

Sign up for the lab platform:

nvda.ws/XXXXXXX

Join the Zoom:

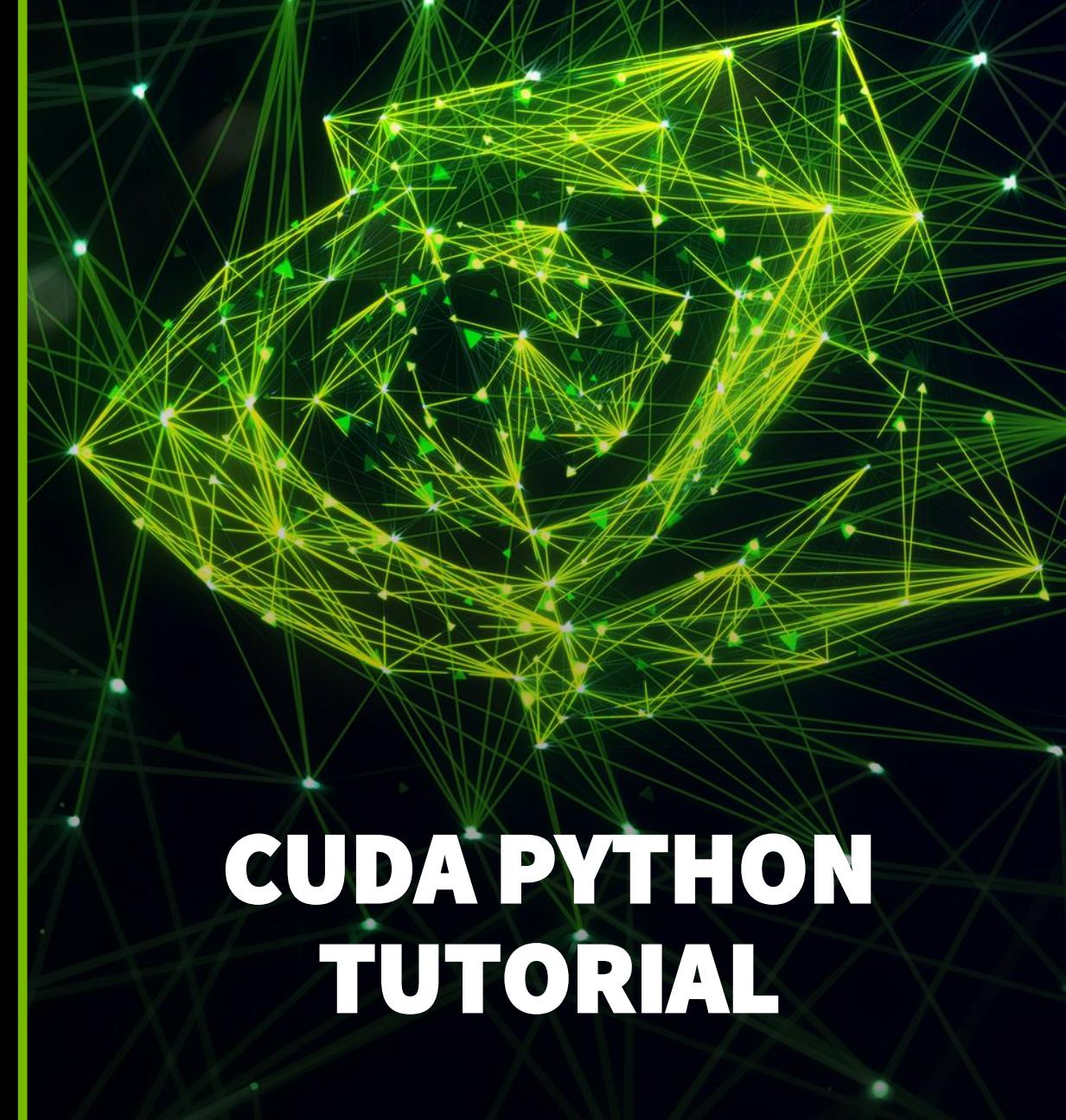
nvda.ws/XXXXXXX

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

nvda.ws/XXXXXXX

WIFI: SSIDSSIDSSIDSSID

PW: PASSWORDPASSWORD





Bryce Adelstein Lelbach

Principal Engineer
Vanguard Programming



Bryce Adelstein Lelbach

Principal Engineer
Vanguard Programming



Andy Terrel

Product Manager
CUDA Python



Bryce Adelstein Lelbach

Principal Engineer
Vanguard Programming



Andy Terrel

Product Manager
CUDA Python



Leo Fang

Senior Software Engineer
CUDA Python



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



Bryce Adelstein Lelbach

Principal Engineer
Vanguard Programming



Andy Terrel

Product Manager
CUDA Python



Leo Fang

Senior Software Engineer
CUDA Python



Mike O'Keefe

Senior Solutions Architect
Higher Education & Research



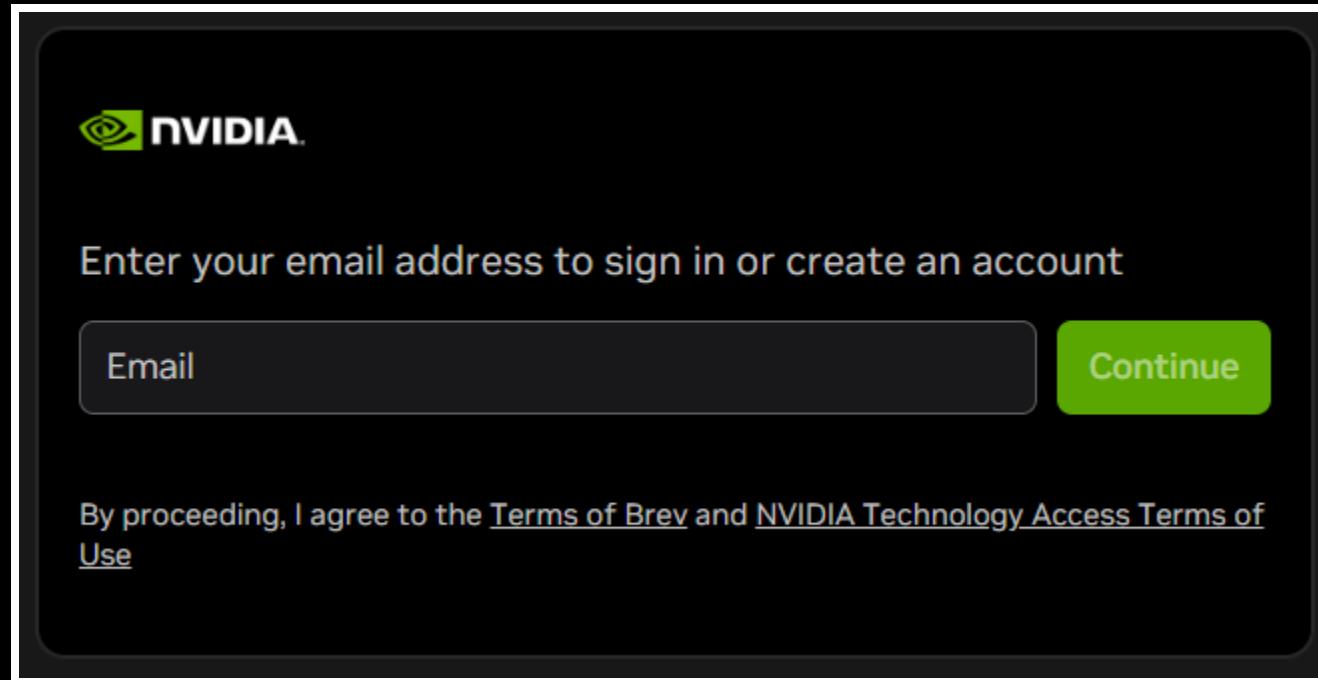
Eliot Eshelman

Senior Account Manager

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

Recommended browser: Chrome (28+).

Javascript and WebRTC must be enabled.



Create Your Account

Email
 ×

Enter your email address.

Password
 (8)

Confirm password
 (8)

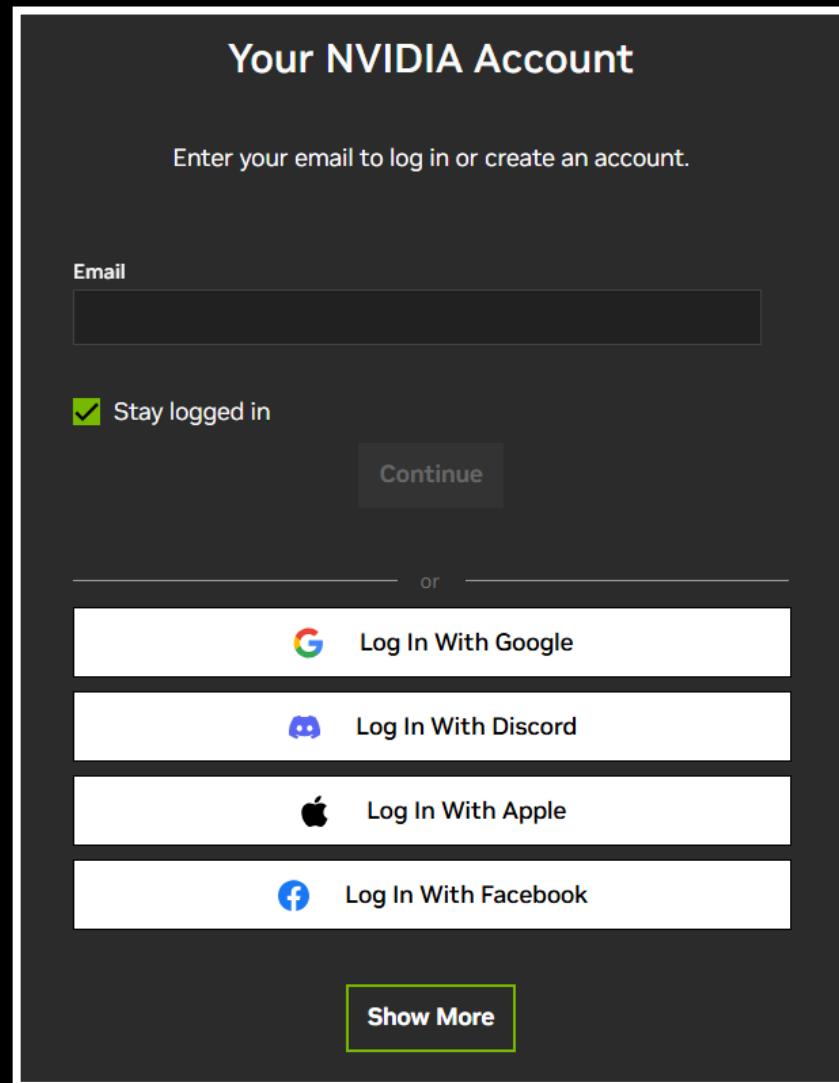
Stay logged in [Log In With Security Device >](#)

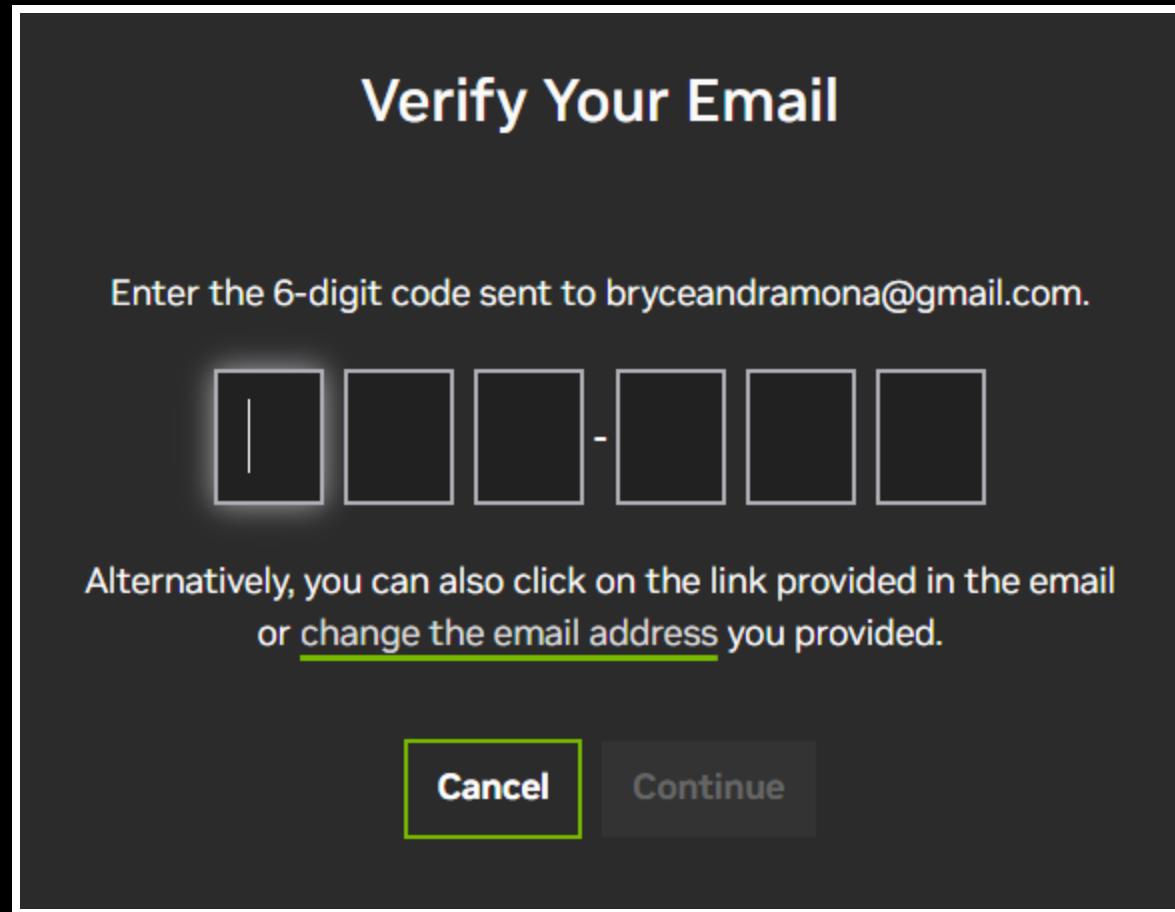
I am human 
Privacy - Terms

By proceeding, I agree to the [NVIDIA Account Terms Of Use](#) and [Privacy Policy](#)

[Create Account](#)

[More Signup Options](#)





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!



Create an NVIDIA Cloud Account

To use our services, your NVIDIA Account needs to be associated with an NVIDIA Cloud Account. Provide your cloud account name to set up.

Provide a Cloud Account Name

Enter a clear, memorable name that you'll easily recognize later. Typically this is your organization name, team name, or project identifier.

Account Name *

Examples: OrganizationName, Engineering-Team, Project-ABC

✗ Name is required.

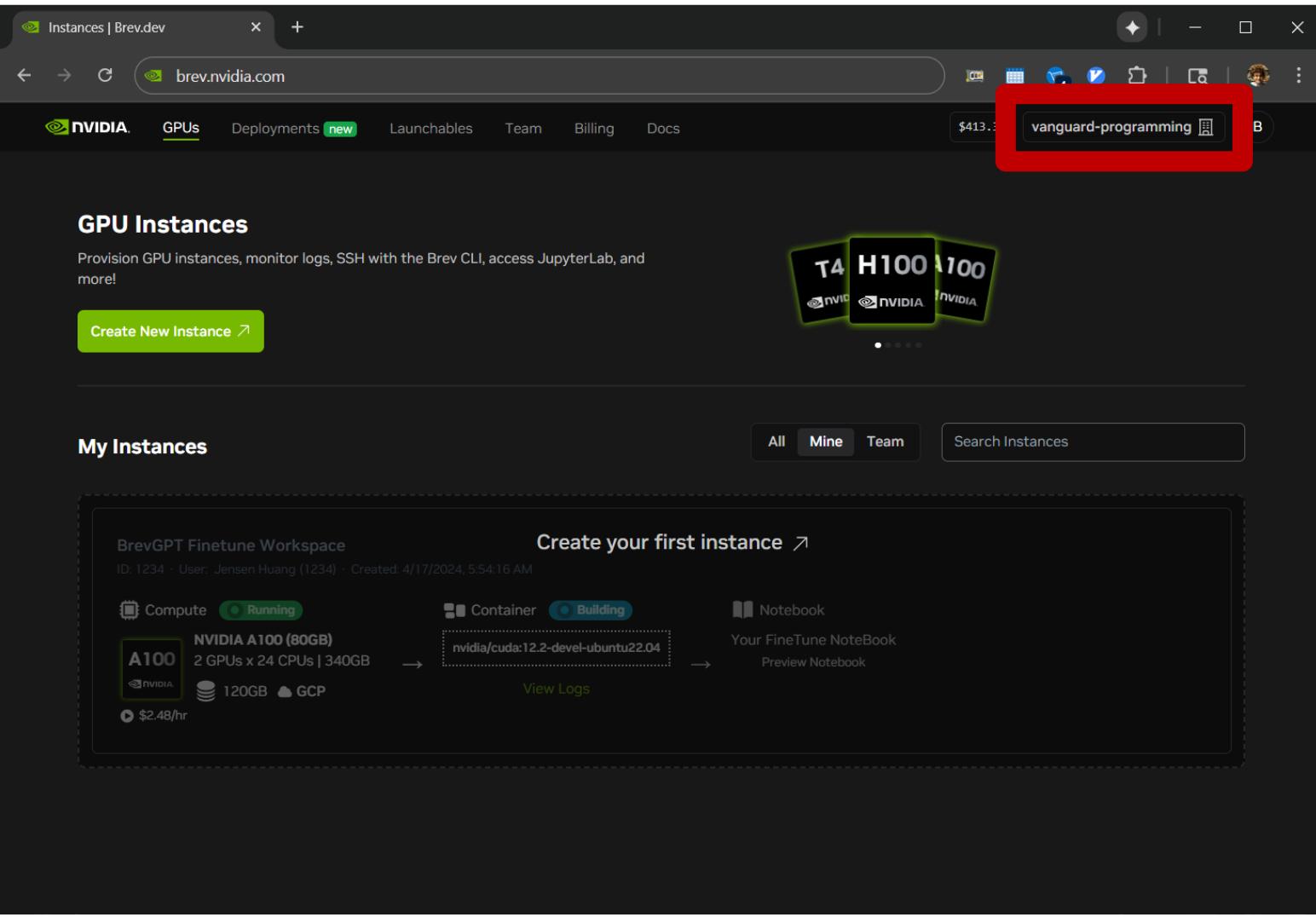
[Create NVIDIA Cloud Account](#)

[Learn more about NVIDIA Cloud Account](#)

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

The screenshot shows the Brev.dev GPU Instances dashboard. At the top, there's a navigation bar with links for NVIDIA, GPUs, Deployments (new), Launchables, Team, Billing, Docs, and a user account section showing \$413.36. Below the navigation is a section titled "GPU Instances" with a sub-section "BrevGPT Finetune Workspace". This workspace details are: ID: 1234 · User: Jensen Huang (1234) · Created: 4/17/2024, 5:54:16 AM. It shows a Compute instance running an NVIDIA A100 (80GB) with 2 GPUs x 24 CPUs | 340GB, 120GB GCP storage, and a price of \$2.48/hr. There are tabs for Compute (Running), Container (Building), and Notebook. The Container tab shows a build status for "nvidia/cuda:12.2-devel-ubuntu22.04". Below the workspace details is a large red text overlay: "To return to the platform: brev.nvidia.com".

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



The screenshot shows the Brev.dev GPU Instances dashboard. At the top, there's a navigation bar with links for NVIDIA, GPUs, Deployments (new), Launchables, Team, Billing, and Docs. A search bar contains the text "vanguard-programming". Below the navigation, there's a section titled "GPU Instances" with a sub-section "My Instances". A green button labeled "Create New Instance" is visible. On the right side, there are icons for T4, H100, and A100 GPUs. The main area displays a workspace named "BrevGPT Finetune Workspace" with an ID of 1234, created by Jensen Huang on 4/17/2024 at 5:54:16 AM. The workspace is currently running an "A100 (80GB)" instance with 2 GPUs, 24 CPUs, and 340GB of memory, costing \$2.48/hr. It also shows a "Container" section for "nvidia/cuda:12.2-devel-ubuntu22.04" which is building, and a "Notebook" section for "Your FineTune NoteBook" which is previewing.

The screenshot shows the NVIDIA Brev Dev interface with a red box highlighting the "Launchables" tab in the navigation bar. The main content area displays the "Create Launchables: GPU Environment Templates" section, featuring a "Create Launchable" button and a 3D rendering of a blue cube representing a launchable environment. Below this are metrics for the organization: Team Champion (blelbach-0), Total Views (2, +2 last 7 days), and Total Deploys (2, +2 last 7 days). The "Organization Launchables" section lists a single entry: "CUDA Python Tutorial" (ID: env-32ukG4DgCaNsDxHQGrSwlDM2rMe, created by blelbach-0). It includes a "View Launchable" link, a "Deploy Now" button, and a "Public" dropdown menu.

The screenshot shows the NVIDIA Brev.dev web interface for managing GPU environment templates. At the top, there's a navigation bar with tabs for Instances, NVIDIA, GPUs, Deployments (new), Launchables (selected), Team, Billing, and Docs. The total cost is listed as \$413.36, and the organization is vanguard-programming.

The main section is titled "Create Launchables: GPU Environment Templates". It explains that Launchables are GPU Environment Templates that can be deployed with one-click. A large green button labeled "Create Launchable" is prominently displayed. To the right is a 3D-style icon of a blue cube representing a GPU environment.

