



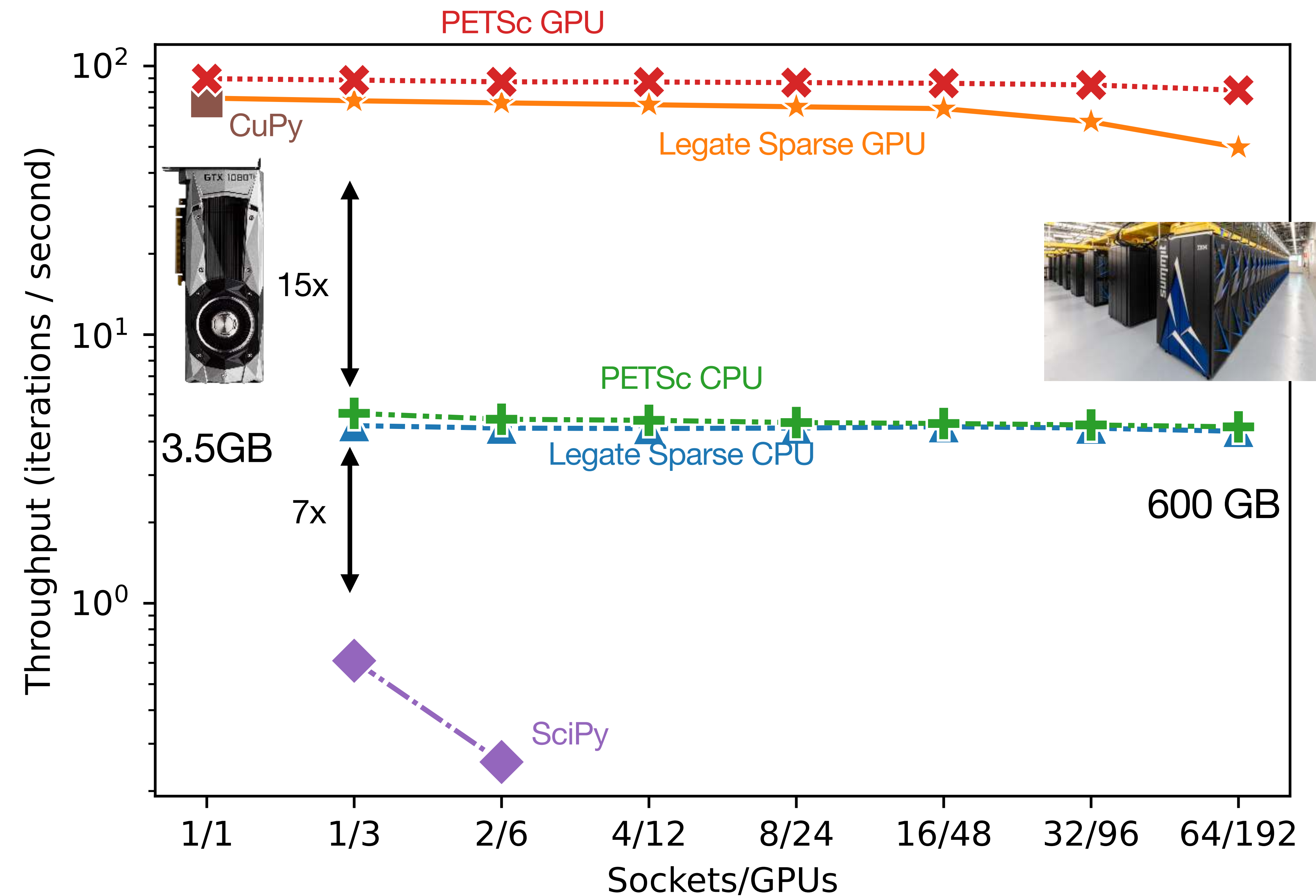
# Legate Sparse

Rohan Yadav, Shriram Jagannathan, Wonchan Lee, Legate@NVIDIA

# Legate Sparse Automatically Scales SciPy Sparse Programs

Another step towards making HPC accessible

```
import legate.sparse as sp
import cupynumeric as np
# Generate a random positive semi-definite matrix.
A = sp.random(n, n, format="csr")
A = 0.5 * (A + A.T) + n * sp.eye(n)
# Setup and run a CG solve.
x, b = np.rand(A.shape[0]), np.zeros(A.shape[0])
r = b - A @ x
p, rho = r, r @ r
while True:
    rho1 = rho
    rho = r @ r
    p = p * (rho / rho1) + r
    q = A @ p
    pq = p @ q
    alpha = rho / pq
    x += alpha * p
    r -= alpha * q
    if np.linalg.norm(r) < 1e-6:
        break
```



