PyFR: An Open Source Python Framework for High-Order CFD on Many-Core Platforms

F. Witherden, A. Farrington, P. E. Vincent

Department of Aeronautics, Imperial College London, SW7 2AZ, UK

Introduction

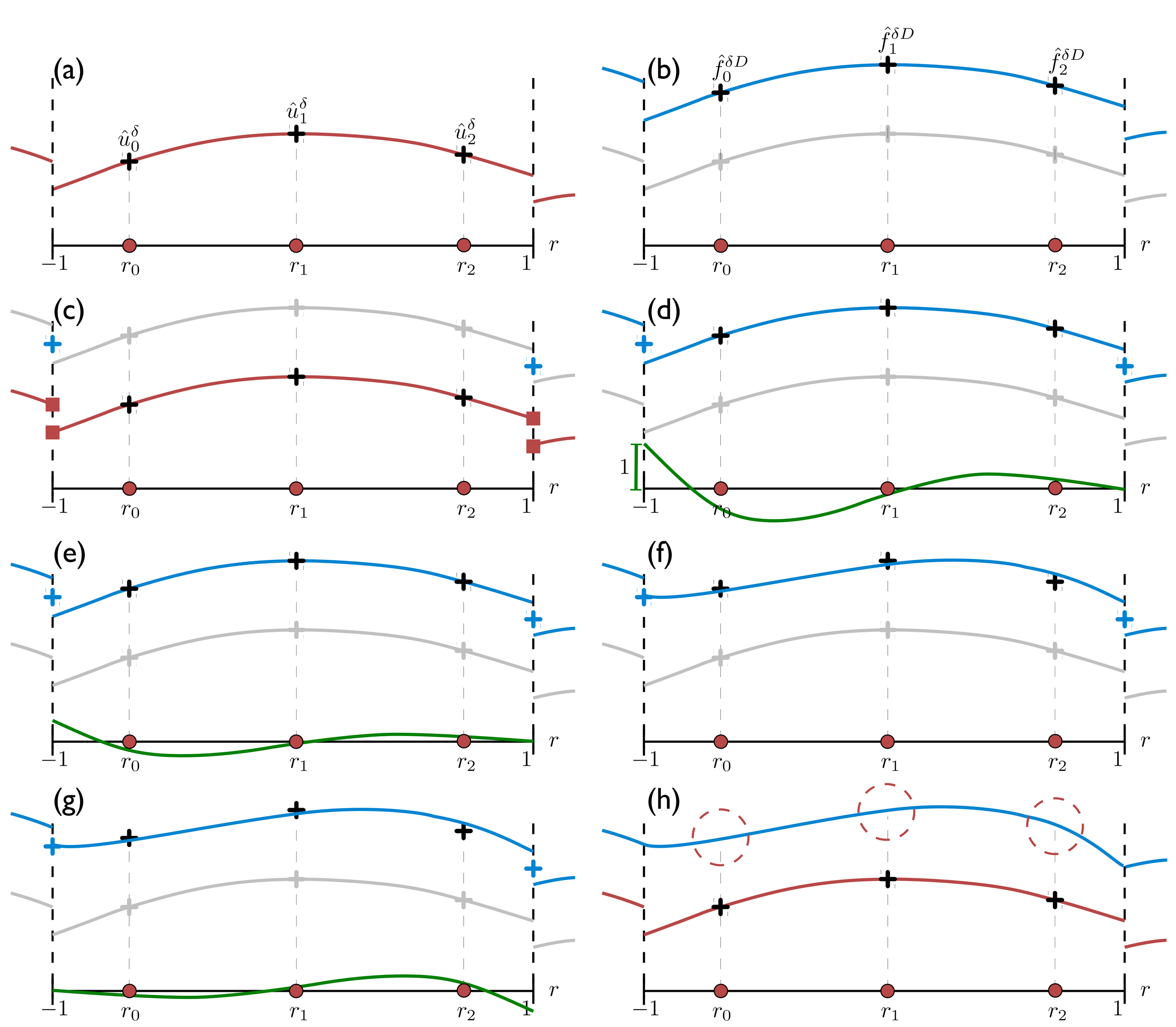
Theoretical studies and numerical experiments suggest that high-order methods for unstructured grids can solve hitherto intractable fluid flow problems in the vicinity of complex geometrical configurations. The flux reconstruction (FR) approach, developed by Huynh [1], is a simple yet efficient high-order scheme that is particularly amenable to the requirements of modern hardware architectures—including GPUs. Using the FR approach it is possible to unify various popular high-order methods such as nodal discontinuous Galerkin (DG) and spectral difference schemes. In 2011 Vincent et al. identified an infinite range of FR schemes which are linearly stable [2].

