EPSRC
Engineering and Physical Sciences Research Council

PyFR: An Open Source Framework for Solving Advection-Diffusion Type Problems on Streaming Architectures

http://www.pyfr.org @PyFR_Solver
F. D. Witherden, A. M. Farrington, P. E. Vincent
Department of Aeronautics, Imperial College London, SW7 2AZ, UK

NVIDIA

Introduction

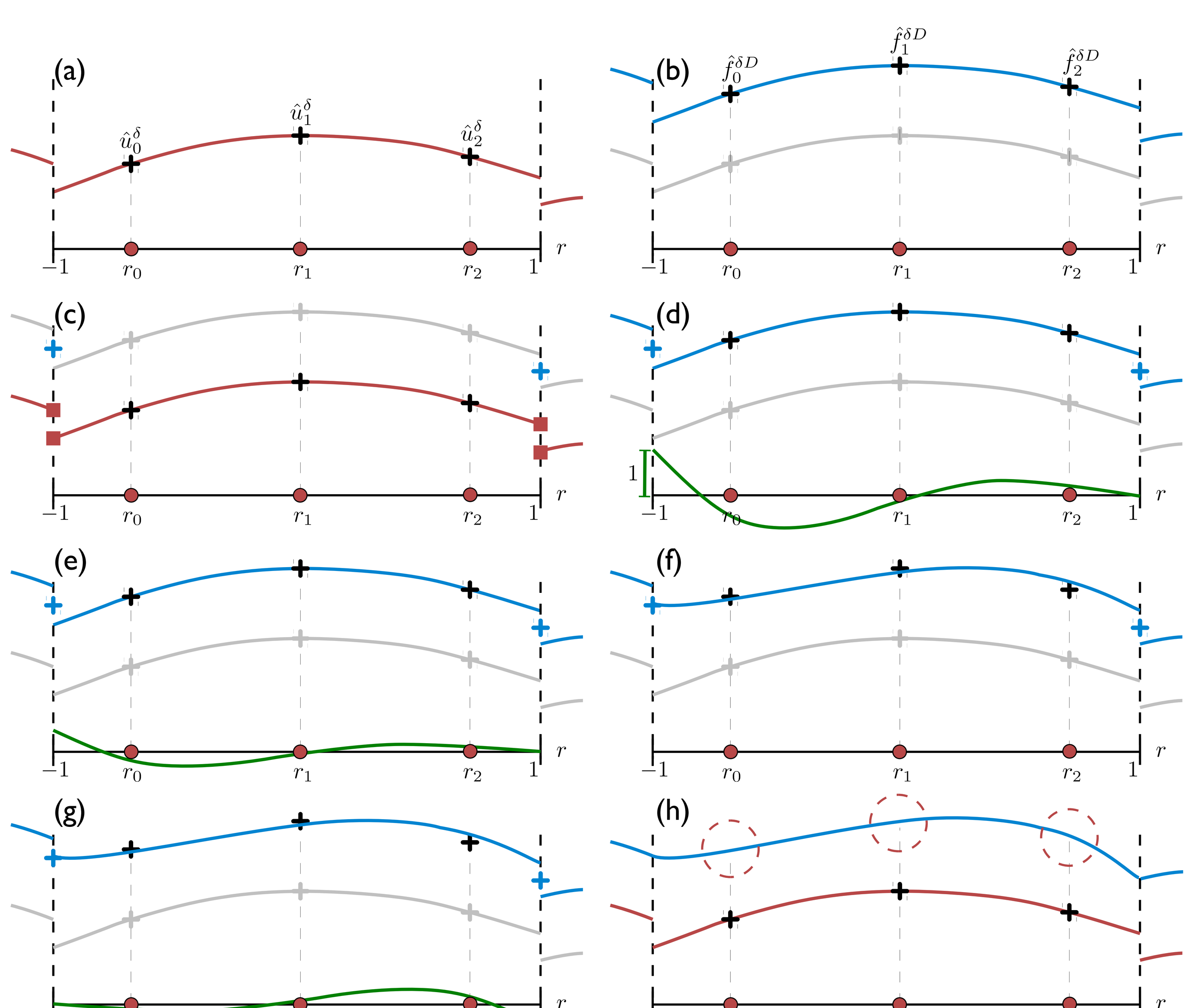
Theoretical studies and numerical experiments suggest that high-order methods for unstructured grids can solve hitherto intractable fluid flow problems in the vicinity of complex geometrical configurations. The flux reconstruction (FR) approach, developed by Huynh [1], is a simple yet efficient high-order 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.

