GPU-Accelerated Computational Methods using Python, Fortran and CUDA

Author:

Thisal Mandula Sugathapala

Supervisor:

Lars Davidson

January 31, 2025

CHALMERS
UNIVERSITY OF TECHNOLOGY

1 Introduction

In fluid mechanics, multiphase flow refers to the simultaneous flow of two or more distinct thermodynamic phases, such as solid, liquid, and gas, each with its own continuous or dispersed phase characteristics. In the numerous fields of multiphase flow analysis, Lagrangian Particle Tracking (LPT) is a widely used technique for understanding and predicting the behavior of particles dispersed within a continuous fluid medium [1]. Researchers use LPT across various environmental engineering applications, including monitoring soot particle dispersion in the atmosphere [2] and tracking microplastic movement in marine ecosystems [3]. As a result, LPT has become invaluable in understanding pollutant behavior, improving environmental remediation strategies, and recommending policy decisions to mitigate the adverse impacts of pollutants in terrestrial and aquatic environments [4].

LPT models adopt a Lagrangian perspective, tracking individual particles as discrete entities moving inside a fluid domain. Unlike the Eulerian approach, which assumes a fixed reference frame for the fluid in space, the Lagrangian approach offers numerous advantages when particles are involved. An LPT approach reduces computational costs by eliminating the need to model or resolve the fluid domain. Such models use parametrized equations to capture flow properties and complex interactions between solid particles and the liquid phase, such as biofouling, stratification, and sedimentation. However, a significant obstacle to using LPT models to track many particles is the computational expense, which can substantially increase based on particle count. Traditionally, Central Processing Units (CPUs) have shown promise using Message Passing Interface (MPI) for parallelizing the workload for LPT models, achieving substantial reductions to simulation speed. Nevertheless, limited CPU cores in modern computers and supercomputing clusters hinder potential performance improvements. In comparison, recent technological advancements in Graphics Processing Units (GPUs) show the potential to overcome these limitations. While initially designed to render images and videos, including thousands of smaller yet efficient cores that can handle processes simultaneously shows tremendous potential for parallelizing large-scale, computationally intensive computational fluid dynamics (CFD) simulations. Specifically, LPT models can massively benefit from such parallelizing methods as thousands of particles can be distributed and tracked on individual threads of a GPU unit. Accordingly, this report aims to study the effectiveness of GPUs in handling LPT models by addressing the following objectives:

- Develop numerical implementations of a well-established LPT oceanographic model using Python and Fortran, optimized for execution on both CPUs and GPUs.

- Perform a comprehensive comparison of the LPT model's performance across each programming language (Python and Fortran) on CPU and GPU platforms.

2 Method

The numerical model used in this study was proposed by Kooi et al. [5] and is extensively used to track the fate of microplastics in the ocean. The model computes the sinking or rising velocity of the particle based on the difference between the particle's density and the surrounding seawater density. The settling velocity (V_{set}) is defined using the expression given below,

$$V_{set} = -\left[\frac{\rho_{tot} - \rho_{sw}}{\rho_{sw}} g \omega_{\star} \nu_{sw}\right]^{\frac{1}{3}} \tag{1}$$

where ρ_{tot} is the total density of the particle, ρ_{sw} is the density of sea water, ν_{sw} is the kinematic viscosity of sea water, and $\omega_{\star}$ is the dimensionless settling velocity. The total density of a biofouled microplastic particle depends on the density of the plastic type (ρ_{pl}) and biofilm density (ρ_{alg}). The equation used to calculate the total particle density is shown below,

$$\rho_{tot} = \frac{r_{pl}^3 \rho_{pl} + [(r_{pl} + t_{alg})^3 - r_{pl}^3]\rho_{alg}}{(r_{pl} + t_{alg})^3} \tag{2}$$

where r_{pl} is the original radius of the plastic particle, t_{alg} is the thickness of the biofilm. The biofilm thickness and weight is based on the number of algae attached (A) to the particle, calculated using the following differential equation.

