One Codebase with Good Performance on both CPUs and GPUs, for Column-based Loop Structures

2026-March-05

Pranay Reddy Kommera and David Appelhans, NVIDIA

Agenda

- Overview
- Code Structures and Implementations
- Loop Restructuring
- Performance Comparison
- Conclusion

Motivation

- You want to maintain CPU performance, while also enabling GPU performance.
- GPUs require 2 orders of magnitude more exposed parallelism.
  - Parallelism should be contained in one “kernel” (not many threads launching low parallelism kernels)
  - Kernels should not have a deep call stack.
- This means for GPU performance you often want to push some outer parallelism down the call stack to be in the same place as inner parallelism.
- Would like to not change CPU performance. Is there a way to achieve both?

Context

- Many atmospheric models employ column-based physics implementations
  - Column-based design:
    - Outer loops iterate over horizontal grid points
    - Inner loops process vertical columns
  - Physics Schemes (e.g., microphysics, CVMix) operate independently on individual vertical columns

- **Column-based** implementation is effective for **CPU**-based processing due to cache efficiency. However not well-suited for GPU implementation as it limits parallelism.

- **Plane-based** implementation (horizontal grid points and vertical columns are processed together) is good for the **GPU** architecture but degrades CPU efficiency.

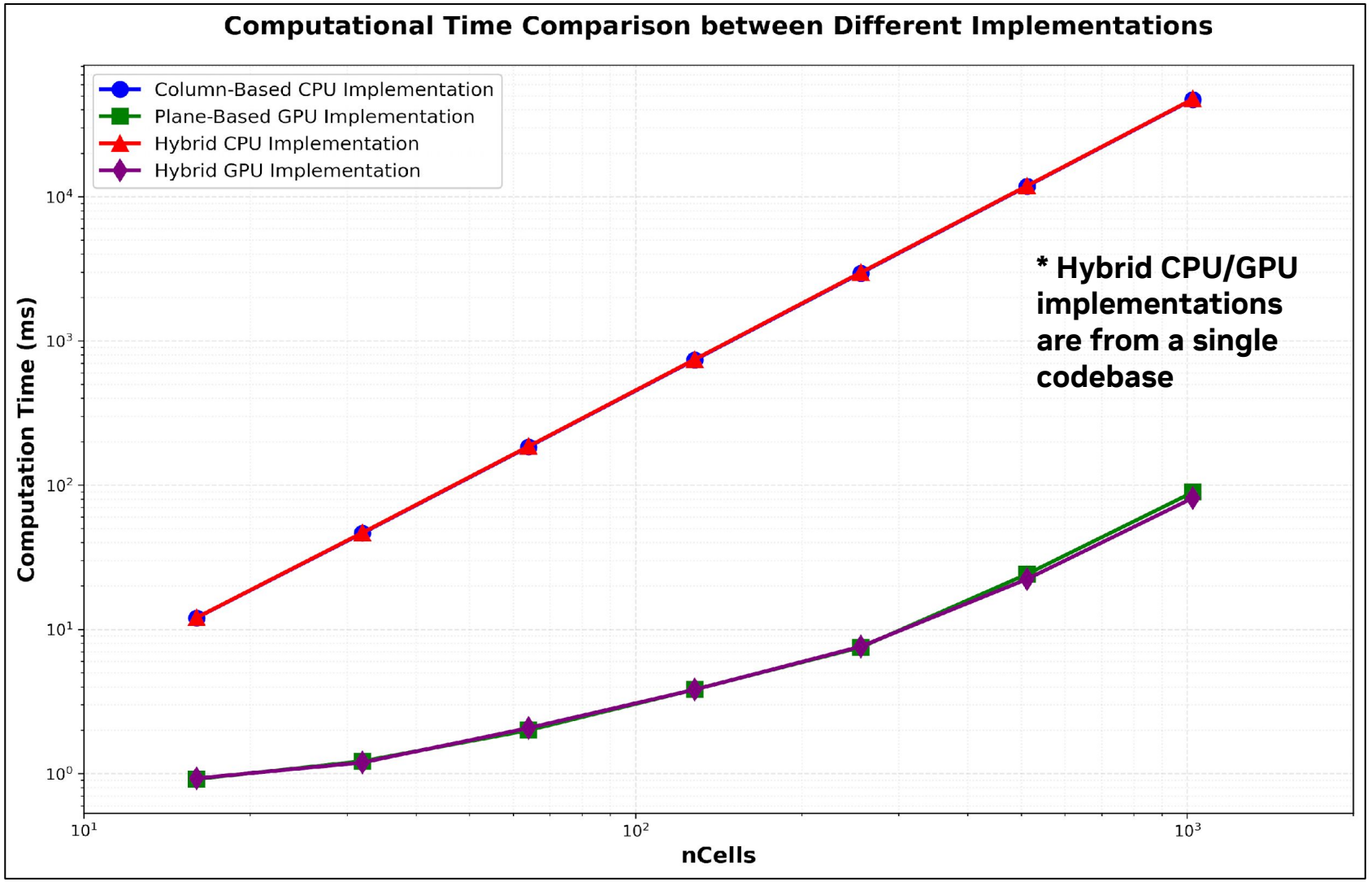
- Different loop structures are needed to achieve optimal performance on the CPU and GPU hardware

- Proposed Solution
  - **Hybrid loop** implementation with ***stride-based*** approach to dynamically switch between the appropriate loop configurations
  - Single unified code base that maintains high performance across platforms

Context

Goal

- Use a **single codebase** to achieve optimal performance on both CPU and GPU architectures.



