Data Structures in PyFR

F.D. Witherden
Department of Ocean Engineering, Texas A&M University

Overview

- One of the keys to extracting good performance is choosing the right **data structures**.

Principles

- Within the context of modern hardware it is important to keep the following points in mind.

Principles

1. On GPUs host to device data transfers are expensive.

    - All **kernels must therefore be offloaded** onto the GPU to there is no need to routinely move large chunks of data.

Principles

2. Memory allocation is expensive.

   - Memory should therefore be **allocated up-front** with no allocations inside the critical path.

Principles

3. Memory bandwidth is a scarce resource.

    - We should therefore design our kernels to minimise data movement even if this means **recomputing quantities on the fly**.

Principles

4. Coalesced access is vital for achieving good bandwidth.

    - Data must be arranged in a manner which **comports with the access patterns** of kernels. **Indirection should be avoided** whenever possible.

Principles

- These principles have dictated our choice of data structures within PyFR.

Data Layouts

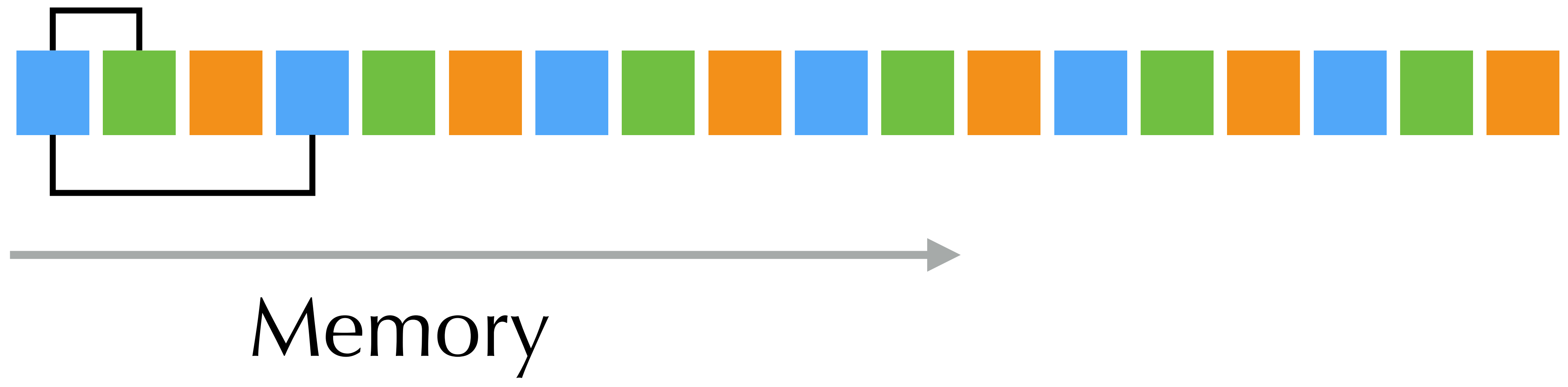
Data Layouts

- At the highest level there are three main layouts:
  - AoS
  - SoA
  - AoSoA(k)

Data Layouts: AoS

```
struct
{
    float rho;
    float rhou;
    float E;
} data[NELES];
```