$$\frac{dA}{dt} = \frac{\beta_A A_A}{\gamma_{pl}} + \mu_A A - m_A A - +Q_{10}^{\frac{T-20}{10}} R_A A \tag{3}$$

The first term on the right-hand side models the growth of the biofilm due to collision between algae and the particle and is dependent on several parameters, including the ambient algae concentration (A_A), the collision frequency (β_A), and the surface area of the particle (γ_{pl}). The second term represents algae growth (μ_A) limited by temperature and light intensity. The expression also captures the decrease of algae concentration due to mortality (m_A) and respiration rate (R_A). For more detailed information on the implementation of this model, please refer to Kooi et al. [5].

3 Results

3.1 CPU vs GPU

The LPT model is developed and implemented on Python and Fortran on CPU and GPU architectures while quantifying the simulation time with increased particle count. Such an approach aims to compare and contrast how LPT models perform with different programming languages and estimate enhancements to performance from porting a CPU code to a GPU. The particle number varied from 4 (2×2) to 16,384 (128×128) to assess the potential of scaling LPT models on CPU vs GPU. The following simulations were conducted on an NVIDIA GeForce RTX 3070 GPU. The results are illustrated in Fig. 1.

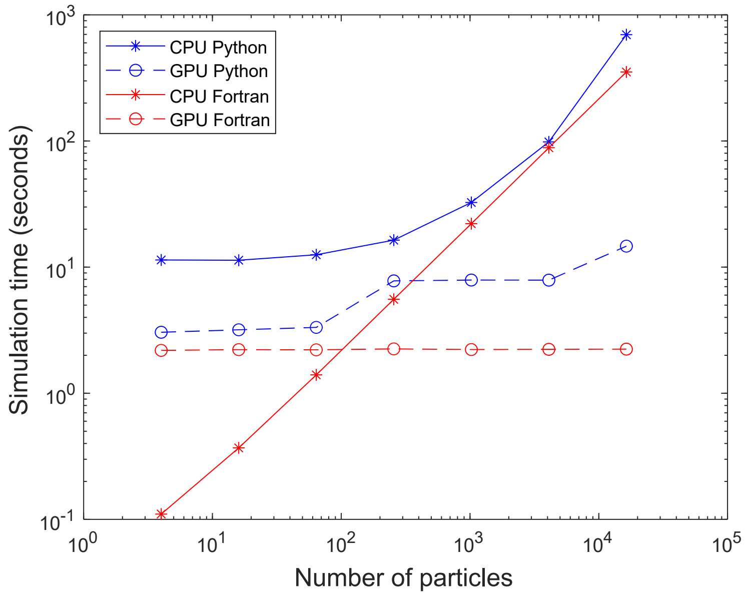

Figure 1: Total simulation time for a range of particle numbers with Python and Fortran running on CPU and GPU. Both axes are shown in logarithmic scale. The simulations were conducted on an NVIDIA GeForce RTX 3070 GPU.

The CPU Fortran code significantly outperformed all other configurations for a small particle count. These results show that Fortran, a native language, can conduct fast simulations with a small particle count. However, most realistic simulations would involve a much larger particle number, and under such conditions, the CPU version of the Fortran code performs and scales poorly. In contrast, the CPU Python version exhibited the poorest performance across all particle numbers. Nevertheless, when ported to CUDA, the Python code showed substantial performance improvements. Python and Fortran with CUDA showed excellent potential to scale with particle count, showing minimal increments to simulation time with an increase in particle numbers. CUDA-

Fortran outperformed CUDA-Python, though the difference was negligible for smaller particle numbers.

3.2 C3SE Vera Cluster GPU Performance Benchmark

Fig. 2 shows a performance benchmark of the GPUs available in the Chalmers C3SE Vera supercomputing cluster compared to a Nvidia GeForce RTX 3070 GPU.