Code Implementation

Parameters and Hardware Configuration

- Standalone C/C++ code
  - Resembling TEMPO Microphysics layout
  - Ported to GPUs using OpenACC directive-based programming model

- Parameters
  - Total grid points
    - Depends on the resolution
    - Varies from tens to millions
    - **NI = NJ = nCells**, per dimension is used in this study
      - Total grid points = nCells * nCells
  - Total vertical columns
    - Generally, < 200
    - **NK = 50** in this study

- Hardware Configuration
  - EOS Cluster
    - H100 GPU
      - 80GB HBM3
      - 3.2 TBs Memory Bandwidth
    - Dual-socket Intel Sapphire Rapids
      - 56 Core per socket

Code Structure

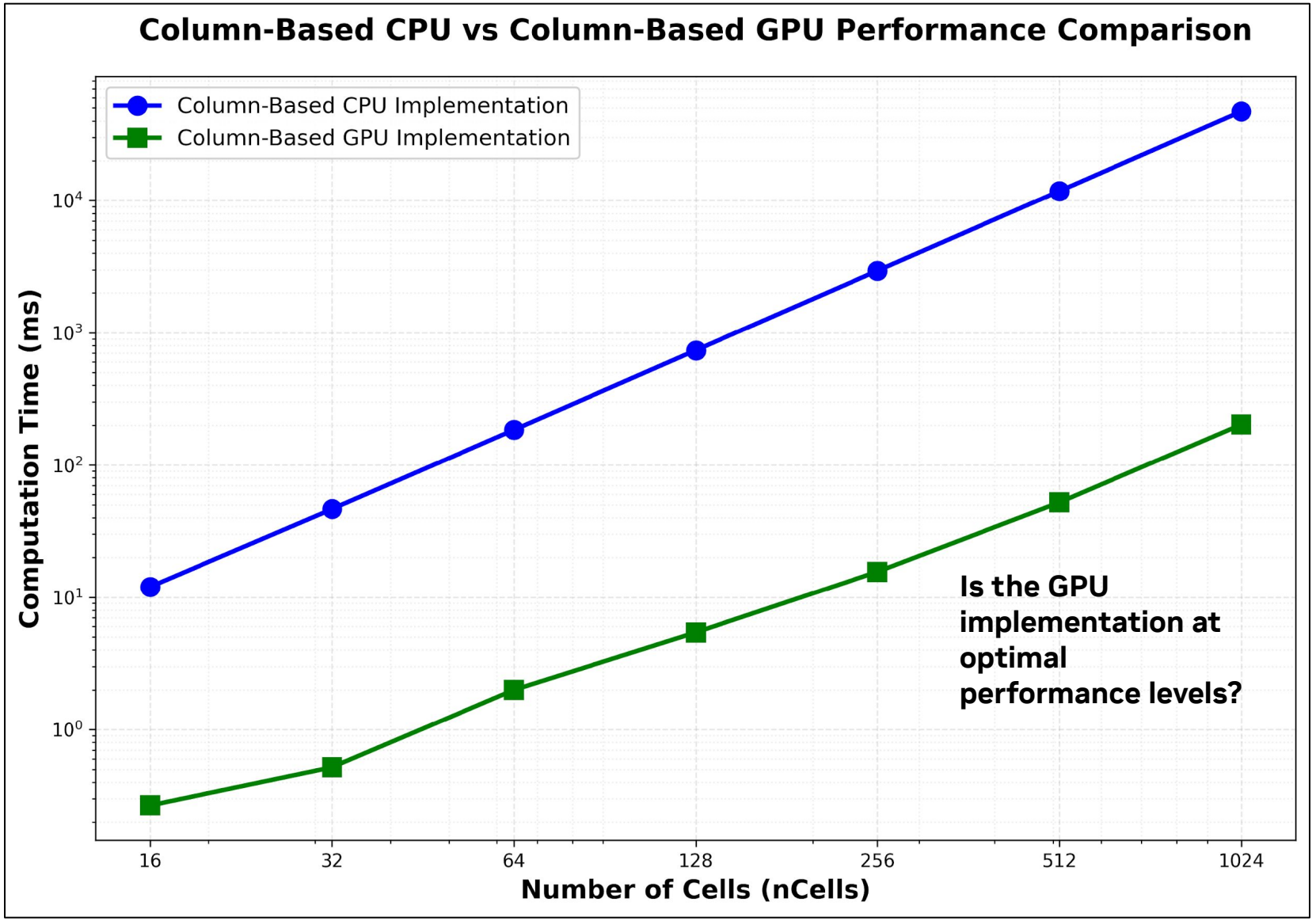
Column-Based Implementation

- Column-Based Implementation
  - Outer loops iterate over the horizontal grid points - inner loops over vertical columns
  - Extracts k-slice per each horizontal dimension
  - Process one column at a time
  - **Optimal cache efficiency** on **CPUs**
  - **Limited GPU parallelism**

```
// Allocate local 1D arrays (size NK)
local_vel_x = new float [NK]

// Outer loops: i, j
!$acc parallel loop gang collapse(2) private(...)
for i = 0 to NI-1:
    for j = 0 to NJ-1:

        //Step 1: Extract 1D arrays based on (i, j)
        !$acc loop vector
        for k = 0 to NK-1:
            [Code]

        //Step 2: Computations
        !$acc loop vector
        for k = 0 to NK-1:
            [Code]

        //Step 3: Call compute function  (k-loops inside)
        compute_on_k_dimensionlocal_arrays, mod_arrays)

        //Step 4: Copy results back
        !$acc loop vector
        for k = 0 to NK-1:
            [Code]
```

Performance Results

Column-Based Implementation

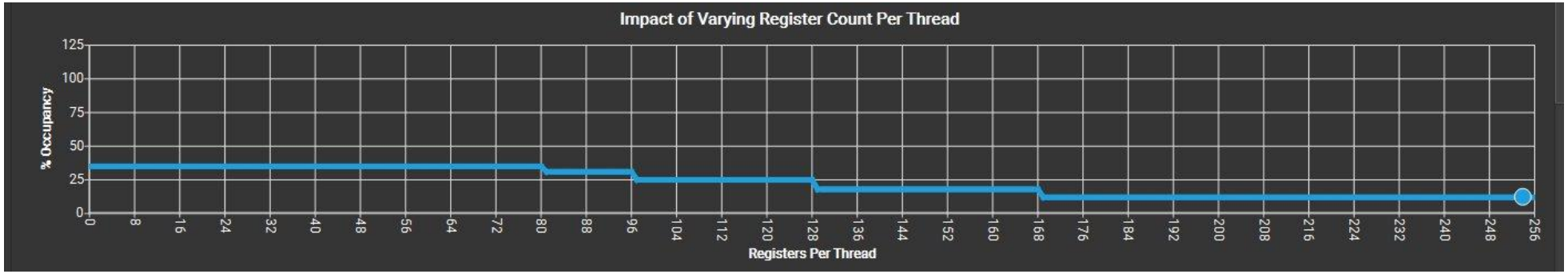


Performance Analysis

Column-Based Implementation

- Is the GPU implementation at optimal performance levels?
  - Are we effectively utilizing the GPU resources?

- GPU Resources
  - Using maximum threads per SM (2048 threads) □ **hide latency and improve occupancy**
  - Occupancy limitations
