Response to Reviewer #1

We sincerely thank the reviewer for the careful and thorough evaluation of our manuscript and for the highly positive assessment of our work. We are deeply encouraged that the reviewer finds the topic timely and relevant, the manuscript well organized, and the implementation technically sound.

We also greatly appreciate the reviewer's constructive and insightful comments, particularly regarding the interpretation of end-to-end performance limitations, hardware portability, and the clarity of our implementation details. These suggestions have helped us to significantly improve the rigor, transparency, and completeness of the manuscript.

In response, we have made substantial revisions, including:
(1) adding a detailed runtime distribution analysis (Table 10) to explicitly quantify the "bottleneck shift" toward the CPU-bound linear solver;
(2) expanding the discussion on performance portability and hardware-specific tuning across different GPU architectures;
(3) ensuring absolute consistency in the reported end-to-end speedups (8.57×) and explaining the nonlinear scaling behavior; and
(4) providing precise quantitative rationale for our kernel launch configurations (block sizes of 128 and 256) based on SM occupancy profiling.
All changes have been incorporated into the revised manuscript and are clearly indicated.

Below we provide a detailed, point-by-point response to each comment.

Major Comment 1:

The manuscript reports very high kernel-level speedups (up to ~900×), while the overall application speedup is approximately 8.57×. This discrepancy is expected in complex applications, but it is not sufficiently discussed in the current manuscript... The authors are encouraged to provide a clearer breakdown of the total runtime and to discuss the limiting factors for end-to-end performance.

Response:
We agree with your observation. To explicitly address this discrepancy, we have added a comprehensive runtime breakdown table (newly inserted Table 10) that quantifies the execution time and percentage of all core modules across three configurations: Baseline (1 CPU), 1 CPU + 1 GPU, and 4 CPUs + 1 GPU.

This new data clearly illustrates the "bottleneck shift" dictated by Amdahl's Law. After GPU offloading, the matrix assembly proportion shrinks to under 1% of the compute cycle. Consequently, the unaccelerated components—primarily the CPU-based PETSc linear solver and other CPU logic—surge to occupy over 85% of the execution time, which fundamentally limits the single-process end-to-end speedup (2.44×). Furthermore, this breakdown explains the

non-linear scaling achieved by the 4 CPUs + 1 GPU configuration (8.57×): domain decomposition among 4 MPI processes drastically reduces the dominant PETSc solver time (e.g., from 5.60 s to 2.19 s) due to improved cache locality, while the GPU matrix assembly remains negligible.

Changes in manuscript: We added Table 10 and a detailed discussion in Section 5.3.4 (Performance of functional modules):

"As detailed in Table 10, this configuration effectively addresses the 'bottleneck shift' dictated by Amdahl's Law. In the 1 CPU + 1 GPU setup, GPU matrix assembly is compressed to less than 1% of the total time, leaving the PETSc solver and other unaccelerated modules as the dominant bottlenecks. However, by leveraging 4 MPI processes, the absolute execution time of the PETSc solver is dramatically reduced from 5.60 s to 2.19 s..."

Major Comment 2:

The current implementation is closely tied to CUDA and NVIDIA GPUs... Given the growing importance of performance portability in geoscientific modeling, it would be helpful to clarify: which parts of the implementation are inherently hardware-specific; what level of effort would be required to adapt the approach to other architectures... A short discussion of the dependence on CUDA-specific features (for example, atomic operations and memory hierarchy)... would improve the completeness of the manuscript.

Response:
Thank you for this constructive suggestion. We have expanded our discussion on performance portability to distinguish between algorithmic generality and hardware-specific optimizations. The core strategies (one-thread-per-element mapping, data layout) are hardware-agnostic, and porting to platforms like AMD GPUs via HIP would require moderate development effort. However, achieving equivalent performance efficiency presents challenges, particularly regarding atomic operations. For instance, recent literature highlights that while NVIDIA A100 architectures natively excel at double-precision atomic updates in unstructured grids, AMD CDNA architectures may incur higher penalties and require alternative register-based aggregation.

Changes in manuscript: We added a dedicated paragraph addressing these specific points at the end of Section 2 (Target Scientific Application Definition):

"Although the current implementation is based on the CUDA programming model and NVIDIA GPUs, the proposed optimization strategies are largely hardware-agnostic at the algorithmic level. Specifically, the element-wise parallelization strategy... are conceptually general... Porting the implementation to other platforms, such as AMD GPUs, would primarily involve adapting CUDA-specific APIs to alternative frameworks such as HIP... However, achieving performance portability will require additional tuning to account for architectural differences... For instance, while NVIDIA V100 and A100 architectures natively deliver exceptional performance for double-precision atomic updates... evaluations on AMD CDNA architectures (e.g., MI100) have shown that standard atomic updates can incur higher penalties...."

Minor Comment 1:

The abstract reports an overall speedup of 8.57×, while later sections mention 10.28× under certain configurations. This inconsistency may cause confusion... The authors are encouraged to clearly specify the conditions under which each value is obtained and ensure consistent reporting.

Response:
We sincerely apologize for this data inconsistency, which was an oversight carried over from an earlier draft. We have strictly standardized the overall speedup data across the entire manuscript based on the rigorous testing reported in Table 9.

Changes in manuscript: The "10.28×" figure has been corrected to 8.57× throughout the manuscript (including the Abstract, Introduction, and Conclusions). The exact configuration (4 MPI processes + 1 GPU) is explicitly stated alongside the speedup value.

Minor Comment 2:

Thread block sizes are described as being “empirically tuned,” but no further details are provided. It would be useful to: report the configurations used in the experiments; briefly indicate how they were selected (e.g., through profiling tools); comment on sensitivity to these parameters, if relevant.

Response:
We appreciate this request for implementation details. To improve reproducibility, we have quantified our configuration selection process. In our implementation, block sizes of 128 and 256 were both utilized depending on the register footprint of the individual kernel. We added a quantitative calculation to demonstrate that for our mesh size (554,394 elements), both 128 and 256 threads generate enough blocks to provide approximately 2.5 full execution waves across the A100’s 108 SMs, which optimally hides memory access latency.

Changes in manuscript: We replaced the vague description with a detailed quantitative and profiling-based explanation at the end of Section 3.1 (Data Structures and Parallelization Strategy):

"In our implementation, thread block sizes of 128 and 256 were both utilized, with the specific parameter for each kernel determined empirically via NVIDIA Nsight Compute. For the target mesh comprising 554,394 elements, a block size of 128 yields 4,332 blocks, while 256 yields 2,166 blocks. Given that the NVIDIA A100 GPU features 108 Streaming Multiprocessors (SMs), each with a theoretical maximum of 2,048 resident threads, both configurations generate sufficient numbers of thread blocks to provide approximately 2.5 execution waves across the GPU, which helps maintain high occupancy and effectively hide memory latency."

We hope these clarifications and revisions fully address your insightful comments. Thank you

once again for your constructive guidance.

GPU-accelerated Finite-Element Method for the Three-dimensional Unstructured Mesh Atmospheric Dynamic Framework

Leisheng Li[1], Ximeng Fu[1,2], Xiyu Zheng[1,2], Huiyuan Li[1], and Jinxi Li[3]

[1]Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
[2]University of Chinese Academy of Sciences, Beijing, 100190, China
[3]State Key Laboratory of Atmospheric Environment and Extreme Meteorology, Institute of Atmospheric Physics, Chinese Academy of Sciences, Beijing 100029, China

Correspondence: Jinxi Li (ljx2311@mail.iap.ac.cn)

Abstract. The three-dimensional unstructured-mesh finite-element atmospheric dynamical framework is gaining significance owing to its flexibility in representing complex topography and capability for multi-scale simulations in high resolutions. However, this framework has substantial bottlenecks. Unlike structured-grid models, the unstructured finite element method (FEM) must frequently access irregular mesh connectivity among nodes, edges, and elements, causing indirect memory addressing, inadequate data locality, and substantial memory bandwidth bottlenecks on conventional CPU architectures. Consequently, element-wise computations and global assembly are the primary contributors to the runtime in high-resolution simulations.

This study develops a GPU-parallel implementation of the Fluidity-Atmosphere dynamical core to address these challenges. The GPU-oriented data structures and optimized kernels are designed to efficiently leverage the computing power of GPUs. These kernels enable parallelized element integration and are efficient solvers for specific size matrices; a parallel assembly strategy enhances memory throughput during global sparse matrix construction. On the NVIDIA A100 GPU, the optimized kernels achieve speeds over 100× for element-wise computations and up to 389.02 times for global matrix assembly, resulting in an overall acceleration of 8.57 times with four messages passing interface (MPI) processes. The proposed framework demonstrates that tailored GPU parallelization is effective in overcoming the computational bottleneck of unstructured FEM-based atmospheric models, facilitating high-resolution simulations on heterogeneous architectures.

