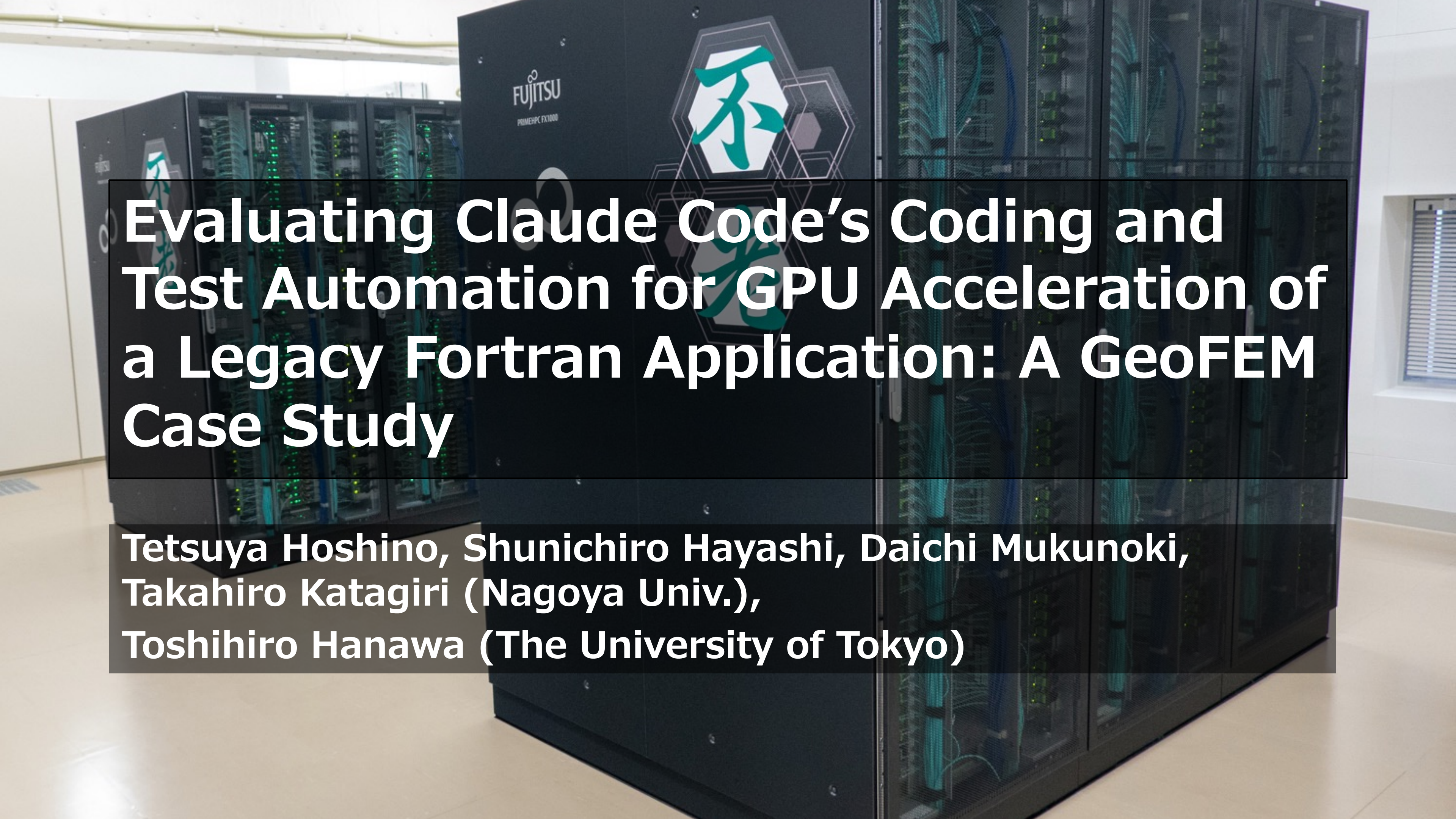
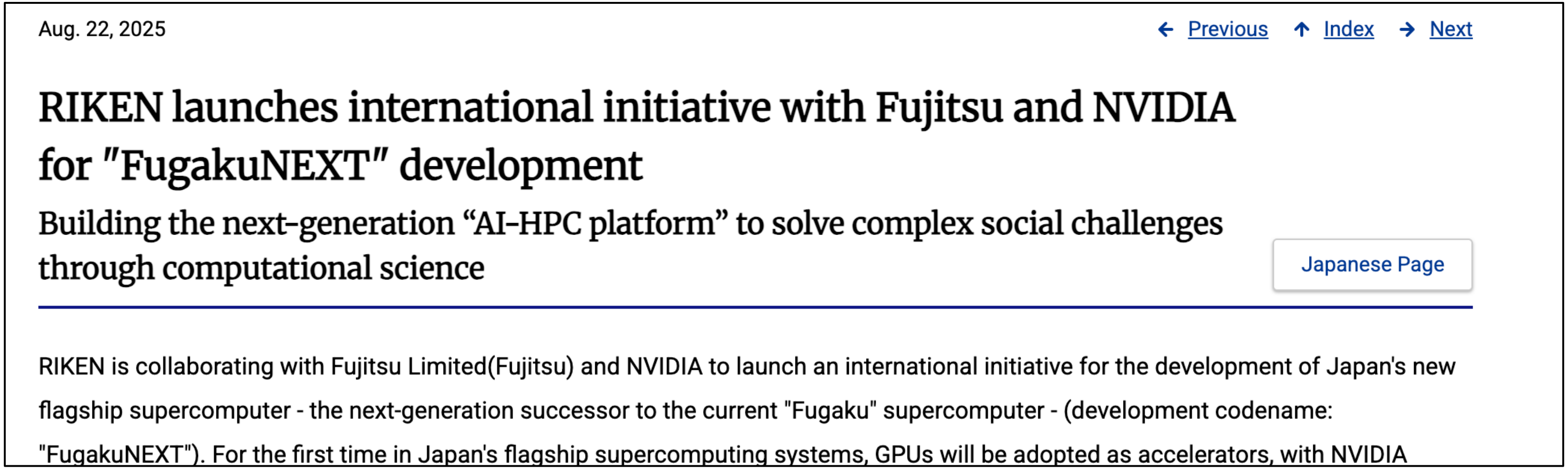


Evaluating Claude Code’s Coding and Test Automation for GPU Acceleration of a Legacy Fortran Application: A GeoFEM Case Study

Tetsuya Hoshino, Shunichiro Hayashi, Daichi Mukunoki, Takahiro Katagiri (Nagoya Univ.),
Toshihiro Hanawa (The University of Tokyo)

Background



Aug. 22, 2025

RIKEN launches international initiative with Fujitsu and NVIDIA for "FugakuNEXT" development

Building the next-generation “AI-HPC platform” to solve complex social challenges through computational science

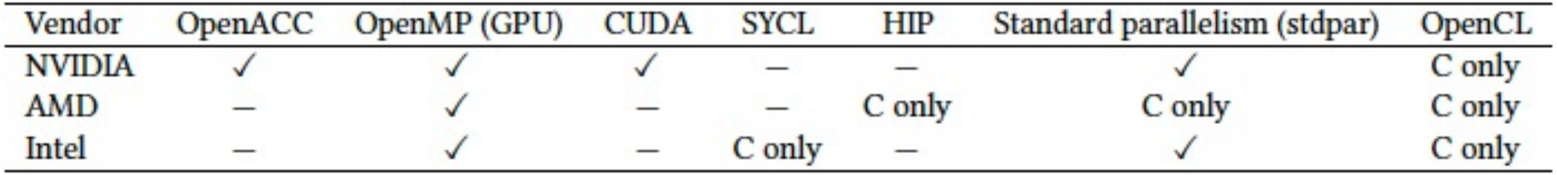
RIKEN is collaborating with Fujitsu Limited(Fujitsu) and NVIDIA to launch an international initiative for the development of Japan's new flagship supercomputer - the next-generation successor to the current "Fugaku" supercomputer - (development codename: "FugakuNEXT"). For the first time in Japan's flagship supercomputing systems, GPUs will be adopted as accelerators, with NVIDIA

- GPU porting of legacy applications is increasingly urgent, but progress is slow due to various challenges.
  - In particular, Fortran programs—common in legacy applications—remain a major challenge.

Challenges in GPU Porting

- Inconsistent support for parallel programming languages/models across vendors

Support status via each GPU vendor’s own compiler



Vendor	OpenACC	OpenMP (GPU)	CUDA	SYCL	HIP	Standard parallelism (stdpar)	OpenCL
NVIDIA	✓	✓	✓	–	–	✓	C only
AMD	–	✓	–	–	C only	C only	C only
Intel	–	✓	–	C only	–	✓	C only

- Increased maintenance cost
  - CPU version + GPU version × (# of programming languages/models)
    - In some cases, conversion to C is also required
- ↑ We want to leverage AI to deal with this
  - There are many success stories at the function level (e.g., BLAS)
  - But what about full HPC applications?

Key Features of Claude Code

- Claude Code
  - A CLI tool developed by Anthropic
  - Functions as an interactive AI assistant for code development
  - Can develop code with direct access to the file system
  - Integrates with large language models such as Claude Opus
    - Can be instructed in Japanese
- Claude Opus 4.1
  - Anthropic’s large language model released in Aug 2025
    - Latest is Opus 4.5 (Nov.28, 2025)
  - With Claude Code, can autonomously execute an end-to-end workflow from coding to test runs

↑ **We want to evaluate how useful it is for GPU code development**

Using Claude Code on a Supercomputer

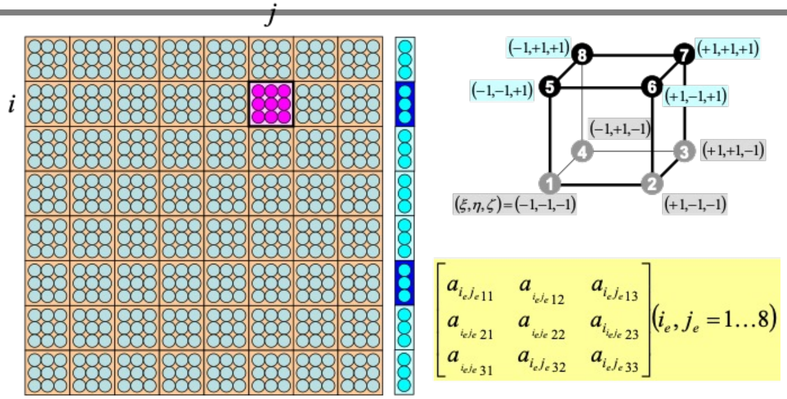
- Install and use on login and compute nodes
  - Inference runs on external servers (requires external network access)
  - Requires a subscription with Anthropic
    - Top personal plan: USD 200/month
- Directly edits and runs source code on the file system
  - By default, permissions are limited to the launch directory and below
  - Requests permission when editing or deleting files
    - Launching with --dangerously-skip-permissions skips permission prompts
  - Any command available under your user privileges can be used (e.g., submitting jobs with qsub)
    - Write instructions to a file; with --dangerously-skip-permissions to skip interaction, the code-development process itself can run as a batch job
- Inference is stochastic; the seed cannot be fixed

