Performance analysis of GPU accelerated meshfree q-LSKUM solvers in Fortran, C, Python, and Julia

1st Nischay Ram Mamidi
Centre for High Performance Computing
BITS Pilani, Hyderabad Campus
Hyderabad, India
nischay@hyderabad.bits-pilani.ac.in

2nd Dhruv Saxena
Department of Mathematics
BITS Pilani, Hyderabad Campus
Hyderabad, India
f20191369@hyderabad.bits-pilani.ac.in

3rd Kumar Prasun
Department of Computer Science
Courant Institute of Mathematical Sciences
New York, USA
kumarprasun@nyu.edu

4th Anil Nemili
Department of Mathematics
BITS Pilani, Hyderabad Campus
Hyderabad, India
anil@hyderabad.bits-pilani.ac.in

5th Bharatkumar Sharma
NVIDIA
Bengaluru, India
bharatk@nvidia.com

6th S. M. Deshpande
FASc, FNAE, FAeSI Fellow
Bengaluru, India
desh1942@gmail.com

***Abstract*—This paper presents a comprehensive analysis of the performance of `Fortran`, `C`, `Python`, and `Julia` based GPU accelerated meshfree solvers for compressible flows. The programming model CUDA is used to develop the GPU codes. The meshfree solver is based on the least squares kinetic upwind method with entropy variables (*q*-LSKUM). To measure the performance of baseline codes, benchmark calculations are performed. The codes are then profiled to investigate the differences in their performance. Analysing various performance metrics for the computationally expensive flux residual kernel helped identify various bottlenecks in the codes. To resolve the bottlenecks, several optimisation techniques are employed. Post optimisation, the performance metrics have improved significantly, with the `C` GPU code exhibiting the best performance.**

***Index Terms*—Fortran, C, Python, Julia, GPUs, CUDA, Meshfree methods, LSKUM, Performance analysis, Code optimisation.**

I. Introduction

High performance computing (HPC) plays a critical part in the numerical simulation of complex aerodynamic configurations. Typically, such simulations require solving the governing Euler or Navier-Stokes equations on fine grids ranging from a few million to several billion grid points. To perform such computationally intensive calculations, the computational fluid dynamics (CFD) codes use either only CPUs or CPU-GPUs. However, for computations on multiple GPUs, CPUs tackle control instructions and file input-output operations, while GPUs perform the compute intensive floating point arithmetic. Over the years, GPUs have evolved as a competitive alternative to CPUs in better performance, cost, and energy efficiency. Furthermore, they consistently outperform CPUs in single instruction multiple data (SIMD) scenarios. Many research groups have developed GPU codes for CFD applications using traditional programming languages such as `Fortran` or `C` [1]–[4].

In recent years, modern languages such as `Python` [5], `Julia` [6], `Regent` [7], and `Chapel` [8] have steadily risen in the domain of scientific computing. Unlike `Fortran` and `C`, these languages are architecture independent with the added advantage of easy code maintenance and readability. Here, `Regent` and `Chapel` provide an implicitly parallel model, where task division and data synchronisation are performed automatically. On the other hand, `Python` and `Julia` adopt an explicit approach to parallelism similar to `Fortran` and `C`.

The journey of accelerating CFD codes starts with implementing baseline code, which largely represents the same structure of the sequential code. However, such baseline GPU codes may not be computationally efficient. This could be due to poor memory access patterns, kernel launch configurations, size of the kernels, and redundant floating-point operation sequences. The next step is the cycle of code optimisation, where baseline codes are profiled to get valuable insights related to performance. The profiled data can be used to analyse the bottlenecks in performance critical codes. Resolving these issues can significantly enhance the computational efficiency of the GPU codes. The profilers help the domain scientists understand the utilisation of the GPU hardware. Furthermore, they provide a guided analysis, which a specialist could have otherwise done in parallel programming. This paper highlights the importance of profiling and the cycle of analysis and optimisation by highlighting metrics important to our code.

To the best of our knowledge, a rigorous investigation and comparison of the performance of GPU codes for CFD written in both traditional and modern languages are not yet pursued. Towards this objective, in this paper, an attempt has been made to present a comprehensive analysis of the performance of GPU solvers for two-dimensional Euler equations. For the CFD solver, the meshfree Least Squares Kinetic Upwind Method with entropy variables (*q*-LSKUM) [9] is used. The LSKUM based CFD codes are being used

in the National Aerospace Laboratories and the Defence Research and Development Laboratory, India, to compute flows around aircraft and flight vehicles [10]–[14]. The programming model CUDA [15] is used to construct the GPU solvers. Here, `Fortran` and `C` represent the traditional languages. `Python` and `Julia` are selected for modern languages as they are increasingly being used in CFD [16] and other domains of scientific computing such as machine learning [17], astrophysics [18], bioinformatics [19] and drug discovery [20]. Although the compilers of the four languages are different, they target a common interface called CUDA API. By analysing the performance of the baseline and the optimised GPU meshfree solvers, we want to investigate how the ecosystem of these four languages has evolved.

This paper is organised as follows. Section II describes the basic theory of the meshfree scheme based on $\boldsymbol{q}$-LSKUM. Section III presents the pseudo-code of the serial and GPU accelerated meshfree solvers. In Section IV, we first present benchmarks comparing the runtime performance of the baseline GPU codes. To investigate the difference in their runtimes, the codes are profiled. A methodology to identify the bottlenecks that hamper the performance is presented by analysing relevant metrics. Section V presents various optimisation strategies to enhance computational efficiency. Furthermore, numerical results are presented to compare the performance of optimised GPU codes with the baseline codes. Finally, Section VI presents the conclusions and a plan for future work.

II. Meshfree $\boldsymbol{q}$-LSKUM solver

The Least Squares Kinetic Upwind Method (LSKUM) [21] is a meshfree scheme for the numerical solution of conservation laws that govern compressible fluid flows. The basic idea of LSKUM is to first approximate the spatial derivatives in the upwind Boltzmann equation using the least squares principle. After taking suitable moments [22], [23], we obtain the meshfree numerical scheme for the Euler or Navier-Stokes equations. To approximate the spatial derivatives in the governing partial differential equations, LSKUM requires a distribution of points, known as a point cloud. The cloud of points can be obtained from various point generation algorithms [24]. This section illustrates the theory of LSKUM for the numerical solution of Euler equations.