1 Introduction

Atmospheric dynamic frameworks are the fundamental tools for weather forecasting and climate simulation. In recent years, growing demands for the accurate prediction and management of extreme pollution and weather events have driven atmospheric models toward higher spatial resolutions and more sophisticated physical parameterizations. These advances create formidable computational challenges: doubling the spatial resolution can lead to exponential growth in computational cost, substantially increasing the demand for computing power and storage resources (Michalakes, 2020). For the discretization methods, the choice of scheme is primarily dependent on the topological structure of the underlying mesh. For high-resolution simulations, models are now able to resolve increasingly complex underlying surface features, such as terrains and urban structures. Owing to their geometric constraints, traditional terrain-following coordinate grids typically produce significant mesh distortions near

steep terrain, introducing significant computational errors. Therefore, conventional atmospheric models based on structured "horizontal + vertical" grid configurations encounter inherent limitations in representing complex terrain boundaries, offering inadequate flexibility for localized adaptation.

To address this challenge, fully three-dimensional (3D) unstructured grids have emerged as an effective solution. Replacing the conventional quasi-3D approach that separates horizontal and vertical discretization, unstructured grids provide superior geometric adaptability and more flexible mesh generation capabilities. The finite element method (FEM) proves particularly well-suited to such grid architectures, offering inherent advantages in handling complex geometrical configurations. This synergistic combination has established FEM as an increasingly prominent methodology in atmospheric modeling (Farrell et al., 2009; Li et al., 2018). FEM not only preserves conservation and numerical stability on irregular geometries but also enables local mesh refinement in critical regions, thereby improving simulation accuracy (Takle and Russell, 1988; Li et al., 2021). To leverage these advantages, the Institute of Atmospheric Physics, Chinese Academy of Sciences, and AMCG group at Imperial College London jointly developed Fluidity-Atmosphere, a 3D adaptive atmospheric model (Davies et al., 2010). Based on the unstructured FEM, Fluidity-Atmosphere tightly couples the Navier-Stokes equations, complete advection-diffusion dynamics, and anisotropic adaptive mesh algorithms, facilitating dynamic mesh optimization during simulations to capture multi-scale flow phenomena.

However, while the FEM on unstructured meshes provides superior geometric flexibility, it also introduces inherent computational challenges that are absent in structured-grid models. In unstructured FEM frameworks such as Fluidity-Atmosphere, each element must frequently access irregularly connected nodes, edges, and faces during numerical integration and global sparse matrix assembly. This results in non-contiguous memory access, indirect addressing, and load imbalance across elements. These challenges are typically absent in structured grids, where data are stored in regular arrays with predictable access patterns. Moreover, the adaptive mesh refinement (AMR) feature of Fluidity-Atmosphere dynamically modifies the mesh topology during simulations, further complicating the memory layout. Consequently, the two most time-consuming components, namely, element-wise computations and global sparse matrix assembly, become dominated by irregular data movement rather than floating-point arithmetic, causing substantial memory bandwidth bottlenecks on CPU architectures (Sulyok et al., 2019).

Meanwhile, the slowdown of Moore's law and energy-efficiency bottlenecks of CPUs render CPU-only optimization inadequate for high-resolution simulations. In this context, GPUs have emerged as the widely used heterogeneous accelerators in high-performance computing. With massive parallelism and high memory bandwidth, GPUs have delivered orders-of-magnitude speedups in many numerical applications (Czarnul et al., 2020). For atmospheric models, particularly those based on FEM, computational intensity primarily arises from solving large-scale linear systems, parameterizing physical processes, and performing massive element-level operations, each amenable to GPU acceleration. Reengineering these components to run efficiently on GPUs has therefore become a key research direction.

Recent studies have reported significant advances in GPU acceleration of atmospheric models, including Navier-Stokes solvers (Goddeke et al., 2009; Thibault and Senocak, 2012), advection-diffusion equations (Simek et al., 2009), and iterative solvers for implicit schemes (Müller et al., 2015), typically with speedups of several orders of magnitude. Some Earth system models such as ocean model (Petersen et al., 2025), tsunami model (Conde et al., 2025), cloud-resolving atmosphere model

(Bogenschutz et al., 2025) and sea-ice model (Jendersie et al., 2025) also were accelerated using GPUs. FEM modules such as numerical integration (Macioł et al., 2010; Sanfui and Sharma, 2020), matrix assembly (Kiran et al., 2020; Sanfui and Sharma, 2021), and linear solvers (Ratnakar et al., 2021; Kiran et al., 2020) have been explored, achieving speedups from tens to hundreds of times. Typically, the solution of large sparse linear systems is regarded as the dominant cost in FEM-based simulations and has been extensively studied, with numerous GPU-parallel implementations already achieving mature performance. However, the computation of elements and subsequent matrix assembly may pose significant challenges, particularly in large-scale unstructured atmospheric models. During each iteration, local element-wise matrices and right-hand sides must be computed and assembled into global sparse matrices. As the simulation domain expands, the number of mesh elements grows exponentially, leading to a proportional increase in computational cost. Without targeted optimization, the efficiency of the entire model can be substantially reduced.

In practice, element-wise computations and global matrix assembly involve frequent access to unstructured mesh data, where the topological relationships among points, edges, faces, and cells are maintained through multiple interlinked index arrays. These indirect data access operations disrupt spatial locality and cause irregular memory traffic, resulting in low cache utilization and high latency. Therefore, despite substantial progress in GPU acceleration of structured or semi-structured atmospheric models, unstructured FEM-based frameworks–characterized by irregular data dependencies and adaptive mesh refinement–still lack systematic and efficient GPU solutions. This study is primarily focused on addressing this research gap.

This study addresses these challenges by focusing on GPU parallelization and optimization of the two most time-consuming components of Fluidity-Atmosphere: **element-wise computations** and **global matrix assembly**. The primary contributions are as follows.

1. **GPU-based element-wise computations**: Multiple optimization techniques are applied to accelerate compute-intensive kernels via GPU offloading, achieving more than 100× speedups compared with CPU versions.
2. **GPU-based global matrix assembly**: Parallel strategies are designed and implemented for element-wise assembly.
3. **Integrated GPU implementation**: The GPU kernels are integrated into Fluidity-Atmosphere with pinned memory to optimize data transfer. The results show that GPU kernels facilitate speedups exceeding two orders of magnitude, with 4 MPI processes and one GPU, achieving an overall acceleration of ~~10.28~~ 8.57 times compared to a single CPU process.

The remainder of this paper is organized as follows. Section 2 introduces the program structure of Fluidity-Atmosphere, with emphasis on element-wise computations and global matrix assembly. Section 3 presents the GPU implementation and optimization of element-wise computations. Section 4 describes GPU-based global matrix assembly strategies. Section 5 evaluates the integrated GPU implementation and its performance. Section 6 concludes the paper, highlighting the future scope.

2 Target Scientific Application Definition: Fluidity-Atmosphere

Fluidity-Atmosphere employs a mixed finite element/finite volume framework that incorporates the Navier-Stokes momentum equations, advection-diffusion equations for potential temperature and water vapor, and the compressible continuity equation

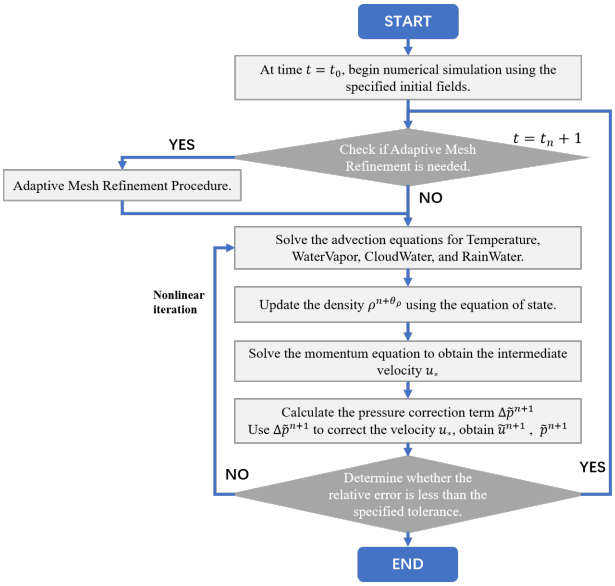

Figure 1. Time loop of Fluidity-Atmosphere.

within an anisotropic adaptive mesh algorithm. This design facilitates dynamic co-optimization of the mesh and physical fields. Fluidity-Atmosphere solves a coupled system of nonlinear equations with (typically) time-varying solutions. The time-marching algorithm employed uses a non-linear iteration scheme known as Picard iteration in which each equation is solved using the currently optimal solution for the other variables. The dynamical framework supports both the continuous Galerkin (CG) and discontinuous Galerkin (DG) finite element formulations. This study is focused exclusively on the CG scheme.

Figure 1 illustrates the general iteration loop of Fluidity-Atmosphere (Li et al., 2021). Fluidity-Atmosphere could invoke the adaptive algorithm at regular timesteps to ensure that the dynamics do not extend beyond the zone of adapted resolution. The adaptive algorithm has been parallelized using MPIs, which run efficiently on CPUs. In our performance profiling, the nonlinear iterations were predominant. In each timestep iteration, the model solves governing equations such as the advection-diffusion and momentum equations. The processes of solving these equations in Fluidity-Atmosphere typically are divided into two major computational stages:

(1) **Construct matrices**: For each element, numerical integration is performed using Gaussian quadrature to evaluate local stiffness, mass matrices, and right-hand side (RHS) vectors. These local contributions are subsequently inserted into the global sparse matrix system.

