

# Simultaneous use of CPU and GPU for fast and energy-efficient implicit wave simulation

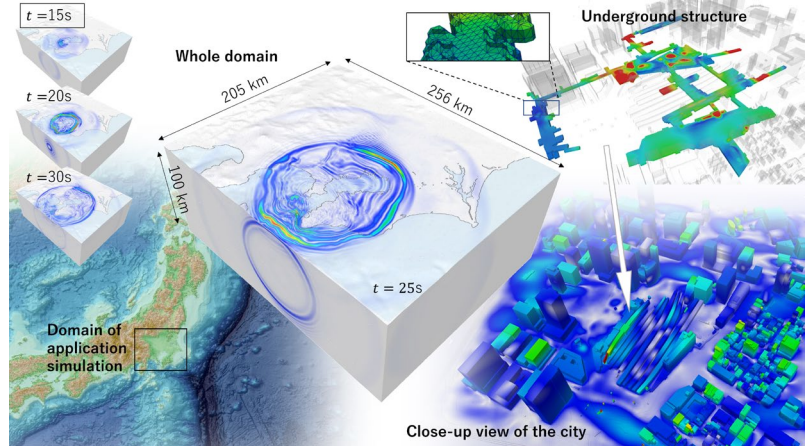
Kohei Fujita

Earthquake Research Institute, The University of Tokyo

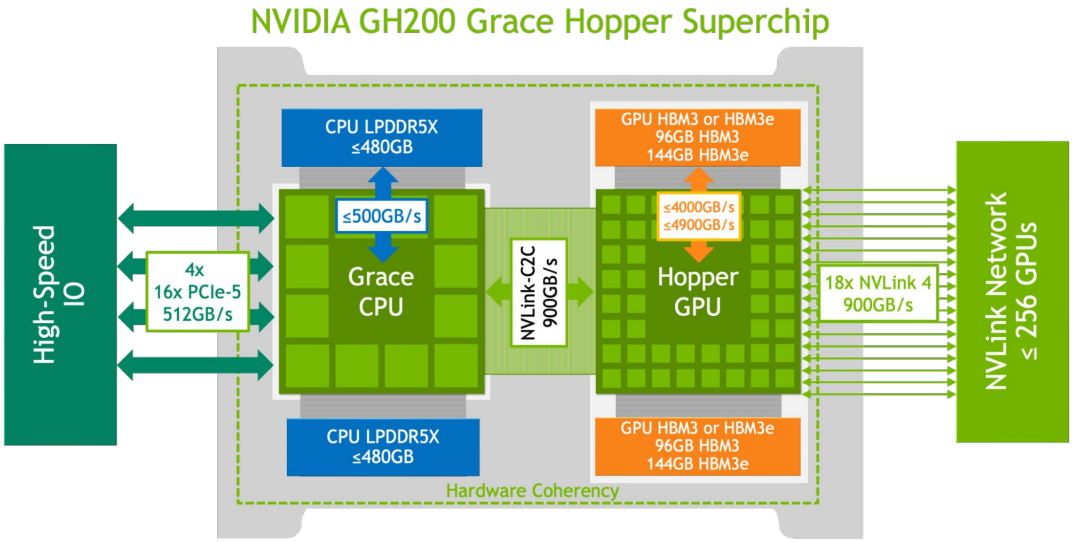
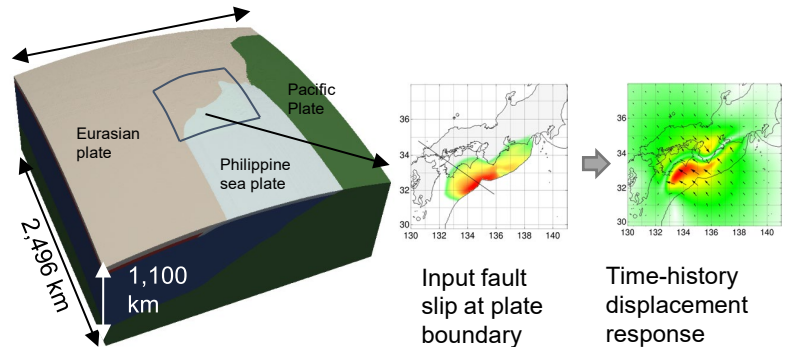
# Introduction

- Earthquake simulations: many-cases of high-resolution PDE-based time history simulations required
- Introduce method that concurrently use CPU and GPU to reduce time- and energy-to-solution

Seismic wave propagation



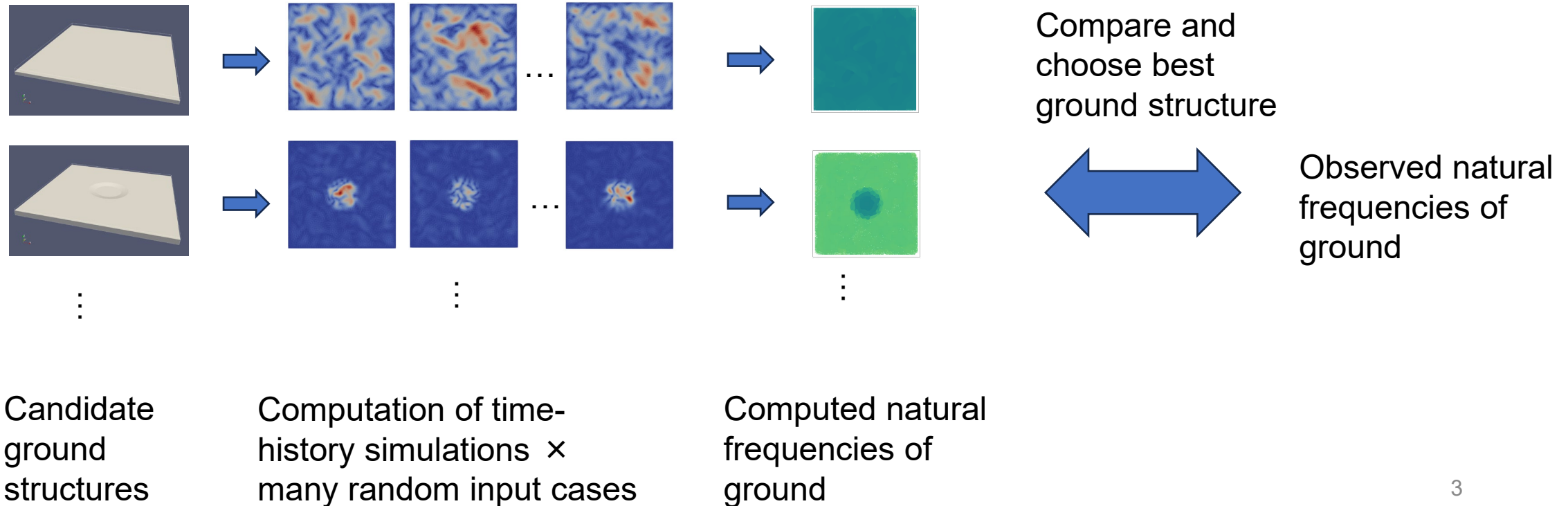
Crustal deformation



**Data-driven part on memory rich CPU, Solver part on high-performance GPU, with fast CPU-GPU synchronization**

# Simulation example

- Estimate the properties of the ground by comparison of numerical analysis results and observation data
  - Compute ground vibration to random wave inputs for many candidate ground models
  - Choose a ground model that has the closest natural frequency to that of actual observations



# Simulation example

- Properties of ground models can be computed using many-case analyses

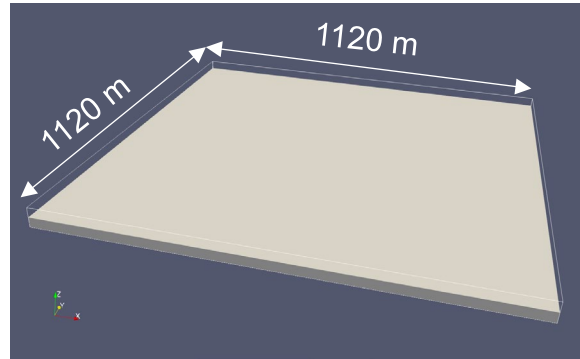
Candidate ground structures (showing the interface between soft and hard ground)



