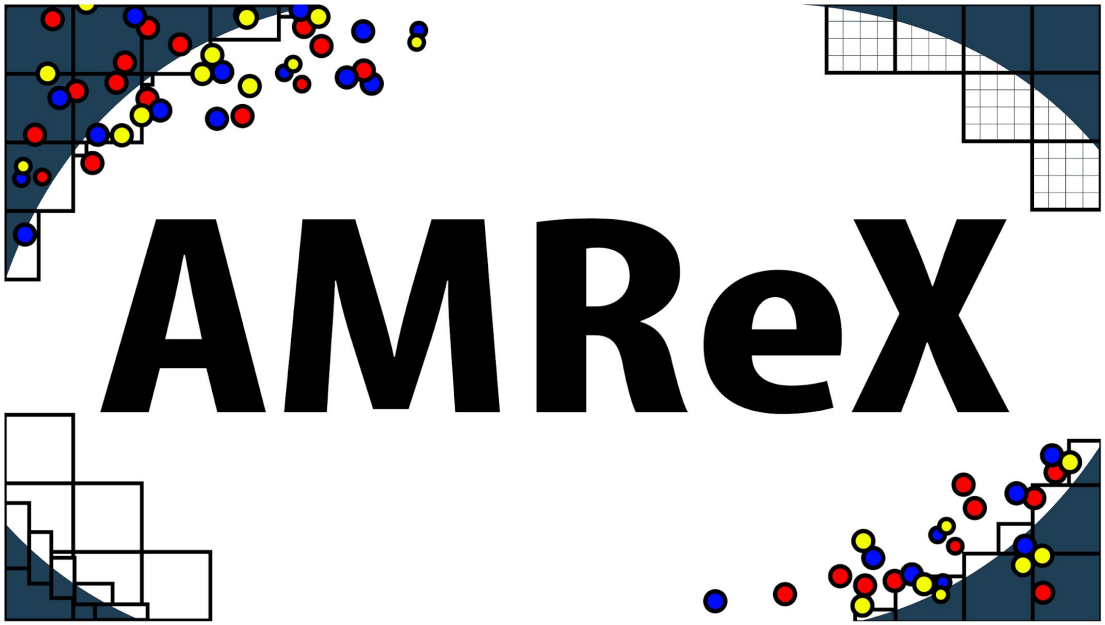


AMReX & Performance Portabiity

Weiqun Zhang, June 25, 2024

U.S. DEPARTMENT OF ENERGY | Office of Science

Applied Mathematics & Computational Research

BERKELEY LAB

Overview of AMReX

Software framework for building massively parallel, block-structured adaptive mesh refinement (AMR) applications. Coded in modern C++17 (with Fortran + Python interfaces). Runs on machines from laptops to supercomputers.

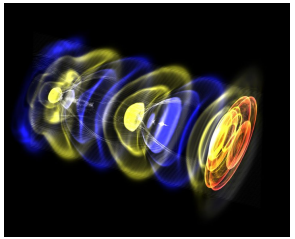
- Solution is defined on a **hierarchy of levels** of resolution, each of which is composed of a union of logically rectangular patches
- Distributed containers for **mesh** + **particle** data
- GPU-optimized **parallel communication**, **domain decomposition**, **load balancing**.
- **Multilevel** synchronization operations, tagging, regridding
- Support for **embedded boundaries** via cut cell approach
- Both native **Geometric multigrid solvers** for elliptic and parabolic systems, plus interfaces to HYPRE, PETSC…
- Performance portability layer with support for multiple GPU backends - **CUDA**, **HIP**, **SYCL** - and **OpenMP** for CPUs
- **xSDK** compliant, part of **E4S**, installable with **spack**, will be in **HPSF**

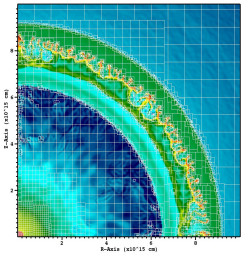
Github: https://github.com/AMReX-Codes/amrex
Docs: https://amrex-codes.github.io/amrex/

Tutorials: https://github.com/AMReX-Codes/amrex-tutorials/

A Sampling of AMReX Application Codes

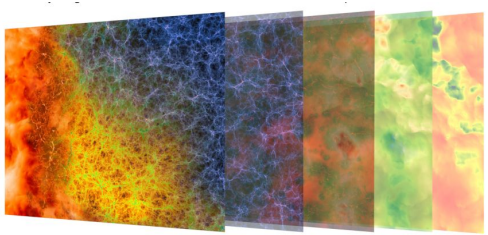
Astrophysics:
Castro (compressible)
MaestroEx (low-Mach)
Quokka (radiation-hydrodynamics)
AsterX (GRMHD)
GRTeclyn (GR)

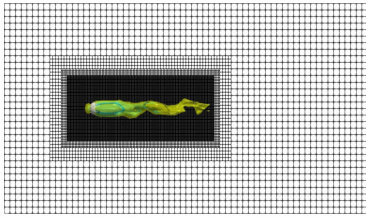




Cosmology:
Nyx

Combustion:
PeleC (Compressible)
PeleLM (Low Mach)





Accelerator Modelling:
WarpX
ImpactX
Hipace++

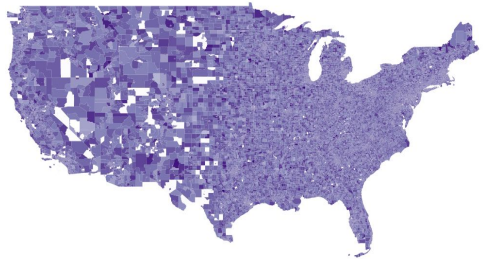
Magnetically-confined fusion:
GEMPIC

Ocean Modeling:
REMORA

Multiscale Modelling and Stochastic Systems:
FHDeX

Electromagnetics:
ARTEMIS

Epidemiology:
ExaEpi

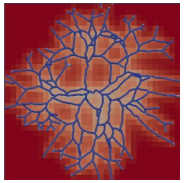


Incompressible Navier-Stokes:
IAMR
incflo

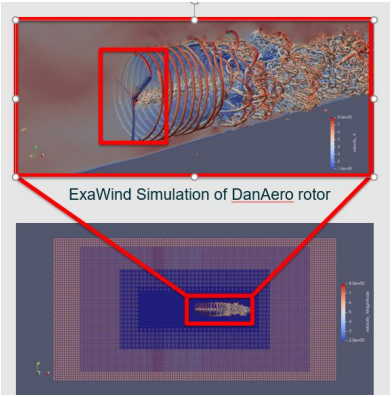
Atmospheric science:
AMR-wind
ERF

Solid Mechanics:
Alamo

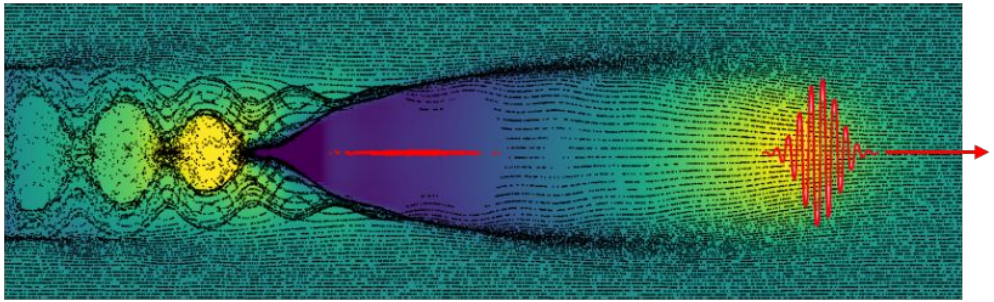
Biological cell modelling:
BoltzmanMFX
CCM