(2) **Solution of linear systems**: The assembled sparse matrices are solved using parallel numerical libraries such as PETSc, employing iterative solvers and preconditioners well-suited for large-scale sparse problems. The PETSc library has good performance and scalability. In the simulated case studies, its contribution to the overall runtime was relatively small. In

addition, PETSc already supports GPUs (Mills et al., 2021). Hence, future studies will be focused on the integration of Fluidity-Atmosphere and PETSc GPU computing.

The computation times of constructing matrices contributes significantly to the overall runtime. For instance, the Mountainwave3D simulation, in which the number of mesh nodes and mesh elements are 103635 and 554394, respectively. Table 1 shows that the computational times of constructing matrices in a timestep. In particular, the pressure diffusion matrix is used to calculate the stabilization term, which is only calculated at the first timestep or when the grid changes because of the adaptive process. In the Mountainwave3D simulation, the pressure diffusion matrix is computed every ten time steps. Without calculating the stabilization term, the total time of the timestep will decrease, but the proportion of time required to construct the matrices will slightly increase. ~~Consecutively~~As quantified in Table 1, the processes of constructing matrices ~~including~~ account for approximately 60% of the total timestep duration in both scenarios (with and without the stabilization term). This high proportion confirms that accelerating **element-wise computations** and **global ~~matrix~~ assembly** ~~have been identified as the primary computational bottlenecks to be addressed in this study~~is the most effective strategy for improving the overall performance of the Fluidity-Atmosphere model.

The processes of constructing matrices share highly similar loop patterns. Hence, their general workflow can be summarized as follows:

Step 1 **Preparing Data**: Initialize element mass matrices, right-hand sides, and local physical variables (such as velocity, density, and viscosity).

Step 2 **Coordinate transformation**: The `transform_to_physical` function maps shape function gradients into physical space and computes Gaussian weights (`detwei`).

Step 3 **Setup test function.**

Step 4 **Contribution evaluation**: Call physics modules to accumulate contributions into element matrices and right-hand sides. Several functions, each corresponding to a distinct physical process, such as mass, advection, diffusion, source, and viscosity terms, are called. In addition to the calculation of physical variables, these functions extensively call the integration calculation function on the elements such as `dshape_tensor_dshape`, `shape_dshape`, and `shape_shape`. In the process of assembling the pressure diffusion matrix, a special function named `get_edge_lengths` (in which the small matrix solver is called) is used to calculate the element length scale in the physical space. All these functions are well-suited for GPU parallel computing.

Step 5 **Assembly**: Local contributions are inserted into the global matrix and RHS. The subsequent irregular insertion operations into the global matrix makes it highly data-intensive and memory-bound. Owing to the high bandwidth of GPUs, these functions can be accelerated on GPUs.

Therefore, the following sections describe the implementation and performance optimization methods of the element-wise computations and global matrix assembly on the GPU. ~~This study is based on~~

Although the current implementation is specifically optimized for the NVIDIA A100 GPU ~~hardware and~~ utilizing the CUDA programming model (NVIDIA, 2025a), the proposed optimization strategies are largely hardware-agnostic at the algorithmic level. Specifically, the element-wise parallelization strategy (one-thread-per-element), the data layout design, and the sparse matrix assembly workflow are general and can be applied to other GPU architectures.

However, certain implementation details rely strictly on CUDA-specific hardware features, such as double-precision atomic operations in the L2 cache, memory hierarchy optimization, and specific kernel launch configurations. Porting the implementation to other platforms, such as AMD GPUs, would primarily involve adapting CUDA-specific APIs to alternative frameworks like HIP (AMD, 2025). Since HIP provides similar abstractions for thread hierarchy, memory management, and atomic operations, the overall syntax porting effort is expected to be moderate.

Nevertheless, achieving true performance portability would require additional tuning to account for architectural differences in memory bandwidth, cache structure, and execution models. For instance, while NVIDIA V100 and A100 GPUs handle massive atomic updates highly efficiently, evaluating similar unstructured-grid applications on AMD CDNA architectures (e.g., MI100) has shown that standard atomic updates can incur higher penalties, sometimes necessitating alternative data restructuring or register-based aggregation to achieve optimal efficiency (Stone et al., 2021). Overall, the proposed algorithmic approach is expected to maintain its fundamental effectiveness across heterogeneous architectures, although achieving peak optimal performance on different platforms will inevitably require platform-specific tuning. ~~Despite the availability of several programming models, CUDA is predominantly used because of its remarkable performance. Several programming models such as the heterogeneous-computing interface for portability strive to be compatible with CUDA. Hence, porting to other GPU hardware platforms will not encounter any significant obstacles (AMD, 2025).~~

3 GPU Parallelization of Element-wise Computations

In Fluidity-Atmosphere, element-level operations such as numerical integration and coordinate transformation constitute the core of unstructured finite element-wise computations. Profiling results indicate that these routines are invoked millions of times per timestep. Each nonlinear iteration involves re-evaluating element Jacobians, transforming basis functions, and accumulating physical contributions for all elements, which cumulatively dominate the computational workload. These observations identify numerical integration and coordinate transformation as the primary performance bottlenecks and thus the key focus of GPU parallelization in this study. The computation for each mesh element can be performed independently of others, making this step an ideal candidate for parallelization (Georgescu et al., 2013). GPUs feature a massively parallel processor architecture, which enables the simultaneous launch of a large number of parallel threads. These threads can be associated with different mesh elements to execute element-wise computations. These results are stored for subsequent assembly into the global matrix. Building on the workflow analysis in Section 2, this section presents the GPU implementation and optimization strategies for key subroutines, followed by a summary of general acceleration techniques.

Table 1. The computation times of constructing matrices

	With stabilization term		Without stabilization term	
Functional module	CPU time(ms)	Percentage	CPU time(ms)	Percentage
The advection-diffusion matrix (Temperature)	2477.545	4.84%	2466.594	5.95%
The advection-diffusion matrix (WaterVapor)	2477.194	4.82%	2471.302	5.96%
The advection-diffusion matrix (CloudWater)	2466.853	4.82%	2494.424	6.02%
The advection-diffusion matrix (RainWater)	1785.827	3.49%	1799.104	4.34%
The momentum matrix	13029.408	25.47%	12093.145	29.19%
The divergence matrix	3662.082	7.16%	3634.111	8.77%
The pressure diffusion matrix	3505.105	6.85%		
The projection matrix	1126.747	2.20%	1125.822	2.72%
Total of constructing matrices	**30530.760**	59.68%	**26084.502**	**62.96%**
Others	20625.871	40.32%	15348.450	37.04%
Total timestep duration	51156.631	100.00%	41432.952	100.00%

3.1 Data Structures and Parallelization Strategy

In unstructured meshes, element connectivity must be explicitly stored. Each tetrahedral element in Fluidity-Atmosphere records four vertex indices. Physical variables are stored in a unified Fields data structure organized as (vertex index $\times$ dimension) arrays: scalars (1D), vectors (3D), and tensors (9D). The same layout is adopted in GPU memory to ensure contiguous access and consistent indexing (Figure 2). Common fields such as density and temperature are stored as scalars, positions and velocities as vectors, and viscosity terms as tensors. The element-wise computations in the FEM are intrinsically independent, making them particularly suitable for massively parallel execution on GPU architectures. Unlike spectral methods that involve global communication or high-order finite-difference schemes relying on extended stencils, finite-element computations are confined strictly within each element boundary. Even when accuracy is increased, through higher-order polynomial interpolation or additional Gauss points, these operations remain entirely local without introducing cross-element dependencies.

This strong locality offers two principal advantages: (1) all element-wise computations can proceed independently, eliminating inter-thread communication during the compute phase; and (2) the workload is naturally balanced across elements, minimizing synchronization overhead. Consequently, a "one-thread-per-element" parallelization strategy is adopted. Each GPU thread handles one element. ~~Meanwhile, thread blocks typically contain~~ In our implementation, thread block sizes of

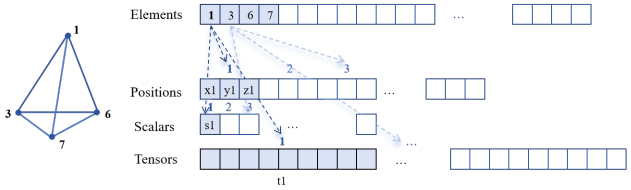

Figure 2. Data layout of elements and variables.

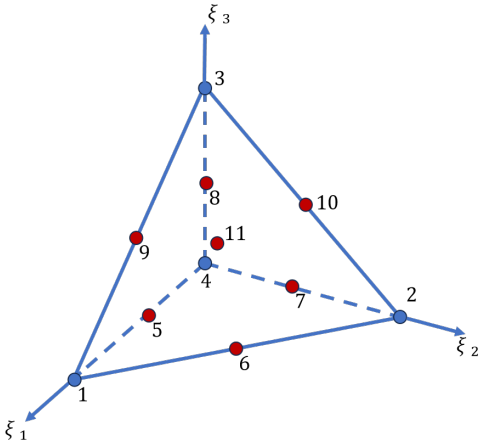

Figure 3. The reference element and Gauss points.

