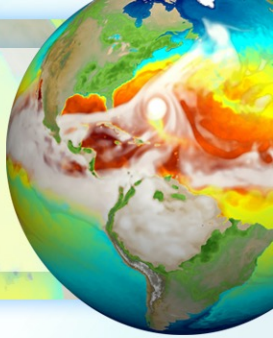
E3SM's C++ based GPU strategy and latest performance



Mark Taylor
Sandia National Laboratories

Andrew Bradley (SNL), Oksana Guba (SNL), Matt Norman (ORNL), Xingqiu Yuan (ANL)

E3SM Nonhydrostatic Atmosphere NGD (Peter Caldwell)
E3SM-MMF Exascale Computing Project. (Mark Taylor)
E3SM Performance Group. (Phil Jones, Sarat Sreepathi)

7th ENES HPC Workshop, May 10, 2022

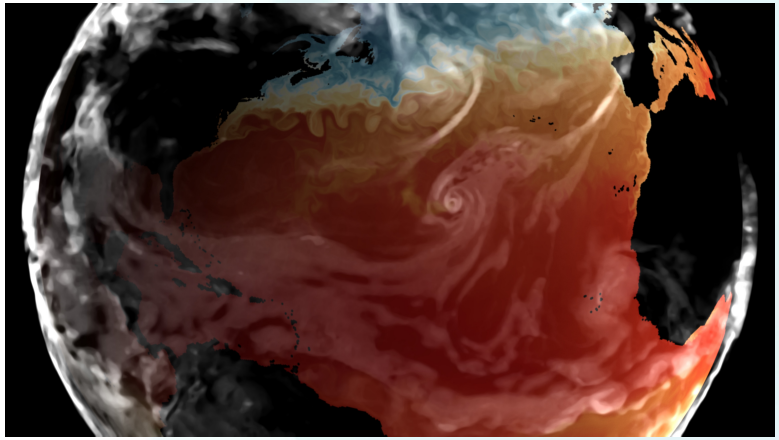
EXASCALE COMPUTING PROJECT

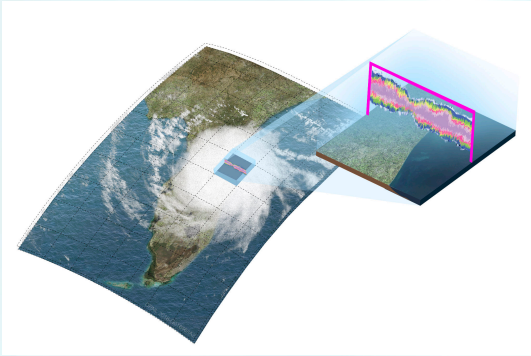
E3SM Energy Exascale Earth System Model

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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. SAND2021-11206 O

U.S. DEPARTMENT OF ENERGY

Energy Exascale Earth System Model (E3SM)



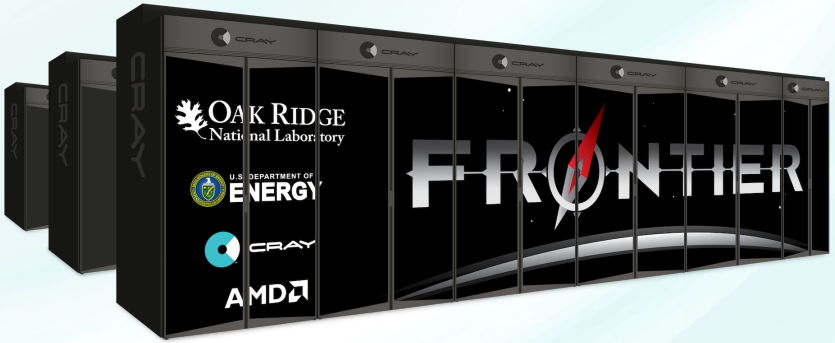


- **US DOE: E3SM Project**
- Virtual modeling center, spread over 8 laboratories
- ~70 FTEs
- DOE driven science mission: Energy & water issues looking out 40 years
- Ensure E3SM will run well on upcoming DOE exascale computers

- **US DOE Exascale Computing Project**
- Invests in ensuring the exascale readiness of multiple applications, including climate modeling
- Developing E3SM-MMF, a superparameterized version of E3SM

Upcoming US DOE machines are GPU based

- 2021: NERSC Perlmutter
  - 6000 GPUs. 8 MW
  - 3000 CPU nodes (2022)
- 2022: OLCF Frontier
  - 1 CPU + 4 AMD GPUs per node.
  - 30 MW
- 2023: ALCF Aurora
  - Intel GPU nodes.
  - ~40MW