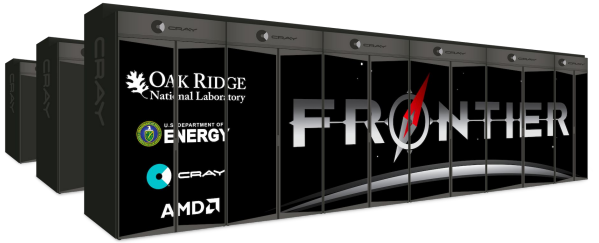


Multi-phase Flow:
MFIX-Exa



WarpX: 500x FOM, Gordon Bell Award





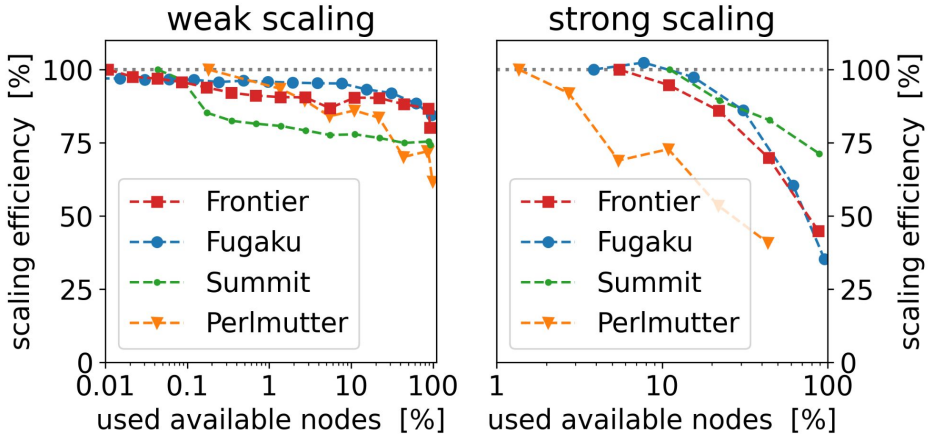


Figure-of-Merit over time

Date	Code	Machine	N_c/Node	Nodes	FOM
3/19	Warp	Cori	0.4e7	6 625	2.2e10
3/19	WarpX	Cori	0.4e7	6 625	1.0e11
6/19	WarpX	Summit	2.8e7	1 000	7.8e11
9/19	WarpX	Summit	2.3e7	2 560	6.8e11
1/20	WarpX	Summit	2.3e7	2 560	1.0e12
2/20	WarpX	Summit	2.5e7	4 263	1.2e12
6/20	WarpX	Summit	2.0e7	4 263	1.4e12
7/20	WarpX	Summit	2.0e8	4 263	2.5e12
3/21	WarpX	Summit	2.0e8	4 263	2.9e12
6/21	WarpX	Summit	2.0e8	4 263	2.7e12
7/21	WarpX	Perlmutter	2.7e8	960	1.1e12
12/21	WarpX	Summit	2.0e8	4 263	3.3e12
4/22	WarpX	Perlmutter	4.0e8	928	1.0e12
4/22	WarpX	Perlmutter†	4.0e8	928	1.4e12
4/22	WarpX	Summit	2.0e8	4 263	3.4e12
4/22	WarpX	Fugaku†	3.1e6	98 304	8.1e12
6/22	WarpX	Perlmutter	4.4e8	1 088	1.0e12
7/22	WarpX	Fugaku	3.1e6	98 304	2.2e12
7/22	WarpX	Fugaku†	3.1e6	152 064	9.3e12
7/22	WarpX	Frontier	8.1e8	8 576	1.1e13

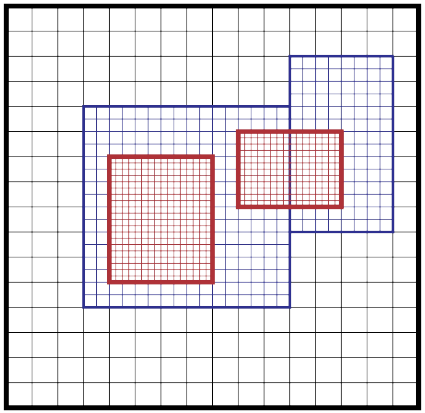
AMReX: Hello World

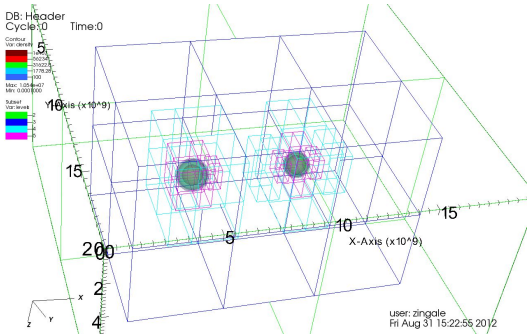
```
#include <AMReX.H>
#include <AMReX_ParallelDescriptor.H>
#include <AMReX_Print.H>
int main(int argc, char* argv[])
{
  amrex::Initialize(argc,argv);
  {
    amrex::AllPrint() << "Hello world from Proc. "
      << amrex::ParallelDescriptor::MyProc()
      << "\n";
  }
  amrex::Finalize();
}
```

```
Initializing AMReX (24.03-8-g8d447be956e3-dirty)...
MPI initialized with 4 MPI processes
MPI initialized with thread support level 0
Hello world from Proc. 3
AMReX (24.03-8-g8d447be956e3-dirty) initialized
Hello world from Proc. 0
Hello world from Proc. 1
Hello world from Proc. 2
AMReX (24.03-8-g8d447be956e3-dirty) finalized
```

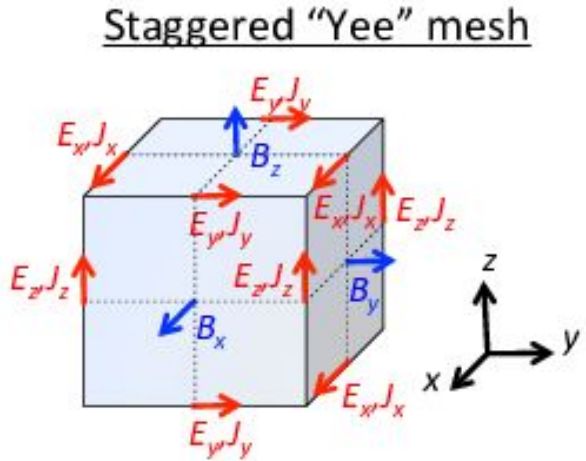
- Initialize is usually the first line. It initializes MPI, OMP, GPU, input parameters in argv, etc. It can also optionally take used provided MPI_Comm.
- Finalize is usually the last line. It frees the resources allocated by Initialize.
- Use a function or {} to create a scope to avoid objects still being alive after Finalize.
- Version string: 24.03-8-g8d447…-dirty
  - 24.03: Monthly release
  - 8: number of commits after the release
  - g8d447…: 8d447… without g is the current git hash.
  - dirty: There are uncommitted changes.