The FR Approach

- Consider the 1D scalar conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial r} = 0$$

  on a domain [a,b].
- Divide the domain up into elements (of any length) and apply a linear transformation to these elements into standard elements in [-1,1]. Inside each standard element we store the approximate solution at a set of $k + 1$ points termed solution points.
- The FR approach then proceeds as follows.
  (a) Construct a degree k Lagrange interpolating polynomial through the solution points.
  (b) Evaluate the flux at each solution point and form a second degree k interpolating polynomial.
  (c) Evaluate the solution point polynomial at $r = -1$ and $r = 1$ and solve the Riemann problem resulting at each element interface to yield a common flux.
  (d) Define a left correction function, $g_L(r)$, such that $g_L(-1) = 1$ and $g_L(1) = 0$.
  (e) Scale this function by the jump between the common and interpolated fluxes.
  (f) Add this scaled correction function to the flux polynomial.
  (g) Repeat steps (d)–(f) for the right hand side with $g_R(r) = g_L(-r)$.
  (h) Take the derivative of the corrected flux polynomial at each solution point. Once suitably transformed this flux derivative can be fed into a time marching algorithm, e.g. RK4.



Implementation

- Operations in FR reduce down to evaluating
  - polynomials and their derivatives;
  - point-wise functions, e.g. the flux;
  - common fluxes at interfaces.
- Majority of operations are therefore element local.
- Possible to cast polynomials evaluations as matrix multiplications which are extremely efficient on today's many-core computing platforms.

PyFR

- Capable of solving the compressible Euler/Navier-Stokes equations on unstructured grids of tensor-product elements (quads and hexes).
- Written in Python and utilizes a backend architecture to target multiple platforms
  - backends currently exist for C/OpenMP and CUDA (PyCUDA);
- Matrix multiplications are offloaded to BLAS.
- Point-wise functions and other bespoke operations are templated using Mako:
  - code is generated, compiled, linked and loaded at runtime;
  - allows PyFR to readily exploit platform-specific instruction sets, e.g. XOP/AVX/….
- Multiple nodes supported through MPI
  - accomplished via the mpi4py library.
- Current statistics:
  - 4000 lines of Python;
  - 800 lines of Mako/CUDA;
  - 800 lines of Mako/C.
- Released under a three-clause 'new style' BSD license.

Sample Code Fragments

- Use of SymPy's symbolic manipulation capabilities for constructing the various polynomial operators.

$$\eta_k \in \left\{0, \frac{k}{k+1}, \frac{k+1}{k}, \dots\right\}$$

$$g_R' = \frac{1}{2}\frac{d}{dr}\left[L_k + \frac{\eta_k L_{k-1} + L_{k+1}}{1 + \eta_k}\right]$$

Theory

```python
def diff_vcjh_correctionfn(k, eta, sym):
    # Expand shorthand forms of eta_k for common schemes
    etacommon = dict(dg='0', sd='k/(k+1)', hu='(k+1)/k')

    eta_k = sy.S(etacommon.get(eta, eta), locals=dict(k=k))

    lkm1, lk, lkp1 = [sy.legendre_poly(m, sym) for m in [k-1, k, k+1]]

    # Correction function derivatives, Eq. 3.46 and 3.47
    diffgr = (sy.S(1)/2 * (lk + (eta_k*lkm1 + lkp1)/(1 + eta_k))).diff()
    diffgl = -diffgr.subs(sym, -sym)

    return diffgl, diffgr
```

Implementation

- Templated kernel generation.

```
<%include file='idx_of.cu.mak' />

__global__ void
negdivconf(int nupts, int neles,
           ${dtype}* __restrict__ tdivtconf,
           const ${dtype}* __restrict__ rcpdjac,
           int ldt, int ldr)
{
    int eidx = blockIdx.x * blockDim.x + threadIdx.x;

    if (eidx < neles)
    {
        for (int uidx = 0; uidx < nupts; ++uidx)
        {
            ${dtype} s = -rcpdjac[IDX_OF(uidx, eidx, ldr)];

        % for i in range(nvars):
            tdivtconf[U_IDX_OF(uidx, eidx, ${i}, neles, ldt)] *= s;
        % endfor
        }
    }
}
```