# A member of the Legate ecosystem

Now, one of many

cuPyNumeric

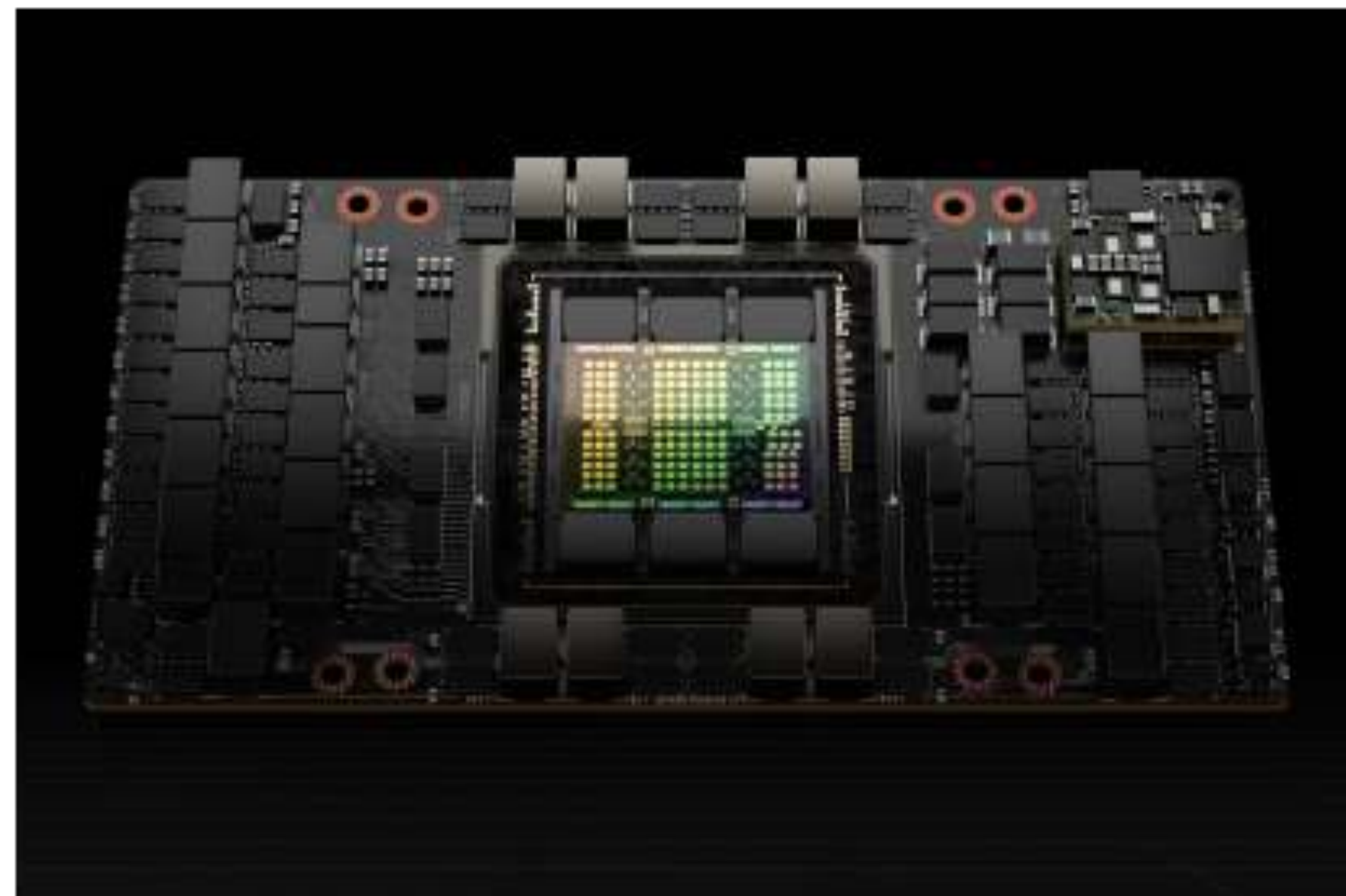
Legate  
Sparse

Legate  
Dataframe

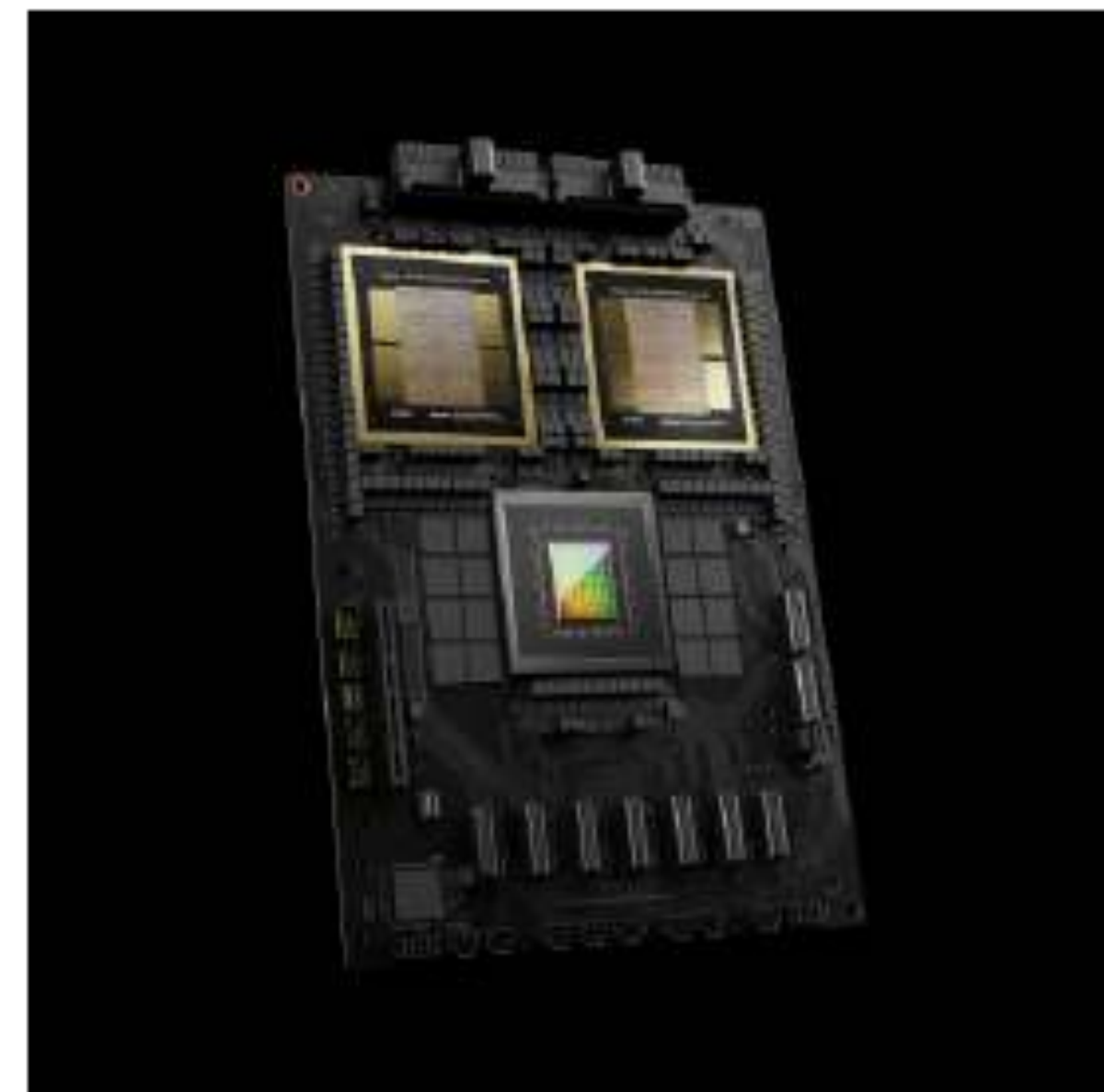
Legate  
Boost

Legate  
RAFT

**Legate** - Distributed Framework and Runtime for Accelerated Computing



**Single GPU**



**Mixed CPU/GPU**



**Single-Node  
Multi-GPU**



**Multi-GPU Multi-Node  
Cloud & Supercomputer**

# Legion Retreat December 2022

Close to the beginning

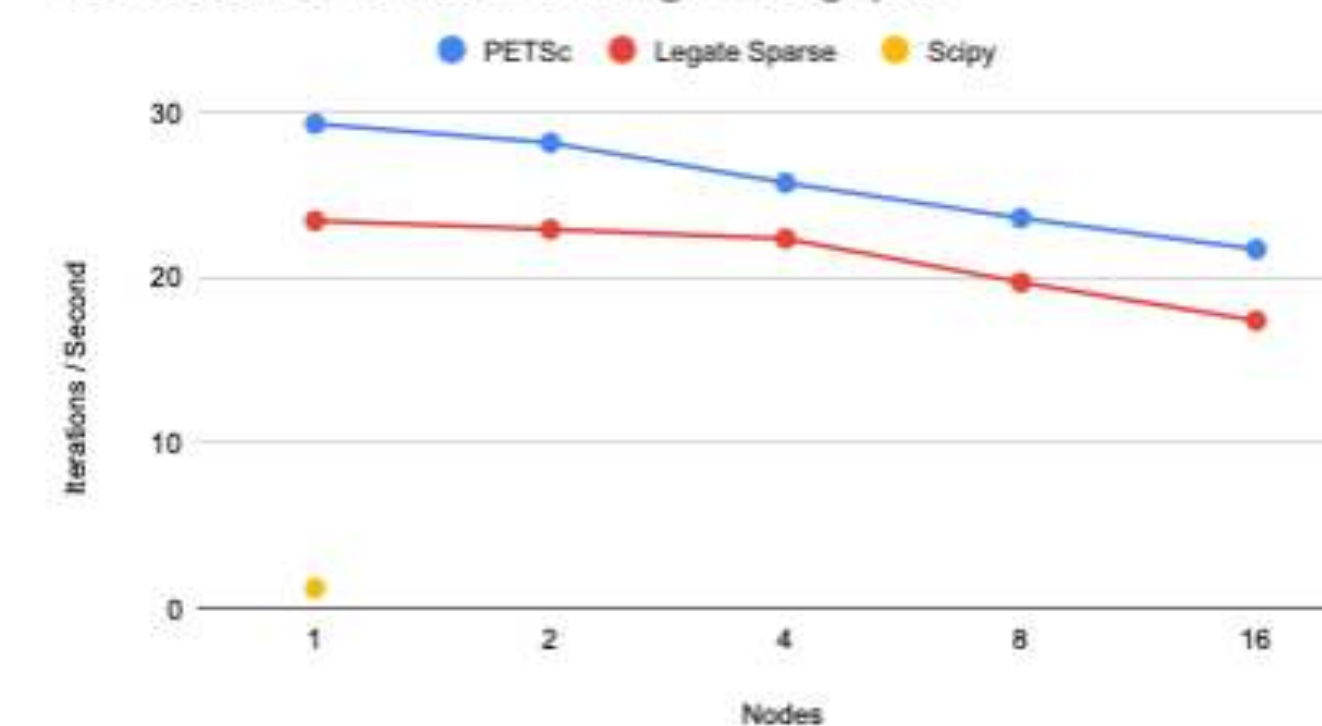
- In-progress NVResearch internship project
- Many modifications from mainline Legate Core required
- A few micro-benchmark results

```
import cunumeric as np
import legate.sparse as sparse
```

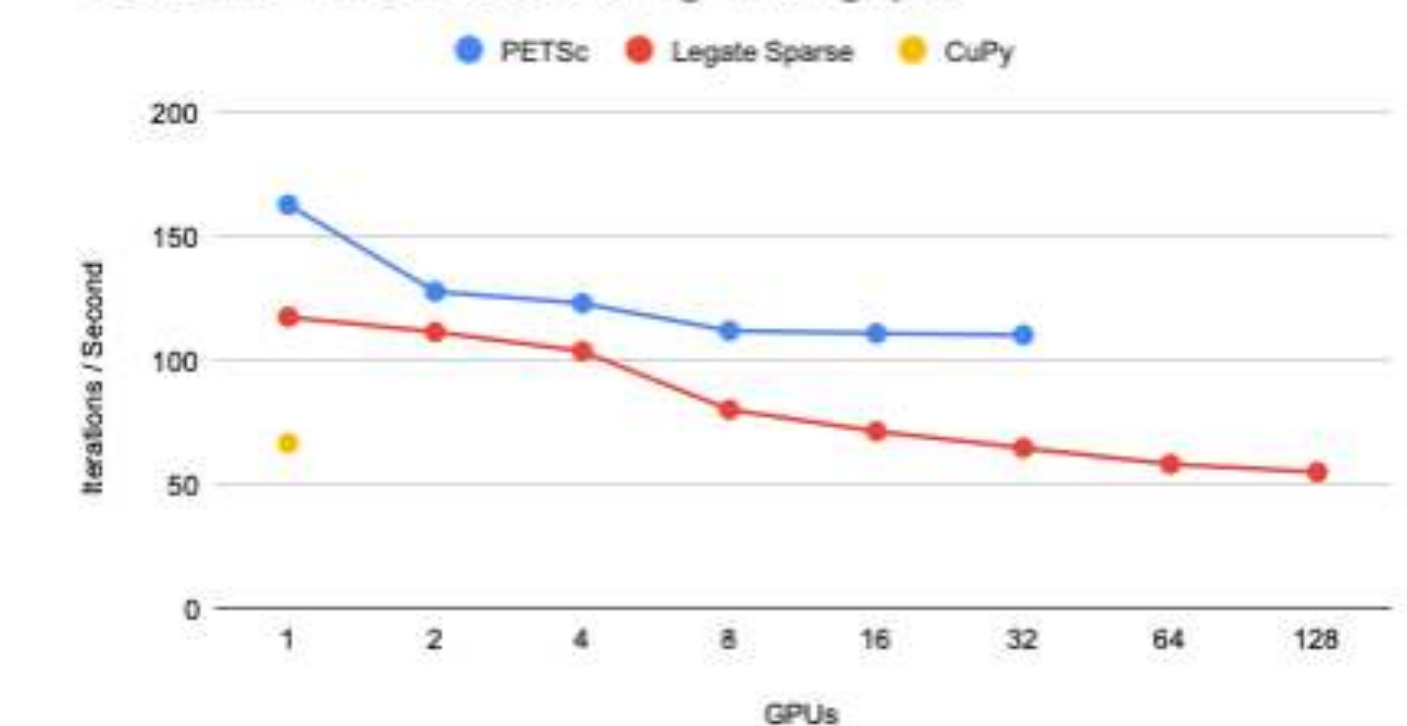
```
A = sparse.diags(...)
y = np.ones(A.shape[1])
x = sparse.linalg.cg(A, y)
assert np.allclose(A @ x, y)
```



CG Solver CPU Weak Scaling Throughput



CG Solver GPU Weak Scaling Throughput



# Where is Legate Sparse now?

Grown so much!

- Official NVIDIA product, backed by eng
  - Compatible with mainline Legate and cuPyNumeric
- Wrote an SC'23 paper
  - Built some non-trivial applications
  - More about Legate Core though...
- Has some real (big) users!



## Legate Sparse: Distributed Sparse Computing in Python

Rohan Yadav  
rohany@cs.stanford.edu  
Stanford University  
USA

Taylor Lee Patti  
tpatti@nvidia.com  
NVIDIA  
USA

Alex Aiken  
aiken@cs.stanford.edu  
Stanford University  
USA

Wonchan Lee  
wonchanl@nvidia.com  
NVIDIA  
USA

Manolis Papadakis  
mpapadakis@nvidia.com  
NVIDIA  
USA

Fredrik Kjolstad  
kjolstad@cs.stanford.edu  
Stanford University  
USA

Melih Elibol  
melibol@nvidia.com  
NVIDIA  
USA

Michael Garland  
mgarland@nvidia.com  
NVIDIA  
USA

Michael Bauer  
mbauer@nvidia.com  
NVIDIA  
USA

### ABSTRACT

The sparse module of the popular SciPy Python library is widely used across applications in scientific computing, data analysis and machine learning. The standard implementation of SciPy is restricted to a single CPU and cannot take advantage of modern distributed and accelerated computing resources. We introduce Legate Sparse, a system that transparently distributes and accelerates unmodified sparse matrix-based SciPy programs across clusters of CPUs and GPUs, and composes with cuNumeric, a distributed NumPy library. Legate Sparse uses a combination of static and dynamic techniques to efficiently compose independently written sparse and dense array programming libraries, providing a unified Python interface for distributed sparse and dense array computations. We show that Legate Sparse is competitive with single-GPU libraries like CuPy and achieves 65% of the performance of PETSc on up to 1280 CPU cores and 192 GPUs of the Summit supercomputer, while offering the productivity benefits of idiomatic SciPy and NumPy.

### ACM Reference Format:

Rohan Yadav, Wonchan Lee, Melih Elibol, Taylor Lee Patti, Manolis Papadakis, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. 2023. Legate Sparse: Distributed Sparse Computing in Python. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*, November 12–17, 2023, Denver, CO, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3581784.3607033>

### 1 INTRODUCTION

Python is a widely used language for data science, machine learning, and scientific computing due to its ease of use and large ecosystem of numerical libraries. This ecosystem includes NumPy [13]

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).  
SC '23, November 12–17, 2023, Denver, CO, USA  
© 2023 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-0109-2/23/11.  
<https://doi.org/10.1145/3581784.3607033>

for dense array-based computations and SciPy's [35] Sparse module for sparse matrix-based computations, both of which serve as foundations for numerous applications and frameworks.

Despite their widespread use, the canonical implementations of NumPy and SciPy target a single CPU node, with only select operations supporting multiple threads. As data set sizes and application computational demands continue to increase, there is a need to target resources more powerful than what a single CPU-only node can provide. Recent work has made great strides in this area for dense array programming systems [5, 22, 24, 29], but the automatic distribution and acceleration of SciPy-based sparse matrix programs has not yet been achieved.

