Maximizing Performance Through Memory Hierarchy-Driven Data Layout Transformations

SC22
Dallas, TX | hpc accelerates.

Benjamin Sepanski*
3rd Year Ph.D. Student
Computer Science Department
University of Texas at Austin

Tuowen Zhao
Computer Science Department
University of Utah

Hans Johansen, Samuel Williams
Computational Research Division
Lawrence Berkeley Labs

COMPUTING SCIENCES RESEARCH
LAWRENCE BERKELEY NATIONAL LABORATORY

SCHOOL OF COMPUTING

The University of Texas at Austin
Computer Science

ECP

DOE CSGF

Acknowledgements

- This research was supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02- 05CH11231, the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number(s) DE-SC0021110.

- This research used resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

- This research was funded by PROTEAS-TUNE, which paid for my GSR in summer 2021, and DOE CSGF/The Krell Institute

Outline

- Introduction: Bricks
- High-Dimensional Bricks
- GENE Microbenchmarks

Introduction: Bricks

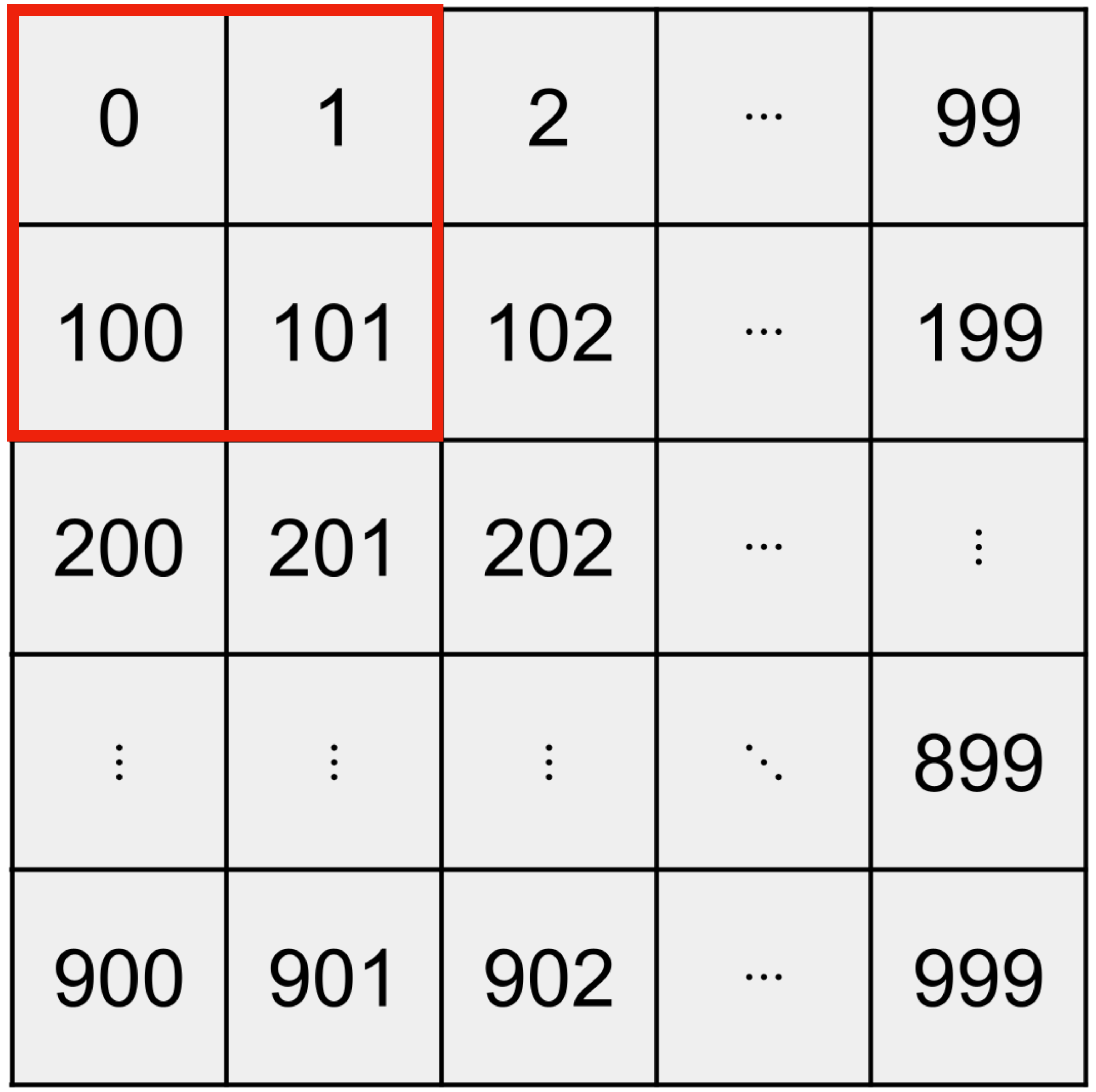
Standard Array Layouts

- Standard array layouts can suffer from poor spatial locality
- Forced to recover locality by loop optimizations
  - Tiling
  - Polyhedral analysis
  - Scheduling languages
- Search for iteration order to support parallelization, locality, vectorization

0	1	2	⋯	99
100	101	102	⋯	199
200	201	202	⋯	⋮
⋮	⋮	⋮	⋱	899
900	901	902	⋯	999

Standard Array Layouts

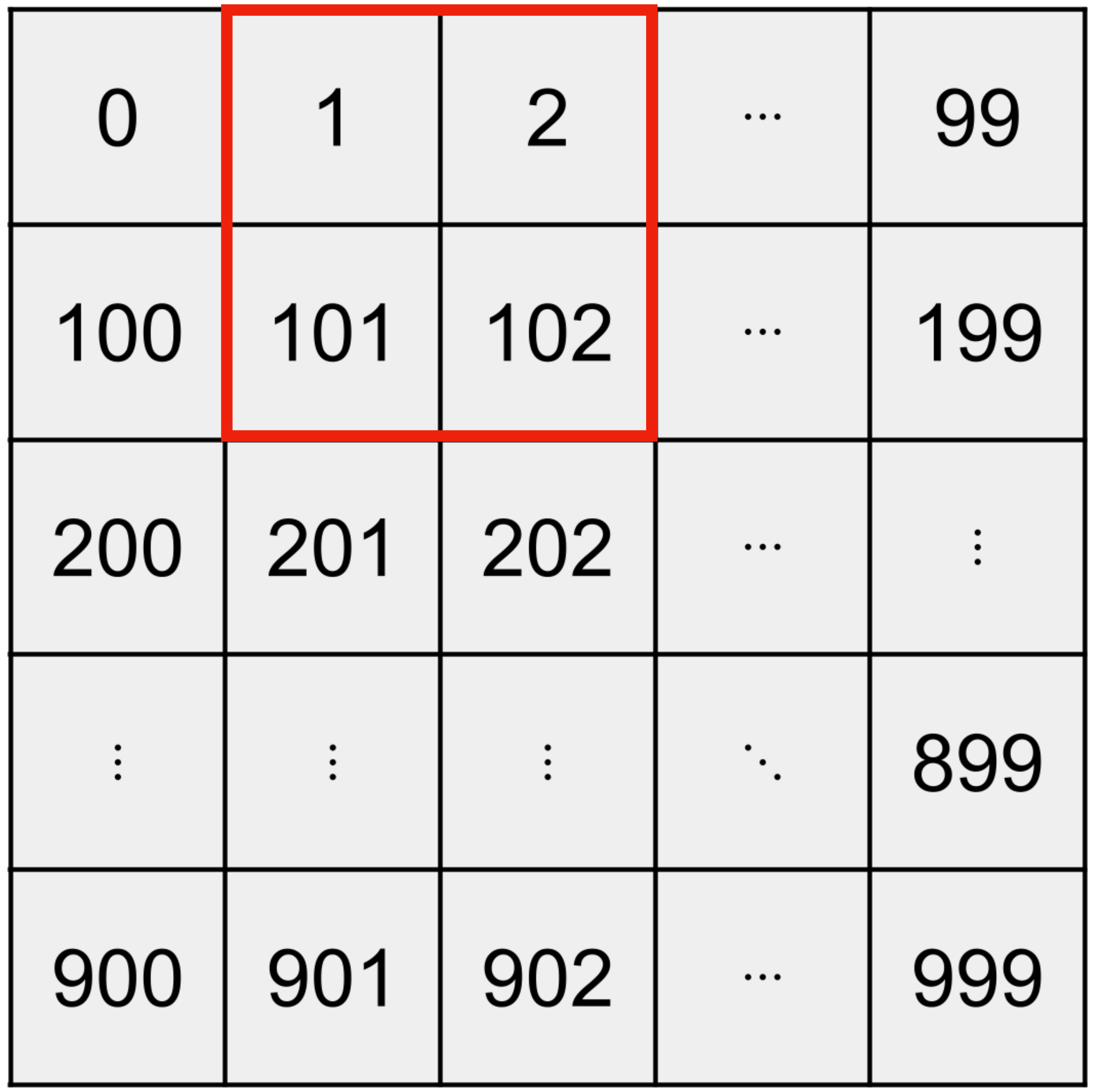
- Standard array layouts can suffer from poor spatial locality
- Forced to recover locality by loop optimizations
  - Tiling
  - Polyhedral analysis
  - Scheduling languages
- Search for iteration order to support parallelization, locality, vectorization



0	1	2	⋯	99
100	101	102	⋯	199
200	201	202	⋯	⋮
⋮	⋮	⋮	⋱	899
900	901	902	⋯	999

Standard Array Layouts

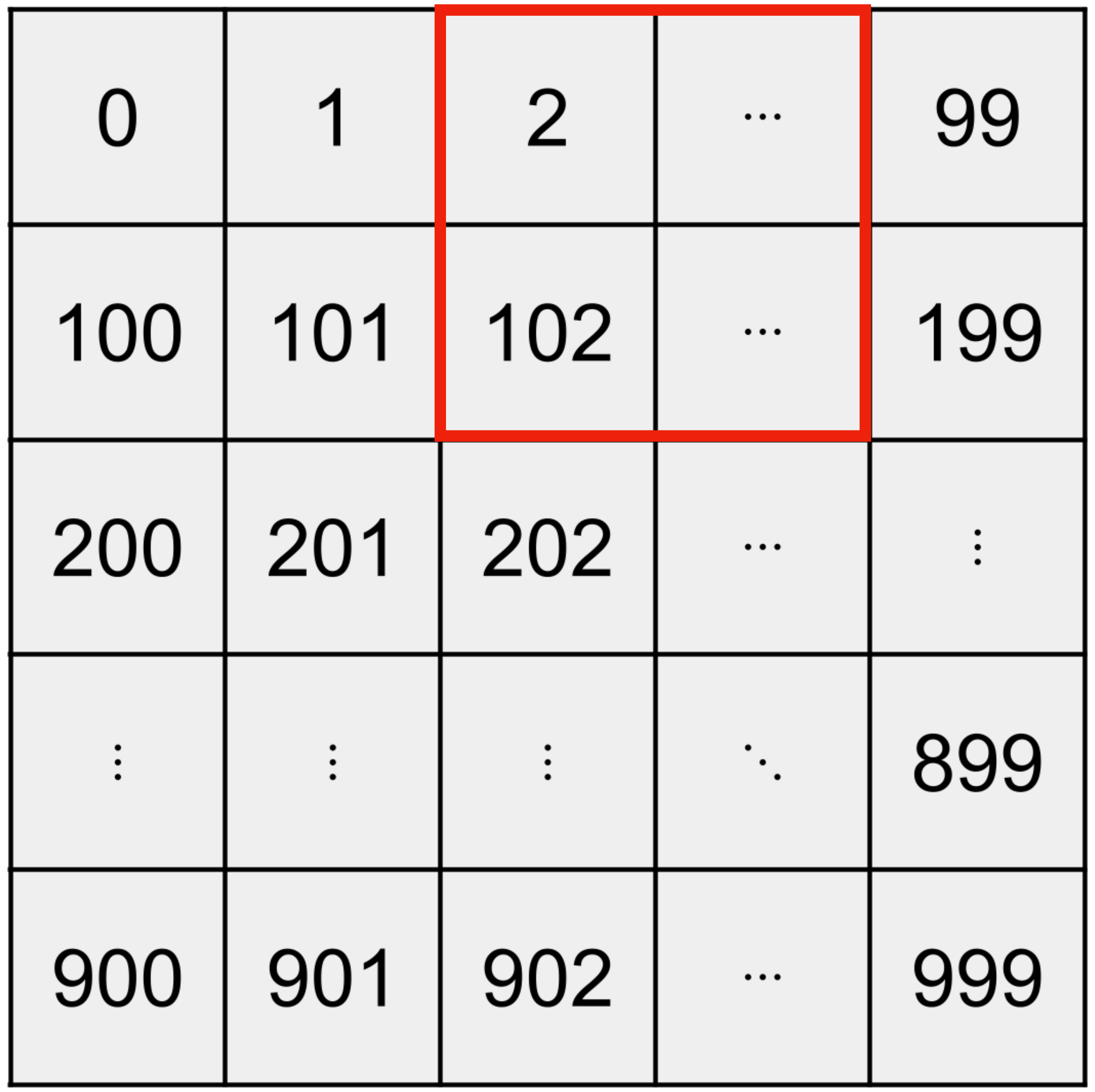
- Standard array layouts can suffer from poor spatial locality
- Forced to recover locality by loop optimizations
  - Tiling
  - Polyhedral analysis
  - Scheduling languages
- Search for iteration order to support parallelization, locality, vectorization



0	1	2	⋯	99
100	101	102	⋯	199
200	201	202	⋯	⋮
⋮	⋮	⋮	⋱	899
900	901	902	⋯	999

Standard Array Layouts

- Standard array layouts can suffer from poor spatial locality
- Forced to recover locality by loop optimizations
  - Tiling
  - Polyhedral analysis
  - Scheduling languages
- Search for iteration order to support parallelization, locality, vectorization

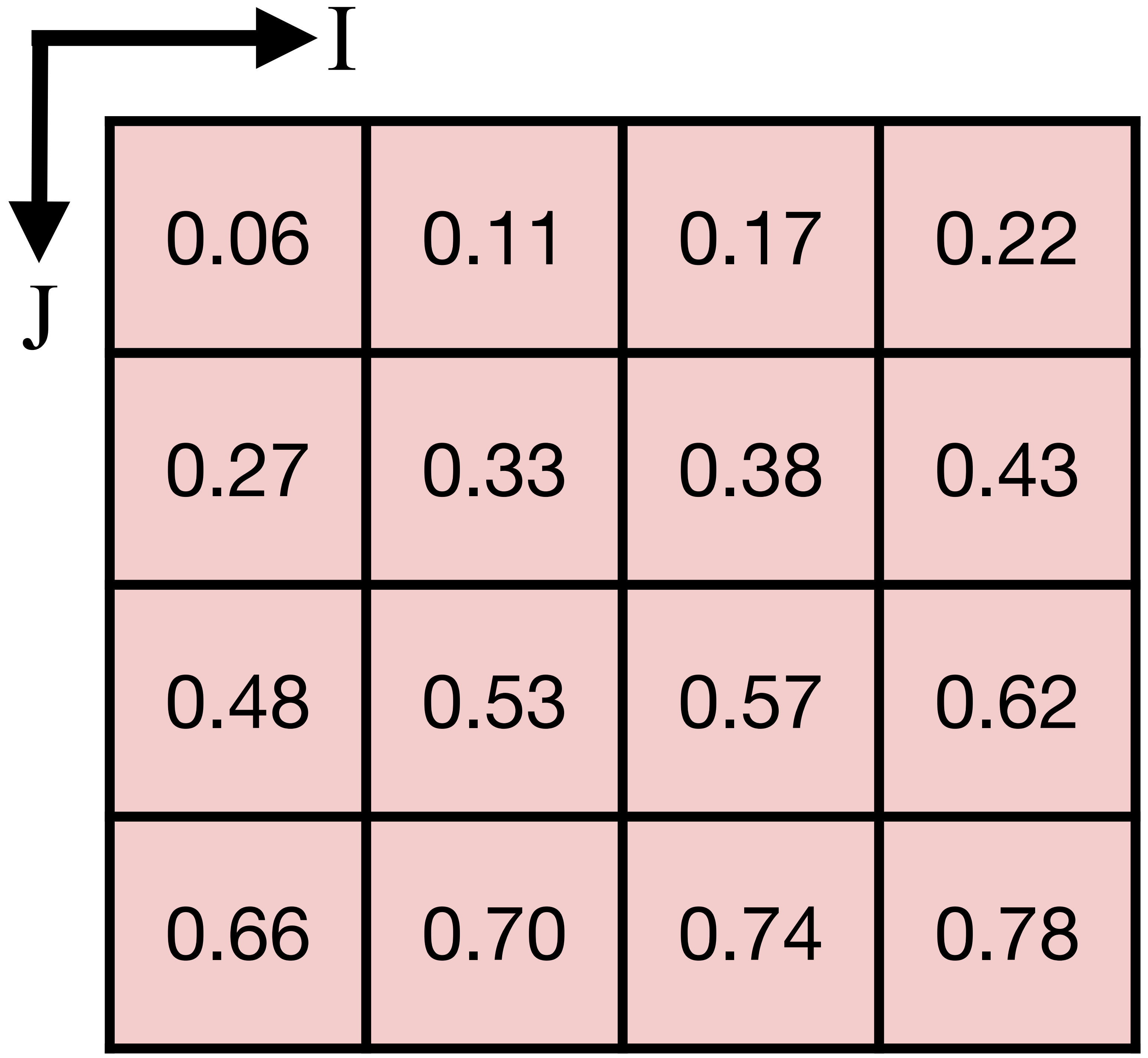


0	1	2	⋯	99
100	101	102	⋯	199
200	201	202	⋯	⋮
⋮	⋮	⋮	⋱	899
900	901	902	⋯	999

The Bricks Layout

What is a Brick?

- Each Brick is a small, **fixed-**size multidimensional array stored **contiguously** in memory
- Each Brick is a “unit” of locality
- Fine-grained parallelism occurs **inside** each Brick
- Coarse-grained parallelism occurs **across** Bricks