In two-dimensions ($2D$), the Euler equations are given by

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{Gx}}{\partial x} + \frac{\partial \boldsymbol{Gy}}{\partial y} = 0 \tag{1}$$

Here, $\boldsymbol{U}$ is the conserved vector, $\boldsymbol{Gx}$ and $\boldsymbol{Gy}$ are the flux vectors along the coordinates x and y, respectively. In order to construct an upwind scheme for the Euler equations, consider the Courant-Issacson-Rees (CIR) split [25] $2D$ Boltzmann equation,

$$\frac{\partial F}{\partial t} + v_1^+ \frac{\partial F}{\partial x} + v_1^- \frac{\partial F}{\partial x} + v_2^+ \frac{\partial F}{\partial y} + v_2^- \frac{\partial F}{\partial y} = 0 \tag{2}$$

where $v_i^{\pm} = \left(v_i \pm |v_i|\right)/2$. Here F is the Maxwellian velocity distribution function, and v_1 and v_2 are the molecular velocities along the coordinates x and y, respectively. Second-order approximations to the spatial derivatives F_x and F_y can be obtained using the defect correction procedure [26], [27]. To derive the desired formulae for the unknowns at a point P_0, consider the Taylor series expansion of F up to quadratic terms at a point $P_i \in N\left(P_0\right)$,

$$\begin{aligned}\Delta F_i =& \Delta x_i F_{x_0} + \Delta y_i F_{y_0} + \frac{\Delta x_i}{2}\left(\Delta x_i F_{xx_0} + \Delta y_i F_{xy_0}\right) \\ &+ \frac{\Delta y_i}{2}\left(\Delta x_i F_{xy_0} + \Delta y_i F_{yy_0}\right) + O\left(\Delta x_i, \Delta y_i\right)^3,\ i = 1, \ldots, n\end{aligned} \tag{3}$$

where $\Delta x_i = x_i - x_0$, $\Delta y_i = y_i - y_0$, $\Delta F_i = F_i - F_0$. Here, n denotes the number of neighbours of the point P_0. $N\left(P_0\right)$ is the set of neighbours or the stencil of P_0. To cancel the second-order derivative terms in the above equation, consider the Taylor series expansions of F_x and F_y to linear terms

$$\begin{aligned}\Delta F_{x_i} =& F_{x_i} - F_{x_0} = \Delta x_i F_{xx_0} + \Delta y_i F_{xy_0} + O\left(\Delta x_i, \Delta y_i\right)^2 \\ \Delta F_{y_i} =& F_{y_i} - F_{y_0} = \Delta x_i F_{xy_0} + \Delta y_i F_{yy_0} + O\left(\Delta x_i, \Delta y_i\right)^2\end{aligned} \tag{4}$$

Substituting the above expressions in eq. (3), we obtain

$$\Delta F_i = \Delta x_i F_{x_0} + \Delta y_i F_{y_0} + \frac{1}{2}\left\{\Delta x_i \Delta F_{x_i} + \Delta y_i \Delta F_{y_i}\right\} \tag{5}$$

Define a modified perturbation in Maxwellians, $\Delta\widetilde{F}_i = \widetilde{F}_i - \widetilde{F}_0$ so that the leading terms in the truncation errors of the formulae for F_x and F_y are of the order of $\left(\Delta x_i, \Delta y_i\right)^2$,

$$\Delta\widetilde{F}_i = \Delta F_i - \frac{1}{2}\left(\Delta x_i \Delta F_{x_i} + \Delta y_i \Delta F_{y_i}\right) \tag{6}$$

Using $\Delta\widetilde{F}_i$, eq. (5) reduces to

$$\Delta\widetilde{F}_i = \Delta x_i F_{x_0} + \Delta y_i F_{y_0} + O\left(\Delta x_i, \Delta y_i\right)^3,\ i = 1, \ldots, n \tag{7}$$

For $n \geq 3$, we get an over-determined linear system of equations. Using the least squares principle, we can obtain the discrete approximations to F_x and F_y at the point P_0 [27]. Taking $\boldsymbol{\Psi}$ - moments of eq. (2) along with the least squares formulae, we get the semi-discrete second-order upwind scheme for Euler equations based on LSKUM,

$$\frac{d\boldsymbol{U}}{dt} + \frac{\partial \boldsymbol{Gx}^+}{\partial x} + \frac{\partial \boldsymbol{Gx}^-}{\partial x} + \frac{\partial \boldsymbol{Gy}^+}{\partial y} + \frac{\partial \boldsymbol{Gy}^-}{\partial y} = 0 \tag{8}$$

Here, $\boldsymbol{Gx}^{\pm}$ and $\boldsymbol{Gy}^{\pm}$ are the kinetic split fluxes [26] along the x and y directions, respectively. Note that the spatial derivatives of $\boldsymbol{Gx}^{\pm}$ and $\boldsymbol{Gy}^{\pm}$ are approximated using the stencils $N_x^{\pm}\left(P_0\right)$ and $N_y^{\pm}\left(P_0\right)$, respectively. These split stencils are defined by $N_x^{\pm}\left(P_0\right) = \{P_i \in N\left(P_0\right) \mid \Delta x_i \lessgtr 0\}$ and $N_y^{\pm}\left(P_0\right) = \{P_i \in N\left(P_0\right) \mid \Delta y_i \lessgtr 0\}$. For example, the expressions for the spatial derivatives of $\boldsymbol{Gx}^{\pm}$ are given by

$$\frac{\partial \boldsymbol{Gx}^{\pm}}{\partial x} = \frac{\sum \Delta y_i^2 \sum \Delta x_i \Delta\widetilde{\boldsymbol{Gx}}_i^{\pm} - \sum \Delta x_i \Delta y_i \sum \Delta y_i \Delta\widetilde{\boldsymbol{Gx}}_i^{\pm}}{\sum \Delta x_i^2 \sum \Delta y_i^2 - \left(\sum \Delta x_i \Delta y_i\right)^2} \tag{9}$$

The perturbations $\Delta\widetilde{\boldsymbol{Gx}}_i^{\pm}$ are defined by

$$\Delta\widetilde{\boldsymbol{Gx}}_i^{\pm} = \Delta \boldsymbol{Gx}_i^{\pm} - \frac{1}{2}\left\{\Delta x_i \frac{\partial}{\partial x}\Delta \boldsymbol{Gx}_i^{\pm} + \Delta y_i \frac{\partial}{\partial y}\Delta \boldsymbol{Gx}_i^{\pm}\right\} \tag{10}$$

A disadvantage of this approach is that it may not yield second-order accuracy near the boundary points as the split

stencils to compute the partial derivatives of $\Delta \boldsymbol{G}\boldsymbol{x}^{\pm}$ may not have enough neighbours. Another drawback is that the numerical solution may not be positive as the distributions $\widetilde{F}_i$ and $\widetilde{F}_0$ need not be Maxwellians [9].

