GPU Accelerated Meshfree Solvers for Compressible Flows

Nischay Ram Mamidi*, Kumar Prasun†, Srikanth C.S.‡ and Anil Nemili§
Department of Mathematics, BITS-Pilani, Hyderabad Campus, Hyderabad, 500078

S.M. Deshpande ¶
Engineering Mechanics Unit, JNCASR, Bengaluru, 560064

Keywords: `Python`, `Julia`, `Fortran`, GPUs, CUDA, LSKUM, Meshfree methods.

In this paper, we present the development of `Python`, `Julia` and `Fortran` based GPU accelerated meshfree solvers for two-dimensional inviscid compressible flows. The meshfree solver is based on the Least Squares Kinetic Upwind Method (LSKUM). The programming interface, Compute Unified Device Architecture (CUDA) is used to perform the calculations on the GPU device. To assess the computational efficiency of the GPU solvers and to compare their relative performance, benchmark calculations are performed on several levels of point distribution. To analyse the overall performance of the GPU solvers, a detailed investigation of the invoked kernels is presented. For computationally intensive kernel, an analysis of various performance metrics such as utilisation of streaming multiprocessors and memory, branch efficiency, occupancy and arithmetic intensity is presented.

I. Introduction

The Least Squares Kinetic Upwind Method (LSKUM) [1] is a meshfree solver for the numerical solution of Euler and Navier-Stokes equations of the compressible fluid flows. Over the years, LSKUM based meshfree solvers developed at Defence Research and Development Laboratory (DRDL) and National Aerospace Laboratories (NAL) have been continuously used to compute flows around a wide variety of configurations.

The meshfree solvers in these Laboratories are written in traditional programming languages like `C`, `C++` and `Fortran`. Porting these legacy codes to HPC platforms with rapidly evolving hardware is challenging as the developers need to tune their codes frequently. One way to circumvent this problem is to employ modern languages such as `Python` or `Julia`. These languages are known to be architecture independent with an added advantage of easy code maintenance and readability. Furthermore, `Julia` is designed for high performance computing of numerically intensive algorithms. Recently, a hybrid parallel CFD solver called `PyFR` [2] has been developed in `Python`. This code has been successfully tested on massively parallel modern hardware platforms scaling to Petaflops. In other works, a parallel code, `Celeste` [3], written in `Julia` is used to perform Petascale operations for astronomical applications.

It is therefore worthwhile to pursue research in the direction of developing hybrid meshfree solvers using `Python` and `Julia`. As a first step towards this objective, in this paper, we present the development of `Python` and `Julia` based serial and GPU accelerated versions of meshfree solvers for two-dimensional inviscid compressible flows. The computational efficiency of these solvers is compared with equivalent

*nischaymamidi@gmail.com
†prasunk2@gmail.com
‡srikanth.c.sathyanarayana@gmail.com
§anil@hyderabad.bits-pilani.ac.in
¶smd@jncasr.ac.in

Fortran based serial and GPU solvers. Furthermore, a detailed analysis of various performance metrics is presented.

II. Performance Analysis of Serial and GPU Meshfree Solvers

In this section, we present numerical results to assess the computational efficiency of the Python, Julia and Fortran based GPU accelerated meshfree Euler solvers. Note that in all versions of the meshfree solver, second order accuracy in space is achieved by q-LSKUM [4]. The temporal term is discretised with the four stage strong stability preserving Runge-Kutta third order accurate scheme [5]. All computations are performed with double precision on a Linux workstation, whose configuration details are presented in Table 1. Table 2 shows a list of compilers, flags and other specifications used to perform simulations. The Python GPU code uses Numba 0.44.1 [6] and NumPy 1.16.4 [7], while Julia GPU code uses libraries such as CUDAnative 2.2.0 [8] and CuArrays 1.0.2.

The test case under investigation is the inviscid fluid flow simulation around the NACA 0012 airfoil at Mach number, $M = 0.85$ and angle of attack, $AoA = 1^o$. For the benchmarks, six levels of uniformly refined point distributions are used. The coarsest distribution consists of $9,600$ points, while the finest distribution consists of $98,30,400$ points. Note that further point refinement is not pursued as the desired memory requirements exceed the available GPU resources.

	CPU	GPU
Model	Intel Xeon $E5-2698$ v4	Nvidia Quadro $M5000$
Cores	40 (20×2)	2048
Core Frequency	2.20 GHz	1.038 GHz
Global Memory	128 GB	8 GB
$L2$ Cache	5 MB	2 MB

Table 1 Configuration of the workstation used to perform simulations.