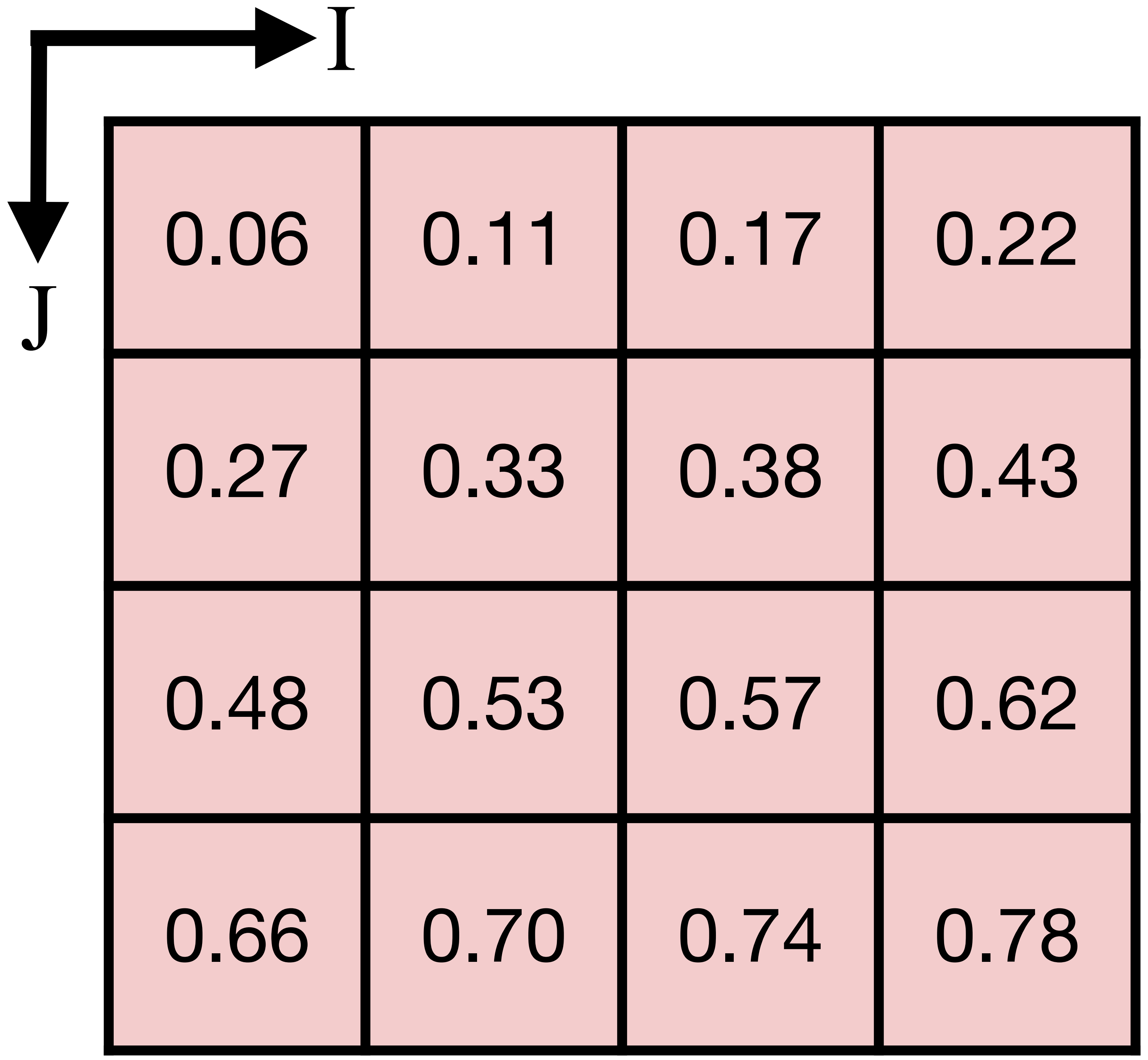


0.06	0.11	0.17	0.22
0.27	0.33	0.38	0.43
0.48	0.53	0.57	0.62
0.66	0.70	0.74	0.78

4x4 Brick storing a 4x4 block of user data

The Bricks Layout

Putting Bricks Together



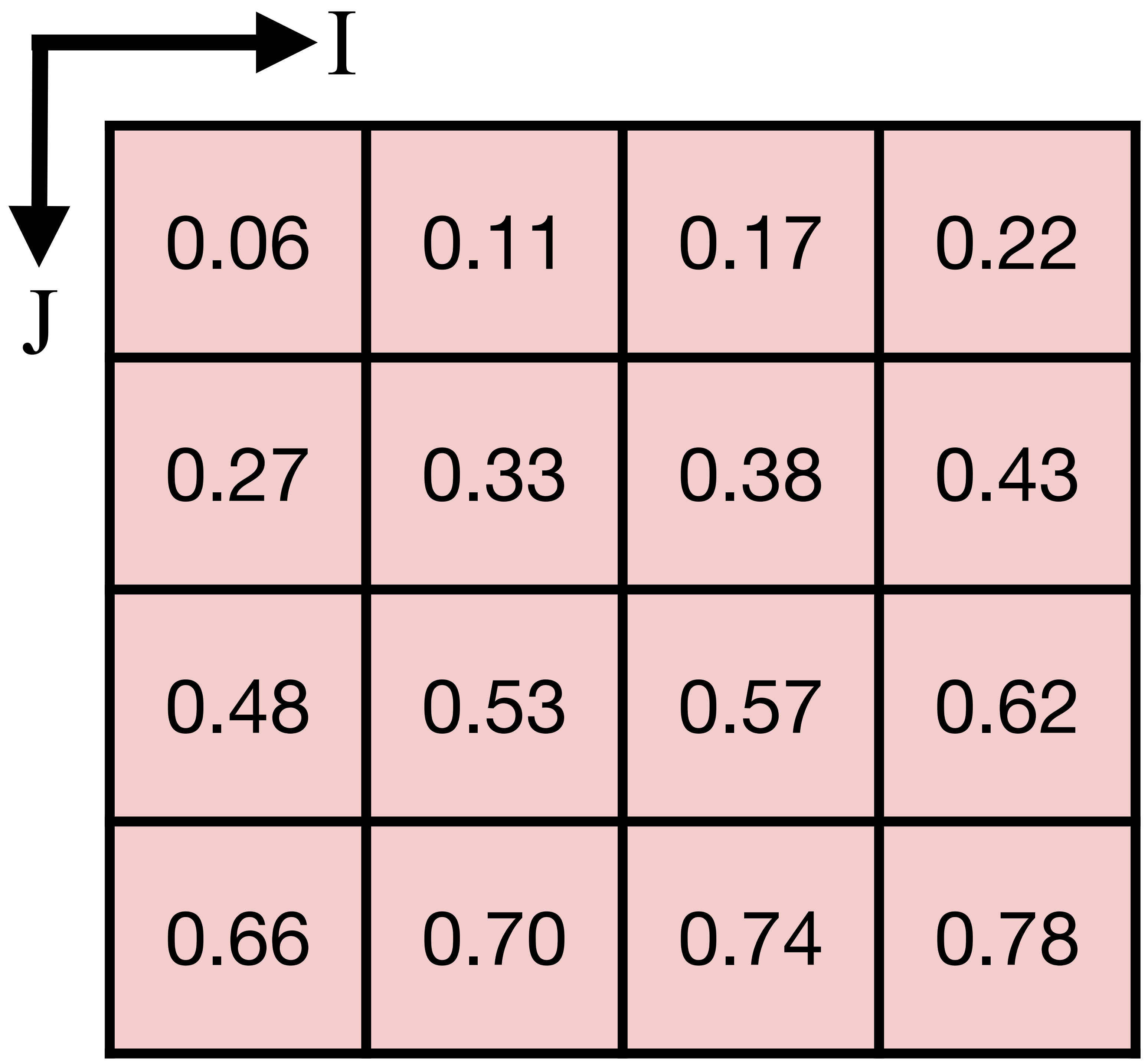
0.06	0.11	0.17	0.22
0.27	0.33	0.38	0.43
0.48	0.53	0.57	0.62
0.66	0.70	0.74	0.78

User data

The Bricks Layout

Putting Bricks Together

- Bricks are stored contiguously



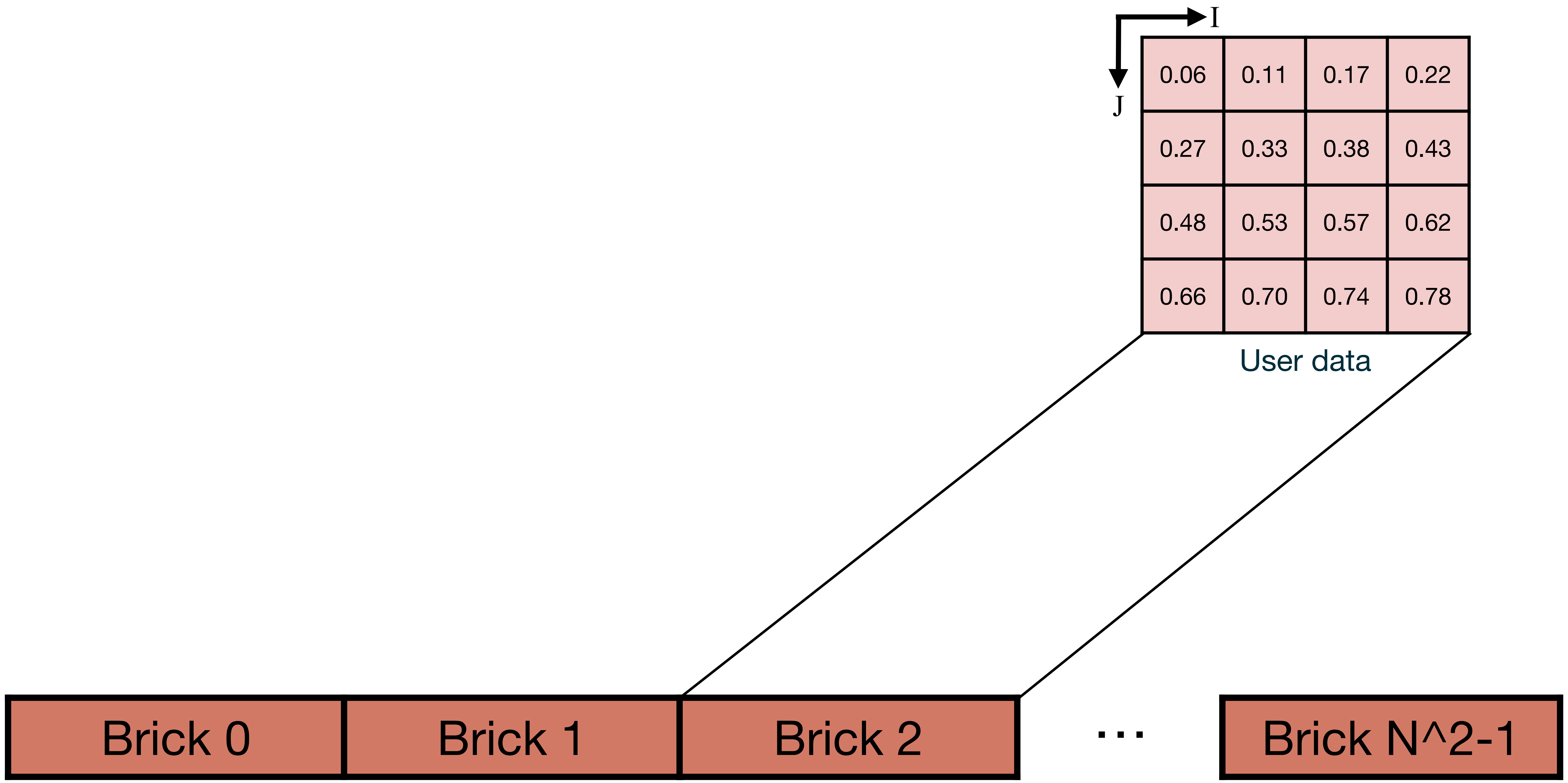
0.06	0.11	0.17	0.22
0.27	0.33	0.38	0.43
0.48	0.53	0.57	0.62
0.66	0.70	0.74	0.78

User data

The Bricks Layout

Putting Bricks Together

- Bricks are stored contiguously



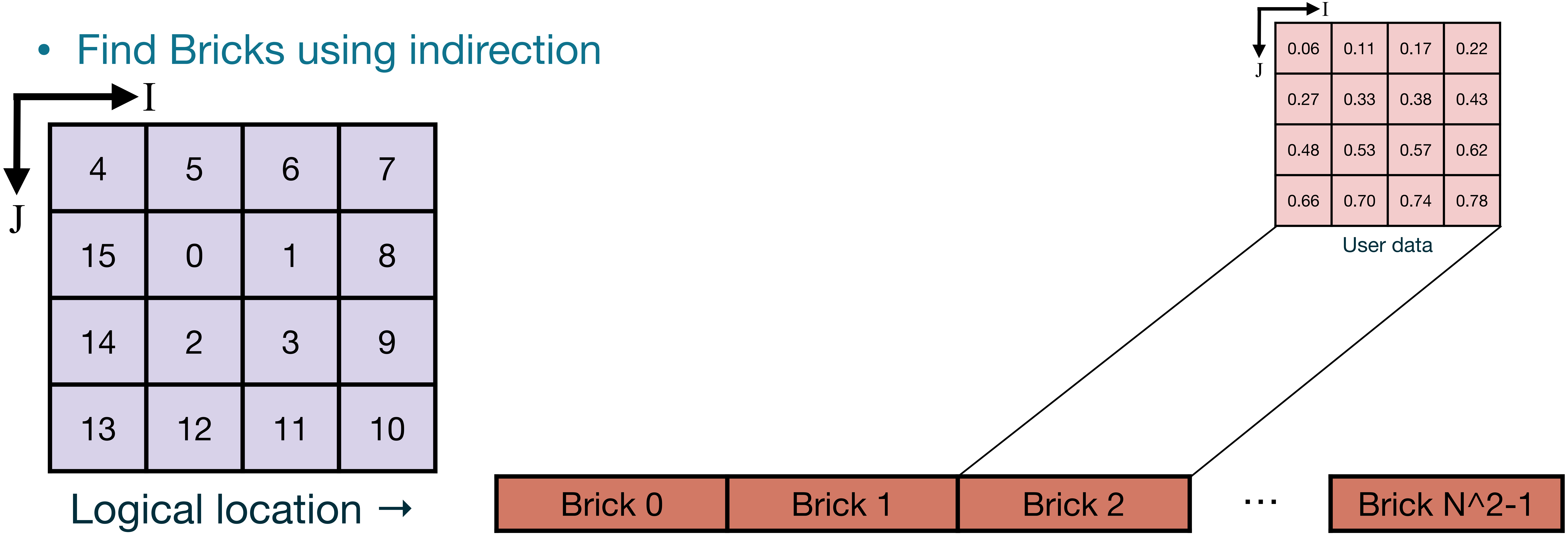
0.06	0.11	0.17	0.22
0.27	0.33	0.38	0.43
0.48	0.53	0.57	0.62
0.66	0.70	0.74	0.78

User data

The Bricks Layout

Putting Bricks Together

- Bricks are stored contiguously
- Find Bricks using indirection



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

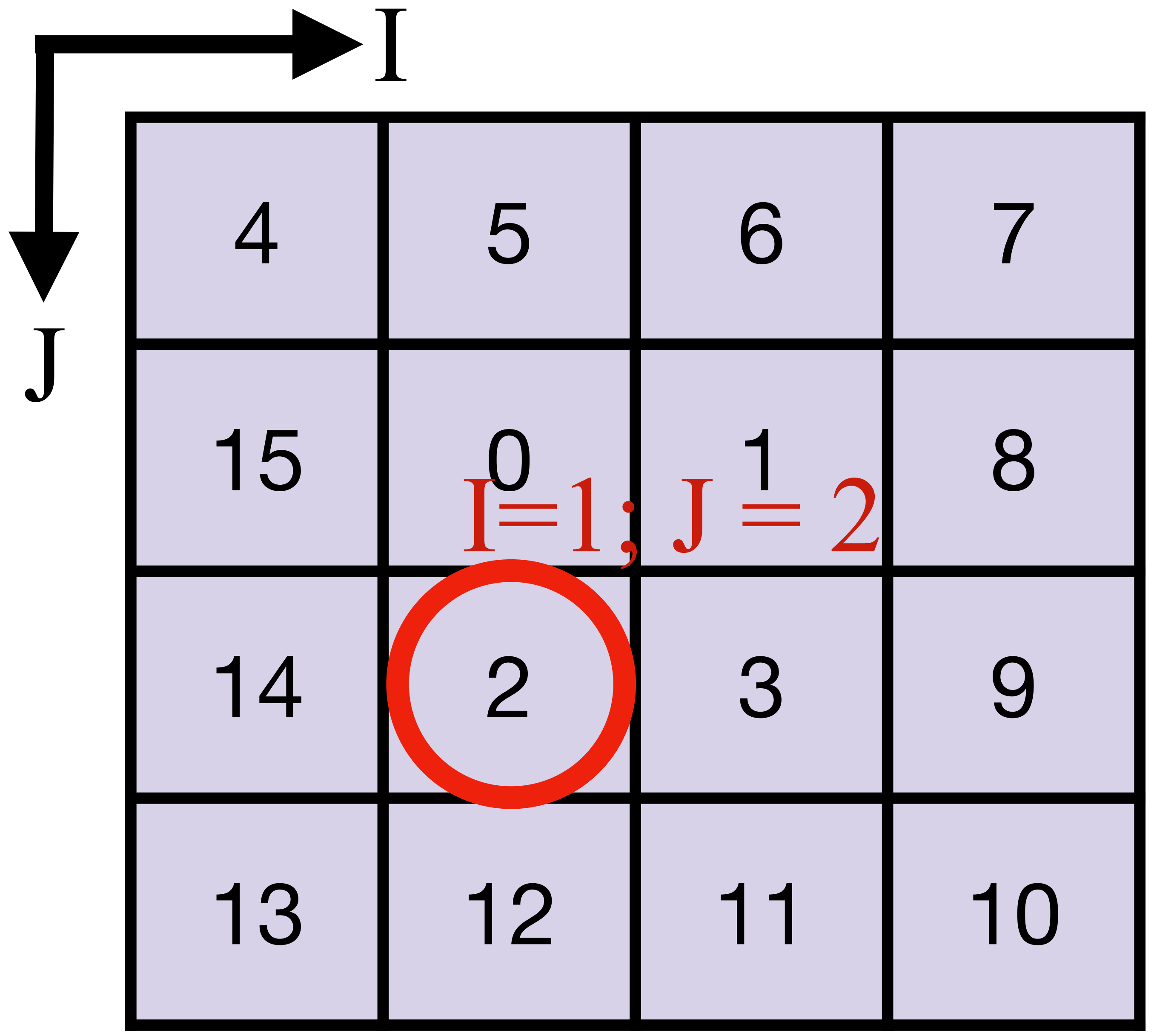
0.06	0.11	0.17	0.22
0.27	0.33	0.38	0.43
0.48	0.53	0.57	0.62
0.66	0.70	0.74	0.78

Logical location → physical location

The Bricks Layout

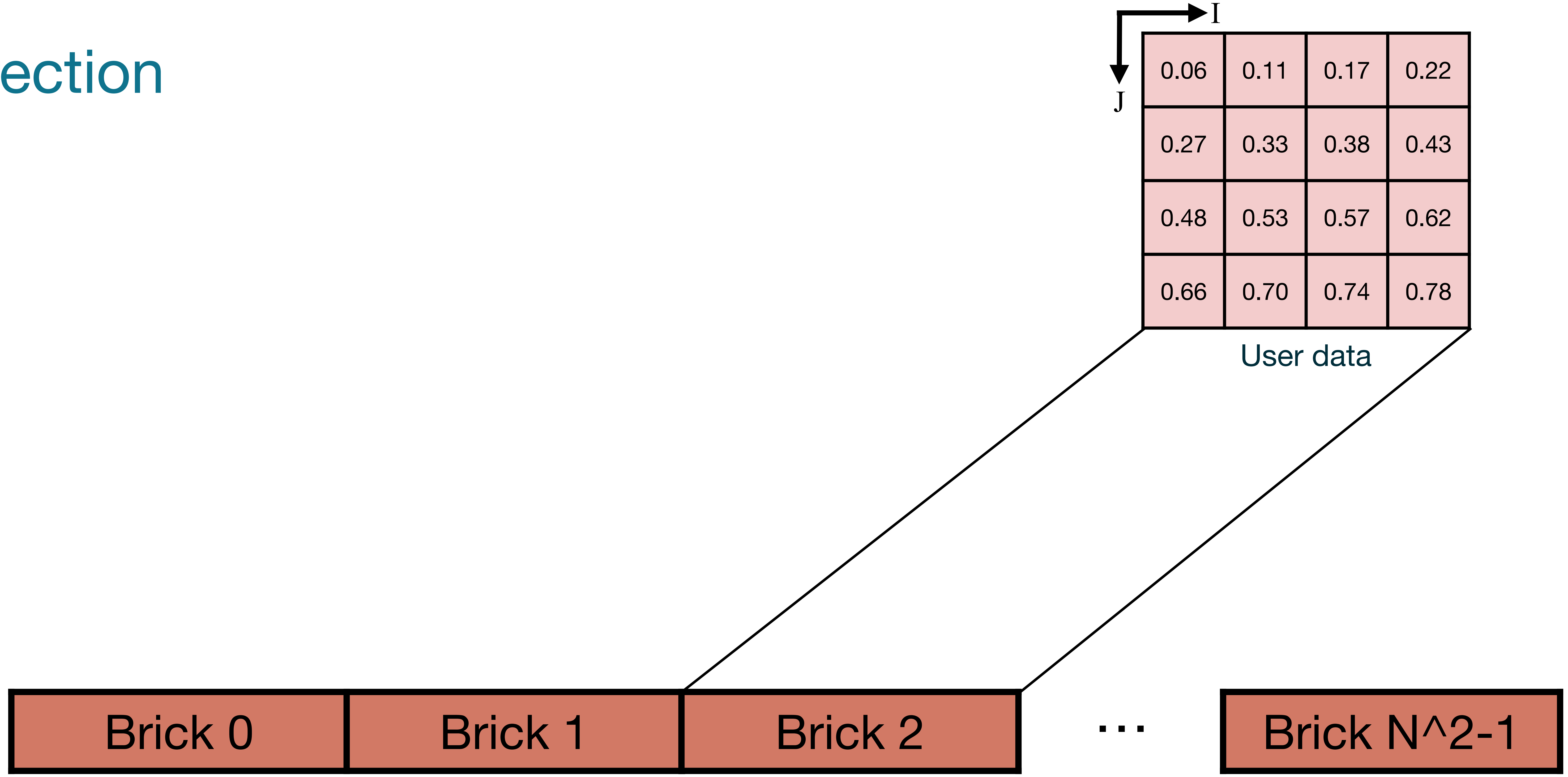
Putting Bricks Together

- Bricks are stored contiguously
- Find Bricks using indirection



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

Logical location → physical location

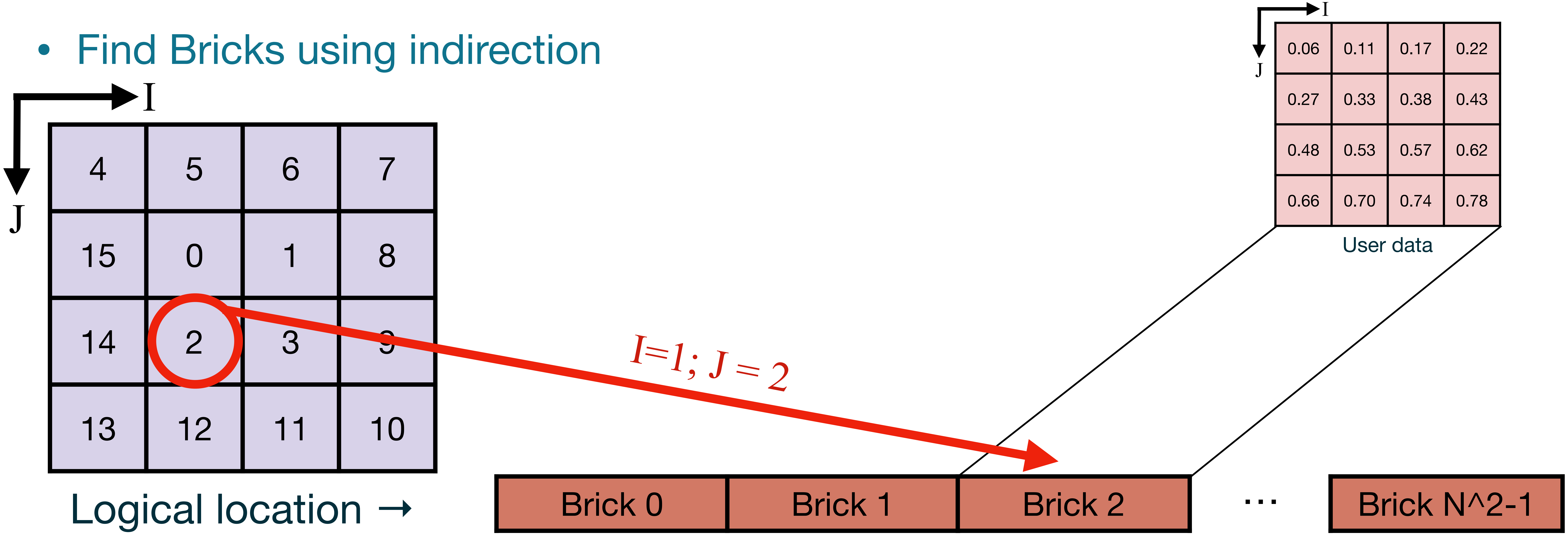


0.06	0.11	0.17	0.22
0.27	0.33	0.38	0.43
0.48	0.53	0.57	0.62
0.66	0.70	0.74	0.78

The Bricks Layout

Putting Bricks Together

- Bricks are stored contiguously
- Find Bricks using indirection



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

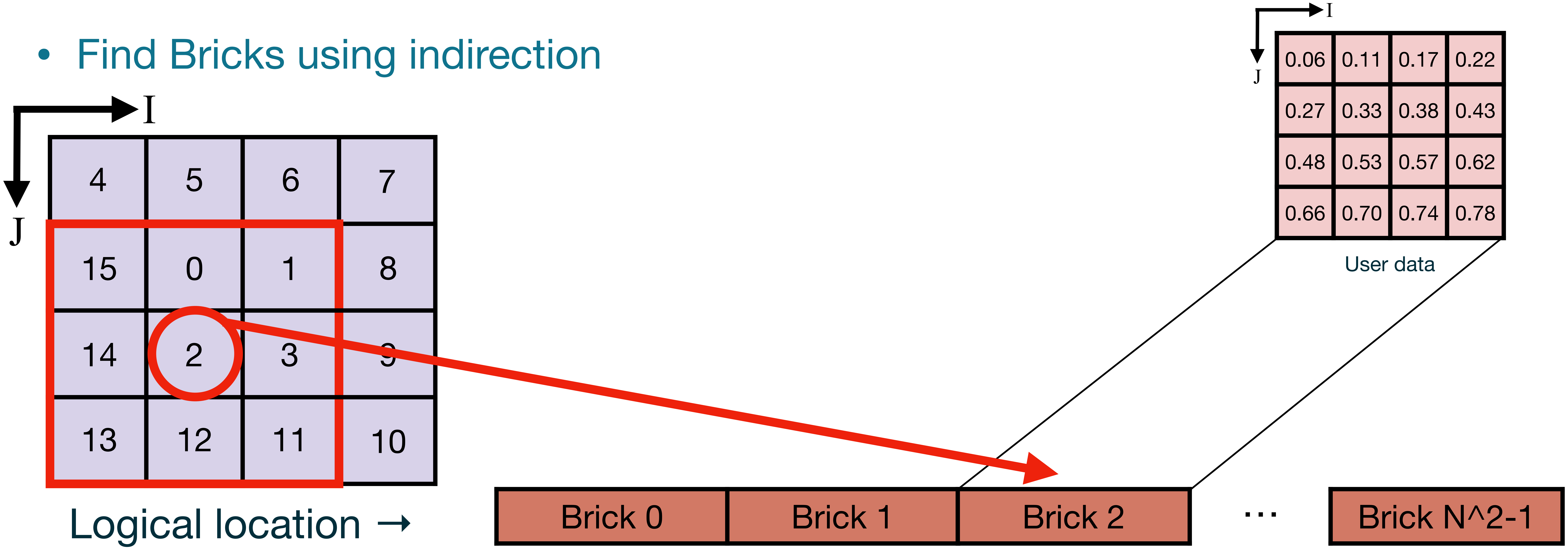
0.06	0.11	0.17	0.22
0.27	0.33	0.38	0.43
0.48	0.53	0.57	0.62
0.66	0.70	0.74	0.78

