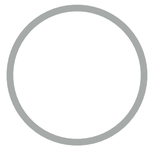
Experiences with OpenCL in PyFR: 2014—Present

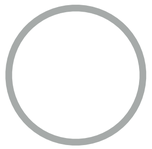
F.D. Witherden[1] and P.E. Vincent[2]

[1]Department of Ocean Engineering, Texas A&M University
[2]Department of Aeronautics, Imperial College London

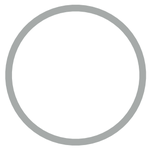
Motivation



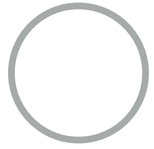
High-Order Methods



PyFR



OpenCL Backend

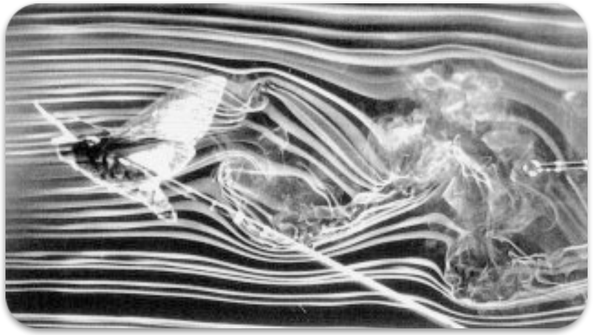


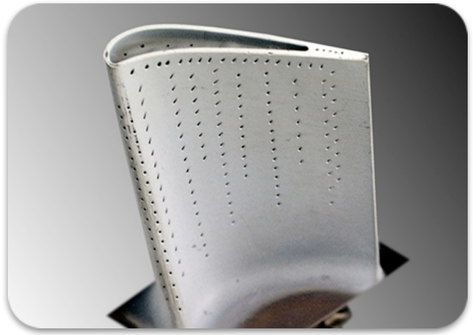
Future

Motivation

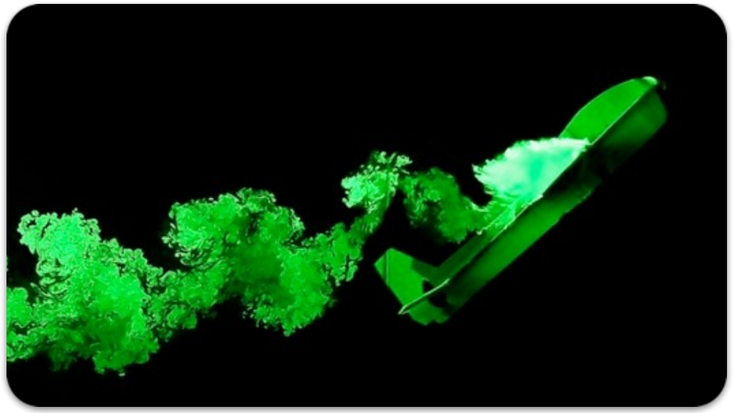
- Computational fluid dynamics (CFD) is the bedrock of several high-tech industries.

Motivation



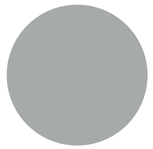


- Interested in simulating unsteady, turbulent, flows.



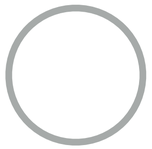
Motivation

- Objective is to solve the **Navier–Stokes** equations in the vicinity of **complex geometrical configurations**.

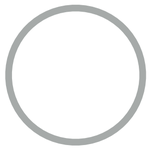


Motivation

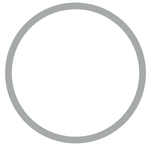
High-Order Methods



PyFR



OpenCL Backend



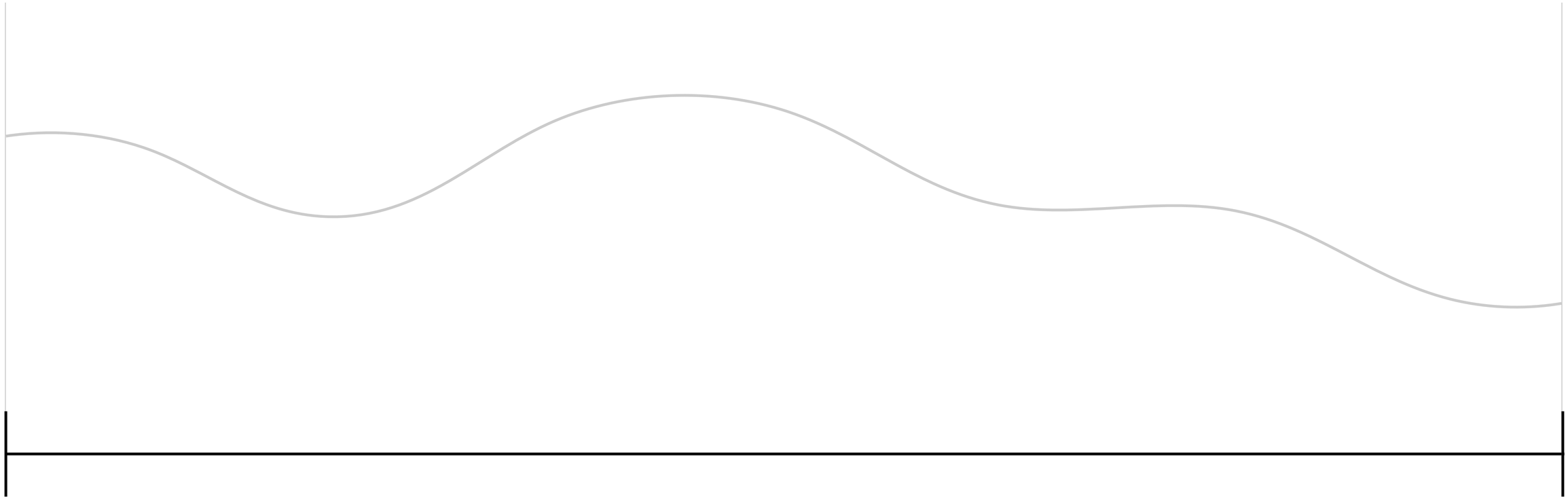
Future

High-Order Methods

- Our choice of method is the high-order accurate **Flux Reconstruction** (FR) approach of Huynh.
- Combines aspects of traditional finite volume (FVM) and finite element methods (FEM).

