WACCPD 2019 @ Denver
Nov. 18, 2019

GPU Implementation of a Sophisticated Implicit Low-Order Finite Element Solver with FP21-32-64 Computation using OpenACC

Takuma Yamaguchi[1], Kohei Fujita[1;2], Tsuyoshi Ichimura[1], Akira Naruse[3], Maddegedara Lalith[1], and Muneo Hori[4]

1. The University of Tokyo

2. RIKEN Center for Computational Science

3. NVIDIA Corporation

4. Japan Agency for Marine-Earth Science and Technology

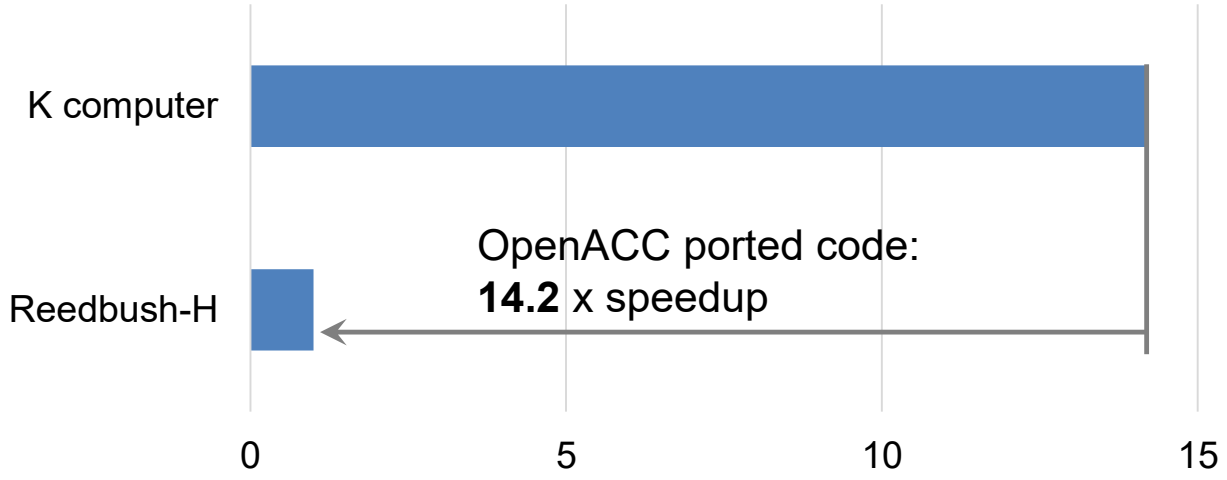
Introduction

- Scientific simulation software needs to keep up with rapid development of computer systems to benefit from improving hardware capabilities
- Implementing hardware/system specific software is costly; thus, developing maintainable code with portability and performance is required for high productivity
  - Especially important for real-world programs with sophisticated algorithms and many lines of code
- OpenACC is an option for performance-portable code development for both CPU- and GPU-based systems
  - Porting sophisticated programs by OpenACC and sharing the process can be useful for porting other applications

Target problem: Implicit low-order finite-element analysis

- De-facto standard for manufacturing and Earth sciences
- Consists of iterative solvers with preconditioners
  - Involves random memory accesses and complex communication patterns
- In WACCPD 2016 & 2017 we demonstrated that CPU-based finite-element solvers can be ported to GPU systems using OpenACC with high performance (Best Paper Award in WACCPD 2016 & 2017)

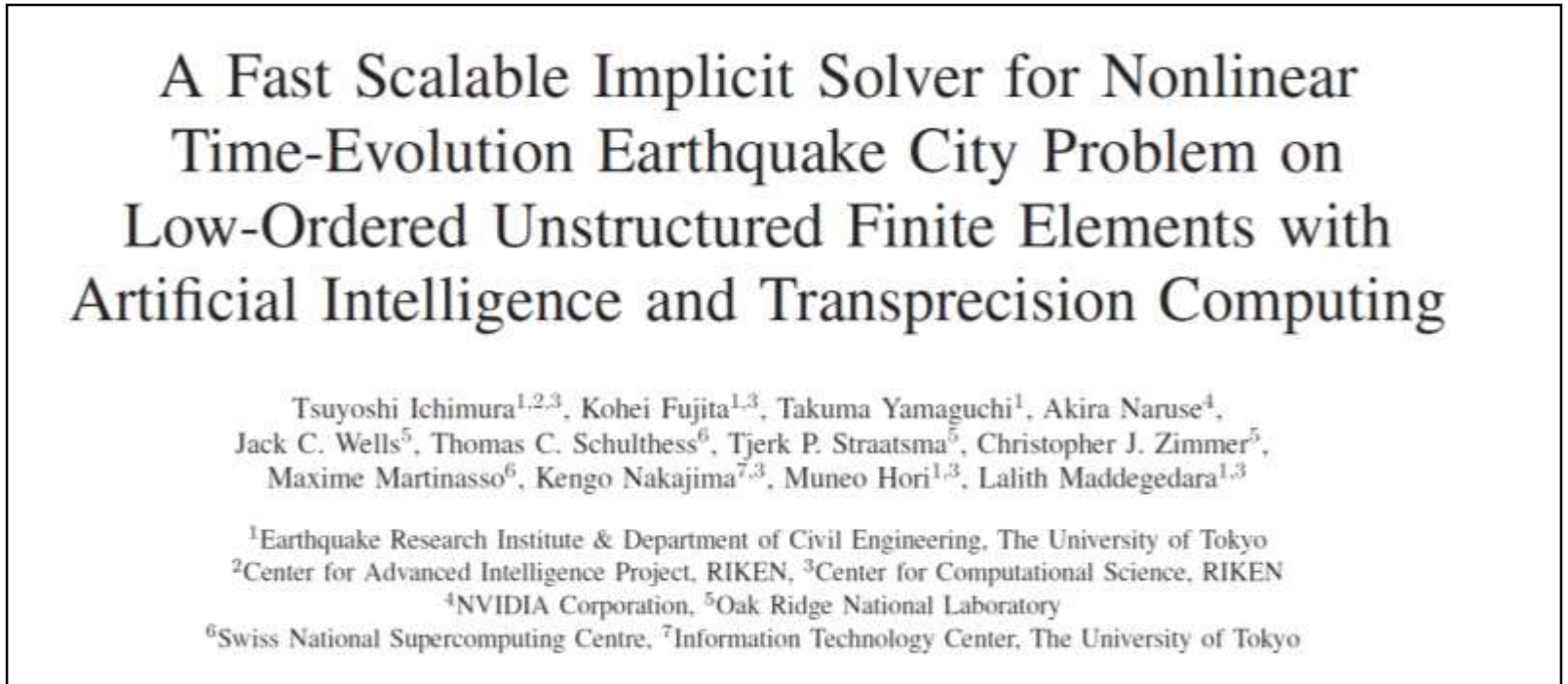
	K computer	Reedbush-H
# of nodes	20	10
CPU/node	1 x SPARC64 VIIIfx	2 x Intel Xeon E5-2695 v4
GPU/node	-	2 x NVIDIA P100
Hardware peak FLOPS ratio	1	**41.4**
Memory bandwidth ratio	1	**11.4**



