OpenACC annual meeting 2019
Sep. 2, 2019

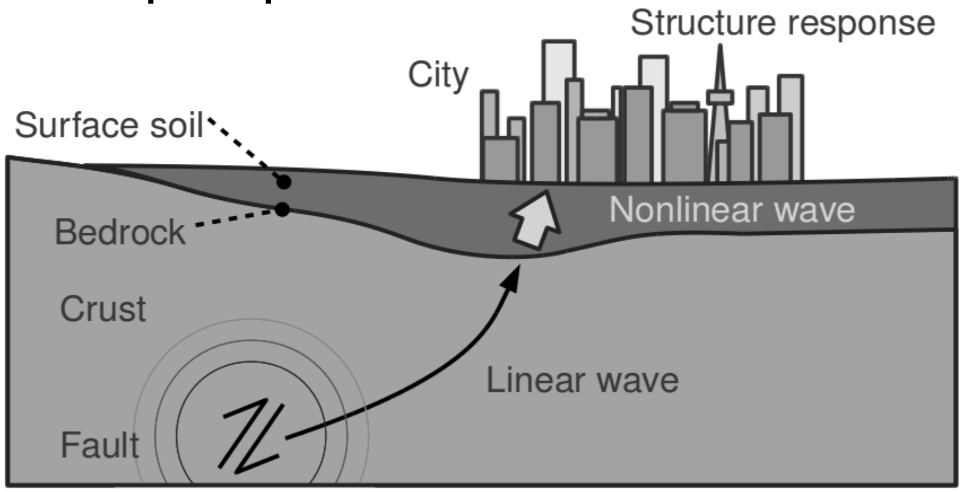
Acceleration of Unstructured Low-Order Finite-Element Earthquake Simulation Using OpenACC

Takuma Yamaguchi

THE UNIVERSITY OF TOKYO

Introduction

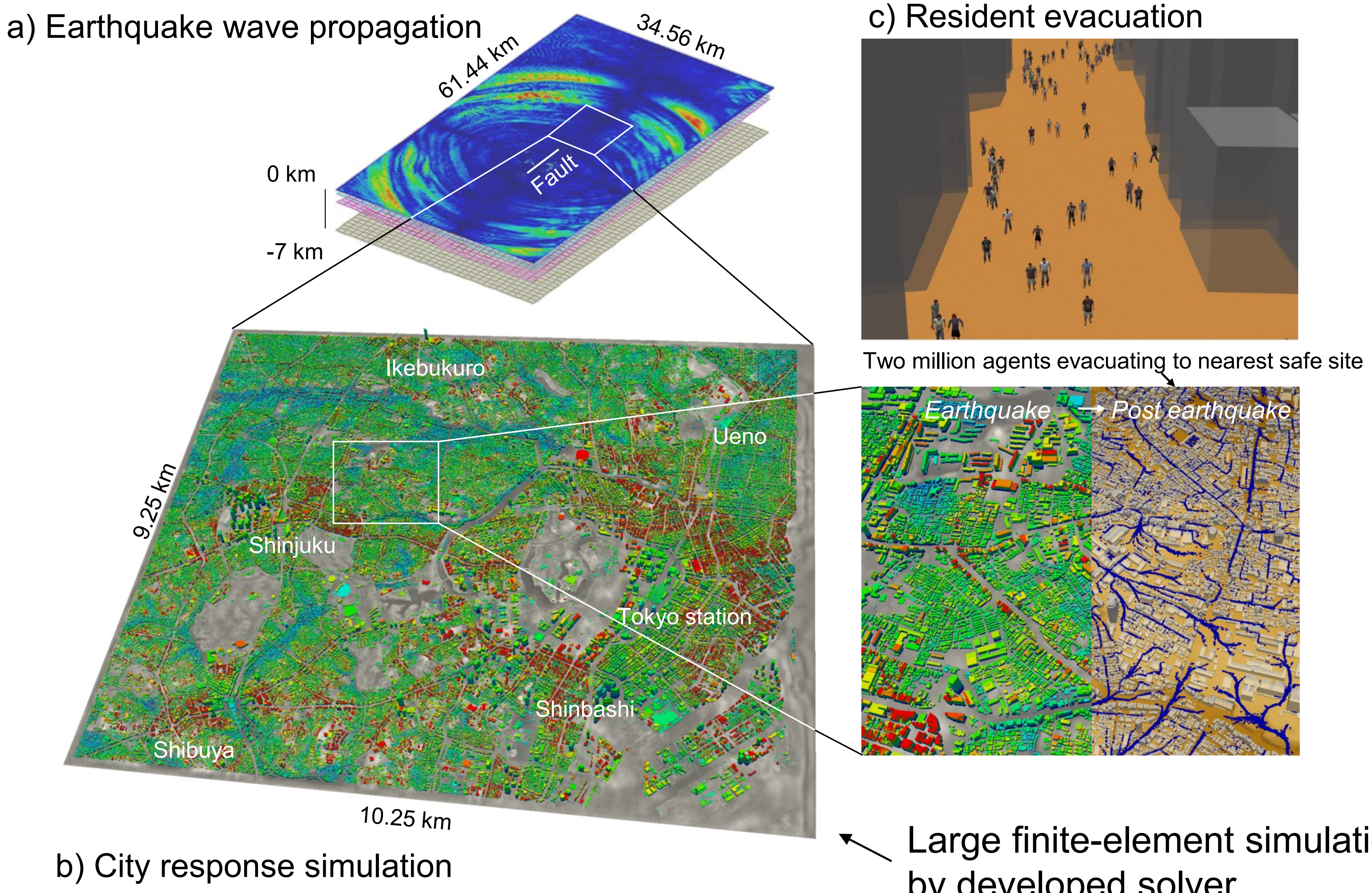
- Contribution of high-performance computing to earthquake mitigation highly anticipated from society
- We are developing comprehensive earthquake simulation that simulate all phases of earthquake disaster by full use of high-performance computers
  - Simulate all phases of earthquake by speeding up core solver
  - **SC14/15/18 Gordon Bell Prize Finalist & SC16/17 Best Poster Awards**
- Today's topic is porting this solver to GPU-based computers
  - Report performance on Volta GPUs