128 ~~or~~ and 256 ~~threads; the parameters are tuned empirically to match kernel characteristics and maximize occupancy. This mapping enables tens of thousands of threads to execute concurrently, achieving high throughput and scalability for large-scale atmospheric simulations~~were both utilized, with the specific parameter for each kernel determined empirically via NVIDIA Nsight Compute. For the target mesh comprising 554,394 elements, a block size of 128 yields 4,332 blocks, while 256 yields 2,166 blocks. Given that the NVIDIA A100 GPU features 108 Streaming Multiprocessors (SMs), each with a theoretical maximum of 2,048 resident threads, both configurations generate sufficient numbers of thread blocks to provide approximately 2.5 execution waves across the GPU, which helps maintain high occupancy and effectively hide memory latency.

3.2 GPU Parallel Coordinate Transformation

With the data layout on GPUs established, the next step is to implement the core finite element operation: coordinate transformation. Using a reference element (in this study, a tetrahedron), shape functions and Gauss points are consistently defined. In Fluidity-Atmosphere, each physical element employs 11 Gauss points: four vertices, six edge midpoints, and the centroid (Figure 3). For each Gauss point, the transformation routine evaluates the integrand and multiplies it by the predefined weight.

Even though element-independent and theoretically parallelizable, this kernel is computationally intensive and one of the most frequently executed routines in the model. For every iteration and every element, it performs numerous small matrix operations: matrix multiplications, adjoint and determinant evaluations, and transformations of derivative matrices.

Algorithm 1 outlines the `transform_to_physical` procedure. In addition, small-matrix operations such as Jacobian construction, adjoint computation, and determinant evaluation are optimized for GPU execution. Input and output arrays are stored contiguously in GPU memory to maximize access efficiency.

Algorithm 1 Transform_to_physical

Input: Element nodal coordinates X, shape functions N, reference derivatives $\nabla_\xi N$, Gauss points g_i

Output: Physical derivatives $\nabla_x N$, weights detwei, J, J^{-1}, $\det J$

for $g_i = 1$ *to* n_{gi} **do**

  compute $\nabla_\xi N(g_i)$

  construct Jacobian matrix $J \leftarrow X \cdot (\nabla_\xi N)^T$

  compute adj(J) and $\det J$

  normalize inverse matrix $J^{-1} \leftarrow \text{adj}(J)/\det J$

  transform to physical derivatives $\nabla_x N \leftarrow J^{-1} \cdot \nabla_\xi N$

  compute quadrature weights detwei$(g_i) \leftarrow |\det J| \cdot w_{gi}$

end

3.3 GPU Parallelization of Element-wise Integration

Element-wise integration routines constitute the computational core of unstructured finite-element models, as they are responsible for evaluating the contributions of each element to the global system. Profiling of Fluidity-Atmosphere indicates that these kernels are among the most frequently executed routines. During each nonlinear iteration, they are called once per element per physical equation, resulting in millions of invocations per timestep. As each call involves multi-dimensional tensor operations and numerical integration over multiple Gauss points, the accumulated cost dominates both the element-computation phase and the overall runtime of the model. Therefore, optimizing element integration is critical to achieving substantial end-to-end performance gains. The principal integration functions are as follows:

- **`dshape_tensor_dshape`** couples shape function gradients with tensor fields, supporting stabilization terms and physical field coupling.
- **`shape_shape`** integrates shape functions to construct the mass matrix, required in momentum and energy conservation equations.
- **`shape_dshape`** performs weighted integration of shape functions and their gradients at nodal points.

Among these functions, the **`dshape_tensor_dshape`** function is the most computationally intensive, as it involves both tensor-vector and vector-vector multiplications for every integration point. On the GPU, this procedure is parallelized at the

thread level, with each thread processing one element. Beyond kernel-level parallelization, a series of GPU-specific performance tuning techniques are applied to completely exploit the hardware potential. These optimizations are guided by profiling by employing NVIDIA Nsight Compute, focusing on reducing memory latency, register pressure, and control overhead:

- **Memory optimization**: The **`__restrict__`** qualifier is applied to pointer variables to eliminate aliasing and enable more aggressive memory-access optimizations. Small **`__device__`** functions are annotated with **`__forceinline__`** to reduce function-call overhead.
- **Loop optimization**: Loop-invariant expressions (particularly global memory accesses) are hoisted outside the loop body; manual loop unrolling or `#pragma unroll` directives are applied to innermost loops to reduce control overhead and increase instruction-level parallelism.
- **Template metaprogramming**: As the dimensions of the different matrices and vectors are already known at compile time, the **`__device__`** functions such as matrix-matrix product and dot product are implemented as templates, facilitating the compiler to apply deeper optimizations for specific instantiations.

These measures are reflected in Nsight Compute reports as improved register utilization and reduced memory bottlenecks. Loop unrolling and templating yield substantial enhancements in small-scale computations. In combination with GPU-parallel element integration and small-matrix solvers, these optimizations form a cohesive strategy that maximizes performance while maintaining full consistency with the original CPU implementation. The optimized code is listed in **Appendix A**.

3.4 GPU Parallelization of Small-Scale Linear Algebra Solvers

In addition to numerical integration, the construction of stabilization terms involves numerous small-scale linear algebraic operations, such as solving 6×6 systems or computing eigen-decompositions of 3×3 matrices. Even though these operations are apparently lightweight, they are called repeatedly for every element and Gauss point, resulting in a substantial cumulative cost. On CPUs, such operations are inefficient because they are very small to amortize function-call overhead and cannot exploit vectorization effectively. Library routines such as LAPACK's DSPEV or Fortran's intrinsic solve introduce additional latency because of memory indirection and cache misses.

To address this challenge, this study implements dedicated GPU-based small-matrix solvers and eigen-decomposition kernels as inline **`__device__`** operators.

- **Small linear systems (e.g., 6×6)**: These are solved on GPUs using Gaussian elimination. The input is an augmented matrix A (in column-major layout, including the right-hand side) with dimensions $m\times n$. Pivoting and row exchanges are applied to control numerical errors, whereas column-major storage and in-place operations reduce register pressure and facilitate the alignment with GPU memory characteristics.
- **Eigen-decomposition of 3×3 real symmetric matrices**: Both iterative and direct GPU solvers are implemented, based on the methods employed in prior research. To improve efficiency, the implementation employs template-based array

classes and includes sorting functions for eigenvalues and eigenvectors. By exploiting symmetry, only six entries (the diagonal and upper triangular part) are stored, minimizing register usage and avoiding spills to local memory.

These GPU-implemented solvers are lightweight and reusable across kernels. The modular `__device__` design also promotes code reuse across physical modules and supports further fusion with integration kernels.

4 GPU Parallelization of Matrix Assembly

In the FEM, matrix assembly serves as the critical bridge between local element-wise computations and the solution of global linear systems. As this stage involves extensive operations on sparse data structures and frequent memory access, it typically hinders the performance of the entire program. Therefore, migrating the matrix assembly procedure to GPUs and applying targeted optimizations is crucial for enhancing the computational efficiency of atmospheric models. This section introduces the global matrix storage formats, the element-wise parallel assembly strategy, and the GPU implementation of RHS vector assembly.

4.1 Global Matrix Assembly

Following local integration, the algorithm accumulates (A^e, F^e) into the global matrix A and vector F according to the connectivity information of the mesh. This procedure, widely recognized as the global assembly, transforms element-level contributions into a consistent global representation based on the mapping between local and global degrees of freedom, as shown in **Algorithm 2**. Here, ε is the set of elements and e is an element. The $elem$ function performs element-wise computations to get element matrices. The L function retrieves the node numbering.

Algorithm 2 The Global Matrix Assembly

Output: Global matrices A, F

Initialize A, F to zero
forall $e \in \varepsilon$ **do**
  Compute $(A_e, F_e) = \text{elem}(e)$
  forall *local degrees of freedom* d_1 *of* e **do**
    $F(L(e,d_1)) \mathrel{+}= F_e(d_1)$
    forall *local degrees of freedom* d_2 *of* e **do**
      $A(L(e,d_1), L(e,d_2)) \mathrel{+}= A_e(d_1,d_2)$
    end
  end
end

In contrast to structured-grid models, where mesh nodes are arranged in a regular (i, j, k) indexing order and memory access can be performed sequentially and predictably, unstructured meshes lack geometric regularity. The connectivity of each element varies depending on its neighbours, and the mapping between local and global indices must be retrieved indirectly through lookup tables. Consequently, the assembly phase in unstructured FEM involves frequent non-contiguous global memory reads and writes, with irregular strides between consecutive memory locations. This destroys spatial locality and drastically increases cache miss rates, thereby amplifying the ratio of memory access time to floating-point computation.

On GPU architectures, these irregular access patterns exacerbate performance degradation. Concurrent threads typically need to update overlapping entries of the global sparse matrix, requiring synchronization or the use of atomic operations to ensure correctness. The combined effects of irregular global memory traffic, low cache reuse, and write conflicts make matrix assembly a highly memory-bound process.

Consequently, global matrix assembly presents two major challenges: (1) the imbalance between memory traffic and arithmetic intensity, caused by the dominance of irregular global data movement over computation; (2) the difficulty of ensuring data consistency during concurrent updates. Cumulatively, these factors make matrix assembly another critical performance bottleneck, and thus a central focus of GPU parallel optimization in this work.