Yamaguchi et al., Implicit Low-Order Unstructured Finite-Element Multiple Simulation Enhanced by Dense Computation using OpenACC, WACCPD 2017

Target problem: Implicit low-order finite-element analysis

- In SC18, we developed a more sophisticated solver algorithm for Summit
  - Incorporates artificial intelligence (AI) and FP16-FP21-FP32-FP64 transprecision computing
  - Thoroughly optimized for NVIDIA Tesla V100 GPUs with CUDA; extremely high performance (**selected as Gordon Bell Prize Finalist**)
  - However, lacks portability and maintainability



A Fast Scalable Implicit Solver for Nonlinear Time-Evolution Earthquake City Problem on Low-Ordered Unstructured Finite Elements with Artificial Intelligence and Transprecision Computing

Tsuyoshi Ichimura[1,2,3], Kohei Fujita[1,3], Takuma Yamaguchi[1], Akira Naruse[4], Jack C. Wells[5], Thomas C. Schulthess[6], Tjerk P. Straatsma[5], Christopher J. Zimmer[5], Maxime Martinasso[6], Kengo Nakajima[7,3], Muneo Hori[1,3], Lalith Maddegedara[1,3]

[1]Earthquake Research Institute & Department of Civil Engineering, The University of Tokyo
[2]Center for Advanced Intelligence Project, RIKEN, [3]Center for Computational Science, RIKEN
[4]NVIDIA Corporation, [5]Oak Ridge National Laboratory
[6]Swiss National Supercomputing Centre, [7]Information Technology Center, The University of Tokyo

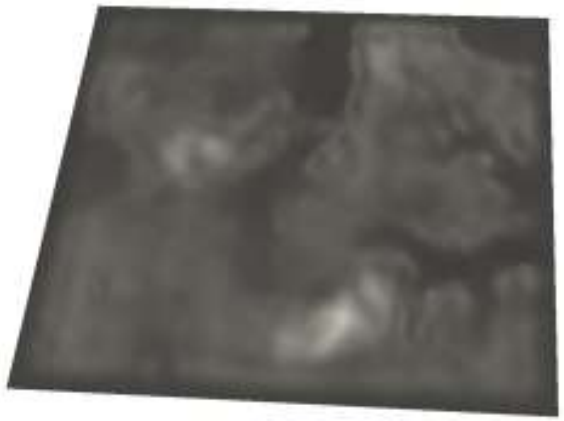
Target problem: Implicit low-order finite-element analysis

- In this paper, we port the SC18 solver algorithm to GPUs using OpenACC
  - Demonstrate that OpenACC porting achieves high speedup with small developmental cost even for sophisticated application with non-standard data-types (FP21)

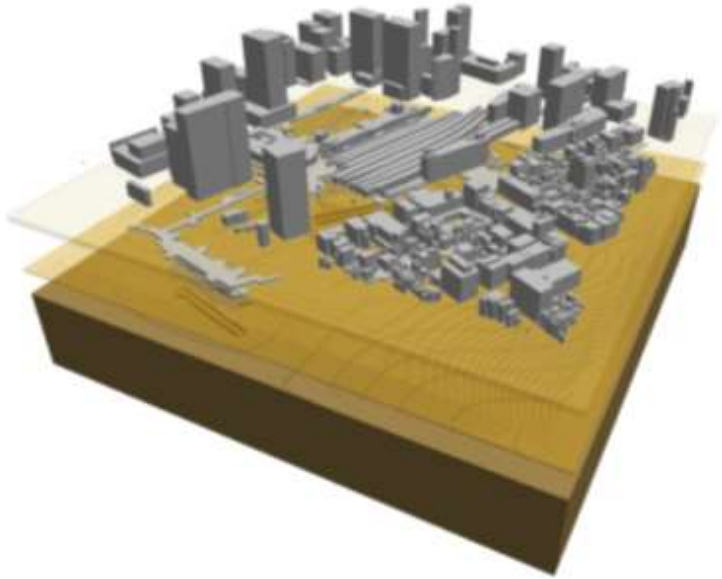
Overview of SC18 solver

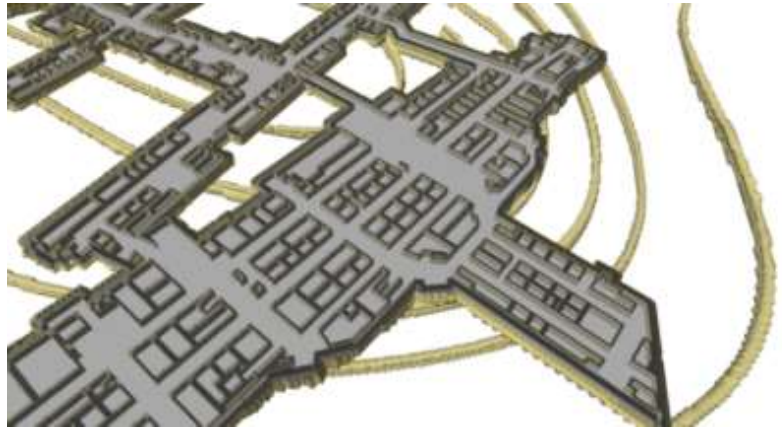
Scientific target: Urban earthquake modeling

- Unstructured mesh with implicit solvers required for urban earthquake modeling
  - We have been developing high-performance implicit unstructured finite-element solvers (SC14 & SC15 Gordon Bell Prize Finalist, SC16 best poster)
- However, simulation incorporating coupling of ground & structure requires finer resolution
  - Traditional physics-based modeling too costly
  - Can we combine use of data analytics to solve this problem?