	Python	Julia	Fortran
Version	3.6.8	1.1.1	Fortran 90
Compiler	Intel	LLVM	PGI
Compiler version	2019.3	6.0+ patches	18.10
Compiler flags	-O3	-O3 –check-bounds=no	-O3 -Mcuda = rdc
CUDA version	10.0.130	10.0.130	10.0.130
NVIDIA driver	430.34	430.34	430.34

Table 2 List of compilers, optimisation flags and other specifications used to execute the GPU codes.

A. RDP Comparison of Serial Meshfree Solvers

To measure the performance of meshfree solvers, 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. The RDP values based on a single core CPU calculations are listed in Table 3. Compared to the `Fortran` code, the performance of the `Python` code is observed to be very poor as its RDP values are larger by an order of $O\left(10^2\right)$. This behaviour is anticipated as pure `Python` is an interpreted and dynamically typed language while `Fortran` is a compiled and statically typed language. On the other hand, while `Julia` is also a dynamically typed language, its compiler can statically type functions through multiple dispatch. Due to this, the RDP values based on `Julia` are better than `Python` but its performance is still slower than `Fortran`. In the case of `Fortran` code, the RDP values decrease continuously with increase in the size of point distribution. The RDP based on the `Python` code decreases upto third level and then increases. On the other hand, the RDP values from the `Julia` code increase with successive point refinement. A possible explanation for this behaviour could be that the number of live `Python` or `Julia` objects increases with the size of the point distribution. A significant amount of CPU time is spent in managing the memory allocation of these objects. Note that these objects are managed by an interface called Garbage Collector [9].

Number of points	`Python` serial code	`Julia` serial code	`Fortran` serial code
$160 \times 60 = 9600$	5.1525×10^{-3}	6.9167×10^{-5}	3.4612×10^{-5}
$320 \times 120 = 38400$	3.9926×10^{-3}	7.5807×10^{-5}	2.6630×10^{-5}
$640 \times 240 = 153600$	3.4258×10^{-3}	7.6146×10^{-5}	2.1607×10^{-5}
$1280 \times 480 = 614400$	3.9027×10^{-3}	9.6940×10^{-5}	1.8972×10^{-5}
$2560 \times 960 = 2457600$	4.5819×10^{-3}	14.7761×10^{-5}	1.8467×10^{-5}
$5120 \times 1920 = 9830400$	4.6491×10^{-3}	19.3568×10^{-5}	1.7929×10^{-5}

Table 3 Comparison of the RDP values for a single core CPU computation.

B. RDP Comparison of GPU Accelerated Meshfree Solvers

It is well-known that the overall performance of a GPU accelerated code depends on the number of threads employed per block. To find the optimal number of threads for the present algorithm and the hardware, we perform numerical experiments with 8, 16, 32, 64, 128 and 256 threads per block. Note that with 512 and 1024 threads, the simulations crashed due to lack of sufficient thread memory. Infact, a similar behaviour is observed with the `Julia` code while using 256 threads. Figures 1 to 3 show the variation of RDP values with the number of threads per block. These plots clearly show that for all levels of point distribution, 32 threads per block yields the best performance.

Table 4 shows a comparison of optimal RDP values based on the GPU accelerated meshfree codes. The tabulated values clearly show that the `Julia` GPU code shows better performance on the first two levels of distribution. On subsequent point distributions, `Fortran` GPU code exhibits superior performance. Perhaps, this could be due to better utilisation of streaming multiprocessors (SM) by the respective GPU codes on these levels. As far as the `Python` GPU code is concerned, a very significant improvement in its performance is observed over its serial version. In fact, on medium to fine levels of point distribution, its RDP values are comparable to the values obtained from the `Fortran` version. On the other hand, the performance of `Julia` GPU code is observed to be slower on medium to finer distributions. This could be due to the current implementation of the `Julia` code, which makes more calls to the global memory and thus increasing the latency significantly.

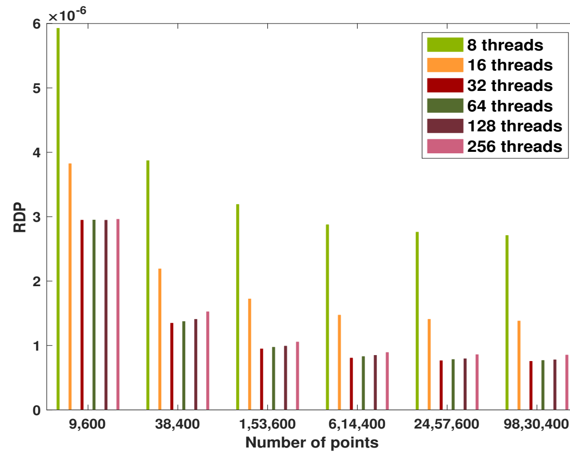

Fig. 1 Variation of `Python` GPU code RDP values with the number of threads per block.

