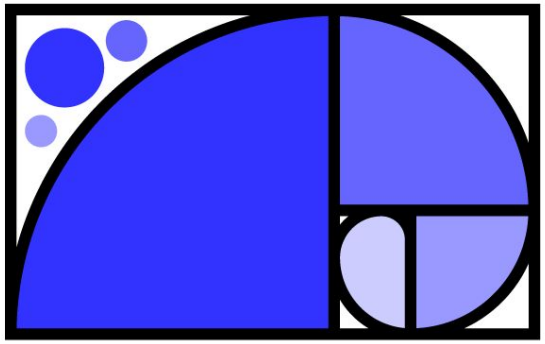
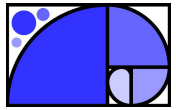


AMReX

AMReX: A Performance Portable Software Framework for Adaptive Mesh Refinement Applications

Weiqun Zhang (LBNL) March, 2026

Overview of AMReX



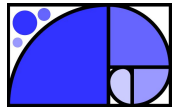
Software framework for building massively parallel, block-structured adaptive mesh refinement (AMR) applications. Coded in modern C++17 (with Fortran + Python interfaces). Runs on machines from laptops to supercomputers.

- 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…
- Parallel FFT
- Performance portability layer with support for multiple GPU backends - **CUDA**, **HIP**, **SYCL** - and **OpenMP** for CPUs
- **xSDK** compliant, part of **E4S**, installable with **spack** and **conda**

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

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

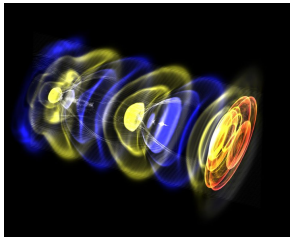
History of AMReX

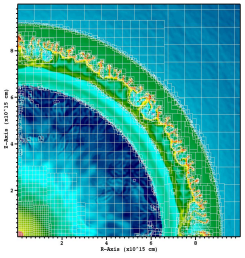


- Started as a co-design center in the US Department of Energy's Exascale Computing Project in 2016.
- Six other ECP projects use AMReX.
- Based on BoxLib.
- C++11 + Fortran -> C++14 -> C++17
- Pure CPU code -> performance portable
- More features & more flexibilities.
- More contributions and feedbacks from the users.
- Fully open source
- A project of Linux Foundation

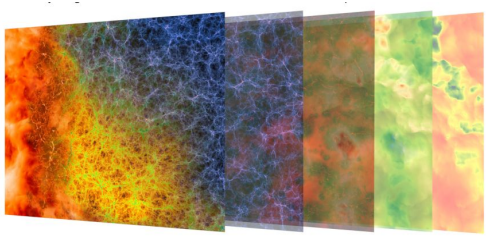
A Sampling of AMReX Application Codes

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

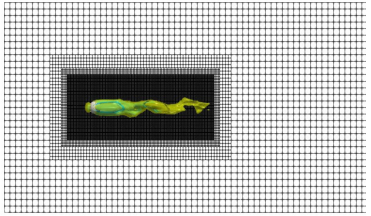




Cosmology:
Nyx



Combustion:
PeleC (Compressible)
PeleLM (Low Mach)



