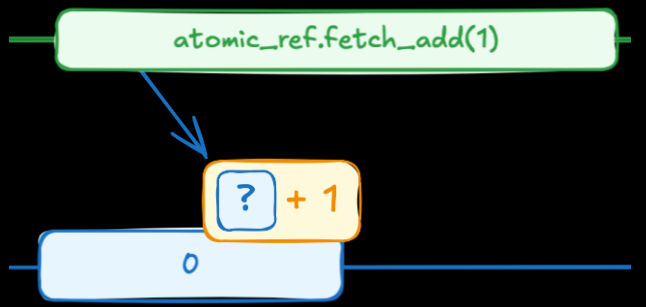

thread 1
old = bins[0]; // 0
new = old + 1; // 1
bins[0] = new; // 1
0
?
=1
histogram
0
1
1
?
=1
0
thread 2
old = bins[0]; // 0
new = old + 1; // 1
bins[0] = new; // 1

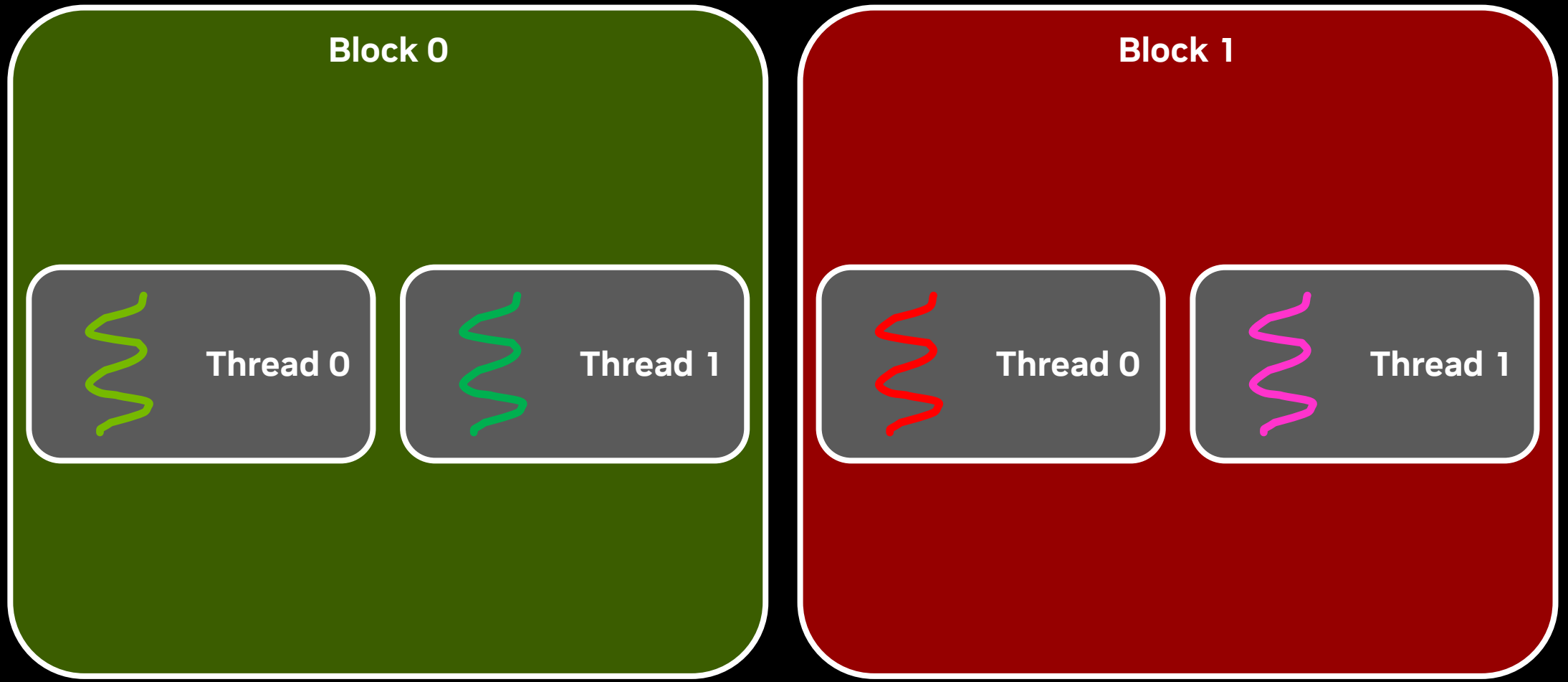
To fix the data race, the read, modify and write must be a single indivisible operation.

