Towards Green Aviation with Python at Petascale

Peter Vincent*, Freddie Witherden†, Brian Vermeire‡, Jin Seok Park§ and Arvind Iyer¶
Department of Aeronautics, Imperial College London, London, United Kingdom
*p.vincent@imperial.ac.uk
†freddie.witherden08@imperial.ac.uk
‡b.vermeire@imperial.ac.uk
§jin-seok.park@imperial.ac.uk
¶a.iyer@imperial.ac.uk

***Abstract*—Accurate simulation of unsteady turbulent flow is critical for improved design of greener aircraft that are quieter and more fuel-efficient. We demonstrate application of PyFR, a Python based computational fluid dynamics solver, to petascale simulation of such flow problems. Rationale behind algorithmic choices, which offer increased levels of accuracy and enable sustained computation at up to 58% of peak DP-FLOP/s on unstructured grids, will be discussed in the context of modern hardware. A range of software innovations will also be detailed, including use of runtime code generation, which enables PyFR to efficiently target multiple platforms, including heterogeneous systems, via a single implementation. Finally, results will be presented from a full-scale simulation of flow over a low-pressure turbine blade cascade, along with weak/strong scaling statistics from the Piz Daint and Titan supercomputers, and performance data demonstrating sustained computation at up to 13.7 DP-PFLOP/s.**

I. Justification for ACM Gordon Bell Prize

- Demonstrated use of Python in a high-end HPC context for simulation of real-world flow problems at up to 13.7 DP-PFLOP/s.
- Detailed how a single Python codebase can target multiple platforms, including heterogeneous systems, using an innovative runtime code-generation paradigm.
- Achieved 58% computational efficiency for an unstructured mesh fluid dynamics simulation.

II. Performance Attributes

Performance attributes of PyFR are listed in Table I.

TABLE I
Performance attributes of PyFR

Attribute	Value
Category of Achievement	Scalability and Peak Performance
Type of Method	Explicit Flux Reconstruction
Nature of Results	Application with and without I/O
Number Precision	Double
System Scale	Measured on Full-Scale Systems
Measurement Mechanism	Timers and Counted FLOPs

III. Overview of the Problem

Air transportation moves over 3.5 billion passengers annually and produces upwards of 700 million tonnes of CO_2 and other greenhouse gases (GHG) [1]. The impact of these emissions is compounded by the fact they are released at high altitudes. As a consequence, GHG emissions from aviation are estimated to be 2-4 times more potent than similar emissions at sea level, and are a significant driver of climate change [2]. In addition to GHG emissions, aviation is a major contributor to noise pollution, which has significant adverse environmental and health impacts. For example, it has been shown that aircraft noise is linked to increased rates of stroke, coronary heart disease, and cardiovascular disease [3]. Global demand for air transportation is rising, particularly in emerging markets and developing countries. The total number of passengers increased by 6.5% in 2015 alone, with similar growth forecasts for the foreseeable future [1].

The European Union (EU) and the United States (US) have embarked on multi-billion dollar research projects to reduce the environmental impacts of aviation. The EU Clean Sky project [4] has the goal of developing and validating technology to enable a 50% reduction in CO_2 emissions, an 80% reduction in NO_X emissions, and a

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00

50% reduction in aircraft noise. Similarly, the National Aeronautics and Space Administration (NASA) has just completed the multi-year Environmentally Responsible Aviation (ERA) [5] project to explore the feasibility, benefits, and technical risks of vehicle concepts that reduce the impact of aviation on the environment. Realising the objectives of Clean Sky and ERA depends on the ability to accurately simulate flow physics associated with various aircraft components using computational fluid dynamics (CFD), which according to the NASA Vision 2030 Study has "fundamentally changed the aerospace design process [...] advanced simulation capabilities provide added physical in-sight, enable superior designs at reduced cost and risk, and open new frontiers in aerospace vehicle design and performance" [6].

One particularly important CFD use-case is the design of jet engine turbines (see Fig. 1). These extract energy from the hot exhaust gasses exiting the combustion chamber of the engine, and use that energy to drive the compressor, fan, and other auxiliary systems. Since this exhaust energy would otherwise contribute directly to thrust, any inefficiencies in the turbine design can have a significant detrimental effect on total aerodynamic efficiency [7]. In addition, turbine cascades typically comprise about one-third of the total engine weight [8]. Every 10kg of aircraft weight reduction can yield over 4 tons of CO_2 reduction per year [1], and so the turbine represents a significant opportunity for reductions in weight and resulting GHG emissions.

Approximately half the weight of a turbine comes from the turbine blades [9][10]. In order to reduce this contribution, modern turbines are designed to use as few blades as possible. However, this results in individual blades being under higher-loading, which can lead to fully-separated flow over the aft-portion of each blade; introducing complex, unsteady, three-dimensional phenomena, and reduced efficiency. This reduction in efficiency translates almost directly to an increase in specific fuel consumption [10]. Therefore, there exists a direct trade-off between reducing the number of blades—and thus total turbine weight—while maintaining an optimum level of aerodynamic performance.

A representative set of highly-loaded configurations are the T106 family of low-pressure turbine (LPT) linear cascades. Based around the mid-span section of the PW2037 LPT blade, these cascades have been tested experimentally in several previous studies [11][12]. In the current study we will apply PyFR [13][14] to simulate unsteady turbulent compressible flow over the T106D configuration described by Stadtmuller et al. [11][12],

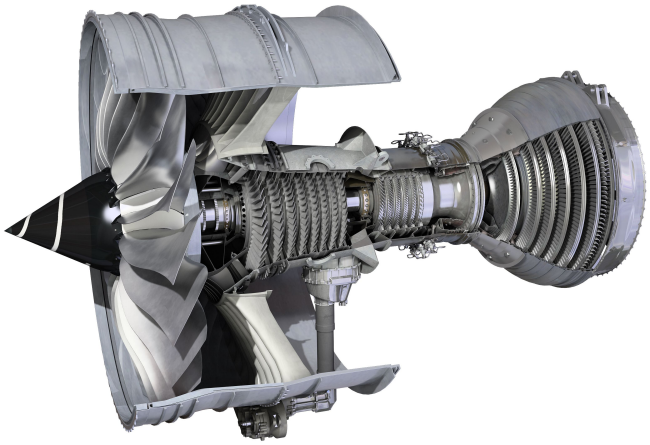

Fig. 1. Schematic illustration of various components in a Rolls-Royce Trent XWB turbofan jet engine. The turbine stages can be seen on the right hand side of the image. Image copyright Rolls-Royce plc. Reused with permission.

which has increased pitch and higher loading relative to the original blade design condition. This case is representative of the general trends in turbine design, with fewer highly-loaded blades per stage, and is particularly challenging as it has a large separation bubble on the suction side of each blade. In particular, we will simulate flow over the full-scale T106D configuration, including five blades at full span, with isotropic resolution throughout the domain. As such, in terms of LPT simulations, we believe the scale and overall degrees of freedom (DOF) count to be unprecedented.

