Portable High-Performance TEMPO Microphysics — A Unified CPU/GPU Codebase via Hybrid Loop Restructuring

Pranay Reddy Kommera, NVIDIA Corporation | 2026-July-22

Agenda

- **Introduction** — Column-based physics in weather & climate codes
- **Column-Based Implementation** — OpenACC port and performance analysis
- **Plane-Based Implementation** — Restructured loops and OpenACC performance
- **Hybrid Strided Restructuring** — A unified CPU/GPU loop structure
- **Performance Comparison** — Speedups across implementations
- **Conclusion** — Key findings and impact

Column-Based Physics in Weather & Climate Codes

Physics runs column by column

- Radiation, microphysics, PBL, convection and land-surface schemes act on independent vertical columns
- Columns carry no horizontal dependency, so the vertical column is the natural unit of computation

GSL physics suite via CCPP

- Column-based schemes include Thompson microphysics, MYNN-EDMF PBL, RUC land surface, Grell-Freitas convection and RRTMG radiation

Why it matters here

- Column independence exposes abundant parallelism, but the limited work per column challenges GPU efficiency — motivating the implementations that follow

Column-Based Implementation

- Thompson-Eidhammer Microphysics Parameterization for Operations (TEMPO) Microphysics is a column-based implementation
- Outer loops iterate over horizontal grid points; inner loops over vertical columns
- Extracts a k-slice per horizontal dimension
- Process one column at a time = **Limited parallelism**
- **Optimal cache efficiency** on **CPUs**

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

// Outer loops: i, j
for i = 0 to NI-1:
    for j = 0 to NJ-1:

        // Step 1: Extract 1D arrays
        for k = 0 to NK-1:
            [Code]

        // Step 2: Computations
        for k = 0 to NK-1:
            [Code]

        // Step 3: Call compute function
        tempo_run(local_arrays, mod_arrays)

        // Step 4: Copy results back
        for k = 0 to NK-1:
            [Code]
```

Code Implementation — Parameters & Hardware

Parameters

- **Total grid points**
  - Depends on the resolution
  - Varies from tens to millions
  - NI = NJ = nCells per dimension (this study)
  - Total grid points = nCells × nCells
- **Total vertical columns**
  - NK = 59 in this study

Hardware Configuration

- DGX H100 Cluster
- H100 GPU
  - 80 GB HBM3
  - 3.2 TB/s memory bandwidth
- Dual-socket Intel Sapphire Rapids
  - 56 cores per socket

OpenACC Implementation — Column-Based

- Outer loops distributed across blocks / gangs
- Inner loops distributed across threads / vector
- Single large OpenACC kernel
  - Many subroutine calls inside the kernel
  - **Higher usage** of '!$acc routine' clauses
  - Resulting in increased metadata usage and higher number of local variables on the device memory
  - Results in **higher register usage** and **memory-spill** to slower local memory

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

// Outer loops: i, j
!$acc parallel loop gang collapse(2) private(...)
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        // Step 1: Extract 1D arrays
        !$acc loop vector
        for k = 0 to NK-1:
            [Code]
        // Step 2: Computations
        !$acc loop vector
        for k = 0 to NK-1:
            [Code]
        // Step 3: Call compute function
        tempo_run(local_arrays, mod_arrays)
        // Step 4: Copy results back
        !$acc loop vector
        for k = 0 to NK-1:
            [Code]
```

Performance Analysis — Column-Based Implementation

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

- Maximum threads per SM (2048 threads) hide latency and improve occupancy
- Occupancy limited by registers per thread (65K per SM) and shared memory per block
- Maximum threads per SM is achievable with **32 registers per thread** – column-based implementation required **255 registers per thread**

What are we losing on the GPU?

- Too many registers (255 per thread) → **only 256 threads in flight → 12.5% occupancy**
- Lower occupancy = difficult to hide latency
- Alternative: **plane-based implementation**

Plane-Based Implementation

- Loops over all horizontal grid points and vertical columns
- Process entire 3D domain at a time
- Increased memory footprint
- May result in cache misses on CPUs