Accelerator Modelling:
WarpX
ImpactX
Hipace++

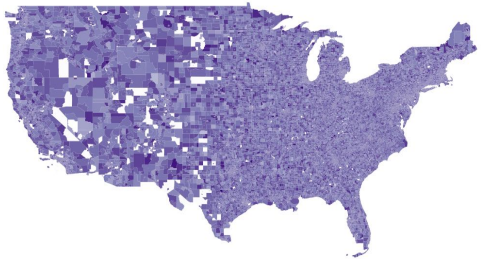
Magnetically-confined fusion:
GEMPIC

Ocean Modeling:
REMORA

Multiscale Modelling and Stochastic Systems:
FHDeX

Electromagnetics:
ARTEMIS

Epidemiology:
ExaEpi

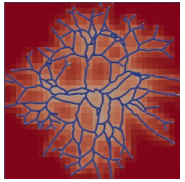


Incompressible Navier-Stokes:
IAMR
incflo

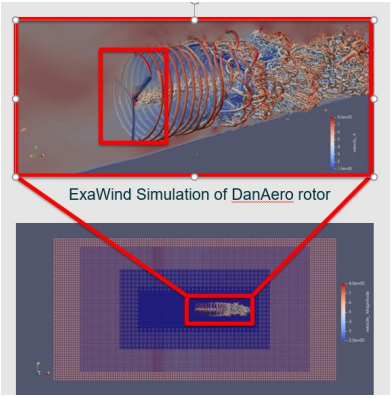
Atmospheric science:
AMR-wind
ERF

Solid Mechanics:
Alamo

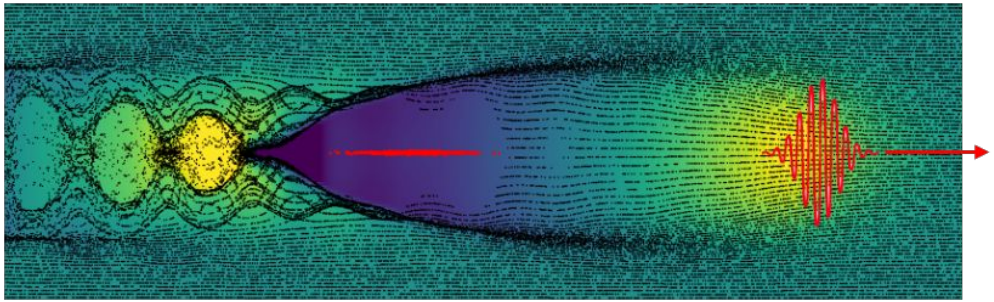
Biological cell modelling:
BoltzmanMFX
CCM

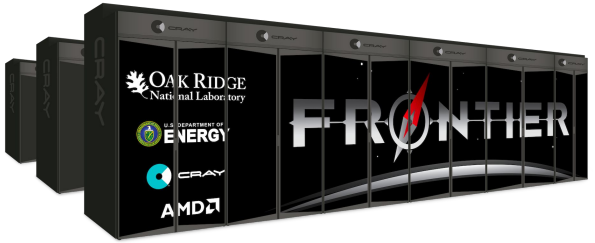


Multi-phase Flow:
MFIX-Exa



WarpX: 500x FOM, Gordon Bell Award





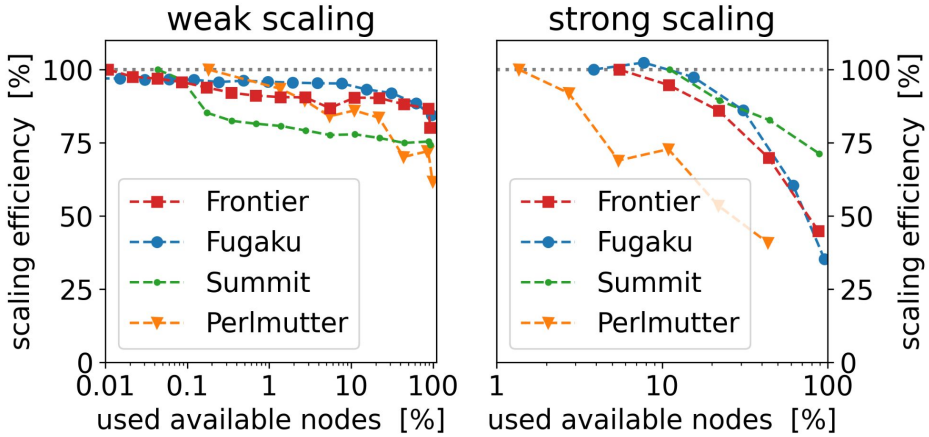
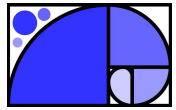