Data Layouts: AoS

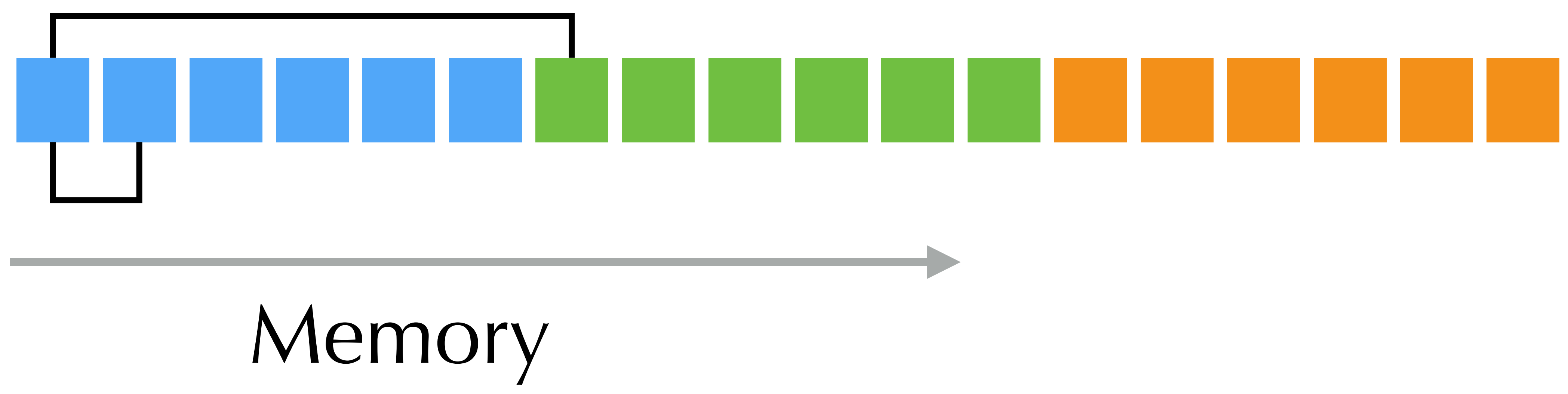


- Cache and TLB friendly.
- Difficult to vectorise.

Data Layouts: SoA

```
struct
{
    float rho[NELES];
    float rhou[NELES];
    float E[NELES];
} data;
```

Data Layouts: SoA

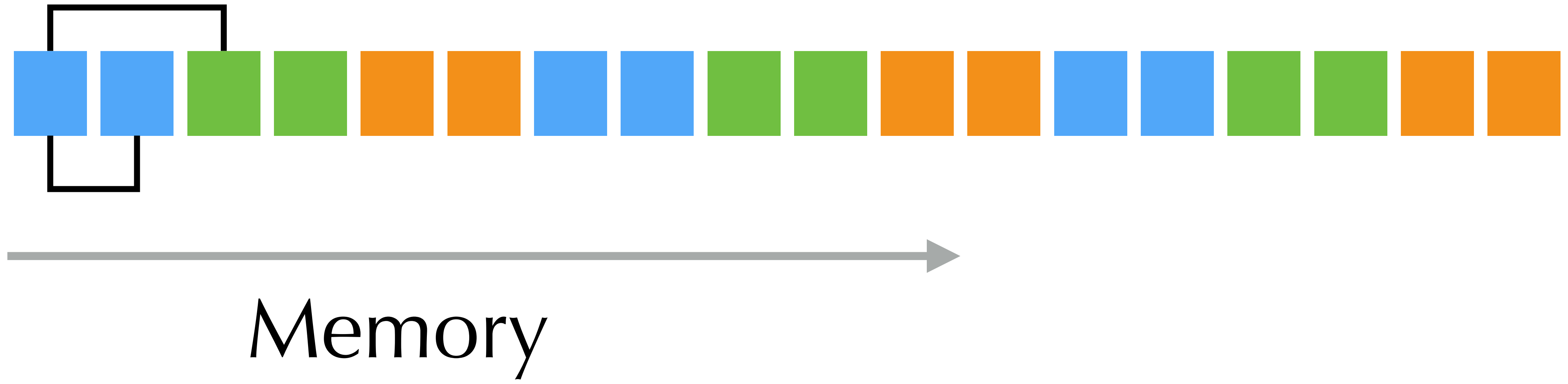


- Trivial to vectorise.
- Can put pressure on TLB and/or hardware pre-fetchers.