Earthquake disaster process


K computer: 8 core CPU x 82944 node system with peak performance of 10.6 PFLOPS

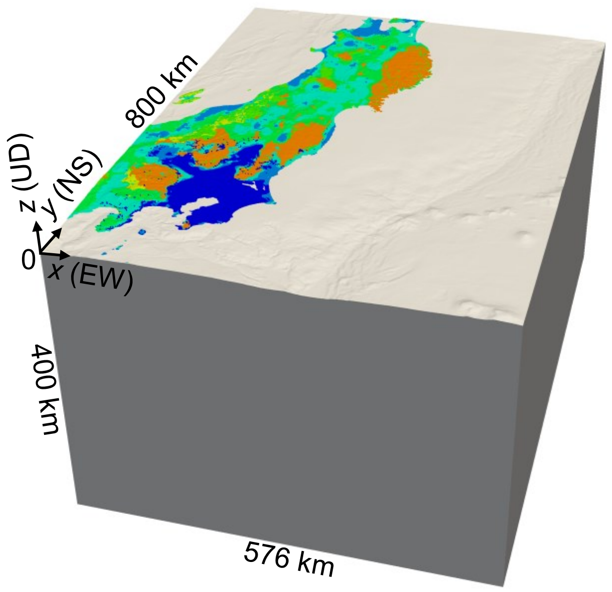
Comprehensive earthquake simulation



Large finite-element simulation enabled by developed solver

Target problem: Earth's crust deformation problem

- Compute elastic response to given fault slip
  - Many case analysis required for inverse analyses and Monte Carlo simulations
- Compute using finite-element method: solve large matrix equation many times
  - Involves many random data access & communication
- Difficulty of problem
  - Attaining load balance & peak-performance & convergency of iterative solver & short time-to-solution at same time
  - Smart use of compute precision space, constraints in solver search space according to physical solution space required



Earth's crust deformation problem

$$Ku = f$$

Outer force vector

Unknown vector with up to 1 trillion degrees of freedom

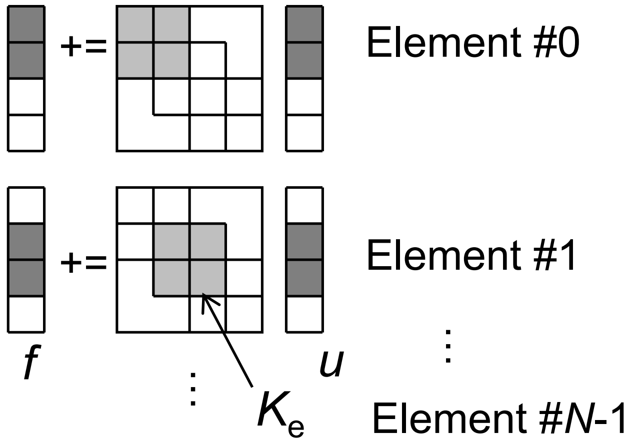
Sparse, symmetric positive definite matrix