Evaluation Policy

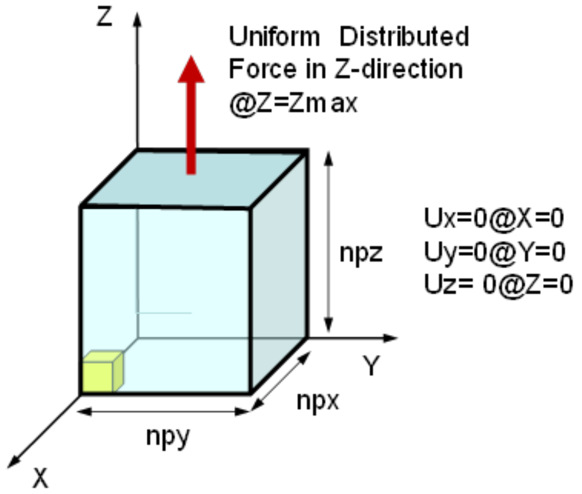
- Develop a GPU-enabled version of GeoFEM/Cube using Claude Code
  - GeoFEM／Cube
    - A Fortran-based application parallelized with MPI + OpenMP
    - It has optimization track records across various environments including GPUs (here we experiment from the basic MPI+OpenMP code)
- We evaluate:
  - Performance of the code generated by Claude Code
  - Time spent on Claude Code’s code-development process

GeoFEM／Cube

- Finite-volume solver for the uniform-field Poisson equation
  - Various optimization achievements in HPC environments
  - Grid: unstructured data structure; 7-point stencil
  - Linear system with an SPD sparse coefficient matrix
- Coloring and reordering
  - CM-RCM + Coalesced/Sequential
- Matrix storage schemes
  - CRS, Sliced-ELL, Sell-C-σ
- Main components: coefficient-matrix generation and the solver



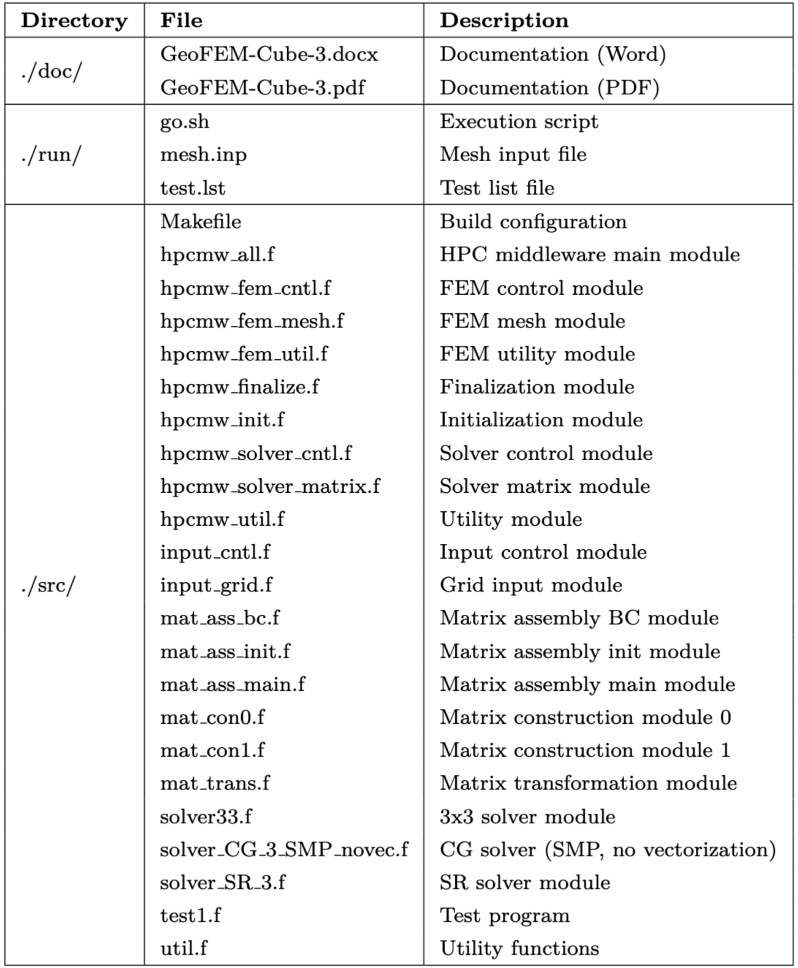
$$\begin{bmatrix} a_{i_ej_e11} & a_{i_ej_e12} & a_{i_ej_e13} \\ a_{i_ej_e21} & a_{i_ej_e22} & a_{i_ej_e23} \\ a_{i_ej_e31} & a_{i_ej_e32} & a_{i_ej_e33} \end{bmatrix} (i_e, j_e = 1\ldots8)$$



GeoFEM/Cube File Layout

- The table on the right was generated by asking Claude Code: “Convert the directory structure into a LaTeX table.”
- Fixed-form Fortran 90 (.f)
- Uses implicit real
- Parallelized with OpenMP
  - Coefficient-matrix generation files
    - mat_ass_*.f
  - Solver-related files
    - solver_*.f
- The doc folder includes application documentation in Word and PDF

表 2: Directory Structure of `GeoFEM-Cube-Hybrid_CG_3`



Directory	File	Description
./doc/	GeoFEM-Cube-3.docx	Documentation (Word)
	GeoFEM-Cube-3.pdf	Documentation (PDF)
./run/	go.sh	Execution script
	mesh.inp	Mesh input file
	test.lst	Test list file
./src/	Makefile	Build configuration
	hpcmw_all.f	HPC middleware main module
	hpcmw_fem_cntl.f	FEM control module
	hpcmw_fem_mesh.f	FEM mesh module
	hpcmw_fem_util.f	FEM utility module
	hpcmw_finalize.f	Finalization module
	hpcmw_init.f	Initialization module
	hpcmw_solver_cntl.f	Solver control module
	hpcmw_solver_matrix.f	Solver matrix module
	hpcmw_util.f	Utility module
	input_cntl.f	Input control module
	input_grid.f	Grid input module
	mat_ass_bc.f	Matrix assembly BC module
	mat_ass_init.f	Matrix assembly init module
	mat_ass_main.f	Matrix assembly main module
	mat_con0.f	Matrix construction module 0
	mat_con1.f	Matrix construction module 1
	mat_trans.f	Matrix transformation module
	solver33.f	3x3 solver module
	solver_CG_3_SMP_novec.f	CG solver (SMP, no vectorization)
	solver_SR_3.f	SR solver module
	test1.f	Test program
	util.f	Utility functions

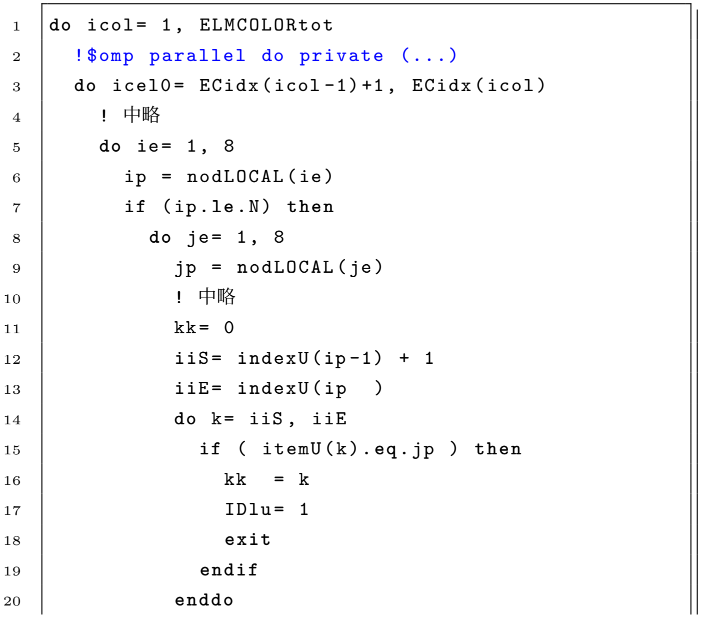
Coefficient-Matrix Generation

- The loop structure is very complex
- Just getting it to run in parallel is not hard
- But parallelizing efficiently on GPUs is not straightforward

