Accelerating Lattice Boltzmann Simulations: Shared, Distributed and GPU Computing

Luke Johnson
Level 4 MSci. School of Physics, University of Bristol.
(Dated: November 27, 2024)

This study aimed to use multithreading, distributed memory systems, and GPU acceleration to optimise a Lattice Boltzmann simulation of car slipstreams. It was found that MPI could achieve a 43.5x speedup with 77.8% efficiency on 56 processes, outperforming shared memory approaches like OpenMP and Numba. However, the GPU's performance was unmatched, offering speedups of 4.09x over an entire node of MPI processes and 9,690x compared to standard Python.

1. INTRODUCTION

The objective of this experiment was to successfully recreate a Lattice Boltzmann simulation of fluid flowing past an obstacle using parallelization techniques to achieve meaningful optimisation and speedup. The investigation aimed to use multithreading, vectorization, and distributed computing to solve various Lattice Boltzmann equations and compare the efficacy and limitations of each approach.

In 1971, Intel released the first commercially available CPU, the 'Intel 4004' [1]. It featured a single physical core meaning tasks could only be executed serially. In 2001, IBM successfully installed two physical cores onto a single silicon chip. This led to the first multi-core processor, called 'Power4' [2], capable of running multiple programs simultaneously. Modern advancements have led to the proliferation of these multi-core systems, offering massive scope for accelerating computations through parallel programming paradigms.

Designing software to harness the full potential of contemporary hardware resources poses numerous challenges and might not always be feasible. Data dependencies require careful synchronisation and communication between cores, engendering overhead and latency. Minimising these costs is essential to achieve meaningful speedups. Unfortunately, it is often the case that some code must run single-threaded and act as a bottleneck. This limitation is quantified by Amdahl's Law,

$$S(N) = \frac{1}{1 - P + \frac{P}{N}}, \tag{1}$$

which states that the parallel speedup of a program is ultimately constrained by the fraction of code that must be executed serially [3]. In Eq. 1, if P is the proportion of the program which is parallelizable and you throw N cores at the problem, you can expect a speedup S.

2. THEORY

2.1. Lattice Boltzmann Method

The Lattice Boltzmann Method adopts a mesoscopic approach, discretising space into a lattice structure [4] to model the probability distribution f of particles in phase-space as a function of time.

By acquiring discrete moments of distributions, microscopic behaviour can be extrapolated to macroscopic systems. This process involves segmenting space and time into discrete intervals:

$$f_j(\mathbf{x}_i + \mathbf{c}_j\Delta t, t + \Delta t) = f_j(\mathbf{x}_i, t) + \Omega_j(f). \tag{2}$$

After each time step $t + \Delta t$, particles propagate to neighbouring sites $\mathbf{x}_i + \mathbf{c}_j\Delta t$ in the 2D lattice. Here, $\mathbf{x}_i$ denotes a particle's lattice position and $\mathbf{c}_j$ represents one of nine discrete velocity directions pointing to adjacent lattice nodes. The source term Ω encapsulates the conservation of mass and momentum and is defined by the Bhatnagar-Gross-Krook collision operator,

$$\Omega_j(f) = -\frac{f_j(\mathbf{x}_i, t) - f_j^{eq}(\mathbf{x}_i, t)}{\tau}\Delta t, \tag{3}$$

where τ is the relaxation time towards the equilibrium distribution f_j^{eq}.

Substituting Eq. 3 into Eq. 2 lets us split the process into the collision and streaming steps:

$$f_j^*(\mathbf{x}_i, t) = (1 - \frac{\Delta t}{\tau})f_j(\mathbf{x}_i, t) + \frac{\Delta t}{\tau}f_j^{eq}(\mathbf{x}_i, t) \tag{4}$$

$$f_j(\mathbf{x}_i + \mathbf{c}_j\Delta t, t + \Delta t) = f_j^*(\mathbf{x}_i, t). \tag{5}$$

The post-collision distribution f_j^* (calculated in Eq. 4) updates the distribution function based on local particle collisions. The streaming step in Eq. 5 transfers velocity information from the current site to its neighbours.