Logical location → physical location

The Bricks Layout

Putting Bricks Together

- Bricks are stored contiguously
- Find Bricks using indirection



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

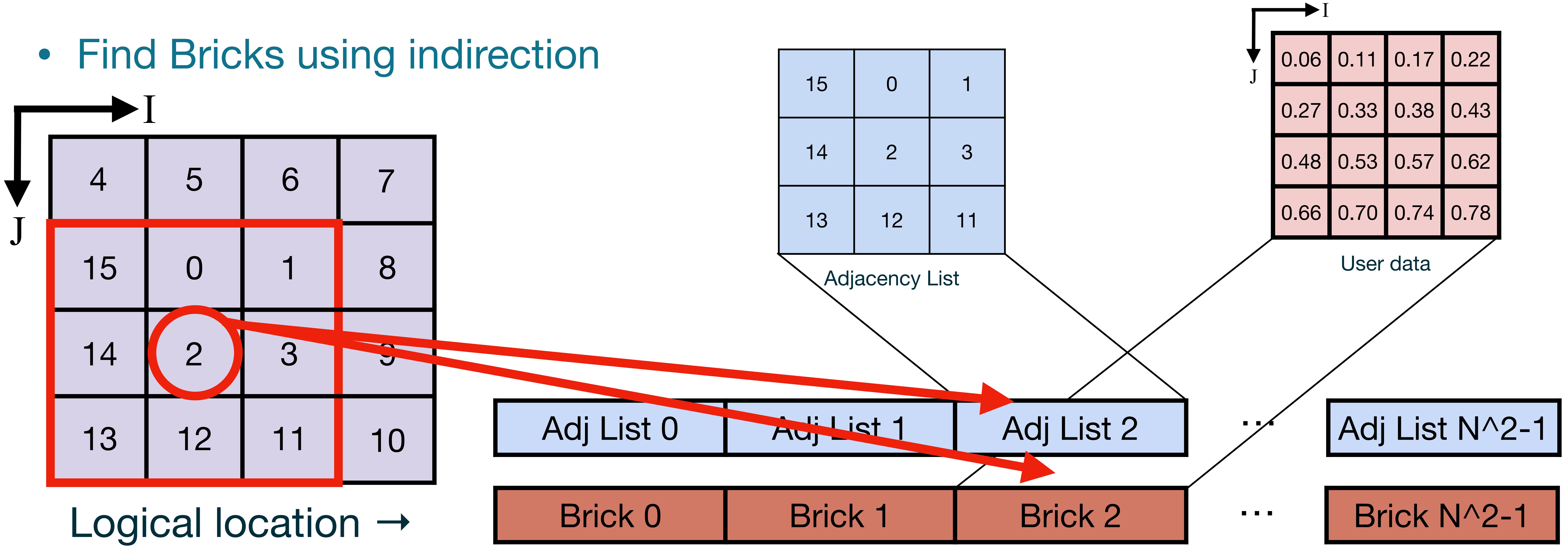
0.06	0.11	0.17	0.22
0.27	0.33	0.38	0.43
0.48	0.53	0.57	0.62
0.66	0.70	0.74	0.78

Logical location → physical location

The Bricks Layout

Putting Bricks Together

- Bricks are stored contiguously
- Find Bricks using indirection



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

15	0	1
14	2	3
13	12	11

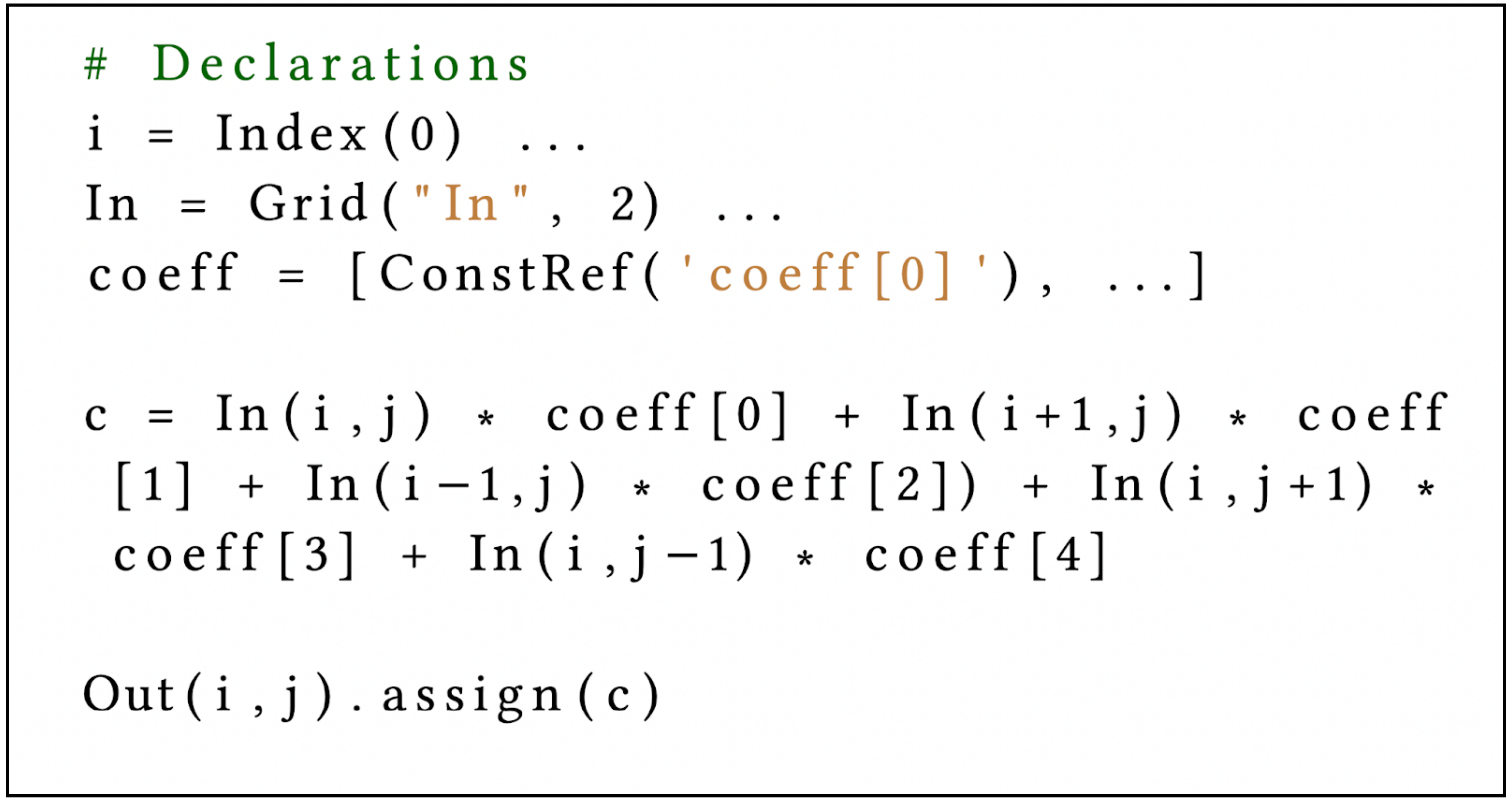
0.06	0.11	0.17	0.22
0.27	0.33	0.38	0.43
0.48	0.53	0.57	0.62
0.66	0.70	0.74	0.78

Logical location → physical location

The Bricks Library

Performance-portable stencil computations

- Code generation for stencil computations[1]
  - Targets CPUs and GPUs (NVIDIA, AMD, Intel, SVE)
  - User only needs to specify Brick shape and computation



```
# Declarations
i = Index(0) ...
In = Grid("In", 2) ...
coeff = [ConstRef('coeff[0]'), ...]

c = In(i,j) * coeff[0] + In(i+1,j) * coeff
 [1] + In(i−1,j) * coeff[2]) + In(i,j+1) *
 coeff[3] + In(i,j−1) * coeff[4]

Out(i,j).assign(c)
```

Example specification of 2D, 5-point stencil[2]

[1] Delivering Performance-Portable Stencil Computations on CPUs and GPUs using Bricks

[2] Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs

Bricks

Hardware-Aligned Layout

- N-D Grid → N-D Array of N-D Bricks

Bricks

Hardware-Aligned Layout

- N-D Grid → N-D Array of N-D Bricks
- Use the extra N dimensions and code generation to fit the computation to a specific hardware

Bricks

Hardware-Aligned Layout

- N-D Grid → N-D Array of N-D Bricks
- Use the extra N dimensions and code generation to fit the computation to a specific hardware

MPI → Partition the Bricks[3]

[3] Improving Communication by Optimizing On-Node Data Movement with Data Layout

Bricks

Hardware-Aligned Layout

- N-D Grid → N-D Array of N-D Bricks
- Use the extra N dimensions and code generation to fit the computation to a specific hardware

MPI → Partition the Bricks[3]

CPU thread/GPU thread block → Parallel loop over Bricks

[3] Improving Communication by Optimizing On-Node Data Movement with Data Layout

Bricks

Hardware-Aligned Layout

- N-D Grid → N-D Array of N-D Bricks
- Use the extra N dimensions and code generation to fit the computation to a specific hardware

MPI → Partition the Bricks[3]

CPU thread/GPU thread block → Parallel loop over Bricks

Vector Register/CUDA Warp → Generated code runs on Brick

[3] Improving Communication by Optimizing On-Node Data Movement with Data Layout

Bricks

Hardware-Aligned Layout

- N-D Grid → N-D Array of N-D Bricks
- Use the extra N dimensions and code generation to fit the computation to a specific hardware

MPI → Partition the Bricks[3]

CPU thread/GPU thread block → Parallel loop over Bricks

Vector Register/CUDA Warp → Generated code runs on Brick

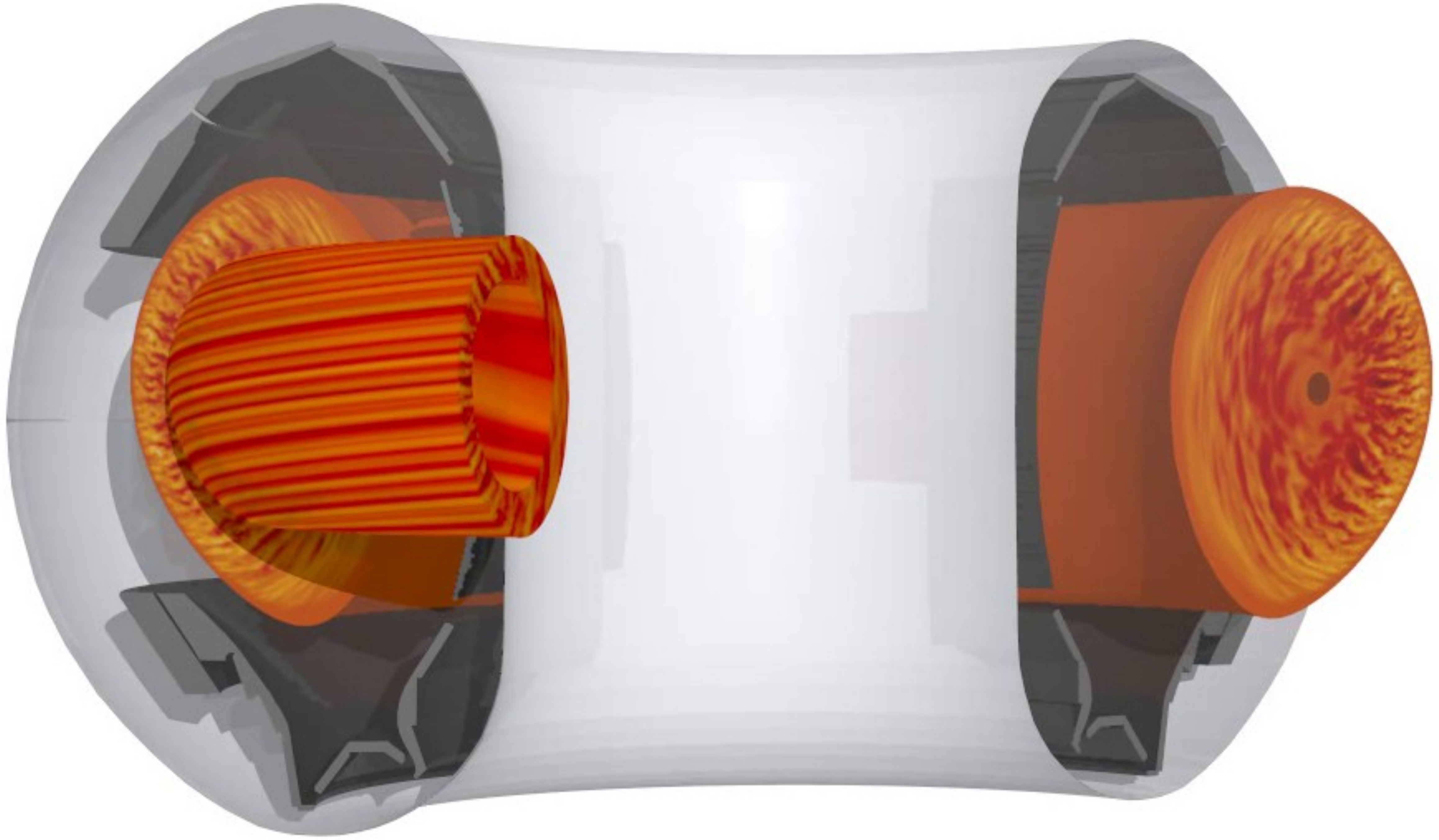
[3] Improving Communication by Optimizing On-Node Data Movement with Data Layout

High-Dimensional Bricks: Example

Example: Bricks in Phase-Space Kernels

GENE

- GENE[4]: 6D physical+phase-space fusion code
  - IJK: space
    - J: in Fourier space
  - LM: phase space
  - N: species
  - GTensor: GPU implementation
- Core computations:
  - Stencils
  - FFT (along J-axis)
  - Gyroaveraging, linear solves…



[4] http://genecode.org

Bricks in Phase-Space Kernels

Challenges

- Extending Bricks library support
  - Complex types
  - Managing metadata in 6D
  - MPI layout optimization in a **subset** of the dimensions
  - Code generation for computations on arrays of mixed dimensionality

- Non-stencil computations with Bricks
  - FFT
  - Linear solves
  - Gyroaveraging

Bricks in Phase-Space Kernels

Challenges

- Extending Bricks library support
    - ✓ Complex types
    - ✓ Managing metadata in 6D
    - ✓ MPI layout optimization in a **subset** of the dimensions
    - – *Partially supported:* code generation for computations on arrays of mixed dimensionality

- Non-stencil computations with Bricks
    - ✓ FFT (for CUDA)
    - – Linear solves
    - – Gyroaveraging

Example

Star-shaped 2D Stencil along K and L axes of a 6D array

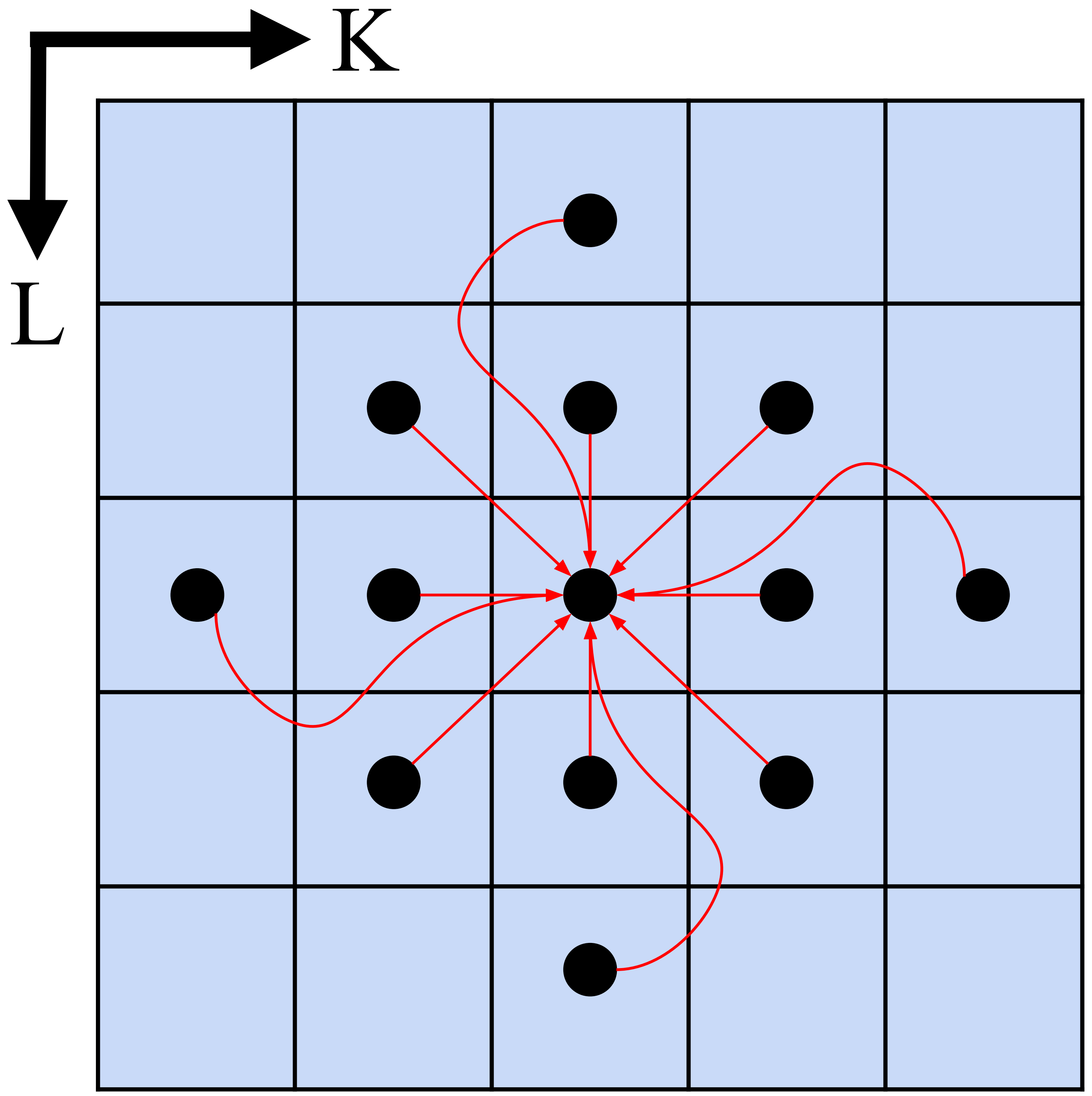
```
i, j, k, l, m, n = map(Index, range(6))

# Declare grid
input = Grid("bIn", 6, complex_valued=True)
output = Grid("bOut", 6, complex_valued=True)
coeffs = [ConstRef('coeff[0]'), ... ]

# Express computation
calc = coeffs[ 0] * input(i, j, k + 0, l - 2, m, n) + \
       coeffs[ 1] * input(i, j, k - 1, l - 1, m, n) + \
       coeffs[ 2] * input(i, j, k + 0, l - 1, m, n) + \
       ...
       coeffs[11] * input(i, j, k + 1, l + 1, m, n) + \
       coeffs[12] * input(i, j, k + 0, l + 2, m, n)

output(i, j, k, l, m, n).assign(calc)
```