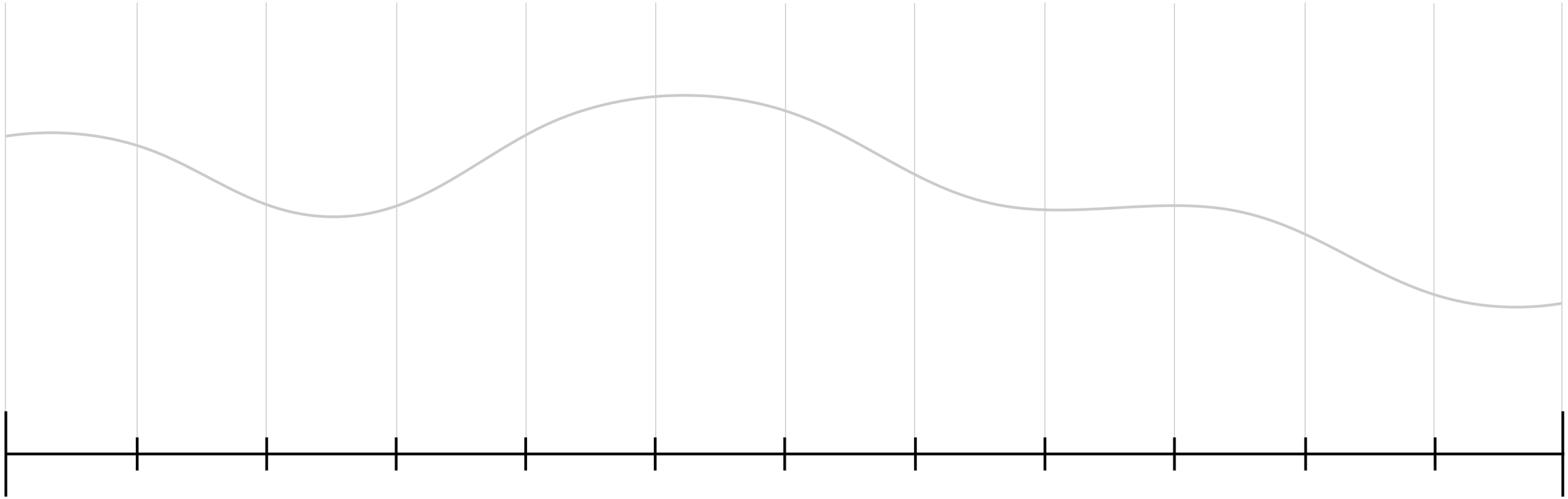
High-Order Methods

- Consider a smooth function



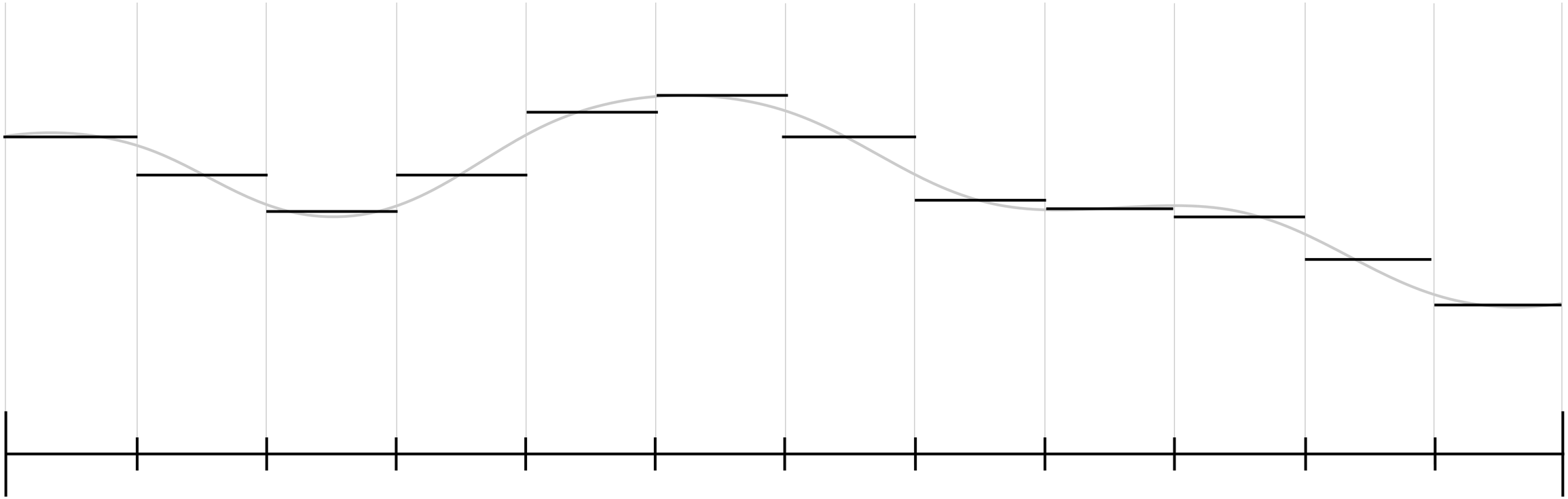
High-Order Methods

- In FVM we divide the domain into **cells**…



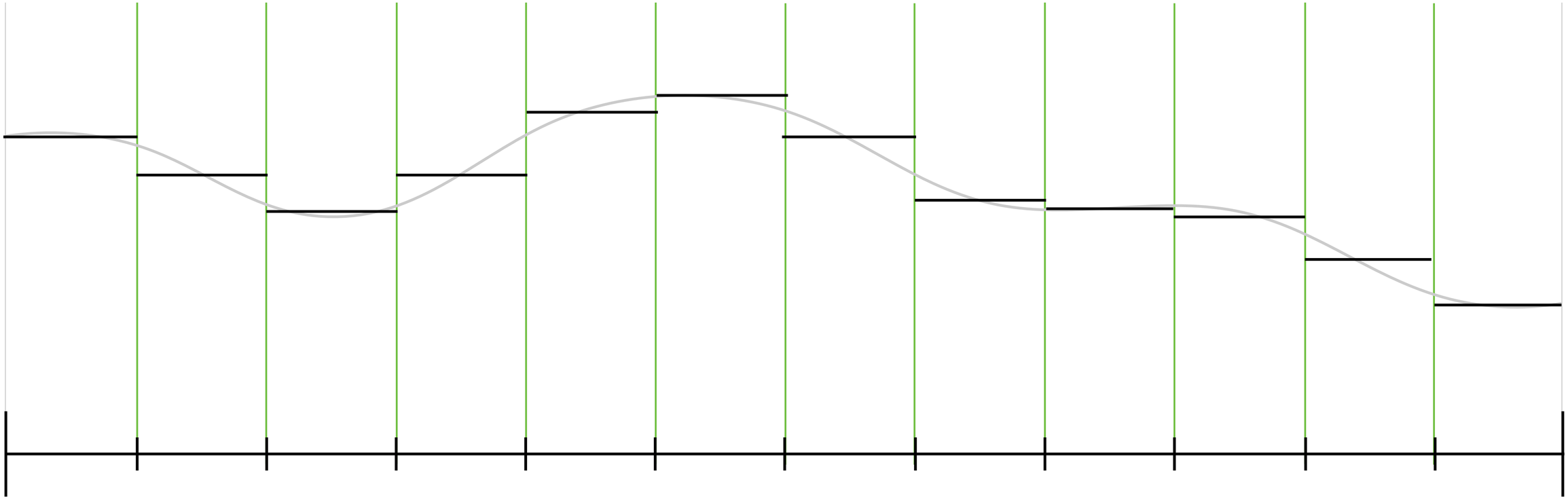
High-Order Methods

- …and in each cell store the **average** of the function.



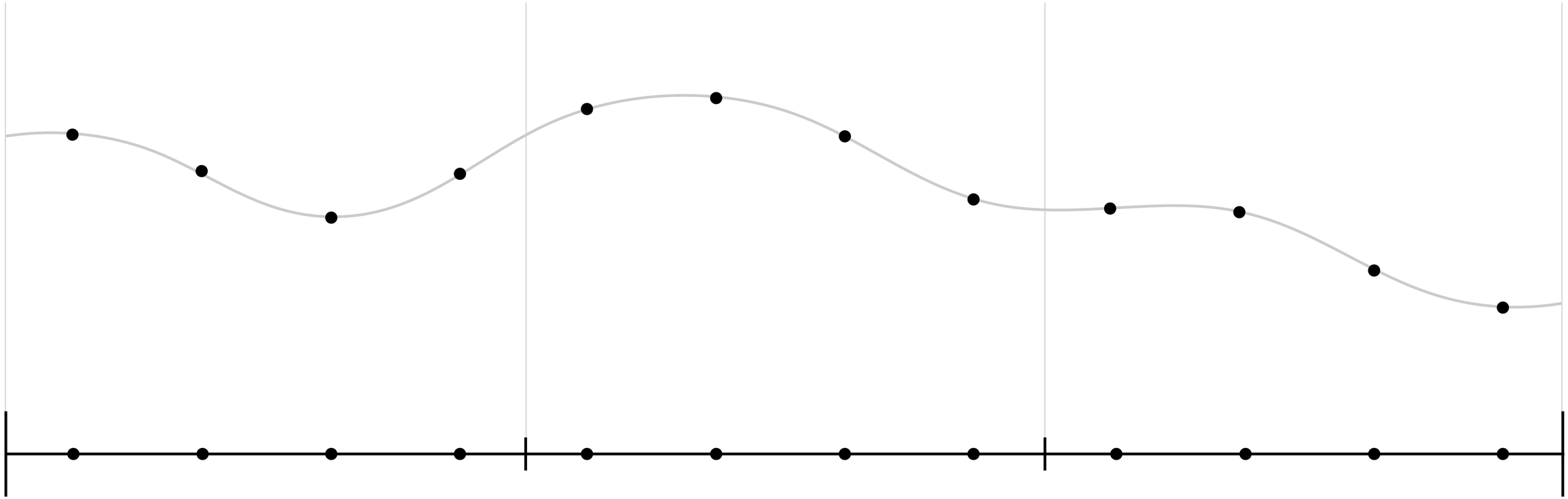
High-Order Methods

- Cells are coupled via Riemann solves at the interfaces.