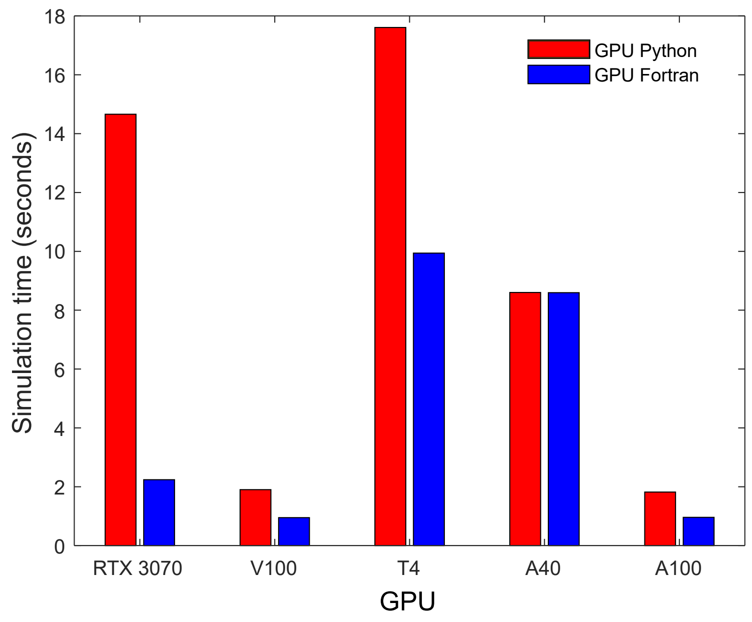

Figure 2: Performance benchmark of GPUs available for use in the Chalmers C3SE Vera supercomputing cluster against Nvidia GeForce RTX 3070. The performance of the LPT code with CUDA Python and CUDA Fortran is visualized.

Fig. 2 also illustrates the execution performance of the LPT code implemented using both CUDA Python and CUDA Fortran on four different types of GPUs:

- Nvidia V100 Tensor Core GPU
- NVIDIA Tesla T4 Tensor Core GPU
- NVIDIA A40 Data Center GPU
- NVIDIA A100 Tensor Core

The first observation is that the LPT performs better on V100 and A100 than on T4 and A40 models. Furthermore, the CUDA Python implementation significantly outperforms the RTX 3070 when benchmarked on V100 and

A100 GPUs, whereas the Fortran version achieves only marginal improvements on the same hardware. CUDA Fortran runs approximately twice as fast as CUDA Python across all benchmarked GPUs, except for the A40, where both implementations showcased comparable performance.

3.3 A100 vs RTX 3070

Figure 3 examines the potential of scaling on the A100 GPU in comparison to the results for RTX 3070, presented in Figure 1. The particle numbers are kept similar to those showcased in Figure 1. The results reveal similar performance trends observed with the RTX 3070. Specifically, the CUDA Fortran version showed near perfect scalability potential while the scalability potential of the model on CUDA Python also demonstrated significant potential but slightly diminishing for a large particle count. Furthermore, considerable performance improvements were observed for the CUDA Python implementation on the A100 when compared against the RTX 3070 GPU.

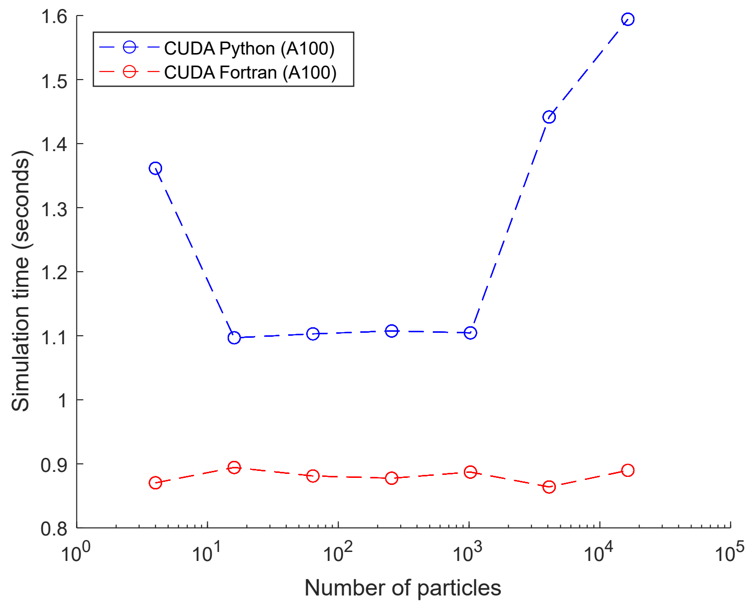

Figure 3: Total simulation time for a range of particle numbers with CUDA Python and CUDA Fortran on NVIDIA A100 Tensor Core GPU

