A Technical Overview of PyFR

F.D. Witherden
Department of Ocean Engineering, Texas A&M University
Why Go High-Order?

• 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**.
Flux Reconstruction

• Our high-order method of choice is the flux reconstruction (FR) scheme of Huynh.

• It is both unifying and capable of operating effectively on mixed unstructured grids.
PyFR

Python + Flux Reconstruction
### PyFR

**Features.**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Governing Equations</td>
<td>Compressible and incompressible Navier Stokes</td>
</tr>
<tr>
<td>Spatial Discretisation</td>
<td>Arbitrary order Flux Reconstruction on mixed unstructured grids (tris, quads, hexes, tets, prisms, and pyramids)</td>
</tr>
<tr>
<td>Temporal Discretisation</td>
<td>Adaptive explicit Runge-Kutta schemes</td>
</tr>
<tr>
<td>Precision</td>
<td>single double</td>
</tr>
<tr>
<td>Sub-grid scale models</td>
<td>None</td>
</tr>
<tr>
<td>Platforms</td>
<td>CPU and Xeon Phi clusters, NVIDIA GPU clusters, AMD GPU clusters</td>
</tr>
</tbody>
</table>
PyFR

• High level structure.

- 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.

- Call GEMM

- C/OpenMP Hardware specific kernels
- OpenCL Hardware specific kernels
- CUDA Hardware specific kernels

- Pass templates through Mako derived templating engine
PyFR

- Enables **heterogeneous computing** from a homogeneous code base.
PyFR

- PyFR can scale up to leadership class DOE machines and was shortlisted for the 2016 Gordon Bell Prize.
Implementing FR Efficiently

1. Use non-blocking communication primitives.


3. Cast key kernels as performance primitives.
Non-Blocking Communication

• Time to solution is heavily impacted by the parallel scaling of a code.

• This, in turn, is influenced by the amount of communication performed at each time step.
Non-Blocking Communication
Non-Blocking Communication

• If a code is to strong scale it is hence essential for it to overlap communication with computation.
Non-Blocking Communication

Compute A

MPI Recv

MPI Send

Compute B

Compute C

Compute D

Compute

$A$

$B$

$C$

$D$

$t$
Non-Blocking Communication

Compute A

MPI ISend

MPI IRecv

Compute C

Compute D

MPI Wait

Compute B

$\text{t}$
Non-Blocking Communication
Implementing FR Efficiently

1. Use non-blocking communication primitives.


3. Cast key kernels as performance primitives.
Data Layouts

• FR is very often a memory bandwidth bound algorithm.

• It is therefore vital that a code arranges its data in a way which enables us to extract a high fraction of peak bandwidth.
Data Layouts

- Three main layouts:
  - AoS
  - SoA
  - AoSoA($k$)
Data Layouts: AoS

```c
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($k = 2$)

```c
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 ‘Goldilocks’ solution
  - …albeit at the cost of **messy indexing**
  - …and requires **coaxing for compilers to vectorise**.
Data Layouts: AoSoA(k) Results

- FR with SoA vs FR AoSoA on an Intel KNL.

![Bar chart showing the time per DOF per RK stage in nanoseconds for different parallel configurations (p = 1, p = 2, p = 3, p = 4).]
Implementing FR Efficiently

1. Use non-blocking communication primitives.
3. Cast key kernels as performance primitives.
Performance Primitives

• On modern hardware it can be extremely difficult to extract a high percentage of peak FLOP/s in otherwise compute-bound kernels.

• To this end it is important—where possible—to cast operations in terms of performance primitives.
Performance Primitives

• Have data at • and want to interpolate to □.
Performance Primitives

• This operation can be recognised as a **matrix-vector product** (GEMV) as $u = Mv$.

• If we are working in transformed space then $M$ is the same for all elements.

• This can be recognised as a **matrix-matrix product** (GEMM) as $U = MV$. 
Performance Primitives

• Both GEMV and GEMM are performance primitives and optimised implementations are readily available from vendor BLAS libraries.

• These routines can perform an order of magnitude better than hand-rolled routines.
Performance Primitives

• In FR the operator matrix $M$ can sometimes be sparse.

• This requires use of a more specialised primitives such as those found in GiMMiK and libxsmm which account for the size/sparsity of FR operators.
Summary

• Use non-blocking communication primitives.

• Arrange data in a cache- and vectorisation-friendly manner.

• Cast key kernels as performance primitives.