Box, IntVect, & IndexType





- Box represents a (logically) rectangular domain in global integer indexing space.
- IntVect is an integer tuple.
- IndexType: cell or node. For example, Bx on x-face: (node, cell, cell)



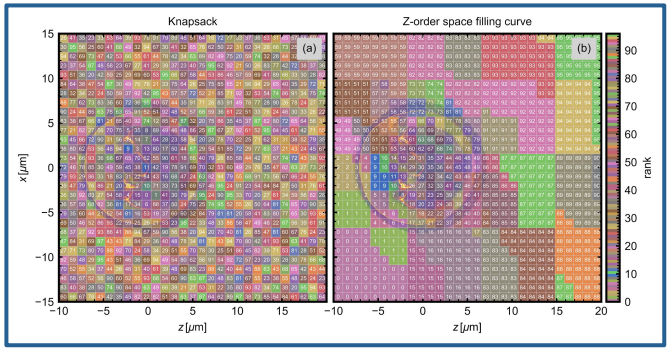
BoxArray & DistributionMapping

```
Box domain(IntVect(0), IntVect(127)); // a box with 128^3 cell
BoxArray ba(domain); // Make a new BoxArray w/ one Box
ba.maxSize(64); // Chop into Boxes of 64^3 cells
Print() << ba << "\n";
```

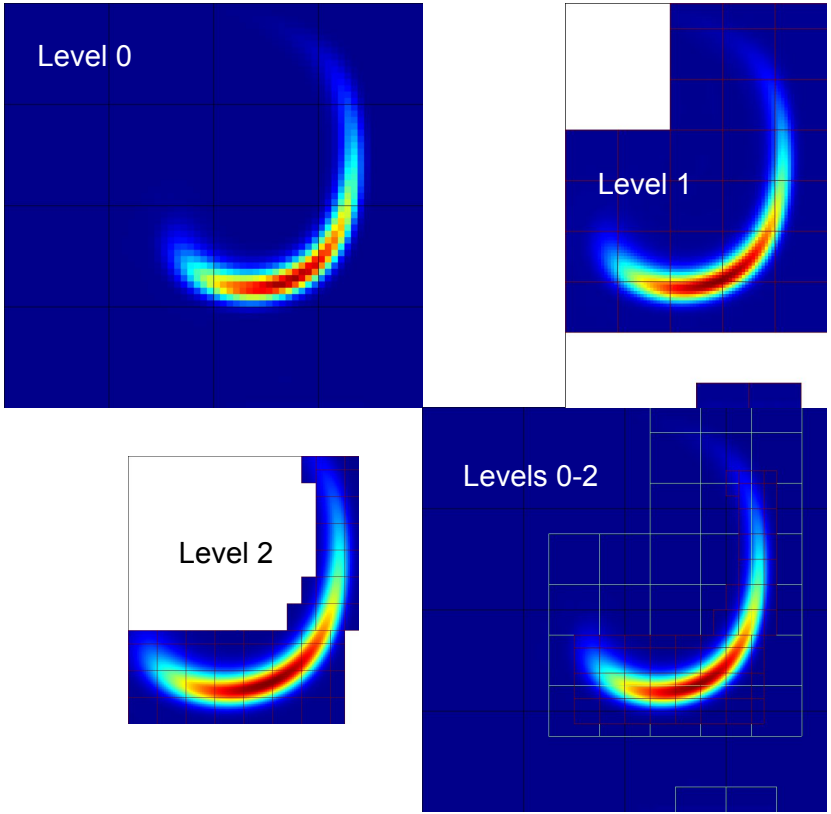
```
(BoxArray maxbox(8)
       m_ref->m_hash_sig(0)
((0,0,0) (63,63,63) (0,0,0)) ((64,0,0) (127,63,63) (0,0,0))
((0,64,0) (63,127,63) (0,0,0)) ((64,64,0) (127,127,63) (0,0,0))
((0,0,64) (63,63,127) (0,0,0)) ((64,0,64) (127,63,127) (0,0,0))
((0,64,64) (63,127,127) (0,0,0)) ((64,64,64) (127,127,127) (0,0,0))
)
```

```
DistributionMapping dm(ba);
```

- BoxArray is an array of Boxes on a single AMR level.
- DistributionMapping describes which MPI process owns the data for Boxes in a BoxArray.
- DistributionMapping strategies: Space filling curve, knapsack, round robin, etc.



MultiFab



- MultiFab is a distributed container for data on a single level.
- Multiple Fab (Fortran Array Box). Fortran multi-D array convention (i.e., column major).
- Can have multiple components. Multiple 4-D arrays.
- Can have ghost cells.
- What memory to use?
- non-copyable, but movable.

```
// ba: BoxArray
// dm: DistributionMapping
int ncomp = 3;
IntVect nghost(1,2,0);
MultiFab mf(ba, dm, ncomp, nghost);
MultiFab hmf(ba, dm, 1, 0,
   MFInfo().SetArena(The_Cpu_Arena()));
MultiFab mf2(std::move(mf));
mf2 = std::move(mf);
std::swap(hmf,mf2);
```

Examples of MultiFab Functions

```
mf = 3.14; // set value

mf.LocalCopy(mf2, …); // Copy data from mf2 to mf. The two have the same layout.
mf.ParallelCopy(mf2, …); // Copy data w/ MPI. The two have different BoxArray
                         //   and DistributioMapping.
mf.FillBoundary(…); // Ghost cell exchange

auto mfmin = mf.min(…); // Return the minimum value

mf.minus(mf2, …); // mf = mf - mf2

// Create an alias MultiFab without deep-copying the data
// Suppose the original MultiFab stores density, x-velocity, y-velocity, and
// z-velocity. We need to call a function expecting x-velocity being the first
// component.
MultiFab mf_alias(mf, amrex::make_alias, xvel_comp, 3);

// We can also take raw pointers and create a MultiFab without deep-copying or
// taking the ownership. Some users use amrex this way in their existing codes.
```

FArrayBox, Array4 & MFIter

- A MultiFab contains multiple FArrayBoxes.
- FArrayBox: container for multi-dimensional (3+1) array. Movable but not copyable.
- Array4: 4D array view similar to C++23 std::mdspan, but with negative indexing support. Fortran array syntax.
- FArrayBox (usually) owns the memory, whereas Array4 never owns the memory making it trivially copyable.
- MFIter: Iterator for MultiFab. It supports logical tiling, which for CPU runs improve cache efficiency and OpenMP performance.

```
auto const lo = amrex::lbound(mfi.validbox());
auto const hi = amrex::ubound(mfi.validbox());
for (int k = lo.z; k <= hi.z; ++k) {
for (int j = lo.y; j <= hi.y; ++j) {
for (int i = lo.x; i <= hi.x; ++i) {
  a(i,j,k) = a2(i,j,k);
}}}
```

→

```
// mf & mf2 have the same layout
for (MFIter mfi(mfi); mfi.isValid();
     ++mfi) {
  FArrayBox& fab = mf[mfi];
  FArrayBox const& fab2 = mf2[mfi];
  fab.template
      copy<RunOn::Device>(fab2);
}

for (MFIter mfi(mfi); mfi.isValid();
     ++mfi) {
  Array4<Real> const& a =
                 mf.array(mfi);
  Array4<Real const> const& a2 =
                 mf.const_array(mfi);
  ParallelFor(mfi.validbox(),
              [=] AMREX_GPU_DEVICE
              (int i, int j, int k) {
      a(i,j,k) = a2(i,j,k);
  });
}
```