Designing scalable & fast finite-element solver

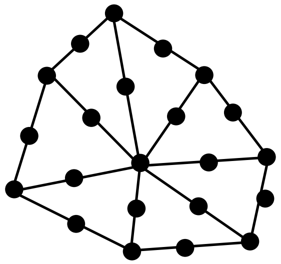
- Design algorithm that can obtain equal granularity at O(million) cores
  - Matrix-free matrix-vector multiplication (Element-by-Element method) is promising: Good load balance when elements per core is equal
    - Also high-peak performance as it is on-cache computation
- Combine Element-by-Element method with multi-grid, mixed precision arithmetic, and adaptive conjugate gradient method
  - Scalability & peak-performance good (core computation kernels are Element-by-Element), convergency good, time-to-solution good

Element-by-Element method

$f = \Sigma_e P_e K_e P_e^T u$
(K_e is generated on-the-fly,



Solver algorithm



Second ordered tetrahedron

CG loop

Solving preconditioning matrix

Computations of outer loop

Outer loop

Equation to be solved (**double precision**)

Solving preconditioning matrix (**single precision**)

Inner loop level 2

Solve system roughly using CG solver

Use $x_1 \leftarrow P_{12}\ x_2$ as initial solution

Inner loop level 1

Solve system roughly using CG solver

Use $x_0 \leftarrow P_{01}\ x_1$ as initial solution

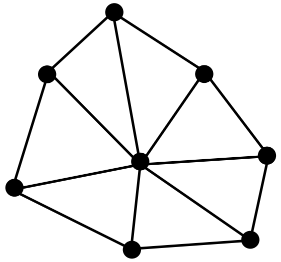
Inner loop level 0

Solve system roughly using CG solver

Use for preconditioner of outer loop

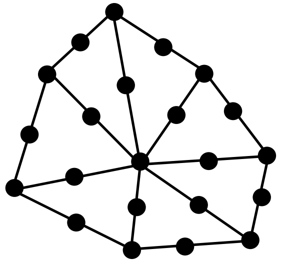
Coarsened equation
$(P_{12}{}^T A_1 P_{12}) x_2 = P_{12}{}^T b_1$