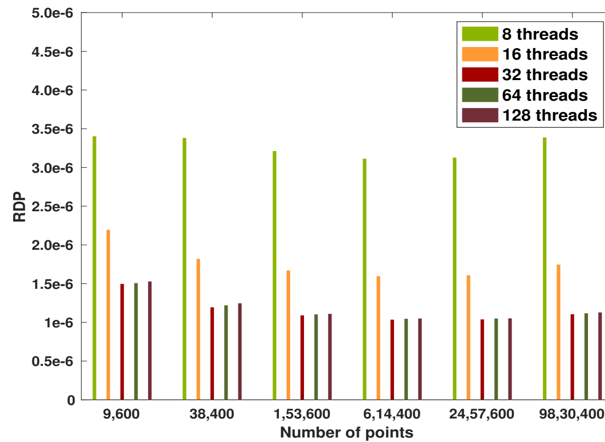

Fig. 2 Variation of `Julia` GPU code RDP values with the number of threads per block.

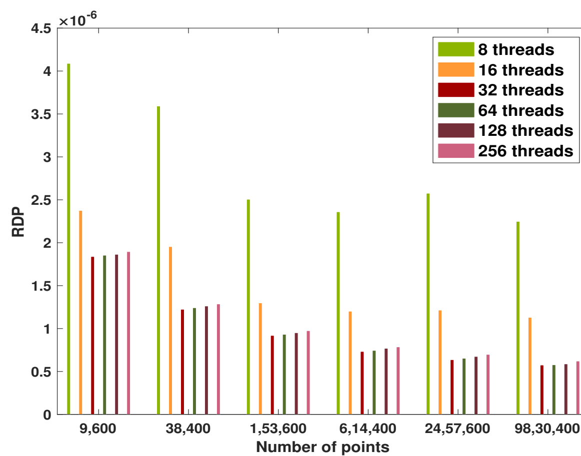

Fig. 3 Variation of `Fortran` GPU code RDP values with the number of threads per block.

Number of points	`Python` GPU code	`Julia` GPU code	`Fortran` GPU code
9600	29.5396×10^{-7}	15.0392×10^{-7}	18.3894×10^{-7}
38400	13.5419×10^{-7}	11.9791×10^{-7}	12.2463×10^{-7}
153600	9.5524×10^{-7}	10.9375×10^{-7}	9.1964×10^{-7}
614400	8.1265×10^{-7}	10.3841×10^{-7}	7.3291×10^{-7}
2457600	7.7198×10^{-7}	10.4166×10^{-7}	6.3746×10^{-7}
9830400	7.6221×10^{-7}	11.0880×10^{-7}	5.7493×10^{-7}

Table 4 Comparison of the RDP values based on the GPU accelerated meshfree codes.

C. Relative Speedup of the GPU Accelerated Meshfree Solvers

In order to assess the overall performance enhancement due to GPU computing, we define the speedup as the ratio of the RDP based on a single core CPU simulation to the RDP obtained using the GPU accelerated code with optimal number of threads per block. Table 5 shows the relative speedup achieved by the GPU codes. It can be observed that continuous point refinement enhanced the utilisation of GPU computing power and thus increased the speedup. Note that a reasonable explanation for massive speedup achieved by the `Python` GPU code is due to a very poor performance of its serial code.

Number of points	`Python` GPU code	`Julia` GPU code	`Fortran` GPU code
9600	1744.26	46.11	18.82
38400	2948.33	63.27	21.74
153600	3586.32	69.61	23.49
614400	4802.43	93.35	25.88
2457600	5935.25	141.85	28.97
9830400	6099.50	174.57	31.18

Table 5 Relative speedup of GPU accelerated codes over a single core CPU computation.

D. Performance Analysis of Various Kernels

To analyse the overall performance of the GPU accelerated meshfree solvers, it is imperative to investigate the kernels employed in the solvers. Following is the list of kernels invoked by all versions of the meshfree solvers. Note that these kernels are invoked in the order they are written below.

