On the Performance of GPU Accelerated Meshfree Solvers in Fortran, C++, Python, and Julia

Nischay Ram Mamidi[1], Kumar Prasun[1], Dhruv Saxena[1], Anil Nemili[1]

Bharatkumar Sharma[2], SM Deshpande[3]

[1]Birla Institute of Technology and Science - Pilani, Hyderabad Campus, India

[2]NVIDIA

[3]524, Tata Nagar, Bengaluru, India (Formerly at JNCASR, Bengaluru)

NVIDIA GPU Technology Conference - 2021

April 12 - 16, 2021

Outline

Introduction

- Numerical simulations of fluid flow problems are computationally intensive
- For example, accurate capture of flow features around aircraft wings, flight vehicles, etc., do require simulations on fine grids with millions of grid points
- Existing parallel codes in CFD: CPU based (MPI) or GPU based (CUDA)
- Traditionally these codes are written in Fortran/C/C++

Introduction

- Alternatively, we can employ modern languages like Python or Julia

Advantages:

- Architecture Independent. Capable of running on any HPC platform
- Easy to maintain, high code readability, few lines of code
- New developers can quickly join and work on the code

Examples of Petascale parallel codes based on Python and Julia:

- PyFR - A compressible Navier-Stokes solver for unstructured grids (Python) (Witherden-2014)
- Celeste - An astronomical image analysis tool (Julia) (Regier-2018)

Introduction

- Alternatively, we can employ modern languages like Python or Julia

Advantages:

- Architecture Independent. Capable of running on any HPC platform
- Easy to maintain, high code readability, few lines of code
- New developers can quickly join and work on the code

Examples of Petascale parallel codes based on Python and Julia:

- PyFR - A compressible Navier-Stokes solver for unstructured grids (Python) (Witherden-2014)
- Celeste - An astronomical image analysis tool (Julia) (Regier-2018)

Objective of this research:

- Develop GPU accelerated meshfree solvers for inviscid compressible flows
- Written in Fortran/C++/Python/Julia
- Assess their relative performance

Meshfree q-LSKUM Solver for 2D Euler Equations

Least Squares Kinetic Upwind Method (LSKUM):

- Euler equations: Govern the inviscid compressible fluid flows

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{G}}{\partial x} + \frac{\partial \boldsymbol{H}}{\partial y} = 0$$

- Introduce upwinding using Kinetic Flux Vector Splitting (KFVS) (Mandal-1989)

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{G}^{+}}{\partial x} + \frac{\partial \boldsymbol{G}^{-}}{\partial x} + \frac{\partial \boldsymbol{H}^{+}}{\partial y} + \frac{\partial \boldsymbol{H}^{-}}{\partial y} = 0$$

- Basic idea of LSKUM: Approximate the spatial derivatives using Least Squares (Ghosh-1995)
- Input: Set of points and their neighbours (known as connectivity)
- Operates on structured, unstructured, cartesian, chimera point distributions, etc.
- Spatial accuracy: Using defect correction method + inner iterations, along with $\boldsymbol{q}$-variables (q-LSKUM) (Deshpande-2002)
- Time accuracy: Strong Stability Preserving Runge-Kutta Schemes (SSP-RK3)

Serial Pseudo Code

Algorithm 1: 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
```

GPU Accelerated Pseudo Code

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
        Compute kernel <<< grid, block >>> timestep
        for rk ← 1 to 4 do
            Compute kernel <<< grid, block >>> q_variables
            Compute kernel <<< grid, block >>> q_derivatives
            Compute kernel <<< grid, block >>> flux_residual
            Compute kernel <<< grid, block >>> state_update(rk)
        end
    end
    cudaDeviceToHost(GPU_data, CPU_data)
    call postprocessor()
end subroutine
```

Numerical Results

Test Case Details:

- Inviscid flow over a NACA 0012 airfoil
- $Ma = 0.63$ and $AoA = 2^o$
- Seven levels of point distributions: 0.625M to 40M

Language and Compiler Specifications

- Fortran 90 and C++ - NVIDIA HPC SDK 21.2
- Python $3.8.6$ - Numba $0.53.0$ and CUDA Toolkit $11.0.221$
- Julia $1.5.3$ - CUDA.jl $2.4.1$

Node Configuration

- Serial runs: AMD EPYC™ 7542 (2x32 cores) with 256 GB RAM
- GPU runs: NVIDIA Tesla V100 32GB (PCIe)