The net force acting on an obstacle can be calculated by applying Newton's 2nd Law [5], relating force to the rate of change of momentum. Summing over all lattice points at the obstacle boundary $\mathbf{x}_b$ yields the total force F exerted on the

obstacle by the fluid [6]:

$$\mathbf{F} = \sum_{all x_b} \sum_{\alpha \neq 0} \mathbf{e}_\alpha [f_\alpha(\mathbf{x}_b, t) + f_{\bar{\alpha}}(\mathbf{x}_b + \mathbf{e}_{\bar{\alpha}} \Delta t, t)]. \quad (6)$$

Here $\mathbf{e}_\alpha$ represents velocity in direction α and $\mathbf{e}_{\bar{\alpha}}$ the reflected velocity in the opposite direction. Using Eq. 6, the transverse force (corresponding to car drag) can now be extracted.

2.2. Shared Memory

Open Multi-Processing (OpenMP) is an API that allows multi-platform programming with shared memory systems. Execution begins with a single, master thread and additional threads are spawned when encountering parallel regions. After completion, all threads synchronise and the program continues on the master thread.

Each physical core on a multi-core processor is equipped with its own arithmetic logic unit, registers, and control unit [7] which allow it to fetch and decode instructions and perform operations independently. Effective parallel execution requires exploiting the microprocessor's memory hierarchy. The L1 cache is extremely low-latency and private to that core, making it ideal to hold frequently accessed instructions and data. Higher-level caches (L2, L3) and the memory controller, which accesses RAM via an interconnect [8], can be shared between cores but operate more slowly. Data sizes must be kept small and within cache capacity to avoid costly cache evictions and reloads.

Race conditions, which arise when multiple threads overwrite the same memory address concurrently, pose a major challenge in shared systems. OpenMP helps mitigate such issues by including thread-private variables and temporary views of shared variables stored in registers and caches, reducing repeated access to slower memory [9].

The main advantage of shared systems is the absence of communication overhead since all cores have access to the same memory. However, synchronisation can still act as a bottleneck: threads must wait for others to reach a defined point before proceeding, causing some to idle while others finish working. This cost is negligible compared to the message-passing latency in distributed systems; however, shared memory's scalability is limited by memory capacity, making distributed systems indispensable for massive workloads.

2.3. Distributed Memory

Distributed memory systems decentralise computation, restricting each processing unit to its private memory. This modularity makes them extremely scalable, as adding more nodes boosts both computational power and memory capacity, making them the preferred choice for large clusters.

Processors still need to communicate, which is facilitated by passing messages over a network using the Messaging Passing Interface (MPI) protocol. Data exchange between nodes involves packaging messages into buffers and waiting for the receiving rank to be ready. This leads to the main challenges of MPI: communication latency and bandwidth limitations. The time to pass messages can become a bottleneck in programs which require frequent or large messages. Non-blocking communication routines like `MPI_Isend` and `MPI_Irecv` can allow a process to initiate data transfer and continue working without waiting for the transfer to complete [10]. Care must be taken to ensure processes are synchronised, using mechanisms like `MPI_Barrier`, before reaching dependent parts of the program.

3. SIMULATION METHODS

3.1. Standard Python

The program began by calculating the velocity field $\mathbf{u}(\mathbf{x}, t)$ and the equilibrium distribution of the fluid in Eq. 3. It was then possible to account for the collisions and streaming at each time step by looping over each lattice point in accordance with Eq. 4 and Eq. 5 respectively. To incorporate obstacles, the streaming step (Eq. 5) needed modification to consider the possibility of being inside a solid boundary. A binary mask array was initialised with 0 values corresponding to empty lattice sites and values of 1 associated with obstacles. This governed whether the program performed streaming, reflection, or set the distribution to 0 (inside the obstacle). Finally, for each run of the program, a `.csv` file was produced which held the total drag force on the vehicle at each time step.

3.2. Vectorized Python

NumPy provides a C-API [11], which ensures contiguous memory allocation and better cache utilisation for arrays [12]. Functions like `np.einsum()` were used in fluid density, velocity, and equilibrium calculations to enable the simultaneous computation of entire array slices.

The standard Python version relied on nested loops which were incredibly slow due to Python's dynamic typing. Every loop iteration required Python to check variable types and execute interpreted bytecode. NumPy bypasses this overhead by using efficient linear algebra libraries such as BLAS and LAPACK [13], often through implementations like Intel's MKL [14]. These libraries exploit SIMD instruction sets (AVX, AVX2, AVX-512), to perform high-throughput, parallelized computations by processing multiple data points simultaneously within a single CPU cycle [15].

3.3. Cython

The Cython compiler translated the `.pyx` file into a `.c` file, which was then compiled into a shared library, `.so` file. This

removed Python's interpreter overhead. Cython allowed variables to be declared with C types, eliminating the runtime type checks. Memoryviews supplanted arrays for efficient data access from buffer-producing objects without copying them [16]. Each time a memoryview was accessed, Cython checked if it was initialised and if the index was in bounds or negative [17]. These checks were disabled for better performance. By default, Python division and modulo operations verify against cases like division by zero [18]. C-style division was adopted instead to improve performance.

