# The Multi-component Flow Code (MFC)

*Ben Wilfong[a], Daniel J. Vickers[a], Spencer Bryngelson[b]*

*[a] Georgia Institute of Technology*

*[b] University of Texas, Austin*

SCC26 MFC Webinar
August 26th, 2026

GitHub

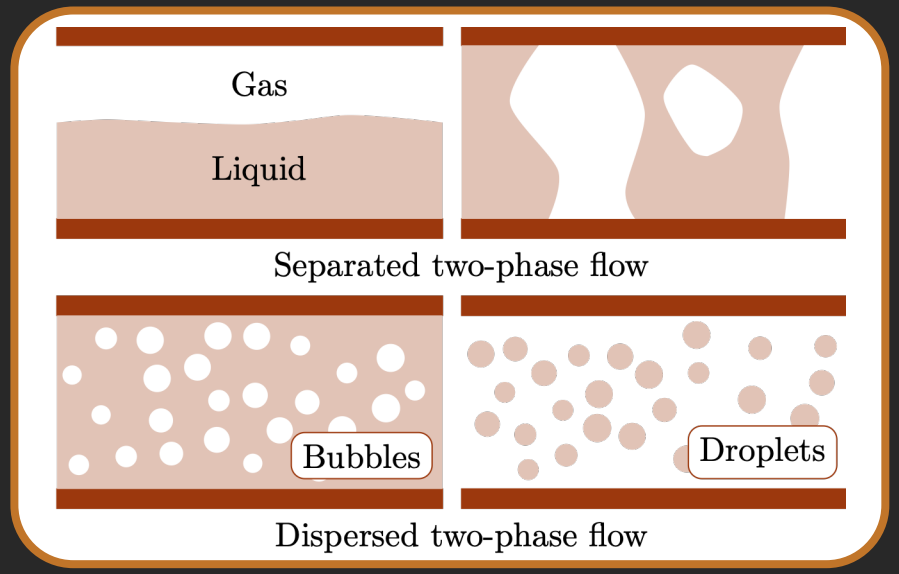
WEBSITE

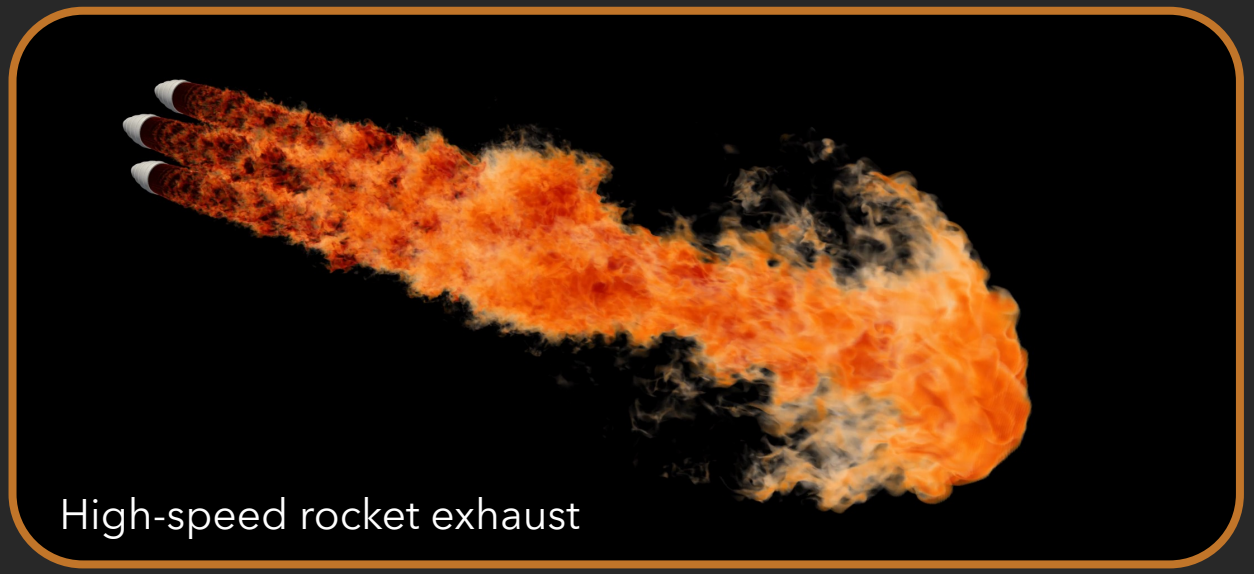
MFC

CPC Computational Physics Group

# Applications of multiphase and compressible flows

- Medical – burst-wave lithotripsy for kidney stone removal
- Medical – design of artificial heart valves
- Engineering – Atomization of liquid droplets
- Engineering – Bubble cavitation





High-speed rocket exhaust

Cavitation

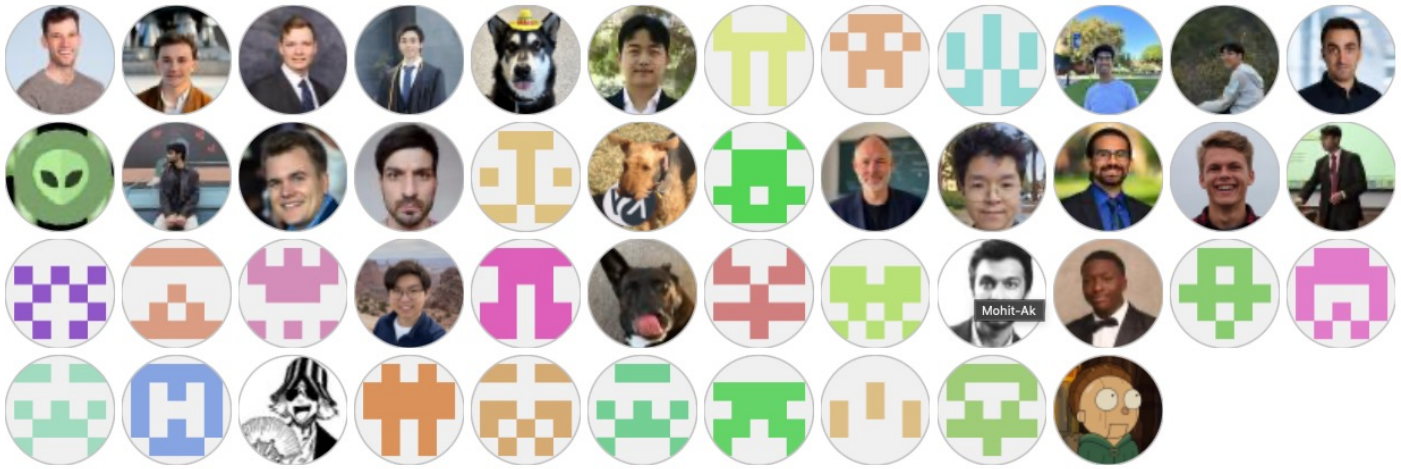
# A brief history of MFC – Cal Tech

- Late 2000's: an unnamed solver is developed by Eric Johnsen
- Early-mid 2010s: Vedran Coralic gives MFC its name after a significant rewrite
- Late 2010s: Spencer Bryngelson and Kevin Schmidmayer add 5- and 6-equation diffuse interface models
- 2020:  MFC 3.0 is open sourced

# A brief history of MFC – Georgia Tech

- 2022: MFC 4.0 is released with OpenACC offloading for NVIDIA GPUs
- 2024: MFC is ported to AMD GPUs to run on Frontier
- 2025: MFC's information geometric regularization implementation is used to perform a simulation with 1 quadrillion degrees of freedom on Frontier
- 2026: MFC 5.0 is published
- 2026: Support for AMDFlang is improved via an OpenACC/OpenMP abstraction layer

# MFC's contributors




Georgia Tech

Caltech

BROWN

NVIDIA

AMD

Hewlett Packard Enterprise

CRAY

UF | UNIVERSITY of FLORIDA

# MFC's core principles

Performance

Portability

Quality

- What does this mean?
  - Any changes you make to MFC *must* maintain application performance, portability, and quality
  - The application developers have the final say in whether your changes meet these criteria
  - Feel free to ask for feedback before making changes

- Limit changes to src/ to +/- 5000 lines

- Introduction
- **Model and Numerical Method**
- Code Design
- Continuous Integration
- Performance Characteristics
- Closing remarks

# Model and numerics

- Compressible and Multiphase
- **Diffuse Interface Method:**
  - Allaire 5-eqn model
  - Saurel 6-eqn model
- **Finite Volume:** WENO3/5/7
  - Shock Capturing
- **Riemann Solve:** HLL/HLLC
- **Time Stepping:** Explicit RK
- **Additional Physics:**
  - Viscosity, surface tension, sub-grid bubbles, phase change, immersed boundaries, and more…

$$\frac{\partial \alpha_i \rho_i}{\partial t} + \nabla \cdot (\alpha_i \rho_i \boldsymbol{u} + \boldsymbol{F}_{\alpha_i \rho_i}) = S_{\alpha_i \rho_i},$$