To preserve the positivity of the solution, $\boldsymbol{q}$-variables [9], [22] can be employed in the defect correction procedure instead of Maxwellians. An advantage of this approach is that second-order spatial accuracy can be obtained even at the boundary points [10], [23]. Furthermore, $\boldsymbol{q}$-variables can represent the fluid flow at the macroscopic level as the transformations $\boldsymbol{U} \longleftrightarrow \boldsymbol{q}$ and $F \longleftrightarrow \boldsymbol{q}$ are unique. The $\boldsymbol{q}$-variables in $2D$ are defined by

$$\boldsymbol{q} = \left[\ln \rho + \tfrac{\ln \beta}{\gamma - 1} - \beta\left(u_1^2 + u_2^2\right),\ 2\beta u_1,\ 2\beta u_2,\ -2\beta\right] \tag{11}$$

Using $\boldsymbol{q}$-variables, a second-order accurate upwind method can be obtained by replacing $\Delta\widetilde{\boldsymbol{G}\boldsymbol{x}}_i^{\pm}$ in eq. (10) with $\Delta \boldsymbol{G}\boldsymbol{x}_i^{\pm}\left(\widetilde{\boldsymbol{q}}\right) = \boldsymbol{G}\boldsymbol{x}^{\pm}\left(\widetilde{\boldsymbol{q}}_i\right) - \boldsymbol{G}\boldsymbol{x}^{\pm}\left(\widetilde{\boldsymbol{q}}_0\right)$. Here, $\widetilde{\boldsymbol{q}}_i$ and $\widetilde{\boldsymbol{q}}_0$ are the modified $\boldsymbol{q}$-variables, defined by

$$\widetilde{\boldsymbol{q}}_{i,0} = \boldsymbol{q}_{i,0} - \frac{1}{2}\left(\Delta x_i \boldsymbol{q}_{x\,i,0} + \Delta y_i \boldsymbol{q}_{y\,i,0}\right) \tag{12}$$

Here $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ are evaluated to second-order using the least squares as

$$\begin{bmatrix}\boldsymbol{q}_x \\ \boldsymbol{q}_y\end{bmatrix} = \begin{bmatrix}\sum \Delta x_i^2 & \sum \Delta x_i \Delta y_i \\ \sum \Delta x_i \Delta y_i & \sum \Delta y_i^2\end{bmatrix}^{-1} \begin{bmatrix}\sum \Delta x_i \Delta \widetilde{\boldsymbol{q}}_i \\ \sum \Delta y_i \Delta \widetilde{\boldsymbol{q}}_i\end{bmatrix} \tag{13}$$

The above formulae are implicit and need to be solved iteratively using full stencil. These iterations are referred to as inner iterations. The present work constructs the state-update formula for eq. (8) using a four stage third-order Runge-Kutta time marching algorithm [28] with local time stepping.

III. GPU ACCELERATED $\boldsymbol{q}$-LSKUM SOLVER

This section presents the development of a GPU accelerated meshfree solver based on `q-LSKUM`. We begin with a brief description of the steps required to compute the flow solution using a serial code.

Algorithm 1: Serial meshfree solver based on q-LSKUM

```
subroutine q_LSKUM
    call preprocessor()
    for n ← 1 to n ≤ N do
        call timestep()
        for rk ← 1 to 4 do
            call q_variables()
            call q_derivatives()
            call flux_residual()
            call state_update(rk)
        end
        call residue()
    end
    call postprocessor()
end subroutine
```

Algorithm 2: GPU accelerated meshfree solver based on q-LSKUM

```
subroutine q_LSKUM:
    call preprocessor()
    cudaHostToDevice(CPU_data, GPU_data)
    for n ← 1 to n ≤ N do
        kernel ⋘ grid, block ⋙ timestep()
        for rk ← 1 to 4 do
            kernel ⋘ grid, block ⋙
             q_variables()
            kernel ⋘ grid, block ⋙
             q_derivatives()
            kernel ⋘ grid, block ⋙
             flux_residual()
            kernel ⋘ grid, block ⋙
             state_update(rk)
        end
        reduction residue()
    end
    cudaDeviceToHost(GPU_data, CPU_data)
    call postprocessor()
end subroutine
```

Algorithm 1 presents a general structure of the serial meshfree `q-LSKUM` solver for steady-state flows [27]. The solver consists of a fixed point iterative scheme, where each iteration evaluates the local time step, four stages of the Runge-Kutta scheme, and the L_2 norm of the residue. The subroutine `q_variables()` evaluates the $\boldsymbol{q}$-variables defined in eq. (11) while `q_derivatives()` computes the second-order accurate approximations of $\boldsymbol{q}_x$ and $\boldsymbol{q}_y$ using the formulae in eq. (13). The most time consuming routine is the `flux_residual()`, which performs the least squares approximation of the kinetic split flux derivatives in eq. (8). `state_update(rk)` updates the flow solution at each Runge-Kutta step. All the input and output operations are performed in `preprocessor()` and `postprocessor()`, respectively. The parameter N represents the number of pseudo-time iterations required to achieve a desired convergence in the flow solution.

Algorithm 2 presents the structure of a GPU accelerated `q-LSKUM` solver written in CUDA. The GPU solver mainly consists of the following sequence of operations: transfer the input data structure from host to device, performing fixed-point iterations on the device, and finally transfer the converged flow solution from device to host. In the baseline implementation, for each serial function in Algorithm 1, equivalent CUDA kernels [15] are constructed in the GPU code. Some of these kernels in-turn call other kernels. For example, the `q_derivatives()` kernel initially calls the

kernel that computes the first-order accurate q_derivatives. This is followed by another kernel to perform inner iterations to obtain the second-order accurate q_derivatives. Similarly, the `flux_residual()` kernel calls the kernels to compute the split flux derivatives $\frac{\partial \boldsymbol{Gx}^{\pm}}{\partial x}$, and $\frac{\partial \boldsymbol{Gy}^{\pm}}{\partial y}$. The evaluation of residue is a reduction operation, performed on the GPU. At each fixed point iteration, the value of the residue is communicated back to the CPU. Note that this is the only data that is communicated at each iteration.

IV. Performance analysis of baseline GPU solvers

This section presents the numerical results to assess the performance of baseline GPU solvers written in `Fortran`, `C`, `Python`, and `Julia`. Note that, the implementation of the meshfree $\boldsymbol{q}$-LSKUM algorithm is the same for all the languages. The test case under investigation is the inviscid fluid flow simulation around the NACA 0012 airfoil at Mach number, $M = 0.63$, and angle of attack, $AoA = 2^o$. For the benchmarks, numerical simulations are performed on seven levels of point distributions. The coarsest distribution consists of $625,000$ points, while the finest distribution consists of 40 million points. For second order spatial accuracy, three inner iterations are performed to solve the implicit formulae for $\boldsymbol{q}$-derivatives in eq. (13).

Table I shows the hardware configuration, while Table II shows the language specifications, compilers, and flags used to execute serial and GPU computations. The `Python` GPU code uses Numba $0.55.0$ [29] and NumPy $1.20.1$ [30], while `Julia` GPU code uses CUDA.jl $2.4.1$ library [6]. Although `Python` provides `PyCUDA`, we want to test the performance of Numba, as it allows the developers to implement CUDA GPU code in native `Python`. All the computations are performed with double precision and `-O3` optimisation flags using CUDA $11.2.2$.