Mako template

```
#ifndef _PYFR_IDX_OF
#define _PYFR_IDX_OF

#define IDX_OF(i, j, ldim) ((i)*(ldim) + (j))

#define U_IDX_OF(upt, ele, var, nele, ldim) \
    IDX_OF(upt, nele*var + ele, ldim)

#endif // _PYFR_IDX_OF

__global__ void
negdivconf(int nupts, int neles,
           double* __restrict__ tdivtconf,
           const double* __restrict__ rcpdjac,
           int ldt, int ldr)
{
    int eidx = blockIdx.x * blockDim.x + threadIdx.x;

    if (eidx < neles)
    {
        for (int uidx = 0; uidx < nupts; ++uidx)
        {
            double s = -rcpdjac[IDX_OF(uidx, eidx, ldr)];

            tdivtconf[U_IDX_OF(uidx, eidx, 0, neles, ldt)] *= s;
            tdivtconf[U_IDX_OF(uidx, eidx, 1, neles, ldt)] *= s;
            tdivtconf[U_IDX_OF(uidx, eidx, 2, neles, ldt)] *= s;
            tdivtconf[U_IDX_OF(uidx, eidx, 3, neles, ldt)] *= s;
            tdivtconf[U_IDX_OF(uidx, eidx, 4, neles, ldt)] *= s;
        }
    }
}
```

Generated CUDA

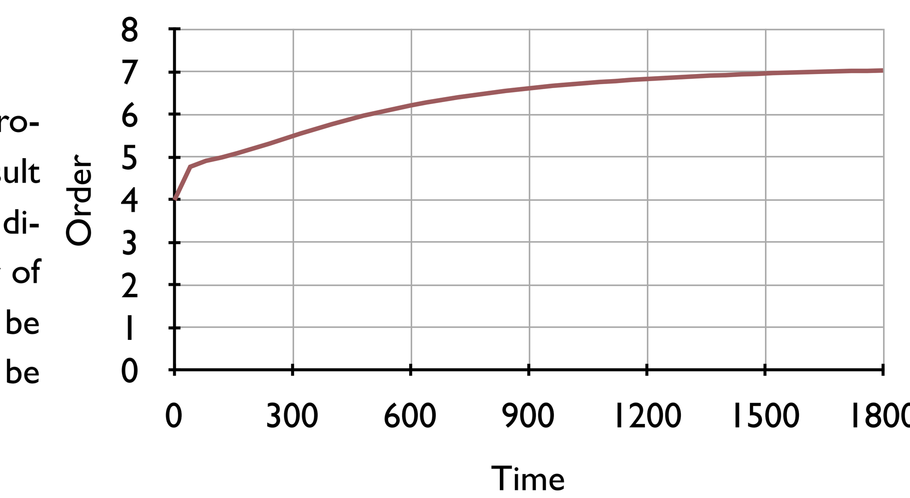
Future Plans

- Support for simplex elements including triangles, prisms and tetrahedra.
- Implement adaptive time-stepping algorithms and the viability of semi-implicit schemes.
- An OpenCL backend to allow PyFR to target AMD S10000 and Intel Xeon Phi accelerators.
- Investigate means of incorporating shock capturing into FR.

Results

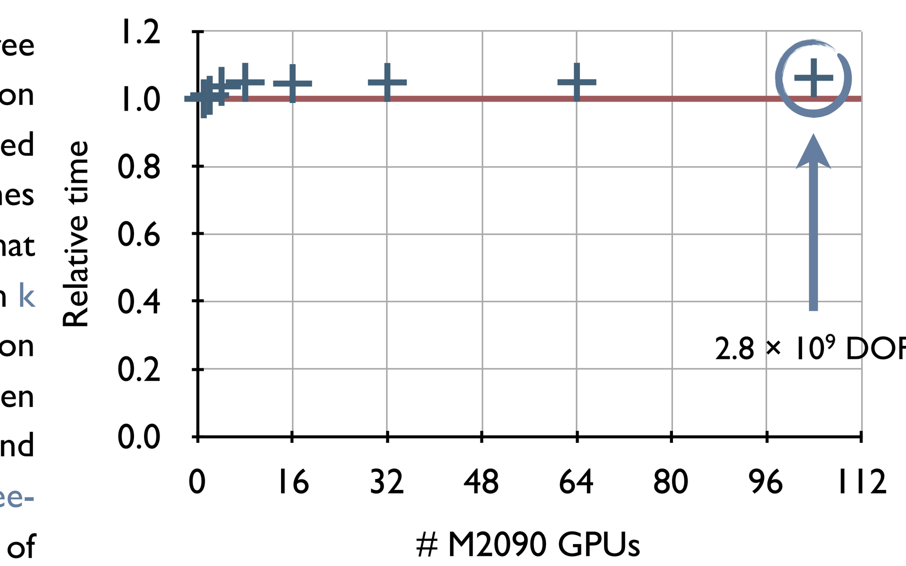
Super Accuracy

Using PyFR we have been able to reproduce the '$2k + 1$' order of accuracy result described in [3] for a non-linear two dimensional Euler test problem. An order of accuracy plot for the case of $k = 3$ can be seen to the right where the order can be clearly seen to asymptote to 7.