IV. Current State of the Art

The development of novel numerical algorithms has been critical to the continued progression of CFD as an analysis and design tool [6]. The Reynolds-averaged Navier-Stokes (RANS) approach, combined with experimental campaigns, is widely used for industrial design. However, the well-known limitations of RANS methods for unsteady separated flows have confined their use to a limited region of the operating design space [6]. Scale-resolving techniques, such as large eddy simulation (LES), implicit LES (ILES), and direct numerical simulation (DNS) are known to be significantly more accurate in flow regimes where RANS methods often fail. However, these methods are currently considered to be prohibitively expensive by industry. Industrial deployment of scale-resolving techniques will be critical in achieving the step-change technology requirements underpinning the objectives of Clear Sky and ERA.

The majority of CFD codes used to perform scale-resolving simulation of unsteady compressible flow are

based on finite volume (FV) methods with second-order accuracy in space eg. OpenFOAM [15], SU2 [16][17], CHARLES [18]. These methods are able to operate on unstructured meshes, and thus in the vicinity of complex geometries, and have been demonstrated to scale to $\sim$1 million cores [19]. However, they are characterised by a low FLOPS-to-bytes ratio and a high degree of indirect memory access. As such they have been unable to leverage the increasing arithmetic capabilities of modern hardware platforms, which have significantly outpaced advances in random access memory. Algorithms which have traditionally been compute bound—such as dense matrix-vector products—are now limited instead by bandwidth to/from memory. This is epitomised in Fig. 2. Whereas the processors of two decades ago had FLOPS-per-byte of $\sim$0.2, more recent chips have ratios well in excess of 5. This disparity is not limited to just conventional CPUs. Massively parallel accelerators and co-processors such as the NVIDIA K20X and Intel Xeon Phi 5110P have ratios of 5.24 and 3.16, respectively. Hence the performance of second-order FV methods has more closely tracked the grey (memory bandwidth) line in Fig. 2, as opposed to the black (compute) line. Specifically, cache-optimised GPU-based implementations of unstructured FV schemes are heavily memory bandwidth bound, and have been observed to achieve less than 3 percent of peak FLOP/s [20].

In addition to low arithmetic intensity, it is unclear whether second-order accurate FV schemes can effectively resolve all relevant phenomena associated with unsteady compressible turbulent flows, since the methods can be overly dissipative. In recent decades a range of so called high-order accurate schemes have emerged for simulating compressible flows on unstructured meshes. These methods attempt to combine the geometrical flexibility of second-order FV schemes with the superior convergence properties of high-order spectral methods (which cannot be used in complex geometries). Broadly speaking such methods can be classified as either high-order FV schemes or high-order discontinuous finite element (FE) schemes. The most popular high-order FV schemes include k-exact methods [21], FV type essentially non-oscillatory (ENO) methods [22][23], and FV type weighted ENO (WENO) methods [24]. The most popular high-order discontinuous FE methods include high-order discontinuous Galerkin (DG) methods [25][26] and spectral difference (SD) methods; which are similar to DG methods, but based on the governing system in its differential form [27][28]. A comprehensive review of all aforementioned methods has been presented by Wang [29]. Whilst all these methods offer increased solution accuracy, high-order FV schemes such as ENO/WENO methods still suffer from low FLOPS-to-byte ratios, and significant indirection. However, the more compact high-order discontinuous FE schemes offer a large degree of structured computation with substantially less indirection. As such, explicit implementations of these methods can effectively leverage the capabilities of modern hardware platforms. The flux reconstruction (FR) approach [30] adopted in this work is a unifying framework encompassing DG and SD schemes, as well as a range of other similar methods with various stability and accuracy properties [31].

In terms of implementation, CFD codes for HPC are typically written in low-level languages such as C, C++, or Fortran. In addition they are typically designed to target a single platform, and where multiple platforms are supported there is often a degree of cross-platform feature disparity. Whilst these low-level implementations can be performant, lack of portability, and in particular the lack of any ability to target heterogeneous systems, will likely limit their deployment on next-generation systems.

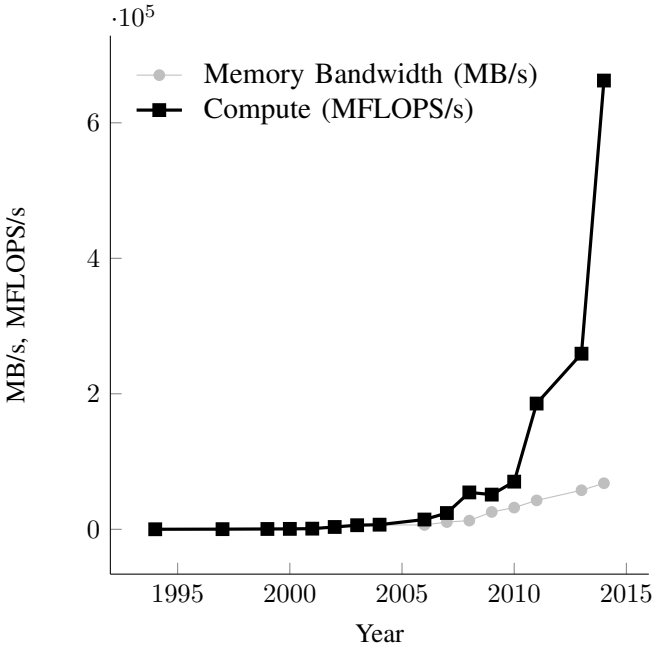

Fig. 2. Trends in the peak compute performance (double precision) and memory bandwidth of server-class Intel processors from 1994-2014. The quotient of these two measures yields the FLOPS-per-byte of a device. Reproduced with modifications from Witherden et al. [13] with permission. Partial data courtesy of Jan Treibig.