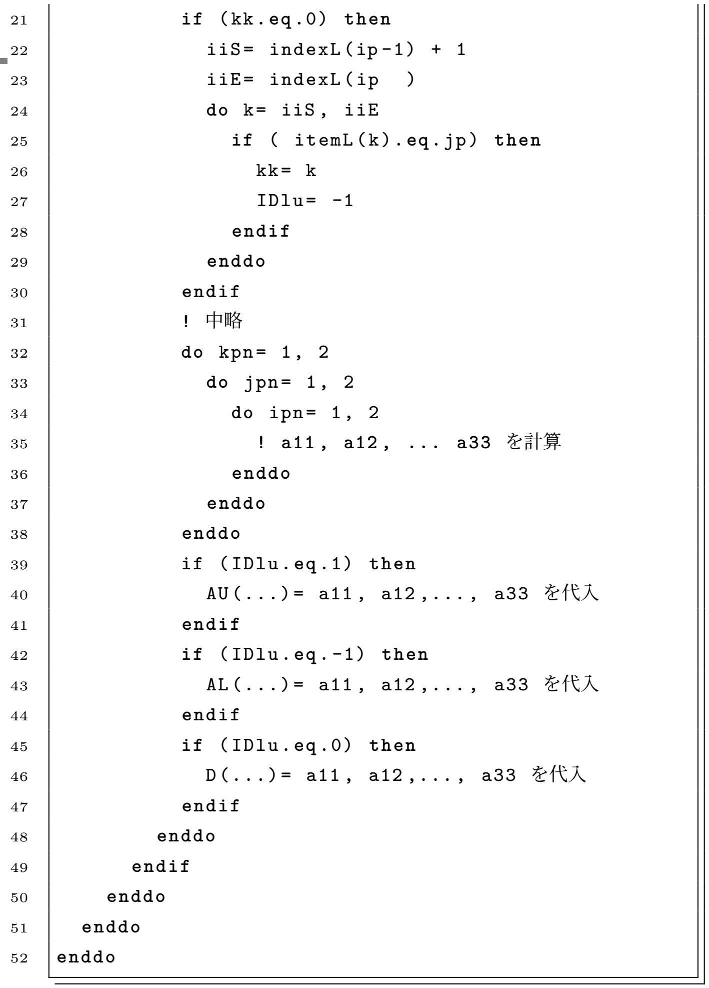
Parallelizable loops →

→

For efficient GPU parallelization, it’s typical to move the je loop right after the ie loop, fuse (ie, je), and parallelize the fused loop



```
do icol= 1, ELMCOLORtot
 !$omp parallel do private (...)
 do icel0= ECidx(icol-1)+1, ECidx(icol)
  ! 中略
  do ie= 1, 8
   ip = nodLOCAL(ie)
   if (ip.le.N) then
    do je= 1, 8
     jp = nodLOCAL(je)
     ! 中略
     kk= 0
     iiS= indexU(ip-1) + 1
     iiE= indexU(ip  )
     do k= iiS, iiE
      if ( itemU(k).eq.jp ) then
       kk   = k
       IDlu= 1
       exit
      endif
     enddo
```



```
     if (kk.eq.0) then
      iiS= indexL(ip-1) + 1
      iiE= indexL(ip  )
      do k= iiS, iiE
       if ( itemL(k).eq.jp) then
        kk= k
        IDlu= -1
       endif
      enddo
     endif
     ! 中略
     do kpn= 1, 2
      do jpn= 1, 2
       do ipn= 1, 2
        ! a11, a12, ... a33 を計算
       enddo
      enddo
     enddo
     if (IDlu.eq.1) then
      AU(...)= a11, a12,..., a33 を代入
     endif
     if (IDlu.eq.-1) then
      AL(...)= a11, a12,..., a33 を代入
     endif
     if (IDlu.eq.0) then
      D(...)= a11, a12,..., a33 を代入
     endif
    enddo
   endif
  enddo
 enddo
enddo
```

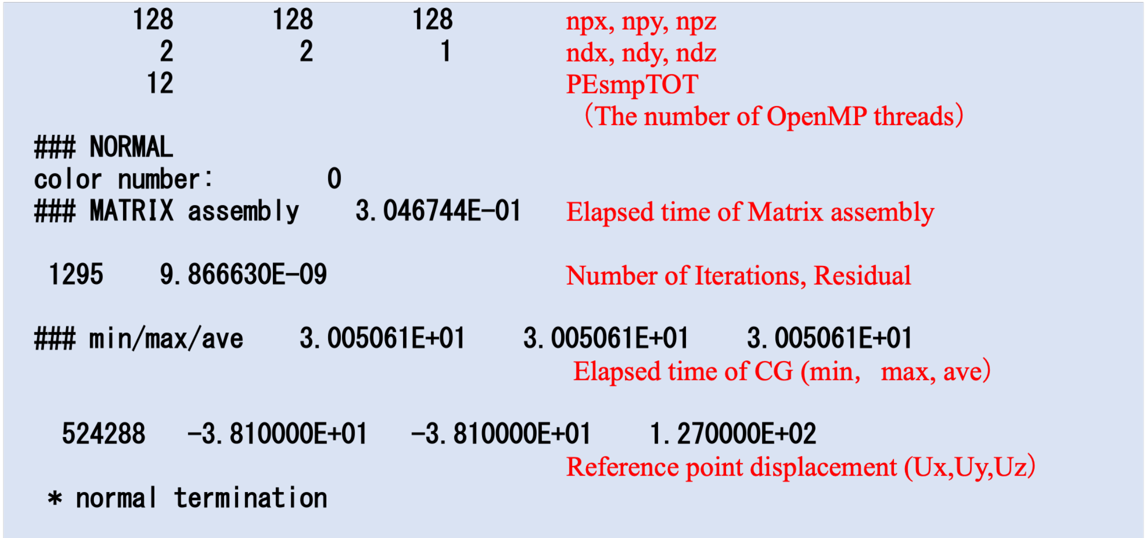
CG Solver

- Computation is a combination of standard linear algebra operations
  - SpMV, dot products, AXPY, etc.
  - Parallelization is easy: a mix of trivially parallel loops and reduction loops

```
Compute $r^{(0)}= b-[A]x^{(0)}$
for $i= 1, 2, …$
    solve $[M]z^{(i-1)}= r^{(i-1)}$
    $\rho_{i-1}= r^{(i-1)} z^{(i-1)}$
    if $i=1$
      $p^{(1)}= z^{(0)}$
    else
      $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
      $p^{(i)}= z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
    endif
    $q^{(i)}= [A]p^{(i)}$
    $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
    $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
    check convergence $|r|$
end
```

GeoFEM/Cube Output Example

- The following output example is taken from the Word file in the doc folder
  - In actual runs, only the black text is output
  - Unless explicitly told via prompt, Claude Code must interpret the Word file or infer the output from the code



```
         128         128         128          npx, npy, npz
           2           2           1          ndx, ndy, ndz
          12                                  PEsmpTOT
                                               (The number of OpenMP threads)
 ### NORMAL
 color number:         0
 ### MATRIX assembly    3.046744E-01    Elapsed time of Matrix assembly

  1295     9.866630E-09                       Number of Iterations, Residual

 ### min/max/ave    3.005061E+01    3.005061E+01    3.005061E+01
                                               Elapsed time of CG (min,   max, ave)

   524288   -3.810000E+01   -3.810000E+01    1.270000E+02
                                              Reference point displacement (Ux,Uy,Uz)
  * normal termination
```

Experiment Details

- Experimental Setup
  - With the same input source code and instruction, generate each case 10 times
  - Measure each generated program 5 times and take the median

- Targets
  - Input source code variants with “GPU-enable this code.”
    - .f: fixed-form Fortran 90, implicit real (original)
    - .f90: free-form Fortran 90, implicit none
    - .f90_woOMP: .f90 with OpenMP directives removed
  - .f90 as the input, target programming model is specified with prompt
    - omp: GPU porting using OpenMP target based on .f90
    - acc: GPU porting using OpenACC based on .f90
    - cuda: GPU porting using CUDA Fortran based on .f90
  - Further optimization
    - fast1: based on acc, instruct “make it faster”
    - fast2: based on fast1, instruct “make it even faster”
    - fast3: based on fast2, instruct “make it even faster”

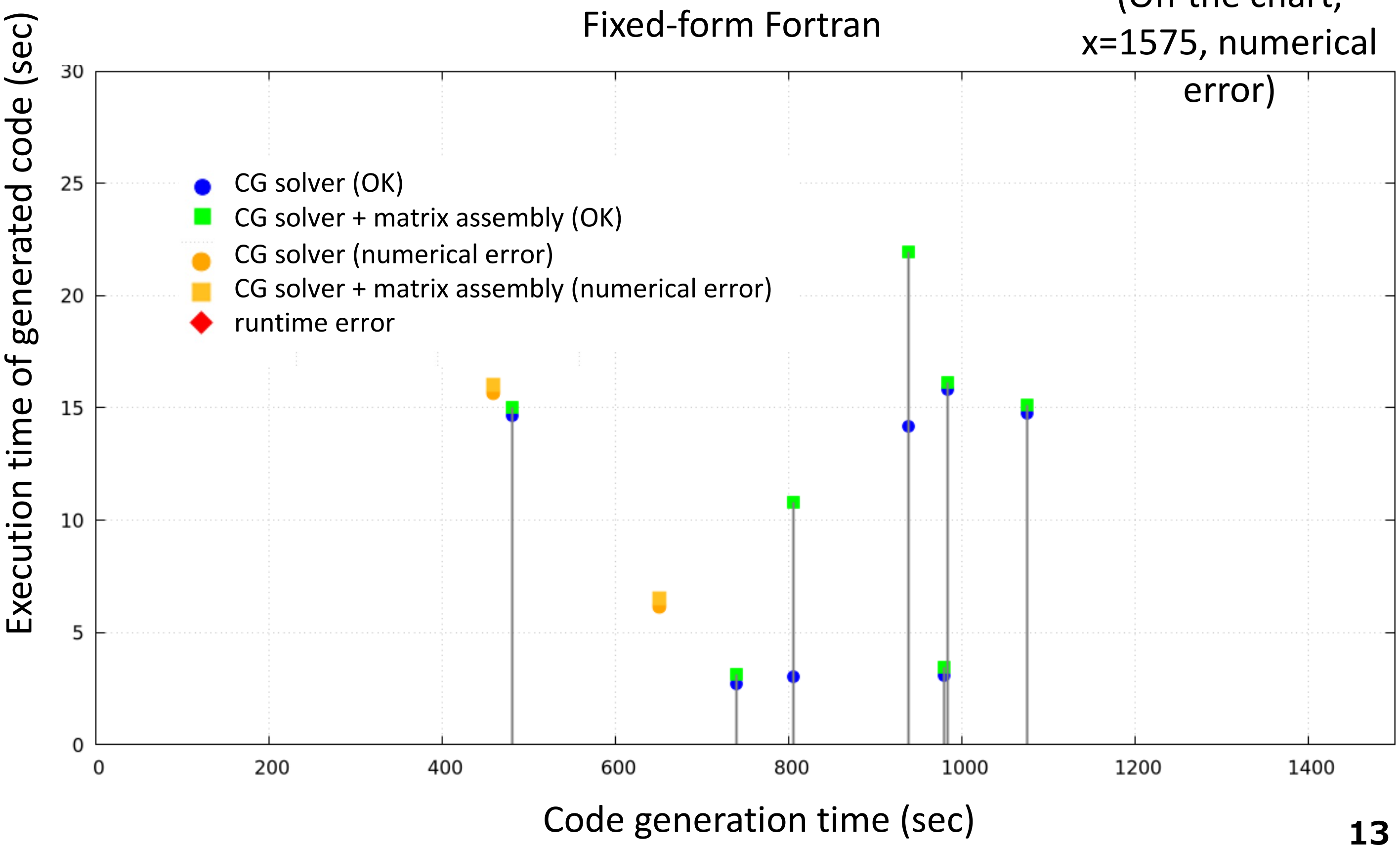
Evaluation Environment

- Miyabi (JCAHPC: Univ. of Tokyo & Univ. of Tsukuba)
  - GH200
  - CPU–GPU connected via NVLink C2C (450 GB/s per direction), cache-coherent

- Software environment
  - GPU compiler: nvfortran 24.9 (default)
    - Claude Code could have used other compilers, but in practice it used this for all cases
  - Claude Code ver.1.0.83 ～ 1.0.90
    - Updated frequently, so we could not pin a fixed version
    - All models were Claude Opus 4.1

Input source code variant (.f)