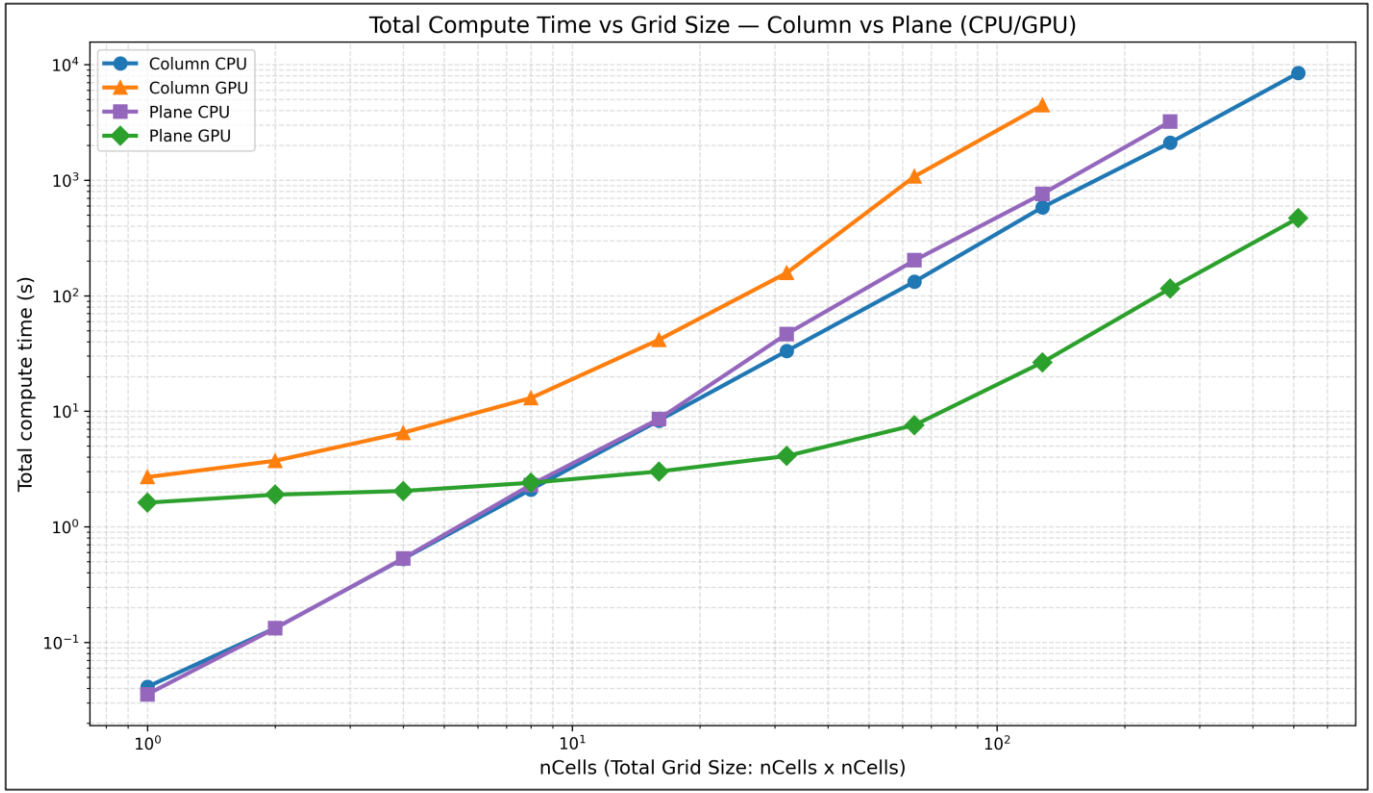
```
// Allocate 3D arrays (NI x NJ x NK)
local_vel_x = allocate_3d_array(NI, NJ, NK)

// Step 1: Pre-fill 3D arrays
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        for k = 0 to NK-1:
            [Code]

// Step 2: Computations
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        for k = 0 to NK-1:
            [Code]

// Step 3: Call compute function
tempo_run(local_arrays, mod_arrays)

// Step 4: Copy results back
for i = 0 to NI-1:
    for j = 0 to NJ-1:
        for k = 0 to NK-1:
            [Code]
```

OpenACC Implementation — Plane-Based

- Collapse all three loops (if possible) to expose higher parallelism
- Reduce register pressure

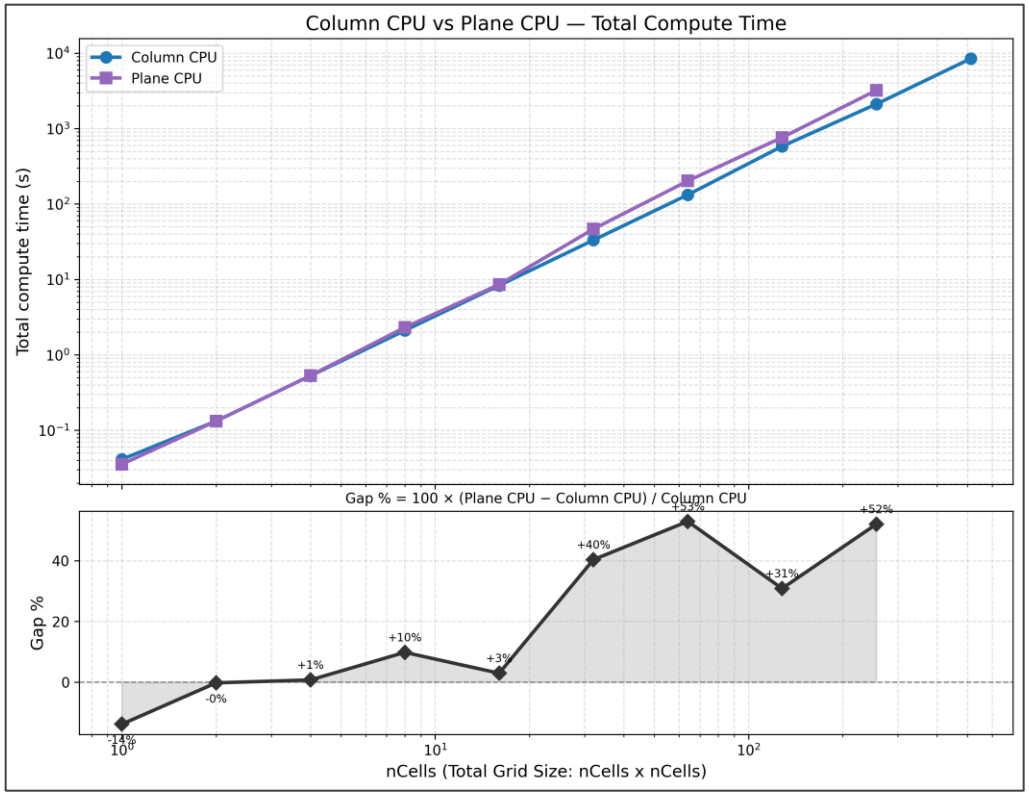
```
// Allocate 3D arrays (NI x NJ x NK)
local_vel_x = allocate_3d_array(NI, NJ, NK)

// 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
tempo_run(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]
```

Performance Numbers — Trade-off across implementations



UIFCW26
A UFS Collaboration Powered by EPIC

Performance Numbers — CPU comparison across implementations



UIFCW26
A UFS Collaboration Powered by EPIC

Loop Restructuring — Motivation

Why?