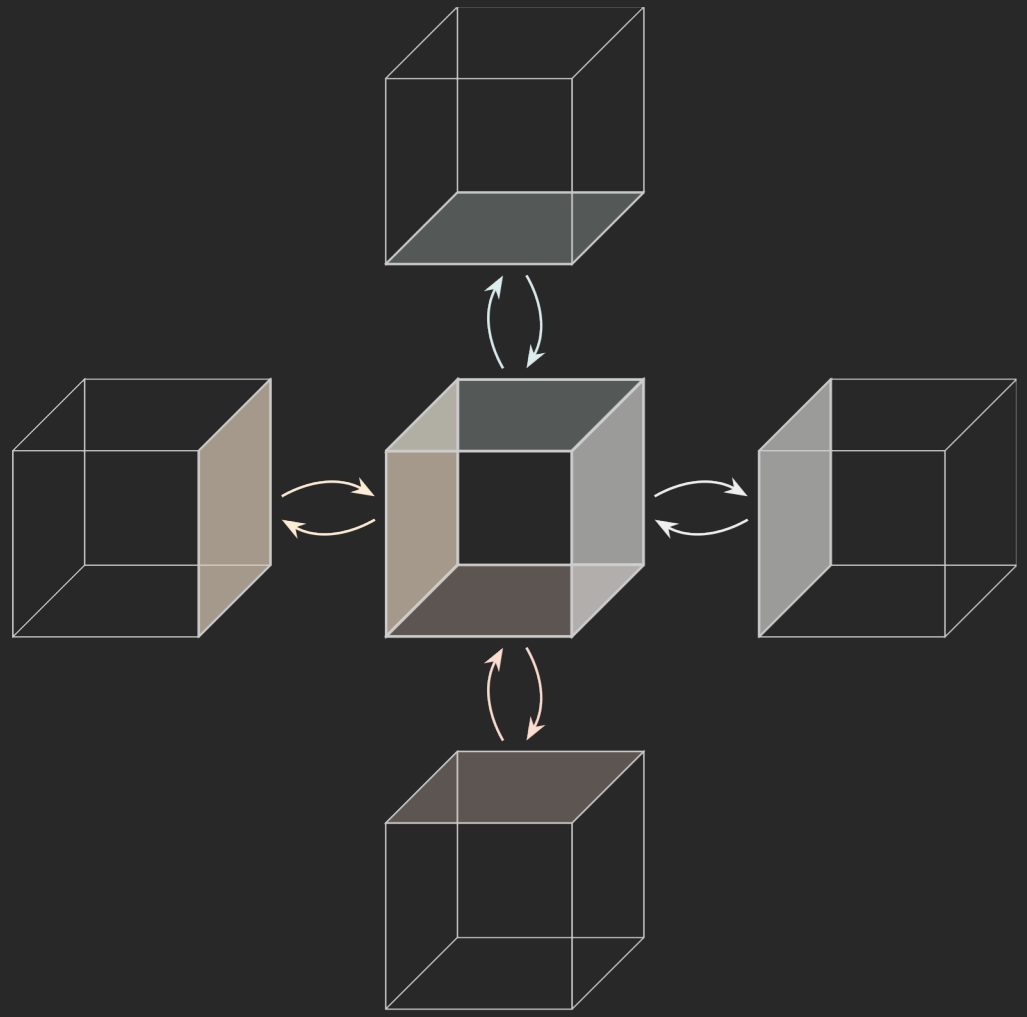
$$\frac{\partial \rho \boldsymbol{u}}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u} + p\boldsymbol{I} + \boldsymbol{F}_{\rho \boldsymbol{u}}) = \boldsymbol{S}_{\rho \boldsymbol{u}},$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + p)\boldsymbol{u} + \boldsymbol{F}_{\rho E}] = S_{\rho E},$$

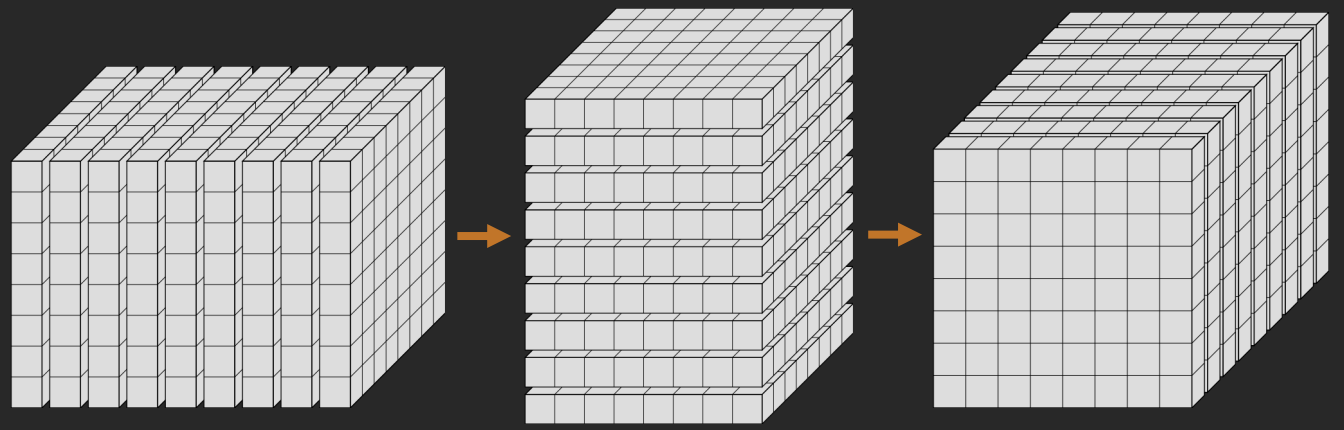
$$\frac{\partial \alpha_i}{\partial t} + \boldsymbol{u} \cdot \nabla \alpha_i + \nabla \cdot \boldsymbol{F}_{\alpha_i} = S_{\alpha_i},$$

# MPI communication

Nearest Neighbor Communication

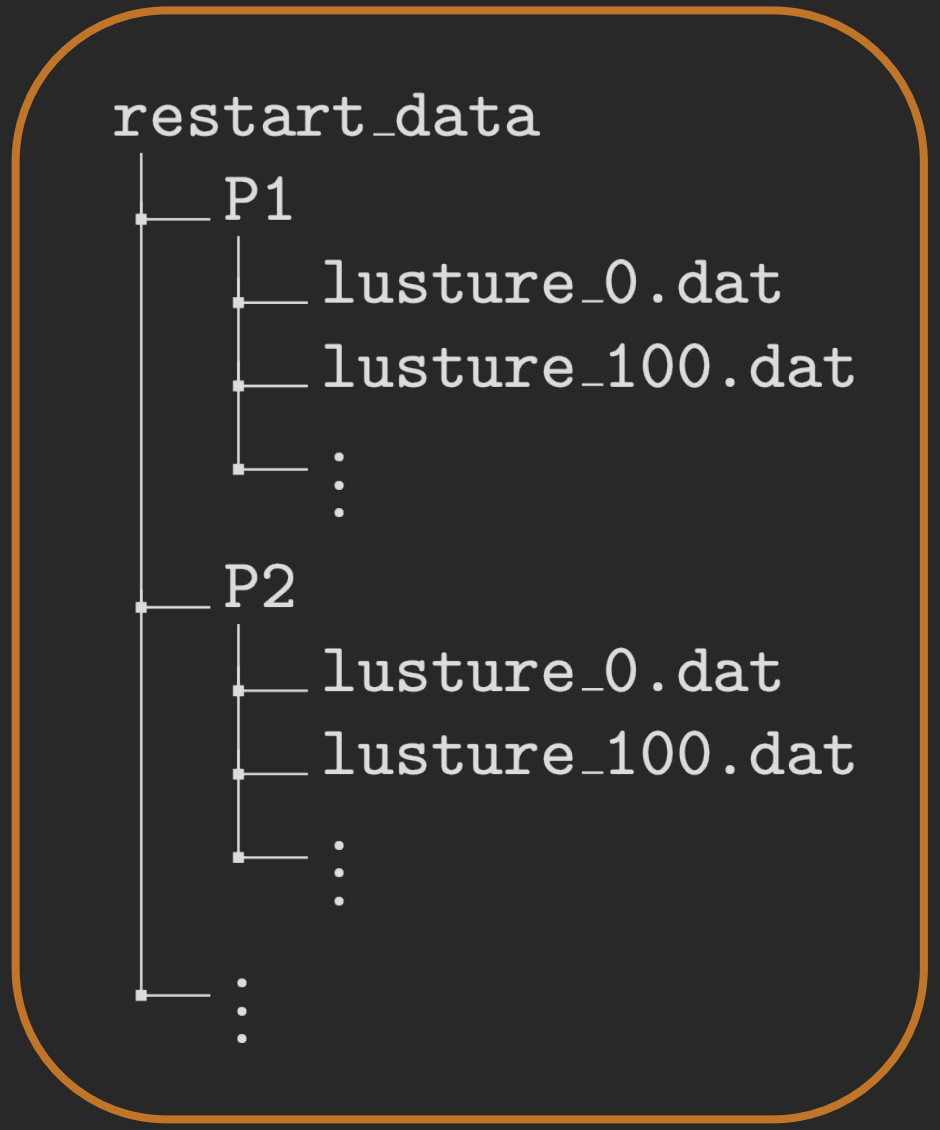


Communication Pattern



Simultaneous communication of many small buffers in each direction results in low communication cost

# Input/Output – File-per-process



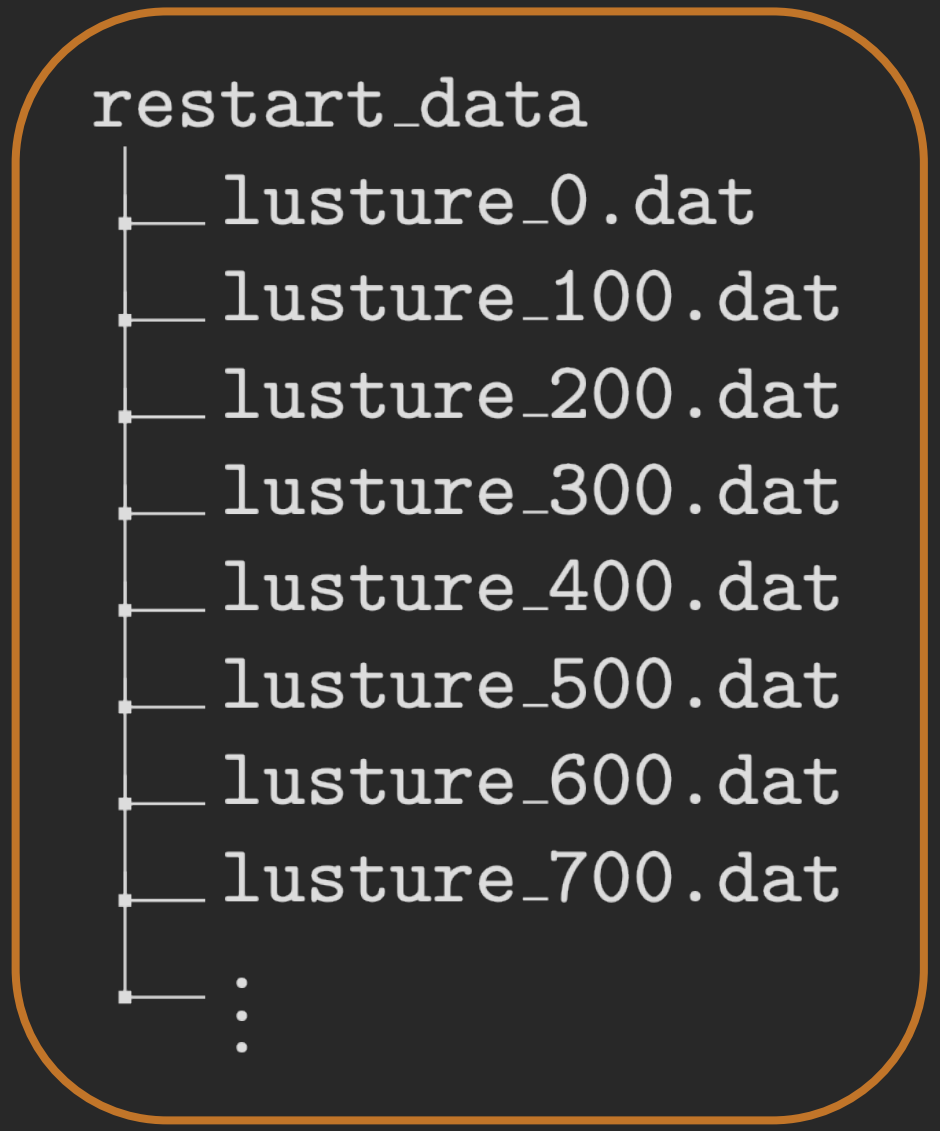

**Pros:**

- Simple
- Fast

**Cons:**

- O($10^6$) files for leadership scale simulation
- Lots of concurrent metadata creation