	CPU	GPU
Model	AMD EPYC$^{\text{TM}}$ 7542	Nvidia Tesla $V100$ PCIe
Cores	64 (2×32)	5120
Core Frequency	2.20 GHz	1.230 GHz
Global Memory	256 GiB	32 GiB
$L2$ Cache	16 MiB	6 MiB

TABLE I: Hardware configuration used to perform numerical simulations.

A. RDP comparison of GPU Solvers

To measure the performance of the GPU codes, we adopt a cost metric called the Rate of Data Processing (RDP). The RDP of a meshfree code can be defined as the total wall clock time in seconds per iteration per point. Note that lower the value of RDP implies better the performance. Table III shows a comparison of the RDP values for all the GPU codes. In the present work, the RDP values are measured by specifying the number of pseudo-time iterations in the GPU solvers to 1000. Numerical simulations are performed with 32, 64, 128, and 256 threads per block. For `Fortran`, `Python` and `Julia` GPU codes, the lowest RDP value on all levels of point distribution is achieved with 64 threads per block. For `C`, the lowest RDP is obtained with 128 threads per block.

Language	Version	Compiler	Version
`Fortran`	Fortran 90	nvfortran	21.2
`C`	C 20	nvcc	21.2
`Python`	Python 3.9.1	Numba	0.55.0
`Julia`	Julia 1.5.3	CUDA.jl	2.4.1

TABLE II: List of language and compiler specifications used to execute the codes.

Level	Points	Fortran	C	Python	Julia
	RDP $\times 10^{-8}$ (Lower is better)				
1	0.625M	14.4090	5.1200	9.4183	7.3120
2	1.25M	12.8570	4.8800	8.9765	6.2160
3	2.5M	11.9100	4.6000	8.7008	5.4800
4	5M	11.5620	4.6673	8.6080	5.2800
5	10M	11.3640	4.5800	8.6409	5.0600
6	20M	11.3130	4.4096	7.9278	4.9650
7	40M	12.2720	4.2573	7.8805	4.9350

TABLE III: Comparison of the RDP values based on baseline GPU codes.

The tabulated values clearly show that the GPU solver based on `C` results in lowest RDP values on all levels of point distribution and thus exhibits superior performance. On the other hand, with the highest RDP values, the `Fortran` code is computationally more expensive. As far as the `Julia` code is concerned, its performance is better than `Python` and closer to `C`.

To assess the overall performance of the GPU meshfree solvers, we define another metric called speedup. The speedup of a GPU code is defined as the ratio of the RDP of the serial `C` code to the RDP of the GPU code. Figure 1 shows the speedup achieved by the GPU codes. We observe that the `C` code is around 2.5 times faster than `Fortran`, while `Julia` and `Python` are respectively 2 and 1.5 times faster than `Fortran`.

B. Run-time analysis of kernels

To analyse the performance of the GPU accelerated meshfree solvers, it is imperative to investigate the kernels employed in the solvers. Towards this objective, NVIDIA Nsight Compute [31] is used to profile the GPU codes. Table IV shows the relative run-time incurred by the kernels on coarse, medium, and finest point distributions. Here, the relative run-time of a kernel is defined as the ratio of the kernel execution time to the overall time taken for the complete simulation.

This table shows that a very significant amount of run-time is taken by the `flux_residual` kernel, followed by `q_derivatives`. Note that the run-time of `q_derivatives` kernel depends on the number of inner

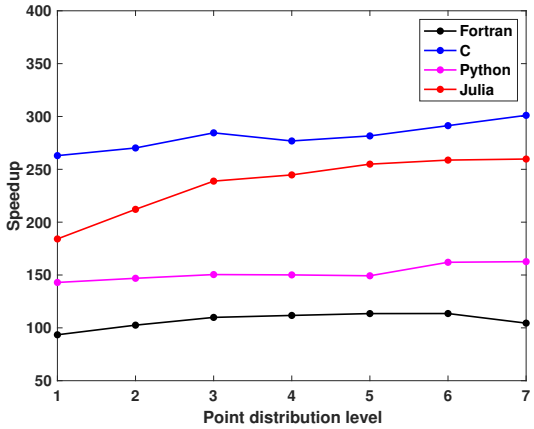

Fig. 1: Speedup achieved by the GPU codes.

Points	Code	q_variables	q_derivatives	flux_residual	state_update
0.625M Coarse	Fortran	0.50%	25.73%	72.67%	0.82%
	C	0.77%	44.70%	50.51%	1.87%
	Python	0.67%	37.48%	59.73%	1.47%
	Julia	1.24%	24.52%	71.71%	1.89%
5M Medium	Fortran	0.42%	25.60%	72.95%	0.74%
	C	0.80%	47.34%	47.68%	1.84%
	Python	0.60%	38.43%	59.10%	1.38%
	Julia	1.37%	24.40%	71.77%	1.85%
40M Fine	Fortran	0.41%	25.38%	73.21%	0.74%
	C	0.81%	42.27%	52.94%	1.85%
	Python	0.58%	38.19%	59.40%	1.35%
	Julia	1.32%	24.12%	72.11%	1.85%

TABLE IV: Run-time analysis of the kernels on the finest point distribution.

iterations. More the number of inner iterations, higher the time spent in its execution. For the kernels `q_variables` and `state_update`, the run-times are less than 2% of the total execution time. For `timestep`, `residue`, and `host` $\leftrightarrow$ `device` operations, the run-times are found to be negligible and therefore not presented.

C. Performance metrics of the kernel flux_residual

To understand the varied run-times of the GPU codes in executing the `flux_residual` kernel, we investigate the kernel's utilisation of streaming multiprocessor (SM) and memory and achieved occupancy [31]. Table V shows a comparison of these metrics on coarse, medium, and finest point distributions. We can observe that the `C` code has the highest utilisation of available SM resources, followed by `Python` and `Julia` codes. On the other hand, the `Fortran` code has the poorest utilisation. Higher SM utilisation indicates an efficient usage of CUDA streaming multiprocessors, while lower values imply that the GPU resources are underutilised. In the present work, poor SM utilization limited the performance of the Fortran code as more time is spent in executing the flux-residual kernel. This resulted in higher RDP values for the `Fortran` code.

Table V also presents the overall memory utilisation of the

Points	Code	SM utilisation	Memory utilisation	Achieved occupancy	Registers per thread
		shown in percentage			
0.625M Coarse	Fortran	11.56	21.27	3.08	220
	C	43.16	10.41	11.76	184
	Python	29.55	25.95	18.03	128
	Julia	26.23	18.28	16.54	152
5M Medium	Fortran	11.70	21.57	3.10	220
	C	45.78	11.34	12.03	184
	Python	30.05	26.35	18.29	128
	Julia	26.61	18.15	16.77	152
40M Fine	Fortran	11.68	21.49	3.10	220
	C	43.58	9.15	12.03	184
	Python	30.31	26.58	18.33	128
	Julia	27.10	18.24	16.76	152

TABLE V: A comparison of performance metrics on coarse, medium and finest point distributions.