Data Layouts: AoSoA($k = 2$)

```
struct
{
    float rho[k];
    float rhou[k];
    float E[k];
} data[NELES / k];
```

Data Layouts: AoSoA($k = 2$)



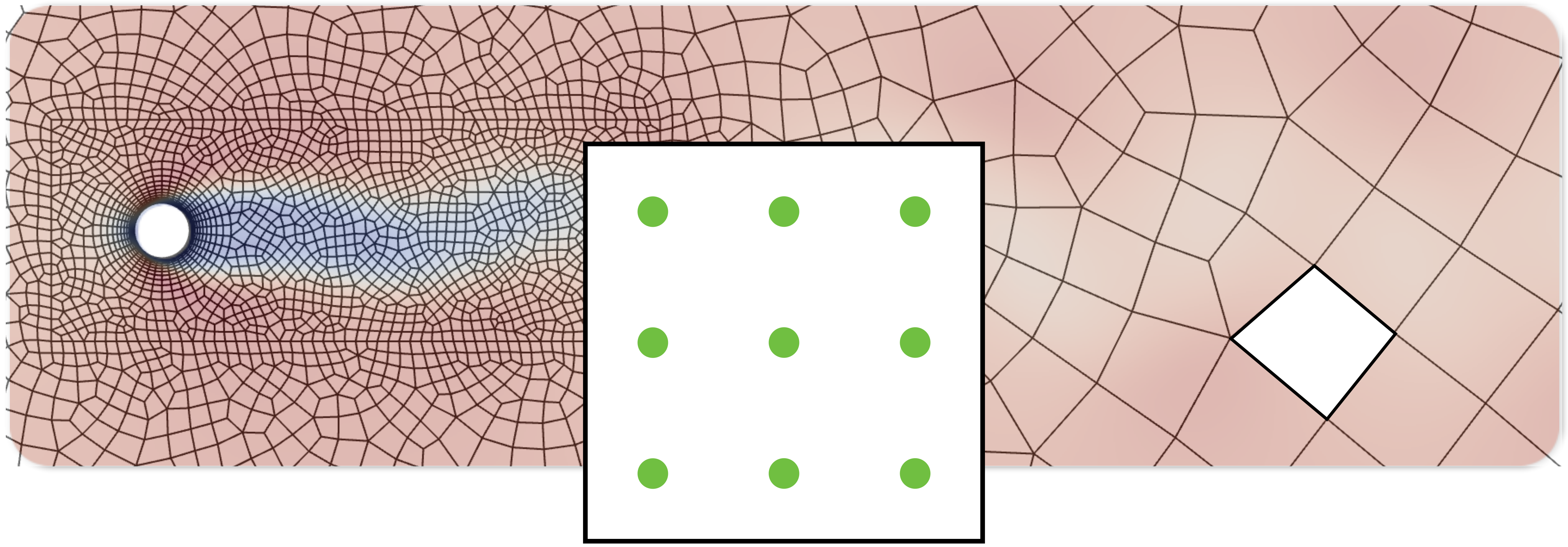
- Can be vectorised efficiently for suitable k.
- Cache and TLB friendly.

Data Layouts: AoSoA($k = 2$)

- The ideal solution
  - …albeit at the cost of **messy indexing**
  - …and requires **coaxing for compilers to vectorise**.