    - Register usage per thread (total 65K per SM)
    - Shared memory usage per block



Register usage per thread in column-based GPU implementation

- What are we losing on the GPU?
  - **Using too many registers (256 registers) □ less threads in flight (256 threads in flight) □ lower occupancy (12.5%)**

- What can be done to improve GPU efficiency?
  - **Plane-based Implementation**

Code Structure

Plane-Based Implementation

- Plane-Based Implementation
  - Loops over all the horizontal grid points and the vertical columns
  - Process entire 3D domain at a time
  - Increased memory footprint
  - May result in **cache misses on CPUs**
  - **Optimal GPU parallelism**

```
// Allocate local 3D arrays (size NI x NJ x NK)
local_vel_x = allocate_3d_array(NI, NJ, NK)


// Step 1: Pre-fill 3D arrays
!$acc parallel loop collapse(3)
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        for k = 0 to NK-1:
            [Code]


// Step 2: Computations
!$acc parallel loop collapse(3)
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        for k = 0 to NK-1:
            [Code]


// Step 3: Call compute function (i,j,k loops inside)
compute_on_k_dimension(local_arrays, mod_arrays)

// Step 4: Copy results back
!$acc parallel loop collapse(3)
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        for k = 0 to NK-1:
            [Code]
```

Code Implementation

Column-Based & Plane-Based Implementations

```
// Allocate local 1D arrays (size NK)
local_vel_x = new float [NK]

// Outer loops: i, j
!$acc parallel loop gang collapse(2) private(...)
for i = 0 to NI-1:
    for j = 0 to NJ-1:

        //Step 1: Extract 1D arrays based on (i, j)
        !$acc loop vector
        for k = 0 to NK-1:
            [Code]

        //Step 2: Computations
        !$acc loop vector
        for k = 0 to NK-1:
            [Code]

        //Step 3: Call compute function  (k-loops inside)
        compute_on_k_dimensionlocal_arrays, mod_arrays)

        //Step 4: Copy results back
        !$acc loop vector
        for k = 0 to NK-1:
            [Code]
```