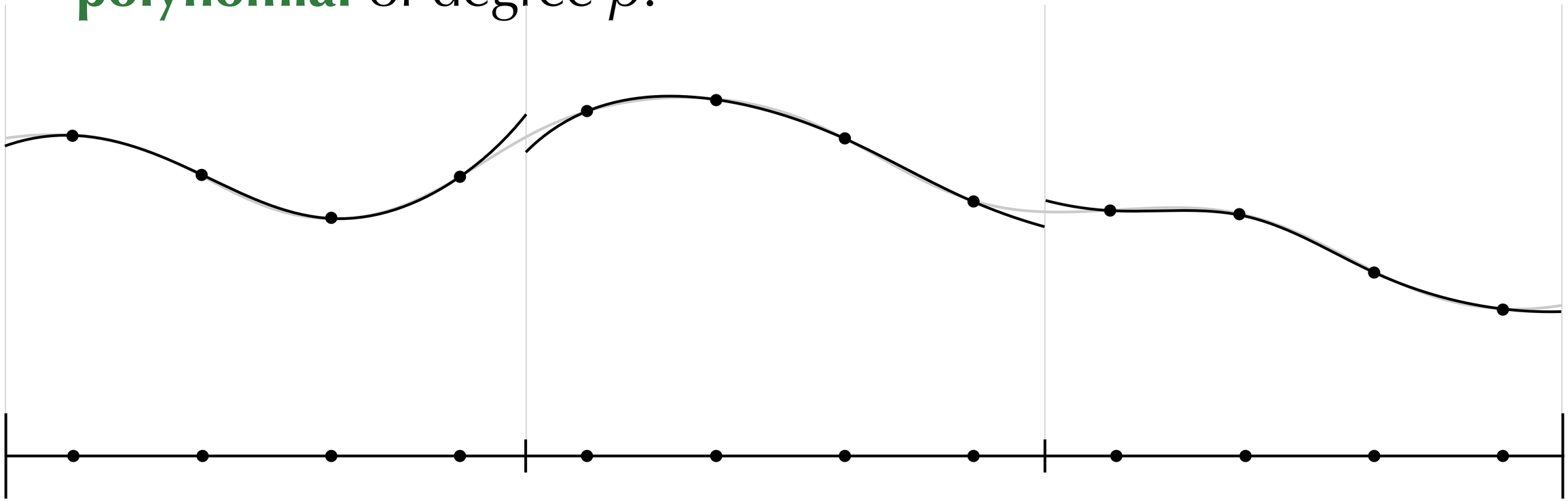
High-Order Methods

- In FR we divide the domain into **elements**…



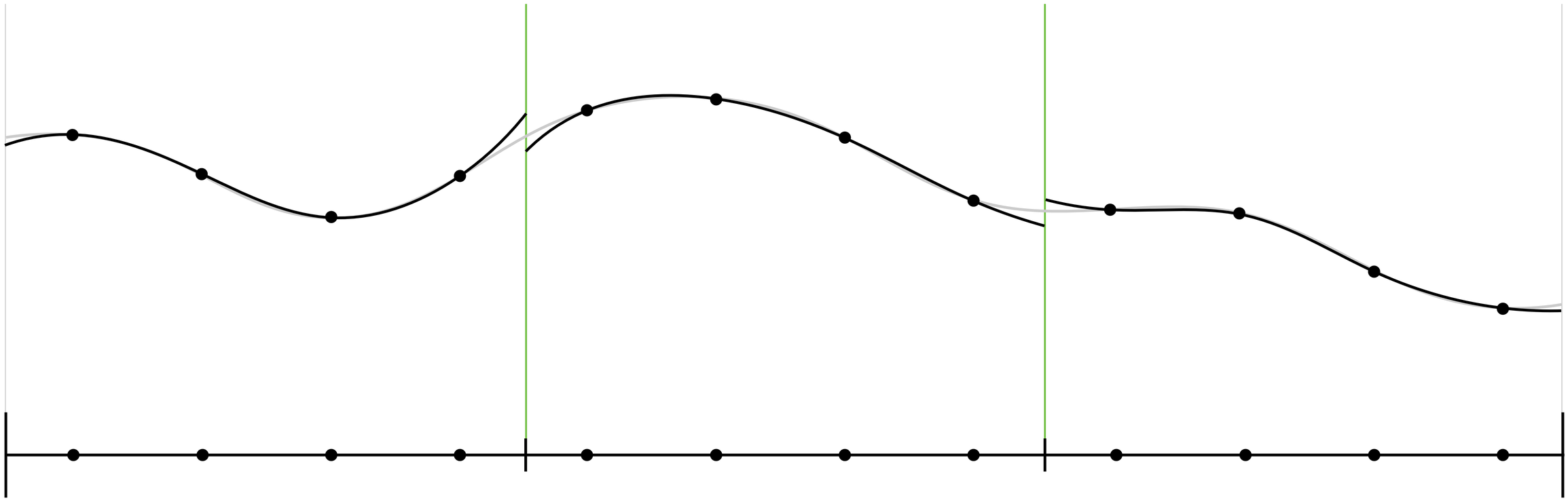
High-Order Methods

- …and in each element store a **discontinuous interpolating polynomial** of degree p.



High-Order Methods

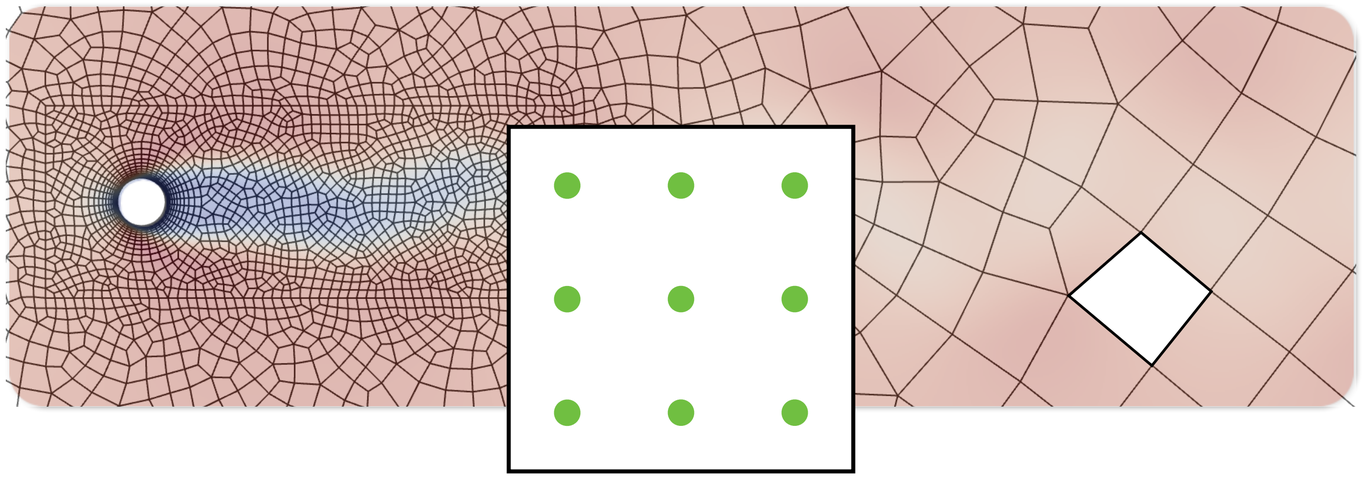
- As before elements are coupled via Riemann solves.