Matrices

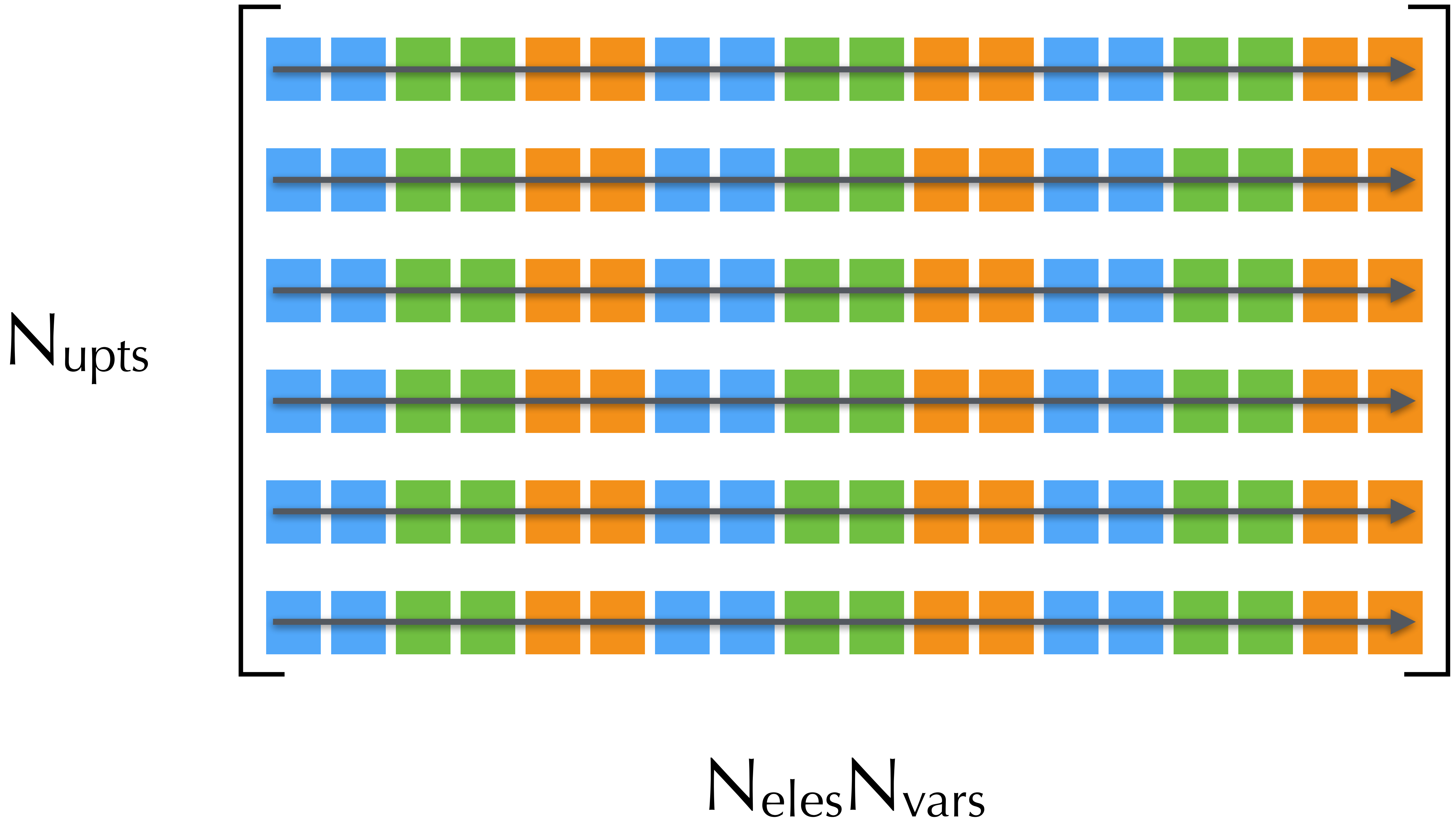
Matrices



- Recall in FR we have a **set of elements** with each element containing some number of **solution points**.