3.4. OpenMP

OpenMP required the code to be compiled into C for compatibility. Memoryviews facilitated the seamless release of the Global Interpreter Lock (GIL), allowing threads to run concurrently. Key parameters were defined in a C-class whose attributes needed declaring in a separate `.pxd` file. By caching these parameters outside performance-critical loops and passing them in as arguments, costly class lookups were avoided leading to a dramatic 53.2x speedup. OpenMP's `prange()` directive was applied to the outermost loop over **x**. This ensured each thread processed rows in contiguous memory, maximising cache locality.

3.5. MPI

Work was distributed by splitting the lattice into vertical columns in the **x-y** plane, with each subdomain assigned to a distinct process. This maximised cache efficiency by aligning with Python's row-major memory layout [19]. Consequently, the boundaries were also contiguous which ensured more efficient MPI data transfers.

A 'halo exchange' method was used to handle the boundaries between processes [20]. The `MPI_Sendrecv()` command expedited bidirectional communication while adhering to periodic boundary conditions [21]. Isolating left and right exchanges mitigated the risk of deadlocks. Conveniently, the simulated wind tunnel had a smaller **y**-dimension which decreased halo sizes with the choice of domain partitioning (fewer elements to transfer between processes).

3.6. Numba

Numba is a Just-In-Time (JIT) compiler that uses an LLVM backend to convert Python bytecode to optimised machine code [22]. The most expensive functions were decorated with `@njit` which circumvents Python's interpreter. Loops could then be multithreaded and distributed across CPU cores. The `fastmath` directive unlocked floating-point optimisations by relaxing strict IEEE 754 compliance (standard for handling floating-point arithmetic) [23]. The GIL was released and cache files were saved for further performance gain.

3.7. CUDA

Numba's CUDA JIT compiler enabled GPU acceleration by rewriting functions as CUDA kernels. `cuda.grid()` was used to determine global thread indices for 2D and 3D data. Large arrays (e.g., u, f) were stored in global memory with coalesced access to minimise latency by ensuring memory requests within a warp (32 threads) were contiguous.

Shared, on-chip memory access is around 200 - 400x faster than uncached global memory access [24] and was employed in the streaming kernel to store partial force results for efficient block-wise reduction. Threads within a block began by performing partial sums in shared memory. Then, a hierarchical reduction algorithm was applied where threads iteratively combined results in strides, ensuring efficient intra-block communication and minimal warp divergence. This two-stage approach avoided expensive global synchronisation, improving scalability.

Localised quantities like `u_dot_c` were independently computed by threads, eliminating inter-thread dependencies and maximising GPU parallelism. To balance occupancy (number of active warps per streaming multiprocessor), shared memory, and register usage, thread blocks were configured as (16, 4, 16) for 3D kernels and (16, 4) for 2D kernels. Host-device data transfers were minimised to only occur during initialisation and visualisation.

4. RESULTS & DISCUSSION

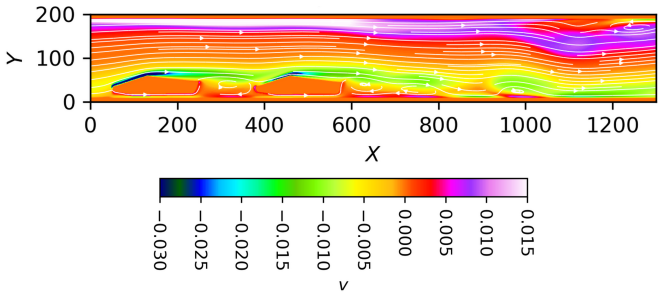

FIG. 1. Image showing the vorticity distribution in the lattice with overlayed velocity streamlines. Taken after 24 time steps with a Reynolds number of 17,280 and 3 MPI processes.

Fig. 1 shows the fluid's turbulent motion at a single time step. Similar images were created for all parallelization methods to ensure consistency. The program extracted transverse force data (corresponding to drag) on the rear car for a variety of separation distances and combined the results onto Fig. 2.

Fig. 2 shows that the single car without any slipstream faced the most drag. Furthermore, as the 2nd car got closer the relative force decreased, supporting the hypothesis that slipstreams provide better shielding from resistive forces. The lowest drag simulated was just 54% (at 2.92 cm) compared to that of a car not in any slipstream.