V. Innovations Realised

A. Overview

PyFR is an open-source Python framework for solving the compressible Navier-Stokes equations on mixed unstructured meshes using a variety of hardware platforms. As a code, PyFR realises a variety of innovations when compared with the current state of the art. Firstly, instead of using a bandwidth-limited FV scheme, PyFR utilises the high-order FR approach. The FR approach allows PyFR to achieve high-order accuracy on mixed unstructured meshes while simultaneously exposing a large degree of structured computation. Secondly, unlike traditional HPC codes PyFR is written predominantly in the Python programming language. Thirdly, PyFR is performance portable across a range of modern hardware platforms including AMD GPUs, NVIDIA GPUs, and CPUs [14]. This is accomplished through a combination of a C-like domain specific language (DSL) based around the Mako templating engine and runtime code generation. Kernels are specified once in this language and are then translated by the various hardware backends in PyFR—at runtime—into either CUDA, OpenCL, or OpenMP annotated C code. The design of PyFR ensures that all compute is performed on-device with the code designed at all levels to overlap communication with computation [13]. Fourthly, through the use of a common MPI wire-format across all supported hardware platforms, PyFR is able to run on heterogeneous systems with different MPI ranks being assigned to different pieces of hardware [14]. This enables PyFR to make better use of emerging architectures which incorporate a mix of CPUs and accelerators. Taken as a whole, these innovations allow PyFR to run on platforms from the Raspberry Pi through to $18,000$ K20X GPUs of Titan, via heterogeneous systems, with less than $8,000$ lines of code.

B. Flux Reconstruction

The Navier–Stokes equations for compressible flow can be written in conservative form as

$$\frac{\partial u}{\partial t} + \nabla \cdot \mathbf{f}(u, \nabla u) = S(\mathbf{x}, t),$$

where $u(\mathbf{x}, t)$ is a state vector describing the solution, $\mathbf{f}$ is the flux of the solution, and S is a source term. To perform a simulation using FR the computational domain of interest is first discretised into an mesh of conforming and potentially curved elements. Inside of each element two sets of points are defined: one set in the interior, commonly termed the *solution points*, and another on the surface, termed the *flux points*.

In FR the solution polynomial inside of each element is defined by the values of u at the solution points, in a nodal fashion, and is in general discontinuous across elements. This gives rise to a so-called Riemann problem on the interfaces between elements. By solving these Riemann problems it is possible to obtain a common normal flux polynomial along each interface between elements. This polynomial can then be used to correct the divergence of the so-called 'discontinuous flux' inside of each element via a so-called 'correction function', to yield an approximation of $\nabla \cdot \mathbf{f}$ that sits in the same polynomial space as the solution. Once $\nabla \cdot \mathbf{f}$ has been obtained it can be used to march the solution forwards in time via a suitable temporal discretisation. Obtaining $\nabla \cdot \mathbf{f}$ requires two distinct kinds of operations (i) interpolating quantities between flux/solution points, and (ii) evaluating quantities point-wise (such as the flux) at either individual solution points or pairs of flux points. When interpolating quantities, say from the solution points to the flux points, the value at each flux point is given as a weighted sum of the quantity at each solution point

$$q_{e,i}^{(f)} = \sum_j \alpha_{e,ij} q_{e,j}^{(u)},$$

where q represents the quantity, e the element number, i the flux point number, and $\alpha_{e,ij}$ is a matrix of coefficients that encodes the interpolation. This can be identified as a matrix-vector product; or in the case of an N-element simulation, N matrix-vector products. If the quantities are first mapped from physical space to a reference space then the $\alpha_{e,ij}$ coefficients become identical for each element of a given type. Hence, the above operation can be reduced to a single matrix-matrix product. Depending on the order of the solution polynomial with each element, between $\sim 50\%$ and $\sim 85\%$ of the wall clock time in PyFR is spent performing such multiplications. The remaining time is spent in the point-wise operations. These kernels are a generalisation of a function taking the form `g(in1[i], in2[i], ..., &out1[i])`. As there are no data dependencies between iterations the point-wise kernels are both readily parallelisable and highly bound by available memory bandwidth.

We note that both matrix multiplication and point-wise evaluation are amenable to acceleration and thus it is possible to offload all computation within an FR step. Moreover, owing to the discontinuous nature of

the solution polynomials, FR schemes are very well suited to distributed memory parallelism. Specifically, all communication between ranks takes the form of shallow halo exchanges, and all of these exchanges can be performed asynchronously permitting communication to be overlapped with computation.

C. High Performance Computing with Python

Traditionally, the majority of large scale HPC codes are written in C, C++, or Fortran. Although performant these languages lack productivity features such as garbage collection, duck typing, and 'batteries included' runtime libraries. When analysing HPC applications authored in these languages it is not usual for a substantial proportion of the total source lines to be associated with boiler-plate tasks such as memory management, input file parsing, and the handling of error conditions; none of which are in the critical path.

Within PyFR we opt to use the rapid application development (RAD) language Python to eliminate much of this boiler plate code. The idea is for high-level Python code to invoke low-level kernels which are generated at run-time (see section V-D). The overheads associated with this type of approach are minimal and typically of the order of the time required for a foreign function call.

The Python programming language is particularly well suited for this approach. The scientific Python community have worked to develop a variety of modules which wrap standard HPC libraries such as MPI (mpi4py), HDF5 (h5py), CUDA (PyCUDA), OpenCL (PyOpenCL), and the offloading mechanism used by the Intel MIC (pyMIC). As such it is possible to code almost all memory management, I/O, and communications related routines in Python.