FArrayBox, Array4 & MFIter

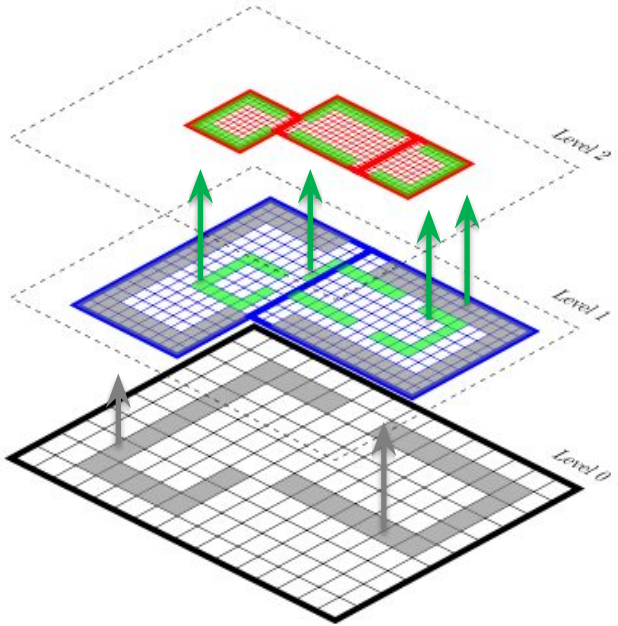
- A MultiFab contains multiple FArrayBoxes.
- FArrayBox: container for multi-dimensional (3+1) array. Movable but not copyable.
- Array4: 4D array view similar to C++23 std::mdspan, but with negative indexing support. Fortran array syntax.
- FArrayBox (usually) owns the memory, whereas Array4 never owns the memory making it trivially copyable.
- MFIter: Iterator for MultiFab. It supports logical tiling, which for CPU runs improve cache efficiency and OpenMP performance.

```
auto const& a = mf.arrays();
auto const& a2 = mf.const_arrays();
// a single kernel for multiple Boxes
ParallelFor(mf, [=] AMREX_GPU_DEVICE
            (int bno, int i, int j, int k)
{
  a[bno](i,j,k) = a2[bno](i,j,k);
});
```

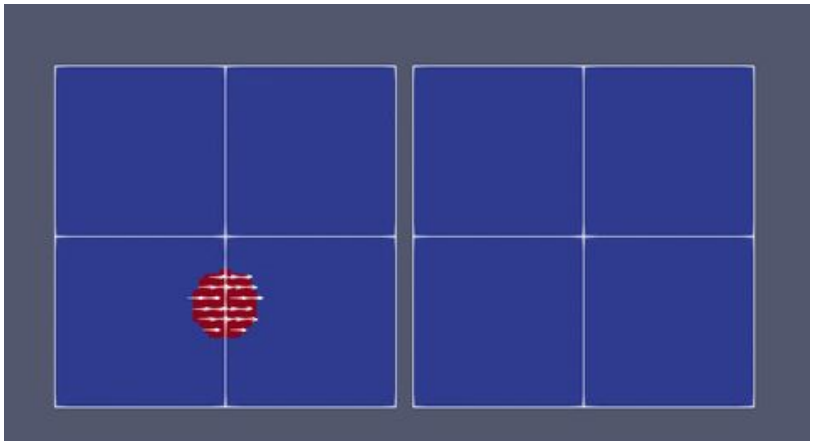
```
// mf & mf2 have the same layout
for (MFIter mfi(mfi); mfi.isValid();
     ++mfi) {
  FArrayBox& fab = mf[mfi];
  FArrayBox const& fab2 = mf2[mfi];
  fab.template
      copy<RunOn::Device>(fab2);
}

for (MFIter mfi(mfi); mfi.isValid();
     ++mfi) {
  Array4<Real> const& a =
                mf.array(mfi);
  Array4<Real const> const& a2 =
                mf.const_array(mfi);
  ParallelFor(mfi.validbox(),
              [=] AMREX_GPU_DEVICE
              (int i, int j, int k) {
      a(i,j,k) = a2(i,j,k);
  });
}
```

AMR Operations

- Classes: Amr/AmrLevel (more built-in functionalities), AmrCore (more flexibilities)
- Regridding: Tagging cells for refinement, grid generation, data filling, …
- Interpolation: Ghost cell filling
- Restriction: Average fine data down to coarse level
- Flux registers: Refluxing for face fluxes in conservation law systems and edge fluxes in MHD on Yee grids.
- Time stepping
  - Subcycling in time or non-subcycling
  - 2nd order Godunov method
  - Method of lines with high-order Runge-Kutta
  - Spectral deferred corrections
- Multi-level linear system solvers



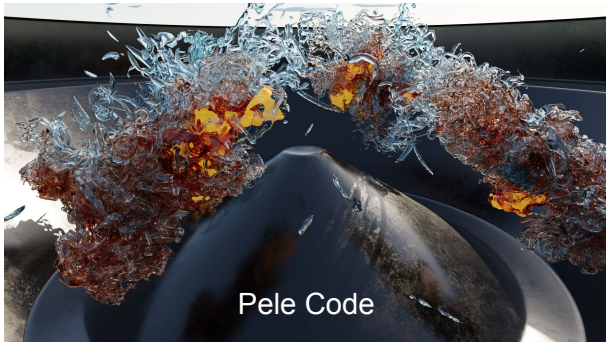
Flexible Communication Functionality

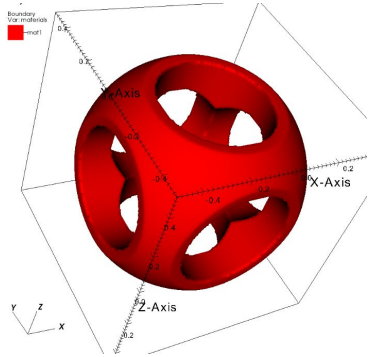


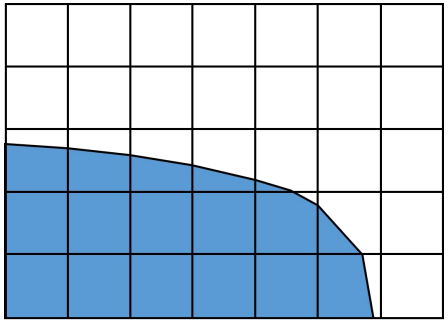
Multi-block advection test

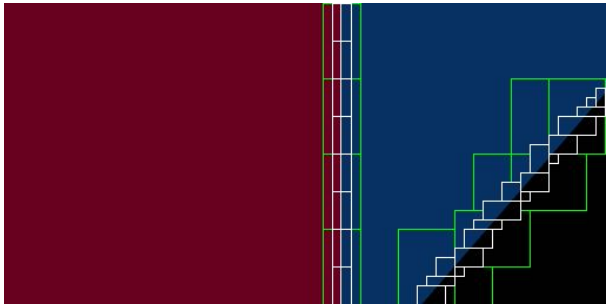
- Communication using user provided connection between blocks.
- Communication support for curvilinear coordinates. For example, filling the lo-x and lo-y boundary regions by rotating the data around the center by 90 degrees.
- MPMD (Multiple Program Multiple Data) support. For example, one C++ code built with amrex and one python code using pyAMReX.

Embedded Boundary







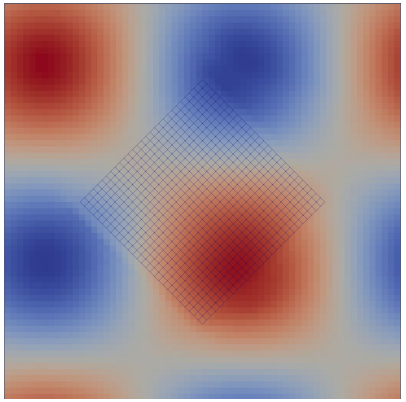


- Geometry can be generated by implicit functions or STL.
- Support for moving EB.
- An example of using CSG (constructive solid geometry).