# Input/Output – Shared-file




**Pros:**

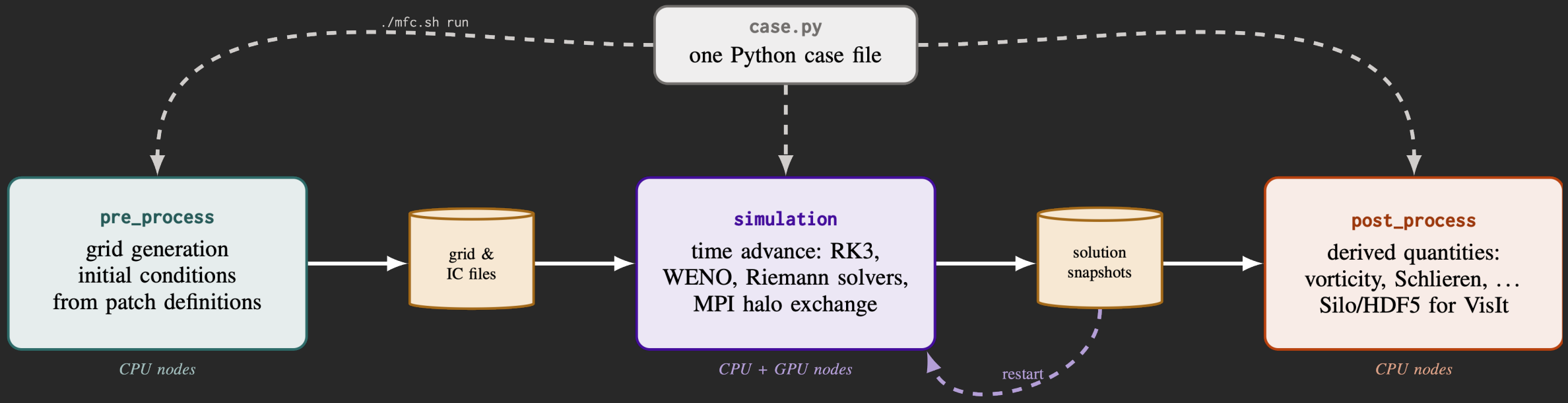
- Parallelism
- O($10^3$) files for leadership scale simulation
- Minor metadata creation

**Cons:**

- Only expressed in MPI
- Not fast on all systems

- Introduction
- Model and Numerical Method
- **Code Design**
- Continuous Integration
- Performance Characteristics
- Closing Remarks

# Code architecture




- Pre-processing, simulation, and post-processing are independent exectubales coupled by files on disk
- Pre-processing and post-processing can be performed on CPU nodes with more ranks/node
- Post-processing can be repeated if more derived quantities are needed
- Single `case.py` drives all three

# GPU Kernels – Overview

| | OpenMP | | OpenACC | |
|---|---|---|---|---|
| | NVIDIA | AMD | NVIDIA | AMD |
| AMDFlang | ✗ | ✔ | ✗ | ✗ |
| CCE† | ✔ | ✔ | ✔ | ✔ |
| NVHPC | ✔ | ✗ | ✔ | ✗ |

† Only available on HPE/Cray systems

- MFC supports OpenACC and OpenMP offloading for increased portability
- No single model supports both dialects on all hardware
- OpenACC preferred for NVIDIA hardware
- OpenMP preferred for AMD hardware

- FYPP + Compiler guards allow for a single developer-facing source to compile to offload with both backends

**(a) Developer-facing code**

```
$:GPU_PARALLEL_LOOP(collapse=3, private='[]',...)
do l = 0, p        ! Z-direction
  do k = 0, n      ! Y-direction
    do j = 0, m    ! X-direction
      $:GPU_LOOP(parallelism='[seq]')
      do i = 1, num_PDEs
        ! Core kernel here
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
$:END_GPU_PARALLEL_LOOP()
```

Fypp Preprocessing

**(b) Backend Implementations**

```
#ifdef MFC_OpenACC
  $:ACC_PARALLEL_LOOP(collapse=3, private='[]',...)
#elifdef MFC_OpenMP
  $:OMP_PARALLEL_LOOP(collapse=3, private='[]',...)
#endif
do l = 0, p        ! Z-direction
  do k = 0, n      ! Y-direction
    do j = 0, m    ! X-direction
#ifdef MFC_OpenACC
      $:ACC_GPU_LOOP(parallelism='[seq]')
#elifdef MFC_OpenMP
      $:OMP_GPU_LOOP(parallelism='[seq]')
#endif
      do i = 1, num_PDEs
        ! Core kernel here
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
#ifdef MFC_OpenACC
  $:ACC_END_PARALLEL_LOOP()
#elifdef MFC_OpenMP
  $:OMP_END_PARALLEL_LOOP()
#endif
```

**(c) OpenMP Codegen (CCE)**

```
!$omp target teams distribute parallel do &
!$omp defaultmap(firstprivate:scalar) &
!$omp defaultmap(tofrom:aggregate) &
!$omp defaultmap(present:allocatable) &
!$omp defaultmap(present:pointer) &
!$omp collapse(3) private(...)
do l = 0, p        ! Z-direction
  do k = 0, n      ! Y-direction
    do j = 0, m    ! X-direction
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$omp end target teams distribute parallel do
```

-DMFC_OpenMP

-DMFC_OpenACC

**(d) OpenACC Codegen**

```
!$acc parallel loop vector gang collapse(3) &
!$acc default(present) private(...)
do l = 0, p        ! Z-direction
  do k = 0, n      ! Y-direction
    do j = 0, m    ! X-direction
      !$acc loop seq
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$acc end parallel loop
```

# GPU kernels – Developer facing code

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**Developer-facing code**

```
$:GPU_PARALLEL_LOOP(collapse=3, private='[]',...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      $:GPU_LOOP(parallelism='[seq]')
      do i = 1, num_PDEs
        ! Core kernel here
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
$:END_GPU_PARALLEL_LOOP()
```

Identification of parallel region

Identification of parallel region

# GPU kernels – Developer facing code

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**Developer-facing code**

```
$:GPU_PARALLEL_LOOP(collapse=3, private='[]',...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      $:GPU_LOOP(parallelism='[seq]')
      do i = 1, num_PDEs
        ! Core kernel here
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
$:END_GPU_PARALLEL_LOOP()
```

Thread serial inner loop

# GPU kernels – Backend implementation

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**Backend Implementations**

```
#ifdef MFC_OpenACC
  $:ACC_PARALLEL_LOOP(collapse=3, private='[]',...)
#elifdef MFC_OpenMP
  $:OMP_PARALLEL_LOOP(collapse=3, private='[]',...)
#endif
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
#ifdef MFC_OpenACC
      $:ACC_GPU_LOOP(parallelism='[seq]')
#elifdef MFC_OpenMP
      $:OMP_GPU_LOOP(parallelism='[seq]')
#endif
      do i = 1, num_PDEs
        ! Core kernel here
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
#ifdef MFC_OpenACC
  $:ACC_END_PARALLEL_LOOP()
#elifdef MFC_OpenMP
  $:OMP_END_PARALLEL_LOOP()
#endif
```

Expansion of
$:GPU_PARALLEL_LOOP

# GPU kernels – Backend implementation

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**Backend Implementations**

```
#ifdef MFC_OpenACC
  $:ACC_PARALLEL_LOOP(collapse=3, private='[]',...)
#elifdef MFC_OpenMP
  $:OMP_PARALLEL_LOOP(collapse=3, private='[]',...)
#endif
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
#ifdef MFC_OpenACC
      $:ACC_GPU_LOOP(parallelism='[seq]')
#elifdef MFC_OpenMP
      $:OMP_GPU_LOOP(parallelism='[seq]')
#endif
      do i = 1, num_PDEs
        ! Core kernel here
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
#ifdef MFC_OpenACC
  $:ACC_END_PARALLEL_LOOP()
#elifdef MFC_OpenMP
  $:OMP_END_PARALLEL_LOOP()
#endif
```

Expansion of $:GPU_LOOP

# GPU kernels – Backend implementation

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**Backend Implementations**

```
#ifdef MFC_OpenACC
  $:ACC_PARALLEL_LOOP(collapse=3, private='[]',...)
#elifdef MFC_OpenMP
  $:OMP_PARALLEL_LOOP(collapse=3, private='[]',...)
#endif
do l = 0, p        ! Z-direction
  do k = 0, n      ! Y-direction
    do j = 0, m    ! X-direction
#ifdef MFC_OpenACC
      $:ACC_GPU_LOOP(parallelism='[seq]')
#elifdef MFC_OpenMP
      $:OMP_GPU_LOOP(parallelism='[seq]')
#endif
      do i = 1, num_PDEs
        ! Core kernel here
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
#ifdef MFC_OpenACC
  $:ACC_END_PARALLEL_LOOP()
#elifdef MFC_OpenMP
  $:OMP_END_PARALLEL_LOOP()
#endif
```

Expansion of
$:END_GPU_PARALLEL_LOOP

# GPU kernels – OpenACC codegen

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**OpenACC Codegen**

```
!$acc parallel loop vector gang collapse(3) &
!$acc default(present) private(...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      !$acc loop seq
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$acc end parallel loop
```