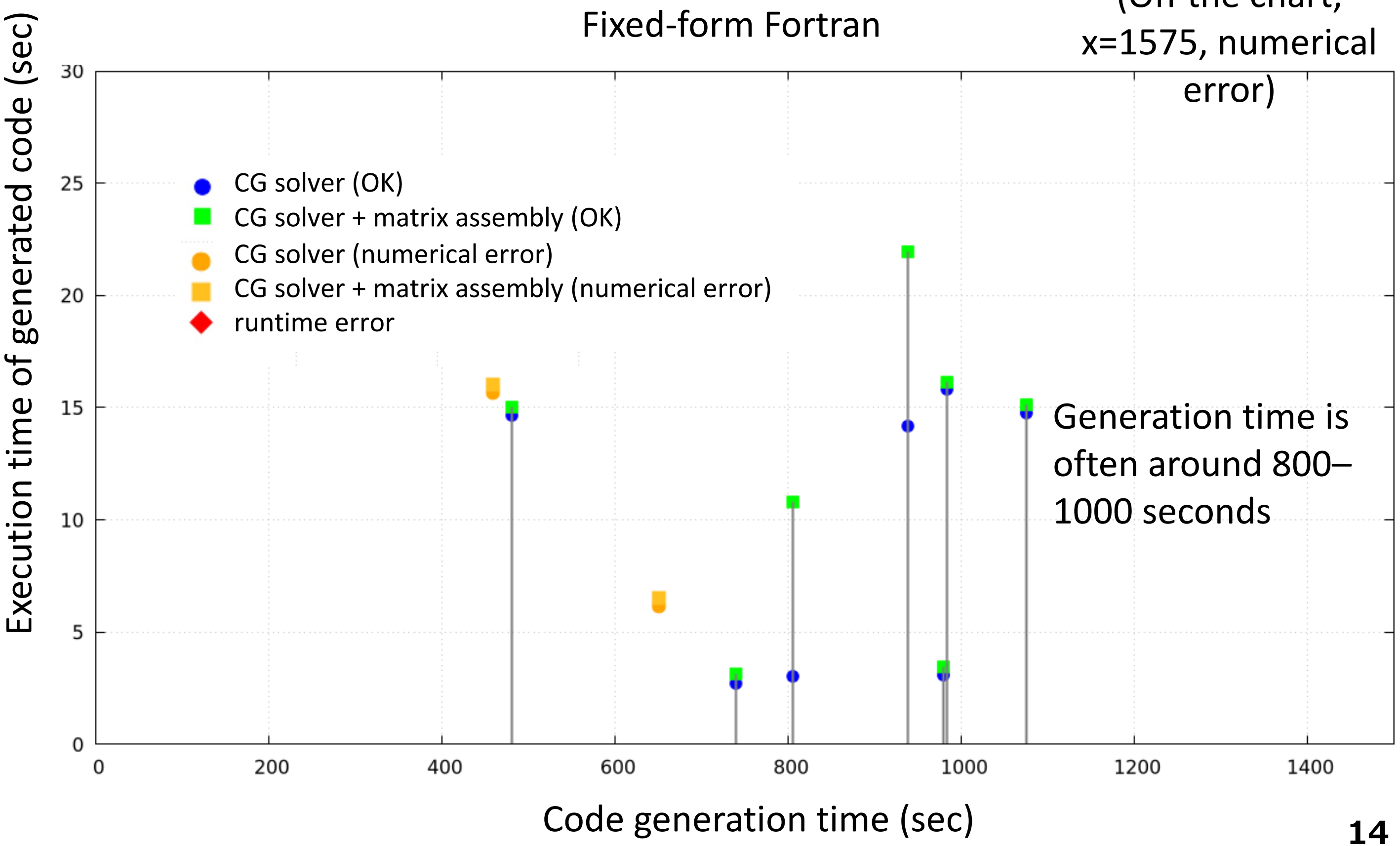
- Blue dots: solver time only
- Green dots: solver + matrix generation
- Yellow/red: did not run correctly
- At least, it judged from the code that the CG solver should be sped up, and it largely succeeded in GPU-ifying it



(Off the chart, x=1575, numerical error)

Input source code variant (.f)

- As with human coding, fixed-form line limits and implicit typing cause bugs, increasing generation time
- Off-chart points indicate it attempted matrix generation and ultimately failed



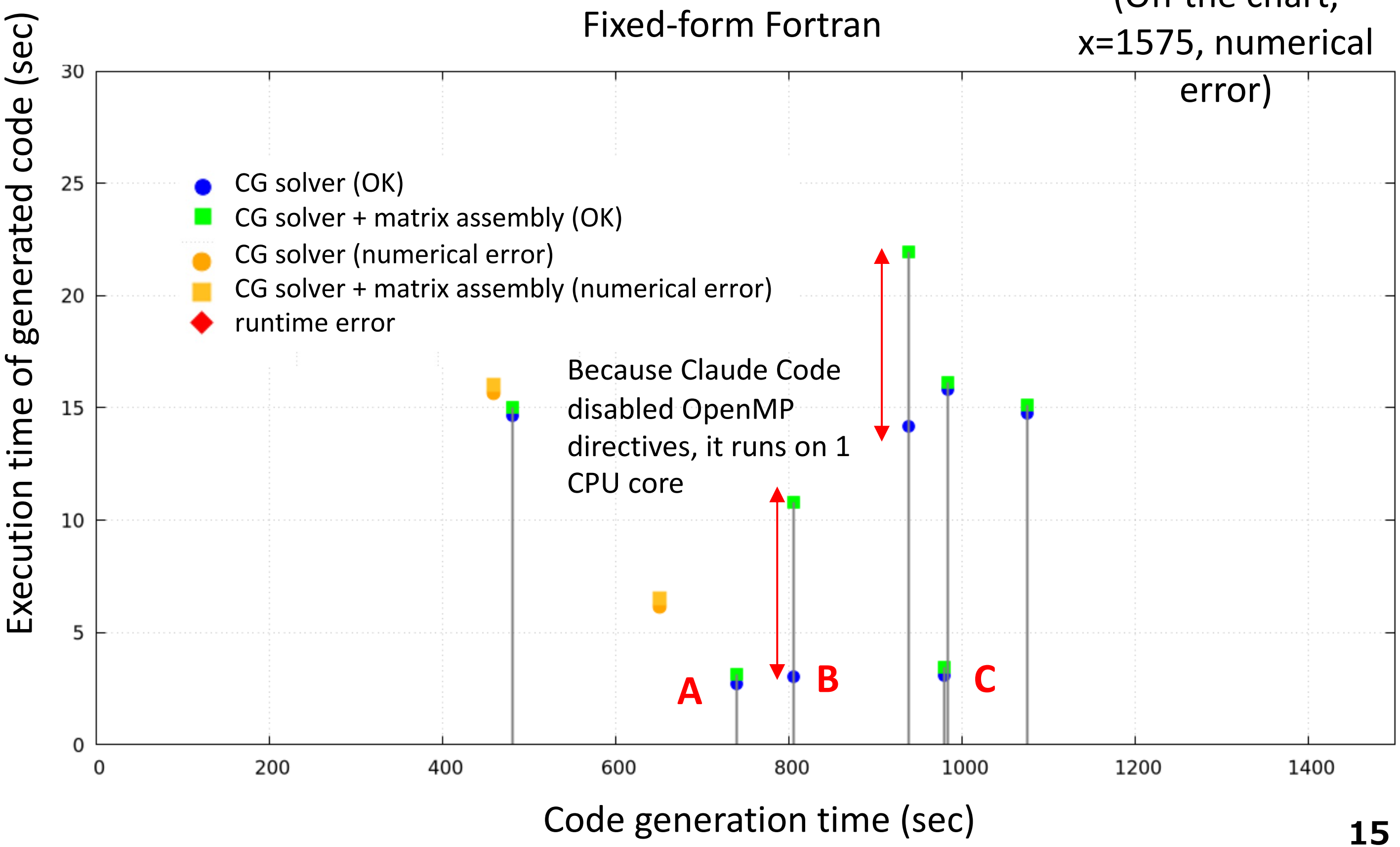
Fixed-form Fortran

(Off the chart, x=1575, numerical error)

Generation time is often around 800–1000 seconds

Input source code variant (.f)

- GPU porting of the matrix-generation part never succeeded
- A and C use OpenACC kernels directives and achieve reasonable performance for the solver part
- B uses OpenACC parallel directives and specifies clauses appropriately
- The others use parallel directives but without appropriate clauses