SciPy or CuPy [22] (a single-GPU implementation of NumPy and SciPy) can be paired with a communication library like MPI or NCCL, or a task-based library like Dask Distributed [29] or Ray [21] to enable distributed execution. However, this composition requires the user to manually partition and communicate data, resulting in non-trivial code modification and necessitating distributed programming expertise. The industry-standard sparse linear algebra systems PETSc [3, 20] and Trilinos [34] expose Python wrappers around their low-level C/C++ APIs. While these APIs provide many high-level sparse matrix operations, they require programmers to reason about data distribution and data movement, a level of expertise many programmers do not have.

Our goal in this work is to develop a system that scales unmodified SciPy Sparse programs across distributed machines with good performance, and efficiently composes with cuNumeric [5], a distributed NumPy library. This system would provide the familiar dense and sparse array programming interfaces to allow users with and without expertise in distributed programming to rapidly prototype distributed applications and scale these applications to the size of machine needed to process their datasets. In this paper, we explore the large design space encompassed by these constraints, and demonstrate one design point that achieves our goals.

Achieving our goal of building a distributed and heterogeneous sparse array programming library that achieves both high performance as well as composability with an external dense array programming library requires solving a global problem of performance composability at multiple layers of the software stack. First,



# NPCI Graph Analysis Workload

SpGEMM's all the way

- Trying to detect fraud in transaction data
- SpGEMM's to perform walks of transaction graphs
- Big speedups over CPU clusters, easy to use!
  - Composition with cuPyNumeric was important

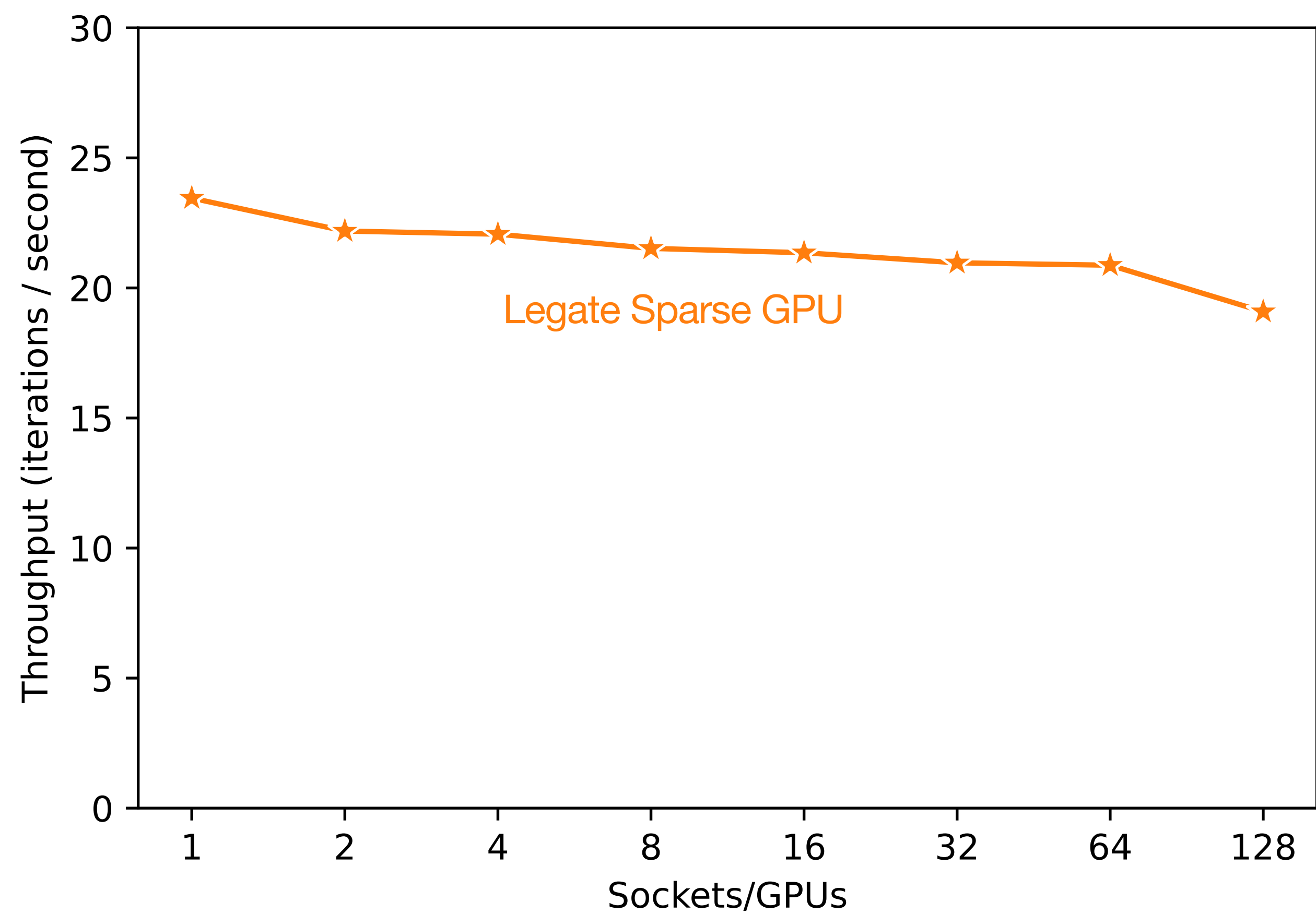
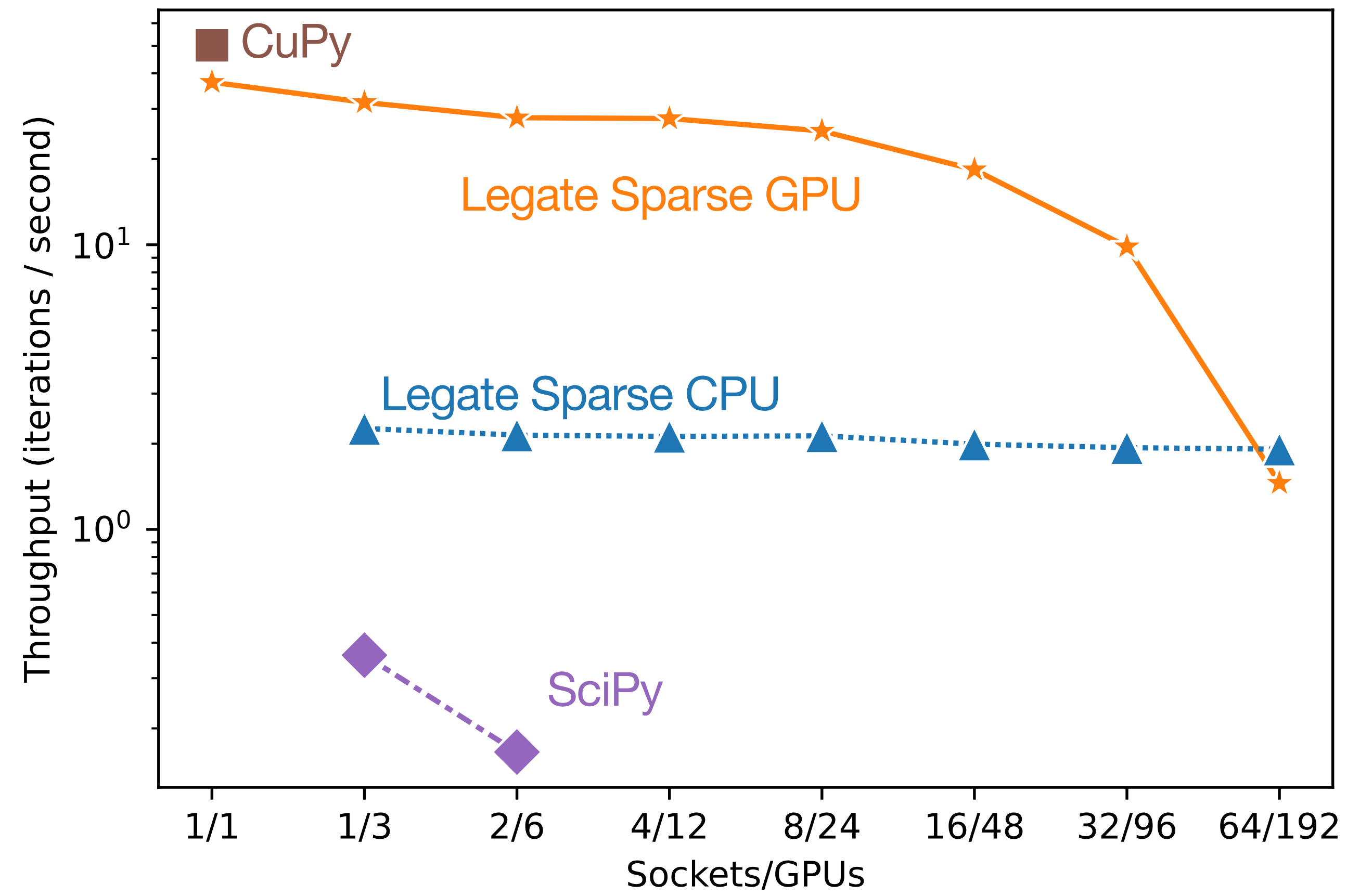


