GPU Performance Comparison of Preconditioned Krylov Solvers

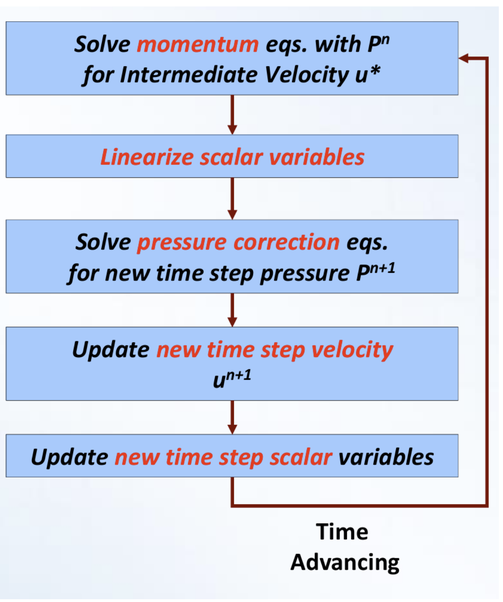
Ki-Ha Kim, Young Kwang Hwang, Jong-Chan Kim

Dept. of Supercomputing Acceleration Research, Korea Institute of Science and Technology Information

Abstract

A three-dimensional two-phase thermal–hydraulic simulation code is employed for nuclear reactor safety analysis. In this framework, the pressure field is obtained from a pressure-correction equation that forms a globally coupled sparse linear system. As the pressure solution accounts for over 80% of the total runtime, it represents the major computational bottleneck. To address this limitation, the present work accelerates the pressure solver on a single-GPU architecture by implementing and comparing three preconditioning strategies: diagonal, ILU(0), and AMG. The BiCGStab solver components were ported to CUDA using cuSPARSE and cuBLAS libraries. Performance was evaluated on KISTI's supercomputer comparing a 64-core Xeon Phi 7250 against an NVIDIA GH200 GPU. Results show that the diagonal preconditioner achieves the best balance between convergence and efficiency due to its fully parallel structure, while ILU(0) is limited by sparse triangular solves. The AMG solver converges rapidly but requires costly setup at each time step, reducing its overall benefit.

Introduction



Momentum (calc_momentum)

$$\begin{pmatrix}\vec{u}_g^{\,*}\\ \vec{u}_l^{\,*}\\ \vec{u}_d^{\,*}\end{pmatrix}=\begin{pmatrix}m_{gg} & m_{gl} & m_{gd}\\ m_{lg} & m_{ll} & m_{ld}\\ m_{dg} & m_{dl} & m_{dd}\end{pmatrix}^{-1}\left[\begin{pmatrix}\vec{s}_g\\ \vec{s}_l\\ \vec{s}_d\end{pmatrix}-\begin{pmatrix}V\alpha_g\\ V\alpha_l\\ V\alpha_d\end{pmatrix}\nabla P^n\right]$$

Scalar (calc_scalar): mass+energy

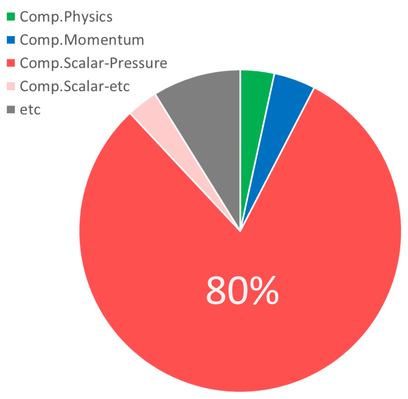
$$\mathbf{AX}=\mathbf{B}+\sum_f \mathbf{G}\psi_{g,f}^{n+1}+\sum_f \mathbf{L}\psi_{l,f}^{n+1}+\sum_f \mathbf{D}\psi_{d,f}^{n+1}$$