High-Order Methods

- Greater **resolving power** per degree of freedom (DOF)…
  - …and thus **fewer overall DOFs** for same accuracy.
- Tight **coupling between DOFs** inside of an element…
  - …reduces indirection and **saves memory bandwidth**.

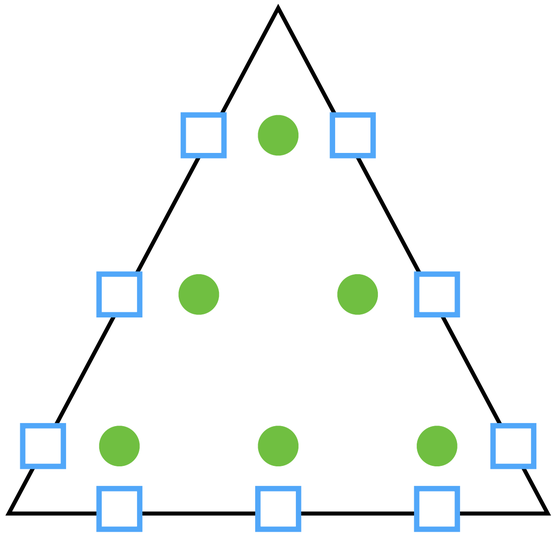
High-Order Methods

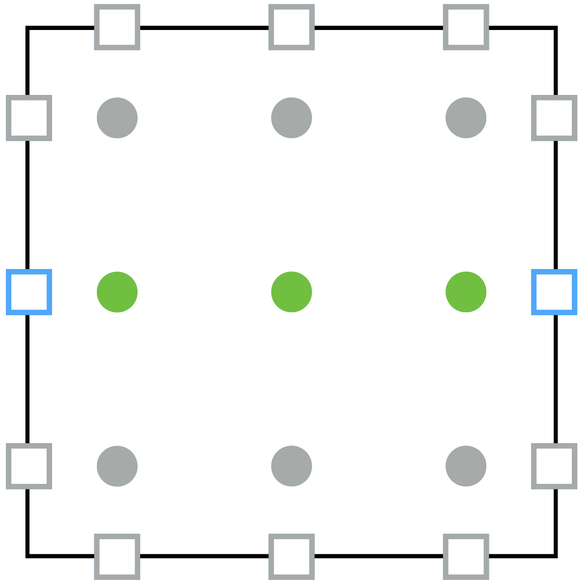


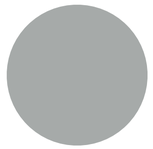
- Direct extension into 2D and 3D.

High-Order Methods

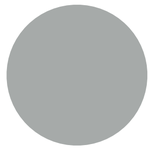
- Most operations can be cast as matrix-matrix products.
- Element type determines if the operation is sparse or dense.





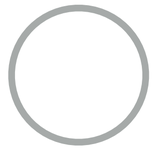


Motivation

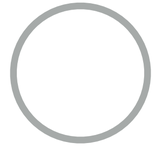


High-Order Methods

PyFR



OpenCL Backend



Future

PyFR

Python + Flux Reconstruction

PyFR

- High level structure.

Python Outer Layer
(**Hardware Independent**)

- Setup
- Distributed memory parallelism
- Outer loop calls **hardware specific kernels**

PyFR

- Need to generate **hardware specific kernels**.

Python Outer Layer
(**Hardware Independent**)

- Setup
- Distributed memory parallelism
- Outer loop calls **hardware specific kernels**

PyFR

- In FR **two types** of kernel are required.

Python Outer Layer
(**Hardware Independent**)

- Setup
- Distributed memory parallelism
- Outer loop calls **hardware specific kernels**

Matrix Multiply Kernels

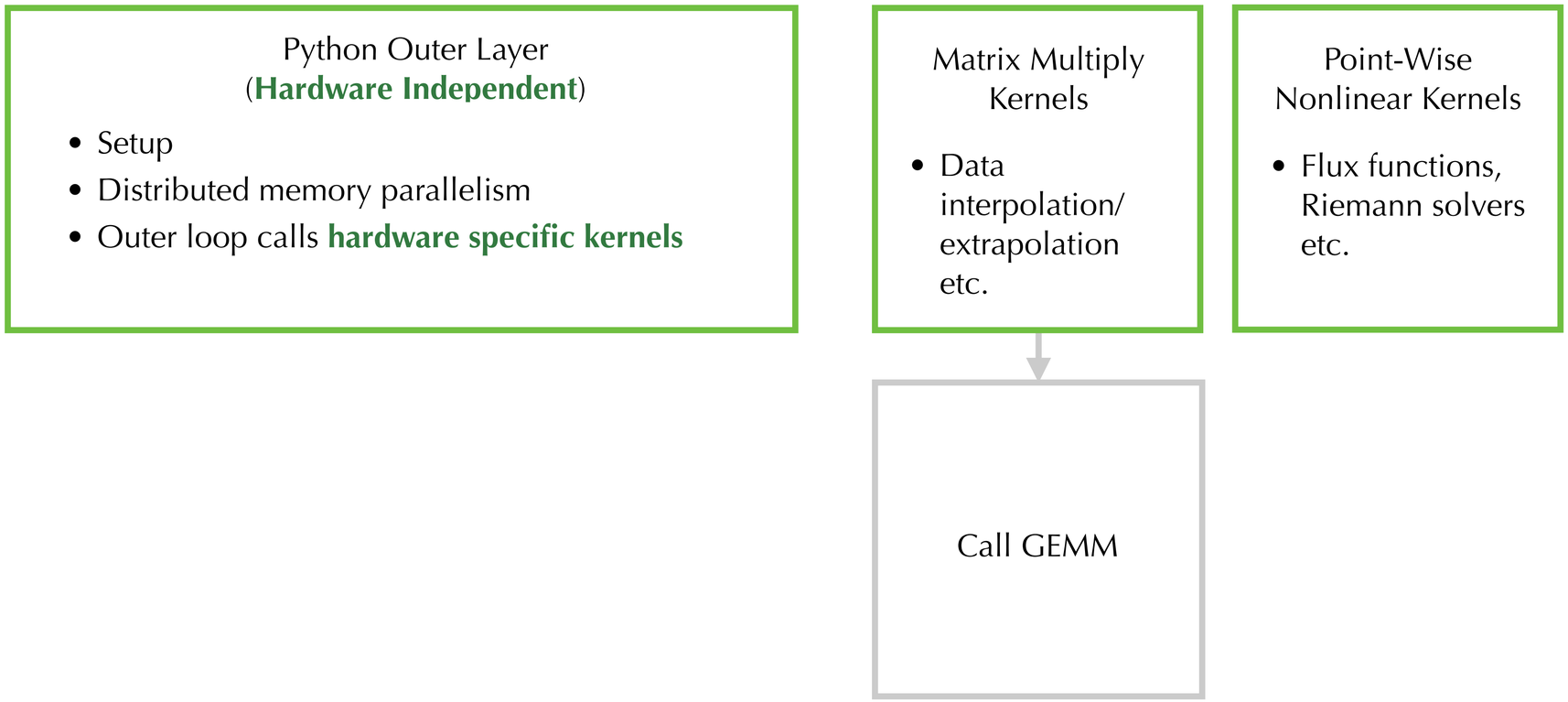
- Data interpolation/ extrapolation etc.

Point-Wise Nonlinear Kernels

- Flux functions, Riemann solvers etc.