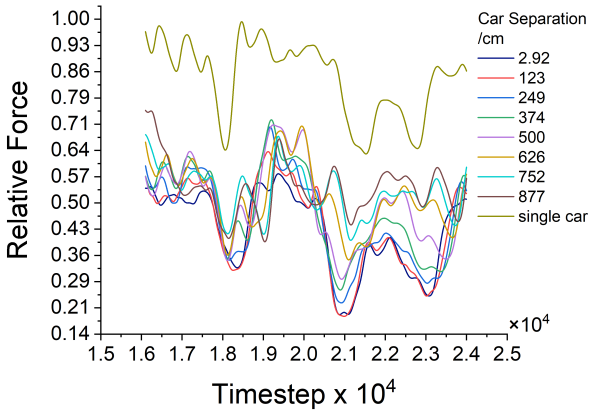

FIG. 2. Graph illustrating the relative force evolution through time for various car separation distances. A repeat was included for a single car without any slipstream (limit of separation $\to \infty$). Data was normalised to the car without any slipstream.

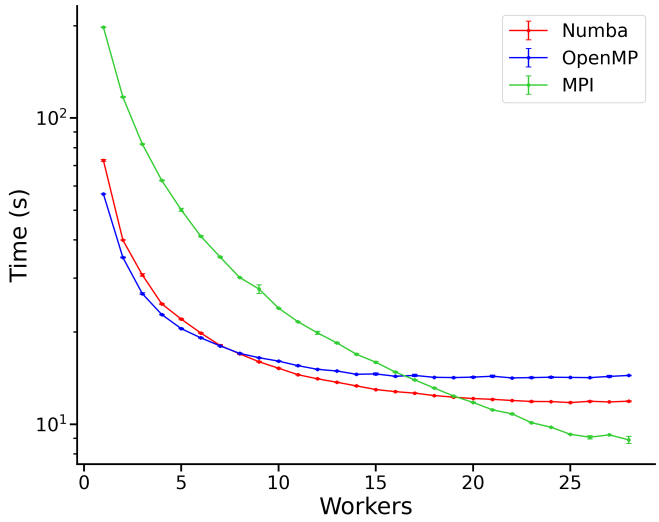

FIG. 3. Execution time on BlueCrystal 4 as a function of number of threads/processes for distributed and shared memory methods. Values are averages taken from 5 repeats. Error bars represent one standard deviation in the mean (plotted but extremely small). Simulation lasted 500 timesteps with a lattice size of 3200×200.

Fig. 3 demonstrates the effectiveness of parallelization by directly comparing the distributed and shared memory methods. As the number of workers increased, the execution time decreased in line with expectations. OpenMP running on 1 thread took 56.5 s, which was only slightly slower than native Cython (53.8 s). This 4.78% increase was primarily due to OpenMP's additional costs from managing thread pools and checking for parallel regions. Furthermore, it was clear that at 20 workers, MPI became the preferred method. OpenMP was the slowest implementation on a full node of BlueCrystal (28 threads), taking 14.4 s. Numba closely followed at 11.9 s, whereas MPI performed the best, only taking 8.89 s.

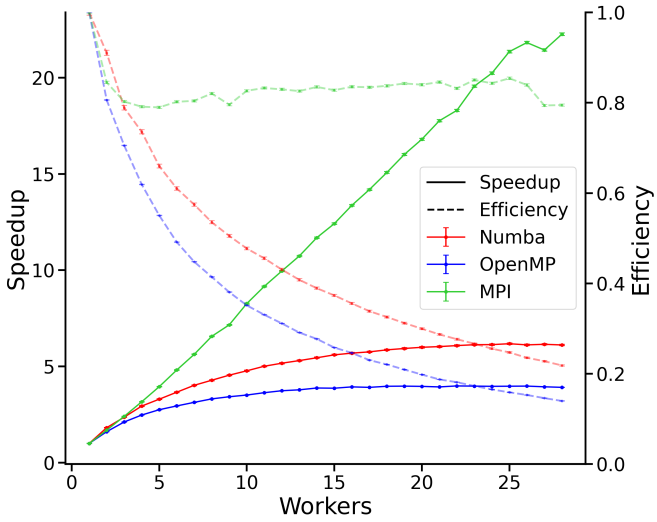

FIG. 4. Speedup (solid lines, left axis) and efficiency (dashed lines, right axis) of the program as a function of number of workers for distributed and shared memory methods. Values are averages from 5 repeats. Error bars represent one standard deviation (plotted but extremely small). Data collected on BlueCrystal 4. Simulated for 500 timesteps and a lattice size of 3200×200.