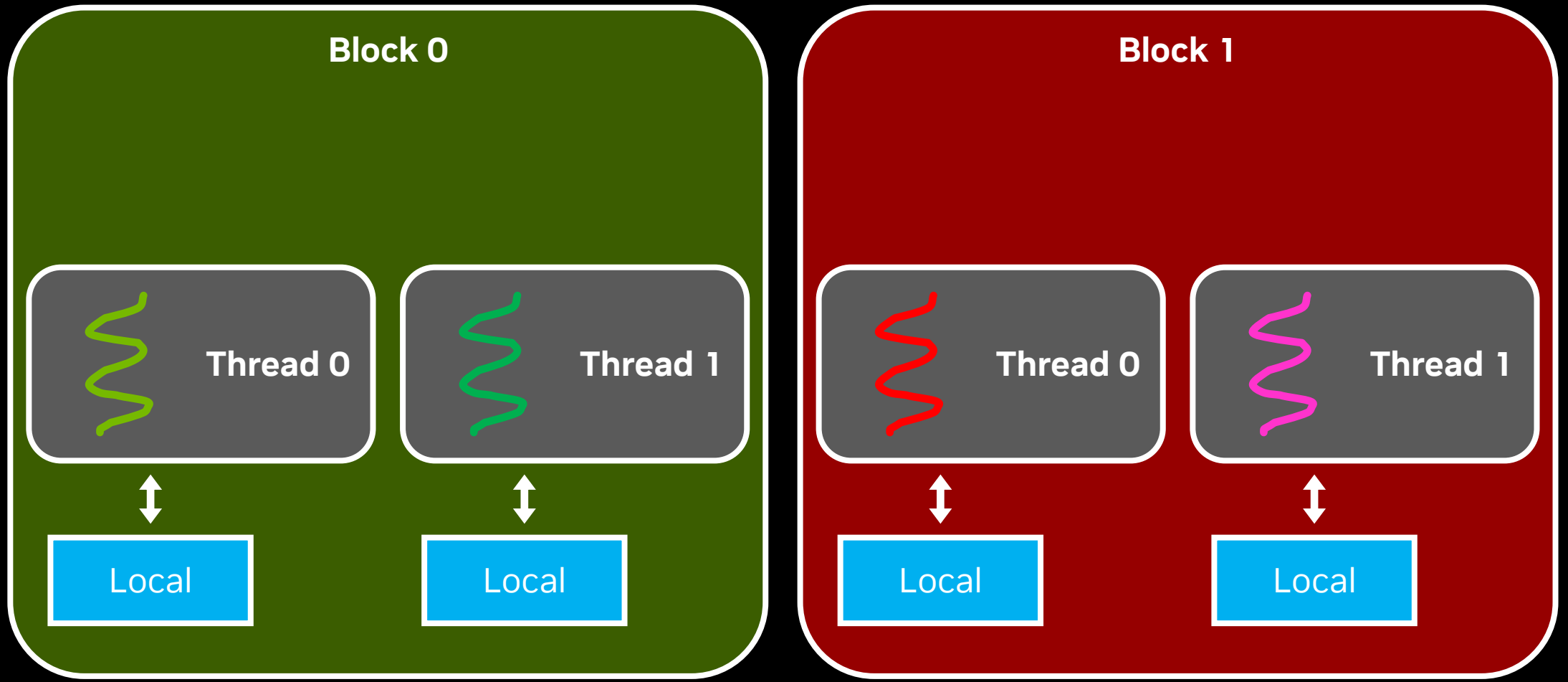
```
old_count = histogram[value]
new_count = old_count + 1
histogram[value] = new_count
```

