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  
F.ASc, FNAE, FAcS 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 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 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

\[ \frac{\partial U}{\partial t} + \frac{\partial Gx}{\partial x} + \frac{\partial Gy}{\partial y} = 0 \]  

(1)

Here, \( U \) is the conserved vector, \( Gx \) and \( Gy \) 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,

\[ \frac{\partial F}{\partial t} + v_1 \frac{\partial F}{\partial x} + v_2 \frac{\partial F}{\partial y} = 0 \]  

(2)

where \( v_1^\pm = (v_1 \pm |v_1|)/2 \). 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 \( P_1 \in N(P_0) \),

\[ \Delta F_i = \Delta x_i F_{x_0} + \Delta y_i F_{y_0} + \frac{\Delta x_i}{2} (\Delta x_i F_{x_{xx}} + \Delta y_i F_{y_{yy}}) + O(\Delta x_i, \Delta y_i)^3, \quad i = 1, \ldots, n \]  

(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 \( P_0 \), \( N(P_0) \) 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

\[ \Delta F_{x_0} = \Delta F_i - F_0 \]  

(4)

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

\[ \Delta F_i = \Delta x_i F_{x_0} + \Delta y_i F_{y_0} + \frac{1}{2} (\Delta x_i, \Delta y_i)^2 \]  

(5)

Define a modified perturbation in Maxwellians, \( \tilde{F}_i = F_i - F_0 \) so that the leading terms in the truncation errors of the formulae for \( F_x \) and \( F_y \) are of the order of \( (\Delta x_i, \Delta y_i)^2 \),

\[ \tilde{F}_i = \Delta F_i - \frac{1}{2} \Delta x_i, \Delta y_i \Delta F_{y_0} \]  

(6)

Using \( \tilde{F}_i \), eq. (5) reduces to

\[ \tilde{F}_i = \Delta x_i F_{x_0} + \Delta y_i F_{y_0} + O(\Delta x_i, \Delta y_i)^3, \quad i = 1, \ldots, n \]  

(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 \( \Psi \) - 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,

\[ \frac{dU}{dt} + \frac{\partial Gx^+}{\partial x} - \frac{\partial Gx^-}{\partial x} + \frac{\partial Gy^+}{\partial y} - \frac{\partial Gy^-}{\partial y} = 0 \]  

(8)

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

\[ \frac{\partial Gx^\pm}{\partial x} = \sum \Delta y_i^2 \Delta x_i \Delta Gx^\pm - \sum \Delta x_i \Delta y_i \Delta Gx^\pm \]  

\[ \sum \Delta x_i^2 \Delta y_i^2 - (\sum \Delta x_i, \Delta y_i)^2 \]  

(9)

The perturbations \( \Delta Gx_i^\pm \) are defined by

\[ \Delta Gx_i^\pm = \Delta Gx_i^\pm + \frac{1}{2} \left\{ \Delta x_i \frac{\partial Gx_i^\pm}{\partial x} + \Delta y_i \frac{\partial Gx_i^\pm}{\partial y} \right\} \]  

(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 $\Delta Gx^\pm$ may not have enough neighbours. Another drawback is that the numerical solution may not be positive as the distributions $F_i$ and $\tilde{F}_i$ need not be Maxwellians [9].

To preserve the positivity of the solution, $q$-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, $q$-variables can represent the fluid flow at the macroscopic level as the transformations $U \leftrightarrow q$ and $F \leftrightarrow q$ are unique. The $q$-variables in 2D are defined by

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

Using $q$-variables, a second-order accurate upwind method can be obtained by replacing $\Delta Gx^\pm_i$ in eq. (10) with $\Delta Gx^\pm_i (q_i) = Gx^\pm_i (\tilde{q}_i) - Gx^\pm_i (\tilde{q}_0)$. Here, $\tilde{q}_i$ and $\tilde{q}_0$ are the modified $q$-variables, defined by

$$\tilde{q}_{i,0} = q_{i,0} - \frac{1}{2} \left( \Delta x_i q_{i+1,0} + \Delta y_i q_{i,0} \right)$$ \quad (12)

