NVIDIA GTC

Speeding up a banded-matrix solver by 3x using the updated cuSparse library

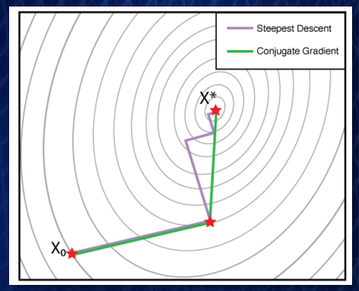
Ronald M. Caplan,
Miko Stulajter, and Jon A. Linker

Predictive Science Inc.

Outline

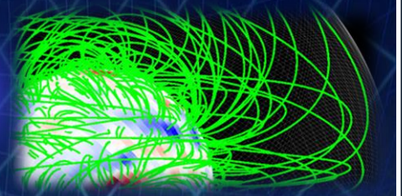
- Iterative solvers for sparse linear systems
- Preconditioned Conjugate Gradient
- NVIDIA **cuSparse** Library
- POT3D
- Test problem & Computational Environment
- CPU & GPU timings
- Mixed-precision
- Summary

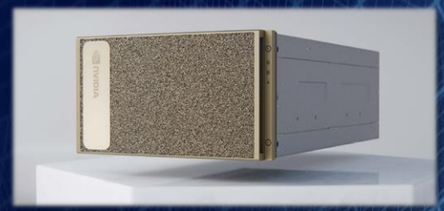
$$\mathbf{A}\,\vec{x} = \vec{b}$$



NVIDIA CUDA
NVIDIA.COM

POT3D



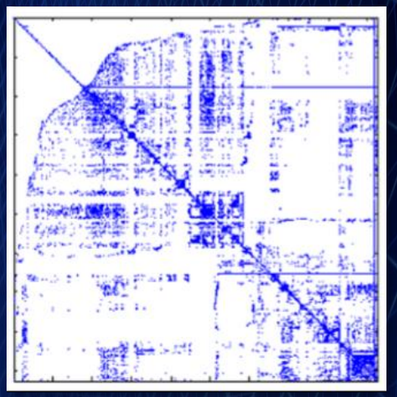


64-bit
32-bit

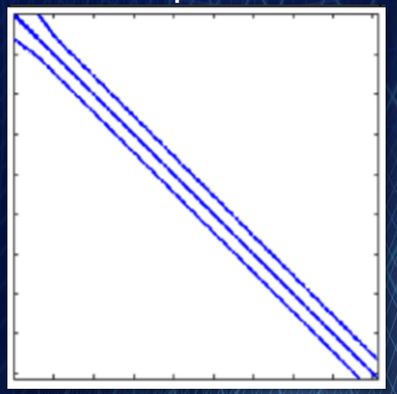
Sparse Matrix Linear System

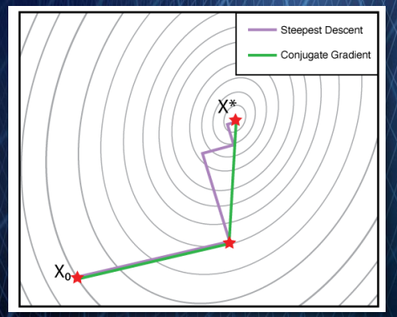
- Linear systems common in HPC $\mathbf{A}\,\vec{x} = \vec{b}$
- Matrix is “sparse” if most entries are zero; can be stored efficiently
- “Banded” matrices are sparse and consist of mostly off-set diagonal entries, typically having many fewer non-zero entries per row than general sparse matrices
- If the matrix is large, standard “dense” solver algorithms are often too slow. Instead, iterative techniques are often used
- The Preconditioned Conjugate Gradient (PCG) is a common method if the matrix is symmetric (or nearly so)

Sparse Matrix



Banded Sparse Matrix





Preconditioned Conjugate Gradient

- PCG consists of matrix-vector products, vector operations, dot products, and preconditioner (PC) application

$$\mathbf{A}\,\vec{x} \qquad \vec{x}\cdot\vec{y} \qquad a\,\vec{x}+b\,\vec{y}$$

- Applying the PC approximates applying the matrix inverse, but much less expensive to compute
- The PC reduces the number of iterations required for convergence
- Choosing a PC not simple; balance between cost and effectiveness

- For our solver, we use two *communication free* preconditioning options:

PC1

Point-Jacobi / Diagonal scaling

Cheap, not very effective

PC2

Non-overlapping domain decomposition zero-fill incomplete LU factorization

Expensive, much more effective!

Preconditioned Conjugate Gradient Cont.

PCG

$x_0 = u^n$ $\quad z_0 = \mathbf{P}^{-1} r_0$

$r_0 = b - \mathbf{A}\, x_0$ $\quad p_0 = z_0$

$\mathbf{P} \approx \mathbf{A}$ $\quad r_r = r_0 \cdot z_0$

Point-2-Point comm+sync

Global comm+sync

```
do k = 0 : k_max
    y_k = A p_k
    α_k = r_r / (p_k · y_k)
    x_{k+1} = x_k + α_k p_k
    r_{k+1} = r_k − α_k y_k
    z_{k+1} = P^{-1} r_{k+1}
    r_old = r_r
    r_r = r_{k+1} · z_{k+1}
    Check r_r for convergence
    β_k = r_r / r_old
    p_{k+1} = β_k p_k + z_k
enddo
```

- **PC1:** Simple vector operation, GPU implementation straight-forward
- **PC2:** Sequential in nature (setup and application); alternative algorithms needed for GPU implementation

PC1

LOAD

```
do j = 1 : N
    P_jj = A_jj
enddo
```

SOLVE

```
do i = 1 : N
    z_i = P_ii r_i
enddo
```

PC2

LOAD

```
LU = A
do i = 2 : N
    do k = 1 : i − 1        (LU_ik ≠ 0)
        LU_ik = LU_ik / LU_kk
        do j = k + 1 : N    (LU
            LU_ij = LU_ij − LU_ik
        enddo
    enddo
enddo
P = LU
```

SOLVE

```
do i = 1 : N
    z*_i = r_i
    do j = 1 : i            (LU_ij ≠ 0)
        z*_i = z*_i − LU_ij z*_j
    enddo
do
= N : 1
    z_i = z*_i
    do j = i + 1 : N        (LU_ij ≠ 0)
        z_i = z_i − LU_ij z_j
    enddo
    z_i = z_i / LU_ii
enddo
```

The NVIDIA cuSparse Library

- NVIDIA's CUDA toolkit contains the cuSparse library; includes GPU sparse triangular solvers

cuSparse

- Since their release in June 2010, these solvers have had two major revisions, resulting in three versions:

v1:	`cusparseDcsrsv_solve()`	CUDA [3.1, 10.2]
v2:	`cusparseDcsrsv2_solve()`	CUDA [4.0, 11.4]
v3:	`cusparseSpSV_solve()`	CUDA >=11.3

- At GTC 2017, we showed that, for banded-matrices with few non-zeros per row, the **v1** and **v2** solvers had non-optimal performance

- Here, we test the performance of **v3**, the first major update in 10 years

- We focus on the triangular solvers, but also test ILU0 factorization

POT3D

[Caplan, et al, Apj. 915 44 (2021)]

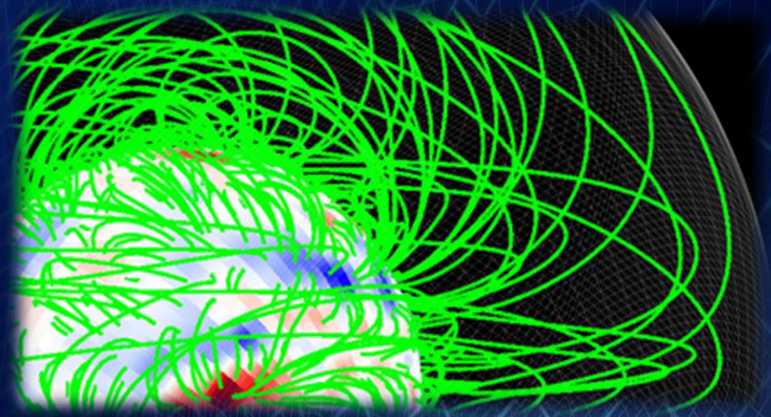
- Fortran MPI+OpenACC code (~7000 lines)
- Used to compute approximate magnetic fields of the solar corona using surface field observations as input
- Open-source

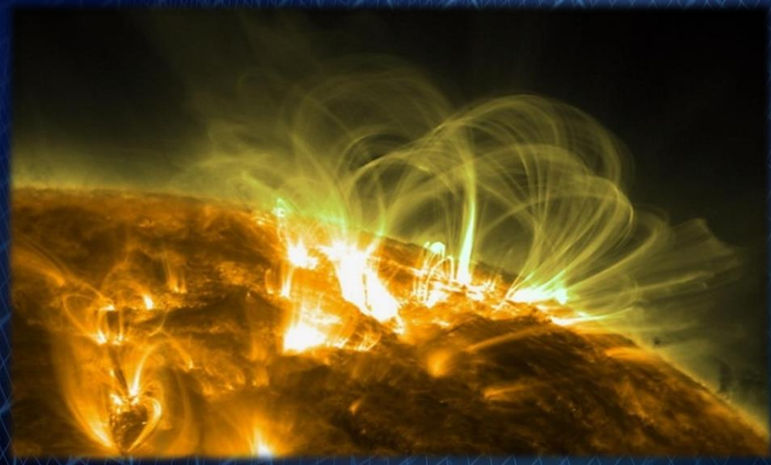
GitHub github.com/predsci/POT3D

- Included in the SPEChpc™ 2021 Benchmark

spec spec.org/hpc2021

F MPI OpenACC



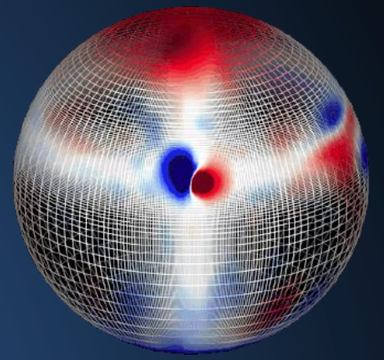


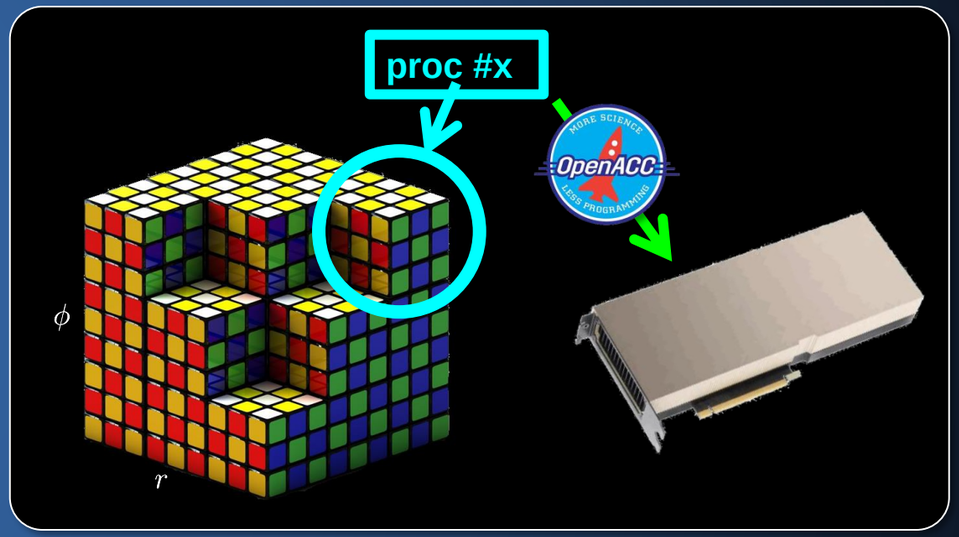
POT3D cont.

- Solves the 3D Laplace equation on a non-uniform logically-rectangular spherical grid
- Parallelized with MPI+OpenACC in a 3D logical domain decomposition
- Uses PCG, with the operator discretized with finite difference and stored as a sparse 7-banded matrix
- Matrix stored in modified DIA format for efficient matrix-vector product
- Another copy of the matrix is stored in a compressed sparse row (CSR) format for easy use with the ILU0 **PC2**

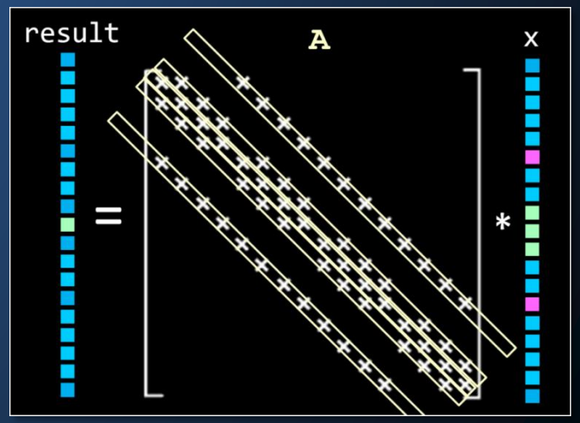
$$\nabla^2 \Phi = 0$$

$$\mathbf{A}\,\vec{x} = \vec{b}$$





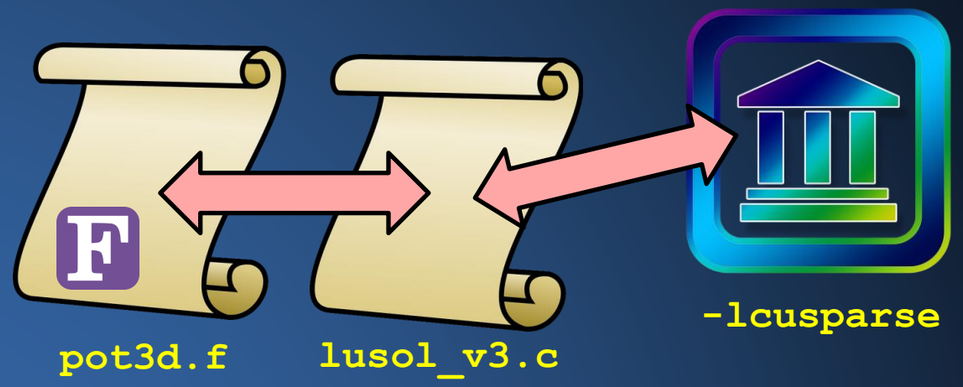

```
do j=2,ny-1
  do i=2,nx-1
    result(i,j) =   A(1,i,j)*x(i  ,j-1)
                  + A(2,i,j)*x(i-1,j  )
                  + A(3,i,j)*x(i  ,j  )
                  + A(4,i,j)*x(i+1,j  )
                  + A(5,i,j)*x(i  ,j+1)
  enddo
enddo
```



Implementing cuSparse into POT3D

- cuSparse contains native Fortran bindings
- For portability, we instead call C code from Fortran (minimal **`#ifdef`** pre-processing)

```
nvcc -c [FLAGS] lusol_v3.c
nvfortran [FLAGS] lusol_v3.o [LIBS] pot3d.f
```



lusol_v3.c

```
void load_v3(double* CSR_LU,int* CSR_I,int* CSR_J,int N,int M){
...
cusparseCreate(&cusparseHandle);
...
cusparseDcsrilu02(cusparseHandle,N,M,M_desc,CSR_LU,CSR_I,
                  CSR_J,M_alyz,M_pol,Mbuf);
...}
```

LOAD

```
void lusol_v3(double* x){...
// Forward solve (Ly=x)
  cusparseSpSV_solve(cusparseHandle, L_trans,
             &alpha_DP, L_mat, DenseVecX, DenseVecY, CUDA_R_64F,
             CUSPARSE_SPSV_ALG_DEFAULT, L_described);
  cudaDeviceSynchronize();
// Backward solve (Ux=y)
  cusparseSpSV_solve(cusparseHandle, U_trans,
             &alpha_DP, U_mat, DenseVecY, DenseVecX, CUDA_R_64F,
             CUSPARSE_SPSV_ALG_DEFAULT, U_described);
  cudaDeviceSynchronize();
...}
```

SOLVE

pot3d.f

```
module cusparse_interface
  interface
    subroutine lusol_v3(x) BIND(C, name="lusol_v3")
      use, intrinsic :: iso_c_binding
      type(C_PTR), value :: x
    end subroutine lusol_v3
  end interface
end module
```

```
use, intrinsic :: iso_c_binding
use cusparse_interface
integer(c_int) :: cN
```

```
!$acc host_data use_device(x)
   call lusol_v3(C_LOC(x(1)))
!$acc end host_data
```

Test Problem

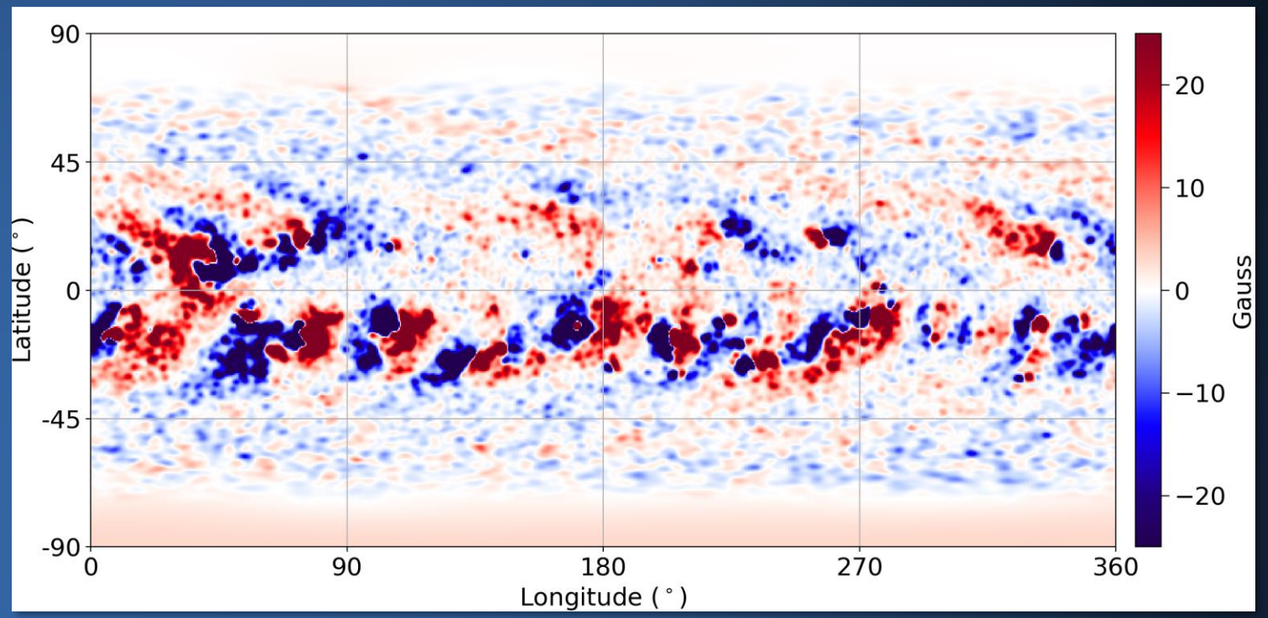
- “Tiny” POT3D benchmark run from the SPEChpc™ 2021 Benchmark Suite

- Single node reference CPU PC1 timing on (2x) 12-core Intel Xeon E5-2680v3: 2130 seconds

- Only uses ~16 GB memory, so strong scaling may be sub-optimal (test chosen to fit on 1 V100 GPU)

- All timings done with internal `MPI_Wtime()` calls, and averaged over three runs

Input surface radial magnetic field map



	NR	NT	NP	Total Points
Resolution	173	361	1171	~73 million

	PC1	PC2
Solver Iterations	13799	1348 to 1846 (# MPI rank dependent)

Computational Environment

EXPANSE
COMPUTING WITHOUT BOUNDARIES

SDSC | XSEDE Extreme Science and Engineering Discovery Environment

```
mpiexec -npersocket 32 singularity
exec --home $PWD $CONT ./pot3d
```

EXPANSE@SDSC CPU NODE

# CPUs x Model	**(2x)** EPYC 7742
# Total Cores	128 *(we use 64)*
Peak FLOP/s	7.0 TFLOP/s
Memory	256 GB
Total Memory Bandwidth	381.4 GB/s
Compiler Flags	`-O3 -tp=zen2`
OpenMPI	`v4.04`

CPU
OpenMPI 4.1.2rc2
nvfortran
NVIDIA HPC SDK 22.1

EXPANSE@SDSC GPU NODE

# CPUs x Model	**2x** Xeon Gold 6248
# GPUs x Model	**4x** V100 SXM2
Peak DP FLOP/s / GPU	7.8 TFLOP/s
Memory / GPU	32 GB
Memory Bandwidth/GPU	897 GB/s
Compiler Flags	`-O3 -tp=skylake -acc=gpu -gpu=cc70,cudaXX.Y`

GPU
OpenMPI 3.1.5
nvfortran
NVIDIA HPC SDK 22.1

```
singularity exec --nv --home $PWD
$CONT mpiexec -np <N> ./pot3d
```

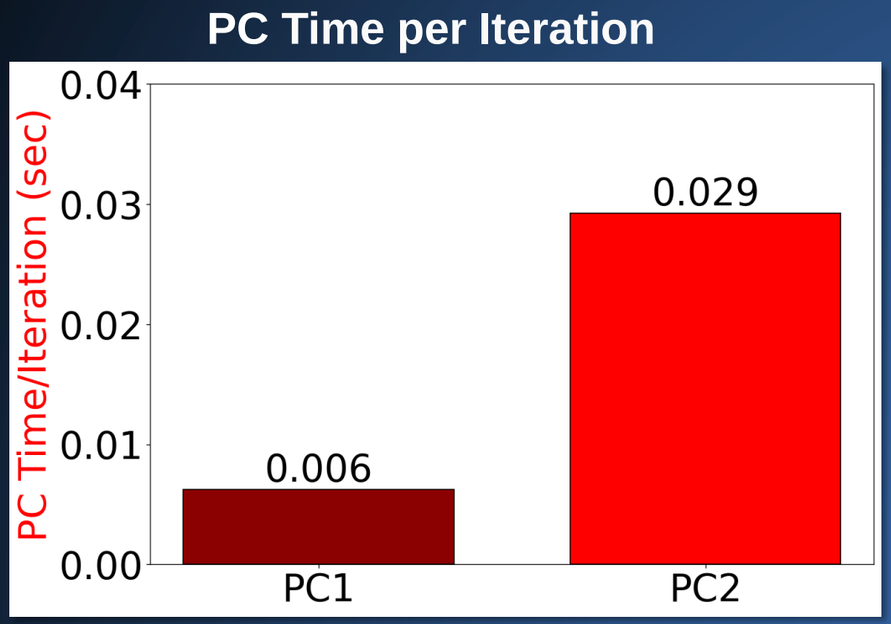
SAN DIEGO STATE UNIVERSITY 1897

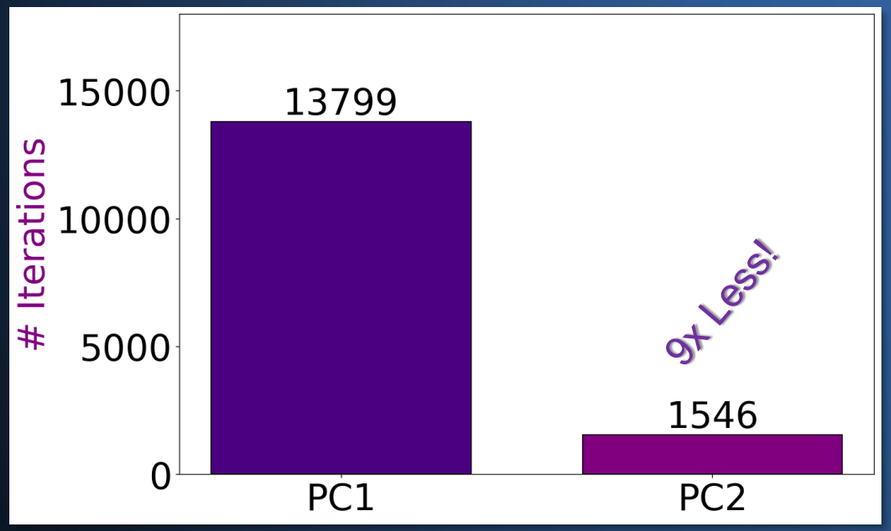
CSRC — gateway to the future — an interdisciplinary

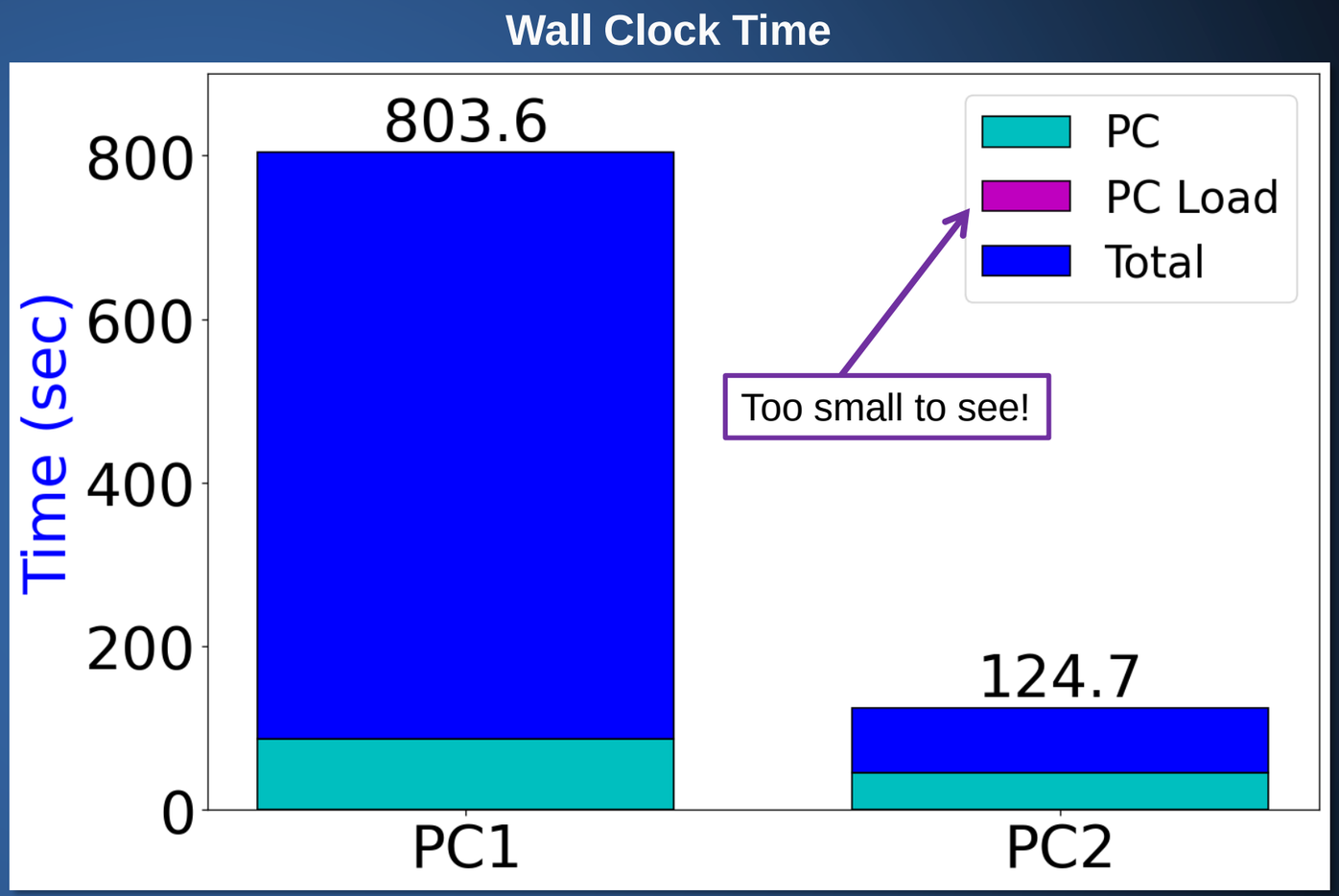
CSRC@SDSU DGX A100

# CPUs x Model	**(2x)** EPYC 7742
# GPUs x Model	**8x** A100-40GB SXM4
Peak DP FLOP/s / GPU	9.8 TFLOP/s
Memory / GPU	40 GB
Memory Bandwidth/GPU	1555 GB/s
Compiler Flags	`-O3 -tp=zen2 -acc=gpu -gpu=cc80,cudaXX.Y`

CPU Performance: Single (2x) EPYC Node







Speedup PC2 vs. PC1: 6.4x

GPU Code Versions

- We test all 3 versions of the cuSparse triangular solvers
- **v1** cannot be run on the A100 (CUDA version too old), so run on V100 only
- An API change occurred for **v2** with CUDA 11, so we designate the updated API version as **v2b**
- **v1** and **v2** compute the **PC2** ILU0 on the CPU, while **v2b** and **v3** compute it on the GPU using cuSparse

v1: `cusparseDcsrsv_solve()` using CUDA 10.0

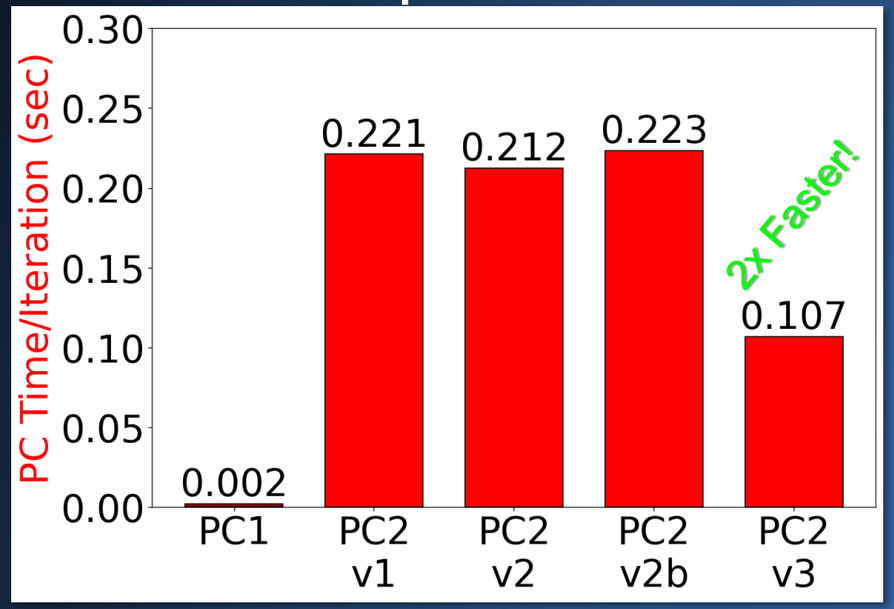
v2: `cusparseDcsrsv2_solve()` using CUDA 10.0

v2b: `cusparseDcsrsv2_solve()` using CUDA 11.5 `cusparseDcsrilu02()` to initialize **PC2** on GPU (includes new API changes)

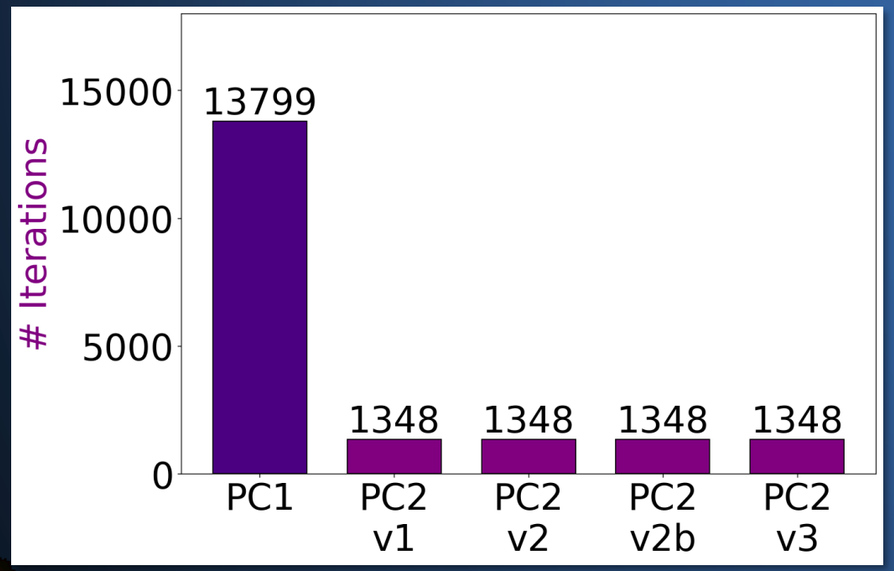
v3: `cusparseSpSV_solve()` using CUDA 11.5 `cusparseDcsrilu02()` to initialize **PC2** on GPU

GPU Results for single V100

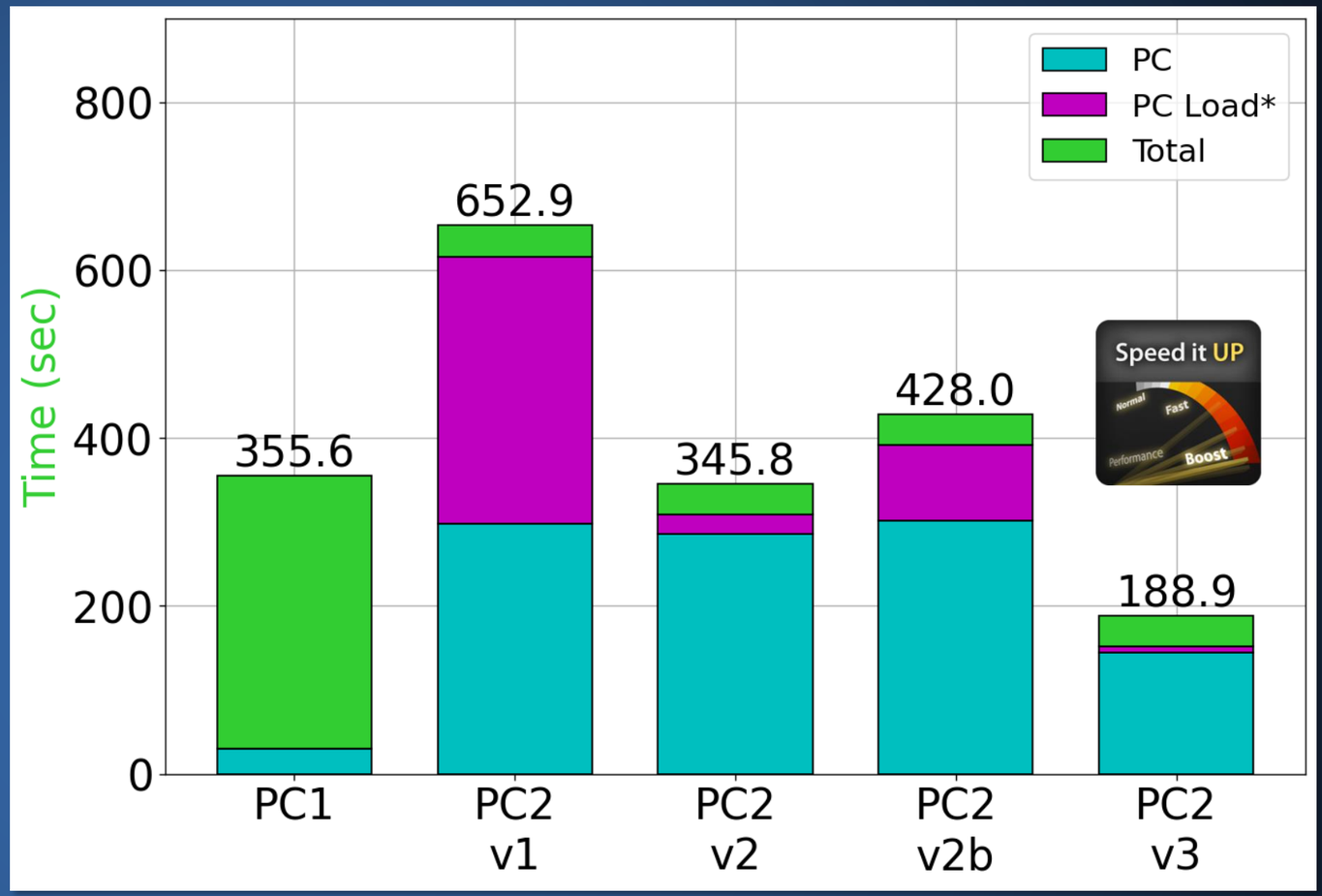
PC Time per Iteration



PC # Iterations



Wall Clock Time

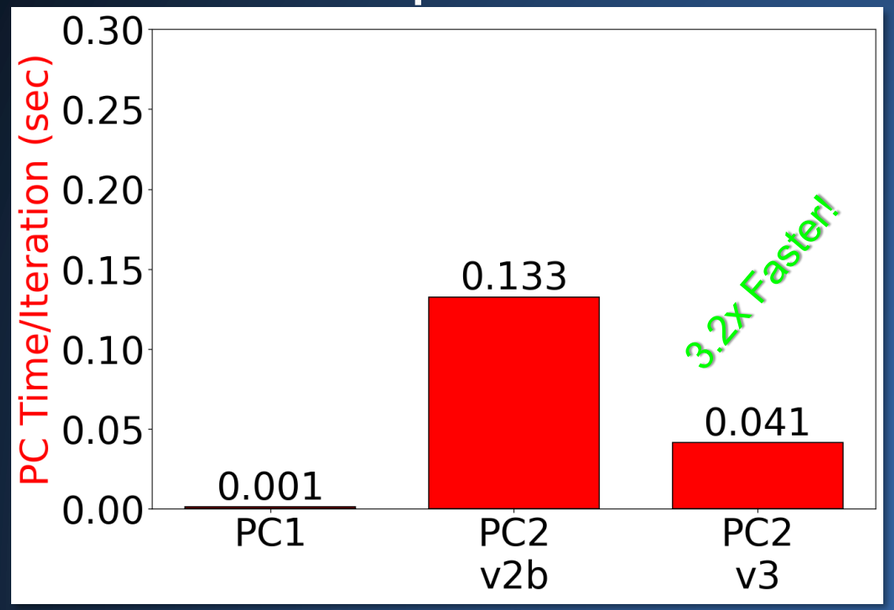


Speedup PC2 (v3) vs. PC2 (v2b) : 2.3x

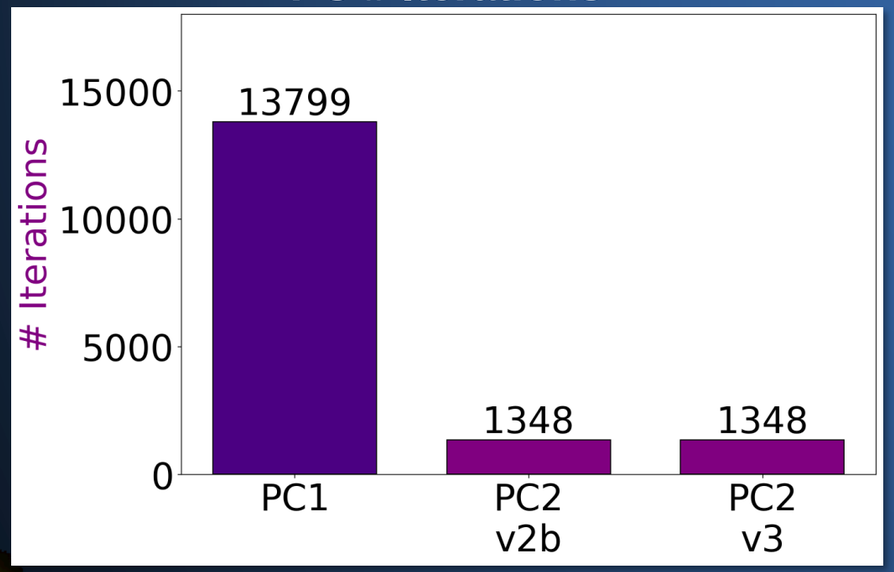
Speedup PC2 (v3) vs. PC1: 1.9x

GPU Results for single A100(40GB)

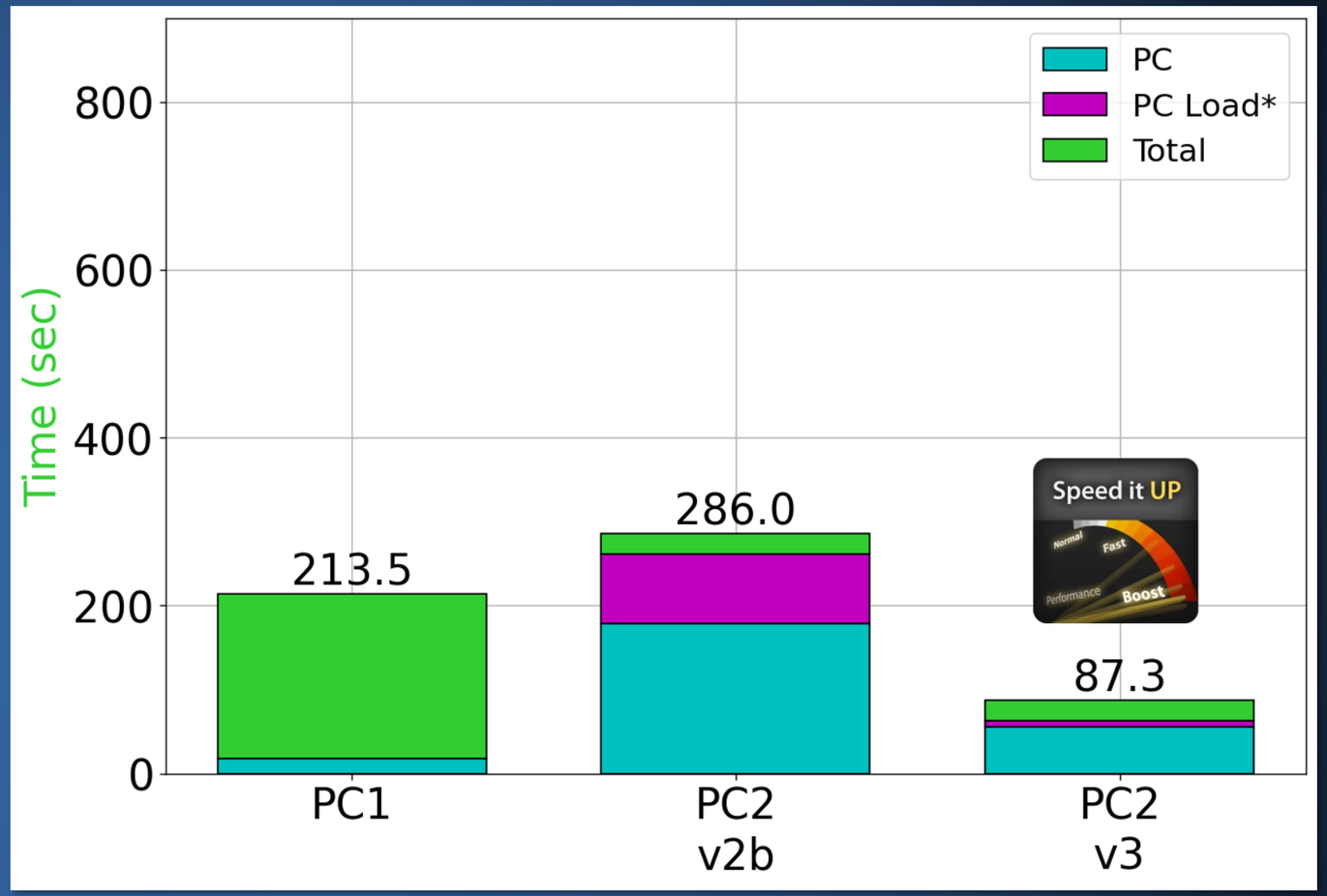
PC Time per Iteration



PC # Iterations



Wall Clock Time



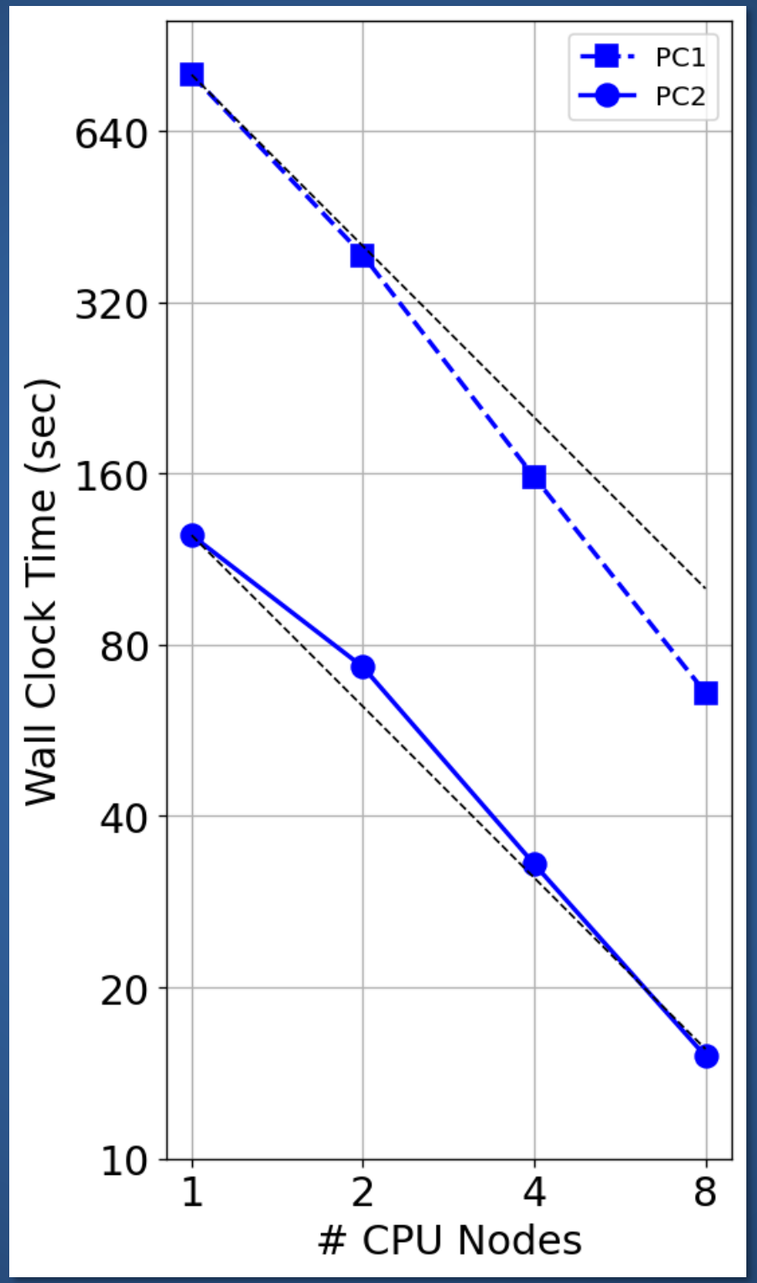
Speedup PC2 (v3) vs. PC2 (v2b) : 3.3x

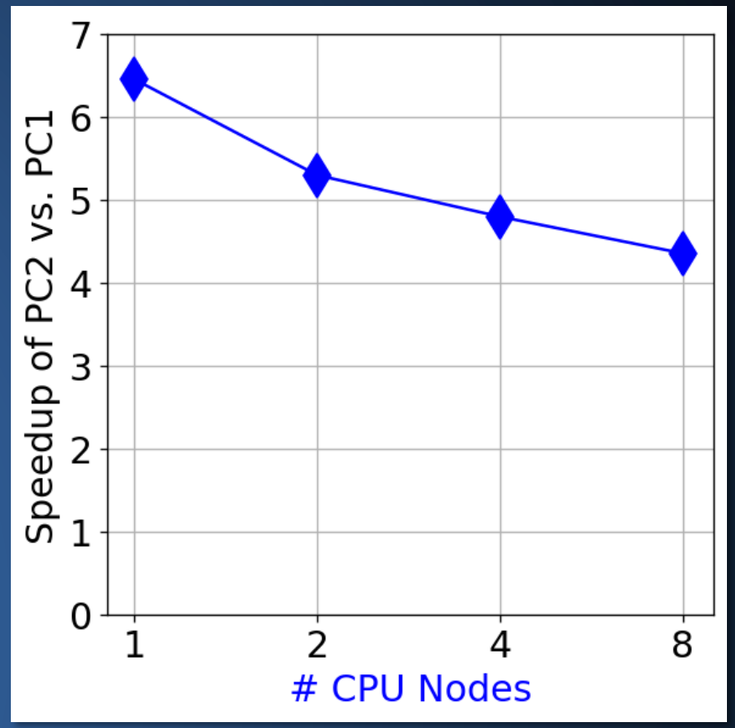
Speedup PC2 (v3) vs. PC1: 2.5x

A100 PC2/PC1 speedups are higher than on the V100

CPU Results: Multi-node Scaling

- **PC1** exhibits “super-scaling” (probably due to cache)
- **PC2** scales well, but due to scaling of **PC1**, its speedup decreases; stays above 4x

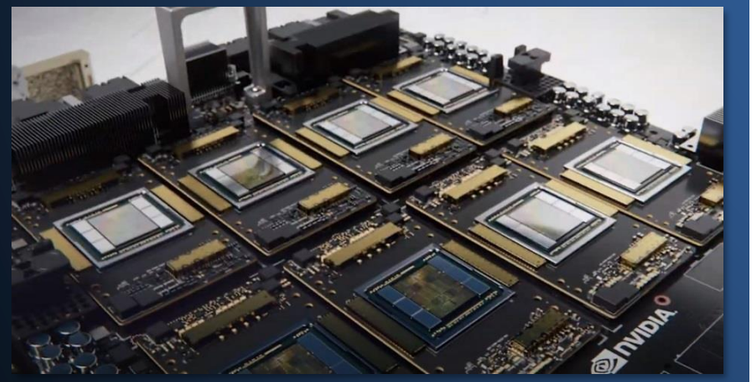


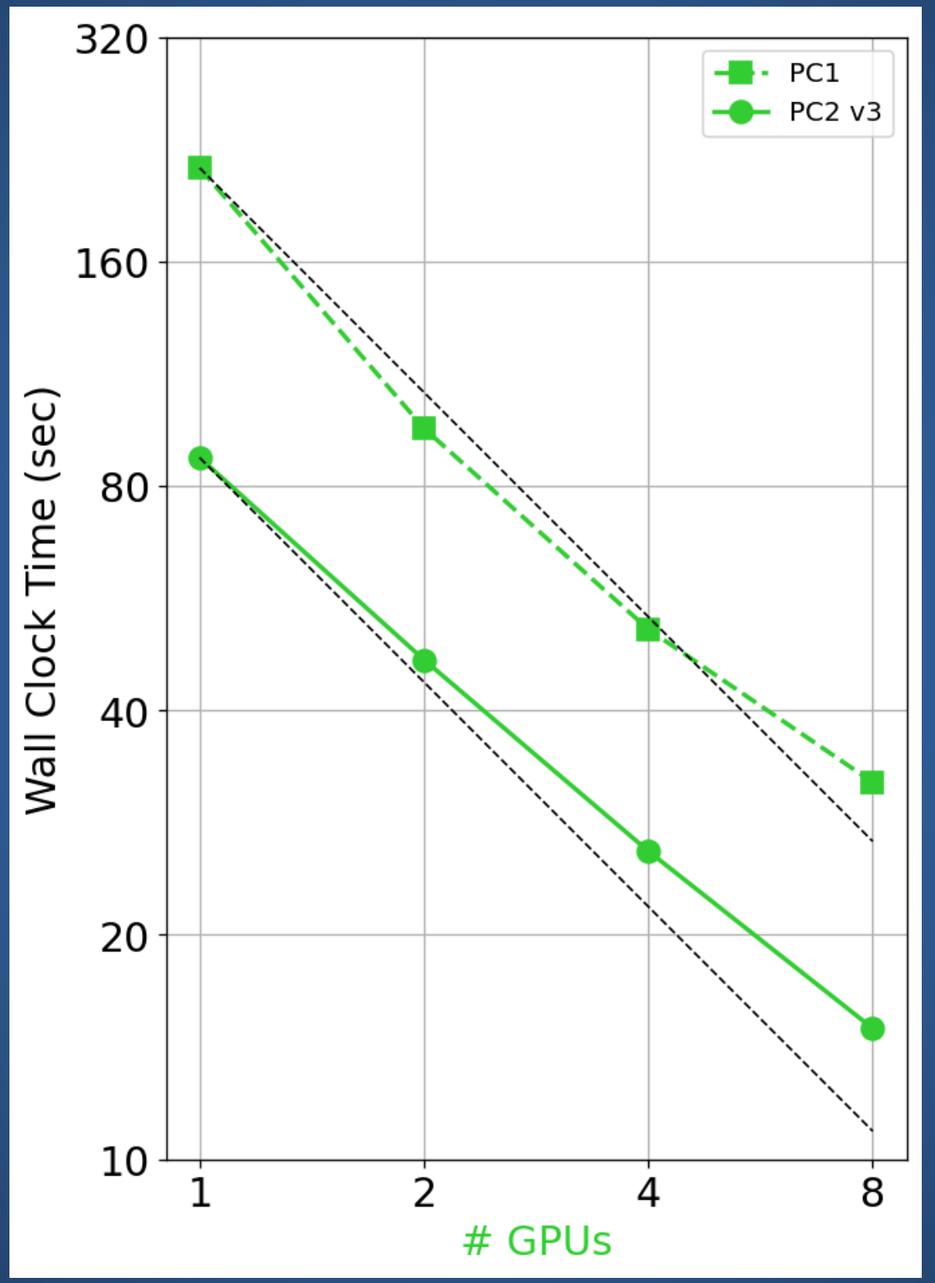


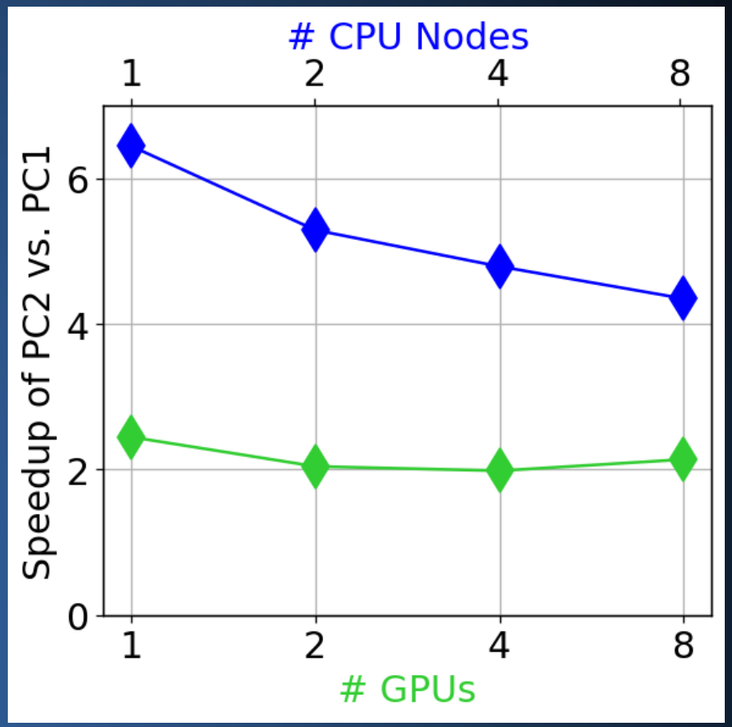
PC2 ITERATIONS	
# CPU Nodes	Iterations
1	1546
2	1841
4	1843
8	1846

GPU Results Multi-GPU

- A100(40GB) DGX with **v3**
- **PC1** scales well
- Scaling of **PC2** tapers off for larger # GPUs (expected due to small problem size)
- Consistent speedup of **PC2** over **PC1** of ~2x







PC2 ITERATIONS	
# GPUs	Iterations
1	1348
2	1395
4	1388
8	1389

Mixed Precision

- Single precision
  - Half the memory footprint
  - Can use faster GPU compute cores

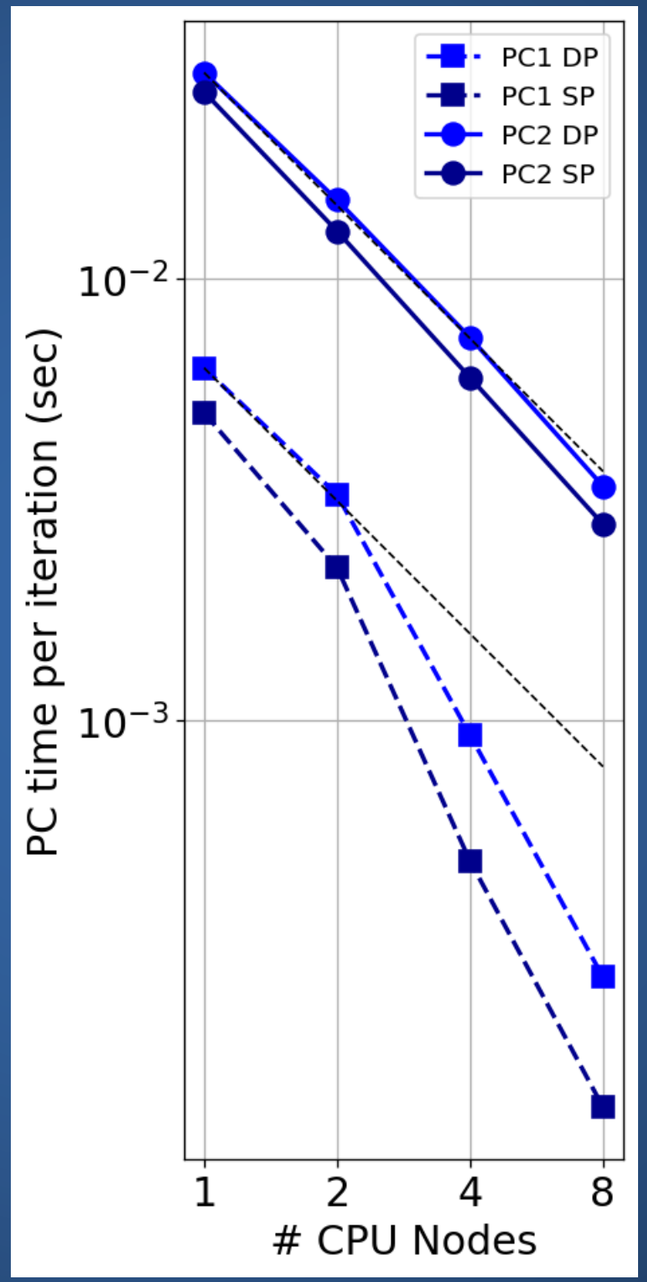
- Can not be used for the overall solve
  - May not converge
  - Solution required to be double precision

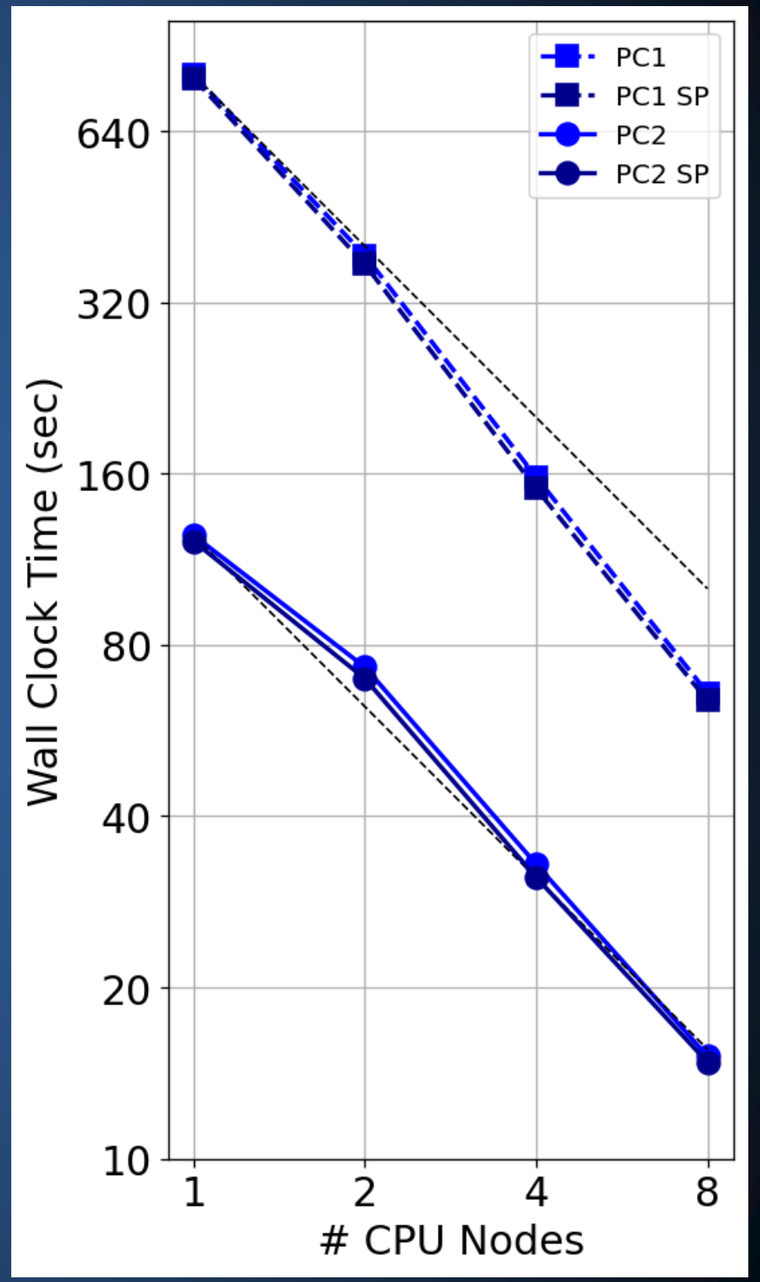
- Use only for the preconditioner!
  - PC an approximation, so could speed up the solve while yielding equivalent results
  - Requires casting arrays in and out
  - Number of iterations may go up

Mixed Precision PC CPU Results

- PC2 performance is improved by ~10%
- Speedup of overall solve is ~3%
- Number of iterations remained the same!

PC2 ITERATIONS		
# CPU Nodes	DP	SP
1	1546	1546
2	1841	1841
4	1843	1843
8	1846	1846

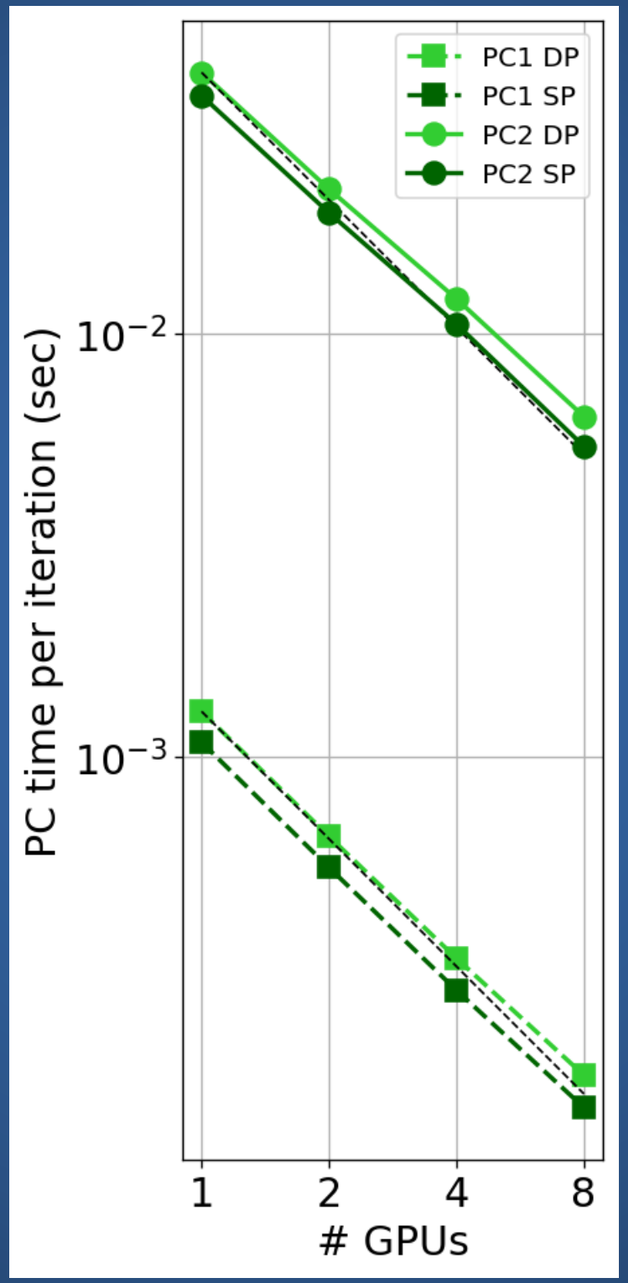


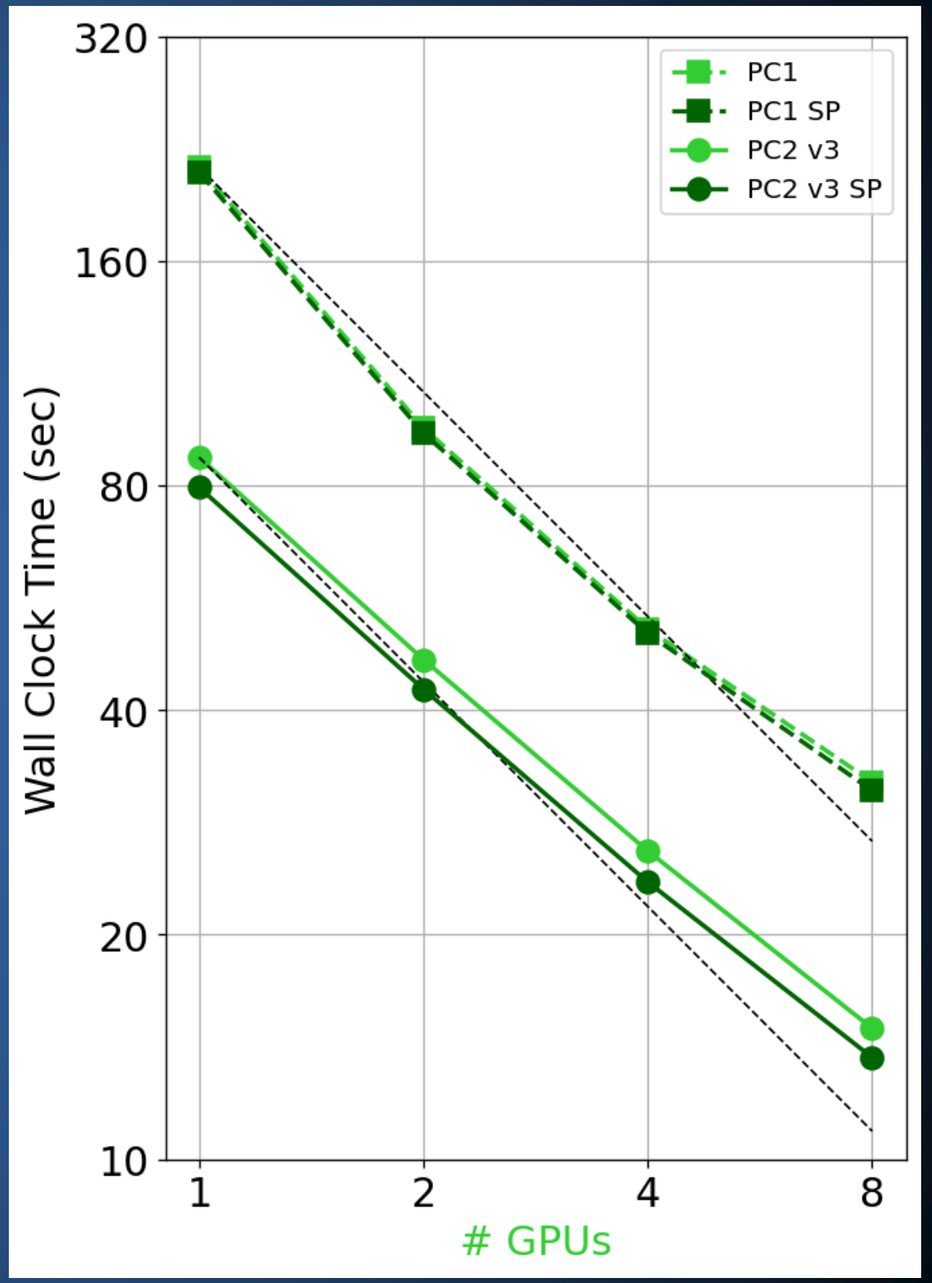


Mixed Precision PC GPU Results

- PC2 performance is improved by ~12%
- Speedup of overall solve is ~9%
- Number of iterations remained the same!

PC2 ITERATIONS		
# GPUs	DP	SP
1	1348	1348
2	1395	1395
4	1388	1388
8	1389	1389





Summary

- New version of the ILU0 factorization and triangular solver in cuSparse (CUDA>=11.3) has a significant speedup over previous versions for banded matrices with small numbers of non-zeros per row
- For our POT3D code, the new version is **3x** faster than the previous one, making the overall code **2x** faster than current production runs
- We recommend users with similar matrices who have avoided the library in past to give it another look
- The use of mixed-precision yields additional performance improvements (~10%)
- POT3D on a DGX 8x A100-40GB GPU server now outperforms **8x** EPYC 7742 dual-socket CPU nodes