Algebraic coarsening



Linear tetrahedron

Geometric coarsening

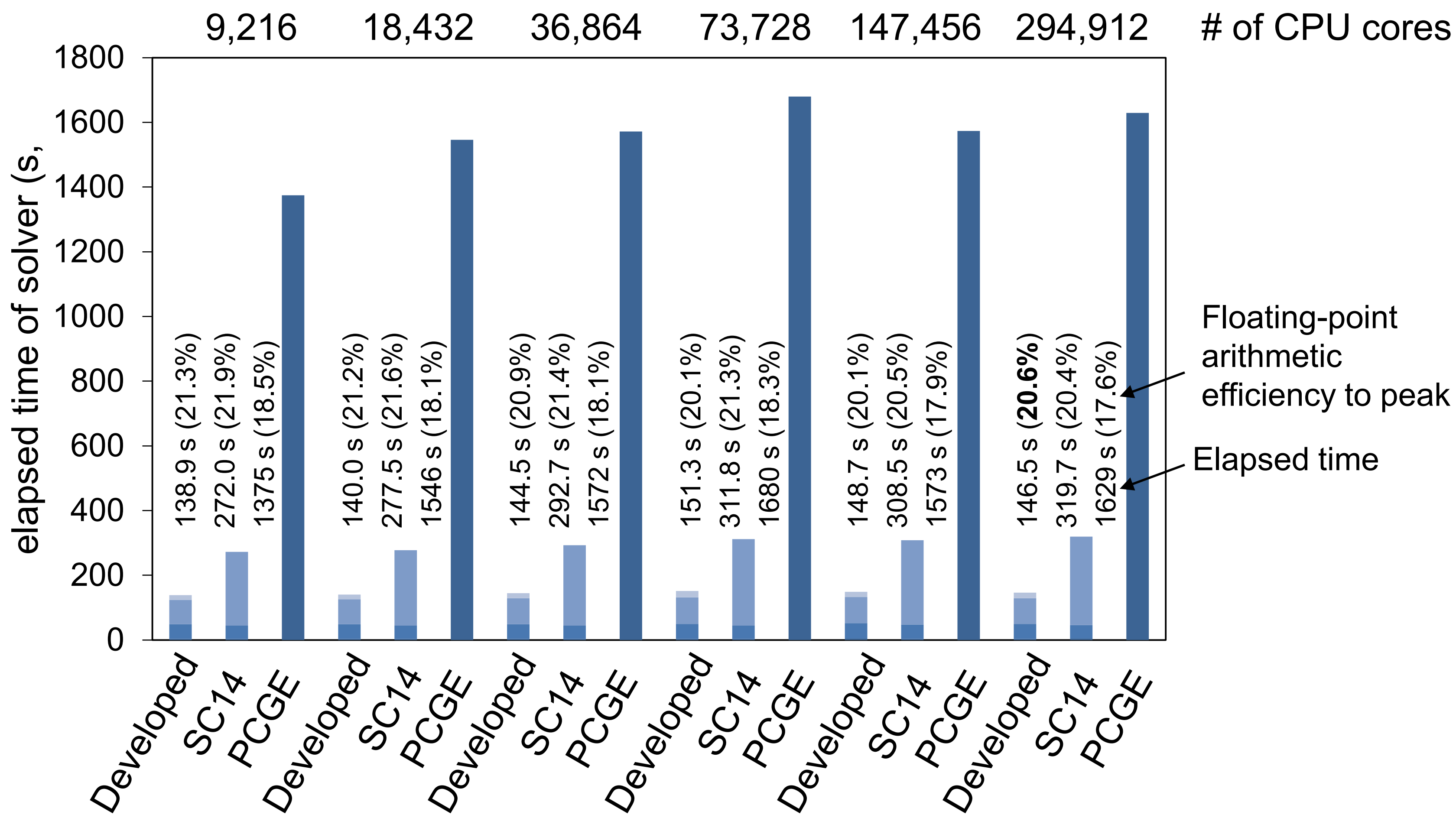


Second ordered tetrahedron

K. Fujita, T. Ichimura, K. Koyama, H. Inoue, M. Hori, L. Maddegedara, Proceedings of the Platform for Advanced Scientific Computing Conference (PASC), June 2017

Performance on K computer

- Developed solver significantly faster than
  - PCGE (standard CG solver algorithm; preconditioning with 3x3 block diagonal matrix)
  - SC14 Gordon Bell Prize finalist solver



- 94.8% scalability from 9216 to 294912 cores
- 4 times peak performance of HPCG benchmark (HPCG on K computer: 5.3% in double precision)

Introduction of GPU computations

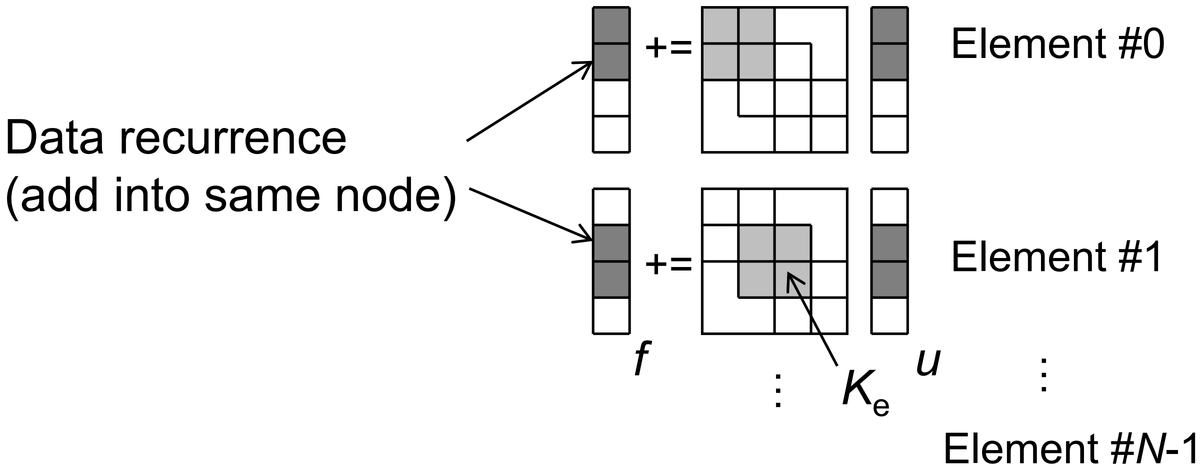
- Further speedup of the simulation by introducing GPUs
  - Good load balance, Reduced computation cost & data transfer size is also beneficial for GPUs
  - High performance can be obtained using OpenACC with low development cost
- GPU architecture is different from CPU architecture
  - More difficult to attain good performance with random data access
  - Relatively smaller cache size