Column-Based Implementation

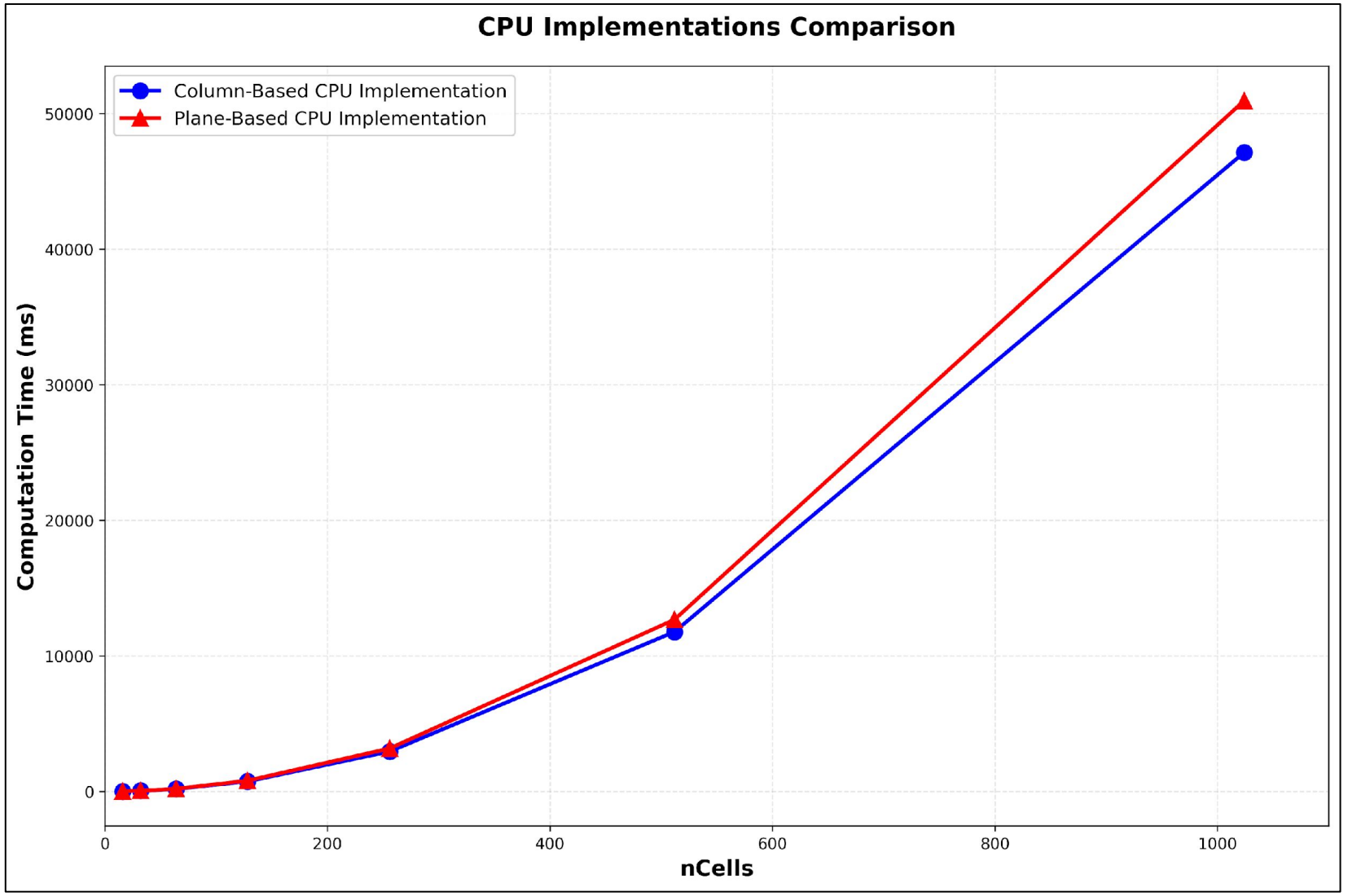
```
// Allocate local 3D arrays (size NI x NJ x NK)
local_vel_x = allocate_3d_array(NI, NJ, NK)


// Step 1: Pre-fill 3D arrays
!$acc parallel loop collapse(3)
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        for k = 0 to NK-1:
            [Code]


// Step 2: Computations
!$acc parallel loop collapse(3)
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        for k = 0 to NK-1:
            [Code]


// Step 3: Call compute function(i,j,k loops inside)
compute_on_k_dimension(local_arrays, mod_arrays)

// Step 4: Copy results back
!$acc parallel loop collapse(3)
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        for k = 0 to NK-1:
            [Code]
```