D. Performance Portability and Runtime Code Generation

Modern hardware is becoming increasingly heterogeneous. This realisation is apparent at a variety of scales—from desktop CPUs featuring fully programmable integrated GPUs to high-end clusters containing a mix of both CPUs and accelerators. It is therefore important that codes are capable of running across a broad spectrum of hardware platforms and, more importantly, maintaining performance portability across these platforms.

As outlined in the previous sections, the building block of the FR approach is matrix-matrix multiplications. This property is extremely desirable from the standpoint of performance portability as vendors typically provide highly optimised, often hand-written, matrix multiplication routines as part of their software development kits. Use of these routines is crucial for achieving satisfactory single-node performance.

One means of achieving portability per se is to use OpenCL. Within the context of FR, however, OpenCL based solutions have been shown to result in substandard performance [14]. This is on account of the fact that most vendor's linear algebra libraries, including NVIDIA cuBLAS and Intel MKL, lack OpenCL interfaces. Hence, to achieve performance portability an FR code must feature 'backends' for multiple programming languages. However, as a consequence it is necessary reimplement the point-wise kernels multiple times—once for each supported backend.

PyFR innovates here by defining a DSL. Kernels are written *once* using this DSL, which the backend then translates into its native language. In PyFR the DSL is built on top of the popular Mako templating engine. The specification of the DSL exploits the fact that—at least for point-wise operations—the major parallel programming languages C/OpenMP, CUDA, and OpenCL differ only in how kernels are prototyped and how elements are iterated over. In addition to platform portability, the use of a run-time based templating language confers several other advantages. Firstly, Mako permits Python expressions to be used inside templates to aid in generating the source code for a kernel. This is significantly more flexible than the C pre-processor and much simpler than C++ templates. Secondly, as the end result is a Python string it is possible to post-process the code before it is compiled. A use case for this capability within PyFR is to ensure that when running at single precision all floating point constants are suffixed by `.f`. Doing so helps to avoided unwanted auto-promotion of expressions and avoids the need for awkward casts inside the kernel itself. Moreover, it is also trivial to allow for user-defined functions and expressions to be inserted into a kernel. PyFR, for example, permits the form of a source term, $S(\mathbf{x}, t)$, to be specified as part of the input configuration file. Without runtime code generation this would require an expression evaluation library, the performance of which is unlikely to be competitive with the code generated by an optimising compiler.

An example of a kernel written in the DSL is shown in Fig. 3. There are several points of note. Firstly, the kernel is purely scalar in nature; choices such as how to vectorise an operation or how to gather data from memory are all delegated to the backend-specific templating engine. The kernel simply states how to

```
 1. <%inherit file='base'/>
 2. <%namespace
 3.   module='pyfr.backends.base.makoutil'
 4.   name='pyfr'/>
 5.
 6. <%pyfr:kernel
 7.   name='negdivconf' ndim='2'
 8.   t='scalar fpdtype_t'
 9.   tdivf='inout fpdtype_t[${str(nvars)}]'
10.   ploc='in fpdtype_t[${str(ndims)}]'
11.   rcpdj='in fpdtype_t'>
12. % for i, ex in enumerate(srcex):
13.   tdivf[${i}] = -rcpdj*tdivf[${i}] + ${ex};
14. % endfor
15. </%pyfr:kernel>
```

Fig. 3. Example kernel *negdivf* written in the PyFR DSL. This kernel takes the transformed divergence of the flux (*tdivf*) as an input and, using the reciprocal of the determinant of the Jacobian (*rcpdj*), computes the negated physical divergence of the flux.

perform a required operation at a single point inside of a single element. This shields a developer from necessarily having to understand how data is arranged in memory, and permits PyFR to use different memory layouts for different platforms. Secondly, it is possible to utilise Python when generating the main body of kernels. For example on line 12 of Fig. 3 we loop over each of the field variables. Thirdly, the template parameters `ndims` and `nvars` refer to the number of spatial dimensions and field variables, respectively, in the equations being solved. It is hence possible to reuse kernels across not only hardware platforms but also governing systems. Finally, we observe that two input arguments, `t` and `ploc`, appear to go unused. These correspond to the simulation time t and the physical location $\mathbf{x}$ where the operation is performed, respectively. They are potentially referenced by the expressions in `ex` which contains a list of source terms to substitute into the kernel body as shown on line 13. During the code generation phase unused arguments are automatically pruned from function prototypes. For example, in the kernel of Fig 3, this allows PyFR to avoid allocating memory for $\mathbf{x}$ should the source terms have no spatial dependency.

E. Plugin Infrastructure

Within PyFR one can define 'plugins'—written in pure Python—which can analyse and process solution data at the end of each time-step. This infrastructure is used to implement a wide range of capabilities. These include checkpointing, which is implemented with parallel HDF5 via the h5py wrappers, and a variety of in-situ processing algorithms such as point data extraction, surface force integration, and time-averaging, which can all be used to reduce I/O requirements for real-world simulations. By design, this infrastructure abstracts away underlying specifics of the backend. It is thus a lightweight and assessable means of adding in-situ processing and analysis capabilities.

TABLE II
MESH ELEMENT COUNTS

Mesh	N_E
B1S1.760	36,066,250
B2S1.760	72,132,500
B3S1.760	108,198,750
B4S1.760	144,265,000
B5S1.760	180,331,250
B1S0.352	7,213,250
B1S1.056	21,639,750

VI. How Performance Was Measured

A. Test Case

Performance was measured by applying PyFR to solve for flow over a T106D LPT linear cascade using an ILES approach. Conditions from Stadtmuller et al. [11][12] were employed. Specifically, the experimental setup had five LPT blades exposed to uniform inflow, a span to chord ratio of $h/c = 1.76$, a Reynolds number based on the chord c of $Re = 80,000$, and an exit isentropic Mach number of $Ma = 0.4$, in addition to other parameters defined in Fig. 4.