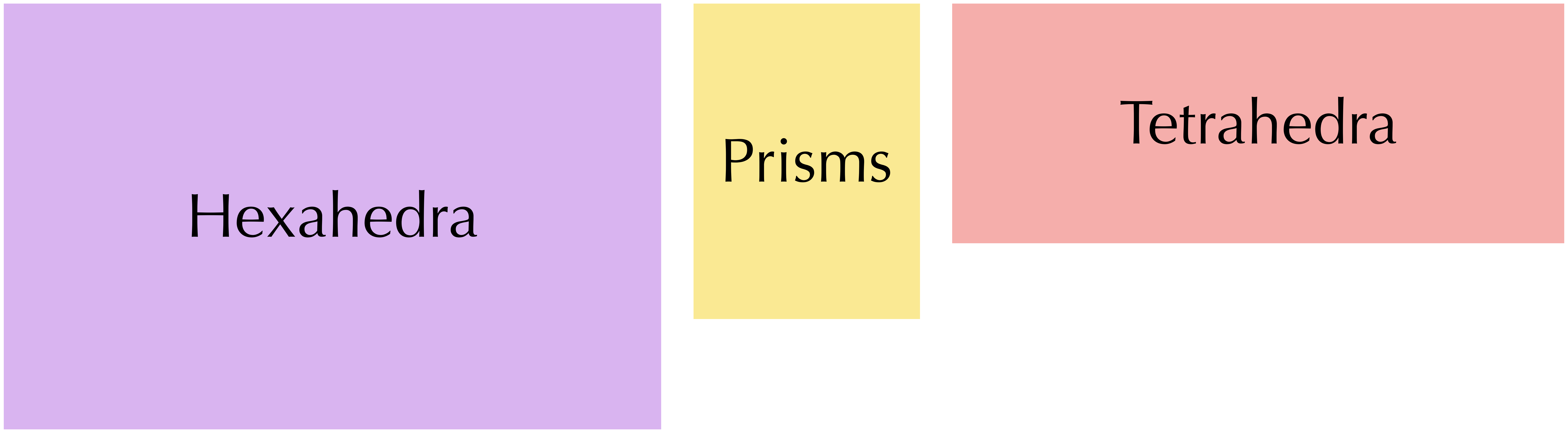
Matrices

- The natural structure for this is a type of **matrix**.



Matrices

- In mixed grid cases we have **one matrix per element type**.



Matrices

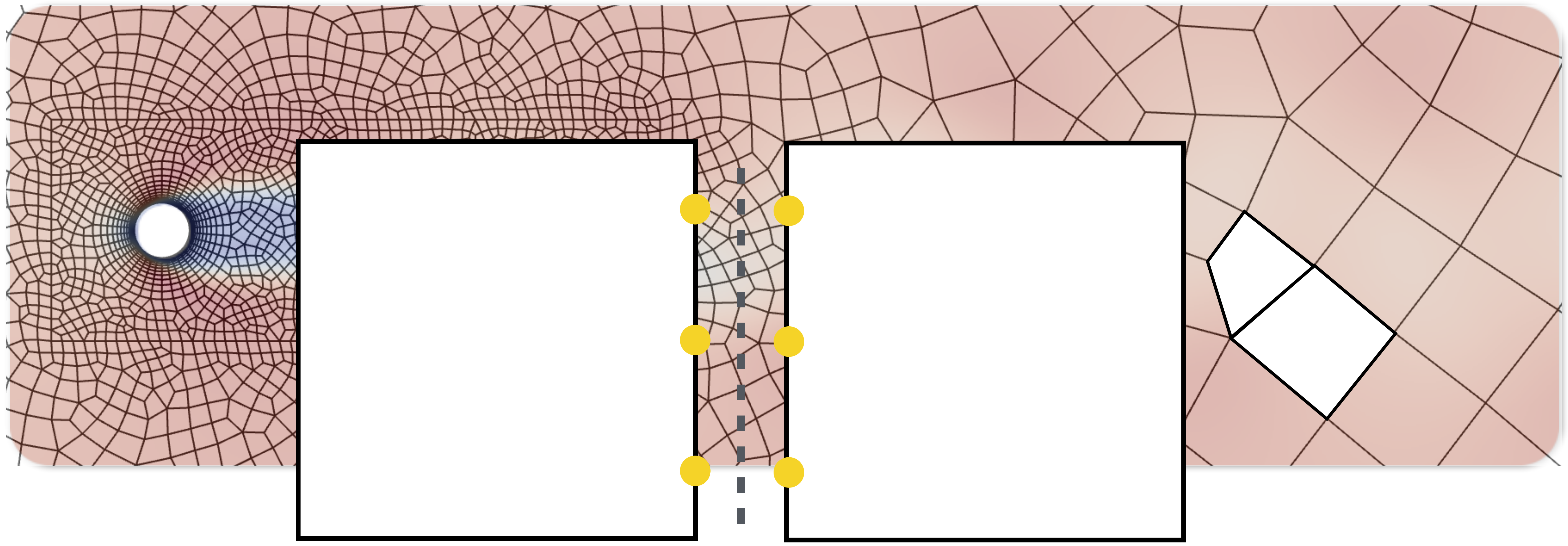
- For storing quantities such as the flux or solution gradient it is convenient to introduce a **fourth pseudo-axis**.

