# NektarIR: a domain-specific compiler for high-order finite element operations on heterogeneous hardware

**David Moxey, Edward Erasmie-Jones**

Department of Engineering, King's College London

**Giacomo Castiglioni**

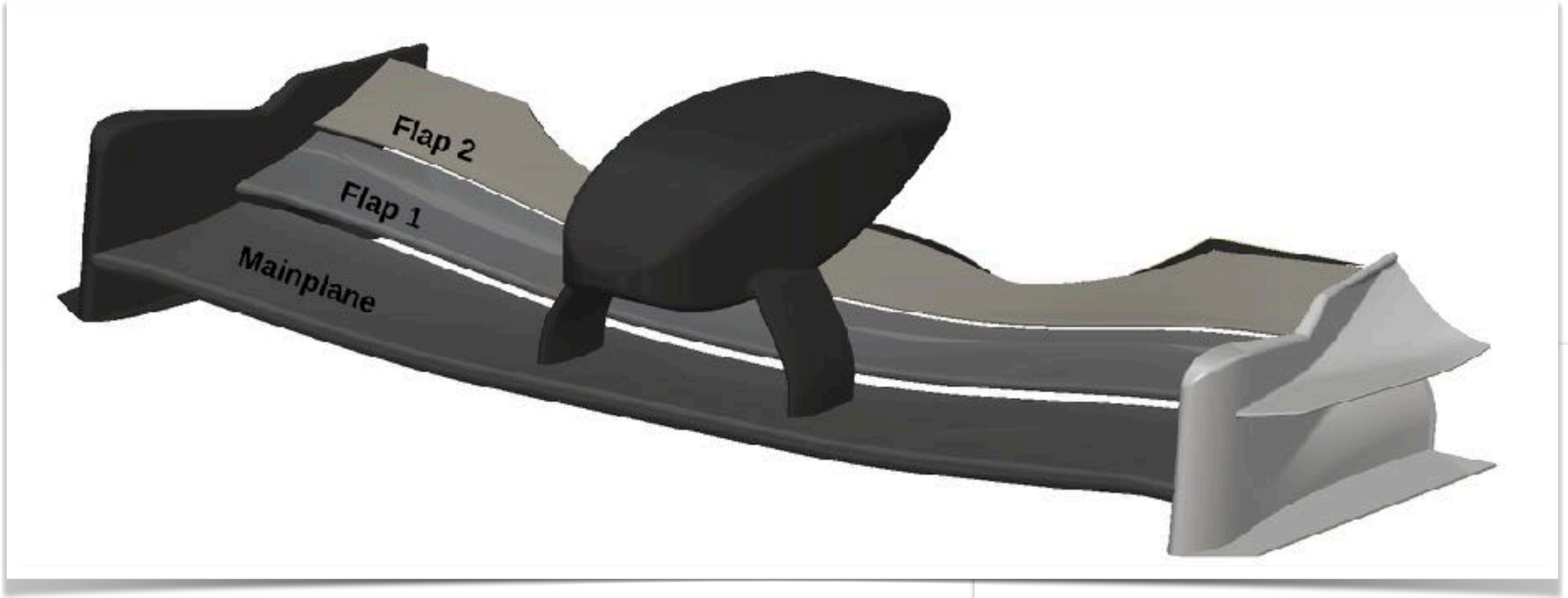
CSCS, Switzerland

**MFEM Seminar Series, 9th February 2026**

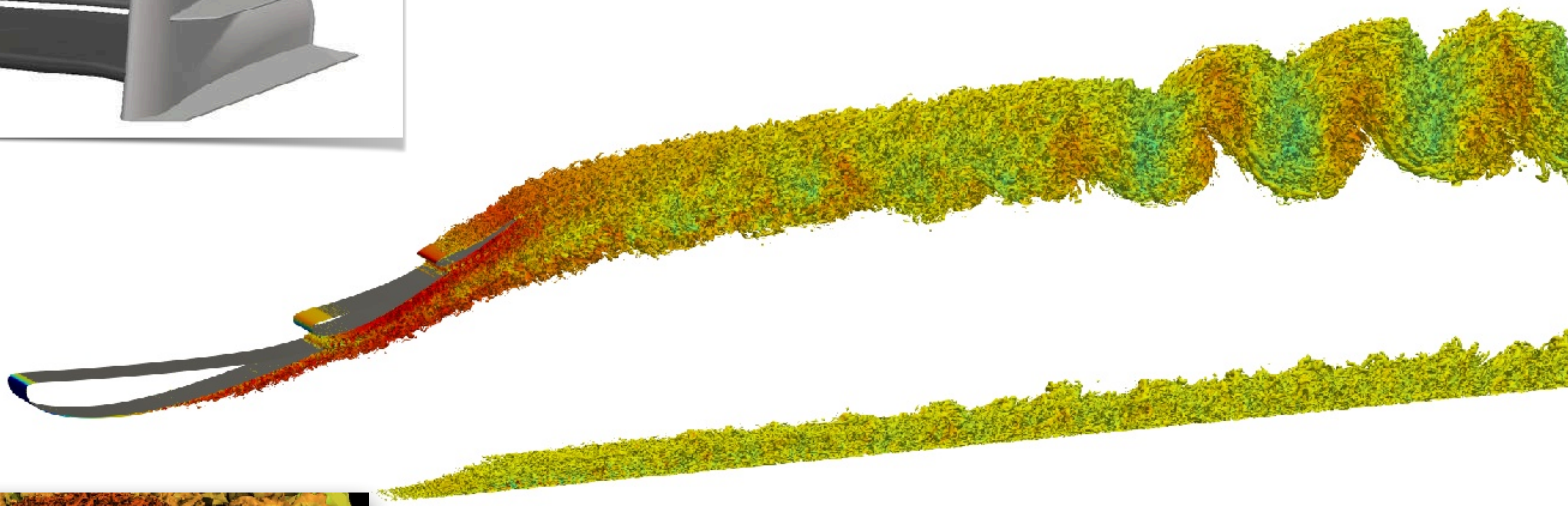
# Outline

- Motivation & Nektar++
- Efficient high-order FEM operations on non-tensorial elements
- NektarIR: a domain-specific compiler for high-order FEM operations
- Summary

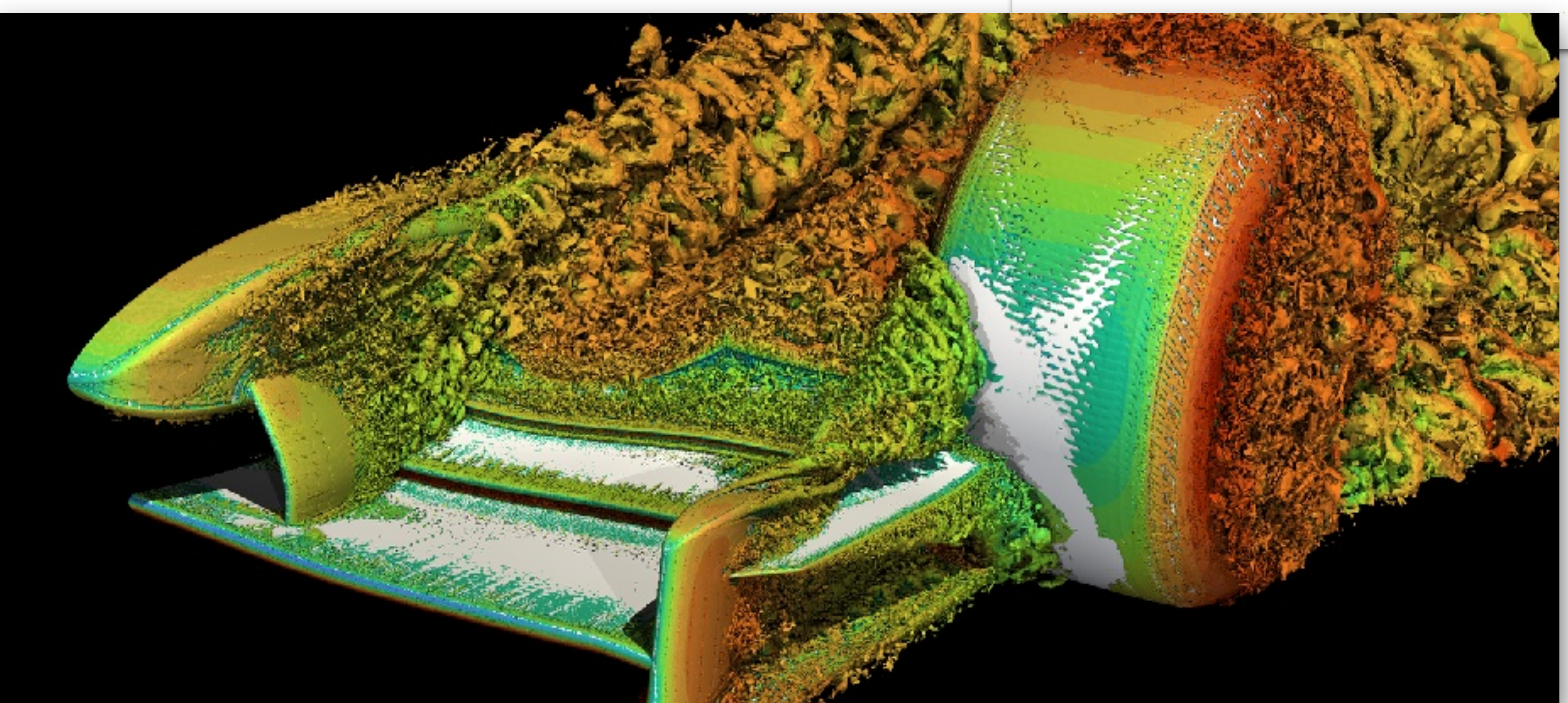
**Goal:** enable scale-resolving simulations for complex geometries of interest to industry at high order.




Imperial Front Wing: a prototype F1 geometry



High-fidelity simulation of a IFW cross-section



Full 3D simulation of IFW + rotating wheel

Nektar++

*spectral/hp element framework*

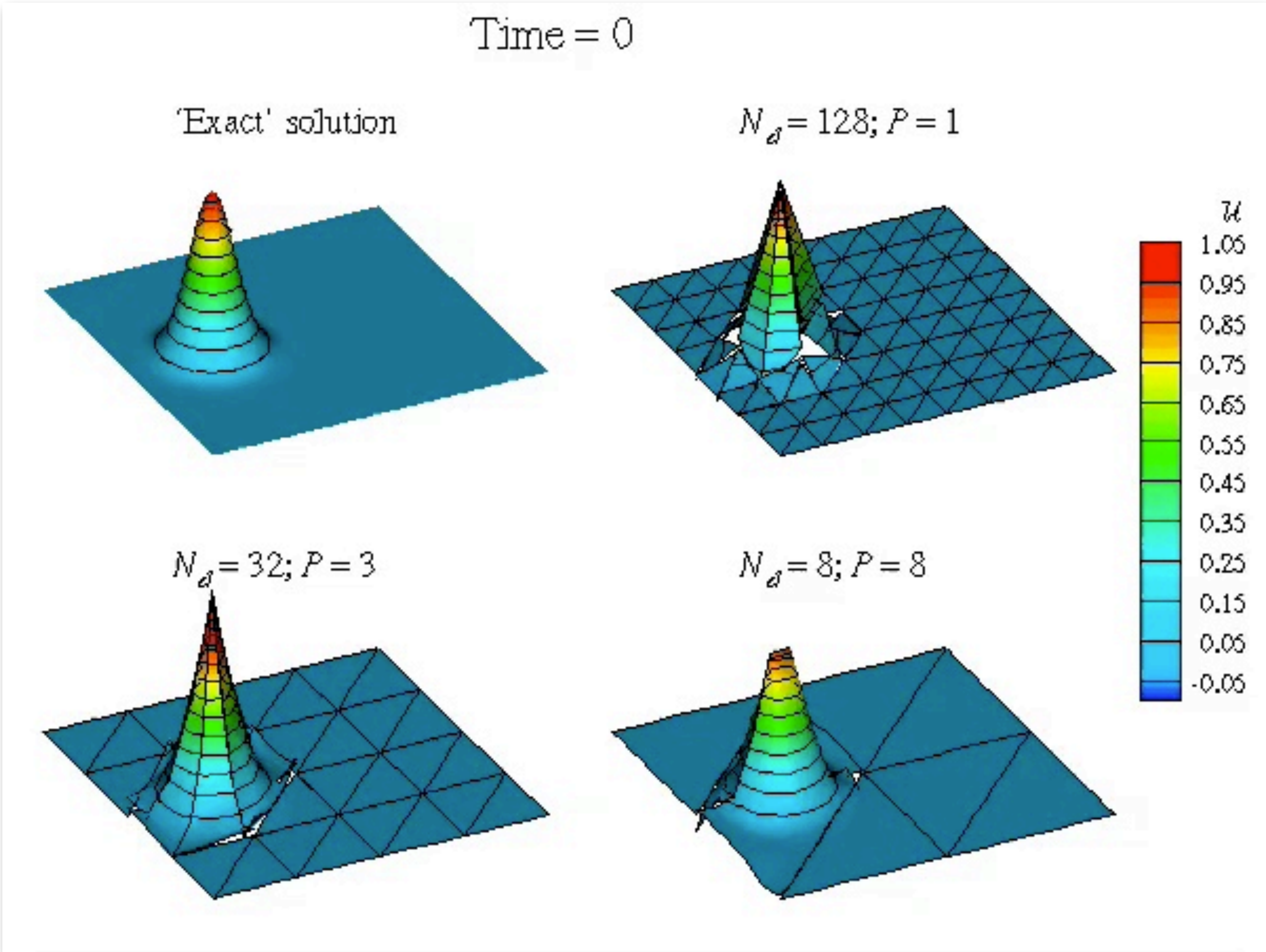
# Why use a high-order method?

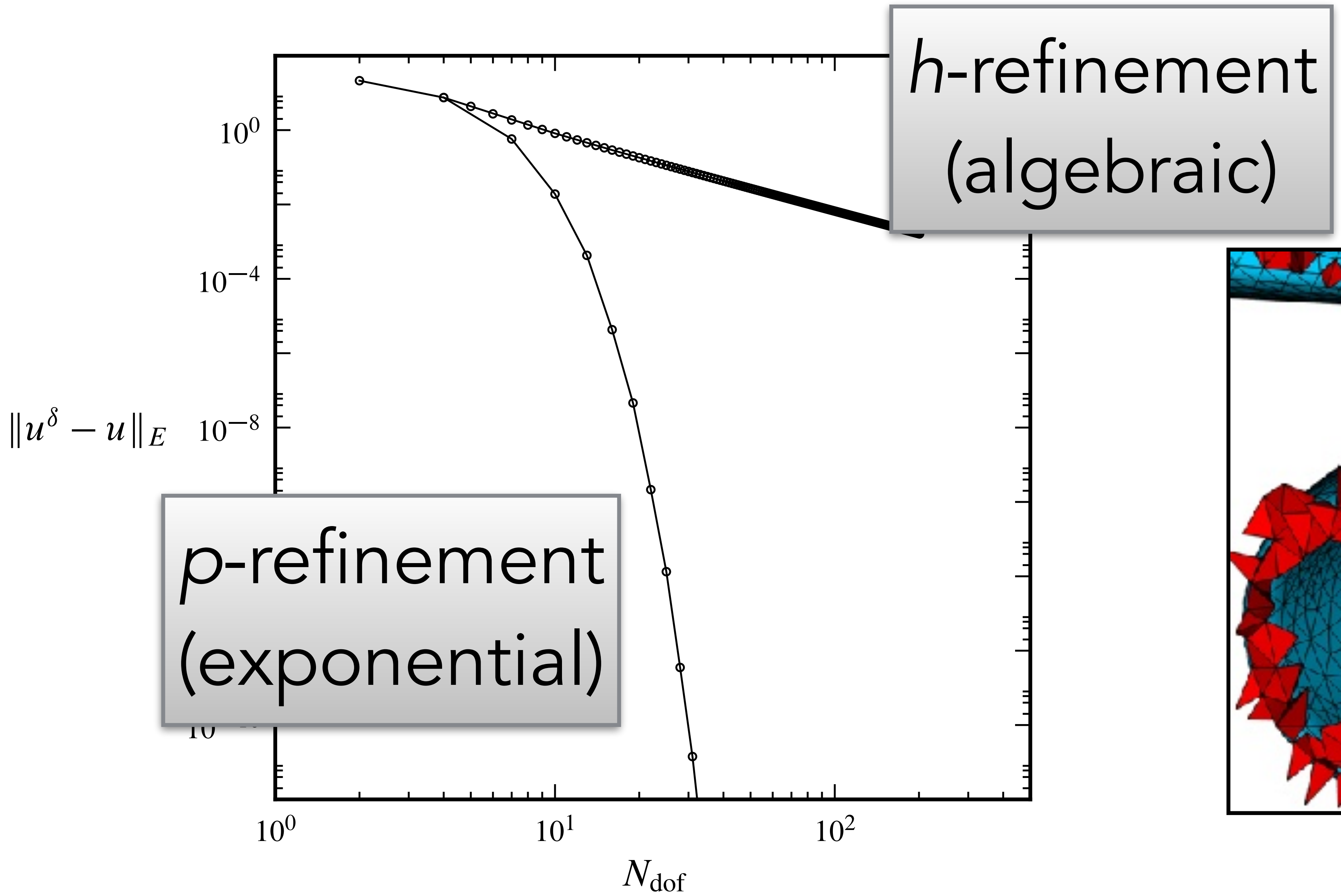
✓ error decays *exponentially* (smooth solutions);

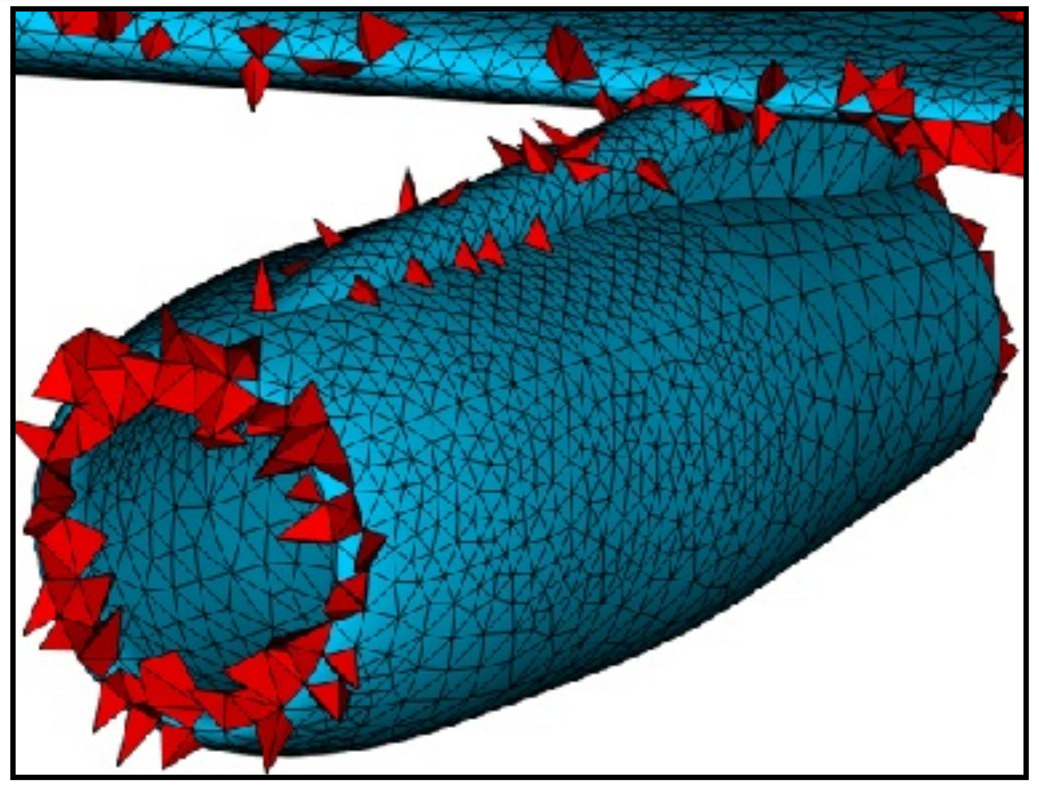
✓ favorable diffusion & dispersion characteristics;

✓ model complex domains

✓ computational advantage: reduced memory bandwidth, better use of hardware.