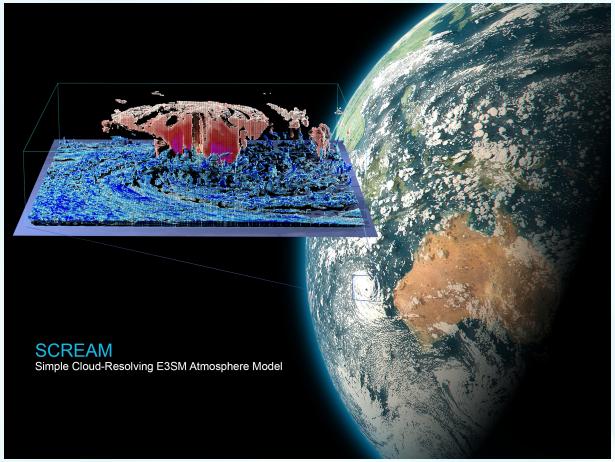
Cloud resolving modeling on GPUs

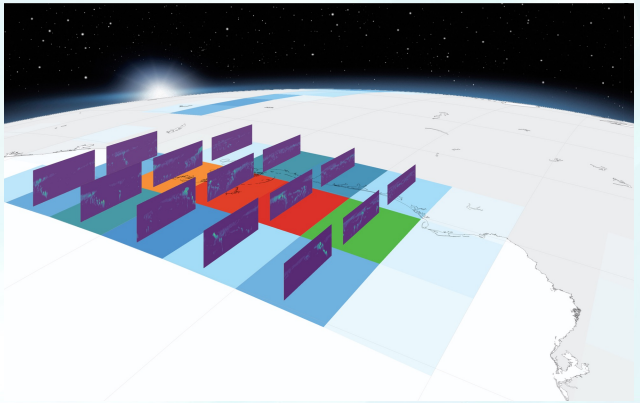
SCREAM:

- Simple Cloud-Resolving E3SM Atmosphere Model
- Entire atmosphere component written in **C++/Kokkos**
- E3SM to transition to SCREAM code base after the 2023 release of E3SM v3

E3SM-MMF

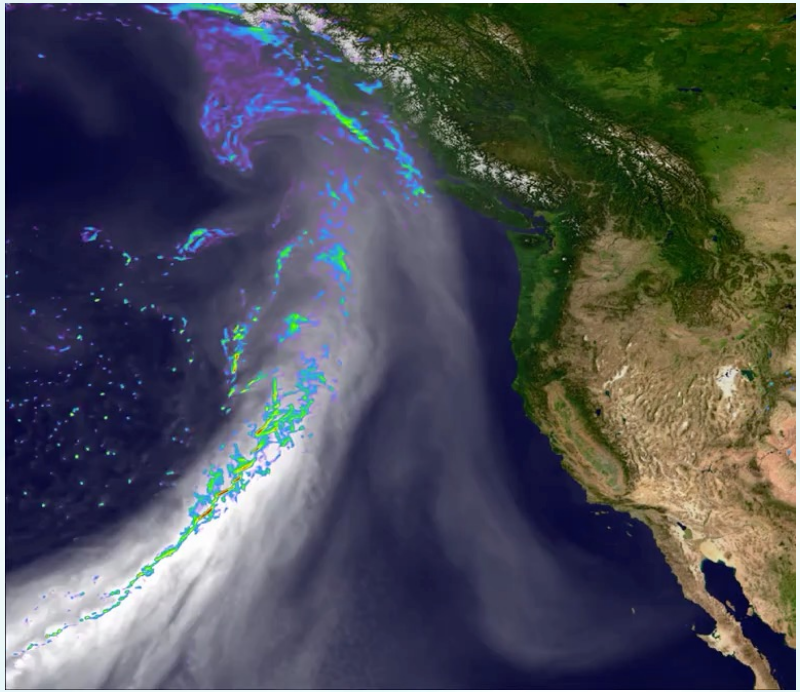
- Superparameterized version of E3SM
- Ideal for GPU acceleration: Run superparameterization and radiation on the GPU, leave the rest of the Earth system model on the CPU
- **C++/YAKL**
- Some aspects of a cloud resolving model while obtaining climate model throughput rates with far less resources





SCREAM Status

- V0: Fortran implementation
  - NH spectral element dycore: Taylor et al., JAMES 2020
  - Physics: SHOC, P3, RRTMG, prescribed aerosols
  - DYAMOND results: Caldwell et al., JAMES 2021
- V1: Rewrite in C++/Kokkos:
  - NH Dycore port completed in 2020
  - Physics port completed in 2021
  - RRTMGP++ (YAKL)
  - Driver to be completed in mid 2022
  - Bertagna et al., GMD 2019, SC 2020
- Remaining slides: Dycore benchmarks