SC14, SC15 & SC16 solvers: ground simulation only

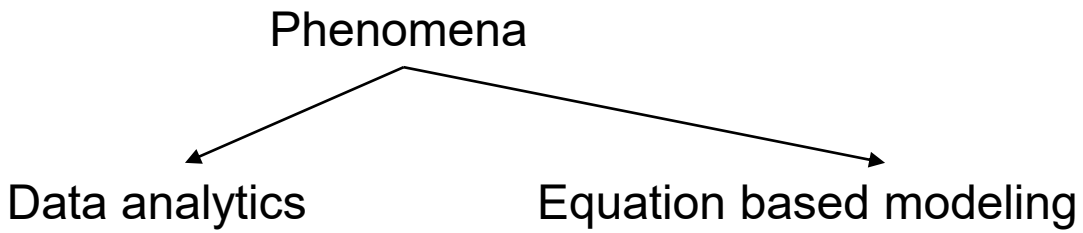




Fully coupled ground-structure simulation with underground structures

Data analytics and equation based modeling

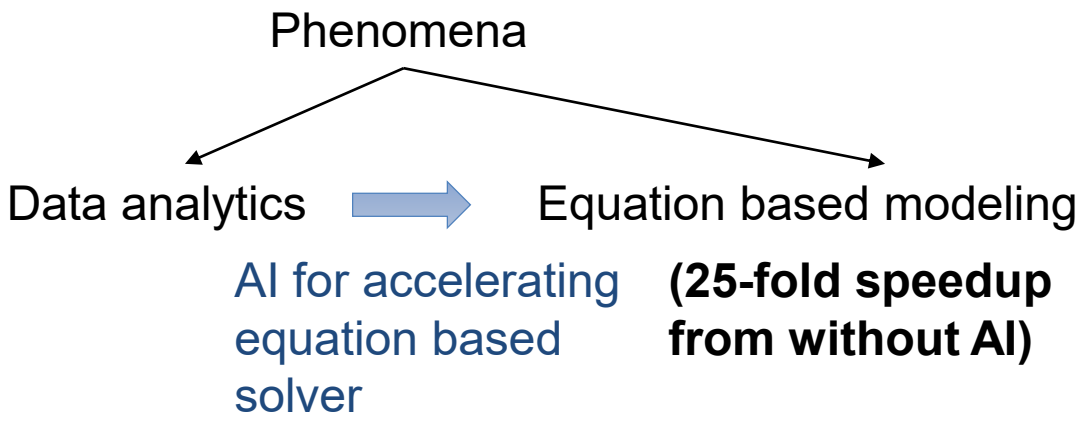
- Equation based modeling
  - Highly precise, but costly
- Data analytics
  - Fast inferencing, but accuracy not as high
- Use both methods to complement each other



Data analytics and equation based modeling

- Equation based modeling
  - Highly precise, but costly
- Data analytics
  - Fast inferencing, but accuracy not as high
- Use both methods to complement each other

In SC18 solver:



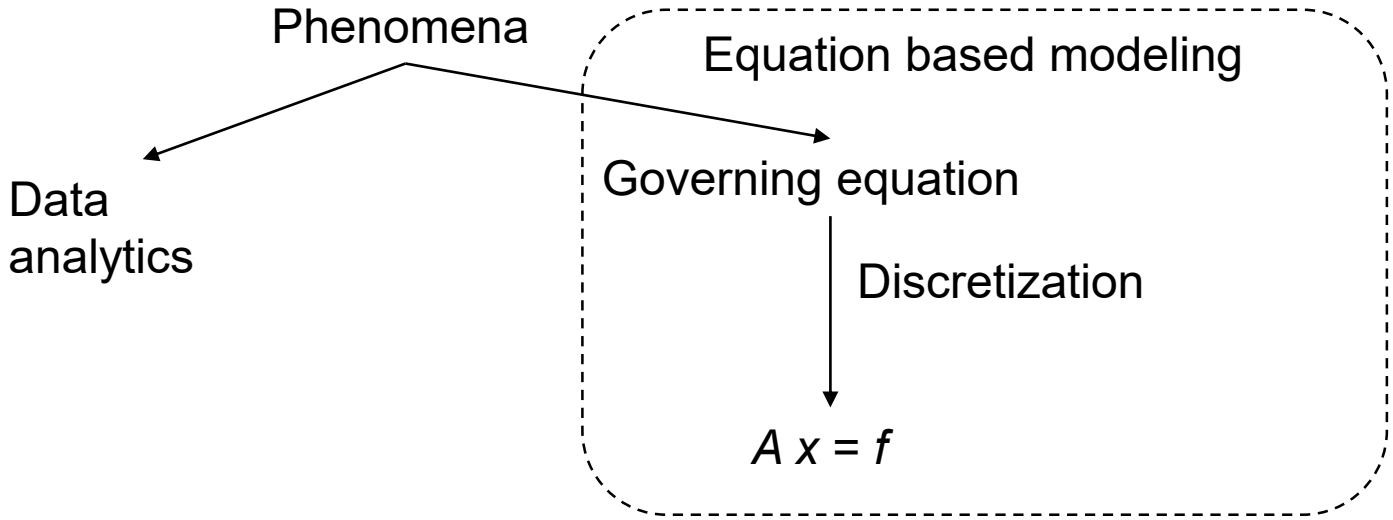
Difficulties of using data analytics to accelerate equation based modeling

- Target: Solve $A\,x = f$
- Difficulty in using data analytics in solver
  - Data analytics results are not always accurate
  - **We need to design solver algorithm that enables robust and cost effective use of data analytics, together with uniformity for scalability on large-scale systems**
- Candidates: Guess A^{-1} for use in preconditioner
  - For example, we can use data analytics to determine the fill-in of matrix; however, challenging for unstructured mesh where sparseness of matrix A is nonuniform (difficult for load balancing and robustness)