4.2 Sparse Matrix Storage Formats

Fluidity-Atmosphere employs different sparse storage formats for different types of physical fields. For scalar field matrices, the **compressed sparse row (CSR)** format is used. For vector field matrices such as three-component velocity, the **block-CSR (BCSR)** format is adopted.

For a scalar field, the local stiffness matrix can be expressed as Eq. (1):

$$K_{ij} = \int_{\Omega_e} (\nabla N_i)^T C \, (\nabla N_j) \, \mathrm{d}\Omega \tag{1}$$

where N_i and N_j are basis functions, Ω_e is the integration domain of the element, and C is the associated tensor. For scalar fields, N_i is a scalar and K_{ij} is a scalar integral, resulting in a 4×4 local element matrix. By obtaining the global node indices n_i and n_j corresponding to the local nodes i and j, the result K_{ij} is updated in the global matrix entry (n_i, n_j).

The global matrix is typically stored in the CSR format, which consists of three arrays: **values** (non-zero entries), **col_index** (column indices), and **row_pointers** (row offsets) (NVIDIA, 2025b). Within a row range, the target column can be located using binary search.

For vector fields, each node contains three degrees of freedom (DOFs). Thus, while assembling the local stiffness matrix, all interactions among the nodes and their DOFs must be considered. This results in a set of 4×4 blocks, each of size 3×3:

$$K = \begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} \\ K_{21} & K_{22} & K_{23} & K_{24} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ K_{41} & K_{42} & K_{43} & K_{44} \end{bmatrix} \tag{2}$$

Each element of the element matrix K_{ij} is actually a 3×3 matrix.

$$K_{ij} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{3}$$

Here, Fluidity adopts the BCSR format, which is structurally similar to CSR, but stores continuous block data in the value array. This block-based representation better supports sparse operations for vector fields.

4.3 GPU Parallel Element-Wise Assembly

In GPU parallelization, the contributions of each mesh element are computed and stored in parallel, with each GPU thread responsible for one element. A dedicated kernel is launched to assemble element matrices.

In the implementation, each thread iterates over the entries K_{ij} of the local element matrix and computes the corresponding global non-zero indices. Given the global node indices n_i and n_j, the **row_pointers** array is first used to locate the row range [row_pointers(n_i), row_pointers($n_i + 1$)). Within this range, the column index n_j is searched using binary search. To enhance the efficiency, the implemented GPU assembly function consistently adopts binary search.

A significant challenge arises during parallel write-back, where race conditions may occur if multiple threads attempt to update the same non-zero entry simultaneously. An atomic operation is indivisible and guarantees consistency by preventing interference from other threads. Modern GPUs provide hardware-level support for atomic addition, enabling conflict-free concurrent updates without explicit locks. In this work, the atomic add approach is adopted for its efficiency and simplicity. The CUDA parallel assembly code is listed in **Appendix B**.

4.4 GPU Parallel Assembly of Right-Hand-Side (RHS) Vectors

Similar to the global matrix, the RHS vector is assembled using an element-wise parallel strategy. Each GPU thread computes the nodal contributions of one element and accumulates them into the corresponding global vector entries.

The implementation first determines the indices of the nodes belonging to each element, then loads the nodal values, and finally applies **`atomicAdd`** operations to ensure correctness during concurrent accumulation. For cases where multiple variables are associated with a node, appropriate offsets are added to the corresponding positions. The **Appendix C** illustrates the procedure of RHS assembly, which achieves efficient large-scale parallel accumulation while preserving correctness.

Table 2. GPU & CPU and compiler information.

GPU NVIDIA	A100
Peak perf. (FP64)	9.7 TFLOPS
Max Clock Freq	1410 MHz
L1 Cache	192 KiB
L2 Cache	40960 KiB
SM per GPU	108
Register file size per SM	256 KiB
PCIe bandwidth	32 GB/s
nvcc	13.0
CPU	AMD EPYC 7713 64-Core Processor
ISA	X86_64
Max Clock Freq	3720 Mhz
L1 Cache	8 MiB
L2 Cache	64 MiB
Cores	64
gcc	13.3
CMake	3.22.1
gfortran	13.3
mpicc	MPICH 3.4.2
Optimization	-O3 -ffast-math

5 Results and Evaluation

5.1 Experimental Environment

This section presents a systematic analysis of the performance of the proposed GPU parallelization strategies. Experiments were conducted to first validate correctness on both CPU and GPU platforms, and then to evaluate performance in four aspects: element-wise computation, global matrix assembly, data transfer with computation overlap, and overall model performance. The GPU and CPU experimental environments are shown in Table 2.

GPU acceleration was applied to the momentum equation, advection equation, and stabilization term construction of Fluidity-Atmosphere. Then, the GPU-accelerated version was validated for correctness and evaluated for performance to ensure both accuracy and efficiency.

5.2 Validation Methodology

For validation of correctness, the 3D idealized mountain wave test case (Li et al., 2021) was used. The mountain wave test serves as a crucial validation procedure for assessing the dynamic framework model of the Fluidity-Atmosphere model in simulating orography-induced airflow. By comparing numerical results against theoretical solutions of idealized orographically forced flows, this test effectively evaluates the capability of the model to capture mountain wave generation and propagation phenomena under complex topographic conditions. The computational domain spans 60 km in both horizontal dimensions with a vertical extent of 16 km. Adaptive mesh resolution dynamically ranges from 125 m to 10 km throughout the simulation domain, with mesh refinement actively guided by variations in velocity magnitude and potential temperature. The mesh consists of 554394 elements and 103635 nodes. A 3D bell-shaped mountain profile is mathematically described as follows:

$$h(x,y) = \frac{h_0}{\left(1 + \frac{x^2+y^2}{a^2}\right)^{\frac{3}{2}}} \tag{4}$$

where $h_0 = 400m$ represents the peak elevation and $a = 1000m$ denotes the characteristic half-width parameter. The stratified atmospheric background is characterized by $N = 0.01s^{-1}$, while the surface potential temperature initializes at $\theta_0 = 293.15K$. The incoming flow maintains a constant velocity of $u = (10,0,0)^T m/s$. For numerical stability, an absorbing layer is implemented in the upper atmospheric region (10-16 km-altitude), with additional dissipative layers extending for 10 km inward from all lateral boundaries (comprehensive specifications available in Li et al., 2021).

In scientific applications that require high-precision floating-point operations, the variations in floating-point results for different architectures are evident and well-established. In the Mountainwave3D simulations, small floating-point differences result in different mesh divisions in the process of mesh adaptivity. For unstructured grids, using interpolation calculations for different mesh divisions result in significant errors. Therefore, a direct comparison of the simulation results of GPU and CPU from time zero cannot verify whether the GPU code is consistent with the CPU code calculation results. In the proposed method, the physical time of 3000 s is simulated using the CPU version to make the simulation reach a stable state. Then, the computation is restarted using both GPU and CPU versions of the program, keeping the mesh unchanged during simulation and advancing the physical time to 5000 s.

Although the current performance evaluation is conducted using a static mesh to ensure a rigorous and consistent validation against CPU results, the implemented GPU kernels are designed to fully support the dynamic nature of Fluidity-Atmosphere. In a practical adaptive mesh refinement (AMR) scenario, the primary additional overhead involves the periodic transfer of the updated connectivity array (`ndglno`) from the CPU to the GPU. As demonstrated by our data transfer analysis (Table 7), such transfers (e.g., 0.348 ms for the `ndglno` array) are negligible compared to the overall timestep duration, thereby ensuring the framework's efficiency during fully adaptive simulations.

The comparison of the calculation results between the two versions is shown in Figure 4. The results indicate that the maximum difference between GPU and CPU is on the negative fourteenth power of ten, indicating that the results of CPU and GPU are consistent.

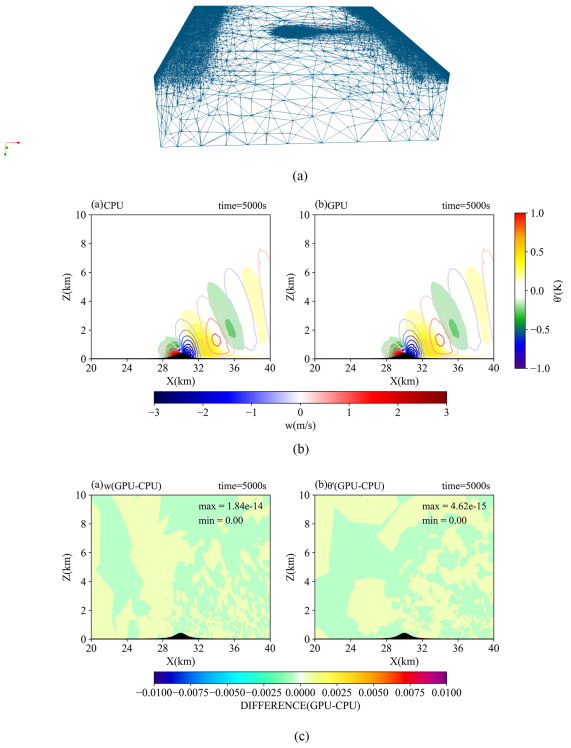

Figure 4. Comparison between GPU-accelerated computation results and CPU computation results. (a)Mesh; (b)The computation results of the CPU and GPU versions; (c)The differences between the CPU and the GPU versions.