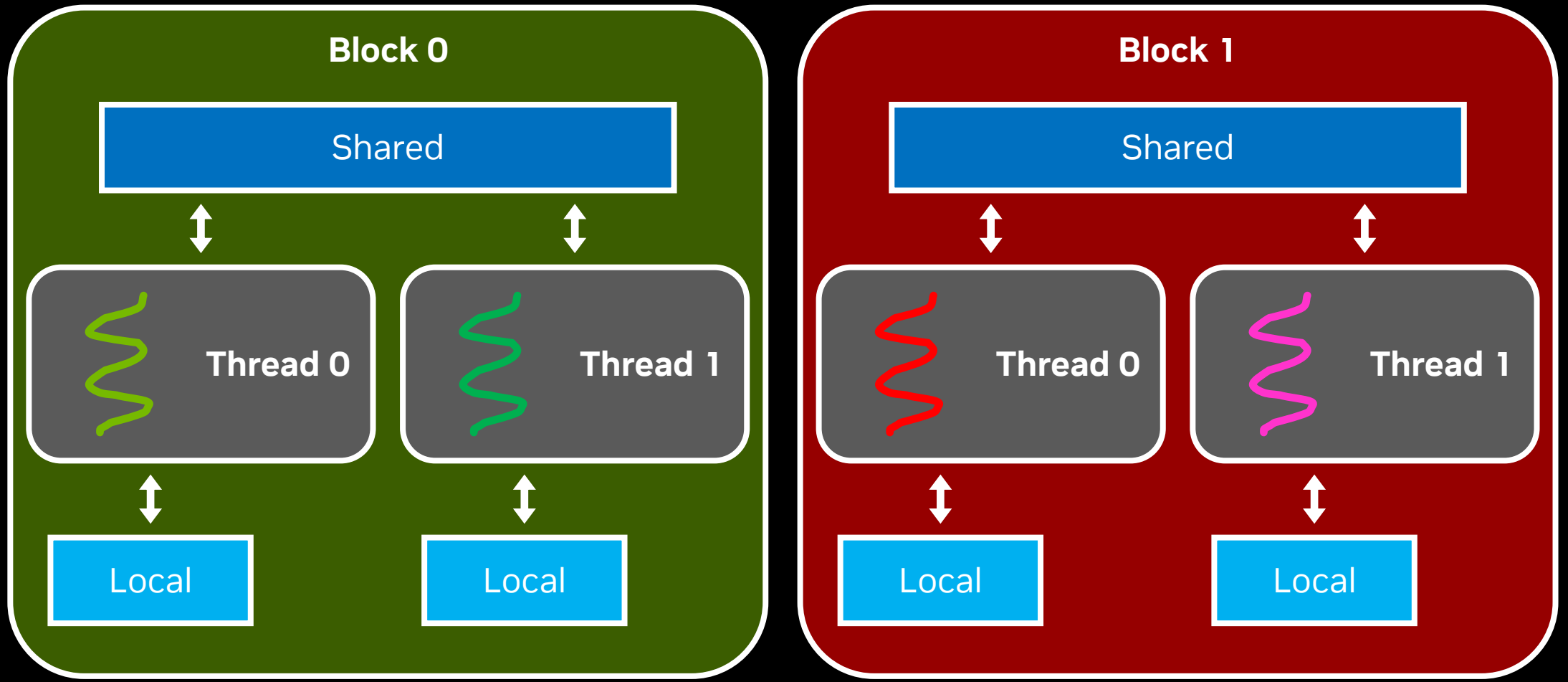
```
cuda.atomic.add(
  histogram, value, 1)
```