3.4 Impact of Block Dimensions

When creating the threads and block structure for parellelizing the code and distributing the particles, this can be done in 1D, 2D, or 3D. While this is mostly relevant for decomposing a computational grid among threads, this can also be done for particles. This threading structure difference is shown below in Figure 4.

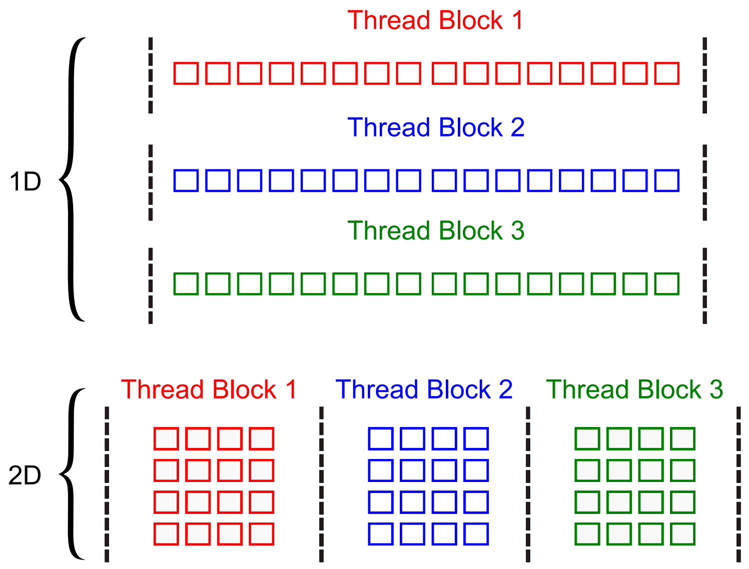

Figure 4: Visualization of threads distribution in blocks for a 1D and 2D thread block structure.

Figure 5 examines the performance of the LPT code on the A100 GPU when a 1D vs 2D threaded block structure is utilized for the simulations. To test the scalability potential of this study, memory allocation was restricted, where no results were saved on GPU device memory. This allowed the potential to simulate for 10,000 days. The final results indicate that the CUDA Python version remained unaffected by the block structure. This is likely due to Python libraries being optimized well enough not to be affected by thread block structure. In contrast, CUDA Fortran implementation showed high sensitivity to thread block structure, especially for small block sizes. The performance improvements observed with CUDA Fortran reduced for larger block sizes, eventually achieving similar performance between 1D and 2D versions for 1024 threads per block, which is the maximum threads allocated per block. To conclude, a block size of 256 threads per block was identified as optimal for CUDA Fortran, while both 256 and 512 threads per block yielded satisfactory performance for CUDA Python.

3.5 CUDA Fortran vs MPI Fortran

While parallelization on GPUs using CUDA Fortran significantly enhanced the simulation speed of the tested LPT code, the CPU-based Fortran version of the LPT code can also leverage multiple cores through MPI Fortran. Although there is no direct method to compare the performance of the LPT code on