GPU codes. This metric shows the total usage of device memory. Furthermore, it also indicates the memory throughput currently being utilised by the kernel. Memory utilisation can become a bottleneck on the performance of a kernel if it reaches its theoretical limit [31]. However, low memory utilisation does not imply that the kernel optimally utilises it. The tabulated values show that the memory utilisation of the GPU codes is well within the acceptable limits.

To understand the poor utilisation of SM resources, we investigate the achieved occupancy of the `flux_residual` kernel. The achieved occupancy is the ratio of the number of active warps per SM to the maximum number of theoretical warps per SM. A code with high occupancy allows the SM to execute more active warps, thus increasing the overall SM utilisation. Low occupancy limits the number of active warps eligible for execution, leading to poor parallelism and latency. In the present work, all the GPU codes exhibited low occupancy for the `flux_residual` kernel. Table V also compares register usage, one of the metrics that determine the number of active warps. In general, the higher the register usage, the lower the number of active warps. With the highest register usage, the `Fortran` code has the lowest occupancy. The tabulated values have shown that the utilisation of SM and memory, and achieved occupancy of the `Python` code is higher than the `Julia` code. However, the RDP values of `Python` are much higher than `Julia`.

To investigate this unexpected behaviour of the `Python` code, we present the roofline analysis [32] of the `flux_residual` kernel. A roofline model is a logarithmic plot that shows a kernel's arithmetic intensity with its maximum achievable performance. The arithmetic intensity is defined as the number of floating-point operations per byte of data movement. Figure 2 shows the roofline analysis for all the GPU codes. Here, achieved performance is measured in trillions of floating-point operations per second. A code with performance closer to the peak boundary uses the GPU resources optimally. The `C` code, being closer to the roofline, yielded the best performance, while `Fortran` is the farthest and resulted

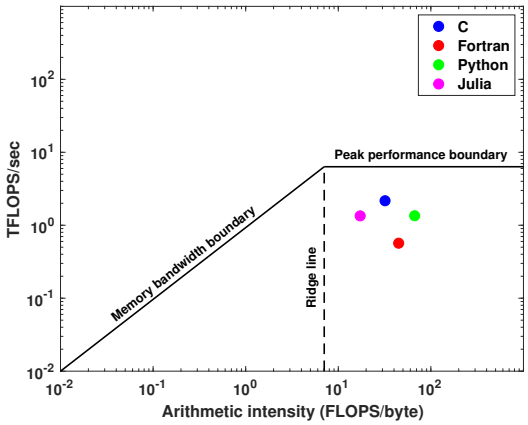

Fig. 2: Roofline analysis of the `flux_residual` kernel.

in poor performance. Although the achieved performance of `Python` is the same as `Julia`'s, its arithmetic intensity is much higher. Due to this, the RDP values of `Python` are higher than `Julia`.

To investigate the difference in the utilisation of SM and memory, and the arithmetic intensity of `Python` and `Julia` codes, we analyse the scheduler and warp state statistics. For comparison, we used `C`, as it exhibited superior performance. Due to this we did not include `Fortran` in the discussion. Typically scheduler statistics consist of the metrics - *GPU maximum warps*, *active*, *eligible*, and *issued warps*. Here, *GPU maximum warps* is the maximum number of warps that can be issued per scheduler. For the NVIDIA V100 GPU card, the maximum warps is 16. The warps for which resources such as registers and shared memory are allocated are known as *active warps*. *Eligible warps* are the subset of *active warps* that have not been stalled and are ready to issue their next instruction. From this set of *eligible warps*, the scheduler selects warps for which one or more instructions are executed. These warps are known as *issued warps*. Note that *active warps* is the sum of *eligible* and *stalled warps*. As far as the warp state statistics are concerned, it comprises several states for which warp stalls can occur. In the present work, the warp stalls due to *no instruction*, *wait*, and *long scoreboards* [31] are dominant. *No instruction* warp stall occurs when a warp is waiting to get selected to execute the next instruction. Furthermore, it can also happen due to instruction cache miss. In general, a cache miss occurs in kernels with many assembly instructions. A warp stalls due to *wait* if it is waiting for fixed latency execution dependencies such as fused multiply-add (FMA) or arithmetic-logic units (ALU). A *Long scoreboard* stall occurs when a warp waits for the requested data from `L1TEX`, such as local or global memory units. If the memory access patterns are not optimal, then the waiting time for retrieving the data increases further.

Table VI shows the *scheduler statistics* on the finest point distribution. From this table we can observe that the `C` code has the lowest number of *active warps*. Although the number

Points	Code	Active	Eligible	Issued	Eligible warps
		warps per scheduler			in percentage
40M	`C`	1.93	0.24	0.21	12.43%
Fine	`Python`	2.93	0.37	0.30	12.62%
	`Julia`	2.69	0.24	0.20	8.92%

TABLE VI: A comparison of scheduler statistics on the finest level of point distribution.

Points	Code	Stall in warp execution (in cycles) due to		
		no instruction	wait	long scoreboard
40M	`C`	2.96	3.12	0.87
Fine	`Python`	4.94	2.14	0.66
	`Julia`	5.4	2.6	3.10

TABLE VII: A comparison of warp state statistics on the finest level of point distribution.

of *active warps* is more in `Python` and `Julia`, they are still much lesser than the GPU *maximum warps*. This is due to high register usage per thread in the `flux_residual` kernel. The tabulated values also show that the *eligible warps* are much less than the *active warps*, as most *active warps* are stalled.

We investigate the *warp state statistics* to understand the reason behind the low eligible warps in the `flux_residual` kernel. Table VII shows a comparison of stall statistics measured in cycles. Note that the cycles spent by a warp in a stalled state define the latency between two consecutive instructions. These cycles also describe a warp's readiness or inability to issue the next instruction. The larger the cycles in the warp stall states, the more warp parallelism is required to hide latency. The tabulated values show that the overall stall in warp execution is maximum for `Julia`. Due to this, `Julia` has the lowest percentage of eligible warps.

The scheduler and warp state statistics analysis did not reveal any conclusive evidence regarding the poor performance of `Python` code over `Julia`. To further analyse, we shift our focus towards the instructions executed inside the warps. In this regard, we investigate the global and shared memory access patterns of the warps and the pipe utilisation of the SM.

Table VIII shows a comparison of metrics related to global memory access. Here, global load corresponds to the load operations to retrieve the data from the global memory. In contrast, global store refers to the store operations to update the data in the global memory. A *sector* is an aligned 32 byte-chunk of global memory. The metric, *sectors per request*, is the average ratio of sectors to the number of load or store operations by the warp. Note that the higher the *sectors per request*, the more cycles are spent processing the load or store operations. We observe that the `Python` code has the highest number of *sectors per request* while `Julia` has the lowest values. With the highest number of *sectors per request*, the `Python` code suffers from poor memory access patterns.