  ➡ Manipulation of A without additional information may be difficult…

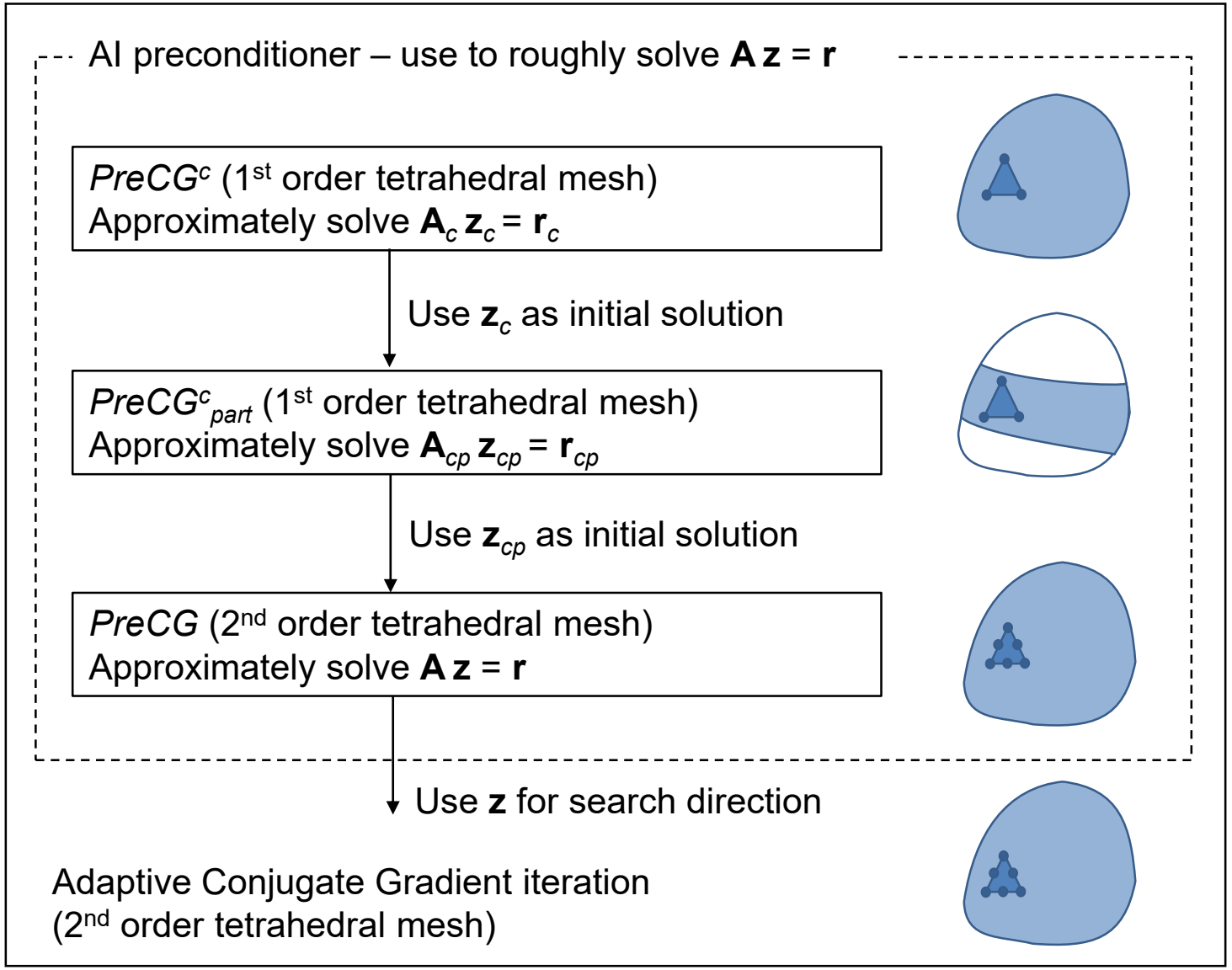
Designing solver suitable for use with AI

- Use information of underlying governing equation
  - Governing equation's characteristics with discretization conditions should include information about the difficulty of convergence in solver
  - Extract parts with bad convergence using AI and extensively solve extracted part



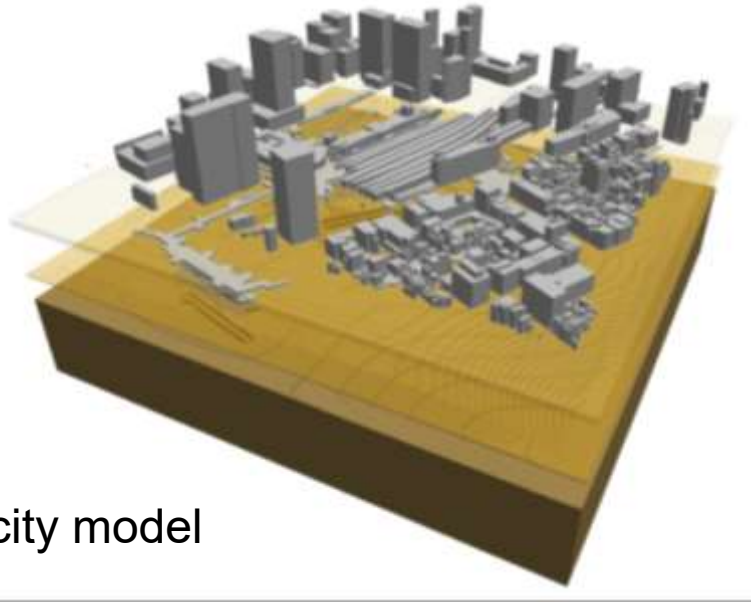
Solver suitable for use with AI

- Transform solver such that AI can be used robustly
  - Select part of domain to be extensively solved in adaptive conjugate gradient solver
  - Based on the governing equation's properties, part of problem with bad convergence is selected using AI

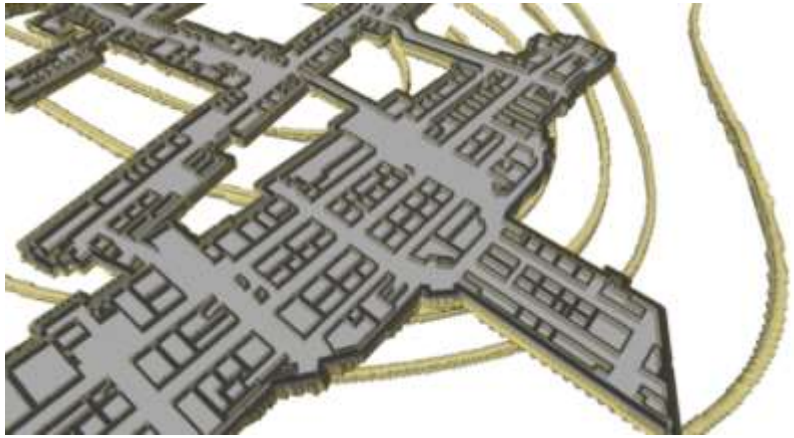


How to select part of problem using AI

- In discretized form, governing equation becomes function of material property, element and node connectivity and coordinates
  - Train an Artificial Neural Network (ANN) to guess the degree of difficulty of convergence from these data



Whole city model



Extracted part by AI (about 1/10 of whole model)