Movie: Precipitation (colors) and integrated water vapor (gray) for an atmospheric river from E3SM's DYAMOND2 simulation. By Paul Ullrich/UC Davis

SCREAM-C++: Basics About Kokkos

- Kokkos is a C++ library which provides an **abstraction layer** around code related to on-node parallelism (like loops and arrays)
  - This allows a single code to run efficiently on CPUs, GPUs, and whatever comes next
- Implementations are provided for Cuda, OpenMP, Serial, Pthreads, etc
  - Kokkos is developed by a large multi-disciplinary team with early access to new machines and compilers
  - Kokkos emphasizes getting its features into the C++ standard

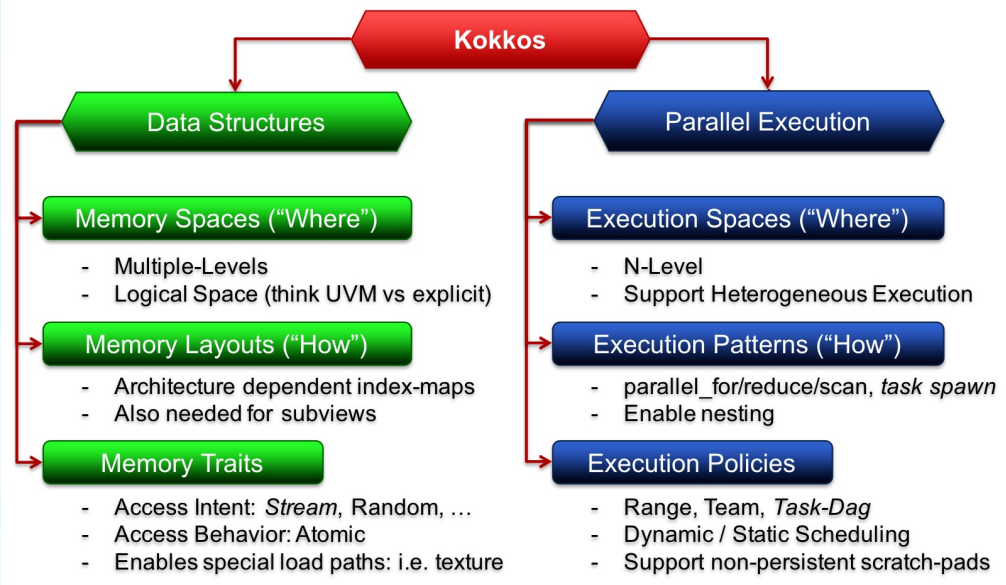
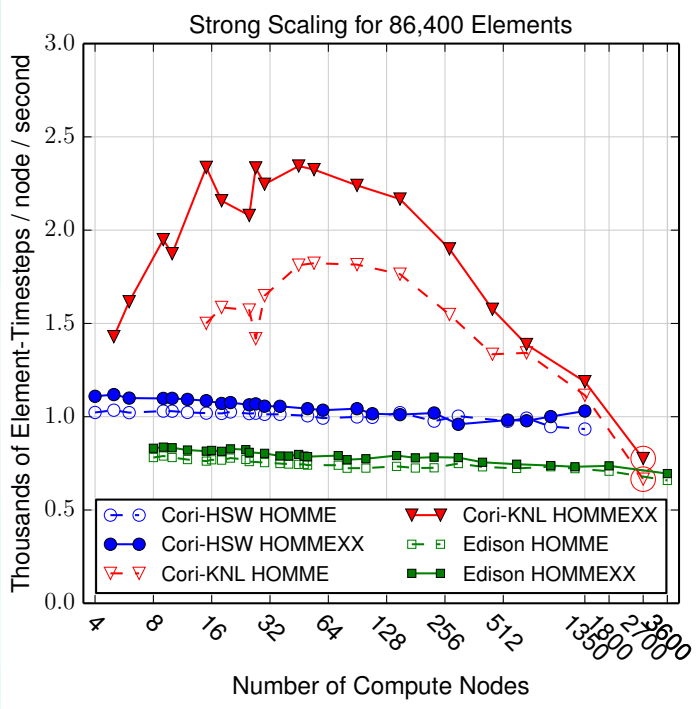


Fig: Diagram of Kokkos' abstraction hierarchies. From https://github.com/kokkos/kokkos-tutorials/blob/master/KokkosCapabilities.pdf