Code	Global Load		Global Store	
	Sectors	Sectors per request	Sectors	Sectors per request
C	3,789,109,860	10.63	43,749,721	8.75
Python	14,637,012,265	26.92	159,999,732	32.00
Julia	7,884,258,310	7.41	40,000,000	8.00

TABLE VIII: A comparison of global load and store metrics on the finest level of point distribution.

Points	Code	Shared memory bank conflicts due to	
		load operations	store operations
40M	C	0	0
	Python	3,824,672	107,628,065
	Julia	4,413,868	0

TABLE IX: A comparison of shared memory bank conflicts due to load and store operations.

Table IX shows a comparison of shared memory bank conflicts for C, Python, and Julia codes. A bank conflict occurs when multiple threads in a warp access the same memory bank. Due to this, the load or store operations are performed serially. The C code does not have any bank conflicts, while Julia has bank conflicts due to load operations only. The Python code has a significantly large number of bank conflicts and thus resulted in the poor performance of the flux_residual kernel.

Table X shows the utilisation of dominant pipelines such as double-precision floating-point (FP64), Fused Multiply Add (FMA), Arithmetic Logic Unit (ALU), and Load Store Unit(LSU) for the flux_residual kernel. The FP64 unit is responsible for executing instructions such as DADD, DMUL, and DMAD. A code with a high FP64 unit indicates more utilisation of 64-bit floating-point operations. The FMA unit handles instructions such as FADD, FMUL, FMAD, etc. This unit is also responsible for integer multiplication operations such as IMUL, IMAD, and integer dot products. The ALU is responsible for the execution of logic instructions. The LSU pipeline issues load, store, atomic, and reduction instructions for global, local, and shared memory. The tabulated values show that Python and Julia codes have similar FP64 and LSU utilisation. However, the Python code has excessive utilisation of FMA and ALU. This is due to the Numba JIT compiler, which is not generating optimal SASS code for the flux_residual kernel.

To analyse the excessive utilisation of the FMA and ALU pipelines in Python, Table XI compares the dominant instructions executed on the SM. We can observe that the Python code has generated an excessive number of IMAD and IADD3 operations that are not part of the meshfree solver. The additional instructions are generated due to CUDA thread indexing. This hampered the overall performance of the Python code.

Points	Code	FP64	FMA	ALU	LSU
40M	C	43.63	6.58	5.87	1.78
	Python	28.67	14.28	21.24	8.05
	Julia	27.09	9.41	9.43	7.97

TABLE X: A comparison of pipe utilisation of the streaming multiprocessor (SM).

Points	Code	DFMA	IMAD	DMUL	IADD3	DADD
		Instructions presented in Billions				
40M	C	6.1262	2.7451	2.0509	0.9514	1.4174
	Python	8.2769	14.1171	2.3879	4.1338	3.1966
	Julia	6.3009	6.8711	2.2617	2.6878	1.4201

TABLE XI: A comparison of various instructions executed by an SM.

In summary, the C code with better utilisation of SM yielded the lowest RDP values. The Fortran code with very low occupancy resulted in the highest RDP. The Python code has better utilisation of SM and memory and achieved occupancy compared to Julia. However, it suffers from global memory coalescing, shared memory bank conflicts, and excessive utilisation of FMA and ALU pipelines. Due to this, the RDP values of the Python code are significantly higher than the Julia code.

V. Performance analysis of optimised GPU solvers

The analysis of several performance metrics has shown that there is scope for further improvement in the computational efficiency of the flux_residual kernel. Towards this objective, various optimisation techniques have been employed. The profiler metrics have shown that the register usage of the flux_residual kernel is very high, which indicates that the size of the kernel is too large. To circumvent this problem, the flux_residual kernel is split into four smaller kernels that compute the spatial derivatives of the split fluxes $\boldsymbol{Gx}^+$, $\boldsymbol{Gx}^-$, $\boldsymbol{Gy}^+$ and $\boldsymbol{Gy}^-$, respectively. Note that these kernels are of similar size. In general, a smaller kernel consumes fewer registers compared to a larger kernel. Furthermore, kernels that are limited by registers will have an improved occupancy. Table XII shows a comparison of register usage per thread, achieved occupancy, and global sectors per request for the baseline and optimised GPU codes. We present a range for metrics with both a lower and an upper bound for all the split flux kernels of the optimised codes. For the optimised codes, we present a range for metrics that has both a lower and an upper bound for all the split flux kernels. The tabulated values show a significant decrease in the register usage of the Fortran code, followed by C and Julia. In the case of Python, the reduction is observed to be marginal. We also observe that the smaller kernels have more achieved occupancy compared to the flux_residual kernel. However, in the case of Python, the occupancy did not improve as it is limited by the shared memory required per thread block.

Code	Registers per thread	Achieved occupancy	Global sectors per request	
			Load	Store
Fortran - baseline	220	3.10	24.34	31.56
Fortran - optimised	156	17.84 – 18.10	17.86 – 18.25	7.11
C - baseline	184	12.03	10.63	8.75
C - optimised	154	17.81 – 18.10	10.19 – 10.31	8.75
Python - baseline	128	18.33	26.92	32.00
Python - optimised	122	17.87 – 18.16	26.30 – 26.51	32.00
Julia - baseline	152	16.76	6.29	4.37
Julia - optimised	128	23.69 – 24.02	6.26 – 6.31	4.42

TABLE XII: A comparison of register usage, occupancy, and global sectors per request of the baseline and optimised GPU codes on the finest point distribution.

To further enhance the computational efficiency of the kernels, the following language-specific optimisations are implemented. For the Fortran code, instead of accessing and updating the arrays in an iterative loop, array slices are used. This improved the memory access patterns and global memory coalescing. Table XII clearly shows a reduction in Fortran's load and store operations, which increases the SM utilisation. However, this optimisation technique does not apply to the C code, as arrays are used instead of vectors. It is also not applicable for Julia code, where values are accessed individually from a two-dimensional array. In the case of Python, the current Numba compiler is unable to compile kernels that use array slices.

In the baseline implementation of Fortran, Python, and Julia codes only thread index is used to access the values of the variables stored in shared memory. In the optimised version both the thread index and block dimensions are used to access the shared memory. This approach optimised the array indexing and allowed the threads to access the memory without any bank conflicts. Figures 3 and 4 show the shared memory access patterns for the baseline and optimised GPU codes in Fortran, Python, and Julia. Listing 1 presents an example code in Python with 4-way shared memory bank conflicts. Listing 2 presents a version of the code with optimised indexing for shared memory arrays.

However, both in the baseline and optimised versions of the GPU codes in C shared memory is not implemented as it deteriorated the performance significantly. This is due to the added latency in accessing shared memory compared to registers. In the case of Fortran, Python, and Julia codes without shared memory implementation led to high register pressure and thread spilling. It is observed that the latency costs due to thread spilling is much more than the costs incurred by accessing shared memory. Therefore implementation of shared memory in these languages enhanced the computational performance.