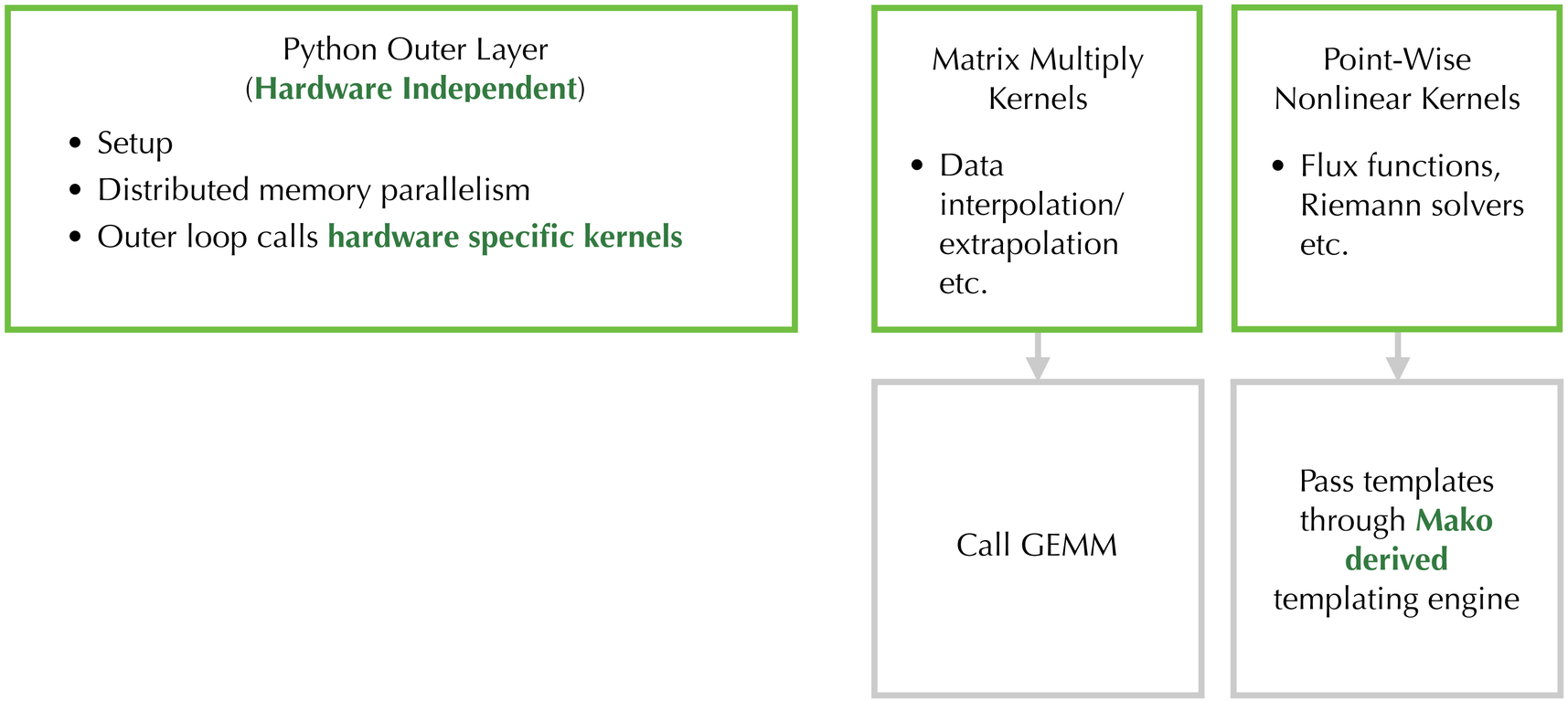
PyFR

- Matrix multiplications are quite simple.



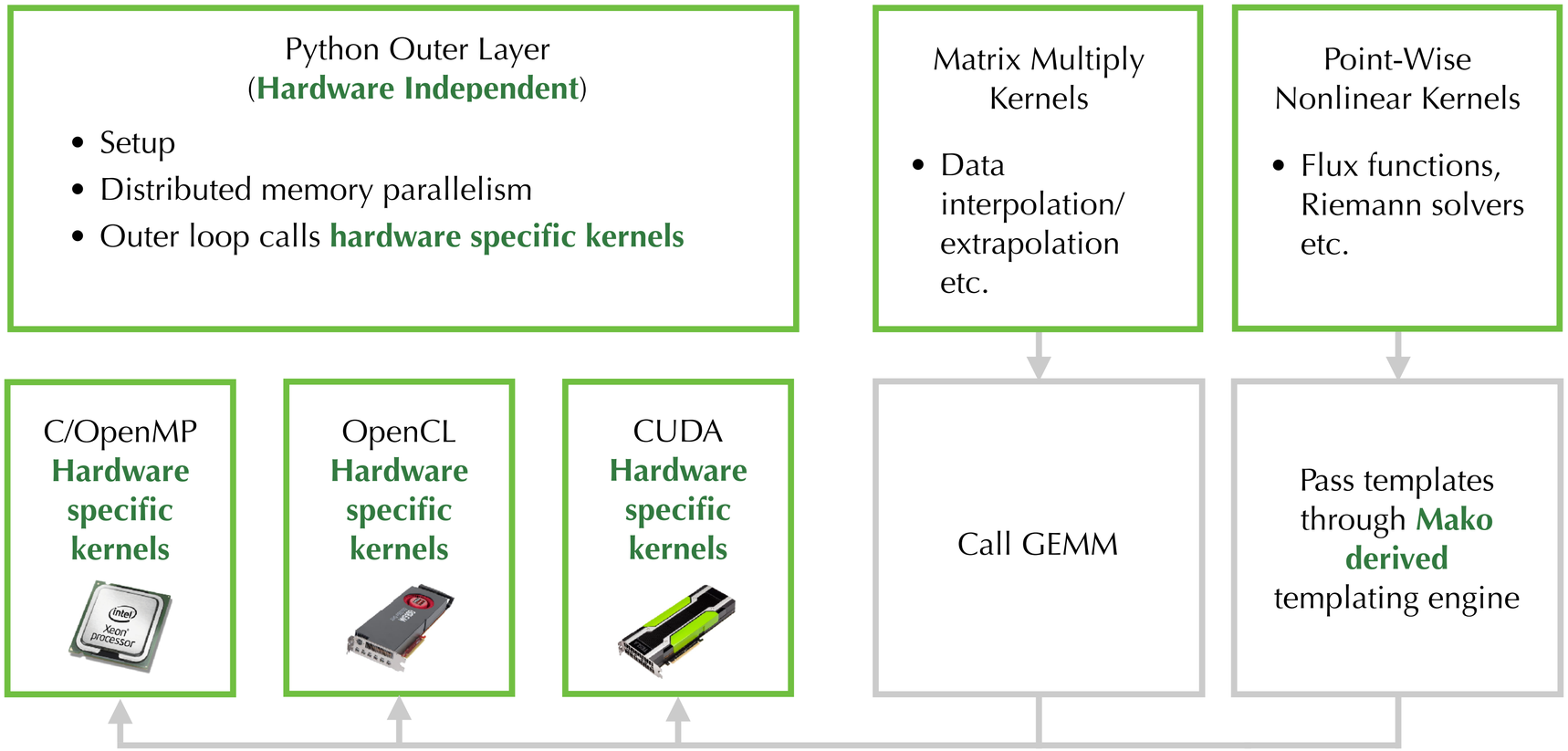
PyFR

- For the point-wise nonlinear kernels we use a DSL.