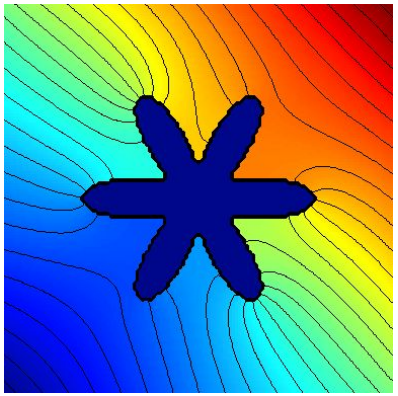
```
EB2::SphereIF sphere(0.5, {0.0,0.0,0.0}, false);
EB2::BoxIF cube({-0.4,-0.4,-0.4}, {0.4,0.4,0.4}, false);
auto cubesphere = EB2::makeIntersection(sphere, cube);

EB2::CylinderIF cylinder_x(0.25, 0, {0.0,0.0,0.0}, false);
EB2::CylinderIF cylinder_y(0.25, 1, {0.0,0.0,0.0}, false);
EB2::CylinderIF cylinder_z(0.25, 2, {0.0,0.0,0.0}, false);
auto three_cylinders = EB2::makeUnion(cylinder_x, cylinder_y, cylinder_z);

auto csg = EB2::makeDifference(cubesphere, three_cylinders);
```

Sparse Linear System Solvers

- Examples of linear operators
  - $\alpha A\phi - \beta\nabla(\vec{B}\cdot\nabla\phi) = \text{rhs}$
  - $\alpha A\vec{v} - \beta\nabla\cdot(\eta(\nabla\vec{v} + \nabla\vec{v}^T) + (\kappa - (2/3)/\eta)\nabla\vec{v}\mathbf{I}) = \text{rhs}$
  - $\nabla\times(\alpha\nabla\times\vec{E}) + \beta\vec{E} = \text{rhs}$
- EB aware
- Overset support
- Geometric Multigrid
- CG or BiCGSTAB as “bottom” solver of multigrid
- GMRES with multigrid as preconditioner.
- Interface to call hypre or PETSc.



Overset mesh coupling



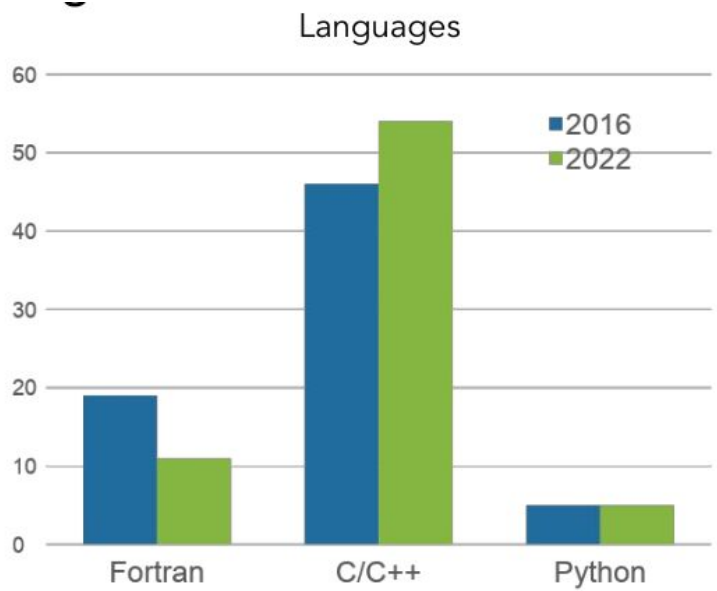
Embedded boundary

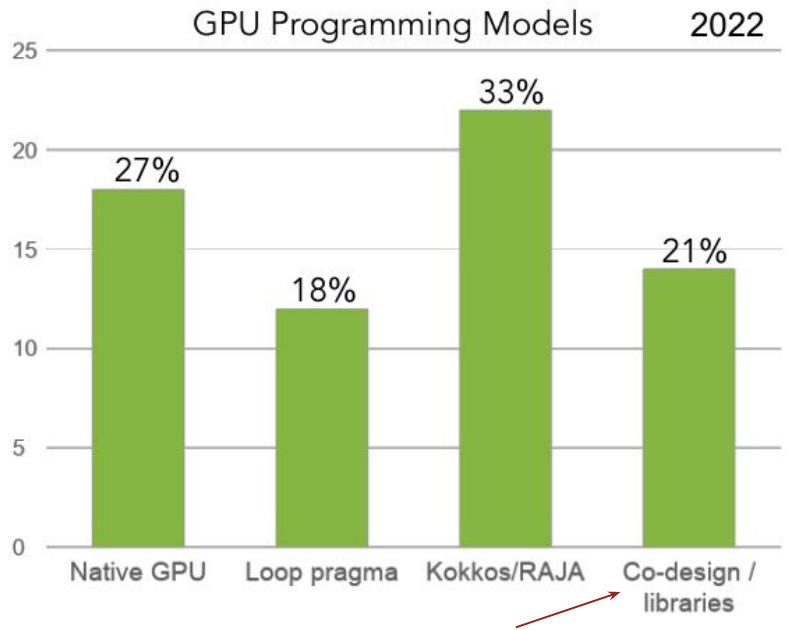
Performance Portability

- Own portability layer based on vendors' C++ programming models: CUDA for NVIDIA, HIP for AMD, and SYCL/oneAPI for Intel. A decision made in early 2018 after exploring a number of options including CUDA Fortran, OpenACC, OpenMP, Kokkos and RAJA.

- Advantages of our own performance portability layer based on vendors' C++ programming models.
  - Flexibility. We were able to run on Intel GPUs two months after the first beta release of oneAPI.
  - Maturity and Stability. Vendors' C++ solutions are arguably much more mature and stable.
  - Performance. Close to metal. Specific performance tuning for AMReX applications.
  - Functionality. Easier to implement new features that require low level functionality.

ECP Programming Languages & Models





10 ECP codes use AMReX.

Evans, et. al., "A survey of software implementations used by application codes in the Exascale Computing Project", 2022, IJHPC, 36, 1

1D ParallelFor CUDA Implementation

```
// cpu code
for (int i = 0; i < n; ++i) { a[i] = b[i]; }

// cuda
__global__ void assign(int n, double*a, double const* b)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  If (i < n) { a[i] = b[i]; }
}

assign<<<(n+255)/256,256>>>(n,a,b);

// amrex
ParallelFor(n, [=] AMREX_GPU_DEVICE (int i) {
  a[i] = b[i];
});
```

```
// amrex detail
#define AMREX_GPU_DEVICE __device__

template <typename F>
__global__ void launch_global (F f) {f();}

template <typename F>
ParallelFor(int n, F&& f)
{
  launch_global<<<(n+256)/256,256>>>(
    [=] AMREX_GPU_DEVICE () {
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      f(i); // this lambda calls the user's lambda
  });
}
```

- C++ lambda function
- [=]: capture by value
  (In this case, pointers, not the data pointed by the pointer.
- AMREX_GPU_DECICE: __device__

3D ParallelFor

```
// FArrayBox fab_a, fab_b
Array4<Real> const& a = fab_a.array(); // or auto const&
auto const& b = fab_b.const_array();
double s = …;
ParallelFor(fab_a.box(),
            [=] AMREX_GPU_DEVICE (int i, int j, int k) {
  a(i,j,k) = s * b(i,j,k);
});
```

```
// Internally,
int tid = blockDim.x * blockIdx.x + threadIdx.x;
auto len = amrex::length(box);
auto lo = amrex::lbound(box);
if (tid < len.x*len.y*len.z) {
  int k = tid/(len.x*len.y);
  int j = (tid-k*(len.x*len.y))/len.x;
  int i = (tid-k*(len.x*len.y)) - j*len.x;
  i += lo.x;
  j += lo.y;
  k += lo.z;
}
```

- FArrayBox cannot be used directly on GPU.
- Array4 is a trivial type working on GPU.
- Separation of ownership and access.
- Box as iteration range
- C++ (extended device) lambda function captures what we need by value. For example, a, b, and s in this example.
- Long int used internally to avoid integer overflow.
- Fast algorithm of integer division.
- STREAM benchmarks using 3D ParallelFor can achieve > 80% of the peak memory bandwidth.