Below this, there are three performance metrics boxes:

- Team Champion:** bleibach-0
- Total Views:** 2 (+2 Last 7 days)
- Total Deploys:** 2 (+2 Last 7 days)

Further down, the "Organization Launchables" section lists a single item: "CUDA Python Tutorial". This entry has a red rectangular highlight around its title and a red box around its "View Launchable" button. The CUDA Python Tutorial entry includes a thumbnail, a link ([https://brev.nvidia.com/launchable/deploy?launchableID=env-32ukG4DgCaNsDxHQ...](https://brev.nvidia.com/launchable/deploy?launchableID=env-32ukG4DgCaNsDxHQGrSwlDM2rMe)), a "Deploy Now" button, and a "Show Configuration" link.

The screenshot shows the NVIDIA Brev.dev web interface. At the top, there are two tabs: "Instances | Brev.dev" and "Console | Brev.dev". Below the tabs, the URL is brev.nvidia.com/launchable/deploy/now?launchableID=env-32ukG4DgCaNsDxHQGnSwID... . The main content area displays a "CUDA Python Tutorial" launchable. It includes a "Compute" section with an "L40S" icon, showing "NVIDIA L40S (48GiB)", "1 GPU x 8 CPUs | 147GiB", and storage details "100GiB CRUSOE". A "Container" section shows a GitHub repository link: "github.com:ial/brev/docker-compose.yml". A "Files" section has a "View Source" link. An "Exposed Ports" section lists "Tunnels: jupyter:8888 | nslight:8080" and "Firewall Rules: 3478". A large green "Deploy Launchable" button is prominently displayed. Below this, a "Content Preview" section indicates the repository will be present in the instance after deployment. The "Repository Info" section shows the GitHub repository "accelerated-computing-hub" by brycelelbach, with 2 stars, 0 forks, and 2 watchers. Finally, the "README.md" section displays the text "Welcome to the NVIDIA Accelerated Computing Hub".

The screenshot shows the NVIDIA Brev Dev interface for deploying a CUDA Python tutorial. The top navigation bar includes tabs for Instances, Console, Deploying GPU, Install Software, Access GPU, and a sidebar with a balance of \$413.36 and a project named vanguard-programming.

CUDA Python Tutorial
\$1.74/hr

Deploying GPU progress: Launchable is now deploying... Go to Instance Page

For additional options, logs, and access.

Compute: NVIDIA L40S (48GiB)
1 GPU x 8 CPUs | 147GiB
100GiB CRUSOE

Container: GitHub Repo: accelerated-computing-hub
View Source

Files: Exposed Ports
Tunnels: jupyter:8888 | nslight:8080
Firewall Rules: 3478

Content Preview:
This repository will be present in the instance after deployment

Repository Info
accelerated-computing-hub by brycelelbach
2 stars 0 forks 2 watchers

README.md
Welcome to the NVIDIA Accelerated Computing Hub

The screenshot shows the Brev.dev web interface for deploying a GPU environment. The main header includes the NVIDIA logo and navigation links for Deploying GPU, Install Software, and Access GPU. A top bar shows a budget of \$413.30 and a session for vanguard-programming. The central area displays the environment 'cuda-python-tutorial-d70581' created on 9/19/2025 at 2:35:40 PM. It shows an NVIDIA L40S (48GiB) configuration with 1 GPU x 8 CPUs | 147GiB, currently syncing and Deploying. Action buttons include Stop (greyed out) and Delete (red). Below the environment details are tabs for Docker Compose YAML (Waiting), Logs, Content, and Access. The 'Logs' tab is active. A section titled 'Using Brev CLI (SSH)' provides instructions for installing the CLI and logging in.

Using Brev CLI (SSH)

Install the CLI

Run this in your terminal

```
curl -fsSL https://raw.githubusercontent.com/brevdev/brev-cli/main/bin/install-latest.sh && brev login
```

Login to your account

Run this in your terminal

```
brev login
```

Open a terminal locally

The screenshot shows the Brev.dev web interface for deploying a GPU. At the top, there are tabs for "Console | Brev.dev" and "cuda-python-tutorial-d70581". The main navigation bar includes "Deploying GPU", "Install Software", "Access GPU", and a user account section with "\$413.30" and "vanguard-programming". A red box highlights the "Stop" and "Delete" buttons for the deployment "cuda-python-tutorial-d70581", which was created on 9/19/2025 at 2:35:40 PM. The deployment status is "Deploying" with 1 GPU x 8 CPUs and 147GiB. Below the deployment details, there are tabs for "Docker Compose YAML" (Waiting) and "Logs", "Content", and "Access".

Using Brev CLI (SSH)

Install the CLI

Run this in your terminal

```
sudo bash -c "$(curl -fsSL https://raw.githubusercontent.com/brevdev/brev-cli/main/bin/install-latest.sh)" && brev login
```

Login to your account

Run this in your terminal

```
brev login
```

Open a terminal locally

The screenshot shows the Brev.dev web interface. At the top, there are two tabs: "Console | Brev.dev" and "cuda-python-tutorial-d70581". The main content area displays the details of a workspace named "cuda-python-tutorial-d70581", which was created on 9/19/2025 at 2:35:40 PM. The workspace configuration is "NVIDIA L40S (48GiB)" with 1 GPU and 8 CPUs. It has 221GiB of memory, is located in us-east1-a, and costs \$1.47/hr. The status is "Running". Below this, there are tabs for "Docker Compose YAML" (highlighted) and "Waiting". To the right are "Stop" and "Delete" buttons, and a "Open Notebook" button. At the bottom, there are three tabs: "Logs", "Content", and "Access".

Using Brev CLI (SSH)

Install the CLI

Run this in your terminal

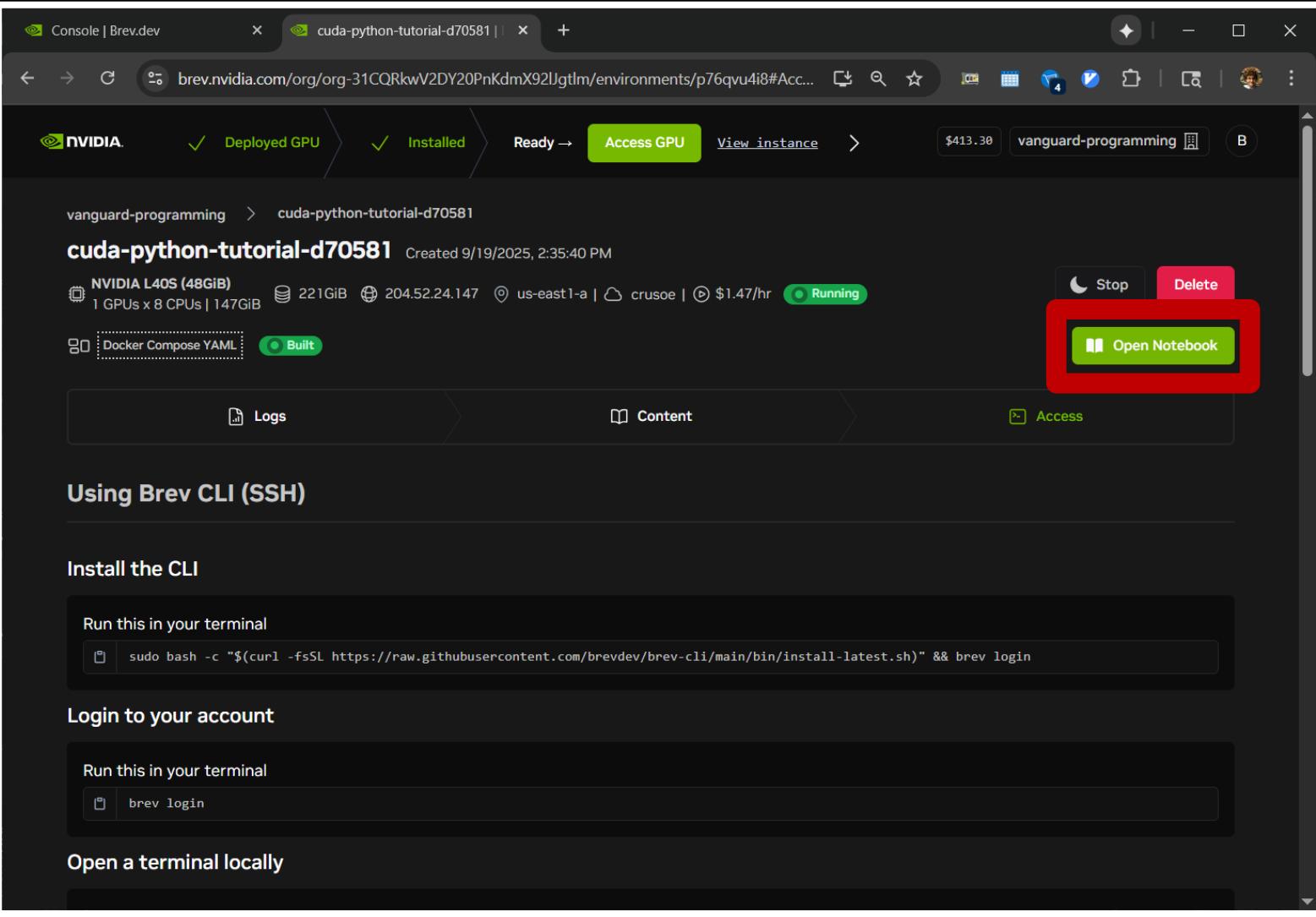
```
curl -fsSL https://raw.githubusercontent.com/brevdev/brev-cli/main/bin/install-latest.sh && brev login
```

Login to your account

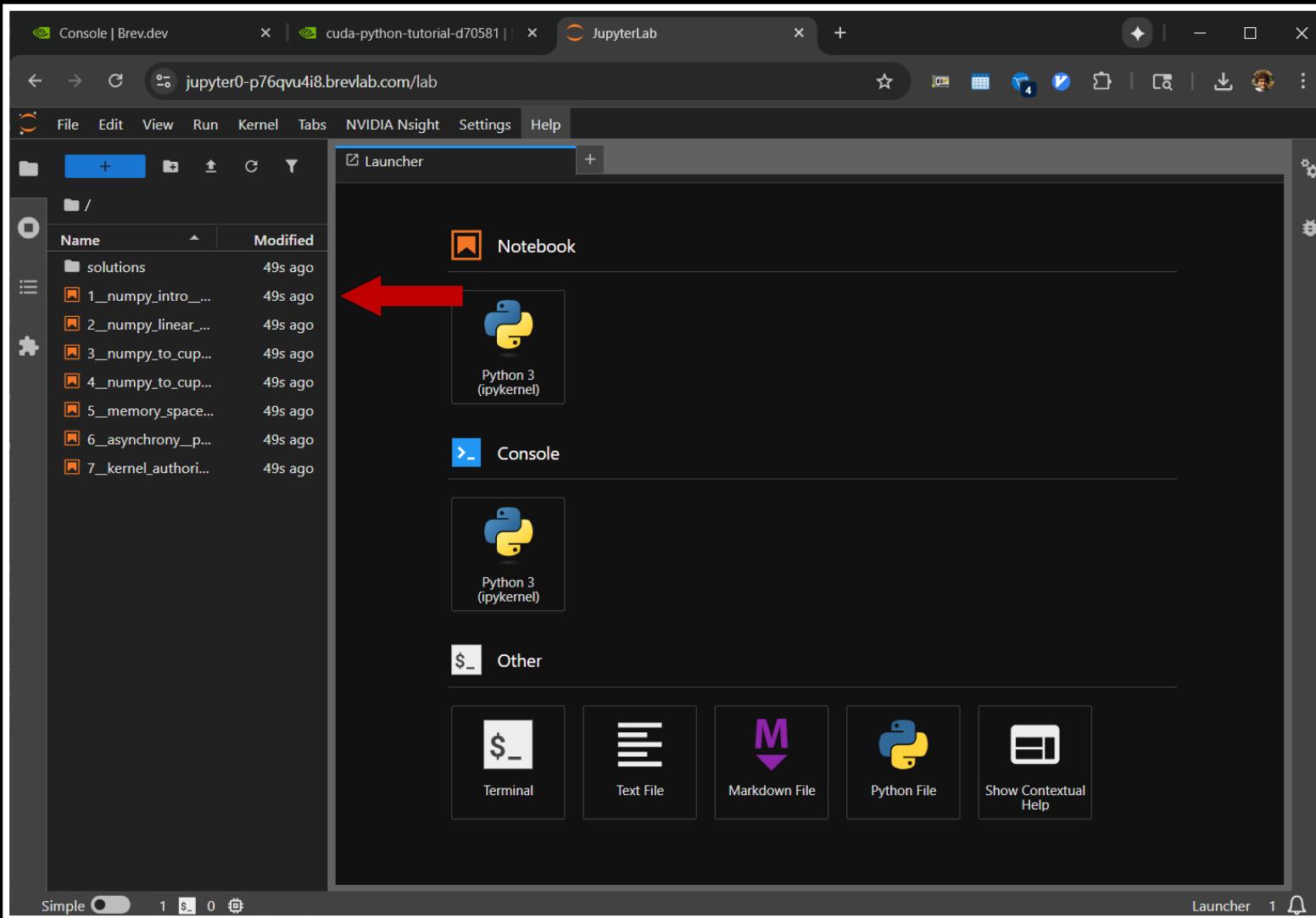
Run this in your terminal

```
brev login
```

Open a terminal locally



The screenshot shows a web browser window with the URL brev.nvidia.com/org/org-31CQRkwV2DY20PnKdmX92Ugtlm/environments/p76qvu4i8#Acc.... The page displays information about a Brev instance named "cuda-python-tutorial-d70581" created on 9/19/2025 at 2:35:40 PM. The instance is running an NVIDIA L40S (48GiB) configuration with 1 GPU and 8 CPUs, and has 147GiB of memory. It is located in the us-east1-a region and costs \$1.47/hr. The status is "Running". A green "Access GPU" button is visible at the top. On the right side, there are "Stop" and "Delete" buttons, and a prominent green "Open Notebook" button which is highlighted with a red rectangular box. Below these buttons are tabs for "Logs", "Content", and "Access". The main content area contains sections for "Using Brev CLI (SSH)", "Install the CLI" (with a terminal command), "Login to your account" (with a terminal command), and "Open a terminal locally".



The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** Shows tabs for "Console | Brev.dev", "cuda-python-tutorial-d70581 |", and "1_numpy_intro_ndarray_basics.ipynb".
- File Menu:** File, Edit, View, Run, Kernel, Tabs, NVIDIA Nsight, Settings, Help.
- Launcher:** Shows a list of notebooks and files in the current directory:
 - solutions (49s ago)
 - 1_numpy_intro_ndarray_basics.ipynb (selected, 49s ago)
 - 2_numpy_linear... (49s ago)
 - 3_numpy_to_cup... (49s ago)
 - 4_numpy_to_cup... (49s ago)
 - 5_memory_space... (49s ago)
 - 6_asynchrony_p... (49s ago)
 - 7_kernel_authori... (49s ago)
- Content Area:** Title: "Exercise - NumPy Intro - ndarray Basics".

Let's practice working with NumPy `ndarray`s. You may find NumPy's reference documentation useful.

[]: `import numpy as np`

TODO: Create the input data array with the numbers 1 to 500_000_000.

[]: `arr = ...`
`arr`

TODO: Calculate how large the array is in GB with `nbytes`. Hint: GB is `1e9`

[]: `arr...`

TODO: How many dimensions does the array have?

[]: `arr...`

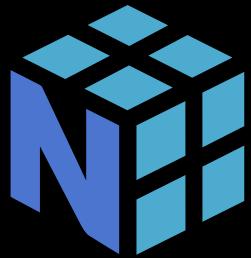
TODO: How many elements does the array have?

[]: `arr...`

TODO: What is the shape of the array?

[]: `arr...`

TODO: Create a new array with 5_000_000 elements containing equally spaced values between 0 to
- Bottom Status Bar:** Simple, Python 3 (ipykernel) | Idle, Mode: Command, Ln 1, Col 1, 1_numpy_intro_ndarray_basics.ipynb, 1, 1



NumPy: Array Programming

```
import numpy as np  
  
# Riemann sum of  $\int_0^1 x^2 dx$   
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!

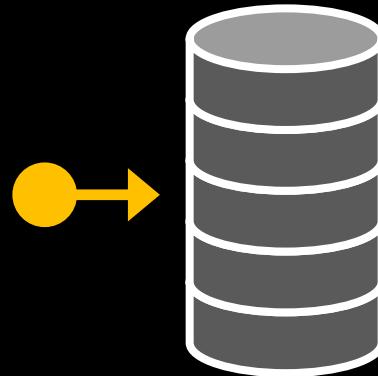
NumPy specifies:

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

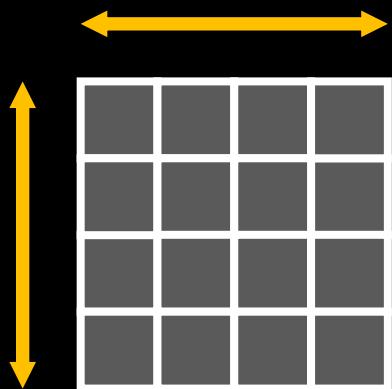
ndarray



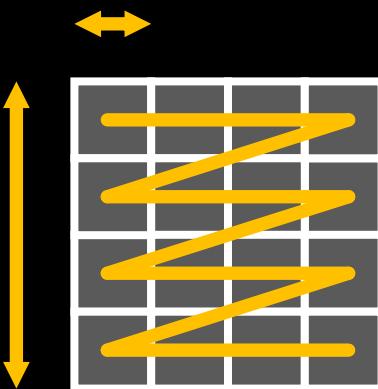
dtype: the data type of all elements of this array.
Ex: int32



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)

Axis 0 →

↓
Axis 1

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)

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!

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

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Logical

(0,0)
(0,1)
(0,2)
(0,3)
(1,0)
(1,1)
(1,2)
(1,3)
(2,0)
(2,1)
(2,2)
(2,3)
(3,0)
(3,1)
(3,2)
(3,3)

Value

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

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Logical

(0,0)
(1,0)
(2,0)
(3,0)
(0,1)
(1,1)
(2,1)
(3,1)
(0,2)
(1,2)
(2,2)
(3,2)
(0,3)
(1,3)
(2,3)
(3,3)

Value

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

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

Views of the original
data, *not a copy*

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

$X[1, 2]$

0D (Scalar)

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

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

$X[1, 2]$

0D (Scalar)

$X[1]$

1D



Views of the original data, *not a copy*

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

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

$X[1, 2]$

0D (Scalar)

$X[:, 1]$

1D



Views of the original data, *not a copy*

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

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

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

`X[1, 2]`

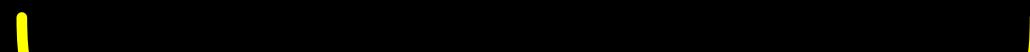
0D (Scalar)

`X[:, 1]`

1D

`X[2:, 2:]`

2D



Views of the original data, *not a copy*

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

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

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

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

`X[1, 2]`

0D (Scalar)

`X[:, 1]`

1D

`X[2:, 2:]`

2D

`A[::-2, ::2]`

2D

Views of the original data, *not a copy*

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

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

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

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

`X[1, 2]`

0D (Scalar)

`X[:, 1]`

1D

`X[2:, 2:]`

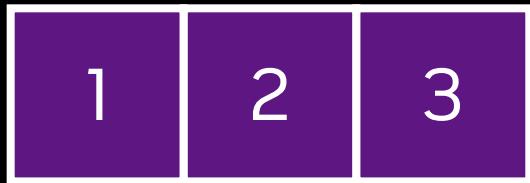
2D

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

2D

`start:stop:step`

(3)



(1)



(3)

1	2	3
---	---	---

+

(1)

2	2	2
---	---	---



Stretch

(3)

1	2	3
---	---	---

(1)

2	2	2
---	---	---

+

(3)

3	4	5
---	---	---



Stretch

(3, 3)

2	2	2
2	2	2
2	2	2

(3)

1	2	3
---	---	---

+

(3, 3)

2	2	2
2	2	2
2	2	2

(3)

1	2	3
1	2	3
1	2	3

+



Stretch

(3, 3)

2	2	2
2	2	2
2	2	2

+

(3)

1	2	3
1	2	3
1	2	3

=

(3, 3)

3	4	5
3	4	5
3	4	5

(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)
```

1	2	3
---	---	---

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

4	2
5	3

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

1	1
1	1

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)
```

1	2	3
---	---	---

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

4	2
5	3

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

1	1
1	1

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

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

4+1	2+1
5+1	3+1

```
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)
```

$$\begin{array}{|c|c|c|} \hline \sqrt{1} & \sqrt{2} & \sqrt{3} \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline e^1 & e^2 & e^3 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline \sqrt{e^1} & \sqrt{e^2} & \sqrt{e^3} \\ \hline \end{array}$$

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)
```

$$\begin{array}{|c|c|c|} \hline \sqrt{1} & \sqrt{2} & \sqrt{3} \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline e^1 & e^2 & e^3 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|} \hline \sqrt{e^1} & \sqrt{e^2} & \sqrt{e^3} \\ \hline \end{array}$$

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

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

$$\begin{array}{|c|c|} \hline 4 & 2 \\ \hline 5 & 3 \\ \hline \end{array} \rightarrow 14$$

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

$$\begin{array}{|c|c|} \hline 4 & 2 \\ \hline 5 & 3 \\ \hline \end{array} \rightarrow [6 \quad 8]$$

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

$$\begin{array}{|c|c|} \hline 4 & 2 \\ \hline 5 & 3 \\ \hline \end{array} \rightarrow \begin{array}{|c|} \hline 6 \\ \hline 8 \\ \hline \end{array}$$

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 \pm 8.76 ms per loop
(mean \pm std. dev. of 7 runs, 1 loop each)
2.63 ms \pm 98.7 μ s per loop
(mean \pm 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.savetxt`, ...)