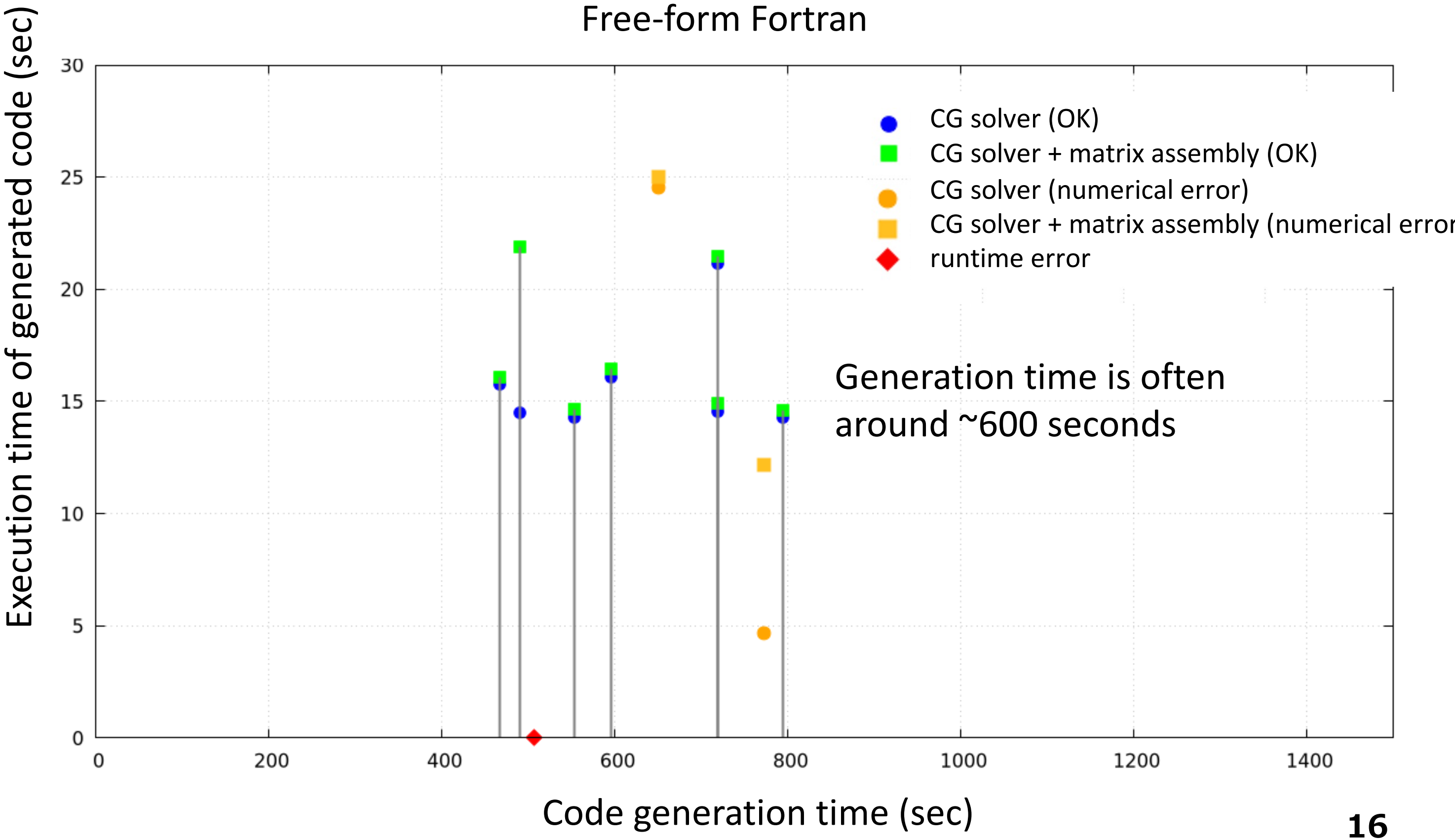


Because Claude Code disabled OpenMP directives, it runs on 1 CPU core

(Off the chart, x=1575, numerical error)

Input source code variant (.f90)

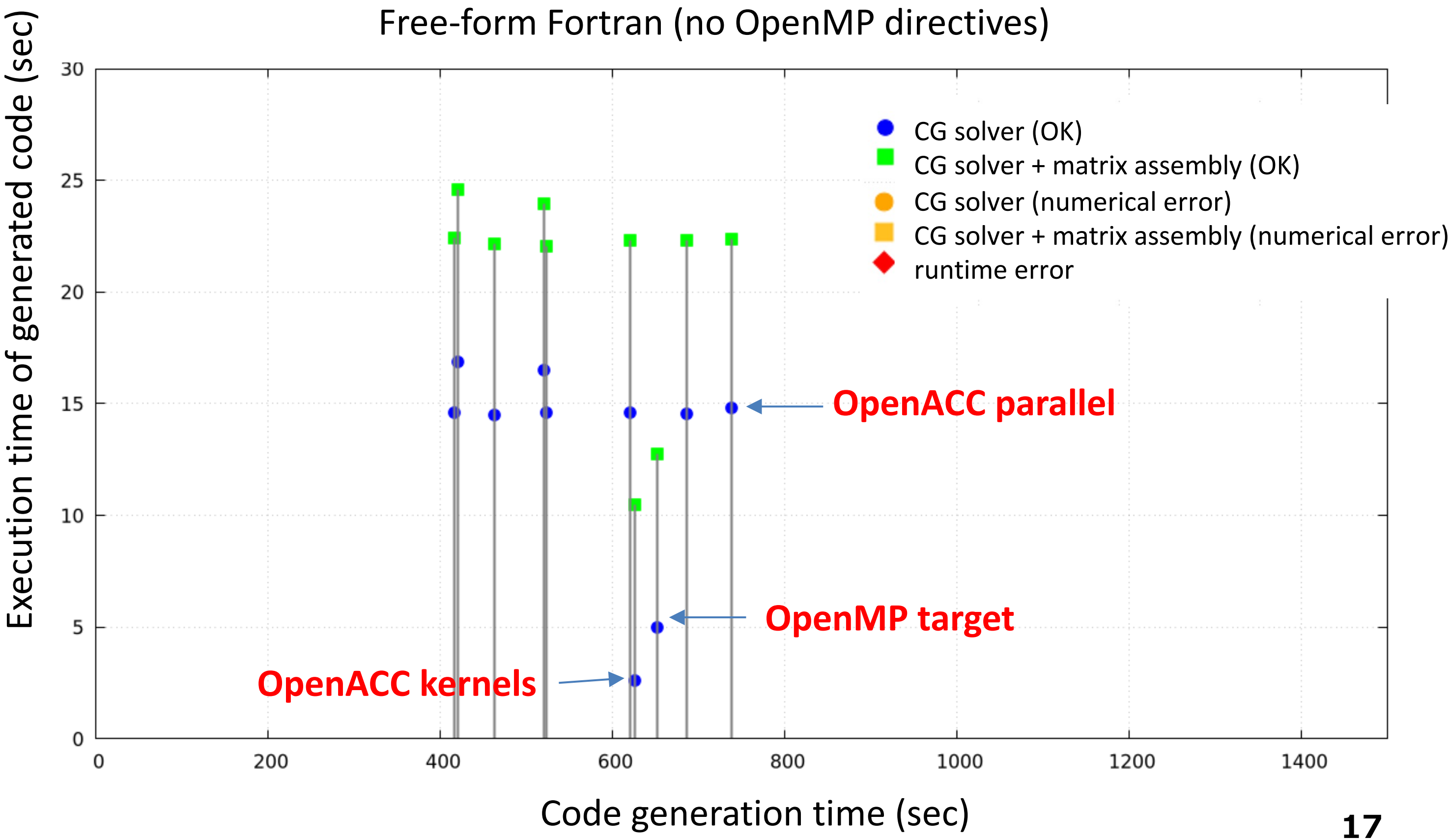
- Generation time is clearly shorter
- With .f it sometimes outputs performance-appropriate code, but with .f90 it does not (reason unknown)



Generation time is often around ~600 seconds

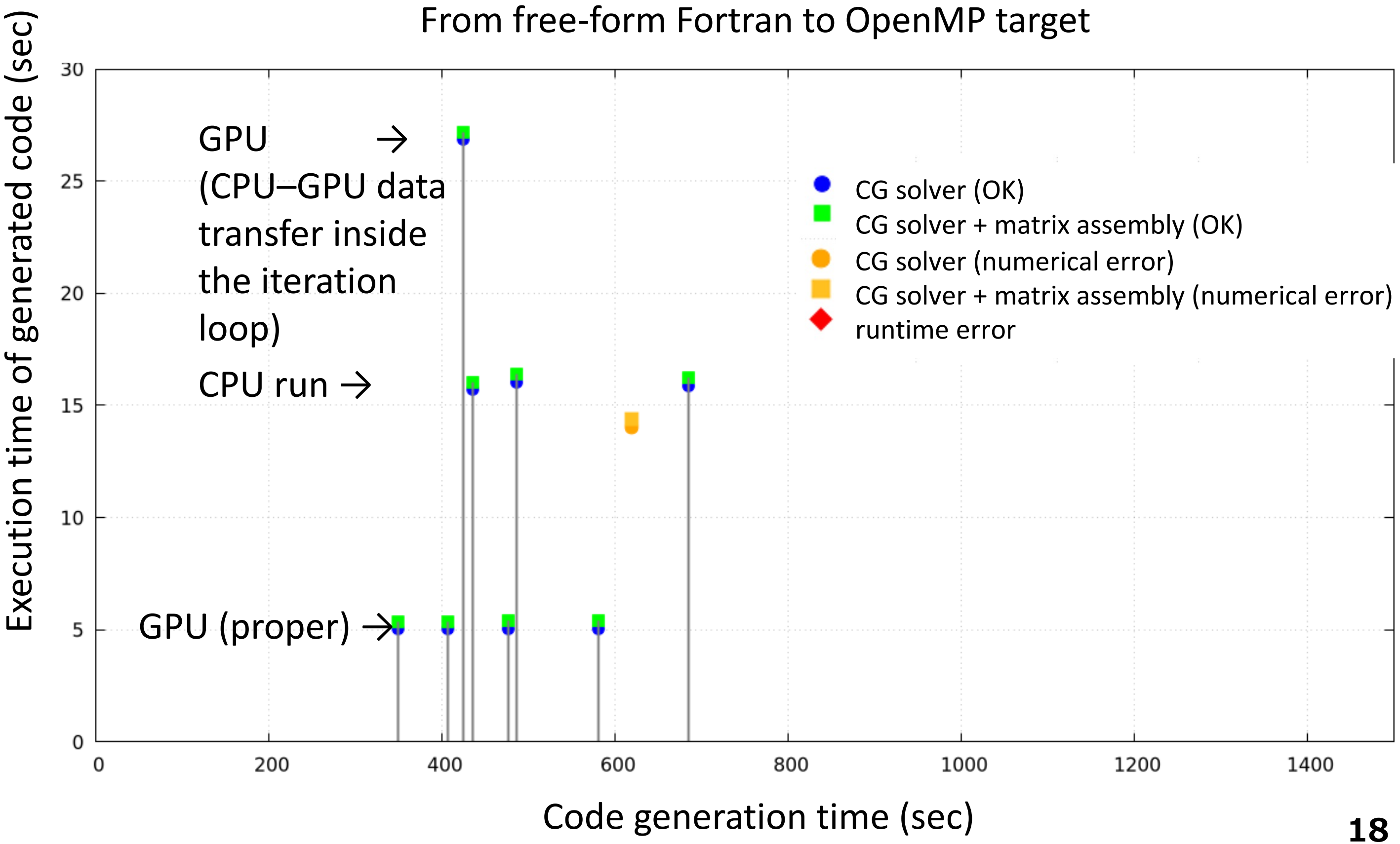
Input source code variant (.f90 without OpenMP directives)

- With OpenMP directives removed, it stopped trying to GPU-ify matrix generation; as a result, success rate increased and generation time decreased
- The matrix-generation part runs on 1 CPU core



