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.

Response to Reviewer #2

We sincerely thank the reviewer for the careful and thorough evaluation of our manuscript and for the positive assessment of our work. We are encouraged that the reviewer finds the study to be solid, well-presented, and of practical value to the atmospheric modeling community.

We also greatly appreciate the reviewer's constructive and insightful comments, particularly regarding performance interpretation, consistency of reported results, and reproducibility. These suggestions have helped us to significantly improve the clarity, rigor, and completeness of the manuscript.

In response, we have made substantial revisions, including:
(1) clarifying the performance implications of atomic operations and explaining our algorithmic choices based on A100 hardware capabilities;
(2) discussing the compatibility of our GPU implementation with adaptive mesh refinement (AMR) and quantifying the minimal data transfer overheads;
(3) introducing a detailed runtime distribution analysis to clearly illustrate the "bottleneck shift" towards the CPU-bound linear solver;
(4) ensuring consistency in the reported speedups and providing a robust explanation for the nonlinear scaling observed in the hybrid MPI+GPU configurations;
(5) improving reproducibility by explicitly including mesh sizes and kernel invocation counts in the relevant table captions;
(6) strengthening the logical flow by explicitly stating how the initial profiling data guided our GPU optimization priorities;
and (7) verifying the appendix code snippets to ensure complete consistency with the main text.
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.

Reviewer Comment 1:

In Section 4.3, the authors use atomic operations to handle concurrent updates during parallel assembly and report speedups up to 389×. Given that multiple elements often share nodes in unstructured meshes, atomic contention may affect performance. It would be helpful to briefly comment on the observed contention level (e.g., cache hit rate, warp efficiency) or explain why this does not become a bottleneck in the current test case (e.g., hardware capabilities of the A100).

Response:
We completely agree with this observation. Unstructured meshes inevitably introduce memory contention during global assembly. During our development, we did evaluate contention-free algorithms (e.g., mesh coloring) to bypass atomicAdd. However, while these methods slightly reduced the sparse matrix update time (from ~5 ms to ~3 ms), they required explicit CPU-side mesh preprocessing that consumed several seconds. This overhead far outweighed the millisecond-level savings in the kernel. Furthermore, recent literature confirms that the NVIDIA

V100 and A100 architectures are highly optimized for double-precision atomic updates, rendering complex restructuring methods less impactful on these specific GPUs.

Changes in manuscript: We have added a dedicated paragraph discussing this trade-off and hardware capability in Section 5.3.2:

"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... This preprocessing consumed several seconds, introducing an overhead that far outweighed the modest kernel-level savings. Therefore, the direct atomic operation algorithm was selected... This design choice is further supported by recent evaluations of unstructured-grid CFD applications on GPUs...."

Reviewer Comment 2:

The Introduction highlights anisotropic adaptive meshing as a key feature of Fluidity-Atmosphere. However, the GPU performance evaluation is conducted with a static mesh. Please clarify whether the current GPU implementation supports AMR, and what additional overheads it introduces.

Response:
Thank you for pointing this out. While we used a static mesh for this specific validation experiment to ensure strict 1:1 floating-point comparison with the CPU baseline over thousands of steps, our GPU kernels are natively designed to fully support Adaptive Mesh Refinement (AMR). The only additional overhead during an AMR step is the PCIe transfer of the updated element-node connectivity array (ndglno) to the GPU. Based on our bandwidth testing, this takes less than a millisecond, which is negligible relative to the overall timestep.

Changes in manuscript: We have clarified this in Section 5.2 (Validation Methodology):

"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)... As demonstrated by our data transfer analysis (Table 7), such transfers (e.g., 0.348 ms) are negligible..."

Reviewer Comment 3:

Several kernels achieve speedups of two to three orders of magnitude (Tables 5 and 9), the single-process end-to-end speedup is 3.36× (Table 10). This gap is understandable given that the linear solver remains on CPU. It would be valuable to include a brief analysis of how the runtime distribution across modules changes after GPU acceleration — specifically, what proportion of time the linear solver now occupies. This would clearly illustrate the "bottleneck shift" and

motivate future work on GPU-based solvers.

Response:
This is an excellent suggestion. To clearly illustrate the Amdahl's Law bottleneck shift, we have introduced a new, comprehensive table (Table 10) that breaks down both the absolute time and the percentage of the total timestep for each core module across three configurations: Baseline (1 CPU), 1 CPU + 1 GPU, and 4 CPUs + 1 GPU. The data clearly shows that after GPU acceleration, matrix assembly drops to under 1% of the total time, while the unaccelerated PETSc solver and other CPU modules surge to occupy over 85% of the execution time.