Figure-of-Merit over time

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

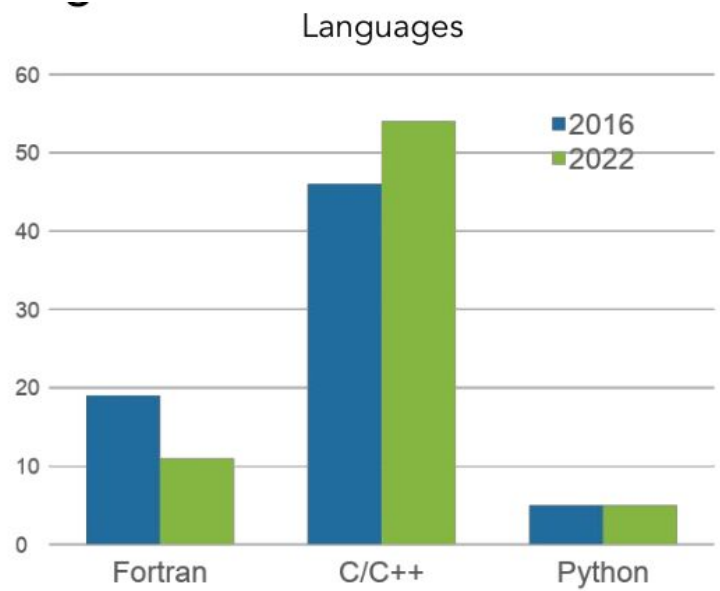
Performance Portability

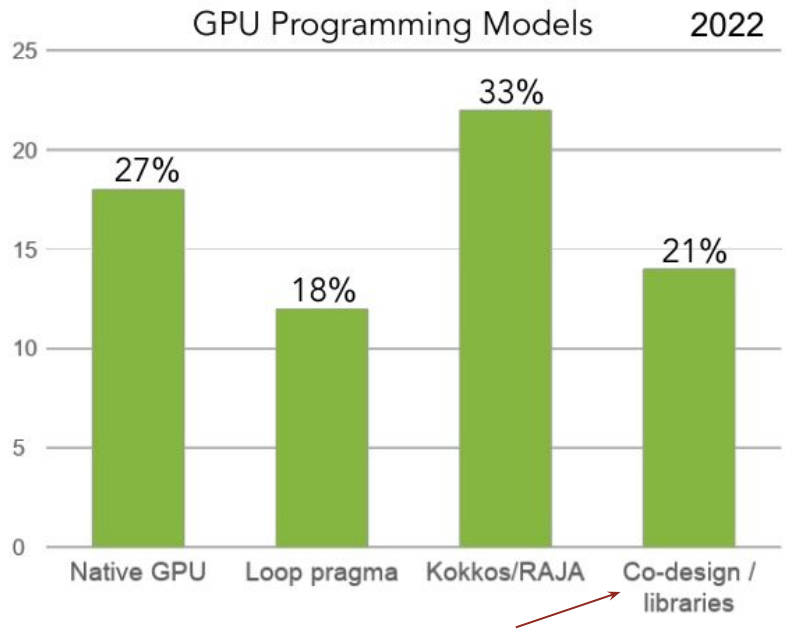


- Own portability layer based on vendors' C++ programming models: CUDA for NVIDIA, HIP for AMD, and SYCL/oneAPI for Intel. A decision made in early 2018 after exploring a number of options including CUDA Fortran, OpenACC, OpenMP, Kokkos and RAJA.

- 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

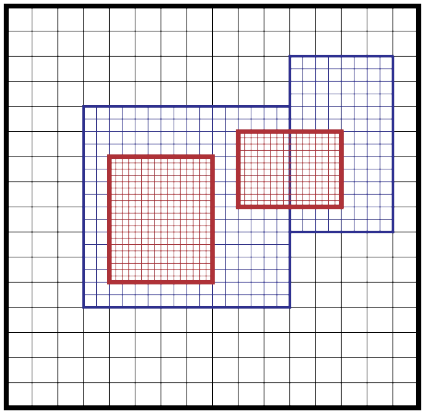


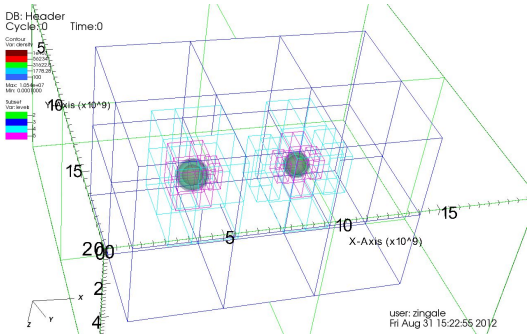


10 ECP codes use AMReX.

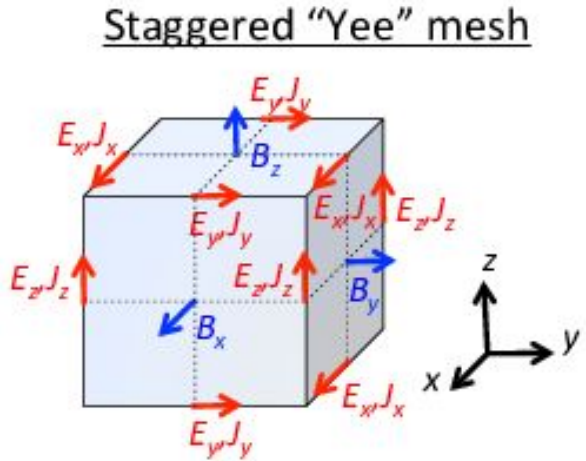
Evans, et. al., "A survey of software implementations used by application codes in the Exascale Computing Project", 2022, IJHPC, 36, 1