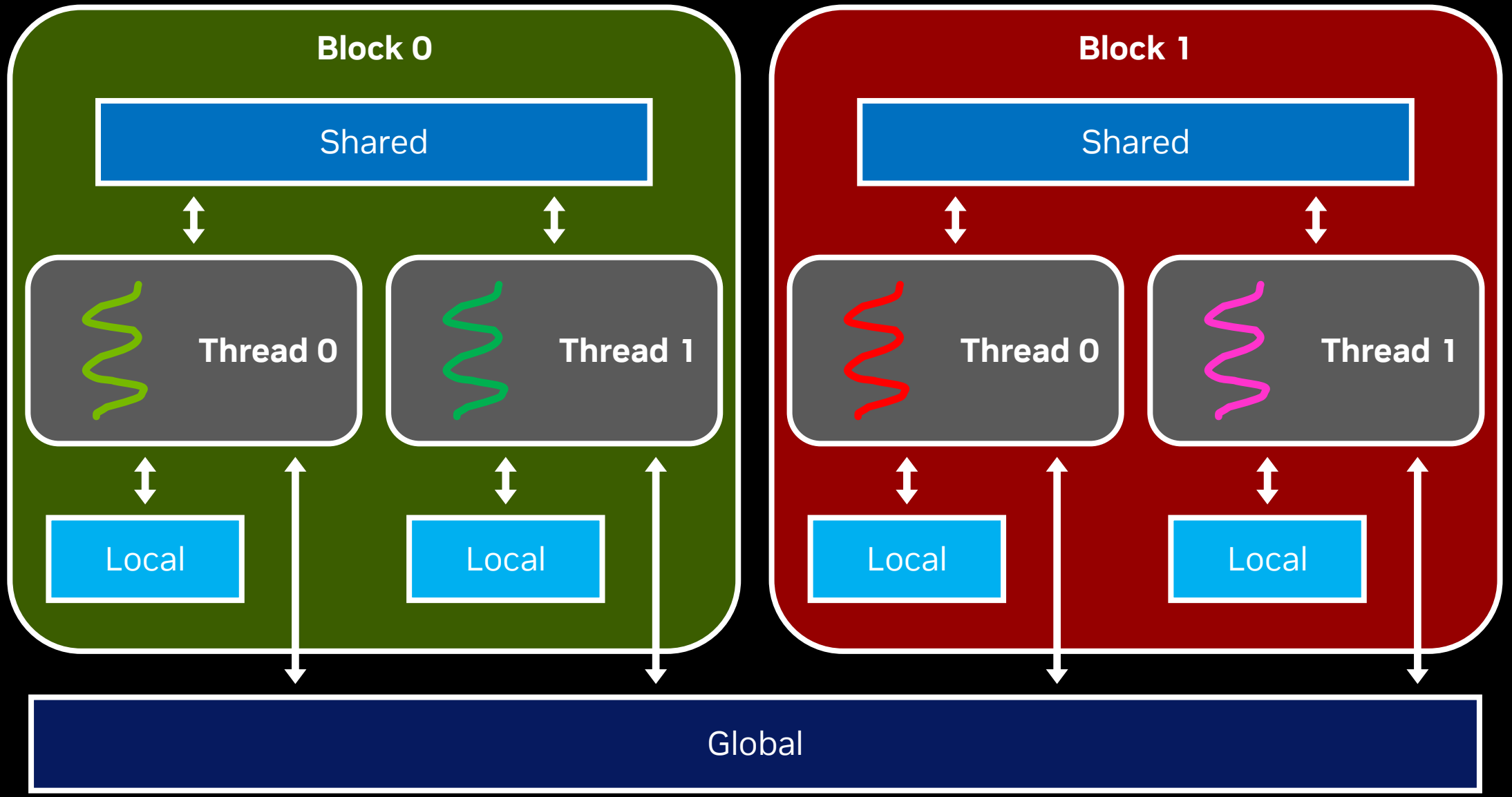












```
block_load  = coop.block.load(cp.int32, 32, 1, 'striped')
block_store = coop.block.store(cp.int32, 32, 1, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def adjacent_difference(input, output):
  shared = cuda.shared.array(32, dtype=cp.int32)

  block_load(input, shared)

  if not threadIdx.x == 0:
    block_store(input,
      shared[threadIdx.x] - shared[threadIdx.x - 1])
```

```
block_load  = coop.block.load(cp.int32, 32, 1, 'striped')
block_store = coop.block.store(cp.int32, 32, 1, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def adjacent_difference(input, output):
  shared = cuda.shared.array(32, dtype=cp.int32)

  block_load(input, shared)

  if not threadIdx.x == 0:
    block_store(input,
      shared[threadIdx.x] - shared[threadIdx.x - 1])
```

```
block_load  = coop.block.load(cp.int32, 32, 1, 'striped')
block_store = coop.block.store(cp.int32, 32, 1, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def adjacent_difference(input, output):
  shared = cuda.shared.array(32, dtype=cp.int32)

  block_load(input, shared)

  if not threadIdx.x == 0:
    block_store(input,
      shared[threadIdx.x] - shared[threadIdx.x - 1])
```

```
block_load  = coop.block.load(cp.int32, 32, 1, 'striped')
block_store = coop.block.store(cp.int32, 32, 1, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def adjacent_difference(input, output):
  shared = cuda.shared.array(32, dtype=cp.int32)

  block_load(input, shared)

  if not threadIdx.x == 0:
    block_store(input,
      shared[threadIdx.x] - shared[threadIdx.x - 1])
```

```
block_load  = coop.block.load(cp.int32, 32, 1, 'striped')
block_store = coop.block.store(cp.int32, 32, 1, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def adjacent_difference(input, output):
  shared = cuda.shared.array(32, dtype=cp.int32)

  block_load(input, shared)

  if not threadIdx.x == 0:
    block_store(input,
      shared[threadIdx.x] - shared[threadIdx.x - 1])
```

```
block_load  = coop.block.load(cp.int32, 32, 1, 'striped')
block_store = coop.block.store(cp.int32, 32, 1, 'striped')

@cuda.jit(link=block_load.files + block_store.files)
def adjacent_difference(input, output):
  shared = cuda.shared.array(32, dtype=cp.int32)

  block_load(input, shared)

  cuda.syncthreads()

  if not threadIdx.x == 0:
    block_store(input,
      shared[threadIdx.x] - shared[threadIdx.x - 1])
```