Maps work to blocks, warps, and threads

# GPU kernels – OpenACC codegen

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**OpenACC Codegen**

```
!$acc parallel loop vector gang collapse(3) &
!$acc default(present) private(...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      !$acc loop seq
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$acc end parallel loop
```

Unrolls loop into a single logical loop

# GPU kernels – OpenACC codegen

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**OpenACC Codegen**

```
!$acc parallel loop vector gang collapse(3) &
!$acc default(present) private(...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      !$acc loop seq
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$acc end parallel loop
```

List of thread private arrays

# GPU kernels – OpenACC codegen

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**OpenACC Codegen**

```
!$acc parallel loop vector gang collapse(3) &
!$acc default(present) private(...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      !$acc loop seq
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$acc end parallel loop
```

Assumes all variables present on device (no implicit transfers)

# GPU kernels – OpenMP codegen

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**OpenMP Codegen (CCE)**

```
!$omp target teams distribute parallel do &
!$omp defaultmap(firstprivate:scalar) &
!$omp defaultmap(tofrom:aggregate) &
!$omp defaultmap(present:allocatable) &
!$omp defaultmap(present:pointer) &
!$omp collapse(3) private(...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$omp end target teams distribute parallel do
```

Maps work to blocks, warps, and threads

# GPU kernels – OpenMP codegen

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**OpenMP Codegen (CCE)**

```
!$omp target teams distribute parallel do &
!$omp defaultmap(firstprivate:scalar) &
!$omp defaultmap(tofrom:aggregate) &
!$omp defaultmap(present:allocatable) &
!$omp defaultmap(present:pointer) &
!$omp collapse(3) private(...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$omp end target teams distribute parallel do
```

Sets default memory map for each variable type, similar to OpenACC's `default(present)`

# GPU kernels – OpenMP codegen

Spatial loops each with range O(100)

Equation loop with range O(1)

Core operations (WENO, HLLC, etc)

**OpenMP Codegen (CCE)**

```
!$omp target teams distribute parallel do &
!$omp defaultmap(firstprivate:scalar) &
!$omp defaultmap(tofrom:aggregate) &
!$omp defaultmap(present:allocatable) &
!$omp defaultmap(present:pointer) &
!$omp collapse(3) private(...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$omp end target teams distribute parallel do
```

List of thread private arrays

# GPU kernels – OpenMP codegen

Spatial loops each with range O(100)

Equation loop with range O(1)

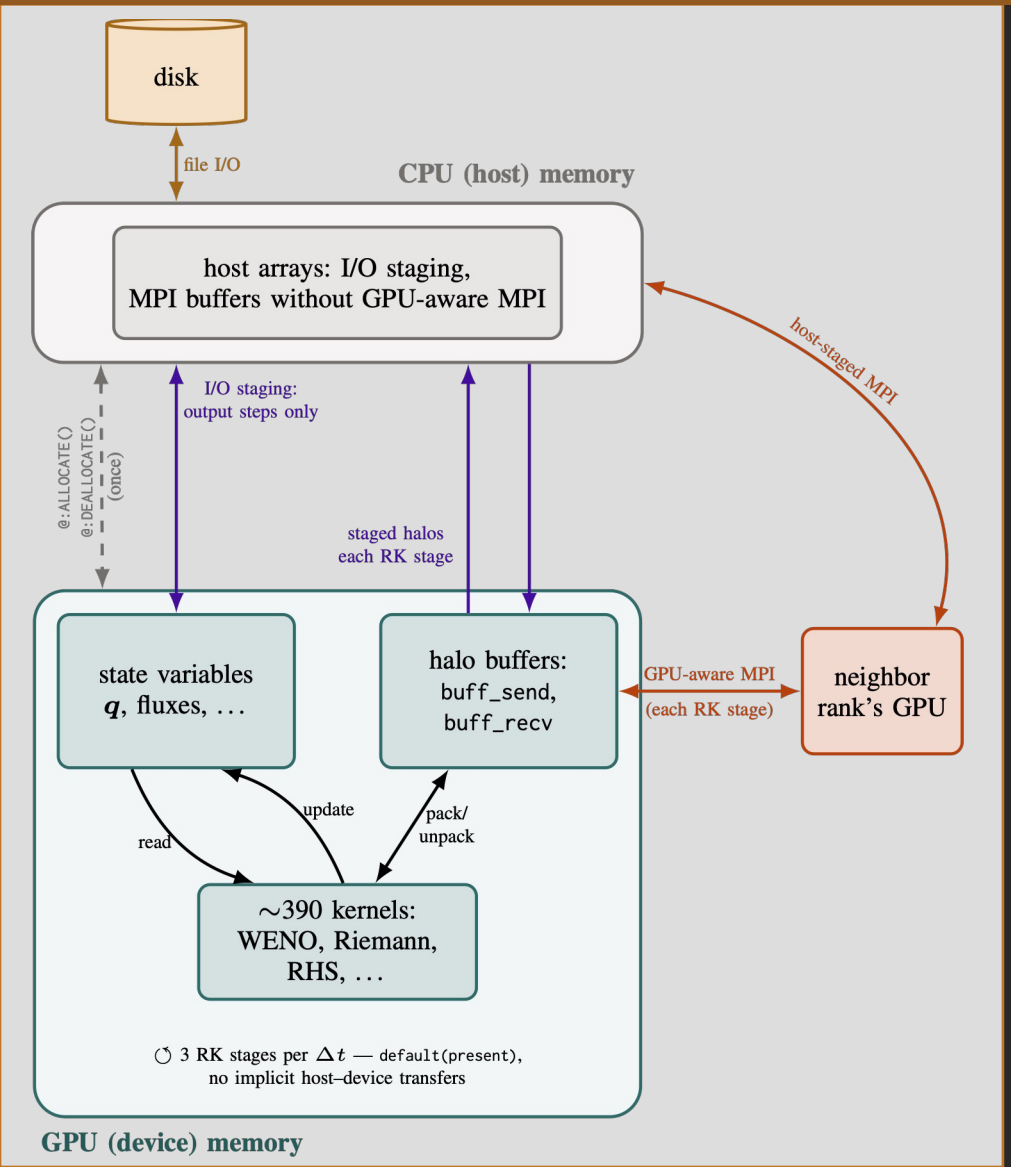
Core operations (WENO, HLLC, etc)

**OpenMP Codegen (CCE)**

```
!$omp target teams distribute parallel do &
!$omp defaultmap(firstprivate:scalar) &
!$omp defaultmap(tofrom:aggregate) &
!$omp defaultmap(present:allocatable) &
!$omp defaultmap(present:pointer) &
!$omp collapse(3) private(...)
do l = 0, p          ! Z-direction
  do k = 0, n        ! Y-direction
    do j = 0, m      ! X-direction
      do i = 1, num_PDEs
        ! Core kernel
        ! O(100) - O(1000) operations
      end do
    end do
  end do
end do
!$omp end target teams distribute parallel do
```

Unrolls loop into a single logical loop

# Device-resident memory model




**@(DE)ALLOCATE macro definition**

```
#:def ALLOCATE(*args)
    #:set variables = ', '.join(args)
    allocate (${variables}$)
    #:set cleaned = strip_parenths(variables)
    #:set joined = ', '.join(cleaned)
    $:GPU_ENTER_DATA(create='[' + joined + ']')
#:enddef ALLOCATE
```

- Host-device memory transfers are expensive, minimize them
- Large host → device transfers happen once during initialization using the `@:ALLOCATE` macro
- Large device → host transfers happen at temporally sparse input/output operations
- Small host ↔ device transfers happen at halo exchanges
  - Can be circumvented with GPU-aware MPI

- Introduction
- Model and Numerical Method
- Code Design
- **Continuous Integration**
- Performance Characteristics
- Closing Remarks

# Continuous integration overview

Testing serves to maintain correctness, performance, portability, and quality at every pull request

### Every push & pull request — GitHub-hosted runners

| | |
|---|---|
| **lint & style** | formatting, spell check, source/toolchain/docs/parameter linters |
| **build & test** | GNU & Intel; Ubuntu & macOS; release & debug; double & single precision; coverage-selected regression tests |
| **NVHPC 23.11–26.3** | CPU build + full test suite; GPU builds (OpenACC & OpenMP); case-optimized build check |
| **code quality** | test-coverage check, cleanliness, source DRYness, line counts |
| **numerics** | grid convergence, floating-point stability |
| **docs & packaging** | documentation build, Homebrew formula, toolchain compatibility |

### Non-draft PRs to `MFlowCode/MFC` — self-hosted & gated jobs

| | |
|---|---|
| **GT Phoenix** | NVHPC GPU regression tests (ACC & OMP); GNU CPU tests |
| **OLCF Frontier** | CCE GPU tests (ACC & OMP) & CPU tests; AMDFlang GPU tests (OMP) & CPU tests |
| **both systems** | case-optimized regression suite |
| **AI review** | Claude PR review & `@claude` assistant (API secret) |

### PRs with maintainer approval*

| | |
|---|---|
| **benchmarking** | grindtime & compile time, PR vs. `master`: Phoenix (CPU, ACC, OMP) and Frontier (ACC, OMP, AMDFlang) |

* or trusted author, or manual dispatch by a maintainer

### Scheduled & release — upstream `master` only

| | |
|---|---|
| **daily** | documentation deploy, coverage-map health check |
| **weekly** | Docker images, coverage-map refresh (on Phoenix) |
| **release tags** | Homebrew formula & tap updates, container publication |

# Continuous integration overview

**Every push & pull request — GitHub-hosted runners**

| | |
|---|---|
| lint & style | formatting, spell check, source/toolchain/docs/parameter linters |
| build & test | GNU & Intel; Ubuntu & macOS; release & debug; double & single precision; coverage-selected regression tests |
| NVHPC 23.11–26.3 | CPU build + full test suite; GPU builds (OpenACC & OpenMP); case-optimized build check |
| code quality | test-coverage check, cleanliness, source DRYness, line counts |
| numerics | grid convergence, floating-point stability |
| docs & packaging | documentation build, Homebrew formula, toolchain compatibility |

**Non-draft PRs to `MFlowCode/MFC` — self-hosted & gated jobs**

| | |
|---|---|
| GT Phoenix | NVHPC GPU regression tests (ACC & OMP); GNU CPU tests |
| OLCF Frontier | CCE GPU tests (ACC & OMP) & CPU tests; AMDFlang GPU tests (OMP) & CPU tests |
| both systems | case-optimized regression suite |
| AI review | Claude PR review & `@claude` assistant (API secret) |