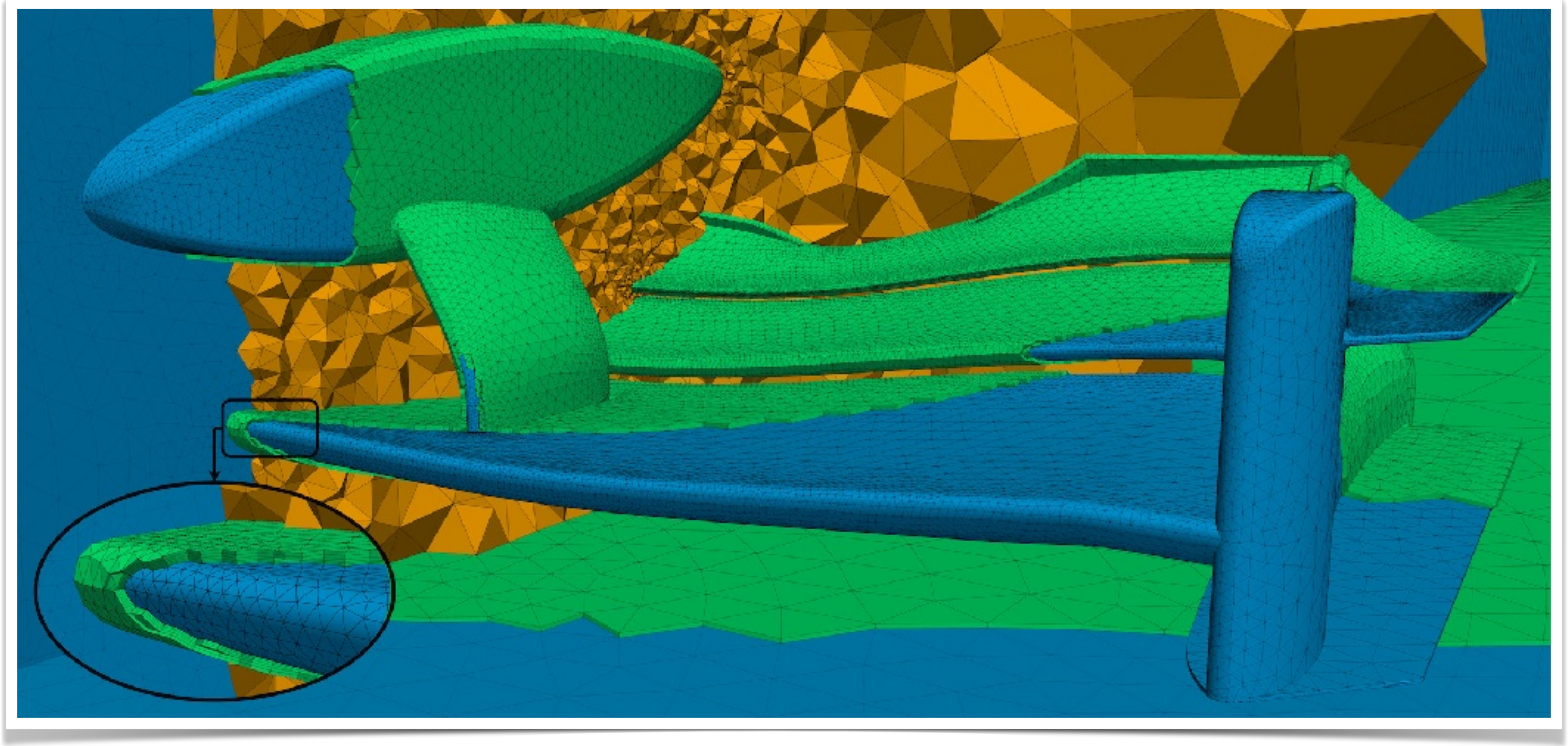



# The challenge: meshing

- Most efficient high-order elements are hexahedra.
- However unstructured hex-only (or even hex-dominant) meshing is still an open problem.
- Therefore need to consider high-order non-tensorial elements: tetrahedra, prisms, pyramids.
- **How do we make FEM operations on these elements efficient?**



Mesh for IFW geometry, $P = 5$

# Nektar++

*spectral/hp element framework*

- Nektar++ is an **open source framework** for the spectral/*hp* element method.
- Want to use these methods in **many areas**, not just fluids; designed with **complex geometries** in mind, supports hybrid 2D/3D meshes.
- Designed to support a range of discretisations (CG, DG) at scales from desktop to HPC.
- Solvers for incompressible/compressible Navier-Stokes & others, with a wide range of features for fluids-based problems (variable *p*, non-conformal meshes for DG, ...)
- Started in 2004: has been CPU-only since the outset, but now significant project underway to port to the GPU.


Mohsen Lahooti (NU)


David Moxey (KCL)


Spencer Sherwin (ICL)


Chris Cantwell (ICL)


Mike Kirby (UoU)


Jacques Xing (ICL)


Henrik Wustenberg (ICL)

# Nektar++

*spectral/hp element framework*


Kaloyan Kirilov (KCL)


Ted Stokes (KCL)


Boyang Xia (KCL)


Edward Erasmie-Jones (KCL)


Chi-Hin Chan (ICL)


Alexandra Liosi (ICL)


Jaou Isler (ICL)

# High-order splitting scheme

Navier–Stokes: $$\partial_t \mathbf{u} + \mathbf{N}(\mathbf{u}) = -\nabla p + \nu \nabla^2 \mathbf{u}$$
$$\nabla \cdot \mathbf{u} = 0$$

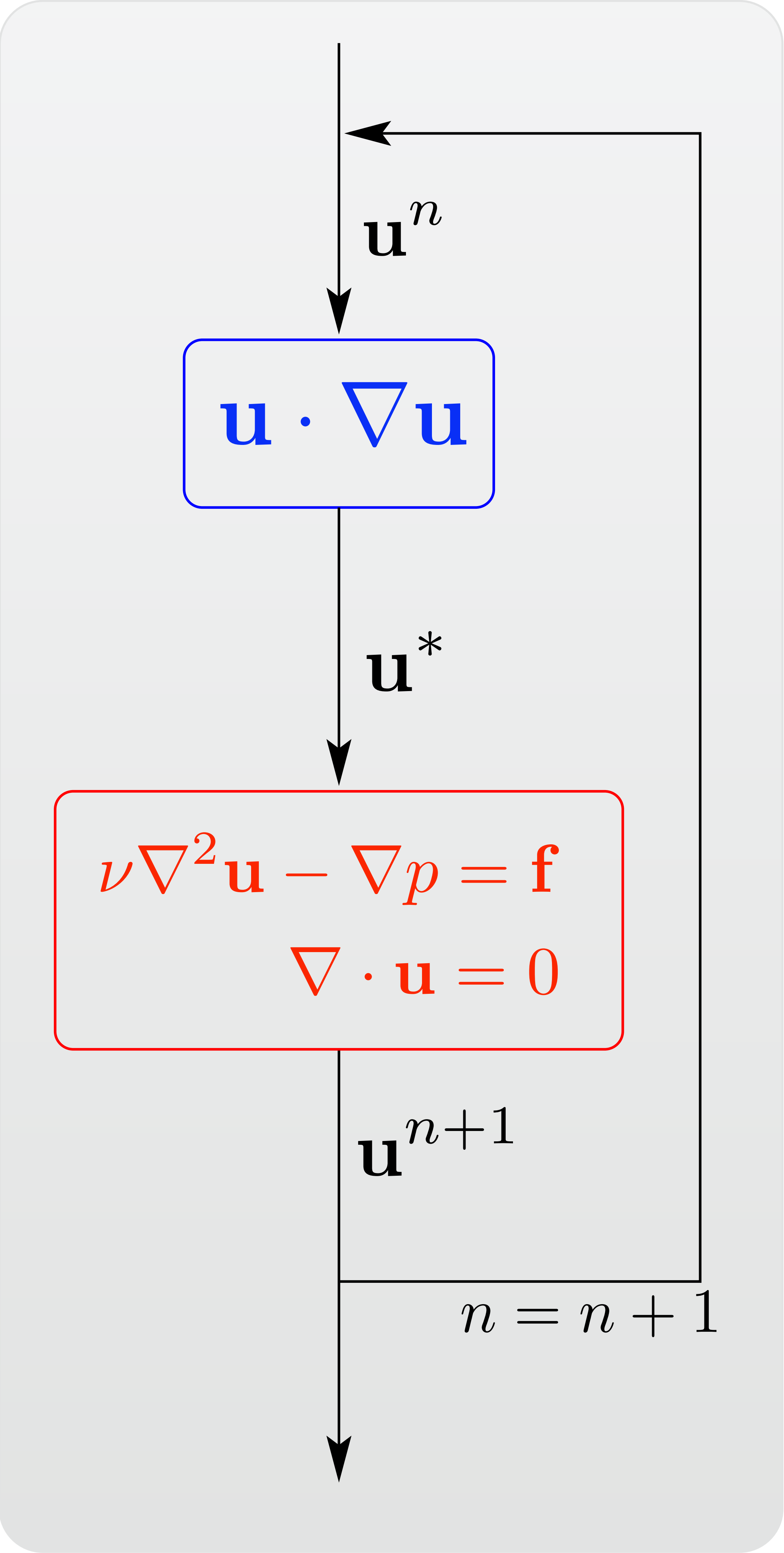
*CG* Velocity correction scheme *(aka stiffly stable):*

*Orszag, Israeli, Deville (90), Karnaidakis Israeli, Orszag (1991), Guermond & Shen (2003)*

Advection: $$u^* = -\sum_{q=1}^{J} \alpha_q \mathbf{u}^{n-q} - \Delta t \sum_{q=0}^{J-1} \beta_q \mathbf{N}(\mathbf{u}^{n-q})$$

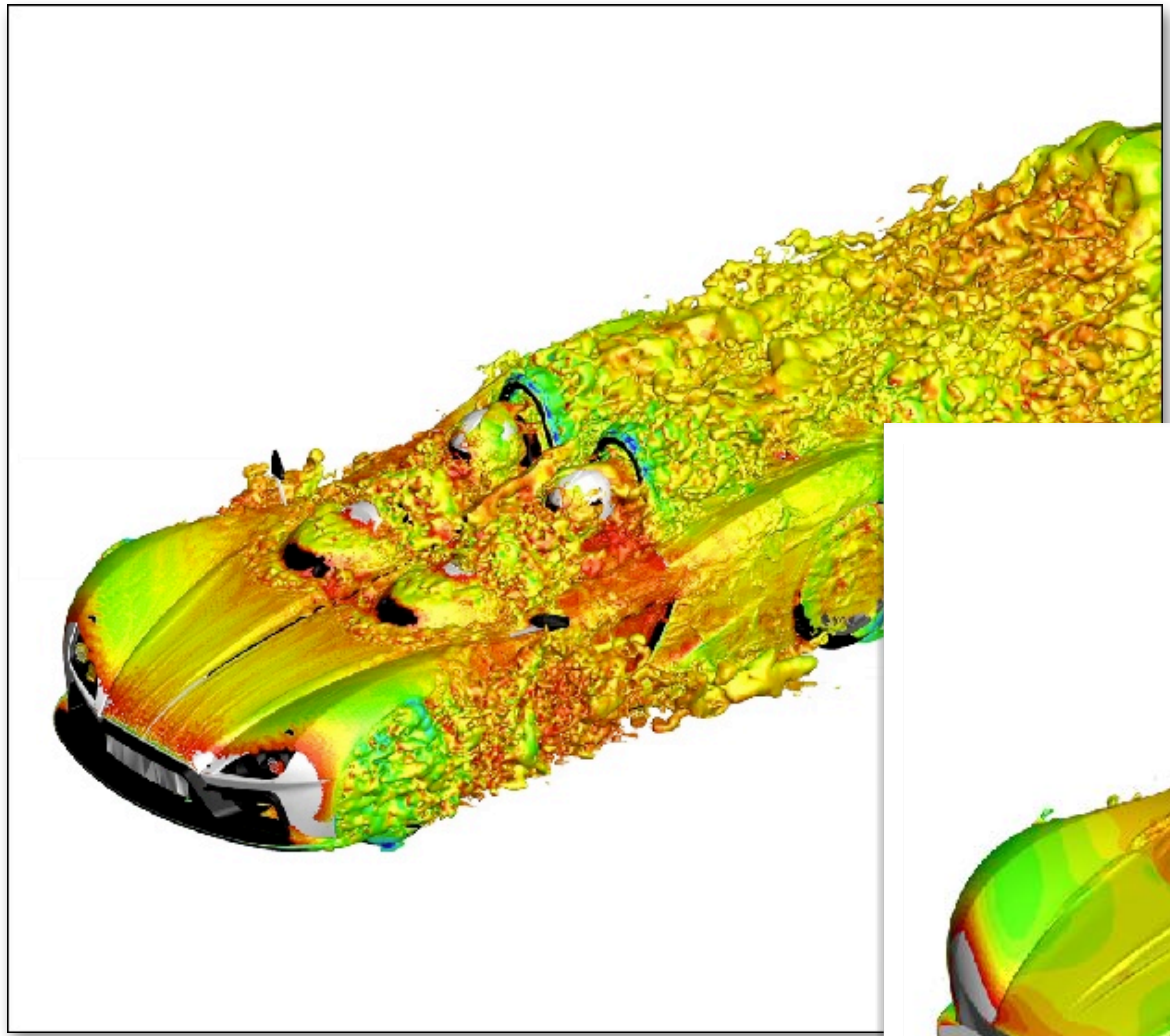
Pressure Poisson: $$\nabla^2 p^{n+1} = \frac{1}{\Delta t} \nabla \cdot \mathbf{u}^*$$

Helmholtz: $$\nabla^2 \mathbf{u}^{n+1} - \frac{\alpha_0}{\nu \Delta t} \mathbf{u}^{n+1} = -\frac{\mathbf{u}^*}{\nu \Delta t} + \frac{1}{\nu} \nabla p^{n+1}$$

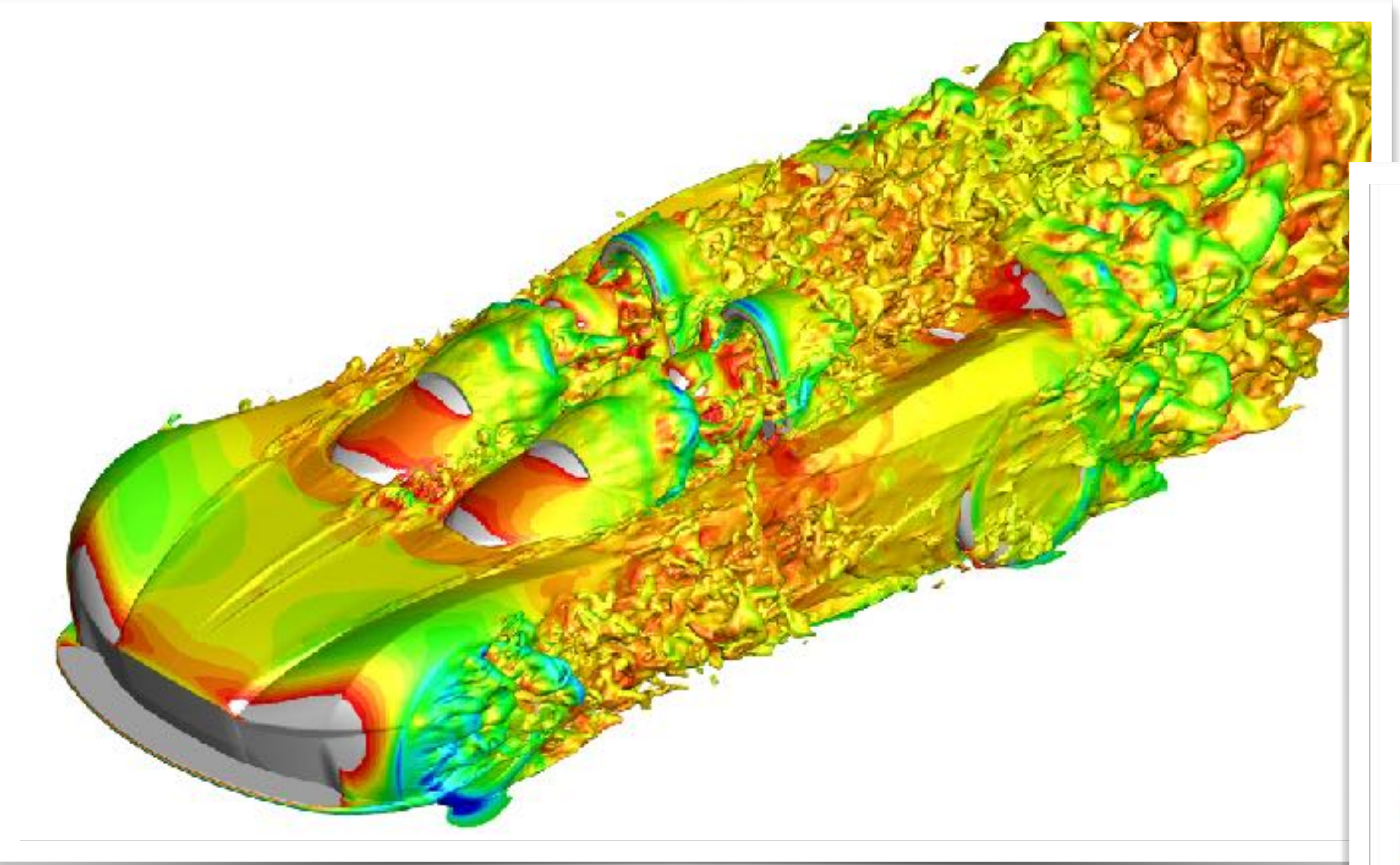



Majority of computational time in linear solves: need **fast matrix-vector operator**, good preconditioners