Changes in manuscript: We added Table 10 and the accompanying analysis in Section 5.3.4:

"...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..."

Reviewer Comment 4:

The claimed "10.28× speedup" in the abstract is inconsistent with the data presented in Table 10: the single-GPU configuration achieves only 3.36× speedup, while the 10.28× figure corresponds to the 4 MPI + GPU hybrid configuration. The abstract does not specify the configuration... Furthermore, the speedup achieved by the 4 MPI + GPU configuration relative to a single CPU is substantially higher than that of a single GPU relative to a single CPU. This nonlinear behavior requires a proper explanation...

Response:
We sincerely apologize for the data inconsistency regarding the speedup figures, which was an oversight from an earlier draft. We have strictly standardized the overall multi-process speedup to 8.57x and the single-process speedup to 2.44x throughout the abstract and text.
Regarding the nonlinear/super-linear scaling observed in the 4 CPUs + 1 GPU setup, this is directly tied to the "bottleneck shift" discussed in Comment 3. Because the PETSc solver is now the primary bottleneck, utilizing 4 MPI processes divides the domain, granting the CPU solver significantly better cache locality. Our new profiling shows the PETSc solver time plummets from 5.60s (1 CPU) to 2.19s (4 CPUs).

Changes in manuscript: 1. Corrected the speedup values in the Abstract, Section 5.3.4, and Conclusions; 2. Added a robust explanation of the nonlinear scaling in Section 5.3.4:

"Notably, the 4 CPUs+1 GPU configuration achieves an 8.57x speedup, which is more than triple the speedup of the 1 CPU+1 GPU configuration (2.44x). 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... the absolute execution time of the PETSc solver is dramatically reduced from 5.60

s to 2.19 s..."

Reviewer Comment 5:

Consider adding a brief note on the number of kernel calls or the mesh size used in these tests to aid reproducibility in Tables 4–6.

Response:
Agreed. To save readers from having to search the text for experimental parameters, we have explicitly added the mesh size and kernel invocation details directly into the table captions.

Changes in manuscript: Modified the captions for Tables 3, 4, 5, and 6 (e.g., Table 3 caption now reads: "GPU Performance of GPU parallel element data access (Mesh: 103,635 nodes, 554,394 elements; timings are averaged over 1,000 invocations).")

Reviewer Comment 6:

The profiling data in Table 1 is presented, but its role in guiding optimization priorities is not discussed. Please clarify how this timing breakdown informed the selection of modules for GPU acceleration.

Response:
We have updated the text immediately following Table 1 to establish a stronger logical flow. We now explicitly state that because matrix construction occupies roughly 60% of the timestep, it was prioritized as the most effective target for GPU offloading.

Changes in manuscript: Added the following sentence directly after Table 1 in Section 2:

"As quantified in Table 1, the processes of constructing matrices 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 assembly is the most effective strategy for improving the overall performance of the Fluidity-Atmosphere model."

Reviewer Comment 7:

Appendices: The CUDA code snippets are valuable for community reuse. Please verify that they are complete and consistent with the descriptions in the main text.

Response:
We have thoroughly reviewed the CUDA code snippets provided in Appendices A, B, and C. We confirm they are fully consistent with our descriptive text (such as the use of atomicAdd, binary search, and templated optimizations) and are complete regarding the core logic of the matrix assembly and numerical integration routines. No structural changes were necessary.

Changes in manuscript: We have reviewed and verified the appendix code in Appendix A–C.

We hope these revisions comprehensively address your concerns and successfully demonstrate the efficacy of our GPU implementation. Thank you again for your time and expertise.

Response to Reviewer #3

We sincerely thank the reviewer for the thorough evaluation and highly constructive comments. We carefully analyzed all suggestions and substantially revised the manuscript to improve its technical clarity, numerical validation, and performance analysis.

Specifically, we clarified that the present study focuses on low-order P1 continuous Galerkin discretization and further distinguished the CG operators accelerated in this work from the DG components used elsewhere in the Fluidity-Atmosphere framework. We refined the descriptions of the GPU execution workflow, including the device-side sparse matrix assembly procedure, CPU–GPU data transfer process, and the interaction with PETSc-based solvers.