Random Numbers & Sampling

(`np.random.rand`, `np.random.normal`, ...)

Shape Manipulation

(`np.transpose`, `np.vstack`, ...)

Math / Logic Functions

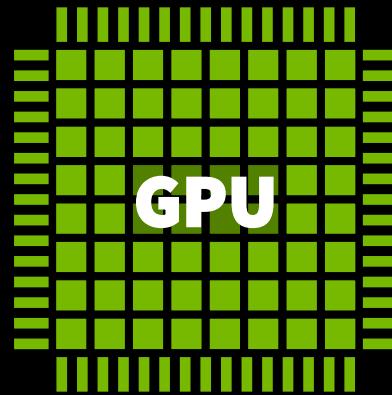
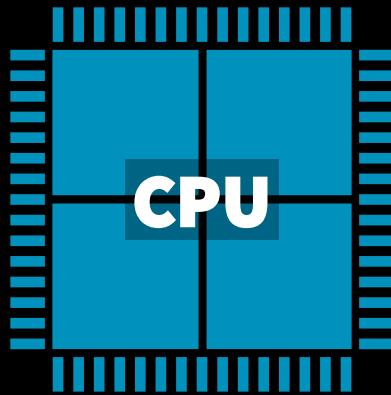
(`np.exp`, `np.sin`, `np.isnan`, ...)

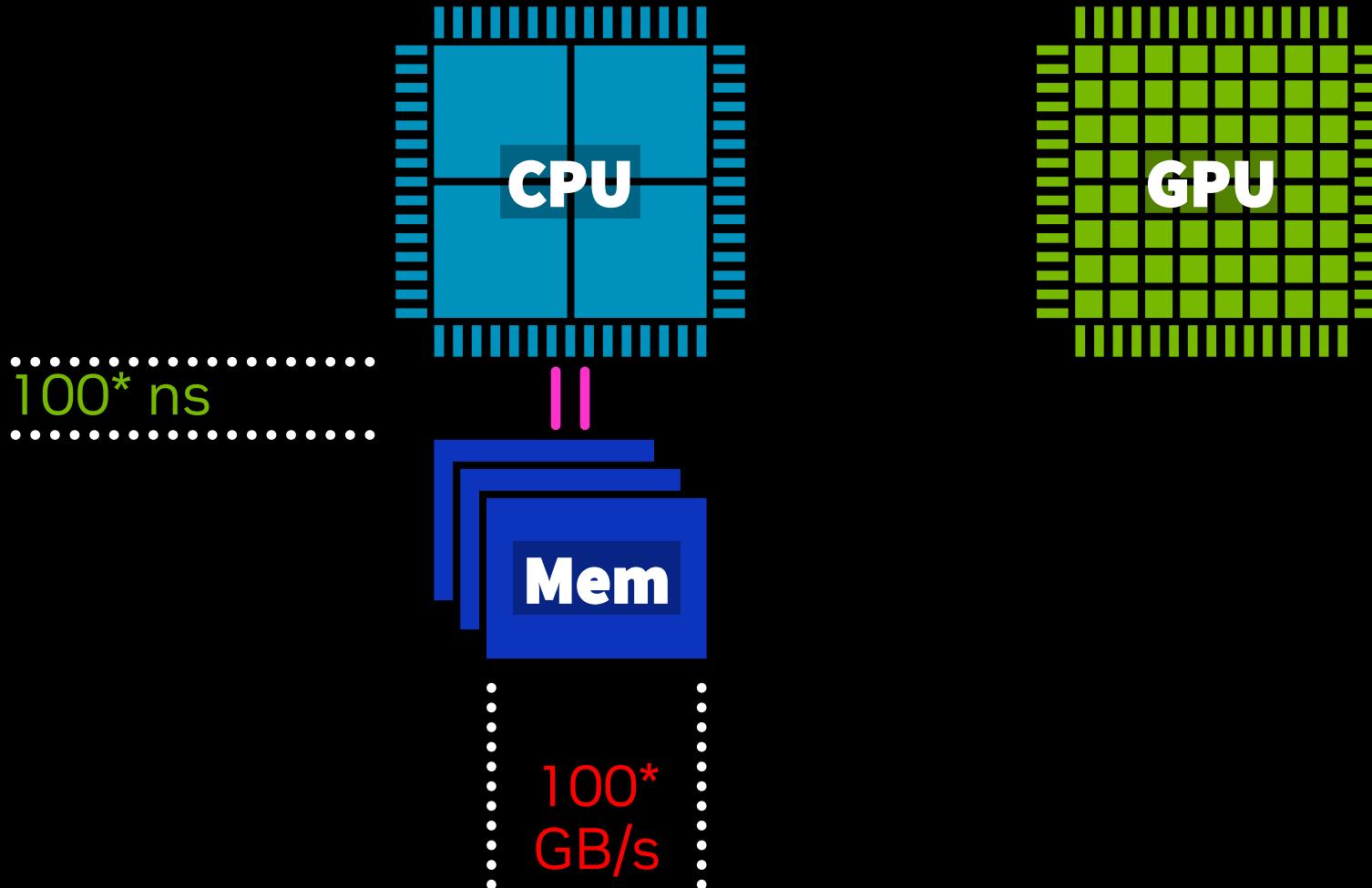
...and many more categories
(and SciPy has even more)

Exercise 2

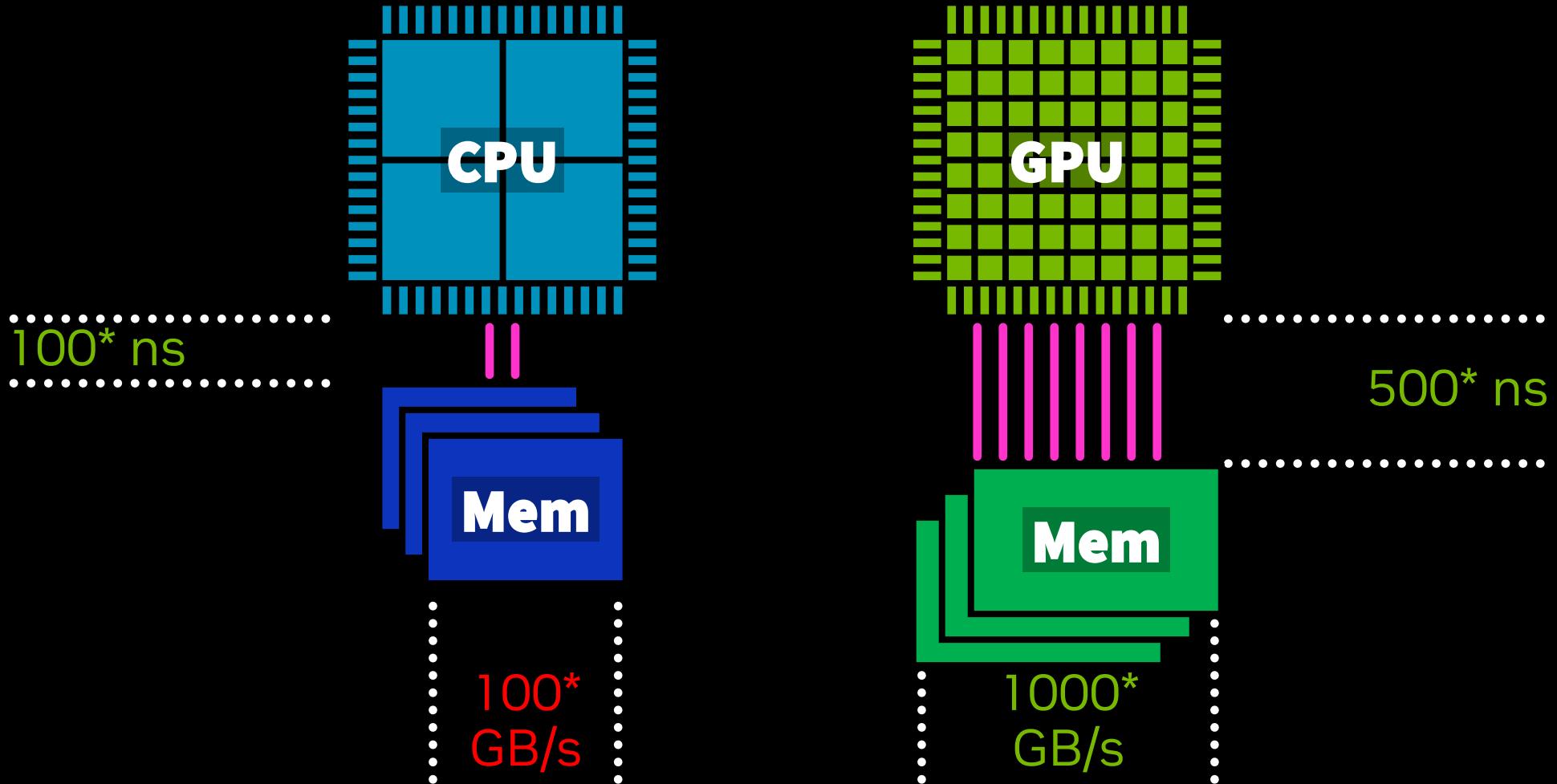
NumPy Linear Algebra

SVD Reconstruction





Numbers are made up and
for expository purposes only.

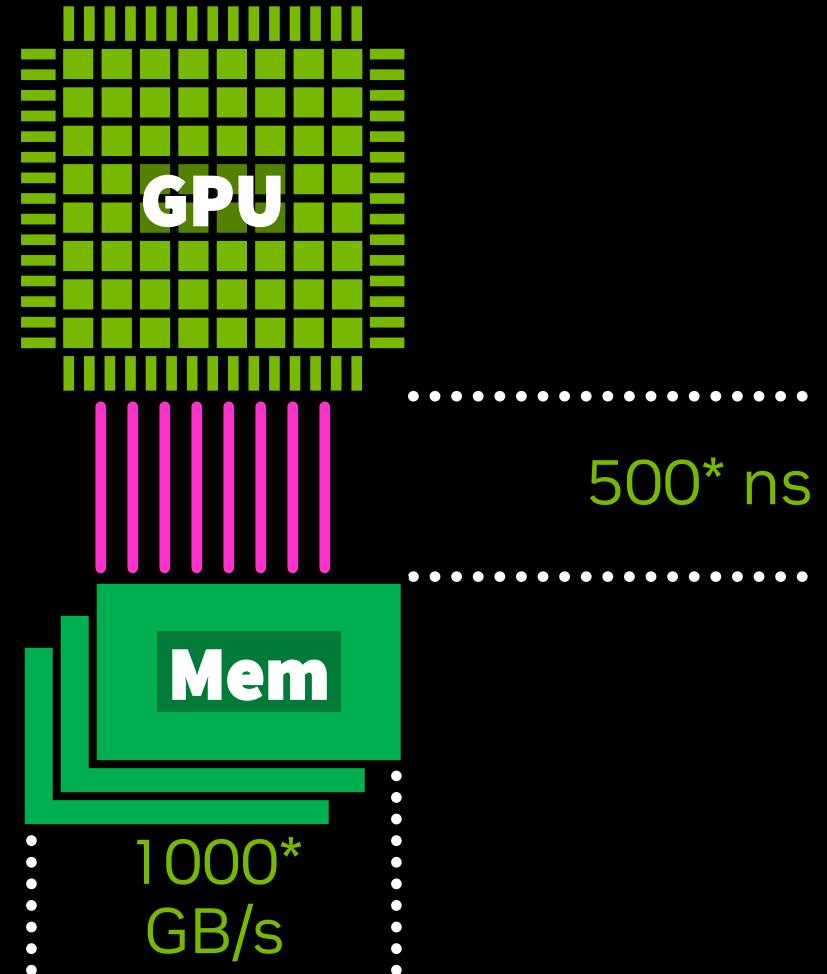


Numbers are made up and
for expository purposes only.

You need a sufficient workload to benefit from GPUs.

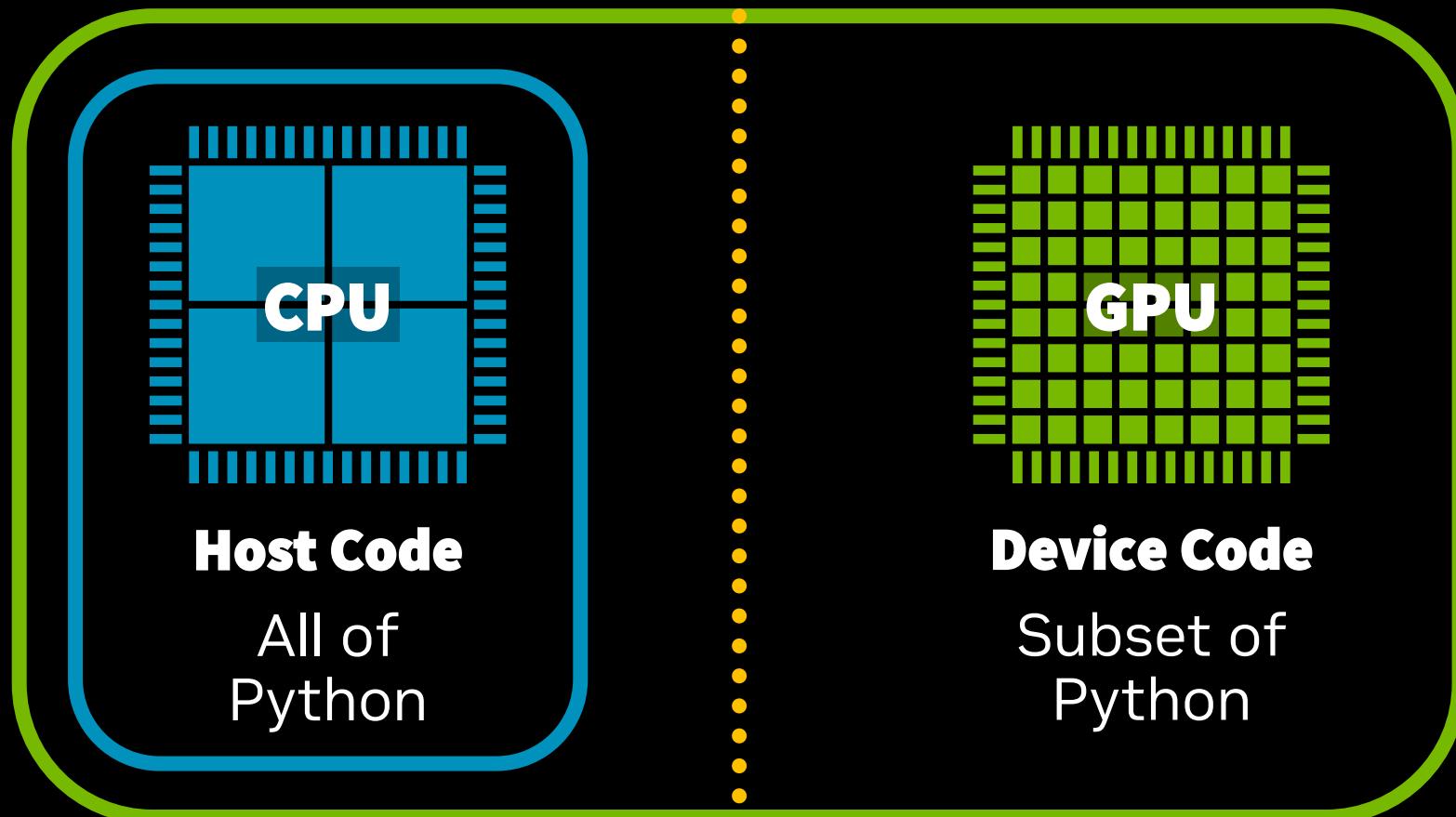
$O(N)$ compute => GBs of data.

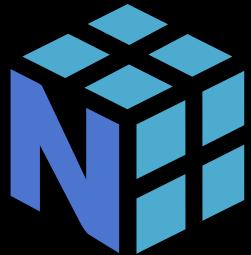
More than $O(N)$ compute => less data.



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

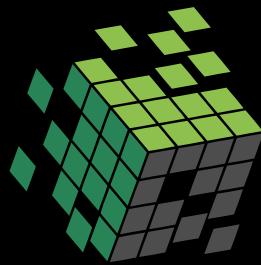
```
import numpy as np  
  
# Riemann sum of  $\int_0^1 x^2 dx$   
x = np.linspace(0, 1, 100)  
y = x**2  
dx = x[1] - x[0]  
I = np.sum(y) * dx
```

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 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)

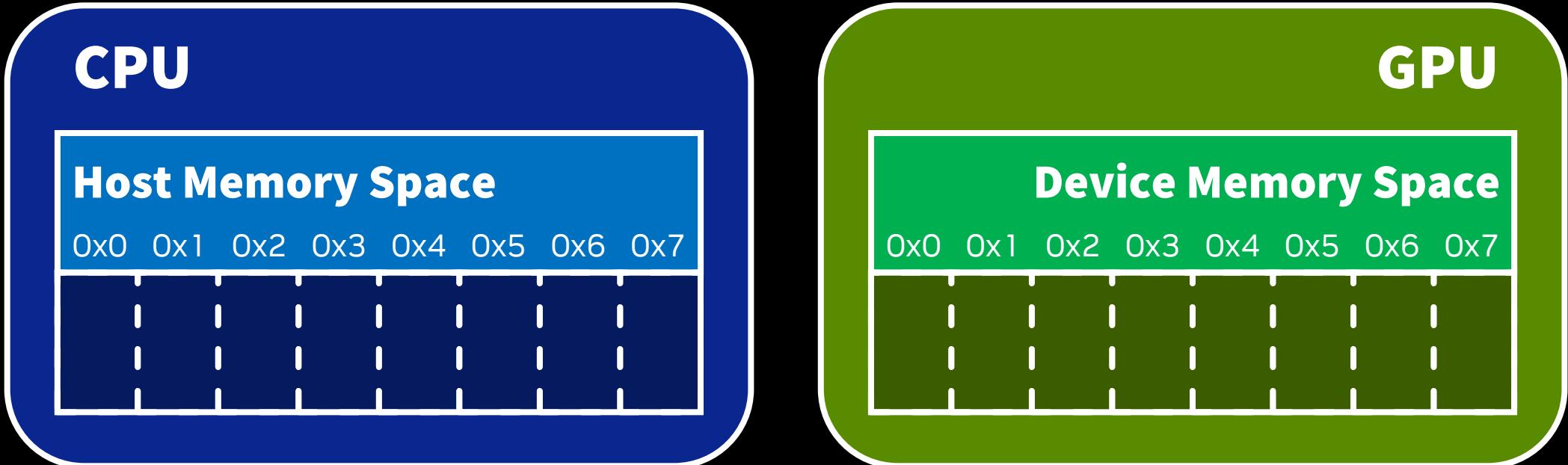
%%timeit
for i in range(a_gpu.shape[0]):
    a_gpu[i] = i

