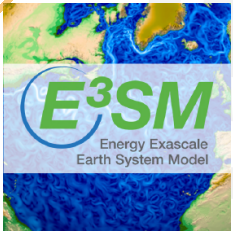
Sandia National Laboratories

Exceptional service in the national interest

Writing a complex atmosphere model, for a complex group of people, to run on complex architectures



L.Bertagna[1], A.S.Donahue[2]

[1]Sandia National Laboratories, Albuquerque, NM,

[2]Lawrence Livermore National Laboratories, Livermore, CA

July 30th, 2025

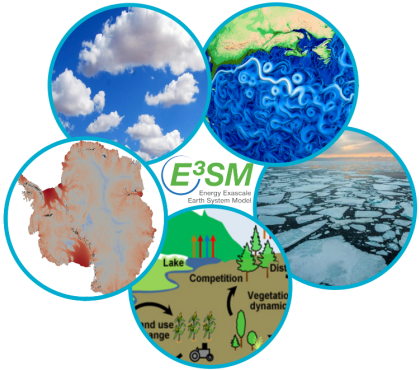
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND NO. SAND2025-09462PE

U.S. DEPARTMENT OF ENERGY

NNSA

Energy Exascale Earth System Model (E3SM)

- US Department of Energy's state-of-the-art Earth system model, forked from the Community Earth System Model (CESM) in 2014.
- Several components: atmosphere, land, ocean, land-ice, sea-ice, etc.
- Broad variety of time/space scales.
- Mostly written in Fortran.
- Developed by hundreds of people, contains snippets of codes written across several decades.

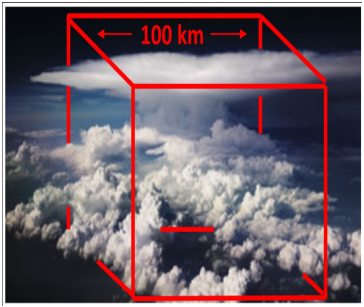


E3SM Atmosphere Model (EAM)

- Broadly speaking, divided in two parts: dynamics and physics
- Dynamics: solves Navier-Stokes equations in rotating spherical framework. It also solves for the transport of tracers in the atmosphere. E3SM uses High Order Methods Modeling Environment (HOMME, M.Taylor) package, which
  - decouples horizontal and vertical differential operators;
  - uses Spectral Element method in horizontal direction;
  - uses Finite Difference methods in vertical direction;
- Physics: approximates processes not resolved by dynamics. Examples include:
  - microphysics: water (vapor, liquid, ice) phase changes and precipitation;
  - macrophysics: subgrid cloud and turbulent processes;
  - radiation: radiative effects on atm temperature;
  - aerosols: cloud and radiative effects of transported particles.
  - deep convection: thermally driven turbulent mixing of air
- All MPI communication (except for I/O) is in dynamics. Physics is usually implemented as a "column model", making it embarrassingly parallel in the horiz direction

Global cloud resolving models. Why?

- Global Cloud-Resolving Models (GCRM) avoid the need for convection parameterizations, which are the main source of climate change uncertainty (Sherwood et al., Nature 2014)
- Resolved convection will substantially reduce major systematic errors in precipitation because of its more realistic and explicit treatment of convective storms.
- Improve our ability to assess regional impacts of climate change on the water cycle that directly affect multiple sectors of the US and global economies, especially agriculture and energy production.



Simple Cloud-Resolving E3SM Atmosphere Model

The Simple Cloud-Resolving E3SM Atmosphere Model (SCREAM) started in 2018, tasked with delivering a cloud-resolving E3SM atmosphere model that runs efficiently on DOE's exascale machines.

The team:

- scattered across 6 US DOE National Laboratories (plus a number of university collaborators)
- backgrounds spanning atm sciences, CFD, computer science, hardware, compilers, ...