To better evaluate the effectiveness of the GPU implementation, we added detailed hardware utilization and bottleneck analyses based on NVIDIA Nsight Compute profiling, including FP64 throughput and memory bandwidth utilization. We also expanded the discussion on the suitability of assembled sparse operators versus matrix-free methods for low-order unstructured FEM frameworks. Furthermore, the validation methodology was significantly strengthened through additional fixed-mesh transient simulations, demonstrating machine-precision agreement between CPU and GPU results.

In addition, we revised Algorithm 1, improved the explanations of template parameters and implementation details in the appendices, clarified the CPU baseline and MPI terminology, and refined numerous technical descriptions and minor language issues throughout the manuscript to improve overall rigor.

Detailed point-by-point responses to all comments, along with a marked-up version of the revised manuscript highlighting the specific changes, are provided in the attached files.

Comment 1:

Reviewer Comment:
Please specify explicitely in the paper at some point what kind of Finite Element is used in the discretisation and in the tests of the code. Thus far, the only explicit statement in the the manuscript is that you are using continuous elements. However, that still leaves a lot of possibilites especially in the context of a mixed element discretisation for a CFD type problem.
This information is an important one, as it makes a significant difference computationally and w.r.t. performance optimisation, if you are using a P1 Lagrange element pair or a P3 one, or maybe a conforming Crouzeix-Raviart element.
From the text (e.g. Fig. 2 or the mentioning of a 4 x 4 local element matrix on p. 12) it seems that you are using a classical P1 element.

Response:
We sincerely thank the reviewer for this insightful observation. In the specific simulations and

benchmarks presented in this study, the dynamic framework utilizes linear Lagrange elements (P1) for the continuous Galerkin (CG) discretization. This inherently yields 4 degrees of freedom per tetrahedral element, which corresponds to the 4×4 local element matrices mentioned in the manuscript. We apologize for not stating this explicitly in the initial draft, as this information is indeed crucial for evaluating computational complexity and optimization strategies. We have now added a clear statement regarding the P1 element choice in Section 2 to eliminate any ambiguity.

Changes in manuscript:

Section 2: Target Scientific Application Definition

"Specifically, in the performance evaluations and benchmarks presented in this study, the dynamic framework predominantly utilizes linear Lagrange elements (P1) for the continuous Galerkin (CG) discretization. This choice inherently yields 4 degrees of freedom per tetrahedral element, resulting in the 4×4 local element matrices evaluated during the simulation."

Comment 2

Reviewer Comment:

In your experiments you measure runtimes and provide speed-up values comparing your GPU implementation to the CPU one. One the one hand the time-to-solution is, of course, a highly important aspect, since this is directly affecting the user. However, in order the evaluate the quality and potential of your optimisations other aspects would also be of significant interest.

a) As you correctly state in Table 2 the theoretical FP64 peak performance of the A100 PCIe is 9.7 TFLOP/s. What percentage of this is reached by your GPU code?

b) You repeatedly mention the "high bandwidth and massive parallelism" of GPUs. However, from Table 5 I would deduce that roughly a factor of speed-up of 20 results from optimisations such as "loop unrolling" and "templating". These would also benefit the CPU code of course. But, if I understand correctly, these are not in the version of the CPU code used for comparison, correct?

c) In line 342 you mention "serial CPU execution", does that mean you are comparing a CPU code running on a single of the AMD Epyc's 64 cores to the GPU version? Does this hold for all results apart from the ones given in Table 9?

d) What is the meaning of "CPU" in Table 9? Are you talking about a setup with 4 physical EPYC CPUs and one GPU here, or is this implying that you use MPI to parallelise the CPU part of the code and run it on 4 cores of the EPYC's 64 cores?

Response:

a) We appreciate the reviewer's question regarding hardware utilization. Using NVIDIA Nsight Compute (NCU), we profiled our GPU implementation. The compute-intensive element integration kernel (dshape_tensor_dshape_unroll_matmul_t) reaches 8.50% of the 9.7 TFLOP/s theoretical FP64 peak (Compute SM Throughput). The global assembly kernels (assemble_csr_matrix and rhs_addto_kernel) reach 7.93% and 18.29%, respectively.

These compute utilizations directly reflect the inherent memory-bound nature of unstructured finite element methods (FEM). The arithmetic intensity is inherently low due to the indirect memory fetching of mesh topology and nodal data required for local evaluations and sparse matrix insertions.

