AMReX on GPUs -- Overview

• Written in a mix of C++14/Fortran (plus an option for using Fortran interfaces)

• Supports parallelism through MPI+X (with UPC++ option)

• Mesh, particle and particle/mesh algorithms

• Solution of parabolic and elliptic systems using geometric multigrid solvers

• Embedded boundary (cut-cell) representation of geometry
AMReX GPU Porting Goals

There are two aspects of “GPU-izing AMReX”:

• GPU-ize the core functionality of AMReX (which the application code never needs to modify or even see)

• Make it as easy for the application teams as possible to manage memory and call kernels -- without limiting potential performance.

Initial approach was to focus on NVIDIA GPUs, aka to explicitly use CUDA for performance.

This strategy is allowing us to relatively easily extend to AMD GPUs.
GPU Porting Strategy Overview

AMReX Porting Topics:
- Fortran to C++
- Finer-Grained Parallelism
- Lambda-based GPU launches
- CUDA streams implementation
- Memory management strategies
- Asynchronous Temporaries
- Particle methods

Laser Driven Plasma Accelerator Test using WarpX on 1 node of Summitdev. Currently achieving 3.5x speed-up on 1 Volta V100 vs. 1 KNL node.
Converting AMReX work to C++

AMReX kernels have been converted from Fortran to C++. 
Applications can still use Fortran (and CUDA Fortran), but AMReX’s floating point operations are entirely in C++.

• Better **Maintainability**  
  – Fortran requires separate Host & Device versions of all functions.

• Ensure future **portability**  
  – Industry trend: C++ is given attention first.

• **Remove dependency** on CUDA Fortran.  
  – Less dependencies create a more stable platform.
Conversion Example

**Fortran**

```
do k = lo(3), hi(3)
  do j = lo(2), hi(2)
    do i = lo(1), hi(1)+1
      fluxx(i,j,k) += 1.0
    end do
  end do
end do
```

**C++**

```
Array4<Real> const& fab = mf.array(mfi);
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(i,j,k) += 1.;
    }
  }
}
```

Array4<Real> holds:

- Pointer to the data of type **Real**
- Size of the object in 3D
- Striding information

and is fully accessible on the GPU.

**Testing:**

- C++ Faster: 18 (Common Functions)
- Fortran Faster: 10 (Compiler Specific)
- Similar Timings: 179
Memory Management

NVIDIA Unified Virtual Memory is used for FAB data:

- Easy partial piece-wise porting and development.
- Minimal overhead if data is moved to device and kept there.
- Not strictly required. Preparation for future architectures.

Allocation performed through structured AMReX Arenas.

- Allow switching of memory types (Host, Device, Managed, Pinned, etc).
- Numerous memory pool strategies available:
  - (Buddy-memory, Slab, custom AMReX allocate-on-the-fly).

Isolate metadata throughout AMReX data structures to reduce data movement.
CPU Parallelism Strategy

“MPI over grids, OpenMP over logical tiles”

Done using MultiFAB iterators called \texttt{MFIter}:

- Handles proper looping \textit{over local grids}.
- Stores relationship of grids across MPI ranks.
- Coordinates \textit{tiling}.

Typical usage is for \textit{OpenMP to loop over all the tiles} (potentially from multiple FABs) on a single MPI rank. Also includes:

- Static vs. dynamic scheduling
- Synchronous vs. asynchronous
  (overlapping communication and computation)
GPU Parallelism is more fine-grained

- OpenMP threads were on the order of tiles across local boxes (~10-100).
- GPU threads are on the order of local number of cells (~thousands).
- GPU Parallelization strategy is shifted to a finer-grained implementation over cells.

CPU thread distribution strategy using tiling with OpenMP.

GPU thread distribution strategy using CUDA threads.

\[ l_{\text{tile}}(:) = h_{\text{tile}}(:) \]

\[ l_{\text{gpu}}( :) = h_{\text{gpu}}(:) \]
Portablility using Lambdas and Streams

- Allows **compile time selection** of CPU or GPU.
- MFIter increments across **CUDA streams** to maximize parallelism for any tiling.

```cpp
amrex::ParallelFor(box,
                   [=] AMREX_GPU_DEVICE (int i, int j, int k)
                   {
                     fab(i,j,k) += 1.;
                   });
```

AMReX launch variations:

- **amrex::launch** – Launch work over a given Box or number of elements.
- **amrex::ParallelFor** – Tightly nested 1D, 3D or 4D loops.
- Fused launches – Sets of differently sized work combined into a single launch.
Example CPU Parallelism

```cpp
phi_old.FillBoundary(geom.periodicity());
#pragma omp parallel
const Real* dx = geom.CellSize();
const Box& domain_bx = geom.Domain();
for ( MFIter mfi(phi_old); mfi.isValid(); ++mfi )
{
    const Box& bx = mfi.validbox();
    compute_flux(BL_TO_FORTRAN_BOX(bx),
                 BL_TO_FORTRAN_BOX(domain_bx),
                 BL_TO_FORTRAN_ANYD(phi_old[mfi]),
                 BL_TO_FORTRAN_ANYD(flux[0][mfi]),
                 BL_TO_FORTRAN_ANYD(flux[1][mfi]),
#if (AMREX_SPACEDIM == 3)
                 BL_TO_FORTRAN_ANYD(flux[2][mfi]),
#endif
                 dx);
}
#pragma omp parallel end
```

Standard MPI Work: Fill Boundary, Redistribute, etc.

Loop over Boxes with local data. OpenMP details hidden inside.

Get Box to work over.

Call Fortran function that does work on Box data.
phi_old.FillBoundary(geom.periodicity());
const Real* dx = geom.CellSize();
const Box& domain_bx = geom.Domain();
for ( MFIter mfi(phi_old); mfi.isValid(); ++mfi )
{
    const Box& xbx = mfi.nodaltilebox(0);
    Array4<Real> const& fluxx = flux[0].array(mfi);
    Array4<Real> const& phi = phi_old.array(mfi);

    amrex::ParallelFor(xbx,
        [=] AMREX_GPU_DEVICE (int i, int j, int k)
        {
            compute_flux_x(i, j, k, fluxx, phi, dxinv);
        });

    // Additional launch for Y and Z fluxes.
}

Standard MPI Work:
Launches used to implement.

Loop over Boxes with local data.
CUDA streams incremented within.

Get Box and pointers to MultiFab data to work over.

Launch lambda function to perform over desired Box.

Device sync during MFIter destructor to guarantee data consistency.
Asynchronous Temporaries

Temporaries that use **CUDA callbacks in the destructors** to allow MFIter loops to remain asynchronous.

- Implemented in FAB, Vector and Scalar data structures.
- Contain pointers to the host and device versions for consistent memory.

```cpp
if (d_data != nullptr) {
    T** p = static_cast<T**>(std::malloc(2*sizeof(T*)));
    p[0] = d_data;
    p[1] = h_data;
    AMREX_GPU_SAFE_CALL(cudaStreamAddCallback(Device::cudaStream(),
                                              amrex_asyncarray_delete, p, 0));
}
```

~AsyncArray():
Particle Specific Methods

Uses CUDA’s Thrust library to perform particle and particle/mesh work on the GPU:

- **Vector-like containers** using specialized allocators to implement on CUDA memory spaces.
- Also compatible with **Thrust vectors**.
- Implements **Thrust functions** for sorting, searching, and partitioning.
- **Redistributes** particles **entirely on the GPU** to minimize data movement.

Laser Driven Plasma Accelerator Test using WarpX on 1 node of Summitdev. Currently achieving 3.5x speed-up on 1 Volta V100 vs. 1 KNL node.
Next Steps for Portability

CudaGraphs
Testing useful implementation patterns for AMReX kernels, such as communication
• Especially useful for e.g., ghost cell filling

Asynchronous I/O

• Write with CPU, calculate with GPU.
• Template for future asynchronous work strategies.

![Diagram showing asynchronous I/O process]

- CPU
  - I/O
  - Copy to CPU
- GPU
  - Perform N Steps
  - Copy to CPU
  - Perform N Steps
  - Copy to CPU
  - Perform N Steps
Current “Full” Application Results

For comparison: Sod shock-tube tutorial with AMR has a $30\times$ speed up.
From NVIDIA → AMD GPUs

- For AMD GPUs, AMReX will use HIP as the backend without changing the application interface.

- HIP is very similar to CUDA in terms of the capabilities that AMReX uses. Most of the GPU kernels in AMReX require no changes at all, and we expect this to be true for application codes.

- For data iterators, multiple streams are used. AMReX provides portable macros and functions for kernel launch. The implementation of the kernel launch uses C++ templates and lambda functions, since both HIP and CUDA are C++ dialects.

- AMReX will continue using the Thrust library because it has been ported to HIP by AMD.
The HIP backend in AMReX is currently under development on NVIDIA GPUs. HIP code can run on NVIDIA GPUs with CUDA and NVCC as its backend. This allows us to mix HIP and CUDA during development. Once the HIP port is complete, the development will be moved onto AMD GPUs.

Currently HIP does not support managed memory, which is a feature AMReX uses to allow application codes a smooth transition to GPU. Because of its current limitation, we will provide functions to help manage the data migration between host and device.