Performance of the naive GPU codes

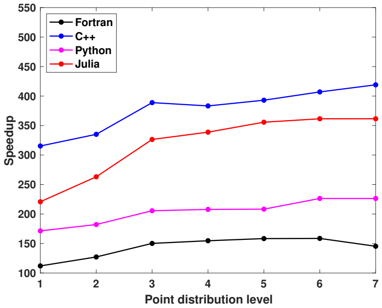
Level	No. of points	Fortran	C++	Python	Julia
RDP $\times 10^{-8}$ (Lower is better)					
1	0.625M	14.4090	5.1200	9.4183	7.3120
2	1.25M	12.8570	4.8800	8.9765	6.2160
3	2.5M	11.9100	4.6000	8.7008	5.4800
4	5M	11.5620	4.6673	8.6080	5.2800
5	10M	11.3640	4.5800	8.6409	5.0600
6	20M	11.3130	4.4096	7.9278	4.9650
7	40M	12.2720	4.2573	7.8805	4.9350

Comparison of the RDP values based on naive GPU codes.

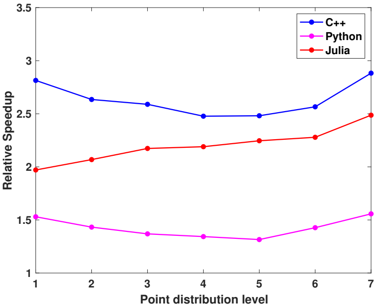
- RDP = Total wall clock time in seconds/No. of iterations/No. of points
- Number of iterations = 1000
- Optimal number of threads per block = 64

Performance of the naive GPU codes

Speedup of the GPU codes



Relative Speedup of the GPU codes



- Speedup of the GPU codes = (RDP of the serial Fortran code) / (RDP of the GPU codes)
- Relative speedup = (RDP of the Fortran GPU code) /(RDP of C++/Python/Julia GPU codes)

Naive GPU codes: Relative run-time of the Kernels

No.of 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%

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

Naive GPU codes: Performance metrics of the kernel - flux_residual

No.of points	Code	SM utilisation	Memory utilisation	Achieved occupancy
0.625M Coarse	Fortran	11.56	21.27	3.08
	C++	43.16	10.41	11.76
	Python	29.55	25.95	18.03
	Julia	26.23	18.28	16.54
40M Fine	Fortran	11.68	21.49	3.10
	C++	43.58	9.15	12.03
	Python	30.31	26.58	18.33
	Julia	27.10	18.24	16.76

A comparison of various performance metrics on coarse and finest point distributions.

- SM utilisation: Total utilisation of compute sub-systems (memory load/store operations, arithmetic and logic operations)
- Achieved occupancy: Total number of running warps / The theoretical maximum warps

Enhancing the Computational Efficiency of GPU Codes

Optimisation techniques employed:

- The profile reports have shown that the kernel flux_residual was latency bounded
- The kernel was split into smaller kernels. This resulted in reduced register pressure and thus increased occupancy
- Kernel splitting also reduced the warp stalls and increased the overall memory utilisation
- To further improve the memory utilisation, uncoalesced global memory access and shared memory bank conflicts were reduced
- These changes significantly increased the overall SM utilisation and FLOPS

Optimised GPU codes: Performance metrics of the kernel - *flux_residual*

Number of points	Code	SM utilisation	Memory utilisation	Achieved occupancy
	Fortran	11.68	21.49	3.10
	Fortran	47.85 – 48.68	41.85 – 43.49	17.84 – 18.10
	C++	43.58	9.15	12.03
40M	C++	56.41 – 58.30	33.25 – 34.70	17.81 – 18.10
Fine	Python	30.31	26.58	18.33
	Python	54.29 – 55.36	37.20 – 37.50	17.87 – 18.16
	Julia	27.10	18.24	16.76
	Julia	34.19 – 34.42	26.98 – 38.37	17.85 – 24.02

A comparison of various performance metrics on the finest point distribution.