Consequently, a more relevant metric for these kernels is Memory Throughput. Our NCU profiling shows the element integration kernel achieves 90.44% Memory Throughput, while the global matrix assembly and RHS addition kernels achieve 79.48% and 82.98%, respectively. Reaching approximately 80-90% of the A100's memory bandwidth under irregular memory access patterns indicates an efficient hardware utilization. We have added a "Hardware Utilization and Bottleneck Analysis" paragraph in Section 5.3 to clarify this point in the manuscript.

Changes in manuscript:

Section 5.3.3: Hardware Utilization and Bottleneck Analysis

"To further comprehend the GPU execution efficiency, we analyzed the hardware utilization of our core kernels using NVIDIA Nsight Compute. The theoretical FP64 peak performance of the A100 GPU is 9.7 TFLOP/s. However, in unstructured FEM frameworks, performance is predominantly bounded by memory bandwidth rather than compute capabilities. ... , maintaining such high memory bandwidth utilization demonstrates that our data layout and atomic-based assembly strategies effectively saturate the hardware limits for these memory-intensive workloads."

Response:

b) You make a highly valid point. Algorithmic optimizations like loop unrolling and templating indeed benefit CPU execution as well. To clarify our experimental baseline, the CPU version used for comparison is the official, unmodified production release of the Fluidity-Atmosphere framework. We chose this as our baseline to provide a realistic and practical reference point for current users of the model. While we did not manually inject templating into the original Fortran CPU source code, we ensured a fair hardware comparison by compiling the CPU code with aggressive compiler optimization flags (-O3 -ffast-math). Modern compilers (like GCC/Gfortran) are highly efficient at automatically applying loop unrolling and instruction scheduling for standard CPU architectures under the -O3 flag. In contrast, explicit loop unrolling and template metaprogramming were strictly necessary for the GPU implementation. Due to the GPU's unique warp-based execution model and strict register allocation limits, the nvcc compiler often struggles to automatically unroll complex, nested element-wise loops or resolve sizes at compile-time without explicit directives. Therefore, these manual optimizations on the GPU were applied to unlock the hardware's inherent capabilities rather than to create an algorithmic advantage over the CPU. We have added a clarification regarding this baseline comparison in Section 3.3 of the revised manuscript.

Changes in manuscript:

Section 3.3 GPU Parallelization of Element-wise Integration

"It should be noted that the CPU baseline used for comparison represents the official, unmodified production code of Fluidity-Atmosphere, compiled with aggressive optimization flags \texttt{-O3 -ffast-math}. ... Thus, these GPU-specific manual optimizations are implemented to fully unlock the hardware's architectural potential rather than to introduce an algorithmic disparity."

Response:

c&d) We apologize for the ambiguity in our terminology. The reviewer's understanding of the configuration is perfectly accurate. When we referred to "serial CPU execution" or "1 CPU", we

meant 1 single CPU core (i.e., 1 MPI process) running on the 64-core AMD EPYC 7713 processor. This applies to all single-process baseline results in the manuscript.
Similarly, in Table 9 and the related text, "4 CPUs" specifically means 4 CPU cores (i.e., 4 MPI processes) utilized on that same single physical AMD EPYC processor, paired with one GPU. It does not refer to 4 distinct physical CPU sockets. To eliminate any confusion, we have systematically reviewed the entire manuscript and updated the terminology from "CPU" to "CPU core" or "MPI process" wherever appropriate, particularly in Table 9, Table 10, and their corresponding discussions.

Comment 3

Reviewer Comment:
You repeatedly mention the high-bandwidth of GPUs. It would be nice to explain in more detail what you are refering to with this. Accessing host memory from the GPU through PCIe has a bandwidth of 32 GB/s (following Table 2). This is significantly slower than the maximal bandwidth for access to memory by the AMD EPYC. The latter being around 204.8 GB/s. So you are presumably refering to the bandwidth to the A100's HBM (device memory) which is 1.56 TB/s, aren't you? I'd also suggest to add these two hardware characteristics to Table 2.

Response:
We greatly appreciate the reviewer’s precise technical correction. We completely agree that the term "high bandwidth" was used too loosely in the original manuscript. The true computational advantage of the GPU for memory-bound unstructured FEM kernels comes from the ultra-fast Device Memory (1.56 TB/s), rather than the PCIe interconnect (32 GB/s) which is indeed slower than the host CPU's memory bandwidth (204.8 GB/s).
Following your excellent suggestion, we have updated Table 2 to explicitly include both the AMD EPYC memory bandwidth and the NVIDIA A100 device memory bandwidth. Furthermore, we have refined the phrasing throughout the manuscript (e.g., in Sections 1, 3.3, and 4.1) to clearly specify "high-bandwidth device memory" whenever discussing the GPU's memory advantages, thereby ensuring technical rigor.