# Scientific Computations

Solving equations

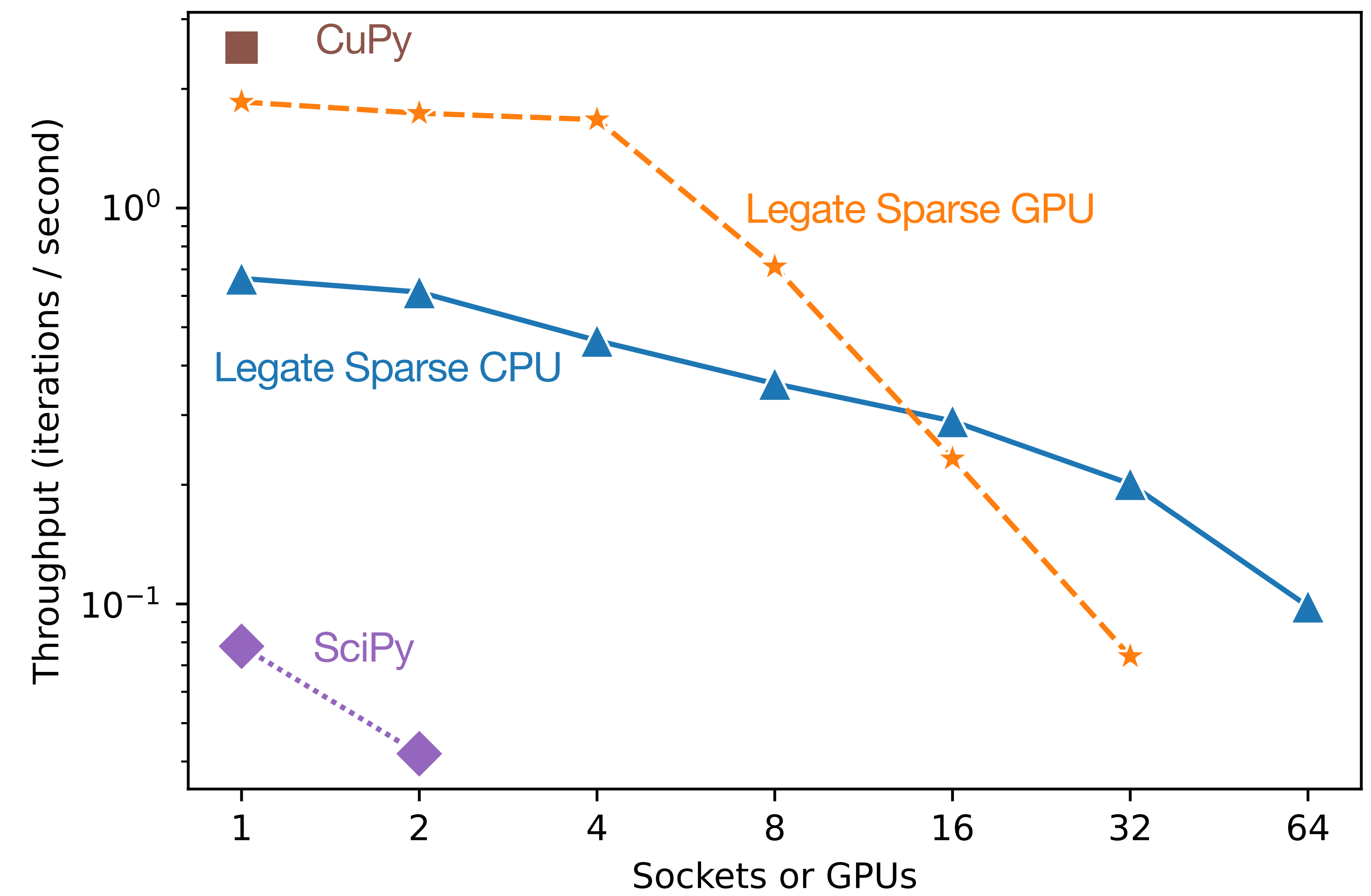
## Multi-grid Solver

(Summit / Selene)



## Quantum Simulation

(Summit)





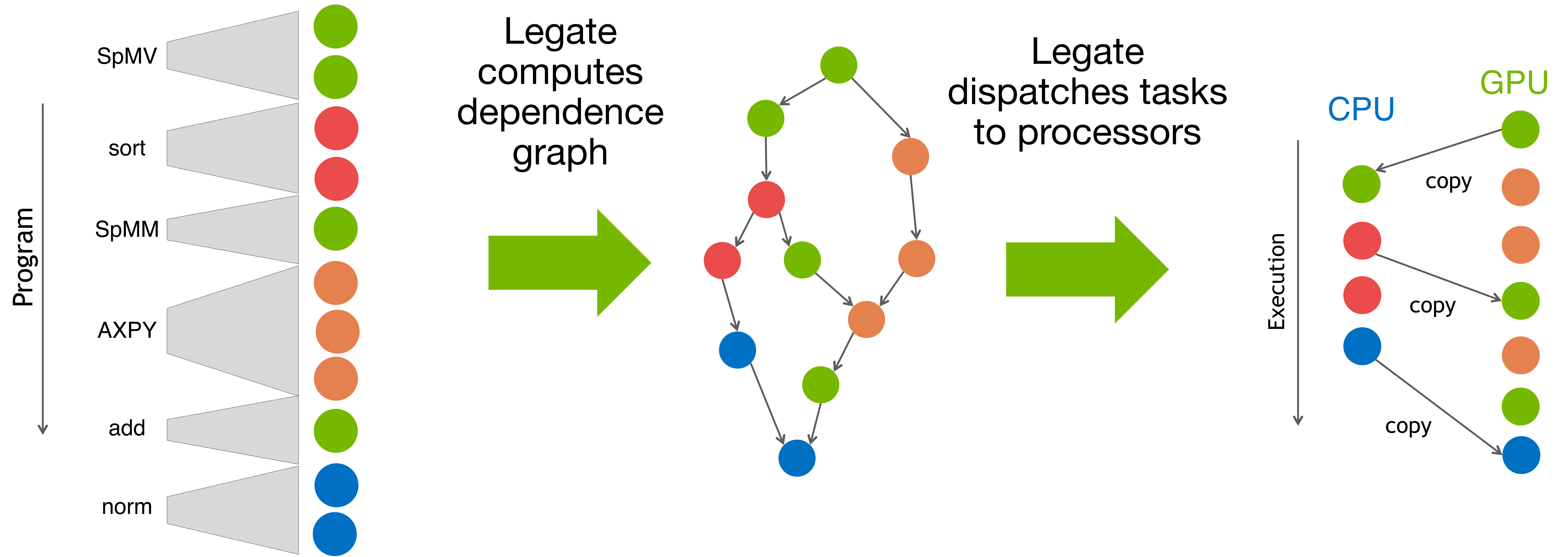
# Legate Sparse Internals



# Legate Core Constructs a Coherent View of Execution

Leveraging Legate Core + Legion

Legate Sparse and cuPyNumeric independently issue Legate tasks

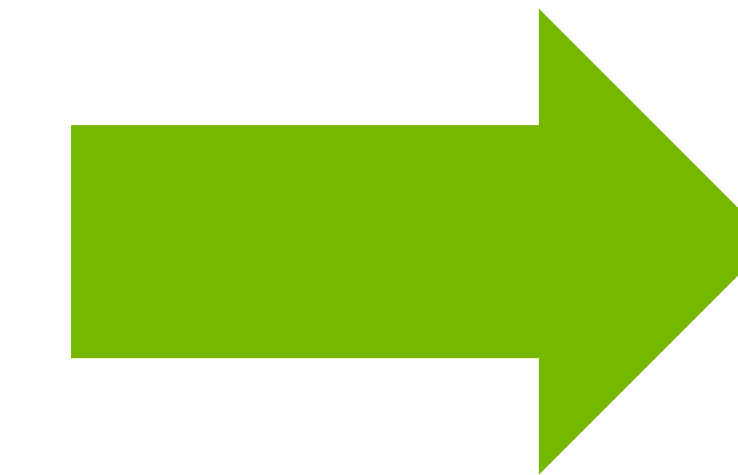
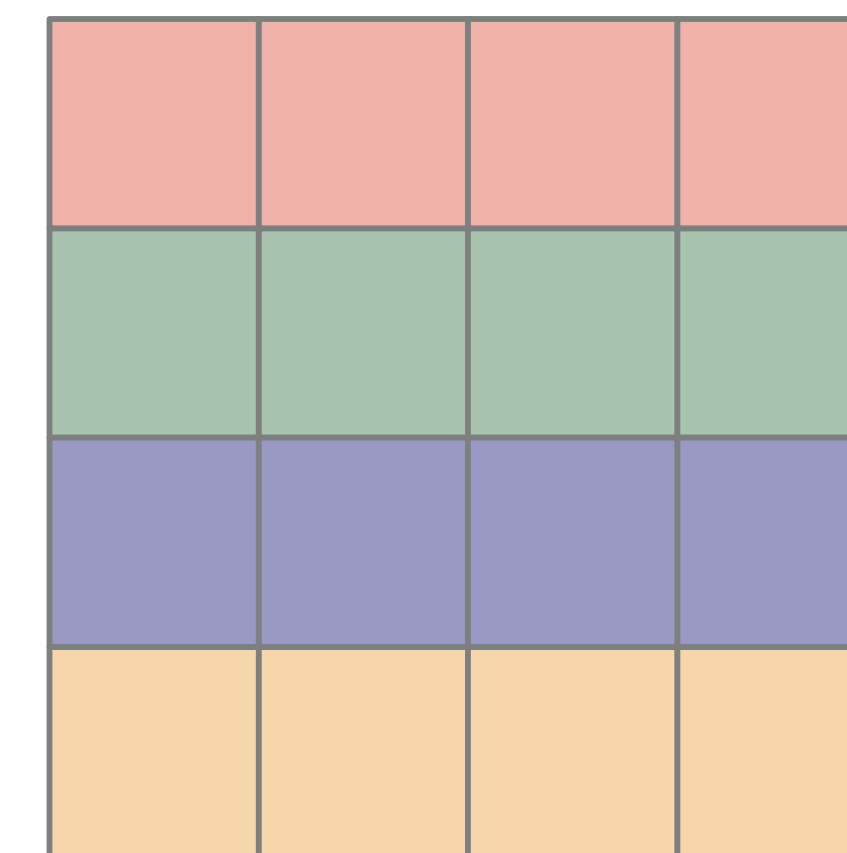


# Partitioning Constraints Enables Composable Partitioning

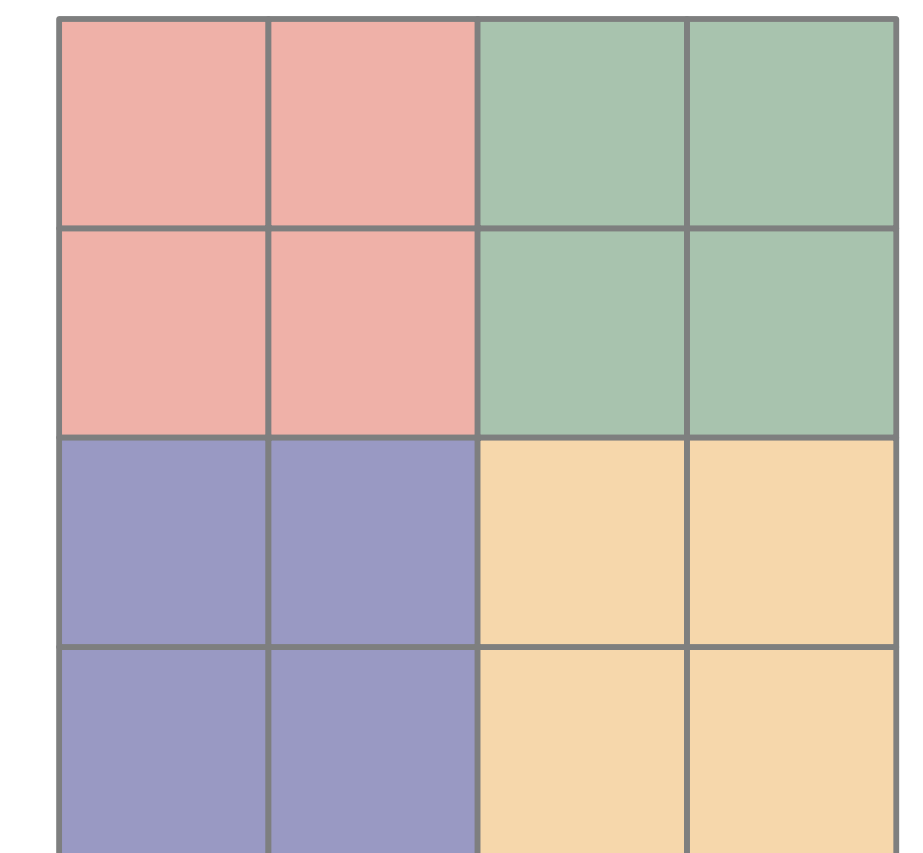
Also simplifies task launching...