Various unstructured quadratically-curved hexahedral meshes of all or part of the experimental configuration were produced using Gmsh [32], including meshes B1S1.760, B2S1.760, B3S1.760, B4S1.760, and B5S1.760 of one, two, three, four and five blades respectively with $h/c = 1.76$ (100% span), mesh B1S0.352 of one blade with $h/c = 0.352$ (20% span), and mesh B1S1.056 of one blade with $h/c = 1.056$ (60% span). All meshes had the same isotropic mesh resolution throughout their respective domains, and were periodic in the span-wise and pitch-wise directions. The number of elements N_E in each mesh are given in Table II, and a view of the mesh surrounding one of the blades is shown in Fig. 5. All runs were started from an initial condition which was obtained by tessellating the domain with the result from a smaller-scale simulation of flow over one blade, with $h/c = 0.176$ (10% span).

Three types of run were undertaken on a selection of the meshes, namely 'scaling' runs a 'peak sustained

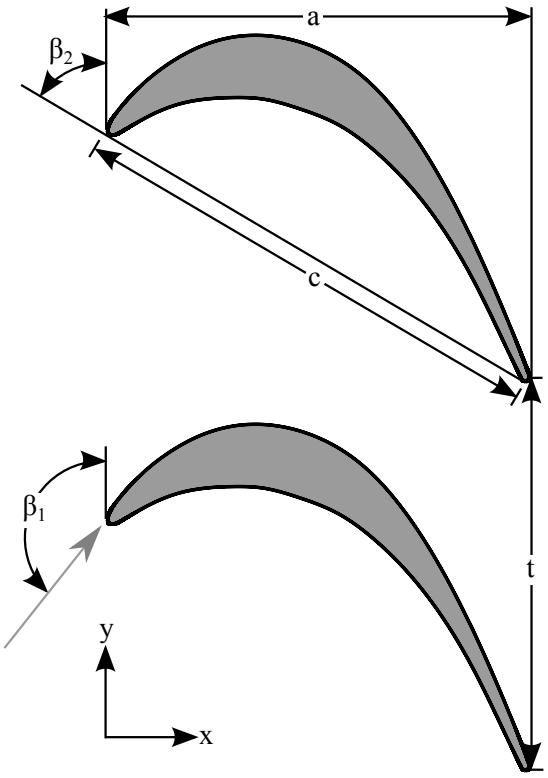

Fig. 4. Schematic of the T106D LPT configuration used in this study. The pitch to chord ratio $t/c = 1.05$, the stagger angle $\beta_2 = 59.28^\circ$ and the inflow angle $\beta_1 = 127.7^\circ$. Light grey arrow indicates direction of inflow.

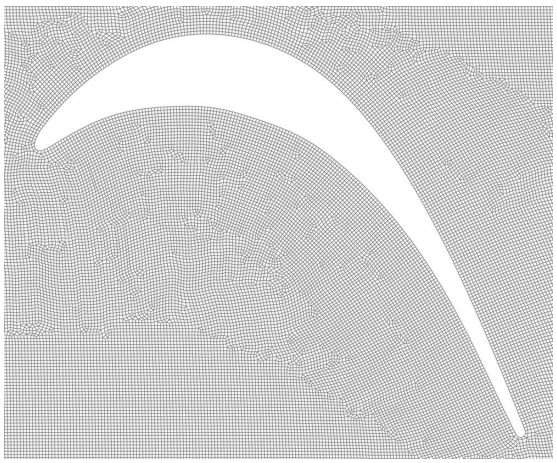

Fig. 5. Cross-sectional view of the mesh surrounding a single LPT blade in the T106D cascade. Note the isotropic nature of the mesh. Each element has a side length of $\sim 0.0055c$.

performance' run, and a 'physics' run. Scaling runs and the peak sustained performance run were designed to assess scaling and efficiency of the solver. They had no plugins enabled (i.e. no I/O or in-situ processing was performed). The scaling runs used fourth-order solution polynomials with each element, and the peak sustained performance run used fifth-order solution polynomials within each element. After start-up, each run proceeded for at least 5 time-steps to bypass any transient hardware or communication behaviour. The total number of time-steps and current wall-clock time were then reported at approximately 10 second intervals over a period of at least 60 time-steps. Combined with prior computation of DP-FLOPs per time-step, these data were used to obtain an overall DP-FLOP/s rate for the solver; where the FLOP count for matrix-multiply kernels between $\mathbf{A}$ and $\mathbf{B}$ was estimated as $2nmk$, with n the number of rows in $\mathbf{A}$, m the number of columns in $\mathbf{B}$, and k the common dimension, and the FLOP count for the point-wise kernels was estimated by hand counting operations. We note that the matrix-multiply kernels contributed two orders-of-magnitude more FLOPs per time-step than the point-wise kernels. We also note that results were within 0.2% of those computed using data from half the number time-steps, demonstrating that the reported performance characteristics were stable.

The physics run was designed to assess the accuracy of the solver by comparison with available experimental data [11][12]. It had the time-averaging and checkpointing plugins enabled, which computed a rolling average of the pressure field in-situ, and wrote this to disk every $1.0c/\sqrt{R_s T_t}$ (where R_s is the specific gas constant and T_t is the total temperature at the inlet), along with a checkpoint file of the entire solution state. The physics run used fourth-order solution polynomials within each element.

All runs used commit `4e2cfb7` to the main PyFR Github repository (https://github.com/vincentlab/PyFR) plus print statements for timing purposes. This commit contains the same functionality as PyFR v1.3.0. All runs were performed at double precision, solution points were located as a tensor product of Gauss-Legendre points within each element, flux points were located as a tensor of Gauss-Legendre points on each face, a Rusanov Riemann solver was employed to calculate common inviscid fluxes between elements, an explicit RK45[2R+] scheme [33] with a fixed time-step was used to advance the solution in time, and full anti-aliasing was employed; of the flux in the volume, the flux on the faces, and the divergence of the flux in the volume. Full setup configurations for each run are available on request.