**PRs with maintainer approval***

| | |
|---|---|
| benchmarking | grindtime & compile time, PR vs. `master`: Phoenix (CPU, ACC, OMP) and Frontier (ACC, OMP, AMDFlang) |

* or trusted author, or manual dispatch by a maintainer

**Scheduled & release — upstream `master` only**

| | |
|---|---|
| daily | documentation deploy, coverage-map health check |
| weekly | Docker images, coverage-map refresh (on Phoenix) |
| release tags | Homebrew formula & tap updates, container publication |

Testing serves to maintain correctness, performance, portability, and quality at every pull request

Correctness Tests:

- Correctness tests compare against golden files stored in git history
- Metadata files document how golden files are created

# Continuous integration overview

**Every push & pull request — GitHub-hosted runners**

| | |
|---|---|
| **lint & style** | formatting, spell check, source/toolchain/docs/parameter linters |
| **build & test** | GNU & Intel; Ubuntu & macOS; release & debug; double & single precision; coverage-selected regression tests |
| **NVHPC 23.11–26.3** | CPU build + full test suite; GPU builds (OpenACC & OpenMP); case-optimized build check |
| **code quality** | test-coverage check, cleanliness, source DRYness, line counts |
| **numerics** | grid convergence, floating-point stability |
| **docs & packaging** | documentation build, Homebrew formula, toolchain compatibility |

**Non-draft PRs to `MFlowCode/MFC` — self-hosted & gated jobs**

| | |
|---|---|
| **GT Phoenix** | NVHPC GPU regression tests (ACC & OMP); GNU CPU tests |
| **OLCF Frontier** | CCE GPU tests (ACC & OMP) & CPU tests; AMDFlang GPU tests (OMP) & CPU tests |
| **both systems** | case-optimized regression suite |
| **AI review** | Claude PR review & `@claude` assistant (API secret) |

**PRs with maintainer approval***

| | |
|---|---|
| **benchmarking** | grindtime & compile time, PR vs. `master`: Phoenix (CPU, ACC, OMP) and Frontier (ACC, OMP, AMDFlang) |

* or trusted author, or manual dispatch by a maintainer

**Scheduled & release — upstream `master` only**

| | |
|---|---|
| **daily** | documentation deploy, coverage-map health check |
| **weekly** | Docker images, coverage-map refresh (on Phoenix) |
| **release tags** | Homebrew formula & tap updates, container publication |

Testing serves to maintain correctness, performance, portability, and quality at every pull request

Correctness Tests:

- Correctness tests compare against golden files stored in git history
- Metadata files document how golden files are created

Performance Tests:

- Measure grindtime between the master branch and the incoming branch for several subsets of physics

# Continuous integration overview

**Every push & pull request — GitHub-hosted runners**

| | |
|---|---|
| lint & style | formatting, spell check, source/toolchain/docs/parameter linters |
| build & test | GNU & Intel; Ubuntu & macOS; release & debug; double & single precision; coverage-selected regression tests |
| NVHPC 23.11–26.3 | CPU build + full test suite; GPU builds (OpenACC & OpenMP); case-optimized build check |
| code quality | test-coverage check, cleanliness, source DRYness, line counts |
| numerics | grid convergence, floating-point stability |
| docs & packaging | documentation build, Homebrew formula, toolchain compatibility |

**Non-draft PRs to `MFlowCode/MFC` — self-hosted & gated jobs**

| | |
|---|---|
| GT Phoenix | NVHPC GPU regression tests (ACC & OMP); GNU CPU tests |
| OLCF Frontier | CCE GPU tests (ACC & OMP) & CPU tests; AMDFlang GPU tests (OMP) & CPU tests |
| both systems | case-optimized regression suite |
| AI review | Claude PR review & `@claude` assistant (API secret) |

**PRs with maintainer approval***

| | |
|---|---|
| benchmarking | grindtime & compile time, PR vs. `master`: Phoenix (CPU, ACC, OMP) and Frontier (ACC, OMP, AMDFlang) |

* or trusted author, or manual dispatch by a maintainer

**Scheduled & release — upstream `master` only**

| | |
|---|---|
| daily | documentation deploy, coverage-map health check |
| weekly | Docker images, coverage-map refresh (on Phoenix) |
| release tags | Homebrew formula & tap updates, container publication |

Testing serves to maintain correctness, performance, portability, and quality at every pull request

Correctness Tests:
- Correctness tests compare against golden files stored in git history
- Metadata files document how golden files are created

Performance Tests:
- Measure grindtime between the master branch and the incoming branch for several subsets of physics

Portability Tests:
- Measure portability across compilers, their versions, and hardware – Correctness + Performance

# Continuous integration overview

**Every push & pull request — GitHub-hosted runners**

| | |
|---|---|
| **lint & style** | formatting, spell check, source/toolchain/docs/parameter linters |
| **build & test** | GNU & Intel; Ubuntu & macOS; release & debug; double & single precision; coverage-selected regression tests |
| **NVHPC 23.11–26.3** | CPU build + full test suite; GPU builds (OpenACC & OpenMP); case-optimized build check |
| **code quality** | test-coverage check, cleanliness, source DRYness, line counts |
| **numerics** | grid convergence, floating-point stability |
| **docs & packaging** | documentation build, Homebrew formula, toolchain compatibility |

**Non-draft PRs to `MFlowCode/MFC` — self-hosted & gated jobs**

| | |
|---|---|
| **GT Phoenix** | NVHPC GPU regression tests (ACC & OMP); GNU CPU tests |
| **OLCF Frontier** | CCE GPU tests (ACC & OMP) & CPU tests; AMDFlang GPU tests (OMP) & CPU tests |
| **both systems** | case-optimized regression suite |
| **AI review** | Claude PR review & `@claude` assistant (API secret) |

**PRs with maintainer approval***

| | |
|---|---|
| **benchmarking** | grindtime & compile time, PR vs. `master`: Phoenix (CPU, ACC, OMP) and Frontier (ACC, OMP, AMDFlang) |

* or trusted author, or manual dispatch by a maintainer

**Scheduled & release — upstream `master` only**

| | |
|---|---|
| **daily** | documentation deploy, coverage-map health check |
| **weekly** | Docker images, coverage-map refresh (on Phoenix) |
| **release tags** | Homebrew formula & tap updates, container publication |

Testing serves to maintain correctness, performance, portability, and quality at every pull request

Correctness Tests:
- Correctness tests compare against golden files stored in git history
- Metadata files document how golden files are created

Performance Tests:
- Measure grindtime between the master branch and the incoming branch for several subsets of physics

Portability Tests:
- Measure portability across compilers, their versions, and hardware – Correctness + Performance

Quality Tests:
- Automated verification of code conformity

# Continuous integration overview

runs on a private repo | needs a public PR | upstream only

### ✓ Any repo, incl. private forks — push, PR, or dispatch

| | |
|---|---|
| **lint & style** | formatting, spell check, source/toolchain/docs/parameter linters |
| **build & test** | GNU & Intel; Ubuntu & macOS; release & debug; double & single precision; coverage-selected regression tests |
| **NVHPC 23.11–26.3** | CPU build + full test suite; GPU builds (OpenACC & OpenMP); case-optimized build check |
| **code quality** | test-coverage check, cleanliness, source DRYness, line counts |
| **numerics** | grid convergence, floating-point stability |
| **docs & packaging** | documentation build, Homebrew formula, toolchain compatibility |

### × Requires a non-draft PR to public `MFlowCode/MFC`

| | |
|---|---|
| **GT Phoenix** | NVHPC GPU regression tests (ACC & OMP); GNU CPU tests |
| **OLCF Frontier** | CCE GPU tests (ACC & OMP) & CPU tests; AMDFlang GPU tests (OMP) & CPU tests |
| **both systems** | case-optimized regression suite |
| **AI review** | Claude PR review & `@claude` assistant (API secret) |

### × Requires upstream PR *and* maintainer approval*

| | |
|---|---|
| **benchmarking** | grindtime & compile time, PR vs. `master`: Phoenix (CPU, ACC, OMP) and Frontier (ACC, OMP, AMDFlang) |

* or trusted author, or manual dispatch by a maintainer

### ○ Upstream `master` automation — no fork or PR path

| | |
|---|---|
| **daily** | documentation deploy, coverage-map health check |
| **weekly** | Docker images, coverage-map refresh (on Phoenix) |
| **release tags** | Homebrew formula & tap updates, container publication |

This is a lot of testing… Not all of it is going to run on your fork or private repository

**What does this mean?**

If you modify code that can negatively impact correctness, performance, or portability, you won't be able to keep those changes private

- Introduction
- Model and Numerical Method
- Code Design
- Continuous Integration
- **Performance Characteristics**
- Closing Remarks

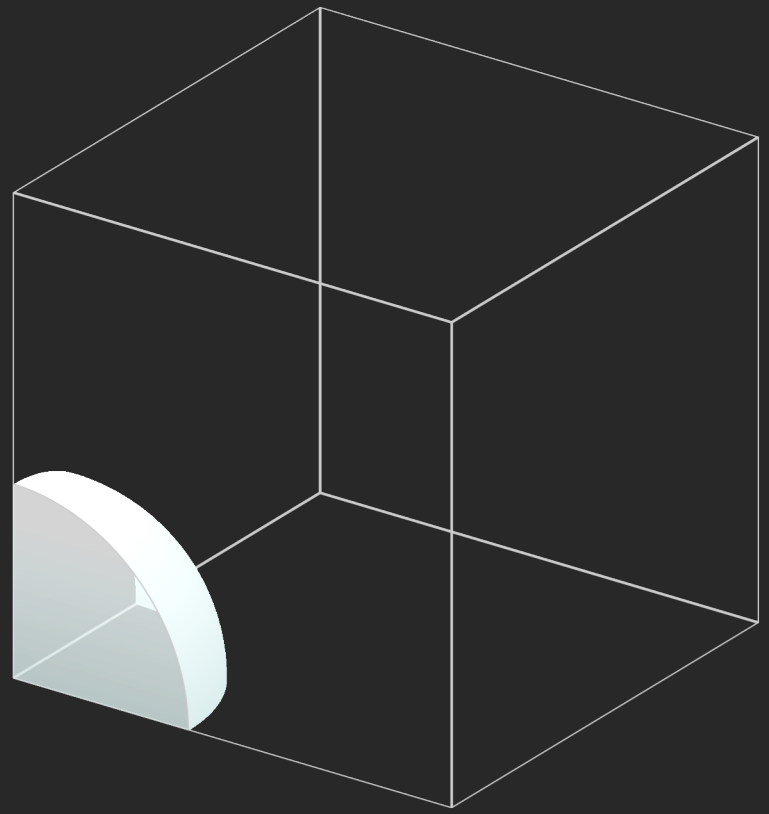
# MFC's standard test case

Common two-fluid problem

Standardized case file and problem size

Compares CPUs to GPUs with normalized QOI

Documented results on ~50 architectures



# What is grindtime?

**Algorithm 1** Grind time: what MFC actually times

**for** time step $t = t_{\text{start}}, \ldots, t_{\text{stop}}$ **do**
  **for** RK stage $s = 1, \ldots, n_{\text{stage}}$ **do**
    $\tau_0 \leftarrow$ WallClock() ▷ **timed region begins**
    ComputeRHS($q$)
    $q \leftarrow$ RK stage update of $q$ using RHS
    body forces, pressure relaxation, IBM correction...
    $\tau_1 \leftarrow$ WallClock() ▷ **timed region ends**