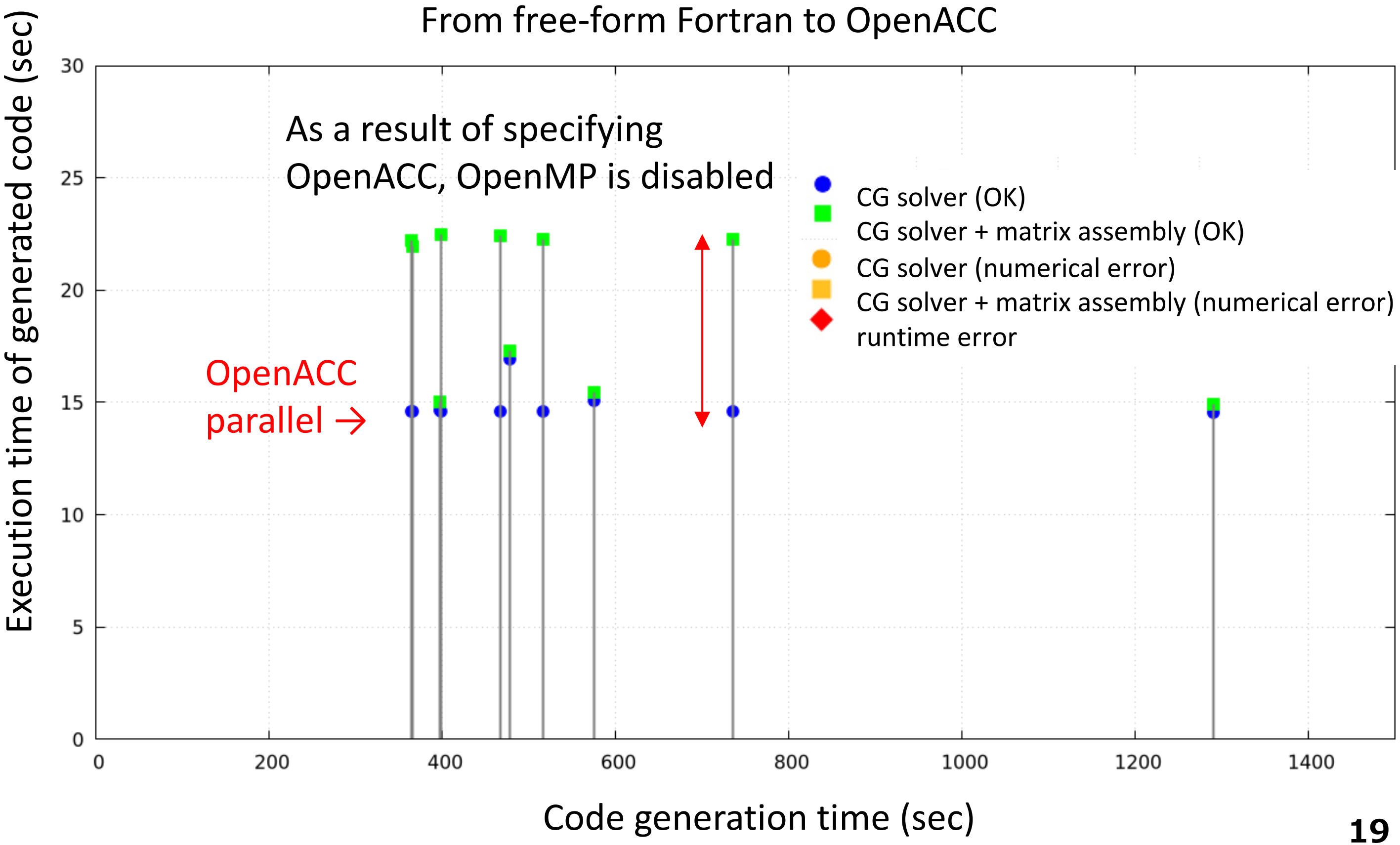
Target programming model (OpenMP target)

- Only 5 cases actually succeeded in running on the GPU
- Generation is generally fast



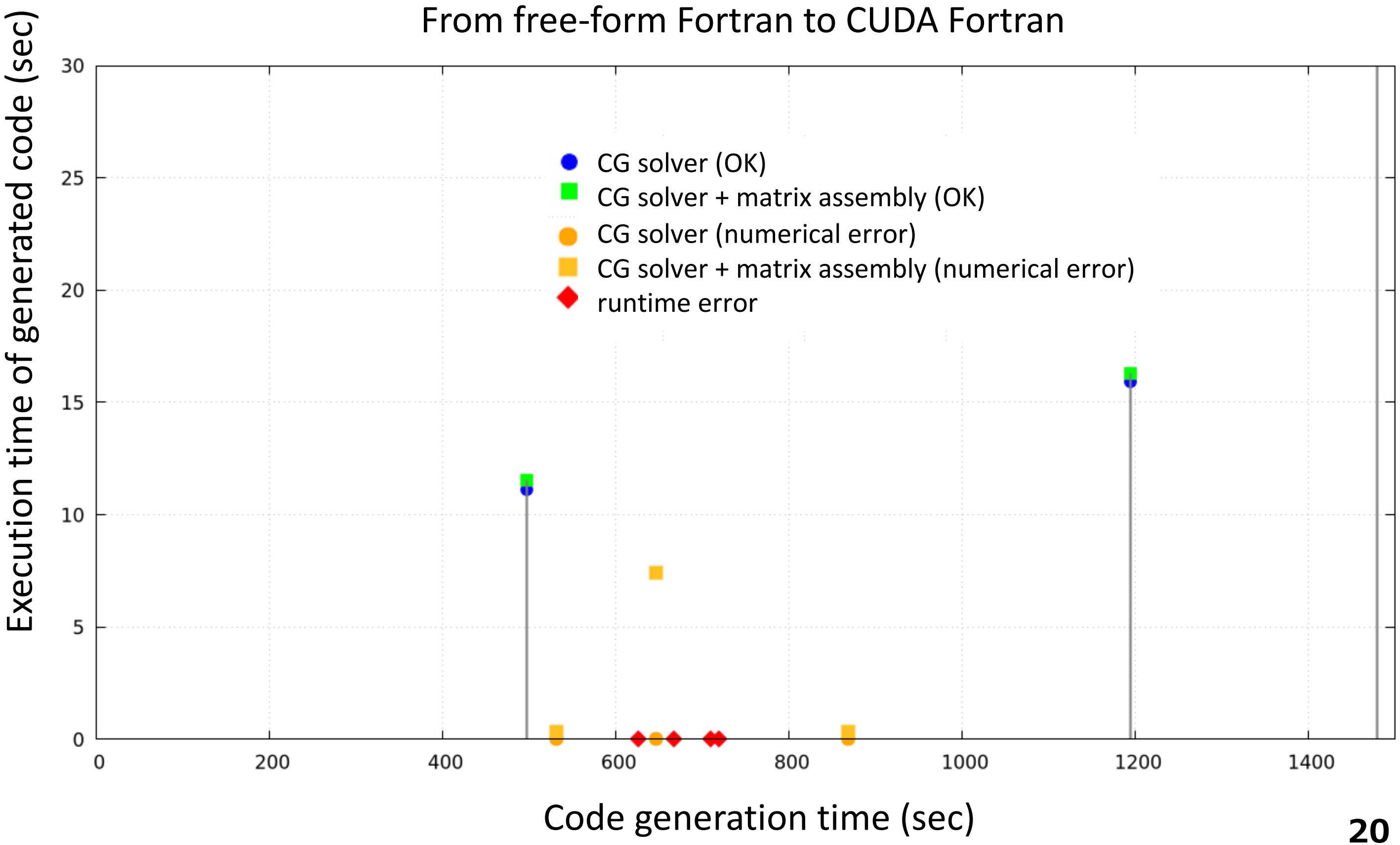
Target programming model (OpenACC)

- High likelihood of generating runnable GPU code
- All use OpenACC parallel directives; because clauses are not specified appropriately, performance is poor (reason unknown)
- Specifying OpenACC removed OpenMP flags from the Makefile, so much of matrix generation runs on 1 CPU core



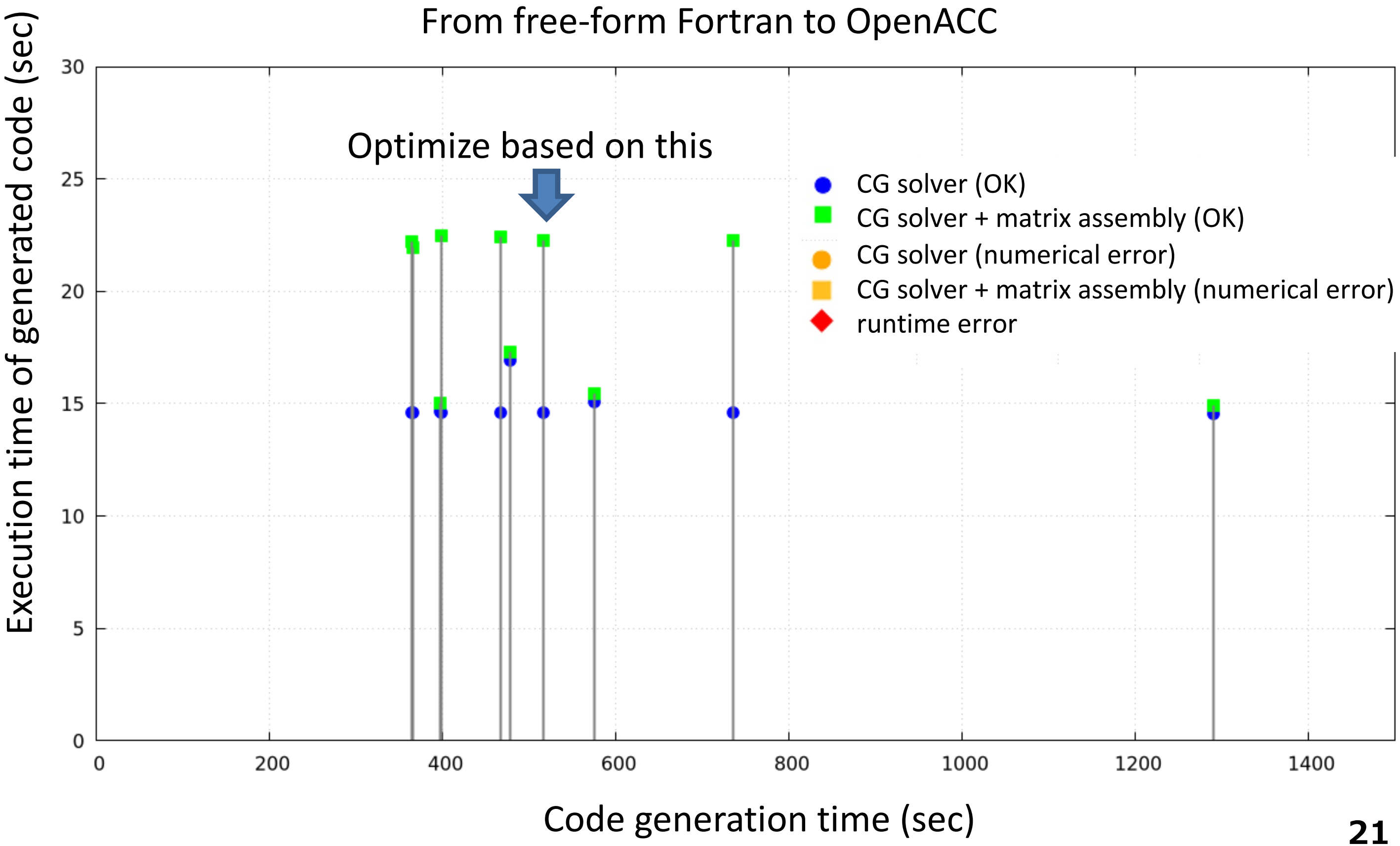
Target programming model (CUDA Fortran)