Comment 4

Reviewer Comment:
As a note, the general trend in Finite-Element simulations for HPC seems to be to discard assembling a global matrix altogether and using matrix-free methods, see e.g. Kronbichler et al. (2012, 2019), Rudi (2015) or May (2015) to name a few. It would be appreciable to at least comment on this in your contribution.

Response:
We deeply appreciate the reviewer for highlighting this important trend in HPC finite-element simulations. We completely agree that matrix-free methods have become state-of-the-art, but as you insightfully pointed out in Comment 1, our current dynamic framework relies on linear Lagrange elements (P1).

The transition to matrix-free evaluation is primarily driven by the performance characteristics of high-order FEM. As polynomial degree increases in high-order elements, the memory access and floating-point operations per degree of freedom (DoF) grow rapidly under a full-assembly framework, severely bottlenecking performance. Matrix-free methods, especially when combined with sum factorization, effectively suppress this per-DoF memory and computational complexity, thereby increasing computational intensity and overall efficiency (Fischer et al., 2020).
However, for low-order discretizations like our P1 elements, the memory access per DoF is already minimal and remains largely comparable between fully assembled and matrix-free operators. Consequently, applying matrix-free methods to low-order elements does not yield the significant bandwidth savings seen in high-order methods. Furthermore, explicitly assembling the global sparse matrix allows our framework to leverage highly optimized, generic Sparse Matrix-Vector multiplication (SpMV) routines and sophisticated algebraic preconditioners (such as Algebraic Multigrid or ILU via PETSc), which are indispensable for our implicit time-stepping schemes.
We have added a comprehensive discussion regarding this design choice and the boundary between low-order assembly and high-order matrix-free methods in Section 4. We also cited the excellent references you provided alongside the scalability analysis by Fischer et al.

Changes in manuscript:
Section 4: GPU Parallelization of Matrix Assembly
"Before detailing our assembly strategy, it is worth noting that a prominent trend in high-performance finite-element simulations is to discard global matrix assembly in favor of matrix-free (or partially assembled) operator evaluations \citep{RN_Kronbichler2012, RN_Kronbichler2019, RN_Rudi2015}. ... Therefore, accelerating the global sparse matrix assembly remains the most rational and impactful optimization pathway for this class of low-order dynamical cores."

Comment 5

Reviewer Comment:
One of your key arguments (see e.g. line 162) is that element-wise computations are strictly local w/o cross-element dependencies. While that is mostly valid for continuous elements, I take from Li (2021) that Fluidity-Atmosphere is using a mixed DG/CG discretisation. Now in DG you typically have coupling to neighbouring elements, because your bilinear forms contain averages, jumps and fluxes on or over the facets of your tetrahedrons. Can you comment on that?

Response:
The reviewer makes a highly valid point, and we appreciate the careful reading of our previous work (Li et al., 2021). It is true that Fluidity-Atmosphere employs a mixed DG/CG formulation, and in the DG components (such as solving the continuity equation), calculating numerical fluxes and jumps inherently requires cross-element coupling over the facets.
However, as stated in Section 2 (around line 60), the GPU acceleration efforts and performance evaluations presented in this specific study focus exclusively on the Continuous Galerkin (CG) scheme (e.g., the momentum and advection-diffusion solvers). In the CG formulation, the

element-wise numerical integration to construct local mass and stiffness matrices is mathematically strictly local. The topological dependence on neighboring elements only manifests during the memory access phase of the subsequent global matrix assembly.
To prevent any confusion for readers familiar with our mixed formulation, we have clarified this distinction in Section 3.1, explicitly emphasizing that our "strictly local" claim applies specifically to the CG element-wise integrations addressed in this paper.

Changes in manuscript:
Section 3.1: Data Structures and Parallelization Strategy
"It is important to clarify that while Fluidity-Atmosphere utilizes a mixed CG/DG discretization, the GPU acceleration in this study focuses exclusively on the Continuous Galerkin (CG) operators (e.g., for the momentum and advection-diffusion equations). In the CG formulation, the numerical integration to evaluate local matrices remains mathematically entirely local. Unlike DG methods, which require calculating inter-element fluxes and jumps over facets, the CG element-wise computations do not introduce cross-element mathematical dependencies during the integration phase. The global coupling arising from shared mesh nodes is manifested exclusively during the global matrix assembly phase through indirect memory access."