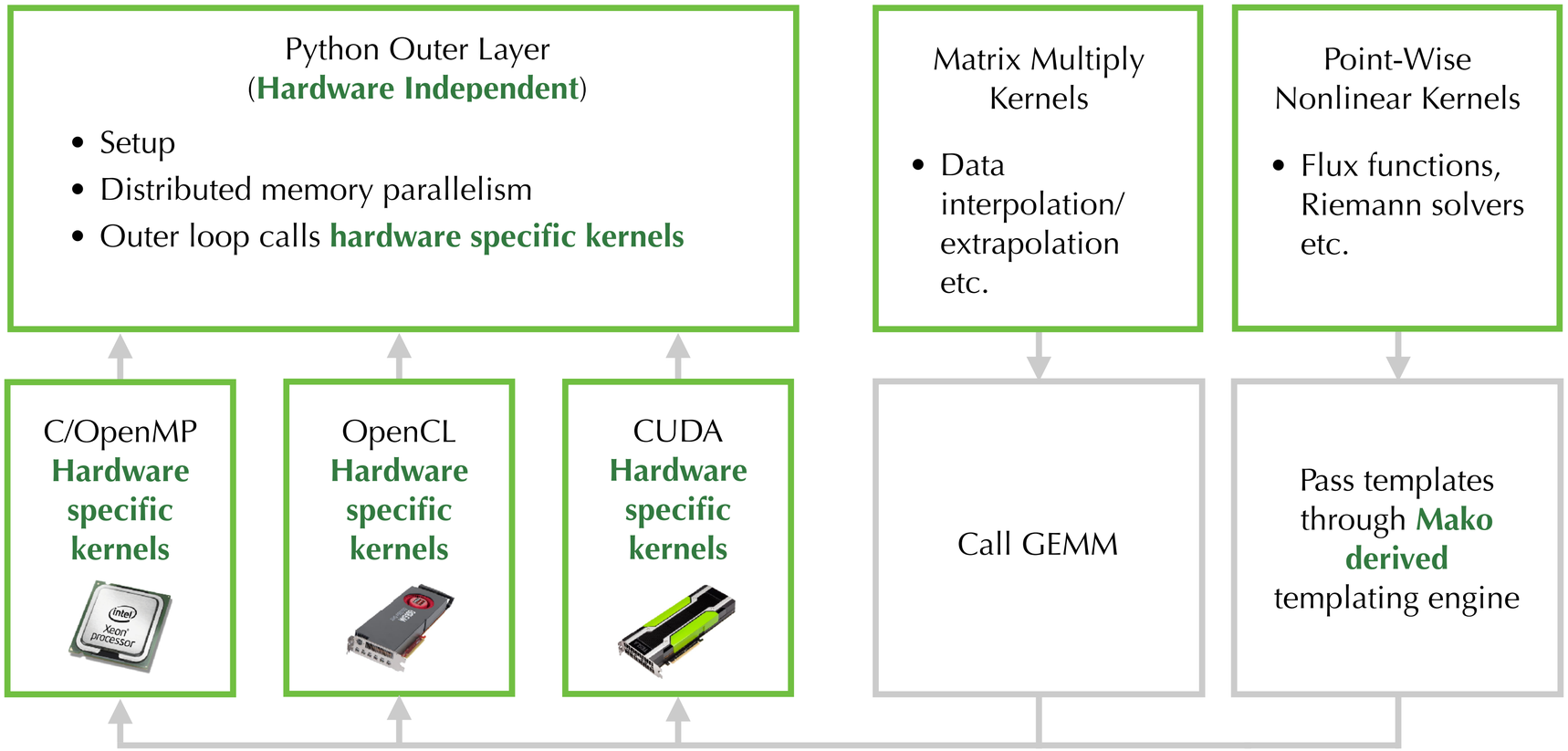
PyFR

- Kernels are generated and compiled at start-up.



PyFR

- Which may then be called by the outer layer.



PyFR

- An example.

PyFR-Mako

```
<%namespace module='pyfr.backends.base.makoutil' name='pyfr'/>

<%pyfr:macro name='inviscid_flux' params='s, f, p, v'>
    fpdtype_t invrho = 1.0/s[0];
    fpdtype_t E = s[${nvars - 1}];
    // Compute the velocities
    fpdtype_t rhov[${ndims}];
% for i in range(ndims):
    rhov[${i}] = s[${i + 1}];
    v[${i}] = invrho*rhov[${i}];
% endfor
    // Compute the pressure
    p = ${c['gamma'] - 1}*(E - 0.5*invrho*${pyfr.dot('rhov[{i}]', i=ndims)});
    // Density and energy fluxes
% for i in range(ndims):
    f[${i}][0] = rhov[${i}];
    f[${i}][${nvars - 1}] = (E + p)*v[${i}];
% endfor
    // Momentum fluxes
% for i, j in pyfr.ndrange(ndims, ndims):
    f[${i}][${j + 1}] = rhov[${i}]*v[${j}]${' + p' if i == j else ''};
% endfor
</%pyfr:macro>
```

CUDA
C/OpenMP
OpenCL

PyFR

- An example.

PyFR-Mako

```
<%namespace module='pyfr.backends.base.makoutil' name='pyfr'/>

<%pyfr:macro name='inviscid_flux' params='s, f, p, v'>
    fpdtype_t invrho = 1.0/s[0];
    fpdtype_t E = s[${nvars - 1}];
    // Compute the velocities
    fpdtype_t rhov[${ndims}];
% for i in range(ndims):
    rhov[${i}] = s[${i + 1}];
    v[${i}] = invrho*rhov[${i}];
% endfor
    // Compute the pressure
    p = ${c['gamma'] - 1}*(E - 0.5*invrho*${pyfr.dot('rhov[{i}]', i=ndims)});
    // Density and energy fluxes
% for i in range(ndims):
    f[${i}][0] = rhov[${i}];
    f[${i}][${nvars - 1}] = (E + p)*v[${i}];
% endfor
    // Momentum fluxes
% for i, j in pyfr.ndrange(ndims, ndims):
    f[${i}][${j + 1}] = rhov[${i}]*v[${j}]${' + p' if i == j else ''};
% endfor
</%pyfr:macro>
```

CUDA

```
//AoSoA macros
#define SOA_SZ 32
#define SOA_IX(a, v, nv) ((((a) / SOA_SZ)*(nv) + (v))*SOA_SZ + (a) %
SOA_SZ)

// Typedefs
typedef double fpdtype_t;


__global__ void tflux(int _ny, int _nx, fpdtype_t* __restrict__ f_v, int ldf,
const fpdtype_t* __restrict__ smats_v, int ldsmats, const fpdtype_t*
__restrict__ u_v, int ldu)
            {
                int _x = blockIdx.x*blockDim.x + threadIdx.x;int _y =
blockIdx.y*blockDim.y + threadIdx.y;
                #define X_IDX (_x)
                #define X_IDX_AOSOA(v, nv) SOA_IX(X_IDX, v, nv)
                if (_x < _nx && _y < _ny)
                {

    // Compute the flux
    fpdtype_t ftemp[2][4];
    fpdtype_t p, v[2];
    {
```

PyFR

- Abstracts **data layout**.

PyFR-Mako

```
<%namespace module='pyfr.backends.base.makoutil' name='pyfr'/>

<%pyfr:macro name='inviscid_flux' params='s, f, p, v'>
    fpdtype_t invrho = 1.0/s[0];
    fpdtype_t E = s[${nvars - 1}];
    // Compute the velocities
    fpdtype_t rhov[${ndims}];
% for i in range(ndims):
    rhov[${i}] = s[${i + 1}];
    v[${i}] = invrho*rhov[${i}];
% endfor
    // Compute the pressure
    p = ${c['gamma'] - 1}*(E - 0.5*invrho*${pyfr.dot('rhov[{i}]', i=ndims)});
    // Density and energy fluxes
% for i in range(ndims):
    f[${i}][0] = rhov[${i}];
    f[${i}][${nvars - 1}] = (E + p)*v[${i}];
% endfor
    // Momentum fluxes
% for i, j in pyfr.ndrange(ndims, ndims):
    f[${i}][${j + 1}] = rhov[${i}]*v[${j}]${' + p' if i == j else ''};
% endfor
</%pyfr:macro>
```

CUDA

```
blockIdx.y*blockDim.y + threadIdx.y;
                #define X_IDX (_x)
                #define X_IDX_AOSOA(v, nv) SOA_IX(X_IDX, v, nv)
                if (_x < _nx && _y < _ny)
                {

    // Compute the flux
    fpdtype_t ftemp[2][4];
    fpdtype_t p, v[2];
    {

    fpdtype_t invrho_ = 1.0/u_v[ldu*_y + X_IDX_AOSOA(0, 4)];
    fpdtype_t E_ = u_v[ldu*_y + X_IDX_AOSOA(3, 4)];

    // Compute the velocities
    fpdtype_t rhov_[2];
    rhov_[0] = u_v[ldu*_y + X_IDX_AOSOA(1, 4)];
    v[0] = invrho_*rhov_[0];
    rhov_[1] = u_v[ldu*_y + X_IDX_AOSOA(2, 4)];
    v[1] = invrho_*rhov_[1];

    // Compute the pressure
    p = 0.4*(E_ - 0.5*invrho_*((rhov_[0])*(rhov_[0]) + (rhov_[1])*(rhov_[1])));

    // Density and energy fluxes
```

PyFR

- Templates based on **runtime parameters**.