$N_{upts}N_{dims}$

$N_{eles}N_{vars}$

Views

Views



- In FR some operations are **inherently indirect in nature**.

Views

- We handle this through a data structure called a **view**.
- They can be thought of as a **matrix of pointers** to elements in other matrices.

Matrix Banks

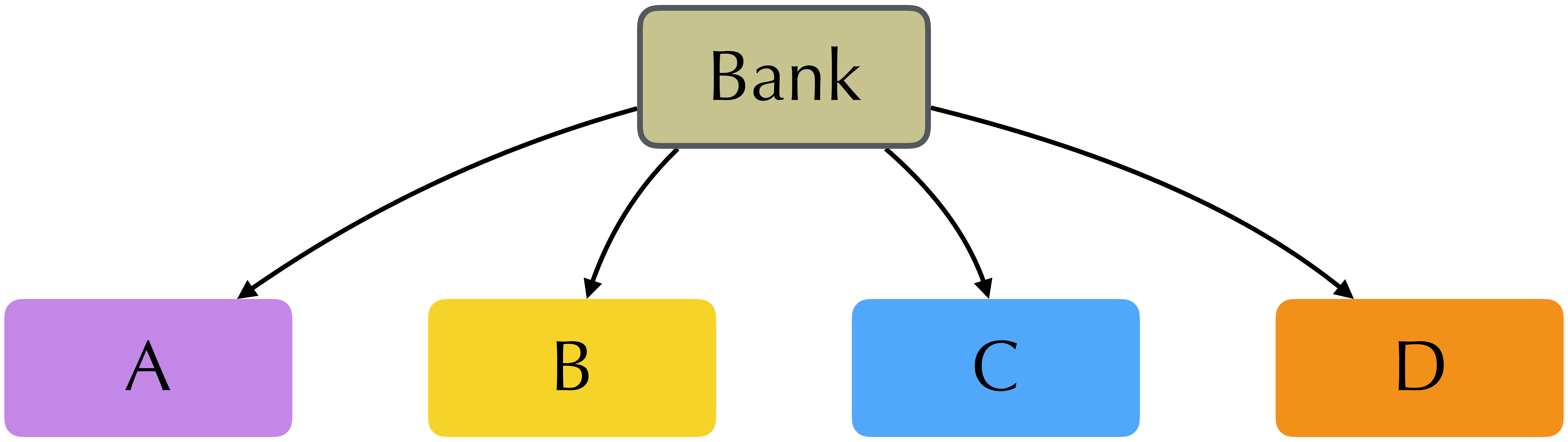
Matrix Banks

- In PyFR all **kernel arguments** must be specified at the **point of instantiation**.
- Whencreating a kernel it is generally necessary to specify exactly what piece of data it will operate on.

Matrix Banks

- One way of relaxing this is through **matrix banks**.
- A matrix bank is a collection of **equivalently sized matrices**.
- At any instant in time the bank **impersonates** one of these matrices.

Matrix Banks



Matrix Banks

- We remark here that it is possible for a matrix to be in **multiple banks**.

- This can enable a single kernel for common operations such as $\mathbf{C} = \mathbf{A} + \mathbf{B}$ without needing to instantiate a distinct kernel for every possible permutation of arguments.