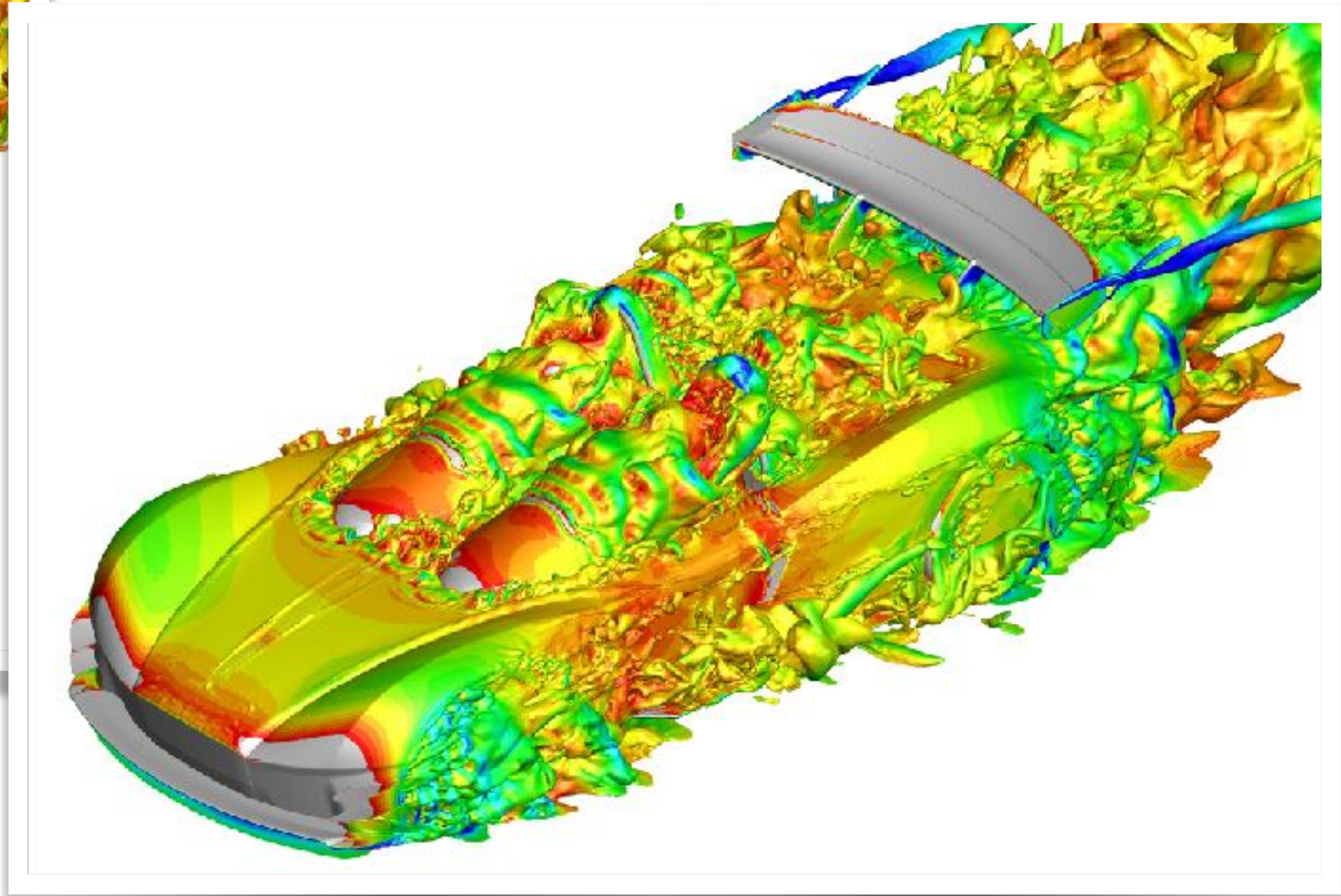
# Virtual wind tunnel



1bn degrees of freedom
Uses only CFD for design

*Design 2: +33% Downforce*



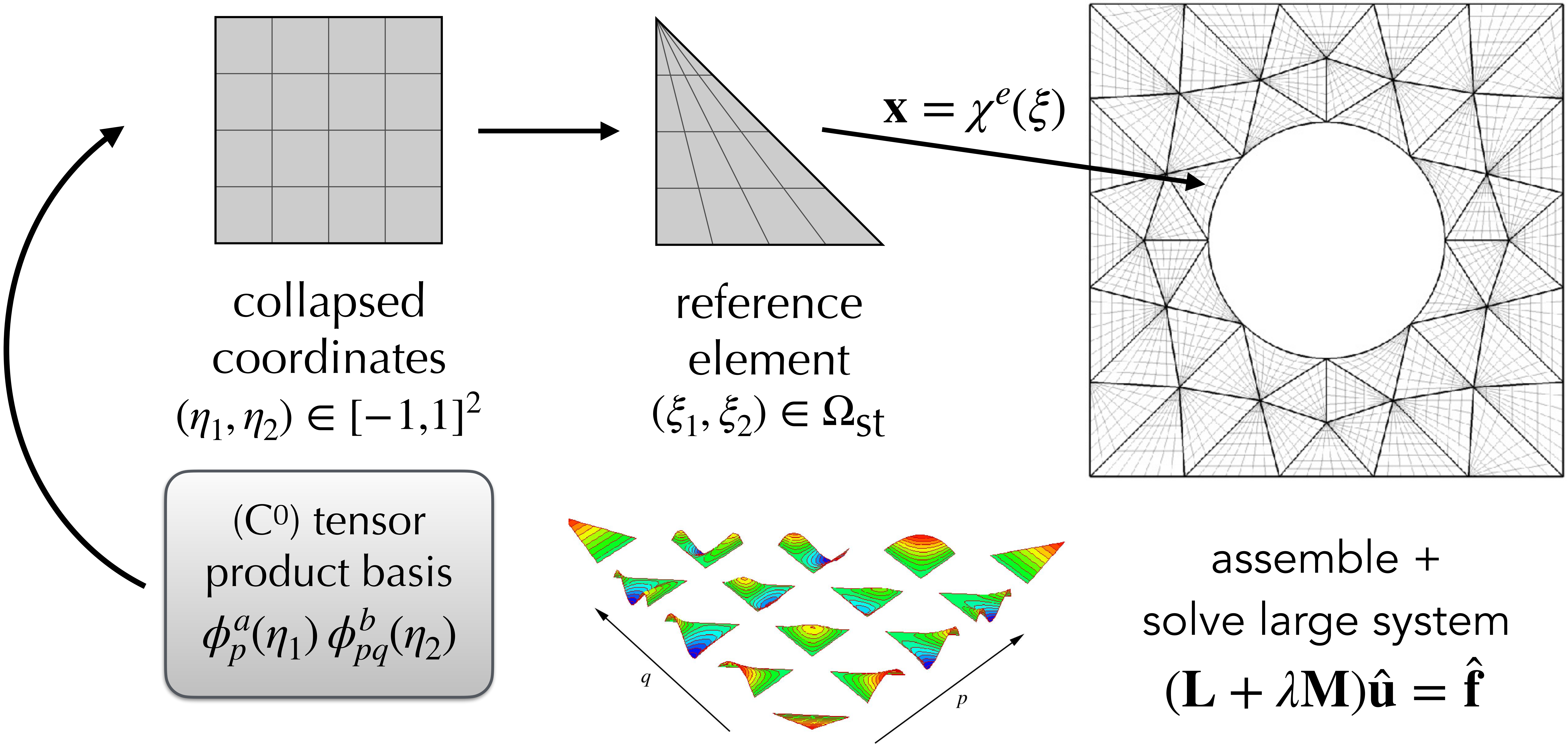
*Design 3: +270% Downforce*

*Mengaldo, Moxey, Turner, Jassim, Taylor, Peiro & Sherwin, SIAM Review (2021)*

# Performant kernels for high-order FEM

- Modern hardware: lots of FLOPS, bottlenecked on memory bandwidth.
- Need codes and algorithms that have **high arithmetic intensity** and exploit **SIMD** parallelism of the hardware.
- **Matrix-free** methods and **sum factorisation/ tensor contractions** help achieve this at high order: widely used by MFEM, deal.ii, ...
- Examine whether this can be applied to more general element types, not just quads/hexes.


# "Defining" features of spectral/*hp* method



# "Defining" features of spectral/*hp* method

Generally *not* collocated

order can vary

$$u(\xi_{1i}, \xi_{2j}) = \sum_{n=0}^{P^2} \hat{u}_n \phi_n(\xi) = \sum_{p=0}^{P} \sum_{q=0}^{Q} \hat{u}_{pq} \phi_p(\xi_{1i}) \phi_q(\xi_{2j})$$

quadrature points

modal coefficients

Uses tensor products of 1D basis functions, even for non-tensor product shapes, e.g. tetrahedron:

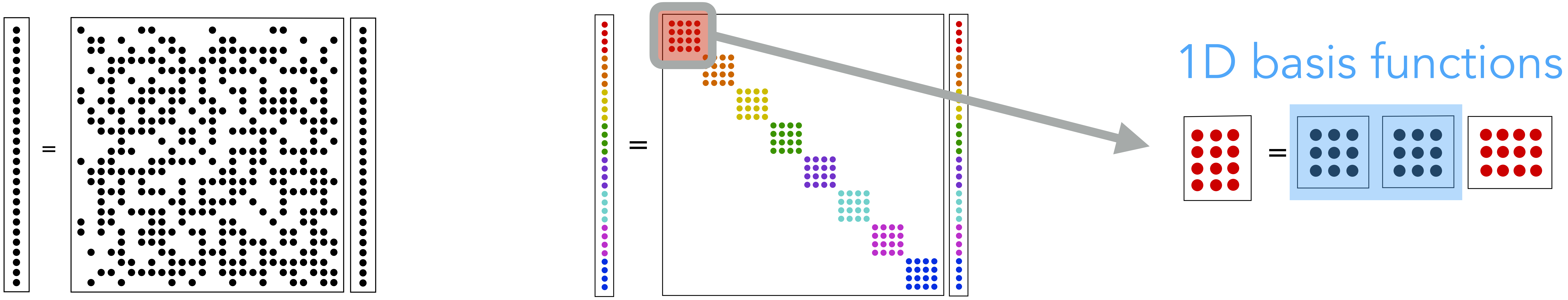
$$u(\xi_{1i}, \xi_{2j}, \xi_{3j}) = \sum_{p=0}^{P} \sum_{q=0}^{Q-p} \sum_{r=0}^{R-p-q} \hat{u}_{pqr} \phi^a_p(\xi_{1i}) \phi^b_{pq}(\xi_{2j}) \phi^c_{pqr}(\xi_{3k})$$

non-uniform quadrature

basis function indexing harder

# Implementation choices

Finite element operation evaluations (e.g. mass matrix) form bulk of simulation cost; however can be evaluated in several ways.




**Global matrix**

assemble a sparse matrix

**Local evaluation**

create elemental dense matrices + assembly map

**Matrix free**

no local matrices at all
**sum factorisation** speedup

increasing arithmetic intensity

# Matrix-free sum-factorisation

- Key to performance at high $P$: do not assemble matrix but evaluate its action instead: matrix-free approach.

- Can reduce cost from $O(P^{2d})$ to $O(P^{d+1})$ by using **sum-factorisation**:

$$\sum_{p=0}^{P}\sum_{q=0}^{Q}\hat{u}_{pq}\phi_p(\xi_{1i})\phi_q(\xi_{2j}) = \sum_{p=0}^{P}\phi_p(\xi_{1i})\left[\sum_{q=0}^{Q}\hat{u}_{pq}\phi_q(\xi_{2j})\right]$$

store this

- Works in essentially the same way for more complex indexing for e.g. triangles:

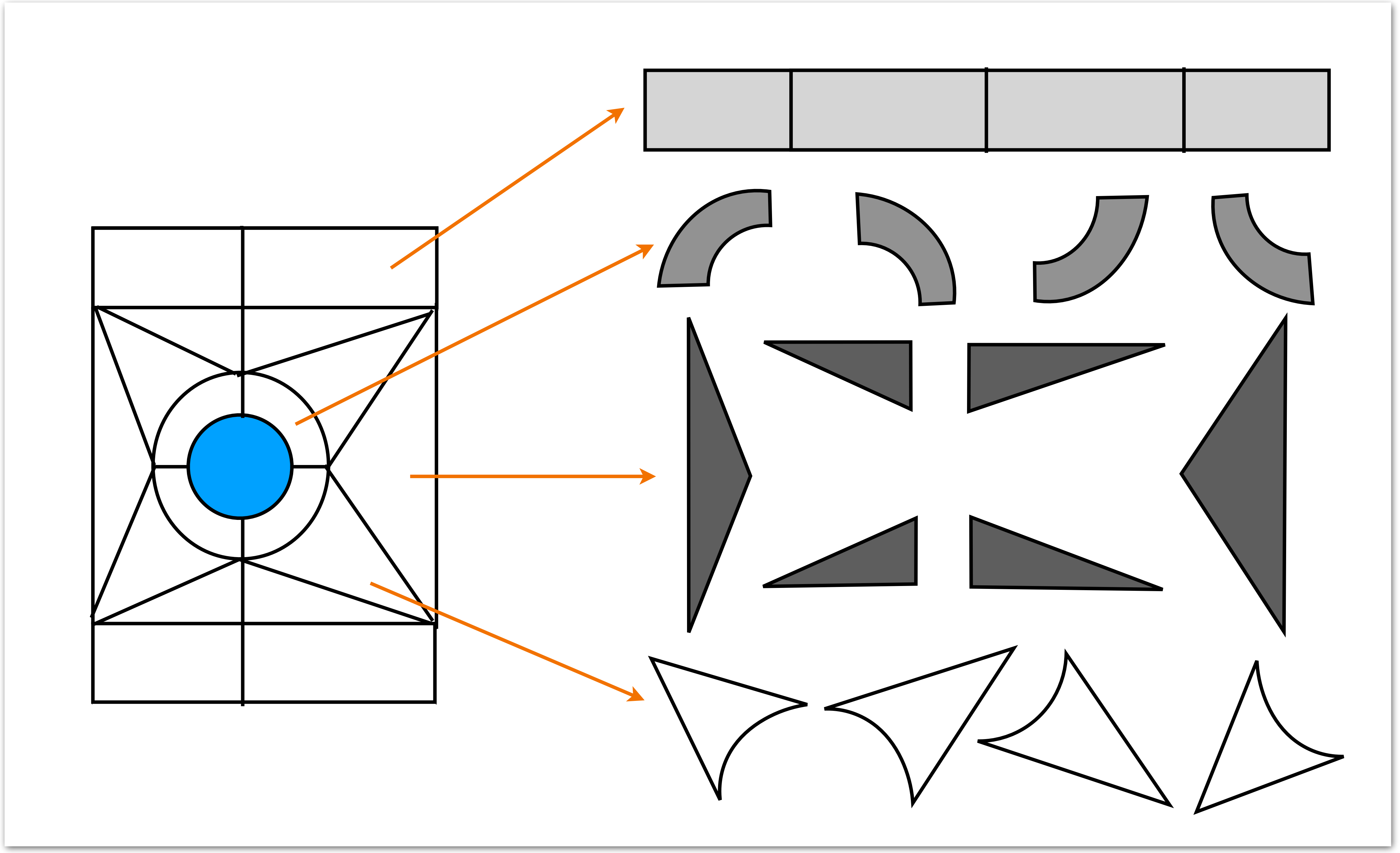
$$\sum_{p=0}^{P}\sum_{q=0}^{Q-p}\hat{u}_{pq}\phi^a_p(\xi_{1i})\phi^b_{pq}(\xi_{2j}) = \sum_{p=0}^{P}\phi^a_p(\xi_{1i})\left[\sum_{q=0}^{Q-p}\hat{u}_{pq}\phi^b_{pq}(\xi_{2j})\right]$$

store this

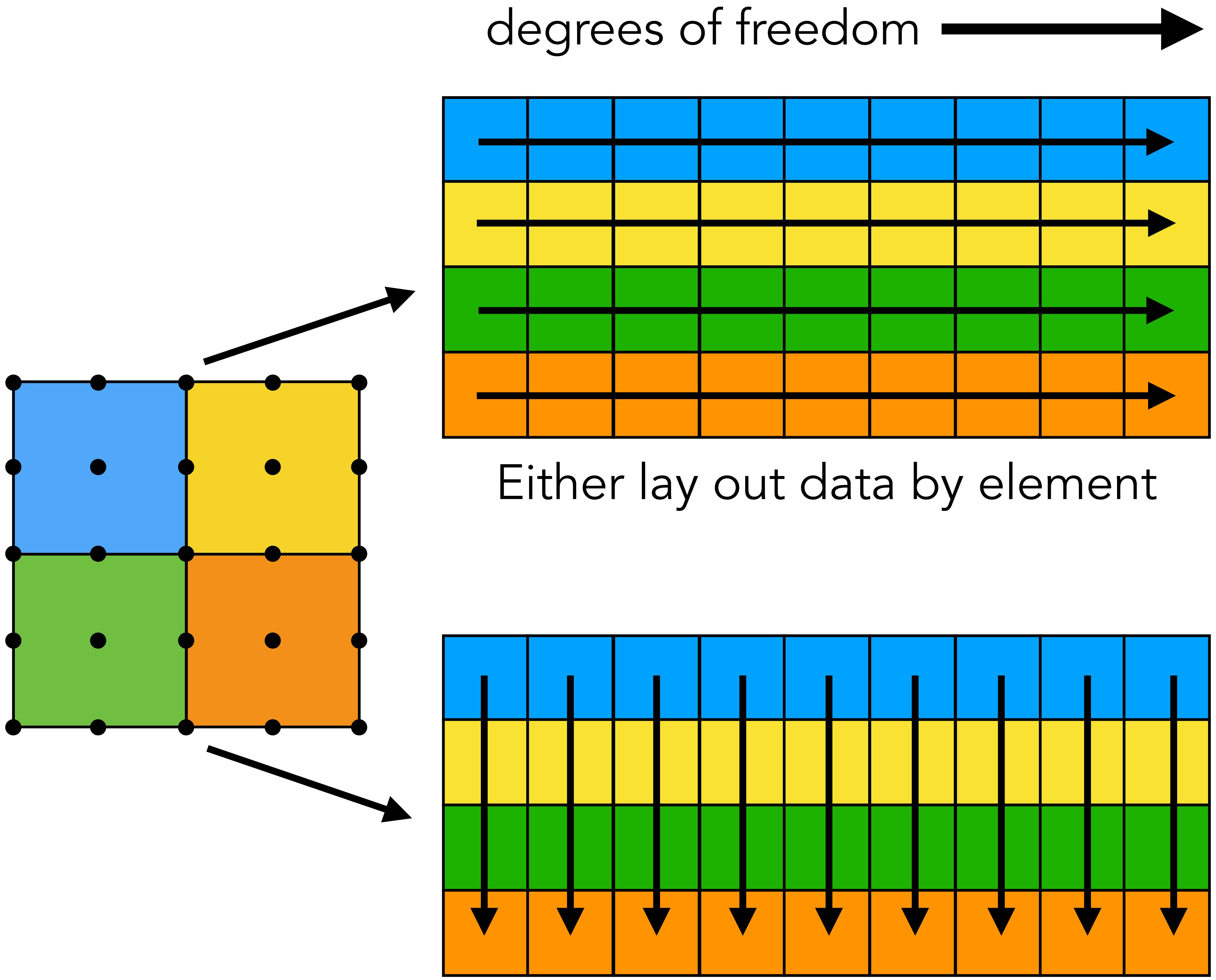
# Standard matrix approach

- Can consider alternative implementation where:
  - compute operator matrices on the standard element;
  - use large (skinny) matrix-matrix multiplication across all elements;
  - incorporate appropriate metrics (Jacobian, derivative factors, etc) with pointwise operators as required.
- Disregards sum-factorisation and potentially >1 matrix multiplication involved, but can exploit optimised small-matrix implementations (e.g. `libxsmm`).
- On GPU, this can be chunked into small groups that are appropriately parallelised over threads.