PyFR-Mako

```
<%namespace module='pyfr.backends.base.makoutil' name='pyfr'/>

<%pyfr:macro name='inviscid_flux' params='s, f, p, v'>
    fpdtype_t invrho = 1.0/s[0];
    fpdtype_t E = s[${nvars - 1}];
    // Compute the velocities
    fpdtype_t rhov[${ndims}];
% for i in range(ndims):
    rhov[${i}] = s[${i + 1}];
    v[${i}] = invrho*rhov[${i}];
% endfor
    // Compute the pressure
    p = ${c['gamma'] - 1}*(E - 0.5*invrho*${pyfr.dot('rhov[{i}]', i=ndims)});
    // Density and energy fluxes
% for i in range(ndims):
    f[${i}][0] = rhov[${i}];
    f[${i}][${nvars - 1}] = (E + p)*v[${i}];
% endfor
    // Momentum fluxes
% for i, j in pyfr.ndrange(ndims, ndims):
    f[${i}][${j + 1}] = rhov[${i}]*v[${j}]${' + p' if i == j else ''};
% endfor
</%pyfr:macro>
```

CUDA

```
                {

// Compute the flux
fpdtype_t ftemp[2][4];
fpdtype_t p, v[2];
{

fpdtype_t invrho_ = 1.0/u_v[ldu*_y + X_IDX_AOSOA(0, 4)];
fpdtype_t E_ = u_v[ldu*_y + X_IDX_AOSOA(3, 4)];

// Compute the velocities
fpdtype_t rhov_[2];
rhov_[0] = u_v[ldu*_y + X_IDX_AOSOA(1, 4)];
v[0] = invrho_*rhov_[0];
rhov_[1] = u_v[ldu*_y + X_IDX_AOSOA(2, 4)];
v[1] = invrho_*rhov_[1];

// Compute the pressure
p = 0.4*(E_ - 0.5*invrho_*((rhov_[0])*(rhov_[0]) + (rhov_[1])*(rhov_[1])));

// Density and energy fluxes
ftemp[0][0] = rhov_[0];
ftemp[0][3] = (E_ + p)*v[0];
ftemp[1][0] = rhov_[1];
```

PyFR

- Can also use **${Python}** to generate code.

PyFR-Mako

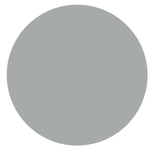
```
<%namespace module='pyfr.backends.base.makoutil' name='pyfr'/>

<%pyfr:macro name='inviscid_flux' params='s, f, p, v'>
    fpdtype_t invrho = 1.0/s[0];
    fpdtype_t E = s[${nvars - 1}];
    // Compute the velocities
    fpdtype_t rhov[${ndims}];
% for i in range(ndims):
    rhov[${i}] = s[${i + 1}];
    v[${i}] = invrho*rhov[${i}];
% endfor
    // Compute the pressure
    p = ${c['gamma'] - 1}*(E - 0.5*invrho*${pyfr.dot('rhov[{i}]', i=ndims)});
    // Density and energy fluxes
% for i in range(ndims):
    f[${i}][0] = rhov[${i}];
    f[${i}][${nvars - 1}] = (E + p)*v[${i}];
% endfor
    // Momentum fluxes
% for i, j in pyfr.ndrange(ndims, ndims):
    f[${i}][${j + 1}] = rhov[${i}]*v[${j}]${' + p' if i == j else ''};
% endfor
</%pyfr:macro>
```

CUDA

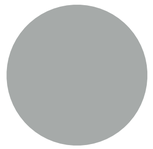
```
    // Compute the pressure
    p = 0.4*(E_ - 0.5*invrho_*((rhov_[0])*(rhov_[0]) + (rhov_[1])*(rhov_[1])));

    // Density and energy fluxes
    ftemp[0][0] = rhov_[0];
    ftemp[0][3] = (E_ + p)*v[0];
    ftemp[1][0] = rhov_[1];
    ftemp[1][3] = (E_ + p)*v[1];

    // Momentum fluxes
    ftemp[0][1] = rhov_[0]*v[0] + p;
    ftemp[0][2] = rhov_[0]*v[1];
    ftemp[1][1] = rhov_[1]*v[0];
    ftemp[1][2] = rhov_[1]*v[1] + p;

};

    // Transform the fluxes
    f_v[(0*_ny + _y)*ldf + X_IDX_AOSOA(0, 4)] = smats_v[(0*_ny +
_y)*ldsmats + X_IDX_AOSOA(0, 2)]*ftemp[0][0] + smats_v[(0*_ny +
_y)*ldsmats + X_IDX_AOSOA(1, 2)]*ftemp[1][0];
    f_v[(0*_ny + _y)*ldf + X_IDX_AOSOA(1, 4)] = smats_v[(0*_ny +
_y)*ldsmats + X_IDX_AOSOA(0, 2)]*ftemp[0][1] + smats_v[(0*_ny +
_y)*ldsmats + X_IDX_AOSOA(1, 2)]*ftemp[1][1];
    f_v[(0*_ny + _y)*ldf + X_IDX_AOSOA(2, 4)] = smats_v[(0*_ny +
```

PyFR

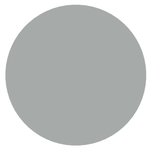
- Architecture enables PyFR to be **performance portable** across a range of platforms with **complete feature parity**.
- Can also **mix and match backends** across MPI ranks enabling **heterogeneous computing**.