Fig. 4 presents the relative performance between the methods more clearly by showing the relative speed-up and efficiency. A perfect system would have a gradient of 1, meaning that 28 cores would speed up the program by 28x. At first glance the MPI results look impressive; however, it is important to note that due to MPI's message-passing protocol, packing and unpacking data buffers still occurs with 1 process. This caused 3.68x slower execution compared to Cython, as shown in Fig. 3, where the script took 198 s to run on 1 process. Nonetheless, MPI achieved a 22.3x speedup on 28 processes, offering an efficiency of 79.6%. The overhead from MPI's communication stack will remain constant when adding more processes, meaning the impact of the slow single-process time should diminish with larger clusters. Unfortunately, OpenMP and Numba offered little speedup beyond 5 threads, only managing 14.0% and 21.8% efficiency respectively at 28 threads.

The Numba and OpenMP computations were likely memory-bound as each thread needed to access and modify slices of the same large arrays, leading to cache contention. As threads shared the same memory space, changing elements within the same cache line triggered invalidations. These required reloads to propagate through the different levels of cache (L1, L2, L3) to maintain consistency between copies, a process known as cache coherence. This worsened latency and main memory traffic.

When operating with large array sizes (e.g. f, rho), the working set often exceeded the L3 cache size which forced frequent access to main memory. Since the number of requests the memory controller and bus can handle concurrently is finite, there was competition for limited memory band-

width, which was exacerbated at higher thread counts. This explains the inefficiencies of shared memory seen in Fig. 4 at larger numbers of threads.

In contrast, MPI computations (even on a single node) benefited from operating on subdomains with independent arrays. The separated memory spaces removed the need for maintaining cache coherence across processes. The localised memory access patterns led to reduced main memory transactions and more efficient use of memory bandwidth.

It is worth noting the "guided" schedule for OpenMP delivered the best results, surpassing the "static" and "dynamic" schedules by 9.73% and 15.4% respectively (at 28 threads). This likely stemmed from the load imbalance caused by conditional statements modifying calculations based on mask values (e.g. no computations are performed inside obstacle regions). The "guided" approach starts by assigning large chunks to each thread, ensuring high initial throughput. As threads complete their work, the chunk size dynamically reduces, allowing for near coincident finishing times while keeping the number of scheduling requests low. The overhead of the "guided" schedule was confirmed to be lower than the load imbalance due to its superior execution time.

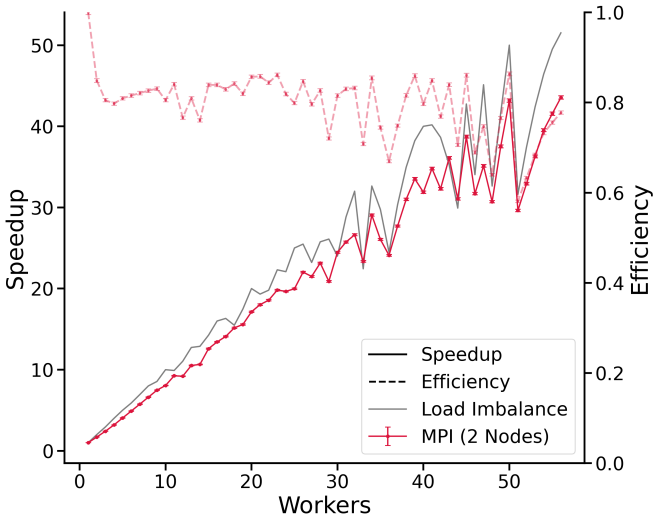

FIG. 5. Speedup (solid) and efficiency (dashed) MPI data across 2 nodes on BlueCrystal. Includes an additional line (grey) illustrating the theoretical effect of workload imbalance. Values are averages from 5 repeats. Error bars show one standard deviation (plotted but small). Data for 500 timesteps and lattice dimensions of 3200×200.

