Performance Analysis of GPU Accelerated Meshfree Solvers in Fortran, C, Python, and Julia

Nischay Ram Mamidi[1], Dhruv Saxena[1], Kumar Prasun[2], Anil Nemili[1]

Bharatkumar Sharma[3], SM Deshpande[4]

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

[2]Courant Institute of Mathematical Sciences, New York, 10012, USA

[3]NVIDIA

[4]524, Tata Nagar, Bengaluru, India (Formerly at JNCASR, Bengaluru)

NVAITC Presentation Global 2022

March 31, 2022

Outline

Introduction

GPU Accelerated Meshfree Solver

Numerical Results

Conclusions & Future Work

Introduction

- Numerical simulation of fluid flow problems involving multi body configurations is computationally expensive
- Such simulations require solving the Euler/Naiver-Stokes equations on grids ranging from a few million to several billion grid points
- To perform these calculations, the CFD parallel codes use CPUs or CPU-GPUs
- GPUs: Alternative to CPUs in performance, cost, and energy
- GPUs consistently outperform CPUs in SIMD calculations
- Several CFD groups have developed GPU codes using Fortran/C/C++

Introduction

Modern languages such as Python, Julia, Regent, and Chapel have steadily risen in scientific computing

Advantages:

- Architecture Independent
- Easy to maintain, high code readability, few lines of code
- New developers can quickly join and work on the code

Implicit parallelism:

- Regent and Chapel support implicit parallelism
- Task division and data synchronisation are performed automatically

Examples of Petascale parallel codes:

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

Objective of this research

- A rigorous investigation and comparison of the GPU codes in traditional and modern languages has not yet been pursued
- In this research we present an analysis of GPU codes for 2D Euler equations
- The CFD solver is based on the meshfree q-LSKUM (Ghosh-1995, Deshpande-2002)
- Traditional languages: Fortran and C
- Modern languages: Python and Julia
- The programming model CUDA is used to construct the GPU solvers
- To investigate how the ecosystem of these languages has evolved

Objective of this research

- Acceleration of CFD codes starts with the implementation of the baseline code
- Baseline codes may not be computationally efficient
- Reasons: Poor memory access patterns, kernel launch configurations, size of the kernels, and redundant floating-point operation sequences
- To optimise the codes, baseline codes are profiled
- Profilers provide a guided analysis to understand the utilisation of the hardware
- Profiled data can be used to analyse performance metrics and identify bottlenecks
- Resolving these issues can enhance the computational efficiency
- This research highlights the importance of profiling and the cycle of analysis and optimisation

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: 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
```

GPU Accelerated Pseudo Code (Baseline)

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
```

Numerical Results

Test case details:

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

Language versions and compiler specifications:

- Fortran 90, C - NVIDIA HPC SDK 21.2
- Python $3.9.1$ - Numba $0.55.0$ and CUDA Toolkit $11.2.2$
- Julia $1.5.3$ - CUDA.jl $2.4.1$

Hardware configuration:

- Serial runs: AMD EPYC$^{\text{TM}}$ 7542 (2x32 cores) with 256 GB RAM
- GPU runs: NVIDIA Tesla V100 32GB (PCIe)

Performance of the baseline GPU codes

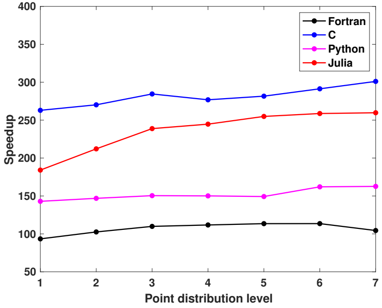
Level	No. of points	Fortran	C	Python	Julia
RDP $\times 10^{-8}$ (Lower is better)					
1	0.625M	14.4090	5.1200	9.4183	7.3120
2	1.25M	12.8570	4.8800	8.9765	6.2160
3	2.5M	11.9100	4.6000	8.7008	5.4800
4	5M	11.5620	4.6673	8.6080	5.2800
5	10M	11.3640	4.5800	8.6409	5.0600
6	20M	11.3130	4.4096	7.9278	4.9650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

Comparison of the RDP values based on baseline GPU codes

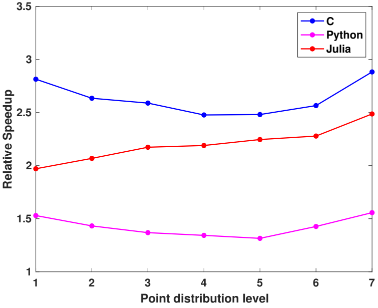
- RDP = Total wall clock time in seconds/No. of iterations/No. of points
- Number of iterations = 1000
- For Fortran, Python, and Julia lowest RDP is achieved with 64 threads per block. For C this value is 128

Performance of the baseline GPU codes

Speedup of the GPU codes



Relative Speedup of the GPU codes



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

Baseline 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

- Relative run-time of a kernel = (Kernel execution time) / (Overall time taken)