5.3 Results and Discussion of Performance Optimization

5.3.1 GPU Element-wise Computation Performance

In unstructured meshes, element node values are typically stored non-contiguously in memory. Thus, the performance of element-wise computation is primarily limited by memory access efficiency. While CPUs incur high memory overhead for non-contiguous access, GPUs leverage high bandwidth and massive thread parallelism to alleviate this bottleneck. Table 3 compares GPU and CPU performance in scalar, vector, and tensor data access. This is primarily because scalar data structures are relatively simple, facilitating GPUs to completely leverage high-bandwidth access. All the performance profiling data were collected with the mesh in the Mountainwave3D simulation, in which the number of mesh nodes and elements are 103635 and 554394, respectively.

Furthermore, Table 4 presents the GPU performance of numerous core functions in Fluidity-Atmosphere (coordinate transformation, integration, and auxiliary routines). Results indicate that parallelizing dense element-wise computations on GPUs significantly accelerates the performance. All listed functions achieved more than **103**× speedups compared with CPU ver-

Table 3. GPU Performance of GPU parallel element data access (Mesh: 103,635 nodes, 554,394 elements; timings are averaged over 1,000 invocations).

Function	CPU time (ms)	GPU time (ms)	Speedup
ele_val_scalar	6.628	0.0276	240.14
ele_val_vector	13.633	0.0768	177.51
ele_val_tensor	32.345	0.1976	163.69

Table 4. GPU kernel performance in element-wise computation (Mesh: 103,635 nodes, 554,394 elements).

Function	CPU time (ms)	GPU time (ms)	Speedup
transform_to_physical	525.575	3.964	132.59
dshape_tensor_dshape	1865.290	1.969	947.33
shape_shape	104.221	1.012	103.01
shape_dshape	246.047	1.371	179.41
get_edge_lengthes	1369.171	7.476	183.14

sions. These functions are computationally intensive, involving repeated local linear system and eigenvalue/eigenvector solves for each element in serial CPU execution. In contrast, the GPU can execute such highly independent tasks concurrently across thousands of threads, comprehensively leveraging parallelism.

To analyse the effects of GPU-oriented optimizations, we take the `dshape_tensor_dshape` function as an example. This function is a hotspot in finite-element computation, involving frequent access to gradients of shape functions at Gauss points and dense matrix multiplications/dot products. Table 5 shows performance enhancement due to loop unrolling and function template optimization. Notably, Fluidity-Atmosphere typically calls this routine with identical `dshape1` and `dshape2` array. However, for generality we also tested cases with two different `dshape1` and `dshape2` arrays. While the same array case performs better, both scenarios show consistent performance gains.

Nsight Compute profiling further indicates that templated matrix product and dot product increased memory throughput from 70 % to 82.35 %, slightly improved compute throughput, and enhanced L1 cache hit rates. SASS code inspection revealed that baseline **`__device__`** functions were only locally reordered, thereby limiting the performance. After templating, the compiler applied deeper optimizations, interleaving SASS instructions between **`__device__`** and **`__global__`** functions, yielding significant performance enhancements.

These results confirm that GPU parallelization achieves order-of-magnitude acceleration in element-wise computations, while targeted optimizations further unlock the hardware potential.

Table 5. Performance of dshape_tensor_dshape with optimizations (Mesh: 554,394 elements).

Version	dshape1 and dshape2 are the same array. Time (ms)	dshape1 and dshape2 are different arrays. Time (ms)
Baseline	35.799	36.314
Unroll	20.847	26.890
Templated matrix product and dot product	1.969	3.689

Table 6. Global matrix assembly performance (Mesh: 103,635 nodes, 554,394 elements).

Task	CPU assembly time (ms)	GPU assembly time (ms)	Speedup
Matrix	327.538	0.842	389.02
RHS	48.254	0.116	415.64

5.3.2 Performance of Global Matrix Assembly Optimization

Table 6 shows assembly performance for the advection-diffusion matrix (Temperature) and the right-hand sides. Compared with CPU execution, GPU parallel assembly achieved speedups of up to 389.02× and 415.64×, respectively, highlighting the advantages of massive threading and high bandwidth. Compared with existing GPU studies on sparse matrix assembly, a substantially higher acceleration is achieved in the proposed method, demonstrating the superior performance of our kernel designs and data structures.

Regarding the parallel assembly, it is worth noting that multiple elements inevitably share nodes in unstructured meshes, which can theoretically introduce memory contention when using atomic operations. To address this, we evaluated alternative algorithms that avoid atomic operations. While this contention-free approach further reduced the sparse matrix update time (e.g., from approximately 5 ms to 3 ms in specific test stages), it required an explicit mesh preprocessing step. This preprocessing consumed several seconds, introducing an overhead that far outweighed the modest kernel-level savings. Therefore, the direct atomic operation algorithm was selected for its simplicity and overall efficiency. This design choice is further supported by recent evaluations of unstructured-grid CFD applications on GPUs, which demonstrate that NVIDIA V100 and A100 architectures inherently deliver exceptional performance on kernels dominated by double-precision atomic updates, often outperforming complex register-based aggregation or data restructuring methods (Stone et al., 2021).

Table 7. Pinned vs non-pinned memory transfer performance

Data	Size(Byte)	Non-pinned time(ms)	Non-pinned Bandwidth(GB/s)	Pinned time(ms)	Pinned BW(GB/s)
ndglno array	8870304	0.686	12.94	0.348	25.47
scalar field array	829080	0.086	9.67	0.047	17.60
vector field array	2487240	0.204	12.18	0.105	23.60
tensor field array	7461720	0.586	12.73	0.332	22.44

5.3.3 CPU-GPU Data Transfer and Overlap with Computation

In hotspot acceleration, CPU-GPU data transfer is another critical performance factor. In our system, CPU and GPU are connected via PCIe 4.0 $\times$16, with a bidirectional bandwidth of 32 GB/s. When the CPU has pinned memory, transfer bandwidth utilization is significantly higher than that with pageable memory. During Fluidity-Atmosphere computation, the type of data transferred frequently between CPU and GPU include scalar, vector, tensor field arrays, and the element-node connectivity array (**ndglno**). Table 7 compares pinned and non-pinned transfer performance, displaying an evidently superior bandwidth with pinned memory.

In addition, CUDA provides stream concurrency and asynchronous APIs such as cudaMemcpyAsync, facilitating simultaneous data transfer and kernel execution. The `transform_to_physical` kernels can run concurrently with transfers of subsequent physical field data. The transfer-computation overlap effectively mitigates communication bottlenecks and further leverages HPC system performance.

5.3.4 Performance of functional modules

After embedding GPU-accelerated element-wise computation and global assembly strategies into Fluidity-Atmosphere, we evaluated the overall performance at three levels: module, single timestep, and multi-process parallelism. In the restarting Mountainwave3D simulation, the pressure diffusion matrix was calculated only once, and the others were the average times of simulation timesteps which ranges from 3000 to 5000 s. Table 8 summarizes performance across major functional modules.

As the global sparse linear solver was not GPU-parallelized, the overall acceleration for the single-process execution remained limited. However, Fluidity-Atmosphere already supports MPI, enabling MPI+GPU hybrid execution for maximal performance. Table 9 compares the one timestep execution performance of the CPU-only and GPU-enabled configurations.

The results show that GPU acceleration achieved a **2.44$\times$ speedup** for single-process execution. With 4 MPI processes (each handling approximately 25,000 nodes and 140,000 elements), the hybrid MPI+GPU version achieved **8.57$\times$ speedup** compared with single CPU execution. Table 9 presents the average times of simulation timesteps which ranges from 3000 to 5000 s.

Table 8. GPU acceleration performance of major modules.

Functional module	CPU time (ms)	GPU time (ms)	Speedup
the advection-diffusion matrix (Temperature)	2563.158	11.566	221.60
the advection-diffusion matrix (WaterVapor)	2542.717	11.251	226.00
the advection–diffusion matrix (CloudWater)	2566.037	11.296	227.17
the advection-diffusion matrix (RainWater)	1877.409	9.481	198.02
the momentum matrix	12641.197	46.363	272.66
the divergence matrix	3657.007	21.497	170.12
the pressure diffusion matrix	3505.105	28.507	122.96
the projection matrix	1126.354	10.944	102.92

Table 9. The performance of GPU accelerated Fluidity-Atmosphere.

Version	Time (s)	Speedup (vs 1CPU)
CPU only	41.838	1.00
1 CPU +1 GPU	17.147	2.44
4 CPUs	13.376	3.13
4 CPUs+1 GPU	4.880	8.57

Notably, the 4 CPUs+1 GPU configuration achieves an 8.57× speedup , which is more than triple the speedup of the 1 CPU+1 GPU configuration (2.44×). This synergistic effect is attributed to two factors: first, the domain decomposition in multi-process execution allows the remaining CPU-bound linear solvers (PETSc) to benefit from improved cache locality on smaller sub-domains; second, multiple MPI processes can concurrently issue kernels to the GPU via CUDA streams, leading to higher hardware occupancy on the NVIDIA A100.As detailed in Table 10, this configuration effectively addresses the "bottleneck shift" dictated by Amdahl's Law. In the 1 CPU + 1 GPU setup, GPU matrix assembly is compressed to less than 1% of the total time, leaving the PETSc solver and other unaccelerated modules as the dominant bottlenecks. However, by leveraging 4 MPI processes, the absolute execution time of the PETSc solver is dramatically reduced from 5.60 s to 2.19 s, and the time for other unaccelerated modules drops from 11.43 s to 2.62 s. This multi-core mitigation of the remaining CPU bottlenecks, combined with sustained GPU efficiency, strictly validates the 8.57× overall performance gain.