Example specification of a 2D stencil along K and L axes of a 6D array



Data access pattern of a star-shaped 2D stencil

Example

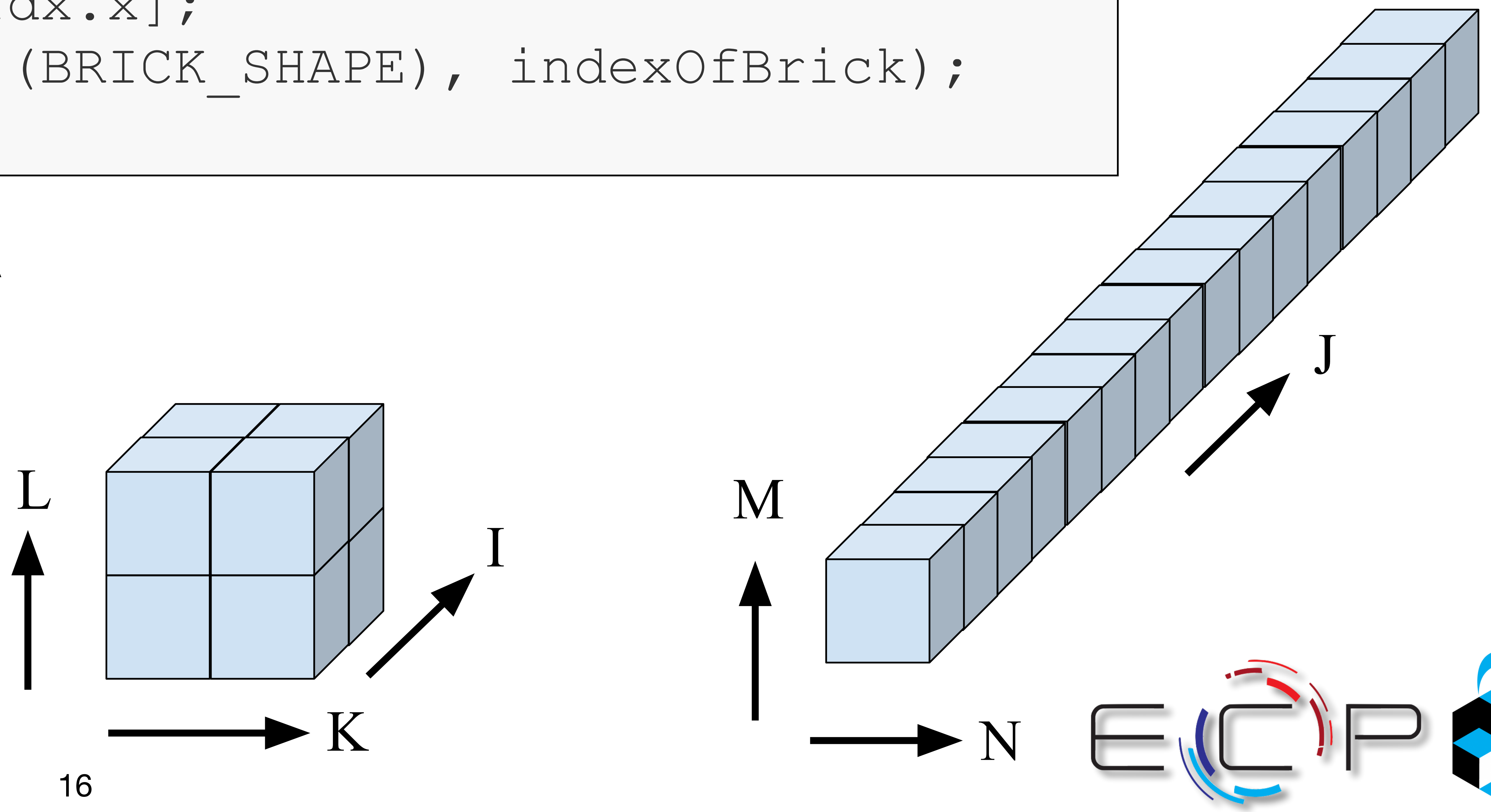
Using Generated Code

```
#define BRICK_SHAPE 1,1,2,2,16,2
//                  N,M,L,K, J,I

__global__ void
starShaped2DStencil(unsigned *brickIndices, // logical -> physical location
                    brick::Brick<BRICK_SHAPE> bIn, // input data
                    brick::Brick<BRICK_SHAPE> bOut, // output data
                    const double coeff[13]) // stencil coefficients
{
  unsigned indexOfBrick = brickIndices[blockIdx.x];
  brick("star_shaped_2d_stencil.py", "CUDA", (BRICK_SHAPE), indexOfBrick);
}
```

Including generated code in C++/CUDA

1 x 1 x 2 x 2 x 16 x 2 6D Brick
with good data reuse properties



Example

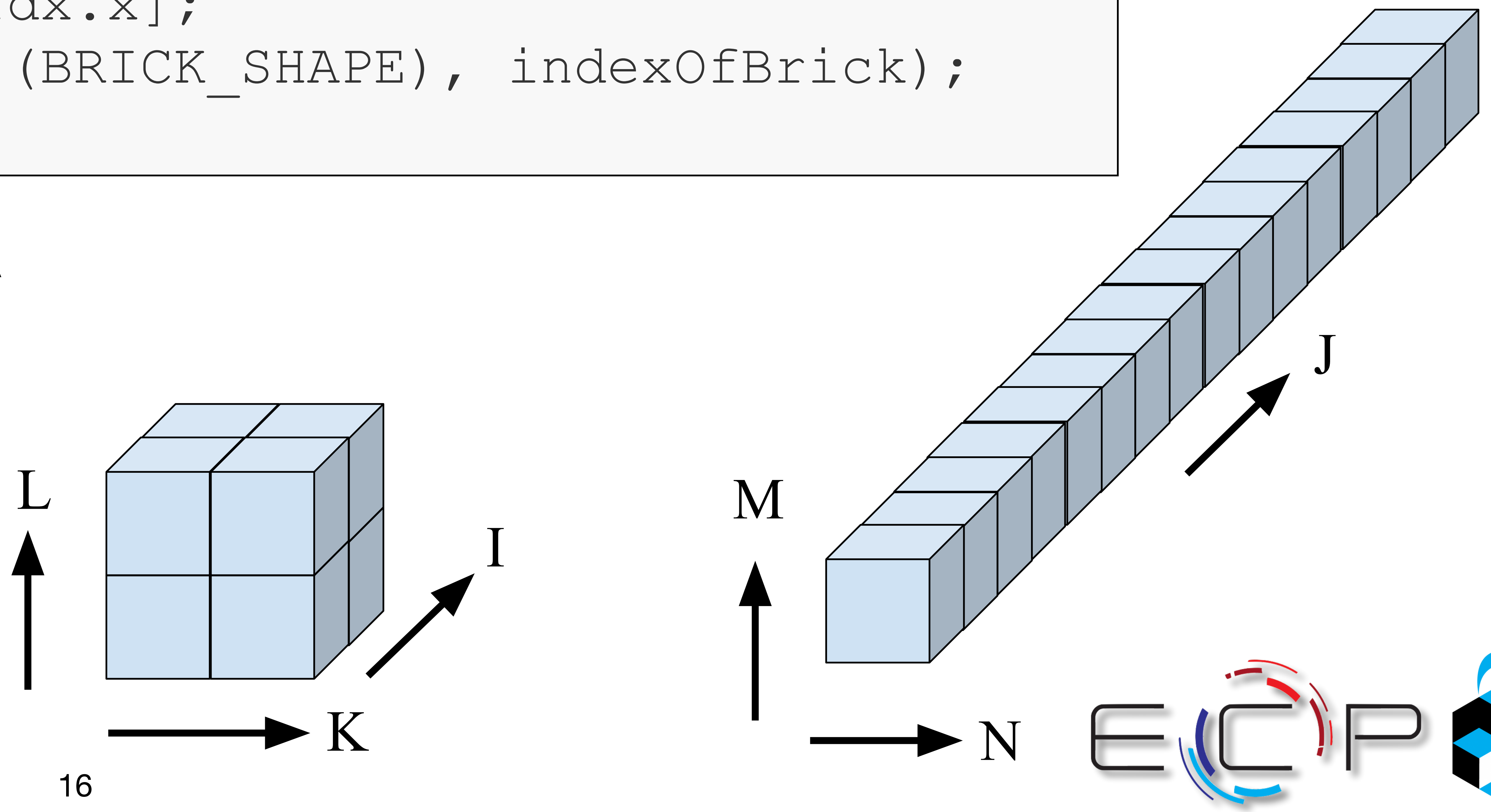
Using Generated Code

```
#define BRICK_SHAPE 1,1,2,2,16,2
//                  N,M,L,K, J,I

__global__ void
starShaped2DStencil(unsigned *brickIndices, // logical -> physical location
                    brick::Brick<BRICK_SHAPE> bIn, // input data
                    brick::Brick<BRICK_SHAPE> bOut, // output data
                    const double coeff[13]) // stencil coefficients
{
  unsigned indexOfBrick = brickIndices[blockIdx.x];
  brick("star_shaped_2d_stencil.py", "CUDA", (BRICK_SHAPE), indexOfBrick);
}
```

Including generated code in C++/CUDA

1 x 1 x 2 x 2 x 16 x 2 6D Brick
with good data reuse properties



Example

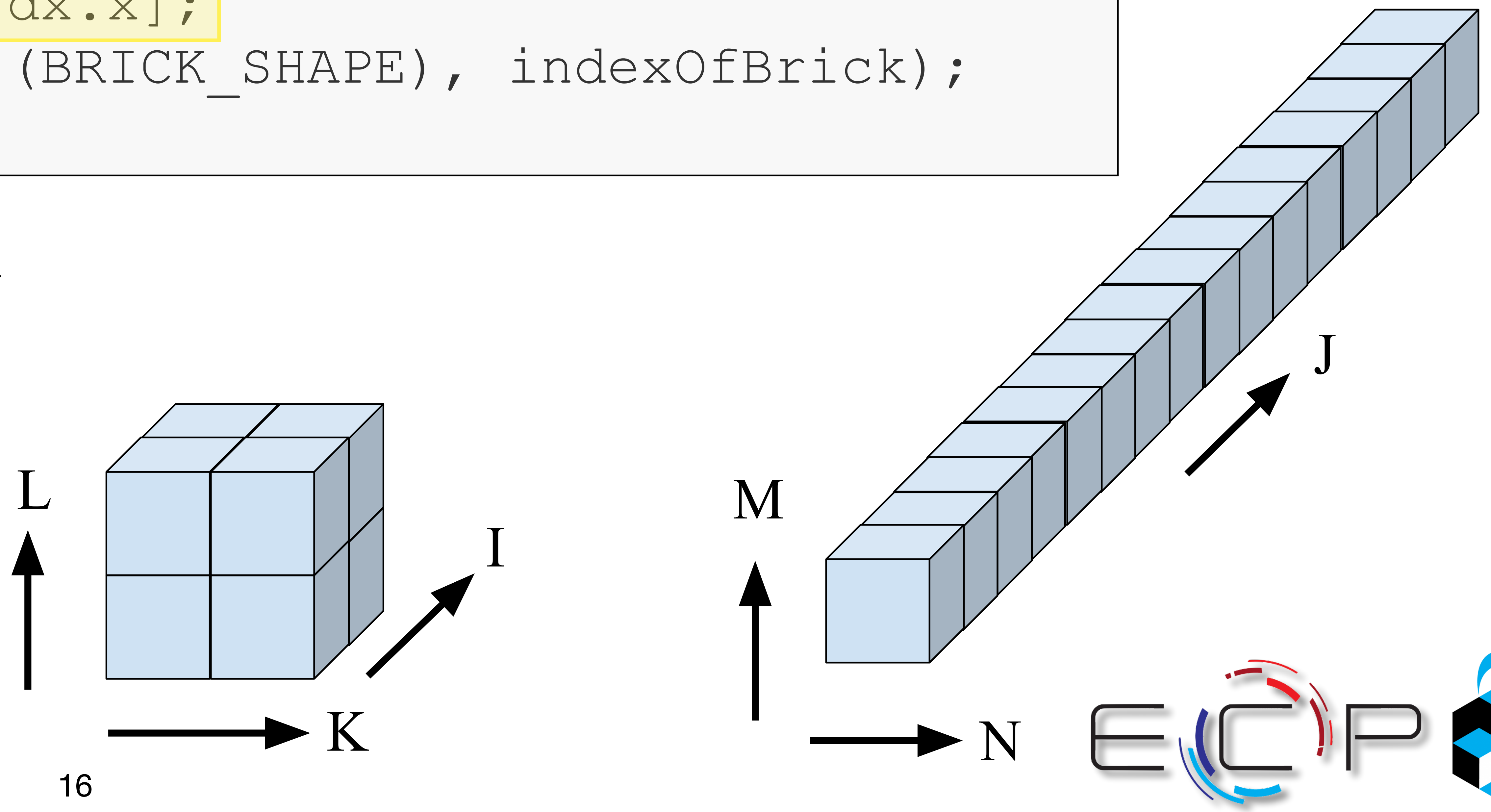
Using Generated Code

```
#define BRICK_SHAPE 1,1,2,2,16,2
//                  N,M,L,K, J,I

__global__ void
starShaped2DStencil(unsigned *brickIndices, // logical -> physical location
                    brick::Brick<BRICK_SHAPE> bIn, // input data
                    brick::Brick<BRICK_SHAPE> bOut, // output data
                    const double coeff[13]) // stencil coefficients
{
  unsigned indexOfBrick = brickIndices[blockIdx.x];
  brick("star_shaped_2d_stencil.py", "CUDA", (BRICK_SHAPE), indexOfBrick);
}
```

Including generated code in C++/CUDA

1 x 1 x 2 x 2 x 16 x 2 6D Brick
with good data reuse properties



Example

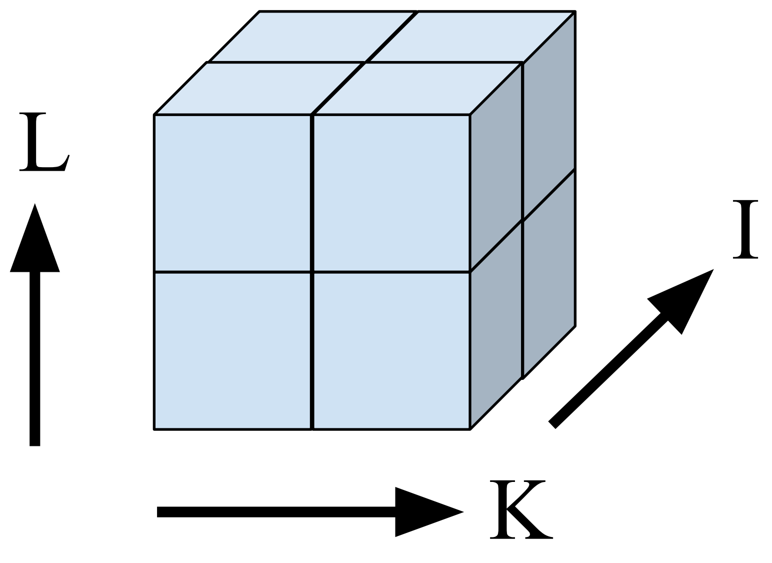
Using Generated Code

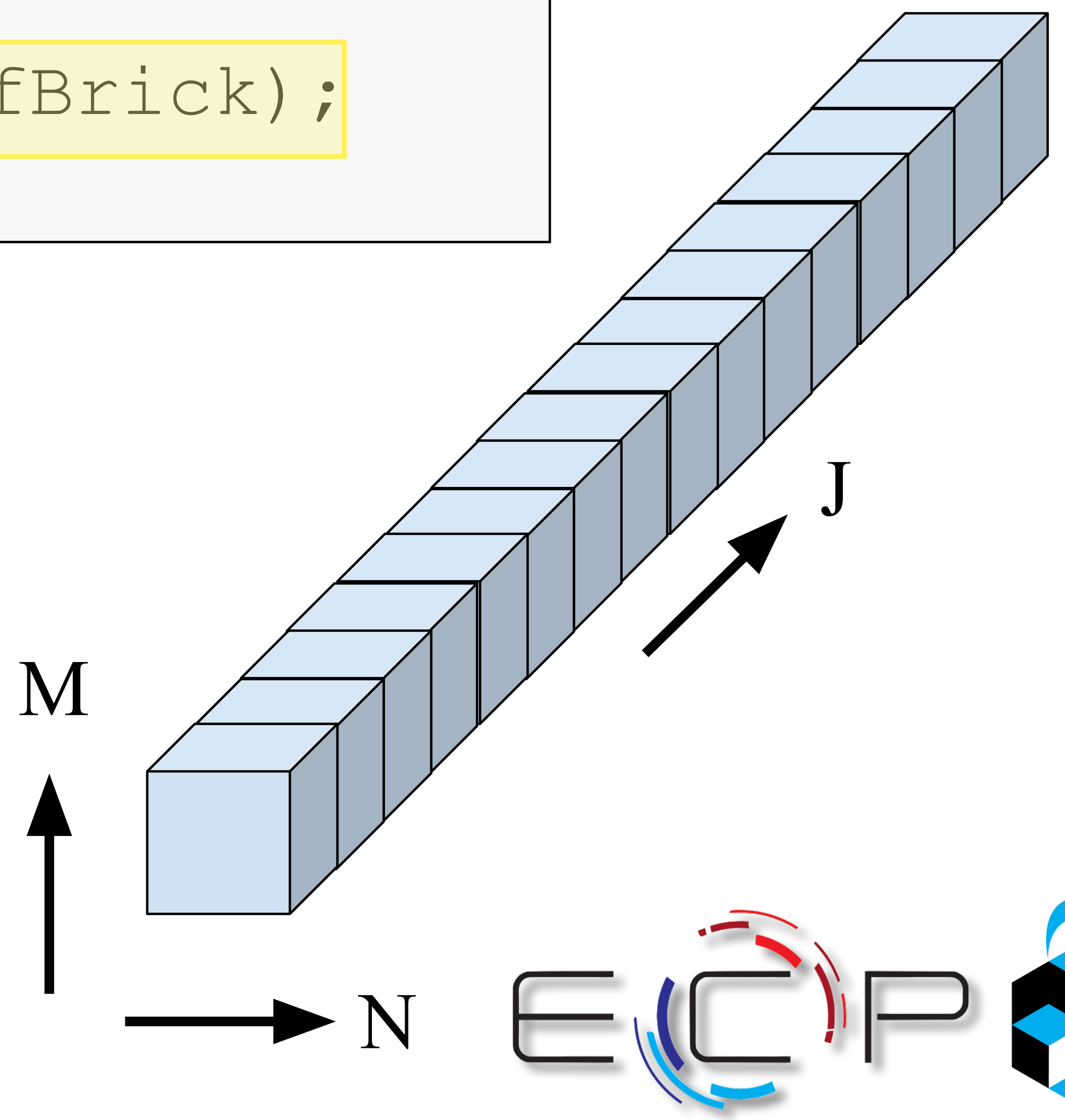
```
#define BRICK_SHAPE 1,1,2,2,16,2
//                  N,M,L,K, J,I

__global__ void
starShaped2DStencil(unsigned *brickIndices, // logical -> physical location
                    brick::Brick<BRICK_SHAPE> bIn, // input data
                    brick::Brick<BRICK_SHAPE> bOut, // output data
                    const double coeff[13]) // stencil coefficients
{
  unsigned indexOfBrick = brickIndices[blockIdx.x];
  brick("star_shaped_2d_stencil.py", "CUDA", (BRICK_SHAPE), indexOfBrick);
}
```