DYCORE Performance on CPU systems FORTRAN vs C++

- Key to good CPU performance is vectorization
- Fortran has excellent auto-vectorization. Competitive performance in C++ requires explicit vectorization via “packs” (Bertagna et al., GMD 2019)
- Vectorized C++ code matches Fortran code on traditional CPUs, faster on Intel (since abandoned) KNL architecture
- Vectorization not necessary on current GPUs, but this may change in the future



The Downside of C++:

- C++ and Kokkos make the code more complicated (see example)
- Code complexity may drive E3SM towards a model where only computer scientists can change code – but we hope not!
- Opportunity to remove decades of outdated code is a major benefit of switching languages
- Most of the complexity is due to "packs" needed to get C++ code to vectorize when running on CPUs

Original F90

```
kloop_sedi_c2: do k = k_qxtop,k_qxbot,-kdir

   qc_notsmall_c2: if (qc_incld(k)>qsmall) then
      !-- compute Vq, Vn
      call get_cloud_dsd2(qc_incld(k),nc_incld(k),mu_c(k),rho(k),nu,dnu,   &
           lamc(k),tmp1,tmp2,lcldm(k))

      nc(k) = nc_incld(k)*lcldm(k)
      dum = 1._rtype / bfb_pow(lamc(k), bcn)
      V_qc(k) = acn(k)*bfb_gamma(4._rtype+bcn+mu_c(k))*dum/(bfb_gamma(mu_c(k)+4._rtype))
      V_nc(k) = acn(k)*bfb_gamma(1._rtype+bcn+mu_c(k))*dum/(bfb_gamma(mu_c(k)+1._rtype))

   endif qc_notsmall_c2
   Co_max = max(Co_max, V_qc(k)*dt_left*inv_dzq(k))

enddo kloop_sedi_c2
```

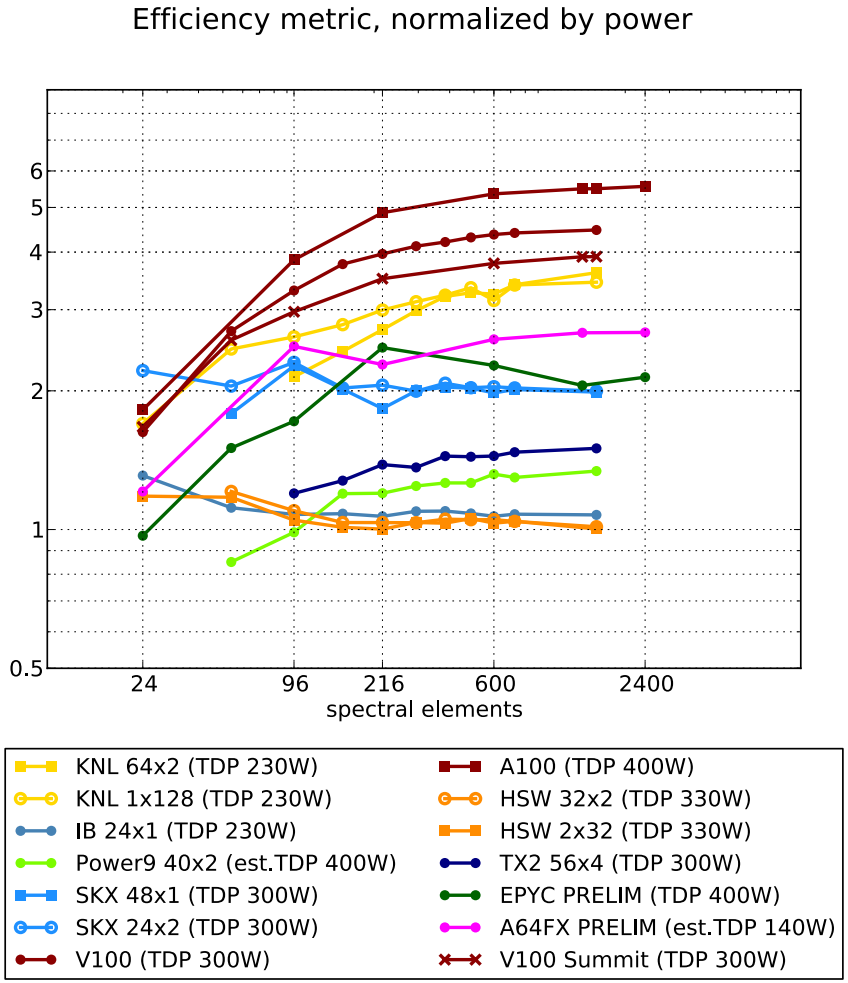
Ported to C++/Kokkos

```
Kokkos::parallel_reduce(
  Kokkos::TeamThreadRange(team, kmax-kmin+1), [&] (int pk_, Scalar& lmax) {
    const int pk = kmin + pk_;
    const auto range_pack = scream::pack::range<IntSmallPack>(pk*Spack::n);
    const auto range_mask = range_pack >= kmin_scalar && range_pack <= kmax_scalar;
    const auto qc_gt_small = range_mask && qc_incld(pk) > qsmall;
    if (qc_gt_small.any()) {
      // compute Vq, Vn
      Spack nu, cdist, cdist1, dum;
      get_cloud_dsd2<false>(qc_gt_small, qc_incld(pk), nc_incld(pk), mu_c(pk), rho(pk), nu, dnu, lamc(pk), cdist,
      nc(pk).set(qc_gt_small, nc_incld(pk)*lcldm(pk));
      dum = 1 / (pack::pow(lamc(pk), bcn));
      V_qc(pk).set(qc_gt_small, acn(pk)*pack::tgamma(4 + bcn + mu_c(pk)) * dum / (pack::tgamma(mu_c(pk)+4)));
      if (log_predictNc) {
        V_nc(pk).set(qc_gt_small, acn(pk)*pack::tgamma(1 + bcn + mu_c(pk)) * dum / (pack::tgamma(mu_c(pk)+1)));
      }

      const auto Co_max_local = max(qc_gt_small, -1,
                                    V_qc(pk) * dt_left * inv_dzq(pk));
      if (Co_max_local > lmax)
        lmax = Co_max_local;
    }
}, Kokkos::Max<Scalar>(Co_max));
team.team_barrier();
```