6 Conclusions

This study focuses on the unstructured-mesh finite-element atmospheric model named Fluidity-Atmosphere, thereby addressing the computational bottlenecks in element-wise computations and matrix assembly. Leveraging the architectural features of the NVIDIA A100 GPU and the CUDA programming model, we designed and implemented GPU-oriented parallel optimiza-

Table 10. Comparison of computational performance and runtime distribution across different configurations (Unit: Time in seconds, Percentage of total timestep).

	Baseline (1 CPU)		1 CPU + 1 GPU		4 CPUs + 1 GPU	
Functional module	Time (s)	Percentage	Time (s)	Percentage	Time (s)	Percentage
Temperature Matrix	2.467	5.95%	0.0226	0.13%	0.0120	0.25%
WaterVapor Matrix	2.471	5.96%	0.0137	0.08%	0.0041	0.08%
CloudWater Matrix	2.494	6.02%	0.0139	0.08%	0.0038	0.08%
RainWater Matrix	1.799	4.34%	0.0104	0.06%	0.0029	0.06%
Momentum Matrix	12.093	29.19%	0.0313	0.18%	0.0384	0.79%
Pressure Correction Matrix	4.760	11.49%	0.0165	0.10%	0.0092	0.19%
Total PETSc (Setup + Solver)	5.212	12.58%	5.6045	32.68%	2.1921	44.92%
Other unaccelerated modules	10.136	24.46%	11.4341	66.68%	2.6175	53.63%
Total timestep duration	41.433	100.00%	17.147	100.00%	4.880	100.00%

tion strategies. The proposed high-performance GPU kernels substantially enhanced the computational efficiency while solving the momentum equations, advection equations, and stabilization term construction, thereby accelerating critical processes in atmospheric numerical simulations. For element-wise computations, template-based kernel design and deep optimizations achieved speedups ranging from tens to several hundred times. With asynchronous data transfer and MPI parallelization, an overall acceleration that is 8.57 times greater than that of the CPU version was achieved using four processes while ensuring correctness and stability in multi-time step simulations. The proposed parallelization methods substantially enhance the performance of atmospheric models on heterogeneous systems, remarkably extending the support for high-performance implementations of unstructured-mesh models. Future work will focus on further overcoming performance bottlenecks in the overall simulation, particularly by migrating the sparse linear solver (which is still CPU-dependent) to GPUs, thereby facilitating end-to-end acceleration.

Code and data availability. The original Fluidity model is distributed free of charge under the GNU Lesser General Public License (LGPL). The source code is publicly available from its official repository at https://fluidityproject.github.io/get-fluidity.html (AMCG, 2014). The version used in this study corresponds to Fluidity release 2025.12.

The GPU-accelerated extension developed in this work is permanently archived at Zenodo. The version used to generate all results presented in this paper corresponds to release v2 and is available at: https://doi.org/10.5281/zenodo.18799735 (Fu, 2026). All simulation data produced in this study are publicly available at https://doi.org/10.5281/zenodo.17824051 (Li et al., 2025).

The Zenodo archive includes: (1) the complete GPU-modified source files, (2) CUDA kernels and GPU interface implementation, (3) compilation scripts, (4) a detailed README file providing step-by-step instructions for reproducing the numerical experiments and figures reported in this paper.

Appendix A: Appendix A

```
template<int T_M>
__device__ double dot_product_t(double*__restrict__ a,
double* __restrict__ b){
    double r=0.0;
    #pragma unroll
    for(int i=0;i<T_M;i++){
        r += a[i]*b[i];
    }
    return r;
}

template<int T_M, int T_N, int T_K>
__device__ void __forceinline__  matmul_t(const double* a, const double* b,
double*C){
    for (int mi = 0; mi < T_M; ++mi) {
        for (int ki = 0; ki < T_K; ++ki) {
            #pragma unroll
            for (int ni = 0; ni < T_N; ++ni) {
                C[ni* T_M + mi] +=  a[ki * T_M + mi] * b[T_K * ni + ki];
            }
        }
    }
}

template<int T_dim, int T_loc1, int T_loc2, int T_ngi>
__device__ void dshape_tensor_dshape_unroll_t(const double *dshape1,
const double *tensor,const double *dshape2, const double *detwei, double *r){
   double tmp[T_dim]={0.0},mat_mul[T_dim]={0.0},dotproduct=0.0;
   for(int gi=0; gi< T_ngi;++gi){
```

```
        double gidetwei=detwei[gi];
        for(int iloc=0;iloc<T_loc1;iloc++){
          #pragma unroll
          for(int n=0;n<T_dim;++n){
            tmp[n]=dshape1[n*T_ngi*T_loc1+gi*T_loc1+iloc];
            mat_mul[n] = 0.0;
          }
          matmul_t<1,3,3>(tmp,&tensor[gi*T_dim*T_dim],mat_mul);
          for(int jloc=0;jloc<T_loc2;jloc++){
            #pragma unroll
            for(int n=0;n<T_dim;++n){
              tmp[n]=dshape2[n*T_ngi*T_loc2+gi*T_loc2+jloc];
            }
            dotproduct = dot_product_t<T_dim>(mat_mul,tmp);
            r[jloc*T_loc1+iloc]=r[jloc*T_loc1+iloc]+dotproduct*gidetwei;
          }
        }
      }
    }
```

Appendix B: Appendix B

```
template<int T_dim, int T_loc, int T_ngi>
__global__ void assemble_csr_matrix(double *result, double* values,
int *row_pointers, int *col_index, int *ndglno, const int ele_num){
    int element_index = blockIdx.x*blockDim.x + threadIdx.x;
    if(element_index>=ele_num) return;
    int loc1=T_loc;
    int loc2=T_loc;
    int mpos=0,base=0,upper_j=0,upper_pos=0,lower_j=0,lower_pos=0,
        this_pos=0,this_j=0;
    int inode[T_loc]={0};
    int j=0;
    double *element_result=result + element_index *T_loc * T_loc;
    int *row=nullptr;
```

```
int iloc=0, jloc=0,  n=0;
//ndglno stores the connectivity for neighbouring elements.
for(int i = 0; i < T_loc; ++i){
  inode[i]= ndglno[element_index * loc1 + i];
}
for(iloc=0;iloc<loc1;iloc++){
    //the index in Fluidity-Atmosphere Fortran code is from 1.
    base = row_pointers[inode[iloc]-1]-1;
    n = row_pointers[inode[iloc]] -1 - base;
    for(jloc=0;jloc<loc2;jloc++){
        if(element_result[jloc*loc1+iloc]==0)continue;
        row = col_index+base;
        upper_pos=n-1;
        upper_j=row[n-1]-1;
        lower_pos=0;
        lower_j=row[0]-1;
        mpos=-2;
        j = inode[jloc]-1;
        if (upper_j<j){
            mpos=-1;
        }else if (upper_j==j){
            mpos=upper_pos+base;
        }else if (lower_j>j){
            mpos=-1;
        }else if(lower_j==j){
            mpos=lower_pos+base;
        }

        while(((upper_pos-lower_pos)>1)&&(mpos==-2)){
            this_pos=(upper_pos+lower_pos)/2;
            this_j=row[this_pos]-1;
            if(this_j == (inode[jloc]-1)){
                mpos=this_pos+base;
            }
            else if(this_j > (inode[jloc]-1)){
```

```
                    upper_j=this_j;
                    upper_pos=this_pos;
                }
                else{
                    lower_j=this_j;
                    lower_pos=this_pos;
                }
            }
            if(mpos<0){
            }else{
                atomicAdd(values+mpos,element_result[jloc*loc1+iloc]);
            }
        }
    }
}
```

Appendix C: Appendix C

```
template<int T_loc>
__global__ void rhs_addto_kernel(GpuScalarField *field, const int ele_num,
double *gpu_ele_val){
    int  element_index = blockDim.x * blockIdx.x + threadIdx.x;
    if(element_index >=ele_num) return;
    //element_index will be used to find the node indexes starting from 1.
    element_index = element_index + 1;
    int nodes[T_loc];
    //get the node indexes of current element.
    gpu_ele_nodes_scalar(*field, element_index,nodes);
    double *ptr = gpu_ele_val + (element_index -1)*T_loc;
    #pragma unroll
    for(int i = 0; i < T_loc;++i){
        atomicAdd(&(*field).val[(nodes[i] -1)],*(ptr+i));
    }
}
```

Author contributions. **Conceptualization & Methodology**: Li L., Li J. and Li H. provided technical guidance and supervision.
Software & Investigation: Li L., Zheng X. and Fu X. developed the software and performed the testing.
Writing - Original Draft: Fu X. prepared the manuscript.
Writing - Review & Editing: Li L. and Li J. revised the manuscript. All co-authors reviewed and approved the final manuscript.