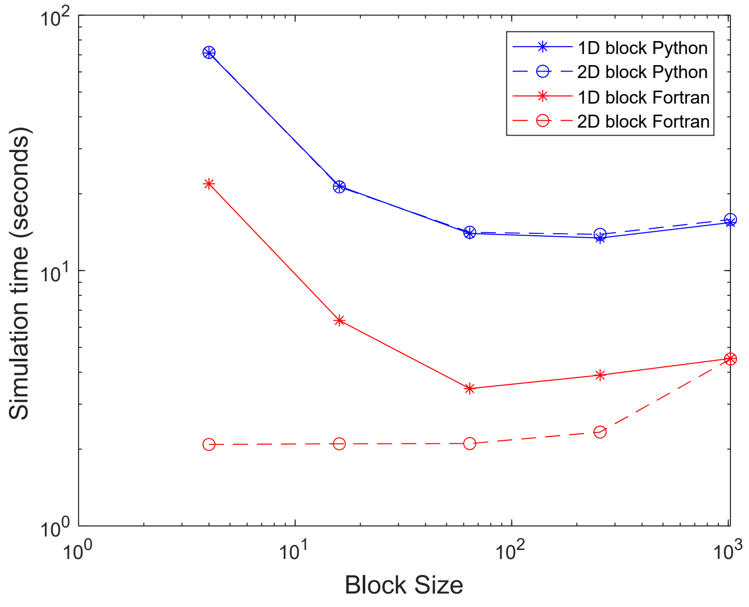

Figure 5: Visualization the of impact of thread block structure on LPT code performance with CUDA Python and CUDA Fortran performance.

GPUs versus multiple CPU cores, it is feasible to evaluate the potential savings in supercomputing core-hour resource allocation. In Sweden, the National Supercomputing Infrastructure (NAISS) is well-established and provides substantial resources to researchers. However, access to these resources is costly and highly competitive, making GPU computing an attractive option if it can achieve comparable results to CPU implementations while consuming fewer monthly resource hours. Therefore, in Fig. 1, the number of MPI cores required to match the simulation speed of a V100 GPU is compared. The results indicate that approximately 200 CPU cores are required to match the performance of a V100 GPU. On the Chalmers C3SE Vera supercomputing cluster, utilizing an A100 GPU costs 80 core hours per hour, while a V100 GPU costs 52 core hours per hour. Given the comparable simulation times for both GPUs when using the CUDA Fortran implementation of the LPT code, as illustrated in Fig. 1, running the LPT code on GPUs can potentially save 120 core-hours per hour with the A100 and 148 core-hours per hour with the V100. This represents significant resource savings compared to executing the MPI Fortran version of the code on CPUs.

3.6 Limitations of GPU computing

Porting CFD codes to GPUs has become increasingly popular due to the substantial performance gains GPUs can offer for parallelizable tasks. However, three main limitations impede the porting of all CFD codes to GPU. Firstly,

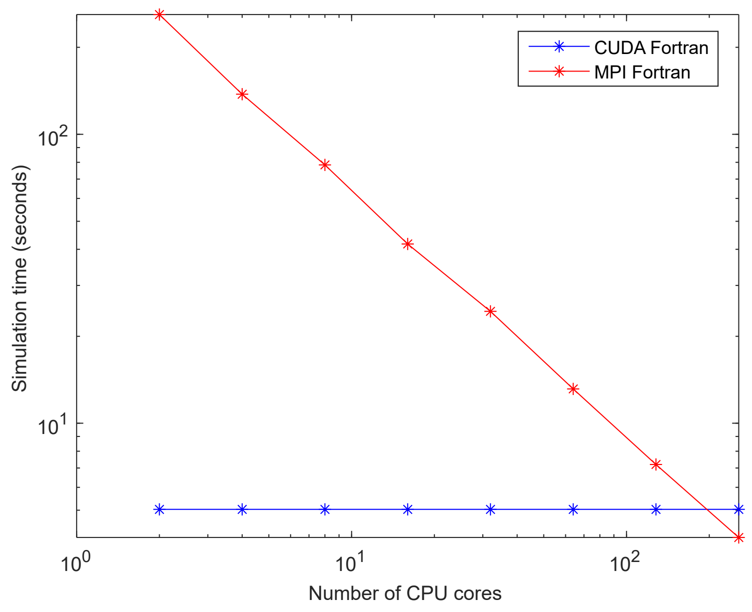

Figure 6: Visualization of impact of thread block structure on LPT code performance with CUDA Python and CUDA Fortran performance.