B. Systems

Performance was measured on two NVIDIA GPU clusters, namely Piz Daint (a Cray XC30 system) at

the Swiss National Supercomputing Centre and Titan (a Cray XK7 system) at Oak Ridge National Laboratory. Both systems are based around the K20X GPU. However, we note that the Aries interconnect used on the XC30 has substantially more bandwidth than the Gemini interconnect of the XK7. In both instances the CUDA backend of PyFR was used to target the GPUs on each system, with each MPI rank being assigned to a single GPU. As PyFR was configured just to target the GPUs, the majority of the CPU cores on the nodes remained idle. Although PyFR is capable of running heterogeneously this comes at the expense of a more complicated domain decomposition and rank-allocation. Given the large discrepancy between CPU and GPU DP-FLOPs it was concluded that the potential benefit from exploiting the CPU cores in this instance did not justify the increased complexity of a heterogeneous simulation. On Piz Daint, Python 3.4.3 and CUDA 6.5.14 were employed for all runs. On Titan, Python 3.5.1 and CUDA 7.5.18 were employed for all runs.

VII. Performance Results

A. Scaling

Scaling runs were undertaken to assess weak and strong scaling on both Piz Daint and Titan. Specifically, scaling runs on meshes B1S1.760, B2S1.760, B3S1.760, B4S1.760, and B5S1.760 were used to assess weak scaling on both systems, and scaling runs on B1S0.352 and B1S1.056 were used to assess strong scaling on Piz Daint and Titan respectively.

Weak scaling results for Piz Daint and Titan can be seen in Tables III and IV, where N_{GPU} is the number of K20X GPUs and T_N is the normalised runtime. Strong scaling results for Piz Daint and Titan can be seen in Tables V and VI, where N_{GPU} is the number of K20X GPUs and S_N is the normalised speedup. It is observed that PyFR achieves excellent weak scaling on both Piz Daint and Titan, attaining in the vicinity of 50% peak accelerator DP-FLOP/s. In term of strong scaling, PyFR achieves an 8.57 fold performance increase on Piz Daint moving from 200 to 2,000 K20X GPUs for a fixed problem size. However, beyond this, moving to 4,000 K20X GPUs, parallel efficiency begins to drop. This can be attributed to the fact that with 4,000 K20X GPUs memory load is only $\sim$5%. Similar results are observed on Titan, with PyFR achieving an 8.41 fold performance increase moving from 600 to 6,000 K20X GPUs for a fixed problem size.

TABLE III
Weak scaling of PyFR on Piz Daint

Mesh	N_{GPU}	DP-PFLOP/s	% Peak DP-FLOP/s	T_N
B1S1.760	1,000	0.67	51.0%	1.00
B2S1.760	2,000	1.33	50.8%	1.00
B3S1.760	3,000	2.00	50.8%	1.00
B4S1.760	4,000	2.65	50.6%	1.01

TABLE IV
Weak scaling of PyFR on Titan

Mesh	N_{GPU}	DP-PFLOP/s	% Peak DP-FLOP/s	T_N
B1S1.760	3,000	1.93	49.2%	1.00
B2S1.760	6,000	3.88	49.3%	1.00
B3S1.760	9,000	5.79	49.2%	1.00
B4S1.760	12,000	7.76	49.3%	1.00
B5S1.760	15,000	9.65	49.1%	1.00

TABLE V
Strong scaling of PyFR on Piz Daint

Mesh	N_{GPU}	DP-PFLOP/s	% Peak DP-FLOP/s	S_N
B1S0.352	200	0.14	52.0%	1.00
B1S0.352	400	0.27	50.8%	1.95
B1S0.352	1,000	0.64	48.6%	4.67
B1S0.352	2,000	1.17	44.6%	8.57
B1S0.352	4,000	1.99	38.1%	14.64

TABLE VI
Strong scaling of PyFR on Titan

Mesh	N_{GPU}	DP-PFLOP/s	% Peak DP-FLOP/s	S_N
B1S1.056	600	0.40	50.9%	1.00
B1S1.056	1,200	0.79	50.0%	1.97
B1S1.056	3,000	1.87	47.7%	4.69
B1S1.056	6,000	3.36	42.8%	8.41

B. Peak Sustained Performance

The peak sustained performance run was undertaken with mesh B5S1.760 on 18,000 K20X GPUs of Titan. The run had 195 billion DOFs, and achieved 13.7 DP-PFLOP/s (58.0% peak accelerator DP-FLOP/s).

C. Physics

The physics run was undertaken with mesh B5S1.760 on 5,000 K20X GPUs of Titan. It had 113 billion DOFs. The simulation ran for $\sim$35 hours, advancing the solution $\sim 9.0c/\sqrt{R_s T_t}$ time units. This equates to approximately 2.7 flow passes over the chord of each blade. We note

that writing each pressure average and checkpoint file to disk took $\sim$7.0 minutes. There were a total of 9 such writes, accounting for $\sim$3.0% of the total run time.

Fig. 6 shows a view of the whole configuration. Specifically, a snapshot of density gradient magnitude is shown on two span-normal planes. Flow separation, and transition to a turbulent wake is apparent. Fig. 7 shows a snapshot of density gradient magnitude in a plane situated within the wake of a single blade. Fig. 8 shows a snapshot of Q-criteria isosurfaces, which defines vortical structures, coloured by velocity magnitude in the wake of a single blade. The high span-wise resolution of the simulation is apparent. Flow structures can be seen to decay as they are advected downstream. Finally, Fig. 9 shows a plot of time-averaged isentropic Mach number on the pressure and suction surfaces at the centre-span of a single LPT blade, along with associated experimental data [11]. We note that PyFR achieves very good agreement with the experimental data.

VIII. Implications

A. High Performance Computing with Python

We have demonstrated the utility of Python as a language for the implementation of accelerated HPC codes that can run at petascale. Future use of Python could reduce development time and ongoing maintenance burden of next-generation HPC software, whilst increasing flexibility and extensibility.

B. Performance Portability and Runtime Code Generation

We have demonstrated the utility of runtime code generation for HPC at petascale. Wider adoption of the paradigms showcased in this work could lead to decreased maintenance burden for cross-platform implementations, and more efficient targeting of heterogeneous systems, which are becoming increasingly prevalent.

C. Computational Fluid Dynamics on Next-Generation Systems

We have demonstrated how, via an appropriate choice of numerical method, the arithmetic capabilities of modern hardware platforms can be effectively harnessed for CFD simulations on unstructured meshes. Even with advances such as high bandwidth memory, it is likely that the FLOPs-to-byte ratio of next-generation systems—such as Summit and Sierra—will be similar to that of current systems. As such this achievement will be of ongoing relevance.