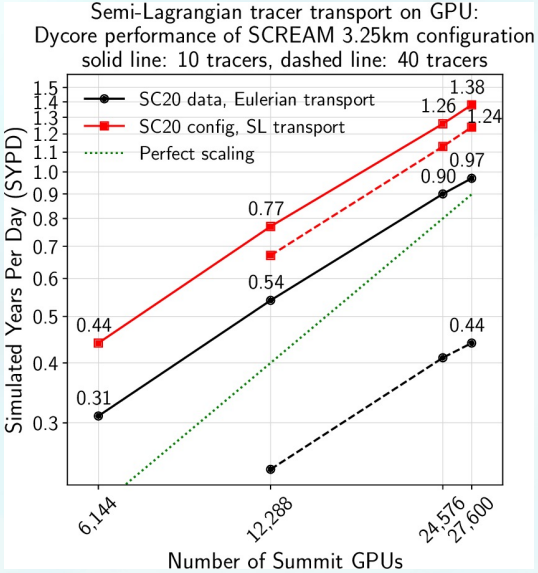
DYCORE Performance: GPUs vs CPUs

- Single-node or single GPU benchmark from Bertagna et al., GMD 2019 (updated with additional CPUs and GPUs).
- Dycore running in a climate configuration (40 tracers) without physics
- **Performance metric per watt** (grid points per second per watt, higher is better) as a function of workload per node or GPU
- GPUs provides significant performance increase, but only in the high-workload (per GPU) regime.
- Future trends for GPUs and CPUs?

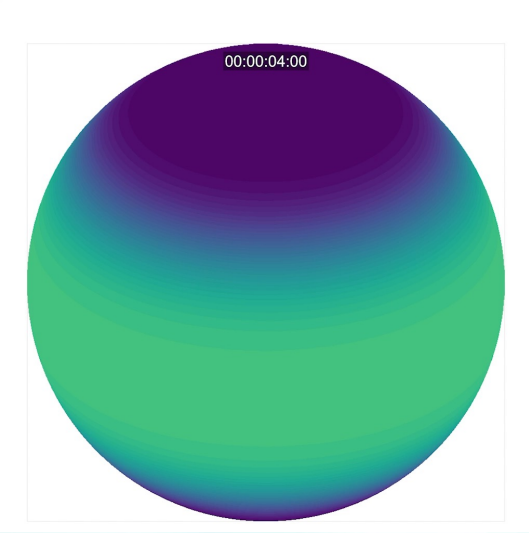


SCREAM dycore performance on Summit

- Summit: DOE's 200PF GPU system
- Using the US NWS cloud resolving GCM benchmark (3km, 128L, 10 tracers, nonhydrostatic, production configuration)
- C++/Kokkos code obtains excellent scaling, **running at 1.38 SYPD on 27K GPUs**
- Such throughput will allow for climate length simulations