Here $q_x$ and $q_y$ are evaluated to second-order using the least squares as

$$\begin{bmatrix} q_x \\ q_y \end{bmatrix} = \left[ \begin{array}{cc} \sum \Delta x_i^2 & \sum \Delta x_i \Delta y_i \\ \sum \Delta x_i \Delta y_i & \sum \Delta y_i^2 \end{array} \right]^{-1} \begin{bmatrix} \sum \Delta x_i \Delta \tilde{q}_x \\ \sum \Delta y_i \Delta \tilde{q}_y \end{bmatrix}$$ \quad (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 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 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 $q$-variables defined in eq. (11) while $q\_derivatives()$ computes the second-order accurate approximations of $q_x$ and $q_y$ 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

---

**Algorithm 1: Serial meshfree solver based on q-LSKUM**

subroutine q_LSKUM:

- call preprocessor()
- for $n \leftarrow 1$ to $n \leq N$ do
  - call timestep()
    - for $rk \leftarrow 1$ 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()
- cudaMemcpyToSymbol(CPU_data, GPU_data)
- for $n \leftarrow 1$ to $n \leq N$ do
  - kernel $\leftarrow$ grid, block $\gg$ timestep()
    - for $rk \leftarrow 1$ to $4$ do
      - kernel $\leftarrow$ grid, block $\gg$ q_variables()
      - kernel $\leftarrow$ grid, block $\gg$ q_derivatives()
      - kernel $\leftarrow$ grid, block $\gg$ flux_residual()
      - kernel $\leftarrow$ grid, block $\gg$ state_update(rk)
    - end
    - cudaMemcpy(residue)
  - end
- cudaMemcpyToSymbol(GPU_data, CPU_data)
- call postprocessor()
end subroutine
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 \texttt{flux_residual()} kernel calls the kernels to compute the split flux derivatives \( \frac{\partial q}{\partial x} \) and \( \frac{\partial q}{\partial y} \). 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 \( q \)-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, \( \text{AoA} = 2^\circ \). 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 \( q \)-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.

| Level Points | Fortran | C | Python |
|--------------|---------|---|---------|
| 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  | 8.4609 | 5.0600 |
| 6            | 20M     | 11.3130 | 4.4096  | 7.9278 | 4.9650 |
| 7            | 40M     | 12.2720 | 4.2573  | 7.8805 | 4.9350 |

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.

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 \texttt{flux_residual} kernel, followed by \texttt{q_derivatives}. Note that the run-time of \texttt{q_derivatives} kernel depends on the number of inner...
variables q
residual state
derivatives flux

Python
C

update
timestep
C
Python

code has the highest utilisation operations, the run-times are found to be negligible
host C
Python
C

code has the lowest occupancy. The tabulated code, being closer to the roofline, yielded the best
code as more time is spent in executing the flux-residual kernel. This resulted in higher RDP values for the
Fortran code. On the other hand, the Julia code has the lowest occupancy. 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

| Points | Code  | SM utilisation | Memory utilisation | Achieved occupancy | Registers per thread |
|--------|-------|----------------|--------------------|--------------------|----------------------|
| 0.625M | Fortran | 11.56          | 21.27              | 3.08               | 220                  |
|        | C      | 43.16          | 10.41              | 11.76              | 184                  |
|        | Coarse | 29.55          | 25.95              | 18.03              | 128                  |
|        | Julia  | 26.23          | 18.28              | 16.54              | 152                  |
| 5M     | Fortran | 41.70          | 21.57              | 3.10               | 220                  |
|        | C      | 45.78          | 11.34              | 12.03              | 184                  |
|        | Medium | 30.05          | 26.35              | 18.29              | 128                  |
|        | Julia  | 26.61          | 18.15              | 16.77              | 152                  |
| 40M    | Fortran | 11.68          | 21.49              | 3.10               | 220                  |
|        | C      | 43.58          | 9.15               | 12.03              | 184                  |
|        | Fine   | 30.31          | 26.58              | 18.33              | 128                  |
|        | Julia  | 27.10          | 18.24              | 16.76              | 152                  |