- Tabulated metrics in the red color correspond to optimised GPU codes
- Metrics in the black color are from the naive GPU codes

Optimised GPU codes: Relative run-time of the Kernels

No.of points	Code	q_variables	q_derivatives	flux_residual	state_update
	Fortran	0.41%	25.38%	73.21%	0.74%
	Fortran	1.08%	50.89%	45.40%	1.95%
	C++	0.81%	42.27%	52.94%	1.85%
40M	C++	0.89%	49.28%	45.48%	2.02%
Fine	Python	0.58%	38.19%	59.40%	1.35%
	Python	0.93%	45.99%	50.17%	2.18%
	Julia	1.32%	24.12%	72.11%	1.85%
	Julia	1.54%	27.66%	67.94%	2.17%

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

- Run-times in the red color correspond to optimised GPU codes
- Run-times in black color are for the naive GPU codes

Optimised GPU codes: Performance metrics of the kernel - *flux_residual*

Number of points	Code	TeraFLOPS (double precision)
40M Fine	Fortran	0.5675
	Fortran	2.3547 – 2.4120
	C++	2.1664
	C++	2.7947 – 2.8830
	Python	1.3491
	Python	2.5794 – 2.6425
	Julia	1.3443
	Julia	1.6862 – 1.6990

Performance of GPU codes in terms of TFLOPS.

- Metrics in the black color are from the naive GPU codes
- Tabulated metrics in the red color correspond to optimised GPU codes

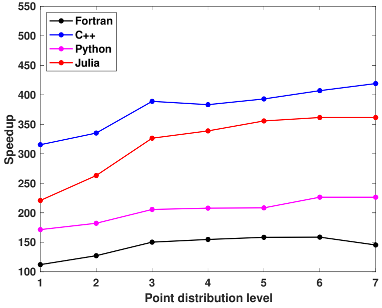
Performance of the optimised GPU codes

No. of points	Fortran	C++	Python	Julia
RDP $\times 10^{-8}$ (Lower is better)				
0.625M	14.4090	5.1200	9.4183	7.3120
0.625M	9.4446	4.0671	6.1372	7.5040
5M	11.5620	4.6673	8.6080	5.2800
5M	4.5856	3.4616	5.2355	4.6900
40M	12.2720	4.2573	7.8805	4.9350
40M	4.3365	3.4100	5.1540	4.6825

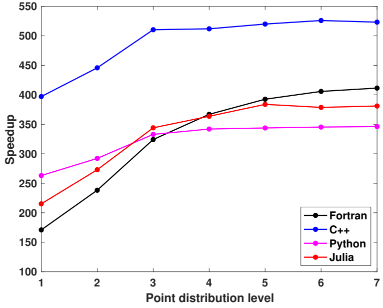
- Entries in the red color show the RDP values based on optimised GPU codes
- Entries in the black color show the naive GPU codes RDP values
- Optimal number of threads per block: 64 (Naive codes), 128 (Optimised codes)

Performance of the optimised GPU codes

Speedup of the naive GPU codes



Speedup of the optimised GPU codes



- Speedup of the GPU codes = (RDP of the serial Fortran code) / (RDP of the GPU codes)

Preliminary Investigations on A100 Card

Number of points	Code	RDP on V100	RDP on A100	Speedup on V100	Speedup on A100	Speedup factor
	`Fortran`	4.3365×10^{-8}	3.0838×10^{-8}	411.42	578.54	1.41
40M	`C++`	3.4100×10^{-8}	1.7582×10^{-8}	523.20	1014.74	1.94
Fine	`Python`	5.1540×10^{-8}	2.6415×10^{-8}	346.16	675.42	1.95
	`Julia`	4.6825×10^{-8}	2.9000×10^{-8}	381.02	615.21	1.61

Run-time comparisons of optimised GPU codes on V100 and A100 cards.

- Speedup factor of the GPU codes = RDP value on V100 / RDP value on A100

Conclusions & Future Work

Conclusions:

- Developed GPU accelerated meshfree compressible flow solvers in Fortran/C++/Python/Julia
- Benchmarked and analysed the performance of kernels
- The RDP values have shown that the C++ GPU code has exhibited superior performance

Future Work:

- Optimise other computationally intensive kernels (eg: q_derivatives)
- Extension to multi GPUs and three dimensional flows
- GPU accelerated discrete adjoint meshfree solvers for aerodynamic optimisation

Conclusions & Future Work

Conclusions:

- Developed GPU accelerated meshfree compressible flow solvers in Fortran/C++/Python/Julia
- Benchmarked and analysed the performance of kernels
- The RDP values have shown that the C++ GPU code has exhibited superior performance

Future Work:

- Optimise other computationally intensive kernels (eg: q_derivatives)
- Extension to multi GPUs and three dimensional flows
- GPU accelerated discrete adjoint meshfree solvers for aerodynamic optimisation

Thank you very much!