Plane-Based Implementation

Performance Results

Column-Based & Plane-Based CPU Implementations

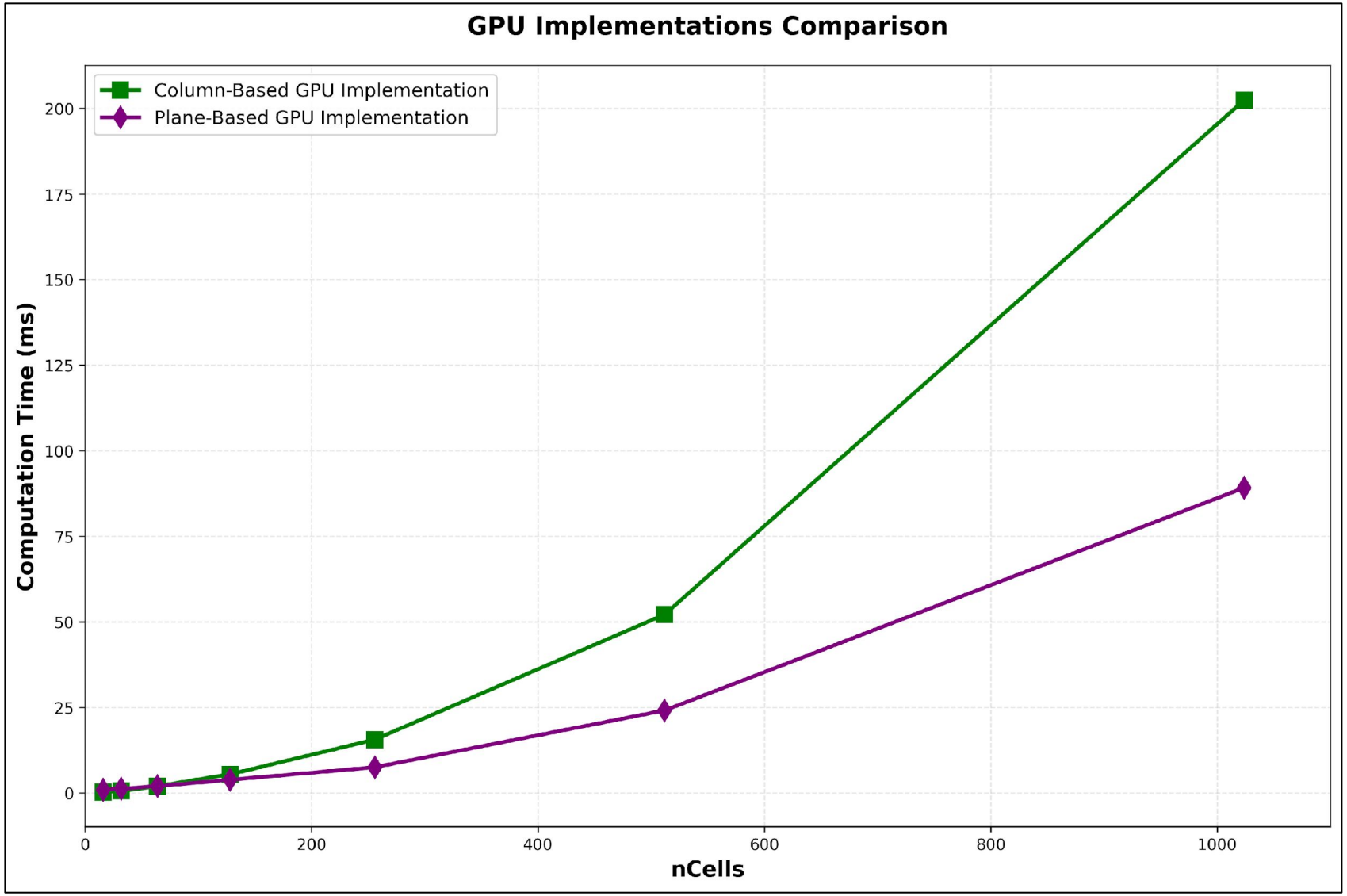
- Plane-based CPU implementation has **8% performance penalty** compared to column-based CPU implementation.



Performance Results

Column-Based & Plane-Based GPU Implementations

- Plane-based GPU implementation has **~2x performance improvement** compared to column-based GPU implementation for higher ‘nCells’.



Loop Restructuring

Why?

	Column-based Implementation	Plane-based Implementation
Implementation	Process one column at a time	Process entire 3D domain simultaneously
CPU Execution	Cache efficiency	Limited Cache benefits
GPU Execution	Limited Parallelism	Full 3D parallelism
Optimal Performance	CPU Implementation	GPU Implementation

- Neither implementations achieve optimal performance on both the CPU and GPU hardware
  - Two independent code versions required to achieve optimal performance

- Drawbacks of two independent code versions
  - Code duplication increasing maintenance burden
  - Development overhead
  - Version control complexity

- Solution
  - **Hybrid Strided Implementation**
    - Single code base that can dynamically switch between column-based and plane-based implementations based on the architecture