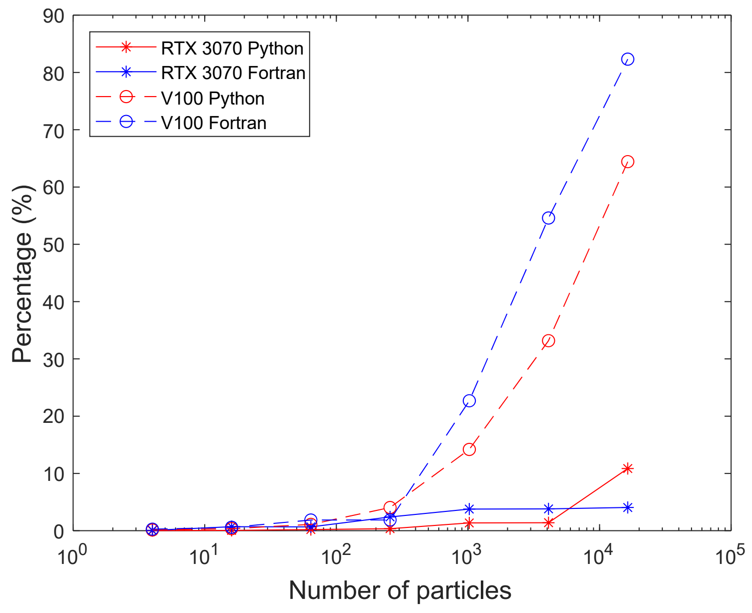

Figure 7: Visualization of impact of thread block structure on LPT code performance with CUDA Python and CUDA Fortran performance.

specific large-scale CFD codes with sophisticated particle and fluid interactions encounter challenges in achieving effective parallelization on GPUs. Secondly, data transfer overheads, including the frequent data movement between GPU and CPU memory for operations such as writing to files, can undermine the performance gains expected from GPU acceleration. This limitation is visualized in Fig. 7 where for the V100 GPU, transferring results from device to host can take up 80 % of total time. Such adverse impacts on simulation time should be accounted for when evaluating the potential of porting CFD codes to GPU. Thirdly, GPUs possess limited memory capacity compared to CPUs, as shown in Fig. 8. This limitation has shown restrictions on porting large-scale and high-resolution CFD codes that inherently demand large memory allocations to the GPU platform.

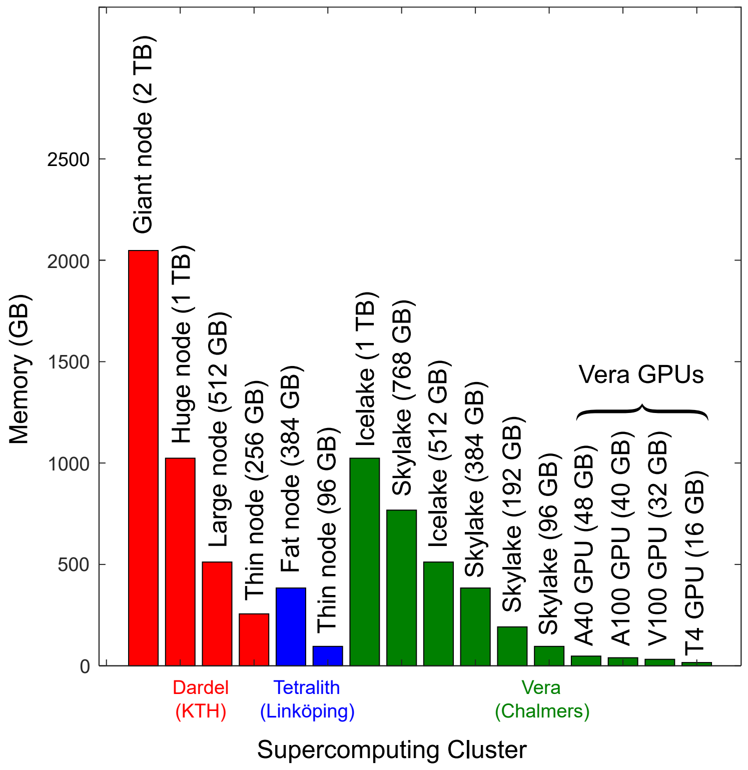

Figure 8: Primary memory available for a wide range of CPU nodes and GPUs avalaible in three supercomputing cluster: Dardel, Tetralith, and Vera.