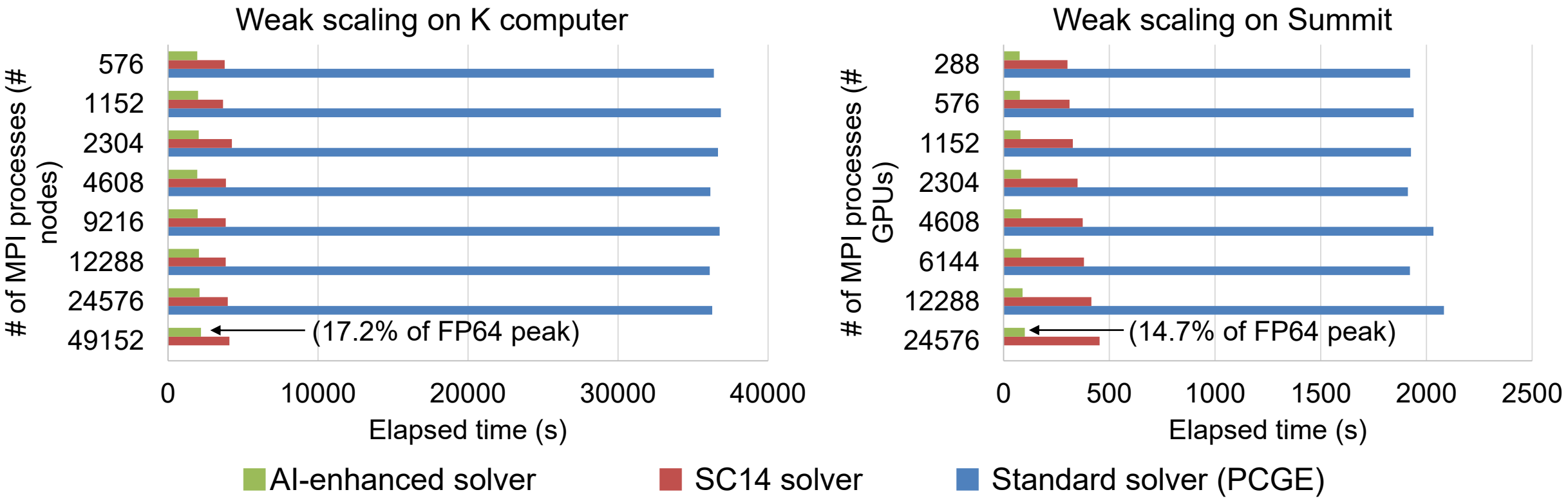
Performance of AI-enhanced solver

- FLOP count decreased by 5.56-times from PCGE (standard solver; Conjugate Gradient solver with block Jacobi preconditioning)

	Without AI	With AI
CG iterations	132,665	88
PreCG^{c} iterations	-	5,803
PreCG$^{c}_{part}$ iterations	-	26,826
PreCG iterations	-	3,103
FLOPS count	184.7 PFLOP	33.2 PFLOP

Performance of AI-enhanced solver on K computer/Summit

- PCGE: standard solver; Conjugate Gradient solver with block Jacobi preconditioning
- SC14: Gordon Bell Prize finalist solver (with multi-grid & mixed-precision arithmetic)



GPU implementation of SC18 solver

- CUDA is used for Summit
- Optimized codes for better performance
  - Specific descriptions for FP16 computations on NVIDIA Tesla V100 GPUs were applied
- Lacks portability and maintainability

⋮

```
const half2 dc_vec1 = half2(dc[1], dc[1]);
const half2 dc_vec2 = half2(dc[2], dc[2]);
const half2 dc_vec3 = half2(dc[3], dc[3]);
const half2 dc_vec4 = half2(dc[4], dc[4]);

tmp1_vec = __hadd2(Bu_vec0,Bu_vec1);
tmp1_vec = __hadd2(tmp1_vec,Bu_vec2);
tmp2_vec = __hmul2(dc_vec1,tmp1_vec);

DBu_vec0 = __hmul2(dc_vec0,Bu_vec0);
DBu_vec0 = __hadd2(DBu_vec0,tmp2_vec);
DBu_vec1 = __hmul2(dc_vec0,Bu_vec1);
DBu_vec1 = __hadd2(DBu_vec1,tmp2_vec);
DBu_vec2 = __hmul2(dc_vec0,Bu_vec2);
DBu_vec2 = __hadd2(DBu_vec2,tmp2_vec);
DBu_vec3 = __hmul2(dc_vec2,Bu_vec3);
DBu_vec4 = __hmul2(dc_vec3,Bu_vec4);
DBu_vec5 = __hmul2(dc_vec4,Bu_vec5);
```

⋮

Example code using CUDA

Summary of first half of talk

- We try to port our SC18 Gordon Bell Prize finalist solver by OpenACC in this paper
- The SC18 solver uses artificial intelligence to improve performance from previous solvers
- High performance attained on Summit; however, maintainability and portability of the code is low
- From now, we will talk about OpenACC porting of SC18 solver to improve maintainability and portability keeping the high performance

