Nov. 14, 2021

WACCPD 2021: Eighth Workshop on Accelerator Programming using Directives

GPU porting of scalable implicit solver with Green's function-based neural networks by OpenACC

Kohei Fujita[1,2], Yuma Kikuchi[1], Tsuyoshi Ichimura[1,2], Muneo Hori[3], Lalith Maddegedara[1], Naonori Ueda[2]

1. The University of Tokyo, 2. RIKEN, 3. Japan Agency for Marine-Earth Science and Technology

Introduction

- Variety of computer architecture and HPC applications increasing
  - Many architectures, e.g., x86/Arm/Power CPUs & NVIDIA/AMD/Intel GPUs
  - New types of applications that combine equation-based methods & data-driven methods
- Challenging to attain application performance on multiple systems with low code development cost
- Directive-based parallel programming models developed for performance portability across systems
  - For example, many equation-based applications ported using OpenACC
  - However, only few porting examples for applications combining equation-based methods & data-driven methods

Aim of this study

- Show effectiveness of directive-based parallel programming models for applications combining equation-based methods & data-driven methods
  - Port a partial differential equation (PDE) solver accelerated by data-driven method: neural network (NN) is used as a preconditioner to accelerate an implicit solver
  - Use OpenACC to port CPU code to GPU: Compare performance with CPU implementation & CUDA-based GPU implementation to show effectiveness of directive-based porting

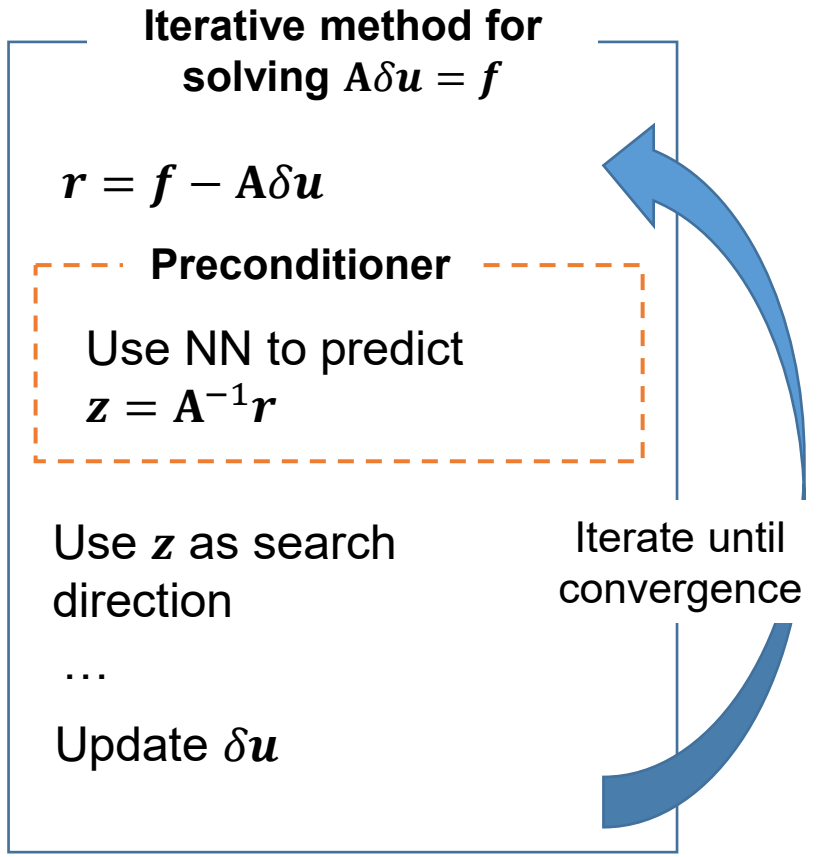
Target application

- Partial differential equation (PDE) solver accelerated by neural network (NN)-based preconditioner
  - Solves 3D wave equation ($\rho \frac{\partial^2 u_i}{\partial t^2} = \frac{\partial}{\partial x_j}\left(c_{ijkl}\frac{\partial u_k}{\partial x_l}\right) + f_i$) using implicit voxel finite-element method: leads to solving $\mathbf{A}\delta\boldsymbol{u} = \boldsymbol{f}$ for each time step
  - NNs are used via *Green's functions* that reflect property of the PDE for cost efficient preconditioner
  - Originally developed for CPU-based systems; attained high performance on Arm-based Fugaku and Xeon-based systems
  - Operations localized for high-peak performance and scalability – high performance also expected on large-scale GPU systems

Tsuyoshi Ichimura, Kohei Fujita, Muneo Hori, Lalith Maddegedara, Naonori Ueda, Yuma Kikuchi, A Fast Scalable Iterative Implicit Solver with Green's function-based Neural Networks, *ScalA20: 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2020.

Using NN for solving PDEs

- Using NN as a preconditioner in PDE solution schemes
  - Accuracy of final solution assured
  - Refinement rate dependent on accuracy of NN estimation
- However, constructing high-accuracy NNs for large-scale problems within reasonable cost is challenging
  - For small scale problems, accurate solution can be obtained by using NNs trained on data (pairs of $\mathbf{x}, \mathbf{Ax}$) that cover the modes of the solution
  - However, data for only a few modes can be stored for large-scale problems (accuracy low for its required cost)
- We designed a NN-based preconditioner via use of *Green's functions* that reflect property of the PDE with localized expansions
  - Resolves the problem of direct construction of NNs for large-scale problems



Green's functions

- Function used to solve target equation $L\,u = b$
  - For linear operator L, Green's function g is the solution of $L\,g = \delta$, where δ is Dirac's delta function
  - Solution of $L\,u = b$ can be obtained as convolution of g (i.e., $u = g * b$)
- Example for solving discretized form of 1D wave equation
  - Target equation:

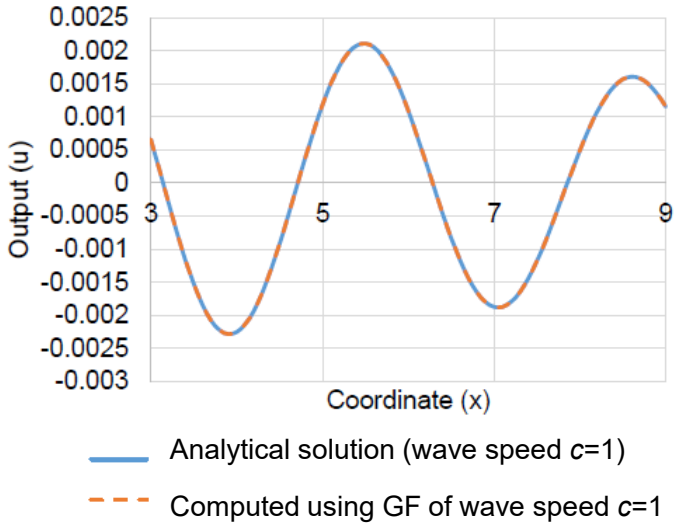
$$\frac{\partial^2 u(x)}{\partial x^2} - \frac{4}{c^2 dt^2} u(x) = b(x)$$