As the number of MPI processes was increased across 2 nodes on BlueCrystal, a speedup of 43.5x was achieved on 56 processes, but an oscillatory pattern emerged (depicted in Fig. 5). This was attributed to how the program assigned remainder lattice points to the last process when the lattice size was not divisible by the thread count. For comparison, a theoretical line was plotted with equation

$$y = (-0.01 \times \text{Remainder} + 1) \times \text{Workers}, \qquad (7)$$

where *Remainder* denotes the surplus $\boldsymbol{x}$ values assigned to the last process, scaled by a factor of (0.01) to fit the scale of the plot. Multiplying by *Workers* reflects the growing significance of this imbalance as the system achieves higher speedup. When fewer processes are used, the overall runtime is slower and the imbalance introduced by a small remainder is negligible. However, as the number of processes increases, the workload per process decreases. This imbalance then becomes a larger fraction of the total computational effort, reducing overall efficiency. The negative sign in the equation reflects the inverse relationship between workload balance and speedup and the $+1$ term ensures linearity in the absence of remainder. The slightly lower gradient of the MPI data is a consequence of imperfect efficiency. A future improvement of the program could distribute remainders as evenly as possible among the ranks to mitigate this effect.

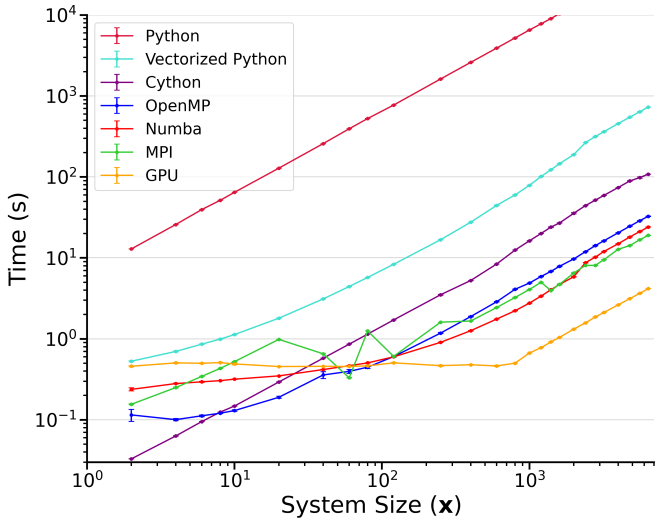

FIG. 6. Time taken for a node on BlueCrystal to execute 500 timesteps with different system sizes (log-log scale). Lattice size varied in **x** but kept at 200 in **y**. Numba, OpenMP and MPI methods all used 28 cores. Includes GPU acceleration on NVIDIA RTX 4070 Ti (orange). Values are averages from 5 repeats and error bars (plotted but extremely small) represent one standard deviation.

BlueCrystal incorporates 2 x Intel Xeon E5-2680 v4 processors per node, each featuring a 35 MB L3 cache. As depicted in Fig. 6, the scalability of the vectorized method slightly deviates from linear upon reaching a system dimension of 2000×200. This arises due to surpassing the cache size at that juncture. The memory requirement for a lattice configuration of 2800×200 is 44.86 MB which is greater than the 35 MB threshold of the system's L3 cache. The same load imbalance seen earlier in Fig. 5 reappears here, causing the MPI data (green) to fluctuate. This time, the number of cores was fixed at 28, meaning lower system sizes exacerbated this imbalance. This phenomenon became suitably small at lattice sizes around 1000×200. Lattice sizes above 10×200 and 40×200 were needed before Numba and OpenMP provided better results than Cython. This is due to the fixed overhead associated with initialising the threads,

which dominates at smaller workloads. Numba's JIT compilation is slower than OpenMP's precompiled model, explaining the disparity at smaller lattice sizes. However, at lattice sizes greater than 120×200, Numba appears to outperform OpenMP potentially due to the fastmath directive enabling aggressive floating-point optimisations (such as fused multiply-add FMA instructions).

TABLE I. NVIDIA RTX 4070 Ti Specifications [25].

Specification/Metric	**Value/Explanation**
Number of SMs	60
Number of Warps per SM	64
Threads per Warp	32 (defines warp size for scheduling)
Maximum Threads per Block	1024 ($x \times y \times z$ in thread configuration)
Threads per Block Config (2D)	$16 \times 4 = 64$
Blocks per Grid (2D)	$\left(\left\lceil \frac{\text{num\_x}}{16} \right\rceil, \left\lceil \frac{\text{num\_y}}{4} \right\rceil\right)$

GPU operations are always performed by a pack of 32 threads called a warp. The NVIDIA RTX 4070 Ti, as detailed in Table I, has 60 streaming multiprocessors (SMs), each capable of running up to 64 active warps (2048 threads) [25]. Using the threads per block and blocks per grid configurations from Table I, a lattice size of 800×200 clearly needs 2,500 blocks. Each block uses $64 \div 32 = 2$ warps, therefore the entire lattice requires 5000 warps. Using the warps per SM information on Table I, this fully saturates 78 SMs. The 4070 Ti houses only 60 SMs which means 1160 warps are queued for execution. For the previous data point (600×200) only 58 SMs are saturated, meaning the GPU would be underutilised. This explains why linear scaling with system size occurs at lattice sizes greater than 800×200 and smaller systems see no performance benefit on Fig. 6.