Competing interests. The authors declare there are no conflicts of interest for this manuscript.

Acknowledgements. This work is jointly supported by the National Key R&D Program of China (Grant No. 2023YFC3705701), the State Key Laboratory of Atmospheric Environment and Extreme Meteorology (2024ZD04). Dr. J. Li thanks for the technical support of the National Large Scientific and Technological Infrastructure "Earth System Numerical Simulation Facility" (Grant 2025-EL-PT-000886, https://cstr.cn/31134.02.EL).

References

AMCG: Fluidity manual, https://fluidityproject.github.io/get-fluidity.html, 2014.

AMD: CUDA to HIP API Function Comparison, https://rocm.docs.amd.com/projects/HIP/en/latest/reference/api_syntax.html, 2025.

Bogenschutz, P. A., Clevenger, T. C., Bradley, A. M., Caldwell, P. M., Beydoun, H., Mahfouz, N., Keen, N. D., Guba, O., Bertagna, L., Foucar, J., Zhang, J., and Donahue, A. S.: High Performance, High Fidelity: A GPU-Accelerated Doubly-Periodic Configuration of the Simple Cloud-Resolving E3SM Atmosphere Model Version 1 (DP-SCREAMv1), Journal of Advances in Modeling Earth Systems, 17, e2025MS005 127, https://doi.org/https://doi.org/10.1029/2025MS005127, 2025.

Conde, D. A. S., Ferreira, R. M. L., Canelas, R., Ricardo, A. M., and Mendes, L.: A Distributed-Heterogeneous Design for Explicit Hyperbolic Solvers. Application to Tsunami Urban Run-Up Modelling, Journal of Advances in Modeling Earth Systems, 17, e2024MS004 602, https://doi.org/https://doi.org/10.1029/2024MS004602, 2025.

Czarnul, P., Proficz, J., and Drypczewski, K.: Survey of Methodologies, Approaches, and Challenges in Parallel Programming Using High-Performance Computing Systems, Scientific Programming, 2020, 4176 794, https://doi.org/https://doi.org/10.1155/2020/4176794, 2020.

Davies, D. R., Wilson, C. R., Kramer, S. C., Piggott, M. D., Le Voci, G., and Collins, G. S.: Fluidity: A New Adaptive, Unstructured Mesh Geodynamics Model, in: EGU General Assembly Conference Abstracts, EGU General Assembly Conference Abstracts, p. 5364, 2010.

Farrell, P. E., Piggott, M. D., Pain, C. C., Gorman, G. J., and Wilson, C. R.: Conservative interpolation between unstructured meshes via supermesh construction, Computer Methods in Applied Mechanics and Engineering, 198, 2632–2642, https://doi.org/https://doi.org/10.1016/j.cma.2009.03.004, 2009.

Fu, X.: GPU-Accelerated Implementation of the Fluidity-Atmosphere Dynamical Core, https://doi.org/10.5281/zenodo.18799735, 2026.

Georgescu, S., Chow, P., and Okuda, H.: GPU Acceleration for FEM-Based Structural Analysis, Archives of Computational Methods in Engineering, 20, 111–121, https://doi.org/10.1007/s11831-013-9082-8, 2013.

Goddeke, D., Buijssen, S., Wobker, H., and Turek, S.: GPU acceleration of an unmodified parallel finite element Navier-Stokes solver, in: Proceedings of the 2009 International Conference on High Performance Computing and Simulation, HPCS 2009, pp. 12 – 21, https://doi.org/10.1109/HPCSIM.2009.5191718, 2009.

Jendersie, R., Lessig, C., and Richter, T.: A GPU parallelization of the neXtSIM-DG dynamical core (v0.3.1), Geosci. Model Dev., 18, 3017–3040, https://doi.org/10.5194/gmd-18-3017-2025, 2025.

Kiran, U., Gautam, S. S., and Sharma, D.: GPU-based matrix-free finite element solver exploiting symmetry of elemental matrices, Computing, 102, 1941–1965, https://doi.org/10.1007/s00607-020-00827-4, 2020.

Li, J., Zheng, J., Zhu, J., Fang, F., Pain, C. C., Steppeler, J., Navon, I. M., and Xiao, H.: Performance of Adaptive Unstructured Mesh Modelling in Idealized Advection Cases over Steep Terrains, Atmosphere, 9, https://doi.org/10.3390/atmos9110444, 2018.

Li, J., Fang, F., Steppeler, J., Zhu, J., Cheng, Y., and Wu, X.: Demonstration of a three-dimensional dynamically adaptive atmospheric dynamic framework for the simulation of mountain waves, Meteorology and Atmospheric Physics, 133, 1627–1645, https://doi.org/10.1007/s00703-021-00828-8, 2021.

Li, L., Fu, X., Zheng, X., Li, H., and Li, J.: Atmospheric Mountain Wave Simulation Dataset (Fluidity-Atmosphere), https://doi.org/10.5281/zenodo.17824052, 2025.

Macioł, P., Płaszewski, P., and Banaś, K.: 3D finite element numerical integration on GPUs, Procedia Computer Science, 1, 1093–1100, https://doi.org/https://doi.org/10.1016/j.procs.2010.04.121, 2010.

Michalakes, J.: HPC for Weather Forecasting, in: Parallel Algorithms in Computational Science and Engineering, edited by Grama, A. and Sameh, A. H., pp. 297–323, Springer International Publishing, Cham, ISBN 978-3-030-43736-7, https://doi.org/10.1007/978-3-030-43736-7_10, 2020.

Mills, R. T., Adams, M. F., Balay, S., Brown, J., Dener, A., Knepley, M., Kruger, S. E., Morgan, H., Munson, T., Rupp, K., Smith, B. F., Zampini, S., Zhang, H., and Zhang, J.: Toward performance-portable PETSc for GPU-based exascale systems, Parallel Computing, 108, 102 831, https://doi.org/https://doi.org/10.1016/j.parco.2021.102831, 2021.

Müller, E. H., Scheichl, R., and Vainikko, E.: Petascale solvers for anisotropic PDEs in atmospheric modelling on GPU clusters, Parallel Computing, 50, 53–69, https://doi.org/https://doi.org/10.1016/j.parco.2015.10.007, 2015.

NVIDIA: CUDA C++ Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2025a.

NVIDIA: Sparse Matrix Formats, https://docs.nvidia.com/nvpl/latest/sparse/storage_format/sparse_matrix.html, 2025b.

Petersen, M. R., Asay-Davis, X. S., Barthel, A. M., Begeman, C. B., Bishnu, S., Brus, S. R., Jones, P. W., Kang, H.-G., Kim, Y., Mametjanov, A., O'Neill, B., Ringel, K. K., Smith, K. M., Sreepathi, S., Van Roekel, L. P., and Waruszewski, M.: The Ocean Model for E3SM Global Applications: Omega Version 0.1.0. A New High-Performance Computing Code for Exascale Architectures, EGUsphere, 2025, 1–37, https://doi.org/10.5194/egusphere-2025-3500, 2025.

Ratnakar, S. K., Sanfui, S., and Sharma, D.: Graphics Processing Unit-Based Element-by-Element Strategies for Accelerating Topology Optimization of Three-Dimensional Continuum Structures Using Unstructured All-Hexahedral Mesh, Journal of Computing and Information Science in Engineering, 22, 021 013, https://doi.org/10.1115/1.4052892, 2021.

Sanfui, S. and Sharma, D.: A three-stage graphics processing unit-based finite element analyses matrix generation strategy for unstructured meshes, International Journal for Numerical Methods in Engineering, 121, 3824–3848, https://doi.org/https://doi.org/10.1002/nme.6383, 2020.

Sanfui, S. and Sharma, D.: Symbolic and Numeric Kernel Division for Graphics Processing Unit-Based Finite Element Analysis Assembly of Regular Meshes With Modified Sparse Storage Formats, Journal of Computing and Information Science in Engineering, 22, https://doi.org/10.1115/1.4051123, 2021.

Simek, V., Dvorak, R., Zboril, F., and Kunovsky, J.: Towards Accelerated Computation of Atmospheric Equations Using CUDA, in: 2009 11th International Conference on Computer Modelling and Simulation, pp. 449–454, https://doi.org/10.1109/UKSIM.2009.25, 2009.

Stone, C. P., Walden, A., Zubair, M., and Nielsen, E. J.: Accelerating unstructured-grid CFD algorithms on NVIDIA and AMD GPUs, in: 2021 IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3), pp. 19–26, https://doi.org/10.1109/IA354616.2021.00010, 2021.

Sulyok, A. A., Balogh, G. D., Reguly, I. Z., and Mudalige, G. R.: Locality optimized unstructured mesh algorithms on GPUs, Journal of Parallel and Distributed Computing, 134, 50–64, https://doi.org/https://doi.org/10.1016/j.jpdc.2019.07.011, 2019.

Takle, E. and Russell, R.: Applications of the finite element method to modeling the atmospheric boundary layer, Computers & Mathematics with Applications, 16, 57–68, https://doi.org/https://doi.org/10.1016/0898-1221(88)90024-7, 1988.

Thibault, J. and Senocak, I.: CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows, in: 47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition, https://doi.org/10.2514/6.2009-758, 2012.