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 quadrangles, triangles and hexahedra.
- 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 kernels implemented using our own domain specific language:
  - implemented on top of the Mako templating engine;
  - kernels are translated to either CUDA or C/OpenMP;
  - code is then compiled, linked and loaded into PyFR at runtime;
  - permits PyFR to readily exploit platform-specific instruction sets, e.g. AVX/FMA3/….
- Multiple nodes supported through MPI
  - accomplished via the mpi4py library.
- 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

- Runtime kernel generation.

```
<%pyfr:kernel name='negdivconf' ndim='2'
              tdivtconf='inout fpdtype_t[${str(nvars)}]'
              rcpdjac='in fpdtype_t'>
% for i in range(nvars):
    tdivtconf[${i}] *= -rcpdjac;
% endfor
</%pyfr:kernel>
```

Templated kernel

```c
__global__ void
negdivconf(int _ny, int _nx,
           const fpdtype_t* __restrict__ rcpdjac_v,
           int ldrcpdjac,
           fpdtype_t* __restrict__ tdivtconf_v,
           int ldtdivtconf)
{
    int _x = blockIdx.x*blockDim.x + threadIdx.x;

    for (int _y = 0; _y < _ny && _x < _nx; ++_y)
    {
        fpdtype_t rcpdjac, tdivtconf[4];

        // Load rcpdjac
        rcpdjac = rcpdjac_v[ldrcpdjac*_y + _x];

        // Load tdivtconf
        tdivtconf[0] = tdivtconf_v[ldtdivtconf*_y + _x + _nx*0];
        tdivtconf[1] = tdivtconf_v[ldtdivtconf*_y + _x + _nx*1];
        tdivtconf[2] = tdivtconf_v[ldtdivtconf*_y + _x + _nx*2];
        tdivtconf[3] = tdivtconf_v[ldtdivtconf*_y + _x + _nx*3];

        tdivtconf[0] *= -rcpdjac;
        tdivtconf[1] *= -rcpdjac;
        tdivtconf[2] *= -rcpdjac;
        tdivtconf[3] *= -rcpdjac;

        // Store tdivtconf
        tdivtconf_v[ldtdivtconf*_y + _x + _nx*0] = tdivtconf[0];
        tdivtconf_v[ldtdivtconf*_y + _x + _nx*1] = tdivtconf[1];
        tdivtconf_v[ldtdivtconf*_y + _x + _nx*2] = tdivtconf[2];
        tdivtconf_v[ldtdivtconf*_y + _x + _nx*3] = tdivtconf[3];
    }
}
```

Generated CUDA

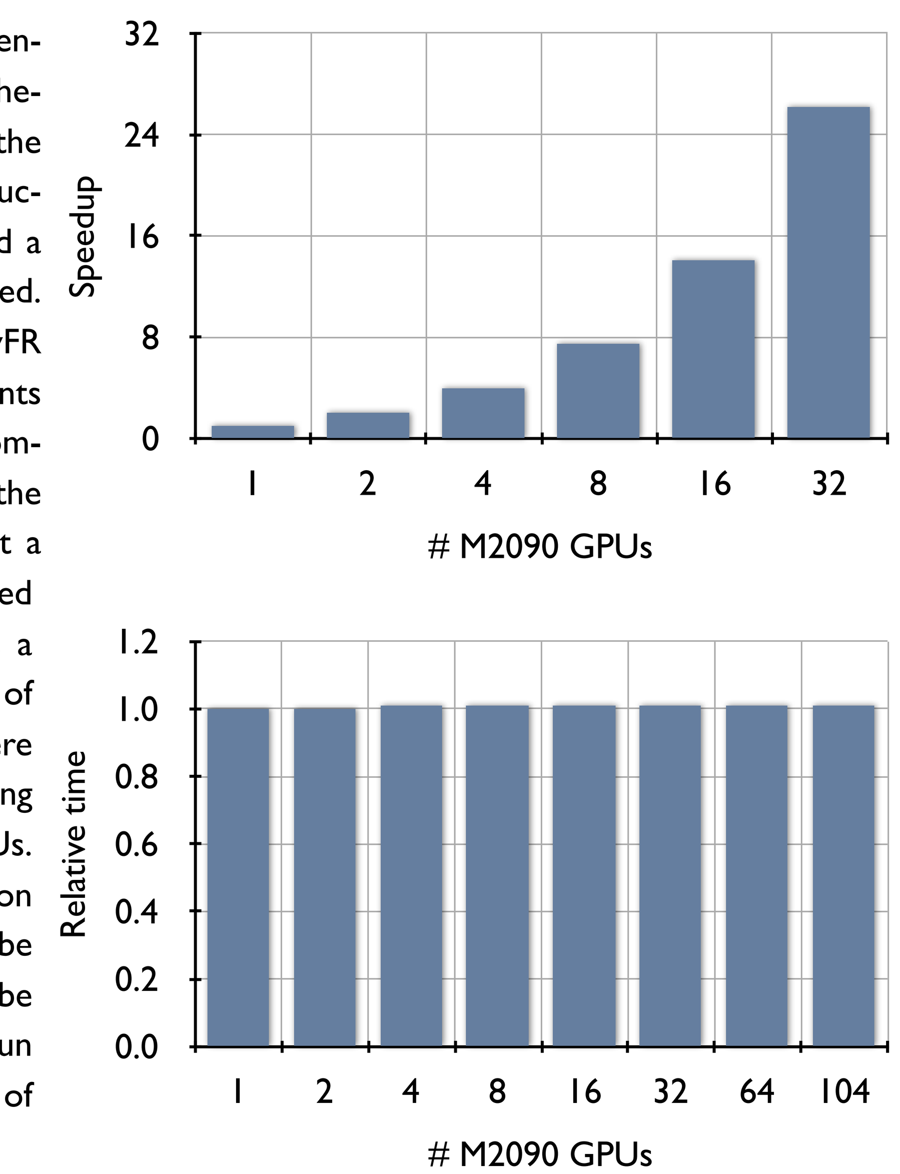
Future Plans

- Support for additional element types including prisms and tetrahedra.
- Implement adaptive time-stepping algorithms and the viability of semi-implicit schemes.
- An OpenCL backend to allow PyFR to target the AMD S10000.