- Neither implementation achieves optimal performance on both CPU and GPU hardware
- Two independent code versions are required to achieve optimal performance

Drawbacks of two independent code versions

- Code duplication, increasing maintenance burden
- Development overhead and version-control complexity

Solution — Hybrid Strided Implementation

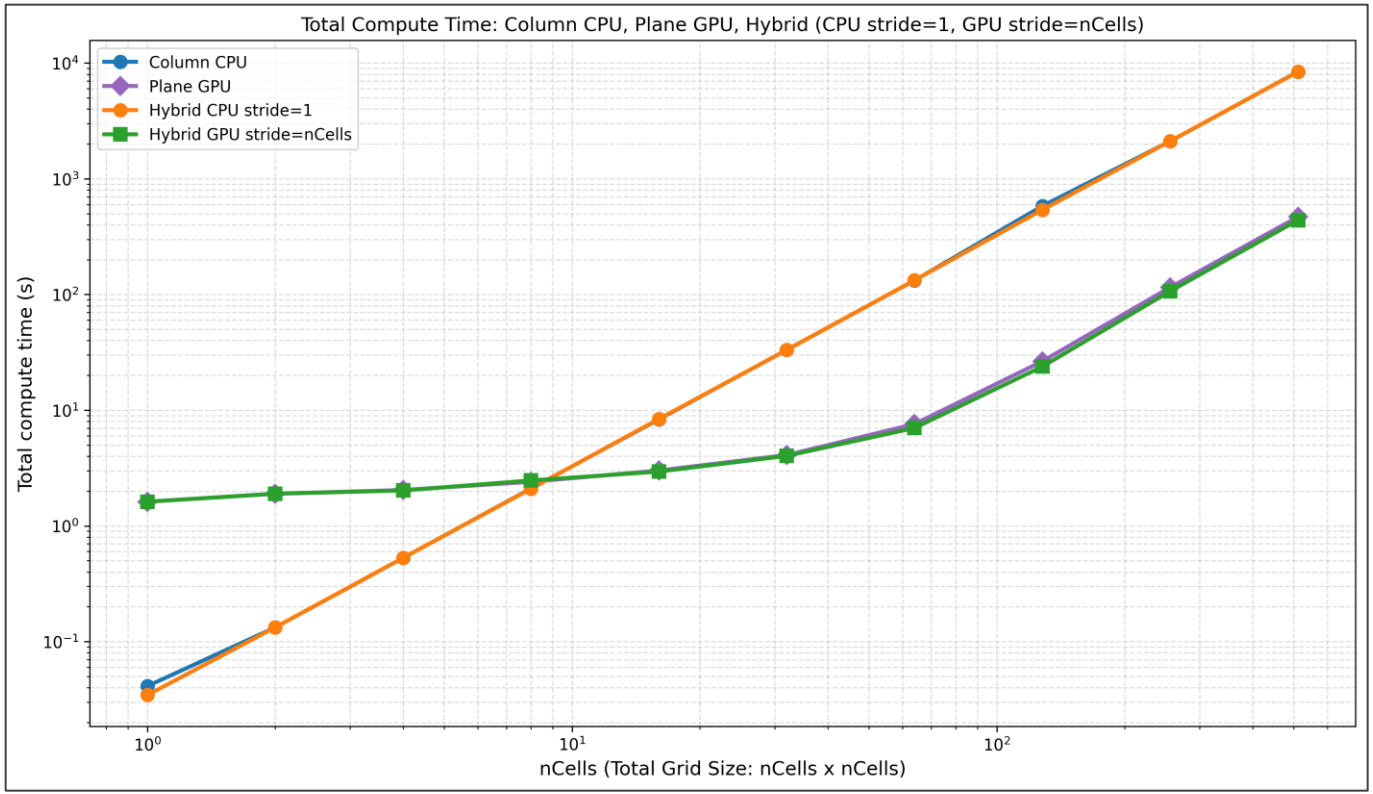
- A single codebase that dynamically switches between column-based and plane-based execution based on the architecture

Code Implementation — Hybrid Strided

- Two sets of loops over horizontal grid points are employed
- Loops iterate over grid points using a **‘stride’** parameter
- **‘stride = 1’** enables the **column-based implementation**
- **‘stride = nCells’** enables the **plane-based implementation**