- Success rate is very low
- Even successful cases are not fast
- Switching the output to CUDA C might work better, but that is future work



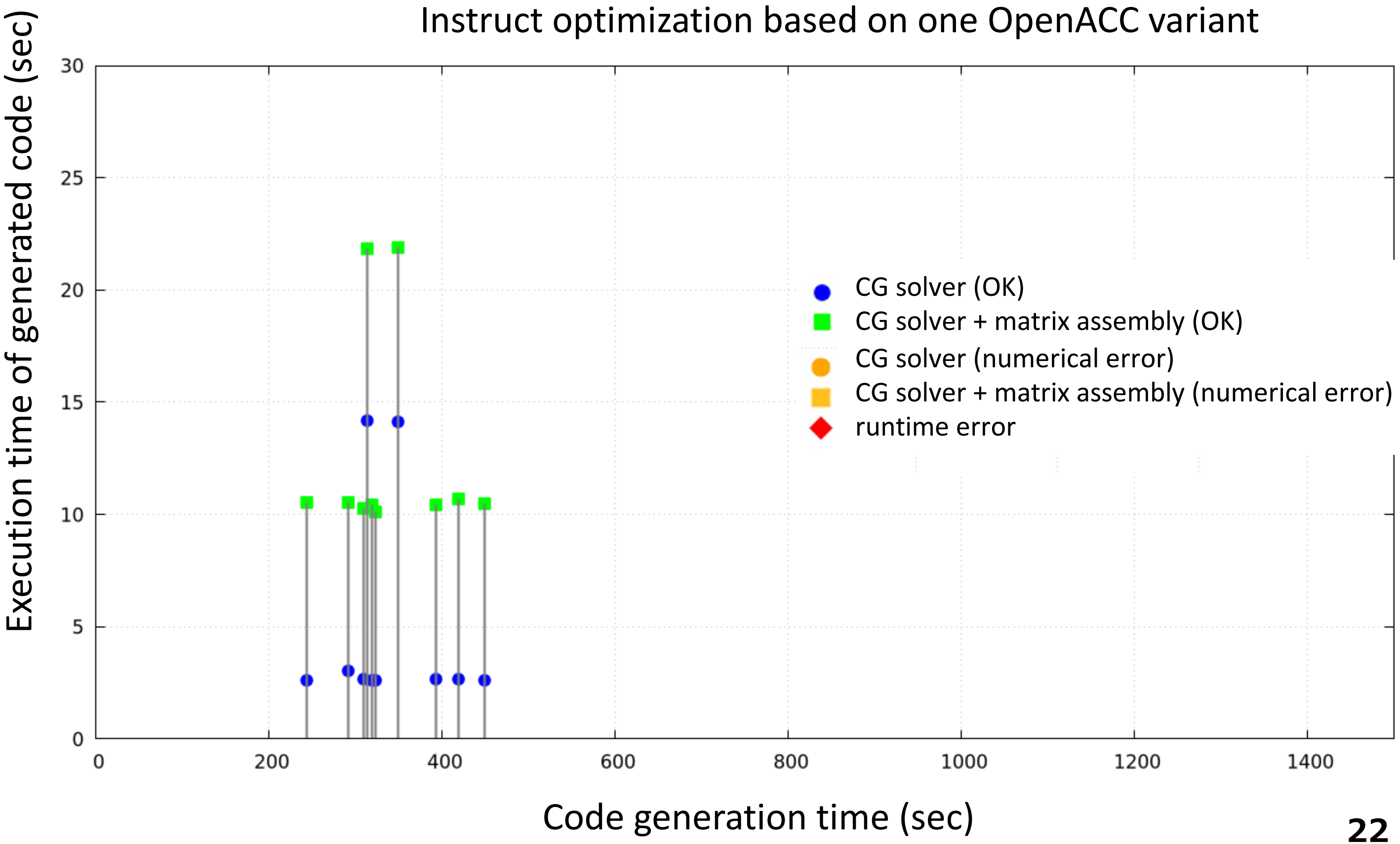
Further optimization (Level 0)

- Further optimization starting from slow code that uses OpenACC parallel directives with inappropriate clauses



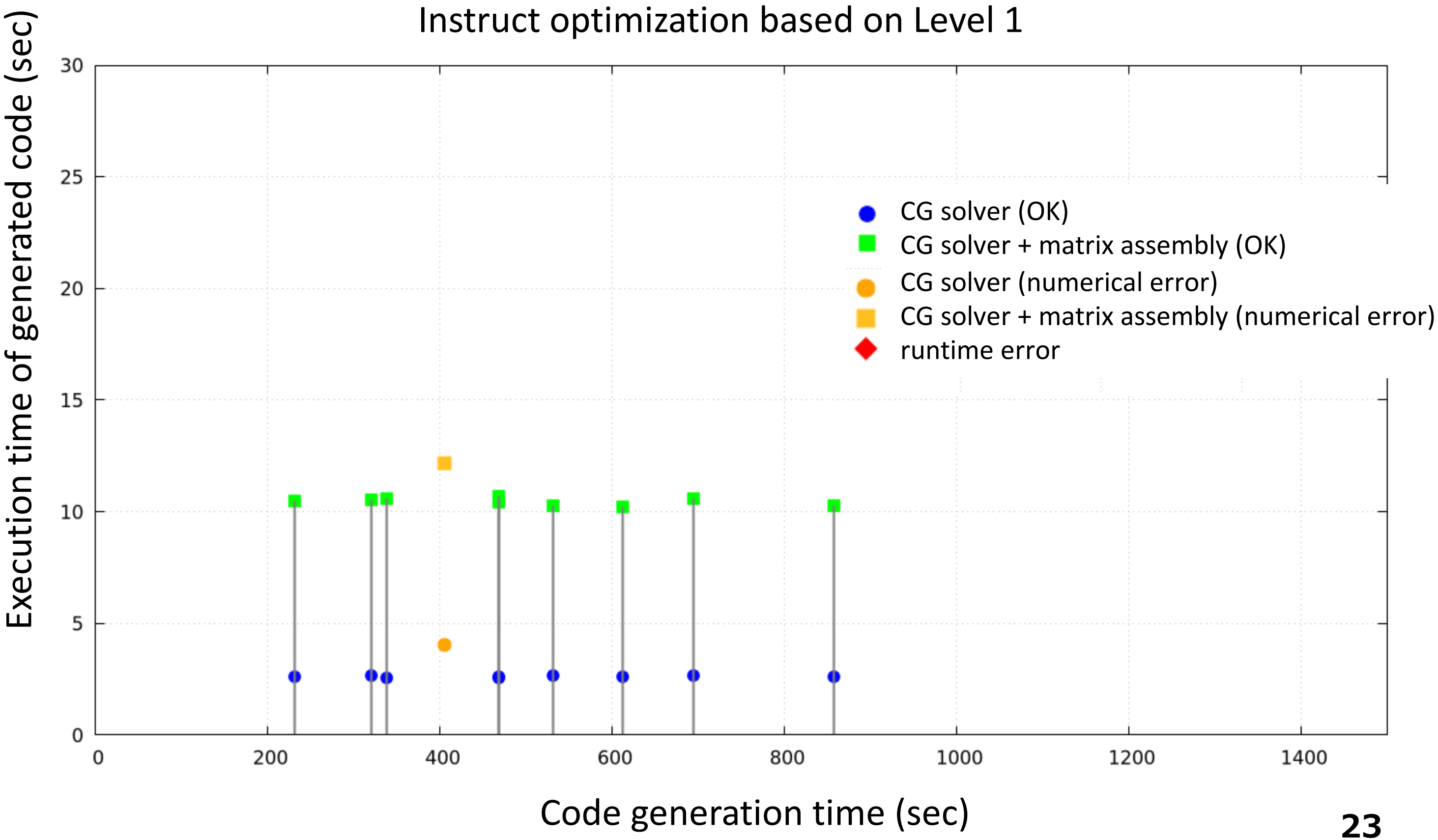
Further optimization (Level 1)

- In many cases, speedup was achieved by specifying parallel directive clauses appropriately



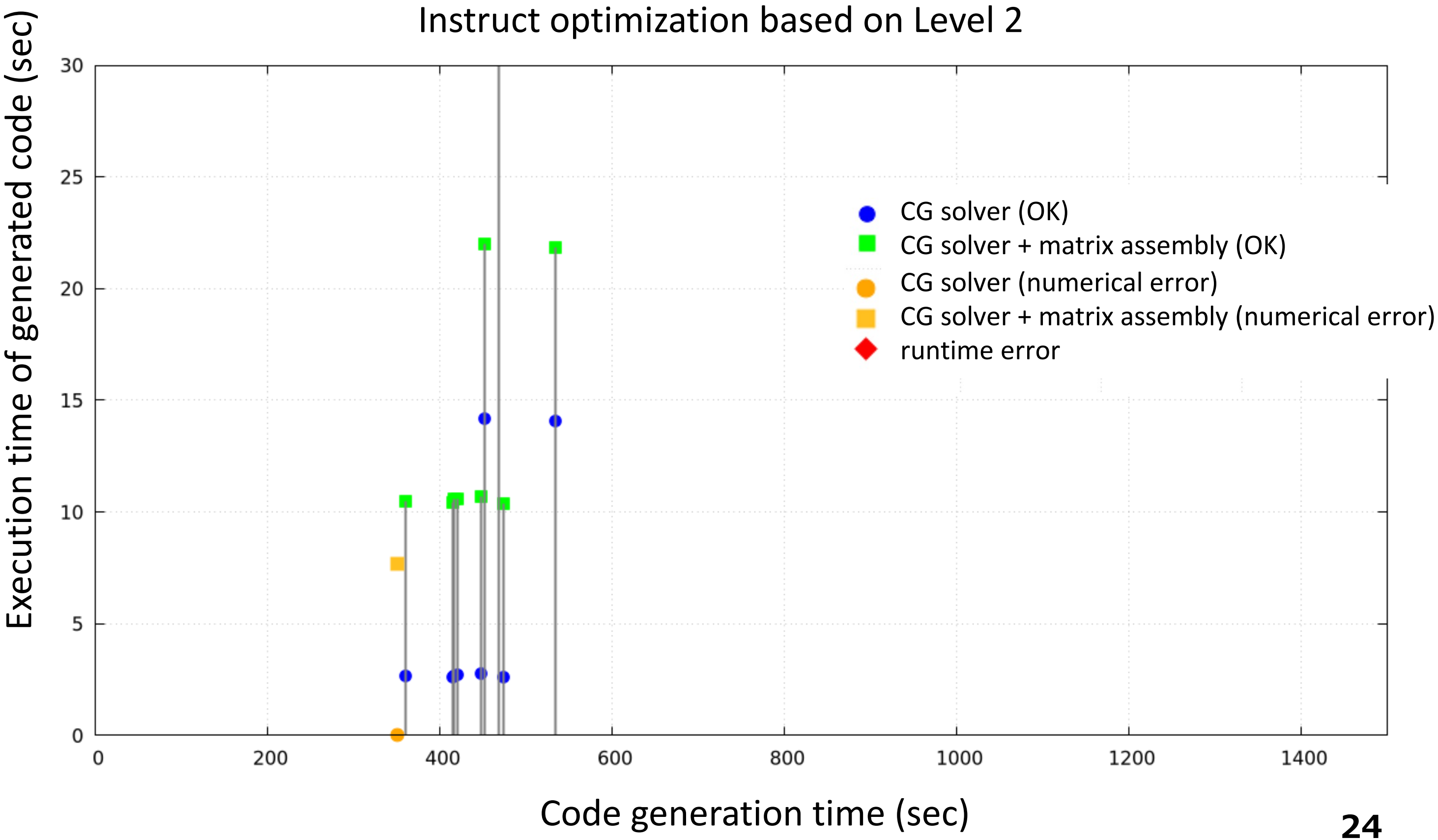
Further optimization (Level 2)