# Data storage & layout



Group elements by type, polynomial order, integration order, curved/regular

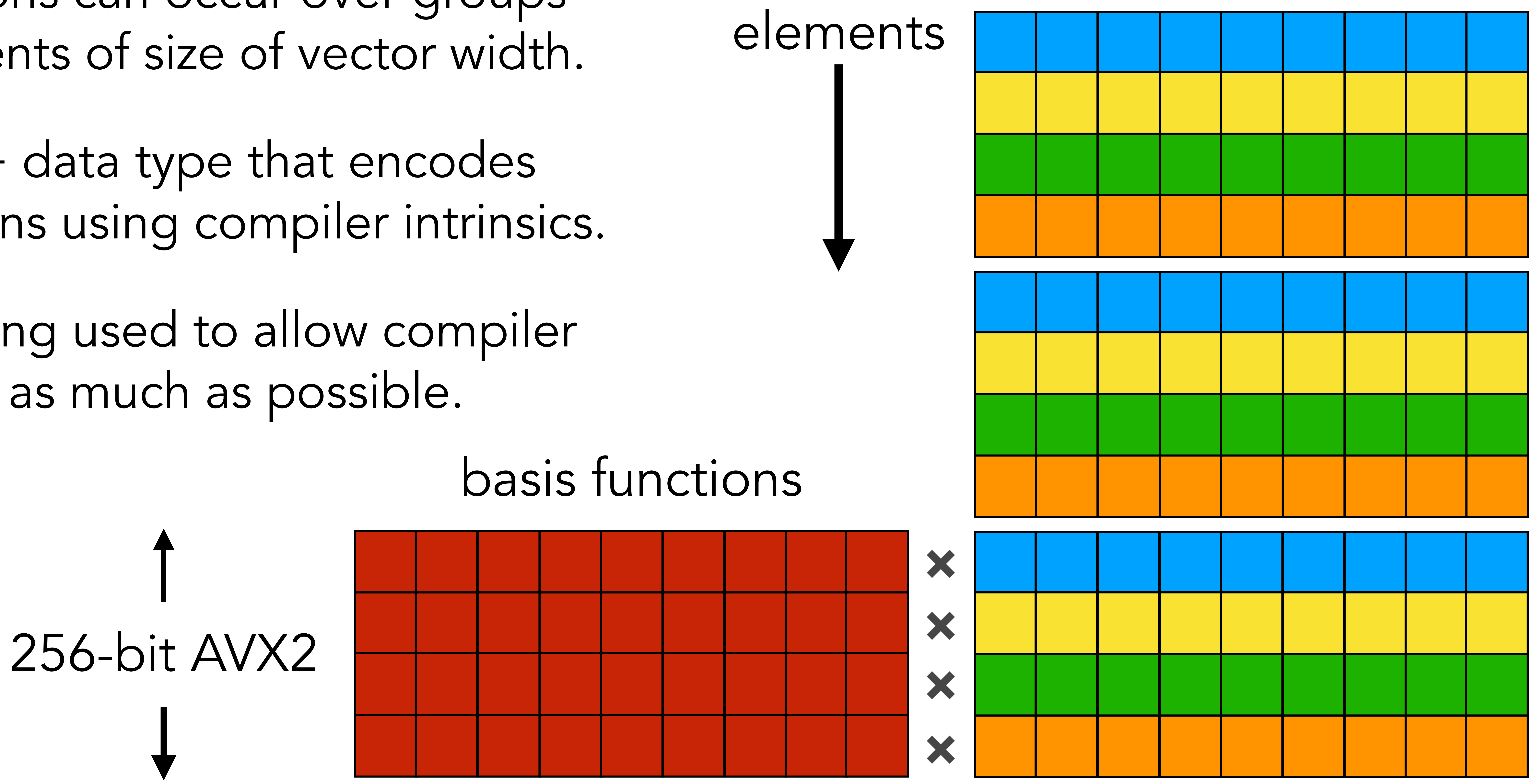



Either lay out data by element

Or interleave to explicitly exploit vectorisation

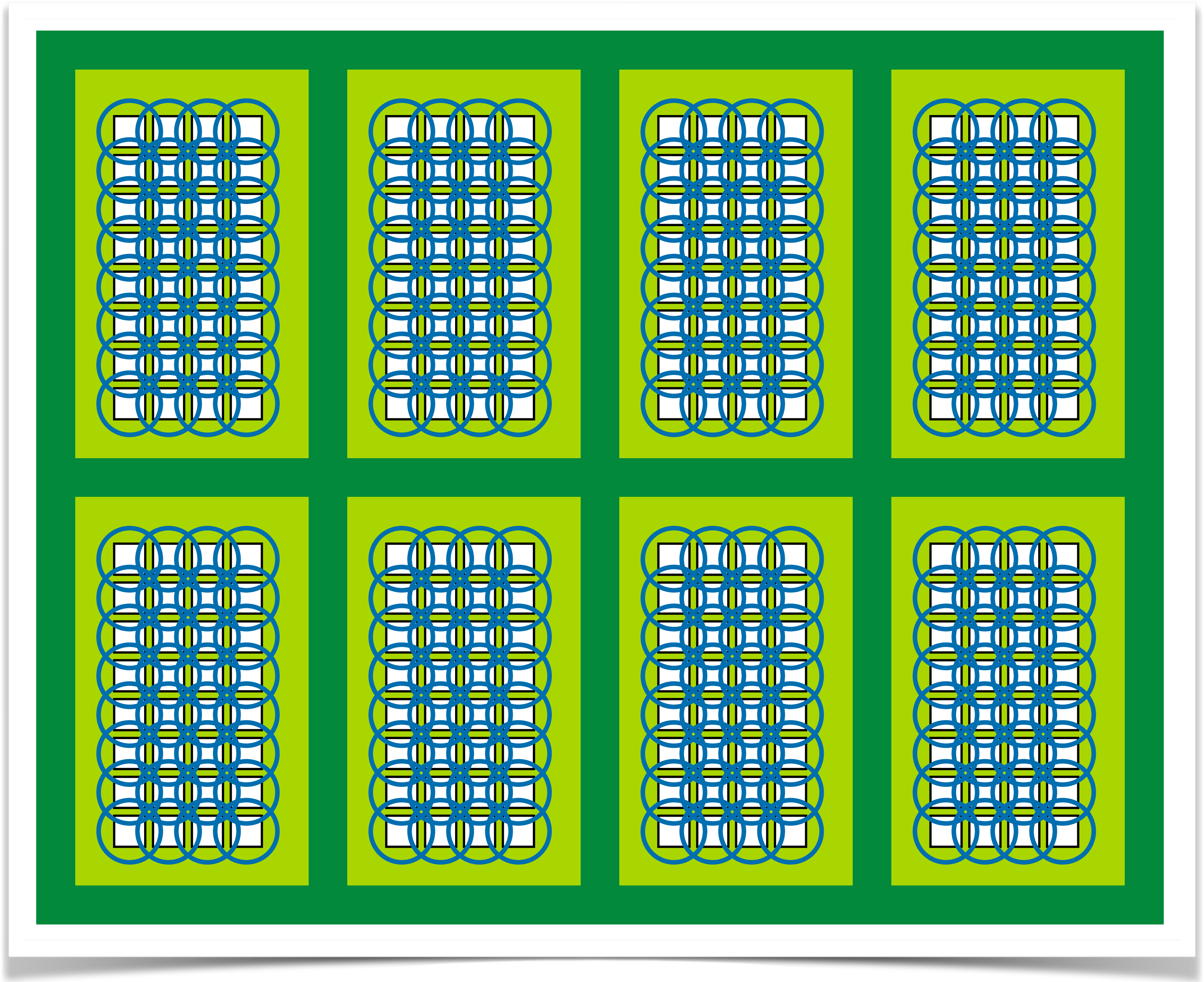
# Element vectorisation

- Operations can occur over groups of elements of size of vector width.
- Use C++ data type that encodes operations using compiler intrinsics.
- Templating used to allow compiler to unroll as much as possible.

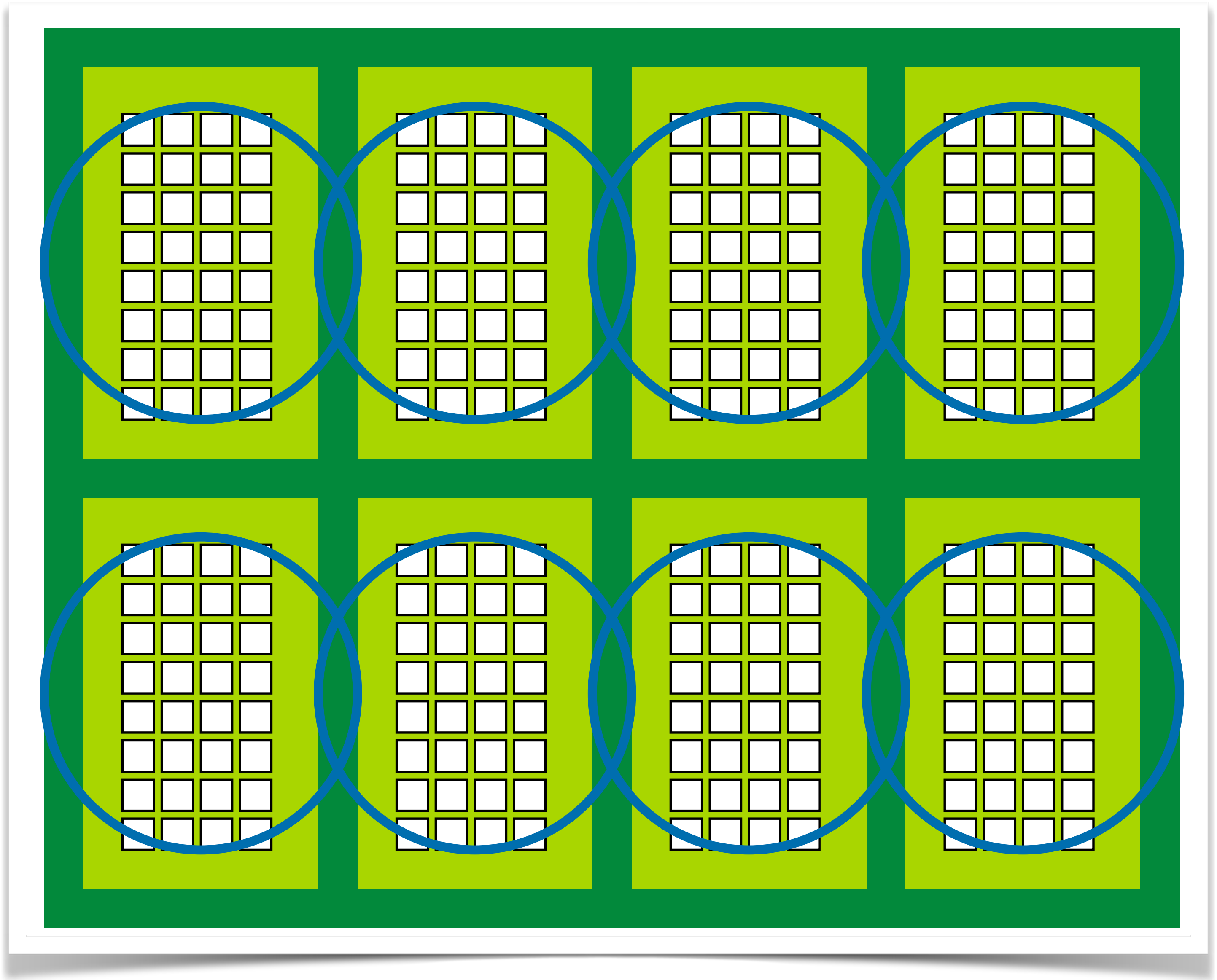


# GPU considerations

- Richer memory and execution hierarchy (warp of 32/64 threads & SM)
- More cores & parallelism, less memory per core and more cache pressure.
- Need to consider different memory storage options & parallelism strategies:
    - **threaded over elements:** each thread owns an entire element & data are interleaved as in CPU implementation.
    - **threaded over points:** element assigned to block, threads given a quadrature point or mode to process.

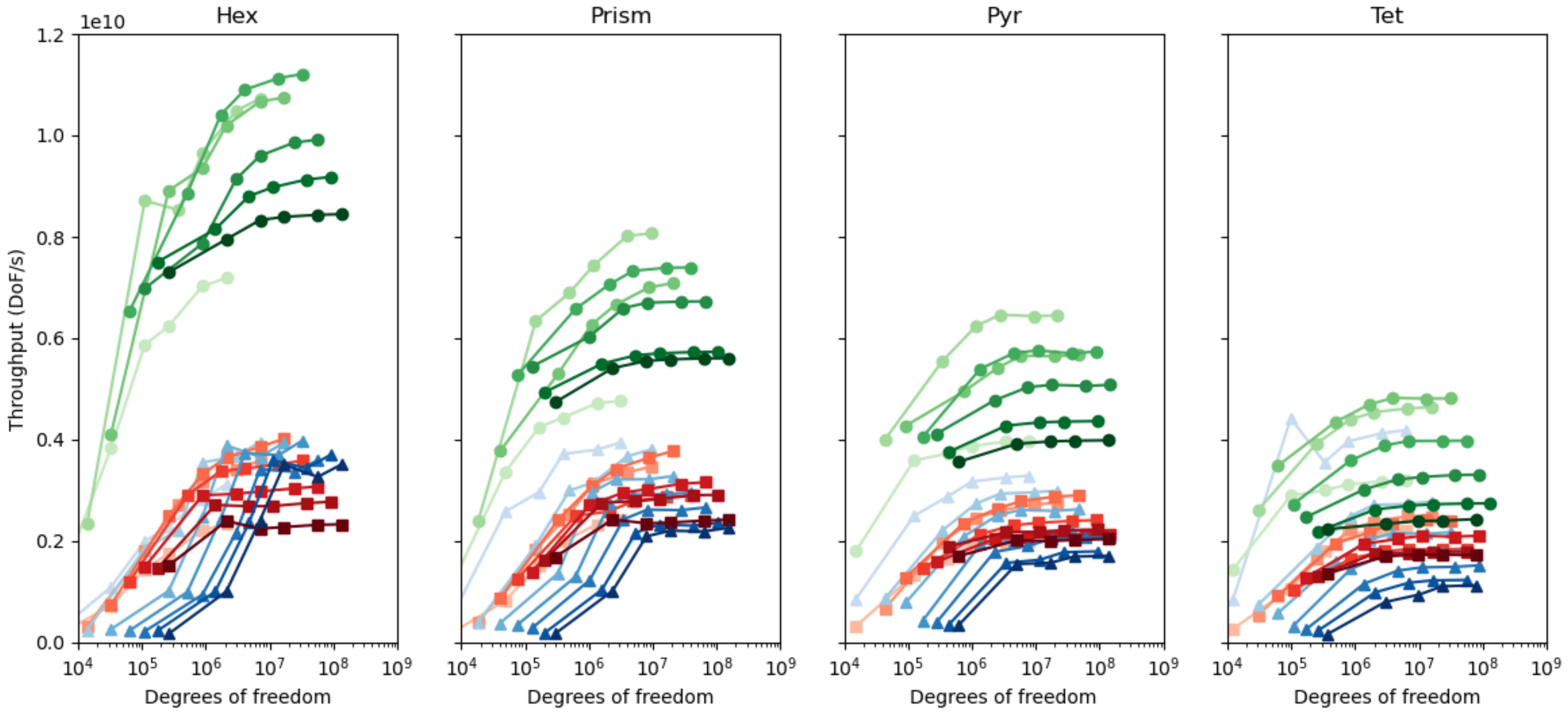


Threaded over **elements** parallelism



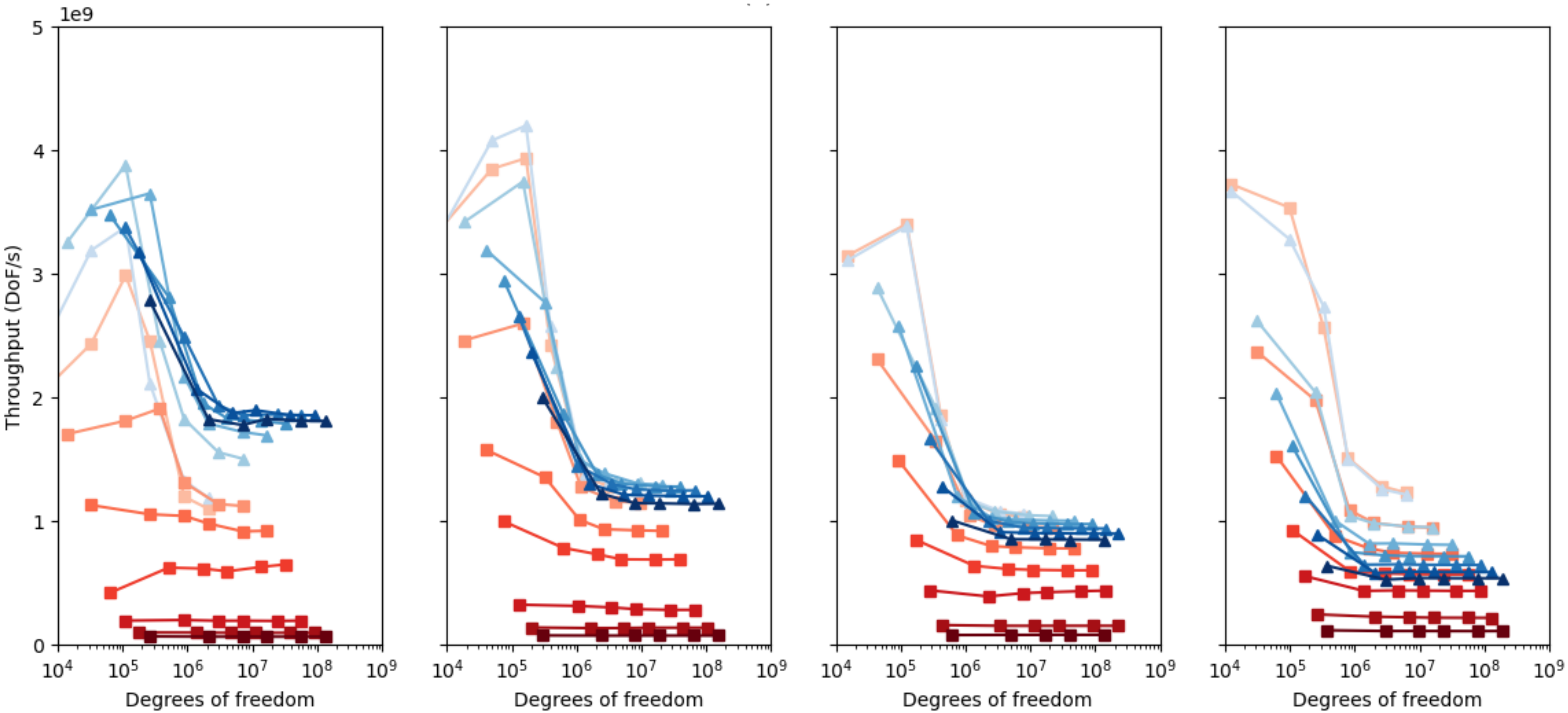
Threaded over **points** parallelism

# Throughput (GPU, NVIDIA GH200)



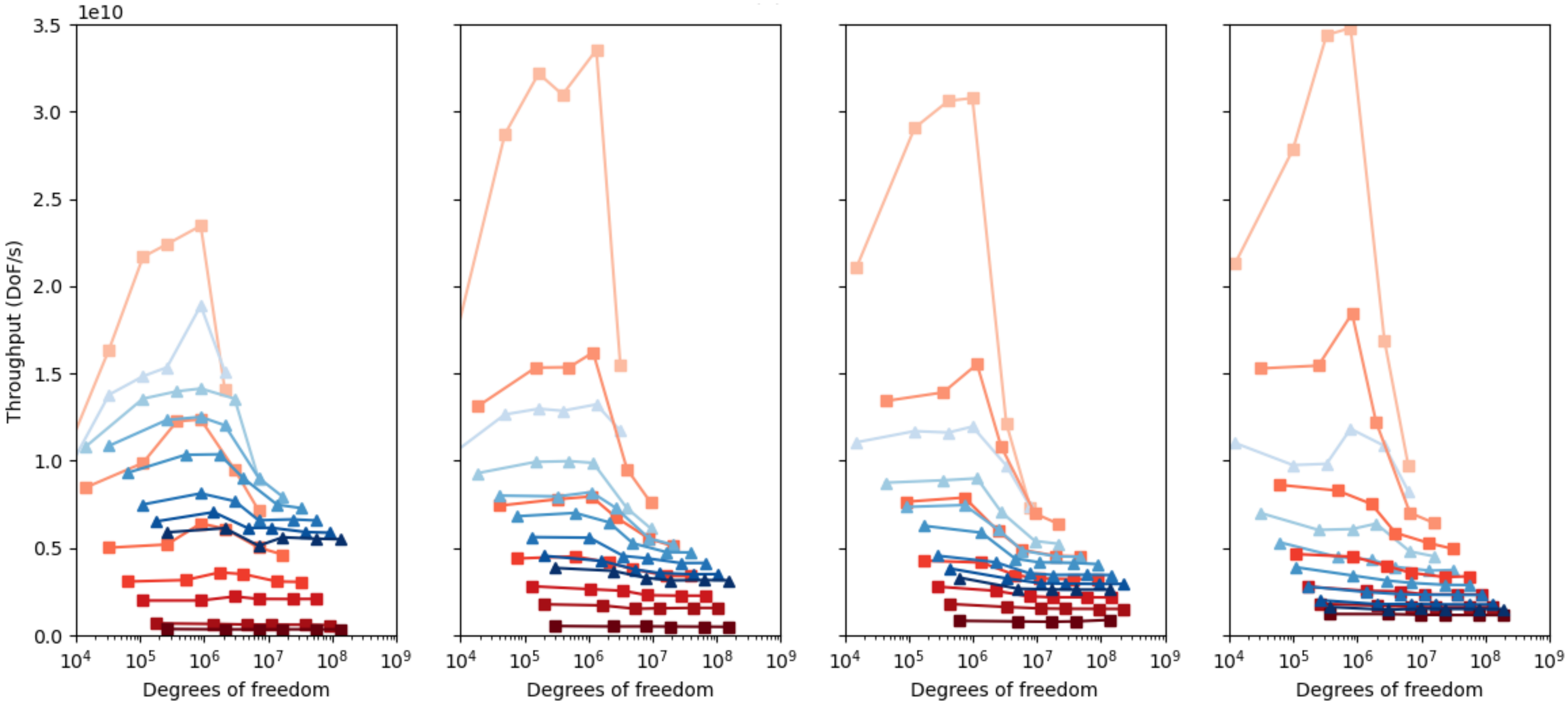

Elemental Helmholtz operator $\boldsymbol{H}^e$

# Throughput (CPU, Intel Xeon 6526Y)




Elemental Helmholtz operator $\boldsymbol{H}^e$

# Throughput (CPU, Intel Xeon 6526Y)



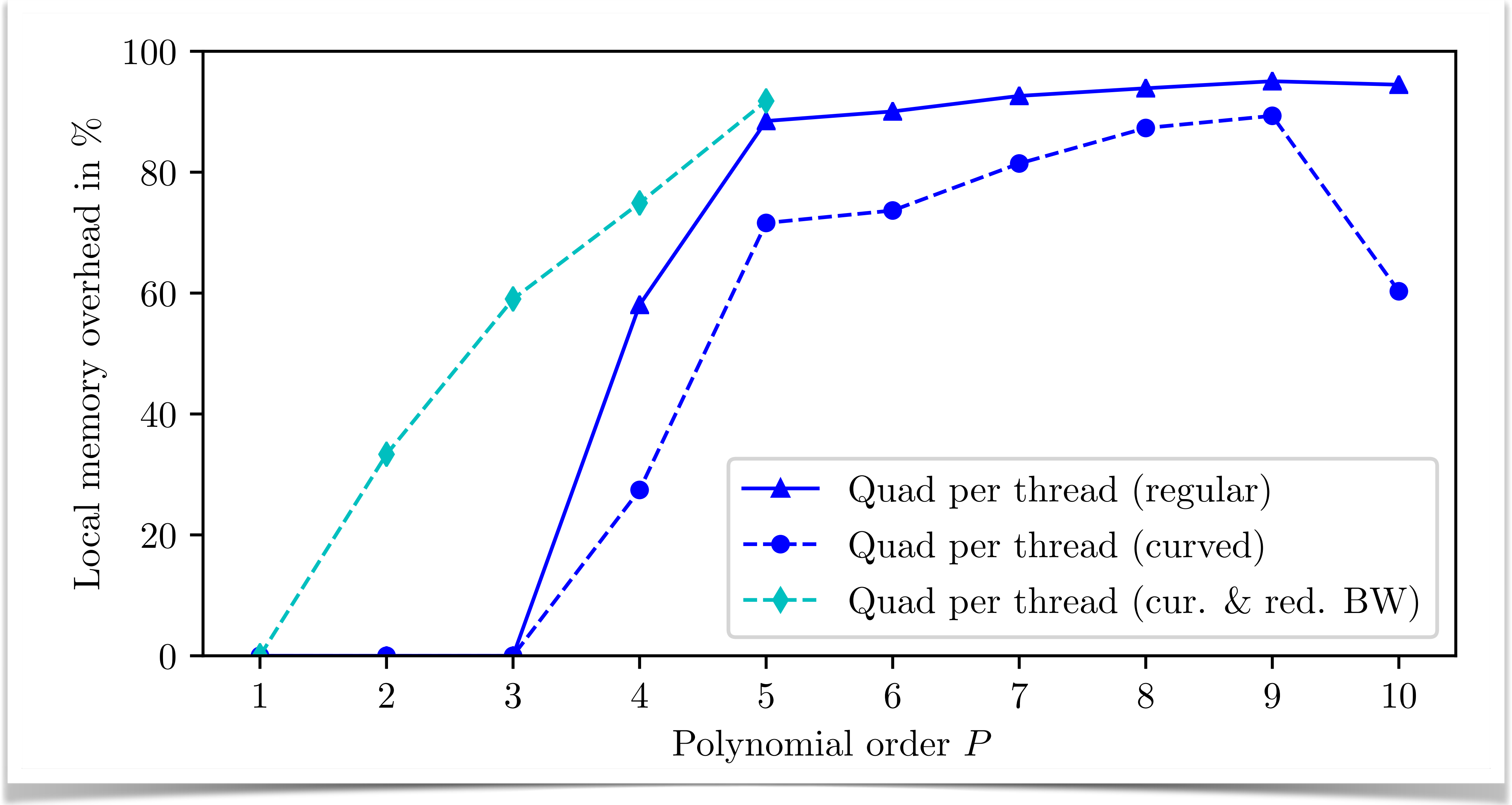

Mass matrix operator $\boldsymbol{M}^e$

# Performance observations

- For more arithmetically intense operators, sum factorisation almost always preferred, but in certain cases matrix-based approaches outperform considerably.