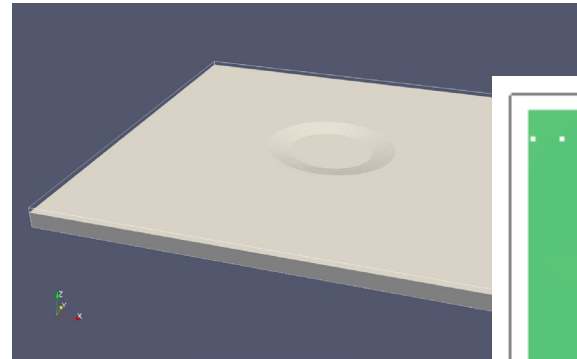
Computation of 16384-time step simulations  
× 8 random input cases



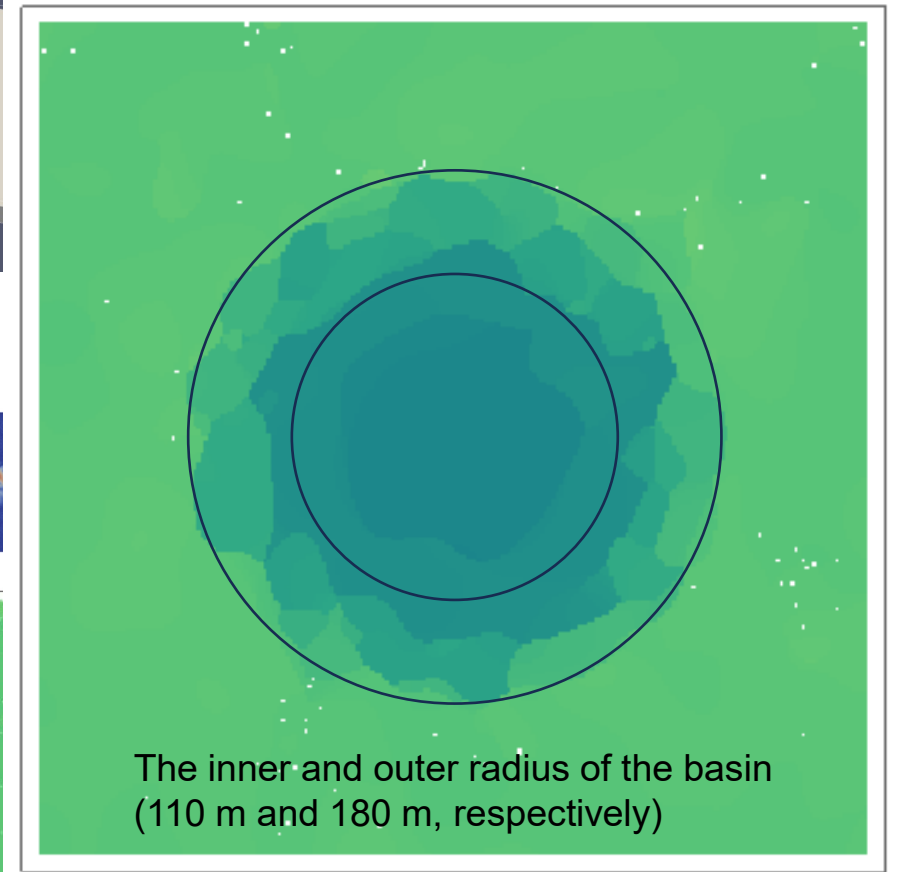
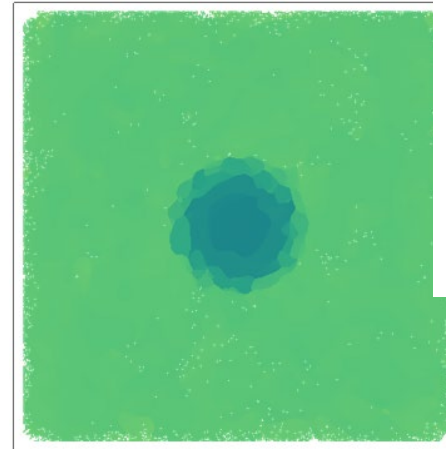
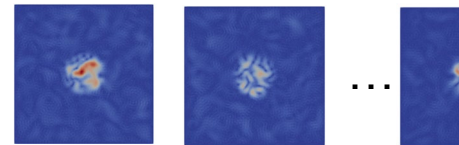
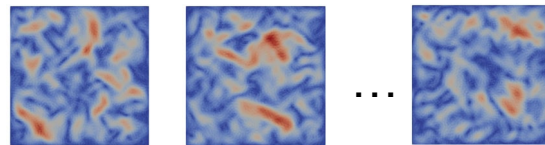
Computed natural frequencies of ground



a) Horizontally stratified ground structure



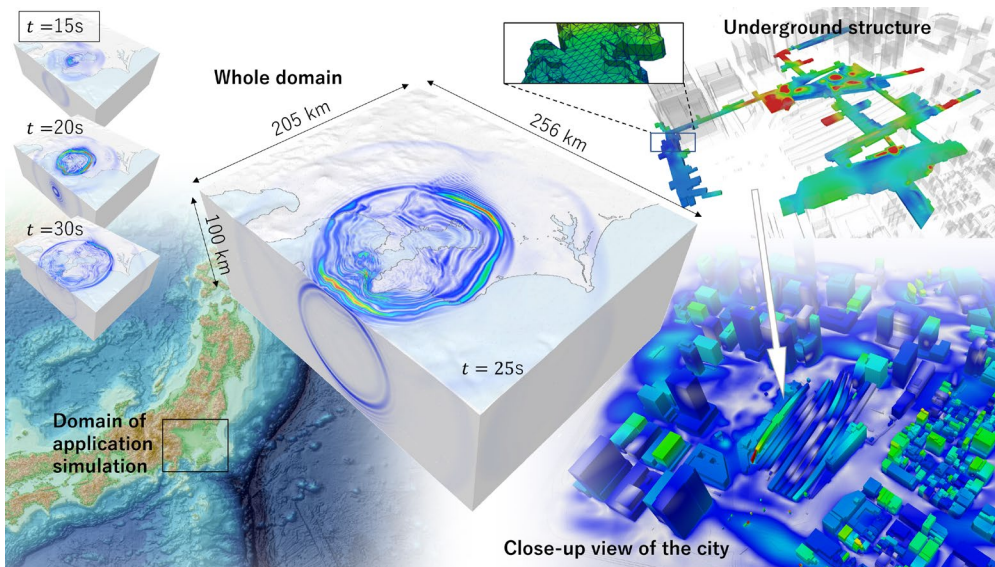
b) Ground structure with circular basin bedrock



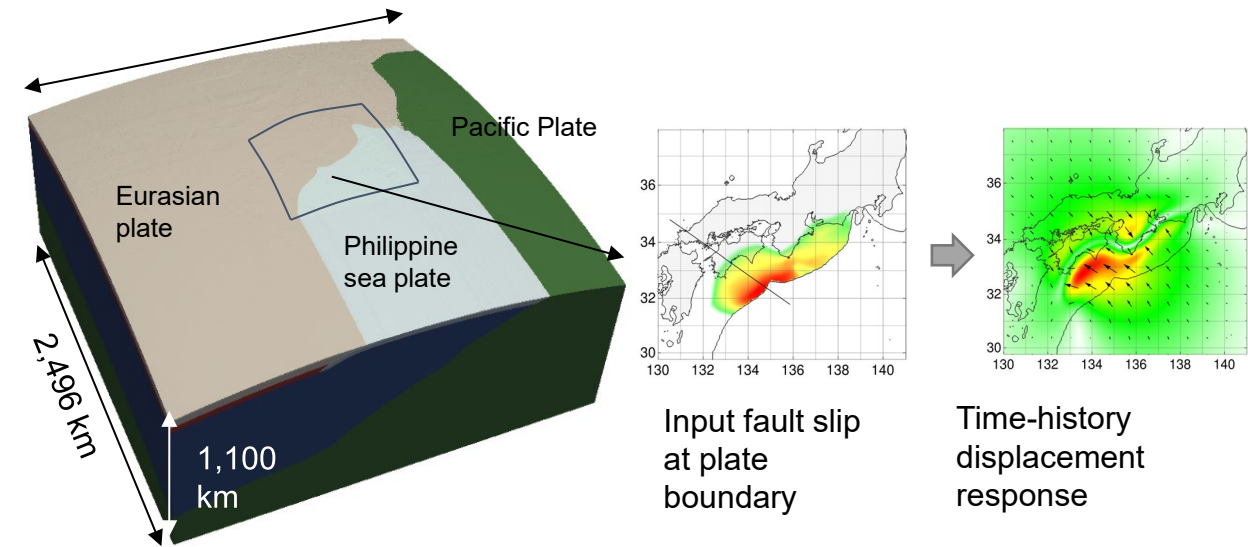
cy

# Use of data-driven method for solving PDEs

- Predict initial solution of implicit solvers in PDE-based time-history simulation using data of previous time steps
- Reduces solver iterations and thus enables speedup without reduction in accuracy



