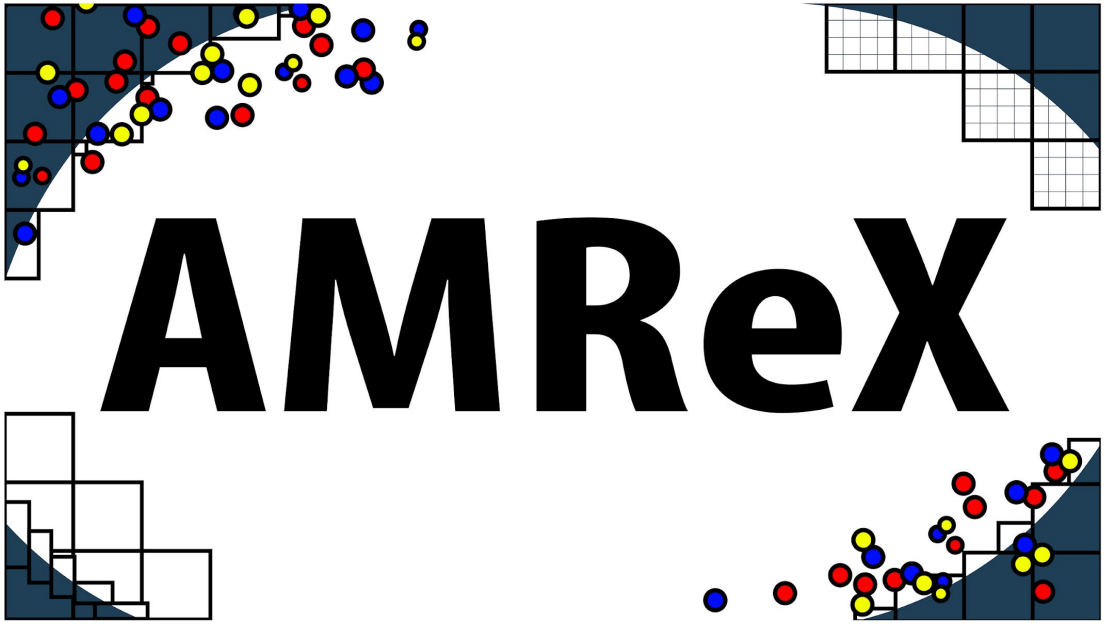


AMReX & GRTeclyn

Weiqun Zhang, June 26, 2024

U.S. DEPARTMENT OF ENERGY | Office of Science

Applied Mathematics & Computational Research

BERKELEY LAB

History of AMReX

- Started as a co-design center in the US Department of Enengy's Exascale Computing Project in 2016.
- Six other ECP projects use AMReX.
- Based on BoxLib.
- Chombo was also 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.

GRChombo -> GRTeclyn

- CH_SPACEDIM -> AMReX_SPACEDIM
- CH_assert -> AMREX_ASSERT, AMREX_ASSERT_WITH_MESSAGE, AMREX_ALWAYS_ASSERT, …
- CH_TIME -> BL_PROFILE
- MayDay::Error -> amrex::Abort
- MayDay::Warning -> amrex::Warning
- pout -> amrex::Print, AllPrint, PrintToFile, AllPrintToFile
- problemDomain -> Geometry (not exactly)
- Intervals -> two ints
- LevelData, BoxLayoutData -> MultiFab, iMultiFab, FabArrayBox
- BoxLayout, DisjointBoxLayout -> BoxArray, DistributionMapping
- procID() -> amrex::ParallelDescriptor::MyProc()

AMReX's Kernel Launch Approach

- MultiFab: Data on a single level. Multiple FArrayBox. Each FArrayBox is a 4D array.
- How to iterate?
- Tiling.
- How to launch kernels? ParallelFor & LoopOnCpu, lambda function
- How does lambda function work (for CPU and GPU)?
- Be aware of asynchronicity.
- Kernel Fusion.

MFIter: For-loop over boxes

```
MultiFab mf(...);
for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    FArrayBox& fab = mf[mfi];
    Box const& box = mfi.validbox();
    // for illustration only
    auto const lo = amrex::lbound(box);
    auto const hi = amrex::ubound(box);
    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) {
        fab(IntVect(i,j,k)) = 3.14;
    }}}
}
```