## 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.

CPU

Host Memory Space

0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7

x_host

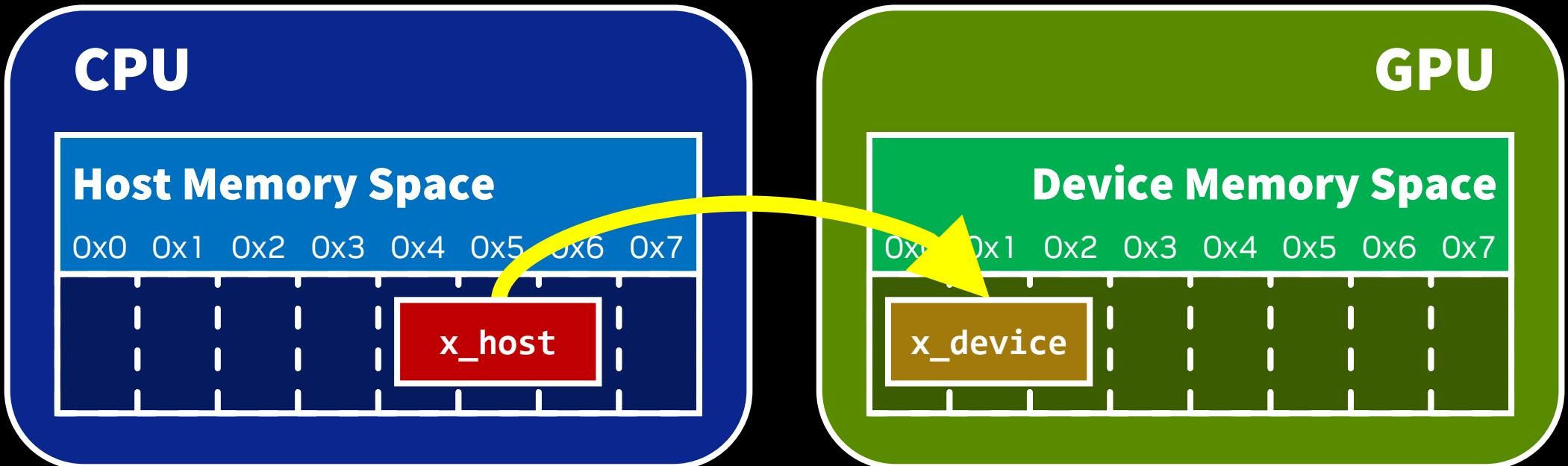
GPU

Device Memory Space

0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7

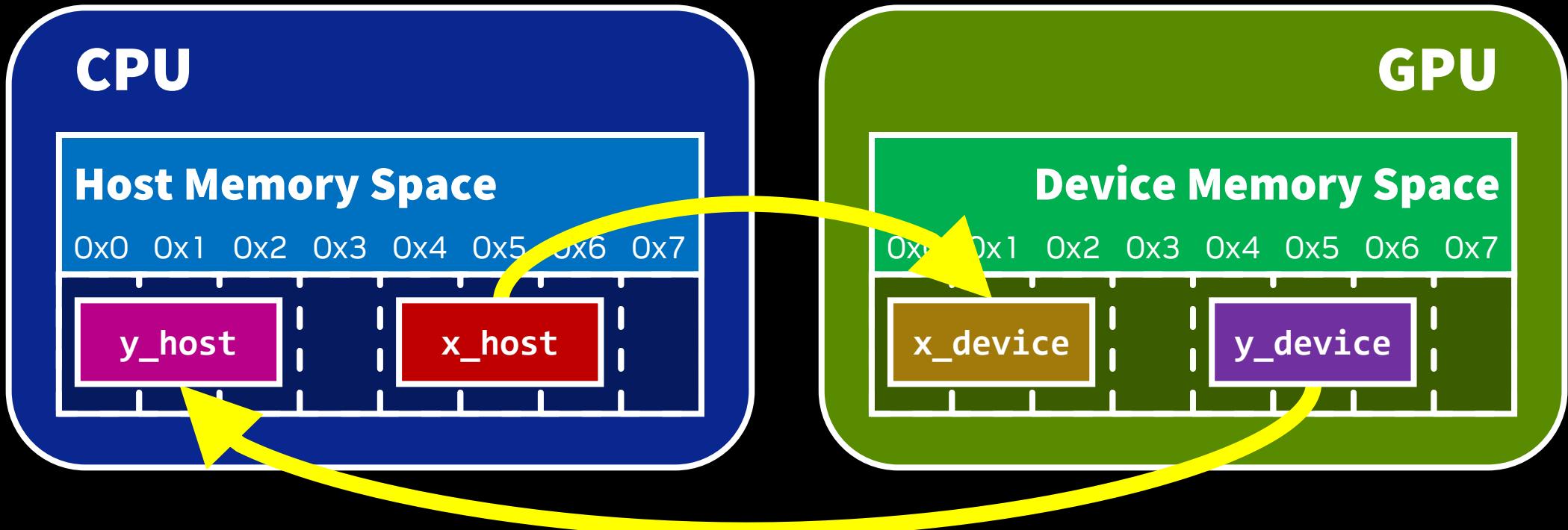
How do we transfer data between host and device?

```
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.
```

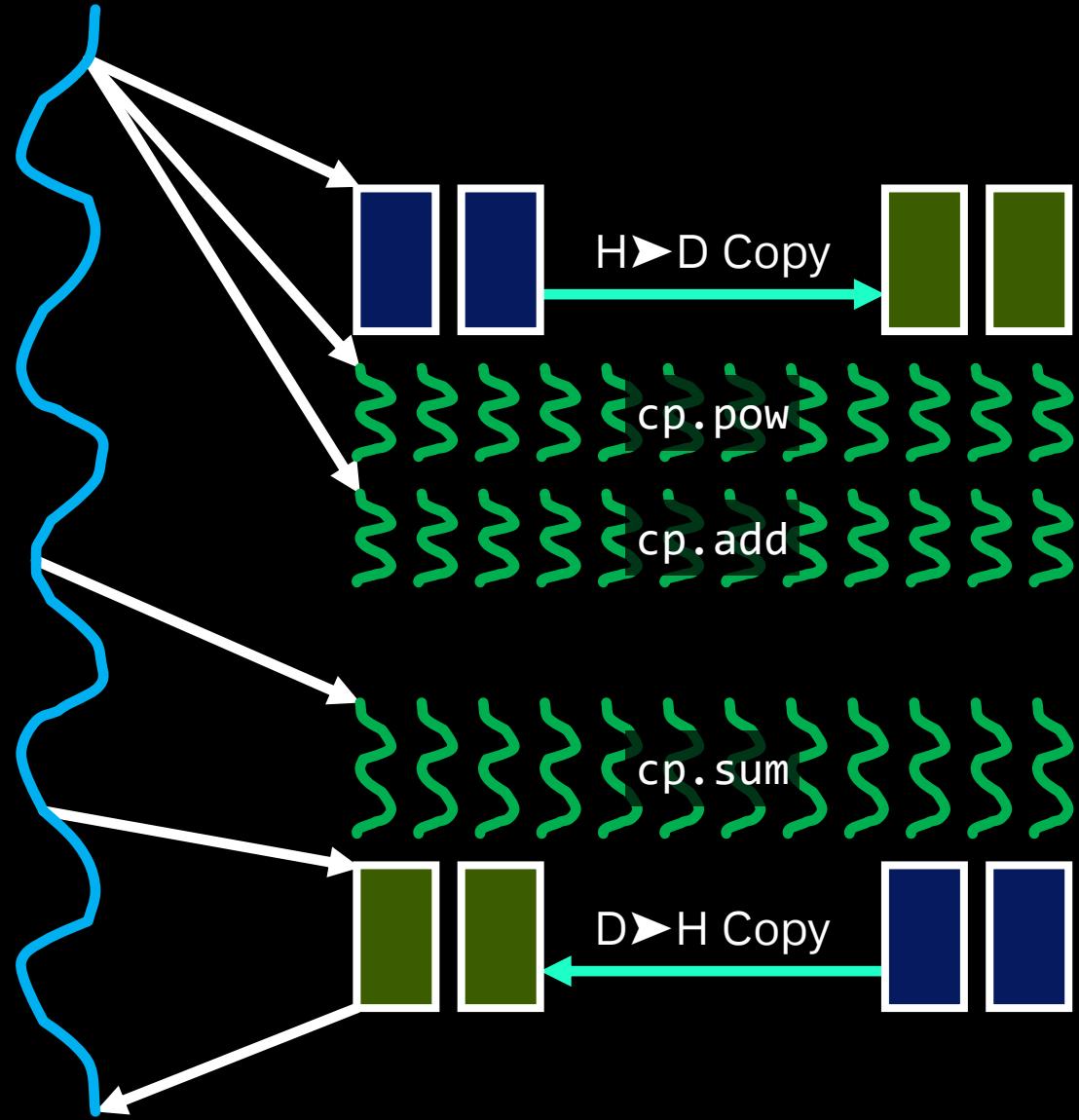
```
C2[0, 0] # Indexing doesn't transfer.
C2[0, :] # Slicing doesn't transfer.
C2[0, 0] = 7 # Scalar assigns transfer w/o sync.
```

Exercise 5

Memory Spaces

Power Iteration

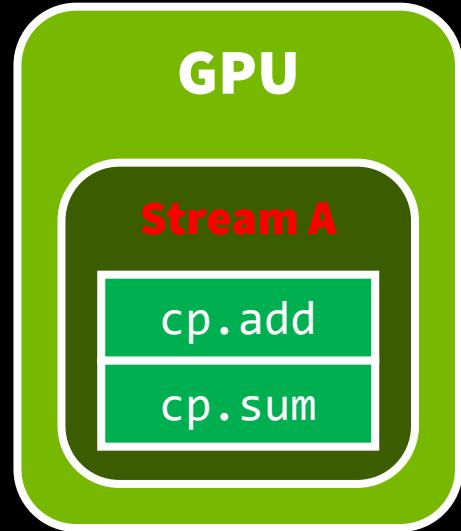
```
A = cp.asarray(...)  
A = A + A**2  
do_cpu_stuff()  
x = A.sum(axis=1)  
do_cpu_other_stuff()  
x_host = cp.asnumpy(x)
```



```
A = cp.cuda.Stream(non_blocking=True)
```

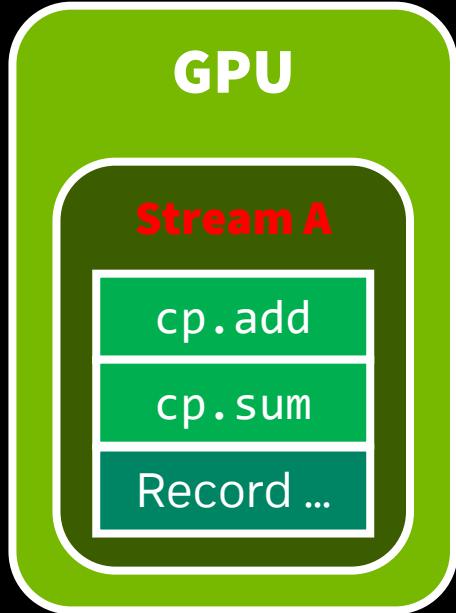
```
with A:
```

```
    cp.add(...)  
    cp.sum(...)
```



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

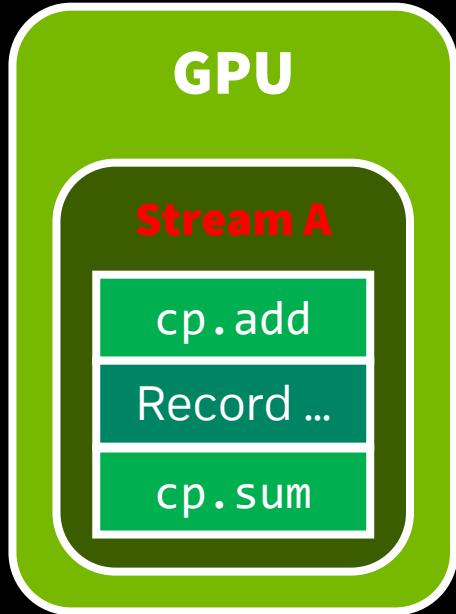
```
A = cp.cuda.Stream(non_blocking=True)  
with A:  
    cp.add(...)  
    cp.sum(...)  
A.synchronize() # Host waits.
```



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(...)
```

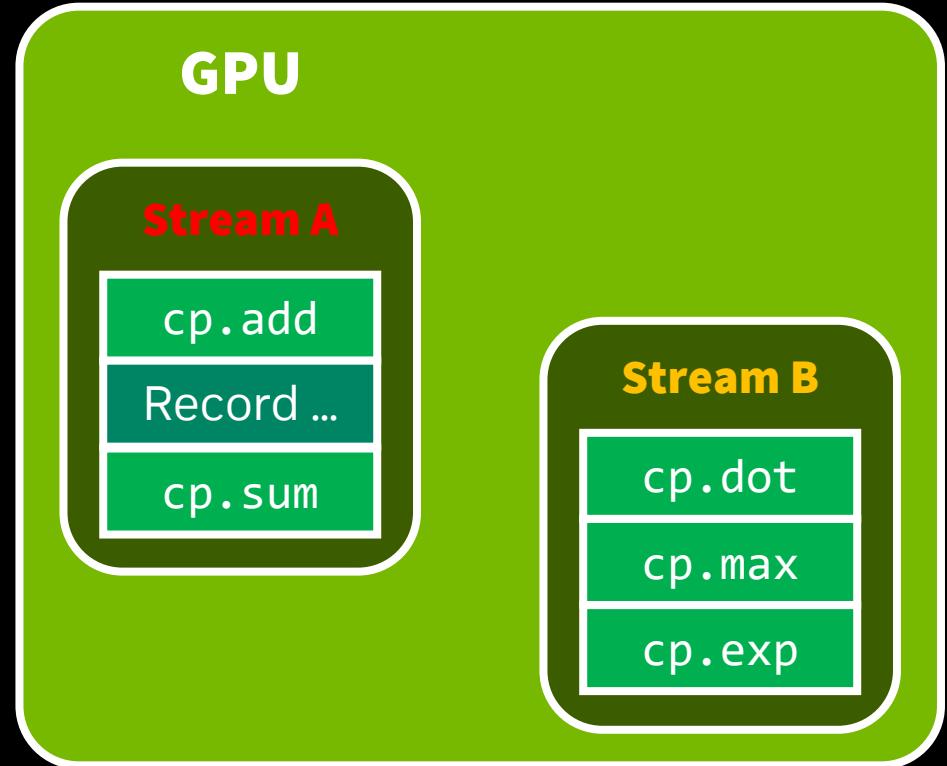


A stream is a chain of device work that is executed in order.

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(...)
```

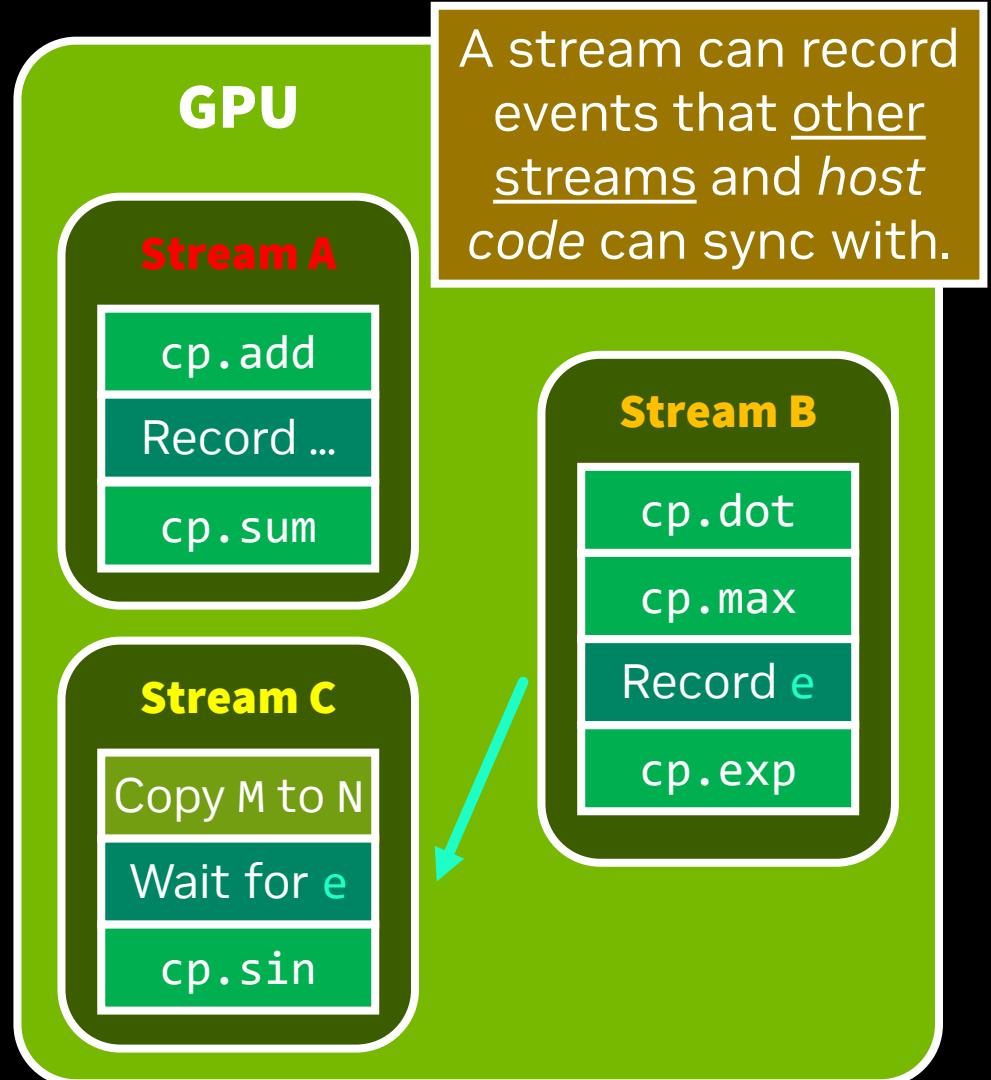


```
A = cp.cuda.Stream(non_blocking=True)
with A:
    cp.add(...)
A.synchronize() # Host waits.
cp.sum(...)
```

```
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.
```

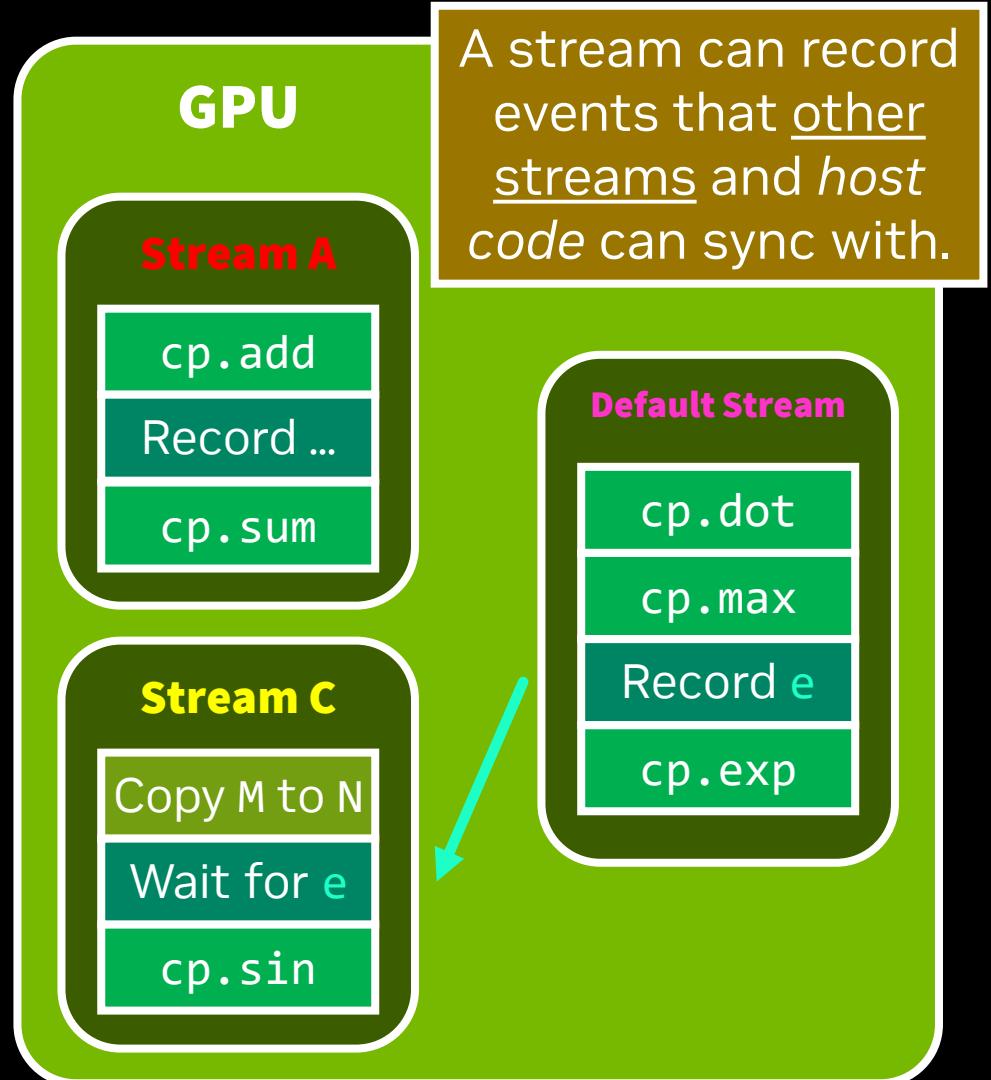


```
A = cp.cuda.Stream(non_blocking=True)
with A:
    cp.add(...)
    A.synchronize() # Host waits.
    cp.sum(...)
```

```
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.
```

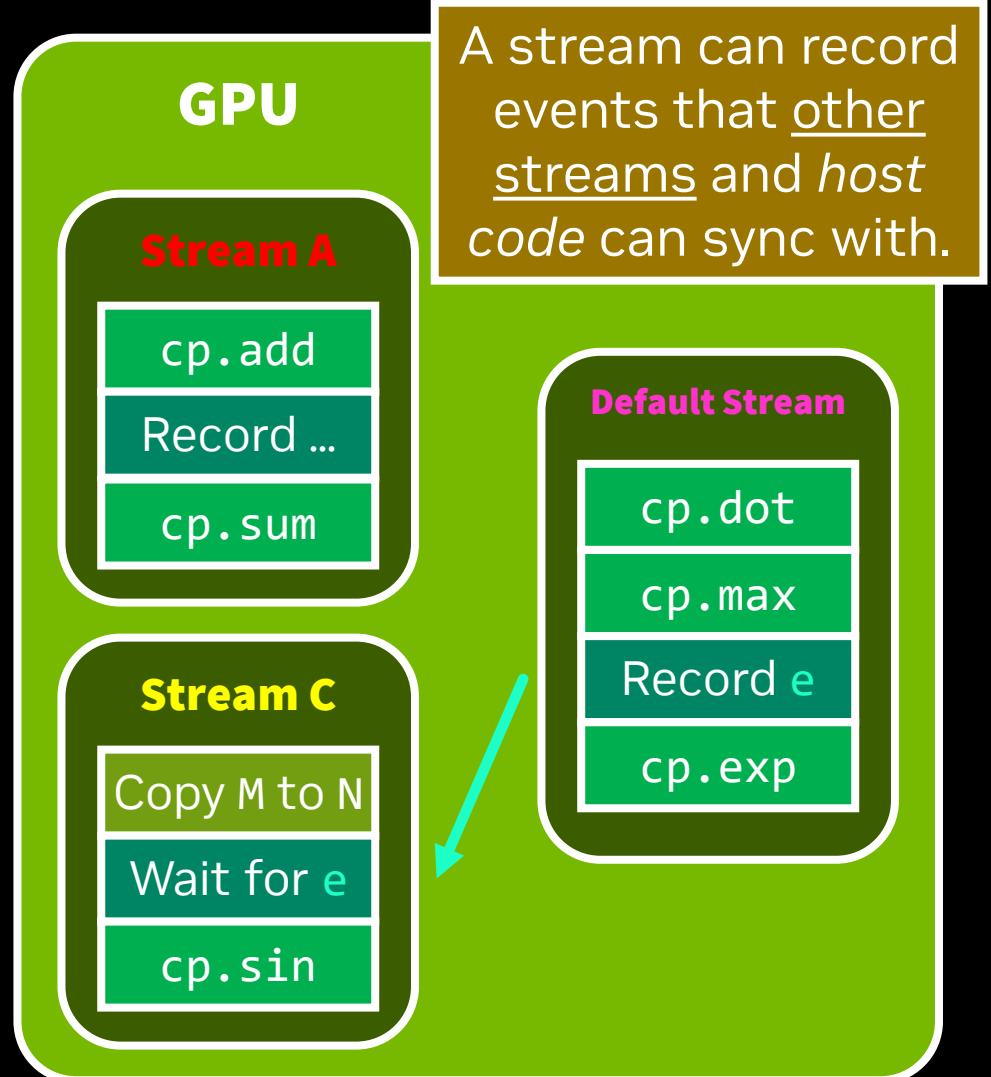


```
A = cp.cuda.Stream(non_blocking=True)
with A:
    cp.add(...)
    A.synchronize() # Host waits.
    cp.sum(...)
```

```
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.
```