Including generated code in C++/CUDA

1 x 1 x 2 x 2 x 16 x 2 6D Brick
with good data reuse properties



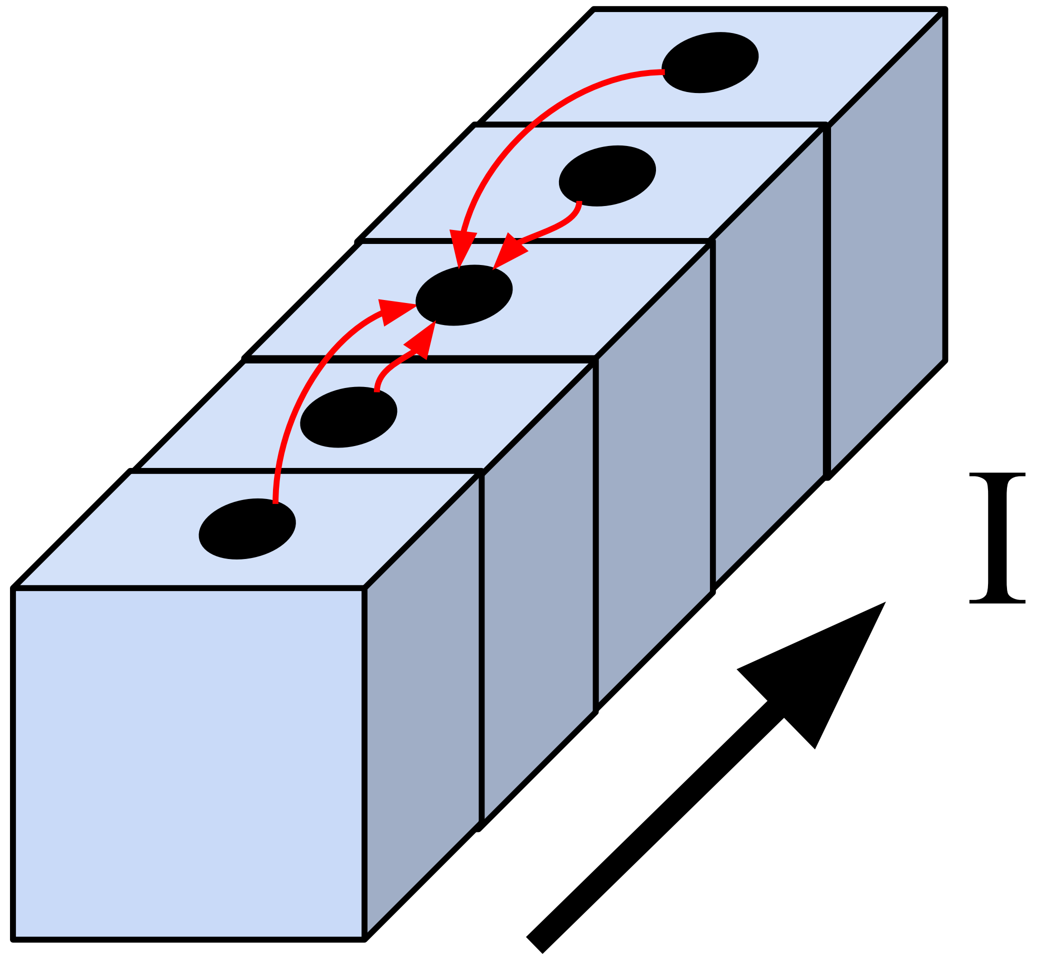


GENE Microbenchmarks

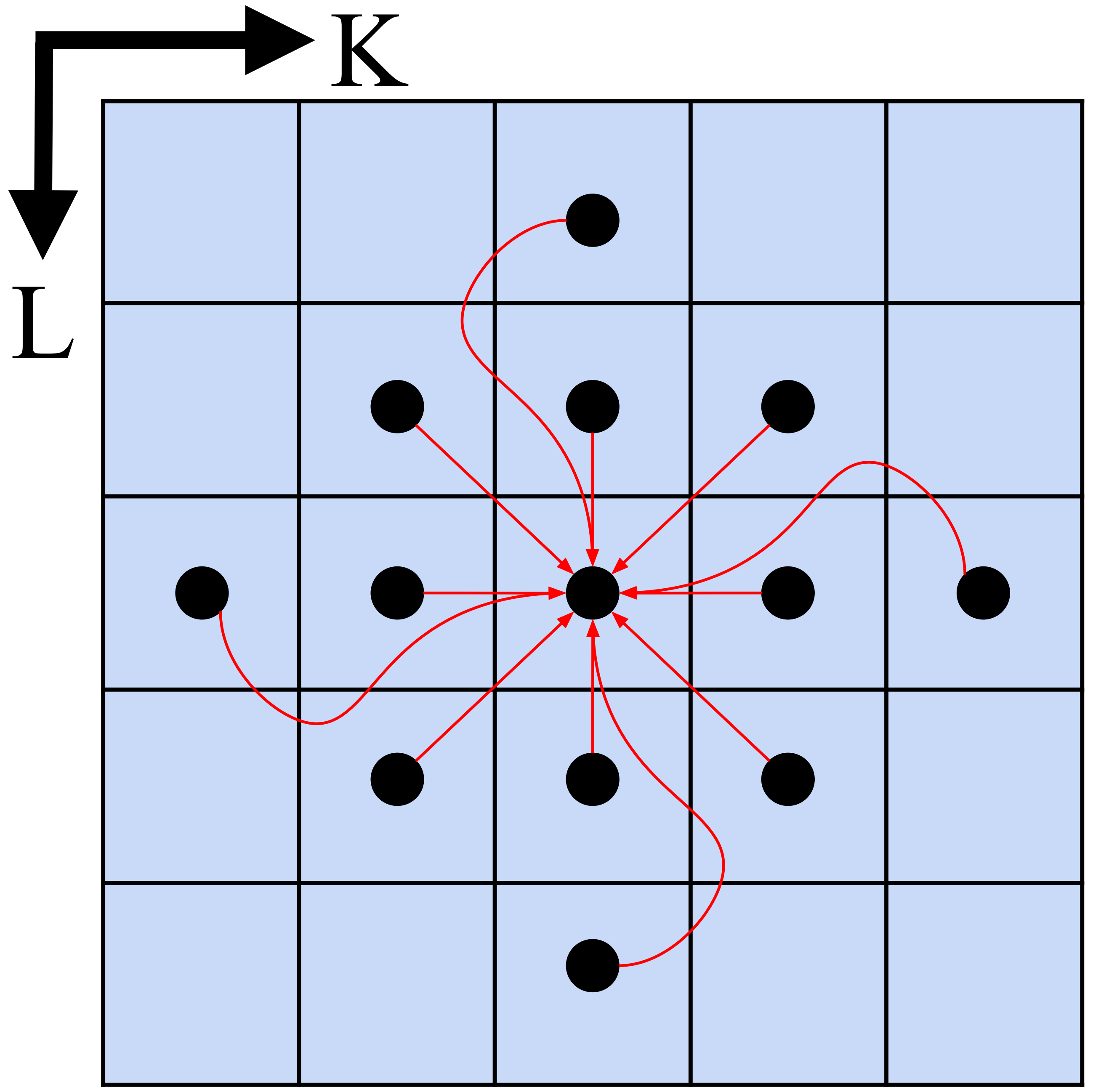
Stencils

GENE Microbenchmarks[4]

- 1D / 2D stencils along 6D arrays
- (1D) 5-point stencil
  - Fused with operations on 5D coefficients (no J-dependence)
- (2D) Arakawa *K-L* stencil
  - 5D coefficients: no J-dependence



Data access pattern of 1D stencil



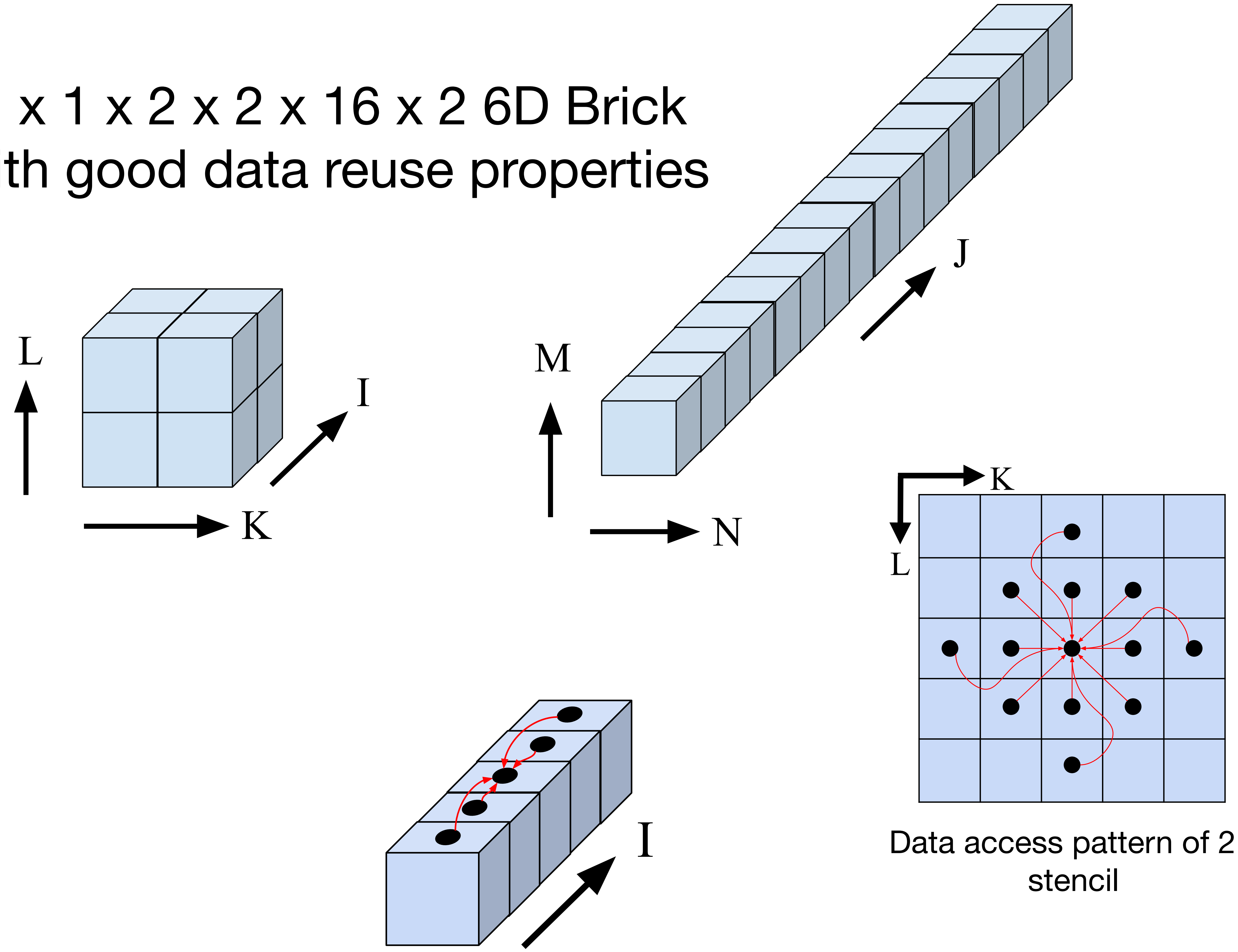
Data access pattern of 2D stencil

[4][6] Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU

Stencils

GENE Microbenchmarks[4]

- 1D / 2D stencils along 6D arrays
- (1D) 5-point stencil
  - Fused with operations on 5D coefficients (no J-dependence)
- (2D) Arakawa *K-L* stencil
  - 5D coefficients: no J-dependence



1 x 1 x 2 x 2 x 16 x 2 6D Brick with good data reuse properties

Data access pattern of 2D stencil

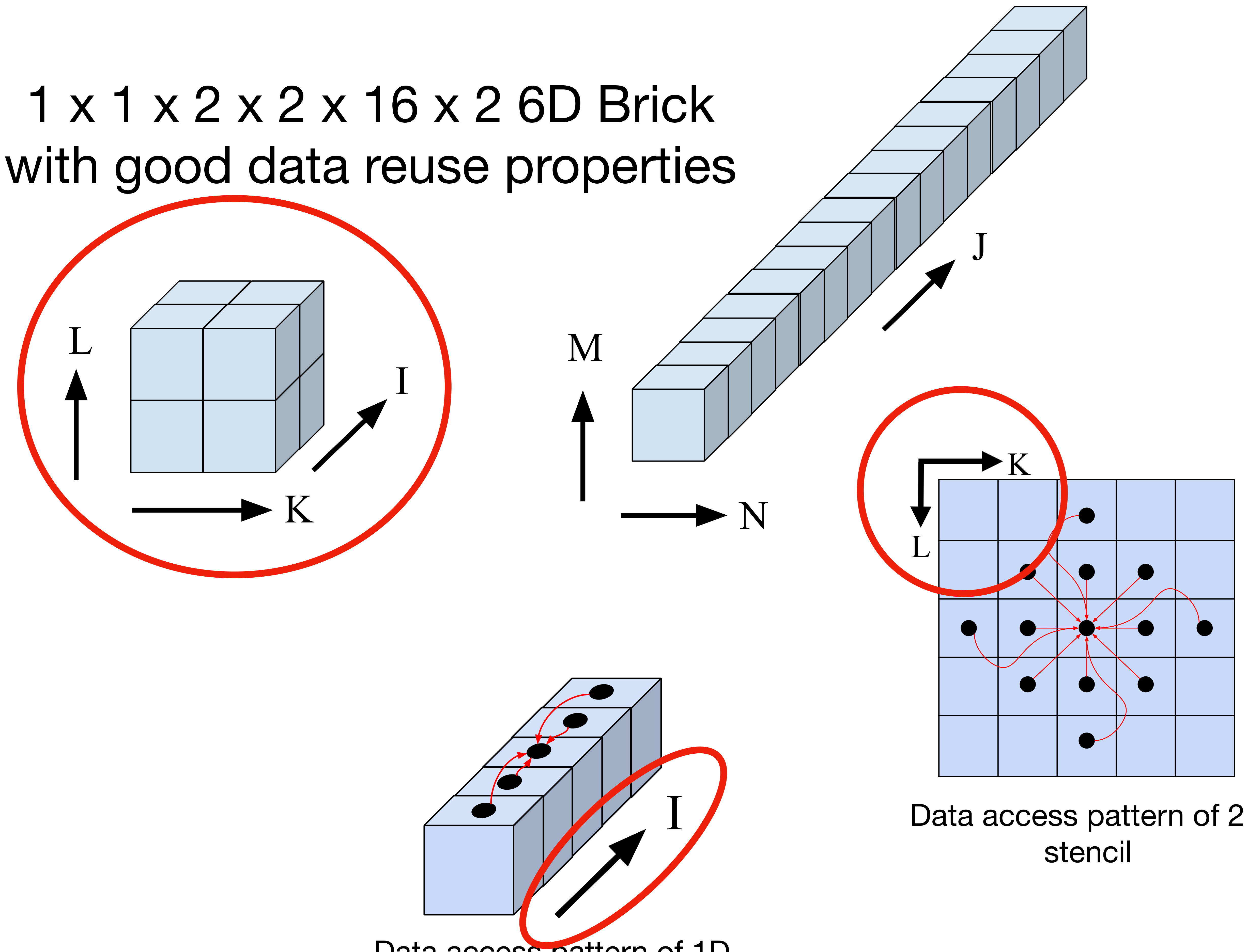
Data access pattern of 1D stencil

[4] [Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU]

Stencils

GENE Microbenchmarks[4]

- 1D / 2D stencils along 6D arrays
- (1D) 5-point stencil
  - Fused with operations on 5D coefficients (no J-dependence)
- (2D) Arakawa *K-L* stencil
  - 5D coefficients: no J-dependence



1 x 1 x 2 x 2 x 16 x 2 6D Brick with good data reuse properties

Data access pattern of 2D stencil

Data access pattern of 1D stencil

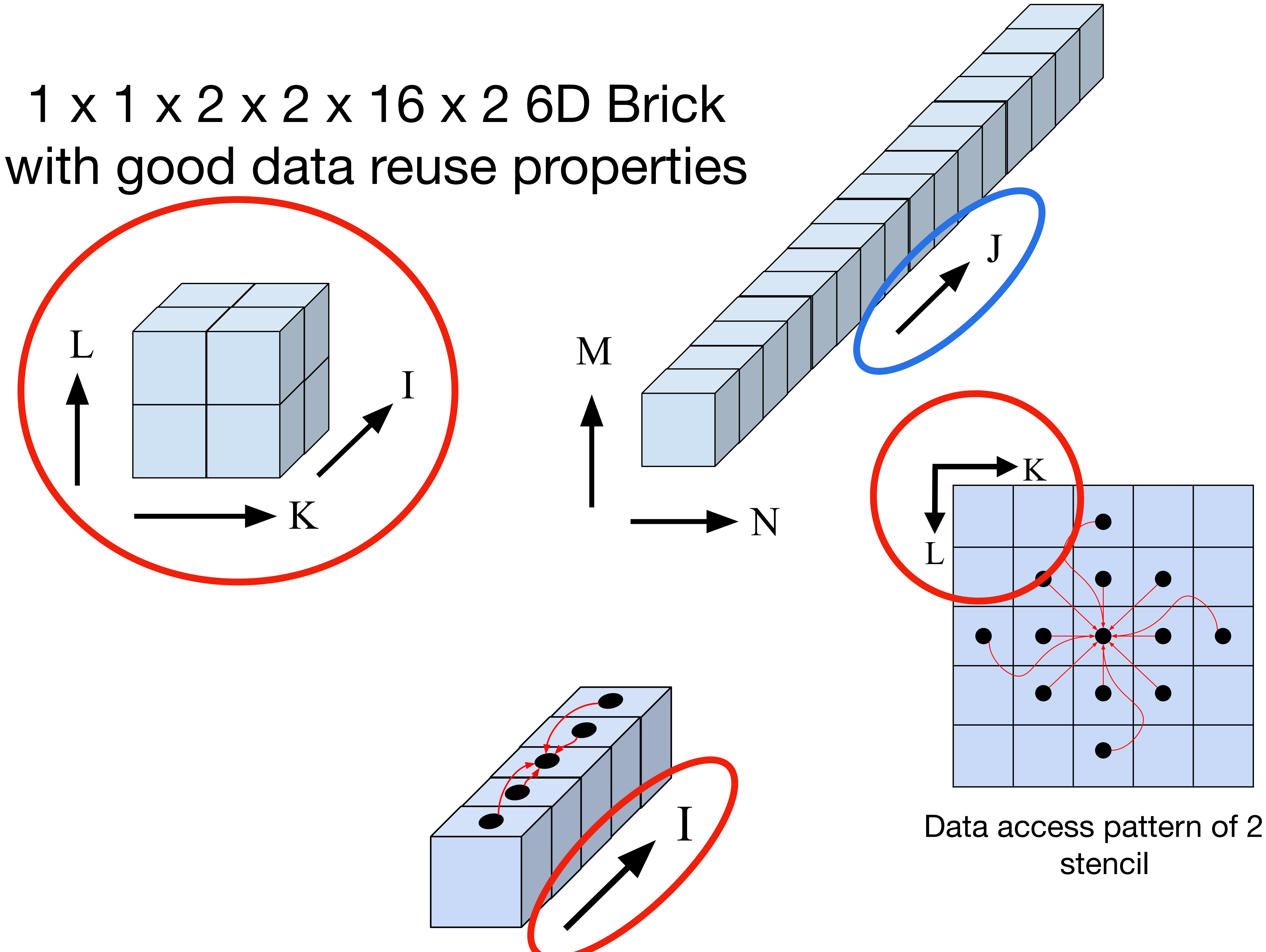
[4]6[Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU]

Stencils

GENE Microbenchmarks[4]

- 1D / 2D stencils along 6D arrays
- (1D) 5-point stencil
  - Fused with operations on 5D coefficients (no J-dependence)
- (2D) Arakawa *K-L* stencil
  - 5D coefficients: no J-dependence

1 x 1 x 2 x 2 x 16 x 2 6D Brick with good data reuse properties



Data access pattern of 2D stencil

Data access pattern of 1D stencil

[4]6Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU

Metadata Reduction

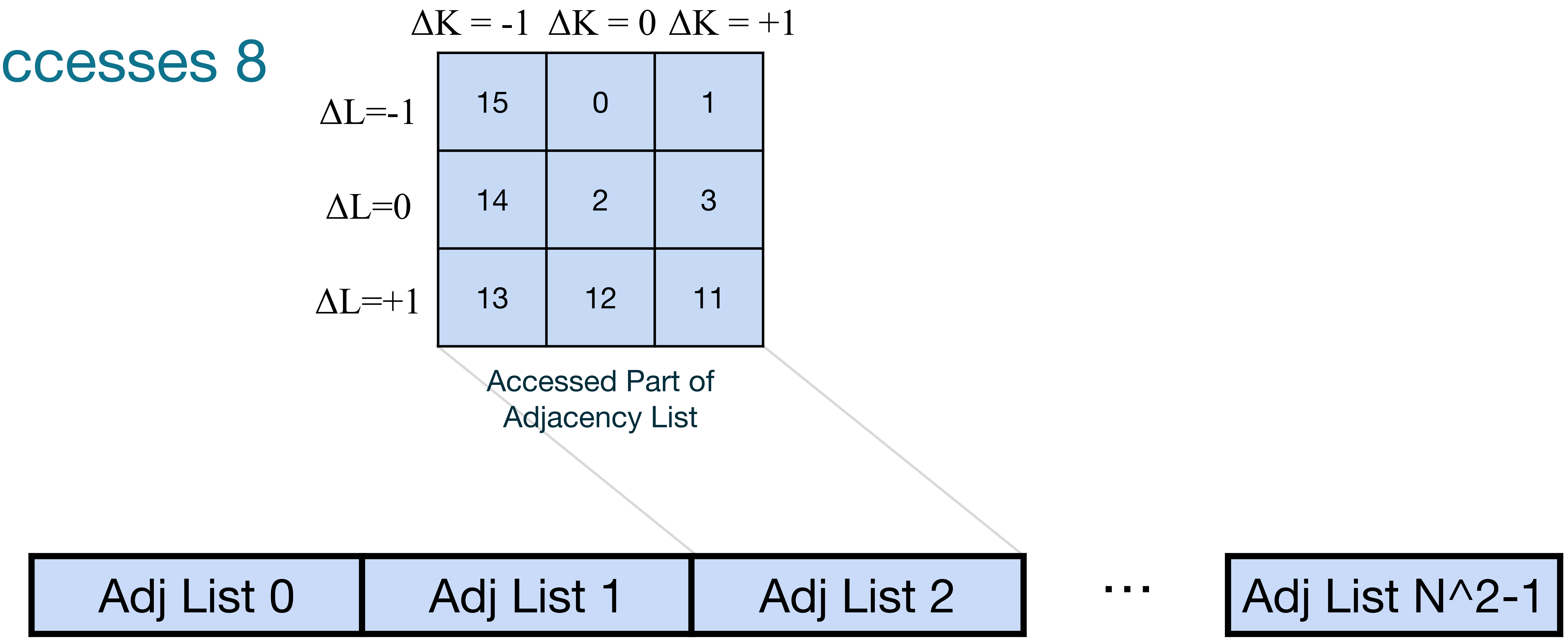
Managing Adjacency Lists

- Full adjacency list includes 3^D elements (Δ = -1, 0, or 1 on each axis)

Metadata Reduction

Managing Adjacency Lists

- Full adjacency list includes 3^D elements (Δ = -1, 0, or 1 on each axis)
- A 2D stencil in 6D only accesses 8 neighbors (32B)