- `host` → `device`
- `q_variables`
- `q_derivatives`
- `flux_residual`
- `state_update`
- `residue`
- `device` → `host`

Here, the kernel `q_variables` transforms the primitive variables to the entropy variables. The kernel `q_derivatives` computes the spatial derivatives of the entropy variables [4]. The derivatives are then used in the defect correction method [10] for second order accuracy of the spatial derivatives in the governing fluid flow equations. To measure the performance metrics of the kernels, NVIDIA visual profiler Nsight Compute is used to profile the numerical simulations on all levels of point distribution. Table 6 shows the relative run time incurred by invoking the kernels. Note that 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 profile data is generated by fixing the number of iterations in the GPU solvers to $4,000$.

From this table it is clear that a very significant amount of run-time is taken by the `flux_residual` kernel, followed by `q_derivatives`. In the case of `Python` and `Fortran`, the relative run-times of the kernel `flux_residual` are decreasing continuously with continuous refinement in point distribution. On the other hand, for `Julia`, the run-times are decreasing upto the fourth level and then increasing gradually. This behaviour can be attributed to an increase in latency caused by more calls to the global memory. Note that the kernel `host → device` copies the entire data structure from host to the device, while the kernel `device → host` fetches only the primitive vector for post processing. Therefore, the run-time incurred by the kernel `host → device` is more than the time taken by `device → host`.

E. Performance Metrics of the Kernel: `flux_residual`

Table 7 displays several key performance metrics collected for the `flux_residual` kernel. These metrics are obtained with the optimal number of threads per block. From the tabulated values, we can infer that the Streaming Multiprocessors (SM) utilisation is higher for the `Fortran` GPU code. This could be due to the present implementation, which masks latency more efficiently for `Fortran`, compared to `Python` and `Julia`. In fact, it is also evident from lesser utilisation of memory by `Fortran` GPU code. It can also be observed that the SM utilisation increases for all versions and then stagnates on fine distributions. This stagnation is a direct consequence of the code creeping to the theoretical occupancy limit of 12.5%, possible on the present GPU configuration. Note that the current limit for theoretical occupancy is an outcome of the complexity of the `flux_residual` kernel, which requires more registers per thread. This results in a decrease in the number of active warps per SM, which implies low theoretical occupancy.

With regard to the arithmetic intensity, on the coarsest distribution, the `Python` GPU solver is *latency bounded* due to poor scheduling of the warps. This is due to the fact that the time taken for accessing the memory supersedes the kernel computational time. As we refine the point distribution, the computations become more intensive and the kernel becomes *compute bounded*.

Code	Relative run-time of the kernels					
	q_variables	q_derivatives	flux_residual	state_update	residue	device ↔ host
			Number of points = 160 × 60			
Python	0.2976%	2.9271%	95.8784%	0.8021%	0.0612%	0.0333%
Julia	0.4812%	5.3478%	92.6963%	1.3144%	0.0659%	0.1118%
Fortran	0.5675%	6.6615%	90.0145%	1.3028%	0.1145%	1.1508%
			Number of points = 320 × 120			
Python	0.5458%	5.0151%	92.8610%	1.4203%	0.0345%	0.1230%
Julia	0.7331%	6.0534%	91.4299%	1.6235%	0.0297%	0.1335%
Fortran	0.7544%	8.6123%	89.3245%	0.9963%	0.0947%	0.5478%
			Number of points = 640 × 240			
Python	0.8403%	6.1845%	90.7797%	1.9292%	0.0169%	0.2491%
Julia	0.8299%	5.9292%	91.3999%	1.6826%	0.0172%	0.1499%
Fortran	0.7159%	9.1141%	89.2249%	0.7962%	0.0788%	0.2347%
			Number of points = 1280 × 480			
Python	0.9807%	6.9040%	89.6051%	2.1973%	0.0101%	0.3025%
Julia	0.8607%	5.8395%	91.3628%	1.7155%	0.0125%	0.2181%
Fortran	0.7158%	9.4217%	88.9547%	0.7421%	0.0612%	0.1542%
			Number of points = 2560 × 960			
Python	1.0330%	7.1957%	89.1651%	2.3013%	0.0080%	0.2963%
Julia	0.8484%	5.8122%	91.5867%	1.5908%	0.0040%	0.1598%
Fortran	0.7124%	9.5386%	88.9217%	0.7312%	0.0417%	0.1171%
			Number of points = 5120 × 1920			
Python	1.0389%	7.3138%	88.7670%	2.4258%	0.0075%	0.4467%
Julia	0.7889%	5.3763%	91.8613%	1.5992%	0.0025%	0.3777%
Fortran	0.7216%	9.5832%	88.8417%	0.7346%	0.0371%	0.1245%

Table 6 Performance metrics of the kernels invoked by the GPU solvers.