Comment 6

Reviewer Comment:
Concerning Algorithm 1 I have several questions:
a) As you appear to be using P1 elements, the gradients of the shape functions are constant. So why do you need to recompute these at every Gauss point?
b) You do not explicitely state it, but it seems to me that you are using a standard affine mapping from the reference to the actual element. In this case the Jacobian matrix is also constant. Thus, it can be computed once for the element and needs not be recomputed at each Gauss point?
c) In general precomputing and storing some quantities can improve performance, see e.g. Kronbichler (2012), Boehm (2025). Have you considered this, or can comment on it?

Response:
The reviewer makes an excellent and precise point: for linear P1 elements under standard affine mappings, both the shape function gradients and Jacobian matrices are constant and should not be redundantly recomputed at every Gauss point.
Prompted by your comment, we reviewed our manuscript and realized that the pseudo-code in Algorithm 1 was oversimplified and did not fully reflect our actual implementation. In the Fluidity-Atmosphere codebase (for both the CPU and our ported GPU versions), there is an explicit conditional check: if (x_nonlinear .or. gi==1). For affine mappings (where x_nonlinear is false, such as with P1 elements), the computationally expensive operations—including the construction of the Jacobian matrix, its determinant, and its inverse—are executed only once at the first Gauss point. For all subsequent Gauss points, the cached constant values are directly reused.
We have updated Algorithm 1 to accurately reflect this conditional optimization logic and added explanatory text in Section 3.2 to remove any ambiguity. We appreciate you pointing this out, as it helped us improve the technical precision of the manuscript.

Regarding the precomputation strategy suggested by the reviewer (e.g., Kronbichler, 2012; Boehm, 2025): while highly effective on CPUs, unstructured FEM on GPUs is predominantly bottlenecked by memory bandwidth rather than compute capability. Storing these arrays for millions of elements would drastically increase the global memory footprint and memory traffic. Therefore, we deliberately adopt a "compute-on-the-fly" strategy, leveraging the GPU's abundant floating-point performance to conserve precious memory bandwidth.

Changes in manuscript:
Section 3.2: GPU Parallel Coordinate Transformation
"Algorithm 1 outlines the transform_to_physical procedure, where input and output arrays are stored contiguously to maximize memory efficiency. ... Furthermore, rather than precomputing and storing these geometric quantities, which would exacerbate memory traffic in this heavily memory-bound framework, we employ a "compute-on-the-fly" strategy, trading abundant GPU arithmetic cycles to conserve precious memory bandwidth \citep{RN_Kronbichler2012, RN_Boehm2025}."

Comment 7

Reviewer Comment:
Concerning your statement in Section 3.4 on the function-call overhead of these lightweight functions, is there no option to force the compiler to do inlining? Also I fail to see why there should be a problem with indirections in the case of using LAPACK's DPSEV? This is dense linear algebra after all. Also a 6 x 6 matrix of doubles occupies 288 bytes, shouldn't that easily fit into a typical 32K L1 data cache?

Response:
We fully agree that a 6x6 double-precision matrix (288 bytes) easily fits within the L1 data cache, and our original mention of cache misses for the dense solver was inaccurate.
However, regarding compiler inlining, standard library routines like LAPACK's DSPEV are typically pre-compiled external libraries. Without aggressive Link-Time Optimization (LTO), the host compiler cannot force-inline these external calls across translation units into the tight element loop. Consequently, calling such library routines millions of times incurs profound function-call overhead. Furthermore, standard dense solvers include internal argument validation and branching that entirely dwarf the actual floating-point arithmetic for tiny matrices. Our custom GPU __device__ solvers bypass this by being fully inlineable and stripping away all external overhead. We have rewritten this section to accurately reflect the true source of these overheads.

Changes in manuscript:
Section 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$\times$6 systems or computing eigen-decompositions of 3$\times$3 matrices. ... To address this challenge, this study implements dedicated GPU-based small-matrix solvers and eigen-decomposition kernels as inline \textbf{\texttt{__device__}} operators."

Comment 8

Reviewer Comment:
With respect to the global matrix assembly, could you please provide more details on the procedure? Where does the assembly happen? Is the matrix first assembled in device memory and afterwards copied to the host memory, so that it can be used in PETSc? Or is the strategy different? I do not see that this is explained somewhere and there is also no mentioning of something in this direction in section 5.3.3.