- Crossover point in performance between threading models on the GPU mostly dictated by register pressure and fallback to local memory.

- 3D results suggest almost always better to use entry-by-entry approach.




Quadrilaterals

Eichstädt, Peiró and Moxey, *Comp. Phys. Comm.* (2023)

# Implementation flexibility

- If we are interested in best performance there are a significant number of critical parameters:
    - polynomial order (may also vary in different directions within the element);
    - shape type & implementation approach;
    - whether the element has constant or variable Jacobian;
    - quadrature order (and whether dealiasing is required);
    - basis type (e.g. collocation property for Lagrange basis/nodal elements);
    - underlying data type (e.g. double, single, half) and the hardware.

# Towards NektarIR

- Current approach is to rely on C++ templating, but this leads to parameter explosion in the general case.
- Code generation clearly an alternative, but how to approach this? Certain desirable qualities:
  - capture as much optimisation at different levels as we can: e.g. collocation
  - be able to target different architectures & optimise for them;
  - just-in-time compiled for e.g. adaptive simulations that vary *p*, quadrature order or any of the other parameters listed previously;
  - don't want to build everything from the ground up.
- To this end we are building an intermediate representation (IR) to represent these operators, and capitalise on the existing infrastructure within LLVM.

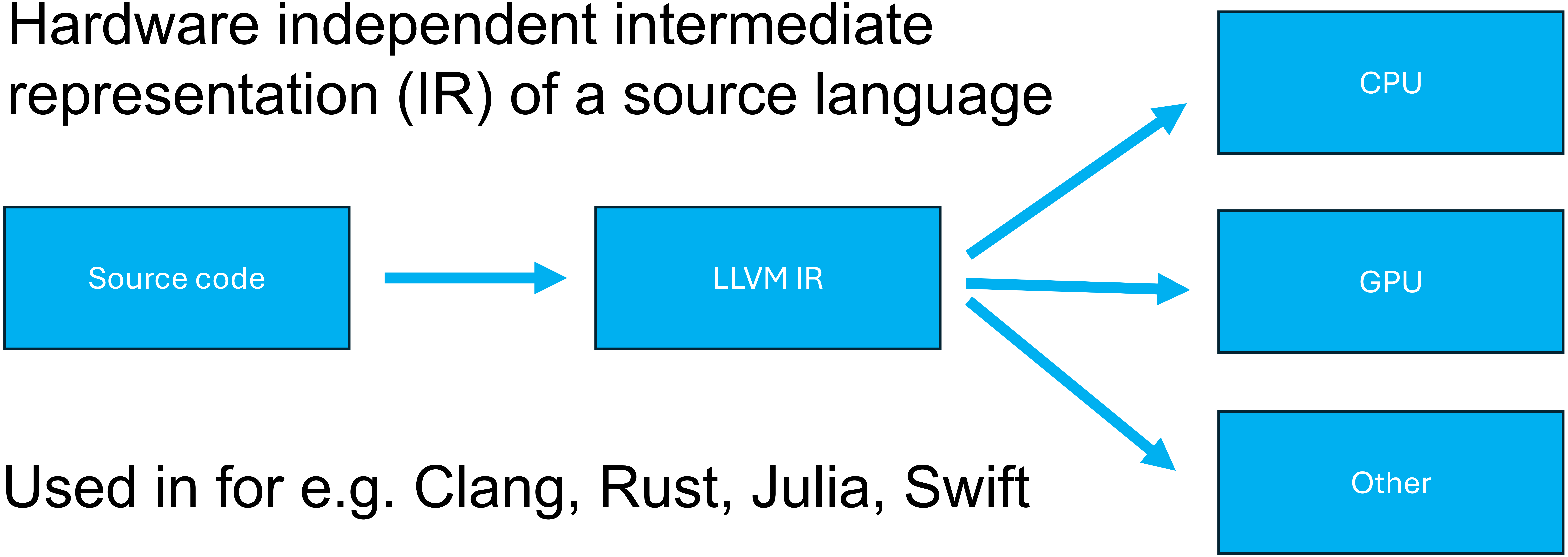
# Outline:

- MLIR and LLVM
- Our Abstraction: An MLIR Dialect for Elemental Operations
- The Compiler Pipeline: Journey from NektarIR to CPU and GPU Kernels
- Performance: Compiler Overhead and Runtime
- The Code

# The LLVM Compiler Infrastructure

https://llvm.org/

- Collection of compiler toolchains
- Hardware independent intermediate representation (IR) of a source language




- Used in for e.g. Clang, Rust, Julia, Swift

# LLVM IR

https://llvm.org/

- Assembly-like language
- Static single-assignment form (SSA)
- Optimizations in the form of passes
- LLVM IR might be too low an abstraction for several applications

# MLIR:

https://mlir.llvm.org/

- *Multi-Level* Intermediate Representation
- Significantly easier to interface with LLVM and build a compiler
- Used extensively in AI and ML applications, e.g Tensorflow and Mojo
- Less popular in scientific computing (for now)

# MLIR:

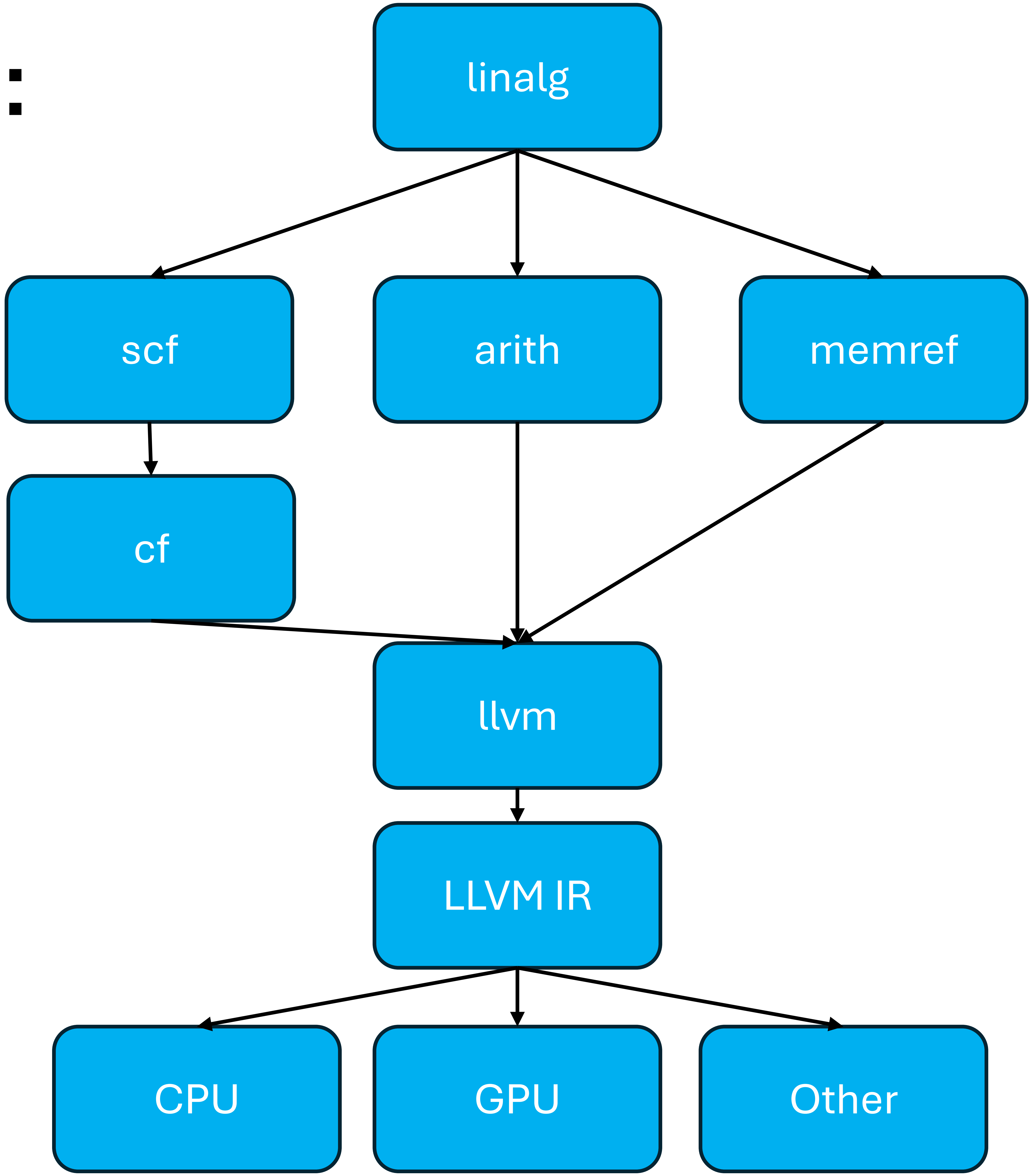
https://mlir.llvm.org/

- Uses **dialects** to represent operations at various abstraction levels
- Dialects for high-level constructs: func, scf, memref
- Architecture independent abstractions: gpu and vector
- Dialect operations, types and attributes are reusable
- IR is rewritten using MLIR passes

# MLIR Pipeline Example:

- A linear algebra operation is converted to loops and arithmetic operations before being lowered further

- A low-level llvm dialect can be translated to LLVM IR before JIT compilation to the desired hardware




https://mlir.llvm.org/

# MLIR: Example rewrite pattern

https://mlir.llvm.org/

```
def f(x):
    a = transpose(x);
    b = transpose(a);
    return b;
```

Redundant transposition that the compiler is not going to catch

transpose(transpose(x)) -> x

```
def f(x):
    return x;
```

IR transformation removes the redundancy

# Our Goals:

- Create an abstraction of common finite element operations as an MLIR dialect
- Facilitate the just-in-time compilation of high-order finite element kernels for use in CFD solvers
- Leverage LLVM to support both CPU and GPU hardware

# The NektarIR Dialect Design:

- Abstraction Level: basic elemental operations acting on blocks of alike elements in a mesh

$$\mathbf{u} = \mathbf{B}\hat{\mathbf{u}}$$

Backward Transform

$$\hat{\mathbf{I}} = \mathbf{B}^T\mathbf{W}\mathbf{u}$$

Inner Product

$$\mathbf{u}_{x_i} = \left[\boldsymbol{\Lambda}\left(\frac{\partial \xi_j}{\partial x_i}\right)\mathbf{D}_{\xi_j}\right]\mathbf{u}$$

Collocation Differentiation

- Each elemental operation produces SSA value result(s)

- No “destination-passing” style or “in-place” operations

# IR Examples

```
!coeffBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Basis: (modified, modified, modified),
    Size:1x104x8x8x8xf64, Layout: (d0,d1, d2) -> (d0,d1,d2)> // coefficient space block type
!physBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Quadrature: (gll, gll, gll),
    Size: 1x104x9x9x9xf64, Layout: (d0,d1,d2) -> (d0, d1 ,d2)> // physical space block type
#map2 = affine_map<(d0,d1) -> (d1,d0)>
module{
    func.func @bwd(%uhat: memref<1x104x512xf64>, %b0: memref<9x8xf64, #map2>,
    %b1: memref<9x8xf64, #map2>, %b2: memref<9x8xf64, #map2>, %u: memref<1x104x729xf64>)
    attributes {llvm.emit_c_interface}
    {
        %b0t = bufferization.to_tensor %b0 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b1t = bufferization.to_tensor %b1 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b2t = bufferization.to_tensor %b2 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>

    %coeffBlock = nir.block_from_memref [     // Associates the buffer containing the coefficient data
        Data: %uhat : memref<1x104x512xf64>   // to the coefficient block type.
        Fields: [u]
        Shape: hex
        Basis: (modified, modified, modified)
        BlockSize: [8,8,8]
        Deformed: false] -> !coeffBlockType

    %out = nir.bwd [                          // The backward transform operation
        Block: %coeffBlock : !coeffBlockType
        Bases: %b0t, %b1t, %b2t: tensor<9x8xf64>, tensor<9x8xf64>, tensor<9x8xf64>] -> !physBlockType
    // Indicates which buffer to store the output of the backward transform to
    nir.materialize_in_destination %out in restrict writable %u: (!physBlockType, memref<1x104x729xf64>) -> ()
    return
}
}
```

```
!coeffBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Basis: (modified, modified, modified),
    Size:1x104x8x8x8xf64, Layout: (d0,d1, d2) -> (d0,d1,d2)> // coefficient space block type
!physBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Quadrature: (gll, gll, gll),
    Size: 1x104x9x9x9xf64, Layout: (d0,d1,d2) -> (d0, d1 ,d2)> // physical space block type
#map2 = affine_map<(d0,d1) -> (d1,d0)>
module{
    func.func @bwd(%uhat: memref<1x104x512xf64>, %b0: memref<9x8xf64, #map2>,
    %b1: memref<9x8xf64, #map2>, %b2: memref<9x8xf64, #map2>, %u: memref<1x104x729xf64>)
    attributes {llvm.emit_c_interface}
    {
        %b0t = bufferization.to_tensor %b0 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b1t = bufferization.to_tensor %b1 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b2t = bufferization.to_tensor %b2 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>

    %coeffBlock = nir.block_from_memref [     // Associates the buffer containing the coefficient data
        Data: %uhat : memref<1x104x512xf64>   // to the coefficient block type.
        Fields: [u]
        Shape: hex
        Basis: (modified, modified, modified)
        BlockSize: [8,8,8]
        Deformed: false] -> !coeffBlockType

    %out = nir.bwd [                          // The backward transform operation
        Block: %coeffBlock : !coeffBlockType
        Bases: %b0t, %b1t, %b2t: tensor<9x8xf64>, tensor<9x8xf64>, tensor<9x8xf64>] -> !physBlockType
    // Indicates which buffer to store the output of the backward transform to
    nir.materialize_in_destination %out in restrict writable %u: (!physBlockType, memref<1x104x729xf64>) -> ()
    return
}
}
```

```
!coeffBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Basis: (modified, modified, modified),
    Size:1x104x8x8x8xf64, Layout: (d0,d1, d2) -> (d0,d1,d2)> // coefficient space block type
!physBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Quadrature: (gll, gll, gll),
    Size: 1x104x9x9x9xf64, Layout: (d0,d1,d2) -> (d0, d1 ,d2)> // physical space block type
#map2 = affine_map<(d0,d1) -> (d1,d0)>
module{
    func.func @bwd(%uhat: memref<1x104x512xf64>, %b0: memref<9x8xf64, #map2>,
    %b1: memref<9x8xf64, #map2>, %b2: memref<9x8xf64, #map2>, %u: memref<1x104x729xf64>)
    attributes {llvm.emit_c_interface}
    {
        %b0t = bufferization.to_tensor %b0 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b1t = bufferization.to_tensor %b1 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b2t = bufferization.to_tensor %b2 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>

    %coeffBlock = nir.block_from_memref [     // Associates the buffer containing the coefficient data
        Data: %uhat : memref<1x104x512xf64>  // to the coefficient block type.
        Fields: [u]
        Shape: hex
        Basis: (modified, modified, modified)
        BlockSize: [8,8,8]
        Deformed: false] -> !coeffBlockType

    %out = nir.bwd [                          // The backward transform operation
        Block: %coeffBlock : !coeffBlockType
        Bases: %b0t, %b1t, %b2t: tensor<9x8xf64>, tensor<9x8xf64>, tensor<9x8xf64>] -> !physBlockType
    // Indicates which buffer to store the output of the backward transform to
    nir.materialize_in_destination %out in restrict writable %u: (!physBlockType, memref<1x104x729xf64>) -> ()
    return
}
}
```

```
!coeffBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Basis: (modified, modified, modified),
    Size:1x104x8x8x8xf64, Layout: (d0,d1, d2) -> (d0,d1,d2)> // coefficient space block type
!physBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Quadrature: (gll, gll, gll),
    Size: 1x104x9x9x9xf64, Layout: (d0,d1,d2) -> (d0, d1 ,d2)> // physical space block type
#map2 = affine_map<(d0,d1) -> (d1,d0)>
module{
    func.func @bwd(%uhat: memref<1x104x512xf64>, %b0: memref<9x8xf64, #map2>,
    %b1: memref<9x8xf64, #map2>, %b2: memref<9x8xf64, #map2>, %u: memref<1x104x729xf64>)
    attributes {llvm.emit_c_interface}
    {
        %b0t = bufferization.to_tensor %b0 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b1t = bufferization.to_tensor %b1 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b2t = bufferization.to_tensor %b2 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>

    %coeffBlock = nir.block_from_memref [    // Associates the buffer containing the coefficient data
        Data: %uhat : memref<1x104x512xf64>  // to the coefficient block type.
        Fields: [u]
        Shape: hex
        Basis: (modified, modified, modified)
        BlockSize: [8,8,8]
        Deformed: false] -> !coeffBlockType

    %out = nir.bwd [                         // The backward transform operation
        Block: %coeffBlock : !coeffBlockType
        Bases: %b0t, %b1t, %b2t: tensor<9x8xf64>, tensor<9x8xf64>, tensor<9x8xf64>] -> !physBlockType
    // Indicates which buffer to store the output of the backward transform to
    nir.materialize_in_destination %out in restrict writable %u: (!physBlockType, memref<1x104x729xf64>) -> ()
    return
}
}
```

```
!coeffBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Basis: (modified, modified, modified),
    Size:1x104x8x8x8xf64, Layout: (d0,d1, d2) -> (d0,d1,d2)> // coefficient space block type
!physBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Quadrature: (gll, gll, gll),
    Size: 1x104x9x9x9xf64, Layout: (d0,d1,d2) -> (d0, d1 ,d2)> // physical space block type
#map2 = affine_map<(d0,d1) -> (d1,d0)>
module{
    func.func @bwd(%uhat: memref<1x104x512xf64>, %b0: memref<9x8xf64, #map2>,
    %b1: memref<9x8xf64, #map2>, %b2: memref<9x8xf64, #map2>, %u: memref<1x104x729xf64>)
    attributes {llvm.emit_c_interface}
    {
        %b0t = bufferization.to_tensor %b0 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b1t = bufferization.to_tensor %b1 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b2t = bufferization.to_tensor %b2 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>

        %coeffBlock = nir.block_from_memref [     // Associates the buffer containing the coefficient data
            Data: %uhat : memref<1x104x512xf64>   // to the coefficient block type.
            Fields: [u]
            Shape: hex
            Basis: (modified, modified, modified)
            BlockSize: [8,8,8]
            Deformed: false] -> !coeffBlockType

        %out = nir.bwd [                          // The backward transform operation
            Block: %coeffBlock : !coeffBlockType
            Bases: %b0t, %b1t, %b2t: tensor<9x8xf64>, tensor<9x8xf64>, tensor<9x8xf64>] -> !physBlockType
        // Indicates which buffer to store the output of the backward transform to
        nir.materialize_in_destination %out in restrict writable %u: (!physBlockType, memref<1x104x729xf64>) -> ()
        return
    }
}
```

```
!coeffBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Basis: (modified, modified, modified),
    Size:1x104x8x8x8xf64, Layout: (d0,d1, d2) -> (d0,d1,d2)> // coefficient space block type
!physBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Quadrature: (gll, gll, gll),
    Size: 1x104x9x9x9xf64, Layout: (d0,d1,d2) -> (d0, d1 ,d2)> // physical space block type
#map2 = affine_map<(d0,d1) -> (d1,d0)>
module{
    func.func @bwd(%uhat: memref<1x104x512xf64>, %b0: memref<9x8xf64, #map2>,
    %b1: memref<9x8xf64, #map2>, %b2: memref<9x8xf64, #map2>, %u: memref<1x104x729xf64>)
    attributes {llvm.emit_c_interface}
    {
        %b0t = bufferization.to_tensor %b0 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b1t = bufferization.to_tensor %b1 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b2t = bufferization.to_tensor %b2 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>

    %coeffBlock = nir.block_from_memref [     // Associates the buffer containing the coefficient data
        Data: %uhat : memref<1x104x512xf64>   // to the coefficient block type.
        Fields: [u]
        Shape: hex
        Basis: (modified, modified, modified)
        BlockSize: [8,8,8]
        Deformed: false] -> !coeffBlockType

    %out = nir.bwd [                          // The backward transform operation
        Block: %coeffBlock : !coeffBlockType
        Bases: %b0t, %b1t, %b2t: tensor<9x8xf64>, tensor<9x8xf64>, tensor<9x8xf64>] -> !physBlockType
    // Indicates which buffer to store the output of the backward transform to
    nir.materialize_in_destination %out in restrict writable %u: (!physBlockType, memref<1x104x729xf64>) -> ()
    return
}
}
```

```
!coeffBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Basis: (modified, modified, modified),
    Size:1x104x8x8x8xf64, Layout: (d0,d1, d2) -> (d0,d1,d2)> // coefficient space block type
!physBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Quadrature: (gll, gll, gll),
    Size: 1x104x9x9x9xf64, Layout: (d0,d1,d2) -> (d0, d1 ,d2)> // physical space block type
#map2 = affine_map<(d0,d1) -> (d1,d0)>
module{
    func.func @bwd(%uhat: memref<1x104x512xf64>, %b0: memref<9x8xf64, #map2>,
    %b1: memref<9x8xf64, #map2>, %b2: memref<9x8xf64, #map2>, %u: memref<1x104x729xf64>)
    attributes {llvm.emit_c_interface}
    {
        %b0t = bufferization.to_tensor %b0 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b1t = bufferization.to_tensor %b1 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b2t = bufferization.to_tensor %b2 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>

    %coeffBlock = nir.block_from_memref [     // Associates the buffer containing the coefficient data
        Data: %uhat : memref<1x104x512xf64>   // to the coefficient block type.
        Fields: [u]
        Shape: hex
        Basis: (modified, modified, modified)
        BlockSize: [8,8,8]
        Deformed: false] -> !coeffBlockType

    %out = nir.bwd [                          // The backward transform operation
        Block: %coeffBlock : !coeffBlockType
        Bases: %b0t, %b1t, %b2t: tensor<9x8xf64>, tensor<9x8xf64>, tensor<9x8xf64>] -> !physBlockType
    // Indicates which buffer to store the output of the backward transform to
    nir.materialize_in_destination %out in restrict writable %u: (!physBlockType, memref<1x104x729xf64>) -> ()
    return
}
}
```

```
!coeffBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Basis: (modified, modified, modified),
    Size:1x104x8x8x8xf64, Layout: (d0,d1, d2) -> (d0,d1,d2)> // coefficient space block type
!physBlockType = !nir.block<Fields: [u], SEShape: hex, Deformed: false, Quadrature: (gll, gll, gll),
    Size: 1x104x9x9x9xf64, Layout: (d0,d1,d2) -> (d0, d1 ,d2)> // physical space block type
#map2 = affine_map<(d0,d1) -> (d1,d0)>
module{
    func.func @bwd(%uhat: memref<1x104x512xf64>, %b0: memref<9x8xf64, #map2>,
    %b1: memref<9x8xf64, #map2>, %b2: memref<9x8xf64, #map2>, %u: memref<1x104x729xf64>)
    attributes {llvm.emit_c_interface}
    {
        %b0t = bufferization.to_tensor %b0 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b1t = bufferization.to_tensor %b1 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>
        %b2t = bufferization.to_tensor %b2 restrict : memref<9x8xf64, #map2> to tensor<9x8xf64>

    %coeffBlock = nir.block_from_memref [     // Associates the buffer containing the coefficient data
        Data: %uhat : memref<1x104x512xf64>  // to the coefficient block type.
        Fields: [u]
        Shape: hex
        Basis: (modified, modified, modified)
        BlockSize: [8,8,8]
        Deformed: false] -> !coeffBlockType

    %out = nir.bwd [                        // The backward transform operation
        Block: %coeffBlock : !coeffBlockType
        Bases: %b0t, %b1t, %b2t: tensor<9x8xf64>, tensor<9x8xf64>, tensor<9x8xf64>] -> !physBlockType
    // Indicates which buffer to store the output of the backward transform to
    nir.materialize_in_destination %out in restrict writable %u: (!physBlockType, memref<1x104x729xf64>) -> ()
    return
}
}
```