What We Learned Today

- CuPy: CUDA-accelerated NumPy, linear algebra, & scientific algos.
- cuDF: CUDA-accelerated Pandas, data frames, & data science.
- Numba CUDA: Python-native CUDA kernel authoring.
- cuda.cccl: CUDA algorithm building blocks.
- Benchmarking and profiling.
- CUDA model and performance fundamentals.

GTC CUDA Developer Track
nvda.ws/4dVjYXs

Accelerated Computing Hub
nvda.ws/43tdsU6

GENERAL CUDA

S72571 What's CUDA All About Anyways?
S72897 How To Write A CUDA Program
S72527 Debugging & Optimizing CUDA with Intelligent Developer Tools

CUDA PYTHON

S72450 Accelerated Python: Tour of Community & Ecosystem
S72448 The CUDA Python Developer's Toolbox
S72449 1001 Ways to Write CUDA Kernels in Python
S74639 Tensor Core Programming in Python with CUTLASS 4.0

CUDA C++

S72574 Building CUDA Software at the Speed-of-Light
S72572 The CUDA C++ Developer's Toolbox
S72575 How You Should Write a CUDA C++ Kernel

MULTI-GPU PROGRAMMING

S72576 Getting Started with Multi-GPU Scaling: Distributed Libraries
S72579 Going Deeper with Multi-GPU Scaling: Task-based Runtimes
S72578 Advanced Multi-GPU Scaling: Communication Libraries