$$\mathbf{A}=\begin{pmatrix}a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16}\\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26}\\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36}\\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46}\\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56}\\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66}\end{pmatrix}\quad \mathbf{X}=\begin{pmatrix}\delta e_g\\ \delta e_l\\ \delta\alpha_g\\ \delta\alpha_d\\ \delta X_n\\ \delta P\end{pmatrix},\mathbf{B}=\begin{pmatrix}b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\end{pmatrix},\mathbf{G}=\begin{pmatrix}g_{1,f}\\ g_{2,f}\\ g_{3,f}\\ g_{4,f}\\ g_{5,f}\\ g_{6,f}\end{pmatrix},\mathbf{L}=\begin{pmatrix}l_{1,f}\\ l_{2,f}\\ l_{3,f}\\ l_{4,f}\\ l_{5,f}\\ l_{6,f}\end{pmatrix},\mathbf{D}=\begin{pmatrix}d_{1,f}\\ d_{2,f}\\ d_{3,f}\\ d_{4,f}\\ d_{5,f}\\ d_{6,f}\end{pmatrix}$$

Pressure correction

$$\begin{pmatrix}\vec{u}_g^{\,n+1}\\ \vec{u}_l^{\,n+1}\\ \vec{u}_d^{\,n+1}\end{pmatrix}=\begin{pmatrix}\vec{u}_g^{\,*}\\ \vec{u}_l^{\,*}\\ \vec{u}_d^{\,*}\end{pmatrix}-\begin{pmatrix}m_{gg} & m_{gl} & m_{gd}\\ m_{lg} & m_{ll} & m_{ld}\\ m_{dg} & m_{dl} & m_{dd}\end{pmatrix}^{-1}\begin{pmatrix}V\alpha_g\\ V\alpha_l\\ V\alpha_d\end{pmatrix}\nabla\delta P$$

$A\delta P = b + H(u^*, \delta P)$



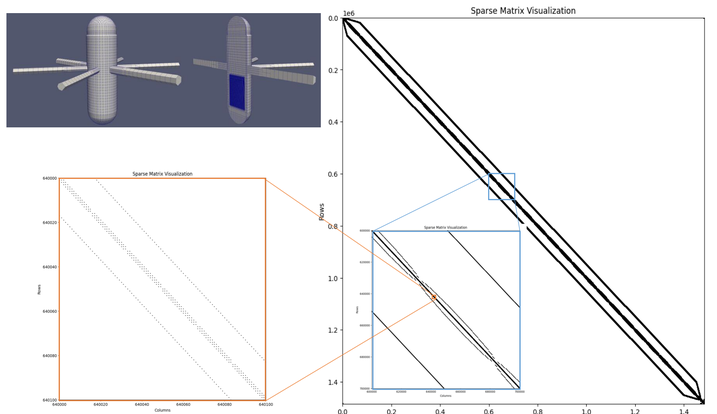
A three-dimensional two-phase thermal–hydraulic code is employed for nuclear reactor safety analysis. It employs unstructured meshes for complex geometries and MPI-based domain decomposition for parallel execution. The simulation proceeds through momentum calculation, scalar transport (mass and energy), and pressure correction stages. Among these, **the pressure correction step dominates the computational cost, accounting for over 80% of the total runtime.** The pressure field is obtained by solving a **globally coupled sparse linear system** derived from the pressure-correction equation. This system must be solved at every time step, making efficient solver design essential for large-scale transient simulations.

Method

Pressure Equation as Sparse Linear System

$$A\delta P = b + H(u^*, \delta P)$$

$$\begin{pmatrix}\left[1+\sum_f \mathbf{CC}_6\right]_1 & \cdots & (\mathbf{CC}_6)_{1,j} & \cdots & (\mathbf{CC}_6)_{1,N}\\ \vdots & \ddots & & & \vdots\\ (\mathbf{CC}_6)_{i,1} & \cdots & \left[1+\sum_f \mathbf{CC}_6\right]_i & \cdots & (\mathbf{CC}_6)_{i,N}\\ \vdots & & & \ddots & \vdots\\ (\mathbf{CC}_6)_{N,1} & \cdots & (\mathbf{CC}_6)_{N,j} & \cdots & \left[1+\sum_f \mathbf{CC}_6\right]_N\end{pmatrix}\begin{pmatrix}\delta P_1\\ \vdots\\ \delta P_i\\ \vdots\\ \delta P_N\end{pmatrix}=\begin{pmatrix}(\mathbf{BB}_6)_1\\ \vdots\\ (\mathbf{BB}_6)_i\\ \vdots\\ (\mathbf{BB}_6)_N\end{pmatrix}$$