Array4: std::mdspan like

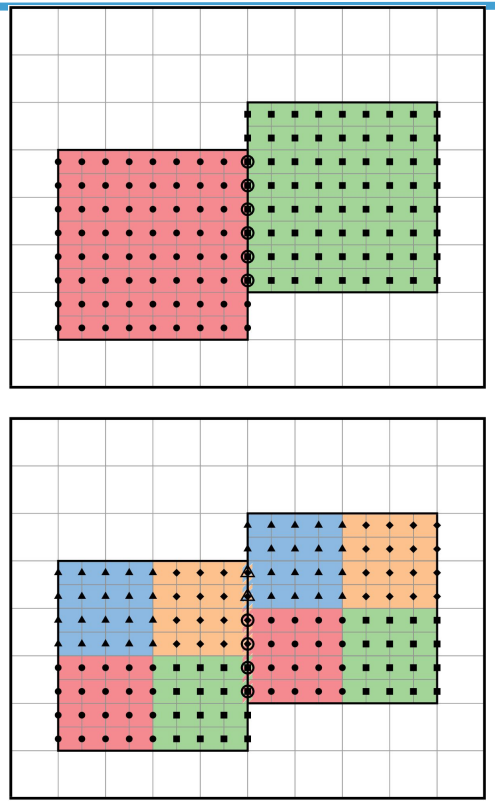
```
MultiFab mf(...);
for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    FArrayBox& fab = mf[mfi];
    Box const& box = mfi.validbox();
    // for illustration only
    auto const lo = amrex::lbound(box);
    auto const hi = amrex::ubound(box);
    for (int k = lo.z; k <= hi.z; ++k) {
    for (int j = lo.y; j <= hi.y; ++j) {
    for (int i = lo.x; i <= hi.x; ++i) {
        fab(IntVect(i,j,k)) = 3.14;
    }}}
}
```

```
MultiFab mf(...);
for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    Array4<Real> const& a = mf.array(mfi);
    Box const& box = mfi.validbox();
    // for illustration only
    auto const lo = amrex::lbound(box);
    auto const hi = amrex::ubound(box);
    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) = 3.14;
    }}}
}
```

Array4 introduced because of GPU programming.

Tiling & OpenMP

```
MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU()); mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    // for illustration only
    auto const lo = amrex::lbound(box);
    auto const hi = amrex::ubound(box);
    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) = 3.14;
    }}}
}
```



CPU Kernel Launch with lambda

```
MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU());
     mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    // for illustration only
    auto const lo = amrex::lbound(box);
    auto const hi = amrex::ubound(box);
    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) = 3.14;
    }}}
}
```

```
MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU());
     mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    LoopOnCpu(box,[&] (int i, int j, int k)
    {
        a(i,j,k) = 3.14;
    });
}


// In AMReX header
template <typename F>
void LoopOnCpu(Box const& box, F&& f)
{
    for (int k = …)
        for (int j = …)
            for (int i = …)
                f(i,j,k);
}
```

How does lambda work for CPU

```
MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU());
     mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    LoopOnCpu(box,[&] (int i, int j, int k)
    {
        a(i,j,k) = 3.14;
    });
}
```

```
// Compiler generates something like
struct lambda {
    Array4<Real> const& a;
    lambda (Array4<Real> const& a_a) : a(a_a) {}
    void operator(int i, int j, int k) {
        a(i,j,k) = 3.14;
    }
};

MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU());
     mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    LoopOnCpu(box,lambda(a));
}
```

GPU Kernel launch with lambda

```
MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU());
     mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    LoopOnCpu(box,[&] (int i, int j, int k)
    {
        a(i,j,k) = 3.14;
    });
}
```

```
MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU());
     mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    ParallelFor(box,
    [=] AMREX_GPU_DEVICE (int i, int j, int k)
    {
        a(i,j,k) = 3.14;
    });
}

// works for CPU build too
```

How does lambda for GPU

```
MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU());
     mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    ParallelFor(box,
    [=] AMREX_GPU_DEVICE (int i, int j, int k)
    {
        a(i,j,k) = 3.14;
    });
}
```

- Array4 allows for capture by value.
- Compiler will copy lambda from host to device.
- Reference does not work because it's a host pointer.

```
// Compiler generates something like
struct lambda {
    Array4<Real> a; // store value
    lambda(Array4<Real> const& a_a) : a(a_a) {}
    __device__ void operator() (int…) {}
};

MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU());
     mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    ParallelFor(box,lambda(a));
}

// works for CPU build too
```

Asynchronicity

```
for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    FArrayBox tmp_fab(...);
    ParallelFor(...); // async
    // Compiler inserts ~FArrayBox
    // tmp_fab gets deleted before GPU finishes
}
```

```
for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    FArrayBox tmp_fab(..., The_Async_Arena());
    ParallelFor(...); // async
    // Compiler inserts ~FArrayBox
    // Memory not released until GPU finishes.
}
```

```
// Alternatively, we can do this.
// But the Async_Arena approach is better.
for (MFIter mfi(mf); mfi.isValid(); ++mfi)
{
    FArrayBox tmp_fab(...);
    ParallelFor(...); // async
    Gpu::streamSynchronize(); // explicit sync
}
```

Kernel Fusion

```
MultiFab mf(...);
#pragma omp parallel if (Gpu::notInLaunchRegion())
for (MFIter mfi(mf,TilingIfNotGPU());
     mfi.isValid(); ++mfi)
{
    auto const& a = mf.array(mfi);
    Box const& box = mfi.tilebox();
    ParallelFor(box,
    [=] AMREX_GPU_DEVICE (int i, int j, int k)
    {
        a(i,j,k) = 3.14;
    });
}
```

```
MultiFab mf(...);
auto const& ma = mf.arrays();
Parallel(mf, [=] AMREX_GPU_DEVICE
             (int box_no, int i, int j, int k)
{
    a[box_no](i,j,k) = 3.14
};

// A single kernel
//
// Slightly worse performance if Boxes are big.
// Order of magnitude faster, if Boxes are small.
```