- Standard Legion and MPI programs embed partitioning choices into the computation
- Delaying partitioning choices to runtime allows libraries to be flexible
- Constraints expose application knowledge to runtime

lib1.foo()



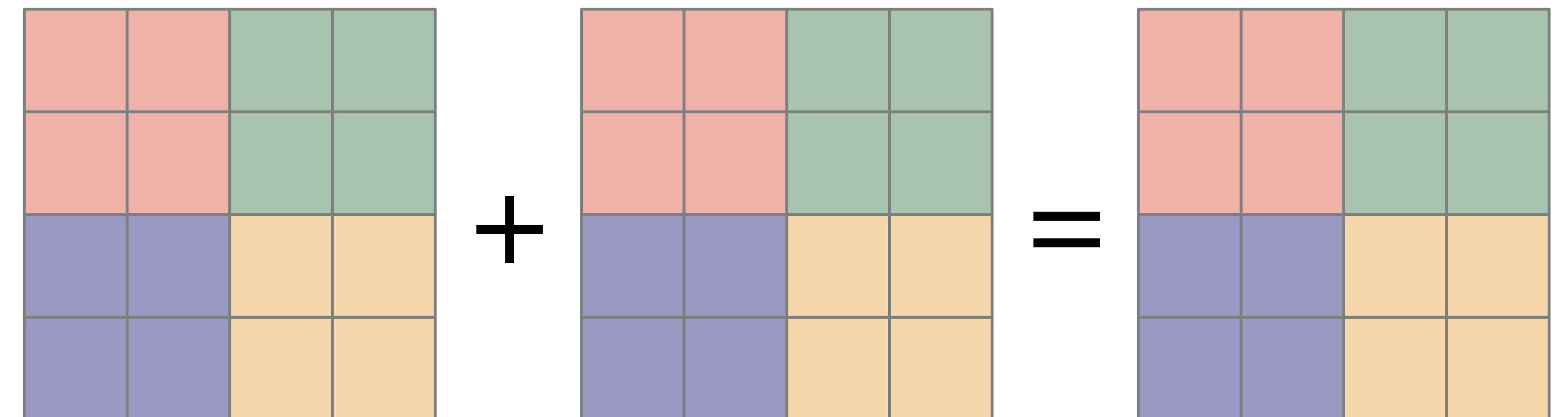
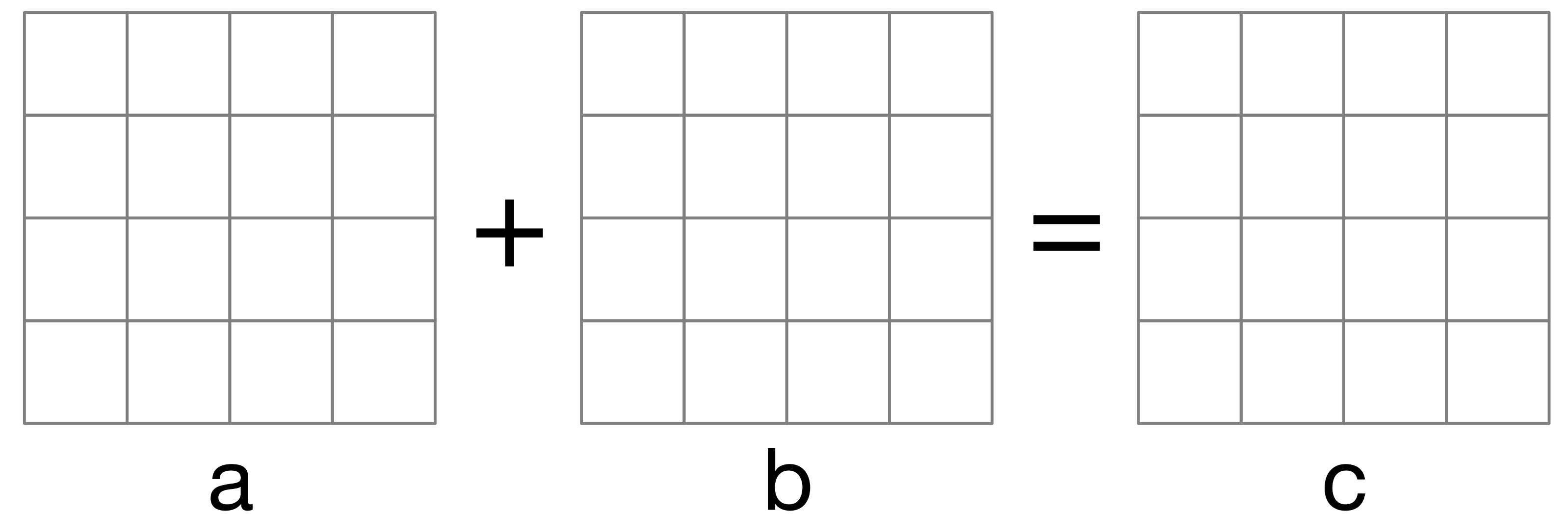
lib2.bar()



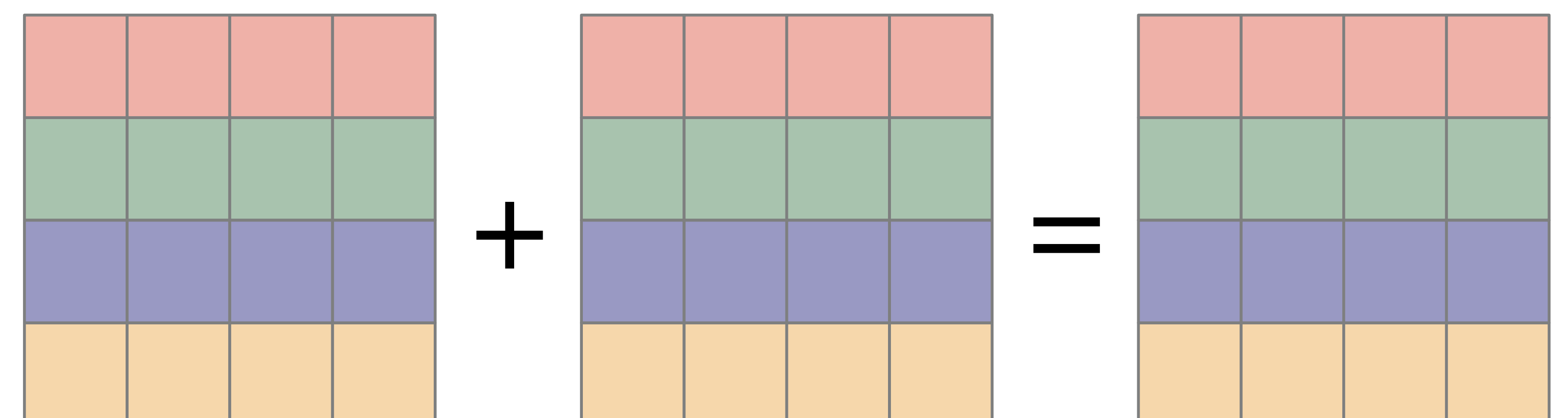
# Simple Example: Element-wise Addition

Library internals, not user facing

```
task = create_task(ADD)
task.add_output(c)
task.add_input(a, b)
task.add_alignment(a, b, c)
task.execute()
```

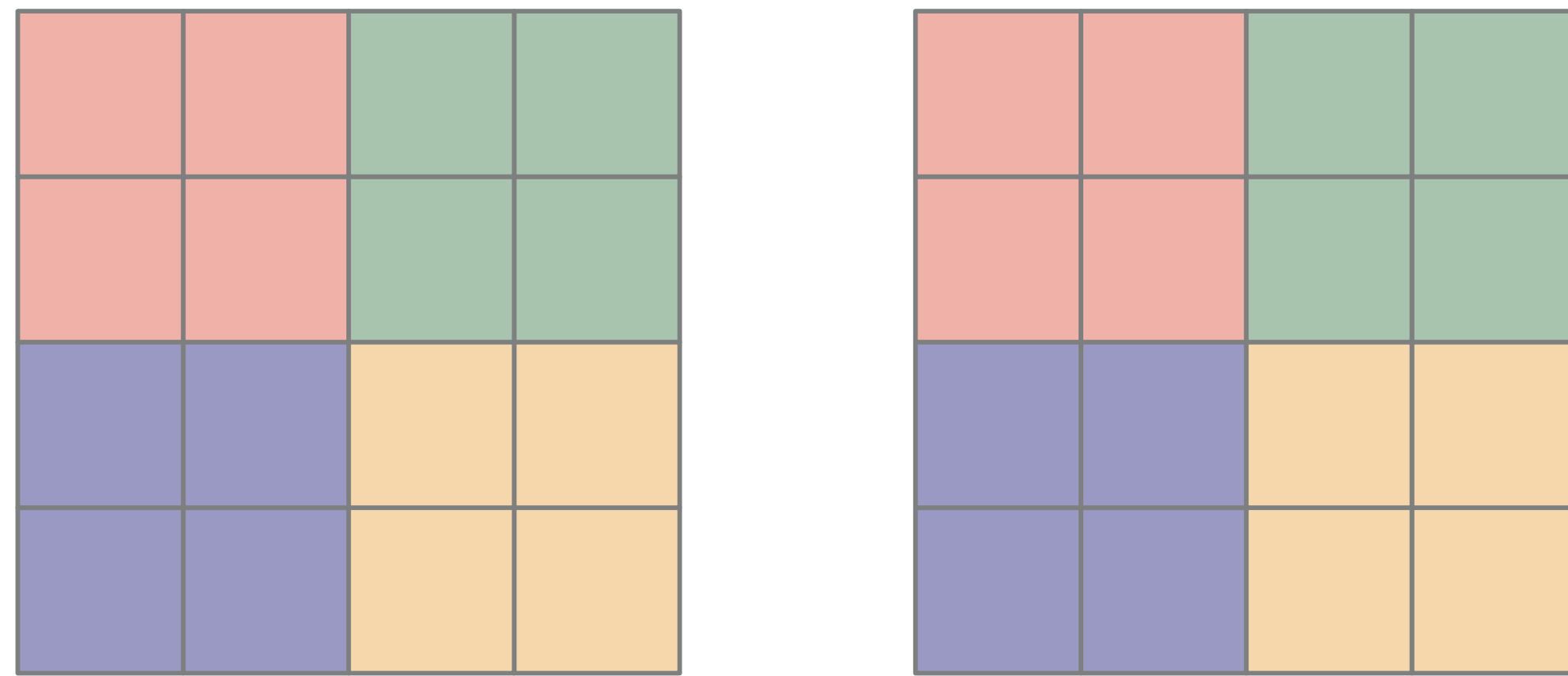


Both satisfy  
constraint!