Reduction on GPU

AMReX Provides various reduction functions. Here are two examples.

```
GpuTuple<double, int> r = ParReduce(TypeList<ReduceSum,ReduceMax>{},   // types of
reduction
                                    TypeList<double,int>{},            // data types
                                    box,                               // 3D iteration
space
    [=] AMREX_GPU_DEVICE (int i, int j, int k) -> GpuTuple<double,int> {
        double x = ..;
        int m = ..;
        return {x,m};
    });
// Perform two types of reductions in one GPU kernel. The input data for reduction are
// generated on the fly.
        using VL = amrex::ValLocPair<double,size_t>;
        Gpu::DeviceVector<double> v(...);
        auto const* p = v.data();
        VL r = Reduce::Max<VL>(v.size(), [=] AMREX_GPU_DEVICE (size_t i) -> VL {
                return {p[i], i};
            });
        // Reduction of a custom type.
        // r.value is the max value in the vector.
        // r.index is the index of the max value.
```

Scan (Prefix Sum, Partial Sum) on GPU

```
PrefixSum<T>(N, [=] AMREX_GPU_DEVICE (int i) -> T {
                    return .. },  // input
                [=] AMREX_GPU_DEVICE (int i, T const& x)
{
                     // x is the sum of inputs from
elements
                     // 0 to i-1.
                });
```

- Both input and output are lambda functions, not iterator based. Lambda is arguably much more powerful and convenient than iterators.
- The input can be computed on the fly. It can be simply return p[i], or more complicated.
- The output can be used without writing to the global device memory.
- Used extensively in AMReX particle operations. For example, we use it to implement partition functions.

Kernel Fusion

```
auto a = mfa.arrays();
auto b = mfb.const_arrays();
auto c = mfc.const_arrays();
ParallelFor(mfa, [=] AMREX_GPU_DEVICE (int boxno, int i, int j, int k) {
    a[boxno](i,j,k) = b[boxno](i,j,k) + c[boxno](i,j,k);
});
// A single GPU kernel working on multiple boxes. Box sizes are not necessarily the same.
// Memory is not contiguous across boxes.
// Fused kernel is often 2x - 10x faster depending on the box sizes.
```

```
// Hundreds of small kernels, very slow
for (int m = 0; m < n; ++m) {
  ParallelFor(boxes[m], [=] AMREX_GPU_DEVICE (int i, int j, int k) { … });
}

// Kernel fusion
amrex::Vector<Tag> tags; // Tag can be a user defined type or amrex predefined type
for (int m = 0; m < n; ++m) {
   tags.emplace_back(Tag{boxs[m], …});// Store information needed for GPU kernel
}
// Launch a single GPU kernel, much faster
ParallelFor(tags, [=] AMREX_GPU_DEVICE (int i, int j, int k, Tag const& tag) { … });
```

Optimization of Kernels with Run Time Parameters

Branches in kernels can be very expensive not just because of thread divergence. If a branch uses a lot of registers, it can significantly affect the performance even if at run time the branch is never executed. One could move the branches out of kernels. But that sometimes results in a lot of code duplication. We provide compile time optimization by using C++17 fold expression to generate codes for all run time variants.

```
int A_runtime_option = ...;
int B_runtime_option = ...;
enum A_options : int { A0, A1, A2, A3};
enum B_options : int { B0, B1 };
// 4*2=8 ParallelFors will be generated.
ParallelFor(TypeList<CompileTimeOptions<A0,A1,A2,A3>,
                     CompileTimeOptions<B0,B1> > {},
            {A_runtime_option, B_runtime_option},
             N, [=] AMREX_GPU_DEVICE (int i, auto A_control, auto
B_control)
{
    ...
    if constexpr (A_control.value == A3 && B_control.value == B1) {
        ...
    } else if constexpr (....) {
        ...
    }
    ...
});
```

WarpX PushPX kernel on MI250X

- QED module not always used.
- With the optimization
  - Time: 2.17 -> 1.34
  - NumVgprs: 256 -> 249
  - ScratchSize: 264 -> 144
  - Occupancy: 1 -> 2

Memory Arena

- Memory allocation using cudaMalloc etc. can be quite expensive.
- AMReX provides a number of memory arenas to speed up memory allocation. By default, we preallocate ¾ of the system's global device memory in one big chunk. Subsequent memory allocations from that arena are much faster.
- Vector resize. May not need to move the data even if its capacity is not big enough.
- The use of Arena allows us to implement a memory safety feature.

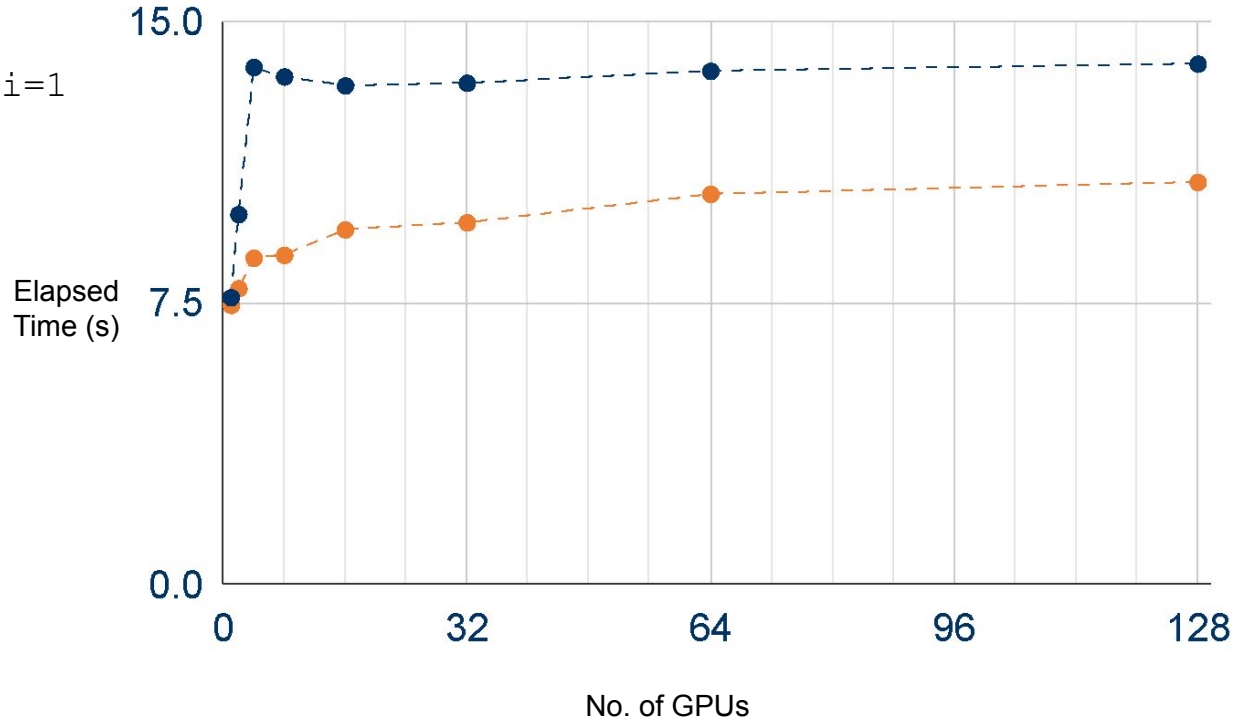
```
{
    FArrayBox tmp(...);
    async_gpu_kerel_using_tmp(tmp);
    Gpu::synchronize();  // Must sync!
    // otherwise there is a compiler inserted
    // destructor that will free the memory in
tmp
    // before the async gpu kernel finishes.
}
```

```
{
    FArrayBox tmp(...,The_Async_Arena());
    async_gpu_kerel_using_tmp(tmp);
} // async safe
```

- The_Arena(): The default arena. Either managed or device.
- The_Async_Arena(): Async safe
- The_Device_Arena(): A separate device arena if The_Arena() is managed, otherwise alias to The_Arena().
- The_Managed_Arena(): A separate managed arena or alias to The_Arena().
- The_Pinned_Arena(): Pinned host memory.
- The_Cpu_Arena(): Pageable host memory.