Baseline GPU codes: Performance metrics of the kernel - flux_residual

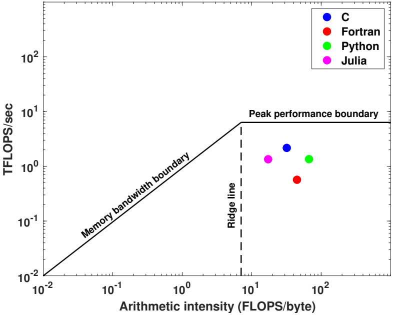
Points	Code	SM utilisation	Memory utilisation	Achieved occupancy	Registers per thread
		shown in percentage			
0.625M Coarse	`Fortran`	11.56	21.27	3.08	220
	`C`	43.16	10.41	11.76	184
	`Python`	29.55	25.95	18.03	128
	`Julia`	26.23	18.28	16.54	152
40M Fine	`Fortran`	11.68	21.49	3.10	220
	`C`	43.58	9.15	12.03	184
	`Python`	30.31	26.58	18.33	128
	`Julia`	27.10	18.24	16.76	152

- 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

Baseline GPU codes: Roofline Analysis

Roofline Model

- Shows a kernel's arithmetic intensity with its achievable performance
- Arithmetic intensity is defined as the number of FLOPs per byte of data movement
- Achieved performance is measured in TFLOPs
- A code with performance closer to the peak boundary uses the GPU resources optimally

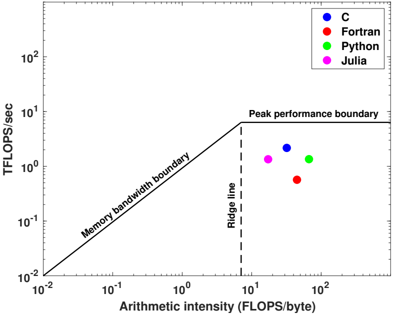


Roofline analysis of the `flux_residual` kernel

Baseline GPU codes: Roofline Analysis

Roofline Model

- Shows a kernel's arithmetic intensity with its achievable performance
- Arithmetic intensity is defined as the number of FLOPs per byte of data movement
- Achieved performance is measured in TFLOPs
- A code with performance closer to the peak boundary uses the GPU resources optimally



Roofline analysis of the `flux_residual` kernel

To investigate the difference in arithmetic intensity of Python and Julia codes we analyse the scheduler and warp state statistics

Baseline GPU codes: Scheduler State Statistics

- A warp is a collection of 32 threads executed simultaneously by an SM
- These warps are executed on the SM via a scheduler
- Scheduler states: GPU maximum warps, active, eligible, and issued warps
- GPU maximum warps: Maximum warps that can be issued per scheduler (For V100 it is 16)
- Active warps: Warps for which resources are allocated (Ex: registers, shared memory)
- Eligible warps: Subset of active warps that are not stalled
- Issued warps: Subset of eligible warps for which instructions are executed
- Note: Active warps $=$ Eligible warps $+$ Stalled warps

Baseline GPU codes: Scheduler State Statistics

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%

A comparison of scheduler statistics on the finest level of point distribution

To understand the low number of eligible warps we investigate the warp state statistics

Baseline GPU codes: Warp State Statistics

- There are several states for which warp stalls can occur
- In the present work, warp stalls due to no instruction, wait, and long scoreboards are dominant
- No instruction: Occurs when a warp is waiting to get selected to execute the next instruction
- It can also happen due to instruction cache miss
- Wait: Warp stalls if it is waiting for a fixed latency execution dependencies (Ex: FMA, ALU)
- Long scoreboard: Occurs when a warp waits for the data from L1TEX (Ex: local / global memory units)

Baseline GPU codes: Warp State Statistics

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

A comparison of warp state statistics on the finest level of point distribution

Baseline GPU codes: Warp State Statistics

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

A comparison of warp state statistics on the finest level of point distribution

- These metrics did not reveal any conclusive evidence regarding the poor performance of Python over Julia
- To further analyse, we investigate the memory access patterns and pipe utilisation

Baseline GPU codes: Global 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

A comparison of global load and store metrics on the finest level of point distribution

- Global load: Operations which retrieve data from the global memory
- Global store: Operations which store data in the global memory
- Sector: An aligned 32 byte-chunk of global memory
- Sectors per request: The average ratio of sectors to the number of load / store operations

Baseline GPU codes: Shared Memory Access Patterns

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

A comparison of shared memory bank conflicts due to load and store operations

- Bank conflict occurs when multiple threads in a warp access the same memory bank

Baseline GPU codes: Pipeline Utilisation

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

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

A comparison of various instructions executed by an SM

Baseline GPU codes: Summary

Summary on the performance of baseline GPU codes:

- The C code with better utilisation of SM has the lowest RDP
- Fortran code with very low occupancy has the highest RDP
- Python code has better SM utilisation and achieved occupancy
- However, it suffers from global memory coalescing, shared memory bank conflicts, excessive utilisation of FMA and ALU pipelines
- Due to this the RDP of Python is significantly higher than Julia

Enhancing the Computational Efficiency of GPU Codes

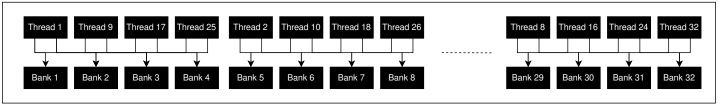
Optimisation techniques employed:

- For baseline codes the register usage of the kernel flux_residual is very high
- This indicates that the size of the kernel is too large
- This kernel is split into smaller kernels that compute the spatial derivatives of $\boldsymbol{Gx}^{\pm}$, $\boldsymbol{Gy}^{\pm}$
- This resulted in reduced register pressure and thus increased occupancy
- Kernel splitting also reduced the warp stalls and increased the overall memory utilisation

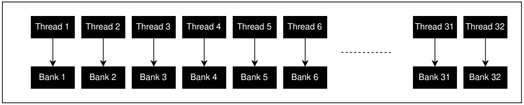
Enhancing the Computational Efficiency of GPU Codes

Language specific optimisation techniques:

- In baseline Fortran, Python, and Julia codes, thread index is used to access the values of the variables stored in shared memory
- This leads to shared memory bank conflicts



- In the optimised codes, both the thread index and block dimensions are used to access the shared memory



- In C code, implementation of shared memory deteriorated the performance

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

Code	Registers per thread	Achieved occupancy	Global sectors per request: Load	Global sectors per request: 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

Comparison of the metrics using baseline and optimised codes on the finest point distribution

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

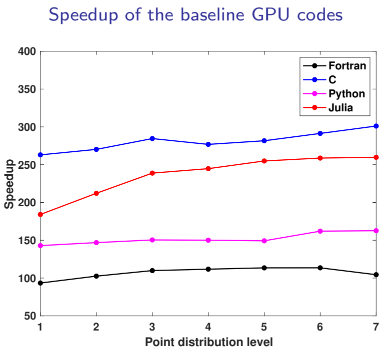
Optimised GPU codes: Performance metrics of the kernel - *flux_residual*

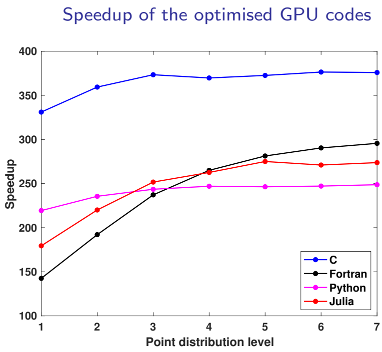
Points	Code	SM utilisation	Performance in TFLOPS	Arithmetic intensity
	`Fortran` - baseline	11.68	0.57	44.89
	`Fortran` - optimised	47.85 – 48.68	2.35 – 2.41	10.71 – 10.90
	`C` - baseline	43.58	2.167	32.00
40M	`C` - optimised	56.41 – 58.30	2.79 – 2.88	9.12 – 9.66
Fine	`Python` - baseline	30.31	1.3491	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

SM utilisation, performance, and arithmetic intensity of the baseline and optimised GPU codes

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

Performance of the optimised GPU codes





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

Preliminary Investigations on A100 GPU Card

Number of points	Code	RDP on V100	RDP on A100	Speedup factor
	`Fortran`	4.3365×10^{-8}	3.0838×10^{-8}	1.41
40M	`C`	3.4100×10^{-8}	1.7582×10^{-8}	1.94
Fine	`Python`	5.1540×10^{-8}	2.6415×10^{-8}	1.95
	`Julia`	4.6825×10^{-8}	2.9000×10^{-8}	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:

- Presented a performance analysis of baseline and optimised GPU meshfree solvers
- Highlighted the underlying software stack differences
- CUDA C exhibited superior performance, followed by Fortran
- With the advent of NVIDIA's CUDA Python and rapid developments in Julia's CUDA library, the performance gap of these languages with C/Fortran can be narrowed

Future Work:

- Comparing the performance of these GPU codes with Regent code
- Extending the meshfree solvers to three dimensional flows and multi GPUs
- GPU accelerated discrete adjoint meshfree solvers for aerodynamic optimisation

Conclusions & Future Work

Conclusions:

- Presented a performance analysis of baseline and optimised GPU meshfree solvers
- Highlighted the underlying software stack differences
- CUDA C exhibited superior performance, followed by Fortran
- With the advent of NVIDIA's CUDA Python and rapid developments in Julia's CUDA library, the performance gap of these languages with C/Fortran can be narrowed

Future Work:

- Comparing the performance of these GPU codes with Regent code
- Extending the meshfree solvers to three dimensional flows and multi GPUs
- GPU accelerated discrete adjoint meshfree solvers for aerodynamic optimisation

Thank you very much