PERFORMANCE OPTIMIZATION

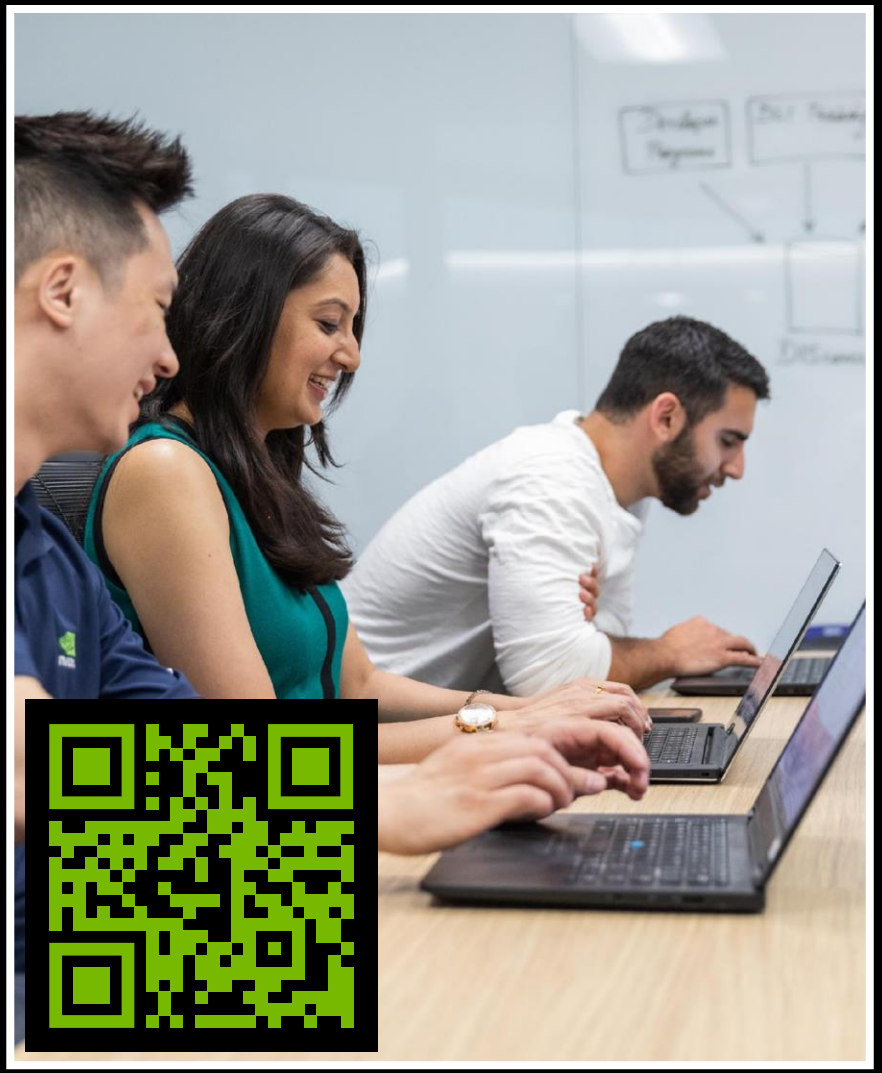
S72683 CUDA Techniques to Maximize Memory Bandwidth & Hide Latency
S72685 CUDA Techniques to Maximize Compute & Instruction Throughput
S72686 CUDA Techniques to Maximize Concurrency & System Utilization
S72687 Get the Most Performance From Grace Hopper

NVIDIA DLI Courses

- Fundamentals of Accelerated Computing with CUDA Python
- Fundamental of Accelerated Computing with Modern CUDA C++
- Fundamentals of Accelerated Data Science
- Accelerate Data Science Workflows with Zero Code Changes
- Optimizing CUDA Machine Learning Codes With Nsight Profiling Tools

Scan the QR code to access the full course list and redeem one free training!