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

| Points | Code   | q_variables | q_derivatives | flux_residual | state_update | Achieved Registers per thread |
|--------|--------|-------------|---------------|---------------|--------------|-------------------------------|
| 0.625M | Fortran| 0.50%       | 25.73%        | 72.67%        | 0.82%        | 31 26 05 26                 |
|        | C      | 0.77%       | 44.76%        | 50.51%        | 1.87%        | 03 184 15 12 18             |
|        | Coarse | 0.67%       | 37.48%        | 59.73%        | 1.47%        | 35 184 15 12 18             |
|        | Julia  | 1.24%       | 24.52%        | 71.71%        | 1.89%        | 34 184 15 12 18             |
| 5M     | Fortran| 0.42%       | 25.60%        | 72.95%        | 0.74%        | 41 228 03 184 03           |
|        | C      | 0.80%       | 47.34%        | 47.68%        | 1.84%        | 28 160 03 184 03           |
|        | Medium | 0.60%       | 38.43%        | 59.10%        | 1.38%        | 35 184 15 12 18             |
|        | Julia  | 1.37%       | 24.46%        | 71.77%        | 1.85%        | 34 184 15 12 18             |
| 40M    | Fortran| 0.41%       | 25.38%        | 73.21%        | 0.74%        | 03 184 15 12 18             |
|        | C      | 0.81%       | 42.27%        | 52.94%        | 1.85%        | 03 184 15 12 18             |
|        | Fine   | 0.58%       | 38.19%        | 59.40%        | 1.35%        | 03 184 15 12 18             |
|        | Julia  | 1.32%       | 24.12%        | 72.11%        | 1.85%        | 03 184 15 12 18             |

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

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 ↔ 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 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.

Fig. 1: Speedup achieved by the GPU codes.

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
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 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.
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 DAMM, 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.

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 $G_{x}^+, G_{x}^-$, $G_{y}^+$, and $G_{y}^-$, 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 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.

Table XII shows a comparison of register usage per thread, achieved occupancy, and global sectors per request. 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.

| Points | Code | DFMA | IMAD | DMUL | IADD3 | DADD |
|--------|------|------|------|------|-------|------|
|        |      |      |      |      |       |      |
| 40M    | Python | 8.2769 | 14.1171 | 2.3879 | 4.1338 | 3.1966 |
| Julia  | 6.3009 | 6.8711 | 2.2617 | 2.6878 | 1.4201 |

Table XII: A comparison of register usage per thread, achieved occupancy, and global sectors per request.
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.

| Code                  | Registers per thread | Achieved occupancy | Global sectors per request |
|-----------------------|----------------------|--------------------|----------------------------|
| 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      | 10.10 – 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                       |

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 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 still 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.

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

| Points | Code                  | SM utilisation | Performance in TFLOPS | Arithmetic intensity |
|--------|-----------------------|----------------|-----------------------|---------------------|
| 4M     | Fortran - baseline    | 11.68          | 0.57                  | 44.89               |
| 4M     | Fortran - optimised   | 47.85 – 48.68  | 2.35 – 2.41           | 10.71 – 10.90       |
| 4M     | C - baseline          | 43.58          | 2.167                 | 32.00               |
|        | C - optimised         | 56.41 – 58.30  | 2.79 – 2.88           | 9.12 – 9.66         |
| Fine   | Python - baseline     | 30.31          | 1.3491                | 66.94               |
| Fine   | 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           | 4.91 – 7.93         |

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

| Points | Version | Fortran | C | Python | Julia |
|--------|---------|---------|---|--------|-------|
|        |         | RDP $\times 10^{-3}$ (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 |
| 0.625M |         | 11.5620 | 4.6673 | 8.6080 | 5.2800 |
| 0.625M |         | 4.5856  | 3.4616 | 5.2355 | 4.6900 |
| 4M     | baseline | 12.2720 | 4.2573 | 7.8805 | 4.9350 |
| 4M     | optimised| 4.3365  | 3.4100 | 5.1540 | 4.6825 |

163
Thread 1
Bank 1
Thread 2
Bank 1
Thread 3
Bank 1
Fig. 3: 4-way bank conflicts in baseline Fortran, Python and Julia GPU codes.

@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.

@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.

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 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.
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 (NV AITC) 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. Lopéz, 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 D. J. Baeder, “Extending a three-dimensional GPU RANS solver for unsteady grid motion and free-wake coupling,” AIAA Paper 2016-1811, 2016.
[3] J. 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. Zulfair, “A mixed precision multicolor point-implicit solver for unstructured grids on gpus,” in 2019 IEEE/ACM 9th Workshop on Irregular Architectures: Applications 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. Fiset, and B. De Sutter, “Effective extensible programming: Unleashing Julia on GPUs,” IEEE Transactions on Parallel and Distributed Systems, 2018.
[7] E. Slaugher, 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,” Sudhana, vol. 39, no. 1, pp. 149–164, 2014.