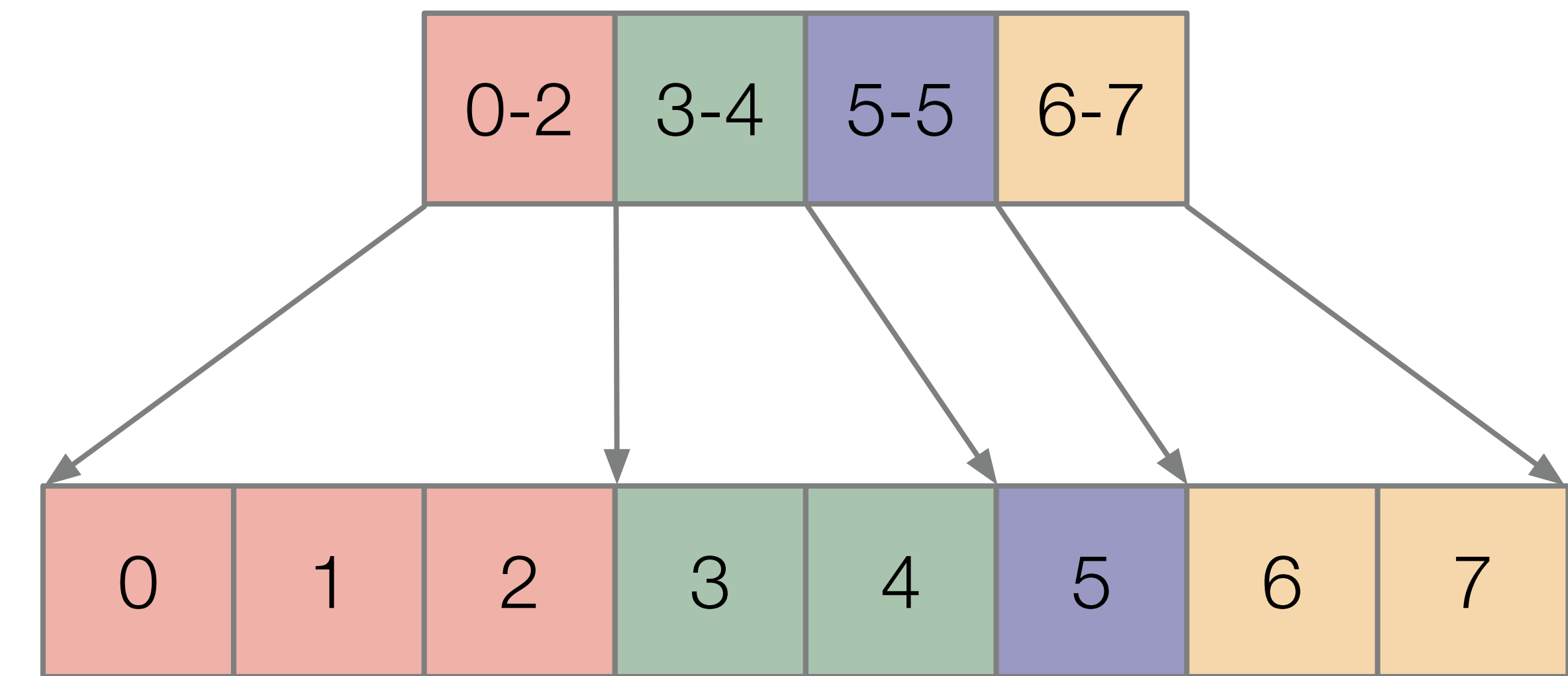
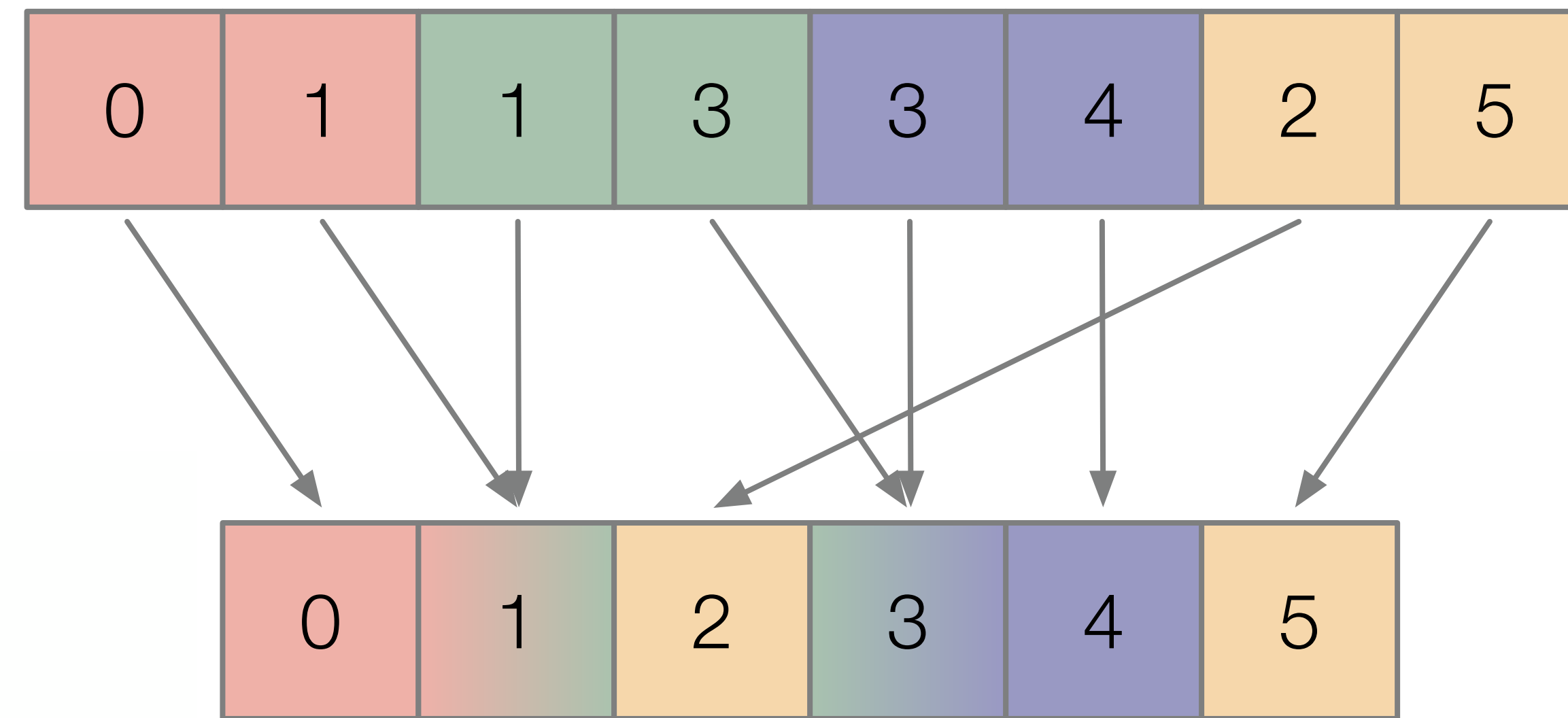


# Relevant Subset of Partitioning Constraints

Used for the next example



align(R1, R2)



image(R1, R2)

# CSR Sparse Matrix Representation

Used for the next example

a	b		c
	d		e
f			
g			h

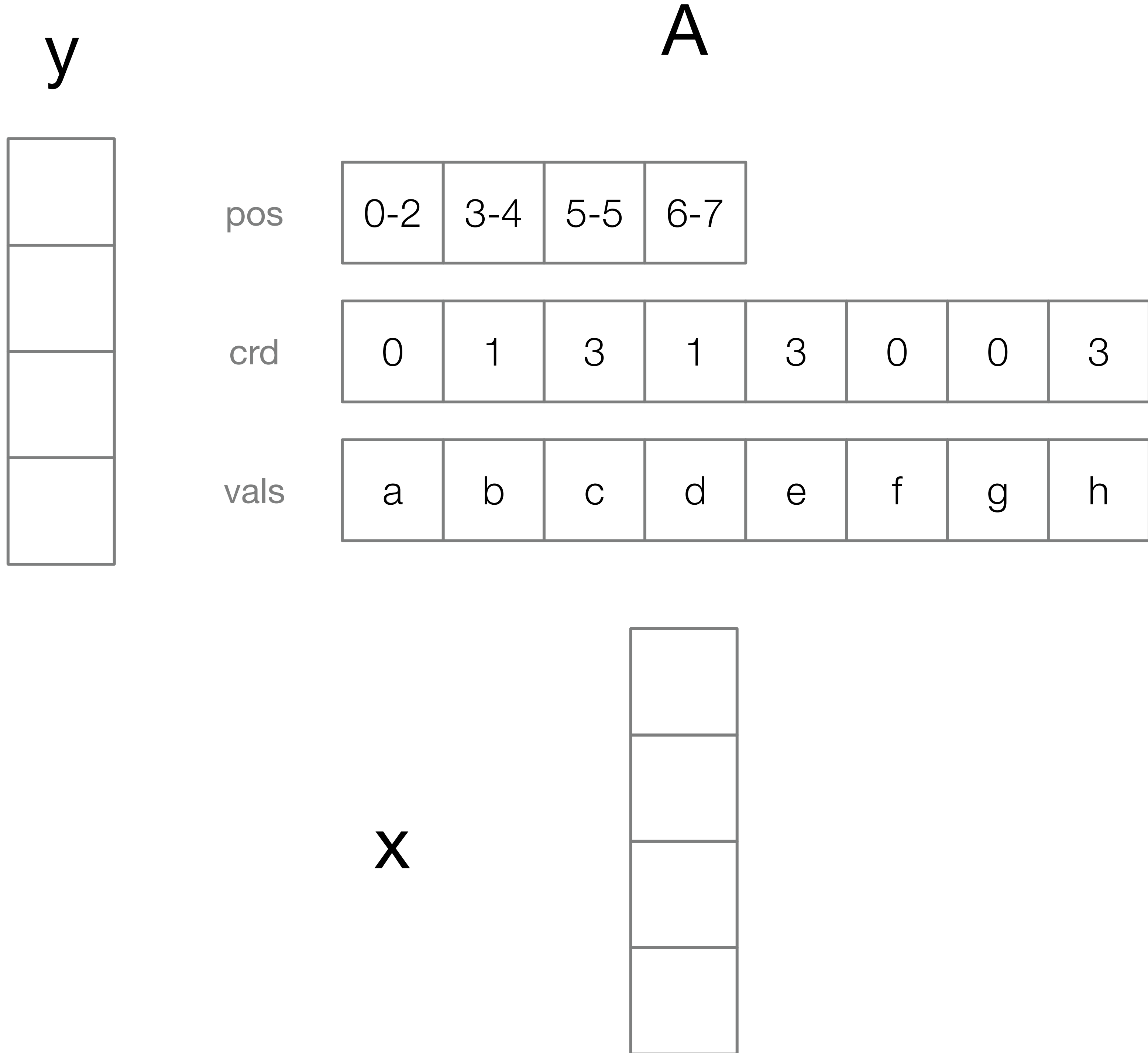
pos	0-2	3-4	5-5	6-7				
crd	0	1	3	1	3	0	0	3
vals	a	b	c	d	e	f	g	h

# Distributed SpMV Implementation Example

Library internals

$$y_i = \sum_j A_{ij} x_j$$

```
task = create_task(SPMV)
task.add_output(y)
task.add_input(A.pos, A.crd, A.vals, x)
task.execute()
```