GPU-Aware MPI

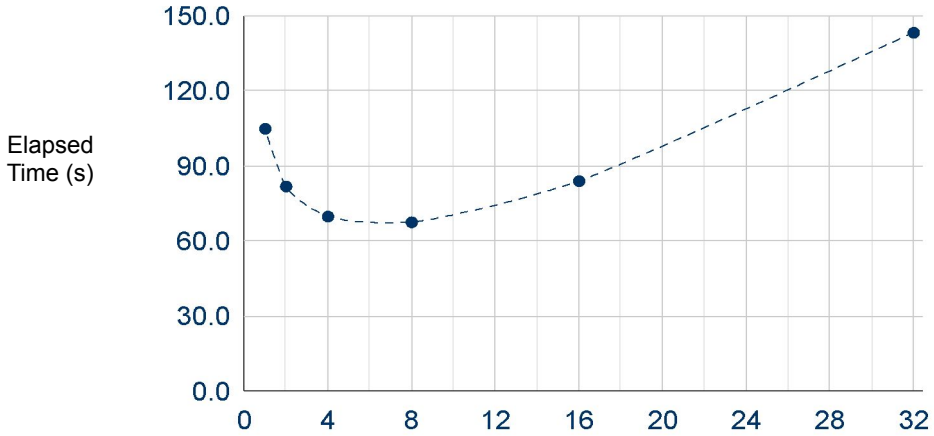
- Runtime parameter:
  `amrex.gpu_aware_mpi=1`
- System specfic setups



OpenMP on CPUs

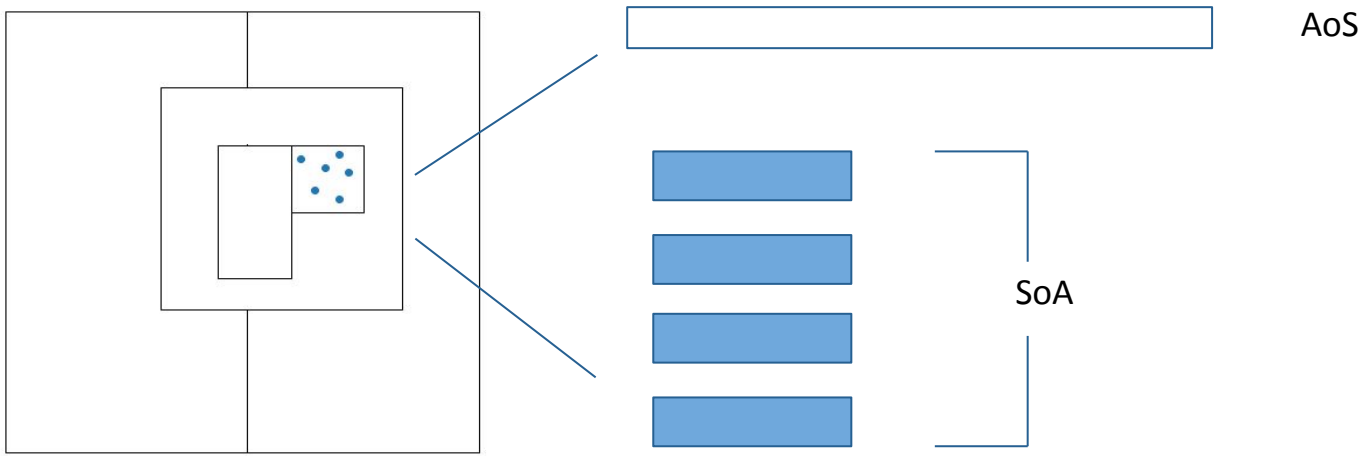
Atmospheric boundary layer simulations using the Energy Research and Forecasting (ERF) code

- 100 time steps, I/O and diagnostic calculations are turned off.
- Tested over 2 nodes (256 physical cores) on NERSC's Perlmutter system. The problem size and number of cores are kept constant while varying the balance of MPI processes and OMP threads per process.

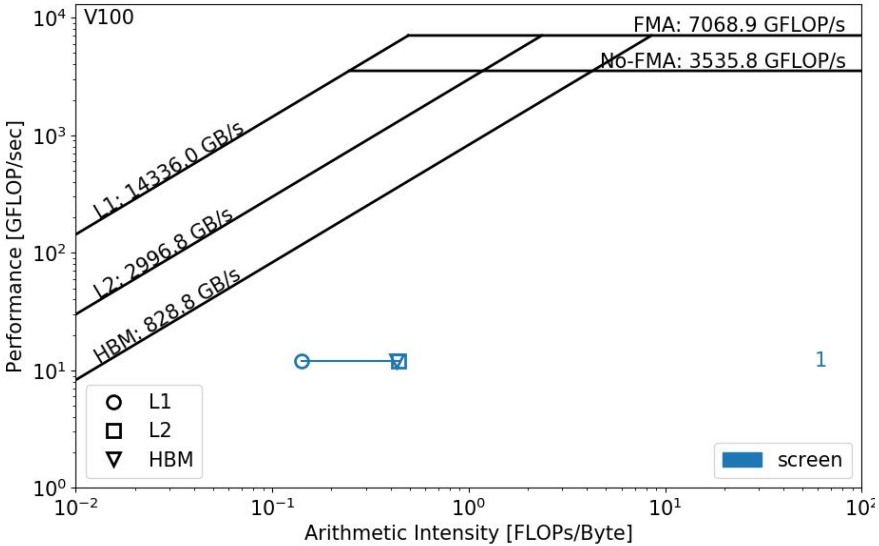


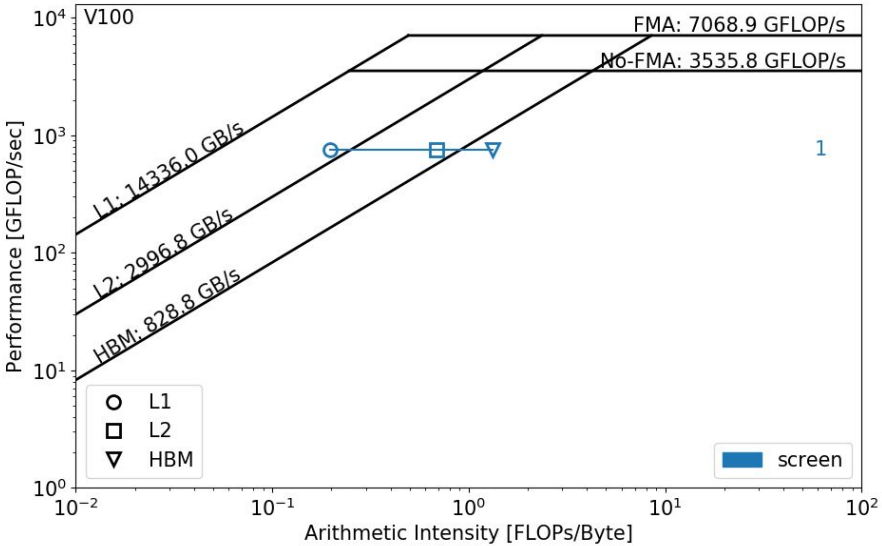
Particle: Flexible Data Layout

Particles on each rank are split up by levels and grid
On a single grid, data is stored as a user-defined mix of Array-of-Structs (AoS) and Struct-of-Arrays (SoA)