Response:
We apologize for this omission in our workflow description. The reviewer's hypothesis is completely accurate. To clarify the complete procedure: the global matrix assembly takes place entirely within the GPU's device memory. After the element-wise computations are completed on the GPU, a dedicated assembly kernel (as detailed in Appendix B) uses atomicAdd to accumulate the local element matrices directly into the pre-allocated global sparse matrix arrays (e.g., the CSR/BCSR values array) residing in the device memory. Once the parallel assembly kernel finishes execution, the fully populated global matrix arrays are copied back from the GPU device memory to the CPU host memory. Finally, this assembled matrix is passed to the CPU-based PETSc library for the subsequent linear system solve.
To ensure transparency and completeness, we have added a concluding paragraph to Section 4.3 to explicitly detail this execution pipeline, and cross-referenced this data transfer in Section 5.3.3.

Changes in manuscript:
Section 4.3: GPU Parallel Element-Wise Assembly
"To summarize the complete execution pipeline, the global matrix assembly occurs entirely within the GPU device memory. Following the element-wise computations, the local element matrices are accumulated into the global sparse matrix arrays (i.e., the \texttt{values} array in CSR/BCSR format), which are pre-allocated on the GPU. This is achieved using the aforementioned parallel assembly kernel equipped with hardware-level \texttt{atomicAdd} operations. Upon the completion of the assembly kernel, the fully constructed global matrix arrays are transferred back to the CPU host memory via PCIe. The assembled matrix is then handed over to the CPU-bound PETSc library to perform the subsequent large-scale linear system solve."

Section 5.3.3: CPU–GPU Data Transfer and Overlap with Computation
"During Fluidity-Atmosphere computation, the type of data transferred frequently between the CPU and GPU includes scalar, vector, and tensor field arrays, the element–node connectivity array (\textbf{ndglno}), \textbf{as well as the fully assembled global sparse matrix arrays returning from the GPU device memory to the host for PETSc solvers}."

Comment 9

Reviewer Comment:
Concerning your validation strategy, I must admit that I find this not overly convincing. I agree

with you that computations on the CPU and GPU and with different codes will not give bitwise identical results, which in the case of AMR indeed might lead to different meshes developping. However, assuming that both approaches leads to a sufficiently accurate approximate solution it should be perfectly okay to compare these as this is a FEM approach, shouldn't it? Maybe I do not understand how the "significant interpolation errors" arise? Anyway, wouldn't it be more convincing to use a fixed fine mesh for both cases and perform simulations that include the non-stationary initial part?

Response:

We sincerely appreciate this constructive suggestion. We fully agree that cross-mesh interpolation introduces numerical noise, and a fixed-mesh simulation starting from t=0 is the most rigorous way to verify mathematical consistency.

Because a globally uniform fine mesh is computationally prohibitive for this 3D problem, we adopted a targeted static-mesh approach. Specifically, we extracted the refined unstructured mesh generated at t=3000 s and used it as a fixed grid for both the CPU and GPU to simulate the non-stationary initial phase from t=0 to t=3000 s. This ensures an identical degree-of-freedom layout and completely eliminates spatial interpolation errors.

The new results (shown in the updated Figs. 4b and 4c) demonstrate that the peak absolute differences at t=3000 s are $1.44*10^{-14}$ for vertical velocity and $4.73*10^{-15}$ for potential temperature perturbation. These differences strictly fall within the double-precision machine epsilon, perfectly validating the arithmetic correctness of our GPU operators.

We have updated Figure 4 to include this new transient validation alongside our original steady-state test, and substantially revised Section 5.2 to reflect this improved methodology.

Changes in manuscript:

Section 5.2: Validation Methodology

"In scientific applications demanding high-precision floating-point operations, microscopic rounding differences between CPU and GPU architectures are inevitable. In a fully dynamic simulation, these bit-wise differences can cause the Adaptive Mesh Refinement (AMR) algorithm to split elements differently over time. ... These minuscule variations strictly correspond to the double-precision machine epsilon, confirming that the GPU-accelerated operators are fundamentally and mathematically consistent with the original CPU baseline across all simulation phases."

Comment 10

Reviewer Comment:

What precisely is shown in Table 3? What is or what does ele_val_scalar?