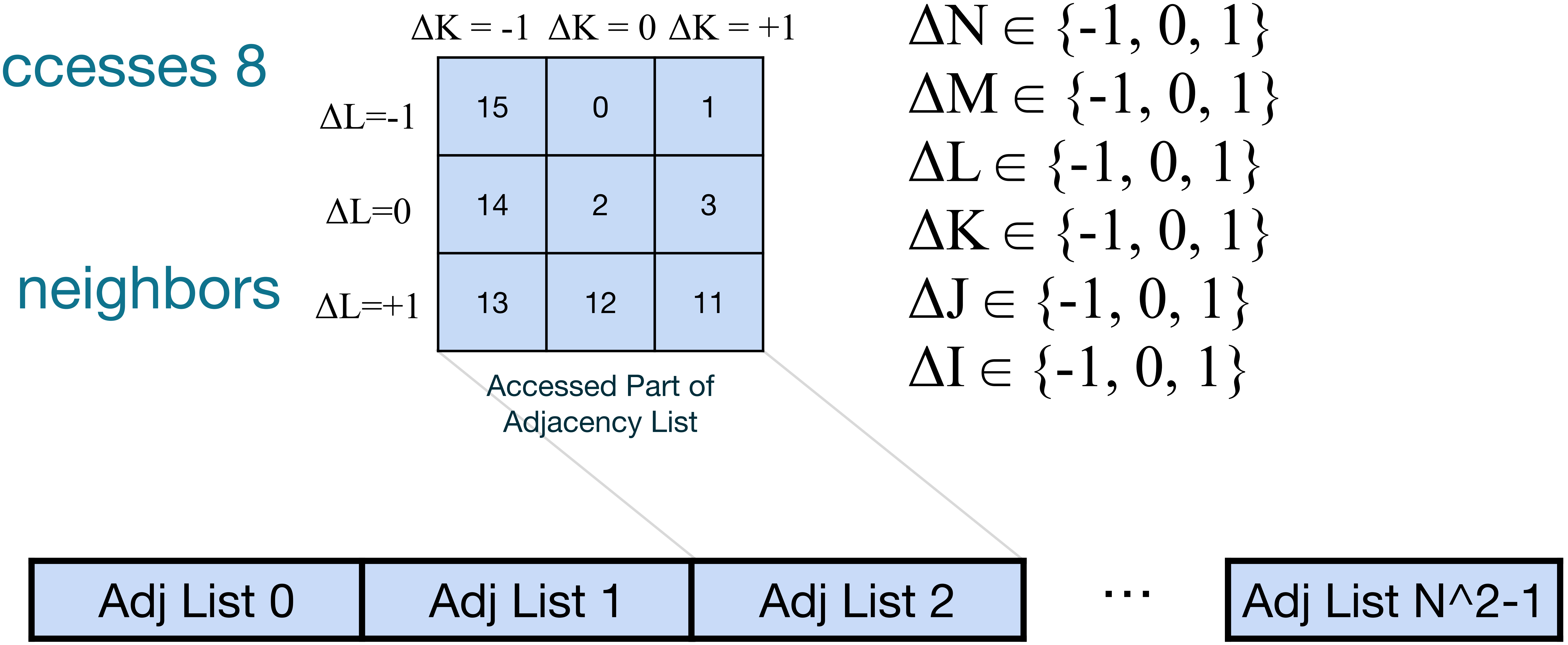
	ΔK = -1	ΔK = 0	ΔK = +1
ΔL=-1	15	0	1
ΔL=0	14	2	3
ΔL=+1	13	12	11

Accessed Part of Adjacency List

Metadata Reduction

Managing Adjacency Lists

- Full adjacency list includes 3^D elements (Δ = -1, 0, or 1 on each axis)
- A 2D stencil in 6D only accesses 8 neighbors (32B)
- Adjacency list holds 729 neighbors (2.9KB)



	$\Delta K = -1$	$\Delta K = 0$	$\Delta K = +1$
$\Delta L = -1$	15	0	1
$\Delta L = 0$	14	2	3
$\Delta L = +1$	13	12	11

Accessed Part of Adjacency List

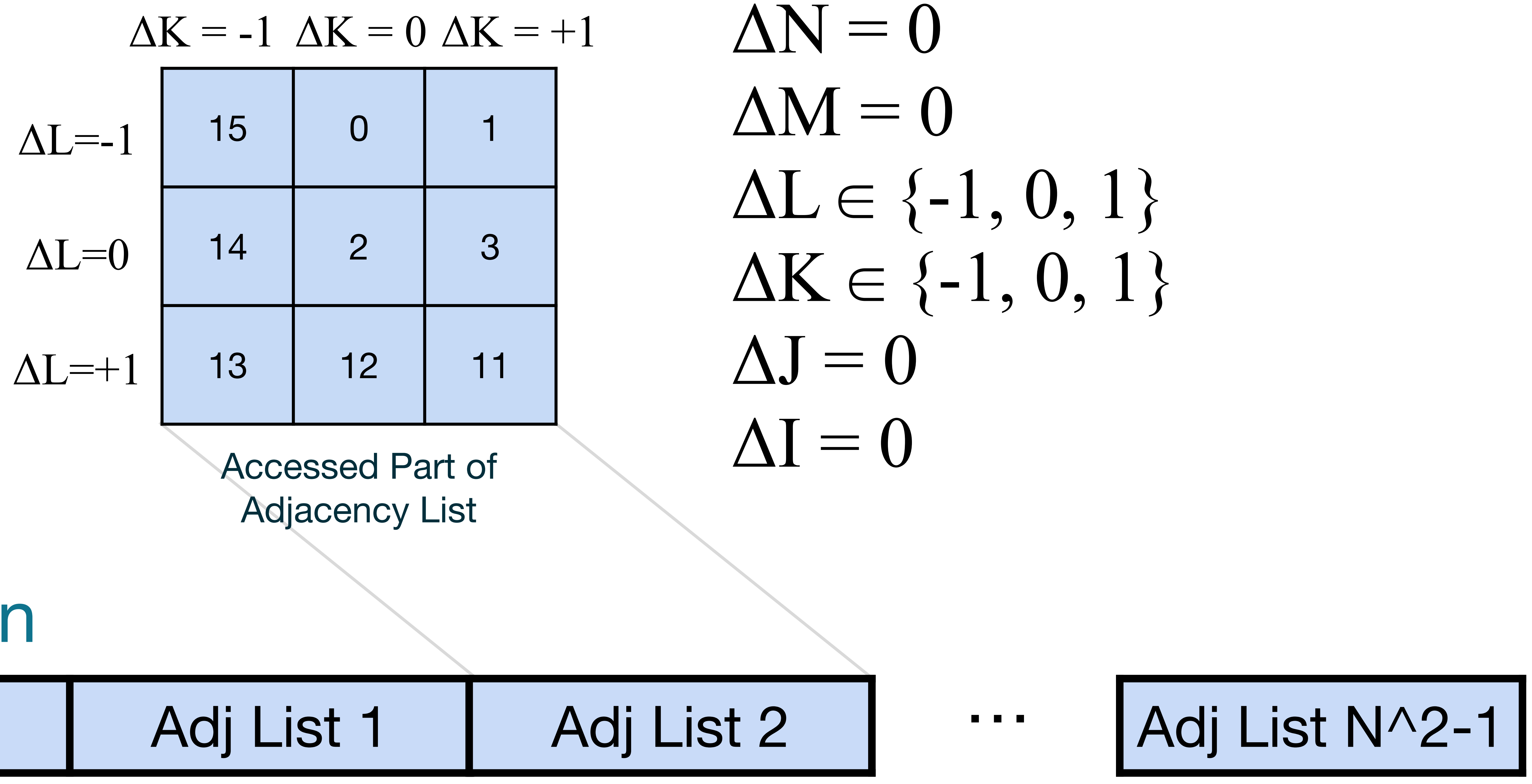
Adj List 0 | Adj List 1 | Adj List 2 | … | Adj List N^2-1

$$\Delta N \in \{-1, 0, 1\}$$
$$\Delta M \in \{-1, 0, 1\}$$
$$\Delta L \in \{-1, 0, 1\}$$
$$\Delta K \in \{-1, 0, 1\}$$
$$\Delta J \in \{-1, 0, 1\}$$
$$\Delta I \in \{-1, 0, 1\}$$

Metadata Reduction

Managing Adjacency Lists

- Full adjacency list includes 3^D elements (Δ = -1, 0, or 1 on each axis)
- A 2D stencil in 6D only accesses 8 neighbors (32B)
- Adjacency list holds 729 neighbors (2.9KB)
- Now users can exclude axes from an adjacency list



	ΔK = -1	ΔK = 0	ΔK = +1
ΔL=-1	15	0	1
ΔL=0	14	2	3
ΔL=+1	13	12	11

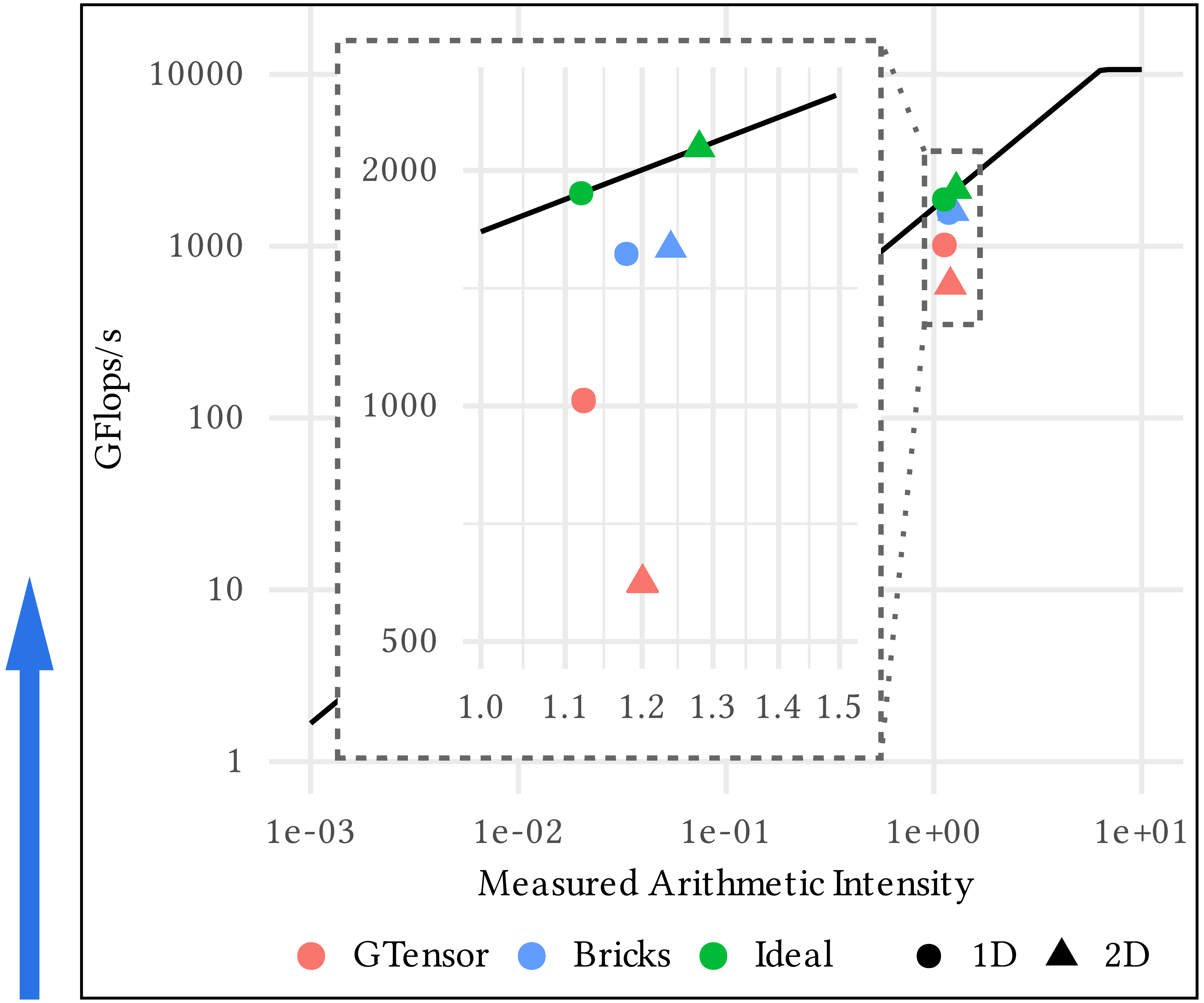
Accessed Part of Adjacency List

$$\Delta N = 0$$
$$\Delta M = 0$$
$$\Delta L \in \{-1, 0, 1\}$$
$$\Delta K \in \{-1, 0, 1\}$$
$$\Delta J = 0$$
$$\Delta I = 0$$

Stencils

Results

- NVIDIA A100 GPUs on Perlmutter
- 3.2% (2D) / 4.9% (1D) increase in arithmetic intensity
- 2.67x (2D) / 1.54x (1D) increase in GStencil/s
- Lower occupancy → increased L1 efficiency

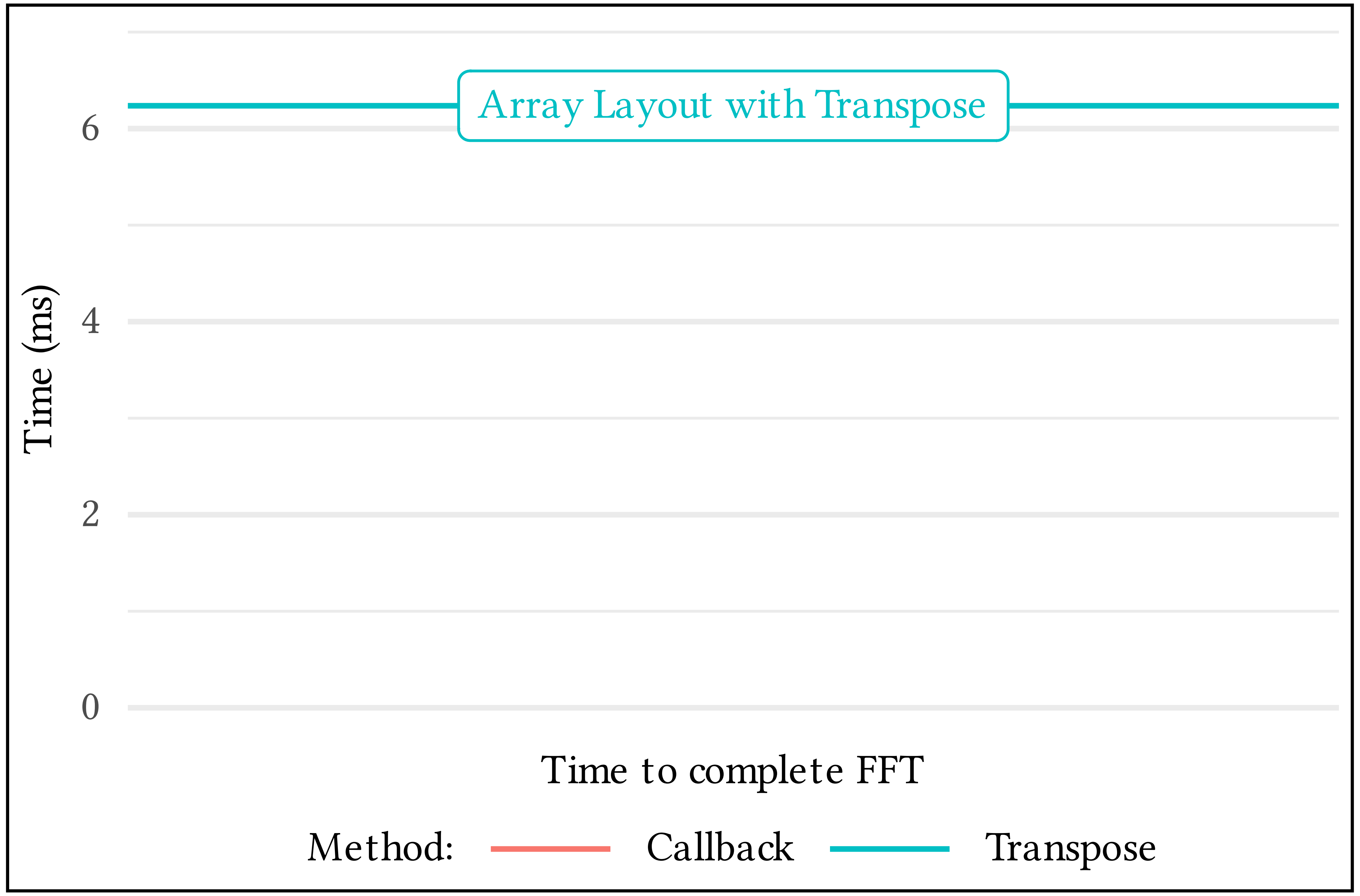


Higher is better

FFT

cuFFT for Bricks

- cuFFT along J-axis requires either:
  - Transpose I-J

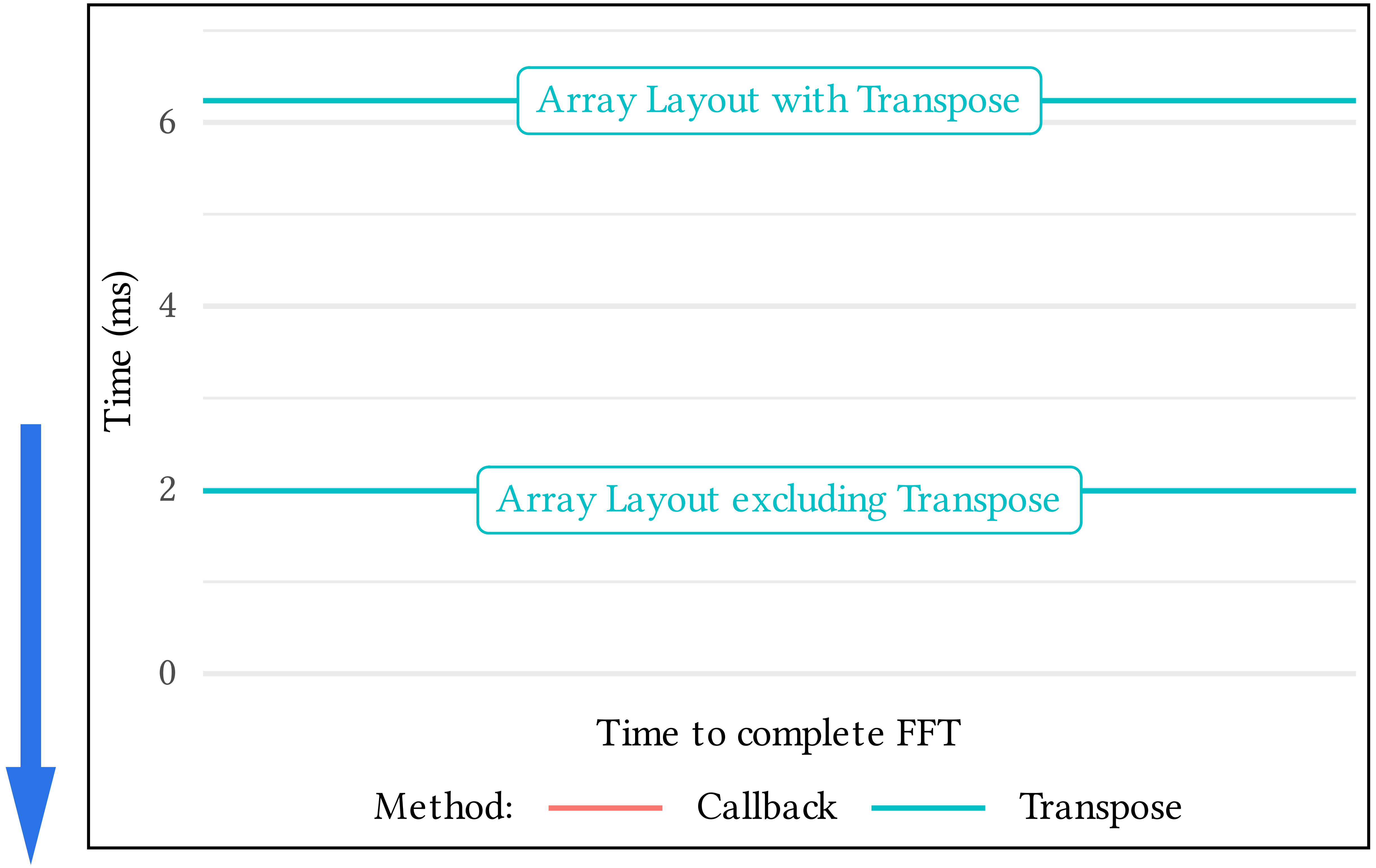


Lower is better

FFT

cuFFT for Bricks

- cuFFT along J-axis requires either:
  - Transpose I-J

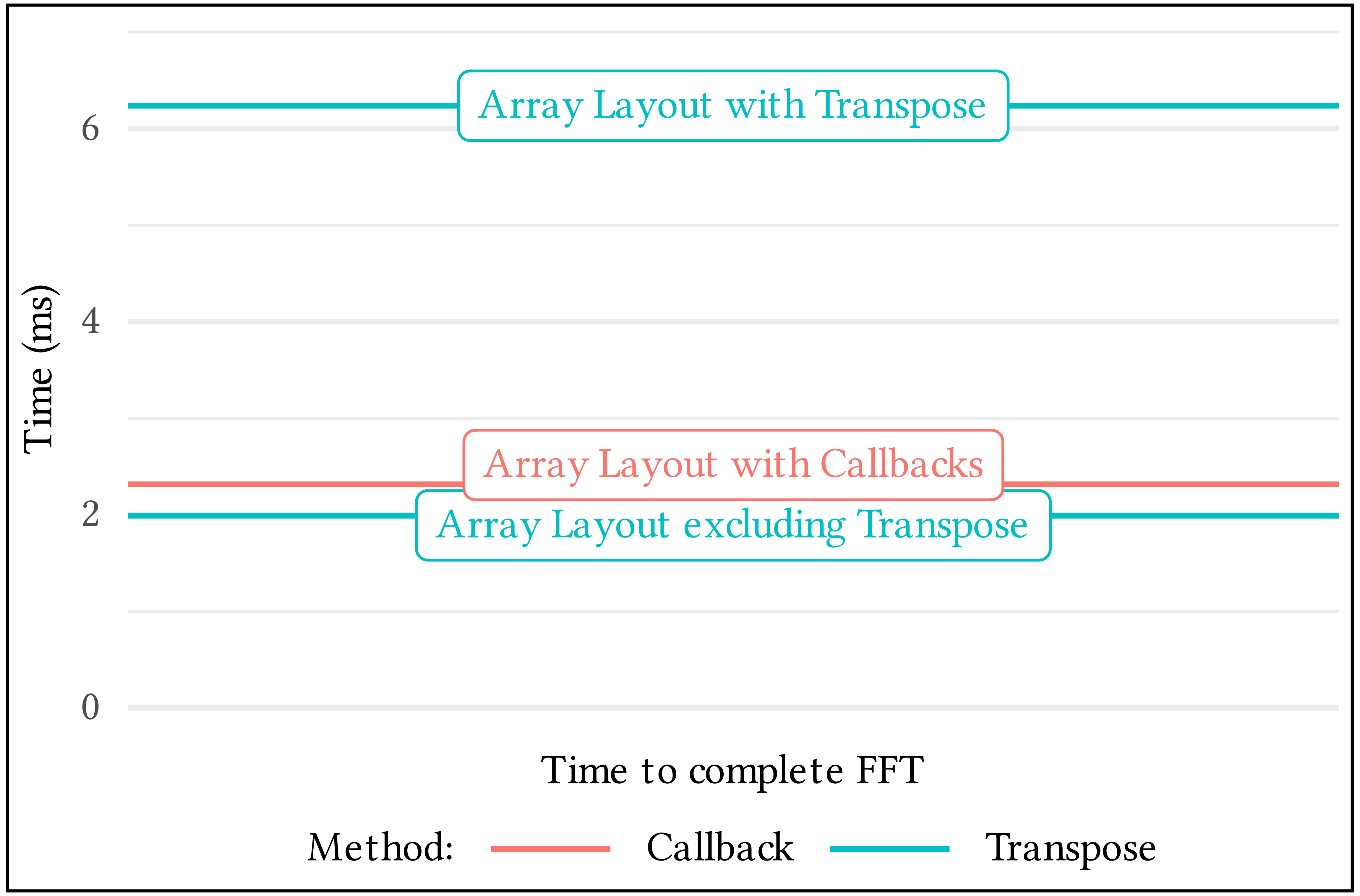


Lower is better

FFT

cuFFT for Bricks

- cuFFT along J-axis requires either:
  - Transpose I-J
  - User-defined callbacks