    **if** $t - t_{\text{start}} \geq n_{\text{warmup}}$ **then**
      $T \leftarrow \min(T,\ \tau_1 - \tau_0)$
    **end if**
  **end for**
**end for**

$T \leftarrow \max_{\text{MPI ranks}}(T)$

$$\text{grind time} \leftarrow \frac{T \times 10^9}{N_{\text{eq}} \cdot m_{\text{glb}} \cdot n_{\text{glb}} \cdot p_{\text{glb}}}$$ ▷ ns / grid point / equation / RHS

Grindtime surrounds the RHS calculation, Runge-Kutta update, and application of additional physics

The minimum is taken on each rank over all stages and steps after a warmup period has passed

Max time is taken over all processes and converted to the correct units

# What is grindtime?

**Algorithm 1** Grind time: what MFC actually times

**for** time step $t = t_{\text{start}}, \ldots, t_{\text{stop}}$ **do**
  **for** RK stage $s = 1, \ldots, n_{\text{stage}}$ **do**
    $\tau_0 \leftarrow$ WallClock() ▷ **timed region begins**
    ComputeRHS($q$)
    $q \leftarrow$ RK stage update of $q$ using RHS
    body forces, pressure relaxation, IBM correction...
    $\tau_1 \leftarrow$ WallClock() ▷ **timed region ends**

    **if** $t - t_{\text{start}} \geq n_{\text{warmup}}$ **then**
      $T \leftarrow \min(T,\ \tau_1 - \tau_0)$
    **end if**
  **end for**
**end for**

$T \leftarrow \max_{\text{MPI ranks}}(T)$

$$\text{grind time} \leftarrow \frac{T \times 10^9}{N_{\text{eq}} \cdot m_{\text{glb}} \cdot n_{\text{glb}} \cdot p_{\text{glb}}}$$ ▷ ns / grid point / equation / RHS

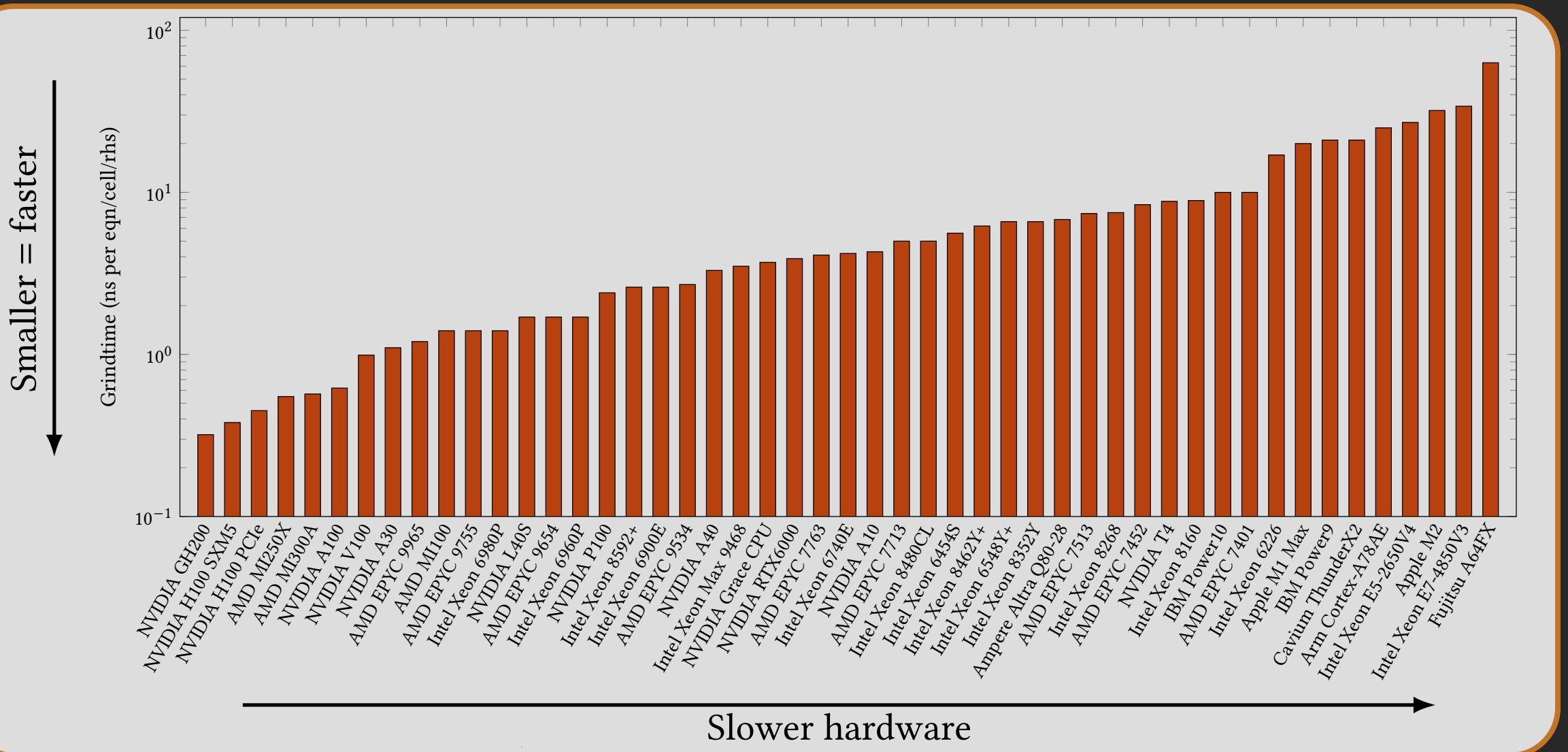
**Why?**

- Timed region measures only time spent doing meaningful computation and physics
- Warmup period skips potentially slow first steps
- Minimum over all time steps and stages gives the most reproducible value for steady-state performance
- Maximum over ranks presents the worst-case value

**What is excluded?**

- I/O – Happens sparsely in time
- MPI spin-wait
- Some chemistry and immersed boundary routines
  - Not enabled in any presented results