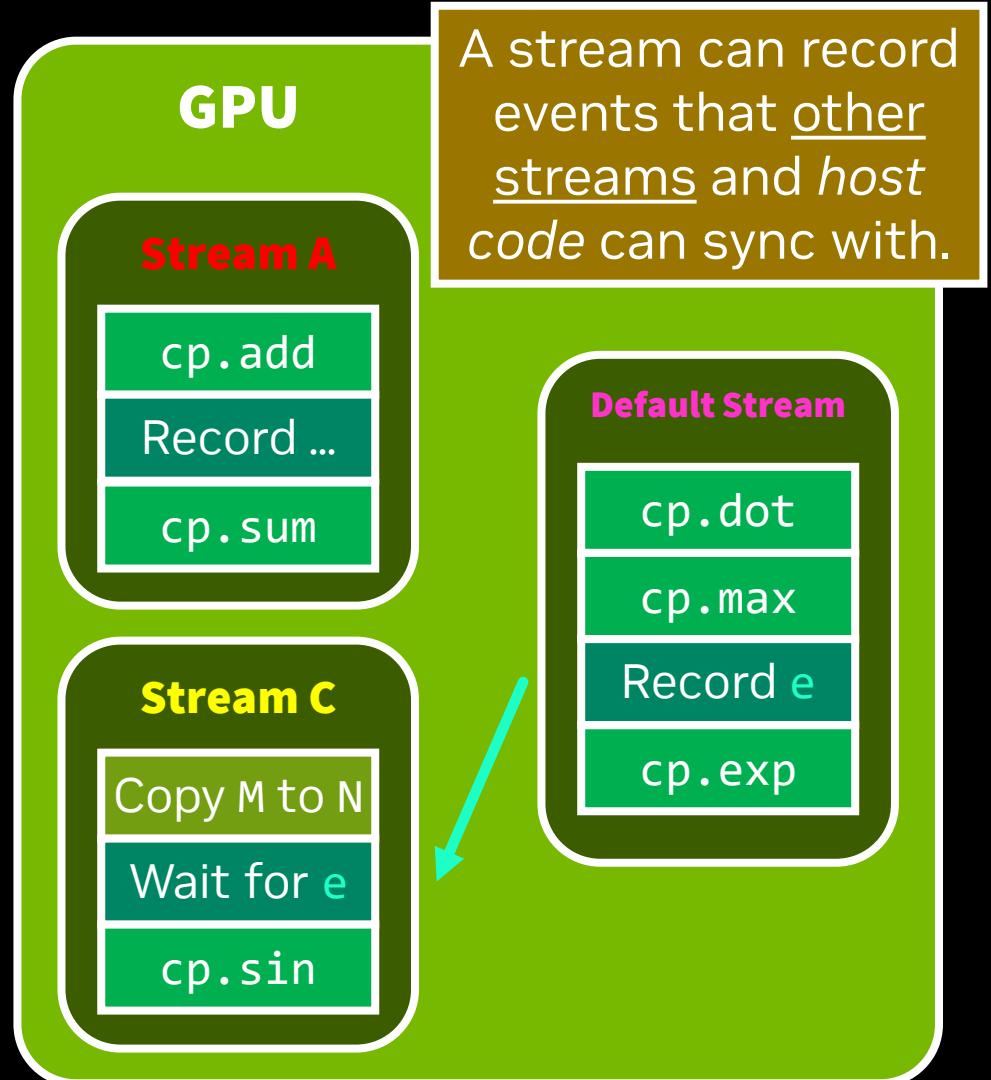


```
A = cp.cuda.Stream(non_blocking=True)
with A:
    cp.add(...)
    A.synchronize() # Host waits.
    cp.sum(...)
```

```
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/>

```
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())
```

```
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

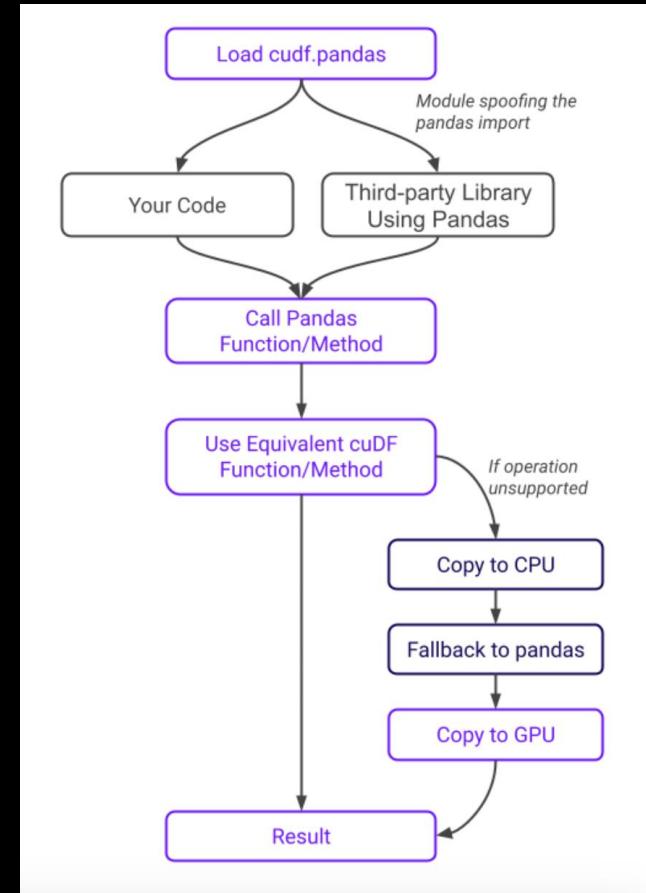
cudf.pandas lets you use pandas, but accelerates parts of the program that can use cudf

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)
```

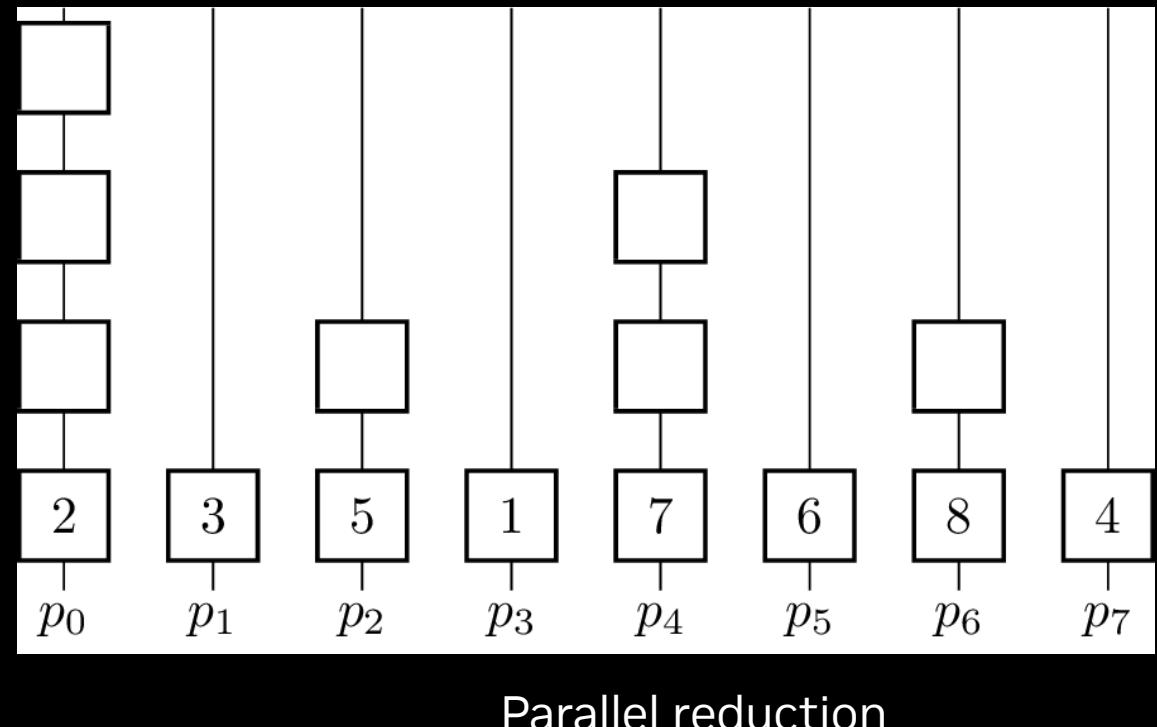
Parallel Algorithms

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 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

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}

template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```



Final Optimized Kernel

35

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)

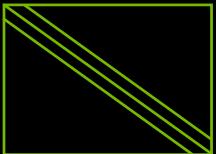
Exercise 9

cuda.cccl

Customizing Algorithms



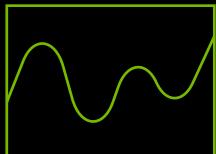
cuBLAS



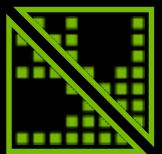
cuSPARSE



**cuFile
(GDS)**



cuFFT



cuDSS

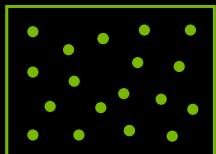
**CV
CUDA**

cuLitho

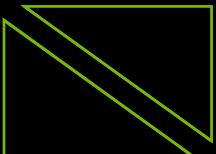
**Cooperative
Groups**



NVBENCH



cuRAND

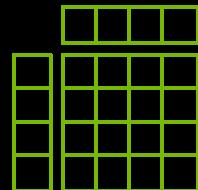


cuSOLVER

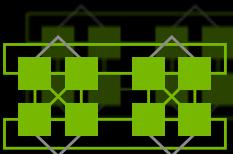
RAPIDS



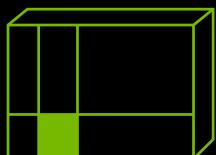
CUDA-Q



CUTLASS



NCCL



cuTENSOR



CUDA Math



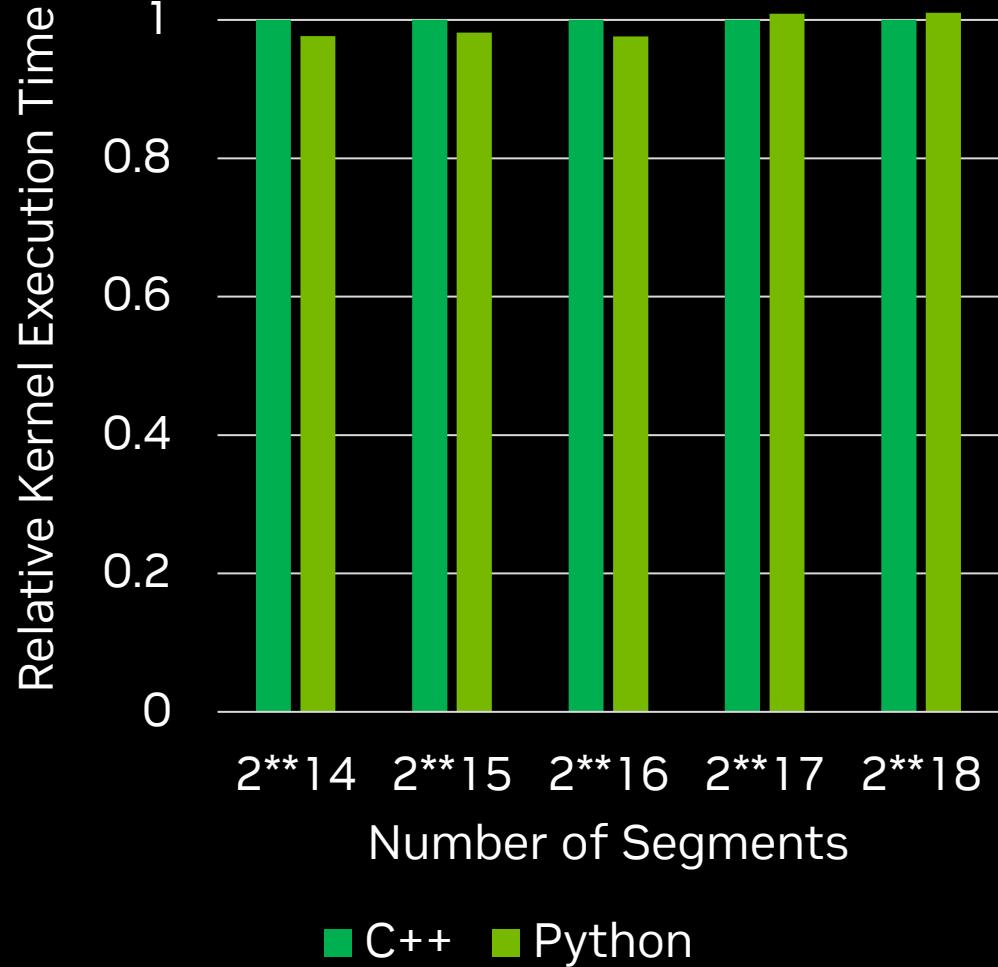
**CUDA
Runtime**



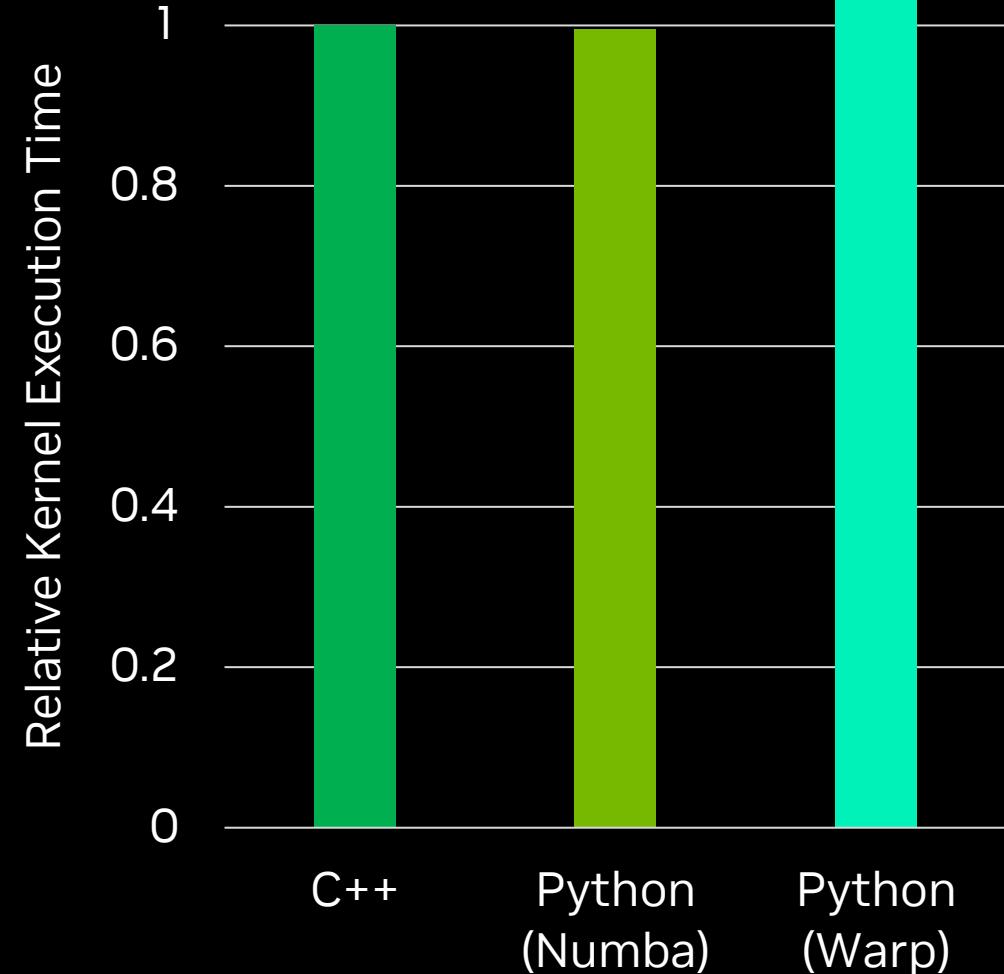
Prefer using libraries!

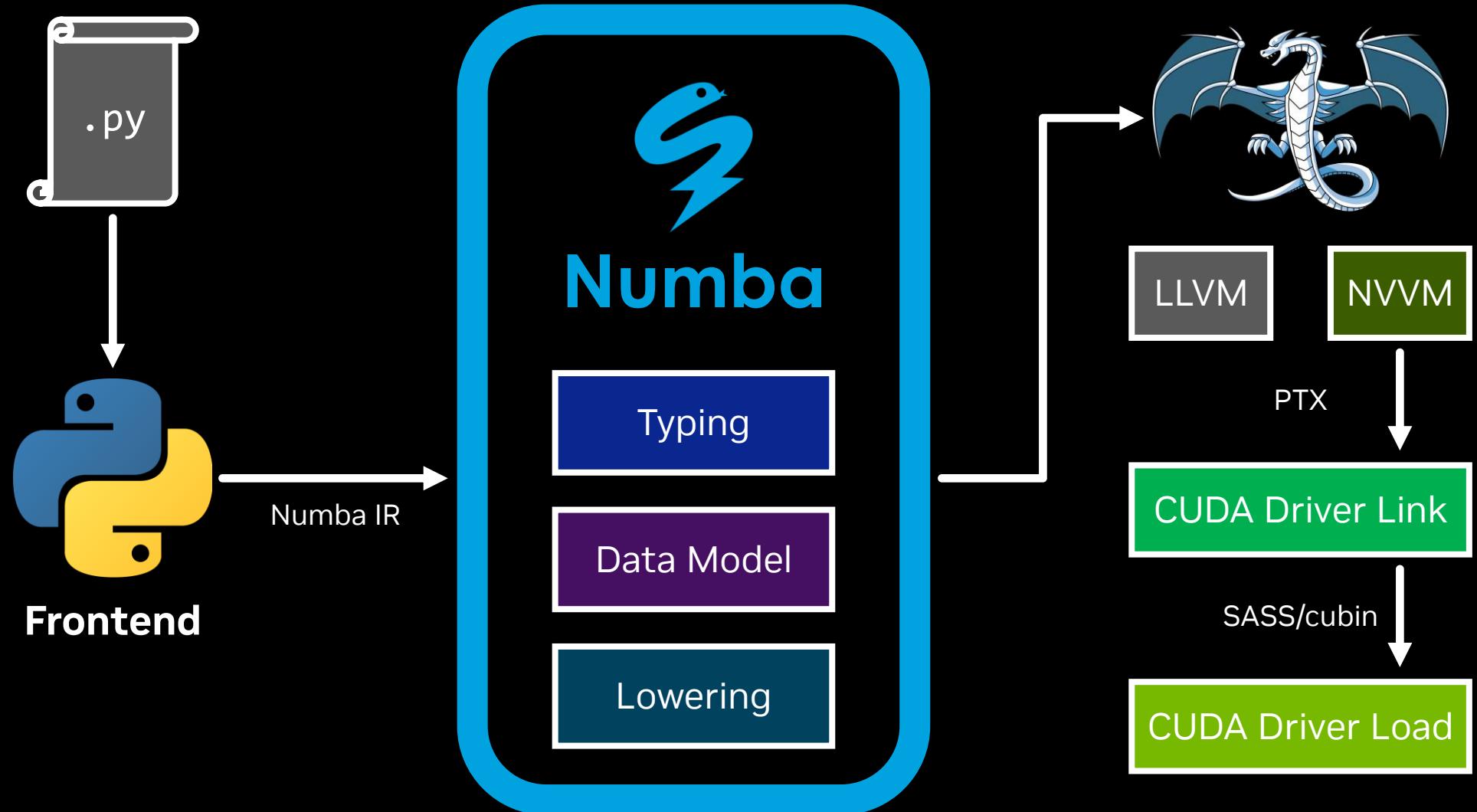
If no library exists or
you can't get the perf
you need with them,
then...

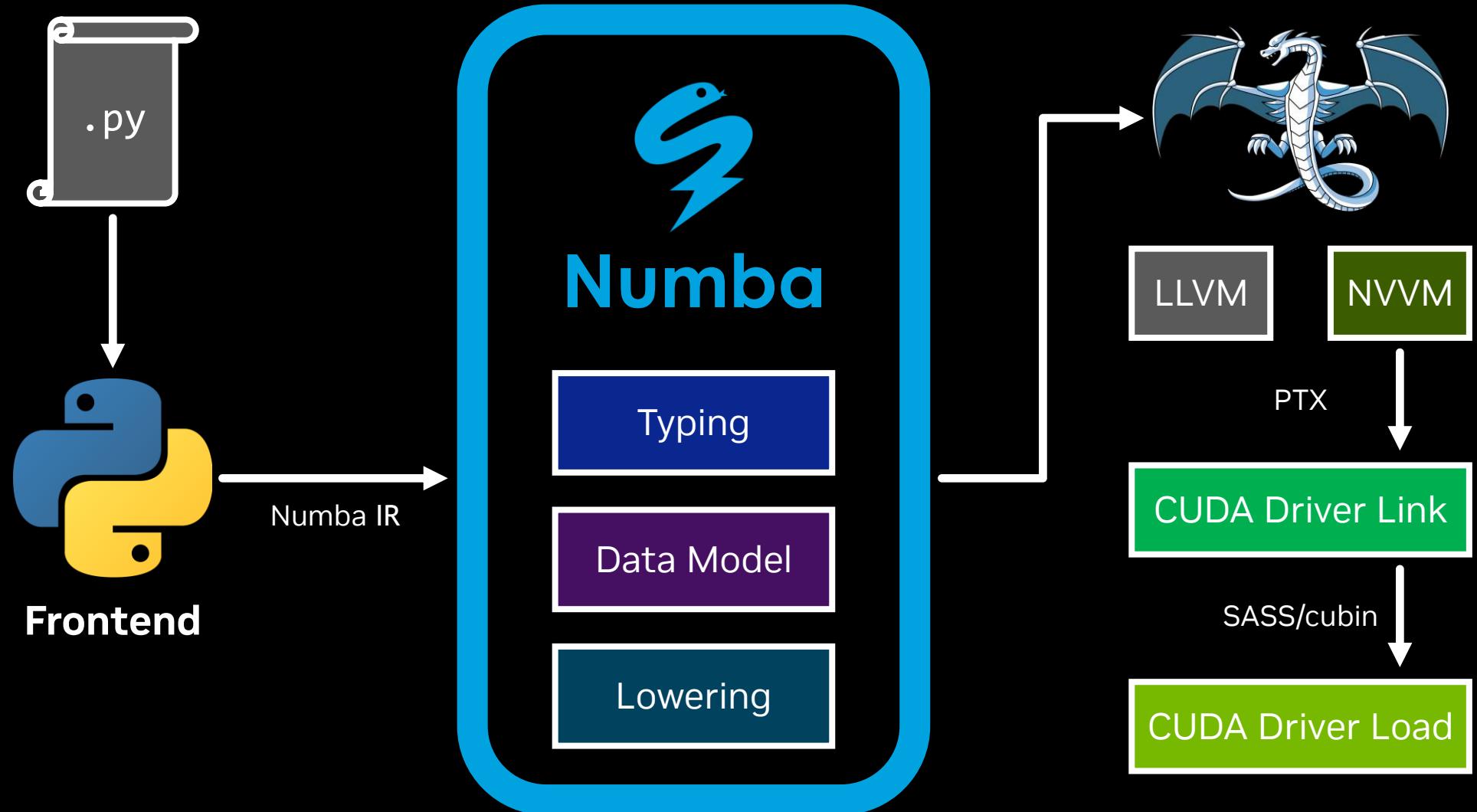
CUDA Segmented Sort Kernel



CUDA FFT Convolution Kernel