Fig. 6. Snapshot of density gradient magnitude on two span-normal planes at $9.0c/\sqrt{R_s T_t}$ time units.

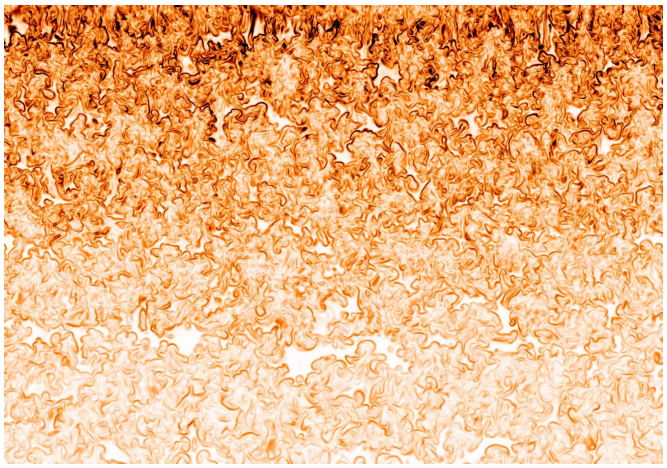

Fig. 7. Snapshot of density gradient magnitude in a plane situated within the wake of a single blade at $9.0c/\sqrt{R_s T_t}$ time units. Flow is from top-to-bottom, the span-wise direction is from left-to-right.

Fig. 8. Snapshot of iso-surfaces of Q-criteria coloured by velocity magnitude in the wake of a single blade at $9.0c/\sqrt{R_s T_t}$ time units.

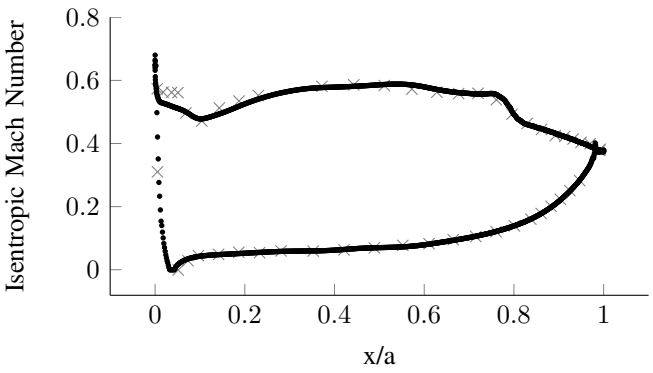

Fig. 9. Time-averaged isentropic Mach number distribution over the pressure and suction surfaces of a single blade measured at mid-span obtained from the PyFR simulation (black dots), and associated experimental data [11] (grey crosses). The PyFR data was obtained by averaging over $9.0c/\sqrt{R_s T_t}$ time units.

D. Design of Low-Pressure Turbine Blades and Green Aviation

We have demonstrated the ability of PyFR to perform ILES of an entire highly-loaded LPT cascade using an isotropic mesh, at what we believe to be an unprecedented scale. The simulations achieved very good agreement with isentropic Mach number distributions obtained from experiments. Going forwards, we believe similar simulations could be used to underpin a full-scale virtual LPT cascade capability, that is able to fully resolve inlet turbulence, including injected wakes from preceding blade rows, as well as the effect of side walls, and other complex three-dimensional features. Such a capability would alleviate current reliance on expensive experimental campaigns for LPT design. Additionally, data generated from such a capability would be more detailed than that currently available from experiments, and could be used to refine and improve existing RANS/LES turbulence models, increasing their utility in the LPT design cycle. In summary, this capability could revolutionise the design of LPT configurations; maximising aerodynamic efficiency, reducing total engine weight, and thus lowering aircraft GHG emissions.

In addition to LPT design, the technology in PyFR could underpin solutions to a range of outstanding green aviation challenges, including development of optimised high-pressure turbines, fans, compressors, high-lift configurations, as well as optimisation of landing gear acoustics, cavity acoustics, and active and passive flow control devices. The ability to use ILES/DNS as a design tool will allow engineers to move beyond the limitations of RANS, and enable the step-changes in design that are required to meet the targets of the Clean Sky and ERA projects. Such advances are essential for the sustainability of commercial aviation in the coming decades.

ACKNOWLEDGMENT

We acknowledge support from the Engineering and Physical Sciences Research Council (EP/K027379/1, EP/L000407/1, EP/M50676X/1, and a Doctoral Training Grant), Innovate UK (101890), and the European Commission under Horizon 2020 (635962). We also thank the Centre for Innovation for access to Emerald, Cambridge University for access to Wilkes, the Swiss National Supercomputing Centre for access to Piz Daint, and Oak Ridge National Laboratory for access to Titan. Finally, we thank Utkarsh Ayachit, T.J. Corona and Robert Maynard at Kitware, Jeremy Purches, Tom Fogal and Peter Messmer at NVIDIA, Jamil Appa, David Standingford and Mark Allan at Zenotech, and Andrei Cimpoeru at CFMS, for useful technical discussions.

REFERENCES

[1] "Air passenger market analysis - December 2015," International Air Transport Association, Tech. Rep., 2015.

[2] Intergovernmental Panel on Climate Change, "Aviation and the global atmosphere: A special report of IPCC working groups I and III." Tech. Rep., 1999.

[3] A. L. Hansell, M. Blangiardo, L. Fortunato, S. Floud, K. de Hoogh, D. Fecht, R. E. Ghosh, H. E. Laszlo, C. Pearson, L. Beale, S. Beevers, J. Gulliver, N. Best, S. Richardson, and P. Elliott, "Aircraft noise and cardiovascular disease near Heathrow airport in London: small area study," *British Medical Journal*, vol. 347, 2013.

[4] Council of European Union, "Council regulation (EU) no 71/2008: Setting up the Clean Sky Joint Undertaking," 2007.

[5] F. Collier, "Overview of NASA's environmentally responsible aviation (ERA) project," in *NASA Environmentally Responsible Aviation Project Pre-Proposal Meeting*, Washington, DC, Feb 2010.