Code Implementation

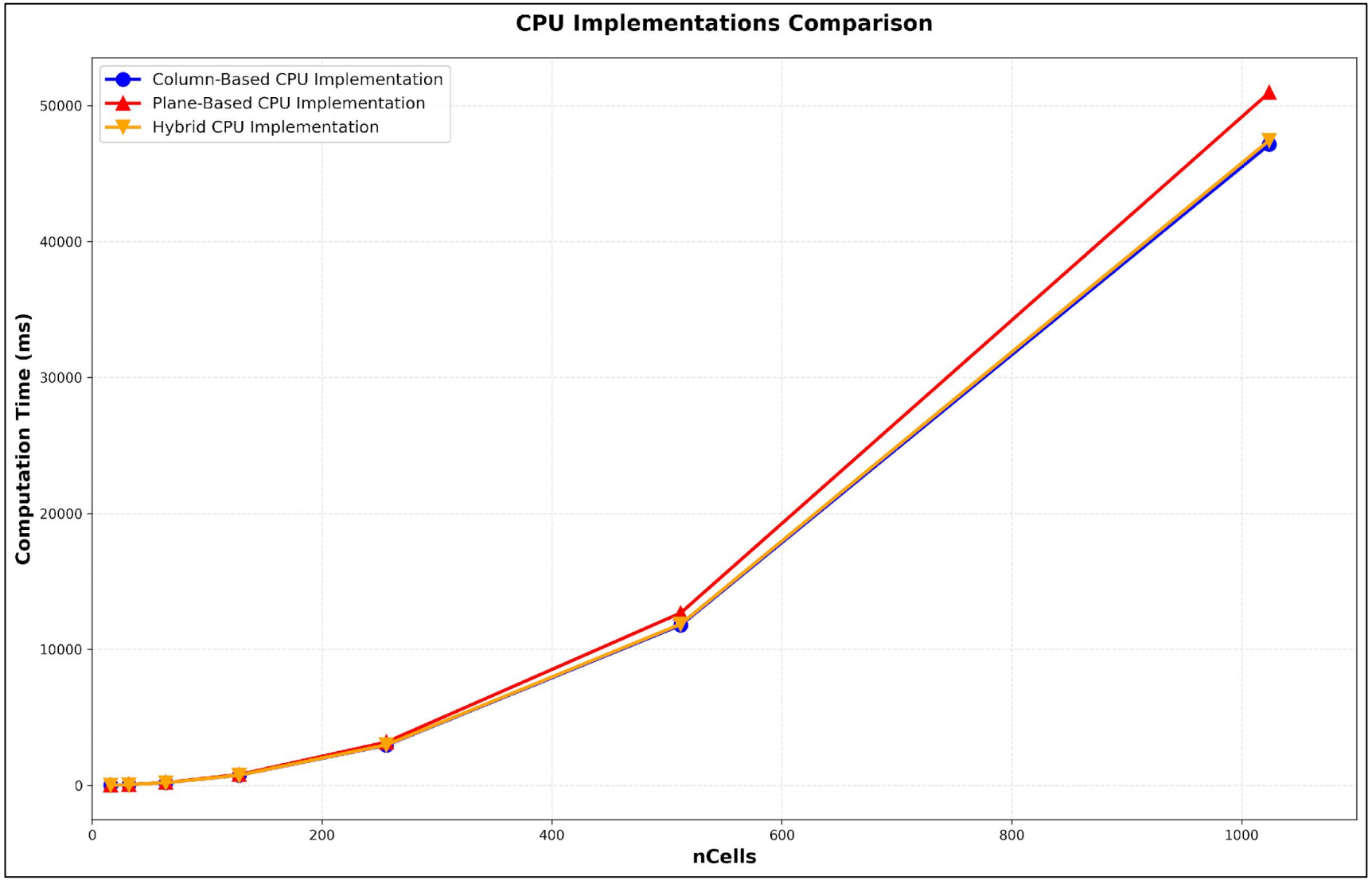
Hybrid Strided Implementation

- Hybrid Strided Implementation
  - Two sets of loops (over horizontal grid points) are employed
  - The loops iterate over the grid points using ‘stride’ parameter
  - **‘stride =1’** enables **column-based** implementation
  - **‘stride = nCells’** enables **plane-based** implementation

```
// Allocate local arrays (size stride x stride x NK)
local_vel_x = allocate_3d_array(stride, stride, NK)

// Outer loops: i_out, j_out (strided, CPU sequential)
for i_out = 0 to NI-1 step stride:
    for j_out = 0 to NJ-1 step stride:

        //Step 1: Inner loops: i_in, j_in
        !$acc parallel loop collapse(3)
        for i_in = 0 to stride-1:
            for j_in = 0 to stride-1:
                for k = 0 to NK-1:
                    i_global= i_out + i_in; j_global= j_out + j_in
                    [Code]

        //Step 2: Computations
        !$acc parallel loop collapse(3)
        for i_in = 0 to stride-1:
            for j_in = 0 to stride-1:
                for k = 0 to NK-1:
                    i_global= i_out + i_in; j_global= j_out + j_in
                    [Code]

        //Step 3: Call compute function (i_in,j_in,k loops inside)
        compute_on_k_dimensionlocal_arrays, mod_arrays)

        //Step 4: Copy results back
        for i_in = 0 to stride-1:
            for j_in = 0 to stride-1:
                for k = 0 to NK-1:
                    i_global= i_out + i_in; j_global= j_out + j_in
                    [Code]
```