# Helmholtz:

```
%helm = nir.helmholtz [
            Block: %coeffBlock : !coeffBlockType
            Bases: %b0t, %b1t, %b2t: tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>
            DMats: %d0t, %d1t, %d2t : tensor<4x4xf64>, tensor<5x5xf64>, tensor<6x6xf64>
            Jac: %jacobianBlock : !jacobianBlockType
            Weights: %w0t, %w1t, %w2t: tensor<4xf64>, tensor<5xf64>, tensor<6xf64>
            DiffCoeffs: %diffCoeffT : tensor<6xf64>
            Factors: %dft : tensor<9xf64>
            Scale: 2.0] -> !coeffBlockType
```

- Operations acting on blocks of elements are lowered to a loop over elements and a series of operations that act on a single element (or a vector-width number of  elements)

```
%13 = nir.empty_block() : !coeffBlockType
   %c0 = arith.constant 0 : index
   %c1000 = arith.constant 1000 : index
   %c1 = arith.constant 1 : index
   %14 = scf.for %arg14 = %c0 to %c1000 step %c1 iter_args(%arg15 = %13) -> (!coeffBlockType) {

     %15 = nir.extract_slice %11  [0, %arg14, 0, 0, 0] [1, 1, 3, 4, 5] [1, 1, 1, 1, 1] : !coeffBlockType to !singleCoeffType
     %16 = nir.extract_slice %12  [0, %arg14, 0, 0, 0] [1, 1, 1, 1, 1] [1, 1, 1, 1, 1] : !jacobianBlockType to !singleJacobianType

     %17 = nir.elmnt_bwd[ // backward transform
      Block : %15 : !singleCoeffType
      Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>] -> !singlePhysType

     %18:3 = nir.elmnt_standard_deriv[ // derivative in local coordinates
      Block : %17 : !singlePhysType
      DMats : %6, %7, %8 : tensor<4x4xf64>, tensor<5x5xf64>, tensor<6x6xf64>]
      -> !singlePhysType, !singlePhysType, !singlePhysType

     %19:3 = nir.elmnt_deriv_metric[ // apply the derivative metric and diffusion
     Blocks : %18#0, %18#1, %18#2 : !singlePhysType, !singlePhysType, !singlePhysType
      Factors : %9 : tensor<9xf64>
      DiffCoeffs : %10 : tensor<6xf64>
     ] -> !singlePhysType, !singlePhysType, !singlePhysType

     %20:4 = nir.elmnt_apply_jw[ // apply weights and jacobian determinants
     Blocks : %17, %19#0, %19#1, %19#2 : !singlePhysType, !singlePhysType, !singlePhysType, !singlePhysType
      Jac : %16 : !singleJacobianType
      Weights : %3, %4, %5 : tensor<4xf64>, tensor<5xf64>, tensor<6xf64>]
      -> !singlePhysType, !singlePhysType, !singlePhysType, !singlePhysType

     %21 = nir.elmnt_test[ // action of B^T
     Block : %20#0 : !singlePhysType
      Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>
       Scale : 2.0: f64] -> !singleCoeffType

     %22 = nir.elmnt_test_grad[ // "dot product" grad(v) and grad(u) and action of B^T
     Blocks : %20#1, %20#2, %20#3 : !singlePhysType, !singlePhysType, !singlePhysType
      DMats : %6, %7, %8 : tensor<4x4xf64>, tensor<5x5xf64>, tensor<6x6xf64>
      Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>]
      -> !singleCoeffType

     %23 = nir.add[
     Blocks : %21, %22 : !singleCoeffType, !singleCoeffType]
     -> !singleCoeffType

     %24 = nir.insert_slice %23 into %arg15[0, %arg14, 0, 0, 0] [1, 1, 3, 4, 5] [1, 1, 1, 1, 1] : !singleCoeffType into !coeffBlockType

     scf.yield %24 : !coeffBlockType

   } {element_shape = #nir.element_shape<hex>}
```

```
%13 = nir.empty_block() : !coeffBlockType
  %c0 = arith.constant 0 : index
  %c1000 = arith.constant 1000 : index
  %c1 = arith.constant 1 : index
  %14 = scf.for %arg14 = %c0 to %c1000 step %c1 iter_args(%arg15 = %13) -> (!coeffBlockType) {

    %15 = nir.extract_slice %11  [0, %arg14, 0, 0, 0] [1, 1, 3, 4, 5] [1, 1, 1, 1, 1] : !coeffBlockType to !singleCoeffType
    %16 = nir.extract_slice %12  [0, %arg14, 0, 0, 0] [1, 1, 1, 1, 1] [1, 1, 1, 1, 1] : !jacobianBlockType to !singleJacobianType

    %17 = nir.elmnt_bwd[ // backward transform
     Block : %15 : !singleCoeffType
     Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>] -> !singlePhysType

    %18:3 = nir.elmnt_standard_deriv[ // derivative in local coordinates
     Block : %17 : !singlePhysType
     DMats : %6, %7, %8 : tensor<4x4xf64>, tensor<5x5xf64>, tensor<6x6xf64>]
     -> !singlePhysType, !singlePhysType, !singlePhysType

    %19:3 = nir.elmnt_deriv_metric[ // apply the derivative metric and diffusion
    Blocks : %18#0, %18#1, %18#2 : !singlePhysType, !singlePhysType, !singlePhysType
     Factors : %9 : tensor<9xf64>
     DiffCoeffs : %10 : tensor<6xf64>
    ] -> !singlePhysType, !singlePhysType, !singlePhysType

    %20:4 = nir.elmnt_apply_jw[ // apply weights and jacobian determinants
    Blocks : %17, %19#0, %19#1, %19#2 : !singlePhysType, !singlePhysType, !singlePhysType, !singlePhysType
     Jac : %16 : !singleJacobianType
     Weights : %3, %4, %5 : tensor<4xf64>, tensor<5xf64>, tensor<6xf64>]
     -> !singlePhysType, !singlePhysType, !singlePhysType, !singlePhysType

    %21 = nir.elmnt_test[ // action of B^T
    Block : %20#0 : !singlePhysType
     Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>
      Scale : 2.0: f64] -> !singleCoeffType

    %22 = nir.elmnt_test_grad[ // "dot product" grad(v) and grad(u) and action of B^T
    Blocks : %20#1, %20#2, %20#3 : !singlePhysType, !singlePhysType, !singlePhysType
     DMats : %6, %7, %8 : tensor<4x4xf64>, tensor<5x5xf64>, tensor<6x6xf64>
     Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>]
     -> !singleCoeffType

    %23 = nir.add[
    Blocks : %21, %22 : !singleCoeffType, !singleCoeffType]
    -> !singleCoeffType

    %24 = nir.insert_slice %23 into %arg15[0, %arg14, 0, 0, 0] [1, 1, 3, 4, 5] [1, 1, 1, 1, 1] : !singleCoeffType into !coeffBlockType

    scf.yield %24 : !coeffBlockType

  } {element_shape = #nir.element_shape<hex>}
```

Loop over elements

Extract a single block

```
%13 = nir.empty_block() : !coeffBlockType
  %c0 = arith.constant 0 : index
  %c1000 = arith.constant 1000 : index
  %c1 = arith.constant 1 : index
  %14 = scf.for %arg14 = %c0 to %c1000 step %c1 iter_args(%arg15 = %13) -> (!coeffBlockType) {

    %15 = nir.extract_slice %11  [0, %arg14, 0, 0, 0] [1, 1, 3, 4, 5] [1, 1, 1, 1, 1] : !coeffBlockType to !singleCoeffType
    %16 = nir.extract_slice %12  [0, %arg14, 0, 0, 0] [1, 1, 1, 1, 1] [1, 1, 1, 1, 1] : !jacobianBlockType to !singleJacobianType

    %17 = nir.elmnt_bwd[ // backward transform
     Block : %15 : !singleCoeffType
     Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>] -> !singlePhysType

    %18:3 = nir.elmnt_standard_deriv[ // derivative in local coordinates
     Block : %17 : !singlePhysType
     DMats : %6, %7, %8 : tensor<4x4xf64>, tensor<5x5xf64>, tensor<6x6xf64>]
     -> !singlePhysType, !singlePhysType, !singlePhysType

    %19:3 = nir.elmnt_deriv_metric[ // apply the derivative metric and diffusion
    Blocks : %18#0, %18#1, %18#2 : !singlePhysType, !singlePhysType, !singlePhysType
     Factors : %9 : tensor<9xf64>
     DiffCoeffs : %10 : tensor<6xf64>
    ] -> !singlePhysType, !singlePhysType, !singlePhysType

    %20:4 = nir.elmnt_apply_jw[ // apply weights and jacobian determinants
    Blocks : %17, %19#0, %19#1, %19#2 : !singlePhysType, !singlePhysType, !singlePhysType, !singlePhysType
     Jac : %16 : !singleJacobianType
     Weights : %3, %4, %5 : tensor<4xf64>, tensor<5xf64>, tensor<6xf64>]
     -> !singlePhysType, !singlePhysType, !singlePhysType, !singlePhysType

    %21 = nir.elmnt_test[ // action of B^T
    Block : %20#0 : !singlePhysType
    Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>
      Scale : 2.0: f64] -> !singleCoeffType

    %22 = nir.elmnt_test_grad[ // "dot product" grad(v) and grad(u) and action of B^T
    Blocks : %20#1, %20#2, %20#3 : !singlePhysType, !singlePhysType, !singlePhysType
     DMats : %6, %7, %8 : tensor<4x4xf64>, tensor<5x5xf64>, tensor<6x6xf64>
     Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>]
     -> !singleCoeffType

    %23 = nir.add[
    Blocks : %21, %22 : !singleCoeffType, !singleCoeffType]
    -> !singleCoeffType

    %24 = nir.insert_slice %23 into %arg15[0, %arg14, 0, 0, 0] [1, 1, 3, 4, 5] [1, 1, 1, 1, 1] : !singleCoeffType into !coeffBlockType

    scf.yield %24 : !coeffBlockType

  } {element_shape = #nir.element_shape<hex>}
```

**1. Backward Transform**

**2. Collocation Derivative**

**3. Derivative Metric**

**4. Apply weights and Jacobian determinants**

**5. (u,v)**

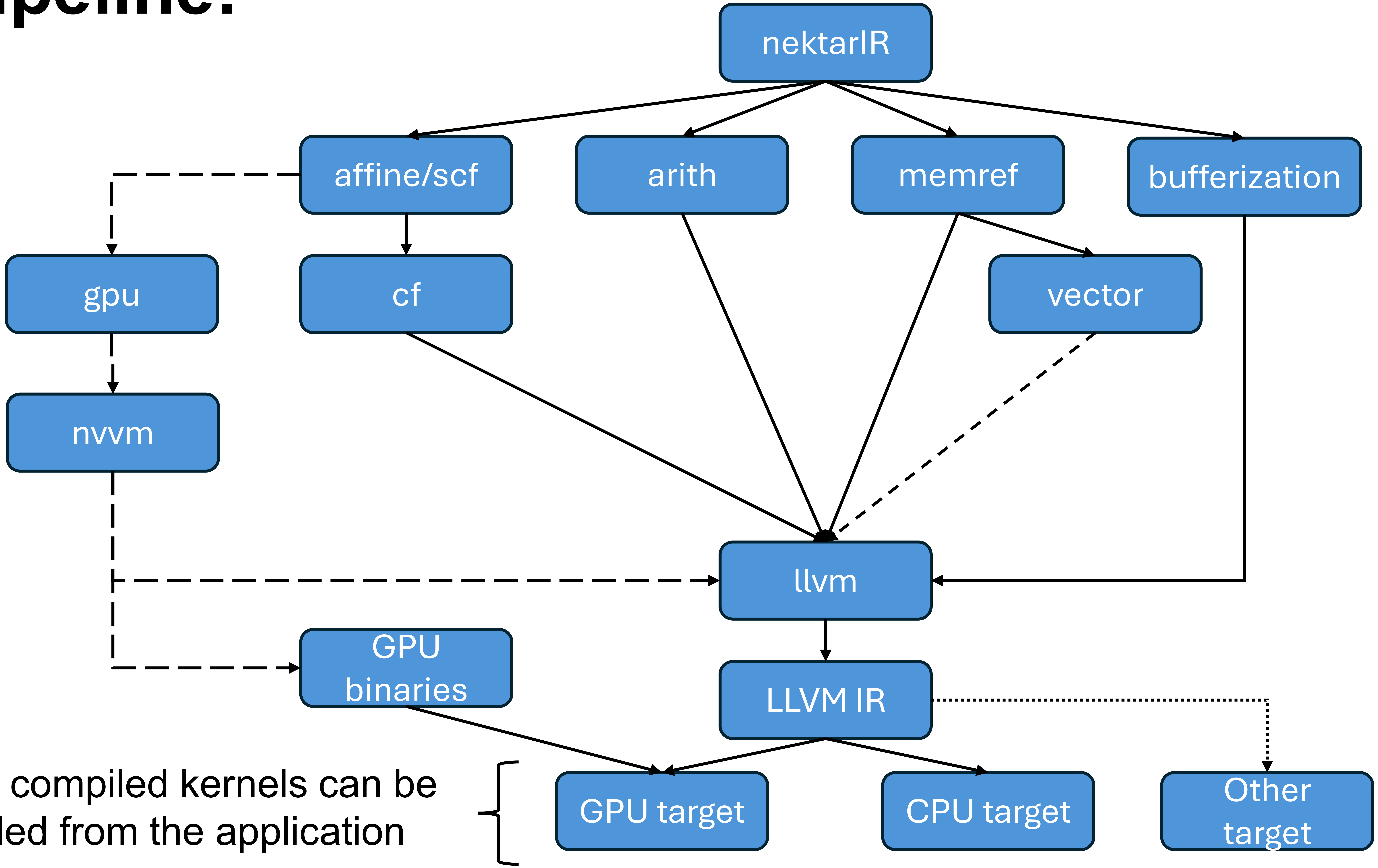
**6. (grad(u), grad(v))**

**7. Add mass and stiffness contributions**

```
%13 = nir.empty_block() : !coeffBlockType
  %c0 = arith.constant 0 : index
  %c1000 = arith.constant 1000 : index
  %c1 = arith.constant 1 : index
  %14 = scf.for %arg14 = %c0 to %c1000 step %c1 iter_args(%arg15 = %13) -> (!coeffBlockType) {

    %15 = nir.extract_slice %11  [0, %arg14, 0, 0, 0] [1, 1, 3, 4, 5] [1, 1, 1, 1, 1] : !coeffBlockType to !singleCoeffType
    %16 = nir.extract_slice %12  [0, %arg14, 0, 0, 0] [1, 1, 1, 1, 1] [1, 1, 1, 1, 1] : !jacobianBlockType to !singleJacobianType

    %17 = nir.elmnt_bwd[ // backward transform
     Block : %15 : !singleCoeffType
     Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>] -> !singlePhysType

    %18:3 = nir.elmnt_standard_deriv[ // derivative in local coordinates
     Block : %17 : !singlePhysType
     DMats : %6, %7, %8 : tensor<4x4xf64>, tensor<5x5xf64>, tensor<6x6xf64>]
     -> !singlePhysType, !singlePhysType, !singlePhysType

    %19:3 = nir.elmnt_deriv_metric[ // apply the derivative metric and diffusion
    Blocks : %18#0, %18#1, %18#2 : !singlePhysType, !singlePhysType, !singlePhysType
     Factors : %9 : tensor<9xf64>
     DiffCoeffs : %10 : tensor<6xf64>
    ] -> !singlePhysType, !singlePhysType, !singlePhysType

    %20:4 = nir.elmnt_apply_jw[ // apply weights and jacobian determinants
    Blocks : %17, %19#0, %19#1, %19#2 : !singlePhysType, !singlePhysType, !singlePhysType, !singlePhysType
     Jac : %16 : !singleJacobianType
     Weights : %3, %4, %5 : tensor<4xf64>, tensor<5xf64>, tensor<6xf64>]
     -> !singlePhysType, !singlePhysType, !singlePhysType, !singlePhysType

    %21 = nir.elmnt_test[ // action of B^T
    Block : %20#0 : !singlePhysType
     Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>
      Scale : 2.0: f64] -> !singleCoeffType

    %22 = nir.elmnt_test_grad[ // "dot product" grad(v) and grad(u) and action of B^T
    Blocks : %20#1, %20#2, %20#3 : !singlePhysType, !singlePhysType, !singlePhysType
     DMats : %6, %7, %8 : tensor<4x4xf64>, tensor<5x5xf64>, tensor<6x6xf64>
     Bases : %0, %1, %2 : tensor<4x3xf64>, tensor<5x4xf64>, tensor<6x5xf64>]
     -> !singleCoeffType

    %23 = nir.add[
    Blocks : %21, %22 : !singleCoeffType, !singleCoeffType]
    -> !singleCoeffType

    %24 = nir.insert_slice %23 into %arg15[0, %arg14, 0, 0, 0] [1, 1, 3, 4, 5] [1, 1, 1, 1, 1] : !singleCoeffType into !coeffBlockType

    scf.yield %24 : !coeffBlockType

  } {element_shape = #nir.element_shape<hex>}
```

Insert result block
and update the result

# NektarIR pipeline:




Other vendor specific dialects exist

JIT compiled kernels can be called from the application

# NektarIR interface:

- Programmatic construction of the IR using our C++ IRGenerator library
- JIT-compilation of generated or written IR snippets using the LLVM execution engine
- JIT library creates a function pointer to the compiled kernel
- Python bindings are planned

# Compiler Overhead and Runtime Performance:

- How long does lowering and compiling a kernel take?
- Is there a difference in the overhead for CPU and GPU code-gen?
- How does runtime performance compare to Nektar++?