4 Conclusion

This project investigates the potential for enhancing the performance of LPT models through GPU architecture optimization. The selected model is an oceanographic particle tracking code developed by Merel Kooi, designed to monitor microplastics in marine environments. The code was implemented in Fortran and Python using the CUDA platform to assess the potential of different programming languages to run the LPT model GPU architecture. The key findings are summarized below.

Performance Gains with GPU Acceleration:

- CUDA Fortran and CUDA Python implementations significantly outperformed their CPU counterparts with performance improvements up to 391 times and 350 times, respectively.
- Fortran on CPU demonstrated superior performance for simulations with low particle counts; however, such scenarios are uncommon in particle tracking applications.

Comparative Efficiency Between Programming Languages:

- CUDA Fortran achieved faster simulation times compared to CUDA Python.
- Both CUDA Fortran and CUDA Python exhibited strong scalability with increasing particle counts, resulting in only minimal rises in simulation time.

Impact of GPU Architecture:

- The choice of graphics card significantly influenced code performance.
- The thread block structure in CUDA Fortran affected simulation times. CUDA Fortran showed much better performance on 2D block structures.
- The optimum number of threads per block is 256 for both CUDA Python and CUDA Fortran.

In addition, this study further identified potential limitations to the widespread use of GPUs for CFD applications. These include, but are not limited to, the restrictive virtual memory allocation of GPUs, the slow data transfer between GPU and CPU memory, and the inability to write and parallelize highly complex CFD codes on GPU architecture. Nevertheless, GPU computing is a highly pursued and well-funded area of research because of its potential to surpass CPU code implementations. As a result, it is predicted that porting CFD codes to run efficiently on GPU will continue to receive a lot of attention with continuous improvements to GPU performance and architectures.

5 Acknowledgment

The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), particularly the Vera supercomputing cluster which is a part of Chalmers e-Commons e-Infrastructure. The GPU codes developed during this study are also open-sourced and publicly available at `https://github.com/thisalmandula/GPU_accelerated_LPT_CFD_code.git`.

References

[1] Giacomo Baldan, Tommaso Bellosta, and Alberto Guardone. "Efficient Lagrangian particle tracking algorithms for distributed-memory architectures". In: *Computers and Fluids* 256 (2023), p. 105856. ISSN: 0045-7930. DOI: `https://doi.org/10.1016/j.compfluid.2023.105856`.

[2] Lucien Gallen et al. "Lagrangian tracking of soot particles in LES of gas turbines". In: *Proceedings of the Combustion Institute* 37.4 (2019), pp. 5429–5436. ISSN: 1540-7489. DOI: `https://doi.org/10.1016/j.proci.2018.06.013`.

[3] Erik van Sebille et al. "Lagrangian ocean analysis: Fundamentals and practices". In: *Ocean Modelling* 121 (2018), pp. 49–75. ISSN: 1463-5003. DOI: `https://doi.org/10.1016/j.ocemod.2017.11.008`.

[4] J.R. Hunter. "The Application of Lagrangian Particle-Tracking Techniques to Modelling of Dispersion in The Sea". In: *Numerical Modelling: Applications to Marine Systems*. Vol. 145. North-Holland Mathematics Studies. North-Holland, 1987, pp. 257–269. DOI: `https://doi.org/10.1016/S0304-0208(08)70037-9`.

[5] Merel Kooi et al. "Ups and Downs in the Ocean: Effects of Biofouling on Vertical Transport of Microplastics". In: *Environmental Science & Technology* 51.14 (2017), pp. 7963–7971. DOI: `10.1021/acs.est.6b04702`.