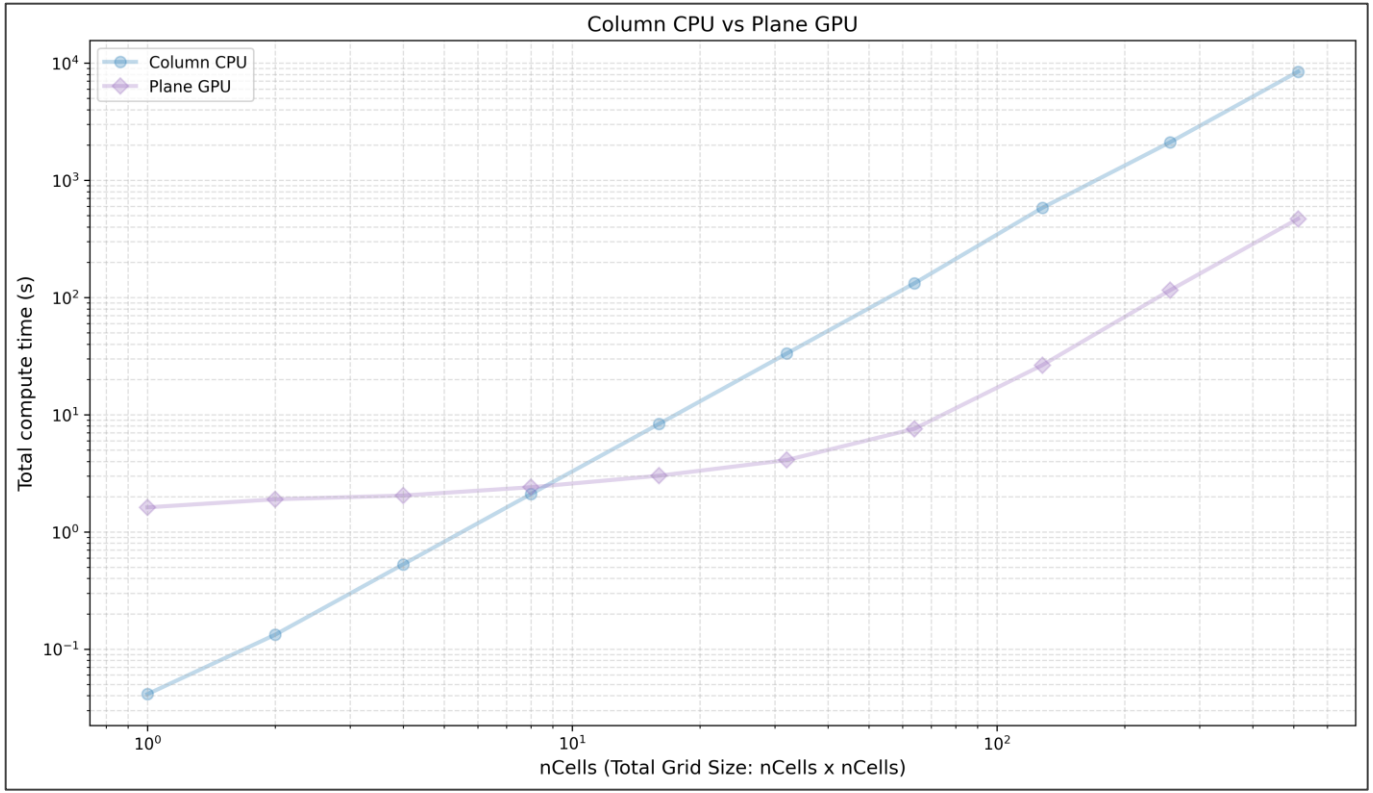
```
// Allocate arrays (stride x stride x NK)
local_vel_x = alloc(stride, stride, NK)

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

    // Step 1: Extract  (!$acc collapse(3))
    for i_in, j_in, k in (stride,stride,NK):
        i_g = i_out+i_in; j_g = j_out+j_in
        [Code]

    // Step 2: Computations (!$acc collapse(3))
    for i_in, j_in, k in (stride,stride,NK):
        [Code]

    // Step 3: Call compute function
    tempo_run(local_arrays, mod_arrays)
    // Step 4: Copy results back

    for i_in, j_in, k in (stride,stride,NK):
        [Code]
```