GPU Acceleration of Preconditioned Krylov Solver

comm(x_0)
1. $r_0 = b - Ax_0$ ← Matmul
2. Choose an arbitrary vector $\hat{r}_0$ such that $(\hat{r}_0, r_0) \neq 0$, e.g., $\hat{r}_0 = r_0$
3. $\rho_0 = (\hat{r}_0, r_0)$ ← InnerProduct
4. $p_0 = r_0$
5. For $i = 1, 2, 3, \ldots$

comm(y)
1. $y = K_2^{-1}K_1^{-1}p_{i-1}$ ← Preconditioner
2. $v = Ay$ ← Matmul
3. $\alpha = \rho_{i-1}/(\hat{r}_0, v)$ ← InnerProduct
4. $h = x_{i-1} + \alpha y$
5. $s = r_{i-1} - \alpha v$ ← Projection x2
6. If h is accurate enough then $x_i = h$ and quit

comm(z)
7. $z = K_2^{-1}K_1^{-1}s$ ← Preconditioner
8. $t = Az$ ← Matmul
9. $\omega = (K_1^{-1}t, K_1^{-1}s)/(K_1^{-1}t, K_1^{-1}t)$ ← InnerProduct x2
10. $x_i = h + \omega z$
11. $r_i = s - \omega t$ ← Projection x2
12. If x_i is accurate enough then quit
13. $\rho_i = (\hat{r}_0, r_i)$ ← InnerProduct
14. $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega)$
15. $p_i = r_i + \beta(p_{i-1} - \omega v)$ ← Projection

Matmul — **Sparse matrix vector mul.**

```
kkh_cuspmv(Ax, matA, x, handle_spmv, dBuffer, n);
  ↳ cusparseSpMV(handle_spmv,…, matA ,…, dBuffer);
```

InnerProduct — **Reduce sum**

```
cublasDdot(handle_dotp, n, r, 1, r, 1, &rho_a);
```

Projection — **Elementwise vector add**

```
kkh_cuvecadd<<<blocksPerGrid, threadsPerBlock>>>
```

Preconditioner — **(1) Diagonal, (2) ILU, (3) AMG**

(1) Diagonal

```
__global__ void kkh_cuvecmul(double *c, double *a, double *b, int n)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) {
        c[idx] = a[idx] * b[idx];
    }
}
```

The diagonal (Jacobi) preconditioner scales each residual component by the inverse of the corresponding diagonal entry. This **element-wise operation** is embarrassingly parallel, ideal for GPU execution with a **simple CUDA kernel**.

(2) ILU

```
kkh_cuspsv_init( handleL, matL, dBufferL, spsvDescrL
               , CUSPARSE_FILL_MODE_LOWER, CUSPARSE_DIAG_TYPE_UNIT
               , L_rpnt, L_cinx, L_val, Ux, x , nnzL, n );
kkh_cuspsv_init( handleU, matU, dBufferU, spsvDescrU
               , CUSPARSE_FILL_MODE_UPPER, CUSPARSE_DIAG_TYPE_NON_UNIT
               , U_rpnt, U_cinx, U_val, Ux, x , nnzU, n );

kkh_cuspsv(Ux, matL,  b, handleL, dBufferL, spsvDescrL, n);
kkh_cuspsv( x, matU, Ux, handleU, dBufferU, spsvDescrU, n);
  ↳ cusparseSpSV_solve(handle, …, spsvDescr );
```

The ILU(0) preconditioner requires incomplete LU factorization followed by forward/backward substitutions. The factorization is performed on CPU due to sequential dependencies, while **triangular solves run on GPU via `cusparseSpSV`**.