While shared memory was used for block-wise reduction in the force calculation, increasing its use for caching frequently accessed global memory data (e.g., $\boldsymbol{c}$ or $\boldsymbol{w}$) could improve performance further. Additionally, as previously mentioned with OpenMP, minimising conditional statements in the code could ameliorate warp divergence (when threads in the same warp take different execution paths) and lead to even better speeds.

The relative performance of all the different implementations is showcased in Fig. 7. The GPU achieved the maximum speedup of 9,690x compared to the standard Python script. The Intel i7 13700K CPU performed 2.64x better than the Intel Xeon E5-2680 v4 on average, as a result of its 1.6x higher boost clock frequency (5.4 GHz vs 3.3 GHz), newer architecture, and higher DDR5 memory speeds.

The profiling data in Table II highlights function-specific optimisations across implementations. NumPy vectorization yielded the greatest improvement in the fluid density function, achieving a 140x speedup by replacing a triple nested loop with a single `np.einsum` call. In contrast, the stream and reflect function (with its complex indexing, boundary conditions, and conditional logic) was harder to fully vectorize and

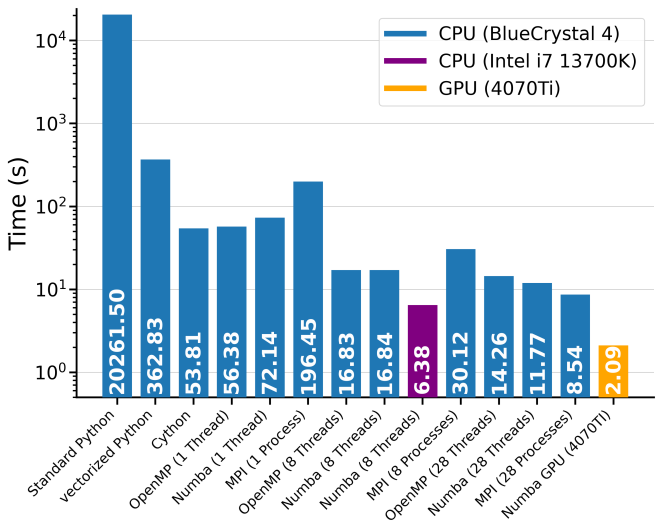

FIG. 7. Comparison of execution times for single and multithreaded programs. Performance was benchmarked on 2 x Intel Xeon E5-2680 v4 processors (blue) and an Intel i7 13700k (purple) as well as GPU acceleration using NVIDIA RTX 4070 Ti (orange). The minimum time was taken from 5 repeats. Data for 500 timesteps and a lattice dimension of 3200×200.

TABLE II. Function profiling per call. Data averaged over 10 repeats. CPU implementations were recorded on Intel i7 13700K with OpenMP, Numba and MPI using 8 cores.

Time (s)	**Collision**	**Stream & Reflect**	**Fluid Density**	**Fluid Velocity**	**Equilibrium**
Python	1.41	4.10	4.76e-1	3.28	4.74
Vectorized	1.74e-2	1.02e-1	3.41e-3	4.28e-2	9.52e-2
Cython	3.52e-3	1.85e-2	2.55e-3	9.03e-3	1.17e-2
OpenMP	1.74e-3	5.48e-3	1.08e-3	2.26e-3	3.94e-3
Numba	2.13e-3	8.16e-3	1.08e-3	2.10e-3	1.54e-3
MPI	5.12e-4	2.45e-2	3.98e-4	1.45e-3	1.89e-3

could not be collapsed into one straightforward computation. This limited it to a speedup of only 40.2x.

Cython excelled in the collision and equilibrium functions, providing 401x and 405x speedups respectively. These functions involved intense arithmetic within nested loops, which Cython could efficiently optimise. This advantage is less pronounced for simpler functions, which could be vectorized into a single NumPy call. Numba achieved the greatest speedup of 3,080x in the equilibrium function due to its even workload distribution and high arithmetic intensity. The collision function saw a smaller 662x speedup due to being memory-bound rather than compute-bound. In the MPI script, the stream and reflect function experienced a 1.32x slowdown relative to Cython due to its conditional logic and needing frequent access to halo layers. Additionally, a fixed inter-process communication cost of 8.17 ms, explains why at 8 processes, MPI remains slower than the shared memory methods.