  - Green's function:

$$g(s) = -\frac{c\,dt}{4} e^{-\frac{2\sqrt{s^2}}{c\,dt}}$$

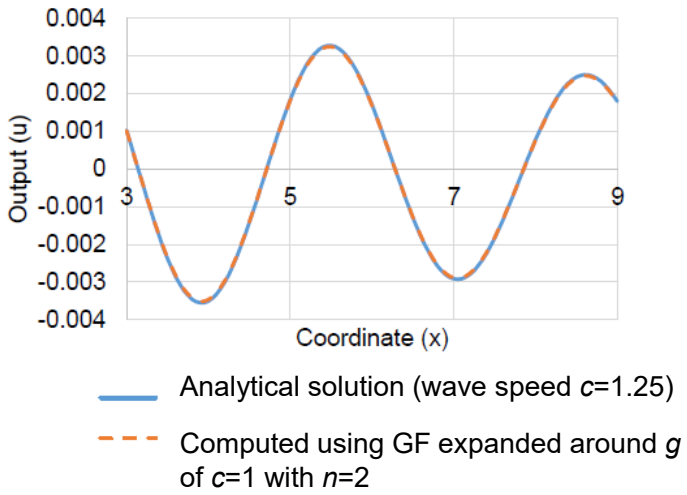
  - Solution of target equation can be obtained as:

$$u(x) = \int g(s-x)b(s)ds$$



Reducing evaluation cost of Green's functions

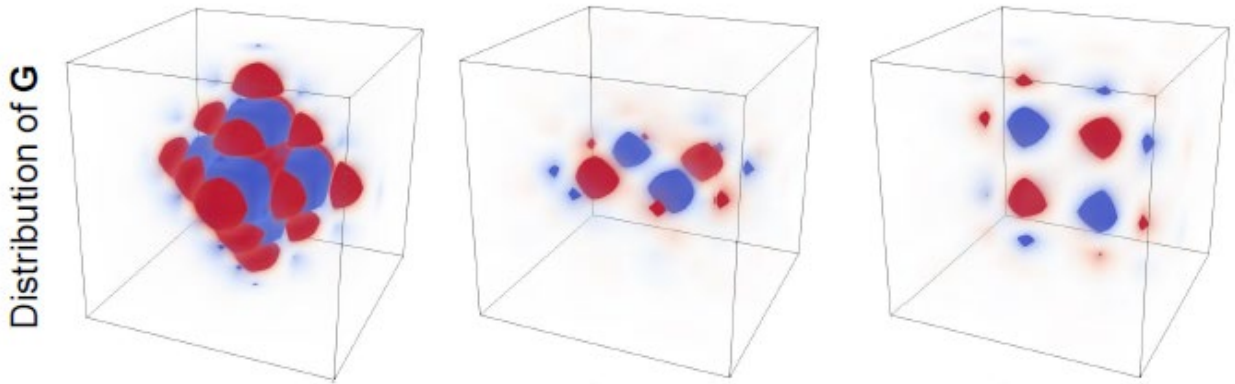
- While accurate solutions can be obtained via Green's functions (GF), its evaluation cost is very large
- Generate NNs that approximate GF to reduce evaluation cost
  - Expand g with respect to g of a uniform problem: $g(s) = g(s)^{base} \sum_{i=1}^{n} a_i \left(\sqrt{s^2}\right)^{i-1}$
  - Use NN to estimate coefficients a_i



Highly accurate solution obtained by only estimating few parameters

Green's function (GF)-based NNs: 3D case

- High-frequency modes included in GF of 3D wave propagation problem
  - Large amount of data required for direct estimation of high-frequency modes
- Evaluate GF of heterogeneous problems by expanding GF around precomputed GF for homogeneous problem
  - Train NNs that evaluate the expansion coefficients
  - Enables generation of high-accuracy GF with limited amount of data



Examples of distributions of GF

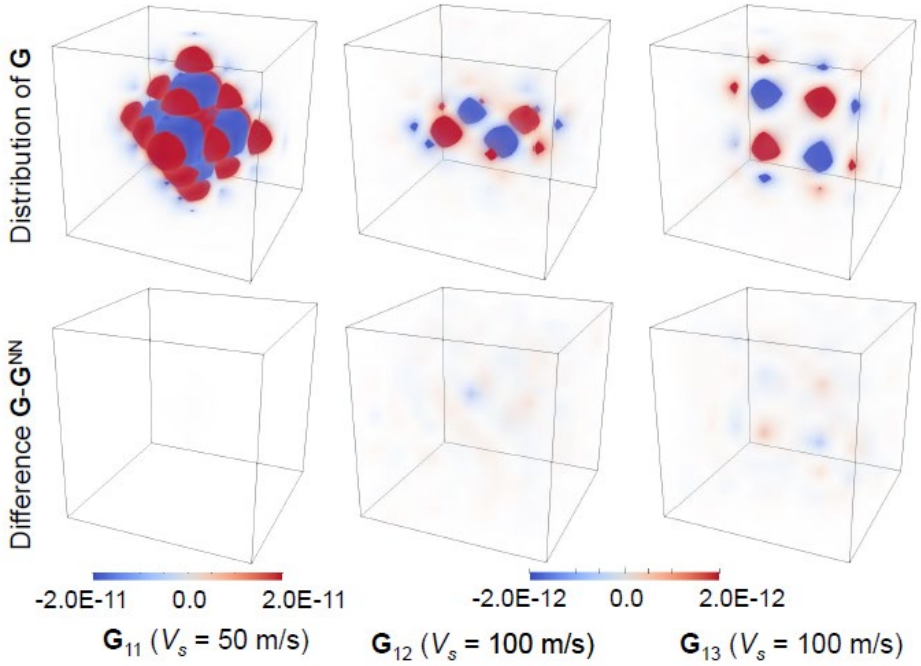
$$G_{ii}(x,y,z) = G_{ii}(x,y,z)^{base}(c_1 + c_2x + c_3y + c_4z)$$
for $i = 1,2,3$

$$G_{ij}(x,y,z) = G_{ij}(x,y,z)^{base}(c_5 + c_6x + c_7y + c_8z)$$
for $i,j = 1,2,3\ (i \neq j)$

Expansion of GF with 8 coefficients $\boldsymbol{c_1, c_2, \dots, c_8}$
Use classifier NNs (input: 216 material properties of 6 x 6 x 6 element domain)

Estimation performance of Green's function-based NNs

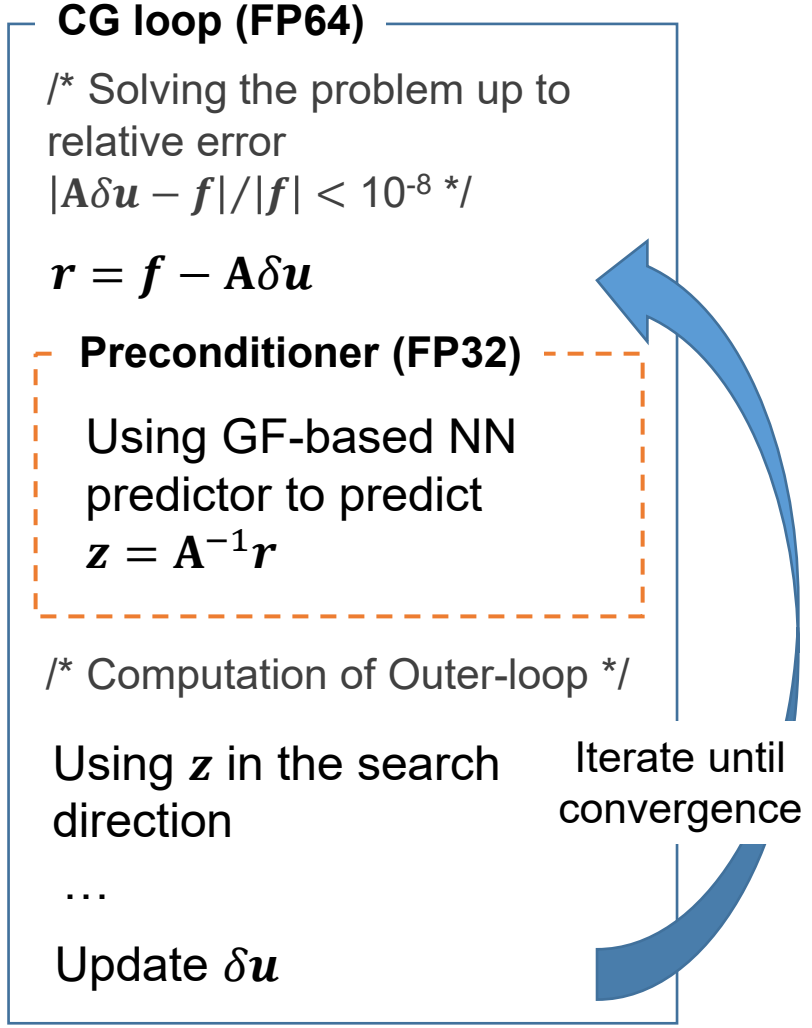
- High-frequency modes are resolved in high accuracy with only a few parameters
- Enables generating NNs with high accuracy with relatively small datasets and low training cost
- Low inferencing cost as a shallow network is used



Evaluation of *g* expanded around *g* of *Vs* = 50 m/s

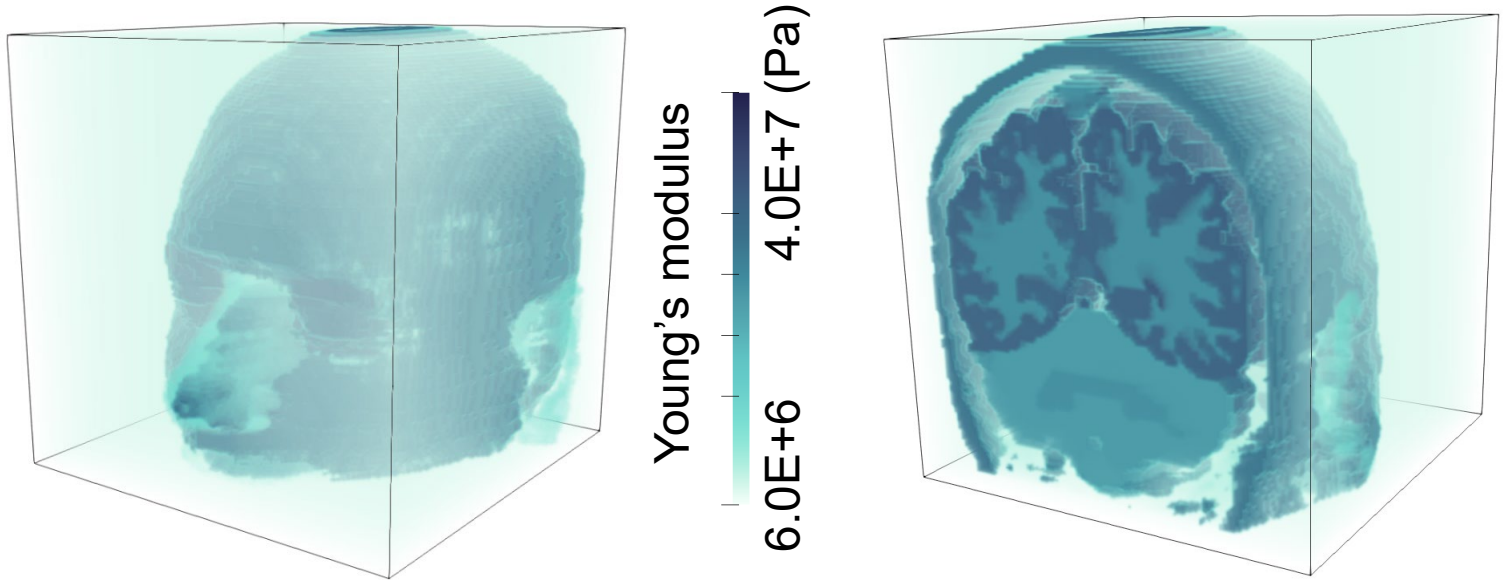
Implicit solver algorithm using GF-based NNs

- Use GF-based NNs in preconditioner of conjugate gradient solver
  - Accuracy of final solution guaranteed as NN is only used as a preconditioner
- Computation pattern of standard PDE-based solvers (random data access with sparse computation) is converted to that of NNs (sequential data access with dense computation)
  - Expected to lead to fast and efficient preconditioning of iterative solvers



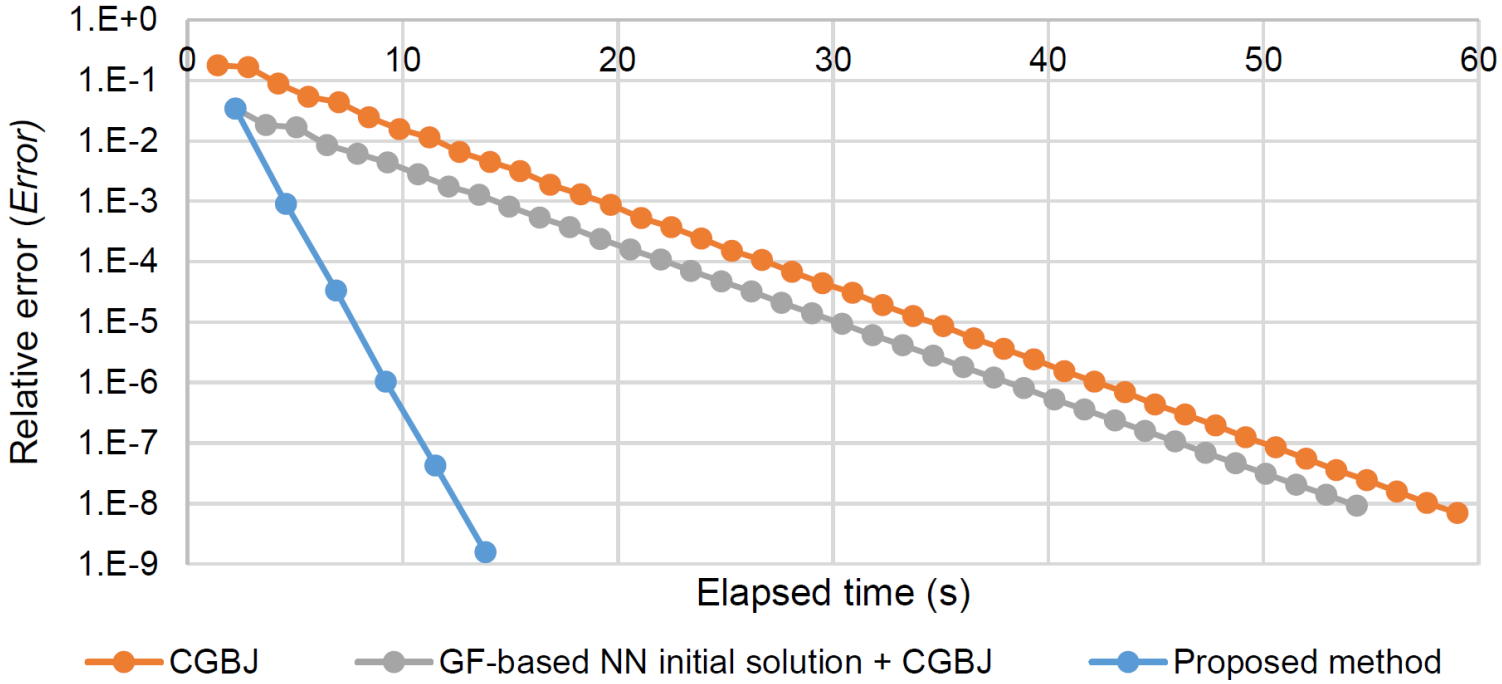
Performance on CPU-based system

- Measure performance for computing wave propagation in a human head model
  - Highly heterogeneous problem with sharp material interfaces
  - Secondary wave speed: *Vs* = 50 to 120 m/s with void part filled with a soft material with *Vs* = 50 m/s
  - Fixed primary wave speed (*Vp* = 200 m/s), density (ρ = 1000 kg/m^3) and damping (*h* = 0.001)
  - Discretized using voxel finite-elements with Newmark-β time integration



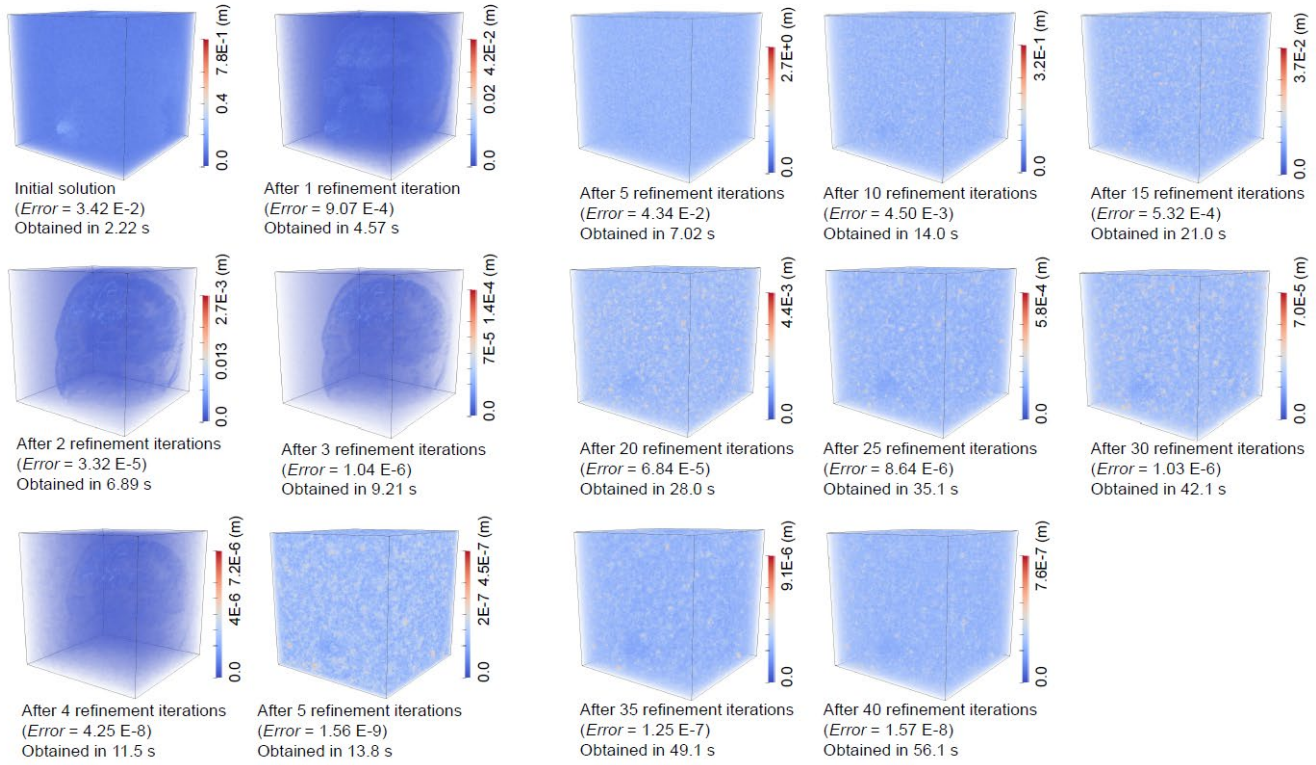
Performance on CPU-based system

- Compare performance with standard conjugate gradient solver (CGBJ: conjugate gradient solver with 3x3 block Jacobi preconditioning)
- By use of Green's function-based NN, the number of iterations reduced by 42/5 = 8.4-fold
- Together with high performance of 24.1% of FP64 peak, 4.26-fold speed up attained from fine-tuned CGBJ on Cascade Lake Xeon-based Oakbridge-CX (Univ. of Tokyo)



Distribution of residual

- Although residual is reduced from regions with uniform material properties (i.e., regions with high NN accuracy), estimation accuracy is also high for heterogeneous regions, leading to 8-fold reduction in number of iterations



Summary of target application

- We target porting of application combining equation-based and data-driven methods by directive-based methods
- Here, we target an implicit PDE solver with NN-based preconditioner
  - Green's functions reflecting the underlying PDE is used to gain accuracy of NN within low cost, leading to 8-fold reduction in number of iterations and 4-fold speedup from standard equation-based solver on Xeon CPU-based system
  - As the NN-based computation is localized and computation cost is uniform, high peak performance/scalability and short time-to-solution also expected on other computer architecture
- In the latter half of presentation, we show GPU porting and measure computational performance on GPU systems

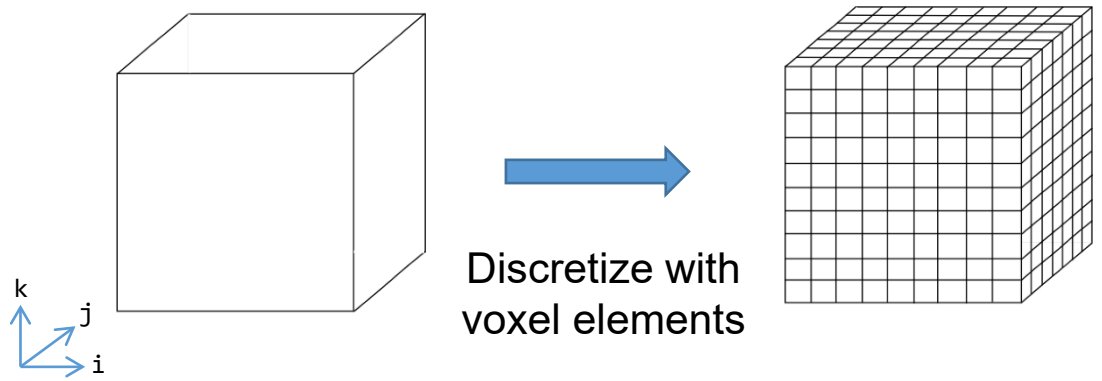
GPU porting of the solver using OpenACC

GPU porting of the solver using OpenACC

- Developed solver comprises data-driven modeling part (GF-based NN-predictor) and equation-based modeling part
  - expected perform well on many architectures including GPUs because of its dense NN computation and structured data access
  - However, appropriate algorithm selection and implementation are necessary at the kernel level
- All computation is offloaded to GPU, but the GF-based NN-predictor and matrix-vector product (EBE) are particularly computationally heavy part, so we focused on tuning them
- In addition, the process layout is optimized for the node configuration of multiple GPUs and multiple communication ports

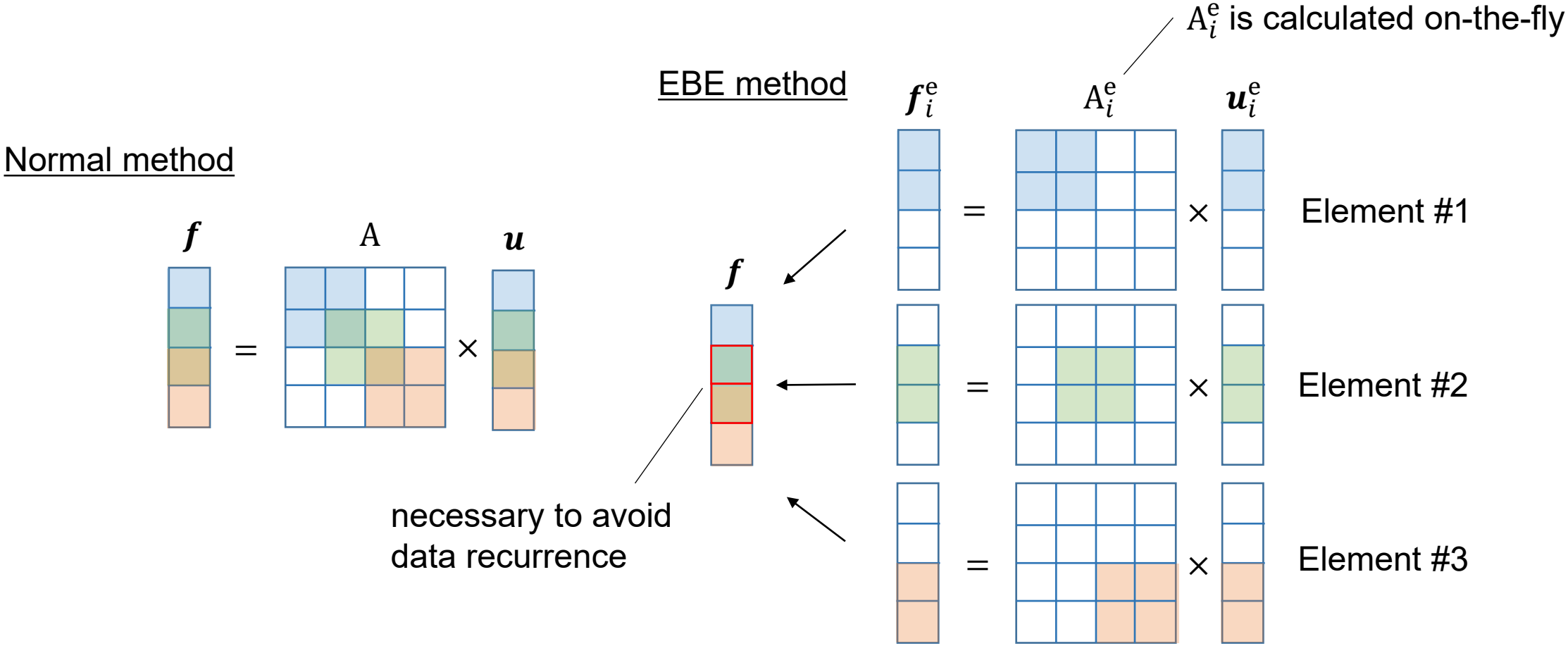
Voxel finite-element method

- Solves 3D wave equation $\rho \frac{\partial^2 u_i}{\partial t^2} = \frac{\partial}{\partial x_j}\left(c_{ijkl} \frac{\partial u_k}{\partial x_l}\right) + f_i$
- We use implicit voxel finite-element method to discretize the domain
- Continuous memory access is increased, and memory required to store the information of element is significantly reduced compared to the case of using tetrahedral elements for discretization



Porting of EBE kernel - about Element-By-Element method

- EBE method performs the matrix-vector product $\boldsymbol{f} = \mathrm{A}\boldsymbol{u}$ element wise like $\boldsymbol{f} \leftarrow \sum \boldsymbol{f}_i^{\mathrm{e}} = \sum \mathrm{A}_i^{\mathrm{e}} \boldsymbol{u}_i^{\mathrm{e}}$

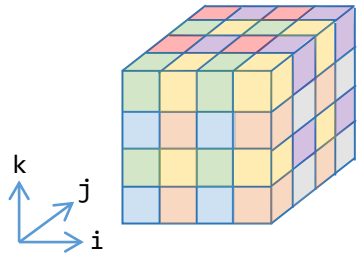


Porting of EBE kernel - using coloring algorithm

- It is necessary to avoid data recurrence when adding to left-side vector
- One of the ways is coloring the elements into eight different colors
  - Simple and can be used on GPUs running many threads
  - But it results in stride 2 data access, leading to performance deterioration

(Original)

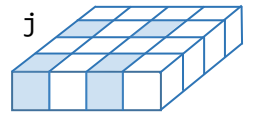
```
! Loop for element
do k=1,nez
do j=1,ney
do i=1,nex
! Compute BDBu
  …

! Data recurrence happens
  q(i,j,k,1)=q(i,j,k,1)+BDBu11
  q(i,j,k,2)=q(i,j,k,2)+BDBu12
  q(i,j,k,3)=q(i,j,k,3)+BDBu13
  q(i+1,j,k,1)=q(i+1,j,k,1)+BDBu21
  q(i+1,j,k,2)=q(i+1,j,k,2)+BDBu22
  q(i+1,j,k,3)=q(i+1,j,k,3)+BDBu23
  …
  q(i+1,j+1,k+1,1)=q(i+1,j+1,k+1,1)+BDBu81
  q(i+1,j+1,k+1,2)=q(i+1,j+1,k+1,2)+BDBu82
  q(i+1,j+1,k+1,3)=q(i+1,j+1,k+1,3)+BDBu83
enddo
enddo
enddo
```



(Coloring)

```
!$acc parallel loop collapse(3)
! Loop for element
do k=1,nez,2
do j=1,ney,2
do i=1,nex,2
! Compute BDBu
  …

  q(i,j,k,1)=q(i,j,k,1)+BDBu11
  q(i,j,k,2)=q(i,j,k,2)+BDBu12
  q(i,j,k,3)=q(i,j,k,3)+BDBu13
  q(i+1,j,k,1)=q(i+1,j,k,1)+BDBu21
  q(i+1,j,k,2)=q(i+1,j,k,2)+BDBu22
  q(i+1,j,k,3)=q(i+1,j,k,3)+BDBu23
  …
  q(i+1,j+1,k+1,1)=q(i+1,j+1,k+1,1)+BDBu81
  q(i+1,j+1,k+1,2)=q(i+1,j+1,k+1,2)+BDBu82
  q(i+1,j+1,k+1,3)=q(i+1,j+1,k+1,3)+BDBu83
enddo
enddo
enddo
!$acc end parallel
```

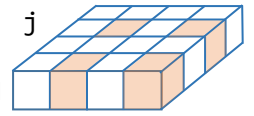
SIMT computation



color #1

```
!$acc parallel loop collapse(3)
do k=1,nez,2
do j=1,ney,2
do i=2,nex,2
 …
enddo
enddo
enddo
!$acc end parallel
```

SIMT computation

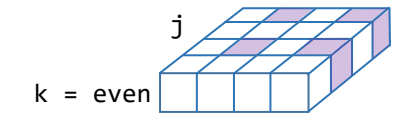


color #2

…

```
!$acc parallel loop collapse(3)
do k=2,nez,2
do j=2,ney,2
do i=2,nex,2
 …
enddo
enddo
enddo
!$acc end parallel
```

SIMT computation



color #8

Porting of EBE kernel - using fast atomics

- Recent NVIDIA GPUs have high throughput hardware-accelerated atomics
- We can expect performance improvement by avoiding data recurrence using this feature
- By utilizing *atomic add* function, stride 2 data access in the coloring algorithm is replaced by continuous access, leading to performance improvement

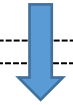
```
!$acc parallel loop collapse(3)
! Loop for element
do k=1,nez                                   SIMT computation
do j=1,ney
do i=1,nex
! Compute BDBu
  …

!$acc atomic add
  q(i,j,k,1)=q(i,j,k,1)+BDBu11
!$acc atomic add
  q(i,j,k,2)=q(i,j,k,2)+BDBu12
!$acc atomic add
  q(i,j,k,3)=q(i,j,k,3)+BDBu13
!$acc atomic add
  q(i+1,j,k,1)=q(i+1,j,k,1)+BDBu21
!$acc atomic add
  q(i+1,j,k,2)=q(i+1,j,k,2)+BDBu22
!$acc atomic add
  q(i+1,j,k,3)=q(i+1,j,k,3)+BDBu23
  …
!$acc atomic add
  q(i+1,j+1,k+1,1)=q(i+1,j+1,k+1,1)+BDBu81
!$acc atomic add
  q(i+1,j+1,k+1,2)=q(i+1,j+1,k+1,2)+BDBu82
!$acc atomic add
  q(i+1,j+1,k+1,3)=q(i+1,j+1,k+1,3)+BDBu83
enddo
enddo
enddo
!$acc end parallel
```

Porting of EBE kernel - element wise computation

- Recent GPUs have large number of registers, good performance is expected by reducing data access even if on register computation is increased
- The kernel algorithm that calculated intermediate variables in advance are changed such that these variables recalculated element wise
  - reduces the amount of GPU memory read, which is expected to lead to speedup

```
! calculation of intermediate variables Bx
B1=…
B2=…
…
B8=…

do k=1,nez
do j=1,ney
do i=1,nex
…
! computation using Bx
  …
enddo
enddo
enddo
```

Before



After

```
do k=1,nez
do j=1,ney
do i=1,nex
…
! on-the-fly calculation of Bx
  B1=…
  B2=…
  …
  B8=…

! computation using Bx
  …
enddo
enddo
enddo
```

Porting of GF-based NN-predictor

- This kernel computes the convolution of the information of the surrounding nodes into the center node
- The outer triple loop is collapsed to effectively utilize many threads on GPUs
- Array dimensions are rearranged to make all data accesses coalesced
- By unrolling the outermost loop for k, the reads in the innermost loop can be reused for further performance increase

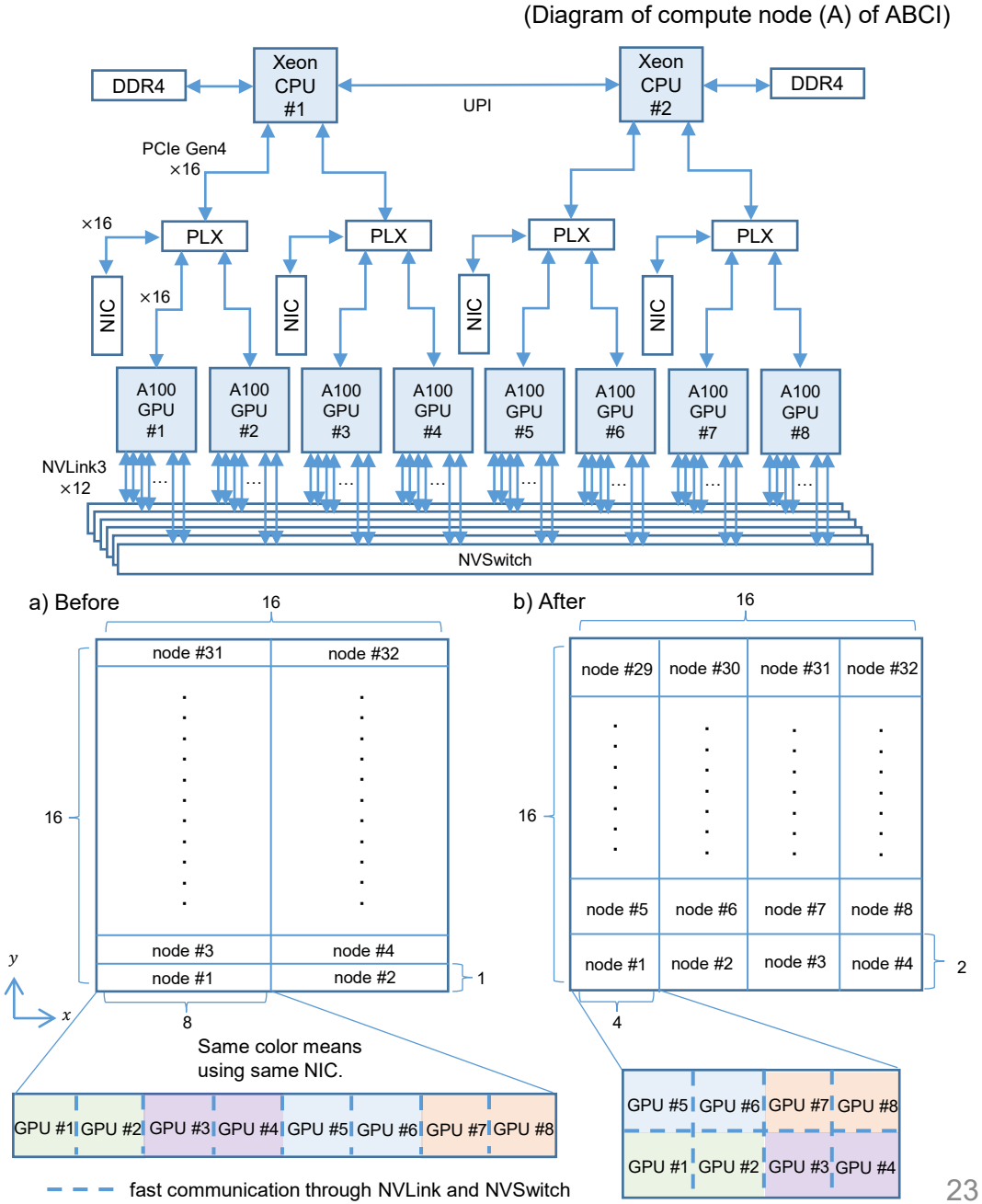
```
!$acc parallel loop collapse(3)
do k=1+nef-1,nez+1-nef+1
do j=1+nef-1,ney+1-nef+1
do i=1+nef-1,nex+1-nef+1
  we1=wei(i,j,k,1)
  we2=wei(i,j,k,2)
  …
  we8=wei(i,j,k,8)

  grs1=0.0
  grs2=0.0
  grs3=0.0
!$acc loop seq
  do k1=1,nd*2+1
!$acc loop seq
  do i1=1,nd*2+1
!$acc loop seq
  do j1=1,nd*2+1
      …
  enddo
  enddo
  enddo
  zs(i,j,k,1)=grs1
  zs(i,j,k,2)=grs2
  zs(i,j,k,3)=grs3
enddo
enddo
enddo
!$acc end parallel
```

SIMT computation

```
rs1=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,1)
rs2=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,2)
rs3=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,3)
cocs1=cocs(i1,j1,k1,1)
cocs2=cocs(i1,j1,k1,2)
cocs3=cocs(i1,j1,k1,3)
ww1=we1+we2*cocs1+we3*cocs2+we4*cocs3
ww2=we5+we6*cocs1+we7*cocs2+we8*cocs3
grs1=grs1+rs1*ww1*coe1s(i1,j1,k1,1)
grs1=grs1+rs2*ww2*coe1s(i1,j1,k1,2)
grs1=grs1+rs3*ww2*coe1s(i1,j1,k1,3)
grs2=grs2+rs1*ww2*coe2s(i1,j1,k1,1)
grs2=grs2+rs2*ww1*coe2s(i1,j1,k1,2)
grs2=grs2+rs3*ww2*coe2s(i1,j1,k1,3)
grs3=grs3+rs1*ww2*coe3s(i1,j1,k1,1)
grs3=grs3+rs2*ww2*coe3s(i1,j1,k1,2)
grs3=grs3+rs3*ww1*coe3s(i1,j1,k1,3)
```

Mapping of process for efficient communication

- Recent GPU based compute nodes are often equipped with multiple GPUs and multiple communication ports with non-uniform latency and bandwidth
- It is important to allocate process and communication ports according to these configuration
- We arranged the process mapping in such a way that the communication ports close to each GPU are used and fast communication through NVLink and NVSwitch within node is increased



Performance measurement environment

- We measured performance on AI Bridging Cloud Infrastructure (ABCI, 12th in TOP 500, June 2021) at National Institute of Advanced Industrial Science and Technology (AIST)
- Compare performance between CPUs and GPUs on one node: 2 CPUs for CPU measurement and 8 GPUs for GPU measurement

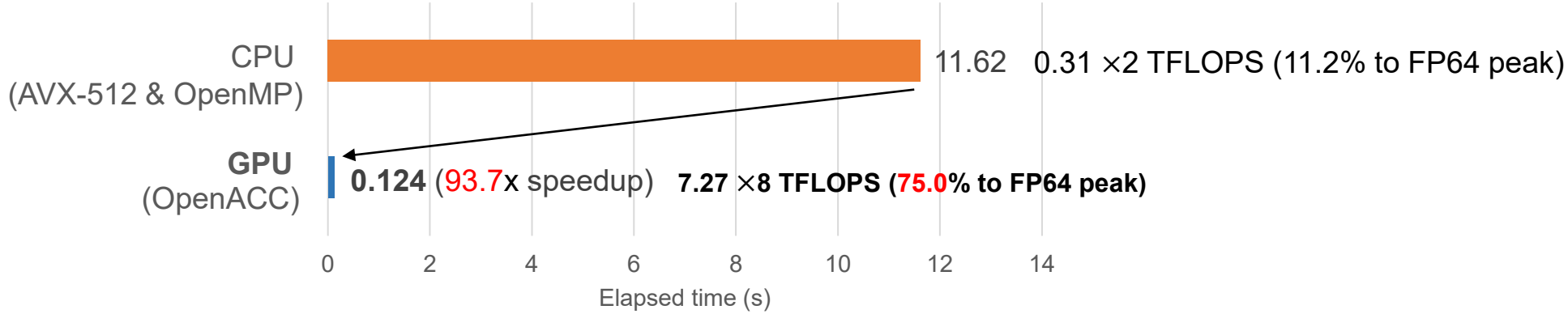
	Compute node (A)	Hardware peak per node
CPU	Intel Xeon Platinum 8360Y (2.40 GHz, 36 Cores) × 2	2.764 × 2 = 5.529 TFLOPS
GPU	NVIDIA A100 NVLink 40 GB HBM2 × 8	9.7 × 8 = 77.6 TFLOPS 1.37 × 8 = 12.4 TB/s
Interconnect	Infiniband HDR (200 Gbps) × 4	100 GB/s

- The FP64 peak performance ratio between CPU and GPU is 14.0x (memory bandwidth is 30.4x)
- All compute nodes are interconnected with full bisection

- Highly-tuned code implemented using SIMD intrinsics (AVX-512) and OpenMP is used for CPU measurement
- Fix the problem size to 256 × 256 × 512 elements (101,649,411 DOF) per GPU

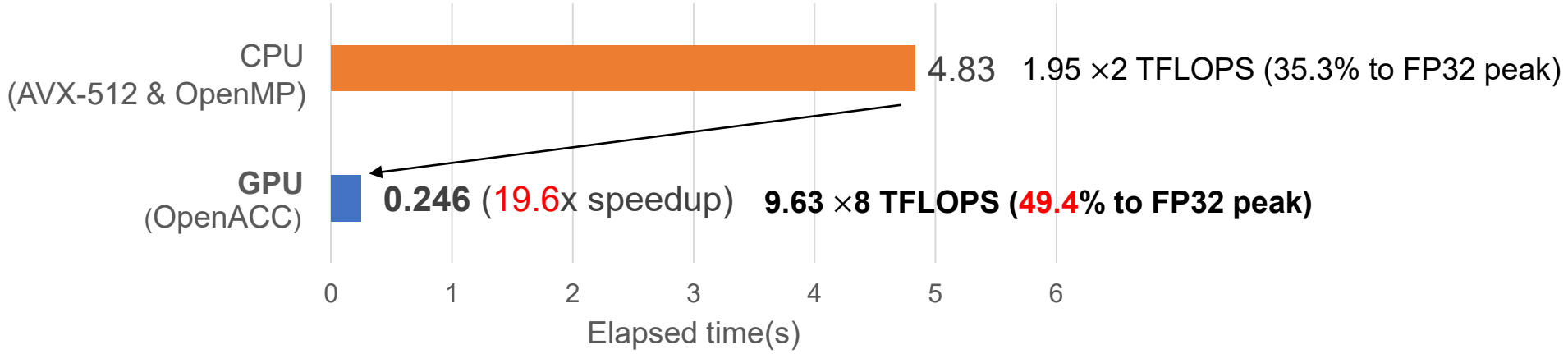
Kernel level performance - EBE kernel (FP64)

- Due to spill/fill caused by many intermediate variables and data access, the performance on CPU is not so high
- On GPU, spill/fill is avoided thanks to the large number of registers
  - Coloring: 55% of FP64 peak, even higher performance than CPU
  - Atomics: 72% of FP64 peak, 1.31x faster than coloring
- Element wise computation of intermediate variables led to 75.0% of FP64 peak
  - 93.0% of performance implemented by CUDA is attained using OpenACC



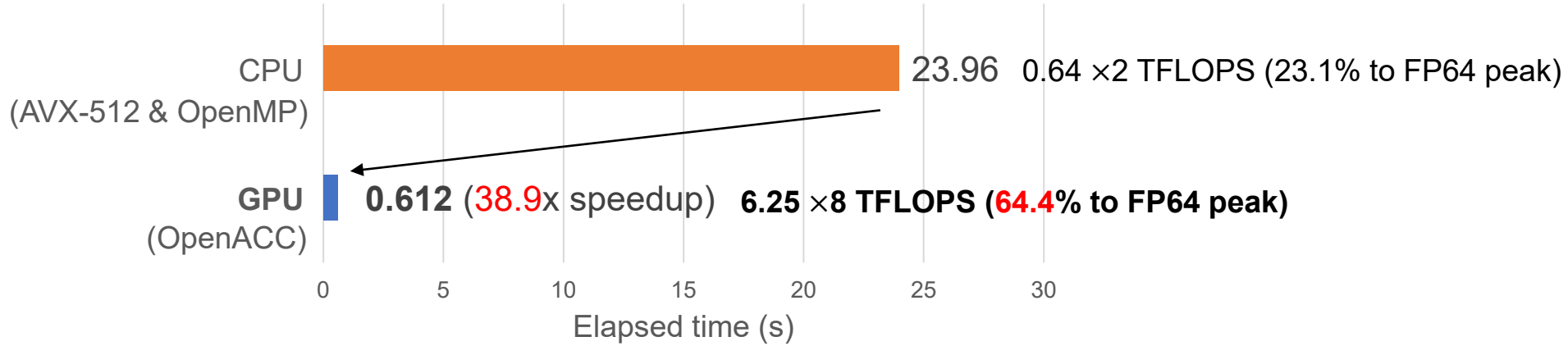
Kernel level performance – GF-based NN predictor kernel (FP32)

- Due to the high affinity of algorithm to recent architectures, the CPU implementation has a high performance of 35.3% of the FP32 peak
- But the A100 GPU implementation has an even higher performance of 49.4% of the FP32 peak
  - 85.5% of performance implemented by CUDA is attained using OpenACC



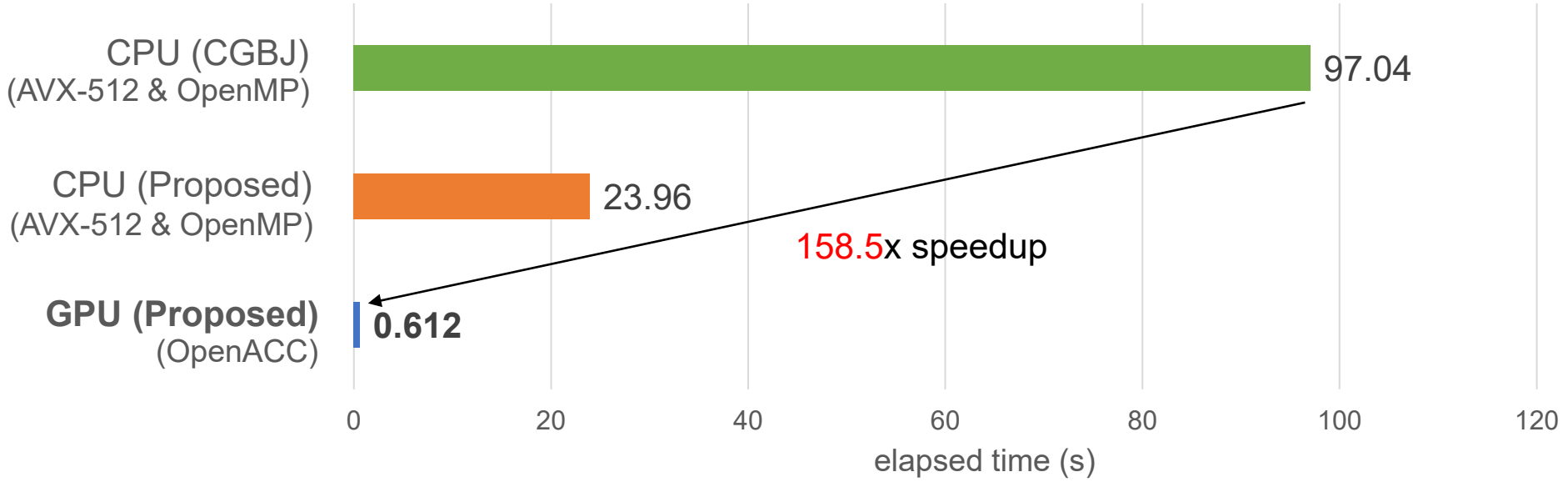
Time-to-solution of the whole solver (mixed-precision)

- Solving the problem up to relative error $|\mathbf{A}\delta\boldsymbol{u} - \boldsymbol{f}|/|\boldsymbol{f}| < 10^{-8}$
- 38.9-fold speedup from the highly-tuned CPU implementation (AVX-512) on the A100 node
  - The FP64 peak performance ratio between CPU and GPU is 14.0x and the memory bandwidth is 30.4x
  - Given these differences and small porting cost of OpenACC, 38.9x speedup seems to be very good



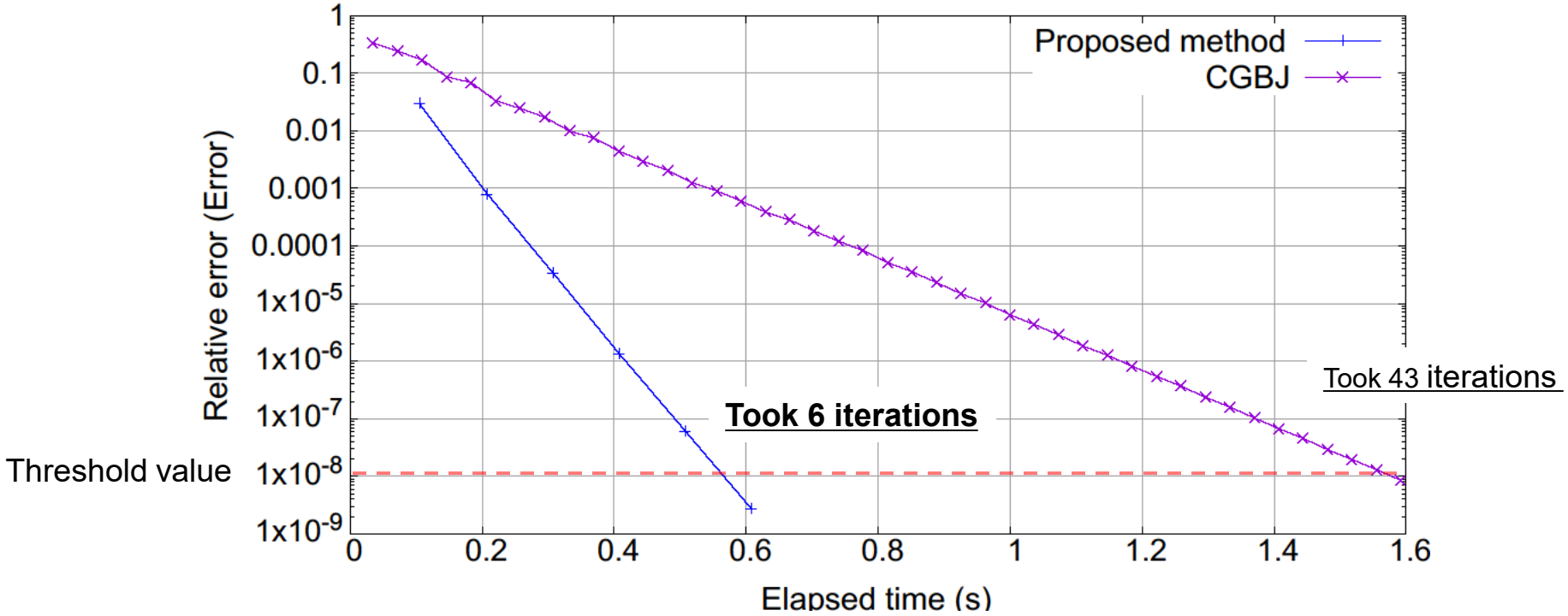
Speedup from conventional method (CGBJ) on CPU

- Compared to the highly-tuned CPU implementation of conventional method (CGBJ: Conjugate Gradient solver with 3×3 Block Jacobi preconditioning), 158.5-fold speedup was achieved on GPU by OpenACC



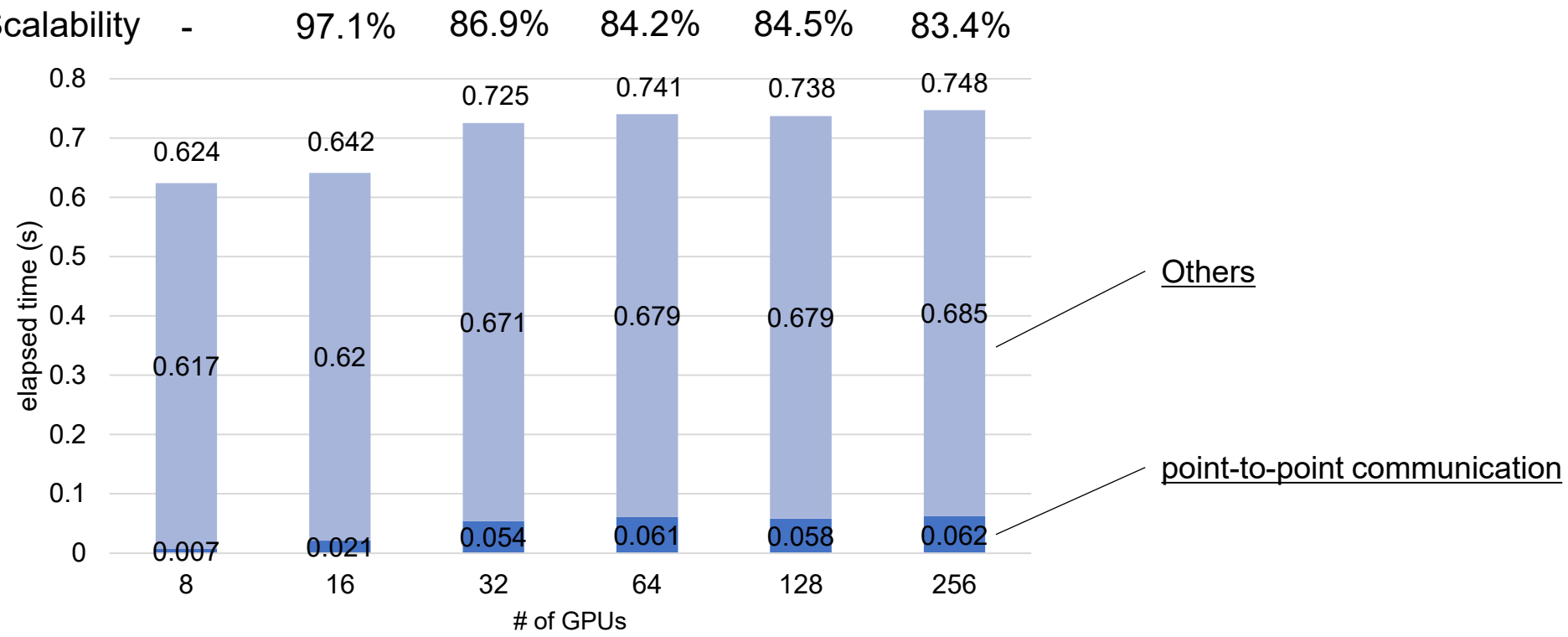
Performance comparison with the conventional method (CGBJ) on GPU

- The refinement rate of preconditioner is better than that of CGBJ
- The peak performance to FP64 improved from 41.9% of CGBJ to 64.4%



Weak Scaling on ABCI A100 node system

- The head model is duplicated in the x and y directions
  - Fix the problem size to $256 \times 256 \times 512$ elements per GPU
- 83.4% weak scaling efficiency from 1 node (8 GPUs) to 32 nodes (256 GPUs)



Summary

- As an example of application combining equation-based and data-driven methods, we ported an implicit PDE solver with NN-based preconditioner
  - Circumvents the problem of low accuracy in NN-based estimations by using NN in Green's functions that reflect property of the PDE
- Selected an appropriate algorithm and tuned it at the kernel level considering the characteristics of the GPU using OpenACC
- At the kernel level;
  - EBE: 93.7x speedup from CPU, 93.0% of performance implemented by CUDA
  - GF-based NN-predictor: 19.6x speedup from CPU, 85.5% of performance implemented by CUDA
- For whole solver;
  - 158x speedup from conventional method on CPU, 38.9x speedup from proposed method on CPU, 64.4% of the FP64 peak
  - 83.4% of weak scaling efficiency was obtained from 1 node (8 GPUs) to 32 nodes (256 GPUs) on ABCI A100 system
- Same approach expected to be effective in porting other HPC applications combining data-driven and equation-based methods with directive-based programming models