Box, IntVect, & IndexType

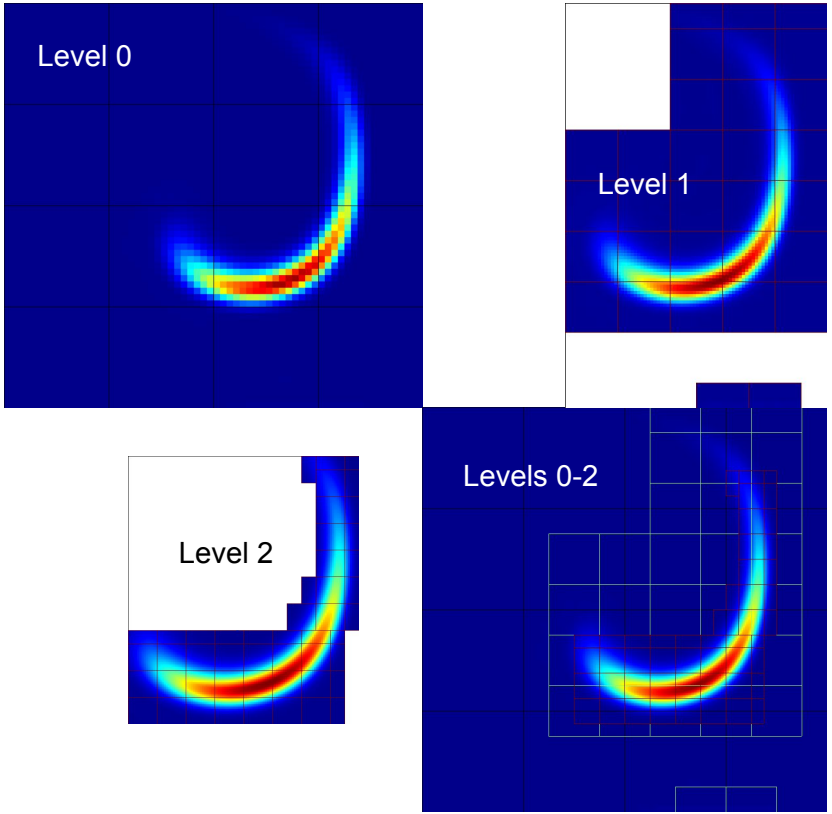




- Box represents a (logically) rectangular domain in global integer indexing space.
- IntVect is an integer tuple.
- IndexType: cell or node. For example, Bx on x-face: (node, cell, cell)



MultiFab



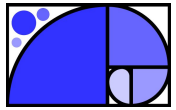
- MultiFab is a distributed container for data on a single level.
- Multiple Fab (Fortran Array Box). Fortran multi-D array convention (i.e., column major).
- Can have multiple components. Multiple 4-D arrays.
- Can have ghost cells.
- What memory to use?

```
// ba: BoxArray, array of boxes
// dm: DistributionMapping, MPI domain
//                          decomposition
int ncomp = 3;
IntVect nghost(1,2,0);

// Use GPU memory by default
MultiFab mf(ba, dm, ncomp, nghost);

MultiFab hmf(ba, dm, 1, 0, // Use CPU
memory
   MFInfo().SetArena(The_Cpu_Arena()));
```

FArrayBox & Array4



- 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, thus suitable for GPU.

```
do k = loz, hiz
  do j = loy, hiy
    do i = lox, hix
      a(i,j,k) = (b(i+1,j,k) - b(i-1,j,k))/2;
    enddo
  enddo
enddo
```



```
// Given FArrayBox fab_b;
Box box(IntVect(lox,loy,loz),
        IntVect(hix,hiy,hiz));
FArrayBox fab_a(box); // Fortran like array
auto a = fab_a.array();
auto b = fab_b.array();
// a & b are lightweight objects for GPU
ParallelFor(box, [=] AMREX_GPU_DEVICE
                     (int i, int j, int k)
{
  a(i,j,k) = (b(i+1,j,k) - b(i-1,j,k))/2;
});
// The code above works for both CPU and
GPU.
```

Loop body almost identical

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];
});
```

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

```
// amrex detail for Nvidia & AMD GPU
// Intel GPU implementation is different.
#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;
      if (i < n)
        f(i); // this lambda calls the user's
lambda
  });
}

// We hide the detail above from the users. And
ParallelFor is portable on CPUs and GPUs.
```

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 (that also work with OpenMP on CPU).

```
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. No atomics needed.
        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.

Lessons Learned

- Performance portability layer
- Interactions with application teams
  - AMReX developers are also members of various application teams.
  - Application driven development.
  - Features implemented for one application are often useful for other applications.
- Interactions with vendors
  - Regular meetings, feature requests, bug reports, etc.
- Interactions with online users
  - New features implemented by users
  - New improvements in code and documentation

Open Source

- Number of GitHub contributors: 193 (was 177 one year ago)
- You contributions are welcome!
- GitHub Issues and Discussions
- AMReX Slack (> 400 people): To join, email me WeiqunZhang@lbl.gov
- Second Users' Meeting as Part of HPSF Meeting: Chicago, March 19, 2026

  https://events.linuxfoundation.org/hpsf-conference/