OpenACC-based implementation of solver

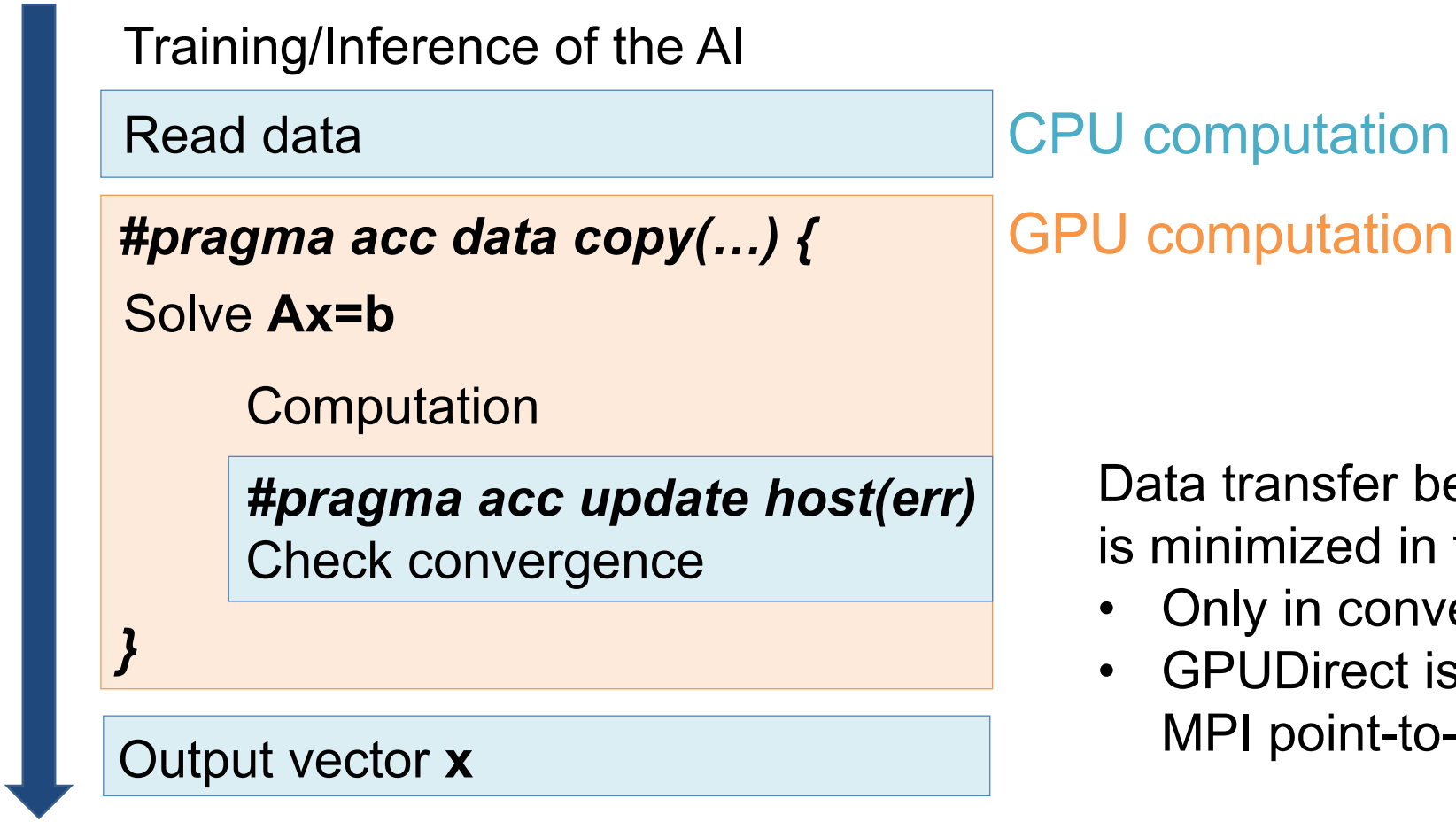
Porting strategy using OpenACC

Demonstrate that the performance is acceptable compared to that in CUDA implementations by following procedures

1. Baseline implementations
   - Define where to apply OpenACC
   - Minimize data transfer between CPU and GPUs
   - Insert directives to parallelize loops
2. Introduction of our custom data type FP21
3. Miscellaneous optimizations for better performance

Baseline implementation of OpenACC

Define where to apply OpenACC



Data transfer between CPU and GPUs is minimized in the solver

- Only in convergence check part
- GPUDirect is used for MPI point-to-point communication

Baseline implementation of OpenACC

Insertion of some directives for parallel computation

Example for
sparse-matrix vector multiplication

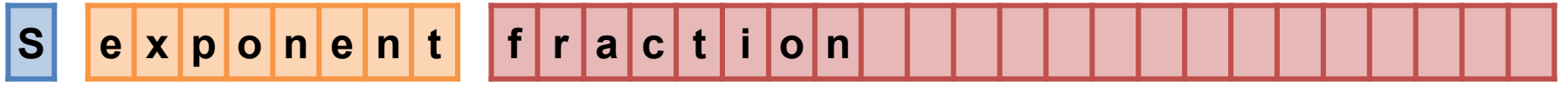
- Collapse element-wise loop and timestep-wise loop to extract parallelism

```
1  #pragma acc parallel loop collapse(2)
2  for(i_ele = 0; i_ele < (*n_element); i_ele++){
3    for(i_vec = 0; i_vec < (*n_vector); i_vec++){
4      cny0 = connect[i_ele][0];
5      cny1 = connect[i_ele][1];
6      …
7      cny9 = connect[i_ele][9];
8
9      u0x = u[cny0][0][i_vec];
10     u0y = u[cny0][1][i_vec];
11     u0z = u[cny0][2][i_vec];
12     …
13     u9z = u[cny9][2][i_vec];
14
15     Au0x = …
16     …
17     Au9z = …
18
19     #pragma acc atomic
20     r[cny0][0][cny0] += Au0x;
21     …
22     #pragma acc atomic
23     r[cny9][2][cny9] += Au9z;
24   }
25 }
```

Introduction of custom data type: FP21

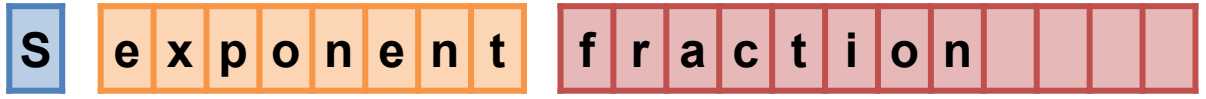
- Most computation in CG loop is memory bound computation
  - However, it's impossible to use FP16 or bfloat16 for whole vector
    - Small dynamic range easily leads to overflow/underflow
    - Less accuracy hamper the convergence of the solver
- Define custom data type FP21 numbers

Single precision (FP32, 32 bits)



1bit sign + 8bits exponent + 23bits fraction

(FP21, 21 bits)



1bit sign + 8bits exponent + 12bits fraction

(bfloat16, 16 bits)

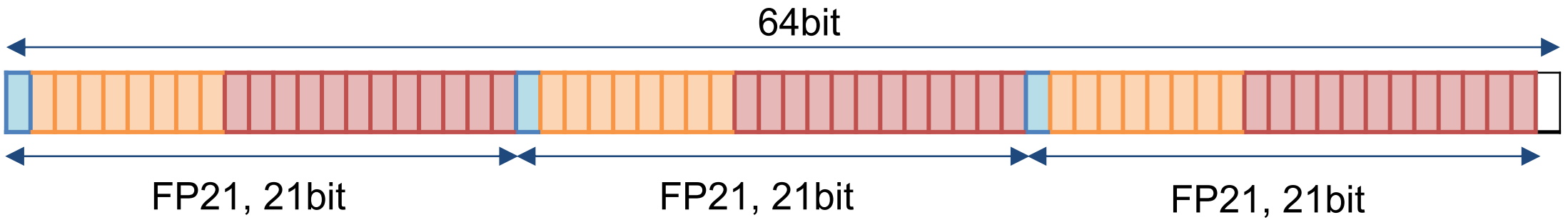
1bit sign + 8bits exponent + 7bits fraction