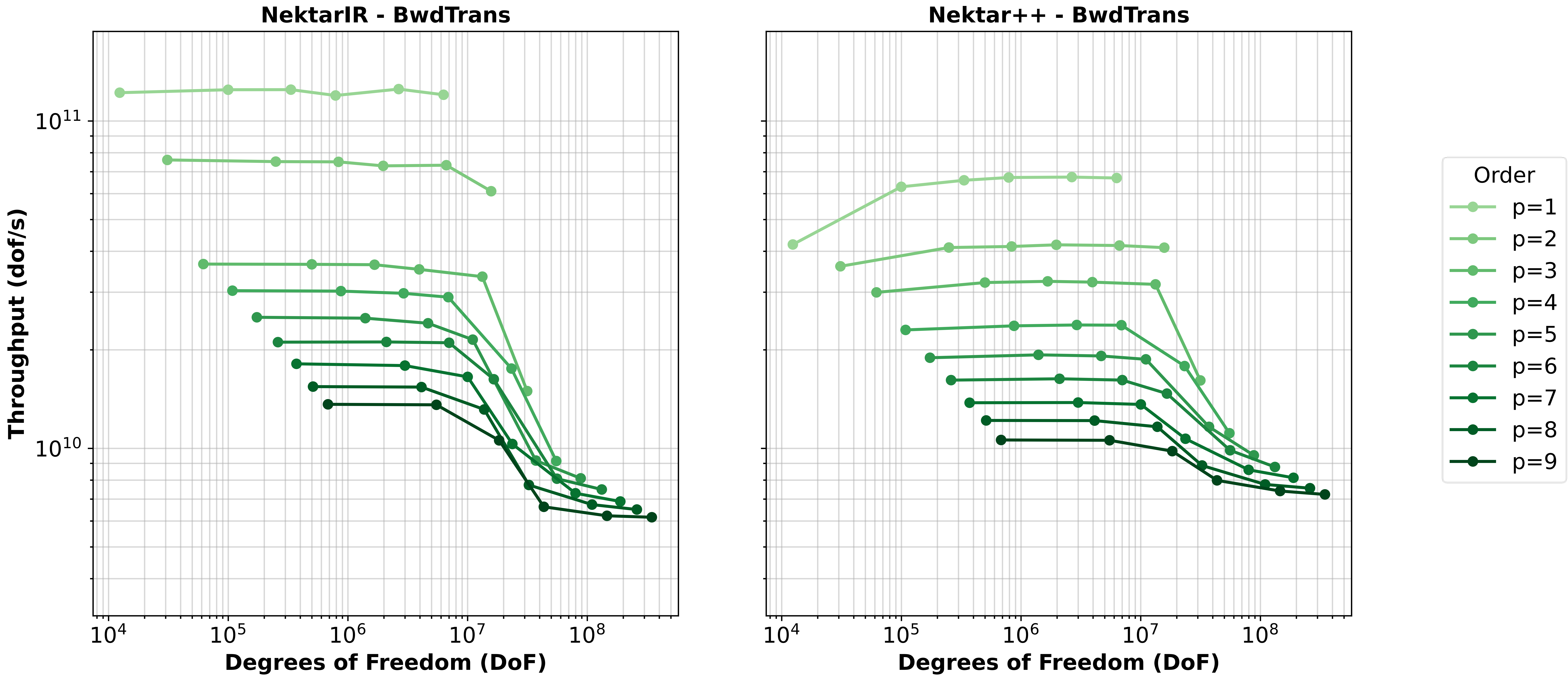
## Runtime Comparison:

- Comparison of both vectorized and GPU kernels from NektarIR and the Nektar++ Redesign
- Host: 128 cores on two AMD EPYC 9554 CPUs
- Device: NVIDIA H100

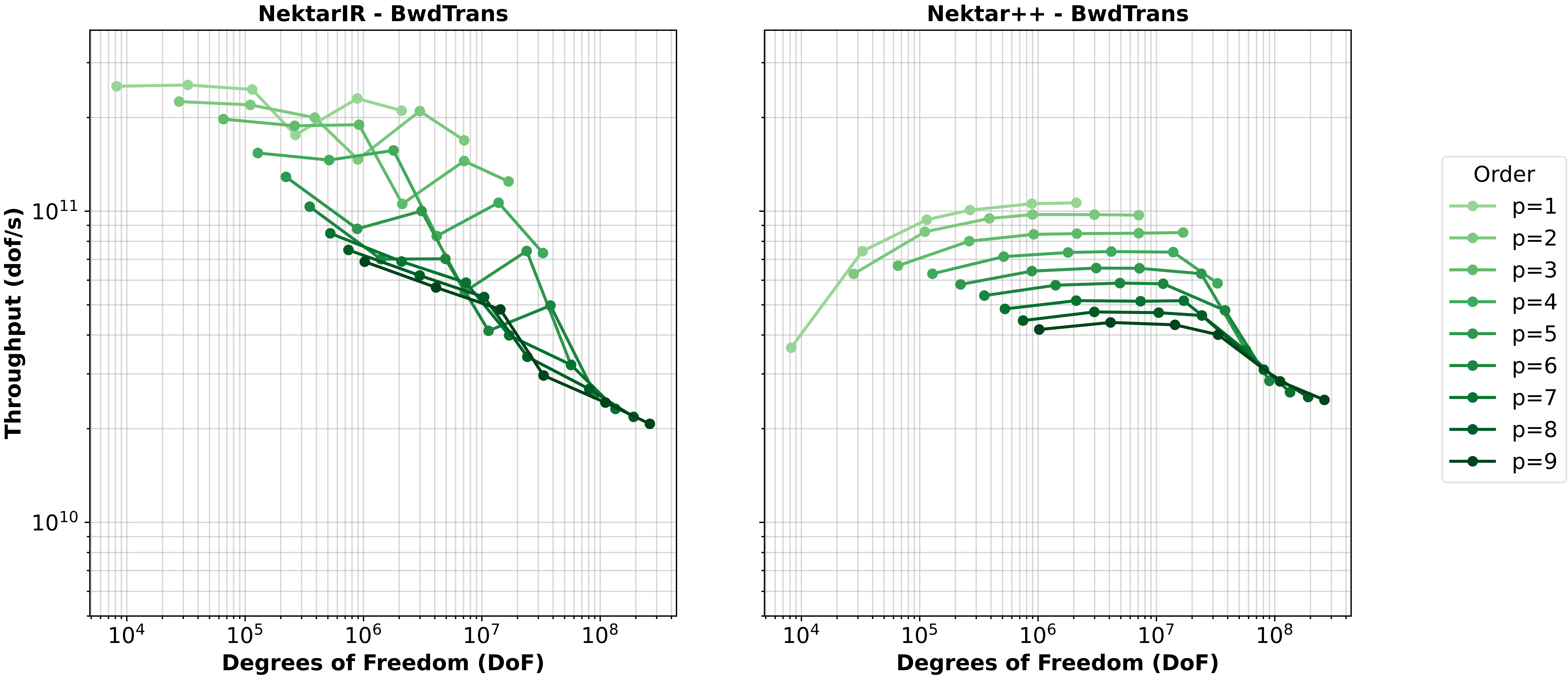
## Overhead:

- Measured time to lower from NektarIR to the LLVM dialect and time to compile

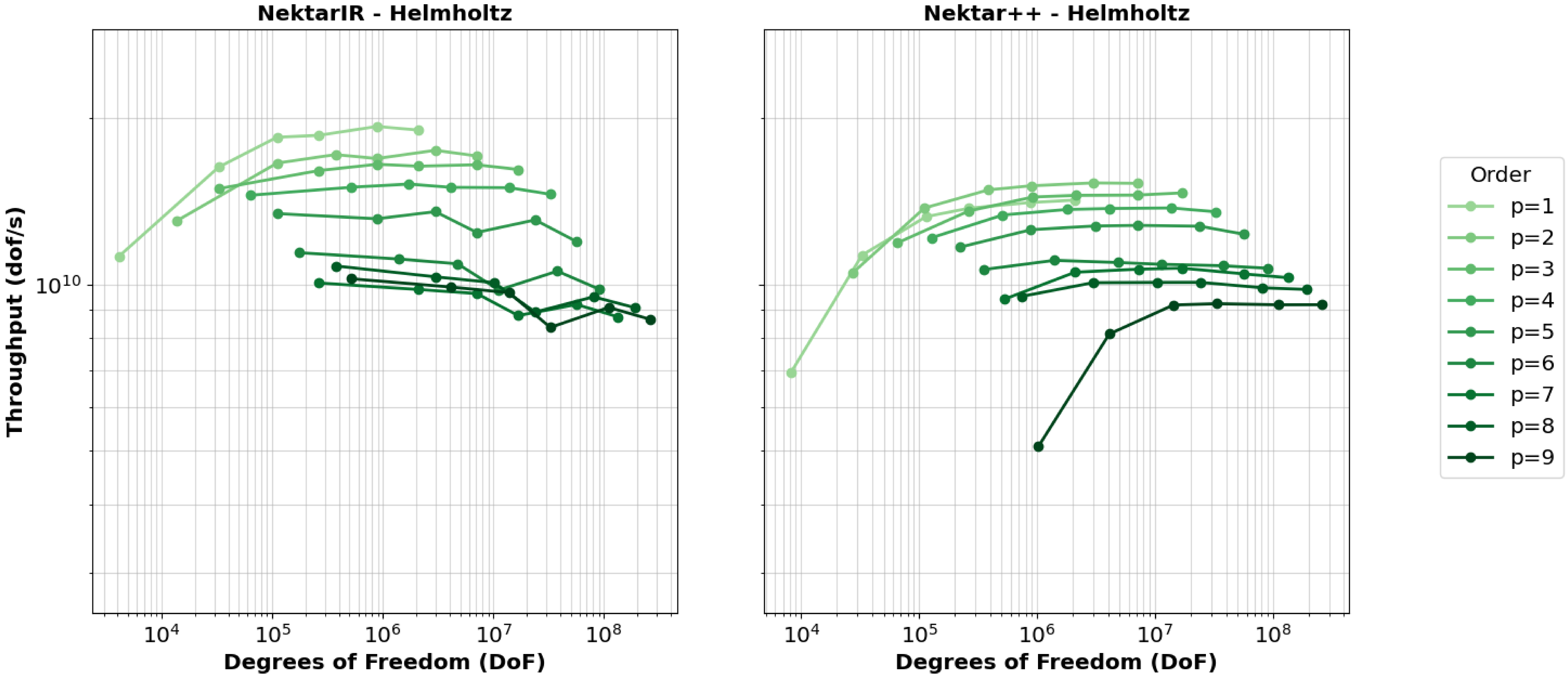
# Tetrahedral Elements, AVX512:



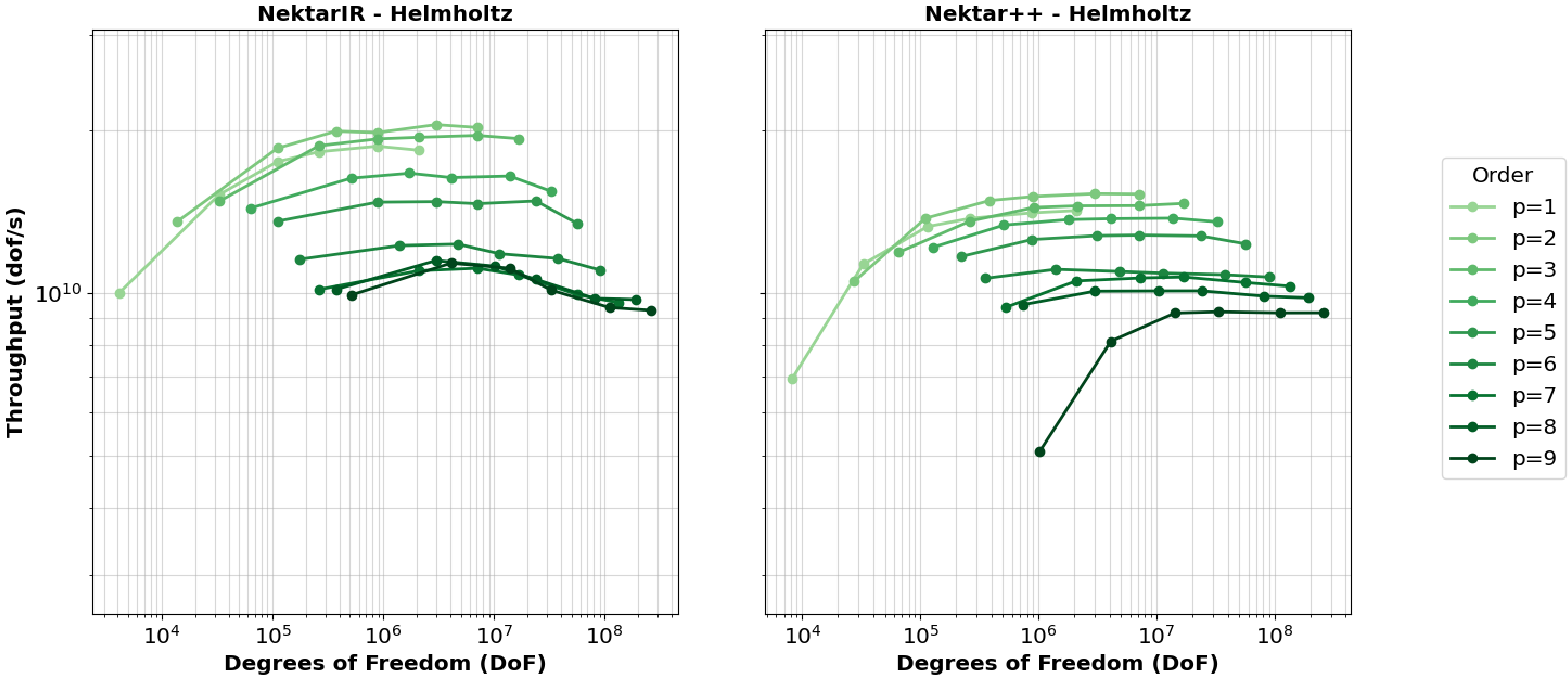
# Hexahedral Elements, AVX512:



# Hexahedral Elements, AVX512:

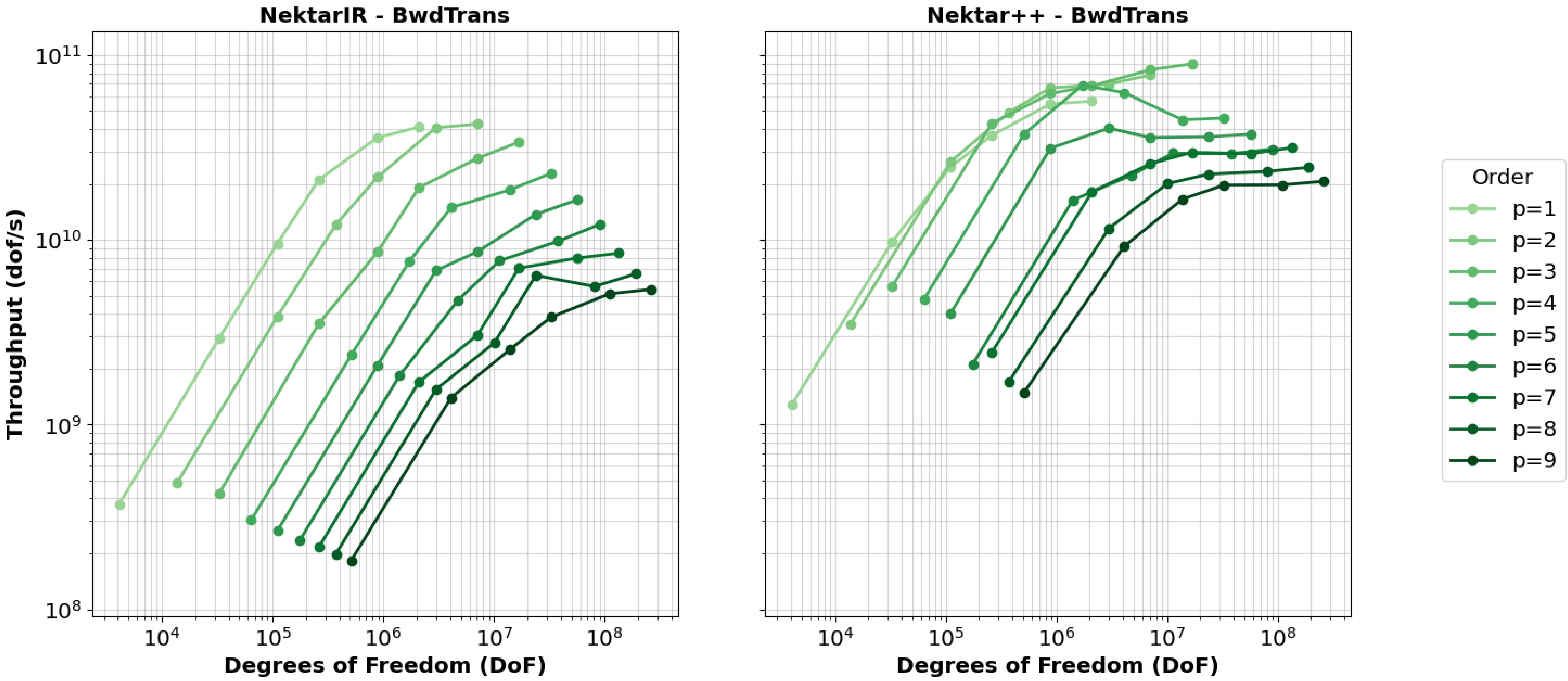


# Hexahedral Elements, AVX512 with loop fusion:

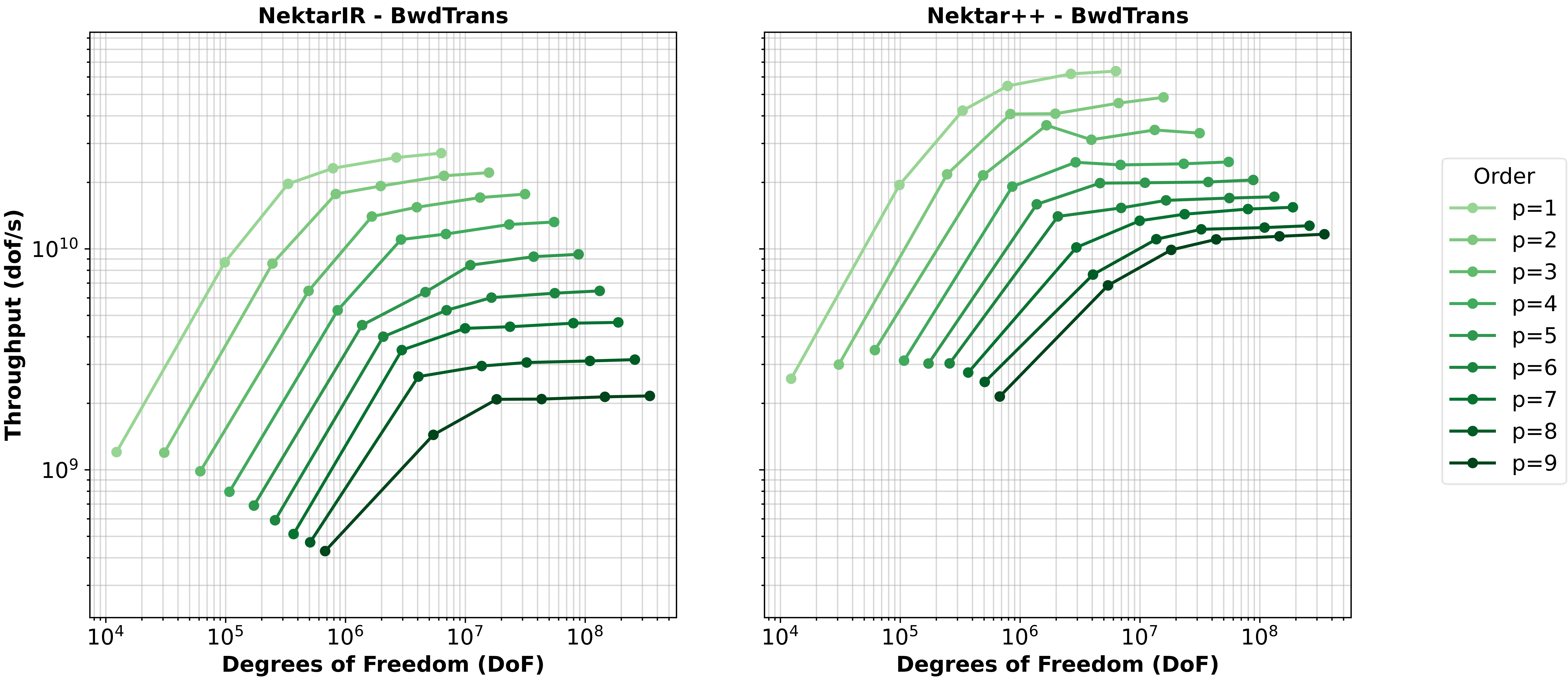


# Hexahedral Elements: Overhead Comparison

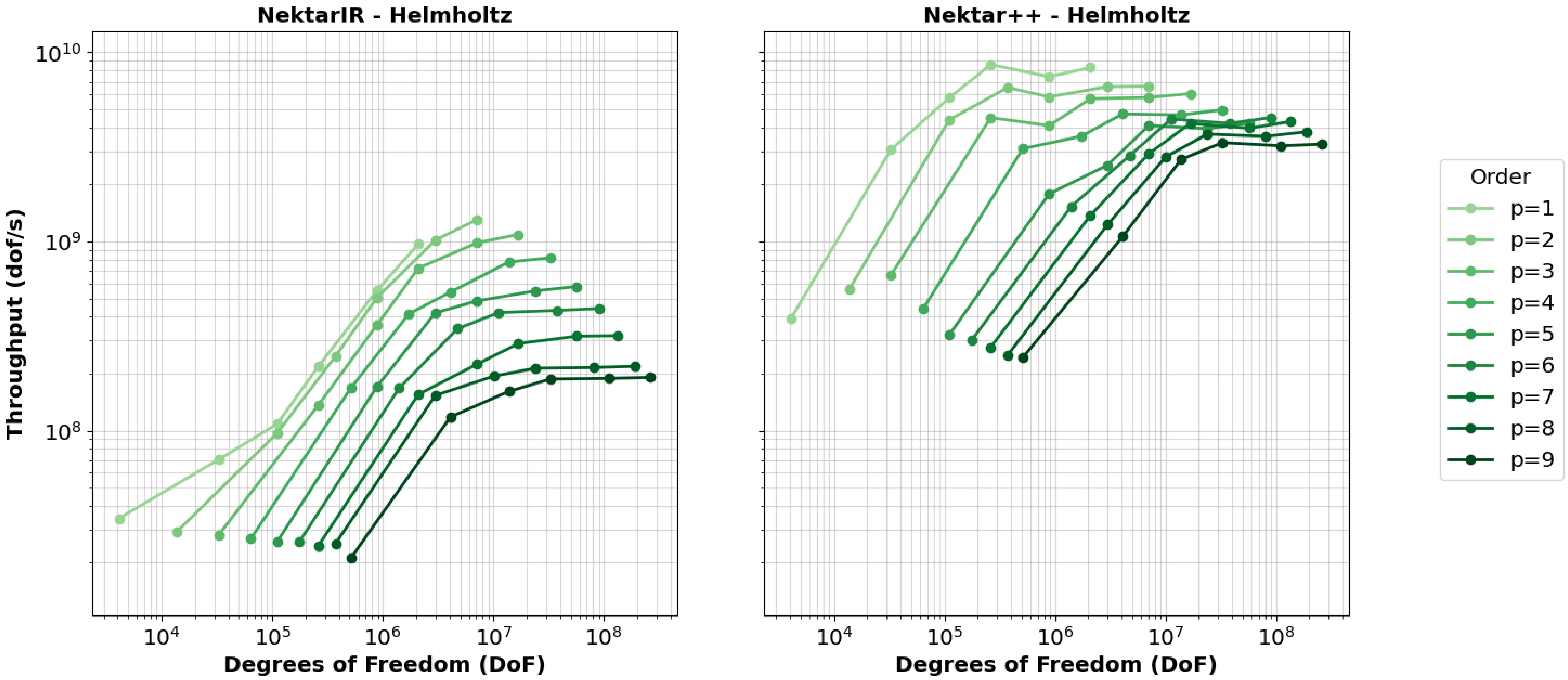
# GPU: Hexahedral Elements, Threading Over Elements



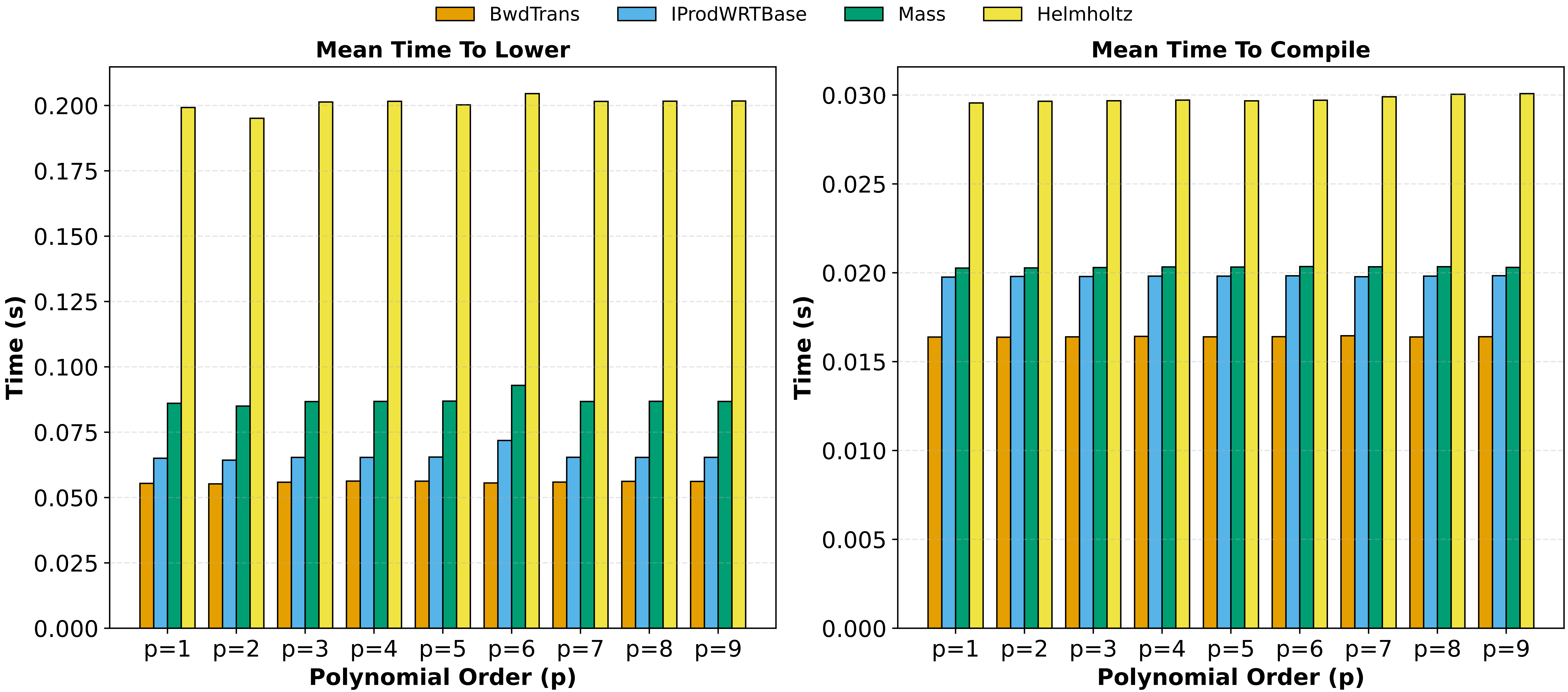
# GPU: Tetrahedral Elements, Threading Over Elements



# GPU: Hexahedral Elements, Threading Over Elements



# Compiler Overhead: Target: NVIDIA H100, Hexahedral Elements



## Summary:

- Code-generation of JIT compiled kernels for both CPU and GPU targets from a single representation using MLIR and LLVM.

- Good performance on CPU without many optimizations.

- GPU kernels need optimization.

- MLIR gives control of the kernel from “math to metal” and lots of optimizations remain to be tested.

# Future work:

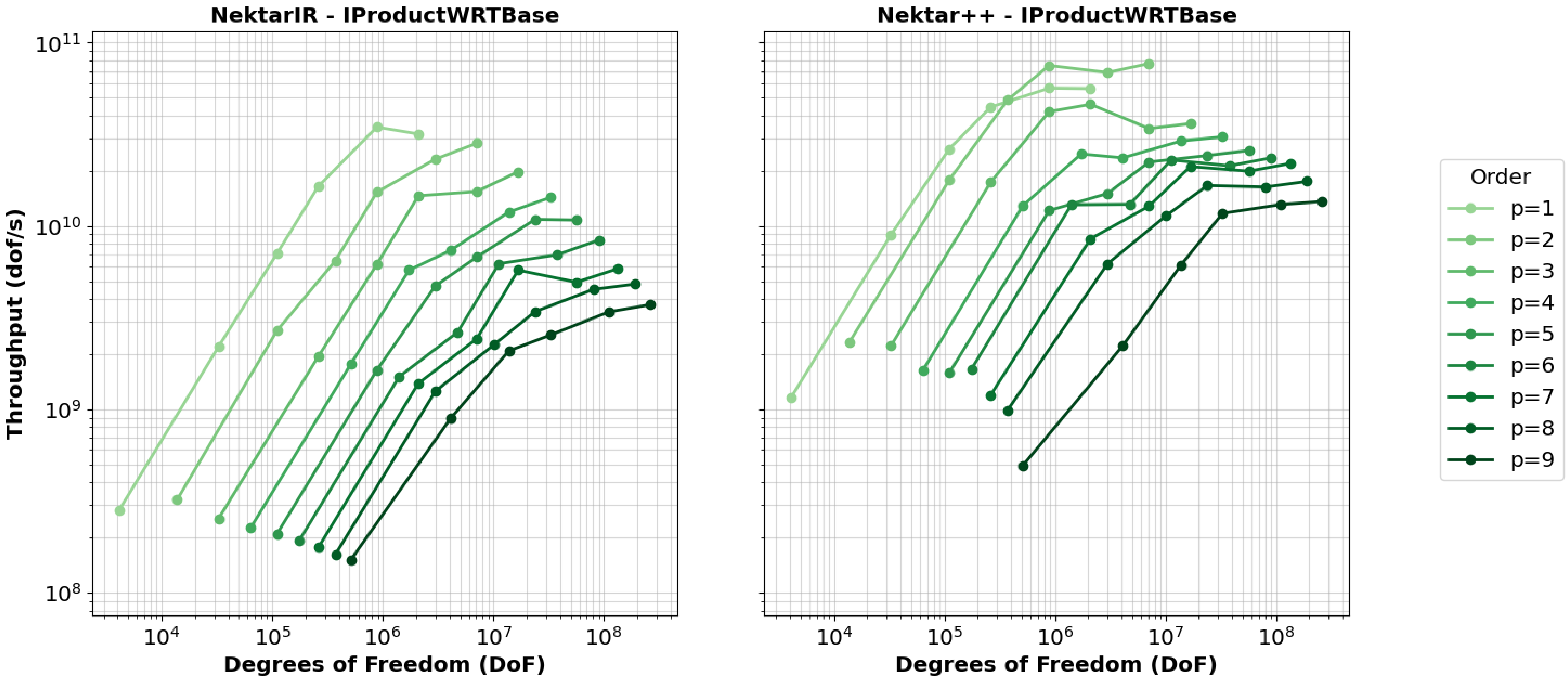
- Optimization.
- Test on more hardware.
- Get all operators working on all shapes.
- Nodal expansions and the collocation property.

# The Code:

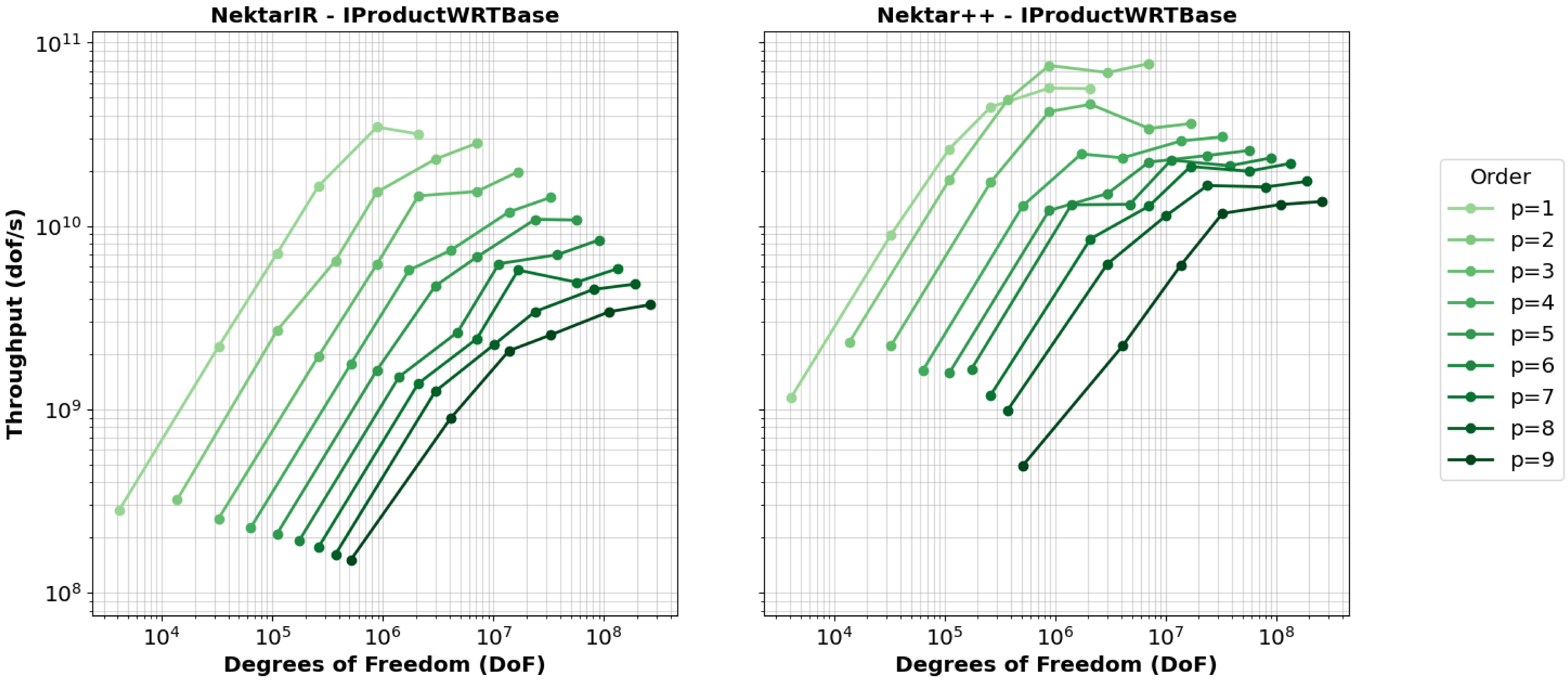
- Will be open-sourced and independent of Nektar++.
- Nearly ready for release but if you want access, you can contact us!

Thank you for listening!

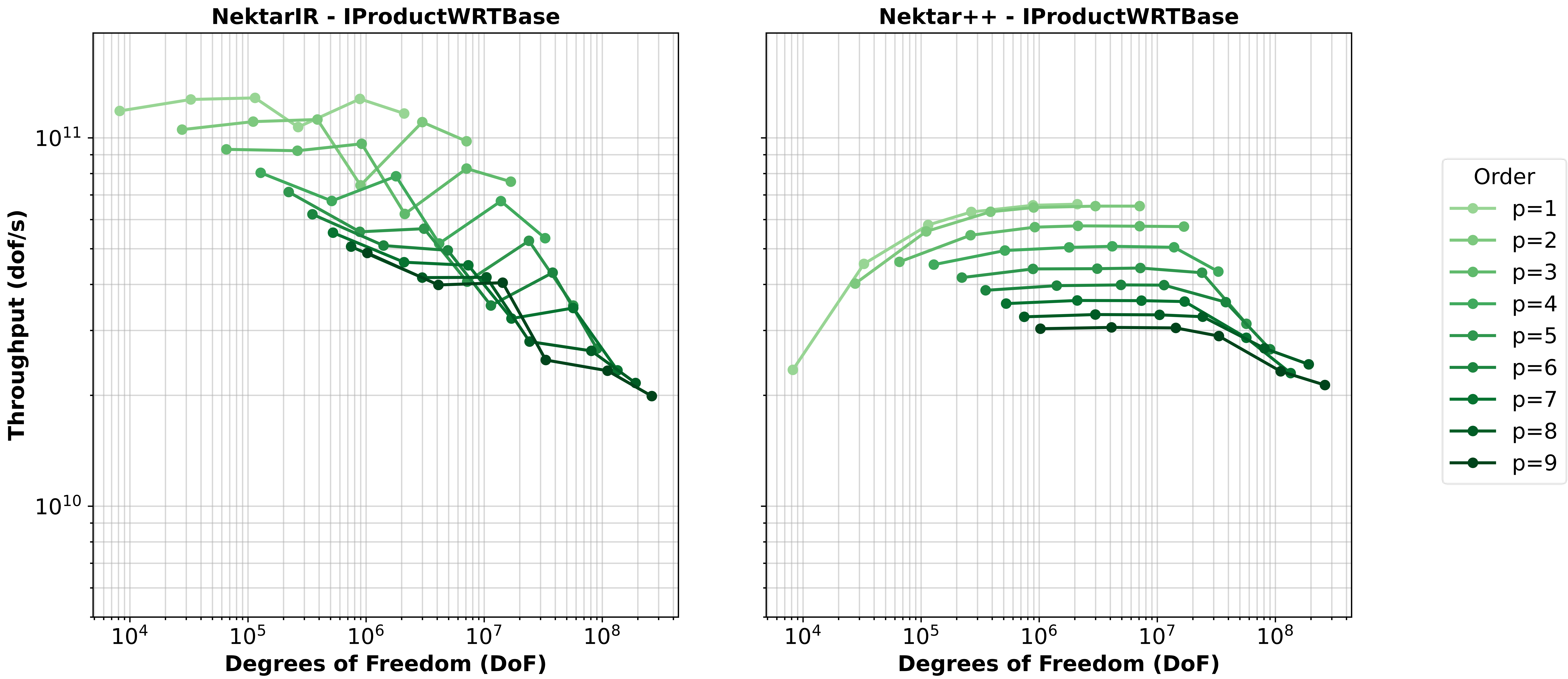
# GPU: Hexahedral Elements, Threading Over Elements



# GPU: Hexahedral Elements, Threading Over Elements



# Hexahedral Elements, AVX512



# Hexahedral Elements, AVX512: