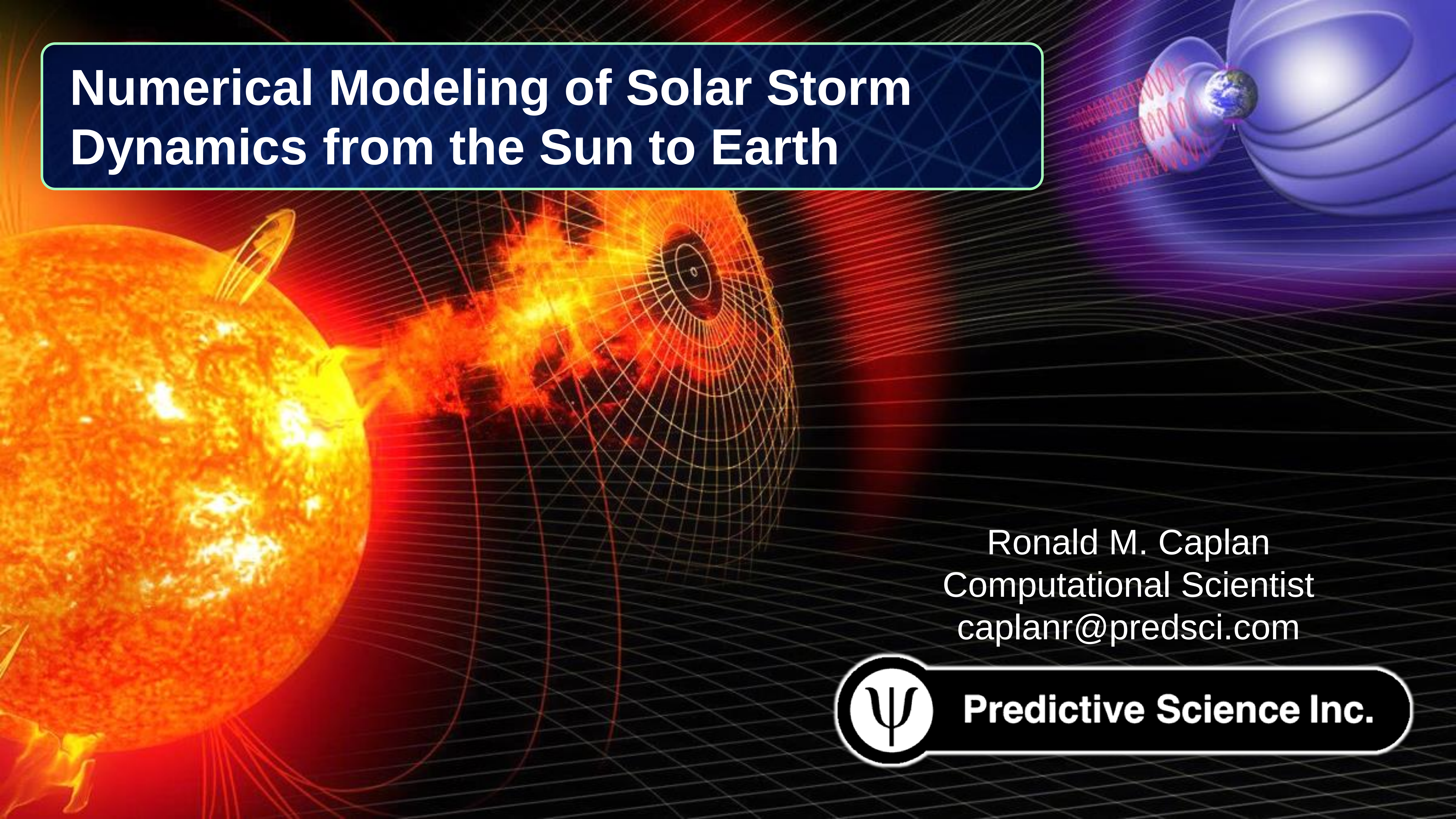


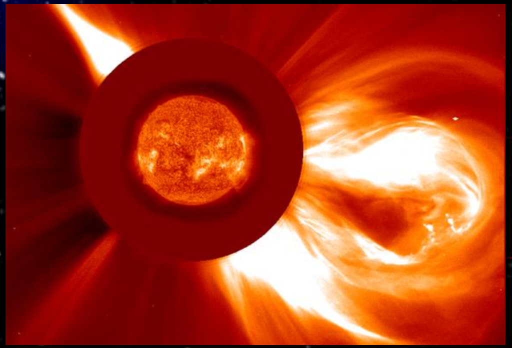
Numerical Modeling of Solar Storm Dynamics from the Sun to Earth

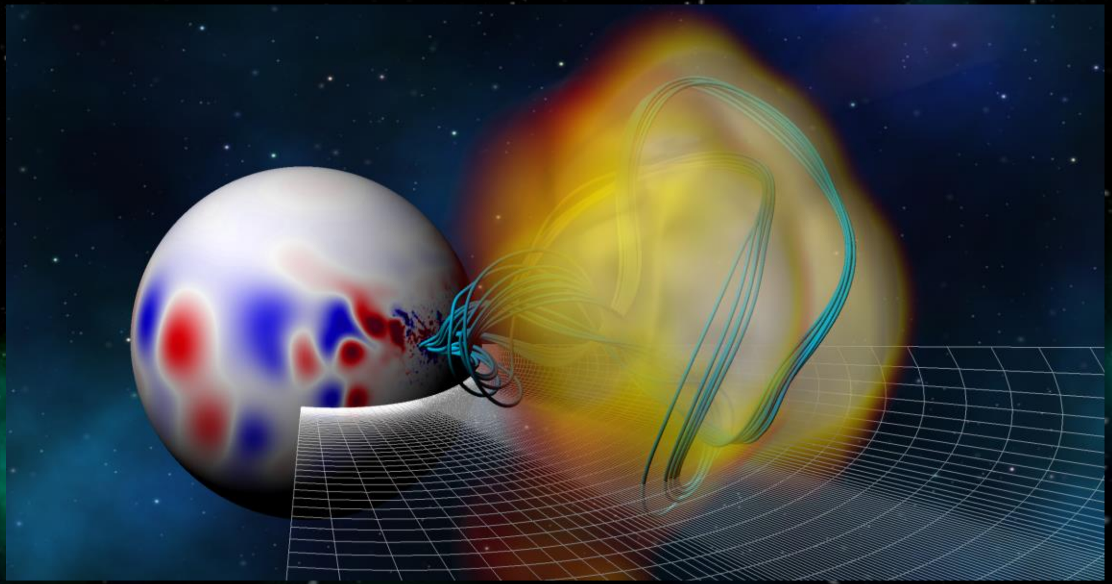
Ronald M. Caplan
Computational Scientist
caplanr@predsci.com

Predictive Science Inc.

Outline

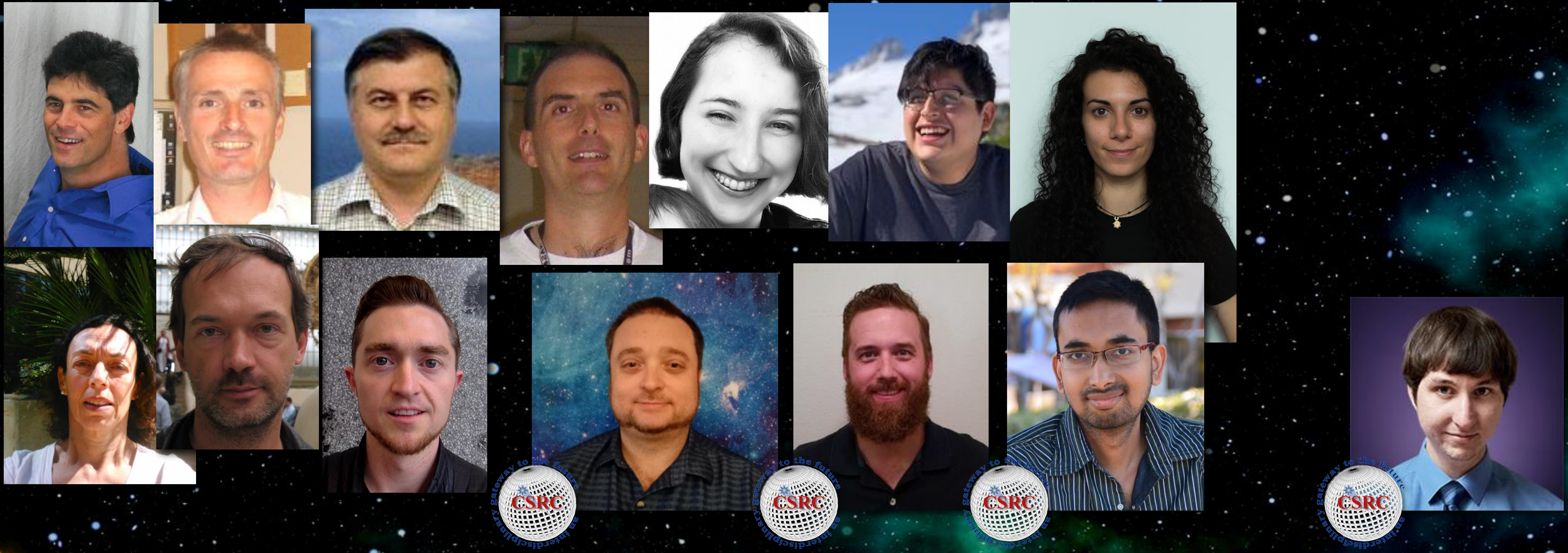
- Predictive Science Inc.
- Solar storms
- MAS Magnetohydrodynamic Model of the Solar Corona and Heliosphere
- Highlighted method 1: STS
- Highlighted method 2: GPU
- Make your own solar storm!





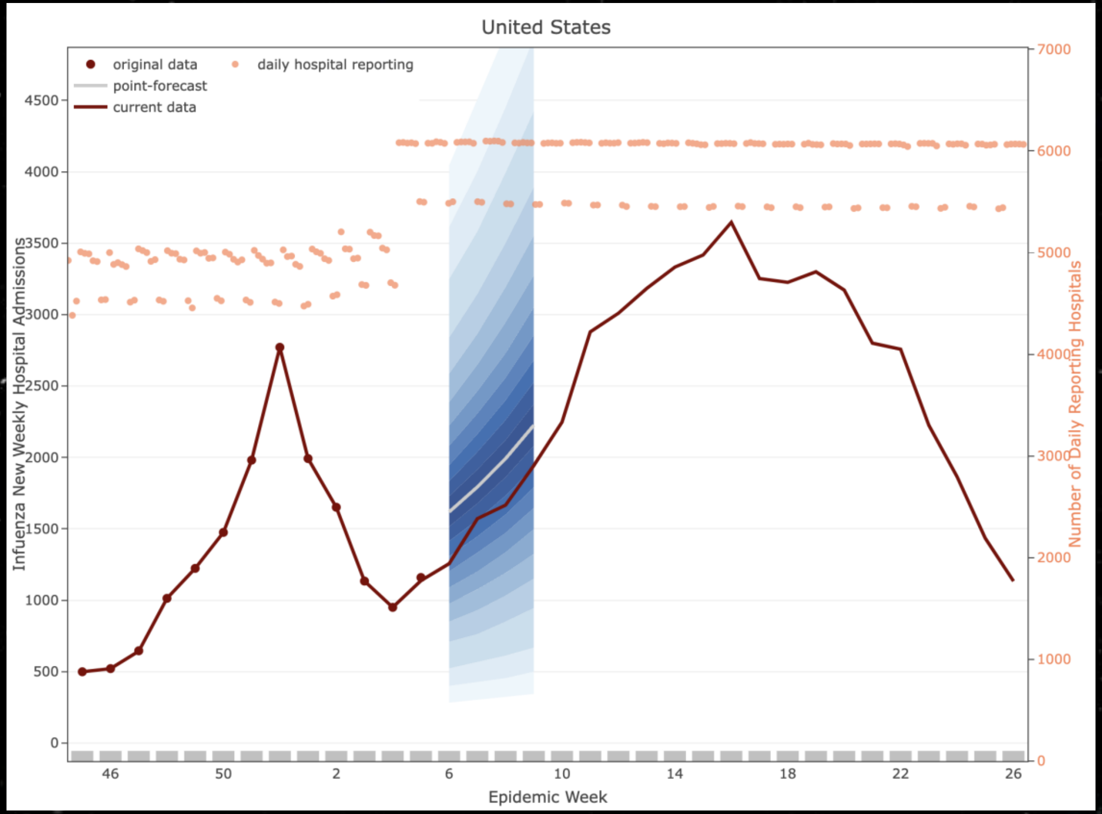
Predictive Science Inc. - Who are we?

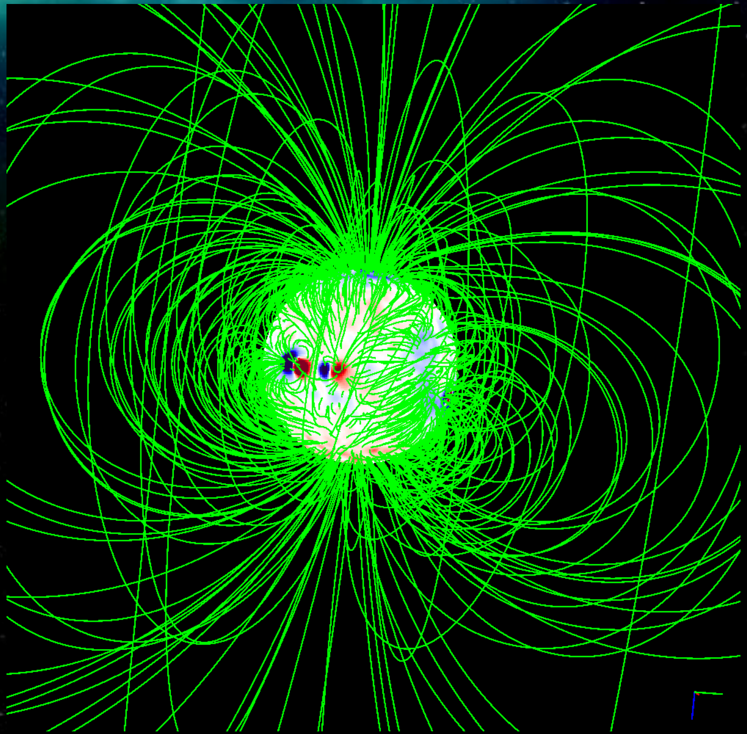
- San Diego based, employee owned, founded in 2008
- 13 full-time scientists & engineers including 3 CSRC alumni
- Multiple internships given and ***available***

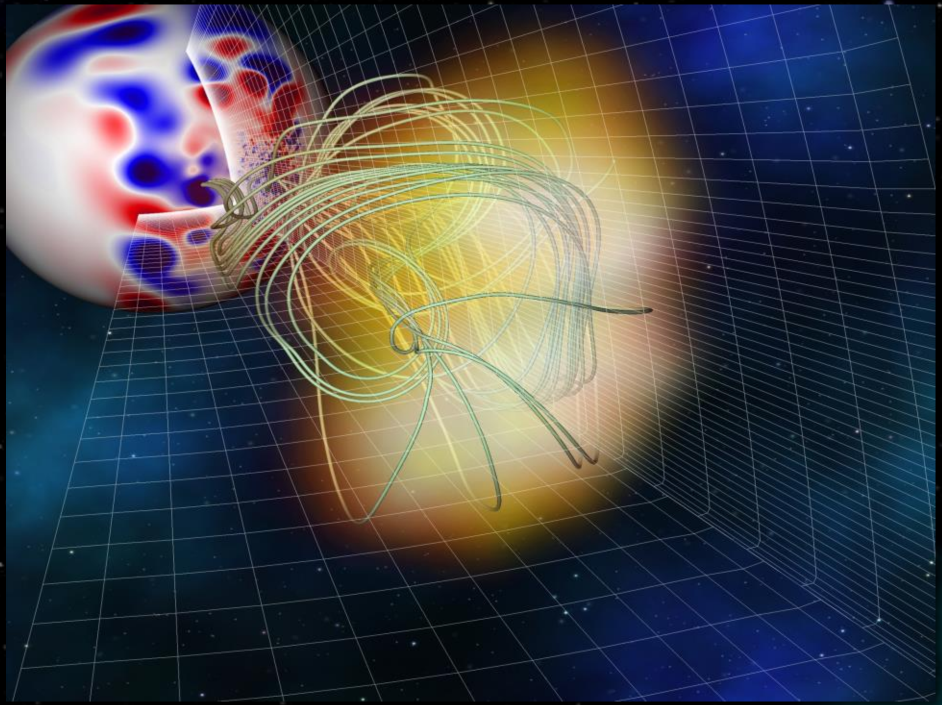


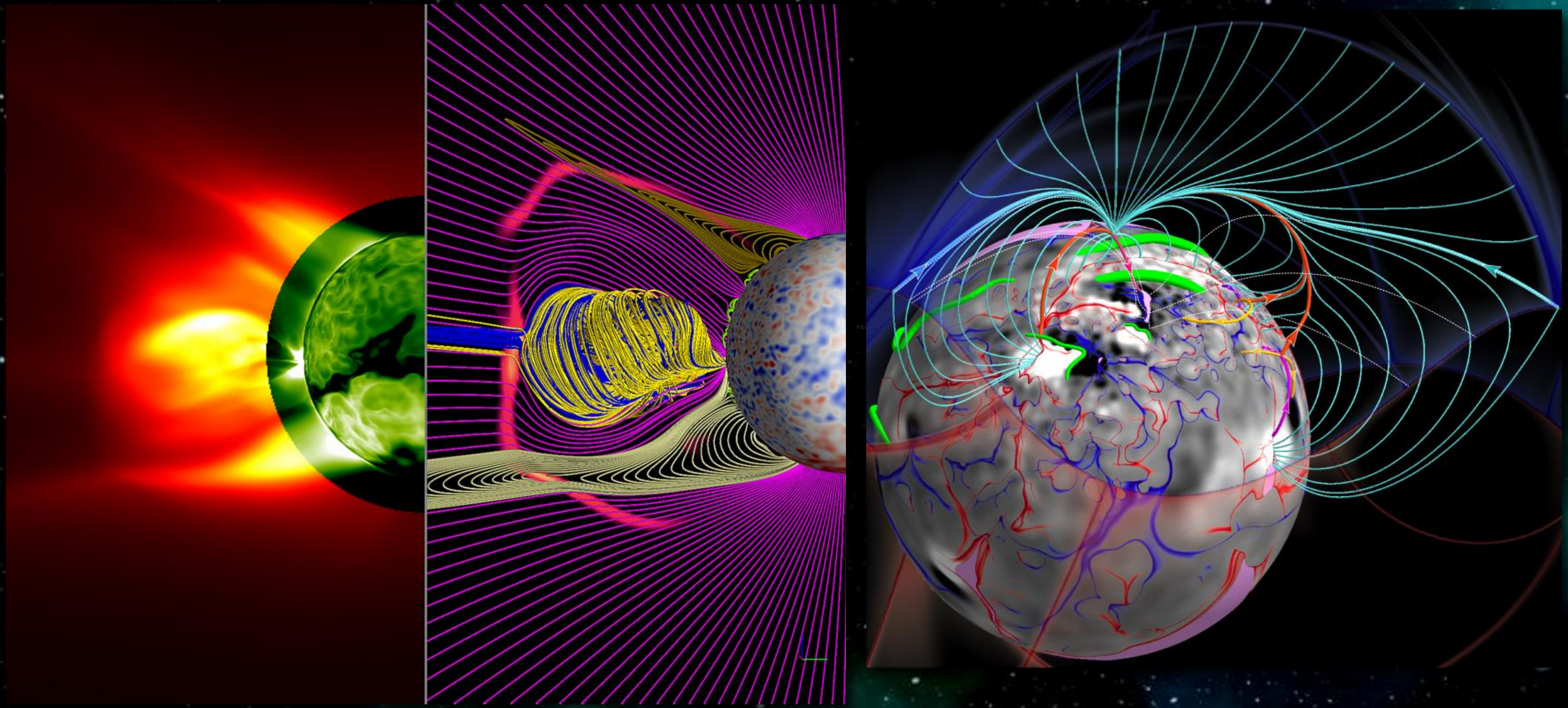
Predictive Science Inc. - What do we do?

- Solar physics (majority of work)
- Epidemiological data science



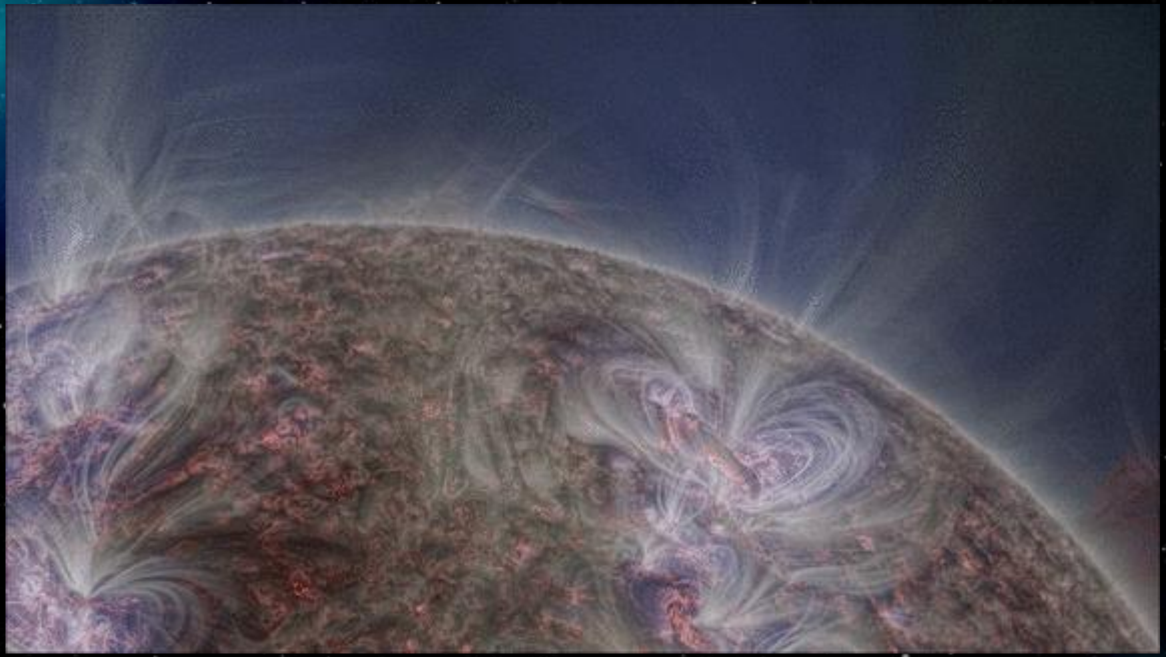


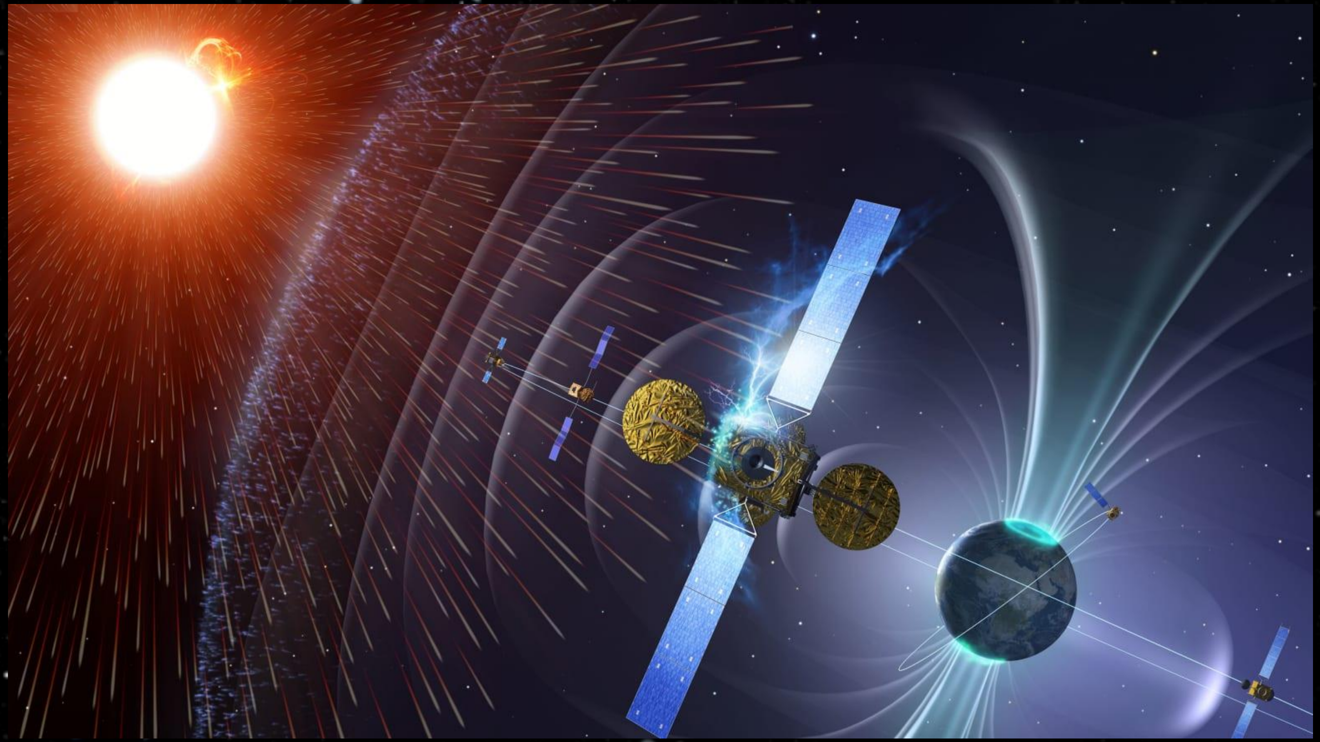




Solar Storms

- Large explosive events on the Sun including solar flares and coronal mass ejections (CME)
- CMEs can eject billions of tons of magnetized million-degree plasma out into space
- CME impacts on Earth can cause interference and damage to electronic infrastructure including GPS satellites and the power grid





MAS MAGNETOHYDRODYNAMIC ALGORITHM OUTSIDE A SPHERE

predsci.com/mas

Purpose: General-purpose simulations of the corona and heliosphere for use with solar physics research.

Model: Spherical 3D resistive thermodynamic MHD equations.

Algorithm: Implicit and explicit time-stepping with finite-difference stencils. Implicit steps use sparse matrix preconditioned iterative solver.

Code: ~70,000 lines of Fortran

Parallelism: MPI + OpenACC

Predicted Corona of the August 21st, 2017 Total Solar Eclipse
www.predsci.com/eclipse2017

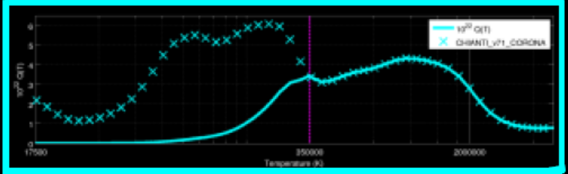
The MAS MHD Model

$$\frac{\partial \mathbf{A}}{\partial t} = \mathbf{v} \times (\nabla \times \mathbf{A}) - \boxed{\frac{c^2 \eta}{4\pi} \nabla \times \nabla \times \mathbf{A}}$$

RESISTIVITY

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \mathbf{v})$$

$$\frac{\partial T}{\partial t} = -\nabla \cdot (T\mathbf{v}) - (\gamma - 2)(T \nabla \cdot \mathbf{v}) + \frac{\gamma - 1}{2k} \frac{m_p}{\rho} \left[\boxed{-\nabla \cdot (\mathbf{q}_1 + \mathbf{q}_2)} - \boxed{\frac{\rho^2}{m_p^2} Q(T)} + \boxed{H} \right]$$



RADIATIVE COOLING

CORONAL HEATING

$$H = H^* + \frac{\rho}{4\lambda_\perp} \left[|z_-| z_+^2 + |z_+| z_-^2 \right]$$

$$\lambda_\perp = \boxed{\lambda_0} \sqrt{\frac{B_w}{|\mathbf{B}|}} \quad |z_\pm(r = R_\odot)| = \boxed{z_0}$$

THERMAL CONDUCTION

$$\mathbf{q}_1 = -f(r)\, \beta_{T\mathrm{cut}}(T)\, \kappa_0 T^{5/2}\, \hat{\mathbf{b}}\hat{\mathbf{b}} \cdot \nabla T$$

$$\mathbf{q}_2 = (1 - f(r)) \frac{k}{(\gamma - 1)} \frac{\rho}{m_p} T\, \mathbf{v}\, \hat{\mathbf{b}}\hat{\mathbf{b}}$$

ALFVEN WAVES

$$\boxed{\frac{\partial \epsilon_\pm}{\partial t} = -\nabla \cdot (\epsilon_\pm [\mathbf{v} \pm \mathbf{v}_A]) - \frac{\epsilon_\pm}{2} \nabla \cdot \mathbf{v}}$$

$$\frac{\partial \mathbf{v}}{\partial t} = -\mathbf{v} \cdot \nabla \mathbf{v} + \frac{1}{\rho} \left[\frac{1}{c} \mathbf{J} \times \mathbf{B} - \nabla p - \boxed{\nabla \left(\frac{\epsilon_+ + \epsilon_-}{2} \right)} + \rho \mathbf{g} \right] + \boxed{\frac{1}{\rho} \nabla \cdot (\nu \rho \nabla \mathbf{v})} + \boxed{\frac{1}{\rho} \nabla \cdot \left(S \rho \nabla \frac{\partial \mathbf{v}}{\partial t} \right)}$$

VISCOSITY

SEMI-IMPLICIT OPERATOR

WAVE TURBULENCE

$$\frac{\partial z_\pm}{\partial t} = -(\mathbf{v} \pm \mathbf{v}_A) \cdot \nabla z_\pm - \frac{z_\pm |z_\mp|}{2\lambda_\perp} + \frac{z_\pm}{4} (\mathbf{v} \mp \mathbf{v}_A) \cdot \nabla (\ln \rho) + \frac{z_\mp}{2} (\mathbf{v} \mp \mathbf{v}_A) \cdot \nabla (\ln |\mathbf{v}_A|)$$

$\nabla \cdot \mathbf{B} = 0$ $\quad p = 2kT\rho/m_p$ $\quad \hat{\mathbf{b}} = \mathbf{B}/|\mathbf{B}|$

$\mathbf{B} = \nabla \times \mathbf{A}$ $\quad \mathbf{g} = -g_0 R_\odot^2 \hat{\mathbf{r}}/r^2$ $\quad \mathbf{v}_A = \mathbf{B}/\sqrt{4\pi\rho}$

$\mathbf{J} = \frac{c}{4\pi} \nabla \times \mathbf{B}$ $\quad \gamma = 5/3$ $\quad B_w = 6.09\,G$ $\quad v_c^2 = \gamma p/\rho$

$$\beta_{T\mathrm{cut}}(T) = \begin{cases} (T/T_{\mathrm{cut}})^{-5/2} & T < T_{\mathrm{cut}} \\ 1, & T \geq T_{\mathrm{cut}} \end{cases}$$

$T_{\mathrm{cut}} = 3.5 \times 10^5\,K$

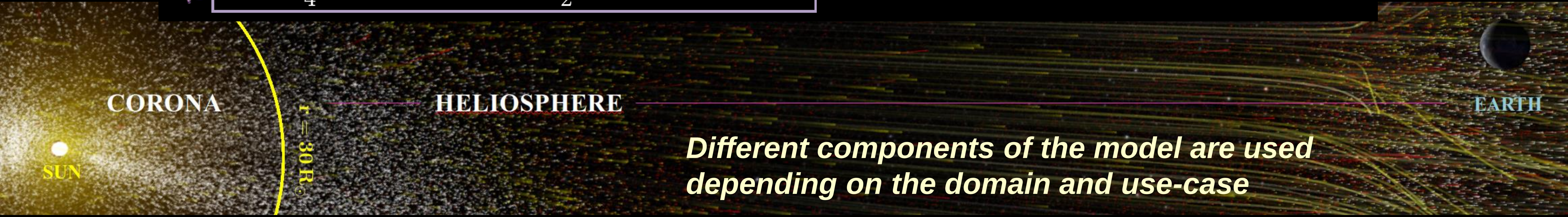
$f(r) = 1 - 0.5 \tanh\left[(r - 10R_\odot)/R_\odot\right]$

$S = (\Delta t^2 \tilde{k}^2)^{-1} \left(C_w^2/(1 - C_f)^2 - 1\right)$

$C_f = \Delta t\, \tilde{k} \cdot \mathbf{v}$

$C_w^2 = 0.25\, \Delta t^2 \tilde{k}^2 (v_c^2 + |\mathbf{v}_A|^2)$

$\tilde{k}^2 = 4\left(\Delta r^{-2} + (r\Delta\theta)^{-2} + (r\Delta\phi \sin\theta)^{-2}\right)$



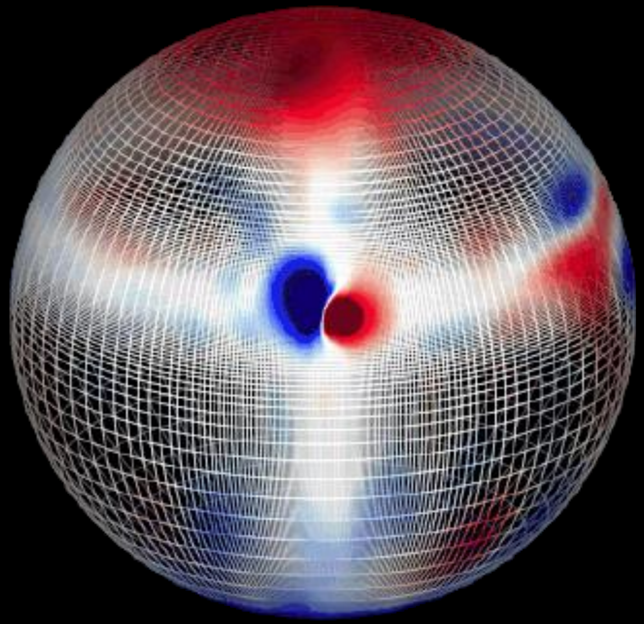
Different components of the model are used depending on the domain and use-case

MAS Code Summary

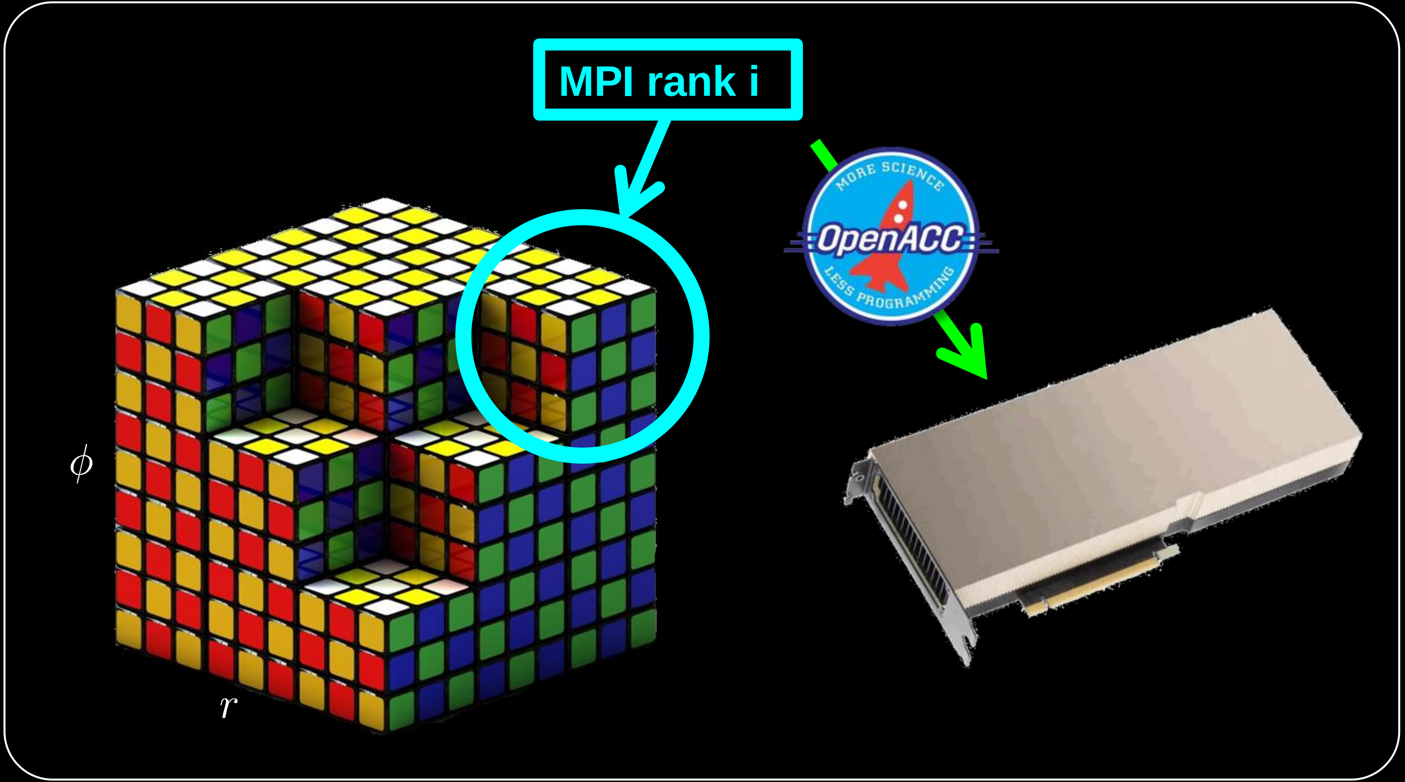
- Non-uniform, logically-rectangular, spherical grid
- MPI parallelism through logical 3D blocks of points
- Each MPI rank computes its local block
- Output data is in hdf file format
- Multiple numerical methods and strategies used
- Here, we focus on two:

  1) Super time-stepping & iterative Krylov schemes for the parabolic operators

  2) GPU acceleration through OpenACC and Fortran standard parallelism



(r_i, θ_j, ϕ_k)



Explicit Super Time-Stepping
&
Implicit Iterative Krylov Solver

The Problem

- MAS has multiple time scales leading to vastly different explicit time-step stability requirements
- In order to make simulations *tractable*, we want to exceed such explicit limits
- Focus on parabolic terms:
  - Implicit methods (need to solve linear system)
  - Explicit sub-cycling (may need MANY cycles)
  - Explicit methods with unconditional stability
- Here, we compare a *super time-stepping* method to an implicit method

$$\Delta t_{\text{flow}} \sim \Delta x / v$$
$$\Delta t_{\text{wave}} \sim \Delta x / v_f$$
$$v_f \gg v$$
$$\Delta t_{\text{para}} \sim \Delta x^2 / \alpha$$
$$\alpha \in \{\kappa, \eta, \nu\}$$

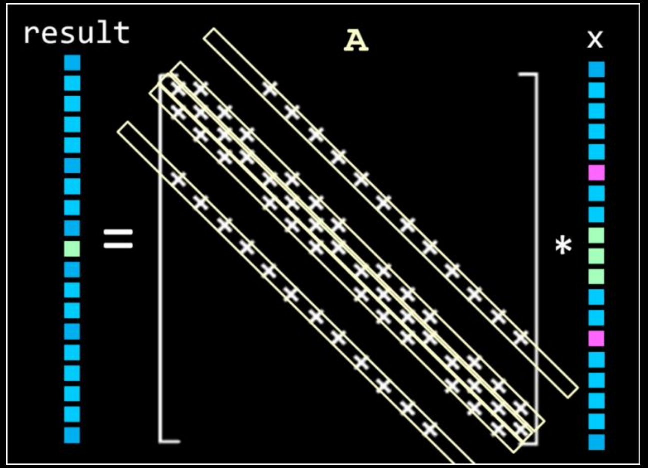
NOTE! When exceeding explicit time-step limits, one must be very careful about accuracy. Using too large of a time step can result in large errors!

Implicit scheme with Krylov solvers: Backward Euler (BE)

- Backward Euler (BE): simplest L-stable method

$$\frac{\partial u}{\partial t} = F(u, \mathbf{r}) \quad \Longrightarrow \quad \frac{u^{n+1} - u^n}{\Delta t} = \mathbf{M}\, u^{n+1}$$

```
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
```



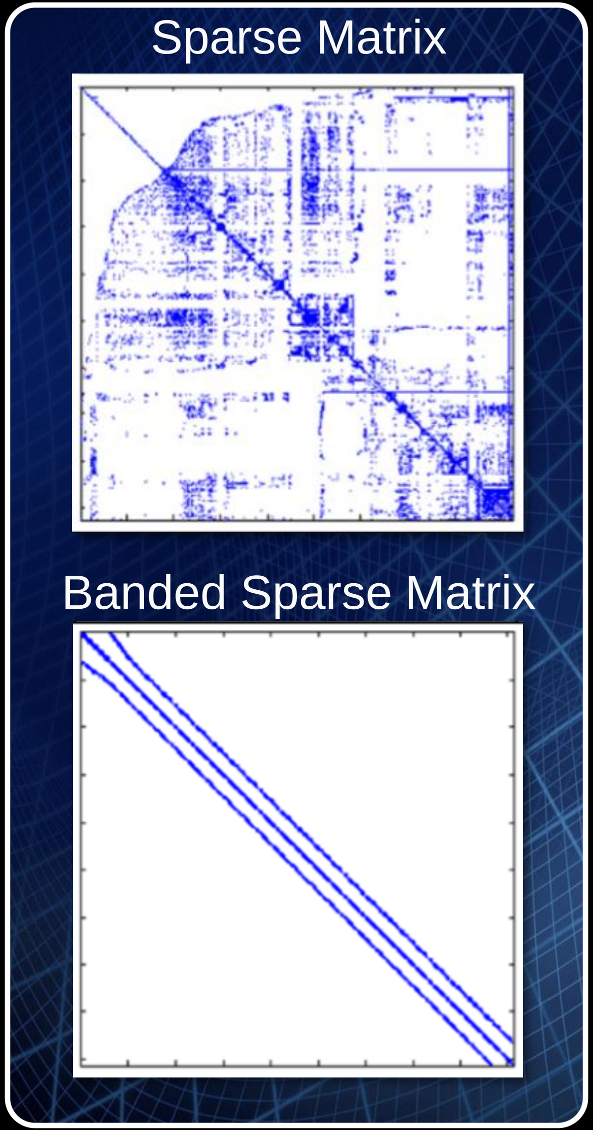
- Applying BE to the parabolic term yields a system of equations to solve

$$(1 - \Delta t\, \mathbf{M})\, u^{n+1} = u^n \quad \Longrightarrow \quad \mathbf{A}\, x = y$$

- To avoid the need for nonlinear solvers, linearize any nonlinear terms (lagged diffusivity).

$$\nabla \cdot \left[\kappa(T^{n+1})\, \nabla T^{n+1}\right] \quad \Longrightarrow \quad \nabla \cdot \left[\kappa(T^{n})\, \nabla T^{n+1}\right]$$

- Linear system $$\mathbf{A}\,\vec{x} = \vec{b}$$
- Matrix is "sparse" (most entries are zero)
- Matrix is banded, allowing us to store it in a modified DIA (diagonal) format for efficient matrix-vector product (stride-1 in memory across rows)
- Since matrix is large, standard "dense" solver algorithms are often too slow. Instead, we use iterative solvers
- The Preconditioned Conjugate Gradient (PCG) is a common method if the matrix is symmetric (or nearly so)



Implicit scheme with Krylov solvers: PCG

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

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

PCG

Point-2-Point comm+sync

Global comm+sync

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

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

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

```
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
- **PC2:** Local sequential algorithm; uses 2nd copy of matrix in a compressed sparse row format

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_ij ≠ 0)
            LU_ij = LU_ij − LU_ik LU_kj
        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
enddo
do i = 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
```

Explicit Super Time-Stepping

- Relatively new methods, relatively uncommon
- Unconditionally stable, but *explicit*!
- **Main idea:** Runge-Kutta method with stages added for more stability, rather than accuracy
- STS methods are used in several MHD codes with success (*FLASH, PLUTO, Lare3D*) and planned for inclusion in others
- Flavors include RKC (Chebyshev-based), RKL (Legendre-based), and RKG (Gegenbauer-based), and they can be recursive or factored
- Here, we demonstrate using the 2nd-order **RKL2** from [Meyer et al, 2014] because it has good stability properties for non-uniform and non-linear diffusion coefficients

RKL Derivation [Meyer et al, 2014]

PDE: $$\frac{\partial u}{\partial t} = \mathbf{M}\, u(t)$$

Solution expansion: $$u(t) = e^{t\mathbf{M}}\, u(0) \approx \left(1 + t\,\mathbf{M} + \frac{1}{2}(t\,\mathbf{M})^2 + \ldots\right) u(0)$$

Discretized form: $$u^{n+1} = e^{z}\, u^{n} \approx \left(1 + z + \frac{z^2}{2} + \ldots\right) u^{n}, \qquad z = \Delta t\,\mathbf{M} \qquad \Delta t \ll 1$$

Multi-step explicit scheme: $u^{n+1} = R(\Delta t\,\mathbf{M})\, u^{n}$

For accuracy, need: $R(z) = 1 + z + z^2/2 + O(z^3)$

For stability, need: $|R(\Delta t\,\lambda)| \leq 1, \ \ \forall \lambda \in \mathbf{M}$

Example: First-order explicit Euler method $(\lambda \in \mathcal{R})$

$$R(z) = 1 + z$$

$$u^{n+1} = (1 + \Delta t\,\mathbf{M})\, u^{n}$$

$$\frac{u^{n+1} - u^{n}}{\Delta t} = \mathbf{M}\, u^{n}$$

$$|1 + \Delta t_{\mathrm{Euler}}\, \lambda| \leq 1$$

$$\Delta t_{\mathrm{Euler}} \leq \frac{2}{|\lambda|_{\max}}$$

1D HEAT EQ: $\Delta t_{\mathrm{Euler}} \leq \dfrac{\Delta x^2}{2}$

Legendre polynomial $P_s(x)$

$$P_j(x) = (1/j)\ [(2\,j - 1)\, x\, P_{j-1}(x) - (j-1)\, P_{j-2}(x)]$$

$$|P_s(x)| \leq 1, \ x \in [-1, 1]$$

RKL: $R(z) = a_s + b_s\, P_s(1 + w_1\, z)$

Accuracy

$O(\Delta t)$

$$a_s = 1 - b_s$$
$$b_s = 1$$
$$w_1 = \frac{2}{s^2 + s}$$

$O(\Delta t^2)$

$$a_s = 1 - b_s$$
$$b_s = \frac{s^2 + s - 2}{2\, s\, (s+1)}$$
$$w_1 = \frac{4}{s^2 + s - 2}$$

Stability

$$-1 \leq 1 + w_1\, \Delta t\, \lambda \leq 1$$

Select s, get max dt:

$$\Delta t \leq \frac{\Delta t_{\mathrm{Euler}}}{w_1}$$

Select dt, get min s:

$$w_1 \leq \frac{\Delta t_{\mathrm{Euler}}}{\Delta t}$$

Recursion relation leads to easy implementation

Predictive Science Inc.

RKL2

Point-2-Point
comm+sync

$$M_0 = \boxed{\mathbf{M}\, u^n}$$

$$u_1 = u^n + \tilde{\mu_1}\, \Delta t\, M_u$$

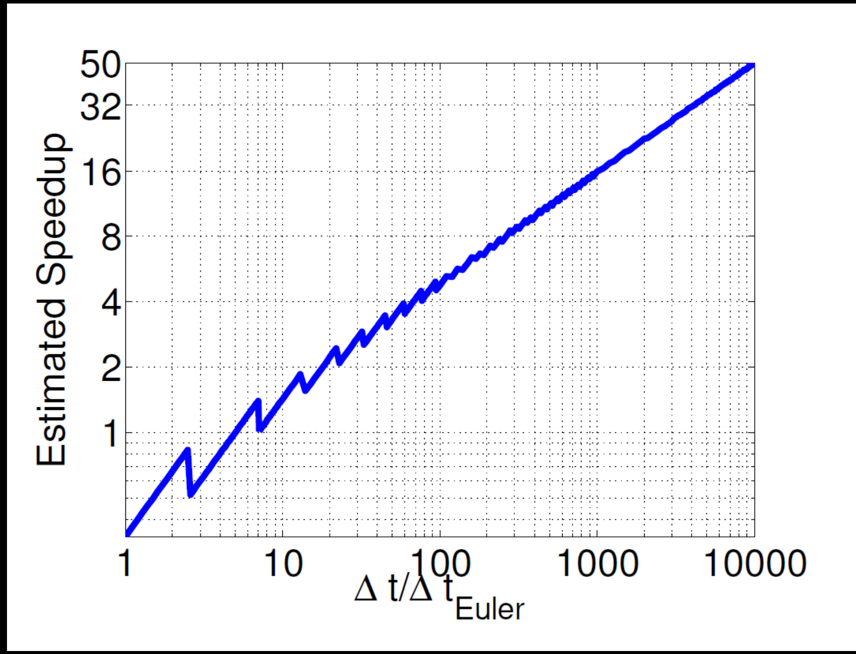
$$\text{do k} = 2:\text{s}$$

$$u_k = \mu_j\, u_{k-1} + \nu_j\, u_{k-2} + (1 - \mu_k - \nu_k)\, u^n$$

$$+\tilde{\mu_k}\, \Delta t\, \boxed{\mathbf{M}\, u_{k-1}} + (b_k - 1)\, \tilde{\mu_k}\, \Delta t\, M_0$$

$$\text{enddo}$$

$$u^{n+1} = u_s,$$



$$b_0 = b_1 = b_2 = \frac{1}{3}, \qquad b_k = \frac{k^2 + k - 2}{2\,k\,(k+1)},$$

$$\tilde{\mu_1} = \frac{4}{3\,(s^2 + s - 2)}, \qquad \mu_k = \frac{2\,k - 1}{k}\,\frac{b_k}{b_{k-1}},$$

$$\tilde{\mu_k} = \frac{4\,(2\,k-1)}{k\,(s^2+s-2)}\,\frac{b_k}{b_{k-1}}, \qquad \nu_k = -\frac{k-1}{k}\,\frac{b_k}{b_{k-2}}.$$

Number of required STS steps:

$$s = \frac{1}{2}\left\lceil \sqrt{9 + 16\,\frac{\Delta t}{\Delta t_{\mathrm{Euler}}}} - 1 \right\rceil$$

Gershgorin circle estimate of Euler time step:

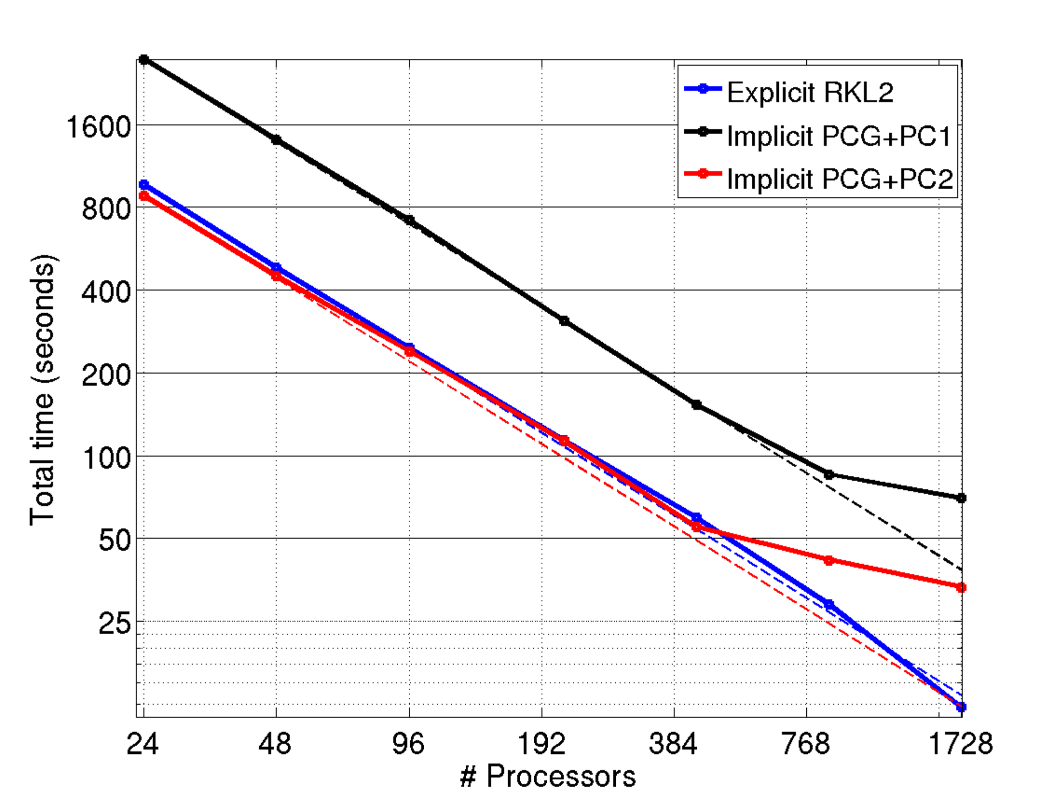
$$\Delta t_{\mathrm{Euler}} \leq \frac{2}{|\lambda|_{\mathrm{max}}}$$

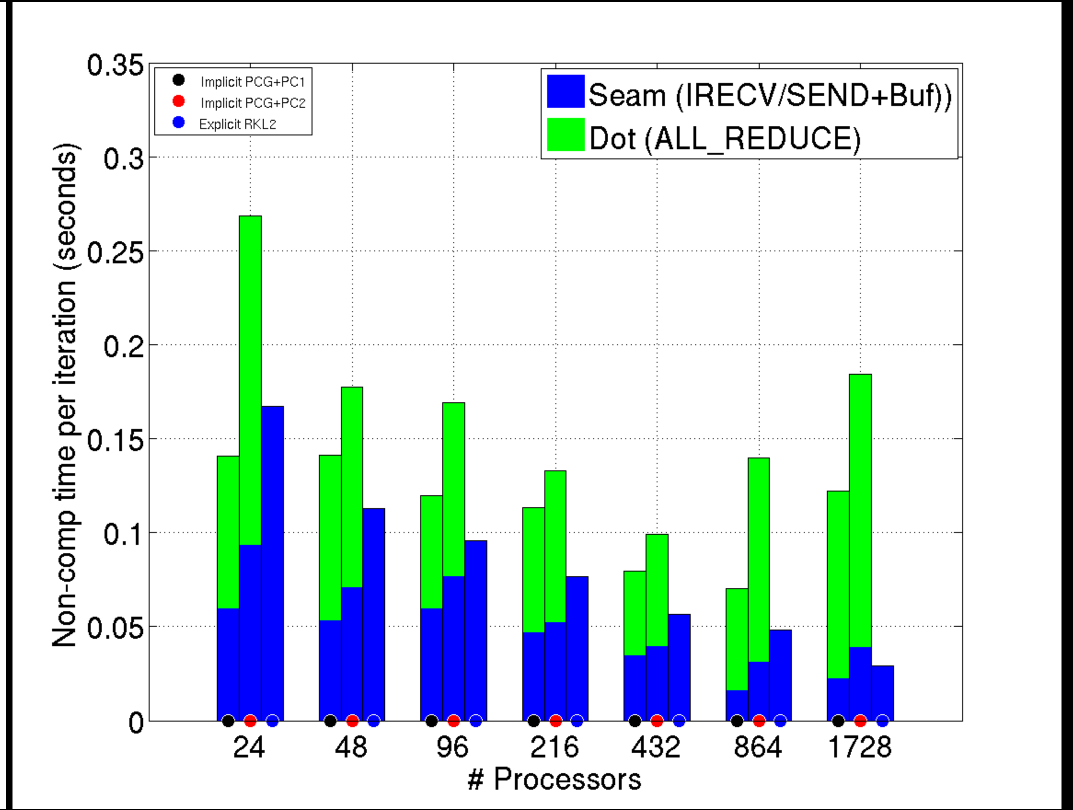
$$|\lambda|_{\mathrm{max}} \leq \max\left\{ \sum_{i=1}^{N} |A_{i,j}|, \forall j \text{ rows} \right\}$$

Performance Results

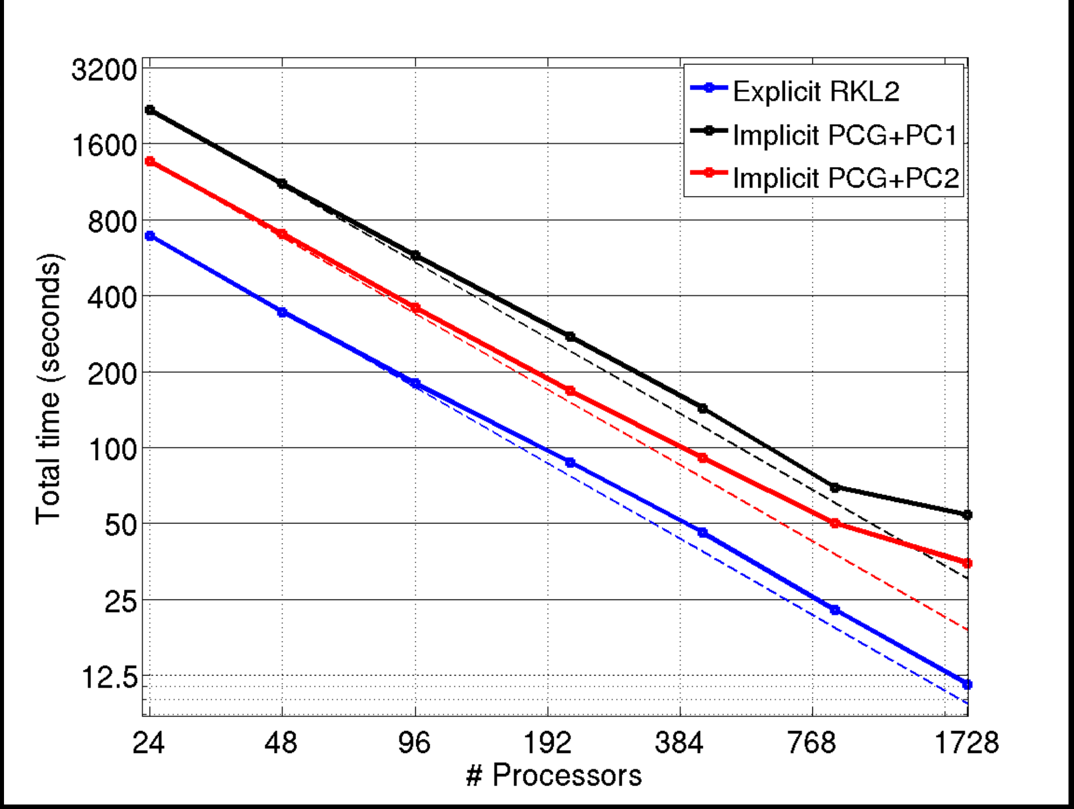
- Production-level full-model MAS run of the solar corona
- For thermal conduction operator, RKL2 is equal to performance of PCG+PC2 for low # of nodes, but scales much better, leading to 2X speedup
- For viscosity operator, RKL2 is 2x faster than PCG+PC2 overall, and with its better scaling, is up to 3x faster

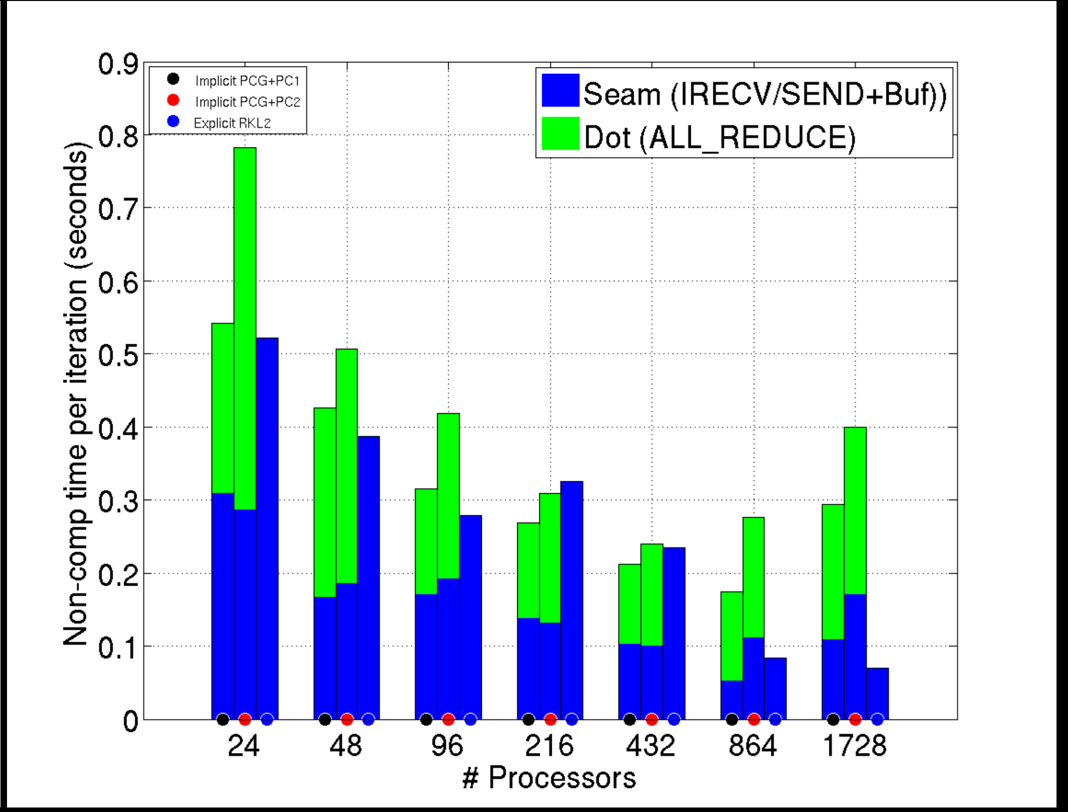
THERMAL CONDUCTION





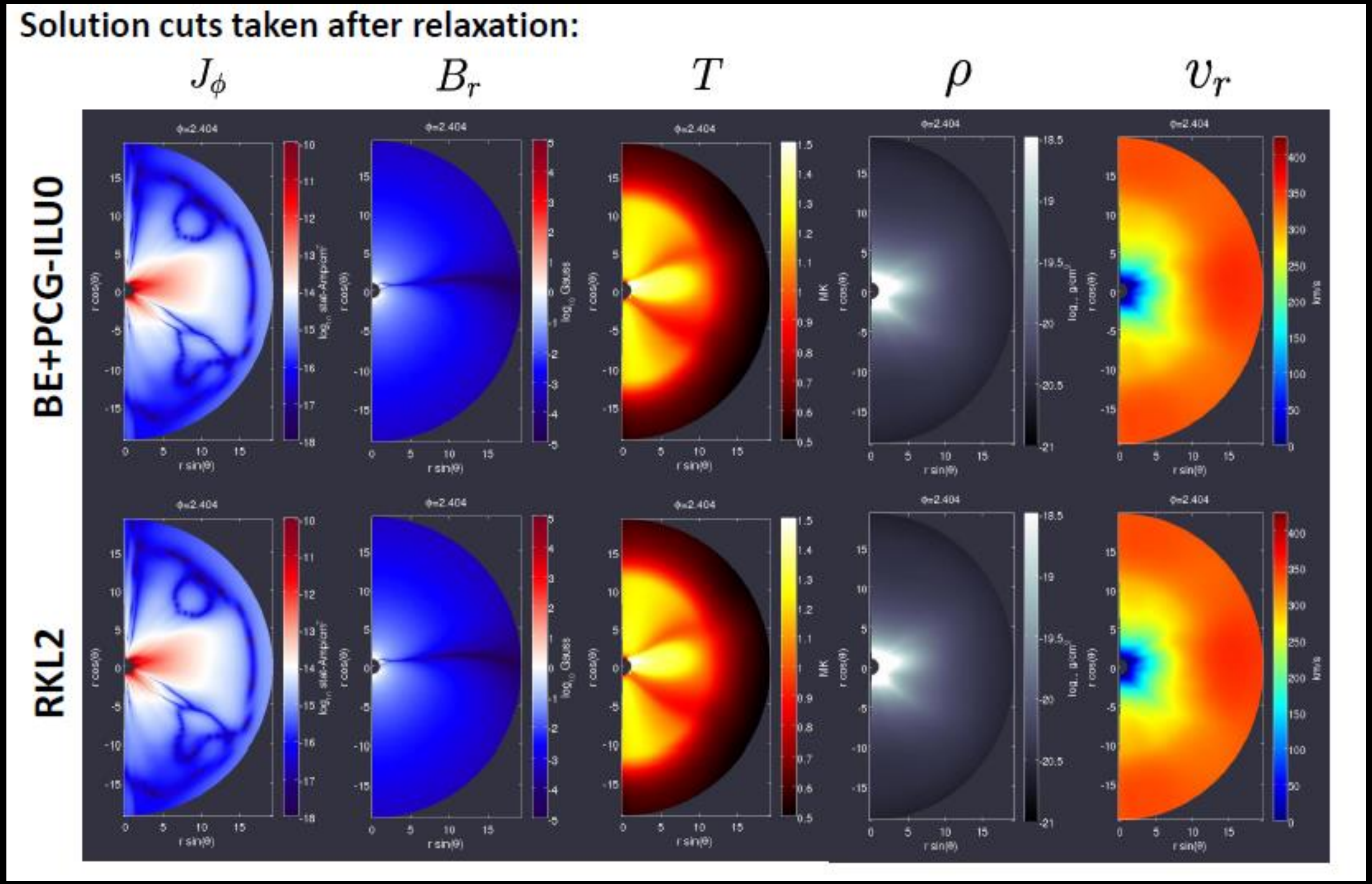
VISCOSITY

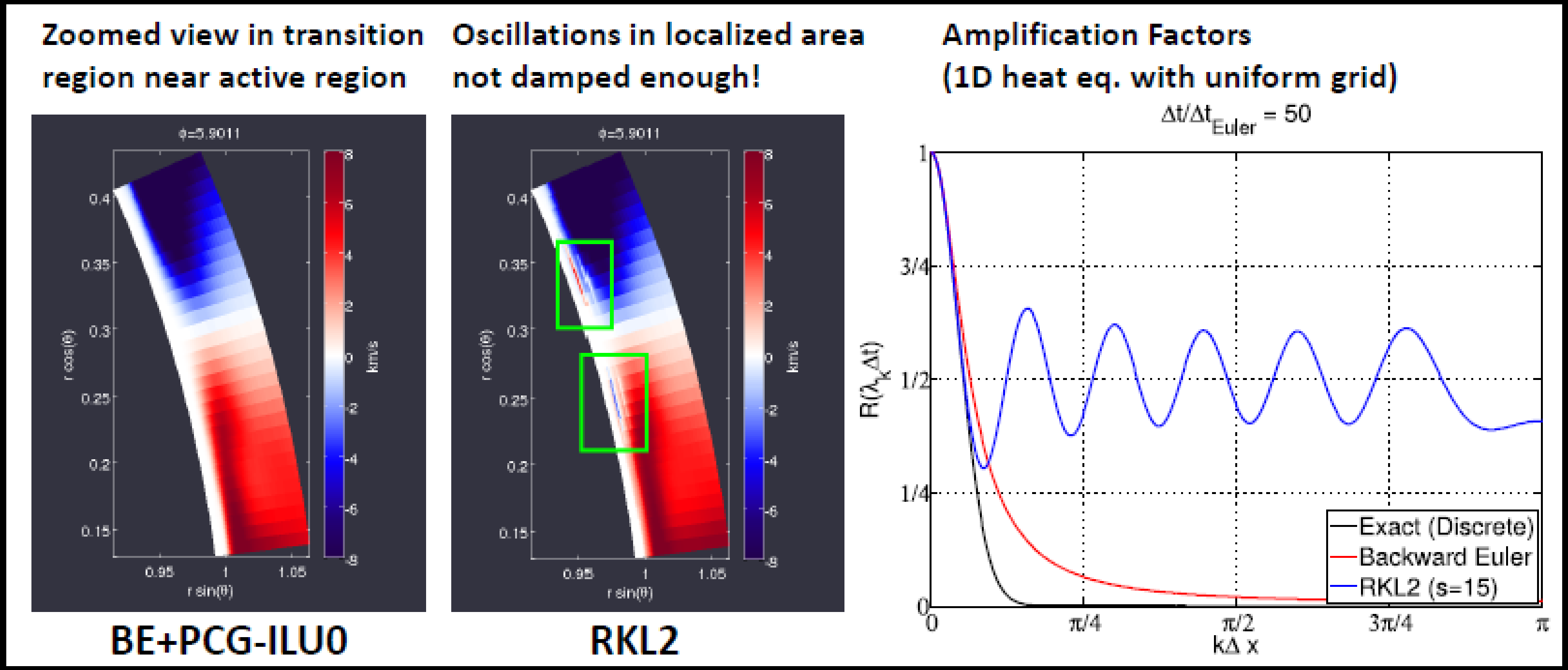




STS Problems

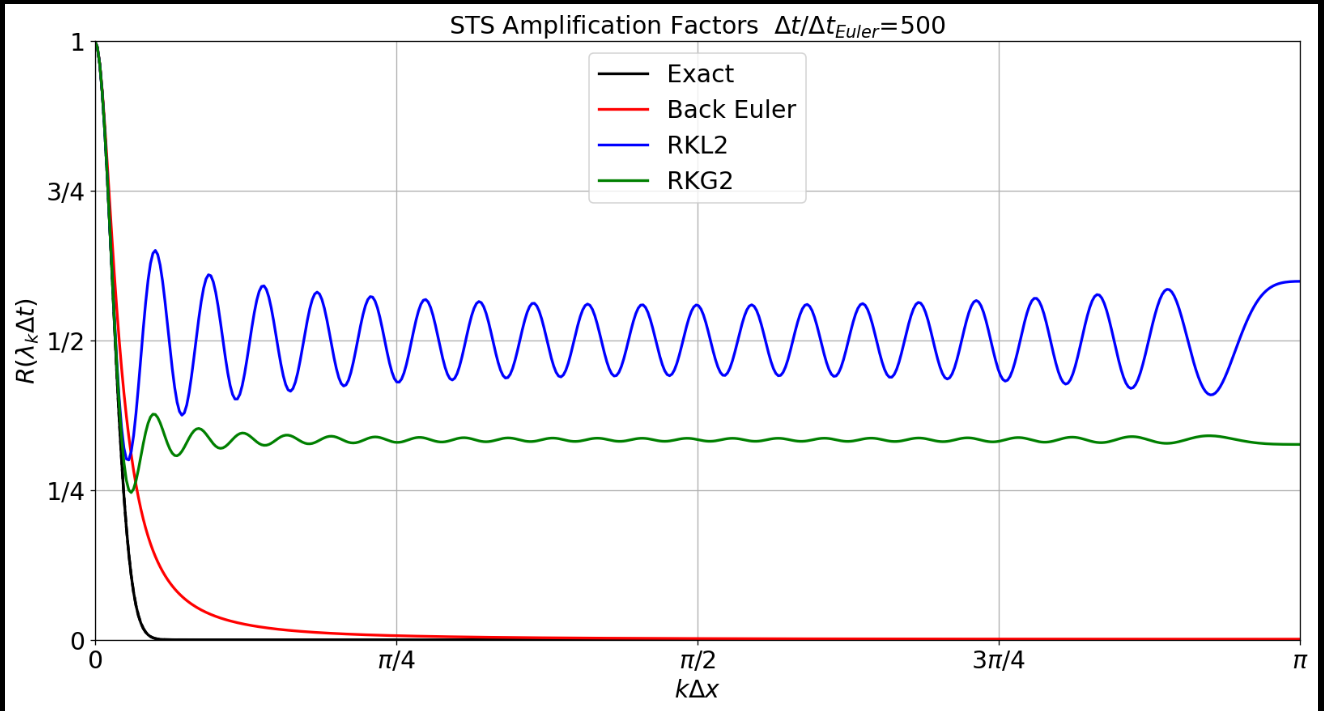
- Overall solution very similar over most of the domain
- Closer inspection shows solution artifacts and gridding
- STS does not damp high wave modes efficiently
- Since viscosity used to damp oscillations in MAS, STS fails to damp them enough





STS Problems cont.

- Can sub-cycle STS, damping high modes (~50 times to get to FP-64 numeric zero)
- However, this can eliminate the performance improvement of STS over PCG!
- Other STS scheme have different amplification factors. The RK**G**2 scheme would only need 30 sub-cycles, but its speedup is slightly lower than RKL2



- New research being done by C.D. Johnston & L. Daldorff may solve the issue by defining a physics-based flux-time-step limit that dynamically sets the needed number of STS sub-cycles
- This has great promise to make STS methods much more robust, while retaining their performance advantage

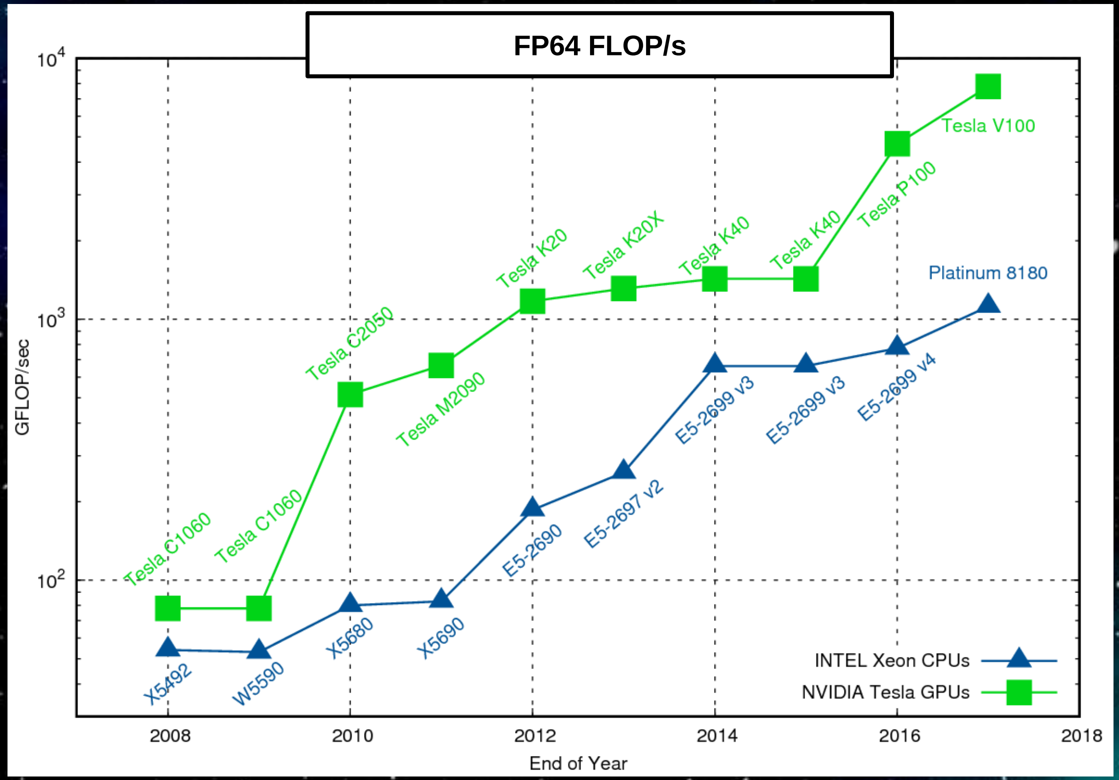
GPU Acceleration with OpenACC & Fortran Standard Parallelism

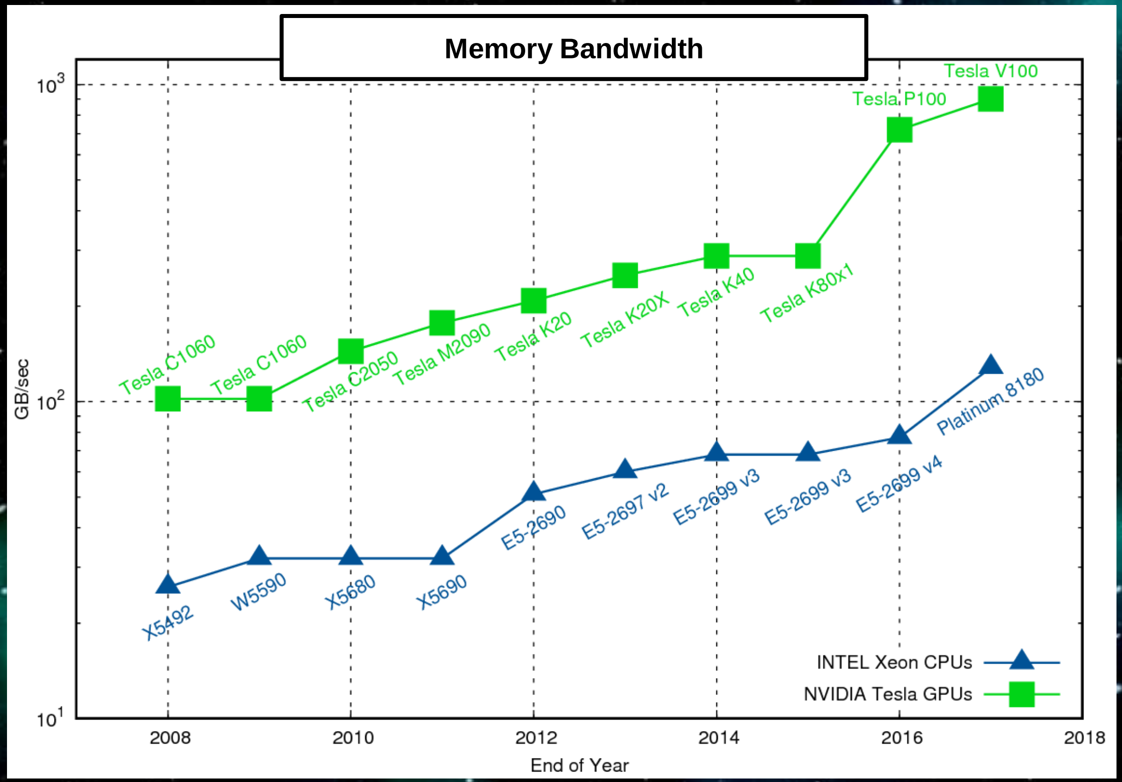
Why use GPUs?

1) Performance (FLOP/s and ***Memory Bandwidth***)

2) Compact performance (workstations, HPC real estate)

3) Can save energy and money





OpenACC

More Science, Less Programming

```
C:            #pragma acc
Fortran:  !$acc
```

"OpenACC is a user-driven directive-based performance-portable parallel programming model. It is designed for scientists and engineers interested in porting their codes to a wide-variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required with a low-level model." - ***openacc.org***

NVIDIA

- Can produce single source code base
- Low-risk (can compile to CPU as before)
- Multiple Targets (**GPU**, Multicore x86, FPGA, etc.)
- Vendor-independent (**NVIDIA**, GCC, AMD GCN, etc.)
- Great for rapid development and accelerating legacy codes

OpenACC Example

Accelerating SAXPY:

```
for (i=0; i<N; i++)
  y[i] = a*x[i] + y[i];
```

NVIDIA CUDA

```
__global__ void saxpy(int N, float a,
                      float * restrict x,
                      float * restrict y){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < N) y[i] = a*x[i] + y[i];
}
...
const int BLOCK_SIZE=2048;
float *d_x,*d_y;
dim3 dimBlock(BLOCK_SIZE);
dim3 dimGrid((int)ceil((N+0.0)/dimBlock.x));
...
cudaMalloc( (void **) &d_x, sizeof(float)*N);
cudaMalloc( (void **) &d_y, sizeof(float)*N);
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

saxpy<<<dimGrid,dimBlock>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
cudaFree(d_x);
cudaFree(d_y);
```

OpenACC

```
#pragma acc enter data copyin(x,y)
#pragma acc parallel present(x,y)
{
#pragma acc loop gang vector(32)
for (i=0; i<N; i++)
   y[i] = a*x[i] + y[i];
}
#pragma acc update self(y)
#pragma acc exit data delete(x,y)
```

```
#pragma acc kernels
for (i=0; i<N; i++)
    y[i] = a*x[i] + y[i];
```

OpenACC Implementation

Basic Loop

```
!$acc parallel default(present)
!$acc loop collapse(2)
   do j=1,n
     do i=1,m
       y(i,j) = a*x(i,j) + y(i,j)
     enddo
   enddo
!$acc end parallel
```

Fortran Array-syntax

```
!$acc kernels default(present)
    y(:,:) = a*x(:,:) + y(:,:)
!$acc end kernels
```

Reductions

```
!$acc parallel loop present(y)
!$acc& reduction(+:sum)
   do j=1,m
     sum = sum + y(j)
   enddo
```

```
!$acc parallel loop collapse(2)
!$acc& default(present)
   do j=1,m
     do i=1,n
!$acc atomic update
       sum(i) = sum(i) + y(i,j)
     enddo
    enddo
```

CPU↔GPU Data transfers

”y” is allocated and initialized on CPU.

`!$acc enter data copyin (y)` (Can now use “y” in OpenACC regions)

`!$acc update self (y)` (CPU version of “y” updated for I/O, etc.)

`!$acc exit data delete(y)` (Free up GPU memory)

OpenACC Implementation Cont.

Multiple GPUs on one or more servers

MPI-3

(here we assume #GPUs/node = #ranks/node)

```
call MPI_Comm_split_type (MPI_COMM_WORLD,MPI_COMM_TYPE_SHARED,
&                                 0,MPI_INFO_NULL,comm_shared,ierr)
call MPI_Comm_size (comm_shared, nprocsh, ierr)
call MPI_Comm_rank (comm_shared, iprocsh, ierr)
igpu = MODULO(iprocsh, nprocsh)
!$acc set device_num(igpu)
```

Use GPU data directly with MPI calls (“CUDA-aware MPI”)

```
!$acc host_data use_device(y) if_present
  call MPI_Allreduce(MPI_IN_PLACE,y,n,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD,ierr)
!$acc end host_data
```

OpenACC Implementation Cont.

<2%

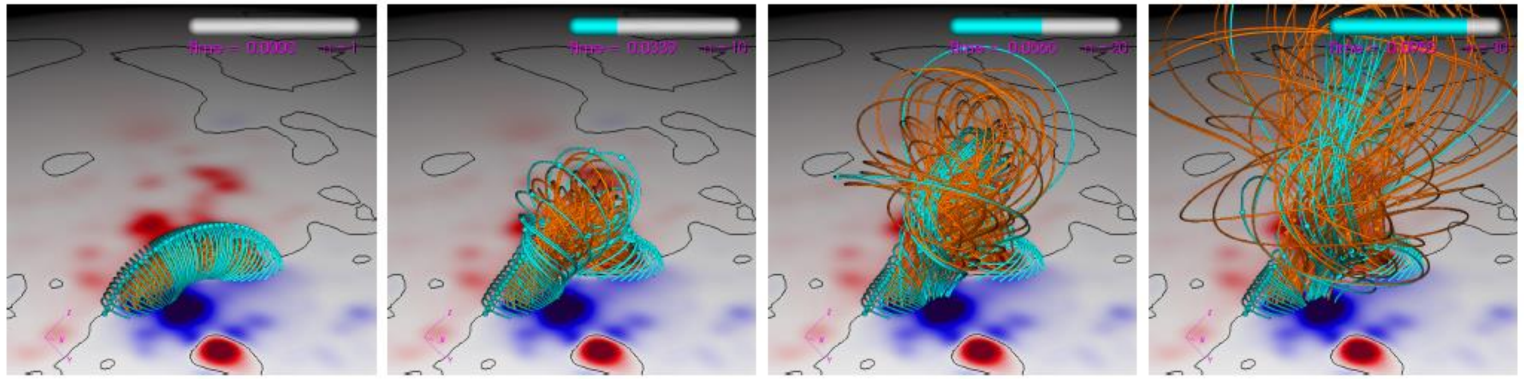
OpenACC comment lines added

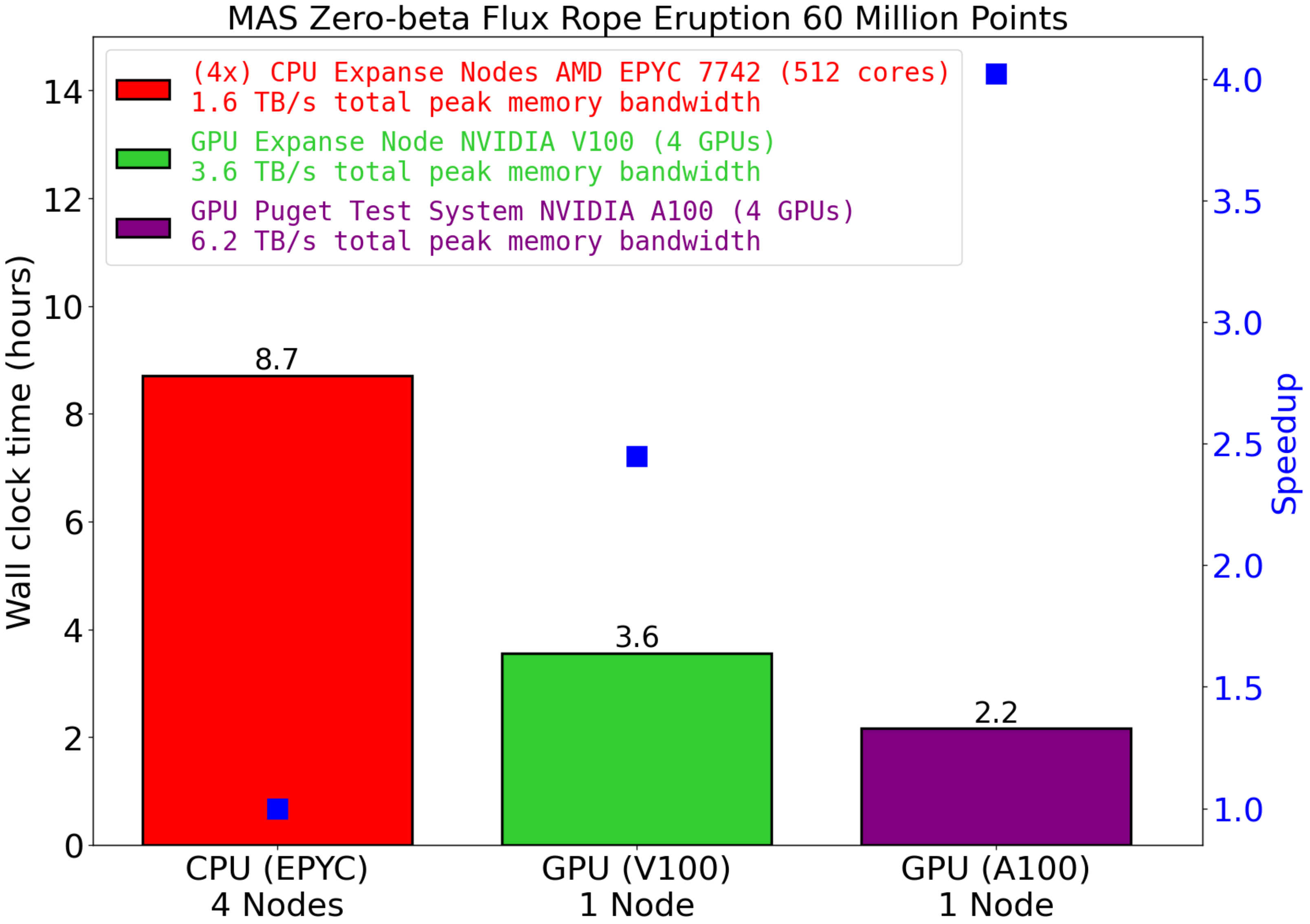
<5%

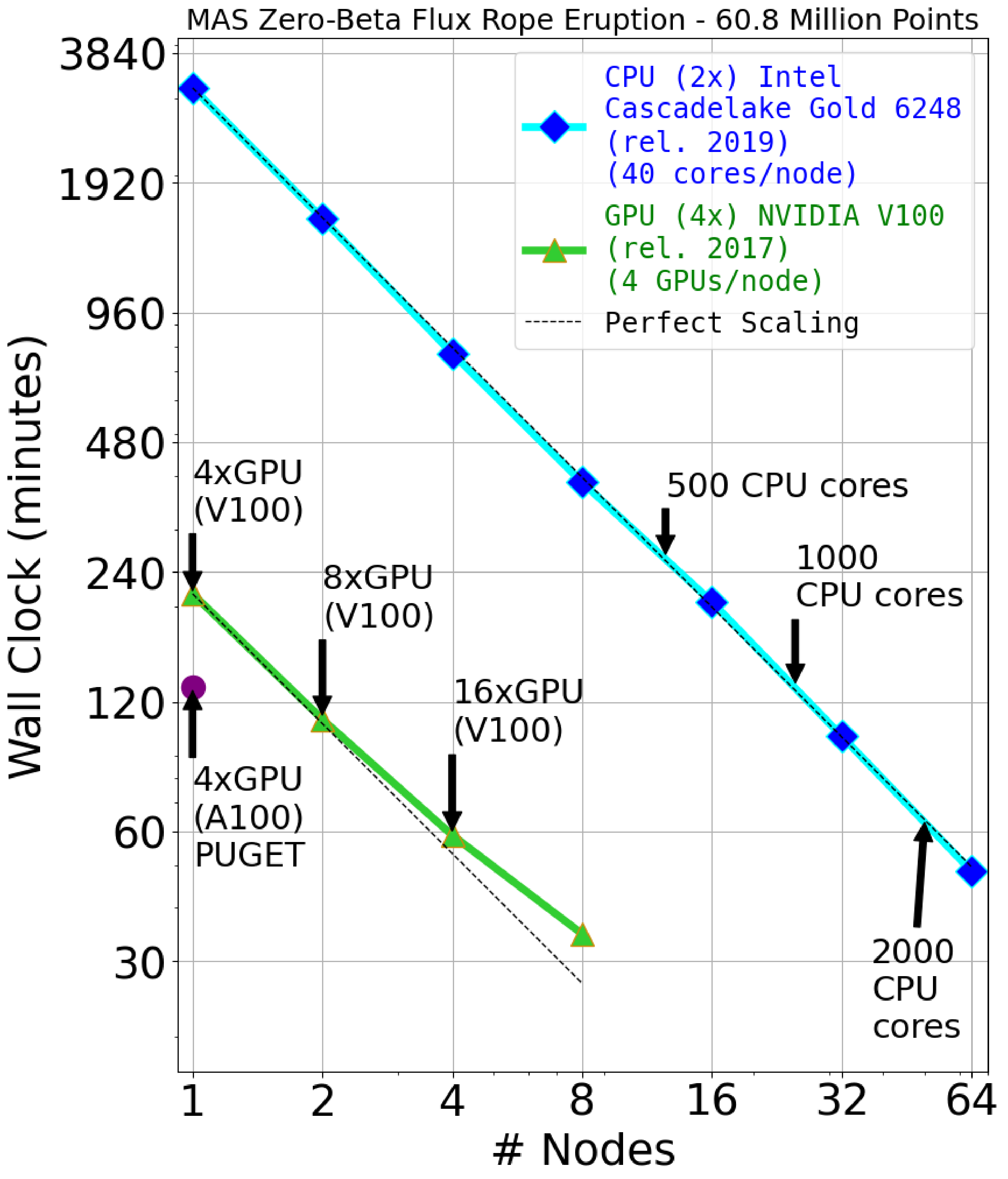
Total modified lines of code

Single portable source for GPU and CPU!

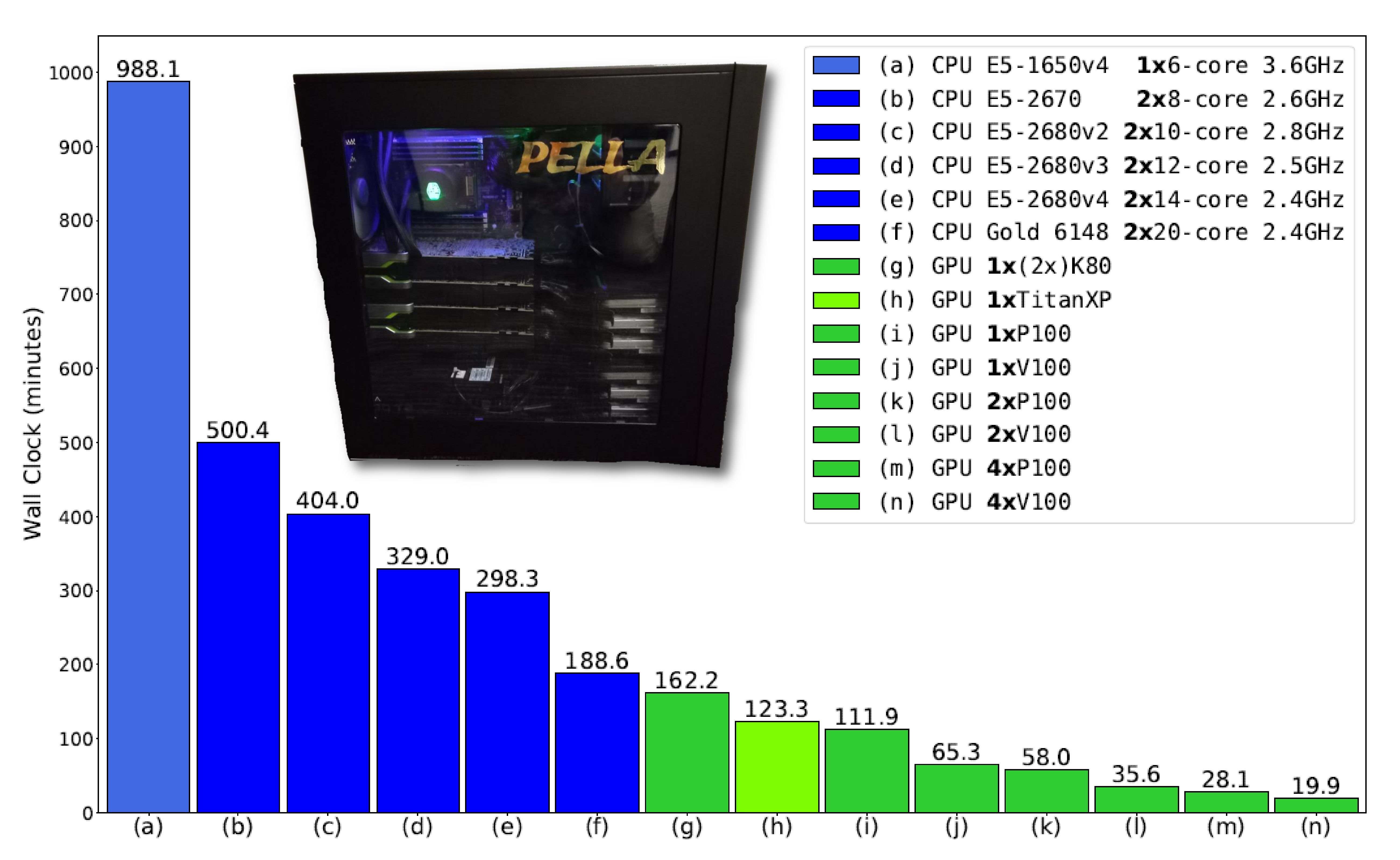
OpenACC Performance







OpenACC Performance (Single Server)

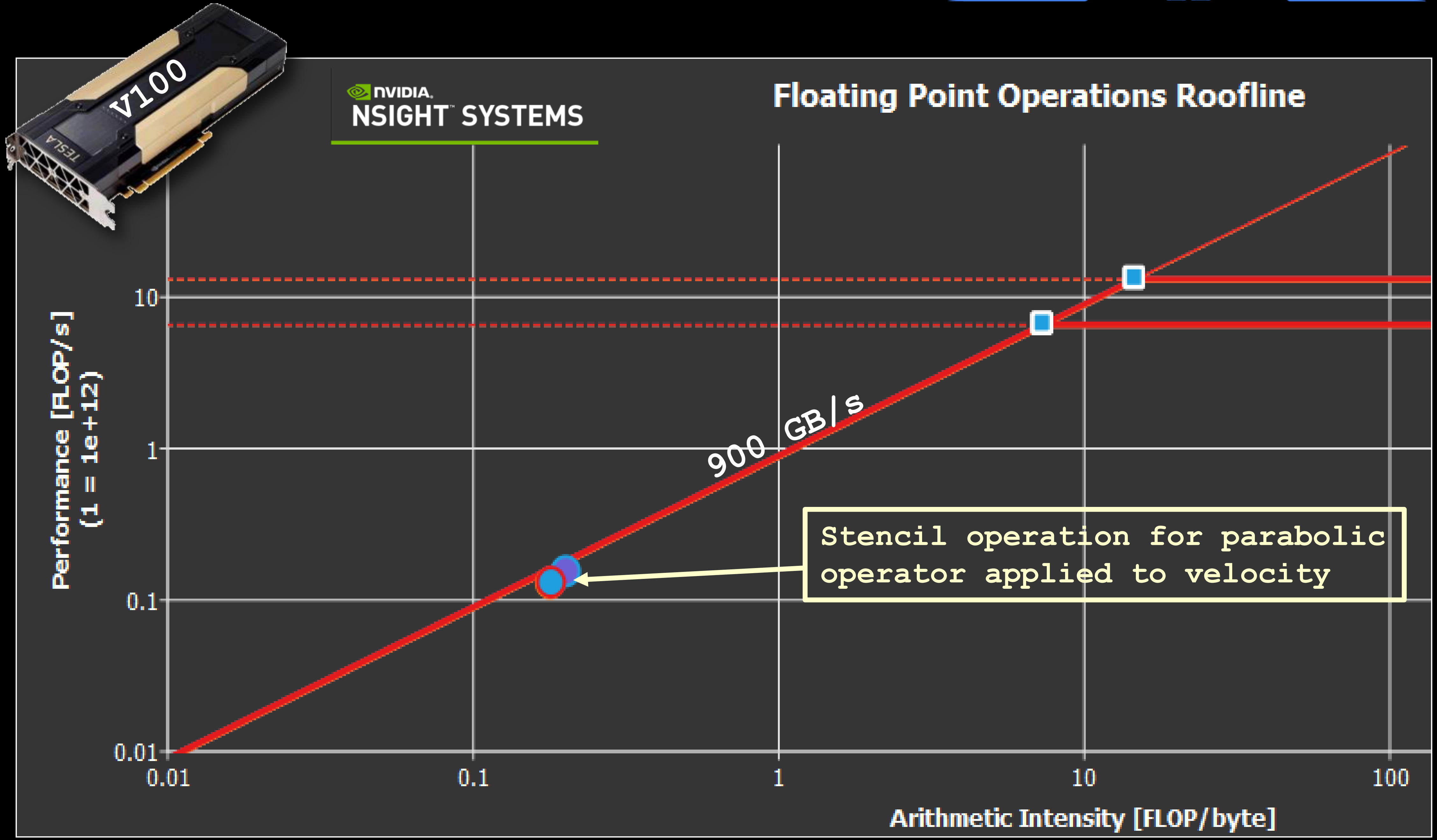


Roofline Analysis

- A **roofline** model shows how well given hardware is being utilized compared to the theoretical maximum for the given code segment's arithmetic intensity

```
do k=2,nz-1
 do j=2,ny-1
  do i=2,nx-1
    result(i,j,k) =
    -6*x(i,   j,   k   )
     + x(i-1,j   ,k   )
     + x(i+1,j   ,k   )
     + x(i   ,j-1,k   )
     + x(i   ,j+1,k   )
     + x(i   ,j   ,k-1)
     + x(i   ,j   ,k+1)
  enddo
 enddo
enddo
```

Floating Point Operations: **7 FLOP**
Data movement (loads/stores): 8*8 Bytes = **64 Bytes**
Arithmetic Intensity: FLOP / BYTE = 0.11



DO CONCURRENT

- Introduced in ISO Standard Fortran 2008
- Indicates loop can be run with out-of-order execution
- Can be hint to the compiler that loop may be parallelizable
- Current specification has no support for reductions, atomics, device selection, conditionals, etc.

```
do i=1,N
  do j=1,M
      Computation
  enddo
enddo
```

```
do concurrent (i=1:N,j=1:M)
  Computation
enddo
```

Compiler	Version	DO CONCURRENT parallelization support
nvfortran	≥ 20.11	Parallelizable on CPU and GPU with "-stdpar" flag. Locality of variables is supported.
ifort	≥ 19.1	Parallelizable on CPU with "-fopenmp" flag. Locality of variables is supported.
gfortran	≥ 9	Parallelizable on CPU with "-ftree-parallelize-loops=X" flag. Locality of variables is not supported.

OpenACC → Standard Parallelism

Original Non-Parallelized Code

```
do k=1,np
  do j=1,nt
    do i=1,nrm1
      br(i,j,k)=(phi(i+1,j,k)-phi(i,j,k ))*dr_i(i)
    enddo
  enddo
enddo
```

OpenACC Parallelized Code

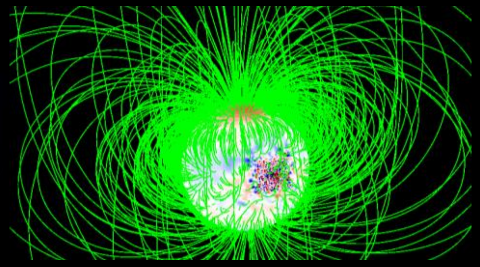
```
!$acc enter data copyin(phi,dr_i)
!$acc enter data create(br)
!$acc parallel loop default(present) collapse(3) async(1)
do k=1,np
  do j=1,nt
    do i=1,nrm1
      br(i,j,k)=(phi(i+1,j,k)-phi(i,j,k))*dr_i(i)
    enddo
  enddo
enddo
!$acc wait
!$acc exit data delete(phi,dr_i,br)
```

Fortran's DO CONCURRENT

```
do concurrent (k=1:np,j=1:nt,i=1:nrm1)
  br(i,j,k)=(phi(i+1,j,k)-phi(i,j,k ))*dr_i(i)
enddo
```

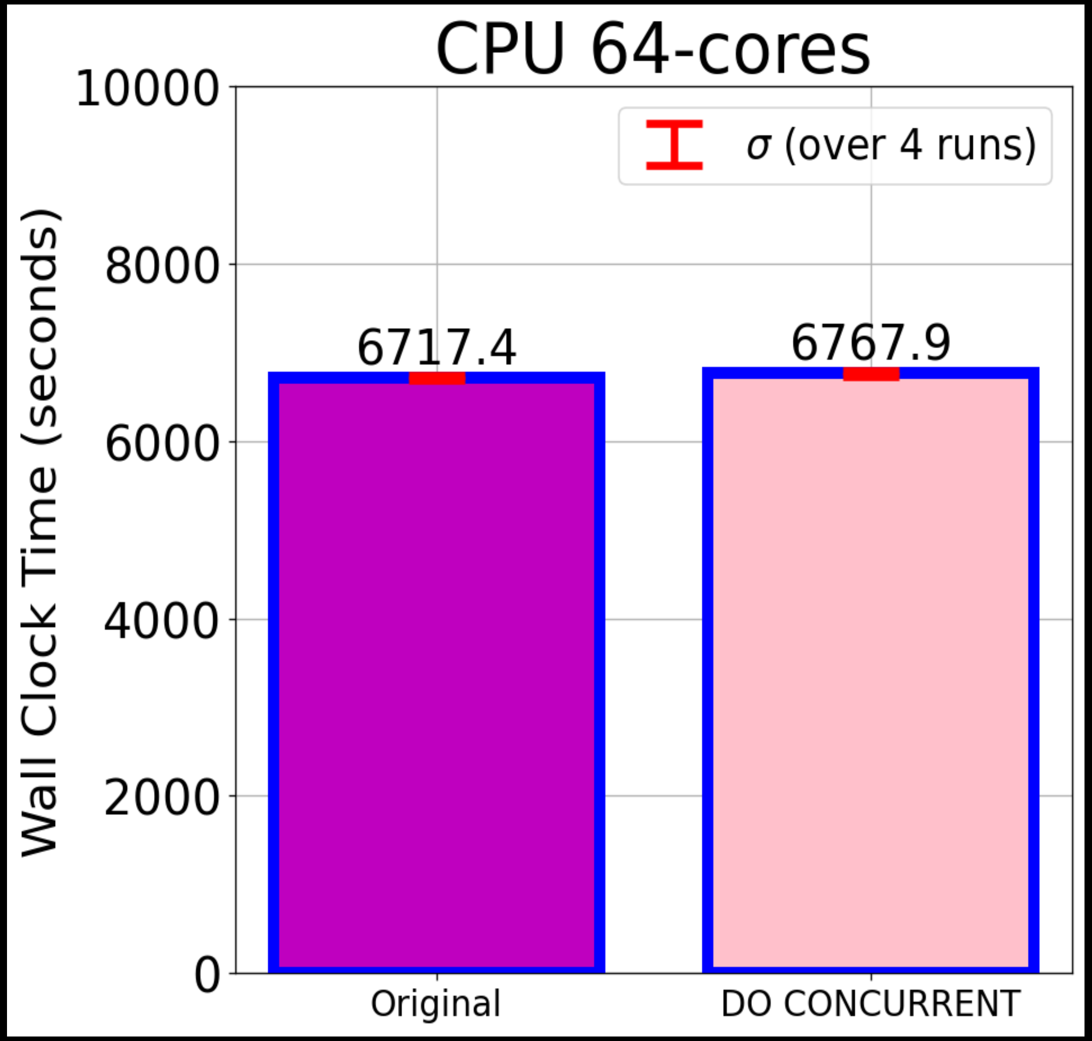
Performance of "do concurrent" vs. OpenACC

- We tested replacing OpenACC with DC in our potential field solver POT3D
- The resulting code reduced the number of needed OpenACC directives substantially
- We plan to implement DC into the MAS code as well

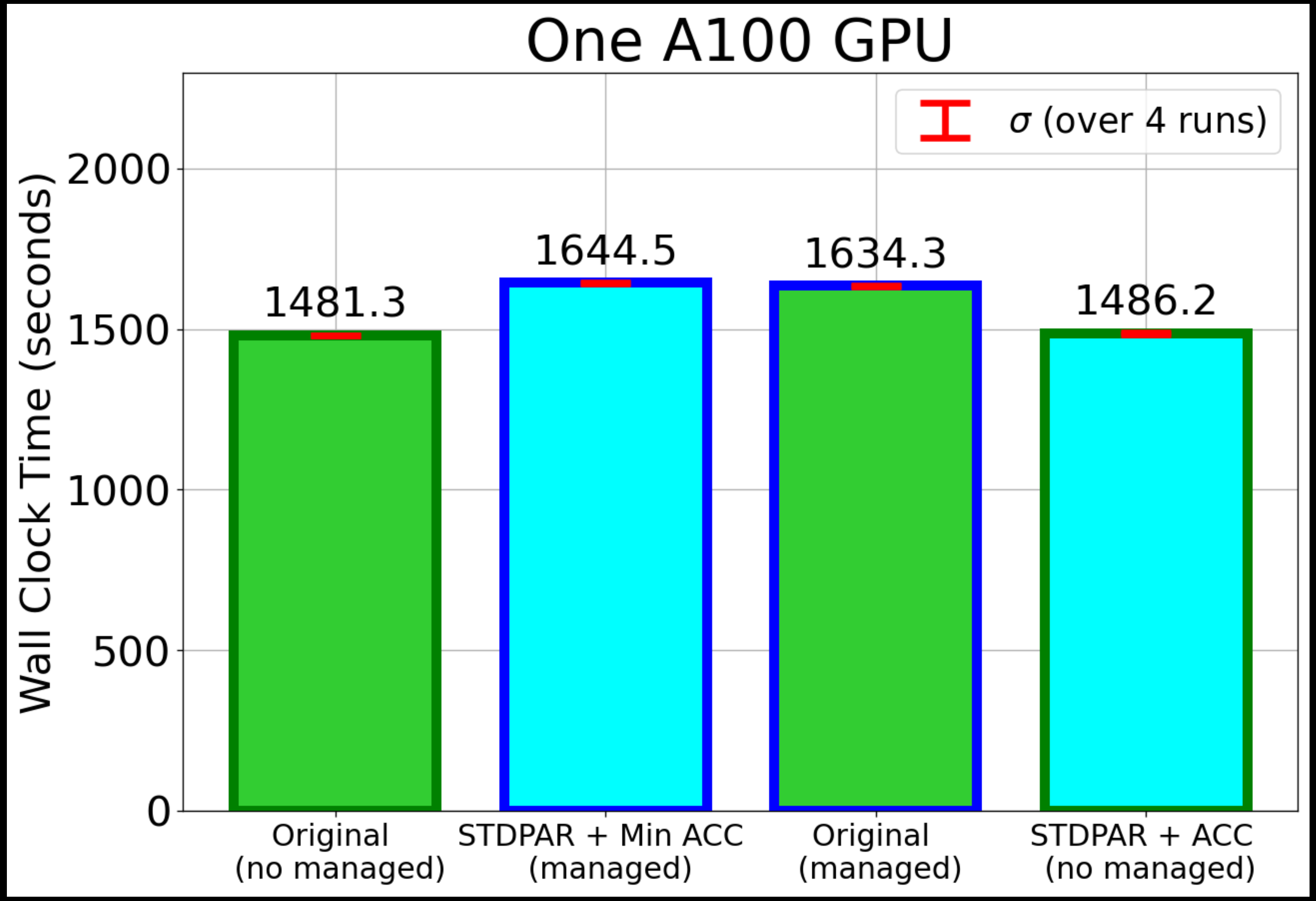


github.com/predsci/POT3D

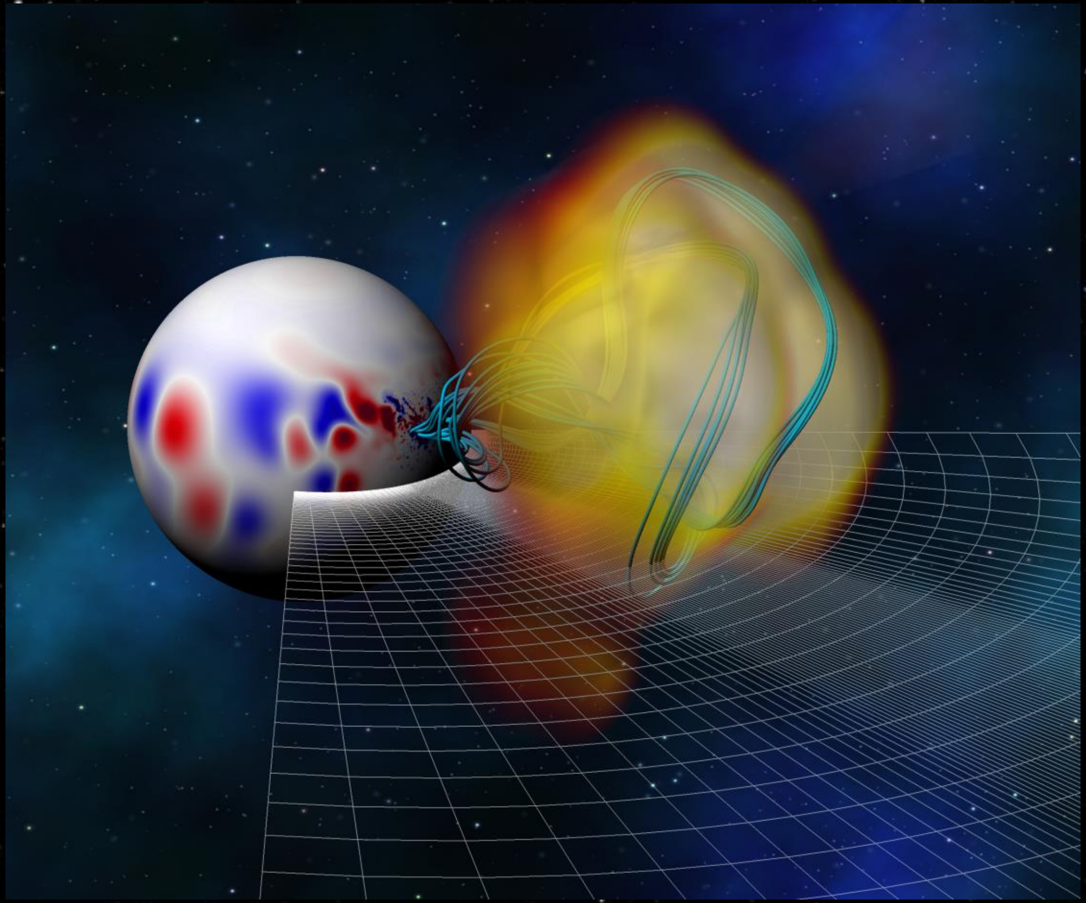
CPU (dual-socket EPYC 7742)

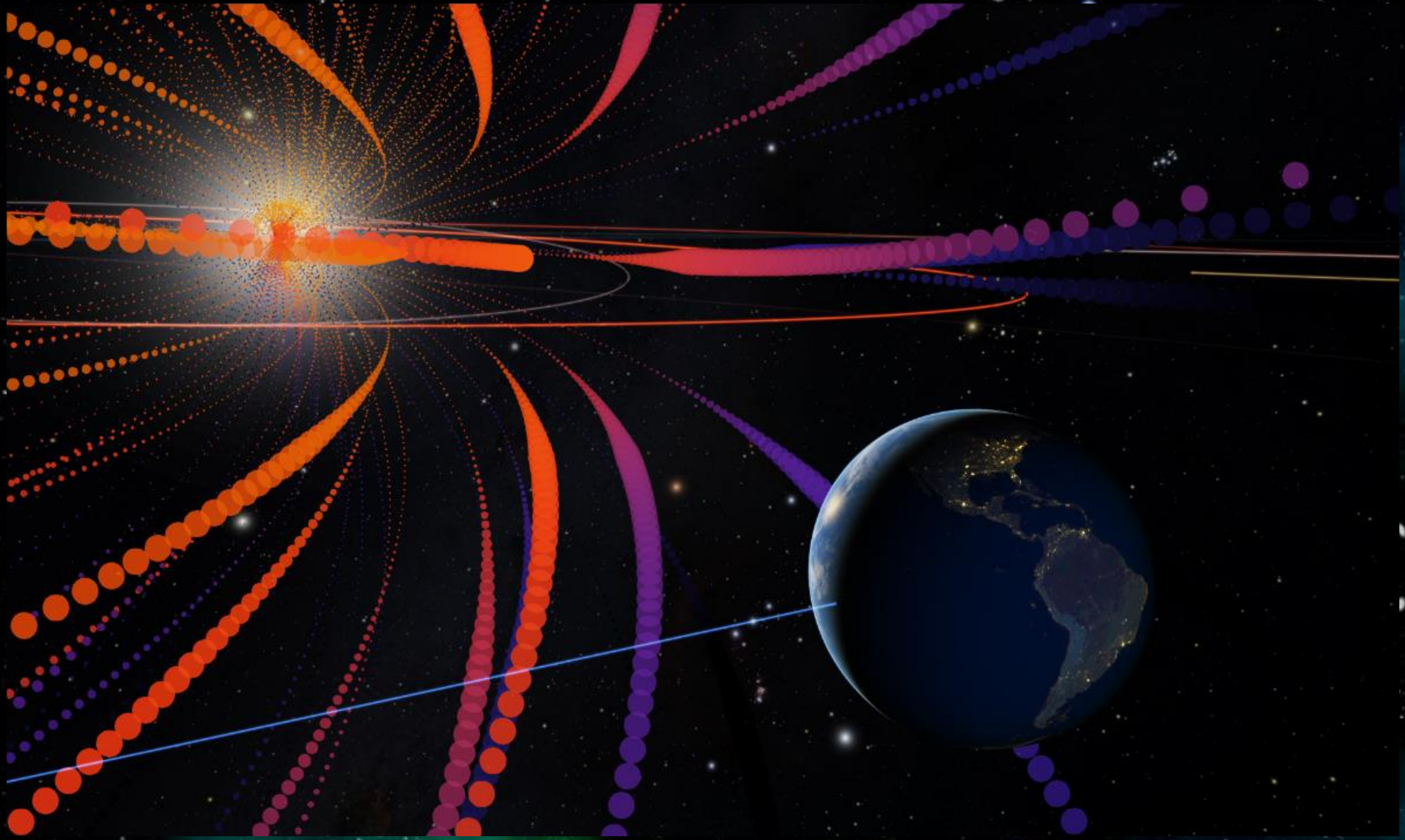


GPU (A100-40GB)



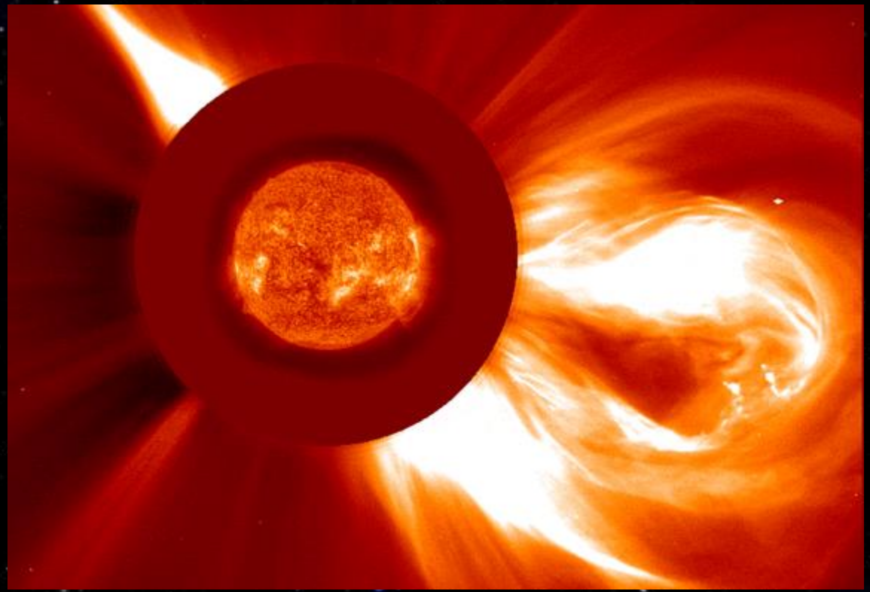
SOLAR STORM SIMULATIONS





Solar Storm Simulation Example

- Modeling a solar storm can be quite complicated
- In order to make routine simulations available (for space weather applications, etc.), we have created CORHEL-AMCG
- CORHEL-AMCG uses a web-based interface to allow a non-expert user to design and run a realistic solar storm simulation
- When completed, CORHEL-AMCG will be delivered to NASA's Community Coordinated Modeling Center for public use



CORHEL-AMCG

NASA

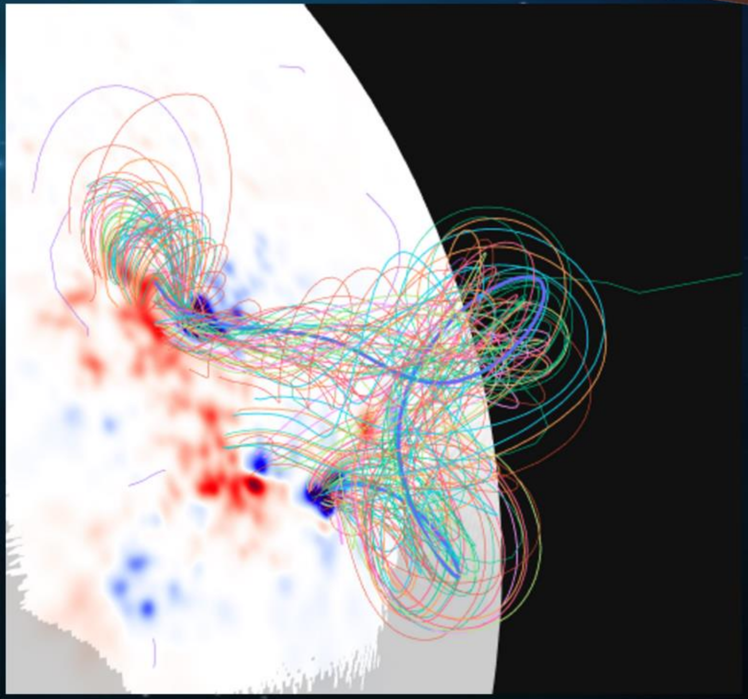
COMMUNITY
COORDINATED
MODELING
CENTER

ccmc.gsfc.nasa.gov

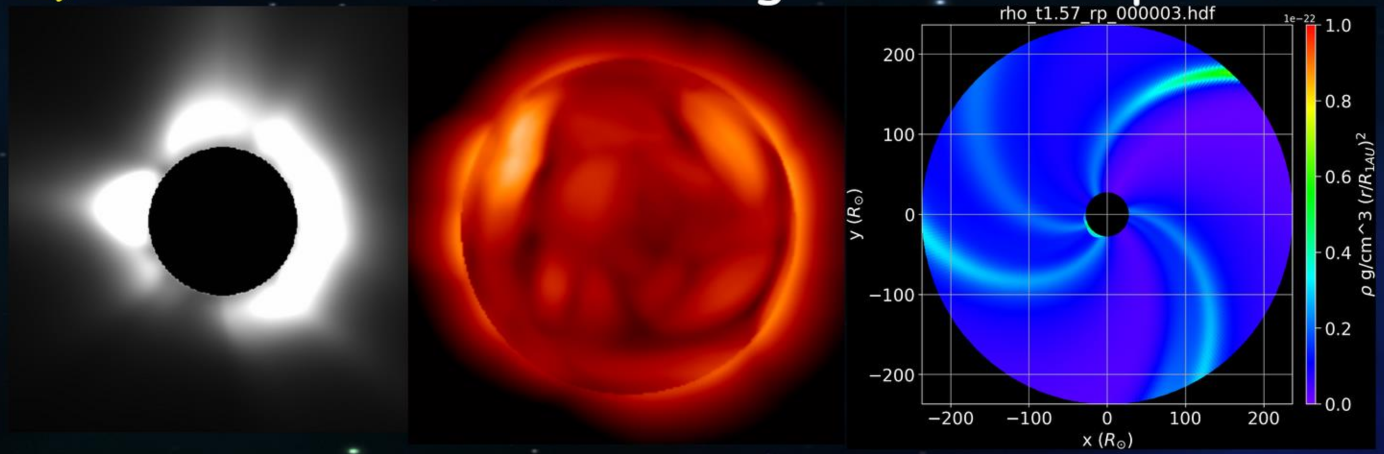
Corhel-Amcg's Recipe for Making Solar Storms

1) Get the Sun's surface magnetic field from satellite observations:

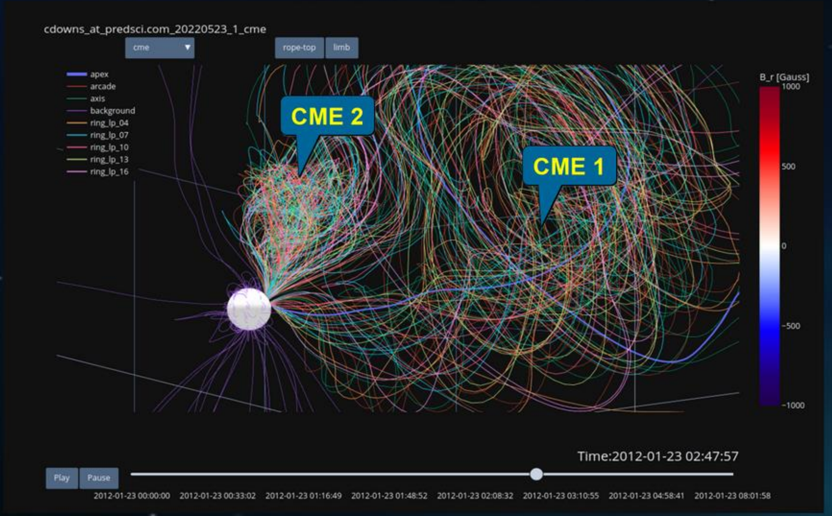
2) Design twisted magnetic rope(s) to erupt:



3) Simulate the Sun's background atmosphere:

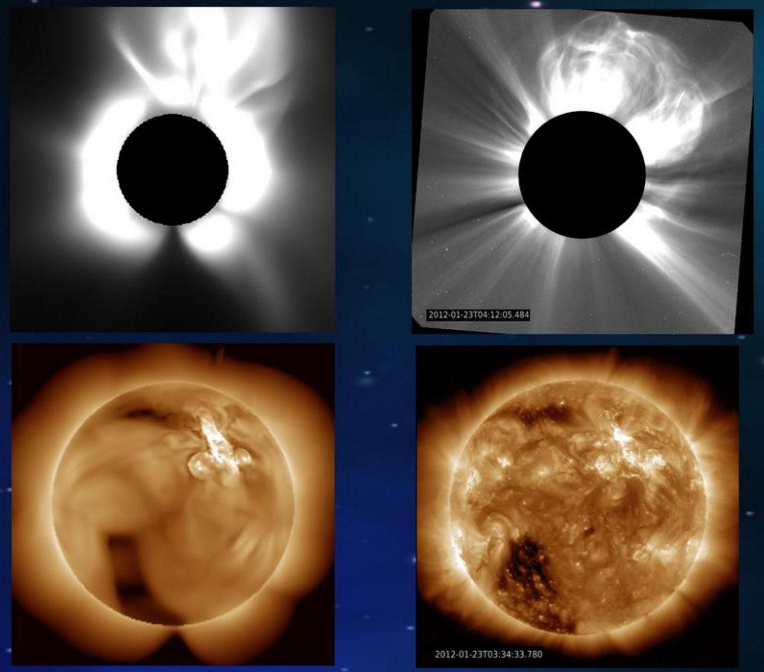


4) Insert the rope(s) and run a simulation to make them erupt and travel to Earth!



Simulation

Actual Sun




SOLAR STORM MAGNETIC STRUCTURE DESIGN
Time = 0.0000
n = 1
BACKGROUND SOLAR ATMOSPHERE
SOLAR STORM ERUPTION AND PROPAGATION
SGI ICE X
sgi
CORHEL-AMCG
Predictive Science Inc.
NASA
SC22
Dallas, TX
hpc accelerates.
Come visit the NASA booth on the exhibit floor for more details!

Opportunities

- Interested?
  - Collaborations
  - Internships
  - Research projects / advising

Predictive Science Inc.