Half precision (FP16, 16 bits)

1bit sign + 5bits exponent + 10bits fraction

Implementation of FP21 computation

- Not supported in hardware, used only for storing
  - FP21(stored)⇐bit operation⇒FP32(computed)
- FP21 × 3 are stored into 64bit array
  - We are solving 3D finite element solver, so x, y, and z components can be stored as one components of 64 bits array



- 1/3 of memory consumption compared to FP64 variables
- This data type is used only in the preconditioning part

Dot product targeting multiple vectors

- Reduction option in OpenACC are only for scalar variables
  - Creating multiple scalar variables and corresponding loops leads to stride memory accesses
- Introduced CUDA
  - Via wrapper, CUDA-based kernels can be called easily from OpenACC-based code

```
1 #pragma acc parallel loop reduction(+:xy0, xy1, xy2, xy3)
2 for(i = 0; i < (*n_node); i++){
3   xy0 += (x[i][0][0]*y[i][0][0]+
            x[i][1][0]*y[i][1][0]+
            x[i][2][0]*y[i][2][0])*z[i];
4   …
5   xy3 += (x[i][0][3]*y[i][0][3]+
            x[i][1][3]*y[i][1][3]+
6           x[i][2][3]*y[i][2][3])*z[i];
7 }
```

```
1 __global__
2 void dotproduct(int *n, float *x, float *y, float *z, float *xy){
3   /* CUDA computation */
4 }
```

```
1 void dotproduct_wrapper(int *n, float *x, float *y, float *z, float *xy){
2   dotproduct<<<960,128>>>(n, x, y, z, xy);
3 }
```

```
1 #pragma acc host_data use_device(n, x, y, z, xy){
2   dotproduct_wrapper(n, x, y, z, xy);
3 }
```

Reduction in overheads for launching kernels

- Overlap the overhead cost
  - Add `async` options for kernels that can be launched at the same time
- Removal of local arrays in the kernel
  - Sometimes local arrays are stored in local memory instead of registers
    - Decrease the performance of computation bound kernels
  - Change local arrays into scalar variables based on the information provided by the compiler

```
bcimplementation:
   2746, Generating present(nodt(:),id,n1,ib,ibc(:),v0(:),vel1(:),vel3(:),vel2(:))
         Generating Tesla code
       2754, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
       2767, !$acc loop seq
   2746, Local memory used for velin
```

Performance measurement

- We solved an earthquake city simulation with 39 million DOF
- Used AI bridging Cloud Infrastructure (ABCI)
  - 1 compute node has 4 × V100 GPU + 2 × Intel Xeon Gold 6148 CPU
- Compare 5 versions of the solver

		Programming model	Precision
1:	CPU-based	OpenMP	FP32-64
2:	Baseline	OpenACC	FP32-64
3:	Baseline	CUDA	FP32-64
4:	Proposed	OpenACC	FP21-32-64
5:	SC'18 Gordon Bell Finalist solver	CUDA	FP16-21-32-64

Performance of FP21 kernels - 1/3

- Measured the performance of each kernel using one compute node of ABCI
  - Peak memory bandwidth of GPU: 900 GB/s, CPU: 63.9 GB/s

		CPU-based	**Baseline OpenACC**	**proposed**
Precision		FP32	FP32	FP21
AXPY	Elapsed time	9.61 ms	0.605 ms	0.401 ms
	Measured bandwidth	50.2 GB/s	797 GB/s	802 GB/s
	Speedup ratio	**1**	**15.8**	**24.0**

× 1.5

- Achieved reasonable speedup in the AXPY kernel

Performance of FP21 kernels - 2/3

- Measured the performance of each kernel using one compute node of ABCI
  - Peak memory bandwidth of GPU: 900 GB/s, CPU: 63.9 GB/s

		CPU-based	**Baseline OpenACC**	**proposed**
Precision		FP32	FP32	FP21
Dot-product	Elapsed time	6.20 ms	0.456 ms	0.277 ms
	Measured bandwidth	54.0 GB/s	735 GB/s	823 GB/s
	Speedup ratio	**1**	**13.6**	**22.4**

× 1.64

- Stride memory access in the baseline implementation has some effect on the performance
- Also achieved reasonable speedup in the dot product kernel

Performance of FP21 kernels - 3/3

- Measured the performance of each kernel using one compute node of ABCI
  - Peak memory bandwidth of GPU: 900 GB/s, CPU: 63.9 GB/s

		CPU-based	**Baseline OpenACC**	**proposed**
Precision		FP32	FP32	FP32/21
Matrix-vector product	Elapsed time	54.61 ms	3.65 ms	3.69 ms
	Speedup ratio	**1**	**15.0**	**14.8**

× 0.989

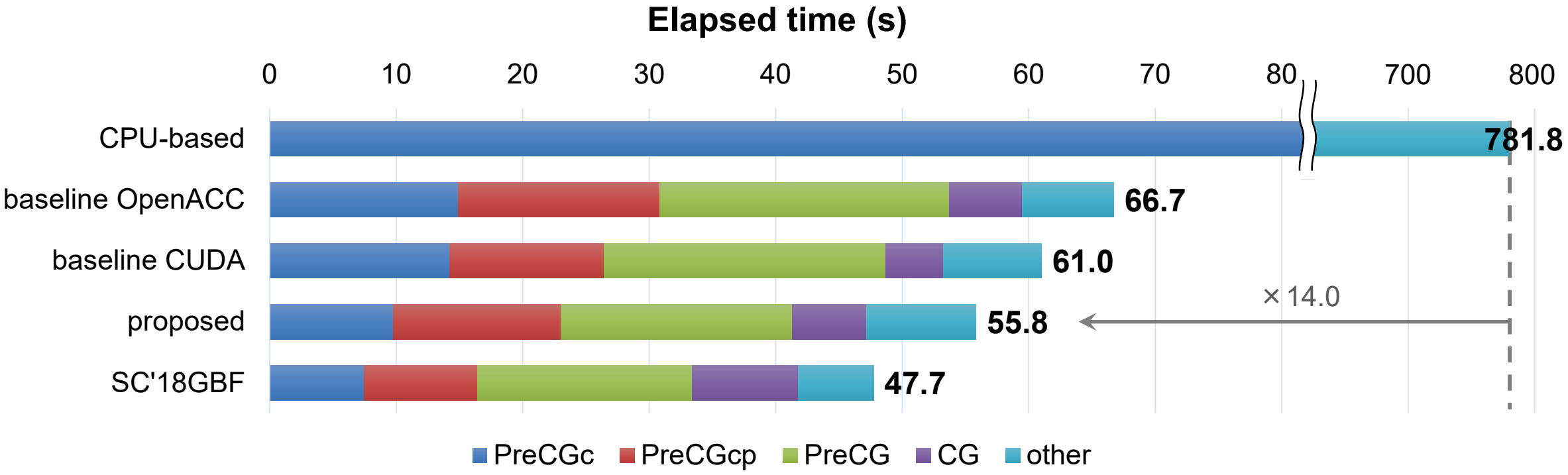
- Matrix-vector product in not memory bound computation
  - Data conversion of FP21/FP32 required additional computation cost
    - Negligible compare to the entire computation cost