Performance Comparison — Across implementations



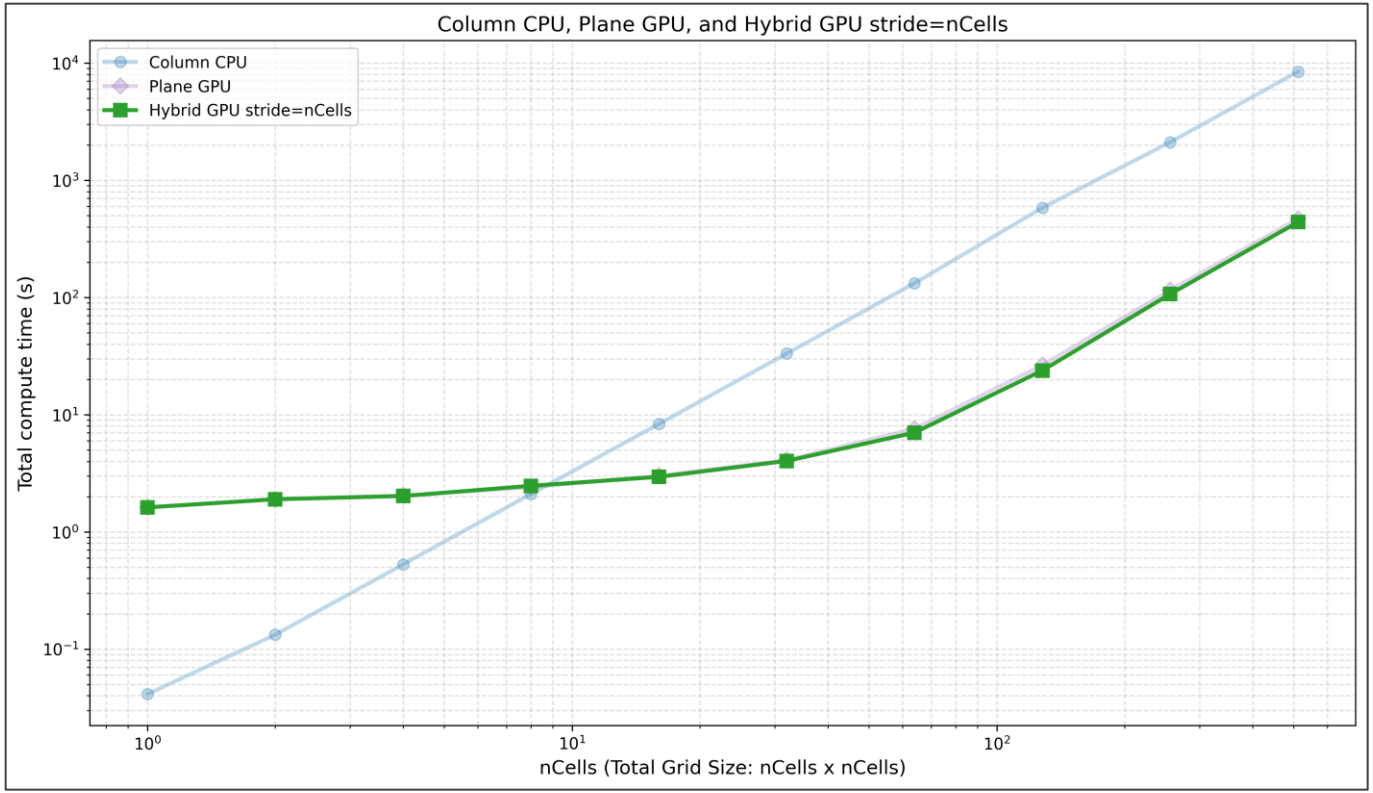
UIFCW26
A UFS Collaboration Powered by EPIC

Performance Comparison — Across implementations cont.



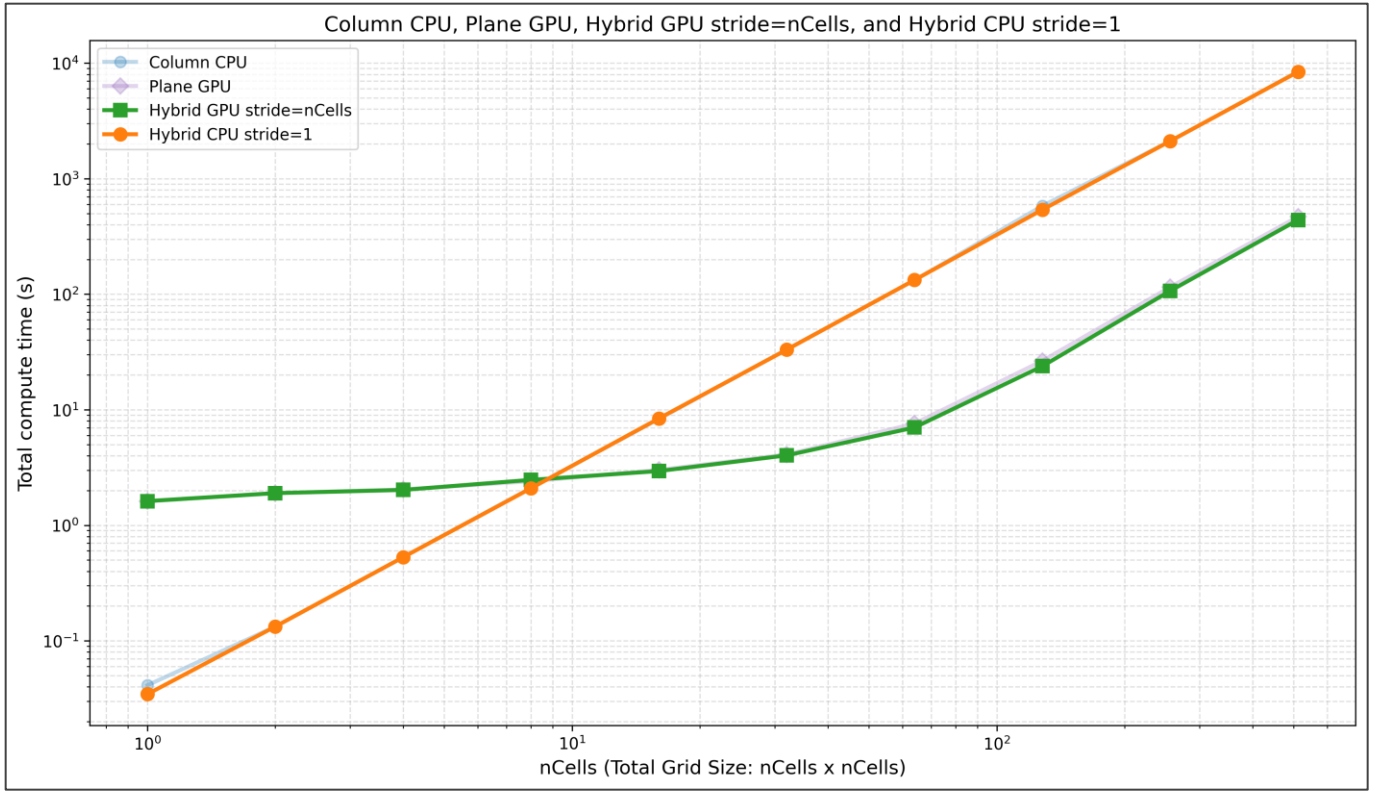
UIFCW26
A UFS Collaboration Powered by EPIC

Performance Comparison — Across implementations cont.