```
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]
```

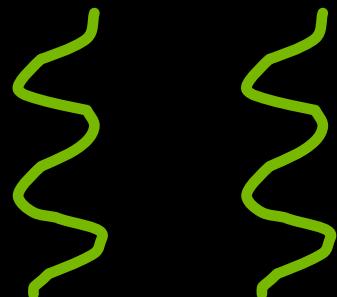
```
from numba import cuda

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

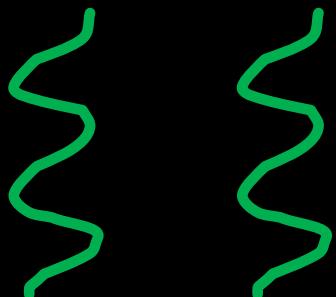
```
from numba import cuda
```

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

$d[0]=s[0]$ $d[1]=s[1]$



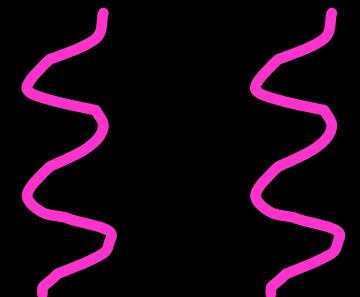
$d[2]=s[2]$ $d[3]=s[3]$



$d[4]=s[4]$ $d[5]=s[5]$

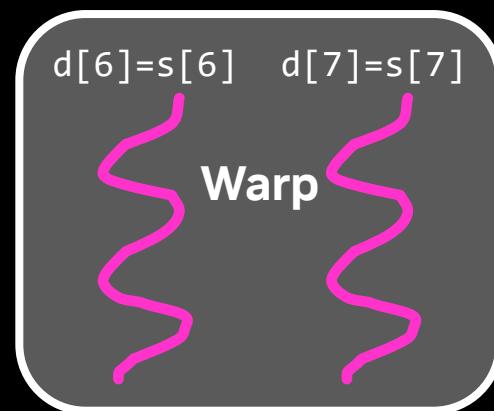
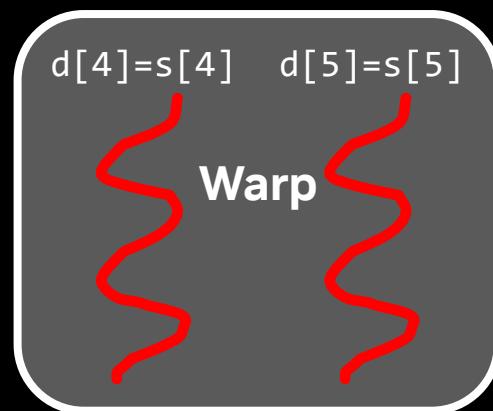
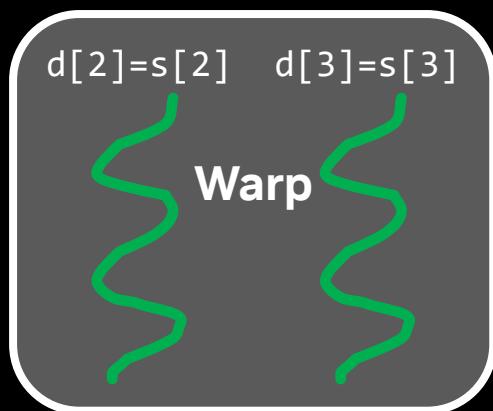
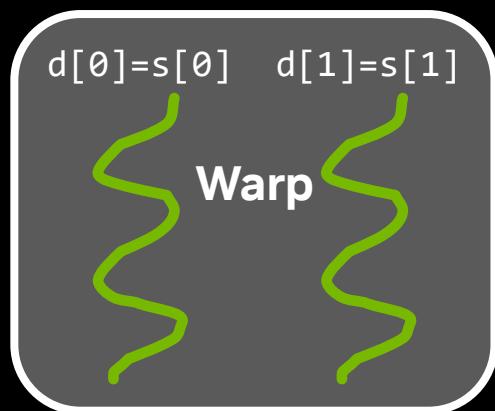


$d[6]=s[6]$ $d[7]=s[7]$



```
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]
```

Block 0

$d[0]=s[0]$ $d[1]=s[1]$

Warp
0

$d[2]=s[2]$ $d[3]=s[3]$

Warp
1

Block 1

$d[4]=s[4]$ $d[5]=s[5]$

Warp
0

$d[6]=s[6]$ $d[7]=s[7]$

Warp
1

```
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)
```

Block 0

$d[0]=s[0]$ $d[1]=s[1]$

Warp
0

$d[2]=s[2]$ $d[3]=s[3]$

Warp
1

Block 1

$d[4]=s[4]$ $d[5]=s[5]$

Warp
0

$d[6]=s[6]$ $d[7]=s[7]$

Warp
1

```
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)
```

Block 0

$d[0]=s[0]$ $d[1]=s[1]$

Warp
0

$d[2]=s[2]$ $d[3]=s[3]$

Warp
1

Block 1

$d[4]=s[4]$ $d[5]=s[5]$

Warp
0

$d[6]=s[6]$ $d[7]=s[7]$

Warp
1

```
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)
```

Block 0

$d[0]=s[0]$ $d[1]=s[1]$

Warp
0

$d[2]=s[2]$ $d[3]=s[3]$

Warp
1

Block 1

$d[4]=s[4]$ $d[5]=s[5]$

Warp
0

$d[6]=s[6]$ $d[7]=s[7]$

Warp
1

```
@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)
```

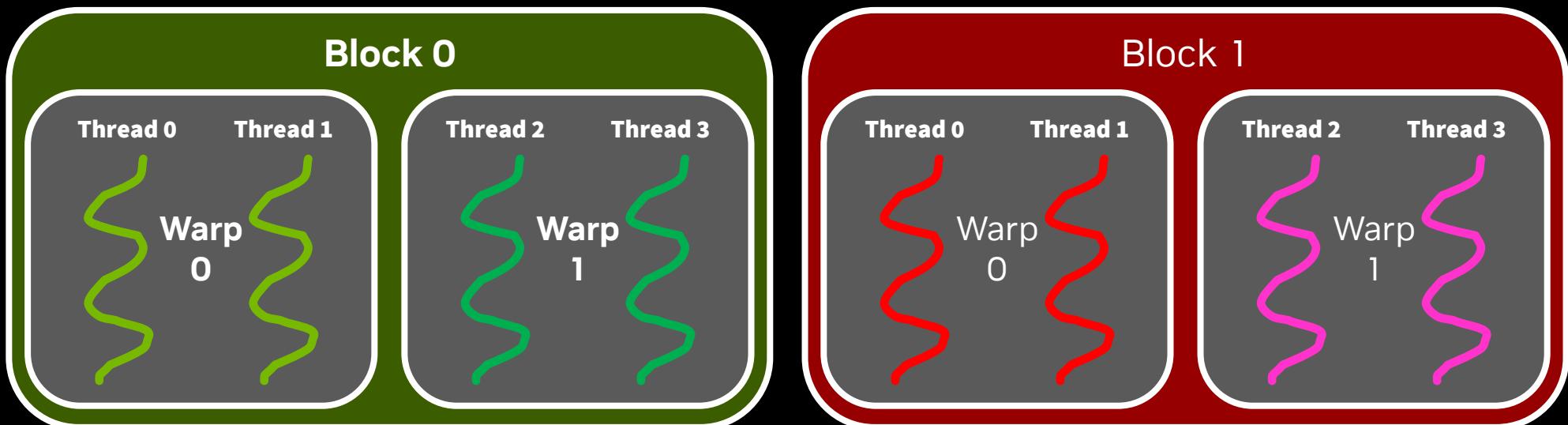
N = Size of the arrays
B = Number of thread blocks
T = Items per thread

```
@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

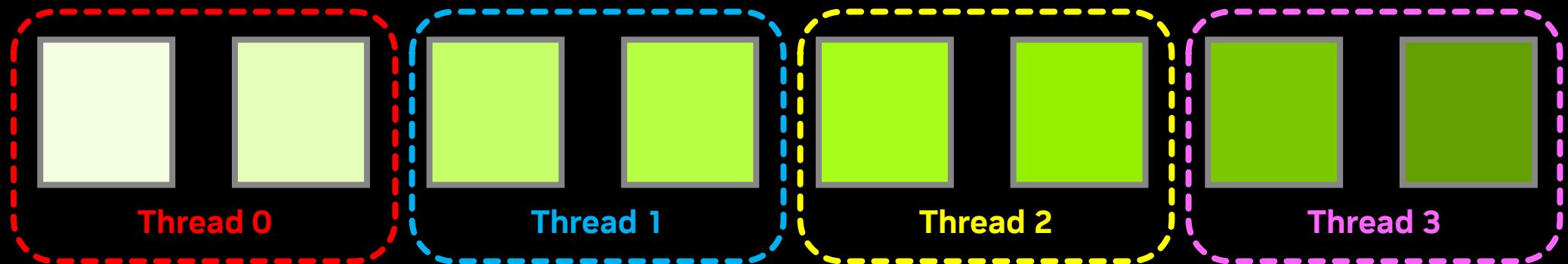
    base = tx + bx * tile
    for i in range(0, tile, bd):
        d[base + i] = s[base + i]

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

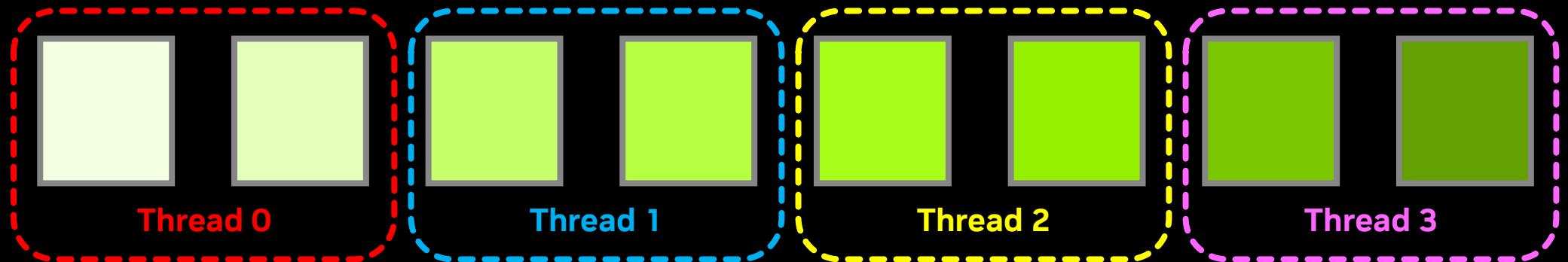
```

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

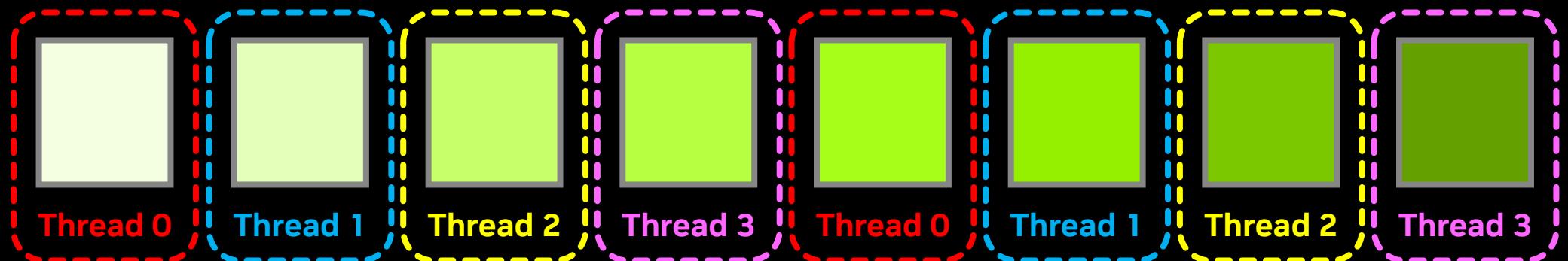
Blocked Arrangement

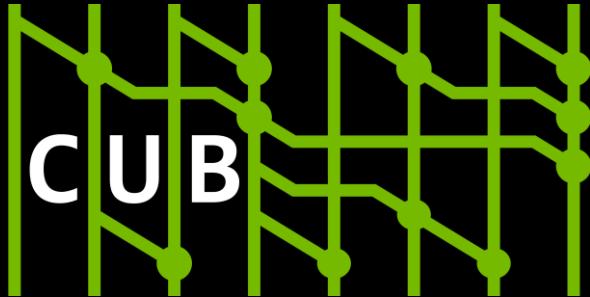


Blocked Arrangement



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;

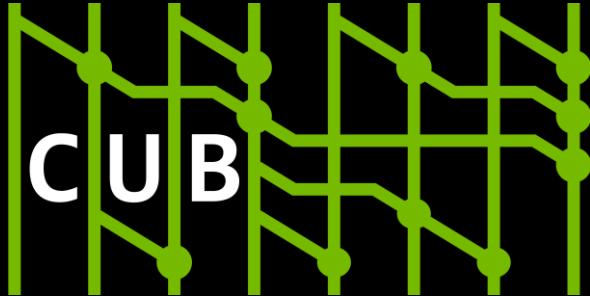
    // Obtain this block's segment of consecutive keys
    int thread_keys[ITEMS];
    int offset = blockIdx.x * (BLOCK_THREADS * ITEMS);
    BlockLoadT(temp.load).Load(d_in + offset, thread_keys);

    __syncthreads();           // Barrier for smem reuse

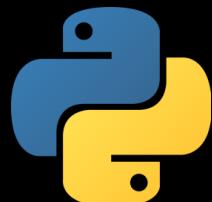
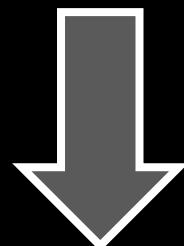
    // Collectively sort the keys
    BlockRadixSortT(temp.sort).Sort(thread_keys);

    __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;

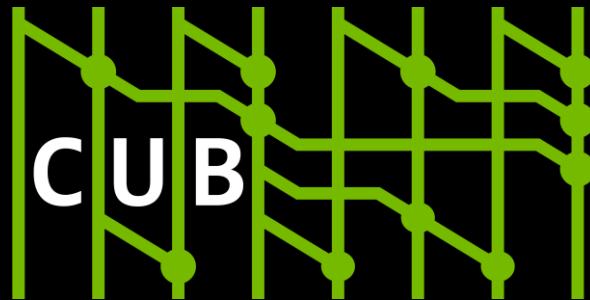
    // Obtain this block's segment of consecutive keys
    int thread_keys[ITEMS];
    int offset = blockIdx.x * (BLOCK_THREADS * ITEMS);
    BlockLoadT(temp.load).Load(d_in + offset, thread_keys);

    __syncthreads();           // Barrier for smem reuse

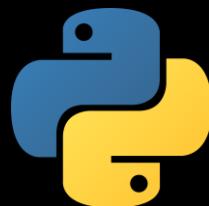
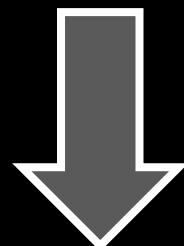
    // Collectively sort the keys
    BlockRadixSortT(temp.sort).Sort(thread_keys);

    __syncthreads();           // Barrier for smem reuse

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

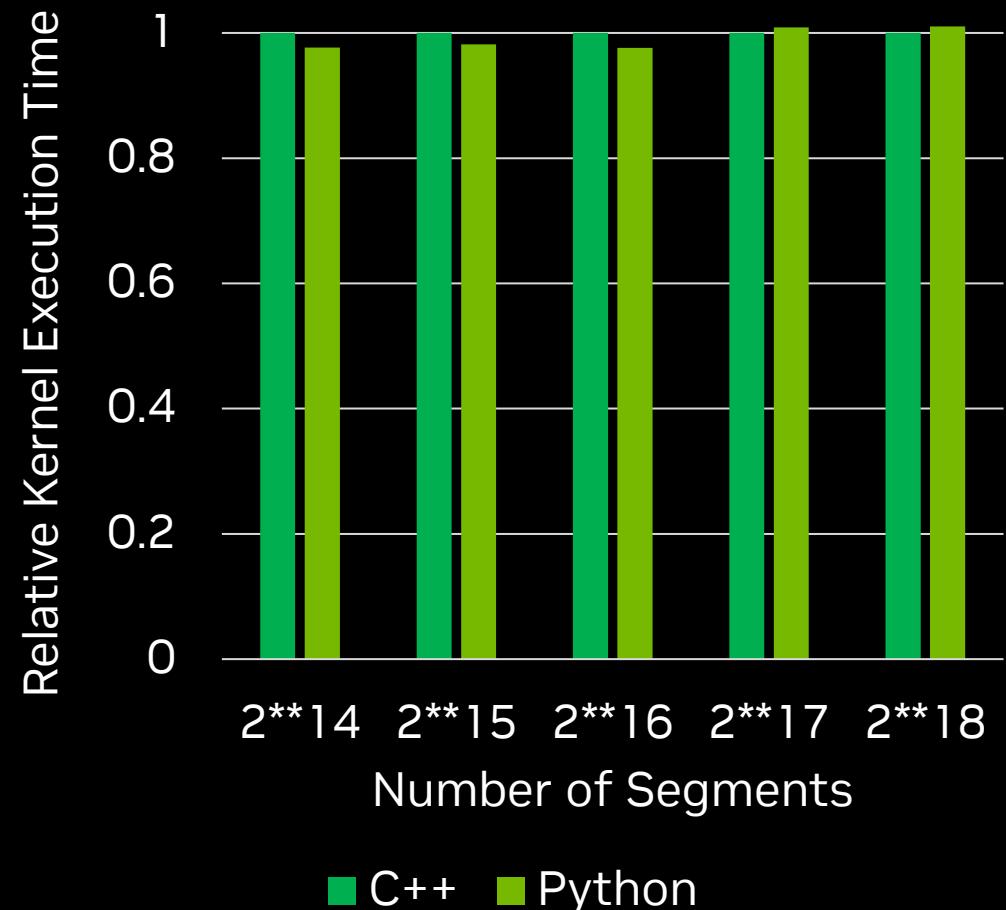


Cooperative primitives for CUDA C++
<https://nvidia.github.io/cccl/cub>



```
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)