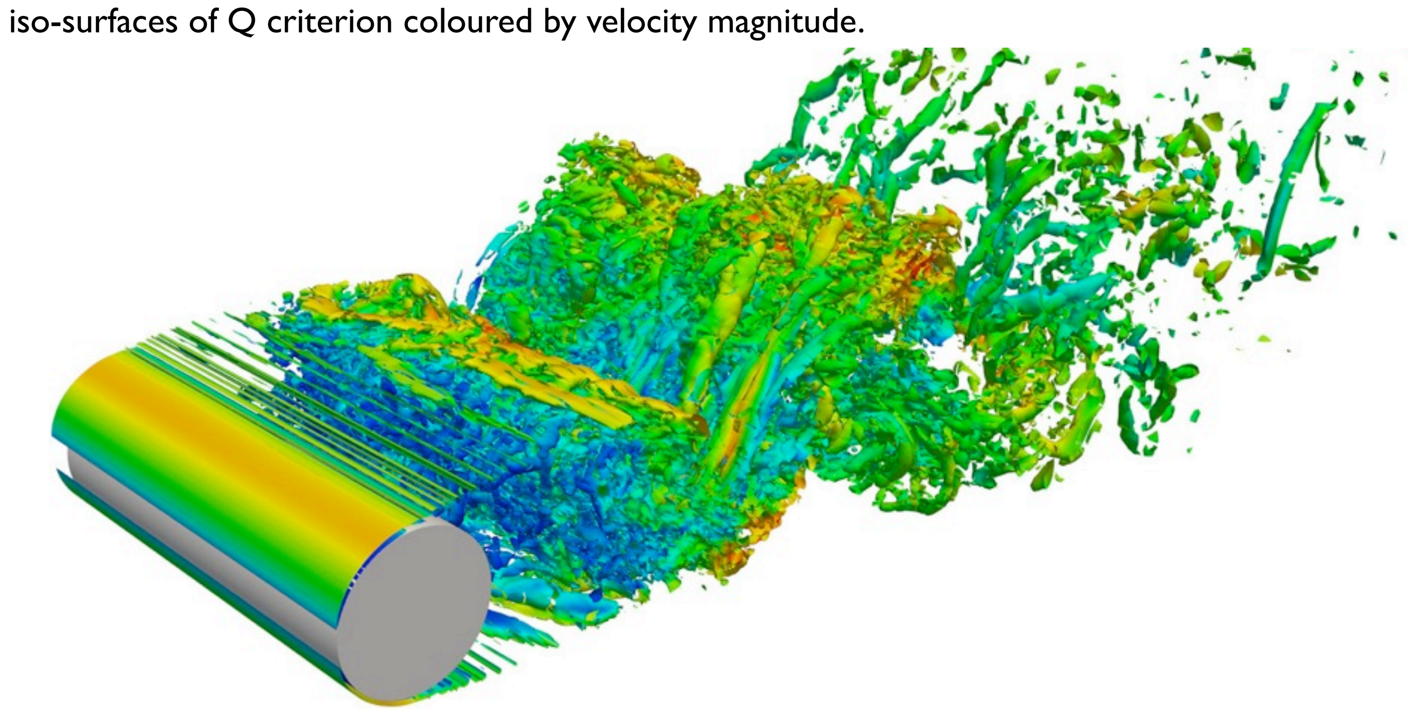
Weak Scalability

The weak scalability of PyFR for the three dimensional Navier-Stokes equations on hexahedral mesh elements was evaluated on the Emerald GPU cluster. Meshes were generated and partitioned such that each M2090 GPU was ~90% loaded with $k = 3$ solution polynomials. The simulation time, relative to a single GPU, can be seen on the right. Even with 104 GPUs—and almost three billion degrees of freedom—the simulation time is within 6% of the single GPU run.



Turbulent Flow over a Cylinder

Flow at $Ma = 0.2$ over a cylinder was simulated at $Re = 3900$. With $k = 4$ the simulation contained ~29 million degrees of freedom and was run on a workstation with four K20c GPUs. Shown below are iso-surfaces of Q criterion coloured by velocity magnitude.



Acknowledgements

Our work is supported by the Engineering and Physical Sciences Research Council (EPSRC) and NVIDIA. Conference participation aided by the Royal Commission for the Exhibition of 1851, the Imperial College General Trust and the Old Centralians' Trust.

References

[1] H. T. Huynh. A flux reconstruction approach to high-order schemes including discontinuous Galerkin methods. AIAA Paper 2007-4079. 2007
[2] P. E. Vincent, P. Castonguay, A. Jameson. A New Class of High-Order Energy Stable Flux Reconstruction Schemes. Journal of Scientific Computing. 2011
[3] P. E. Vincent, P. Castonguay, A. Jameson. Insights from von Neumann analysis of high-order flux reconstruction schemes. Journal of Computational Physics. 2011