- In many cases, an async clause was added and performance improved slightly (a common technique)
- For the CG part, it successfully inserted OpenACC parallel + gang/vector + async clauses appropriately
- The matrix-generation part is not accelerated

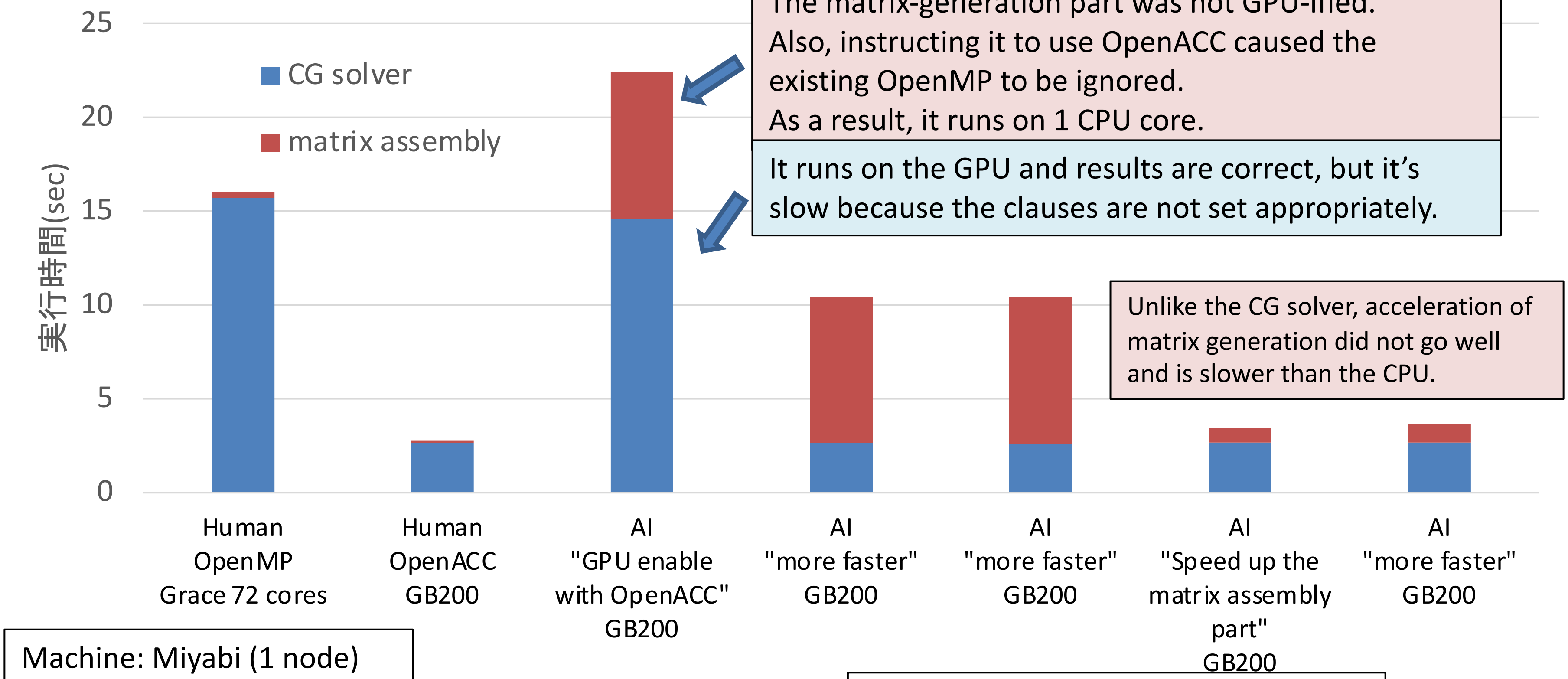


Further optimization (Level 3)

- Many cases failed and became slower
- Even though matrix generation clearly dominates runtime, no optimization was performed



GeoFEM GPU Porting by Claude Code Summary



The matrix-generation part was not GPU-ified.
Also, instructing it to use OpenACC caused the existing OpenMP to be ignored.
As a result, it runs on 1 CPU core.

It runs on the GPU and results are correct, but it’s slow because the clauses are not set appropriately.

Unlike the CG solver, acceleration of matrix generation did not go well and is slower than the CPU.

Machine: Miyabi (1 node)
Problem size: 128^3
Coloring/Reordering: RCM

Instructions like “make it even faster” tend to be effective

Discussion

- For the CG solver, it judged it should be accelerated regardless of OpenMP directives and generally succeeded in generating GPU code
  - Success rate is high with OpenACC
    - Likely because there is a lot of existing GPU-ported Fortran code
    - If you instruct “further optimization,” you generally get reasonably appropriate code

  → Because it already “knows” a parallel CG solver?
- For matrix generation, even with OpenMP directives present, parallelization is not done (or fails even if attempted)

  → Because it does not “know” a parallel matrix-generation implementation?

If it were a human,
“This loop can be parallelized” → apply directives
A code-generation AI,
“This is CG” → “the corresponding parallel CG is this”
…maybe that’s the difference?

Summary

- We evaluated an AI code generator’s ability to develop a GPU-enabled version based on the MPI+OpenMP parallelized, Fortran-based GeoFEM/Cube
- Even for the same computation, development time varied greatly depending on the input source variant (fixed-form vs free-form Fortran, etc.)
- The CG part was generated well, but the highly bespoke matrix-generation part did not go well
  - If instructed to speed up matrix generation, it can at least generate code that runs on the GPU
- Future work
  - Develop methods to improve GPU code-generation success for highly bespoke parts
    - E.g., add in-code guidance like “parallelize this loop and this loop”

Additional experiments beyond the paper: GPU porting of matrix generation

- Focus on matrix generation and instruct: “speed it up on the GPU”
  - Succeeded in generating code that can run on GPUs
  - But performance is slow
- Instruct further optimization
  - It created v1–v3 on its own, but it didn’t get faster
    - v1: simple parallelization
    - v2: loop unrolling (runnable but slow)
    - v3: memory-access optimization (compile error)

What’s the problem?

- The intent of the OpenMP version’s parallelization strategy was not conveyed
  - By coloring,
  - avoid write conflicts
- Doesn’t consider loop length (?)
  - 2 * 2 * 2 = 8 is too short for GPU thread-level parallelism

OpenMP version

```
do icol= 1, ELMCOLORtot
   !$omp parallel do private (…)
   do icel0= ELMCOLORindex(icol-1)+1, ELMCOLORindex(icol)
      …
      do ie= 1, 8
         ip = nodLOCAL(ie)
         if (ip.le.N) then
            do je= 1, 8
               jp = nodLOCAL(je)
                 …
               do kpn= 1, 2
                  do jpn= 1, 2
                     do ipn= 1, 2
                     …
                     enddo
                  enddo
               enddo
               …
               if (IDlu.eq.1) then
                  AU(9*kk-8)= AU(9*kk-8) + a11
               endif
               …
            enddo
         endif
enddo; enddo; enddo
```

What’s the problem?

- Loop-parallelization strategy is not great
  - Pattern: simply parallelize OpenMP-parallel loops with (gang, vector)
  - Or: use gang for OpenMP-parallel loops and collapse(3) the innermost 2×2×2 loop with vector
- Unnecessary atomic operations
  - If the ie/je loops are not parallelized, coloring avoids conflicts

OpenACC version generated by Claude Code

```
do icol= 1, ELMCOLORtot
   !$acc parallel loop gang vector private(…)
   do icel0= ELMCOLORindex(icol-1)+1, ELMCOLORindex(icol)
      …
      do ie= 1, 8
         ip = nodLOCAL(ie)
         if (ip.le.N) then
            do je= 1, 8
               jp = nodLOCAL(je)
                 …
               do kpn= 1, 2
                  do jpn= 1, 2
                     do ipn= 1, 2
                     …
                     enddo
                  enddo
               enddo
               …
               if (IDlu.eq.1) then
                  !$acc atomic update
                  AU(9*kk-8)= AU(9*kk-8) + a11
               endif
               …
            enddo
         endif
enddo; enddo; enddo
```

What’s the problem?

- Reorder loops so ie/je can be collapsed, then vector-parallelize

- Then use atomic operations

It’s important to teach the code-generation AI the preconditions for (non-)parallelizability

Desired OpenACC version

```
do icol= 1, ELMCOLORtot
   !$acc parallel num_gangs(…) vector_length(64) loop gang private(…)
   do icel0= ELMCOLORindex(icol-1)+1, ELMCOLORindex(icol)
      …
      !$acc loop collapse(2) vector(64)
      do ie= 1, 8
         do je= 1, 8
            ip = nodLOCAL(ie)
            if (ip.le.N) then
               jp = nodLOCAL(je)
                 …
               do kpn= 1, 2
                  do jpn= 1, 2
                     do ipn= 1, 2
                     …
                     enddo
                  enddo
               enddo
               …
               if (IDlu.eq.1) then
                  !$acc atomic update
                  AU(9*kk-8)= AU(9*kk-8) + a11
               endif
               …
            enddo
         endif
```