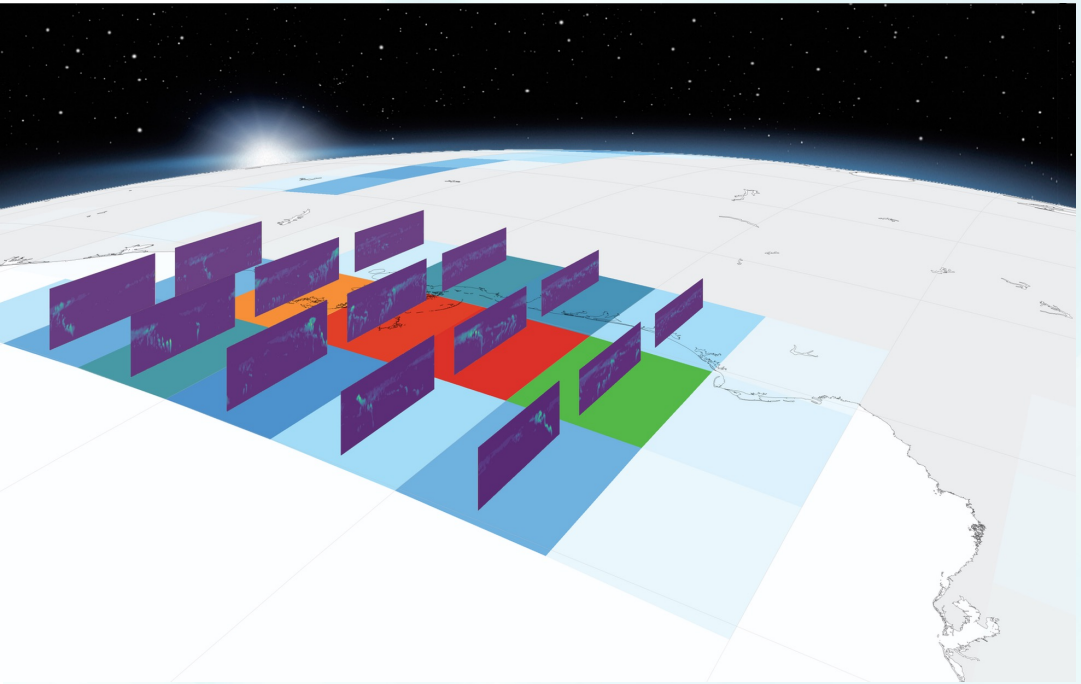
SCREAM dycore strong scaling out to 27K GPUs (NVIDIA V100). Recent adoption of SL tracer transport (solid red) improved performance to 1.38 SYPD. Dashed curves show performance with number of tracers increased from 10 to 40.



Animation of 500mb specific humidity in the idealized baroclinic instability test case used by the NWS benchmark.

Citation: L. Bertagna et. al., SC '20: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2020

The Multiscale Modeling Framework (MMF)



- E3SM-MMF approach addresses structural uncertainty in cloud processes by replacing most traditional parameterizations with cloud resolving “superparameterization” within each vertical column of the global climate model
- Superparameterization includes the cloud resolving model SAM and RRTMGP.
- Represents >90%+ of the model cost
- Both SAM and RRTMGP rewritten in C++/YAKL

Yet Another Kernel Launcher (YAKL) Portable C++ Library

- Lead by Mathew Norman (ORNL) and collaborators (1-1.5 FTE effort)
- Small portable C++ library with ≈ 5K lines of core code
- Allows Fortran-like behavior in the resulting C++ code
  - Multi-dimensional arrays with column-major ordering and arbitrary lower bounds that default to 1
  - Growing library of Fortran intrinsic functions: size(), maxval(), allocated(), etc.
- An efficient automatically enabled pool allocator for all YAKL allocations
  - Faster frequent allocations and deallocations
- Emphasis on simple, readable syntax with minimal developer concerns
- Runs on CPUs, CPU threads (OpenMP 3.5), Nvidia GPUs (CUDA), AMD GPUs (HIP), and Intel GPUs (SYCL)
- https://github.com/mrnorman/YAKL
- https://github.com/mrnorman/miniWeather

YAKL Example

Original F90 with OpenACC directive

```
function max_stable_dt(height, u, v, cfl, grav, dx, dy)  result(dt)
  real(8), dimension(:,:), intent(in) :: height, u, v
  real(8)                , intent(in) :: cfl, grav, dx, dy
  real(8)                             :: dt
  integer                             :: i, j, nx, ny
  real(8)                             :: gw, dtloc, eps
  nx = size(height,1)
  ny = size(height,2)
  dt = huge(height)       ! Initialize to a large value
  eps = epsilon(height)   ! To avoid division by zero
  !$acc parallel loop collapse(2) present(height,u,v) reduction(min:dt)
  do j = 1 , ny
    do i = 1 , nx
      gw = sqrt(grav * height(i,j))   ! Speed of gravity waves
      ! Compute local maximum stable time step
      dtloc = min( cfl * dx / ( abs(u(i,j)) + gw + eps ) , &
                   cfl * dy / ( abs(v(i,j)) + gw + eps ) )
      ! Compute global minimum of the local maximum stable time steps
      dt = min( dt , dtloc )
    enddo
  enddo
endfunction max_stable_dt
```

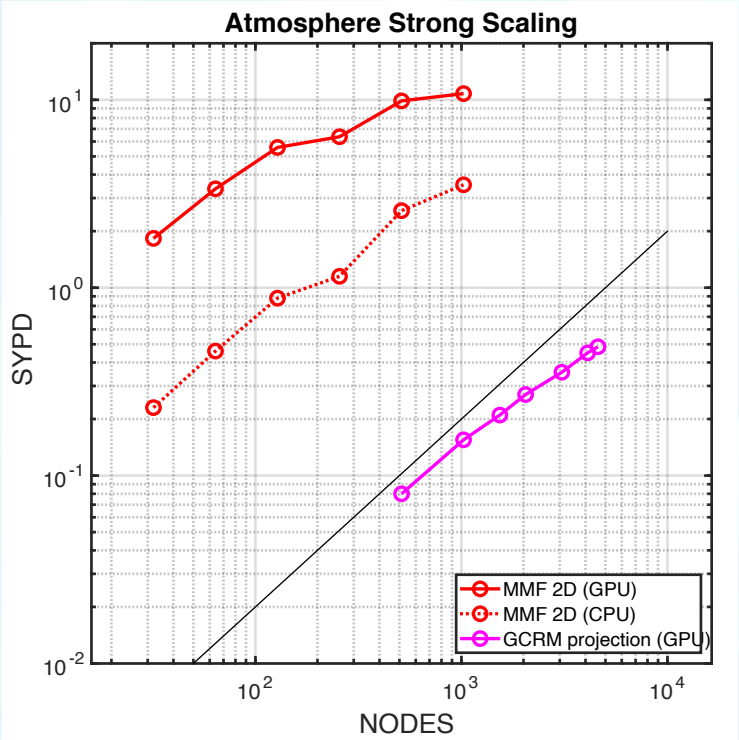
C++/YAKL version

```
// The lines before the function would be placed in a header file somewhere else
// using and typedef allow the latter code to be more readable
using yakl::fortran::Bounds;
using yakl::fortran::parallel_for;
using yakl::intrinsics::minval;
typedef double real;
typedef yakl::Array<real const,2,yakl::memDevice,yakl::sytleFortran> realConst2d;
typedef yakl::Array<real      ,2,yakl::memDevice,yakl::sytleFortran> real2d;

// Here begins the main user-level YAKL code:
real max_stable_dt(realConst2d height, realConst2d u, realConst2d v,
                   real cfl, real grav, real dx, real dy) {
  int nx = size(height,1);
  int ny = size(height,2);
  real eps = epsilon(height); // To avoid division by zero
  real2d dt2d("dt2d",nx,ny);  // Allocate array to store the local max stable time steps
  // do j = 1 , ny
  //   do i = 1 , nx
  parallel_for( "Max stable timestep" , Bounds<2>(ny,nx) , YAKL_LAMBDA (int j, int i) {
    real gw = sqrt(grav * height(i,j));  // Speed of gravity waves
    // Compute local maximum stable time step
    dt2d(i,j) = min( cfl * dx / ( abs(u(i,j)) + gw + eps ) ,
                     cfl * dy / ( abs(v(i,j)) + gw + eps ) );
  });
  // With the local max stable time steps stored, compute the minimum among all of them
  return minval( dt2d );
}
```

E3SM-MMF Atmosphere Performance: Strong Scaling

- Node to Node comparison
  - 2 IBM P9 cpus vs 6 V100 GPUs
- E3SM-MMF
  - 1km resolution superparameterization within global 0.75 degree E3SM atmosphere model
  - Significantly faster than the (estimated) performance of SCREAM, a GCRM running at 3km
  - Can obtain 10 SYPD, making it possible to run long climate simulations with some aspects of a cloud resolving model



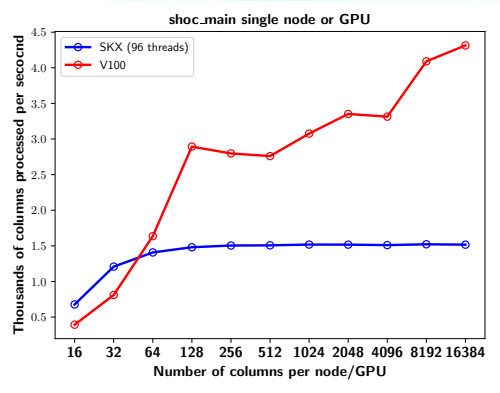
Summary:

- Kokkos:
  - Heavy use of C++ abstractions
  - Supports many advanced features: e.g. hierarchical parallelism
- YAKL:
  - Simpler approach, focused on loop level parallelism
  - Fortran-like arrays and syntax
- Both provide good GPU performance,
  - C++ compiler infrastructure far more mature than Fortran + openACC or OpenMP
  - Robustly supported across NVIDIA, AMD and Intel GPUs.
- CPU performance:
  - Vectorization orthogonal to Kokkos/YAKL but naturally supported in C++ via templates and operator overloading. Critical for best performance on newer CPUs.
  - Vectorization in Fortran is easy.
  - Vectorization in C++ requires even more detailed programming than GPU parallelization
  - We've done this work for our Kokkos codes, but not for our YAKL codes

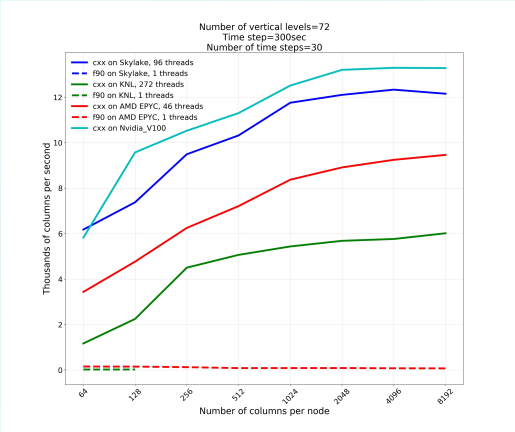
Thanks!

Extra Slides

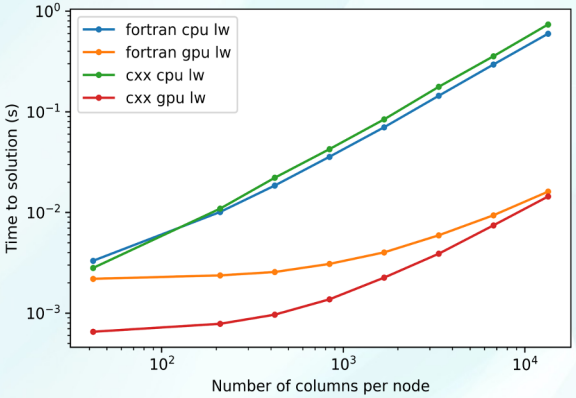
SCREAM standalone physics results



- **SHOC**

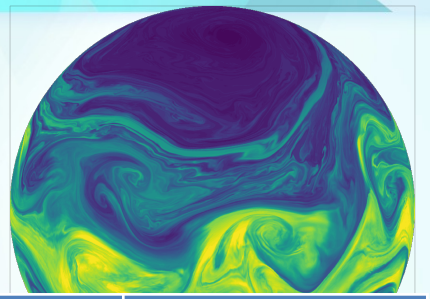


- **P3**



- **RRTMGP**

NGGPS Full System Benchmark



- NGGPS 3km (cloud resolving) benchmark problem. Benchmark from the National Weather Service
- Atmosphere dycore with realistic/operational configuration, 10 tracers and idealized physics
- Highlights from several generation of TOP500 computers and dycores.
- Dycore performance not keeping up with Linpack performance (but does track HPGC results)

Model	Computer (Linpack rating)	NGGPS 3km Benchmark
NOAA FV3 2015	Edison (2.6PF)	0.16 SYPD
HOMME (CESM) 2017	TaihuLight (125 PF)	0.34 SYPD
HOMME++ (SCREAM) 2021	Summit (200 PF)	1.38 SYPD
	Fugaku (440 PF)	?

E3SM-MMF Atmosphere Performance: Strong Scaling

- Node to Node comparison
  - 2 IBM P9 cpus vs 6 V100 GPUs
- E3SM-MMF
  - Global model running at 0.75 degree resolution
  - 1km resolution superparameterization (either 2D or 3D)
  - Both configurations faster than the (estimated) performance of SCREAM, a GCRM running at 3km
  - Can obtain 10 SYPD, making it possible to run long climate simulations with some aspects of a cloud resolving model