Seismic wave propagation@full Fugaku (152352 nodes)  
25-fold speedup from previous solver without data-driven method  
Ichimura et al. HPC Asia 2022 [Best Paper]



Crustal deformation@Fugaku (73728 nodes)  
Fujita et al. ScalaH 2022

# Use of data-driven method for solving PDEs

Fugaku  
(CPU only)

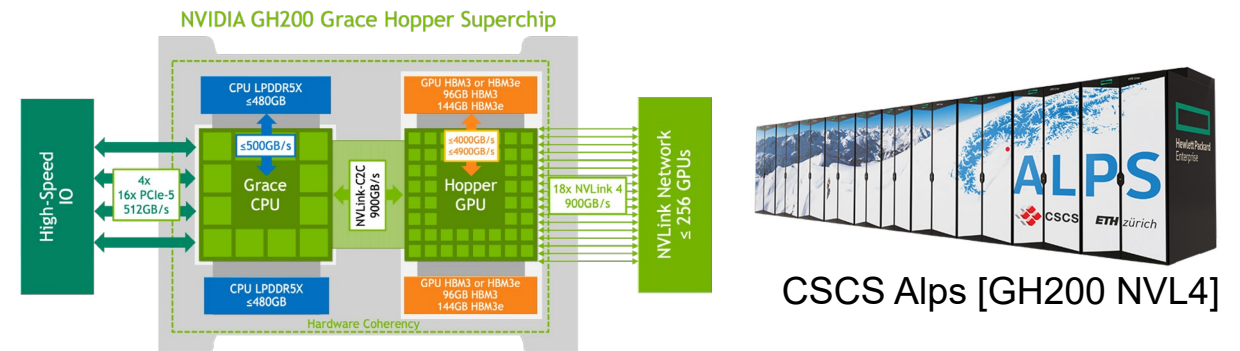
A100  
(GPU only)

GH200  
(CPU+GPU)

HPC Asia (Ichimura et al. 2022)  
ScalaH@SC22 (Fujita et al. 2022)

ICCS 2023  
(Murakami et al.)

WACCPD@SC24 (Ichimura et al.)



**Data-driven part on memory rich CPU,  
Solver part on high-performance GPU**

- In this talk,
  - Review work [1] on concurrently using CPU and GPU for accelerating PDE-based wave simulation
  - Show step by step implementation procedure

[1] Tsuyoshi Ichimura, Kohei Fujita, Muneo Hori, Madgededara Lalith, Jack Wells, Alan Gray, Ian Karlin, John Linford: Heterogeneous computing in a strongly-connected CPU-GPU environment: fast multiple time-evolution equation-based modeling accelerated using data-driven approach, *Workshop on Accelerator Programming and Directives (WACCPD) 2024@SC24*

# Outline

- Concurrent CPU and GPU computing method
  - Target problem and baseline solver
  - Proposed method
  - Performance measurement results
- Implementation procedure
  - GPU acceleration
  - Multi-core CPU acceleration
  - CPU-GPU overlap
- Summary

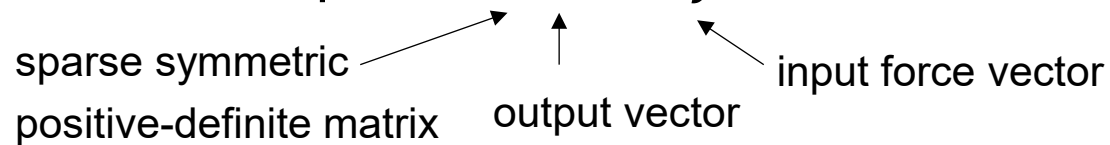
# Introduction

- Accelerating PDE-based time-evolution simulations using recent computational capabilities
  - Large CPU memory: enables storing large-scale data for data-driven methods
  - Fast GPUs + lower programming cost (e.g., directive-based programming models): GPUs widely used for acceleration of PDE-based time-evolution problems
- Use of data-driven methods for acceleration of PDE-based time-evolution problems
  - CPU-based data-driven acceleration using data stored in main memory is available; however, CPUs slower than GPUs
  - Difficult to accelerate GPU-based analysis with data-driven methods while maintaining target problem size as GPU memory is relatively small

# Target problem and baseline solver

- Solve many cases of PDE-based time-evolution problems in a strongly-connected CPU-GPU environment

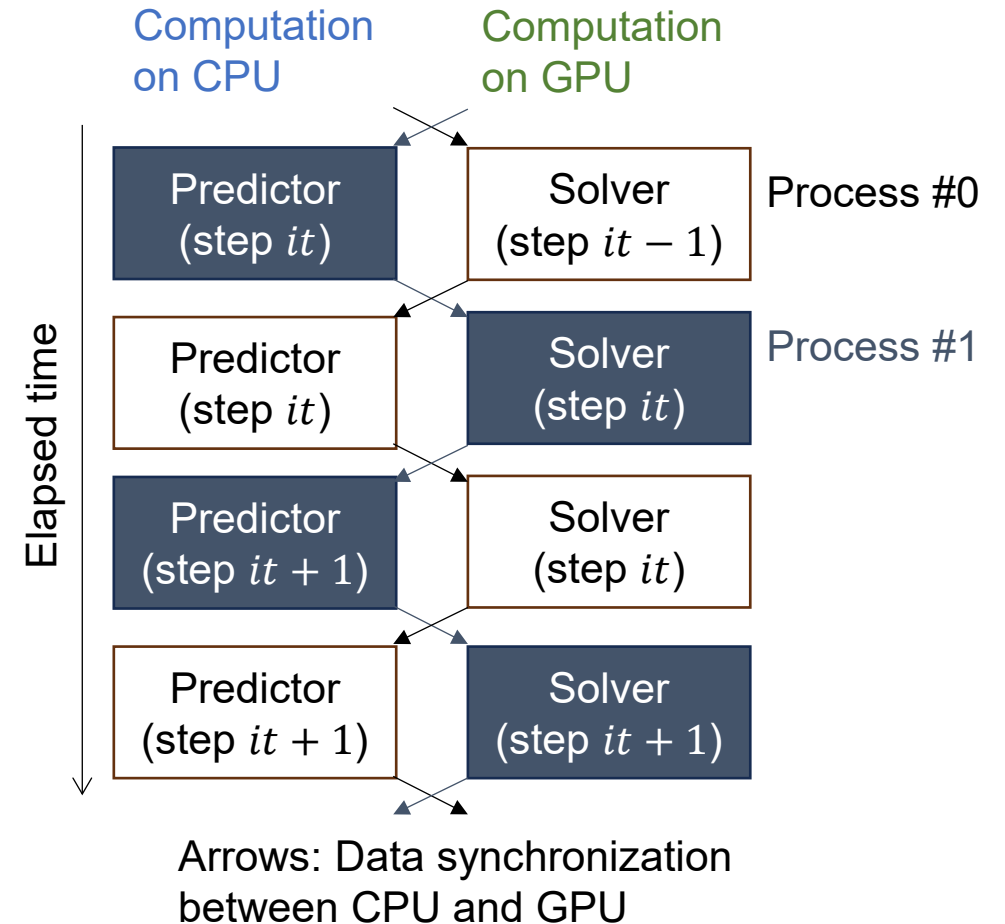
- Utilize both CPU with large memory and high-performance GPU by fast interconnect between CPU and GPU, by use of a data-driven method
- Solve linear equation  $Ax^{it} = f^{it}$  for each time step  $it$



- **Baseline solver:**
  - Use iterative solver as problem size becomes large
  - In particular, use conjugate gradient method with Compressed Row Storage (CRS) for sparse matrix-vector product computation

# Proposed concurrent CPU and GPU computing method

- Solve many cases at same time
- Use CPU to predict solution of next time step using past data
- Use GPU to iteratively solve equation
- Data synchronized before and after predictor/iterative solver using fast CPU-GPU interconnect



# Proposed concurrent CPU and GPU computing method (GPU part)

- Exploit computing power of GPU
  - Use matrix-free SpMV (sparse matrix-vector) multiplication
    - Instead of computing  $Ax^{it}$  by reading CRS-formatted  $A$  from global memory, compute  $\sum_e P_e^T (A_e (P_e x^{it}))$
    - Here  $A_e$  is element matrix and  $P_e$  is a matrix mapping global node number to element-local node number
    - While more computation is involved, memory access is reduced; thus, accelerates computation on compute-rich GPUs
  - Use of matrix-free SpMV leads to reduction in memory footprint: enables solving multiple simulation cases simultaneously using the extra memory
    - Enables converting the SpMV computation to SpMV with multiple right-hand side vectors; which are high performance as random data access is reduced

# Proposed concurrent CPU and GPU computing method (CPU part)

- Accelerate solver while ensuring accuracy by use of data-driven method@CPU

- Predict the initial solution of the iterative solver as

$$\bar{\mathbf{x}}^{it} = \text{predictor}(\mathbf{X}^{it}, \mathbf{F}^{it}, \mathbf{f}^{it}),$$

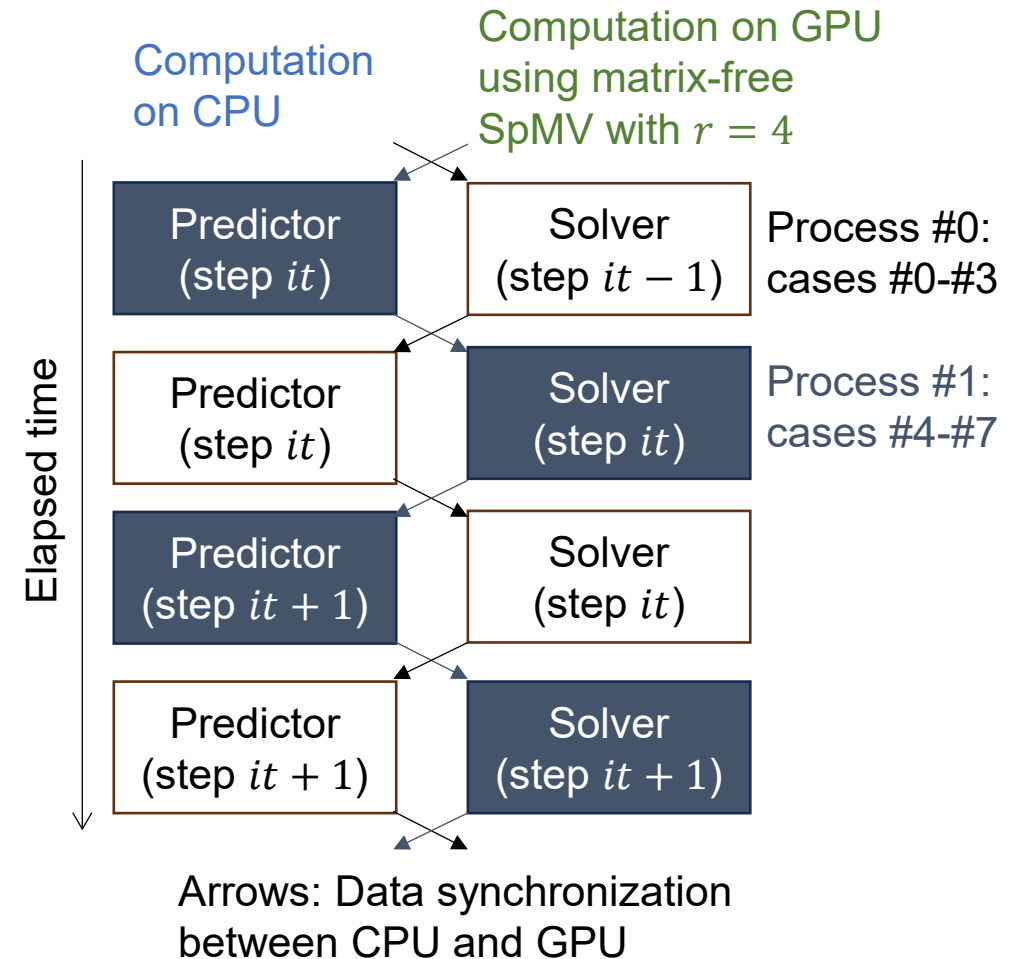
using a data-driven method using data from past  $s$  time-steps

$$\mathbf{X}^{it} = \{\mathbf{x}^{it-s}, \mathbf{x}^{it-s+1}, \dots, \mathbf{x}^{it-1}\}, \mathbf{F}^{it} = \{\mathbf{f}^{it-s}, \mathbf{f}^{it-s+1}, \dots, \mathbf{f}^{it-1}\}$$

- High accuracy initial solution leads to reduction in number of solver iterations and shortened total elapsed time
    - Compute on CPU as time-history data is too large to fit on GPU memory

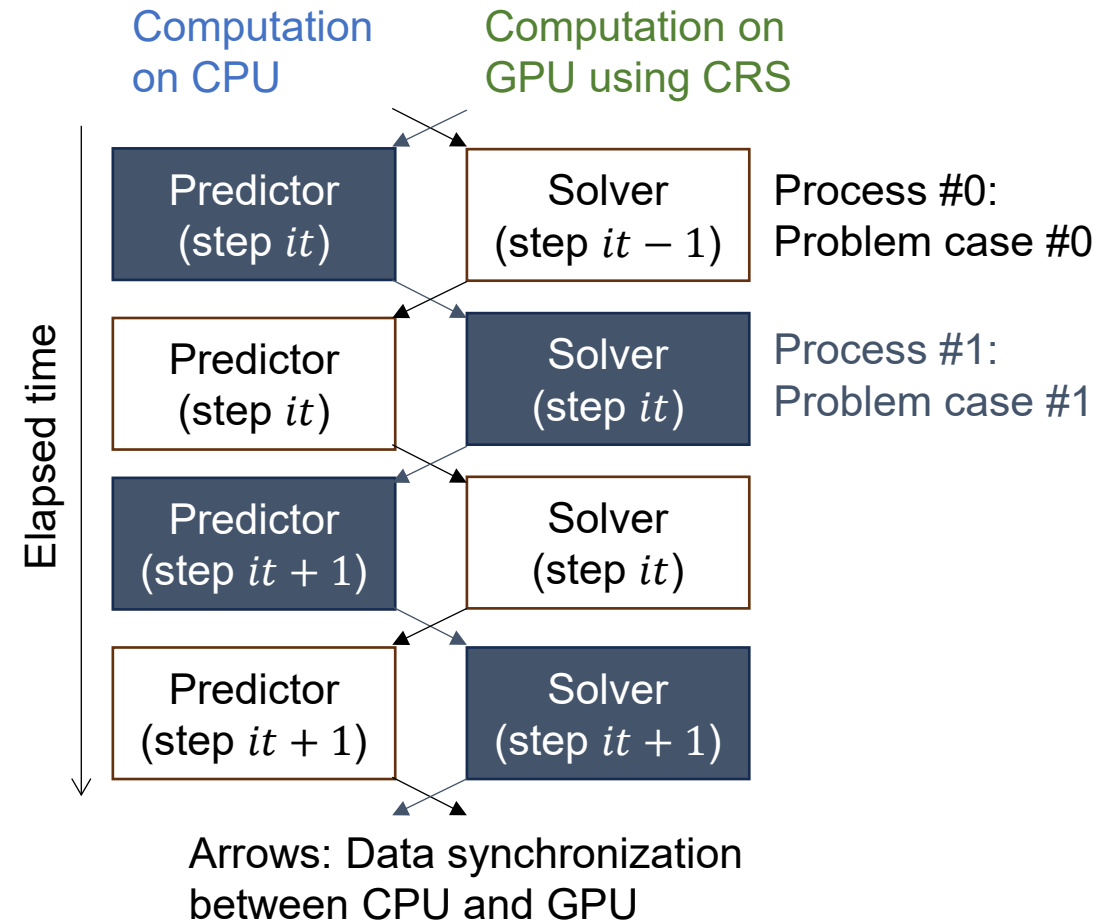
# Proposed concurrent CPU and GPU computing method

- Concurrently use CPU and GPU by solving  $2r$  sets of problems
  - CPU used for data-driven predictor for one set ( $r$  cases) of problem
  - GPU used for iterative solver for the other set ( $r$  cases) of problem
  - Data synchronized before and after predictor/iterative solver using fast CPU-GPU interconnect
- Both CPU and GPU can be fully utilized throughout the computation
  - Time-steps used in training/prediction of predictor chosen dynamically to make elapsed time of predictor@CPU and solver@GPU equal



# Proposed heterogeneous computing method in reduced form

- Matrix-free SpMV not always available for all problems
- Proposed heterogeneous computing method can be applied by solving 2 sets of problems using CRS-based solvers



# Numerical experiment: target problem

- Evaluate the performance on a linear dynamic elasticity problem

$$\rho \ddot{\mathbf{u}} - (\nabla \cdot \mathbf{c} \cdot \nabla) \cdot \mathbf{u} = \mathbf{f}$$

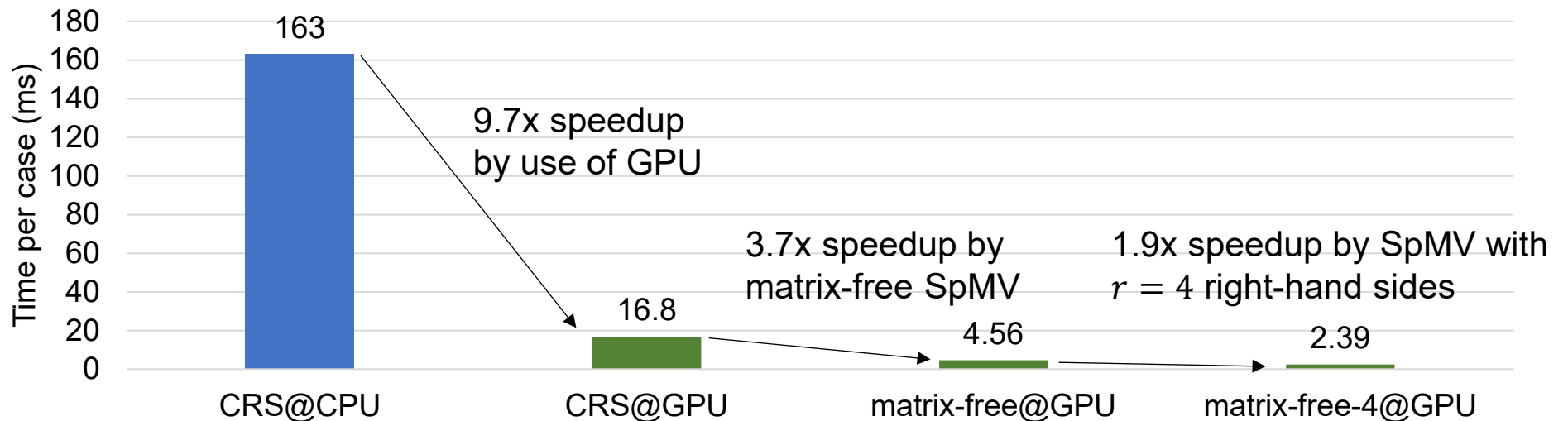
- $\mathbf{u}$ : displacement,  $\rho$ : density,  $\mathbf{c}$ : elasticity tensor,  $\mathbf{f}$ : outer force
- Discretization using second-order tetrahedral elements and Newmark- $\beta$  time integration leads to

$$\left( \frac{\mathbf{M}}{dt^2} + \frac{\mathbf{C}}{dt} + \mathbf{K} \right) \mathbf{u}^{it} = \mathbf{f}^{it} + \mathbf{C} \mathbf{v}^{it-1} + \mathbf{M} \left( \mathbf{a}^{it-1} + \frac{4}{dt} \mathbf{v}^{it-1} \right)$$

- Solve this equation for each time step  $it$  using
  - Baseline method (CPU or GPU): a conjugate gradient method using matrix-vector products based on block-CRS format
  - Proposed method: use a multi-vector matrix-free SpMV-based conjugate gradient solver with data-driven method [1] as the initial solution
  - Same preconditioner used for both methods (3x3 block Jacobi preconditioning)
  - Above methods implemented using OpenACC (GPU computing), OpenMP (CPU computing), and MPI

# Performance of cost dominant SpMV kernel

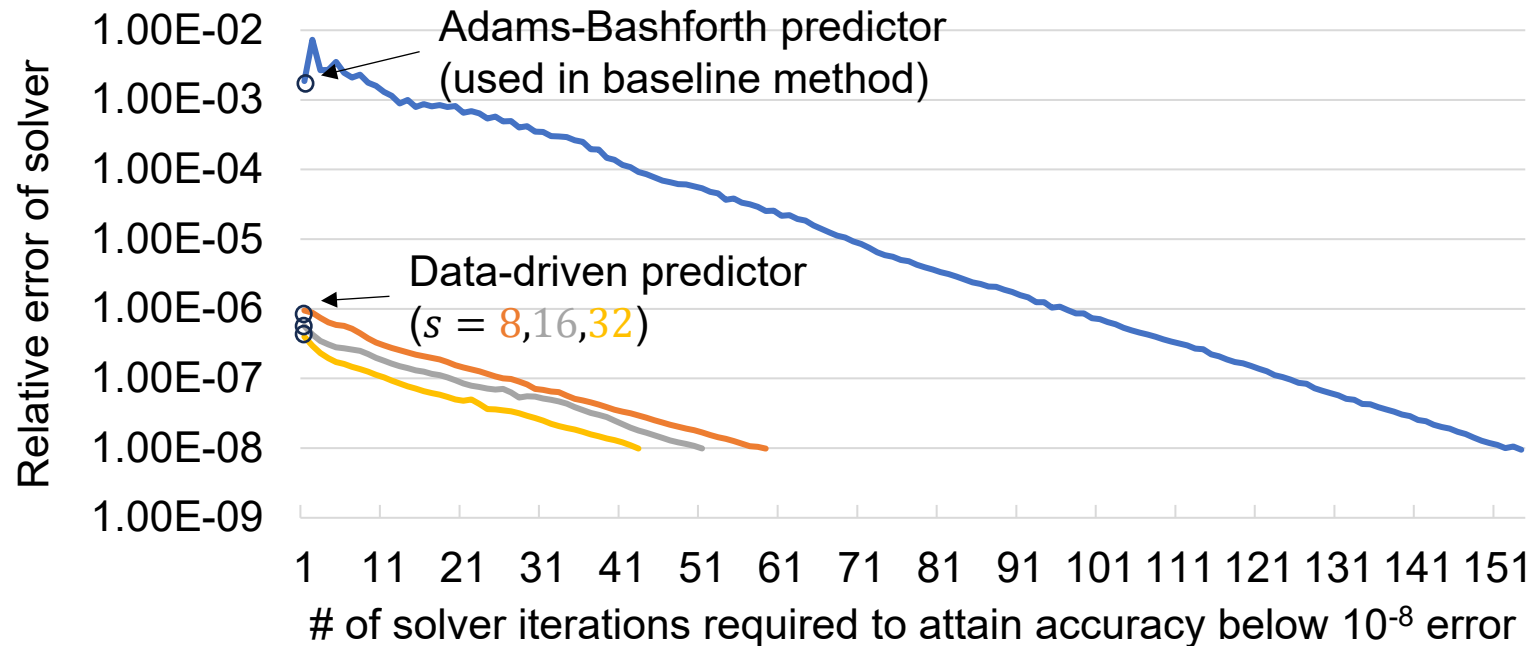
- Measure performance of a Single GH200 node
  - **CPU**: 72-core ARMv9a Grace with 480 GB (384 GB/s) memory
  - **GPU**: H100 (34 TFLOPS) with 96 GB (4000 GB/s) memory



Memory Bandwidth (% to peak)	0.21 TB/s (55%)	2.0 TB/s (51%)	0.58 TB/s (15%)	0.51 TB/s (13%)
TFLOPS (% to peak)	0.049 TFLOPS (1.4%)	0.47 TFLOPS (1.4%)	9.5 TFLOPS (28%)	18 TFLOPS (53%)

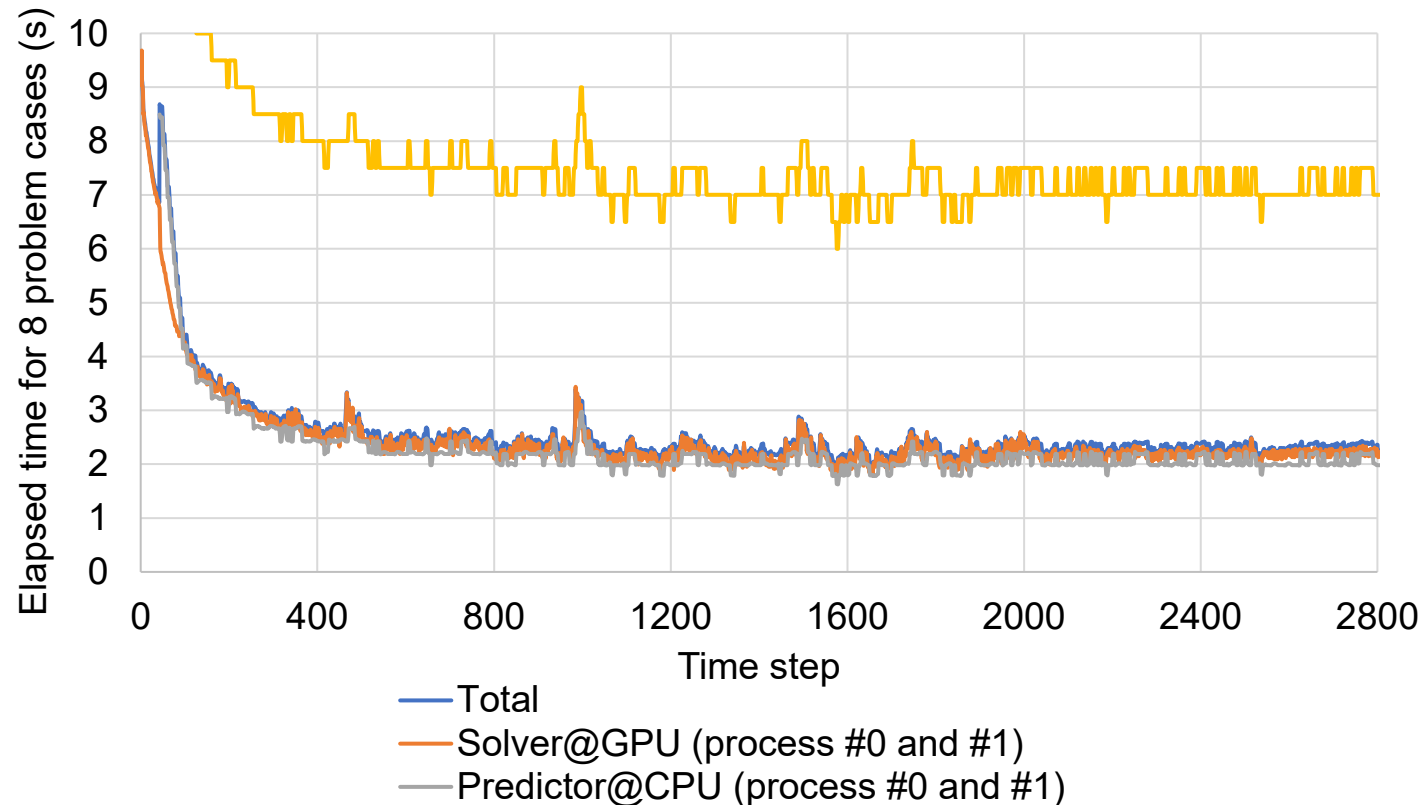
# Effectiveness of data-driven method

- Improvement in initial solution accuracy reduces number of solver iterations
  - Use of more time-steps ( $s$ ) used for the predictor leads to further reduction in number of iterations

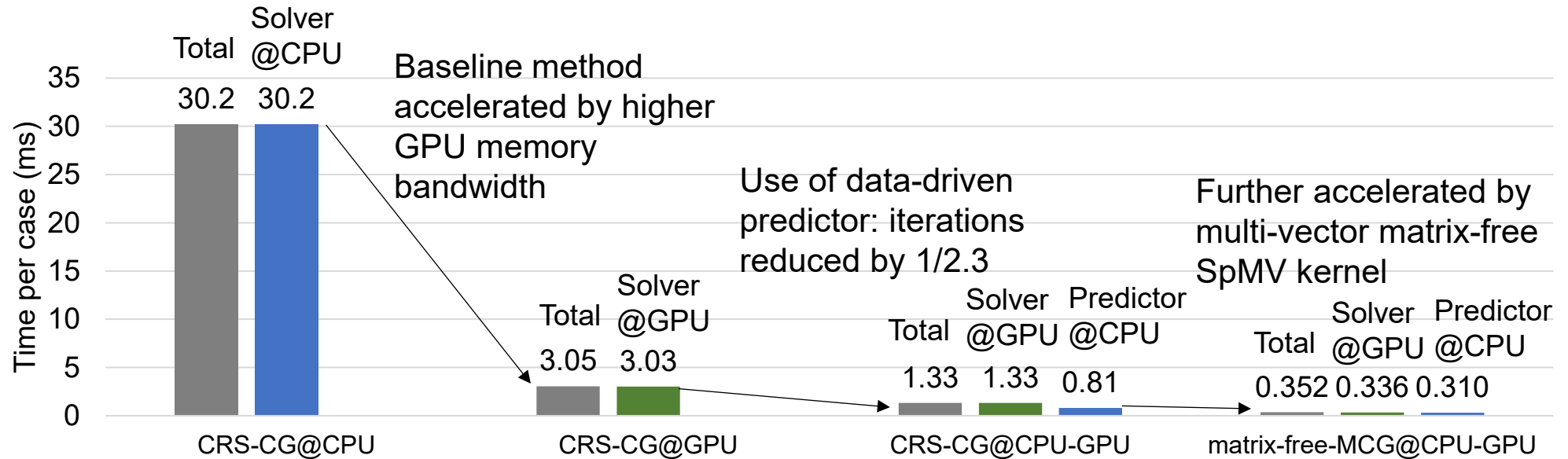


# Automatic balancing of CPU and GPU load

- Time steps  $s$  used for predictor dynamically adjusted to balance elapsed time of CPU and GPU
  - Larger  $s$ : Higher prediction accuracy, shorter GPU time but longer CPU time
  - Smaller  $s$ : Lower prediction accuracy, longer GPU time but shorter CPU time



# Application performance on a Single-GH200 node

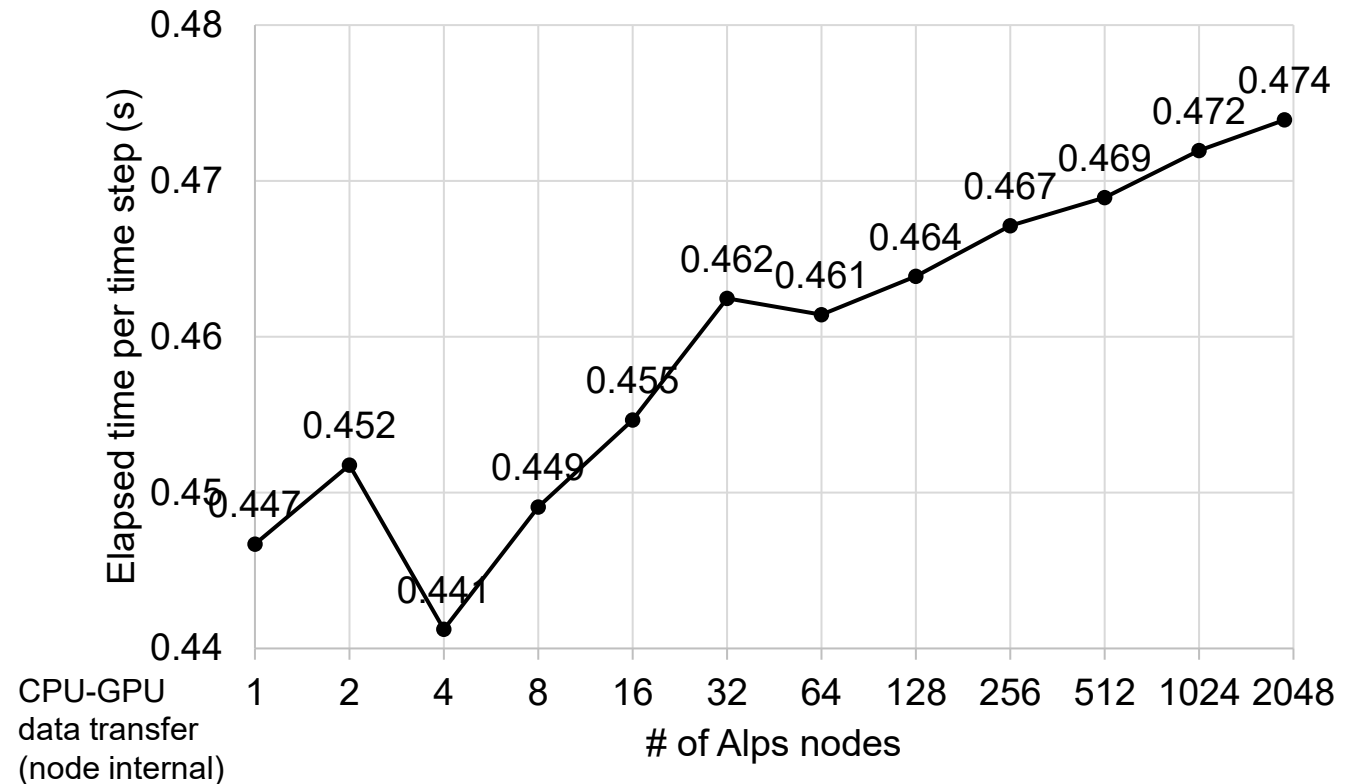
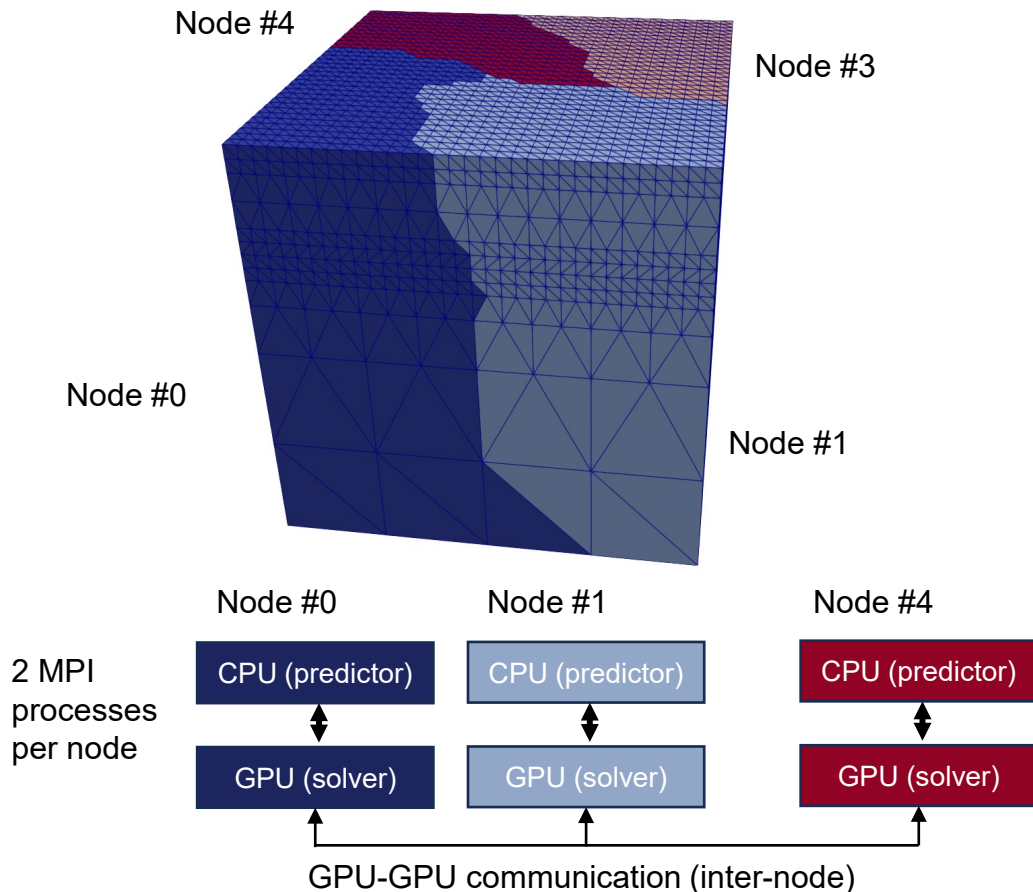


Relative speedup	1	9.96	26.1	86.4
Total module power (GPU power)	327 W (76 W)	709 W (608 W)	858 W (604 W)	877 W (652 W)
Total energy usage	9944 J	2163 J	1001 J	309 J

- 86-fold speedup and 32-fold reduction in energy from using only CPU
- 8.7-fold speedup and 7.0-fold reduction in energy from using only GPU

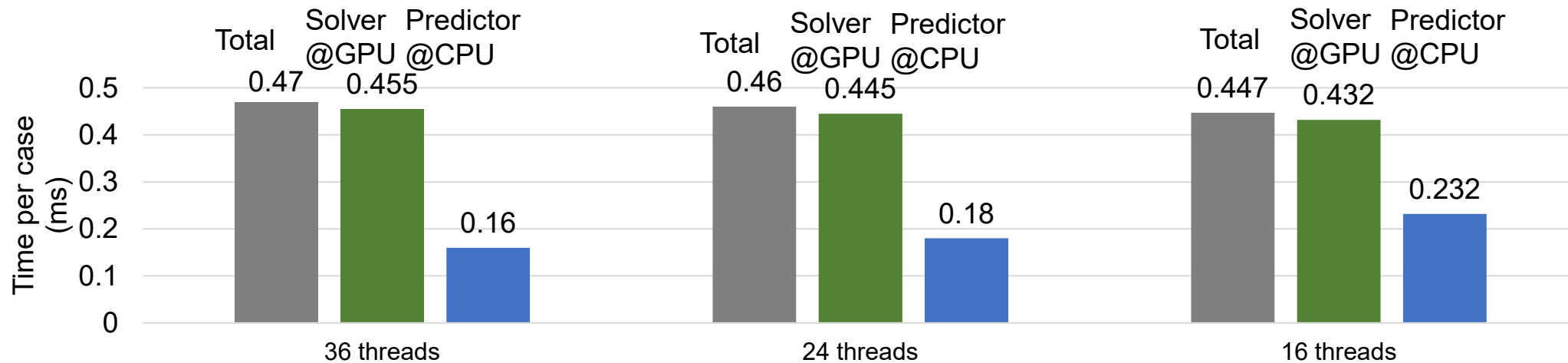
# Weak scaling on Alps@CSCS

- Scalable as data transfer between predictor and solver is internal to each node
- 94.3% weak scalability to 1920 Alps nodes (7680 GH200 modules)



# Method flexible for systems with power caps and varying memory size, CPU or GPU capabilities

- Example: On CSCS Alps system, CPU and GPU cannot be fully used simultaneously due to power cap (617 W)
  - Adjust power usage among CPU and GPU by changing number of CPU threads



Relative speedup	1.0	1.02	1.05
Relative energy usage	1.05	1.03	1.0

Reducing number of CPU threads leads to power usage on CPU spread out in time, leading to shorter elapsed time and reduced energy

# Summary of proposed algorithm and performance

- Accelerated PDE-based time-evolution problems by concurrent use of CPU and GPU
  - Utilize both large CPU memory by the data-driven method and high-performance GPU by fast interconnect between CPU and GPU
- Speedup and reduction in energy-to-solution attained without reduction in accuracy
  - 86-fold speedup and 32-fold reduction in energy from using only CPU
  - 8.7-fold speedup and 7.0-fold reduction in energy from using only GPU
- 94% weak scalability up to 1920 CSCS Alps compute nodes

# Implementation design

- Enable incremental development from flat MPI code (serial CPU code in each process)
  - Step 1: accelerate solver using GPU (OpenACC)
  - Step 2: accelerate predictor using multi-core CPU (OpenMP)
  - Step 3: enable CPU-GPU overlap by communicator splitting
- Steps 1 and 2 are the same as case without CPU-GPU overlap

# Step 1: accelerate solver using GPU (OpenACC)

- Conjugate gradient solver
  - Process-local computation
    - **Matrix-vector products** (e.g.,  $y \leftarrow Ax$ )
    - Vector updates (e.g.,  $z \leftarrow x + \alpha y$ )
    - Local inner products (e.g.,  $\alpha \leftarrow (x, y)$ )
  - Inter-process communication
    - Point-to-point communication for synchronizing matrix-vector products (MPI\_Isend/MPI\_Irecv/MPI\_Wait)
    - Global inner products (MPI\_Allreduce)
- Except for matrix-vector products, GPU computation is straightforward
  - Matrix-vector product part explained in following slides

# Step 1: GPU acceleration of matrix-vector product

- Use element-by-element method to reduce memory reads
  - Add element-wise matrix vector product in SpMV
$$Ap = \sum_e \left( P_e^T (A_e (P_e p)) \right)$$
  - Global matrix  $A$  is not stored
  - Less memory access and more computation compared to CRS

Random cache load (accessed multiple times)      DRAM load (accessed only once)

```
Ap(:) = 0
!$acc parallel loop gang vector
do ie = 1, ne
  Ae(1:m,1:m) = element_wise_matrix(params(ie))
  do j1 = 1, m
    pe(j1) = p(local_to_global(j1,ie))
  enddo
  Ape(1:m) = matrix_vector_product(Ae,pe)
  do j1 = 1, m
    Ap(local_to_global(j1,ie)) += Ape(j1)
  enddo
enddo
```

Random atomic add (accessed multiple times)

local\_to\_global: mapping from element-wise index to global index  
ne: number of elements  
m: number of degrees-of-freedom per element  
p: a vector to be multiplied by A  
Ae: element-wise matrix  
Ap: results of the matrix vector product

# Step 1: GPU acceleration of matrix-vector product

- Element-by-element method further accelerated by multiplication of multiple right-hand sides (NVEC)
  - Random access to **p** and **Ap** becomes block random accesses
  - `matrix_vector_product` becomes `matrix_matrix_product` (i.e., matrix **Ae** can be reused NVEC times)

Block random cache load  
(accessed multiple times)

```
Ap(1:NVEC,1:n) = 0
!$acc parallel loop gang vector
do ie = 1, ne
  Ae(1:m,1:m) = element_wise_matrix(params(ie))
  do j1 = 1, m
    pe(1:NVEC,j1) = p(1:NVEC,local_to_global(j1,ie))
  enddo
  Ape(1:NVEC,1:m) = matrix_matrix_product(Ae,pe)
  do j1 = 1, m
    Ap(1:NVEC,local_to_global(j1,ie)) += Ape(1:NVEC,j1)
  enddo
enddo
```

Block random atomic add (accessed multiple times)

`local_to_global`: mapping from element-wise index to global index

`ne`: number of elements

`m`: number of degrees-of-freedom per element

`p`: a vector to be multiplied by A

`Ae`: element-wise matrix

`Ap`: results of the matrix vector product

# Step 2: accelerate predictor using multi-core CPU (OpenMP)

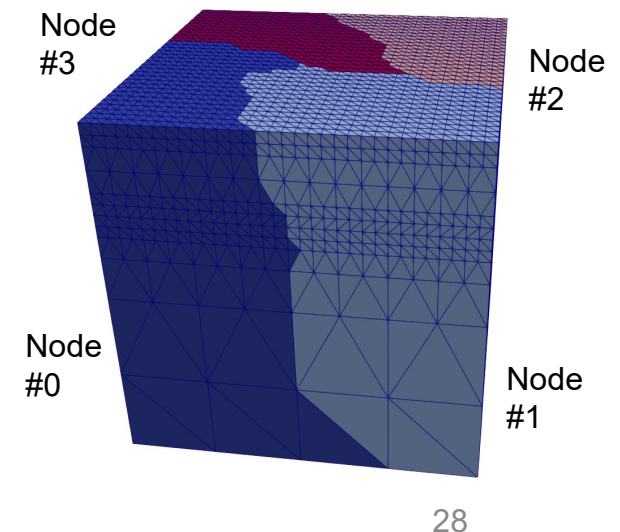
- Predictor computation
  - Partition each MPI partition into many subdomains
  - Use Gram-Schmidt orthonormalization internal to subdomain to predict solution
- Parallelize subdomain loop using OpenMP
  - Straightforward parallelization possible (no communication required between cores or processes)

# Step 3: enable CPU-GPU overlap by communicator splitting

- Use MPI\_Comm\_split to generate two communicator groups (MPI\_Comm: MPI\_COMM\_local)
  - globalrankid is even: grpid=0
  - globalrankid is odd: grpid=1
- Call solver using [MPI\_COMM\_local and localrankid] instead of [MPI\_COMM\_WORLD and globalrankid]
  - Same code can be used

Example when using four GH200 modules

MPI_COMM_WORLD:globalrankid	0	1	2	3	4	5	6	7
MPI_COMM_local:grpid	0	1	0	1	0	1	0	1
MPI_COMM_local:localrankid	0	0	1	1	2	2	3	3



# Step 3: enable CPU-GPU overlap by communicator splitting

```
int main(){
  MPI_Init(...); ...; // set globalrankid

  read_mesh(globalrankid,...); // read mesh
  ...

  for(int it=0; it < nt; ++it) // time step loop
  {
    predictor(...);

    solver(MPI_COMM_WORLD,globalrankid,...);
  }

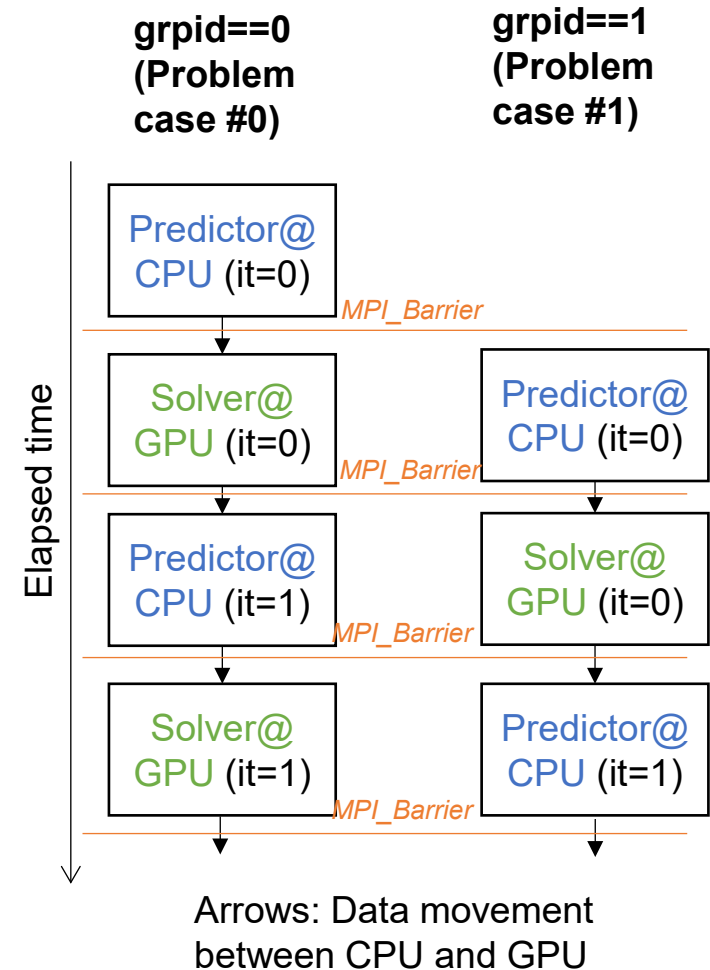
  MPI_Finalize();
}
```

Without CPU-GPU overlap:  
Launch with  $n$  processes



```
int main(){
  MPI_Init(...); ...; // set globalrankid
  MPI_Comm_split(...); // set localrankid, grpid
  read_mesh(localrankid,...); // read mesh
  ...
  if(grpid==1) MPI_Barrier(MPI_COMM_WORLD);
  for(int it=0; it < nt; ++it) // time step loop
  {
    predictor(...);
    MPI_Barrier(MPI_COMM_WORLD);
    solver(MPI_COMM_local,localrankid,...);
    MPI_Barrier(MPI_COMM_WORLD);
  }
  if(grpid==0) MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
}
```

With CPU-GPU overlap:  
Launch with  $2n$  processes  
(same executable used for all ranks)



# Summary and future work

- Accelerated PDE-based time-evolution problems by concurrent use of CPU and GPU
  - Utilize both large CPU memory by the data-driven method and high-performance GPU by fast interconnect between CPU and GPU
- Speedup and reduction in energy-to-solution attained without reduction in accuracy
  - 86-fold speedup and 32-fold reduction in energy from using only CPU
  - 8.7-fold speedup and 7.0-fold reduction in energy from using only GPU
- 94% weak scalability up to 1920 CSCS Alps compute nodes
- Step-by-step implementation possible using OpenACC/OpenMP/MPI
- Use of sophisticated AI methods in data-driven part expected to improve performance and applicability to wider range of problems

# Acknowledgements

- This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) on Alps
- We thank Yukihiro Hirano (NVIDIA) for coordination of the collaborative research project
- This work was supported by JSPS KAKENHI Grant Numbers 23H00213, 22K18823
- This work was supported by JST SPRING, Grant Number JPMJSP2108