```

```

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."

e	h
1	1

o
1

s
1

w
1

"Whose woods these are I think I know."

e	h
1	1

o
1

s
1

w
1

1

"Whose woods these are I think I know."

e	h
1	1

o
1

s
1

w
2

1

"Whose woods these are I think I know."

e	h
1	1

o
2

s
1

w
2

1

"Whose wods these are I think I know."

e	h
1	1

o
3

s
1

w
2

1

"Whose woodss these are I think I know."

d	e	h
1	1	1

o
3

s
1

w
2

1

"Whose woods these are I think I know."

d	e	h
1	1	1

o
3

s
2

w
2

1

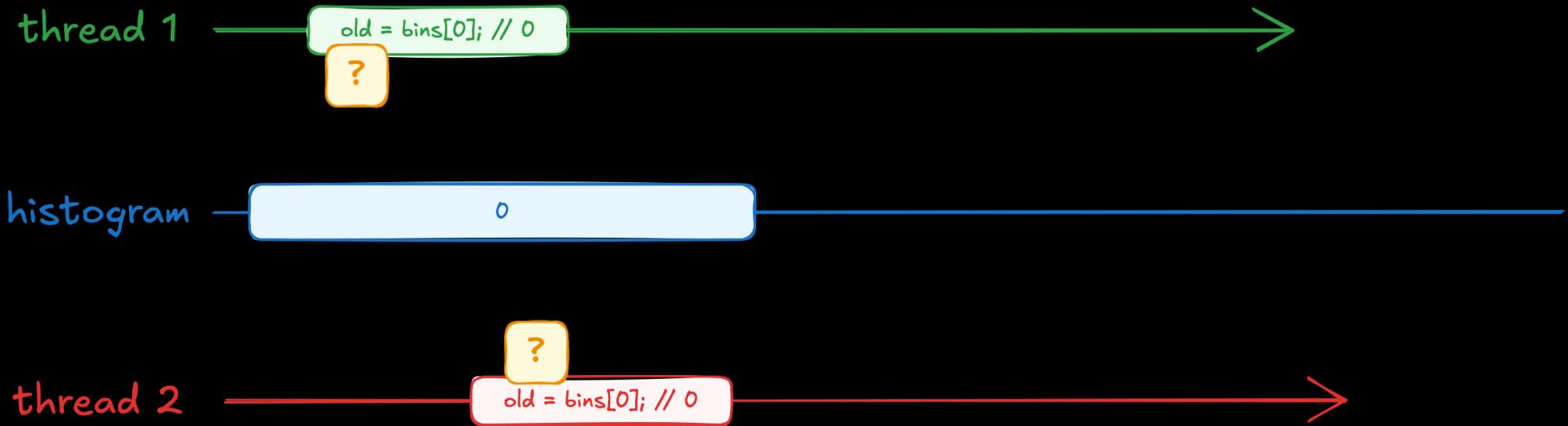
"Whose woods these are I think I know."

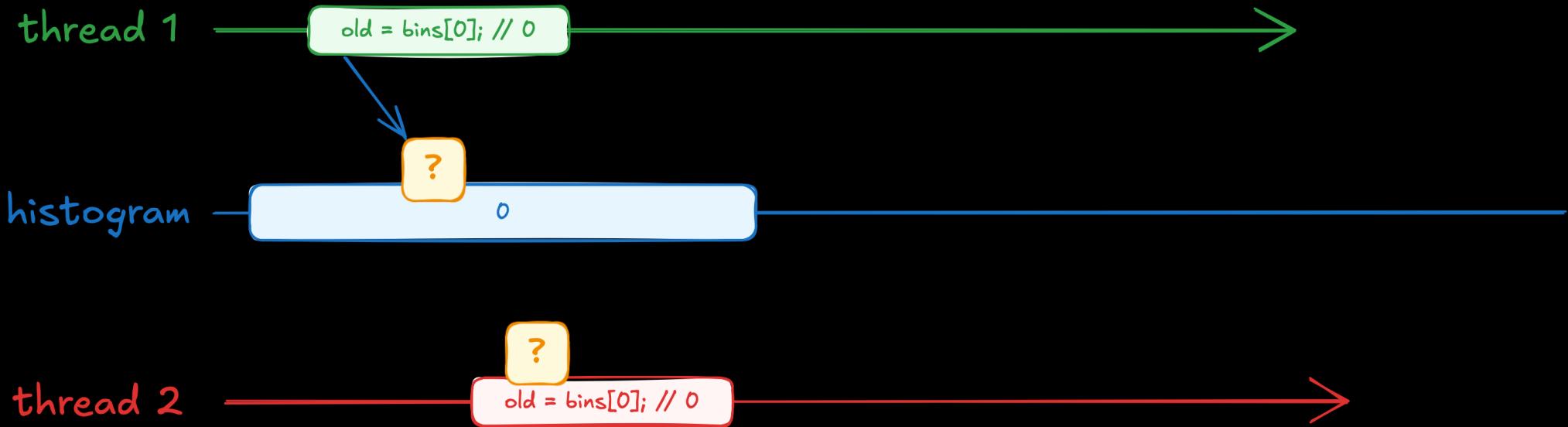
a	d	e	h	i	k	n	o	r	s	t	w	.	
1	1	4	3	3	2	2	4	1	3	2	3	1	7

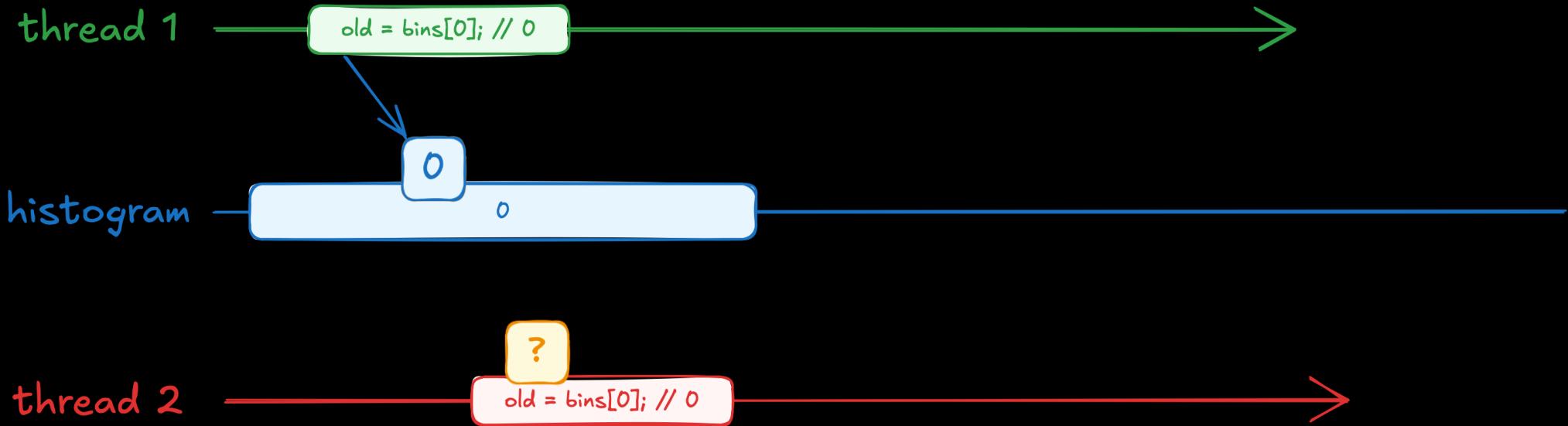
Exercise 11

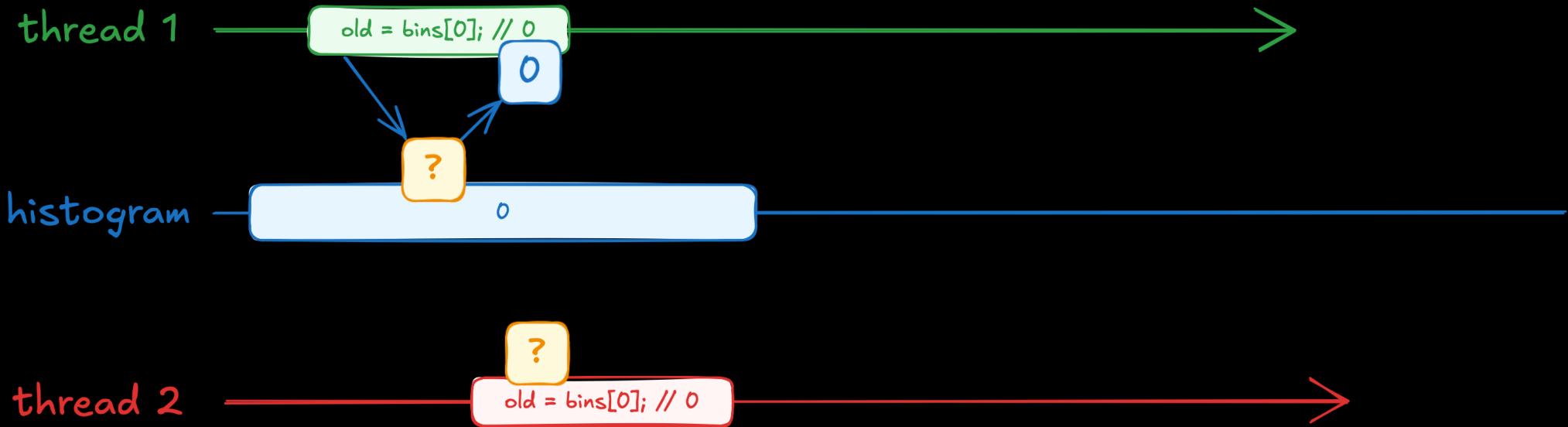
Kernel Authoring

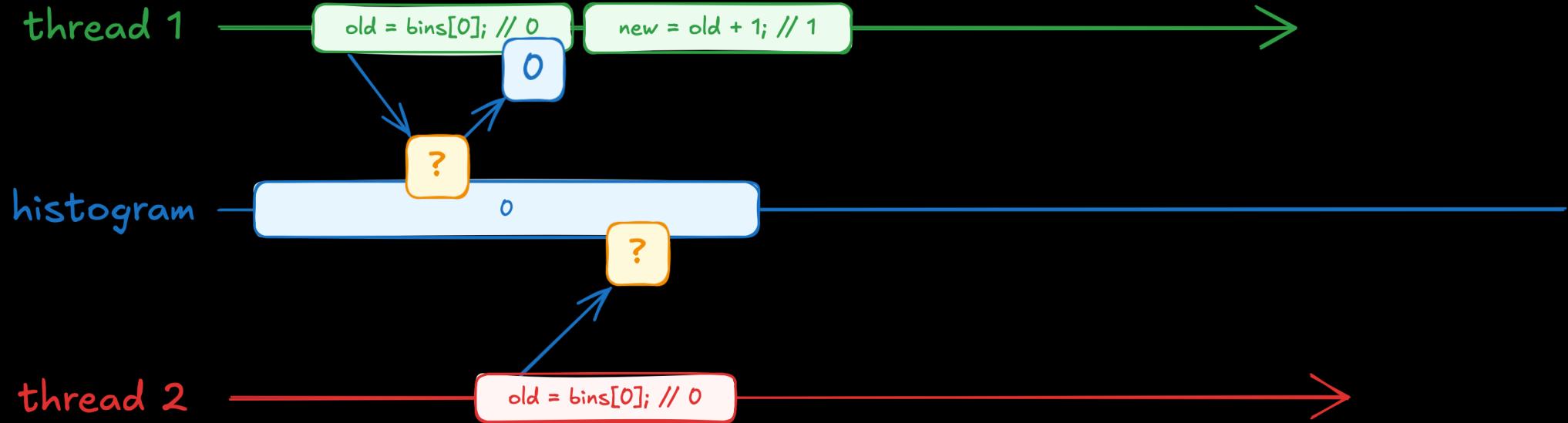
Book Histogram

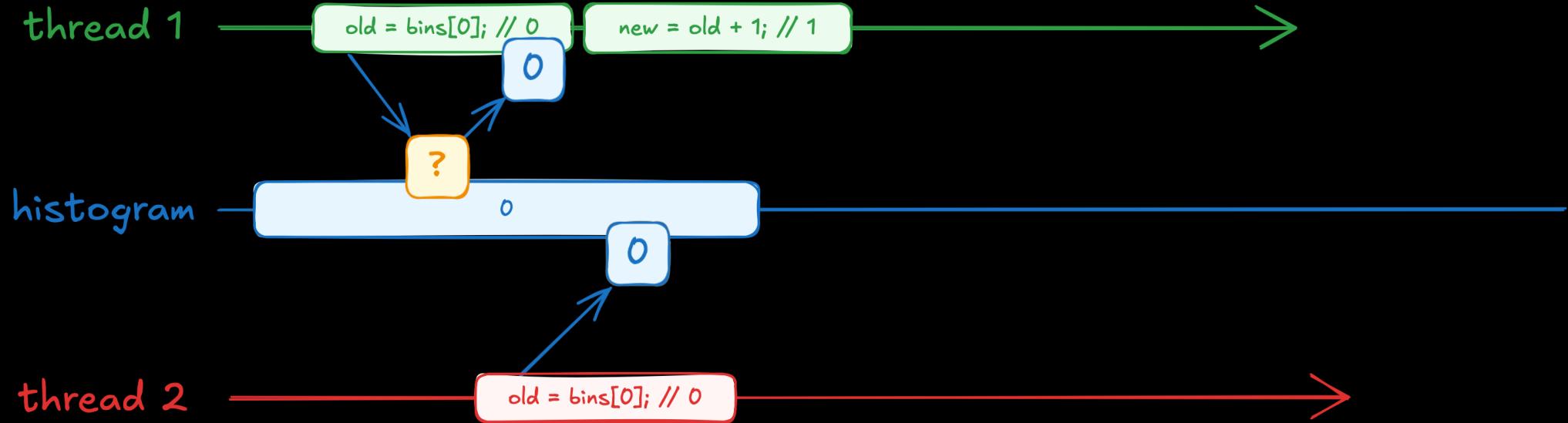


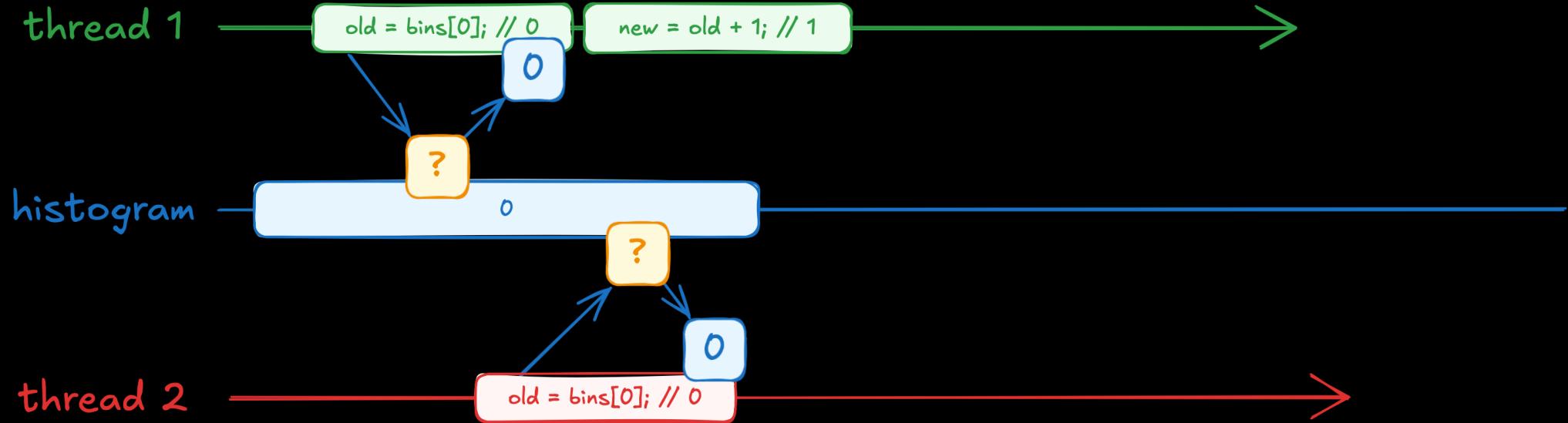


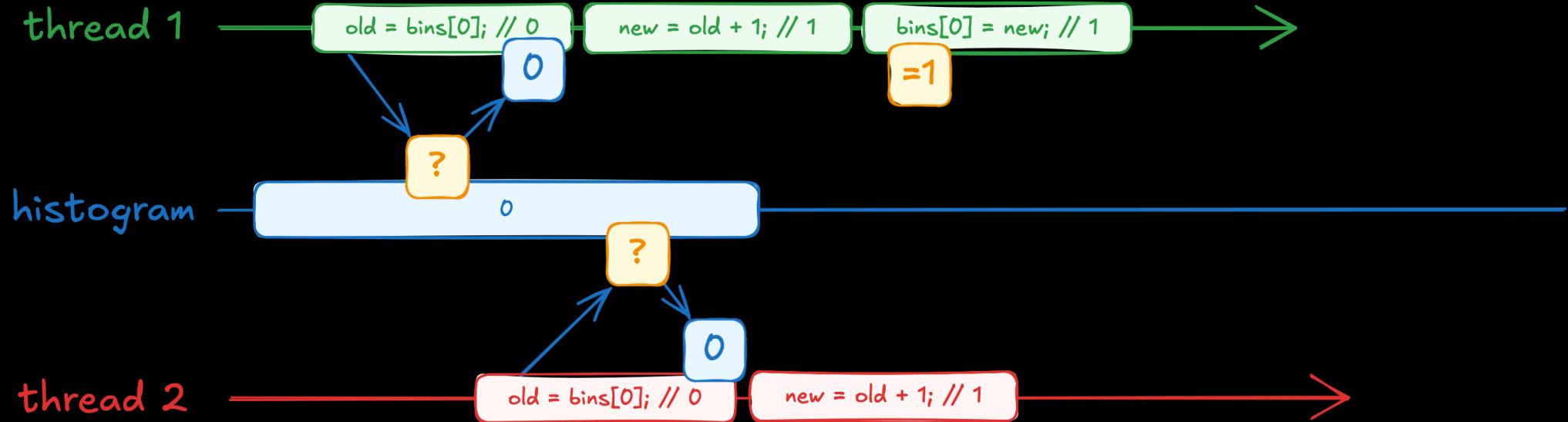


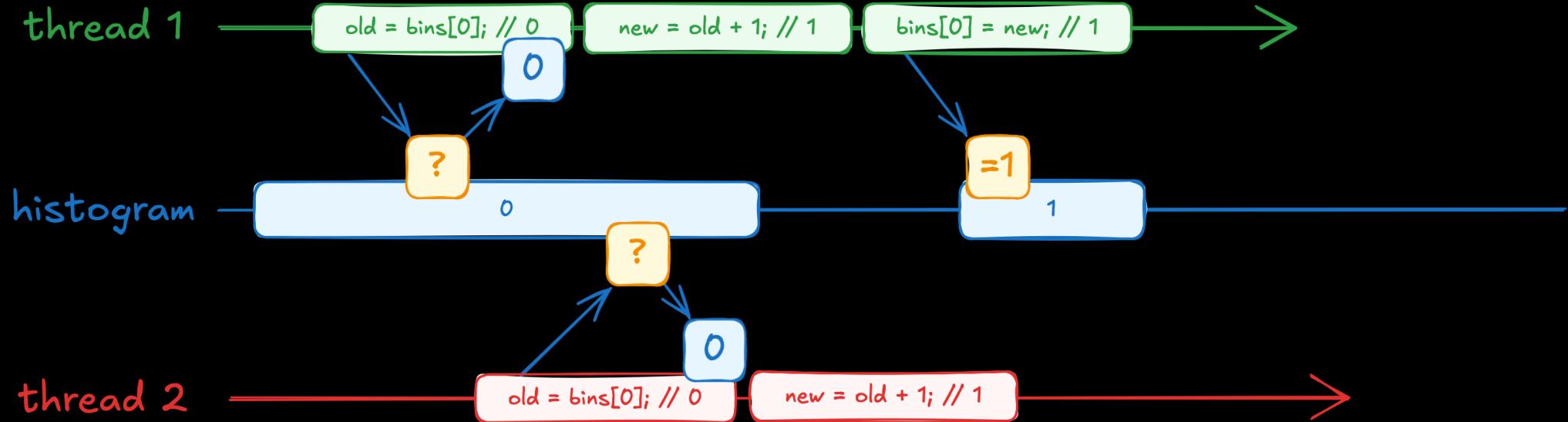


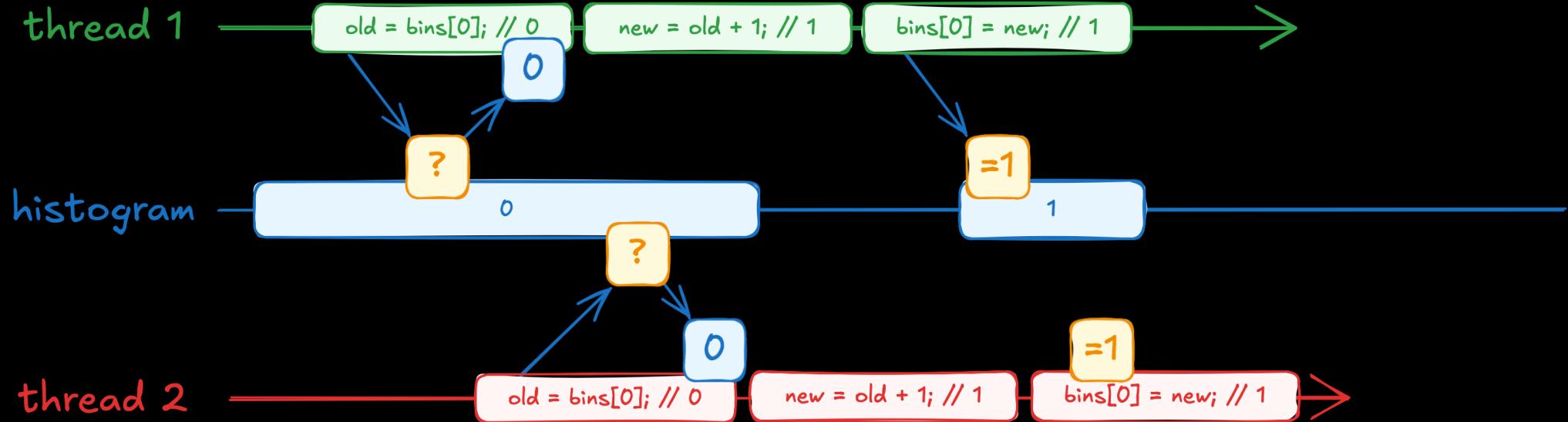


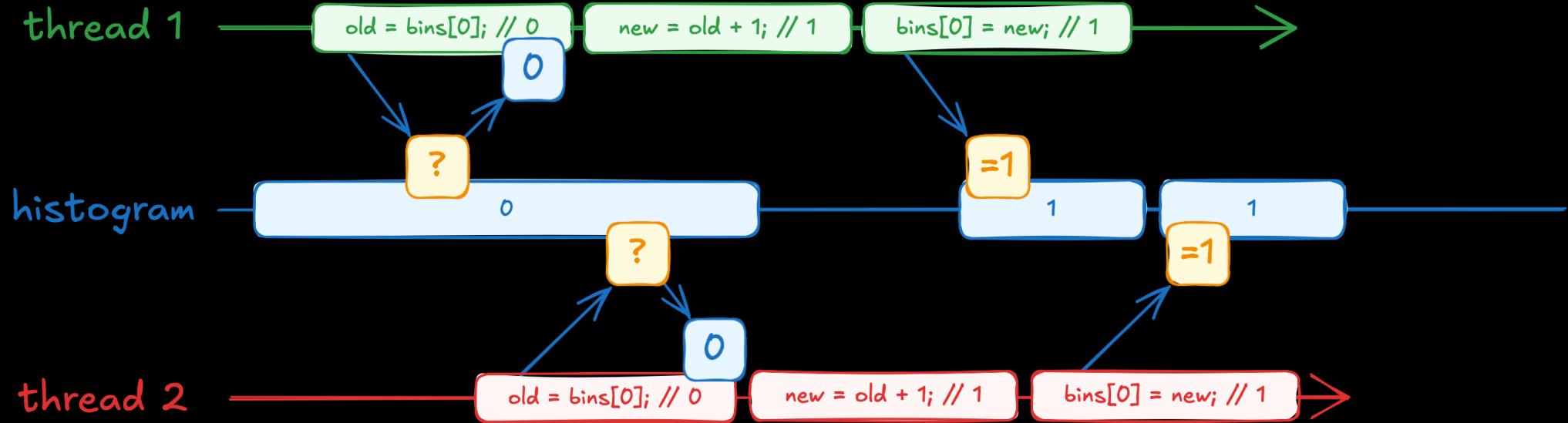








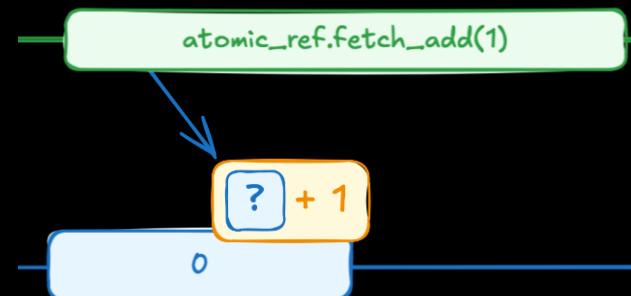
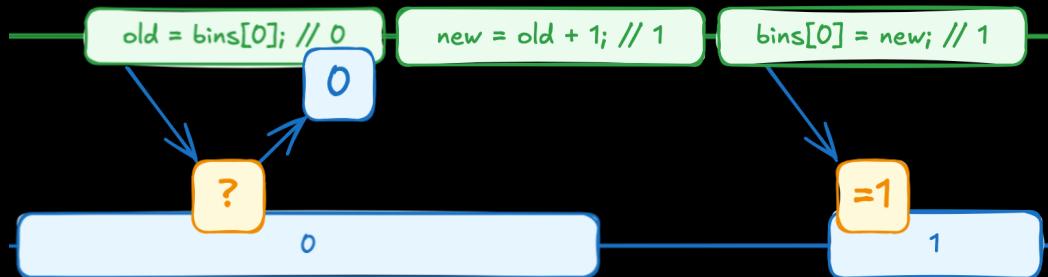




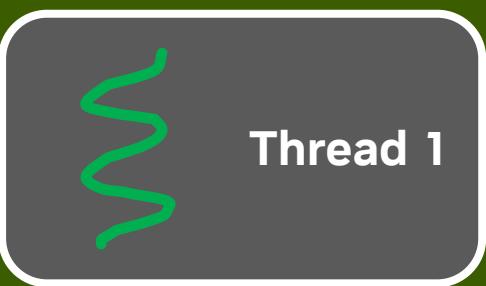