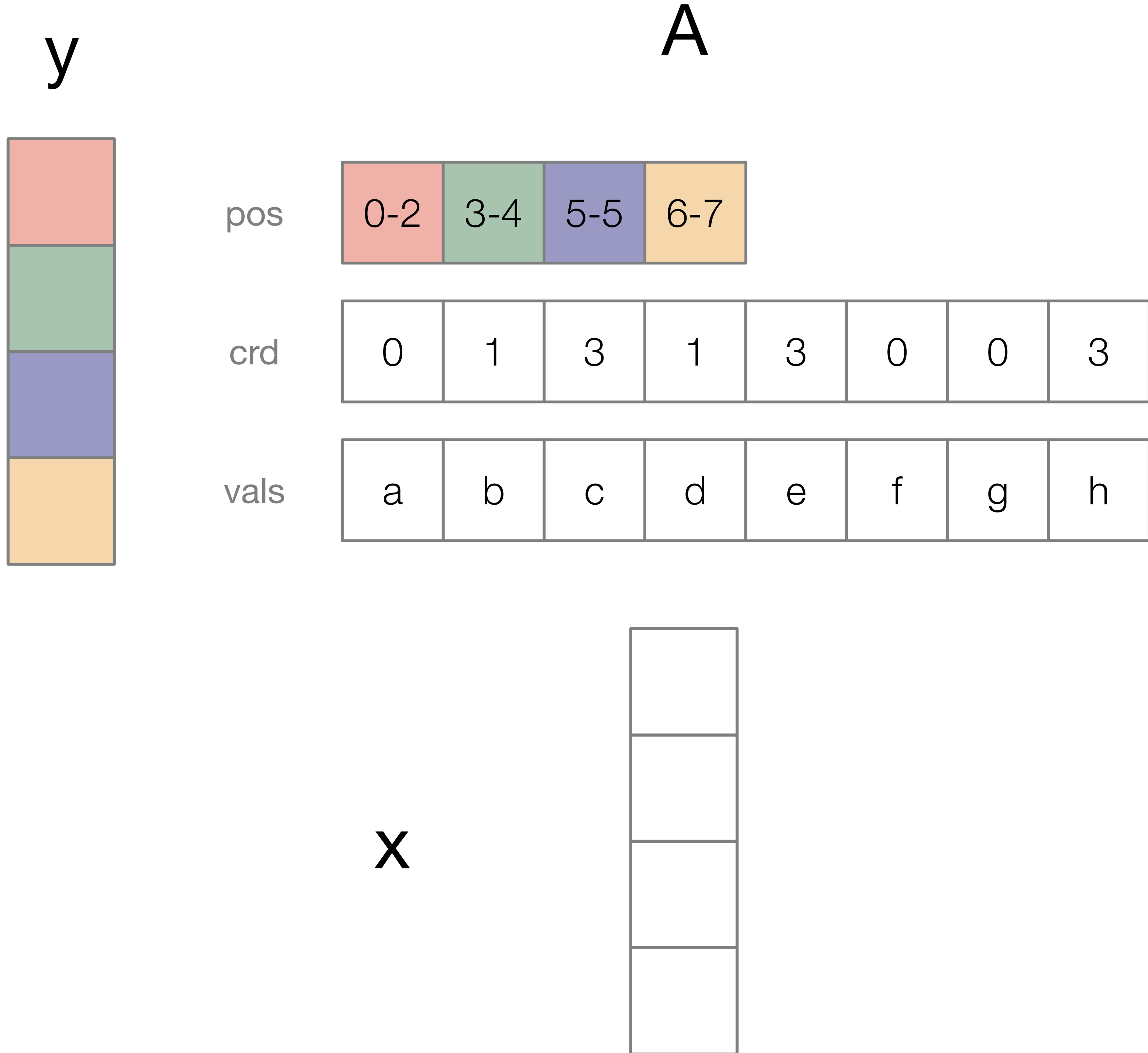


# Distributed SpMV Implementation Example

Library internals

$$y_i = \sum_j A_{ij} x_j$$

```
task = create_task(SPMV)
task.add_output(y)
task.add_input(A.pos, A.crd, A.vals, x)
task.add_alignment(y, A.pos)
task.execute()
```



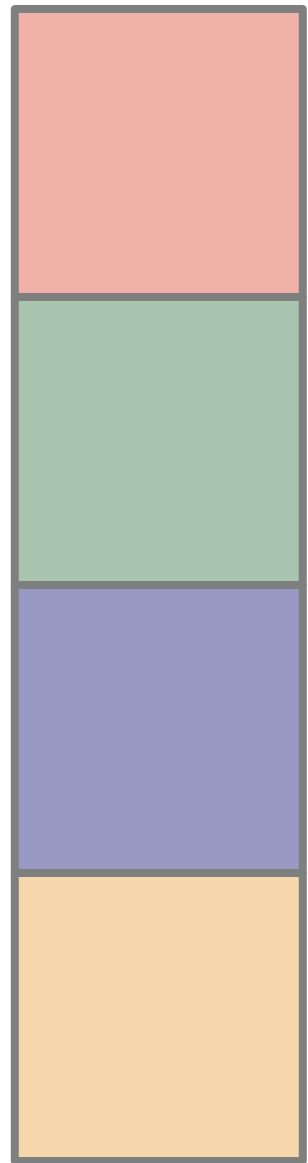
# Distributed SpMV Implementation Example

Library internals

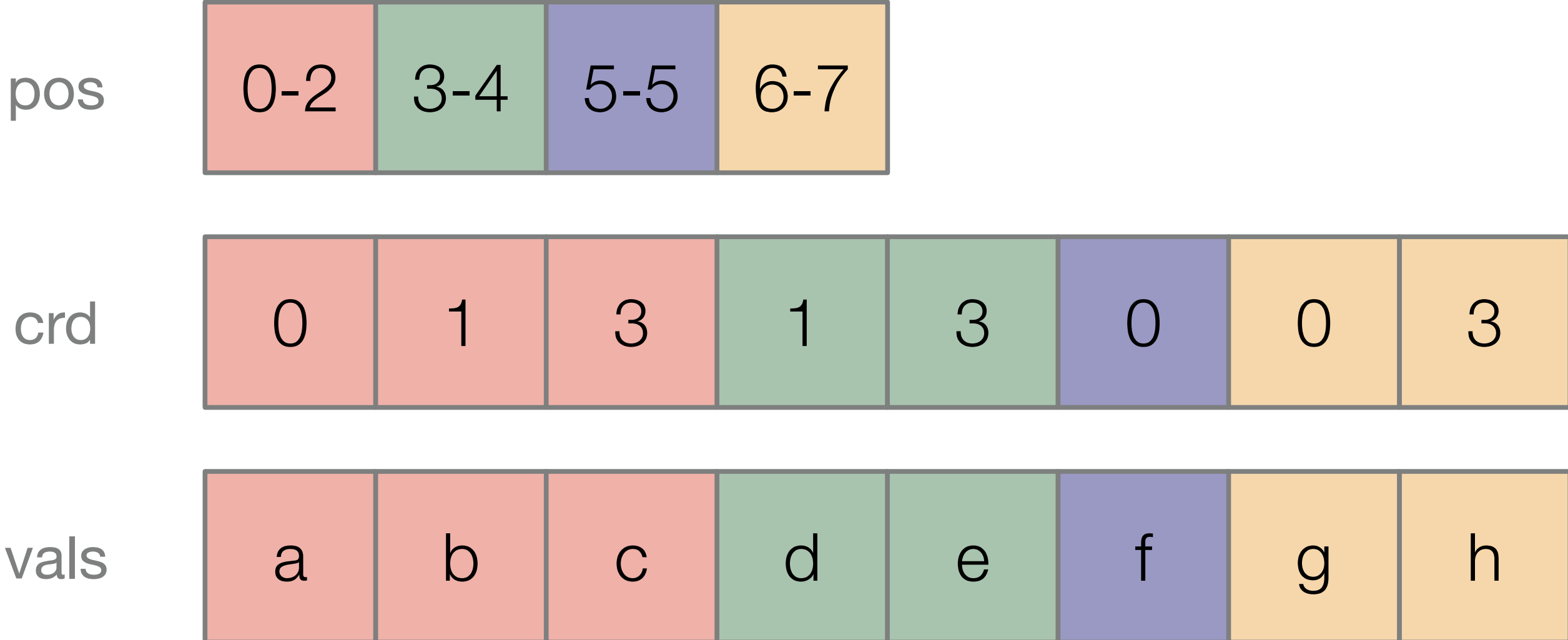
$$y_i = \sum_j A_{ij} x_j$$

```
task = create_task(SPMV)
task.add_output(y)
task.add_input(A.pos, A.crd, A.vals, x)
task.add_alignment(y, A.pos)
task.add_image(A.pos, A.crd)
task.add_image(A.pos, A.vals)
task.execute()
```