Motivation

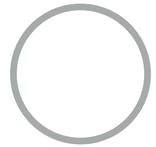


High-Order Methods



PyFR

OpenCL Backend

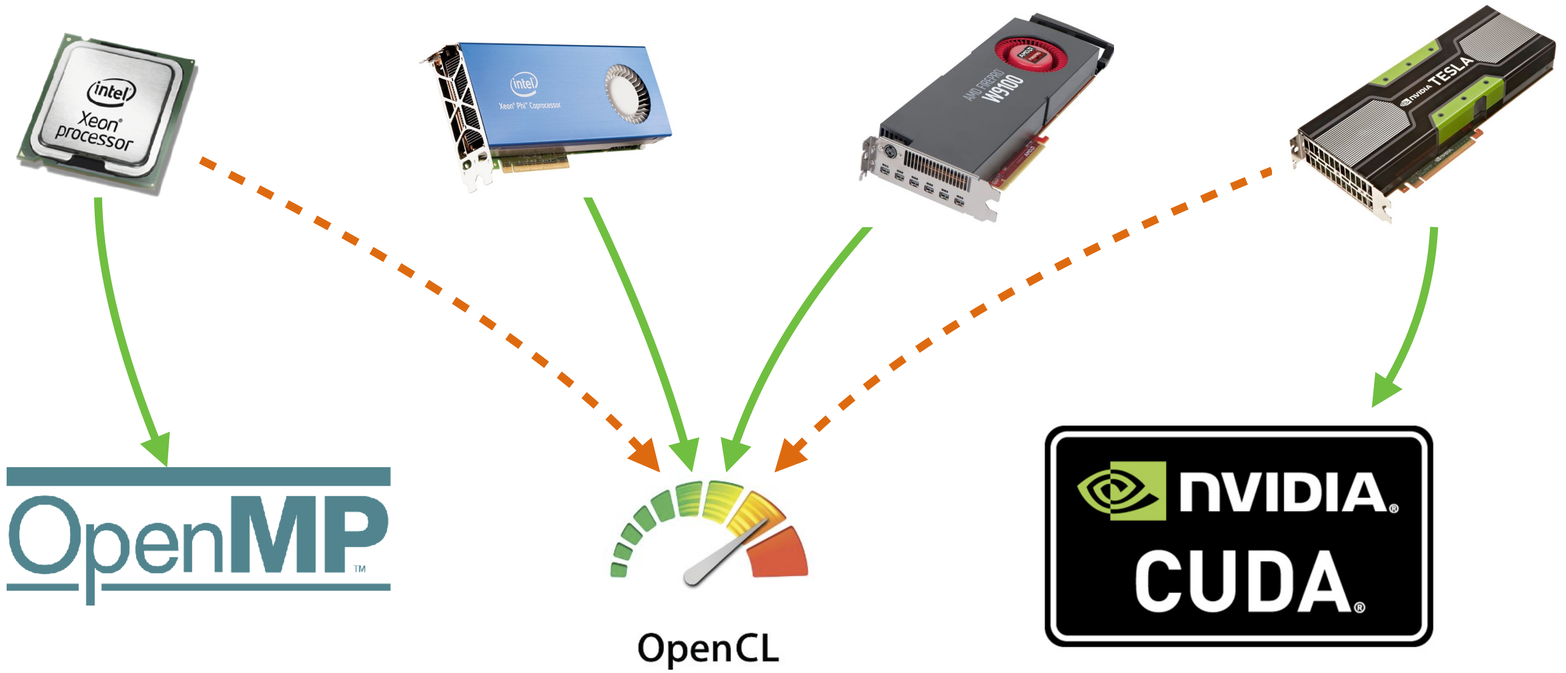


Future

OpenCL Backend

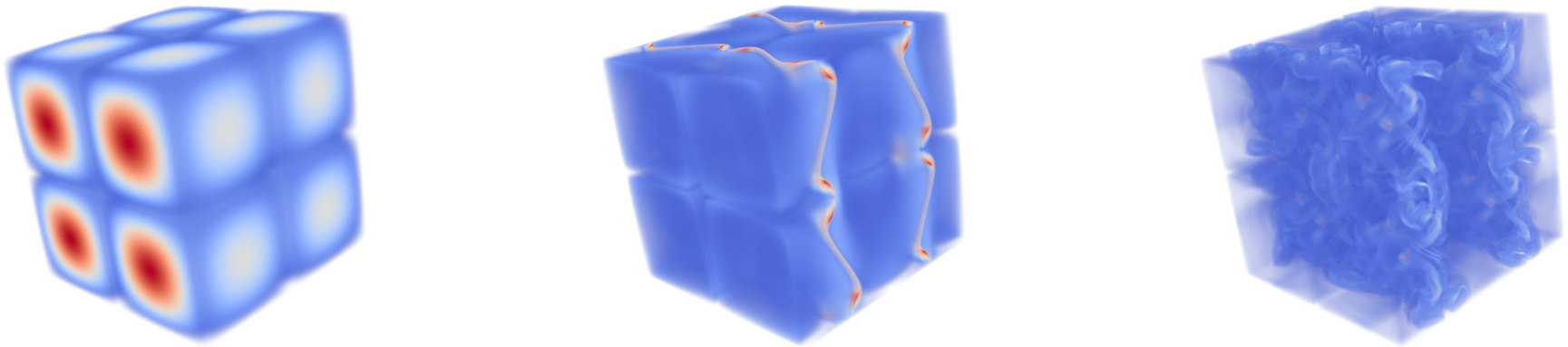
- PyFR's OpenCL backend was written in 2014.
- Uses the **PyOpenCL** wrappers by Andreas Kloeckner.
  - As OpenCL is a C API it is **easy to wrap**.
- The code generation model also fits well with PyFR.

OpenCL Backend



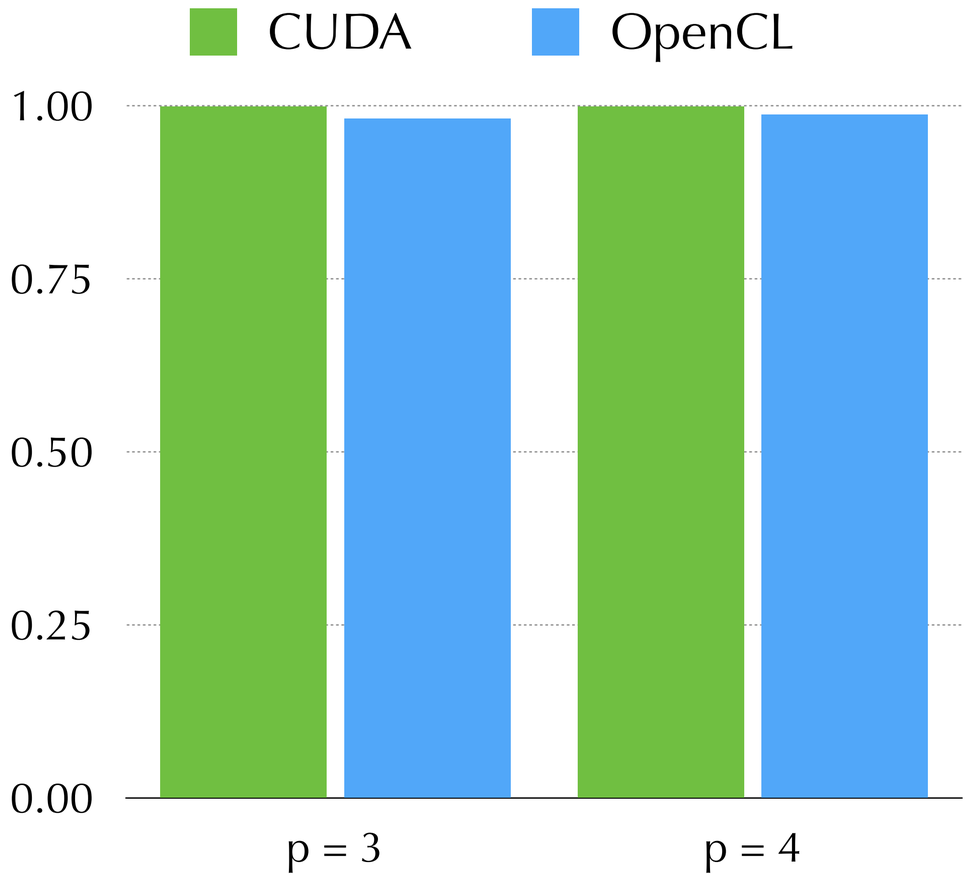
OpenCL Backend

- Performance in **bandwidth-bound scenarios** is usually on a par with ‘native’ approaches.
- Consider breakdown of a Taylor–Green vortex.



OpenCL Backend

- Normalised run-time on an NVIDIA V100 GPU.
- OpenCL is **slightly faster** at both $p = 3$ and $p = 4$.



OpenCL Backend

- Limitation #1: Lack of performance primitives.
- Getting a substantial fraction of peak FLOPS for GEMM requires **hardware specific assembly code**.
- These routines are usually **provided by vendors** in the form of BLAS libraries.

OpenCL Backend

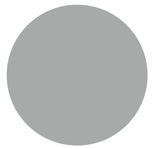
- However, for OpenCL we have to call out to generic BLAS libraries; originally **clBLAS** and since 2018 **CLBlast**.
- Although both employ auto-tuning their performance is not competitive with **Intel MKL or NVIDIA cuBLAS**.
  - Unable to push past 40–50% of peak FLOP/s.

OpenCL Backend

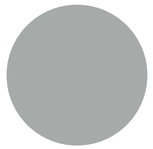
- Limitation #2: No MPI interoperability.
- With CUDA it is possible to pass device pointers to MPI.
- Under the right conditions it can **improve strong scaling by ~10-15%**.

OpenCL Backend

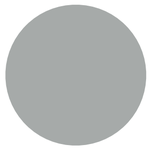
- Limitation #3: Implementation quality.
- Observed **incorrect results** with Mesa on AMD Fiji GPUs and the Intel Graphics Compute Runtime on Gen9 GPUs.



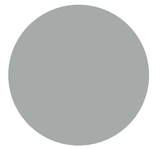
Motivation



High-Order Methods



PyFR



OpenCL Backend

Future

Future Work

- Currently investigating how to support upcoming Intel GPUs.
- As we're a Python application **SYCL/DPC++ is not viable**.
- Initial support will likely be through our OpenCL backend.

Future Work

- In order to better support AMD GPUs we're looking to add a **HIP backend** into PyFR.
- This will enable us to avoid some of the issues we've encountered with the OpenCL backend.