Convergency of the solver

# of iterations	Precision in Preconditioning	PreCGc	PreCGcp	PreCG	CG (FP64)
CPU-based	FP32	6199	28830	2674	91
baseline OpenACC	FP32	6300	28272	2729	89
baseline CUDA	FP32	6210	28491	2735	89
proposed	FP32/21	4751	28861	2575	122
SC18GBF	FP32/21/16	4308	26887	2797	129

- The number of Iteration of CG loops with FP64 computation increased
  - Negligible for the entire number of iterations
- Demonstrated the effect of FP21 computations
  - Small difference in the number of iterations when using FP21
  - Failed to converge when using FP16 or bfloat16 instead of FP21

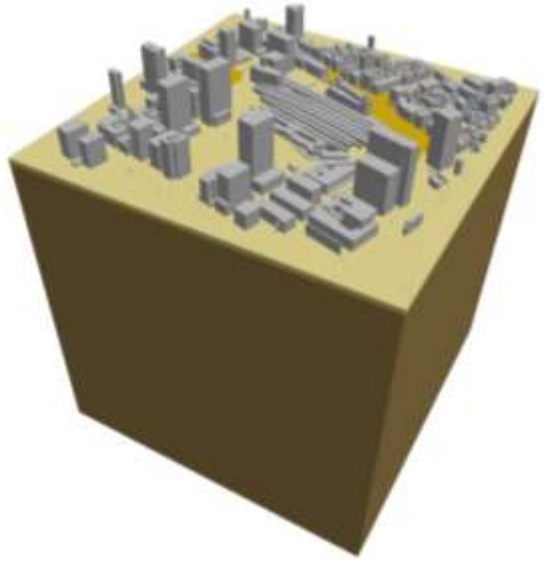
Performance of the whole solver



- Achieved 14-fold speedup compared to the CPU-based codes on CPUs
  - Reasonable speedup considering the ratio of peak memory bandwidth (1:14.1)
- 1.19-fold speedup over the baseline implementation with FP32-64 computations
- 84% of the performance in extremely tuned solver using CUDA

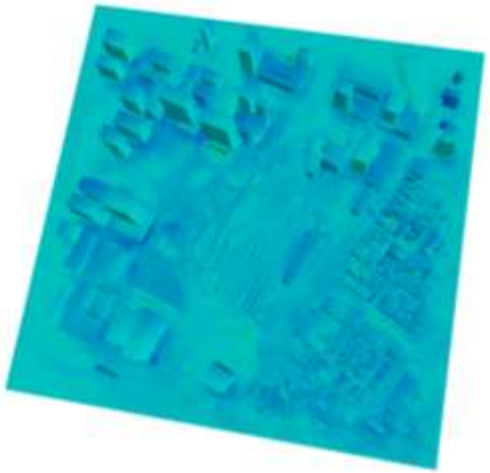
Application Example

- Ground shaking analysis with complicated geometry
  - 1,024 m × 1,024 m city area with underground and building structures surrounded by two-layered ground modeled
  - 1.6×10^{10} degrees of freedom
  - Computed using 384 compute nodes of Summit



a) Target domain

b) Underground structure in the domain



c) Displacement distribution

Future tasks

- I rewrote our Fortran-based codes in C
  - When using Fortran, inline of data conversion between FP21-FP32 requires additional local memory
    - Slightly decrease the performance

```
1  !$acc parallel loop collapse(2)
2  do i_ele = 1, n_element
3    do i_vec = 1, n_vector
4      cny1  = connect(1,i_ele);
5      cny2  = connect(2,i_ele);
6      …
7      cny10 = connect(10,i_ele);
8      …
9      u0x = conv_fp21x3_to_fp32_x(u(i_vec,cny1));
10     u0y = conv_fp21x3_to_fp32_y(u(i_vec,cny1));
11     u0z = conv_fp21x3_to_fp32_z(u(i_vec,cny1));
12     …
```

```
1  #pragma acc parallel loop collapse(2)
2  for(i_ele = 0; i_ele < (*n_element); i_ele++){
3    for(i_vec = 0; i_vec < (*n_vector); i_vec++){
4      cny0 = connect[i_ele][0];
5      cny1 = connect[i_ele][1];
6      …
7      cny9 = connect[i_ele][9];
8      …
9      u0x = conv_fp21x3_to_fp32_x(u[cny0][i_vec]);
10     u0y = conv_fp21x3_to_fp32_y(u[cny0][i_vec]);
11     u0z = conv_fp21x3_to_fp32_z(u[cny0][i_vec]);
12     …
```

- I'm seeking a way to achieve reasonable performance also in Fortran

Summary and future implications

- Portable and maintainable codes with good performance are important for the productivity in science and engineering
- Target our implicit unstructured low-order finite element solver
  - Sophisticated algorithms including AI and transprecision computing are used
  - Latest version extremely tuned for Volta GPU to attain better performance
- Apply OpenACC to the solver
  - FP21 data types can be implemented even in OpenACC
- In performance measurement on ABCI,
  a 14.0-fold speedup over the original CPU codes was achieved
  - 86% of the performance of our extremely tuned solver using CUDA

Summary and future implications

- Source code to use FP21 is available to the public
  → https://github.com/y-mag-chi/fp21axpy

Acknowledgments

- Our results were obtained using Computational resource of AI Bridging Cloud Infrastructure (ABCI), National Institute of Advanced Industrial Science and Technology (AIST).
- We acknowledge support from Post K computer project (Priority Issue 3 - Development of integrated simulation systems for hazards and disasters induced by earthquakes and tsunamis), and Japan Society for the Promotion of Science (17K14719, 18H05239, 18K18873).
- Part of our results were obtained using the Summit at Oak Ridge Leadership Computing Facility, a US Department of Energy, Office of Science User Facility at Oak Ridge National Laboratory.