Response:

We apologize for the lack of clarity. In Fluidity-Atmosphere, ele_val_scalar (along with its vector and tensor counterparts) is a data-gathering routine. Because node-based physical fields (e.g., density, velocity) are stored non-contiguously in global memory on unstructured meshes, these routines gather the scattered node values into contiguous local arrays for each element prior to

numerical integration. Table 3 demonstrates the GPU's ability to accelerate this heavily memory-bound "gather" operation. We have updated the text in Section 5.3.1 to explicitly define the purpose of these functions.

Changes in manuscript:
Section 5.3.1: GPU Element-wise Computation Performance
"Table 3 compares GPU and CPU performance in scalar, vector, and tensor data access. Specifically, functions such as \texttt{ele_val_scalar} serve as data-gathering routines that collect scattered node-based field values from global arrays into contiguous local memory arrays for each element prior to numerical integration. The results demonstrate the GPU's efficiency in accelerating these heavily memory-bound 'gather' operations."

Comment 11

Reviewer Comment:
Appendix A, B & C: Showing these codelets is generally a nice idea. However, it would be more helpful, if there was explanation of the details of the template parameters and the function parameters. Can you explain, why __restrict__ is used in dot_product_t, but not for the pointers in matmul_t? I'd assume that at least the memory buffer C must not alias with a or b? Also in matmul_t, line 428, it looks as if access to C and b was strided? Is this behaviour not counter-intuitive performance-wise in the innermost loop? In line 446 it seems a little bit fishy that the template arguments in the call to matmul_t are int literals and do not depend on the template arguments of dshape_tensor_dshape_unroll_t?

Response:
We appreciate the reviewer's detailed code review. We address these specific technical points as follows:
__restrict__ in matmul_t: The reviewer is technically correct that the memory buffer C does not alias with a or b. During our optimization phase, we did test applying the __restrict__ qualifier to all pointers in matmul_t. However, because these pointers reference extremely small, thread-local arrays that are aggressively lowered into registers by the NVCC compiler, adding __restrict__ yielded no measurable performance difference in our profiling. Consequently, we kept the function signature simple and have opted to maintain the code snippet as originally implemented.

Strided access in matmul_t: The strided access (b[T_K * ni + ki]) is mathematically necessitated by Fluidity's underlying Fortran-based Column-Major memory layout. However, because a, b, and C are extremely small, thread-local arrays (typically heavily cached in registers or L1 cache), this strided access does not incur the severe performance penalties associated with global memory strides.

1,3,3 int literals: The dshape_tensor_dshape function inherently models 3D Cartesian spatial gradients interacting with a 3x3 physical tensor. We used 1,3,3 literals here intentionally to allow the compiler to aggressively unroll this specific 3D physical operation without generating generic overhead. We have added detailed comments to the Appendices explaining the template

parameters (e.g., T_dim, T_loc, T_ngi) and the rationale behind the implementations.

Comment 12

Reviewer Comment:
Minor line-by-line comments (Lines 6, 11, 32, 64, 76, 92/93, 98, 107, 112, 166, 277, 279, 334, 346, References).

Response: We thank the reviewer for these highly constructive corrections. We have addressed all of them in the revised manuscript:

Lines 6, 45, & 107: We have corrected the formulation to consistently state that both matrix assembly and the linear solver are major contributors. Specifically, we rephrased Line 107 to accurately quantify that PETSc occupied roughly 37% of the runtime in the CPU baseline (referencing the new Table 10) and becomes the absolute bottleneck after GPU acceleration, resolving the previous contradiction.
Line 32: Rephrased to "can be formulated to satisfy discrete conservation properties," acknowledging that not all FE discretizations are inherently mass-conserving.
Line 64: Elaborated the challenges as "primarily due to indirect memory addressing and low arithmetic intensity."
Line 76: Added an explicit example: "(e.g., evaluating local mass or stiffness matrices via numerical integration)."
Line 166: In the revised manuscript, we have completely rewritten this paragraph (in Section 3.1) to explicitly state: "...thread block sizes of 128 and 256 were both utilized, with the specific parameter for each kernel determined empirically via NVIDIA Nsight Compute."
Line 279: Changed "computes" to "looks up the global index."
Line 334: Clarified "scalar data structure" as "arrays storing single-component variables per node."
Terminology & Formatting: Unified "nonlinear," corrected "MPI," fixed "shows," corrected "In our GPU," changed "function template optimization" to "optimization via the use of template functions," and cleaned the DOI resolver prefix in the BibTeX file.