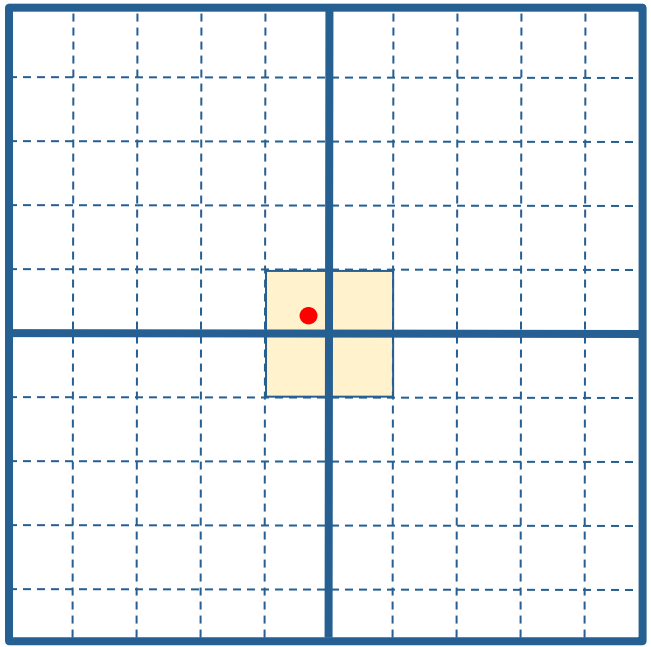
AoS vs. SoA: MFiX-Exa





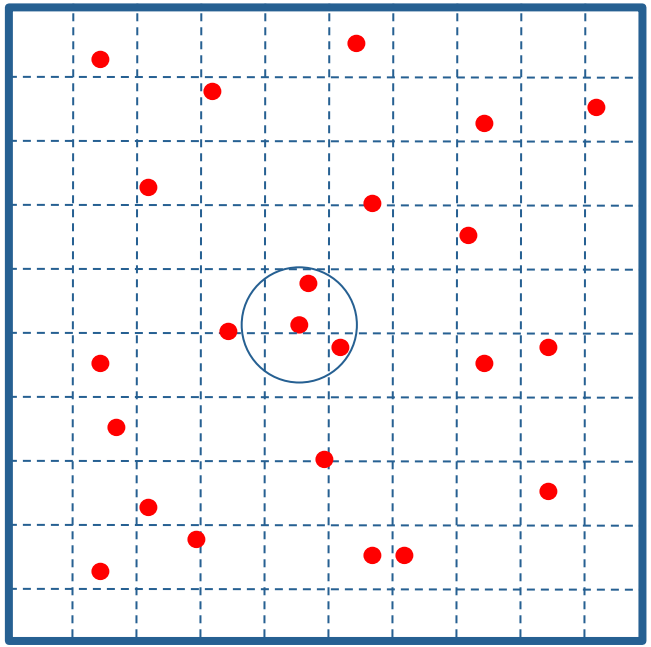
Particle-Mesh Interaction

- AMReX provides a set of tools for handling particle-mesh interaction
- Simple interpolation kernels (NGP, CIC) for mesh-to-particle and particle-to-mesh interpolation are provided
- Lambda function interface for performing user-provided operations interpolations (e.g. high-order particle shapes, Gaussian kernels)
- Performs needed parallel communication “under-the-hood” (SumBoundary)
- Support for dual-grid
- Uses data duplication strategy for CPU threading, per-particle atomics for GPU threading.



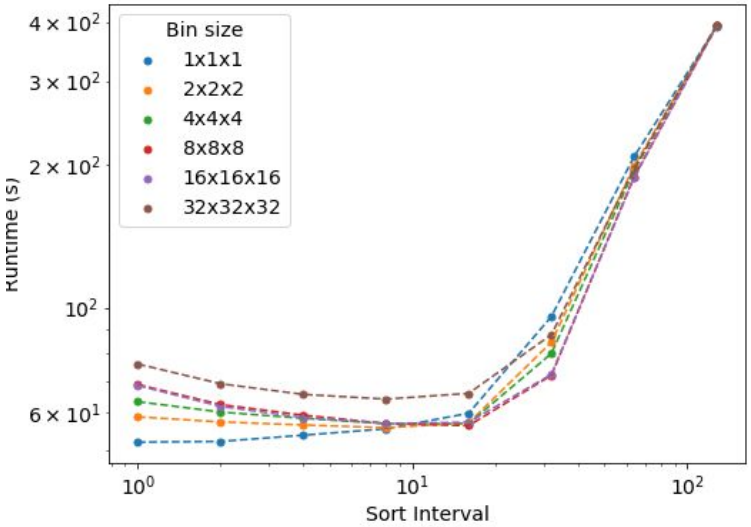
Particle-Particle Interaction

- AMReX includes tools for building and iterating over neighbor lists for MD-style particle-particle interactions
- Pre-compute a list of potential interaction partners that can be reused for multiple time steps
- Uses a cell list to avoid direct N^2 search of all possible pairs
- Users can specify operation using lambda function.
- Support for half and full neighbor lists (tradeoff based on amount of contention / atomics performance)

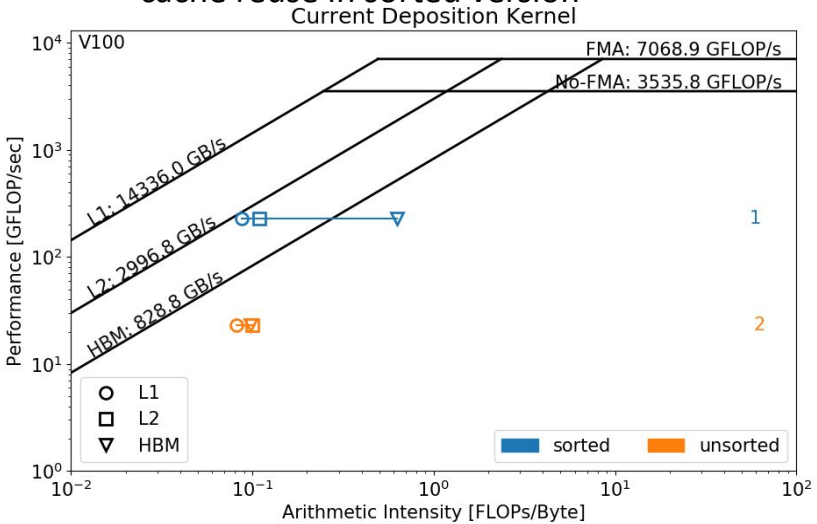


Sorting Particles

8x speed up between not sorting and optimal sorting



Roofline analysis confirms better L2 cache reuse in sorted version



Open Source

- Number of contributors on GitHub:
- Your Contributions Are Welcome!
- GitHub Issues & Discussions
- AMReX Slack (Email me WeiqunZhang@lbl.gov)
- Thank you!