nvda.ws/3ZLPyAL

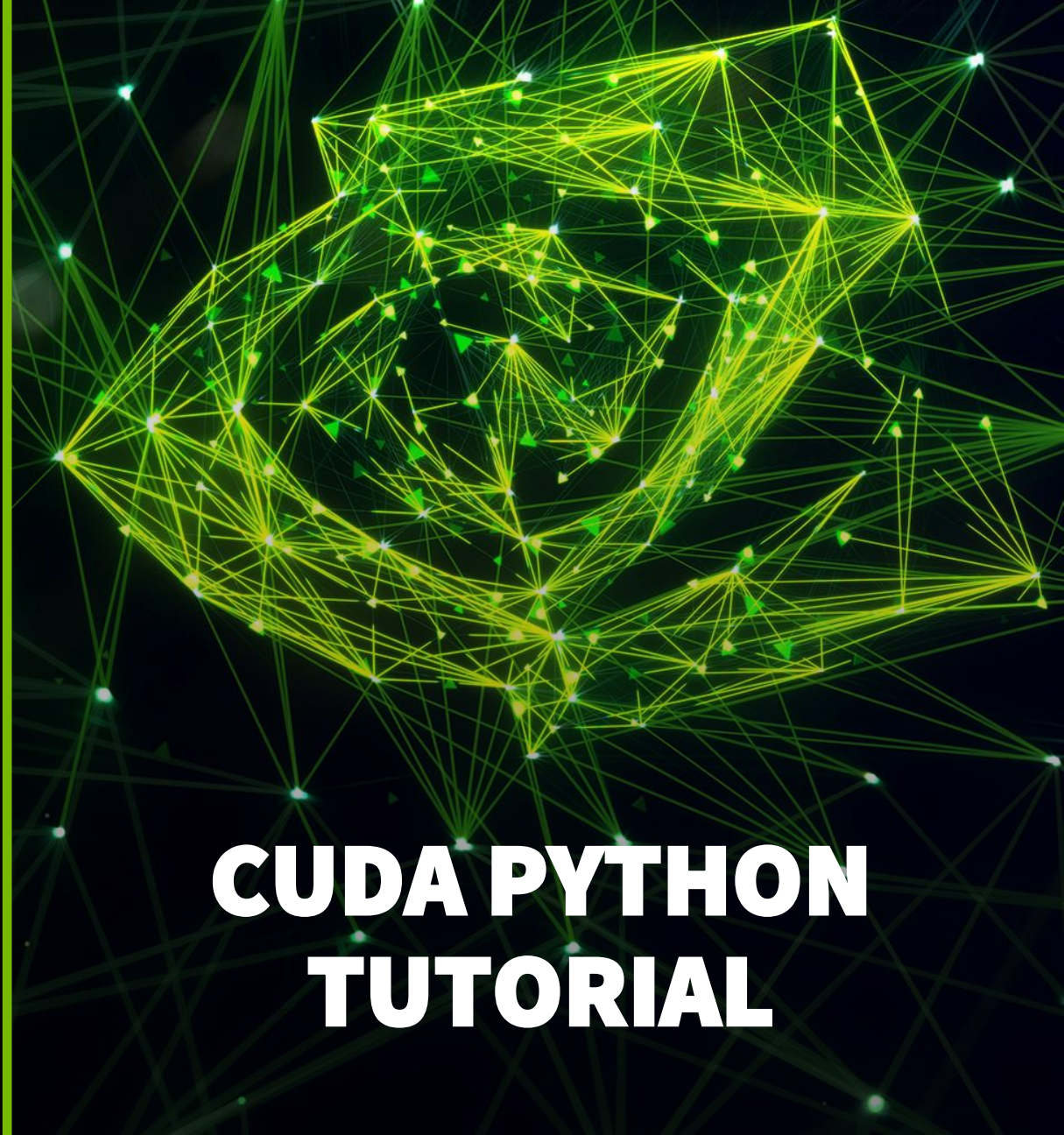


GTC CUDA
Developer Track
nvda.ws/4dVjYXs

Accelerated
Computing Hub
nvda.ws/43tdsU6

NVIDIA
DLI Courses
nvda.ws/3ZLPyAL

ADSP
Podcast
adspthepodcast.com



CUDA PYTHON TUTORIAL