(3) AMG

```
AMGX_solver_create(&solver, rsrc, AMGX_mode_dDDI, cfg);

AMGX_solver_setup(solver, Amatrix);

AMGX_solver_solve(solver, rhs, sol);
```

The algebraic multigrid (AMG) preconditioner constructs a hierarchy of coarser grids to accelerate convergence. **NVIDIA's AmgX library** provides GPU-optimized AMG coupled with GMRES solver.

Result

(1) Convergence History Comparison

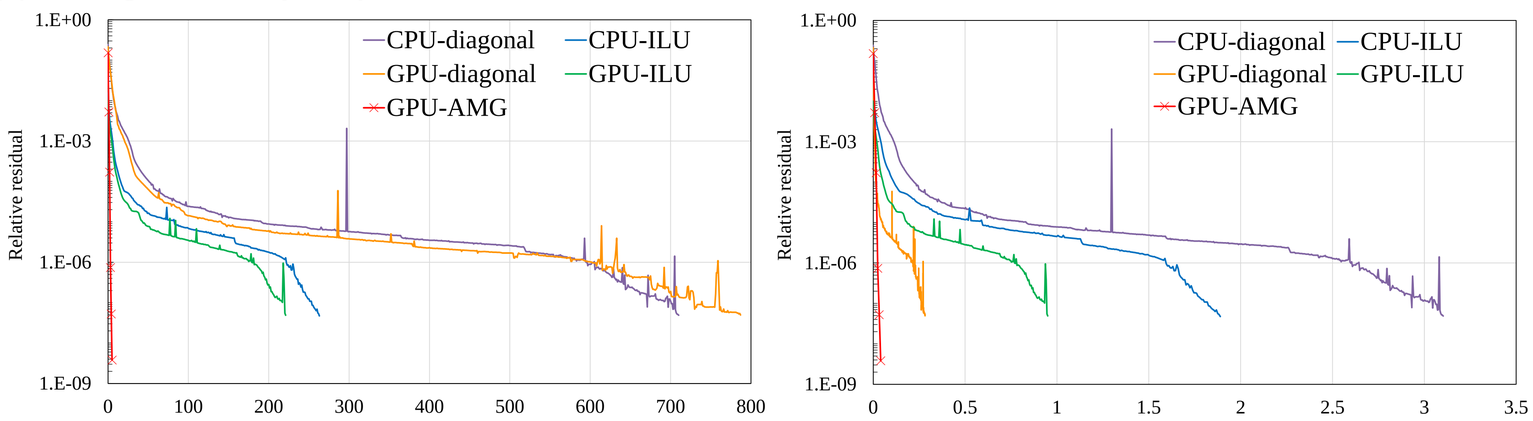


Fig1. Convergence history of preconditioned solvers: residual versus iteration count (left) and residual versus wall-clock time (right).

Figure 1 compares residual convergence histories by iteration count (left) and wall-clock time (right). In terms of iterations, ILU(0) converges in approximately 200 iterations while diagonal requires about 700, and AMG achieves convergence within only 5 iterations. However, when measured by wall time, the **GPU diagonal solver** reaches the convergence threshold **faster than GPU ILU**, despite requiring more iterations. This is because the diagonal preconditioner has much lower per-iteration cost due to its simple parallel structure. The AMG solver demonstrates the fastest convergence in both metrics, completing in under 0.1 seconds.