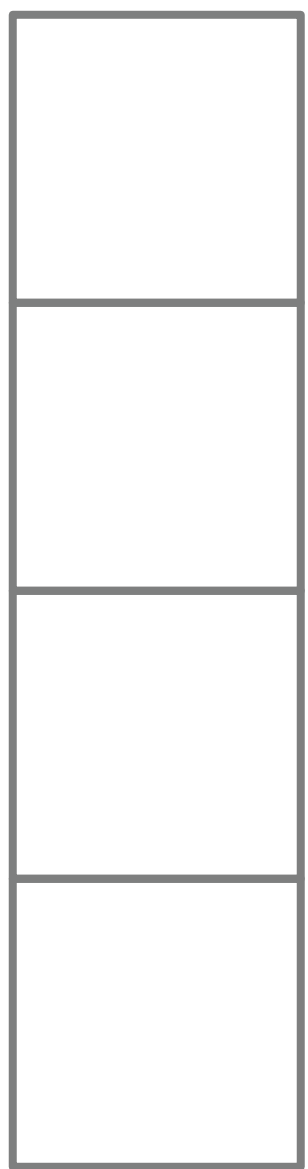
y



A



X





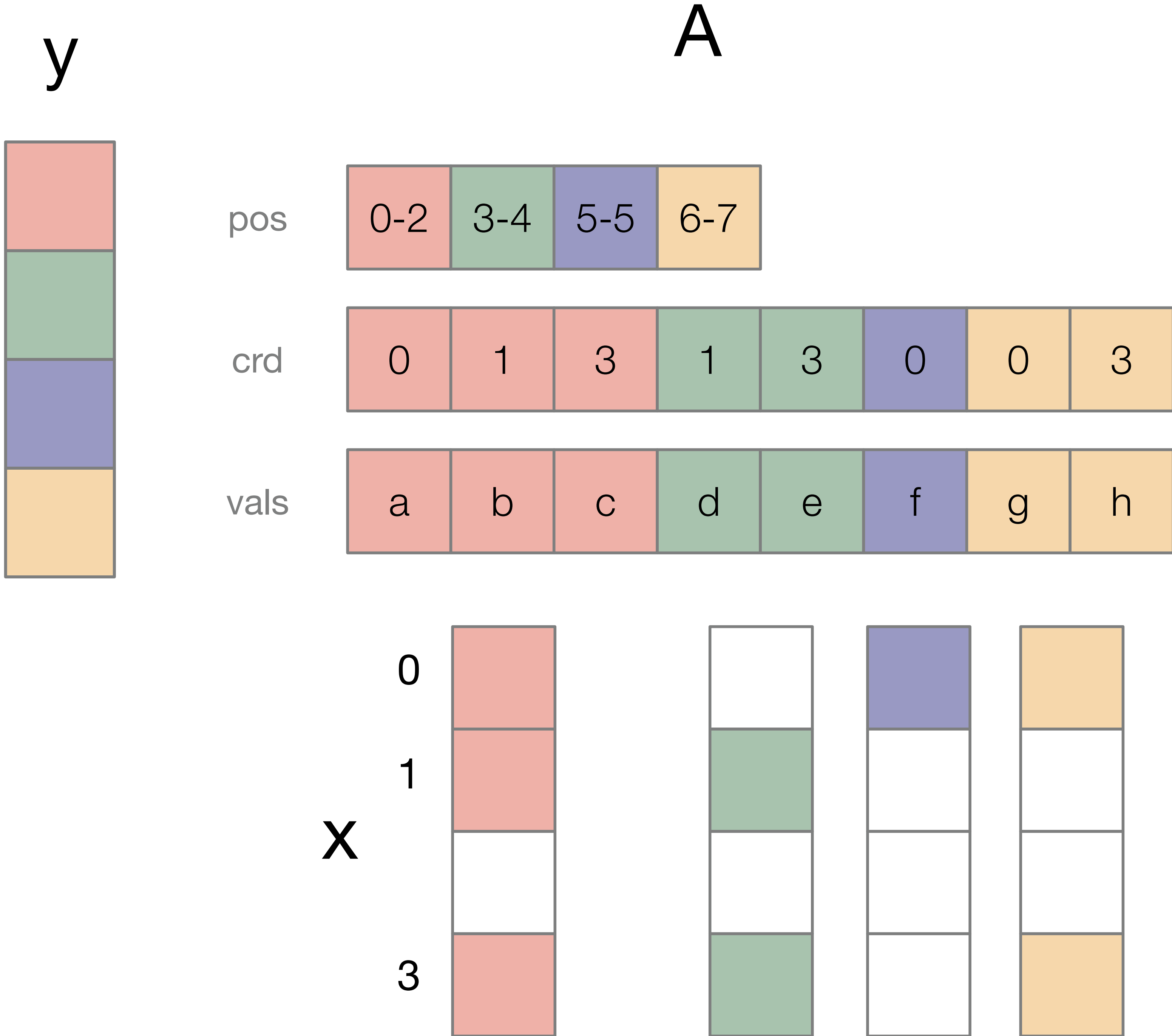
# Distributed SpMV Implementation Example

Library internals

$$y_i = \sum_j A_{ij} x_j$$

```

task = create_task(SPMV)
task.add_output(y)
task.add_input(A.pos, A.crd, A.vals, x)
task.add_alignment(y, A.pos)
task.add_image(A.pos, A.crd)
task.add_image(A.pos, A.vals)
task.add_image(A.crd, x)
task.execute()
    
```



# Initial Legate Sparse Development Process

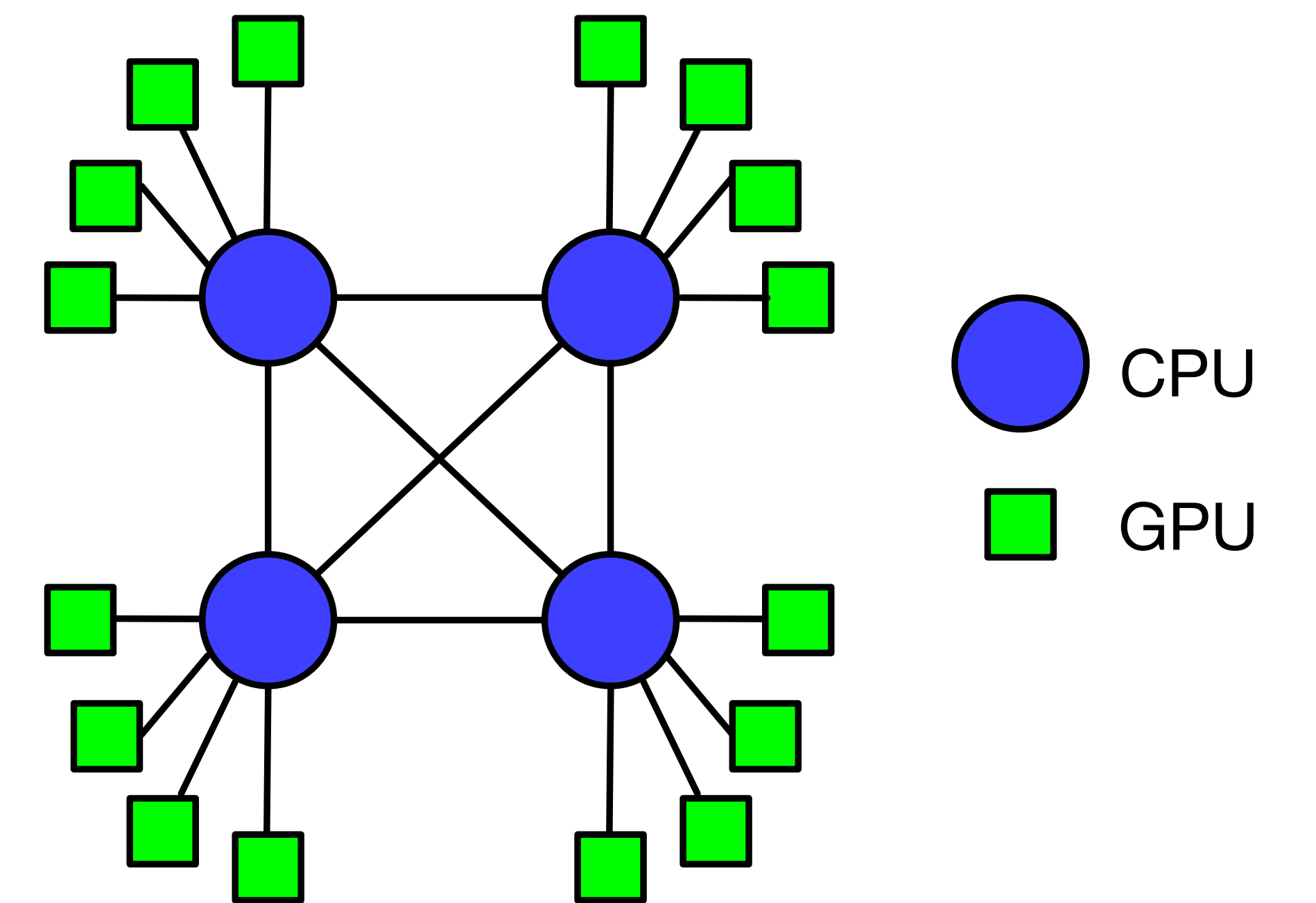
Back when it was research

**What to compute**  
*(Tensor Index Notation)*

**How data is compressed**  
*(Format Language)*

**How computation is distributed**  
*(Scheduling Language)*

# DISTAL



# Initial Legate Sparse Development Process

Back when it was research

Performance engineer develops high-performance schedule

```
divide(i, io, ii, procs)
distribute(io)
communicate(io, {y, A, x})
parallelize(ii, CPUThread)
```

Mechanically lift partitions to constraints

```
task = create_task(SPMV)
task.add_output(y)
task.add_input(mat.pos, mat.crd, mat.vals, x)
task.add_alignment(y, mat.pos)
task.add_image(mat.pos, mat.crd)
task.add_image(mat.pos, mat.vals)
task.add_image(mat.crd, x)
task.execute()
```

DISTAL generates Legion code

```
runtime->create_index_partition(...)
runtime->partition_by_image(...)
...
runtime->execute_index_space(...)
```

```
void task_body(...) {
  kernel<<<>>(...)
}
```

DISTAL generated  
**45% (2854/6135 LOC)** of C++/CUDA  
**12% (697/5748 LOC)** of Python

# Initial Legate Sparse Development Process

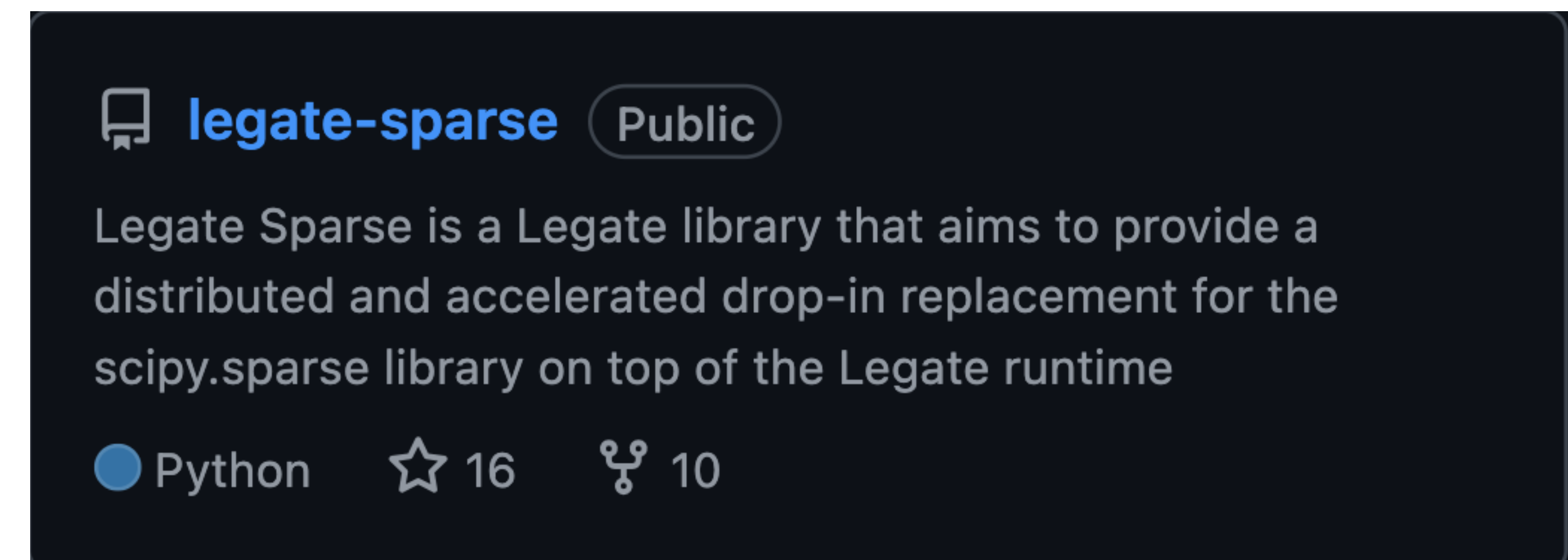
Back when it was research

- ~156 functions ported from CuPy/SciPy by just changing imports
  - Legate ecosystem enables rapid + “free” reuse of existing library components!

# Summary

<https://github.com/nv-legate/legate-sparse>

- Distributed+Accelerated drop-in replacement for SciPy Sparse computations
  - Try it today!
- The Legate ecosystem enables composition with cuPyNumeric + other external libraries
- Legate Core abstractions simplify development over raw Legion



A screenshot of the GitHub repository page for `legate-sparse`. The repository is public and is categorized as Python. It has 16 stars and 10 forks. The description states: "Legate Sparse is a Legate library that aims to provide a distributed and accelerated drop-in replacement for the `scipy.sparse` library on top of the Legate runtime".

## Use conda packages

The `legate-sparse` conda package already depends on `legate` and `cupynumeric`, and it will install these dependencies automatically.

To create a new environment and install:

```
conda create -n myenv -c legate -c conda-forge legate-sparse
```

or to install in an existing environment:

```
conda install -c legate -c conda-forge legate-sparse
```