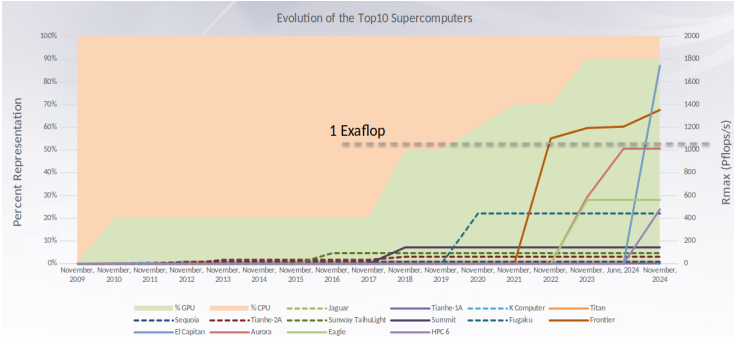
SCREAM configuration

- resolution: 3.25km horizontal, 128 vertical levels
- dynamics: HOMME non-hydrostatic dycore (Taylor, JAMES, 2020)
- microphysics: Predicted Particle Properties (Morrison, Milbrandt, J.Atm.Sci. 2015)
- macrophysics: Simple High Order Closure (Bogenschutz, Kruger, JAMES 2015)
- radiation: RTE+RRTMGP package (Pincus et al, JAMES 2019)
- aerosol: prescribed
- ~~**deep convection**~~

Note: at 3.25km horizontal resolution, with 128 vertical levels, we have ∼7.2B degrees-of-freedom *per variable* on the dynamics grid.

Programming in the era of GPUs

- The Top500 list shows the largest machines are increasingly GPU-based
- GPU architectures are not all the same
- Vendors may change strategy and/or abandon certain designs (remember KNL?)
- CPUs are NOT dead, and still play an essential role
- Who knows what computing devices will look like in 10y?
- Adapting to code for GPUs takes time and training



Performance portability strategies

While GPUs are king in the exascale era, CPU performance is still crucial in E3SM.

Performance Portability: capability of a code base to run "efficiently" on a variety of computer architectures. Three main approaches:

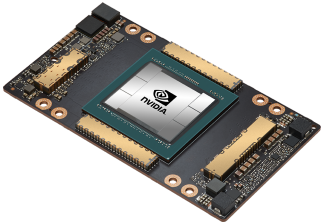
- **compiler directives**: hint/force compiler on how to optimize (e.g., OpenMP, OpenACC).
- **general purpose lib**: delegate architecture-specific choices to a library (e.g., Kokkos, Raja, etc.)
- **domain-specific lang/lib**: add intermediate compilation step, to generate optimal source for a given architecture (e.g., psyclone, gridtools, etc.).

Note: maintaining multiple versions of E3SM (one for each HPC architecture) is not viable approach. Performance portability **is a must** in the path to exascale.

The Kokkos programming model

- C++ library for on-node parallelism
- Provides constructs for expressing parallelism: execution space, execution policy, parallel operation.
- Provides constructs for multi-dimensional arrays: data type, memory space, layout, memory access/handling.
- Supports several back ends: OpenMP, Pthreads, Cuda, HIP, SYCL, etc.
- Very Reliable: large pool of (world-class) developers, heavily tested, countless downstream apps, closely follows new HPC architectures.

www.kokkos.org



Challenges of Fortran->C++ rewriting for portability

Rewriting a whole atmosphere model in C++ to run efficiently on GPUs and CPUs was not easy.

Some of the challenges:

- C++ is a syntax-heavy language. Sometimes hard to read for C++ experts too.
- GPU and CPU work in drastically different ways.
- Domain scientists are very familiar with Fortran (maybe Python), but not C++ ("SFINAE you said? What's that?")
- Computer scientists are not familiar with atm sciences ("a parametrization you said? What's that?")
- The existing code is so vast and spans so many years that
  - nobody has a good understanding of all of it (person 1: "ask person 2", person2: "ask person 3", person 3: retired long ago)
  - some snippets are still there years after last time they were used
  - some parts coded when Fortran had line length limits, and/or by people used to that, resulting in cryptic var names (pblhp, jt2slv, nvcfin_f,...)

Challenges: the language barrier

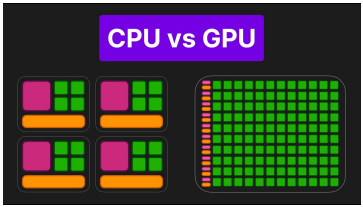
- SCREAM team paired folks new to C++ (or relatively inexperienced) with seasoned developers.
- Ramp up complexity of tasks over time (don't start from code involving template specialization).
- Provide design patterns, to be used over and over (even if sometimes you *may* leave some perf on the table)
- While not scalable, at the beginning it's ok to split work between devs and domain scientists based on background
- WARNING: do not degenerate in the pattern "scientists will develop/tune the parametrization/scheme in f90/python, and developers will take care of porting it"; nobody learns, expertise is not shared, everyone gets frustrated.

Challenges: the domain barrier

- Developers may not understand the science. Or the numerical schemes. Looking at the solution, they may only see numbers.
- Atm scientists don't appreciate (enough) the value of unit tests. Debugging with only a full E3SM bfb test is a nightmare.
- Developers do not know enough about the science to come up with *property* tests (e.g., field A should be monotonic)
- Patterns and abstractions can help to separate concerns and *compartmentalize*.
- Some cross-education is needed: developers need to familiarize with "typical" solutions and eye-ball metrics statistics, while atm scientists need to understand the rules for a robust code (e.g., testing)

Challenges: GPUs vs CPUs

- Kokkos (or other perf-portability libraries) can help to make code "uniform" (single-code, multiple-backends)
- Must train devs on fundamental differences, like memory spaces management, or race conditions, synchronizations, etc.
- The trick to get performance on GPU may be drastically different from CPU. In order to achieve a single-code, one needs to think carefully and plan on design choices (data structures, algorithms, loops order, etc).
- Participate to hackathons: help from vendors and cluster admins can drastically help, especially with familiarizing with toolchains (debuggers and profilers above all)
- Automating GPU testing (via CI) can help with developers that don't put too much attention to GPU during development (some devs may not get access to GPU machine right away)



Challenges: learn from existing code

- Writing an atm model from scratch is an opportunity to change what is fragile, unclear, or obsolete.
- Existing codes often have a reason for the way they are. Most of the time it is a good one. Sometimes it is habit.
- E3SM build system is a soup of XML, python, perl, bash, with a sprinkle of CMake. SCREAM was entirely CMake-based from the start, but had to interact with main E3SM build system.
- Existing EAM contained a lot of "spaghetti-code". Hard to follow the code unless you already know it.
- For some parts of the existing code people only knew "how to use it", but did not know "how it worked" or "why it's like that". Hard to distinguish between what needs to be "copied" and what can be redesigned if you don't understand the reasons behind it. Sometimes, the person that knows the answer (if any) is not right next door.

Software Engineering: hierarchical parallelism

- Parallelizing nested loops allow to expose maximum parallelism (fundamental for GPUs)
- In Kokkos, this is implemented via "team" execution policies, grouping threads in "teams", so that they can share local scratch memory, and cooperatively work to compute and reuse intermediate quantities.
- In SCREAM, we only use 2 layers of parallelism, over columns and levels respectively. The exception is dynamics, where 3 layers are used.
- By slicing-away outer dimensions, SCREAM can use the same array layout (`Kokkos::LayoutRight`, or "row major")
- Team policies allow for cached memory access on CPU, and coalesced memory access on GPU.
- Repetitive patterns help domain scientists (and also developers)

Software Engineering: hierarchical parallelism

```
for (int ie=0; ie<num_elements; ++ie) {
  for (int idx=0; idx<NP*NP; ++idx) {
    int i = idx / NP; int j = idx % NP;
    double v0 = v(ie,0,i,j); double v1 = v(ie,1,i,j);
    buf(0,i,j) = (J(0,0,i,j)*v0 + J(1,0,i,j)*v1)*metdet(i,j);
    buf(1,i,j) = (J(0,1,i,j)*v0 + J(1,1,i,j)*v1)*metdet(i,j);
  }

  for (int idx=0; idx<NP*NP; ++idx) {
    int i = idx / NP; int j = idx % NP;
    double dudx = 0.0, dvdy = 0.0;
    for (int k = 0; k < NP; ++k) {
      dudx += D(j,k) * buf(0,i,k);
      dvdy += D(i,k) * buf(1,k,j);
    }
    div(ie,i,j) = (dudx+dvdy) / (metdet(i,j)*rearth);
  }
  ...
}
```

Software Engineering: hierarchical parallelism

```
for (int ie=0; ie<num_elements; ++ie) {
  for (int idx=0; idx<NP*NP; ++idx) {
    int i = idx / NP; int j = idx % NP;
    double v0 = v(ie,0,i,j); double v1 = v(ie,1,i,j);
    buf(0,i,j) = (J(0,0,i,j)*v0 + J(1,0,i,j)*v1)*metdet(i,j);
    buf(1,i,j) = (J(0,1,i,j)*v0 + J(1,1,i,j)*v1)*metdet(i,j);
  }

  for (int idx=0; idx<NP*NP; ++idx) {
    int i = idx / NP; int j = idx % NP;
    double dudx = 0.0, dvdy = 0.0;
    for (int k = 0; k < NP; ++k) {
      dudx += D(j,k) * buf(0,i,k);
      dvdy += D(i,k) * buf(1,k,j);
    }
    div(ie,i,j) = (dudx+dvdy) / (metdet(i,j)*rearth);
  }
  ...
}
```

← ∥ over # teams

Software Engineering: hierarchical parallelism

```
for (int ie=0; ie<num_elements; ++ie) {          || over # teams
  for (int idx=0; idx<NP*NP; ++idx) {            || over # threads in a team
    int i = idx / NP; int j = idx % NP;
    double v0 = v(ie,0,i,j); double v1 = v(ie,1,i,j);
    buf(0,i,j) = (J(0,0,i,j)*v0 + J(1,0,i,j)*v1)*metdet(i,j);
    buf(1,i,j) = (J(0,1,i,j)*v0 + J(1,1,i,j)*v1)*metdet(i,j);
  }

  for (int idx=0; idx<NP*NP; ++idx) {            || over # threads in a team
    int i = idx / NP; int j = idx % NP;
    double dudx = 0.0, dvdy = 0.0;
    for (int k = 0; k < NP; ++k) {
      dudx += D(j,k) * buf(0,i,k);
      dvdy += D(i,k) * buf(1,k,j);
    }
    div(ie,i,j) = (dudx+dvdy) / (metdet(i,j)*rearth);
  }
  ...
}
```

Software Engineering: hierarchical parallelism

```
for (int ie=0; ie<num_elements; ++ie) {
  for (int idx=0; idx<NP*NP; ++idx) {
    int i = idx / NP; int j = idx % NP;
    double v0 = v(ie,0,i,j); double v1 = v(ie,1,i,j);
    buf(0,i,j) = (J(0,0,i,j)*v0 + J(1,0,i,j)*v1)*metdet(i,j);
    buf(1,i,j) = (J(0,1,i,j)*v0 + J(1,1,i,j)*v1)*metdet(i,j);
  }
  team barrier
  for (int idx=0; idx<NP*NP; ++idx) {
    int i = idx / NP; int j = idx % NP;
    double dudx = 0.0, dvdy = 0.0;
    for (int k = 0; k < NP; ++k) {
      dudx += D(j,k) * buf(0,i,k);
      dvdy += D(i,k) * buf(1,k,j);
    }
    div(ie,i,j) = (dudx+dvdy) / (metdet(i,j)*rearth);
  }
  ...
}
```

over # teams

over # threads in a team

over # threads in a team

shared within team

Software Engineering: simd-like types for vectorization

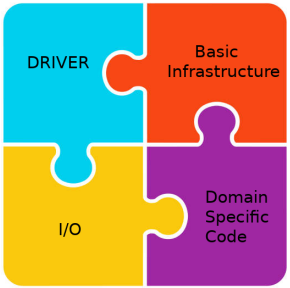
- CPU performance is still crucial for lower-resolution, debugging, as well as quickly experimenting new features.
- Vectorization is by far the most effective way to obtain performance in terms of FLOPS/Watt.
- Fortran is built for N-dim arrays manipulation, but C++ isn't. C++ compilers do not always vectorize efficiently.
- Several projects reached the same solution: use simd-like data structures to facilitate the compiler vectorization. In SCREAM, we called this data structure "Pack".
- Some parametrizations are full of conditional statements, so "masked" simd operations are necessary.
- On GPU, no vectorization. If code hard-codes Pack as scalar type, must use a pack size of 1 (with no overhead).
- Overload math operators (and functions) to allow for seamless code

Software Engineering: simd-like types for vectorization

```
constexpr int pack_size = 8;
constexpr double pi = 3.1415;
using pack_t = Pack<double,pack_size>;

// Get a view from somewhere
Kokkos::View<double*> v = ...;
v(0)=pi; v(1)=2*pi; ..., v(n) = n*pi; // Init with range values

// Reinterpret the view (must ensure v.size() is a multiple of pack_size)
auto npacks = (v.size() + pack_size - 1) / pack_size;
Kokkos::View<pack_t*> vp(reinterpret_cast<pack_t*>(v.data()),npacks);

// Each entry is now a pack, and we can do math on it
auto& p = vp(0);

// Option 1: assumes pack type
auto mask = p>=(5*pi);
p.set(mask,sin(p)); // Will yield p = {pi,2pi,3pi,4pi,0,0,0,0}

// Option 2: works with Pack and double (via template specializations)
auto where_small = where(p>=(5*pi),p); // Gets a handle of p, along with mask
where_small = sin(p); // Only updates where mask is true
```

Software Engineering: separation of concerns

- Force interfaces to be explicit on what the require/compute, and track it.
- Avoid global structures: data is provided at specific times/places. Cannot circumvent access points.
- When possible, use abstract interfaces.
- When possible, avoid assumptions on specific types or operations order.
- Prefer encapsulating 3 small classes over 1 big class (enhances testability)

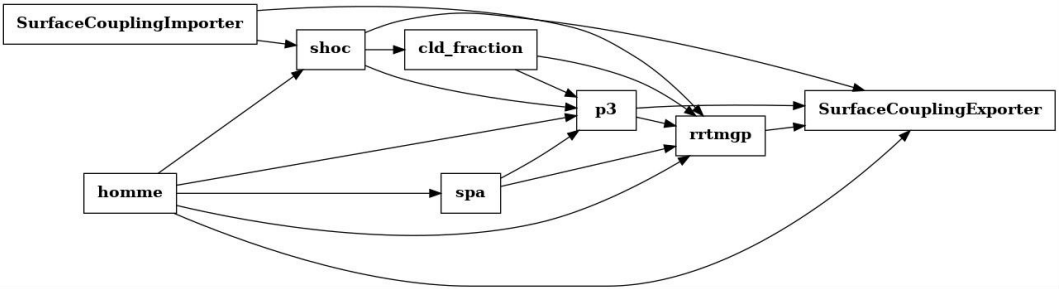


Software Engineering: separation of concerns

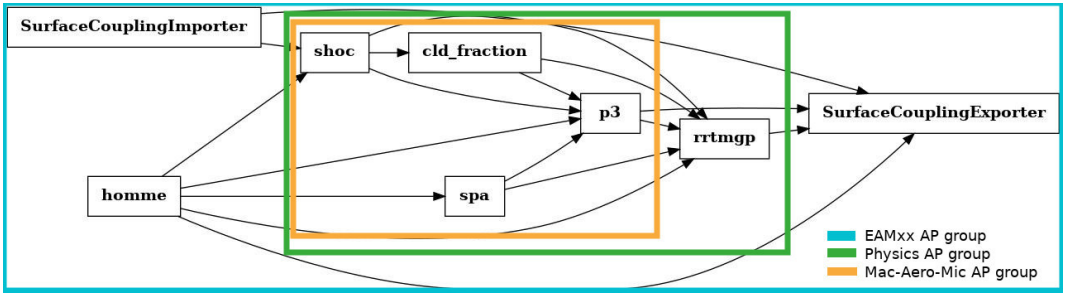
Example: `AtmosphereProcess`.

- Provides the glue between the atm driver infrastructure and the parametrization implementation library.
- From the driver point of view, it's a box that takes input fields and computes output fields.
- Fields used must be declared as input or output (or both), so that driver can provide a handle with proper cv qualifier.
- Many steps of setup/run phases are common, so they can be implemented in base class, reducing implementation burden for the parametrization developers.
- Can allow nesting and grouping: a group of 2 atm procs `A` and `B` can be "viewed" as a single process. E.g., this allows to subcycle them as `A,B,A,B`, handling everything from the base class (again, lowering implementation burden on developers).
- For basic parametrizations, the concrete implementation of the corresponding atm process can take as little as 200 lines of code. Namely,
  - at init: declare input/output fields (name, layout, units, ...)
  - at run: grab views from input/output fields, call implementation details, ensure output fields are synced to device

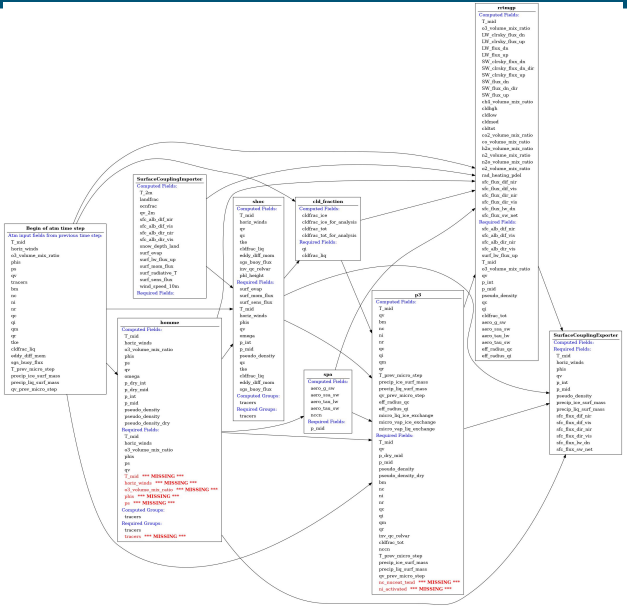
Software Engineering: separation of concerns



Software Engineering: separation of concerns



Software Engineering: separation of concerns



Conclusions

- GPUs are here to stay, so codes that need to use supercomputer need to adapt
- Expertise needed in both Computer and Domain sciences is such that very few can master both
- Splitting work into "science/algorithms development" vs "language-conversion/portability development" is not maintainable in the long run. Compromise is needed: devs need to know *some* science, and scientists need to know *some* coding
- Code encapsulation can help separate concerns, increase readability in science-relevant parts, ease of debugging, code maintainability, and overall productivity.
- Develop data structures and code patterns to take care of performance-critical sections. Examples include hierarchical parallelism patterns, vectorization, arrays manipulations (scans, reductions).
- Rewriting a model from scratch is an opportunity: spend time understanding *in depth* the reasons behind code/algorithm choices; keep what has a strong motivation behind, but don't be afraid to change approach if an advantageous alternative is available.