Code	SM utilisation	Memory utilisation	Achieved occupancy	Theoretical occupancy	Branch efficiency	Arithmetic Intensity
Number of points = 160×60						
`Python`	58.17%	23.54%	10.87%	12.50%	99.34%	Latency-Bound
`Julia`	70.59%	40.85%	10.42%	12.50%	99.86%	Compute-Bound
`Fortran`	79.90%	20.25%	10.89%	12.50%	97.94%	Compute-Bound
Number of points = 320×120						
`Python`	80.37%	35.26%	11.77%	12.50%	99.67%	Compute-Bound
`Julia`	79.76%	49.61%	11.83%	12.50%	99.91%	Compute-Bound
`Fortran`	85.89%	21.39%	11.62%	12.50%	99.10%	Compute-Bound
Number of points = 640×240						
`Python`	87.01%	41.53%	12.26%	12.50%	99.86%	Compute-Bound
`Julia`	82.88%	53.48%	12.26%	12.50%	99.96%	Compute-Bound
`Fortran`	91.00%	22.36%	12.21%	12.50%	99.66%	Compute-Bound
Number of points = 1280×480						
`Python`	88.02%	43.09%	12.36%	12.50%	99.94%	Compute-Bound
Julia	83.05%	54.74%	12.39%	12.50%	99.99%	Compute-Bound
`Fortran`	92.88%	22.73%	12.37%	12.50%	99.88%	Compute-Bound
Number of points = 2560×960						
`Python`	87.30%	42.73%	12.40%	12.50%	99.97%	Compute-Bound
`Julia`	82.68%	54.96%	12.43%	12.50%	99.99%	Compute-Bound
`Fortran`	93.35%	22.84%	12.42%	12.50%	99.95%	Compute-Bound
Number of points = 5120×1920						
`Python`	86.46%	42.12%	12.41%	12.50%	99.99%	Compute-Bound
`Julia`	80.68%	46.40%	12.43%	12.50%	100.00%	Compute-Bound
`Fortran`	93.32%	22.83%	12.43%	12.50%	99.98%	Compute-Bound

Table 7 Performance metrics of the kernel `flux_residual` for optimal number of threads per block.

III. Conclusions

In summary, this paper presented GPU accelerated kinetic meshfree solvers for two-dimensional inviscid compressible flows. The GPU codes are written in `Python`, `Julia` and `Fortran` languages and their performances are analysed on six levels of point distribution, ranging from ten thousand to ten million points. Numerical results have shown that the GPU codes achieved impressive speedups over their serial counterparts. In terms of RDP, `Julia` has shown better performance on the coarse distributions. On medium to fine distributions, `Fortran` exhibited superior performance. Overall, the performance of the `Python` GPU code is comparable to that of `Fortran`.

The run-times incurred by various kernels employed in the GPU meshfree solvers was presented. A detailed analysis of several key performance metrics of the computationally intensive kernel was presented. These investigations have shown that all versions of GPU codes yielded similar performance in terms of branch efficiency, occupancy and arithmetic intensity. For this kernel the SM utilisation is observed to be higher for `Fortran` compared to `Python` and `Julia`. Furthermore, the memory utilisation of `Fortran` solver was observed to be relatively lower.

Further investigations are required to enhance the computational efficiency of the solvers by reducing the memory access from global and local memory to the shared memory of the GPU. Work is going on in this direction to improve the memory access. The long term aim is to develop LSKUM based `Python` and `Julia` meshfree solvers that can be run on hybrid platforms such as GPGPUs and CPUs+GPUs. Research is under progress to extend the present codes to hybrid parallel codes.

References

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

[2] F.D. Witherden, A.M. Farrington, and P.E. Vincent. PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications*, 185(11):3028–3040, nov 2014.

[3] 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)*, pages 44–53, May 2018.

[4] 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*, 40:47–62, 2002.

[5] J. F. B. M. Kraaijevanger. Contractivity of runge-kutta methods. *BIT Numerical Mathematics*, 31(3):482–528, 1991.

[6] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[7] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006.

[8] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[9] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. pages 1–116. Springer-Verlag, 1995.

[10] Suresh Deshpande, V. Ramesh, Keshav Malagi, and Konark Arora. Least squares kinetic upwind mesh-free method. *Defence Science Journal*, 60(6):583–597, Sep. 2010.