[6] J. Slotnick, A. Khodadoust, J. Alonso, D. Darmofal, W. Gropp, E. Lurie, and D. Mavriplis, "CFD vision 2030 study: A path to revolutionary computational aerosciences," National Aeronautics and Space Administration, Tech. Rep. NASA/CR-2014-218178, NF1676L-18332, 2014.

[7] P. Tucker, "Computation of unsteady turbomachinery flows: Part 1–progress and challenges," *Progress in Aerospace Sciences*, vol. 47, no. 7, pp. 522–545, 2011.

[8] E. M. Curtis, H. P. Hodson, M. R. Banieghbal, J. D. Denton, R. J. Howell, and N. W. Harvey, "Development of blade profiles for low-pressure turbine applications," *Journal of Turbomachinery*, vol. 119, no. 3, pp. 531–538, 1997.

[9] J. Cui, V. Rao, and P. G. Tucker, "Numerical investigation of contrasting flow physics in different zones of a high-lift low-pressure turbine blade," *Journal of Turbomachinery*, vol. 138, no. 1, p. 10, 2015.

[10] J. Gier, M. Franke, N. Hubner, and T. Schroder, "Designing low pressure turbines for optimized airfoil lift," *Journal of Turbomachinery*, vol. 132, no. 3, p. 12, 2010.

[11] P. Stadtmüller, L. Fottner, and A. Fiala, "Experimental and numerical investigation of wake-induced transition on a highly loaded LP turbine at low Reynolds numbers," in *Proceedings of ASME Turbo Expo 2000*, Munich, Germany, 2000, 2000-GT-0269.

[12] P. Stadtmüller and L. Fottner, "A test case for the numerical investigation of wake passing effects on a highly loaded LP turbine cascade blade," in *Proceedings of ASME Turbo Expo 2001*, New Orleans, Louisiana, 2001, 2001-GT-0311.

[13] F. D. Witherden, A. M. Farrington, and P. E. Vincent, "PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach," *Computer Physics Communications*, vol. 185, no. 11, pp. 3028–3040, 2014.

[14] F. D. Witherden, B. C. Vermeire, and P. E. Vincent, "Heterogeneous computing on mixed unstructured grids with PyFR," *Computers & Fluids*, vol. 120, pp. 173–186, 2015.

[15] "OpenFOAM." [Online]. Available: http://www.openfoam.org

[16] "SU2." [Online]. Available: http://su2.stanford.edu

[17] F. Palacios, J. Alonso, K. Duraisamy, M. Colonno, J. Hicken, A. Aranake, A. Campos, S. Copeland, T. Economon, A. Lonkar, T. Lukaczyk, and T. Taylor, "Stanford University Unstructured (SU2): An open-source integrated computational environment for multi-physics simulation and design," in *Proceedings of 51st AIAA Aerospace Sciences Meeting*, Grapevine, Texas, January 2013, pp. 1–60.

[18] "CHARLES." [Online]. Available: http://www.cascadetechnologies.com/charles

[19] "Stanford Researchers Break Million-core Supercomputer Barrier." [Online]. Available: http://engineering.stanford.edu/news/stanford-researchers-break-million-core-supercomputer-barrier

[20] J. Langguth, N. Wu, J. Chai, and X. Cai, "On the GPU performance of cell-centered finite volume method over unstructured tetrahedral meshes," in *Proceedings of 3rd Workshop on Irregular Applications Architectures and Algorithms - IA^3 '13*. Denver, Colorado: ACM Press, 2013.

[21] T. J. Barth and P. O. Frederickson, "Higher order solution of the Euler equations on unstructured grids using quadratic reconstruction," in *Proceedings of 28th AIAA Aerospace Sciences Meeting*, Reno, Nevada, 1990, AIAA 1990-0013.

[22] A. Harten, B. Engquist, S. Osher, and S. R. Chakravarthy, "Uniformly high order accurate essentially non-oscillatory schemes, III," *Journal of Computational Physics*, vol. 71, no. 2, pp. 231–303, 1987.

[23] R. Abgrall, "On essentially non-oscillatory schemes on unstructured meshes: analysis and implementation," *ICASE Report*, no. 92-74, 1992.

[24] X.-D. Liu, S. Osher, and T. Chan, "Weighted essentially non-oscillatory schemes," *Journal of Computational Physics*, vol. 115, no. 1, pp. 200–212, 1994.

[25] W. H. Reed and T. R. Hill, "Triangular mesh methods for the neutron transport equation," Los Alamos National Laboratory, New Mexico, USA, Tech. Rep. LA-UR-73-479, 1973.

[26] J. S. Hesthaven and T. Warburton, *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer Verlag New York, 2008, vol. 54.

[27] D. A. Kopriva and J. H. Kolias, "A conservative staggered-grid Chebyshev multidomain method for compressible flows," *Journal of Computational Physics*, vol. 125, no. 1, pp. 244–261, 1996.

[28] Y. Liu, M. Vinokur, and Z. J. Wang, "Spectral difference method for unstructured grids I: Basic formulation," *Journal of Computational Physics*, vol. 216, no. 2, pp. 780–801, 2006.

[29] Z. J. Wang, "High-order methods for the Euler and Navier-Stokes equations on unstructured grids," *Progress in Aerospace Sciences*, vol. 43, no. 1-3, pp. 1–41, 2007.

[30] H. T. Huynh, "A flux reconstruction approach to high-order schemes including discontinuous Galerkin methods," in *Proceedings of 18th AIAA Computational Fluid Dynamics Conference*, Miami, Florida, 2007, AIAA 2007-4079.

[31] P. E. Vincent, P. Castonguay, and A. Jameson, "A new class of high-order energy stable flux reconstruction schemes," *Journal of Scientific Computing*, vol. 47, no. 1, pp. 50–72, 2011.

[32] C. Geuzaine and J.-F. Remacle, "Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities," *International Journal for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.

[33] C. A. Kennedy, M. H. Carpenter, and R. M. Lewis, "Low-storage, explicit runge–kutta schemes for the compressible navier–stokes equations," *Applied numerical mathematics*, vol. 35, no. 3, pp. 177–219, 2000.