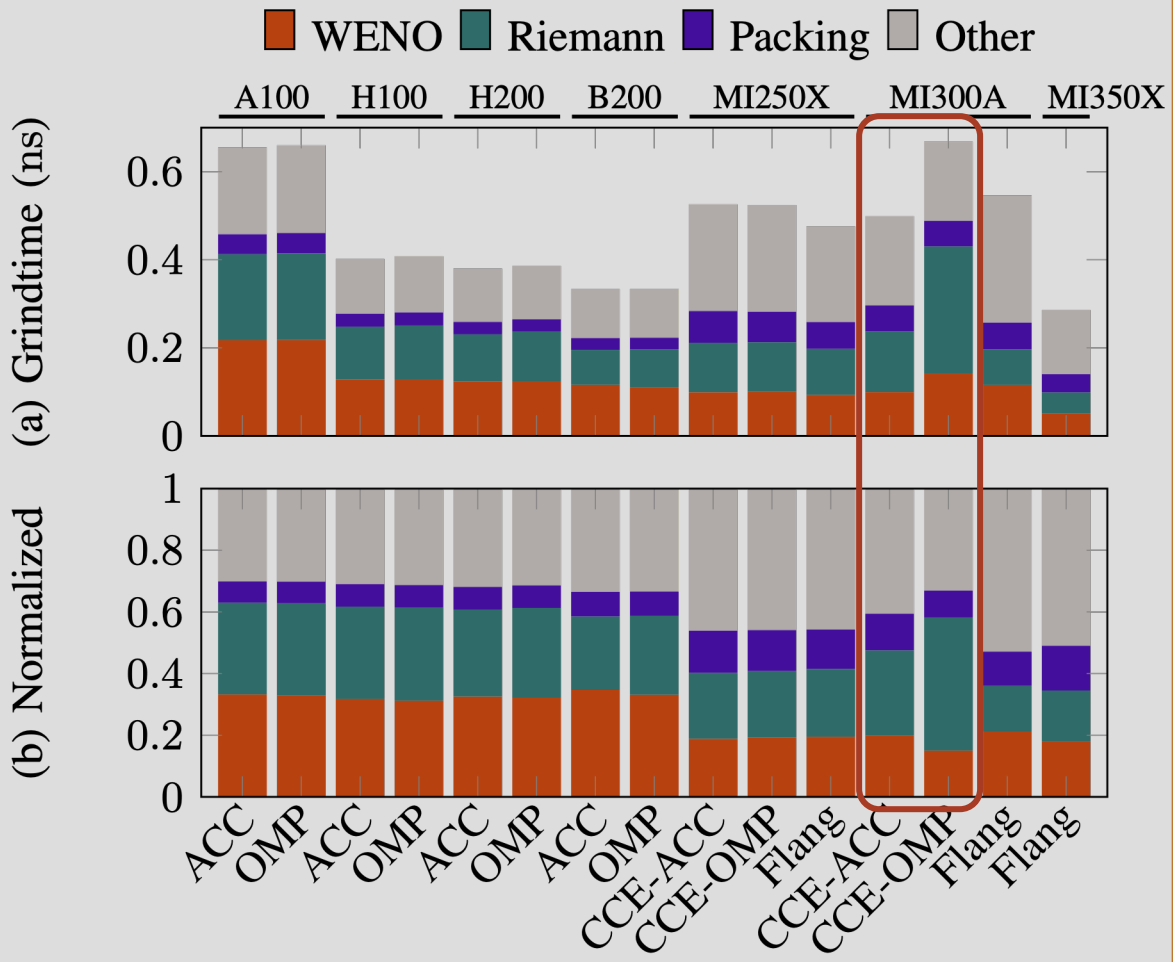
# Standard test case grindtime



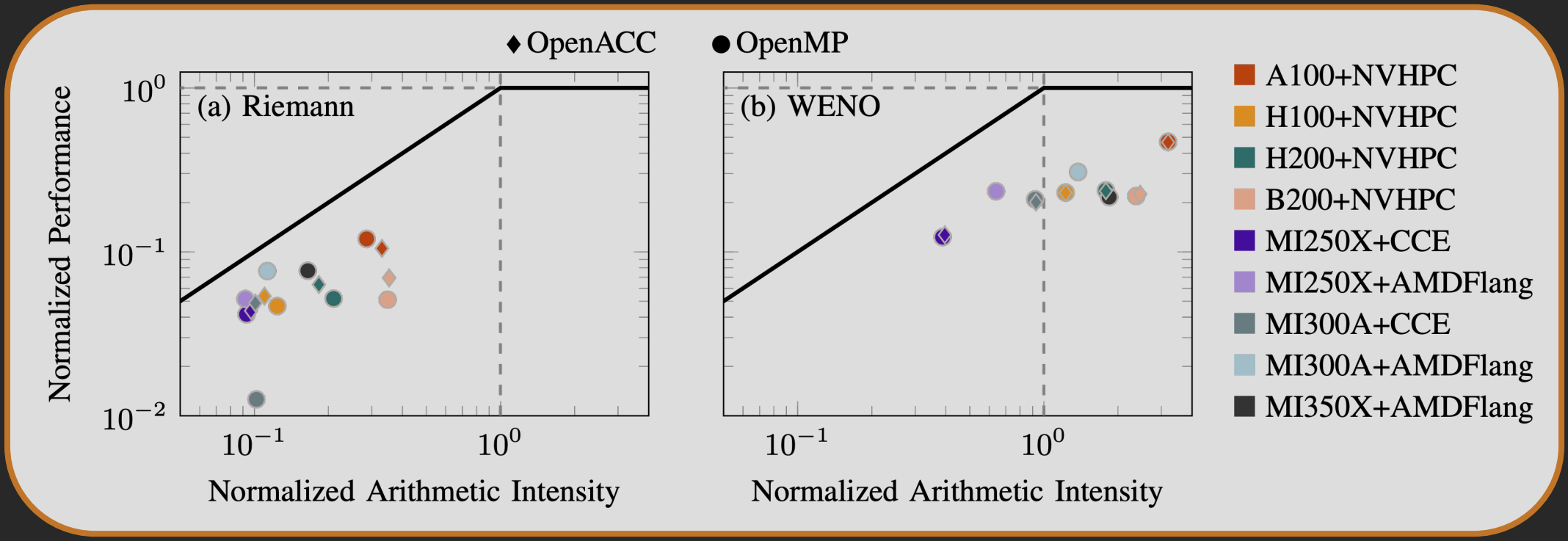
# Time step breakdown

- Large portion of each time step is spent in the WENO and HLLC Riemann kernels
- Array packing makes up a smaller fraction of runtime on NVIDIA hardware than AMD hardware
  - Struct of arrays → 4D array for WENO
- What is other?
  - Boundary conditions
  - Source terms
  - Communication
  - Etc…

Supporting both OpenMP and OpenACC allows us to circumvent a performance regression with the CCE compiler on MI300A

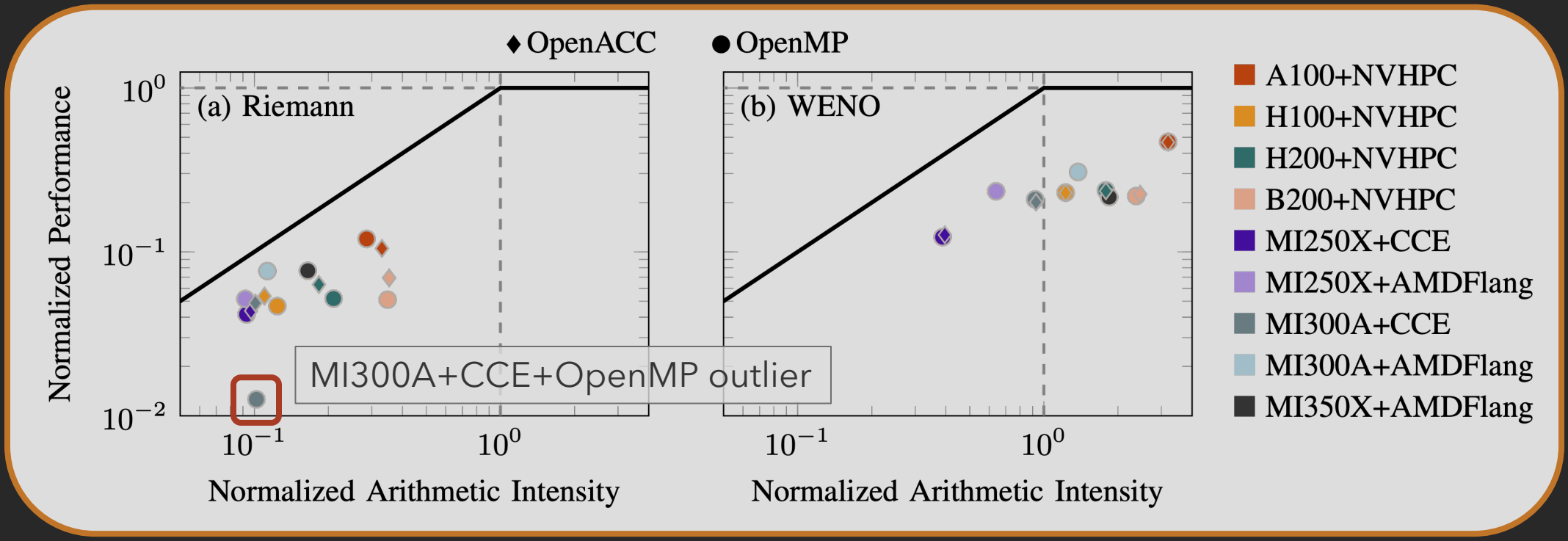


# Roofline performance



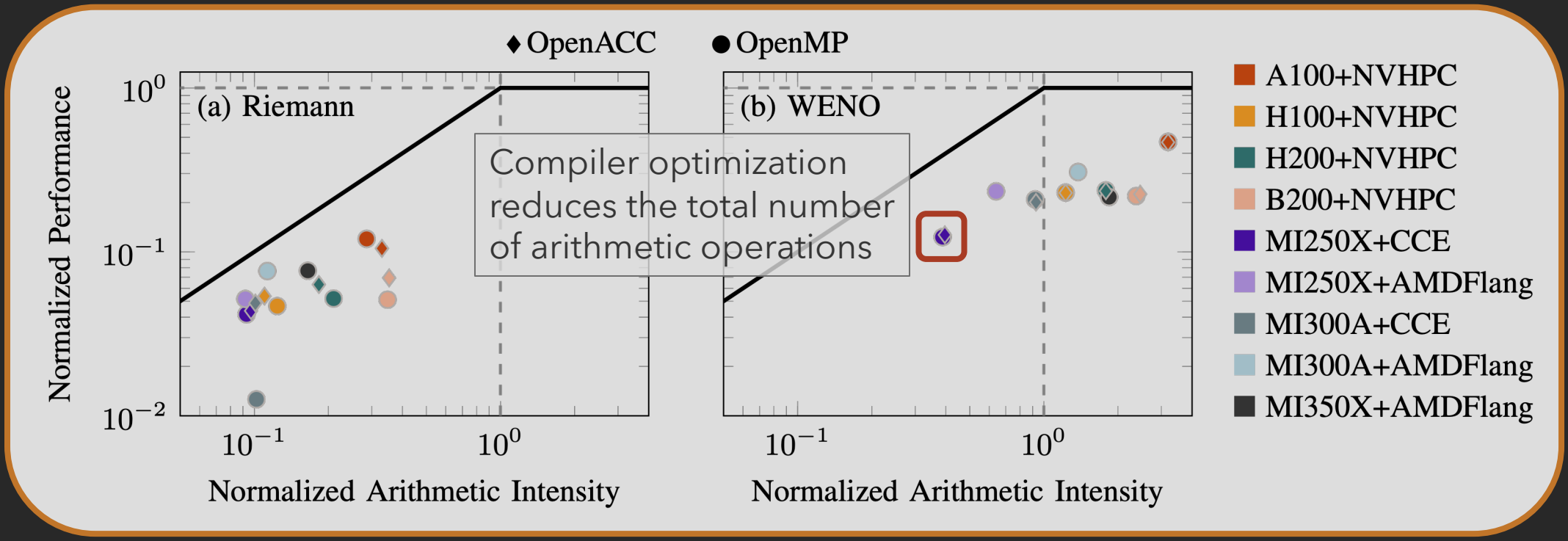

- Riemann: Solidly in the memory-bound regime on all hardware
  - Relies on thread-private arrays for intermediate values, potential for optimization
- WENO: Compute-bound on most hardware