➔Simple porting of the CPU code is not sufficient

Key kernel: Element-by-Element kernel

- Most costly kernel; involves data recurrence
- Algorithm for avoiding data recurrence on **CPUs**
  - Use temporary buffers per core & per SIMD lane
  - Suitable for small core counts with large cache capacity
- Algorithm for avoiding data recurrence on **GPUs**
  - It’s difficult to use buffer per core,
    but recent GPUs have atomic operations supported on hardware
  - Random access becomes bottleneck

Element-by-Element method

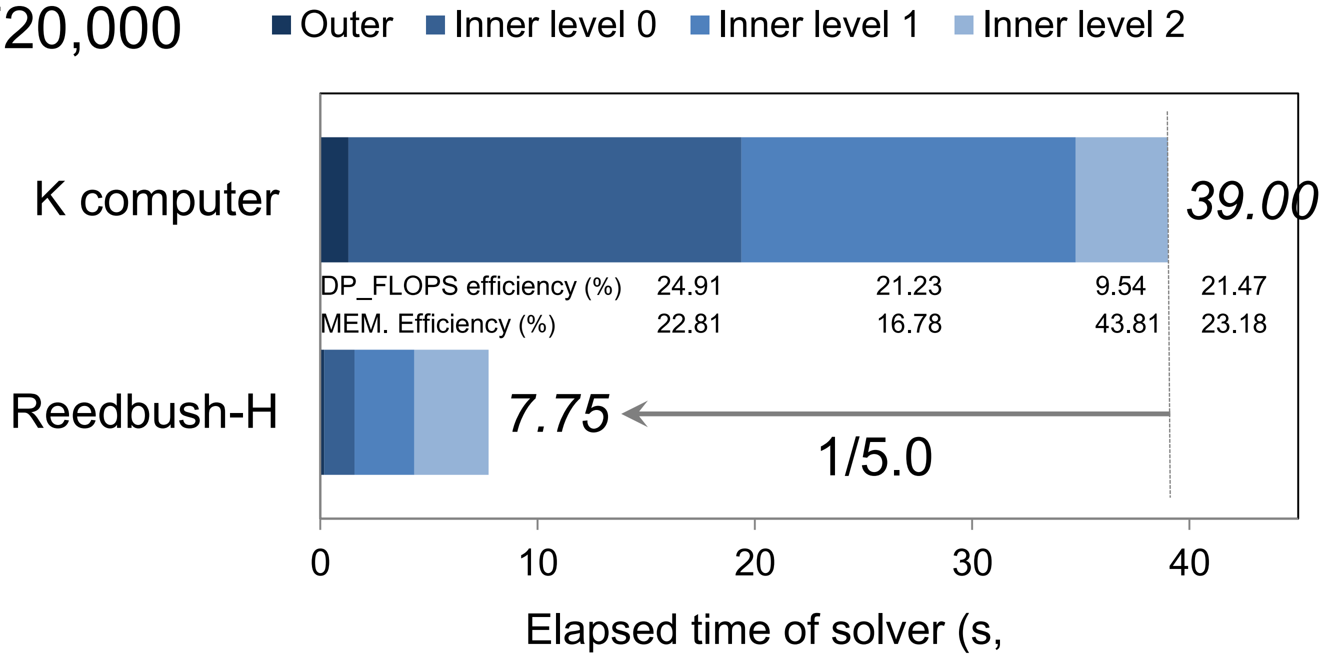


Performance in simple porting

DOF: 125,177,217, # of elements: 30,720,000

Computational Environment

	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 /process	128 GFLOPS	5.30 TFLOPS
Memory bandwidth /process	64 GB/s	732 GB/s



