PyFR and GiMMiK on Intel KNL

F.D. Witherden¹, J.S. Park², A. Heinecke³, P. Kelly², P.E. Vincent² and A. Jameson¹

¹Department of Aeronautics & Astronautics, Stanford University
²Departments of Aeronautics and Computing, Imperial College London
³Intel Corporation
Motivation: Turbulent Flows

• Interested in simulating unsteady, turbulent, flows.
The PyFR Framework

- Uses high-order flux reconstruction (FR) to solve the compressible Navier–Stokes equations on mixed unstructured grids with explicit time stepping.
The PyFR Framework

• **Performance portable** across a range of platforms.

• Finalist for the **2016 Gordon Bell Prize**.
The PyFR Framework

• Existing support for KNC based around offloading via pyMIC.
• Python outer layer.

Python Outer Layer
(Hardware Independent)

• Setup
• Distributed memory parallelism
• Outer ‘for’ loop and calls to hardware specific kernels
PyFR

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

Python Outer Layer
(Hardware Independent)

• Setup
• Distributed memory parallelism
• Outer ‘for’ loop and calls to **hardware specific kernels**
PyFR

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

<table>
<thead>
<tr>
<th>Python Outer Layer (Hardware Independent)</th>
<th>Matrix Multiply Kernels</th>
<th>Point-Wise Nonlinear Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Setup</td>
<td>• Data interpolation/ extrapolation etc.</td>
<td>• Flux functions, Riemann solvers etc.</td>
</tr>
<tr>
<td>• Distributed memory parallelism</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Outer ‘for’ loop and calls to hardware specific kernels</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PyFR

• Matrix multiplications are quite simple.

Python Outer Layer (Hardware Independent)
- Setup
- Distributed memory parallelism
- Outer ‘for’ loop and calls to hardware specific kernels

Matrix Multiply Kernels
- Data interpolation/extrapolation etc.

Point-Wise Nonlinear Kernels
- Flux functions, Riemann solvers etc.

Call GEMM
PyFR

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

Python Outer Layer
(Hardware Independent)

• Setup
• Distributed memory parallelism
• Outer ‘for’ loop and calls to hardware specific kernels

Matrix Multiply Kernels

• Data interpolation/ extrapolation etc.

Point-Wise Nonlinear Kernels

• Flux functions, Riemann solvers etc.

Call GEMM

Pass templates through Mako derived templating engine
PyFR

• Kernels are generated and compiled at start-up.

Python Outer Layer (Hardware Independent)
- Setup
- Distributed memory parallelism
- Outer ‘for’ loop and calls to 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
pyMIC Hardware specific kernels
CUDA Hardware specific kernels
OpenCL Hardware specific kernels

Pass templates through Mako derived templating engine
PyFR

- Which may then be called by the outer layer.
Matrix Multiplications in PyFR

- Multiplications are of the **block-by-panel** variety:

  \[
  \begin{align*}
  \begin{array}{c}
  \text{C} \\
  \downarrow \\
  N \\
  \end{array}
  & = 
  \begin{array}{c}
  \text{A} \\
  \end{array}
  \times 
  \begin{array}{c}
  \text{B} \\
  \end{array}
  \\
  \end{align*}
  \]

- where \( N \sim 10^5 \) with \( N \gg (M, K) \) and \( A \) is constant.
GEMM in PyFR

- On x86 S/DGEMM has **three kernels providers**.

<table>
<thead>
<tr>
<th></th>
<th>Dense A (Small)</th>
<th>Dense A (Large)</th>
<th>Sparse A (Small)</th>
<th>Sparse A (Large)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL</td>
<td>Ø</td>
<td>★</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>GiMMiK</td>
<td>★</td>
<td>Ø</td>
<td>★</td>
<td>★</td>
</tr>
<tr>
<td>Libxsmm (new)</td>
<td>★★</td>
<td>▲</td>
<td>★★</td>
<td>▲</td>
</tr>
</tbody>
</table>
Initial Results

• Flow over a cylinder at $Re = 3,900$ and $Ma = 0.2$.

• Quadratically curved hexahedral mesh with $N_E = 118,820$. 
Initial Results
Initial Results

• PyFR 1.4.0: **K40c (cuBLAS)** vs **KNL 7250F (MKL)**.
Initial Results

• Profiling indicates point-wise kernels are the bottleneck.

• Surprising!

• Must therefore rethink our data layout and push for further vectorisation.
Data Layout 101

• Three main layouts:
  • AoS
  • SoA (used by PyFR 1.4.0)
  • AoSoA(k)
struct
{
    float rho;
    float rhou;
    float E;
}
data[NELES];
AoS

- Cache and TLB friendly.
- Difficult to vectorise.
SoA

struct
{
    float rho[NELES];
    float rhou[NELES];
    float E[NELES];
} data;
• Trivial to vectorise.

• Can put pressure on TLB and/or hardware pre-fetchers.
AoSoA\( (k = 2) \)

```
struct {
    float rho[k];
    float rhou[k];
    float E[k];
}
```
$\text{AoSoA}(k = 2)$

- Can be vectorised efficiently for suitable $k$.
- Cache and TLB friendly.
AoSoA($k = 2$)

- The Goldilocks solution
  - …albeit at the cost of **messy indexing**
  - …also requires **coaxing for compilers to vectorise**.
AoSoA($k = 2$)

- Here PyFR’s DSL pays dividends.
- Iteration and indexing are hidden from kernels.
- Can therefore change data structures with ease.
AoSoA(\(k\)) Results

- PyFR 1.4.0 **KNL (MKL)** vs PyFR 1.5.0 **KNL (MKL)**.

![Bar chart showing time per DOF per RK stage in ns for different parallel configurations (p = 1, 2, 3, 4).](chart.png)
AoSoA\(k\) Results

- Comparing with PyFR 1.4.0 \textbf{K40c (cuBLAS)}
AoSoA($k$) Results

- Large **performance boost**.
  - KNL now outperforms a K40c in the **dense regime**.
  - Other backends improve, too, but only slightly.
Sparsity Exploitation

- Hexahedral operator matrices are sparse.
- Can therefore use GiMMiK/libxsmm.
Sparsity Exploitation

- All with PyFR 1.5.0
- **K40c (GiMMiK)**
- **KNL (GiMMiK)**
- **KNL (libxsmm)**
Summary

• Performance is promising.

• Lots of room for improvement in GiMMiK and libxsmm.
Acknowledgements

• Air Force Office of Scientific Research for support under grant FA9550-14-1-0186.
Backup Slides
Start Up Time

• Normally start up time is a non-issue for PyFR.

• A **few minutes** for a simulation which will run for **hours to days**.
Start Up Time

• Time is split roughly equally between

  (i) running serial Python code;

  (ii) using ICC to compile run-time generated kernels.
Start Up Time

• Difficult to improve

• …Python is *virtually impossible to JIT*

• …and due to the **GIL** does not benefit from multi-threading.
Start Up Time

- But did add a **kernel cache**.
- Huge reduction in time for $p = 4$ cylinder case with GiMMiK.