(2) Execution Time Breakdown and Speedup

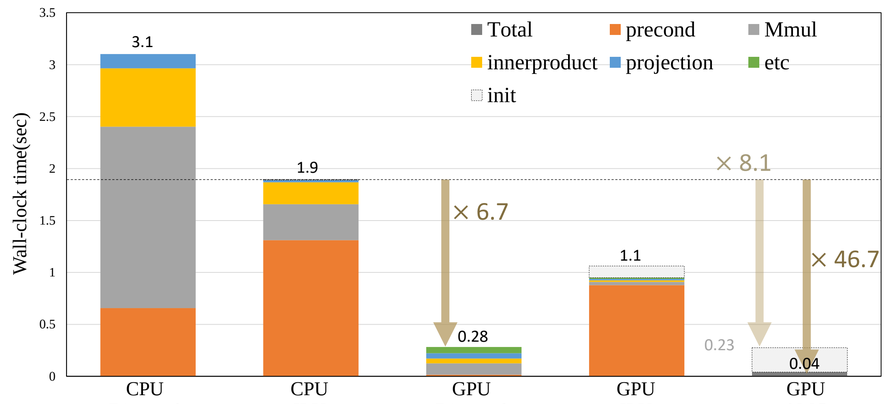


Fig2. Breakdown of execution time by solver component on CPU and GPU, with speedup ratios indicated.

Figure 2 shows the execution time breakdown. The GPU diagonal solver achieves 6.7× speedup with high parallel efficiency. The GPU ILU shows limited speedup due to sequential dependencies in sparse triangular solves (cusparseSpSV). The AMG solver is extremely fast (0.04 sec solve time), but since the pressure matrix changes every time step, the hierarchy must be rebuilt each time step, making total runtime comparable to diagonal.

(3) Residual Distribution at Convergence

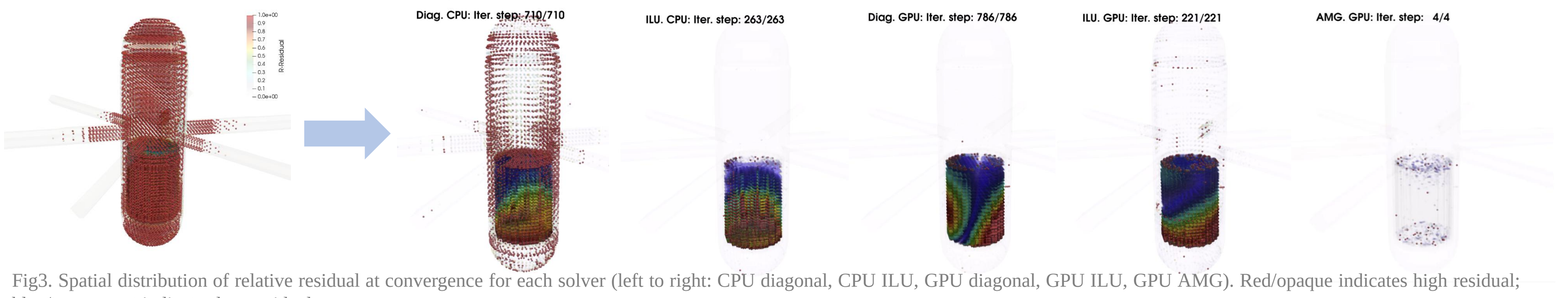


Fig3. Spatial distribution of relative residual at convergence for each solver (left to right: CPU diagonal, CPU ILU, GPU diagonal, GPU ILU, GPU AMG). Red/opaque indicates high residual; blue/transparent indicates low residual.

Conclusion

- The pressure solver in the thermal–hydraulic code was **successfully accelerated on a single GPU using cuSPARSE and cuBLAS libraries**, achieving significant speedup over the 64-core CPU implementation.
- The **diagonal preconditioner** demonstrated **the best balance** between convergence rate and computational efficiency, providing the highest speedup due to its simple and fully parallel structure.
- The **ILU(0) preconditioner** reduces iteration count but is **limited by sparse triangular solve efficiency**. The **AMG solver** converges **extremely fast but requires costly setup** at each time step when matrix coefficients change.

This research is supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MIST) (RS-2023-00282764). This work is also supported by Global top research lab funded by National research council of science and technology (GTL24031-000).

KiSTi www.kisti.re.kr **Korea Institute of Science and Technology Information**