5. CONCLUSION

Overall, the investigation successfully achieved its goal of parallelizing and accelerating a fluid dynamics simulation. Despite the limitations of memory bandwidth and cache contention faced by the shared memory methods, MPI showcased strong scalability, achieving a 43.5x speedup on 56 processes with an impressive 77.8% efficiency. This highlights the suitability of distributed memory systems for large workloads. GPU acceleration on the NVIDIA RTX 4070 Ti outperformed all other methods, achieving a 4.09x speedup over an entire node of MPI processes and an extraordinary 9,690x speedup compared to standard Python. Furthermore, the GPU's results match the hypothesis for underutilisation with smaller lattices. Further optimisation could address workload imbalances and warp divergences by restructuring conditional statements in the code.

6. REFERENCES

[1] *A Brief History of the Multi-Core Desktop CPU*, Matthew Bio, Techspot, (2021).

[2] *IBM Heritage, Advancing Humanity: The IBM Power4*, IBM, (2024).

[3] *Encyclopedia of Parallel Computing: Amdahl's Law*, Dr. John L. Gustafson, Intel Labs, (2011).

[4] *Computational fluid dynamics: Principles and applications*, Jiri Blazek, (2015).

[5] *Classical mechanics. Second ed.*, Tai L Chow, (2013).

[6] *Force Evaluation in the Lattice Boltzmann Method Involving Curved Geometry*, Renwei Mei, Dazhi Yu, and Wei Shyy, University of Florida, Gainesville, Florida. NASA Center for AeroSpace Information, (2002).

[7] *Step-by-Step Design and Simulation of a Simple CPU Architecture*, Derek C. Schuurman, Redeemer University College, (2013).

[8] *Handling the problems and opportunities posed by multiple on-chip memory controllers*, M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, A. Davis, School of Computing, University of Utah, (2010).

[9] *OPENMP API Specification: Version 5.1*, OpenMP Architecture Review Board, (2020).

[10] *Understanding the Behavior and Performance of Non-blocking Communications in MPI*, T. Saif, M. Parashar, European Conference on Parallel Processing, (2004).

[11] *Python Like You Mean It: "Vectorized" Operations: Optimized Computations on NumPy Arrays*, Ryan Soklaski, (2021).

[12] *SciPy and NumPy: An Overview for Developers*, Eli Bressert, O'Reilly Media, (2012).

[13] *What is BLAS and LAPACK in NumPy*, Jason Brownlee, Concurrent Numpy, (2023).

[14] *Intel® one API Math Kernel Library (oneMKL)*, Intel Corporation, (2024).

[15] *PGAS for Distributed Numerical Python Targeting Multi-core Clusters*, M. R. B. Kristensen, Y. Zheng, B. Vinter, International Parallel and Distributed Processing Symposium, (2012).

[16] *Cython: A Guide for Python Programmers*, Kurt W. Smith, O'Reilly Media, (2015).

[17] *Cython 3.1.0a1 documentation: Users Guide - Typed Memoryviews*, S. Behnel, R. Bradshaw, D. S. Seljebotn, G. Ewing, W. Stein, G. Gellner, et al, (2024).

[18] *Enhancements Division: CEP 516 - Division Semantics*, Robertwb, GitHub, Cython, (2009).

[19] *An Exhaustive Evaluation of Row-Major, Column-Major and Morton Layouts for Large Two-Dimensional Arrays*, J. Thiyagalingam, O. Beckmann, P. Kelly, Department of Computing, Imperial College, (2003).

[20] *An Interface for Halo Exchange Pattern*, Mauro Bianco, Swiss National Supercomputing Centre (CSCS), (2013).

[21] *Point to Point Communication Routines: Blocking Message Passing Routines*, Blaise Barney, Lawrence Livermore National Laboratory, (2015).

[22] *Definition of Numba*, NVIDIA, Data Science Glossary (2024).

[23] *Performance Optimization of Ice Sheet Simulation Models*, Fredrika Brink, Uppsala universitet, (2023).

[24] *CUDA Cuts: Fast Graph Cuts on the GPU*, V. Vineet, P. J. Narayanan, Centre for Visual Information Technology, International Institute of Information Technology, (2008).

[25] *NVIDIA GeForce RTX 4070 Ti Specs*, Techpowerup.com, GPU Database, (n.d accessed 2024)