Results

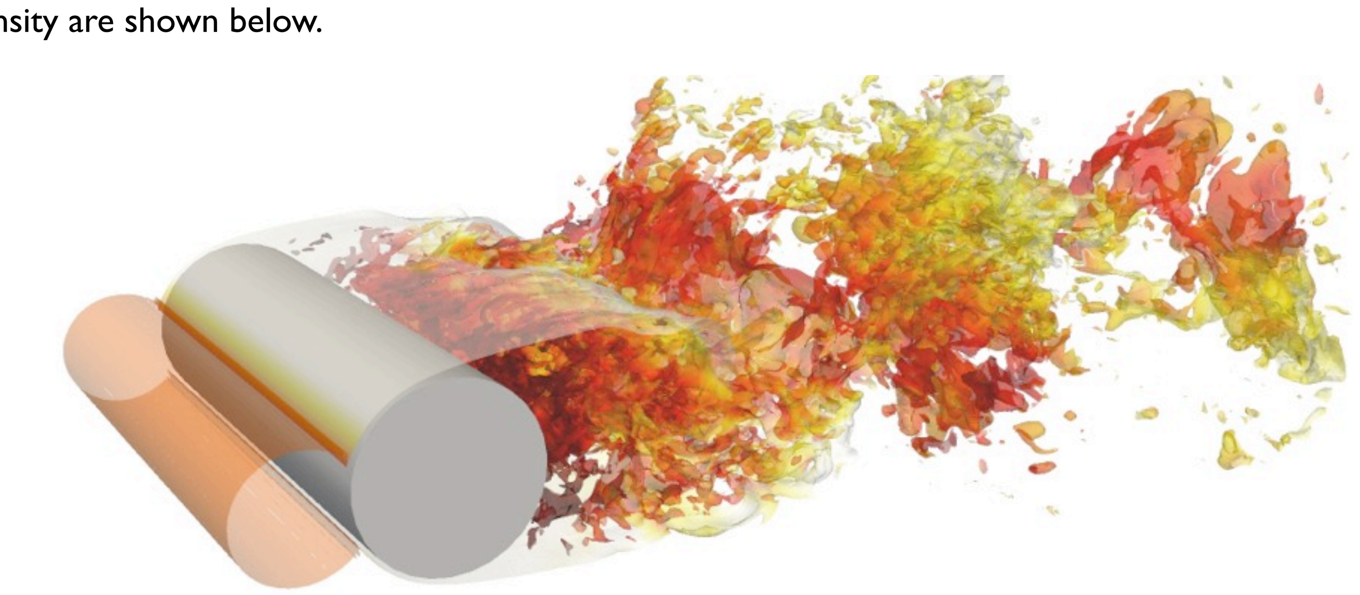
Scalability

The scalability of PyFR for the three dimensional Navier-Stokes equations on hexahedral mesh elements was evaluated on the Emerald GPU cluster with $k = 3$. A structured mesh was generated to ~90% load a single NVIDIA M2090 with ECC enabled. To evaluate the strong scalability of PyFR this mesh was partitioned into N segments of equal size. The resulting speedups compared to a single M2090 can be seen on the right. We note that with 32 GPUs that a speedup of ~26 times can be observed with each GPU being ~3% loaded. As a means of evaluating the weak scalability of the code another series of meshes were created with the number of elements being proportional to the number of GPUs. Each GPU was ~90% loaded. Simulation times—relative to a single M2090—can be seen to the right. Weak scalability can be seen to be near perfect. The 104 GPU run contained almost four billion degrees of freedom and a working set of ~485 GiB.



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. Isosurfaces of density are shown below.



Acknowledgements

The authors would like to thank the Engineering and Physical Sciences Research Council for their support via two Doctoral Training Grants and an Early Career Fellowship (EP/K027379/1). The authors would also like to thank the e-Infrastructure South Centre for Innovation for granting access to the Emerald supercomputer, and NVIDIA for donation of three K20c GPUs.

References

[1] H. T. Huynh. A flux reconstruction approach to high-order schemes including discontinuous Galerkin methods. AIAA Paper 2007-4079. 2007
[2] F. D. Witherden, A. M. Farrington, P. E. Vincent. PyFR: An Open Source Framework for Solving Advection-Diffusion Type Problems on Streaming Architectures using the Flux Reconstruction Approach. Submitted for publication in *Computer Physics Communications*. http://arxiv.org/abs/1312.1638