All the above optimisation techniques, except kernel splitting are implemented in other kernels wherever applicable. Table IV shows that, after the flux_residual, the q_derivatives is the most computationally intensive kernel. Splitting of this kernel is not feasible as $\boldsymbol{q}$-derivatives in eq. (13) are evaluated implicitly. Note that these optimisations may not yield a considerable reduction in the RDP values of smaller point distributions. However, on finest point distributions involving millions of points, these changes will significantly reduce the RDP values.

Table XIII shows a comparison of SM utilisation, performance in TFLOPS, and arithmetic intensity. Compared to the flux_residual kernel, the split flux kernels have more SM utilisation and thus resulted in more TFLOPS. For the split kernels based on Fortran, C, and, Python the arithmetic intensity is to the right of the ridgeline value of 7.05. This implies that the kernels of these codes are compute bounded. On the other hand, the Julia code is memory bounded as the arithmetic intensity of its split kernels lies to the left of the ridgeline.

Table XIV shows a comparison of RDP values based on the optimised GPU codes. Note that for all the optimised codes, the optimal number of threads per block is 128. We can observe that the optimisation has significantly enhanced the efficiency of the codes and thus resulted in smaller RDP values. The C code has the lowest RDP values on all levels of point distribution, followed by Fortran. Although optimisation techniques have reduced the RDP values of the Python code, they are till higher than the Julia code. Figure 5 shows the speedup achieved by the optimised codes. From this figure, on the finest point distribution, the C code is around 1.5 times faster than the Python code, while Fortran and Julia codes are faster than Python by 1.2 and 1.1 times respectively.

Points	Code	SM utilisation	Performance in TFLOPS	Arithmetic intensity
40M Fine	Fortran - baseline	11.68	0.57	44.89
	Fortran - optimised	47.85 – 48.68	2.35 – 2.41	10.71 – 10.90
	C - baseline	43.58	2.167	32.00
	C - optimised	56.41 – 58.30	2.79 – 2.88	9.12 – 9.66
	Python - baseline	30.31	1.3491	66.84
	Python - optimised	54.29 – 55.36	2.58 – 2.64	18.20 – 18.30
	Julia - baseline	27.10	1.3443	17.25
	Julia - optimised	34.19 – 34.42	1.69 – 1.70	4.93 – 7.93

TABLE XIII: A comparison of SM utilisation, performance, and arithmetic intensity of the naive and optimised GPU codes.

Points	Version	Fortran	C	Python	Julia
RDP $\times 10^{-8}$ (Lower is better)					
0.625M	baseline	14.4090	5.1200	9.4183	7.3120
0.625M	optimised	9.4446	4.0671	6.1372	7.5040
5M	baseline	11.5620	4.6673	8.6080	5.2800
5M	optimised	4.5856	3.4616	5.2355	4.6900
40M	baseline	12.2720	4.2573	7.8805	4.9350
40M	optimised	4.3365	3.4100	5.1540	4.6825

TABLE XIV: A comparison of the RDP values based on baseline and optimised GPU codes.

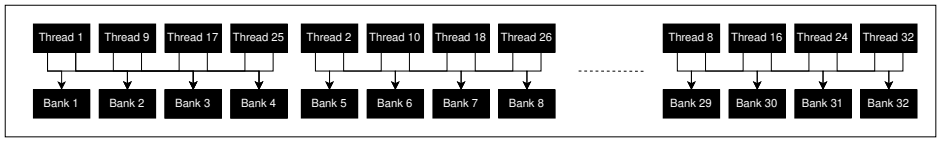

Fig. 3: 4-way bank conflicts in baseline Fortran, Python and Julia GPU codes.

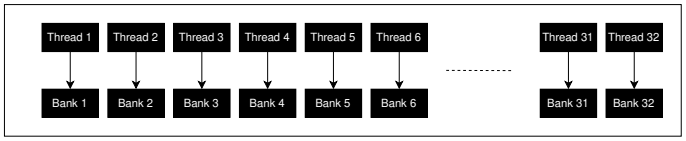

Fig. 4: Shared memory access pattern in optimised Fortran, Python and Julia GPU codes.

```python
@cuda.jit
def example_kernel(a):
    thread_index_x = cuda.threadIdx.x
    block_index_x = cuda.blockIdx.x
    block_width_x = cuda.blockDim.x # 128 threads per block
    temp = cuda.shared.array(shape = (128 * 4), dtype=numba.float64)
    for i in range(4):
        temp[thread_index_x * 4 + i] = a[i]
```

Listing 1: An example in `Python` with baseline version of indexing for shared memory arrays.

```python
@cuda.jit
def example_kernel(a):
    thread_index_x = cuda.threadIdx.x
    block_index_x = cuda.blockIdx.x
    block_width_x = cuda.blockDim.x # 128 threads per block
    temp = cuda.shared.array(shape = (128 * 4), dtype=numba.float64)
    for i in range(4):
        temp[thread_index_x + block_width_x * i] = a[i]
```

Listing 2: An example in `Python` with optimised version of indexing for shared memory arrays.

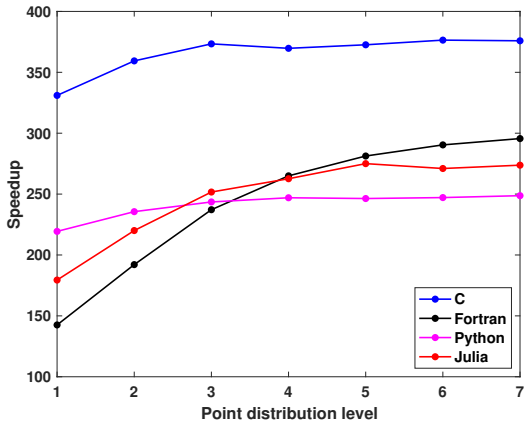

Fig. 5: Speedup achieved by the optimised GPU codes.

For `C` and `Python`, the RDP values decreased with point refinement and saturated after the third level of point distribution. This is due to the SM utilisation stagnating as we move towards finer point distributions. In the case of `Fortran` we observed an increase in SM utilisation with successive point refinement, resulting in a continuous reduction in RDP values. Similar arguments can be drawn for `Julia`. The same behaviour is exhibited in the speedup plot.

VI. Conclusions

In this paper, we have presented a performance analysis of `Fortran`, `C`, `Python`, and `Julia` based baseline and optimised GPU meshfree solvers. The meshfree solver chosen for the analysis was based on the least squares kinetic upwind method with entropy variables (q-LSKUM). Benchmark simulations were performed on seven levels of point distribution ranging from 0.625 million to 40 million points. To measure the performance of the GPU solvers, a metric called the rate of data processing (RDP) was introduced. This metric was defined as the total wall clock time in seconds per iteration per point.

The latest NVIDIA profiling tools were used to investigate the differences in the RDP values of the baseline GPU codes. For the computationally expensive kernel `flux_residual`, various performance metrics were captured. Analysing these metrics helped identify bottlenecks such as low occupancy, high warp stalls, uncoalesced global memory access patterns, shared memory bank conflicts, excessive FMA and ALU pipeline utilisation.