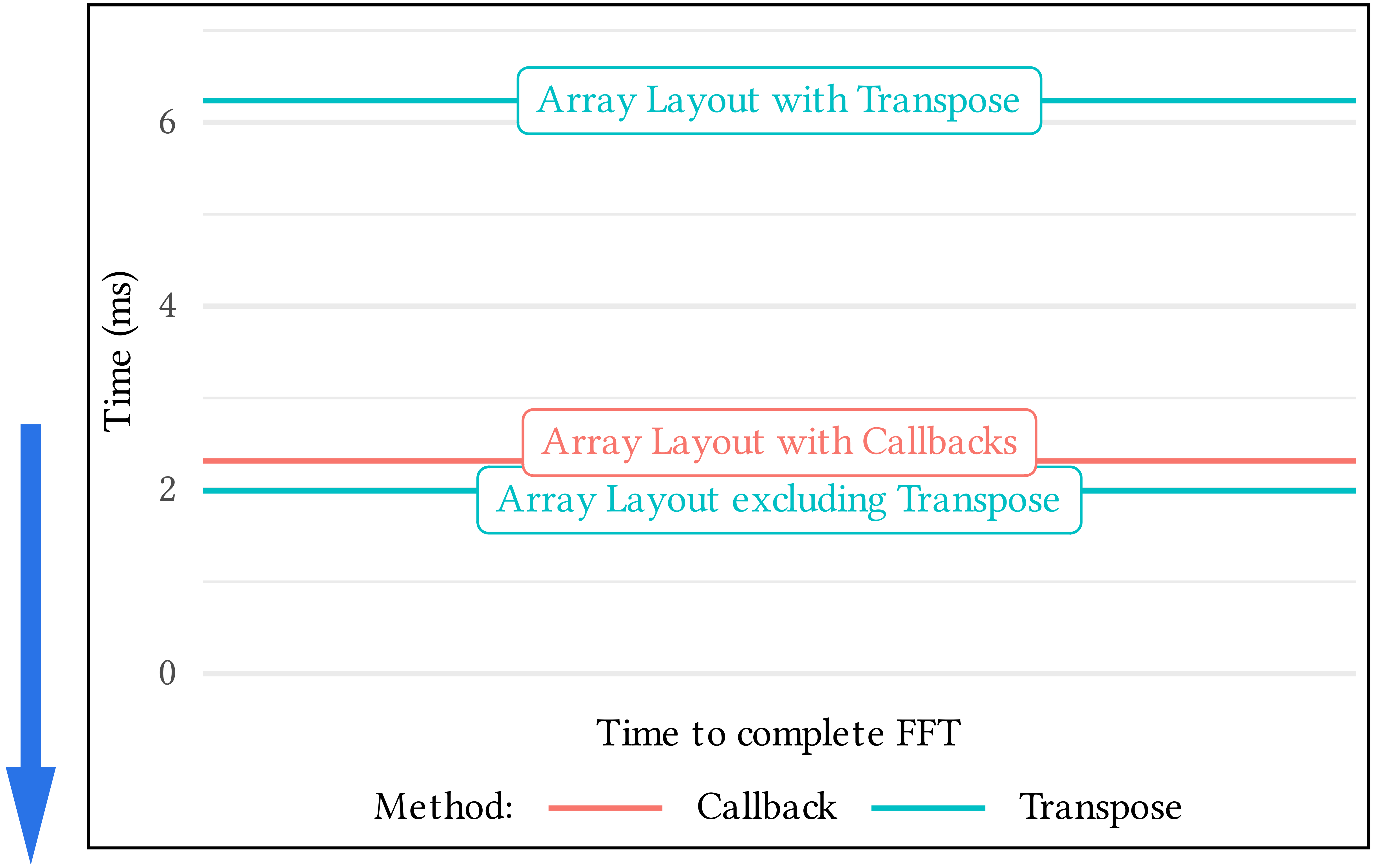


Lower is better

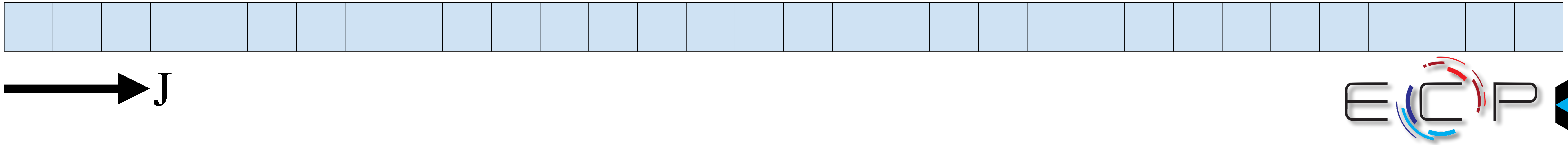
FFT

cuFFT for Bricks

- cuFFT along J-axis requires either:
  - Transpose I-J
  - User-defined callbacks
- Bricks implementation creates callbacks using C++ templates



Lower is better

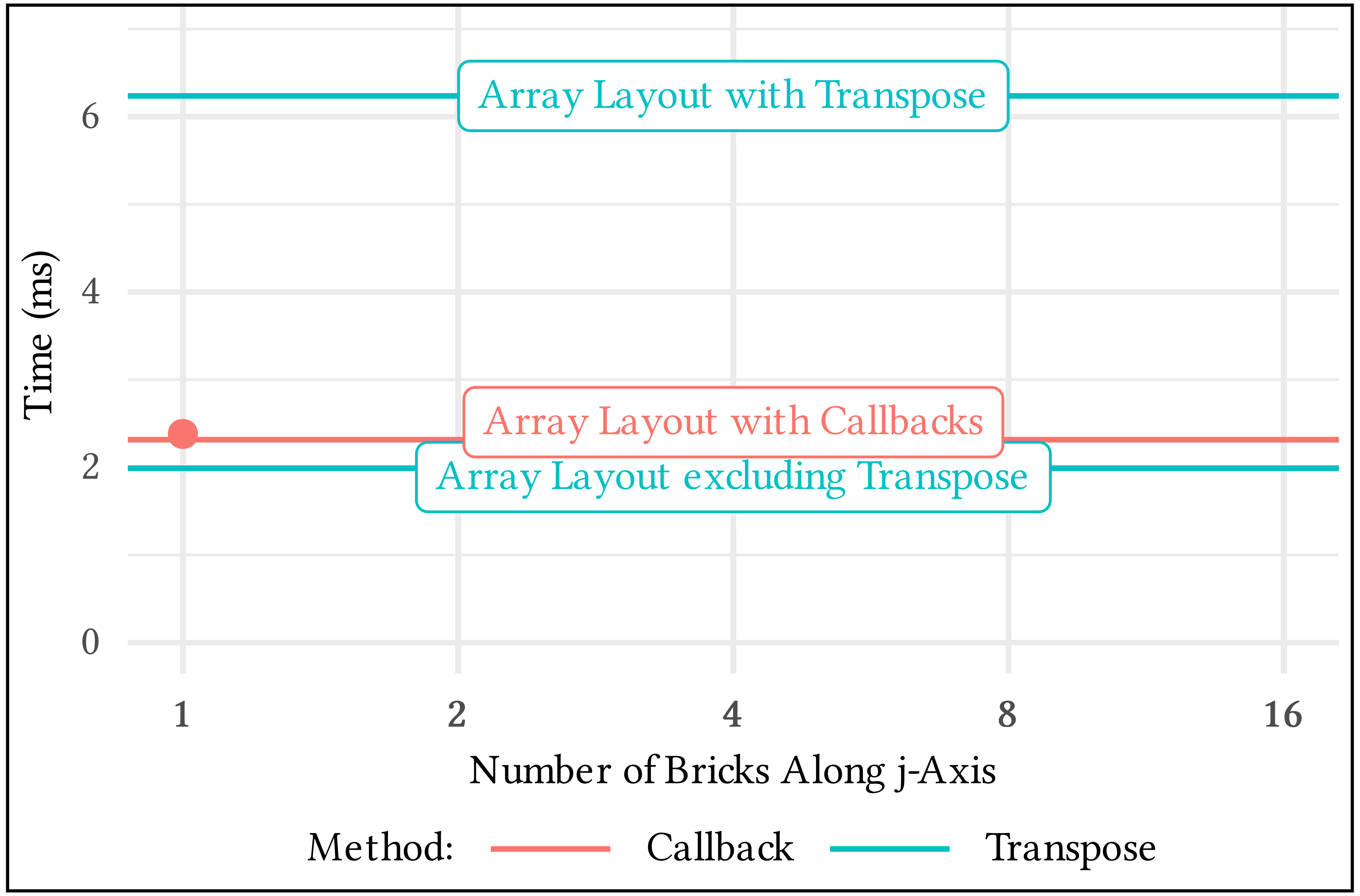


FFT

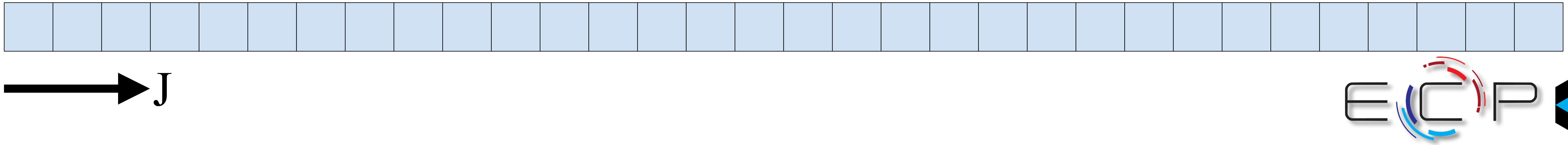
cuFFT for Bricks

- cuFFT along J-axis requires either:
  - Transpose I-J
  - User-defined callbacks

- Bricks implementation creates callbacks using C++ templates



Lower is better

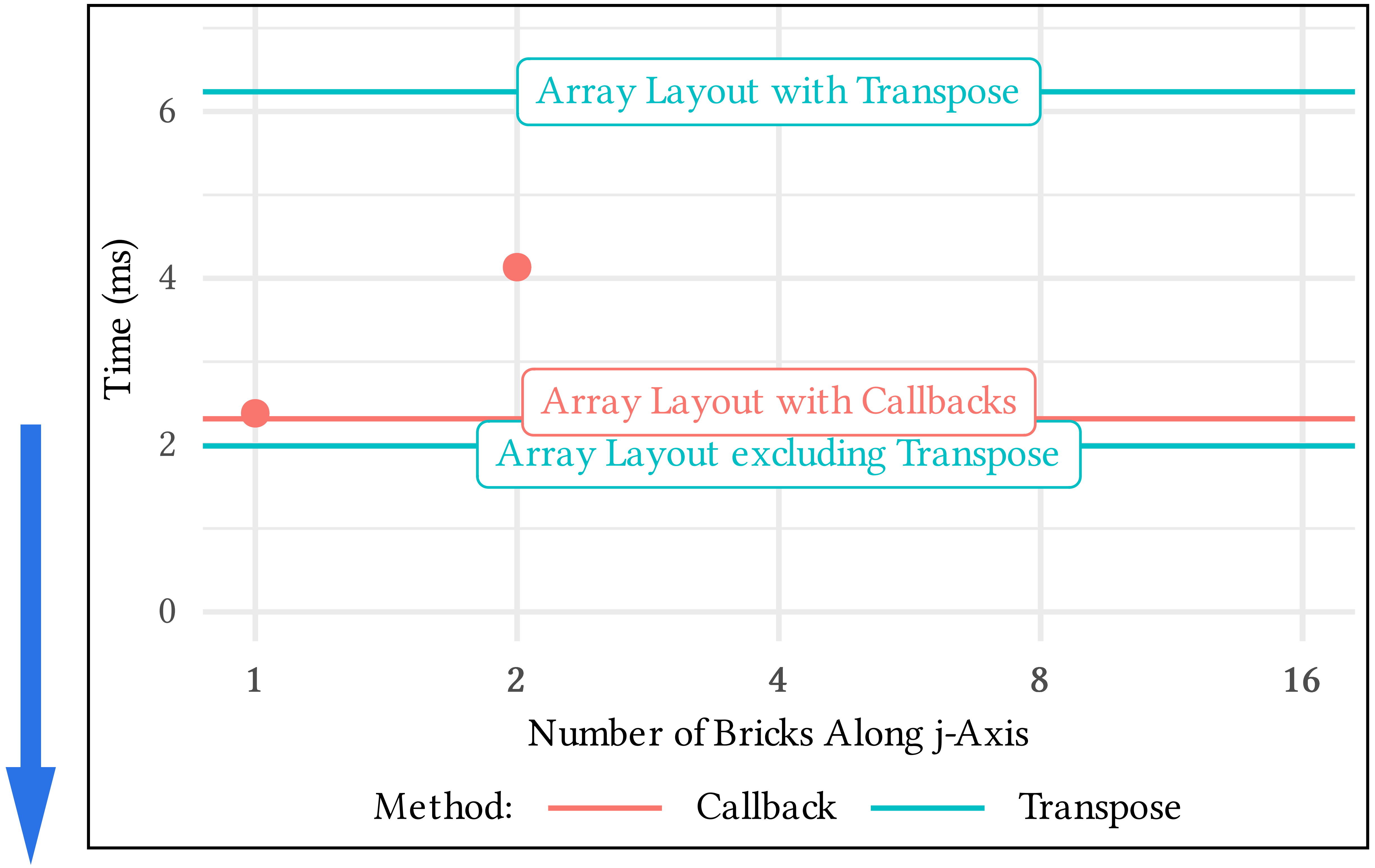


FFT

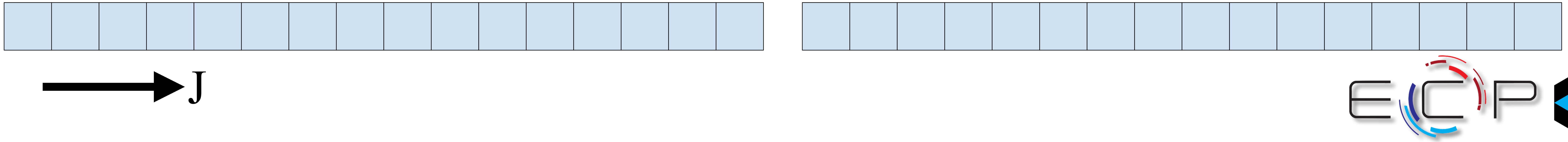
cuFFT for Bricks

- cuFFT along J-axis requires either:
  - Transpose I-J
  - User-defined callbacks

- Bricks implementation creates callbacks using C++ templates



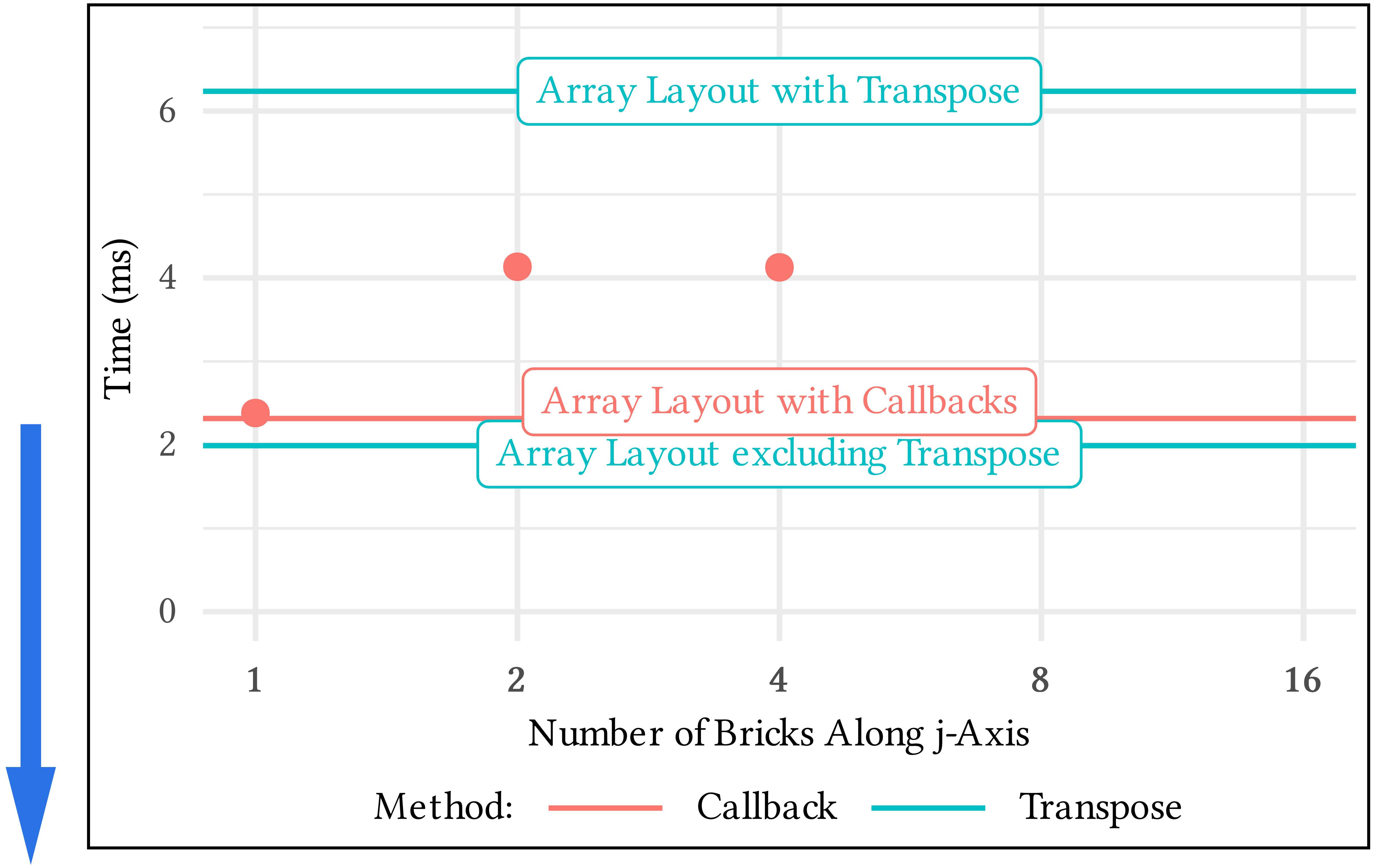
Lower is better



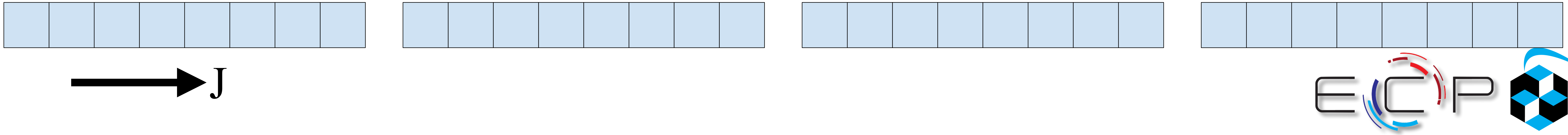
FFT

cuFFT for Bricks

- cuFFT along J-axis requires either:
  - Transpose I-J
  - User-defined callbacks
- Bricks implementation creates callbacks using C++ templates