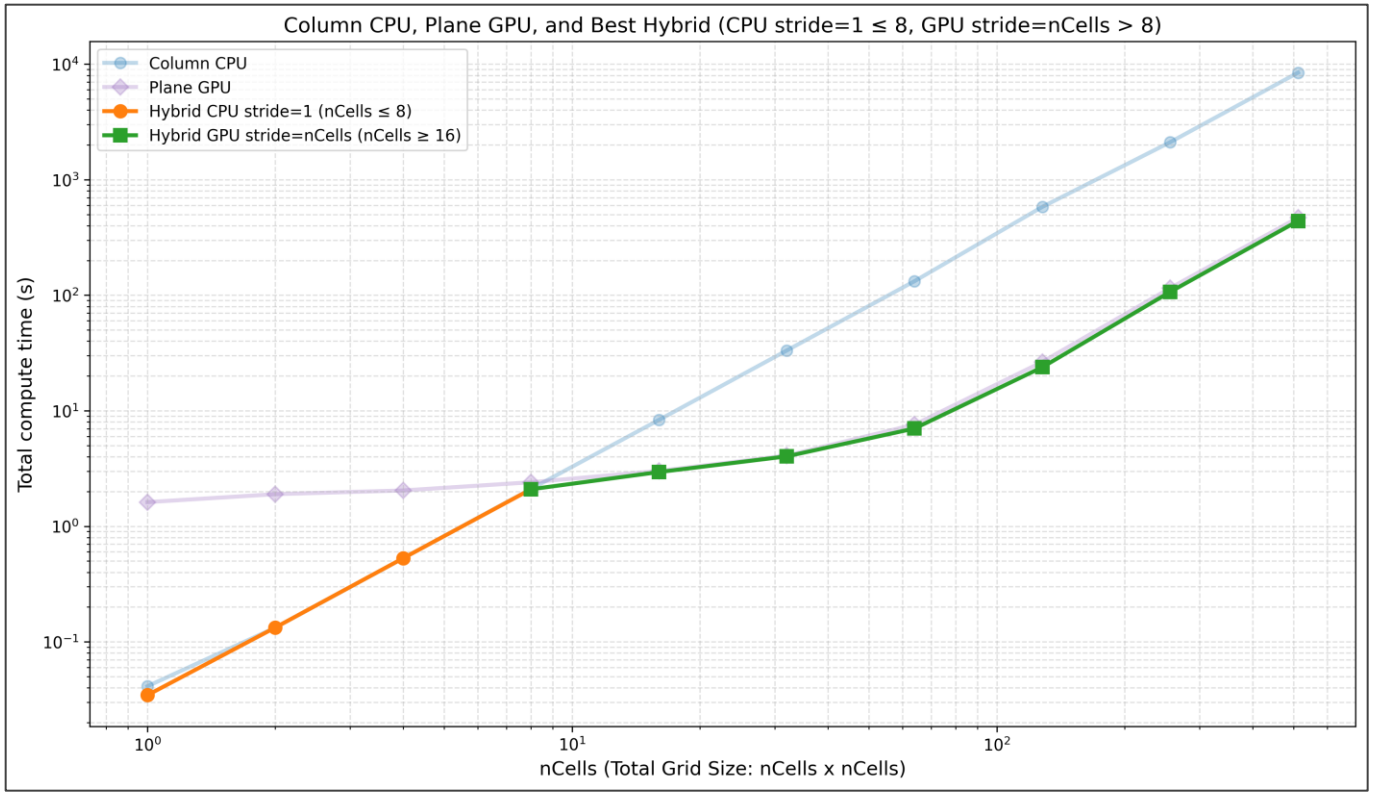
UIFCW26
A UFS Collaboration Powered by EPIC

Performance Comparison — Across implementations cont.