- Simple porting achieved 5.0 times speedup
- However, there is some room for improvement
  - Memory bandwidth is 11 times larger

Strategy for Introduction of OpenACC

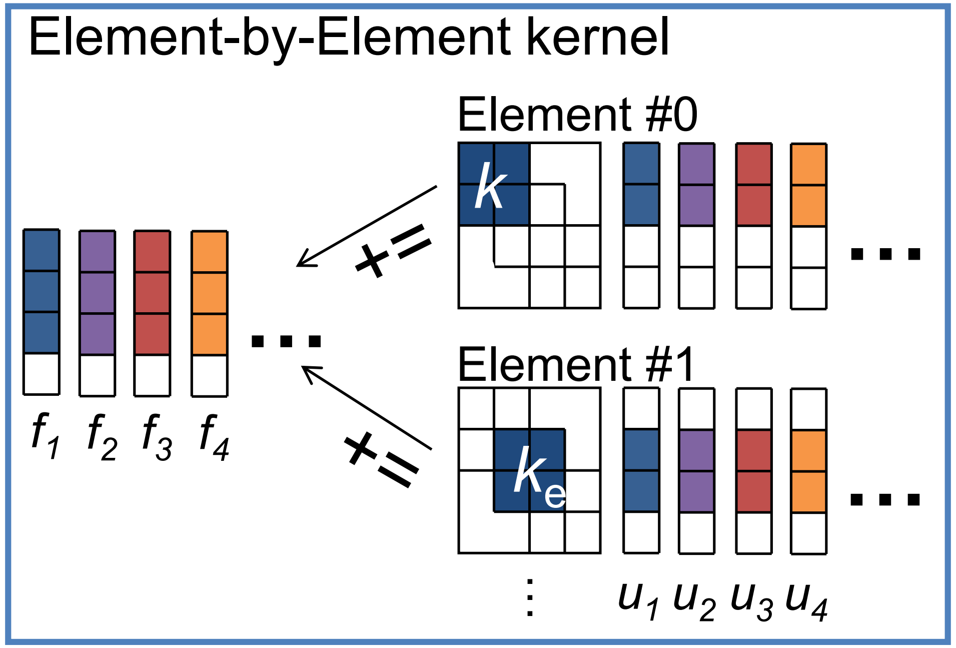
- To attain higher performance, algorithm/implementation suitable for GPUs should differ from that for CPUs

Thereby, we

1. Design the solver algorithm suitable for the GPU architectures
2. Port the solver to GPUs using OpenACC

Modification of Algorithm for GPUs

- Reduce random memory accesses
- Target applications (Inverse analyses, Monte Carlo method etc.) solve many systems of equations
  - Same stiffness matrix
  - Different right-hand side input vectors



- Multiple equations at the same time

$K[u_1, u_2, u_3, \ldots, u_{16}]^T = [f_1, f_2, f_3, \ldots, f_{16}]^T$

Instead of $Ku_1 = f_1, Ku_2 = f_2, Ku_3 = f_3, \ldots$

Introduction of OpenACC – 1/3

Control of data transfer

Read data

Input vector **f**

CPU computation

!$acc data copy(u, …) copyin(f, …)

Solve $\mathbf{u}=\mathbf{K}^{-1}\mathbf{f}$

Computation (i.e, EBE)

!$acc update host(err)

Check convergence

!$acc end data

GPU computation

Output vector **u**

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

Introduction of OpenACC – 2/3

Insertion of some directives for parallel computation

Example for Element-by-Element multiplication

- Assign 16 threads for one element
- Introduce atomic functions to avoid data race

```
!$acc parallel loop collapse(2)
do i_ele = 1, n_element
do i_vec = 1, 16
cny1 = connect(1, i_ele)
   ⋮
cny10 = connect(10, i_ele)

u0101 = u(i_vec, 1, cny1)
   ⋮
u1003 = u(i_vec, 3, cny10)

Ku01 = …
   ⋮
Ku30 = …

!$acc atomic
r(i_vec, 1, cny1) = r(i_vec, 1, cny1) + Ku01
   ⋮
!$acc atomic
r(i_vec, 3, cny10) = r(i_vec, 3, cny10) + Ku30
enddo
enddo
!$acc end parallel
```