Performance Results

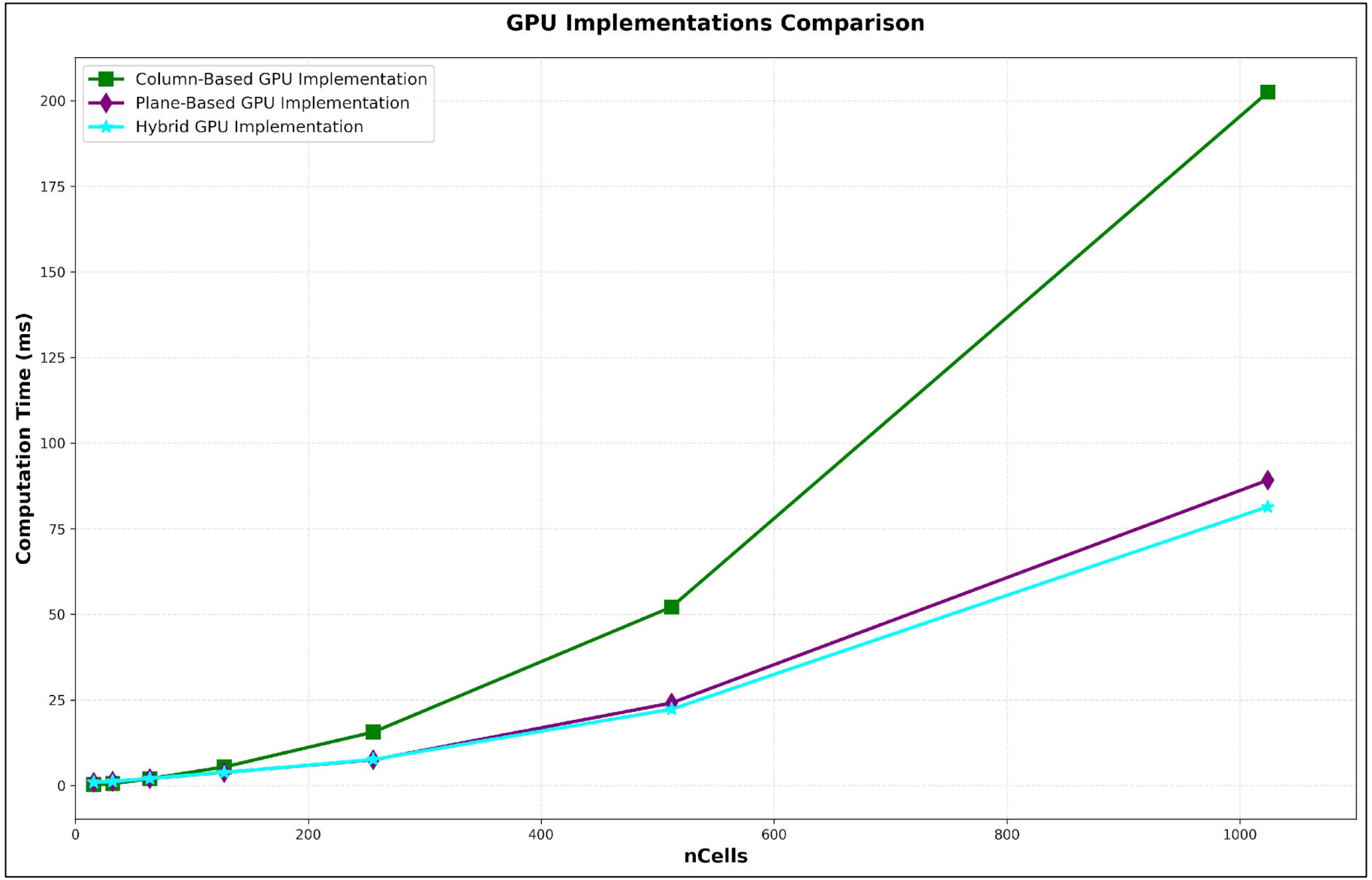
Hybrid CPU Implementation



- Hybrid CPU implementation has less than 1% performance penalty compared to column-based CPU implementation.