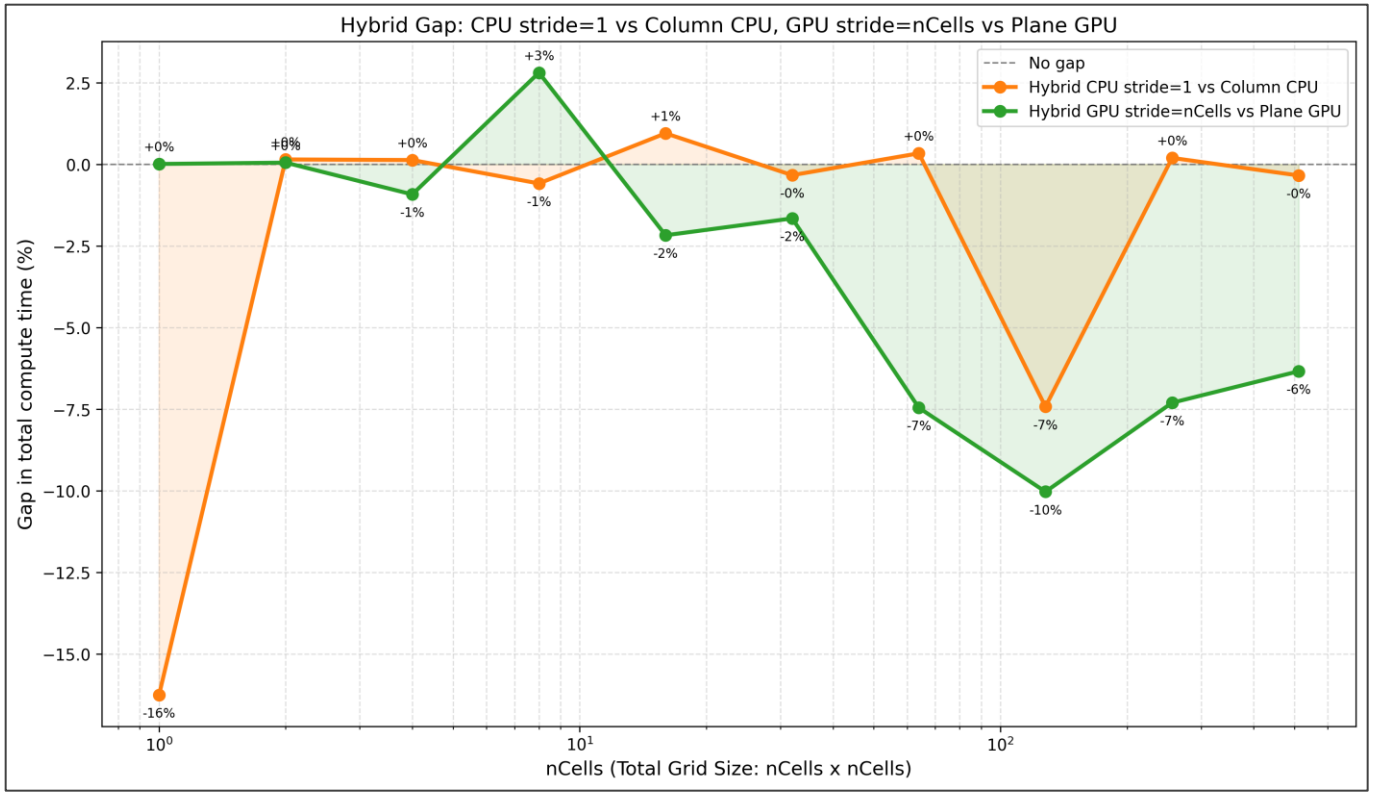
UIFCW26
A UFS Collaboration Powered by EPIC

Performance Comparison — Across implementations



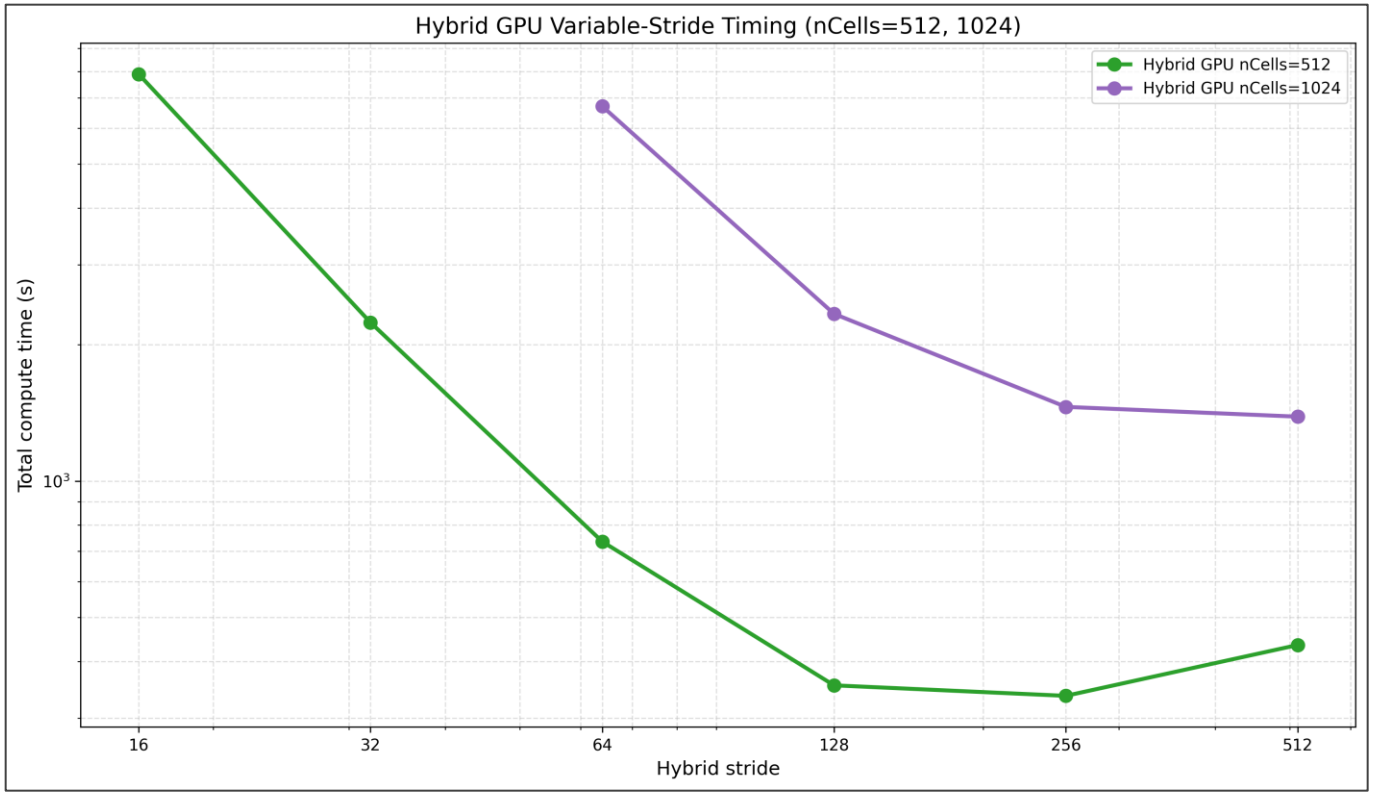
UIFCW26
A UFS Collaboration Powered by EPIC