Lower is better

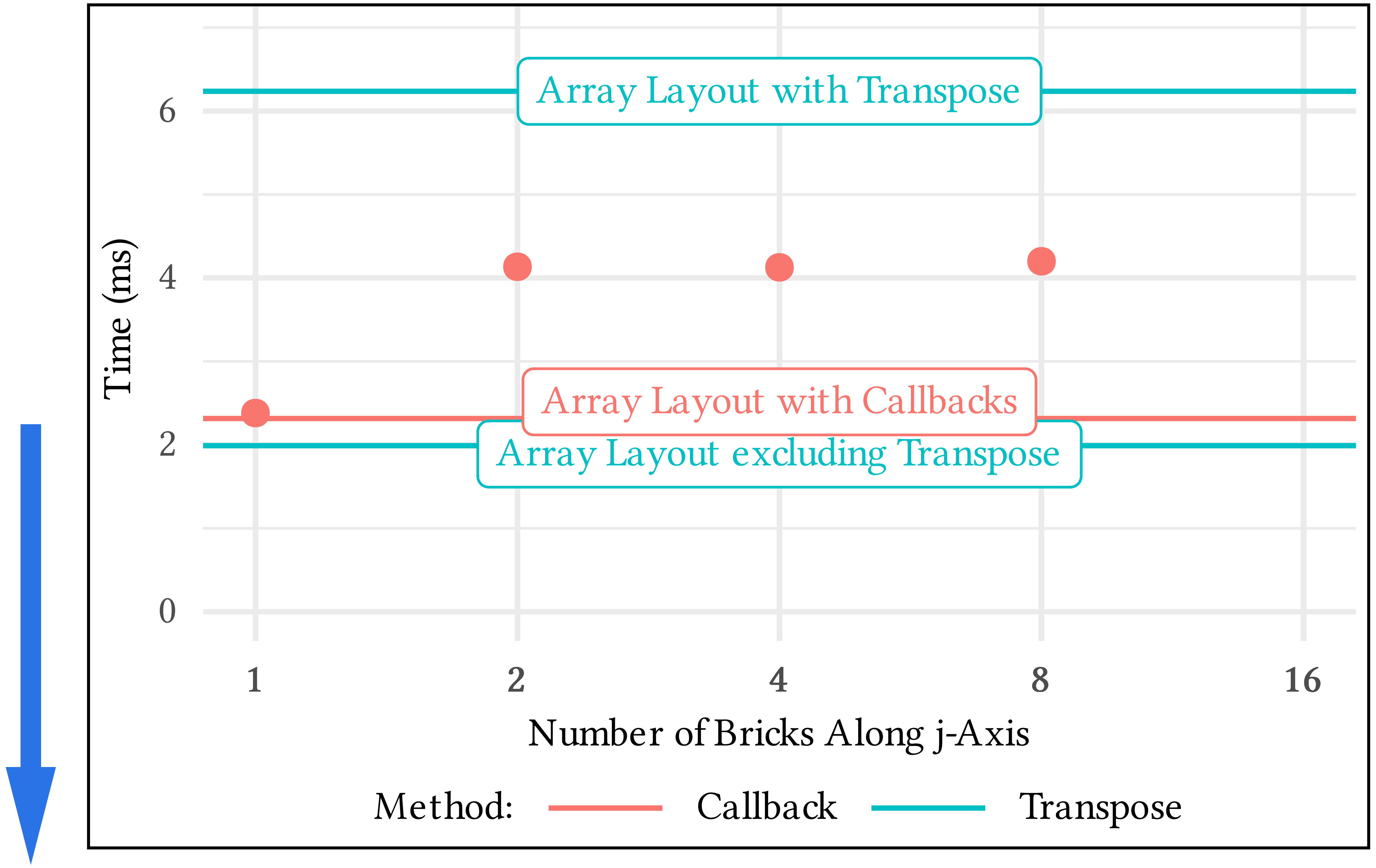


FFT

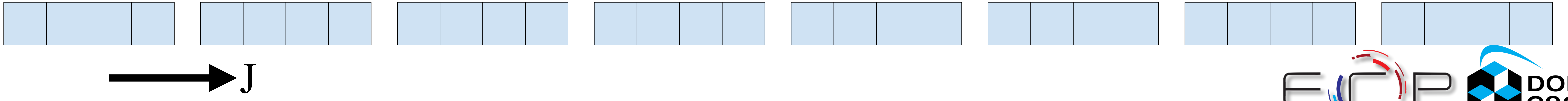
cuFFT for Bricks

- cuFFT along J-axis requires either:
    - Transpose I-J
    - User-defined callbacks

- Bricks implementation creates callbacks using C++ templates



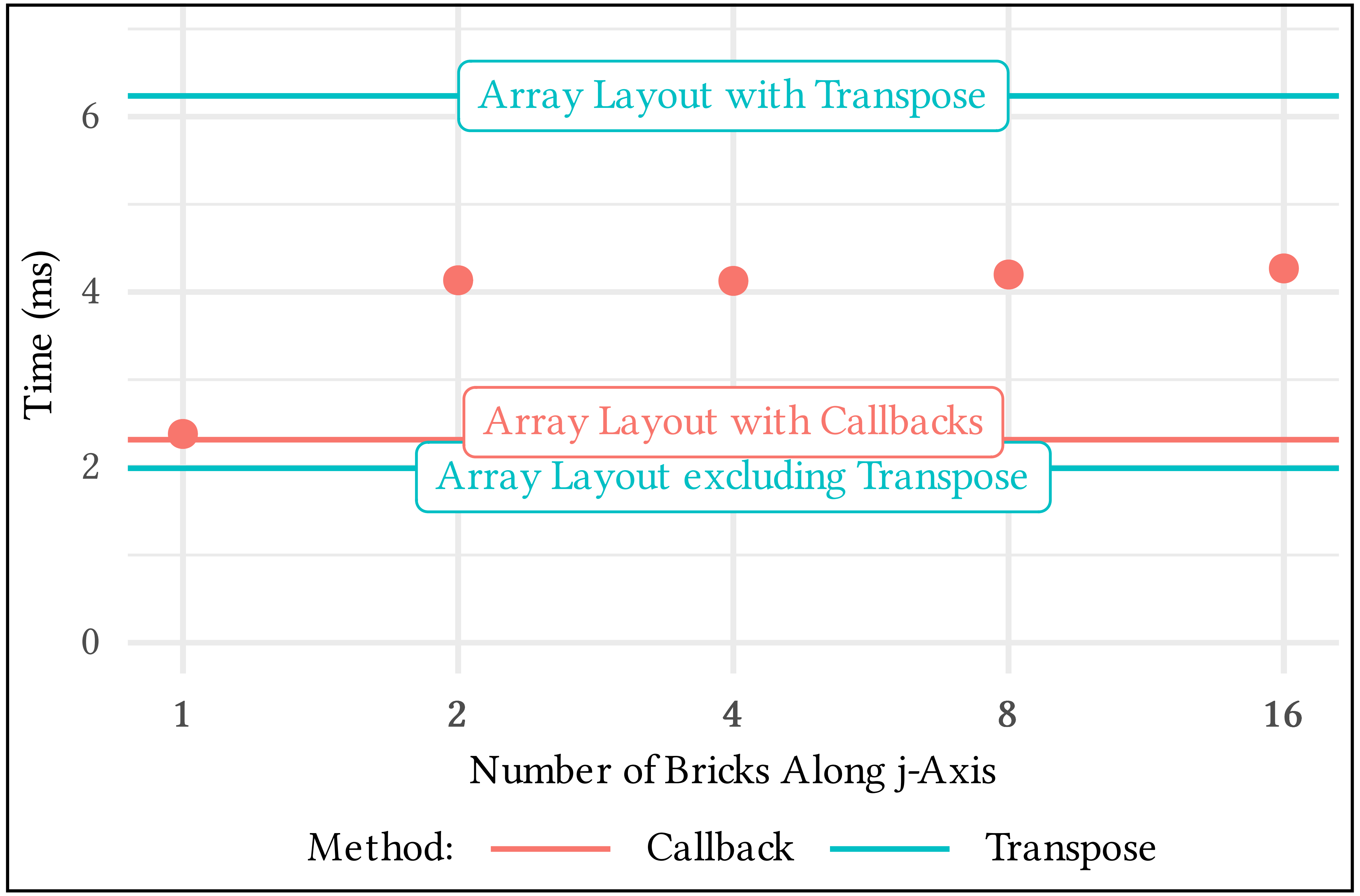
Lower is better



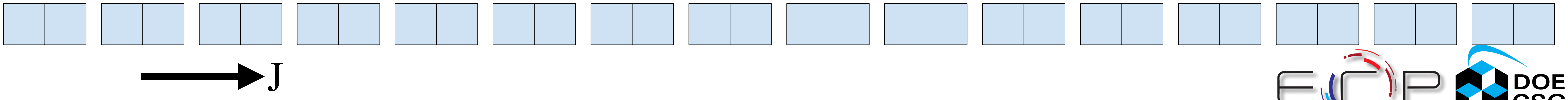
FFT

cuFFT for Bricks

- cuFFT along J-axis requires either:
  - Transpose I-J
  - User-defined callbacks
- Bricks implementation creates callbacks using C++ templates



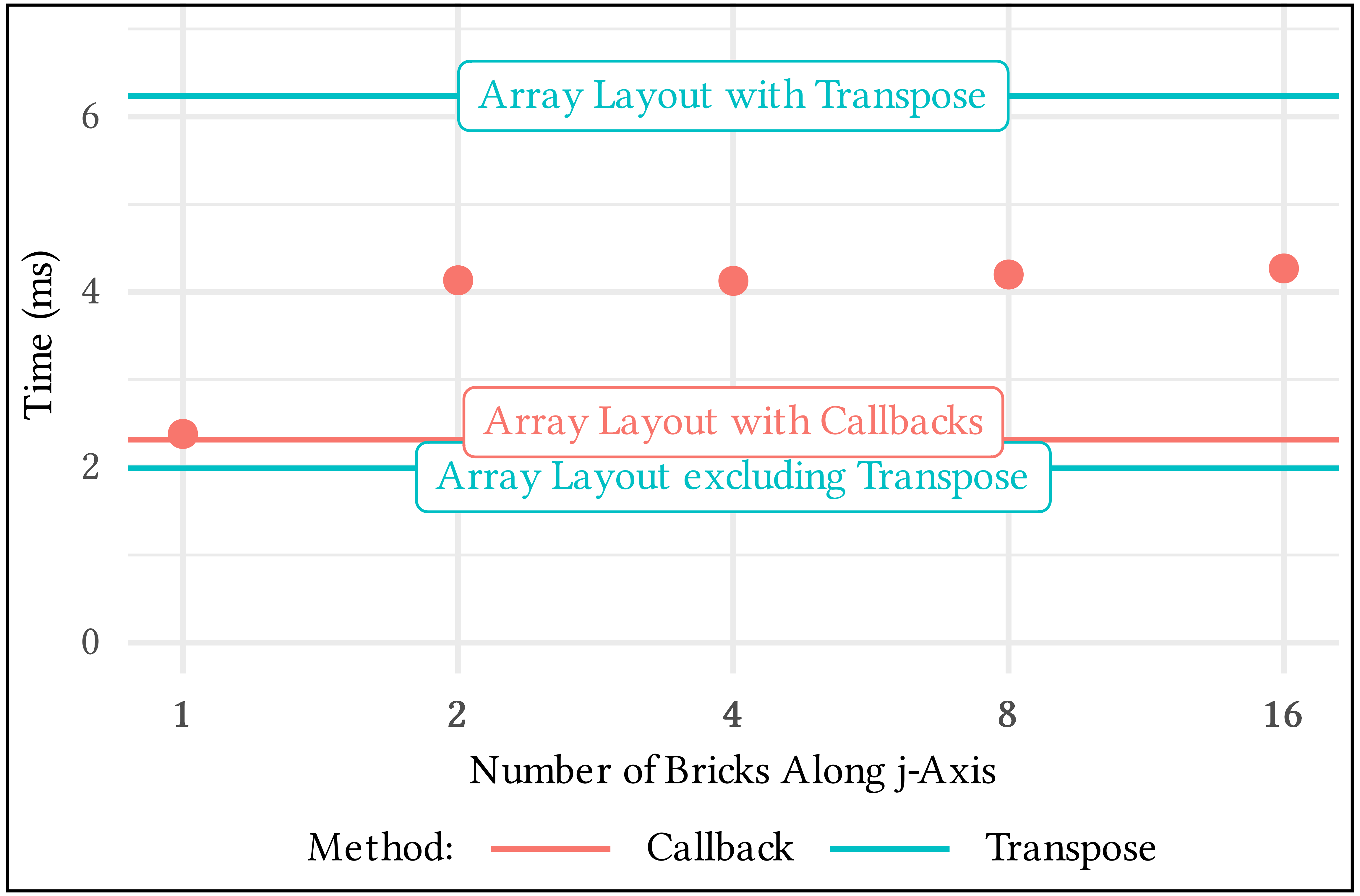
Lower is better



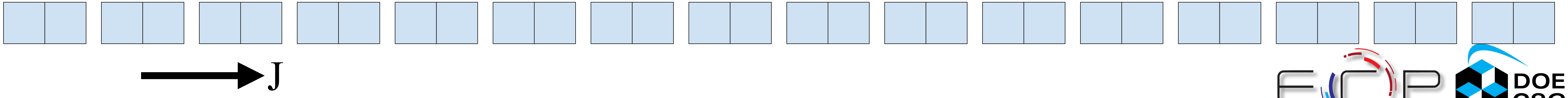
FFT

cuFFT for Bricks

- cuFFT along J-axis requires either:
  - Transpose I-J
  - User-defined callbacks
- Bricks implementation creates callbacks using C++ templates
- Slowdown 1.83x-1.93x moving from 1 Brick/FFT → 16 Bricks/FFT



Lower is better



Summary

- We extended the Bricks layout/library to efficiently handle high-dimensional layouts, complex types, and FFT computations on GPUs

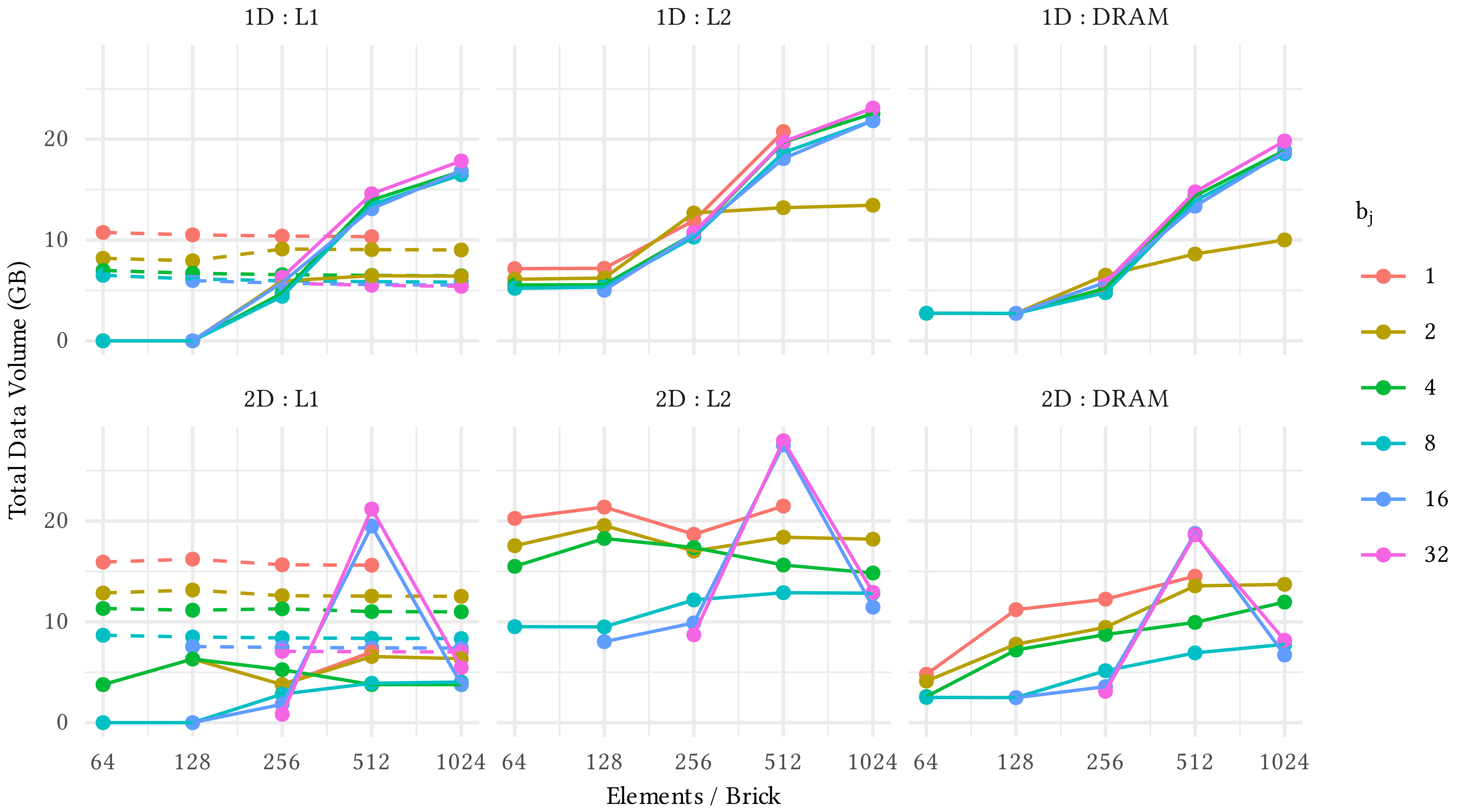
- Bricks can navigate trade-offs in high-dimensional settings by tuning Brick shape

- **Moving Forward:** Other high-dimensional applications such as QCD, custom Bricks solutions to FFTs and other non-stencil operations

Backup Slides

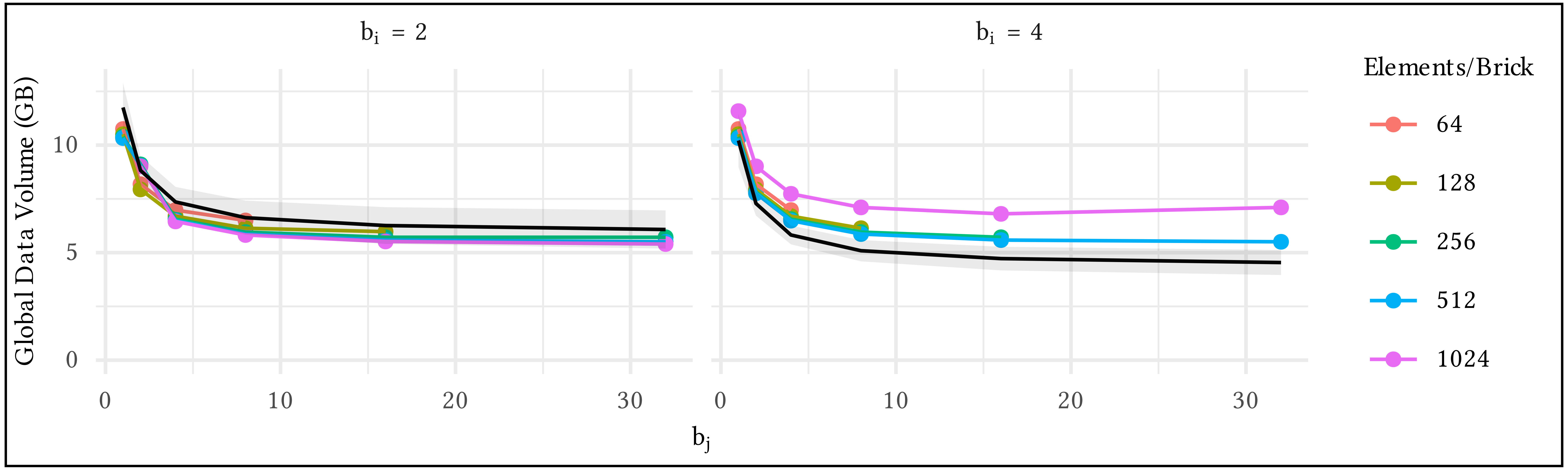
Tuning Bricks

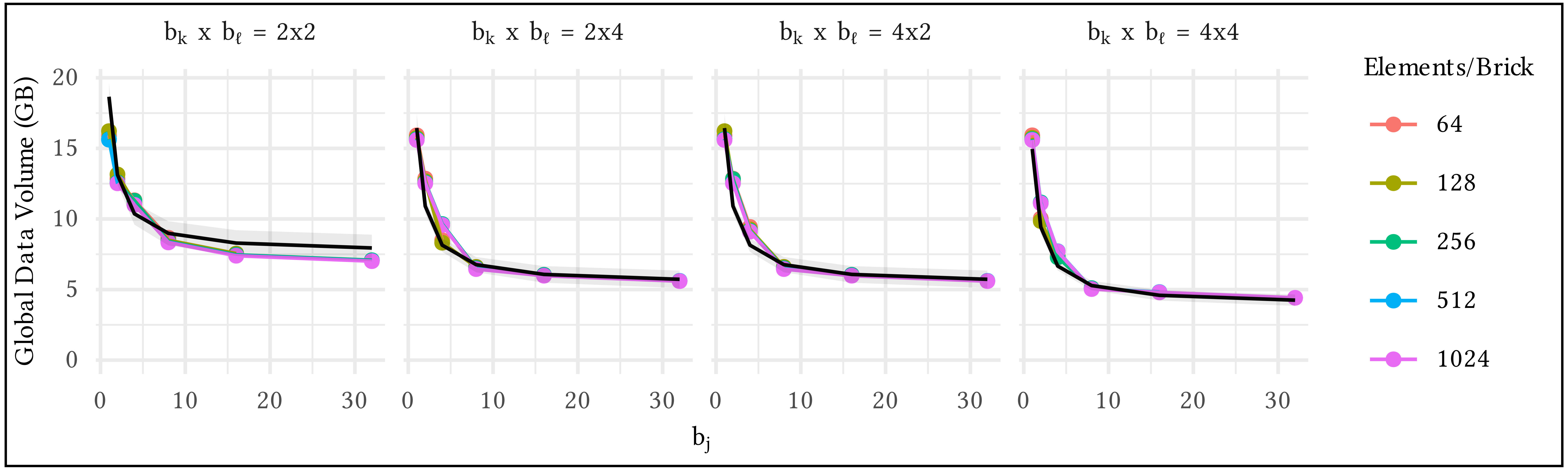
Hardware Resource Usage



Tuning Bricks

Bricks Shape


b_i = 2
b_i = 4
Global Data Volume (GB)
b_j
Elements/Brick


b_k x b_ℓ = 2x2
b_k x b_ℓ = 2x4
b_k x b_ℓ = 4x2
b_k x b_ℓ = 4x4
Global Data Volume (GB)
b_j
Elements/Brick