To resolve these bottlenecks, various optimisation strategies were employed. These include kernel splitting, shared memory indexing, and array splicing. The impact of these optimisation techniques on `Fortran` code was significant, as the RDP

values decreased significantly. However, C code still exhibited superior performance, followed by Fortran, Julia, and Python.

The present work highlighted the underlying software stack differences that resulted in varied metrics across prominent programming languages used in CFD. CUDA C has been supported by NVIDIA since its inception in 2007 and provides more compiler-level optimisation strategies. This generated an efficient GPU SASS code yielding the smallest RDP values. Modern languages Python and Julia provide more programming productivity and code maintainability at the cost of some performance hit. However, with the advent of NVIDIA's CUDA Python and rapid developments in Julia's CUDA library, the performance gap of these languages with Fortran and C can certainly be narrowed.

ACKNOWLEDGMENT

The authors gratefully acknowledge NVIDIA AI Technology Center (NVAITC) for supporting this research. They also acknowledge the computing time provided on the high performance computing facility, Sharanga, at the Birla Institute of Technology and Science - Pilani, Hyderabad Campus.

REFERENCES

[1] M. R. López, A. Sheshadri, J. R. Bull, T. D. Economon, J. Romero, J. E. Watkins, D. M. Williams, F. Palacios, A. Jameson, and D. E. Manosalvas, "Verification and validation of HiFiLES: a High-Order LES unstructured solver on multi-GPU platforms," *AIAA Paper 2014-3168*, 2014.

[2] D. Jude and J. D. Baeder, "Extending a three-dimensional GPU RANS solver for unsteady grid motion and free-wake coupling," *AIAA Paper 2016-1811*, 2016.

[3] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. J. Kelly, and D. Radford, "Acceleration of a full-scale industrial cfd application with op2," *IEEE Transactions on Parallel & Distributed Systems*, vol. 27, no. 05, pp. 1265–1278, may 2016.

[4] A. Walden, E. Nielsen, B. Diskin, and M. Zubair, "A mixed precision multicolor point-implicit solver for unstructured grids on gpus," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2019, pp. 23–30.

[5] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[6] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[7] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for HPC with logical regions," in *Supercomputing (SC)*, 2015.

[8] B. L. Chamberlain, *Chapel (Cray Inc. HPCS Language)*. Boston, MA: Springer US, 2011, pp. 249–256.

[9] S. M. Deshpande, K. Anandhanarayanan, C. Praveen, and V. Ramesh, "Theory and application of 3-D LSKUM based on entropy variables," *Int. J. Numer. Meth. Fluids*, vol. 40, pp. 47–62, 2002.

[10] V. Ramesh and S. M. Deshpande, "Least squares kinetic upwind method on moving grids for unsteady Euler computations," *Comp. & Fluids*, vol. 30, no. 5, pp. 621–641, 2001.

[11] K. Anandhanarayanan, K. Arora, V. Shah, R. Krishnamurthy, and D. Chakraborty, "Separation dynamics of air-to-air missile using a grid-free euler solver," *Journal of Aircraft*, vol. 50, no. 3, pp. 725–731, 2013.

[12] K. Anandhanarayanan, R. Krishnamurthy, and D. Chakraborty, "Development and validation of a grid-free viscous solver," *AIAA Journal*, vol. 54, no. 10, pp. 3312–3315, 2016.

[13] V. Ramesh and S. M. Deshpande, "Kinetic mesh-free method for flutter prediction in turbomachines," *Sadhana*, vol. 39, no. 1, pp. 149–164, 2014.

[14] C. Praveen, A. Ghosh, and S. M. Deshpande, "Positivity preservation, stencil selection and applications of LSKUM to 3-D inviscid flows," *Computers and Fluids*, vol. 38, no. 8, pp. 1481–1494, 2009.

[15] D. Kirk, "Nvidia cuda software and gpu parallel computing architecture," vol. 7, 01 2007, pp. 103–104.

[16] F. Witherden, A. Farrington, and P. 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, nov 2014. [Online]. Available: https://doi.org/10.1016/j.cpc.2014.07.011

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[18] J. Regier, K. Pamnany, K. Fischer, A. Noack, M. Lam, J. Revels, S. Howard, R. Giordano, D. Schlegel, J. McAuliffe, R. Thomas, and Prabhat, "Cataloging the visible universe through bayesian inference at petascale," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 44–53.

[19] D. Blankenberg, G. Von Kuster, E. Bouvier, D. Baker, E. Afgan, N. Stoler, J. Taylor, A. Nekrutenko, and G. Team, "Dissemination of scientific software with galaxy toolshed," *Genome Biology*, vol. 15, no. 2, p. 403, Feb 2014.

[20] B. Ramsundar, P. Eastman, P. Walters, V. Pande, K. Leswing, and Z. Wu, *Deep Learning for the Life Sciences*. O'Reilly Media, 2019.

[21] A. K. Ghosh and S. M. Deshpande, "Least squares kinetic upwind method for inviscid compressible flows," *AIAA paper 1995-1735*, 1995.

[22] S. M. Deshpande, "On the Maxwellian distribution, symmetric form, and entropy conservation for the Euler equations," *NASA-TP-2583*, 1986.

[23] J. C. Mandal and S. M. Deshpande, "Kinetic flux vector splitting for Euler equations," *Comp. & Fluids*, vol. 23, no. 2, pp. 447–478, 1994.

[24] S. M. Deshpande, V. Ramesh, K. Malagi, and K. Arora, "Least squares kinetic upwind mesh-free method," *Defence Science Journal*, vol. 60, no. 6, pp. 583–597, 2010.

[25] R. Courant, E. Issacson, and M. Rees, "On the solution of nonlinear hyperbolic differential equations by finite differences," *Comm. Pure Appl. Math.*, vol. 5, pp. 243–255, 1952.

[26] S. M. Deshpande, P. S. Kulkarni, and A. K. Ghosh, "New developments in kinetic schemes," *Computers Math. Applic.*, vol. 35, no. 1, pp. 75–93, 1998.

[27] R. Soi, N. R. Mamidi, E. Slaughter, K. Prasun, A. Nemili, and S. Deshpande, "An implicitly parallel meshfree solver in regent," in *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*, 2020, pp. 40–54.

[28] J. F. B. M. Kraaijevanger, "Contractivity of Runge-Kutta methods," *BIT Numerical Mathematics*, vol. 31, no. 3, pp. 482–528, 1991.

[29] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: ACM, 2015, pp. 7:1–7:6.

[30] T. Oliphant, "NumPy: A guide to NumPy," USA: Trelgol Publishing, 2006. [Online]. Available: http://www.numpy.org/

[31] N. Corporation, "Developer Tools Documentation," 2021. [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

[32] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," 2009. [Online]. Available: https://www.osti.gov/biblio/1407078