Performance Comparison — Gap percentage across CPU/GPU implementations



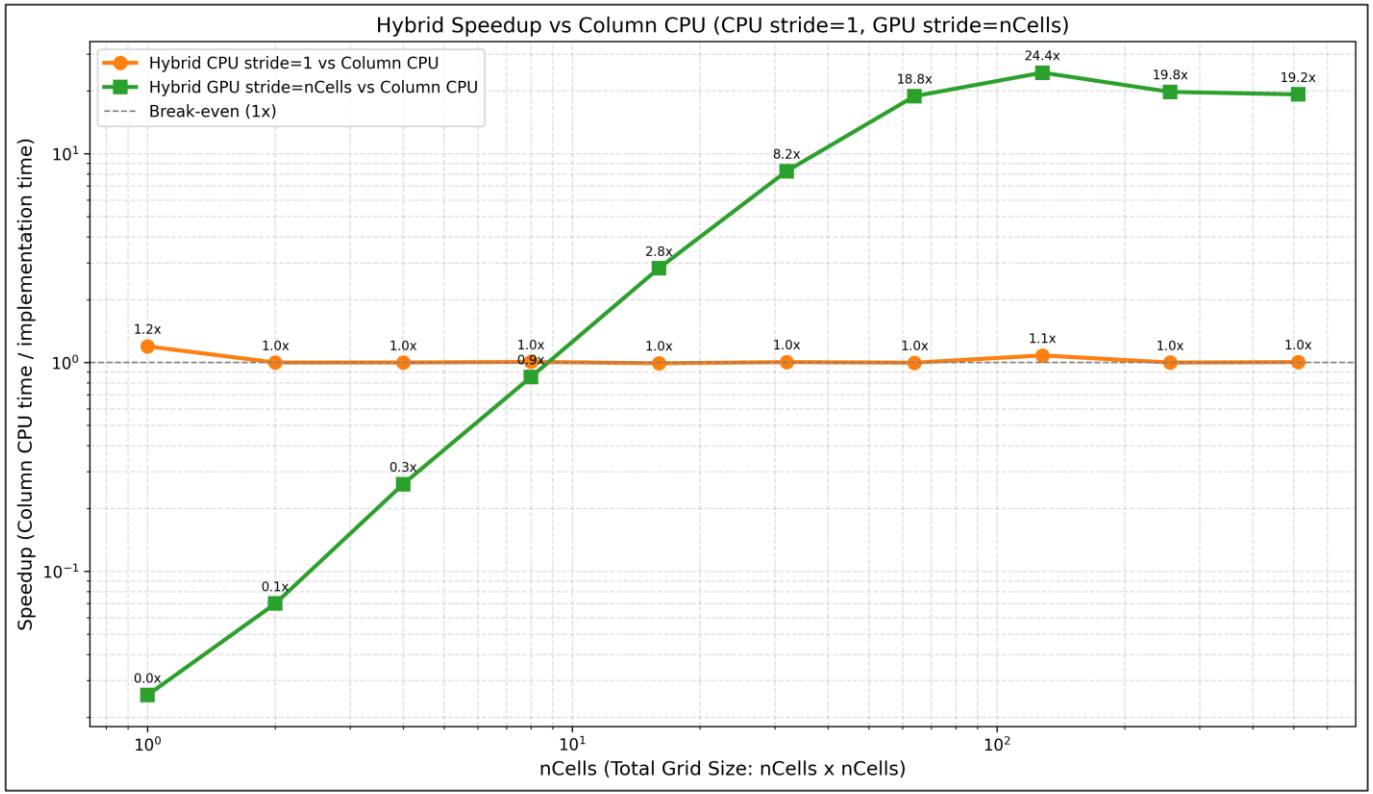
UIFCW26
A UFS Collaboration Powered by EPIC

Performance Comparison — Variable Stride



UIFCW26
A UFS Collaboration Powered by EPIC

Performance Comparison — Speedup



UIFCW26
A UFS Collaboration Powered by EPIC

Conclusion

- **Unified Codebase —** The hybrid strided loop structure, driven by a stride parameter, eliminates code duplication while maintaining optimal performance across CPU and GPU architectures
- **Performance Retention —** Achieves 99% of baseline CPU performance (0.99×) while delivering ~24× GPU speedup
- **Practical GPU Acceleration —** Column-based physics implementations can be efficiently ported to GPUs without compromising CPU performance or requiring separate codebases
- **Impact —** Reduces maintenance burden, eliminates version-control complexity, and enables seamless portability across CPU and GPU platforms

Thank You

Loop Restructuring — Column-Based vs Plane-Based

	Column-Based	Plane-Based
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
Single-Codebase Solution	**Hybrid Strided Implementation**	