# Roofline performance



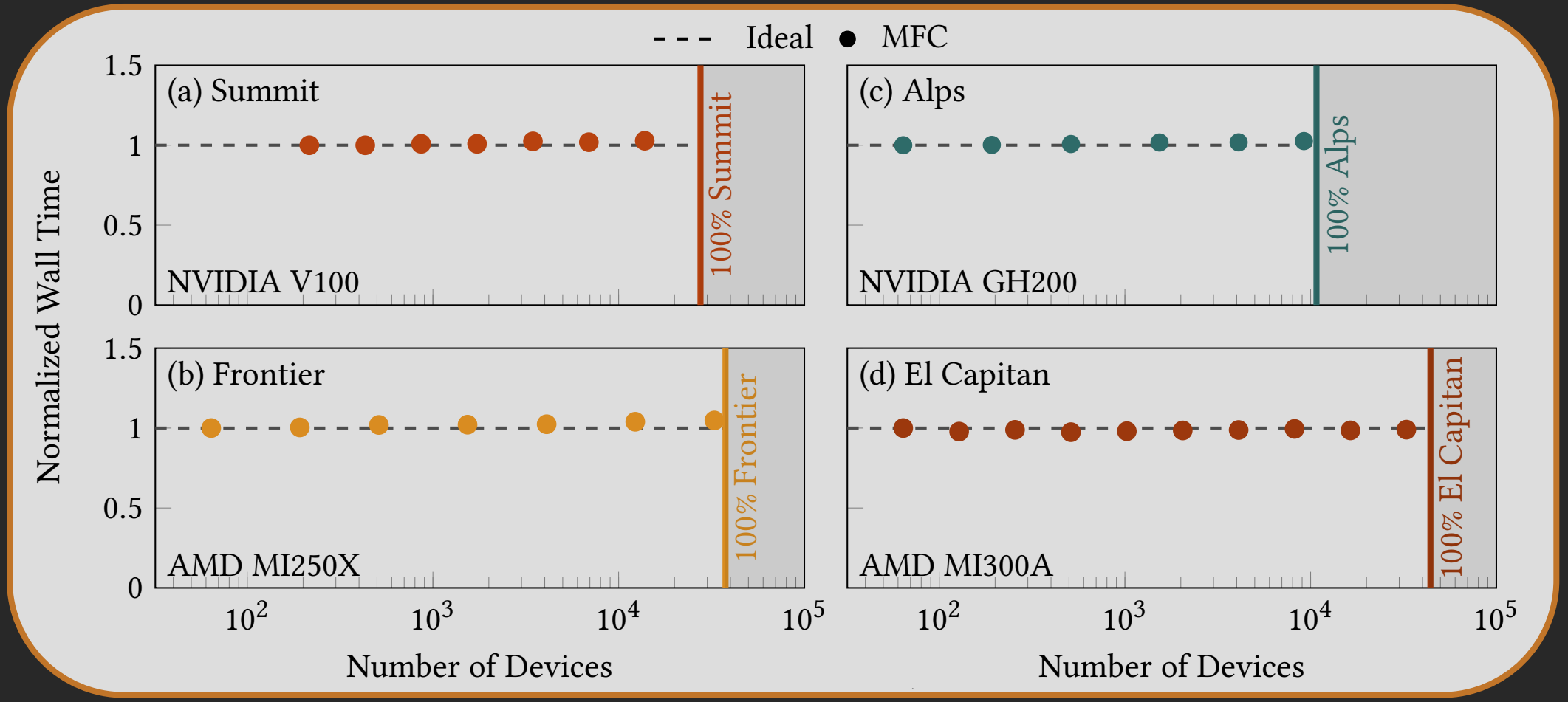

- Riemann: Solidly in the memory-bound regime on all hardware
  - Relies on thread-private arrays for intermediate values, potential for optimization
- WENO: Compute-bound on most hardware

# Roofline performance



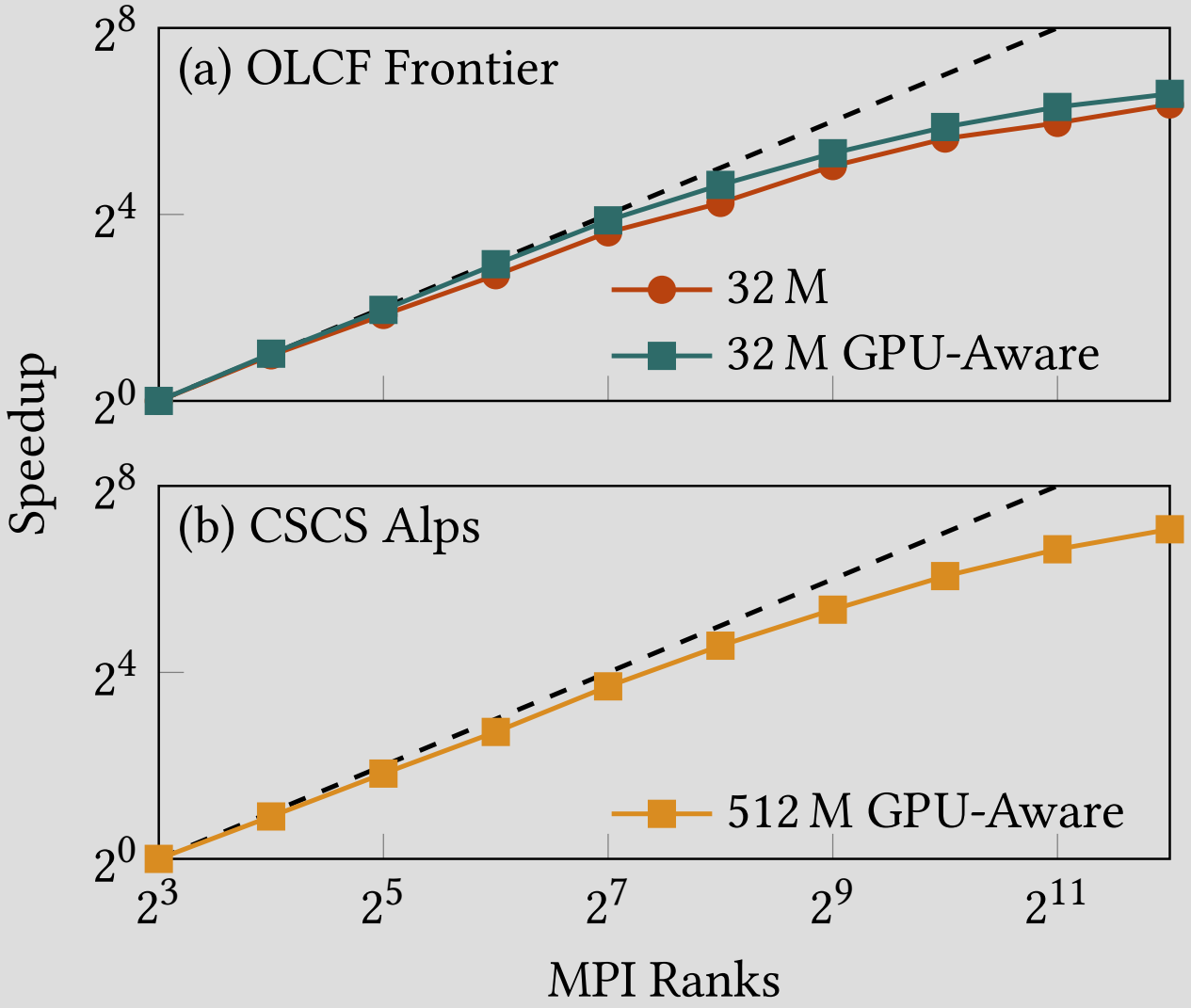

- Riemann: Solidly in the memory-bound regime on all hardware
  - Relies on thread-private arrays for intermediate values, potential for optimization
- WENO: Compute-bound on most hardware

# Weak scaling




Within 5% of ideal over 100-1000x increase in device count

# Strong scaling




Impressive strong scaling performance over large increase in device count

**Reminder:** No global or collective operations

- Introduction
- Model and Numerical Method
- Code Design
- Continuous Integration
- Performance Characteristics
- **Closing Remarks**

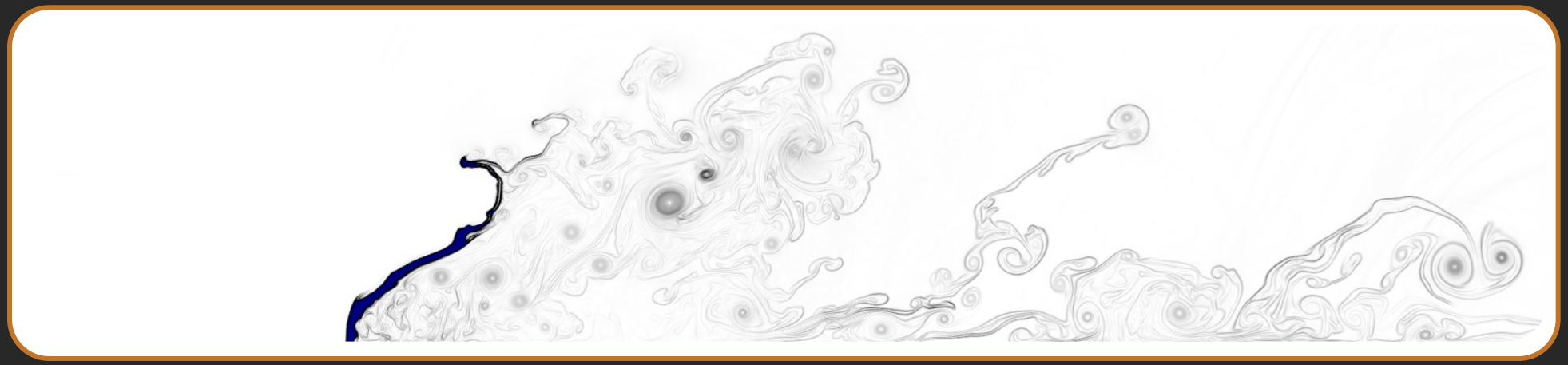
# Ground rules for code changes

- Any changes you make to MFC *must* maintain application performance, portability, and quality
- Application developers have final say
- Any changes to `src/` must pass the full CI test suite via a public GitHub PR to ensure portability is maintained
  - Limit `src/` changes to +/- 5000 lines
- Changes to compiler options (CMake) do not need to pass the full CI test suite, but you must demonstrate that `./mfc.sh test` passes on your hardware and disclose the changes to the application judges in your writeup

# Practice Problems Summary

1. Verify your environment and installation with the test suite
2. Run an unmodified case file and analyze the results
3. Run a modified case file to explore physics and numerics
4. Visualize your results using ParaView or VisIt

Mach 1.4 shock-cylinder interaction



Practice problems will be posted on SCC26 Google Group later today!

# Conclusions

- GPU acceleration is enabled with a custom macro layer that generates OpenACC and OpenMP directives for GPU offload
- Inter-process communication uses MPI
- Continuous integration ensures correctness, performance, portability, and quality are maintained
- Resulting code is largely memory-bound
- Grindtime is used to quantify overall performance

GitHub

MFC

mflowcode.github.io

WEBSITE

# Thank you!

Lawrence Livermore National Laboratory
OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY
U.S. DEPARTMENT OF ENERGY
National Nuclear Security Administration
NSF
ACCESS
Sandia National Laboratories