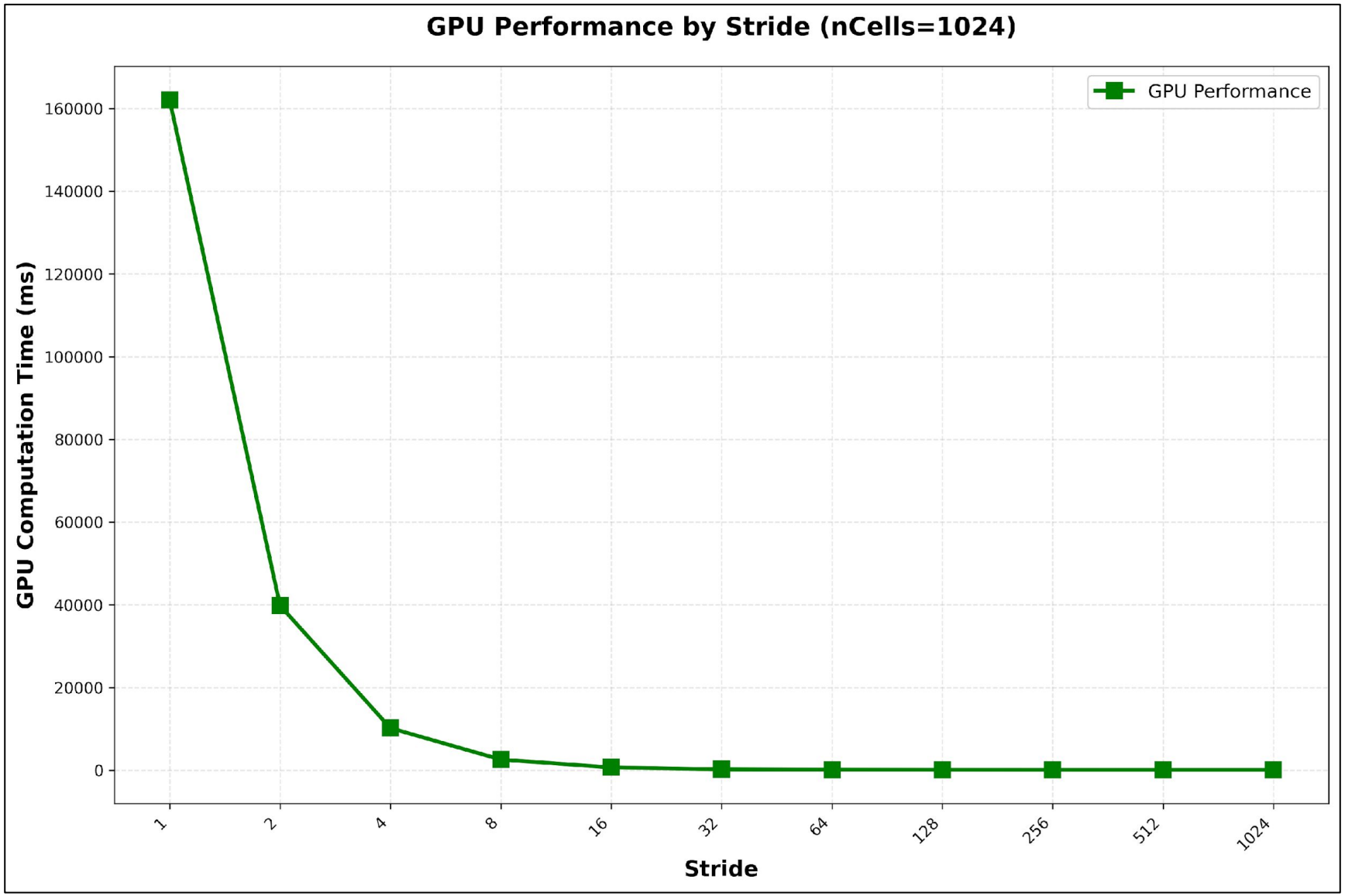
Performance Results

Hybrid GPU Implementation



Performance Results

Hybrid GPU Implementation



Performance Results

Hybrid Implementation

nCells	Speedup - Hybrid CPU Implementation	Speedup - Hybrid GPU Implementation
16	1.00x	0.29x
32	0.99x	0.42x
64	0.99x	1.00x
128	0.99x	1.42x
256	0.99x	2.08x
512	0.99x	2.16x
1024	0.99x	2.27x

*** Speedup of Hybrid CPU implementation (stride = 1) is with respect to the column-based CPU implementation**
*** Speedup of Hybrid GPU implementation (stride = nCells) is with respect to the column-based GPU implementation**

- Advantages of hybrid implementation
  - Optimal performance on both CPU and GPU implementation
  - Single Codebase
  - Runtime Adaption

Conclusion

- Unified Codebase
  - The hybrid strided loop structure, implemented through a stride parameter, eliminates code duplication while maintaining optimal performance across both CPU and GPU architectures
- Performance Retention
  - The hybrid implementation achieves 99% of baseline CPU performance (0.99x) while delivering improved GPU speedups (~2x) compared to the baseline GPU implementation
- Practical GPU Acceleration
  - Demonstrated that column-based physics implementations can be efficiently ported to GPUs without compromising CPU performance or requiring separate codebases
- Impact
  - The hybrid implementation reduces maintenance burden, eliminates version control complexity, and enables seamless portability across CPU and GPU platforms.

Thank You

Performance Results

Hybrid CPU Implementation