Introduction of OpenACC – 3/3

Minor tuning for OpenACC parameters

- The allocation of gang and vector
- The length of vector

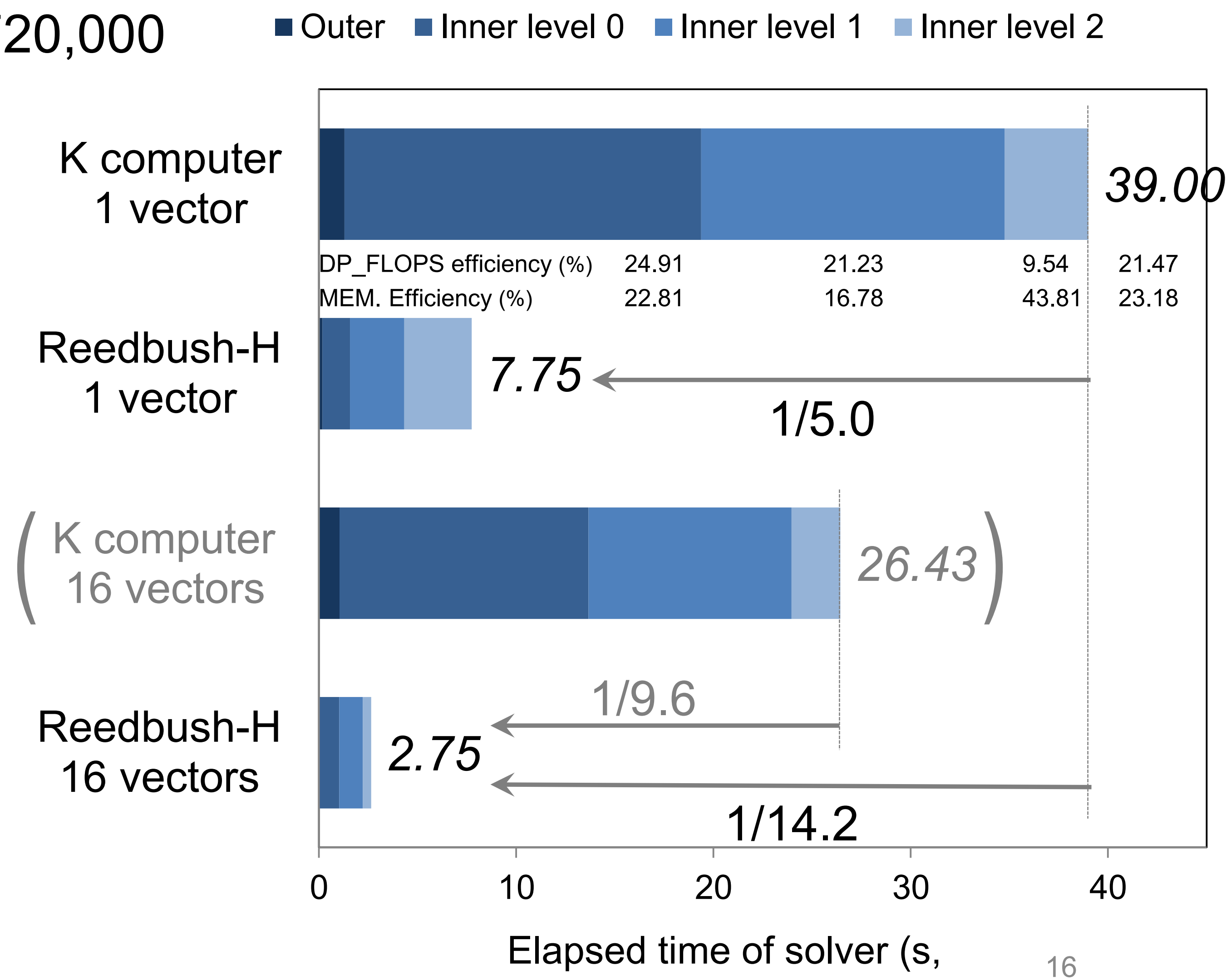
Optimize fine-grain control of parallelism
(Not large effect on performance)

Performance of the proposed solver

DOF: 125,177,217, # of elements: 30,720,000

Computational Environment

	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 /process	128 GFLOPS	5.30 TFLOPS
Memory bandwidth /process	64 GB/s	732 GB/s



Conclusion

- Accelerate the unstructured implicit low-order finite element solvers by OpenACC
  - Design the solver appropriate for GPU computations
  - Port the key kernel to GPUs
- Obtain high performance with low development costs and better portability and maintainability