To fix the data race, the read, modify and write must be a single indivisible operation.

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

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



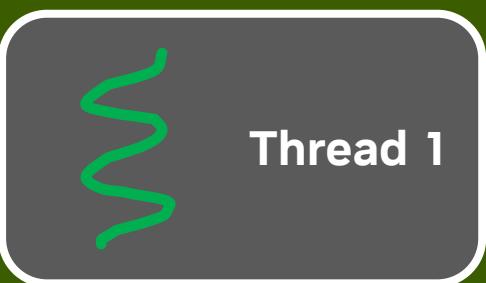
Block 0



Block 1



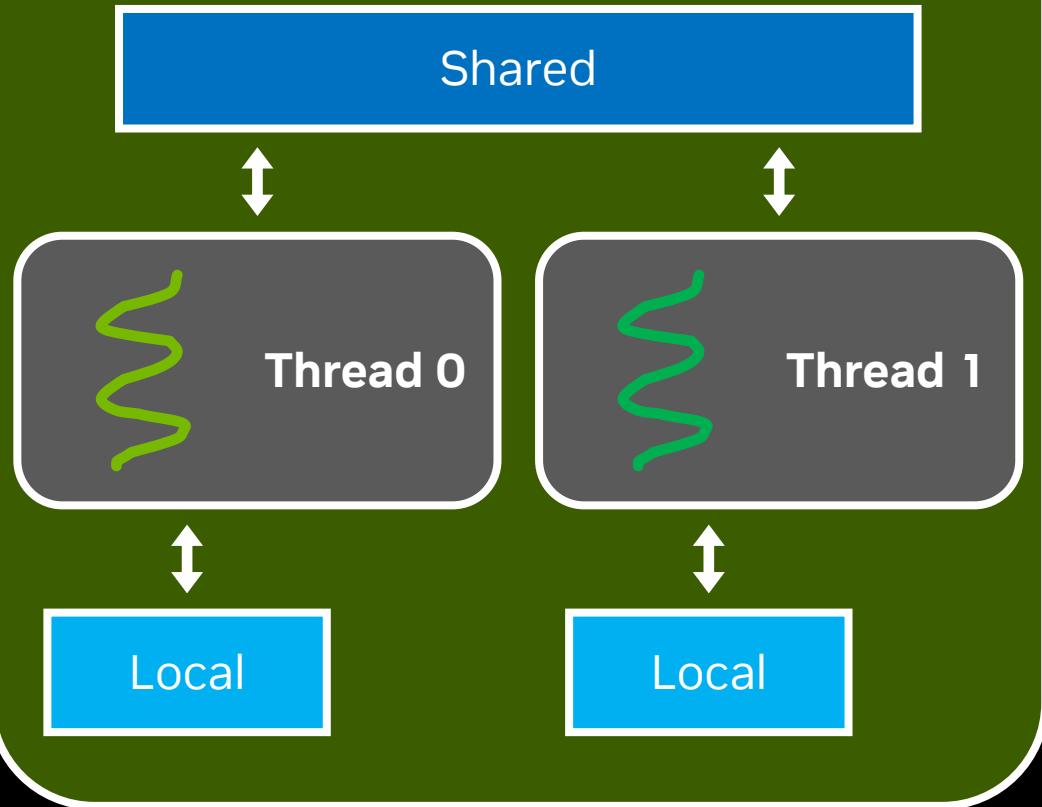
Block 0



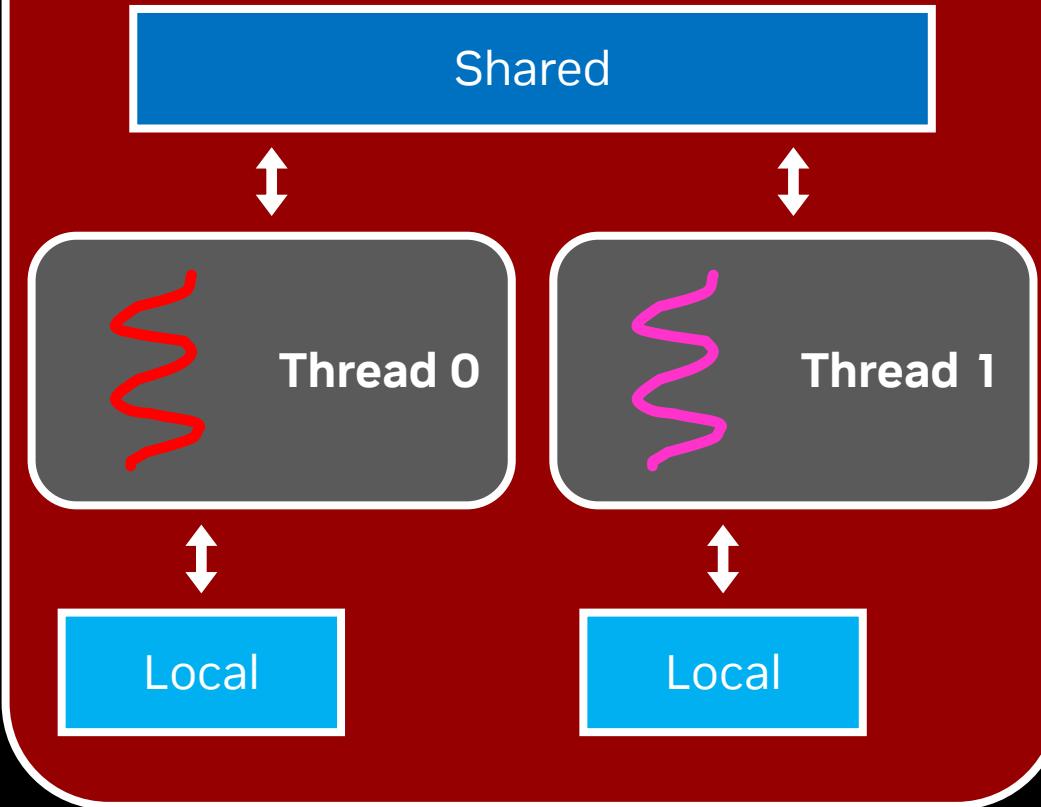
Block 1



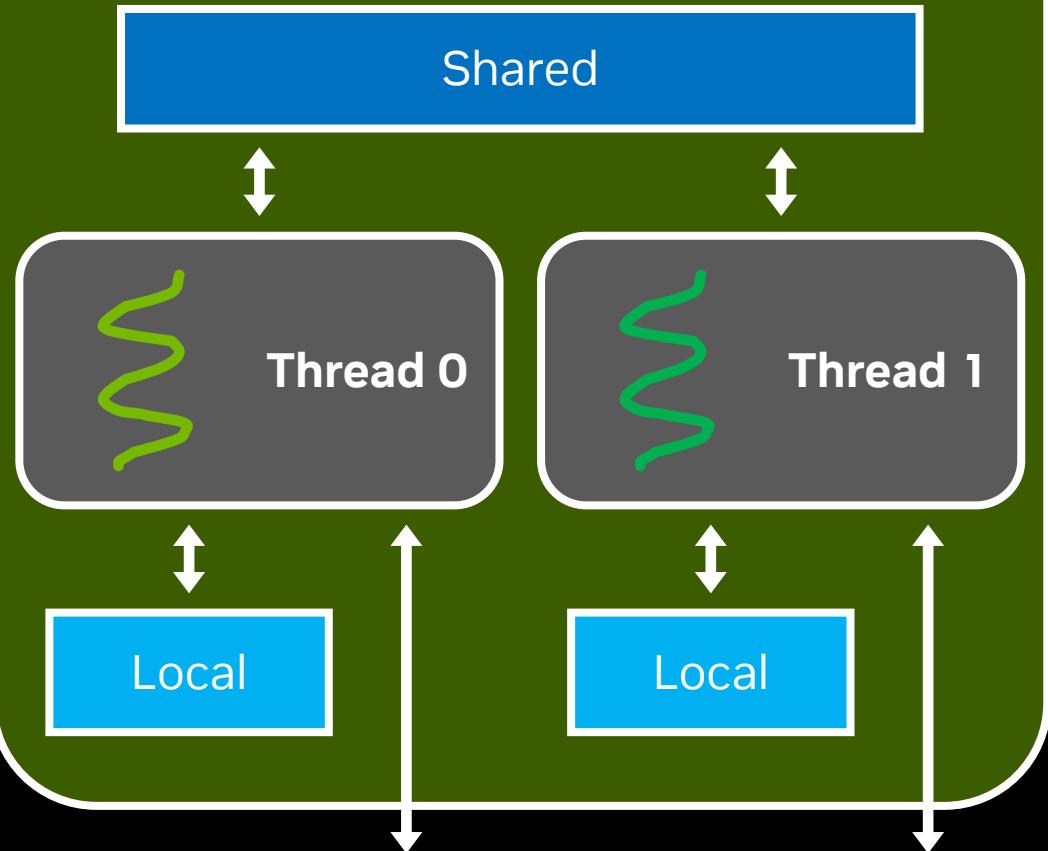
Block 0



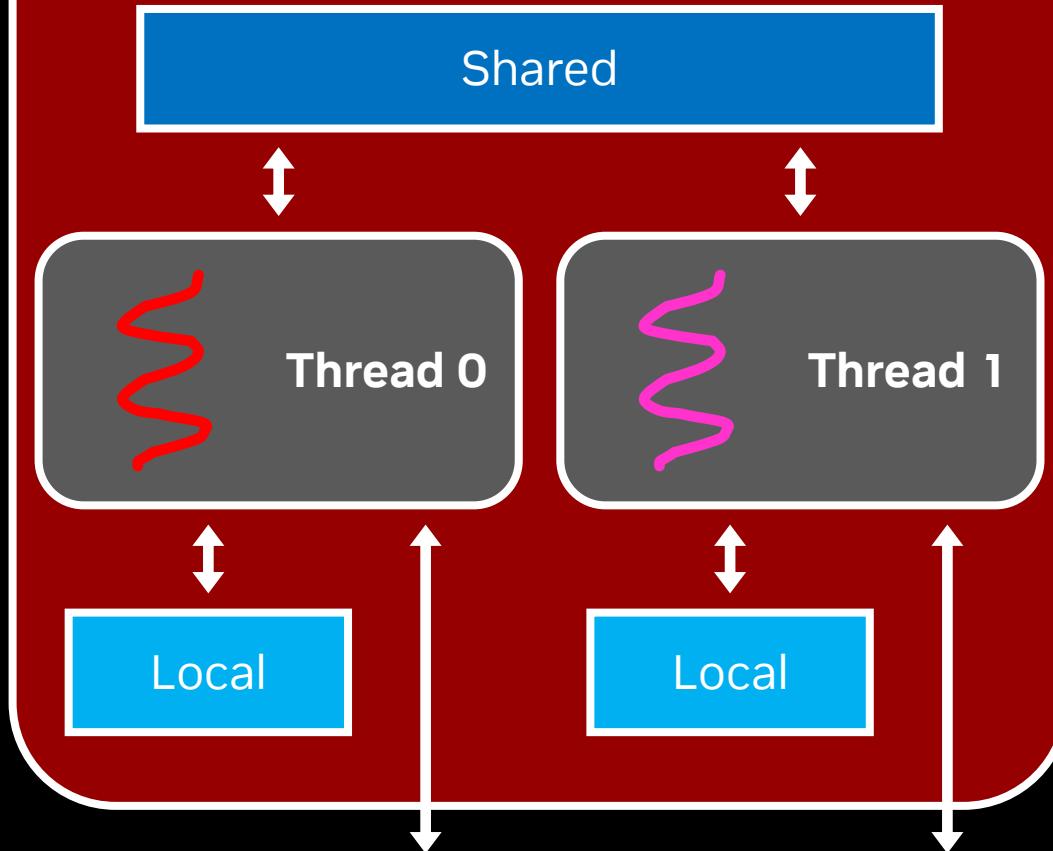
Block 1



Block 0



Block 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)

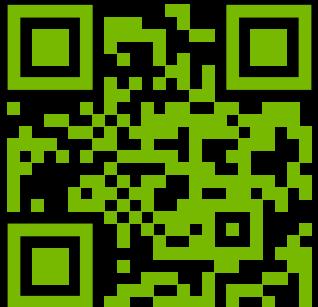
    block_load(input, shared)

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

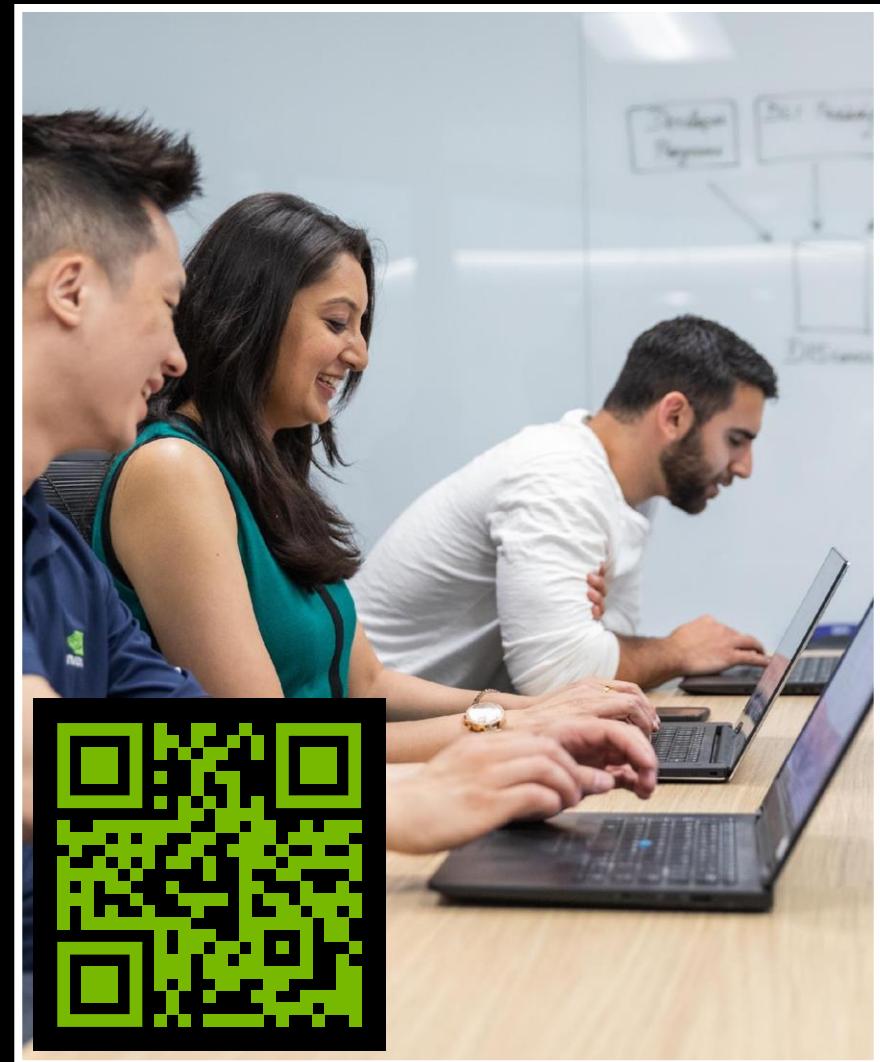
- [S72683](#) CUDA Techniques to Maximize Memory Bandwidth & Hide Latency
- [S72685](#) CUDA Techniques to Maximize Compute & Instruction Throughput
- [S72686](#) CUDA Techniques to Maximize Concurrency & System Utilization
- [S72687](#) Get the Most Performance From Grace Hopper

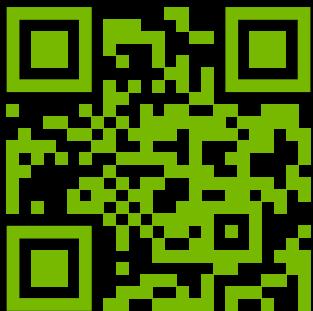
NVIDIA DLI Courses

- 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

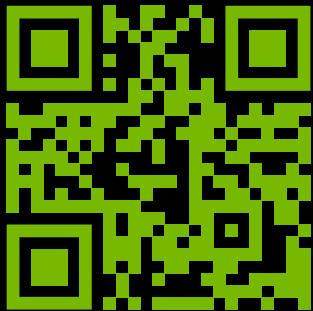




GTC CUDA
Developer Track
nvda.ws/4dVjYXs



Accelerated
Computing Hub
nvda.ws/43tdsU6



NVIDIA
DLI Courses
nvda.ws/3ZLPyAL



ADSP
Podcast
adspthepodcast.com

